```
import random
import time
class Thing:
    """
    This represents any physical object that can appear in an Environment.
    """
    def is_alive(self):
        """Things that are 'alive' should return true."""
        return hasattr(self, "alive") and self.alive
    def show_state(self):
        print("I don't know how to show_state.")
class Agent(Thing):
        def __init__(self, program=None):
        self.alive = True
        self.performance = 0
        self.program = program
    def can_grab(self, thing):
                return False
def TableDrivenAgentProgram(table):
    percepts = []
    def program(percept):
        action = None
        percepts.append(percept)
        action = table.get(tuple(percepts))
        return action
    return program
room_A, room_B = (0,0), (1,0) # The two locations for the Doctor to treat
def TableDrivenDoctorAgent():
       table = {
        ((room_A, "healthy"),): "Right",
        ((room_A, "unhealthy"),): "treat",
        ((room_B, "healthy"),): "Left",
        ((room_B, "unhealthy"),): "treat",
        ((room_A, "unhealthy"), (room_A, "healthy")): "Right",
        ((room_A, "healthy"), (room_B, "unhealthy")): "treat",
        ((room_B, "healthy"), (room_A, "unhealthy")): "treat",
        ((room_B, "unhealthy"), (room_B, "healthy")): "Left",
        ((room_A, "unhealthy"), (room_A, "healthy"), (room_B, "unhealthy")):
"treat",
        ((room_B, "unhealthy"), (room_B, "healthy"), (room_A, "unhealthy")):
"treat",
    }
    return Agent(TableDrivenAgentProgram(table))
class Environment:
       def __init__(self):
        self.things = []
        self.agents = []
```

```
      def percept(self, agent):
                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):
                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):
               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 TrivialDoctorEnvironment(Environment):
    def __init__(self):
        super().__init__()
        self.status = {
            room_A: random.choice(["healthy", "unhealthy"]),
            room_B: random.choice(["healthy", "unhealthy"]),
        }
    def thing_classes(self):
        return [TableDrivenDocterAgent]
    def percept(self, agent):
        """Returns the agent's location, and the location status
(unhealthy/healthy)."""
        return agent.location, self.status[agent.location]

    def execute_action(self, agent, action):
        """Change agent's location and/or location's status; track performance.
        Score 10 for each treatment; -1 for each move."""
        if action == "Right":
            agent.location = room_B
            agent.performance -= 1
        elif action == "Left":
            agent.location = room_A
            agent.performance -= 1






      elif action == "treat":
```

```
            tem=float(input("Enter your temperature"))
            if tem>=98.5:
                      self.status[agent.location] == "unhealthy"
                      print("medicine prescribed: paracetamol and 
anti-biotic(low dose)")
                      agent.performance += 10
            else:
                self.status[agent.location] = "healthy"
            self.status[agent.location] = "healthy"


    def default_location(self, thing):
        """Agents start in either location at random."""
        return random.choice([room_A, room_B])


if __name__ == "__main__":
    agent = TableDrivenDoctorAgent()
    environment = TrivialDoctorEnvironment()
    environment.add_thing(agent)
    print("\tStatus of patients in rooms before treatment")
    print(environment.status)
    print("AgentLocation : {0}".format(agent.location))
    print("Performance : {0}".format(agent.performance))
    time.sleep(3)

    for i in range(2):
        environment.run(steps=1)
        print("\n\tStatus of patient in room after the treatment")
        print(environment.status)
        print("AgentLocation : {0}".format(agent.location))
        print("Performance : {0}".format(agent.performance))
        time.sleep(3)
```

# Breadth-First Search

```
%matplotlib inline

import matplotlib.pyplot as plt

import random

import math

import sys

from collections import defaultdict, deque, Counter
```

```
from itertools import combinations

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 Node:

    "A Node in a search tree."

    def __init__(self, state, parent=None, action=None, path_cost=0):

        self.__dict__.update(state=state, parent=parent, action=action,
path_cost=path_cost)




def __str__(self):
```

```
        return '<{0}>'.format(self.state)

    def __len__(self):

        return 0 if self.parent is None else (1 + len(self.parent))

    def __lt__(self, other):

        return self.path_cost < other.path_cost

failure = Node('failure', path_cost=math.inf) # Indicates an algorithm couldn't
find a solution.

cutoff  = Node('cutoff',  path_cost=math.inf) # Indicates iterative deepening
search was cut off.

def expand(problem, node):

    "Expand a node, generating the children nodes."

    s = node.state

    for action in problem.actions(s):

        s1 = problem.result(s, action)

        cost = node.path_cost + problem.action_cost(s, action, s1)

        yield Node(s1, node, action, cost)

def path_actions(node):

    "The sequence of actions to get to this node."

    if node.parent is None:

        return []

    return path_actions(node.parent) + [node.action]



def path_states(node):
```

```
    "The sequence of states to get to this node."
    if node in (cutoff, failure, None):
        return []
    return path_states(node.parent) + [node.state]
FIFOQueue = deque
```

Search Algorithm : Breadth First Search

```
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
    # Remove the following comments to initialize the data structure
    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):

       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:

       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

saveetha_nearby_locations = Map(

    {('PERUNGALATHUR', 'TAMBARAM'):  3, ('TAMBARAM', 'CHROMRPET'): 7,
('TAMBARAM', 'THANDALAM'): 10,

     ('CHROMRPET', 'MEDAVAKAM'): 10, ('CHROMRPET', 'THORAIPAKKAM'): 12,
('CHROMRPET', 'GUINDY'): 13,

     ('MEDAVAKAM', 'SIRUSERI'):  11, ('SIRUSERI', 'KELAMBAKKAM'): 8,
('KELAMBAKKAM', 'THORAIPAKKAM'): 17,

     ('KELAMBAKKAM', 'VGP'): 18, ('VGP', 'THIRUVALLUVAR'): 8, ('THIRUVALLUVAR',
'ADYAR'):  5, ('ADYAR', 'GUINDY'): 5,

     ('GUINDY', 'THORAIPAKKAM'): 9, ('GUINDY', 'T-NAGAR'): 5,
('T-NAGAR','MARINABEACH'): 6, ('T-NAGAR','KOYAMBEDU'): 9,

     ('GUINDY','PORUR'): 10, ('KOYAMBEDU','AMBATTUR'): 10,
('AMBATTUR','AVADI'): 10, ('AVADI','POONAMALLEE'): 9,

     ('THANDALAM','SAVEETHAENGINEERINGCOLLEGE'): 18,
('SAVEETHAENGINEERINGCOLLEGE','POONAMALLEE'): 10,

     ('POONAMALLEE','PORUR'): 7, ('THANDALAM','PORUR'): 7})
```

```
r0 = RouteProblem('PERUNGALATHUR', 'KELAMBAKKAM',
map=saveetha_nearby_locations)

r1 = RouteProblem('PERUNGALATHUR', 'MARINABEACH',
map=saveetha_nearby_locations)

r2 = RouteProblem('MARINABEACH', 'SAVEETHAENGINEERINGCOLLEGE',
map=saveetha_nearby_locations)

r3 = RouteProblem('SAVEETHAENGINEERINGCOLLEGE', 'VGP',
map=saveetha_nearby_locations)

r4 = RouteProblem('TAMBARAM', 'T-NAGAR', map=saveetha_nearby_locations)

r5 = RouteProblem('KOYAMBEDU', 'POONAMALLEE', map=saveetha_nearby_locations)

r6 = RouteProblem('KELAMBAKKAM', 'KOYAMBEDU', map=saveetha_nearby_locations)

r7 = RouteProblem('THIRUVALLUVAR', 'PERUNGALATHUR',
map=saveetha_nearby_locations)

r8 = RouteProblem('KELAMBAKKAM', 'SAVEETHAENGINEERINGCOLLEGE',
map=saveetha_nearby_locations)

r9 = RouteProblem('CHROMRPET', 'AVADI', map=saveetha_nearby_locations)

print(r0)

print(r1)

print(r2)

print(r3)

print(r4)

print(r5)

print(r6)

print(r7)

print(r8)
```

```
print(r9)

goal_state_path=breadth_first_search(r2)

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

```
%matplotlib inline

import matplotlib.pyplot as plt

import random

import math

import sys

from collections import defaultdict, deque, Counter

from itertools import combinations

import heapq

Problems

This is the abstract class. Specific problem domains will subclass this.


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 Node:

    "A Node in a search tree."

    def __init__(self, state, parent=None, action=None, path_cost=0):

        self.__dict__.update(state=state, parent=parent, action=action,
path_cost=path_cost)

    def __str__(self):

        return '<{0}>'.format(self.state)

    def __len__(self):

        return 0 if self.parent is None else (1 + len(self.parent))

    def __lt__(self, other):

        return self.path_cost < other.path_cost

failure = Node('failure', path_cost=math.inf) # Indicates an algorithm couldn't
find a solution.

cutoff  = Node('cutoff',  path_cost=math.inf) # Indicates iterative deepening
search was cut off.
```

Helper functions

```
def expand(problem, node):

    "Expand a node, generating the children nodes."

    s = node.state

    for action in problem.actions(s):

        s1 = problem.result(s, action)

        cost = node.path_cost + problem.action_cost(s, action, s1)

        yield Node(s1, node, action, cost)

def path_actions(node):

    "The sequence of actions to get to this node."

    if node.parent is None:

        return []

    return path_actions(node.parent) + [node.action]

def path_states(node):

    "The sequence of states to get to this node."

    if node in (cutoff, failure, None):

        return []

    return path_states(node.parent) + [node.state]
```

Search Algorithm : Dijkstra's shortest path algorithm

```
class PriorityQueue:

      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):

    # Write your code here ; modify the below mentioned line to find the actual
cost

    return n.path_cost
```

Route Finding Problems

```
class RouteProblem(Problem):

    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:
```

```
    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

saveetha_nearby_locations = Map(

    {('PERUNGALATHUR', 'TAMBARAM'):  3, ('TAMBARAM', 'CHROMRPET'): 7,
('TAMBARAM', 'THANDALAM'): 10,

     ('CHROMRPET', 'MEDAVAKAM'): 10, ('CHROMRPET', 'THORAIPAKKAM'): 12,
('CHROMRPET', 'GUINDY'): 13,

     ('MEDAVAKAM', 'SIRUSERI'):  11, ('SIRUSERI', 'KELAMBAKKAM'): 8,
('KELAMBAKKAM', 'THORAIPAKKAM'): 17,

     ('KELAMBAKKAM', 'VGP'): 18, ('VGP', 'THIRUVALLUVAR'): 8, ('THIRUVALLUVAR',
'ADYAR'):  5, ('ADYAR', 'GUINDY'): 5,
```

```
    ('GUINDY', 'THORAIPAKKAM'): 9, ('GUINDY', 'T-NAGAR'): 5,
('T-NAGAR','MARINABEACH'): 6, ('T-NAGAR','KOYAMBEDU'): 9,

    ('GUINDY','PORUR'): 10, ('KOYAMBEDU','AMBATTUR'): 10,
('AMBATTUR','AVADI'): 10, ('AVADI','POONAMALLEE'): 9,

    ('THANDALAM','SAVEETHAENGINEERINGCOLLEGE'): 18,
('SAVEETHAENGINEERINGCOLLEGE','POONAMALLEE'): 10,

    ('POONAMALLEE','PORUR'): 7, ('THANDALAM','PORUR'): 7})

r0 = RouteProblem('PERUNGALATHUR', 'KELAMBAKKAM',
map=saveetha_nearby_locations)

r1 = RouteProblem('PERUNGALATHUR', 'MARINABEACH',
map=saveetha_nearby_locations)

r2 = RouteProblem('MARINABEACH', 'SAVEETHAENGINEERINGCOLLEGE',
map=saveetha_nearby_locations)

r3 = RouteProblem('SAVEETHAENGINEERINGCOLLEGE', 'VGP',
map=saveetha_nearby_locations)

r4 = RouteProblem('TAMBARAM', 'T-NAGAR', map=saveetha_nearby_locations)

r5 = RouteProblem('KOYAMBEDU', 'POONAMALLEE', map=saveetha_nearby_locations)

r6 = RouteProblem('KELAMBAKKAM', 'KOYAMBEDU', map=saveetha_nearby_locations)

r7 = RouteProblem('THIRUVALLUVAR', 'PERUNGALATHUR',
map=saveetha_nearby_locations)

r8 = RouteProblem('KELAMBAKKAM', 'SAVEETHAENGINEERINGCOLLEGE',
map=saveetha_nearby_locations)

r9 = RouteProblem('CHROMRPET', 'AVADI', map=saveetha_nearby_locations)

goal_state_path=best_first_search(r1,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

```
%matplotlib inline

import matplotlib.pyplot as plt

import random

import math

import sys

from collections import defaultdict, deque, Counter

from itertools import combinations

import heapq

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 Node:

    "A Node in a search tree."

    def __init__(self, state, parent=None, action=None, path_cost=0):

        self.__dict__.update(state=state, parent=parent, action=action,
path_cost=path_cost)

    def __str__(self):

        return '<{0}>'.format(self.state)

    def __len__(self):

        return 0 if self.parent is None else (1 + len(self.parent))

    def __lt__(self, other):

        return self.path_cost < other.path_cost

failure = Node('failure', path_cost=math.inf) # Indicates an algorithm couldn't
find a solution.

cutoff  = Node('cutoff',  path_cost=math.inf) # Indicates iterative deepening
search was cut off.

def expand(problem, node):

       s = node.state

    for action in problem.actions(s):

        s1 = problem.result(s, action)

        cost = node.path_cost + problem.action_cost(s, action, s1)

        yield Node(s1, node, action, cost)
```

```
def path_actions(node):

    "The sequence of actions to get to this node."

    if node.parent is None:

        return []

    return path_actions(node.parent) + [node.action]

def path_states(node):

    "The sequence of states to get to this node."

    if node in (cutoff, failure, None):

        return []

    return path_states(node.parent) + [node.state]

class PriorityQueue:

       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):

                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):

    def __init__(self, initial=(5, 10), goal=(5, 3), 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):

              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={(1,1),(1,6),(2,2),(2,3),(3,5),(3,6),(4,5),(4,8),(5,1),(5,2),(5,9),(5,10)}

grid1 = GridProblem(initial=(1,2), goal =(5,3) ,obstacles=obstacles)

solution1 = astar_search(grid1)

path_states(solution1)
```

# Hill Climbing Algorithm for Eight Queens Problem

```
%matplotlib inline

import time

import matplotlib.pyplot as plt

import numpy as np

import random

import math

import sys

from collections import defaultdict, deque, Counter

from itertools import combinations
```

```
from IPython.display import display

from notebook import plot_NQueens
```

Problems

This is the abstract class. Specific problem domains will subclass this.

```
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 Node:

  def __init__(self, state, parent=None, action=None, path_cost=0):
```

```
        self.__dict__.update(state=state, parent=parent, action=action,
path_cost=path_cost)


    def __str__(self):

        return '<{0}>'.format(self.state)

    def __len__(self):

        return 0 if self.parent is None else (1 + len(self.parent))

    def __lt__(self, other):

        return self.path_cost < other.path_cost

failure = Node('failure', path_cost=math.inf) # Indicates an algorithm couldn't
find a solution.

cutoff  = Node('cutoff',  path_cost=math.inf) # Indicates iterative deepening
search was cut off.
```

Helper functions

```
def expand(problem, state):

    return problem.actions(state)
```

Solving NQueens Problem using Hill Climbing

```
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
        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

        neighbor = argmin_random_tie(neighbors,key=lambda node:
problem.h(node))

        if problem.h(neighbor)<=problem.h(current):

            current.state= neighbor.state

            if problem.goal_test(current.state) == True:

                print("Goal test succeeded at iteration {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)

print("Initial Conflicts = {0}".format(num_conflicts))

start=time.time()

sol1=hill_climbing(nq1,iterations=20000)
```

```
end=time.time()

print("Timetaken={0}".format(end-start))

sol1.state

num_conflicts = nq1.h(sol1)

print("Final Conflicts = {0}".format(num_conflicts))

plot_NQueens(list(sol1.state))

import time

iterations=[10,20,30,40,50,1000,2000,3000,4000,5000,10000]

timetaken=[]

num=1

for i in iterations:

    start=time.time()

    sol1=hill_climbing(nq1,iterations=i)

    end=time.time()

    print("The total time required for 2000 iterations is {0:.4f}
seconds\n\n".format(end-start))

    timetaken.append(end-start)

    num+=1

import numpy as np

import numpy as np

from scipy.interpolate import make_interp_spline

import matplotlib.pyplot as plt
```

```
# Dataset

x = np.array(iterations)

y = np.array(timetaken)


X_Y_Spline = make_interp_spline(x, y)


# Returns evenly spaced numbers

# over a specified interval.

X_ = np.linspace(x.min(), x.max(), 500)

Y_ = X_Y_Spline(X_)


# Plotting the Graph

plt.plot(X_, Y_)

plt.title("graph between iteration and timetaken")

plt.xlabel("iterations")

plt.ylabel("timetaken")

plt.show()




# Dataset

x = np.array(iterations)

y = np.array(timetaken)
```

```
# Plotting the Graph

plt.plot(x, y)

plt.title("graph between x and y")

plt.xlabel("timetaken")

plt.ylabel("number of iteration")

plt.show()
```

# Sudoku Solver

```
%matplotlib inline

import random

import matplotlib.pyplot as plt

import math

import sys

import time

rows='ABCDEFGHI'

cols='123456789'

def cross(a,b):

    return[s+t for s in a for t in b]

boxes=cross(rows,cols)

print(boxes)

row_units=[cross(r,cols) for r in rows]

column_units=[cross(rows,c) for c in cols]
```

```
square_units=[cross(rs,cs) for rs in ('ABC','DEF','GHI') for cs in
('123','456','789')]

unitlist=row_units+column_units+square_units

units=dict((s, [u for u in unitlist if s in u ])for s in boxes)

peers=dict((s, set(sum(units[s],[]))-set([s])) for s in boxes)

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

    return dict(zip(boxes, values))

puzzle_dict_improved=grid_values_improved(puzzle)

print(puzzle_dict_improved)

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
display(puzzle_dict_improved)


def eliminate(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 unitlist:
        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 reduced_puzzle(values):
    stalled = False
    while not stalled:
        solved_values_before = len([box for box in values.keys() if
len(values[box]) == 1])
```

```
        eliminate(values)

        only_choice(values)

        solved_values_after = len([box for box in values.keys() if
len(values[box]) == 1])

        stalled = solved_values_before == solved_values_after

        if len([box for box in values.keys() if len(values[box]) == 0]):

            return False

    return values

def search(values):

    values_reduced=reduced_puzzle(values)

    if not values_reduced:

        return False

    else:

        values=values_reduced

    if len([b1 for b1 in boxes if len(values[b1])==1])==81:

        return values

    possibility_count_list=[(len(values[b1]),b1) for b1 in boxes if
len(values[b1])>1]

    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 False

def solve(grid):

    values = grid_values_improved(grid)

    return search(values)

if __name__ == '__main__':


puzzle='.2.86...34..5.17285.9....64.6897...........1..7.136893...2984...23.6..7.7.15....'

    start_time = time.time()

    display(solve(puzzle))

    time_taken=time.time() - start_time

    print("\n\n{0} seconds".format(time_taken))

result=search(puzzle_dict_improved)

if result:

    display(result)

else:

    print("Failed!!!")
```