Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Hirean Roxana - Irimes Cristina
Group: 30234
Email: roxana.hirean19@gmail.com - crisirimes@gmail.com

Teaching Assistant: Alexandru Lecu
lecu.alex12@gmail.com

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

1.1 Introducere:

Pac-Man este un joc video care a fost creat de Namco și proiectat de Toru Iwatani. A fost lansat în 1980 și a devenit foarte popular în istoria jocurilor.

În Pac-Man, jucătorul face un Pac-Man, un disc galben, să se miște într-un labirint. Fantomele sunt Blinky, Pinky, Inky și Clyde. Scopul este să mănânce fiecare cerc galben în timp ce nu este prins de fantome. Pentru puncte suplimentare, pot fi consumate și fructele care apar. Când Pac-Man mănâncă un cerc mare, fantomele devin albastre pentru o scurtă perioadă de timp și pot fi mâncate. Timpul în care fantomele sunt albastre scade în general de la o etapă la alta.

Primul capitol consta in probleme de cautare in contextul jocului Pacman, unde am implementat si am testat cele mai usoare si non-optimale solutii, dar si cele optimale. Problema de cautare are mai multe aspecte, dar vorbind in general, scopul ei este acela de a gasi cel mai scurt drum de la un punct de plecare la un golden point. Aici avem trei situatii in care startul si golden point-ul creeaza probleme diferite:

- Gasirea unei anumite pozitii din joc
- Vizitarea tuturor celor patru colturi ale scenei jocului
- Colectarea mancarii

In cadrul acestui joc, am implementat rezolvarea a doua probleme care apar:

1. Corners Problem si implementarea euristicii care rezolva aceasta problema
2. Eating All Food Problem si implementarea euristicii care rezolva aceasta problema

1. Corners Problem: In cadrul acestei teme, prima problema pe care o vom aborda este cea a vizitarii tuturor colturilor. Aici starea mai are, pe langa pozitie, o lista de patru elemente, care reprezinta cele 4 colturi care trebuie vizitate. Starea initiala este zero, iar dupa vizitarea pozitiei corespunzatoare, va deveni 1. In momentul in care toate pozitiile din lista vor fi 1, am ajus la golden point. Starea succesorului se poate afla in 3 situatii:

- Pozitia nu este un colt
- Pozitia este un colt vizitat
- Pozitia este un colt nevizitat

Primele doua semnifica faptul ca lista cu colturi vizitate ramane la fel ca si pentru starea curenta. Ultima presupune crearea unei noi liste cu colturile vizitate si marcarea pozitiei corespunzatoare ca fiind vizitata.

```
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.
    You must select a suitable state space and successor function
    """

    def __init__(self, startingGameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height - 2, self.walls.width - 2
        self.corners = ((1, 1), (1, top), (right, 1), (right, top))
        self.cornersStatus = []
        for corner in self.corners:
            self.cornersStatus.append('unknownTerritory')
            if not startingGameState.hasFood(*corner):
                print('Warning: no food in corner ' + str(corner))
        self._expanded = 0  # DO NOT CHANGE; Number of search nodes expanded
        # Please add any code here which you would like to use
        # in initializing the problem
        "*** YOUR CODE HERE ***"

        self.startPoint = self.startingPosition


    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman
        state space)
        """
        "*** YOUR CODE HERE ***"

        return self.startPoint, self.cornersStatus

        util.raiseNotDefined()

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"

        cornersStatus = []
        for item in state[1]:
            cornersStatus.append(item)
```

```
        return not ("unknownTerritory" in cornersStatus)

        util.raiseNotDefined()

    def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a cost of 1.
         As noted in search.py:
            For a given state, this should return
            a list of triples, (successor,
            action, stepCost),
            where 'successor' is a successor to the current
            state, 'action' is the action required
            to get there, and 'stepCost'
            is the incremental cost of expanding to that successor
        """

        successors = []
        for action in [Directions.NORTH,
        Directions.SOUTH, Directions.EAST,
        Directions.WEST]:
            # Add a successor state to the successor list
            if the action is legal
            # Here's a code snippet for figuring out
            whether a new position hits a wall:
            currentPosition = state[0][:]
            x,y = currentPosition
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            #hitsWall = self.walls[nextx][nexty]

            "*** YOUR CODE HERE ***"

            statusCorners = state[1][:]
            hitsWall = self.walls[nextx][nexty]

            if not hitsWall:
                posNxt = (nextx, nexty)
                if posNxt in self.corners:
                    statusCorners[self.corners.index(posNxt)] =
                "visitedTerritory"

                stateNxt = (posNxt, statusCorners)
                successors.append((stateNxt, action, 1, ))

        self._expanded += 1  # DO NOT CHANGE
        return successors

    def getCostOfActions(self, actions):
```

```
        """
        Returns the cost of a particular sequence of actions.
        If those actions
        include an illegal move,
        return 999999.  This is implemented for you.
        """
        if actions == None:
            return 999999
        x, y = self.startingPosition
        for action in actions:
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]:
                return 999999
        return len(actions)
```

Corners Heuristic: În primul rând, folosim euristica Manhattan pentru a calcula distanta intre două poziții și ignorăm toți pereții. În al doilea rând, luam în considerare că, dacă am ajuns deja la un colț, atunci trebuie doar să mergem de-a lungul granițelor hărții pentru a atinge toate colțurile nevizitate. Condiția cea mai ideală este să mergem mai întâi de-a lungul părții scurte a hărții și să atingem cel de-al doilea colț nevizitat, apoi mergem de-a lungul părții lungi a hărții pentru a atinge al treilea colț nevizitat, în cele din urmă mergem de-a lungul părții scurte a hărții pentru a atinge ultimul colț nevizitat. Presupunem că această condiție se întâmplă întotdeauna pentru a face euristica noastră admisibilă, așa că pentru a calcula valoarea euristică a acestei părți, trebuie să ne referim doar la numărul de colțuri nevizitate în loc de pozițiile acestora. În al treilea rând, presupunem că întotdeauna mergem mai întâi la cel mai apropiat colț nevizitat, apoi facem al doilea pas de mai sus, așa că trebuie doar să calculăm distanțele Manhattan dintre poziția curentă și colțurile nevizitate pentru a găsi cea mai mică distanță de adăugat la valoarea euristicii. După actualizarea valorii euristicii, înlocuim poziția curentă ca cel mai apropiat colț nevizitat și o facem din nou, până când toate colțurile sunt vizitate.

```
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.
      state:   The current search state
               (a data structure you chose in your search problem)
      problem: The CornersProblem instance for this layout.
    This function should always return a number
    that is a lower bound on the
    shortest path from the state to a goal
    of the problem; i.e.  it should be
    admissible (as well as consistent).
    """
    corners = problem.corners  # These are the corner coordinates
    # These are the walls of the maze, as a Grid (game.py)
    walls = problem.walls

    "*** YOUR CODE HERE ***"
```

```
from util import manhattanDistance

currentPosition = state[0]
statusCorners = state[1]
heuristic = 0

if problem.isGoalState(state):
    return heuristic

for index, item in enumerate(statusCorners):
    if item == "unknownTerritory":
        compare = manhattanDistance(
            currentPosition, corners[index])
        heuristic = compare if compare > heuristic else heuristic

return heuristic

return 0
```

2. Eating All Food Problem: Pentru a creste dificultatea jocului, Pac-Man va trebui sa manance toate punctele in cat mai putini pasi, spre deosebire de versiunea anterioara, cand trebuia doar sa ajunga in cele 4 colturi. In vederea solutionarii acestei probleme, ne folosim de functia FoodSearchProblem, deja implementata in proiect, si de functia FoodHeuristic, implementata mai jos.

Eating All Food Heuristic: Euristica returneaza distanta maxima dintre pozitia curenta si cel mai indepartat punct de mancare. La inceput, se verifica in lista daca avem puncte de mancare. Nu folosim Manhattan pentru ca ar genera o valoare maxima mai mare decat mazeDistance, pe care o avem definita. MazeDistance este pusa la dispozitie in searchAgents si calculeaza distanta exacta intre 2 puncte in grafic. Prin urmare, vom folosi mazeDistance pentru o valoare de prag mai mica.

```
def foodHeuristic(state, problem):
"""
Your heuristic for the FoodSearchProblem goes here.
This heuristic must be consistent to ensure correctness.
First, try to come
up with an admissible heuristic;
almost all admissible heuristics will be
consistent as well.
If using A* ever finds a solution
that is worse uniform cost search finds,
your heuristic is *not* consistent,
and probably not admissible! On the
other hand, inadmissible or inconsistent heuristics may find optimal
solutions, so be careful.
The state is a tuple ( pacmanPosition, foodGrid )
where foodGrid is a Grid
(see game.py) of either True or False. You can call foodGrid.asList()
to get a list of food coordinates instead.
```

```
    If you want access to info like walls, capsules, etc.,
    you can query the
    problem.  For example, problem.walls gives you a Grid
    of where the walls are.
    If you want to *store* information to be reused
    in other calls to the heuristic, there is a
    dictionary called problem.heuristicInfo that you can
    use. For example, if you only want to count the walls
    once and store that value, try: problem.heuristicInfo['wallCount'] =
    problem.walls.count()
    Subsequent calls to this heuristic can access
    problem.heuristicInfo['wallCount']
    """
    position, foodGrid = state
    "*** YOUR CODE HERE ***"

    startingPosition = problem.startingGameState
    listOfFood = foodGrid.asList()

    if len(listOfFood) == 0:
        return 0
    else:
        heuristic = -1
        for item in listOfFood:
            foodDistance = mazeDistance(position, item, startingPosition)
            heuristic = foodDistance if foodDistance > heuristic
            else heuristic

        return heuristic

    return 0
```

Find Path To Closest Dot: Aceasta functie gaseste distanta minima pana la un punct. Problema indică faptul că pana și o euristică bună nu ar reuși să găsească calea optimă într-un timp scurt. În acest caz, este mai realist să găsim o cale destul de bună, deși nu la fel de bună ca cea optimă, într-un timp relativ scurt. Unul dintre acesti agenți este unul care mănâncă întotdeauna punctul cel mai apropiat de Pac-Man. Clasa AnyFoodSearchProblem rezolva gasirea unui drum catre orice punct de mancare. Aceasta mosteneste metodele de la PositionSearchProblem si ajuta la implementarea functiei findPathToClosestDot

```
    class AnyFoodSearchProblem(PositionSearchProblem):
    """
    A search problem for finding a path to any food.
    This search problem is just like the PositionSearchProblem, but has a
    different goal test, which you need to fill in below.
    The state space and
    successor function do not need to be changed.
    The class definition above, AnyFoodSearchProblem(PositionSearchProblem),
    inherits the methods of the PositionSearchProblem.
    You can use this search problem to help you fill
```

```
    in the findPathToClosestDot
    method.
    """

    def __init__(self, gameState):
        "Stores information from the gameState.
        You don't need to change this."
        # Store the food for later reference
        self.food = gameState.getFood()

        # Store info for the PositionSearchProblem (no need to change this)
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        self.costFn = lambda x: 1
        self._visited, self._visitedlist, self._expanded = {}, [], 0
# DO NOT CHANGE

    def isGoalState(self, state):
        """
        The state is Pacman's position.
        Fill this in with a goal test that will
        complete the problem definition.
        """
        x, y = state

        "*** YOUR CODE HERE ***"

        listOfFood = self.food.asList()
        return (x, y) in listOfFood

        util.raiseNotDefined()

    def findPathToClosestDot(self, gameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """
        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

        "*** YOUR CODE HERE ***"

        return search.breadthFirstSearch(problem)

        util.raiseNotDefined()
```

Chapter 2

A2: Logics

Chapter 3

A3: Planning

Bibliography

Appendix A

Your original code

Don't be a cheater! Cheating affects your colleagues, scholarships and a lot more. This section should contain only code developed by you, without any line re-used from other sources. This section helps me to correctly evaluate your amount of work and results obtained.


```
class CornersProblem(search.SearchProblem):
"""
This search problem finds paths through all four corners of a layout.
You must select a suitable state space and successor function
"""

def __init__(self, startingGameState):
    """
    Stores the walls, pacman's starting position and corners.
    """
    self.walls = startingGameState.getWalls()
    self.startingPosition = startingGameState.getPacmanPosition()
    top, right = self.walls.height - 2, self.walls.width - 2
    self.corners = ((1, 1), (1, top), (right, 1), (right, top))
    self.cornersStatus = []
    for corner in self.corners:
        self.cornersStatus.append('unknownTerritory')
        if not startingGameState.hasFood(*corner):
            print('Warning: no food in corner ' + str(corner))
    self._expanded = 0  # DO NOT CHANGE; Number of search nodes expanded
   # Please add any code here which you would like to use
   # in initializing the problem
   "*** YOUR CODE HERE ***"

    self.startPoint = self.startingPosition
```

```
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman sta
    space)
    """
    "*** YOUR CODE HERE ***"

    return self.startPoint, self.cornersStatus

    util.raiseNotDefined()

def isGoalState(self, state):
    """
    Returns whether this search state is a goal state of the problem.
    """
    "*** YOUR CODE HERE ***"

    cornersStatus = []
    for item in state[1]:
        cornersStatus.append(item)

    return not ("unknownTerritory" in cornersStatus)

    util.raiseNotDefined()

def getSuccessors(self, state):
    """
    Returns successor states, the actions they require, and a cost of 1.
     As noted in search.py:
        For a given state, this should return
        a list of triples, (successor,
        action, stepCost),
        where 'successor' is a successor to the current
        state, 'action' is the action required
        to get there, and 'stepCost'
        is the incremental cost of expanding to that successor
    """

    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, I
        # Add a successor state to the successor list
        if the action is legal
        # Here's a code snippet for figuring out
        whether a new position hits a wall:
        currentPosition = state[0][:]
        x,y = currentPosition
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        #hitsWall = self.walls[nextx][nexty]
```

```
        "*** YOUR CODE HERE ***"

        statusCorners = state[1][:]
        hitsWall = self.walls[nextx][nexty]

        if not hitsWall:
            posNxt = (nextx, nexty)
            if posNxt in self.corners:
                statusCorners[self.corners.index(posNxt)]=
        "visitedTerritory"

            stateNxt = (posNxt, statusCorners)
            successors.append((stateNxt, action, 1, ))

    self._expanded += 1  # DO NOT CHANGE
    return successors

def getCostOfActions(self, actions):
    """
    Returns the cost of a particular sequence of actions.  If those actio
    include an illegal move, return 999999.  This is implemented for you.
    """
    if actions == None:
        return 999999
    x, y = self.startingPosition
    for action in actions:
        dx, dy = Actions.directionToVector(action)
        x, y = int(x + dx), int(y + dy)
        if self.walls[x][y]:
            return 999999
    return len(actions)

def cornersHeuristic(state, problem):
 """
 A heuristic for the CornersProblem that you defined.
   state:   The current search state
            (a data structure you chose in your search problem)
   problem: The CornersProblem instance for this layout.
 This function should always return a number
 that is a lower bound on the
 shortest path from the state to a goal
 of the problem; i.e.  it should be
 admissible (as well as consistent).
 """
 corners = problem.corners  # These are the corner coordinates
 # These are the walls of the maze, as a Grid (game.py)
 walls = problem.walls

 "*** YOUR CODE HERE ***"
```

```
from util import manhattanDistance

currentPosition = state[0]
statusCorners = state[1]
heuristic = 0

if problem.isGoalState(state):
    return heuristic

for index, item in enumerate(statusCorners):
    if item == "unknownTerritory":
        compare = manhattanDistance(
            currentPosition, corners[index])
        heuristic = compare if compare > heuristic else heuristic

return heuristic

return 0

def foodHeuristic(state, problem):
"""
Your heuristic for the FoodSearchProblem goes here.
This heuristic must be consistent to ensure correctness.
First, try to come
up with an admissible heuristic;
almost all admissible heuristics will be
consistent as well.
If using A* ever finds a solution
that is worse uniform cost search finds,
your heuristic is *not* consistent,
and probably not admissible! On the
other hand, inadmissible or inconsistent heuristics may find optimal
solutions, so be careful.
The state is a tuple ( pacmanPosition, foodGrid )
where foodGrid is a Grid
(see game.py) of either True or False. You can call foodGrid.asList()
to get a list of food coordinates instead.
If you want access to info like walls, capsules, etc.,
you can query the
problem.  For example, problem.walls gives you a Grid
of where the walls are.
If you want to *store* information to be reused
in other calls to the heuristic, there is a
dictionary called problem.heuristicInfo that you can
use. For example, if you only want to count the walls
once and store that value, try: problem.heuristicInfo['wallCount'] =
problem.walls.count()
Subsequent calls to this heuristic can access
problem.heuristicInfo['wallCount']
"""
position, foodGrid = state
```

```
    "*** YOUR CODE HERE ***"

    startingPosition = problem.startingGameState
    listOfFood = foodGrid.asList()

    if len(listOfFood) == 0:
        return 0
    else:
        heuristic = -1
        for item in listOfFood:
            foodDistance = mazeDistance(position, item, startingPosition)
            heuristic = foodDistance if foodDistance > heuristic
            else heuristic

        return heuristic

    return 0

    class AnyFoodSearchProblem(PositionSearchProblem):
    """
    A search problem for finding a path to any food.
    This search problem is just like the PositionSearchProblem, but has a
    different goal test, which you need to fill in below.
    The state space and
    successor function do not need to be changed.
    The class definition above, AnyFoodSearchProblem(PositionSearchProblem),
    inherits the methods of the PositionSearchProblem.
    You can use this search problem to help you fill
    in the findPathToClosestDot
    method.
    """

    def __init__(self, gameState):
        "Stores information from the gameState.
        You don't need to change this."
        # Store the food for later reference
        self.food = gameState.getFood()

        # Store info for the PositionSearchProblem (no need to change this)
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        self.costFn = lambda x: 1
        self._visited, self._visitedlist, self._expanded = {}, [], 0
# DO NOT CHANGE

    def isGoalState(self, state):
        """
        The state is Pacman's position.
        Fill this in with a goal test that will
        complete the problem definition.
        """
```

```
        x, y = state

        "*** YOUR CODE HERE ***"

        listOfFood = self.food.asList()
        return (x, y) in listOfFood

        util.raiseNotDefined()

    def findPathToClosestDot(self, gameState):
        """
        Returns a path (a list of actions) to the closest dot, starting from
        gameState.
        """
        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

        "*** YOUR CODE HERE ***"

        return search.breadthFirstSearch(problem)

        util.raiseNotDefined()
```