

Planning Lab - Lesson 1

Search Strategies

Alessandro Farinelli, Celeste Veronese

Slides realized by Luca Marzari

University of Verona
Department of Computer Science

Contact: celeste.veronese@univr.it

October 11, 2023



UNIVERSITÀ
di **VERONA**

Dipartimento
di **INFORMATICA**

What is it

Gym is a toolkit for developing and comparing autonomous agents, focusing specifically on reinforcement learning algorithms.

It supports teaching agents everything from walking to playing games like Pong or Pinball

What is it for

- An open-source collection of environments that can be used for benchmarks
- A standardized set of tools to define and to work with environments

Where to find it

<https://gym.openai.com>

During the lab lessons, we will use Jupyter notebook files. In order to use these files you should install the following dependencies.¹

Detailed guide for the installation process:

<https://github.com/vrncst/PlanningLab>

- Download the *Anaconda* package manager for Python 3.7 from <https://www.anaconda.com/distribution/#download-section>
- Install Conda on your system
- Open a terminal and digit:

```
> git clone https://github.com/vrncst/PlanningLab
> cd Planning-Lab
> conda env create -f tools/planning-lab-env.yml
> conda activate planning-lab
```

¹For help contact: celeste.veronese@univr.it

To open the tutorial:

- Navigate to your local Planning-Lab folder
- Ensure that you have activated the *planning-lab* conda environment and launch Jupyter Notebook (`> jupyter notebook`) from your folder
- Navigate with your browser to: *lesson_1/lesson_1_tutorial.ipynb*

Assignments

- Your assignments for this lesson are at: *lesson_1/lesson_1_problem.ipynb*
- You will be required to implement both some Uninformed Search and some Informed Search algorithms
- In the following you can find pseudocodes for such algorithms

Uninformed Search: tree and graph search versions

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

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

Search algorithms require a data structure to keep track of the search tree. A *Node* in the tree is represented by a data structure with three components:

Node(state, parent, pathcost)

- state: the state to which the node corresponds;
- parent: the node in the tree that generated this node;
- pathcost: the total cost of the path from the initial state to this node;
- depth: the depth of the node in the search tree. You do not need to initialize this, as it is automatically set by the constructor.

Breadth-First Search (BFS): graph search version

Require: *problem*

Ensure: *solution*

```
1: node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
2: if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
3: frontier  $\leftarrow$  NODE-QUEUE
4: explored  $\leftarrow \emptyset$ 
5: while not IS-EMPTY(frontier) do
6:   node  $\leftarrow$  REMOVE(frontier)
7:   explored  $\leftarrow$  explored  $\cup$  node.STATE
8:   for each action in problem.ACTIONS(node.STATE) do
9:     child  $\leftarrow$  CHILD-NODE(problem, node, action)
10:    if child.STATE not in explored or frontier then
11:      if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
12:    frontier  $\leftarrow$  INSERT(child)
return FAILURE
```

▷ Remove last node

Iterative Deepening Search (IDS): tree search version

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? \leftarrow false

for each *action* **in** *problem*.ACTIONS(*node*.STATE) **do**

child \leftarrow CHILD-NODE(*problem*, *node*, *action*)

result \leftarrow RECURSIVE-DLS(*child*, *problem*, *limit* - 1)

if *result* = *cutoff* **then** *cutoff_occurred?* \leftarrow true

else if *result* \neq failure **then return** *result*

if *cutoff_occurred?* **then return** *cutoff* **else return** failure

function ITERATIVE-DEEPENING-SEARCH(*problem*) **returns** a solution, or failure

for *depth* = 0 **to** ∞ **do**

result \leftarrow DEPTH-LIMITED-SEARCH(*problem*, *depth*)

if *result* \neq cutoff **then return** *result*

Uniform-Cost Search Example

In your notebook you can also find an implementation of the last uninformed search algorithm you have seen in class, the Uniform-Cost Search (UCS). The pseudocode is on the next slide.

Uniform-Cost Search (UCS): graph search version

Require: *problem*

Ensure: *solution*

```
1: node  $\leftarrow$  a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
2: frontier  $\leftarrow$  PRIORITY-QUEUE ordered by PATH-COST, with node as the only element
3: explored  $\leftarrow \emptyset$ 
4: loop
5:   if IS-EMPTY(frontier) then return FAILURE
6:   node  $\leftarrow$  REMOVE(frontier) ▷ Remove node with highest priority
7:   if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
8:   explored  $\leftarrow$  explored  $\cup$  node.STATE
9:   for each action in problem.ACTIONS(node.STATE) do
10:    child  $\leftarrow$  CHILD-NODE(problem, node, action) ▷ Increase path cost over parent
11:    if child.STATE not in explored or frontier then
12:      frontier  $\leftarrow$  INSERT(child, frontier)
13:    else
14:      if child.STATE is in frontier with higher path - cost then
15:        replace that frontier node with child
```

- You are now required to implement some informed search algorithms
- The pseudocodes are variations of the Uniform-Cost Search (UCS) where the *priority queue* is ordered by h and $f = g + h$ respectively