Uninformed Search

Search for a solution with no clue about the goal.

Outline

I. Search algorithms

II. Breadth-first search

III. Depth-first search

IV. Iterative deepening

V. Bi-directional searches

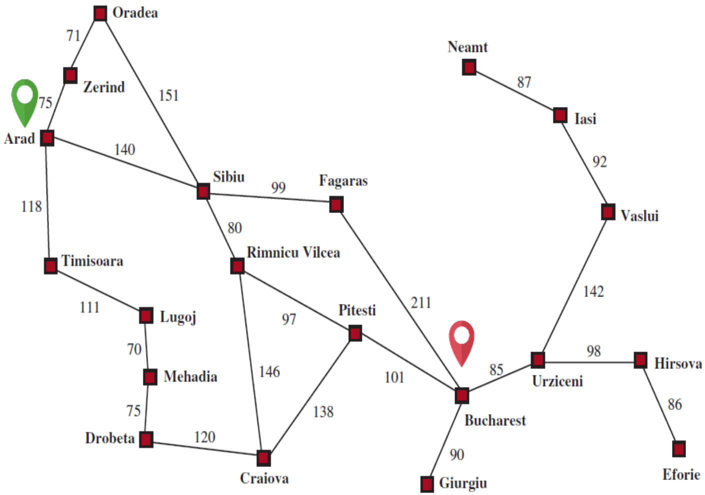
1975 ACM Turing Award Lecture:

Computer science as empirical inquiry: symbols and search

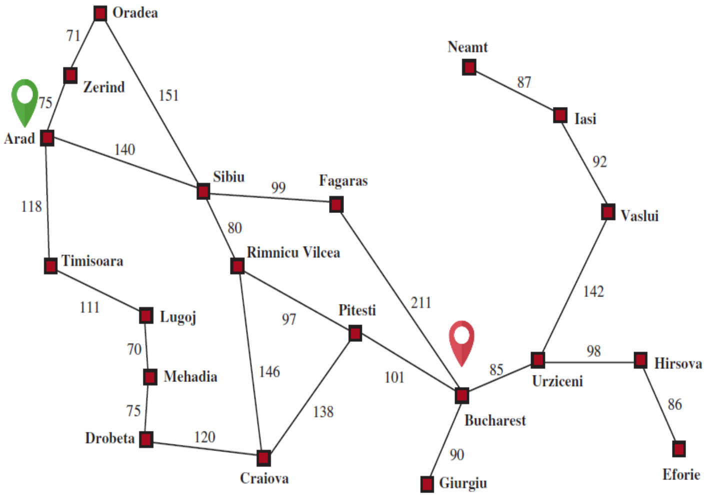
Allen Newell and Herbert Simon

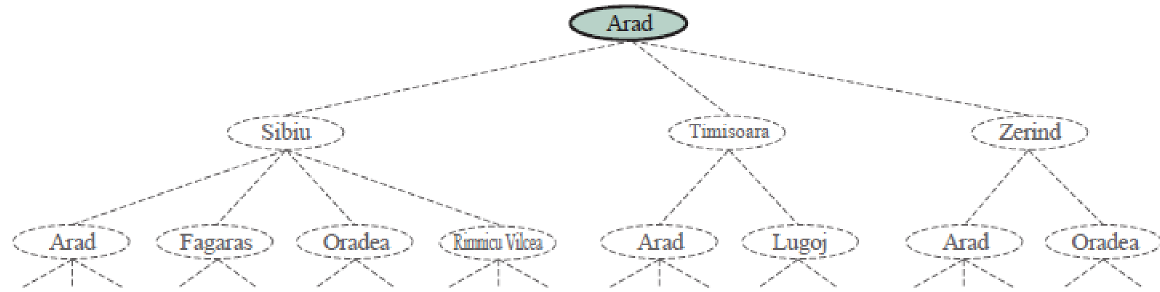
* Figures are from the textbook site (or drawn by the instructor) unless the source is specifically cited.

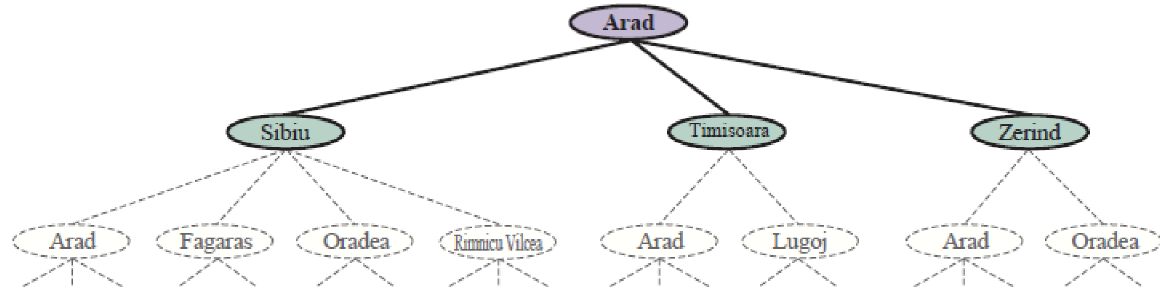
I. Tree Search

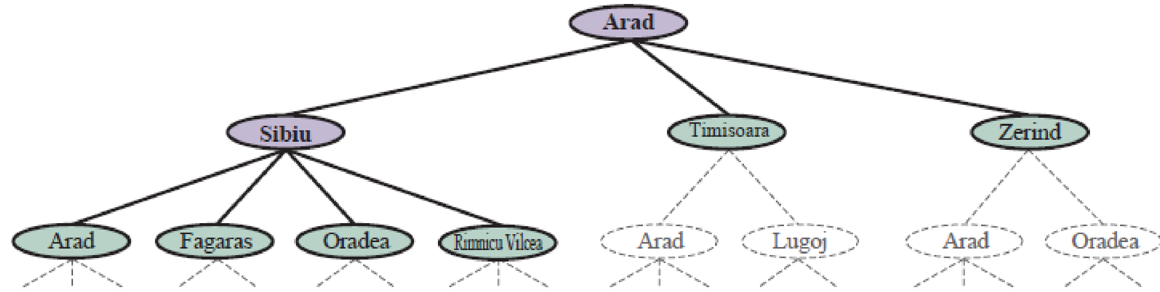


I. Tree Search

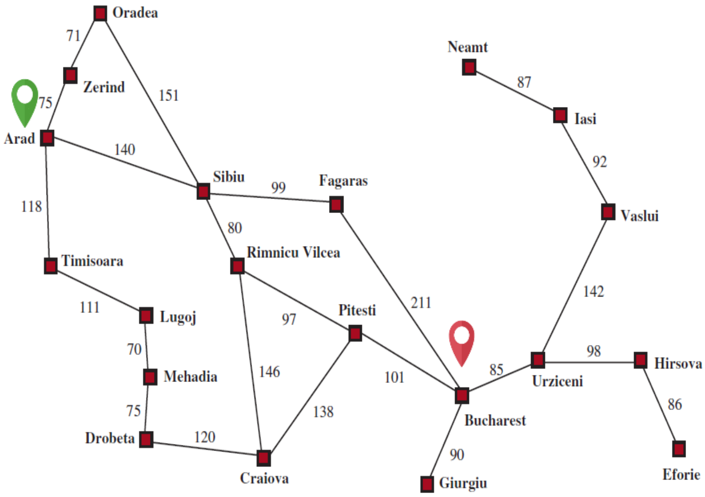


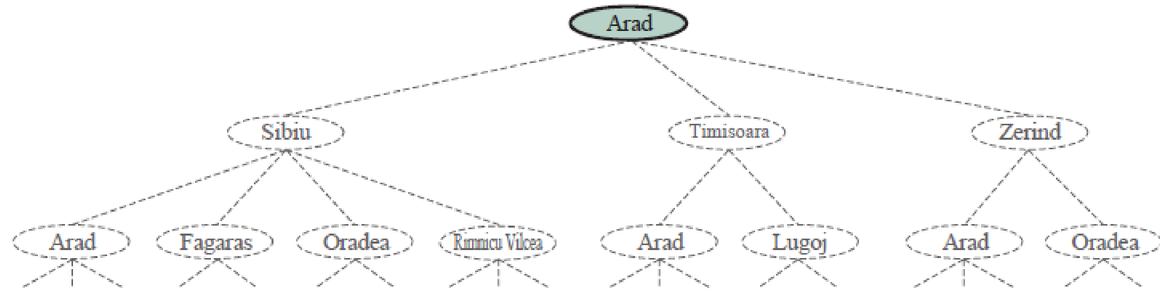


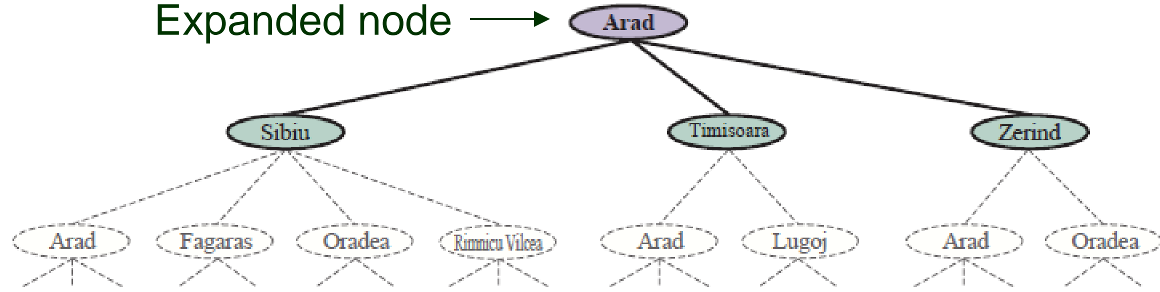


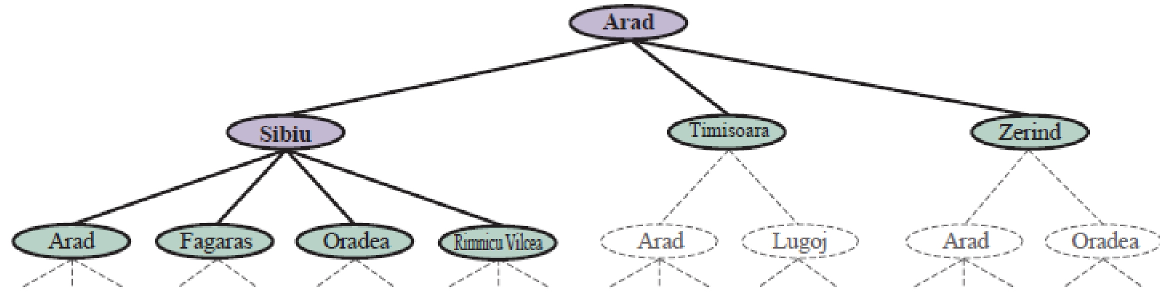


I. Tree Search

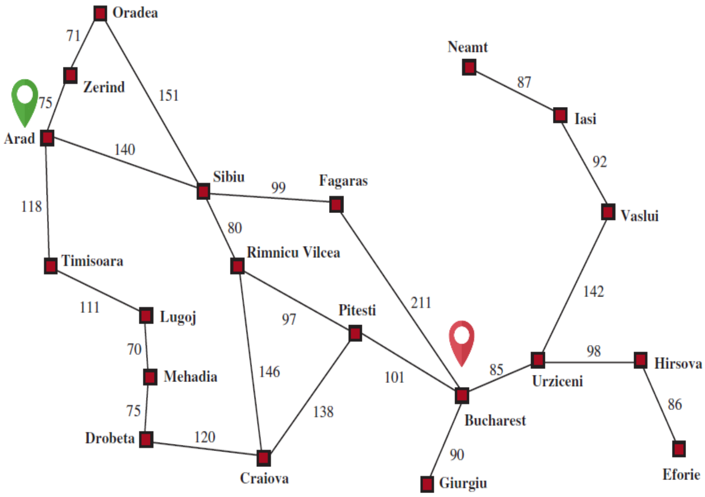


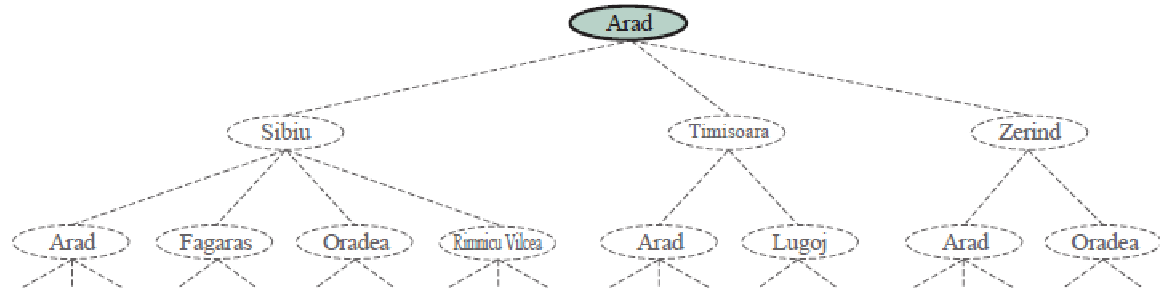






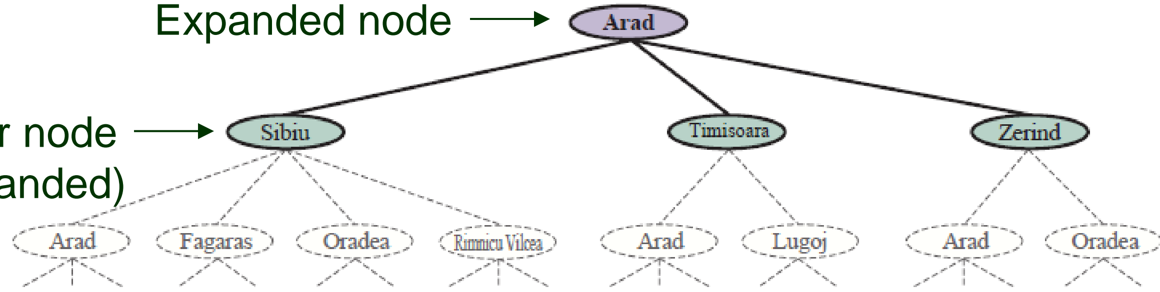
I. Tree Search

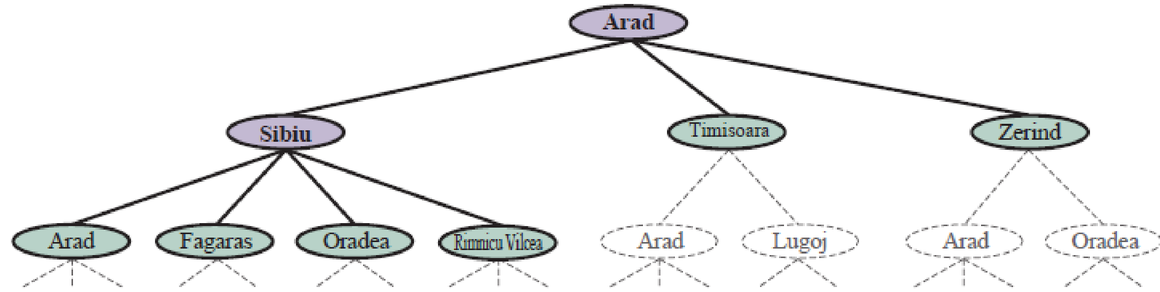




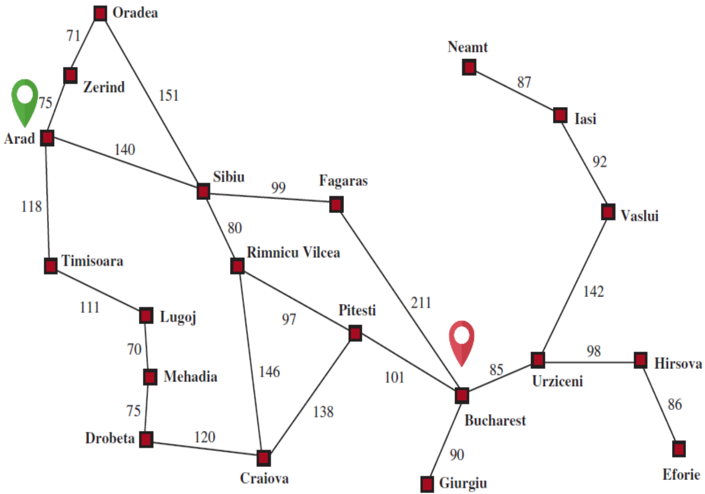
Expanded node

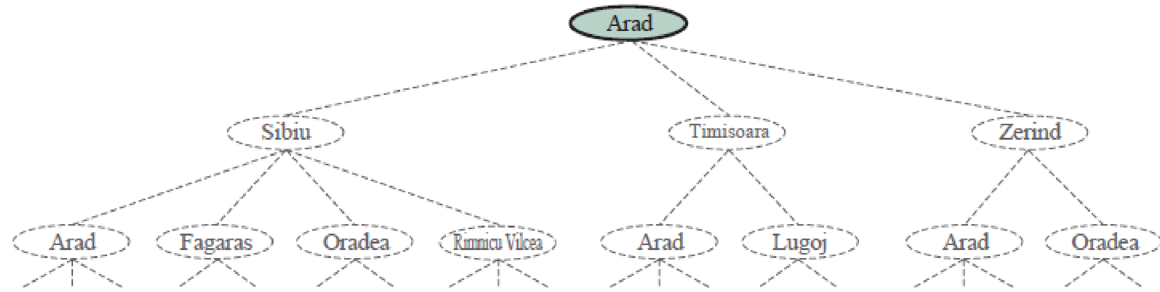
Frontier node (unexpanded)

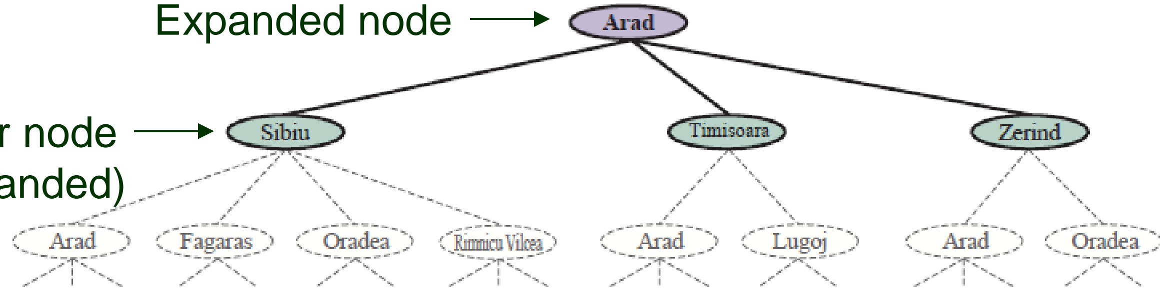




I. Tree Search



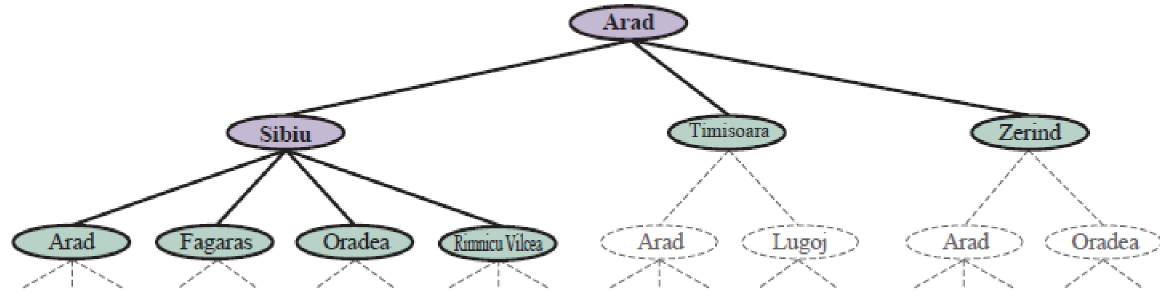




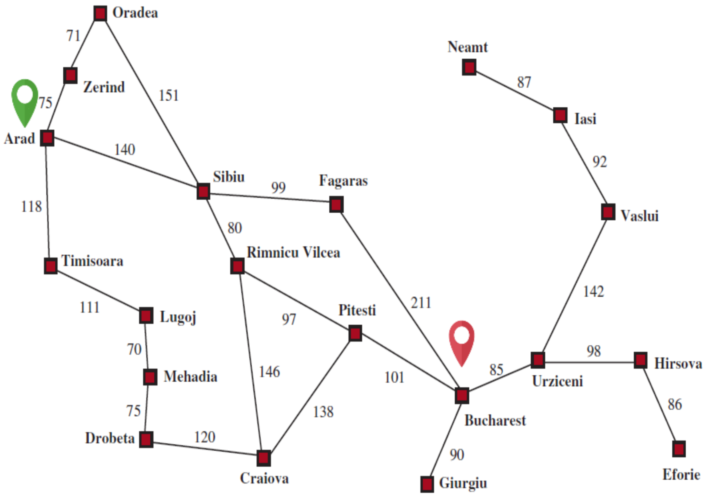
Expanded node

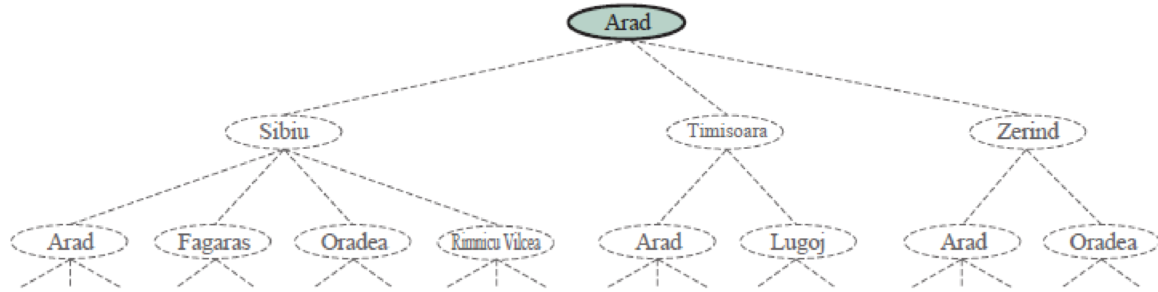
Frontier node (unexpanded)

Node to be generated next



I. Tree Search

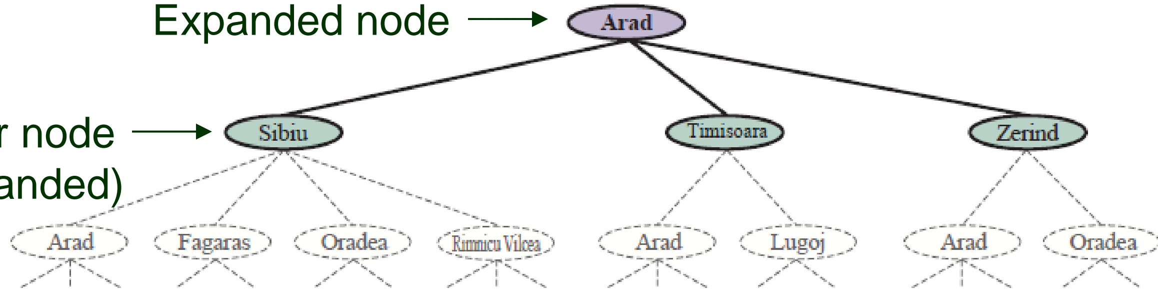




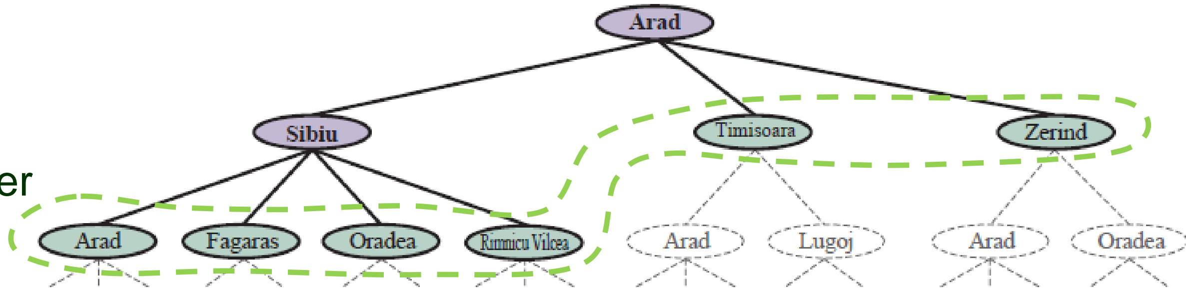
Expanded node

Frontier node (unexpanded)

Node to be generated next

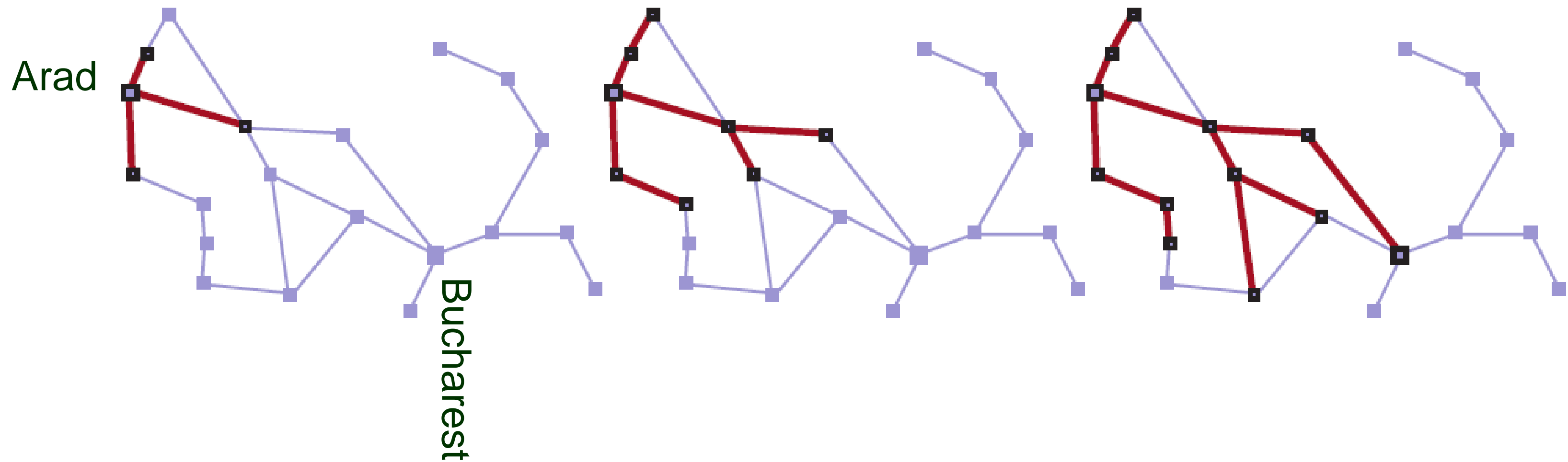


Frontier



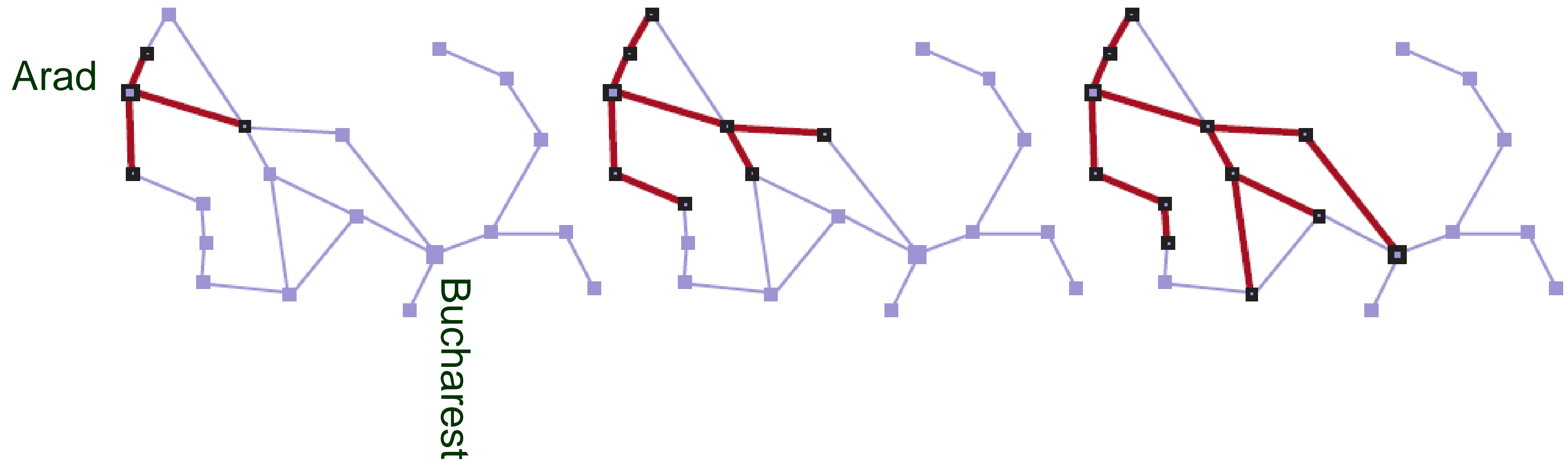
Generated Search Trees

Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.



Generated Search Trees

Superimpose a search tree over the state space graph and find a path from the initial state to the goal state.



Which node on the frontier to expand next?

Best-First Search

Choose a node n with minimum value of some *evaluation function* $f(n)$.

- Return it if its state is a goal state.
- Otherwise generate child nodes and add them to the frontier.

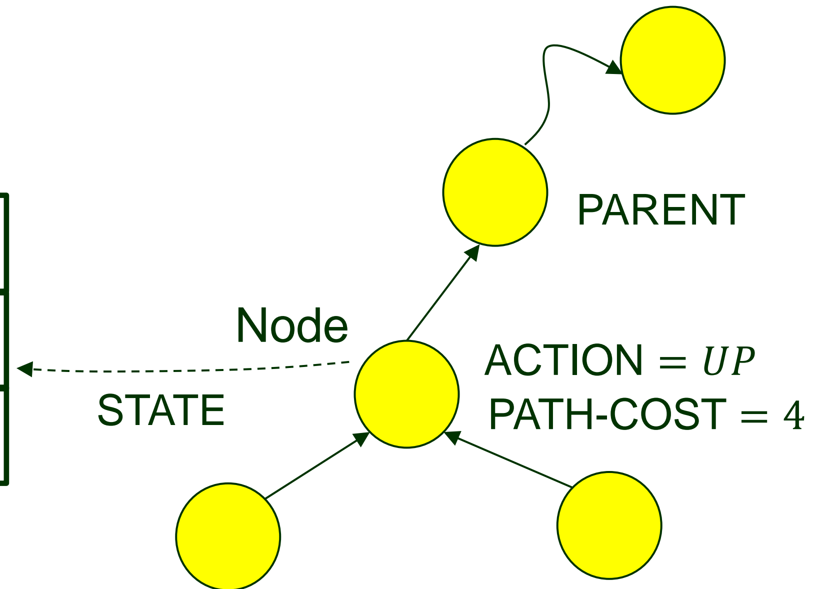
Best-First Search

Choose a node n with minimum value of some *evaluation function* $f(n)$.

- Return it if its state is a goal state.
- Otherwise generate child nodes and add them to the frontier.

Node structure:

1	2	3
6	7	4
8	5	



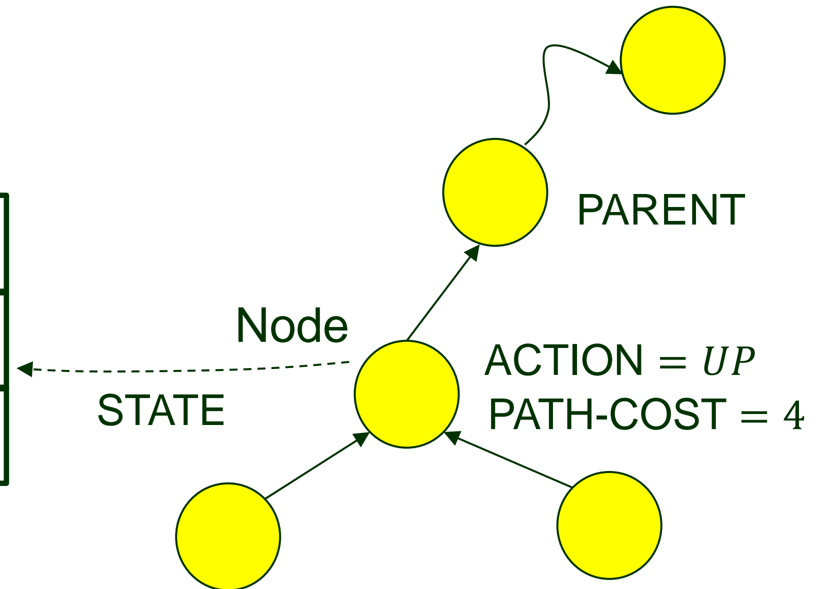
Best-First Search

Choose a node n with minimum value of some *evaluation function* $f(n)$.

- Return it if its state is a goal state.
- Otherwise generate child nodes and add them to the frontier.

Node structure:

1	2	3
6	7	4
8	5	



- *node*.STATE
- *node*.PARENT
- *node*.ACTION: action applied at the parent node to generate this node
- *node*.PATH-COST: total cost of the past from the initial state to this node.

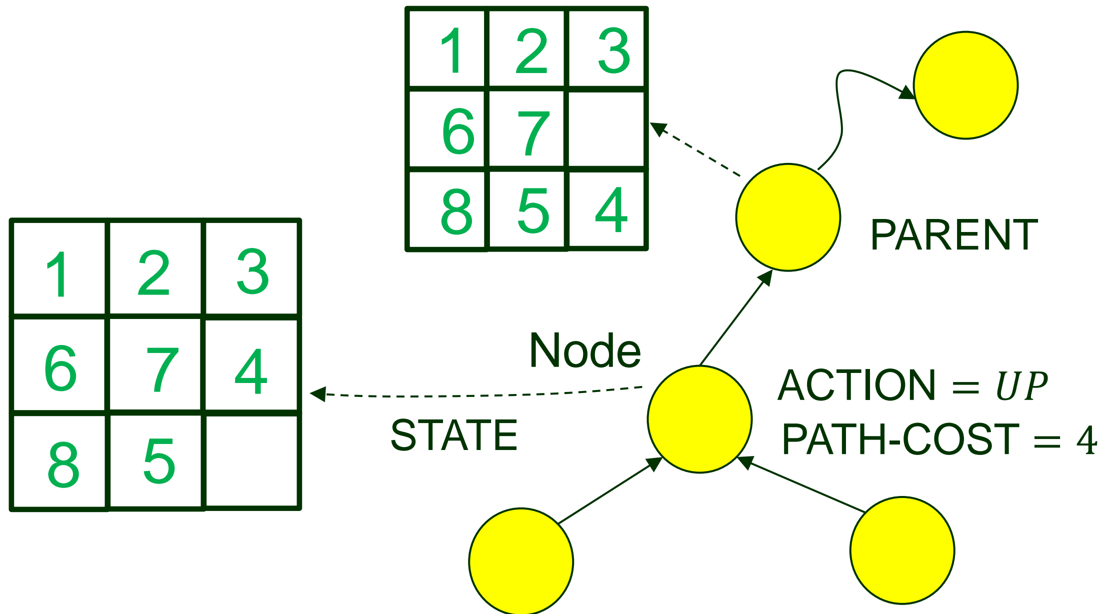
Best-First Search

Choose a node n with minimum value of some *evaluation function* $f(n)$.

- Return it if its state is a goal state.
- Otherwise generate child nodes and add them to the frontier.

Node structure:

- *node*.STATE
- *node*.PARENT



- *node*.ACTION: action applied at the parent node to generate this node
- *node*.PATH-COST: total cost of the past from the initial state to this node.

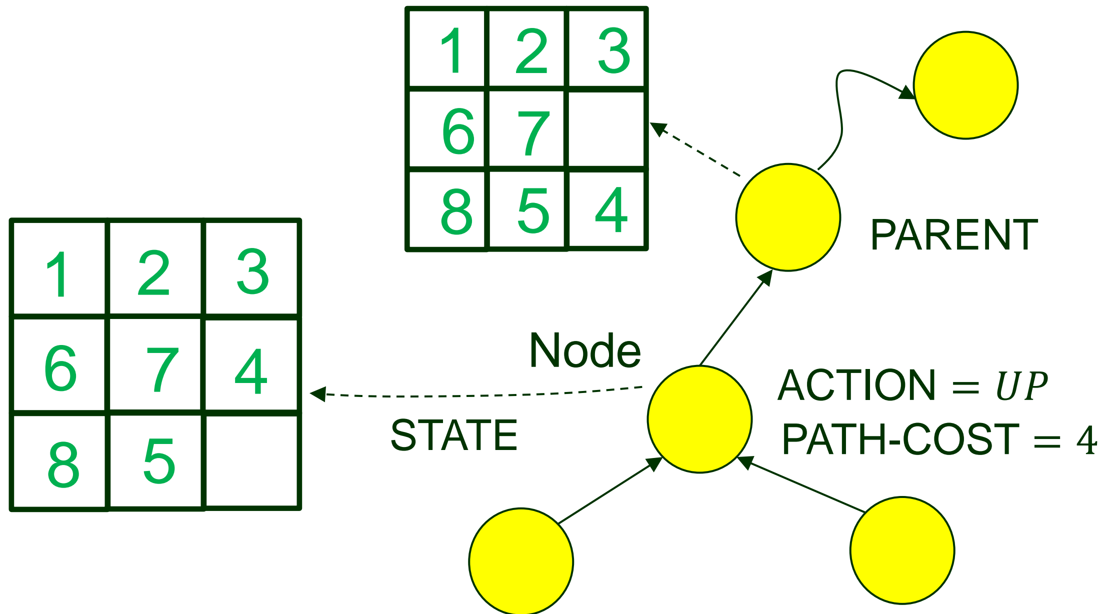
Best-First Search

Choose a node n with minimum value of some *evaluation function* $f(n)$.

- Return it if its state is a goal state.
- Otherwise generate child nodes and add them to the frontier.

Node structure:

- *node*.STATE
- *node*.PARENT

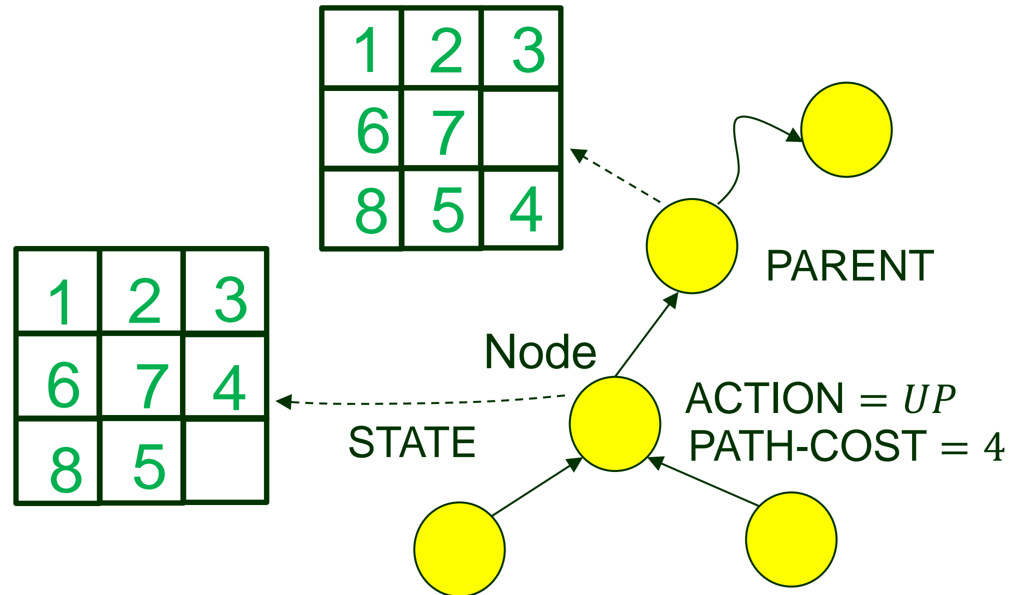


- *node*.ACTION: action applied at the parent node to generate this node
- *node*.PATH-COST: total cost of the past from the initial state to this node. $g(n)$

More on Data Structure

Frontier:

- IS-EMPTY(*frontier*)
- POP(*frontier*)
- TOP(*frontier*)
- ADD(*node*, *frontier*)



Three kinds of queues:

- A priority queue pops the node with the minimum cost.
- A FIFO queue pops the first added node (used in BFS).
- A LIFO queue pops the most recently added node (used in DFS).

Best-First Search Algorithm

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element
  reached ← a lookup table, with one entry with key problem.INITIAL and value node
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Best-First Search Algorithm

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element  // states on the frontier
  reached ← a lookup table, with one entry with key problem.INITIAL and value node  // states
  while not IS-EMPTY(frontier) do                                          // that have been reached
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

Best-First Search Algorithm

```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element  // states on the frontier
  reached ← a lookup table, with one entry with key problem.INITIAL and value node  // states
  while not IS-EMPTY(frontier) do                                     // that have been reached
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s′ ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s′)
    yield NODE(STATE=s′, PARENT=node, ACTION=action, PATH-COST=cost)
```

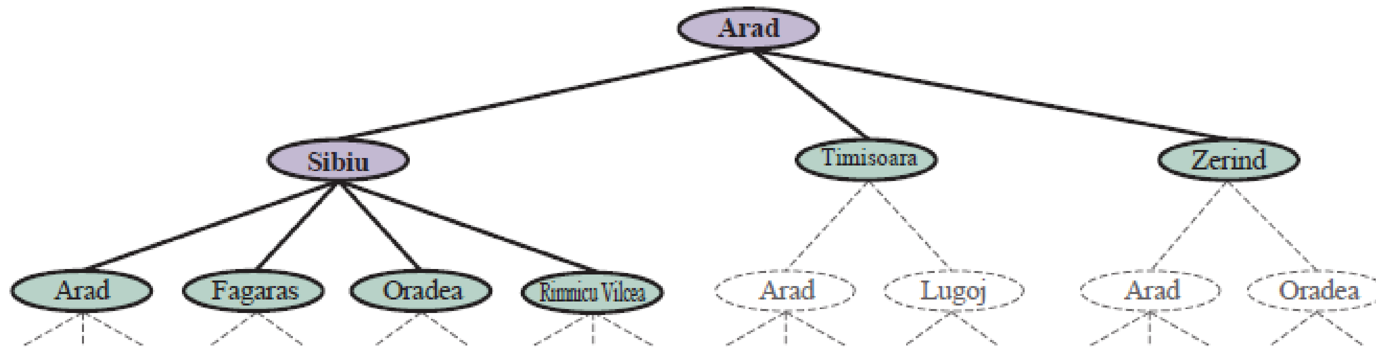
Can implement BFS and DFS.

Best-First Search Algorithm

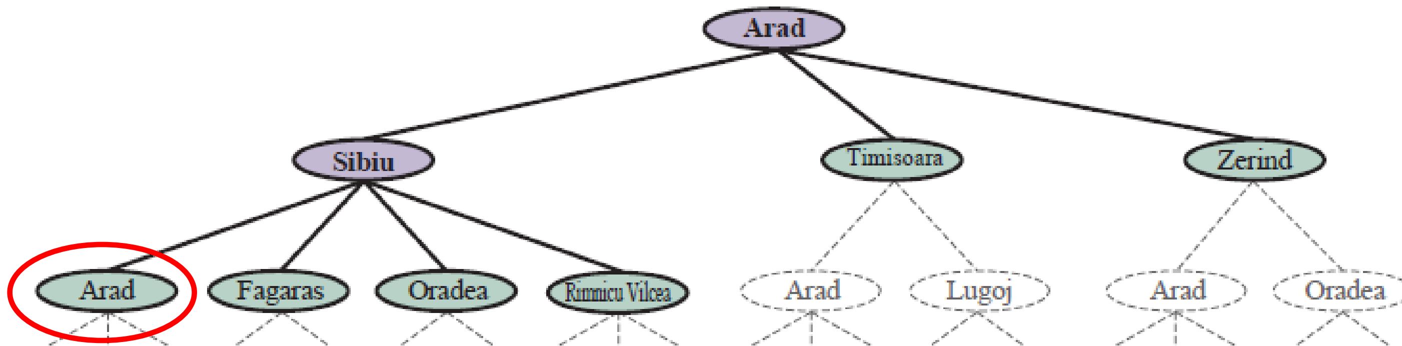
```
function BEST-FIRST-SEARCH(problem, f) returns a solution node or failure
  node ← NODE(STATE=problem.INITIAL)
  frontier ← a priority queue ordered by f, with node as an element  // states on the frontier
  reached ← a lookup table, with one entry with key problem.INITIAL and value node  // states
  while not IS-EMPTY(frontier) do                                       // that have been reached
    node ← POP(frontier)
    if problem.IS-GOAL(node.STATE) then return node
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if s is not in reached or child.PATH-COST < reached[s].PATH-COST then
        reached[s] ← child    // state s is reached at the node child.
        add child to frontier
  return failure

function EXPAND(problem, node) yields nodes
  s ← node.STATE
  for each action in problem.ACTIONS(s) do
    s' ← problem.RESULT(s, action)
    cost ← node.PATH-COST + problem.ACTION-COST(s, action, s')
    yield NODE(STATE=s', PARENT=node, ACTION=action, PATH-COST=cost)
```

Can implement BFS and DFS.

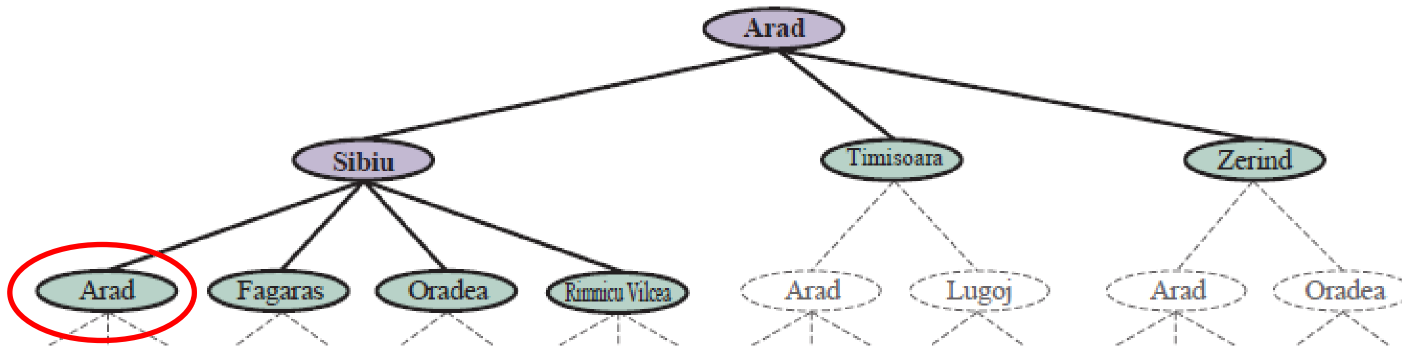
Repeated State

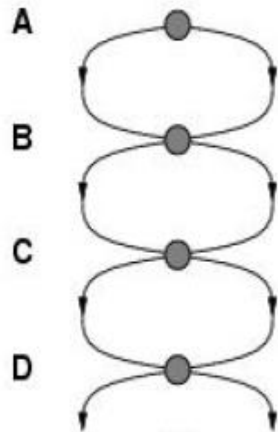


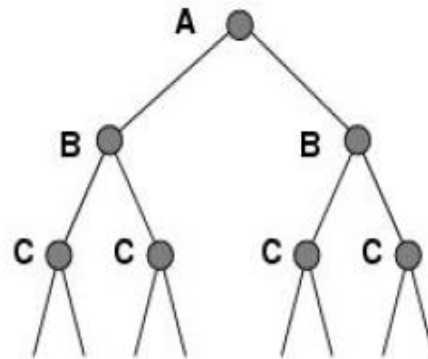
Repeated State



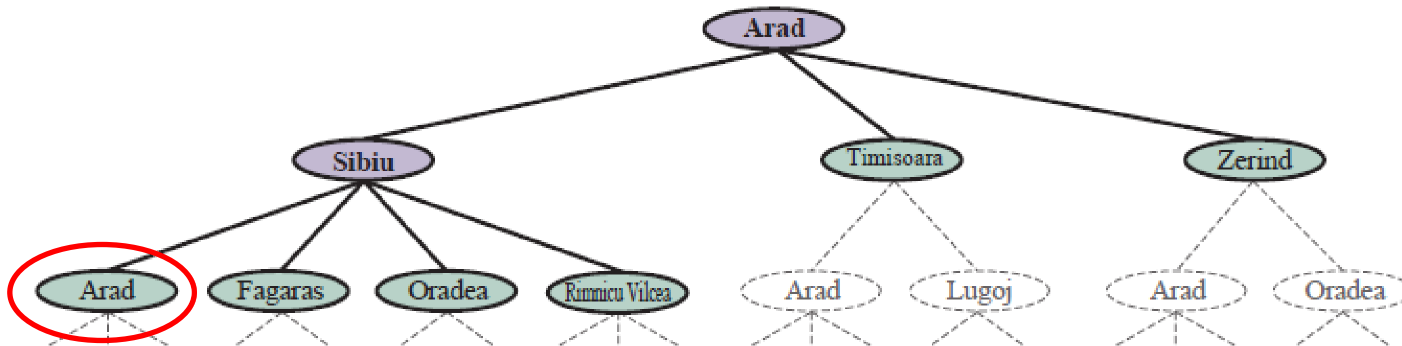
Repeated State

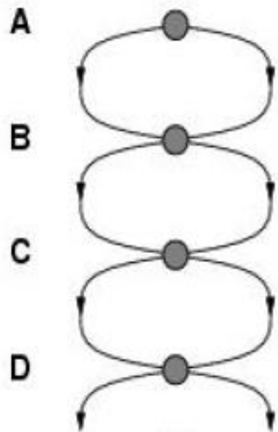


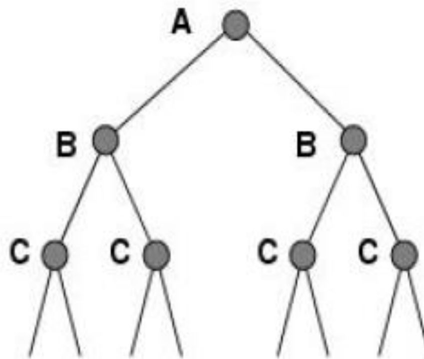




Repeated State

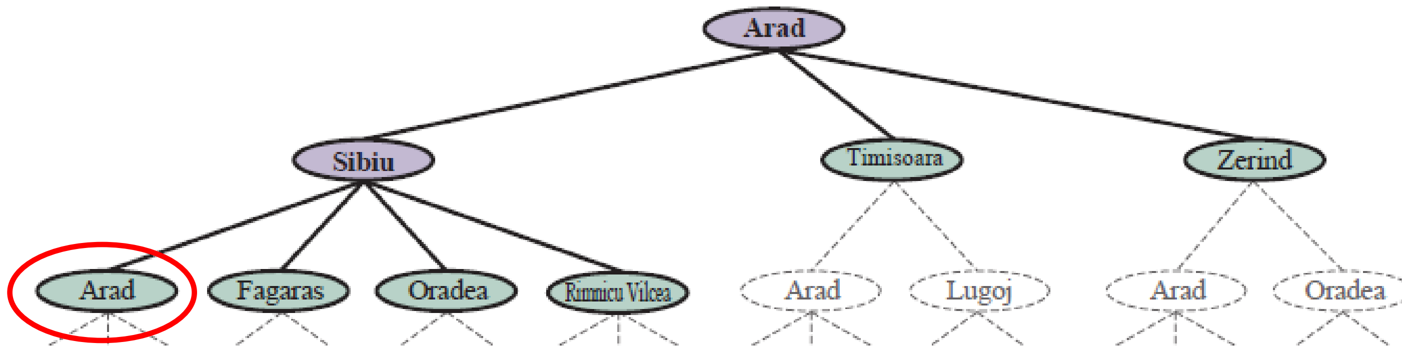


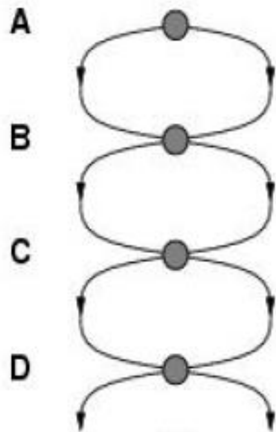


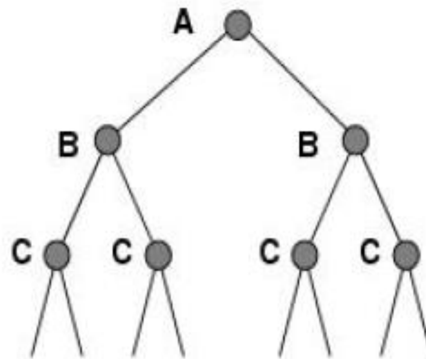


- Failure to detect repeated states can turn a solvable problem into an unsolvable one.

Repeated State







- Failure to detect repeated states can turn a solvable problem into an unsolvable one.
- Keep only the best path to each state.

Performance Measures

- Completeness: Is the algorithm guaranteed to find a solution whenever one exists?

  The state space may be infinite!

- Cost optimality: Does it find a solution with the lowest path cost of all solutions?

- Time complexity: Physical time or the number of states and actions.

- Space complexity: Memory needed for the search.

  $|V| + |E|$?

Depth and Branching Factor

- ♣ The measure in terms of $|V| + |E|$ is appropriate when the graph is *explicit*.

Depth and Branching Factor

- ♣ The measure in terms of $|V| + |E|$ is appropriate when the graph is *explicit*.

- But often *implicit* graph representation of a state space in AI:

Depth and Branching Factor

- ♣ The measure in terms of $|V| + |E|$ is appropriate when the graph is *explicit*.

- But often *implicit* graph representation of a state space in AI:
  - the initial state
  - actions
  - a transition model

Depth and Branching Factor

♣ The measure in terms of $|V| + |E|$ is appropriate when the graph is *explicit*.

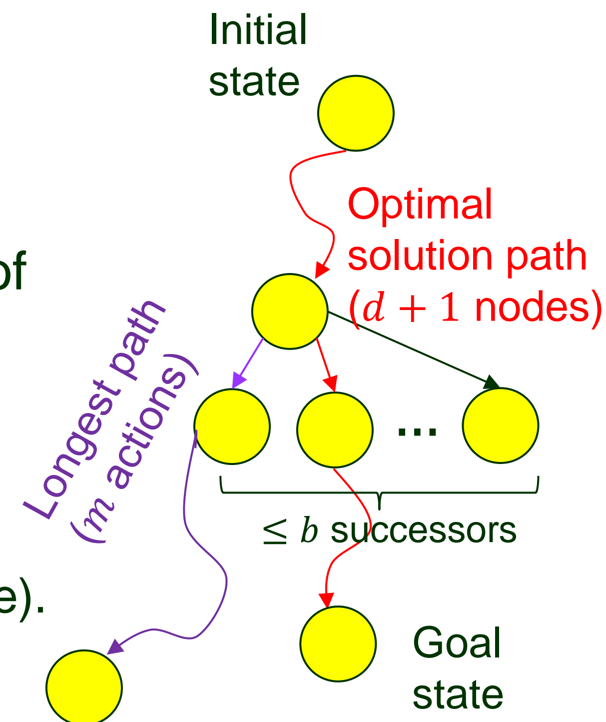
- But often *implicit* graph representation of a state space in AI:
  - the initial state
  - actions
  - a transition model

- Accordingly, complexity is measured in terms of

♦ d: *depth* (number of actions in the optimal solution)

♦ m: *maximum number of actions* in any path

♦ b: *branching factor* (number of successors of a node).



Infinite State Space

Knuth's conjecture: Any integer > 4 can be reached from 4 via a sequence of square root, floor, and factorial operations.

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

* Photo from https://amturing.acm.org/byyear.cfm.

Infinite State Space

Knuth's conjecture: Any integer > 4 can be reached from 4 via a sequence of square root, floor, and factorial operations.

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

Algorithm Repeatedly applies the "factorial operator".

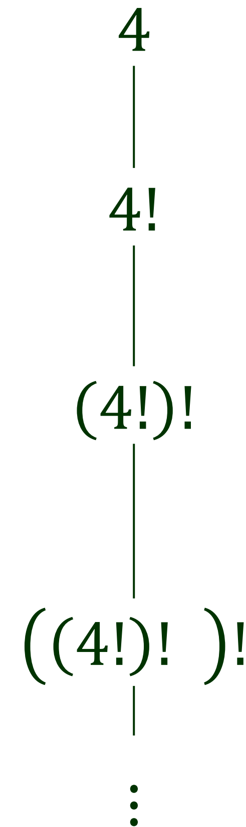
* Photo from https://amturing.acm.org/byyear.cfm.

Infinite State Space

Knuth's conjecture: Any integer > 4 can be reached from 4 via a sequence of square root, floor, and factorial operations.

$$\left\lfloor \sqrt{\sqrt{\sqrt{\sqrt{\sqrt{(4!)!}}}}} \right\rfloor = 5$$

Algorithm Repeatedly applies the "factorial operator".



* Photo from https://amturing.acm.org/byyear.cfm.

II. Breadth-First Search

Uninformed search: No clue about how close a state is to the goal.

II. Breadth-First Search

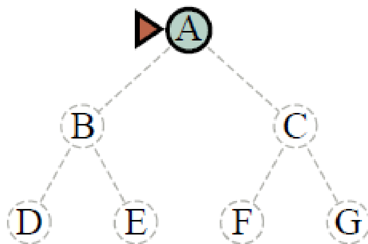
Uninformed search: No clue about how close a state is to the goal.

Expand the root first, then all its successors, next their successors, and so on.

II. Breadth-First Search

Uninformed search: No clue about how close a state is to the goal.

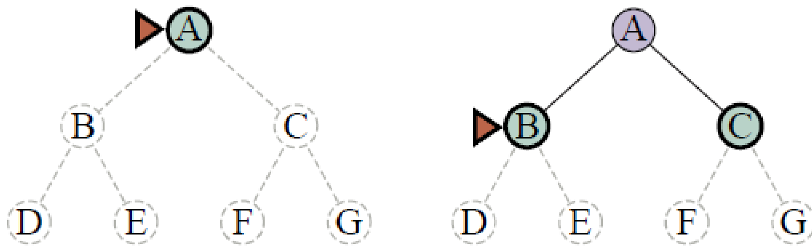
Expand the root first, then all its successors, next their successors, and so on.



II. Breadth-First Search

Uninformed search: No clue about how close a state is to the goal.

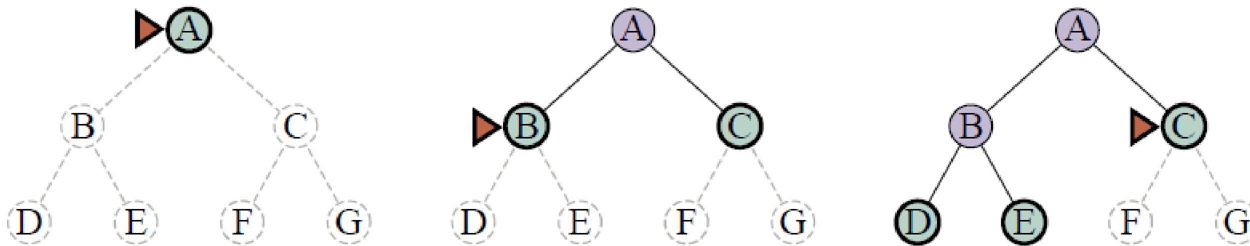
Expand the root first, then all its successors, next their successors, and so on.



II. Breadth-First Search

Uninformed search: No clue about how close a state is to the goal.

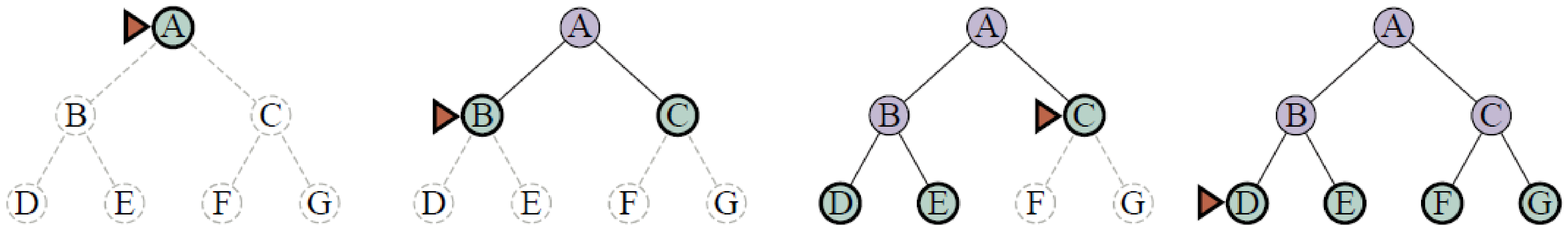
Expand the root first, then all its successors, next their successors, and so on.



II. Breadth-First Search

Uninformed search: No clue about how close a state is to the goal.

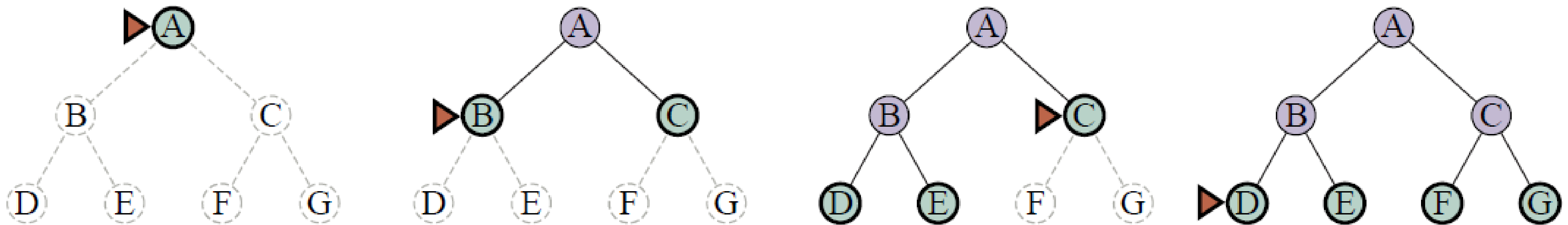
Expand the root first, then all its successors, next their successors, and so on.



II. Breadth-First Search

Uninformed search: No clue about how close a state is to the goal.

Expand the root first, then all its successors, next their successors, and so on.



- Systematic search.
- Complete even when the state space is infinite.
- Always finds a solution with a minimum number of actions.

BFS Algorithm

- Can call BEST-FIRST-SEARCH by letting the evaluation function $f(n)$ = node depth.

♠ Not efficient: Use a FIFO queue and adopt early goal test (when a node is generated).

```
function BREADTH-FIRST-SEARCH(problem) returns a solution node or failure
  node ← NODE(problem.INITIAL)
  if problem.IS-GOAL(node.STATE) then return node
  frontier ← a FIFO queue, with node as an element
  reached ← {problem.INITIAL}
  while not IS-EMPTY(frontier) do
    node ← POP(frontier)
    for each child in EXPAND(problem, node) do
      s ← child.STATE
      if problem.IS-GOAL(s) then return child
      if s is not in reached then
        add s to reached
        add child to frontier
  return failure
```

Time and Space Complexities

BFS on a uniform tree where every node has b successors.

branching factor

- b nodes at depth 1 generated by the root.
- Each node at depth 1 generates b nodes. $\Rightarrow b^2$ nodes at depth 2.
- And so on.

Time and Space Complexities

BFS on a uniform tree where every node has b successors.

branching factor

- b nodes at depth 1 generated by the root.
- Each node at depth 1 generates b nodes. $\Rightarrow$ b^2 nodes at depth 2.
- And so on.

Solution at depth d

Time and Space Complexities

BFS on a uniform tree where every node has b successors.

branching factor

- b nodes at depth 1 generated by the root.
- Each node at depth 1 generates b nodes. $\Rightarrow b^2$ nodes at depth 2.
- And so on.

Solution at depth $d \Longrightarrow$ #nodes $= 1 + b + \cdots + b^d$

Time and Space Complexities

BFS on a uniform tree where every node has b successors.

branching factor

- b nodes at depth 1 generated by the root.
- Each node at depth 1 generates b nodes. $\Rightarrow b^2$ nodes at depth 2.
- And so on.

Solution at depth d $\Longrightarrow$ #nodes $= 1 + b + \cdots + b^d$

$$= \frac{b^{d+1}-1}{b-1} \text{ (assuming } b > 1\text{)}$$

Time and Space Complexities

BFS on a uniform tree where every node has b successors.

branching factor

- b nodes at depth 1 generated by the root.
- Each node at depth 1 generates b nodes. $\Rightarrow b^2$ nodes at depth 2.
- And so on.

Solution at depth d $\Longrightarrow$ #nodes $= 1 + b + \cdots + b^d = O(b^d)$

$$= \frac{b^{d+1}-1}{b-1} \text{ (assuming } b > 1)$$

Time and Space Complexities

BFS on a uniform tree where every node has b successors.

branching factor

- b nodes at depth 1 generated by the root.
- Each node at depth 1 generates b nodes. $\Rightarrow$ b^2 nodes at depth 2.
- And so on.

Solution at depth d $\Longrightarrow$ #nodes $= 1 + b + \cdots + b^d = O(b^d)$

$$= \frac{b^{d+1}-1}{b-1} \text{ (assuming } b > 1\text{)}$$

Time & space complexities
(since every node remains in memory)

Time and Space Complexities

BFS on a uniform tree where every node has b successors.

↑ branching factor

- b nodes at depth 1 generated by the root.
- Each node at depth 1 generates b nodes. $\Rightarrow b^2$ nodes at depth 2.
- And so on.

Solution at depth d ⟹ #nodes $= 1 + b + \cdots + b^d = O(b^d)$

$$= \frac{b^{d+1}-1}{b-1} \text{ (assuming } b > 1\text{)}$$

Time & space complexities
(since every node remains in memory)

♣ Only small search problems are solvable due to exponential time complexity.

Time and Space Complexities

BFS on a uniform tree where every node has b successors.

branching factor

- b nodes at depth 1 generated by the root.
- Each node at depth 1 generates b nodes. $\Rightarrow b^2$ nodes at depth 2.
- And so on.

Solution at depth d ⟹ #nodes $= 1 + b + \cdots + b^d = O(b^d)$

$$= \frac{b^{d+1}-1}{b-1} \text{ (assuming } b > 1\text{)}$$

Time & space complexities
(since every node remains in memory)

♣ Only small search problems are solvable due to exponential time complexity.

♣ Memory is a bigger issue than time.

Time and Memory Requirements for BFS

Tree search: $O(b^d)$

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 KB
4	11,110	11 milliseconds	10.6 MB
6	10^6	1.1 seconds	1 GB
8	10^8	2 minutes	103 GB
10	10^{10}	3 hours	10 TB
12	10^{12}	13 days	1 PB
14	10^{14}	3.5 years	99 PB
16	10^{16}	350 years	10 EB

$b = 10$

Time and Memory Requirements for BFS

Tree search: $O(b^d)$

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 KB
4	11,110	11 milliseconds	10.6 MB
6	10^6	1.1 seconds	1 GB
8	10^8	2 minutes	103 GB
10	10^{10}	3 hours	10 TB
12	10^{12}	13 days	1 PB
14	10^{14}	3.5 years	99 PB
16	10^{16}	350 years	10 EB

$$b = 10$$

Graph search is preferred since its time and space are proportional to the size of the state space (often less than $O(b^d)$).

Uniform-Cost Search (Dijkstra's Algorithm)

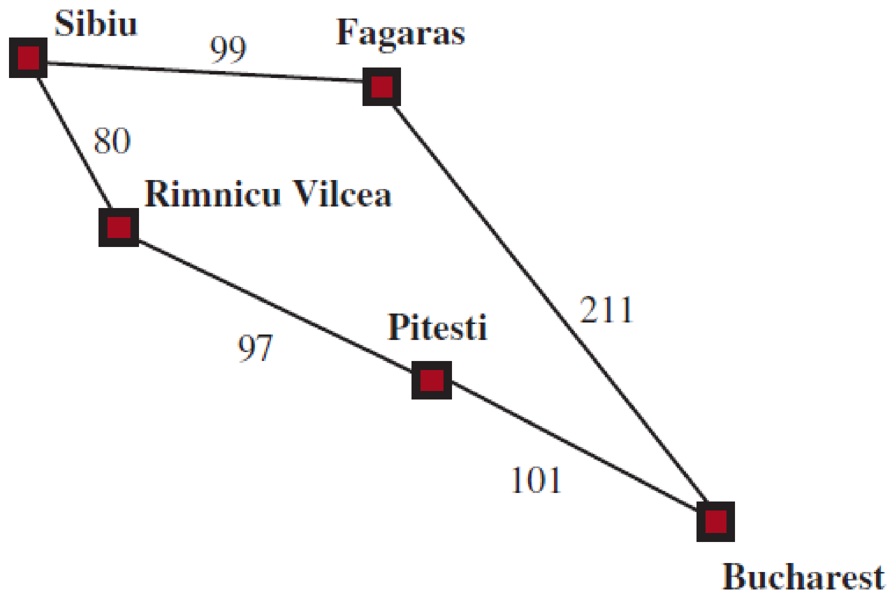
function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

- Spreads out in waves of uniform path-cost.

Uniform-Cost Search (Dijkstra's Algorithm)

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

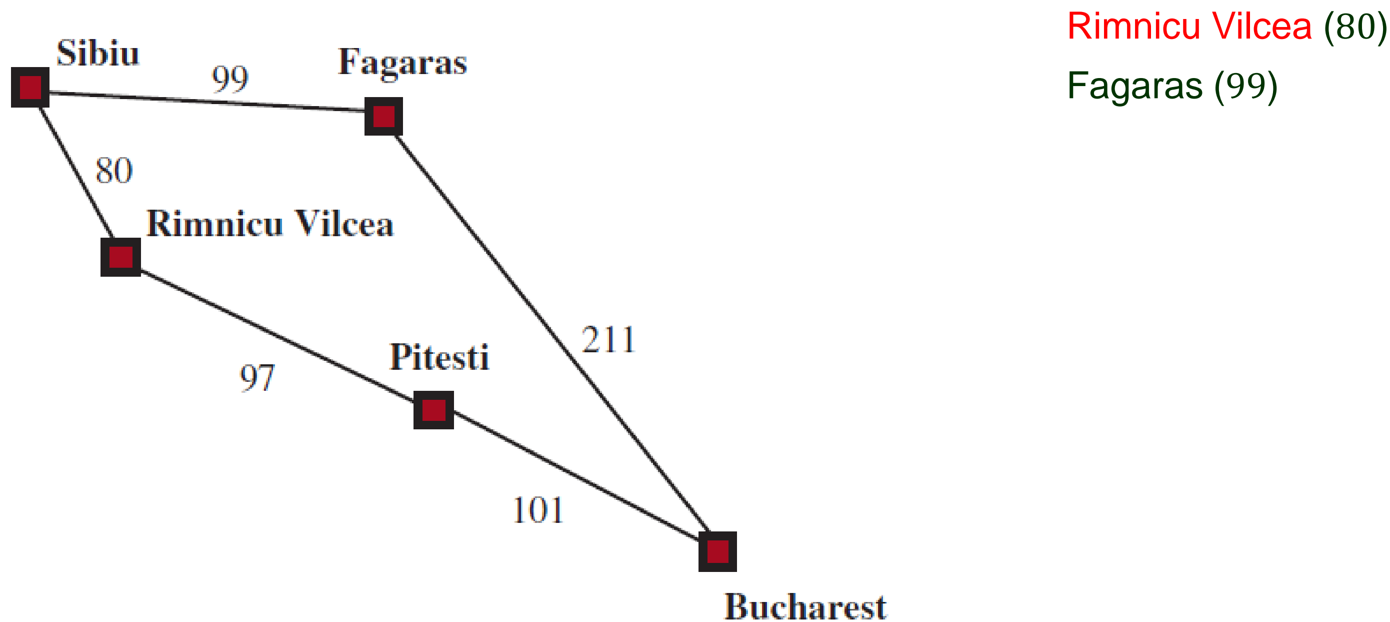
- Spreads out in waves of uniform path-cost.



Uniform-Cost Search (Dijkstra's Algorithm)

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

- Spreads out in waves of uniform path-cost.

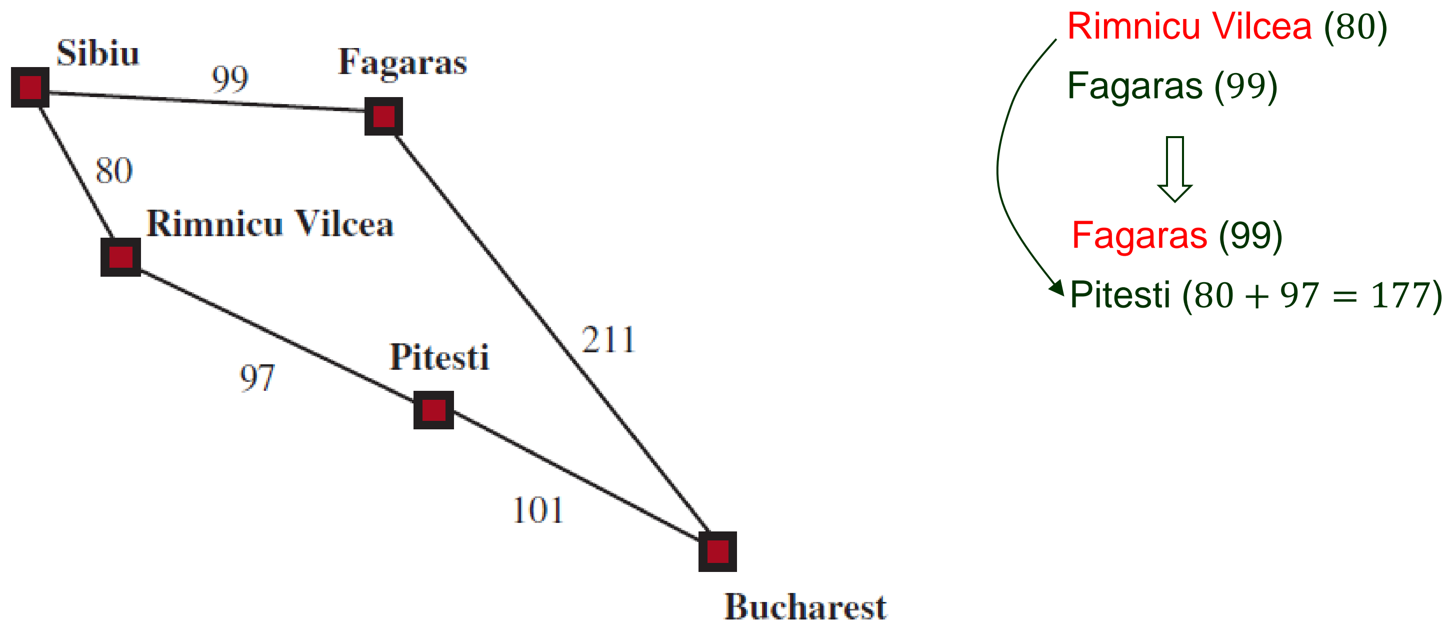


Rimnicu Vilcea (80)
Fagaras (99)

Uniform-Cost Search (Dijkstra's Algorithm)

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

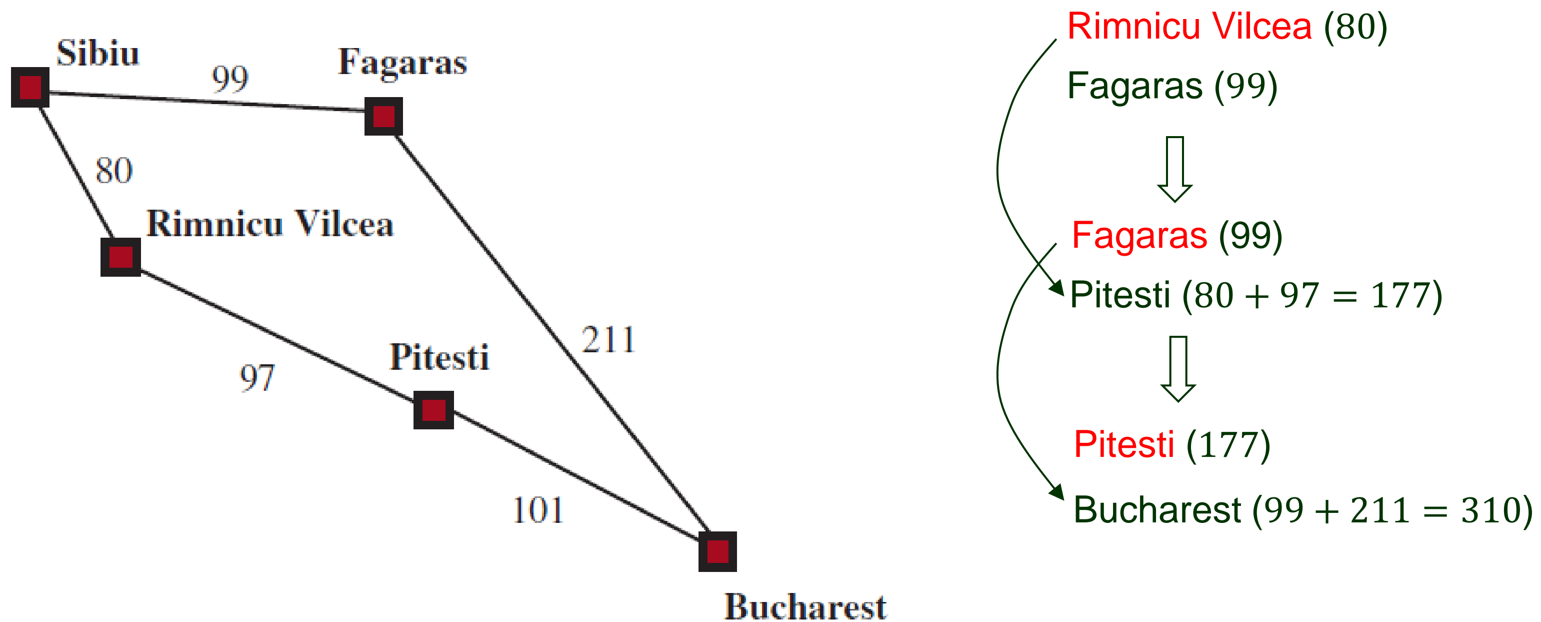
- Spreads out in waves of uniform path-cost.



Uniform-Cost Search (Dijkstra's Algorithm)

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

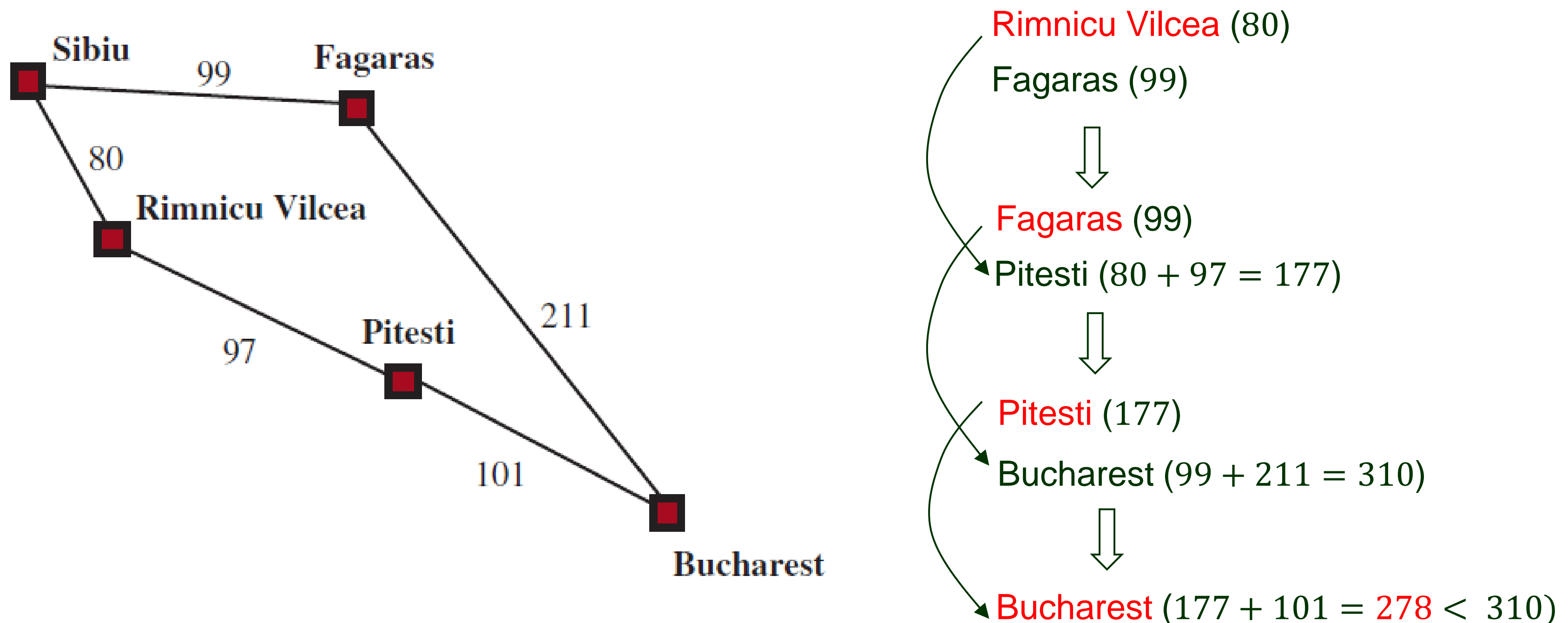
- Spreads out in waves of uniform path-cost.



Uniform-Cost Search (Dijkstra's Algorithm)

function UNIFORM-COST-SEARCH(*problem*) **returns** a solution node, or *failure*
  return BEST-FIRST-SEARCH(*problem*, PATH-COST)

- Spreads out in waves of uniform path-cost.



Uniform-Cost Search: Completeness and Complexity

- Completeness: systematic exploration of all paths – no chance of being trapped in one.

Uniform-Cost Search: Completeness and Complexity

- Completeness: systematic exploration of all paths – no chance of being trapped in one.

- Optimality: following from that of Dijkstra’s algorithm.

Uniform-Cost Search: Completeness and Complexity

- Completeness: systematic exploration of all paths – no chance of being trapped in one.

- Optimality: following from that of Dijkstra's algorithm.

- Complexity: guided by costs not depth

$$O\left(b^{1+\lfloor C^*/\epsilon \rfloor}\right)$$

C^*: cost of the optimal solution

ϵ: lower bound on the costs of all actions

Uniform-Cost Search: Completeness and Complexity

- Completeness: systematic exploration of all paths – no chance of being trapped in one.

- Optimality: following from that of Dijkstra's algorithm.

- Complexity: guided by costs not depth

$$O(b^{1+\lfloor C^*/\epsilon \rfloor})$$

C^*: cost of the optimal solution

ϵ: lower bound on the costs of all actions

$\gg$ b^d possible (e.g., exploring futile paths of small steps first)

Uniform-Cost Search: Completeness and Complexity

- Completeness: systematic exploration of all paths – no chance of being trapped in one.

- Optimality: following from that of Dijkstra's algorithm.

- Complexity: guided by costs not depth

$$O(b^{1+\lfloor C^*/\epsilon \rfloor})$$

C^*: cost of the optimal solution

ϵ: lower bound on the costs of all actions

$\gg\ b^d$ possible (e.g., exploring futile paths of small steps first)

$=\ b^{d+1}$ if all actions have the same cost.

III. Depth-First Search

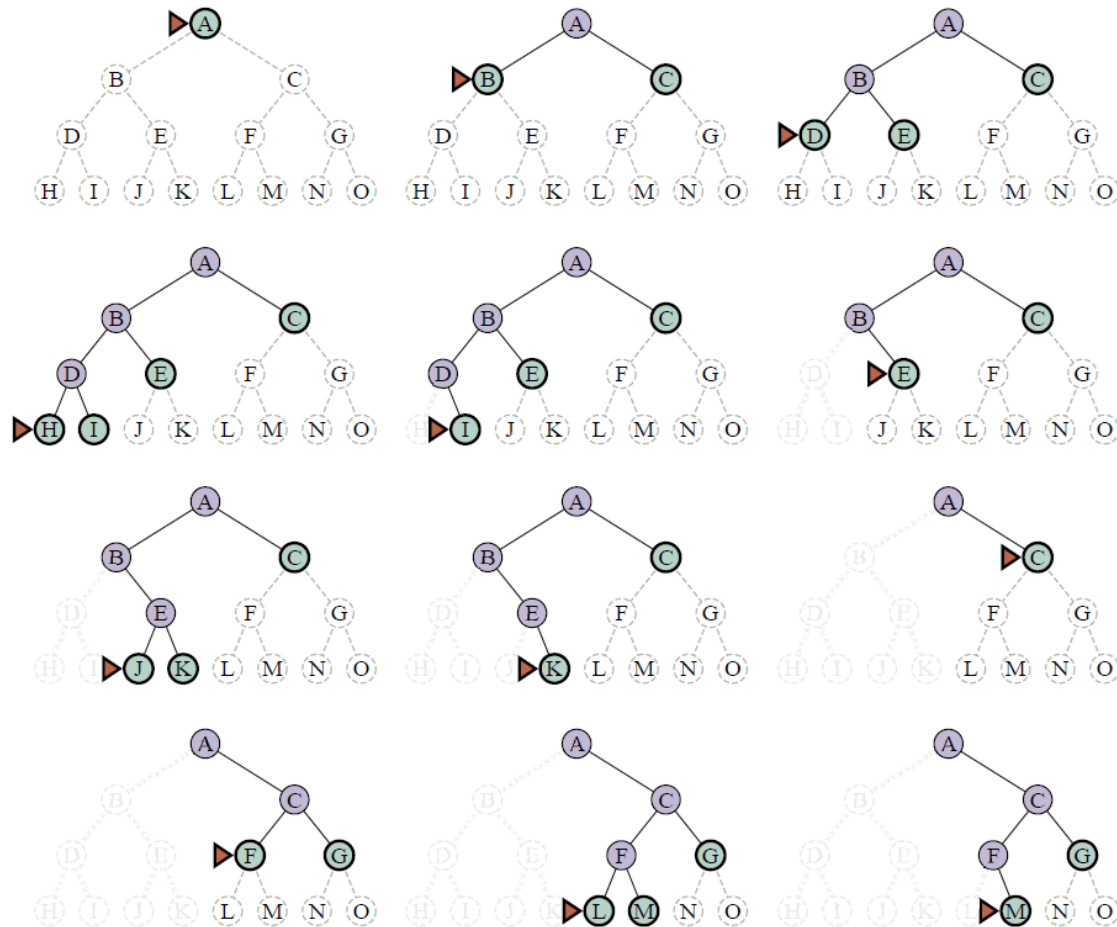
Expand the *deepest* node in the frontier first.

- Implementable as a call to BEST-FIRST-SEARCH by setting $f(n)$ to the negative depth.

III. Depth-First Search

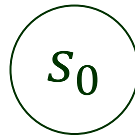
Expand the *deepest* node in the frontier first.

- Implementable as a call to BEST-FIRST-SEARCH by setting $f(n)$ to the negative depth.



Non-Optimality of DFS

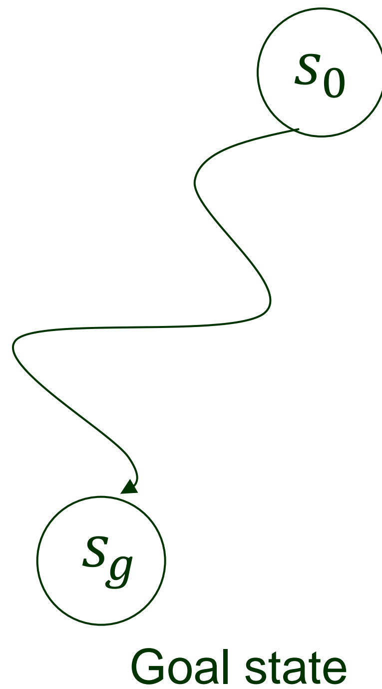
Returns the first solution it finds, even if it is not the cheapest.



Goal state

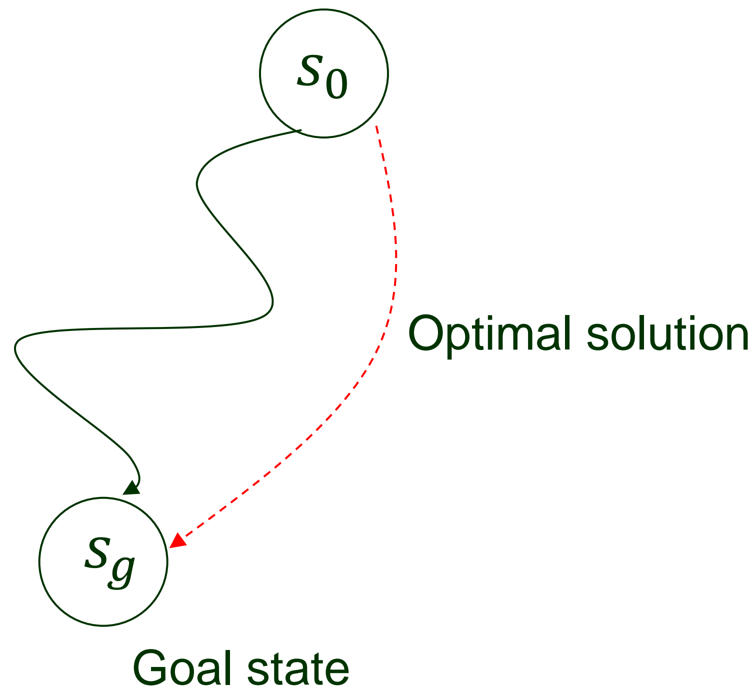
Non-Optimality of DFS

Returns the first solution it finds, even if it is not the cheapest.



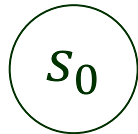
Non-Optimality of DFS

Returns the first solution it finds, even if it is not the cheapest.



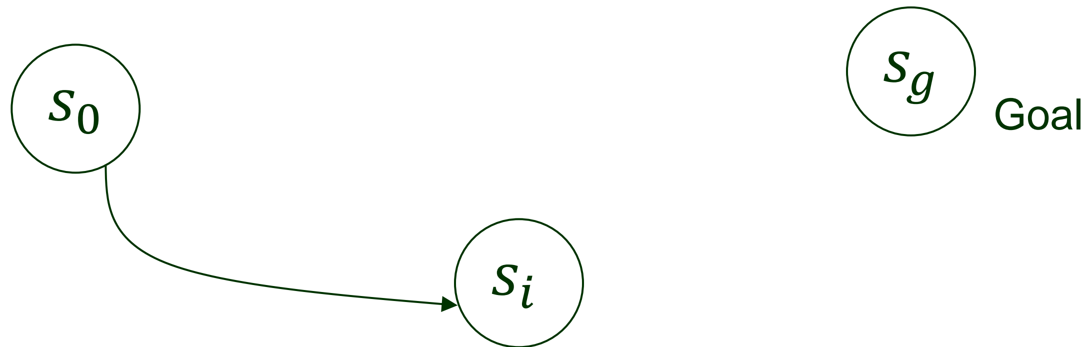
Inefficiency of DFS

- May expand the same state many times via different paths, and even systematically the entire space.



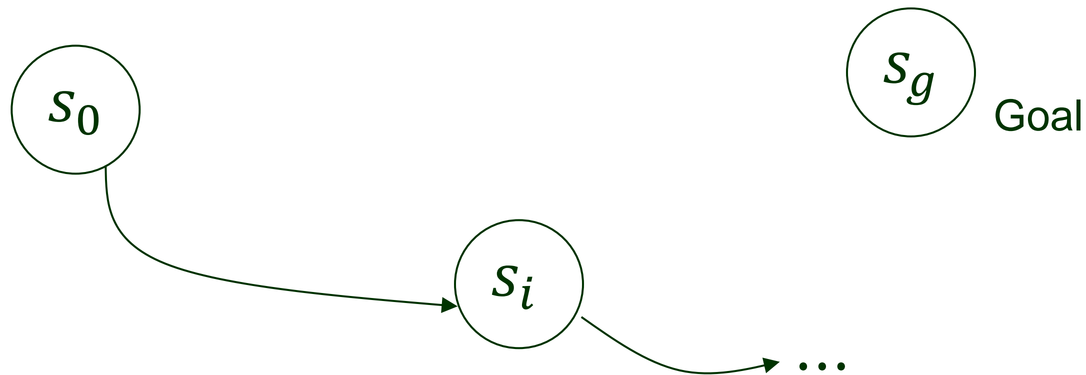
Inefficiency of DFS

- May expand the same state many times via different paths, and even systematically the entire space.



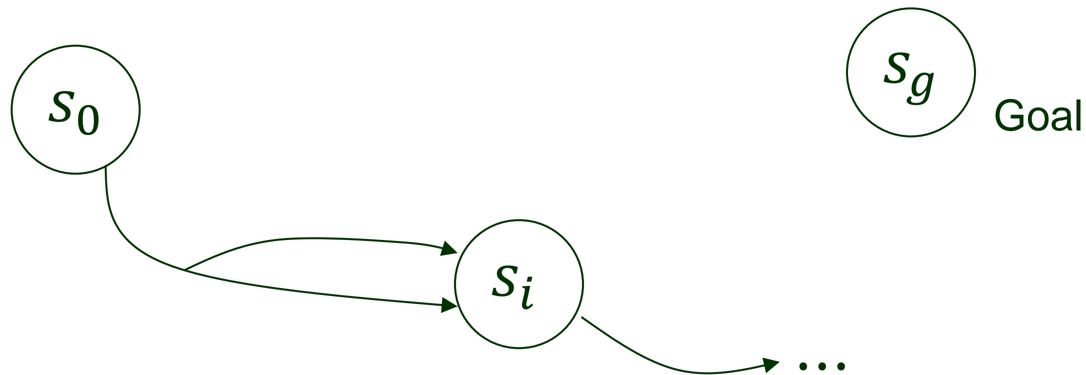
Inefficiency of DFS

- May expand the same state many times via different paths, and even systematically the entire space.



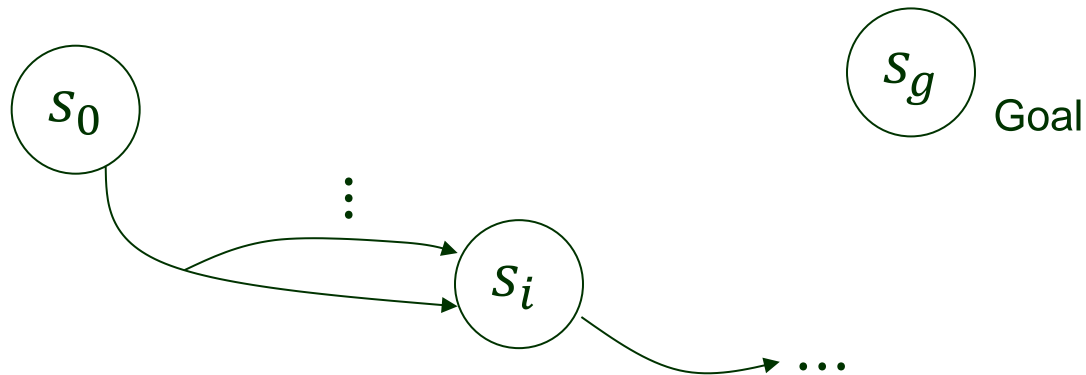
Inefficiency of DFS

- May expand the same state many times via different paths, and even systematically the entire space.



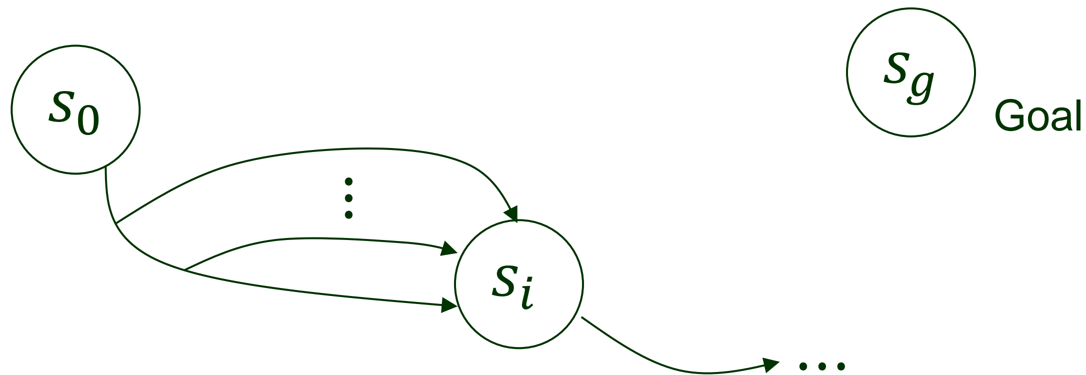
Inefficiency of DFS

- May expand the same state many times via different paths, and even systematically the entire space.



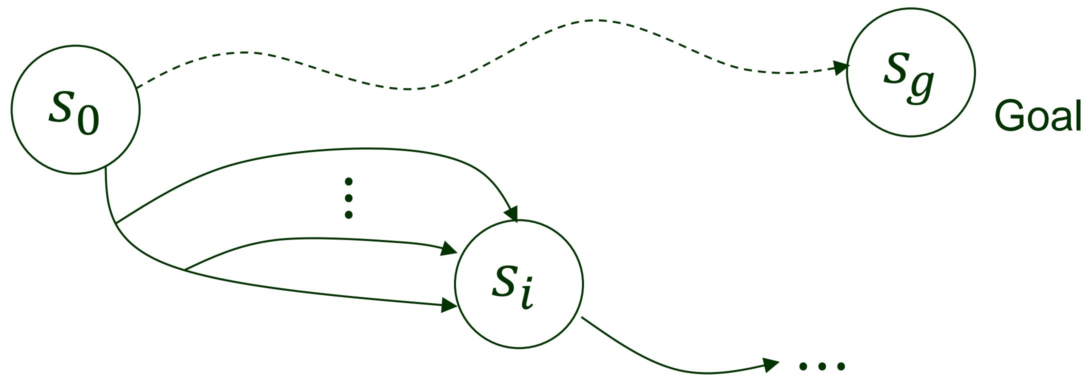
Inefficiency of DFS

- May expand the same state many times via different paths, and even systematically the entire space.



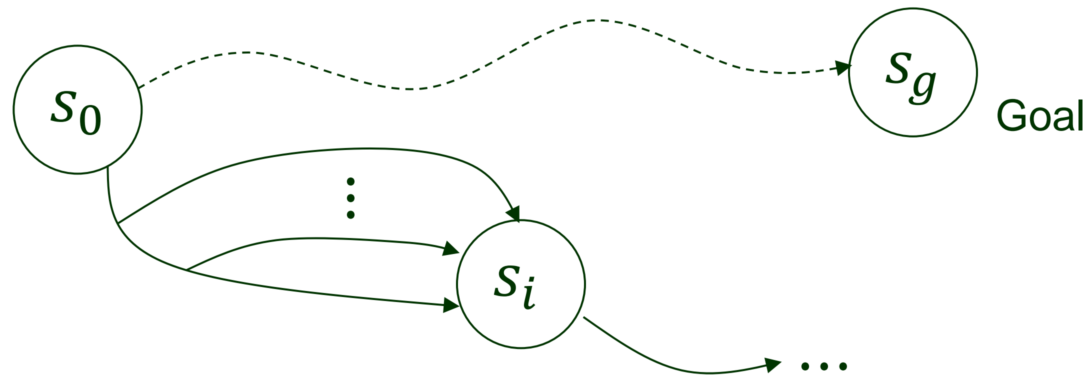
Inefficiency of DFS

- May expand the same state many times via different paths, and even systematically the entire space.



Inefficiency of DFS

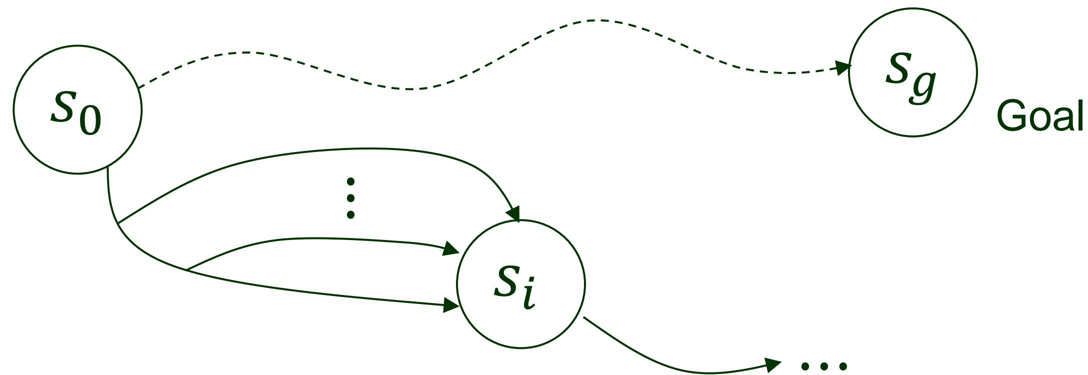
♠ May expand the same state many times via different paths, and even systematically the entire space.



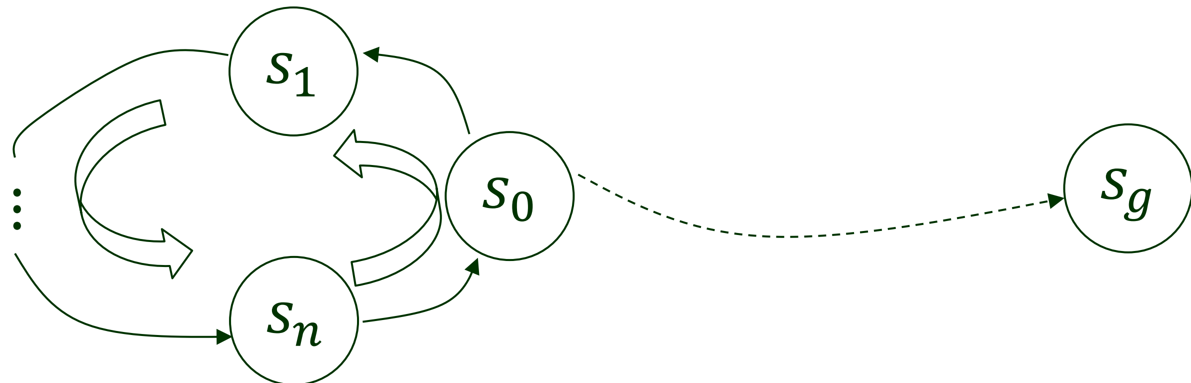
♠ May get stuck in an infinite loop in a cyclic state space.

Inefficiency of DFS

♠ May expand the same state many times via different paths, and even systematically the entire space.

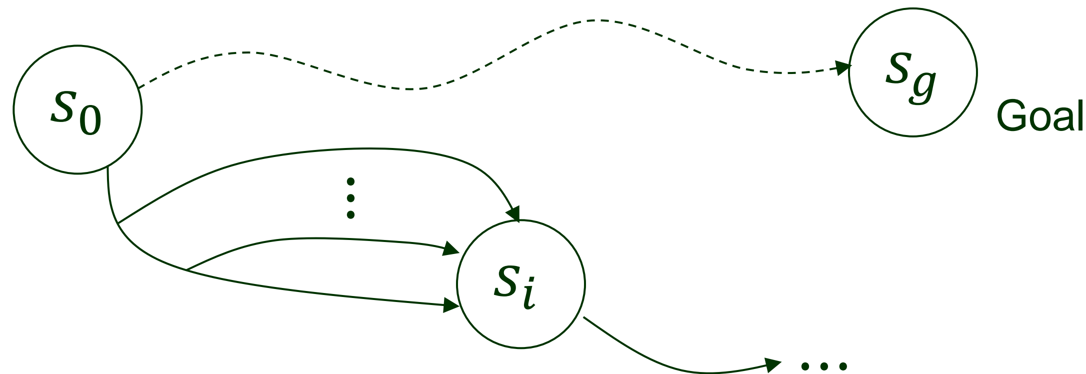


♠ May get stuck in an infinite loop in a cyclic state space.

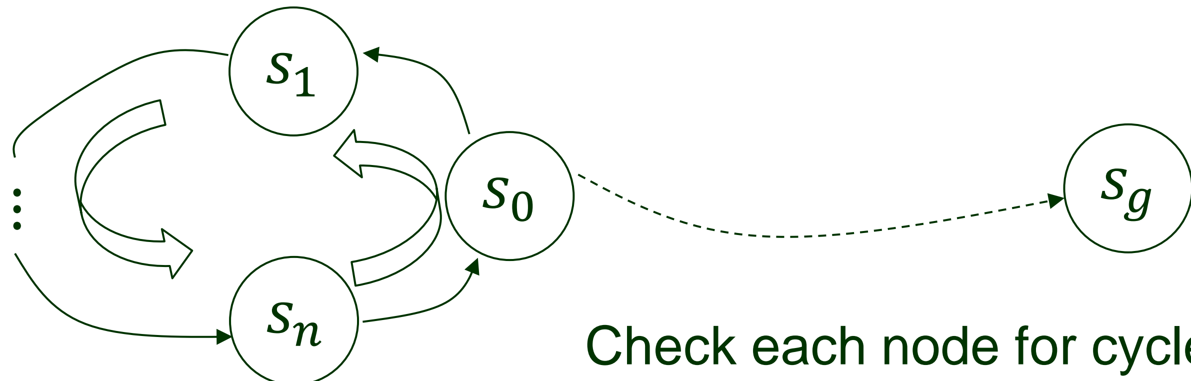


Inefficiency of DFS

- ♠ May expand the same state many times via different paths, and even systematically the entire space.

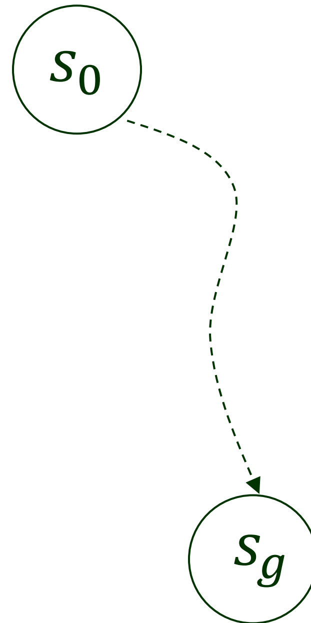


- ♠ May get stuck in an infinite loop in a cyclic state space.



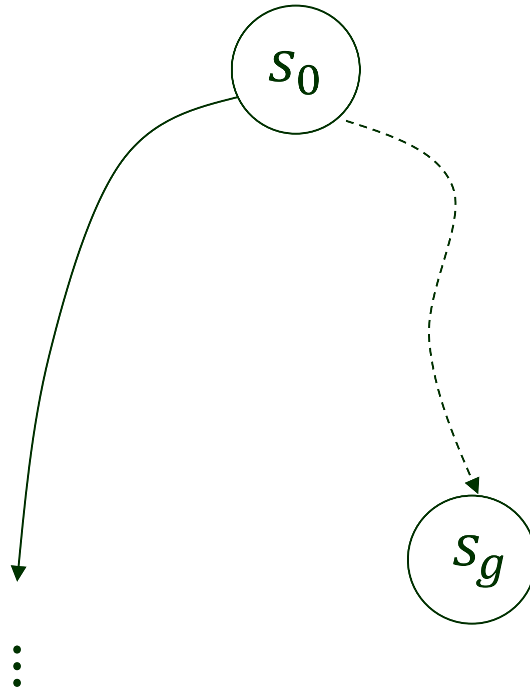
Incompleteness of DFS

Can go down an infinite path forever.



Incompleteness of DFS

Can go down an infinite path forever.

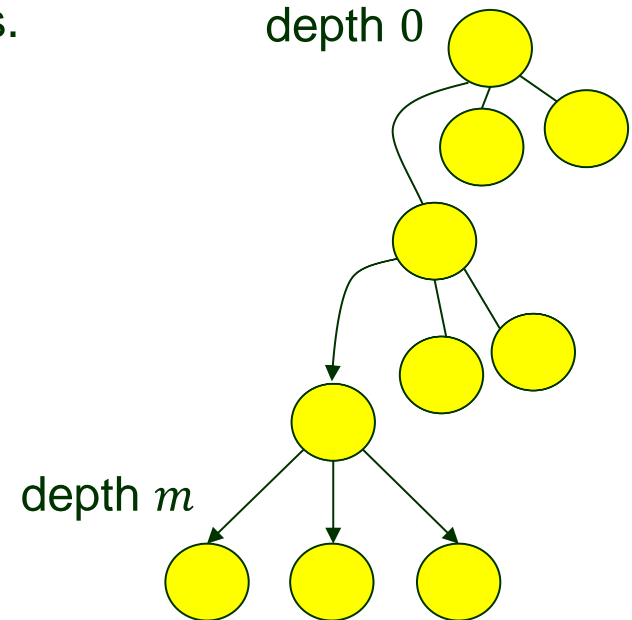


Why Consider DFS?

- Small memory for problems admitting tree-like search.

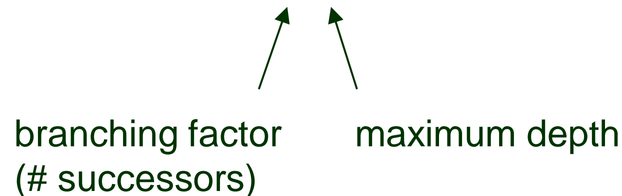
  No need for a table of reached nodes.

  Small frontier.



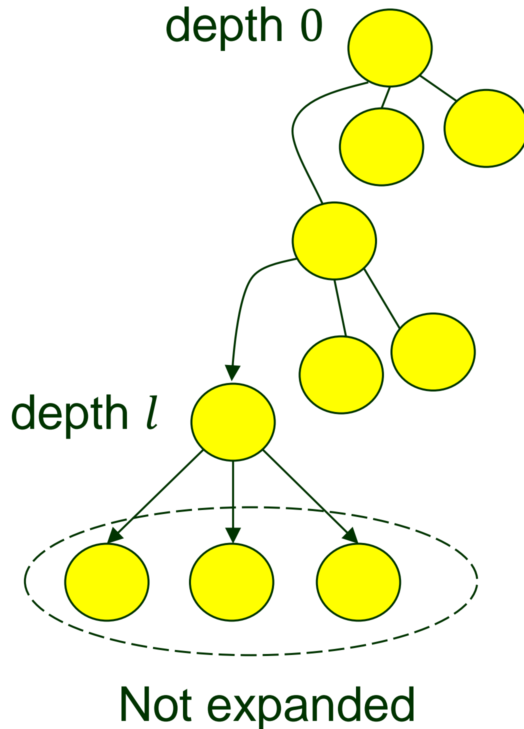
- Search time O(#nodes).

- Memory consumption $O(bm)$.



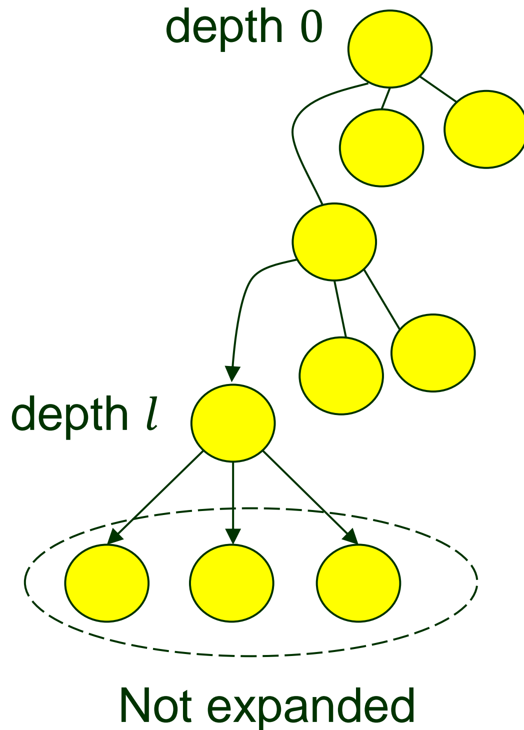
- Workhorse of constraint satisfaction, logic programming, etc.

Depth-Limited Seach



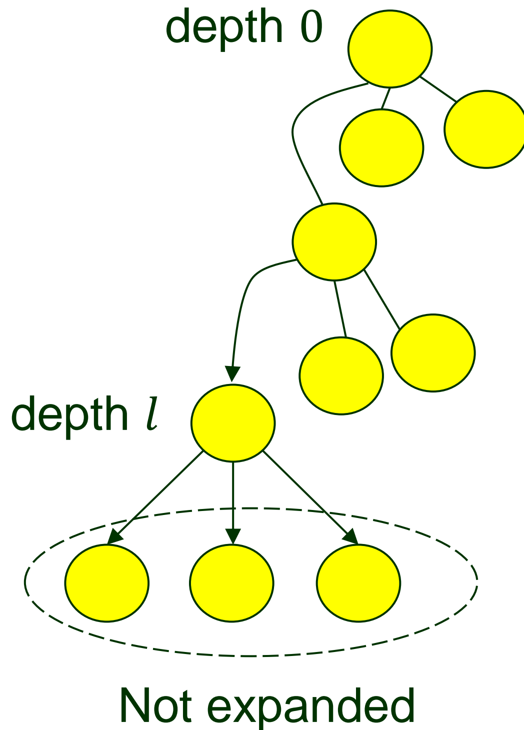
- To avoid exploring an infinite path, add a *depth limit* l.

Depth-Limited Seach



- To avoid exploring an infinite path, add a *depth limit* l.
- All nodes at depth l are considered dead ends even if they have successors.

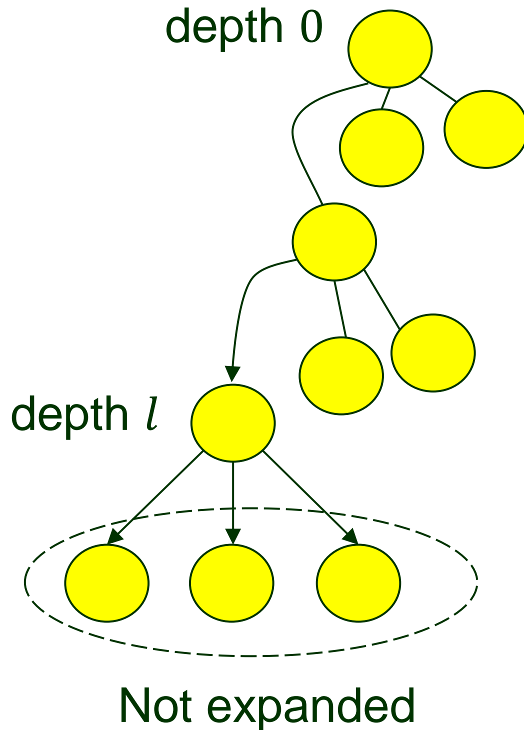
Depth-Limited Seach



- To avoid exploring an infinite path, add a *depth limit* l.
- All nodes at depth l are considered dead ends even if they have successors.

Time: $O(b^l)$

Depth-Limited Seach

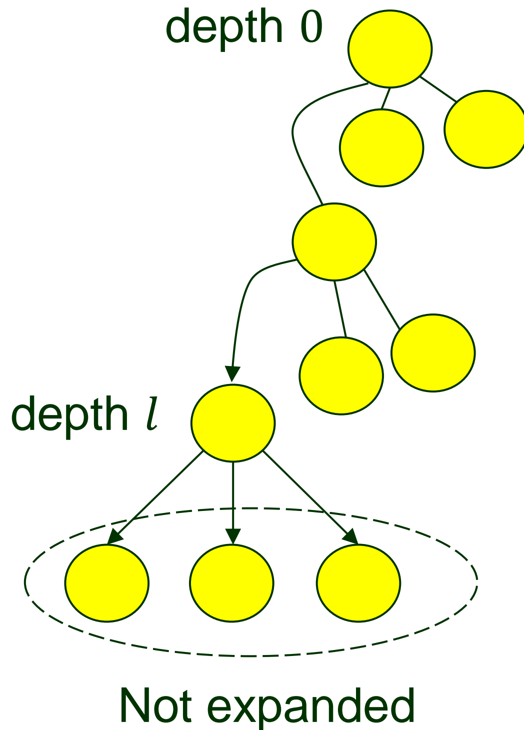


- To avoid exploring an infinite path, add a *depth limit* l.
- All nodes at depth l are considered dead ends even if they have successors.

Time: $O(b^l)$

Memory: $O(bl)$

Depth-Limited Seach



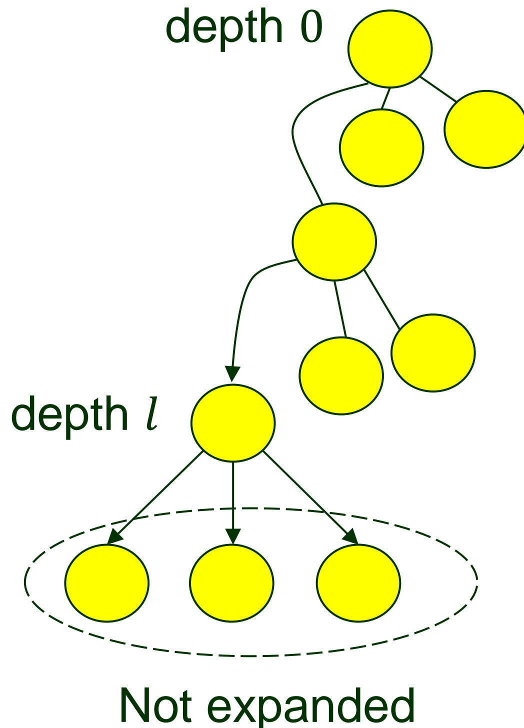
- To avoid exploring an infinite path, add a *depth limit* l.
- All nodes at depth l are considered dead ends even if they have successors.

Time: $O(b^l)$

Memory: $O(bl)$

♣ Performance sensitive to the choice for l.

Depth-Limited Seach



- To avoid exploring an infinite path, add a *depth limit* l.
- All nodes at depth l are considered dead ends even if they have successors.

Time: $O(b^l)$

Memory: $O(bl)$

♣ Performance sensitive to the choice for l.

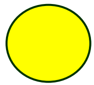
What strategy to address this?

IV. Iterative Deepening Search

Pick a good value for l by trying all values: 0, 1, 2, and so on.

IV. Iterative Deepening Search

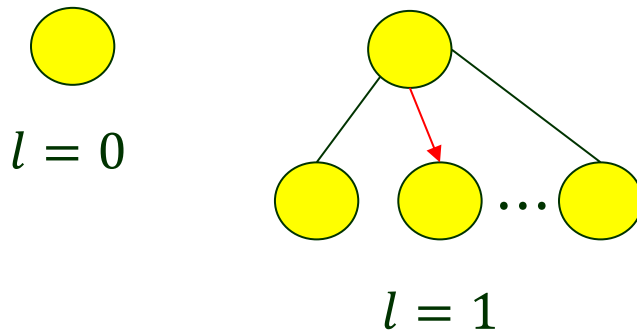
Pick a good value for l by trying all values: 0, 1, 2, and so on.



$l = 0$

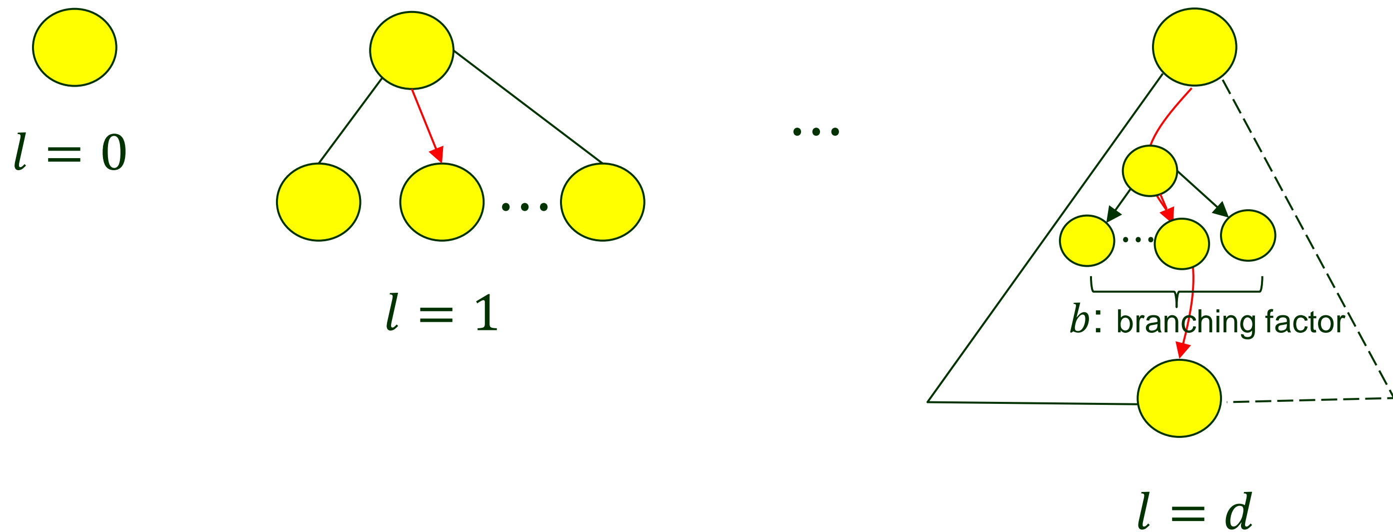
IV. Iterative Deepening Search

Pick a good value for l by trying all values: 0, 1, 2, and so on.



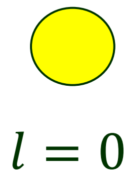
IV. Iterative Deepening Search

Pick a good value for l by trying all values: 0, 1, 2, and so on.

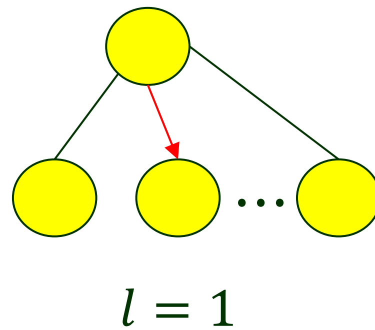


IV. Iterative Deepening Search

Pick a good value for l by trying all values: 0, 1, 2, and so on.

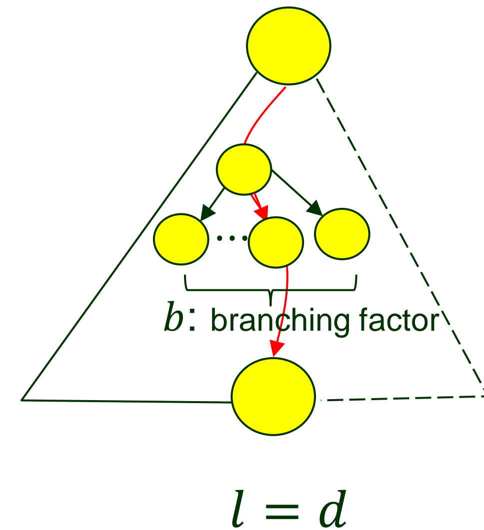


$l = 0$



$l = 1$

…

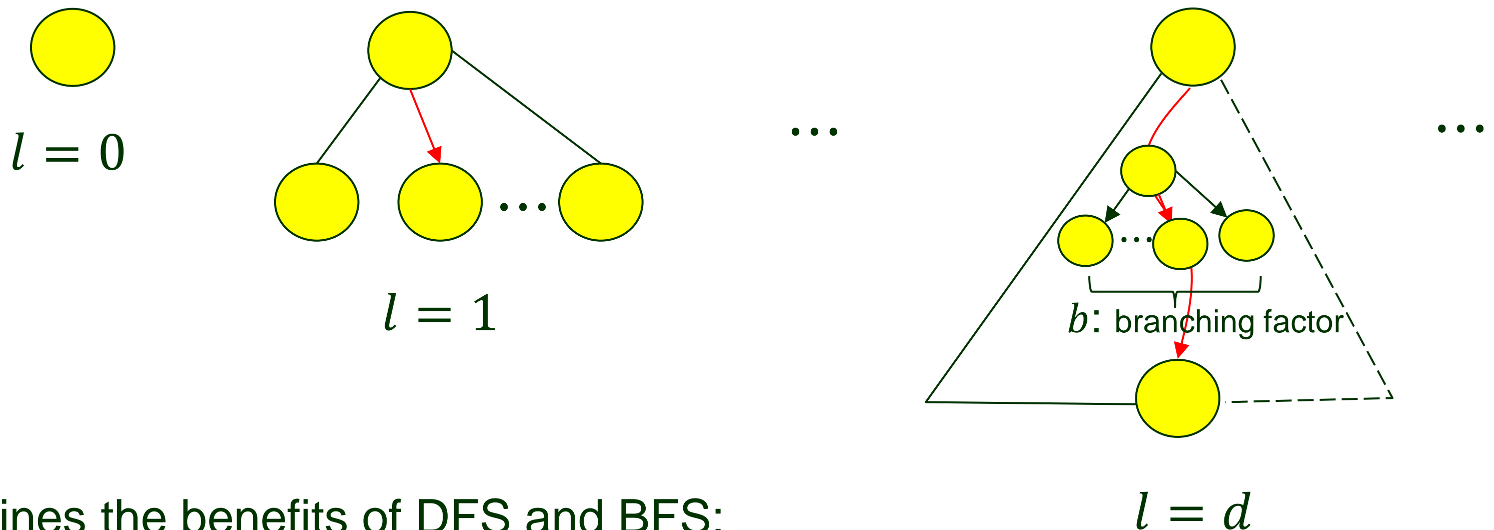


$l = d$

…

IV. Iterative Deepening Search

Pick a good value for l by trying all values: 0, 1, 2, and so on.

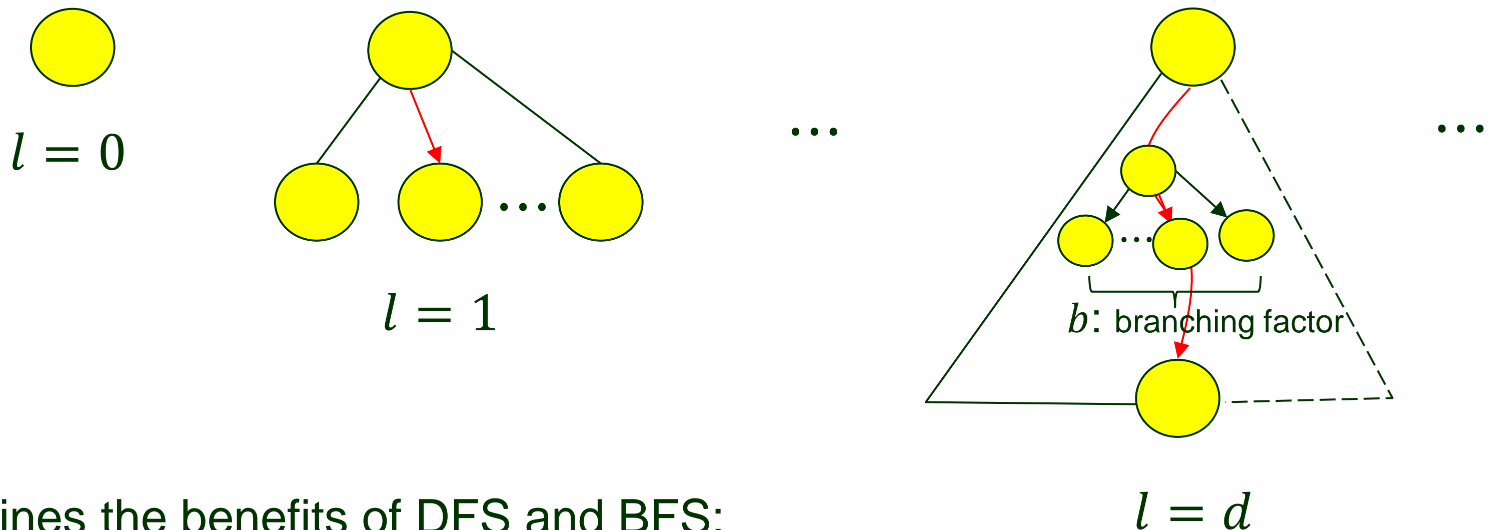


Combines the benefits of DFS and BFS:

- Completeness and optimality of BFS.
- Small memory requirement of DFS (only the current search path is stored).

IV. Iterative Deepening Search

Pick a good value for l by trying all values: 0, 1, 2, and so on.



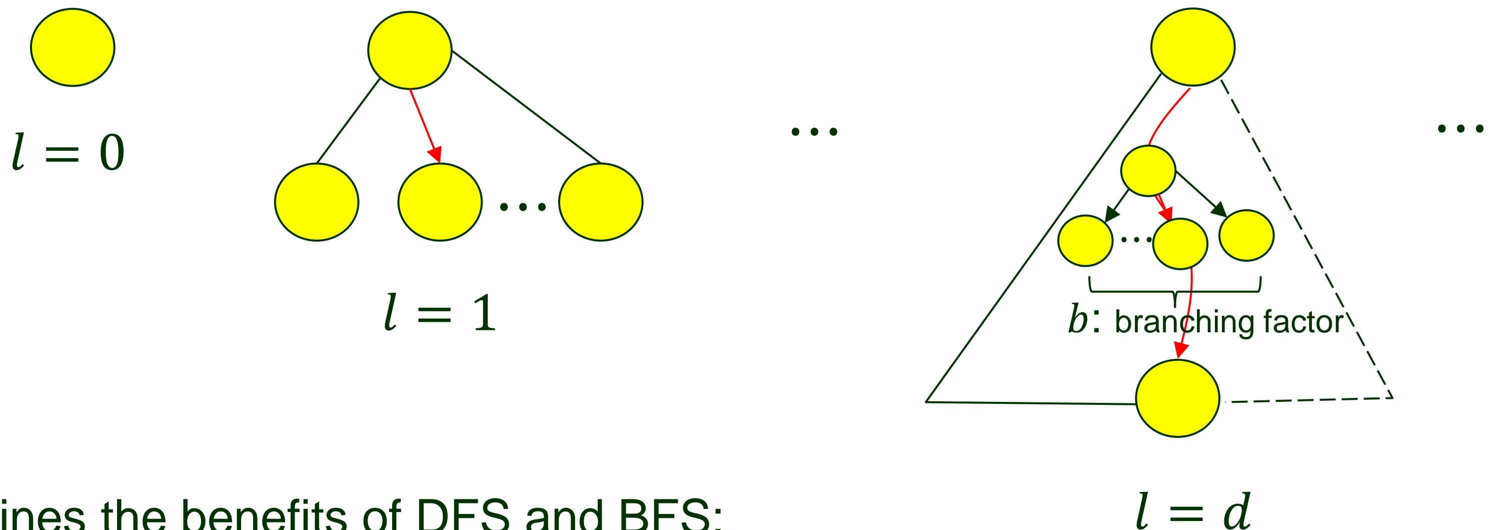
Combines the benefits of DFS and BFS:

- Completeness and optimality of BFS.
- Small memory requirement of DFS (only the current search path is stored).

Why not store all the nodes at depth d before moving on to depth $d + 1$?

IV. Iterative Deepening Search

Pick a good value for l by trying all values: 0, 1, 2, and so on.



Combines the benefits of DFS and BFS:

- Completeness and optimality of BFS.
- Small memory requirement of DFS (only the current search path is stored).

Why not store all the nodes at depth d before moving on to depth $d + 1$?

$O(b^d)$ memory!

IDS Pseudocode

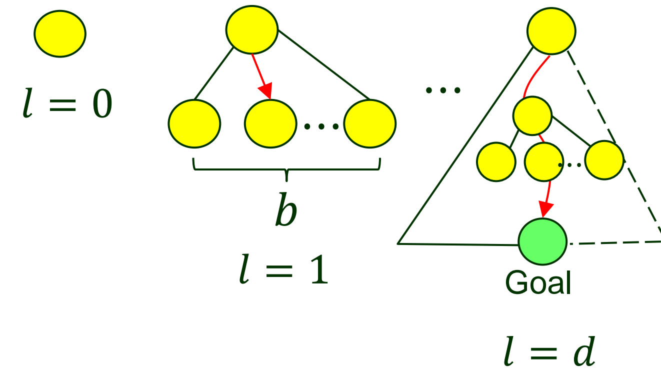
function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution node or *failure*
  for *depth* = 0 **to** ∞ **do**
    result ← DEPTH-LIMITED-SEARCH(*problem*, *depth*)
    if *result* ≠ *cutoff* **then return** *result*

DEPTH-LIMITED-SEARCH returns

- a goal state if it's found
- *failure* if it has exhausted all nodes and is certain no solution exists at any nodes.
- *cutoff* if no solution has been found for the current search limit l (but there might be a solution at some depth $> l$.

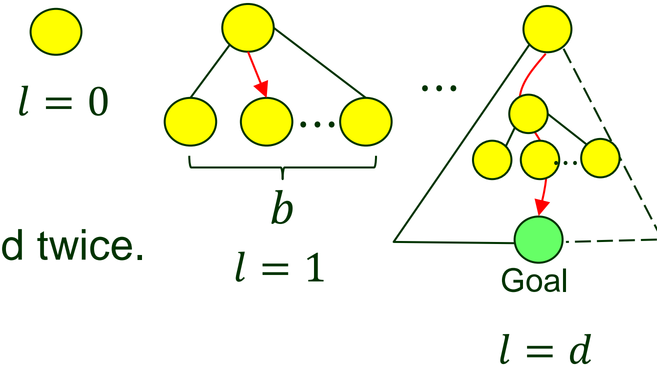
Time and Memory of IDS

Case 1 Goal state is found at depth *d*.



Time and Memory of IDS

Case 1 Goal state is found at depth *d*.



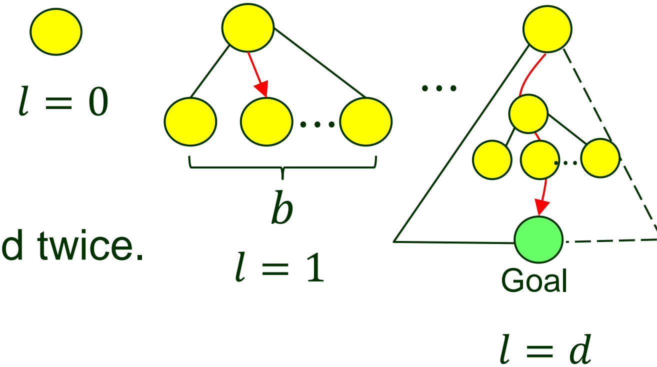
Nodes at the bottom level are generated once.

Nodes at the next-to-bottom level are generated twice.

⋮

Time and Memory of IDS

Case 1 Goal state is found at depth *d*.



Nodes at the bottom level are generated once.

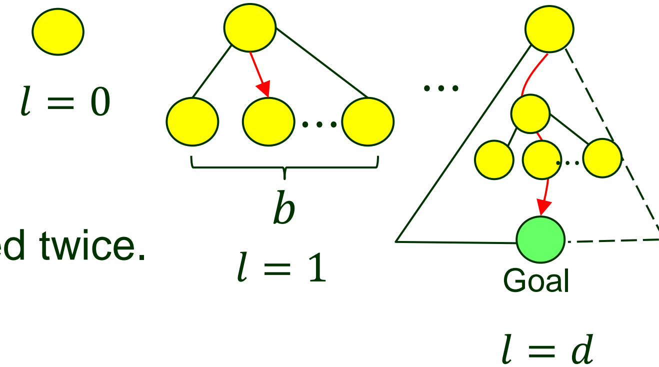
Nodes at the next-to-bottom level are generated twice.

⋮

Time ~ #nodes $\leq b^d + 2b^{d-1} + \cdots + (d-1)b^2 + db$

Time and Memory of IDS

Case 1 Goal state is found at depth *d*.



Nodes at the bottom level are generated once.

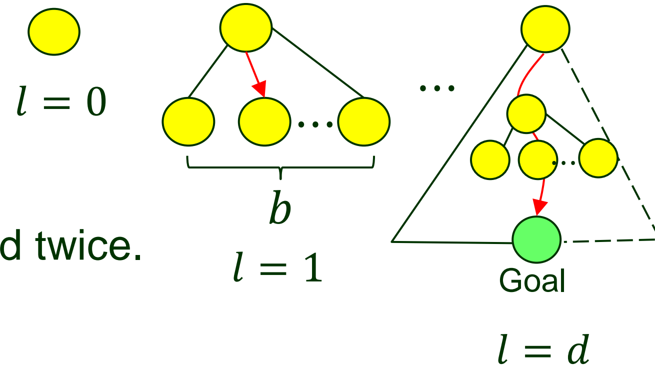
Nodes at the next-to-bottom level are generated twice.

⋮

Time ~ #nodes $\leq b^d + 2b^{d-1} + \cdots + (d-1)b^2 + db$

$= O(b^d)$

Time and Memory of IDS



Case 1 Goal state is found at depth *d*.

Nodes at the bottom level are generated once.

Nodes at the next-to-bottom level are generated twice.

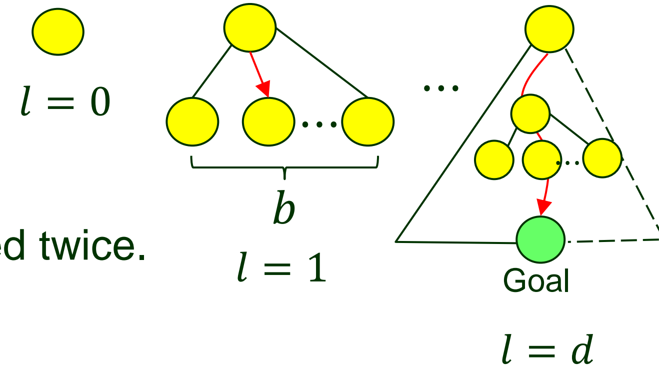
⋮

Time ~ #nodes $\leq b^d + 2b^{d-1} + \cdots + (d-1)b^2 + db$

$= O(b^d)$ Same as BFS!

Time and Memory of IDS

Case 1 Goal state is found at depth *d*.



Nodes at the bottom level are generated once.

Nodes at the next-to-bottom level are generated twice.

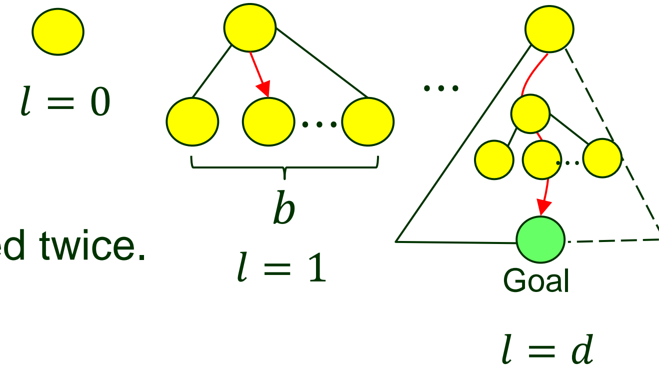
⋮

Time ~ #nodes $\leq b^d + 2b^{d-1} + \cdots + (d-1)b^2 + db$

$= O(b^d)$ Same as BFS!

Memory: $O(bd)$

Time and Memory of IDS



Case 1 Goal state is found at depth *d*.

Nodes at the bottom level are generated once.

Nodes at the next-to-bottom level are generated twice.

⋮

Time ~ #nodes $\leq b^d + 2b^{d-1} + \cdots + (d-1)b^2 + db$

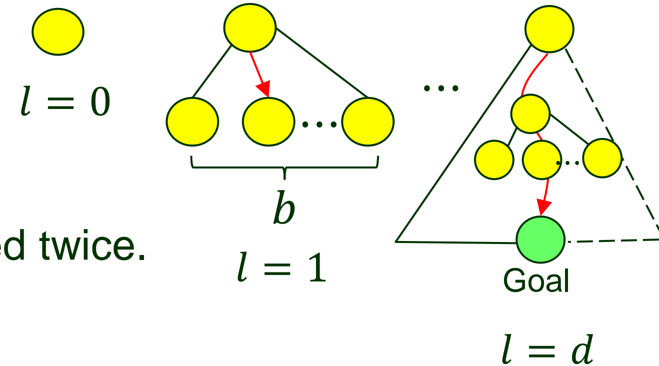
$= O(b^d)$ Same as BFS!

Memory: $O(bd)$

Case 2 Goal state is not found in a finite state space.

m: maximum number of actions in any path

Time and Memory of IDS



Case 1 Goal state is found at depth *d*.

Nodes at the bottom level are generated once.

Nodes at the next-to-bottom level are generated twice.

⋮

Time ~ #nodes $\leq b^d + 2b^{d-1} + \cdots + (d-1)b^2 + db$

$= O(b^d)$ Same as BFS!

Memory: $O(bd)$

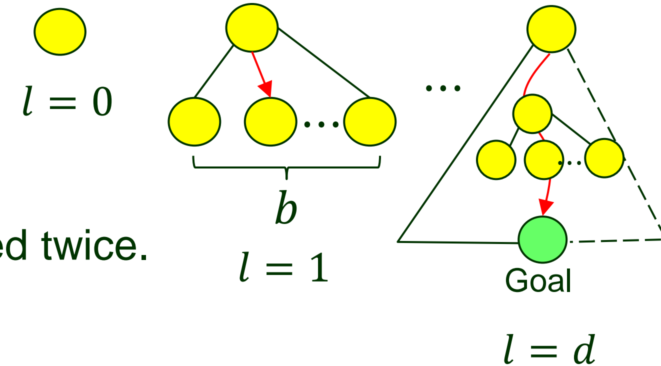
Case 2 Goal state is not found in a finite state space.

Time $O(b^m)$

Memory $O(bm)$

m: maximum number of actions in any path

Time and Memory of IDS



Case 1 Goal state is found at depth *d*.

Nodes at the bottom level are generated once.

Nodes at the next-to-bottom level are generated twice.

⋮

Time ~ #nodes $\leq b^d + 2b^{d-1} + \cdots + (d-1)b^2 + db$

$= O(b^d)$ Same as BFS!

Memory: $O(bd)$

Case 2 Goal state is not found in a finite state space.

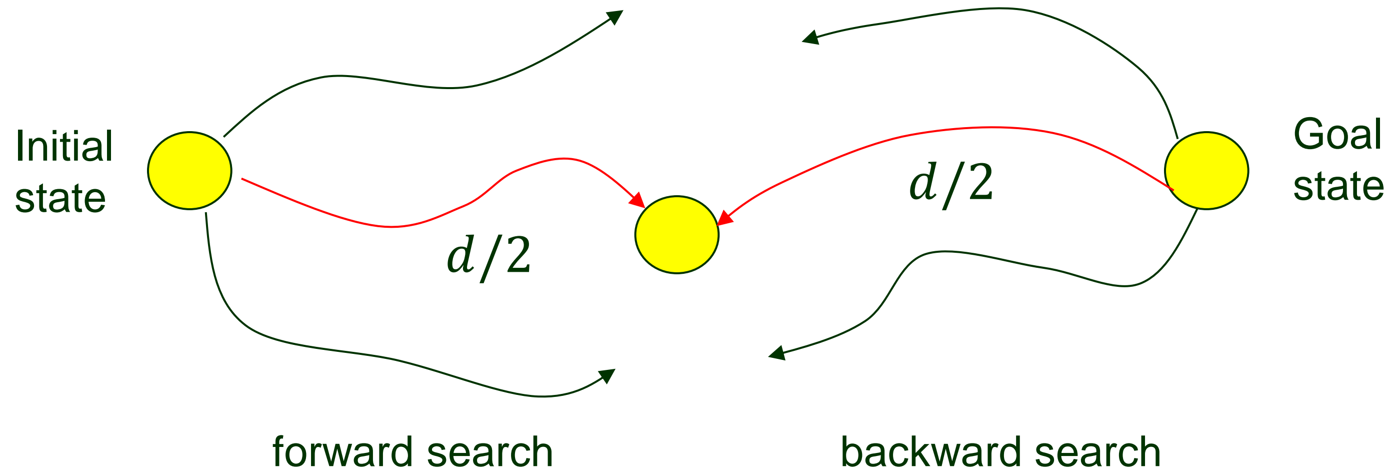
Time $O(b^m)$

Memory $O(bm)$

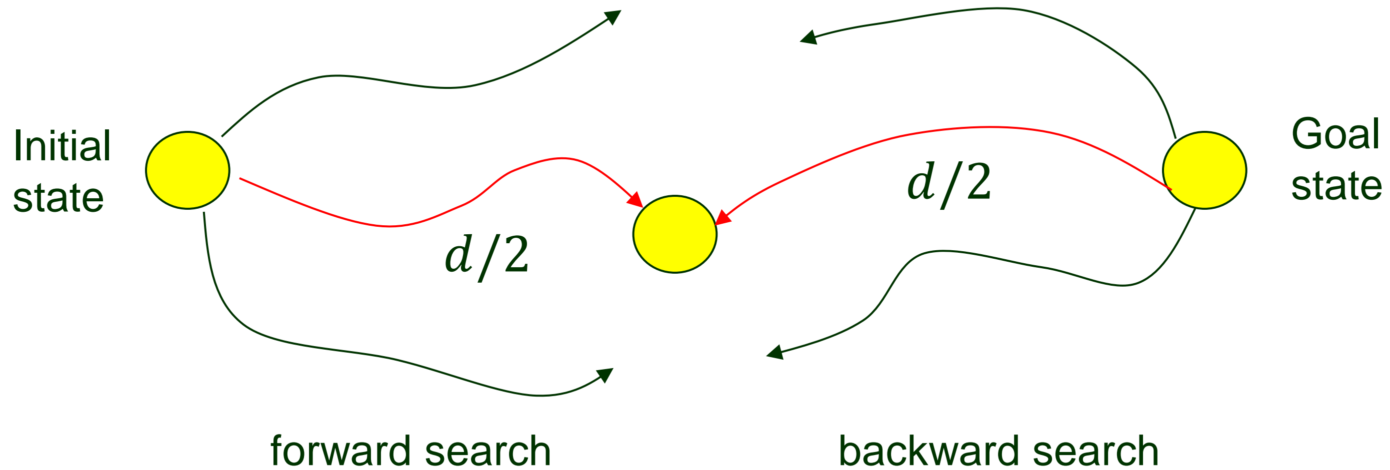
m: maximum number of actions in any path

- Modest memory requirement like DFS.

V. Bidirectional Search

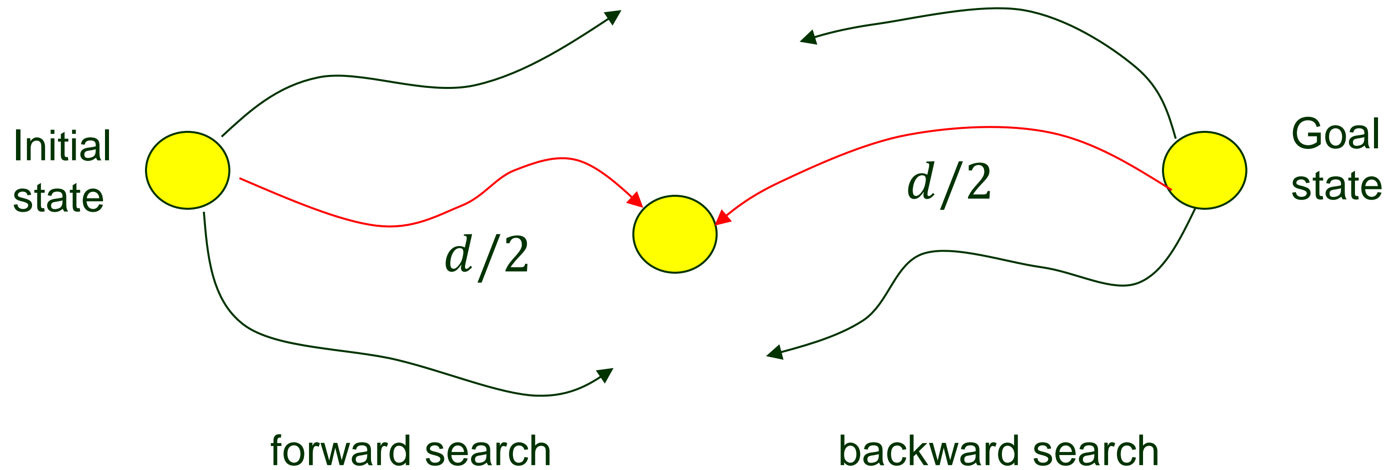


V. Bidirectional Search



Motivation: $$b^{d/2} + b^{d/2} \ll b^d$$

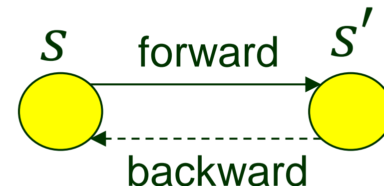
V. Bidirectional Search



Motivation:

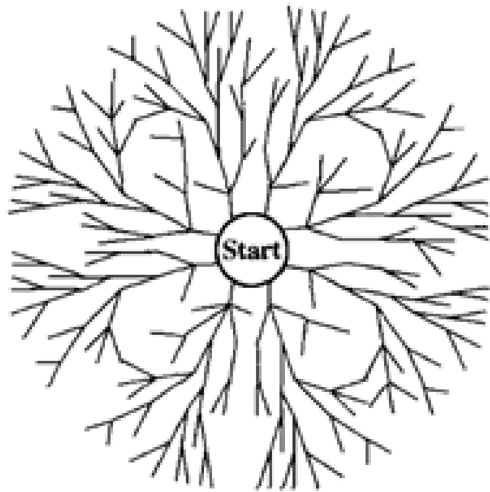
$$b^{d/2} + b^{d/2} \ll b^d$$

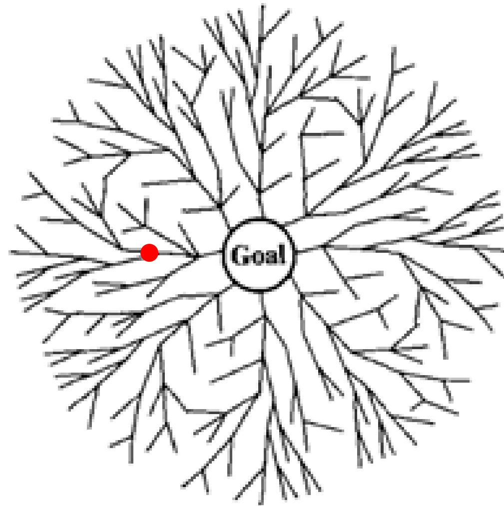
Needs to reason backwards:



s is a successor of s' in the backward direction (i.e., a predecessor in the forward direction).

Backward Reasoning





- Easy if all the actions are reversible.

  8-puzzle

  Finding a route in Romania

- Difficult to conduct if the goal is abstractly specified.

  8-queen

* Figure 3.20 in the 3rd edition

Bidirectional Best-First Search

function BIBF-SEARCH($problem_F$, f_F, $problem_B$, f_B) **returns** a solution node, or *failure*
  $node_F \leftarrow$ NODE($problem_F$.INITIAL) // *Node for a start state*
  $node_B \leftarrow$ NODE($problem_B$.INITIAL) // *Node for a goal state*
  $frontier_F \leftarrow$ a priority queue ordered by f_F, with $node_F$ as an element
  $frontier_B \leftarrow$ a priority queue ordered by f_B, with $node_B$ as an element
  $reached_F \leftarrow$ a lookup table, with one key $node_F$.STATE and value $node_F$
  $reached_B \leftarrow$ a lookup table, with one key $node_B$.STATE and value $node_B$
  $solution \leftarrow failure$
  while not TERMINATED($solution$, $frontier_F$, $frontier_B$) **do**
    if f_F(TOP($frontier_F$)) $<$ f_B(TOP($frontier_B$)) **then**
      $solution \leftarrow$ PROCEED(F, $problem_F$ $frontier_F$, $reached_F$, $reached_B$, $solution$)
    else $solution \leftarrow$ PROCEED(B, $problem_B$, $frontier_B$, $reached_B$, $reached_F$, $solution$)
  return $solution$

// Expand node on frontier; check if any child node has a state
// already reached in the other frontier.

Comparing Uninformed Search Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes[1]	Yes[1,2]	No	No	Yes[1]	Yes[1,4]
Optimal cost?	Yes[3]	Yes	No	No	Yes[3]	Yes[3,4]
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$

b: branching factor (> 1)

d: minimum depth of a solution

l: depth limit

m: maximum search tree depth

C^*: optimal solution cost

ϵ: minimum action cost

1. if b is finite, and the state space either has a solution or is finite.
2. if all costs are $\geq \varepsilon > 0$.
3. if all costs identical.
4. if both directions are breadth-first or uniform-cost.