LINGI2261: Artificial Intelligence
Assignment 2: Solving Problems with Informed Search

Jean-Baptiste Mairy, Cyrille Dejemeppe, Yves Deville
October 2013

Guidelines

- This assignment is due on **Wednesday 23 october, 6:00 pm**.
- *No delay* will be tolerated.
- Not making a *running implementation* in *Python 3* able to solve (some instances of) the problem is equivalent to fail. Writing some lines of code is easy but writing a correct program is much more difficult.
- *Document* your source code (at least the difficult or more technical parts of your programs). Python docstrings for important classes, methods and functions are also welcome.
- Indicate clearly in your report if you have *bugs* or problems in your program. Do not let the instructor discover it.
- Copying code from other groups (or from the internet) is strictly forbidden. Each source of inspiration must be clearly indicated. The consequences of *plagiarism* is *0/20 for all assignments*.
- Answers to all the questions must be delivered at the INGI *secretary* (paper version). Put your names and your group number on it. Remember, the more concise the answers, the better.
- Source code shall be submitted via the *SVN* tool. Only programs submitted via this procedure will be corrected. No report or program sent by email will be corrected.
- Respect carefully the *specifications* given for your program (arguments, input/output format, etc.) as we will use fully-automated tests. When we provide a way to test your program, it *must* pass this test.

Deliverables

- The answers to all the questions in paper format **at the secretary** (do not forget to put your group number on the front page).
- The following files inside the `milestone2` directory of your SVN repository:
  - Two files `mazeCollect.py` and `mazeCollect2.py` containing your two Python 3 versions of the the mazeCollect problem solver. Your program should take the path to the instance file as only argument. The search strategy that should be enabled by default in your programs is A^* with your best heuristic. Both versions should print the solution to the problem to the standard output in the format described further. We provide a online checker to test the validity of your outputs on `http://verifymazecollect.appspot.com/`. The outputs returned by your two versions of the solver must pass this test as your programs will be automatically tested.

1 Search Algorithms and their relations (3 pts)

1.1 $A^\star$ versus uniform-cost search

Consider the maze problem given on Figure 1. The goal is to find a path from 🚹 to € moving up, down, left or right. The dashed positions represent walls. This question must be answered by hand and doesn't require any programming.

Questions

1. Give a consistent heuristic for this problem. Prove that it is admissible.
2. Show on the left maze the states (board positions) that are visited during an execution of a uniform-cost graph search. We assume that when different states in the fringe have the smallest value, the algorithm chooses the state with the smallest coordinate (i, j) ($(0, 0)$ being the bottom left position, i being the horizontal index and j the vertical one) using a lexicographical order.
3. Show on the right maze the board positions visited by $A^\star$ graph search with a manhattan distance heuristic (ignoring walls). A state is visited when it is selected in the fringe and expanded. When several states have the smallest path cost, this uniform-cost search visits them in the same lexicographical order as the one used for uniform-cost graph search.

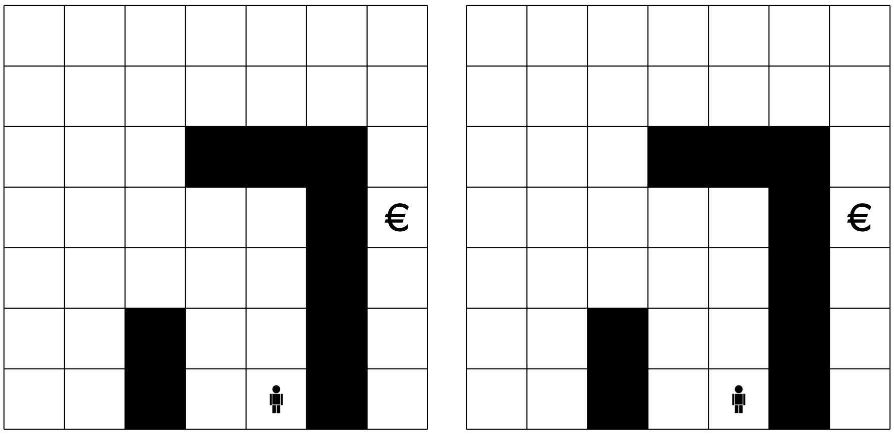

Figure 1

2 MazeCollect problem (17 pts)

The problem you will solve for this assignment is the *mazeCollect* problem. Again, the search procedures from aima-python3 will help you solve it! The goal of the problem is to guide a person that can move up, right, down and left to collect the money that is scattered in a maze and bring it to a safe that is also inside the maze. The problem has an additional requirement: the number of moves that the player does must be minimal.

2.1 Input and output format

Resources for this problem are available on icampus in *Document* > *Assignments* > *Assignment2*. Your are given two sets of instances, namely *Benchs_Small* and *Benchs_Large*. Each instance file contains the initial state of the problem, where you can find the initial position of the player, the position of the money and the safe as well as the walls of the maze. The format used to represent the states is the following:

- @ is the position of the player
- $ are the positions of the money items
- + is the position of the safe
- # are the wall positions
- an empty position is represented with a white space

For instance, the file *mazeCollect0* contains:

```
# ## #  # #
#  ###   #       #
 #  #  #    # ## +#
##   #    #$##     #
 #  ##    ###  #
##  #    ##    ##
 # # #    # #   # #
          # #   #
# #     ### #      #
  ##     ###  # ##
##
  # # #  #     ## ##
   #  # #    #  $ #
#  # # $      ##  $
 #      #  # #  ##
    $ #  #  #     #
   #$ #   #        #
  ##    ## # # #  #
  ####  #@ # $
##  #    #   # #  #
```

A solution to this problem is composed of the successive states of the game, induced by the displacement of the player, leading to the collection of all the money and its storage in the safe in a minimal number of steps. Three possible last states of a solution for the mazeCollect0 are represented in Figure 2. Each state is represented using the same format as the input and separated from the others with an empty line. The player can move only from one position to an adjacent one between two successive states in the solution file and once collected, a $ signs don't appear in the subsequent states.

```
# ## #  # #
#  ###   #      @#
 #  #  #    # ## +#
##   #    # ##     #
 #  ##    ###  #
##  #    ##    ##
 # # #    # #   # #
          # #   #
# #     ### #      #
  ##     ###  # ##
##
  # # #  #     ## ##
   #  # #    #    #
#  # #        ##
 #      #  # #  ##
      #  #  #     #
   #  #   #        #
  ##    ## # # #  #
  ####  #  #
##  #    #   # #  #
```

```
# ## #  # #
#  ###   #       #
 #  #  #    # ##@+#
##   #    # ##     #
 #  ##    ###  #
##  #    ##    ##
 # # #    # #   # #
          # #   #
# #     ### #      #
  ##     ###  # ##
##
  # # #  #     ## ##
   #  # #    #    #
#  # #        ##
 #      #  # #  ##
      #  #  #     #
   #  #   #        #
  ##    ## # # #  #
  ####  #  #
##  #    #   # #  #
```

```
# ## #  # #
#  ###   #       #
 #  #  #    # ## @#
##   #    # ##     #
 #  ##    ###  #
##  #    ##    ##
 # # #    # #   # #
          # #   #
# #     ### #      #
  ##     ###  # ##
##
  # # #  #     ## ##
   #  # #    #    #
#  # #        ##
 #      #  # #  ##
      #  #  #     #
   #  #   #        #
  ##    ## # # #  #
  ####  #  #
##  #    #   # #  #
```

Figure 2: Three possible last states of a solution to *mazeCollect0*

2.2 Your programs

For this problem, you will do two versions of the mazeCollect A* solver. The difference between the first version and the second will be the moves considered **in your model** (off course, the moves printed in your solution will remain the same). But before diving into the code, answer the questions below.

Questions

1. Give an upper bound on the number of different states a mazeCollect problem of size $n \times m$ with k money items to collect ? Justify your answer precisely.

The efficiency of A* is greatly influenced by the heuristic it uses. As you will be performing A* inside a graph search and searching for the optimal number of steps, **all your heuristics must be consistent**. For such a problem, good heuristics typically include information about the distance from the player to the money items, information about the distances between the different money items and some information about the distance to the goal. The more precise the heuristic, the better the results of A* are.

Questions

1. Describe your best consistent heuristic for the mazeCollect problem. When using distance, precise which distance you use. Justify precisely the admissibility and the consistency of your heuristic.

It's now time to code the first version of your mazeCollect A* solver. As for the previous assignment, you will use the implementation of the book's primitives inside *aimp-python3* (on iCampus: *Document > Assignments*).

Questions

1. Implement the first version of your solver. For this version, the moves considered in your *successor* function must correspond to the atomic player moves (up, right, down and left). Extend the *Problem* class and implement the necessary methods and other class(es) if necessary. Your program file must be named *mazeCollect.py*. Your program must print to the standard output a solution to the mazeCollect instance for which the path to the instance file is given in argument. This solution must satisfy the described format.
2. Experiment, compare and analyze informed (*astar_graph_search*) and uninformed (*breadth_first_graph_search*) graph search of aima-python3 on the 10 instances of mazeCollect inside *Benchs_Small*. Report in a table the time, the number of explored nodes and the number of steps to reach the solution. Be aware that the last two instances can only be solved using A* with a good heuristic.
   When no solution can be found by a strategy in a reasonable time (say 3 min), explain the reason (time-out and/or swap of the memory).
3. In your experiments, is the time taken by A* always smaller than the one taken by breadth first search and why ?
4. In your experiments, is the number of nodes explored by A* always smaller than the number of nodes explored by breadth first search and why ?

The moves considered in the model of the first version correspond to the actual moves of the player in the solution. This is an unnecessary requirement. For the second version of the mazeCollect solver, you will consider moving directly the player to a money item. Be careful with the path cost in this version as it must be the true cost of moving the player to the box (taking the walls into account).

Questions

1. Is your previous heuristic still adapted for this model ?
2. Implement the second version of your solver. For this version, the moves considered in your *successor* function correspond to moving the player directly on one money item or on the safe. Extend the *Problem* class and implement the necessary methods and other class(es) if necessary. Your file must be named *mazeCollect2.py*. Your program must print to the standard output a solution to the mazeCollect instance for which the path to the instance file is given in argument. The solution must satisfy the described format. The moves in your solution must correspond to atomic moves of the player (up, right, down and left).
3. Experiment, compare and analyze the differences in performances of your first version of the solver and the second one on the 5 instances of mazeCollect inside *Benchs_Large*. Report in a table the time, the number of explored nodes and the number of steps to reach the solution.
   When no solution can be found by a strategy in a reasonable time (say 3 min), explain the reason (time-out and/or swap of the memory).
4. What is the problem when using breadth first graph search with this second version of the solver ?