Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Lazar Catalin and Vaida Diana
Group: 30433
Email:lazar.mihai@protonmail.com
Email:dianalaura5445@gmail.com

Teaching Assistant: Adrian Groza
Adrian.Groza@cs.utcluj.ro

Contents

Table 1: Lab scheduling

Activity	**Deadline**
Searching agents, Linux, Latex, Python, Pacman	W_1
Uninformed search	W_2
Informed Search	W_3
Adversarial search	W_4
Propositional logic	W_5
First order logic	W_6
Inference in first order logic	W_7
Knowledge representation in first order logic	W_8
Classical planning	W_9
Contingent, conformant and probabilistic planning	W_{10}
Multi-agent planing	W_{11}
Modelling planning domains	W_{12}
Planning with event calculus	W_{14}

Lab organisation.

1. Laboratory work is 25% from the final grade.
2. There are three deliverables in total: 1. Search, 2. Logic, 3. Planning.
3. Before each deadline, you have to send your work (latex documentation/code) at moodle.cs.utcluj.ro
4. We use Linux and Latex
5. Plagiarism: Don't be a cheater! Cheating affects your colleagues, scholarships and a lot more.

Chapter 1

A1: Search

We've started working on the file called 'Comparing Search Algorithm - Pacman Tournament' that is presented on the second laboratory work in Moodle.
We wanted to show how different types of algorithms work on the same kind of maze.
The project is a tournament between pacmans.

Chapter 2

A2: Logics

Another pacman

The template we worked on already had 2 pacmans. Some of the changes to add another pacman were in the template but others weren't so we had to find them. This doesn't seem much, because you can't really see 'externally' much, but we had to modify some lines. We added another pacman, adding or modifying the following lines -

In the graphicDisplay.py class

```
PACMAN_COLOR2 = formatColor(255.0/255.0,100.0/255.0,61.0/255)

self.textColor2 = PACMAN_COLOR2

self.scoreText2 = text( self.toScreen(400, 0), self.textColor2, ”SCORE: 0”, ”Times”,
self.fontSize, ”bold”)

changeText(self.scoreText2, ”SCORE: % 4d” % score)

if agent.isPacman==1 or agent.isPacman==0 or agent.isPacman==2:

if index==2:
outlineColor = PACMAN_COLOR2
fillColor= PACMAN_COLOR2

if self.capture:
outlineColor = TEAM_COLORS[index % 3]
```

In the layout.py class

```
elif layoutChar == 'P':
self.agentPositions.append((0, (x, y)))
self.agentPositions.append((1, (x, y)))
self.agentPositions.append((2, (x, y)))
```

In the pacman.py class

```
if agentIndex == 0 or agentIndex ==1 or agentIndex == 2:   Pacman is moving
return PacmanRules.getLegalActions( self,agentIndex )
else:
return GhostRules.getLegalActions( self, agentIndex )

if agentIndex == 0 or agentIndex == 1 or agentIndex == 2:   Pacman is moving
state.data._eaten = [False for i in range(state.getNumAgents())]
PacmanRules.applyAction( state, action, agentIndex )

if agentIndex == 0 or agentIndex == 1 or agentIndex == 2:
```

state.data.scoreChange += -TIME_PENALTY Penalty for waiting around
else:
GhostRules.decrementTimer(state.data.agentStates[agentIndex])

pacmanType2 = loadAgent(options.pacman2, noKeyboard)

agentOpts2= parseAgentArgs(options.agentArgs2)

pacman2 = pacmanType2(**agentOpts2)
args['pacman2'] = pacman2

def runGames(layout, pacman, pacman1, pacman2, ghosts, display, numGames, record, numTraining = 0, catchExceptions=False, timeout=30):

def newGame(self, layout, pacmanAgent, pacmanAgent1, pacmanAgent2, ghostAgents, display, quiet = False, catchExceptions=False):

agents = [pacmanAgent] + [pacmanAgent1] + [pacmanAgent2] +
ghostAgents[:layout.getNumGhosts()]

if agentIndex == 0 or agentIndex == 1 or agentIndex == 2: Pacmans just moved; Anyone can kill them
for index in range(3, len(state.data.agentStates)):

and the most important change, to give commands to the third pacman:
parser.add_option('-d', '–agentArgs2', dest='agentArgs2',
help='Comma separated values sent to agent2. e.g. "opt1=val1,opt2,opt3=val3"')

Weighted A* search

Weighted A*: expands states in the order of f = g+bh values, b bigger than 1 = bias towards states that are closer to goal
Next, we also implemented the A* search algorithm, with the weight as an input from the user. We noticed that in some mazes, A* weighted search behaves better than basic A* search. The route pacman is going from start to goal is the same, but the number of noded expanded decreases with the weight increasing.
We added another paramater 'weight'.
def aStarSearchw(problem, weight, heuristic):

The main difference between the A* search and the A* weighted search is in this line of code where weight is multiplied with the heuristic.
heuristicCost = newCost + float(weight)*heuristic(nextLocation, problem)

The changes are:
def __init__(self, fn='depthFirstSearch', prob='PositionSearchProblem', heuristic='nullHeuristic', weight=1):
self.searchFunction = lambda x: func(x, weight, heuristic=heur)
We had to add the weight=1 because even if the user doesn't add a weight one is needed, because modifying the program to ask for an input weight causes this.
The default weight is 1, but the user can input one if he wants to.

Chebyshev heuristic

In mathematics, Chebyshev distance (or Tchebychev distance) is a metric defined on a vector space where the distance between two vectors is the greatest of their differences along any

coordinate dimension. It is named after Pafnuty Chebyshev.
We also implement a new heuristic, Chebyshev distance. The user can pick it.
Chebyshev heuristic is better than Euclidean and Manhattan heuristics in almost any maze, because it allows you to move diagonally between two grid cells, resulting in less nodes expanded.
The function was implemented in the ’SearchAgents.py’ where are the other two heuristics already implemented.

Fringe search

In computer science, fringe search is a graph search algorithm that finds the least-cost path from a given initial node to one goal node. In essence, fringe search is a middle ground between A* and the iterative deepening A* variant (IDA*).
An important difference here between fringe and A* is that the contents of the lists in fringe do not necessarily have to be sorted - a significant gain over A*, which requires the often expensive maintenance of order in its open list. Unlike A*, however, fringe will have to visit the same nodes repeatedly, but the cost for each such visit is constant compared to the worst-case logarithmic time of sorting the list in A*.
We also implemented the Fringe search algorithm, which is a variation of the A* search one.
We used a priority queue and two arrays.

Adding eyes

We tried adding eyes to every pacman adding the following lines/modying already existing lines.

```
def circle2(pos, r, outlineColor, fillColor, endpoints=None, style=’pieslice’, width=2):
x, y = pos
x0, x1 = x - r - 1, x + r
y0, y1 = y - r - 1, y + r
if endpoints == None:
e = [0, 380]
else:
e = list(endpoints)
while e[0] >e[1] ]: e[1] = e[1] + 380
return _canvas.create_arc(x0, y0, x1, y1, outline=outlineColor, fill=fillColor, extent=e[1]-e[0],
start=e[0], style=style, width=width)
```

```
    def moveCircle2(id, pos, r)
global _canvas_x, _canvas_y
x, y = pos
x0, x1 = x - r - 1, x + r
y0, y1 = y - r - 1, y + r
move_to(id, x0, y0)
```

```
return [circle(screen_point, PACMAN_SCALE * self.gridSize,
fillColor = fillColor, outlineColor = outlineColor,
endpoints = endpoints,
width =width),circle2(screen_point, PACMAN_SCALE * 0.4*self.gridSize,
```

fillColor = eyes2, outlineColor = eyes,
endpoints = endpoints,width=width),circle2(screen_point, PACMAN_SCALE * 0.2*
self.gridSize,fillColor = eyes, outlineColor = eyes,endpoints = endpoints,width=width)]

Chapter 3

A3: Planning

Abbreviations
bfs = breadthFirstSearch
dfs = depthFirstSearch
astar = aStarSearch
ucs = uniformCostSearch
rs = randomSearch
ids = iterativeDeepeningSearch
astarw= aStarSearchw
fringe=Fringe

Bibliography

https://en.wikipedia.org/wiki/Fringe_search
https://www.researchgate.net/publication/221157471_Fringe_Search_Beating_A_at_Pathfindin
on_Game_Maps
Artificial Intelligence course on moodle, lab template -Comparing Search Algorithm - Pacman
Tournament
https://www.cs.cmu.edu/~motionplanning/lecture/Asearch_v8.pdf https://en.wikipedia.
org/wiki/Chebyshev_distance

Appendix A

Your original code

```
def chebyshevHeuristic(position,problem,info={}):
    "The Chebyshev distance heuristic for a PositionSearchProblem"
    xy1=position
    xy2=problem.goal
    return max(abs(xy1[0]-xy2[0]),abs(xy1[1]-xy2[1]))

def Fringe(problem, weight, heuristic):
    print("Start A* search:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState
    (problem.getStartState()))
    print("Start's successors:", problem.getSuccessors
    (problem.getStartState()))

    startingLocation = problem.getStartState()
    fringe = util.PriorityQueue()
    visited = []
    paths = []

    fringe.push((startingLocation, paths), heuristic(startingLocation,
    problem))

    while not fringe.isEmpty():
        state, solution = fringe.pop()
        "here help was needed from ChatGPT with the syntax, because the
         function pop() for the PriorityQueue unpacks 3 values, and we
         didn't know we can put them in 2"

        if problem.isGoalState(state):
            return solution

        if state not in visited:
            visited.append(state)
            successors = problem.getSuccessors(state)

            for successor in successors:
                next_state, action, step_cost = successor
                new_path = solution + [action]
                total_cost = len(new_path) + heuristic(next_state, problem)
                "help was needed here again in this line of code, for the
                 len(...), this was a problem that took a lot of time but
                 had a relatively easy solution"
                "python syntax is really something else"
                fringe.push((next_state, new_path), total_cost)
    util.raiseNotDefined()
```