



مجموعة جامعة
الجميع الذكية
للتعليم القابضة



Master of Science in Artificial intelligent

Lecture 4

AI511

Foundations of Artificial Intelligence

Prepared by
Prof. Lamiaa Fattouh Ibrahim

www.midocean.ae

- **Search: Basic idea**
 - **Search tree**
 - **Tree Search Algorithm Outline**
 - **Handling repeated states**
 - **Backtracking Search**
 - **Backtracking Algorithm Data Structure**
 - **Backtracking Algorithm**

Outline

- **Blind vs. Heuristic Strategies**
- **Blind Strategies**
- **Depth-First Strategy**
- **Breadth- First Strategy**

Given:

Initial state

Actions

Transition model

Goal state

Path cost

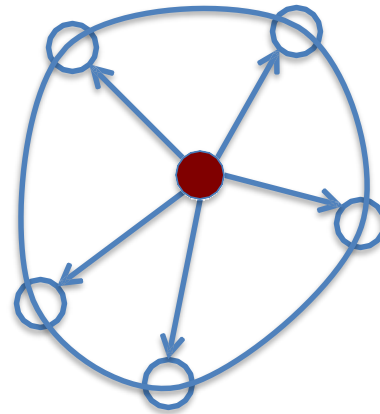
How do we find the optimal solution?

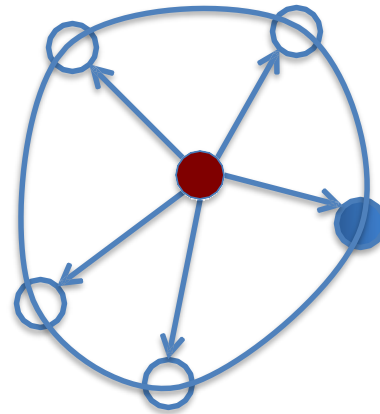
- Let's begin at the start state and **expand** it by making a list of all possible successor states.

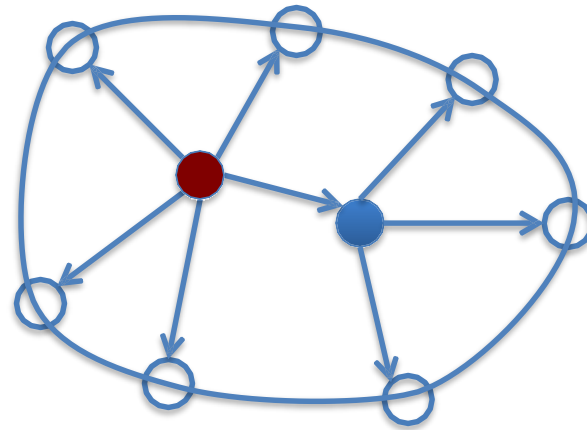
successor states → frontier
- Maintain a **frontier (or fringe)** or a list of unexpanded states.
- At each step, pick a state from the frontier to expand.
- Keep going until you reach a goal state.
- Try to expand as few states as possible.

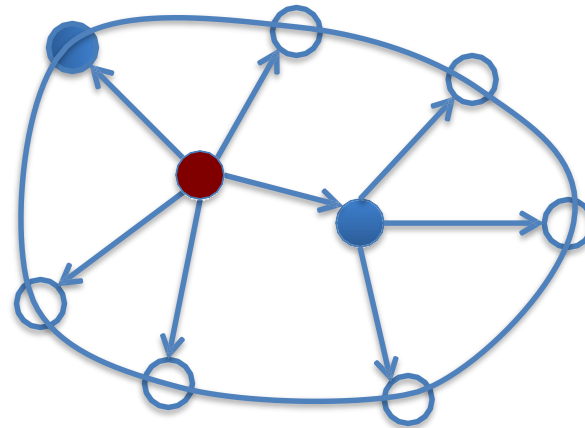
Start state

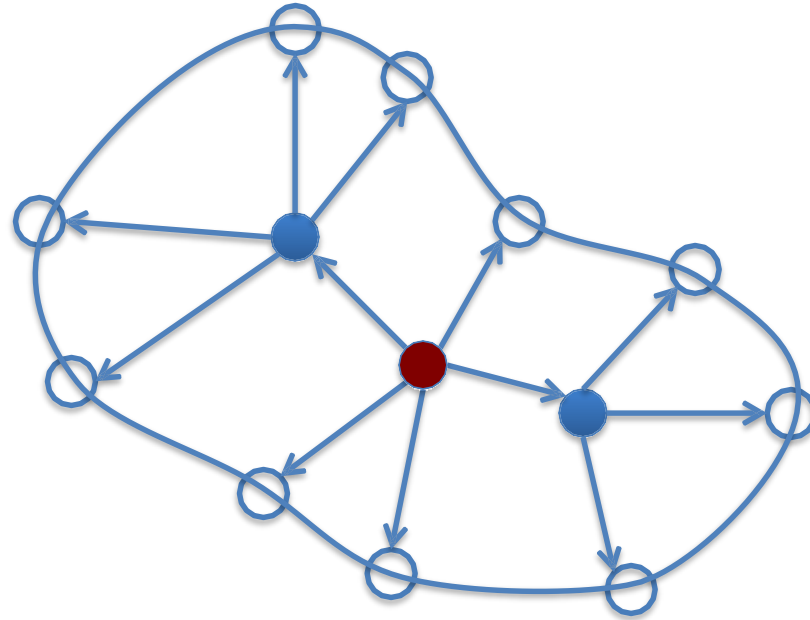


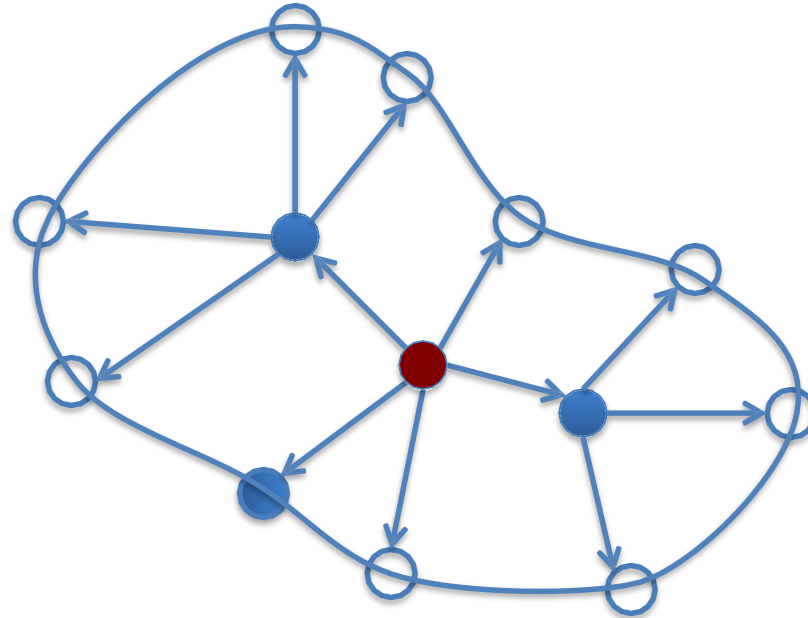


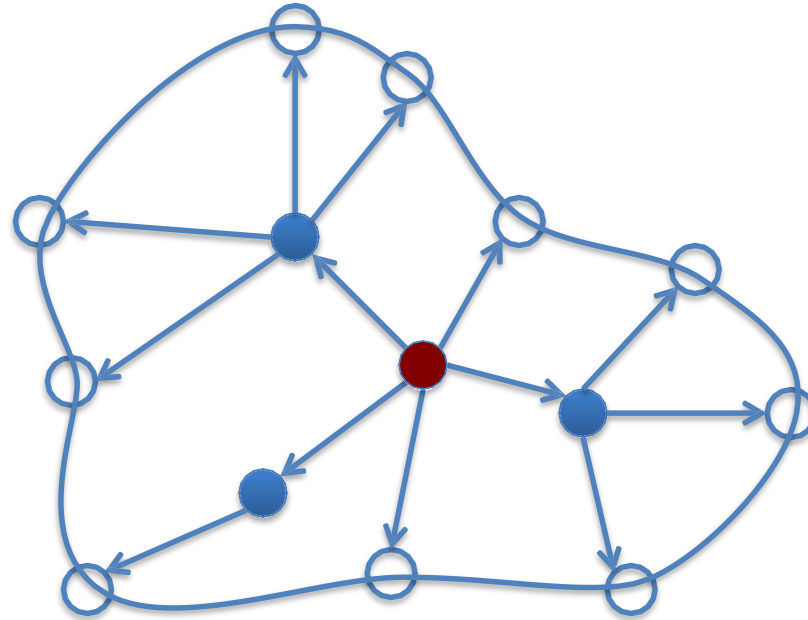


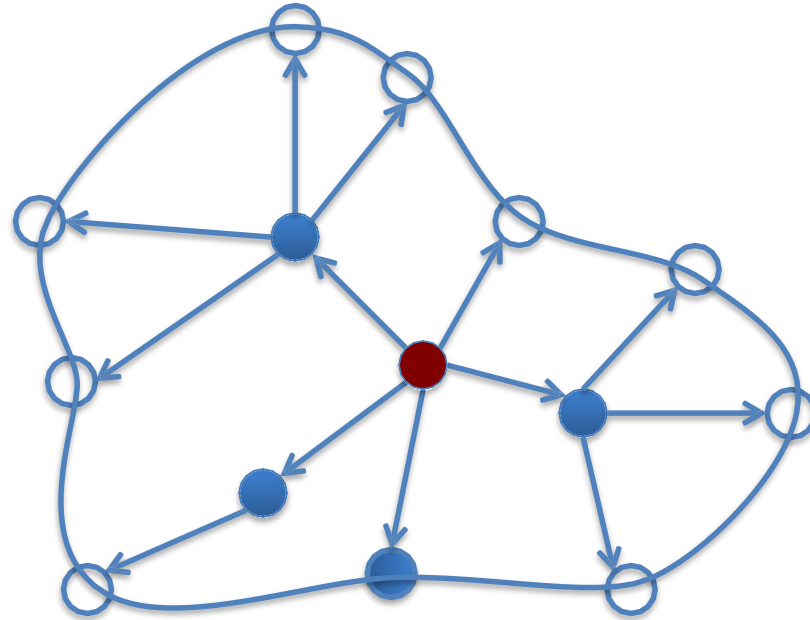


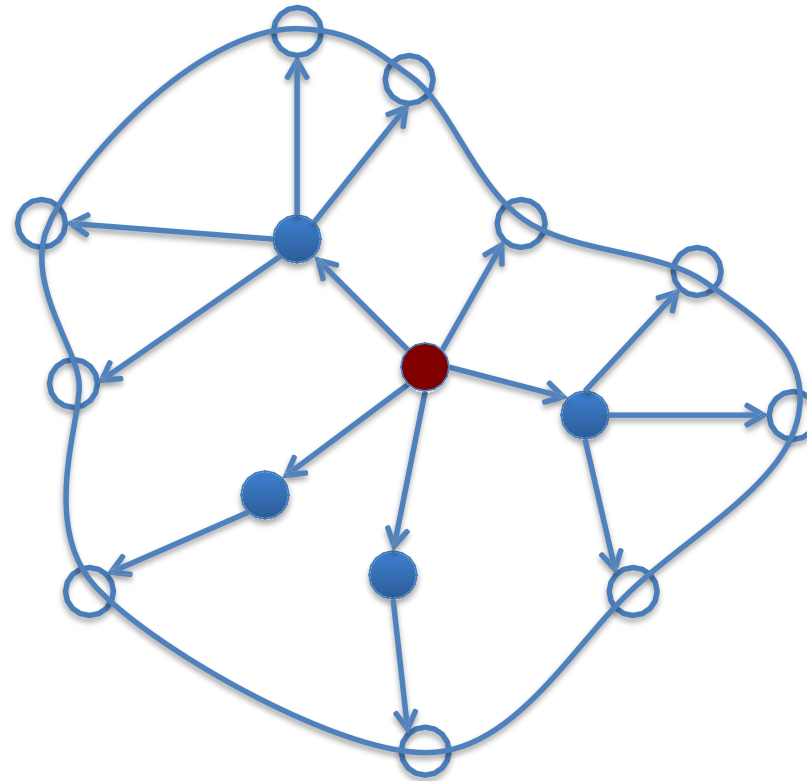


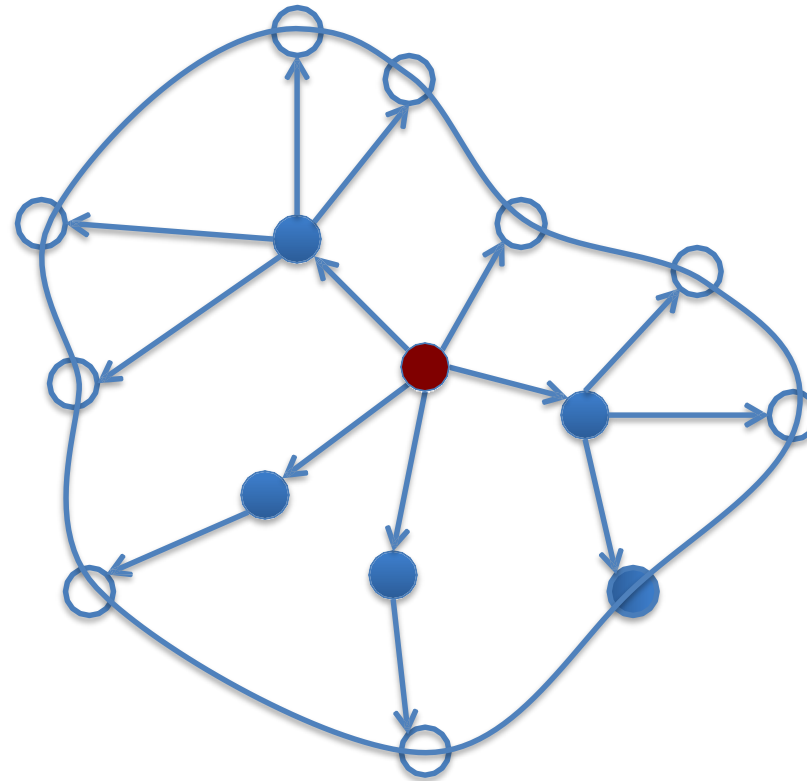


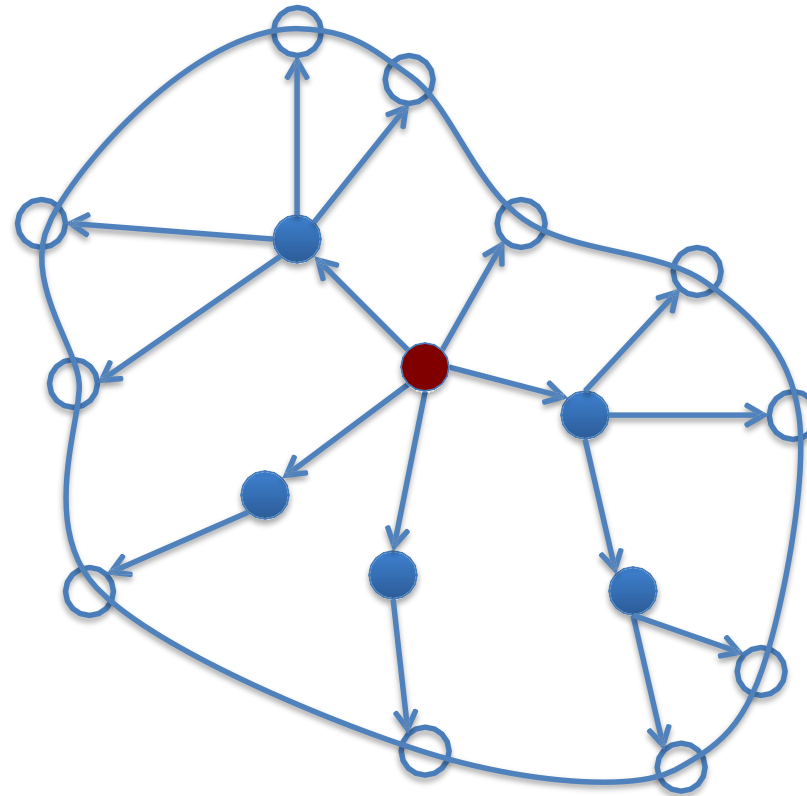










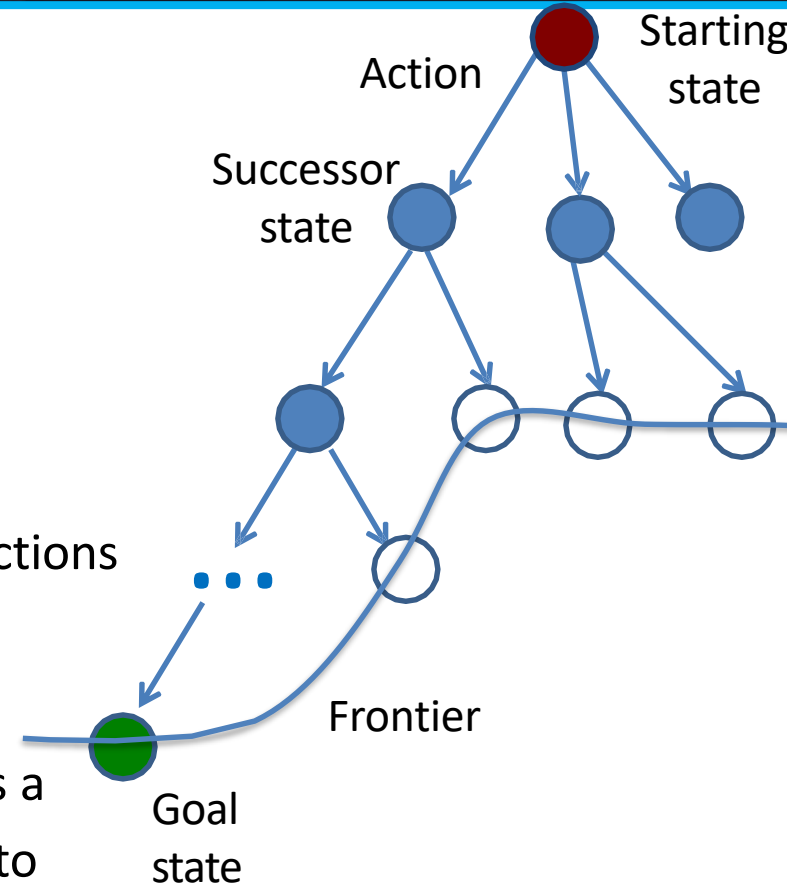


“What if” tree of sequences of actions and outcomes

- When we are searching, we are not acting in the world, merely “thinking” about the possibilities
- 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



Tree Search Algorithm Outline

Initialize the **frontier** using the **starting state**.

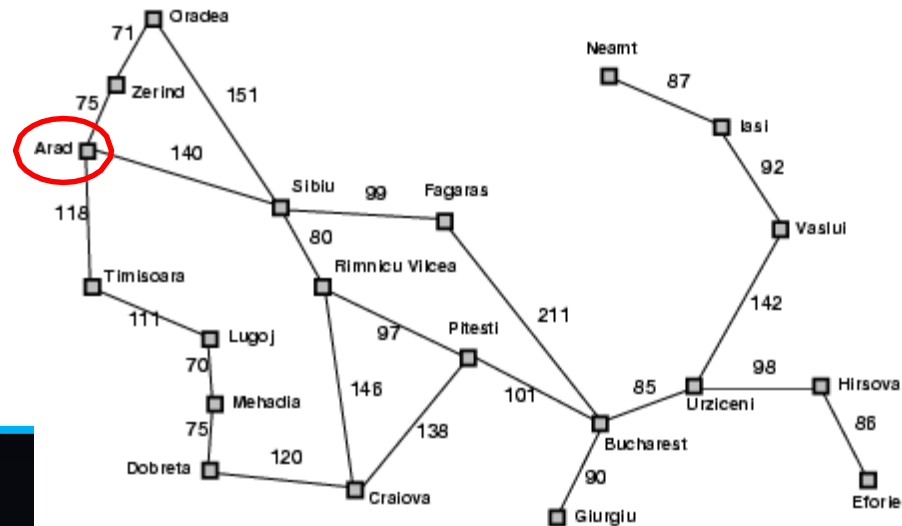
While the frontier is not empty:

- Choose a frontier node according to **search strategy**.
- If the node contains the **goal state**, return solution.
- Else **expand** the node and add its children to the frontier.

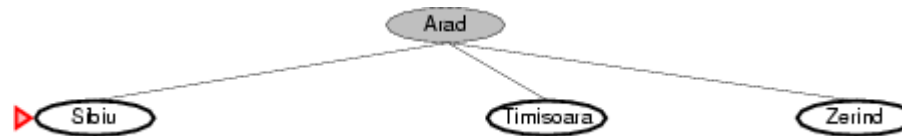
Tree Search Example

Arad

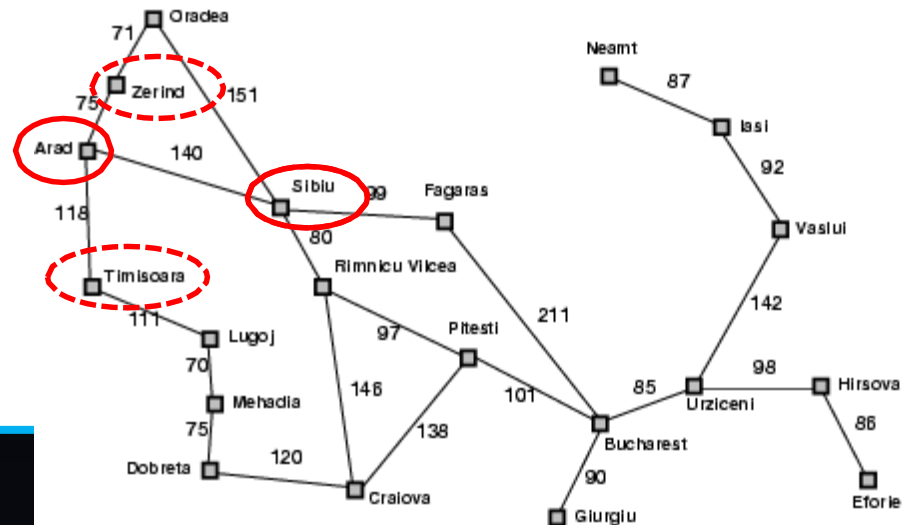
Start: Arad
Goal: Bucharest



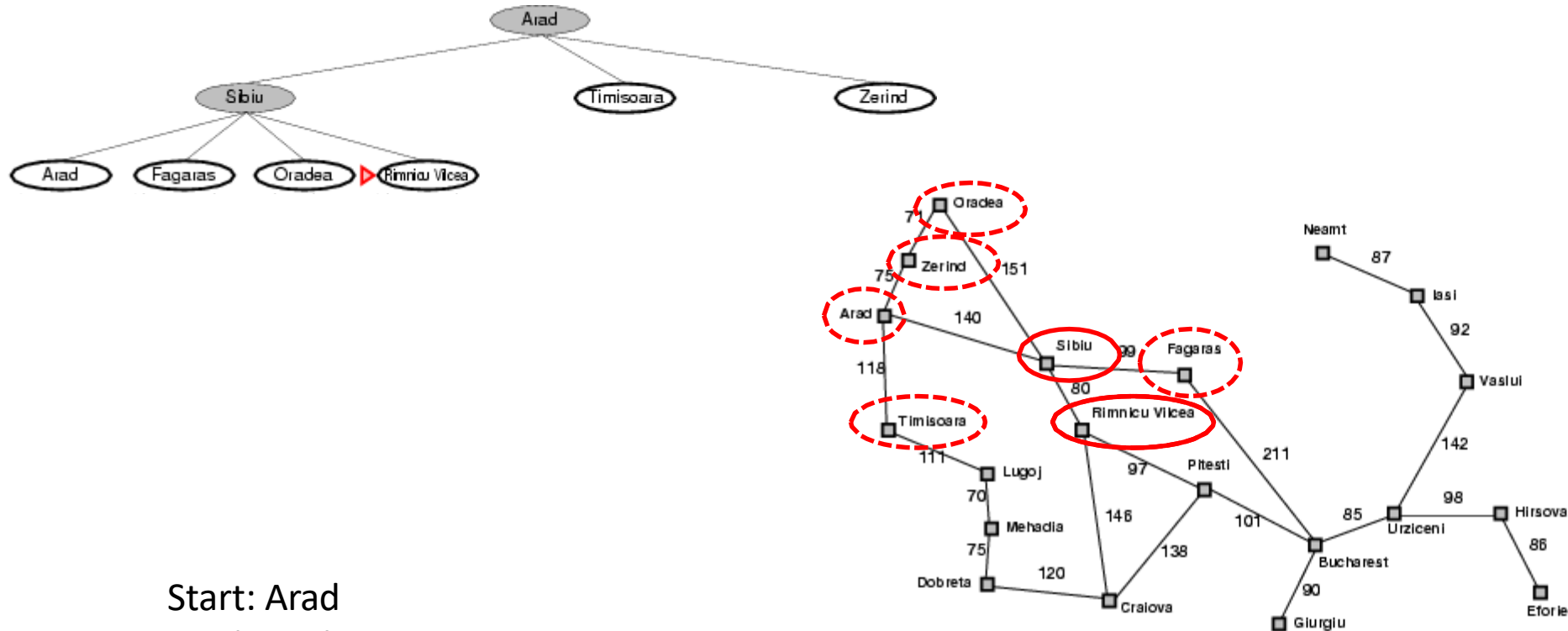
Tree Search Example



Start: Arad
Goal: Bucharest



Tree Search Example



Initialize the **frontier** using the **starting state**

While the frontier is not empty

Choose a frontier node according to **search strategy** and take it off the frontier.

If the node contains the **goal state**, return solution

Else **expand** the node and add its children to the frontier.

To handle repeated states:

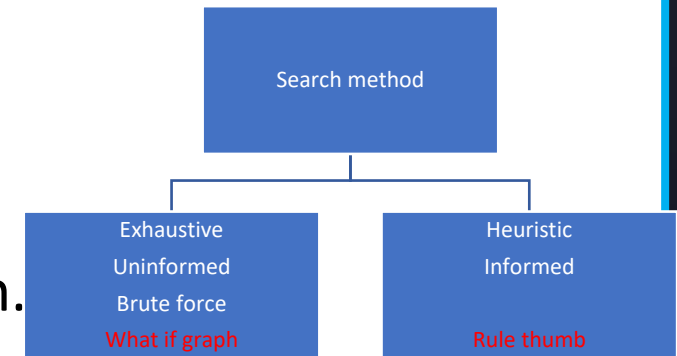
Every time you expand a node, add that state to the **Explored set (Close Or Deadend list)**; do not put explored states on the frontier again.

Every time you add a node to the frontier, check whether it already exists with a higher path cost, and if yes, replace the path with the new one with low cost.

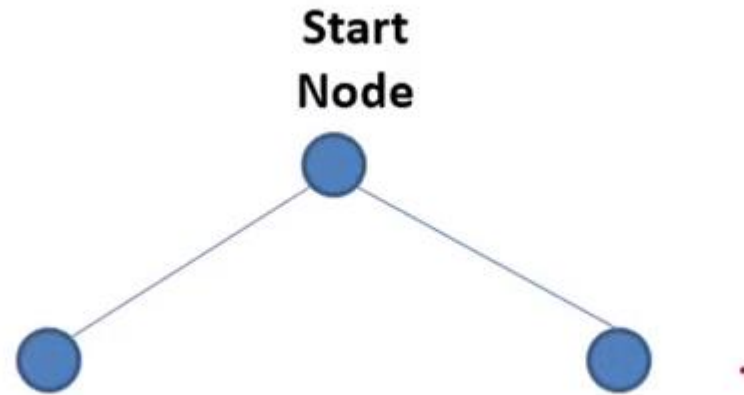
“Backtracking is a technique for systematically trying all paths through a state space”

It begins at the start state and pursues a path until:

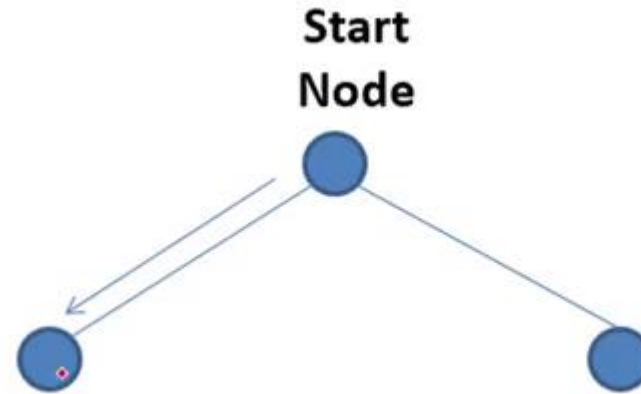
- Finding a goal, then quit and return the solution path.
- Finding a dead end, then backtrack to the most recent unexamined node and continue down one of its branches.



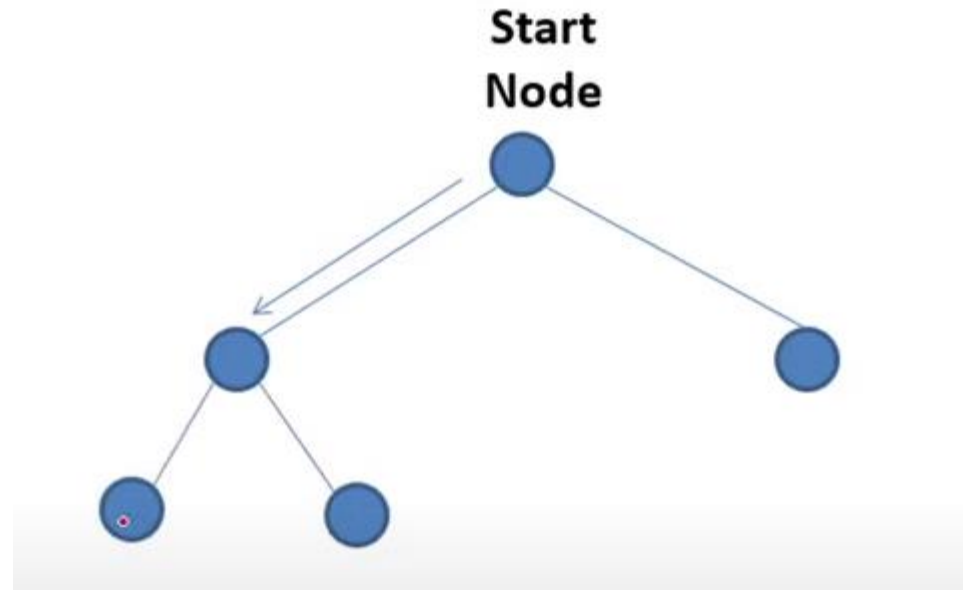
Backtracking Search



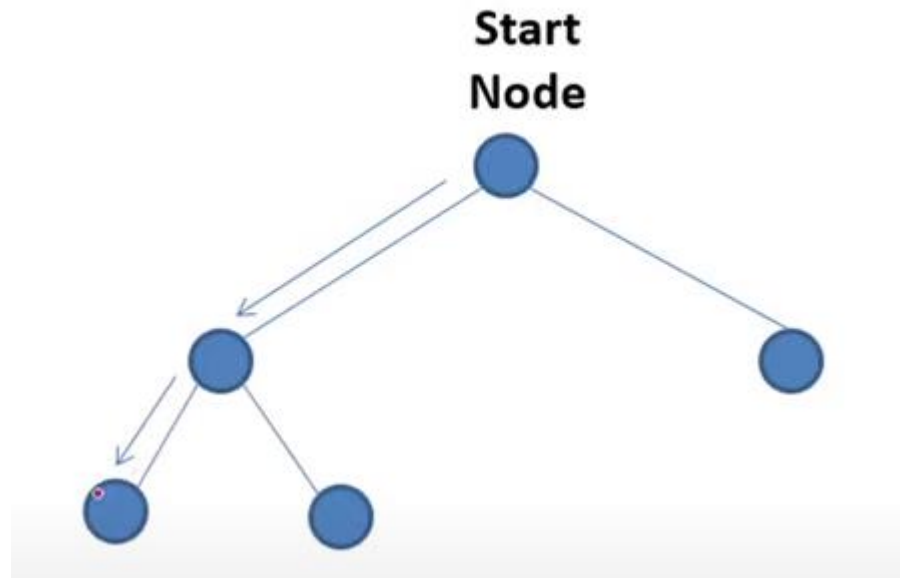
Backtracking Search



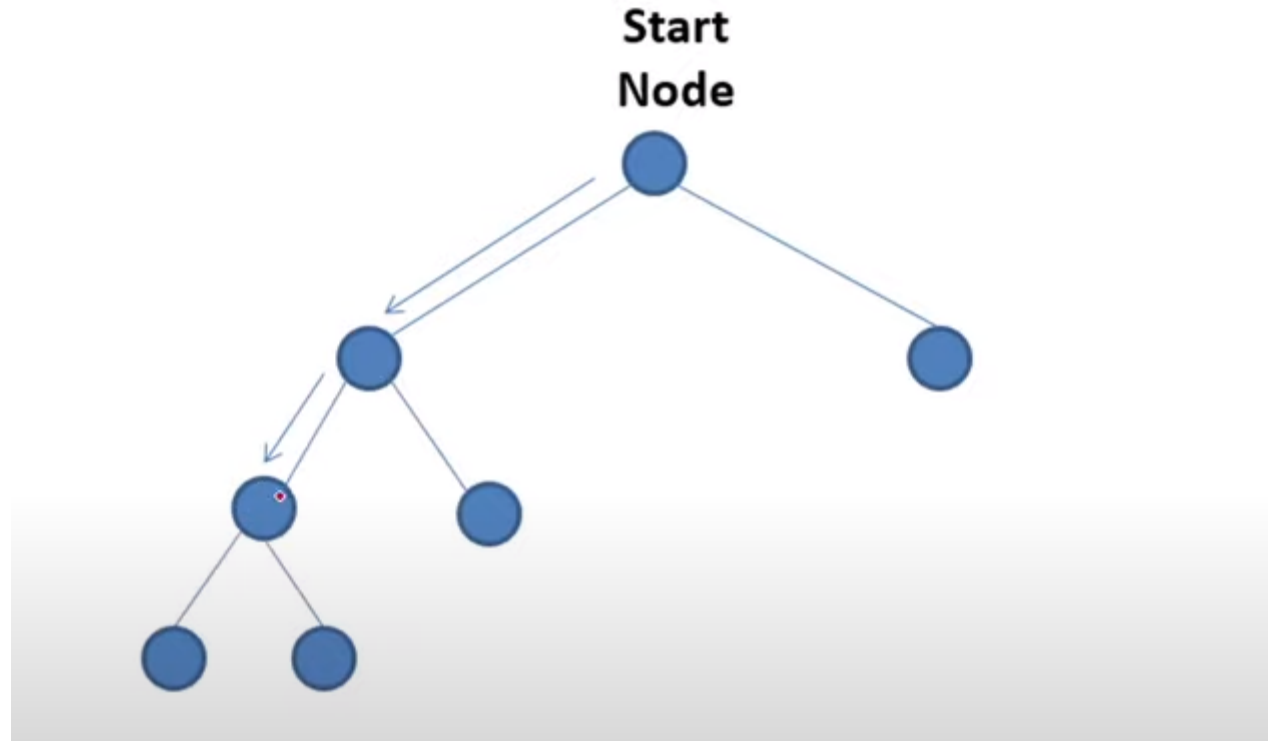
Backtracking Search



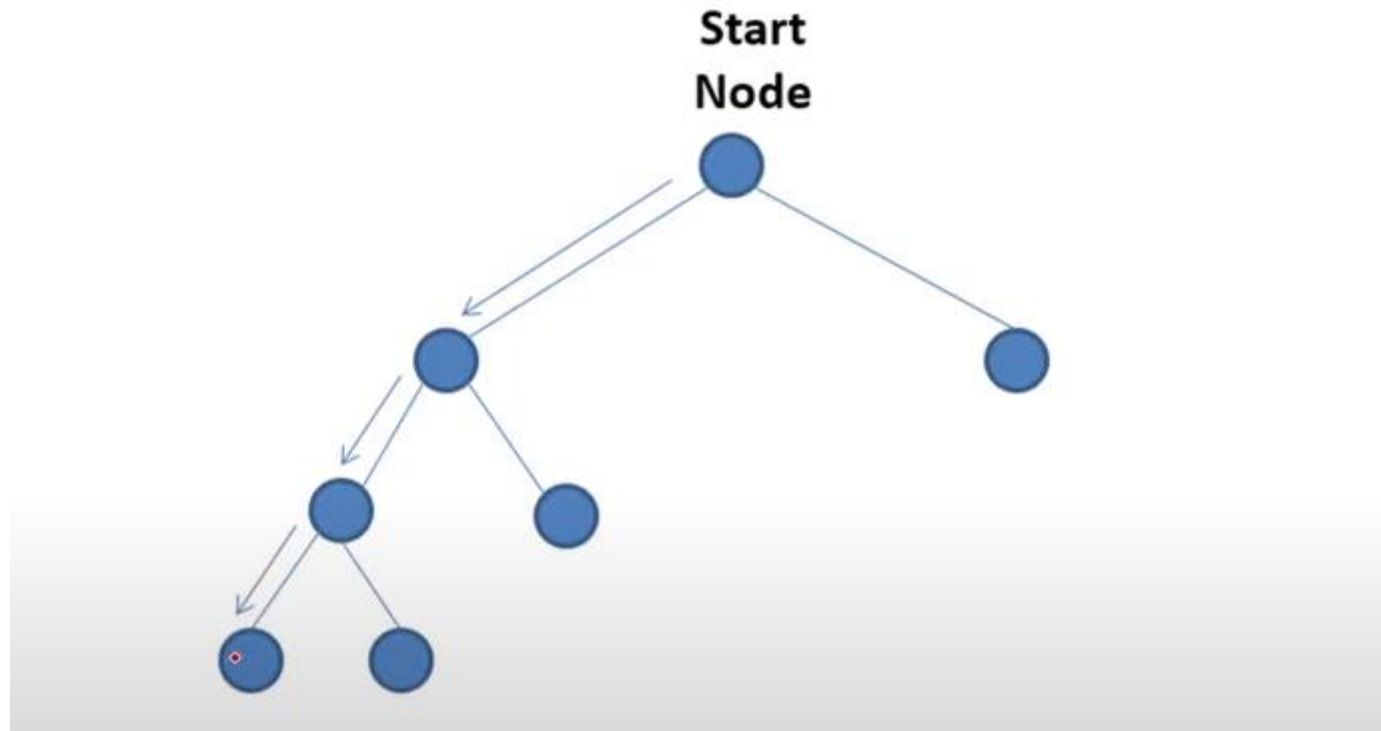
Backtracking Search



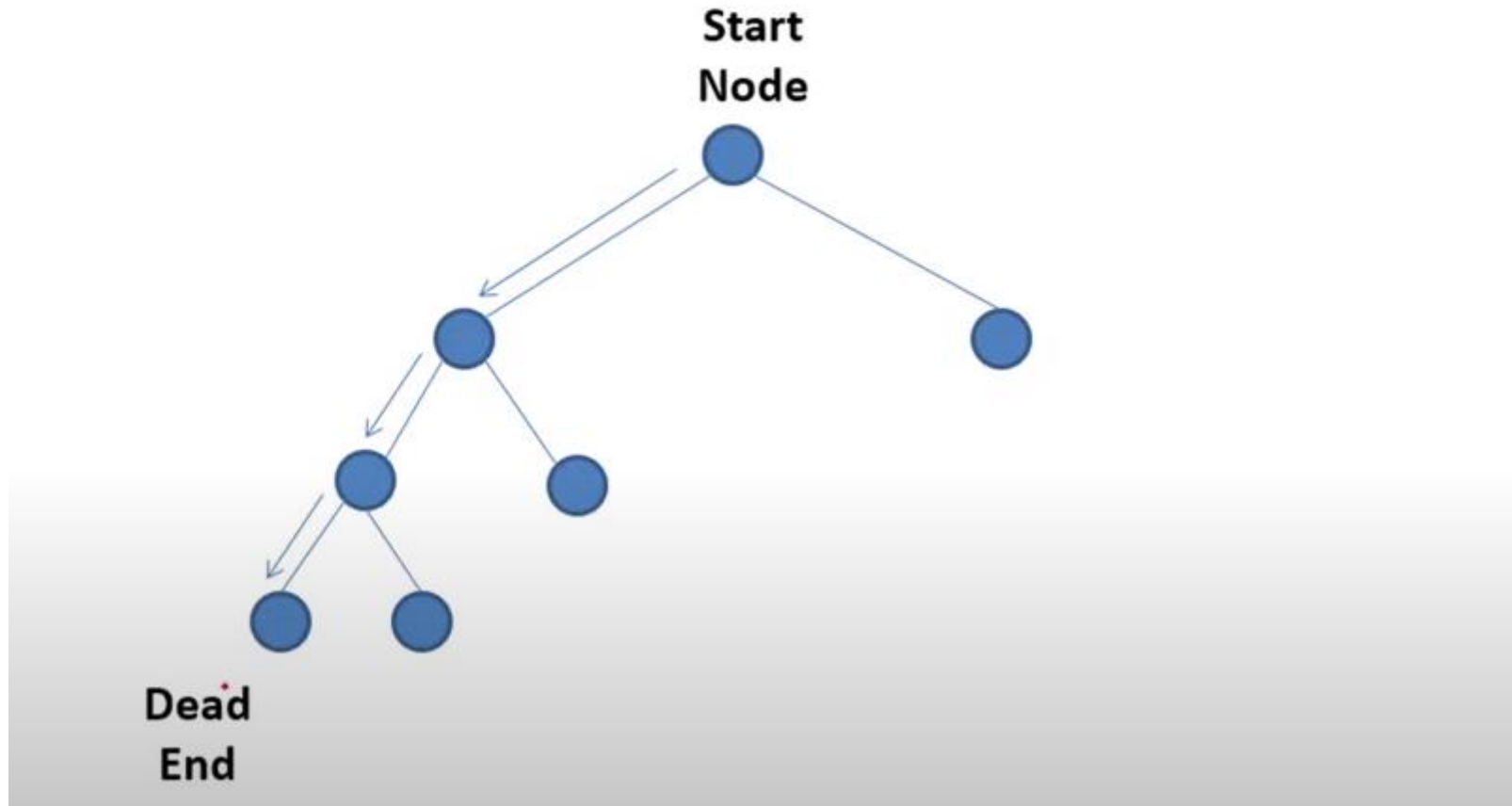
Backtracking Search



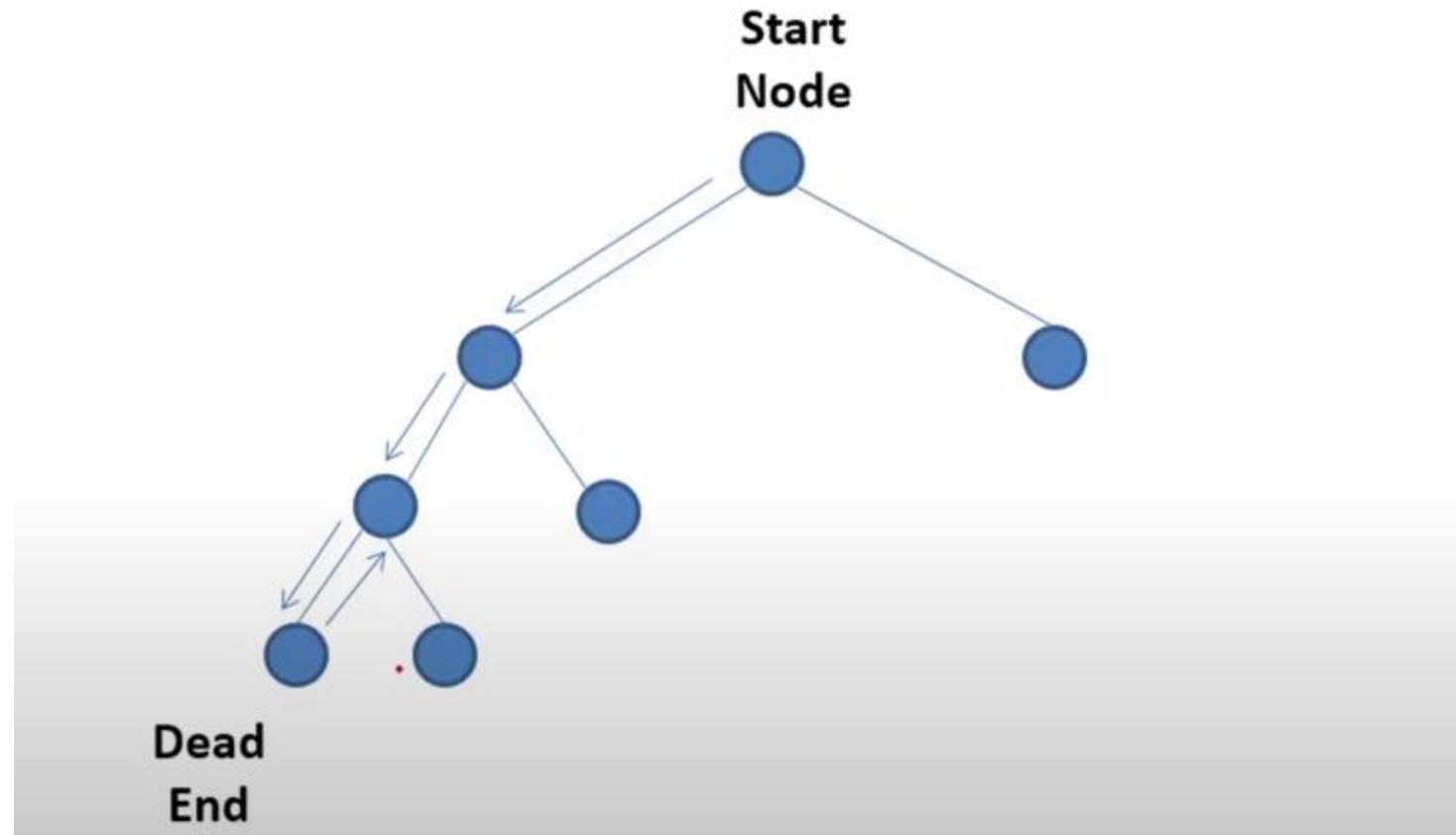
Backtracking Search



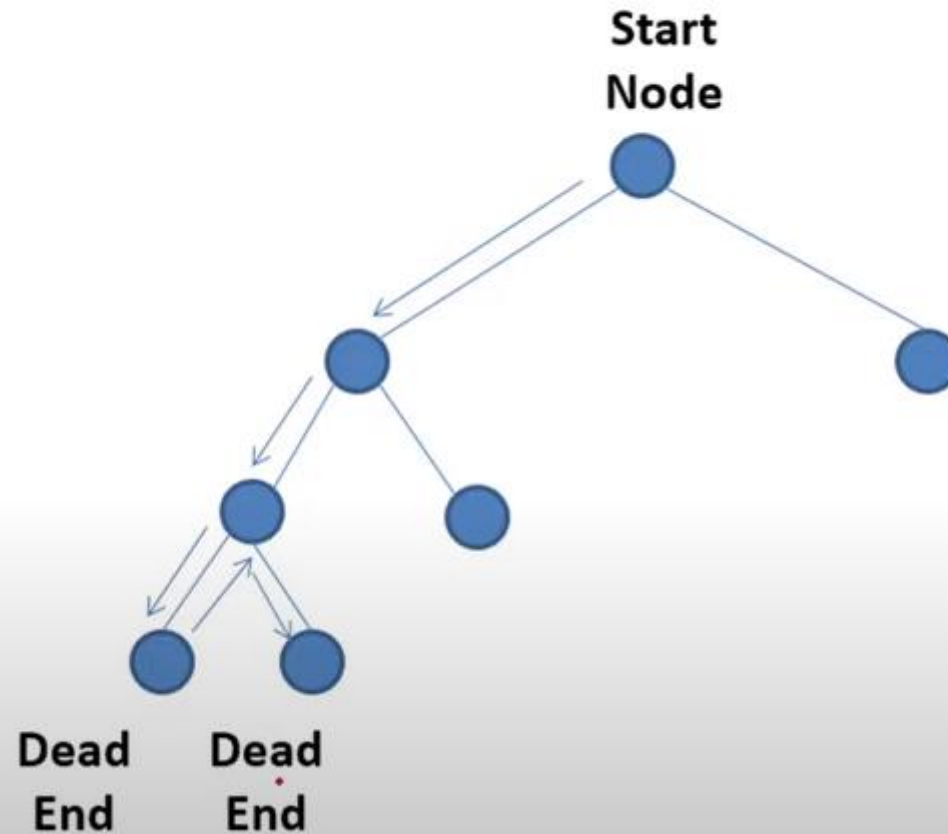
Backtracking Search



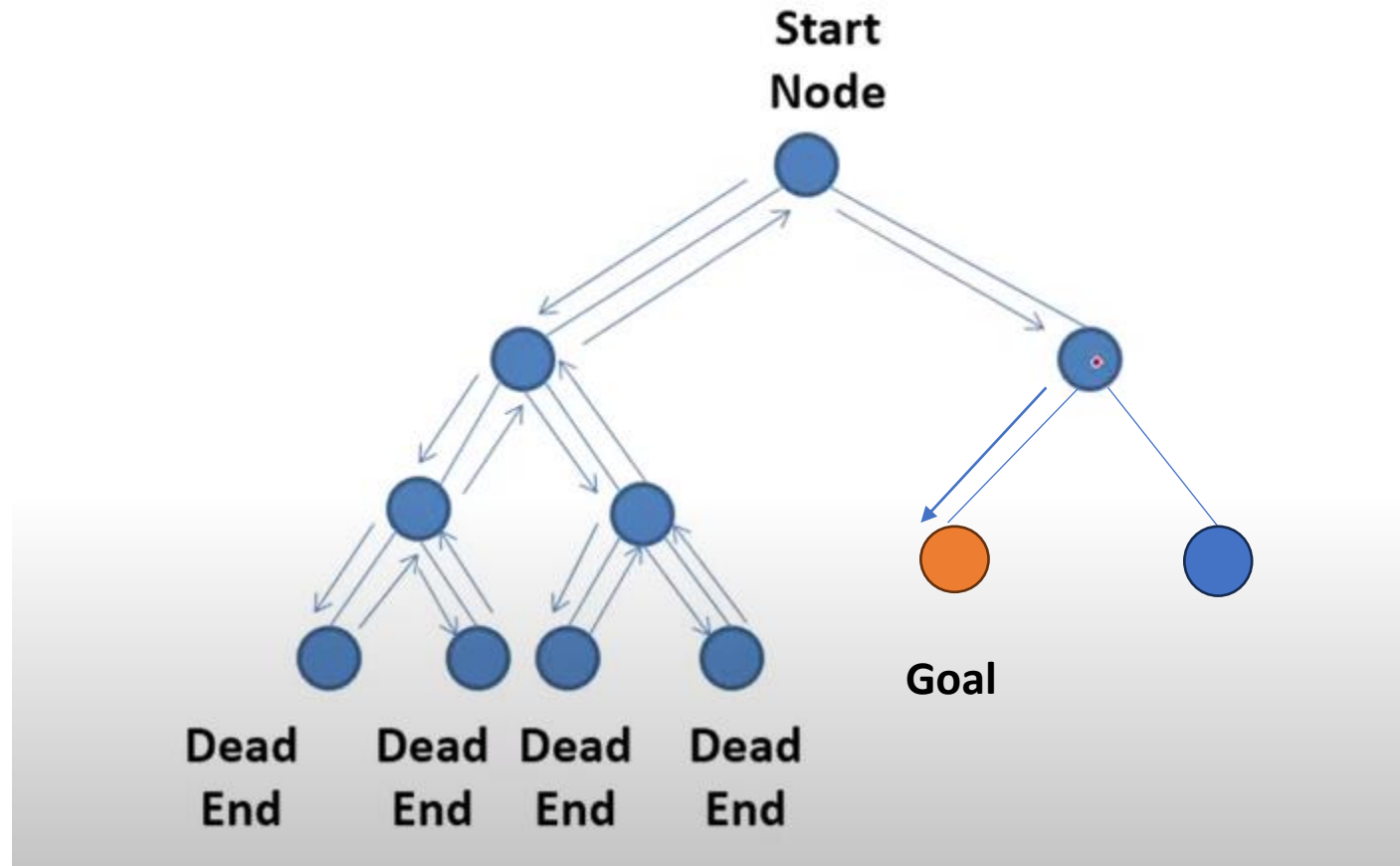
Backtracking Search



Backtracking Search



Backtracking Search



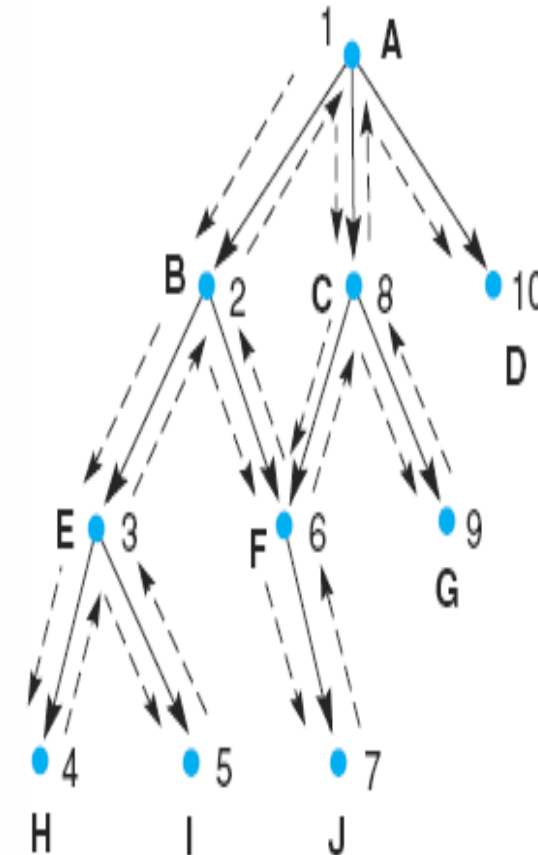
- State List (SL): lists the states in the current path being tried. If the goal is found, then it contains the **solution path**.
- New State List (NSL): contains nodes a waiting evaluation (frontier or open).
- Dead Ends (DE): lists states whose descendants have failed to contain a goal node.
- Current State (CS): a state currently under consideration pointer.

The Backtracking Algorithm

```
function backtrack;  
begin  
  SL := [Start]; NSL := [Start]; DE := [ ]; CS := Start;           % initialize:  
  while NSL ≠ [ ] do                                             % while there are states to be tried  
  begin  
    if CS = goal (or meets goal description)  
    then return SL;                                             % on success, return list of states in path.  
    if CS has no children (excluding nodes already on DE, SL, and NSL)  
    then begin  
      while SL is not empty and CS = the first element of SL do  
      begin  
        add CS to DE;                                           % record state as dead end  
        remove first element from SL;                          % backtrack  
        remove first element from NSL;  
        CS := first element of NSL;  
      end  
      add CS to SL;  
    end  
    else begin  
      place children of CS (except nodes already on DE, SL, or NSL) on NSL;  
      CS := first element of NSL;  
      add CS to SL  
    end  
  end;  
  return FAIL;  
end.
```

```
function backtrack;
```

```
begin
  SL := [Start]; NSL := [Start]; DE := []; CS := Start;           % initialize:
  while NSL ≠ [] do                                             % while there are states to be tried
  begin
    if CS = goal (or meets goal description)
    then return SL;                                             % on success, return list of states in path.
    if CS has no children (excluding nodes already on DE, SL, and NSL)
    then begin
      while SL is not empty and CS = the first element of SL do
      begin
        add CS to DE;                                           % record state as dead end
        remove first element from SL;                           %backtrack
        remove first element from NSL;
        CS := first element of NSL;
      end
      add CS to SL;
    end
    else begin
      place children of CS (except nodes already on DE, SL, or NSL) on NSL;
      CS := first element of NSL;
      add CS to SL
    end
  end
end;
return FAIL;
end.
```

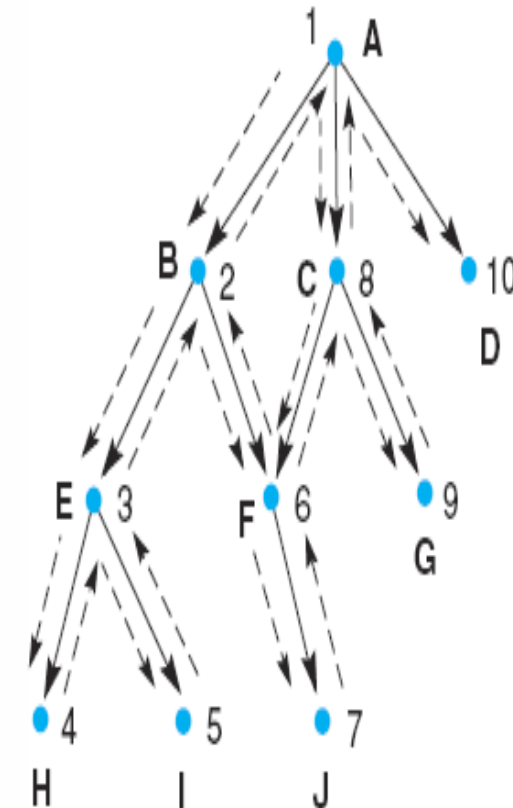


Lecture 4

```

function backtrack;
begin
  SL := [Start]; NSL := [Start]; DE := []; CS := Start;           % initialize:
  while NSL ≠ [] do                                             % while there are states to be tried
  begin
    if CS = goal (or meets goal description)
    then return SL;                                             % on success, return list of states in path.
    if CS has no children (excluding nodes already on DE, SL, and NSL)
    then begin
      while SL is not empty and CS = the first element of SL do
      begin
        add CS to DE;                                           % record state as dead end
        remove first element from SL;                           % backtrack
        remove first element from NSL;
        CS := first element of NSL;
      end
      add CS to SL;
    end
    else begin
      place children of CS (except nodes already on DE, SL, or NSL) on NSL;
      CS := first element of NSL;
      add CS to SL
    end
  end
  end;
  return FAIL;
end.
  
```

AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[]
1	B	[B A]	[B C D A]	[]

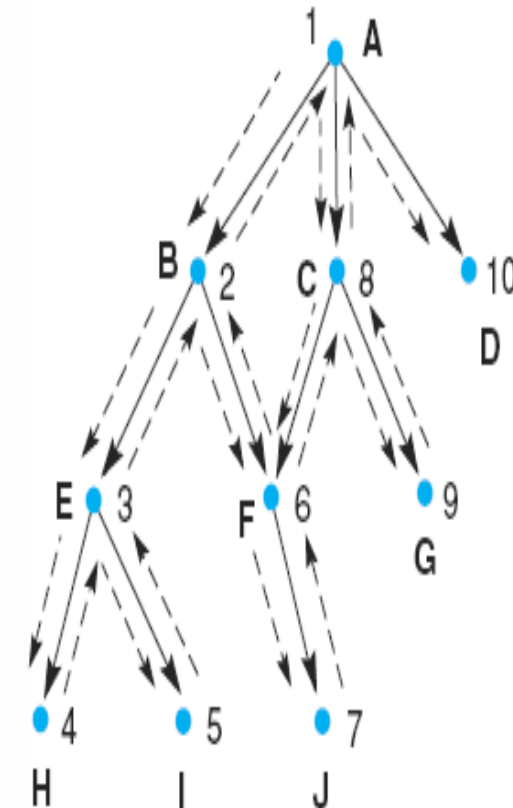


Lecture 4

```

function backtrack;
begin
  SL := [Start]; NSL := [Start]; DE := []; CS := Start;           % initialize:
  while NSL ≠ [] do                                             % while there are states to be tried
  begin
    if CS = goal (or meets goal description)
    then return SL;                                           % on success, return list of states in path.
    if CS has no children (excluding nodes already on DE, SL, and NSL)
    then begin
      while SL is not empty and CS = the first element of SL do
      begin
        add CS to DE;                                     % record state as dead end
        remove first element from SL;                     % backtrack
        remove first element from NSL;
        CS := first element of NSL;
      end
      add CS to SL;
    end
    else begin
      place children of CS (except nodes already on DE, SL, or NSL) on NSL;
      CS := first element of NSL;
      add CS to SL
    end
  end
  end;
  return FAIL;
end.
  
```

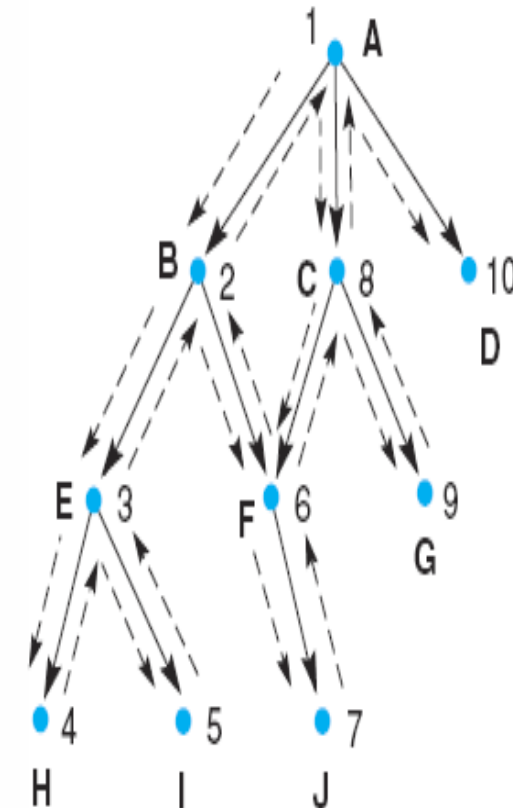
AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[]
1	B	[B A]	[B C D A]	[]
2	E	[E B A]	[E F B C D A]	[]



Lecture 4

```

function backtrack;
begin
  SL := [Start]; NSL := [Start]; DE := []; CS := Start;           % initialize:
  while NSL ≠ [] do                                             % while there are states to be tried
  begin
    if CS = goal (or meets goal description)
    then return SL;                                             % on success, return list of states in path.
    if CS has no children (excluding nodes already on DE, SL, and NSL)
    then begin
      while SL is not empty and CS = the first element of SL do
      begin
        add CS to DE;                                           % record state as dead end
        remove first element from SL;                           % backtrack
        remove first element from NSL;
        CS := first element of NSL;
      end
      add CS to SL;
    end
    else begin
      place children of CS (except nodes already on DE, SL, or NSL) on NSL;
      CS := first element of NSL;
      add CS to SL
    end
  end
  end;
  return FAIL;
end.
  
```



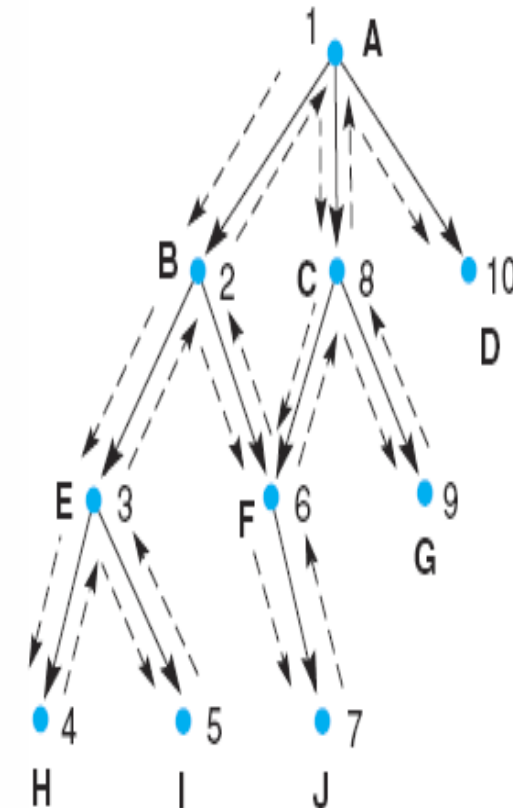
AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[]
1	B	[B A]	[B C D A]	[]
2	E	[E B A]	[E F B C D A]	[]
3	H	[H E B A]	[H I E F B C D A]	[]

Lecture 4

```

function backtrack;
begin
  SL := [Start]; NSL := [Start]; DE := []; CS := Start;           % initialize:
  while NSL ≠ [] do                                             % while there are states to be tried
  begin
    if CS = goal (or meets goal description)
    then return SL;                                           % on success, return list of states in path.
    if CS has no children (excluding nodes already on DE, SL, and NSL)
    then begin
      while SL is not empty and CS = the first element of SL do
      begin
        add CS to DE;                                     % record state as dead end
        remove first element from SL;                     % backtrack
        remove first element from NSL;
        CS := first element of NSL;
      end
      add CS to SL;
    end
    else begin
      place children of CS (except nodes already on DE, SL, or NSL) on NSL;
      CS := first element of NSL;
      add CS to SL
    end
  end
  end;
  return FAIL;
end.
  
```

AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[]
1	B	[B A]	[B C D A]	[]
2	E	[E B A]	[E F B C D A]	[]
3	H	[H E B A]	[H I E F B C D A]	[]
4	I	[I E B A]	[I E F B C D A]	[H]



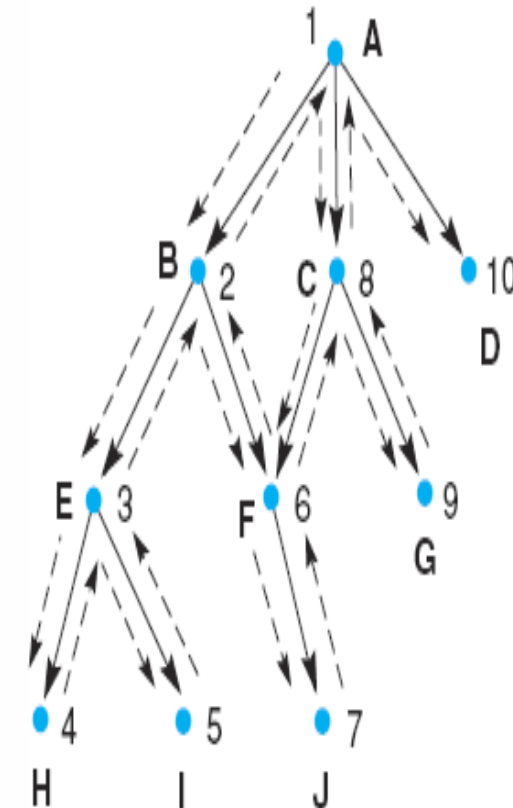
Lecture 4

```

function backtrack;
begin
  SL := [Start]; NSL := [Start]; DE := []; CS := Start;           % initialize:
  while NSL ≠ [] do                                             % while there are states to be tried
  begin
    if CS = goal (or meets goal description)
    then return SL;                                           % on success, return list of states in path.
    if CS has no children (excluding nodes already on DE, SL, and NSL)
    then begin
      while SL is not empty and CS = the first element of SL do
      begin
        add CS to DE;                                     % record state as dead end
        remove first element from SL;                     % backtrack
        remove first element from NSL;
        CS := first element of NSL;
      end
      add CS to SL;
    end
    else begin
      place children of CS (except nodes already on DE, SL, or NSL) on NSL;
      CS := first element of NSL;
      add CS to SL
    end
  end
  end;
  return FAIL;
end.

```

AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[]
1	B	[B A]	[B C D A]	[]
2	E	[E B A]	[E F B C D A]	[]
3	H	[H E B A]	[H I E F B C D A]	[]
4	I	[I E B A]	[I E F B C D A]	[H]
5	F	[F B A]	[F B C D A]	[E I H]



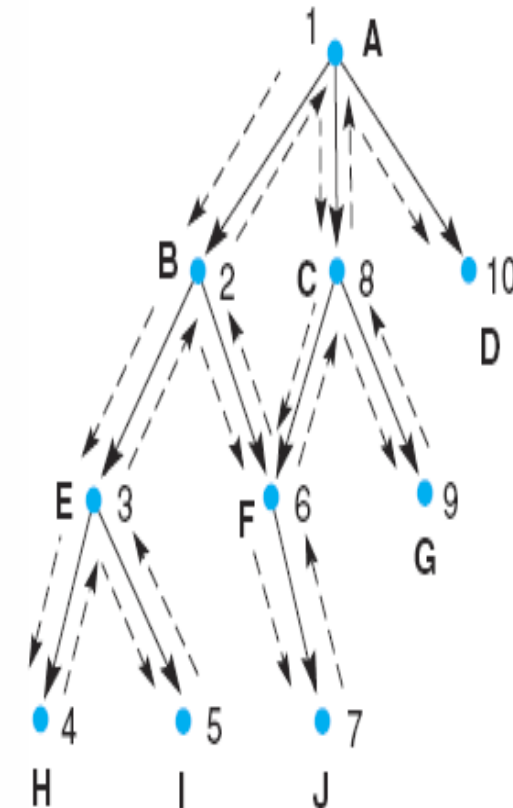
Lecture 4

```

function backtrack;
begin
  SL := [Start]; NSL := [Start]; DE := []; CS := Start;           % initialize:
  while NSL ≠ [] do                                             % while there are states to be tried
  begin
    if CS = goal (or meets goal description)
    then return SL;                                             % on success, return list of states in path.
    if CS has no children (excluding nodes already on DE, SL, and NSL)
    then begin
      while SL is not empty and CS = the first element of SL do
      begin
        add CS to DE;                                           % record state as dead end
        remove first element from SL;                           % backtrack
        remove first element from NSL;
        CS := first element of NSL;
      end
      add CS to SL;
    end
    else begin
      place children of CS (except nodes already on DE, SL, or NSL) on NSL;
      CS := first element of NSL;
      add CS to SL
    end
  end
  end;
  return FAIL;
end.

```

AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[]
1	B	[B A]	[B C D A]	[]
2	E	[E B A]	[E F B C D A]	[]
3	H	[H E B A]	[H I E F B C D A]	[]
4	I	[I E B A]	[I E F B C D A]	[H]
5	F	[F B A]	[F B C D A]	[E I H]
6	J	[J F B A]	[J F B C D A]	[E I H]

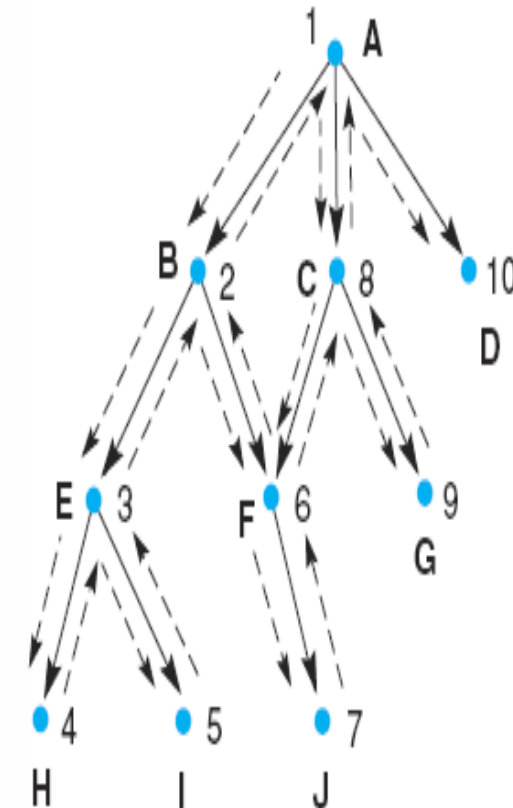


Lecture 4

```

function backtrack;
begin
  SL := [Start]; NSL := [Start]; DE := []; CS := Start;           % initialize:
  while NSL ≠ [] do                                             % while there are states to be tried
  begin
    if CS = goal (or meets goal description)
    then return SL;                                             % on success, return list of states in path.
    if CS has no children (excluding nodes already on DE, SL, and NSL)
    then begin
      while SL is not empty and CS = the first element of SL do
      begin
        add CS to DE;                                           % record state as dead end
        remove first element from SL;                           % backtrack
        remove first element from NSL;
        CS := first element of NSL;
      end
      add CS to SL;
    end
    else begin
      place children of CS (except nodes already on DE, SL, or NSL) on NSL;
      CS := first element of NSL;
      add CS to SL
    end
  end
  end;
  return FAIL;
end.
  
```

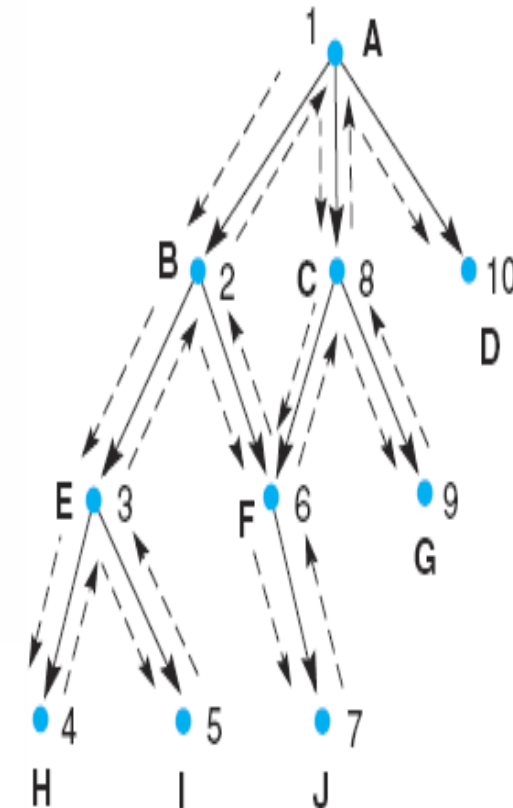
AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[]
1	B	[BA]	[BCDA]	[]
2	E	[EBA]	[EFBCDA]	[]
3	H	[HEBA]	[HIEFBCDA]	[]
4	I	[IEBA]	[IEFBCDA]	[H]
5	F	[FBA]	[FBCDA]	[EIH]
6	J	[JFBA]	[JFBCDA]	[EIH]
7	C	[CA]	[CDA]	[BFJEIH]



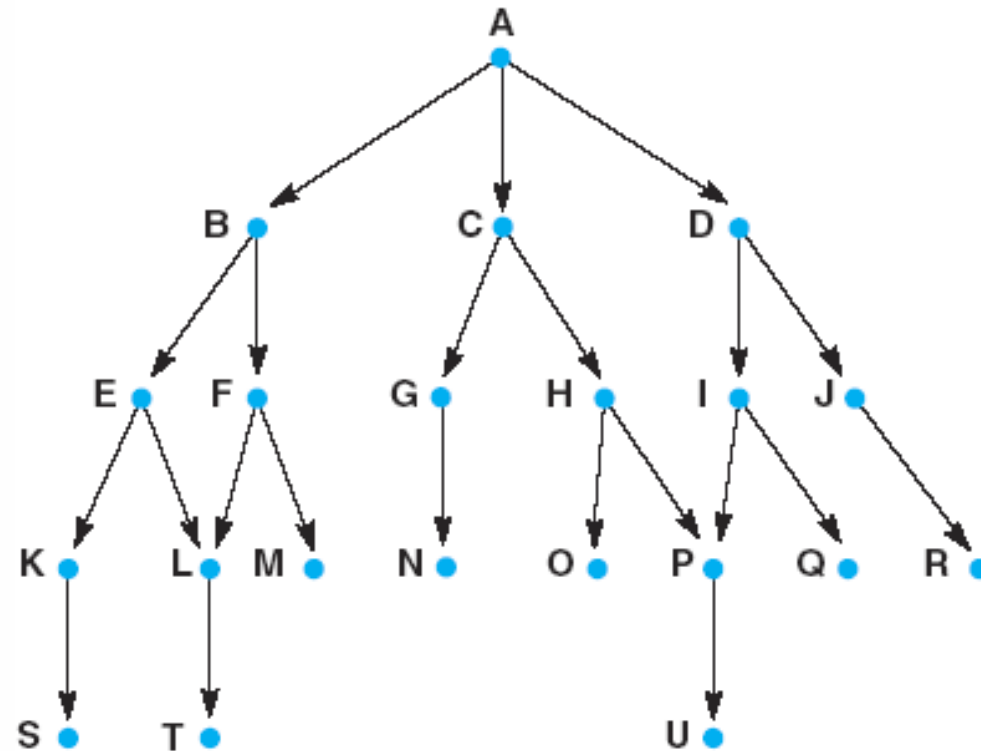
Lecture 4

```
function backtrack;
begin
  SL := [Start]; NSL := [Start]; DE := []; CS := Start;           % initialize:
  while NSL ≠ [] do                                             % while there are states to be tried
  begin
    if CS = goal (or meets goal description)
    then return SL;                                             % on success, return list of states in path.
    if CS has no children (excluding nodes already on DE, SL, and NSL)
    then begin
      while SL is not empty and CS = the first element of SL do
      begin
        add CS to DE;                                           % record state as dead end
        remove first element from SL;                           % backtrack
        remove first element from NSL;
        CS := first element of NSL;
      end
      add CS to SL;
    end
    else begin
      place children of CS (except nodes already on DE, SL, or NSL) on NSL;
      CS := first element of NSL;
      add CS to SL
    end
  end
  end;
  return FAIL;
end.
```

AFTER ITERATION	CS	SL	NSL	DE
0	A	[A]	[A]	[]
1	B	[B A]	[B C D A]	[]
2	E	[E B A]	[E F B C D A]	[]
3	H	[H E B A]	[H I E F B C D A]	[]
4	I	[I E B A]	[I E F B C D A]	[H]
5	F	[F B A]	[F B C D A]	[E I H]
6	J	[J F B A]	[J F B C D A]	[E I H]
7	C	[C A]	[C D A]	[B F J E I H]
8	G	[G C A]	[G C D A]	[B F J E I H]



Example 2



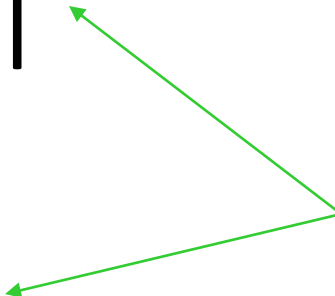
Blind (or **un-informed** or **exhaustive** or **brute-force**) strategies do not exploit any of the information contained in a state.

Heuristic (or **informed**) strategies exploits such information to assess that one node is “more promising” than another.

Breadth-first
Bidirectional

Depth-first
Depth-limited
Iterative deepening

Step cost = 1

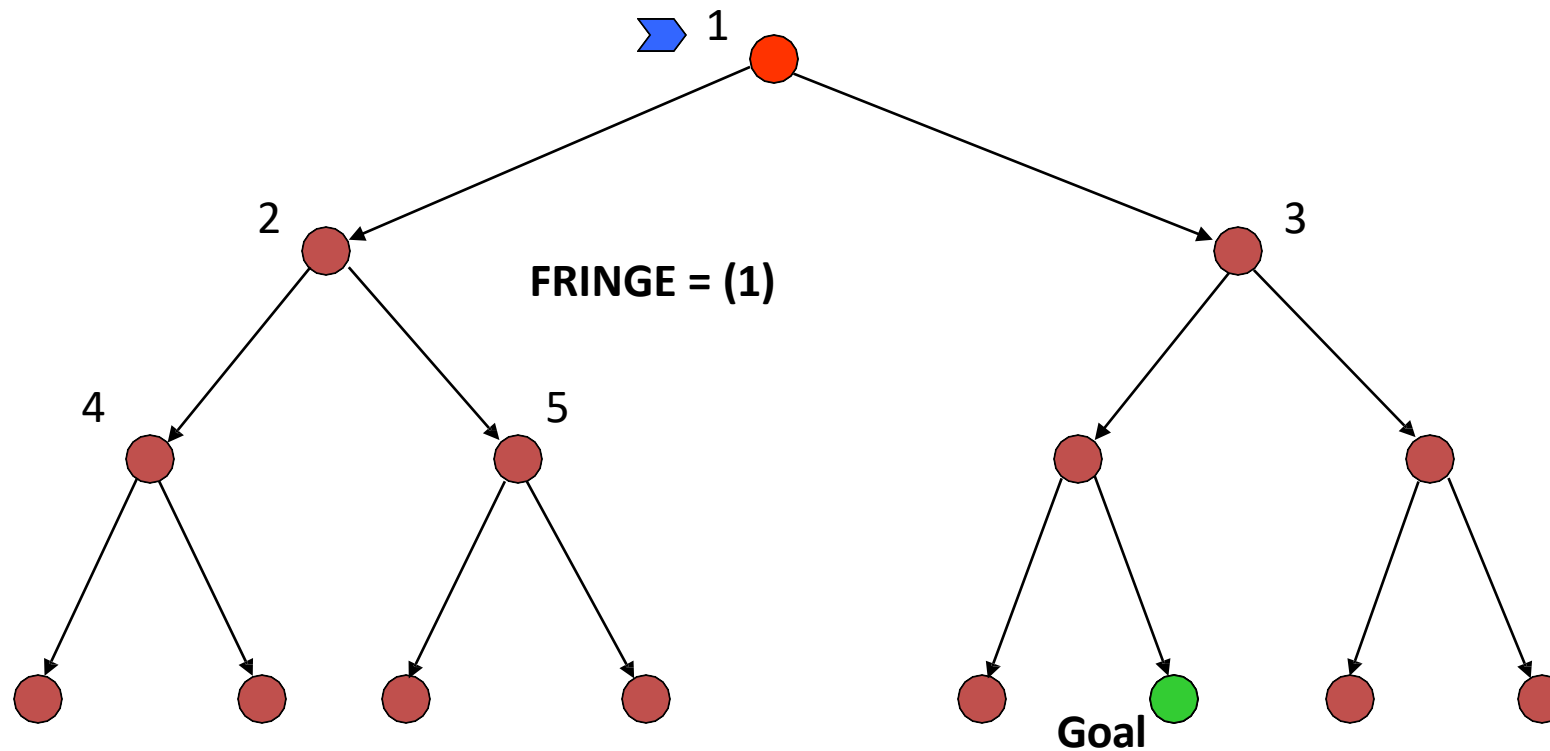


Uniform-Cost

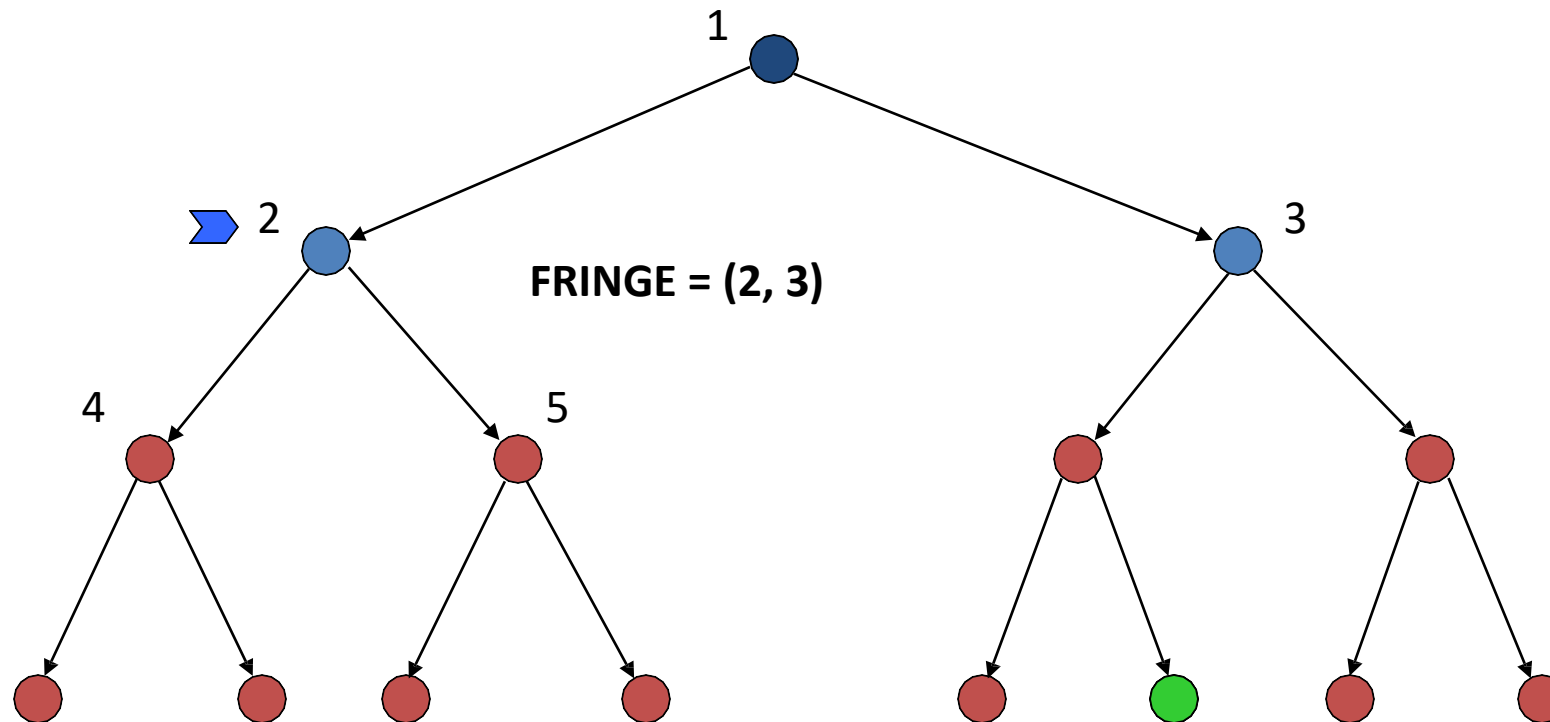
Step cost = $c(\text{action}) \geq \epsilon > 0$



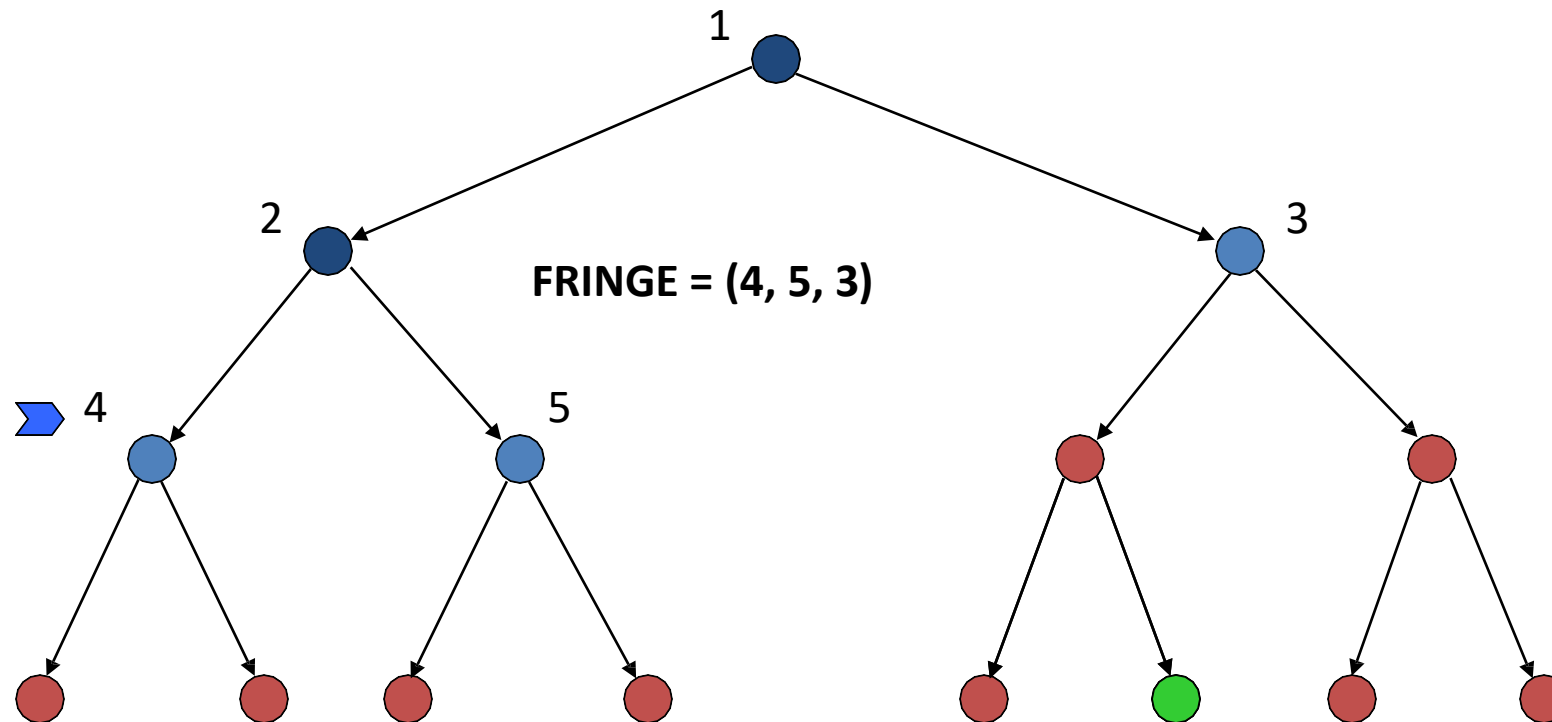
New nodes are inserted **at the front** of FRINGE



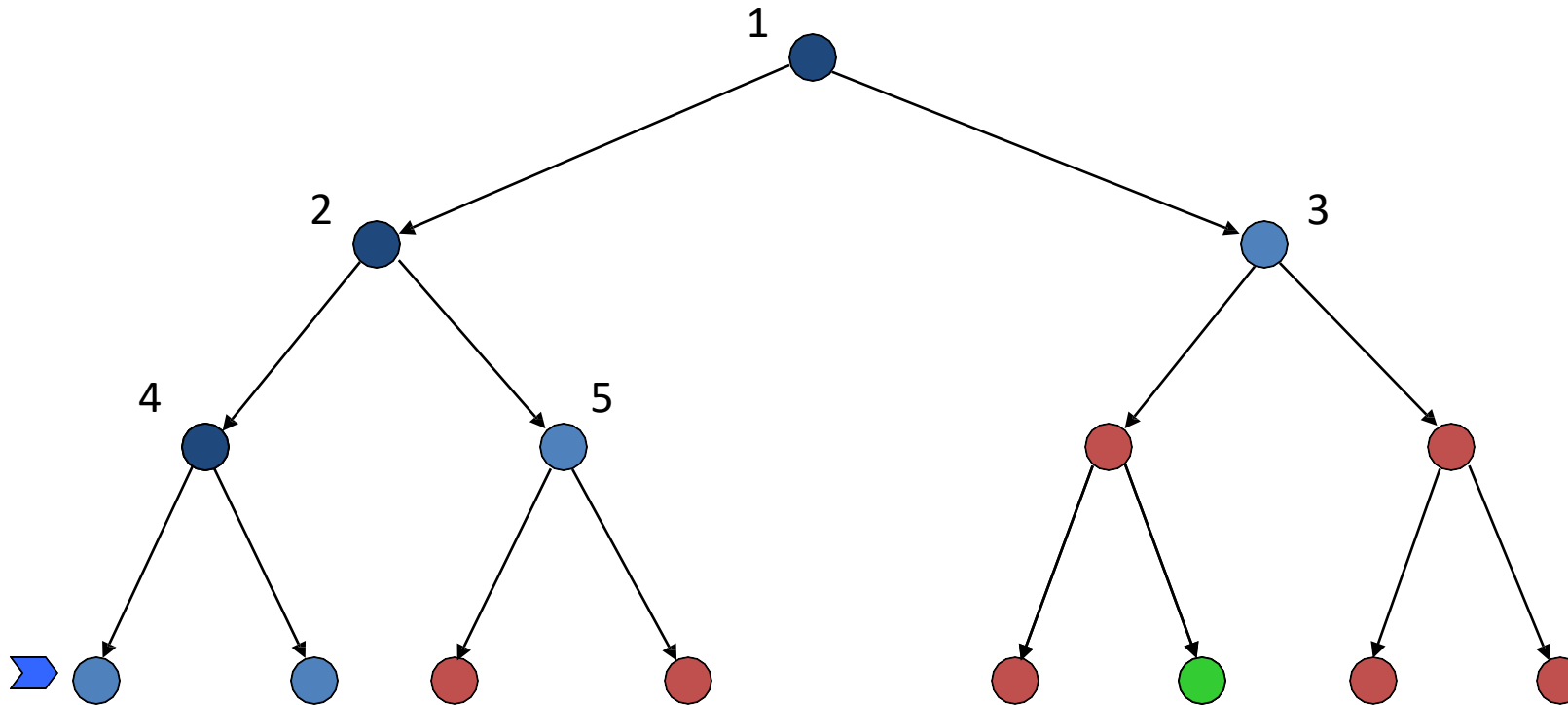
New nodes are inserted **at the front** of FRINGE



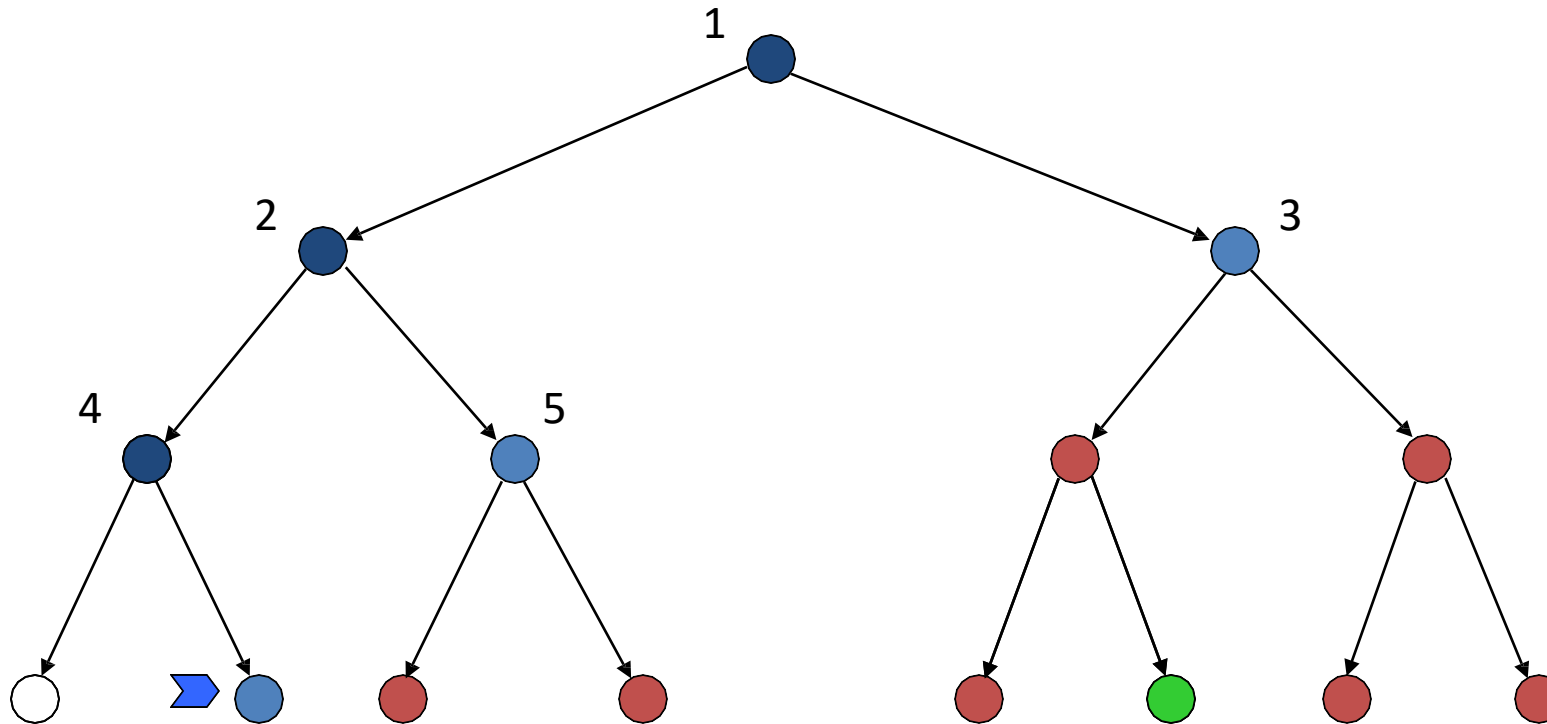
New nodes are inserted **at the front** of FRINGE



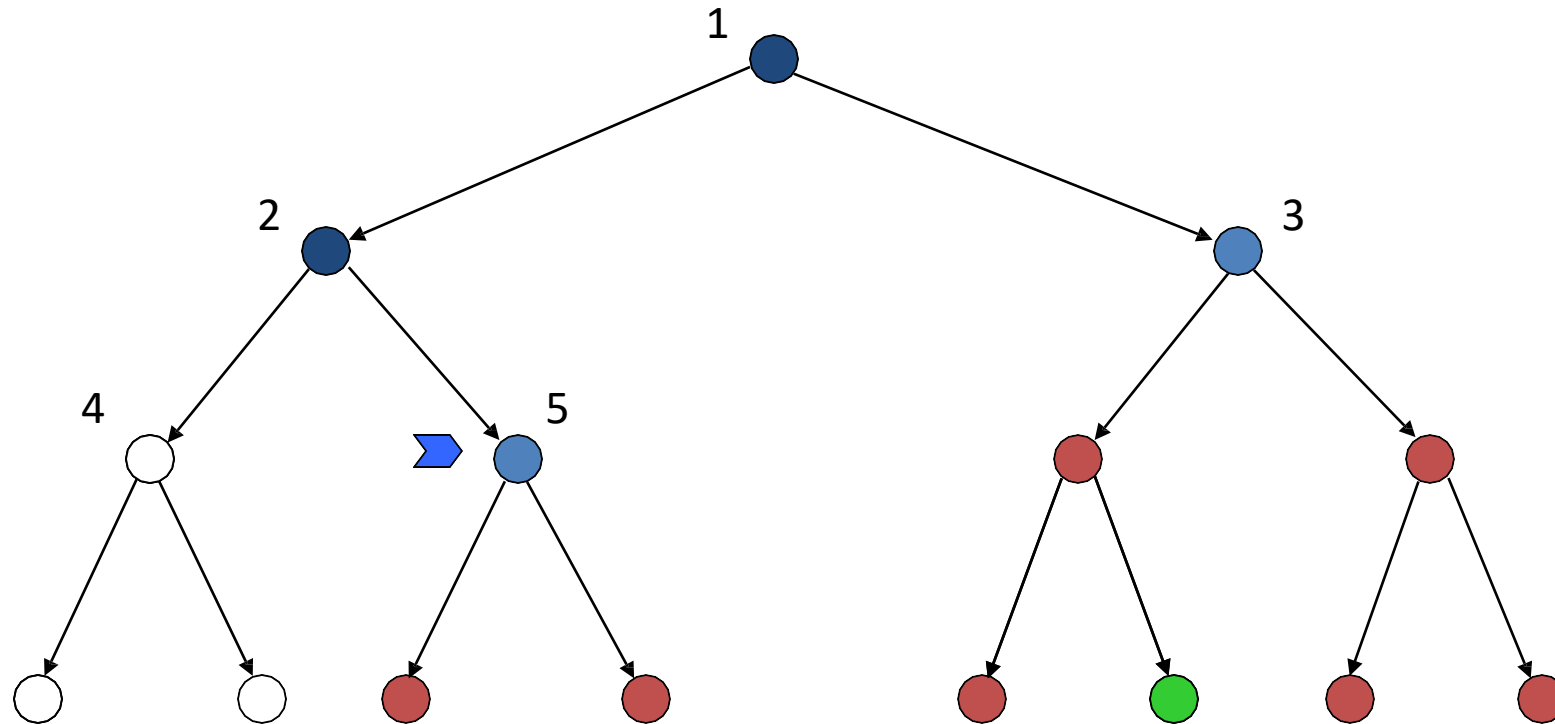
New nodes are inserted **at the front** of FRINGE



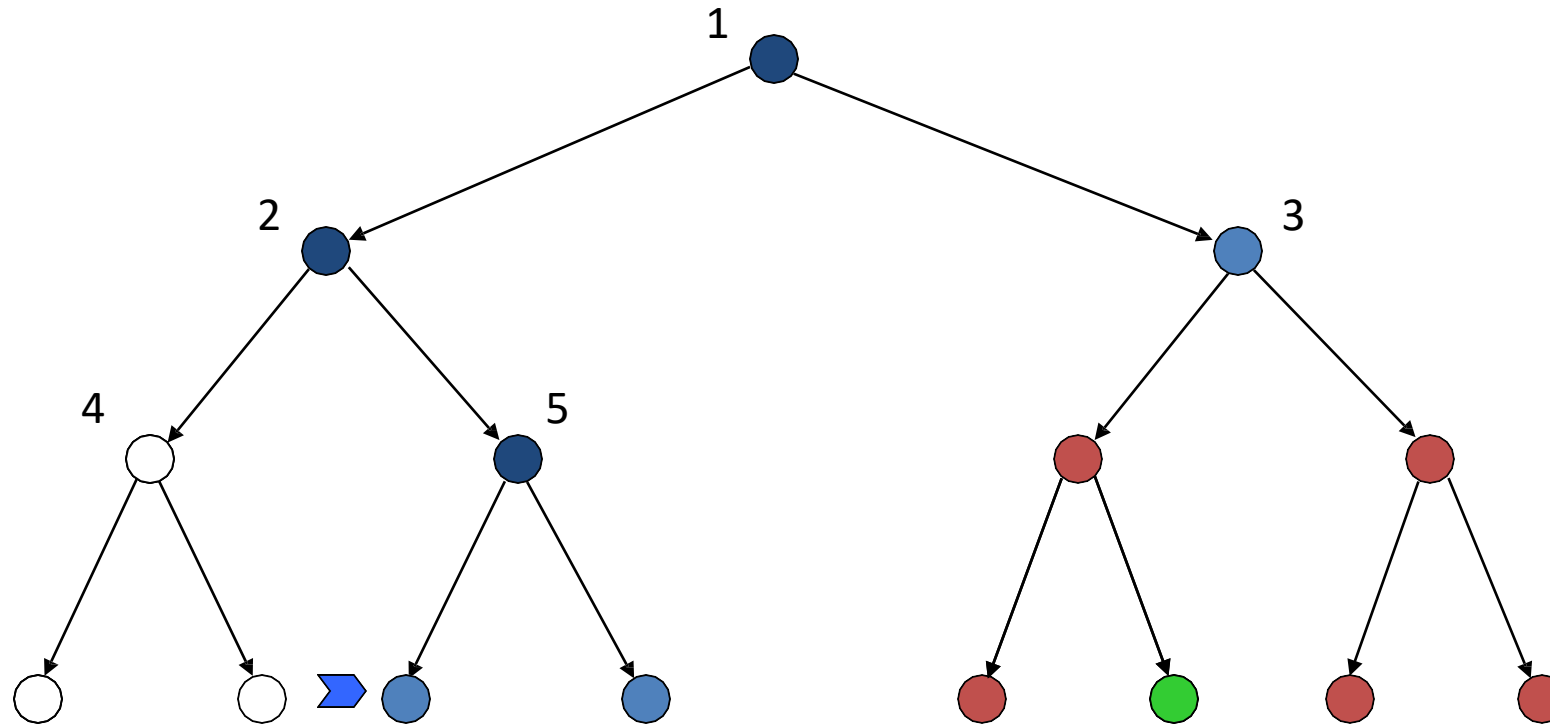
New nodes are inserted **at the front** of FRINGE



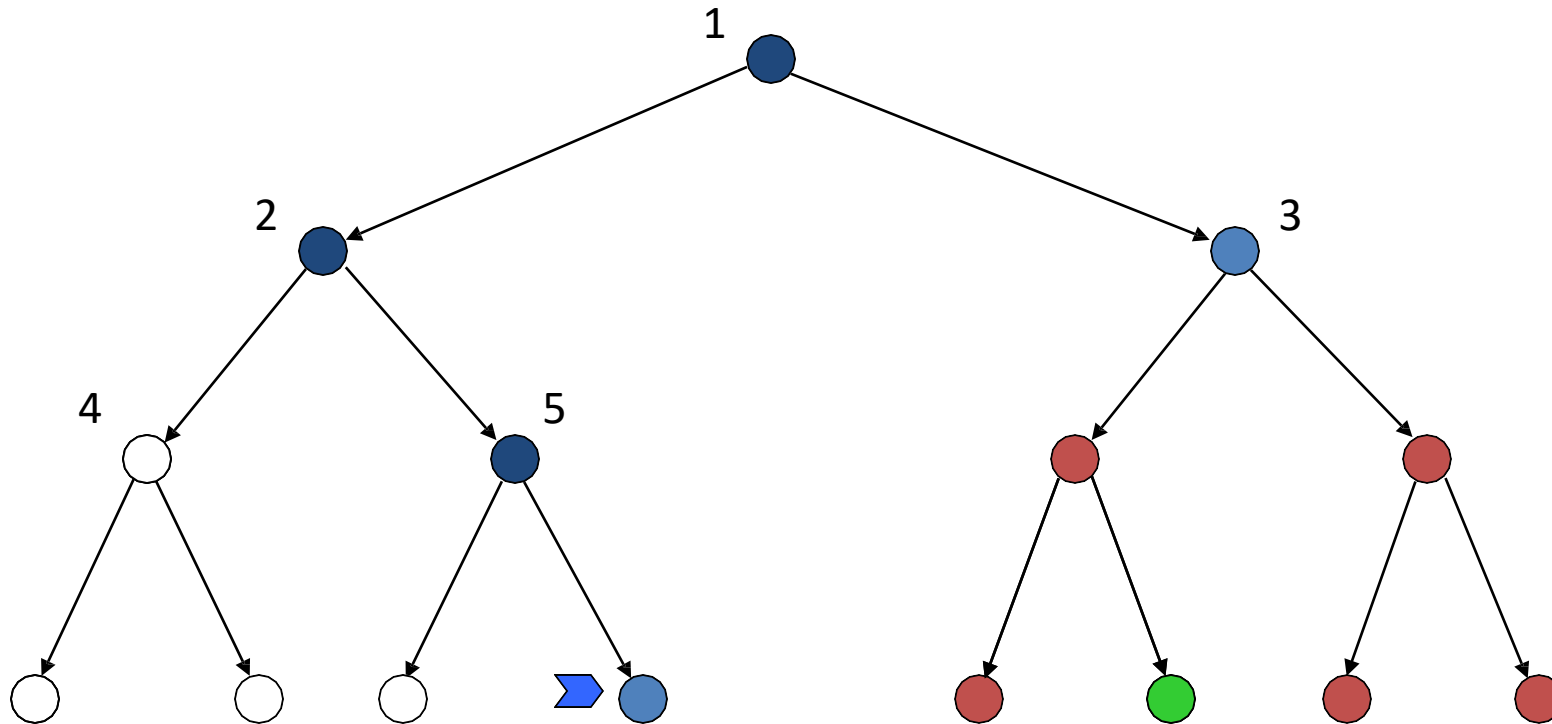
New nodes are inserted **at the front** of FRINGE



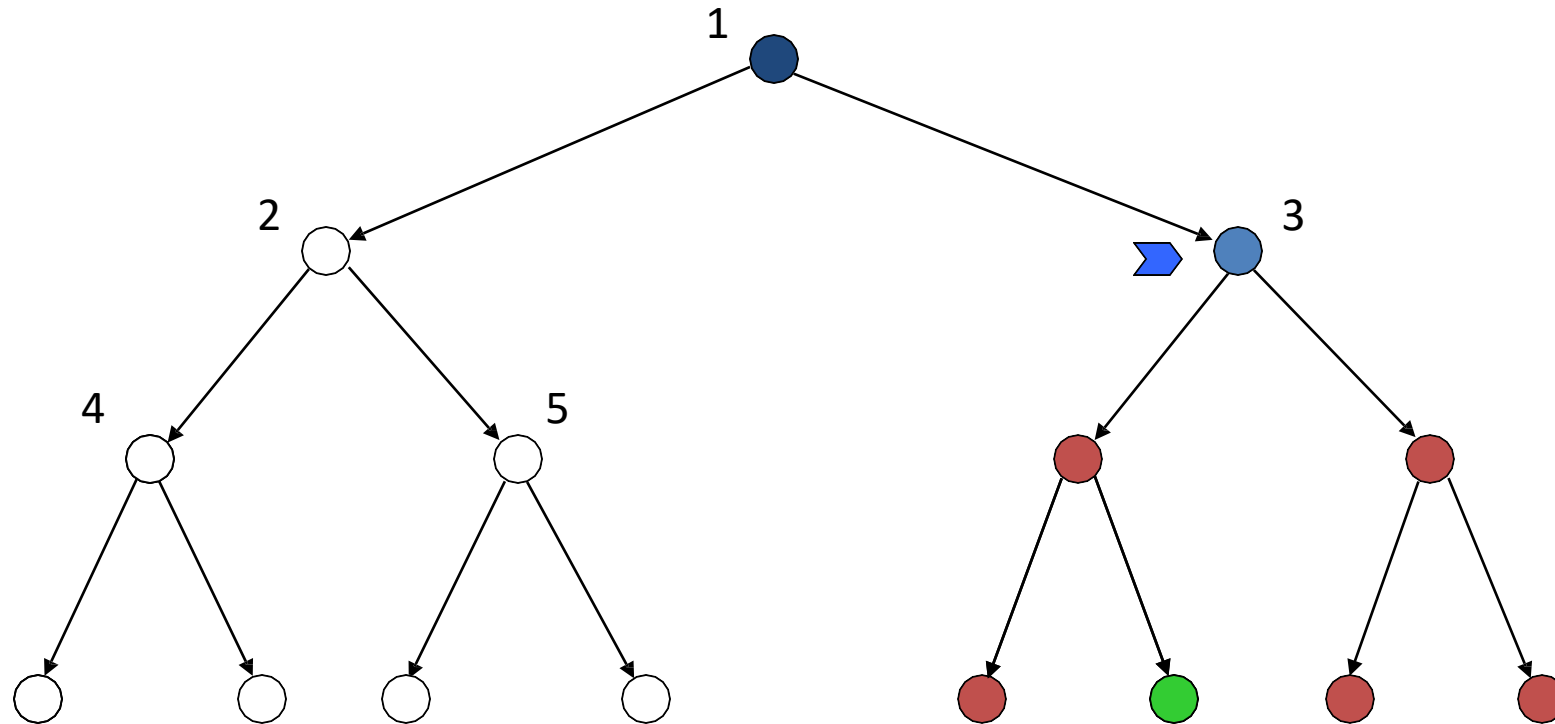
New nodes are inserted **at the front** of FRINGE



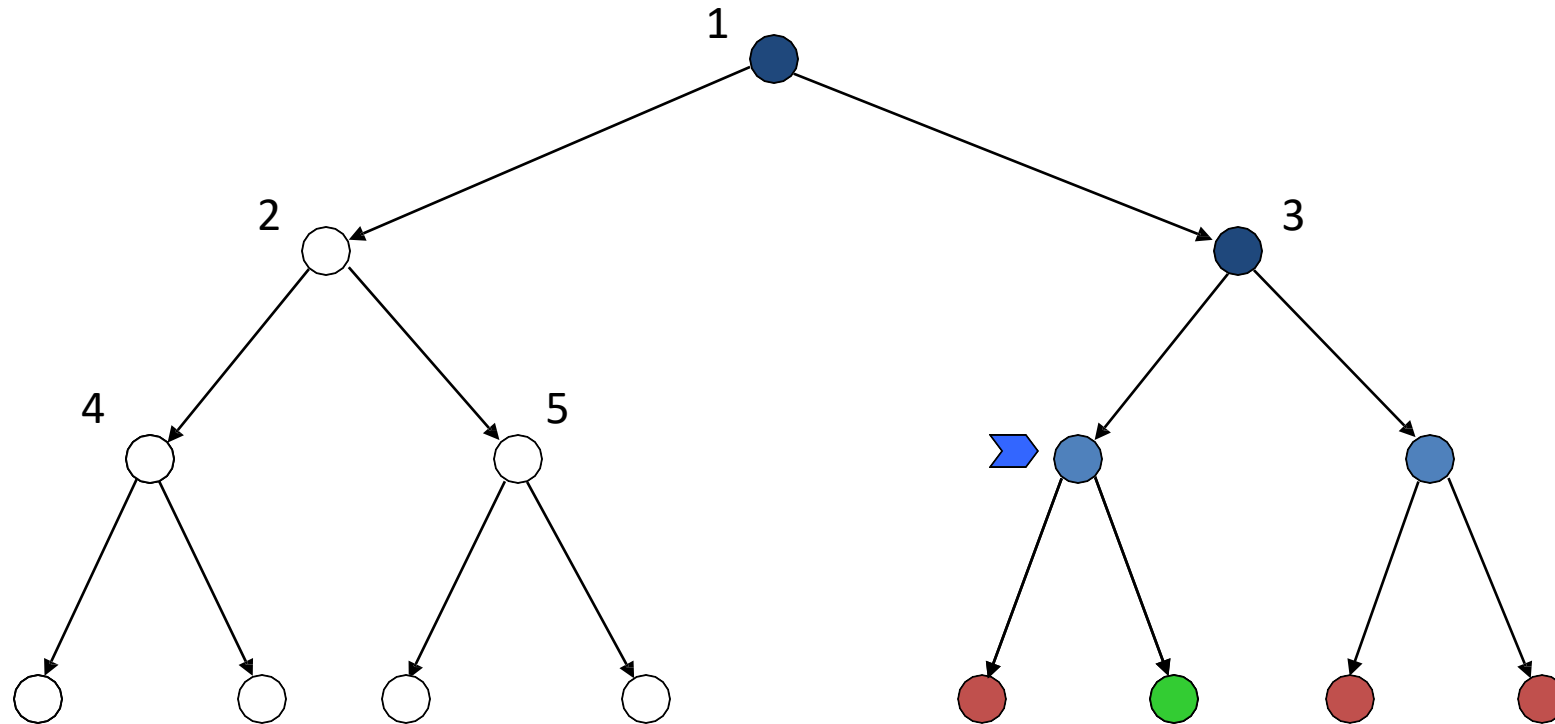
New nodes are inserted **at the front** of FRINGE



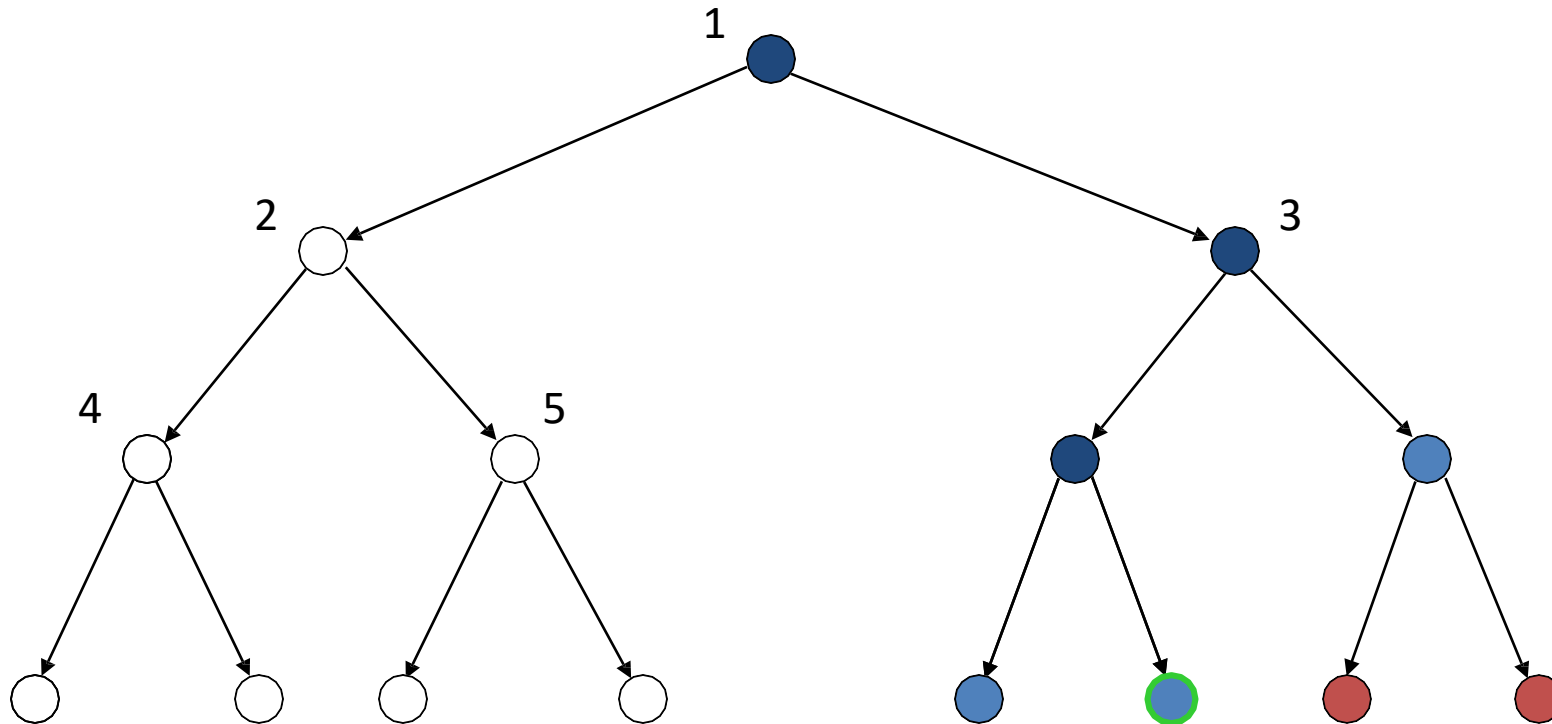
New nodes are inserted **at the front** of FRINGE



New nodes are inserted **at the front** of FRINGE

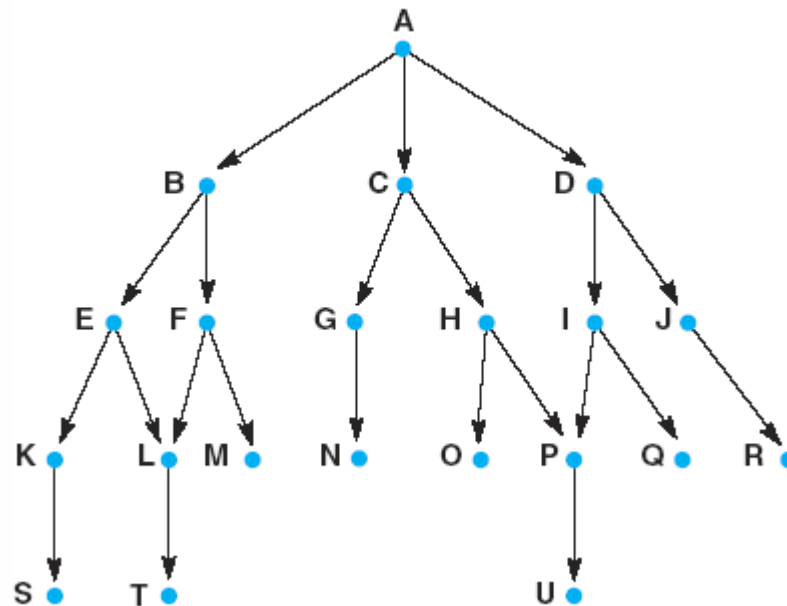


New nodes are inserted **at the front** of FRINGE

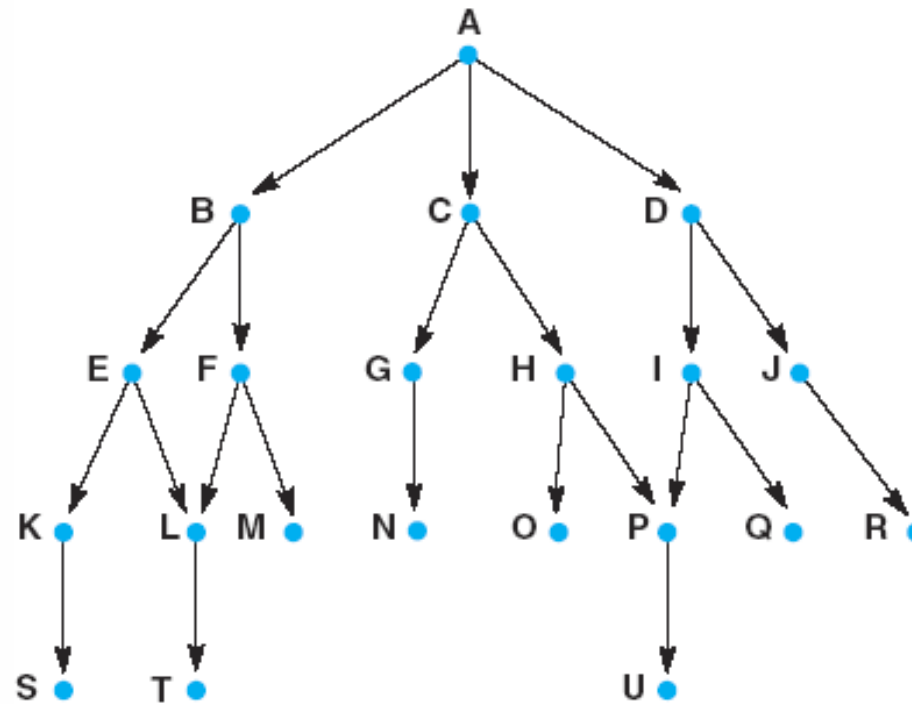


Graph for depth first search examples.

A Depth first search of the graph considers the states in the order: A,B,E,K,S,L,T,F,M,C,G,N,H,O,P,U,D,I,Q,J,R



A breadth first search of the graph considers the states in the order:
A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, Q, R, S, T, U.

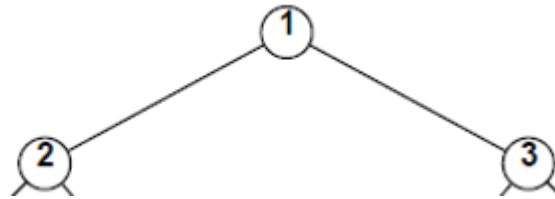


Consider a state space where the start state is number 1 and each state k has two successors: numbers $2k$ and $2k + 1$.

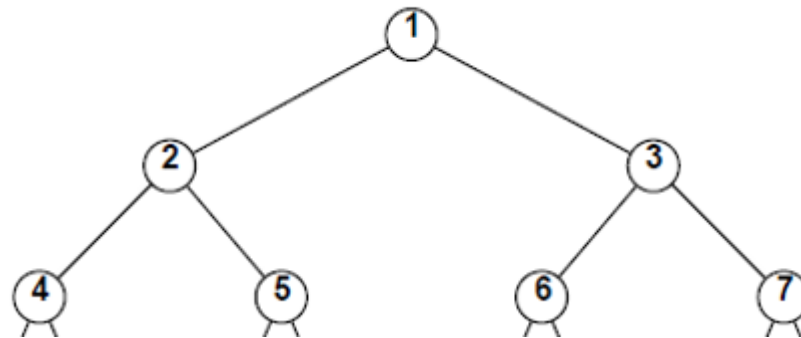
- Draw the portion of the state space for states 1 to 15.

Consider a state space where the start state is number 1 and each state k has two successors: numbers $2k$ and $2k + 1$.

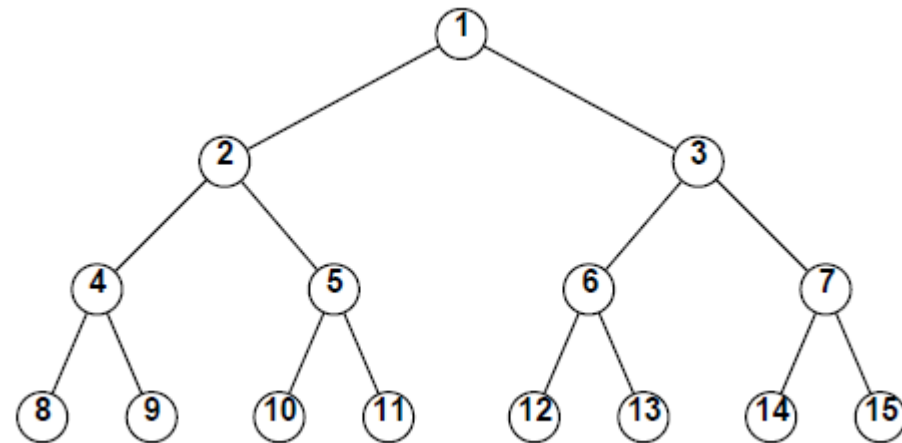
- Draw the portion of the state space for states 1 to 15.



Lecture 4



Lecture 4



- **Search tree**
 - Tree Search Algorithm Outline
 - Handling repeated states
- **Backtracking Search**
 - Backtracking Algorithm Data Structure
 - Backtracking Algorithm
- **Blind Strategy**
 - Depth-First Strategy
 - Breadth- First Strategy



مجموعة جامعة
الجميع الذكية
للتعليم القابضة

MIDOCEAN
UNIVERSITY
BY EVERYONE'S SMART UNIVERSITY



Thank You

