ARTIFICIAL INTELLIGENCE

SOLVING PROBLEMS BY SEARCHING

Chapter 3

December 8, 2018

PROBLEM SOLVING

We want:

- To automatically solve a problem

We need:

- A representation of the problem
- Algorithms that use some strategy to solve the problem defined in that representation

PROBLEM REPRESENTATION

General:

- **State space:** a problem is divided into a set of resolution steps from the initial state to the goal state
- **Reduction to sub-problems:** a problem is arranged into a hierarchy of sub-problems

Specific:

- Game resolution
- Constraints satisfaction

STATE SPACE SEARCH

- Many problems in AI take the form of ***state-space search***.
- The ***states*** might be legal board configurations in a game, towns and cities in some sort of route map, collections of mathematical propositions, etc.
- The ***state-space*** is the configuration of the possible states and how they connect to each other e.g. the legal moves between states.
- When we don't have an ***algorithm*** which tells us definitively how to negotiate the state-space we need to search the state-space to find an optimal path from a start state to a goal state.
- We can only decide what to do (or where to go), by considering the possible moves from the current state, and trying to look ahead as far as possible. Chess, for example, is a very difficult state-space search problem.

STATES

- A problem is defined by its elements and their relations.
- In each instant of the resolution of a problem, those elements have specific descriptors (How to select them?) and relations.
- A state is a representation of those elements in a given moment.
- Two special states are defined:
  - **Initial state** (starting point)
  - **Final state** (goal state)

STATE SPACE

- The state space is the set of all states reachable from the initial state.
- It forms a graph (or map) in which the nodes are states and the arcs between nodes are actions.
- A path in the state space is a sequence of states connected by a sequence of actions.
- The solution of the problem is part of the map formed by the state space.

PROBLEM SOLUTION

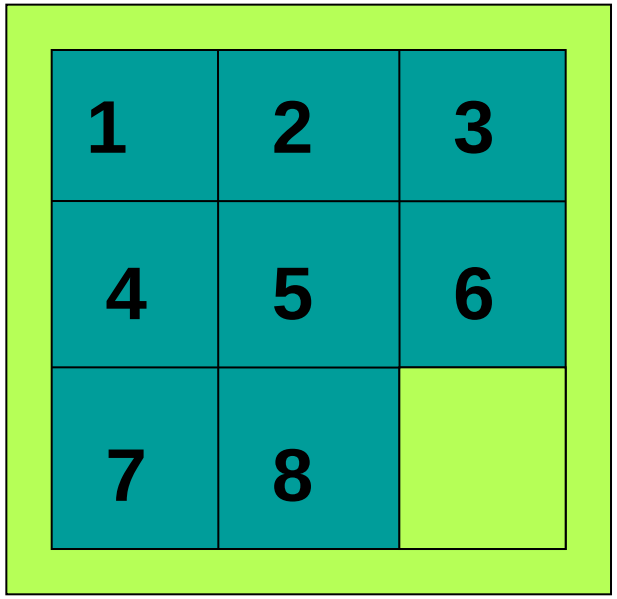
- A solution in the state space is a path from the initial state to a goal state or, sometimes, just a goal state.
- Path/solution cost: function that assigns a numeric cost to each path, the cost of applying the operators to the states
- Solution quality is measured by the path cost function, and an optimal solution has the lowest path cost among all solutions.
- Solutions: any, an optimal one, all. Cost is important depending on the problem and the type of solution sought.

PROBLEM DESCRIPTION

Components:

- State space (explicitly or implicitly defined)
- Initial state
- Goal state (or the conditions it has to fulfill)
- Available actions (operators to change state)
- Restrictions (e.g., cost)
- Elements of the domain which are relevant to the problem (e.g., incomplete knowledge of the starting point)
- Type of solution:
  - Sequence of operators or goal state
  - Any, an optimal one (cost definition needed), all

EXAMPLE: 8-PUZZLE



EXAMPLE: 8-PUZZLE

State space: configuration of the eight tiles on the board

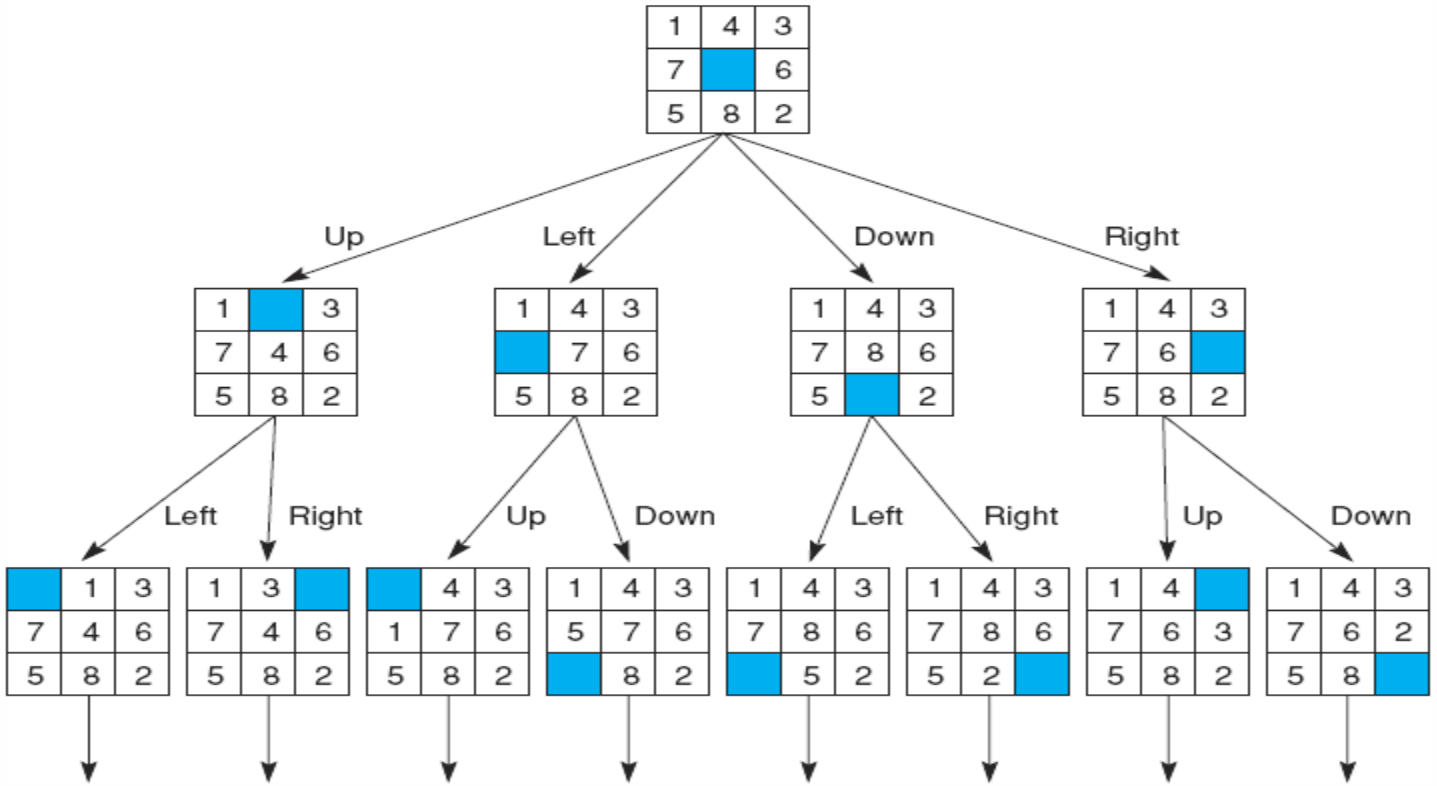
Initial state: any configuration

Goal state: tiles in a specific order

Operators or actions: “blank moves”

- Condition: the move is within the board
- Transformation: blank moves *Left*, *Right*, *Up*, or *Down*

Solution: optimal sequence of operators


1 4 3
7 6
5 8 2
Up
Left
Down
Right
1 3
7 4 6
5 8 2
1 4 3
7 6
5 8 2
1 4 3
7 8 6
5 2
1 4 3
7 6
5 8 2
Left
Right
Up
Down
Left
Right
Up
Down
1 3
7 4 6
5 8 2
1 3
7 4 6
5 8 2
4 3
1 7 6
5 8 2
1 4 3
5 7 6
8 2
1 4 3
7 8 6
5 2
1 4 3
7 8 6
5 2
1 4
7 6 3
5 8 2
1 4 3
7 6 2
5 8

STRUCTURE OF THE STATE SPACE

Data structures:

- Trees: only one path to a given node
- Graphs: several paths to a given node

Operators: directed arcs between nodes

The search process explores the state space.

In the worst case all possible paths between the initial state and the goal state are explored.

DEFINITION

GRAPH

A graph consists of:

A set of *nodes* $N_1, N_2, N_3, \ldots, N_n, \ldots$, which need not be finite.

A set of *arcs* that connect pairs of nodes.

Arcs are ordered pairs of nodes; i.e., the arc (N_3, N_4) connects node N_3 to node N_4. This indicates a direct connection from node N_3 to N_4 but not from N_4 to N_3, unless (N_4, N_3) is also an arc, and then the arc joining N_3 and N_4 is undirected.

If a directed arc connects N_j and N_k, then N_j is called the *parent* of N_k and N_k, the *child* of N_j. If the graph also contains an arc (N_j, N_l), then N_k and N_l are called *siblings*.

A *rooted* graph has a unique node N_s from which all paths in the graph originate. That is, the root has no parent in the graph.

A *tip* or *leaf* node is a node that has no children.

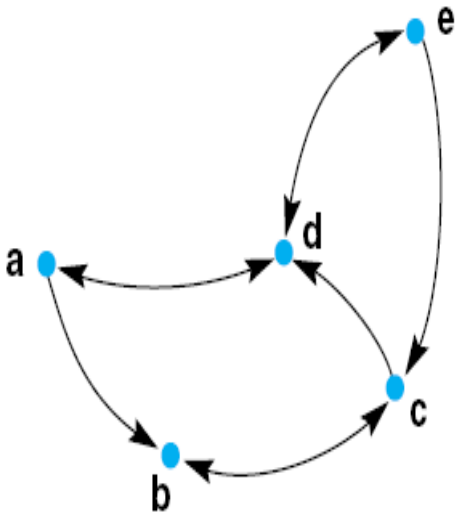
An ordered sequence of nodes $[N_1, N_2, N_3, \ldots, N_n]$, where each pair N_i , N_{i+1} in the sequence represents an arc, i.e., (N_i, N_{i+1}), is called a *path* of length $n - 1$.

On a path in a rooted graph, a node is said to be an *ancestor* of all nodes positioned after it (to its right) as well as a *descendant* of all nodes before it.

A path that contains any node more than once (some N_j in the definition of path above is repeated) is said to contain a *cycle* or *loop*.

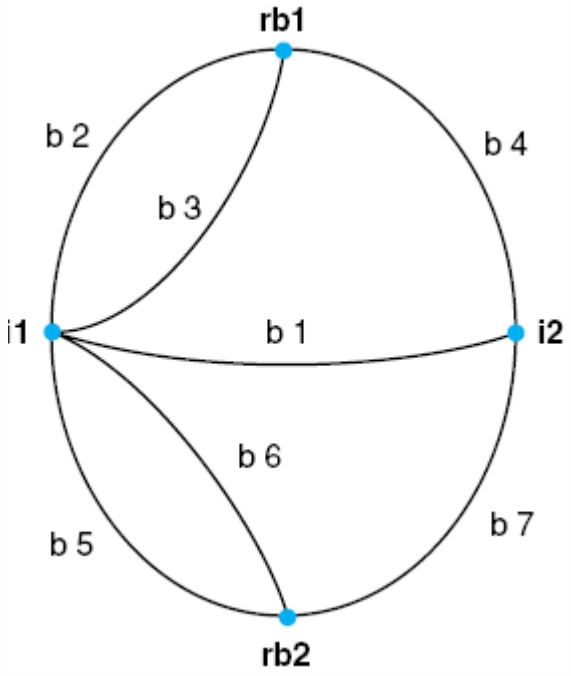
A *tree* is a graph in which there is a unique path between every pair of nodes. (The paths in a tree, therefore, contain no cycles.)

DIRECTED/UNDIRECTED GRAPH



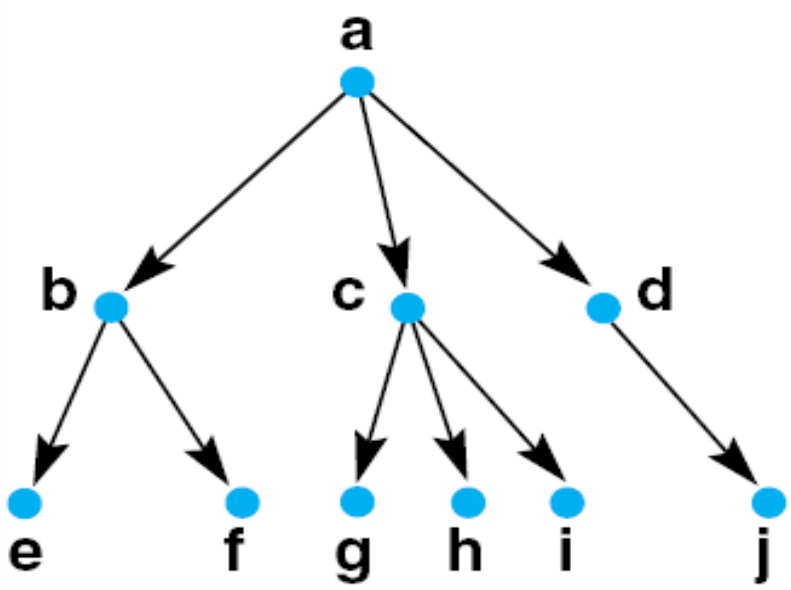
Nodes = {a,b,c,d,e}

Arcs = {(a,b),(a,d),(b,c),(c,b),(c,d),(d,a),(d,e),(e,c),(e,d)}



TREE

- Trees are always rooted
- Loops can be found among states



SEARCH AS GOAL SATISFACTION

Satisfying a goal

- Agent knows what the goal is
- Agent cannot evaluate intermediate solutions (uninformed)
- The environment is:
  - Static
  - Discrete
  - Observable
  - Deterministic

EXAMPLE: HOLIDAY IN ROMANIA

- On holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest at 13:00
- Let’s configure this to be an AI problem

ROMANIA

What's the problem?

- Accomplish a *goal*
  - Reach Bucharest by 13:00

- **So this is a goal-based problem**

ROMANIA

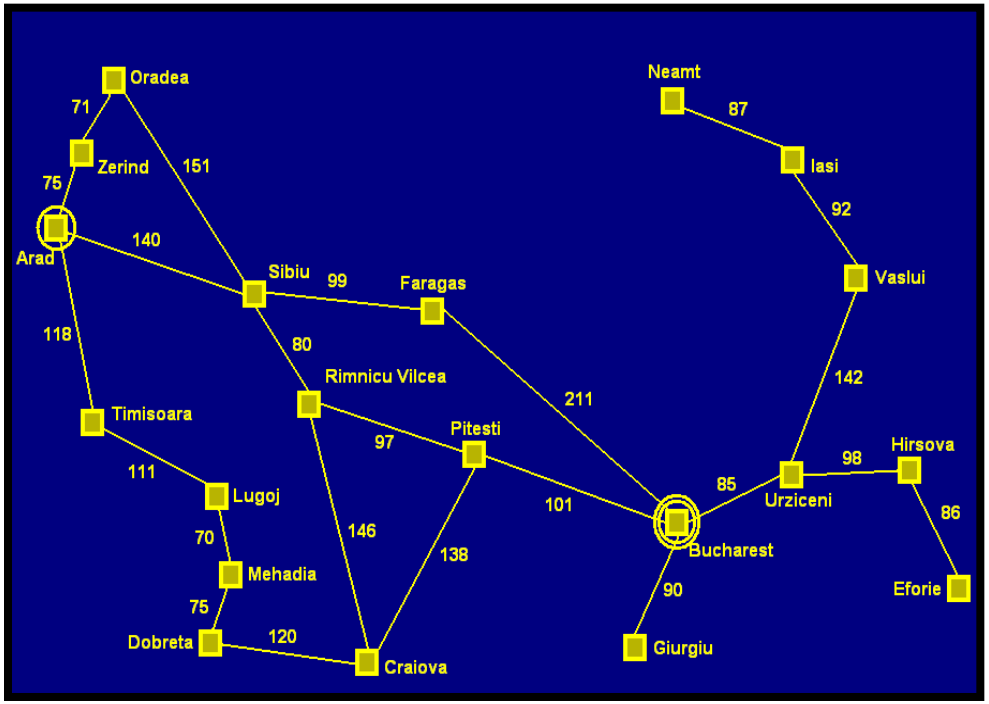
What qualifies as a solution?

- You can/cannot reach Bucharest by 13:00
- The actions one takes to travel from Arad to Bucharest along the shortest (in time) path

ROMANIA

What additional information does one need?

- A map



MORE CONCRETE PROBLEM DEFINITION

A state space	*Which cities could you be in?*
An initial state	*Which city do you start from?*
A goal state	*Which city do you aim to reach?*
A function defining state transitions	*When in city foo, the following cities can be reached*
A function defining the “cost” of a state sequence	*How long does it take to travel through a city sequence?*

MORE CONCRETE PROBLEM DEFINITION

A state space	***Choose a representation***
An initial state	***Choose an element from the representation***
A goal state	***Create `goal_function(state)` such that TRUE is returned upon reaching goal***
A function defining state transitions	***`successor_function(state`$_i$`) =` {<action$_a$, state$_a$>, <action$_b$, state$_b$>, …}***
A function defining the “cost” of a state sequence	***`cost (sequence) =` number***

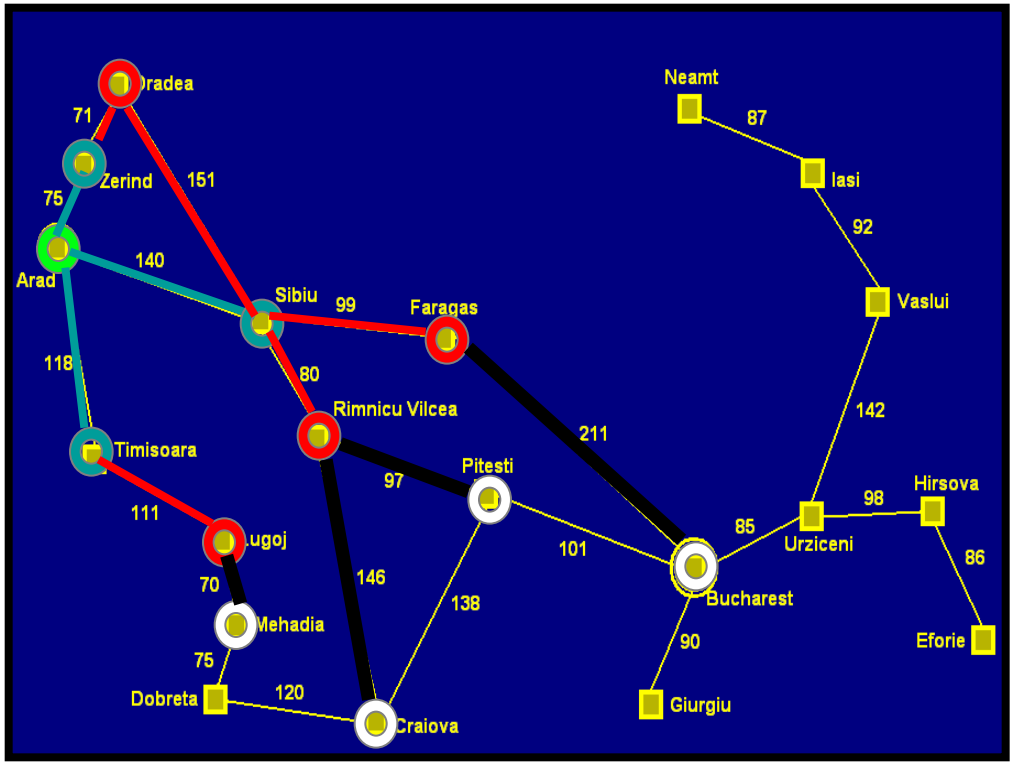
IMPORTANT NOTES ABOUT THIS EXAMPLE

- **Static environment** (available states, successor function, and cost functions don't change)
- **Observable** (the agent knows where it is)
- **Discrete** (the actions are discrete)
- **Deterministic** (successor function is always the same)

TREE SEARCH ALGORITHMS

Basic idea:

- Simulated exploration of state space by generating successors of already explored states (AKA expanding states)

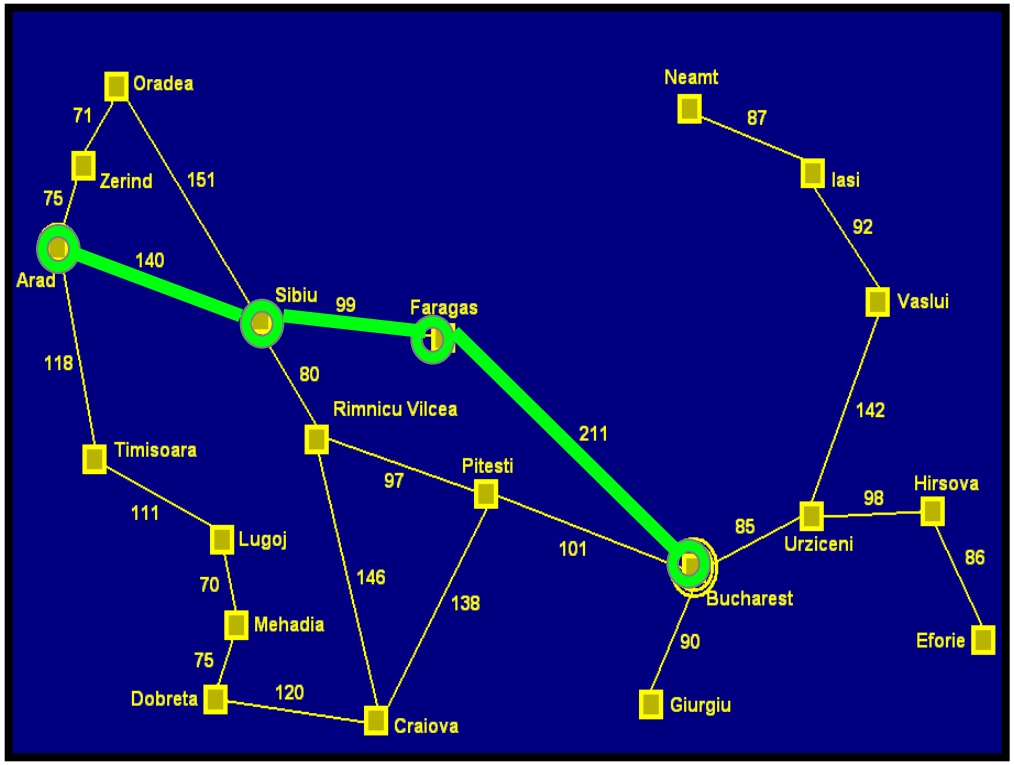


Sweep out from start (breadth)

TREE SEARCH ALGORITHMS

Basic idea:

- Simulated exploration of state space by generating successors of already explored states (AKA expanding states)



Go East, young man! (depth)

IMPLEMENTATION: GENERAL SEARCH ALGORITHM

```
Algorithm General Search
  Open_states.insert (Initial_state)
  Current= Open_states.first()
    while not is_final?(Current) and not Open_states.empty?() do
      Open_states.delete_first()
      Closed_states.insert(Current)
      Successors= generate_successors(Current)
      Successors= process_repeated(Successors, Closed_states,  Open_states)
      Open_states.insert(Successors)
      Current= Open_states.first()
    eWhile
eAlgorithm
```

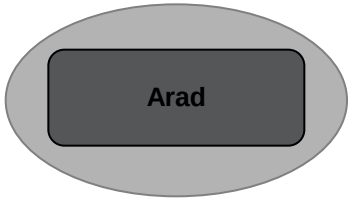
EXAMPLE: ARAD → BUCHAREST

Algorithm General Search

Open_states.insert (Initial_state)

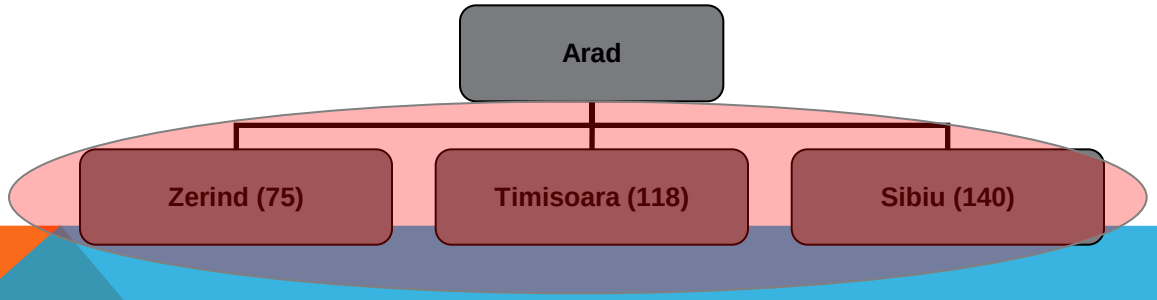
EXAMPLE: ARAD → BUCHAREST

Current= Open_states.first()



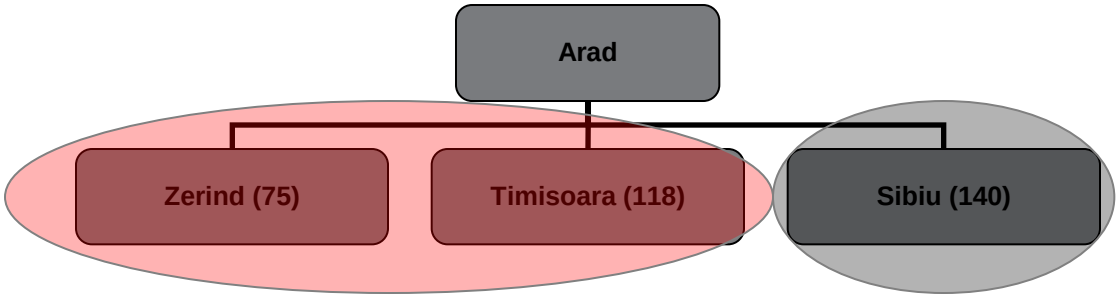
EXAMPLE: ARAD → BUCHAREST

```
while not is_final?(Current) and not Open_states.empty?() do
    Open_states.delete_first()
    Closed_states.insert(Current)
    Successors= generate_successors(Current)
    Successors= process_repeated(Successors, Closed_states,
            Open_states)
    Open_states.insert(Successors)
```



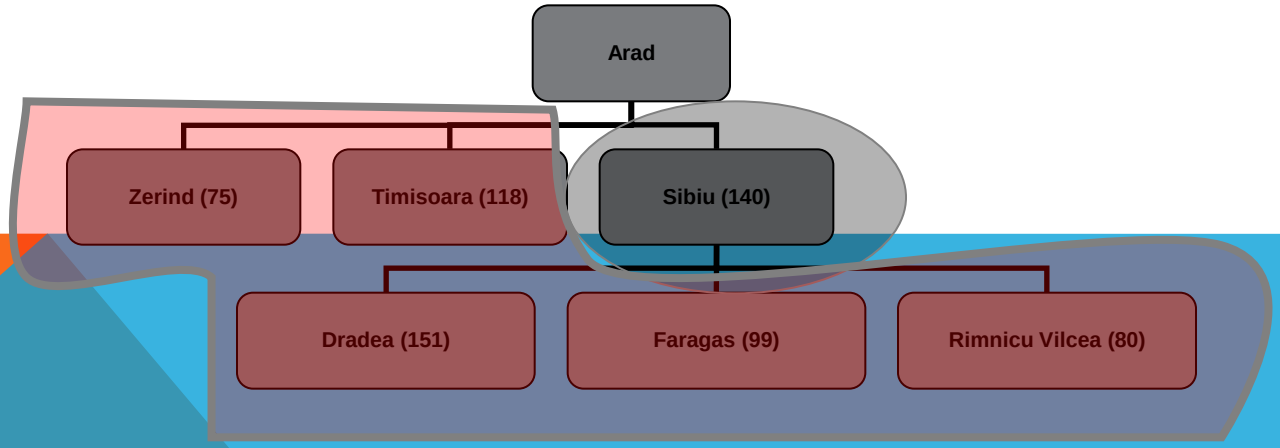
EXAMPLE: ARAD → BUCHAREST

Current= Open_states.first()



EXAMPLE: ARAD → BUCHAREST

```
while not is_final?(Current) and not Open_states.empty?() do
    Open_states.delete_first()
    Closed_states.insert(Current)
    Successors= generate_successors(Current)
    Successors= process_repeated(Successors, Closed_states,
            Open_states)
    Open_states.insert(Successors)
```



IMPLEMENTATION: STATES VS. NODES

State

- (Representation of) a physical configuration

Node

- Data structure constituting part of a search tree
  - Includes *parent, children, depth, path cost* g(x)

States do not have parents, children, depth, or path cost!

SEARCH STRATEGIES

A 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/expanded
- Space complexity – maximum nodes in memory
- Optimality – does it always find a least-cost solution?

SEARCH STRATEGIES

Time and space complexity are measured in terms of:

- b – maximum branching factor of the search tree (may be infinite)
- d – depth of the least-cost solution
- m – maximum depth of the state space (may be infinite)

Searching Algorithm

- Uninformed search strategies (blind search)
- Formed search strategies (Heuristic search)

UNINFORMED SEARCH STRATEGIES

Uninformed strategies use only the information available in the problem definition

- Breadth-first search
- Depth-first search
- Backtracking search
- Iterative deepening search

NODES

Open nodes:

- Generated, but not yet explored
- Explored, but not yet expanded

Closed nodes:

- Explored and expanded

BREADTH-FIRST SEARCH

```
function breadth_first_search;

begin
   open := [Start];                                              % initialize
   closed := [ ];
   while open ≠ [ ] do                                           % states remain
      begin
         remove leftmost state from open, call it X;
            if X is a goal then return SUCCESS                  % goal found
               else begin
                  generate children of X;
                  put X on closed;
                  discard children of X if already on open or closed;   % loop check
                  put remaining children on right end of open           % queue
               end
      end
   return FAIL                                                   % no states left
end.
```

BREADTH-FIRST SEARCH

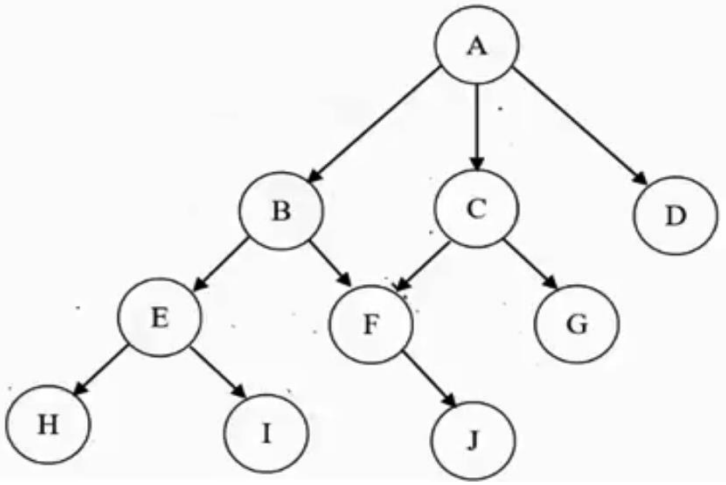
Expand shallowest unexpanded node

Implementation:

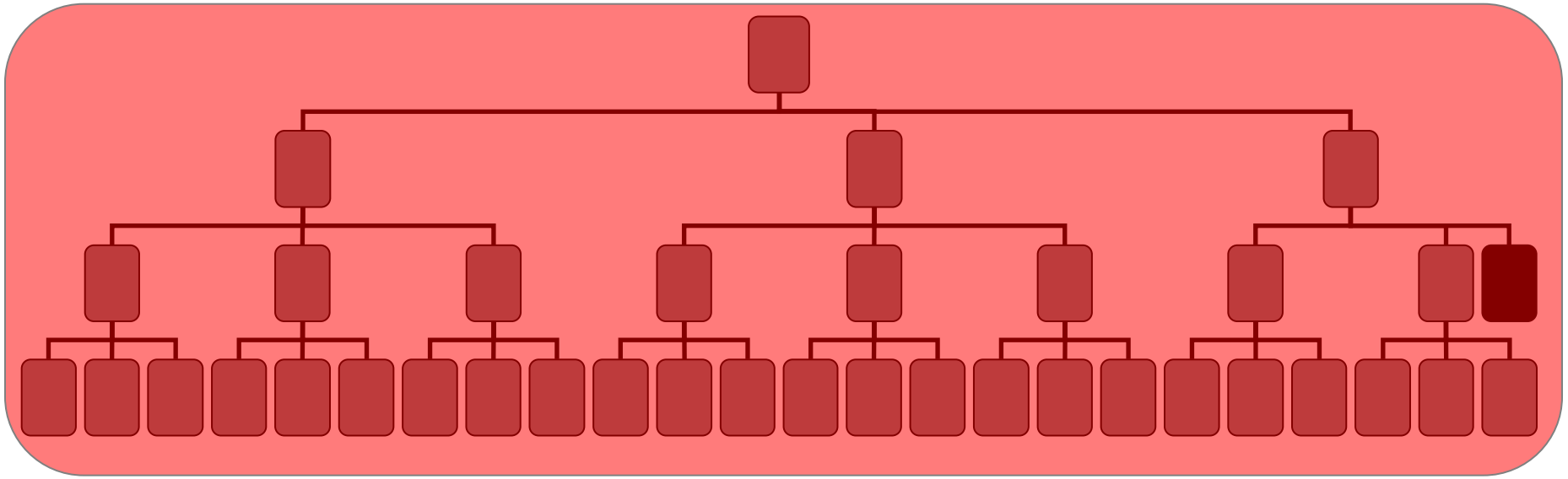
– A FIFO queue, i.e., new successors go at end

Q) Apply the breadth first search algorithm on the following graph , where the start state is (A) and the desired goal state is (G),show the successive values of open and closed ,and the traversed path

Iteration #	X	open	closed
0	-	[A]	[ ]
1	A	[BCD]	[A]
2	B	[CDEF]	[BA]
3	C	[DEFG]	[CBA]
4	D	[EFG]	[DCBA]
5	E	[FGHI]	[EDCBA]
6	F	[GHIJ]	[FEDCBA]
7	G	G is the goal	



SPACE COST OF BFS



Because you must be able to generate the path upon finding the goal state, all visited nodes must be stored O (b^{d+1})

PROPERTIES OF BREADTH-FIRST SEARCH

Complete?

- Yes (if $b_{(\text{max branch factor})}$ is finite)

Time?

- $1 + b + b^2 + \ldots + b^d + b(b^d-1) = O(b^{d+1})$, i.e., exponential in d

Space?

- $O(b^{d+1})$ (keeps every node in memory)

Optimal?

- Only if cost = 1 per step, otherwise not optimal in general

Space is the big problem; it can easily generate nodes at 10 MB/s, so 24 hrs = 860GB!

DEPTH-FIRST SEARCH

Algorithm Depth-First-Search

Begin

Initialization: $open = [Start]$, $close = []$, $parent\ [Start] = "null"$, $found = No$.

While $open <> []$ do

Begin

- remove the first state from left of open, call it X;
- if X is a goal then $found = true$, break;
- generate all children of X and put them in list L;
- put X in $close$;
- eliminate from L any states already in $close$;
- eliminate from $open$ any states already in L;
- append L to the left of $open$;
- for each child Y in L set $parent[Y] = X$;
- empty L

End;

If ($found = true$) compute the solution path;

Else return fail;

End.

DEPTH-FIRST SEARCH

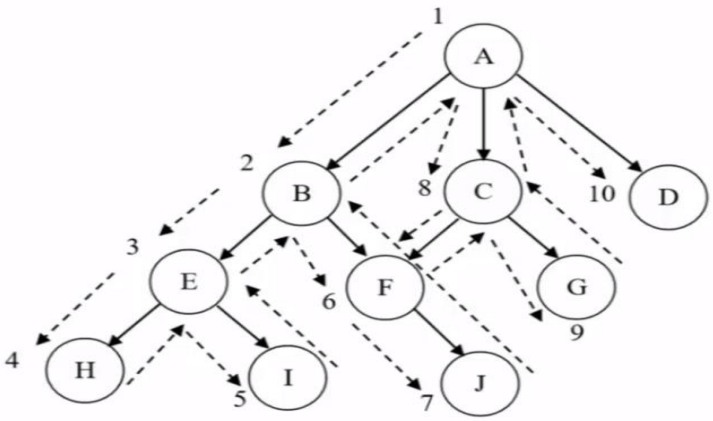
Expand deepest unexpanded node

Implementation:

– A LIFO queue, i.e., a stack

Q) Apply the depth first search algorithm on the following graph , where the start state is (A) and the desired goal state is (G),show the successive values of open and closed ,and the traversed path

Iteration #	X	open	closed
0	-	[A]	[ ]
1	A	[BCD]	[A]
2	B	[EFCD]	[BA]
3	E	[HIFCD]	[EBA]
4	H	[IFCD]	[HEBA]
5	I	[FCD]	[IHEBA]
6	F	[JCD]	[FIHEBA]
7	J	[CD]	[JFIHEBA]
8	C	[GD]	[CJFIHEBA]
9	G	G is the goal	



DEPTH-FIRST SEARCH

- **Complete?**
  - No: fails in infinite-depth spaces, spaces with loops.
  - Can be modified 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?**
  - *$O(b^m)$, i.e., linear space!*
- **Optimal?**
  - No

BACKTRACK ALGORITHM

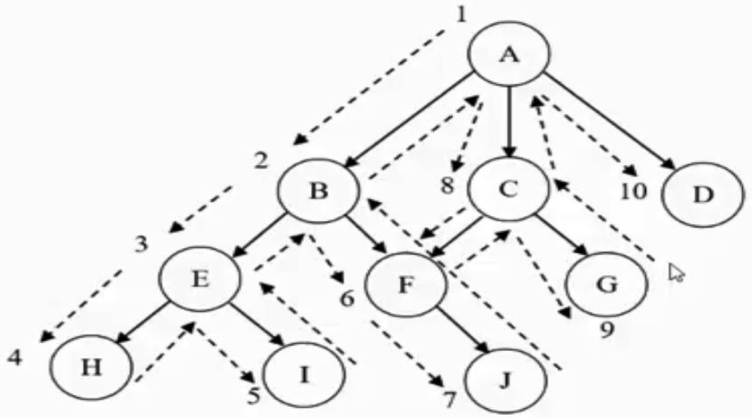
```
function backtrack;

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.
```

BACKTRACK ALGORITHM

- CS: current state
- SL: states list
- NSL: New states list (store children with old state)
- DE: dead end list

Q) Apply the backTrack search algorithm on the following graph , where the start state is (A) and the desired goal state is (G),show the successive values of SE , NSL , DE ,and the traversed path

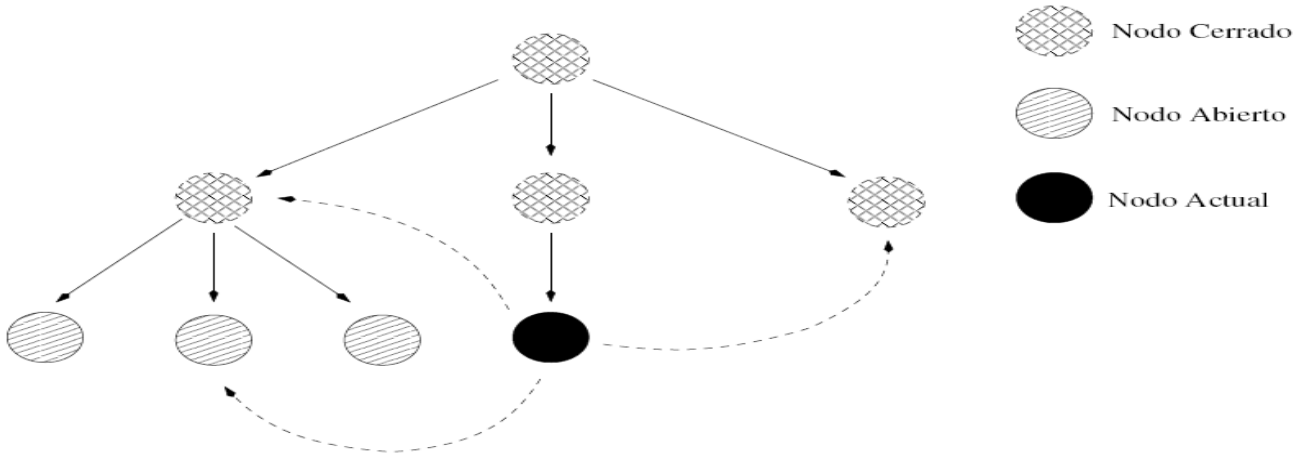
Iteration #	CS	SL	NSL	DE
0	A	[A]	[A]	[ ]
1	B	[BA]	[BCDA]	[ ]
2	E	[EBA]	[EFBCDA]	[ ]
3	H	[HEBA]	[HIEFBCDA]	[ ]
	I	[EBA]	[IEFBCDA]	[H]
4	I	[IEBA]	[IEFBCDA]	[H]
	E	[EBA]	[EFBCDA]	[IH]
	F	[BA]	[FBCDA]	[EIH]
5	F	[FBA]	[FBCDA]	[EIH]
6	J	[JFBA]	[JFBCDA]	[EIH]
	F	[FBA]	[FBCDA]	[JEIH]
	B	[BA]	[BCDA]	[FJEIH]
	C	[A]	[CDA]	[BFJEIH]
7	C	[CA]	[CDA]	[BFJEIH]
8	G	[GCA]	[GCDA]	[BFJEIH]
9	G is the Goal, Path = [GCA]			



TREATMENT OF REPEATED STATES

Breadth-first:

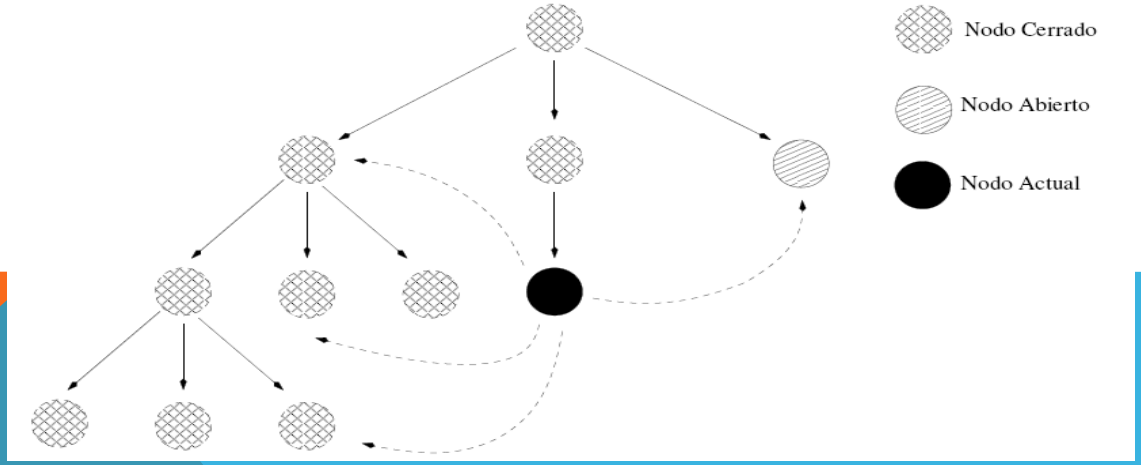
- If the repeated state is in the structure of closed or open nodes, the actual path has equal or greater depth than the repeated state and can be forgotten.



TREATMENT OF REPEATED STATES

Depth-first:

- If the repeated state is in the structure of closed nodes, the actual path is kept if its depth is less than the repeated state.
- If the repeated state is in the structure of open nodes, the actual path has always greater depth than the repeated state and can be forgotten.



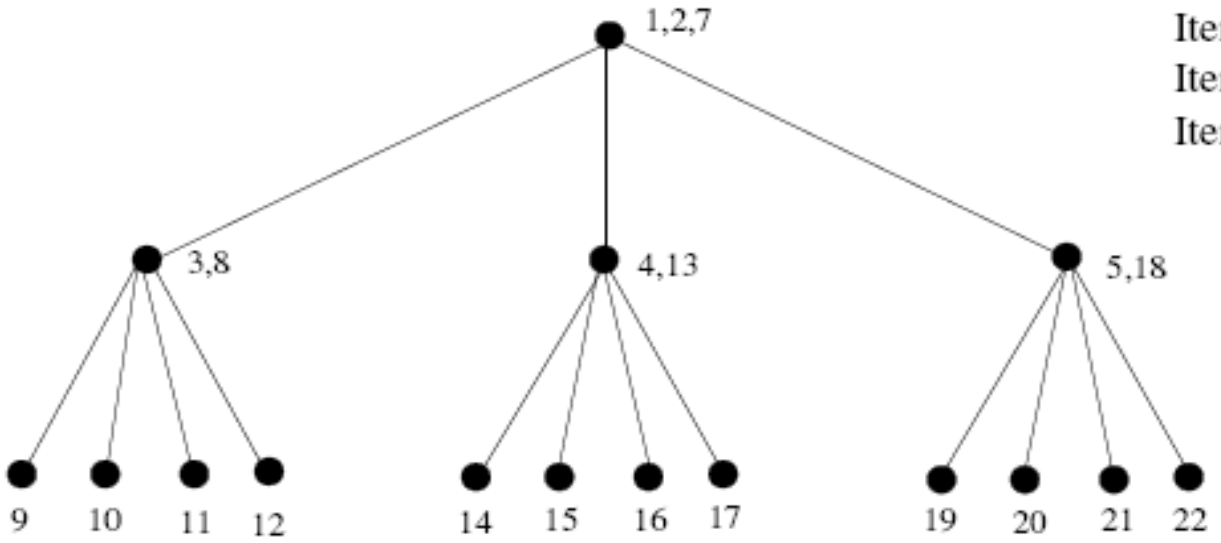
ITERATIVE DEEPENING SEARCH

```
Algoritmo Busqueda en profundidad iterativa (limite: entero)
 prof=1;
 Est_abiertos.inicializar()
 mientras (no es_final?(Actual)) y prof<limite hacer
  Est_abiertos.insertar(Estado inicial)
  Actual= Est_abiertos.primero()
   mientras (no es_final?(Actual)) y no Est_abiertos.vacia?() hacer
    Est_abiertos.borrar_primero()
    Est_cerrados.insertar(Actual)
    si profundidad(Actual) <= prof entonces
     Hijos= generar_sucesores(Actual)
     Hijos= tratar_repetidos(Hijos, Est_cerrados, Est_abiertos)
    fsi
    Est_abiertos.insertar(Hijos)
    Actual= Est_abiertos.primero()
   fmientras
   prof=prof+1
   Est_abiertos.inicializar()
 fmientras
fAlgoritmo
```

ITERATIVE DEEPENING SEARCH

- The algorithm consists of iterative, depth-first searches, with a maximum depth that increases at each iteration. Maximum depth at the beginning is 1.
- Behavior similar to BFS, but without the spatial complexity.
- Only the actual path is kept in memory; nodes are regenerated at each iteration.
- DFS problems related to infinite branches are avoided.
- To guarantee that the algorithm ends if there is no solution, a general maximum depth of exploration can be defined.

ITERATIVE DEEPENING SEARCH



Iteracion 1: 1
Iteracion 2: 3,4,5
Iteracion 3: 7,8,9,...27

SUMMARY

- All uninformed searching techniques are more alike than different.
- Breadth-first has space issues, and possibly optimality issues.
- Depth-first has time and optimality issues, and possibly completeness issues.
- Depth-limited search has optimality and completeness issues.
- Iterative deepening is the best uninformed search we have explored.

UNINFORMED VS. INFORMED

Blind (or uninformed) search algorithms:

- Solution cost is not taken into account.

Heuristic (or informed) search algorithms:

- A solution cost estimation is used to guide the search.
- The optimal solution, or even a solution, are not guaranteed.

END OF LECTURE 3

NEXT LECTURE: FORMAL SEARCHES (HEURISTIC SEARCH)