Trabalho prático individual nº 1

Inteligência Artificial / Introdução à Inteligência Artificial
Ano Lectivo de 2021/2022

19-20 de Novembro de 2021

I Important remarks

1. This assignment should be submitted via *GitHub* within 28 hours after the publication of this description. The assignment can be submitted after 28 hours, but will be penalized at 5% for each additional hour.

2. Complete the requested functions in module `"tpi1.py"`, provided together with this description. Keep in mind that the language adopted in this course is Python3.

3. Include your name and number and comment or delete non-relevant code (e.g. test cases, print statements); submit only the mentioned module `"tpi1.py"`.

4. You can discuss this assignment with colleagues, but you cannot copy their programs neither in whole nor in part. Limit these discussions to the general understanding of the problem and avoid detailed discussions about implementation.

5. Include a comment with the names and numbers of the colleagues with whom you discussed this assigment. If you turn to other sources, identify those sources as well.

6. All submitted code must be original; although trusting that most students will do this, a plagiarism detection tool will be used. Students involved in plagiarism will have their submissions canceled.

7. The submitted programs will be evaluated taking into account: performance; style; and originality / evidence of independent work. Performance is mainly evaluated concerning correctness and completeness, although efficiency may also be taken into acount. Performance is evaluated through automatic testing. If necessary, the submitted modules will be analyzed by the teacher in order to appropriately credit the student's work.

II Exercices

Together with this description, you can find the module `tree_search`, similar to the one initially provided for the practical classes, but with small changes and additions, namely:

- All generated nodes are stored in the list `all_nodes`.
- The position of each node in that list is used as a unique identifier of the node. Therefore, the parent of a node is also identified by the respective integer identifier.
- Terminal and non-terminal nodes are being counted.
- There is a new search strategy `"rand_depth"` which uses a pseudo-random approach to produce diferent results each time it is invoked with a different seed.

You can also find in this assignment the module `cidades` containing the `Cidades(SearchDomain)` class that you already know.

Don't change the `tree_search` and `cidades` modules.

The module `tpi1_tests` contains several test cases. If needed, you can add other test code in this module.

Module `tpi1` contains the classes `MyNode(SearchTree)`, `MyTree(SearchTree)` and `MyCities (Cidades)`. In the following exercices, you are asked (or may need) to complete certain methods in these classes. All code that you need to develop should be integrated in the module `tpi1`.

1. Develop a method `maximum_tree_size(depth)` in the `MyCities` class which, given a certain depth, estimates the maximum size of a tree with that depth. In order to produce this estimate, you should use the average number of neighbors per state as an estimate for average branching factor. Assume also that there is no loop prevention (therefore, states can be repeated along a path in the search tree).

2. Implement the `astar_add_to_open(lnewnodes)` method to support the A* search strategy. Create a method `search2()` in `MyTree` similar to the one in `SearchTree`, but supporting A* search.

3. Implement the `repeated_random_depth(num_attempts)` method which carries out a certain number of attempts (`num_attempts`) of `'rand_depth'` search and returns the best obtained result. The best tree should be stored in `self.solution_tree`

4. Add code in the `search2()` method to maintain in each node a new attribute `eval`. When the node is created, this attribute receives the value of the A* evaluation function. After expansion of a node, `eval` should always contain the lowest value of the evaluation function in a leaf (terminal node) under that node. In order to achieve this, develop the method `propagate_eval_upwards(node)` which, upon expansion of `node`, updates its `eval` attribute and propagates this update to its ancestors.

5. Add code in the `search2()` method to perform graph search when the `atmostonce` input flag is `True`. As you know, graph search is a kind of tree search that keeps only the best occurrence of each state, i.e., the occurrence where the state has the lowest acumulated cost. Therefore, each state occurs at most once in the generated tree.

6. One way to optimize solutions produced by tree search is to perform successive passes through the solution, replacing parts of the solution with "shortcuts". Develop a method make_shortcuts() in the MyTree class which, given a solution previously stored in self . solution , tries to produce a better (but not necessarily optimal) path. In each iteration, the function walks the path from left to right trying to detect states S_i and S_j, where $j - i > 1$, for which there is a direct transition from S_i to S_j.

   For example, in the path

```
['Porto','Aveiro','Figueira','Coimbra','Leiria']
```

we see there is a shortcut (direct connection) from the second state (Aveiro) to the fourth state (Coimbra). Therefore, we can replace the sub-path

```
['Aveiro','Figueira','Coimbra']
```

by the direct connection, obtaining a new path:

```
['Porto','Aveiro','Coimbra','Leiria']
```

The procedure that detects replaceable subpaths is repeated until no subpaths are detected under these conditions. Finally, the function returns the optimized path.

The function should also store the performed optimizations, in the form of a list of tuples (S_i, S_j), in `self.used_shortcuts`.

III Clarification of doubts

This work will be followed through `http://detiuaveiro.slack.com`. The clarification for the main doubts will be added here.

1. ... **Resposta**: