CS362/633: Artificial Intelligence Pre-Mid Lab Report Team - OG

Anurag Jadhav *202051031* | Hardik Garbyal *202051079* | Pratik Mathpati *202051115* | Mayank Mangal Mourya *202051116*

I. Solved Assignments

Lab Assignment 1, Lab assignment 3, Lab Assignment 4, Lab Assignment 7 are solved which are in order -

- Lab Assignment 7
- Lab Assignment 1
- Lab Assignment 3
- Lab Assignment 4

II. Lab Assignment 7

To model the low level image processing tasks in the framework of Markov Random Field and Conditional Random Field. To understand the working of Hopfield network and use it for solving some interesting combinatorial problems

III. Many low level vision and image processing problems are posed as minimization of energy function defined over a rectangular grid of pixels. We have seen one such problem, image segmentation, in class. The objective of image denoising is to recover an original image from a given noisy image, sometimes with missing pixels also. MRF models denoising as a probabilistic inference task. Since we are conditioning the original pixel intensities with respect to the observed noisy pixel intensities, it usually is referred to as a conditional Markov random field. Refer to (3) above. It describes the energy function based on data and prior (smoothness). Use quadratic potentials for both singleton and pairwise potentials. Assume that there are no missing pixels. Cameraman is a standard test image for benchmarking denoising algorithms. Add varying amounts of Gaussian noise to the image for testing the MRF based denoising approach. Since the energy function is quadratic, it is possible to find the minima by simple gradient descent. If the image size is small (100x100) you may use any iterative method for solving the system of linear equations that you arrive at by equating the gradient to zero.

Some of the Important term related with "Markov's Random Field

- **Energy Minimization**
  Many vision task are naturally posed as energy minimization problems on a rectangular grid of pixels

  $$E(u) = E_{data(u)} + E_{smoothness(u)}$$

  Where the term $E_{data(u)}$ expresses our goal that the optimal model u be consistent with the measurement. The smoothness energy $E_{smoothness(u)}$ is derived from the prior knowledge about plausible solutions.
  "The goal of energy Minimization is to find a set of coordinates representing the minimum energy conformation for the given structure or we can say for a given grid of pixels"
- **Image Denoising**
  Given a noisy image with some pixel value representing in nd array form, where some measurement may be missing, recover the original image I(x,y) which is typically assumed to be smooth.
- **Potential**
  A potential is a non-negative function of variable x. Where a joint potential is non negative func of the set of variables. We don't have to normalize it because it is a non-negative function.
- **Markov's Random field**
  It is defined as the product of potential of the maximal clique(it is the set of random variables taken from set X) with 1/z as normalizer.

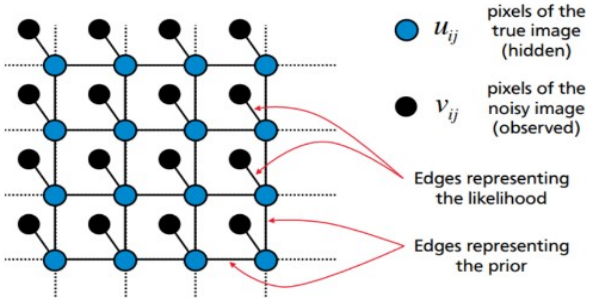
  $$P(X) = 1/z \prod_{C=1}^{c} \phi(X_c)$$

  Given a $G(v, e)$ a graph, where X = set of node's associated with random variable u_i and there would be N_i neighbor node's.

  Then Markov's random field satisfies the,

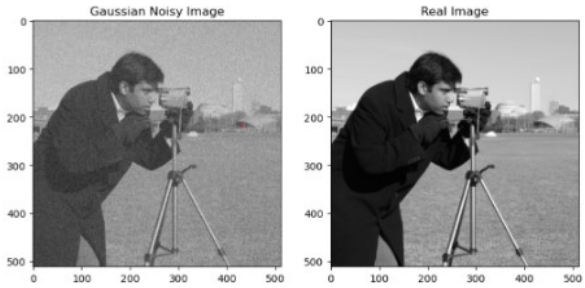
  "Probability distribution of i^th Random Variable given that for all those random variable's j which belongs to the set of node's for given i. is equal to the probability distribution of a i^th random variable given that for all random variables which are the neighbor of node i."

MRF based Image denoising



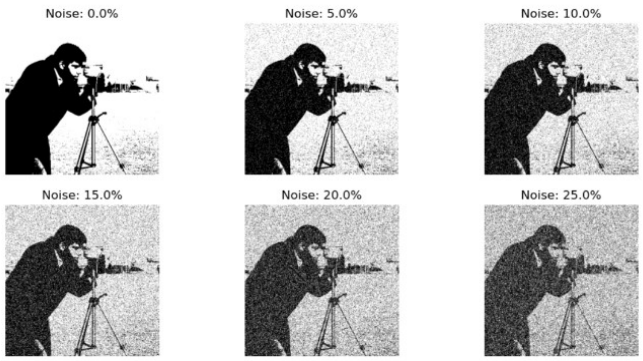
A. *Implementation of MRF's Model.....*

Suppose we have given a Image(v, e) with dimension of 255 x 255 having $nd-array(pixel's)$. Which is the standard test image for bench marking algorithm " Cameraman Image" we will introduce Gaussian noise having parameters are (mean : 0, variance : 0.01, sigma : $sqrt(var)$). Now we have our Gaussian image in the form of $nd-array$. We will simply add this Gaussian image to our original image of the cameraman. After doing this task, we successfully make our image as noisy image(original + $gaussian_noise$).

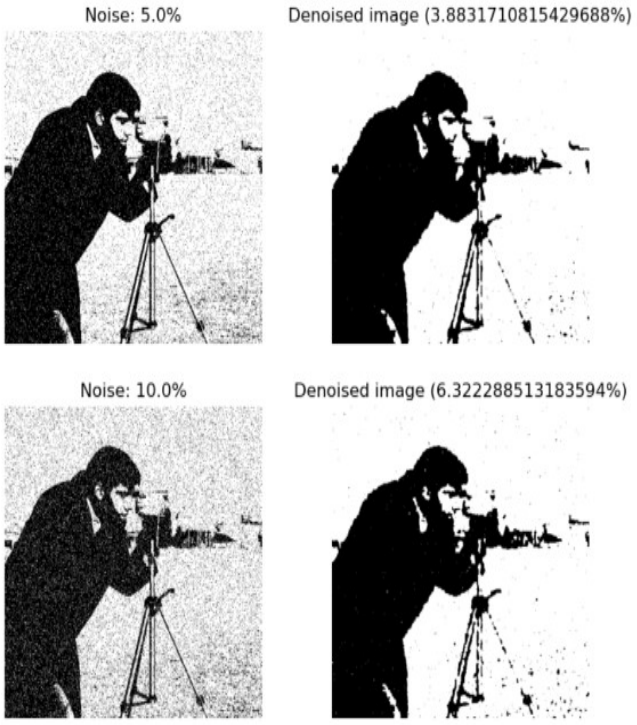


We want to work with a normalized image and we will have to binarized every pixel of our image with the condition of ($image_pixel <= 0.5(binarized0)$ else $(binarized(1))$. Basic analysis part of image noise denoised can be implemented on binarized images for better results.

Now for analysis from scratch we will try to introduce some amount of Gaussian noise with threshold value of 0.05 we will gone do a task with 5 images having following delocalized pixel's $(Noise : 5\%, Noise : 10\%, Noise : 15\%, Noise : 20\%, Noise : 25\%)$. Given below the following aspect's.



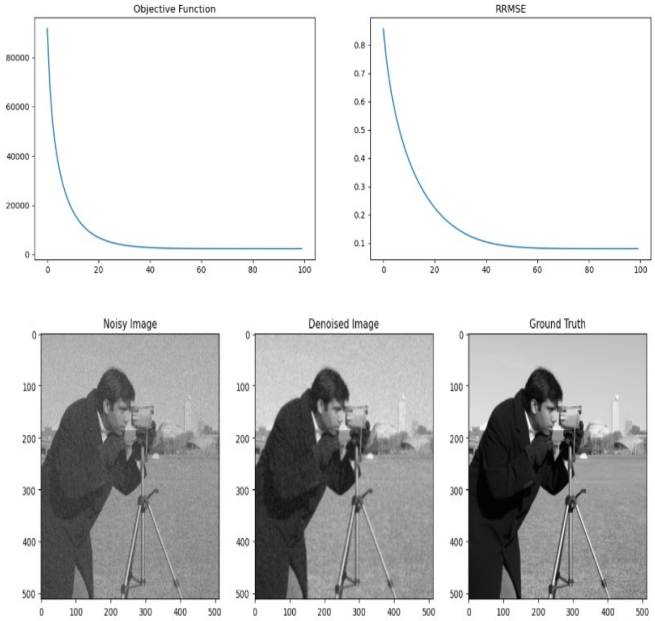
Now we will be familiar with the concept of energy minimization, joint potential and maximal cliques. Which defines Markov's as the Normalized(product(potential(maximal(clique)))). Our next task is given that we have different amounts of images in the same picture. We will have to denoised it for a better result. We gonna do energy minimization of noisy pixel with respect to the conditional probability of neighbor pixels. We will do 1300000 iteration for every image which will denoised the pixels with amount of percentage. Like 3.88 in case 1.



B. *Relative Root mean squared analysis*

On the basis of energy minimization concept we will able to denoised our noisy image with some noisy pixels(e,v) we just calculate the $MRF_{potential}$ as energy of quadratic value MRF_{prior} with addition of the delocalized energy of pixel relative with its adjacency pixel's. For each iteration of the gaussian noisy image we will calculate the errors, losses, After each iteration it will give some error and loss we will plot it as an objective function with how the Relative RMSE will change.

RRMSE Initial: 0.17178742420692128
RRMSE Final : 0.08032410988951892



C. *Conclusion*

- Initially when we denoised our noisy image we will train our epoches and for every epoches we get some objective function loss at every epoche.
- Parallely we get the relative root mean squared error curve for every epoche.
- we will be getting denoised image after energy minimization. and try to compare with ground truth which is our real image we can clearly seen some of the noise will removed by minimization of energy with conditional probability of neighbor node through normalization of product of minimal click's which is our subset of random variable taken from X.
- we successfully do our task of image denoising

IV. For the sample code HOPFIELD.M supplied in the lab-work folder, find out the amount of error (in bits) tolerable for each of the stored patterns.

Hopfield networks are a special kind of recurrent neural networks that can be used as associative memory. Associative memory is memory that is addressed through its contents. That is, if a pattern is presented to an associative memory, it returns whether this pattern coincides with a stored pattern.

That is if we add a certain amount (tolerable) of error to any of the patterns that is known to the network , then it is able to retrieve the original pattern using the Hopfield Network.

For Example :-

A. *Hebbian learning*

A simple model due to Donald Hebb (1949) captures the idea of associative memory. Imagine that the weights between neurons whose activities are positively correlated are increased:

$$\frac{\mathrm{d}w_{ij}}{\mathrm{d}t} \sim \mathrm{Correlation}(x_i, x_j).$$

B. *Binary Hopfield network*

- Convention for weights. Our convention in general will be that wij denotes the connection from neuron j to neuron i.
- Architecture:- A Hopfield network consists of I neurons. They are fully connected through symmetric, bidirectional connections with weights wij = wji . There are no self-connections, so wii = 0 for all i. Biases wi0 may be included (these may be viewed as weights from a neuron '0' whose activity is permanently x0 = 1). We will denote the activity of neuron i (its output) by xi

C. *Testing Algorithm of Discrete Hopfield Net*

1) Step 0: Initialize the weights to store patterns, i.e., weights obtained from training algorithm using Hebb rule.
2) Step 1: When the activations of the net are not converged, then perform Steps 2-8
3) Step 2: Perform Steps 3-7 for each input vector X.
4) **step 3:** Make the initial activations of the net equal to the external input vector X:

$$y_i = x_i \; for \; i = 1 \; to \; n$$

5) **step 4 :** Perform Steps 5-7 for each unit yi. (Here, the units are updated in random order.)
6) **step 5 :** Calculate the net input of the network:

$$y_{ini} = x_i + \sum_j y_i w_{ji}$$

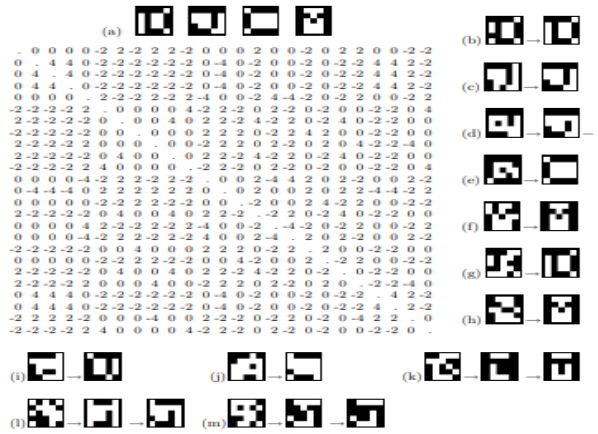
7) **step 6 :** Apply the activations over the net input to calculate the output:

$$y_i = \begin{cases} 1 & if \; y_{ini} > \theta_i \\ y_i & if \; y_{ini} = \theta_i \\ 0 & if \; y_{ini} < \theta_i \end{cases}$$

where θ_i is the threshold and is normally taken as zero.

8) **step 7 :** Now feed back the obtained output yi to all other units. Thus, the activation vectors are updated.

9) **step 8 :** Finally, test the network for convergence.



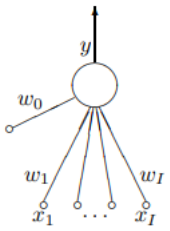
D. Reference :

Single Neuron and Hopfield Network: Chapter 40, 41, 42 Information Theory, Inference and Learning Algorithms, David MacKay

http://www.inference.phy.cam.ac.uk/mackay/itila/

V. Solve a TSP (traveling salesman problem) of 10 cities with a Hopfield network. How many weights do you need for the network?

Many neural network models are built out of single neuron. The architecture of a single neuron has a number I of inputs xi and one output(y).Associated with each input is a weight wi. There may be an additional parameter w0 of the neuron called a bias which may view as being the weight associated with an input.



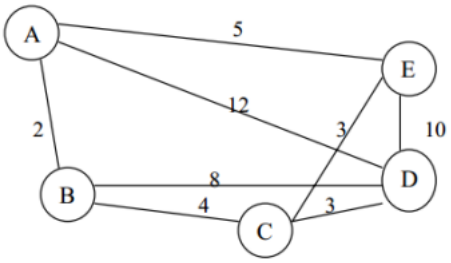
Structure of a neuron

A. Travelling Salesman Problem

Travelling Salesman problem is one of the most famous combinatorial issues of all time. This problem belongs to a category referred to as NP-complete.

Following are the constraints of a TSP problem:

1) The total length of the loop should be a minimum.
2) The salesperson cannot be at two different places at a particular time.
3) The salesperson should visit each city only once.



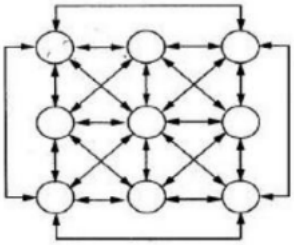
A graph with weights on its edges

According to the above figure, the problem lies in finding the shortest path to pass through all vertices at-least once.
For example, both Path1 (ABCDEA) and Path2 (ABCEDA) passes through all the vertices. However it can be seen that the total length of Path1 is 24, where as the total length of Path2 is 31.

TSP poses a problem in many transportation and logistics based applications. For example, an application to arrange school routes such that all the children are picked up from a particular school district.

B. Hopfield Network

Hopfield network is a dynamic network, which iterates to converge from an arbitrary input state. The Network works as a minimizing energy function. Hopfield is a weighted network where the output of the network is fed back to the neurons and each link has a particular weight. A fully connected Hopfield network is shown figure below,



Hopfield network for 3 neurons

The 3 cities TSP need 9 neurons. This is because n*n neurons are used in the network, where n is the total number of cities.

Consider an example of 4 cities in the order of traveling

$$City2 \rightarrow City1 \rightarrow City4 \rightarrow City3 \rightarrow City2$$

The total distance traveled would be,

$$D = D_{21} + D_{14} + D_{43} + D_{32}$$

	#1	#2	#3	#4
C1	0	1	0	0
C2	1	0	0	0
C3	0	0	0	1
C4	0	0	1	0

Tour matrix obtained as the output of the network

The neural network for any application can be best understood by its energy function. The energy function contains various sections that represent the patterns stored in the network. The input pattern of the network, represents a particular point in the energy landscape. As the input pattern iterates its way to a solution, the point slowly moves towards one of the sections of the landscape. The iterations are either carried out for a fixed number of time or until a stable state is reached.

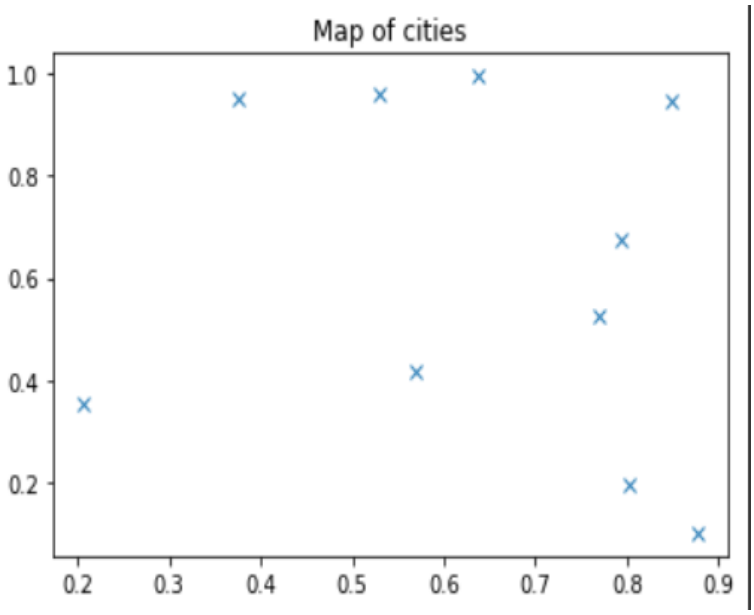
The energy function in a HNN network should satisfy the following conditions:

- The function should lead to a stable state.
- It should provide the shortest travelling path.

The neurons in the network must also satisfy the following criteria (Hopfield networks -:

- The value of each input, xi is determined and the weighted sum of all inputs is $\sum w_i x_i$ calculated.
- The output state of the neuron is set to +1 if the weighted input sum is larger or equal to
- It is set to -1 if the weighted input sum is smaller than 0.
- A neuron retains its output state until it is updated again.

$$o = \begin{cases} 1 & : \sum_i w_i x_i \geq 0 \\ -1 & : \sum_i w_i x_i < 0 \end{cases}$$

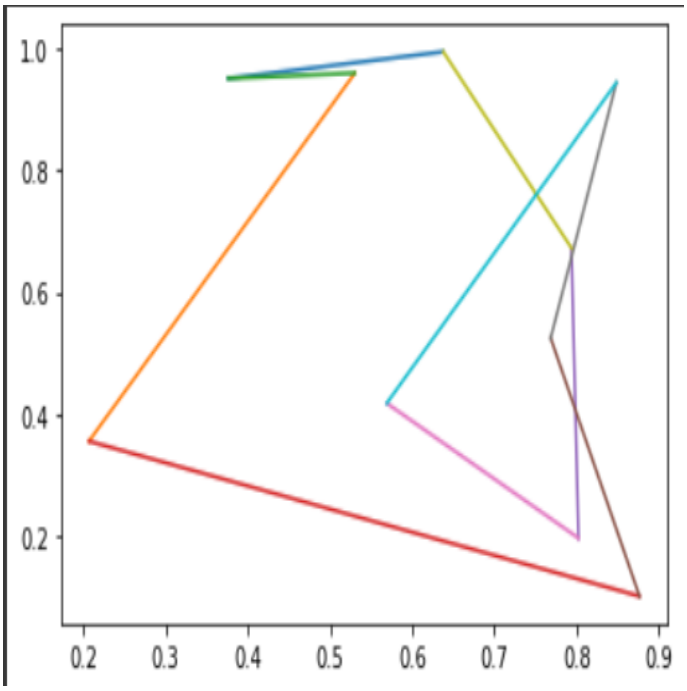


Random coordinates of 10 cities

```
[[0.         0.35807148 0.81573822 0.58147507 0.21656386 0.48808245
  0.92568318 0.77142556 0.11311749 0.26552503]
 [0.35807148 0.         0.47670045 0.339881   0.27701019 0.15006578
  0.57798566 0.6672573  0.3898174  0.50213349]
 [0.81573822 0.47670045 0.         0.32174578 0.74960555 0.33044132
  0.12085163 0.61645266 0.81094295 0.86724883]
 [0.58147507 0.339881   0.32174578 0.         0.59608994 0.22634713
  0.44160779 0.3675888  0.54374395 0.5674716 ]
 [0.21656386 0.27701019 0.74960555 0.59608994 0.         0.42703408
  0.84406297 0.87091882 0.3184699  0.47246543]
 [0.48808245 0.15006578 0.33044132 0.22634713 0.42703408 0.
  0.43765362 0.58667242 0.49615187 0.57961783]
 [0.92568318 0.57798566 0.12085163 0.44160779 0.84406297 0.43765362
  0.         0.71649062 0.92647874 0.98695653]
 [0.77142556 0.6672573  0.61645266 0.3675888  0.87091882 0.58667242
  0.71649062 0.         0.68581728 0.61951786]
 [0.11311749 0.3898174  0.81094295 0.54374395 0.3184699  0.49615187
  0.92647874 0.68581728 0.         0.15458182]
 [0.26552503 0.50213349 0.86724883 0.5674716  0.47246543 0.57961783
  0.98695653 0.61951786 0.15458182 0.        ]]
```

```
Epoch 0: Ran for 1 steps, total distance 3.658638710592205
Epoch 1: Ran for 6 steps, total distance 3.2329706502091726
Epoch 2: Ran for 8 steps, total distance 3.8803370541525335
Epoch 3: Ran for 4 steps, total distance 2.93611622121963
Epoch 4: Ran for 2 steps, total distance 3.2678708135271552
Epoch 5: Ran for 11 steps, total distance 2.9915460840608548
Epoch 6: Ran for 5 steps, total distance 3.452048266569812
Epoch 7: Ran for 14 steps, total distance 3.453017132061736
Epoch 8: Ran for 13 steps, total distance 3.004195797105245
Epoch 9: Ran for 6 steps, total distance 3.5768668986619843
2.93611622121963
[[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]]
```

Different answers over 10 iterations and Best path matrix



Final Output

VI. LAB ASSIGNMENT 1

A. Write a pseudocode for a graph search agent. Represent the agent in the form of a flow chart. Clearly mention all the implementation details with reasons.

Pseudocode :

```
def Graph_Search(env, start, goal):
        frontier = PriorityQueue()
        explored = dict()
        start_node = agent(start.state,start.cost,none)
        frontier.push(start_node)
        while not frontier.is_empty():
            current = frontier.pop()
            if(current == goal):
                return true
            else
                add it to explored
                its childs to frontier(find all childs using env class)
        return false
```

Implementation Details :

1) frontier : This is the priority queue used to store the nodes to be explored.
2) explored : This is the dictionary which stores the explored nodes.
3) First of all we are initializing our start node with the help of agent passing some parameters.
4) Then we are running while loop till our frontier get empty.
5) In the loop we pop the element from frontier, check that is it our goal node.If true we return true
6) If poped state is not the goal state, then we will add it to explored and its childs to frontier.
7) Till getting frontier empty,if we never get goal state then return false.

B. Write a collection of functions imitating the environment for Puzzle-8.

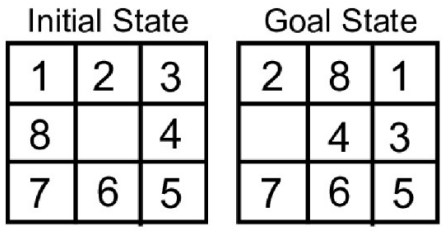


Fig. 1. 8-puzzle

C. References

Single Neuron and Hopfield Network: Chapter 40, 41, 42 Information Theory, Inference and Learning Algorithms, David MacKay

http://www.inference.phy.cam.ac.uk/mackay/itila/

C. Describe what is Iterative Deepening Search.

BFS takes less time but more memory. And in case of DFS it consumes more time ,less memory, but it not always able to find goal state. Also DFS can stuck into infinite loop as it never keeps record of visited node.

In depth limited search we supply a depth limit ‘l’, and treat all nodes at depth ‘l’ as if they had no successors.

```
-In function we have took one state ,depth as input.
-Then we are searching for blank space and storing it in tuple.
Space(0,0)
 for i in range(3):
    for j in range(3):
          if(state[i,j] == '_':
               space = (i,j)
-now on the basis of blank position we are applying all possible swapping functions as follows.(Basically we are swapping numbers)
If space[0] > 0 then we can move it up:
      new_state = copy(state)
      val = new_state[space[0], space[1]]
      new_state[space[0], space[1]]  = new_state[space[0]-1, space[1]]
      new_state[space[0]-1, space[1]] = val
if space[0] < 2 then we can move it down:
      new_state = copy(state)
      val = new_state[space[0], space[1]]
      new_state[space[0], space[1]]  = new_state[space[0]+1, space[1]]
      new_state[space[0]+1, space[1]] = val
if space[1] < 2 then we can move it right:
      new_state = copy(state)
      val = new_state[space[0], space[1]]
      new_state[space[0], space[1]] = new_state[space[0], space[1]+1]
      new_state[space[0], space[1]+1] = val
if space[1] > 0 then  we can move it left:
      new state = copy(state)  val = new_state[space[0], space[1]]
      new_state[space[0], space[1]] = new_state[space[0], space[1]-1]
      new_state[space[0], space[1]-1] = val
```

But choosing such 'l' such that we never miss desirable node is challenging, this problem is solved by iterative deepening search.

In Iterative deepening search, it solve this problem by trying all values for 'l' starting from 0,then 1,then 2, so on until either a solution is found or depth limited search returns the failure.

Thus we will get appropriate 'l' such that we get our goal state. First we perform DFS till 'l' ,then BFS at depth 'l' in this way it reduces space complexity a lot(.ie same as DFS) with assurity of getting solution(i.e. completeness).
Time Complexity : O(b^d) when there is solution, or O(b^m) when there is no solution.
Space Complexity : O(bd)

It is preferred uninformed search when state space is larger than provided memory and d is unknown.

```
def Path_to_goal(start,goal,graph):
        stack = []          //stack to store path(backtracking)
        set = {}            //to store visited node
        stack.push(start)
        set.add(start)
        while(set.size() ne number of node) :
                current = stack.pop()
                if(current eq goal)
                        return stack content in reverse order
                else
                        push such a child state to stack which is not in set
                        if such state not exist then pop from stack
        return goal state not found
```

D. Considering the cost associated with every move to be the same (uniform cost), write a function which can backtrack and produce the path taken to reach the goal state from the source/ initial state.

E. Generate Puzzle-8 instances with the goal state at depth "d".

F. Prepare a table indicating the memory and time requirements to solve Puzzle-8 instances (depth "d") using your graph search agent.

```
0 8.1634521484375e-05 56.0
10 0.0005667829513549805 803.04
20 0.002464246749877929 2906.4
30 0.020881495475769042 13380.64
40 0.14380372524261475 46206.72
50 0.23339185237884522 72383.36
```

Output

As we can see from the above output table, as the depth increases while searching by the agent , cost time taken and the memory usage also increases exponentially.
Time Complexity : O(b^d)
Space Complexity : O(b^d)
Where b = branching factor ,d = depth

```
def generate_start_state(self,depth,goal_state):
    past_state = goal_state
    i=0
    while i!= depth:
        new_states = self.get_next_states(past_state)
        choice = np.random.randint(low=0, high=len(new_states))

        if np.array_equal(new_states[choice], past_state):
            continue

        past_state = new_states[choice]
        i+=1

    return past_state
```

VII. Lab Assignment 3

A. Travelling Salesman Problem (TSP) is a hard problem, and is simple to state. Given a graph in which the nodes are locations of cities, and edges are labelled with the cost of travelling between cities, find a cycle containing each city exactly once, such that the total cost of the tour is as low as possible. For the state of Rajasthan, find out at least twenty important tourist locations. Suppose your relatives are about to visit you next week. Use Simulated Annealing to plan a cost effective tour of Rajasthan. It is reasonable to assume that the cost of travelling between two locations is proportional to the distance between them.

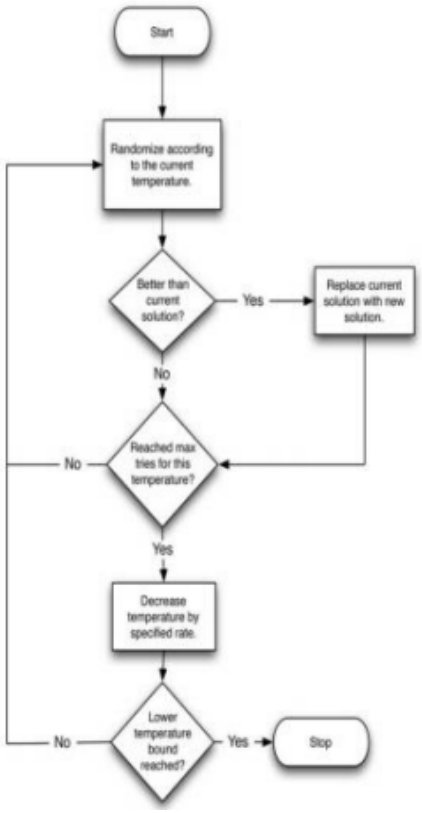
Simulated Annealing is a generic probabilistic meta-algorithm used to find an approximate solution to global optimization problems. Simulated Annealing is inspired by the concept of annealing in metallurgy which is a technique of controlled material cooling used to reduce defects

To use Simulated Annealing, the system must first be initialised with a particular configuration. The SA algorithm then starts with a random solution. With every iteration, a new random nearby solution is formed. If the solution is better, then it will replace the current solution.

If it is worse, then based on the probability of the temperature parameter, it may be chosen to replace the current solution. As the algorithm progresses, the temperature tends to decrease, thus making sure that worse solutions have a lesser chance of replacing the current solution.

Pseudo Code

```
Let s = s0
For k = 0 through kmax (exclusive):
T ← temperature(k / kmax)
Pick a random neighbour, snew ← neighbour(s)
If P(E(s), E(snew), T) ≥ random(0, 1), move to
the new state:
s ← snew
Output: the final state s
```



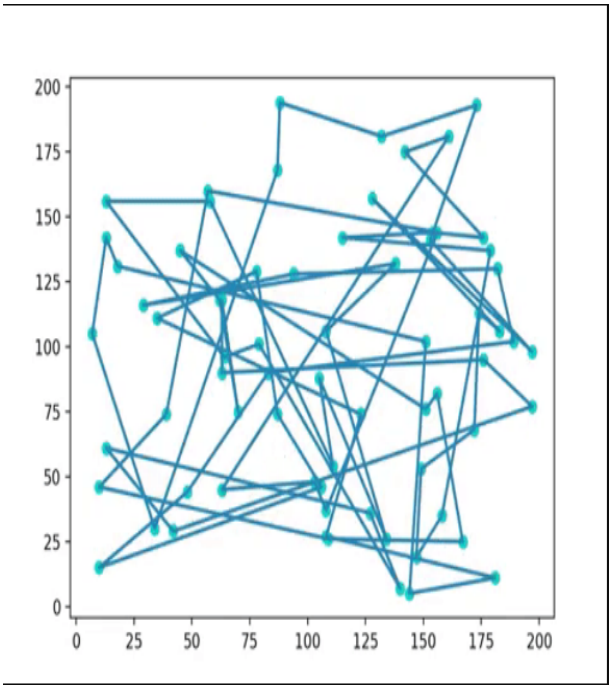
Flowchart of Simulated Annealing

As per the flowchart, if the energy of the new state is less than that of its previous one, then the change is accepted unconditionally and the system is updated. But if the energy is greater, than the latest configuration has the probabilistic chance or being accepted based on the temperature parameter.

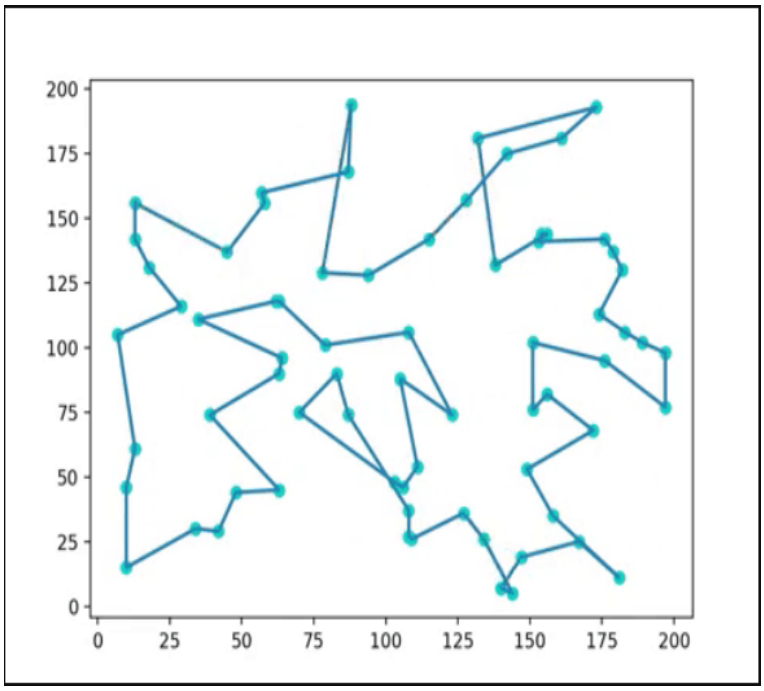
Algorithm 2 SA algorithm for TSP instance

$N \leftarrow$ number of cities
$D \leftarrow N \times N$ distance matrix
$s \leftarrow$ {initial tour} (random permutation of N)
$d_{curr} \leftarrow tourCost(s,D)$
$T \leftarrow$ temperature value
loop:
select 2 random numbers a,b between 1 to N
$s_{next} \leftarrow twoEdgeExchange(s,a,b)$
$d_{next} \leftarrow tourCost(s_{next},D)$
$E \leftarrow d_{curr} - d_{next}$
$p \leftarrow transitionProbability(E,T_i)$
if $E>0$ **then**
    $s \leftarrow s_{next}$
else
    $s \leftarrow s_{next}$ with probability p
return s, d_{curr}

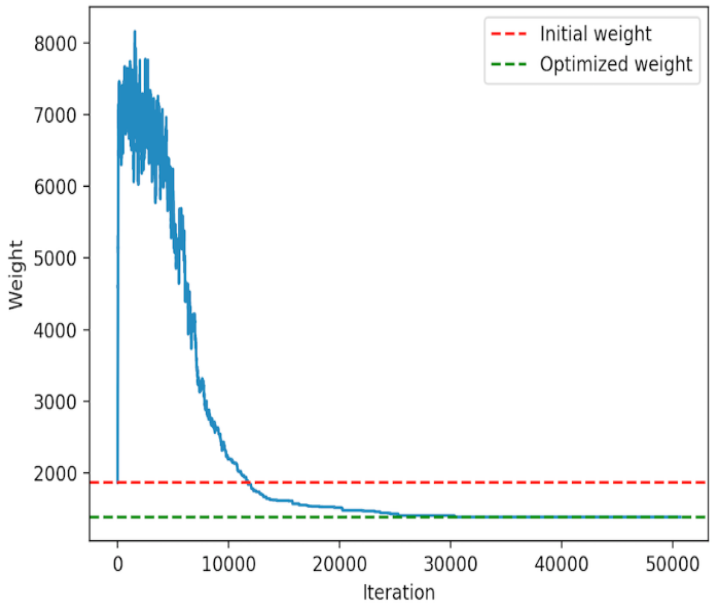
1) Output:



Before applying Simulated Annealing



After applying Simulated Annealing



Learning Progress Expressed as Distance of nodes over time

Week : 5 Learning Objective :

Game Playing Agent — Minimax — Alpha-Beta Pruning Systematic adversarial search can lead to savings in terms of pruning of sub-trees resulting in lesser node evaluations

Problem : 1
What is the size of the game tree for Noughts and Crosses? Sketch the game tree.

Tic-Tac-Toe

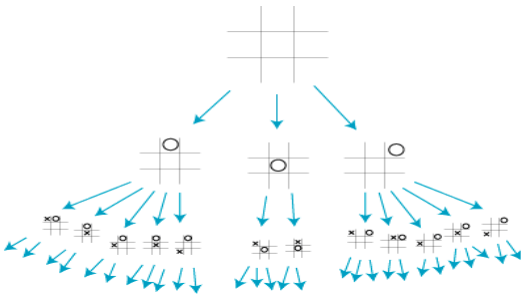
Noughts and Crosses, also known as Tic-Tac-Toe, is a simple two-player game played on a 3x3 grid. At the start of the game, the grid is empty, and one player places X's on the grid, and the other player places O's. The first player to get three of their marks in a row (either horizontally, vertically, or diagonally) wins the game. If all the squares are filled, and no player has three in a row, the game is a draw.

To calculate the size of the game tree, we can count the number of possible board configurations at each turn. In the first move, there are nine possible squares to place the first mark. In the second move, there are eight remaining squares to place the second mark, and so on. Therefore, the total number of possible board configurations is:

9 × 8 × 7 × 6 × 5 × 4 × 3 × 2 × 1 = 362,880

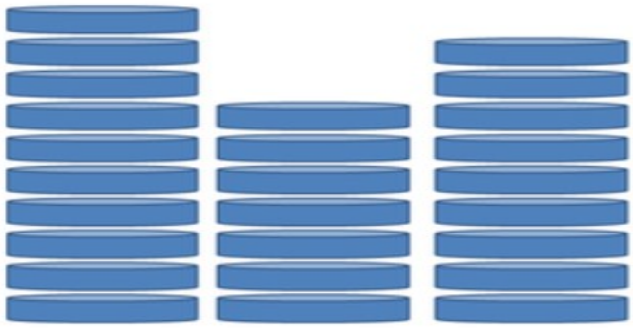
However, many of these board configurations are impossible, as they violate the rules of the game. For example, if a player has already won, it is impossible to make further moves. Similarly, if all the squares are filled, it is impossible to make any further moves. Therefore, the actual size of the game tree is much smaller than 362,880.

To sketch the game tree, we can start with the initial empty board, and then draw all possible moves and resulting board configurations. For simplicity, we will only show the first few moves, as the tree quickly becomes too large to draw by hand.



Problem : 2

Read about the game of Nim (a player left with no move losing the game). For the initial configuration of the game with three piles of objects as shown in Figure, show that regardless of the strategy of player-1, player-2 will always win. Try to explain the reason with the MINIMAX value backup argument on the game tree.



The game of Nim is a two-player game in which players take turns removing objects from piles. The game ends when there are no objects left on the board, and the player who takes the last object loses. It consists of 'n' piles where each pile contains i number of nims such that

$$1 \leq \mathrm{i} \leq n \quad (4)$$

The player to pick the last nim is going to win the game The winning in the Nim game depends on the two factor:

- One who starts the game
- Initial configuration of game

Here with observation, if initial configuration of the Nims i.e if the XOR sum of the no. of the Nims is non-zero then who starts the game first will definitely lose. Here in the question starting configuration is $[7, 9, 10]$ whose XOR is

$$4 \neq 0$$

Process of Playing

To analyze this game using the MINIMAX value backup argument, we can assign a score to each terminal state. A terminal state where player-1 wins has a score of +1, a terminal state where player-2 wins has a score of -1, and a terminal state where both players tie has a score of 0.

With this scoring system, we can compute the MINIMAX value of each node in the game tree. The MINIMAX value of a node represents the best possible outcome for the player who is making the move at that node, assuming that the other player plays optimally.

If we work our way up the tree using the MINIMAX value backup argument, we can see that the root node has a MINIMAX value of -1. This means that player-2 has a winning strategy for the initial configuration with three piles of objects, regardless of player-1's strategy.

The reason for this is that no matter how many objects player-1 removes from any pile, player-2 can always make a move that leaves the game in a winning position. In other words, player-2 can always force a win if they play optimally, regardless of what player-1 does.

Problem : 3

What is the size of the game tree for Noughts and Crosses? Sketch the game tree.

Size of the game tree for that the number of different states explored during finding of best move for a state in Minimax algorithm: 29

```
print("Total number of different states explored during finding of best move for a state = ",len(states.keys()))
```

```
Total number of different states explored during finding of best move for a state =  12
```

Total No. of state Visited without alpha beta pruning

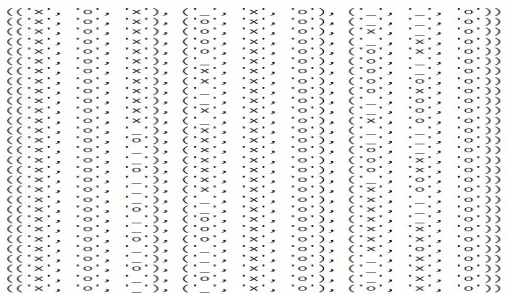
```
different states

(('x', 'o', 'x'), ('_', 'x', 'o'), ('_', '_', 'o'))
(('x', 'o', 'x'), ('o', 'x', 'o'), ('_', '_', 'o'))
(('x', 'o', 'x'), ('o', 'x', 'o'), ('x', '_', 'o'))
(('x', 'o', 'x'), ('o', 'x', 'o'), ('_', 'x', 'o'))
(('x', 'o', 'x'), ('o', 'x', 'o'), ('o', 'x', 'o'))
(('x', 'o', 'x'), ('_', 'x', 'o'), ('o', 'o', 'o'))
(('x', 'o', '_'), ('x', 'x', 'o'), ('o', '_', 'o'))
(('x', 'o', 'o'), ('x', 'x', 'o'), ('o', '_', 'o'))
(('x', 'o', '_'), ('x', 'x', 'o'), ('o', 'o', 'o'))
(('x', 'o', '_'), ('_', 'x', 'o'), ('o', 'x', 'o'))
(('x', 'o', 'o'), ('_', 'x', 'o'), ('o', 'x', 'o'))
(('x', 'o', '_'), ('o', 'x', 'o'), ('o', 'x', 'o'))
```

```
print("Total number of different states explored during finding of best move for a state = ",len(states.keys()))
```

```
Total number of different states explored during finding of best move for a state =  29
```

Total No. of state Visited without alpha beta pruning



Problem : 4

Use recurrence to show that under perfect ordering of leaf nodes, the alpha-beta pruning time complexity is $O(b^{m/2})$, where b is the effective branching factor and m is the depth of the tree.

The time complexity of Minimax algorithm is O(b^m) which can reduced by without checking each node. The technique of reduction in time complexity is called pruning. This involves two threshold parameters alpha and beta, thus this is also called alpha-beta pruning.

To determine exact value of a state, we need the exact value of one of its children and bounds on the rest of the children.For determining a bound of state's value, exact value of one of its children must be known.

Let S(k) be the minimum number of states to be considered k ply from a given state for knowing the exact value of the state. Similarly let R(k) be the minimum number of states to be considered k ply from a given state when we need to know a bound on the state's value.Let b be the branching factor. Thus,

$$S(k) = S(k-1) + (b-1)R(k-1) \tag{1}$$

i.e., the exact value of one child and bounds on the rest,and

$$R(k) = S(k-1) \tag{2}$$

i.e., the exact value of one child. The base is S(0) = R(0) = 1. Note, for fig, this gives S(3) = b^2 + b -1 = 11 for b = 3 when we expand the recursive equation, we get:

$$\begin{aligned} S(k) &= S(k-1) + (b-1)R(k-1) \\ &= (S(k-2) + (b-1)R(k-2)) + (b-1)S(k-2) \\ &= bS(k-2) + (b-1)R(k-2) \\ &= bS(k-2) + (b-1)S(k-3) \end{aligned}$$

It is obvious that S(k-3)¡S(k-2),so:

$$\begin{aligned} S(k) &< (2b-1)S(k-2) \\ S(K) &< 2bS(k-2) \end{aligned}$$

That is, the branching factor every two levels is less than 2b, which means the effective branching factor is less than $\sqrt{2b}$.

So, for even k, we derive

$$S(k) \leq (\sqrt{2b})^k,$$

which is not too far off the asymptotic upper bound of

$$(\sqrt{b} + \frac{1}{2})^{k+1}$$

In effect, alpha-beta pruning can nearly double the depth that a game tree can be searched in comparison to straightforward minimax.

GitHub Link - https://github.com/Anurag2-5/CS302-AI-OG-lab-report