Nandhini Manikandan **UIN: 332004080**

Programming Assignment 2

Aero 689 – Introduction to Aerospace Autonomy

1) Check_safe_unvisited function:

```
Comment Code
def check_safe_unvisited(x,y,facts):
    # This function should check if location (x, y) is safe and unvisited by querying the KB.
    ### YOUR CODE HERE
    safe_and_unvisited = False
    for fact in facts:
      fact_dict = dict(fact)
      if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["safe"] == "true" and fact_dict["visited"] == "false":
          safe_and_unvisited = True
    ###
    return safe_and_unvisited
```

2) ask_cell_bio function:

```
Comment Code
def ask_cell_bio(x,y,facts):
    # This function queries the KB to see if there is a biosample in location (x, y).
    ###
    ### YOUR CODE HERE
    bio = False
    for fact in facts:
      fact_dict = dict(fact)
      if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["biosample"] == "true":
        bio = True
    ###
    return bio
```

3) rotate_cw_rule and rotate_ccw_rule:

a. rotate_cw_rule:

```
rotate_cw_rule = """
(defrule rotate_cw_rule
    ; This rule rotates the agent clockwise when a rotate_cw action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "rotate_cw")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $?ah))
    =>
    (bind ?new_o ?o)
    (if (eq ?o "up") then (bind ?new_o "right"))
    (if (eq ?o "right") then (bind ?new_o "down"))
    (if (eq ?o "down") then (bind ?new_o "left"))
    (if (eq ?o "left") then (bind ?new_o "up"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (orientation ?new_o) (time ?t) (action_history (create$ $?ah ?type)))
)
"""
```

b. rotate_ccw_rule:

```
rotate_ccw_rule = """
(defrule rotate_ccw_rule
    ; This rule rotates the agent counterclockwise when a rotate_ccw action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "rotate_ccw")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $?ah))
    =>
    (bind ?new_o ?o)
    (if (eq ?o "up") then (bind ?new_o "left"))
    (if (eq ?o "left") then (bind ?new_o "down"))
    (if (eq ?o "down") then (bind ?new_o "right"))
    (if (eq ?o "right") then (bind ?new_o "up"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (orientation ?new_o) (time ?t) (action_history (create$ $?ah ?type)))
)
"""
```

4) air_quality_rule:

```
air_quality_rule="""
(defrule air_quality_rule
    ; This rule creates an aq_measurement when an air_quality action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "air_quality")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?ah))
    ?hidden-cell <- (hidden-cell(xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (storm_nearby ?sn))
    =>
    (if (eq ?sn "true") then (assert (aq_measurement (xloc ?x) (yloc ?y) (air_quality "high_ppm"))))
    (if (eq ?sn "false") then (assert (aq_measurement (xloc ?x) (yloc ?y) (air_quality "low_ppm"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""
```

5) biosample_meas_rule:

```
biosample_meas_rule="""
(defrule biosample_meas_rule
    ; This rule checks to see if a cell has a biosample based on the bio measurements.
    ; YOUR CODE HERE
    ?meas <- (bio_measurement (xloc ?x) (yloc ?y) (organic ?org))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?bio "unknown")
    (if (eq ?org "true") then (bind ?bio "true"))
    (if (eq ?org "false") then (bind ?bio "false"))
    (modify ?cell (biosample ?bio))
)
"""
```

6) **cell_safe_rule:**

```
safe_cell_rule="""
(defrule safe_cell_rule
    ; This rule checks if the cell has no rock or storm and if so, sets the cell to safe.
    ; YOUR CODE HERE
    ?cell <- (cell (rock ?r) (storm ?s) (safe "unknown"))
    =>
    (bind ?safe "unknown")
    (if (and (eq ?r "false") (eq ?s "false")) then (bind ?safe "true"))
    (modify ?cell (safe ?safe))
)
"""
```

7) **check for unvisited but safe spots to visit:**

```
###
### YOUR CODE HERE
if len(plan) == 0:
    safe = list()
    for cell in cell_facts:
        cell_dict = dict(cell)
        if cell_dict["safe"] == "true" and cell_dict["visited"] == "false" and not (cell_dict["xloc"] == x and cell_dict["yloc"] == y):
            safe.append([cell_dict["xloc"],cell_dict["yloc"]])
    if len(safe) > 0:
        valid_temps = []
        for safe_point in safe:
            safe_points.append(safe_point)
            temp = plan_route(rover_pos, safe_point, safe_points, len(grid[:,0]))
            safe_points.remove(safe_point)
            if temp is not None:
                valid_temps.append(temp)
        if len(valid_temps) > 0:
            temp = random.choice(valid_temps)
            plan.extend(temp)
###
```

8) **number of trials (out of 100) where the rover succeeded in its goal:**
   a. **Doesn't account for sandy squares – BFS Algorithm**

```
Action taken: forward
You win!
Number of victories: 41
Movement cost average: 41.2
```

9) **logic to avoid sandy squares:**
   a. **Implementing a modified path_cost**

```
def path_cost(self, cost, state, action, next_state):
    ###
    ### YOUR CODE HERE (grad students)
    ### Undergrads can comment this function out (or delete the whole class.)
    if action == "sandy_cells":
      return cost + 2
    else:
      return cost + 1
```

b. Implementing A-star or greedy best-first search

```
def astar_search(problem):
  node = Node(problem.initial)
  frontier = PriorityQueue()
  frontier.append(node)

  came_from = {}
  cost_so_far = {}

  came_from[node.state] = None
  cost_so_far[node.state] = 0

  while not frontier.empty():
    current = frontier.pop()

    if problem.goal_test(current.state):
      return current

    for next in current.expand(problem):
      new_cost = cost_so_far[current.state] + problem.path_cost(cost_so_far[current.state],
                                                                current.state,
                                                                current.action,
                                                                next.state)

      if next.state not in cost_so_far or new_cost < cost_so_far[next.state]:
        cost_so_far[next.state] = new_cost
        priority = new_cost + problem.h(next)
        frontier.append(next)
        came_from[next.state] = current
  return None
```

10) Difference in total movement cost when accounting for sand and when not accounting for sand, averaged over 100 trials

	Movement Cost Average	**Total Number of Victories**
Doesn't account for sandy squares – BFS Algorithm	41.2	41
Account for sandy squares – BFS Algorithm	40.68	48
Doesn't account for sandy squares – A* Algorithm	53.7	50
Account for sandy squares – A* Algorithm	46.44	59

Doesn't account for sandy squares – BFS Algorithm:

```
Action taken: forward
You win!
Number of victories: 41
Movement cost average: 41.2
```

Account for sandy squares – BFS Algorithm

```
You win!
Number of victories: 48
Movement cost average: 40.68
```

Doesn't account for sandy squares – A* Algorithm

```
You win!
Number of victories: 50
Movement cost average: 53.7
```

Account for sandy squares – A* Algorithm

```
You win!
Number of victories: 59
Movement cost average: 46.44
```

```
pip install clipspy
```

```
Collecting clipspy
  Downloading clipspy-1.0.2-cp310-cp310-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (891 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 891.6/891.6 kB 10.0 MB/s eta 0:00:00
Requirement already satisfied: cffi>=1.0.0 in /usr/local/lib/python3.10/dist-packages (from clipspy) (1.16.0)
Requirement already satisfied: pycparser in /usr/local/lib/python3.10/dist-packages (from cffi>=1.0.0->clipspy) (2.21)
Installing collected packages: clipspy
Successfully installed clipspy-1.0.2
```

```
import functools
import heapq
import copy
from collections import deque

class RoverPosition:
    def __init__(self, x, y, orientation):
        self.X = x
        self.Y = y
        self.orientation = orientation

    def get_location(self):
        return self.X, self.Y

    def set_location(self, x, y):
        self.X = x
        self.Y = y

    def get_orientation(self):
        return self.orientation

    def set_orientation(self, orientation):
        self.orientation = orientation

    def __eq__(self, other):
        if (other.get_location() == self.get_location() and
                other.get_orientation() == self.get_orientation()):
            return True
        else:
            return False

    def __hash__(self):
        # We use the hash value of the state
        # stored in the node instead of the node
        # object itself to quickly search a node
        # with the same state in a Hash Table
        return hash(self.X)*hash(self.Y)*hash(self.orientation)

class PriorityQueue:
    """A Queue in which the minimum (or maximum) element (as determined by f and
    order) is returned first.
    If order is 'min', the item with minimum f(x) is
    returned first; if order is 'max', then it is the item with maximum f(x).
    Also supports dict-like lookup."""

    def __init__(self, order='min', f=lambda x: x):
        self.heap = []
        if order == 'min':
            self.f = f
        elif order == 'max':  # now item with max f(x)
            self.f = lambda x: -f(x)  # will be popped first
        else:
            raise ValueError("Order must be either 'min' or 'max'.")

    def append(self, item):
        """Insert item at its correct position."""
        heapq.heappush(self.heap, (self.f(item), item))

    def extend(self, items):
        """Insert each item in items at its correct position."""
        for item in items:
            self.append(item)

    def pop(self):
        """Pop and return the item (with min or max f(x) value)
        depending on the order."""
```

```
            if self.heap:
                return heapq.heappop(self.heap)[1]
            else:
                raise Exception('Trying to pop from empty PriorityQueue.')

        def __len__(self):
            """Return current capacity of PriorityQueue."""
            return len(self.heap)

        def __contains__(self, key):
            """Return True if the key is in PriorityQueue."""
            return any([item == key for _, item in self.heap])

        def __getitem__(self, key):
            """Returns the first value associated with key in PriorityQueue.
            Raises KeyError if key is not present."""
            for value, item in self.heap:
                if item == key:
                    return value
            raise KeyError(str(key) + " is not in the priority queue")

        def __delitem__(self, key):
            """Delete the first occurrence of key."""
            try:
                del self.heap[[item == key for _, item in self.heap].index(True)]
            except ValueError:
                raise KeyError(str(key) + " is not in the priority queue")
            heapq.heapify(self.heap)

        def empty(self):

          return len(self.heap) == 0


    class Node:
        """A node in a search tree. Contains a pointer to the parent (the node
        that this is a successor of) and to the actual state for this node. Note
        that if a state is arrived at by two paths, then there are two nodes with
        the same state. Also includes the action that got us to this state, and
        the total path_cost (also known as g) to reach the node. Other functions
        may add an f and h value; see best_first_graph_search and astar_search for
        an explanation of how the f and h values are handled. You will not need to
        subclass this class."""

        def __init__(self, state, parent=None, action=None, path_cost=0):
            """Create a search tree Node, derived from a parent by an action."""
            self.state = state
            self.parent = parent
            self.action = action
            self.path_cost = path_cost
            self.depth = 0
            if parent:
                self.depth = parent.depth + 1

        def __repr__(self):
            return "<Node {}>".format(self.state)

        def __lt__(self, node):
            return self.path_cost < node.path_cost

        def expand(self, problem):
            """List the nodes reachable in one step from this node."""
            return [self.child_node(problem, action)
                    for action in problem.actions(self.state)]

        def child_node(self, problem, action):
            """[Figure 3.10]"""
            x, y = self.state.get_location()
            orientation = self.state.get_orientation()
            rover_pos = RoverPosition(x,y,orientation)
            next_state = problem.result(rover_pos, action)
            next_node = Node(next_state, self, action, problem.path_cost(self.path_cost, self.state, action, next_state))
            return next_node

        def solution(self):
            """Return the sequence of actions to go from the root to this node."""
            return [node.action for node in self.path()[1:]]
```

```
        def path(self):
            """Return a list of nodes forming the path from the root to this node."""
            node, path_back = self, []
            while node:
                path_back.append(node)
                node = node.parent
            return list(reversed(path_back))

        # We want for a queue of nodes in breadth_first_graph_search or
        # astar_search to have no duplicated states, so we treat nodes
        # with the same state as equal. [Problem: this may not be what you
        # want in other contexts.]

        def __eq__(self, other):
            return isinstance(other, Node) and self.state == other.state and self.action == other.action

        def __hash__(self):
            # We use the hash value of the state
            # stored in the node instead of the node
            # object itself to quickly search a node
            # with the same state in a Hash Table
            return hash(self.state)*hash(self.action)

    class Problem:
        """The abstract class for a formal problem. You should subclass
        this and implement the methods actions and result, and possibly
        __init__, goal_test, and path_cost. Then you will create instances
        of your subclass and solve them with the various search functions."""

        def __init__(self, initial, goal=None):
            """The constructor specifies the initial state, and possibly a goal
            state, if there is a unique goal. Your subclass's constructor can add
            other arguments."""
            self.initial = initial
            self.goal = goal

        def actions(self, state):
            """Return the actions that can be executed in the given
            state. The result would typically be a list, but if there are
            many actions, consider yielding them one at a time in an
            iterator, rather than building them all at once."""
            raise NotImplementedError

        def result(self, state, action):
            """Return the state that results from executing the given
            action in the given state. The action must be one of
            self.actions(state)."""
            raise NotImplementedError

        def goal_test(self, state):
            """Return True if the state is a goal. The default method compares the
            state to self.goal or checks for state in self.goal if it is a
            list, as specified in the constructor. Override this method if
            checking against a single self.goal is not enough."""
            if isinstance(self.goal, list):
                return is_in(state, self.goal)
            else:
                return state == self.goal

        def path_cost(self, c, state1, action, state2):
            """Return the cost of a solution path that arrives at state2 from
            state1 via action, assuming cost c to get up to state1. If the problem
            is such that the path doesn't matter, this function will only look at
            state2. If the path does matter, it will consider c and maybe state1
            and action. The default method costs 1 for every step in the path."""
            return c + 1

        def value(self, state):
            """For optimization problems, each state has a value. Hill Climbing
            and related algorithms try to maximize this value."""
            raise NotImplementedError

    class PlanRoute(Problem):
        """ The problem of moving the Hybrid Wumpus Agent from one place to other """

        def __init__(self, initial, goal, allowed, dimrow):
            """ Define goal state and initialize a problem """
```

```
        super().__init__(initial, goal)
        self.dimrow = dimrow
        self.goal = goal
        self.allowed = allowed

    def actions(self, state):
        """ Return the actions that can be executed in the given state.
        The result would be a list, since there are only three possible actions
        in any given state of the environment """

        possible_actions = ['Forward', 'Turnleft', 'Turnright']
        x, y = state.get_location()
        orientation = state.get_orientation()

        # Prevent Bumps
        if x == 0 and orientation == 'left':
            if 'Forward' in possible_actions:
                possible_actions.remove('Forward')
        if y == 0 and orientation == 'up':
            if 'Forward' in possible_actions:
                possible_actions.remove('Forward')
        if x == self.dimrow-1 and orientation == 'right':
            if 'Forward' in possible_actions:
                possible_actions.remove('Forward')
        if y == self.dimrow-1 and orientation == 'down':
            if 'Forward' in possible_actions:
                possible_actions.remove('Forward')
        return possible_actions

    def result(self, state, action):
        """ Given state and action, return a new state that is the result of the action.
        Action is assumed to be a valid action in the state """
        x, y = state.get_location()
        proposed_loc = list()

        # Move Forward
        if action == 'Forward':
            if state.get_orientation() == 'up':
                proposed_loc = [x, y - 1]
            elif state.get_orientation() == 'down':
                proposed_loc = [x, y + 1]
            elif state.get_orientation() == 'left':
                proposed_loc = [x - 1, y]
            elif state.get_orientation() == 'right':
                proposed_loc = [x + 1, y]
            else:
                raise Exception('InvalidOrientation')

        # Rotate counter-clockwise
        elif action == 'Turnleft':
            if state.get_orientation() == 'up':
                state.set_orientation('left')
            elif state.get_orientation() == 'down':
                state.set_orientation('right')
            elif state.get_orientation() == 'left':
                state.set_orientation('down')
            elif state.get_orientation() == 'right':
                state.set_orientation('up')
            else:
                raise Exception('InvalidOrientation')

        # Rotate clockwise
        elif action == 'Turnright':
            if state.get_orientation() == 'up':
                state.set_orientation('right')
            elif state.get_orientation() == 'down':
                state.set_orientation('left')
            elif state.get_orientation() == 'left':
                state.set_orientation('up')
            elif state.get_orientation() == 'right':
                state.set_orientation('down')
            else:
                raise Exception('InvalidOrientation')
        if proposed_loc in self.allowed:
            state.set_location(proposed_loc[0], proposed_loc[1])
        return state
```

```
        def goal_test(self, state):
            """ Given a state, return True if state is a goal state or False, otherwise """

            return state.get_location() == tuple(self.goal)

        def h(self, node):
            """ Return the heuristic value for a given state."""

            # Manhattan Heuristic Function
            x1, y1 = node.state.get_location()
            x2, y2 = self.goal

            return abs(x2 - x1) + abs(y2 - y1)

    class PlanRouteSandy(Problem):
        """ The problem of moving the Hybrid Wumpus Agent from one place to other """

        def __init__(self, initial, goal, allowed, dimrow):
            """ Define goal state and initialize a problem """
            super().__init__(initial, goal)
            self.dimrow = dimrow
            self.goal = goal
            self.allowed = allowed

        def actions(self, state):
            """ Return the actions that can be executed in the given state.
            The result would be a list, since there are only three possible actions
            in any given state of the environment """

            possible_actions = ['Forward', 'Turnleft', 'Turnright']
            x, y = state.get_location()
            orientation = state.get_orientation()

            # Prevent Bumps
            if x == 0 and orientation == 'left':
                if 'Forward' in possible_actions:
                    possible_actions.remove('Forward')
            if y == 0 and orientation == 'up':
                if 'Forward' in possible_actions:
                    possible_actions.remove('Forward')
            if x == self.dimrow-1 and orientation == 'right':
                if 'Forward' in possible_actions:
                    possible_actions.remove('Forward')
            if y == self.dimrow-1 and orientation == 'down':
                if 'Forward' in possible_actions:
                    possible_actions.remove('Forward')
            return possible_actions

        def result(self, state, action):
            """ Given state and action, return a new state that is the result of the action.
            Action is assumed to be a valid action in the state """
            x, y = state.get_location()
            proposed_loc = list()

            # Move Forward
            if action == 'Forward':
                if state.get_orientation() == 'up':
                    proposed_loc = [x, y - 1]
                elif state.get_orientation() == 'down':
                    proposed_loc = [x, y + 1]
                elif state.get_orientation() == 'left':
                    proposed_loc = [x - 1, y]
                elif state.get_orientation() == 'right':
                    proposed_loc = [x + 1, y]
                else:
                    raise Exception('InvalidOrientation')

            # Rotate counter-clockwise
            elif action == 'Turnleft':
                if state.get_orientation() == 'up':
                    state.set_orientation('left')
                elif state.get_orientation() == 'down':
                    state.set_orientation('right')
                elif state.get_orientation() == 'left':
                    state.set_orientation('down')
                elif state.get_orientation() == 'right':
                    state.set_orientation('up')
                else:
```

```
                else:
                    raise Exception('InvalidOrientation')

            # Rotate clockwise
            elif action == 'Turnright':
                if state.get_orientation() == 'up':
                    state.set_orientation('right')
                elif state.get_orientation() == 'down':
                    state.set_orientation('left')
                elif state.get_orientation() == 'left':
                    state.set_orientation('up')
                elif state.get_orientation() == 'right':
                    state.set_orientation('down')
                else:
                    raise Exception('InvalidOrientation')
            if proposed_loc in self.allowed:
                state.set_location(proposed_loc[0], proposed_loc[1])
            return state

        def goal_test(self, state):
            """ Given a state, return True if state is a goal state or False, otherwise """

            return state.get_location() == tuple(self.goal)

        def path_cost(self, cost, state, action, next_state):
            ###
            ### YOUR CODE HERE (grad students)
            ### Undergrads can comment this function out (or delete the whole class.)
            if action == "sandy_cells":
              return cost + 2
            else:
              return cost + 1

        def h(self, node):
            """ Return the heuristic value for a given state."""

            # Manhattan Heuristic Function
            x1, y1 = node.state.get_location()
            x2, y2 = self.goal

            return abs(x2 - x1) + abs(y2 - y1)

    def breadth_first_graph_search(problem):
        """[Figure 3.11]
        Note that this function can be implemented in a
        single line as below:
        return graph_search(problem, FIFOQueue())
        """
        node = Node(problem.initial)
        if problem.goal_test(node.state):
            return node
        frontier = deque([node])
        explored = set()
        while frontier:
            node = frontier.popleft()
            explored.add(node.state)
            for child in node.expand(problem):
                if child.state not in explored and child not in frontier:
                    if problem.goal_test(child.state):
                        return child
                    frontier.append(child)
        return None

    def astar_search(problem):
      node = Node(problem.initial)
      frontier = PriorityQueue()
      frontier.append(node)

      came_from = {}
      cost_so_far = {}

      came_from[node.state] = None
      cost_so_far[node.state] = 0

      while not frontier.empty():
        current = frontier.pop()

        if problem.goal_test(current.state):
```

```
        return current

      for next in current.expand(problem):
        new_cost = cost_so_far[current.state] + problem.path_cost(cost_so_far[current.state],
                                                                  current.state,
                                                                  current.action,
                                                                  next.state)

        if next.state not in cost_so_far or new_cost < cost_so_far[next.state]:
          cost_so_far[next.state] = new_cost
          priority = new_cost + problem.h(next)
          frontier.append(next)
          came_from[next.state] = current

    return None

def is_in(elt, seq):
    """Similar to (elt in seq), but compares with 'is', not '=='."""
    return any(x is elt for x in seq)
```

Doesn't account sandy squares - BFS algorithm

```
import clips
import random
import numpy as np
# from utils import *


def get_nearest_rock(grid, x, y, orientation):
    # Based on the grid, this function determines the nearest rock for lidar measurements using x, y, and orientation
    init_x = x
    init_y = y
    if orientation == "up":
        while y > 0:
            if (grid[y,x] == 2):
                return (init_y - y)
            y = y-1
    elif orientation == "down":
        while y < len(grid[:,0]):
            if (grid[y,x] == 2):
                return (y - init_y)
            y = y+1
    elif orientation == "left":
        while x > 0:
            if (grid[y,x] == 2):
                return (init_x - x)
            x = x-1
    elif orientation == "right":
        while x < len(grid[0,:]):
            if (grid[y,x] == 2):
                return (x - init_x)
            x = x+1
    return -1

def check_storm_nearby(grid, x, y):
    # Based on the grid, this function checks if there is a nearby storm (for the air quality sensor)
    if y+1 < len(grid[:,0]) and grid[y+1,x] == 4:
        return "true"
    elif y-1 >= 0 and grid[y-1,x] == 4:
        return "true"
    elif x+1 < len(grid[0,:]) and grid[y,x+1] == 4:
        return "true"
    elif x-1 >= 0 and grid[y,x-1] == 4:
        return "true"
    else:
        return "false"

def check_safe_unvisited(x,y,facts):
    # This function should check if location (x, y) is safe and unvisited by querying the KB.
    ### YOUR CODE HERE
    safe_and_unvisited = False
    for fact in facts:
      fact_dict = dict(fact)
      if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["safe"] == "true" and fact_dict["visited"] == "false":
```

```
            safe_and_unvisited = True
        ###
        return safe_and_unvisited

    def check_sandy_hidden_cell(x,y,facts):
        # This function should check if location (x, y) is sandy and unvisited by querying the KB.
        movement_cost = 1
        for fact in facts:
            fact_dict = dict(fact)
            if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["sandy"] == "true":
                movement_cost = 2
        return movement_cost

    def ask_bio_unknown(x,y,facts):
        # This function queries the KB to see if it is unknown if there is a biosample in location (x, y).
        unknown = False
        for fact in facts:
            fact_dict = dict(fact)
            if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["biosample"] == "unknown":
                unknown = True
        return unknown

    def ask_sandy_unknown(x,y,facts):
        # This function queries the KB to see if it is unknown if there is a biosample in location (x, y).
        unknown = False
        for fact in facts:
            fact_dict = dict(fact)
            if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["sandy"] == "unknown":
                unknown = True
        return unknown

    def ask_stormnearby_unknown(x,y,facts):
        # This function queries the KB to see if it is unknown if there is a storm nearby in location (x, y).
        unknown = False
        for fact in facts:
            fact_dict = dict(fact)
            if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["storm_nearby"] == "unknown":
                unknown = True
        return unknown

    def ask_cell_bio(x,y,facts):
        # This function queries the KB to see if there is a biosample in location (x, y).
        ###
        ### YOUR CODE HERE
        bio = False
        for fact in facts:
          fact_dict = dict(fact)
          if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["biosample"] == "true":
            bio = True
        ###
        return bio

    def ask_current_pos(facts,environment):
        # This function queries the KB to see the current position of the agent.
        for fact in facts:
            if fact.template == environment.find_template('agent'):
                agent_dict = dict(fact)
        x = agent_dict["xloc"]
        y = agent_dict["yloc"]
        return [x,y]

    def ask_current_orientation(facts,environment):
        # This function queries the KB to see the current orientation of the agent.
        for fact in facts:
            if fact.template == environment.find_template('agent'):
                agent_dict = dict(fact)
        orientation = agent_dict["orientation"]
        return orientation


    def ask_rover_destroyed(facts,environment):
        # This function queries the KB to see if the agent is destroyed.
        for fact in facts:
            if fact.template == environment.find_template('agent'):
                agent_dict = dict(fact)
        destroyed = agent_dict["destroyed"]
        if destroyed == "true":
```

```
            return True
        else:
            return False

    def ask_agent_sample(facts,environment):
        # This function queries the KB to see if the agent has retrieved the biosample.
        for fact in facts:
            if fact.template == environment.find_template('agent'):
                agent_dict = dict(fact)
        if agent_dict["sample_retrieved"] == "true":
            return True
        else:
            return False

    def ask_if_lidar_this_loc(x,y,o,facts,environment):
        # This function asks if a lidar measurement has been taken in this location/orientation yet.
        lidar_this_loc = False
        action_dicts = []
        for fact in facts:
            if fact.template == environment.find_template('action'):
                action_dict = dict(fact)
                action_dicts.append(action_dict)
        for action in action_dicts:
            if action["xloc"] == x and action["yloc"] == y and action["orientation"] == o:
                lidar_this_loc = True
        return lidar_this_loc


    def plan_route(current, goal, allowed, game_size):
        # This function tries to create a route from current to goal based on the allowed spaces provided for travel.
        problem = PlanRoute(current, goal, allowed, game_size)
        search_result = breadth_first_graph_search(problem)
        ### GRAD STUDENTS REPLACE BREADTH FIRST WITH BEST FIRST OR ASTAR ###
        if search_result is not None:
            return search_result.solution()
        else:
            return None

    ### TEMPLATES ###

    cell_template = """
    (deftemplate cell
        (slot xloc)
        (slot yloc)
        (slot rock)
        (slot biosample)
        (slot storm)
        (slot storm_nearby)
        (slot sandy)
        (slot safe)
        (slot visited)
        (slot up_cell)
        (slot down_cell)
        (slot right_cell)
        (slot left_cell)
        (slot time_checked)
    )
    """

    hidden_cell_template = """
    (deftemplate hidden-cell
        (slot xloc)
        (slot yloc)
        (slot rock)
        (slot biosample)
        (slot storm)
        (slot storm_nearby)
        (slot sandy)
        (slot lidar_right)
        (slot lidar_up)
        (slot lidar_down)
        (slot lidar_left)
    )
    """

    agent_template = """
    (deftemplate agent
```

```
    (slot xloc)
    (slot yloc)
    (slot orientation)
    (slot batt_soc)
    (slot time)
    (slot sample_retrieved)
    (slot destroyed)
    (multislot loc_history)
    (multislot action_history)
)
"""

lidar_measurement_template = """
(deftemplate lidar_measurement
    (slot xloc)
    (slot yloc)
    (slot orientation)
    (slot distance)
    (slot rock_xloc)
    (slot rock_yloc)
)
"""

aq_measurement_template = """
(deftemplate aq_measurement
    (slot xloc)
    (slot yloc)
    (slot air_quality)
)
"""

bio_measurement_template = """
(deftemplate bio_measurement
    (slot xloc)
    (slot yloc)
    (slot organic)
)
"""

traction_measurement_template = """
(deftemplate traction_measurement
    (slot xloc)
    (slot yloc)
    (slot traction)
)
"""

action_template = """
(deftemplate action
    (slot type)
    (slot time)
    (slot xloc)
    (slot yloc)
    (slot orientation)
)
"""
templates = [action_template,traction_measurement_template,aq_measurement_template,bio_measurement_template,lidar_measurement_template,action_
             cell_template,hidden_cell_template]

### RULES ###

lidar_rule = """
(defrule lidar_rule
    ; This rule creates a lidar_measurement fact when a lidar action is taken.
    ?act <- (action (type ?type&:(eq ?type "lidar")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?a
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (lidar_up ?lidar_up) (lidar_down ?lidar_down) (lidar_left ?l
    =>
    (if (eq ?o "up") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_up) (rock_xloc ?x) (rock_yloc (- ?y
    (if (eq ?o "down") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_down) (rock_xloc ?x) (rock_yloc
    (if (eq ?o "left") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_left) (rock_xloc (- ?x ?lidar_le
    (if (eq ?o "right") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_right) (rock_xloc (+ ?x ?lidar_r
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""
rotate_ccw_rule = """
```

```
(defrule rotate_ccw_rule
    ; This rule rotates the agent counterclockwise when a rotate_ccw action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "rotate_ccw")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    =>
    (bind ?new_o ?o)
    (if (eq ?o "up") then (bind ?new_o "left"))
    (if (eq ?o "left") then (bind ?new_o "down"))
    (if (eq ?o "down") then (bind ?new_o "right"))
    (if (eq ?o "right") then (bind ?new_o "up"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (orientation ?new_o) (time ?t) (action_history (create$ $?ah ?type)))
)
"""

rotate_cw_rule = """
(defrule rotate_cw_rule
    ; This rule rotates the agent clockwise when a rotate_cw action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "rotate_cw")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    =>
    (bind ?new_o ?o)
    (if (eq ?o "up") then (bind ?new_o "right"))
    (if (eq ?o "right") then (bind ?new_o "down"))
    (if (eq ?o "down") then (bind ?new_o "left"))
    (if (eq ?o "left") then (bind ?new_o "up"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (orientation ?new_o) (time ?t) (action_history (create$ $?ah ?type)))
)
"""

forward_rule = """
(defrule forward_rule
    ; This rule moves the agent forward when a forward action is taken.
    ?act <- (action (type ?type&:(eq ?type "forward")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    =>
    (if (eq ?o "up") then (bind ?new_x ?x) (bind ?new_y (- ?y 1)))
    (if (eq ?o "down") then (bind ?new_x ?x) (bind ?new_y (+ ?y 1)))
    (if (eq ?o "left") then (bind ?new_x (- ?x 1)) (bind ?new_y ?y))
    (if (eq ?o "right") then (bind ?new_x (+ ?x 1)) (bind ?new_y ?y))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (xloc ?new_x) (yloc ?new_y) (action_history (create$ $?ah ?type)))
)
"""

unvisited_cell_rule = """
(defrule unvisited_cell_rule
    ; This rule sets the cell to visited if it was previously unvisited. It also checks for agent destruction.
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (loc_history $?lh) (action_history $?ah))
    ?hidden-cell <- (hidden-cell (xloc ?hcx&:(eq ?hcx ?x)) (yloc ?hcy&:(eq ?hcy ?y)) (rock ?rock) (storm ?storm))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y))  (visited "false"))
    =>
    (bind ?destroyed "false")
    (if (eq ?rock "true") then (bind ?destroyed "true"))
    (if (eq ?storm "true") then (bind ?destroyed "true"))
    (modify ?cell (rock ?rock) (storm ?storm) (visited "true") (safe "true"))
    (modify ?ag (destroyed ?destroyed))
)
"""

air_quality_rule="""
(defrule air_quality_rule
    ; This rule creates an aq_measurement when an air_quality action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "air_quality")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?ah))
    ?hidden-cell <- (hidden-cell(xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (storm_nearby ?sn))
    =>
    (if (eq ?sn "true") then (assert (aq_measurement (xloc ?x) (yloc ?y) (air_quality "high_ppm"))))
    (if (eq ?sn "false") then (assert (aq_measurement (xloc ?x) (yloc ?y) (air_quality "low_ppm"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""
```

```
bio_rule="""
(defrule bio_rule
    ; This rule creates a bio_measurement when a bio action is taken.
    ?act <- (action (type ?type&:(eq ?type "spectrometer")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?a
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (biosample ?biosample))
    =>
    (if (eq ?biosample "true") then (assert (bio_measurement (xloc ?x) (yloc ?y) (organic "true")))
     else (assert (bio_measurement (xloc ?x) (yloc ?y) (organic "false"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""

traction_rule="""
(defrule traction_rule
    ; This rule creates a traction_measurement when a traction action is taken.
    ?act <- (action (type ?type&:(eq ?type "traction")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?a
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (sandy ?sandy))
    =>
    (if (eq ?sandy "true") then (assert (traction_measurement (xloc ?x) (yloc ?y) (traction "poor")))
     else (assert (traction_measurement (xloc ?x) (yloc ?y) (traction "good"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""

storm_nearby_rule="""
(defrule storm_nearby_rule
    ; This rule checks to see if a storm is nearby a cell based on the air quality measurements.
    ?meas <- (aq_measurement (xloc ?x) (yloc ?y) (air_quality ?aq))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?sn "unknown")
    (if (eq ?aq "low_ppm") then (bind ?sn "false"))
    (if (eq ?aq "high_ppm") then (bind ?sn "true"))
    (modify ?cell (storm_nearby ?sn))
)
"""

traction_meas_rule="""
(defrule traction_meas_rule
    ; This rule checks to see if a cell is sandy based on the traction measurements.
    ?meas <- (traction_measurement (xloc ?x) (yloc ?y) (traction ?tr))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?s "unknown")
    (if (eq ?tr "good") then (bind ?s "false"))
    (if (eq ?tr "poor") then (bind ?s "true"))
    (modify ?cell (sandy ?s))
)
"""

biosample_meas_rule="""
(defrule biosample_meas_rule
    ; This rule checks to see if a cell has a biosample based on the bio measurements.
    ; YOUR CODE HERE
    ?meas <- (bio_measurement (xloc ?x) (yloc ?y) (organic ?org))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?bio "unknown")
    (if (eq ?org "true") then (bind ?bio "true"))
    (if (eq ?org "false") then (bind ?bio "false"))
    (modify ?cell (biosample ?bio))
)
"""

lidar_update_rule="""
(defrule lidar_update_rule
    ; This rule checks to see if a cell has a rock based on the lidar measurements.
    ?meas <- (lidar_measurement (distance ?d&:(neq ?d -1)) (rock_xloc ?rx) (rock_yloc ?ry))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?rx)) (yloc ?cy&:(eq ?cy ?ry)))
    =>
    (modify ?cell (rock "true") (safe "false"))
)
"""
```

```
lidar_y_clear_rule="""
(defrule lidar_y_clear_rule
    ; This cell infers that there is no rock based on lidar measurements returning a -1 value.
    ?meas <- (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o&:(or (eq ?o "up") (eq ?o "down"))) (distance ?d&:(eq ?d -1)))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy) (rock "unknown"))
    =>
    (bind ?rock "unknown")
    (if (eq ?o "up")
     then (if (< ?cy ?y)
           then (bind ?rock "false")
          )
    )
    (if (eq ?o "down")
     then (if (> ?cy ?y)
           then (bind ?rock "false")
          )
    )
    (modify ?cell (rock ?rock))
)
"""

lidar_x_clear_rule="""
(defrule lidar_x_clear_rule
    ; This cell infers that there is no rock based on lidar measurements returning a -1 value.
    ?meas <- (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o&:(or (eq ?o "left") (eq ?o "right"))) (distance ?d&:(eq ?d -1)))
    ?cell <- (cell (xloc ?cx) (yloc ?cy&:(eq ?cy ?y)) (rock "unknown"))
    =>
    (bind ?rock "unknown")
    (if (eq ?o "left")
     then (if (< ?cx ?x)
           then (bind ?rock "false")
          )
    )
    (if (eq ?o "right")
     then (if (> ?cx ?x)
           then (bind ?rock "false")
          )
    )
    (modify ?cell (rock ?rock))
)
"""

drill_rule="""
(defrule drill_rule
    ; This rule retrieves a sample if a drill action is taken in a cell with a biosample.
    ?act <- (action (type ?type&:(eq ?type "drill")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (biosample ?bio))
    =>
    (bind ?sample "false")
    (if (eq ?bio "true") then (bind ?sample "true"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (sample_retrieved ?sample) (time ?t) (action_history (create$ $?ah ?type)))
)
"""

up_cell_rule="""
(defrule up_cell_rule
    ; This rule sets the cell that is above the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (up_cell nil))
    ?up-cell <- (cell (xloc ?cx&:(eq ?x ?cx)) (yloc ?cy&:(eq (- ?y 1) ?cy)))
    =>
    (modify ?cell (up_cell $?up-cell))
)
"""

down_cell_rule="""
(defrule down_cell_rule
    ; This rule sets the cell that is below the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (down_cell nil))
    ?down-cell <- (cell (xloc ?cx&:(eq ?x ?cx)) (yloc ?cy&:(eq (+ ?y 1) ?cy)))
    =>
    (modify ?cell (down_cell ?down-cell))
)
"""
```

```
left_cell_rule="""
(defrule left_cell_rule
    ; This rule sets the cell that is to the left of the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (left_cell nil))
    ?left-cell <- (cell (xloc ?cx&:(eq (- ?x 1) ?cx)) (yloc ?cy&:(eq ?y ?cy)))
    =>
    (modify ?cell (left_cell ?left-cell))
)
"""

right_cell_rule="""
(defrule right_cell_rule
    ; This rule sets the cell that is to the right of the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (right_cell nil))
    ?right-cell <- (cell (xloc ?cx&:(eq (+ ?x 1) ?cx)) (yloc ?cy&:(eq ?y ?cy)))
    =>
    (modify ?cell (right_cell ?right-cell))
)
"""

storm_safe_rule="""
(defrule storm_safe_rule
    ; This rule infers that there is no storm in a cell based on two adjacent conflicting air quality measurements.
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time) (loc_history $?lh) (action_history $?ah))
    ?cell <- (cell (xloc ?cx) (yloc ?cy) (right_cell ?rc) (left_cell ?lc) (up_cell ?uc) (down_cell ?dc) (storm "unknown") (time_checked ?t&:(
    =>
    (bind ?storm "unknown")
    (bind ?safe "unknown")
    (bind ?lcsnb "unknown")
    (bind ?rcsnb "unknown")
    (bind ?ucsnb "unknown")
    (bind ?dcsnb "unknown")
    (if (neq ?lc nil)
     then (if (eq (fact-slot-value ?lc storm_nearby) "false")
           then (bind ?lcsnb "false")
          )
    )
    (if (neq ?rc nil)
     then (if (eq (fact-slot-value ?rc storm_nearby) "false")
           then (bind ?rcsnb "false")
          )
    )
    (if (neq ?uc nil)
     then (if (eq (fact-slot-value ?uc storm_nearby) "false")
           then (bind ?ucsnb "false")
          )
    )
    (if (neq ?dc nil)
     then (if (eq (fact-slot-value ?dc storm_nearby) "false")
           then (bind ?dcsnb "false")
          )
    )
    (if (or (eq ?lcsnb "false") (eq ?rcsnb "false") (eq ?ucsnb "false") (eq ?dcsnb "false"))
     then (bind ?storm "false")
    )
    (bind ?new_t ?agent_time)
    (modify ?cell (storm ?storm) (time_checked ?new_t))
)
"""

safe_cell_rule="""
(defrule safe_cell_rule
    ; This rule checks if the cell has no rock or storm and if so, sets the cell to safe.
    ; YOUR CODE HERE
    ?cell <- (cell (rock ?r) (storm ?s) (safe "unknown"))
    =>
    (bind ?safe "unknown")
    (if (and (eq ?r "false") (eq ?s "false")) then (bind ?safe "true"))
    (modify ?cell (safe ?safe))
)
"""
rules = [lidar_rule,rotate_ccw_rule,rotate_cw_rule,forward_rule,air_quality_rule,traction_rule,bio_rule,storm_nearby_rule,traction_meas_rule,l
         unvisited_cell_rule,lidar_update_rule,drill_rule,left_cell_rule,right_cell_rule,up_cell_rule,down_cell_rule,storm_safe_rule,lidar_x_
         safe_cell_rule]


def new_game(templates,rules):
```

```
# This function creates a new game, which consists of a random 6x6 grid with 3 rocks, 3 sandy cells, 2 storms, 1 goal and 1 start cell.
# It also loads all of the rules and templates into CLIPS.
environment = clips.Environment()
for template in templates:
    environment.build(template)
for rule in rules:
    environment.build(rule)

# Generate random grid.
random_grid = np.zeros((6,6))
stuff_cells = np.random.choice(random_grid.size, 10, replace=False) # 10 = 3 rocks + 3 sandy + 2 storms + 1 goal + 1 start
# 1 is start, 2 is rocks, 3 is sandy, 4 is storm, 5 is organic sample
random_grid.ravel()[stuff_cells[0]] = 1
random_grid.ravel()[stuff_cells[1:4]] = 2
random_grid.ravel()[stuff_cells[4:7]] = 3
random_grid.ravel()[stuff_cells[7:9]] = 4
random_grid.ravel()[stuff_cells[9]] = 5
start_x = None
start_y = None
print(random_grid)

# populate hidden cells (the real environment if the problem was fully observable)
for i in range(len(random_grid[:,0])):
    for j in range(len(random_grid[0,:])):
        x = j
        y = i
        # row i, column j
        # print("Row "+str(i)+", column "+str(j)+" is "+str(random_grid[i,j]))
        lidar_up = get_nearest_rock(random_grid,x,y,"up")
        lidar_down = get_nearest_rock(random_grid,x,y,"down")
        lidar_left = get_nearest_rock(random_grid,x,y,"left")
        lidar_right = get_nearest_rock(random_grid,x,y,"right")
        storm_nearby = check_storm_nearby(random_grid,x,y)

        if random_grid[y,x] == 4:
            storm = "true"
        else:
            storm = "false"
        if random_grid[y,x] == 3:
            sandy = "true"
        else:
            sandy = "false"
        if random_grid[y,x] == 2:
            rock = "true"
        else:
            rock = "false"
        if random_grid[y,x] == 5:
            biosample = "true"
        else:
            biosample = "false"
        if random_grid[y,x] == 1:
            start_x = x
            start_y = y
        # assert a new fact through its template
        hidden_cell_template = environment.find_template('hidden-cell')
        fact = hidden_cell_template.assert_fact(xloc=x,
                                    yloc=y,
                                    rock=rock,
                                    biosample=biosample,
                                    storm=storm,
                                    storm_nearby=storm_nearby,
                                    sandy=sandy,
                                    lidar_up=lidar_up,
                                    lidar_down=lidar_down,
                                    lidar_left=lidar_left,
                                    lidar_right=lidar_right
                                    )

# populate agent's cell KB (what the agent knows)
for i in range(len(random_grid[:,0])):
    for j in range(len(random_grid[0,:])):
        x = j
        y = i
        if not (x == start_x and y == start_y):
            storm = "unknown"
            storm_nearby = "unknown"
            sandy = "unknown"
```

```
                rock = "unknown"
                safe = "unknown"
                biosample = "unknown"
                visited = "false"
            else:
                storm = "false"
                sandy = "false"
                storm_nearby = "unknown"
                rock = "false"
                biosample = "false"
                safe = "true"
                visited = "true"
            cell_template = environment.find_template('cell')
            cell_fact = cell_template.assert_fact(xloc=x,
                                        yloc=y,
                                        rock=rock,
                                        biosample=biosample,
                                        storm=storm,
                                        storm_nearby=storm_nearby,
                                        sandy=sandy,
                                        safe=safe,
                                        visited=visited,
                                        time_checked=0
                                        )

    # Populate initial agent state.
    agent_template = environment.find_template('agent')
    agent = agent_template.assert_fact(xloc=start_x,
                                    yloc=start_y,
                                    orientation="right",
                                    batt_soc=1,
                                    time=0,
                                    sample_retrieved="false",
                                    destroyed="false",
                                    loc_history=[],
                                    action_history=[]
                                    )
    return environment, random_grid, start_x, start_y

def hybrid_agent(environment,grid,start_x,start_y):
    # Grad students will need to augment this agent with a way to track the movement actions taken in sandy vs. non-sandy areas.
    sim_length = 10000
    t = 0
    total_movement_cost = 0
    while t < sim_length:
        # Get all of the facts associated with cells.
        cell_facts = []
        for fact in environment.facts():
            if fact.template == environment.find_template('cell'):
                cell_facts.append(fact)
        # Get all of the facts associated with hidden cells. (Only used for sandy check)
        hidden_cell_facts = []
        for fact in environment.facts():
            if fact.template == environment.find_template('hidden-cell'):
                hidden_cell_facts.append(fact)


        # Get all facts.
        facts = list(environment.facts())
        plan = []

        # Get current rover conditions from KB.
        current_pos = ask_current_pos(facts,environment)
        x = current_pos[0]
        y = current_pos[1]
        destroyed = ask_rover_destroyed(facts,environment)
        orientation = ask_current_orientation(facts,environment)
        rover_pos = RoverPosition(x,y,orientation)

        # Check if the rover has been destroyed, end the game if so.
        if destroyed:
            print("Game over.")
            return 1, total_movement_cost

        # Check if the agent has retrieved the sample and is back in the start position.
        if ask_agent_sample(facts,environment) and x == start_x and y == start_y:
            print("You win!")
```

```
    return 0, total_movement_cost

# Get a list of safe cells that we can traverse without worry.
safe_points = list()
for cell in cell_facts:
    cell_dict = dict(cell)
    if cell_dict["safe"] == "true":
        safe_points.append([cell_dict["xloc"],cell_dict["yloc"]])

# Check if agent has retrieved the sample.
if len(plan) == 0:
    if ask_agent_sample(facts,environment):
        print("Sample acquired!")
        goals = list()
        goals.append([start_x, start_y])
        actions = plan_route(rover_pos, goals[0], safe_points, len(grid[:,0]))
        plan.extend(actions)

# Check if the agent is in a cell with the biosample. If so, drill.
if len(plan) == 0:
    if ask_cell_bio(x,y,cell_facts):
        action = "drill"
        plan.append(action)


# Here you should write code that checks for unvisited but safe spots to visit, and adds maneuvers to those spots to the plan,
# using the plan_route function. See the "not_unsafe" code below for a similar implementation.

###
### YOUR CODE HERE
if len(plan) == 0:
    safe = list()
    for cell in cell_facts:
        cell_dict = dict(cell)
        if cell_dict["safe"] == "true" and cell_dict["visited"] == "false" and not (cell_dict["xloc"] == x and cell_dict["yloc"] == y
            safe.append([cell_dict["xloc"],cell_dict["yloc"]])
    if len(safe) > 0:
        valid_temps = []
        for safe_point in safe:
            safe_points.append(safe_point)
            temp = plan_route(rover_pos, safe_point, safe_points, len(grid[:,0]))
            safe_points.remove(safe_point)
            if temp is not None:
                valid_temps.append(temp)
        if len(valid_temps) > 0:
            temp = random.choice(valid_temps)
            plan.extend(temp)
###

# Check if lidar has been done in this cell/orientation. If not, lidar.
if len(plan) == 0:
    if not ask_if_lidar_this_loc(x,y,ask_current_orientation(facts,environment),facts,environment):
        action = "lidar"
        plan.append(action)

# Check if traction has been measured in this cell/orientation. If not, traction.
if len(plan) == 0:
    if ask_sandy_unknown(x,y,cell_facts):
        action = "traction"
        plan.append(action)

# Check if air quality/nearby storm status has been measured in this cell/orientation. If not, air_quality.
if len(plan) == 0:
    if ask_stormnearby_unknown(x,y,cell_facts):
        action = "air_quality"
        plan.append(action)

# Check if the spectrometer has been used in this cell. If not, spectrometer.
if len(plan) == 0:
    if ask_bio_unknown(x,y,cell_facts):
        action = "spectrometer"
        plan.append(action)

# If there are still no safe and unvisited spots remaining and this cell has had all measurements performed, try moving to an unvisite
# that at least is not unsafe.
if len(plan) == 0:
    not_unsafe = list()
```

```
            for cell in cell_facts:
                cell_dict = dict(cell)
                if not cell_dict["safe"] == "false" and cell_dict["visited"] == "false" and not (cell_dict["xloc"] == x and cell_dict["yloc"]
                    not_unsafe.append([cell_dict["xloc"],cell_dict["yloc"]])
            if len(not_unsafe) > 0:
                valid_temps = []
                for not_unsafe_point in not_unsafe:
                    safe_points.append(not_unsafe_point)
                    temp = plan_route(rover_pos, not_unsafe_point, safe_points, len(grid[:,0]))
                    safe_points.remove(not_unsafe_point)
                    if temp is not None:
                        valid_temps.append(temp)
                if len(valid_temps) > 0:
                    temp = random.choice(valid_temps)
                    plan.extend(temp)

        # If there are no unknown spots to explore, all of the unvisited spots are known to be unsafe, the goal must be blocked off.
        if len(plan) == 0:
            print("No actions to take!")
            print("Game over.")
            return 1, total_movement_cost

        # Execute the first action in the plan.
        action = plan[0]

        # Translating from plan_route terminology to clips rule terminology.
        if action == "Forward":
            action = "forward"
        elif action == "Turnright":
            action = "rotate_cw"
        elif action == "Turnleft":
            action = "rotate_ccw"

        print("Current time: "+str(t))
        print("Current position: "+str(current_pos))
        print("Action taken: "+str(action))

        plan = plan[1:]

        # Update the KB with the action taken. Run the rule environment so that any new firings can occur.
        action_template = environment.find_template('action')
        action_template.assert_fact(type=action,time=t,xloc=x,yloc=y,orientation=ask_current_orientation(facts,environment))
        if action == "forward" or action == "rotate_cw" or action == "rotate_ccw":
            movement_cost = check_sandy_hidden_cell(x,y,hidden_cell_facts)
            total_movement_cost += movement_cost
        environment.run()
        t += 1
    return 1, total_movement_cost


# This code runs 100 trials and tracks the number of victories and the average movement cost.
sum = 0
sims = 100
mvmt_cost_sum = 0
for i in range(sims):
    env, grid, start_x, start_y = new_game(templates,rules)
    result, mvmt_cost = hybrid_agent(env,grid,start_x,start_y)
    sum += result
    mvmt_cost_sum += mvmt_cost
print("Number of victories: "+str(sims-sum))
print("Movement cost average: "+str(mvmt_cost_sum/sims))

# This prints out the working memory for the last trial run.
print("Final grid observations: ")
for fact in env.facts():
    if fact.template == env.find_template('cell'):
        d = dict(fact)
        exclude_keys = ['up_cell', 'down_cell','right_cell','left_cell']
        new_d = {k: d[k] for k in set(list(d.keys())) - set(exclude_keys)}
        print(", ".join([key+": "+str(value) for key, value in sorted(new_d.items(), key=lambda x: x[0])]))
```

```
Current time: 38
Current position: [0, 5]
Action taken: rotate_ccw
Sample acquired!
Current time: 39
Current position: [0, 5]
Action taken: rotate_ccw
Sample acquired!
Current time: 40
Current position: [0, 5]
Action taken: forward
You win!
Number of victories: 41
Movement cost average: 41.2
Final grid observations:
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: true, safe: false, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: false,
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 41, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 37, visited: fals
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 3, visited: false
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 9, visited: false
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 14, visited: fals
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 19, visited: fals
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 24, visited: fals
biosample: true, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 3, visited: true, xloc: 0, yl
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 0, visited: true, xloc: 1, y
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 3, visited: true, xloc: 2, y
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 9, visited: true, xloc: 3, y
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 14, visited: true, xloc: 4,
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 19, visited: true, xloc: 5
```

Doesn't account for sandy squares – A* Algorithm

```
import clips
import random
import numpy as np
# from utils import *


def get_nearest_rock(grid, x, y, orientation):
    # Based on the grid, this function determines the nearest rock for lidar measurements using x, y, and orientation
    init_x = x
    init_y = y
    if orientation == "up":
        while y > 0:
            if (grid[y,x] == 2):
                return (init_y - y)
            y = y-1
    elif orientation == "down":
        while y < len(grid[:,0]):
            if (grid[y,x] == 2):
                return (y - init_y)
            y = y+1
    elif orientation == "left":
        while x > 0:
            if (grid[y,x] == 2):
                return (init_x - x)
            x = x-1
```

```
        elif orientation == "right":
            while x < len(grid[0,:]):
                if (grid[y,x] == 2):
                    return (x - init_x)
                x = x+1
        return -1

    def check_storm_nearby(grid, x, y):
        # Based on the grid, this function checks if there is a nearby storm (for the air quality sensor)
        if y+1 < len(grid[:,0]) and grid[y+1,x] == 4:
            return "true"
        elif y-1 >= 0 and grid[y-1,x] == 4:
            return "true"
        elif x+1 < len(grid[0,:]) and grid[y,x+1] == 4:
            return "true"
        elif x-1 >= 0 and grid[y,x-1] == 4:
            return "true"
        else:
            return "false"

    def check_safe_unvisited(x,y,facts):
        # This function should check if location (x, y) is safe and unvisited by querying the KB.
        ### YOUR CODE HERE
        safe_and_unvisited = False
        for fact in facts:
          fact_dict = dict(fact)
          if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["safe"] == "true" and fact_dict["visited"] == "false":
              safe_and_unvisited = True
        ###
        return safe_and_unvisited

    def check_sandy_hidden_cell(x,y,facts):
        # This function should check if location (x, y) is sandy and unvisited by querying the KB.
        movement_cost = 1
        for fact in facts:
            fact_dict = dict(fact)
            if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["sandy"] == "true":
                movement_cost = 2
        return movement_cost

    def ask_bio_unknown(x,y,facts):
        # This function queries the KB to see if it is unknown if there is a biosample in location (x, y).
        unknown = False
        for fact in facts:
            fact_dict = dict(fact)
            if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["biosample"] == "unknown":
                unknown = True
        return unknown

    def ask_sandy_unknown(x,y,facts):
        # This function queries the KB to see if it is unknown if there is a biosample in location (x, y).
        unknown = False
        for fact in facts:
            fact_dict = dict(fact)
            if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["sandy"] == "unknown":
                unknown = True
        return unknown

    def ask_stormnearby_unknown(x,y,facts):
        # This function queries the KB to see if it is unknown if there is a storm nearby in location (x, y).
        unknown = False
        for fact in facts:
            fact_dict = dict(fact)
            if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["storm_nearby"] == "unknown":
                unknown = True
        return unknown

    def ask_cell_bio(x,y,facts):
        # This function queries the KB to see if there is a biosample in location (x, y).
        ###
        ### YOUR CODE HERE
        bio = False
        for fact in facts:
          fact_dict = dict(fact)
          if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["biosample"] == "true":
            bio = True
        ###
```

```
        return bio

    def ask_current_pos(facts,environment):
        # This function queries the KB to see the current position of the agent.
        for fact in facts:
            if fact.template == environment.find_template('agent'):
                agent_dict = dict(fact)
        x = agent_dict["xloc"]
        y = agent_dict["yloc"]
        return [x,y]

    def ask_current_orientation(facts,environment):
        # This function queries the KB to see the current orientation of the agent.
        for fact in facts:
            if fact.template == environment.find_template('agent'):
                agent_dict = dict(fact)
        orientation = agent_dict["orientation"]
        return orientation


    def ask_rover_destroyed(facts,environment):
        # This function queries the KB to see if the agent is destroyed.
        for fact in facts:
            if fact.template == environment.find_template('agent'):
                agent_dict = dict(fact)
        destroyed = agent_dict["destroyed"]
        if destroyed == "true":
            return True
        else:
            return False

    def ask_agent_sample(facts,environment):
        # This function queries the KB to see if the agent has retrieved the biosample.
        for fact in facts:
            if fact.template == environment.find_template('agent'):
                agent_dict = dict(fact)
        if agent_dict["sample_retrieved"] == "true":
            return True
        else:
            return False

    def ask_if_lidar_this_loc(x,y,o,facts,environment):
        # This function asks if a lidar measurement has been taken in this location/orientation yet.
        lidar_this_loc = False
        action_dicts = []
        for fact in facts:
            if fact.template == environment.find_template('action'):
                action_dict = dict(fact)
                action_dicts.append(action_dict)
        for action in action_dicts:
            if action["xloc"] == x and action["yloc"] == y and action["orientation"] == o:
                lidar_this_loc = True
        return lidar_this_loc


    def plan_route(current, goal, allowed, game_size):
        # This function tries to create a route from current to goal based on the allowed spaces provided for travel.
        problem = PlanRoute(current, goal, allowed, game_size)
        search_result = astar_search(problem)
        ### GRAD STUDENTS REPLACE BREADTH FIRST WITH BEST FIRST OR ASTAR ###
        if search_result is not None:
            return search_result.solution()
        else:
            return None

    ### TEMPLATES ###

    cell_template = """
    (deftemplate cell
        (slot xloc)
        (slot yloc)
        (slot rock)
        (slot biosample)
        (slot storm)
        (slot storm_nearby)
        (slot sandy)
        (slot safe)
```

```
        (slot visited)
        (slot up_cell)
        (slot down_cell)
        (slot right_cell)
        (slot left_cell)
        (slot time_checked)
    )
    """

    hidden_cell_template = """
    (deftemplate hidden-cell
        (slot xloc)
        (slot yloc)
        (slot rock)
        (slot biosample)
        (slot storm)
        (slot storm_nearby)
        (slot sandy)
        (slot lidar_right)
        (slot lidar_up)
        (slot lidar_down)
        (slot lidar_left)
    )
    """

    agent_template = """
    (deftemplate agent
        (slot xloc)
        (slot yloc)
        (slot orientation)
        (slot batt_soc)
        (slot time)
        (slot sample_retrieved)
        (slot destroyed)
        (multislot loc_history)
        (multislot action_history)
    )
    """

    lidar_measurement_template = """
    (deftemplate lidar_measurement
        (slot xloc)
        (slot yloc)
        (slot orientation)
        (slot distance)
        (slot rock_xloc)
        (slot rock_yloc)
    )
    """

    aq_measurement_template = """
    (deftemplate aq_measurement
        (slot xloc)
        (slot yloc)
        (slot air_quality)
    )
    """

    bio_measurement_template = """
    (deftemplate bio_measurement
        (slot xloc)
        (slot yloc)
        (slot organic)
    )
    """

    traction_measurement_template = """
    (deftemplate traction_measurement
        (slot xloc)
        (slot yloc)
        (slot traction)
    )
    """

    action_template = """
    (deftemplate action
        (slot type)
```

```
    (slot time)
    (slot xloc)
    (slot yloc)
    (slot orientation)
)
"""
templates = [action_template,traction_measurement_template,aq_measurement_template,bio_measurement_template,lidar_measurement_template,action_
             cell_template,hidden_cell_template]

### RULES ###

lidar_rule = """
(defrule lidar_rule
    ; This rule creates a lidar_measurement fact when a lidar action is taken.
    ?act <- (action (type ?type&:(eq ?type "lidar")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?a
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (lidar_up ?lidar_up) (lidar_down ?lidar_down) (lidar_left ?l
    =>
    (if (eq ?o "up") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_up) (rock_xloc ?x) (rock_yloc (- ?y
    (if (eq ?o "down") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_down) (rock_xloc ?x) (rock_yloc
    (if (eq ?o "left") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_left) (rock_xloc (- ?x ?lidar_le
    (if (eq ?o "right") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_right) (rock_xloc (+ ?x ?lidar_r
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""
rotate_ccw_rule = """
(defrule rotate_ccw_rule
    ; This rule rotates the agent counterclockwise when a rotate_ccw action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "rotate_ccw")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    =>
    (bind ?new_o ?o)
    (if (eq ?o "up") then (bind ?new_o "left"))
    (if (eq ?o "left") then (bind ?new_o "down"))
    (if (eq ?o "down") then (bind ?new_o "right"))
    (if (eq ?o "right") then (bind ?new_o "up"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (orientation ?new_o) (time ?t) (action_history (create$ $?ah ?type)))
)
"""

rotate_cw_rule = """
(defrule rotate_cw_rule
    ; This rule rotates the agent clockwise when a rotate_cw action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "rotate_cw")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    =>
    (bind ?new_o ?o)
    (if (eq ?o "up") then (bind ?new_o "right"))
    (if (eq ?o "right") then (bind ?new_o "down"))
    (if (eq ?o "down") then (bind ?new_o "left"))
    (if (eq ?o "left") then (bind ?new_o "up"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (orientation ?new_o) (time ?t) (action_history (create$ $?ah ?type)))
)
"""

forward_rule = """
(defrule forward_rule
    ; This rule moves the agent forward when a forward action is taken.
    ?act <- (action (type ?type&:(eq ?type "forward")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    =>
    (if (eq ?o "up") then (bind ?new_x ?x) (bind ?new_y (- ?y 1)))
    (if (eq ?o "down") then (bind ?new_x ?x) (bind ?new_y (+ ?y 1)))
    (if (eq ?o "left") then (bind ?new_x (- ?x 1)) (bind ?new_y ?y))
    (if (eq ?o "right") then (bind ?new_x (+ ?x 1)) (bind ?new_y ?y))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (xloc ?new_x) (yloc ?new_y) (action_history (create$ $?ah ?type)))
)
"""

unvisited_cell_rule = """
(defrule unvisited_cell_rule
```

```
    ; This rule sets the cell to visited if it was previously unvisited. It also checks for agent destruction.
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (loc_history $?lh) (action_history $?ah))
    ?hidden-cell <- (hidden-cell (xloc ?hcx&:(eq ?hcx ?x)) (yloc ?hcy&:(eq ?hcy ?y)) (rock ?rock) (storm ?storm))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y))  (visited "false"))
    =>
    (bind ?destroyed "false")
    (if (eq ?rock "true") then (bind ?destroyed "true"))
    (if (eq ?storm "true") then (bind ?destroyed "true"))
    (modify ?cell (rock ?rock) (storm ?storm) (visited "true") (safe "true"))
    (modify ?ag (destroyed ?destroyed))
)
"""

air_quality_rule="""
(defrule air_quality_rule
    ; This rule creates an aq_measurement when an air_quality action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "air_quality")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?ah))
    ?hidden-cell <- (hidden-cell(xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (storm_nearby ?sn))
    =>
    (if (eq ?sn "true") then (assert (aq_measurement (xloc ?x) (yloc ?y) (air_quality "high_ppm"))))
    (if (eq ?sn "false") then (assert (aq_measurement (xloc ?x) (yloc ?y) (air_quality "low_ppm"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""
bio_rule="""
(defrule bio_rule
    ; This rule creates a bio_measurement when a bio action is taken.
    ?act <- (action (type ?type&:(eq ?type "spectrometer")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?a
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (biosample ?biosample))
    =>
    (if (eq ?biosample "true") then (assert (bio_measurement (xloc ?x) (yloc ?y) (organic "true")))
     else (assert (bio_measurement (xloc ?x) (yloc ?y) (organic "false"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""

traction_rule="""
(defrule traction_rule
    ; This rule creates a traction_measurement when a traction action is taken.
    ?act <- (action (type ?type&:(eq ?type "traction")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?a
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (sandy ?sandy))
    =>
    (if (eq ?sandy "true") then (assert (traction_measurement (xloc ?x) (yloc ?y) (traction "poor")))
     else (assert (traction_measurement (xloc ?x) (yloc ?y) (traction "good"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""

storm_nearby_rule="""
(defrule storm_nearby_rule
    ; This rule checks to see if a storm is nearby a cell based on the air quality measurements.
    ?meas <- (aq_measurement (xloc ?x) (yloc ?y) (air_quality ?aq))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?sn "unknown")
    (if (eq ?aq "low_ppm") then (bind ?sn "false"))
    (if (eq ?aq "high_ppm") then (bind ?sn "true"))
    (modify ?cell (storm_nearby ?sn))
)
"""

traction_meas_rule="""
(defrule traction_meas_rule
    ; This rule checks to see if a cell is sandy based on the traction measurements.
    ?meas <- (traction_measurement (xloc ?x) (yloc ?y) (traction ?tr))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?s "unknown")
    (if (eq ?tr "good") then (bind ?s "false"))
    (if (eq ?tr "poor") then (bind ?s "true"))
```

```
    (modify ?cell (sandy ?s))
)
"""

biosample_meas_rule="""
(defrule biosample_meas_rule
    ; This rule checks to see if a cell has a biosample based on the bio measurements.
    ; YOUR CODE HERE
    ?meas <- (bio_measurement (xloc ?x) (yloc ?y) (organic ?org))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?bio "unknown")
    (if (eq ?org "true") then (bind ?bio "true"))
    (if (eq ?org "false") then (bind ?bio "false"))
    (modify ?cell (biosample ?bio))
)
"""

lidar_update_rule="""
(defrule lidar_update_rule
    ; This rule checks to see if a cell has a rock based on the lidar measurements.
    ?meas <- (lidar_measurement (distance ?d&:(neq ?d -1)) (rock_xloc ?rx) (rock_yloc ?ry))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?rx)) (yloc ?cy&:(eq ?cy ?ry)))
    =>
    (modify ?cell (rock "true") (safe "false"))
)
"""

lidar_y_clear_rule="""
(defrule lidar_y_clear_rule
    ; This cell infers that there is no rock based on lidar measurements returning a -1 value.
    ?meas <- (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o&:(or (eq ?o "up") (eq ?o "down"))) (distance ?d&:(eq ?d -1)))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy) (rock "unknown"))
    =>
    (bind ?rock "unknown")
    (if (eq ?o "up")
     then (if (< ?cy ?y)
           then (bind ?rock "false")
          )
    )
    (if (eq ?o "down")
     then (if (> ?cy ?y)
           then (bind ?rock "false")
          )
    )
    (modify ?cell (rock ?rock))
)
"""

lidar_x_clear_rule="""
(defrule lidar_x_clear_rule
    ; This cell infers that there is no rock based on lidar measurements returning a -1 value.
    ?meas <- (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o&:(or (eq ?o "left") (eq ?o "right"))) (distance ?d&:(eq ?d -1)))
    ?cell <- (cell (xloc ?cx) (yloc ?cy&:(eq ?cy ?y)) (rock "unknown"))
    =>
    (bind ?rock "unknown")
    (if (eq ?o "left")
     then (if (< ?cx ?x)
           then (bind ?rock "false")
          )
    )
    (if (eq ?o "right")
     then (if (> ?cx ?x)
           then (bind ?rock "false")
          )
    )
    (modify ?cell (rock ?rock))
)
"""

drill_rule="""
(defrule drill_rule
    ; This rule retrieves a sample if a drill action is taken in a cell with a biosample.
    ?act <- (action (type ?type&:(eq ?type "drill")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (biosample ?bio))
    =>
```

```
    (bind ?sample "false")
    (if (eq ?bio "true") then (bind ?sample "true"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (sample_retrieved ?sample) (time ?t) (action_history (create$ $?ah ?type)))
)
"""

up_cell_rule="""
(defrule up_cell_rule
    ; This rule sets the cell that is above the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (up_cell nil))
    ?up-cell <- (cell (xloc ?cx&:(eq ?x ?cx)) (yloc ?cy&:(eq (- ?y 1) ?cy)))
    =>
    (modify ?cell (up_cell $?up-cell))
)
"""

down_cell_rule="""
(defrule down_cell_rule
    ; This rule sets the cell that is below the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (down_cell nil))
    ?down-cell <- (cell (xloc ?cx&:(eq ?x ?cx)) (yloc ?cy&:(eq (+ ?y 1) ?cy)))
    =>
    (modify ?cell (down_cell ?down-cell))
)
"""

left_cell_rule="""
(defrule left_cell_rule
    ; This rule sets the cell that is to the left of the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (left_cell nil))
    ?left-cell <- (cell (xloc ?cx&:(eq (- ?x 1) ?cx)) (yloc ?cy&:(eq ?y ?cy)))
    =>
    (modify ?cell (left_cell ?left-cell))
)
"""

right_cell_rule="""
(defrule right_cell_rule
    ; This rule sets the cell that is to the right of the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (right_cell nil))
    ?right-cell <- (cell (xloc ?cx&:(eq (+ ?x 1) ?cx)) (yloc ?cy&:(eq ?y ?cy)))
    =>
    (modify ?cell (right_cell ?right-cell))
)
"""

storm_safe_rule="""
(defrule storm_safe_rule
    ; This rule infers that there is no storm in a cell based on two adjacent conflicting air quality measurements.
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time) (loc_history $?lh) (action_history $?ah))
    ?cell <- (cell (xloc ?cx) (yloc ?cy) (right_cell ?rc) (left_cell ?lc) (up_cell ?uc) (down_cell ?dc) (storm "unknown") (time_checked ?t&:(
    =>
    (bind ?storm "unknown")
    (bind ?safe "unknown")
    (bind ?lcsnb "unknown")
    (bind ?rcsnb "unknown")
    (bind ?ucsnb "unknown")
    (bind ?dcsnb "unknown")
    (if (neq ?lc nil)
     then (if (eq (fact-slot-value ?lc storm_nearby) "false")
           then (bind ?lcsnb "false")
          )
    )
    (if (neq ?rc nil)
     then (if (eq (fact-slot-value ?rc storm_nearby) "false")
           then (bind ?rcsnb "false")
          )
    )
    (if (neq ?uc nil)
     then (if (eq (fact-slot-value ?uc storm_nearby) "false")
           then (bind ?ucsnb "false")
          )
    )
    (if (neq ?dc nil)
     then (if (eq (fact-slot-value ?dc storm_nearby) "false")
           then (bind ?dcsnb "false")
```

```
            )
        )
        (if (or (eq ?lcsnb "false") (eq ?rcsnb "false") (eq ?ucsnb "false") (eq ?dcsnb "false"))
         then (bind ?storm "false")
        )
        (bind ?new_t ?agent_time)
        (modify ?cell (storm ?storm) (time_checked ?new_t))
    )
    """

    safe_cell_rule="""
    (defrule safe_cell_rule
        ; This rule checks if the cell has no rock or storm and if so, sets the cell to safe.
        ; YOUR CODE HERE
        ?cell <- (cell (rock ?r) (storm ?s) (safe "unknown"))
        =>
        (bind ?safe "unknown")
        (if (and (eq ?r "false") (eq ?s "false")) then (bind ?safe "true"))
        (modify ?cell (safe ?safe))
    )
    """
    rules = [lidar_rule,rotate_ccw_rule,rotate_cw_rule,forward_rule,air_quality_rule,traction_rule,bio_rule,storm_nearby_rule,traction_meas_rule,
             unvisited_cell_rule,lidar_update_rule,drill_rule,left_cell_rule,right_cell_rule,up_cell_rule,down_cell_rule,storm_safe_rule,lidar_x_
             safe_cell_rule]


    def new_game(templates,rules):
        # This function creates a new game, which consists of a random 6x6 grid with 3 rocks, 3 sandy cells, 2 storms, 1 goal and 1 start cell.
        # It also loads all of the rules and templates into CLIPS.
        environment = clips.Environment()
        for template in templates:
            environment.build(template)
        for rule in rules:
            environment.build(rule)

        # Generate random grid.
        random_grid = np.zeros((6,6))
        stuff_cells = np.random.choice(random_grid.size, 10, replace=False) # 10 = 3 rocks + 3 sandy + 2 storms + 1 goal + 1 start
        # 1 is start, 2 is rocks, 3 is sandy, 4 is storm, 5 is organic sample
        random_grid.ravel()[stuff_cells[0]] = 1
        random_grid.ravel()[stuff_cells[1:4]] = 2
        random_grid.ravel()[stuff_cells[4:7]] = 3
        random_grid.ravel()[stuff_cells[7:9]] = 4
        random_grid.ravel()[stuff_cells[9]] = 5
        start_x = None
        start_y = None
        print(random_grid)

        # populate hidden cells (the real environment if the problem was fully observable)
        for i in range(len(random_grid[:,0])):
            for j in range(len(random_grid[0,:])):
                x = j
                y = i
                # row i, column j
                # print("Row "+str(i)+", column "+str(j)+" is "+str(random_grid[i,j]))
                lidar_up = get_nearest_rock(random_grid,x,y,"up")
                lidar_down = get_nearest_rock(random_grid,x,y,"down")
                lidar_left = get_nearest_rock(random_grid,x,y,"left")
                lidar_right = get_nearest_rock(random_grid,x,y,"right")
                storm_nearby = check_storm_nearby(random_grid,x,y)

                if random_grid[y,x] == 4:
                    storm = "true"
                else:
                    storm = "false"
                if random_grid[y,x] == 3:
                    sandy = "true"
                else:
                    sandy = "false"
                if random_grid[y,x] == 2:
                    rock = "true"
                else:
                    rock = "false"
                if random_grid[y,x] == 5:
                    biosample = "true"
                else:
                    biosample = "false"
```

```
            if random_grid[y,x] == 1:
                start_x = x
                start_y = y
            # assert a new fact through its template
            hidden_cell_template = environment.find_template('hidden-cell')
            fact = hidden_cell_template.assert_fact(xloc=x,
                                        yloc=y,
                                        rock=rock,
                                        biosample=biosample,
                                        storm=storm,
                                        storm_nearby=storm_nearby,
                                        sandy=sandy,
                                        lidar_up=lidar_up,
                                        lidar_down=lidar_down,
                                        lidar_left=lidar_left,
                                        lidar_right=lidar_right
                                        )

    # populate agent's cell KB (what the agent knows)
    for i in range(len(random_grid[:,0])):
        for j in range(len(random_grid[0,:])):
            x = j
            y = i
            if not (x == start_x and y == start_y):
                storm = "unknown"
                storm_nearby = "unknown"
                sandy = "unknown"
                rock = "unknown"
                safe = "unknown"
                biosample = "unknown"
                visited = "false"
            else:
                storm = "false"
                sandy = "false"
                storm_nearby = "unknown"
                rock = "false"
                biosample = "false"
                safe = "true"
                visited = "true"
            cell_template = environment.find_template('cell')
            cell_fact = cell_template.assert_fact(xloc=x,
                                        yloc=y,
                                        rock=rock,
                                        biosample=biosample,
                                        storm=storm,
                                        storm_nearby=storm_nearby,
                                        sandy=sandy,
                                        safe=safe,
                                        visited=visited,
                                        time_checked=0
                                        )

    # Populate initial agent state.
    agent_template = environment.find_template('agent')
    agent = agent_template.assert_fact(xloc=start_x,
                                    yloc=start_y,
                                    orientation="right",
                                    batt_soc=1,
                                    time=0,
                                    sample_retrieved="false",
                                    destroyed="false",
                                    loc_history=[],
                                    action_history=[]
                                    )
    return environment, random_grid, start_x, start_y

def hybrid_agent(environment,grid,start_x,start_y):
    # Grad students will need to augment this agent with a way to track the movement actions taken in sandy vs. non-sandy areas.
    sim_length = 10000
    t = 0
    total_movement_cost = 0
    while t < sim_length:
        # Get all of the facts associated with cells.
        cell_facts = []
        for fact in environment.facts():
            if fact.template == environment.find_template('cell'):
                cell_facts.append(fact)
```

```
# Get all of the facts associated with hidden cells. (Only used for sandy check)
hidden_cell_facts = []
for fact in environment.facts():
    if fact.template == environment.find_template('hidden-cell'):
        hidden_cell_facts.append(fact)


# Get all facts.
facts = list(environment.facts())
plan = []

# Get current rover conditions from KB.
current_pos = ask_current_pos(facts,environment)
x = current_pos[0]
y = current_pos[1]
destroyed = ask_rover_destroyed(facts,environment)
orientation = ask_current_orientation(facts,environment)
rover_pos = RoverPosition(x,y,orientation)

# Check if the rover has been destroyed, end the game if so.
if destroyed:
    print("Game over.")
    return 1, total_movement_cost

# Check if the agent has retrieved the sample and is back in the start position.
if ask_agent_sample(facts,environment) and x == start_x and y == start_y:
    print("You win!")
    return 0, total_movement_cost

# Get a list of safe cells that we can traverse without worry.
safe_points = list()
for cell in cell_facts:
    cell_dict = dict(cell)
    if cell_dict["safe"] == "true":
        safe_points.append([cell_dict["xloc"],cell_dict["yloc"]])

# Check if agent has retrieved the sample.
if len(plan) == 0:
    if ask_agent_sample(facts,environment):
        print("Sample acquired!")
        goals = list()
        goals.append([start_x, start_y])
        actions = plan_route(rover_pos, goals[0], safe_points, len(grid[:,0]))
        plan.extend(actions)

# Check if the agent is in a cell with the biosample. If so, drill.
if len(plan) == 0:
    if ask_cell_bio(x,y,cell_facts):
        action = "drill"
        plan.append(action)


# Here you should write code that checks for unvisited but safe spots to visit, and adds maneuvers to those spots to the plan,
# using the plan_route function. See the "not_unsafe" code below for a similar implementation.

###
### YOUR CODE HERE
if len(plan) == 0:
    safe = list()
    for cell in cell_facts:
        cell_dict = dict(cell)
        if cell_dict["safe"] == "true" and cell_dict["visited"] == "false" and not (cell_dict["xloc"] == x and cell_dict["yloc"] == y
            safe.append([cell_dict["xloc"],cell_dict["yloc"]])
    if len(safe) > 0:
        valid_temps = []
        for safe_point in safe:
            safe_points.append(safe_point)
            temp = plan_route(rover_pos, safe_point, safe_points, len(grid[:,0]))
            safe_points.remove(safe_point)
            if temp is not None:
                valid_temps.append(temp)
        if len(valid_temps) > 0:
            temp = random.choice(valid_temps)
            plan.extend(temp)
###

# Check if lidar has been done in this cell/orientation. If not, lidar.
```

```
        if len(plan) == 0:
            if not ask_if_lidar_this_loc(x,y,ask_current_orientation(facts,environment),facts,environment):
                action = "lidar"
                plan.append(action)

        # Check if traction has been measured in this cell/orientation. If not, traction.
        if len(plan) == 0:
            if ask_sandy_unknown(x,y,cell_facts):
                action = "traction"
                plan.append(action)

        # Check if air quality/nearby storm status has been measured in this cell/orientation. If not, air_quality.
        if len(plan) == 0:
            if ask_stormnearby_unknown(x,y,cell_facts):
                action = "air_quality"
                plan.append(action)

        # Check if the spectrometer has been used in this cell. If not, spectrometer.
        if len(plan) == 0:
            if ask_bio_unknown(x,y,cell_facts):
                action = "spectrometer"
                plan.append(action)

        # If there are still no safe and unvisited spots remaining and this cell has had all measurements performed, try moving to an unvisite
        # that at least is not unsafe.
        if len(plan) == 0:
            not_unsafe = list()
            for cell in cell_facts:
                cell_dict = dict(cell)
                if not cell_dict["safe"] == "false" and cell_dict["visited"] == "false" and not (cell_dict["xloc"] == x and cell_dict["yloc"]
                    not_unsafe.append([cell_dict["xloc"],cell_dict["yloc"]])
            if len(not_unsafe) > 0:
                valid_temps = []
                for not_unsafe_point in not_unsafe:
                    safe_points.append(not_unsafe_point)
                    temp = plan_route(rover_pos, not_unsafe_point, safe_points, len(grid[:,0]))
                    safe_points.remove(not_unsafe_point)
                    if temp is not None:
                        valid_temps.append(temp)
                if len(valid_temps) > 0:
                    temp = random.choice(valid_temps)
                    plan.extend(temp)

        # If there are no unknown spots to explore, all of the unvisited spots are known to be unsafe, the goal must be blocked off.
        if len(plan) == 0:
            print("No actions to take!")
            print("Game over.")
            return 1, total_movement_cost

        # Execute the first action in the plan.
        action = plan[0]

        # Translating from plan_route terminology to clips rule terminology.
        if action == "Forward":
            action = "forward"
        elif action == "Turnright":
            action = "rotate_cw"
        elif action == "Turnleft":
            action = "rotate_ccw"

        print("Current time: "+str(t))
        print("Current position: "+str(current_pos))
        print("Action taken: "+str(action))

        plan = plan[1:]

        # Update the KB with the action taken. Run the rule environment so that any new firings can occur.
        action_template = environment.find_template('action')
        action_template.assert_fact(type=action,time=t,xloc=x,yloc=y,orientation=ask_current_orientation(facts,environment))
        if action == "forward" or action == "rotate_cw" or action == "rotate_ccw":
            movement_cost = check_sandy_hidden_cell(x,y,hidden_cell_facts)
            total_movement_cost += movement_cost
        environment.run()
        t += 1
    return 1, total_movement_cost
```

```
# This code runs 100 trials and tracks the number of victories and the average movement cost.
sum = 0
sims = 100
mvmt_cost_sum = 0
for i in range(sims):
    env, grid, start_x, start_y = new_game(templates,rules)
    result, mvmt_cost = hybrid_agent(env,grid,start_x,start_y)
    sum += result
    mvmt_cost_sum += mvmt_cost
print("Number of victories: "+str(sims-sum))
print("Movement cost average: "+str(mvmt_cost_sum/sims))

# This prints out the working memory for the last trial run.
print("Final grid observations: ")
for fact in env.facts():
    if fact.template == env.find_template('cell'):
        d = dict(fact)
        exclude_keys = ['up_cell', 'down_cell','right_cell','left_cell']
        new_d = {k: d[k] for k in set(list(d.keys())) - set(exclude_keys)}
        print(", ".join([key+": "+str(value) for key, value in sorted(new_d.items(), key=lambda x: x[0])]))
```

```
Current position: [2, 0]
Action taken: forward
Sample acquired!
Current time: 64
Current position: [2, 1]
Action taken: forward
Sample acquired!
Current time: 65
Current position: [2, 2]
Action taken: forward
Sample acquired!
Current time: 66
Current position: [2, 3]
Action taken: forward
Sample acquired!
Current time: 67
Current position: [2, 4]
Action taken: forward
You win!
Number of victories: 50
Movement cost average: 53.7
Final grid observations:
biosample: unknown, rock: false, safe: true, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 58, visited: false, xl
biosample: true, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 29, visited: true, xloc: 1, y
biosample: false, rock: false, safe: true, sandy: true, storm: false, storm_nearby: false, time_checked: 24, visited: true, xloc: 2, y
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 29, visited: true, xloc: 3,
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 36, visited: true, xloc: 4,
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 41, visited: true, xloc: 5,
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 24, visited: fals
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 19, visited: true, xloc: 2,
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 24, visited: fals
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 41, visited: fals
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 46, visited: fals
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 19, visited: fals
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 14, visited: true, xloc: 2,
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 19, visited: fals
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 14, visited: fals
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 9, visited: true, xloc: 2, y
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 14, visited: fals
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 9, visited: false
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 3, visited: true, xloc: 2, y
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 9, visited: false
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 3, visited: false
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 0, visited: true, xloc: 2, y
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 3, visited: false
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: fa
biosample: unknown, rock: true, safe: false, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 68, visited: false
```

Accounts sandy squares - BFS algorithm

```
import clips
import random
import numpy as np
# from utils import *


def get_nearest_rock(grid, x, y, orientation):
    # Based on the grid, this function determines the nearest rock for lidar measurements using x, y, and orientation
    init_x = x
    init_y = y
    if orientation == "up":
        while y > 0:
            if (grid[y,x] == 2):
                return (init_y - y)
            y = y-1
    elif orientation == "down":
        while y < len(grid[:,0]):
            if (grid[y,x] == 2):
                return (y - init_y)
            y = y+1
    elif orientation == "left":
        while x > 0:
            if (grid[y,x] == 2):
                return (init_x - x)
            x = x-1
    elif orientation == "right":
        while x < len(grid[0,:]):
            if (grid[y,x] == 2):
                return (x - init_x)
            x = x+1
    return -1

def check_storm_nearby(grid, x, y):
    # Based on the grid, this function checks if there is a nearby storm (for the air quality sensor)
    if y+1 < len(grid[:,0]) and grid[y+1,x] == 4:
        return "true"
    elif y-1 >= 0 and grid[y-1,x] == 4:
        return "true"
    elif x+1 < len(grid[0,:]) and grid[y,x+1] == 4:
        return "true"
    elif x-1 >= 0 and grid[y,x-1] == 4:
        return "true"
    else:
        return "false"

def check_safe_unvisited(x,y,facts):
    # This function should check if location (x, y) is safe and unvisited by querying the KB.
    ### YOUR CODE HERE
    safe_and_unvisited = False
    for fact in facts:
      fact_dict = dict(fact)
      if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["safe"] == "true" and fact_dict["visited"] == "false":
          safe_and_unvisited = True
    ###
    return safe_and_unvisited

def check_sandy_hidden_cell(x,y,facts):
    # This function should check if location (x, y) is sandy and unvisited by querying the KB.
    movement_cost = 1
    for fact in facts:
        fact_dict = dict(fact)
        if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["sandy"] == "true":
            movement_cost = 2
    return movement_cost

def ask_bio_unknown(x,y,facts):
    # This function queries the KB to see if it is unknown if there is a biosample in location (x, y).
    unknown = False
    for fact in facts:
        fact_dict = dict(fact)
        if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["biosample"] == "unknown":
```

```
            unknown = True
    return unknown

def ask_sandy_unknown(x,y,facts):
    # This function queries the KB to see if it is unknown if there is a biosample in location (x, y).
    unknown = False
    for fact in facts:
        fact_dict = dict(fact)
        if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["sandy"] == "unknown":
            unknown = True
    return unknown

def ask_stormnearby_unknown(x,y,facts):
    # This function queries the KB to see if it is unknown if there is a storm nearby in location (x, y).
    unknown = False
    for fact in facts:
        fact_dict = dict(fact)
        if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["storm_nearby"] == "unknown":
            unknown = True
    return unknown

def ask_cell_bio(x,y,facts):
    # This function queries the KB to see if there is a biosample in location (x, y).
    ###
    ### YOUR CODE HERE
    bio = False
    for fact in facts:
      fact_dict = dict(fact)
      if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["biosample"] == "true":
        bio = True
    ###
    return bio

def ask_current_pos(facts,environment):
    # This function queries the KB to see the current position of the agent.
    for fact in facts:
        if fact.template == environment.find_template('agent'):
            agent_dict = dict(fact)
    x = agent_dict["xloc"]
    y = agent_dict["yloc"]
    return [x,y]

def ask_current_orientation(facts,environment):
    # This function queries the KB to see the current orientation of the agent.
    for fact in facts:
        if fact.template == environment.find_template('agent'):
            agent_dict = dict(fact)
    orientation = agent_dict["orientation"]
    return orientation


def ask_rover_destroyed(facts,environment):
    # This function queries the KB to see if the agent is destroyed.
    for fact in facts:
        if fact.template == environment.find_template('agent'):
            agent_dict = dict(fact)
    destroyed = agent_dict["destroyed"]
    if destroyed == "true":
        return True
    else:
        return False

def ask_agent_sample(facts,environment):
    # This function queries the KB to see if the agent has retrieved the biosample.
    for fact in facts:
        if fact.template == environment.find_template('agent'):
            agent_dict = dict(fact)
    if agent_dict["sample_retrieved"] == "true":
        return True
    else:
        return False

def ask_if_lidar_this_loc(x,y,o,facts,environment):
    # This function asks if a lidar measurement has been taken in this location/orientation yet.
    lidar_this_loc = False
    action_dicts = []
    for fact in facts:
```

```
            if fact.template == environment.find_template('action'):
                action_dict = dict(fact)
                action_dicts.append(action_dict)
        for action in action_dicts:
            if action["xloc"] == x and action["yloc"] == y and action["orientation"] == o:
                lidar_this_loc = True
        return lidar_this_loc


    def plan_route(current, goal, allowed, game_size):
        # This function tries to create a route from current to goal based on the allowed spaces provided for travel.
        problem = PlanRouteSandy(current, goal, allowed, game_size)
        search_result = breadth_first_graph_search(problem)
        ### GRAD STUDENTS REPLACE BREADTH FIRST WITH BEST FIRST OR ASTAR ###
        if search_result is not None:
            return search_result.solution()
        else:
            return None

    ### TEMPLATES ###

    cell_template = """
    (deftemplate cell
        (slot xloc)
        (slot yloc)
        (slot rock)
        (slot biosample)
        (slot storm)
        (slot storm_nearby)
        (slot sandy)
        (slot safe)
        (slot visited)
        (slot up_cell)
        (slot down_cell)
        (slot right_cell)
        (slot left_cell)
        (slot time_checked)
    )
    """

    hidden_cell_template = """
    (deftemplate hidden-cell
        (slot xloc)
        (slot yloc)
        (slot rock)
        (slot biosample)
        (slot storm)
        (slot storm_nearby)
        (slot sandy)
        (slot lidar_right)
        (slot lidar_up)
        (slot lidar_down)
        (slot lidar_left)
    )
    """

    agent_template = """
    (deftemplate agent
        (slot xloc)
        (slot yloc)
        (slot orientation)
        (slot batt_soc)
        (slot time)
        (slot sample_retrieved)
        (slot destroyed)
        (multislot loc_history)
        (multislot action_history)
    )
    """

    lidar_measurement_template = """
    (deftemplate lidar_measurement
        (slot xloc)
        (slot yloc)
        (slot orientation)
        (slot distance)
        (slot rock_xloc)
```

```
    (slot rock_yloc)
)
"""

aq_measurement_template = """
(deftemplate aq_measurement
    (slot xloc)
    (slot yloc)
    (slot air_quality)
)
"""

bio_measurement_template = """
(deftemplate bio_measurement
    (slot xloc)
    (slot yloc)
    (slot organic)
)
"""

traction_measurement_template = """
(deftemplate traction_measurement
    (slot xloc)
    (slot yloc)
    (slot traction)
)
"""

action_template = """
(deftemplate action
    (slot type)
    (slot time)
    (slot xloc)
    (slot yloc)
    (slot orientation)
)
"""
templates = [action_template,traction_measurement_template,aq_measurement_template,bio_measurement_template,lidar_measurement_template,action_
             cell_template,hidden_cell_template]

### RULES ###

lidar_rule = """
(defrule lidar_rule
    ; This rule creates a lidar_measurement fact when a lidar action is taken.
    ?act <- (action (type ?type&:(eq ?type "lidar")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?a
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (lidar_up ?lidar_up) (lidar_down ?lidar_down) (lidar_left ?li
    =>
    (if (eq ?o "up") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_up) (rock_xloc ?x) (rock_yloc (- ?y
    (if (eq ?o "down") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_down) (rock_xloc ?x) (rock_yloc
    (if (eq ?o "left") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_left) (rock_xloc (- ?x ?lidar_le
    (if (eq ?o "right") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_right) (rock_xloc (+ ?x ?lidar_r
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""
rotate_ccw_rule = """
(defrule rotate_ccw_rule
    ; This rule rotates the agent counterclockwise when a rotate_ccw action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "rotate_ccw")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    =>
    (bind ?new_o ?o)
    (if (eq ?o "up") then (bind ?new_o "left"))
    (if (eq ?o "left") then (bind ?new_o "down"))
    (if (eq ?o "down") then (bind ?new_o "right"))
    (if (eq ?o "right") then (bind ?new_o "up"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (orientation ?new_o) (time ?t) (action_history (create$ $?ah ?type)))
)
"""

rotate_cw_rule = """
(defrule rotate_cw_rule
    ; This rule rotates the agent clockwise when a rotate_cw action is taken.
```

```
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "rotate_cw")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    =>
    (bind ?new_o ?o)
    (if (eq ?o "up") then (bind ?new_o "right"))
    (if (eq ?o "right") then (bind ?new_o "down"))
    (if (eq ?o "down") then (bind ?new_o "left"))
    (if (eq ?o "left") then (bind ?new_o "up"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (orientation ?new_o) (time ?t) (action_history (create$ $?ah ?type)))
)
"""

forward_rule = """
(defrule forward_rule
    ; This rule moves the agent forward when a forward action is taken.
    ?act <- (action (type ?type&:(eq ?type "forward")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    =>
    (if (eq ?o "up") then (bind ?new_x ?x) (bind ?new_y (- ?y 1)))
    (if (eq ?o "down") then (bind ?new_x ?x) (bind ?new_y (+ ?y 1)))
    (if (eq ?o "left") then (bind ?new_x (- ?x 1)) (bind ?new_y ?y))
    (if (eq ?o "right") then (bind ?new_x (+ ?x 1)) (bind ?new_y ?y))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (xloc ?new_x) (yloc ?new_y) (action_history (create$ $?ah ?type)))
)
"""

unvisited_cell_rule = """
(defrule unvisited_cell_rule
    ; This rule sets the cell to visited if it was previously unvisited. It also checks for agent destruction.
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (loc_history $?lh) (action_history $?ah))
    ?hidden-cell <- (hidden-cell (xloc ?hcx&:(eq ?hcx ?x)) (yloc ?hcy&:(eq ?hcy ?y)) (rock ?rock) (storm ?storm))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y))  (visited "false"))
    =>
    (bind ?destroyed "false")
    (if (eq ?rock "true") then (bind ?destroyed "true"))
    (if (eq ?storm "true") then (bind ?destroyed "true"))
    (modify ?cell (rock ?rock) (storm ?storm) (visited "true") (safe "true"))
    (modify ?ag (destroyed ?destroyed))
)
"""

air_quality_rule="""
(defrule air_quality_rule
    ; This rule creates an aq_measurement when an air_quality action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "air_quality")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?ah))
    ?hidden-cell <- (hidden-cell(xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (storm_nearby ?sn))
    =>
    (if (eq ?sn "true") then (assert (aq_measurement (xloc ?x) (yloc ?y) (air_quality "high_ppm"))))
    (if (eq ?sn "false") then (assert (aq_measurement (xloc ?x) (yloc ?y) (air_quality "low_ppm"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""
bio_rule="""
(defrule bio_rule
    ; This rule creates a bio_measurement when a bio action is taken.
    ?act <- (action (type ?type&:(eq ?type "spectrometer")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?a
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (biosample ?biosample))
    =>
    (if (eq ?biosample "true") then (assert (bio_measurement (xloc ?x) (yloc ?y) (organic "true")))
     else (assert (bio_measurement (xloc ?x) (yloc ?y) (organic "false"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""

traction_rule="""
(defrule traction_rule
    ; This rule creates a traction_measurement when a traction action is taken.
    ?act <- (action (type ?type&:(eq ?type "traction")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?a
```

```
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (sandy ?sandy))
    =>
    (if (eq ?sandy "true") then (assert (traction_measurement (xloc ?x) (yloc ?y) (traction "poor")))
     else (assert (traction_measurement (xloc ?x) (yloc ?y) (traction "good"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""

storm_nearby_rule="""
(defrule storm_nearby_rule
    ; This rule checks to see if a storm is nearby a cell based on the air quality measurements.
    ?meas <- (aq_measurement (xloc ?x) (yloc ?y) (air_quality ?aq))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?sn "unknown")
    (if (eq ?aq "low_ppm") then (bind ?sn "false"))
    (if (eq ?aq "high_ppm") then (bind ?sn "true"))
    (modify ?cell (storm_nearby ?sn))
)
"""

traction_meas_rule="""
(defrule traction_meas_rule
    ; This rule checks to see if a cell is sandy based on the traction measurements.
    ?meas <- (traction_measurement (xloc ?x) (yloc ?y) (traction ?tr))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?s "unknown")
    (if (eq ?tr "good") then (bind ?s "false"))
    (if (eq ?tr "poor") then (bind ?s "true"))
    (modify ?cell (sandy ?s))
)
"""

biosample_meas_rule="""
(defrule biosample_meas_rule
    ; This rule checks to see if a cell has a biosample based on the bio measurements.
    ; YOUR CODE HERE
    ?meas <- (bio_measurement (xloc ?x) (yloc ?y) (organic ?org))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?bio "unknown")
    (if (eq ?org "true") then (bind ?bio "true"))
    (if (eq ?org "false") then (bind ?bio "false"))
    (modify ?cell (biosample ?bio))
)
"""

lidar_update_rule="""
(defrule lidar_update_rule
    ; This rule checks to see if a cell has a rock based on the lidar measurements.
    ?meas <- (lidar_measurement (distance ?d&:(neq ?d -1)) (rock_xloc ?rx) (rock_yloc ?ry))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?rx)) (yloc ?cy&:(eq ?cy ?ry)))
    =>
    (modify ?cell (rock "true") (safe "false"))
)
"""

lidar_y_clear_rule="""
(defrule lidar_y_clear_rule
    ; This cell infers that there is no rock based on lidar measurements returning a -1 value.
    ?meas <- (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o&:(or (eq ?o "up") (eq ?o "down"))) (distance ?d&:(eq ?d -1)))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy) (rock "unknown"))
    =>
    (bind ?rock "unknown")
    (if (eq ?o "up")
     then (if (< ?cy ?y)
           then (bind ?rock "false")
          )
    )
    (if (eq ?o "down")
     then (if (> ?cy ?y)
           then (bind ?rock "false")
          )
    )
    (modify ?cell (rock ?rock))
```

```
)
"""

lidar_x_clear_rule="""
(defrule lidar_x_clear_rule
    ; This cell infers that there is no rock based on lidar measurements returning a -1 value.
    ?meas <- (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o&:(or (eq ?o "left") (eq ?o "right"))) (distance ?d&:(eq ?d -1)))
    ?cell <- (cell (xloc ?cx) (yloc ?cy&:(eq ?cy ?y)) (rock "unknown"))
    =>
    (bind ?rock "unknown")
    (if (eq ?o "left")
     then (if (< ?cx ?x)
           then (bind ?rock "false")
          )
    )
    (if (eq ?o "right")
     then (if (> ?cx ?x)
           then (bind ?rock "false")
          )
    )
    (modify ?cell (rock ?rock))
)
"""

drill_rule="""
(defrule drill_rule
    ; This rule retrieves a sample if a drill action is taken in a cell with a biosample.
    ?act <- (action (type ?type&:(eq ?type "drill")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (biosample ?bio))
    =>
    (bind ?sample "false")
    (if (eq ?bio "true") then (bind ?sample "true"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (sample_retrieved ?sample) (time ?t) (action_history (create$ $?ah ?type)))
)
"""

up_cell_rule="""
(defrule up_cell_rule
    ; This rule sets the cell that is above the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (up_cell nil))
    ?up-cell <- (cell (xloc ?cx&:(eq ?x ?cx)) (yloc ?cy&:(eq (- ?y 1) ?cy)))
    =>
    (modify ?cell (up_cell $?up-cell))
)
"""

down_cell_rule="""
(defrule down_cell_rule
    ; This rule sets the cell that is below the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (down_cell nil))
    ?down-cell <- (cell (xloc ?cx&:(eq ?x ?cx)) (yloc ?cy&:(eq (+ ?y 1) ?cy)))
    =>
    (modify ?cell (down_cell ?down-cell))
)
"""

left_cell_rule="""
(defrule left_cell_rule
    ; This rule sets the cell that is to the left of the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (left_cell nil))
    ?left-cell <- (cell (xloc ?cx&:(eq (- ?x 1) ?cx)) (yloc ?cy&:(eq ?y ?cy)))
    =>
    (modify ?cell (left_cell ?left-cell))
)
"""

right_cell_rule="""
(defrule right_cell_rule
    ; This rule sets the cell that is to the right of the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (right_cell nil))
    ?right-cell <- (cell (xloc ?cx&:(eq (+ ?x 1) ?cx)) (yloc ?cy&:(eq ?y ?cy)))
    =>
    (modify ?cell (right_cell ?right-cell))
)
"""
```

```
storm_safe_rule="""
(defrule storm_safe_rule
    ; This rule infers that there is no storm in a cell based on two adjacent conflicting air quality measurements.
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time) (loc_history $?lh) (action_history $?ah))
    ?cell <- (cell (xloc ?cx) (yloc ?cy) (right_cell ?rc) (left_cell ?lc) (up_cell ?uc) (down_cell ?dc) (storm "unknown") (time_checked ?t&:(
    =>
    (bind ?storm "unknown")
    (bind ?safe "unknown")
    (bind ?lcsnb "unknown")
    (bind ?rcsnb "unknown")
    (bind ?ucsnb "unknown")
    (bind ?dcsnb "unknown")
    (if (neq ?lc nil)
     then (if (eq (fact-slot-value ?lc storm_nearby) "false")
           then (bind ?lcsnb "false")
          )
    )
    (if (neq ?rc nil)
     then (if (eq (fact-slot-value ?rc storm_nearby) "false")
           then (bind ?rcsnb "false")
          )
    )
    (if (neq ?uc nil)
     then (if (eq (fact-slot-value ?uc storm_nearby) "false")
           then (bind ?ucsnb "false")
          )
    )
    (if (neq ?dc nil)
     then (if (eq (fact-slot-value ?dc storm_nearby) "false")
           then (bind ?dcsnb "false")
          )
    )
    (if (or (eq ?lcsnb "false") (eq ?rcsnb "false") (eq ?ucsnb "false") (eq ?dcsnb "false"))
     then (bind ?storm "false")
    )
    (bind ?new_t ?agent_time)
    (modify ?cell (storm ?storm) (time_checked ?new_t))
)
"""

safe_cell_rule="""
(defrule safe_cell_rule
    ; This rule checks if the cell has no rock or storm and if so, sets the cell to safe.
    ; YOUR CODE HERE
    ?cell <- (cell (rock ?r) (storm ?s) (safe "unknown"))
    =>
    (bind ?safe "unknown")
    (if (and (eq ?r "false") (eq ?s "false")) then (bind ?safe "true"))
    (modify ?cell (safe ?safe))
)
"""
rules = [lidar_rule,rotate_ccw_rule,rotate_cw_rule,forward_rule,air_quality_rule,traction_rule,bio_rule,storm_nearby_rule,traction_meas_rule,l
         unvisited_cell_rule,lidar_update_rule,drill_rule,left_cell_rule,right_cell_rule,up_cell_rule,down_cell_rule,storm_safe_rule,lidar_x_
         safe_cell_rule]


def new_game(templates,rules):
    # This function creates a new game, which consists of a random 6x6 grid with 3 rocks, 3 sandy cells, 2 storms, 1 goal and 1 start cell.
    # It also loads all of the rules and templates into CLIPS.
    environment = clips.Environment()
    for template in templates:
        environment.build(template)
    for rule in rules:
        environment.build(rule)

    # Generate random grid.
    random_grid = np.zeros((6,6))
    stuff_cells = np.random.choice(random_grid.size, 10, replace=False) # 10 = 3 rocks + 3 sandy + 2 storms + 1 goal + 1 start
    # 1 is start, 2 is rocks, 3 is sandy, 4 is storm, 5 is organic sample
    random_grid.ravel()[stuff_cells[0]] = 1
    random_grid.ravel()[stuff_cells[1:4]] = 2
    random_grid.ravel()[stuff_cells[4:7]] = 3
    random_grid.ravel()[stuff_cells[7:9]] = 4
    random_grid.ravel()[stuff_cells[9]] = 5
    start_x = None
    start_y = None
```

```
    print(random_grid)

    # populate hidden cells (the real environment if the problem was fully observable)
    for i in range(len(random_grid[:,0])):
        for j in range(len(random_grid[0,:])):
            x = j
            y = i
            # row i, column j
            # print("Row "+str(i)+", column "+str(j)+" is "+str(random_grid[i,j]))
            lidar_up = get_nearest_rock(random_grid,x,y,"up")
            lidar_down = get_nearest_rock(random_grid,x,y,"down")
            lidar_left = get_nearest_rock(random_grid,x,y,"left")
            lidar_right = get_nearest_rock(random_grid,x,y,"right")
            storm_nearby = check_storm_nearby(random_grid,x,y)

            if random_grid[y,x] == 4:
                storm = "true"
            else:
                storm = "false"
            if random_grid[y,x] == 3:
                sandy = "true"
            else:
                sandy = "false"
            if random_grid[y,x] == 2:
                rock = "true"
            else:
                rock = "false"
            if random_grid[y,x] == 5:
                biosample = "true"
            else:
                biosample = "false"
            if random_grid[y,x] == 1:
                start_x = x
                start_y = y
            # assert a new fact through its template
            hidden_cell_template = environment.find_template('hidden-cell')
            fact = hidden_cell_template.assert_fact(xloc=x,
                                        yloc=y,
                                        rock=rock,
                                        biosample=biosample,
                                        storm=storm,
                                        storm_nearby=storm_nearby,
                                        sandy=sandy,
                                        lidar_up=lidar_up,
                                        lidar_down=lidar_down,
                                        lidar_left=lidar_left,
                                        lidar_right=lidar_right
                                        )

    # populate agent's cell KB (what the agent knows)
    for i in range(len(random_grid[:,0])):
        for j in range(len(random_grid[0,:])):
            x = j
            y = i
            if not (x == start_x and y == start_y):
                storm = "unknown"
                storm_nearby = "unknown"
                sandy = "unknown"
                rock = "unknown"
                safe = "unknown"
                biosample = "unknown"
                visited = "false"
            else:
                storm = "false"
                sandy = "false"
                storm_nearby = "unknown"
                rock = "false"
                biosample = "false"
                safe = "true"
                visited = "true"
            cell_template = environment.find_template('cell')
            cell_fact = cell_template.assert_fact(xloc=x,
                                        yloc=y,
                                        rock=rock,
                                        biosample=biosample,
                                        storm=storm,
                                        storm_nearby=storm_nearby,
```

```
                                        sandy=sandy,
                                        safe=safe,
                                        visited=visited,
                                        time_checked=0
                                        )

    # Populate initial agent state.
    agent_template = environment.find_template('agent')
    agent = agent_template.assert_fact(xloc=start_x,
                                       yloc=start_y,
                                       orientation="right",
                                       batt_soc=1,
                                       time=0,
                                       sample_retrieved="false",
                                       destroyed="false",
                                       loc_history=[],
                                       action_history=[]
                                       )
    return environment, random_grid, start_x, start_y

def hybrid_agent(environment,grid,start_x,start_y):
    # Grad students will need to augment this agent with a way to track the movement actions taken in sandy vs. non-sandy areas.
    sim_length = 10000
    t = 0
    total_movement_cost = 0
    while t < sim_length:
        # Get all of the facts associated with cells.
        cell_facts = []
        for fact in environment.facts():
            if fact.template == environment.find_template('cell'):
                cell_facts.append(fact)
        # Get all of the facts associated with hidden cells. (Only used for sandy check)
        hidden_cell_facts = []
        for fact in environment.facts():
            if fact.template == environment.find_template('hidden-cell'):
                hidden_cell_facts.append(fact)

        sandy_cells = []
        for fact in hidden_cell_facts:
          fact_dict = dict(fact)

          if fact_dict["sandy"] == "true":
            sandy_cells.append([fact_dict["xloc"], fact_dict["yloc"]])

        # Get all facts.
        facts = list(environment.facts())
        plan = []

        # Get current rover conditions from KB.
        current_pos = ask_current_pos(facts,environment)
        x = current_pos[0]
        y = current_pos[1]
        destroyed = ask_rover_destroyed(facts,environment)
        orientation = ask_current_orientation(facts,environment)
        rover_pos = RoverPosition(x,y,orientation)

        # Check if the rover has been destroyed, end the game if so.
        if destroyed:
            print("Game over.")
            return 1, total_movement_cost

        # Check if the agent has retrieved the sample and is back in the start position.
        if ask_agent_sample(facts,environment) and x == start_x and y == start_y:
            print("You win!")
            return 0, total_movement_cost

        # Get a list of safe cells that we can traverse without worry.
        safe_points = list()
        for cell in cell_facts:
            cell_dict = dict(cell)
            if cell_dict["safe"] == "true":
                safe_points.append([cell_dict["xloc"],cell_dict["yloc"]])

        # Check if agent has retrieved the sample.
        if len(plan) == 0:
            if ask_agent_sample(facts,environment):
                print("Sample acquired!")
```

```
        goals = list()
        goals.append([start_x, start_y])
        actions = plan_route(rover_pos, goals[0], safe_points, len(grid[:,0]))
        plan.extend(actions)

# Check if the agent is in a cell with the biosample. If so, drill.
if len(plan) == 0:
    if ask_cell_bio(x,y,cell_facts):
        action = "drill"
        plan.append(action)


# Here you should write code that checks for unvisited but safe spots to visit, and adds maneuvers to those spots to the plan,
# using the plan_route function. See the "not_unsafe" code below for a similar implementation.

###
### YOUR CODE HERE
if len(plan) == 0:
    safe = list()
    for cell in cell_facts:
        cell_dict = dict(cell)
        if cell_dict["safe"] == "true" and cell_dict["visited"] == "false" and not (cell_dict["xloc"] == x and cell_dict["yloc"] == y
            safe.append([cell_dict["xloc"],cell_dict["yloc"]])
    if len(safe) > 0:
        valid_temps = []
        for safe_point in safe:
            safe_points.append(safe_point)
            temp = plan_route(rover_pos, safe_point, safe_points, len(grid[:,0]))
            safe_points.remove(safe_point)
            if temp is not None:
                valid_temps.append(temp)
        if len(valid_temps) > 0:
            temp = random.choice(valid_temps)
            plan.extend(temp)
###

# Check if lidar has been done in this cell/orientation. If not, lidar.
if len(plan) == 0:
    if not ask_if_lidar_this_loc(x,y,ask_current_orientation(facts,environment),facts,environment):
        action = "lidar"
        plan.append(action)

# Check if traction has been measured in this cell/orientation. If not, traction.
if len(plan) == 0:
    if ask_sandy_unknown(x,y,cell_facts):
        action = "traction"
        plan.append(action)

# Check if air quality/nearby storm status has been measured in this cell/orientation. If not, air_quality.
if len(plan) == 0:
    if ask_stormnearby_unknown(x,y,cell_facts):
        action = "air_quality"
        plan.append(action)

# Check if the spectrometer has been used in this cell. If not, spectrometer.
if len(plan) == 0:
    if ask_bio_unknown(x,y,cell_facts):
        action = "spectrometer"
        plan.append(action)

# If there are still no safe and unvisited spots remaining and this cell has had all measurements performed, try moving to an unvisite
# that at least is not unsafe.
if len(plan) == 0:
    not_unsafe = list()
    for cell in cell_facts:
        cell_dict = dict(cell)
        if not cell_dict["safe"] == "false" and cell_dict["visited"] == "false" and not (cell_dict["xloc"] == x and cell_dict["yloc"]
            not_unsafe.append([cell_dict["xloc"],cell_dict["yloc"]])
    if len(not_unsafe) > 0:
        valid_temps = []
        for not_unsafe_point in not_unsafe:
            safe_points.append(not_unsafe_point)
            temp = plan_route(rover_pos, not_unsafe_point, safe_points, len(grid[:,0]))
            safe_points.remove(not_unsafe_point)
            if temp is not None:
                valid_temps.append(temp)
        if len(valid_temps) > 0:
```

```
                    temp = random.choice(valid_temps)
                    plan.extend(temp)

        # If there are no unknown spots to explore, all of the unvisited spots are known to be unsafe, the goal must be blocked off.
        if len(plan) == 0:
            print("No actions to take!")
            print("Game over.")
            return 1, total_movement_cost

        # Execute the first action in the plan.
        action = plan[0]

        # Translating from plan_route terminology to clips rule terminology.
        if action == "Forward":
            action = "forward"
        elif action == "Turnright":
            action = "rotate_cw"
        elif action == "Turnleft":
            action = "rotate_ccw"

        print("Current time: "+str(t))
        print("Current position: "+str(current_pos))
        print("Action taken: "+str(action))

        plan = plan[1:]

        # Update the KB with the action taken. Run the rule environment so that any new firings can occur.
        action_template = environment.find_template('action')
        action_template.assert_fact(type=action,time=t,xloc=x,yloc=y,orientation=ask_current_orientation(facts,environment))
        if action == "forward" or action == "rotate_cw" or action == "rotate_ccw":
            movement_cost = check_sandy_hidden_cell(x,y,hidden_cell_facts)
            total_movement_cost += movement_cost
        environment.run()
        t += 1
    return 1, total_movement_cost


# This code runs 100 trials and tracks the number of victories and the average movement cost.
sum = 0
sims = 100
mvmt_cost_sum = 0
for i in range(sims):
    env, grid, start_x, start_y = new_game(templates,rules)
    result, mvmt_cost = hybrid_agent(env,grid,start_x,start_y)
    sum += result
    mvmt_cost_sum += mvmt_cost
print("Number of victories: "+str(sims-sum))
print("Movement cost average: "+str(mvmt_cost_sum/sims))

# This prints out the working memory for the last trial run.
print("Final grid observations: ")
for fact in env.facts():
    if fact.template == env.find_template('cell'):
        d = dict(fact)
        exclude_keys = ['up_cell', 'down_cell','right_cell','left_cell']
        new_d = {k: d[k] for k in set(list(d.keys())) - set(exclude_keys)}
        print(", ".join([key+": "+str(value) for key, value in sorted(new_d.items(), key=lambda x: x[0])]))
```

```
Final grid observations:
biosample: unknown, rock: false, safe: true, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 28, visited: false, xl
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: true, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 23, visited: true, xloc: 0, y
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 28, visited: fals
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 16, visited: true, xloc: 0,
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 11, visited: fals
biosample: unknown, rock: true, safe: false, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 3, visited: false, xlo
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 11, visited: true, xloc: 0,
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 3, visited: true, xloc: 1, y
biosample: false, rock: false, safe: true, sandy: false, storm: false, storm_nearby: false, time_checked: 0, visited: true, xloc: 2, y
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 3, visited: false
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: true, safe: false, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: false,
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 16, visited: fals
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 11, visited: fals
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: false, storm_nearby: unknown, time_checked: 3, visited: false
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
biosample: unknown, rock: unknown, safe: unknown, sandy: unknown, storm: unknown, storm_nearby: unknown, time_checked: 36, visited: fa
```

Accounts sandy squares check - A* algorithm

```
import clips
import random
import numpy as np
# from utils import *


def get_nearest_rock(grid, x, y, orientation):
    # Based on the grid, this function determines the nearest rock for lidar measurements using x, y, and orientation
    init_x = x
    init_y = y
    if orientation == "up":
        while y > 0:
            if (grid[y,x] == 2):
                return (init_y - y)
            y = y-1
    elif orientation == "down":
        while y < len(grid[:,0]):
            if (grid[y,x] == 2):
                return (y - init_y)
            y = y+1
    elif orientation == "left":
        while x > 0:
            if (grid[y,x] == 2):
                return (init_x - x)
            x = x-1
    elif orientation == "right":
        while x < len(grid[0,:]):
            if (grid[y,x] == 2):
                return (x - init_x)
            x = x+1
    return -1

def check_storm_nearby(grid, x, y):
    # Based on the grid, this function checks if there is a nearby storm (for the air quality sensor)
```

```
        if y+1 < len(grid[:,0]) and grid[y+1,x] == 4:
            return "true"
        elif y-1 >= 0 and grid[y-1,x] == 4:
            return "true"
        elif x+1 < len(grid[0,:]) and grid[y,x+1] == 4:
            return "true"
        elif x-1 >= 0 and grid[y,x-1] == 4:
            return "true"
        else:
            return "false"

    def check_safe_unvisited(x,y,facts):
        # This function should check if location (x, y) is safe and unvisited by querying the KB.
        ### YOUR CODE HERE
        safe_and_unvisited = False
        for fact in facts:
          fact_dict = dict(fact)
          if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["safe"] == "true" and fact_dict["visited"] == "false":
              safe_and_unvisited = True
        ###
        return safe_and_unvisited

    def check_sandy_hidden_cell(x,y,facts):
        # This function should check if location (x, y) is sandy and unvisited by querying the KB.
        movement_cost = 1
        for fact in facts:
            fact_dict = dict(fact)
            if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["sandy"] == "true":
                movement_cost = 2
        return movement_cost

    def ask_bio_unknown(x,y,facts):
        # This function queries the KB to see if it is unknown if there is a biosample in location (x, y).
        unknown = False
        for fact in facts:
            fact_dict = dict(fact)
            if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["biosample"] == "unknown":
                unknown = True
        return unknown

    def ask_sandy_unknown(x,y,facts):
        # This function queries the KB to see if it is unknown if there is a biosample in location (x, y).
        unknown = False
        for fact in facts:
            fact_dict = dict(fact)
            if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["sandy"] == "unknown":
                unknown = True
        return unknown

    def ask_stormnearby_unknown(x,y,facts):
        # This function queries the KB to see if it is unknown if there is a storm nearby in location (x, y).
        unknown = False
        for fact in facts:
            fact_dict = dict(fact)
            if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["storm_nearby"] == "unknown":
                unknown = True
        return unknown

    def ask_cell_bio(x,y,facts):
        # This function queries the KB to see if there is a biosample in location (x, y).
        ###
        ### YOUR CODE HERE
        bio = False
        for fact in facts:
          fact_dict = dict(fact)
          if fact_dict["xloc"] == x and fact_dict["yloc"] == y and fact_dict["biosample"] == "true":
            bio = True
        ###
        return bio

    def ask_current_pos(facts,environment):
        # This function queries the KB to see the current position of the agent.
        for fact in facts:
            if fact.template == environment.find_template('agent'):
                agent_dict = dict(fact)
        x = agent_dict["xloc"]
        y = agent_dict["yloc"]
```

```
        return [x,y]

    def ask_current_orientation(facts,environment):
        # This function queries the KB to see the current orientation of the agent.
        for fact in facts:
            if fact.template == environment.find_template('agent'):
                agent_dict = dict(fact)
        orientation = agent_dict["orientation"]
        return orientation


    def ask_rover_destroyed(facts,environment):
        # This function queries the KB to see if the agent is destroyed.
        for fact in facts:
            if fact.template == environment.find_template('agent'):
                agent_dict = dict(fact)
        destroyed = agent_dict["destroyed"]
        if destroyed == "true":
            return True
        else:
            return False

    def ask_agent_sample(facts,environment):
        # This function queries the KB to see if the agent has retrieved the biosample.
        for fact in facts:
            if fact.template == environment.find_template('agent'):
                agent_dict = dict(fact)
        if agent_dict["sample_retrieved"] == "true":
            return True
        else:
            return False

    def ask_if_lidar_this_loc(x,y,o,facts,environment):
        # This function asks if a lidar measurement has been taken in this location/orientation yet.
        lidar_this_loc = False
        action_dicts = []
        for fact in facts:
            if fact.template == environment.find_template('action'):
                action_dict = dict(fact)
                action_dicts.append(action_dict)
        for action in action_dicts:
            if action["xloc"] == x and action["yloc"] == y and action["orientation"] == o:
                lidar_this_loc = True
        return lidar_this_loc


    def plan_route(current, goal, allowed, game_size):
        # This function tries to create a route from current to goal based on the allowed spaces provided for travel.
        problem = PlanRouteSandy(current, goal, allowed, game_size)
        search_result = astar_search(problem)
        ### GRAD STUDENTS REPLACE BREADTH FIRST WITH BEST FIRST OR ASTAR ###
        if search_result is not None:
            return search_result.solution()
        else:
            return None

    ### TEMPLATES ###

    cell_template = """
    (deftemplate cell
        (slot xloc)
        (slot yloc)
        (slot rock)
        (slot biosample)
        (slot storm)
        (slot storm_nearby)
        (slot sandy)
        (slot safe)
        (slot visited)
        (slot up_cell)
        (slot down_cell)
        (slot right_cell)
        (slot left_cell)
        (slot time_checked)
    )
    """
```

```
hidden_cell_template = """
(deftemplate hidden-cell
    (slot xloc)
    (slot yloc)
    (slot rock)
    (slot biosample)
    (slot storm)
    (slot storm_nearby)
    (slot sandy)
    (slot lidar_right)
    (slot lidar_up)
    (slot lidar_down)
    (slot lidar_left)
)
"""

agent_template = """
(deftemplate agent
    (slot xloc)
    (slot yloc)
    (slot orientation)
    (slot batt_soc)
    (slot time)
    (slot sample_retrieved)
    (slot destroyed)
    (multislot loc_history)
    (multislot action_history)
)
"""

lidar_measurement_template = """
(deftemplate lidar_measurement
    (slot xloc)
    (slot yloc)
    (slot orientation)
    (slot distance)
    (slot rock_xloc)
    (slot rock_yloc)
)
"""

aq_measurement_template = """
(deftemplate aq_measurement
    (slot xloc)
    (slot yloc)
    (slot air_quality)
)
"""

bio_measurement_template = """
(deftemplate bio_measurement
    (slot xloc)
    (slot yloc)
    (slot organic)
)
"""

traction_measurement_template = """
(deftemplate traction_measurement
    (slot xloc)
    (slot yloc)
    (slot traction)
)
"""

action_template = """
(deftemplate action
    (slot type)
    (slot time)
    (slot xloc)
    (slot yloc)
    (slot orientation)
)
"""
templates = [action_template,traction_measurement_template,aq_measurement_template,bio_measurement_template,lidar_measurement_template,action_
             cell_template,hidden_cell_template]
```

```
### RULES ###

lidar_rule = """
(defrule lidar_rule
    ; This rule creates a lidar_measurement fact when a lidar action is taken.
    ?act <- (action (type ?type&:(eq ?type "lidar")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?a
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (lidar_up ?lidar_up) (lidar_down ?lidar_down) (lidar_left ?l
    =>
    (if (eq ?o "up") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_up) (rock_xloc ?x) (rock_yloc (- ?y
    (if (eq ?o "down") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_down) (rock_xloc ?x) (rock_yloc
    (if (eq ?o "left") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_left) (rock_xloc (- ?x ?lidar_le
    (if (eq ?o "right") then (assert (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o) (distance ?lidar_right) (rock_xloc (+ ?x ?lidar_r
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""
rotate_ccw_rule = """
(defrule rotate_ccw_rule
    ; This rule rotates the agent counterclockwise when a rotate_ccw action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "rotate_ccw")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    =>
    (bind ?new_o ?o)
    (if (eq ?o "up") then (bind ?new_o "left"))
    (if (eq ?o "left") then (bind ?new_o "down"))
    (if (eq ?o "down") then (bind ?new_o "right"))
    (if (eq ?o "right") then (bind ?new_o "up"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (orientation ?new_o) (time ?t) (action_history (create$ $?ah ?type)))
)
"""

rotate_cw_rule = """
(defrule rotate_cw_rule
    ; This rule rotates the agent clockwise when a rotate_cw action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "rotate_cw")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    =>
    (bind ?new_o ?o)
    (if (eq ?o "up") then (bind ?new_o "right"))
    (if (eq ?o "right") then (bind ?new_o "down"))
    (if (eq ?o "down") then (bind ?new_o "left"))
    (if (eq ?o "left") then (bind ?new_o "up"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (orientation ?new_o) (time ?t) (action_history (create$ $?ah ?type)))
)
"""

forward_rule = """
(defrule forward_rule
    ; This rule moves the agent forward when a forward action is taken.
    ?act <- (action (type ?type&:(eq ?type "forward")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    =>
    (if (eq ?o "up") then (bind ?new_x ?x) (bind ?new_y (- ?y 1)))
    (if (eq ?o "down") then (bind ?new_x ?x) (bind ?new_y (+ ?y 1)))
    (if (eq ?o "left") then (bind ?new_x (- ?x 1)) (bind ?new_y ?y))
    (if (eq ?o "right") then (bind ?new_x (+ ?x 1)) (bind ?new_y ?y))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (xloc ?new_x) (yloc ?new_y) (action_history (create$ $?ah ?type)))
)
"""

unvisited_cell_rule = """
(defrule unvisited_cell_rule
    ; This rule sets the cell to visited if it was previously unvisited. It also checks for agent destruction.
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (loc_history $?lh) (action_history $?ah))
    ?hidden-cell <- (hidden-cell (xloc ?hcx&:(eq ?hcx ?x)) (yloc ?hcy&:(eq ?hcy ?y)) (rock ?rock) (storm ?storm))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y))  (visited "false"))
    =>
    (bind ?destroyed "false")
    (if (eq ?rock "true") then (bind ?destroyed "true"))
    (if (eq ?storm "true") then (bind ?destroyed "true"))
    (modify ?cell (rock ?rock) (storm ?storm) (visited "true") (safe "true"))
```

```
    (modify ?ag (destroyed ?destroyed))
)
"""

air_quality_rule="""
(defrule air_quality_rule
    ; This rule creates an aq_measurement when an air_quality action is taken.
    ; YOUR CODE HERE
    ?act <- (action (type ?type&:(eq ?type "air_quality")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?ah))
    ?hidden-cell <- (hidden-cell(xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (storm_nearby ?sn))
    =>
    (if (eq ?sn "true") then (assert (aq_measurement (xloc ?x) (yloc ?y) (air_quality "high_ppm"))))
    (if (eq ?sn "false") then (assert (aq_measurement (xloc ?x) (yloc ?y) (air_quality "low_ppm"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""
bio_rule="""
(defrule bio_rule
    ; This rule creates a bio_measurement when a bio action is taken.
    ?act <- (action (type ?type&:(eq ?type "spectrometer")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?a
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (biosample ?biosample))
    =>
    (if (eq ?biosample "true") then (assert (bio_measurement (xloc ?x) (yloc ?y) (organic "true")))
     else (assert (bio_measurement (xloc ?x) (yloc ?y) (organic "false"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""

traction_rule="""
(defrule traction_rule
    ; This rule creates a traction_measurement when a traction action is taken.
    ?act <- (action (type ?type&:(eq ?type "traction")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time))(loc_history $?lh) (action_history $?a
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (sandy ?sandy))
    =>
    (if (eq ?sandy "true") then (assert (traction_measurement (xloc ?x) (yloc ?y) (traction "poor")))
     else (assert (traction_measurement (xloc ?x) (yloc ?y) (traction "good"))))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (time ?t) (action_history (create$ $?ah ?type)))
)
"""

storm_nearby_rule="""
(defrule storm_nearby_rule
    ; This rule checks to see if a storm is nearby a cell based on the air quality measurements.
    ?meas <- (aq_measurement (xloc ?x) (yloc ?y) (air_quality ?aq))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?sn "unknown")
    (if (eq ?aq "low_ppm") then (bind ?sn "false"))
    (if (eq ?aq "high_ppm") then (bind ?sn "true"))
    (modify ?cell (storm_nearby ?sn))
)
"""

traction_meas_rule="""
(defrule traction_meas_rule
    ; This rule checks to see if a cell is sandy based on the traction measurements.
    ?meas <- (traction_measurement (xloc ?x) (yloc ?y) (traction ?tr))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?s "unknown")
    (if (eq ?tr "good") then (bind ?s "false"))
    (if (eq ?tr "poor") then (bind ?s "true"))
    (modify ?cell (sandy ?s))
)
"""

biosample_meas_rule="""
(defrule biosample_meas_rule
    ; This rule checks to see if a cell has a biosample based on the bio measurements.
    ; YOUR CODE HERE
    ?meas <- (bio_measurement (xloc ?x) (yloc ?y) (organic ?org))
```

```
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)))
    =>
    (bind ?bio "unknown")
    (if (eq ?org "true") then (bind ?bio "true"))
    (if (eq ?org "false") then (bind ?bio "false"))
    (modify ?cell (biosample ?bio))
)
"""

lidar_update_rule="""
(defrule lidar_update_rule
    ; This rule checks to see if a cell has a rock based on the lidar measurements.
    ?meas <- (lidar_measurement (distance ?d&:(neq ?d -1)) (rock_xloc ?rx) (rock_yloc ?ry))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?rx)) (yloc ?cy&:(eq ?cy ?ry)))
    =>
    (modify ?cell (rock "true") (safe "false"))
)
"""

lidar_y_clear_rule="""
(defrule lidar_y_clear_rule
    ; This cell infers that there is no rock based on lidar measurements returning a -1 value.
    ?meas <- (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o&:(or (eq ?o "up") (eq ?o "down"))) (distance ?d&:(eq ?d -1)))
    ?cell <- (cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy) (rock "unknown"))
    =>
    (bind ?rock "unknown")
    (if (eq ?o "up")
     then (if (< ?cy ?y)
           then (bind ?rock "false")
          )
    )
    (if (eq ?o "down")
     then (if (> ?cy ?y)
           then (bind ?rock "false")
          )
    )
    (modify ?cell (rock ?rock))
)
"""

lidar_x_clear_rule="""
(defrule lidar_x_clear_rule
    ; This cell infers that there is no rock based on lidar measurements returning a -1 value.
    ?meas <- (lidar_measurement (xloc ?x) (yloc ?y) (orientation ?o&:(or (eq ?o "left") (eq ?o "right"))) (distance ?d&:(eq ?d -1)))
    ?cell <- (cell (xloc ?cx) (yloc ?cy&:(eq ?cy ?y)) (rock "unknown"))
    =>
    (bind ?rock "unknown")
    (if (eq ?o "left")
     then (if (< ?cx ?x)
           then (bind ?rock "false")
          )
    )
    (if (eq ?o "right")
     then (if (> ?cx ?x)
           then (bind ?rock "false")
          )
    )
    (modify ?cell (rock ?rock))
)
"""

drill_rule="""
(defrule drill_rule
    ; This rule retrieves a sample if a drill action is taken in a cell with a biosample.
    ?act <- (action (type ?type&:(eq ?type "drill")) (time ?action_time))
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time&:(eq ?agent_time ?action_time)) (loc_history $?lh) (action_history $
    ?hidden-cell <- (hidden-cell (xloc ?cx&:(eq ?cx ?x)) (yloc ?cy&:(eq ?cy ?y)) (biosample ?bio))
    =>
    (bind ?sample "false")
    (if (eq ?bio "true") then (bind ?sample "true"))
    (bind ?t (+ ?agent_time 1))
    (modify ?ag (sample_retrieved ?sample) (time ?t) (action_history (create$ $?ah ?type)))
)
"""

up_cell_rule="""
(defrule up_cell_rule
```

```
    ; This rule sets the cell that is above the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (up_cell nil))
    ?up-cell <- (cell (xloc ?cx&:(eq ?x ?cx)) (yloc ?cy&:(eq (- ?y 1) ?cy)))
    =>
    (modify ?cell (up_cell $?up-cell))
)
"""

down_cell_rule="""
(defrule down_cell_rule
    ; This rule sets the cell that is below the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (down_cell nil))
    ?down-cell <- (cell (xloc ?cx&:(eq ?x ?cx)) (yloc ?cy&:(eq (+ ?y 1) ?cy)))
    =>
    (modify ?cell (down_cell ?down-cell))
)
"""

left_cell_rule="""
(defrule left_cell_rule
    ; This rule sets the cell that is to the left of the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (left_cell nil))
    ?left-cell <- (cell (xloc ?cx&:(eq (- ?x 1) ?cx)) (yloc ?cy&:(eq ?y ?cy)))
    =>
    (modify ?cell (left_cell ?left-cell))
)
"""

right_cell_rule="""
(defrule right_cell_rule
    ; This rule sets the cell that is to the right of the cell at location (x, y).
    ?cell <- (cell (xloc ?x) (yloc ?y) (right_cell nil))
    ?right-cell <- (cell (xloc ?cx&:(eq (+ ?x 1) ?cx)) (yloc ?cy&:(eq ?y ?cy)))
    =>
    (modify ?cell (right_cell ?right-cell))
)
"""

storm_safe_rule="""
(defrule storm_safe_rule
    ; This rule infers that there is no storm in a cell based on two adjacent conflicting air quality measurements.
    ?ag <- (agent (xloc ?x) (yloc ?y) (orientation ?o) (time ?agent_time) (loc_history $?lh) (action_history $?ah))
    ?cell <- (cell (xloc ?cx) (yloc ?cy) (right_cell ?rc) (left_cell ?lc) (up_cell ?uc) (down_cell ?dc) (storm "unknown") (time_checked ?t&:(
    =>
    (bind ?storm "unknown")
    (bind ?safe "unknown")
    (bind ?lcsnb "unknown")
    (bind ?rcsnb "unknown")
    (bind ?ucsnb "unknown")
    (bind ?dcsnb "unknown")
    (if (neq ?lc nil)
     then (if (eq (fact-slot-value ?lc storm_nearby) "false")
           then (bind ?lcsnb "false")
          )
    )
    (if (neq ?rc nil)
     then (if (eq (fact-slot-value ?rc storm_nearby) "false")
           then (bind ?rcsnb "false")
          )
    )
    (if (neq ?uc nil)
     then (if (eq (fact-slot-value ?uc storm_nearby) "false")
           then (bind ?ucsnb "false")
          )
    )
    (if (neq ?dc nil)
     then (if (eq (fact-slot-value ?dc storm_nearby) "false")
           then (bind ?dcsnb "false")
          )
    )
    (if (or (eq ?lcsnb "false") (eq ?rcsnb "false") (eq ?ucsnb "false") (eq ?dcsnb "false"))
     then (bind ?storm "false")
    )
    (bind ?new_t ?agent_time)
    (modify ?cell (storm ?storm) (time_checked ?new_t))
)
"""
```

```
safe_cell_rule="""
(defrule safe_cell_rule
    ; This rule checks if the cell has no rock or storm and if so, sets the cell to safe.
    ; YOUR CODE HERE
    ?cell <- (cell (rock ?r) (storm ?s) (safe "unknown"))
    =>
    (bind ?safe "unknown")
    (if (and (eq ?r "false") (eq ?s "false")) then (bind ?safe "true"))
    (modify ?cell (safe ?safe))
)
"""
rules = [lidar_rule,rotate_ccw_rule,rotate_cw_rule,forward_rule,air_quality_rule,traction_rule,bio_rule,storm_nearby_rule,traction_meas_rule,l
         unvisited_cell_rule,lidar_update_rule,drill_rule,left_cell_rule,right_cell_rule,up_cell_rule,down_cell_rule,storm_safe_rule,lidar_x_
         safe_cell_rule]


def new_game(templates,rules):
    # This function creates a new game, which consists of a random 6x6 grid with 3 rocks, 3 sandy cells, 2 storms, 1 goal and 1 start cell.
    # It also loads all of the rules and templates into CLIPS.
    environment = clips.Environment()
    for template in templates:
        environment.build(template)
    for rule in rules:
        environment.build(rule)

    # Generate random grid.
    random_grid = np.zeros((6,6))
    stuff_cells = np.random.choice(random_grid.size, 10, replace=False) # 10 = 3 rocks + 3 sandy + 2 storms + 1 goal + 1 start
    # 1 is start, 2 is rocks, 3 is sandy, 4 is storm, 5 is organic sample
    random_grid.ravel()[stuff_cells[0]] = 1
    random_grid.ravel()[stuff_cells[1:4]] = 2
    random_grid.ravel()[stuff_cells[4:7]] = 3
    random_grid.ravel()[stuff_cells[7:9]] = 4
    random_grid.ravel()[stuff_cells[9]] = 5
    start_x = None
    start_y = None
    print(random_grid)

    # populate hidden cells (the real environment if the problem was fully observable)
    for i in range(len(random_grid[:,0])):
        for j in range(len(random_grid[0,:])):
            x = j
            y = i
            # row i, column j
            # print("Row "+str(i)+", column "+str(j)+" is "+str(random_grid[i,j]))
            lidar_up = get_nearest_rock(random_grid,x,y,"up")
            lidar_down = get_nearest_rock(random_grid,x,y,"down")
            lidar_left = get_nearest_rock(random_grid,x,y,"left")
            lidar_right = get_nearest_rock(random_grid,x,y,"right")
            storm_nearby = check_storm_nearby(random_grid,x,y)

            if random_grid[y,x] == 4:
                storm = "true"
            else:
                storm = "false"
            if random_grid[y,x] == 3:
                sandy = "true"
            else:
                sandy = "false"
            if random_grid[y,x] == 2:
                rock = "true"
            else:
                rock = "false"
            if random_grid[y,x] == 5:
                biosample = "true"
            else:
                biosample = "false"
            if random_grid[y,x] == 1:
                start_x = x
                start_y = y
            # assert a new fact through its template
            hidden_cell_template = environment.find_template('hidden-cell')
            fact = hidden_cell_template.assert_fact(xloc=x,
                                    yloc=y,
                                    rock=rock,
                                    biosample=biosample,
```

```
                                    storm=storm,
                                    storm_nearby=storm_nearby,
                                    sandy=sandy,
                                    lidar_up=lidar_up,
                                    lidar_down=lidar_down,
                                    lidar_left=lidar_left,
                                    lidar_right=lidar_right
                                    )

    # populate agent's cell KB (what the agent knows)
    for i in range(len(random_grid[:,0])):
        for j in range(len(random_grid[0,:])):
            x = j
            y = i
            if not (x == start_x and y == start_y):
                storm = "unknown"
                storm_nearby = "unknown"
                sandy = "unknown"
                rock = "unknown"
                safe = "unknown"
                biosample = "unknown"
                visited = "false"
            else:
                storm = "false"
                sandy = "false"
                storm_nearby = "unknown"
                rock = "false"
                biosample = "false"
                safe = "true"
                visited = "true"
            cell_template = environment.find_template('cell')
            cell_fact = cell_template.assert_fact(xloc=x,
                                    yloc=y,
                                    rock=rock,
                                    biosample=biosample,
                                    storm=storm,
                                    storm_nearby=storm_nearby,
                                    sandy=sandy,
                                    safe=safe,
                                    visited=visited,
                                    time_checked=0
                                    )

    # Populate initial agent state.
    agent_template = environment.find_template('agent')
    agent = agent_template.assert_fact(xloc=start_x,
                                   yloc=start_y,
                                   orientation="right",
                                   batt_soc=1,
                                   time=0,
                                   sample_retrieved="false",
                                   destroyed="false",
                                   loc_history=[],
                                   action_history=[]
                                   )
    return environment, random_grid, start_x, start_y

def hybrid_agent(environment,grid,start_x,start_y):
    # Grad students will need to augment this agent with a way to track the movement actions taken in sandy vs. non-sandy areas.
    sim_length = 10000
    t = 0
    total_movement_cost = 0
    while t < sim_length:
        # Get all of the facts associated with cells.
        cell_facts = []
        for fact in environment.facts():
            if fact.template == environment.find_template('cell'):
                cell_facts.append(fact)
        # Get all of the facts associated with hidden cells. (Only used for sandy check)
        hidden_cell_facts = []
        for fact in environment.facts():
            if fact.template == environment.find_template('hidden-cell'):
                hidden_cell_facts.append(fact)

        sandy_cells = []
        for fact in hidden_cell_facts:
          fact_dict = dict(fact)
```

```
  if fact_dict["sandy"] == "true":
    sandy_cells.append([fact_dict["xloc"], fact_dict["yloc"]])

# Get all facts.
facts = list(environment.facts())
plan = []

# Get current rover conditions from KB.
current_pos = ask_current_pos(facts,environment)
x = current_pos[0]
y = current_pos[1]
destroyed = ask_rover_destroyed(facts,environment)
orientation = ask_current_orientation(facts,environment)
rover_pos = RoverPosition(x,y,orientation)

# Check if the rover has been destroyed, end the game if so.
if destroyed:
    print("Game over.")
    return 1, total_movement_cost

# Check if the agent has retrieved the sample and is back in the start position.
if ask_agent_sample(facts,environment) and x == start_x and y == start_y:
    print("You win!")
    return 0, total_movement_cost

# Get a list of safe cells that we can traverse without worry.
safe_points = list()
for cell in cell_facts:
    cell_dict = dict(cell)
    if cell_dict["safe"] == "true":
        safe_points.append([cell_dict["xloc"],cell_dict["yloc"]])

# Check if agent has retrieved the sample.
if len(plan) == 0:
    if ask_agent_sample(facts,environment):
        print("Sample acquired!")
        goals = list()
        goals.append([start_x, start_y])
        actions = plan_route(rover_pos, goals[0], safe_points, len(grid[:,0]))
        plan.extend(actions)

# Check if the agent is in a cell with the biosample. If so, drill.
if len(plan) == 0:
    if ask_cell_bio(x,y,cell_facts):
        action = "drill"
        plan.append(action)


# Here you should write code that checks for unvisited but safe spots to visit, and adds maneuvers to those spots to the plan,
# using the plan_route function. See the "not_unsafe" code below for a similar implementation.

###
### YOUR CODE HERE
if len(plan) == 0:
    safe = list()
    for cell in cell_facts:
        cell_dict = dict(cell)
        if cell_dict["safe"] == "true" and cell_dict["visited"] == "false" and not (cell_dict["xloc"] == x and cell_dict["yloc"] == y
            safe.append([cell_dict["xloc"],cell_dict["yloc"]])
    if len(safe) > 0:
        valid_temps = []
        for safe_point in safe:
            safe_points.append(safe_point)
            temp = plan_route(rover_pos, safe_point, safe_points, len(grid[:,0]))
            safe_points.remove(safe_point)
            if temp is not None:
                valid_temps.append(temp)
        if len(valid_temps) > 0:
            temp = random.choice(valid_temps)
            plan.extend(temp)
###

# Check if lidar has been done in this cell/orientation. If not, lidar.
if len(plan) == 0:
    if not ask_if_lidar_this_loc(x,y,ask_current_orientation(facts,environment),facts,environment):
        action = "lidar"
```

```
                plan.append(action)

        # Check if traction has been measured in this cell/orientation. If not, traction.
        if len(plan) == 0:
            if ask_sandy_unknown(x,y,cell_facts):
                action = "traction"
                plan.append(action)

        # Check if air quality/nearby storm status has been measured in this cell/orientation. If not, air_quality.
        if len(plan) == 0:
            if ask_stormnearby_unknown(x,y,cell_facts):
                action = "air_quality"
                plan.append(action)

        # Check if the spectrometer has been used in this cell. If not, spectrometer.
        if len(plan) == 0:
            if ask_bio_unknown(x,y,cell_facts):
                action = "spectrometer"
                plan.append(action)

        # If there are still no safe and unvisited spots remaining and this cell has had all measurements performed, try moving to an unvisite
        # that at least is not unsafe.
        if len(plan) == 0:
            not_unsafe = list()
            for cell in cell_facts:
                cell_dict = dict(cell)
                if not cell_dict["safe"] == "false" and cell_dict["visited"] == "false" and not (cell_dict["xloc"] == x and cell_dict["yloc"]
                    not_unsafe.append([cell_dict["xloc"],cell_dict["yloc"]])
            if len(not_unsafe) > 0:
                valid_temps = []
                for not_unsafe_point in not_unsafe:
                    safe_points.append(not_unsafe_point)
                    temp = plan_route(rover_pos, not_unsafe_point, safe_points, len(grid[:,0]))
                    safe_points.remove(not_unsafe_point)
                    if temp is not None:
                        valid_temps.append(temp)
                if len(valid_temps) > 0:
                    temp = random.choice(valid_temps)
                    plan.extend(temp)

        # If there are no unknown spots to explore, all of the unvisited spots are known to be unsafe, the goal must be blocked off.
        if len(plan) == 0:
            print("No actions to take!")
            print("Game over.")
            return 1, total_movement_cost

        # Execute the first action in the plan.
        action = plan[0]

        # Translating from plan_route terminology to clips rule terminology.
        if action == "Forward":
            action = "forward"
        elif action == "Turnright":
            action = "rotate_cw"
        elif action == "Turnleft":
            action = "rotate_ccw"

        print("Current time: "+str(t))
        print("Current position: "+str(current_pos))
        print("Action taken: "+str(action))

        plan = plan[1:]

        # Update the KB with the action taken. Run the rule environment so that any new firings can occur.
        action_template = environment.find_template('action')
        action_template.assert_fact(type=action,time=t,xloc=x,yloc=y,orientation=ask_current_orientation(facts,environment))
        if action == "forward" or action == "rotate_cw" or action == "rotate_ccw":
            movement_cost = check_sandy_hidden_cell(x,y,hidden_cell_facts)
            total_movement_cost += movement_cost
        environment.run()
        t += 1
    return 1, total_movement_cost


# This code runs 100 trials and tracks the number of victories and the average movement cost.
sum = 0
sims = 100
```

```
mvmt_cost_sum = 0
for i in range(sims):
    env, grid, start_x, start_y = new_game(templates,rules)
    result, mvmt_cost = hybrid_agent(env,grid,start_x,start_y)
    sum += result
    mvmt_cost_sum += mvmt_cost
print("Number of victories: "+str(sims-sum))
print("Movement cost average: "+str(mvmt_cost_sum/sims))

# This prints out the working memory for the last trial run.
print("Final grid observations: ")
for fact in env.facts():
    if fact.template == env.find_template('cell'):
        d = dict(fact)
        exclude_keys = ['up_cell', 'down_cell','right_cell','left_cell']
        new_d = {k: d[k] for k in set(list(d.keys())) - set(exclude_keys)}
        print(", ".join([key+": "+str(value) for key, value in sorted(new_d.items(), key=lambda x: x[0])]))
```

```
Action taken: lidar
Current time: 183
Current position: [0, 0]
Action taken: traction
Current time: 184
Current position: [0, 0]
Action taken: air_quality
Current time: 185
Current position: [0, 0]
Action taken: spectrometer
Current time: 186
Current position: [0, 0]
Action taken: drill
Sample acquired!
Current time: 187
Current position: [0, 0]
Action taken: rotate_ccw
Sample acquired!
Current time: 188
Current position: [0, 0]
Action taken: rotate_ccw
Sample acquired!
Current time: 189
Current position: [0, 0]
Action taken: forward
Sample acquired!
Current time: 190
Current position: [0, 1]
Action taken: forward
Sample acquired!
Current time: 191
Current position: [0, 2]
Action taken: rotate_ccw
Sample acquired!
Current time: 192
Current position: [0, 2]
Action taken: forward
Sample acquired!
Current time: 193
Current position: [1, 2]
Action taken: forward
Sample acquired!
Current time: 194
Current position: [2, 2]
Action taken: forward
Sample acquired!
Current time: 195
Current position: [3, 2]
Action taken: forward
You win!
[[2. 0. 0. 0. 0. 0.]
 [0. 2. 0. 0. 0. 0.]
 [0. 3. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 5.]
 [0. 4. 0. 0. 0. 3.]
 [1. 3. 4. 2. 0. 0.]]
Current time: 0
```