Planning Lab - Lesson 1

Uninformed Search

Luca Marzari and Alessandro Farinelli

University of Verona
Department of Computer Science

Contact: `luca.marzari@univr.it`

October 19, 2022

The OpenAI Gym Framework

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

Installation Process

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

Detailed guide for the installation process:

https://github.com/LM095/Planning-Lab

- 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/LM095/Planning-Lab
> cd Planning-Lab
> conda env create -f tools/planning-lab-env.yml
> conda activate planning-lab
```

[1]For help contact: luca.marzari@univr.it

Tutorial

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 some Uninformed 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

Node data structure

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 this is automatically set by the constructor.

Breadth-First Search (BFS): graph search version

Require: $problem$
Ensure: $solution$

$node \leftarrow$ a node with $\textsc{State} = problem.\textsc{Initial-State}, \textsc{Path-Cost} = 0$
if $problem.\textsc{Goal-Test}(node.\textsc{State})$ **then return** $\textsc{Solution}(node)$
$frontier \leftarrow \textsc{Node-Queue}$
$explored \leftarrow \emptyset$
while not $\textsc{Is-Empty}(frontier)$ **do**
    $node \leftarrow \textsc{Remove}(frontier)$ ▷ Remove last node
    $explored \leftarrow explored \cup node.STATE$
    for each $action$ **in** $problem.\textsc{Actions}(node.\textsc{State})$ **do**
        $child \leftarrow \textsc{Child-Node}(problem, node, action)$
        if $child.\textsc{State}$ **not in** $explored$ **or** $frontier$ **then**
            if $problem.\textsc{Goal-Test}(child.\textsc{State})$ **then return** $\textsc{Solution}(child)$
            $frontier \leftarrow \textsc{Insert}(child)$
return $\textsc{Failure}$

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? ← 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
```

```
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
```