Developing AI Agent with PEAS Description

AIM

To find the PEAS description for the given AI problem and develop an AI agent.

THEORY

A vacuum-cleaner world with 6 locations. Each location can be clean or dirty. The agent can move left, right, up, down,and can clean the square that it occupies.

PEAS DESCRIPTION

Agent type	performance measurement	environment	Actuators	sensors
vaccum cleaner	cleanliness ,number of moments	Rooms	wheels,suction tool	location, cleanliness

DESIGN STEPS

STEP 1:

The inputs are location of the agent and the status of the location

STEP 2:

The output of the system is Right Left and Suck.

STEP 3:

Agent Type:Vaccum Cleaner Performance Measure: Cleanliness , Number of Movements Environment: Rooms Actuators: Wheels and Suction tool Sensor: Location Sensor and Cleanliness sensor

STEP 4:

The agent should detect the location and suck if the location it is dirty,else it should move to the next location.

STEP 5:

The performance is measured with the number of movements and the cleaning action of the agent.

PROGRAM

```
def TableDrivenAgentProgram(table):
    """
    This agent selects an action based on the percept sequence.
    It is practical only for tiny domains.
    To customize it, provide as table a dictionary of all
    {percept_sequence:action} pairs.
    """
    percepts = []

    def program(percept):
        percepts.append(percept)
        action=table.get(tuple(percept))
        return action

    return program

loc_A, loc_B, loc_C, loc_D, loc_E, loc_F = (0,0), (0,1), (0,2), (1,2), (1,1), (1,0) # The two l
#G-20 H-21 I-22
#F-10 E-11 D-12
#A-00 B-01 C-02

def TableDrivenVacuumAgent():
    """
    Tabular approach towards vacuum world
    """
    table = {(loc_A, 'Clean'): 'Right1',
             (loc_A, 'Dirty'): 'Suck',
             (loc_B, 'Clean'): 'Right2',
             (loc_B, 'Dirty'): 'Suck',
             (loc_C, 'Clean'): 'Up',
             (loc_C, 'Dirty'): 'Suck',
             (loc_D, 'Clean'): 'Left1',
             (loc_D, 'Dirty'): 'Suck',
             (loc_E, 'Clean'): 'Left2',
             (loc_E, 'Dirty'): 'Suck',
             (loc_F, 'Clean'): 'Down',
             (loc_F, 'Dirty'): 'Suck',


    }
    return Agent(TableDrivenAgentProgram(table))
#right1,2,3,4 start left1,2 up1,2
```

```
class Environment:
    """Abstract class representing an Environment. 'Real' Environment classes
    inherit from this. Your Environment will typically need to implement:
        percept:           Define the percept that an agent sees.
        execute_action:    Define the effects of executing an action.
                           Also update the agent.performance slot.
    The environment keeps a list of .things and .agents (which is a subset
    of .things). Each agent has a .performance slot, initialized to 0.
    Each thing has a .location slot, even though some environments may not
    need this."""

    def __init__(self):
        self.things = []
        self.agents = []

    def percept(self, agent):
        """Return the percept that the agent sees at this point. (Implement this.)"""
        raise NotImplementedError

    def execute_action(self, agent, action):
        """Change the world to reflect this action. (Implement this.)"""
        raise NotImplementedError

    def default_location(self, thing):
        """Default location to place a new thing with unspecified location."""
        return None

    def is_done(self):
        """By default, we're done when we can't find a live agent."""
        return not any(agent.is_alive() for agent in self.agents)

    def step(self):
        """Run the environment for one time step. If the
        actions and exogenous changes are independent, this method will
        do. If there are interactions between them, you'll need to
        override this method."""
        if not self.is_done():
            actions = []
            for agent in self.agents:
                if agent.alive:
                    actions.append(agent.program(self.percept(agent)))
                else:
                    actions.append("")
            for (agent, action) in zip(self.agents, actions):
                self.execute_action(agent, action)

    def run(self, steps=1000):
        """Run the Environment for given number of time steps."""
        for step in range(steps):
```

```
            if self.is_done():
                return
            self.step()

    def add_thing(self, thing, location=None):
        """Add a thing to the environment, setting its location. For
        convenience, if thing is an agent program we make a new agent
        for it. (Shouldn't need to override this.)"""
        if not isinstance(thing, Thing):
            thing = Agent(thing)
        if thing in self.things:
            print("Can't add the same thing twice")
        else:
            thing.location = location if location is not None else self.default_location(thing)
            self.things.append(thing)
            if isinstance(thing, Agent):
                thing.performance = 0
                self.agents.append(thing)

    def delete_thing(self, thing):
        """Remove a thing from the environment."""
        try:
            self.things.remove(thing)
        except ValueError as e:
            print(e)
            print("  in Environment delete_thing")
            print("  Thing to be removed: {} at {}".format(thing, thing.location))
            print("  from list: {}".format([(thing, thing.location) for thing in self.things]))
        if thing in self.agents:
            self.agents.remove(thing)


class TrivialVacuumEnvironment(Environment):
    """This environment has two locations, A and B. Each can be Dirty
    or Clean. The agent perceives its location and the location's
    status. This serves as an example of how to implement a simple
    Environment."""

    def __init__(self):
        super().__init__()
        self.status = {loc_A: random.choice(['Clean', 'Dirty']),
                       loc_B: random.choice(['Clean', 'Dirty']),
                       loc_C: random.choice(['Clean', 'Dirty']),
                       loc_D: random.choice(['Clean', 'Dirty']),
                       loc_E: random.choice(['Clean', 'Dirty']),
                       loc_F: random.choice(['Clean', 'Dirty']),}

    def thing_classes(self):
        return [ TableDrivenVacuumAgent]
```

```
    def percept(self, agent):
        """Returns the agent's location, and the location status (Dirty/Clean)."""
        return agent.location, self.status[agent.location]
    def clean_check(self):
        number_of_clean_rooms=0
        for key in [loc_A, loc_B, loc_C, loc_D, loc_E, loc_F]:
            if self.status[key] =='Clean':
                number_of_clean_rooms=number_of_clean_rooms+1
        return number_of_clean_rooms
    def execute_action(self, agent, action):
        """Change agent's location and/or location's status; track performance.
        Score 10 for each dirt cleaned; -1 for each move."""
        if (self.clean_check()!=9):

            if action=='Right1':
                agent.location = loc_B
                agent.performance -=1
            elif action=='Right2':
                agent.location = loc_C
                agent.performance -=1
            elif action=='Left1':
                agent.location = loc_E
                agent.performance -=1
            elif action=='Left2':
                agent.location = loc_F
                agent.performance -=1
            elif action=='Up':
                agent.location = loc_D
                agent.performance -=1
            elif action=='Start':
                agent.location = loc_A
                agent.performance -=1
            elif action=='Suck':
                if self.status[agent.location]=='Dirty':
                    agent.performance+=10
                self.status[agent.location]='Clean'

    def default_location(self, thing):
        """Agents start in either location at random."""
        return random.choice([loc_A, loc_B, loc_C, loc_D, loc_E, loc_F])


if __name__ == "__main__":
    agent = TableDrivenVacuumAgent()
    environment = TrivialVacuumEnvironment()
    environment.add_thing(agent)
    print('Agent Before Action\n\n',environment.status)
    print('\nAgent Location : ',agent.location)
```

```
print('\nAgent Performance : ',agent.performance)
print("\nClean rooms : ",environment.clean_check())

environment.run(steps=6)
print('\n\nAgent after Action\n\n',environment.status)
print('\nAgent Location : ',agent.location)
print('\nAgent Performance : ',agent.performance)
print("\nClean rooms : ",environment.clean_check())
```

Breadth First Search

AIM

To develop an algorithm to find the route from the source to the destination point using breadth-first search.

THEORY

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

DESIGN STEPS

STEP 1:

Identify a location in the google map:

STEP 2:

Select a specific number of nodes with distance

STEP 3:

Import required packages.

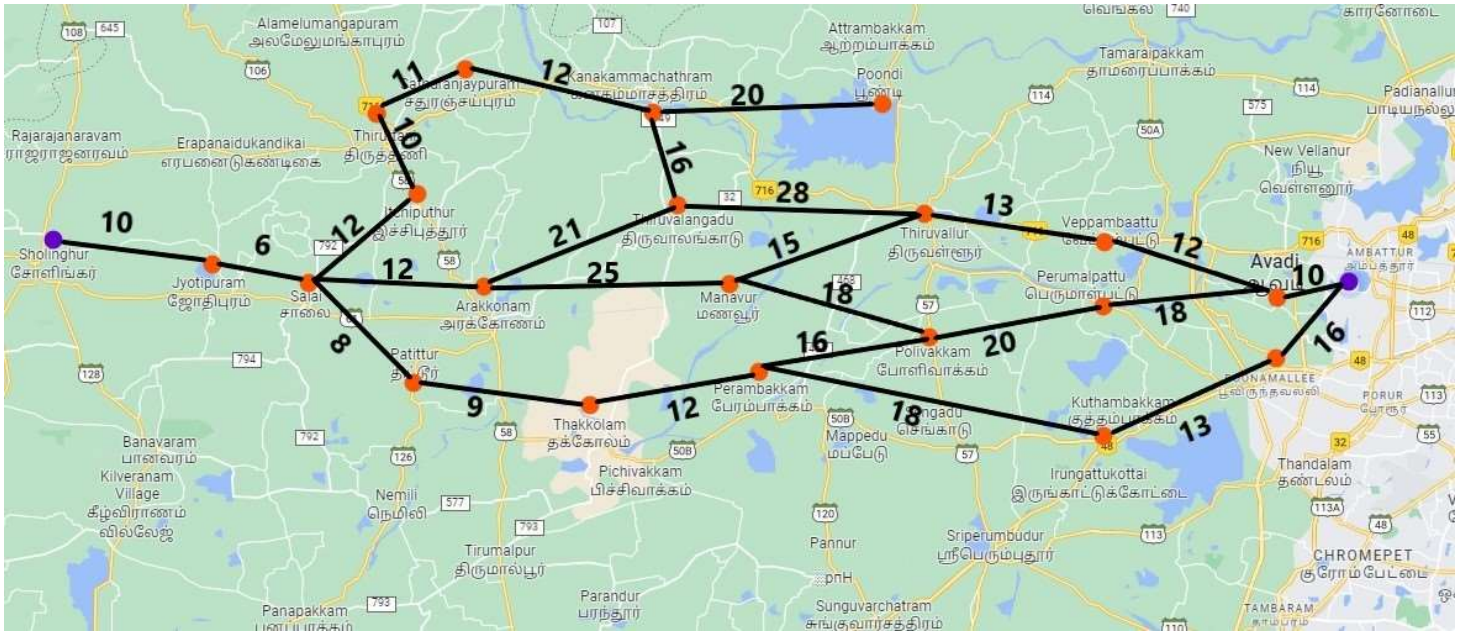
STEP 4:

Include each node and its distance separately in the dictionary data structure.

STEP 5:

End of program.

ROUTE MAP



PROGRAM

Search Algorithm : Breadth First Search

```python
def breadth_first_search(problem):
    "Search shallowest nodes in the search tree first."
    node = Node(problem.initial)
    if problem.is_goal(problem.initial):
        return node
    frontier = FIFOQueue([node])
    reached = {problem.initial}
    while frontier:
        node = frontier.pop()
        for child in expand(problem, node):
            s = child.state
            if problem.is_goal(s):
                return child
            if s not in reached:
                reached.add(s)
                frontier.appendleft(child)
    return failure
```

Route Finding Problems

```
class RouteProblem(Problem):
    """A problem to find a route between locations on a `Map`.
    Create a problem with RouteProblem(start, goal, map=Map(...)}).
    States are the vertexes in the Map graph; actions are destination states."""
```

```
    def actions(self, state):
        """The places neighboring `state`."""
        return self.map.neighbors[state]

    def result(self, state, action):
        """Go to the `action` place, if the map says that is possible."""
        return action if action in self.map.neighbors[state] else state

    def action_cost(self, s, action, s1):
        """The distance (cost) to go from s to s1."""
        return self.map.distances[s, s1]

    def h(self, node):
        "Straight-line distance between state and the goal."
        locs = self.map.locations
        return straight_line_distance(locs[node.state], locs[self.goal])


class Map:
    """A map of places in a 2D world: a graph with vertexes and links between them.
    In `Map(links, locations)`, `links` can be either [(v1, v2)...] pairs,
    or a {(v1, v2): distance...} dict. Optional `locations` can be {v1: (x, y)}
    If `directed=False` then for every (v1, v2) link, we add a (v2, v1) link."""

    def __init__(self, links, locations=None, directed=False):
        if not hasattr(links, 'items'): # Distances are 1 by default
            links = {link: 1 for link in links}
        if not directed:
            for (v1, v2) in list(links):
                links[v2, v1] = links[v1, v2]
        self.distances = links
        self.neighbors = multimap(links)
        self.locations = locations or defaultdict(lambda: (0, 0))


def multimap(pairs) -> dict:
    "Given (key, val) pairs, make a dict of {key: [val,...]}."
    result = defaultdict(list)
    for key, val in pairs:
        result[key].append(val)
    return result


nearby_locations = Map(
    {('Sholinghur', 'Jyotipuram'):  10, ('Jyotipuram', 'Salai'): 6, ('Salai', 'Itchiputhur'): 1
     ('Itchiputhur', 'Thirutani'): 10, ('Thirutani', 'Sathuranjayapuram'):  11, ('Sathuranjayap
     ('Kanakammachathram', 'Thiruvalangadu'): 16, ('Thiruvalangadu', 'Arakkonam'): 21, ('Thiruv
     ('Manavur', 'Tiruvallur'): 15, ('Manavur', 'Polivakkam'): 18, ('Tiruvallur', 'Veppambattu'
     ('Polivakkam', 'Perumalpattu'): 20, ('Perumalpattu', 'Avadi'): 18, ('Tiruvallur', 'Veppamb
```

```
    ('Arakkonam', 'Manavur'): 25, ('Salai', 'Patittur'): 8, ('Patittur','Thakkolam'): 9, ('Tha


r0 = RouteProblem('Sholinghur', 'Poondi', map=nearby_locations)
r1 = RouteProblem('Sholinghur', 'Ambattur', map=nearby_locations)
r2 = RouteProblem('Arakkonam', 'Poonamallee', map=nearby_locations)
r3 = RouteProblem('Manavur', 'Poonamallee', map=nearby_locations)
r4 = RouteProblem('Thiruvalangadu', 'Ambattur', map=nearby_locations)

goal_state_path=breadth_first_search(r1)

print("GoalStateWithPath:{0}".format(goal_state_path))

path_states(goal_state_path)

print("Total Distance={0} Kilometers".format(goal_state_path.path_cost))
```

Dijkstra's Shortest Path Algorithm

AIM

To develop a code to find the shortest route from the source to the destination point using Dijkstra's shortest path algorithm.

THEORY

Dijkstra algorithm is a single-source shortest path algorithm. Here, single-source means that only one source is given, and we have to find the shortest path from the source to all the destination nodes.

DESIGN STEPS

STEP 1:

Identify a location in the google map:

STEP 2:

Select a specific number of nodes with distance

STEP 3:

Create a dictionary with all the node pairs (keys) and their respective distances as the values

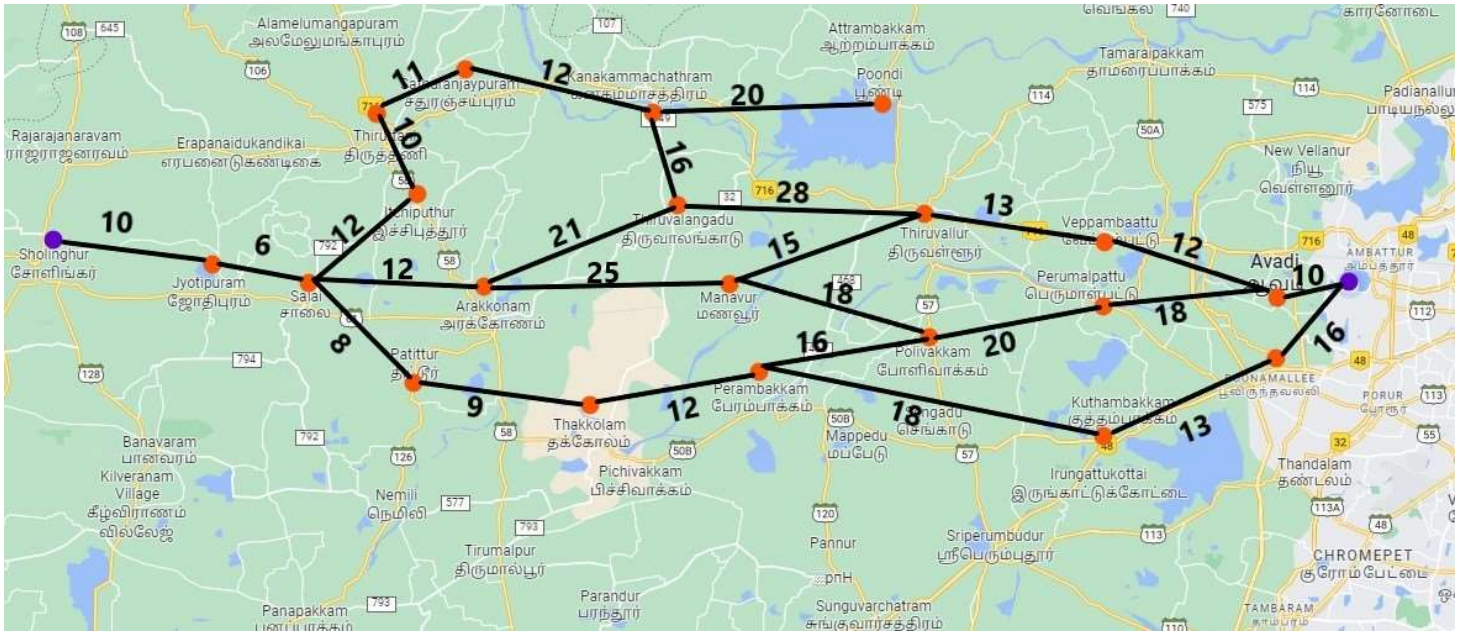
STEP 4:

Implement the search algorithm by passing any two nodes/places to find a possible route.

STEP 5:

Display the route sequence.

ROUTE MAP



PROGRAM

Search Algorithm : Dijkstra's shortest path algorithm

```
class PriorityQueue:
    """A queue in which the item with minimum f(item) is always popped first."""

    def __init__(self, items=(), key=lambda x: x):
        self.key = key
        self.items = [] # a heap of (score, item) pairs
        for item in items:
            self.add(item)

    def add(self, item):
        """Add item to the queuez."""
        pair = (self.key(item), item)
        heapq.heappush(self.items, pair)

    def pop(self):
        """Pop and return the item with min f(item) value."""
        return heapq.heappop(self.items)[1]

    def top(self): return self.items[0][1]

    def __len__(self): return len(self.items)

def best_first_search(problem, f):
    "Search nodes with minimum f(node) value first."
    node = Node(problem.initial)
    frontier = PriorityQueue([node], key=f)
```

```
    reached = {problem.initial: node}
    while frontier:
        node = frontier.pop()
        if problem.is_goal(node.state):
            return node
        for child in expand(problem, node):
            s = child.state
            if s not in reached or child.path_cost < reached[s].path_cost:
                reached[s] = child
                frontier.add(child)
    return failure

def g(n):
    return n.path_cost
```

Route Finding Problems

```
class RouteProblem(Problem):
    """A problem to find a route between locations on a `Map`.
    Create a problem with RouteProblem(start, goal, map=Map(...)}).
    States are the vertexes in the Map graph; actions are destination states."""

    def actions(self, state):
        """The places neighboring `state`."""
        return self.map.neighbors[state]

    def result(self, state, action):
        """Go to the `action` place, if the map says that is possible."""
        return action if action in self.map.neighbors[state] else state

    def action_cost(self, s, action, s1):
        """The distance (cost) to go from s to s1."""
        return self.map.distances[s, s1]

    def h(self, node):
        "Straight-line distance between state and the goal."
        locs = self.map.locations
        return straight_line_distance(locs[node.state], locs[self.goal])



class Map:
    """A map of places in a 2D world: a graph with vertexes and links between them.
    In `Map(links, locations)`, `links` can be either [(v1, v2)...] pairs,
    or a {(v1, v2): distance...} dict. Optional `locations` can be {v1: (x, y)}
    If `directed=False` then for every (v1, v2) link, we add a (v2, v1) link."""
```

```
    def __init__(self, links, locations=None, directed=False):
        if not hasattr(links, 'items'): # Distances are 1 by default
            links = {link: 1 for link in links}
        if not directed:
            for (v1, v2) in list(links):
                links[v2, v1] = links[v1, v2]
        self.distances = links
        self.neighbors = multimap(links)
        self.locations = locations or defaultdict(lambda: (0, 0))


def multimap(pairs) -> dict:
    "Given (key, val) pairs, make a dict of {key: [val,...]}."
    result = defaultdict(list)
    for key, val in pairs:
        result[key].append(val)
    return result

nearby_locations = Map(
    {('Sholinghur', 'Jyotipuram'):  10, ('Jyotipuram', 'Salai'): 6, ('Salai', 'Itchiputhur'): 1
     ('Itchiputhur', 'Thirutani'): 10, ('Thirutani', 'Sathuranjayapuram'):  11, ('Sathuranjayap
     ('Kanakammachathram', 'Thiruvalangadu'): 16, ('Thiruvalangadu', 'Arakkonam'): 21, ('Thiruv
     ('Manavur', 'Tiruvallur'): 15, ('Manavur', 'Polivakkam'): 18, ('Tiruvallur', 'Veppambattu'
     ('Polivakkam', 'Perumalpattu'): 20, ('Perumalpattu', 'Avadi'): 18, ('Tiruvallur', 'Veppamb
     ('Arakkonam', 'Manavur'): 25, ('Salai', 'Patittur'): 8, ('Patittur','Thakkolam'): 9, ('Tha


r0 = RouteProblem('Sholinghur', 'Poondi', map=nearby_locations)
r1 = RouteProblem('Sholinghur', 'Ambattur', map=nearby_locations)
r2 = RouteProblem('Arakkonam', 'Poonamallee', map=nearby_locations)
r3 = RouteProblem('Jyotipuram', 'Polivakkam', map=nearby_locations)
r4 = RouteProblem('Thiruvalangadu', 'Ambattur', map=nearby_locations)

goal_state_path=best_first_search(r3,g)

print("GoalStateWithPath:{0}".format(goal_state_path))

path_states(goal_state_path)

print("Total Distance={0} Kilometers".format(goal_state_path.path_cost))
```

A* Path Finding Algorithm for 2D Grid World

AIM

To develop a code to find the route from the source to the destination point using A* algorithm for 2D grid world.

THEORY

We try to use the A* algorithm to navigate through a 2D Gird environment. We provide the algorithm with the inital and goal states, and then let the algorithm calculate the Heuristic function to decide the path nodes. And finally, we return the path nodes to the user.

DESIGN STEPS

STEP 1:

Build a 2D grid world with initial state and goal state

Initial State: (2,2)

Goal State: (5,8)

STEP 2:

Mention the Obstacles in the 2D grid World

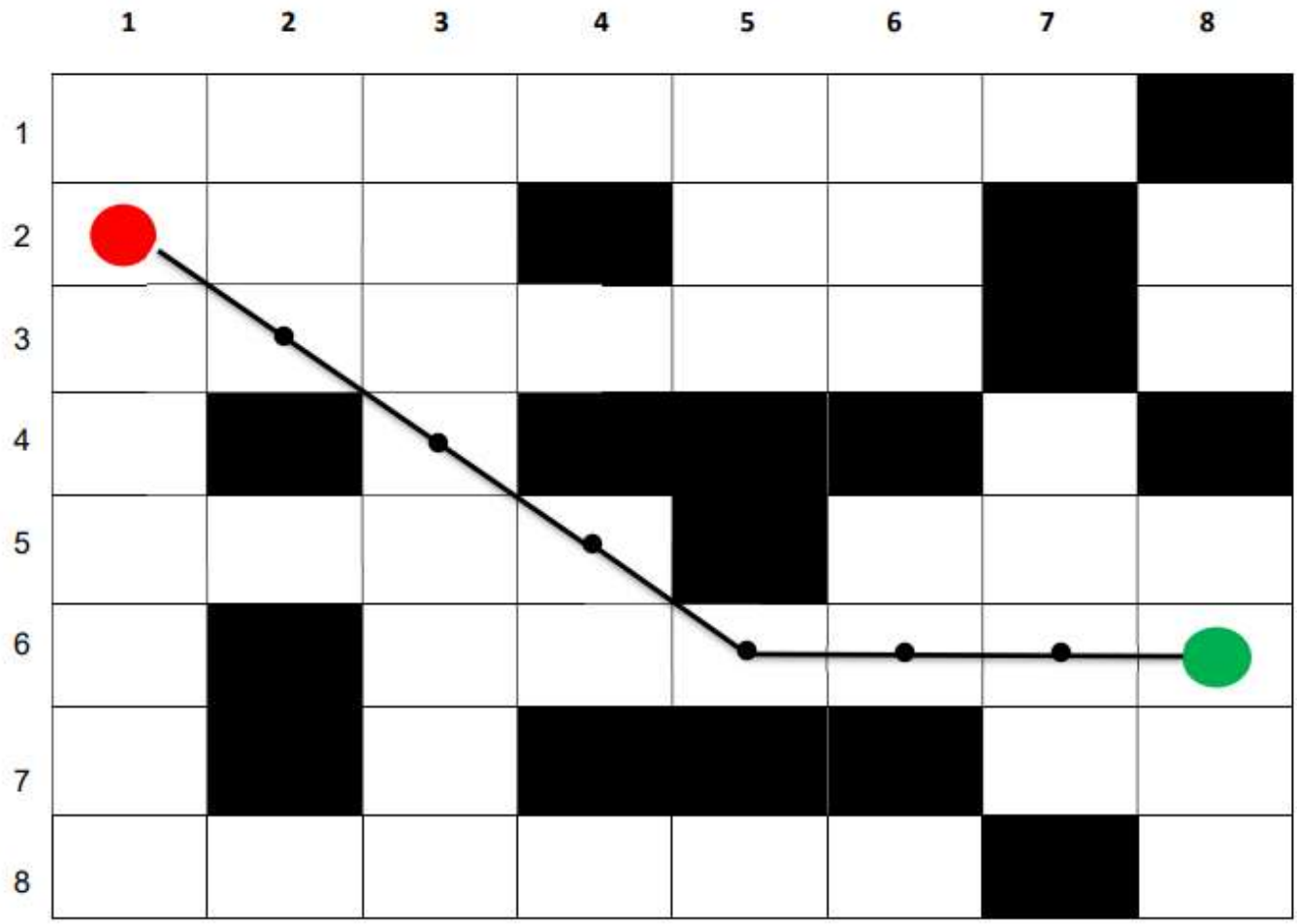
STEP 3:

Define the function for the distance function for the heuristic function

STEP 4:

Pass all the values to the GirdProblem, and print the solution path.

Draw the 2D



PROGRAM

Search Algorithm : Best First Search

```
class PriorityQueue:
    """A queue in which the item with minimum f(item) is always popped first."""

    def __init__(self, items=(), key=lambda x: x):
        self.key = key
        self.items = [] # a heap of (score, item) pairs
        for item in items:
            self.add(item)

    def add(self, item):
        """Add item to the queuez."""
        pair = (self.key(item), item)
        heapq.heappush(self.items, pair)

    def pop(self):
        """Pop and return the item with min f(item) value."""
```

```
        return heapq.heappop(self.items)[1]

    def top(self): return self.items[0][1]

    def __len__(self): return len(self.items)

def best_first_search(problem, f):
    "Search nodes with minimum f(node) value first."
    node = Node(problem.initial)
    frontier = PriorityQueue([node], key=f)
    reached = {problem.initial: node}
    while frontier:
        node = frontier.pop()
        if problem.is_goal(node.state):
            return node
        for child in expand(problem, node):
            s = child.state
            if s not in reached or child.path_cost < reached[s].path_cost:
                reached[s] = child
                frontier.add(child)
    return failure

def g(n):
    return n.path_cost
```

2D Grid Pathfinding Problem

```
class GridProblem(Problem):
    """Finding a path on a 2D grid with obstacles. Obstacles are (x, y) cells."""

    def __init__(self, initial=(15, 30), goal=(130, 30), obstacles=(), **kwds):
        Problem.__init__(self, initial=initial, goal=goal,
                         obstacles=set(obstacles) - {initial, goal}, **kwds)

    directions = [(-1, -1), (0, -1), (1, -1),
                  (-1, 0),           (1,  0),
                  (-1, +1), (0, +1), (1, +1)]

    def action_cost(self, s, action, s1):
        return straight_line_distance(s, s1)

    def h(self, node):
        return straight_line_distance(node.state, self.goal)

    def result(self, state, action):
        "Both states and actions are represented by (x, y) pairs."
        return action if action not in self.obstacles else state
```

```
    def actions(self, state):
        """You can move one cell in any of `directions` to a non-obstacle cell."""

        x, y = state
        return {(x + dx, y + dy) for (dx, dy) in self.directions} - self.obstacles


def straight_line_distance(A, B):
    "Straight-line distance between two points."

    return sum(abs(a - b)**2 for (a, b) in zip(A, B)) ** 0.5

def g(n):

    return n.path_cost

def astar_search(problem, h=None):
    """Search nodes with minimum f(n) = g(n) + h(n)."""
    h = h or problem.h
    return best_first_search(problem, f=lambda n: g(n) + h(n))

obstacles={(8,1),(7,2),(7,3),(2,4),(4,2),(4,4),(5,4),(6,4),(8,4),(5,5),(2,6),(2,7),(4,7),(5,7),

grid1 = GridProblem(initial=(1,2), goal =(8,6) ,obstacles=obstacles)

solution1 = astar_search(grid1)

path_states(solution1)
```

Hill Climbing Algorithm for Eight Queens Problem

AIM

To develop a code to solve eight queens problem using the hill-climbing algorithm.

THEORY

To implement Hill Climbing Algorithm for 8 queens problem

DESIGN STEPS:

STEP 1:

Import the necessary libraries

STEP 2:

Define the Intial State and calculate the objective function for that given state

STEP 3:

Make a decision whether to change the state with a smaller objective function value, or stay in the current state.

STEP 4:

Repeat the process until the total number of attacks, or the Objective function, is zero.

STEP 5:

Display the necessary states and the time taken.

PROGRAM

```
class Problem(object):


    def __init__(self, initial=None, goal=None, **kwds):
```

```
        self.__dict__.update(initial=initial, goal=goal, **kwds)

    def actions(self, state):
        raise NotImplementedError
    def result(self, state, action):
        raise NotImplementedError
    def is_goal(self, state):
        return state == self.goal
    def action_cost(self, s, a, s1):
        return 1

    def __str__(self):
        return '{0}({1}, {2})'.format(
            type(self).__name__, self.initial, self.goal)



class NQueensProblem(Problem):

    def __init__(self, N):
        super().__init__(initial=tuple(random.randint(0,N-1) for _ in tuple(range(N))))
        self.N = N

    def actions(self, state):
        """ finds the nearest neighbors"""
        neighbors = []
        for i in range(self.N):
            for j in range(self.N):
                if j == state[i]:
                    continue
                s1 = list(state)
                s1[i]=j
                new_state = tuple(s1)
                yield Node(state=new_state)

    def result(self, state, row):
        """Place the next queen at the given row."""
        col = state.index(-1)
        new = list(state[:])
        new[col] = row
        return tuple(new)

    def conflicted(self, state, row, col):
        """Would placing a queen at (row, col) conflict with anything?"""
        return any(self.conflict(row, col, state[c], c)
                   for c in range(col))

    def conflict(self, row1, col1, row2, col2):
        """Would putting two queens in (row1, col1) and (row2, col2) conflict?"""
```

```
        return (row1 == row2 or  # same row
                col1 == col2 or  # same column
                row1 - col1 == row2 - col2 or  # same \ diagonal
                row1 + col1 == row2 + col2)  # same / diagonal

    def goal_test(self, state):
        return not any(self.conflicted(state, state[col], col)
                       for col in range(len(state)))

    def h(self, node):
        """Return number of conflicting queens for a given node"""
        num_conflicts = 0
        # Write your code here
        for (r1,c1) in enumerate(node.state):
            for (r2,c2) in enumerate(node.state):
                if (r1,c1)!=(r2,c2):
                    num_conflicts += self.conflict(r1,c1,r2,c2)
        return num_conflicts

def shuffled(iterable):
    """Randomly shuffle a copy of iterable."""
    items = list(iterable)
    random.shuffle(items)
    return items

def argmin_random_tie(seq, key):
    """Return an element with highest fn(seq[i]) score; break ties at random."""
    return min(shuffled(seq), key=key)

def hill_climbing(problem,iterations = 10000):
    # as this is a stochastic algorithm, we will set a cap on the number of iterations
    current = Node(problem.initial)
    i=1
    while i < iterations:
        neighbors = expand(problem,current.state)
        if not neighbors:
            break
        neighbour = argmin_random_tie(neighbors,key=lambda node:problem.h(node))
        if problem.h(neighbour)<=problem.h(current):
            current.state= neighbour.state
            if problem.goal_test(current.state)==True:
                print('The Goal state is reached at {0}'.format(i))
                return current

        i += 1
    return current

nq1=NQueensProblem(8)
plot_NQueens(nq1.initial)
```

```
n1 = Node(state=nq1.initial)
num_conflicts = nq1.h(n1)
import time
start=time.time()
sol1=hill_climbing(nq1,iterations=20000)
end=time.time()
sol1.state
num_conflicts = nq1.h(sol1)
print("Final Conflicts = {0}".format(num_conflicts))
plot_NQueens(list(sol1.state))
print("The total time required for 20000 iterations is {0:.4f} seconds".format(end-start))
```

Sudoku Solver

AIM:

To develop a code to solve a given sudoku puzzle.

THEORY:

We create a python program to solve a given sudoku puzzle by using Elimination and Only Choice strategies.
Firstly, we form the puzzle itself, either by getting data in String form or by indexes and values, from the user.
Then we fill the empty boxes with all the possible values, 1-9. We iteratively apply Elimination and Only Choice Strategies to finally end up with a result.

DESIGN STEPS:

STEP 1:

We define the puzzle. We initialize those boxes with only one value, with the data provided by the user, either in String format or in the Index & value format.

STEP 2:

We identify the Row units, Column units, and the square units. And then we form a unit list using the prior data.

STEP 3:

Two Dictionaries with units and peers of each boxes is defined.

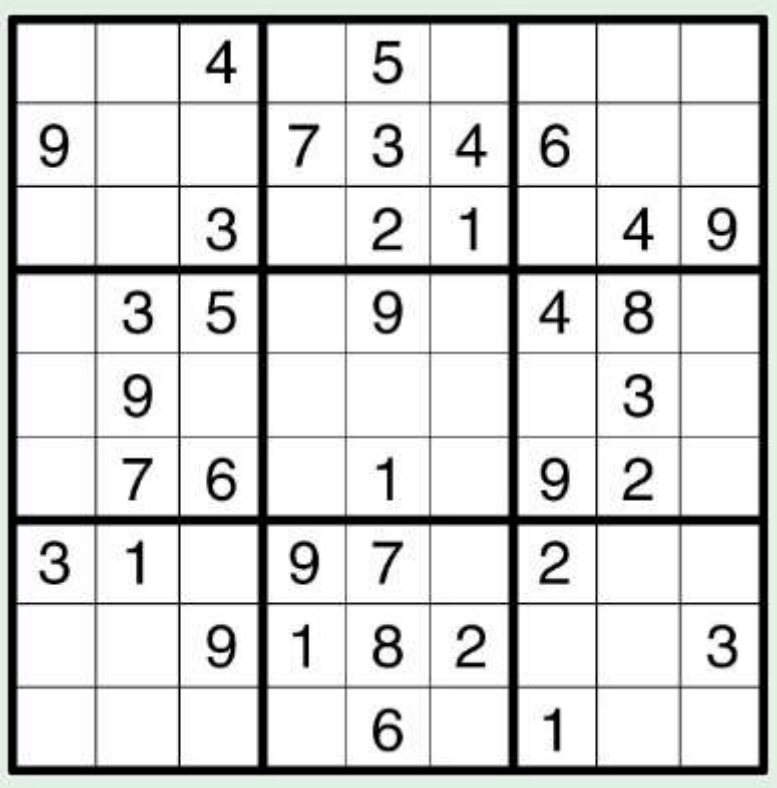
STEP 4:

We reduce the puzzle using the two strategies.

STEP 5:

Search function is defined to find the final solution to a given sudoku puzzle.

Sudoku puzzle



		4		5				
9			7	3	4	6		
		3		2	1		4	9
	3	5		9		4	8	
	9						3	
	7	6		1		9	2	
3	1		9	7		2		
		9	1	8	2			3
				6		1		

PROGRAM:

```
%matplotlib inline
import matplotlib.pyplot as plt
import random
import math
import sys
import time

rows = 'ABCDEFGHI'
cols = '123456789'

def cross(a,b):
    return [i+j for i in a for j in b]
boxes=cross(rows,cols)
ru=[cross(r,cols) for r in rows]
cu=[cross(rows,c) for c in cols]
su=[cross(rs,cs) for rs in ('ABC','DEF','GHI') for cs in('123','456','789')]
ul=ru+cu+su
units = dict((s, [u for u in ul if s in u])  for s in boxes)
peers = dict((s, set(sum(units[s],[]))-set([s]))for s in boxes)
def display(values):
    width = 1+max(len(values[s]) for s in boxes)
    line = '+'.join(['-'*(width*3)]*3)
    for r in rows:
        print(''.join(values[r+c].center (width)+('|' if c in '36' else '') for c in cols))
        if r in 'CF': print(line)
```

```
    return
def grid_values_improved(grid):
    values = []
    all_digits = '123456789'
    for c in grid:
        if c == '.':
            values.append(all_digits)
        elif c in all_digits:
                values.append(c)
    assert len(values) == 81
    boxes = cross(rows,cols)
    return dict(zip(boxes,values))
def elimination(values):
    solved_values = [box for box in values.keys() if len(values[box])==1]
    for box in solved_values:
        digit = values[box]
        for peer in peers[box]:
            values[peer] = values[peer].replace(digit,'')
    return values
def only_choice(values):
    for unit in ul:
        for digit in '123456789':
            dplaces = [box for box in unit if digit in values[box]]
            if len(dplaces) == 1:
                values[dplaces[0]] = digit
    return values
def reduce_puzzle(values):
    stalled =False
    while not stalled:
        solved_values_before = len([box for box in values.keys() if len(values[box])==1])
        elimination(values)
        only_choice(values)
        solved_values_after = len([box for box in values.keys() if len(values[box])==1])
        stalled = solved_values_after == solved_values_before
        if len([box for box in values.keys() if len(values[box])==1])==0:
            return False
    return values
def search(values):
    values_reduced = reduce_puzzle(values)
    if not values_reduced:
        return False
    else:
        values=values_reduced
    if len([box for box in values.keys() if len(values[box])==1])==81:
        return values
    possibility_count_list = [(len(values[box]),box) for box in values.keys() if len(values[box
    possibility_count_list.sort()
    for(_,t_box_min) in possibility_count_list:
        for i_digit in values[t_box_min]:
```

```
            new_values = values.copy()
            new_values[t_box_min]=i_digit
            new_values = search(new_values)
            if new_values:
                return new_values
    return values


    return values
p='..4.5....9..7346....3.21.49.35.9.48..9.....3..76.1.92.31.97.2....9182..3....6.1..'

start_time = time.time()
p1=grid_values_improved(p)
print("Before solving Sudoku Puzzle")
print("\n")
display(p)
result = search(p1)
print("\n\n")
print("After solving Sudoku Puzzle")
display(result)
time_taken=time.time() - start_time
print("\n\n\nThe time required to solve the sudoku puzzle : {0} seconds".format(time_taken))
```