**Artificial Intelligence**
**AIC_4301C**
**TP1**
2022-2023

1. This lab uses python 3.
2. You should execute your code in a linux environment by executing your linux commands on a terminal.

## Configuration and tests

1. Create a folder TP1_AIC_4301C_names-of-the-group-members
2. If you are using your own laptop:
   (a) In this folder download the file **requirements_AIC.txt** from Blackboard.
   (b) Execute pip install -r requirements_AIC.txt
3. Download the project **search_AIC.zip** from Blackboard, unzip it.
4. In the folder search_AIC execute python3 pacman_AIC.py
5. If you have the message if No module named 'tkinter' using your own laptop, execute sudo apt-get install python3-tk
6. Execute python3 pacman_AIC.py −−layout testMaze −− pacman GoWestAgent

   You should have in the terminal:

   ```
   Pacman emerges victorious! Score: 503
   Average Score: 503.0
   Scores: 503.0
   Win Rate: 1/1 (1.00)
   Record: Win
   ```
7. To see the list of all options and their default values use:

   ```
   python3 pacman_AIC.py -h
   ```
8. The commands that appear in the TPs are in **commands_AIC.txt**, you can run all these commands in order with:

   ```
   bash commands_AIC.txt
   ```

## First Agent: Goal Based Agent = Search Agent

- In this lab you will have to implement and test search algorithms used by a search agent to solve a Position Search Problem.
- A search problem defines:
  1. the state space that consists of (x,y) positions in a pacman game,
  2. start state,

3. goal test,
4. successor function,
5. cost function.

- The Position Search problem is defined in the python file **searchAgents.py** in the **class PositionSearchProblem(search.SearchProblem)**.
- To understand the state representation used in the Position Search Problem, read the class PositionSearchProblem(search.SearchProblem).
- The goal of this search problem is to find a Fixed Food Dot. We will implement in this lab three search algorithms to solve this search problem: Depth-first search (exercise 1), Breath-first search (exercise 2) and Uniform-Cost Search (exercise 3).

Your search functions have to be implemented in the file search.py with:

1. All of your search functions need to return a list of actions that will lead the agent from the start to the goal.
2. Make sure to use the Stack, Queue and PriorityQueue data structures provided in util.py.

## Python Files

In the folder **search_AIC** you will find the following python files:

1. search.py: Where all of your search algorithms will reside. You will have to write the code of this TP in this file.
2. searchAgents.py: Where all of your search-based agents will reside.
3. pacman_AIC.py The main file that runs Pacman games. This file describes a Pacman GameState type, which you use in this TP.
4. game.py The logic behind how the Pacman world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
5. util.py Useful data structures for implementing search algorithms. We encourage you to look through util.py for some data structures that may be useful in your implementations.
6. You can ignore all other .py files.

## Exercise 1: Depth-first search (DFS)

1. Implement the depth-first search (DFS) algorithm in the **depthFirstSearch** function in **search.py**. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states. Your search algorithm needs to **return a list of actions** that reaches the goal.
2. Test your code:

   ```
   python3 pacman_AIC.py -l tinyMaze -p SearchAgent
   python3 pacman_AIC.py -l mediumMaze -p SearchAgent
   python3 pacman_AIC.py -l bigMaze -z .5 -p SearchAgent
   ```

## Exercise 2: Breadth-first Search (BFS)

1. Implement the breadth-first search (BFS) algorithm in the **breadthFirstSearch** function in **search.py**. Write a graph search algorithm that avoids expanding any already visited states. Your search algorithm needs to **return a list of actions** that reaches the goal.

2. Test your code:

   ```
   python3 pacman_AIC.py -l mediumMaze -p SearchAgent -a fn=bfs
   ```

   ```
   python3 pacman_AIC.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
   ```

## Exercise 3: Uniform Cost Serach (UCS)

1. Implement the uniform-cost graph search algorithm in the **uniformCostSearch** function in **search.py**.

2. Test your code:

   ```
   python3 pacman_AIC.py -l mediumMaze -p SearchAgent -a fn=ucs
   ```

   ```
   python3 pacman_AIC.py -l mediumDottedMaze -p StayEastSearchAgent
   ```

   ```
   python3 pacman_AIC.py -l mediumScaryMaze -p StayWestSearchAgent
   ```

## Exercise 4: Finding All the Corners Problem

In corner mazes, there are four dots, one in each corner.

Our **new search problem** is to find the shortest path through the maze that touches all four corners (whether the maze actually has food there or not).

1. Implement the **CornersProblem** search problem in **searchAgents.py** in class CornersProblem(search.SearchProblem).

   You will have to choose a state representation that encodes all the information necessary to detect whether all four corners have been reached. This state representation will be implemented in:

   (a) def __init__(self, startingGameState),

   (b) getStartState(self),

   (c) isGoalState(self, state),

   (d) getSuccessors(self, state)

2. Test your code:

   ```
   python3 pacman_AIC.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
   ```

   ```
   python3 pacman_AIC.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
   ```