EPL - ECOLE POLYTECHNIQUE DE LOUVAIN

LINGI2261 - ARTIFICIAL INTELLIGENCE

Report of first assignement:

The koutack Problem

Professor :

Yves DEVILLE

Students : (Group 1)

Benoît BAUFAYS 22200900

Julien COLMONTS 41630800

Program :

SINF21MS/G

Academic year 2013-2014

Contents

1 Aima-Python3 questions

Question 1 : In order to perform a search, what are the classes that you must define or extend? What are they used for?

You have to extend problem class. This new class must implement at least the following methods :

- init: the objective of this method is to initialize the problem environment. In our case, we build the map at the beginning of the game.
- successor: this method will return all the possible new actions, given by the problem rules, from a single state. The idea behind this function is to evaluate all the next states reachable from the current state.
- goal_test: this method is able to check if a state reaches the goal of the problem.
- path_cost: the objective of this method is to compute the cost from one state to another.

Question 2 : in the expand method of the class Node what is the advantage of using a yield instead of building a list and returning it afterwards?

The yield instruction is using the call-by-need strategy to delay variable initialisation to the moment the program need access to it. The advantage is a reduced space used by the program.

Question 3 : both breadth_first_graph_search and depth_first_graph_search are making a call to the same function. How is their fundamental difference implemented?

The main difference can be seen in the data structure used to look over the nodes. In breadth_first_graph_search, the data structure is first-in first-out queue. The order to run through different nodes is level per level. In depth_first_graph_search, the data structure is a simple stack. The order will be very different because the program always evaluate the successor of the last state computed. It will work more deeply than in terms of level.

Question 4 : what is the difference between the implementation of the graph_search and the tree_search methods? What is the special structure used in graph_search but not in tree_search;

what is it used for; and how is it implemented? The main difference between tree search and graph search is the permission to make repetition. The graph search avoid them, using a list to keep all states already stored.

Question 5 : what is the programming technique used to obtain a new node in the expand method of the Node class?
It's the lazy evaluation. As explained in question 2, it delays the evaluation of an expression to the moment it needs it.

Question 6 : how technically can you use the implementation of the closed list to deal with symmetrical states ? (hint: if two symmetrical states are considered to be the same, they will not be visited twice)
A clever way to solve this problem can be summarized in storing states avoiding symmetrical problem. In other words, you have to store informations in a formal way. For example, in the koutack problem, we can store a tuple with the action as first element and then all the letters in the alphabetical order.

Question 7 : what is the method used to get the path from the root to the solution Node you have found? Explain how this path is built starting from a solution Node. A node knows only his parent. The implementation just iterates over nodes from solution to parent until root is reached. Now, we have a way from solution to root, stored as a list. But, the goal is to have a way from root to solution, showing the right sequence of actions to reach the objective. To obtain this sequence, we just need to reverse this list, using reverse method from list class.

2 Koutack problem

Here are the answers to questions relative to the koutack problem.

Question 1 : Explain the advantages and weaknesses of each search strategy on this problem (not in general): depth first, breadth first, depth limited, iterative deepening, uniform cost.

- depth first : advantage of this search strategy is space used. This algorithm will keep in memory just one path. Completeness is reached in this case, because next depth in the search means that number of occupied squares decreased from one or more units. Completeness can be assimilated as an advantage. The most important weakness of this search strategy is that it can't ensure the optimal solution to be found and we need the shortest path to a solution. We can still use it, we know we'll find a solution if it exists.
- breadth first : this search strategy has the advantage of exploring all nodes of a level before going to the next one. Optimal solution will be found in the shortest time in front of all others search strategies. There is still a very important weakness that will dismiss this strategy to be used : the space used. Breadth first search keep the all tree in memory. For small cases, it's not a big deal and we can afford it. But as soon as the board grow a little, space used in memory climb to a very important size.
- depth limited : this strategy can't be used here because it can't even ensure to find a solution. The space advantage is better than in depth first search. But the completeness advantage of depth search (in this problem !) is cancelled because the limit couldn't be large enough to reach a solution.
- iterative deepening : this strategy combines advantages of depth first search and breadth first search. The only weakness here is the execution time. All the tree must be computed at all iterations of the limit depth.
- uniform cost : this strategy is exactly the same as breadth first search if the cost is calculated in terms of 'move' on the board. Each move will cost one unity so there

will be no benefit to compute it to change to order of the queue. It will be exactly the same as breadth first. If cost is computed in terms of number of occupied squares on the board combined with depth in the tree, it can be interessant to order the queue with the moves gathering more squares than others into the front.

Question 2 : Are there equivalent (symmetrical) states in this problem? What are the potential consequences on the search? Is it possible to handle them in your program?

Yes, there are equivalent states. In our implementation, the algorithm build the tree in searching moves for all occupied squares. In a same state, if square 'a' can merge with square 'b', the opposite is true too. In the beginning of the development, we didn't see that mistake. The execution time was really important. We can handle them using a graph search. We implemented the action statement to sort the list of letters gathered on a square in the alphabetic order. The state for a move to gather 'a' and 'b', or 'b' and 'a', will be **('move' coord 'a','b')**.

Question 3 : What are the advantages and disadvantages of using the tree and graph search for this problem. Which approach would you choose? Which approach allows you to avoid expending twice symmetrical states?

The problem with the tree search is that it allows symmetrical states. It doesn't keep a trace of all action made before in the tree. We chose graph search to avoid symmetrical states. We made the implementation accordingly to this choice as said in question above.

Question 4 : Implement this problem in Python 3. Extend the Problem class and implement the necessary methods and other class(es) if necessary. Your file must be named koutack.py. You program must print to the standard output a solution to the problem satisfying the above format.

Utilisation : python3 koutack.py FILEPATH [SEARCHTYPE]

Types de recherche disponibles :

1 : Graph - BFS

2 : Graph - DFS

3 : Tree - BFS

4 : Tree - DFS

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

from search import *


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

class Koutack(Problem):
    action=('init','gagne','merge','rien')
    def __init__(self,init):
        self.createMap(init)
        self.nodeExplored=0
        pass


    def goal_test(self, state):
        #if we have a size of 1 for state, we have only one case occuped so we have reached the goal
        if len(state)==1:
            return True
        else:
            return False


    def successor(self, state):
        if state[0]==self.action[0]:
            return self.successorInit(state)
        else:
            return self.getSuccessor(state)

    def getSuccessor(self, mapJeu):
        states=[]

        for choice in mapJeu:
            states.extend(self.ia(choice,mapJeu))
        self.nodeExplored+=1
        return tuple(states)


```

```
    #the state's format is different when you init the model (it's because, after one run, we save in the node the action)
    def successorInit(self, state):
        mapJeu=state[1]
        mapJeu=mapJeu[0:len(mapJeu)-1]

        return self.getSuccessor(mapJeu)


    #read the file, create the map and code it for the state schema
    def createMap(self,path):
        sizeY=0
        x=0
        y=-1
        mapInfo=[]
        f = open(path,'r')
        for line in f:
            y=y+1
            sizeY=sizeY+1
            x=-1
            for elem in line.split(' '):
                x=x+1
                if(elem!= '.' and elem != '.\n'):
                    mapInfo.append((x,y,elem.replace('\n', '')))
        mapInfo.append((x+1,sizeY))
        self.initial=('init',tuple(mapInfo))

    #ia method
    def ia(self,choice,mapJeu):
        states=[]
        #a position can  merge with is vertical friend
        for a in [-1,1]:
            for b in [-1,1]:
                possibilite=(choice[0]+a,choice[1]+b)
                #the new position is occuped by something ?
                value=self.isOccuped(mapJeu, possibilite)
                if value!='none' and value!='outofbound':
                    possibilite=(possibilite[0],possibilite[1],value)
                    #where can you merge ?
                    if(self.isOccuped(mapJeu, (choice[0]+a,choice[1]))=='none'):
                        newMap=list(mapJeu)
                        newMap.remove(possibilite)
                        newMap.remove(choice)
```

```
                            #is just one choice
                            newCase=(choice[0]+a,choice[1],self.putall(value,choice))
                            states.append(('move',tuple(self.getFriends(newCase,newMap,newCase[2]) )))
                        if(self.isOccuped(mapJeu, (choice[0],choice[1]+b))=='none'):
                            newMap=list(mapJeu)
                            newMap.remove(possibilite)
                            newMap.remove(choice)

                            newCase=(choice[0],choice[1]+b,self.putall(value,choice))
                            states.append(('move',tuple(self.getFriends(newCase,newMap,newCase[2]) )))
        #a position can also merge with a  friend 2 ligne
        for a in [(-2,0),(0,-2),(2,0),(0,-2)]:
            possibiliteA=(choice[0]+a[0],choice[1]+a[1])
            valueA=self.isOccuped(mapJeu, possibiliteA)
            if valueA!='none' and valueA!='outofbound':
                #where can you merge ?
                if(self.isOccuped(mapJeu, (choice[0]+(a[0]/2),choice[1]+(a[1]/2)))=='none'):
                    newMap=list(mapJeu)
                    possibiliteA=(possibiliteA[0],possibiliteA[1],valueA)
                    newMap.remove(possibiliteA)
                    newMap.remove(choice)
                    newCordoX=round(choice[0]+(a[0]/2))
                    newCordoY=round(choice[1]+(a[1]/2))
                    newCase=(newCordoX,newCordoY,self.putall(valueA,choice))
                    states.append(('move',tuple(self.getFriends(newCase,newMap,newCase[2]) )))
        if len(states)==0:
            #('nothing to do',mapJeu)
            return []
        return states

    #quand o na trouve un nouveau mouvement, o nregardes si d'autres cases adjacentes contiennent des elements afin de les empiler.
    def getFriends(self, position, mapJeu,values):
        valueL=list()
```

```
        for value in values:
            valueL.extend(list(value))
        for a in [(-1,0),(0,1),(1,0),(0,-1)]:
                possibilite=(position[0]+a[0],position[1]+a[1])
                value=self.isOccuped(mapJeu, possibilite)
                if value!='none' and value!='outofbound':
                    possibilite=(possibilite[0],possibilite[1],value)
                    mapJeu.remove(possibilite)
                    valueL.extend(value)
        valueL.sort()
        newCase=(position[0],position[1],tuple(valueL))
        mapJeu.append(newCase)
        return mapJeu


    def putall(self,value,choice):
        newV=list()
        for val in value:
            newV.extend(list(val))
        for val in choice[2]:
            newV.extend(list(val))

        valueL=newV

        valueL.sort()
        return tuple(valueL)

    #test if possibilite is occuped in mapJeu.  Si c'est le cas, il renvoit la valeur presente dans la case
    def isOccuped(self,mapJeu, possibilite):
        #pas chercher si pas dans le cadre
        if possibilite[0]<0 or possibilite[1]<0:
            return 'outofbound'
        for position in mapJeu:
            if possibilite[0]==position[0]and possibilite[1]==position[1]:
                return position[2]
        return 'none'
```

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

problem=Koutack(sys.argv[1])
#example of bfs search
if len(sys.argv)>2:
    searchType = int (sys.argv[2])
else:
    searchType = 1
if searchType == 1:
    node=breadth_first_graph_search(problem)
elif searchType == 2:
    node=depth_first_graph_search(problem)
if searchType == 3:
    node=breadth_first_tree_search(problem)
elif searchType == 4:
    node=depth_first_tree_search(problem)

#example of print
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]
    for elem in state:
        if len(elem)==3:
            a[elem[1]][elem[0]]=elem[2]
    for ligne in a:
        ligneP=""
        #print(ligne)
        for elem in ligne:
            if elem!='.':
                if len(elem)>1:
                    listE=list(elem)
                    elemStr=str(listE)[1:-1]
                    elemStr=elemStr.replace('\'','')
                    elemStr=elemStr.replace(' ','')
                    ligneP=ligneP+"["+elemStr+"] "
                else:
```

```
                            ligneP=ligneP+elem+"  "
                else:

                    ligneP=ligneP+'. '
            print(ligneP)
        print('')
#print('nodes to solution :' + str (len(path)-1) + ' nodes
    explored :'+ str (problem.nodeExplored))
```

Question 5 : Experiments must be realized with the 19 instances of the koutack problem provided. Report in a table the results on the 19 instances for depth-first and breadth- first strategies on both tree and graph search (4 settings). You must report the time, the number of explored nodes and the number of steps from root to solution. When no solution can be found by a strategy in a reasonable time (3 min), explain the reason (time-out and/or swap of the memory).

koutack0	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.25	0.04	19.24	0.05
Path to solution length	5	7	5	7
Explored Nodes	1066	8	137724	8

koutack1	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.53	0.04	110.15	0.05
Path to solution length	7	7	7	7
Explored Nodes	3634	12	1498271	18

koutack2	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.43	0.06	21.19	0.05
Path to solution length	5	7	5	7
Explored Nodes	1929	8	194321	8

koutack3	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.20	0.06	30.71	0.06
Path to solution length	7	8	7	8
Explored Nodes	1274	34	633101	188

koutack4	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.38	0.08	46.33	1.19
Path to solution length	6	7	6	7
Explored Nodes	1990	213	490811	30126

koutack5	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	155.58	0.05	Swap	0.54
Path to solution length	8	11	Swap	11
Explored Nodes	496111	125	Swap	14054

koutack6	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	20.50	0.17	Swap	11.07
Path to solution length	9	12	Swap	12
Explored Nodes	87429	1009	Swap	254675

koutack7	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	195.46	0.07	Swap	1.42
Path to solution length	9	11	Swap	11
Explored Nodes	761176	254	Swap	29961

koutack8	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	15.61	0.74	Swap	Time-out
Path to solution length	8	12	Swap	Time-out
Explored Nodes	55383	5287	Swap	Time-out

koutack9	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	107.48	0.06	Swap	0.07
Path to solution length	8	9	Swap	9
Explored Nodes	353479	37	Swap	314

koutackAG2	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.04	0.06	0.05	0.04
Path to solution length	2	2	2	2
Explored Nodes	8	2	25	2

koutackAG3	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.07	0.04	0.05	0.04
Path to solution length	4	4	4	4
Explored Nodes	26	9	175	12

koutackAG4	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.06	0.06	0.07	0.06
Path to solution length	4	4	4	4
Explored Nodes	54	4	481	4

koutackAG5	Graph Search		Tree Search	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.06	0.04	0.61	0.04
Path to solution length	5	6	5	6
Explored Nodes	134	23	7469	174

koutackAG6	Graph Search		Tree Search	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.23	0.05	16.99	0.12
Path to solution length	6	7	6	7
Explored Nodes	1582	58	257359	1497

koutackAG7	Graph Search		Tree Search	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	8.87	0.05	Swap	0.04
Path to solution length	7	8	Swap	8
Explored Nodes	44346	8	Swap	8

koutackAG8	Graph Search		Tree Search	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.13	0.11	9.55	1.22
Path to solution length	8	8	8	8
Explored Nodes	872	487	204767	28380

koutackAG9	Graph Search		Tree Search	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.06	0.03	1.36	0.04
Path to solution length	5	7	5	7
Explored Nodes	362	46	16471	476

koutackAG10	**Graph Search**		**Tree Search**	
	Breadth First	**Depth First**	**Breadth First**	**Depth First**
Time (s.)	0.76	0.05	104.10	0.06
Path to solution length	6	8	6	8
Explored Nodes	4521	13	990516	31

Question 6 : Imagine that a new 'split' move is introduced. The split move is a kind of inverse move; it allows to split any pile onto the free positions around. The pile should contain at least two tiles and there should be at least two free positions around the pile (up, right, down, left). The tiles are evenly distributed on the free positions (using the order up, right, down, left). What would be the impact of such a modification on the different search strategies (depth first, breadth first, depth limited, iterative deepening, uniform cost)? Also, what would be the impact on the tree and graph search?

Splitting piles have advantages and disadvantages. First, the advantage is that it can solve problems that are impossible with the standard koutack problem. But, it brings in a real problem in our algorithm : the program can be stuck in a infinite loop. Imagine a pile of three tiles. It splits and put 2 tiles up. Then, the new pile splits again, putting last tile up. The tiles can gather and the result is a pile which stand one square up than at the beginning. If you do the same move down, you go back to start state. You can do this two moves infinitely. So we'll have to use search strategy that dooesn't suffer from completeness issue. Depth first search will be prohibited. The advantages and weaknesses of the others strategies are equivalent in the split case but the general complexity is increased because new moves are availables. The graph search will be a important choice here to avoid repetition in splits.