First Lab Report CS 362 (Group 5 Naagraaj)

April 27, 2021
Submitted to: Prof Pratik Shah

Aditya Prakash[1], Anuj Puri[2], Ashutosh Singh[3], Pushkar Patel[4] and Yash Shah[5]
[1]201851008@iiitvadodara.ac.in [2]201851025@iiitvadodara.ac.in [3]201851029@iiitvadodara.ac.in
[4]201851094@iiitvadodara.ac.in [5]201851150@iiitvadodara.ac.in

CONTENTS

I. Introduction

In this Report, we have summarized our experiments, observations and results while performing 8 experiments given to us in the labs. We chose the following experiments for this report:

1) Lab Assignment 1: Graph Search Agent for 8-Puzzle
2) Lab Assignment 3: TSP using Simulated Annealing
3) Lab Assignment 4: Game Playing Agent — Minimax — Alpha-Beta Pruning
4) Lab Assignment 5: Building Bayesian Networks in R
5) Lab Assignment 6: Hidden Markov Model and Expectation Maximization algorithm
6) Lab Assignment 7: Markov Random Field and Hopfield Network
7) Lab Assignment 8: Markov Decision Process and RL
8) Lab Assignment 9: Understanding Exploitation and n-armed bandit reinforcement learning

Visualizing our results through tables and graphs helped us understand the results in a better and more convenient way, as well as formulate our results better. We hope the following discussion in the sections below help the reader understand these topics in a better light that what the they started with.

We performed the experiments primarily in Google Colab for Python Codes and RStudio for R. This report mainly discusses how we approached the problem, and the observations and results that we got from it. We have added links to our codes in this doc under the specific session, as well as at the end of this section.

Links to codes and graphs:

1) Colab Notebook for Graph Search Agent for 8-Puzzle
2) Drive folder for Graphs and Colab Notebook for TSP with Simulated Annealing
3) Colab Notebook for Game Playing Agent — Minimax — Alpha-Beta Pruning
4) Drive folder for Graphs and R Markdown file for Building Bayesian Networks in R
5) Drive folder for Colab Notebook for HMM and EM algorithm
6) Colab Notebook for Markov Random Field and Hopfield Network
7) Google Colab Notebook for MENACE code
8) Google Colab Notebook for N-arm Bandit

II. Week 1 Lab Assignment 1

Learning Objective: To design a graph search agent and understand the use of a hash table, queue in state space search. In this lab, we need prior knowledge of the types of agents involved and use this knowledge to solve a puzzle called 8-puzzle.

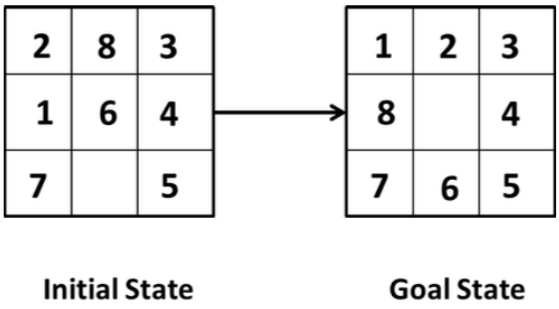


Fig. 1: Initial and final state of 8 puzzle[15]

An 8 puzzle is a simple game consisting of a 3 x 3 grid (containing 9 squares). One of the squares is empty and can be used to slide the other tiles in the grid. The objective is to move the tiles around into different positions to reach the configuration shown in "Goal State" in Fig 1.

The following discussion is based on the codes that were run here.

A. Part 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.

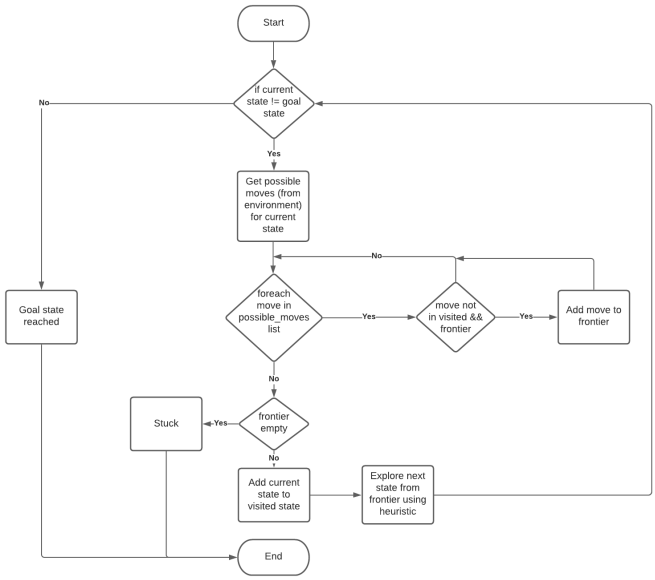


Fig. 2: Flowchart for Agent

Algorithm 1 8 puzzle

```
1: procedure BOOLEAN SOLUTION::SEARCH(PRIORITYQUEUE PQ,ARRAY VISITED)
2:     if pq.isEmpty() then return false
3:     puz ← pq.extract() //all possible successors to puz
4:     if search(pq) then return true
5:     for each suc in successors do
6:         if suc not in visited then
7:             pq.insert(suc).
8:             visited.insert(suc).
9:     if search(pq.visited) then return true
10:    else return false;
```

B. Part B

Write a collection of functions imitating the environment for Puzzle-8. Our code consists of following functions:

- **h1**: Heuristic for calculating the distance of goal state using Manhattan distance.

  Parameters:
  curr_state(np.ndarray): A 3x3 numpy array with each cell containing unique elements
  goal_state(np.ndarray): A 3x3 numpy array with each cell containing unique elements

  returns:
  h(int): Heuristic value

- **h2**: Heuristic for calculating the distance of goal state using number of misplaced tiles

  Parameters:
  curr_state(np.ndarray): A 3x3 numpy array with each cell containing unique elements
  goal_state(np.ndarray): A 3x3 numpy array with each cell containing unique elements

  returns:
  h(int): Heuristic value

- **generate_instance**: Heuristic for calculating the distance of goal state using number of misplaced tiles

  Parameters:
  goal_state(np.ndarray): A 3x3 numpy array with each cell containing unique elements representing the goal state depth(int): The depth at which the state is to be generated
  debug(bool): To print intermediate states and the heuristic values, or not. Default: False

  returns:
  curr_state(np.ndarray): A 3x3 numpy array with each cell containing unique elements representing the state at the given depth form, the goal state

- **get_possible_moves**: Function to get a list of possible states after valid moves form current state tiles

  Parameters:
  curr_state(np.ndarray): A 3x3 numpy array with each cell containing unique elements, representing the current states
  parent(string): The path taken to reach the current state from initial Arrangement

  returns:
  possible_moves(list): List of possible states after valid moves form current state
  possible_paths(list): List of possible paths after valid moves form current state

- **sort_by_heuristic**: Helper function to sort the next possible states based on heuristic value passed
  Parameters:
  possible_moves(list): List of possible states after valid moves form current state
  goal_state(np.ndarray): A 3x3 numpy array with each cell containing unique elements representing the goal state
  heuristic(Integer): An integer indicating the heuristic function to use. 1 for heuristic h1 and 2 for heuristic h2
  possible_paths(list): List of possible moves after valid moves form current state

  returns:
  sorted_possible_moves(list): List of possible states after valid moves form current state, sorted according to heuristic
  sorted_possible_moves(list): List of possible paths after valid moves form current state, sorted according to heuristic

- **solve**: Solves the puzzle by finding a path from current state to the goal state, using the heuristic provided. If no heuristic is provided, solves using normal BFS. Prints "GOAL REACHED!!!" if goal is reached and prints "STRUCK!!!" if no possible move is left
  Parameters:
  curr_state(np.ndarray): A 3x3 numpy array with each cell containing unique elements, representing the current state
  goal_state(np.ndarray): A 3x3 numpy array with each cell containing unique elements representing the goal state
  heuristic(Integer): An integer indicating the heuristic function to use. 1 for heuristic h1 and 2 for heurostic h2. If not provided or passed as 0, solves using normal BFS

  returns:
  sorted_possible_moves(list): List of possible states after

valid moves form current state, sorted according to heuristic
sorted_possible_moves(list): List of possible paths after valid moves form current state, sorted according to heuristic

C. Part C

Describe what is Iterative Deepening Search.

Iterative deepening depth first search (IDDFS) is a hybrid of BFS and DFS. In IDDFS, we perform DFS up to a certain "limited depth," and keep increasing this "limited depth" after every iteration.
Basically it performs DFS at every depth thus reducing the space complexity compared to BFS. This is illustrated in Fig 3.

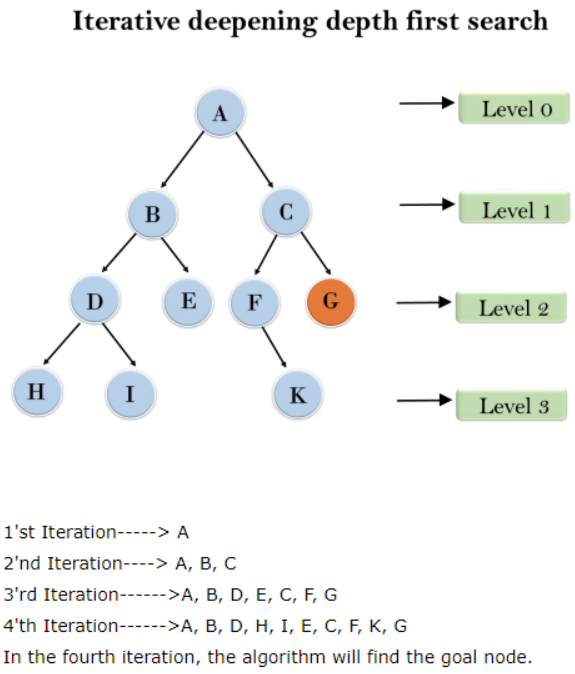


Fig. 3: Iterative Deepening Search

Let's suppose b is the branching factor and depth is d then

- **Time Complexity:**

$$O(b^d) \tag{1}$$

- **Space Complexity:**

$$O(bd) \tag{2}$$

D. Part 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.

We utilized the function "solve2" and "get_possible_moves" function to find the path. The "get_possible_moves" function finds the path from the start state to the goal state. The proper implementation for this can function can be found in the GitHub repository and Google Colab notebook.

E. Part E

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

We created a function "generate_instances" to create goal state at depth "d". The function takes the goal state and depth as input, generates a random instance of the 8-puzzle at the given depth from the goal state and return a 3x3 numpy array with each cell containing unique elements representing the state at the given depth form the goal state. The proper implementation for this can function can be found in the GitHub repository and Google colab notebook.

F. Part F

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

We used the memory_profiler package available in python to check the memory requirement of the program for each depth. The package provides decorators which can be used to map the memory used by a function. We summarized the results in table I, II and III.

Depth	Time (sec)	Memory (KiB)
2	0.008	52668
4	0.063	52780
8	0.661	52968
16	163.2	61904
32	865.7	72660

TABLE I: Time and memory requirement for 8 puzzle using Manhattan

Depth	Time (sec)	Memory (KiB)
2	0.015	52756
4	0.144	52680
8	0.615	52740
16	174.1	61120
32	3586	71270

TABLE II: Time and memory requirement for 8 puzzle using misplaced tiles

Depth	Time (sec)	Memory (KiB)
2	0.016	52936
4	0.136	52732
8	1.194	53052
16	208.6	60848
32	3600+	??

TABLE III: Time and memory requirement for 8 puzzle using no heuristic

III. Week 3 Lab Assignment 3

Learning Objective: Non-deterministic Search | Simulated Annealing
For problems with large search spaces, randomized search becomes a meaningful option given partial/full-information about the domain.

Problem Statement: 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.

The Google Colab Python notebook with the code, along with the full resolution images of the graphs for this assignment can be found **here**.

Consider visiting 25 random nodes/points

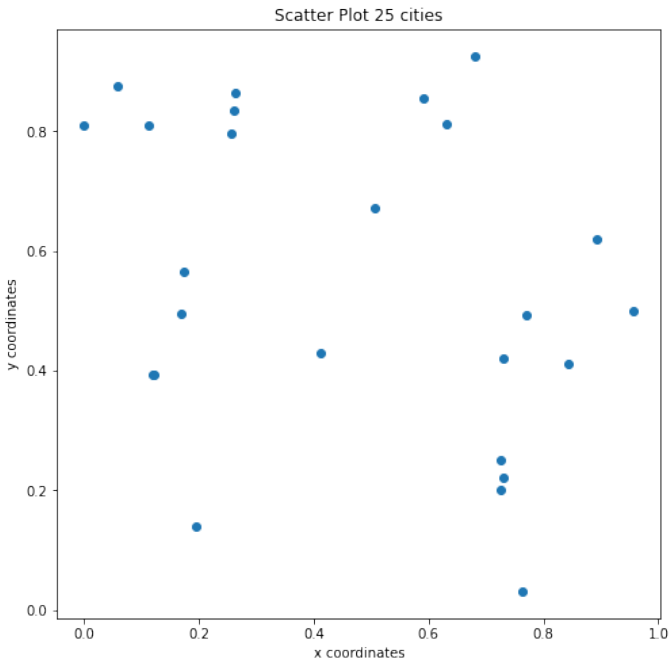


Fig. 4: Scatter Plot for 25 nodes

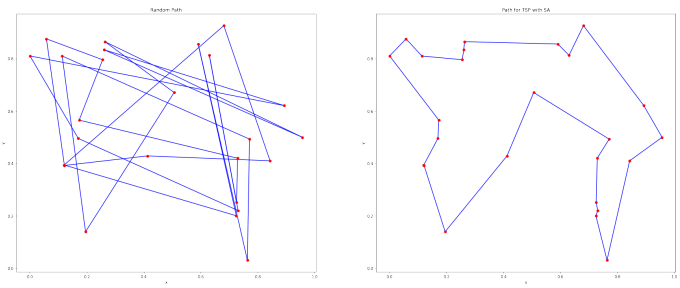


Fig. 5: Comparison between the routes for 25 nodes

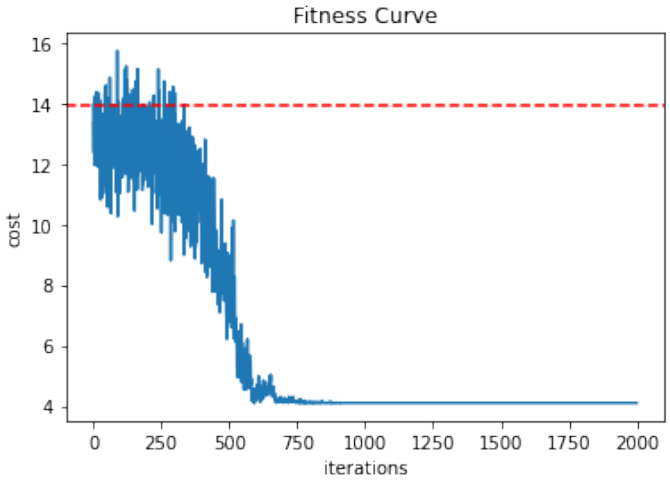


Fig. 6: Fitness curve for 25 nodes

The second plot in Fig. 5 shows the optimal path to cover all the nodes and return to the starting node using simulated annealing to reduce the cost.

Fig. 6 shows how simulated annealing reduces the cost with each iteration. Fitness curve shows the behaviour of the cost w.r.t to the no. of iterations we are running to obtain the optimal path. At first the cost is higher than the random path cost which eventually reduces after successive iterations and as soon as the temperature is low, then it is harder to accept worst solution cost, therefore the cost move towards optimal cost with decrease in temperature [6], curve becomes stable after some particular amount of iterations from which we can infer that there are very small changes in the cost and we can say that the optimal/sub-optimal cost is reached.[5]

A. Part 1

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.

We selected 25 locations for us to visit in Rajasthan, then we calculated euclidean distance for all pair of coordinates for all the locations and plotted a random route connecting all the locations. Using the simulated annealing to reduce the route cost, we calculated the optimal/sub-optimal path to visit the locations as shown in Fig. 8.

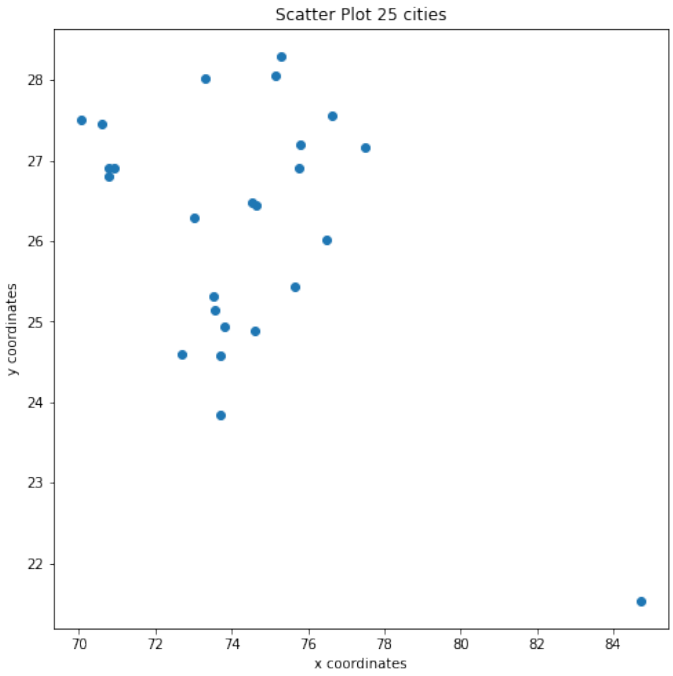


Fig. 7: Scatter Plot for 25 locations in Rajasthan

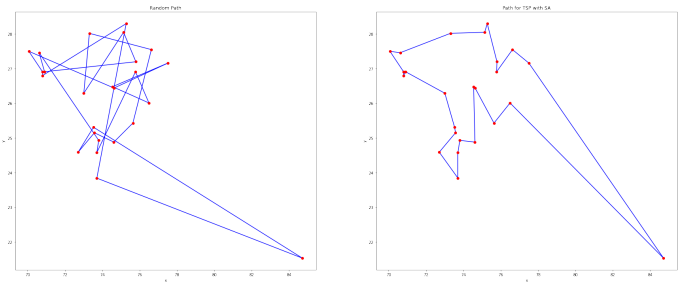


Fig. 8: Optimal path for 25 locations in Rajasthan

1) XQF131 - 131 points

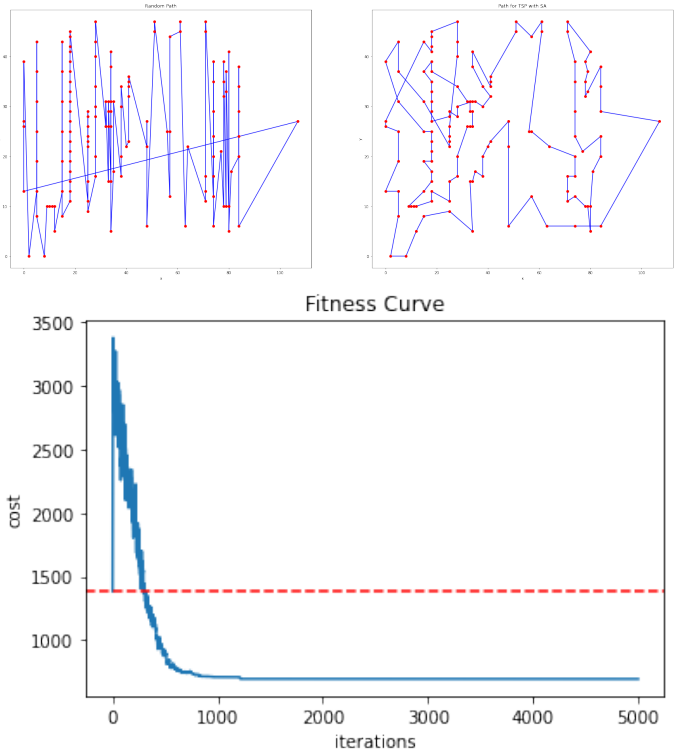


Fig. 9: Plots for 131 points

B. Part 2

VLSI: datasets

Attempt at least five problems from the above list and compare your results.

Repeating the same flow for finding an optimal/sub-optimal route as in the case for Rajasthan tourist locations part using datasets from VLSI i.e. 131 points, 237 points, 343 points, 379 points, 380 points.

2) XQG237 - 237 points

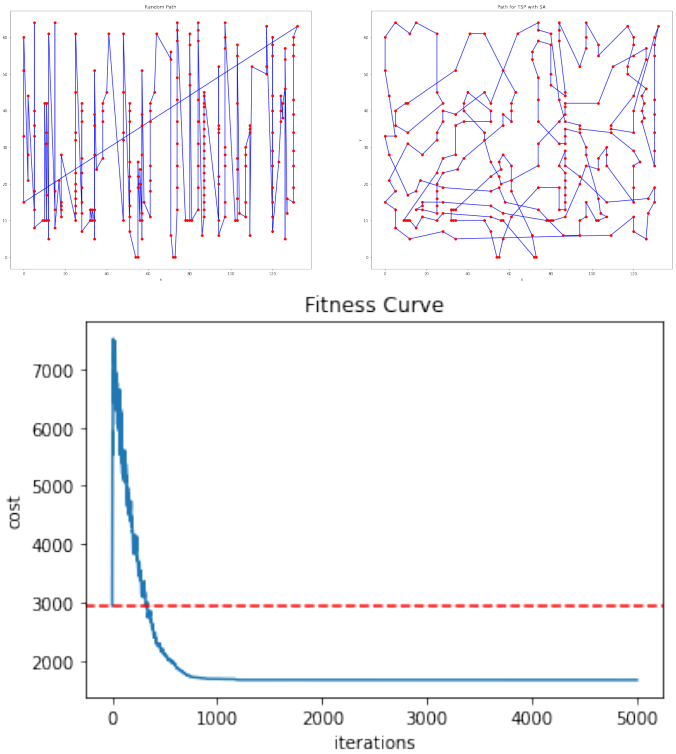


Fig. 10: Plots for 237 points

3) PMA343 - 343 points

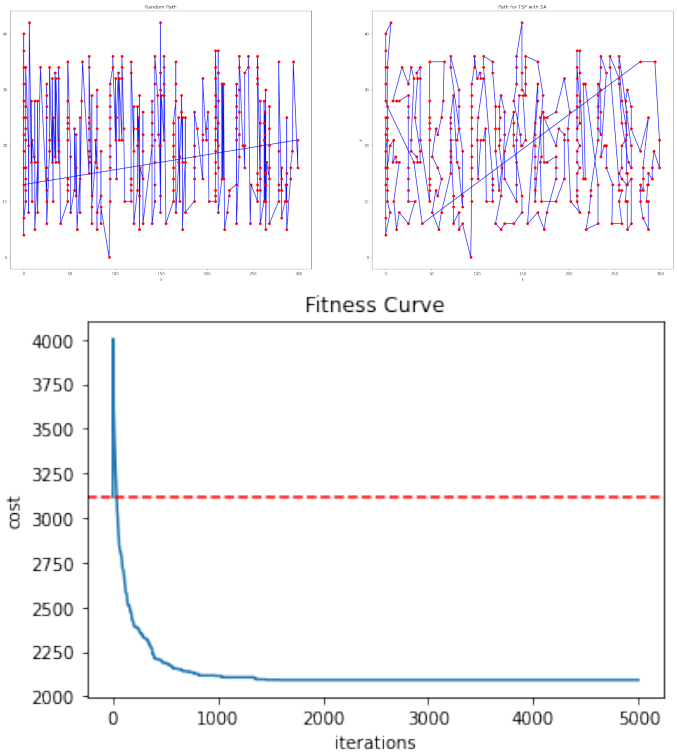


Fig. 11: Plots for 343 points

4) PKA379 - 379 points

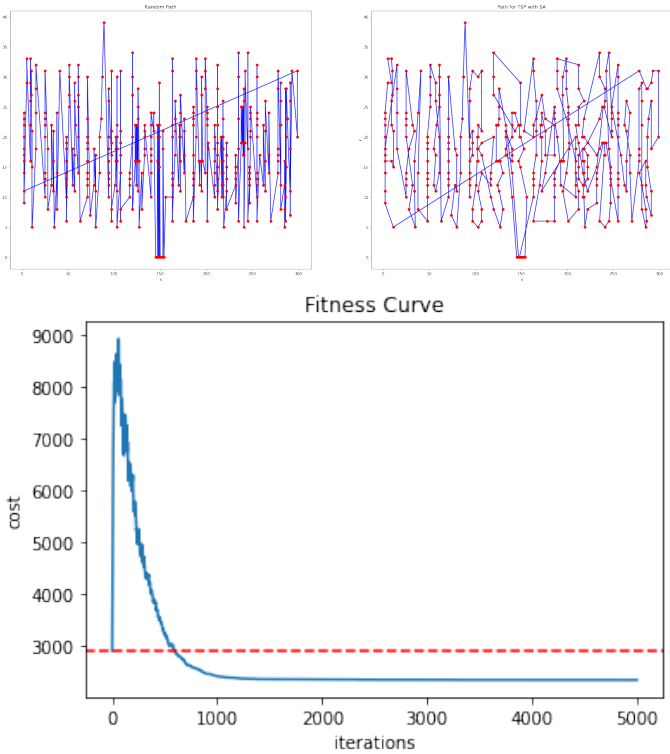


Fig. 12: Plots for 379 points

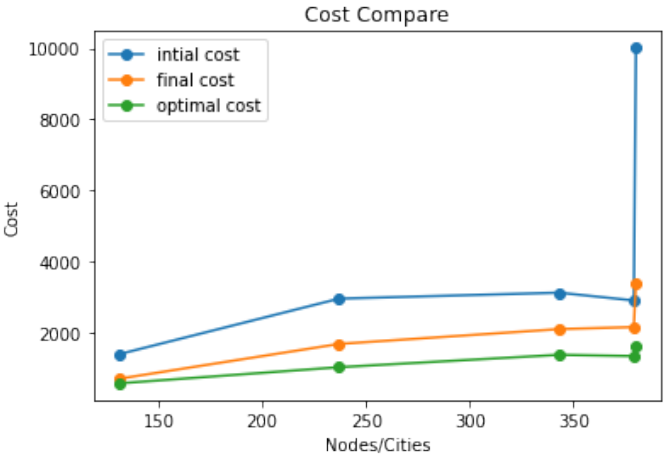


Fig. 13: Comparing costs

5) BLC380 - 380 points

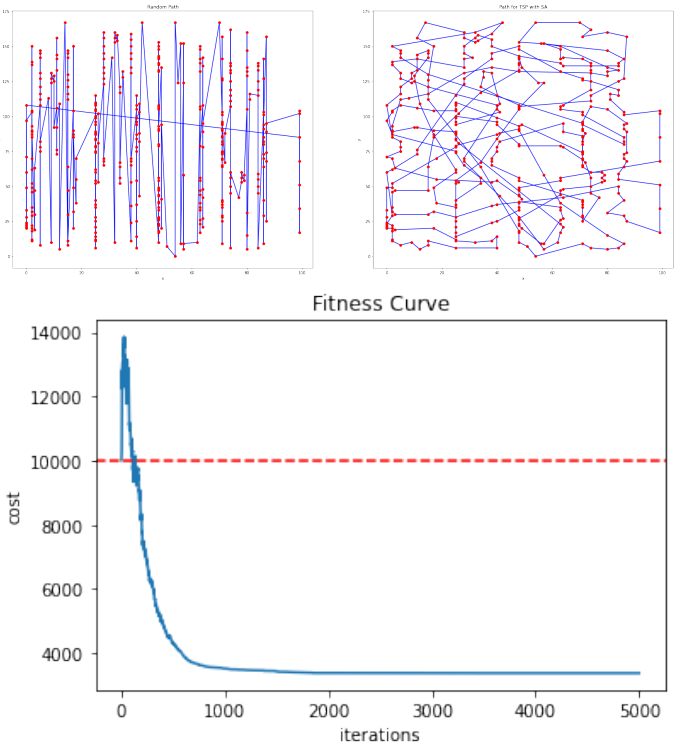


Fig. 14: Plots for 380 points

6) Comparing Results

In Fig. 13 we can see that the final cost or the cost for route calculated using simulated annealing is smaller than the cost for random route. When the comparing the cost with the optimal cost from VLSI logs of computation for each data set, we can see that our calculated final cost is comparable with the optimal cost for data points like 131 and 237.

We can further decrease the cost by increasing the number of iterations and using heuristic functions to solve the traveling salesman problem.

Points	Initial Cost	Final Cost	Optimal Cost
131	1383.916	693.312	564
237	2949.579	1673.994	1019
343	3117.179	2091.522	1368
379	2898.213	2148.960	1332
380	10013.540	3378.900	1621

TABLE IV: Cost Table

IV. Week 5 Lab Assignment 4

Learning Objective: Game Playing Agent | Minimax | Alpha-Beta Pruning

A. Part A

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

In the game of Noughts and Crosses, assume Player 1 is 'X' and Player 2 (computer) is 'O'. Then if Computer goes First we have 9! possible moves i.e., 362,880 possible moves, and if computer goes second we have 8! possible moves i.e., 40,320 possible moves.

At depth = 1, we can have 9 different states.
At depth = 2, we can have 9*8 different states.
At depth = 3, we can have 9*8*7 different states,
and so on till
At depth = 9, we can have 9! different states.

Total different states available will be :
$9 + 9 * 8 + 9 * 8 * 7 + ... + 9! \approx 10^6$

For Graph tree check Fig: 15

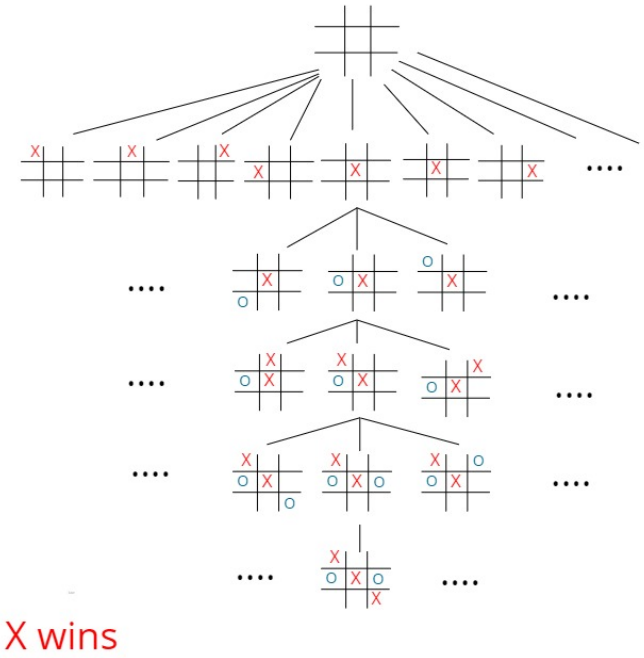


Fig. 15: Game tree for one state

Here Player 'X' wins.

B. Part B

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.

Check Fig : 16

Fig. 16: Nim Game Initial Configuration

With Minimax Algorithm both the players will try to maximize their chances of wining which leaves everything to the number of tower, since any number of tiles can be picked from a single tower at once, we can check

$$M(depth) = \begin{cases} 0 & \text{if current player loses} \\ -1 & \text{if tiles still remain} \\ 1 & \text{if current player wins} \end{cases} \quad (3)$$

If Player 1 always move such that XOR of three towers is equal to 0 then this will result in player 1 always winning no matter what player 2 does.[13] For example in Fig : 17

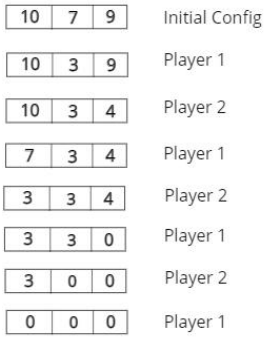


Fig. 17: Right section represents move played by player 1 or Player 2

C. Part C

Implement MINIMAX and alpha-beta pruning agents. Report on number of evaluated nodes for Noughts and Crosses game tree.

We ran this code on Google Colab which can be found **here**.

Initial State	Minimax	Alpha-beta pruining
empty-state	549946	18297

TABLE V: Evaluated States

D. Part D

Using recurrence relation 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.

Best case : Each player's best move is the left-most alternative (i.e., evaluated first)

Let m be the depth of the tree and b be the effective branching factor.

T(m) be the minimum number of states to be traversed to find the exact value of the current state, and

K(m) be the minimum number of states to be traversed to find the bound on the current state.

we determine the recursive relation for the alpha- beta pruning as :

$$T(m) = T(m-1) + (b-1)K(m-1) \tag{4}$$

[14] here T(m-1) is to traverse to child node and find the exact value, and (b-1)K(m-1) is to find min/max bound for the current depth of the tree.

In best case, we know that

$$T(0) = K(0) = 1 \tag{5}$$

we can also say that to the exact value of one child is

$$K(m) = T(m-1) \tag{6}$$

Now we expand the recursive relation and solve it further we get :

$$T(m) = T(m-1) + (b-1)K(m-1) \tag{7}$$

$$T(m-1) = T(m-2) + (b-1)K(m-2) \tag{8}$$

$$T(m-2) = T(m-3) + (b-1)K(m-3) \tag{9}$$

and so on till

$$\text{T(1) = T(0) + (b-1)K(0) = 1 + b-1 = b} \tag{10}$$

solve for T(m) from 7, 8, 9, we get

$$T(m) = T(m-2)+(b-1)K(m-2)+(b-1)K(m-1) \tag{11}$$

$$T(m) = T(m-3) + (b-1)K(m-3) + (b-1)K(m-2)$$
$$\text{+ (b-1)K(m-1)} \tag{12}$$

using 6 on 11,

$$T(m) = T(m-2) + (b-1)T(m-3) + (b-1)T(m-2)$$
$$\text{= b(T(m-2)) + (b-1)T(m-3)} \tag{13}$$

we can clearly say that

$$T(m-2) > T(m-3) \tag{14}$$

therefore,

$$T(m) < (2b-1)T(m-2) \tag{15}$$

since b can be really large we can say b-1 $\approx$ b, so

$$T(m) < 2bT(m-2) \tag{16}$$

That is, the branching factor every two levels is less than 2b, which means the effective branching factor is less than $\sqrt{2b}$. [14]

$$T(m) \leq \sqrt{b}^{m} \tag{17}$$

which is same as $b^{m/2}$.

V. Week 5 Lab Assignment 5

Learning Objective: Understand the graphical models for inference under uncertainty, build Bayesian Network in R, Learn the structure and CPTs from Data, naive Bayes classification with dependency between features.

Problem Statement: A table containing grades earned by students in respective courses is made available to you in (codes folder) 2020_bn_nb_data.txt.

The R Markdown file, having the code, along with full-resolution graphs can be found **here**. The below sections discusses our observations and results while performing the experiments and has graphs from the same.

A. Part 1

Consider grades earned in each of the courses as random variables and learn the dependencies between courses.

Using the hill climbing greedy search function 'hc' of the 'bnlearn' package in R, we can learn the structure of the Bayesian network and the dependencies between grades obtained in different courses.
Hill climbing is a score-based structure learning algorithm. Since our dataset is categorical in nature, we two scores learn the discrete Bayesian network - k2 and bic - and compare both of them. [11]

1) Using k2 score

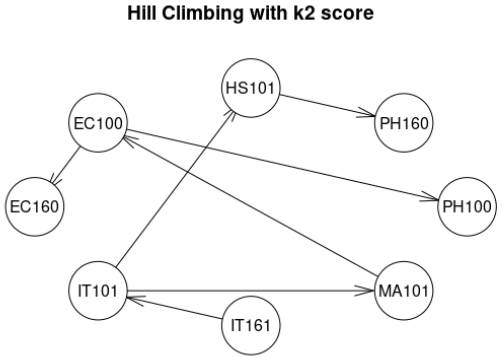


Fig. 18: Hill climbing Bayesian network using the k2 score

There are 7 arcs in the Fig. 18 and the model string showing this dependency is:
'[IT161] [IT101|IT161] [MA101|IT101] [HS101|IT101] [EC100|MA101] [PH160|HS10] [EC160|EC100] [PH100|EC100]'

2) Using bic score

There are only 2 arcs in the Fig. 19 and the model string showing this dependency is:
'[EC100] [EC160] [IT101] [IT161] [PH160] [HS101] [MA101|EC100] [PH100|EC100]'

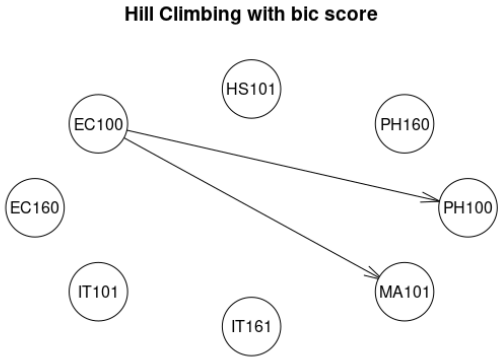


Fig. 19: Hill climbing Bayesian network using the bic score

Since we are interested to find the dependency of the different grades, we can see that the k2 score gives a better idea of how the scores are dependent on each other. Moreover, k2 is generally considered a good choice for large datasets. [10] So, for the further parts, we'll only use the network learned using the k2 score.

B. Part 2

Using the data, learn the CPTs for each course node.

According to the network learned using the k2 score, we plot the conditional probabilities as shown in Fig. 20.

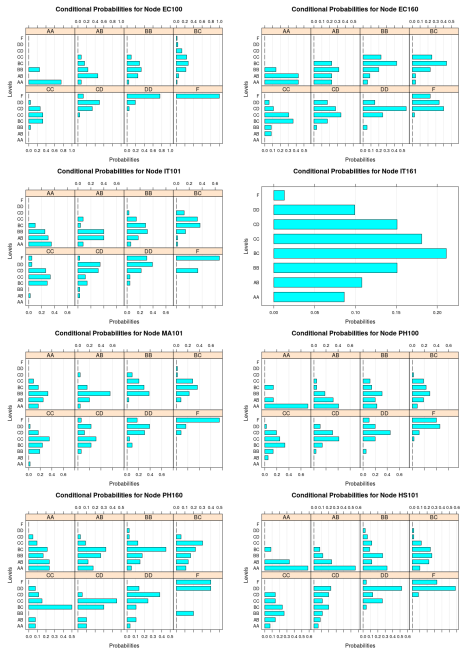


Fig. 20: Conditional probability distributions made form the CPTs of the learned data

The full-sized distribution graphs of Fig. 20 can be found here.

C. Part 3

What grade will a student get in PH100 if he earns DD in EC100, CC in IT101 and CD in MA101.

We use the 'cpdist' function of the 'bnlearn' package in R to get the distributioin of grades of PH100 when the student has earned DD in EC100, CC in IT101 and CD in MA101. The distribution graph is shown in Fig. 21 and the corresponding distribution table is as shown in Tab. VI.

According to this, the student seems most likely to receive a CD grade.

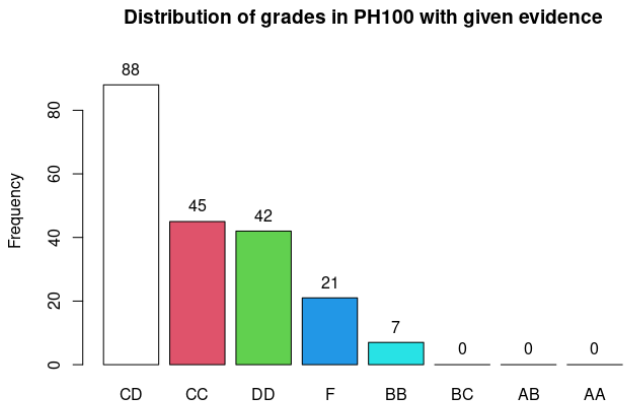


Fig. 21: Probability distribution of getting a grade in PH100 as per the given evidence

Grade	Frequency	Percent	Cum. percent
CD	101	43.5	43.5
DD	49	21.1	64.7
CC	47	20.3	84.9
F	23	9.9	94.8
BB	12	5.2	100
BC	0	0	0
AB	0	0	0
AA	0	0	0
Total	232	100	100

TABLE VI: Frequency distribution table

D. Part 4

The last column in the data file indicates whether a student qualifies for an internship program or not. From the given data, take 70 percent data for training and build a naive Bayes classifier (considering that the grades earned in different courses are independent of each other) which takes in the student's performance and returns the qualification status with a probability. Test your classifier on the remaining 30 percent data. Repeat this experiment for 20 random selection of training and testing data. Report results about the accuracy of your classifier.

We begin by first splitting the dataset into training and test datasets. We have a total of 231 data examples and we split them, into train and test sets of 162 and 69 data examples, respectively.

We use the Naive Bayes Classifier using the function 'nb' from the package 'bnclassify' in R tolearn the naive Bayes network structure and then the function 'lp' to learn the parameters. Here, we do not assume any dependency between the grade nodes, so the classifier learns assuming the data to be independent. The classifier learns the structure as shown in Fig 22.

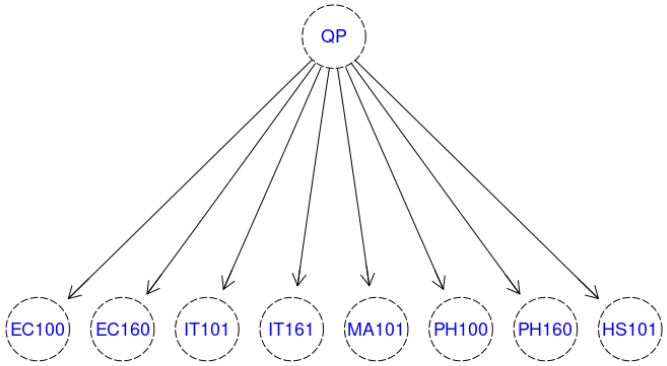


Fig. 22: Naive Bayes Classifier for independent data

This network learns on the training dataset that we split earlier, and we test it on the test dataset. Repeating this 20 times, we get the accuracy as shown in Tab VII.

Experiment no.	Accuracy
1	0.9130435
2	0.942029
3	0.9565217
4	0.942029
5	0.9565217
6	0.9710145
7	0.9710145
8	0.9855072
9	0.9855072
10	0.9565217
11	0.9710145
12	0.9710145
13	0.9714286
14	0.9855072
15	0.942029
16	0.9710145
17	0.942029
18	0.942029
19	0.9857143
20	0.9857143

TABLE VII: Accuracy on test dataset on Naive Bayes classifier considering independent data

E. Part 5

Repeat 4, considering that the grades earned in different courses may be dependent.

To learn the dependent features in out network, we use the 'tan_chow' function of the 'bnlearn' library in R. This function learns a one-dependence Bayesian classifier using Chow-Liu's algorithm. [12] Then, we go on to learn the paramaters using the 'lp' function, on the training dataset, so that the classifier learns from the dependent data.The classifier learns the structure as shown in Fig 23.

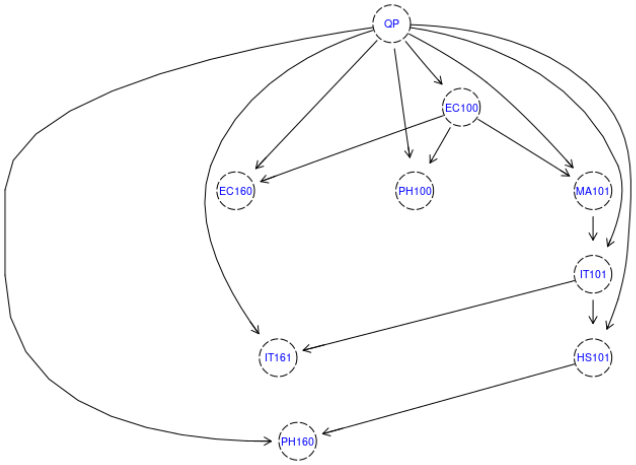


Fig. 23: Naive Bayes Classifier for dependent data

This network learns on the training dataset that we split earlier, and we test it on the test dataset. Repeating this 20 times, we get the accuracy as shown in Tab VIII.

Experiment no.	Accuracy
1	0.9710145
2	0.942029
3	0.9565217
4	0.942029
5	0.9275362
6	0.9565217
7	0.9571429
8	0.9565217
9	0.9857143
10	0.9130435
11	0.9428571
12	0.9714286
13	0.9275362
14	0.9565217
15	0.9142857
16	0.9710145
17	0.9
18	0.9130435
19	0.9565217
20	0.9710145

TABLE VIII: Accuracy on test dataset on Naive Bayes classifier considering independent data

From the above two tables, we can see that both the models have very good accuracy and are almost similar in results.

VI. Week 6 Lab Assignment 6

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: The Google Drive folder with the codes and datasets can found **here**.

A. PART A

Read through the reference carefully. Implement routines for learning the parameters of HMM given in section 7. In section 8, "A not-so-simple example", an interesting exercise is carried out. Perform a similar experiment on "War and Peace" by Leo Tolstoy.

From the reference "A Revealing Introduction to Hidden Markov Models" [16], 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 $\mathcal{O}$, 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 [16]. We initialized each element of π and A randomly to approximately 1/2. The initial values are:

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

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

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

After initial iteration,

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

After 100 iterations,

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

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

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

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

with final values of transpose of B in the Tab IX.

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 in Tab IX are the vowels and the second column are the consonants.

The Google Colab code and the dataset for War and Peace can be found **here**.

	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 and tossed 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 [19] 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[20]. 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 parameters θ.

We tried to implement EM for 10 bent coins to estimate their unknown bias on Google Colab, it's code can be found **here**. We took reference from Karl Rosaen's solution for 2 bent coins problem[21].

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 [19].

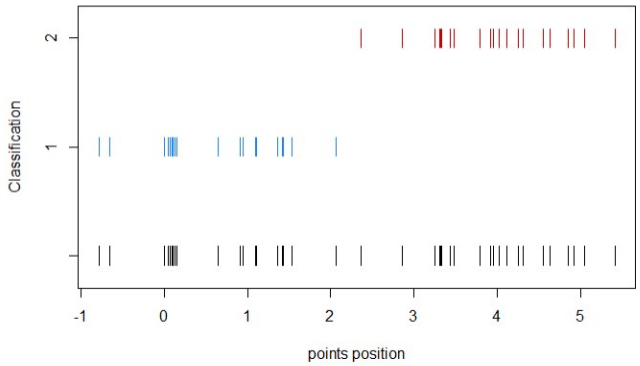


Fig. 24: EM clustering

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

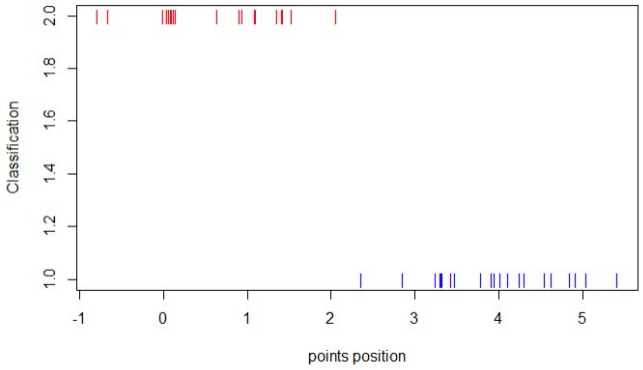


Fig. 25: k-means clustering

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.

The R-markdown and the high resolution images can be found **here**.

VII. Week 7 Lab Assignment 7

Learning Objective: 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

A. Part 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. 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.

We started with first importing the image of the 'cameraman', which is a standard image for benchmarking denoising algorithms, as it is very dynamic in the grayscale pixel range. [22]

This is a 512x512 grayscale image. We normalize the pixel values to be between 0 and 1, by dividing all values by 255, and then 'binarizing' it for the Markov Random Field by converting all the normalized pixel values below 0.5 to 0 and the rest to 1, as shown in Fig 26.

We, then introduce noise to this 'binarized' image. In order to test the capability, we add varying levels of noise, from 5% to 25% of the pixel values.

The varying levels of noises are shown in Fig 28.

Markov random fields sue a quadratic potential function to measure the energy potential of the image when changing a particular pixel, with respect to the neighbouring pixels.

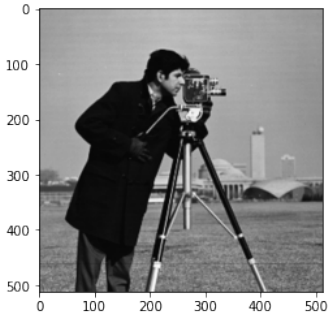
The quadratic potential function is given by:

$$E(u) = \sum_{n=1}^{N}(u_n - v_n)^2 + \lambda \sum_{n=1}^{N-1}(u_{n+1} - u_n)^2 \qquad (18)$$

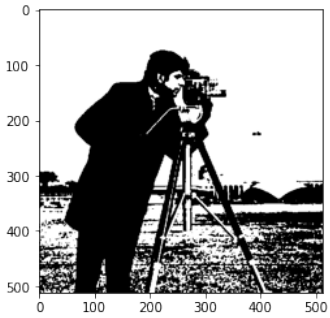
where v is the smooth ID signal, u is the IID and E is the energy function.

This is because for most cases, the values around a pixel are close to the pixel value. [23], [24]. We use the value of the constant lambdas as -100, while computing the quadratic potential function.

We ran the algorithm for 5*512*512 = 1310720 iterations.



(a) Original Cameraman Image



(b) 'Binarized' Cameraman Image

Fig. 26: Cameraman Images

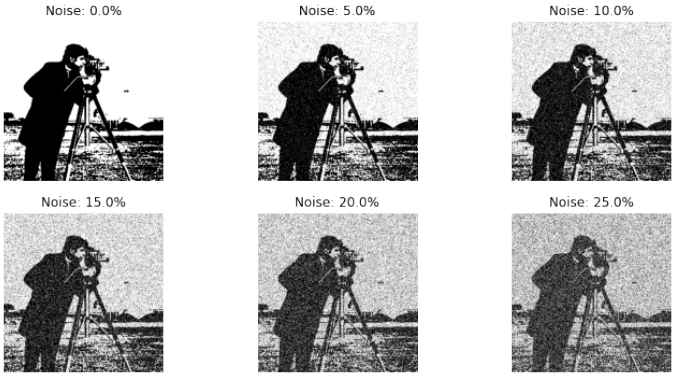


Fig. 27: Varying levels of noise added in the cameraman image

The max pixel denoised for all the noise levels are given in table **??**

The images after denoising with the MRF are: as shown in fig **??**.

% noise Level	% pixels denoised
5	4.70
10	6.99
15	9.30
20	11.67
25	14.02

TABLE X: Noise level and percent of pixels denoised

B. Part B

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.

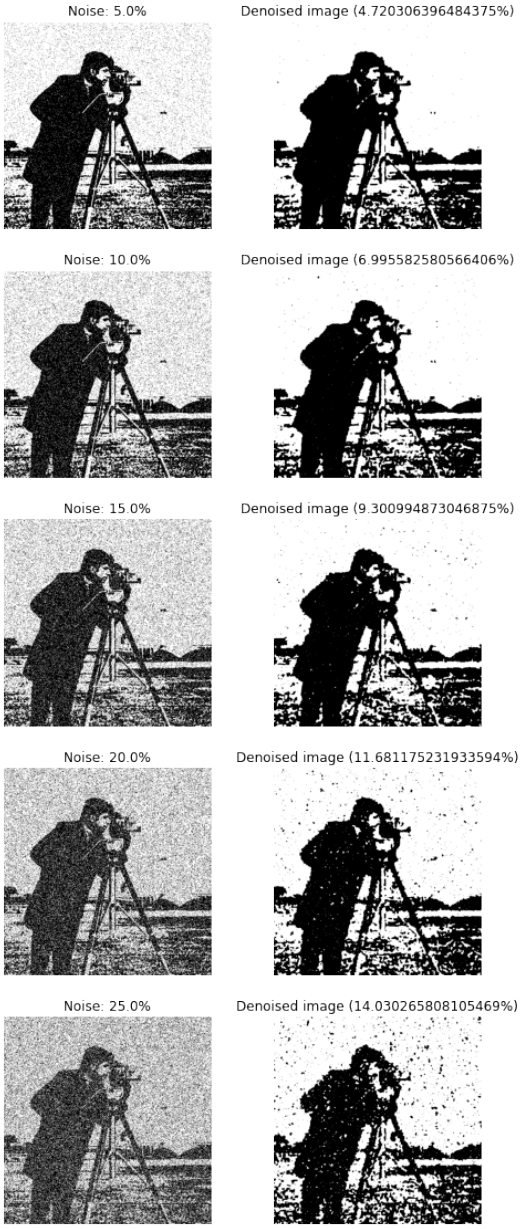


Fig. 28: Denoised images using MRF

For this task, we converted the hopfield.m MATLAB codes to Python so that we could do it in the same Notebook as the other parts.

The original images looked as shown in fig. 29.

The network was trained using Hebb's rule, as in the file and then they were noised by changing some random pixel values. At max, 15 pixel values were noised.

We can see that suprisingly, the network can correct upto max 8 errors. The results are show in fig 30.

C. Part C

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

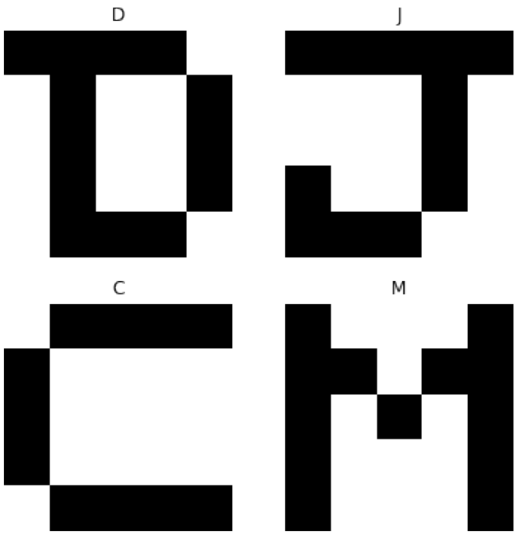


Fig. 29: The original letters for the hopfield networks

This is the usual famous NP-hard problem of Travelling Salesman, that is done using the a Hopfield Networks.

Since in a Hopfield Ntwrosk, each node is connected to each other node, we needed a total of 10x10 = 100 weights.

We first generate 10 cities randomly, as shown in fig 31.

And then let the hopfield network predict an optimal least path cost.

The path that we got was as shown in fig 32.

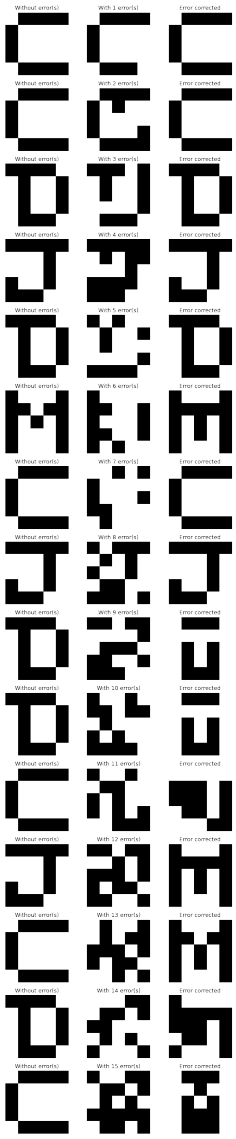


Fig. 30: The original, noised and corrected letters

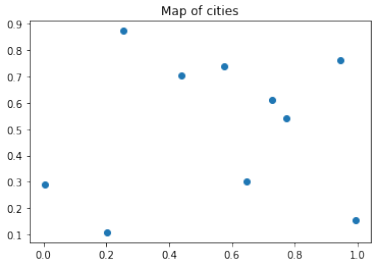


Fig. 31: Cities

VIII. Week 8 Lab Assignment 8

Learning Objective: Basics of data structure needed for state-space search tasks and use of random numbers required for MDP and RL.

Problem Statement: Read the reference on MENACE by Michie and check for its implementations. Pick the one that you like the most and go through the code carefully. Highlight the parts that you feel are crucial. If possible, try to code the MENACE in any programming language of your liking.

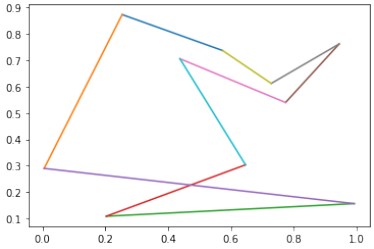


Fig. 32: Shortest Path

A. What is MENACE?

MENACE stands for Machine Educable Noughts and Crosses Engine [17]. It was originally described by Donald Michie, who used matchboxes to record each game he played against this algorithm. This provides an adequate conceptual basis for a trial-and-error learning device, provided that the total number of choice-points which can be encountered is small enough for them to be individually listed. Michie's aim was to prove that a computer could "learn" from failure and success to become good at a task.

B. Why Matchbox machine?

Matchboxes were used because each box contained an assortment of variously coloured beads. The different colours correspond to the different unoccupied squares to which moves could be made.

1 WHITE	2 LILAC	3 SILVER
8 BLACK	0 GOLD	4 GREEN
7 AMBER	6 RED	5 PINK

Fig. 33: The colour code used in the matchbox machine. The system of numbering the squares is that adopted for the subsequent computer simulation program

C. MENACE Strategies and how it learns.

A loss is punished by a removing the beads that were chosen from the boxes. This means that MENACE will be

less likely to pick the same colors again and has learned. A win is rewarded with three beads of the chosen color which is added to each box, reinforcing the MENACE to make the same move again. If a game is a draw, one bead is added to each box.

From the figure [34] , we can see the number of beads present inside a box on any given stage. Removing one bead from each box after losing means that later these moves are less likely to be picked and this helps MENACE learn more quickly, as the later moves are more likely to have led to the loss.

It is possible that after few games some boxes may end up empty. If one of these boxes is to be used, then MENACE resigns. When playing against skilled players, it is possible that the first move box runs out of beads. In this case, MENACE should be reset with more beads in the earlier boxes to give it more time to learn before it starts resigning.[18]

STAGE OF PLAY	NUMBER OF TIMES EACH COLOUR IS REPLICATED
1	4
3	3
5	2
7	1

Fig. 34: Variation of the number of colour-replicates of a move according to the stage of play.

D. Results

We tried to implement MENACE on Google Colab, it's code can be found **here**.

IX. Week 9 Lab Assignment 9

Learning Objective: Understanding Exploitation - Exploration in simple n-arm bandit reinforcement learning task, epsilon-greedy algorithm

A. Part A

Consider a binary bandit with two rewards 1-success, 0-failure. The bandit returns 1 or 0 for the action that you select, i.e. 1 or 2. The rewards are stochastic (but stationary). Use an epsilon-greedy algorithm discussed in class and decide upon the action to take for maximizing the expected reward. There are two binary bandits named binaryBanditA.m and binaryBanditB.m are waiting for you.

After using both the rewards(function) binaryBanditA and binaryBanditB here is what we observe :

Here we can observe that when probability of rewards were low (in the case of banditA) the expected rewards were initially

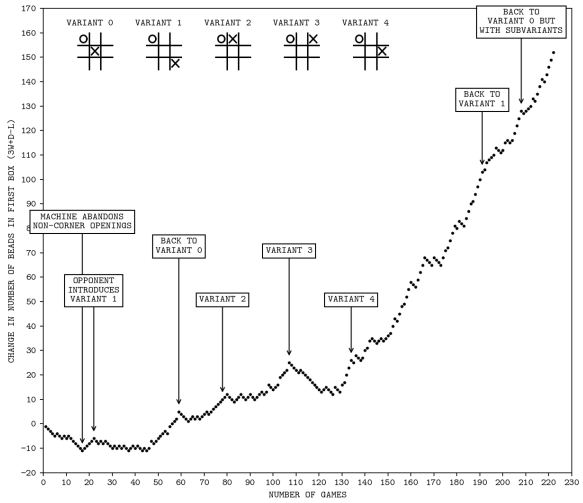


Fig. 35: The progress of MENACE'S maiden tournament against a human opponent. The line of dots drops one level for a defeat, rises one level for a draw and rises three levels for a victory.

0 as the n o. of trials increased we can see that expected rewards too become higher but this is not in the case of banditB as the probability of the reward were high(0.8 and 0.9) so the expected rewards for m the start are a close to 1 and the steps go by the expected rewards became a kind of constant because of the averaging factor

B. Part B

Develop a 10-armed bandit in which all ten mean-rewards start out equal and then take independent random walks (by adding a normally distributed increment with mean zero and standard deviation 0.01 to all mean-rewards on each time step).

This is the case of 10 arm-bandit where the mean-rewards are initially taken as a constant valued array initialised by 1. We performed exploration and exploitation based on the Epsilon Greedy Algorithm. A random number between 0 and 1 is generated and if it comes out to be greater than epsilon, we perform exploitation, which is based on the prior knowledge otherwise exploration. For every iteration an array of ten values is generated such that they are normally distributed with mean value zero and standard deviation of 0.01. This array is added to the mean array and this updated array is used every time. For every action a reward is given from this updated mean-rewards array. The rewards given in this case are non-stationary.

Here we observe that as initially all the rewards were equal spo that we get a constant reward of say like 1 but as the steps increases , it affects the rewarding policy of every action and

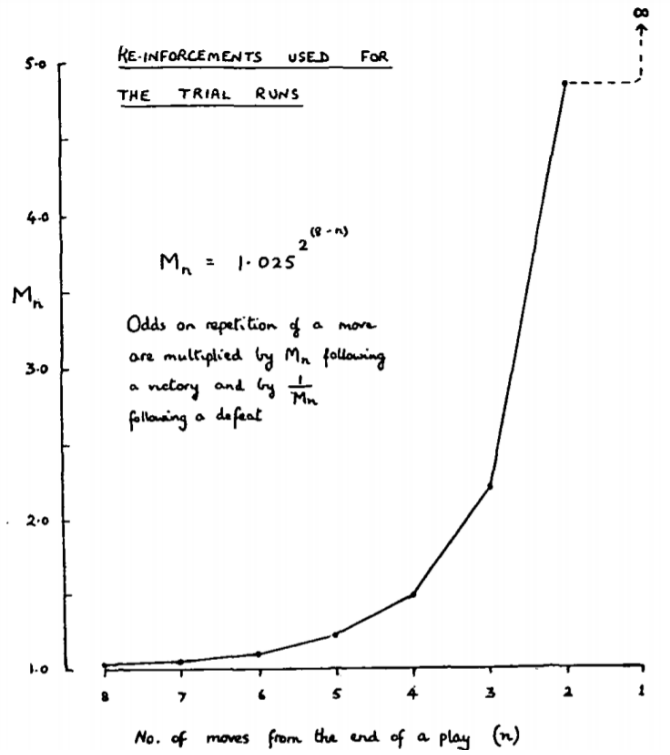


Fig. 36: The multipliers used for reinforcement in the trial runs.

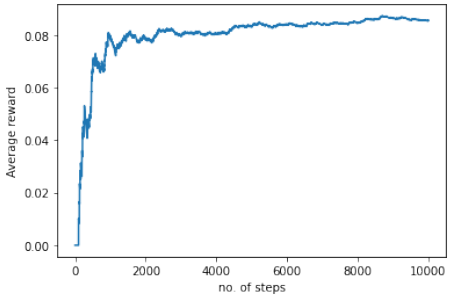


Fig. 37: Expected rewards we get after using banditA

thus we see that the expected rewards also started increasing at a very high non zero rate (Fig.31)

C. Part D

The 10-armed bandit that you developed (banditnonstat) is difficult to crack with standard epsilon-greedy algorithm since the rewards are non-stationary. We did discuss about how to track non-stationary rewards in class. Write modified epsilon-greedy agent and show whether it is able to latch onto correct actions or not.

This is same as problem 2 except the calculation of action rewards. In this case, instead of using averaging method to update estimation of action reward, we are assigning more weights to the current reward earned by using alpha parameter having value 0.7

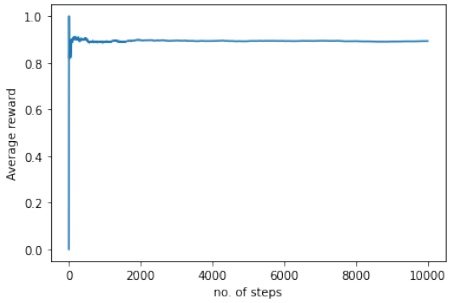


Fig. 38: Expected rewards we get after using banditB

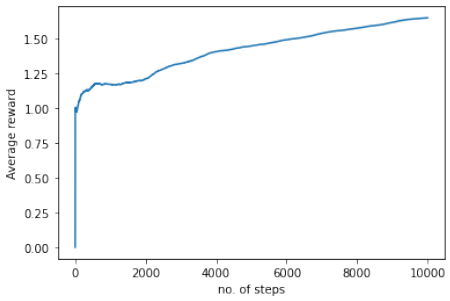


Fig. 39: Expected rewards we get when rewards are non stationary (averaging method)

Here we can see that as compared to problem 2, in which the expected rewards were low , we saw that when instead of averaging the profit percentage of state when we gave higher weightage to the to the current value at every step (a normally distributed array is added to the rewards array), the expected rewards we get were much higher as in the case 2 and the slope of the graph was too steeper when compared with case 2.

The Google Colab Notebook code for this assignment can be found **here**.

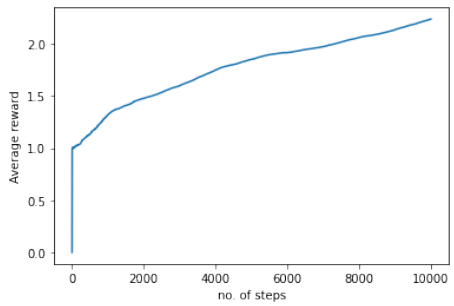


Fig. 40: Expected rewards we get when rewards are non stationary (non averaging method)

X. Conclusion

As demonstrated above we have solved 8 puzzle problem, Travelling Salesman Problem, Noughts and Crosses game, Nim Game and understood the Bayesian graphical models for to build network graphs.

In 8-puzzle problem we were able observe performance of different heuristic functions, where heuristic with Manhattan distance gave best results whereas when solved without heuristic performance was the worst.

For the Travelling Salesman Problem, we observed that with simulated annealing it provides an optimal or sub optimal solution which reduces the cost for given set of points/coordinates. We also observed that how the decrease in temperature and number of iterations affects the solution.

From our observation for mini-max and alpha-beta pruning less nodes were evaluated in alpha-beta pruning than in mini-max algorithm (See table V), which was the essence of the assignment. We can clearly say that alpha-beta pruning is not a different algorithm than mini-max it is just a speed up process which does not evaluate approximate values instead evaluates to perfectly same values.

We explored the Bayesian network and were able to model it using grades data-set, we then calculated Conditional Probability Tables (CPTs) for them and saw how they were dependent (See Fig. 20). Naive Bayes Classifier gave very accurate results on the test data-set which can be observed here at Fig 23 and Fig 22.

From the Markov Random Field and Hopfield networks, we learned how to construct learning machines using Markov Decision Processes. Hopfield Network showed how we can use a simple collection of neurons to construct a very robust network, that can resist ignore large noises and still predict a correct output and lastly, we saw how they can be used in Travelling Salesman Problem to get an optimal path.

We learnt about MENACE [17] and how it learns through reinforcing its decision. The Aim was to prove that machines can learn from failure and make better decisions after trial and error. From the fig. 35, we can see how it performs against humans as more and more games are played. Initially, it struggled a lot and lost but after sometime there was consecutive draws but it faltered again and lost few matches and finally it kept wining.

From the HMM and EM algorithm problem, we applied a HMM model to the 50,000 letters observations including space and all lowercase characters. After about 100 iterations we observed that the B matrix (see Table IX) tells us about two distinct categories of characters. Upon closely observing, we find that one set contains consonants while the other contains vowels. For the second part we apply EM algorithm to the 10 bent coin problem (similar to well known 2 bent coin example) to find the latent parameters and estimate the hidden biases for the coins. In the third part we learnt about soft and hard clustering and EM algorithm clustering for given dataset, we also learnt about the k-means and how k-means compares to EM algorithm.

For the n arm bandit we learned how the problem works and the epsillon greedy algorithm ,we saw that how that algorithm can be modelled for both the stationary environment case and the non stationary case just be simply giving more weightage to the current step.

Acknowledgment

We would like to thank Prof. Pratik Shah for guiding us through this course and the contents, both during the lecture and laboratory hours. The lectures have helped us immensely in understanding the contents of this course and to apply this knowledge in performing these experiments.
We'd also like to thank our fellow colleagues, that help in healthy discussions of the course content.
Lastly, we would like to thank any source that we might have referred to, for performing these experiments but may have missed to give credits to, in the reference section.

References

[1] Josh Richard, *An Application Using Artificial Intelligence*, January 2016
[2] dpthegrey, *8 puzzle problem* June 2020.
[3] Ajinkya Sonawane, *Solving 8-Puzzle using A* Algorithm* September 2018
[4] javatpoint *Uninformed Search Algorithms*
[5] Zhou, Ai-Hua, *Traveling-salesman-problem algorithm based on simulated annealing and gene-expression programming*. Information 10.1 2019.
[6] Frank Liang, *Optimization Techniques for Simulated Annealing*. https://towardsdatascience.com/optimization-techniques-simulated-annealing-d6a4785a1de7
[7] Traveling Salesperson Problem. https://www.youtube.com/watch?v=35fzyblVdmA&t=656s.
[8] Marco Scutar, *Learning Bayesian Networks with the bnlearn R Package*, Issue 3. Journal of Statistical Software, July 2010.
[9] Bojan Mihaljević, *Bayesian networks with R* November 2018.
[10] Alexandra M. Carvalho, *Scoring functions for learning Bayesian networks*
[11] Shah Pratik, *An Elementary Introduction to Graphical Models* February 2021.
[12] The R Foundation, *R: Documentation* February 2021.
[13] Marianne Freiberger, *Play to win with Nim*.July 21, 2014. https://plus.maths.org/content/play-win-nim#:~:text=The%20strategy%20is%20to%20always,B%20has%20a%20winning%20strategy.

[14] Best-Case Analysis of Alpha-Beta Pruning. http://www.cs.utsa.edu/~bylander/cs5233/a-b-analysis.pdf
[15] 8 puzzle problem. https://medium.com/@dpthegrey/8-puzzle-problem-2ec7d832b6db
[16] Stamp, Mark, *A revealing introduction to hidden Markov models*, 2004.
[17] MENACE: Machine Educable Noughts And Crosses Engine https://people.csail.mit.edu/brooks/idocs/matchbox.pdf
[18] MENACE: Machine Educable Noughts And Crosses Engine Blog https://www.mscroggs.co.uk/blog/19
[19] What is the expectation maximization algorithm? Chuong B Do and Serafim Batzoglou, Nature Biotechnology, Vol 26, Num 8, August 2008 https://datajobs.com/data-science-repo/Expectation-Maximization-Primer-[Do-and-Batzoglou].pdf
[20] Expectation–maximization algorithm https://en.wikipedia.org/wiki/Expectation%E2%80%93maximization_algorithm
[21] Expectation Maximization with Coin Flips http://karlrosaen.com/ml/notebooks/em-coin-flips/
[22] What is the reason the test image "Cameraman" is used widely to test algorithms in image processing and image encryption? https://www.researchgate.net/post/What_is_the_reason_the_test_image_Cameraman_is_used_widely_to_test_algorithms_in_image_processing_and_image_encryption
[23] Image Denoising Benchmark https://www.cs.utoronto.ca/~strider/Denoise/Benchmark/
[24] Markov Random Fields https://web.cs.hacettepe.edu.tr/~erkut/bil717.s12/w11a-mrf.pdf