Tecnológico de Monterrey

CS4000 Intelligent Systems

Solving Problems by Searching

Santiago Conant, Ph.D.
sconant@itesm.mx

Contents

Problem-Solving Agents (PSAs)

Characteristics of a PSA

- One kind of **goal-based agent**.
- Determines a sequence of actions that allow it to maximize its performance measure.
- **Idea:** Initial state => final state
- **Goal** = Set of states that maximize agent's performance measure.
- *Goal formulation*:
  - Current State + Performance Measure => Goal

Characteristics of a PSA

- **Problem formulation**
  - Elements that implicitly describe their solution space.
- **Solution space**
  - Space of states and actions.
- **Solution**
  - Sequence of actions leading to a goal state from an initial state.
- **Solution method**
  - Search the solution space.
- **Search**
  - Process that examines possible sequences of actions to find a solution.

A simple PSA program

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
  persistent: seq, an action sequence, initially empty
              state, some description of the current world state
              goal, a goal, initially null
              problem, a problem formulation

  state ← UPDATE-STATE(state, percept)
  if seq is empty then
     goal ← FORMULATE-GOAL(state)
     problem ← FORMULATE-PROBLEM(state, goal)
     seq ← SEARCH(problem)
     if seq = failure then return a null action
  action ← FIRST(seq)
  seq ← REST(seq)
  return action
```

Task environments for PSA

- The solutions are implemented without taking into account the percepts.
  - Open-loop control system.
- *Characteristics of the Task Environment*
  - Static
  - Fully observable
  - Discrete
  - Deterministic
- *Conclusion*
  - Task environments of the easiest type

Problem formulation

1 **Configuration State** = Elements that describe the different situations of the task environment.

2 **Initial state** = State where the agent begins.

3 Set of possible **actions.**

4 **Transition model** (what actions do)
- Successor function = given a state returns a set of pairs <action, successor> (alternative = set of operators).
- Space of States (EE) = set of all states attainable from the Initial State.
- Representation EE = graph where nodes = states and arcs = actions.
- Path = Sequence of states connected by a sequence of actions.

5 **Goal test** = Way to determine whether a given state is a goal state.

6 **Trajectory cost** = Function that assigns a numeric cost to each path.

Problems scope

- *Toy problem*
  - PROBLEM used to illustrate or exercise several problem-solving methods.
- *Real-world problem*
  - PROBLEM whose solution interests people.

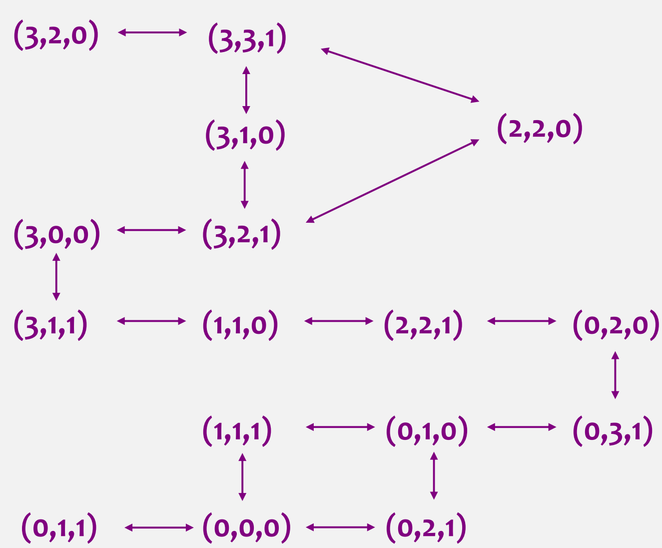
Missionaries and Cannibals (toy problem)

- Three missionaries meet 3 hungry cannibals at the edge of a river.
- Next to them is a boat that has places for a maximum of two people.
- The boat can not cross empty the river.
- Missionaries must find a sequence of crossings that allow both them and the cannibals to cross the river safely.
- At no time should more cannibals than missionaries be left on one of the banks of the river, for if that happened, the missionaries would perish in the jaws of the hungry cannibals.

Problem formulation for a PSA

- **State:** An ordered sequence of three numbers representing the number of missionaries, cannibals, and boats that are in the starting bank of the river.
- **Initial state:** (3, 3, 1).
- **Actions:** Cross the river with a missionary, with a cannibal, with two missionaries, with two cannibals, or with one of each.
- **Transition model:** Each boat crossing reduces the number of people on the river side where the crossing originates and increases it on the destination side.
- **Goal test:** Current state = (0,0,0).
- **Path cost:** One for each crossing of the river with the boat.

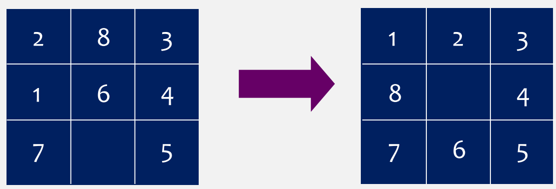
State-space shape



8-Puzzle (toy problem)

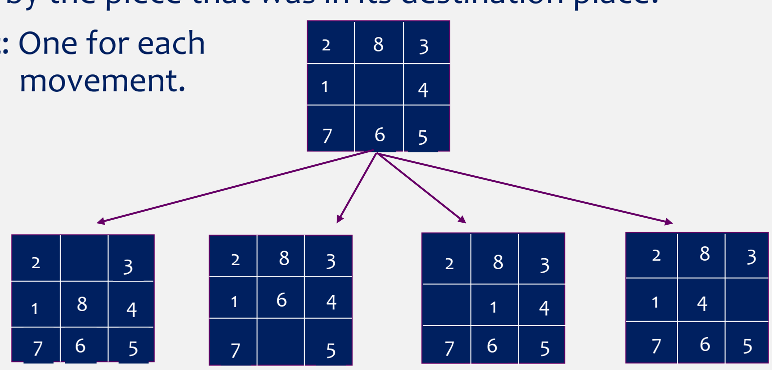
In a 3x3 board with 8 pieces and a free square, we want to achieve a final configuration by means of a sequence of unit movements of the pieces towards the free square.

- **States:** 3x3 matrix showing the position of each piece and the free square.
- **Initial state:** Initial configuration.
- **Goal test:** Current state = final configuration.



Problem formulation

- **Actions:** Operators to move the free square in the four directions.
- **Transition model:** When moving the free square, its place is occupied by the piece that was in its destination place.
- **Path cost:** One for each movement.



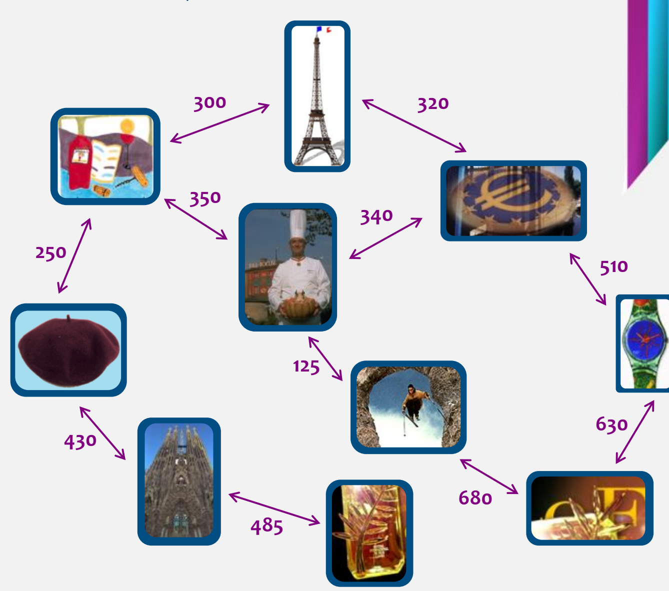
Other toy problems

- The World of the Vacuum Cleaner.
- The 8-queens problem.
- Crypto arithmetic problems.
- The jugs problem.

Path planning (real problem)

The agent must find a path between two points on a map.

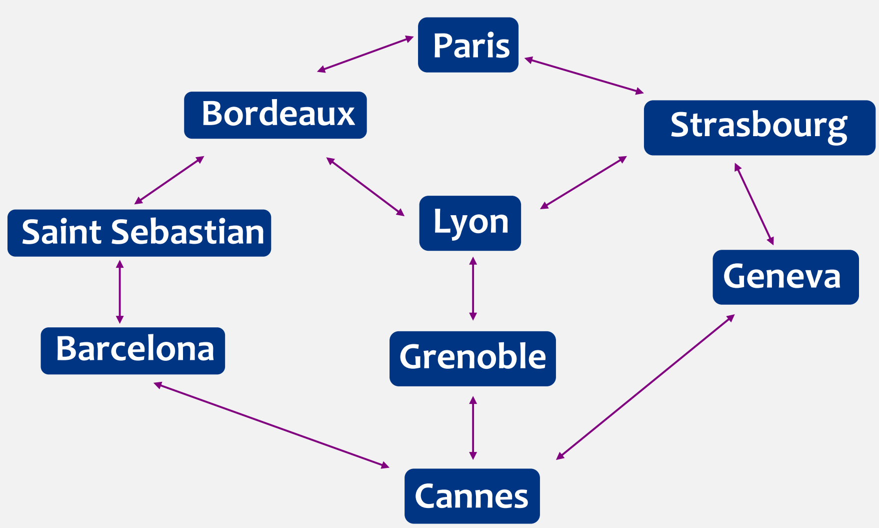
Example: Find the sequence of cities to travel from Paris to Cannes.



Problem formulation

- **State:** Names of cities.
- **Initial state:** Departure city.
- **Actions:** Travel to a neighboring city.
- **Transition model:** The traveler reaches the neighboring city.
- **Goal test:** CURRENT city = DESTINATION city.
- **Path cost:** Sum of distances separating the adjacent cities on the path.

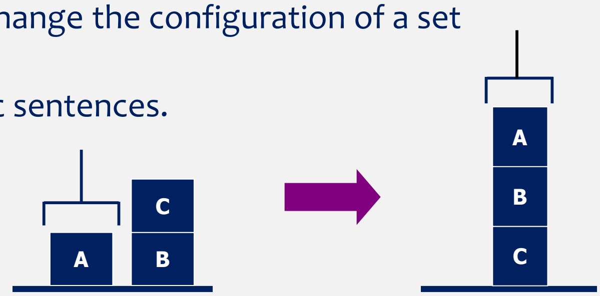
State space of the problem



Action plans (real problem)

Use a robotic arm to change the configuration of a set of blocks.

- **State:** predicate logic sentences.



Initial state
free-hand
on-table (A)
on-table (B)
on (C, B)
free (A)
free (C)

Goal state
free-hand
on-table (C)
on (A, B)
on (B, C)
free (A)

Operator table

OPERATORS	PRE-CONDITIONS	DELETE	ADD
Lift (X)	free-hand on-table (X) free (X)	free-hand on-table (X) free (X)	has (X)
Download (X)	has (X)	has (X)	free-hand on-table (X) free (X)
Remove (X, Y)	free-hand on (X, Y) free (X)	free-hand on (X, Y) free (X)	has (X) Free (Y)
Stack (X, Y)	has (X) free (Y)	has (X) free (Y)	free-hand on (X, Y) free (X)

Other real problems

- Traveling Salesman Problem (TSP).
- Distribution in VLSI.
- Robot navigation.
- Automatic assembly sequencing.
- Internet search.

Blind Search for Problem Solving

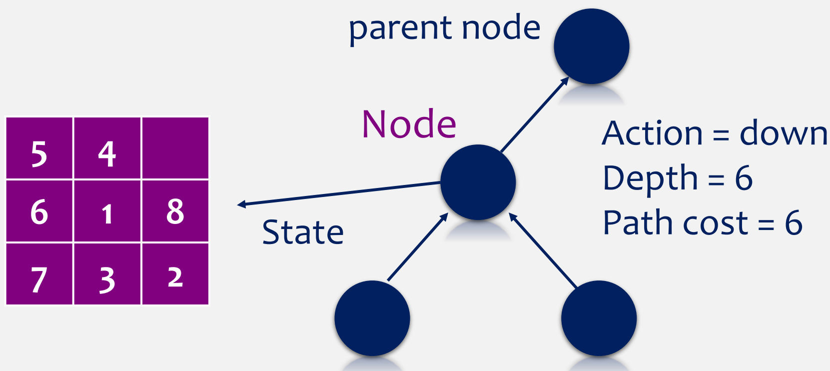
General search algorithm

Function SEARCH(problem, strategy) **return** solution or failure
initializes the search tree using the initial state
repeat
if there is no candidate nodes for expansion **then**
return failure
choose a leaf node for expansión according to strategy
if the node contains a goal state **then**
return solution
expand node and adds resulting nodes to the search tree

Searching concepts

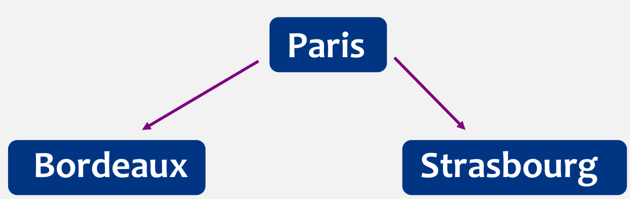
- Search tree.
- Search node.
- Node vs. State.
- Node expansion.
- Search strategy.
- State space vs. search tree.

Node representation



Search strategy

Node to be expanded



Which of 2 open nodes we explore first?

Tree search

```
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
```

Search graph

```
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
```

Algorithm selection

- Performance criteria:
  - Completeness
  - Optimality
  - Time complexity
  - Space complexity
- Measuring complexity:
  - time = nodes generated while searching
  - space = nodes stored in memory

Uninformed search

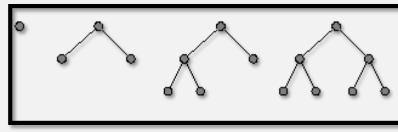
The algorithm only uses the formulation information of the problem.

Blind search algorithms:

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

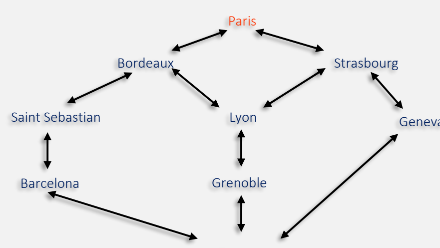
Breadth-first search

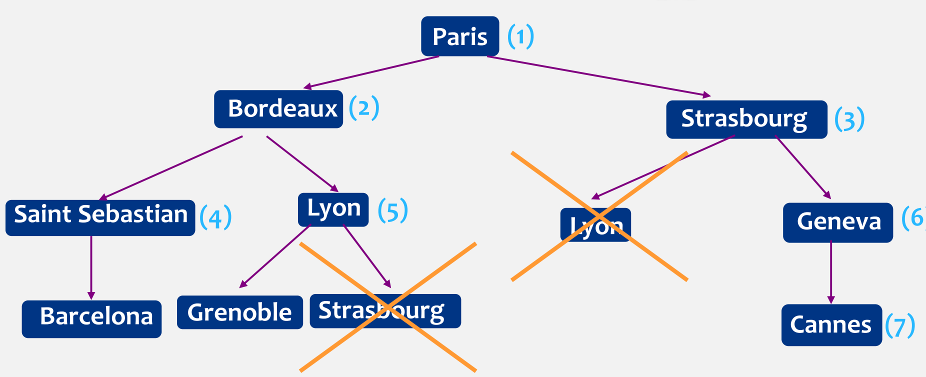
Check all child nodes of the first level before checking the tree to a deeper level, i.e., **level-by-level search.**



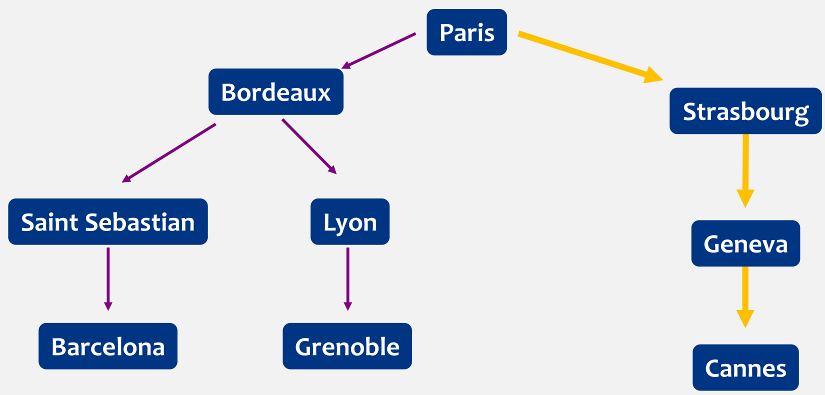
```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  frontier ← a FIFO queue with node as the only element
  explored ← an empty set
  loop do
     if EMPTY?(frontier) then return failure
     node ← POP(frontier)   /* chooses the shallowest node in frontier */
     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
           if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
           frontier ← INSERT(child, frontier)
```

Breadth-first search





Breadth-first solution



Combinatorial explosion

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

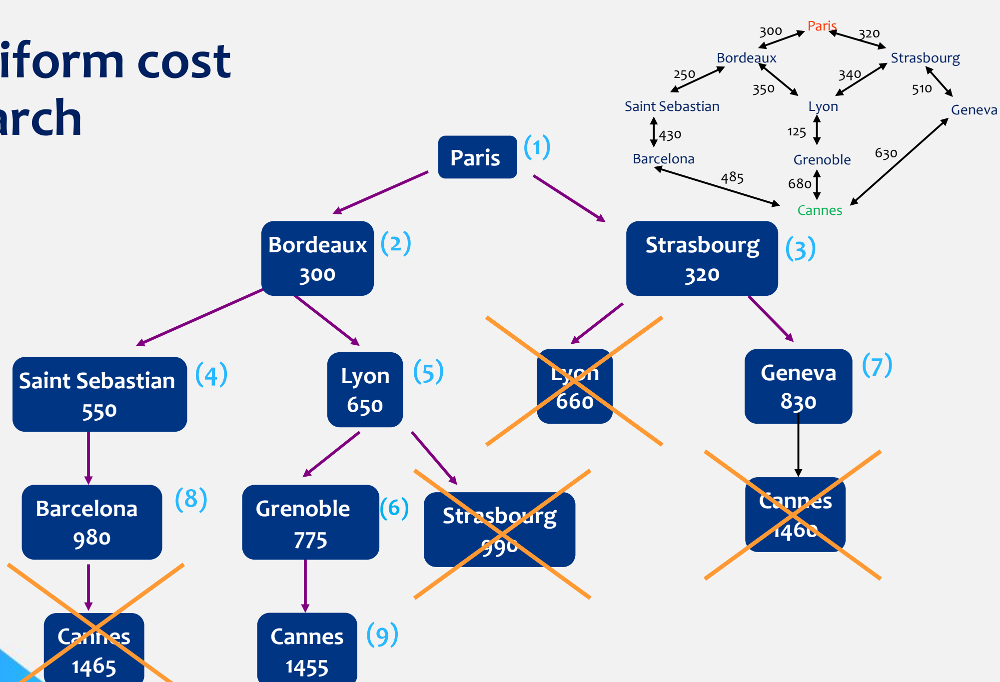
Figure 3.13 Time and memory requirements for breadth-first search. The numbers shown assume branching factor $b = 10$; 1 million nodes/second; 1000 bytes/node.

Uniform cost algorithm

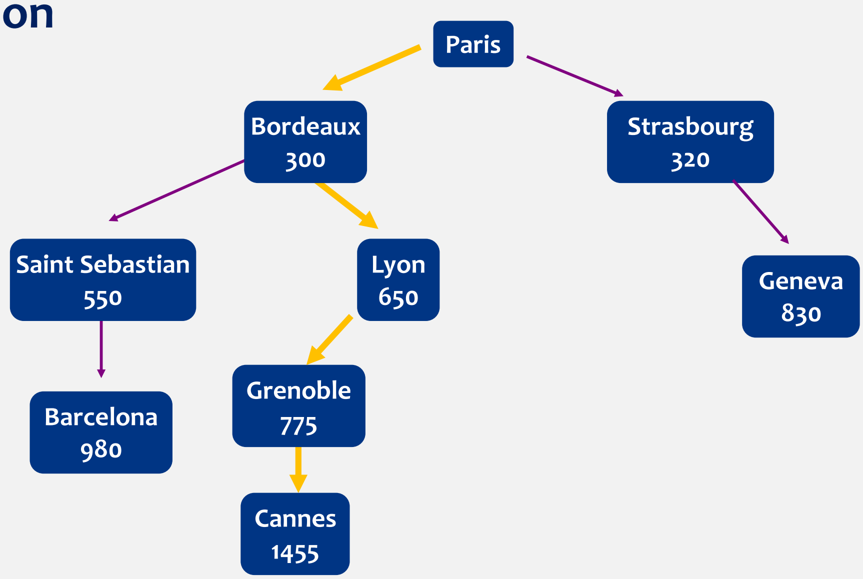
Similar to the breadth-first search, but instead of expanding first the lower level node, **choose the node with the lowest cumulative cost.**

```
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
```

Uniform cost search



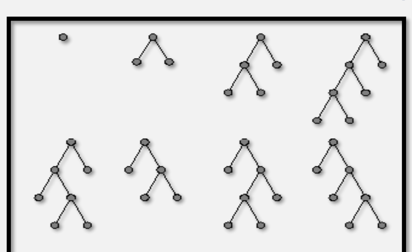
Uniform cost solution



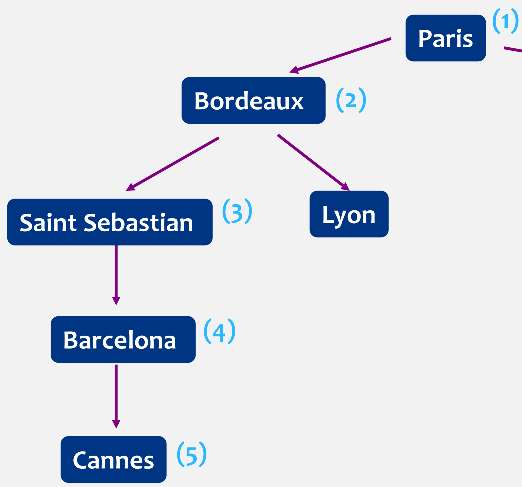
Depth-first search

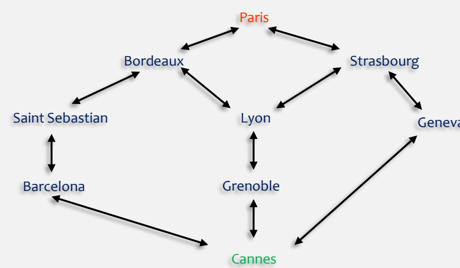
Expands the search tree **branch by branch**. If a branch does not lead to a solution, the search returns to a previous point to explore the closest unexpanded branch (**backtracking**).

Algorithm: Similar to breadth-first search, but using a stack instead of a queue.

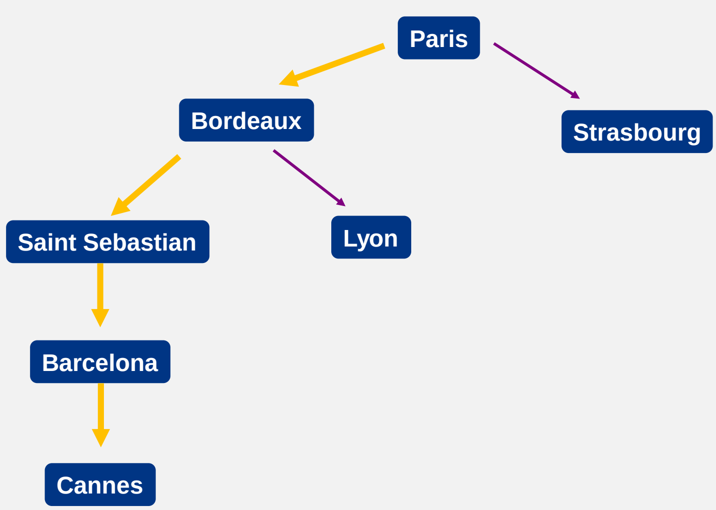


Depth-first search





Depth-first solution



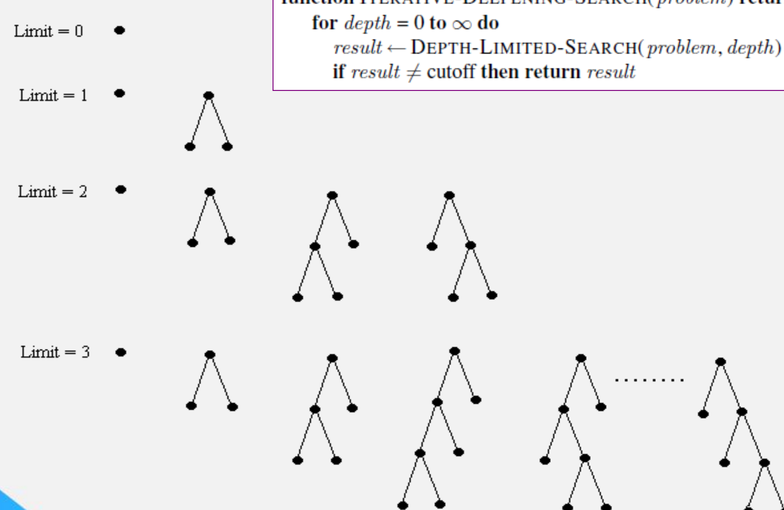
Depth-limited search

A recursive version of depth-first search with a depth limit.

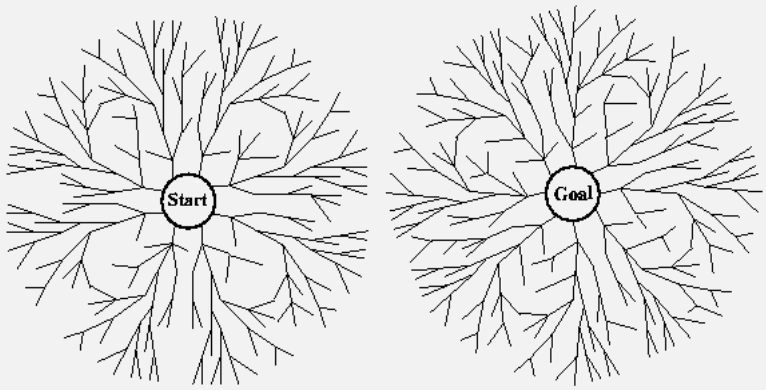
```
function DEPTH-LIMITED-SEARCH(problem, limit) returns a solution, or failure/cutoff
  return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

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 ≠ failure then return result
      if cutoff_occurred? then return cutoff else return failure
```

Iterative deepening search

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



Bidirectional search



The idea of bidirectional search is to look in both directions, forward from the initial state and backward from the goal.

Comparing Marketing Strategies

Criterion	Breadth-First	Uniform-Cost	Depth-First	Depth-Limited	Iterative Deepening	Bidirectional (if applicable)
Complete?	Yes^a	Yes^{a,b}	No	No	Yes^a	Yes^{a,d}
Time	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(b^m)$	$O(b^\ell)$	$O(b^d)$	$O(b^{d/2})$
Space	$O(b^d)$	$O(b^{1+\lfloor C^*/\epsilon \rfloor})$	$O(bm)$	$O(b\ell)$	$O(bd)$	$O(b^{d/2})$
Optimal?	Yes^c	Yes	No	No	Yes^c	Yes^{c,d}

Figure 3.21 Evaluation of tree-search strategies. b is the branching factor; d is the depth of the shallowest solution; m is the maximum depth of the search tree; l is the depth limit. Superscript caveats are as follows: ^{a} complete if b is finite; ^{b} complete if step costs $\geq \epsilon$ for positive ϵ; ^{c} optimal if step costs are all identical; ^{d} if both directions use breadth-first search.

Conclusions

Conclusions

- The search problem is one of the most important in AI.
- A search problem can be represented in a graph, in terms of states and operators.
- Uninformed search algorithms are systematic but inefficient in time and space.
- To solve real problems it is necessary to use domain-specific knowledge.

Tecnológico de Monterrey