

# Search Strategies LAB

- *Aim of this lesson:*
  1. Assess the comprehension of the search strategies saw in classrooms
  2. Learn to exploit existing libraries, in particular the AIMA implementation available on the net
- *Content:*
  1. Short recap of main search strategies
  2. Few considerations on how to represent a state
  3. Short introduction to the AIMA python library
  4. Exercises

# Requirements

- Python 3 (correctly installed)
- Editor for Python (e.g., Pycharm CE)
- (git-)clone the AIMA repository:  
<https://github.com/aimacode/aima-python>
  - If you don't have git installed, then it's time to install it!!!
- Prepare a new Python project
  - Virtual Env is suggested, but not mandatory
- Install module “numpy”
- Copy in your project the following files:
  - search.py
  - utils.py
- Install Jupyter, there are some notebook already prepared in the AIMA repository

# Looking for Solutions

## Few concepts:

- *Expansion*: given a state, one or more actions are applied, and new states are generated.
- *Search strategy*: at every step, let us choose which will be the next state to be expanded.
- *Search Tree*: the expansion of the states, starting from the initial state, linked to the root node of the tree.
- The leaves of the tree (the *frontier*) represent the set of states/nodes to be expanded.

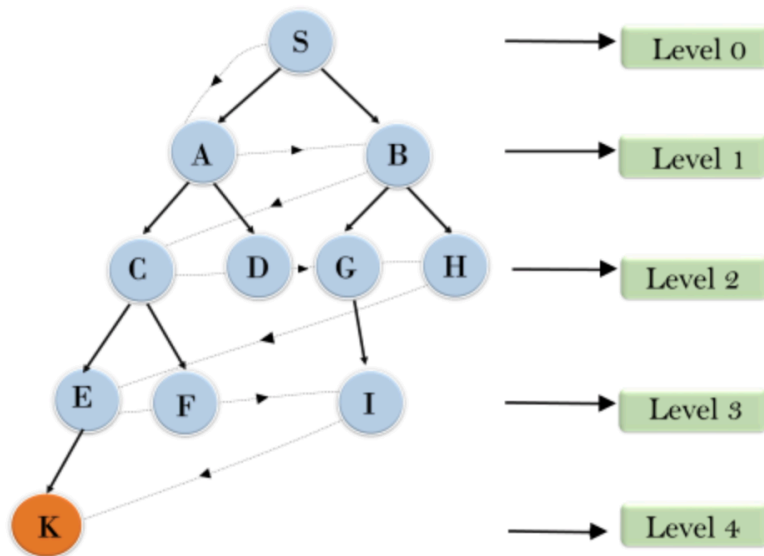
# Search Strategies

- **NON-INFORMED** search strategies:
  - breadth-first (uniform cost);
  - depth-first;
  - limited depth-first;
  - iterative deepening.

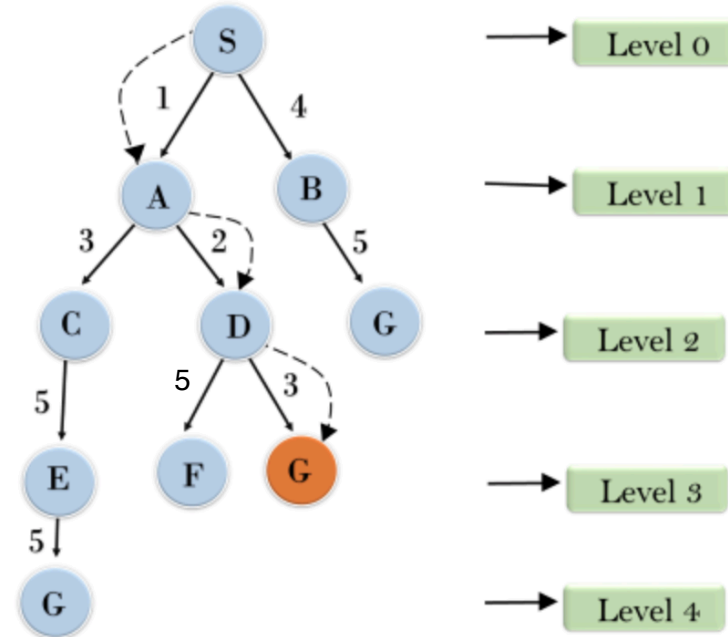
# Search Strategies

- **NON-INFORMED** search strategies:

**Breadth First Search**



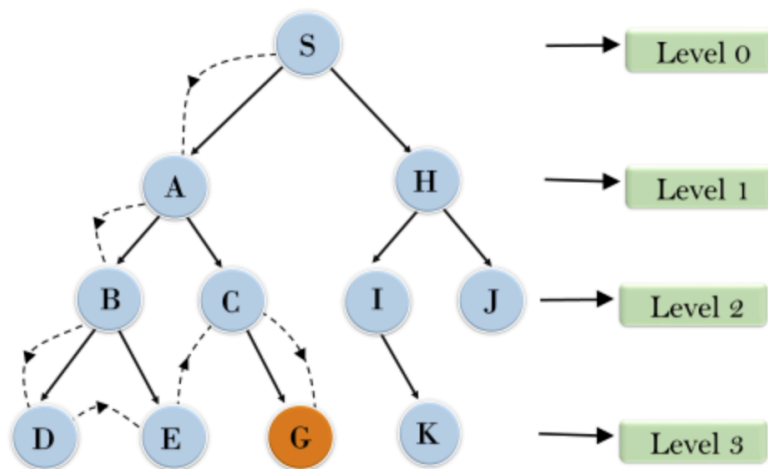
**Uniform Cost Search**



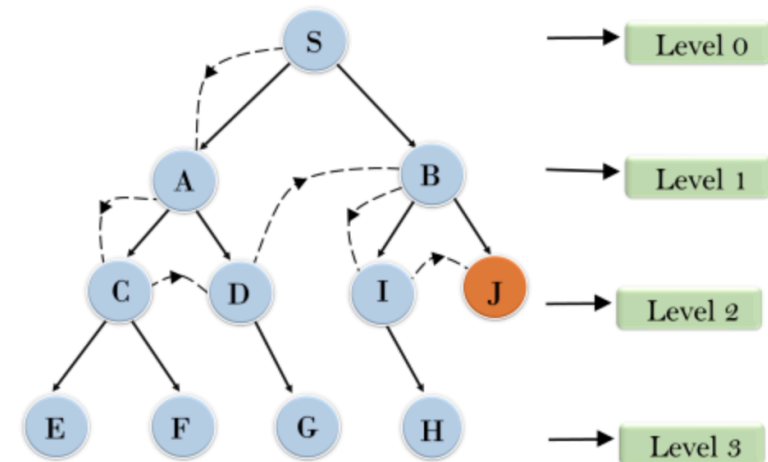
# Search Strategies

- **NON-INFORMED** search strategies:

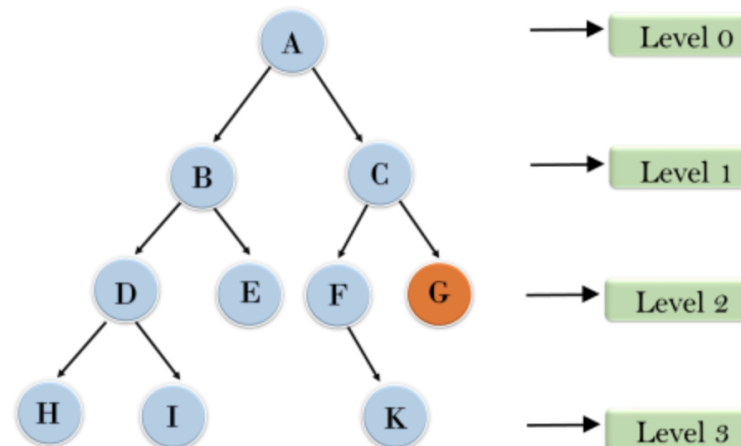
Depth First Search



Depth Limited Search



Iterative deepening depth first search



# Search Strategies

- **INFORMED** search strategies:


- Best first

1. Greedy   $f(n) = h(n)$

2.  $A^*$

3.  $IDA^*$

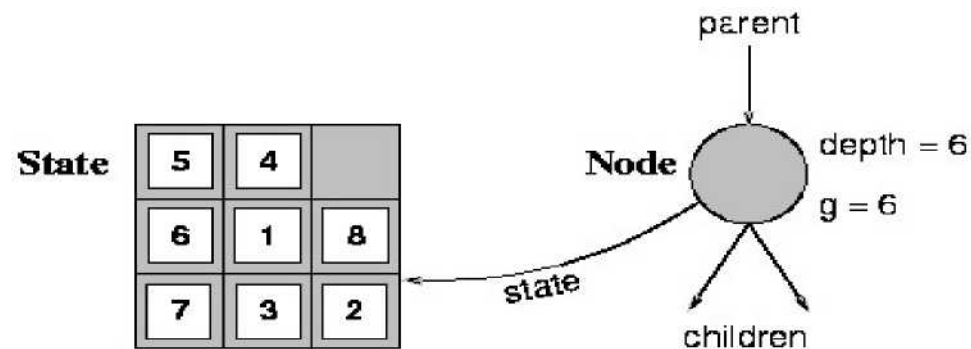
4.  $SMA^*$


$$f(n) = g(n) + h'(n)$$

where  $g(n)$  is the depth of the node, and  $h'(n)$  the estimated distance from the goal.

# Data structure for the search tree (structure of a Node)

- The state (in the state space) to which this Node corresponds.
- The parent Node.
- The action applied to get this node.
- The path cost for reaching this Node



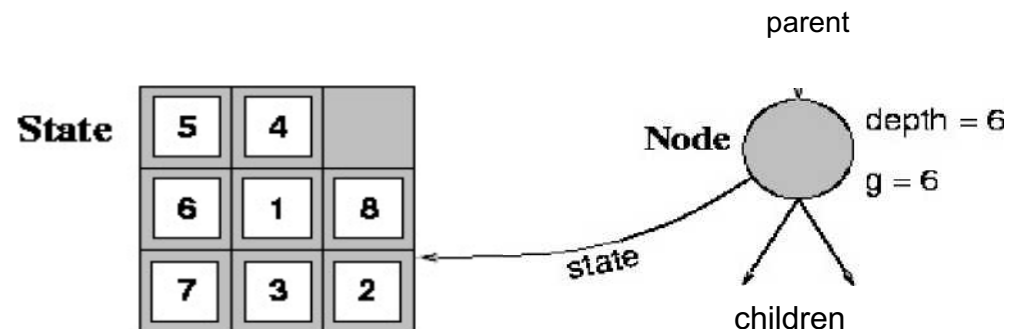


## The general search algorithm

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

# First Step: the Problem definition

- How to represent a "problem"?
  1. Usually, represented by means of "states", together with state operators (aka actions)
  2. there is a initial state
  3. a goal to reach (a property that should hold, i.e. a state for which the property holds)
- How to represent a state?
  - through data structures...



# The AIMA search library

- The AIMA LIBRARY provides two classes:
  - class Problem: our problem instance will extend this class
  - class Node: we will just access it...
- The class Problem is abstract (i.e., you must implement some methods)
- **def `_init_(self, initial, goal=None)`:**  
*"""The constructor specifies the initial state, and possibly a goal state, if there is a unique goal. Your subclass's constructor can add other arguments."""*
- However, the state must be a tuple (why? Graph search, the possibility of computing the hash.)

# The ALMA search library - the class Problem

- **`def actions(self, state):`**

*"""Return the actions that can be executed in the given state. The result would typically be a **list**, but if there are many actions, consider yielding them one at a time in an iterator, rather than building them all at once."""*

- **`def result(self, state, action):`**

*"""Return the state that results from executing the given action in the given state. The action must be one of `self.actions(state)` . """*

- **`def goal_test(self, state):`**

*"""Return True if the state is a goal. The default method compares the state to `self.goal` or checks for state in `self.goal` if it is a list, as specified in the constructor. Override this method if checking against a single `self.goal` is not enough."""*

## ...let's implement these methods...

- Which are the search strategies that I can apply?
  1. Breadth-first
  2. Depth-first
  3. Depth-bounded
  4. Iterated Deepening
- What about the path of a solution? the Node class keeps track of the path

# What about the Uniform Cost Strategy?

We need to keep into account also **the cost of the operators/actions** that we applied to reach the goal...

You should implement the method:

```
def path_cost(self, c, statel, action, state2):
```

```
    """Return the cost of a solution path that arrives  
    at state2 from statel via action, assuming cost c to get up  
    to statel. If the problem is such that the path doesn't  
    matter, this function will only look at state2. If the path  
    does matter, it will consider c and maybe statel and action.  
    The default method costs 1 for every step in the path."""
```

A default method is provided, which considers the constant cost 1 for every action

# Informed Strategies... ???

- Remember the notion of heuristic: a function that estimates (with a certain error) the distance of a state from the goal...
- You should implement the method:

```
def h(self, node):  
    """Returns the heuristic applied to the Node node"""
```

## Summing up...

1. Breadth-first
  2. Depth-first
  3. Depth-bounded
  4. Iterated Deepening
5. Uniform Cost
6. Greedy
7. A\*

`def path_cost(self, c, state1, action, state2)`

`def h(self, node)`

`def actions(self, state)`  
`def result(self, state,`  
`action)`  
`def goal_test(self, state)`

Notice: Applying intelligent strategies means a cost in terms of the knowledge you should provide...



# The library search.py

Given an implementation of the problem,  
provides the following functions:

- ```
def breadth_first_tree_search(problem) :  
    """Search the shallowest nodes in the search tree first.  
    Search through the successors of a problem to find a goal.  
    The argument frontier should be an empty queue.  
    Repeats infinitely in case of loops."""
```
- ```
def breadth_first_graph_search(problem):  
    " " " . . . " " "
```
- ```
def depth_first_tree_search(problem):  
    """Search the deepest nodes in the search tree first.  
    Search through the successors of a problem to find a goal.  
    The argument frontier should be an empty queue.  
    Repeats infinitely in case of loops."""
```
- ```
def depth_first_graph_search(problem):  
    """Search the deepest nodes in the search tree first.  
    Search through the successors of a problem to find a goal.  
    Does not get trapped by loops.  
    If two paths reach a state, only use the first one."""
```

# The library search.py

Given an implementation of the problem,  
provides the following functions:

- `def uniform_cost_search(problem):`
- `def depth_limited_search(problem, limit=50):`
- `def iterative_deepening_search(problem) :`
- `def astar_search(problem, h=None):`
- `def hill_climbing(problem):`
- `def simulated_annealing(problem, schedule=exp_schedule()):`
- `def genetic_search(problem, fitness fn, ngen=1000, pmut=0.1, n=20):`

# The library search.py

How to use it?

- define the class Problem, representing your problem
- import the file "**reporting.py**" (utilities...)
- Looking for a solution?

```
myP = MyProblem(...)
soln = breadth_first_tree_search(myP)
path = path_actions(soln)
print(path)
print("Cost: ", soln.path_cost)
path = path_states(soln)
print(path)
```

- Want to compare strategies?

```
report([
    breadth_first_tree_search,
    breadth_first_graph_search,
    # depth_first_tree_search,
    depth_first_graph_search,
    astar_search],
    [myP])
```

## Today...

1. Prepare a new python project
2. Choose the first problem, define the state and the functions, and test the different strategies
  - Missionaries and Cannibals
3. Define the state and the functions, and test the different strategies for the second and the third problem
  - U2
  - Fill the 10x10 matrix

# A simple problem:

## Missionaries and Cannibals

- 3 missionaries and 3 cannibals are on the same shore of a river, and want to cross such river. There is a single boat (on the same initial shore), able to bring around two persons at a time.
- Should it happen that in ANY shore there are more cannibals than missionaries, the cannibals will eat the missionaries (failure states).
- Which state representation?
  - State: a tuple of three numbers representing the number of missionaries, the number of cannibals, and the presence of the boat on the starting shore.



- The initial state is:  $(3,3,1)$

# A simple problem:

## Missionars and Cannibals

- Actions: the boat cross the river with passengers:
  - 1 missionary, 1 cannibal,
  - 2 missionaries,
  - 2 cannibals,
  - 1 missionary
  - 1 cannibal.
- Goal: state (0,0,0)
- Path cost: the number of river crossings.

## A second problem:

The U2 Bride (exam test --16 December 2005)

- U2 are giving a concert in Dublin.
- There are still 17 minutes. Unfortunately, to reach the stage, the members of the band must cross a small, dark, and dangerous bridge... do not despair! They have a torch!!! (only one)
- The bridge allows the passing of two persons at a time. The torch is mandatory to cross the bridge, and should be brought back and forth (cannot be thrown.). All the members are on the wrong side of the bridge, far from the stage.
- Every member of the U2 walks at a different velocity, and they takes different time to cross the bridge:
- Bono, 1 minute
- Edge, 2 minutes
- Adam, 5 minutes
- Larry, 10 minutes

# A second problem:

## The U2 Bride (exam test --16 December 2005)

- If two members cross the bridge together, it will take them the highest time to cross it (i.e., the faster will walk at the velocity of the slower).
- For example: if Bono and Larry will cross together, it will take them 10 minutes to get over the bridge. If Larry brings back the torch, another 10 minutes will pass, and the mission will be failed.
- Think about an heuristic function for this problem, and try to solve it using A\* over graphs. To limit the search space, suppose that the members move always in couple in one direction, and only one member brings back the torch.

How to represent the state?

Choice: state as a list of lists, indicating who is in which shore, and the presence of the torch

initial state is [ ["Bono", "Edge", "Adam", "Larry"], [ ], 1]

goal state will be [ [ ], ["Bono", "Edge", "Adam", "Larry"], 0]



# A second problem:

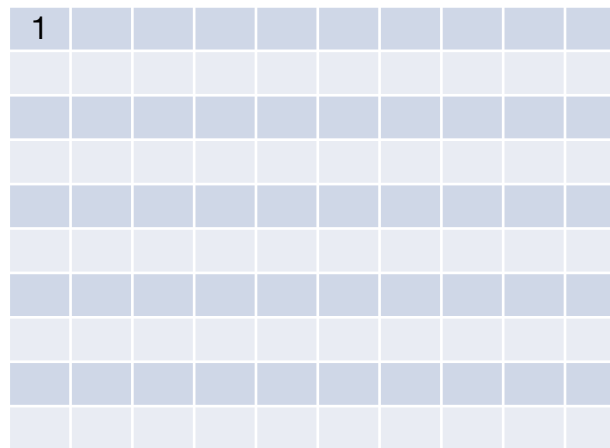
## The U2 Bride (exam test --16 Dicember 2005)

A suggestion for the heuristic function:

- At most, only two persons can move at a time. Let us group the members in couples. Sort the member in descending order on the base of the crossing time, and:
- put in the first group the two slower members
- put in the second group the remaining members
- Every group will move at the velocity of the slowest in the group. The heuristic function is the sum of the time required by each group still in the wrong side of the bridge.
- Is this heuristic function admissible? Will it find the best solution?

## A third problem: Fill a square

- A matrix of 10x10 cells is given.
- In the initial state, all the cells are empty, except the left upper corner, that is initialized to 1.



- Problem: assign a consecutive value to all the cells, starting from 1 up to 100, with the following rules:
- Starting from a cell with value  $x$ , you can assign value  $(x+1)$  to:
  - empty cells that are two cells away in vertical or horizontal direction, or
  - one cell away in diagonal direction.

## A third problem: Fill a square

- If the matrix is empty, a cell has 7 possible “next cells” to be filled branching factor
- when the matrix is partially filled, the number of free cell that are reachable decreases -> branching factor lowers
- The depth of the search tree is 100...