



EPL - ECOLE POLYTECHNIQUE DE LOUVAIN

LINGI2261 - ARTIFICIAL INTELLIGENCE

Report of second assignement

Professor :

Yves DEVILLE

Students : (Group 1)

Benoît BAUFAYS 22200900

Program :

Julien COLMONTs 41630800

SINF21MS/G

Academic year 2013-2014

Contents

1	Answering questionnaire	1
2	MazeCollect Program	2
3	MazeCollect2 Program	10

1 Answering questionnaire

Question 1 : *Give a consistent heuristic for this problem. Prove that it is admissible.*

A consistent heuristic could be the manhattan distance between character position and euro position.

Proof of admissibility. $h(n)$ cannot overestimate cost to reach euro position because manhattan distance is the minimum cost without taking the dashed positions in account. So $h(n)$ is optimistic. \square

Question 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.*

Since cost while moving character is 1, an uniform-cost graph search will visit almost all board positions.

Question 3 : *Show on the right maze the board positions visited by A^* 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. A^* graph search is much more efficient because it will only visit nodes that minimise manhattan distance.*

You can observe result on figure 1.

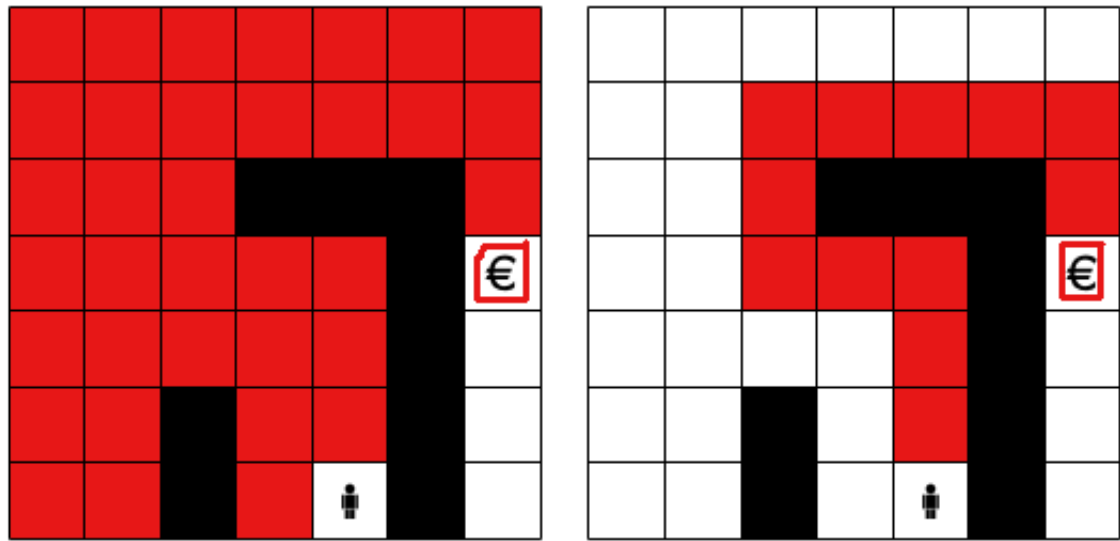


Figure 1

2 MazeCollect Program

- 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.

Upper bound : $(n \times m) * 2^k$.

2^k . A money displayed can have two states. The characted already reached it or not. To represent all the possible combinations for the money on the board, we have 2^k ☐

$n \times m$. It represents all possible positions of the character. ☐

If we put this justifications together, we have the upper bound shown above. We must put together all the possible states for money and character position.

- 2 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.

Our heuristic computes the sum of manhattan distance (shortest way) between all non-taken dollars (and the goal if all dollars are taken) and manhattan distance between our current position and closest dollar.

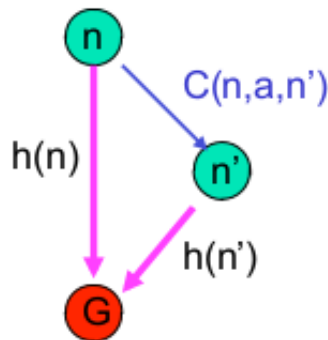
Admissibility. $h(n) \leq \text{realCost}(n)$: it only uses manhattan distance between dollars, current position and goal. The cost computed is the minimal expected. The value computed is the real cost if there are no walls, it's optimistic if there are walls.

$h(n) > 0$ if n is not the goal : the distance between goal and current position is at least 1 so $h(n) > 0$

□

Consistency. We have to prove that $h(n) \leq c(n, a, n') + h(n')$.

Since the cost of moving from current position to closest dollar decreased from 1 after an action a , the cost computed for successor is the cost of current state minus one. Actually, we have $h(n) = c(n, a, n') + h(n')$, with $c(n, a, n') = 1$ and $h(n') = h(n) - 1$.



□

- 3** 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.

```
1  '''NAMES OF THE AUTHOR(S): Baufays Benoit - Colmonts Julien'''
2
3  from search import *
4  import math
5
6
7  ##### Implement the search
8  #####
9
10 class MazeCollect(Problem):
11     #index for the money tuple in the state
12     money_Index=1
13
14     def __init__(self,init):
15         #dictionnary of vide position
16         self.vide=None
17         #position of the goal
18         self.goal=None
19         self.createMap(init)
20
21     def goal_test(self, state):
22         if(state[0]=="init"):
23             mapInfo=state[1]
24         else:
25             mapInfo=state
26
27         return (len(mapInfo[self.money_Index])==0) and mapInfo
28             [0]==self.goal
29
30     def successor(self, state):
31         if(state[0]=="init"):
32             mapInfo=state[1]
33         else:
34             mapInfo=state
35         retour=(tuple(self.IA(mapInfo)))
36         return retour
37
38     def IA(self, mapInfo):
39         pos=[]
```

```

40         position=mapInfo[0]
41         #the four possibilities
42         for a in [(-1,0),(0,-1),(1,0),(0,1)]:
43             #take the money list
44             moneyStack=list(mapInfo[self.money_Index])
45             newPosition=(position[0]+a[0],position[1]+a[1])
46             #if the new position is a entry of the vide
            dictionnary
47             if self.getString(newPosition) in self.vide:
48                 if(newPosition in moneyStack):
49                     #if it is a dollar case, we must recompute the
                        minimal distance between all the dollars
                        \{newPosition}
50                     moneyStack.remove(newPosition)
51                     saveMoney=list(mapInfo[self.money_Index])
52                     saveMoney.remove(newPosition)
53                     distance=self.dist(saveMoney)
54                 else:
55                     distance=mapInfo[2]
56                 #we use the yield to return a iteraor. With this,
                    we use less memory
57                 yield (('move',((newPosition,tuple(moneyStack),
58                                     distance))))
59
60 #find the closest dollar of a position
61 def closestDollar(self, dollars,position):
62     closestDollar = None
63     dst = 0
64
65     if len(dollars) is not 0 :
66         dst = self.manhattan(position, dollars[0])
67         for elem in dollars :
68             temp=self.manhattan(position, elem)
69             if temp <= dst:
70                 dst = temp
71                 closestDollar = elem
72     return closestDollar
73
74 #see the rapport for a explication of this heuristique
75 def Heuristique(self,node):
76     state=node.state
77     if(state[0]=="init"):
78         mapInfo=state[1]
79     else:
80         mapInfo=state
81     infoDollar=mapInfo[2]

```

```
81     allNodes=list(mapInfo[self.money_Index])
82     #if we have reach all dollar, we must now find the best
      past to the goal
83     if len(allNodes) is 0:
84         return self.manhattan(self.goal,mapInfo[0])
85     else:
86         closest=self.closestDollar(allNodes,mapInfo[0])
87         return self.manhattan(mapInfo[0], closest) +mapInfo[2]
88
89
90     #the manhattan distance between two points
91     def manhattan(self, toPoint,fromPoint):
92         return (math.fabs(toPoint[0] - fromPoint[0]) + math.fabs(
            toPoint[1] - fromPoint[1]))
93
94     #get a String representation of a point (utility for the key
      of the vide dictionnary)
95     def getString(self,point):
96         final="{0}#{1}".format(point[0],point[1])
97         return final
98
99     #method to find the small path between all non reached dollars
100    def dist(self,moneyT):
101        dist=10000000000000
102        money=list(moneyT)
103        for node in money:
104            nodeMin=None
105            distance=0
106            moneyStack=money
107            moneyStack.remove(node)
108            while len(moneyStack)>0:
109                distMin=dist=self.manhattan(moneyStack[0],node)
110                nodeMin=(moneyStack[0])
111                for dollar in moneyStack:
112                    dist=self.manhattan(dollar,node)
113                    if(dist<distMin):
114                        distMin=dist
115                        nodeMin=dollar
116                distance+=distMin
117                moneyStack.remove(nodeMin)
118            #we add the path between the last node and the goal
119            if nodeMin is not None:
120                distance+=self.manhattan(self.goal,nodeMin)
121            if dist>distance:
122                dist=distance
123
```



```
124         return dist
125
126
127     #read the file, create the map and code it for the state
    schema
128     def createMap(self,path):
129         sizeY=0
130         x=0
131         y=-1
132         moneyStack=[]
133         debut=[]
134         vide={}
135
136         f = open(path,'r')
137         for line in f:
138             y=y+1
139             sizeY=sizeY+1
140             x=-1
141             for elem in line:
142                 x=x+1
143                 if(elem!='#' and elem!="\n"):
144                     if(elem=='$'):
145                         moneyStack.append((x,y))
146                     elif(elem=='@'):
147                         debut=(x,y)
148                     elif(elem=='+'):
149                         self.goal=(x,y)
150                         vide[self.getString((x,y))]=(x,y)
151
152
153         saveMoneyStack=list(moneyStack)
154         distance=self.dist(saveMoneyStack)
155         mapInfo=('init',(debut,tuple(moneyStack),distance,(x,sizeY)))
156         self.vide=vide
157         self.initial=mapInfo
158
159
160     ##### Launch the search #####
161
162     if(len(sys.argv)>1):
163         problem=MazeCollect(sys.argv[1])
164     else:
165         problem=MazeCollect("Benchs_Small/mazeCollect0")
166     node=astar_graph_search(problem, problem.Heuristique)
167     path=node.path()
```

```

168 path.reverse()
169 sizeElements=len(path[0].state[1])
170 size=path[0].state[1][sizeElements-1]
171 for n in path:
172     a=['#'] * size[1]
173     for i in range(0,size[1]):
174         a[i]=['#'] * size[0]
175     state=n.state
176     if(state[0]=="init"):
177         state=state[1]
178
179     #place libre
180     freeSpaces=problem.vide
181     for spaceS in freeSpaces:
182         space=spaceS.split('#')
183         a[int(space[1])][int(space[0])]=' '
184
185     #coffre
186     elem=problem.goal
187     a[elem[1]][elem[0]]= '+'
188
189     #current position
190     elem=state[0]
191     a[elem[1]][elem[0]]= '@'
192
193     #money
194     moneyStack=state[problem.money_Index]
195     for money in moneyStack:
196         a[money[1]][money[0]]= '$'
197
198
199     for ligne in a:
200         ligneP=""
201         for elem in ligne:
202             ligneP=ligneP+elem
203         print(ligneP)
204     print(' ')

```

- 4 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).

mazeCollect.py	BFS			A^* search		
Benchs_Small	Time (s.)	Explored nodes	Steps	Time (s.)	Explored nodes	Steps
mazeCollect0	0,74	27.761	76	0,63	10.200	76
mazeCollect1	0,34	11.917	70	0,64	10.636	70
mazeCollect2	6,53	180.602	84	3,33	43.979	84
mazeCollect3	0,93	35.789	103	0,83	14.160	103
mazeCollect4	5,48	169.287	99	6,38	86.559	99
mazeCollect5	9,68	283.145	132	15,37	188.149	132
mazeCollect6	12,78	335.952	123	6,83	91.388	123
mazeCollect7	8,89	253.242	158	11,99	158.156	158
mazeCollect8	43,02	1.020.263	103	88,09	498.016	103
mazeCollect9	75,02	1.530.281	245	26.17	255.260	245

5 *In your experiments, is the time taken by A^* always smaller than the one taken by breadth first search and why ?*

No, it depends on the cases. Because of errors from heuristic (too optimistic), A^* can try to search in a way it will be stucked by walls. The time used to compute manhattan distance between all the elements explained earlier can take several time. This time is lost if the subtree chosen by the heuristic seems finally not being an optimal one. BFS make a way less computations even if it goes through many more nodes.

6 *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 ?*

Yes, it is very logical. BFS try to search in all nodes, even the ones describing a very bad move. During this time, A^* chooses only nodes which seems to do good moves.

3 MazeCollect2 Program

Question 1 : *Is your previous heuristic still adapted for this model ?*

Yes, it is. It's a particular case of mazeCollect. In the first case, heuristic was whatever the current position was. In this case, heuristic is used only when current position stands on a dollar. Since we proved our heuristic for first case, it's exactly the same here.

Question 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).*

```
1  '''NAMES OF THE AUTHOR(S): Baufays Benoit - Colmonts Julien'''
2
3  from search import *
4  import math
5
6  #we use a minimal mazeCollect to get the cost for a minimal path
   between two point
7  ##### class to find the small cost for one path
   #####
8  class MazeCollectMinimal(Problem):
9      def __init__(self, mapInfo):
10         #a dictionary of all vide points of the map
11         self.vide=mapInfo[3]
12         #the goal
13         self.goal=mapInfo[0]
14         self.initial=('init', (mapInfo[1], mapInfo[2]))
15
16         #get a String representation of a point (utility for the key
17         of the vide dictionary)
18         def getString(self, point):
19             final="{0}#{1}".format(point[0], point[1])
```

```

19         return final
20
21     def goal_test(self, state):
22         if (state[0]=="init"):
23             mapInfo=state[1]
24         else:
25             mapInfo=state
26
27         return self.goal==mapInfo[0]
28
29     def successor(self, state):
30         if (state[0]=="init"):
31             mapInfo=state[1]
32         else:
33             mapInfo=state
34         retour=(tuple(self.IA(mapInfo)))
35         return retour
36
37     def IA(self, mapInfo):
38         pos=[]
39         position=mapInfo[0]
40         for a in [(-1,0),(0,-1),(1,0),(0,1)]:
41             newPosition=(position[0]+a[0],position[1]+a[1])
42             if self.getString(newPosition) in self.vide:
43                 yield(('move',(newPosition,mapInfo[1])))
44
45     #a basic heuristique that get the manhattan distance between
46     the current position and the goal
47     def Heuristique(self,node):
48         state=node.state
49         if (state[0]=="init"):
50             mapInfo=state[1]
51         else:
52             mapInfo=state
53         return self.manhattan(mapInfo[0],self.goal)
54
55     #the manhattan distance between two points
56     def manhattan(self, toPoint,fromPoint):
57         #print(toPoint)
58         return (math.fabs(toPoint[0] - fromPoint[0]) + math.fabs(
59             toPoint[1] - fromPoint[1]))
60
61     ##### Implement the search
62     #####
63     class MazeCollect2(Problem):

```

```
62     money_Index=1
63     def __init__(self,init):
64         #a dictionnary of all vide points of the map
65         self.vide=None
66         #the goal position
67         self.goal=None
68         self.createMap(init)
69
70
71     def goal_test(self, state):
72         if(state[0]=="init"):
73             mapInfo=state[1]
74         else:
75             mapInfo=state
76
77         return (len(mapInfo[1])==0) and mapInfo[0]==self.goal
78
79
80     def successor(self, state):
81         if(state[0]=="init"):
82             mapInfo=state[1]
83         else:
84             mapInfo=state
85         retour=(tuple(self.IA(mapInfo)))
86         return retour
87
88
89     def IA(self,mapInfo):
90         pos=[]
91         moneyStack=list(mapInfo[1])
92         if len(moneyStack)>0:
93             #each money is a successor
94             for money in moneyStack:
95                 newMoneyStack=list(mapInfo[1])
96                 newMoneyStack.remove(money)
97                 yield (('move',((money,tuple(newMoneyStack)))))
98         else:
99             #we have not enough money to find, go to the goal
100             yield (('move',((self.goal,tuple(moneyStack)))))
101
102
103     #find the closest dollar of a position
104     def closestDollar(self, dollars,position):
105         closestDollar = None
106         dst = 0
107
```

```
108         if len(dollars) is not 0 :
109             dst = self.manhattan(position, dollars[0])
110             for elem in dollars :
111                 temp=self.manhattan(position, elem)
112                 if temp <= dst:
113                     dst = temp
114                     closestDollar = elem
115             return closestDollar
116
117         #get a String representation of a point (utility for the key
118         of the vide dictionnary)
119         def getString(self,point):
120             final="{0}#{1}".format(point[0],point[1])
121             return final
122
123         #method to find the small path between all non reached dollars
124         def dist(self,moneyT):
125             dist=1000000000000
126             money=list(moneyT)
127             for node in money:
128                 nodeMin=None
129                 distance=0
130                 moneyStack=money
131                 moneyStack.remove(node)
132                 while len(moneyStack)>0:
133                     distMin=dist=self.manhattan(moneyStack[0],node)
134                     nodeMin=(moneyStack[0])
135                     for dollar in moneyStack:
136                         dist=self.manhattan(dollar,node)
137                         if(dist<distMin):
138                             distMin=dist
139                             nodeMin=dollar
140                     distance+=distMin
141                     moneyStack.remove(nodeMin)
142                 #we add the path between the last node and the goal
143                 if nodeMin is not None:
144                     distance+=self.manhattan(self.goal,nodeMin)
145                 if dist>distance:
146                     dist=distance
147             return dist
148
149         def Heuristique(self,node):
150             state=node.state
151             if(state[0]=="init"):
152                 mapInfo=state[1]
153             else:
```

```
153         mapInfo=state
154     allNodes=list(mapInfo[self.money_Index])
155     if len(allNodes) is 0:
156         return self.manhattan(self.goal,mapInfo[0])
157     else:
158         closest=self.closestDollar(allNodes,mapInfo[0])
159         dist=self.dist(allNodes)
160         return self.manhattan(mapInfo[0], closest) +dist
161
162     #the manhattan distance between two points
163     def manhattan(self, toPoint,fromPoint):
164         return (math.fabs(toPoint[0] - fromPoint[0]) + math.fabs(
165             toPoint[1] - fromPoint[1]))
166
167     def path_cost(self, c, state1, action, state2):
168         if(state1[0]=="init"):
169             mapInfo=state1[1]
170         else:
171             mapInfo=state1
172
173         problem=MazeCollectMinimal((mapInfo[0],state2[0],mapInfo
174             [1],self.vide))
175         node=astar_graph_search(problem, problem.Heuristique)
176         path=node.path()
177         return c + len(path)
178
179     def createMap(self,path):
180         sizeY=0
181         x=0
182         y=-1
183         moneyStack=[]
184         debut=[]
185         vide={}
186
187         f = open(path,'r')
188         for line in f:
189             y=y+1
190             sizeY=sizeY+1
191             x=-1
192             for elem in line:
193                 x=x+1
194                 if(elem!='#' and elem!="\n"):
195                     if(elem=='$'):
196                         moneyStack.append((x,y))
197                     elif(elem=='@'):
```



```

197         debut=(x,y)
198         elif(elem=='+'):
199             self.goal=(x,y)
200             vide[self.getString((x,y))]=(x,y)
201
202     mapInfo=('init',(debut,tuple(moneyStack),(x,sizeY)))
203     self.vide=vide
204     self.initial=mapInfo
205
206     def printOneState(self,n):
207         a=['#'] * size[1]
208         for i in range(0,size[1]):
209             a[i]=['#'] * size[0]
210         state=n.state
211         if(state[0]=="init"):
212             state=state[1]
213
214         freeSpaces=problem.vide
215         for spaceS in freeSpaces:
216             space=spaceS.split('#')
217             a[int(space[1])][int(space[0])]=' '
218
219
220         #money
221         moneyStack=state[1]
222         for money in moneyStack:
223             a[money[1]][money[0]]='$'
224
225         #coffre
226         elem=self.goal
227         a[elem[1]][elem[0]]='+'
228
229         #current position
230         elem=state[0]
231         a[elem[1]][elem[0]]='@'
232
233
234         for ligne in a:
235             ligneP=""
236             for elem in ligne:
237                 ligneP=ligneP+elem
238             print(ligneP)
239         print(' ')
240
241
242     ##### Launch the search #####

```

```

243
244
245 if (len(sys.argv)>1):
246     problem=MazeCollect2(sys.argv[1])
247 else:
248     problem=MazeCollect2("Benchs_Small/mazeCollect0")
249 node=astar_graph_search(problem, problem.Heuristique)
250 path=node.path()
251 path.reverse()
252 sizeElements=len(path[0].state[1])
253 size=path[0].state[1][sizeElements-1]
254 tmp=None
255 number=0
256 for n in path:
257     if tmp is not None:
258         subProblem=MazeCollectMinimal((tmp[0],n.state[0],tmp[1],
259                                         problem.vide))
259         subNode=astar_graph_search(subProblem, problem.Heuristique)
260         subNode=breadth_first_graph_search(subProblem)
261         subPath=subNode.path()
262         l=0
263         subPath.reverse()
264         for sub in subPath:
265             if l>0:
266                 problem.printOneState(sub)
267                 l+=1
268             number+=l-1
269         if n.state[0] is "init":
270             tmp=n.state[1]
271             problem.printOneState(n)
272         else:
273             tmp=n.state

```

Question 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).*

The limit of 3 minuts was too short for our algorithm, all results for large maps lead to time

	Benchs_Large	mazeCollect.py			mazeCollect2.py		
		Time (s.)	Explored nodes	Steps	Time (s.)	Explored nodes	Steps
out.	mazeCollect10	TIMEOUT	•	•	TIMEOUT	•	•
	mazeCollect11	TIMEOUT	•	•	TIMEOUT	•	•
	mazeCollect12	TIMEOUT	•	•	TIMEOUT	•	•
	mazeCollect13	TIMEOUT	•	•	TIMEOUT	•	•
	mazeCollect14	TIMEOUT	•	•	TIMEOUT	•	•

Question 4 : *What is the problem when using breadth first graph search with this second version of the solver ?*

In the second problem, the maximum depth of the tree is $k+1$, with k , number of dollars. The main difference with the first case is that the cost on a fixed level on the tree isn't constant. The BFS search will stop on the first node which reach goal, after taking all dollars, but it's not necessary an optimal one.