Department of Computer Science
Technical University of Cluj-Napoca

Artificial Intelligence

Laboratory activity

Name: Știrb Călin-Alexandru
Name: Horst Jaden
Group: 30432
Email: stirbalex@yahoo.com
Email: jadenhorst@gmail.com

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

Contents

Table 1: Lab scheduling

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

Lab organisation.

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

Chapter 1

A1: Search

1.1 Introduction:

- **This chapter covers the search algorithms used in a game of Pacman in order to solve a list of problems, and determine which algorithm is better.**
- **One such problem is finding the fastest route to a single Pacman ”food” object placed at predefined position in the maze**
- **Another problem is finding an optimal path for Pacman to eat the food objects placed at each of the corners of the maze**
- **Finally we have the problem where we need Pacman to eat all of the multiple food objects scattered inside of the maze**

1.2 Search Algorithms:

The following search algorithms have been implemented in order to solve the problems:

1.2.1 Depth First Search (DFS)

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch of a graph before backtracking. This algorithm uses a stack to visit nodes.

```
def depthFirstSearch(problem):
  """
  Search the deepest nodes in the search tree first [p 85].

  Your search algorithm needs to return a list of actions that reaches
  the goal.  Make sure to implement a graph search algorithm [Fig. 3.7].

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

```
print("Start:", problem.getStartState())
print("Is the start a goal?",
   problem.isGoalState(problem.getStartState()))
print("Start's successors:",
   problem.getSuccessors(problem.getStartState()))

queue = util.Stack()
visitedNodes = []
start = problem.getStartState()
firstNode = (start,[])

queue.push(firstNode)

while not queue.isEmpty():
  state,actions = queue.pop()
  if state not in visitedNodes:
    visitedNodes.append(state)
    if problem.isGoalState(state):
      return actions
    else:
      successors = problem.getSuccessors(state)
      for state,action,cost in successors:
        newAction = actions + [action]
        newNode = (state,newAction)
        queue.push(newNode)
return []
```

1.2.2 Breadth First Search (BFS)

Breadth-First Search (BFS) is a graph traversal algorithm that explores all neighbors of a node before moving on to their child nodes. It searches level by level, making it suitable for finding the shortest path in unweighted graphs. This algorithm uses a queue to visit nodes.

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

    start = [problem.getStartState(), ""]
    open_ds.push([start])

    visited_state = [start[0]]

    while not open_ds.isEmpty():
        node = open_ds.pop()
        end = node[-1]

        if problem.isGoalState(end[0]):
            return [state[1] for state in node[1:]]
```

```
        successors = problem.getSuccessors(end[0])
        for succ in successors:
            if succ[0] not in visited_state:
                visited_state.append(succ[0])
                new_node = copy.deepcopy(node)
                new_node.append(succ)
                open_ds.push(new_node)

    return []
```

1.2.3 Uniform-Cost Search (UCS)

Uniform-Cost Search (UCS) is a graph traversal algorithm used to find the path with the lowest cost in a weighted graph. It explores nodes in increasing order of path cost, ensuring that it always considers the least costly path first. The algorithm uses a priority queue to visit nodes

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    open_ds = util.PriorityQueue()

    start = [problem.getStartState(),""]
    open_ds.push([start], 0)
    visited_state = []

    while not open_ds.isEmpty():
        node = open_ds.pop()
        end = node[-1]

        if problem.isGoalState(end[0]):
            return [state[1] for state in node[1:]]

        if end[0] not in visited_state:
            visited_state.append(end[0])
            successors = problem.getSuccessors(end[0])
            for succ in successors:
                new_node = node + [succ]
                total_cost = problem.getCostOfActions([state[1]
                   for state in new_node[1:]])
                open_ds.push(new_node, total_cost)

    return []
```

1.2.4 A star (A*)

A* Search is a graph traversal algorithm that finds the shortest path from a start node to a goal node in a weighted graph. It combines the benefits of Dijkstra's algorithm and heuristic approaches. A* uses a priority queue to explore nodes, considering both the cost to reach a node and an estimated cost to reach the goal.

```
def aStarSearch(problem, heuristic=nullHeuristic):
    "*** YOUR CODE HERE ***"
    print "Start:", problem.getStartState()
    print "Is the start a goal?", problem.isGoalState(problem.getStartState())
    print "Start's successors:", problem.getSuccessors(problem.getStartState())

    current = problem.getStartState()
    priority_queue = util.PriorityQueue()
    visited = []
    solution = []

    if problem.isGoalState(current):
        print "Solution: ", solution
        return solution

    while not problem.isGoalState(current):
        if current not in visited:
            visited.append(current)
            successors = problem.getSuccessors(current)

            for nextState, action, step_cost in successors:
                temp = solution + [action]
                que_pos = problem.getCostOfActions(temp) + heuristic(nextState,problem)
                priority_queue.push((temp,nextState),que_pos)
        aux = priority_queue.pop()
        solution = aux[0]
        current = aux[1]

    return solution
    util.raiseNotDefined()
```

1.2.5 RandomSearch algorithm

This algorithm, as the name suggests, picks a successors at random from the successor list and traverses them until the goal state of the problem has been reached

```
def RandomSearch(problem):
    print "Start:", problem.getStartState()
    print "Is the start a goal?", problem.isGoalState(problem.getStartState())
    print "Start's successors:", problem.getSuccessors(problem.getStartState())

    solution =[]
    current = problem.getStartState()

    while not problem.isGoalState(current):
```

```
        successors = problem.getSuccessors(current)
        if not successors:
            return []
        random_successor = random.choice(successors)
        next_state, action, _ = random_successor
        solution.append(action)
        current = next_state
    return solution
```

1.2.6 GreedyBestFirstSearch algorithm

Greedy Best-First Search is an informed search algorithm that aims to find a path from the start state to a goal state by prioritizing states based on a heuristic evaluation of their potential to reach the goal. This algorithm expands states with the lowest heuristic values, as it assumes that the states closer to the goal are more promising.
The algorithm is efficient for finding solutions in scenarios where a good heuristic estimate of the distance to the goal is available. However, it may not guarantee the optimal solution and can get stuck in local optima when the heuristic is not admissible.

```
    def greedyBestFirstSearch(problem, heuristic=nullHeuristic):
    print "Start:", problem.getStartState()
    print "Is the start a goal?",
       problem.isGoalState(problem.getStartState())
    print "Start's successors:",
       problem.getSuccessors(problem.getStartState())

    start_state = problem.getStartState()
    open_list = util.PriorityQueue()
    start_heuristic = heuristic(start_state, problem)
    open_list.push((start_state, [], start_heuristic),
       start_heuristic)
    visited = set()

    while not open_list.isEmpty():
        current_state, actions, h_val = open_list.pop()
        if problem.isGoalState(current_state):
            return actions
        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for next_state,action,_ in successors:
                if next_state not in visited:
                    h_next = heuristic(next_state,problem)
                    open_list.push((next_state,actions
                       +[action], h_val), h_val)
    return []
```

1.3 Problem Introductions

In this Section we delve into the logics and hueristics employed by Pacman to navigate the game world's mazes effectively. The main focus of this chapter is to present the main functions implemented in order to achieve optimal results for the Four Corners problem as well as the Food Search problem

1.3.1 Four Corners Problem

One of the key challenges in the Pacman world is to visit all four corners of a given maze. Achieving this involves the use of algorithms that can determine Pacman's path. In this chapter we will explore the logic behind the "Four Corners Problem"

1.3.2 Algorithmic insight

We delve into the algorithms that drive Pacman's journey through the maze. The algorithms we examine include Depth-First Search (DFS), Breadth-First Search (BFS), Uniform Cost Search (UCS) and A Star Search (A*). These algorithms play a pivotal role in determining Pacman's path, the order in which it explores the maze, and ultimately achieving the goal of visiting all four corners.

1.3.3 Food Search Problem

This problem almost functions like the original game of pacman (minus the ghosts), in which the search agent has to find the best path that collects all of the food objects inside of a maze.

1.4 The four corners problem

This section will go into detail on how the 4 corners problem was solved

1.4.1 The getStartState function

- This function returns the starting state for the Pacman game
- A state is represented as a tuple which consists of the Pacman's current coordinates
- The coordinates are represented as a tuple of the form **(x,y)**
- The second object of the tuple, '()' is used to represent the corners visited corners
- Essentially the function returns the starting coordinates of Pacman and an empty visited corners list

```
def getStartState(self):
  return (self.startingPosition, ())
```

1.4.2 The isGoalState function

- This function checks if the given state is a goal state for the corner's problem
- The state parameter is expected to be the same as the returned value of **getStartState**, meaning a touple of the form **((x,y), [visitedCorners])**
- The goal of the Pacman game is to visit all 4 corners of the maze where food has been placed
- To determine if the state is a goal, the function compares the set of visited corners (set(state[1])) with the set of all corners in the maze (set(self.corners))
- If the sets are equal then the function returns 'True' otherwise it returns 'False'

```
def isGoalState(self, state):
    return set(state[1]) == set(self.corners)
```

1.4.3 The getSuccsessors function

- This function is repsonsible for generating a list of succsessors states based on the Pacman's current state in the game
- The current state is of the same form as it is for the isGoalState function
- The function iterates over each possible action, meaning: **Nort, South, East and West**
- For each action it calculates the next position 'sucPos' based on the current state and the action taken
- It checks whether the position after taking the action hits a wall by examining the 'self.walls' atribute; if the move is hits a wall it is considered an **illegal action**
- If the next position is a valid move, the successor state is generated
- if the found position is a not yet visited corner, it adds the corner into the 'corners' list by updating the state
- The function keeps track of how many search nodes it has expanded for analysis purposes
- The function returns the list of successors states 'successors'

```
def getSuccessors(self, state):
    successors = []
    currentPos,corners = state
    for action in [Directions.NORTH, Directions.SOUTH,
       Directions.EAST, Directions.WEST]:
        x, y = currentPos
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:#if it doesnt hit wall
            sucPos = (nextx, nexty)
            sucCorner = list(corners)
            if sucPos in self.corners and sucPos not in corners:
```

```
                sucCorner.append(sucPos)
            newSuccession = ((sucPos, tuple(sucCorner)), action, 1)
            successors.append(newSuccession)
    self._expanded += 1 # DO NOT CHANGE
    return successors
```

1.4.4 The cornerHeuristic function

- This function is used to calculate a **heuristic value** for the four corners problem of the Pacman game
- The function takes two parameters: 'state' (the current state, of the same form as the previous functions) and 'problem' (the problem instance, which contains the information about the maze)
- It calculates the set of unvisited corners by subtracting the set of visited corners 'vCorners' from the set of all corners 'problem.corners'
- Within the loop the function calculates the **Manhattan distances** between the current position and all of the unvisited corners
- The index of the closest corner is determined using the **distances.index(min(distances))** operation
- After identifying the closest corner, we use it to calculate the Manhattan distance between it and the current position
- This distance is added to the 'heuristic' variable, which accumulates the estimated cost of reaching the closest corner
- The current position is updated to be the closest corner for the next iteration of the loop
- In summary, the **cornersHeuristic** function calculates the heuristic value based on the Manhattan distances of the unvisited corners

```python
def cornersHeuristic(state, problem):
    curPos, vCorners = state
    unvisited_corners = set(problem.corners) - set(vCorners)

    heuristic = 0

    while unvisited_corners:
        distances = [util.manhattanDistance(curPos, corner) for corner in unvisited_corners]
        closest_index = distances.index(min(distances))
        closest_corner = unvisited_corners[closest_index]
        heuristic += util.manhattanDistance(curPos, closest_corner)
        curPos = closest_corner
        unvisited_corners.remove(closest_corner)

    return heuristic
```

1.5 The Food Search Problem

In this section we will go into detail on how the heuristic function for this problem was implemented

1.5.1 The foodHeuristic function

- The 'foodHeuristic' function calculates a heuristic estimate for the cost to collect all remaining food objects in the Pacman game
- It takes 2 parameters: the current game state and the problem instance
- the function defines 2 helper methods: 'find' and 'union' which are used for finding the connected components and performing union operations for MST
- The algorithm constructs a graph of food object distances by calculating the **Manhattan distance** between pairs of food dots
- It then uses Kruskal's algorithm to find the MST of the graph. This MST represents the minimum total distance needed to visit all unvisited food objects
- The heuristic is determined by adding the distance to the nearest unvisited food dot to the total distance represented by the MST
- In summary the foodHeuristic algorithm combines the distance to the nearest unvisited food dot with the minimum total distance represented by the MST of food dot distances

```
def foodHeuristic(state, problem):
    def find(parent, i):
        if parent[i] is None:
            return i
        return find(parent, parent[i])

    def union(parent, rank, x, y):
        xroot = find(parent, x)
        yroot = find(parent, y)

        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    position, foodGrid = state
    unvisited_foods = foodGrid.asList()

    if not unvisited_foods:
        return 0

    graph = []
```

```
    for i in range(len(unvisited_foods)):
        for j in range(i + 1, len(unvisited_foods)):
          graphList = [unvisited_foods[i], unvisited_foods[j],
             util.manhattanDistance(unvisited_foods[i],
             unvisited_foods[j])]
            graph.append(graphList)

    mst_length = 0
    i = 0
    e = 0

    graph = sorted(graph, key=lambda x: x[2])

    parent = {}
    rank = {}

    for food in unvisited_foods:
        parent[food] = None
        rank[food] = 0

    while e < len(unvisited_foods) - 1:
        u, v, w = graph[i]
        i += 1
        x = find(parent, u)
        y = find(parent, v)

        if x != y:
            e += 1
            mst_length += w
            union(parent, rank, x, y)

    closest_food = min([util.manhattanDistance(position, food)
       for food in unvisited_foods])

    return closest_food + mst_length
```

Chapter 2

A2: Logics

2.1 Theoretical approach

First Order Logic (FOL): First Order Logic is a fundamental mathematical language used to represent statements and reason about relationships within a domain. It employs quantifiers like "forall" and "exists" to describe properties and relationships between objects, utilizing variables, predicates, functions, and constants to express complex ideas in a formal and precise manner. FOL serves as a basis for various automated reasoning systems and theorem provers.

Mace4: Mace4 is a powerful tool in the field of automated reasoning and model-building specifically designed for finite first-order models. It operates by searching for counterexamples to determine the satisfiability of a given set of first-order logic formulas. Mace4 explores possible interpretations of the formulas within a specified domain and can find models that satisfy the logical constraints, aiding in detecting errors or inconsistencies within theories or systems.

Prover9: Prover9 stands as a widely-used automated theorem prover for first-order logic. Employing resolution-based theorem proving techniques, it endeavors to establish the validity of conjectures or theorems within the specified logical framework. Prover9 utilizes inference rules and strategies to derive new statements from a given set of axioms, aiming to either find a proof for a conjecture or demonstrate its inconsistency if it exists.

2.2 A simple code cracker

CodeCracker.in: The 'CodeCracker.in' file encapsulates a set of first-order logic rules encoding a code-cracking puzzle. It employs finite model generation principles within the domain of arithmetic, specifically restricting values to a domain size of 10 and aiming to find up to 2 models that satisfy the specified logic.

The 'CodeCracker' formulas define rules governing the correct placement of values within a code. These rules encompass conditions where a single value is correctly placed, where no value is placed correctly, where one value is correctly placed but in the wrong position and where two values are correctly placed but in the wrong positions.

2.3 Sudoku

Sudoku App in Python: a Sudoku solving application using the Tkinter library for the GUI. It defines functions to create a graphical interface for entering Sudoku puzzles, along with buttons to fill the grid randomly or with a predetermined sample solution. Users can click on individual cells to enter numbers manually. The core function, `solve_sudoku()`, encodes the Sudoku puzzle as first-order logic rules in a format compatible with Mace4, a tool for finite model finding in first-order logic.

The script's structure involves defining a sample Sudoku solution and functions to fill the grid with random numbers or the sample solution. The GUI is set up using Tkinter's Entry widgets to create a 9x9 grid for input. Additionally, buttons for filling with random or sample solutions, and a "Mace4 proof" button, which triggers the `solve_sudoku()` function to attempt solving the puzzle using Mace4.

The `solve_sudoku()` function generates first-order logic rules based on the entries in the grid and writes them to a file named `sudoku.in`. It then calls Mace4 using a subprocess to check the solvability of the puzzle. If solvable, it confirms this; otherwise, it reports that the Sudoku puzzle is unsolvable.

The GUI is structured with buttons for various functionalities, allowing users to interactively solve Sudoku puzzles or generate new ones.

2.4 A game of Tic-Tac-Toe

Tic Tac Toe: This Python script implements a graphical Tic Tac Toe game using the Tkinter library. It creates a 3x3 grid of buttons where players take turns placing their marks ('X' or 'O') by clicking on the buttons. The game logic is managed by a `TicTacToe` class that tracks the current player, the state of the board, and button clicks.

Upon a player's move, the game checks for a winner by examining rows, columns, and diagonals using the `check_winner()` method, ensuring three marks of the same type in a row. Additionally, the `check_winner_with_mace4()` method attempts to use the Mace4 tool, encoding the board state as a set of first-order logic equations to check for winning configurations.

The script includes functionality to reset the game once a player wins or when the board fills up with no winner. It utilizes subprocesses to interact with Mace4, attempting to find solutions that validate the winning conditions on the board.

Overall, this app offers an interactive Tic Tac Toe game interface and demonstrates an attempt to employ Mace4 for logical verification of winning states within the game grid.

Chapter 3

A3: Planning

Bibliography

The 2nd AI course slides: 02 IA search
The examples given on moodle
Documentation template provided on moodle
https://www.github.com/
https://stackoverflow.com/

Appendix A

Your original code

A.1 Chapter A1:

A.1.1 search.py

```
def depthFirstSearch(problem):
  print("Start:", problem.getStartState())
  print("Is the start a goal?",
     problem.isGoalState(problem.getStartState()))
  print("Start's successors:",
     problem.getSuccessors(problem.getStartState()))

  queue = util.Stack()
  visitedNodes = []
  start = problem.getStartState()
  firstNode = (start,[])

  queue.push(firstNode)

  while not queue.isEmpty():
    state,actions = queue.pop()
    if state not in visitedNodes:
      visitedNodes.append(state)
      if problem.isGoalState(state):
        return actions
      else:
        successors = problem.getSuccessors(state)
        for state,action,cost in successors:
          newAction = actions + [action]
          newNode = (state,newAction)
          queue.push(newNode)
  return []

def breadthFirstSearch(problem):
    open_ds = util.Queue()

    start = [problem.getStartState(), ""]
    open_ds.push([start])
```

```python

    visited_state = [start[0]]

    while not open_ds.isEmpty():
        node = open_ds.pop()
        end = node[-1]

        if problem.isGoalState(end[0]):
            return [state[1] for state in node[1:]]

        successors = problem.getSuccessors(end[0])
        for succ in successors:
            if succ[0] not in visited_state:
                visited_state.append(succ[0])
                new_node = copy.deepcopy(node)
                new_node.append(succ)
                open_ds.push(new_node)

    return []

def uniformCostSearch(problem):
    open_ds = util.PriorityQueue()

    start = [problem.getStartState(),""]
    open_ds.push([start], 0)
    visited_state = []

    while not open_ds.isEmpty():
        node = open_ds.pop()
        end = node[-1]

        if problem.isGoalState(end[0]):
            return [state[1] for state in node[1:]]

        if end[0] not in visited_state:
            visited_state.append(end[0])
            successors = problem.getSuccessors(end[0])
            for succ in successors:
                new_node = node + [succ]
                total_cost = problem.getCostOfActions([state[1]
                   for state in new_node[1:]])
                open_ds.push(new_node, total_cost)

    return []

def aStarSearch(problem, heuristic=nullHeuristic):
    open_ds = util.PriorityQueue()

    start_state = problem.getStartState()
```

```
    start = (start_state, "", (0, heuristic(start_state, problem)))
    open_ds.push([start], start[2])

    visited_state = {start_state[0]: sum(start[2])}

    while not open_ds.isEmpty():
        node = open_ds.pop()

        if not open_ds.isEmpty():
            next_node = open_ds.pop()
            temp = [node, next_node]

            while not open_ds.isEmpty() and sum(node[-1][2]) == sum(next_node[-1][2]):
                next_node = open_ds.pop()
                temp.append(next_node)

            tie_nodes = temp if sum(node[-1][2]) == sum(next_node[-1][2]) else temp[:-1]

            for n in tie_nodes:
                if node[-1][2][0] < n[-1][2][0]:
                    node = n

            for n in temp:
                if node != n:
                    open_ds.push(n, sum(n[-1][2]))

        end = node[-1]
        if end[0] not in visited_state or sum(end[2]) <= visited_state[end[0]]:
            if problem.isGoalState(end[0]):
                return [state[1] for state in node[1:]]

            successors = problem.getSuccessors(end[0])
            for succ in successors:
                gn_succ = end[2][0] + succ[2]
                hn_succ = heuristic(succ[0], problem)
                fn_succ = gn_succ + hn_succ
                if succ[0] not in visited_state or fn_succ < visited_state[succ[0]]:
                    visited_state[succ[0]] = fn_succ
                    new_node = copy.deepcopy(node)
                    new_succ = (succ[0], succ[1], (gn_succ, hn_succ))
                    new_node.append(new_succ)
                    open_ds.push(new_node, fn_succ)
    return []
```

```
def RandomSearch(problem):
    print "Start:", problem.getStartState()
    print "Is the start a goal?", problem.isGoalState(problem.getStartState())
    print "Start's successors:", problem.getSuccessors(problem.getStartState())

    solution =[]
    current = problem.getStartState()

    while not problem.isGoalState(current):
        successors = problem.getSuccessors(current)
        if not successors:
            return []
        random_successor = random.choice(successors)
        next_state, action, _ = random_successor
        solution.append(action)
        current = next_state
    return solution

def greedyBestFirstSearch(problem, heuristic=nullHeuristic):
    print "Start:", problem.getStartState()
    print "Is the start a goal?", problem.isGoalState(problem.getStartState())
    print "Start's successors:", problem.getSuccessors(problem.getStartState())

    start_state = problem.getStartState()
    open_list = util.PriorityQueue()
    start_heuristic = heuristic(start_state, problem)
    open_list.push((start_state, [], start_heuristic), start_heuristic)
    visited = set()

    while not open_list.isEmpty():
        current_state, actions, h_val = open_list.pop()
        if problem.isGoalState(current_state):
            return actions
        if current_state not in visited:
            visited.add(current_state)
            successors = problem.getSuccessors(current_state)
            for next_state,action,_ in successors:
                if next_state not in visited:
                    h_next = heuristic(next_state,problem)
                    open_list.push((next_state,actions+[action], h_val), h_val)
    return []
```

A.1.2 searchAgents.py

```

def getStartState(self):
    return (self.startingPosition, ())

def isGoalState(self, state):
    return set(state[1]) == set(self.corners)

def getSuccessors(self, state):
        successors = []
        currentPos,corners = state
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:
            x, y = currentPos
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:#if it doesnt hit wall
                sucPos = (nextx, nexty)
                sucCorner = list(corners)
                if sucPos in self.corners and sucPos not in corners:
                    sucCorner.append(sucPos)
                newSuccession = ((sucPos, tuple(sucCorner)), action, 1)
                successors.append(newSuccession)
        self._expanded += 1 # DO NOT CHANGE
        return successors

def cornersHeuristic(state, problem):
    curPos, vCorners = state
    unvisited_corners = set(problem.corners) - set(vCorners)

    heuristic = 0

    while unvisited_corners:
        distances = [util.manhattanDistance(curPos, corner) for corner in unvisited_corners]
        closest_index = distances.index(min(distances))
        closest_corner = unvisited_corners[closest_index]
        heuristic += util.manhattanDistance(curPos, closest_corner)
        curPos = closest_corner
        unvisited_corners.remove(closest_corner)

    return heuristic

def foodHeuristic(state, problem):
    def find(parent, i):
```

```
        if parent[i] is None:
            return i
        return find(parent, parent[i])

    def union(parent, rank, x, y):
        xroot = find(parent, x)
        yroot = find(parent, y)

        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1

    position, foodGrid = state
    unvisited_foods = foodGrid.asList()

    if not unvisited_foods:
        return 0

    graph = []
    for i in range(len(unvisited_foods)):
        for j in range(i + 1, len(unvisited_foods)):
          graphList = [unvisited_foods[i], unvisited_foods[j],
             util.manhattanDistance(unvisited_foods[i],
             unvisited_foods[j])]
            graph.append(graphList)

    mst_length = 0
    i = 0
    e = 0

    graph = sorted(graph, key=lambda x: x[2])

    parent = {}
    rank = {}

    for food in unvisited_foods:
        parent[food] = None
        rank[food] = 0

    while e < len(unvisited_foods) - 1:
        u, v, w = graph[i]
        i += 1
        x = find(parent, u)
        y = find(parent, v)

        if x != y:
```

```
            e += 1
            mst_length += w
            union(parent, rank, x, y)

    closest_food = min([util.manhattanDistance(position, food)
       for food in unvisited_foods])

    return closest_food + mst_length
```

A.2 Chapter A2

A.2.1 CodeCracker

Listing A.1: Mace4 Code for CodeCracker Puzzle

```
set(arithmetic).
assign(domain_size,10).
assign(max_models,2).

formulas(CodeCracker).
%X = 1.
%Y = 6.
%Z = 4.
one_value_correctly_placed(x, y, z) -> (x = X & y != Y & z != Z)
   | (x != X & y = Y & z != Z) | (x != X & y != Y & z != z).
nothing_correct(x,y,z) -> (x != X & y != Y & z != Z) | (x != X &
   y != Z & z != Y) | (x != Y & y != Y & z != X) | (x != Y & y
   != X & z != Z).
one_value_correct_wrongly_placed(x,y,z) -> (x = Y & y != Y & z
   != Z) | (x = Z & y != Y & z != Z) | (x != X & y = X & z != Z)
   | (x != X & y = Z & z != Z) | (x != X & y != Y & z = X) | (x
   != X & y != Y & z = Y).
two_values_correct_wrongly_placed(x,y,z) -> (x = Z & y != Y & z
   = X) | (x = Y & y != Y & z = X) | (x = Z & y != Y & z = Y) |
   (x != X & y = Z & z = Y) | (x != X & y = X & z = Y) | (x != X
   & y = Z & z = X).
end_of_list.

formulas(puzzle1).
one_value_correctly_placed(1,7,5).
nothing_correct(9,7,2).
one_value_correct_wrongly_placed(9,4,3).
two_values_correct_wrongly_placed(7,4,6).
end_of_list.
```

A.2.2 Sudoku

Listing A.2: Python Sudoku Solver

```python
import random
import tkinter as tk
import subprocess

def solve_sudoku(grid_entries):
    sudoku_logic = [
        "assign(domain_size, 9).",
        "assign(max_seconds, 2).",
        "formulas(sudoku_rules).",
        "all x all y1 all y2 (f(x, y1) = f(x, y2) -> y1 = y2).",
        "all x1 all x2 all y (f(x1, y) = f(x2, y) -> x1 = x2).",
        "all x same_interval(x,x).",
        "all x all y (same_interval(x,y) -> same_interval(y,x)).",
        "all x all y all z (same_interval(x,y) & same_interval(y,z) -> same_interval(x,z)).",
        "same_interval(0,1). same_interval(1,2).",
        "same_interval(3,4). same_interval(4,5).",
        "same_interval(6,7). same_interval(7,8).",
        "-same_interval(0,3). -same_interval(3,6). -same_interval(0,6).",
        "all x1 all y1 all x2 all y2",
        "  (same_interval(x1,x2) & same_interval(y1,y2) & f(x1, y1) = f(x2, y2)",
        "  -> x1 = x2 & y1 = y2 ).",
        "all x all z exists y (f(x,y) = z).",
        "all y all z exists x (f(x,y) = z).",
        "end_of_list.",
        "formulas(sample_puzzle)."
    ]

    # Extract numbers from the grid_entries and write to the puzzle rules
    for i, row in enumerate(grid_entries):
        for j, entry in enumerate(row):
            number = entry.get()
            if number != '':
                sudoku_logic.append(f"f({i},{j}) = {number}.")

    sudoku_logic.append("end_of_list.")

    with open("sudoku.in", "w") as file:
        file.write("\n".join(sudoku_logic))

    # Run Mace4 to check solvability
    command = "mace4 -f sudoku.in > sudoku.out"
    try:
        subprocess.run(command, shell=True, check=True)
        print("Sudoku puzzle is solvable!")
    except subprocess.CalledProcessError as e:
```

```
        print("Sudoku puzzle is not solvable!:", e)

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

def fill_sample(entries):
    # Fill the grid with the sample solution
    for i in range(9):
        for j in range(9):
            number = sample_solution[i][j]
            entries[i][j].delete(0, tk.END)  # Clear any existing content
            entries[i][j].insert(tk.END, str(number))

def fill_random(entries):
    # Generate random numbers and fill the grid
    for i in range(9):
        for j in range(9):
            number = random.randint(0, 8)
            entries[i][j].delete(0, tk.END)  # Clear any existing content
            entries[i][j].insert(tk.END, str(number))

def create_gui():
    root = tk.Tk()
    root.title("Sudoku Solver")

    entries = []

    def on_click(row, col):
        print("Clicked cell:", row, col)

    for i in range(9):
        entry_row = []
        for j in range(9):
            cell = tk.Entry(root, width=5)
            cell.grid(row=i, column=j)
            cell.bind('<Button-1>', lambda e, row=i, col=j: on_click(row, col))
            entry_row.append(cell)
        entries.append(entry_row)
```

```python

        solve_button = tk.Button(root, text="Mace4 proof",
            command=lambda: solve_sudoku(entries))
    solve_button.grid(row=10, column=4, columnspan=2)

    fill_button = tk.Button(root, text="Fill Random",
        command=lambda: fill_random(entries))
    fill_button.grid(row=10, column=0, columnspan=2)

    sample_button = tk.Button(root, text="Fill Sample",
        command=lambda: fill_sample(entries))
    sample_button.grid(row=10, column=2, columnspan=2)

    root.mainloop()

if __name__ == "__main__":
    create_gui()
```