Trabalho prático individual nº 1

Inteligência Artificial / Introdução à Inteligência Artificial
Ano Lectivo de 2017/2018

26 de Outubro de 2018

I Observações importantes

1. This assignment should be submitted via *Moodle* within 32 hours after the publication of this description. The assignment can be submitted after 32 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 module `tree_search`, similar to the one used in practical classes, with small changes and additions. In particular, some extra parameters were

added to the constructor of class SearchNode, which are stored without any processing. If needed, these parameters can be used to store additional data in the search tree.

The module `tpi1_tests` contains the `Cidades` class and the `cidades_portugal` search domain, which you already know. This domain is used for testing. The module already contains several tests. If needed, you can add other test code in this module.

Don't change the `tree_search` module. Module `tpi1` contains the classes `TSP` and `MyTree`. In the following exercices, you are asked to complete certain methods of these classes. All code that you are asked to develop should me integrated in the same module.

1. Create a search method search2() similar to the original method search() of class SearchTree, and add code to assign values to the following attributes of the search tree:

   - self . solution_cost - Total cost of the found solution.
   - self . tree_size - Total number of nodes of the generated tree.

   Example:

```
>>> p = SearchProblem(cidades_portugal,'Beja','Viseu')
>>> t1 = MyTree(p,'breadth')
>>> t1.search2()
['Beja', 'Evora', 'Santarem', 'Castelo Branco', 'Viseu']
>>> t1.tree_size,t1.solution_cost
(186, 561)
```

2. In SearchTree, implement methods `uniform_add_to_open()` and `astar_add_to_open()` to support the uniform cost and A* search strategies. These methods are already called in method `add_to_open()` of `SearchTree`. Add code in `search2()` to compute all the required information.

   Examples:

```
>>> t2 = MyTree(p,'uniform')
>>> t2.search2()
['Beja', 'Evora', 'Santarem', 'Leiria', 'Coimbra', 'Viseu']
>>> t2.tree_size,t2.solution_cost
(286, 469)

>>> t3 = MyTree(problems[0],'astar')
>>> t3.search2()
['Beja', 'Evora', 'Santarem', 'Leiria', 'Coimbra', 'Viseu']
>>> t3.tree_size,t3.solution_cost
(49, 469)
```

3. To apply the `tree_search` module to solve problems in the "Blocks World", we would need to develop a new search domain to compute the state transitions. In particular, given a state and a STRIPS operator, we would need to determine all possible instanciations of the operator in that state. These instanciations are based on assignments of constants in the world to variables in the operator. Develop a function `assigments(lvars,lconsts)` in the `tpi1` module that, given a list with all variables in a certain operator and a list with all constants (e.g. blocks in the blocks world) in the current state, computes a list with all possible assignments of constants to variables. Each assignment takes the form of a dictionary.

   Example:

```
>>> assignments(['X','Y'],['a','b','c'])
[ {'X': 'a', 'Y': 'a'}, {'X': 'b', 'Y': 'a'}, {'X': 'c', 'Y': 'a'},
  {'X': 'a', 'Y': 'b'}, {'X': 'b', 'Y': 'b'}, {'X': 'c', 'Y': 'b'},
  {'X': 'a', 'Y': 'c'}, {'X': 'b', 'Y': 'c'}, {'X': 'c', 'Y': 'c'} ]
```

4. Develop a search domain, `TSP`, to solve instances of the following Travelling Salesman Problem formulation:

   Given an inicial city and a list of other cities to visit, find a path that starts and ends in the initial city and passes only once in each of the other specified cities. The cities to visit are not necessarily adjacent, therefore it is acceptable to visit additional cities not in the given list. Domain information includes road connections between cities, respective distances and coordinates of cities.

   In the `TSP` domain, each state will be a pair `(lvisited,current)`, where `current` is the city where the travelling salesman is in a given moment and `lvisited` is a list with the sequence of cities he visited before.

   Note that, as specified in the `SearchDomain` class (see the `tree_search` module), domains now have, in addition to the methods you already know, i.e. `actions(s)`, `result(s,a)`, `cost(s,a)` and `heuristic(s,g)`, two additional methods:

   - `equivalent(s1,s2)`, which returns true if the two given states are equivalent.
   - `satisfies(state,goal)`, which returns true if the given goal is satisfied in the given state.

   You have to implement these six methods for `TSP`. The object `tsp_portugal` is already created in `tpi1_tests` as an instance of `TSP`.

   The following simple heristic (there are better ones ...) should be implemented: $CR + (n-1) * R + RL$, where CR is the minimum (straight line) distance from the current city to one of the remaining cities (not counting the initial/final city), n is the number of remaining cities, R is the minimum distance between pairs of remaining cities, and RL is the minimum distance from the initial/last city to one of the remaining cities.

   Example:

```
>>> tsp_portugal.actions((['Aveiro'],'Porto'))
[ ('Porto', 'Agueda'), ('Porto', 'Aveiro'), ('Porto', 'Braga'),
  ('Porto', 'Guimaraes')]

>>> tsp_portugal.result((['Aveiro'],'Porto'),('Porto','Braga'))
(['Aveiro', 'Porto'], 'Braga')

>>> tsp_portugal.cost((['Aveiro'],'Porto'),('Porto','Braga'))
57

>>> tsp_portugal.actions((['Aveiro','Porto'],'Aveiro'))
[]

>>> tsp_portugal.actions((['Aveiro','Porto'],'Agueda'))
[('Agueda', 'Aveiro'), ('Agueda', 'Coimbra'), ('Agueda', 'Viseu')]

>>> tsp_portugal.heuristic(([],'Porto'),(['Coimbra','Viseu'],'Porto'))
227.9228618619931
```

```
>>> tsp_portugal.equivalent( (['Porto','Aveiro','Agueda'],'Coimbra'),
                             (['Porto','Agueda','Aveiro'],'Coimbra') ))
True

>>> tsp_portugal.satisfies( (['Porto','Aveiro','Coimbra','Agueda'],'Porto'),
                            (['Porto','Coimbra'],'Porto') )
True


>>> p = formulateTSP(tsp_portugal,'Porto', \
              ['Agueda','Aveiro','Guimaraes','Viseu'])
>>> solveTSP(p,'breadth'))
( [ 'Porto', 'Aveiro', 'Agueda', 'Viseu', 'Lamego', 'Guimaraes', 'Porto' ],
  1255, 370 )


>>> q = formulateTSP(tsp_portugal,'Porto',['Aveiro','Guimaraes','Viseu'])
>>> solveTSP(q,'uniform')
( [ 'Porto', 'Guimaraes', 'Lamego', 'Viseu', 'Agueda', 'Aveiro', 'Porto' ],
  512, 370 )

>>> r = formulateTSP(tsp_portugal,'Porto',['Evora','Aveiro','Santarem','Viseu'])
>>> solveTSP(r,'astar')
( [ 'Porto', 'Aveiro', 'Coimbra', 'Leiria', 'Santarem', 'Evora',
    'Portalegre', 'Castelo Branco', 'Covilha', 'Viseu', 'Agueda', 'Porto'],
  4178, 950 )
```

III Clarification of doubts

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

1. (16h38) The definition of the heuristic for the last exercice was improved and the example was updated with the new results for the heuristic and A*.

2. Os resultados das soluções do exercício 1 e 2 apresentados no pdf só funcionam se *não* evitarmos ciclos. É suposto deixarmos?

   Resposta: Sim, não afecta o trabalho. Não vamos usar pesquisa em profundidade.

3. No exercicio 3, função assignments, importa a ordem dos dicionários na lista retornada?

   Resposta: A ordem não é relevante, quer na lista, quer dentro de cada dicionário.

... por causa da verificação de loops nem todos os nos gerados entram na lnewnodes.

Resposta: Não é pedida a prevenção de ciclos. Todos os resultados do enunciado são sem prevenção de ciclos. É melhor tirar a prevenção de ciclos, caso contrário não vai conseguir comparar completamente os resultados; e para avaliarmos a rapidez, convém também que sigam todos a mesma especificação

4. No primeiro exemplo da função actions(), no estado (['Aveiro'], 'Porto'), a acção ('Porto', 'Aveiro') é retornada, enquanto no terceiro exemplo é retornada a acção ('Agueda', 'Aveiro') mas não ('Agueda', 'Porto').

Resposta: No terceiro exemplo, voltar ao Porto não é possível porque o caixeiro viajante não repete cidades. Pode voltar a Aveiro, porque é a cidade inicial, para onde tem de voltar.

5. O guião diz "find a path that starts and ends in the initial city and passes only once in each of the other SPECIFIED cities". Pode-se repetir as cidades que não são de visita obrigatória?

   Resposta: Não.

6. Enunciado diz-nos que "... where CR is the minimum (straight line) distance from the current city to one of the remaining cities (not counting the initial/final city) ...". Esta distância é suposto ser também minimizada quando há mais que uma remaining city?

   Resposta: Exactamente: Calcular as distâncias da cidade actual a todas as cidades remanescentes e usar a menor. O mesmo se aplica a RL.

7. No ficheiro de teste, para a heurística, o código é:

   ```
   print('heuristic:',tsp_portugal.heuristic(
     ([],'Porto'), (['Coimbra','Viseu'],'Porto')))
   ```

   Mas no enunciado do pdf aparece:

   ```
   print('heuristic:',tsp_portugal.heuristic(
     ([],'Porto'), (['Porto','Coimbra','Viseu'],'Porto')))
   ```

   Devem ambos produzir o mesmo resultado de 227?

   Resposta: Era suposto estar no enunciado como está no módulo de teste. Agora já corrigi. Mas de facto ambas devem dar o mesmo. Quando se especifica o problema do TSP, especifica-se a cidade inicial e as outras a visitar; vejam a função `formulateTSP()`. O facto de Porto ser incluído não acrescenta nada porque Porto já é a cidade de partida.

8. Se a lista de cidades remanescentes está vazia, a distancia minima entre 'remaining' e 'Current' é 0?

   Resposta: A regra que consta no enunciado, só é aplicável se houver pelo menos duas cidades remanescentes. Não dá para calcular distâncias entre pares de cidades, quando só temos 1 ou nenhuma cidade. A lógica da heurística é a seguinte: para n cidades remanescentes, mais cidade actual mais cidade inicial/final, temos $n+2$ cidades, e portanto $n+1$ saltos. Para cada um desses saltos, temos que ter um valor mínimo como descrito no enunciado. Se já só há 1 cidade remanescente (3 cidades no total), temos 2 saltos. Se há 0 cidades remanescentes (2 cidades no total), temos 1 salto. Para estes casos especiais (1 ou 2 saltos), a heurística é óbvia ...

9. Na função `actions()`, a ordem tem que ser alfabética ou não é relevante? Se a lista de acções não estiver ordenada, é possível que a pesquisa em largura dê resultado diferente do que está no enunciado.

   Resposta: Com efeito, para evitar resultados diferentes, devem retornar a lista de acções ordenada; por exemplo: `return sorted(actlist)`. Fazendo a mesma alteração no meu código, o número de nós do exemplo de pesquisa em largura no TSP passa de 1250 para 1255 (exemplo actualizado acima).