Course 16 :198 :520 : Introduction To Artificial Intelligence
Lecture 2

Solving Problems by Searching

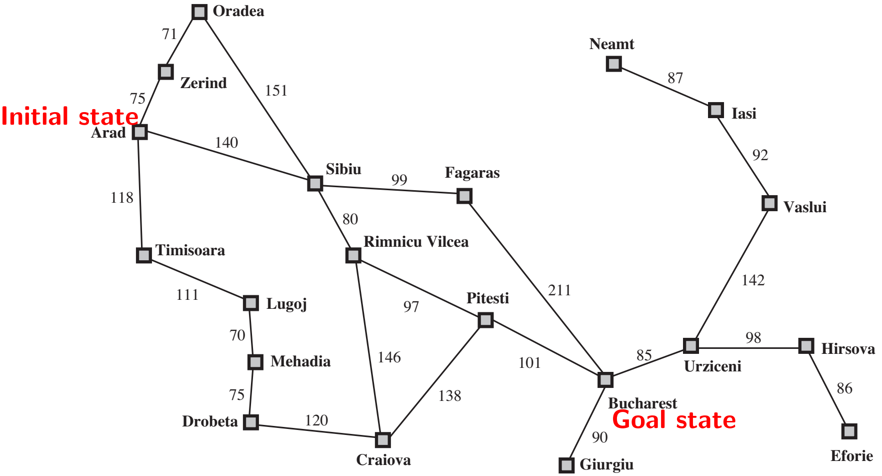
Abdeslam Boularias

Friday, September 9, 2016

RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

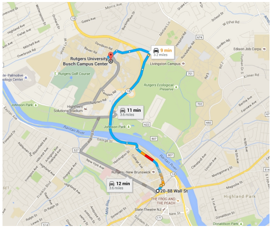
Solving Problems by Searching (Planning agents)

How can an agent find a sequence of actions that achieves its goals when no single action will do ?



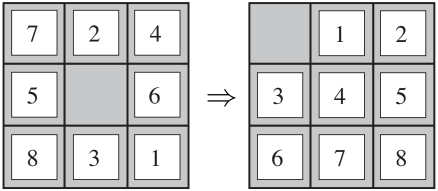
Route-finding problem using a simplified road map of Romania

Solving Problems by Searching : Examples



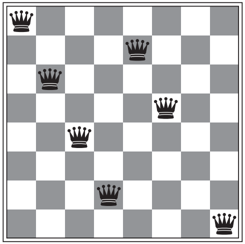
Route planning

Robot navigation
(Copyright Wikimedia Commons)



Start State Goal State

Sliding-block puzzle



Eight queens puzzle

A well-defined problem is described by

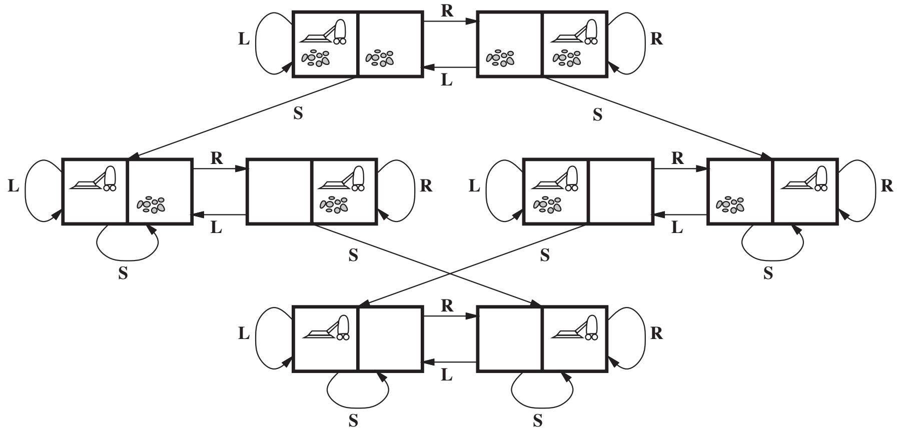
- **States** : set of all possible situations (configurations, positions). Example : locations on a map.
- **Initial state** : starting state. Example : *In(Busch campus)*
- **Goal states** : destinations, final positions. Example : *In(New Brunswick)*. The goal state can be implicitly defined by test, such as in the *eight queens puzzle*.
- **Actions** : set of everything an agent can do to change its current state. Examples : *Go(Livingston campus)*
- **Transition model** : the effect of each action. Example : $\mathrm{RESULT(In(Busch\ campus),Go(Livingston\ campus)) = In(Livingston\ campus)}$
- **Path cost** : function that assigns a numeric cost to each path. A path is a sequence of states connected by a sequence of actions. Example : *length in kilometers.*

We make the following assumptions

- The cost of a path is the sum of the costs of the individual actions along the path. The **step cost** of action a in state s to reach state s' is denoted by $c(s, a, s')$.
- The environment is known and fully observable : the agent knows exactly where it is and how its actions will change its state.
- Actions are deterministic : each action has exactly one outcome.

Under these assumptions, the solution to any problem is a fixed sequence of actions.
Under these assumptions, the agent can ignore its observations, why ?
This is known as open-loop control (in contrast with closed-loop control where actions are chosen according to observations).

Toy example : the vaccum world

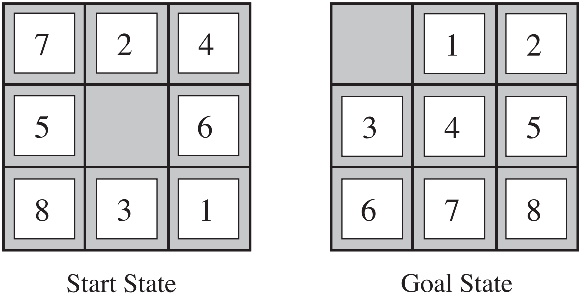


The state space for the vacuum world.
Links denote actions : L = *Left*, R = *Right*, S = *Suck*.

Toy problem : the vaccum world

- **States** : (agent location, dirt location). There are two locations, each of which may or may not contain dirt. Thus, there are $2 \times 2^2 = 8$ possible world states. In an environment with n locations, there are $n2^n$ states.
- **Initial state** : any state can be a starting state.
- **Goal states** : all the squares are clean (two goal states).
- **Actions** : Suck, move left, move right.
- **Transition model** : Actions have their expected effects, except that moving left in the left square or right in the right square has no effect. Also, sucking in an already clean square has no effect.
- **Path cost** : Each action has a cost of 1 (or a value related to time or energy).

Toy example : the sliding-block puzzle

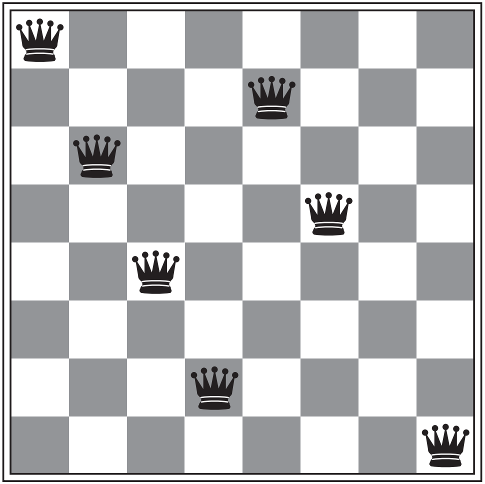


Instance of the 8-puzzle

Toy problem : the sliding-block puzzle

- **States** : location of each of the eight tiles and the blank. There are $9!$ states.
- **Initial state** : Any state.
- **Goal state** : Any chosen configuration (only half of the states are accessible from a given initial state).
- **Actions** : Move the blank space to *left*, *right*,*up* or *down*.
- **Transition model** : deterministic, depends on the state and action.
- **Path cost** : Each action has a cost of 1. We try to minimize the number of steps needed to solve the puzzle.

Toy example : the eight queens puzzle



Attempt to solve the eight queens puzzle

Toy problem : the eight queens puzzle

- **States** : Placements of up to eight queens on a chessboard. There are $P(64, 8)$ states. $P(64, 8) = 64 \times 63 \times 62 \times 61 \times 60 \times 59 \times 58 \times 57 \approx 1.8 \times 10^{14}$.
- **Initial state** : Empty board.
- **Goal state** : 8 queens on the board and none is attacked.
- **Actions** : Put a queen in any empty square.
- **Transition model** : adds a queen to the specified square.
- **Path cost** : Each action has a cost of 0. Each path to the goal will have exactly 8 steps, so we do not care about minimizing the number of steps.

Toy problem : the eight queens puzzle

- **Actions** : Put a queen in an empty square that is not being attacked, and that is on the leftmost column that is empty. The first queen is placed anywhere on the first column.
- **States** : Placements of up to eight queens on a chessboard such that no queen is attacking another. This more clever representation reduces the number of possible states from 1.8×10^{14} to 2057.

Toy problem : Donald Knuth's conjecture (1964)

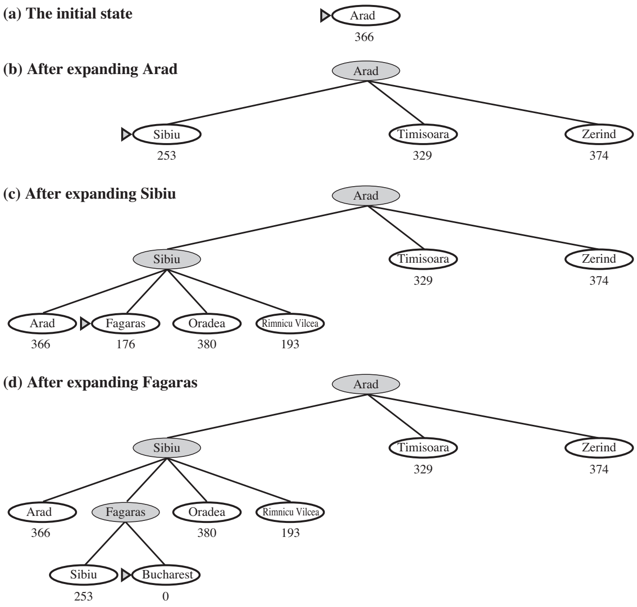
One can start at 3 and reach any integer by iterating factorial, sqrt, and floor.
Example : $\left\lfloor \sqrt{\sqrt{(3!)!}} \right\rfloor = 5$

- **States** : set of natural numbers $\mathbb{N}$
- **Initial state** : number 3.
- **Goal state** : any given natural number.
- **Actions** : floor, square root, and factorial operations.
- **Transition model** : result of the operation.
- **Path cost** : Each action has a cost of 1.

Search tree

- A solution is an action sequence (path).
- Search algorithms work by considering various possible action sequences.
- Starting from the initial state, we consider all the different actions that we can execute.
- Each action leads to a new state.
- For each new state, we consider all the possible actions and the resulting states.
- This process forms a tree of states, the initial state is the root of the tree.
- The goal state (or states) is somewhere in the tree.
- The process of finding the goal state is called *tree searching*.
- The different searching methods are called *tree search algorithms*.

Search tree



Search tree for finding a route from Arad to Bucharest

Tree searching

- A node is said to be **expanded** when its children are added to the search.
- The set of nodes that have been added to the search but have not yet been expanded (the leaves) is called the **open list**, the **fringe**, or the **frontier**.

```
function TREE-SEARCH(problem) returns a solution, or failure
  initialize the frontier using the initial state of problem
  loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      expand the chosen node, adding the resulting nodes to the frontier
```

Generic algorithm for tree searching

Search algorithms vary according to how they choose which state to expand next (search strategy).

Searching for solutions

- Notice the repeated states in the search tree in the example.
- Loopy paths can lead to infinite search trees.
- The set of expanded nodes is called the **closed** (or **explored**) **list**.
- Graph searching algorithms avoid redundant paths by keeping track of the explored nodes.

```
function GRAPH-SEARCH(problem) returns a solution, or failure
   initialize the frontier using the initial state of problem
   initialize the explored set to be empty
   loop do
      if the frontier is empty then return failure
      choose a leaf node and remove it from the frontier
      if the node contains a goal state then return the corresponding solution
      add the node to the explored set
      expand the chosen node, adding the resulting nodes to the frontier
         only if not in the frontier or explored set
```

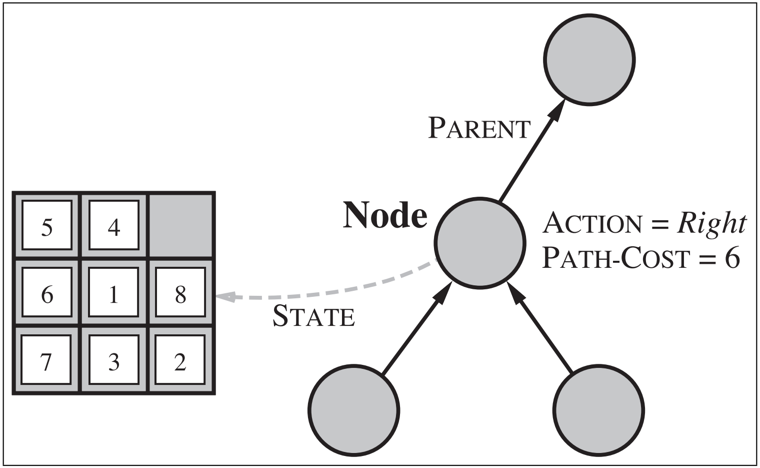
Generic algorithm for graph searching

Implementing a search tree algorithm

In a search tree, each node n is represented by a data structure that contains the following fields

- **n.State** : the state that the node corresponds to.
  Example : n.State = In(Busch campus)
- **n.Parent** : the parent of node n.
- **n.Action** : the action that lead to node n (from n.Parent).
- **n.Path-cost** : the cost path cost from the root to node n.
- **n.Depth** : the path cost from the root to the current node

Data structure used for nodes in a search tree



Implementing a search tree algorithm

The nodes are kept in a queue. The operations on a queue are as follows :

- **EMPTY ?(queue)** : returns true of the queue is empty.
- **POP(queue)** : removes the first element and returns it.
- **INSERT(element,queue)** : insert an element and returns the resulting queue.
- **INITIALIZE(element)** : returns a queue that contains element.

Queues are characterized by the order in which they store the inserted nodes.

- **FIFO queue** : first-in, first-out
- **LIFO queue** : last-in, first-out
- **Priority queue** : pops the element with the highest priority according to some function

Different queue structures give different search algorithms.

Search Algorithm Comparison Criteria And Complexity Parameters

There are four criteria comparing the various search tree algorithms :

- **Completeness** : Is the algorithm guaranteed to find a solution when there is one ?
- **Optimality** : Does the strategy find the optimal solution ?
- **Time complexity** : How long does it take to find a solution ?
- **Space complexity** : How much memory is needed to perform the search ?

The following parameters are used to calculate and compare the performance of an algorithm :

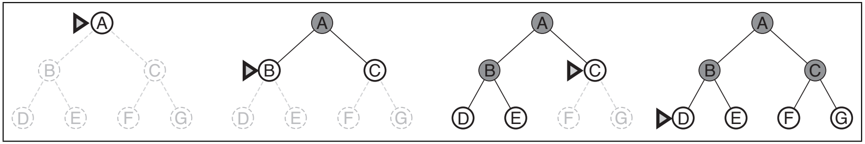
- **b** : the maximum branching factor (number of children per node)
- **d** : the depth of the shallowest goal state
- **m** : maximum length of path in the tree

Search Algorithms

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening depth-first search
- Bidirectional search

Breadth-First Search (BFS)

Breadth-First Search expands all the nodes first before expanding the nodes at the next level.



Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

Breadth-first search always has the shallowest path to every node on the frontier.

Breadth-First Search (BFS)

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node $\leftarrow$ a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier $\leftarrow$ a FIFO queue with node as the only element
  explored $\leftarrow$ an empty set
  loop do
      if EMPTY?(frontier) then return failure
      node $\leftarrow$ POP(frontier)   /* chooses the shallowest node in frontier */
      add node.STATE to explored
      for each action in problem.ACTIONS(node.STATE) do
          child $\leftarrow$ CHILD-NODE(problem, node, action)
          if child.STATE is not in explored or frontier then
              if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
              frontier $\leftarrow$ INSERT(child, frontier)
```

Properties of Breadth-First Search

- **Frontier queue** : First In First Out (FIFO)
- **Completeness** : Complet if branching factor b is finite
- **Optimality** : Optimal only if all the costs of the edges are equal (shallowest path in the tree is not always the shortest in the problem).
- **Time complexity** : $O(\sum_{i=0}^{d-1} b^i) = O(b^d)$
- **Space complexity** : $O(\sum_{i=0}^{d-1} b^i) = O(b^d)$

Scalability of Breadth-First Search

Exponential complexity $O(b^d)$ is scary !

Depth	Nodes	Time	Memory
2	110	.11 milliseconds	107 kilobytes
4	11,110	11 milliseconds	10.6 megabytes
6	10^6	1.1 seconds	1 gigabyte
8	10^8	2 minutes	103 gigabytes
10	10^{10}	3 hours	10 terabytes
12	10^{12}	13 days	1 petabyte
14	10^{14}	3.5 years	99 petabytes
16	10^{16}	350 years	10 exabytes

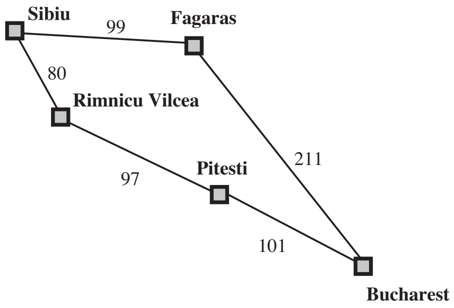
Time and memory requirements for breadth-first search. The numbers shown assume branching factor b = 10 ; 1 million nodes/second ; 1000 bytes/node.

Bad news : It doesn't get much better with faster computers.

Uniform-cost Search

- Breadth-First Search is optimal only when all step costs are equal.
- To solve this problem, Uniform-cost Search expands the node n with the lowest path cost $g(n)$, instead of the shallowest node.
- The queue of the frontier is ordered by path cost.
- In BFS all the nodes in the queue have the same path cost, the goal test is performed when a node is generated.
- In Uniform-cost Search, the nodes in the queue have different path costs, the goal test is performed when a node is expanded.

Uniform-cost Search

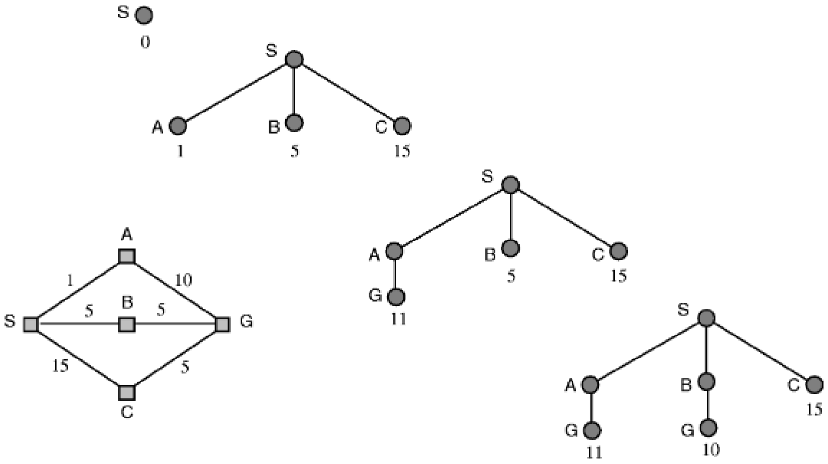


Part of Romania, selected to illustrate uniform-cost search

Nodes are expanded in the following order :

- Sibiu,
- Rimnicu,
- Fagaras (goal node Bucharest generated here, but this is not the optimal path yet),
- Pitesti,
- Bucharest (optimal path found).

Uniform-cost Search



Uniform-cost Search

```
function UNIFORM-COST-SEARCH($problem$) returns a solution, or failure
    $node$ ← a node with STATE = $problem$.INITIAL-STATE, PATH-COST = 0
    $frontier$ ← a priority queue ordered by PATH-COST, with $node$ as the only element
    $explored$ ← an empty set
    loop do
        if EMPTY?($frontier$) then return failure
        $node$ ← POP($frontier$)   /* chooses the lowest-cost node in $frontier$ */
        if $problem$.GOAL-TEST($node$.STATE) then return SOLUTION($node$)
        add $node$.STATE to $explored$
        for each $action$ in $problem$.ACTIONS($node$.STATE) do
            $child$ ← CHILD-NODE($problem$, $node$, $action$)
            if $child$.STATE is not in $explored$ or $frontier$ then
                $frontier$ ← INSERT($child$, $frontier$)
            else if $child$.STATE is in $frontier$ with higher PATH-COST then
                replace that $frontier$ node with $child$
```

Dijkstra's algorithm (a slightly different variant of uniform-cost search)

Dijkstra uses a list of unvisited vertices (initially contains all vertices), and a table that keeps track the shortest distance to each vertex so far (initialized to infinity) instead of the open and closed lists.

```
function Dijkstra(Graph, source):

    create vertex set Q

    for each vertex v in Graph:             // Initialization
        dist[v] ← INFINITY                   // Unknown distance from source to v
        prev[v] ← UNDEFINED                  // Previous node in optimal path from source
        add v to Q                          // All nodes initially in Q (unvisited nodes)

    dist[source] ← 0                         // Distance from source to source

    while Q is not empty:
        u ← vertex in Q with min dist[u]     // Source node will be selected first
        remove u from Q

        for each neighbor v of u:           // where v is still in Q.
            alt ← dist[u] + length(u, v)
            if alt < dist[v]:               // A shorter path to v has been found
                dist[v] ← alt
                prev[v] ← u

    return dist[], prev[]
```

Inconvenient : the graph could be too large to store in memory !

Properties of Uniform-cost Search

- **Frontier queue** : Ordered by path cost
- **Completeness** : Complet if the branching factor is finite
- **Optimality** : Optimal. Whenever uniform-cost search selects a node n for expansion, the optimal path to that node has been found.
- **Time complexity** : $O(b^{1+\lfloor C^*/\epsilon \rfloor})$, where C^* is the path cost of the optimal path and ϵ is the minimum step cost.
- **Space complexity** : $O(b^{1+\lfloor C^*/\epsilon \rfloor})$

$O(b^{1+\lfloor C^*/\epsilon \rfloor})$ can be much worse than $O(b^d)$ (the complexity of BFS), because Uniform-cost Search can explore a large number of small steps before trying longer steps that may lead immediately to the goal.

Properties of Uniform-cost Search

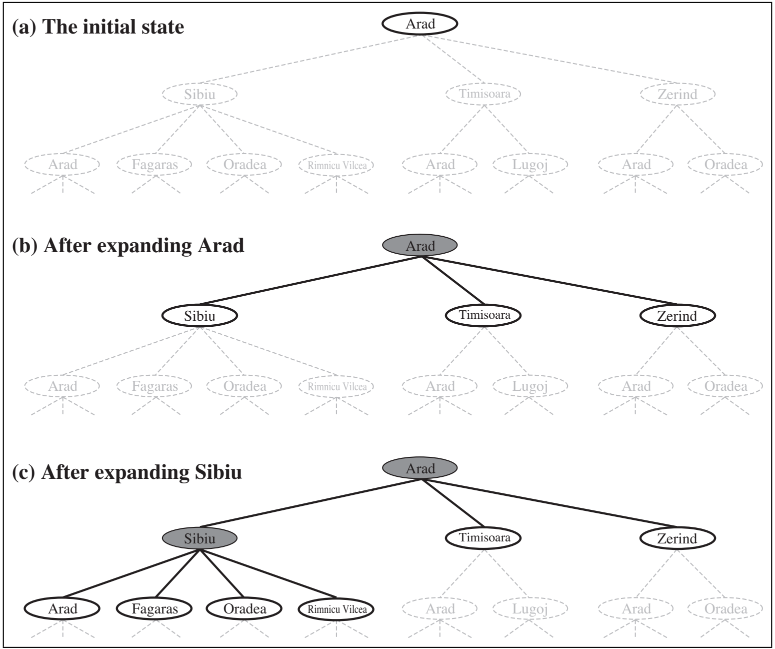
- Another worst-case bound on the time and space complexities can be obtained by considering the maximum number of operations on vertices (states) in the graph.
- Let V be the set of vertices in the graph, and let $|V|$ denote the total number of vertices.
- Each vertex is expanded only once at most, then there are $|V|$ expansions at most.
- Every time a vertex is expanded, at most $|V|$ children are pushed to the open list after updating their distances, because a vertex cannot be connected to more than $|V|$.
- Therefore, there are $|V||V|$ operations at most.

The time and space complexity of uniform-cost search is $O(\min\{|V|^2, b^{1+\lfloor C^*/\epsilon \rfloor}\})$

Depth-First Search (DFS)

- Depth-first search always expands the deepest node in the current frontier of the search tree.
- The frontier is stored in a LIFO (Last In First Out) queue.

Depth-First Search (DFS)



Properties of Depth-First Search (DFS)

- DFS can run into an infinite loop when repeated states are allowed.
- In infinite state spaces, DFS fails if an infinite non-goal path is encountered. Example : Knuth's problem.
- DFS is non-optimal, it returns the first path that contains the goal.
- Time complexity : $O(b^m)$ where m is the maximum depth of any node. This is much worse than BFS complexity $O(b^d)$ where d is the depth of the shallowest node.
- Space complexity : $O(bm)$, and only $O(m)$ using backtracking (where we do not need to generate all the successors of each node).

Depth-limited Search

- To avoid failure of DFS in infinite state spaces, we supply depth-first search with a predetermined depth limit l.
- This introduces a source of incompleteness if $l < d$.
- Time complexity : $O(b^l)$
- Space complexity : $O(bl)$
- Depth limit l is chosen depending on the problem. Example : we know that in the map of Romania there are 20 cities, therefore $l = 19$ makes sense. Moreover, any city can be reached from any other city in at most 9 steps. This number is known as the diameter of the state space. Therefore, $l = 9$ is a better choice.

Recursive Depth-limited Search

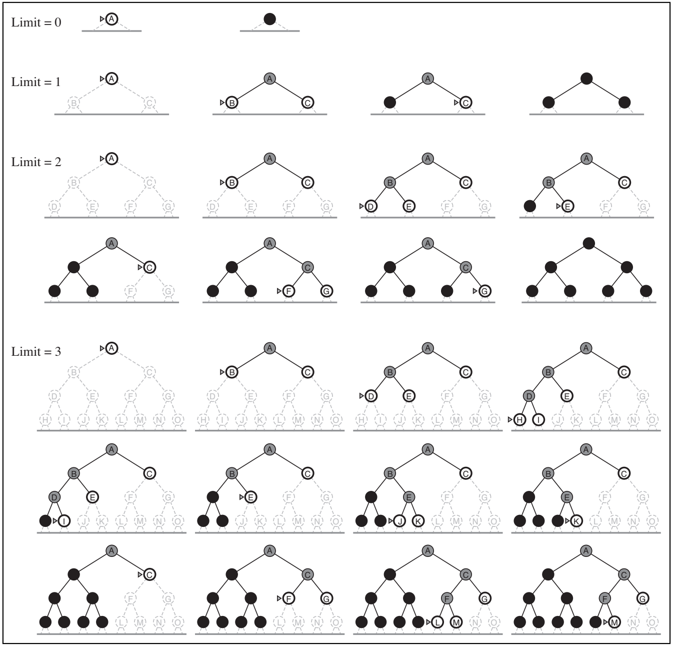
```
function DEPTH-LIMITED-SEARCH($problem, limit$) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE($problem$.INITIAL-STATE), $problem, limit$)

function RECURSIVE-DLS($node, problem, limit$) returns a solution, or failure/cutoff
  if $problem$.GOAL-TEST($node$.STATE) then return SOLUTION($node$)
  else if $limit$ = 0 then return $cutoff$
  else
      $cutoff\_occurred?$ ← false
      for each $action$ in $problem$.ACTIONS($node$.STATE) do
          $child$ ← CHILD-NODE($problem, node, action$)
          $result$ ← RECURSIVE-DLS($child, problem, limit - 1$)
          if $result$ = $cutoff$ then $cutoff\_occurred?$ ← true
          else if $result \neq failure$ then return $result$
      if $cutoff\_occurred?$ then return $cutoff$ else return $failure$
```

Iterative Deepening Depth-First Search

- To solve the problem of choosing the right depth l, one can start with $l = 0$ and iteratively increase it during the search.
- Iterative Deepening DFS runs repeatedly through the tree, increasing the depth limit with each iteration until it reaches the depth of the shallowest goal.
- Iterative Deepening DFS has the space complexity of DFS and the completeness properties of BFS.

Iterative Deepening Depth-First Search



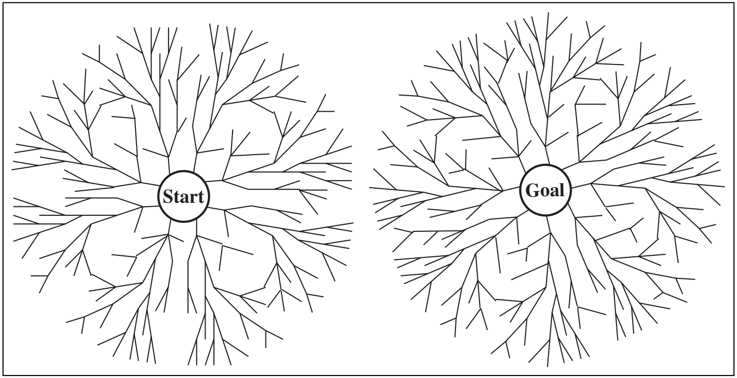
Iterative Deepening Depth-First Search

Properties :

- Complete : by iteratively increasing the limit, at some point the algorithm reaches a goal node and the algorithm never searches an infinite path.
- Time complexity : $O(b^d)$, the number of nodes using depth limit d is equal to the number of nodes using all depths $< d$.
- Space complexity : $O(bd)$, the same as in DFS.
- Optimality : Yes, if all the edges have the same cost.

Bidirectional Search

- Runs two simultaneous breadth-first searches : one forward from the initial state, and one backward from the goal.
- Stop when the two meet in the middle.



Bidirectional Search Properties

- Complete : like in BFS.
- Time complexity : $O(2b^{d/2}) = O(b^{d/2})$
- Space complexity : $O(2b^{d/2}) = O(b^{d/2})$
- Optimality : Yes, if all the edges have the same cost, like in BFS.