Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Sand Elena-Andreea
Group: 30233
Email: andreea-sand@yahoo.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

1.1 Introducere

Pacman este un joc de tip arcade, de indemanare, in care jucatorul il controleaza pe pac-man printr-un labirint, in incercarea de a manca toate punctele disponibile. Doar ca pac-man nu este singur, iar cele patru fantome fac tot posibilul sa-i iasa in cale. La un anumit nivel, in cele patru colturi ale labirintului, se afla patru puncte speciale care odata mancate, ii asigura lui pac-man supremația asupra fantomelor pe o durata de timp limitata.
In implementarea acestui proiect am aplicat o serie de concepte fundamentale AI, cum ar fi cautarea informata in spatiul static, inferenta probabilistica si invatarea prin consolidare.Aceste concepte stau la baza domeniilor de aplicare din lumea reală, cum ar fi procesarea limbajului natural, viziunea computerizată și robotica.
Acest capitol acopera problema cautarii in contextul jocului Pacman, pe baza caruia am implemenatat algoritmi de cautare.
Cautarea este de doua feluri:

- Cautare neinformata
  - DFS
    - Se utilizeaza o stiva pentru a adauga starile si o cale de la pozitia de inceput catre acea stare. Am utilizat stiva pentru a le putea scoate din lista in ordinea inversa a adaugarii. Se expandeaza fiecare nod din stiva adaugand vecinii sai si se verifica daca este scop inainte de eliminare.
  - BFS
    - Este asemanatoar DFS, diferenta fiind faptul ca la BFS se utilizeaza o coada in locul stivei, starile fiind expandate in ordinea in care au fost introduse on coada.
  - Uniform Cost Search
    - Algoritmul implementat este asemanator cu bfs, diferenta fiind ca UCS ia in calcul si costul drumului de la sursa la starea data. Este utilizata o coada de prioritati, nodurile fiind parcurse in ordine crescatoare a costului.
- Cautare informata
  - A* Search

* A* are o implementare asemanatoare cu cea a algoritmului BFS, diferentele fiind faptul ca in coada, pe langa pozitie si drumul spre acea pozitie din starea initiala, se mai adauga si valoarea unei functii, elementele scotandu se din coada in functie de valoarea acestui parametru. Functia se calculeaza adunand costul distantelor de la pozitia de start la pozitia respectiva si valoarea unei euristici in acel punct, aceasta reprezentand o aproximare a distantei fata de scop.

Problema căutării presupune găsirea celui mai scurt drum, de la poziția de start până poziția finală.
Am ales sa implementez:

- Corners Problem
- Euristica pentru Corners Problem
- Euristica pentru Eating All Food
- Closest Dot

1.Corners Problem

In fiecare colt al labirintului exita cate o bucata de mancare. Problema colturilor trebuie sa gaseasca cea mai scurta cale pentru pacman, astfel incat acesta sa treaca prin toate cele patru colturi. Starea state, pe langa pozitie, are o lista corners in care sunt stocate coordonatele celor patru colturi. Algoritmul se incheie cand pozitia curenta este ultimul colt din lista, verificand daca len(corners) este 1. Astfel In functia getSuccessors, pentru fiecare pozitie verificam daca aceasta este colt, in acest caz eliminand acel colt din lista corners (pentru ca a fost deja vizitat), iar in caz contrar, lista ramane la fel.

```
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))
    for corner in self.corners:
      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 ***"

    "vectorul pentru colturile vizitate"
    visited = [0, 0, 0, 0]
```

```
    self.start = (self.startingPosition, visited)

  def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    "*** YOUR CODE HERE ***"
    return self.startingPosition, self.corners
    util.raiseNotDefined()

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

    "*** YOUR CODE HERE ***"

    position, corners = state

    if position in corners and len(corners) ==1:
      return True
    return False
    util.raiseNotDefined()


  def getSuccessors(self, state):
    successors = []
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
   Directions.WEST]:
      pos,corners=state
      x,y = pos
      dx, dy= Actions.directionToVector(action)
      nextx, nexty= int(x+dx), int(y+dy)
      hitsWall= self.walls[nextx][nexty]
      if hitsWall == 0:
        if pos not in corners:
          nextState = ((nextx, nexty), corners)
        else:
          newCorners = []
          for corner in corners:
            if corner != pos:
              newCorners.append(corner)
          nextState = ((nextx, nexty), tuple(newCorners))
        successors.append((nextState, 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)
```

2.Corners Heuristic

Problema colturilor poate fi rezolvata cu orice algoritm de cautare, dar pentru rezultate mai bune este recomandat sa folosim o euristica pentru algoritmul A*.
Prima data am incercat sa merg de la pozitia curenta la cel mai apropiat colt, dupa care sa parcurg marginea labirintului pana vizitez toate colturile, dar aceasta euristica era inconsistenta.
In urmatoarea euristica am calculat distanta Manhattan de la pozitia curenta la fiecare colt nevizitat si am adunat la rezultat distanta maxima dintre acestea, insa si aceasta euristica a fost incnsistenta.
In urmatoarea euristica am adunat la rezultat distanta minima, iar rezultatele au fost mai bune. :)

- Numar noduri expandate: 1675

```
def cornersHeuristic(state, problem):

  corners = problem.corners # These are the corner coordinates
  walls = problem.walls # These are the walls of the maze, as a Grid (game.
   py)

  x, y = state[0]
  distances = []

  if problem.isGoalState(state):
    return 0

  for c in corners:
      x2, y2 = c
      #distance = abs(x-x2) + abs(y-y2)
      distance=util.manhattanDistance((x,y),(x2,y2))
      distances.append(distance)

  heuristic = min(distances)

  return heuristic
```

3.Eating All Food Heuristic

Contextul problemei este creat, existand o stare care contine pozitia actuala si o matrice care are valoarea True pe pozitiile pe care se afla mancare.
Am creat o euristica in care am calculat distanta Manhattan de la pozitia curenta la fiecare bucata de mancare si am ales distanta minima (adica am aflat pozitia celei mai apropiate bucati de mancare). Returnez mazeDistance (care foloseste bfs) de la pozitia curenta la cea mai apropiata bucata de mancare.

- Noduri expandate: 7452

```
def foodHeuristic1(state, problem):
  position, foodGrid = state
  foodList= foodGrid.asList()
  poz_min = position  # pozitia celei mai apropiate bucati de mancare
  dist_min = 999999
  for f in foodList:
    dist = util.manhattanDistance(position, f)
    if dist_min >= dist:  # alegem distanta minima
      dist_min = dist
      poz_min = f  # actualizam pozitia celei mai apropiate bucati de
   mancare
```

```
11
12  return mazeDistance(position, poz_min, problem.startingGameState)  #
     returnam distanta de la pozitia data la cea mai apropiata bucata de
     mancare
```

In a doua varianta am calculat distanta de la pozitia curenta la fiecare bucata de mancare, folosind mazeDistance, iar apoi am ales distanta maxima dintre acestea si am adaugat-o la rezultatul final.

- Noduri expandate: 4137

```
1     def foodHeuristic(state, problem):
2   Your heuristic for the FoodSearchProblem goes here.
3
4   This heuristic must be consistent to ensure correctness.  First, try to
     come
5   up with an admissible heuristic; almost all admissible heuristics will be
6   consistent as well.
7
8   If using A* ever finds a solution that is worse uniform cost search finds,
9   your heuristic is *not* consistent, and probably not admissible!  On the
10  other hand, inadmissible or inconsistent heuristics may find optimal
11  solutions, so be careful.
12
13  The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is a Grid
14  (see game.py) of either True or False. You can call foodGrid.asList() to
     get
15  a list of food coordinates instead.
16
17  If you want access to info like walls, capsules, etc., you can query the
18  problem.  For example, problem.walls gives you a Grid of where the walls
19  are.
20
21  If you want to *store* information to be reused in other calls to the
22  heuristic, there is a dictionary called problem.heuristicInfo that you can
23  use. For example, if you only want to count the walls once and store that
24  value, try: problem.heuristicInfo['wallCount'] = problem.walls.count()
25  Subsequent calls to this heuristic can access
26  problem.heuristicInfo['wallCount']
27
28  position, foodGrid = state
29  "*** YOUR CODE HERE ***"
30
31  distances=[]
32  foodList= foodGrid.asList()
33  if problem.isGoalState(state):
34    return 0
35  for f in foodList:
36    distance= mazeDistance(position, f, problem.startingGameState)
37    distances.append(distance)
38  rez = max(distances)
39  return rez
```

4.Closest Dot

Am modificat functiile isGoalState si findPathToClosestDot pentru a gasii cea mai scurta cale catre bucatile de mancare.Functia Is goal state verifica daca pe pozitie se afla vreo bucata de mancare, caz in care returneaza TRUE. Functia FindPathToClosestDot returneaza drumul pana la cea mai apropiata bucata de mancare, apeland functia de cautare BFS implementata la Q2.

```
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
  return state in self.food.asList() #returneaza true atunci cand pe
 pozitia data se afla mancare

  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 ***"
  problem = AnyFoodSearchProblem(gameState)
  return search.bfs(problem)
```

Tabel cu rezultate

Algorithm	Maze	Agent	Score	Cost	E.N	Time
DFS	tinyMaze	SearchAg	500	10	15	0.0
DFS	medMaze	SearchAg	380	130	146	0.3
DFS	bigMaze	SearchAg	300	210	390	0.2
BFS	tinyMaze	SearchAg	502	8	15	0.0
BFS	medMaze	SearchAg	442	68	269	0.1
BFS	bigMaze	SearchAg	300	210	620	0.1
UCS	tinyMaze	SearchAg	502	8	15	0.0
UCS	medMaze	SearchAg	442	68	269	0.1
UCS	bigMaze	SearchAg	300	210	620	0.2
A*	bigMaze	SearchAg	300	210	620	0.2
BFS	tinyCorners	SearchAg	512	28	269	0.0
BFS	medCorners	SearchAg	434	106	1988	0.2
A*	medCorners	A*CoAg	434	106	1675	0.3
A*	bigCorners	A*CoAg	378	162	6514	0.5
A*	testSearch	A*FoAg	513	7	11	0.0
A*	trickySearch	A*FoAg	570	60	7452	6.2
A*	bigSearch	CloDot	2360	350	-	-

Chapter 2

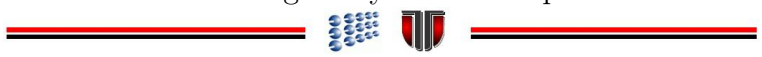
A2: Logics

3.Eating All Food

Chapter 3

A3: Planning

Bibliography



```
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))
    for corner in self.corners:
      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 ***"

    "vectorul pentru colturile vizitate"
    visited = [0, 0, 0, 0]
    self.start = (self.startingPosition, visited)

  def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    "*** YOUR CODE HERE ***"
    return self.startingPosition, self.corners
    util.raiseNotDefined()

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

```
41
42      "*** YOUR CODE HERE ***"
43
44      position, corners = state
45
46      if position in corners and len(corners) ==1:
47        return True
48      return False
49      util.raiseNotDefined()
50
51
52    def getSuccessors(self, state):
53      successors = []
54      for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
       Directions.WEST]:
55        pos,corners=state
56        x,y = pos
57        dx, dy= Actions.directionToVector(action)
58        nextx, nexty= int(x+dx), int(y+dy)
59        hitsWall= self.walls[nextx][nexty]
60        if hitsWall == 0:
61          if pos not in corners:
62            nextState = ((nextx, nexty), corners)
63          else:
64            newCorners = []
65            for corner in corners:
66              if corner != pos:
67                newCorners.append(corner)
68            nextState = ((nextx, nexty), tuple(newCorners))
69          successors.append((nextState, action, 1))
70      self._expanded += 1 # DO NOT CHANGE
71      return successors
72
73    def getCostOfActions(self, actions):
74      """
75      Returns the cost of a particular sequence of actions.  If those actions
76      include an illegal move, return 999999.  This is implemented for you.
77      """
78      if actions == None: return 999999
79      x,y= self.startingPosition
80      for action in actions:
81        dx, dy = Actions.directionToVector(action)
82        x, y = int(x + dx), int(y + dy)
83        if self.walls[x][y]: return 999999
84      return len(actions)
```

```
1 def cornersHeuristic(state, problem):
2
3   corners = problem.corners # These are the corner coordinates
4   walls = problem.walls # These are the walls of the maze, as a Grid (game.
     py)
5
6   x, y = state[0]
7   distances = []
8
9   if problem.isGoalState(state):
10    return 0
11
12  for c in corners:
13      x2, y2 = c
14      #distance = abs(x-x2) + abs(y-y2)
```

```
      distance=util.manhattanDistance((x,y),(x2,y2))
      distances.append(distance)

  heuristic = min(distances)

  return heuristic
```

```
def foodHeuristic1(state, problem):
  position, foodGrid = state
  foodList= foodGrid.asList()
  poz_min = position  # pozitia celei mai apropiate bucati de mancare
  dist_min = 999999
  for f in foodList:
    dist = util.manhattanDistance(position, f)
    if dist_min >= dist:  # alegem distanta minima
      dist_min = dist
      poz_min = f  # actualizam pozitia celei mai apropiate bucati de
   mancare

  return mazeDistance(position, poz_min, problem.startingGameState)  #
   returnam distanta de la pozitia data la cea mai apropiata bucata de
   mancare
```

```
    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 ***"

  distances=[]
  foodList= foodGrid.asList()
  if problem.isGoalState(state):
    return 0
  for f in foodList:
    distance= mazeDistance(position, f, problem.startingGameState)
```

```
    distances.append(distance)
  rez = max(distances)
  return rez
```

```

  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
    return state in self.food.asList() #returneaza true atunci cand pe
   pozitia data se afla mancare

    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 ***"
    problem = AnyFoodSearchProblem(gameState)
    return search.bfs(problem)
```