```
#
# A Boltzmann Machines attempting
# to solve Traveling Salesman Problems
#

import math, random
import string
import numpy as np
import curses

# global function
sigmoid = lambda dE,T: 1 / (1 + math.exp(dE/T))

# # Primary classes for representing the components of a boltzmann
machines
#

class node():
    """Abstract class to represent a city in the travelling
    salesman problem
    """

    def __init__(self,
                 state=0):
        """Initialize the node
        """
        self.state = state

    def get_value(self,states):
        """Returns the sums of the weights * states provided

        Args:
            states (np.matrix): states of connected nodes (binary
representation)

        Returns:
            sum (float): sum of weights * states
        """
        return np.sum(self.weights[states == 1]) * self.state

    def get_state(self):
        """Returns current state
        """
        return self.state

    def set_state(self,state):
        """Sets the node state
        """
        self.state = state
        return
```

```
class boltzmann():
    """The boltzmann machine object, equipped with the necessary tools
    to perform simulated annealing on traveling salesman problems
    """

    def __init__(self,
                 get_state_values=np.vectorize(lambda n:
n.get_state()),
                 hamiltonian_error_charge=0.1,
                 bias_charge=-0.2):
        """Initializes the boltzmann machine with needed
configurations

        Args:
            get_state_values (np.vectorize): mapping function to
create a state matrix given a matrix of nodes
            hamiltonian_error_charge (float): weight charge against
breaking hamiltonian requirements
            bias_charge (float): weight charge that favors at least
one node within an epoch activating
        """

        self.get_state_values = get_state_values
        self.hamiltonian_error_charge = hamiltonian_error_charge
        self.bias_charge = bias_charge

    def
create_network(self,distances,node_map=string.ascii_uppercase):
        """Creates a network and configures the nodes of the network
according
        to a pre-defined use of its distance matrix

        Args:
            distances (array): distance matrix representing the travel
distances between any nodes
            node_map (np.matrix): dictionary storing the titles for
each node (city)
        """
        # store a dictionary to identify the
        # nodes within the network
        self.node_map = node_map

        # number of cities
        n_count = np.size(distances,0)
        # number of epochs
        t_count = n_count + 1

        # create the boltzmann machine with
        # unconfigured nodes
```

```
        self.network = np.matrix([ node() for n in range(n_count *
t_count) ])
        self.network.shape = (n_count,t_count)

        # map a weight, w, with the provided mapping function
        # to achieve the property that consensus value declines
        # when shorter paths are chosen AND rises when no path is
chosen
        self.distances = distances
        map_weights = np.vectorize(lambda w: -100 * math.exp(-w) if
w != 0 else 0.0)
        modified_distances = map_weights(distances)

        # iterate through all nodes (city and epoch) and configure
        # their weight matrices
        for city in range(n_count):
            for epoch in range(t_count):
                # initialize the weight matrix to zeros
                self.network[city,epoch].weights =
np.matrix(np.zeros(n_count * t_count))
                self.network[city,epoch].weights.shape =
(n_count,t_count)

                # encourage visits to short-distance destinations
(epoch +1/-1) using the
                # modified distance matrix
                self.network[city,epoch].weights[:,(epoch + 1) %
t_count] = modified_distances[city].T
                self.network[city,epoch].weights[:,(epoch - 1) %
t_count] = modified_distances[city].T

                # avoid visiting the same city twice in a tour
                self.network[city,epoch].weights[city] =
self.hamiltonian_error_charge

                # avoid visiting more than one city within an epoch
                self.network[city,epoch].weights[:,epoch] =
self.hamiltonian_error_charge

                # if it's the first or final epoch, encourage
completing
                # the hamiltonian tour by adding a high cost to
closing the HT loop
                # with different cities but encourange closing it with
the same city
                if epoch == 0:
                    self.network[city,epoch].weights[:,-1] =
self.hamiltonian_error_charge
                    self.network[city,epoch].weights[city,-1] = -1
                elif epoch == t_count - 1:
```

```
                    self.network[city,epoch].weights[:,0] =
self.hamiltonian_error_charge
                    self.network[city,epoch].weights[city,0] = -1

                # finally, encourage at least one city to be activated
(acts as a bias)
                self.network[city,epoch].weights[city,epoch] =
self.bias_charge

    def get_states(self):
        """Returns a matrix of node states representing the current
network

        Returns:
            (np.matrix): node state values
        """
        return self.get_state_values(self.network)

    def get_distance(self):
        """Returns travel distance of current network, if it makes a
hamiltonian tour (HT)

        Returns:
            distance (float): total distance traveled in current
network
        """
        return get_distance(self.get_states(),self.distances)

    def get_proposed_states(self,activate,deactivate):
        """Generates a matrix that represents a new proposed states by
flipping
        the activated states and the deactivated states

        Args:
            activate ([int,int]): indices of the nodes to be activated
            activate ([int,int]): indices of the nodes to be
deactivated

        Returns:
            proposed_states (np.matrix): new state matrix with flipped
node state
        """
        proposed_states = np.matrix(self.get_states())
        for deactive in deactivate:
            proposed_states[deactive[0],deactive[1]] = 0
        for active in activate:
            proposed_states[active[0],active[1]] = 1

        return proposed_states
```

```
    def get_consensus(self):
        """Computes the consensus value of the current network
        """
        get_values = np.vectorize(lambda n:
n.get_value(self.get_states()))
        total = np.sum(get_values(self.network))
        return total

    def get_next_nodes(self):
        """Returns the next randomnly-selected node within the current
network

        Returns:
            activate (array[int,int]): indices of nodes to turn on
            deactivate (array[int,int]): indices of nodes to turn off
        """
        cities,epochs = self.network.shape
        current_states = self.get_states()

        # choose the nodes to switch across two epochs
        t1 = random.randint(0,epochs-1)
        t2 = random.randint(0,epochs-1)
        n1 = list(current_states[:,t1].A1).index(1)
        n2 = list(current_states[:,t2].A1).index(1)

        # index the swap
        activate = [[n1,t2],[n2,t1]]
        deactivate = [[n1,t1],[n2,t2]]

        # if one is a starting/finishing city, ensure the loop is
completed
        if t1 == 0 or t1 == epochs-1:
            tf = abs(t1 - (epochs-1))
            activate.append([n2,tf])

deactivate.append([list(current_states[:,tf].A1).index(1),tf])
        elif t2 == 0 or t2 == epochs-1:
            tf = abs(t2 - (epochs-1))
            activate.append([n1,tf])

deactivate.append([list(current_states[:,tf].A1).index(1),tf])

        return activate,deactivate

    def propose_states(self,dE,T):
        """Proposes to the node to change state, and the node
        follows the Metropolis Acceptance Criterion

        Args:
            dE (float): change in objective value with the node
```

```
changing state
            T (float): current temperature in annealing process

        Returns:
            acceptance (bool): whether or not the new states should be
accepted
        """
        if dE <= 0:
            return True
        else:
            prob = (1/self.network.size) * math.log10(T) *
sigmoid(math.exp(dE),T)
            if random.random() < prob:
                return True
            else:
                return False

    def print_states(self):
        """Prints out the current state matrix in readable format

        Returns:
            states (str): string-formatted representation of the
current states
        """
        states = self.get_states()
        cities,epochs = states.shape

        header = '-- network state --\n  '
        for e in range(epochs):
            header += ' %s' % (e+1)
        print_str = header + '\n'

        for i in range(cities):
            print_str += str(self.node_map[i]) + ' ' +
str(states[i].A1) + '\n'

        return print_str

    def print_tour(self):
        """Prints the hamiltonian tour in a readable format

        Returns:
            tour (str): string-formatted representation of the tour
taken
        """
        return print_tour(self.get_states(),self.node_map)

#
# Supporting functions for simulated annealing
# on boltzmann machines
```

```
#

def hamiltonian(states):
    """Checks that the given state matrix creates a valid hamiltonian
tour (HT)

    Returns:
        status (bool): True if the state matrix creates a HT; False
otherwise
    """
    cities,epochs = states.shape
    tour = {}
    for t in range(epochs):
        event = states[:,t].A1 # collects the epoch column containing
nodes within a given epoch
        if np.sum(event) != 1: # if < 1 or > 1 nodes are activated,
breaks the HT
            return False

        # all of these epochs below have only one node activated
        for i,n in enumerate(event):
            if n == 1: # if a node is activated
                if t == epochs - 1: # is it the last stop in the HT
                    if tour[i] != 0: # and does it return to the
starting locations
                        return False
                elif i in tour: # or does it repeat another
destination
                    return False
                else: # if not, add it to the tour to review later
                    tour[i] = t

    return True

def get_distance(states,distances):
    """Returns the distance traveled by a state matrix given the
distance matrix

    Args:
        states (np.matrix): state matrix representing nodes visited
along a tour
        distances (np.matrix): distance matrix describing the distance
between any two nodes

    Returns:
        distance (float): total distance traveled on the hamiltonian
tour
    """
    tour = get_tour(states)
    distance = 0
```

```
    for i,stop in enumerate(tour[1:]):
        distance += distances[stop,tour[i]]

    return distance

def get_tour(states):
    """Create a string sequence cities traveled at each epoch to
    represent the hamiltonian tour

    Returns:
        tour (string): the active tour in the current network
    """
    if not hamiltonian(states): # if a HT hasn't been completed, then
distance can't be computed
        return 'hamiltonian tour not complete'

    cities,epochs = states.shape
    tour = []

    for t in range(epochs): # for each epoch
        event = states[:,t].A1 # grab the array describing the states
of its nodes
        for i,n in enumerate(event): # and, for each node within it
            if n == 1: # if it's active
                tour.append(i) # add its index to the tour object

    return tour

def print_tour(states,node_map):
    """Prints the tour in a readable format

    Args:
        states (np.matrix): state matrix
        node_map (dict): dictionary of city titles for each node
    Returns:
        tour (str): string-formatted representation of the your taken
    """
    insert = '->'
    tour = get_tour(states)
    print_str = '%s' % node_map[tour[0]]
    for i in tour[1:]:
        print_str += (insert + node_map[i])

    return print_str

def consensus(states,network):
    """Consensus function, computing the summated values of weights
and states between
    every node (city,epoch) in the network
```

```
    Args:
        states (np.matrix): matrix representing the states of the
nodes provided
        network (np.matrix): matrix of nodes holding the weights

    Returns:
        total (int): total consensus value
    """
    get_values = np.vectorize(lambda n: n.get_value(states))
    total = 0.5 * np.sum(get_values(network))
    return total

def anneal(machine,T=500,schedule=lambda T: math.log10(T) if T > 10
else 0.1):
    """Simulated annealing function on a boltzmann machine (or
Hopfield network, in general)

    Args:
        machine (boltzmann): machine with network of nodes to be
optimized
        T (float): starting temperature
        schedule (lambda): function to apply to T after each cycle
    """
    stop_T = 1
    min_dist = np.inf
    min_conf = None

    # first, create a hamiltonian tour
    cities,epochs = machine.get_states().shape
    cities_left = [ i for i in range(cities) ]
    first_city = 0
    for i in range(cities): # place the cities in random order across
epochs
        selected = random.randint(0,len(cities_left)-1)
        machine.network[cities_left[selected],i].set_state(1)
        if i == 0: first_city = selected
        del cities_left[selected]
    machine.network[first_city,cities].set_state(1) # add the first
city added to end of the list
    print('-- annealing %s --' % machine)

    # the temperature schedule starts here
    while T > stop_T:
        print('iterating, current T=%s...' % T)

        # randomnly select a node and generate a new proposed state
        activate,deactivate = machine.get_next_nodes()
#             print('activate: %s' % activate)
#             print('deactivate: %s' % deactivate)
        current_states = machine.get_states()
```

```
        proposed_states =
machine.get_proposed_states(activate,deactivate)
#             print('current states:\n%s' % current_states)
#             print('proposed_states:\n%s' % proposed_states)

        # compute the energy of the current and proposed state and
take their difference
        E1 = consensus(current_states,machine.network)
        E2 = consensus(proposed_states,machine.network)
        dE = E2 - E1

        # check if the new states meet the metropolis criterion
        if machine.propose_states(dE,T):
            # since they do, change the configuration to the proposed
state
            for n in range(cities):
                for t in range(epochs):
                    if proposed_states[n,t] == 1:
                        machine.network[n,t].set_state(1)
                    else:
                        machine.network[n,t].set_state(0)

        final_states = machine.get_states()

        final_dist = get_distance(final_states,machine.distances)
        if final_dist < min_dist:
            min_dist = final_dist
            min_conf = machine.get_states()

        print(machine.print_tour())
        print('distance=%s' % machine.get_distance())

        T -= schedule(T)

    print('-- annealing complete --')
    print('final T: %s' % T)
    print('minimum:')
    print('\ttour = %s' % print_tour(min_conf,machine.node_map))
    print('\tdistance = %s' % min_dist)
    print('final:')
    print('\ttour = %s' % machine.print_tour())
    print('\tdistance = %s' % machine.get_distance())
```