Materials

Ch 2 & 3 of Artificial Intelligence A Systems Approach by Tim Jones
Chapters 3 and 4 of Artificial Intelligence a Modern Approach by Russell and Norvig

Problem solving, search and control strategies

What are problem solving, search and control strategies ?

- **Problem solving is fundamental to many AI-based applications.**

  There are two types of problems.
  - The Problems like, computation of the sine of an angle or the square root of a value. These can be solved through the use of deterministic procedure and the success is guaranteed.
  - In the real world, very few problems lend themselves to straightforward solutions.

  Most real world problems can be solved only by searching for a solution.
  AI is concerned with these type of problems solving.

- **Problem solving is a process** of generating solutions from observed data.
  - a problem is characterized by a set of goals,
  - a set of objects, and
  - a set of operations.

  These could be ill-defined and may evolve during problem solving.

- **Problem space** is an abstract space.
  - A problem space encompasses all *valid states* that can be generated by the application of any combination of *operators* on any combination of *objects*.
  - The problem space may contain one or more *solutions*.

  Solution is a combination of operations and objects that achieve the goals.

- **Search** refers to the search for a solution in a problem space.
  - Search proceeds with different types of *search control strategies*.
  - The *depth-first search and breadth-first search* are the two common search strategies.

General Problem solving

Problem solving has been the key areas of concern for Artificial Intelligence.

- **Problem solving is a process of generating solutions** from observed or given data. It is however not always possible to use direct methods (i.e. go directly from data to solution). Instead, problem solving often need to use indirect or model-based methods.
- **General Problem Solver (GPS) was a computer program** created in 1957 by *Simon* and *Newell* to build a universal problem solver machine. GPS was based on Simon and Newell's theoretical work on logic machines. GPS in principle can solve any formalized symbolic problem, like : theorems proof and geometric problems and chess playing.
- GPS solved many simple problems such as the Towers of Hanoi, that could be sufficiently formalized, but **GPS could not solve any real-world problems**.

To build a system to solve a particular problem, we need to

- Define the problem precisely – find input situations as well as final situations for acceptable solution to the problem.
- Analyze the problem – find few important features that may have impact on the appropriateness of various possible techniques for solving the problem.
- Isolate and represent task knowledge necessary to solve the problem
- Choose the best problem solving technique(s) and apply to the particular problem.

GENERAL PROBLEM SOLVING

Problem Definitions :

A *problem* is defined by its *elements* and their *relations*.

To provide a formal description of a problem, we need to do following:

a. Define a *state space* that contains all the possible configurations of the relevant objects, including some impossible ones.
b. Specify one or more states, that describe possible situations, from which the problem-solving process may start. These states are called *initial states*.
c. Specify one or more states that would be acceptable solution to the problem. These states are called *goal states*.
d. Specify a set of *rules* that describe the *actions* (operators) available.

The problem can then be solved by using the *rules*, in combination with an appropriate *control strategy*, to move through the *problem space* until a *path* from an *initial state* to a *goal state* is found.

This process is known as **search**.

- Search is fundamental to the problem-solving process.
- Search is a general mechanism that can be used when more direct method is not known.
- Search provides the framework into which more direct methods for solving subparts of a problem can be embedded.

A very large number of AI problems are formulated as search problems.

Problem Space

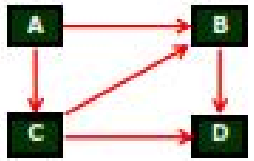
A *problem space* is represented by directed *graph*, where *nodes* represent *search state* and *paths* represent the *operators* applied to change the *state*.

To simplify a search algorithms, it is often convenient to logically and programmatically represent a problem space as a **tree**. A tree usually decreases the complexity of a search at a **cost**. Here, the cost is due to duplicating some nodes on the tree that were linked numerous times in the graph; e.g., node **B** and node **D** shown in example below.

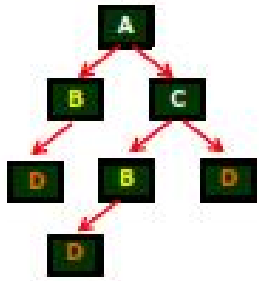
A **tree** is a graph in which any two vertices are connected by exactly one path. Alternatively, any connected graph with no cycles is a tree.

Examples

Graph



Trees



GENERAL PROBLEM SOLVING

Problem Solving

The term Problem Solving relates analysis in AI. Problem solving may be characterized as a *systematic search* through a range of possible actions to reach some predefined goal or solution. Problem-solving methods are categorized as *special purpose* and *general purpose*.

- **Special-purpose method** is tailor-made for a particular problem, often exploits very specific features of the situation in which the problem is embedded.
- **General-purpose method** is applicable to a wide variety of problems. One general-purpose technique used in AI is "means-end analysis". It is a step-by-step, or incremental, reduction of the difference between current state and final goal.

Examples : Tower of Hanoi puzzle

- For a Robot this might consist of PICKUP, PUTDOWN, MOVEFORWARD, MOVEBACK, MOVELEFT, and MOVERIGHT—until the goal is reached.
- Puzzles and Games have explicit rules : e.g., the Tower of Hanoi puzzle.

Tower of Hanoi puzzle.

- This puzzle may involves a set of rings of different sizes that can be placed on three different pegs.
- The puzzle starts with the rings arranged as shown in Fig. (a)
- The goal of this puzzle is to move them all as to Fig. (b)
- Condition : Only the top ring on a peg can be moved, and it may only be placed on a smaller ring, or on an empty peg.

In this Tower of Hanoi puzzle : Situations encountered while solving the problem are described as *states*. The set of all possible configurations of rings on the pegs is called *problem space*.

States

A *state* is a representation of elements at a given moment. A problem is defined by its *elements* and their *relations*.

At each instant of a problem, the elements have specific descriptors and relations; the *descriptors* tell - how to select elements ?

Among all possible states, there are two special states called :

- *Initial state* is the start point
- *Final state* is the goal state

State Change: Successor Function

A *Successor Function* is needed for state change.

The successor function moves one state to another state.

Successor Function :

- Is a description of possible actions; a set of operators.
- Is a transformation function on a state representation, which converts that state into another state.
- Defines a relation of accessibility among states.
- Represents the conditions of applicability of a state and corresponding transformation function

State Space

A *State space* is the set of all states reachable from the *initial state*.

Definitions of terms :

- A *state space* forms a *graph* (or map) in which the *nodes* are states and the *arcs* between nodes are actions.
- In *state space*, a *path* is a sequence of states connected by a sequence of actions.
- The *solution* of a problem is part of the map formed by the *state space*.

GENERAL PROBLEM SOLVING

Structure of a State Space

The *Structures* of *state space* are *trees* and *graphs*.

- Tree is a hierarchical structure in a graphical form; and
- Graph is a non-hierarchical structure.

- **Tree** has only one path to a given node;
  i.e., a *tree* has one and only one path from any point to any other point.
- **Graph** consists of a set of nodes (vertices) and a set of edges (arcs).
  Arcs establish relationships (connections) between the nodes;
  i.e., a graph has several paths to a given node.
- **operators** are directed *arcs* between nodes.

Search process explores the *state space*. In the worst case, the search explores all possible *paths* between the *initial state* and the *goal state*.

Problem Solution

In the *state space*, a *solution is a path* from the *initial state* to a *goal state* or sometime just a *goal state*.

- A Solution cost function assigns a numeric cost to each path;
  It also gives the cost of applying the operators to the states.
- A Solution quality is measured by the path cost function; and
  An optimal solution has the lowest path cost among all solutions.
- The solution may be any or optimal or all.
- The importance of cost depends on the problem and the type of solution asked.

Problem Description

A problem consists of the description of :

- *current state* of the world,
- *actions* that can transform one state of the world into another,
- *desired state* of the world.

- **State space** is defined explicitly or implicitly
  A state space should describe everything that is needed to solve a problem and nothing that is not needed to the solve the problem.
- **Initial state** is start state
- **Goal state** is the conditions it has to fulfill
  - A description of a desired state of the world;
  - The description may be complete or partial.
- **Operators** are to change state
  - Operators do actions that can transform one state into another.
  - Operators consist of : Preconditions and Instructions;
    - Preconditions provide partial description of the state of the world that must be true in order to perform the action,
    - Instructions tell on how to create next state.
  - Operators should be as general as possible, to reduce their number.
- **Elements of the domain** has relevance to the problem
  - Knowledge of the starting point.
- **Problem solving** is finding solution
  - Finding an ordered sequence of operators that transform the current (start) state into a goal state;
- **Restrictions** are solution quality any, optimal, or all
  - Finding the shortest sequence, or
  - Finding the least expensive sequence defining cost , or
  - Finding any sequence as quickly as possible.

GENERAL PROBLEM SOLVING

Examples of Problem Definitions

Example 1 :

A game of 8–Puzzle

- State space : configuration of **8 - tiles** on the board
- Initial state : any configuration
- Goal state : tiles in a specific order
- Action : "blank moves"
  - Condition: the move is within the board
  - Transformation: blank moves Left, Right, Up, Dn
- Solution : optimal sequence of operators

Solution

Example 2 :

A game of n - queens puzzle; n = 8

- State space : configurations **n = 8** queens on the board with only one queen per row and column
- Initial state : configuration without queens on the board
- Goal state : configuration with **n = 8** queens such that no queen attacks any other
- Operators or actions : place a queen on the board.
  - Condition: the new queen is not attacked by any other already placed
  - Transformation: place a new queen in a particular square of the board
- Solution : one solution (cost is not considered)

One Solution

AI: SEARCH AND CONTROL STRATEGIES

Word "Search" refers to the search for a solution in a problem space.

- Search proceeds with different types of **"Search Control strategies"**.
- A strategy is defined by picking the order in which the nodes expand.

The Search strategies are evaluated in the following dimensions:

Completeness, Time complexity, Space complexity, Optimality.

(the search related terms are first explained, then the search algorithms and control strategies are illustrated).

2.1 Search related terms

- **Algorithm's Performance and Complexity**

Ideally we want a common measure so that we can compare approaches in order to select the most appropriate algorithm for a given situation.

◇ **Performance** of an algorithm depends on internal and external factors.

Internal factors	External factors
‡ **Time** required, to run	‡ **Size** of input to the algorithm
‡ **Space** (memory) required to run	‡ **Speed** of the computer
	‡ **Quality** of the compiler

◇ **Complexity** is a measure of the performance of an algorithm. It measures the internal factors, usually in time than space.

Computational Complexity

A measure of resources in terms of Time and Space.

◇ If **A** is an algorithm that solves a decision problem **f** then run time of **A** is the number of steps taken on the input of length **n.**

◇ **Time Complexity T(n)** of a decision problem **f** is the run time of the 'best' algorithm **A** for **f**.

◇ **Space Complexity S(n)** of a decision problem **f** is the amount of memory used by the 'best' algorithm **A** for **f**.

"Big - O" notation

The "Big-O" is theoretical measure of the execution of an algorithm, usually indicates the *time* or the *memory* needed, given the problem size **n**, which is usually the number of items.

◇ **Big-O notation**

The **Big-O** notation is used to give an approximation to the run-time-efficiency of an algorithm ; the letter **"O"** is for order of magnitude of operations or space at run-time.

◇ **The Big-O of an Algorithm A**

- If an algorithm **A** requires time proportional to **f(n)**, then the algorithm **A** is said to be of order **f(n)**, and it is denoted as **O(f(n))**.
- If algorithm **A** requires time proportional to n^2, then order of the algorithm is said to be $O(n^2)$.
- If algorithm **A** requires time proportional to **n**, then order of the algorithm is said to be **O(n)**.

The function **f(n)** is called the algorithm's **growth-rate function**.

If an algorithm has performance complexity **O(n)**, this means that the run-time **t** should be directly proportional to **n**, ie $t \propto n$ or $t = k\,n$ where **k** is constant of proportionality.

Similarly, for algorithms having performance complexity $O(\log_2(n))$, **O(log N)**, **O(N log N)**, $O(2^N)$ and so on.

◇ **Example 1 :**

1-D array, determine the Big-O of an algorithm ;

Calculate the sum of the **n** elements in an integer array **a[0 . . . . n-1]**.

Line no	Instructions	No of execution steps
line 1	sum = 0	1
line 2	for (i = 0; i < n; i++)	n + 1
line 3	sum += a[i]	n
line 4	print sum	1
	Total	2n + 3

For the polynomial **(2*n + 3)** the Big-O is dominated by the 1st term as **n** while the number of elements in the array becomes very large. Also in determining the **Big-O**

‡ Ignoring constants such as **2** and **3**, the algorithm is of the order **n**.

‡ So the **Big-O** of the algorithm is **O(n)**.

‡ In other words the run-time of this algorithm increases roughly as the size of the input data **n**, say an array of size **n**.

Tree Structure

Tree is a way of organizing objects, related in a hierarchical fashion.

Tree is a type of data structure where

- each *element* is attached to one or more elements directly beneath it.
- the connections between elements are called *branches*.
- tree is often called *inverted trees* because its *root* is at the top.
- the elements that have no elements below them are called *leaves*.
- a *binary tree* is a special type, each element has two branches below it.

◇ **Example**

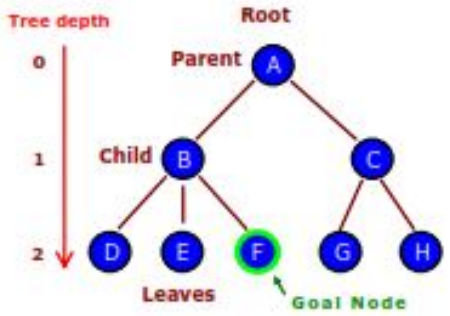


Fig. A simple example unordered tree

◇ **Properties**

- Tree is a special case of a *graph*.
- The topmost node in a tree is called the *root node*; at root node all operations on the tree begin.
- A node has at most one *parent*. The topmost node called root node has no parents.
- Each node has either zero or more *child nodes* below it .
- The Nodes at the bottom most level of the tree are called *leaf nodes*. The *leaf nodes* do not have children.
- A node that has a child is called the child's *parent node*.
- The *depth of a node* ***n*** is the length of the path from the root to the node; The root node is at depth zero.

Search

Search is the systematic examination of states to find path from the start/root state to the goal state.

- search usually results from a lack of knowledge.
- search explores knowledge alternatives to arrive at the best answer.
- search algorithm output is a solution, ie, a path from the initial state to a state that satisfies the goal test.

For general-purpose problem solving : "Search" is an approach.

- search deals with finding nodes having certain properties in a graph that represents search space.
- search methods explore the *search space* "intelligently", evaluating possibilities without investigating every single possibility.

Example : Search tree

The search trees are multilevel indexes used to guide the search for data items, given some search criteria.

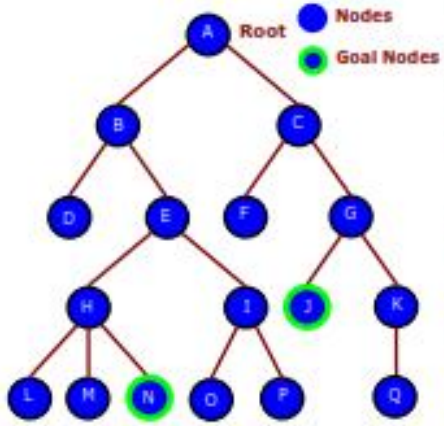


Fig. Tree Search.

The search starts at *root* and explores *nodes* looking for a *goal node*, that satisfies certain conditions depending on the problem.

For some problems, any goal node, **N** or **J** , is acceptable;

For other problems, it may be a minimum depth goal node, say **J** which is nearest to root.

Search Algorithms :

Many traditional search algorithms are used in AI applications. For complex problems, the traditional algorithms are unable to find the solution within some practical time and space limits. Consequently, many special techniques are developed, using **heuristic functions .**

The algorithms that use heuristic functions are called **heuristic algorithms**.

- Heuristic algorithms are *not really intelligent*; they appear to be intelligent because they achieve better performance.
- Heuristic algorithms are *more efficient* because they take advantage of feedback from the data to direct the search path.
- Uninformed search algorithms or Brute-force algorithms search, through the search space, all possible candidates for the solution checking whether each candidate satisfies the problem's statement.
- Informed search algorithms use heuristic functions, that are specific to the problem, apply them to guide the search through the search space to try to reduce the amount of time spent in searching.

A good heuristic can make an informed search dramatically out-perform any uninformed search. For example, the Traveling Salesman Problem (TSP) where the goal is to find a *good solution* instead of finding the *best solution*.

In TPS like problems, the search proceeds using current information about the problem to predict which path is closer to the goal and follow it, although it does not always guarantee to find the best possible solution. Such techniques help in finding a solution within reasonable time and space.

Some prominent intelligent search algorithms are stated below.

1. **Generate and Test Search**
2. **Best-first Search**
3. **Greedy Search**
4. **A* Search**
5. **Constraint Search**
6. **Means-ends analysis**

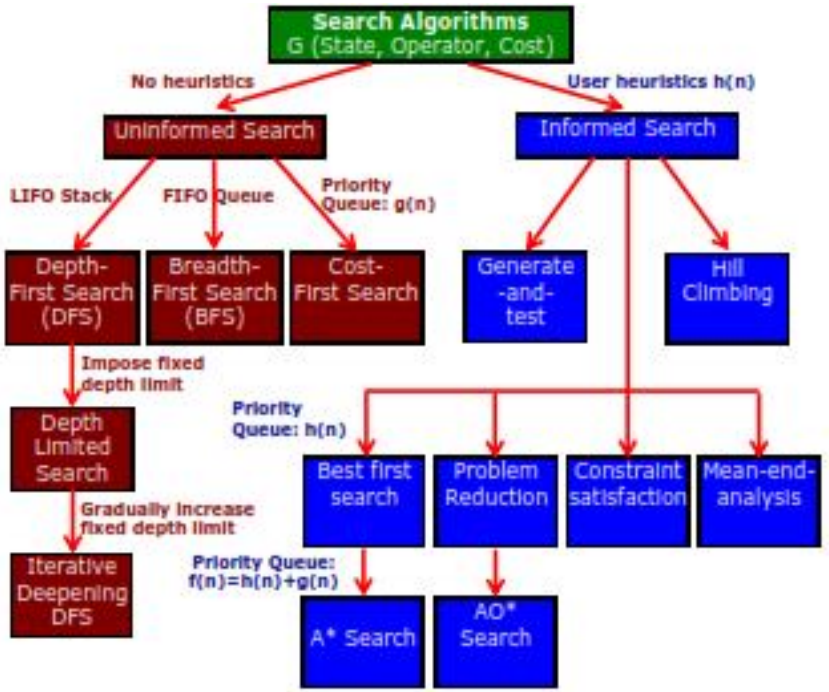
There are more algorithms, either an improvement or combinations of these.

Hierarchical Representation of Search Algorithms

A representation of most search algorithms is illustrated below. It begins with two types of search - Uninformed and Informed.

Uninformed Search : Also called *blind, exhaustive or brute-force* search, uses no information about the problem to guide the search and therefore may not be very efficient.

Informed Search : Also called *heuristic* or *intelligent* search, uses information about the problem to guide the search, usually guesses the distance to a goal state and therefore efficient, but the search may not be always possible.



Search Space

A set of all states , which can be reached, constitute a search space.

This is obtained by applying some combination of operators defining their connectivity.

Example :

Find route from **Start** to **Goal** state.

Consider the vertices as city and the edges as distances.

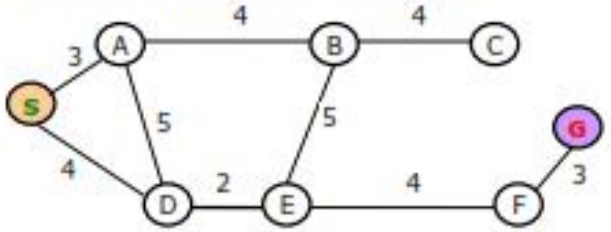


Fig. Search Space

- Initial State **S**
- Goal State **G**
- Nodes represent cities
- Arcs represent distances

Exhaustive Search

Besides Forward and Backward chaining explained, there are many other search strategies used in computational intelligence. Among the most commonly used approaches are :

Breadth-first search (BFS) and *depth-first search (DFS)*.

A search is said to be exhaustive if the search is guaranteed to generate all reachable states (outcomes) before it terminates with failure.

A graphical representation of all possible reachable states and the paths by which they may be reached is called *decision tree*.

Breadth-first search (BFS) : A Search strategy, in which the highest layer of a decision tree is searched completely before proceeding to the next layer is called Breadth-first search (BFS).

- In this strategy, no viable solution is omitted and therefore guarantee that optimal solution is found.
- This strategy is often not feasible when the search space is large.

Depth-first search (DFS) : A search strategy that extends the current path as far as possible before backtracking to the last choice point and trying the next alternative path is called Depth-first search (DFS).

- This strategy does not guarantee that the optimal solution has been found.
- In this strategy, search reaches a satisfactory solution more rapidly than breadth first, an advantage when the search space is large.

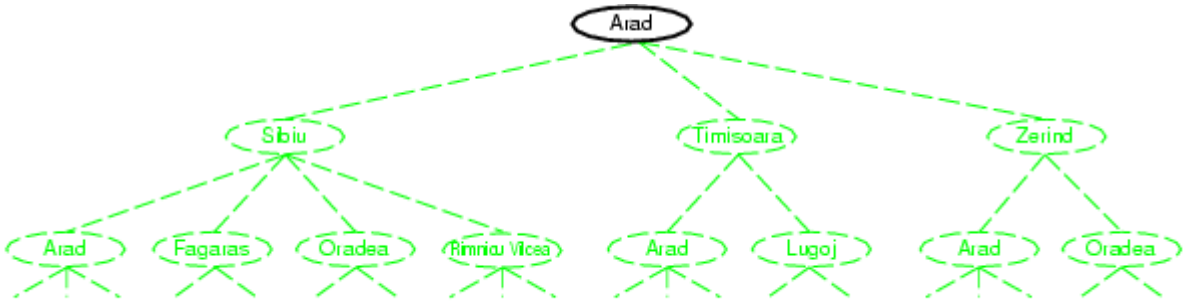
The Breadth-first search (BFS) and depth-first search (DFS) are the foundation for all other search techniques.

Tree search algorithms

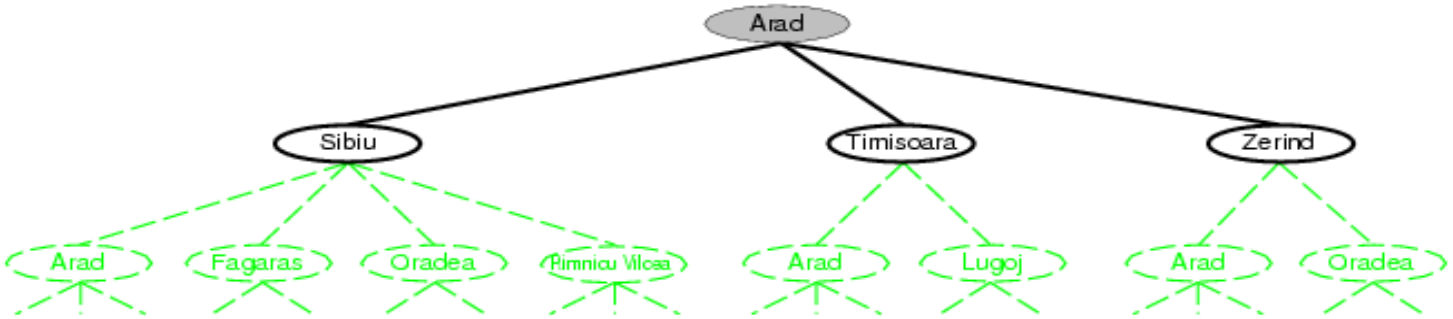
- Basic idea:
  - offline, simulated exploration of state space by generating successors of already-explored states (a.k.a.~expanding states)

```
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
```

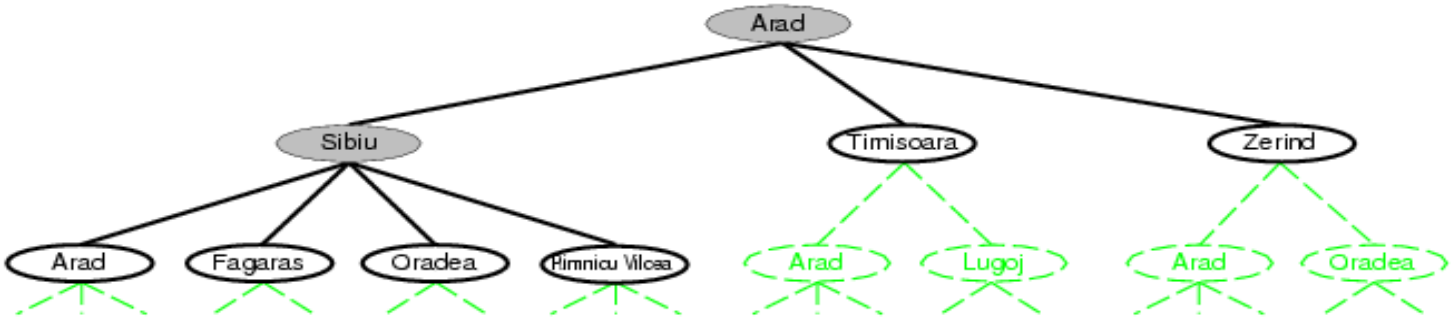
Tree search example



Tree search example



Tree search example



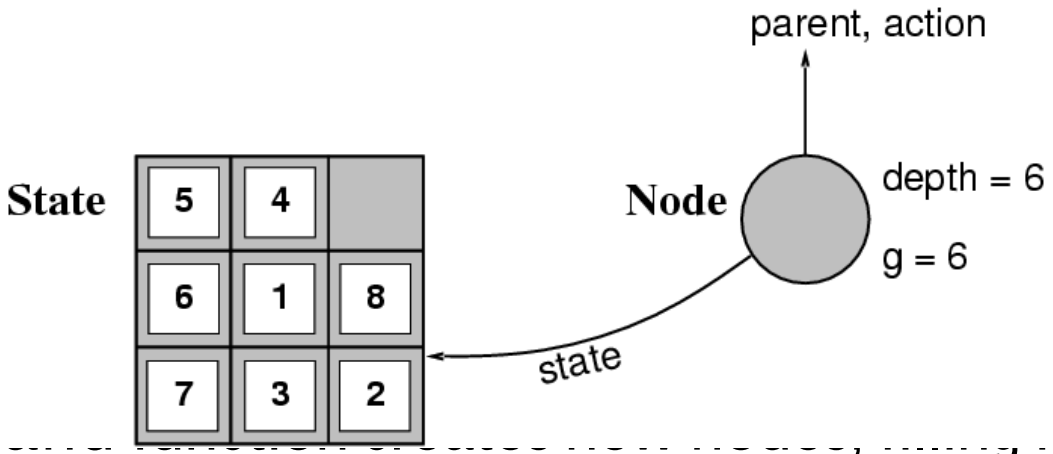
Implementation: general tree search

```
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 SOLUTION(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
```

Implementation: states vs. nodes

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



- The Exp n the various fields and using the `SuccessorFn` of the problem to create the corresponding states.
-

Search strategies

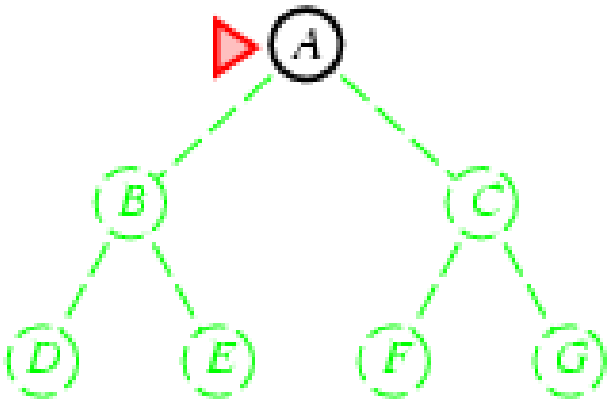
- A search strategy is defined by picking the order of node expansion
- Strategies are evaluated along the following dimensions:
  - completeness: does it always find a solution if one exists?
  - time complexity: number of nodes generated
  - space complexity: maximum number of nodes in memory
  - optimality: does it always find a least-cost solution?
  -
- Time and space complexity are measured in terms of
  - *b:* maximum branching factor of the search tree
  - *d:* depth of the least-cost solution
  - *m*: maximum depth of the state space (may be ∞)
  -

Uninformed search strategies

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

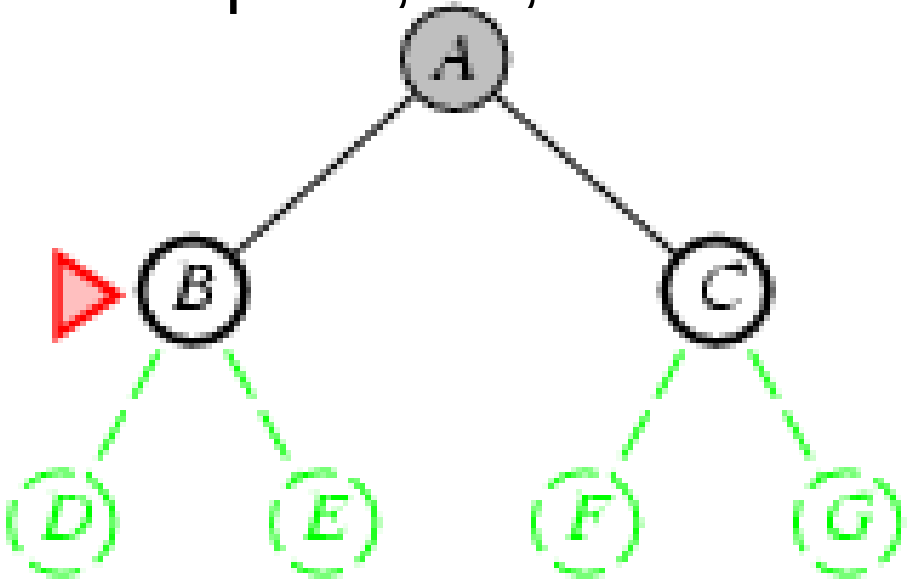
Breadth-first search

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



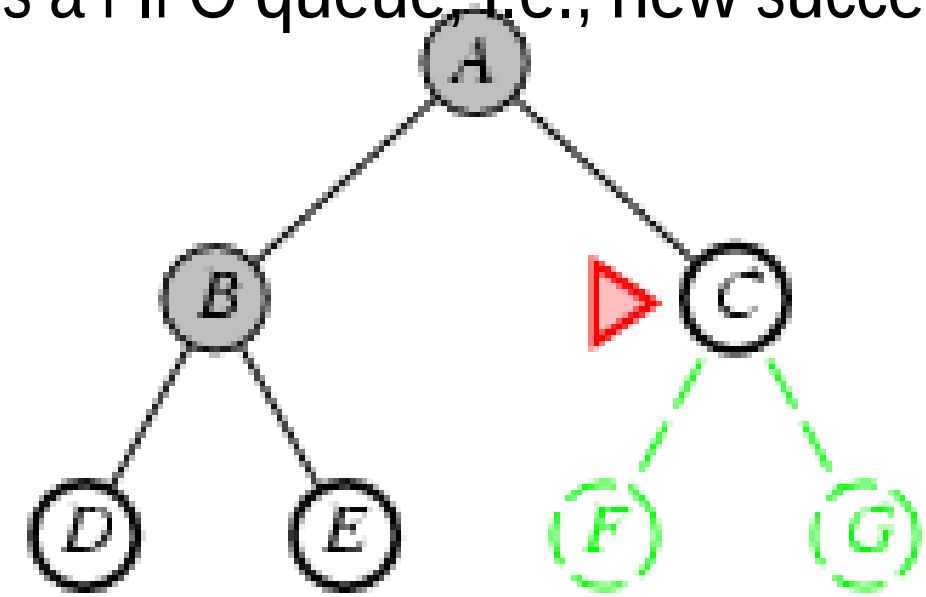
Breadth-first search

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



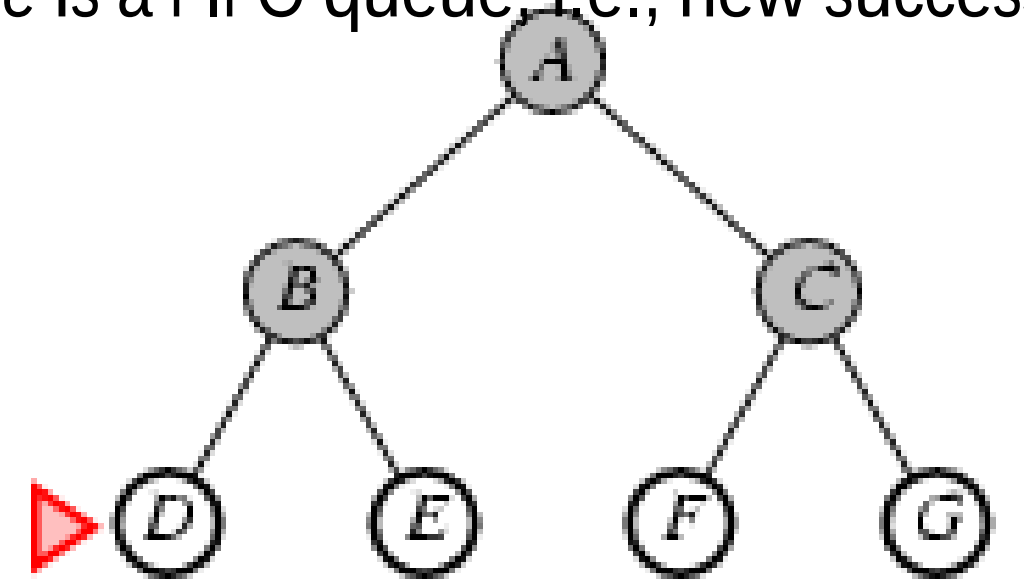
Breadth-first search

- Expand shallowest unexpanded node
-
- Implementation:
  - *fringe* is a FIFO queue, i.e., 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
  -



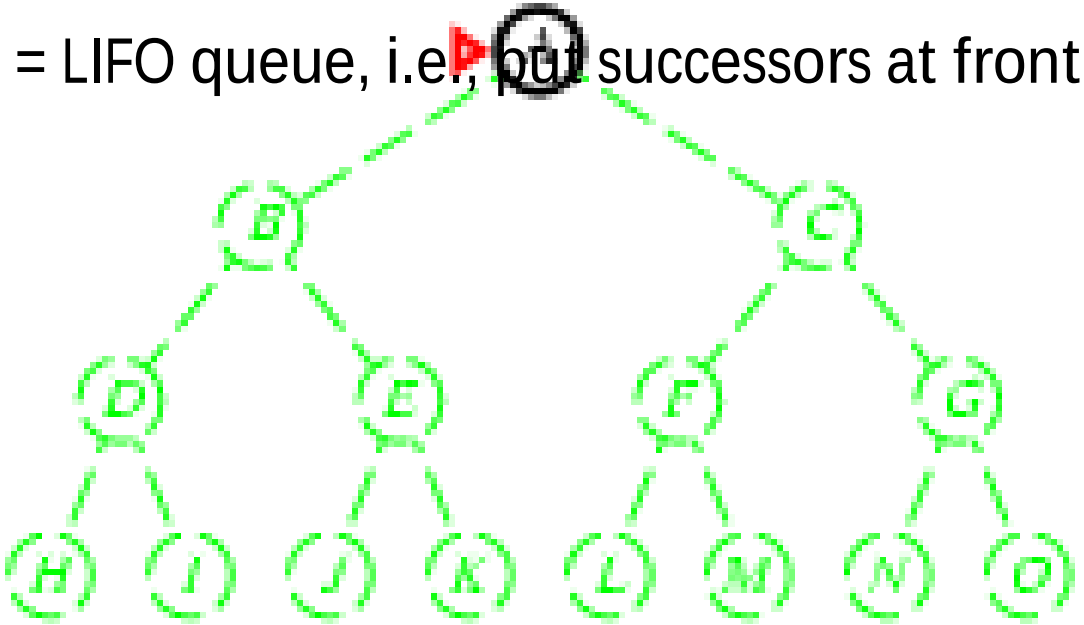
Properties of breadth-first search

- Complete? Yes (if *b* is finite)
-
- Time? $1+b+b^2+b^3+\ldots+b^d + b(b^d-1) = O(b^{d+1})$
-
- Space? $O(b^{d+1})$ (keeps every node in memory)
-
- Optimal? Yes (if cost = 1 per step)
-

- Space is the bigger problem (more than time)
-

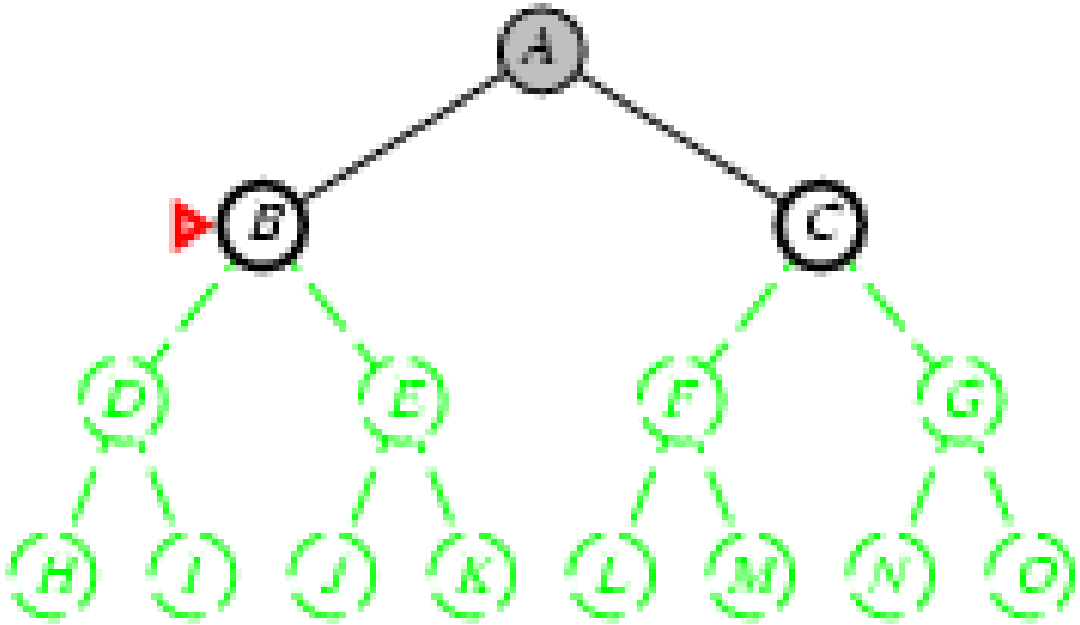
Depth-first search

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



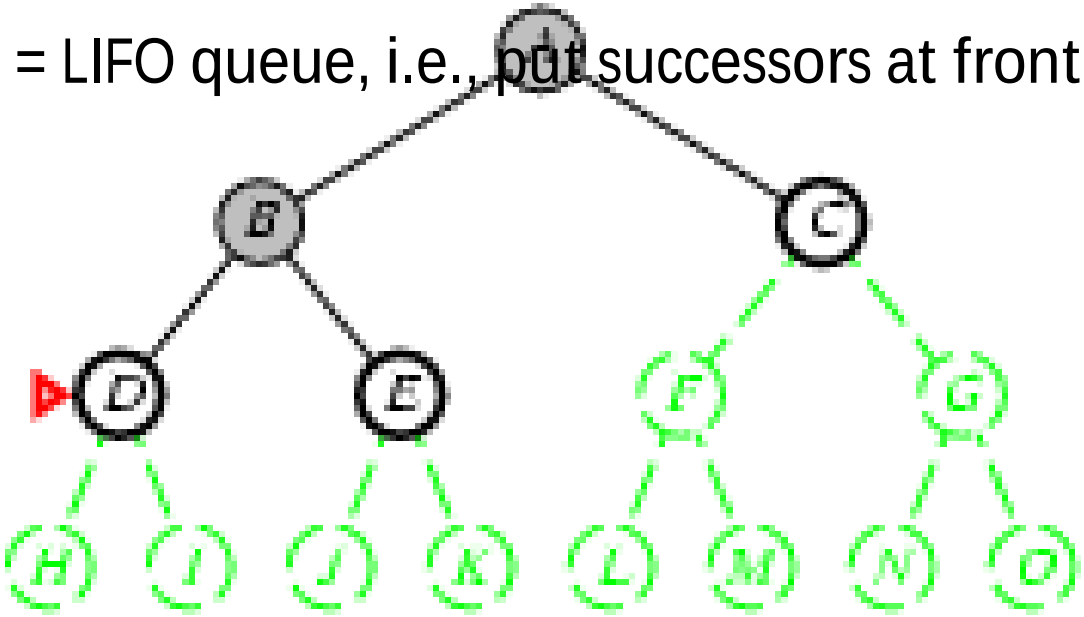
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe*
  -



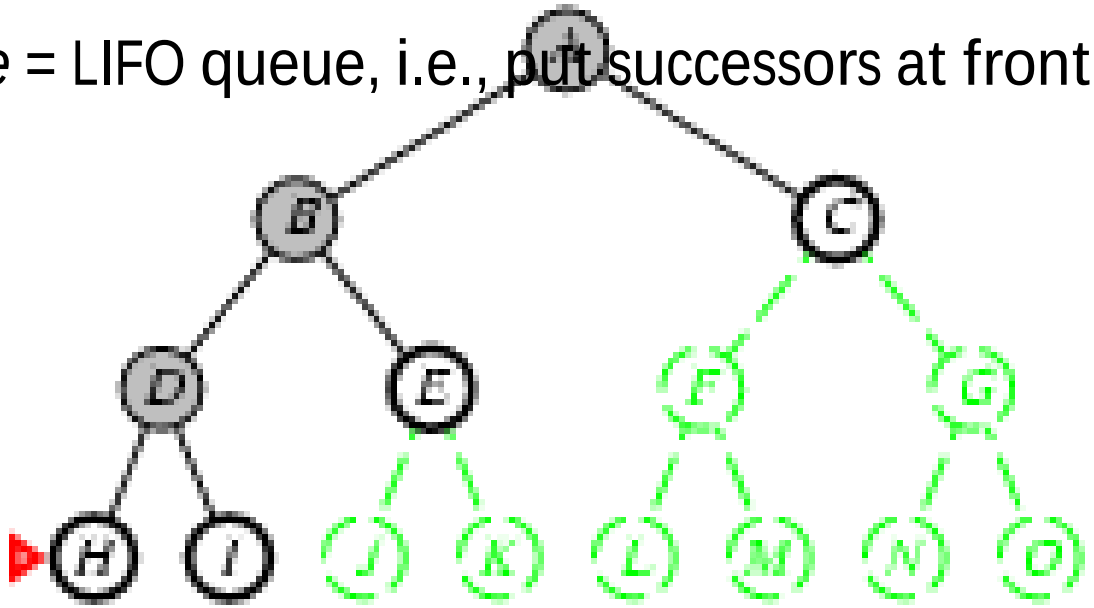
Depth-first search

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



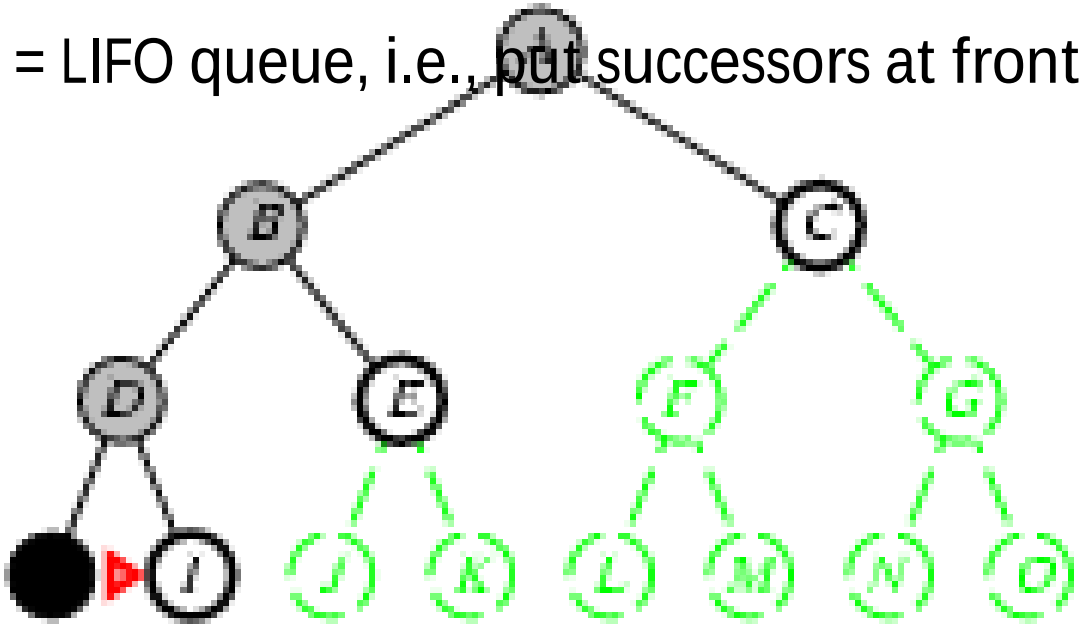
Depth-first search

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



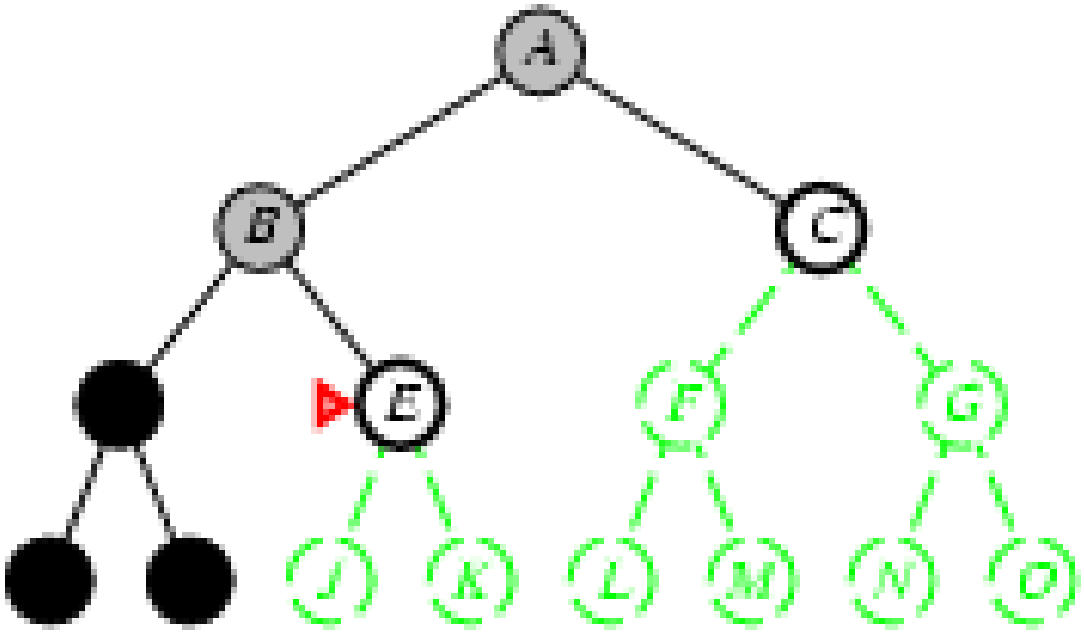
Depth-first search

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



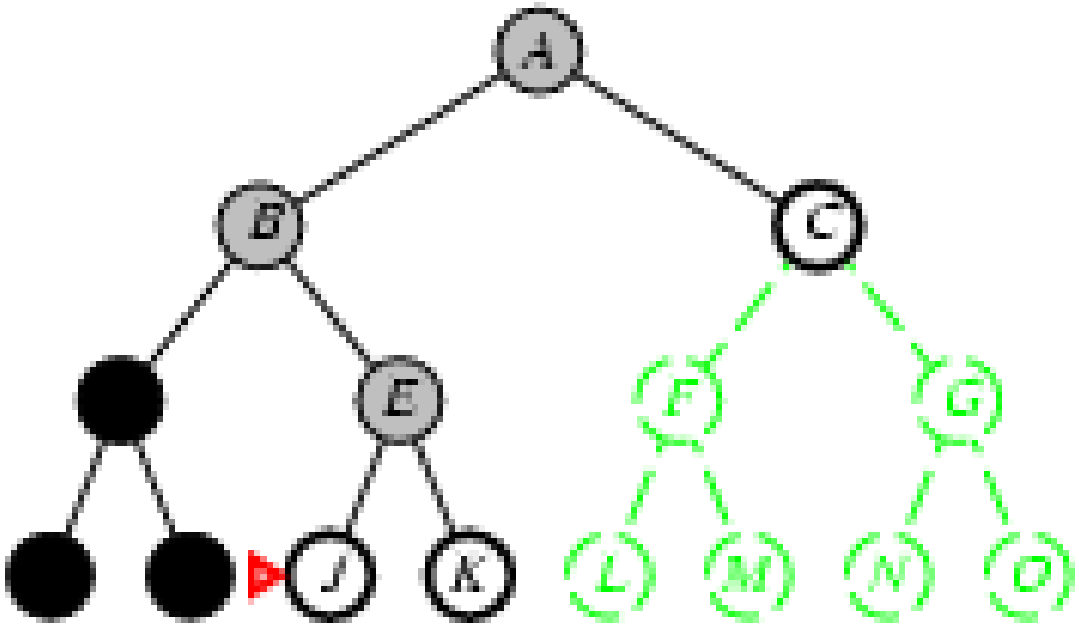
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe*
  -



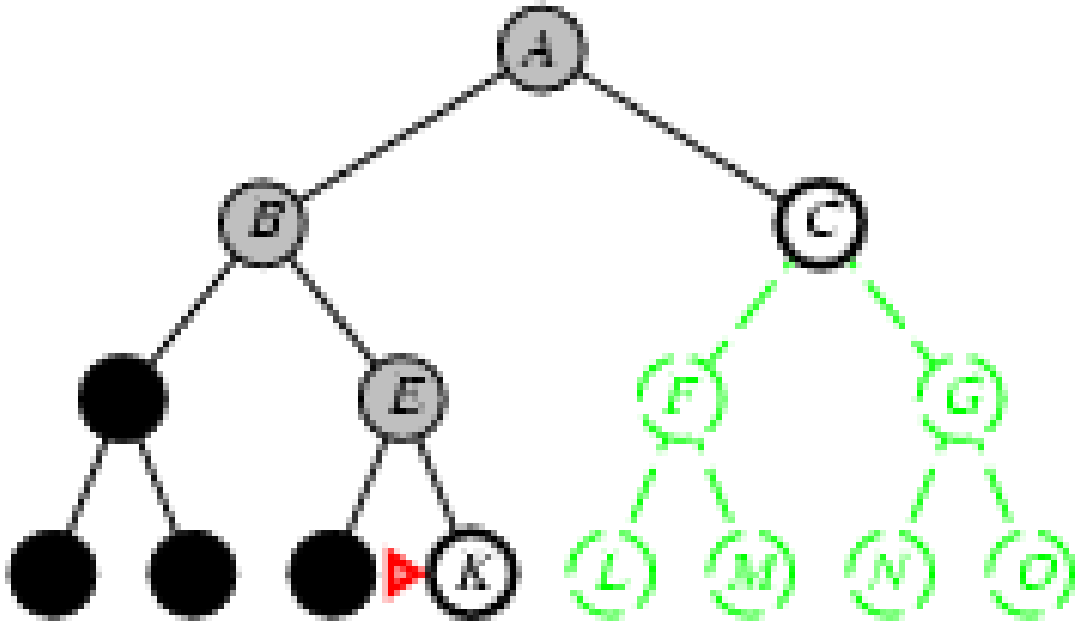
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe*
  -



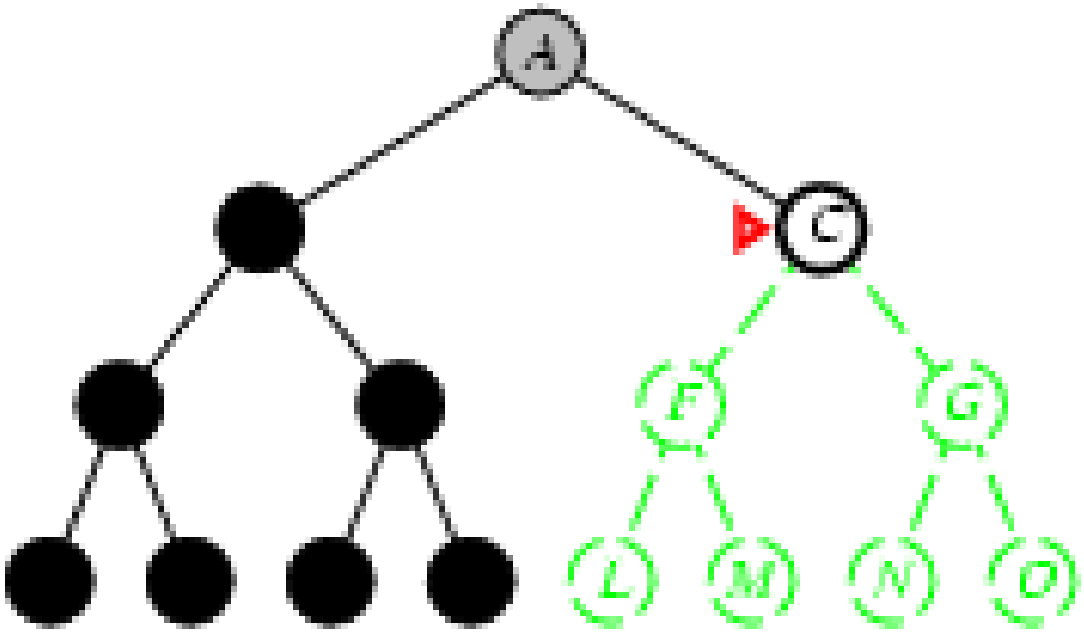
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe*
  -



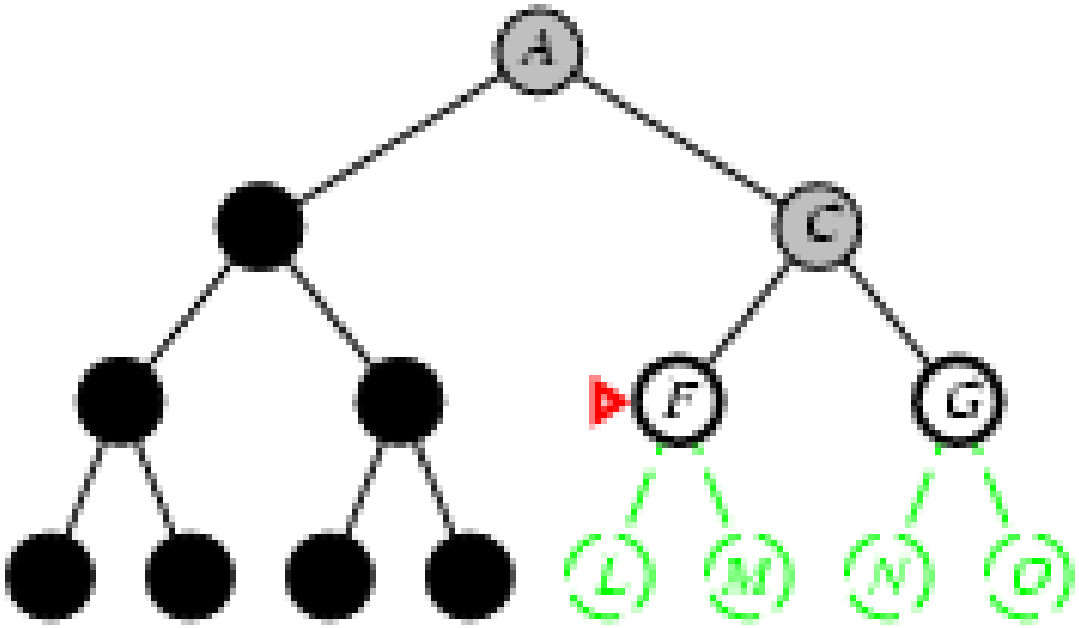
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe*
  -



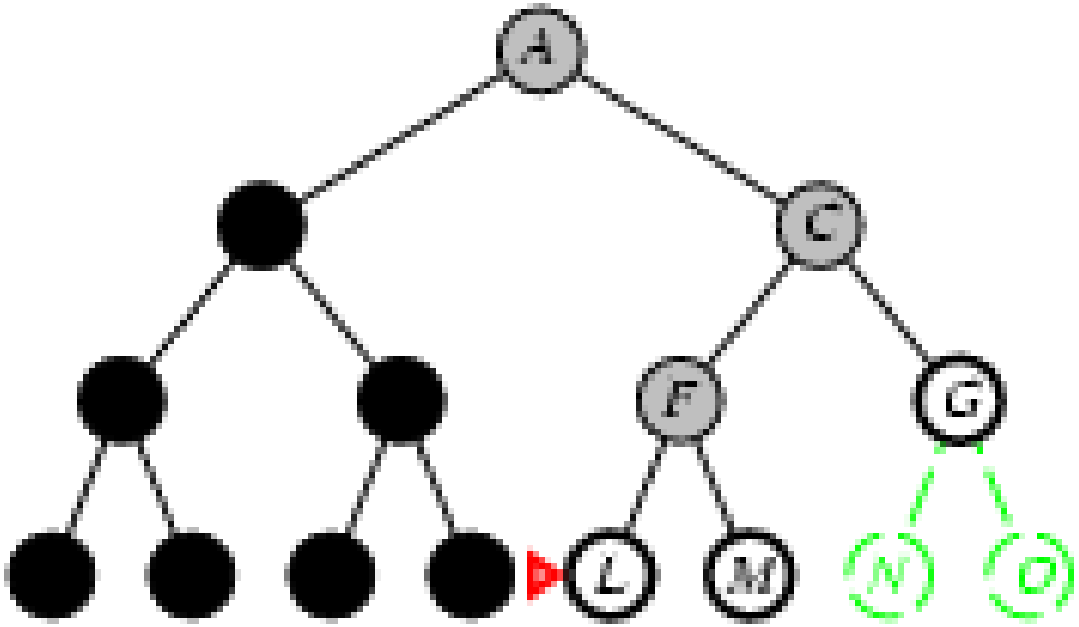
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe*
  -



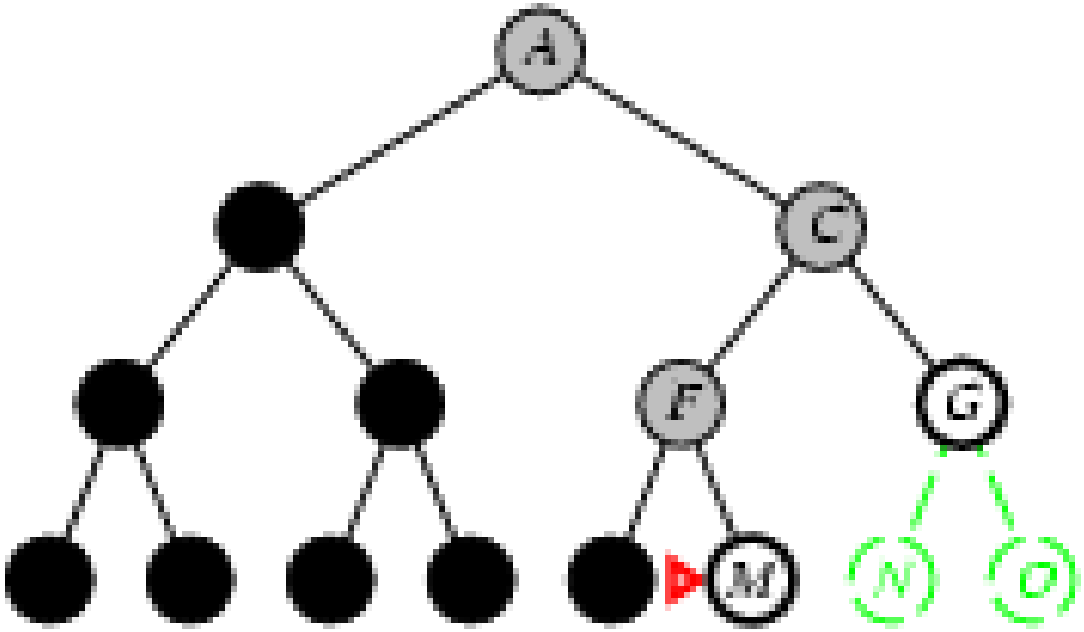
Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe*
  -



Depth-first search

- Expand deepest unexpanded node
-
- Implementation:
  - *fringe*
  -



Properties of depth-first search

- Complete? No: fails in infinite-depth spaces, spaces with loops
  - Modify to avoid repeated states along path
  -
    → complete in finite spaces
- Time? $O(b^m)$: terrible if m is much larger than d
  - but if solutions are dense, may be much faster than breadth-first
  -
- Space? $O(bm)$, i.e., linear space!
-
- Optimal? No
-

Depth – Limited search

- Alleviating the embarrassing failure of depth-first search in infinite state spaces by supplying depth-first search with a predetermined depth-limit of *l*.
- Nodes at depth *l* are treated as if they have no successors.
- Unfortunately, it also introduces an additional source of incompleteness if we choose $l<d$
- It will also be non-optimal if we choose $l>d$
- ts time complexity is $O(b^l)$and its space complexity is $O(bl)$.
- Depth-first search can be viewed as a special case of depth-limited search with $l=\infty$.
- depth-limited search has two modes of failure:
    - standard failure - no solution.
    - cutoff failure - no solution within the depth limit

Depth-limited search

= depth-first search with depth limit *l*,

i.e., nodes at depth *l* have no successors

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

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

Iterative deepening search

general strategy, often used in combination with depth-first tree search, that finds the best depth limit. It does this by gradually increasing the limit - first 0, then 1, then 2, and so on - until a goal is found. This will occure when the depth reaches *d,* the depth of the shallowest goal node. The algorithm is shown in Figure DFS-15:

```
function ITERATIVE-DEEPENING-SEARCH( problem) returns a solution, or fail-
ure
    inputs: problem, a problem
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH( problem, depth)
        if result ≠ cutoff then return result
```

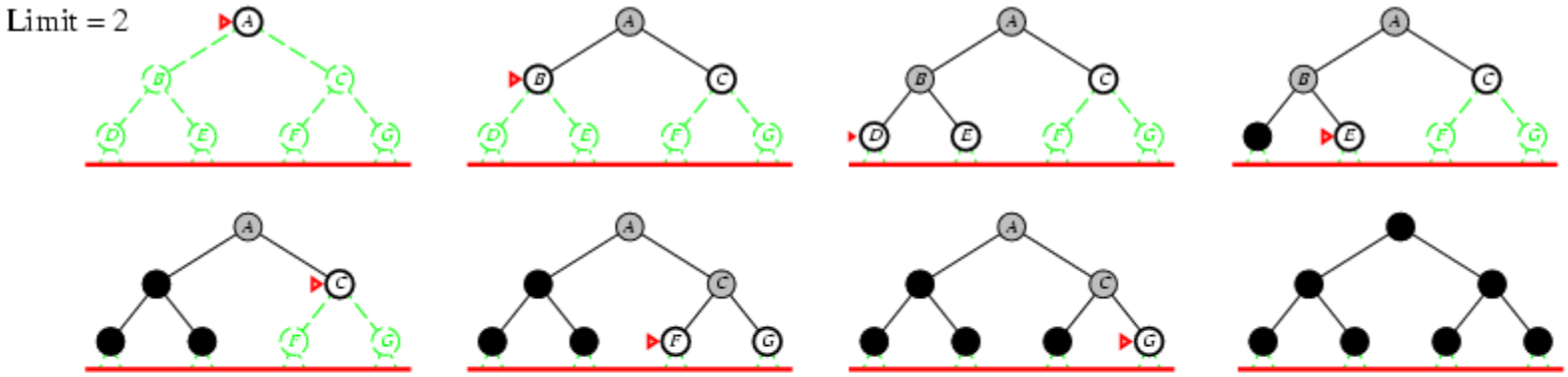
Combines the benefits of depth-first search and breadth-first search.
Memory requirements are: *O(bd)*
N(IterativeDeepeningSearch)=*(d)b + (d-1)b*^{2} *+ ... + (1)b*^{2}
This gives a time complexity of *O(b*^{d}*)*

Iterative deepening search *l* =0

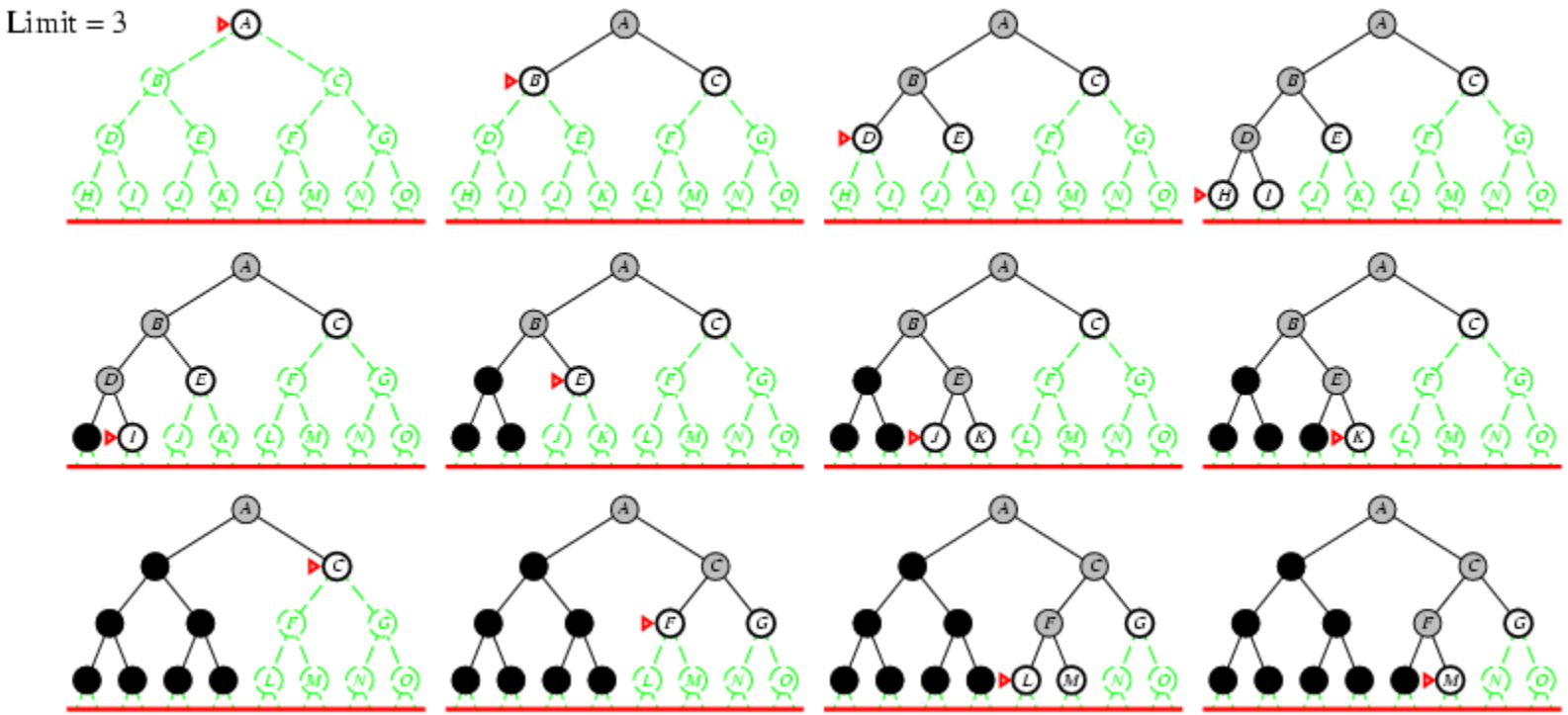
Limit = 0

Iterative deepening search *l* =1

Iterative deepening search *l* =2



Iterative deepening search *l* =3



Iterative deepening search

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

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

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

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

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

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

Properties of iterative deepening search

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

Summary of algorithms

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening
Complete?	Yes	Yes	No	No	Yes
Time	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(b^m)$	$O(b^l)$	$O(b^d)$
Space	$O(b^{d+1})$	$O(b^{\lceil C^*/\epsilon \rceil})$	$O(bm)$	$O(bl)$	$O(bd)$
Optimal?	Yes	Yes	No	No	Yes

Graph search

```
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 SOLUTION(node)
        if STATE[node] is not in closed then
            add STATE[node] to closed
            fringe ← INSERTALL(EXPAND(node, problem), fringe)
```

Summary

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