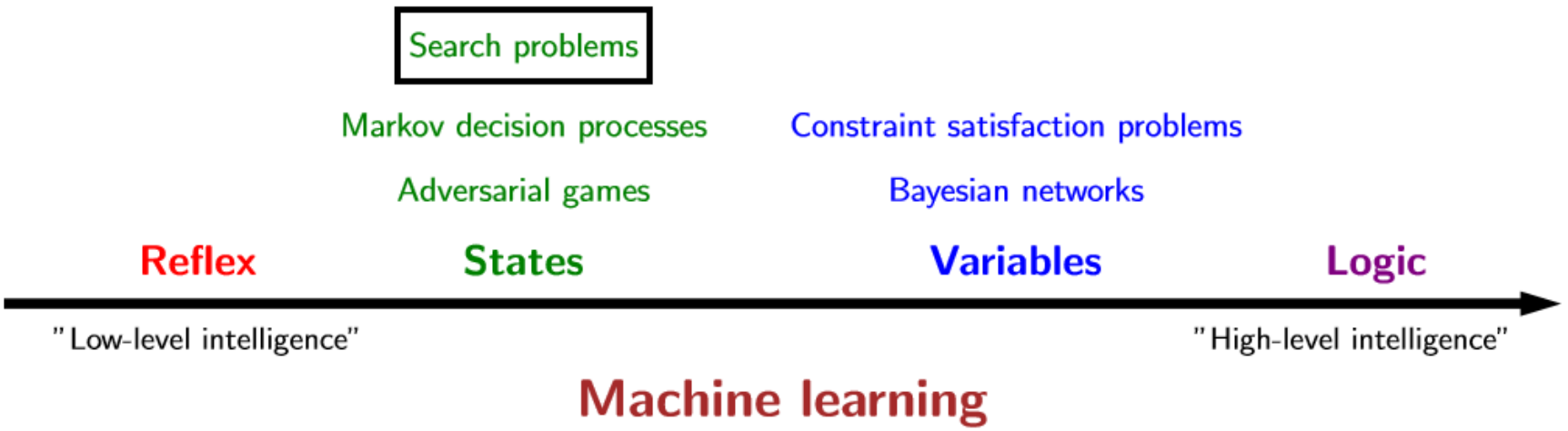
Uninformed/Blind Search

Artificial Intelligence

Slides are mostly adapted from AIMA, MIT Open Courseware and Svetlana Lazebnik (UIUC)


Search problems
Markov decision processes
Constraint satisfaction problems
Adversarial games
Bayesian networks
Reflex
States
Variables
Logic
"Low-level intelligence"
"High-level intelligence"
Machine learning

agent
entity that perceives its environment and acts upon that environment

Types of agents

Reflex agent

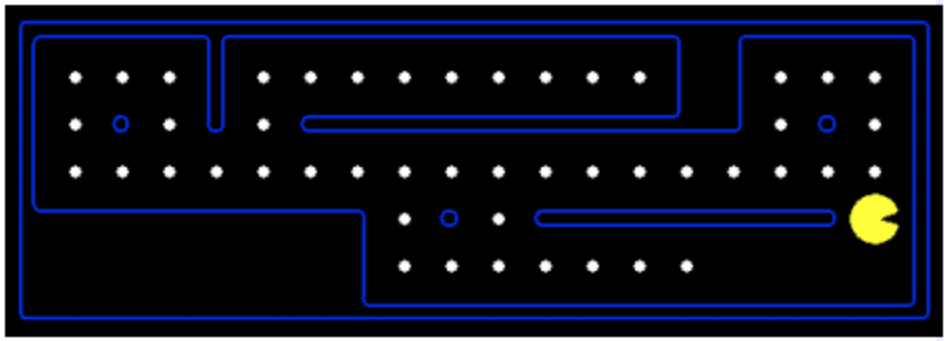
- Consider how the world IS
- Choose action based on current percept
- Do not consider the future consequences of actions

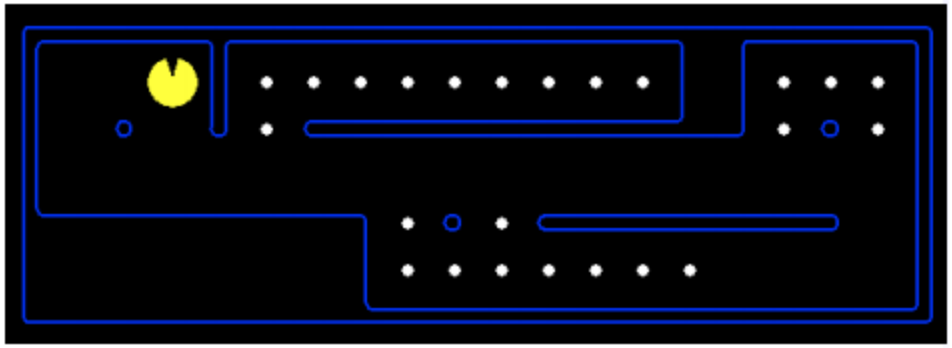
Planning agent

- Consider how the world WOULD BE
- Decisions based on (hypothesized) consequences of actions
- Must have a model of how the world evolves in response to actions
- Must formulate a goal

Reflex Agents

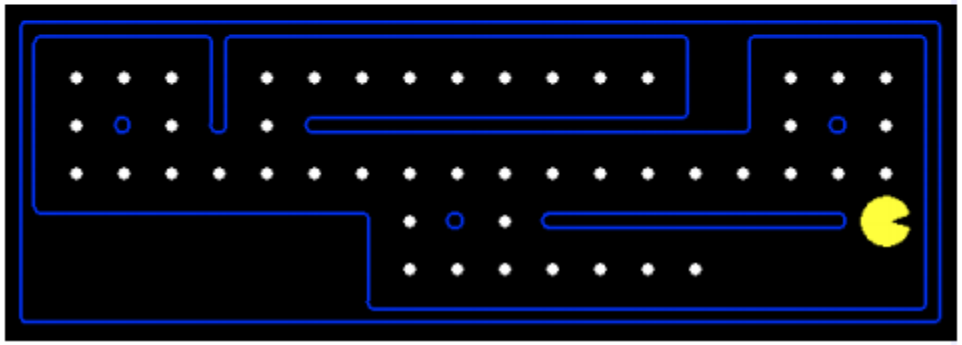
- Reflex Agents:
  - Choose action based on current percept (and maybe memory)
  - May have memory or a model of the world's current state
  - Do not consider the future consequences of their actions
  - Act on how the world IS
- Can a reflex agent be rational?

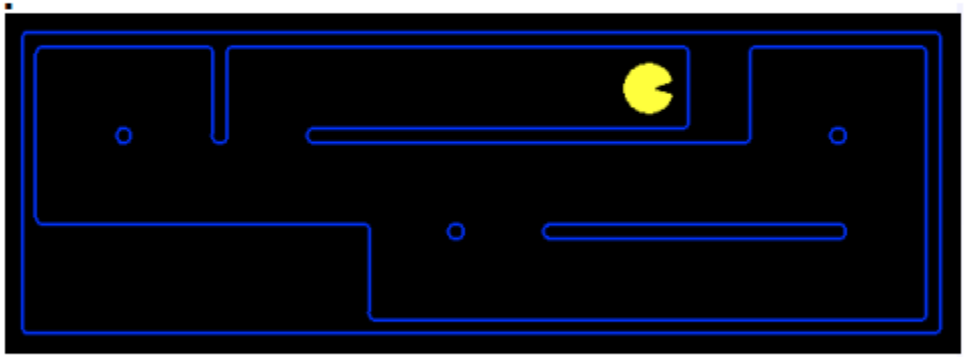




Goal Based Agents

- Goal-based agents:
  - Plan ahead
  - Decisions based on (hypothesized) consequences of actions
  - Must have a model of how the world evolves in response to actions
  - Act on how the world WOULD BE





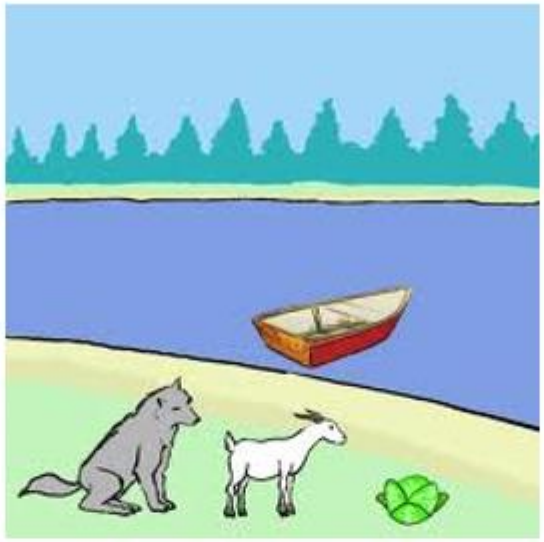
Simple-reflex vs goal based agents

- Simple-reflex agents directly maps states to actions.
- Therefore, they cannot operate well in environments where the mapping is too large to store or takes too much to learn
- Goal-based agents can succeed by considering future actions and desirability of their outcomes
- Problem solving agent is a goal-based agent that decides what to do by finding sequences of actions that lead to desirable states

A farmer wants to get his cabbage, goat, and wolf across a river. He has a boat that only holds two. He cannot leave the cabbage and goat alone or the goat and wolf alone.
How many river crossings does he need

When you solve this problem, try to think about how you did it. You probably simulated the scenario in your head, trying to send the farmer over with the goat, observing the consequences. If nothing got eaten, you might continue with the next action. Otherwise, you undo that move and try something else.

How can we get a machine to do this automatically? One of the things we need is a systematic approach that considers all the possibilities. We will see that search problems define the possibilities, and search algorithms explore these possibilities.

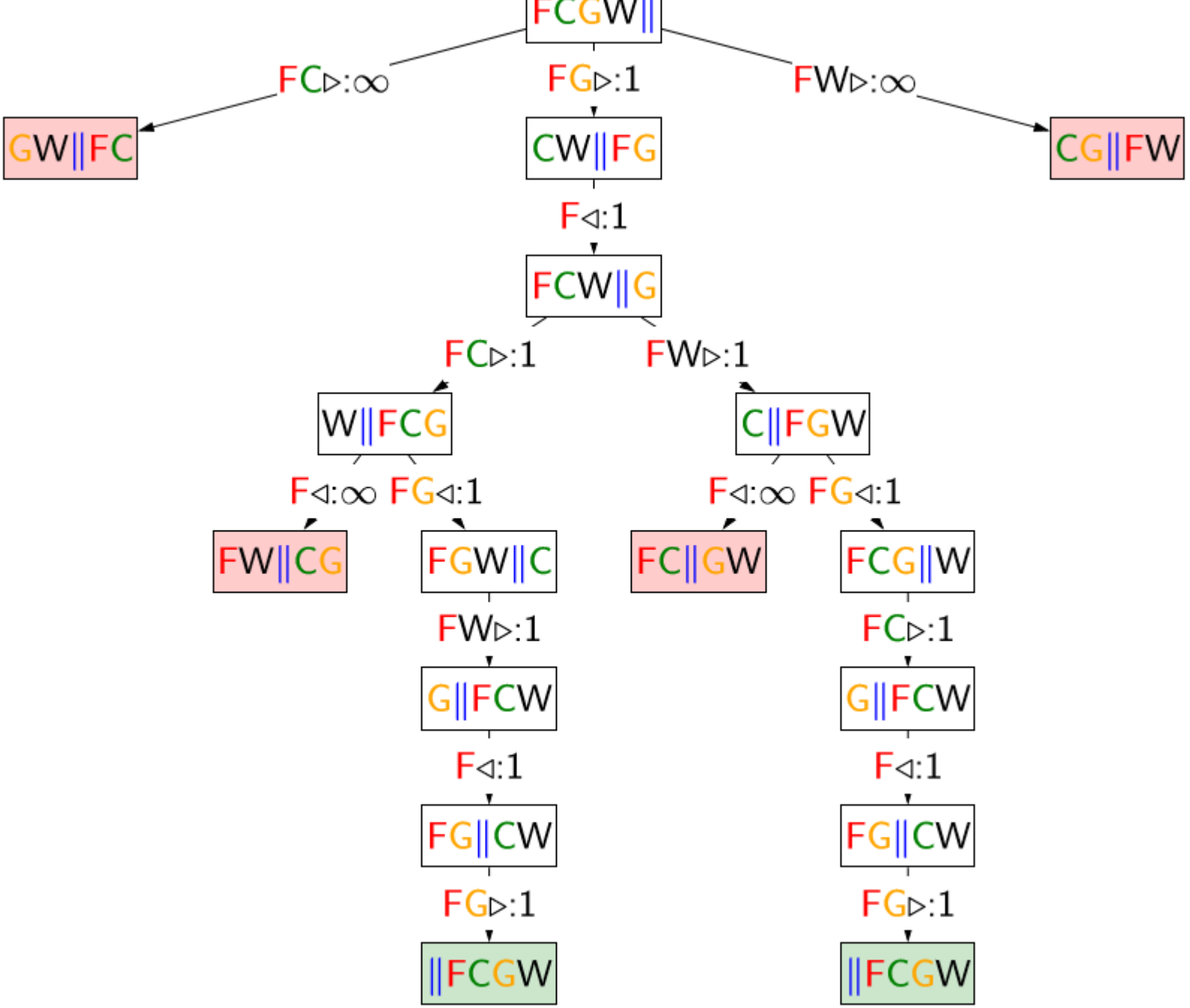


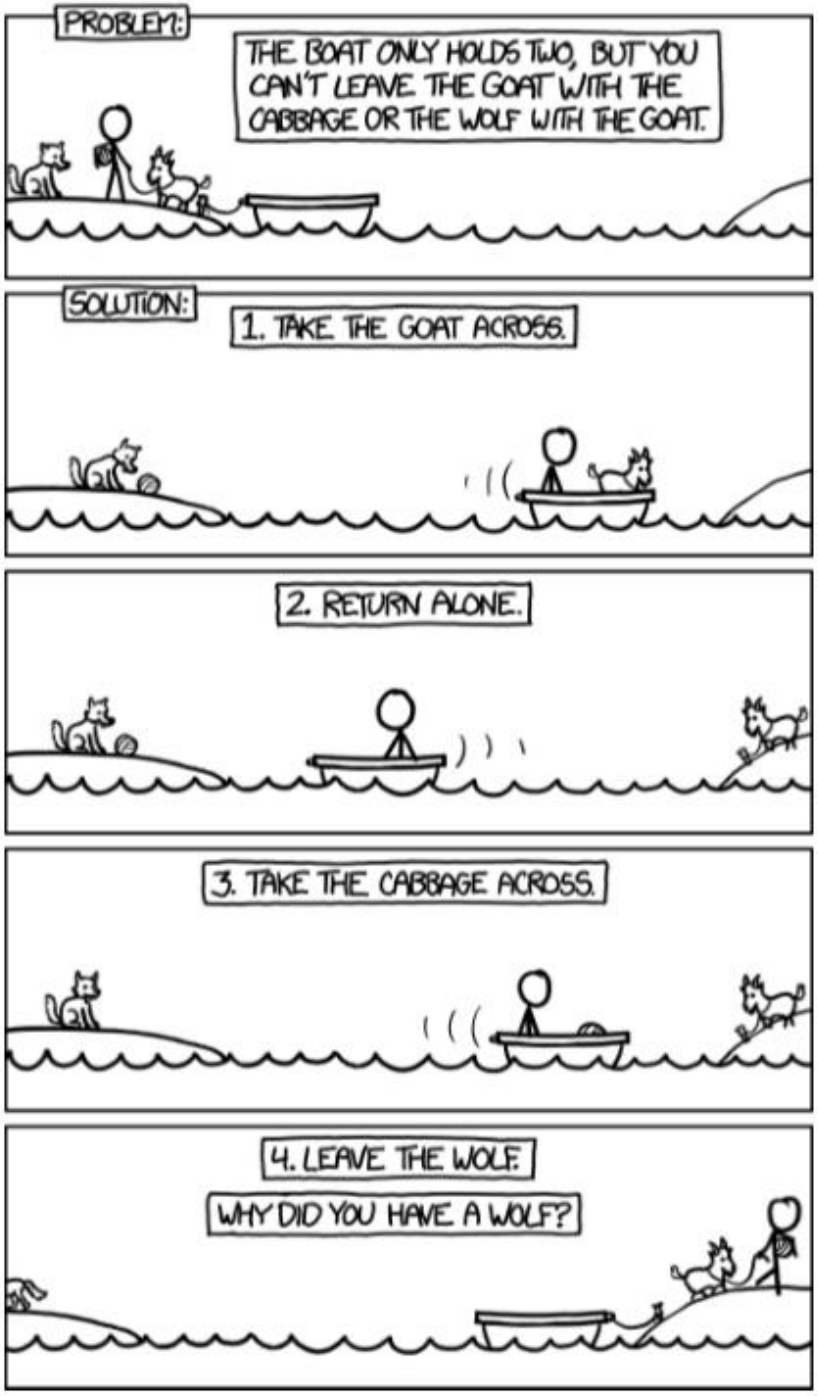
Farmer Cabbage Goat Wolf

Actions:

F▷	F◁
FC▷	FC◁
FG▷	FG◁
FW▷	FW◁

Approach: build a **search tree** ("what if?")


FCGW||
FC▷:∞
FG▷:1
FW▷:∞
GW||FC
CW||FG
CG||FW
F◁:1
FCW||G
FC▷:1
FW▷:1
W||FCG
C||FGW
F◁:∞
FG◁:1
F◁:∞
FG◁:1
FW||CG
FGW||C
FC||GW
FCG||W
FW▷:1
FC▷:1
G||FCW
G||FCW
F◁:1
F◁:1
FG||CW
FG||CW
FG▷:1
FG▷:1
||FCGW
||FCGW



Sometimes you can do better if you change the model (perhaps the value of having a wolf is zero) instead of focusing on the algorithm.

Problem solving agents

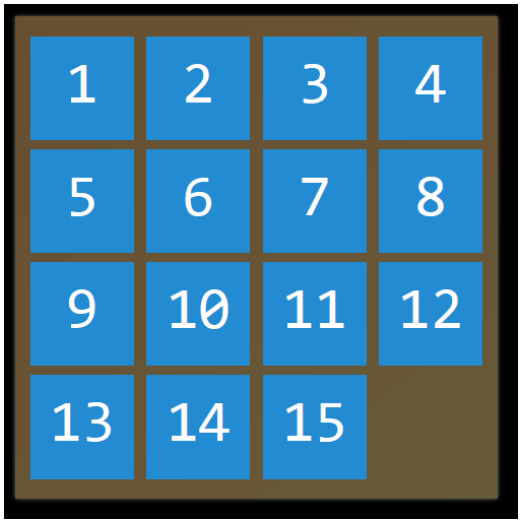
- Intelligent agents are supposed to maximize their performance measure
- This can be simplified if the agent can adopt a goal and aim at satisfying it
- Goals help organize behaviour by limiting the objectives that the agent is trying to achieve
- Goal formulation, based on the current situation and the agent's performance measure, is the first step in problem solving
- Goal is a set of states. The agent's task is to find out which sequence of actions will get it to a goal state
- Problem formulation is the process of deciding what sorts of actions and states to consider, given a goal

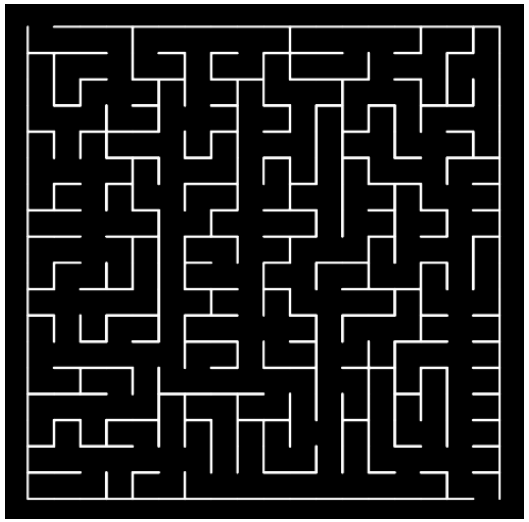
Problem solving agents

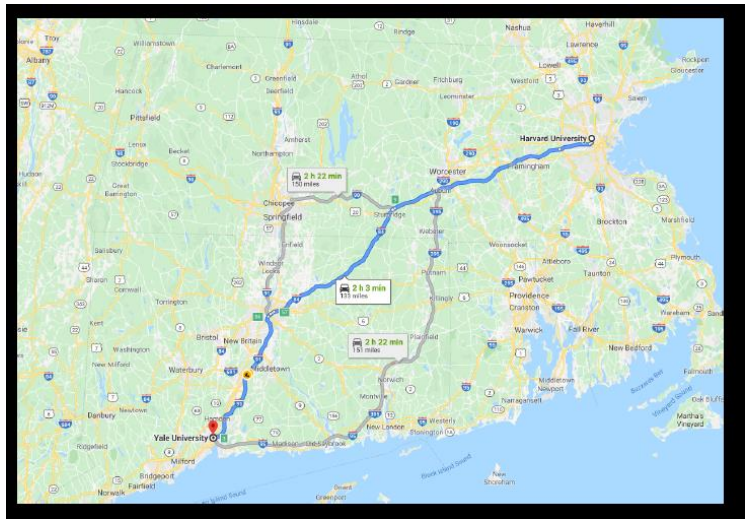
- An agent with several immediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence
- Looking for such a sequence is called search
- A search algorithm takes a problem as input and returns a solution in the form of action sequence
- One a solution is found the actions it recommends can be carried out – execution phase

Problem solving agents

- "formulate, search, execute" design for the agent
- After formulating a goal and a problem to solve the agent calls a search procedure to solve it
- It then uses the solution to guide its actions, doing whatever the solution recommends as the next thing to do (typically the first action in the sequence)
- Then removing that step from the sequence
- Once the solution has been executed, the agent will formulate a new goal





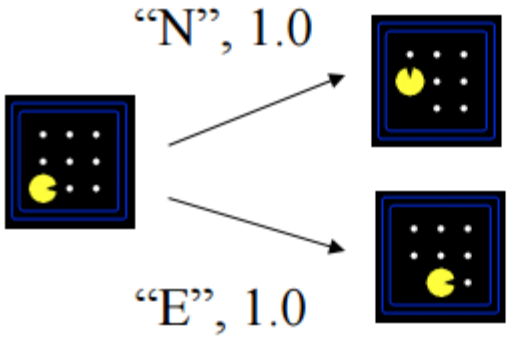


Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Search Problems

- A search problem consists of:
  - A state space

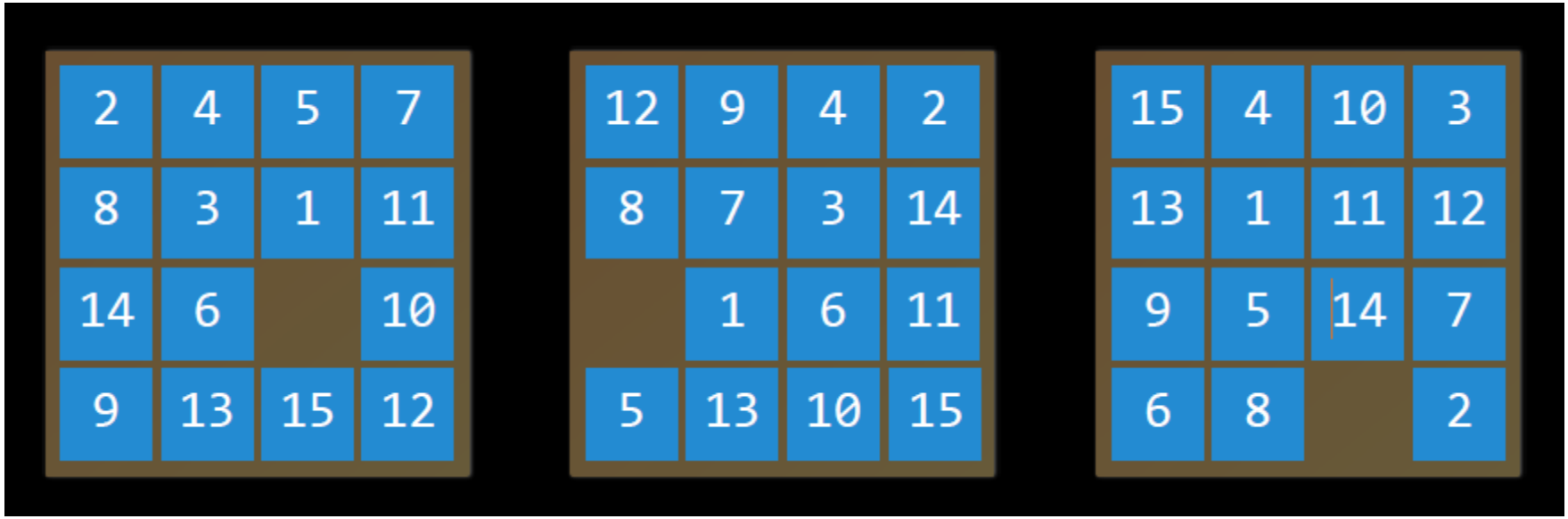
  - A successor function



  - A start state and a goal test
- A solution is a sequence of actions (a plan) which transforms the start state to a goal state
  - The **performance measure** is defined by (a) reaching the goal and (b) how "expensive" the path to the goal is

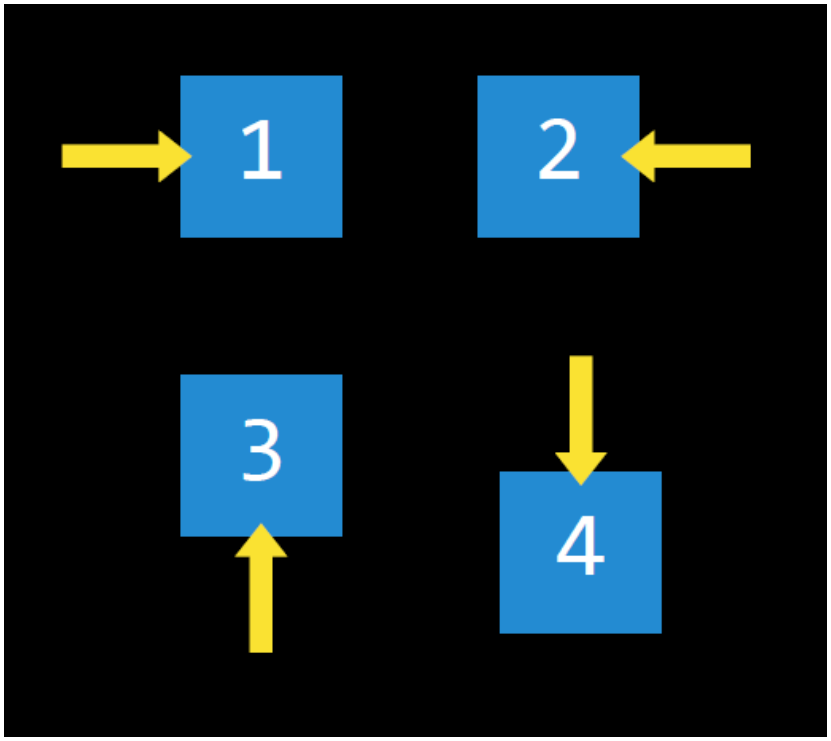
State

a configuration of the agent and its environment



Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

actions
choices that can be made in a state



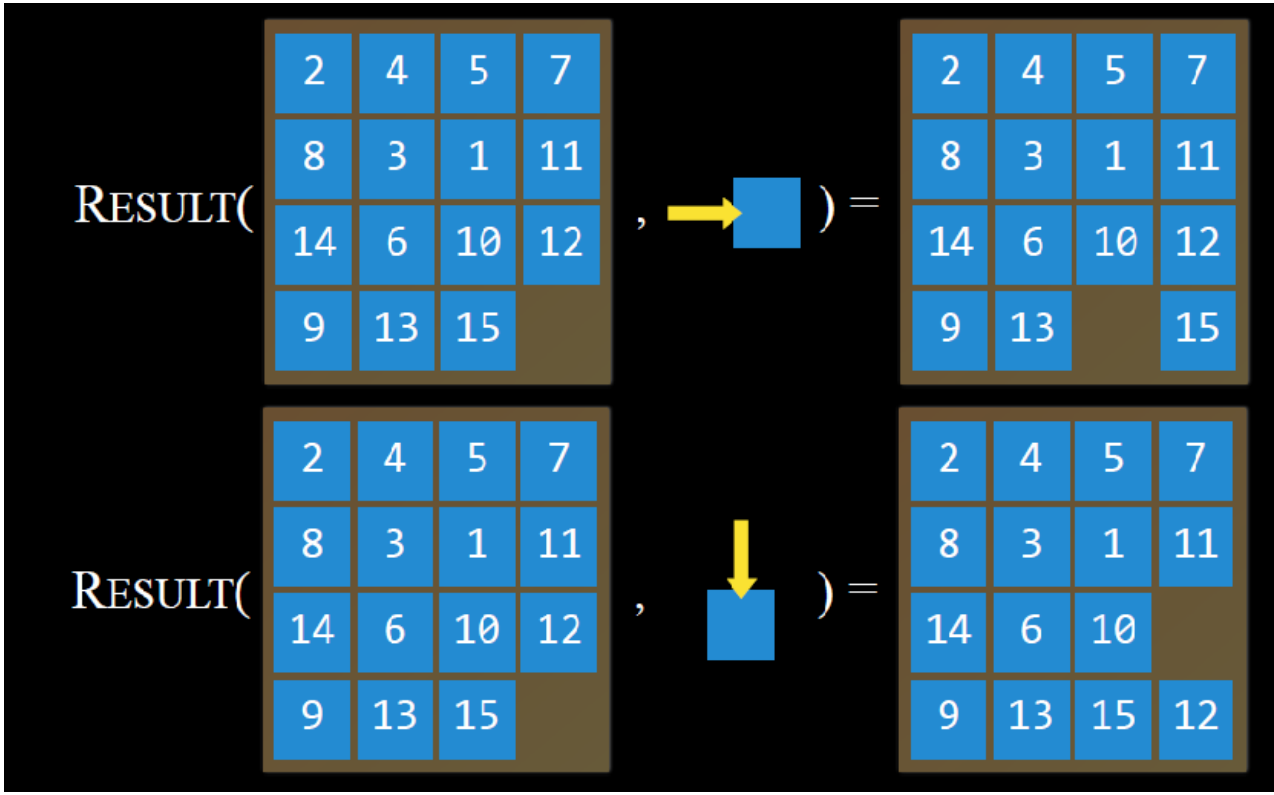
ACTIONS(*s*) returns the set of actions that can be executed in state *s*

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

transition model

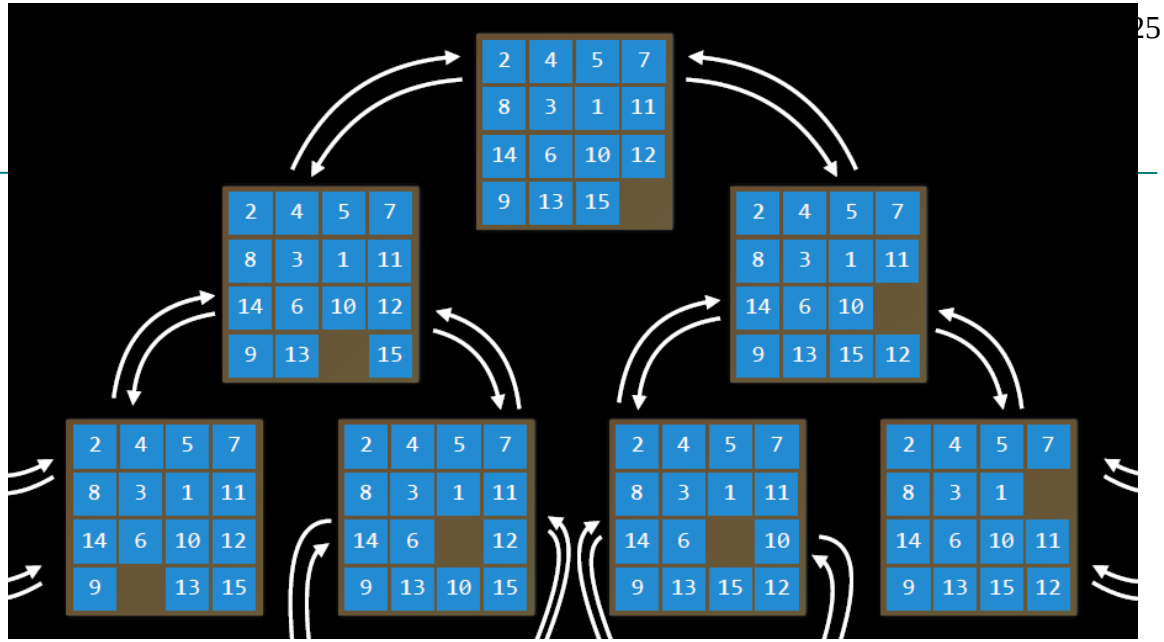
a description of what state results from performing any applicable action in any state

RESULT(*s*, *a*) returns the state resulting from performing action *a* in state *s*

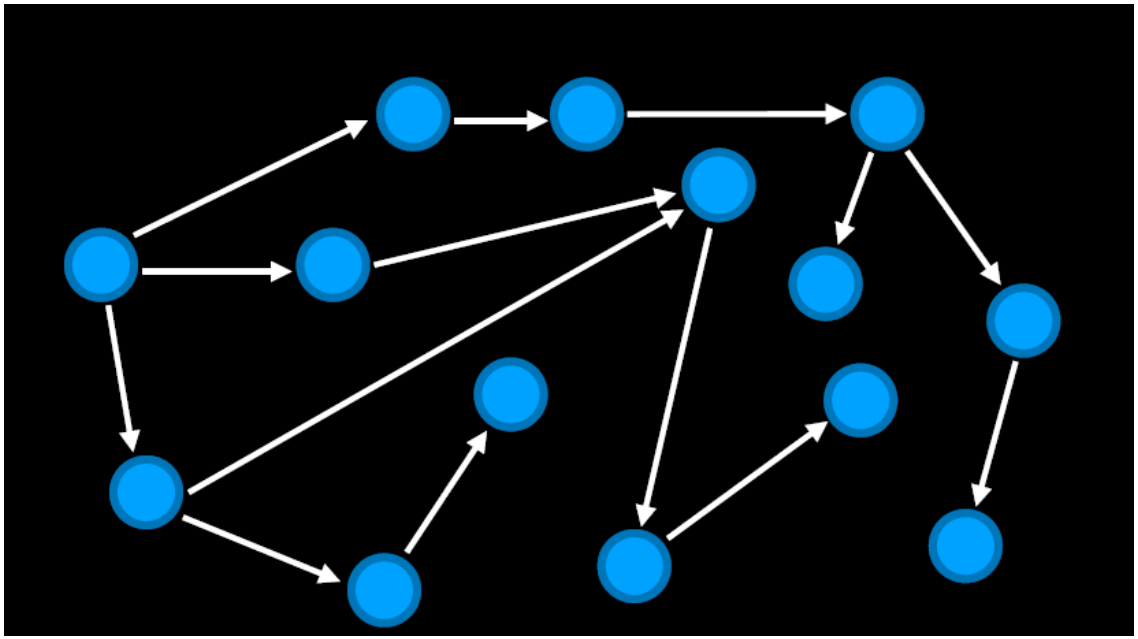


Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

state space
the set of all states reachable
from the initial state by
any sequence of actions



goal test
way to determine
whether a given state is a
goal state

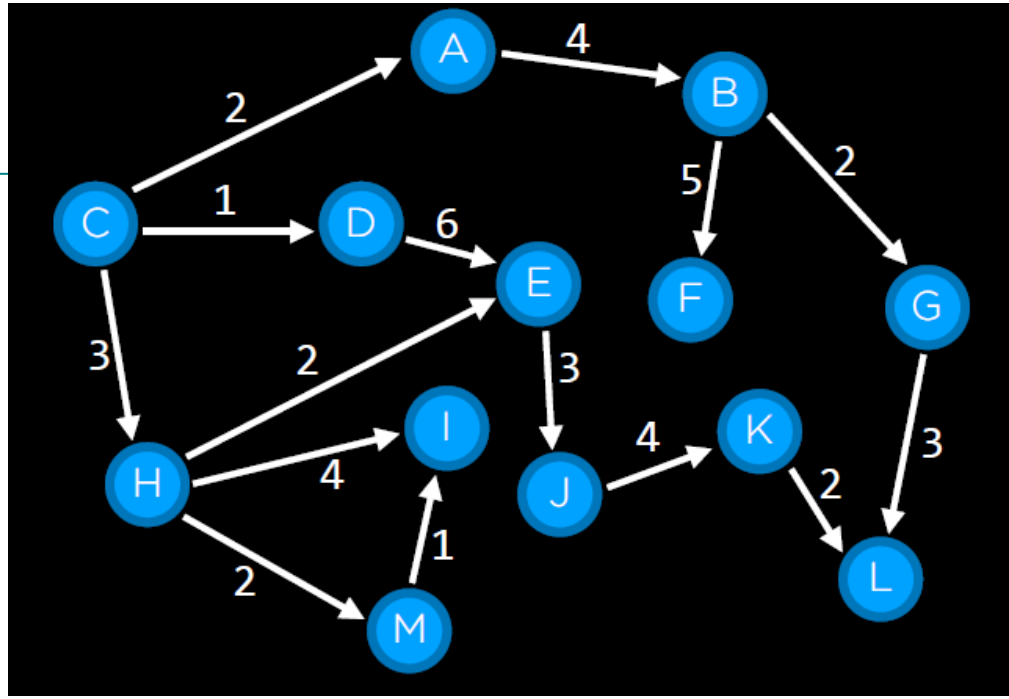


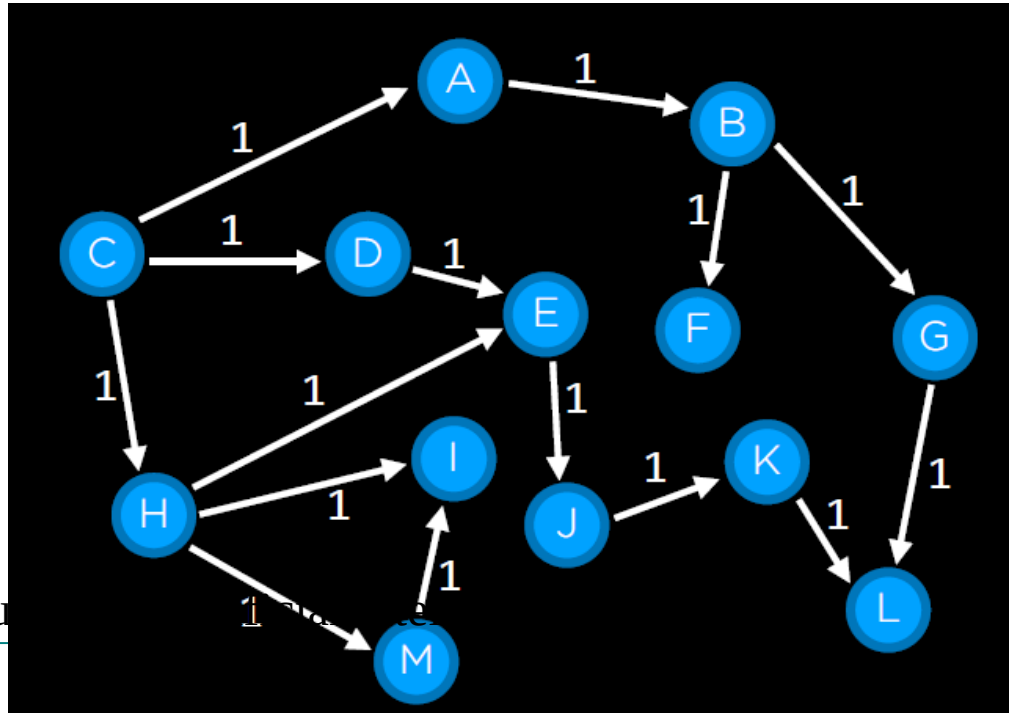
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

path cost
numerical cost associated with a given path

solution
a sequence of actions that leads from the initial state to a goal state

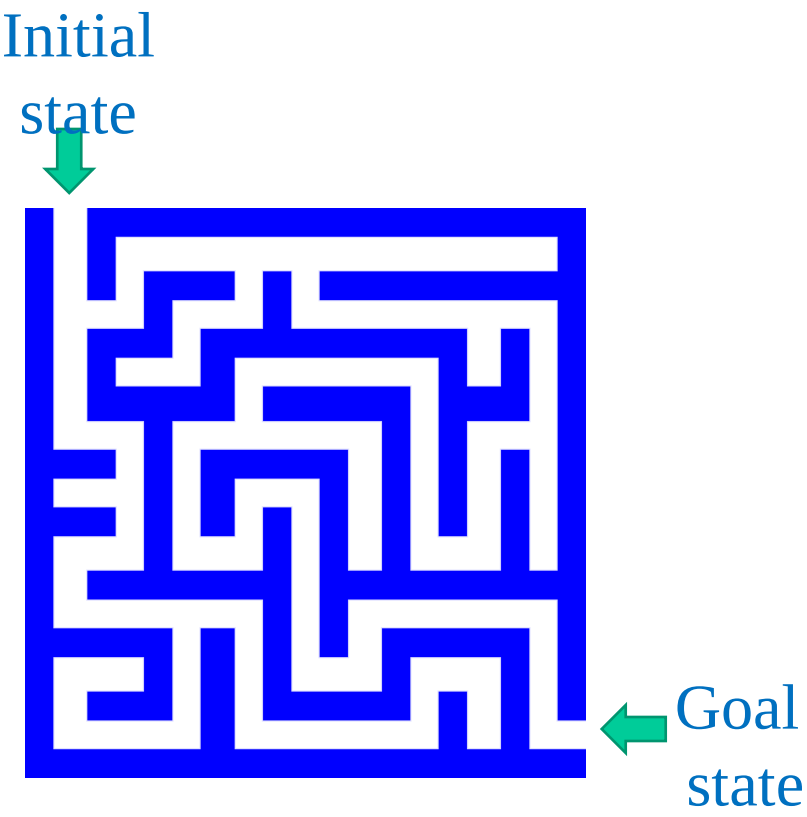
optimal solution
a solution that has the lowest path cost among all solutions





Search problem components

- **Initial state**
- **Actions**
- **Transition model**
  - What state results from performing a given action in a given state?
- **Goal state**
- **Path cost**
  - Assume that it is a sum of nonnegative *step costs*

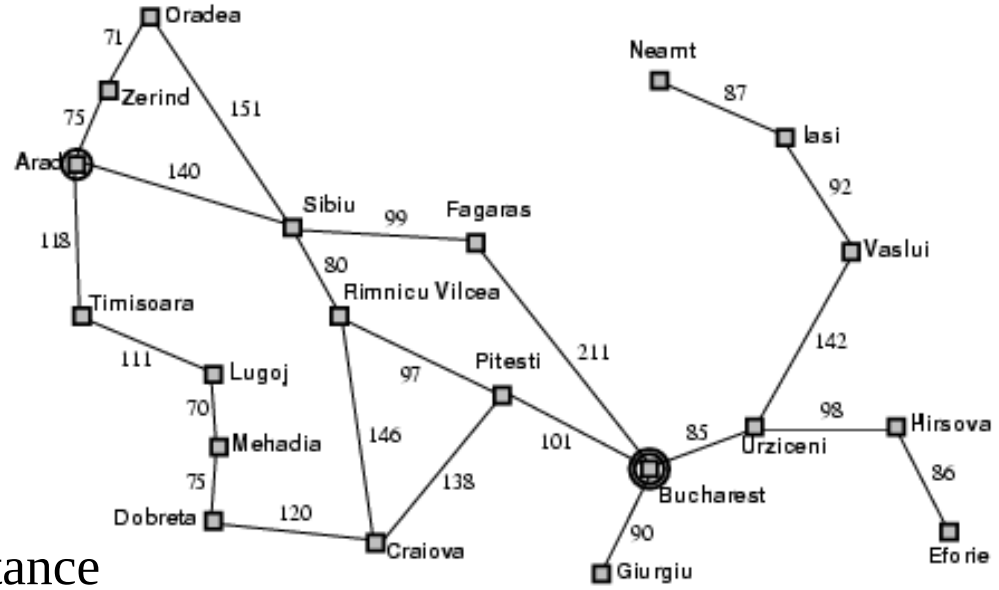


- The **optimal solution** is the sequence of actions that gives the *lowest* path cost for reaching the goal

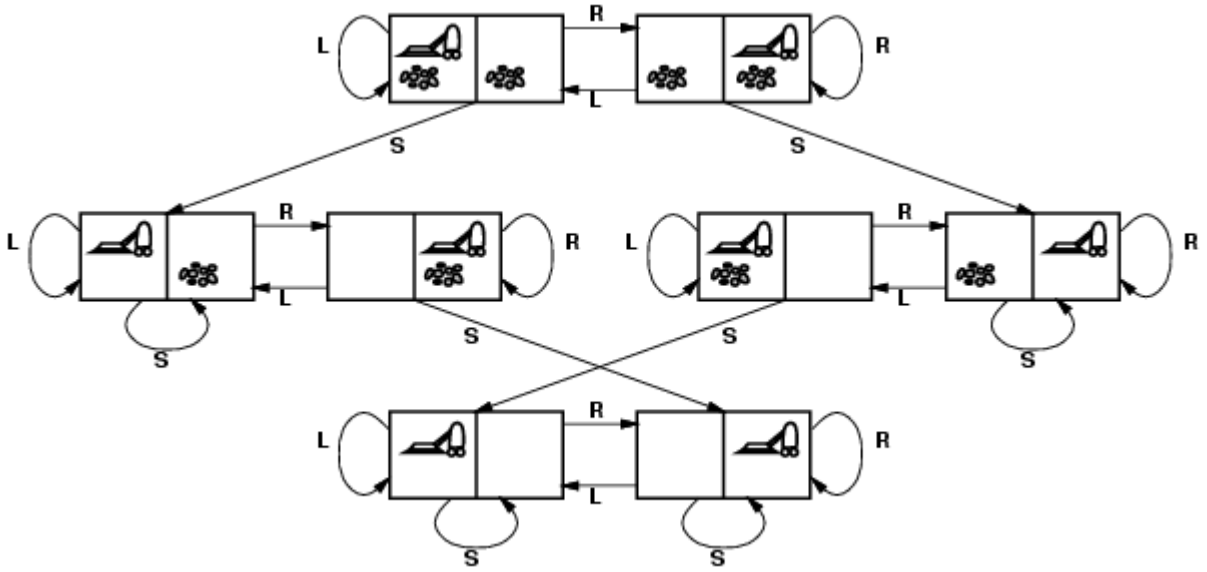
Example: Romania

- On vacation in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest

- **Initial state**
  - Arad
- **Actions**
  - Go from one city to another
- **Transition model**
  - If you go from city A to city B, you end up in city B
- **Goal state**
  - Bucharest
- **Path cost**
  - Sum of edge costs (total distance traveled)

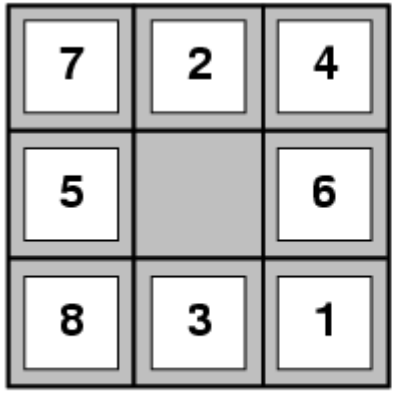


Vacuum world state space graph

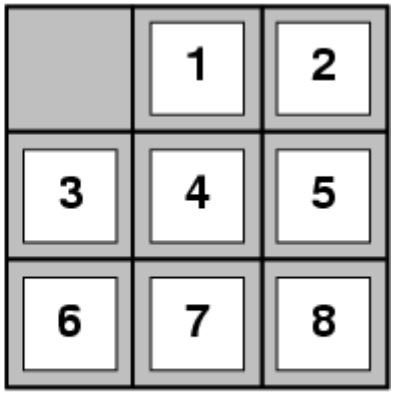


- states integer dirt and robot location.
  - The agent is in one of two locations, each of which might or might not contain dirt – 8 possible states
- Initial state: any state
- actions *Left*, *Right*, *Suck*
- goal test no dirt at all locations
- path cost 1 per action

Example: The 8-puzzle



Start State

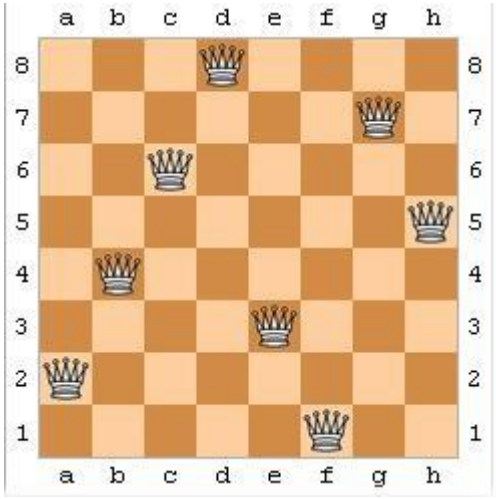


Goal State

- states: locations of tiles
- Initial state: any state
- actions: move blank left, right, up, down
- goal test: goal state (given)
- path cost: 1 per move

[Note: optimal solution of *n*-Puzzle family is NP-hard]

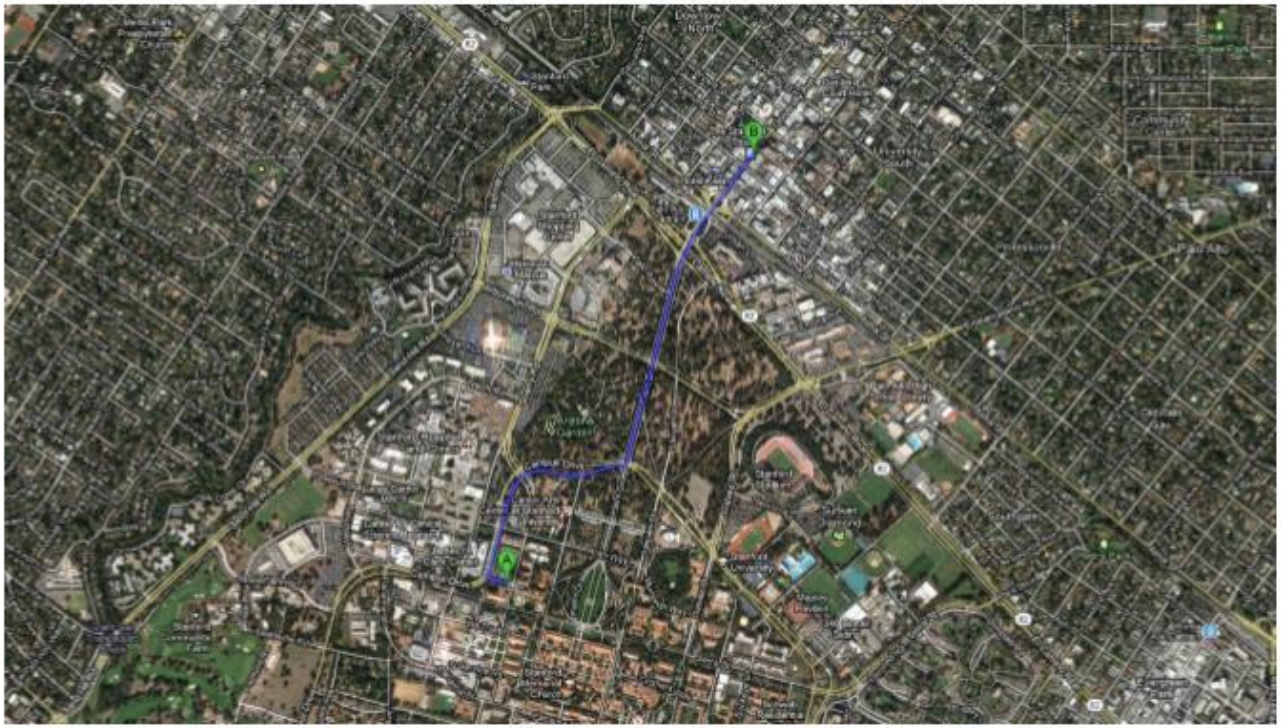
Example: 8-queens problem



- states: any arrangement of 0-8 queens on the board is a state
- Initial state: no queens on the board
- actions: add a queen to any empty square
- goal test: 8 queens are on the board, none attacked

64.63...57 = 1.8×10^{14} possible sequences

Application: route finding



Objective: shortest? fastest? most scenic?

Actions: go straight, turn left, turn right

Example: Route finding problem

- states: each is represented by a location (e.g. An airport) and the current time
- Initial state: specified by the problem
- Successor function: returns the states resulting from taking any scheduled flight, leaving later than the current time plus the within airport transit time, from the current airport to another
- goal test: are we at the destination by some pre-specified time
- Path cost:monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards, etc

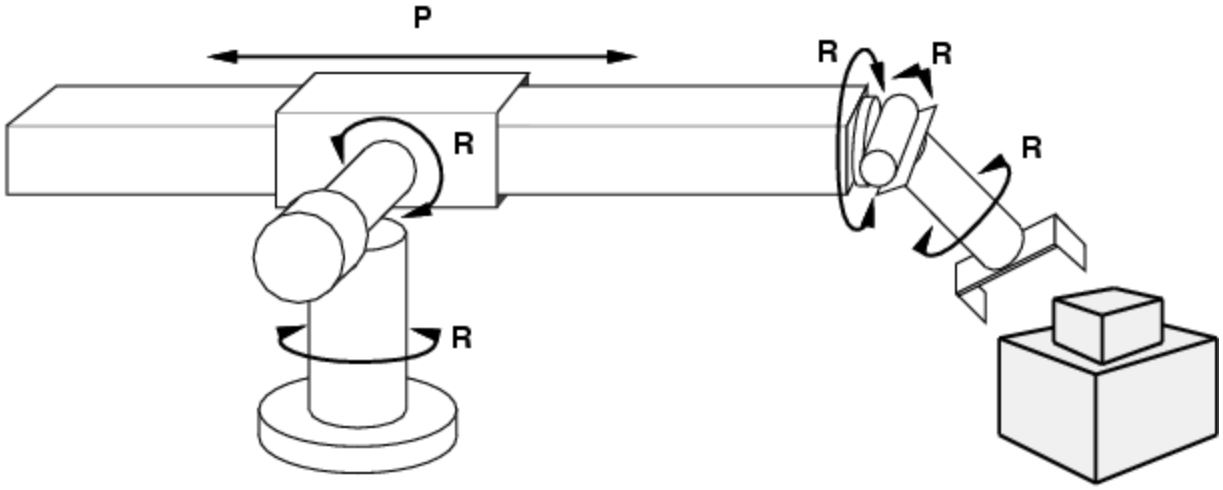
- Route finding algorithms are used in a variety of applications, such as routing in computer networks, military operations planning, airline travel planning systems

Application: robot motion planning

Objective: fastest? most energy efficient? safest?

Actions: translate and rotate joints

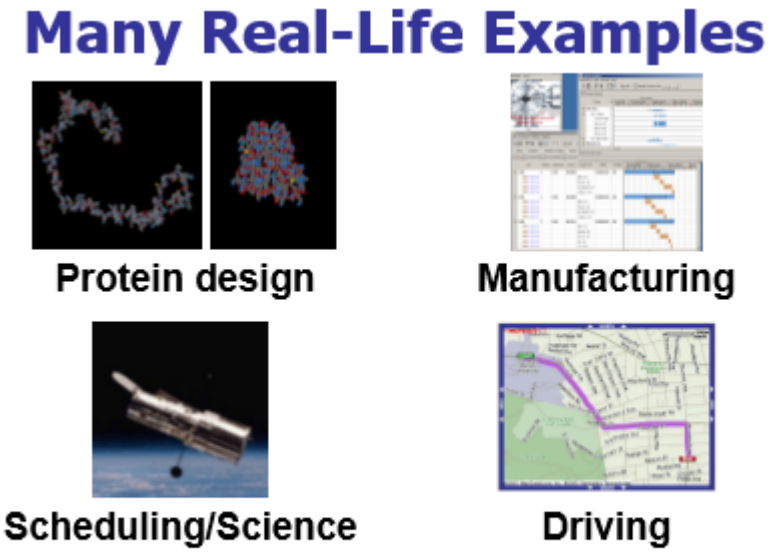
Example: robotic assembly



- states: real-valued coordinates of robot joint angles parts of the object to be assembled
- actions: continuous motions of robot joints
- goal test: complete assembly
- path cost: time to execute

Other example problems

- Touring problems: visit every city at least once, starting and ending at Bucharest
- Travelling salesperson problem (TSP) : each city must be visited exactly once – find the shortest tour
- VLSI layout design: positioning millions of components and connections on a chip to minimize area, minimize circuit delays, minimize stray capacitances, and maximize manufacturing yield
- Robot navigation
- Internet searching
- Automatic assembly sequencing
- Protein design



node

a data structure that keeps track of

- a **state**
- a **parent** (node that generated this node)
- an **action** (action applied to parent to get node)
- a **path cost** (from initial state to node)

Approach

Start with a **frontier** that contains the initial state.

Repeat:

If the frontier is empty, then no solution.

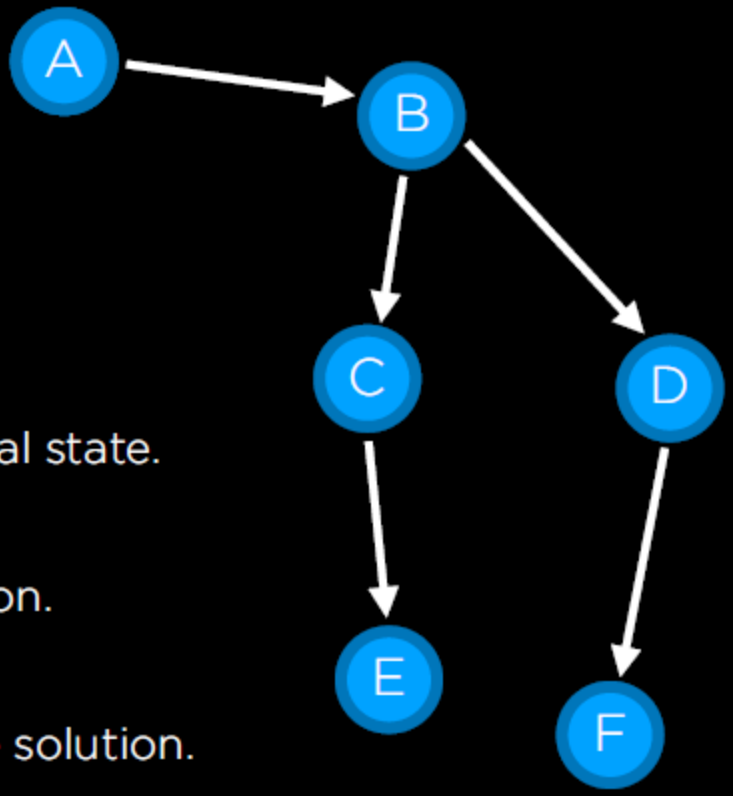
Remove a node from the frontier.

If node contains goal state, return the solution.

Expand node, add resulting nodes to the frontier.

Find a path from A to E.

Frontier



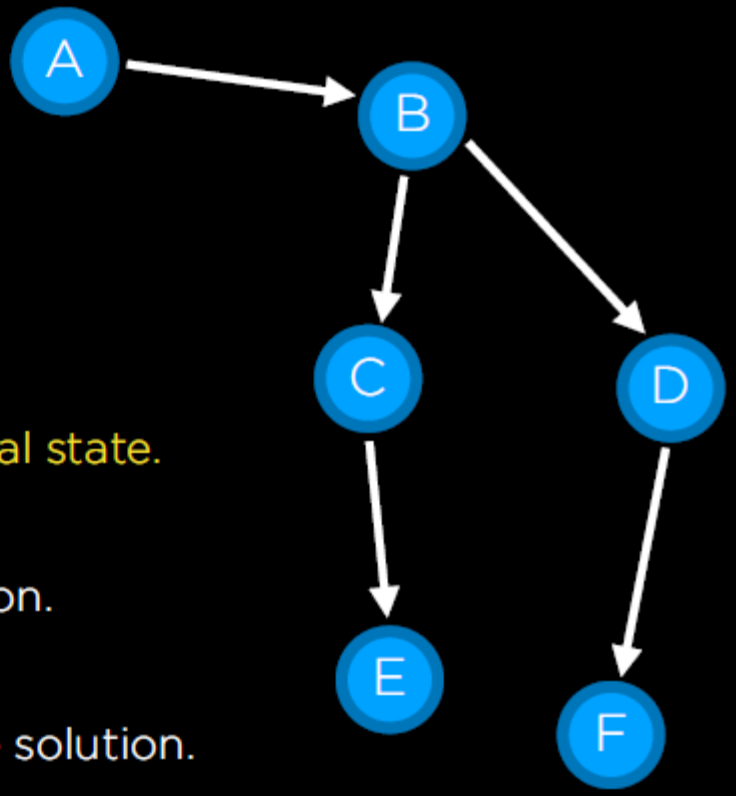
- Start with a **frontier** that contains the initial state.
- Repeat:
  - If the frontier is empty, then no solution.
  - Remove a node from the frontier.
  - If node contains goal state, return the solution.
  - **Expand** node, add resulting nodes to the frontier.

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Find a path from A to E.

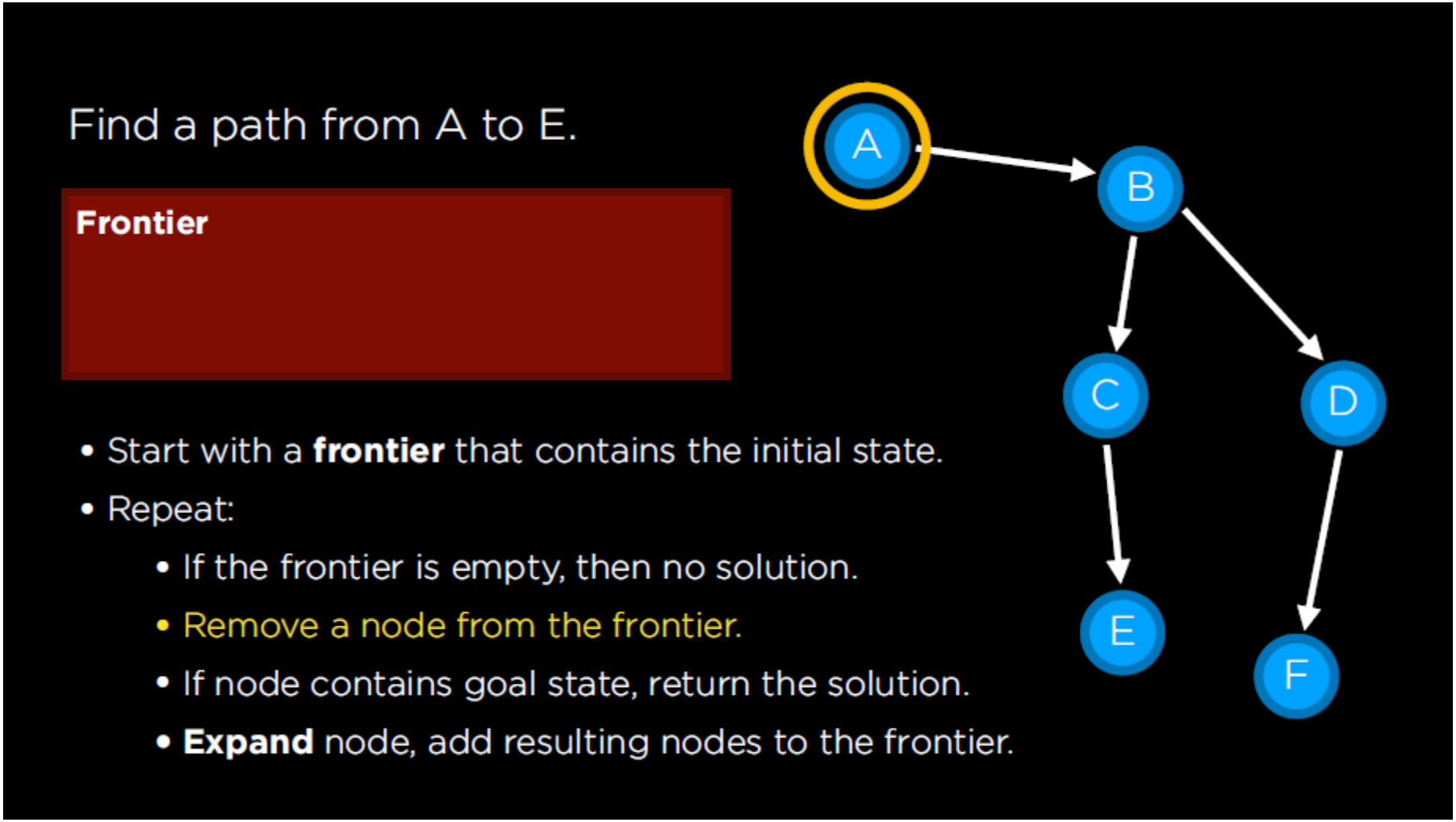
Frontier

A

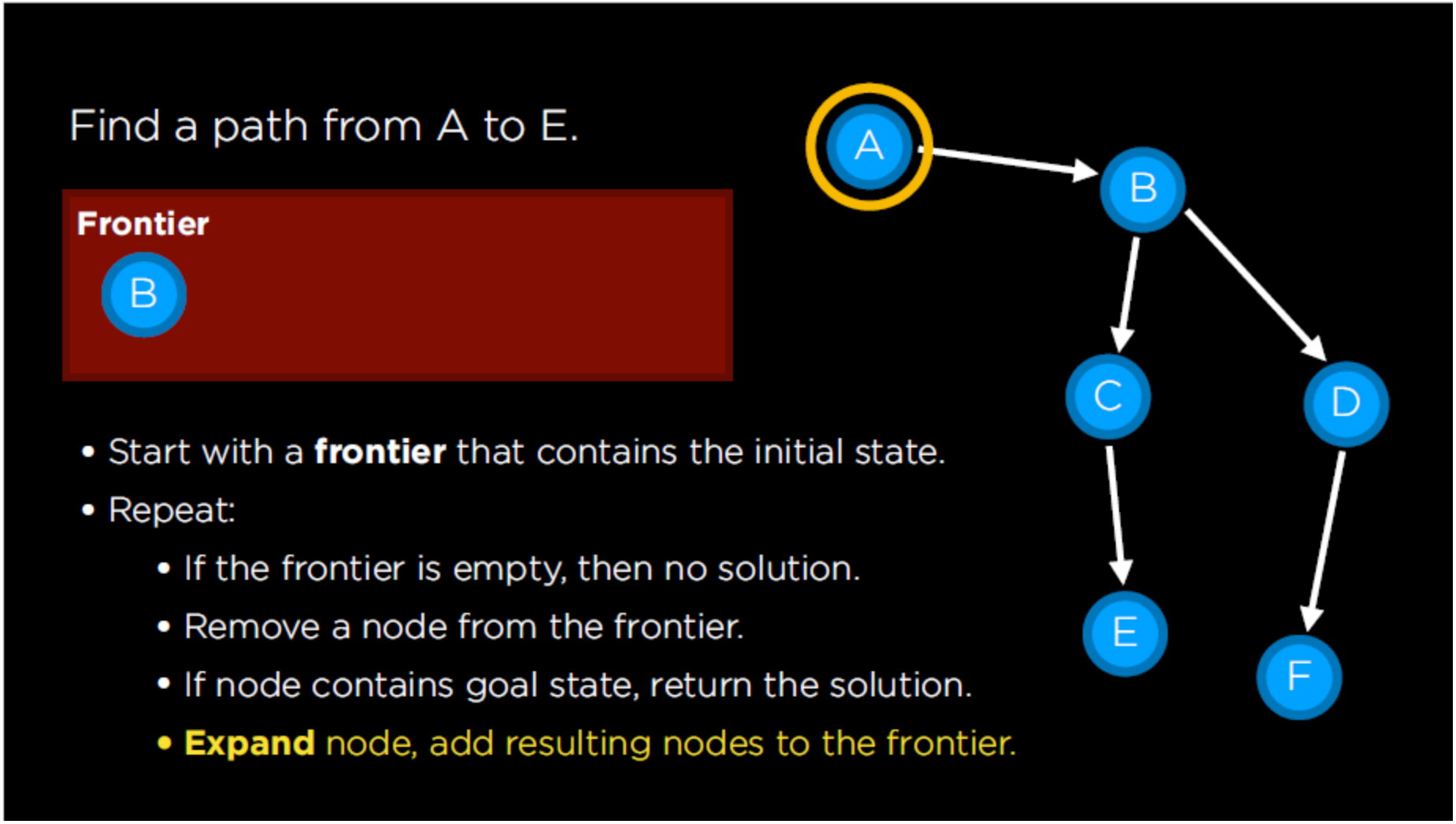


- Start with a **frontier** that contains the initial state.
- Repeat:
  - If the frontier is empty, then no solution.
  - Remove a node from the frontier.
  - If node contains goal state, return the solution.
  - **Expand** node, add resulting nodes to the frontier.

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



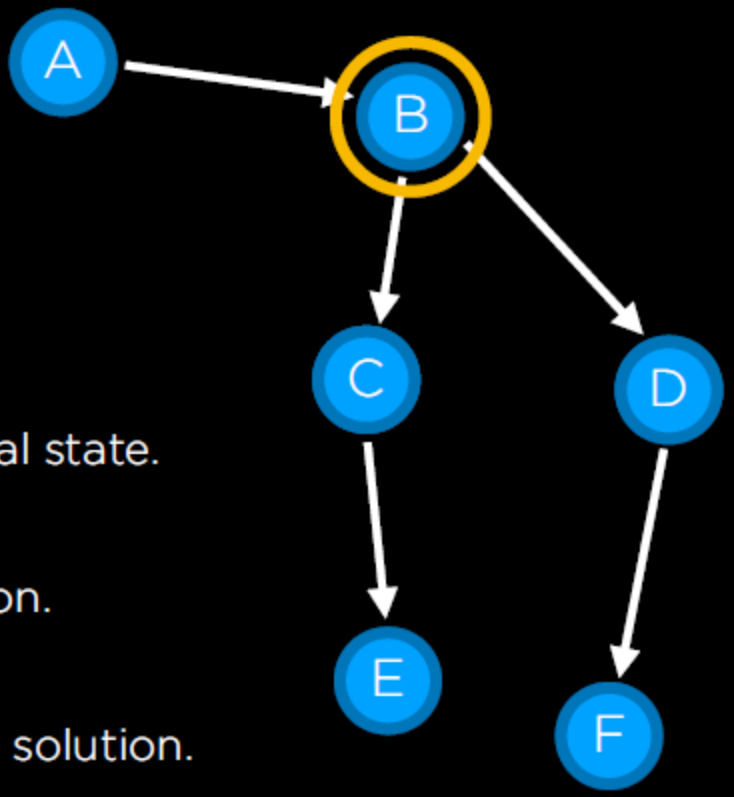
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Find a path from A to E.

Frontier



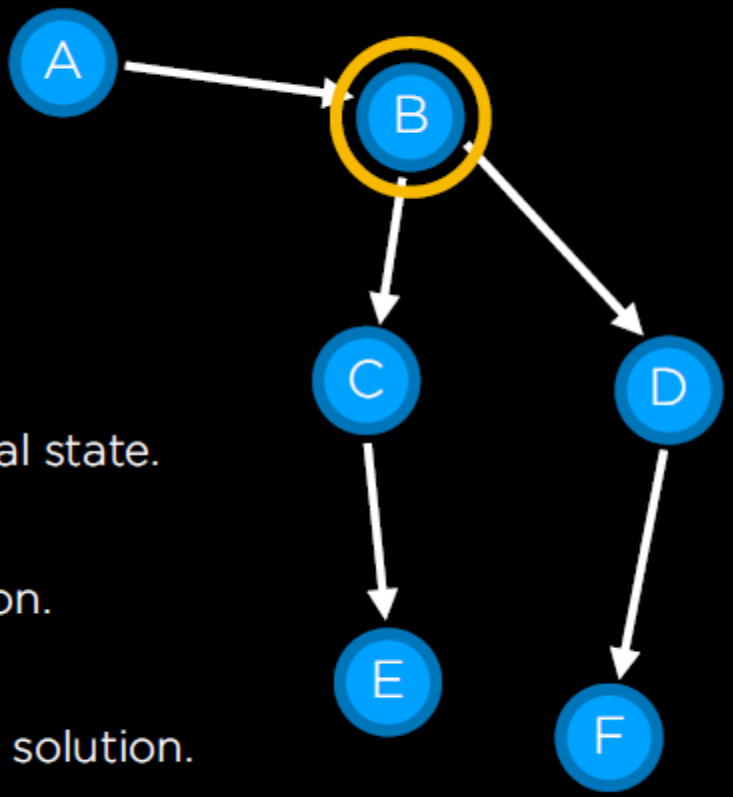
- Start with a **frontier** that contains the initial state.
- Repeat:
  - If the frontier is empty, then no solution.
  - Remove a node from the frontier.
  - If node contains goal state, return the solution.
  - **Expand** node, add resulting nodes to the frontier.

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Find a path from A to E.

Frontier

C D



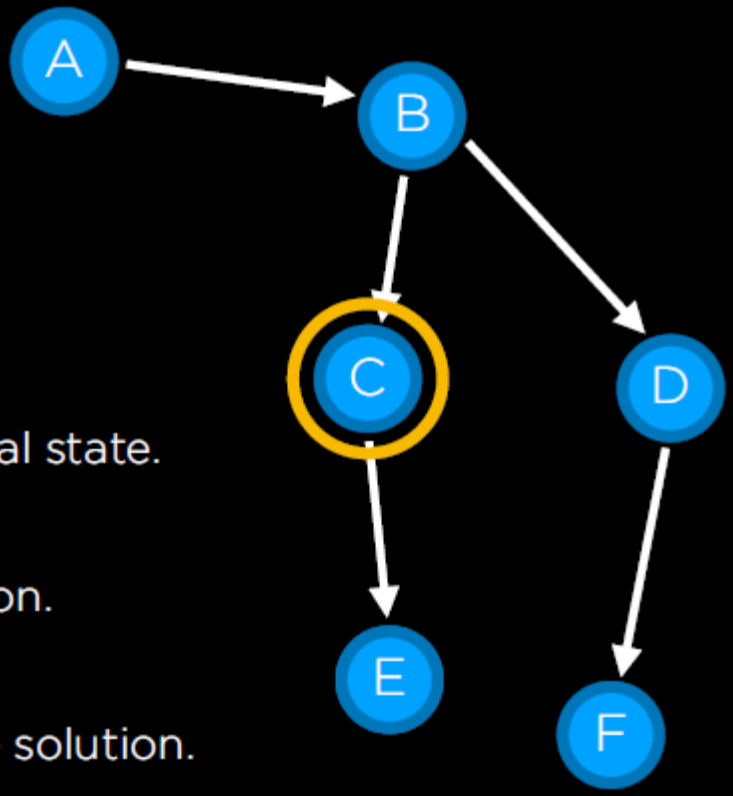
- Start with a **frontier** that contains the initial state.
- Repeat:
  - If the frontier is empty, then no solution.
  - Remove a node from the frontier.
  - If node contains goal state, return the solution.
  - **Expand** node, add resulting nodes to the frontier.

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Find a path from A to E.

Frontier

D



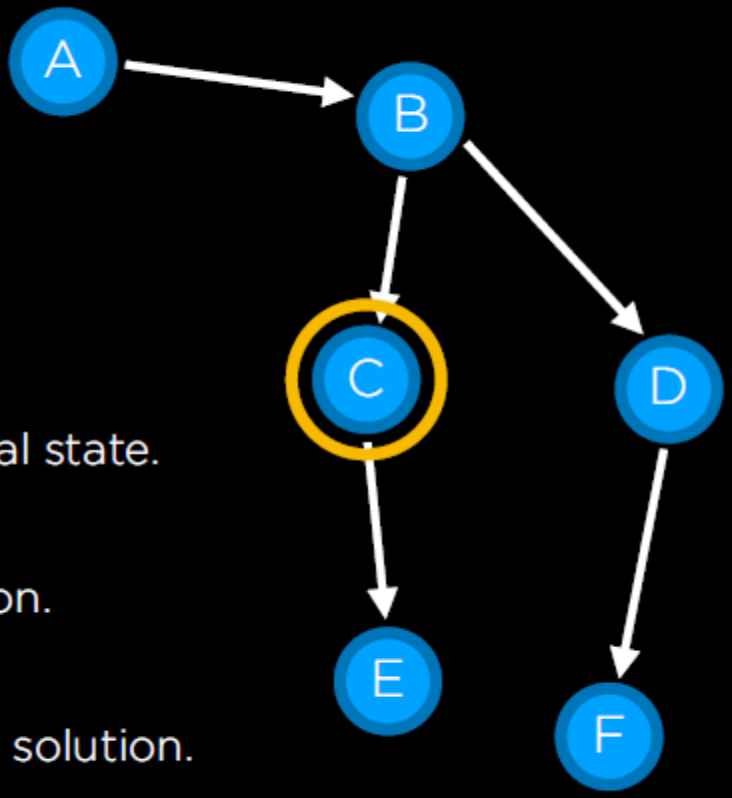
- Start with a **frontier** that contains the initial state.
- Repeat:
  - If the frontier is empty, then no solution.
  - Remove a node from the frontier.
  - If node contains goal state, return the solution.
  - **Expand** node, add resulting nodes to the frontier.

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Find a path from A to E.

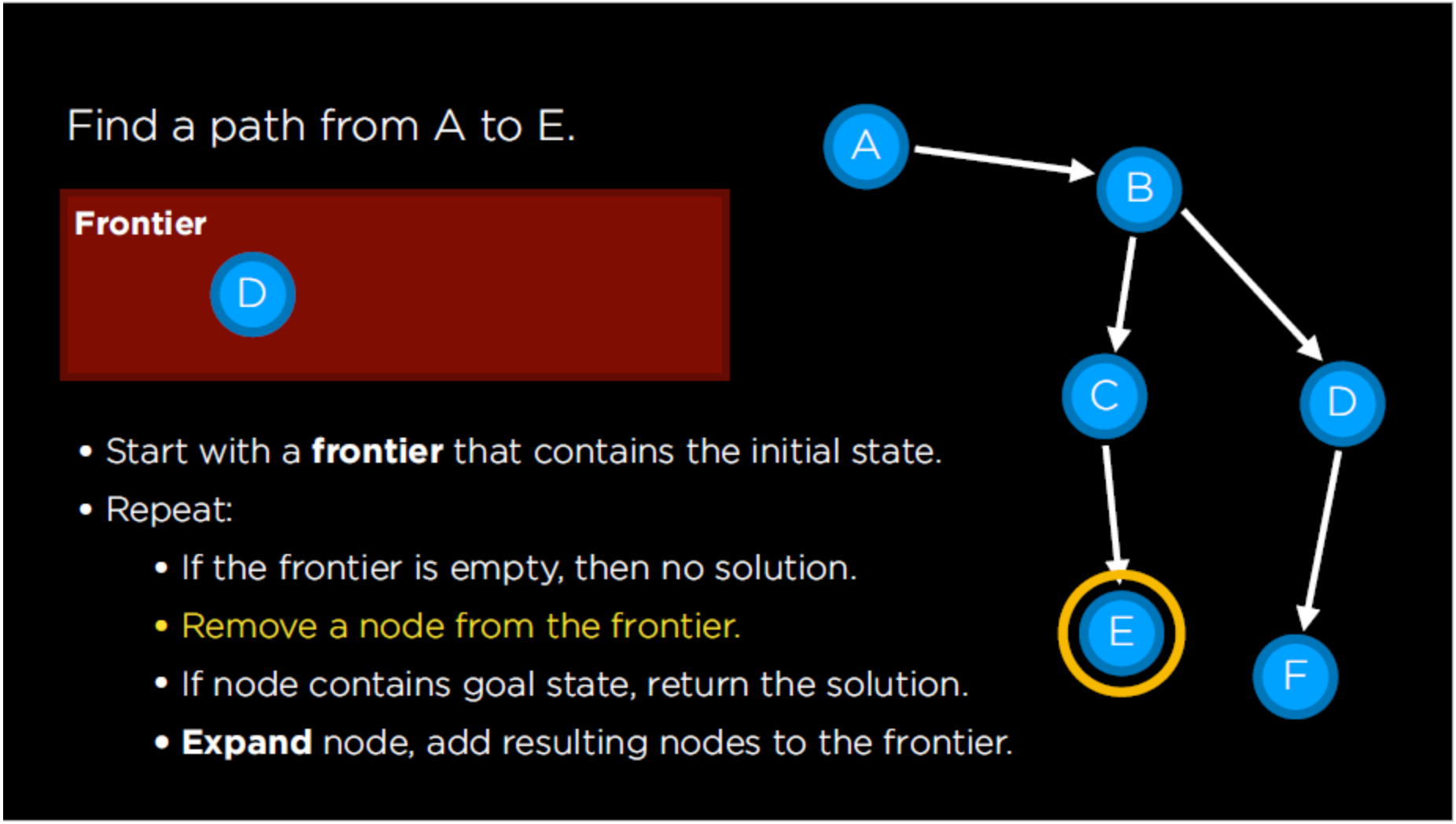
Frontier

E D

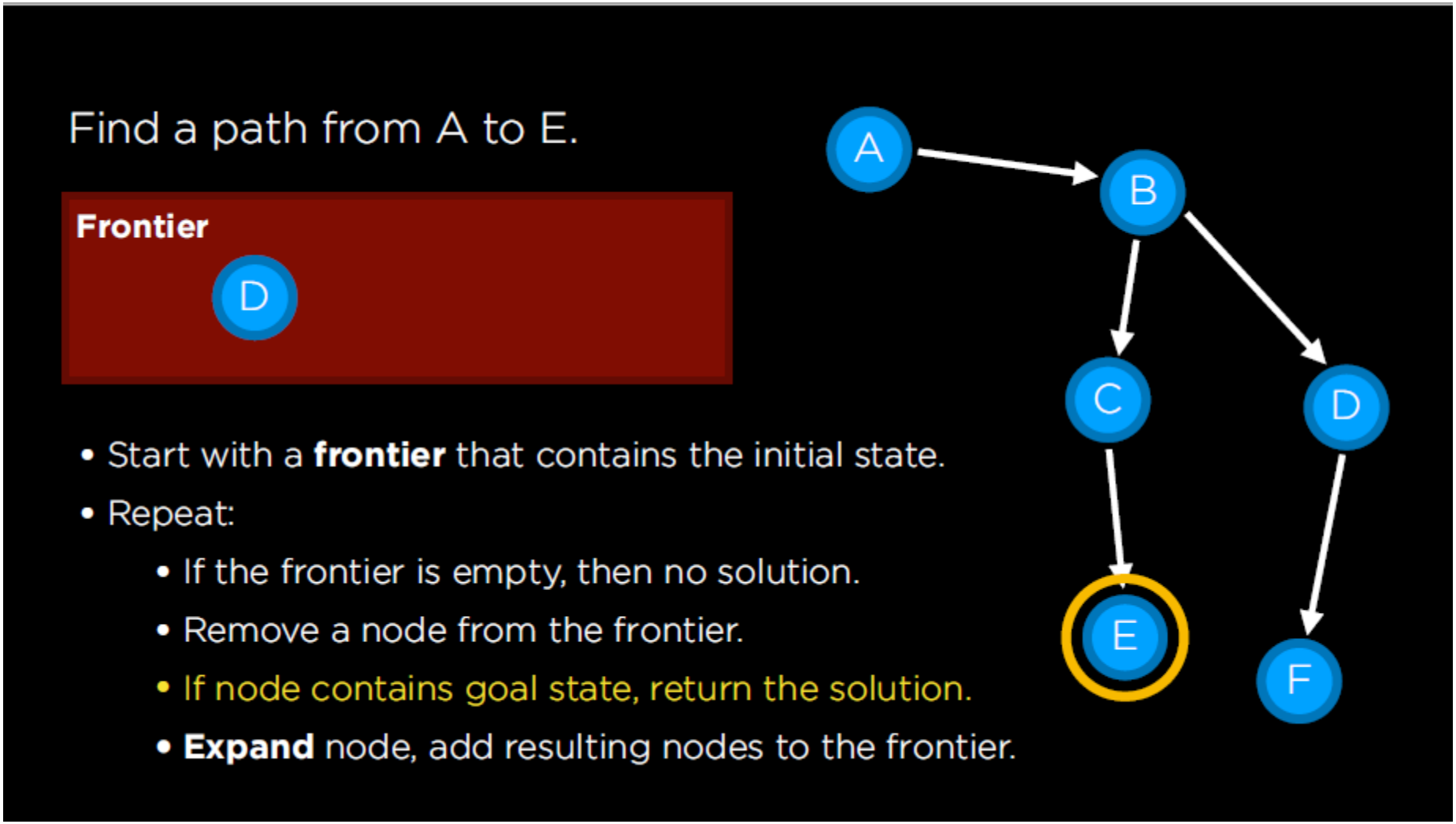


- Start with a **frontier** that contains the initial state.
- Repeat:
  - If the frontier is empty, then no solution.
  - Remove a node from the frontier.
  - If node contains goal state, return the solution.
  - **Expand** node, add resulting nodes to the frontier.

Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



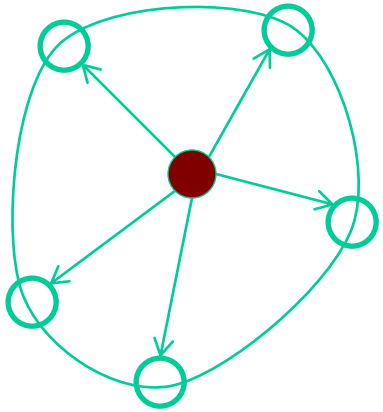
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



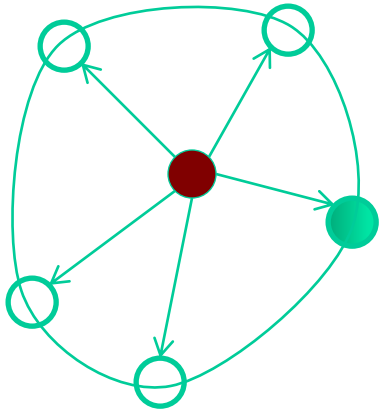
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Search: Basic idea

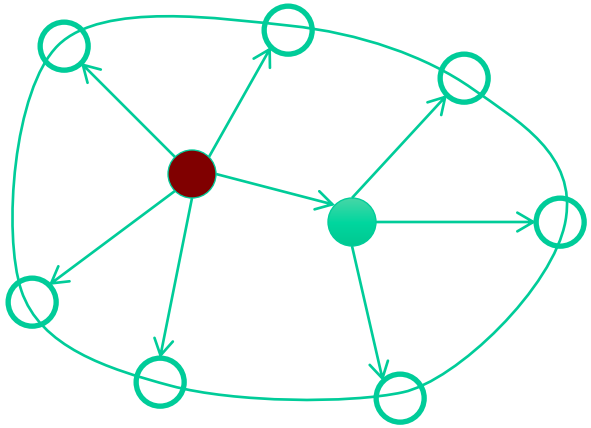
Search: Basic idea



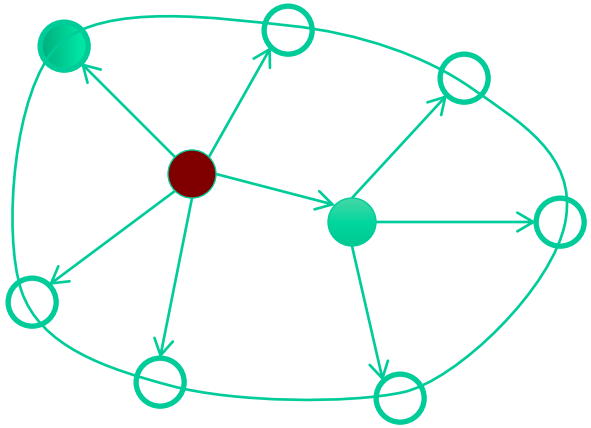
Search: Basic idea



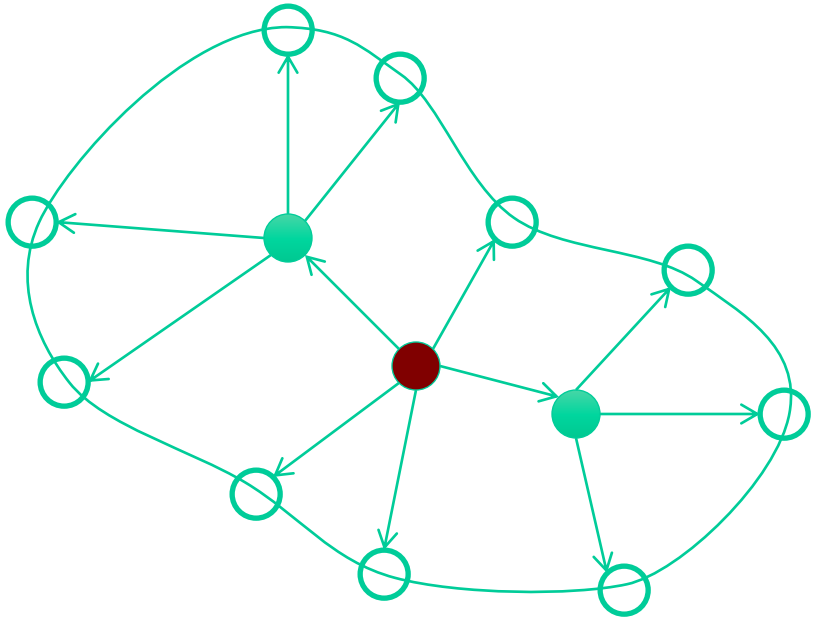
Search: Basic idea



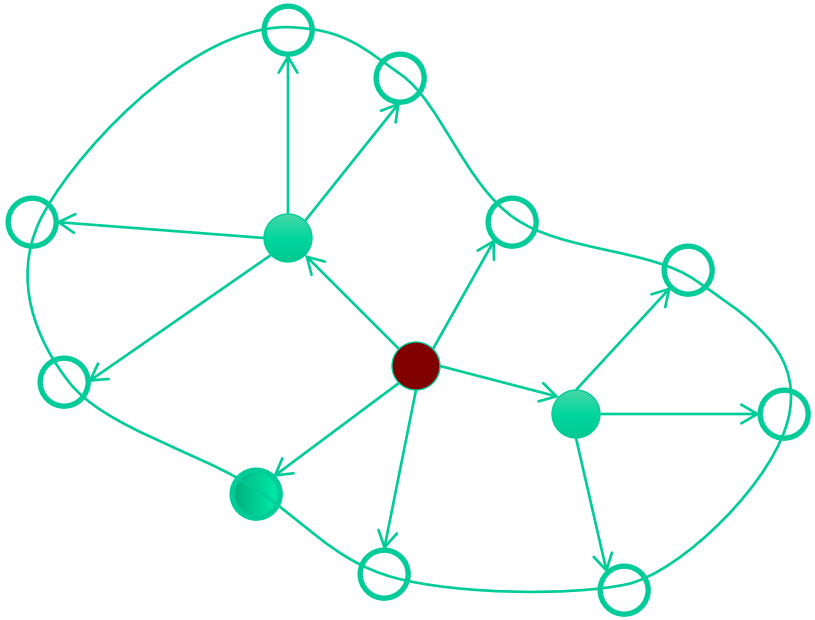
Search: Basic idea



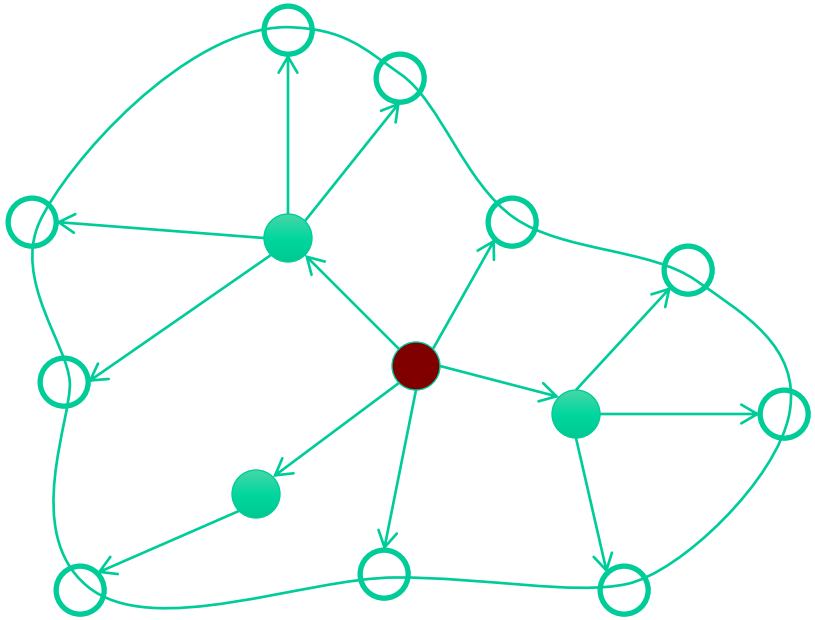
Search: Basic idea



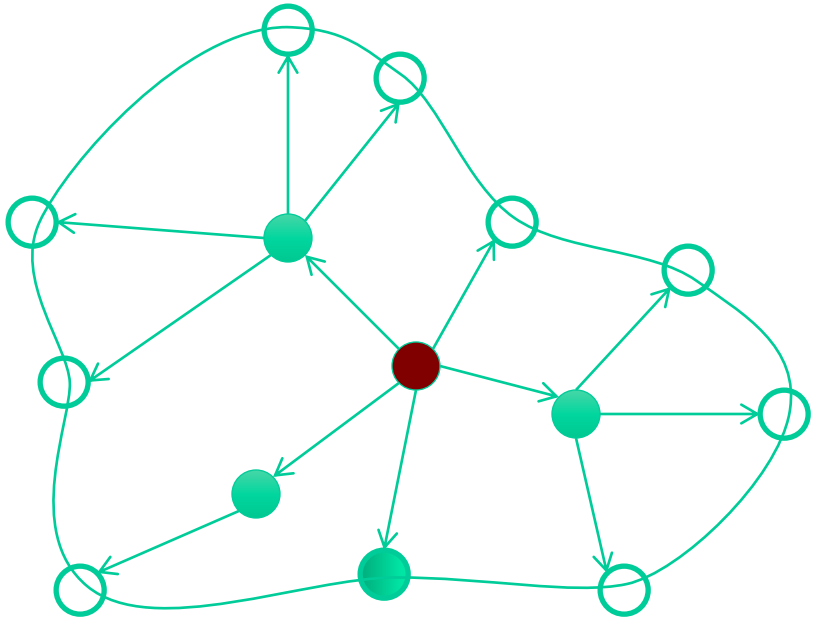
Search: Basic idea



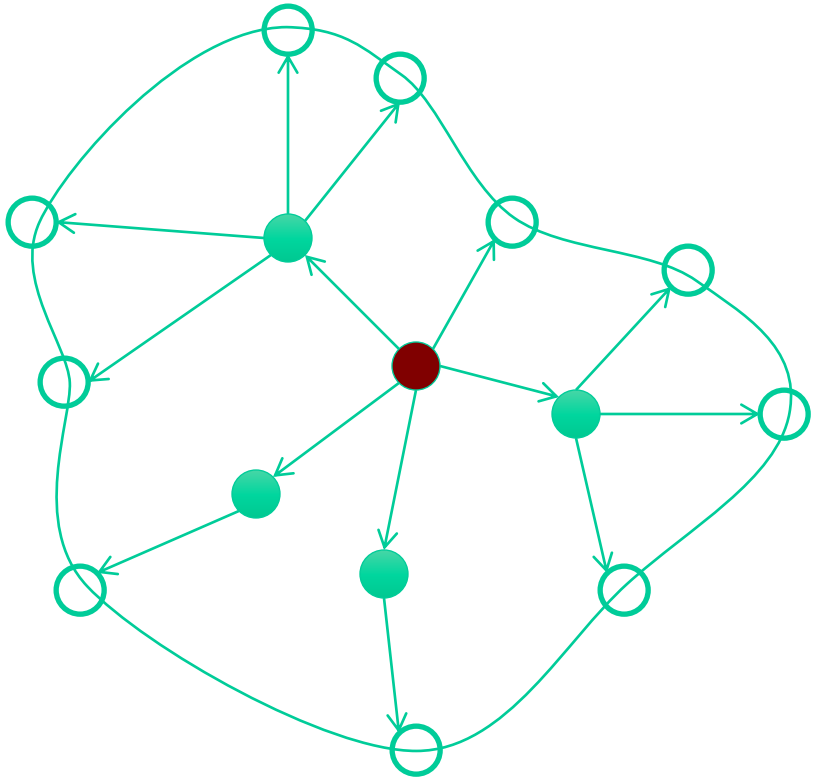
Search: Basic idea



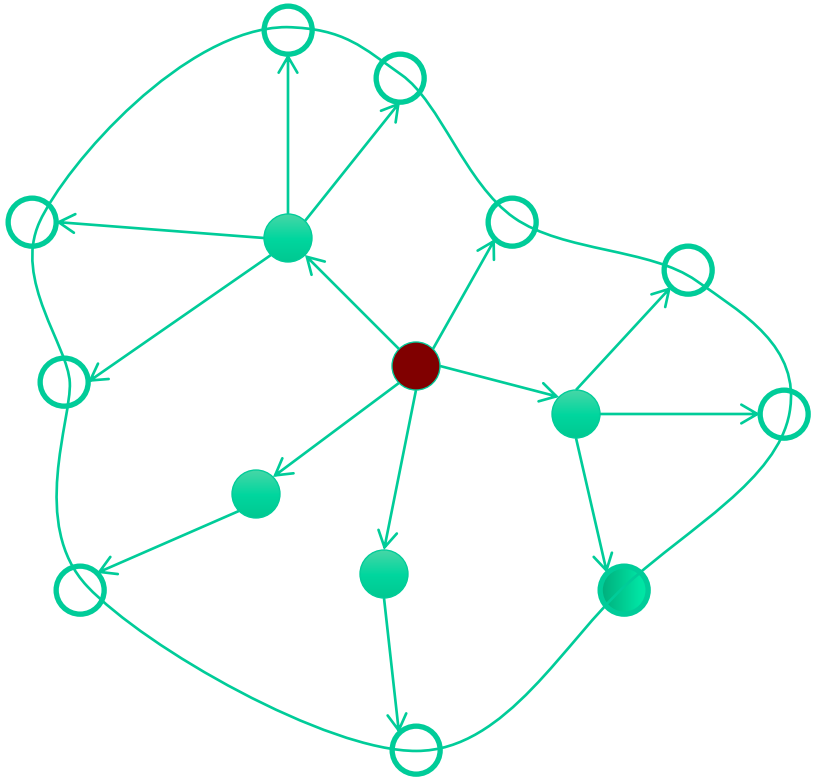
Search: Basic idea



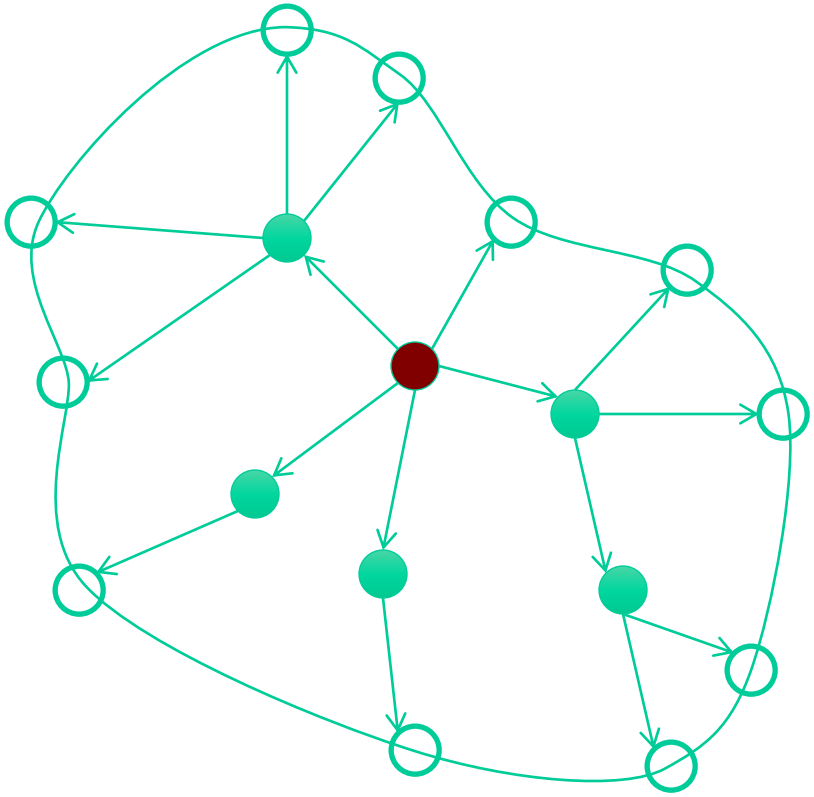
Search: Basic idea



Search: Basic idea

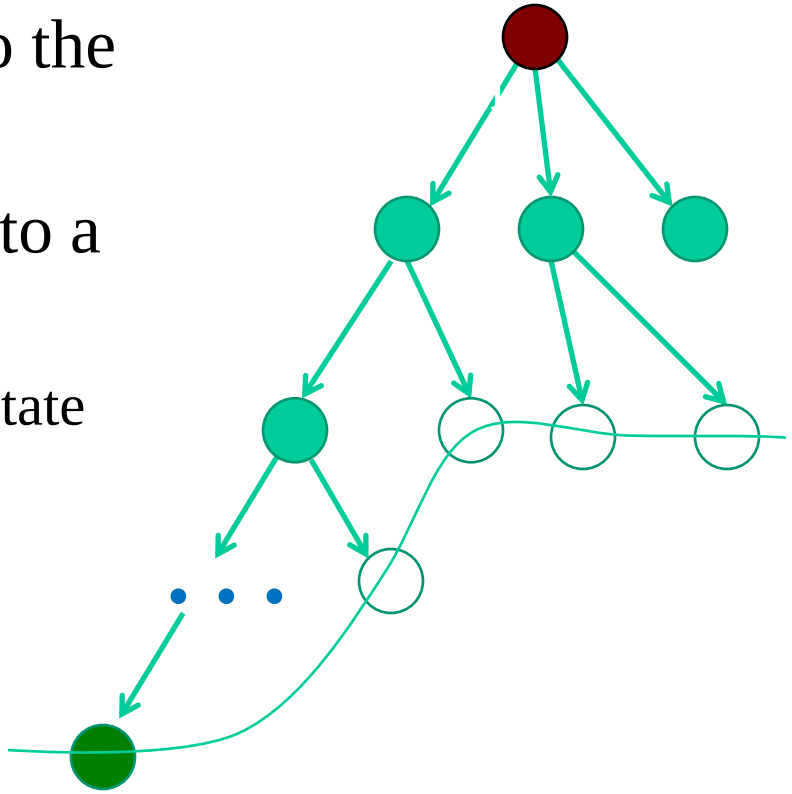


Search: Basic idea



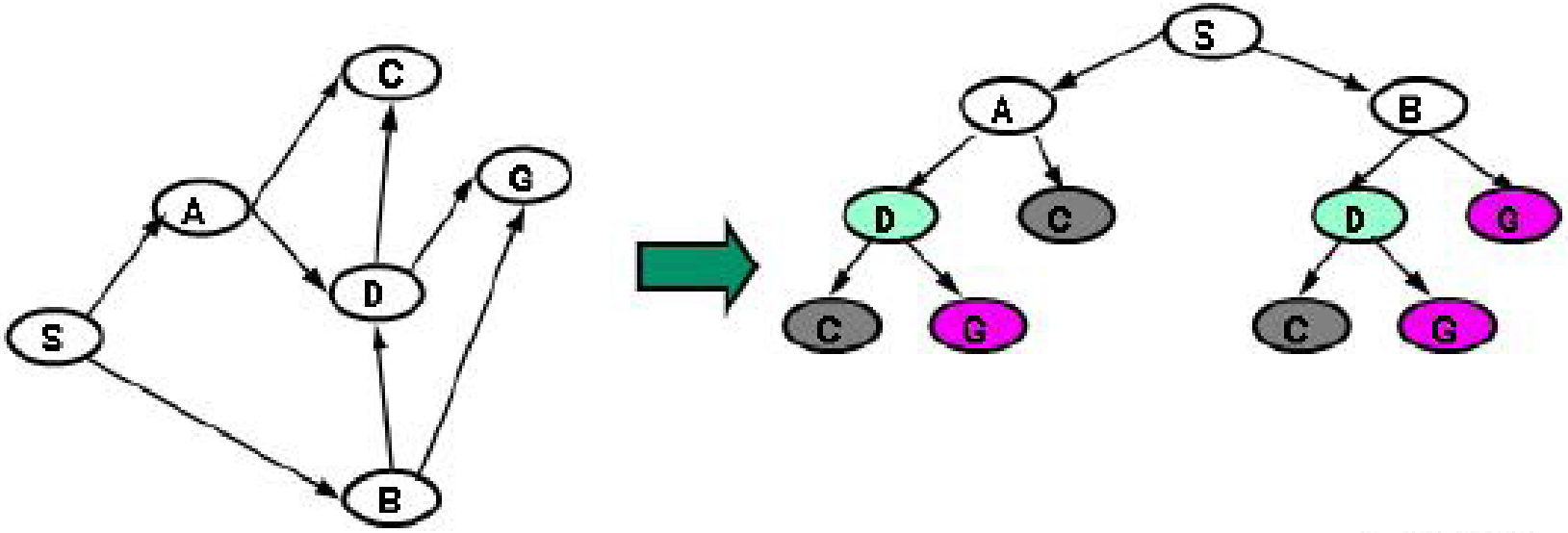
Search tree

- The root node corresponds to the starting state
- The children of a node correspond to the **successor states** of that node's state
- A path through the tree corresponds to a sequence of actions
  - A solution is a path ending in the goal state
- Nodes vs. states
  - A state is a representation of the world, while a node is a data structure that is part of the search tree
    - Node has to keep pointer to parent, path cost, possibly other info

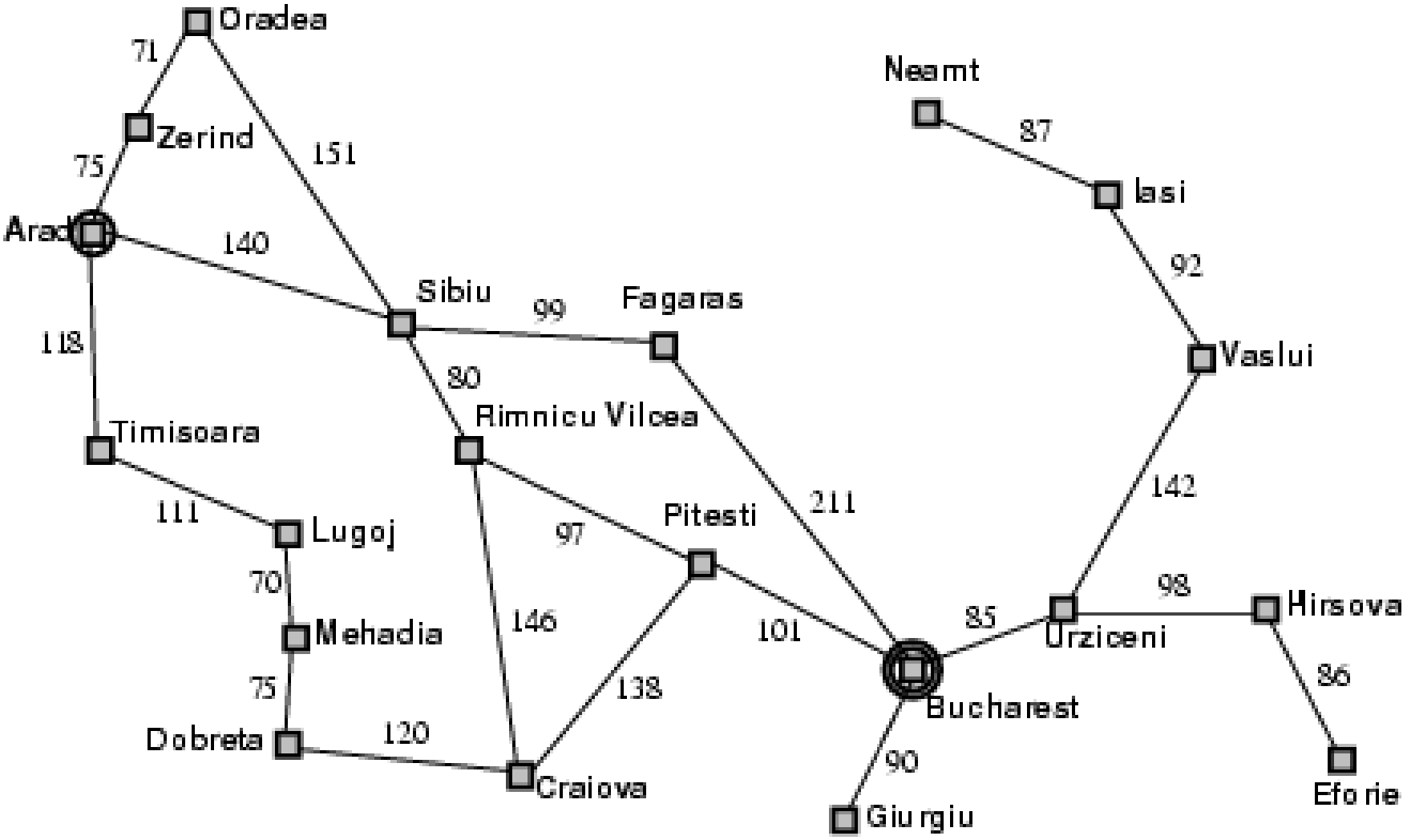


Graph Search as Tree Search

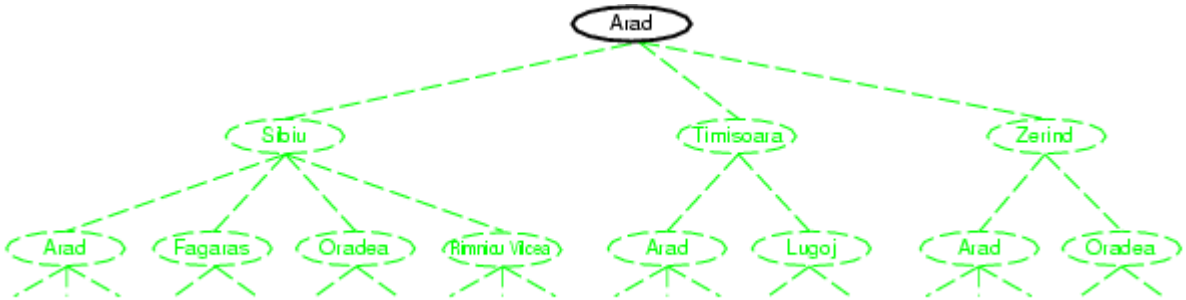
- Trees are directed graphs without cycles and with nodes having <= 1 parent
- We can turn graph search problems (from S to G) into tree search problems by:
  - replacing undirected links by 2 directed links
  - avoiding loops in path (or keeping track of visited nodes globally)

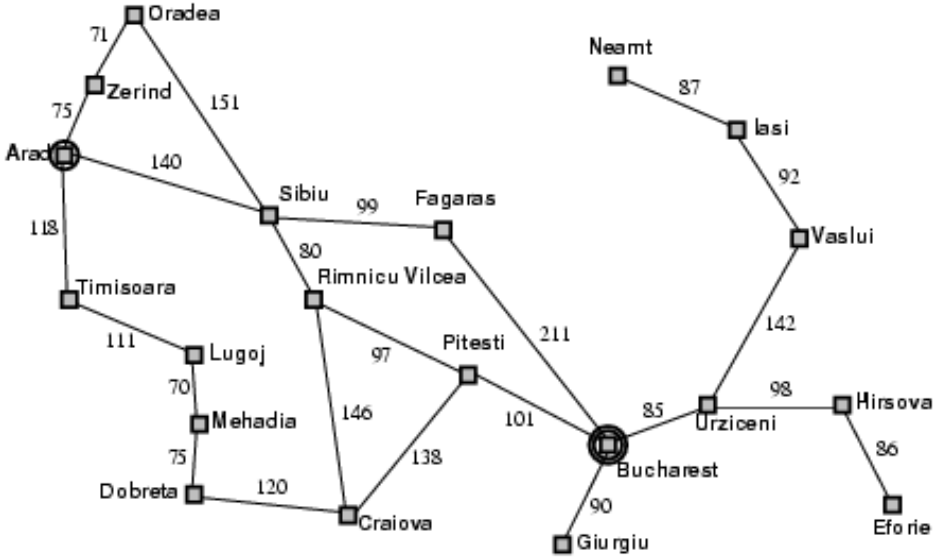


Example: Romania

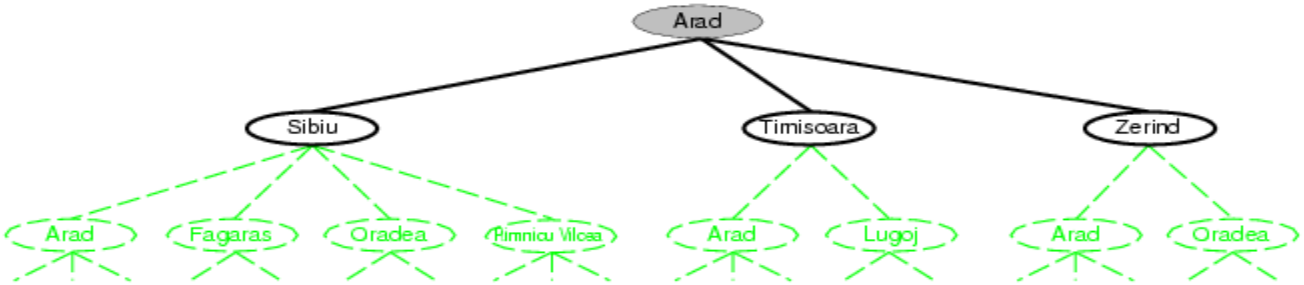


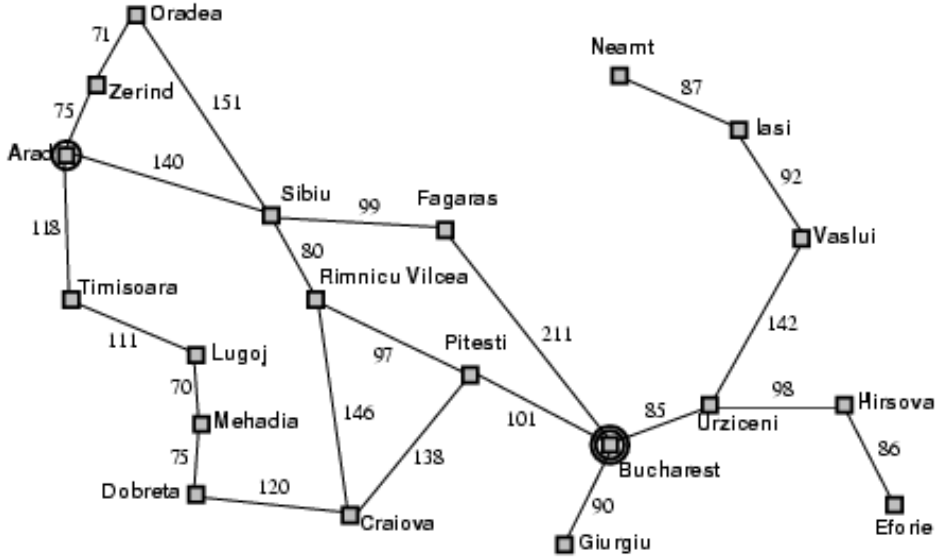
Tree search example



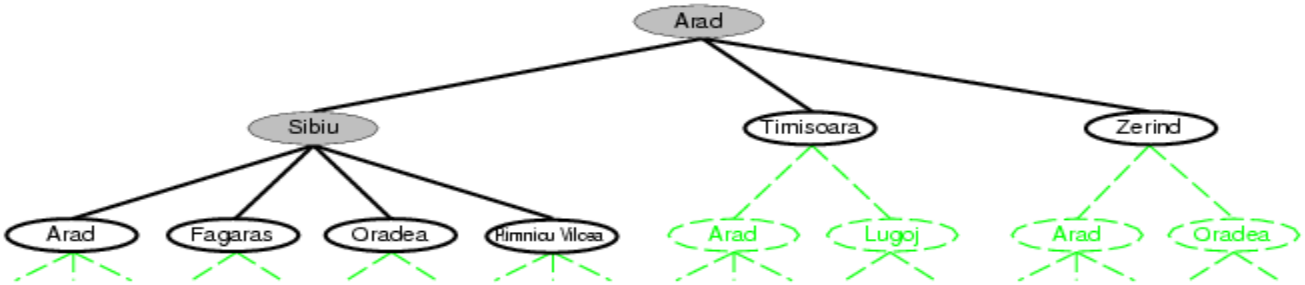


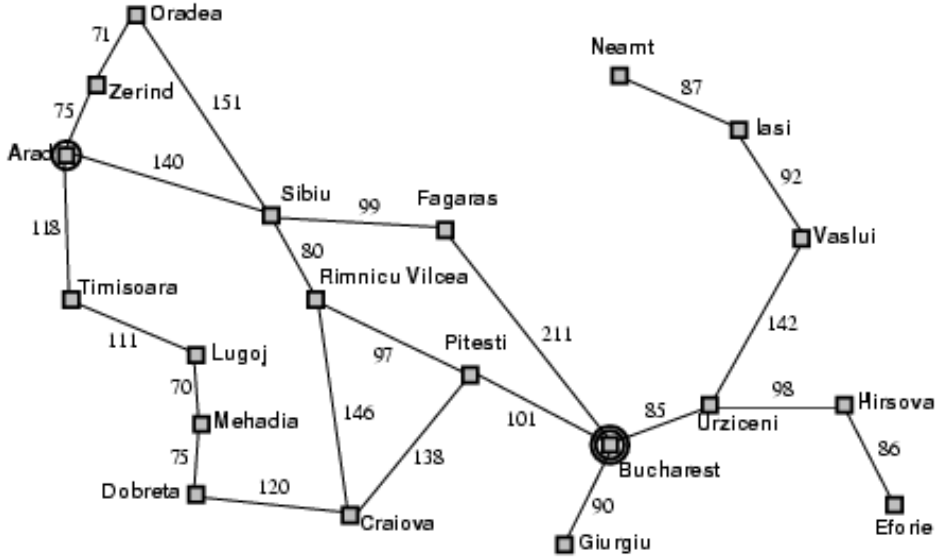
Tree search example



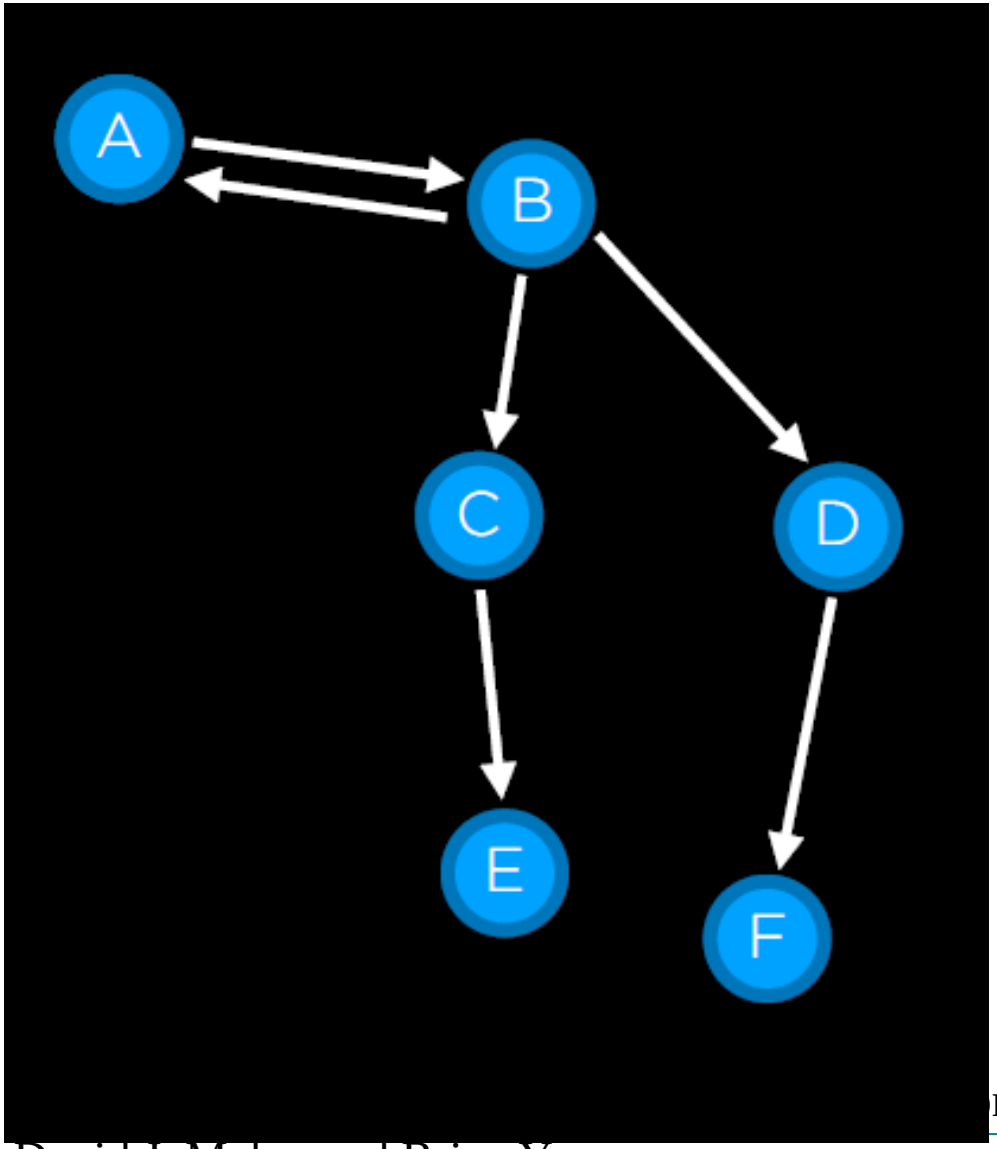


Tree search example





What could go wrong?

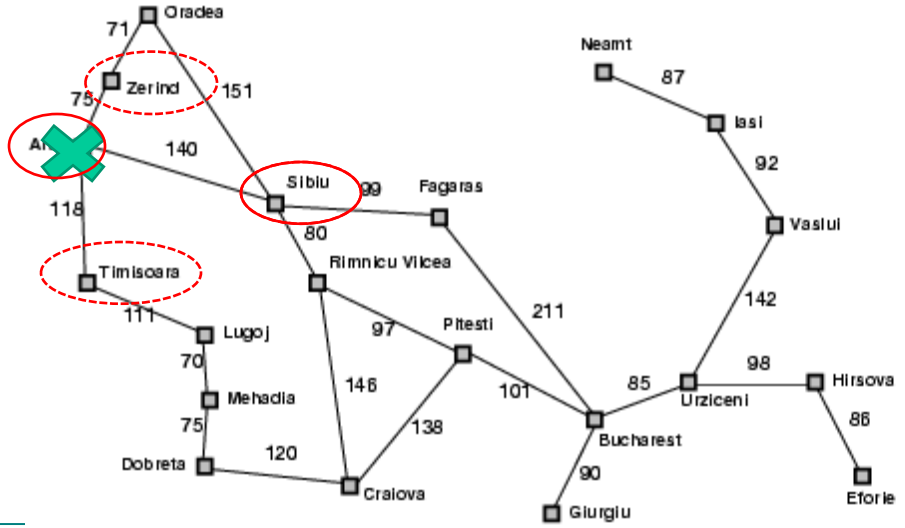


Revised Approach

- Start with a **frontier** that contains the initial state.
- Start with an empty **explored set**.
- Repeat:
  - If the frontier is empty, then no solution.
  - Remove a node from the frontier.
  - If node contains goal state, return the solution.
  - Add the node to the explored set.
  - **Expand** node, add resulting nodes to the frontier if they aren't already in the frontier or the explored set.

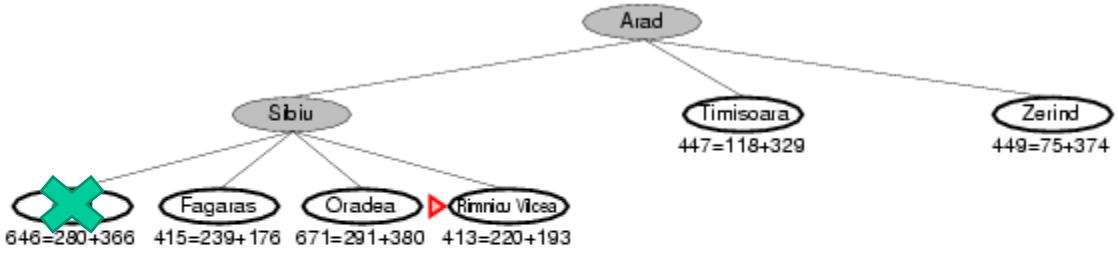
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

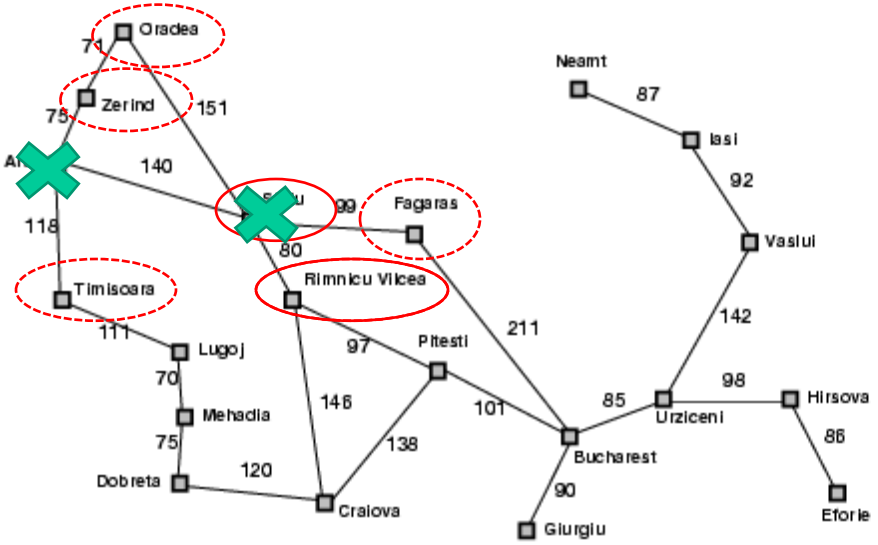
Search without repeated states



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

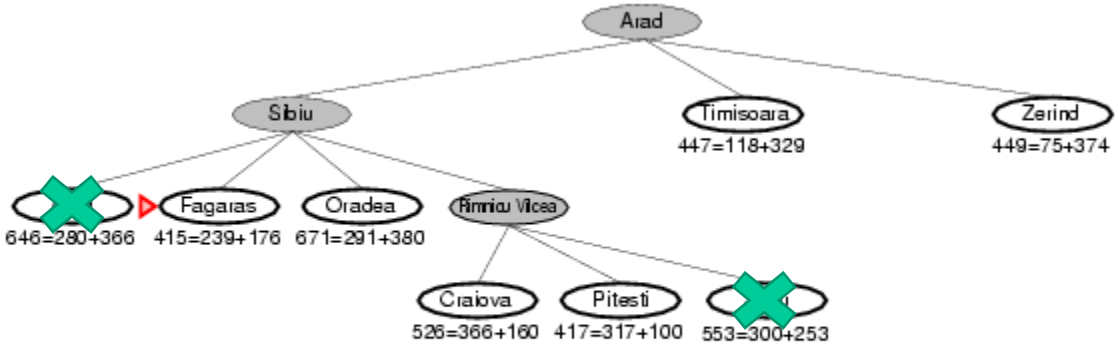
Search without repeated states

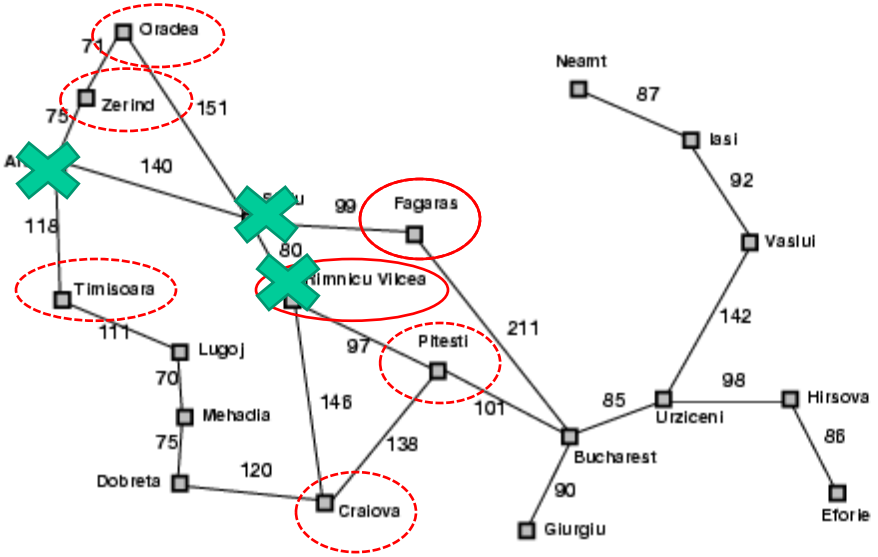




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

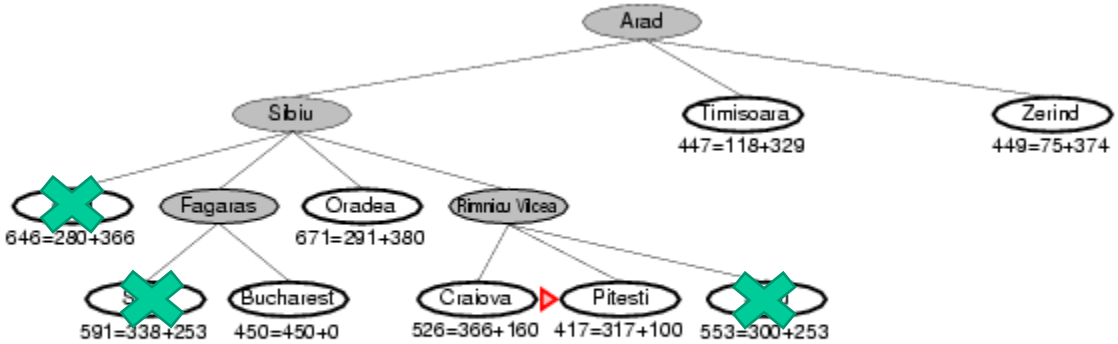
Search without repeated states

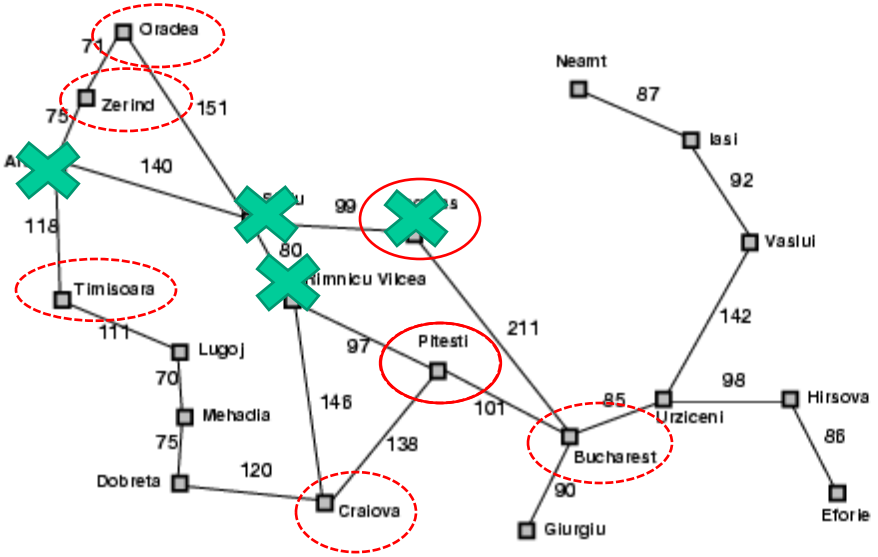




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

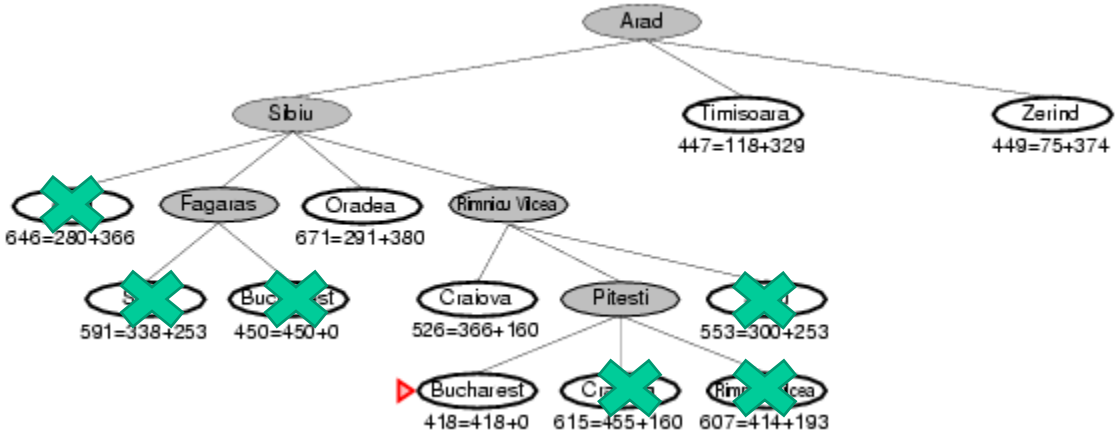
Search without repeated states

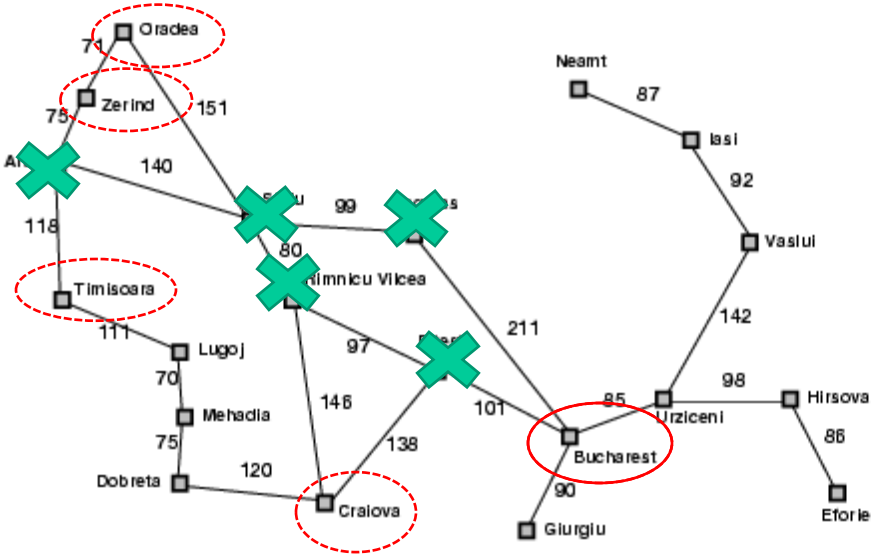




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

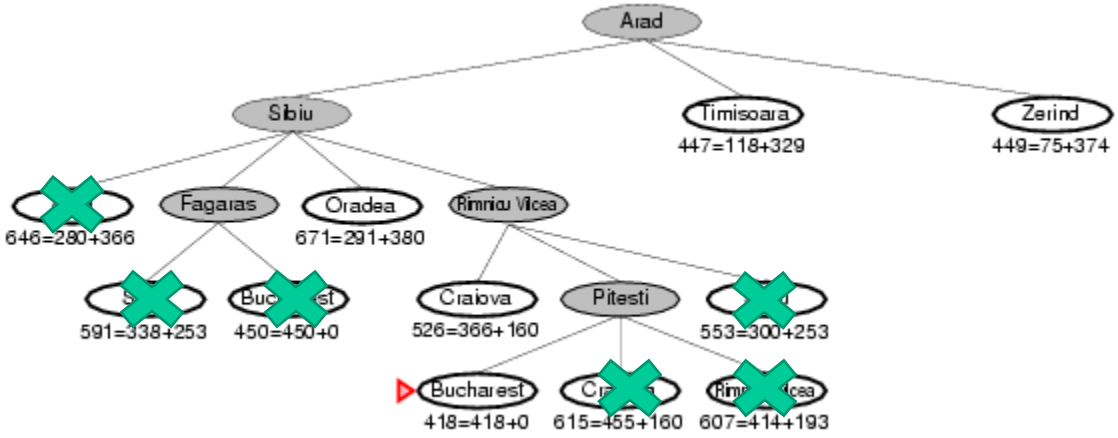
Search without repeated states

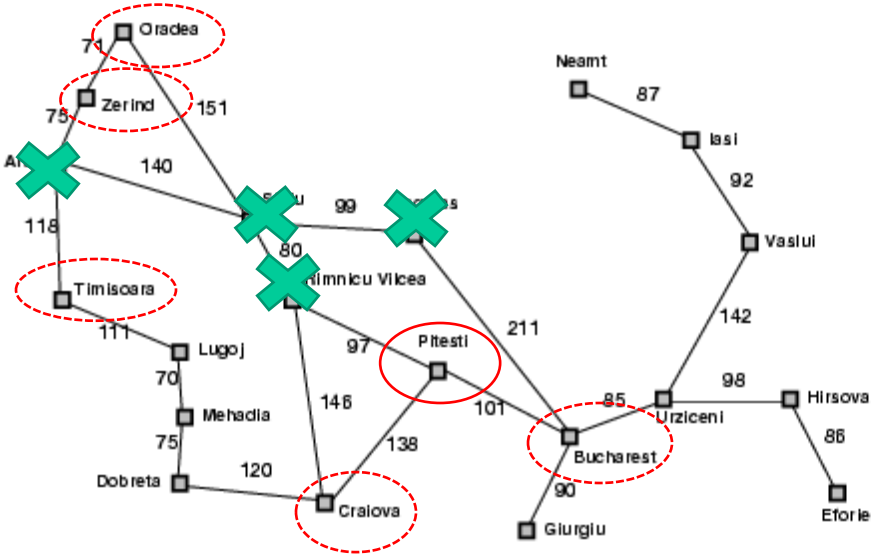




Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Search without repeated states

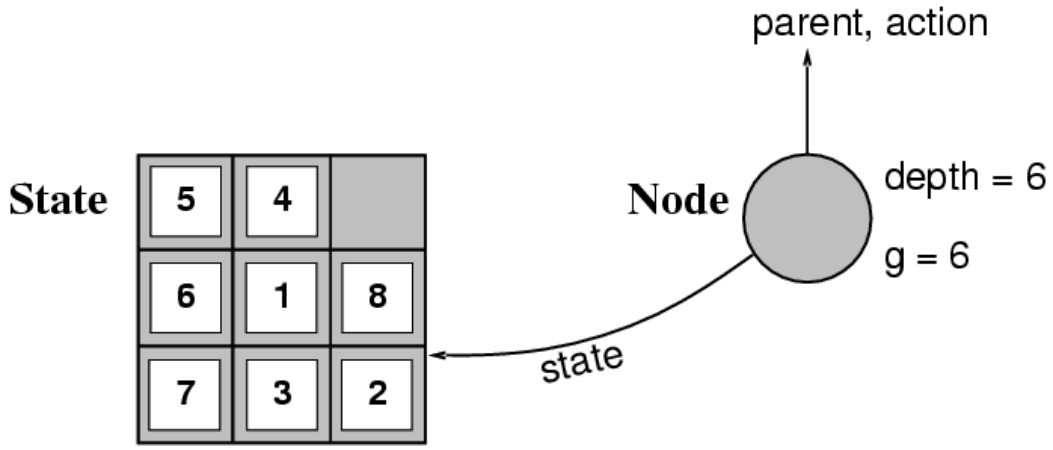




Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Implementation: states vs. nodes

- A state is a (representation of) a physical configuration
- A node is a data structure constituting part of a search tree includes state, parent node, action, path cost $g(x)$, depth



Implementation: general tree search

- **Fringe:** the collection of nodes that have been generated but not yet been expanded
- Each element of a fringe is a leaf node, a node with no successors
- **Search strategy:** a function that selects the next node to be expanded from fringe
- We assume that the collection of nodes is implemented as a queue
- The operations on the queue are:
  - Make-queue(queue)
  - Empty?(queue)
  - first(queue)
  - remove-first(queue)
  - insert(element, queue)
  - insert-all(elements, queue)

Simple search algorithms - revisited

- A search node is a path from state X to the start state (e.g. X B A S)
- The state of a search node is the most recent state of the path (e.g. X)
- Let Q be a list of search nodes (e.g. (X B A S) (C B A S)) and S be the start state

- Algorithm

1. Initialize Q with search node (S) as only entry, set Visited = (S)
2. If Q is empty, fail. Else pick some search node N from Q
3. If state(N) is a goal, return N (we have reached the goal)
4. Otherwise remove N from Q
5. Find all the children of state(N) not in visited and create all the one-step extensions of N to each descendant
6. Add the extended paths to Q, add children of state(N) to Visited
7. Go to step 2

- Critical decisions
  - Step2: picking N from Q
  - Step 6: adding extensions of N to Q

Examples: Simple search strategies

- Depth first search
  - Pick first element of Q
  - Add path extensions to front of Q
- Breadth first search
  - Pick first element of Q
  - Add path extensions to the end of Q

Visited versus expanded

- **Visited:** a state M is first visited when a path to M first gets added to Q. In general, a state is said to have been visited if it has ever shown up in a search node in Q. The intuition is that we have briefly visited them to place them on Q, but we have not yet examined them carefully

- **Expanded:** a state M is expanded when it is the state of a search node that is pulled off of Q. At that point, the descendants of M are visited and the path that led to M is extended to the eligible descendants. In principle, a state may be expanded multiple times. We sometimes refer to the search node that led to M as being expanded. However, once a node is expanded, we are done with it, we will not need to expand it again. In fact, we discard it from Q

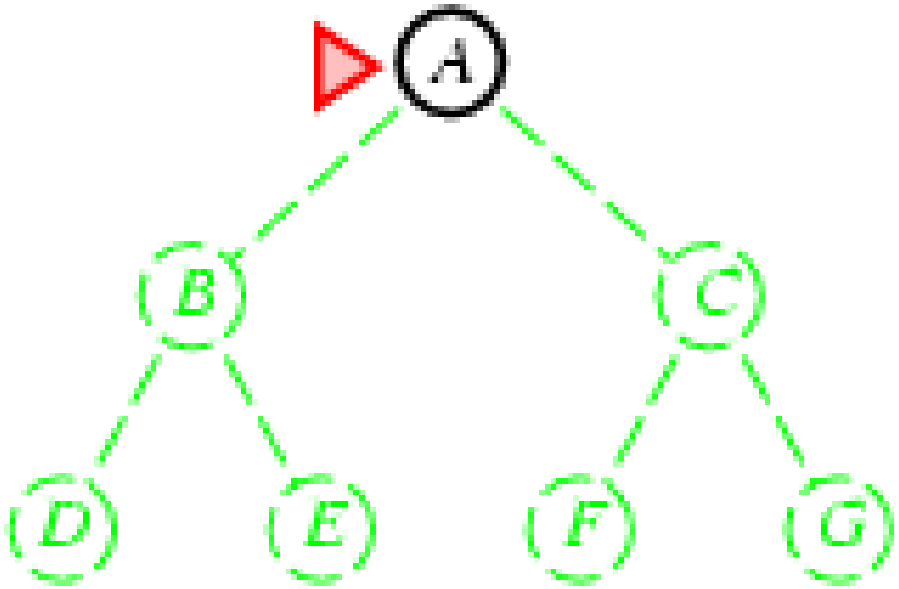
Testing for the goal

- This algorithm stops (in step 3) when state(N) = G or, in general when state(N) satisfies the goal test
- We could have performed the test in step 6 as each extended path is added to Q. This would catch termination earlier
- However, performing the test in step 6 will be incorrect for the optimal searches

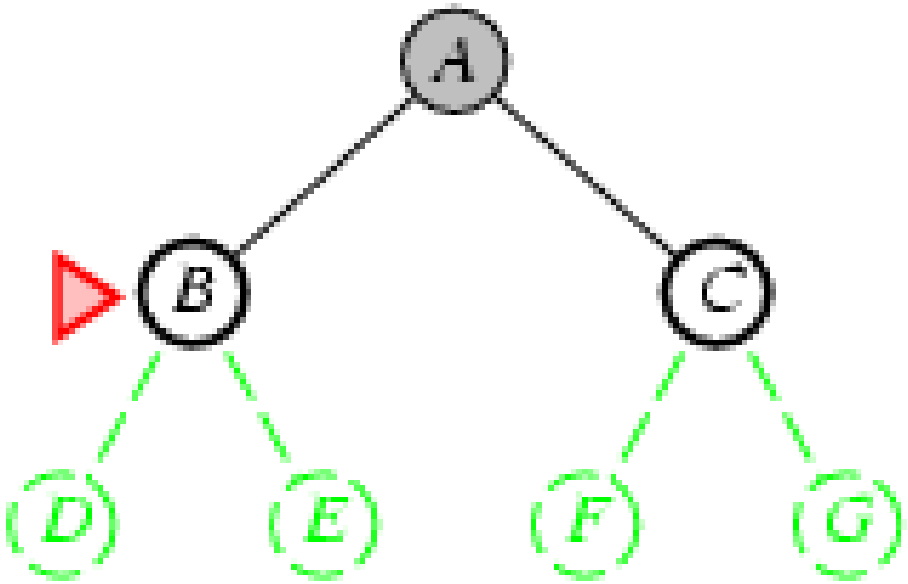
Breadth-first search

- The root node is expanded first, then all the successors of the root node, and their successors and so on
- In general, all the nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded
- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue,
  - the nodes that are visited first will be expanded first
  - All newly generated successors will be put at the end of the queue
  - Shallow nodes are expanded before deeper nodes

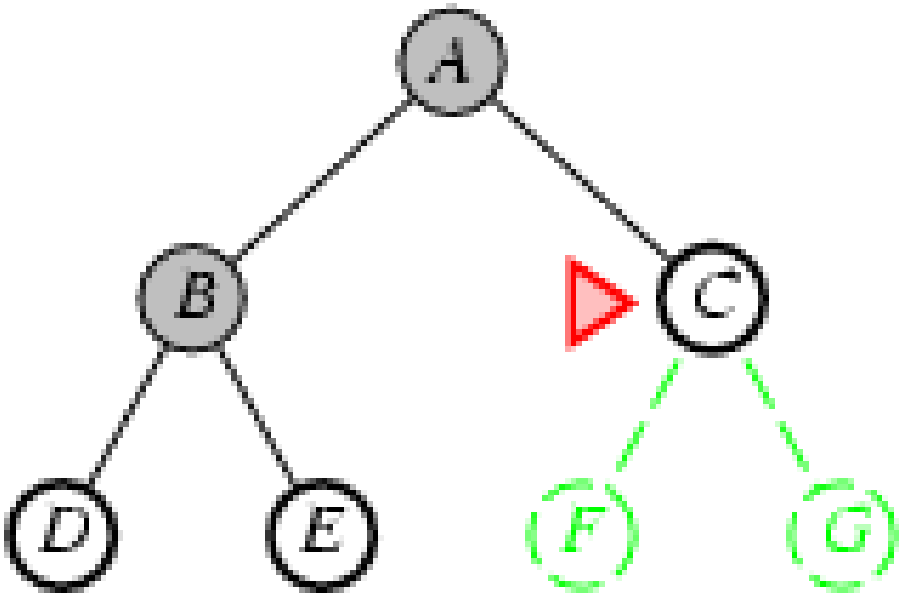
Breadth-first search



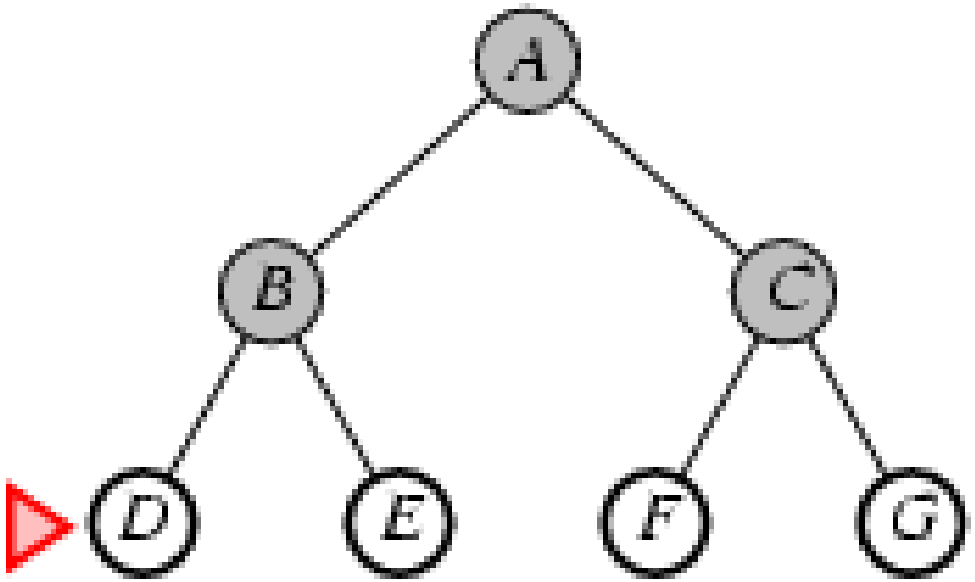
Breadth-first search



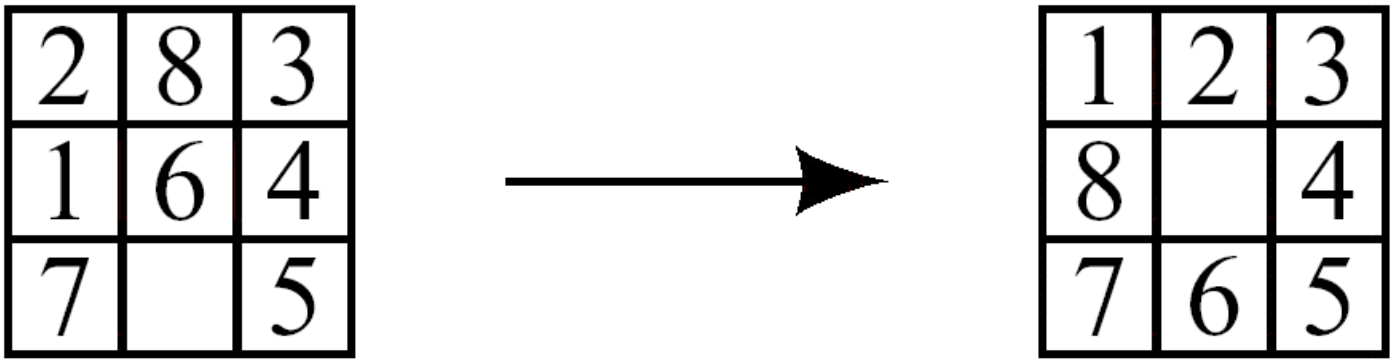
Breadth-first search



Breadth-first search

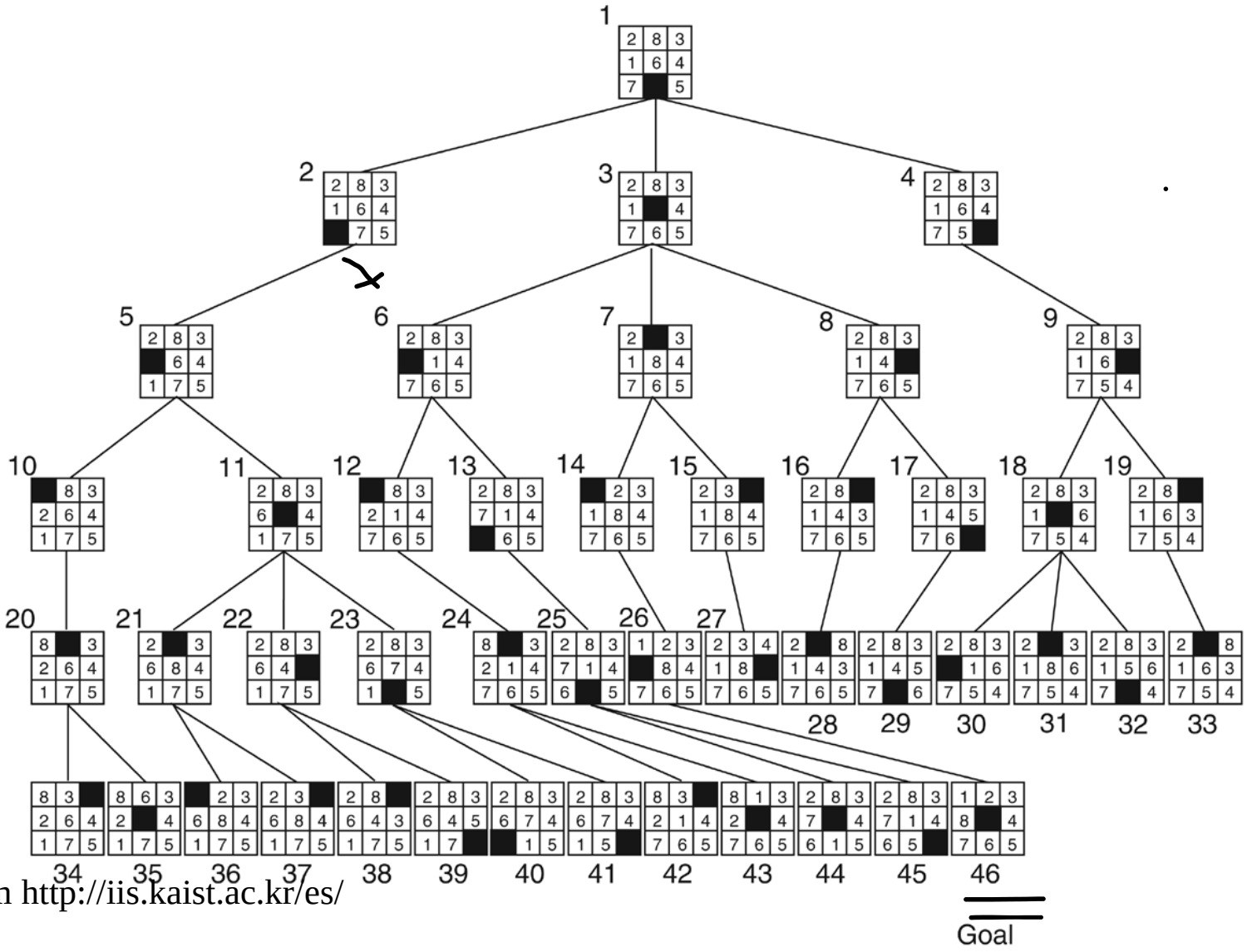


8-puzzle problem



Breadth-first search

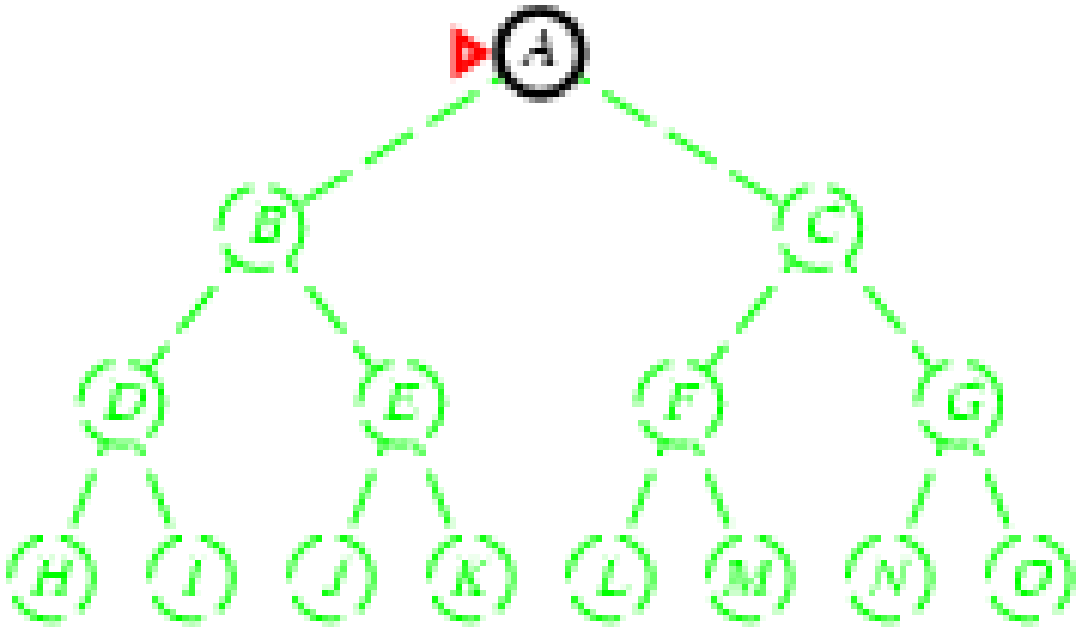
Breadth-first search of the 8-puzzle, showing order in which states were removed from open.



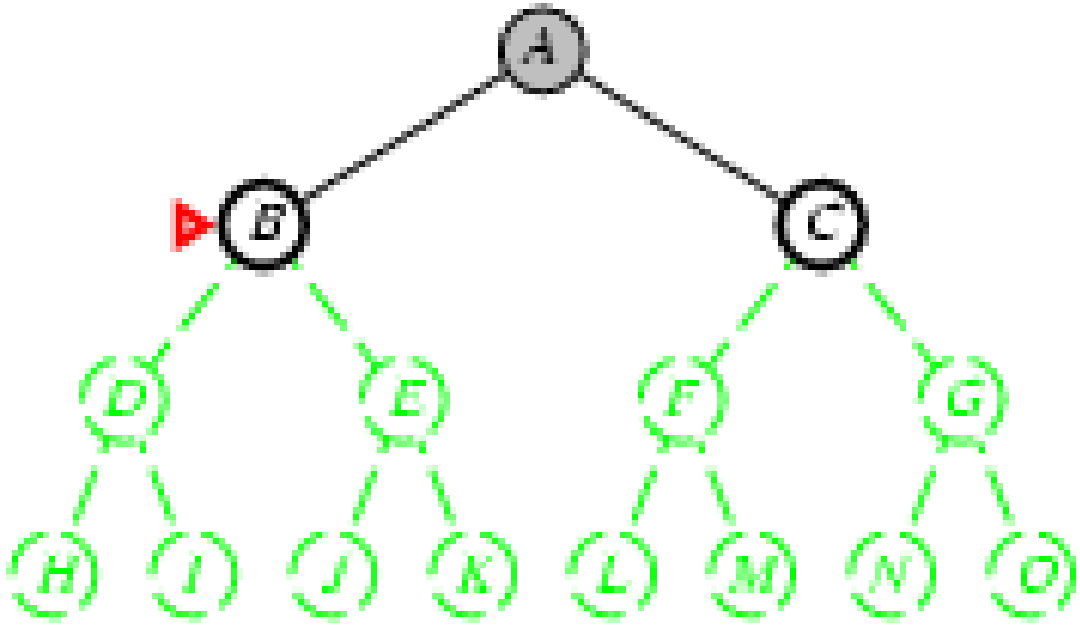
Taken from http://iis.kaist.ac.kr/es/

Depth-first search

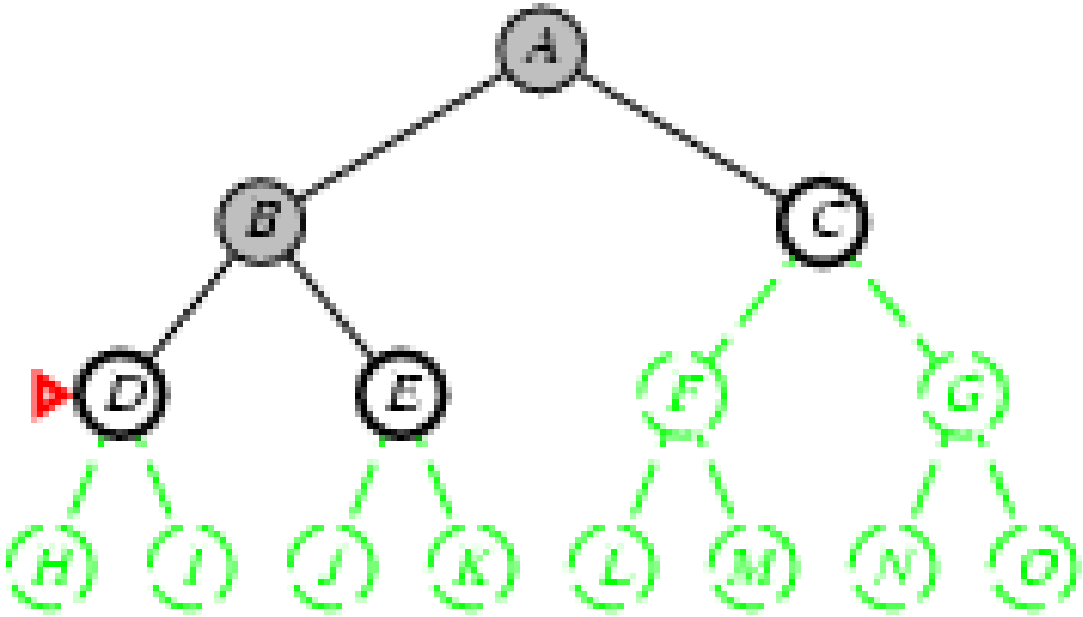
- Expand deepest unexpanded node
- Implementation:
  - *fringe* = LIFO queue (stack) , i.e., put successors at front



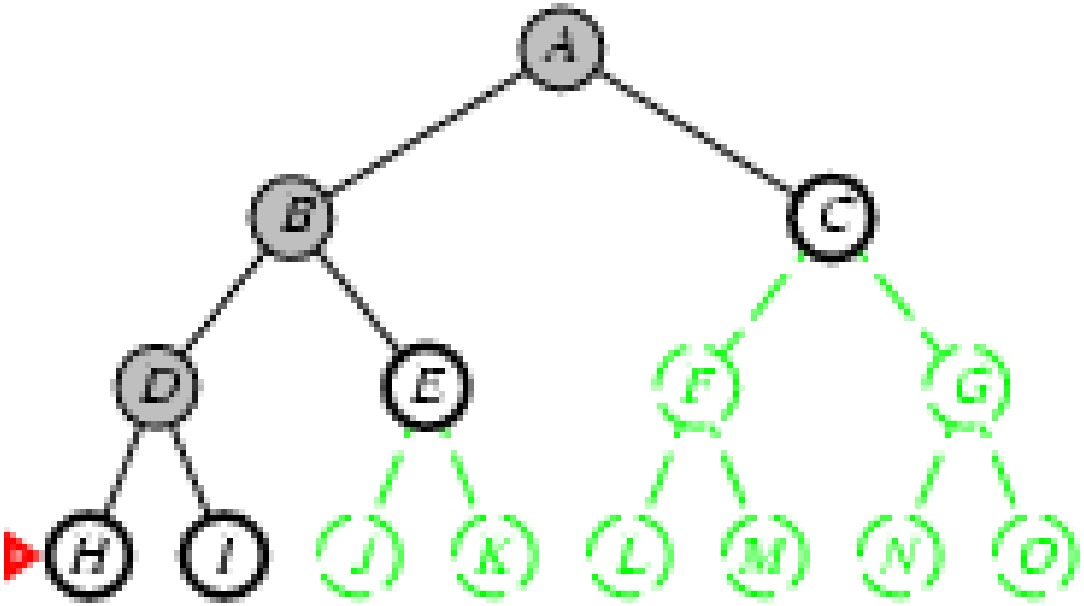
Depth-first search



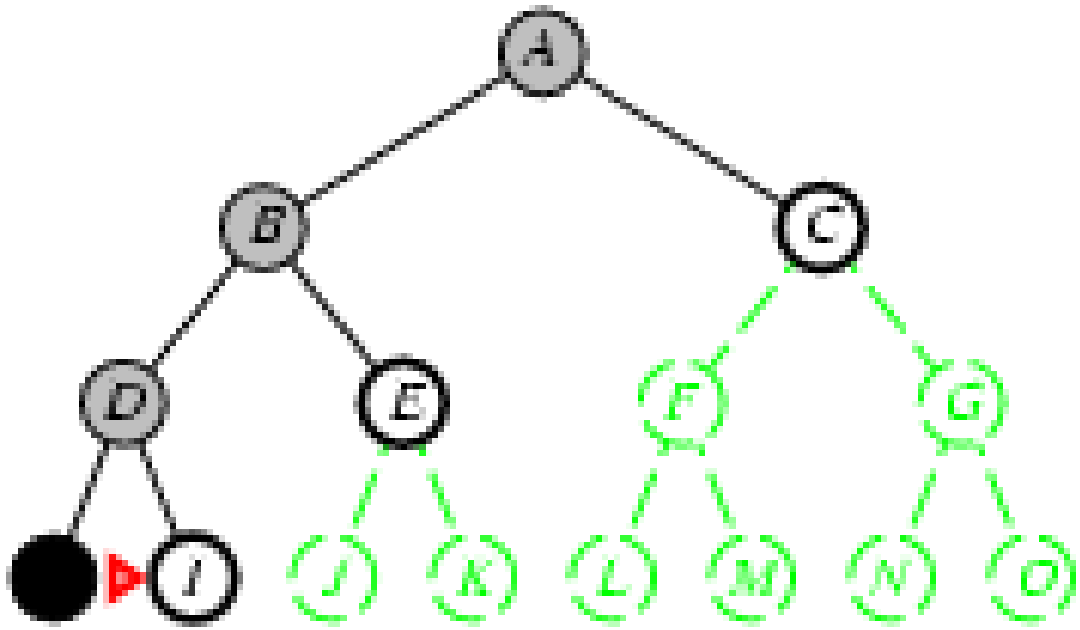
Depth-first search



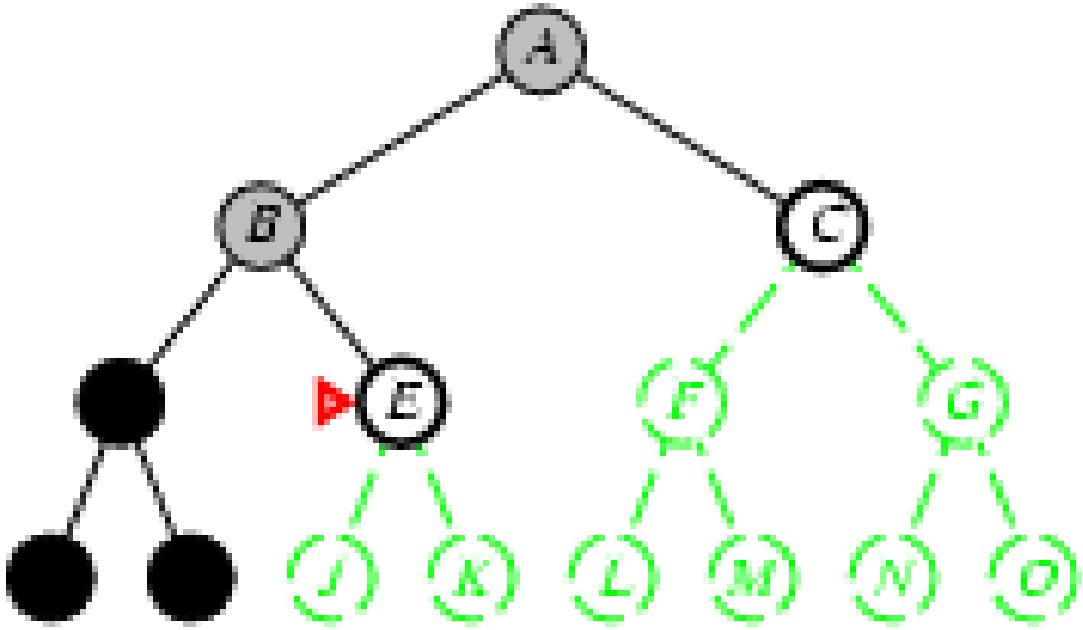
Depth-first search



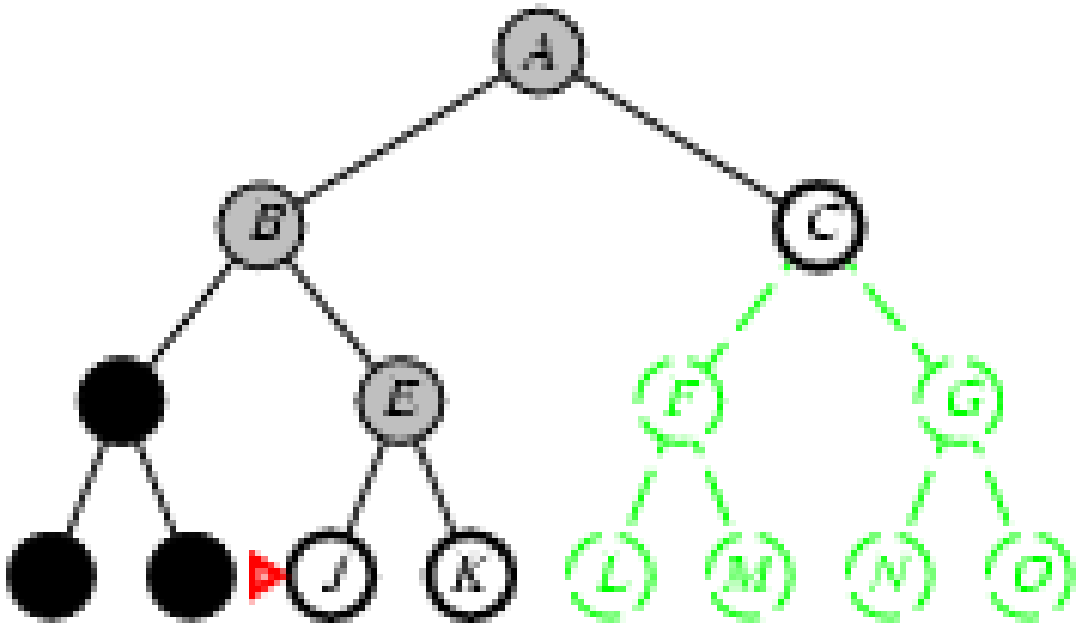
Depth-first search



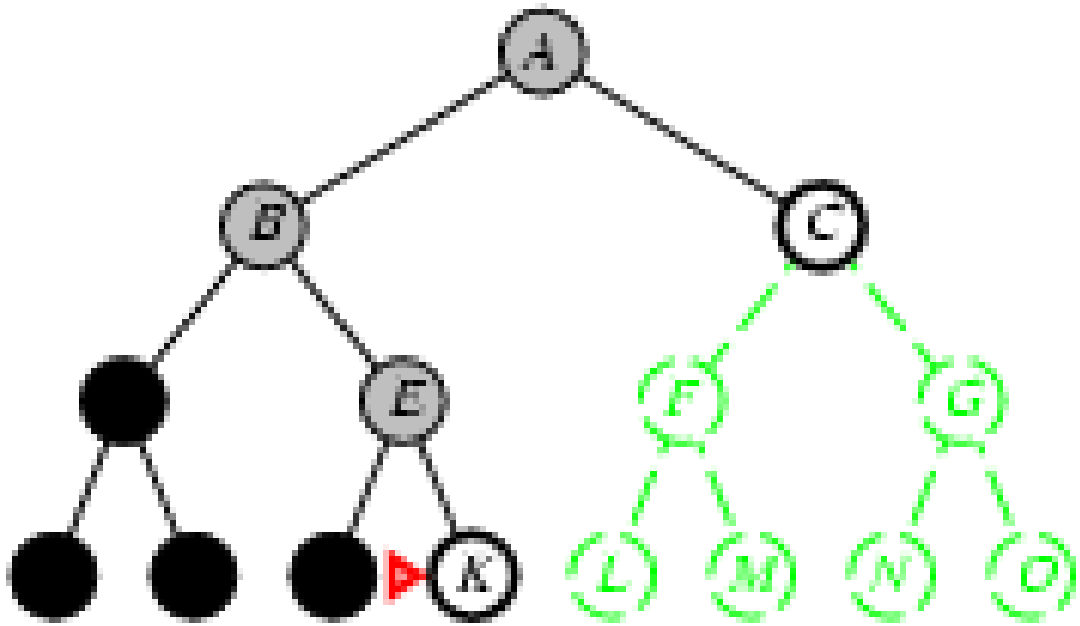
Depth-first search



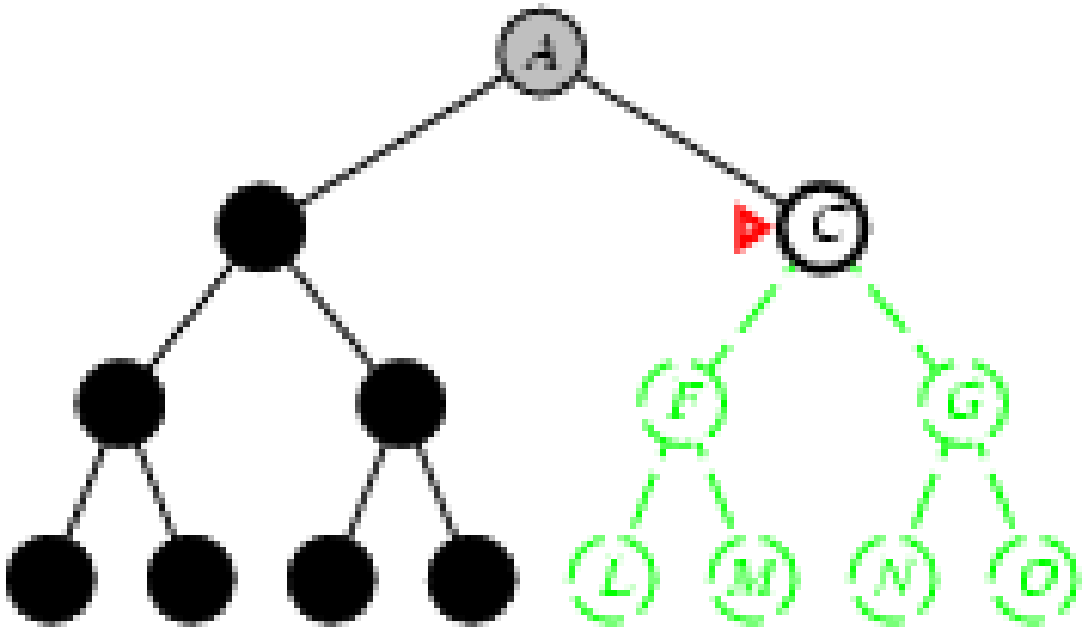
Depth-first search



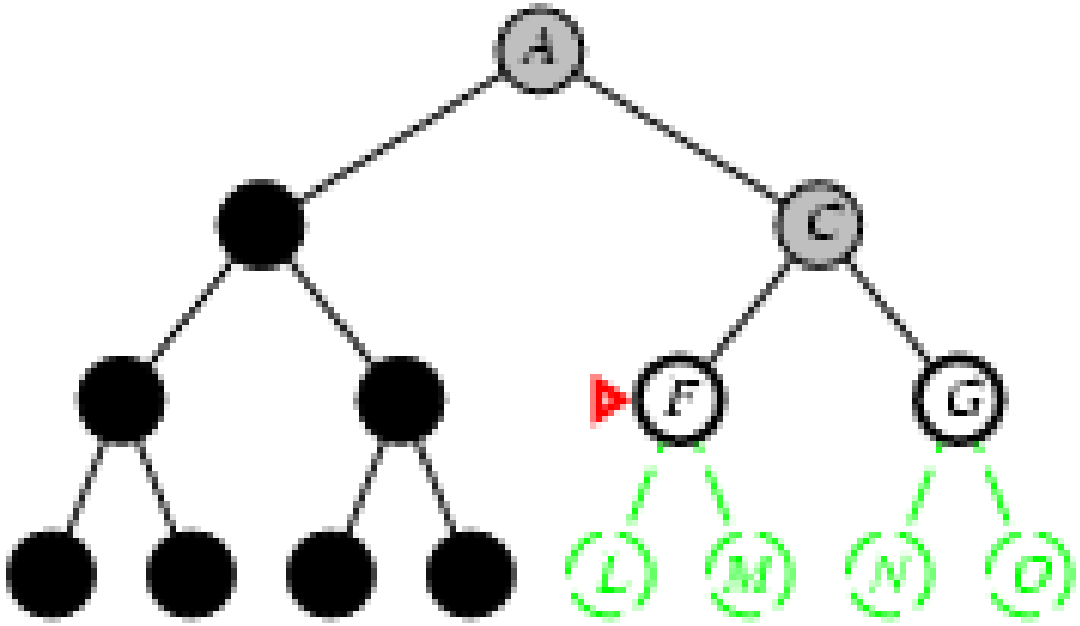
Depth-first search



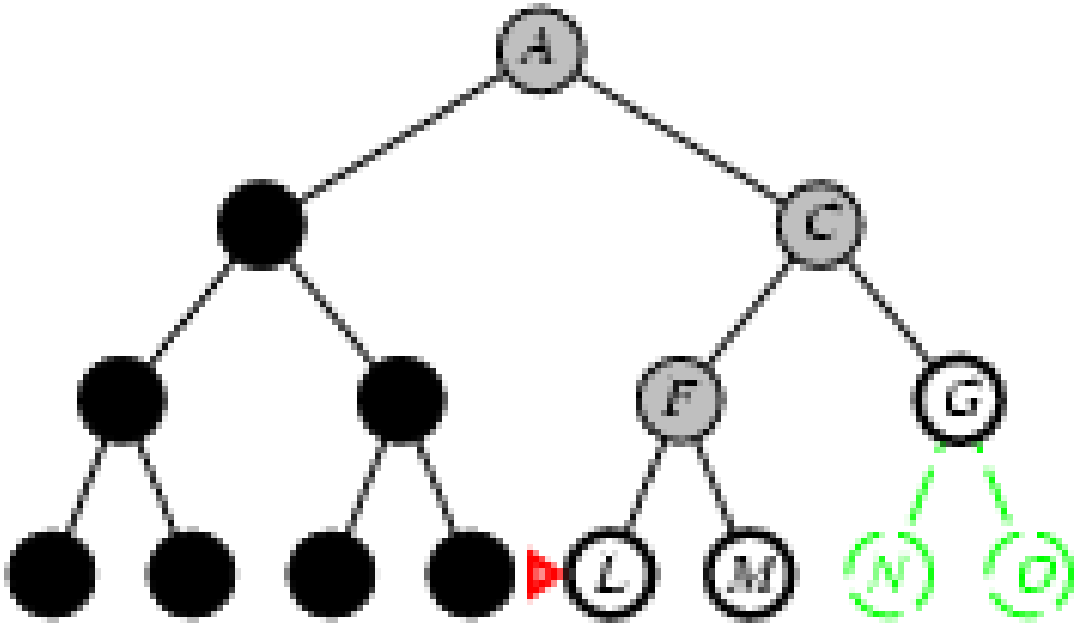
Depth-first search



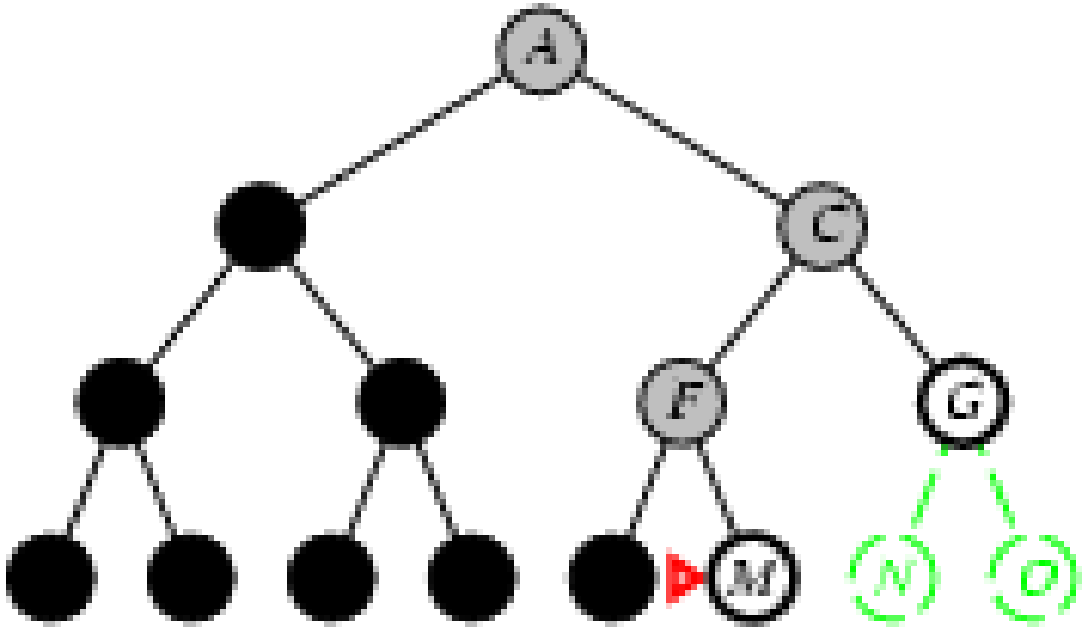
Depth-first search

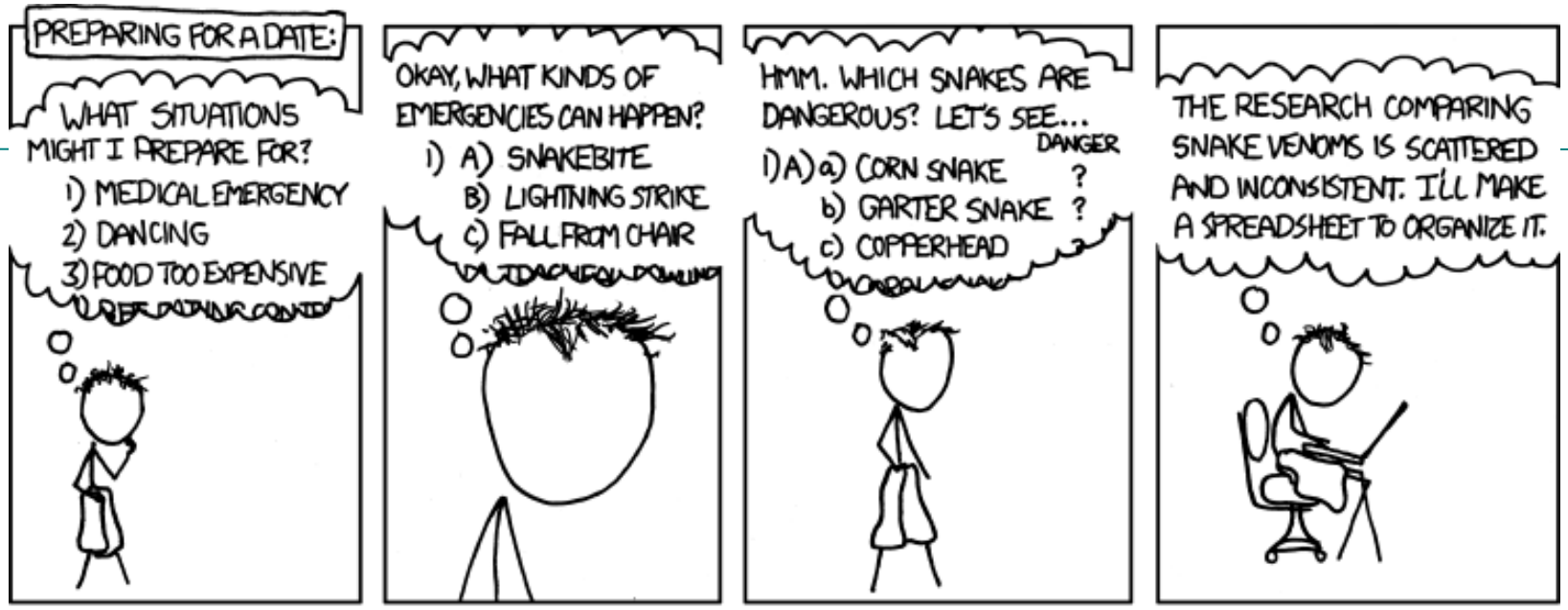


Depth-first search



Depth-first search

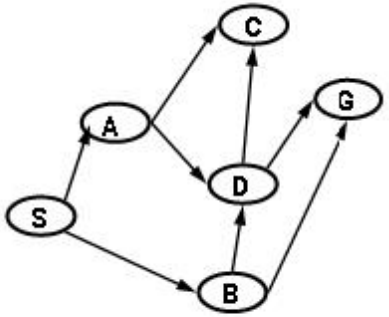




http://xkcd.com/761/

Breadth first search

	Q	Visited
1	(S)	S
2		
3		
4		
5		
6		



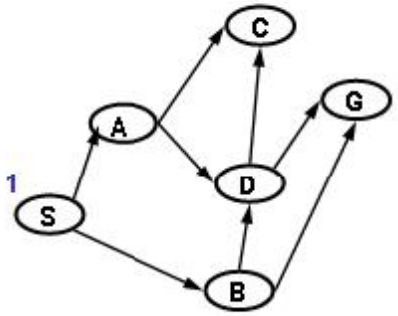
Pick first element of Q; Add path extensions to end of Q

Added paths in blue

We show the paths in reversed order; the node's state is the first entry.

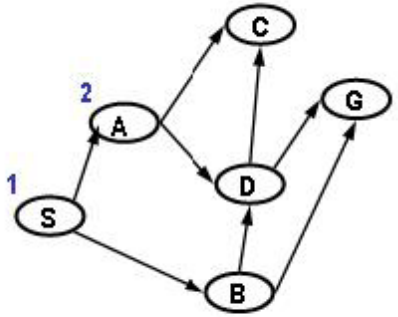
Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3		
4		
5		
6		



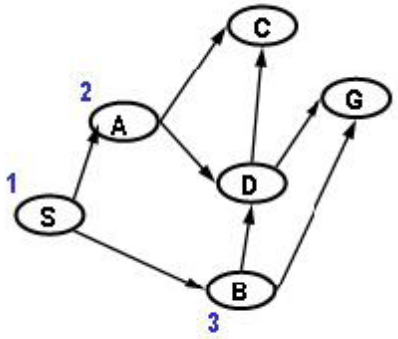
Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4		
5		
6		



Breadth first search

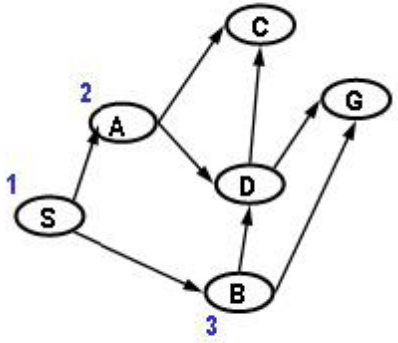
	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5		
6		



* We could have stopped here, when the first path to the goal was generated.

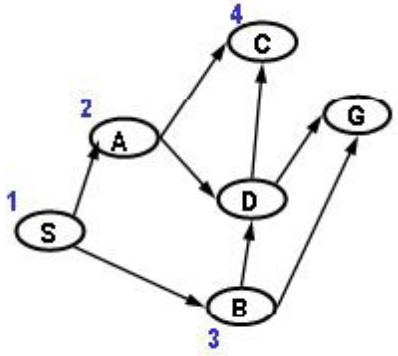
Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5		
6		



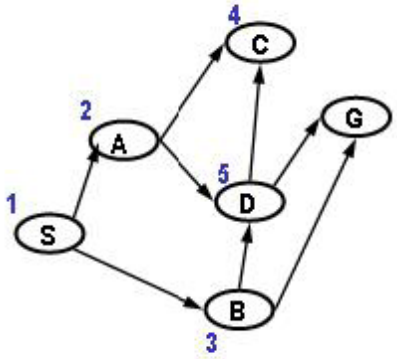
Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6		



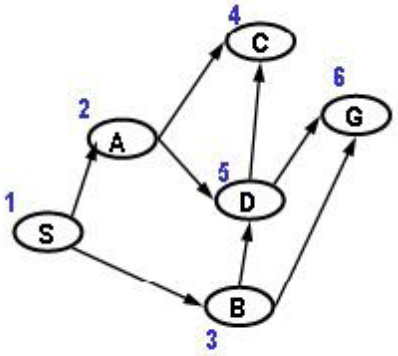
Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6		



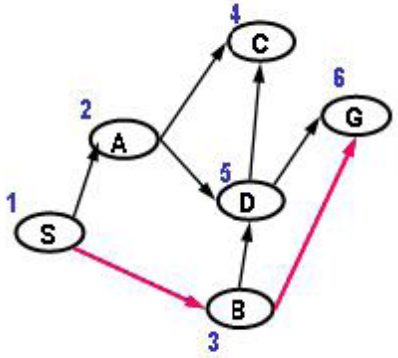
Breadth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6	(G B S)	G,C,D,B,A,S

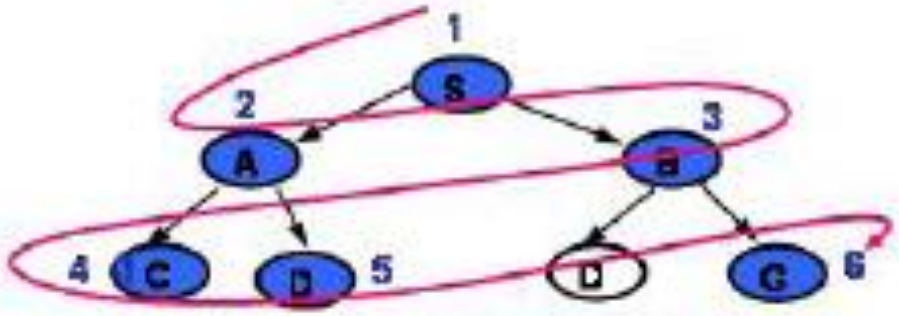


Breadth first search

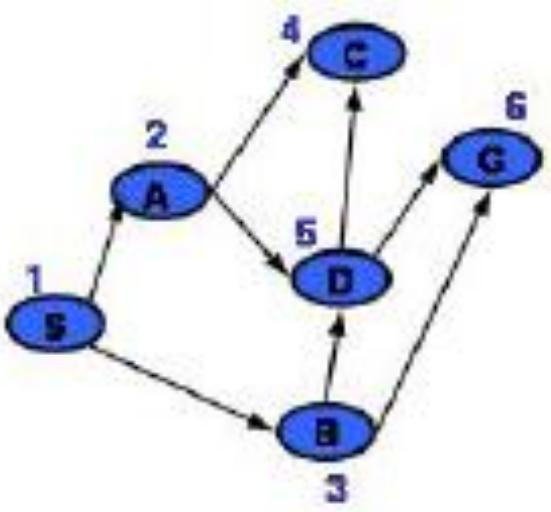
	Q	Visited
1	(S)	S
2	(A S) (B S)	A,B,S
3	(B S) (C A S) (D A S)	C,D,B,A,S
4	(C A S) (D A S) (G B S)*	G,C,D,B,A,S
5	(D A S) (G B S)	G,C,D,B,A,S
6	(G B S)	G,C,D,B,A,S



Breadth first search



NB: D is not visited again



Numbers indicate order pulled off of Q (expanded)

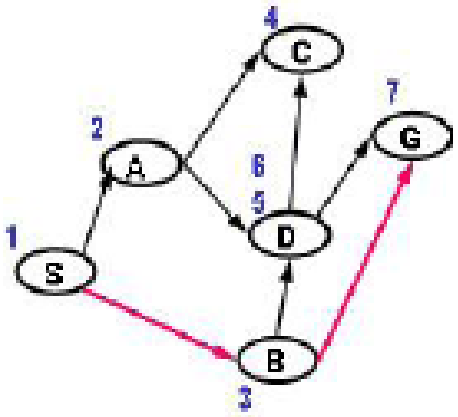
Dark blue fill = Visited & Expanded

Light gray fill = Visited

Breadth first (Without visited list)

Pick first element of Q; Add path extensions to end of Q

	Q
1	(S)
2	(A S) (B S)
3	(B S) (C A S) (D A S)
4	(C A S) (D A S) (D B S) (G B S)*
5	(D A S) (D B S) (G B S)
6	(D B S) (G B S) (C D A S) (G D A S)
7	(G B S) (C D A S) (G D A S) (C D B S) (G D B S)



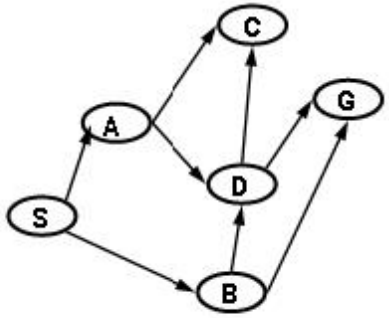
Added paths in blue

We show the paths in reversed order; the node's state is the first entry.

* We could have stopped here, when the first path to the goal was generated.

Depth first search

	Q	Visited
1		
2		
3		
4		
5		



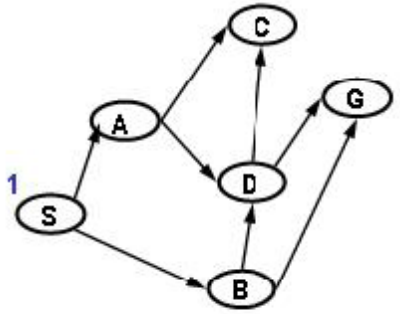
Pick first element of Q; Add path extensions to front of Q

Added paths in blue

We show the paths in reversed order; the node's state is the first entry.

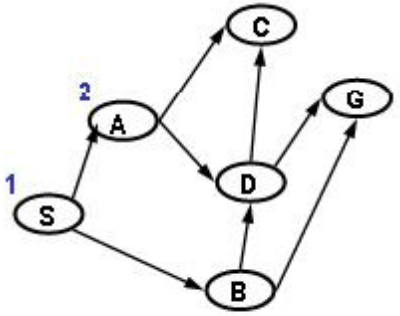
Depth first search

	Q	Visited
1	(S)	S
2		
3		
4		
5		



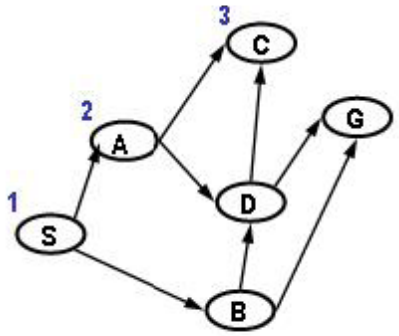
Depth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3		
4		
5		



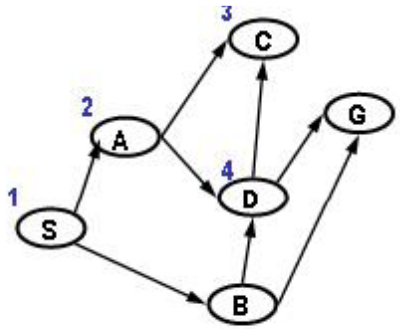
Depth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C,D,B,A,S
4		
5		



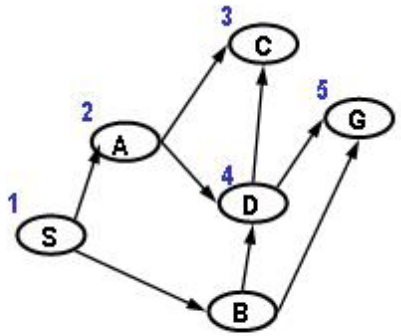
Depth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C,D,B,A,S
4	(D A S) (B S)	C,D,B,A,S
5		



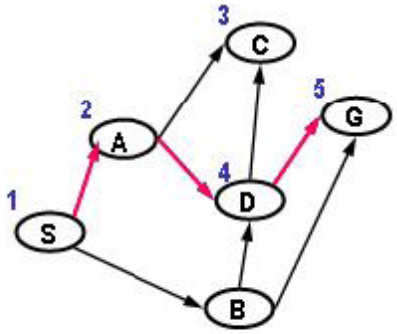
Depth first search

	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C,D,B,A,S
4	(D A S) (B S)	C,D,B,A,S
5	(G D A S) (B S)	G,C,D,B,A,S



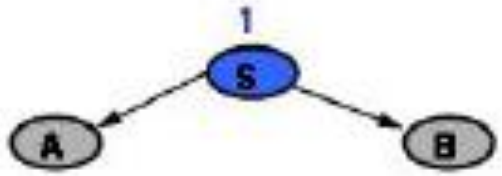
Depth first search

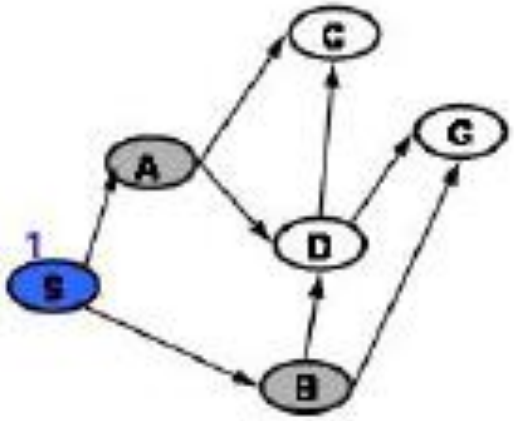
	Q	Visited
1	(S)	S
2	(A S) (B S)	A, B, S
3	(C A S) (D A S) (B S)	C,D,B,A,S
4	(D A S) (B S)	C,D,B,A,S
5	(G D A S) (B S)	G,C,D,B,A,S



Depth-first search

Another (easier?) way to see it





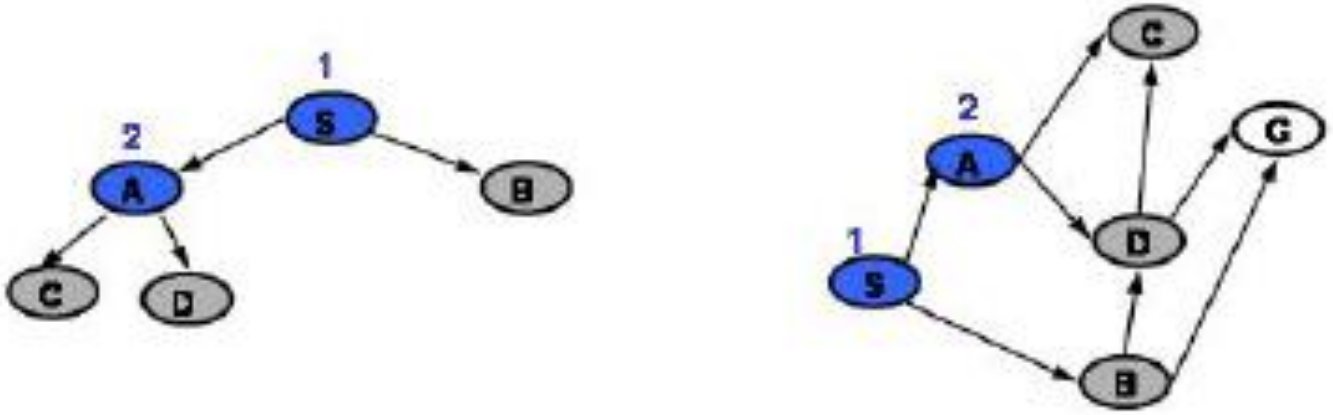
Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

Light gray fill = Visited

Depth-first search

Another (easier?) way to see it

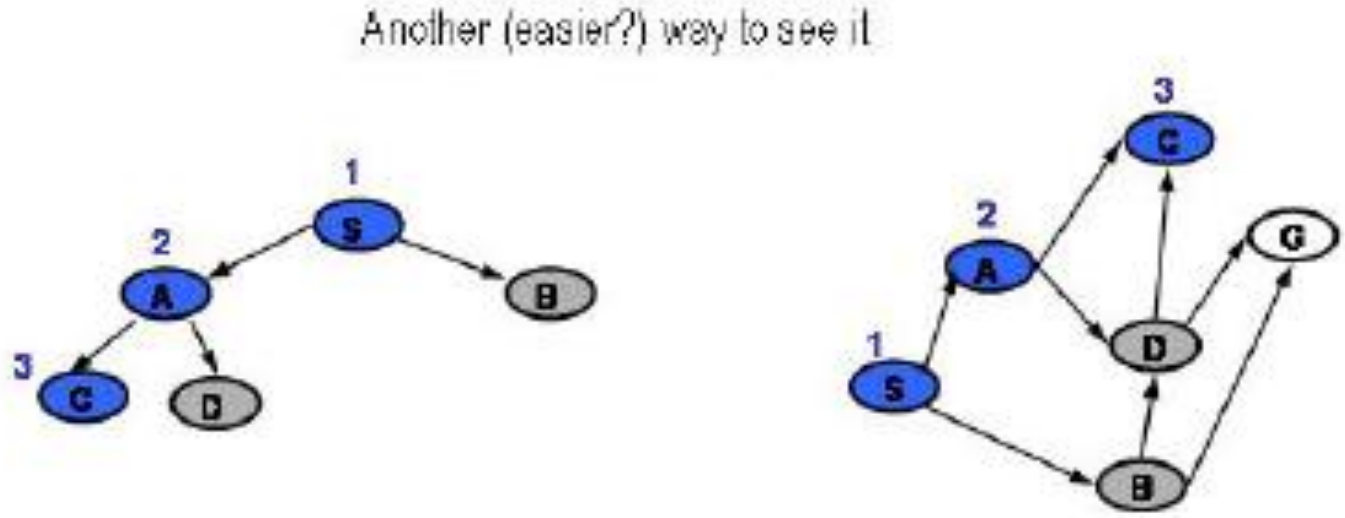


Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

Light gray fill = Visited

Depth-first search

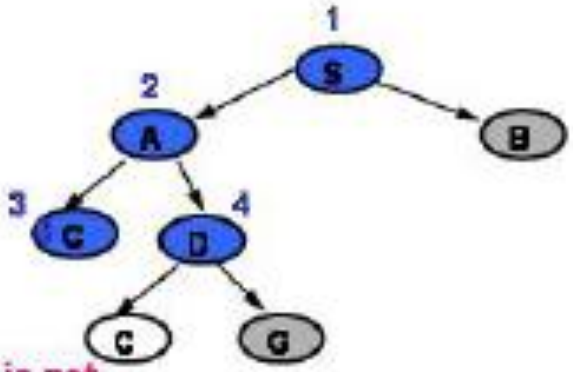


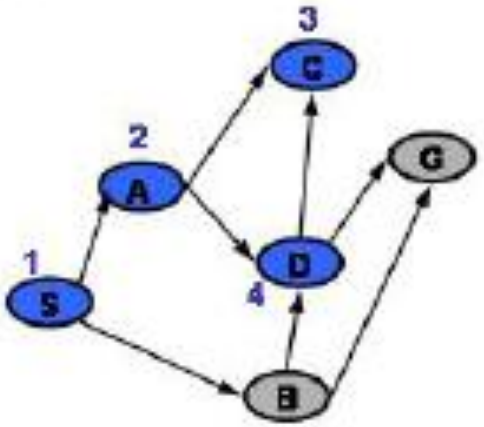
Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded
Light gray fill = Visited

Depth-first search

Another (easier?) way to see it





NB: C is not visited again

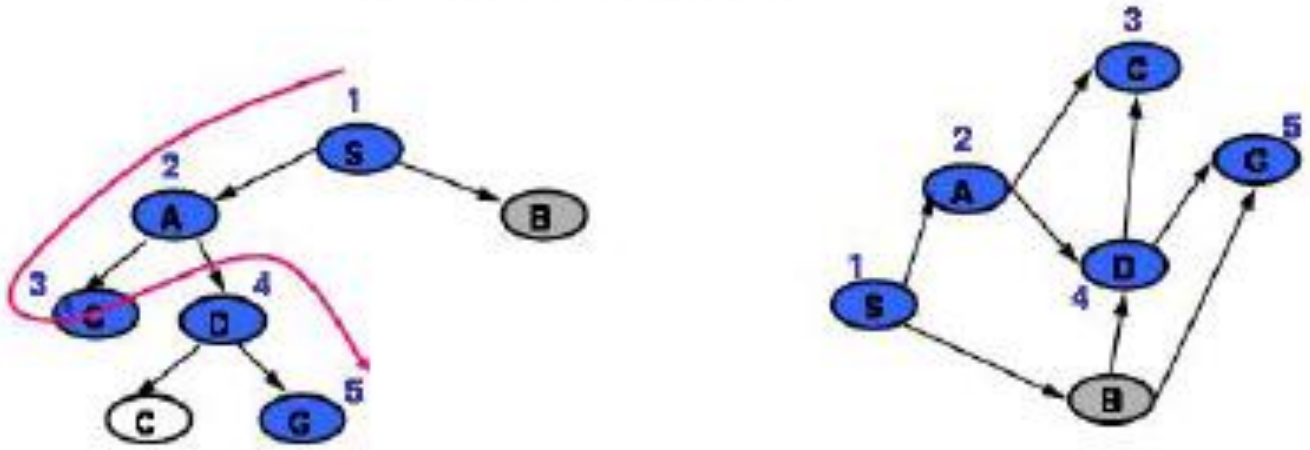
Numbers indicate order pulled off of Q (expanded)

Dark blue fill = Visited & Expanded

Light gray fill = Visited

Depth-first search

Another (easier?) way to see it



Numbers indicate order pulled off of Q (expanded)

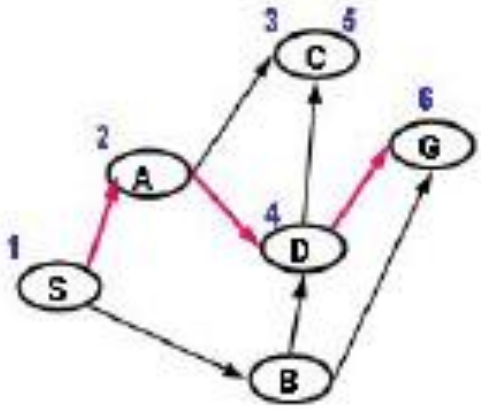
Dark blue fill = Visited & Expanded

Light gray fill = Visited

Depth-first search

Pick first element of Q; Add path extensions to front of Q

	Q
1	(S)
2	(A S) (B S)
3	(C A S) (D A S) (B S)
4	(D A S) (B S)
5	(C D A S) (G D A S) (B S)
6	(G D A S) (B S)

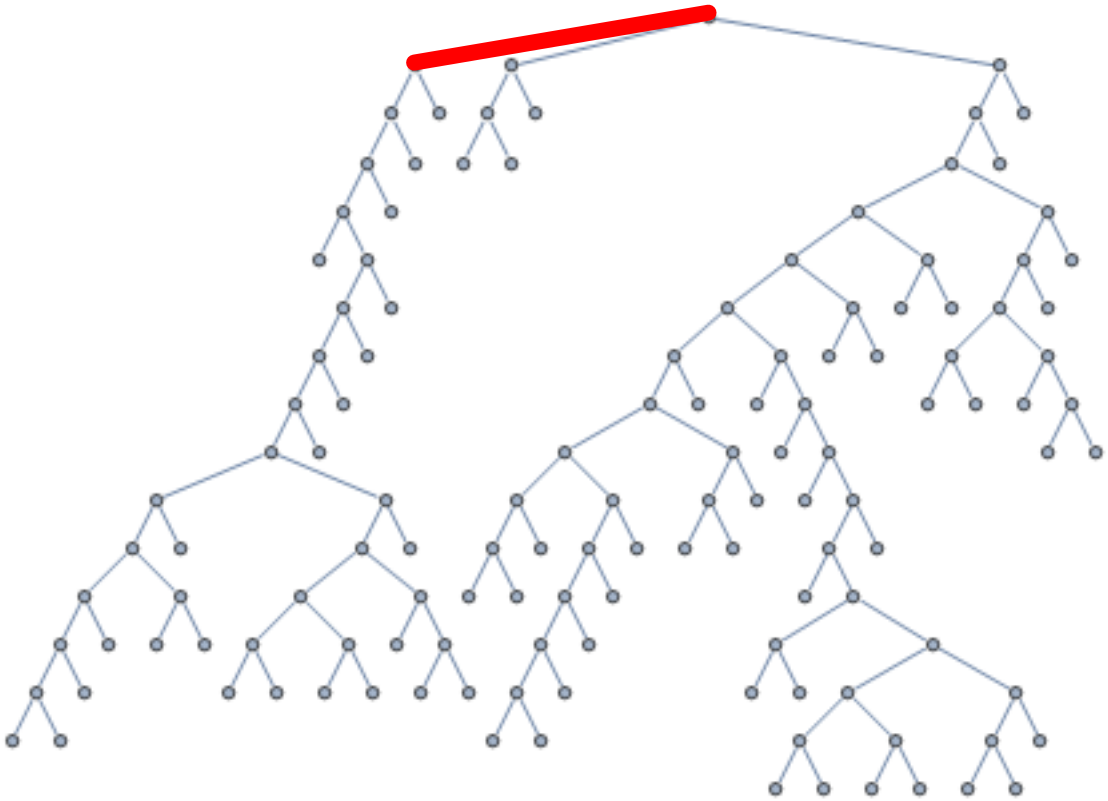


Added paths in blue

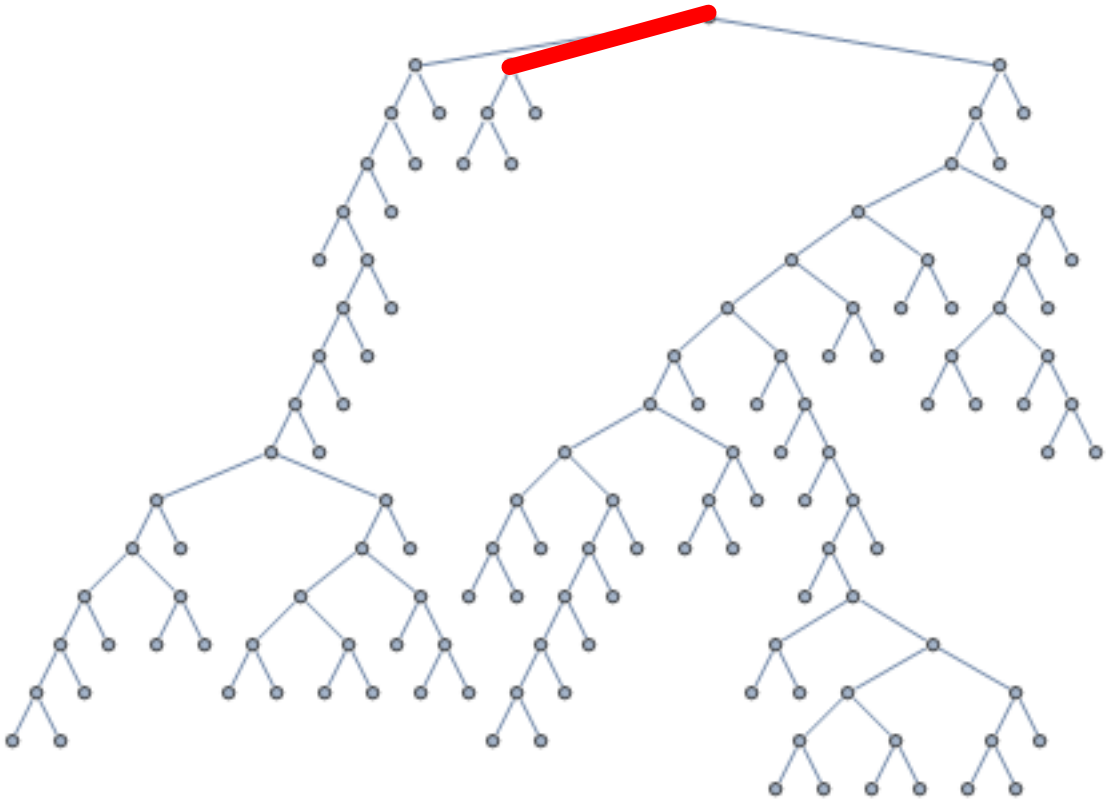
We show the paths in reversed order; the node's state is the first entry.

Do not extend a path to a state if the resulting path would have a loop.

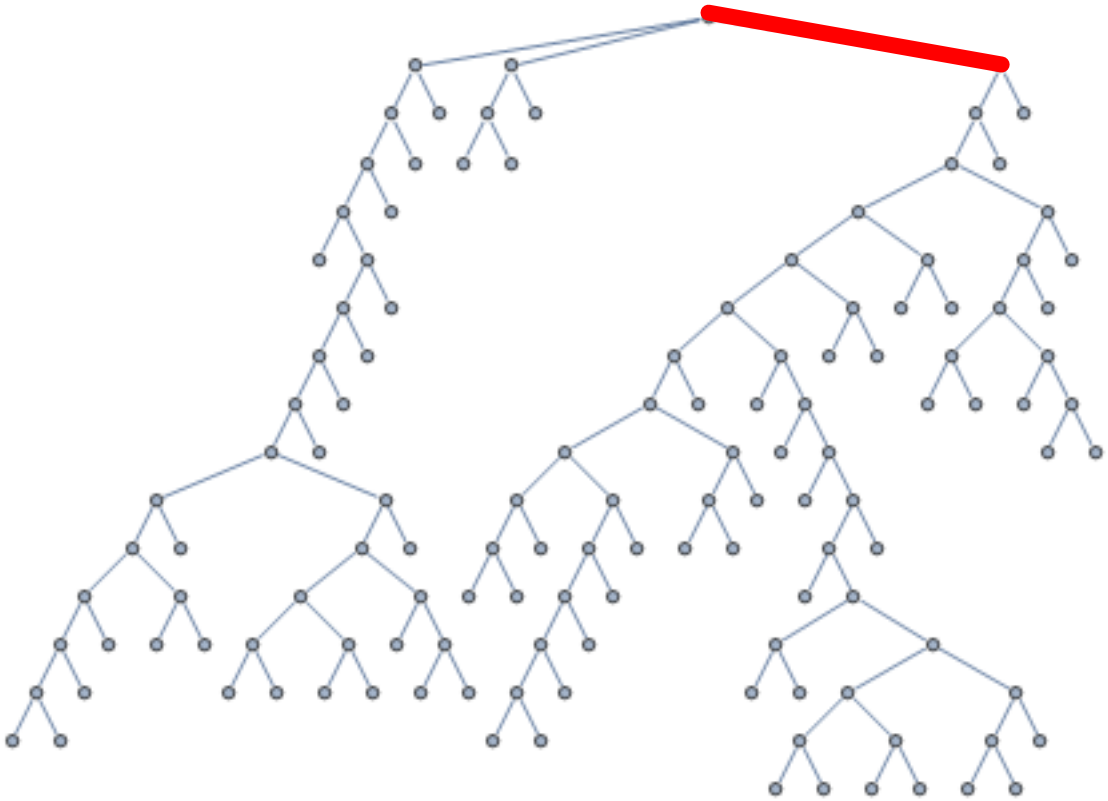
BFS vs. DFS



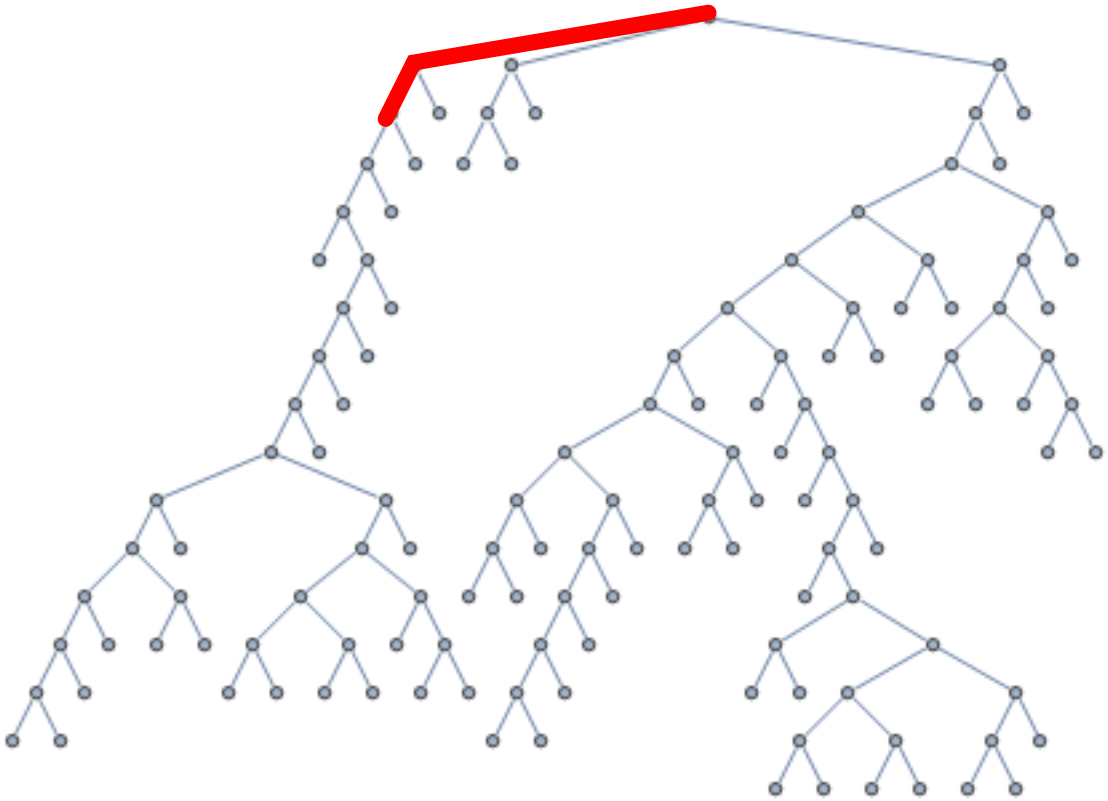
BFS vs. DFS



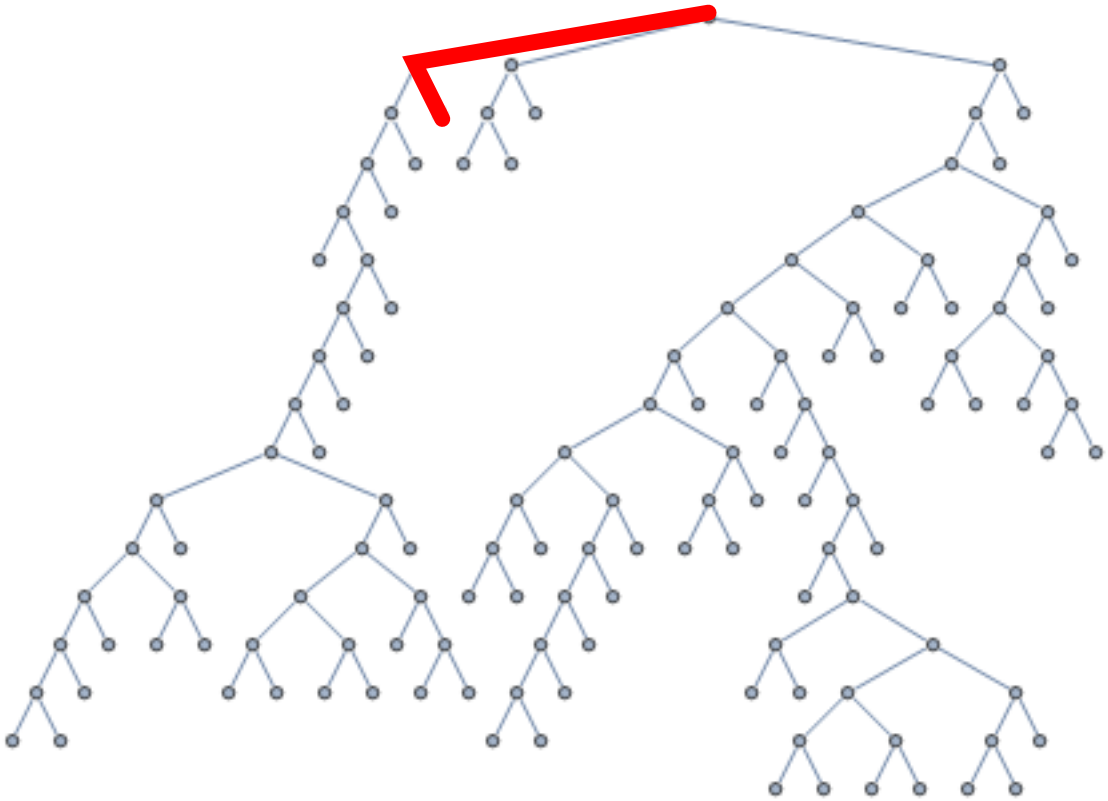
BFS vs. DFS



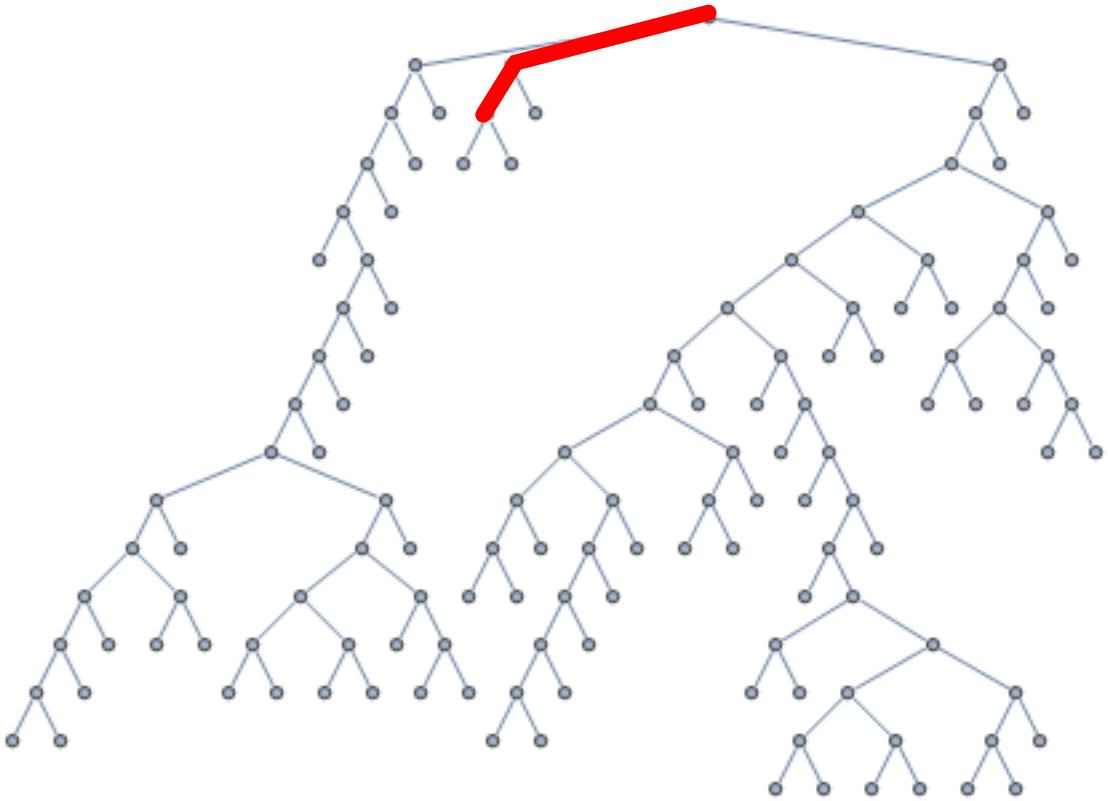
BFS vs. DFS



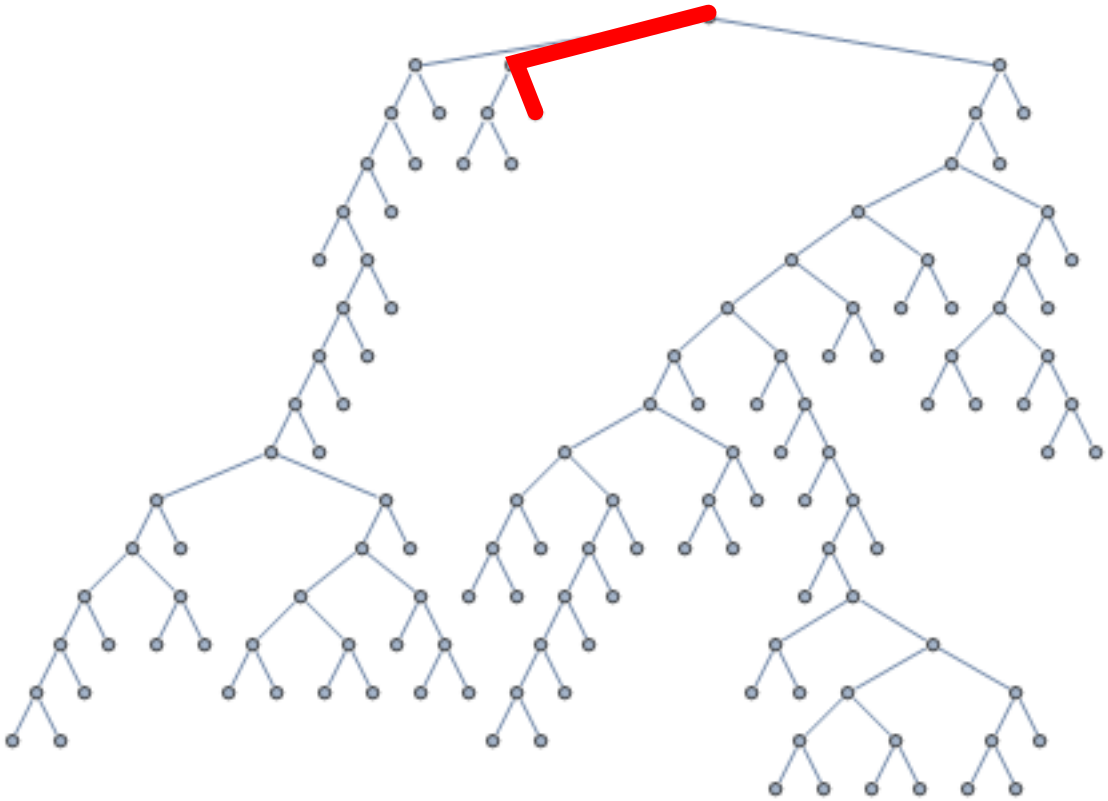
BFS vs. DFS



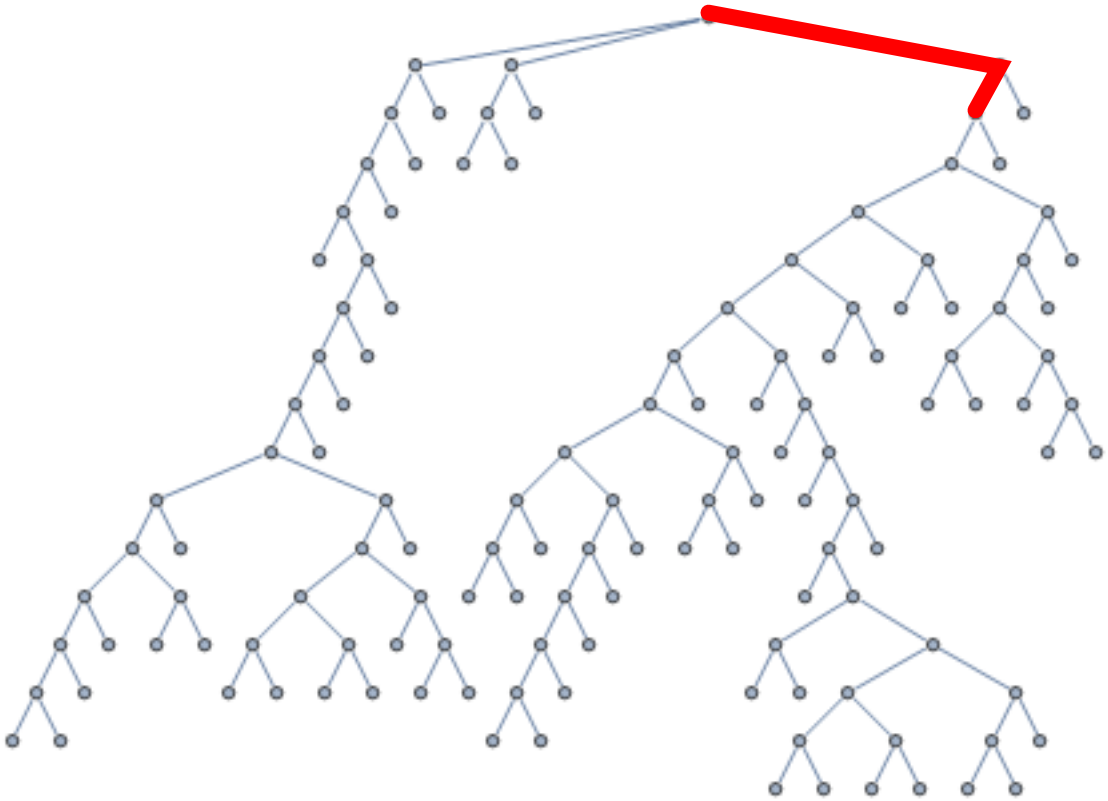
BFS vs. DFS



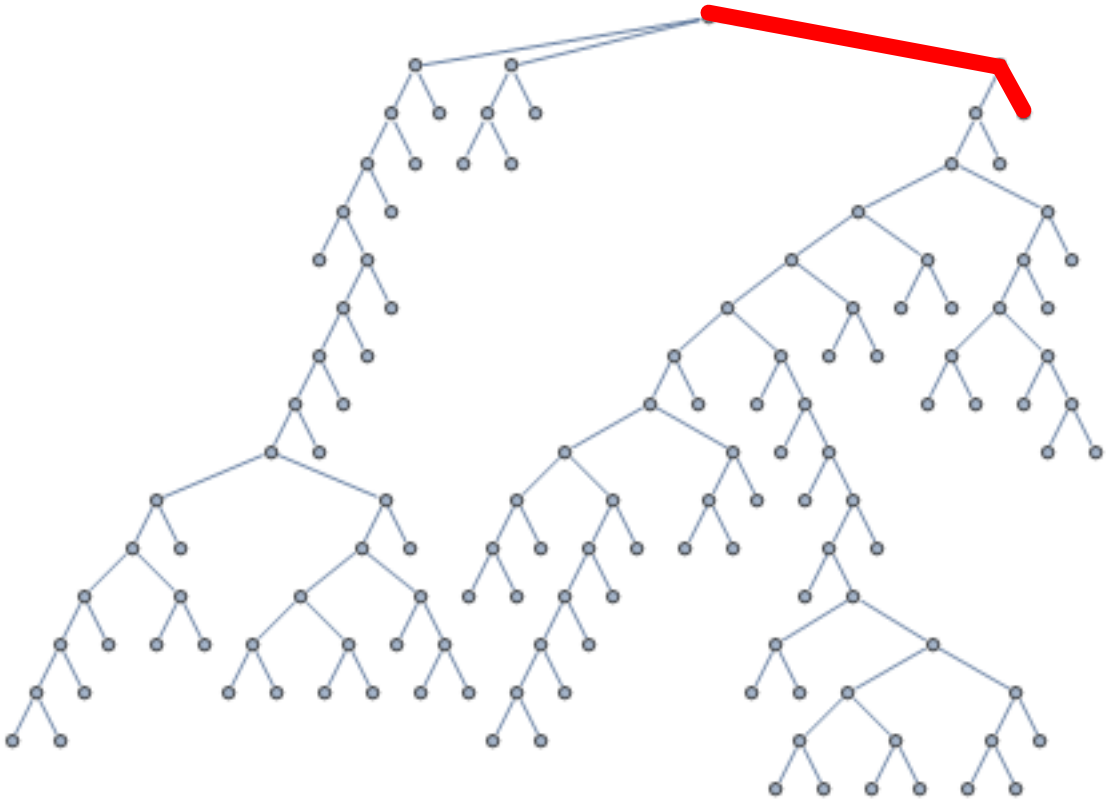
BFS vs. DFS



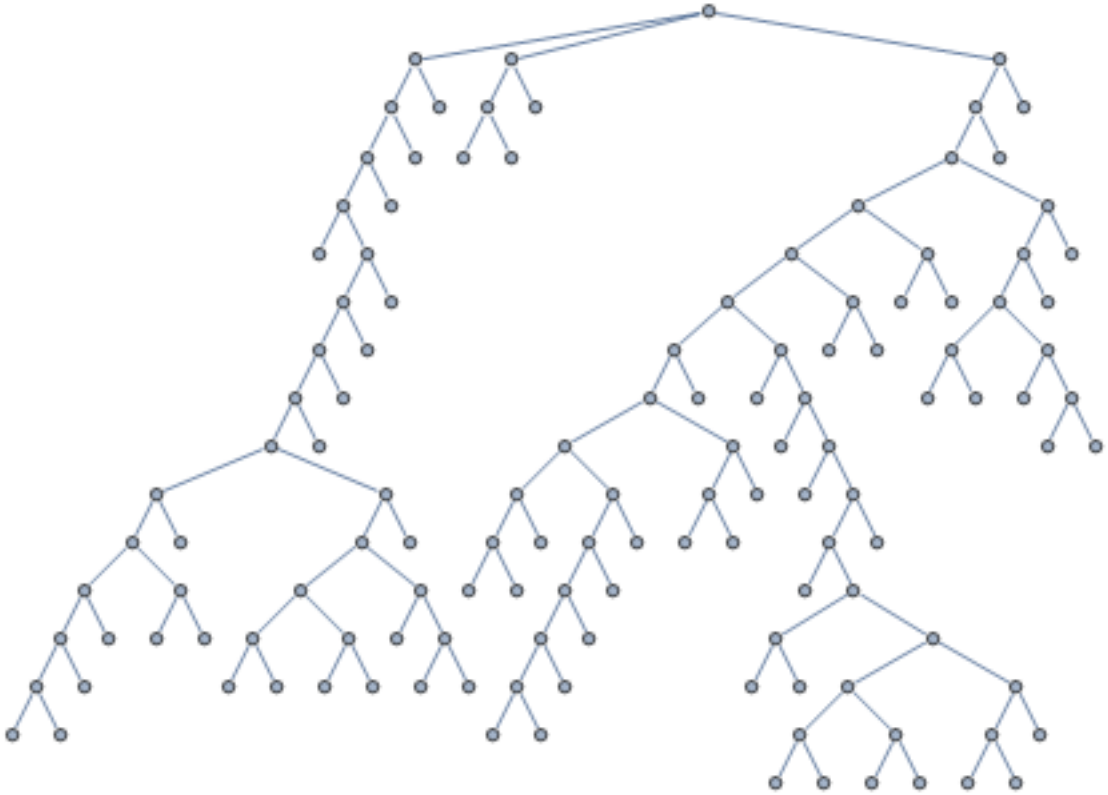
BFS vs. DFS



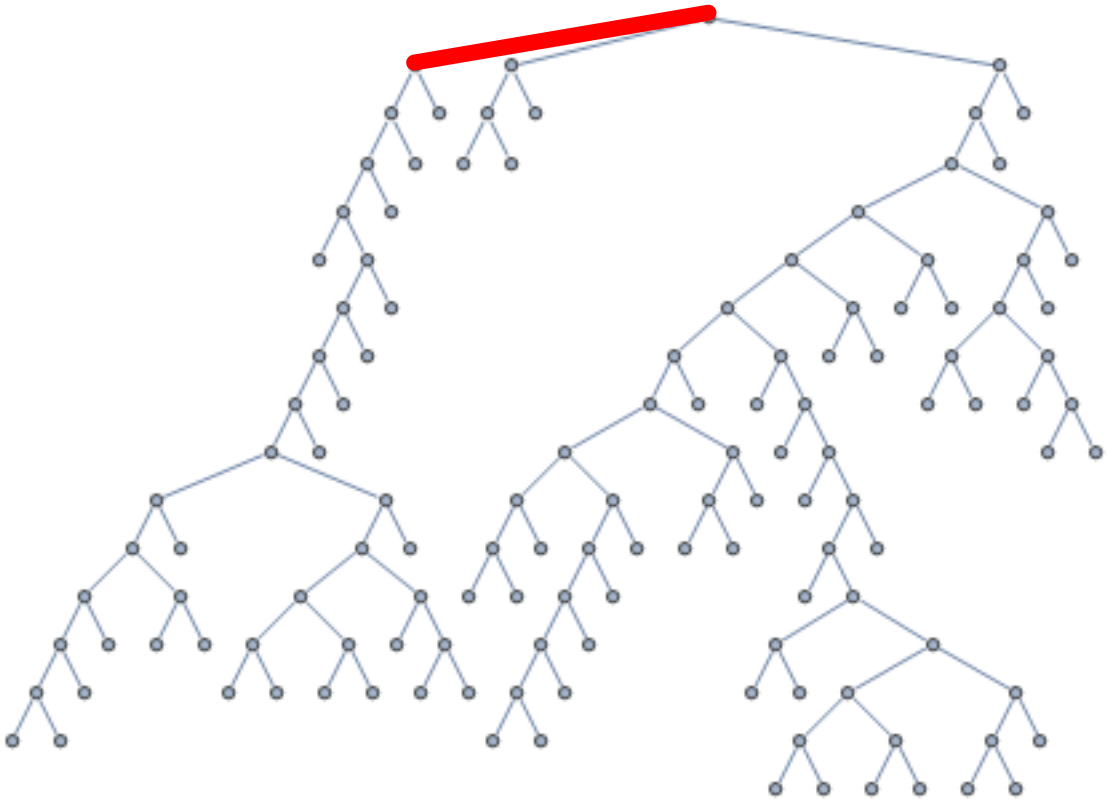
BFS vs. DFS



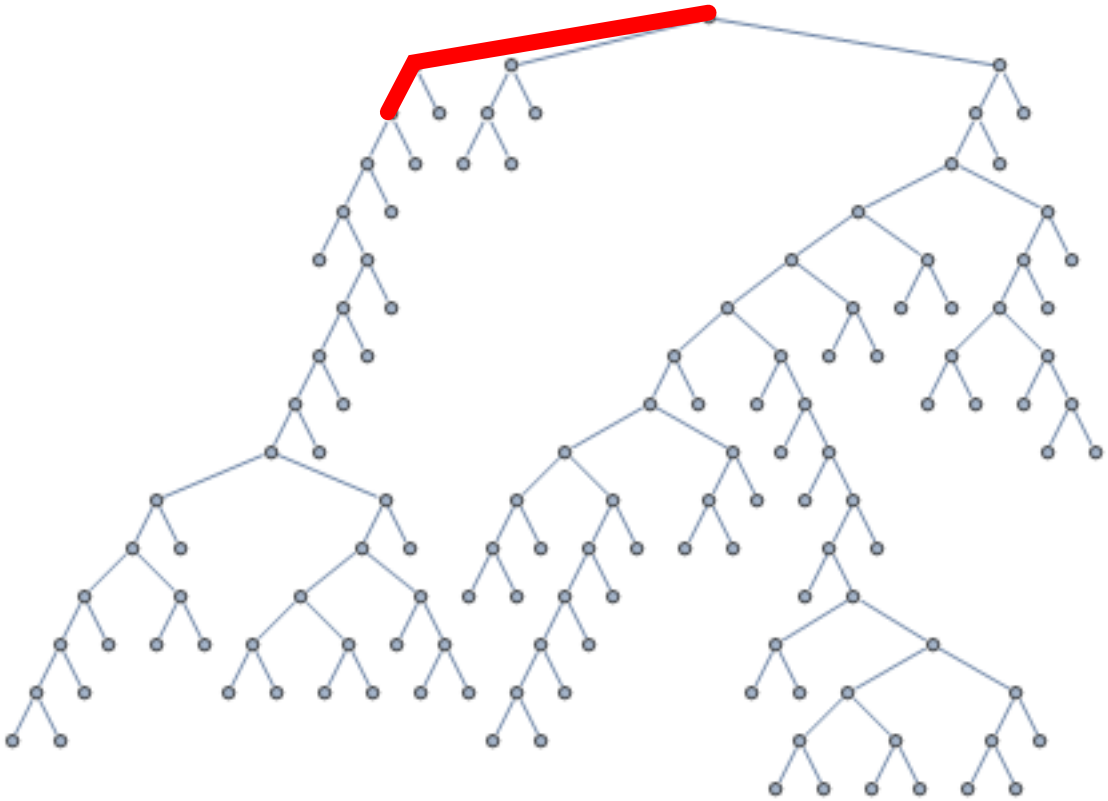
BFS vs. **DFS**



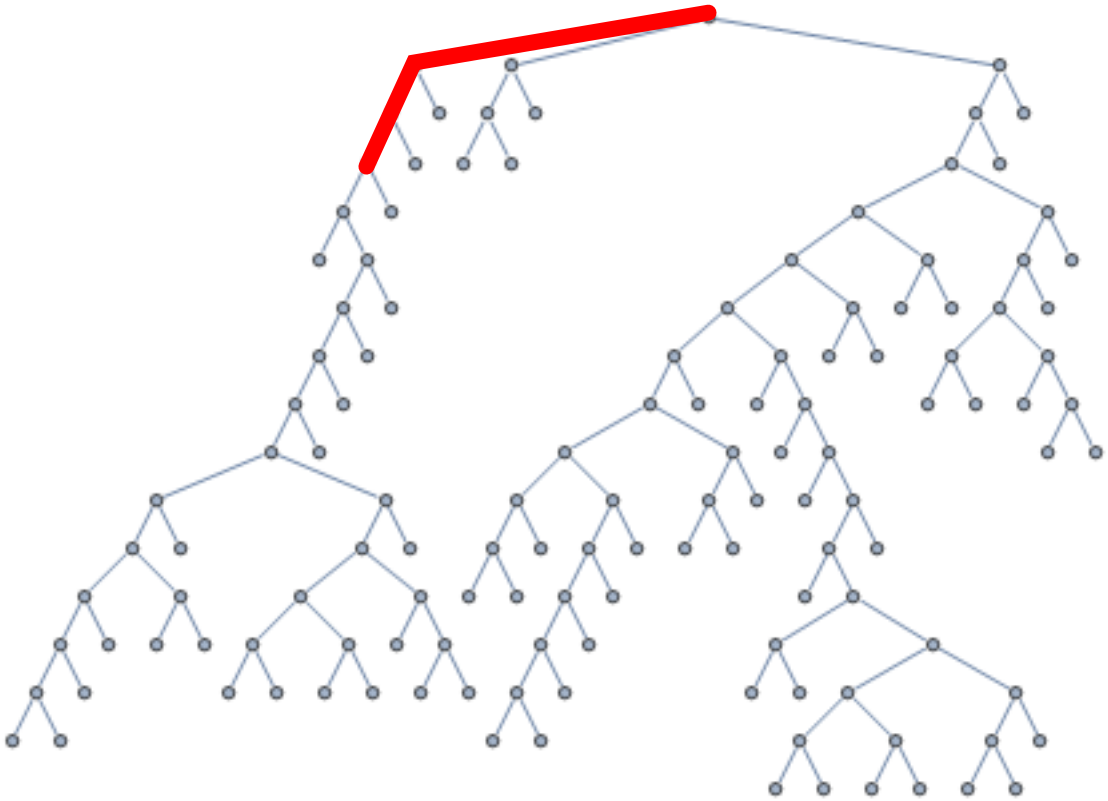
BFS vs. **DFS**



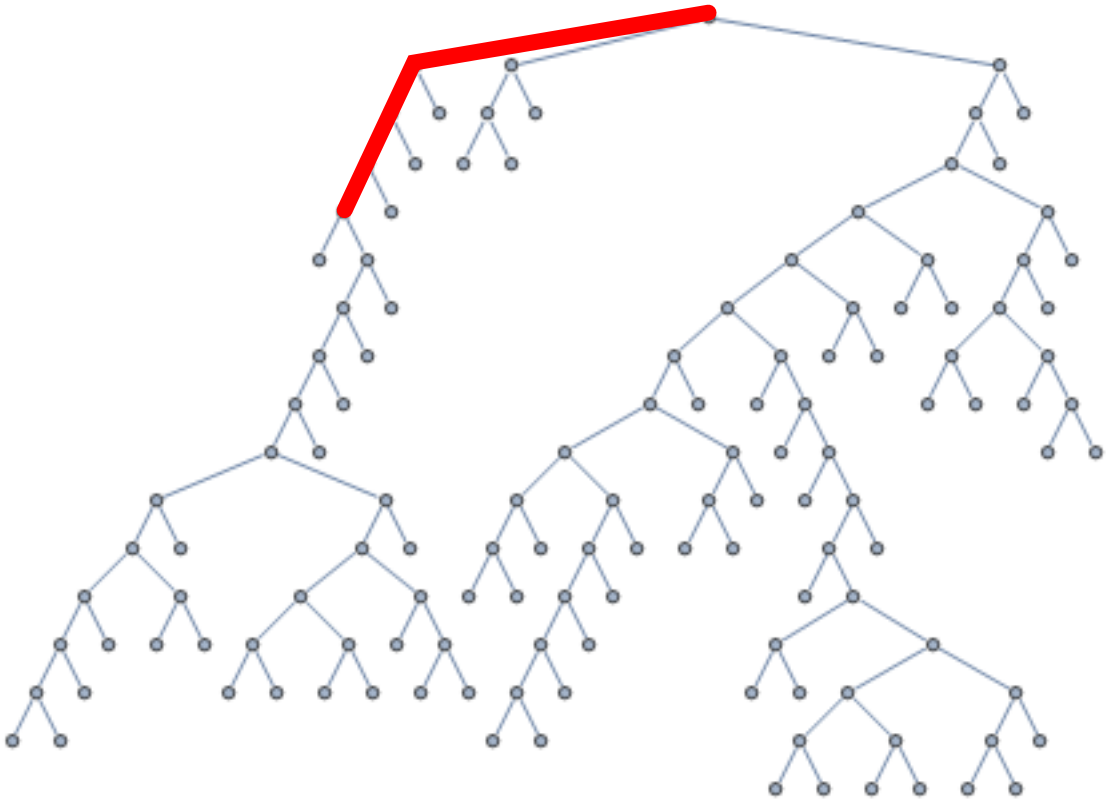
BFS vs. **DFS**



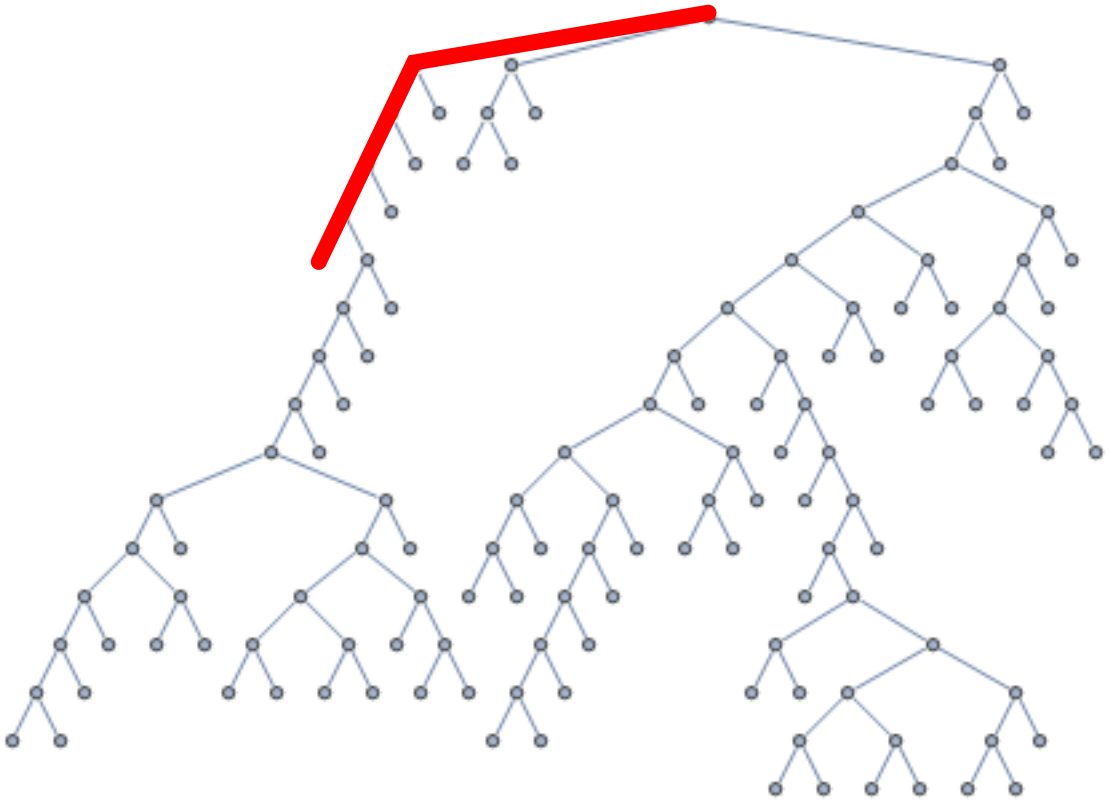
BFS vs. **DFS**



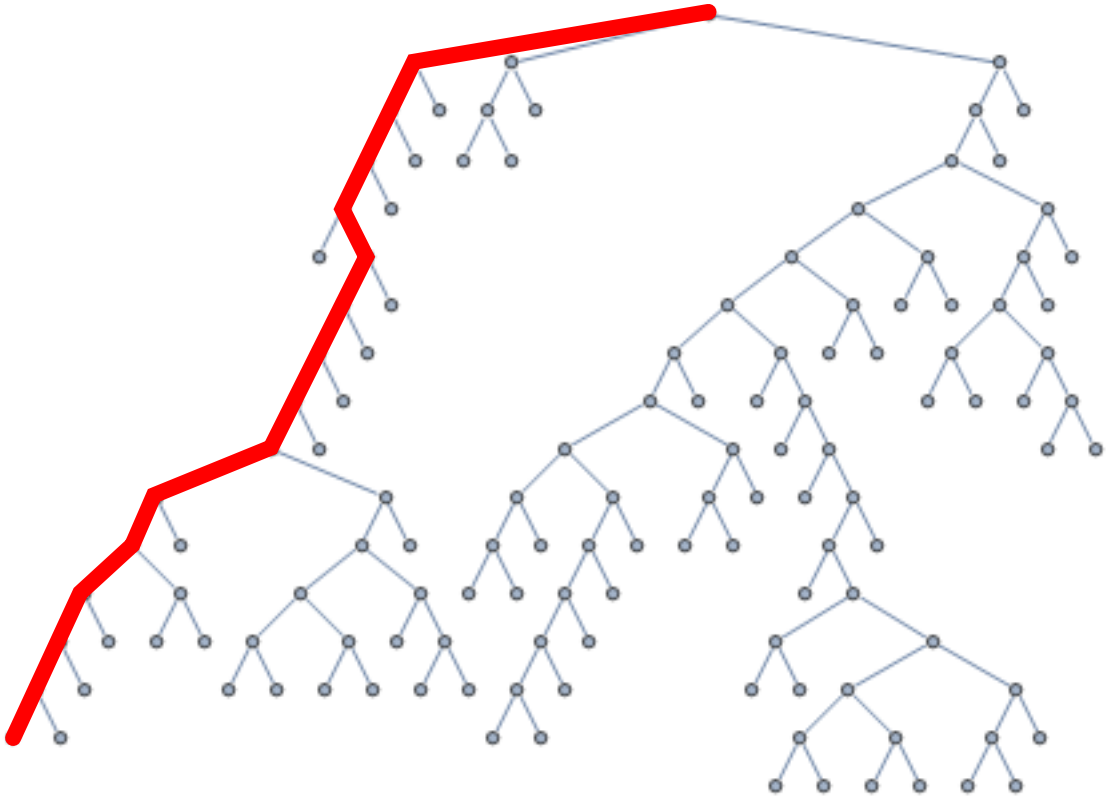
BFS vs. **DFS**



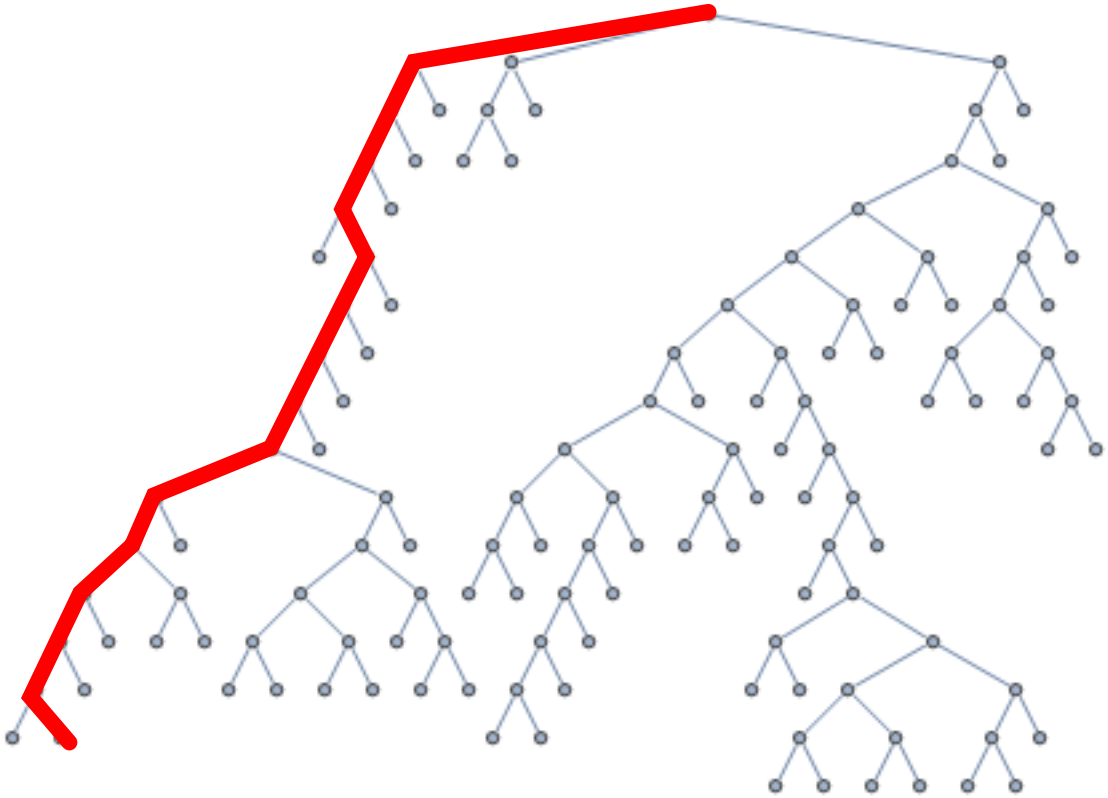
BFS vs. **DFS**



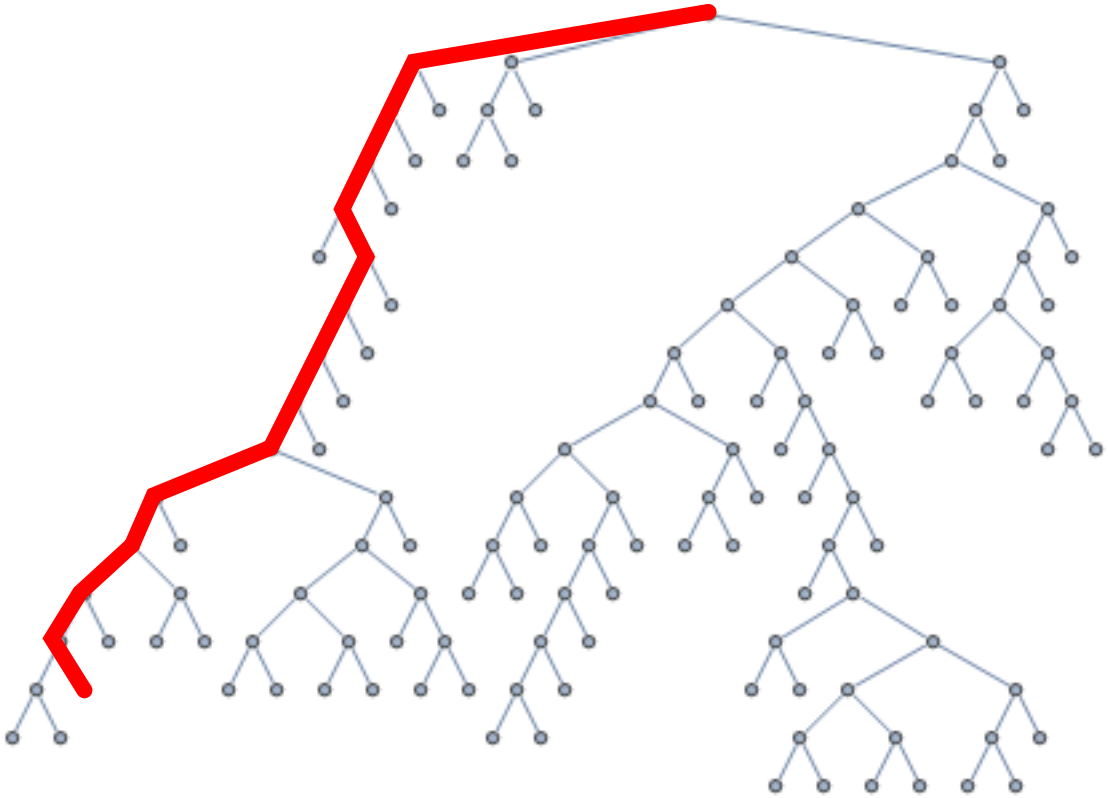
BFS vs. **DFS**



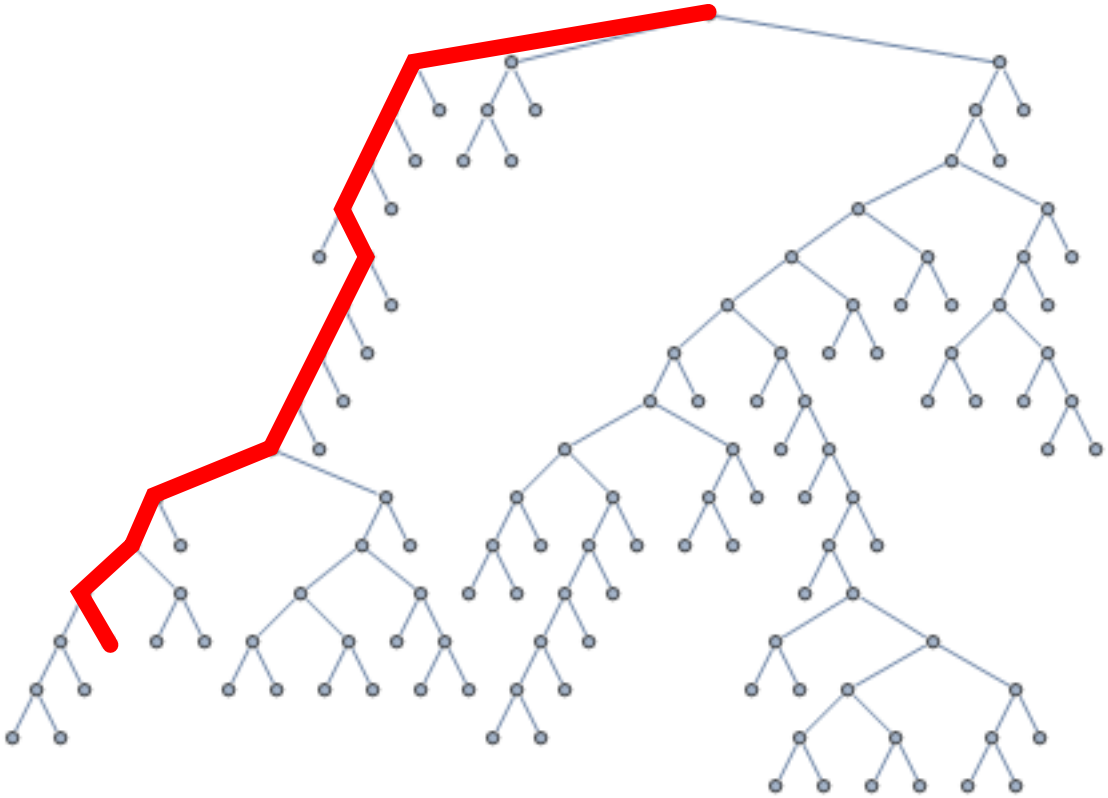
BFS vs. **DFS**



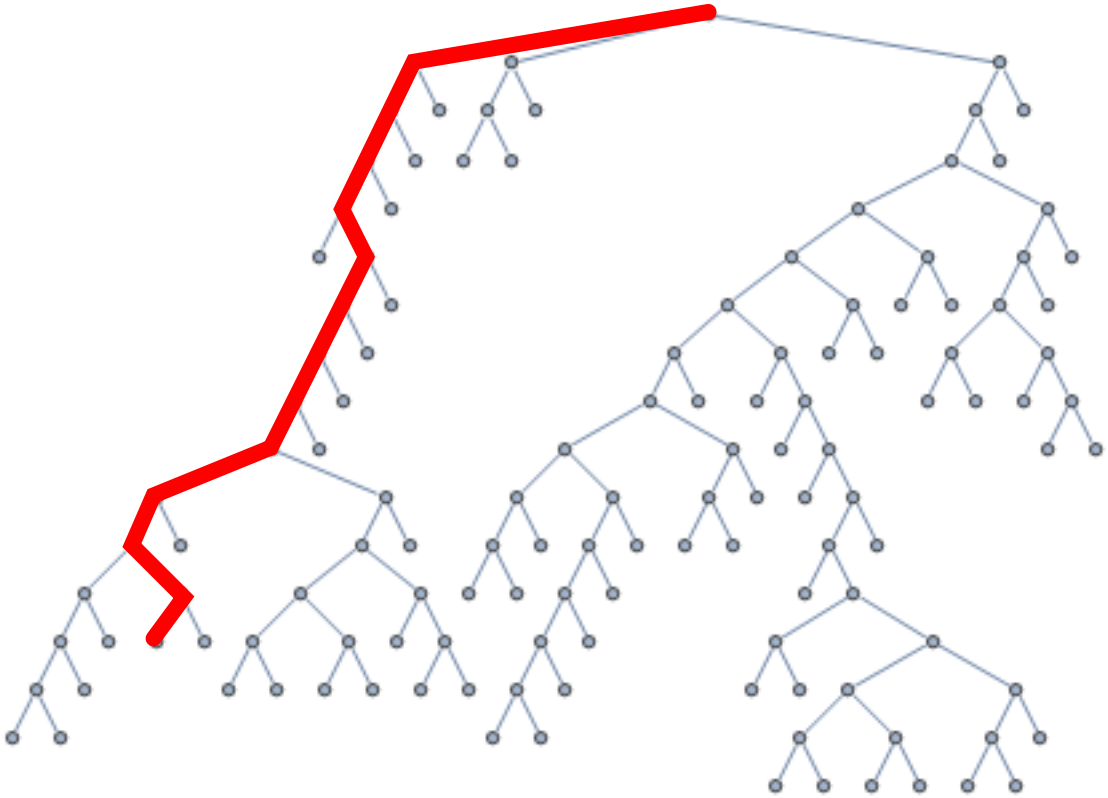
BFS vs. **DFS**



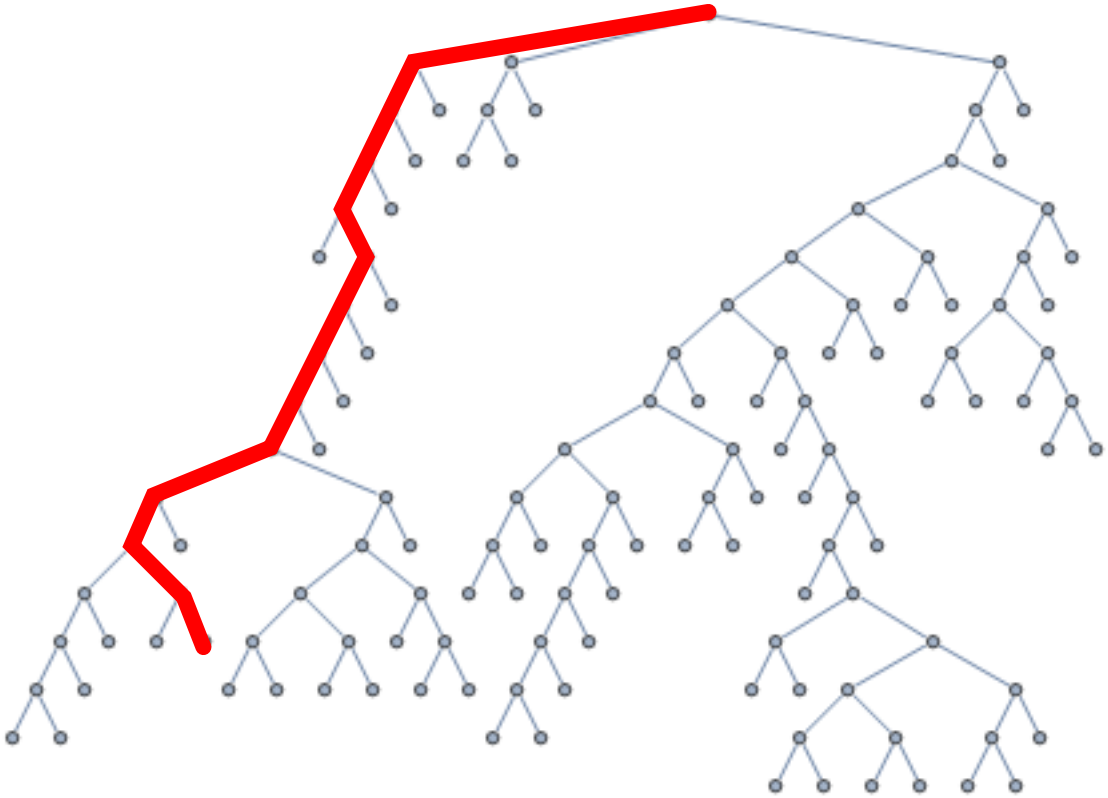
BFS vs. **DFS**



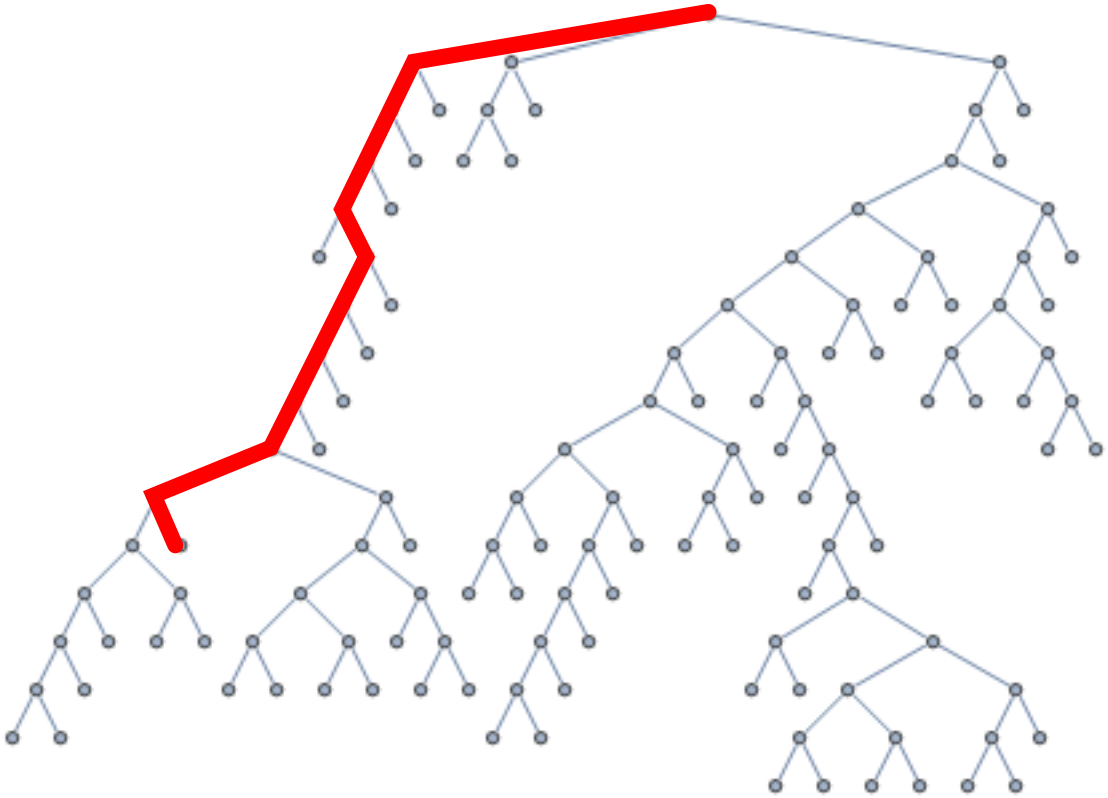
BFS vs. **DFS**



BFS vs. **DFS**



BFS vs. **DFS**

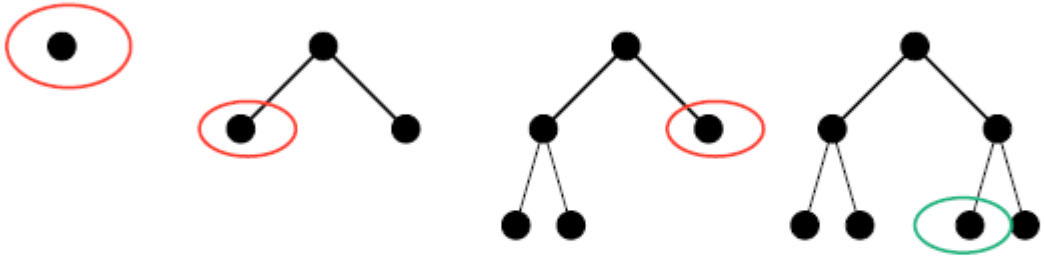


Uninformed search strategies

- A search strategy is defined by picking the order of node expansion
- Uninformed search (blind search) strategies use only the information available in the problem definition
- **Can only distinguish goal states from non-goal state**
- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search

Breadth First Search

function BREADTH-FIRST-SEARCH (*problem*) **returns** a solution or failure
  return GENERAL-SEARCH (*problem*, ENQUEUE-AT-END)

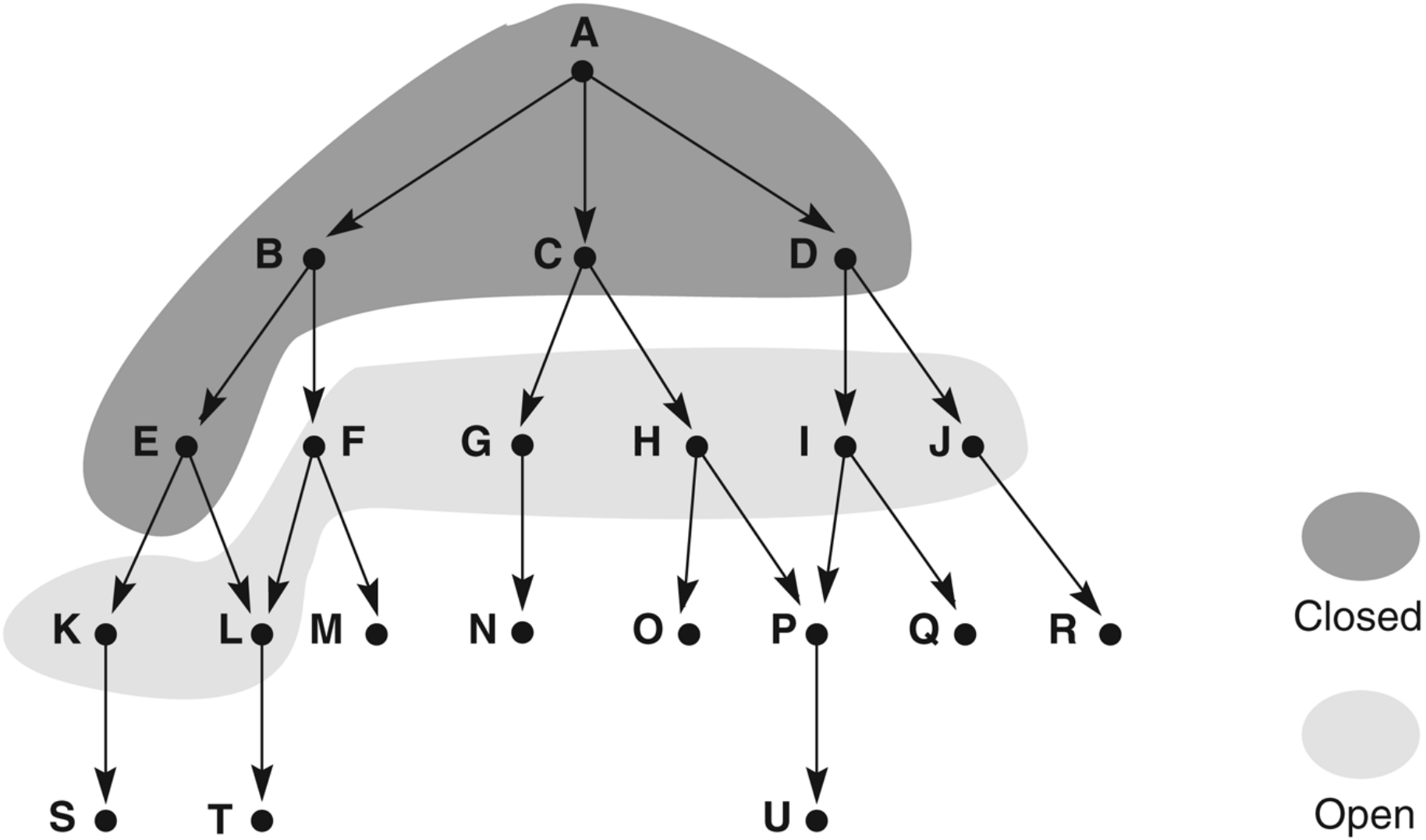


Breadth-first search tree after 0,1,2 and 3 node expansions

Expand: red nodes; Goal: green node

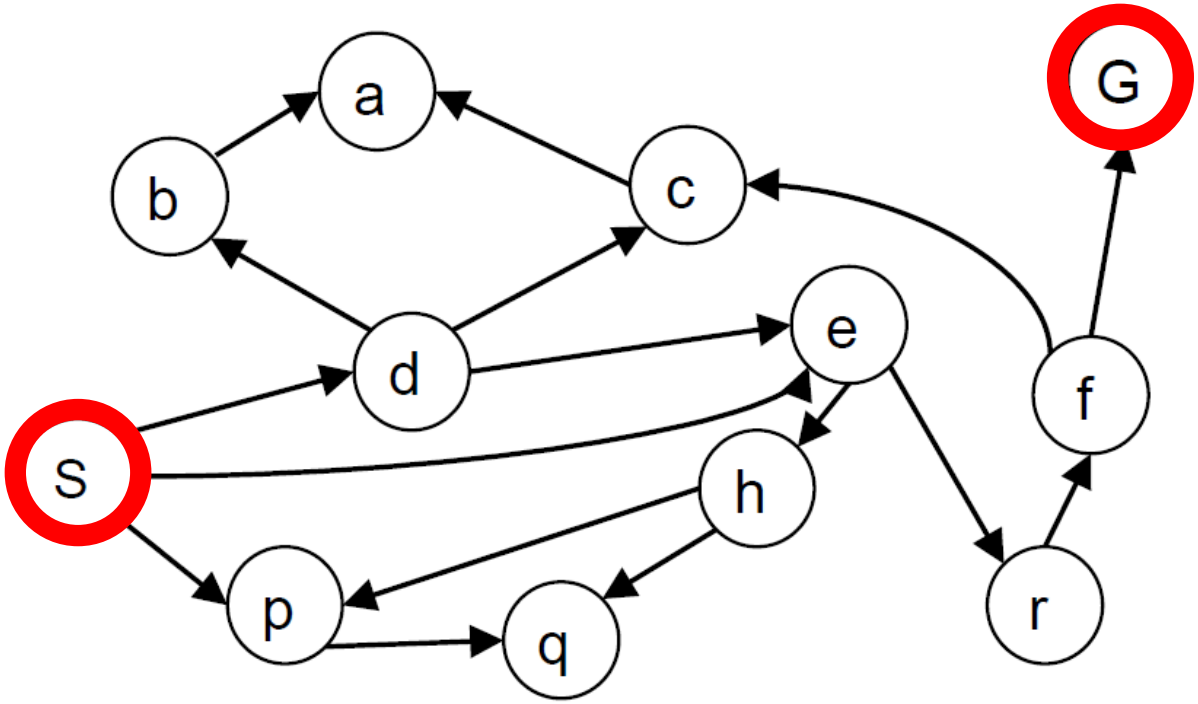
In this example, b = 2, and d = 2

Breadth-first search



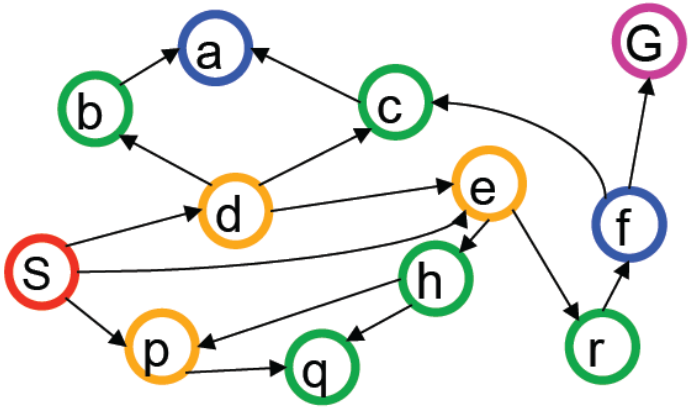
Breadth-first search

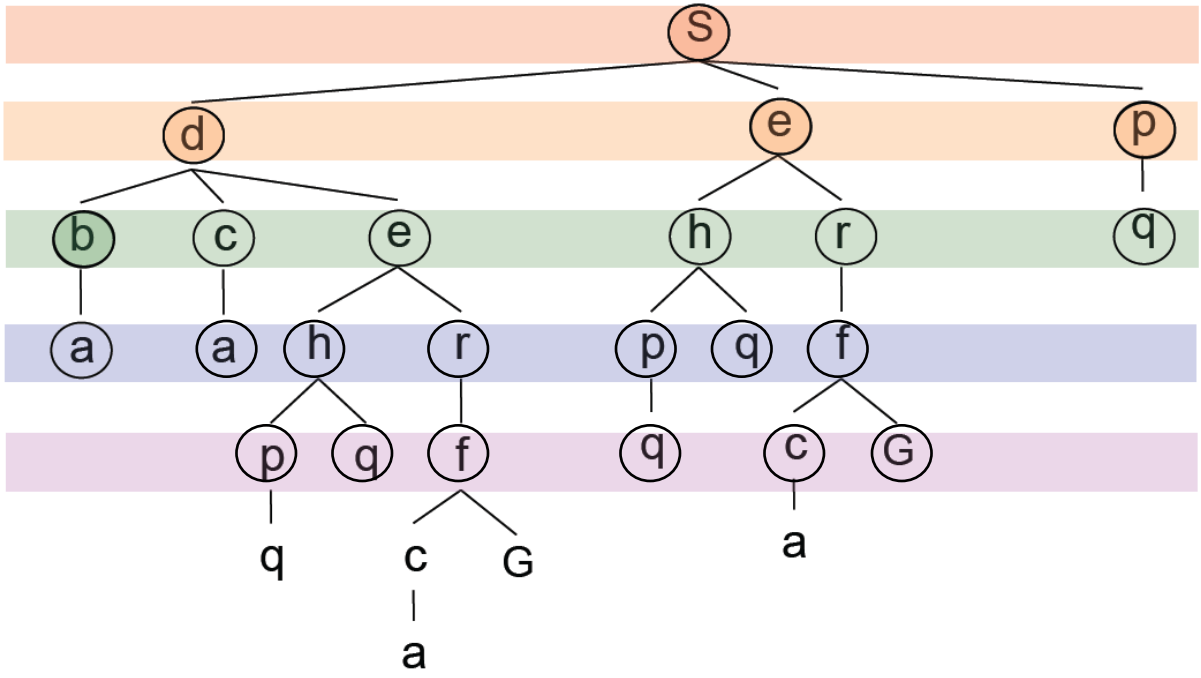
- Expand shallowest unexpanded node
- Implementation: *frontier* is a FIFO queue



Breadth-first search

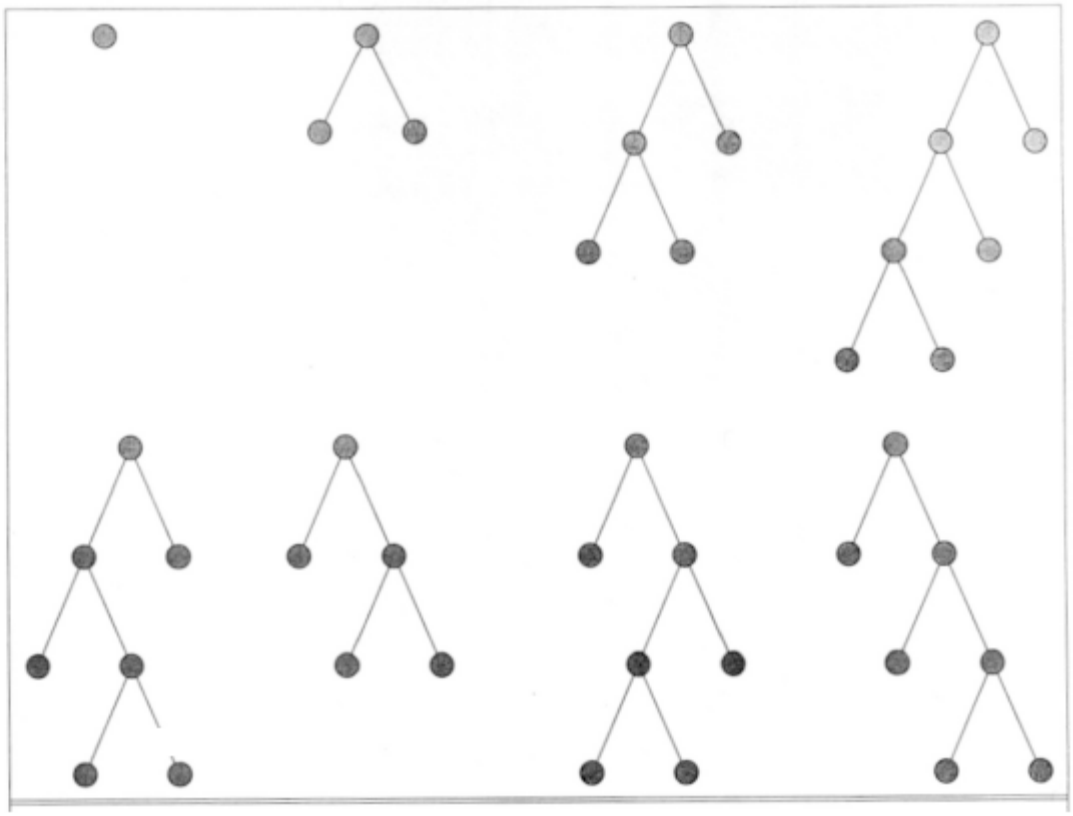
- Expansion order:
  (S,d,e,p,b,c,e,h,r,q,a,a,
  h,r,p,q,f,p,q,f,q,c,G)



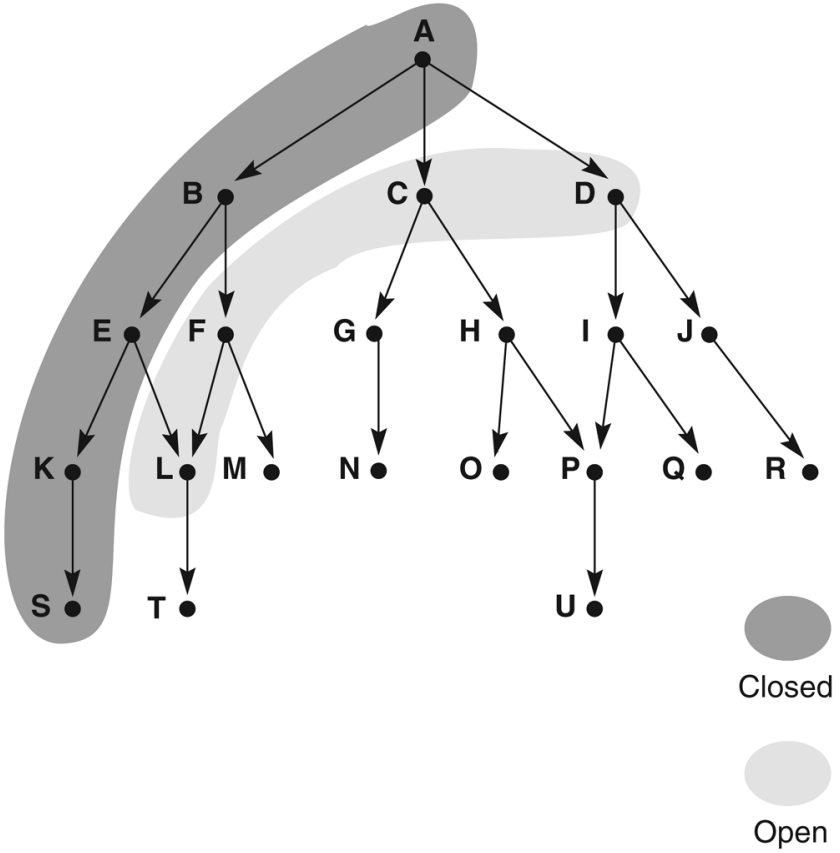


function DEPTH-FIRST-SEARCH (*problem*) **returns** a solution or failure
GENERAL-SEARCH (*problem*, ENQUEUE-AT-FRONT)

Alternatively can use a recursive implementation.

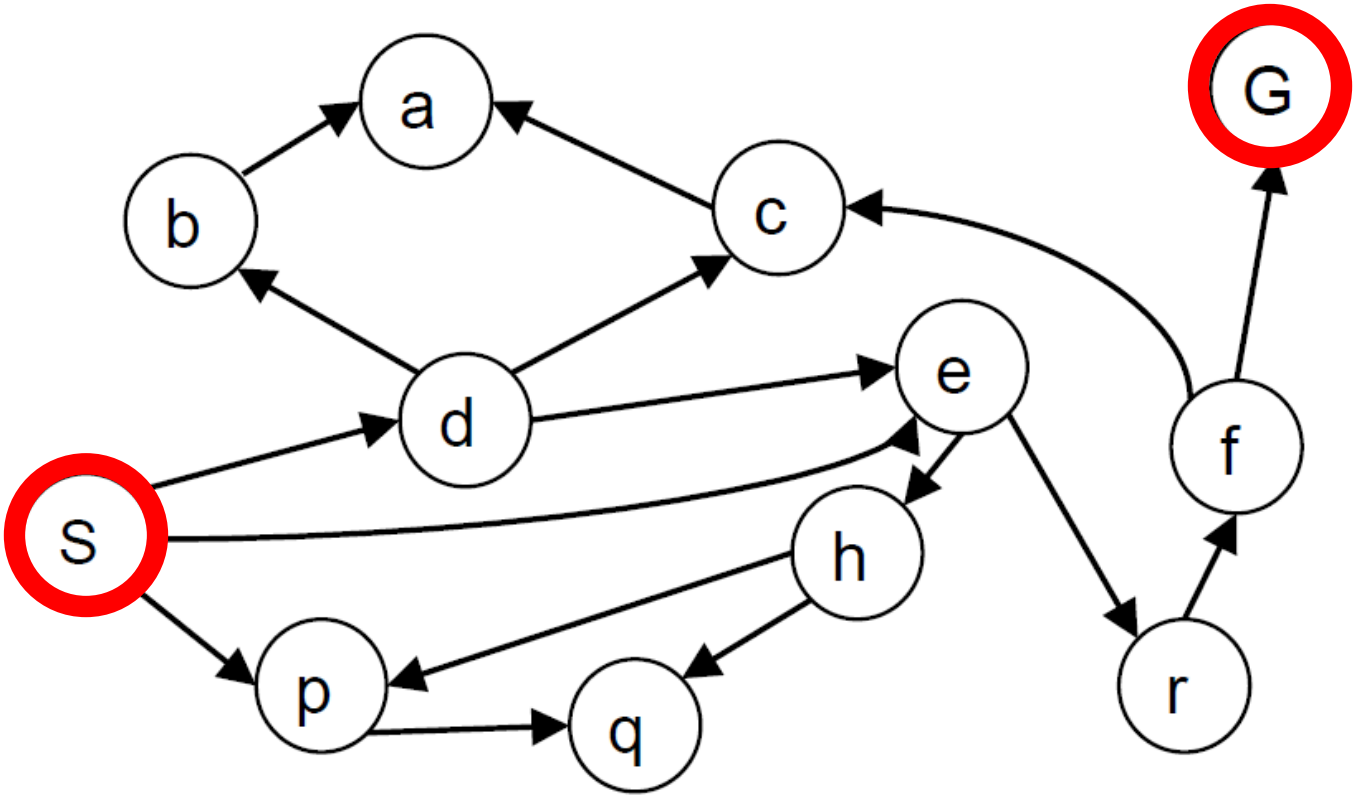


Depth-first search



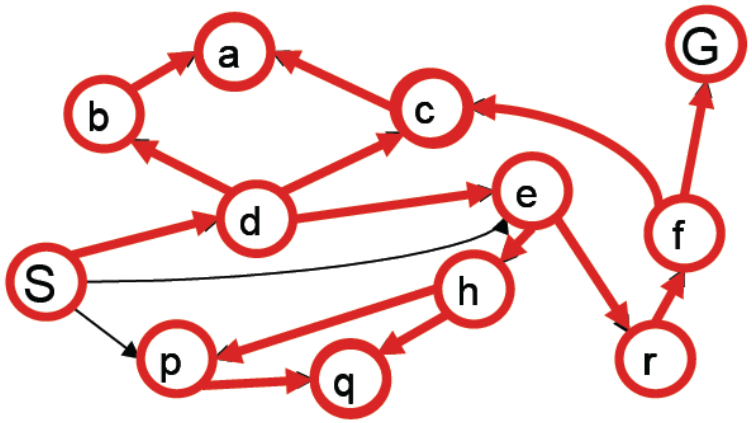
Depth-first search

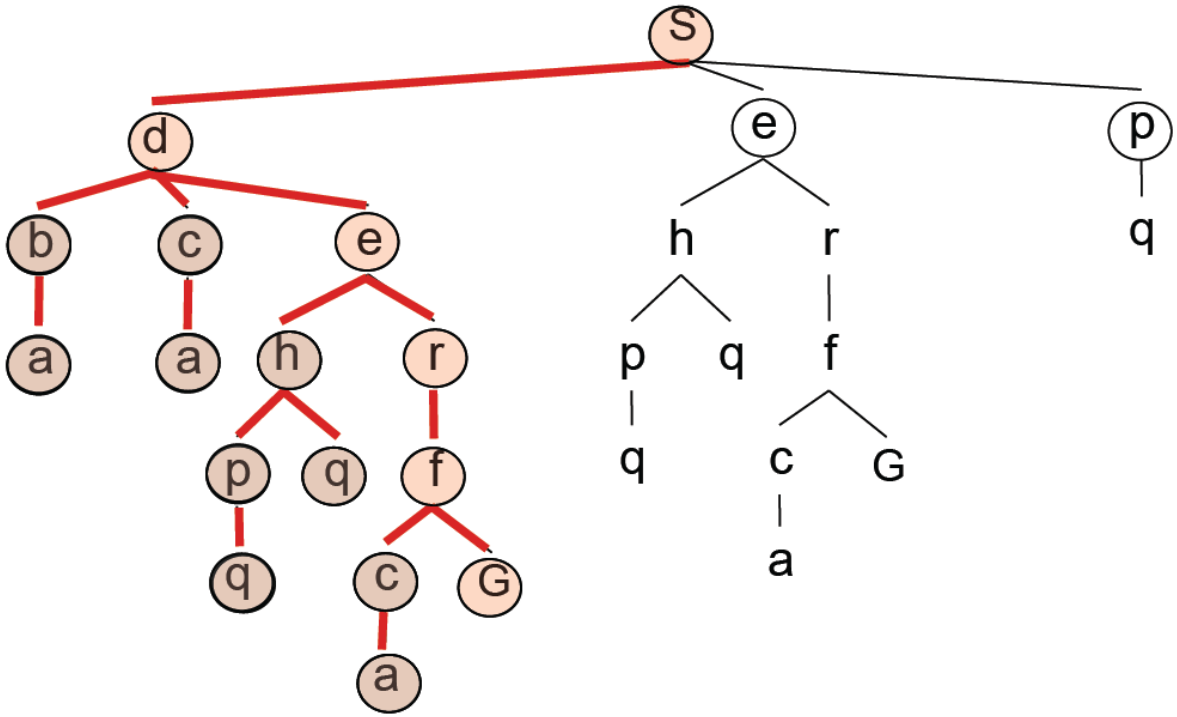
- Expand deepest unexpanded node
- Implementation: *frontier* is a LIFO queue

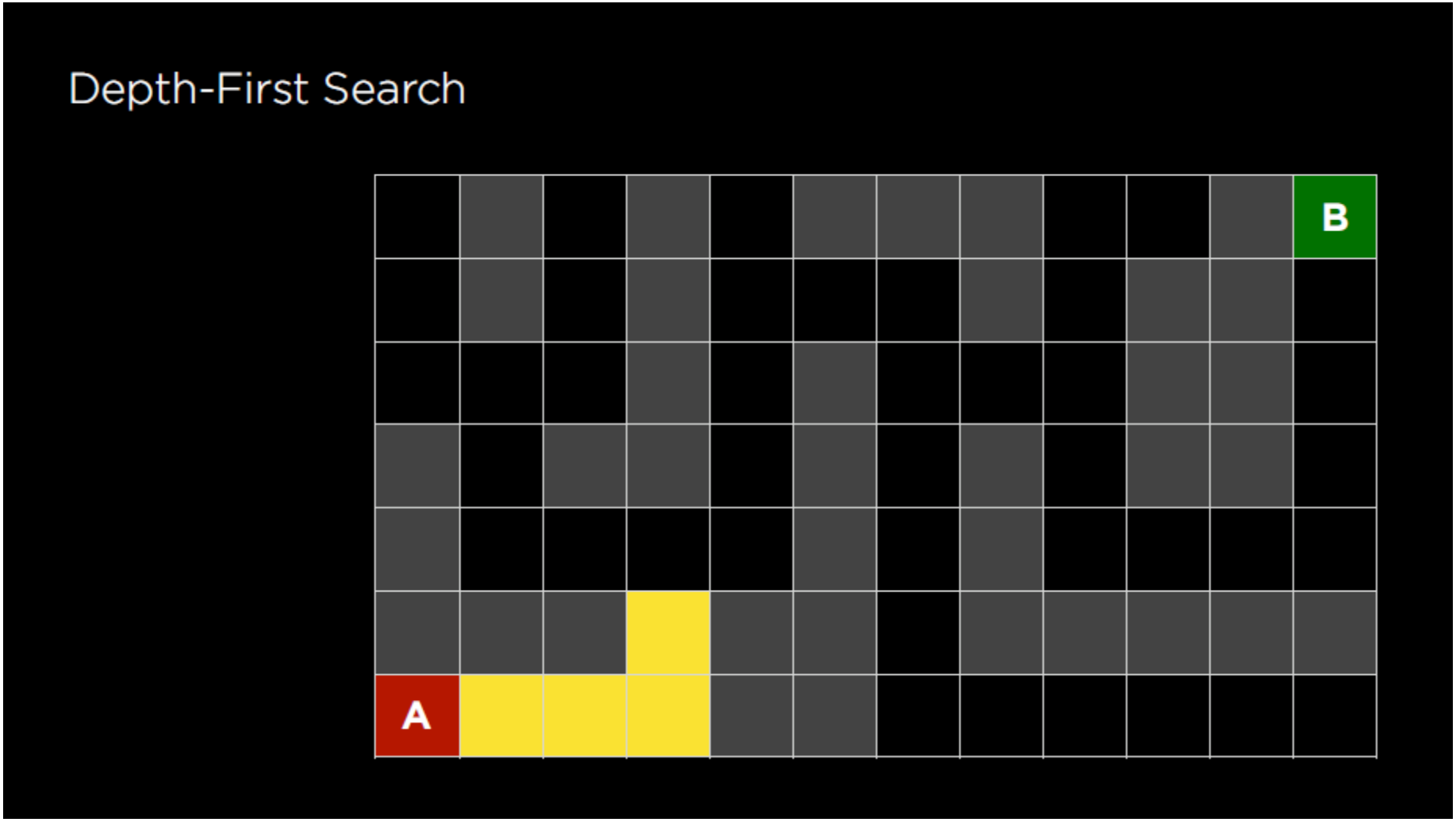


Depth-first search

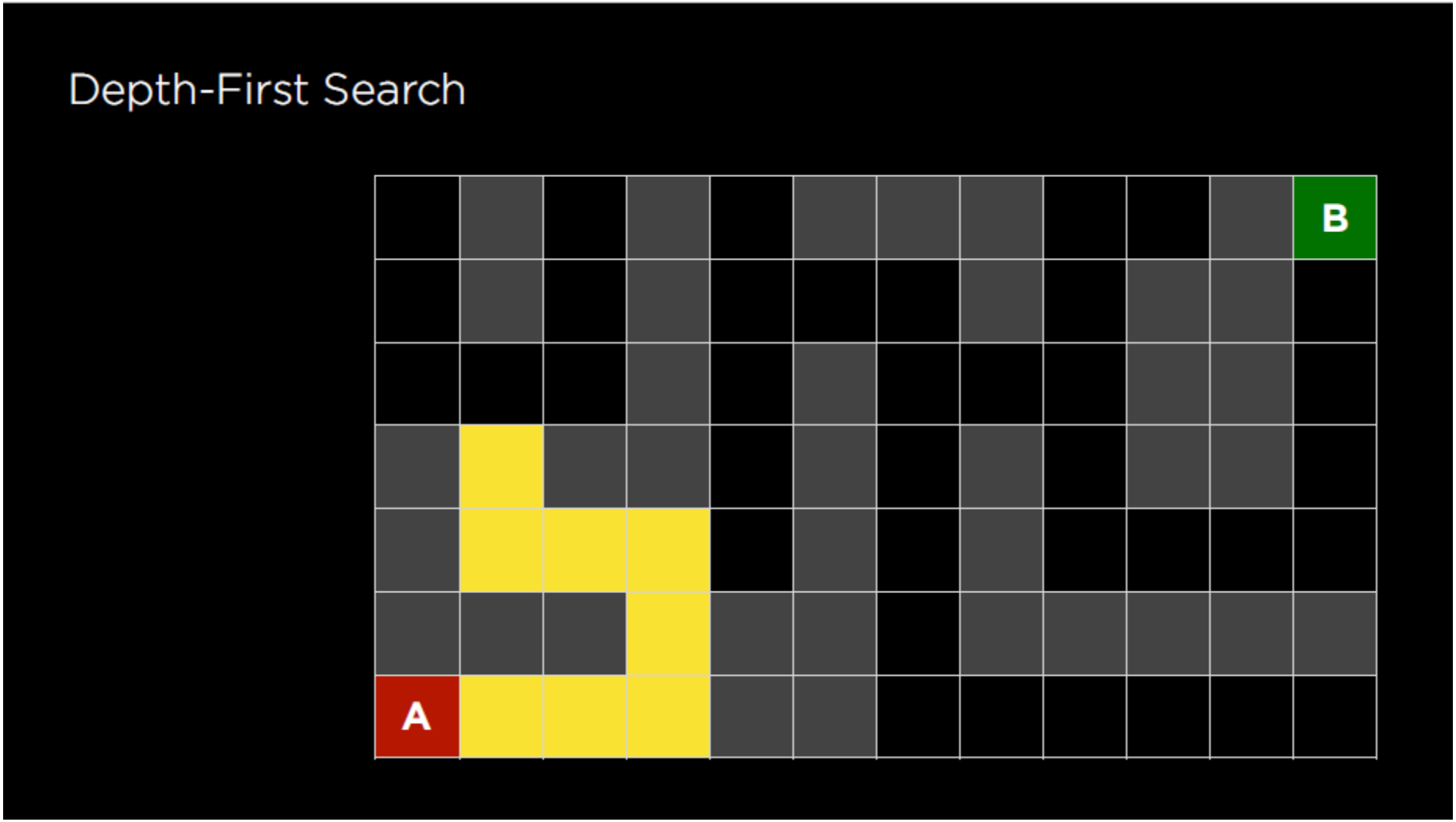
- Expansion order:
  (S,d,b,a,c,a,e,h,p,q,q,
  r,f,c,a,G)



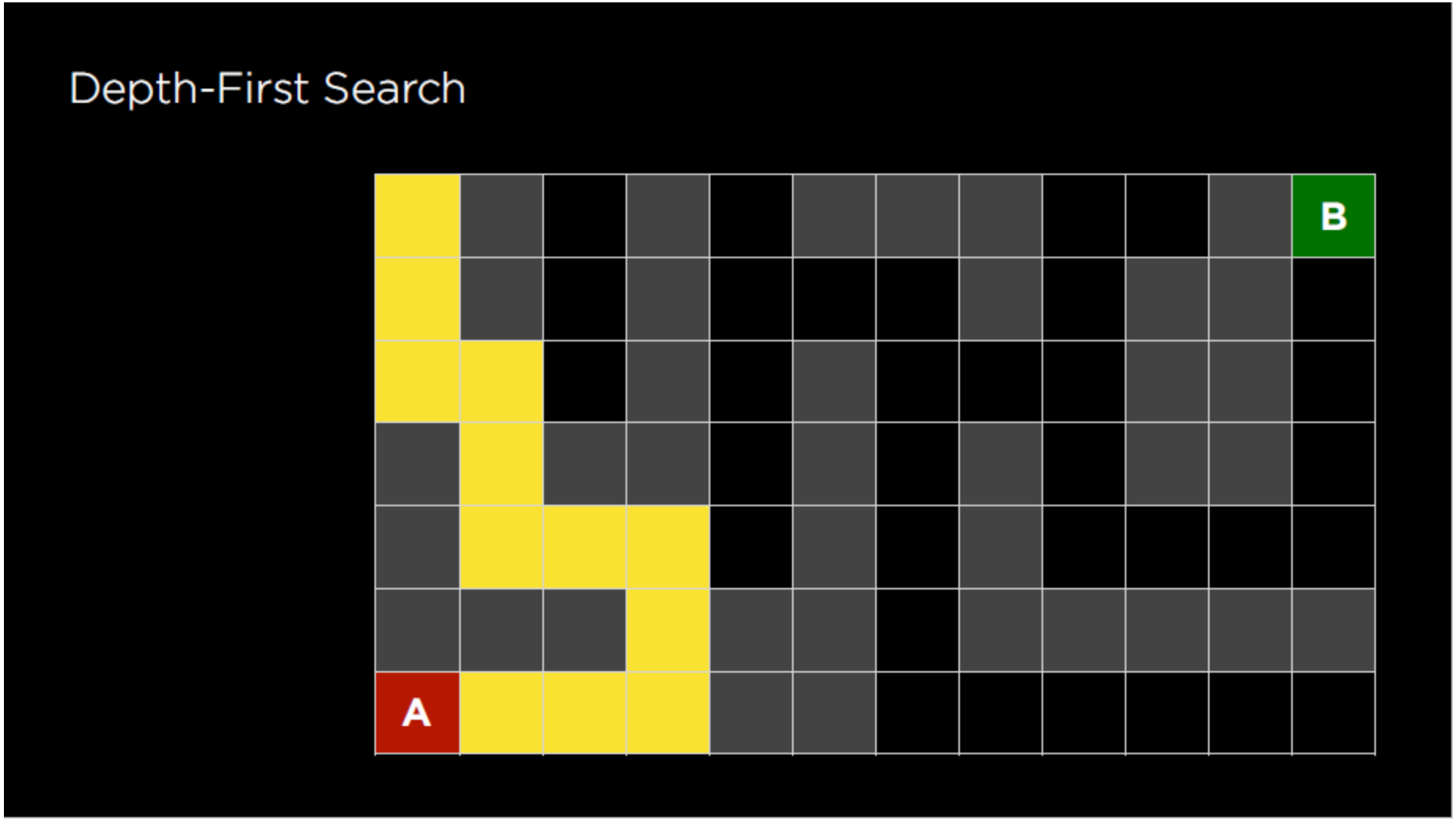




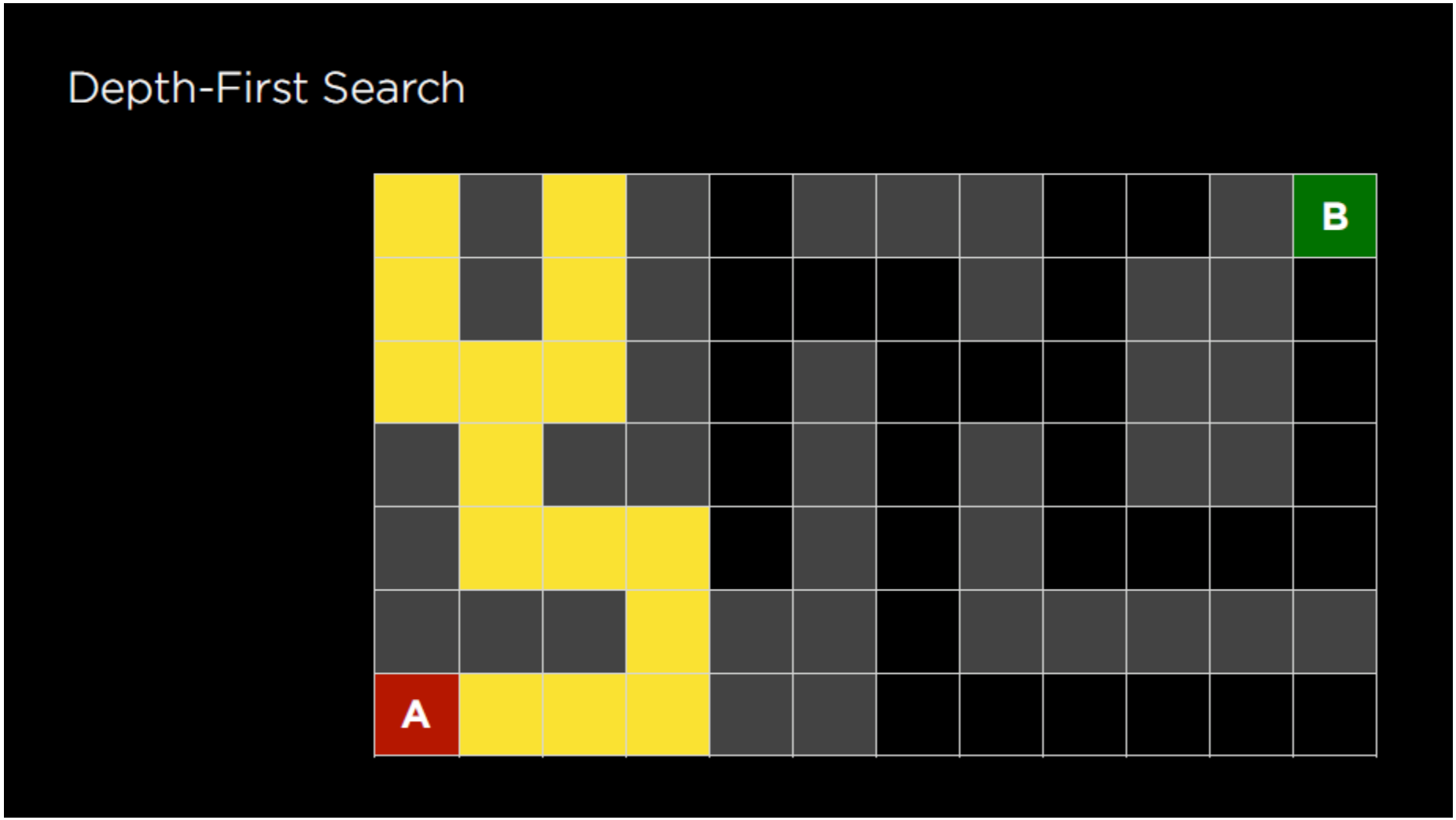
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

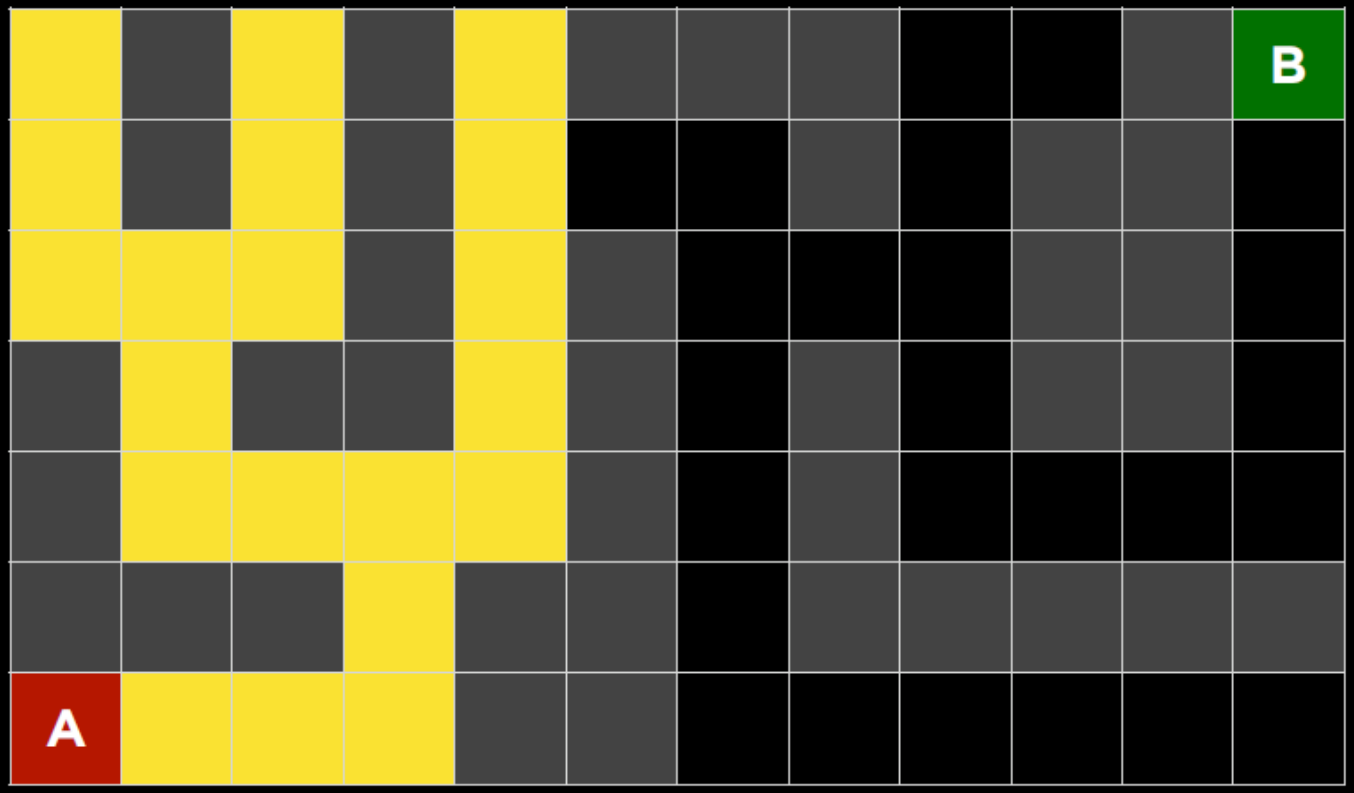


Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

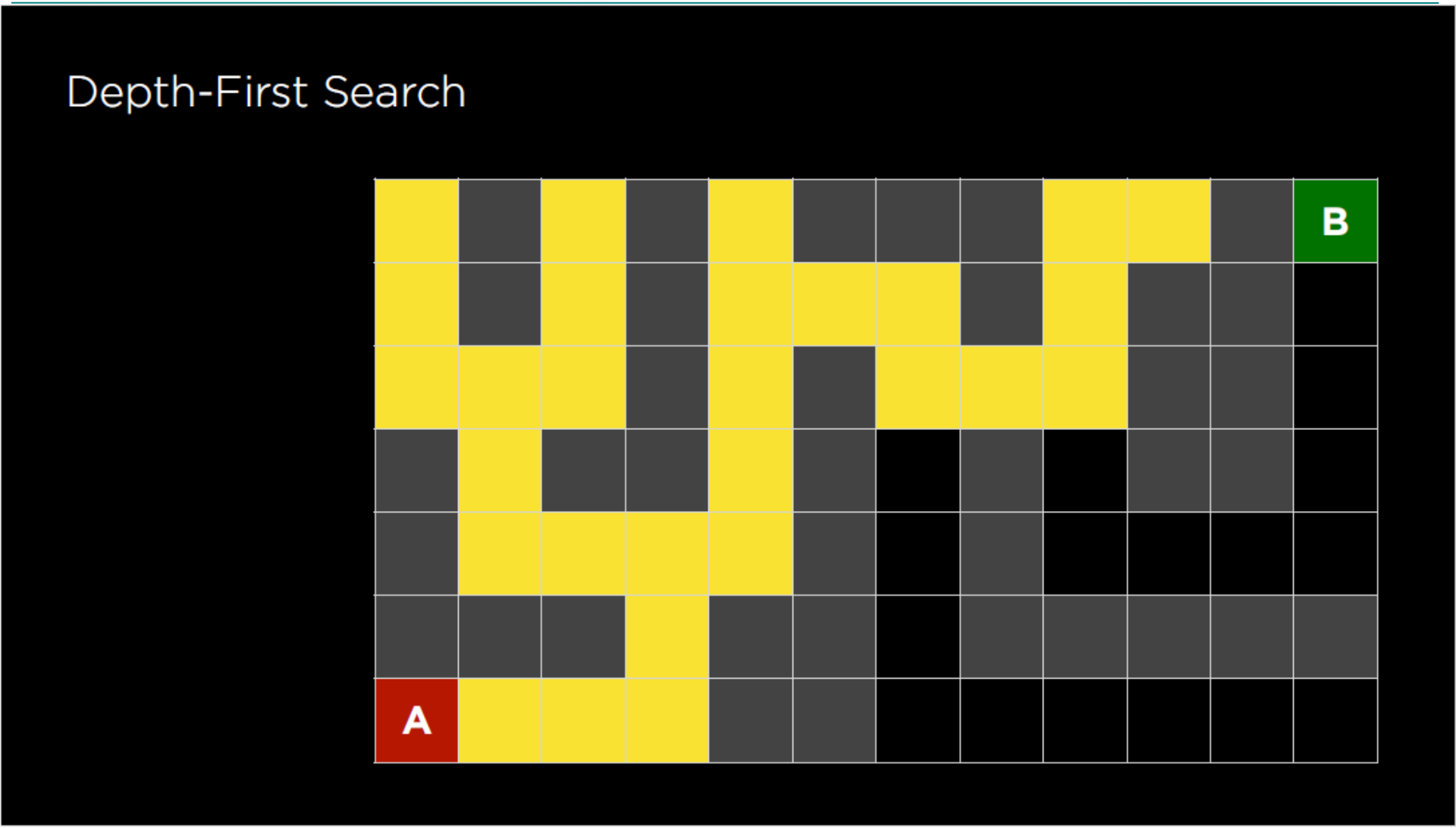


Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Depth-First Search

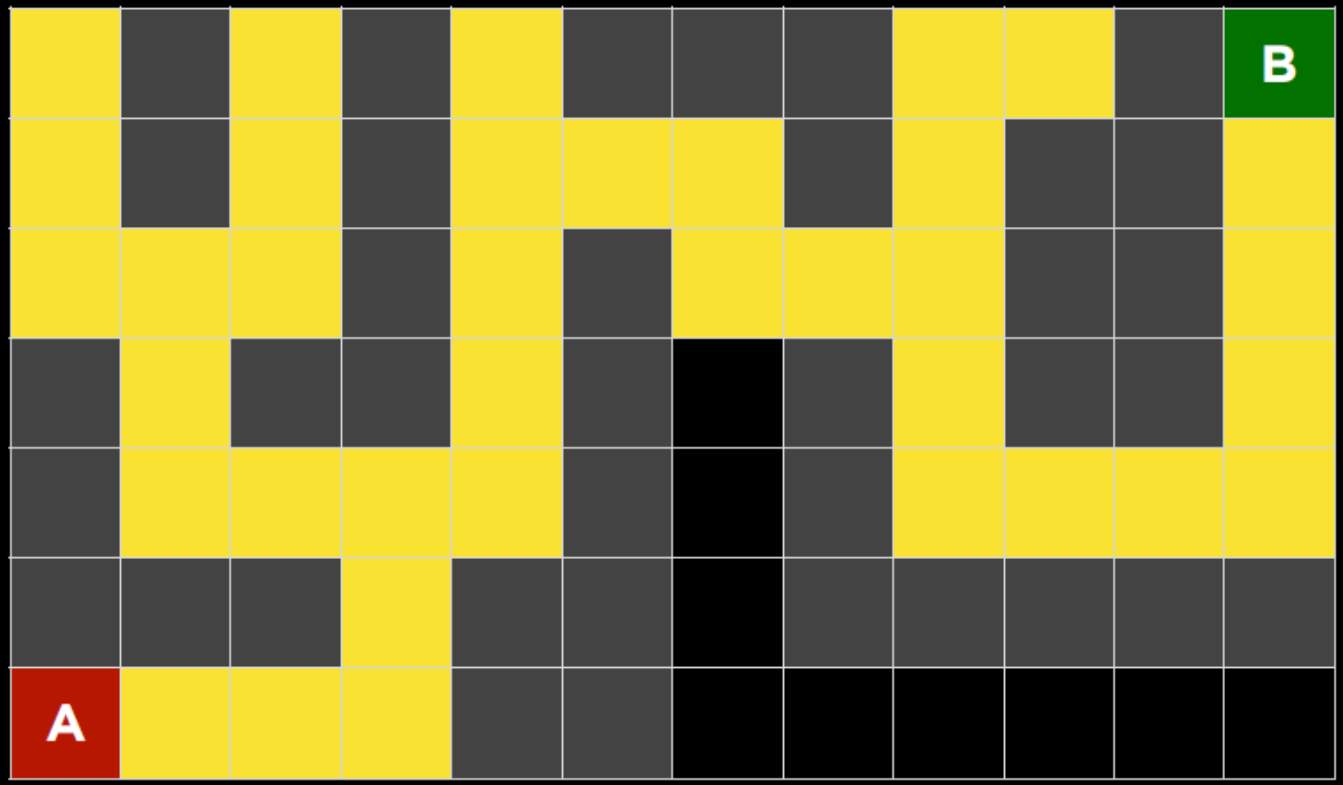


Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



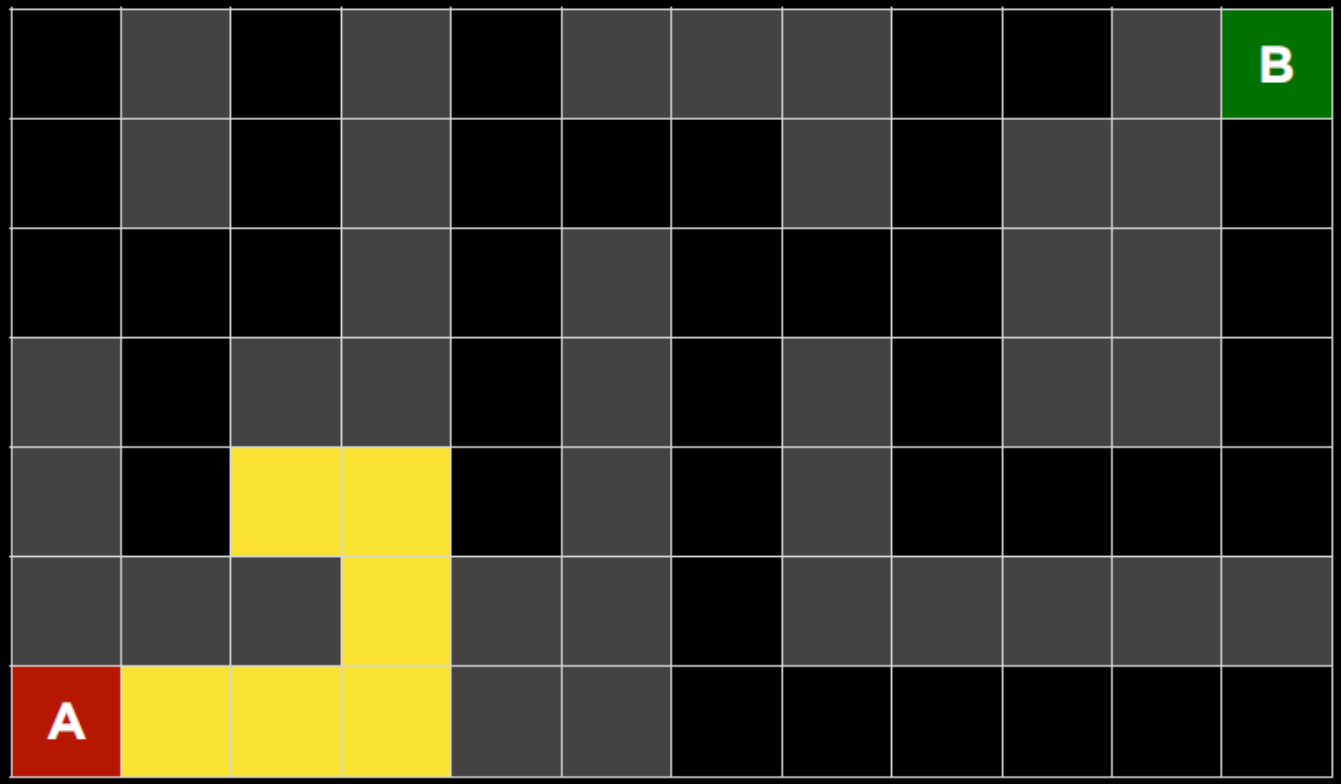
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Depth-First Search



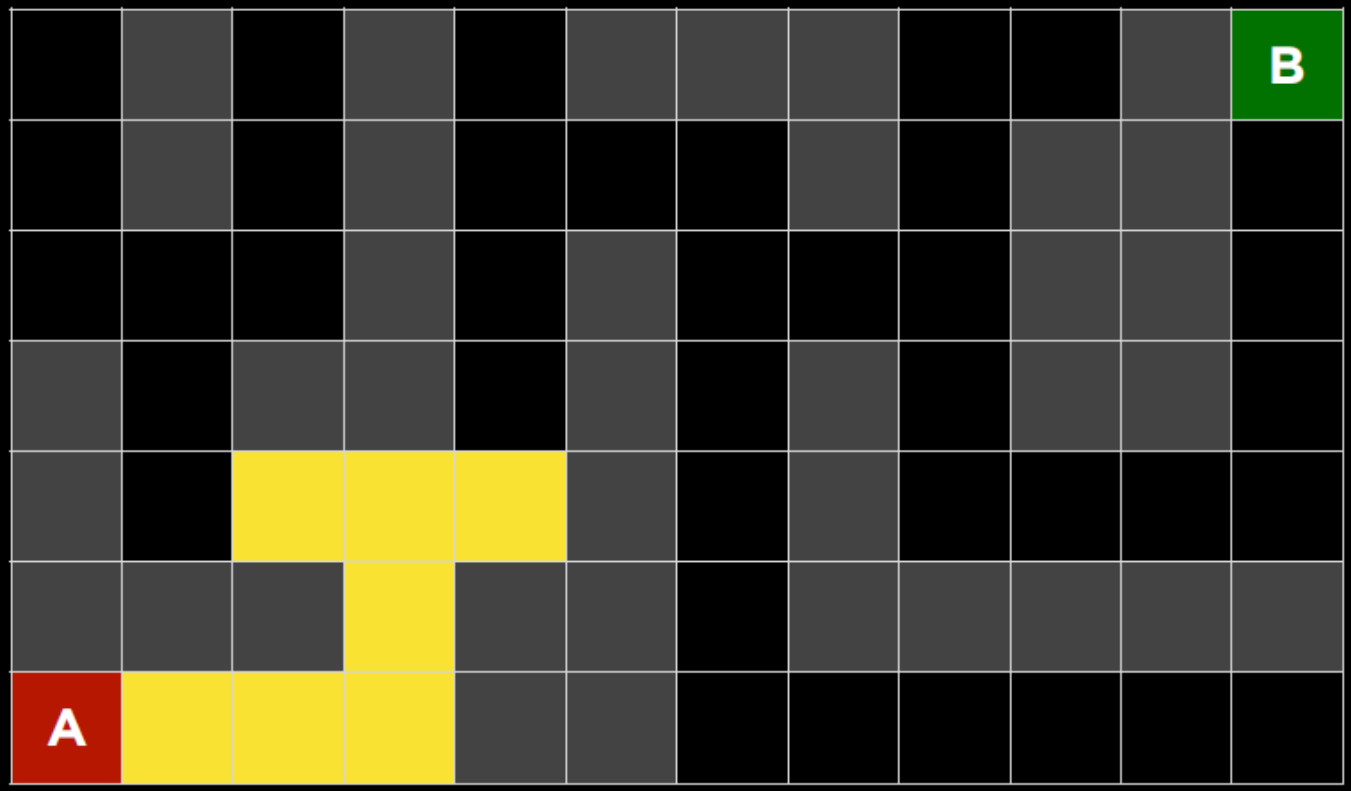
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Breadth-First Search



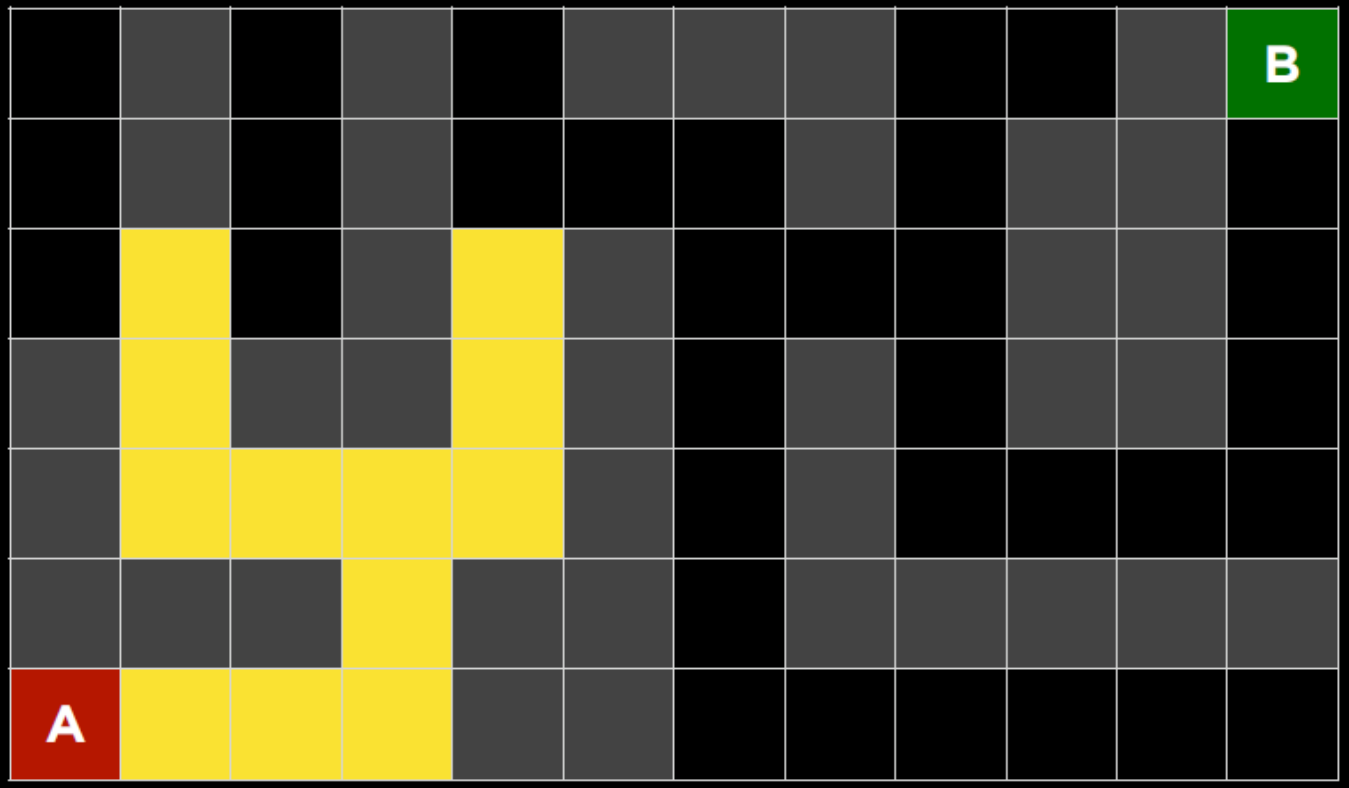
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Breadth-First Search

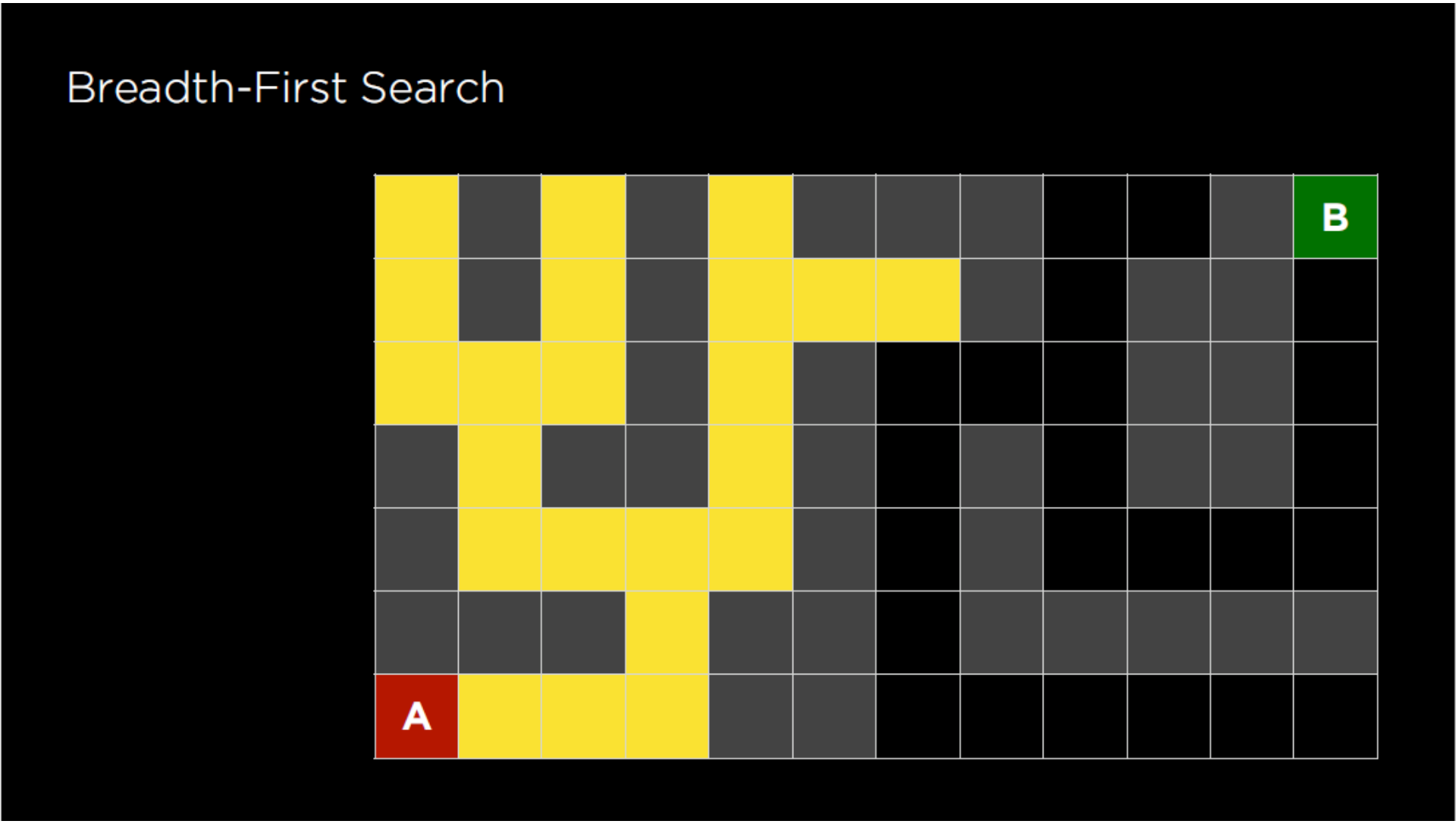


Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Breadth-First Search

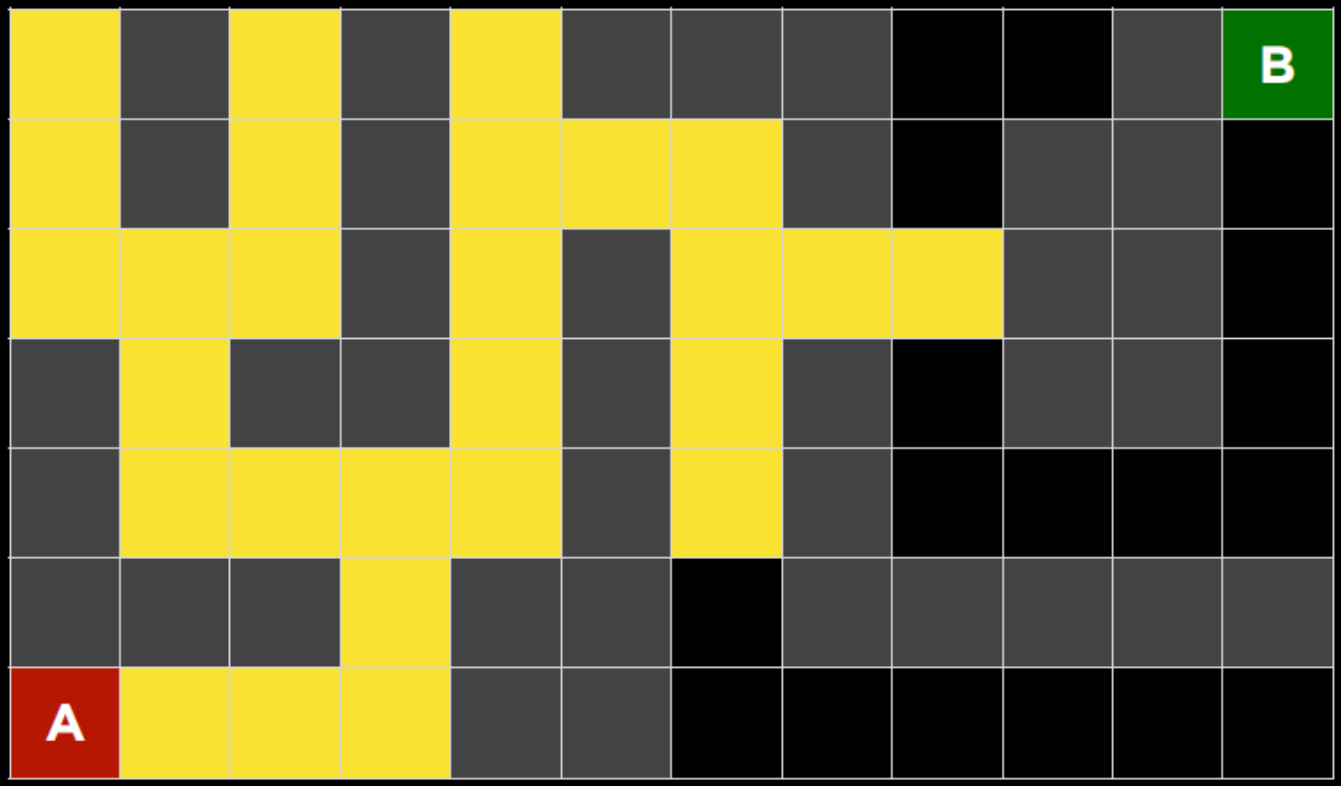


Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu



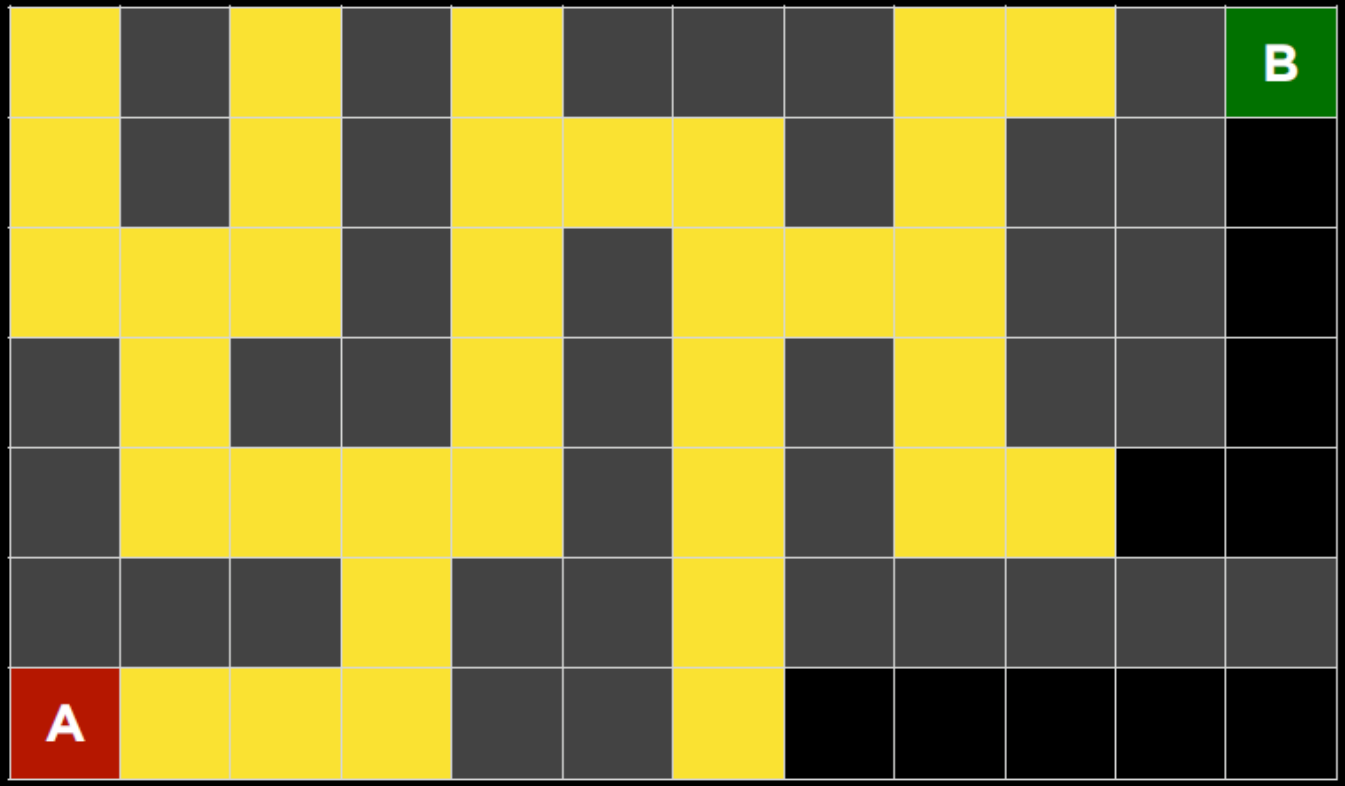
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Breadth-First Search



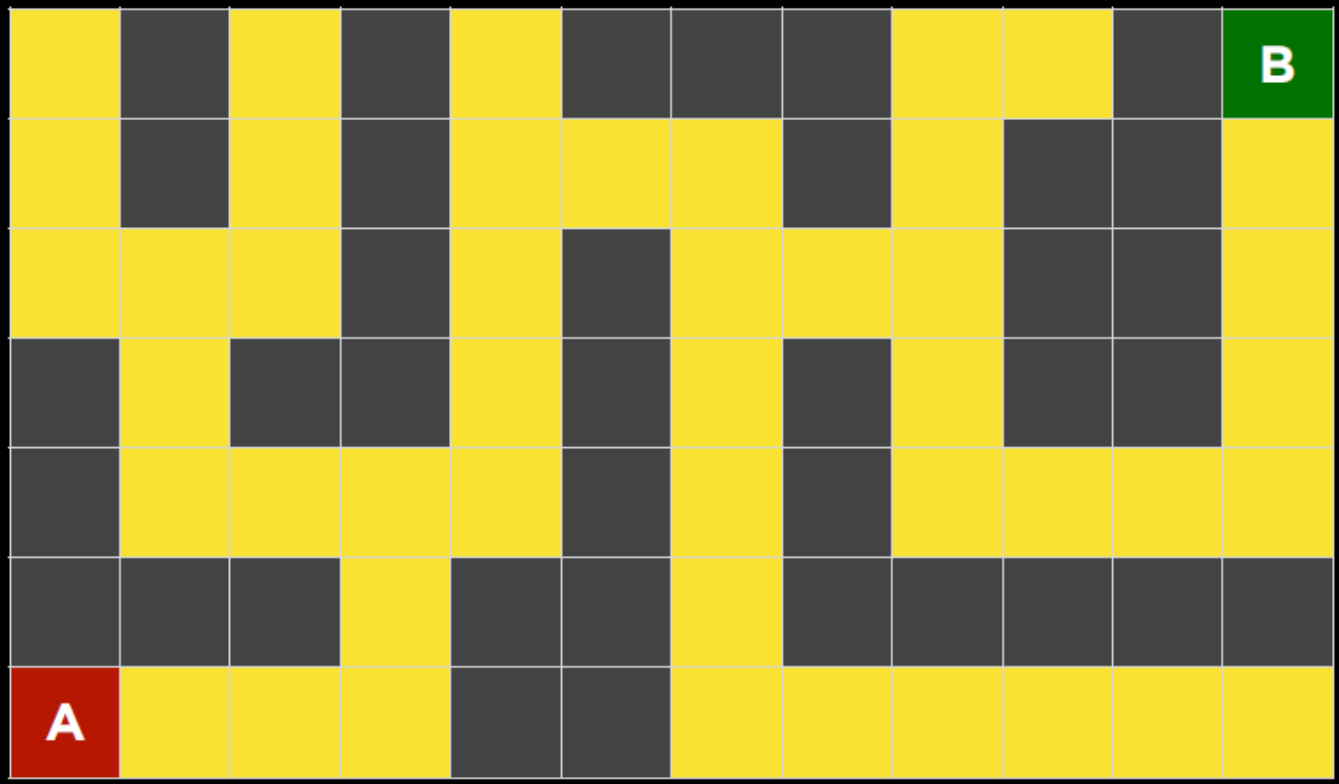
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Breadth-First Search



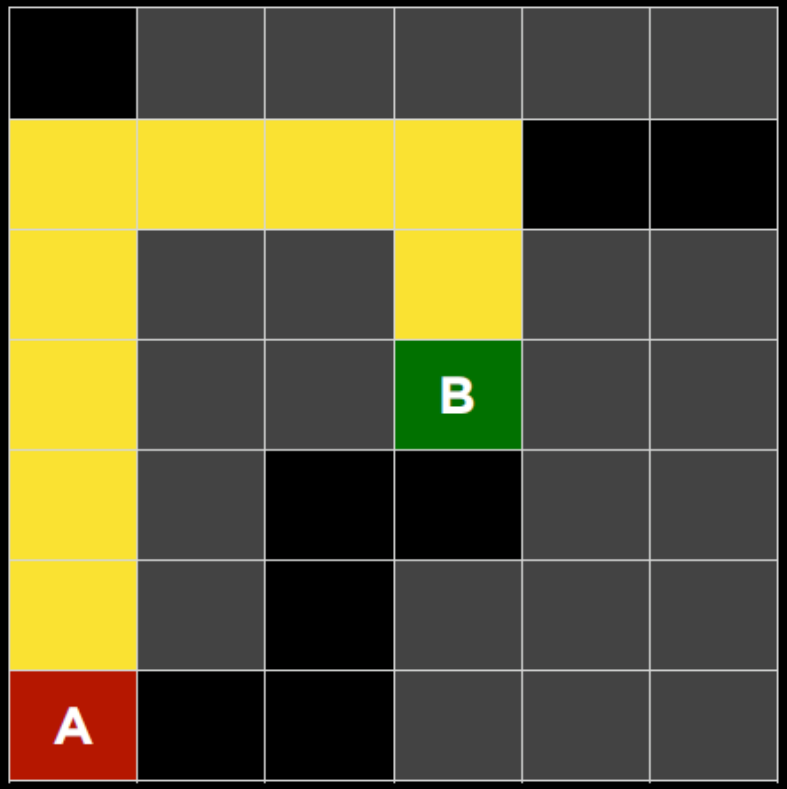
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Breadth-First Search



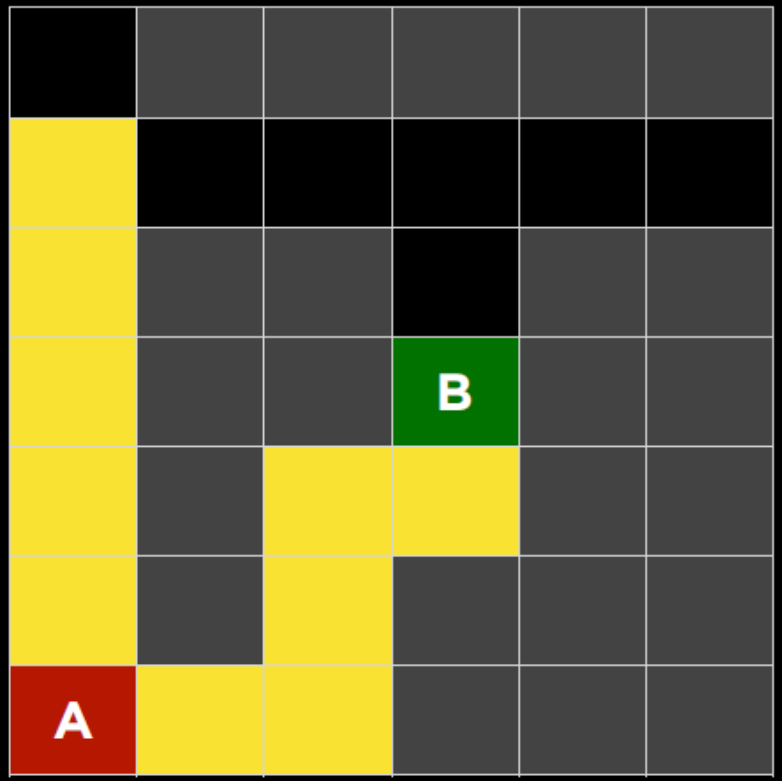
Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Depth-First Search



Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Breadth-First Search



Slide credit : HarvardX CS50AICS50's Introduction to Artificial Intelligence with Python, David J. Malan and Brian Yu

Analysis of Search strategies

- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
- 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 ∞)

Properties of breadth-first search

- Complete? Yes (if b is finite)
- Time?

  Number of nodes in a b-ary tree of depth d:

  (d is the depth of the optimal solution)

  $1+b+b^2+b^3+\ldots+b^d = O(b^d)$
- Space? $O(b^d)$ (keeps every node in memory)
- Optimal? Yes (if cost = 1 per step)
- Space is the bigger problem (more than time)

Depth	Nodes	Time	Memory
0	1	1 millisecond	100 kbytes
2	111	0.1 second	11 kilobytes
4	11,111	11 seconds	1 megabyte
6	10^6	18 minutes	111 megabytes
8	10^8	31 hours	11 gigabytes
10	10^{10}	128 days	1 terabyte
12	10^{12}	35 years	111 terabytes
14	10^{14}	3500 years	11,111 terabytes

Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
    → complete in finite spaces
- Time?

  Could be the time to reach a solution at maximum depth m: $O(b^m)$

  Terrible if m is much larger than d

  But if there are lots of solutions, may be much faster than BFS
- Space? *O(bm)*, i.e., linear space!
- Optimal? No - – returns the first solution it finds

Depth-limited search

= depth-first search with depth limit *l*,
i.e., nodes at depth *l* have no successors

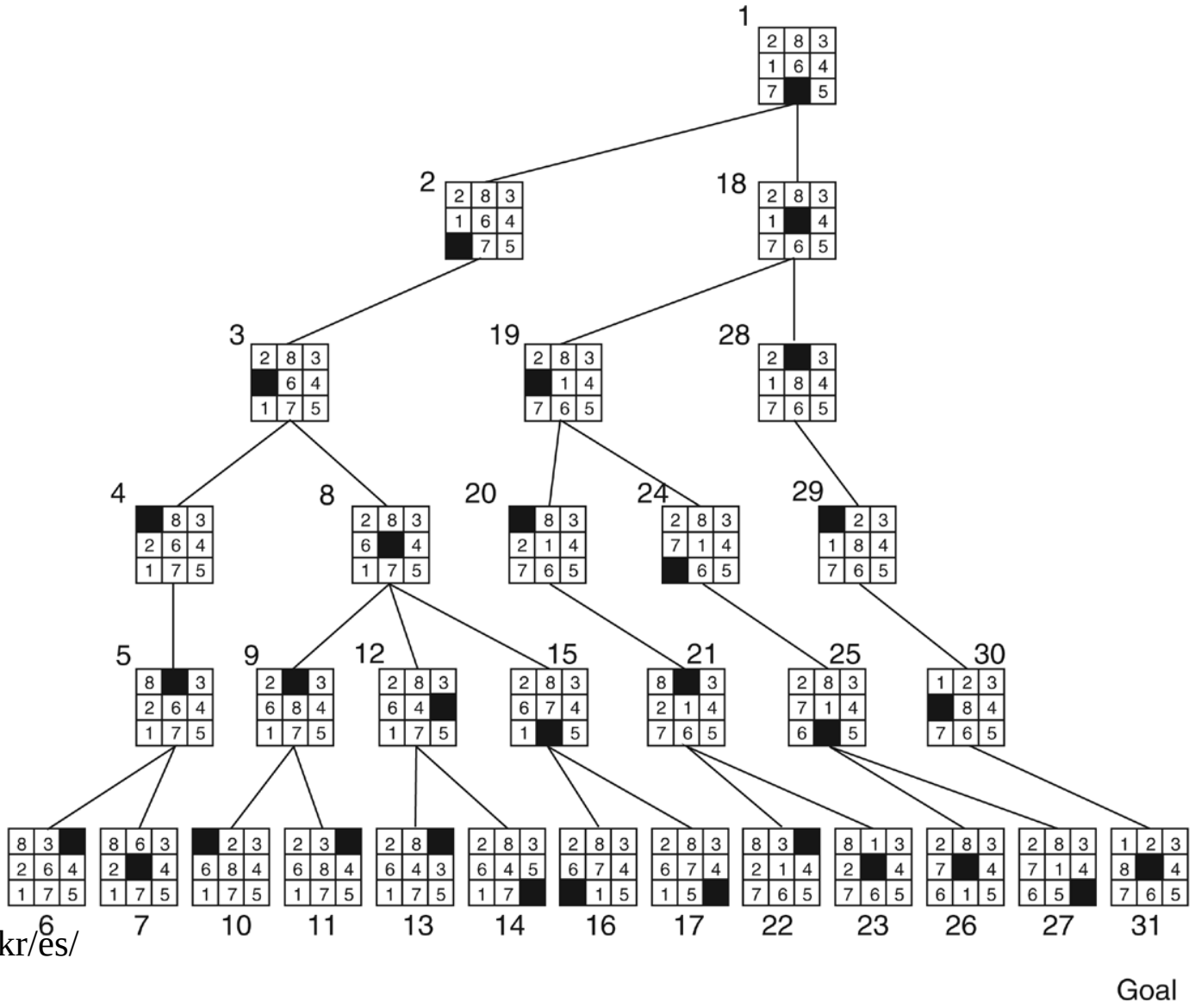
- Recursive implementation:

```
function DEPTH-LIMITED-SEARCH(problem, limit) returns soln/fail/cutoff
    RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS(node, problem, limit) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST[problem](STATE[node]) then return SOLUTION(node)
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND(node, problem) do
        result ← RECURSIVE-DLS(successor, problem, limit)
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```

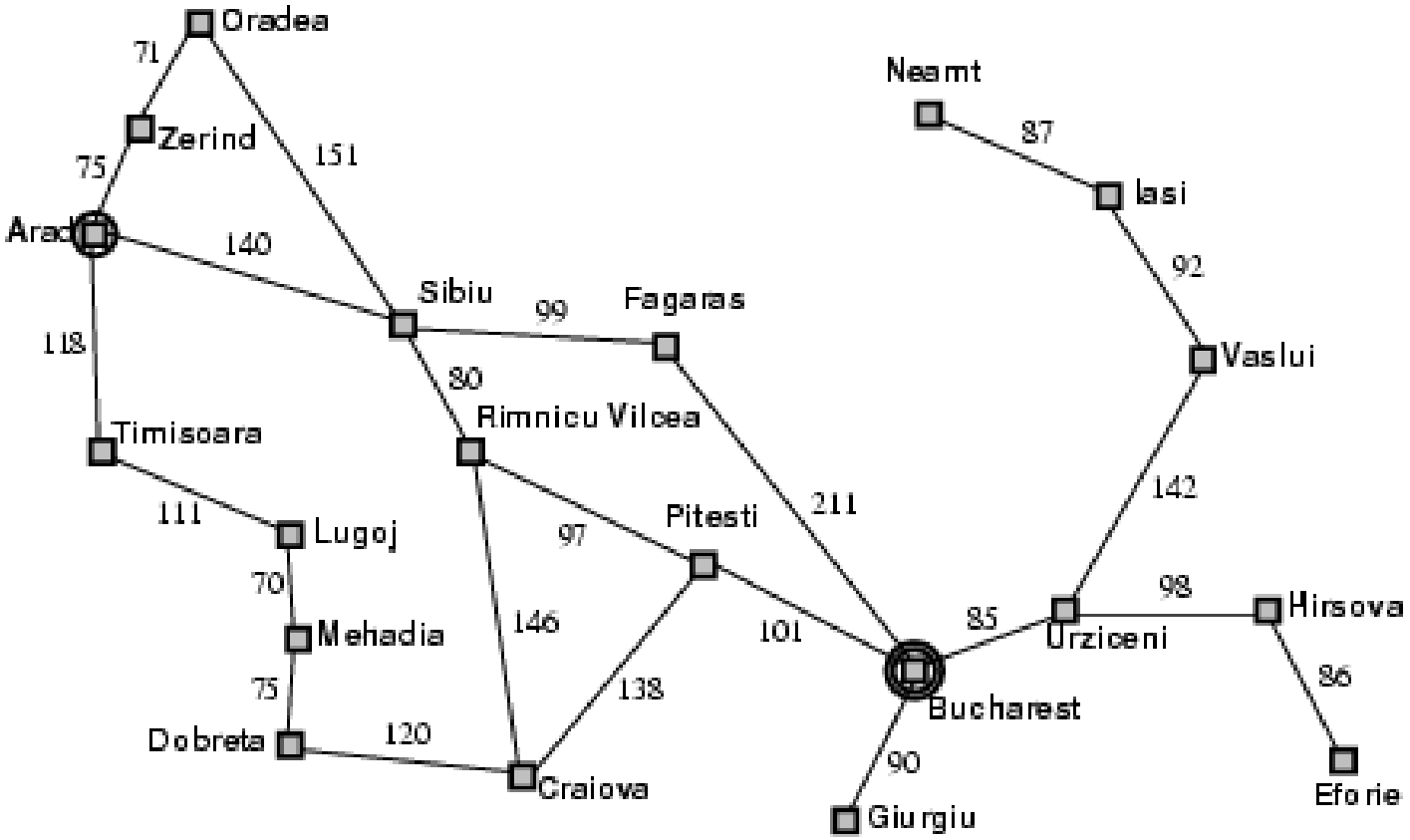
Depth limited search

Depth-first search of the 8-puzzle with a depth bound of 5.



Taken from http://iis.kaist.ac.kr/es/

Example: Romania



Depth limited search

- Like depth-first search, except:
  - Depth limit in the expand functin

L = depth limit

Complete if $L \geq d$ (d is the depth of the shallowest goal)

Not optimal (even if one continues the search after the first solution has been found, because an optimal solution may not be within the depth limit L)

$O(b^L)$ time

O(bL) space

Iterative deepening search

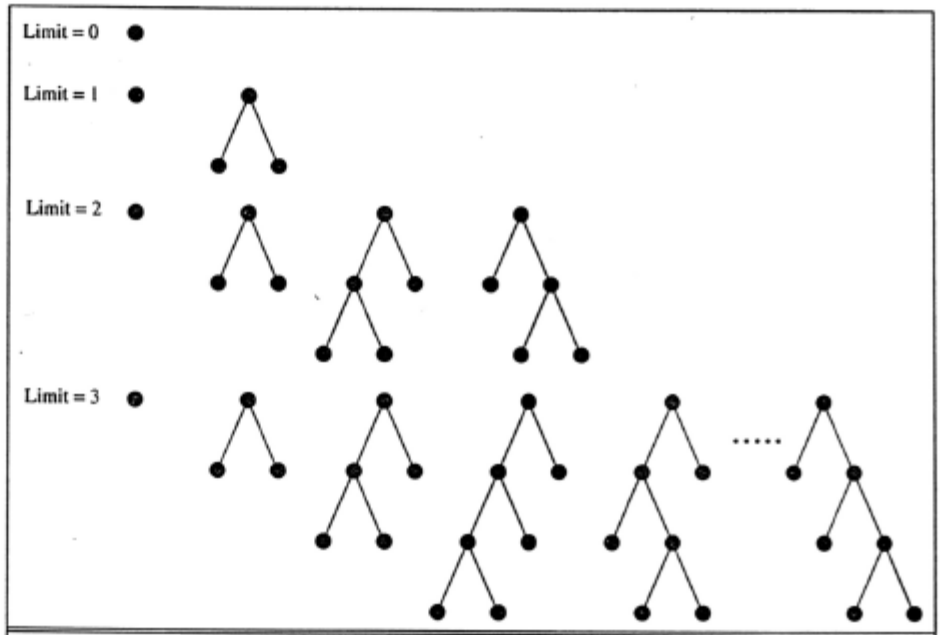
- Use DFS as a subroutine
  1. Check the root
  2. Do a DFS searching for a path of length 1
  3. If there is no path of length 1, do a DFS searching for a path of length 2
  4. If there is no path of length 2, do a DFS searching for a path of length 3…

Iterative deepening search

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

    for depth ← 0 to ∞ do
      if DEPTH-LIMITED-SEARCH(problem, depth) succeeds then return its result
    end
    return failure
```

Depth-first search for each depth.



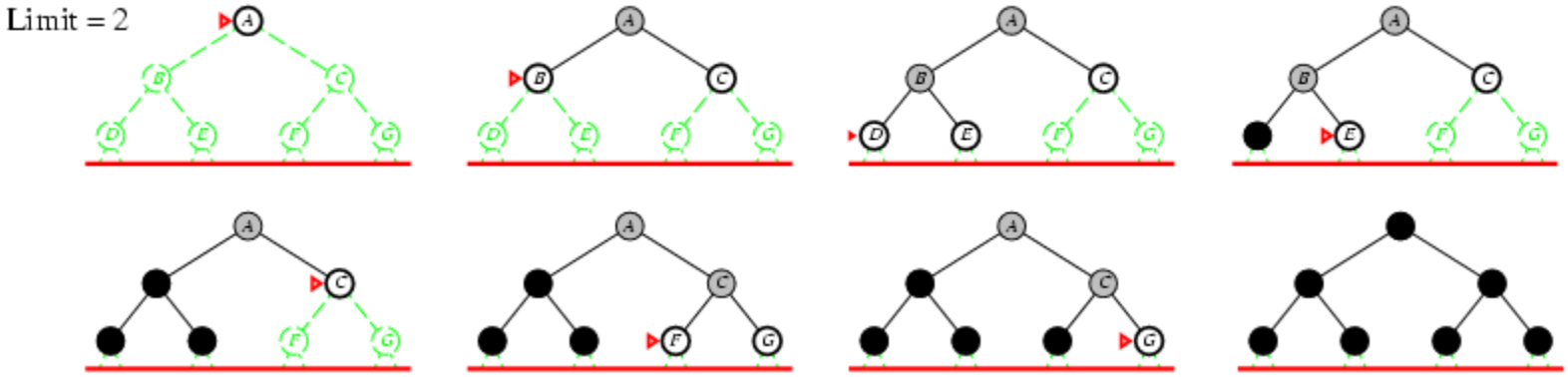
Four iterations of iterative deepening search on a binary tree.

Iterative deepening search l =0

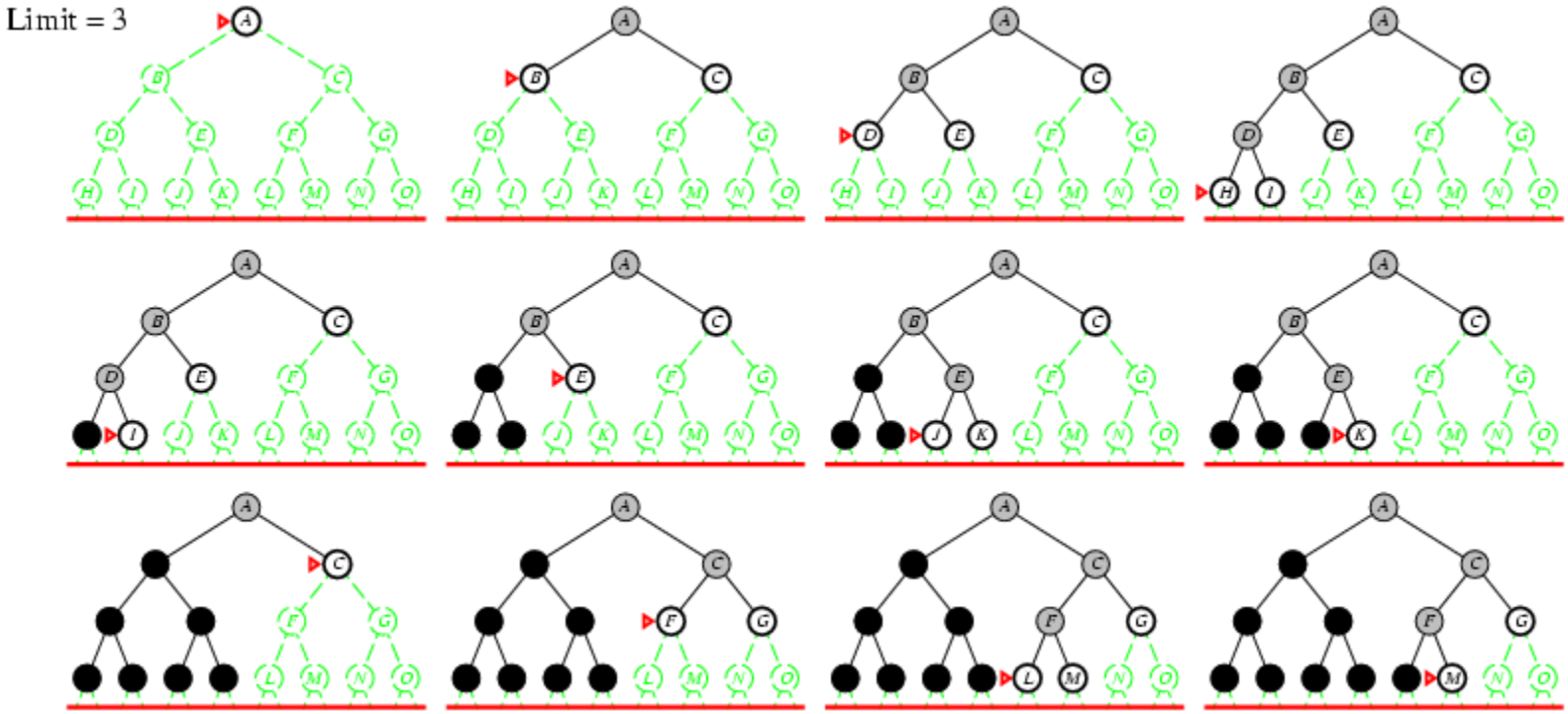
Limit = 0

Iterative deepening search l =1

Iterative deepening search l =2



Iterative deepening search l =3



Iterative deepening search

- Number of nodes generated in a depth-limited search to depth *d* with branching factor *b*:

$$N_{DLS} = b^0 + b^1 + b^2 + \ldots + b^{d-2} + b^{d-1} + b^d$$

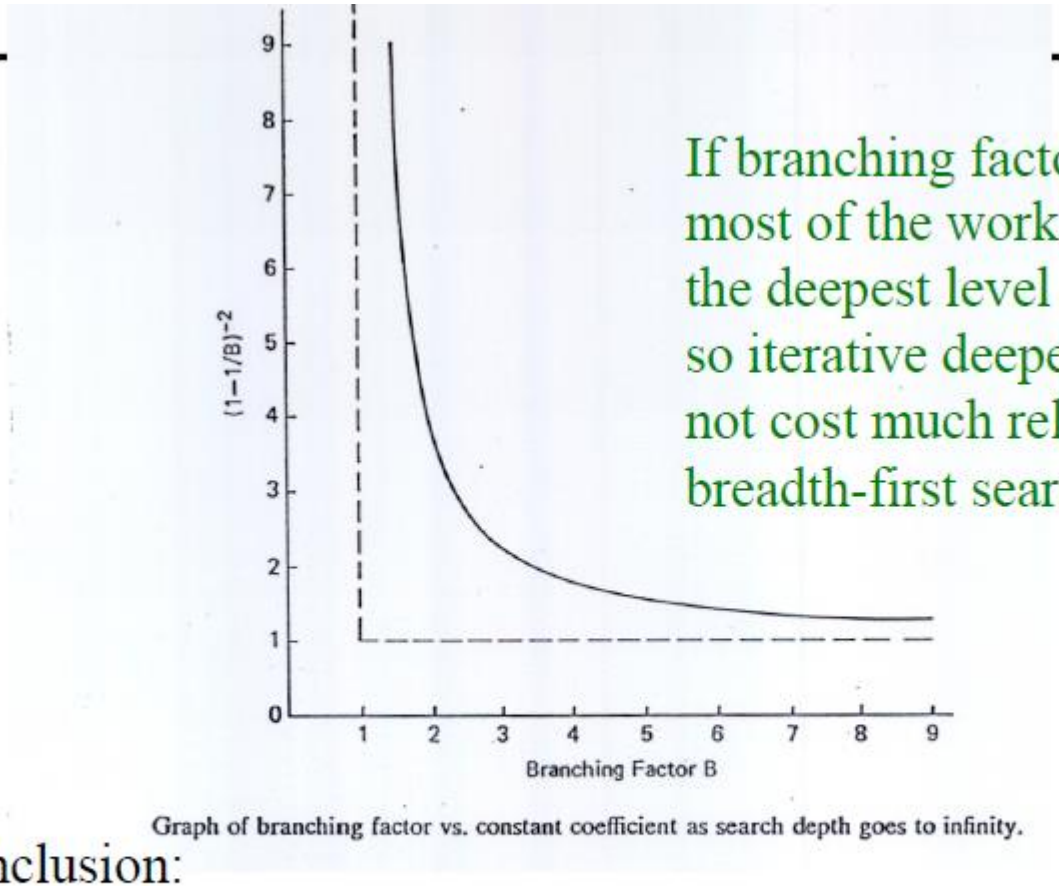
- Number of nodes generated in an iterative deepening search to depth *d* with branching factor *b*:

$$N_{IDS} = (d+1)b^0 + d\,b\text{^}1 + (d-1)b\text{^}2 + \ldots + 3b^{d-2} + 2b^{d-1} + 1b^d$$

- For *b = 10*, *d = 5*,
  - $N_{DLS} = 1 + 10 + 100 + 1{,}000 + 10{,}000 + 100{,}000 = 111{,}111$
  - $N_{IDS} = 6 + 50 + 400 + 3{,}000 + 20{,}000 + 100{,}000 = 123{,}456$

- Overhead = (123,456 - 111,111)/111,111 = 11%

Iterative deepening search



Graph of branching factor vs. constant coefficient as search depth goes to infinity.

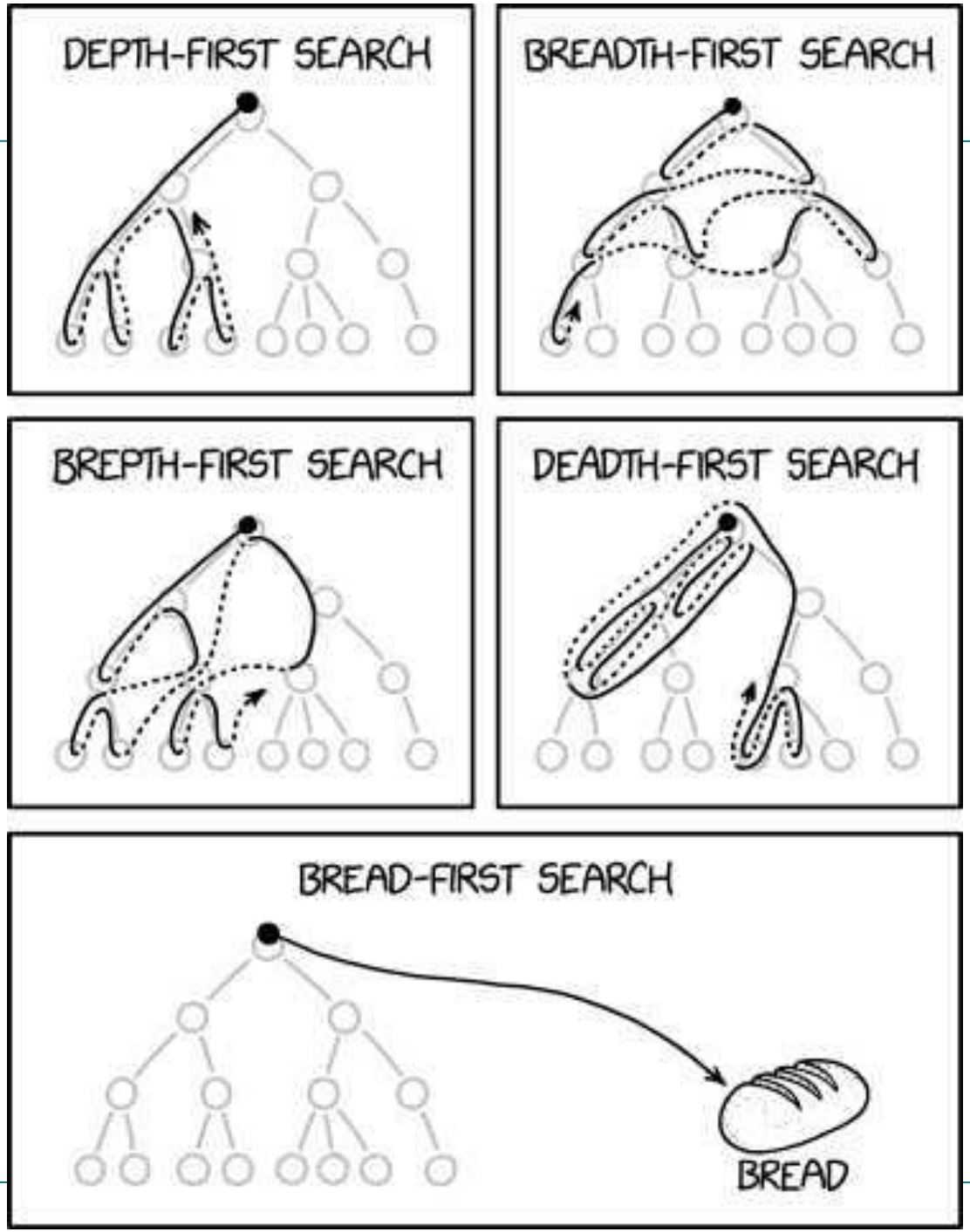
If branching factor is *large*, most of the work is done at the deepest level of search, so iterative deepening does not cost much relatively to breadth-first search.

Conclusion:

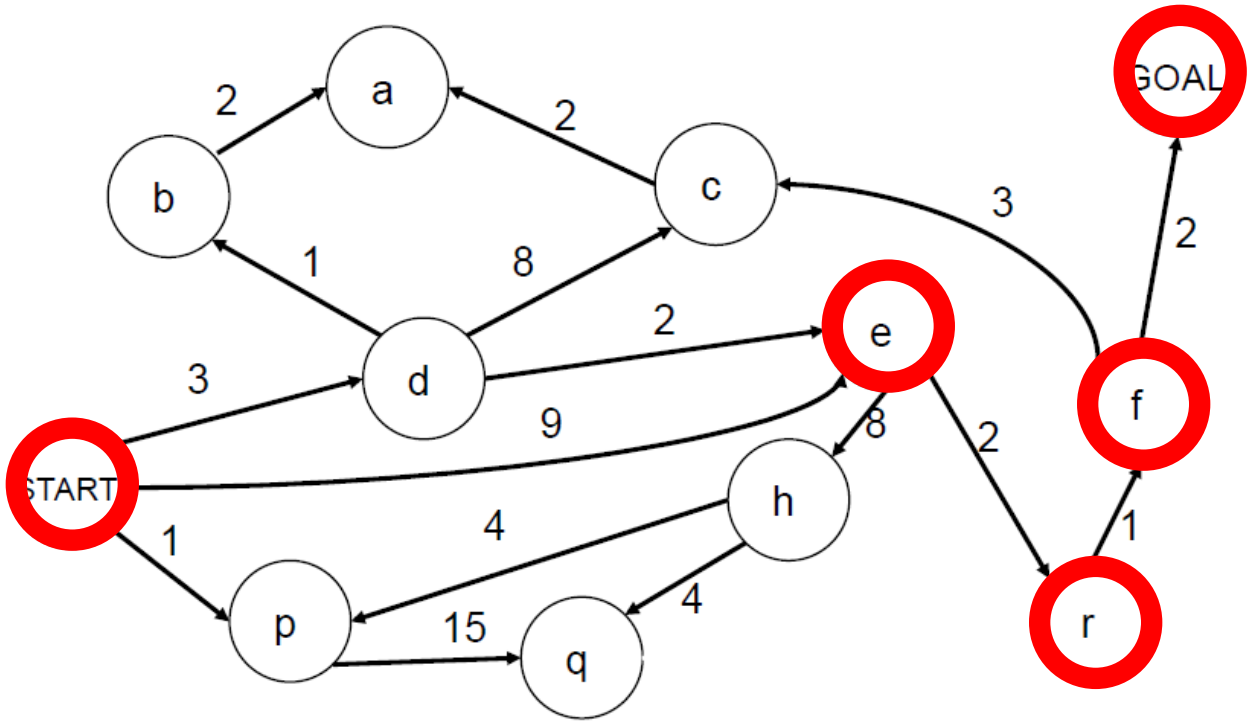
- Iterative deepening is preferred when search space is large and depth of (optimal) solution is unknown
- Not preferred if branching factor is tiny (near 1)

Properties of iterative deepening search

- Complete? Yes
- Time? $(d+1)b^0 + d\,b^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$
- Space? $O(bd)$
- Optimal? Yes, if step cost = 1


DEPTH-FIRST SEARCH
BREADTH-FIRST SEARCH
BREPTH-FIRST SEARCH
DEADTH-FIRST SEARCH
BREAD-FIRST SEARCH
BREAD

Search with varying step costs



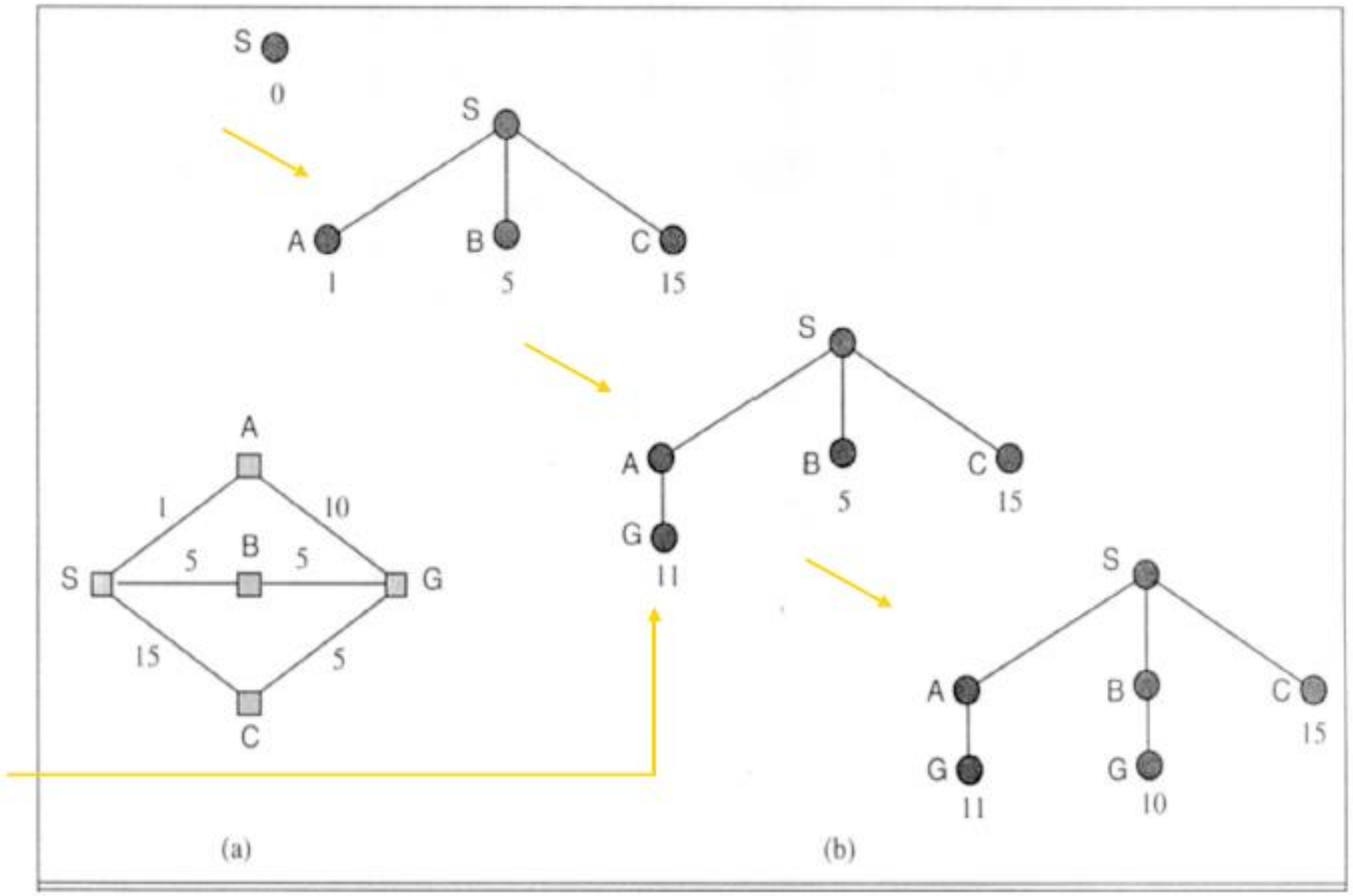
- BFS finds the path with the fewest steps, but does not always find the cheapest path

Uniform-cost search

- For each frontier node, save the total cost of the path from the initial state to that node
- Expand the frontier node with the lowest path cost
- Implementation: *frontier* is a priority queue ordered by path cost
- Equivalent to breadth-first if step costs all equal
- Equivalent to Dijkstra's algorithm in general

Uniform-cost search

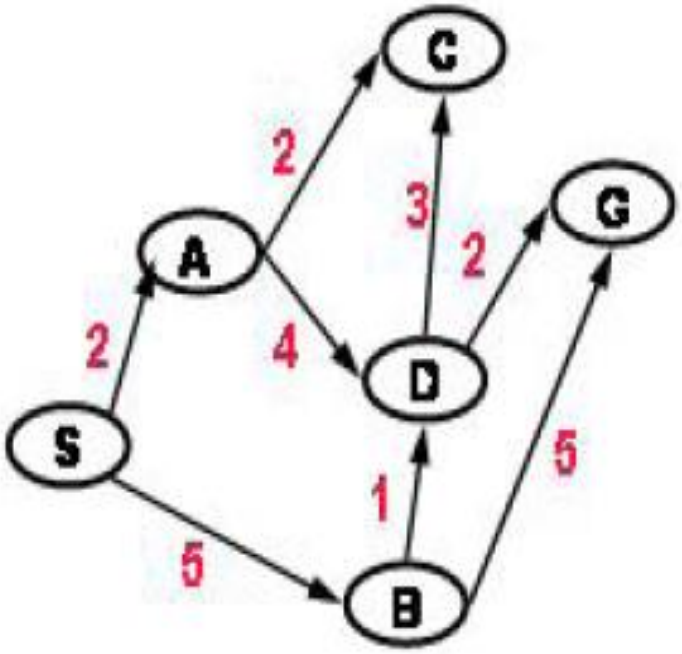
Insert nodes onto open list in ascending order of **cost** of path to root.



Taken from http://www.cs.nott.ac.uk/~gxk/courses/g5aiai/003blindsearches/blind_searches.htm

Uniform cost search

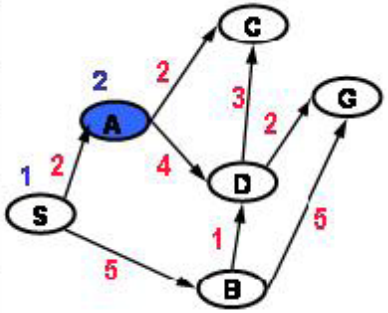
- Each link has a length or cost (which is always greater than 0)
- We want shortest or least cost path



Total path cost:	
(S A C)	4
(S B D G)	8
(S A D C)	9

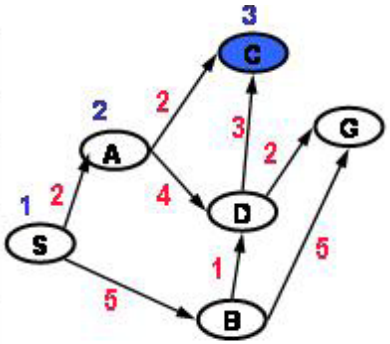
Uniform cost search

	Q
1	(0 S)
2	(2 A S) (5 B S)



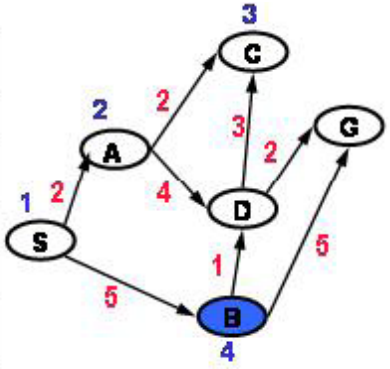
Uniform cost search

	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)



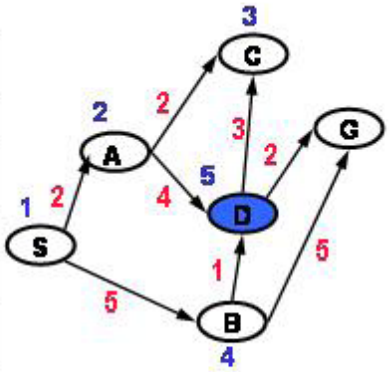
Uniform cost search

	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)



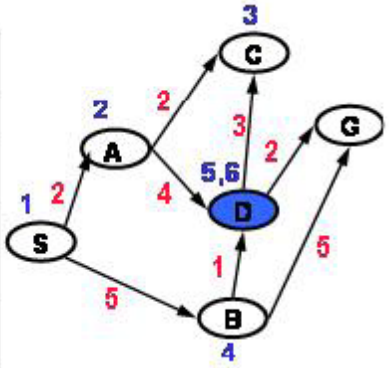
Uniform cost search

	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)



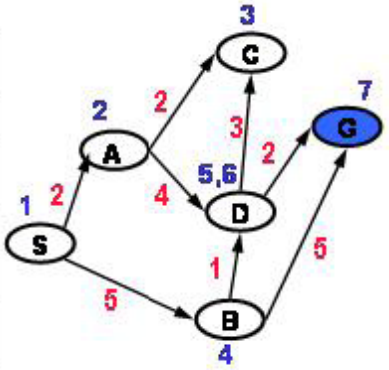
Uniform cost search

	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) (6 D A S)



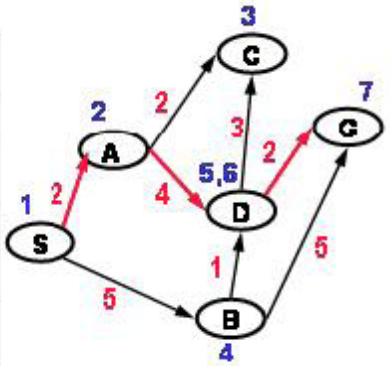
Uniform cost search

	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) (6 D A S)
7	(8 G D A S) (9 C D A S) (8 G D B S) (9 C D B S) (10 G B S)



Uniform cost search

	Q
1	(0 S)
2	(2 A S) (5 B S)
3	(4 C A S) (6 D A S) (5 B S)
4	(6 D A S) (5 B S)
5	(6 D B S) (10 G B S) (6 D A S)
6	(8 G D B S) (9 C D B S) (10 G B S) (6 D A S)
7	(8 G D A S) (9 C D A S) (8 G D B S) (9 C D B S) (10 G B S)

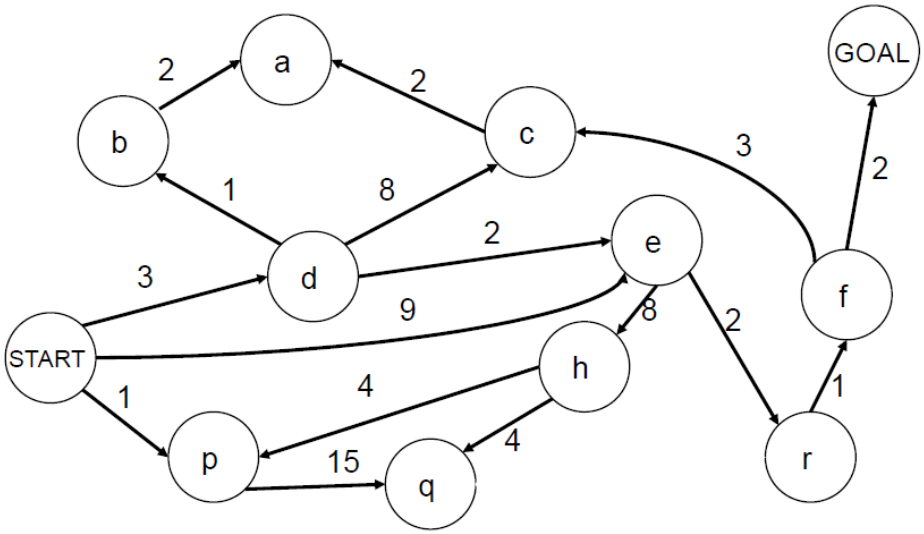


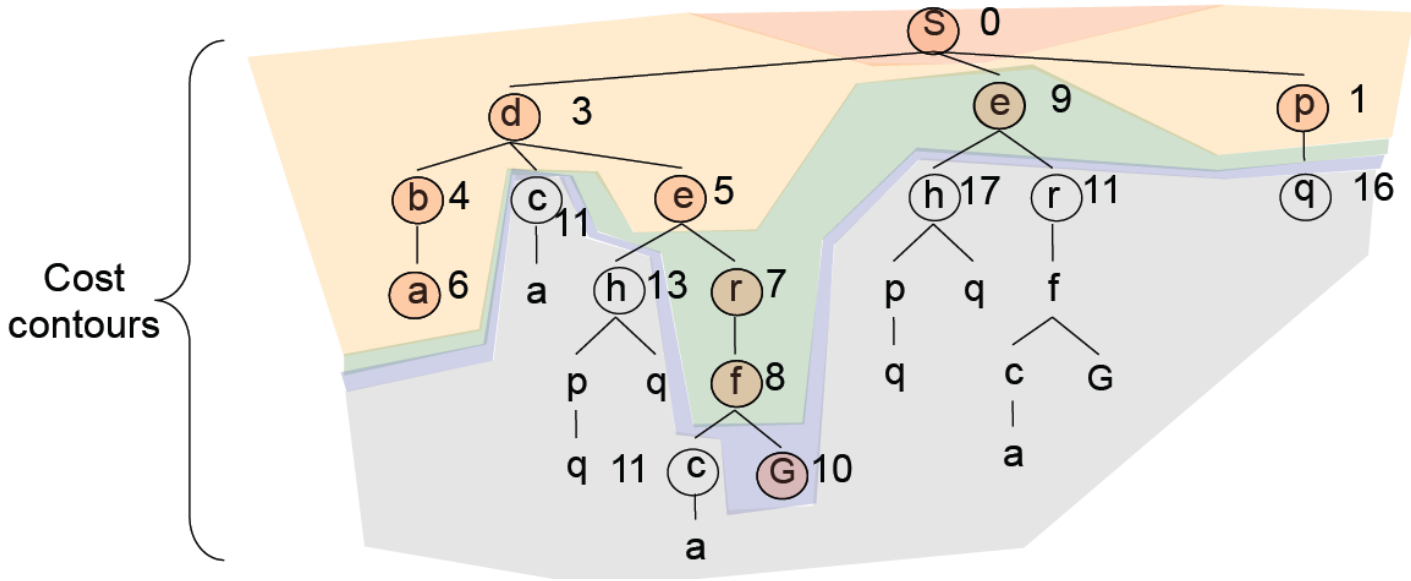
Why not stop on the first goal

- When doing Uniform Cost, it is not correct to stop the search when the first path to a goal is generated, that is, when a node whose state is a goal is added to Q.
- We must wait until such a path is pulled off the Q and tested in step 3. It is only at this point that we are sure it is the shortest path to a goal since there are no other shorter paths that remain unexpanded.
- This contrasts with the Any Path searches where the choice of where to test for a goal was a matter of convenience and efficiency, not correctness.
- In the previous example, a path to G was generated at step 5, but it was a different, shorter, path at step 7 that we accepted.

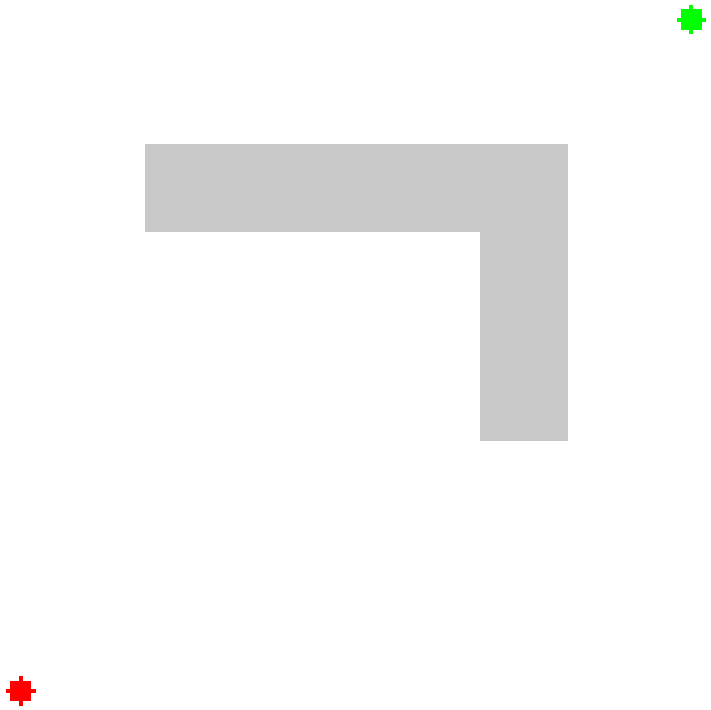
Uniform-cost search example

- Expansion order:
  (S,p,d,b,e,a,r,f,e,G)





Another example of uniform-cost search



Wikipedia

Properties of uniform-cost search

- Expand least-cost unexpanded node
- Implementation:
  - *fringe* = queue ordered by path cost
- Equivalent to breadth-first if step costs all equal

- **Complete?**

  Yes, if step cost is greater than some positive constant ε (we don't want infinite sequences of steps that have a finite total cost)

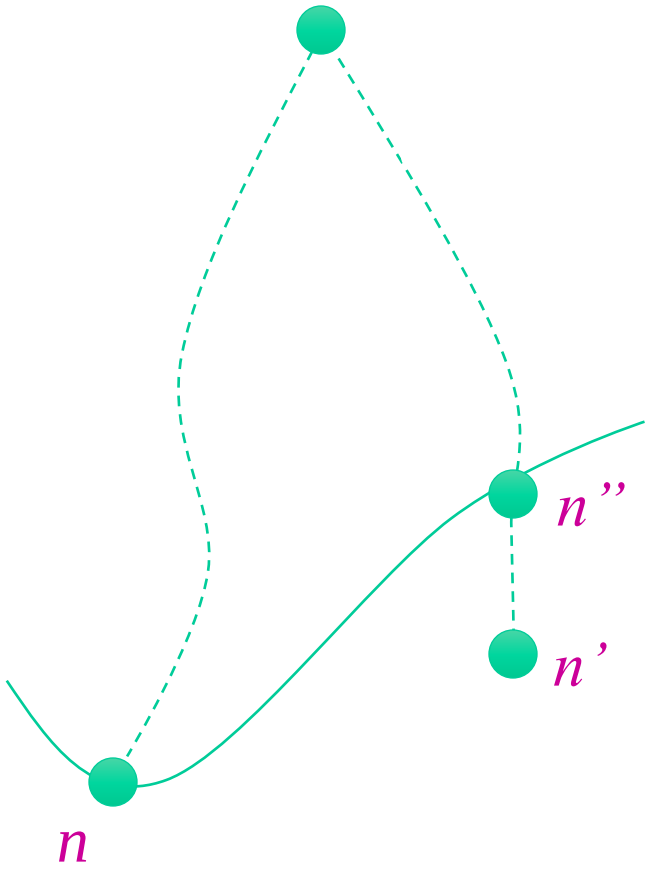
- **Optimal?**

  Yes

Optimality of uniform-cost search

- **Graph separation property**: every path from the initial state to an unexplored state has to pass through a state on the frontier
  - Proved inductively

- Optimality of UCS: proof by contradiction
  - Suppose UCS terminates at goal state n with path cost $g(n)$ but there exists another goal state n' with $g(n') < g(n)$
  - By the graph separation property, there must exist a node n'' on the frontier that is on the optimal path to n'
  - But because $g(n'') \leq g(n') < g(n)$, n'' should have been expanded first!



Uniform-cost search

- Complete? Yes, if step cost $\geq \varepsilon$
- Time? # of nodes with *pathn cost* $g \leq$ cost of optimal solution,

  $O(b^{ceiling(C^*/\varepsilon)})$ where C^* is the cost of the optimal solution
- This can be greater than $O(b^d)$: the search can explore long paths consisting of small steps before exploring shorter paths consisting of larger steps
- Space? # of nodes with $g \leq$ cost of optimal solution, $O(b^{ceiling(C^*/\varepsilon)})$
- Optimal? Yes – nodes expanded in increasing order of $g(n)$

Review: Uninformed search strategies

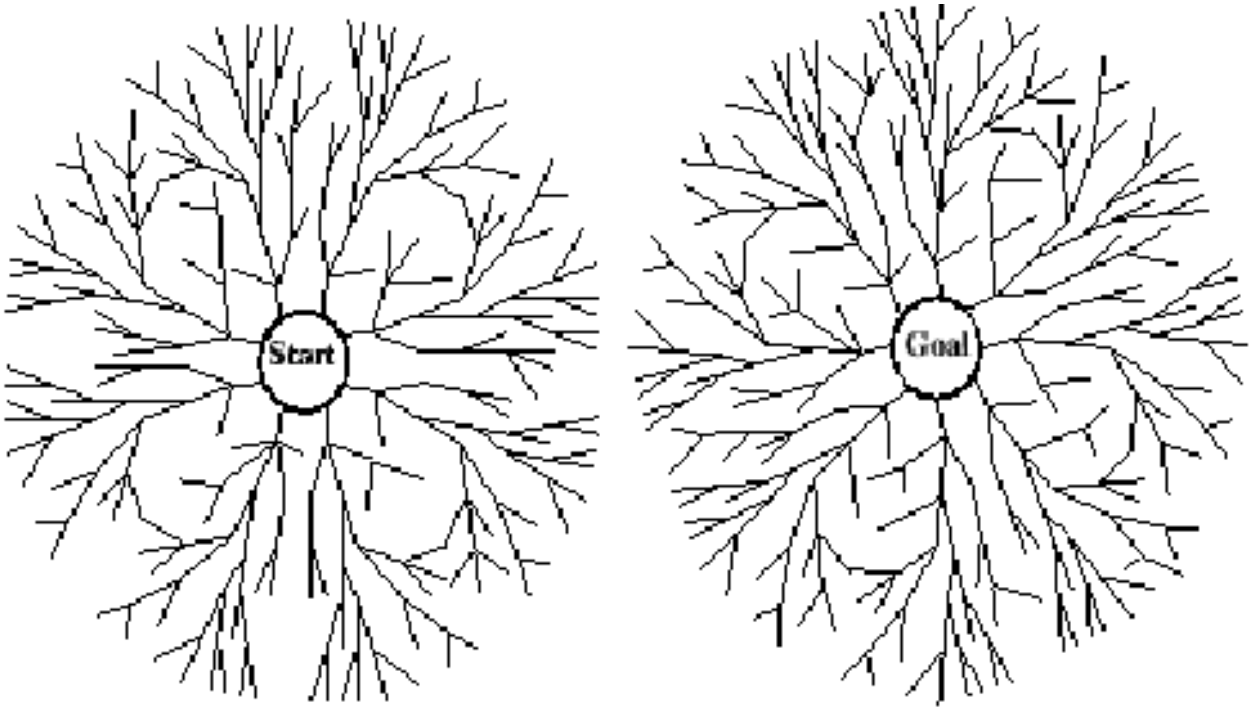
Algorithm	Complete?	Optimal?	Time complexity	Space complexity
BFS	Yes	If all step costs are equal	$O(b^d)$	$O(b^d)$
DFS	No	No	$O(b^m)$	$O(bm)$
IDS	Yes	If all step costs are equal	$O(b^d)$	$O(bd)$
UCS	Yes	Yes	Number of nodes with $g(n) \leq C^*$	

b: maximum branching factor of the search tree
d: depth of the optimal solution
m: maximum length of any path in the state space
C*: cost of optimal solution
g(n): cost of path from start state to node n

Bidirectional search

- Run two simultaneous searches – one forward from the initial state, and the other backward from the goal, stopping when two searches meet in the middle

$b^{d/2} + b^{d/2} < b^d$



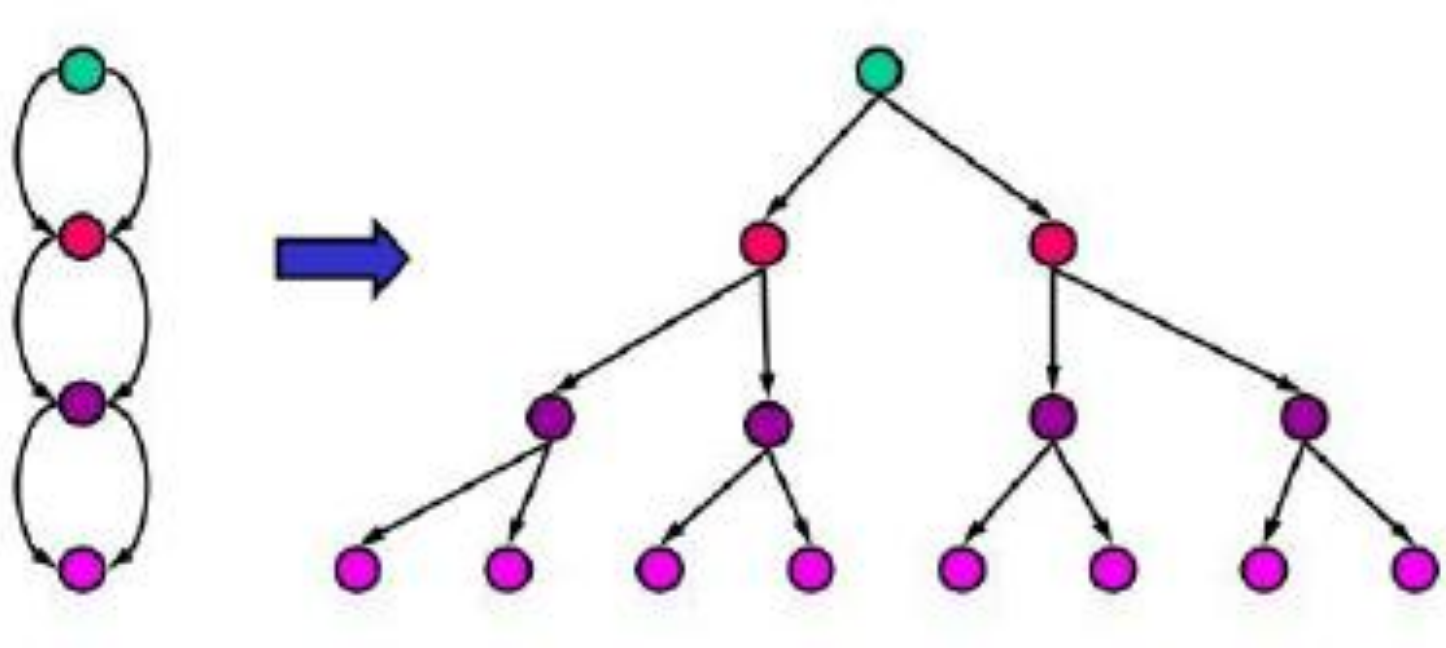
For b = 10, d= 6 , BFS 1,111, 111 nodes, BS 2,222 nodes

Bi-directional Search discussion

- Need to have operators that calculate **predecessors**.
- Need a way to check that **states meet (are the same)**
- Efficient way to check when searches meet: hash table
- Can use IDS or BFS or DFS in each half
- Optimal, complete, $O(b^{d/2})$ time.
- Still $O(b^{d/2})$ space (even with iterative deepening) because the nodes of at least one of the searches have to be stored to check matches.

Repeated states

- Failure to detect repeated states can turn a linear problem into an exponential one!



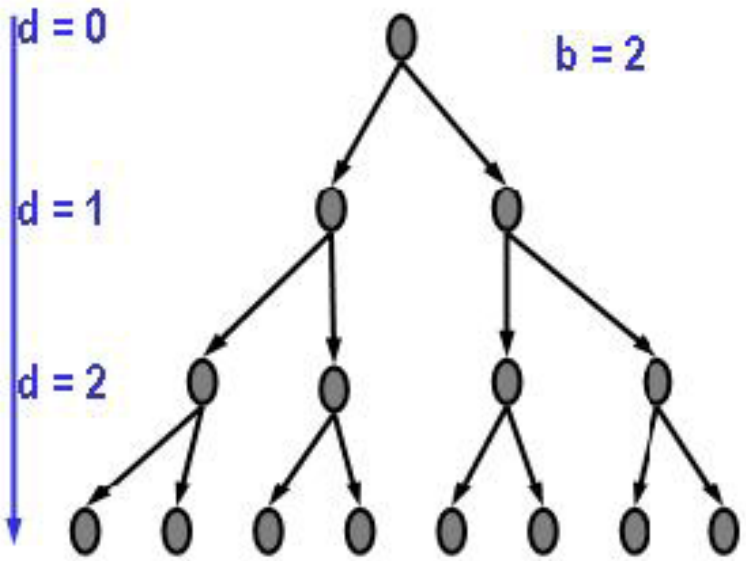
Handling repeated states

- Do not return to the state just you came from
- Do not create paths with cycles in them
- Do not generate any state that was ever generated before

Worst Case Running Time

Max Time ∝ Max #Visited

- The number of states in the search space may be exponential in some "depth" parameter, e.g. number of actions in a plan, number of moves in a game.
- All the searches, with or without visited list, may have to visit each state at least once, in the worst case.
- So, all searches will have worst case running times that are at least proportional to the total number of states and therefore exponential in the "depth" parameter.



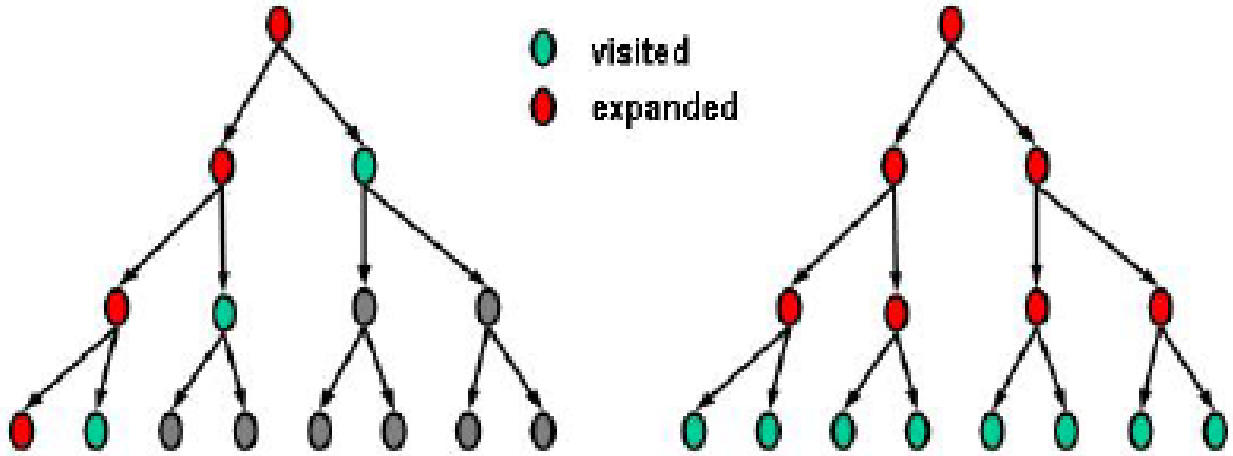
d is depth
b is branching factor

$b^d < (b^{d+1} - 1) / (b - 1) < b^{d+1}$

states in tree

Worst Case Space

Max Q size = Max (#Visited – #Expanded)



Depth First max Q size
$(b-1)d \approx bd$

Breadth First max Q size
b^d

Summary

- Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored
- Variety of uninformed search strategies
- Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Time	b^d	b^d	b^m	b^l	b^d	$b^{d/2}$
Space	b^d	b^d	bm	bl	bd	$b^{d/2}$
Optimal?	Yes	Yes	No	No	Yes	Yes
Complete?	Yes	Yes	No	Yes, if $l \geq d$	Yes	Yes

b = branching factor
d = depth of shallowest goal state
m = maximum depth of the search tree
l = depth limit of the algorithm

Classes of Search

Class	Name	Operation
Any Path Uninformed	**Depth-First Breadth-First**	Systematic exploration of whole tree until a goal node is found.
Any Path Informed	**Best-First**	Uses heuristic measure of goodness of a state, e.g. estimated distance to goal.
Optimal Uninformed	**Uniform-Cost**	Uses path "length" measure. Finds "shortest" path.
Optimal Informed	**A***	Uses path "length" measure and heuristic Finds "shortest" path