# Lecture Plan

1. **Introduction**
   - Problem definition
   - The Tree-Search algorithm
   - The Graph-Search algorithm
2. Graph as a model of the Search Space
3. Blind search strategies

# Problem definition

## We define a problem by providing:

- State space — $S$.
- Initial state — $s_I \in S$.
- Actions available within given state.[a]
- Goal test or goal state — $s_G \in S$.[b]
- Cost function $\gamma$,
- Forbidden states — $F \subset S$.

[a] Usually defined as so-called *successor function* which, for a given state, returns the set of available actions.

[b] Explicit (designated state) or implicit (goal satisfaction solutions).

## Formal problem definition

A problem P is defined as six-tuple:

$$P = (S, s_I, s_G, F, O, \gamma).$$

# State-Space and Problem Solution

## State-Space

A *state-space* is a set of *potential/feasible/legal states* of some system. A *state-space* can be discrete (finite) or continuous.

## States

A *state* representes *local* description of a system which is:

- complete,
- consistent,
- minimal.

## Problem solution

For $P = (S, s_I, s_G, F, O, \gamma)$ its *solution* is defined by a sequnece $(o_1, o_2, \ldots, o_n)$, such that:

$$o_1(s_I) = s_1, o_2(s_1) = s_2, \ldots, o_n(s_{n-1}) = s_G$$

Induced sequnece of states $s_I = s_0, s_1, s_2, \ldots, s_n = s_G$.

# Example: a trip to Romania

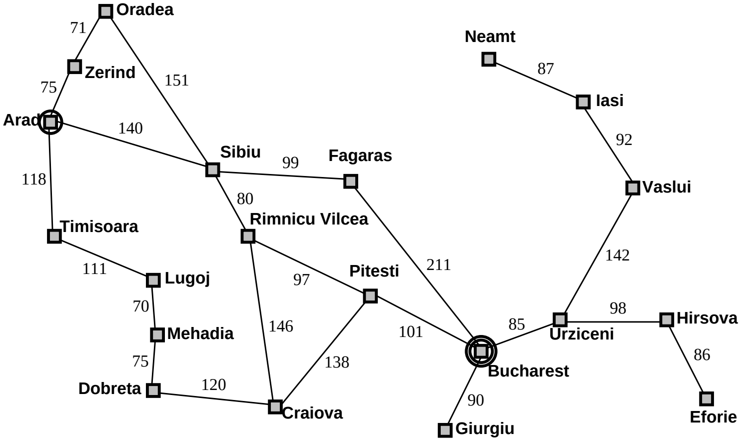
## Example problem statement

We are in Arad in Romania.
Our plane leaves tomorrow from Bucharest.

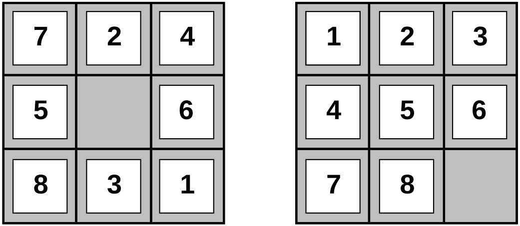
## Problem statement

- **State space:** distingushed cities
- **Initial state:** Arad
- **Available actions:** travel to another city (see map)
- **Goal test:** Are we in Bucharest? or
- **Goal state:** Bucharest
- **Cost function:** sum of road lengths to given city
- **Forbidden states:** optional

# Map of Romania



# Example: 8-puzzle




**Start State** **Goal State**

## Problem statement

- **State space:** all possible combinations of tile locations
- **Initial state:** any of the above states
- **Available actions:** legal moves (from: *Left*, *Right*, *Up*, *Down*)
- **Goal test:** are all tiles in order?
- **Cost function:** number of steps in path

# Example State-Space Search Problems



## Toyproblems

- Missionaries and cannibals,
- Towers of Hanoi,
- Block World,
- Criptoarithmetic problems,
- N-queens problem.

## More serious applications

- Route planning,
- Agent action planning,
- Robot navigation,
- Symbolic integration, term rewriting,
- Configuration, assembly, package planning.

# State space vs. search tree

## The difference

- State space represents the states of the search space.
- Search tree shows how we proceed within the state space.

## A node in the search tree consists of:

- **state** to which it corresponds,
- **parent node** which generated (among others) this node,
- **action** applied to generate this node,
- **path cost** $g(n)$ from initial state to this node,
- **depth**, i.e. number of steps from initial state.

# Informal Tree-Search algorithm

```
function TREE-SEARCH( problem, strategy) returns a solution, or failure
   initialize the search tree using the initial state of problem
   loop do
      if there are no candidates for expansion then return failure
      choose a leaf node for expansion according to strategy
      if the node contains a goal state then return the corresponding solution
      else expand the node and add the resulting nodes to the search tree
   end
```

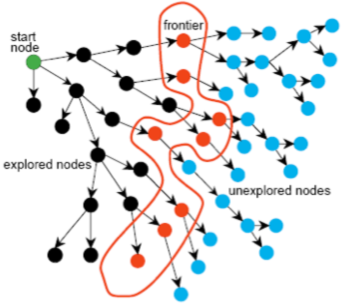
# Important definitions

## Search strategy

Decides, which node (among those determined by available actions) is expanded next.

## Fringe (frontier)

Queue of nodes to be expanded.



# Evaluation of search algorithms

## Completeness

Is the algorithm guaranteed to find a solution if one exists?

## Optimality

When the algorithm finds a solution, is this the optimal one?

## Time complexity

How long does it take to find a solution?[a]

[a] Often measured as number of nodes generated during search.

## Space complexity

How much memory is needed to perform the search?[a]

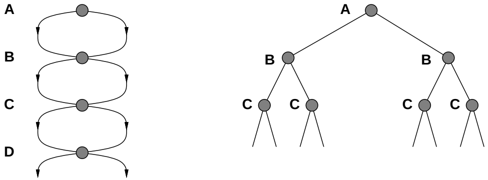
[a] Often measured as maximum number of nodes stored in memory.

# The Tree-Search algorithm

```
function TREE-SEARCH( problem, fringe) returns a solution, or failure
   fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
   loop do
       if fringe is empty then return failure
       node ← REMOVE-FRONT(fringe)
       if GOAL-TEST(problem, STATE(node)) then return node
       fringe ← INSERTALL(EXPAND(node, problem), fringe)


function EXPAND( node, problem) returns a set of nodes
   successors ← the empty set
   for each action, result in SUCCESSOR-FN(problem, STATE[node]) do
       s ← a new NODE
       PARENT-NODE[s] ← node;   ACTION[s] ← action;   STATE[s] ← result
       PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
       DEPTH[s] ← DEPTH[node] + 1
       add s to successors
   return successors
```

# Avoiding repeated states




## Problems

- Loops (solution: remember path).
- Infinite paths (solution: limit cost).
- Repeated search of nodes (solution: store all nodes).
- Any algorithm that forgets its history is doomed to repeat it.

## Solution

- Modify the **Tree-Search** algorithm to include a so-called *closed list*, storing every expanded node.
- The new algorithm is called **Graph-Search**.

# The Graph-Search algorithm

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
    closed ← an empty set
    fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
    loop do
        if fringe is empty then return failure
        node ← REMOVE-FRONT(fringe)
        if GOAL-TEST(problem, STATE[node]) then return node
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
    end
```

# Lecture Plan

# Definitions of a graph

## Basic intuitive formulation

A *graph* is a set of *nodes* (vertices) interconnected with *links* (edges). A graph is a *model* of a *search space*.

# Formal definitions of a graph I



## Definition: Simple Directed Graph

- $V$ — a finite set of *vertices* (or *nodes*), $V = \{v_1, v_2, \ldots, v_n\}$,
- $E$ — a finite set of *edges* (or *links*); $E \subseteq V \times V$.

A *simple directed graph* $G$ is defined as

$$G = (V, E).$$

## Definition: Directed Graph

A directed graph $G$ is any four-tuple

$$G = (V, E, \alpha, \omega)$$

- $\alpha\colon E \to V$ is a function defining the starting point of an edge,
- $\omega$ is a function $\omega\colon E \to V$ defining the end point of an edge.

# Formal definitions of a graph II



## Definition: Undirected Graph

An undirected graph $G$ is any triple

$$G = (V, E, \lambda)$$

where $\lambda$ is a function of the form $\lambda \colon E \to V2$, $V2 = \{\{v_i, v_j\} \colon v_i, v_j \in V\}$ defining the endpoints for an edge.[a]

[a] An alternative definition is also possible: $G = (V, E, \lambda), \lambda : E \to V^2$; however, the definition used is more appropriate for further statements.

# Formal definitions of a graph III

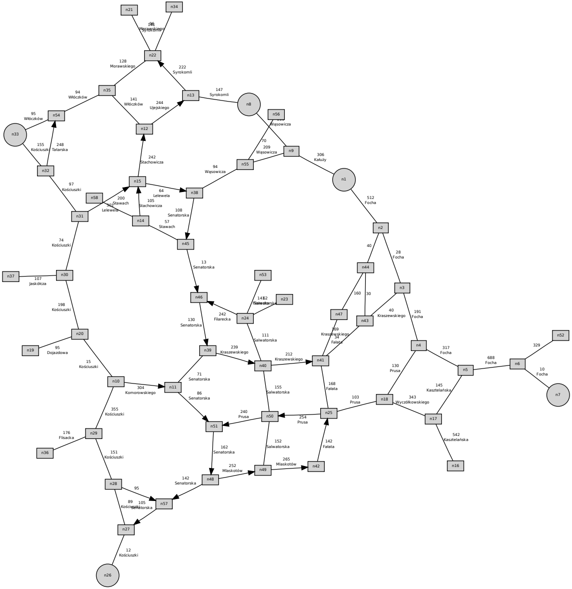
## Definition: Mixed Graph

A mixed graph $G$ is any five-tuple

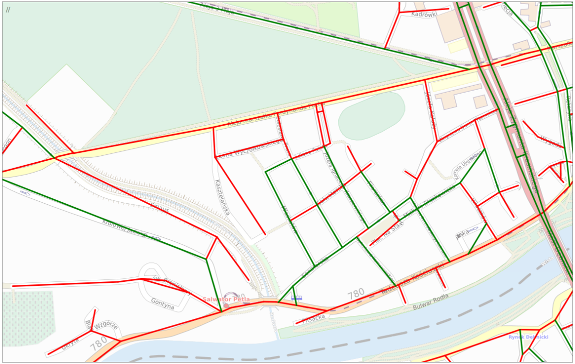
$$G = (V, E_1, E_2, \alpha, \omega, \lambda)$$

where $E_1$ is a set of directed links, $E_2$ is a set of undirected links; sets $E_1$ and $E_2$ are disjoint ($E_1 \cap E_2 = \emptyset$). Further $\lambda$, $\alpha$ and $\omega$ are defined as before; $\alpha$ and $\omega$ are defined over $E_1$ and $\lambda$ is defined over $E_2$.

# Graphs: visualisation



# Graphs: visualisation



# Lecture Plan

1 Introduction

2 Graph as a model of the Search Space

3 Blind search strategies

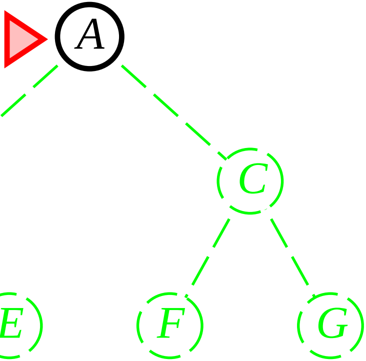
- Breadth-First Search (BFS)
- Uniform-Cost Search (UCS)
- Depth-First Search (DFS)
- Iterative Deepening Search
- Dijkstra's Algorithm
- Comparison of Algorithms

# Breadth-First Search (BFS)



## Breadth-first search strategy

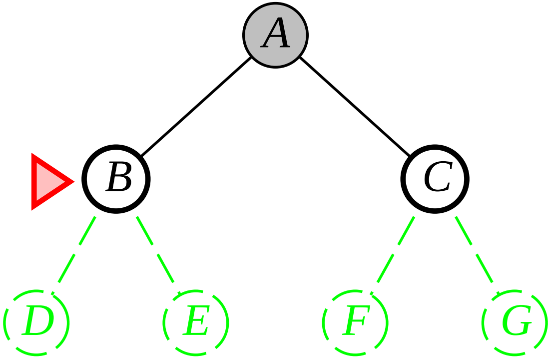
- All nodes on a given level are expanded first.
- Only then deeper nodes are expanded.



# Breadth-First Search (BFS)

## Breadth-first search strategy

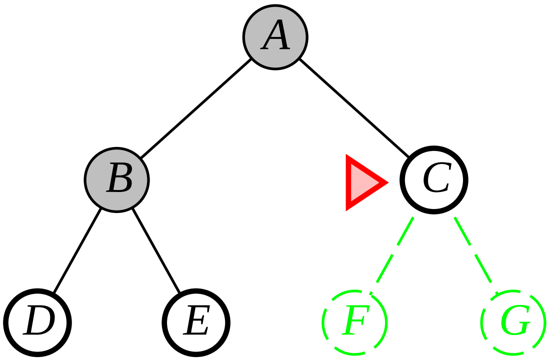
- All nodes on a given level are expanded first.
- Only then deeper nodes are expanded.



# Breadth-First Search (BFS)

## Breadth-first search strategy

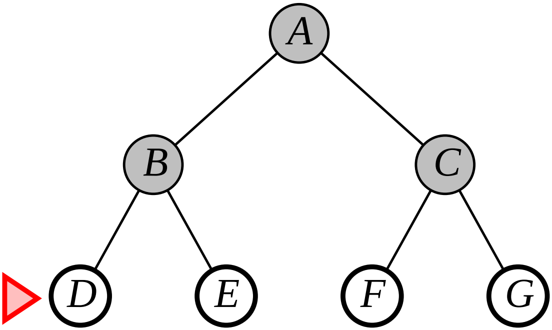
- All nodes on a given level are expanded first.
- Only then deeper nodes are expanded.



# Breadth-First Search (BFS)

## Breadth-first search strategy

- All nodes on a given level are expanded first.
- Only then deeper nodes are expanded.



# Breadth-First Search (BFS)

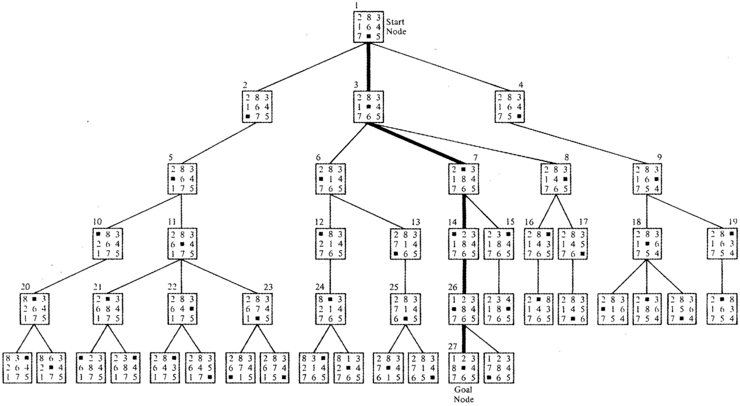


## BFS characteristics

- **Completeness:** if the branching factor is finite and the goal node is at depth $d$, BFS will eventually find it.
- **Optimality:** BFS is optimal if path cost is a non-decreasing function of depth.[a]
- **Time complexity:** $1 + b + b^2 + b^2 + \ldots + b^d + b(b^d - 1) = O(b^{d+1})$.
- **Space complexity:** $O(b^{d+1})$.[b]

[a] Otherwise, the shallowest node may not necessarily be optimal.
[b] $b$ – branching factor; $d$ – depth of the goal node

# BFS in 8-puzzle problem



# Uniform-Cost Search (UCS)



## Uniform-cost search strategy

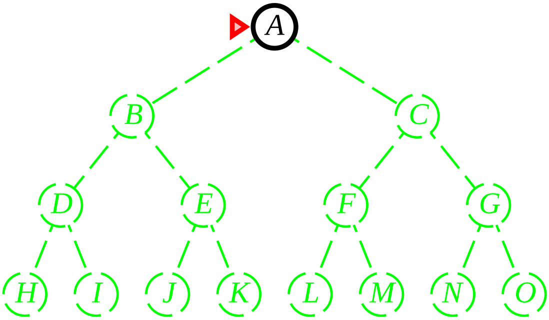
- Expands the node with the lowest cost from the start node first.[a]
- **Completeness:** yes, if cost of each step $\geq \epsilon > 0$.
- **Optimality:** same as above – nodes are expanding in increasing order of cost.
- **Time and space complexity:** $O(b^{1+\lceil C*/\epsilon \rceil})$.[b]

[a] If all step costs are equal, UCS=BFS.
[b] $C*$ – cost of optimal solution

# Depth-First Search (DFS)

## Depth-first search strategy

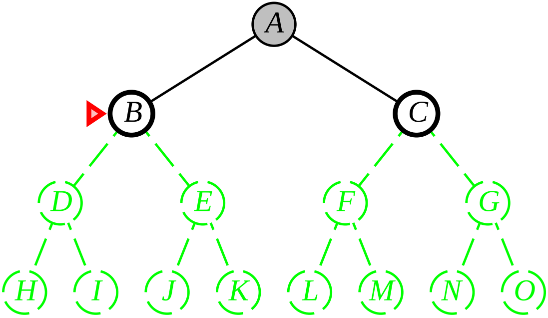
- DFS first expands the deepest node in the fringe.



# Depth-First Search (DFS)

## Depth-first search strategy

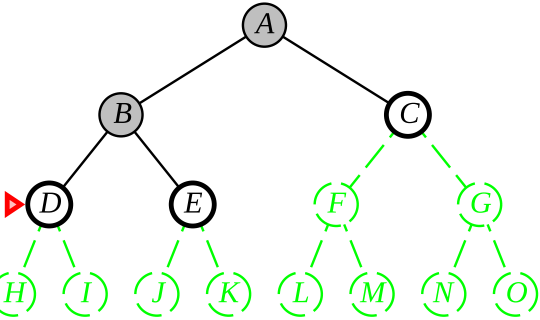
- DFS first expands the deepest node in the fringe.



# Depth-First Search (DFS)

## Depth-first search strategy

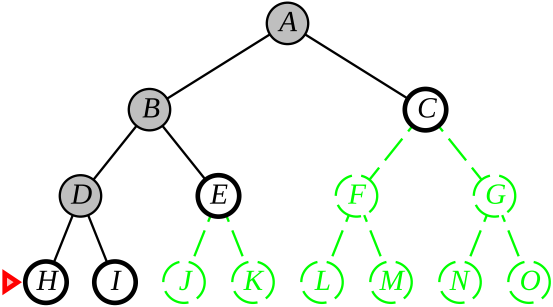
- DFS first expands the deepest node in the fringe.



# Depth-First Search (DFS)

## Depth-first search strategy

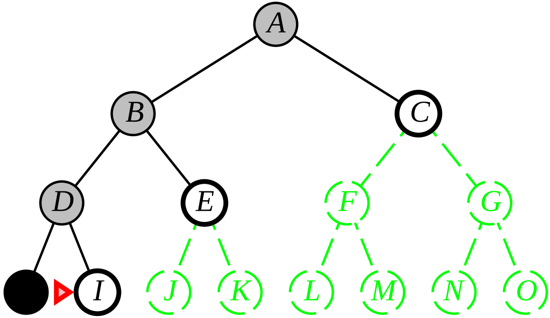
- DFS first expands the deepest node in the fringe.



# Depth-First Search (DFS)

## Depth-first search strategy

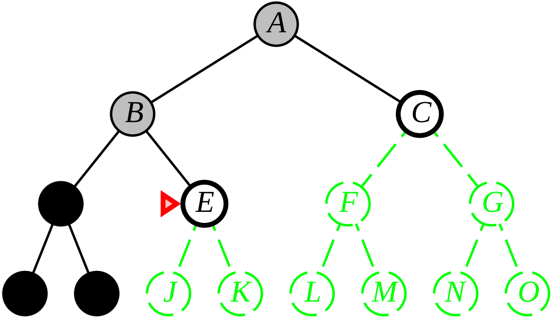
- DFS first expands the deepest node in the fringe.



# Depth-First Search (DFS)

## Depth-first search strategy

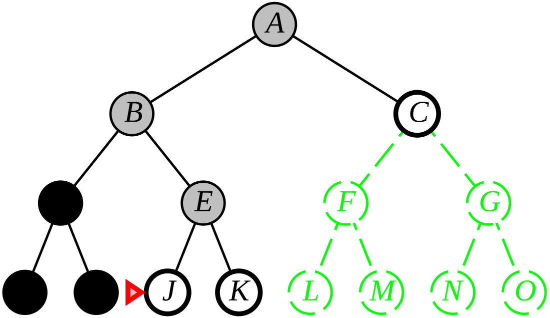
- DFS first expands the deepest node in the fringe.



# Depth-First Search (DFS)

## Depth-first search strategy

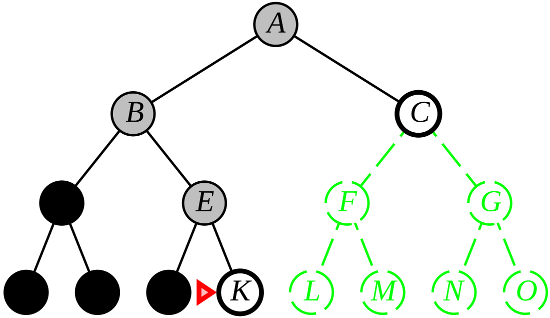
- DFS first expands the deepest node in the fringe.



# Depth-First Search (DFS)

## Depth-first search strategy

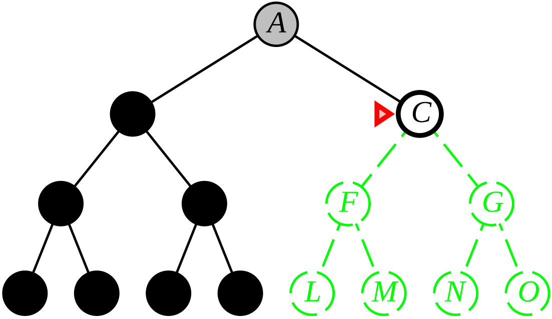
- DFS first expands the deepest node in the fringe.



# Depth-First Search (DFS)

## Depth-first search strategy

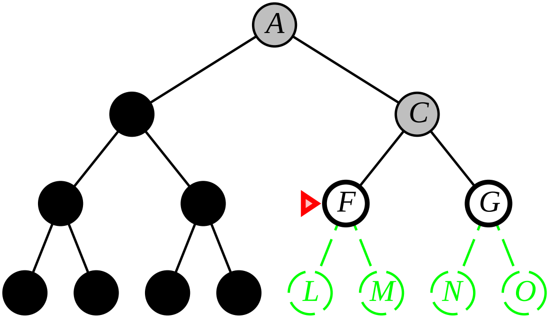
- DFS first expands the deepest node in the fringe.



# Depth-First Search (DFS)

## Depth-first search strategy

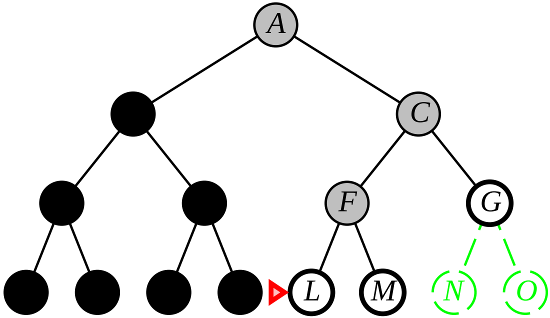
- DFS first expands the deepest node in the fringe.



# Depth-First Search (DFS)

## Depth-first search strategy

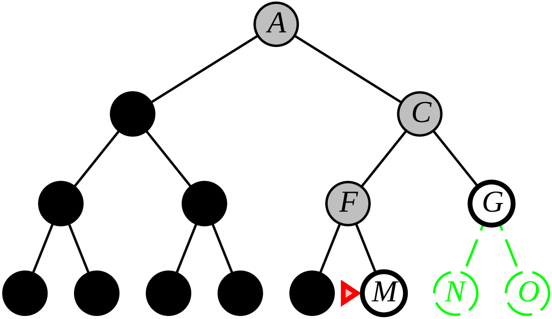
- DFS first expands the deepest node in the fringe.



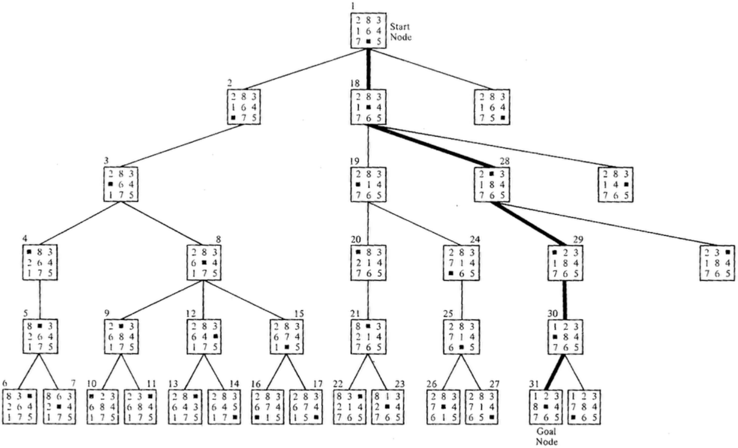
# Depth-First Search (DFS)

## Depth-first search strategy

- DFS first expands the deepest node in the fringe.



# DFS in 8-puzzle problem



# Depth-First Search (DFS)

## DFS characteristics

- Small space requirements: only the path to the current node and the siblings of each node in the path are stored.
- *Backtracking search* generates only one successor for each node.
- **Completeness:** no, if the expanded subtree has an infinite depth.
- **Optimality:** no, if a solution located deeper, but located in a subtree expanded earlier, is found.
- **Time complexity:** $O(b^m)$.
- **Space complexity:** $O(bm)$ (linear!).

# Depth-Limited Search (DLS)



## Depth-limited search strategy

- Modification of Depth-First Search.
- We introduce a maximum depth $\ell$; nodes located at depth $\ell$ are treated as if they had no successors.
- Returns two error types: *failure* means no solution, *cutoff* means no solution within given depth limit.

## Why DLS is not widely used?

For most problems, one does not know a good depth limit until the problem is solved...

# Depth-Limited Search (DLS)

```
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 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
```

# Iterative-Deepening Search (IDS)

## Iterative-deepening depth-first search strategy

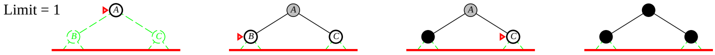
- IDS – *Iterative Deepening Search*.
- IDS is a strategy for determination of optimal depth limit.
- Just like BFS, IDS expands an entire layer of new nodes before going deeper.

Limit = 0

# Iterative-Deepening Search (IDS)

## Iterative-deepening depth-first search strategy

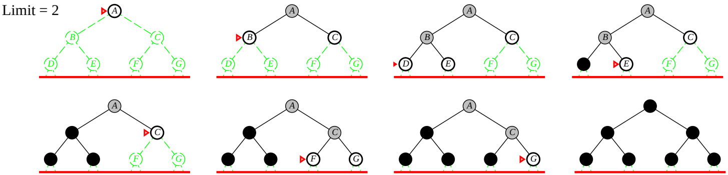
- IDS – *Iterative Deepening Search*.
- IDS is a strategy for determination of optimal depth limit.
- Just like BFS, IDS expands an entire layer of new nodes before going deeper.



# Iterative-Deepening Search (IDS)

## Iterative-deepening depth-first search strategy

- IDS – *Iterative Deepening Search*.
- IDS is a strategy for determination of optimal depth limit.
- Just like BFS, IDS expands an entire layer of new nodes before going deeper.

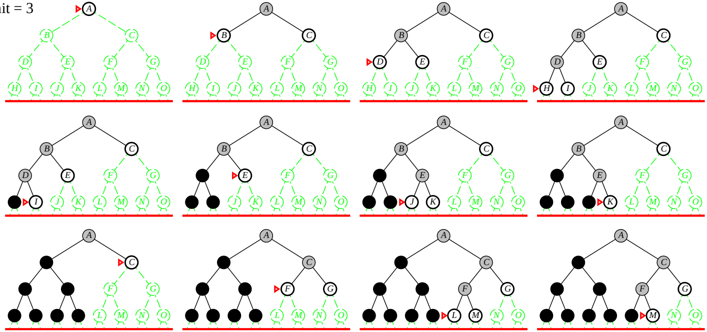


# Iterative-Deepening Search (IDS)

## Iterative-deepening depth-first search strategy

- IDS – *Iterative Deepening Search*.
- IDS is a strategy for determination of optimal depth limit.
- Just like BFS, IDS expands an entire layer of new nodes before going deeper.

Limit = 3



# Iterative-Deepening Search (IDS)

## IDS characteristics

- **Completeness:** yes.
- **Optimality:** yes, if step cost = 1.
- **Time complexity:** $(d+1)b^0 + db^1 + (d-1)b^2 + \ldots + b^d = O(b^d)$.
- **Space complexity:** $O(bd)$.

## Numerical comparison for $b = 10$, $d = 5$

$$N(IDS) = 50 + 400 + 3000 + 20000 + 100000 = 123450$$

$$N(BFS) = 10 + 100 + 1000 + 10000 + 100000 + 999990 = 1111100$$

## Conclusion

IDS exhibits better performance, because it does not expand other nodes at depth $d$.

# Dijkstra's Algorithm

## General description

- Finds the shortest path with a single source in a graph with non-negative edge weights.

## More information and examples (clickable links)

- Wikipedia PL
- Wikipedia EN
- I LO w Tarnowie

# Bidirectional Search

### Ideas for bidirectional search

Start the search from the initial node and the goal node in parallel. The main advantage: depth cut by 2.

### Problems with bidirectional search

- works only for BFS-like strategies,
- costly test for achieving the goal,
- goal node must be defined explicitly (not a goal test),
- expand function must be reversible.

# Comparison of Algorithms

| Criterion | BFS | UCS | DFS | DLS | IDS |
|---|---|---|---|---|---|
| Complete? | Yes[1] | Yes[2] | No | Yes[3] | Yes |
| Time | $b^{d+1}$ | $b^{1+\lceil C*/\epsilon \rceil}$ | $b^m$ | $b^l$ | $b^d$ |
| Space | $b^{d+1}$ | $b^{1+\lceil C*/\epsilon \rceil}$ | $bm$ | $bl$ | $bd$ |
| Optimal? | Yes[4] | Yes | No | No | Yes[5] |

where:

- $b$ – branching factor,
- $d$ – depth of shallowest solution,
- $m$ – maximum depth of search tree,
- $l$ – depth limit.

[1] if $b < \infty$
[2] if $b < \infty$ and step costs $\geq \epsilon > 0$
[3] if $l \geq d$
[4] if step costs are equal
[5] if step costs are equal

# Concluding Remarks

## For blind strategies

- DL is often preferred,
- the current path is used to avoid cycles,
- depth is checked to stay within the current depth limit,
- the cost function may be used to restrict search,
- forbidden states restrict the search,
- dynamic search-space reconfiguration with constraint propagation.

# DFS in Prolog

```
path(F,F,T,T).
path(I,F,Path,T):-
   p(I,N),
   not(member(N,Path)),
   path(N,F,[N|Path],T).

go(I,F,Path):-
   path(I,F,[I],Path).
```

# BFS in Prolog

```
bf(F,[[F|Path]|_],[F|Path]).
  bf(F,[Path|SetOfPaths],T):-
     extend(Path,NewPaths),
     append(SetOfPaths,NewPaths,NewSetOfPaths),
     bf(F,NewSetOfPaths,T).

  extend([N|Path],NewPaths):-
     bagof([NextN,N|Path],(p(N,NextN),
      not(member(NextN,[N|Path]))),NewPaths),!.
  extend(_,[]).

  go(I,F,Path):-
     bf(F,[[I]],Path).
```