Trabalho prático individual nº 1

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

5-6 de Novembro de 2022

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`. You can also find attached the modules `cidades` and `strips`, containing the `Cidades(SearchDomain)` and `STRIPS(SearchDomains)`

classes. These modules are similar to the ones 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.
- The methods of the `Cidades` search domain are implemented based on a set of functions provided separately. These functions will be used in some of the exercices.

Don't change the `tree_search`, `cidades` and `strips` 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)`, `MyCities(Cidades)` and `MySTRIPS(STRIPS)`. In the following exercices, you are asked to complete certain methods in these classes. All code that you need to develop should be integrated in the module `tpi1`.

1. Implement auxiliary function `func_branching()` which returns an estimate of the branching factor in `Cidades` problems. This estimate is given by the average number of neighboor cities computed over all cities, substracting 1.
2. Create a new method `search2()` in class `MyTree`, similar to the original search method, and make sure that nodes (class `MyNode`) have attributes `cost`, `heuristic` and `depth`, with the usual meaning. In addition, make sure that the branching estimate (from ex. 1) is stored in `self.problem.domain`.
3. Using classes may be computationally heavier than using alternative data structures, such as tuples. Modify `MyTree()` so that it allows to choose an optimized node representation. If `MyTree` argument `optimize` is equal to 0 (zero), this means there is no optimization, and `MyNode` is used as usual. However, if `optimize==1`, nodes should be represented as tuples `(state,parent,cost,heuristic,depth)`, while providing identical search results, hopefully faster.
4. Problems and search domains are also represented by classes in `tree_search`. Let's allow `MyTree` to use tuples for this purpose as well. If `optimize==2`, nodes are represented by tuples, as before. In addition, domains are represented by tuples `( f_actions , f_result , f_cost , f_heuristic , f_satisfies ,branching)`, where `branching` is the branching estimate and the other elements are functions that compute actions, results, costs, heuristic values and goal satisfaction. And problems are represented by tuples `(domain,initial,goal)`, where the domain is a tuple. Did this produce a visible improvement?
5. Let's improve `MyTree` further by using the graph search algorithm, which ensures that no state appears more than once in the tree. When `optimize==4`, nodes, problems and domains are represented by tuples and no repeated states are allowed in the whole search tree.
6. Implement function `astar_add_to_open(lnewnodes)` which adds the new nodes to the queue according to the A* criterion. Make sure this works for any of the node representations.

7. Sometimes, it is worth working with some limitation on the number of nodes in the queue of open (terminal) nodes. Develop a method forget_worst_terminals () which should remove from the queue those nodes with lowest A* evaluation function. For this purpose, the `MyTree` constructor receives an optional argument, `keep`, that specfies the fraction of nodes that should remain in the queue. Based on that, the following number of nodes should be kept: $NumKeep = keep * max\_nodes\_given\_depth$, where $max\_nodes\_given\_depth$ is given by the number of nodes in a tree with a complete level at depth d and no nodes further down. To compute that, use the branching estimate previously computed. In addition, since the search is dynamic, with various terminal nodes at different depths, you should use for d the average depth of all terminal nodes currently in the queue. This function is used for the IBA* (Incrementally Bounded A*) strategy. It must be called at the end of each iteration of the main search loop.

8. Develop a method simulate_plan (state , plan) in the MySTRIPS class that, given a STRIPS state and a plan (sequence of STRIPS actions presented as a list), computes the final state, after the plan is fully executed.

III Clarification of doubts

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