Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Iobaj Andrei Sebastian
Group: 30433
Email: Iobaj.Se.Andrei@student.utcluj.ro

Name: Moldovan Alexandru Cristian
Group: 30433
Email: Moldovan.Io.Cristian@student.utcluj.ro

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

Introduction:

- This section of the lab activity is related to the development and implementation of multiple search related algorithms that will help Pacman reach different goals in more or less relaxed conditions. Also, algorithms related to the Eight Puzzle Problem will be presented as well, in this section.
- In order to achieve our goals, we have implemented multiple search algorithms that will be listed below, solutions to the Search All Food Problem and All Corners Problem, multiple heuristics and a multitude of ways for solving the Eight Puzzle Problem, plus a little easter egg that will be kept for when the time is right!

Search Algorithms:

Search problems are very common in Artificial Intelligence and involve a Search Agent, in our case, Pacman himself, to search for his goal in a maze. This is achieved via the use of Search Algorithms. Their purpose is to guide the Search Agent to the goal, through the shortest path or through the path with the smallest cost value.

In our project, we have implemented a few of these Search Algorithms. We will present them moving forward:

1. Depth First Search (DFS)

In the Depth First Search algorithm, the goal is to explore a graph by visiting a node and then, using recursion, traverse as far as possible along one of the branches. Only after we meet a visited cell or a leaf, we backtrack.

In our case, the graph is represented by a series of game states, representing Pacman's position, food dots and walls. The algorithm will try to explore as deeply as possible a path in the maze until it either collects all the food dots or meets a dead-end, backtracking to the most recently discovered path.

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
```

```
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:
    """

    "*** YOUR CODE HERE ***"

    moves = []
    s = []
    l = {}

    done = [False]

    l[problem.getStartState()] = True

    for v in problem.getSuccessors(problem.getStartState()):
        dfs_rec(problem, v, l, moves, done)
        if done[0]:
            break

    return moves
```

Listing 1.1: Depth First Search

2. Breadth First Search (BFS)

In the Breadth First Search Algorithm, the goal for searching a graph, starting with the root and exploring all the neighbours. The process is repeated, traversing the graph layer by layer. Unlike the DFS algorithm, no recursion is needed.

In our case, the root is the initial position of Pacman and the graph is the maze, just as before. We prioritize the exploration of all possible paths uniformly, thus ensuring that an optimal path will be found.

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"

    moves = []
    queue = []
    l = {}
    parent = {}
    directions = {}
    goal = None

    l[problem.getStartState()] = True
    queue.append(problem.getStartState())

    parent[problem.getStartState()] = None
    directions[problem.getStartState()] = None

    while queue:
        v = queue.pop(0)
        if problem.isGoalState(v):
            goal = v
            break
        for w in problem.getSuccessors(v):
```

```
            if w[0] not in l.keys():
                l[w[0]] = True
                parent[w[0]] = v
                queue.append(w[0])
                directions[w[0]] = w[1]

    while parent[goal]:
        moves.insert(0, directions[goal])
        goal = parent[goal]

    return moves
```

Listing 1.2: Breadth First Search

3. Uniform Cost Search (UCS)

In the Uniform Cost Search algorithm, just as the previously mentioned algorithms with the goal of exploration, this one is the same with one exception, it adds a cost. As such, the algorithm will prioritise moves with the least cumulative cost.

In our case, the algorithm will focus on finding the path with the shortest total cost, while also achieving all of the given goals.

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    moves = []
    queue = util.PriorityQueue()
    explored = []
    cost = {}
    cost[problem.getStartState()] = 0
    queue.push(problem.getStartState(), 0)
    parent = {}
    goal = None

    while not queue.isEmpty():
        v = queue.pop()
        cost_v = cost[v]

        if problem.isGoalState(v):
            goal = v
            break

        if v not in explored:
            explored.append(v)

            for w in problem.getSuccessors(v):
                cost_w = cost_v + w[2]
                if (w[0] not in cost.keys()) or (cost[w[0]] > cost_w):
                    cost[w[0]] = cost_w
                    parent[w[0]] = (v, w[1])
                    queue.push(w[0], cost_w)

    while goal != problem.getStartState():
        v = parent[goal]
        moves.insert(0, v[1])
        goal = v[0]

```

```
    return moves
```

Listing 1.3: Uniform Cost Search

4. A Star Search (A*)

In the A Star Search algorithm, considered more complex than the other algorithms previously mentioned, the goal remains, however, like in the case of UCS, with the addition of a cost, we also add a heuristic. The goal of this heuristic is to guide the agent with an approximation cost value to achieve the end goal. The closer this value is to the true total cost, the better the heuristic.

In our case, we implemented a number of heuristics to work with this algorithm. As such, we can help Pacman with a good estimation of the total cost, prioritising those paths that return the lowest values, optimising the performance and end result finding time.

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first
    ."""
    "*** YOUR CODE HERE ***"

    openSet = util.PriorityQueue()
    cameFrom = {}

    gScore = defaultdict(lambda: float('inf'))
    gScore[problem.getStartState()] = 0

    fScore = defaultdict(lambda: float('inf'))
    fScore[problem.getStartState()] = heuristic(problem.getStartState(),
    problem)

    openSet.push(problem.getStartState(), fScore[problem.getStartState()])

    goal = None
    moves = []

    while not openSet.isEmpty():
        current = openSet.pop()

        if problem.isGoalState(current):
            goal = current
            break

        for neighbour in problem.getSuccessors(current):
            tentativeGScore = gScore[current] + neighbour[2]

            if tentativeGScore < gScore[neighbour[0]]:
                cameFrom[neighbour[0]] = (current, neighbour[1])
                gScore[neighbour[0]] = tentativeGScore
                fScore[neighbour[0]] = tentativeGScore + heuristic(neighbour
    [0], problem)
                openSet.update(neighbour[0], fScore[neighbour[0]])

    while goal != problem.getStartState():
            v = cameFrom[goal]
            moves.insert(0, v[1])
            goal = v[0]
```

```
    return moves
```

Listing 1.4: A Star Search

5. Weighted A Star Search

In Weighted A Star Search algorithm, derived from the A star Search algorithm, we use the same principle as before, but we assign a weight and the goal is to find the path which minimizes the weighted sum of both the path cost and a heuristic estimate of the remaining cost to the goal.

In our case, we used the above mentioned principle and assigned the maze a weight of 1.5. We used the A* Search code for this, added the weight and as such, help the agent make better decisions that prioritize specific objectives.

```
def weightedAStarSearch(problem, weight = 1.5, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first
    ."""
    "*** YOUR CODE HERE ***"

    openSet = util.PriorityQueue()
    cameFrom = {}

    gScore = defaultdict(lambda: float('inf'))
    gScore[problem.getStartState()] = 0

    fScore = defaultdict(lambda: float('inf'))
    fScore[problem.getStartState()] = fScoreFunc(0, heuristic(problem.
    getStartState(), problem), weight)

    openSet.push(problem.getStartState(), fScore[problem.getStartState()])

    goal = None
    moves = []

    while not openSet.isEmpty():
        current = openSet.pop()

        if problem.isGoalState(current):
            goal = current
            break

        for neighbour in problem.getSuccessors(current):
            tentativeGScore = gScore[current] + neighbour[2]

            if tentativeGScore < gScore[neighbour[0]]:
                cameFrom[neighbour[0]] = (current, neighbour[1])
                gScore[neighbour[0]] = tentativeGScore
                fScore[neighbour[0]] = fScoreFunc(tentativeGScore, heuristic
    (neighbour[0], problem), weight)
                openSet.update(neighbour[0], fScore[neighbour[0]])

    while goal != problem.getStartState():
            v = cameFrom[goal]
            moves.insert(0, v[1])
            goal = v[0]
```

```python

    return moves

def fScoreFunc(g, h, w):
    if g < h:
        return g + h
    else:
        return (g + (2 * w - 1) * h) / w
```

Listing 1.5: Weighted A Star Search

6. Random Search

In Random Search algorithm, the strategy is to find the goal using random moves at each step. Surprisingly, the algorithm is complete, because, if the agent is given enough time, the task will be completed.

In our case, it is as simple as it gets, Pacman will move in a random direction at each step (obviously, it being a legal move). It will randomly navigate through the maze until all the food dots are collected.

```python
def randomSearch(problem):
    stack = util.Stack()
    stack.push(problem.getStartState())
    visited = []
    visited.append(problem.getStartState())
    cameFrom = {}
    moves = []
    goal = None
    while not stack.isEmpty():
        currentState = stack.pop()
        if problem.isGoalState(currentState):
            goal = currentState
            break

        successors = problem.getSuccessors(currentState)
        next = random.choice(successors)
        successors.remove(next)

        while next[0] in visited:
            if len(successors) == 0:
                break
            next = random.choice(successors)
            successors.remove(next)

        if next[0] not in visited:
            cameFrom[next[0]] = (currentState, next[1])
            stack.push(next[0])
            visited.append(next[0])

    if goal == None:
        return []

    while goal != problem.getStartState():
        v = cameFrom[goal]
        moves.insert(0, v[1])
        goal = v[0]
```

```
        return moves
```

Listing 1.6: Random Search

Search Problems:

More advanced search problems, such as Find all Corners Problem or Eat all Food Problem, are a good way to test the previously mentioned algorithms in more complex or dynamic scenarios and conclude if their behaviour is as good as expected.

In our work, we implemented solutions for both of the previously mentioned problems which will be presented below, with the code being provided in the Appendix A section

7. Find all Corners Problem

The Find all Corners Problem involves helping Pacman find the food situated in the 4 corners of the maze. The execution ends when Pacman successfully collected these 4 dots of food.

In our implementation we modified getStartState(), isGoalState() and getSuccessors() functions from CornersProblem and changed the state to be a tuple that contains the position and remaining corners. We modified getStartState() to return a tuple that contains the start state and the corners. In isGoalState() we simply check if there are no remaining corners and return True in that case, otherwise False. For getSuccessors() we check to see if the next position is not a wall, if is not, we compute the nextState as the next position and the remaining corners. If the next state is a corner, we remove it from the list of corners of that state. For the cornerHeuristic function we compute the distance to all remaining corners and return the maximum distance

Look at Appendix A.1 for the implementation

8. Eat all Food Problem

The Eat all Food Problem involves finding paths for Pacman to collect all the food dots within a given maze. The execution ends when all the food in the maze has been eaten.

For FoodHeuristic we used the already implemented mazeDistance() function to compute the distance between the current position and the food on the map. We then store the position to the nearest food and the position to the furthest one and also the minimum and maximum distance. We then compute the value of the heuristic as min distance + maze distance between nearestf ood and furthest food.

Look at Appendix A.2 for the implementation

9. Diagonal Search Problem

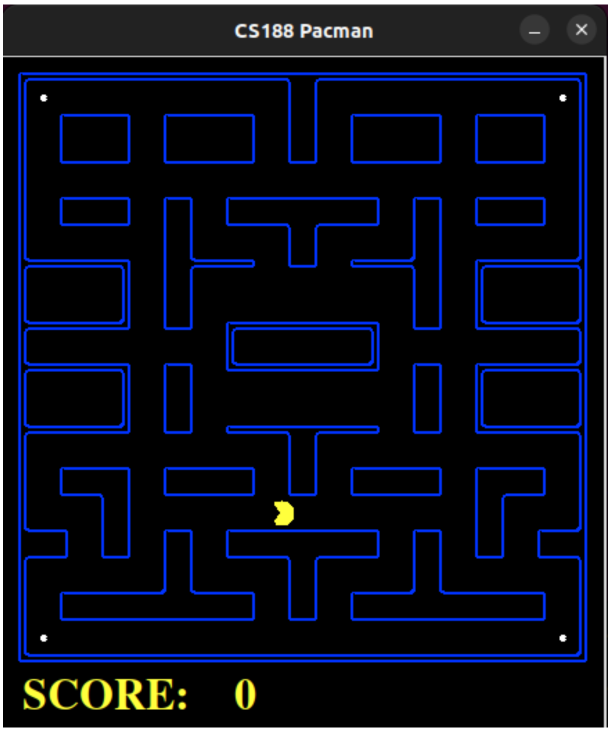
Diagonal Search Problem is a more relaxed search problem in which we allow Pacman to move not just up, down, right or left, but also in diagonal. This helps the agent find their goal faster by possibly skipping a few extra moves that would be required to achieve their goal.

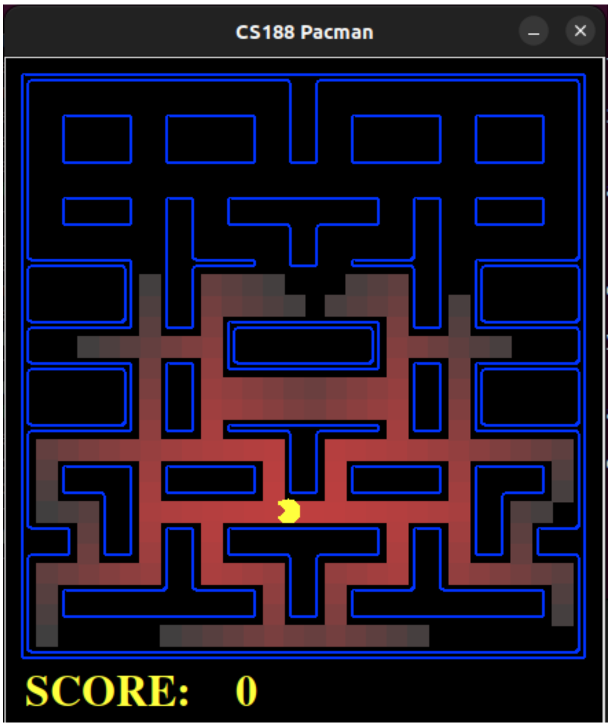
In our implementation we simply changed the way a successor state is found, by allowing Pacman to move to an increased number of neighbouring cells. After that, we let a search algorithm do their job until the goal is reached.

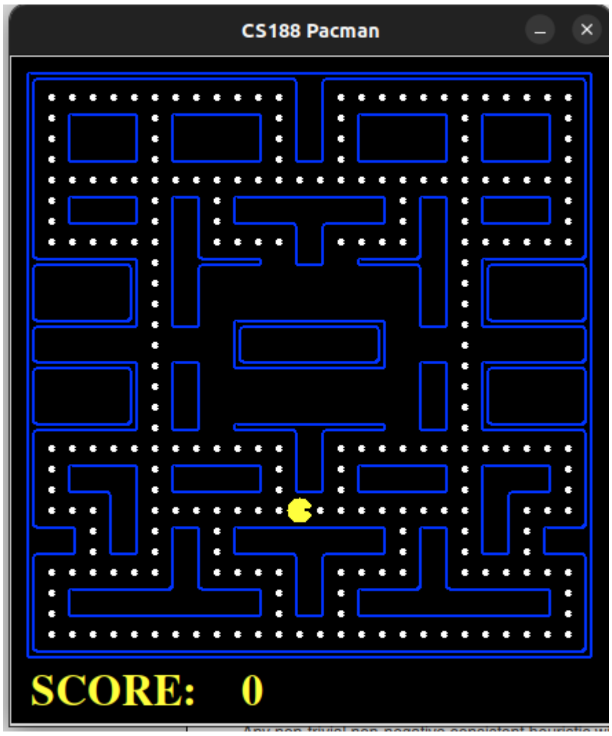
Look at Appendix A.3 for the implementation

10. Custom Maps

We decided to test the limits of our code and we put Pacman to the test on a few custom mazes we did. We managed to test his skills in a few different scenarios, showing promising results. After that, we made it more difficult by making a maze with a lot of food dots. Unfortunately, this maze takes a long time to complete and as such, we will only show the maze configurations here, but not his performance in any of them.







Eight Puzzle:

The Eight Puzzle is a classic problem in Artificial Intelligence and puzzle-solving. It consists of a 3x3 grid with eight numbered tiles and one empty space. The objective is to rearrange the tiles from an initial, presumably scrambled configuration to a well defined goal state. This problem is considered one of the most fundamental ones in AI, because it can use a multitude of search algorithms to reach the final goal.

Because we wanted to try something different, we moved on from the Pacman game and instead focused more of our time here, where we developed a multitude of ways we could solve the Eight Puzzle Problem, that we will present below.

What we worked on?

- The 8 Puzzle Problem can be run with the A* Search or Weighted A* Search, not with just BFS
- A multitude of heuristics
- A comparison table, between all of the implemented heuristics
- The possibility to run the program from the terminal with a set or predefined settings and options
- Larger sized puzzles(16 and 25)
- The Easter egg!

Let us discuss what we did!

• Algorithm runs with the A* or Weighted A* Search algorithms

We wanted to make things interesting and as such, we decided to switch the already implemented BFS algorithm with A* for more action!

At first we thought a simple copy paste of the code would work, making our life easy from the get-go. That was not the case and a bunch of compilation errors quickly made us realise we need to adapt the data structures for the problem at hand.

After the modifications were implemented (see Listing 1.4), the problem would get solved, but the time it took to get solved would differ, it not begin very time efficient. We quickly decided that we had to implement some better heuristics than the default Null Heuristc.

• Everyone gets a heuristic!

Because we were very curious about the topic of heuristic comparison, we implemented 6 heuristics and we wanted to compare them (more on than later). As such, we did a little research and picked 6 heuristics we wanted to implement and compare:

- Null heuristic -> default heuristic that always returns 0.
- Tile Misplaced heuristic -> counts the number of tiles that are not in the correct position, but only once and returns that value
- Manhattan Distance heuristic -> adds the the distance on the oX and oY axis to reach the goal and returns the sum.
- Euclidian Distance heuristic -> computes the the distance that goes straight to the goal (straight line) and returns it.

- Out of Row Out of Column heuristic -> counts on each row and column the number of misplaced tiles, returning the sum of the 2 values.
- Swap heuristic -> it computes the number of swaps it takes to place every single tile in the correct configuration and returns that value

Every one of them works and gives more or less different times of completion, with the same grid configuration, of course. Now, let us see how they fare against each other!

• Which one is the best one?

It is now time for some statistics! Below we have included the data we collected from the test runs we did. We decided to use the puzzle [5, 0, 8, 1, 4, 2, 3, 6, 7] for our testing, on a size of 3x3.

Heuristic	Completion Time	No. Expanded Nodes
Null Heuristic	14.888	7048
Tile Misp.	0.064	422
Manh. Dist.	0.026	150
Euclid. Dist.	0.030	167
Out Row Out Col.	0.019	145
Swap	0.234	902

From the above data we can draw some conclusions:

- The best heuristic is the Out of Row Out of Column heuristic, with the smallest number of expanded nodes, this was a surprise, because we thought that the best one would be the Manhattan distance heuristic, which came in second place.
- The worst heuristic in the Null heuristic. No big surprise here, considering it always returns 0, helping with basically nothing the searching algorithm. So, informed heuristics have a big impact on the performance of the search.
- While it's true that the data does not lie, it may be possible that, on other grid configurations, another heuristic may prove more efficient than the Out of Row Out of Column heuristic. This means that every heuristic has its own purpose and the table above is specific to our example and we cannot conclude for sure, which heuristic is, objectively speaking, the best.

• Running the program from the terminal

We wanted to make the program run through the terminal. As such, we grouped the program better into functions, imported a few libraries and set the options for the run commands. It is possible to set the heuristic from the 6 possible heuristics (by default, Null Heuristic is selected) and the size of the grid (by default, 3x3 is selected) and the function (by default, A* is selected).

In order to use the command line arguments, you have to write a syntax like:

python eightpuzzle.py --heuristic <NameOfHeuristic> -s <Size as an Int> -f <Name of the function>

- Null heuristic -> by default
- Tile Misplaced heuristic -> "tileMisplaced"
- Manhattan Distance heuristic -> "manhattan"
- Euclidian Distance heuristic -> "euclidian"
- Out of Row Out of Column heuristic -> "outOfColumnRow"
- Swap heuristic -> "swap"

- A* Search -> "astar", or by default
- Weighted A* Search -> "wastar"
- Stalin Sort (more on this later) -> "StalinSort"

- Larger grids!

We wanted to see if our implementations hold true for larger puzzles. We changed some hard-coded sizes, made a few adjustments for our program to be able to take a desired size and it would appear that the code works great!

- Easter egg hunt!

The following is a PAMPHLET, treat it as such!

When we did a little research on a few more 'not so general' functions we could implement for our project, we stumbled on a very sketchy site with a few strange functions that work, but are a lot more memes than actual coding. Here, we saw a type of sorting called "Stalin Sort". Yes, it actually exists and works like this: We travel an array and if an elemnt is not in the correct position for the array to be sorted, it is simply 'executed' (taken out of the array) and you repeat this until the array becomes sorted. The number of elements left is irrelevant, just like in real life, with the political adversaries of Stalin.

Because we wanted something to make the project our own, we decide to implement this kind of logic in the Eight Puzzle, simply taking out the elements that, by default, are not in the correct position. Quick and easy! *Pew*

The code can be found in the Appendix!

Chapter 2

A2: Logics

Chapter 3

A3: Planning

Bibliography

- AI Courses on Moodle from Adrian Groza
- https://en.wikipedia.org/wiki/Depth-first_search
- https://en.wikipedia.org/wiki/Breadth-first_search
- https://en.wikipedia.org/wiki/A*_search_algorithm
- http://theory.stanford.edu/~amitp/GameProgramming/Variations.html
- https://cse.iitk.ac.in/users/cs365/2009/ppt/13jan_Aman.pdf
- https://stackoverflow.com/questions/9994913/pacman-what-kinds-of-heuristics-are-ma
  36404229#36404229
- http://ai.berkeley.edu/search.html

Appendix A

Your original code

Note: Depth First Search, Breadth First Search, Uniform Cost Search, A Star Search, Weighted A Star Search and Random Search were all implemented using the pseudocode of the respective algorithm, using original work and ideas for the implementation.

```python
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 ***"

    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)

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

        return len(state[1]) == 0
```

```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]:
            # Add a successor state to the successor list if the action is legal
            x,y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextx][nexty]
            if not hitsWall:
                remainingGoals = tuple(element for element in state[1] if element != (nextx, nexty))
                nextState = ((nextx, nexty), remainingGoals)
                cost = 1
                successors.append((nextState, action, cost))
            "*** YOUR CODE HERE ***"

        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)


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

        "*** YOUR CODE HERE ***"

        leng = 0

        x0, y0 = state[0]
        for element in state[1]:
            x1, y1 = element
            leng = max(abs(x1 - x0) + abs(y1 - y0), leng)

        return leng
```

Listing A.1: Find all Corners Problem

```
def foodHeuristic(state, problem):
    position, foodGrid = state
    minDistance = 99999
    maxDistance = -1
    nearestFood = None
    furthestFood = None

    for element in foodGrid.asList():
        distance = mazeDistance(position, element, problem.startingGameState)
        if distance < minDistance:
            minDistance = distance
            nearestFood = element
        if distance > maxDistance:
            maxDistance = distance
            furthestFood = element

    if nearestFood == None and furthestFood == None:
        return 0
    else:
        return minDistance + mazeDistance(nearestFood, furthestFood, problem.startingGameState)
```

Listing A.2: Eat all Food Problem

```
class DiagonalSearchProblem(search.SearchProblem):
    def __init__(self, gameState, costFn = lambda x: 1, goal=(1,1), start=None, warn=True, visualize=True):
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        if start != None: self.startState = start
        self.goal = goal
        self.costFn = costFn
        self.visualize = visualize
```

```python
        if warn and (gameState.getNumFood() != 1 or not gameState.hasFood(*goal)):
            print 'Warning: this does not look like a regular search maze'

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

    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

    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, Directions.NORTHEAST, Directions.NORTHWEST, Directions.SOUTHEAST, Directions.SOUTHWEST]:
            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

    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
```

Listing A.3: Diagonal Search Problem

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%            %%            %
% %%%% %%%%% %% %%%%% %%%% %
% %%%% %%%%% %% %%%%% %%%% %
% %%%% %%%%% %% %%%%% %%%% %
%                          %
% %%%% %% %%%%%%%% %% %%%% %
% %%%% %% %%%%%%%% %% %%%% %
%      %%    %%    %%      %
%%%%%% %%%%% %% %%%%% %%%%%%
%    % %%          %% %    %
%    % %%          %% %    %
%%%%%% %% %%%%%%%% %% %%%%%%
%         %      %         %
%%%%%% %% %%%%%%%% %% %%%%%%
%    % %%          %% %    %
%    % %%          %% %    %
%%%%%% %% %%%%%%%% %% %%%%%%
%            %%            %
% %%%% %%%%% %% %%%%% %%%% %
% %%%% %%%%% %% %%%%% %%%% %
%   %%       P        %%   %
%%% %% %% %%%%%%%% %% %% %%%
%%% %% %% %%%%%%%% %% %% %%%
%      %%    %%    %%      %
% %%%%%%%%%% %% %%%%%%%%%% %
% %%%%%%%%%% %% %%%%%%%%%% %
%.                         %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing A.4: Search Maze

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%............%%............%
%.%%%%.%%%%%.%%.%%%%%.%%%%.%
%.%%%%.%%%%%.%%.%%%%%.%%%%.%
%.%%%%.%%%%%.%%.%%%%%.%%%%.%
%..........................%
%.%%%%.%%.%%%%%%%%.%%.%%%%.%
%.%%%%.%%.%%%%%%%%.%%.%%%%.%
%......%%....%%....%%......%
```

```
%%%%%%.%%%%% %% %%%%%.%%%%%%
%    %.%%          %%.%    %
%    %.%%          %%.%    %
%%%%%%.%% %%%%%%%% %%.%%%%%%
%     .   %      %   .     %
%%%%%%.%% %%%%%%%% %%.%%%%%%
%    %.%%          %%.%    %
%    %.%%          %%.%    %
%%%%%%.%% %%%%%%%% %%.%%%%%%
%............%%............%
%.%%%%.%%%%%.%%.%%%%%.%%%%.%
%.%%%%.%%%%%.%%.%%%%%.%%%%.%
%...%%.......P........%%...%
%%%.%%.%%.%%%%%%%%.%%.%%.%%%
%%%.%%.%%.%%%%%%%%.%%.%%.%%%
%......%%....%%....%%......%
%.%%%%%%%%%%.%%.%%%%%%%%%%.%
%.%%%%%%%%%%.%%.%%%%%%%%%%.%
%..........................%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing A.5: Food Maze

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%.           %%           .%
% %%%% %%%%% %% %%%%% %%%% %
% %%%% %%%%% %% %%%%% %%%% %
% %%%% %%%%% %% %%%%% %%%% %
%                          %
% %%%% %% %%%%%%%% %% %%%% %
% %%%% %% %%%%%%%% %% %%%% %
%      %%    %%    %%      %
%%%%%% %%%%% %% %%%%% %%%%%%
%    % %%          %% %    %
%    % %%          %% %    %
%%%%%% %% %%%%%%%% %% %%%%%%
%         %      %         %
%%%%%% %% %%%%%%%% %% %%%%%%
%    % %%          %% %    %
%    % %%          %% %    %
%%%%%% %% %%%%%%%% %% %%%%%%
%            %%            %
% %%%% %%%%% %% %%%%% %%%% %
% %%%% %%%%% %% %%%%% %%%% %
%   %%       P        %%   %
%%% %% %% %%%%%%%% %% %% %%%
%%% %% %% %%%%%%%% %% %% %%%
%      %%    %%    %%      %
% %%%%%%%%%% %% %%%%%%%%%% %
% %%%%%%%%%% %% %%%%%%%%%% %
%.                        .%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Listing A.6: Corners Maze

The following is the Eight Puzzle class. We modified something in every function of this class, as an example the size of the maze.

```
    Eight puzzle:

# eightpuzzle.py
# --------------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.
#
# Attribution Information: The Pacman AI projects were developed at UC
    Berkeley.
# The core projects and autograders were primarily created by John DeNero
# (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
# Student side autograding was added by Brad Miller, Nick Hay, and
# Pieter Abbeel (pabbeel@cs.berkeley.edu).


import search
import random
import time
import sys
from optparse import OptionParser

# Module Classes

size = 3

class EightPuzzleState:
    """
    The Eight Puzzle is described in the course textbook on
    page 64.

    This class defines the mechanics of the puzzle itself.  The
    task of recasting this puzzle as a search problem is left to
    the EightPuzzleSearchProblem class.
    """

    def __init__( self, numbers ):
        """
          Constructs a new eight puzzle from an ordering of numbers.

        numbers: a list of integers from 0 to 8 representing an
          instance of the eight puzzle.  0 represents the blank
          space.  Thus, the list

            [1, 0, 2, 3, 4, 5, 6, 7, 8]

          represents the eight puzzle:
            -------------
            | 1 |   | 2 |
            -------------
            | 3 | 4 | 5 |
            -------------
            | 6 | 7 | 8 |
            ------------

        The configuration of the puzzle is stored in a 2-dimensional
        list (a list of lists) 'cells'.
```

```
        """
        self.cells = []
        numbers = numbers[:] # Make a copy so as not to cause side-effects.
        numbers.reverse()
        for row in range(size):
            self.cells.append( [] )
            for col in range(size):
                self.cells[row].append( numbers.pop() )
                if self.cells[row][col] == 0:
                    self.blankLocation = row, col

    def isGoal( self ):
        """
          Checks to see if the puzzle is in its goal state.

            -------------
            |   | 1 | 2 |
            -------------
            | 3 | 4 | 5 |
            -------------
            | 6 | 7 | 8 |
            -------------

        >>> EightPuzzleState([0, 1, 2, 3, 4, 5, 6, 7, 8]).isGoal()
        True

        >>> EightPuzzleState([1, 0, 2, 3, 4, 5, 6, 7, 8]).isGoal()
        False
        """
        current = 0
        for row in range(size):
            for col in range(size):
                if current != self.cells[row][col]:
                    return False
                current += 1
        return True

    def legalMoves( self ):
        """
          Returns a list of legal moves from the current state.

        Moves consist of moving the blank space up, down, left or right.
        These are encoded as 'up', 'down', 'left' and 'right' respectively.

        >>> EightPuzzleState([0, 1, 2, 3, 4, 5, 6, 7, 8]).legalMoves()
        ['down', 'right']
        """
        moves = []
        row, col = self.blankLocation
        if(row != 0):
            moves.append('up')
        if(row != size - 1):
            moves.append('down')
        if(col != 0):
            moves.append('left')
        if(col != size - 1):
            moves.append('right')
        return moves

    def result(self, move):
```

```
        """
          Returns a new eightPuzzle with the current state and blankLocation
        updated based on the provided move.

        The move should be a string drawn from a list returned by legalMoves.
        Illegal moves will raise an exception, which may be an array bounds
        exception.

        NOTE: This function *does not* change the current object.  Instead,
        it returns a new object.
        """
        row, col = self.blankLocation
        if(move == 'up'):
            newrow = row - 1
            newcol = col
        elif(move == 'down'):
            newrow = row + 1
            newcol = col
        elif(move == 'left'):
            newrow = row
            newcol = col - 1
        elif(move == 'right'):
            newrow = row
            newcol = col + 1
        else:
            raise "Illegal Move"

        # Create a copy of the current eightPuzzle
        newPuzzle = EightPuzzleState([0 for _ in range(size * size)])
        newPuzzle.cells = [values[:] for values in self.cells]
        # And update it to reflect the move
        newPuzzle.cells[row][col] = self.cells[newrow][newcol]
        newPuzzle.cells[newrow][newcol] = self.cells[row][col]
        newPuzzle.blankLocation = newrow, newcol

        return newPuzzle

    # Utilities for comparison and display
    def __eq__(self, other):
        """
            Overloads '==' such that two eightPuzzles with the same configuration
          are equal.

          >>> EightPuzzleState([0, 1, 2, 3, 4, 5, 6, 7, 8]) == \
              EightPuzzleState([1, 0, 2, 3, 4, 5, 6, 7, 8]).result('left')
          True
        """
        for row in range(size):
            if self.cells[row] != other.cells[row]:
                return False
        return True

    def __hash__(self):
        return hash(str(self.cells))

    def __getAsciiString(self):
        """
          Returns a display string for the maze
```

```python
        """
        lines = []
        horizontalLine = ('-' * (4 * size + 1))
        lines.append(horizontalLine)
        for row in self.cells:
            rowLine = '|'
            for col in row:
                if col == 0:
                    col = ' '
                rowLine = rowLine + ' ' + col.__str__() + ' |'
            lines.append(rowLine)
            lines.append(horizontalLine)
        return '\n'.join(lines)

    def __str__(self):
        return self.__getAsciiString()

class EightPuzzleSearchProblem(search.SearchProblem):
    """
      Implementation of a SearchProblem for the  Eight Puzzle domain

      Each state is represented by an instance of an eightPuzzle.
    """
    def __init__(self,puzzle):
        "Creates a new EightPuzzleSearchProblem which stores search information."
        self.expanded = 0
        self.puzzle = puzzle

    def getStartState(self):
        return self.puzzle

    def isGoalState(self,state):
        return state.isGoal()

    def getSuccessors(self,state):
        """
          Returns list of (successor, action, stepCost) pairs where
          each succesor is either left, right, up, or down
          from the original state and the cost is 1.0 for each
        """
        self.expanded += 1
        succ = []
        for a in state.legalMoves():
            succ.append((state.result(a), a, 1))
        return succ

    def getCostOfActions(self, actions):
        """
         actions: A list of actions to take

        This method returns the total cost of a particular sequence of actions.  The sequence must
        be composed of legal moves
        """
        return len(actions)

EIGHT_PUZZLE_DATA = [[1, 0, 2, 3, 4, 5, 6, 7, 8],
                     [1, 7, 8, 2, 3, 4, 5, 6, 0],
                     [4, 3, 2, 7, 0, 5, 1, 6, 8],
```

```python
                        [5, 1, 3, 4, 0, 2, 6, 7, 8],
                        [1, 2, 5, 7, 6, 8, 0, 4, 3],
                        [0, 3, 1, 6, 8, 2, 7, 5, 4]]

def loadEightPuzzle(puzzleNumber):
    """
      puzzleNumber: The number of the eight puzzle to load.

      Returns an eight puzzle object generated from one of the
      provided puzzles in EIGHT_PUZZLE_DATA.

      puzzleNumber can range from 0 to 5.

      >>> print loadEightPuzzle(0)
      -------------
      | 1 |   | 2 |
      -------------
      | 3 | 4 | 5 |
      -------------
      | 6 | 7 | 8 |
      -------------
    """
    return EightPuzzleState(EIGHT_PUZZLE_DATA[puzzleNumber])

def createRandomEightPuzzle(moves=100):
    """
      moves: number of random moves to apply

      Creates a random eight puzzle by applying
      a series of 'moves' random moves to a solved
      puzzle.
    """
    puzzle = EightPuzzleState([i for i in range(size * size)])
    for i in range(moves):
        # Execute a random legal move
        puzzle = puzzle.result(random.sample(puzzle.legalMoves(), 1)[0])
    return puzzle

def nullHeuristic(state, problem=None):
    """
    A heuristic function estimates the cost from the current state to the
   nearest
    goal in the provided SearchProblem.  This heuristic is trivial.
    """

    return 0

def tileMisplacedHeuristic(state, problem = None):
    count = 0
    current = 0
    for row in range(size):
        for col in range(size):
            if current != state.cells[row][col]:
                count += 1
            current += 1
    return count

def manhattanDistance(position1, position2):
    xy1 = position1
    xy2 = position2
```

```
    return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])

def manhattanDistanceToCorrectPositionHeuristic(state, problem = None):
    coordinates = [(x, y) for x in range(size) for y in range(size)]

    total_distance = 0

    current = 0
    for row in range(size):
        for col in range(size):
            if current != state.cells[row][col]:
                total_distance += manhattanDistance(coordinates[state.cells[row][col]], (row, col))
            current += 1

    return total_distance

def outOfColumnRowHeuristic(state, problem = None):
    coordinates = [(x, y) for x in range(size) for y in range(size)]

    outOfRow = 0
    outOfColumn = 0

    current = 0
    for row in range(size):
        for col in range(size):
            if current != state.cells[row][col]:
                if row != coordinates[state.cells[row][col]][0]:
                    outOfRow += 1
                if col != coordinates[state.cells[row][col]][1]:
                    outOfColumn += 1

    return outOfRow + outOfColumn

def euclideanDistanceToCorrectPositionHeuristic(state, problem = None):
    total_distance = 0
    current = 0
    for row in range (size):
        for col in range (size):
            if current != state.cells[row][col] :
                goal_row, goal_col = divmod(state.cells[row][col], 3)
                total_distance += ((row - goal_row) ** 2 + (col - goal_col) ** 2) ** 0.5
            current += 1
    return total_distance

def swapHeuristic(state, problem):
    coordinates = [(x, y) for x in range(size) for y in range(size)]

    total_cost = 0
    cells2 = [row [:] for row in state.cells]

    for row in range (size):
        for col in range (size):
            x,y = coordinates[cells2[row][col]]
            if (x != row) or (y != col):
                aux = cells2[row][col]
                cells2[row][col] = cells2[x][y]
                cells2[x][y] = aux
                total_cost += 1
```

```
    return total_cost

def StalinSort():
    puzzle = createRandomEightPuzzle(100)
    print('A random puzzle:')
    print(puzzle)

    problem = EightPuzzleSearchProblem(puzzle)
    current = 0
    for row in range (size):
        for col in range (size):
            if current != puzzle.cells[row][col]:
                puzzle.cells[row][col] = 0
            else:
                puzzle.cells[row][col] = current
            current += 1

    print("\nStalin's hand falls upon this puzzle and it now becomes:")
    print(puzzle)

def readCommand(args):
    parser = OptionParser()
    parser.add_option("--heuristic", dest='heuristic', action="store", type="string", default="null")
    parser.add_option("-s", "--size", dest="size", action="store", type="int", default=3)
    parser.add_option("-f", "--function", dest="function", action="store", type="string", default="astar")

    options, arg = parser.parse_args(args)

    if len(arg) != 0:
        print("Commands not understood")
        return

    heuristic = None

    if options.heuristic == "manhattan":
        heuristic = manhattanDistanceToCorrectPositionHeuristic
    elif options.heuristic == "euclidian":
        heuristic = euclideanDistanceToCorrectPositionHeuristic
    elif options.heuristic == "tileMisplaced":
        heuristic = tileMisplacedHeuristic
    elif options.heuristic == "outOfColumnRow":
        heuristic = outOfColumnRowHeuristic
    elif options.heuristic == "swap":
        heuristic = swapHeuristic
    elif options.heuristic == "null":
        heuristic = nullHeuristic
    else:
        print("No such heuristic found")
        return

    func = None
    if options.function == "StalinSort":
        StalinSort()
        return
    elif options.function == "wastar":
        func = search.weightedAStarSearch
    elif options.function == "astar":
```

```
        func = search.aStarSearch
    else:
        print("That function doesn't exist")
        return

    global size
    size = options.size

    runGame(func, heuristic)

def runGame(func, heuristic):
    puzzle = createRandomEightPuzzle(30)
    #puzzle = EightPuzzleState([7, 2, 4, 5, 0, 6, 8, 3, 1])
    # puzzle = EightPuzzleState([8, 7, 6, 5, 4, 3, 2, 1, 0])
    # puzzle = EightPuzzleState([5, 0, 8, 1, 4, 2, 3, 6, 7])
    print(puzzle)
    problem = EightPuzzleSearchProblem(puzzle)
    start = time.time()
    path = func(problem=problem, heuristic=heuristic)
    end = time.time()
    print(end - start)
    print("Expanded nodes: %d" % problem.expanded)
    print('A* found a path of %d moves: %s' % (len(path), str(path)))
    curr = puzzle
    i = 1
    for a in path:
        curr = curr.result(a)
        print('After %d move%s: %s' % (i, ("", "s")[i>1], a))
        print(curr)

        raw_input("Press return for the next state...")   # wait for key
   stroke
        i += 1

if __name__ == '__main__':
    args = readCommand(sys.argv[1:])
```

Listing A.7: Eight Puzzle Problem Class

```
  class Directions:
    NORTH = 'North'
    SOUTH = 'South'
    EAST = 'East'
    WEST = 'West'
    STOP = 'Stop'
    NORTHEAST = 'NorthEast'
    NORTHWEST = 'NorthWest'
    SOUTHEAST = 'SouthEast'
    SOUTHWEST = 'SouthWest'

    LEFT =       {NORTH: WEST,
                   SOUTH: EAST,
                   EAST:  NORTH,
                   WEST:  SOUTH,
                   STOP:  STOP}

    RIGHT =      dict([(y,x) for x, y in LEFT.items()])

    REVERSE = {NORTH: SOUTH,
```

```
                SOUTH: NORTH,
                EAST: WEST,
                WEST: EAST,
                STOP: STOP}

class Actions:
    """
    A collection of static methods for manipulating move actions.
    """
    # Directions
    _directions = {Directions.NORTH: (0, 1),
                   Directions.SOUTH: (0, -1),
                   Directions.EAST:  (1, 0),
                   Directions.WEST:  (-1, 0),
                   Directions.STOP:  (0, 0),
                   Directions.NORTHEAST: (1, 1),
                   Directions.NORTHWEST: (-1, 1),
                   Directions.SOUTHEAST: (1, -1),
                   Directions.SOUTHWEST: (-1, -1)}

    _directionsAsList = _directions.items()

    ............
```

Listing A.8: game.py file modifications