CSCE-4613

Artificial Intelligence

Assignment 3: Review

Spring 2023

Prof. Khoa Luu

Prepared by Thanh-Dat Truong

Data Structure

```python
# search.py
class SearchProblem:

    def getStartState(self):
        util.raiseNotDefined()

    def isGoalState(self, state):
        util.raiseNotDefined()

    def getSuccessors(self, state):
        util.raiseNotDefined()

    def getCostOfActions(self, actions):
        util.raiseNotDefined()
```

Data Structure

```python
# search.py
class SearchProblem:

    def getStartState(self):
        util.raiseNotDefined()

    def isGoalState(self, state):
        util.raiseNotDefined()

    def getSuccessors(self, state):
        util.raiseNotDefined()

    def getCostOfActions(self, actions):
        util.raiseNotDefined()
```

Return the start state

Data Structure

```python
# search.py
class SearchProblem:

    def getStartState(self):
        util.raiseNotDefined()

    def isGoalState(self, state):
        util.raiseNotDefined()

    def getSuccessors(self, state):
        util.raiseNotDefined()

    def getCostOfActions(self, actions):
        util.raiseNotDefined()
```

Return the start state

Return the state is whether the goal state or not

Data Structure

```python
# search.py
class SearchProblem:

    def getStartState(self):
        util.raiseNotDefined()

    def isGoalState(self, state):
        util.raiseNotDefined()

    def getSuccessors(self, state):
        util.raiseNotDefined()

    def getCostOfActions(self, actions):
        util.raiseNotDefined()
```

Return the start state

Return the state is whether the goal state or not

Return the successors of the current state

Data Structure

```python
# search.py
class SearchProblem:

    def getStartState(self):
        util.raiseNotDefined()

    def isGoalState(self, state):
        util.raiseNotDefined()

    def getSuccessors(self, state):
        util.raiseNotDefined()

    def getCostOfActions(self, actions):
        util.raiseNotDefined()
```

Return the start state

Return the state is whether the goal state or not

Return the successors of the current state

Computer the cost of a list of actions

Example: PositionSearchProblem

```python
class PositionSearchProblem(search.SearchProblem):
    """
    A search problem defines the state space, start state, goal test, successor
    function and cost function.  This search problem can be used to find paths
    to a particular point on the pacman board.

    The state space consists of (x,y) positions in a pacman game.

    Note: this search problem is fully specified; you should NOT change it.
    """

    def __init__(self, gameState, costFn = lambda x: 1, goal=(1,1), start=None, warn=True, visualize=True):
        """
        Stores the start and goal.

        gameState: A GameState object (pacman.py)
        costFn: A function from a search state (tuple) to a non-negative number
        goal: A position in the gameState
        """
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        if start != None: self.startState = start
        self.goal = goal
        self.costFn = costFn
        self.visualize = visualize
        if warn and (gameState.getNumFood() != 1 or not gameState.hasFood(*goal)):
            print 'Warning: this does not look like a regular search maze'

        # For display purposes
        self._visited, self._visitedlist, self._expanded = {}, [], 0 # DO NOT CHANGE
```

Example: PositionSearchProblem

```python
def getStartState(self):
    return self.startState

def isGoalState(self, state):
    isGoal = state == self.goal

    # For display purposes only
    if isGoal and self.visualize:
        self._visitedlist.append(state)
        import __main__
        if '_display' in dir(__main__):
            if 'drawExpandedCells' in dir(__main__._display): #@UndefinedVariable
                __main__._display.drawExpandedCells(self._visitedlist) #@UndefinedVariable

    return isGoal
```

Example: PositionSearchProblem

```python
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]:
        x,y = state
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:
            nextState = (nextx, nexty)
            cost = self.costFn(nextState)
            successors.append( ( nextState, action, cost) )

    # Bookkeeping for display purposes
    self._expanded += 1 # DO NOT CHANGE
    if state not in self._visited:
        self._visited[state] = True
        self._visitedlist.append(state)

    return successors
```

Example: PositionSearchProblem

```python
def getCostOfActions(self, actions):
    """
    Returns the cost of a particular sequence of actions. If those actions
    include an illegal move, return 999999.
    """
    if actions == None: return 999999
    x,y= self.getStartState()
    cost = 0
    for action in actions:
        # Check figure out the next state and see whether its' legal
        dx, dy = Actions.directionToVector(action)
        x, y = int(x + dx), int(y + dy)
        if self.walls[x][y]: return 999999
        cost += self.costFn((x,y))
    return cost
```

Problem 1: Corners Problem

```
class CornersProblem(search.SearchProblem):
    def __init__(self, startingGameState):
        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)
```

Problem 1: Corners Problem

```
class CornersProblem(search.SearchProblem):
  ...
    def getStartState(self):
        return (self.startingPosition,[])
```

The state contains the position and current visiting corner positions

Problem 1: Corners Problem

```
class CornersProblem(search.SearchProblem):
  …
    def getStartState(self):
        return (self.startingPosition,[])

    def isGoalState(self, state):
      xy = state[0]
      visitedCorners = state[1]
      if xy in self.corners:
        if not xy in visitedCorners:
          visitedCorners.append(xy)
        return len(visitedCorners) == 4
      return False
```

The state contains the position and current visiting corner positions

Problem 1: Corners Problem

```
class CornersProblem(search.SearchProblem):
  def getSuccessors(self, state):
    successors = []
    x,y = state[0]
    visitedCorners = state[1]
```

Problem 1: Corners Problem

```
class CornersProblem(search.SearchProblem):
  def getSuccessors(self, state):
    successors = []
    x,y = state[0]
    visitedCorners = state[1]
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
      dx, dy = Actions.directionToVector(action)
      nextx, nexty = int(x + dx), int(y + dy)
      hitsWall = self.walls[nextx][nexty]
```

Problem 1: Corners Problem

```
class CornersProblem(search.SearchProblem):
  def getSuccessors(self, state):
    successors = []
    x,y = state[0]
    visitedCorners = state[1]
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
      dx, dy = Actions.directionToVector(action)
      nextx, nexty = int(x + dx), int(y + dy)
      hitsWall = self.walls[nextx][nexty]
      if not hitsWall:
       successorVisitedCorners = list(visitedCorners)
       next_node = (nextx, nexty)
       if next_node in self.corners:
         if not next_node in successorVisitedCorners:
           successorVisitedCorners.append(next_node)
```

Problem 1: Corners Problem

```
class CornersProblem(search.SearchProblem):
  def getSuccessors(self, state):
    successors = []
    x,y = state[0]
    visitedCorners = state[1]
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
      dx, dy = Actions.directionToVector(action)
      nextx, nexty = int(x + dx), int(y + dy)
      hitsWall = self.walls[nextx][nexty]
      if not hitsWall:
       successorVisitedCorners = list(visitedCorners)
       next_node = (nextx, nexty)
       if next_node in self.corners:
         if not next_node in successorVisitedCorners:
           successorVisitedCorners.append(next_node)
       successor = ((next_node, successorVisitedCorners), action, 1)
       successors.append(successor)
```

Problem 1: Corners Problem

```
class CornersProblem(search.SearchProblem):
  def getSuccessors(self, state):
    successors = []
    x,y = state[0]
    visitedCorners = state[1]
    for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
      dx, dy = Actions.directionToVector(action)
      nextx, nexty = int(x + dx), int(y + dy)
      hitsWall = self.walls[nextx][nexty]
      if not hitsWall:
       successorVisitedCorners = list(visitedCorners)
       next_node = (nextx, nexty)
       if next_node in self.corners:
         if not next_node in successorVisitedCorners:
           successorVisitedCorners.append(next_node)
       successor = ((next_node, successorVisitedCorners), action, 1)
       successors.append(successor)
    self._expanded += 1 # DO NOT CHANGE
    return successors
```

Problem 1: Corners Problem

```
class CornersProblem(search.SearchProblem):
 def getCostOfActions(self, actions):
   if actions == None:
     return 999999 # Invalid
   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 # Invalid
   return len(actions)
```

Problem 1: Corners Problem

- Testing:
  - **python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem**
  - **python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem**

Problem 2: Corners Heuristic

```
def mazeDistance(point1, point2, gameState):
  x1, y1 = point1
  x2, y2 = point2
  walls = gameState.getWalls()
  assert not walls[x1][y1], 'point1 is a wall: ' + str(point1)
  assert not walls[x2][y2], 'point2 is a wall: ' + str(point2)
```

Problem 2: Corners Heuristic

```
def mazeDistance(point1, point2, gameState):
  x1, y1 = point1
  x2, y2 = point2
  walls = gameState.getWalls()
  assert not walls[x1][y1], 'point1 is a wall: ' + str(point1)
  assert not walls[x2][y2], 'point2 is a wall: ' + str(point2)
  prob = PositionSearchProblem(gameState = gameState,
                               start     = point1,
                               goal      = point2,
                               warn      = False,
                               visualize = False)
  return len(search.bfs(prob))
```

Problem 2: Corners Heuristic

```
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)
```

Problem 2: Corners Heuristic

```
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)

  xy = state[0]
  visitedCorners = state[1]
  unvisitedCorners = []
  for corner in corners:
    if not (corner in visitedCorners):
      unvisitedCorners.append(corner)
```

Problem 2: Corners Heuristic

```
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)

  xy = state[0]
  visitedCorners = state[1]
  unvisitedCorners = []
  for corner in corners:
    if not (corner in visitedCorners):
      unvisitedCorners.append(corner)

  heuristicvalue = [0]
  for corner in unvisitedCorners:
    heuristicvalue.append(mazeDistance(xy,corner,problem.startingGameState))

  return max(heuristicvalue)
```

Problem 2: Corners Heuristic

- Testing:
  - ***python pacman.py -l mediumCorners -p SearchAgent -a \ fn=aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic***

Problem 3: Eating All Food

```
def foodHeuristic(state, problem):
  position, foodGrid = state
  foodposition = foodGrid.asList()
```

Problem 3: Eating All Food

```
def foodHeuristic(state, problem):
  position, foodGrid = state
  foodposition = foodGrid.asList()

  heuristic = [0]
  for pos in foodposition:
    heuristic.append(mazeDistance(position,pos,problem.startingGameState))
  return max(heuristic)
```

Problem 3: Eating All Food

- Testing:
    - ***python pacman.py -l trickySearch -p SearchAgent ***
      -a fn=astar,prob=FoodSearchProblem,heuristic=foodHeuristic

Problem 4: Suboptimal Search

```
class AnyFoodSearchProblem(PositionSearchProblem):

  def __init__(self, gameState):
   self.food = gameState.getFood()
   self.walls = gameState.getWalls()
   self.startState = gameState.getPacmanPosition()
   self.costFn = lambda x: 1
   self._visited, self._visitedlist, self._expanded = {}, [], 0

  def isGoalState(self, state):
    x,y = state
    return state in self.food.asList()
```

Problem 4: Suboptimal Search

```
class ClosestDotSearchAgent(SearchAgent):

  def findPathToClosestDot(self, gameState):
    startPosition = gameState.getPacmanPosition()
    food = gameState.getFood()
    walls = gameState.getWalls()
    problem = AnyFoodSearchProblem(gameState)

    from search import breadthFirstSearch
    return breadthFirstSearch(problem)
```

Problem 4: Suboptimal Search

- Testing:
  - ***python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5***