Final Project Assignment
ROUTE SEARCH IN LOGISTICS PROBLEMS

Artificial Intelligence
Computer Science Degree
Spring 2019

In this project you will work with search algorithms to be applied in a simulated environment where a delivery person has to deliver pizzas to different predefined locations. The problem environment is discretized as shown in Figure 1. The delivery person starts his route from his base, where he has to return once all the pizzas are delivered. Tiles depicted as green buildings are those where the delivery person has to deliver the pizza. Tiles depicted as a pizza are restaurants where the delivery person can stock up on pizzas. The environment may have different types of terrains.

The programming language is Python. For purposes of this work, we will use the library SIMPLE-AI 0.8.1 (`https://pypi.python.org/pypi/simpleai/`), which implements many artificial intelligence algorithms, including uniformed and informed search algorithms.

The project consists of two parts:

- Part I (7p): a basic part that consists of solving a problem in different environments using different search algorithms.
- Part II (3p): an advanced part that consists of including additional characteristics to the simplified problems, such as types of terrain, number of orders per client, maximum load capacity, etc.

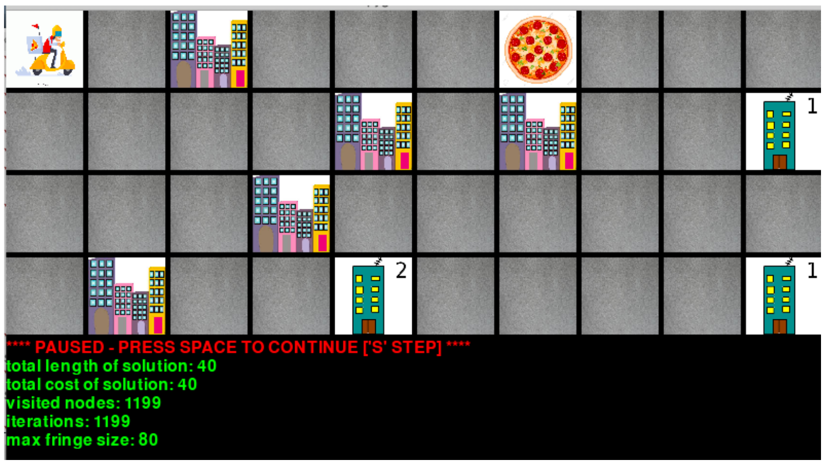
1 Basic problem (7p)

The idea is to solve a search problem where a delivery person rides his bike from his base to different locations to deliver pizzas; then he rides back to his base. For purposes of this part, you must take into consideration that (1) there is only one restaurant, (2) there is only one client, who can make two orders at most; (3) the maximum load capacity of the delivery person's bike is two; (4) the delivery person cannot ride through tiles different than delivery locations or restaurants; and (5) all actions have unit-cost. The total cost of the solution is the total number of actions performed by the delivery person (movements between tiles, loading pizza, and unloading pizza).

Student's task

Your task consists of two parts:

1. Design and implementation: you will design and implement the problem representation (state and actions), a non-trivial heuristic function, and other additional functions required by the search module SIMPLE-AI. Additionally, you must run different search algorithms to solve the problem in order to check that the implementation is correct.
2. Experimental evaluation and comparison:



```
**** PAUSED - PRESS SPACE TO CONTINUE ['S' STEP] ****
total length of solution: 40
total cost of solution: 40
visited nodes: 1199
iterations: 1199
max fringe size: 80
```

Figure 1: Problem example: the delivery person is located at his base; green buildings represent delivery locations and the number of orders in each building; the pizzas represent restaurant locations; gray tiles represent roads; other buildings represent apartment complexes that the delivery person cannot cross.

- An experimental evaluation to solve the basic problem by using different search algorithms. The SIMPLE-AI library includes, among others, the following search algorithms: *breadth_first*, *depth_first*, *limited_depth_first*, *astar* (an example of a call to any of these algorithms is included in the given code). You will do a comparison among the different search algorithms considering number of expanded nodes, cost of the solution, and memory usage. Information about the solution is shown at the end of the simulation.
- Development of at least three new scenarios that you consider interesting. Then, you will repeat the previous process to compare them. You can: evaluate the same scenario with and without terrain types representing obstacles, modify the size of the grid, number of goals and their locations, etc. You must analyze the impact of these changes in the solution of the problem.

Software installation

To develop this project, you need to install Python 2.7 and the packages `pygame` and `pydot`[1]. To install the software, unzip `201819_AI_software.zip`. To execute it, open a command line terminal and type `python startGame.py`.

The files that you must modify are in the `student` folder, which has the following files:

- `config.py`, which defines the initial settings for the problem.
- `gameProblem.py`, which contains the functions required by the search module SIMPLE-AI and that must be implemented.
- `map.txt`, which defines the configuration of the problem.

Code description

You will implement some functions required by the search module SIMPLE-AI. You will do that in the `gameProblem.py` file given, which contains a class with the same name and the definition of the functions that need to be implemented (indicated with comments in the code).

The class *gameProblem* represents the search problem. It has a list of attributes (described in the Appendix) that you can use to access the grid, types of terrains, etc. These variables can be consulted, but cannot be modified.

[1]These are already installed in the machines at the university.

The functions you need to implement are the following:

- *actions(s)*: function that receives a state *s* and returns a list of applicable actions in *s*.
- *result(s_1,a)*: function that receives a state s_1 and an action *a* and returns a state s_2, which is the result of applying *a* in s_1.
- *is_goal(s)*: functions that receives a state *s* and returns *True* if *s* is a goal state. Otherwise, it returns *False*.
- *cost(s_1,a,s_2)*: function that receives two states s_1 and s_2 and an action *a*, and returns the cost of achieving s_2 from s_1 applying *a*.
- *heuristic(s)*: function that receives a state *s* and returns the heuristic value of *s*; that is, the estimated cost of reaching the goal state from *s*.
- *setup()*: function that configures the initial state, the goal state, and the search algorithm to use.
- *printState (s)*: function that receives a state *s* and prints its value in the interface.
- *getPendingRequests (s)*: functions that receives a state *s* and returns the number of pending orders, if *s* is a delivery location. Otherwise, it returns `None`.

2 Advanced problem (3p)

The idea is to perform a similar experimental evaluation as before, but using problems with additional characteristics. To include them, you will need to make some changes in the design and implementation developed in Part I. We propose to include the following:

- **Terrain with costs**. Each terrain type will have a different cost of riding through it. This cost could represent, for instance, the time consumed (it is easier to ride on a flat road that on a hill). These costs could be modified in the configuration file.
- **Cost per load**. Consider that the cost of the actions depends on the actual load.
- **Heuristics**. Define different heuristics and evaluate them.

3 Submission

This project should be done in pairs. You must submit a zip file through a link that will be posted on Aula Global. Note that, if you work in pairs, only one submission is needed. The zip file name must be *pr-ai-2019-students-code.zip*, where *students-code* is the last six digits of each students's NIA (e.g., pr-ai-2019-123456-654321.zip). The zip file will consist of:

- An essay containing the following sections (PDF file):
  1. Introduction
  2. For each part:
     - Description and explanation of the tasks performed within the project: problem representation and implemented functions.
     - Description and explanation of the performed tests and the results obtained from such tests.
  3. Conclusions: technical comments related to the development of this project.
  4. Personal comments: difficulties, challenges, benefits, etc.
- A directory, namely pr-ia-2019/student-code, containing the source code, including the parts you have implemented and the other folders needed to run the project.

The deadline for the submission of the project is on May 12th at 11.55pm!

Appendix: software documentation

The given software consists of the following elements:

1. The SIMPLE-AI library, which implements the search algorithms.
2. The GAME-IA library, which is the graphic component. It consists of several files that must not be modified.
3. The GAMEPROBLEM class, which is the class where you have to do your implementation.

To execute the game open a terminal and type `python startGame.py`

The graphical component interacts with the student's component in the following way:

1. During the initialization of the game, and object of *gameProblem* is created.
2. Then, the function *setup()* is invoked; here you must specify the initial state, the goal state, and the search algorithm.
3. The search is performed using the search algorithm specified and returns a solution, if there is one. Then, the output shows the statistics for the performed search process.
4. The graphical component performs a simulation to show the solution found. To do that, it simulates the solution using the predefined actions: **'North'**,**'East'**,**'South'**,**'West'** (the graphical component ignores any other action).

The output shows information about the performed search; it could be useful for the experimental part.

```
total length of solution: 47
total cost of solution: 51
visited nodes: 8001
iterations: 8001
max fringe size: 554
```

You could save this output in a file.

Configuration file

The configuration file *config.py* defines the configuration parameters of the problem. They are described in Table 1. You can modify the values of the fields of this file. Table 2 describes the fields of a tile. The *basic* tile is defined in *basicTile*. You can access the attributes by calling the method GETATTRIBUTE(POSITION, KEY).

Each tile has two different versions that have identical attributes. One of them has the prefix *-traversed*; it is used by the simulator to change the color of the tile when the delivery person rides through it. Tiles tagged as *traversed* are not considered during the search process.

The *gameProblem* class

Table 3 describes the attributes of the *gameProblem* class, which has the definition of the finctions that you must implement.

Table 1: Description of fields of the *configuration* file

Attribute	Type	Description
“text_size”	Int	Font size
“tile_size”	Int	Cell size
“type”	“random”	Generates a random grid
	“load”	Loads a grid from a file
“seed”	Int	Seed for the random number generator
“file”	String	File to load or save a grid
“save”	Boolean	True, save the grid to a file
“map_size”	Tuple (Int, Int)	Grid size (column, row)
“delay”	Float	Simulation speed (0.1)
“debugMap”	Boolean	True, shows the MAP variable
“basicTile”	String (key)	default tile
“maxBags”	Int	Maximum load capacity
“agent”	Dict.	Information about the agent
“agent[’start’]”	Tuple (Int, Int)	Initial position of the agent (column, row)
“maptiles”	Dict.	Cells configuration
“debug”		Do not modify
“hazards”		Do not modify

Table 2: Description of fields in *tile*

Attribute	Type	Description
graphics	String	File containing the graphical image for the tile
id	String	Id of the tile in MAP and POSITIONS.
marker	Char	Character used to symbolize the tile when loading or reading a grid from a file
num	Int	Number of tiles of this types in the grid when randomly generated
state	Dict.	Number of tiles of this type in the grid
		“agent” initialized to `None`
attributes	Dict.	Dictionary with attributes such as:
		“cost” indicates de cost of the tile
		“objects” indicates the number of orders

Table 3: Description of attributes in *gameProblem*

Attribute	Type	Description
MAP	List	Map of the grid. Each tile is a list of:
	[0]	Type of tile
	[1]	Id
	[2]	Attributes of the tile, loaded from the configuration file
POSITIONS	Dict.	It contains the keys to all types of tiles of the grid
	key	Type of tile
	value	List of the positions for the tiles of the type in key
CONFIG	Dict.	Configuration from configuration file
AGENT_START	Tuple	Initial position of the agent (from configuration file)
	[0]	Coordinate X
	[1]	Coordinate Y
INITIAL_STATE	Student definition	Initial state, loaded during the initialization phase
SHOPS	Student definition	Restaurant positions, loaded during the initialization phase
CUSTOMERS	Student definition	Clients positions, loaded during the initialization phase
MAXBAGS	Int	Maximum load capacity, loaded during the initialization