AI Lab - Session 1

Search Algorithms

Riccardo Sartea

University of Verona
Department of Computer Science

March 15^{th} 2018

UNIVERSITÀ di VERONA
Dipartimento di INFORMATICA

Outline

1. Introduction
   - OpenAI Gym
   - Environment Characteristics
   - Installation
   - Guidelines
2. Practice

The OpenAI Gym Framework

What is it

Gym is a toolkit for developing and comparing 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

Environments for the First Session

- In this first lab session all the environments are grid-worlds
- Every cell has its (x, y) coordinates and a content, e.g., start (S), wall (W), goal (G)...
- A state is identified by an integer
- An action is identified by an integer

Environment Properties

Variables:

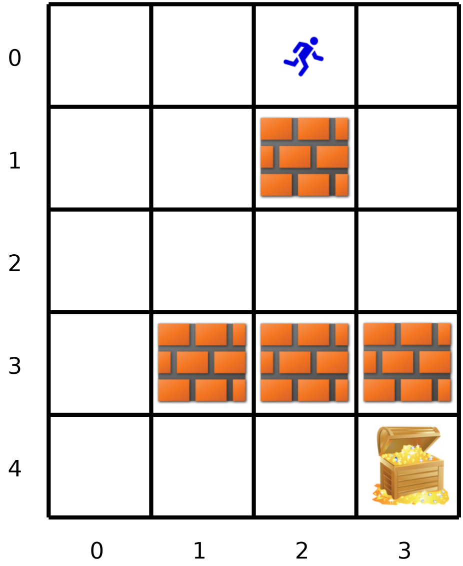
- *action_space* - the space of possible actions: usually a range of integers [0, ..., n]
- *observation_space* - the space possible observations (states): usually a range of integers [0, ..., n]
- *startstate* - start state (only one)
- *goalstate* - goal state (only one)
- *grid* - flattened grid (1-dimensional array)

Environment Methods

Methods:

- $render()$ - renders the environment
- $sample(state, action)$ - returns a new state sampled from the ones that can be reached from $state$ executing $action$
- $pos\_to\_state(x, y)$ - returns the state given its position in x and y coordinates
- $state\_to\_pos(state)$ - returns the coordinates x and y of a state

The SmallMaze Environment



Installation Process

Listing 1: Installation

```
mkdir aicourse
cd aicourse
git clone https://github.com/openai/gym
cd gym
pip3 install --user -e .
cd ..
git clone https://github.com/SaricVr/gym-ai-lab
cd gym-ai-lab
pip3 install --user -I --no-deps -e .
cd ..
git clone https://github.com/SaricVr/ai-lab
```

Guidelines I

- You will be working within the **ai-lab** repository
- Each algorithm to implement has its already available prototypes of the functions, e.g., [search.algorithms.ids(problem, stype)]
- In order to learn how to use the main features of OpenAI Gym environments you can look at some example code in [session1/examplecode.py]
- Different fringe datastructures are already at your disposal [datastructures.fringe]. $PriorityFringe$ handles priority using $FringeNode.value$ property
- Every step, i.e., moving from a state to another, has cost 1. $FringeNode.pathcost$ then will always be equal to the depth of such node
- Use the L_1 norm as distance heuristic between a state and the goal [search.heuristics.l1_norm(p1, p2)]

Guidelines II

- Files [session1/graph_search.py] and [session1/tree_search.py] are the "main" entry point for graph search and tree search algorithms respectively, i.e., the runnable files
- The solution of your search algorithms must be sequences of state identifiers (integers) or $None$
- Along with the solution, you must also return the stats (time, nexp, maxmem) of you algorithms:
  - Time elapsed between the start and the end of the algorithm
  - Number of states expanded
  - Maximum number of states in memory at the same time (fringe + closed)
- Validate carefully the solutions and also the stats returned by your code

Outline

Iterative Deepening Search (IDS)

```
function ITERATIVE-DEEPENING-SEARCH(problem)
    for depth ← 0 to ∞ do
        result ← DEPTH-LIMITED-SEARCH(problem, depth)
        if result ≠ CUTOFF then return result

function DEPTH-LIMITED-SEARCH(problem, limit)
    return RECURSIVE-DLS(MAKE-NODE(problem.INITIAL-STATE), problem, limit)

function RECURSIVE-DLS(node, problem, limit)
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    if limit = 0 then return CUTOFF
    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
    return FAILURE
```

Note: this is a tree search version

Task 1

Implement the IDS (iterative deepening search) algorithm based on tree search. Work with the *SmallMaze-v0* environment. Proceed by steps:

1. Implement the search.algorithms.dls_ts(problem, limit) procedure. Test it and debug it with different limit values
2. Implement the search.algorithms.ids(problem, stype) procedure based on search.algorithms.dls_ts(problem, limit)

Task 2

Implement the IDS (iterative deepening search) algorithm based on graph search [search.algorithms.dls_gs(problem, limit)] version. Work with the *SmallMaze-v0* environment.

Are there any differences in results between task 1 and task 2? If so, why?

Uniform-Cost Search (UCS)

Input: *problem*
Output: *solution*

```
node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
fringe ← PRIORITY-QUEUE ordered by PATH-COST, with node as the only element
closed ← ∅
loop
    if IS-EMPTY(fringe) then return FAILURE
    node ← REMOVE(fringe)                                   ▷ Remove node with highest priority
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    closed ← closed ∪ node
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE not in fringe and child.STATE not in closed then
            fringe ← INSERT(child, fringe)
        else if child.STATE in fringe with higher PATH-COST then
            replace that fringe node with child
```

Note: this is a graph search version

Task 3

Implement the UCS (uniform-cost search) algorithm based on graph search. Work with the *SmallMaze-v0* environment. Proceed by steps:

1. Implement the general search.algorithms.graph_search(problem, fringe, f) procedure
2. Verify that search.algorithms.ucs(problem, stype) with $stype = search.algorithms.graph\_search$ works as expected

Task 4

Implement the UCS (uniform-cost search) algorithm based on tree search. Work with the *SmallMaze-v0* environment. Proceed by steps:

1. Implement the general search.algorithms.tree_search(problem, fringe, f) procedure
2. Verify that search.algorithms.ucs(problem, stype) with $stype = search.algorithms.tree\_search$ works as expected

Are there any differences in results between task 3 and task 4? If so, why?

Task 5

Analyze and run the code of the BFS[1] (breadth-first search) algorithm in both graph search and tree search versions. Work with the *SmallMaze-v0* environment.
What can you say about the results?

[1] Remember that the BFS does not use any evaluation function

Task 6

Implement the following algorithms based on
search.algorithms.graph_search(problem, fringe, f) and
search.algorithms.tree_search(problem, fringe, f):

- Greedy best-first [search.algorithms.greedy(problem, stype)]
- A* best-first [search.algorithms.astar(problem, stype)]

Work with the *SmallMaze-v0* environment.
hint: Have a look at the implementation of UCS in algorithms.py to see how you can define and use a cost function.
Modify this to be the heuristic for Greedy and the heuristic + cost for A*.
You can find already implemented heuristics in heuristics.py

Task 7

Run all of your your search algorithms (graph and tree based versions) on the *SmallMaze-v0*, *GrdMaze-v0* and *BlockedMaze-v0*. Discuss the results: which are optimal? Which are not and why?

Compare your results

You can compare the results achieved by your solution with the one provided by us.
Have a look at the following files:

- Results for tree search
- Results for graph search

Alternative you can update the git repository and then look at the files

`session1/graph_search_results.txt` and
`session1/tree_search_results.txt`

To update the repository do the following:

```
cd ai-lab
git commit -a -m "a message"
git pull
```