Lecture 3

Note: Some slides and/or pictures are adapted from Lecture slides / Books of

- Dr Zafar Alvi.
- Text Book - *Aritificial Intelligence Illuminated* by Ben Coppin, Narosa Publishers.
- Ref Books
  - *Artificial Intelligence- Structures & Strategies for Complex Problem Solving by* George F. Luger, 4th edition, Pearson Education.
  - *Artificial Intelligence A Modern Approach* by Stuart Russell & Peter Norvig.
  - Artificial Intelligence, Third Edition by Patrick Henry Winston

Outline

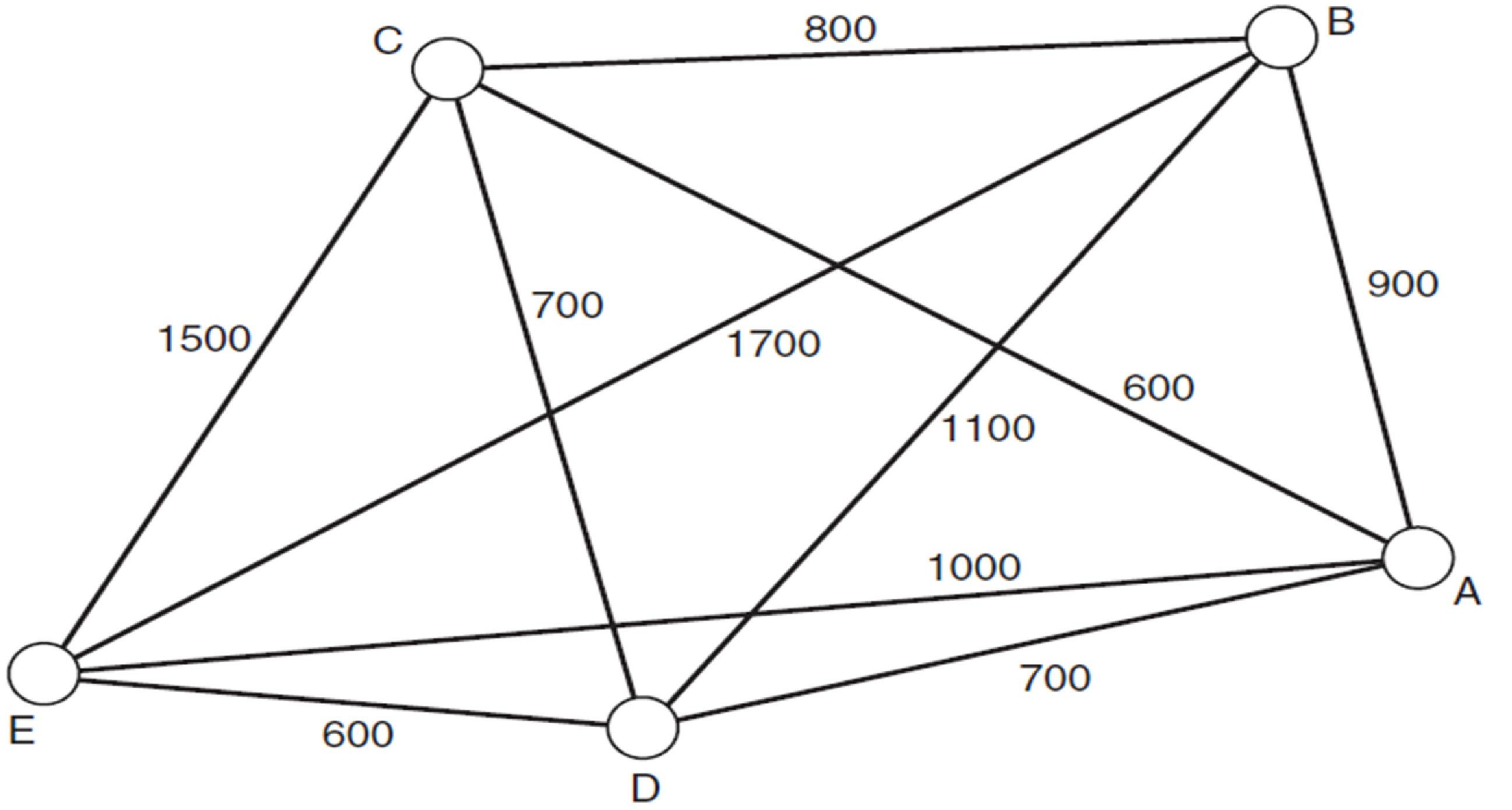
- Problems and their representation
- Goal Driven VS Data Driven
- Properties of search Methods
- Tree search algorithm
  - Depth First algorithm
  - Breadth First algorithm
  - Iterative Deepening algorithm

Problems and their Representations

- *Three men and three lions are on one side of a river, with a boat. They all want to get to the other side of the river. The boat can only hold one or two at a time. At no time should there be more lions than men on either side of the river, as this would probably result in the men being eaten.*
- *Representation could be* 3, 3, 1   0, 0, 0

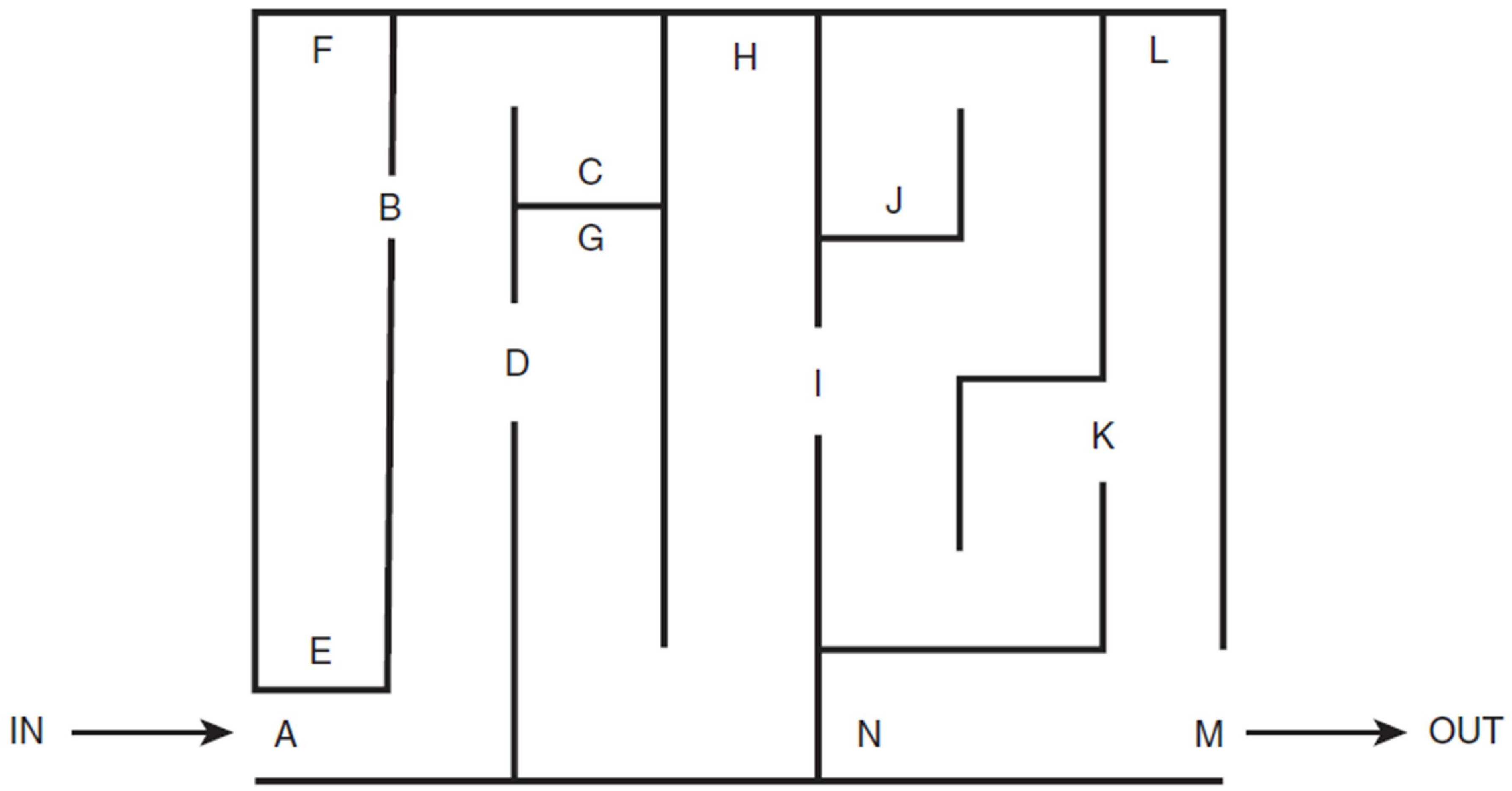
Problems and their Representations

- Traveling Salesman problem (NP complete)



Problems and their Representations

- Traversing a Maze



Data Driven or Goal Driven Search

- **Data-driven search** starts from an initial state and uses actions that are allowed to move forward until a goal is reached. This approach is also known as **forward chaining.**
- Search can start at the goal and work back toward a start state, by seeing what moves could have led to the goal state. This is **goal-driven** search, also known as **backward chaining.**

Properties of Search Methods

- Complexity
- Completeness
- Optimality
- Admissibility
- Irrevocability

Complexity

- it is useful to describe how efficient that method is, over time and space.
- The **time complexity** of a method is related to the length of time that the method would take to find a goal state.
- The **space complexity** is related to the amount of memory that the method needs to use.

Completeness

- A search method is described as being **complete** if it is guaranteed to find a goal state if one exists.
- A method that is not complete has the disadvantage that it cannot necessarily be believed if it reports that no solution exists.

Optimality

- A search method is **optimal** if it is guaranteed to find the best solution that exists.
- In other words, it will find the path to a goal state that involves taking the least number of steps.
- This does not mean that the search method itself is efficient—it might take a great deal of time for an optimal search method to identify the optimal solution—but once it has found the solution, it is guaranteed to be the best one.

Admissibility

- In some cases, the word *optimal* is used to describe an algorithm that finds a solution in the quickest possible time, in which case the concept of **admissibility** is used in place of optimality.
- An algorithm is then defined as **admissible** if it is guaranteed to find the best solution.

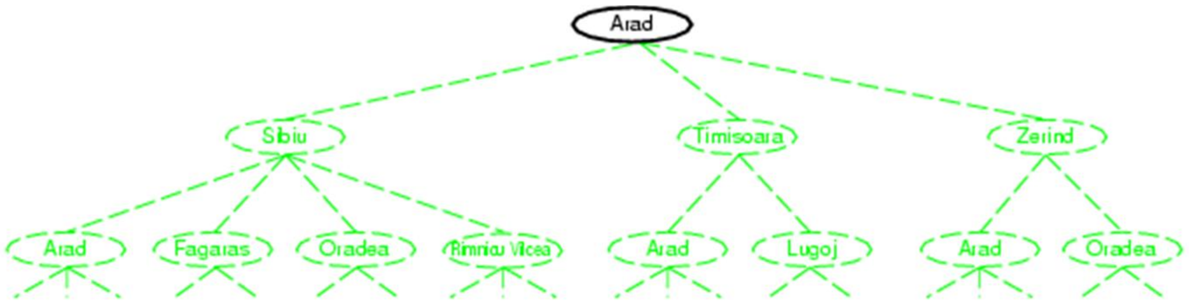
Irrevocability

- Methods that do not use backtracking, and which therefore examine just one path, are described as irrevocable.
- Irrevocable search methods will often find suboptimal solutions to problems because they tend to be fooled by local optima—solutions that look good locally but are less favorable when compared with other solutions elsewhere in the search space.

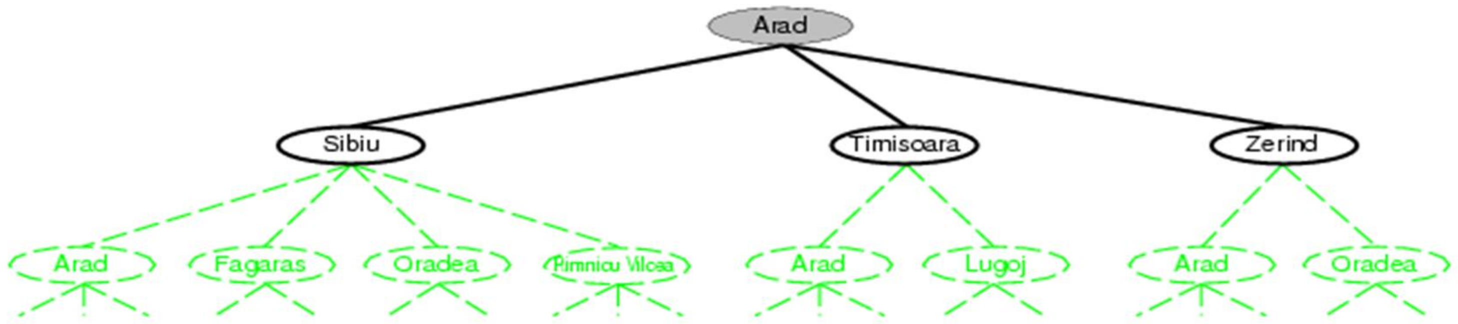
Tree search algorithms

- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored

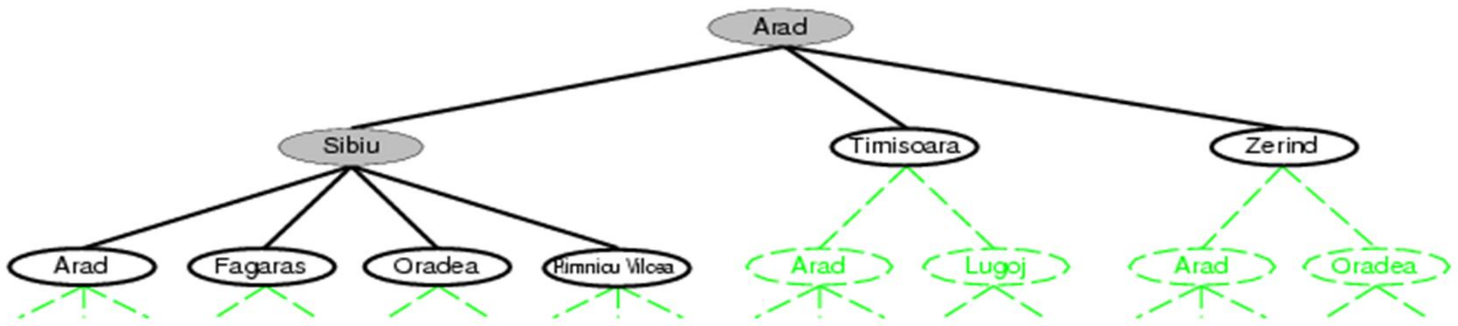
Tree search example



Tree search example



Tree search example

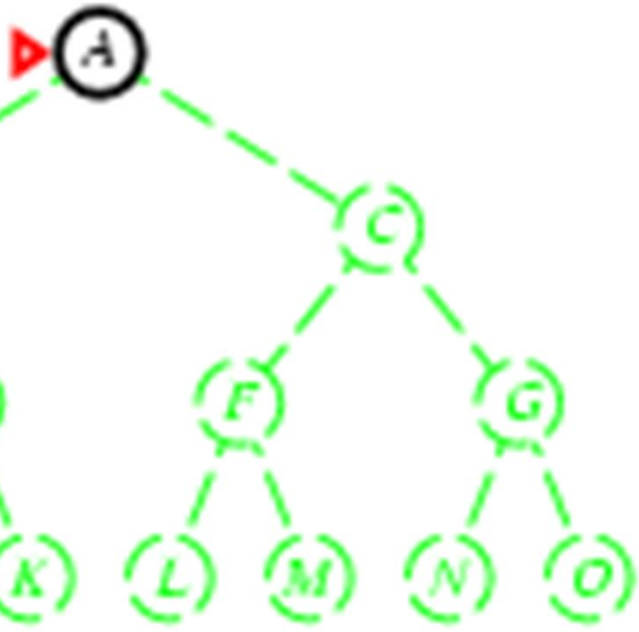


Uninformed search strategies

- Uninformed search strategies use only the information available in the problem definition
  - Depth-first search
  - Breadth-first search
  - Iterative deepening search
  - etc

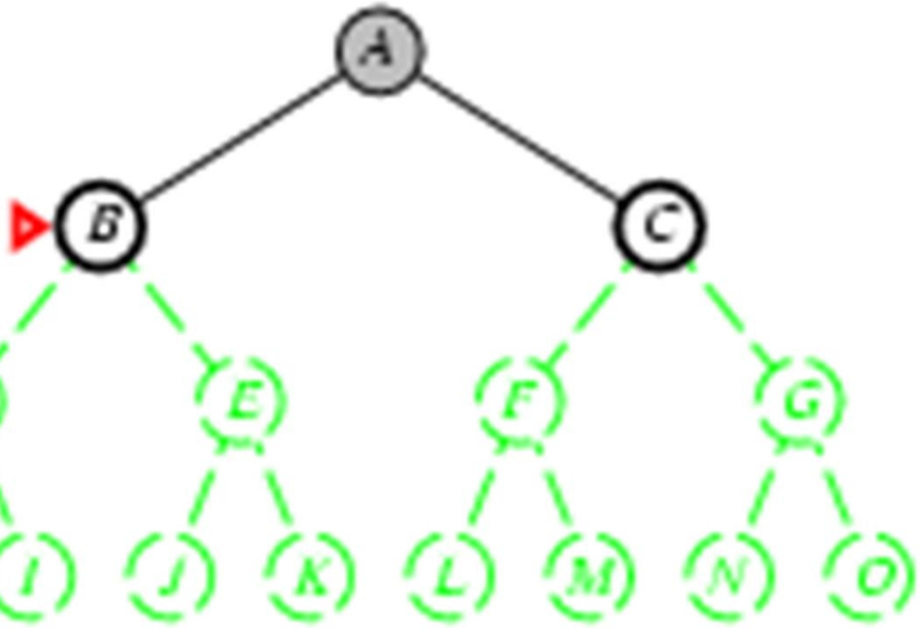
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - put successors at front
  -



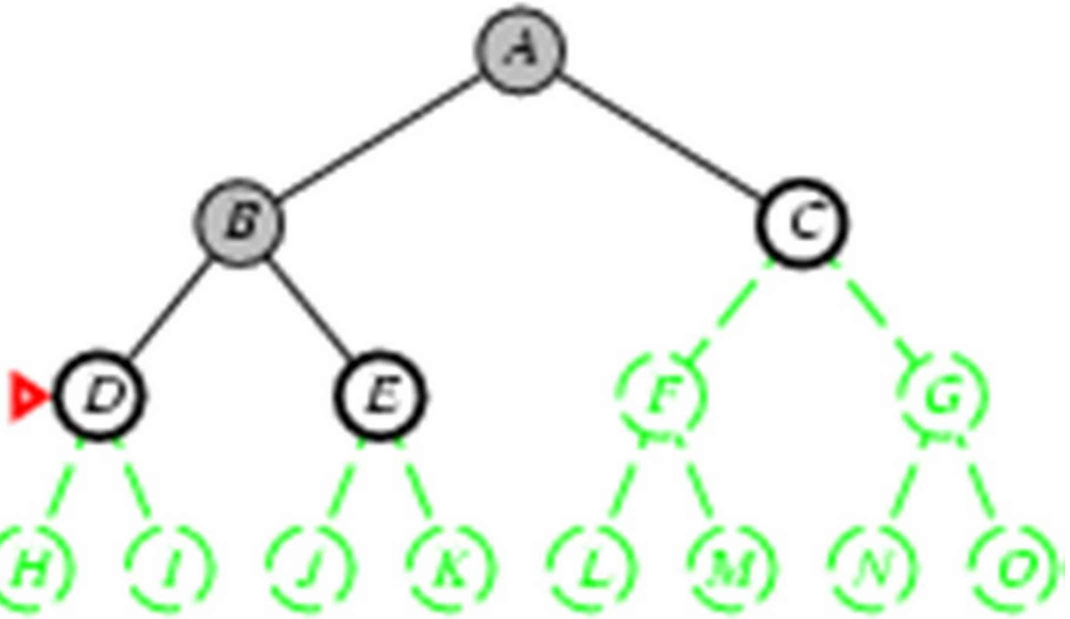
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - put successors at front
  -



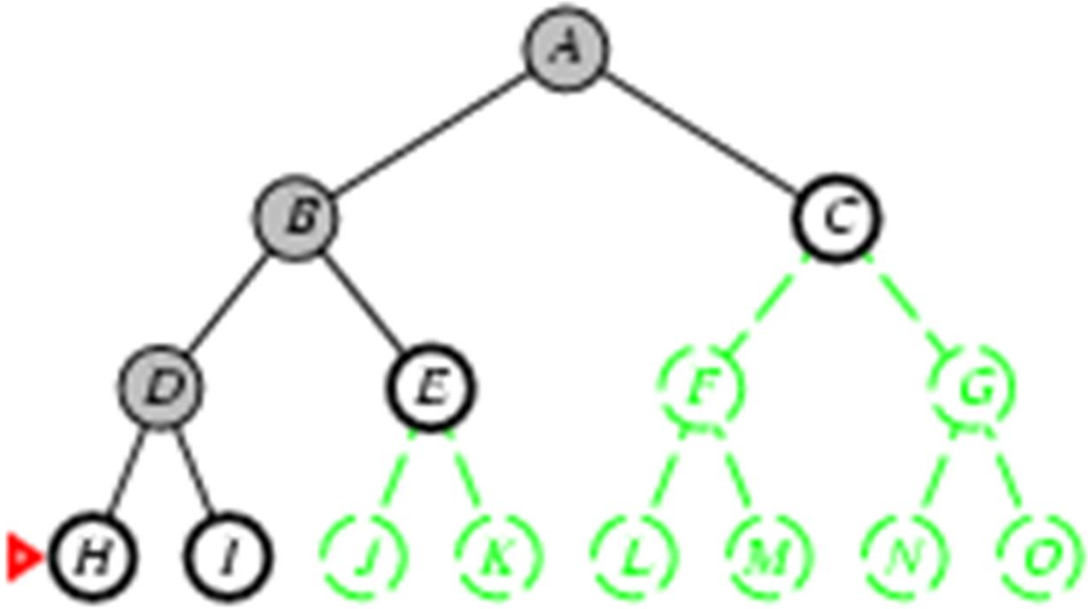
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - put successors at front
  -



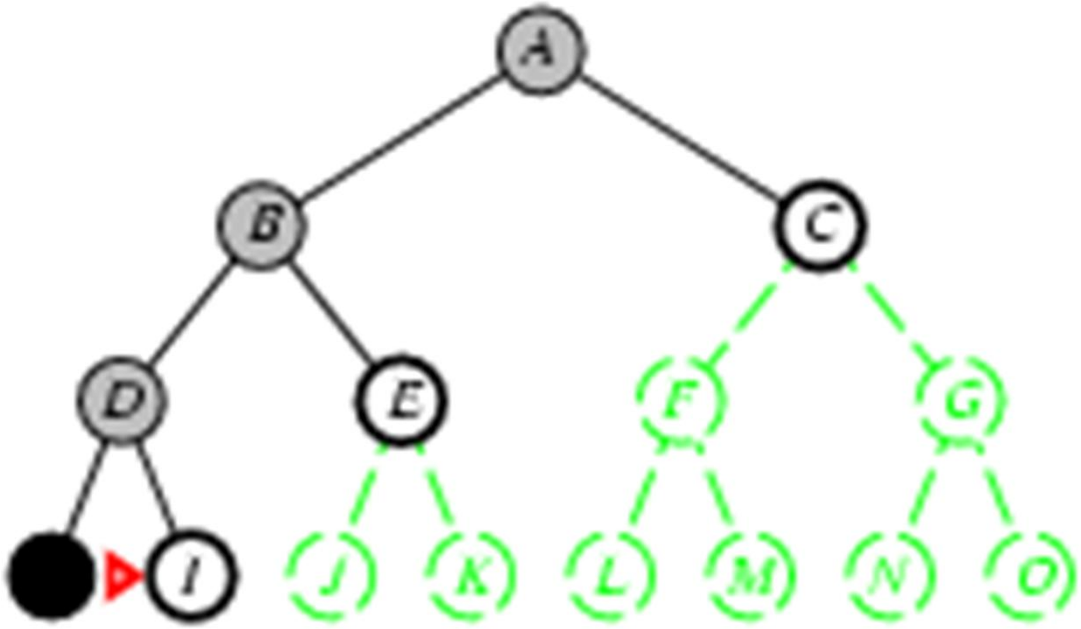
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - put successors at front
  -



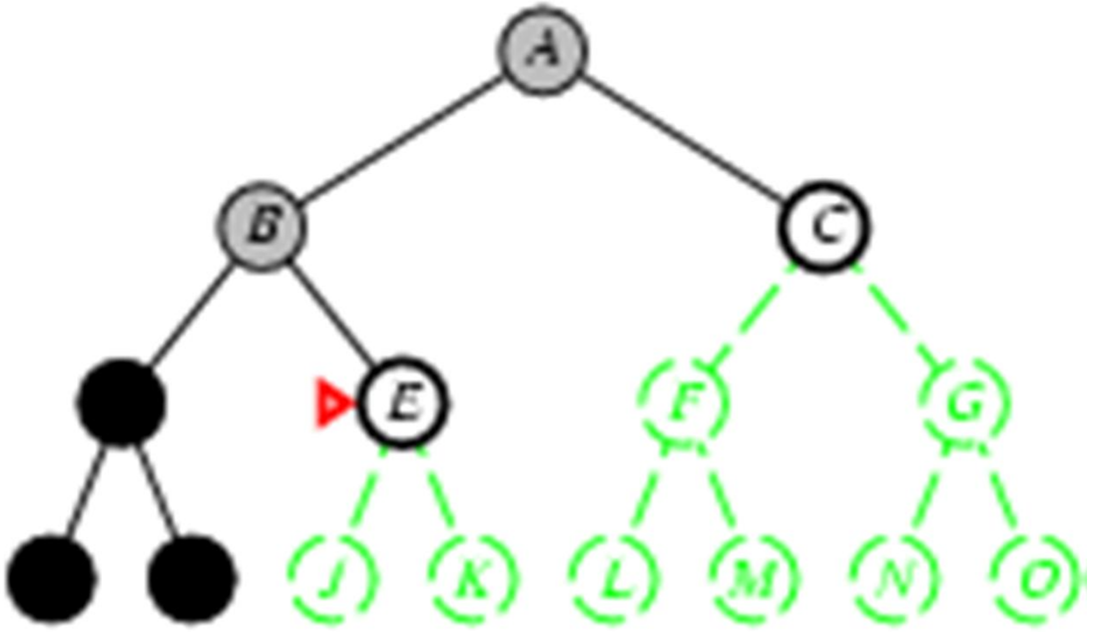
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - put successors at front
  -



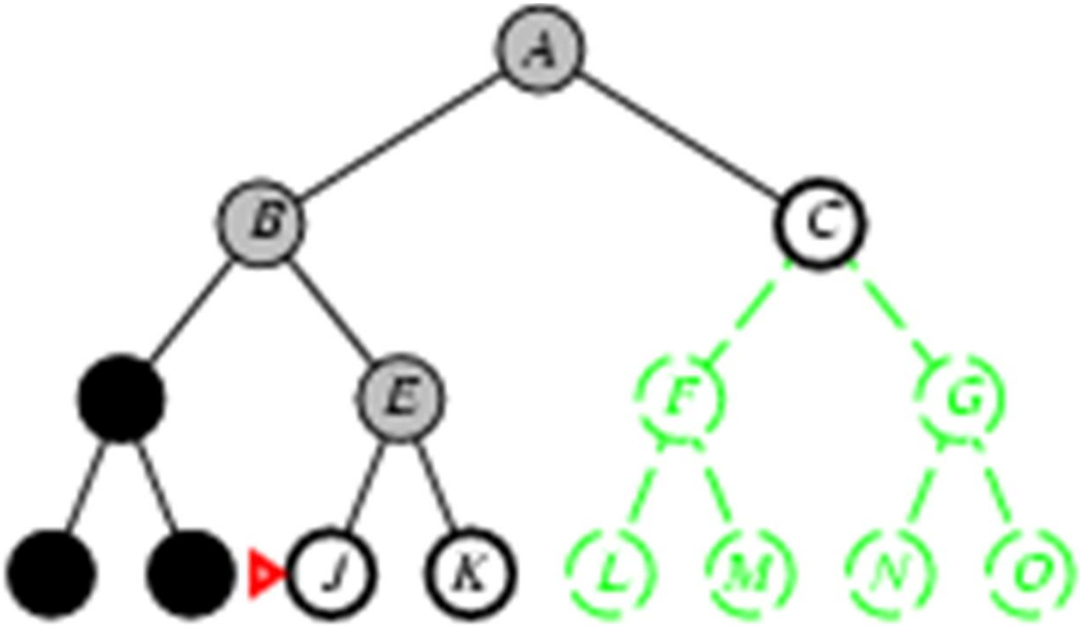
Depth-first search

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



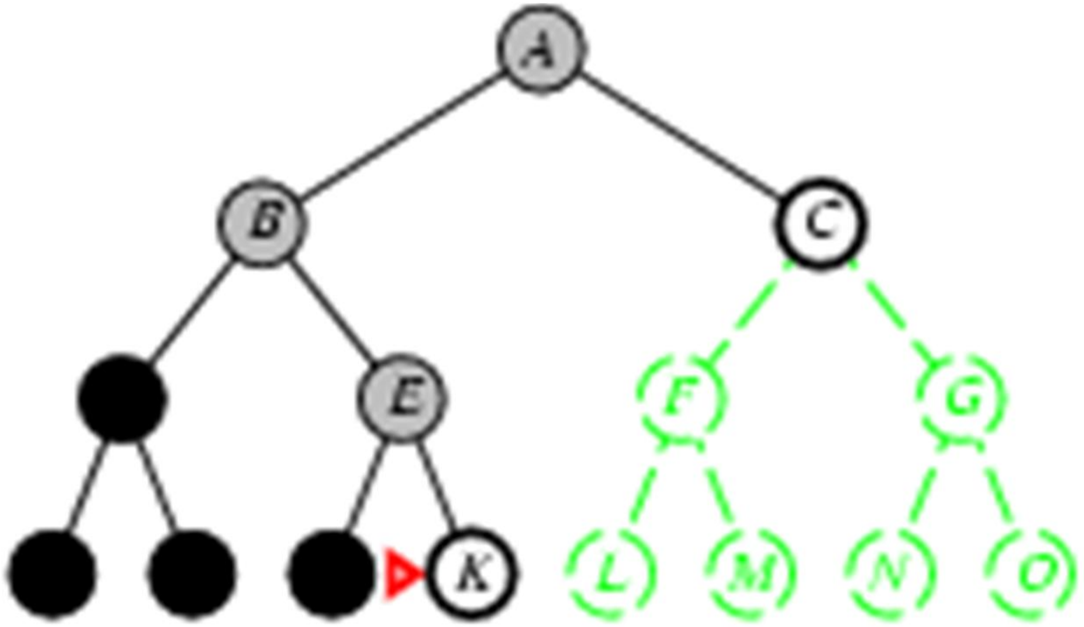
Depth-first search

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



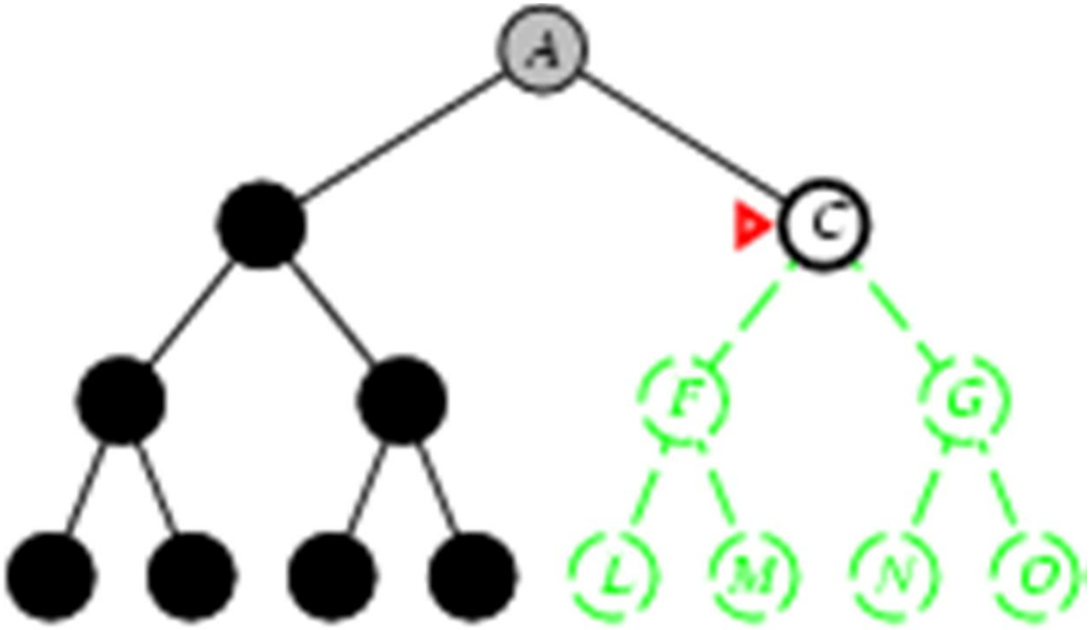
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - put successors at front
  -



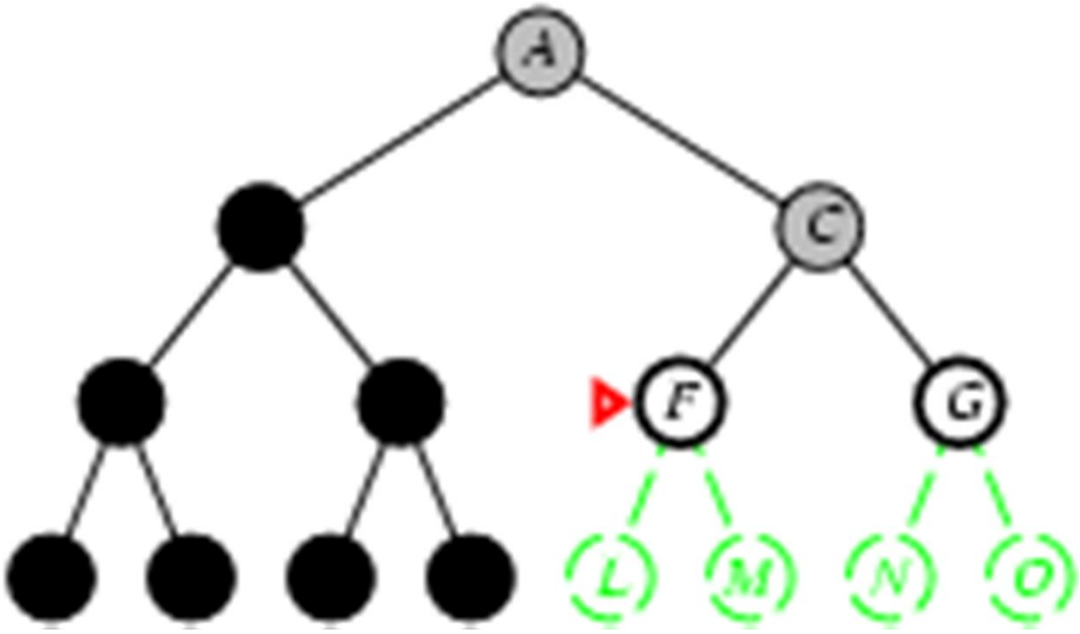
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - put successors at front
  -



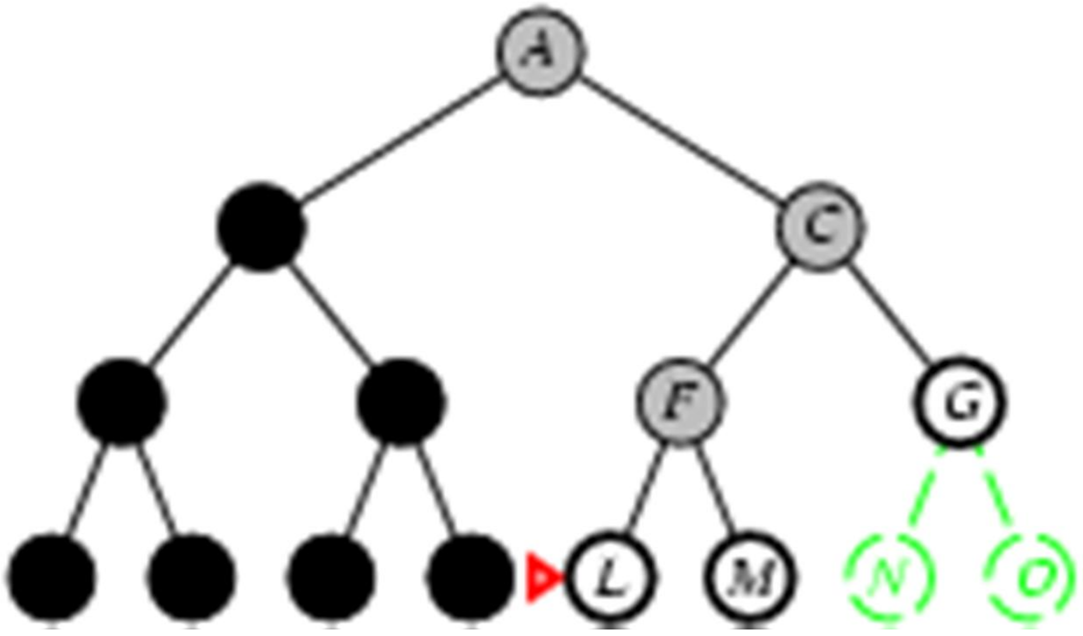
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - put successors at front
  -



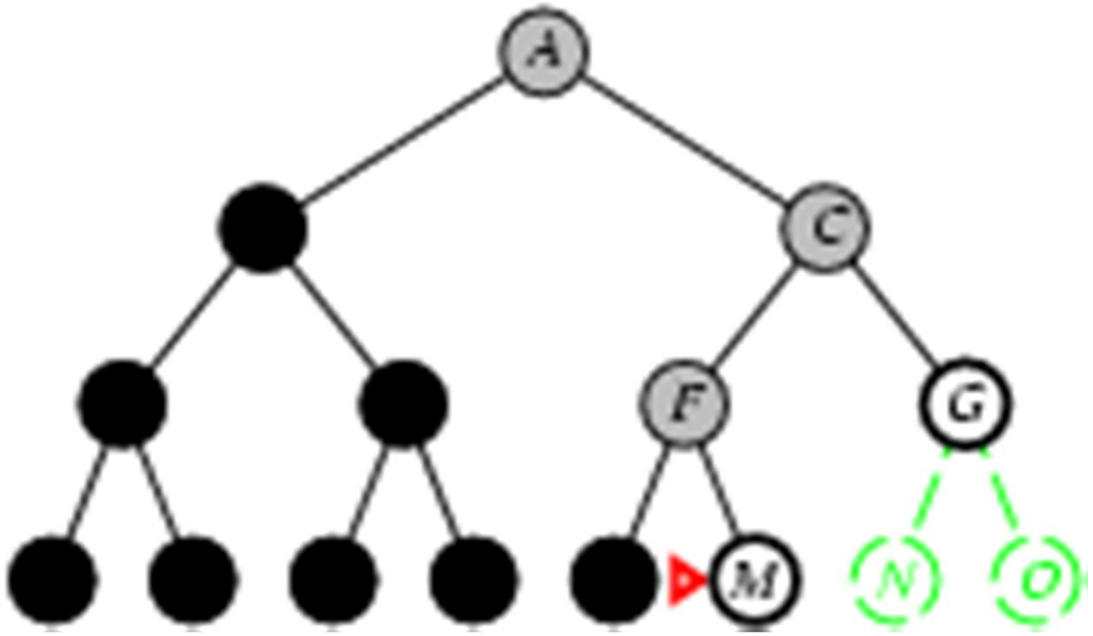
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - put successors at front
  -



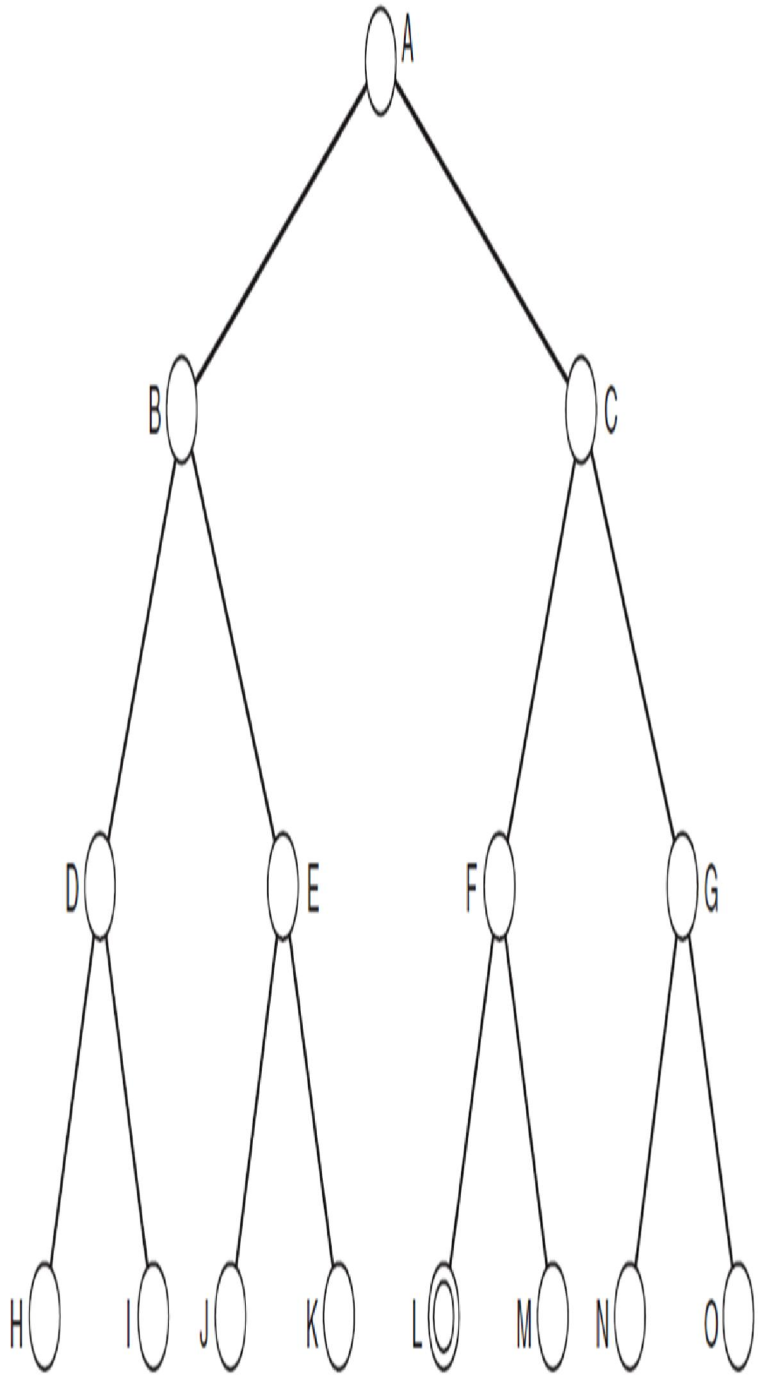
Depth-first search

- Expand deepest unexpanded node
- Implementation:
  - put successors at front
  -



Depth-first search

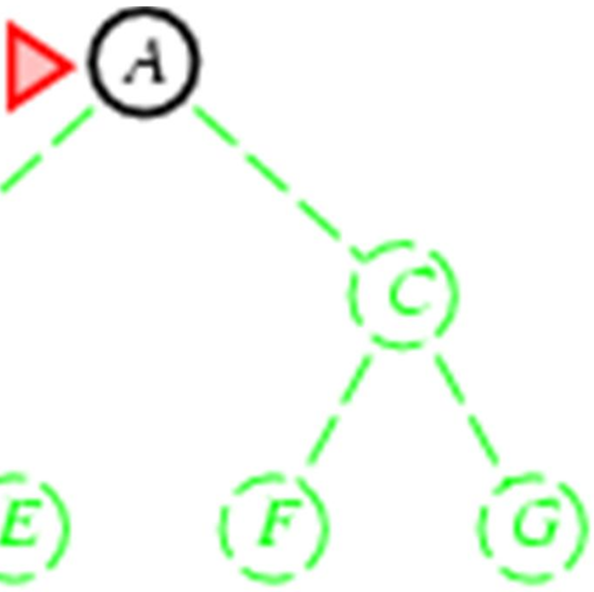
```
Function depth ()
{
    queue = []; // initialize an empty queue
    state = root_node; // initialize the start state
    while (true)
   {
        if is_goal (state)
           then return SUCCESS
        else add_to_front_of_queue (successors (state));

        if queue == []
        then report FAILURE;
        state = queue [0]; // state = first item in queue
        remove_first_item_from (queue);
   }
}
```



Step	State	Queue
1	A	(empty)
2	A	B,C
3	B	C
4	B	D,E,C
5	D	E,C
6	D	H,I,E,C
7	H	I,E,C
8	I	E,C
9	E	C
10	E	J,K,C
11	J	K,C
12	K	C
13	C	(empty)
14	C	F,G
15	F	G
16	F	L,M,G
17	L	M,G

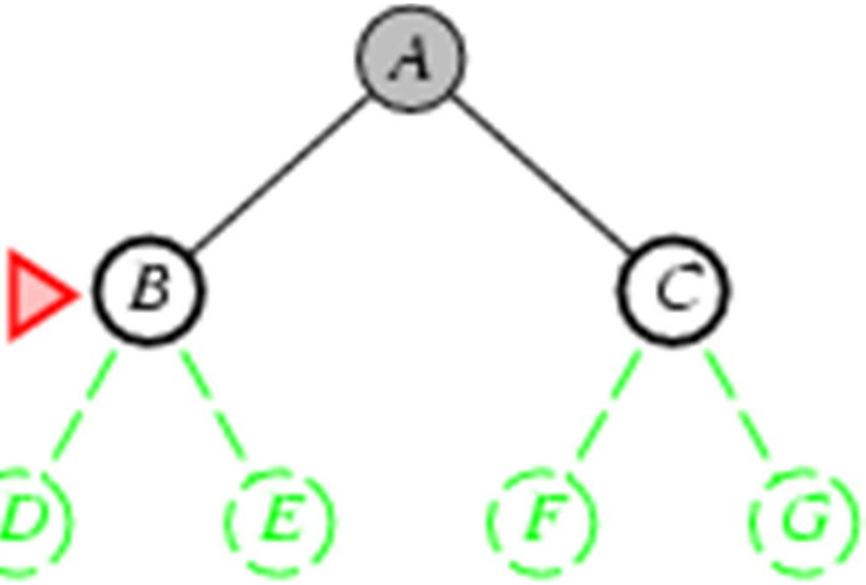
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - new successors go at end
  -



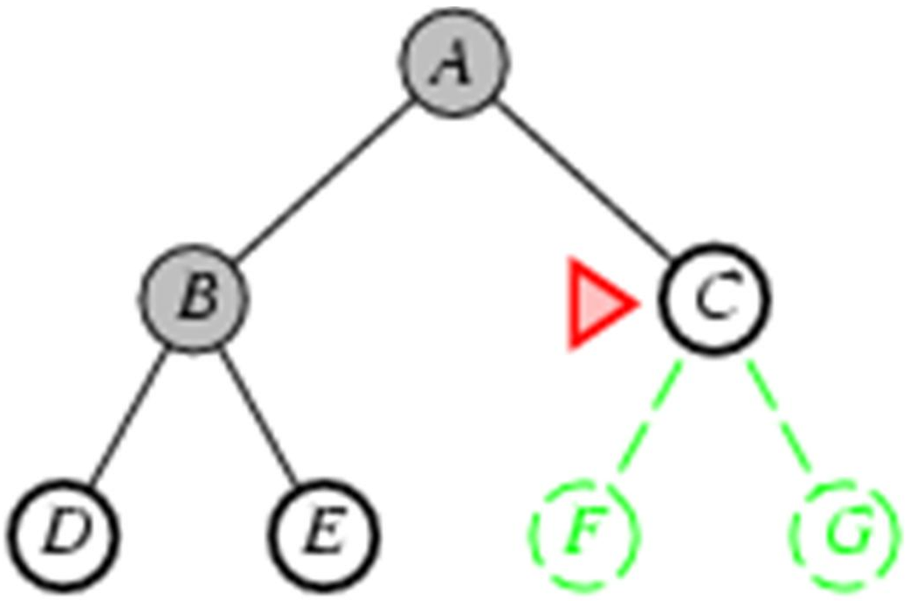
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - new successors go at end
  -



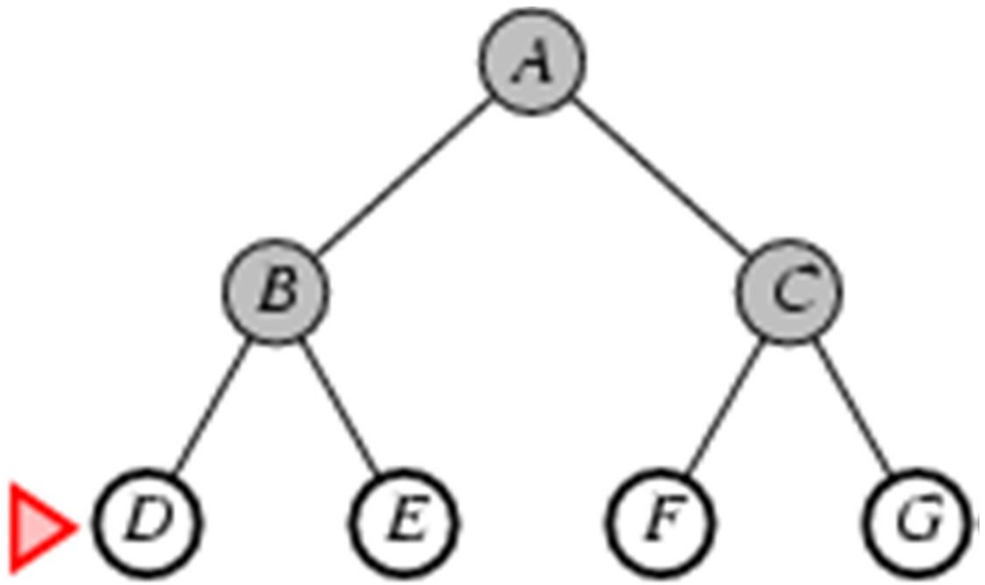
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - new successors go at end
  -



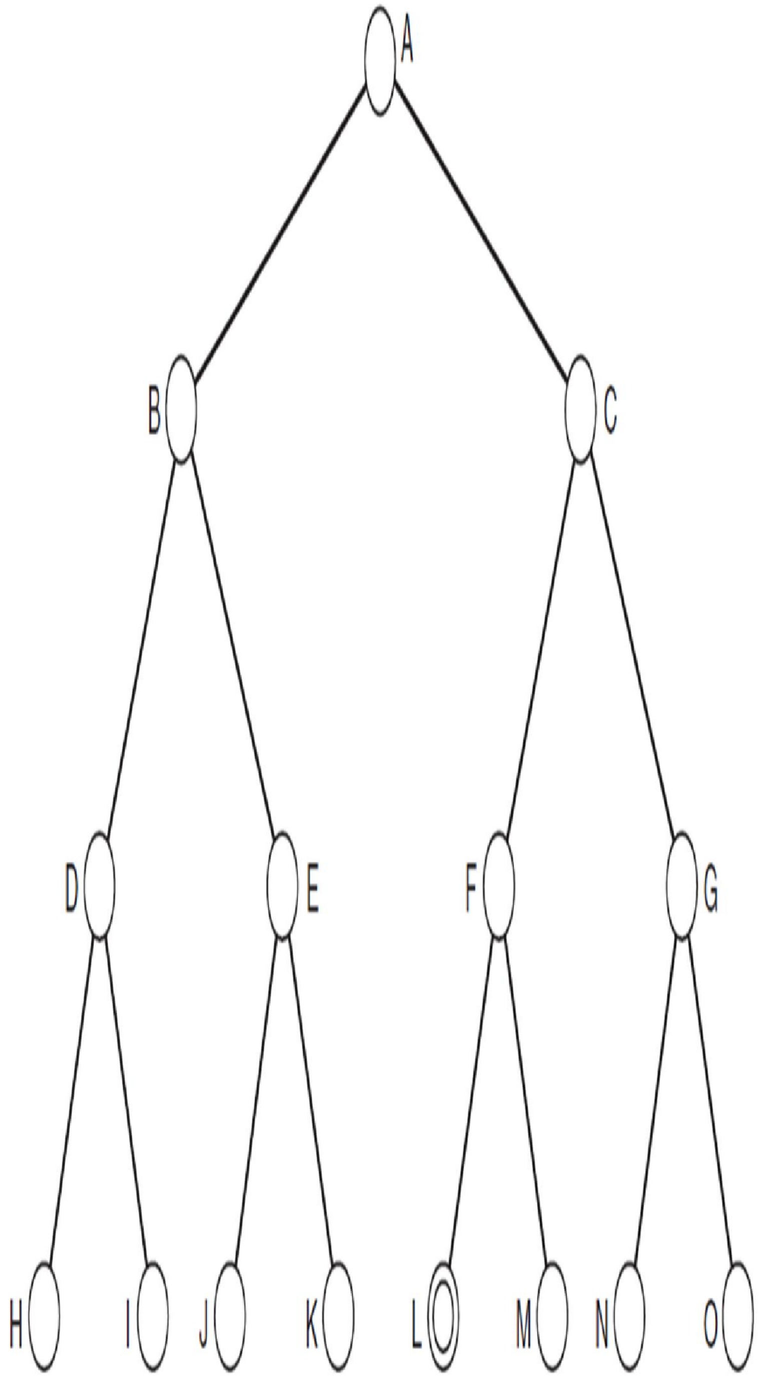
Breadth-first search

- Expand shallowest unexpanded node
- Implementation:
  - *fringe* is a FIFO queue, i.e., new successors go at end
  -



Breadth-first search

```
Function breadth ()
{
    queue = []; // initialize an empty queue
    state = root_node; // initialize the start state
    while (true)
    {
        if is_goal (state)
        then return SUCCESS
        else add_to_back_of_queue (successors (state));
        if queue == []
        then report FAILURE;
        state = queue [0]; // state = first item in queue
        remove_first_item_from (queue);
    }
}
```

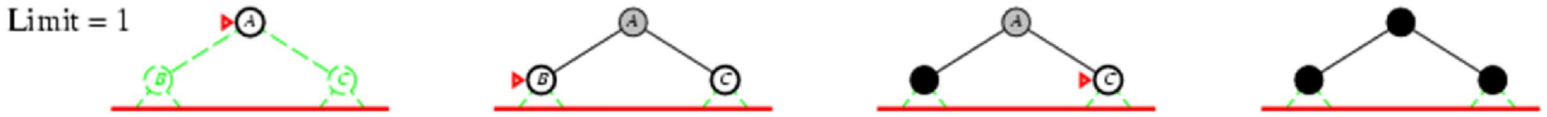


Step	State	Queue
1	A	(empty)
2	A	B,C
3	B	C
4	B	C,D,E
5	C	D,E
6	C	D,E,F,G
7	D	E,F,G
8	D	E,F,G,H,I
9	E	F,G,H,I
10	E	F,G,H,I,J,K
11	F	G,H,I,J,K
12	F	G,H,I,J,K,L,M
13	G	H,I,J,K,L,M
14	G	H,I,J,K,L,M,N,O
15	H	I,J,K,L,M,N,O
16	I	J,K,L,M,N,O
17	J	K,L,M,N,O
18	K	L,M,N,O
19	L	M,N,O

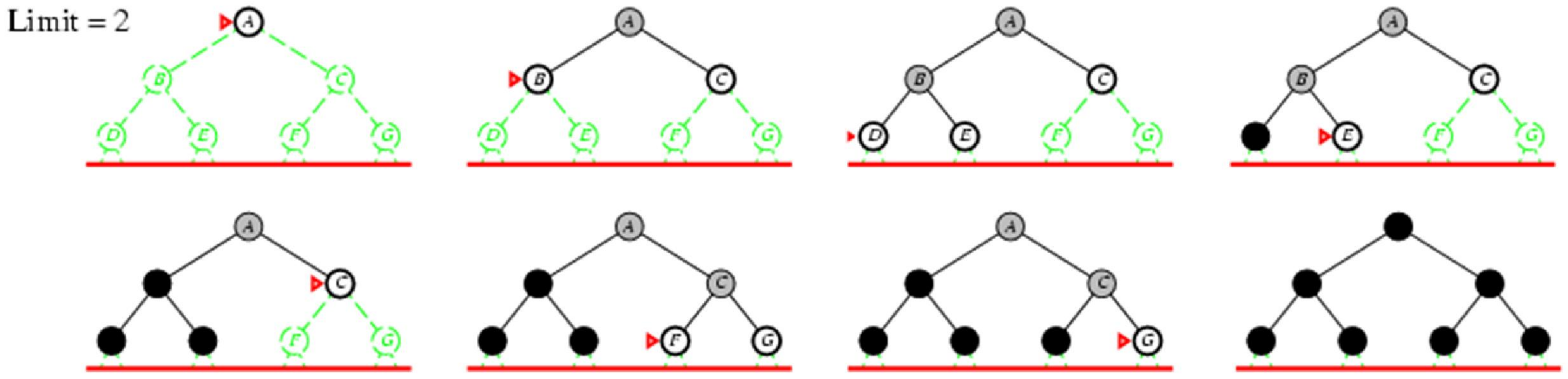
Iterative deepening search *l* =0

Limit = 0

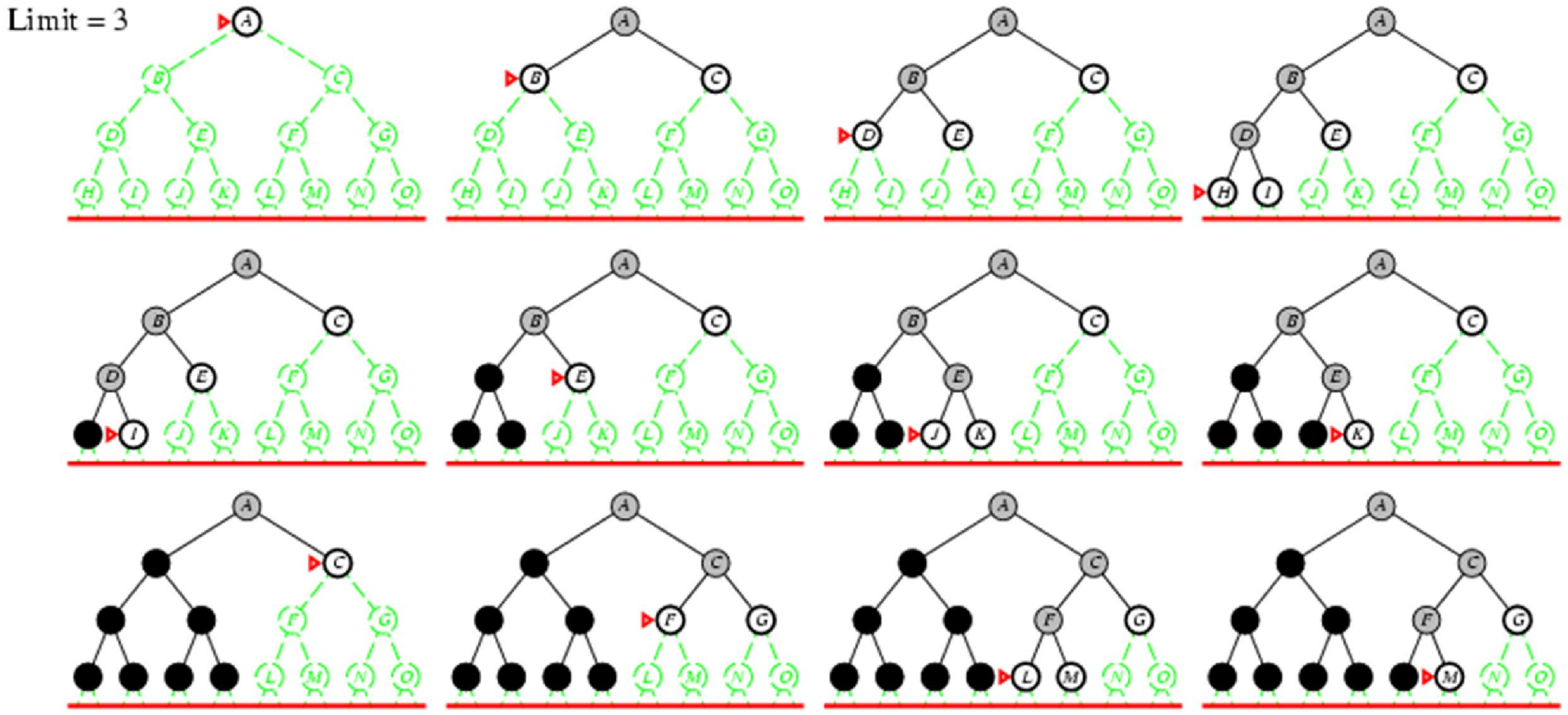
Iterative deepening search *l* =1



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*:

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

- GP $$\frac{1 - b^{d+1}}{1 - b}$$

- 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%