Indain Institute of Information Technology, Vadodara

Artificial Intelligence CS362
MidSem Lab Report

Members:
Sagar Deware (202051166)
Yash Agrawal (202051205)
Bibhav Shah (202051212)
Sushil Patel (202051188)

Lecturer:
Dr. Pratik Shah

Academic Batch 2020-24

LAB-01

Artificial Intelligence

Group Members:

- Sushil kumar patel (202051188)
- Yash Agarwal (202051205)
- Bibhav Shah (202051212)
- Sagar Deware (202051166)

Problem Statement:

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.
B. Write a collection of functions imitating the environment for Puzzle-8.
C. Describe what is Iterative Deepening Search.
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.

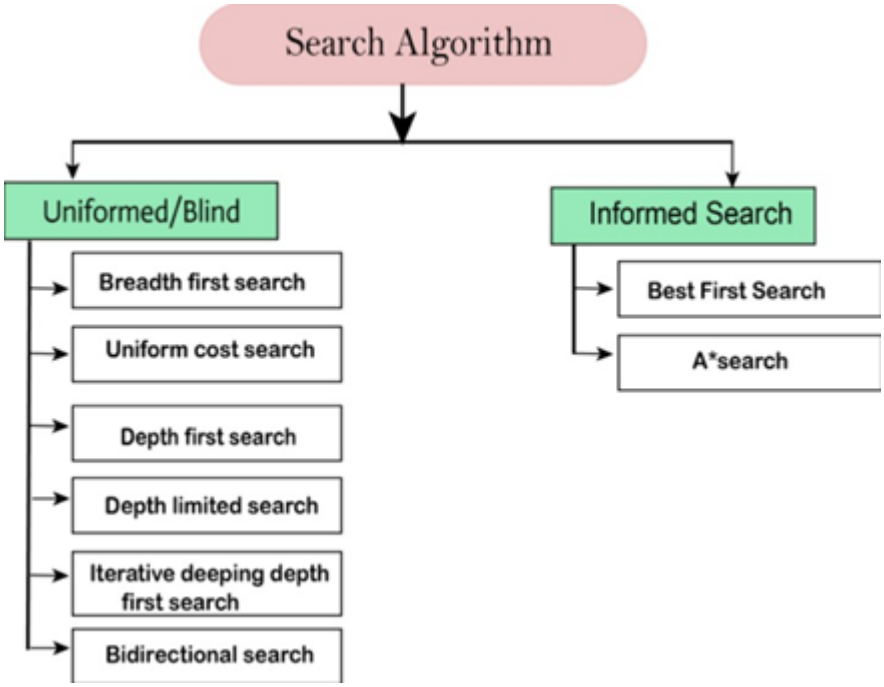
Concepts:

Un informed Search Algorithms: It is a class of general-purpose search algorithms which operates in brute force way . It does not have additional information about state or search space other than how to traverse the tree, so it is also called blind search .

Informed Search Algorithms: In an informed search algorithm, It stores knowledge such as how far we are from the goal, path cost, how to reach the goal in the form of set, array or table. This knowledge helps agents to explore less of the search space and find the goal node more efficiently.

The informed search algorithm is more useful for large search space . Informed search algorithms uses the idea of heuristic, so it is also called Heuristic search. Ex. N*N -1 Puzzle .

Heuristics function: It is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close the agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time . Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.



Analysis & Observations:

For problem A:

Here is a pseudocode for a graph search agent:

```
# we have the graph in which we want to search,
start / root and goal node.  function
graph_search(graph, start, goal):  # initialize
an empty list to store the path  path = []
  # initialize a queue to store the nodes
 that will be explored  queue = []
  # add the start node to the queue
 queue.append(start)
  # initialize a set to store
 the visited nodes  visited =
 set()

  # loop until the queue
 is empty  while queue:
    # remove the first node in the queue
   and mark it as visited  current_node
```

```
    = queue.pop(0)
    visited.add(current_node)
     # if the current node is the
    goal, return the path  if
    current_node == goal:
       return path
     # get the neighbors of
    the current node
    neighbors =
    graph[current_node]  #
    loop through the
    neighbors  for neighbor
    in neighbors:
 # if the neighbor has not been visited, add it to the
                                queue and update the path
       if neighbor not in visited:
        queue.append(neighbor)
        path.append((current_node,
        neighbor))
    #  if  the  queue  is  empty  and  the  goal  has
not  been  reached,  return  "Goal  not  found"  return
"Goal not found”
```

For problem B:

```
# initialize the puzzle
with a given state  def
initialize_puzzle(state
):
  # state is a list of integers representing the
 puzzle, with 0 representing the empty square
  # for example, state = [1, 2, 3, 4, 5, 6, 7, 8, 0]
 represents the puzzle in the solved state
  # store the state
 in a global
 variable  global
 puzzle  puzzle =
 state
```

```
 # return the current
state of the puzzle
def get_state():
   return puzzle

 # return a list of possible
actions given the current state
def get_actions():
   # get the index of
  the empty square
  empty_index =
  puzzle.index(0)
   # initialize a
  list of actions
  actions = []
   # if the empty square is not in the first
  column, the "left" action is possible  if
  empty_index % 3 > 0:

    actions.append("left
    ")
   # if the empty square is not in the last
  column, the "right" action is possible  if
  empty_index % 3 < 2:

    actions.append("righ
    t")
   # if the empty square is not in the first
  row, the "up" action is possible  if
  empty_index > 2:

    actions.append("up")
   # if the empty square is not in the last
  row, the "down" action is possible  if
  empty_index < 6:

    actions.append("down
    ")
```

```
    return actions

  # apply an action to the puzzle and
 return the resulting state  def
 apply_action(action):
    # get the index of
   the empty square
   empty_index =
   puzzle.index(0)
    # initialize a new state as a copy
   of the current state  new_state =
   puzzle.copy()
    #  perform  the  action  by  swapping  the  empty
 square  with  the  appropriate  neighboring  square  if
 action == "left":
new_state[empty_index],  new_state[empty_index  -  1]  =
                                 new_state[empty_index  -
  1], new_state[empty_index]   elif action == "right":
   new_state[empty_index],  new_state[empty_index  +  1]
   =  new_state[empty_index  +
  1],  new_state[empty_index]    elif  action  ==  "up":
   new_state[empty_index],  new_state[empty_index  -  3]
   =  new_state[empty_index  -
  3],  new_state[empty_index]   elif  action  ==  "down":
   new_state[empty_index],  new_state[empty_index  +  3]
   =  new_state[empty_index  +
  3],
 new_state[empty_index]
    # update the puzzle
   with the new state
   initialize_puzzle(new
   _state)  # return the
   new state  return
   new_state

  # check if the puzzle
 is in the solved state
 def is_solved():  return
```

```
puzzle == [1, 2, 3, 4,
5, 6, 7, 8, 0]
```

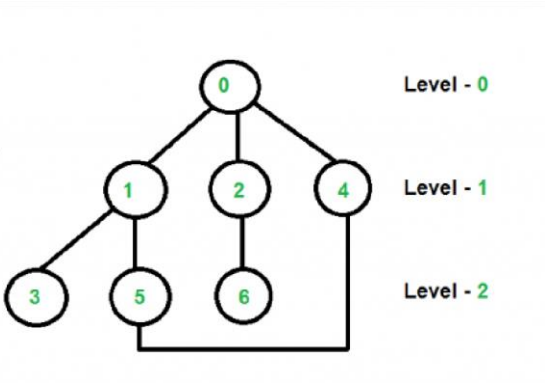
For Problem C:BFS: consumes more memory and less time DFS: consumes less memory and more time Iterative Deepening Search (IDS) is an iterative graph searching strategy that takes advantage of the completeness of the Breadth-First Search (BFS) strategy but uses much less memory in each iteration.

IDS achieves the desired completeness by enforcing a depth-limit on DFS that mitigates the possibility of getting stuck in an infinite or a very long branch.

It searches each branch of a node from left to right until it reaches the required depth. Once it has, IDS goes back to

the root node and explores a different branch that is similar to DFS.

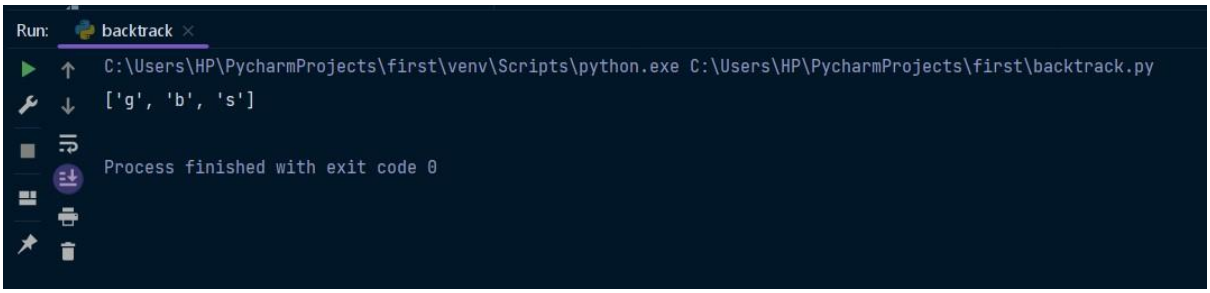
Depth	Iterative Deepening Depth First Search
0	0
1	0 1 2 4
2	0 1 3 5 2 6 4 5
3	0 1 3 5 4 2 6 4 5 1



For Problem D:

```
path = [('s', 'a'), ('s', 'b'), ('b', 'g')]
 def  func(startState, finalState, path):
curr = path.pop()[0]
finalPath = [finalState, curr]
 while  curr != startState:
 for  i  in  path:
 if  i[1] == curr:  curr = i[0]
  finalPath.append(curr)
   break
 return  finalPath
  print (func('s', 'g', path))
```

Output:



For problem E:

1 2 3

5 6 0

7 8 4

At depth 1 following are the instances:

1 2 3
5 0 6
7 8 4

1 2 0
5 6 3
7 8 4

1 2 3
5 6 4
7 8 0

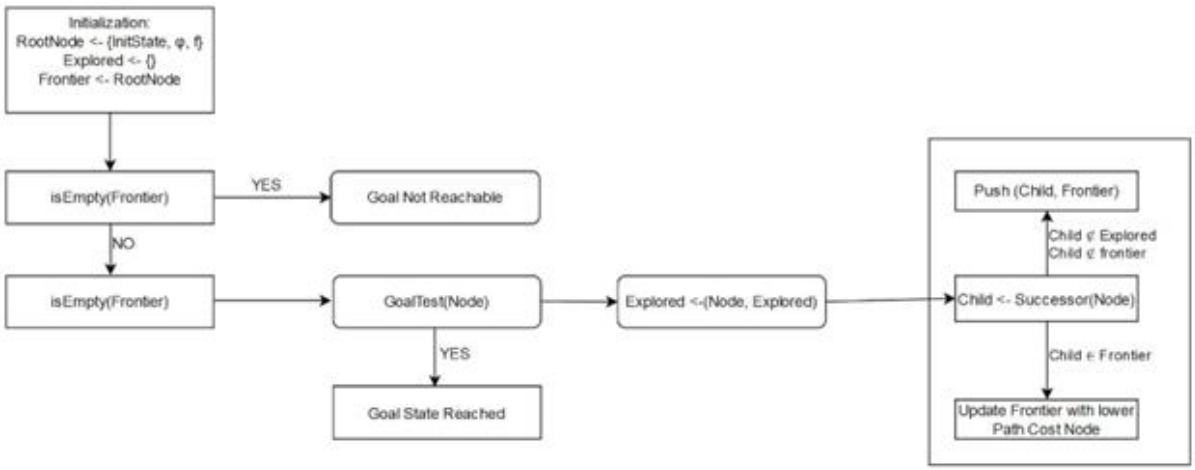
To generate Puzzle-8 instances with the goal state at depth "d", we can use the following approach:

1. Initialize the puzzle with the solved state [1, 2, 3, 4, 5, 6, 7, 8, 0] .
2. Randomly apply a series of actions to the puzzle, ensuring that the empty square is not moved out of the grid. The number of actions should be equal to "d".
3. Return the resulting puzzle state.

```
import
random  def
generate_puz
zle(d):
   # initialize the puzzle with
  the solved state
  initialize_puzzle([1, 2, 3, 4, 5,
```

```
6, 7, 8, 0])  # apply a series of
random actions to the puzzle
for i in range(d):
    actions =
   get_actions()
   action =
   random.choice(acti
   ons)
   apply_action(actio
   n)
 # return the resulting
puzzle state  return
get_state()
```

For Problem F:The space requirement for BFS search agent is $O(b^d)$ The time requirement for BFS search agent is $O(b^d)$ b - branch factor d - depth factor


Initialization:
RootNode <- {InitState, φ, f}
Explored <- {}
Frontier <- RootNode
isEmpty(Frontier)
YES
Goal Not Reachable
NO
isEmpty(Frontier)
GoalTest(Node)
YES
Goal State Reached
Explored <-(Node, Explored)
Child <- Successor(Node)
Child ∉ Explored
Child ∉ frontier
Child ∈ Frontier
Push (Child, Frontier)
Update Frontier with lower Path Cost Node

Lab 04 Report CS362 AI

Sagar Deware *202051166*
Yash Agrawal *202051205*
Bibhav Kumar shah *202051212*
Sushil Petal *202051188*

***Abstract*—Our goal is to use the Minimax algorithm to improvise a version of the game of tic-tac-toe while also using alpha beta pruning. We also want to explain the game of Nim, in which player 2 always prevails regardless of player 1's movement by using the minimax value backup parameter on the game tree.**

***Index Terms*—component, formatting, style, styling, insert**

I. Problem Statement

- What is the size of the game tree for tic-tac-toe? Sketch the game tree.
- 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
- Implement MINIMAX and alpha-beta pruning agents. Report on number of evaluated nodes for Noughts and Crosses game tree.
- 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

II. Introduction

- The recursive algorithm known as Minimax searches every conceivable node in the game and, depending on what that node needs, either returns the smallest or greatest value found in that node's subtree.
- Alpha-Beta Pruning is a technique for enhancing the Minimax algorithm that involves tracking each node's maximum possible values and eliminating the unnecessary nodes from the search.

III. Explanation

A. Tic-Tac-Toe

1) Theory: It's sometimes referred to as Xs and Os or Tic-Tac-Toe. The game has two players. The 33 grid is marked with an X or an O by each player in turn. If both players perform at their highest level, the game will end in a tie.

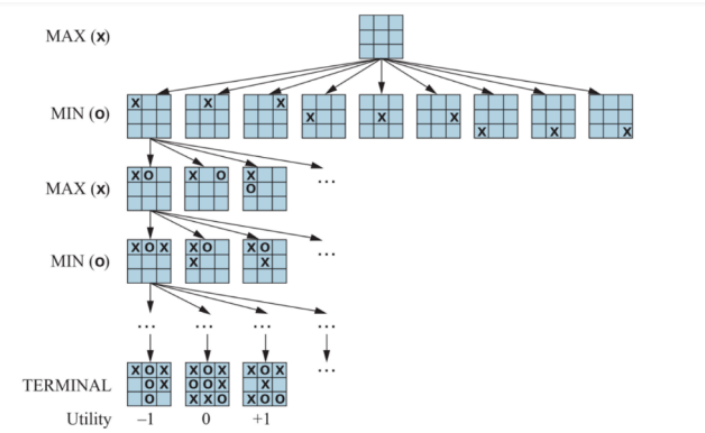


fig: 1 partial Game Tree for Tic-Tac-Toe

2) Game Tree and it's size: When the first player places X on a 3 by 3 empty grid, there are 9 possible choices for the first ply; when the second player moves and places O on the grid, there are 8 alternatives for each of the 9 possible states formed by the first ply. This entire series of two-ply strokes finishes one game move. Given the aforementioned observation, we may assert that a generic grid of size nm has (n+m) states for depth 1 and (n+m)(n+m-1) states for depth 2, and so on for further depths.

9 leaf nodes make up the game tree.

Game Tree Size = 9 + (9*8).... + 9!

Game Tree Size = 623529

3) Minimax and Alpha-Beta Pruning:In order to implement the Minimax and Alpha-Beta Pruning, we assume that the players execute their actions as effectively as possible. So, while using the Minimax Algorithm, we assessed 549946 nodes in total, whereas when using Alpha-Beta pruning, we evaluated 18297 nodes in total, which is far fewer than the nodes reviewed using the Minimax Algorithm.

B. The Game of Nim

1) Theory: Several stone stacks are used in the two-player game of Nim. We'll start with a few modest heaps of stones to make the game easier to comprehend, but you can use as many piles and as many stones in each pile as you like. Each player takes a stone out of the game in turn. The player that is taking out stones has a single pile from which they may take out as many as they want each time. If they so choose, they

could even take the entire pile out of the game. The one who removes the final stone is the winner.

Fig: 2 . Initial configuration of Game of Nim

2) Solution to given configuration: The default settings for the game are 10, 7, and 9. In contrast to the problem statement, which predicts that Player 2 would prevail, the initial setup favours Player 1 and it can win the game if played optimally (since "XOR" of the initial state is non-zero). As both players are constantly aware of every detail of the game, including the quantity of items in each pile, Nim is known as a game of perfect information. The XOR strategy for the game of Nim states that Player 2 will always win if Player 2 plays optimally if the initial configurations of the piles have a "XOR" value of "0".

3) Recurrence Relation for Alpha-Beta Pruning: In a tree with n nodes, let's define T(n) as the quantity of nodes that the alpha-beta pruning method visited. If the best child can be identified right away, we just need to analyse that youngster, and the rest can be eliminated. If the best child is not found right away, we must examine the next child and continue this process until we do. In the worst-case scenario, we must assess each of the b offspring. The best child, however, is likely to be in the first half of the group, so we only need to assess the first b/2 kids (on perfect ordering).

$$T(N) = 1 + \frac{b}{2} * T(\frac{N}{b})$$

Now we can solve this recurrence using the master theorem.

$$T(N) = a * T(\frac{N}{b}) + f(N)$$

where a = $\frac{b}{2}$, and f(N) = O(1). Thus, we have:

$$T(N) = O(b^m)$$

for Minimax algorithm but, or Alpha-Beta pruning the perfect ordering lets in reduction of nodes to be searched for half of the depth to only check a single node and prune others.

$$T(N) = O(b * 1 * b * 1...b = O(b^{m/2})$$

Time Complexity for Alpha-Beta Pruning for depth m and branching factor b under perfect ordering comes is $O(b^{m/2})$

IV. OBSERVATIONS

- The complexity of Minimax is $O(b^m)$, where as Alpha beta Pruning is $O(b^{m/2})$, where b is the effective branching factor and m is the depth of the game tree.
- We found that, when done systematically, the number of nodes investigated for Alpha Beta Pruning is much less than the number of nodes evaluated using just the Minimax approach.
- Systematic adversarial search can result in cost savings through sub-tree trimming that results in lower node assessments.
- In the Nim game, the nimsum is crucial. We need to find a mechanism to guarantee that the nimsum xor is always zero. The winner is the person who achieves nimsum xor 0. Player 2 will always prevail if the initial state has xor zero; if not, Player 1 will always prevail. We assumed that both players were performing at their highest level.

V. CONCLUSION

Alpha beta pruning proven to be more efficient than the Minimax algorithm when used systematically, according to our comparison of their effectiveness in the Game of Noughts and Crosses. We learned that if player 2 plays optimally in accordance with game theory, player 2 will always prevail in games of Nim regardless of player 1's actions.

REFERENCES

[1] Artificial Intelligence: a Modern Approach, Russell and Norvig (Fourth edition) (Chapter 5)

[2] A first course in Artificial Intelligence, Deepak Khemani (Chapter 8)

Lab 06 Artificial intelligence

Sagar Deware *202051166* | Yash Agrawal *202051205* | Bibhav Kumar shah *202051212* | Sushil Petal *202051188*

Learning Objective: To implement Expectation Maximization routine for learning parameters of a Hidden Markov Model, to be able to use EM framework for deriving algorithms for problems with hidden or partial information.

Problem Statement:

A. PART A

Read through the reference carefully. Implement routines forlearning the parameters of HMM given in section 7. In section8, "A not-so-simple example", an interesting exercise is carriedout. Perform a similar experiment on "War and Peace" by LeoTolstoy.

From the reference "A Revealing Introduction to Hidden Markov Models", we calculated α-pass, β-pass, di gammas for re-estimating the state transition probability matrix (A), observation probability matrix (B), initial state distribution (π) for Leo Tolstoy's book War and Peace. We re-estimated A,B and π based on the observed sequence , taking initial values for A, B and π and calculating α, β, the di-gammas and log probability for the data i.e. War and Peace.We took 50000 letters from the book after removing all the punctuation and converting the letters to lower case just as given in the example problem in section 8. We initialized each element of π and A randomly to approximately 1/2. The initial values are:

$$\pi = \begin{matrix} 0.51316 & 0.48684 \end{matrix}$$

$$A = \begin{matrix} 0.47468 & 0.52532 \\ 0.51656 & 0.48344 \end{matrix}$$

Each element of B was initialized to approximately 1/27. The precise values in the initial B are given.

After initial iteration,

$log([P(O|\lambda)]) = -142533.41283009356$

After 100 iterations,

$$log([P(O|\lambda)]) = -138404.4971796029$$

This shows that the model $\lambda = (A, B, \pi)$ has improved significantly. After 100 iterations the model converged to

$$\pi = \begin{matrix} 0.00000 & 1.00000 \end{matrix}$$

$$A = \begin{matrix} 0.28438805 & 0.71561195 \\ 0.81183208 & 0.18816792 \end{matrix}$$

with final values of transpose of B.

By looking at the B matrix we can see that the hidden state contains vowels and consonants and space is counted as a vowel. The first column of initial and final values are the vowels and the second column are the consonants.

	Initial		Final	
a	0.03735	0.03909	7.74626608e-02	6.20036245e-02
b	0.03408	0.03537	9.00050395e-10	2.34361673e-02
c	0.03455	0.03537	2.85482586e-08	5.54954482e-02
d	0.03828	0.03909	1.90968101e-10	6.91132175e-02
e	0.03782	0.03583	1.70719479e-01	2.11816156e-02
f	0.03922	0.03630	2.33305198e-12	2.99248707e-02
g	0.03688	0.04048	3.57358519e-07	3.08209307e-02
h	0.03408	0.03537	6.11276932e-02	4.10475218e-02
i	0.03875	0.03816	1.04406415e-01	1.70940624e-02
j	0.04062	0.03909	6.60956491e-26	1.92099741e-03
k	0.03735	0.03490	2.53743574e-04	1.21345926e-02
l	0.03968	0.03723	1.94001259e-02	4.17688047e-02
m	0.03548	0.03537	4.65877545e-12	3.85907034e-02
n	0.03735	0.03909	4.83856571e-02	6.14790535e-02
o	0.04062	0.03397	1.05740124e-01	4.23129392e-05
p	0.03595	0.03397	2.82866053e-02	1.84540755e-02
q	0.03641	0.03816	9.92576058e-19	1.32335377e-03
r	0.03408	0.03676	8.29107989e-06	1.07993337e-01
s	0.04062	0.04048	2.54927739e-03	9.75975025e-02
t	0.03548	0.03443	3.96236489e-05	1.50347802e-01
u	0.03922	0.03537	2.94555063e-02	8.54757059e-03
v	0.04062	0.03955	2.83949667e-19	3.05652032e-02
w	0.03455	0.03816	4.70315572e-25	3.37241767e-02
x	0.03595	0.03723	2.20419809e-03	1.38065996e-02
y	0.03408	0.03769	6.42635319e-04	3.04765034e-02
z	0.03408	0.03955	4.88081324e-27	1.10990961e-03
space	0.03688	0.03397	3.49317577e-01	4.28089789e-08

TABLE IX: Initial and final B transpose

B. PART B

Ten bent (biased) coins are placed in a box with unknown bias values. A coin is randomly picked from the box andtossed 100 times. A file containing results of five hundred such instances is presented in tabular form with 1 indicating head and 0 indicating tail. Find out the unknown bias values. (2020 ten bent coins.csv) To help you, a sample code for two bent coin problem along with data is made available in the work folder: two bent coins.csv and embentcoinsol.m

For the above problem, we used Expectation Maximization algorithm from the reference material already shared.An expectation–maximization (EM) algorithm is an iterative method to find (local) maximum likelihood or maximum a posteriori (MAP) estimates of parameters in statistical models,where the model depends on unobserved latent variables. In our case, the latent variables are z, the coin types. We have no knowledge of the coins but the final outcome of 500 trials. We used EM to estimate the probabilities for each possible completion of the missing data, using the current parametersϑ.

C. PART C

A point set with real values is given in 2020_em_clustering.csv. Considering that there are two clusters, use EM to group together points belonging to the same cluster. Try and argue that k-means is an EM algorithm. We used Expectation Maximization algorithm for grouping the points belonging to the same cluster. We also used *k-means* and compared the results of EM algorithm clustering and *k-means* clustering.

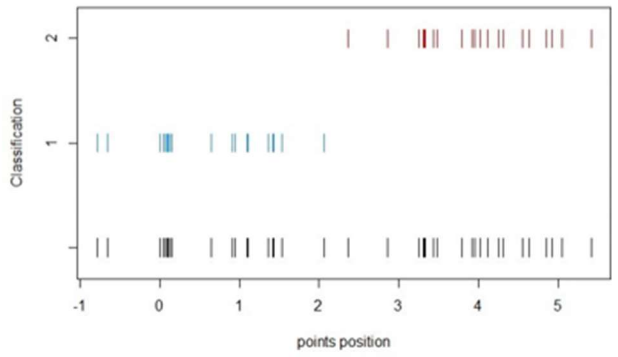


Fig. 24: EM clustering

In Fig 24, the lines depicted with black color are the points from the given dataset and the lines depicted with blue and redrespectively are the two clusters formed by the EM algorithm from the dataset. The model has grouped the points belonging to the same cluster.

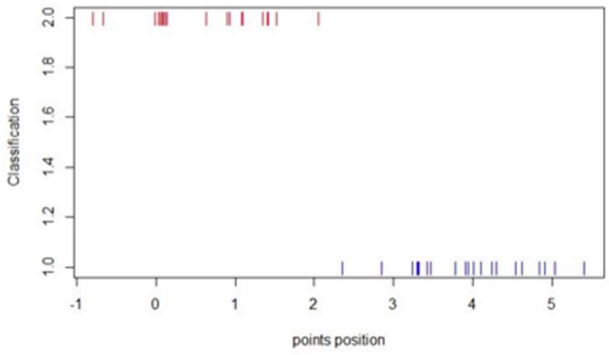


Fig. 25: k-means clustering

From the Fig 25, we can infer that *k-means* and EM algorithm results are same i.e. 19 points lie in the first cluster and 21 points lie in the second cluster out 40 given points.

k means uses hard-clustering which means that a point either belongs to the cluster or it does not belong the cluster. EM algorithm uses soft-clustering technique to assign points to a cluster, it gives the probability that a particular point belongs to a cluster. In this assignment, from the given dataset both the methods EM algorithm and *k means* yield the same result because the dataset was in such a way that the probability of a point belonging to a cluster was sufficient to assign the point to that cluster therefore the soft-clustering and hard-clustering yield the same result.

Lab 07 Artificial intelligence*

Sagar Deware
202051166

Yash Agrawal
202051205

Bibhav Kumar shah
202051212

Sushil Petal
202051188

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

I. Problem Statements

A. 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. 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. Extra Credit Challenge: Implement and compare MRF denoising with Stochastic denoising.

B. For the sample code hop field.m supplied in the lab-work folder, find out the amount of error (in bits) tolerable for each of the stored patterns.

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

II. Concepts

A. Markov random field

A Markov Random Field (MRF) is a statistical model used to represent the probability distribution of a set of random variables, where the dependencies between the variables are modeled by a graph. The graph is typically an undirected graph, where the nodes represent random variables, and the edges represent the dependencies between them. MRFs are commonly used in image processing, computer vision, and natural language processing.

Here is an example. Consider an image on a rectangular grid. It's composed of pixels. Each pixel has a value, denoting its color. Internally a pixel may be represented as a binary variable for black-and-white images, a continuous variable for grey-scale images, and use mixtures of multiple variables (e.g., RGB) for color images. These differences will not matter to us here. As far as we are concerned, a pixel has a value that represents its color.

B. Conditional Random Field

Conditional Random Fields (CRFs) are a type of probabilistic graphical model used for modeling structured data, such as sequential or spatial data, where the output depends on the input. CRFs are a variant of the more general Markov Random Fields (MRFs) where the focus is on modeling the conditional distribution of output variables given input variables, rather than the joint distribution of all variables.

In CRFs, the goal is to model the conditional probability of a set of output variables Y, given a set of input variables X. The input variables can be any type of data, such as text, images, or signals, while the output variables are usually categorical variables, such as labels, tags, or sequences.

CRFs are commonly used in natural language processing, computer vision, and bioinformatics, for tasks such as named entity recognition, part-of-speech tagging, image segmentation, and protein structure prediction.

One of the main advantages of CRFs is their ability to capture the dependencies between output variables, while taking into account the input variables. This makes them more suitable for structured prediction tasks compared to other methods, such as maximum entropy models or support vector machines.

III. Analysis and Observation

A. For Problem A

We used a benchmarking image for denoising algorithms, this 'page', which has a high dynamic range in gray scale pixels, with dimensions of 2019×1826 pixels. To prepare the image for Markov Random Field, we normalized the pixel

Identify applicable funding agency here. If none, delete this.

values between 0 and 1, then binarized them by setting all normalized pixel values below 0.5 to 0 and the rest to 1.

$$E(u) = \sum_{n=1}^{N}(u_n - v_n)^2 + \lambda \sum_{n=1}^{N-1}(u_{n+1} - u_n)^2$$

In order to test its effectiveness and capability, we introduced noise to the image and varied the level of noise. The Markov Random Field uses a quadratic potential function to determine the energy potential of an image when a pixel is modified in relation to its neighboring pixels. The quadratic potential function is defined by the smooth ID signal represented by v, the IID represented by u, and the energy function represented by E. As a result, there are a total of 5 × 191 × 384 iterations.

Please go through fig 01, 02,03,04.

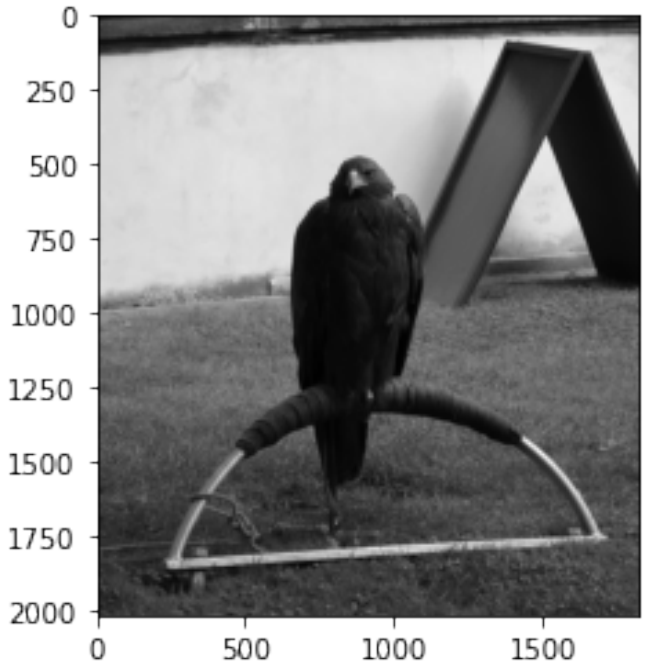


Fig. 1. Original Eagle Image

B. For Problem B

We computed the denoising percentage for each level of noise in the images. The original images are depicted in the figure. We trained the network using Hebb's rule, as described in the file, and then added noise by randomly altering some pixel values, with a maximum of 14 pixels being affected. Surprisingly, the network was able to correct up to 8 errors at most. The outcomes are presented in the figure below.

Please go through fig 05,06.

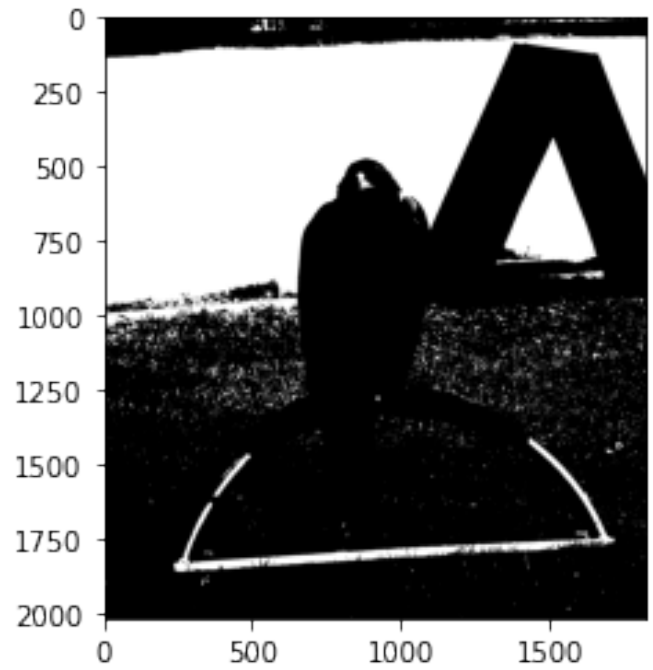


Fig. 2. Binarized Eagle image

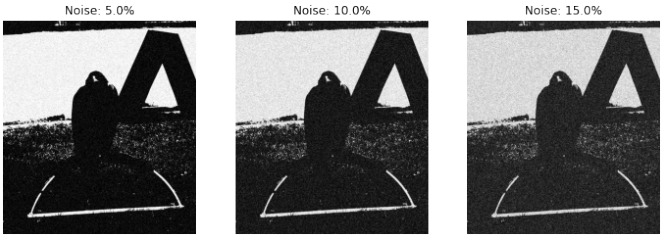


Fig. 3. Varing level of noise in image

C. For Problem C

The commonly known Traveling Salesman problem was solved utilizing Hopfield Networks. As each node in a Hopfield Network is connected to every other node, 100 weights were necessary for this problem with a 10 × 10 city grid. Initially, we randomly generated 10 cities and allowed the Hopfield Network to predict the optimal least-cost path.

Please go through fig 07,08.

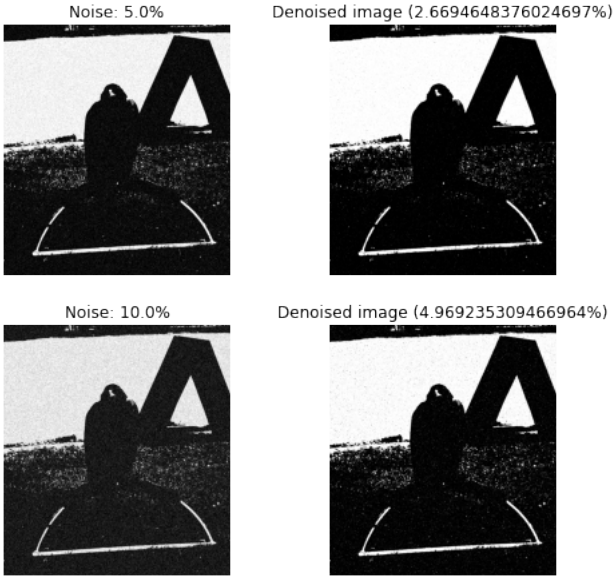


Fig. 4. Denoising image using MRF

Fig. 5. Original Letter for Hopfield network

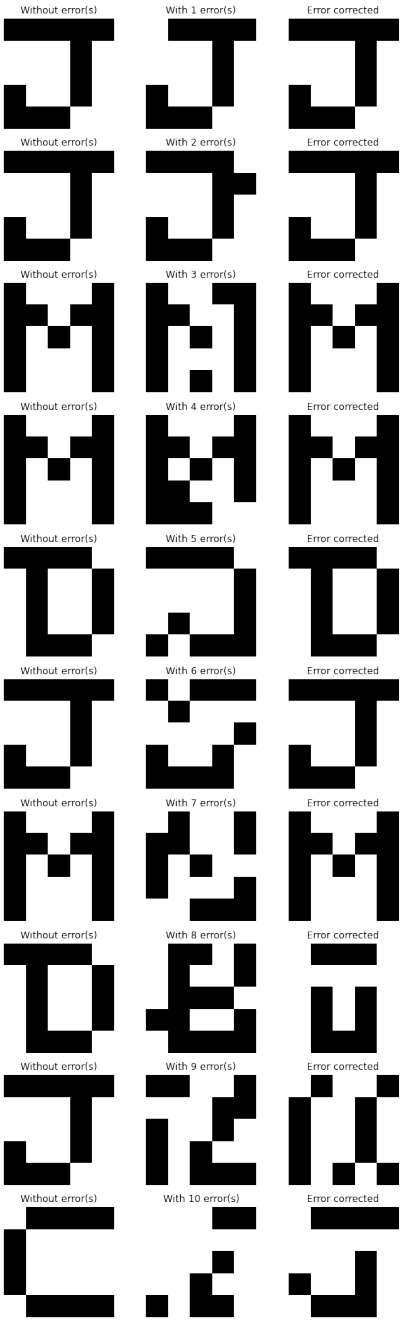


Fig. 6. The original, noised, and Corrected letters

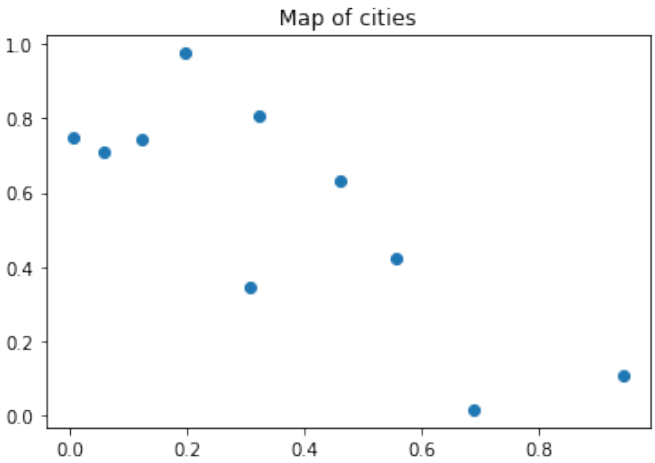


Fig. 7. Cities

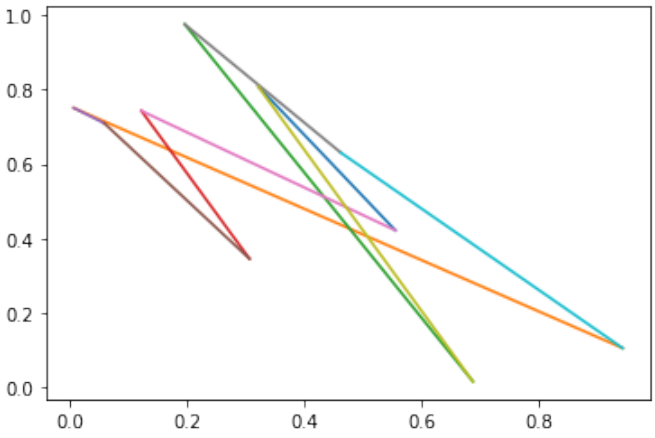


Fig. 8. Shortest path