Automated Problem Solving

2019/2020

©Luís Seabra Lopes

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

Última actualização: 2019-10-20

I Objectives

This document focuses on the problem of automatic problem solving using different search techniques. In particular, through a list of exercices, this document guides the students in exploring the use of tree search techniques. Constraint satisfaction search for assignment problems is also convered. This document is used in the courses of *Introduction on Artificial Intelligence*, of the Integraed Master on Computers and Telematics Engineering, and *Artificial Intelligence*, of the Bachelor (licenciatura) on Informatics Engineering. ***The exercises will be done in 4 practical classes. For a good progress, the exercises that are in the thematic scope of a given class should be completed before the next class.***

II Tree Search

1 The base code

A complete implementation of the basic tree search algorithm can be found in module `tree\_search`, provided together with this document. This module contains the following classes.

- Class `SearchDomain()` - abstract class that defines the structure of application domains;
- Class `SearchProblem(domain,initial,goal)` - class for representing specific problems to be solved;
- Class `SearchNode(state,parent)` - class for representing search tree nodes
- Class `SearchTree(problem)` - class containing methods for the generation of a search tree for a given problem.

As can be inferred from the adopted data structures, each instance of the class `SearchTree` has access to the following attributes and methods:

- `self.problem` - The problem to be solved (an instance of `SearchProblem`);

- `self.problem.domain` - The application domain (an instance of `SearchDomain`) to which the problem belongs;
- `self.problem.domain.actions(state)` - returns a list of the actions applicable in `state`;
- `self.problem.domain.result(state,action)` - returns the result of `action` in `state`;
- `self.problem.domain.cost(state,action)` - returns the cost of `action` in `state`;
- `self.problem.domain.heuristic(state1,state2)` - returns an estimate of the cost of going from `state1` to `state2`;
- `self.problem.initial` - the initial state;
- `self.problem.goal` - the goal state;
- `self.problem.goal_test(state)` - checks if `state` is the goal;
- `self.strategy` - the used search strategy;
- `self.open_nodes` - the queue of open nodes (tree leaves, to be expanded), where each node is an instance of `SearchNode`;
- `self.search()` - the main search method.

O método principal da classe `SearchTree` implementa um procedimento genérico de pesquisa, baseado em fila de nós abertos:
The main method of the `SearchTree` class implements a generic tree search procedure, based on a queue of open nodes:

```
def search(self):
    while self.open_nodes != []:
        node = self.open_nodes[0]
        if self.problem.goal_test(node.state):
            return self.get_path(node)
        self.open_nodes[0:1] = []
        lnewnodes = []
        for a in self.problem.domain.actions(node.state):
            newstate = self.problem.domain.result(node.state,a)
            lnewnodes += [SearchNode(newstate,node)]
        self.add_to_open(lnewnodes)
    return None
```

The module `cidades`, with a specific application domain, which can be used for testing, is also available with this document.

2 Exercices

The following exercices are extensions to the provided `tree_search` module.

1. The implementation provided does not prevent loops. This leads to using more memory space than needed in *breadth-first search* and to infinite loops in *depth-first search*. So, change and/or add the necessary code in order to prevent the creation of branches with loops. Test the program with the depth-first search strategy.

2. In the data structure used to represent the nodes in the search module, add a field to record the depth of the node. It is considered that the root of the search tree is at depth 0.

3. Modify the search algorithm to register, in the search tree (an instance of `SearchTree`), the length of the solution found, given by the number of state transitions from the initial state to the goal state.

4. Make the necessary changes to the `tree_search` module in order to support depth-limited search.

5. Add code to the `search()` method from class `SearchTree` to calculate the total number of terminal and non-terminal nodes in the tree after completion of the search. This information should be stored in attributes of the `self`. Consider that an expanded node with no children counts as a non-terminal node.

6. As you know, the average branching factor is given by the ratio between the number of child nodes (that is, all nodes except the root of the tree) and the number of parent nodes (nonterminal nodes). Add code to the `search()` method of class `SearchTree` to calculate the respective average branching factor, storing it in a field of the `self`.

7. In class `Cidades` (i.e. cities) of the module `cidades`, implement the `cost()` method, which, given a state and an action, returns the cost of performing that action in that state. In this case, for an action $(C1, C2)$, corresponding to a transition from city $C1$ to city $C2$, the cost should be the (road) distance between these cities.

8. In the data structure used in module `tree_search` to represent the nodes, add a field to store the accumulated cost of all actions in the path from the root to the node. Modify the search algorithm to record the accumulated cost in each node inserted in the tree.

9. Modify the search algorithm to register, in the search tree (an instance of `TreeSearch`), the total cost of the solution found, given by the sum of the costs of successive transitions.

10. Make the necessary changes to the code of the `tree_search` module to support *uniform cost search*.

11. In the data structure used to represent the nodes in the `tree_search` module, add a field to record an estimate (heuristic) of the cost of reaching a solution from the state in that node.

12. Identify a suitable heuristic for the problem domain class defined in module `cidades` (class `Cidades` and implement the method `heuristic()` of this class.

13. Make the necessary changes to the `tree_search` module to support greedy research.

14. Make the necessary changes to the `tree_search` module to support A* search. Compare the results of different search techniques.

15. Add code to the `search()` from class `SearchTree` to determine the node or nodes with higher accumulated cost. This information should be stored in the form of a list in a field of `self`.

16. Add code to the `search()` method of class `SearchTree` to determine the average depth of the nodes in the generated tree. This information should be stored in a field of the `self`.

III Search with STRIPS operators

Attached to this description, you can find the module `strips`, with a new search domain based on STRIPS operators. The module `blocksworld` implements predicates and STRIPS operators for the well-known "blocks world ".

An operator, represented by a class derived from `Operator`, specifies the preconditions, negative effects and positive effects of a class of actions:

```
class Stack(Operator):
    args = ['X','Y']                          # arguments
    pc   = [Holds('X'),Free('Y')]              # pre-conditions
    neg  = [Holds('X'),Free('Y')]               # negative effects
    pos  = [On('X','Y'),HandFree(),Free('X')] # positive effects
```

We can instantiate an operator as in the following example:

```
>>> op = Stack.instanciate(['a','b'])
>>> op
Stack(a,b)
>>> print(op)
Stack([a,b], [Holds(a),Free(b)], [Holds(a),Free(b)],
[On(a,b),HandFree(),Free(a)])
```

This method is used to implement the `actions()` method in the search domain class `STRIPS`.

```
>>> initial_state
[ Floor(a),Floor(b),Floor(d),Holds(e),On(c,d),Free(a),Free(b),Free(c) ]
>>> bwdomain = STRIPS()
>>> bwdomain.actions(initial_state)
[ Stack(e,a), Stack(e,b), Stack(e,c), Putdown(e) ]
```

1 Exercícios

Although most of the work has already been done, some details remain to be completed:

1. Implement the methods `result()` e `satisfies()` in the `STRIPS` class.
2. In the data structure used to represent the nodes in module `tree_search`, add an attribute to record the action that led to that node. On the search tree, add an attribute `plan` to register the sequence which constitutes the solution found. Modify the search algorithm to assign the correct values to these attributes.
3. You can now try it, starting with the example created in module `blocksworld`:

   ```
   >>> goal_state
   [ Floor(c), On(d,c), On(e,d), On(a,e), Floor(b) ]
   >>> p = SearchProblem(bwdomain,initial_state,goal_state)
   >>> t = SearchTree(p)
   ```

```
>>> t.search()
>>> t.plan
[ Stack(e,a), Unstack(c,d), Putdown(c), Pickup(d), Stack(d,c),
  Unstack(e,a), Stack(e,d), Pickup(a), Stack(a,e) ]
```

This search takes some time. [1] In part this is due to the complexity of the application domain. But on the other hand, the check of repeated states that you implemented in exercise II.2.1 is not appropriate for the blocks world . Understand why and change the way the control of repeated states is implemented, so that it works for any application domain.

IV Constraint-based search for assignment problems

Together with this document, you can find the`constraintsearch` module, similar to the one developed in theoretical classes. The module provides a class `ConstraintSearch` that allows solving assignment problems with constraints. The module `rainhas` (queens) creates an instance of `ConstraintSearch` to solve the the 4 queens problem.

1 Exercises

1. Solve exercises IV.4 and IV.5 of the theoretical-practical guide using the `constraintsearch` module.

2. The `search()` of class `ConstraintSearch` does not propagate constraints. Add a method to propagate and use it in the `search()` method.

[1] Note that, given the use of dictionaries in the module `strips`, search behavior is non-deterministic and the time it takes for the same problem is variable.