EPL - ECOLE POLYTECHNIQUE DE LOUVAIN

LINGI2261 - ARTIFICIAL INTELLIGENCE

Report of second assignement

Professor :

Yves DEVILLE

Students : (Group 1)

Benoît BAUFAYS 22200900

Julien COLMONTS 41630800

Program :

SINF21MS/G

Academic year 2013-2014

Contents

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. □

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.

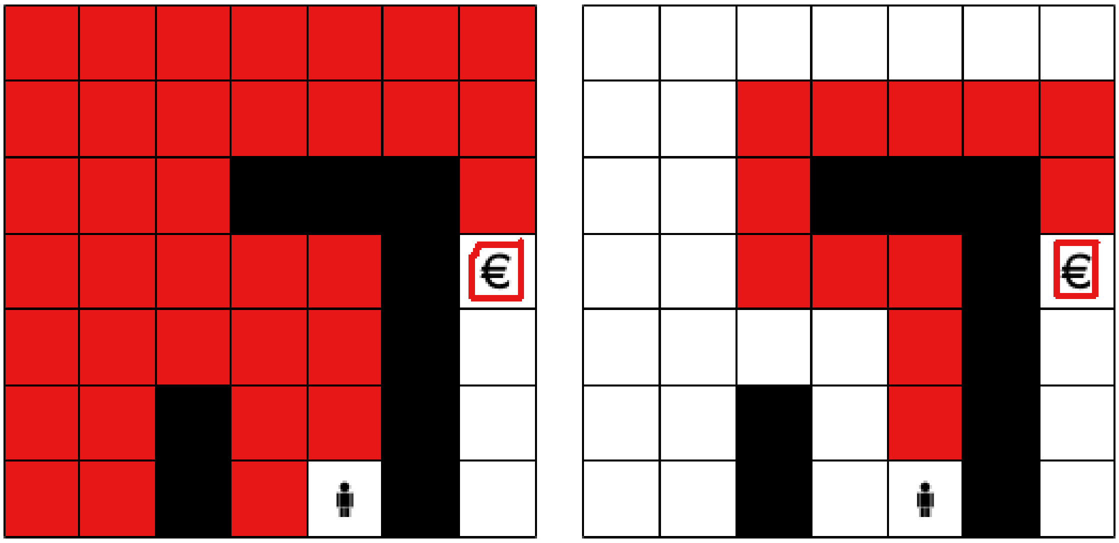


Figure 1

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.

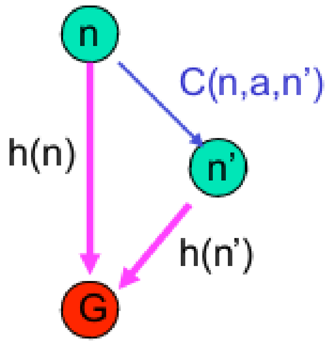
2 *Describe your best consistent heuristic for the mazeCollect problem. When using distance, precise which distance you use. Justify precisely the admissibility and the consistency of your* *heuristic.*

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. $h(n) \leq realCost(n)$: 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.*

```
'''NAMES OF THE AUTHOR(S): Baufays Benoit - Colmonts Julien'''

from search import *
import math


######################  Implement the search  #######################

class MazeCollect(Problem):
    #index for the money tuple in the state
    money_Index=1

    def __init__(self,init):
        #dictionnary of vide position
        self.vide=None
        #position of the goal
        self.goal=None
        self.createMap(init)


    def goal_test(self, state):
        if(state[0]=="init"):
            mapInfo=state[1]
        else:
            mapInfo=state

        return (len(mapInfo[self.money_Index])==0) and mapInfo[0]==self.goal


    def successor(self, state):
        if(state[0]=="init"):
            mapInfo=state[1]
        else:
            mapInfo=state
        retour=(tuple(self.IA(mapInfo)))
        return retour

    def IA(self,mapInfo):
        pos=[]
```

```
        position=mapInfo[0]
        #the four possibilities
        for a in [(-1,0),(0,-1),(1,0),(0,1)]:
            #take the money list
            moneyStack=list(mapInfo[self.money_Index])
            newPosition=(position[0]+a[0],position[1]+a[1])
            #if the new position is a entry of the vide dictionnary
            if self.getString(newPosition) in self.vide:
                if(newPosition in moneyStack):
                    #if it is a dollar case, we must recompute the minimal distance between all the dollars \{newPosition}
                    moneyStack.remove(newPosition)
                    saveMoney=list(mapInfo[self.money_Index])
                    saveMoney.remove(newPosition)
                    distance=self.dist(saveMoney)
                else:
                    distance=mapInfo[2]
                #we use the yield to return a iteraor.  With this, we use less memory
                yield (('move',((newPosition,tuple(moneyStack),distance))))

    #find the closest dollar of a position
    def closestDollar(self, dollars,position):
        closestDollar = None
        dst = 0

        if len(dollars) is not 0 :
            dst = self.manhattan(position, dollars[0])
            for elem in dollars :
                temp=self.manhattan(position, elem)
                if temp <= dst:
                    dst = temp
                    closestDollar = elem
        return closestDollar

    #see the rapport for a explication of this heuristique
    def Heuristique(self,node):
        state=node.state
        if(state[0]=="init"):
            mapInfo=state[1]
        else:
            mapInfo=state
        infoDollar=mapInfo[2]
```

```
        allNodes=list(mapInfo[self.money_Index])
        #if we have reach all dollar, we must now find the best
           past to the goal
        if len(allNodes) is 0:
            return self.manhattan(self.goal,mapInfo[0])
        else:
            closest=self.closestDollar(allNodes,mapInfo[0])
            return self.manhattan(mapInfo[0], closest) +mapInfo[2]


    #the manhattan distance between two points
    def manhattan(self, toPoint,fromPoint):
        return (math.fabs(toPoint[0] - fromPoint[0]) + math.fabs(
           toPoint[1] - fromPoint[1]))

    #get a String representation of a point (utility for the key
       of the vide dictionnary)
    def getString(self,point):
        final="{0}#{1}".format(point[0],point[1])
        return final

    #method to find the small path between all non reached dollars
    def dist(self,moneyT):
        dist=1000000000000
        money=list(moneyT)
        for node in money:
            nodeMin=None
            distance=0
            moneyStack=money
            moneyStack.remove(node)
            while len(moneyStack)>0:
                distMin=dist=self.manhattan(moneyStack[0],node)
                nodeMin=(moneyStack[0])
                for dollar in moneyStack:
                    dist=self.manhattan(dollar,node)
                    if(dist<distMin):
                        distMin=dist
                        nodeMin=dollar
                distance+=distMin
                moneyStack.remove(nodeMin)
            #we add the path between the last node and the goal
            if nodeMin is not None:
                distance+=self.manhattan(self.goal,nodeMin)
            if dist>distance:
                dist=distance

```

```python
        return dist


    #read the file, create the map and code it for the state schema
    def createMap(self,path):
        sizeY=0
        x=0
        y=-1
        moneyStack=[]
        debut=[]
        vide={}

        f = open(path,'r')
        for line in f:
            y=y+1
            sizeY=sizeY+1
            x=-1
            for elem in line:
                x=x+1
                if(elem!='#'and elem!="\n"):
                    if(elem=='$'):
                        moneyStack.append((x,y))
                    elif(elem=='@'):
                        debut=(x,y)
                    elif(elem=='+'):
                        self.goal=(x,y)
                    vide[self.getString((x,y))]=(x,y)


        saveMoneyStack=list(moneyStack)
        distance=self.dist(saveMoneyStack)
        mapInfo=('init',(debut,tuple(moneyStack),distance,(x,sizeY)))
        self.vide=vide
        self.initial=mapInfo


##################### Launch the search #######################

if(len(sys.argv)>1):
    problem=MazeCollect(sys.argv[1])
else:
    problem=MazeCollect("Benchs_Small/mazeCollect0")
node=astar_graph_search(problem, problem.Heuristique)
path=node.path()
```

```
path.reverse()
sizeElements=len(path[0].state[1])
size=path[0].state[1][sizeElements-1]
for n in path:
    a=['#' ] * size[1]
    for i in range(0,size[1]):
        a[i]=['#'] * size[0]
    state=n.state
    if(state[0]=="init"):
        state=state[1]

    #place libre
    freeSpaces=problem.vide
    for spaceS in freeSpaces:
        space=spaceS.split('#')
        a[int(space[1])][int(space[0])]=' '

    #coffre
    elem=problem.goal
    a[elem[1]][elem[0]]='+'

    #current position
    elem=state[0]
    a[elem[1]][elem[0]]='@'

    #money
    moneyStack=state[problem.money_Index]
    for money in moneyStack:
        a[money[1]][money[0]]='$'


    for ligne in a:
        ligneP=""
        for elem in ligne:
            ligneP=ligneP+elem
        print(ligneP)
    print('')
```

4 *Experiment, compare and analyze informed (astar_graph_search) and uninformed (breadth_first_graph_searc 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).*

```
'''NAMES OF THE AUTHOR(S): Baufays Benoit - Colmonts Julien'''

from search import *
import math

#we use a minimal mazeCollect to get the cost for a minimal path between two point
###############  class to find the small cost for one path ########
class MazeCollectMinimal(Problem):
    def __init__(self,mapInfo):
        #a dictionnary of all vide points of the map
        self.vide=mapInfo[3]
        #the goal
        self.goal=mapInfo[0]
        self.initial=('init',(mapInfo[1],mapInfo[2]))

    #get a String representation of a point (utility for the key of the vide dictionnary)
    def getString(self,point):
        final="{0}#{1}".format(point[0],point[1])
```

```
            return final

    def goal_test(self, state):
        if(state[0]=="init"):
            mapInfo=state[1]
        else:
            mapInfo=state

        return self.goal==mapInfo[0]

    def successor(self, state):
        if(state[0]=="init"):
            mapInfo=state[1]
        else:
            mapInfo=state
        retour=(tuple(self.IA(mapInfo)))
        return retour

    def IA(self,mapInfo):
        pos=[]
        position=mapInfo[0]
        for a in [(-1,0),(0,-1),(1,0),(0,1)]:
            newPosition=(position[0]+a[0],position[1]+a[1])
            if self.getString(newPosition) in self.vide:
                yield(('move',(newPosition,mapInfo[1])))

    #a basic heuristique that get the manhattan distance between the current position and the goal
    def Heuristique(self,node):
        state=node.state
        if(state[0]=="init"):
            mapInfo=state[1]
        else:
            mapInfo=state
        return self.manhattan(mapInfo[0],self.goal)

    #the manhattan distance between two points
    def manhattan(self, toPoint,fromPoint):
        #print(toPoint)
        return (math.fabs(toPoint[0] - fromPoint[0]) + math.fabs(toPoint[1] - fromPoint[1]))


######################  Implement the search ######################
class MazeCollect2(Problem):
```

```
    money_Index=1
    def __init__(self,init):
        #a dictionnary of all vide points of the map
        self.vide=None
        #the goal position
        self.goal=None
        self.createMap(init)


    def goal_test(self, state):
        if(state[0]=="init"):
            mapInfo=state[1]
        else:
            mapInfo=state

        return (len(mapInfo[1])==0) and mapInfo[0]==self.goal


    def successor(self, state):
        if(state[0]=="init"):
            mapInfo=state[1]
        else:
            mapInfo=state
        retour=(tuple(self.IA(mapInfo)))
        return retour


    def IA(self,mapInfo):
        pos=[]
        moneyStack=list(mapInfo[1])
        if len(moneyStack)>0:
            #each money is a successor
            for money in moneyStack:
                newMoneyStack=list(mapInfo[1])
                newMoneyStack.remove(money)
                yield (('move',((money,tuple(newMoneyStack)))))
        else:
            #we have not enough money to find, go to the goal
            yield (('move',((self.goal,tuple(moneyStack)))))


    #find the closest dollar of a position
    def closestDollar(self, dollars,position):
        closestDollar = None
        dst = 0

```

```python
        if len(dollars) is not 0 :
            dst = self.manhattan(position, dollars[0])
            for elem in dollars :
                temp=self.manhattan(position, elem)
                if temp <= dst:
                    dst = temp
                    closestDollar = elem
        return closestDollar

    #get a String representation of a point (utility for the key of the vide dictionnary)
    def getString(self,point):
        final="{0}#{1}".format(point[0],point[1])
        return final

    #method to find the small path between all non reached dollars
    def dist(self,moneyT):
        dist=1000000000000
        money=list(moneyT)
        for node in money:
            nodeMin=None
            distance=0
            moneyStack=money
            moneyStack.remove(node)
            while len(moneyStack)>0:
                distMin=dist=self.manhattan(moneyStack[0],node)
                nodeMin=(moneyStack[0])
                for dollar in moneyStack:
                    dist=self.manhattan(dollar,node)
                    if(dist<distMin):
                        distMin=dist
                        nodeMin=dollar
                distance+=distMin
                moneyStack.remove(nodeMin)
            #we add the path between the last node and the goal
            if nodeMin is not None:
                distance+=self.manhattan(self.goal,nodeMin)
            if dist>distance:
                dist=distance
        return dist

    def Heuristique(self,node):
        state=node.state
        if(state[0]=="init"):
            mapInfo=state[1]
        else:
```

```
            mapInfo=state
        allNodes=list(mapInfo[self.money_Index])
        if len(allNodes) is 0:
            return self.manhattan(self.goal,mapInfo[0])
        else:
            closest=self.closestDollar(allNodes,mapInfo[0])
            dist=self.dist(allNodes)
            return self.manhattan(mapInfo[0], closest) +dist

    #the manhattan distance between two points
    def manhattan(self, toPoint,fromPoint):
        return (math.fabs(toPoint[0] - fromPoint[0]) + math.fabs(toPoint[1] - fromPoint[1]))


    def path_cost(self, c, state1, action, state2):
        if(state1[0]=="init"):
            mapInfo=state1[1]
        else:
            mapInfo=state1

        problem=MazeCollectMinimal((mapInfo[0],state2[0],mapInfo[1],self.vide))
        node=astar_graph_search(problem, problem.Heuristique)
        path=node.path()
        return c + len(path)

    def createMap(self,path):
        sizeY=0
        x=0
        y=-1
        moneyStack=[]
        debut=[]
        vide={}

        f = open(path,'r')
        for line in f:
            y=y+1
            sizeY=sizeY+1
            x=-1
            for elem in line:
                x=x+1
                if(elem!='#'and elem!="\n"):
                    if(elem=='$'):
                        moneyStack.append((x,y))
                    elif(elem=='@'):
```

```
                            debut=(x,y)
                        elif(elem=='+'):
                            self.goal=(x,y)
                        vide[self.getString((x,y))]=(x,y)

        mapInfo=('init',(debut,tuple(moneyStack),(x,sizeY)))
        self.vide=vide
        self.initial=mapInfo

    def printOneState(self,n):
        a=['#' ] * size[1]
        for i in range(0,size[1]):
            a[i]=['#'] * size[0]
        state=n.state
        if(state[0]=="init"):
            state=state[1]

        freeSpaces=problem.vide
        for spaceS in freeSpaces:
            space=spaceS.split('#')
            a[int(space[1])][int(space[0])]=' '


        #money
        moneyStack=state[1]
        for money in moneyStack:
            a[money[1]][money[0]]='$'

        #coffre
        elem=self.goal
        a[elem[1]][elem[0]]='+'

        #current position
        elem=state[0]
        a[elem[1]][elem[0]]='@'


        for ligne in a:
            ligneP=""
            for elem in ligne:
                ligneP=ligneP+elem
            print(ligneP)
        print('')


##################### Launch the search #######################
```

```


if(len(sys.argv)>1):
    problem=MazeCollect2(sys.argv[1])
else:
    problem=MazeCollect2("Benchs_Small/mazeCollect0")
node=astar_graph_search(problem, problem.Heuristique)
path=node.path()
path.reverse()
sizeElements=len(path[0].state[1])
size=path[0].state[1][sizeElements-1]
tmp=None
number=0
for n in path:
    if tmp is not None:
        subProblem=MazeCollectMinimal((tmp[0],n.state[0],tmp[1],
            problem.vide))
        subNode=astar_graph_search(subProblem, problem.Heuristique
            )
        subNode=breadth_first_graph_search(subProblem)
        subPath=subNode.path()
        l=0
        subPath.reverse()
        for sub in subPath:
            if l>0:
                problem.printOneState(sub)
            l+=1
        number+=l-1
    if n.state[0] is "init":
        tmp=n.state[1]
        problem.printOneState(n)
    else:
        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 out.

	mazeCollect.py			mazeCollect2.py		
Benchs_Large	Time (s.)	Explored nodes	Steps	Time (s.)	Explored nodes	Steps
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.