First Lab Report CS 362 (Group 5 Naagraaj)

February 26, 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 4 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

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

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.

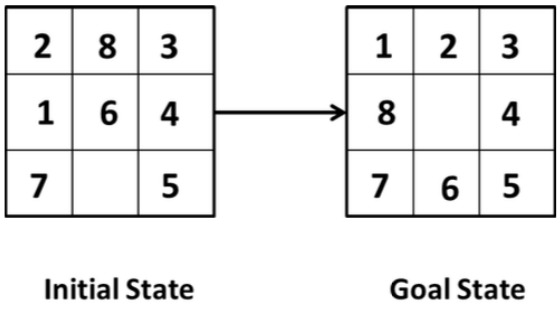

Fig. 1: Initial and final state of 8 puzzle[15]

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.

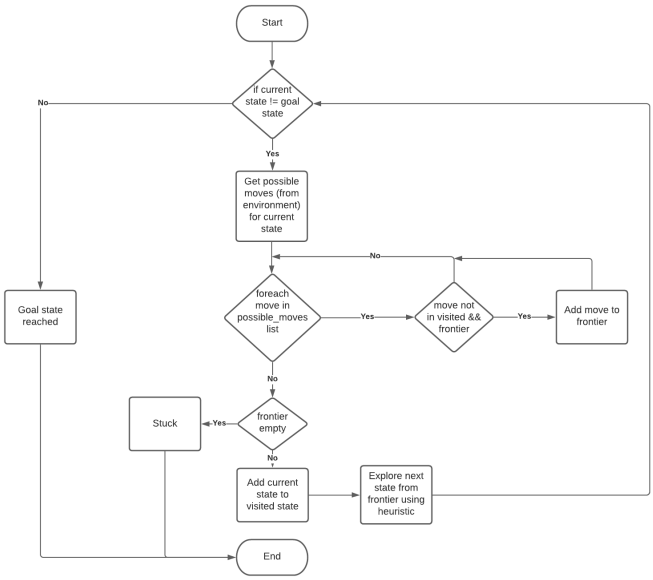

Fig. 2: Flowchart for Agent

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.

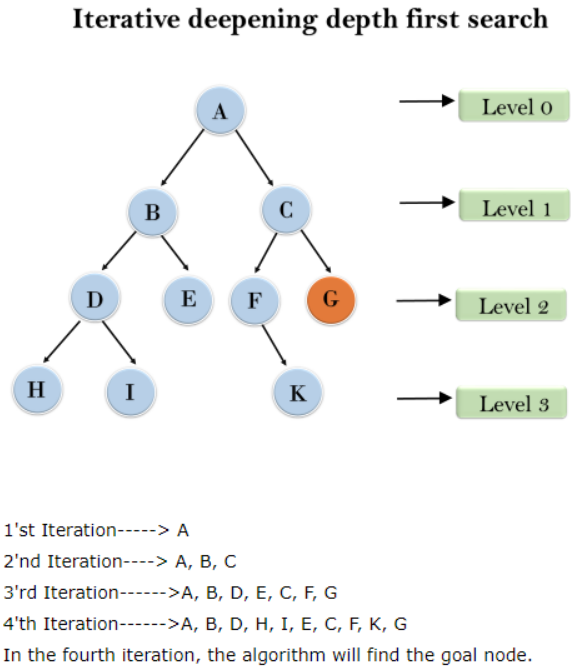

Fig. 3: Iterative Deepening Search

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

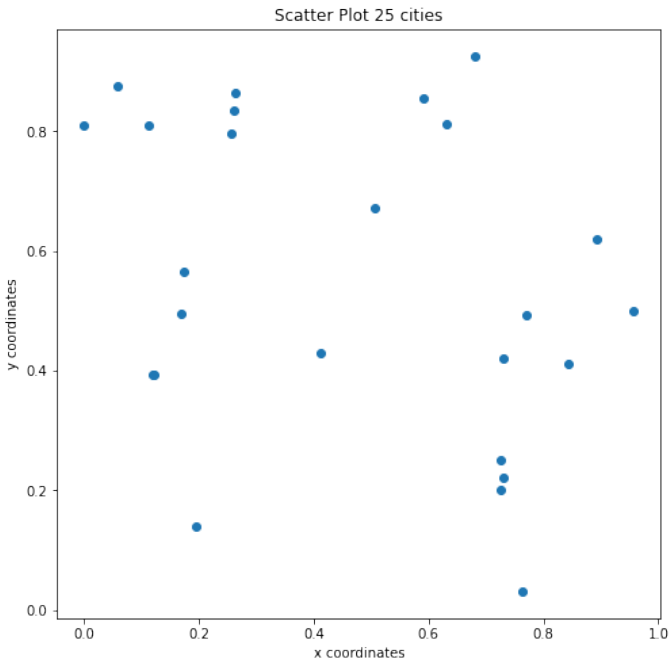

Fig. 4: Scatter Plot for 25 nodes

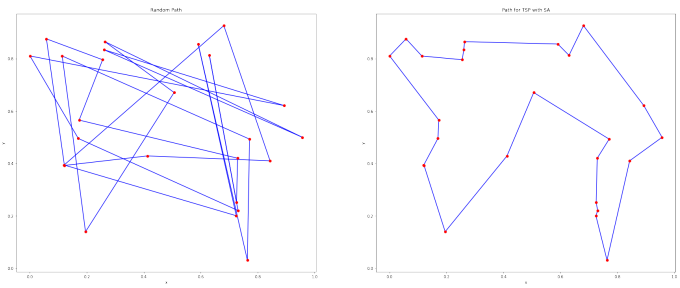

Fig. 5: Comparison between the routes for 25 nodes

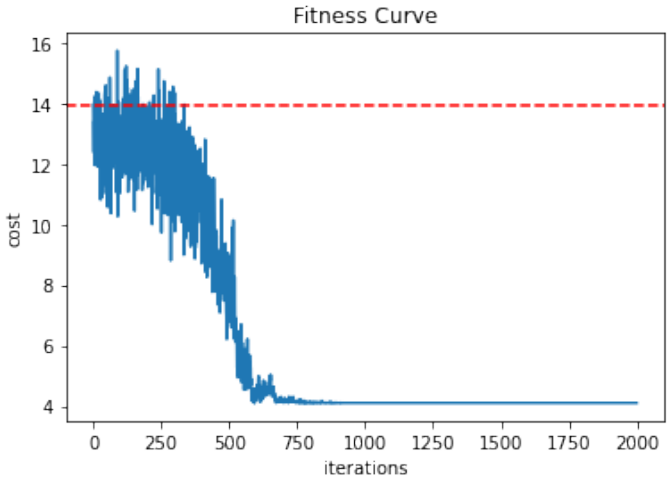

Fig. 6: Fitness curve for 25 nodes

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.

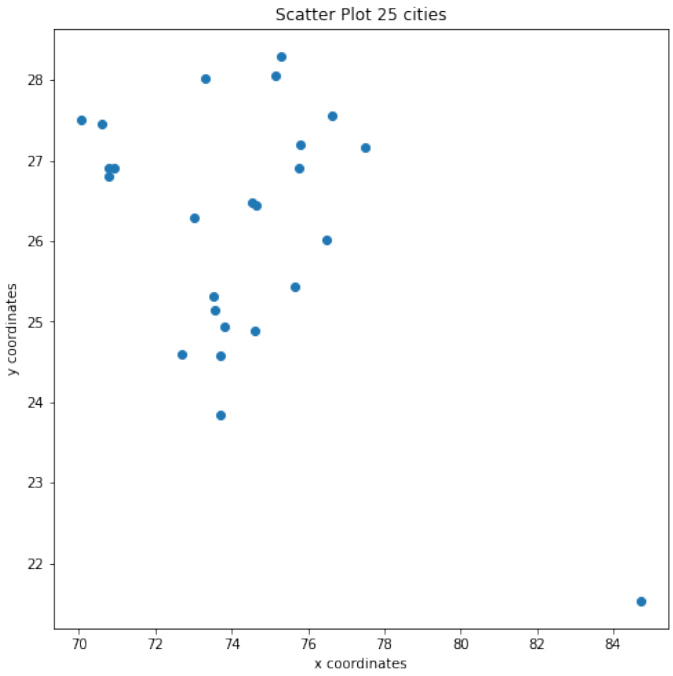

Fig. 7: Scatter Plot for 25 locations in Rajasthan

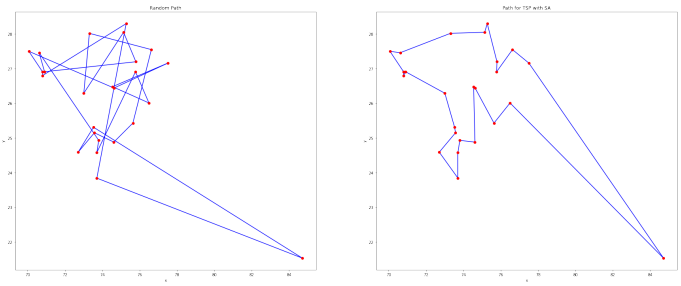

Fig. 8: Optimal path for 25 locations in Rajasthan

1) XQF131 - 131 points

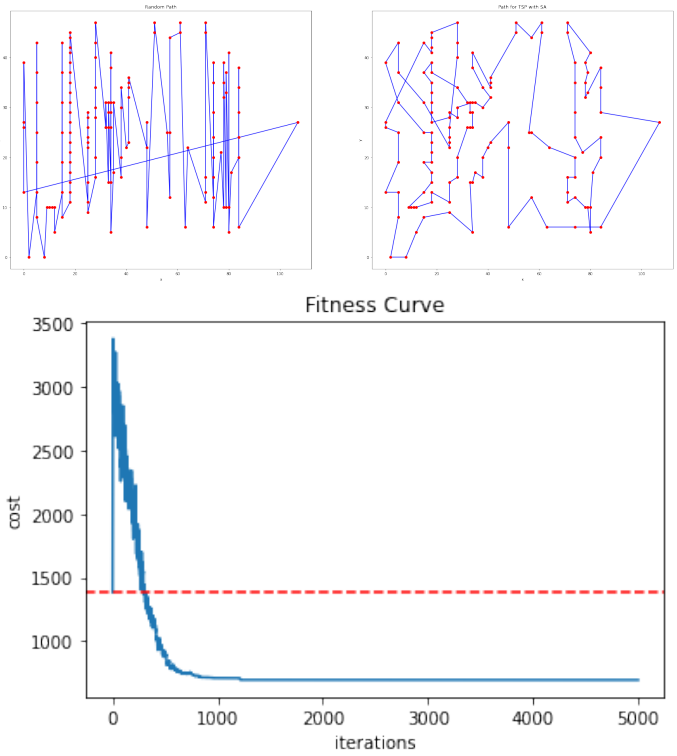

Fig. 9: Plots for 131 points

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

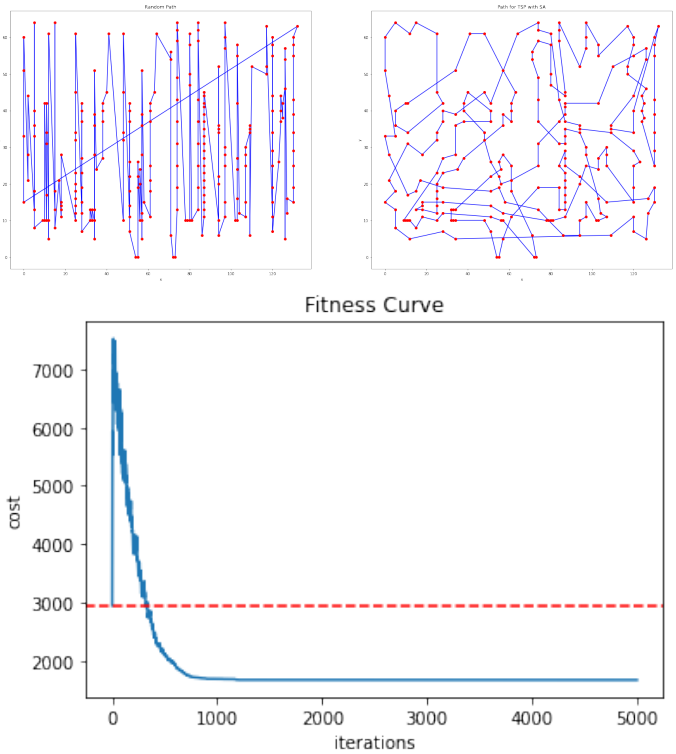

Fig. 10: Plots for 237 points

3) PMA343 - 343 points

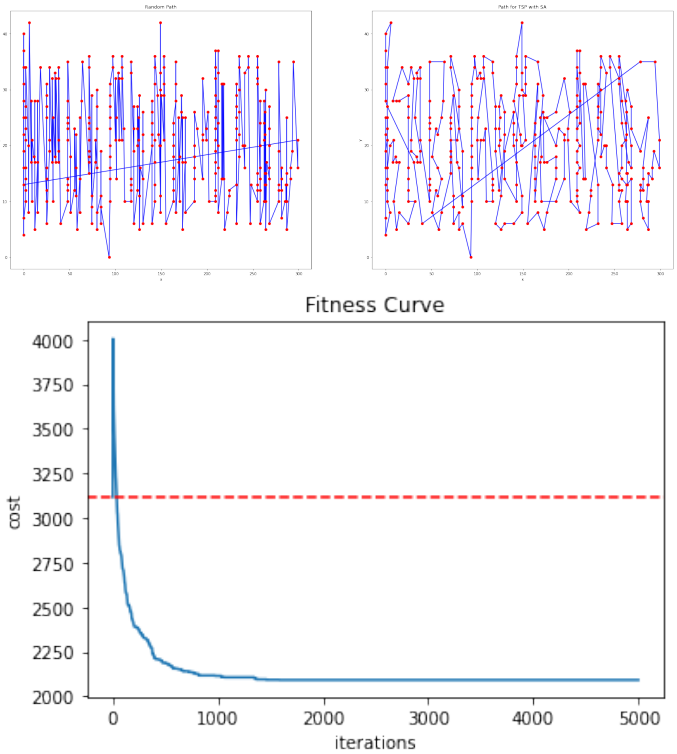

Fig. 11: Plots for 343 points

4) PKA379 - 379 points

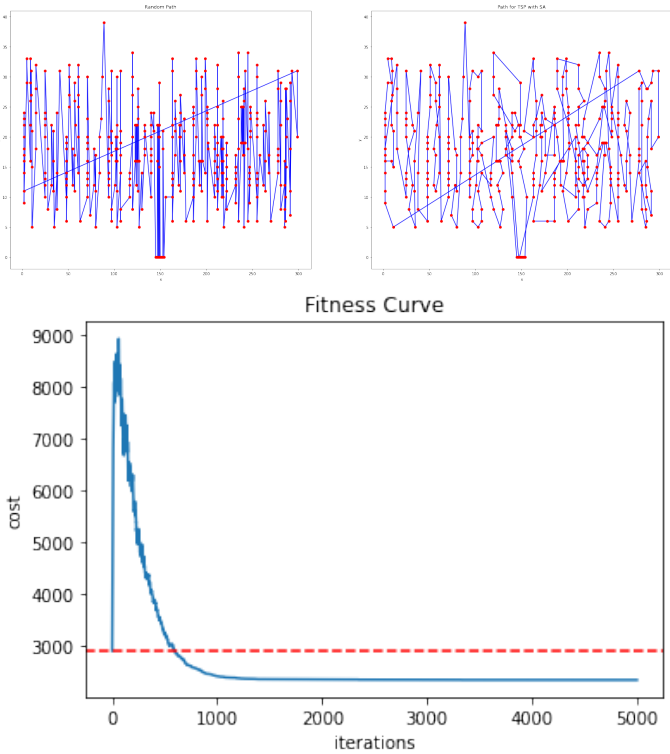

Fig. 12: Plots for 379 points

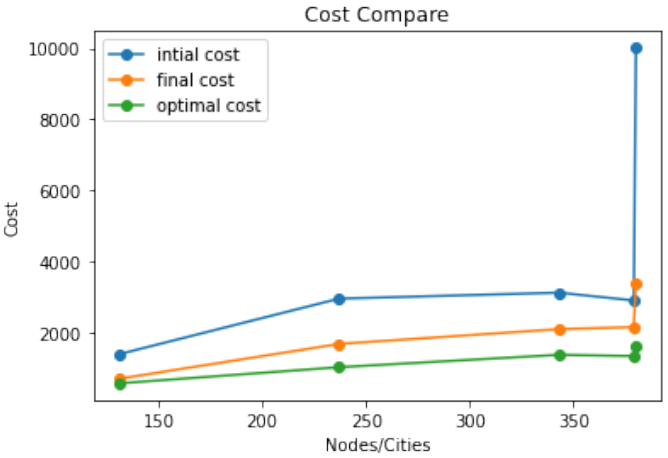

Fig. 13: Comparing costs

5) BLC380 - 380 points

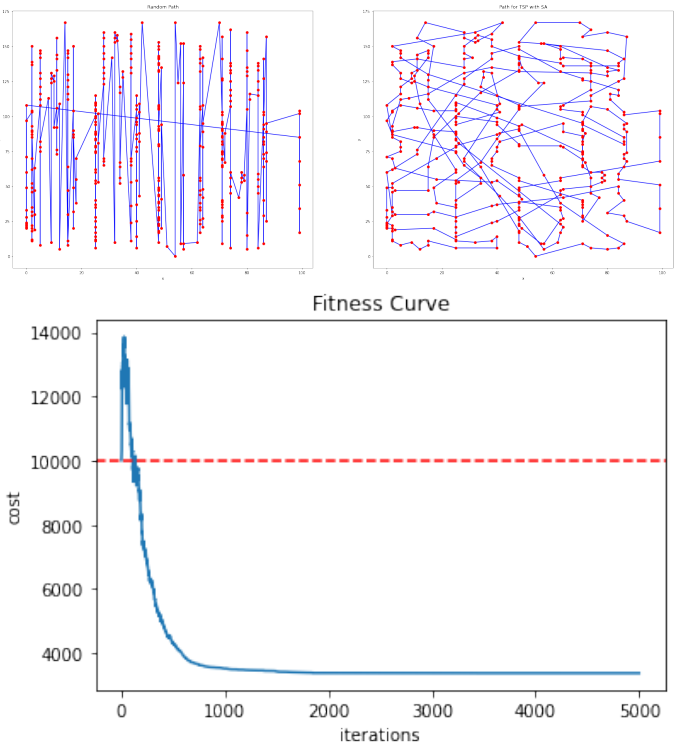

Fig. 14: Plots for 380 points

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

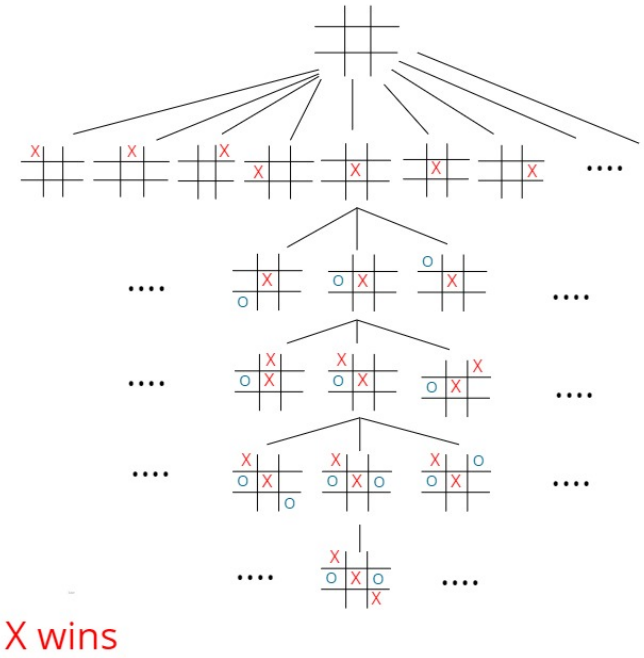

Fig. 15: Game tree for one state

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

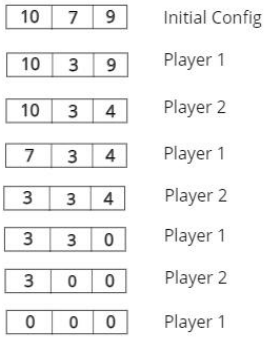

Fig. 17: Right section represents move played by player 1 or Player 2

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

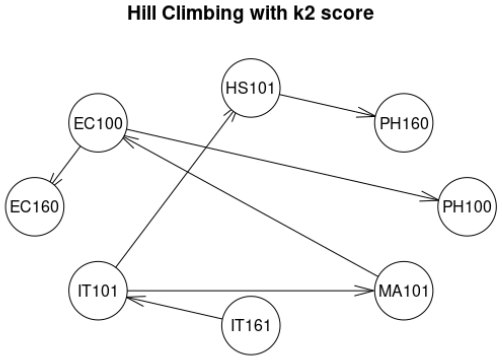

Fig. 18: Hill climbing Bayesian network using the k2 score

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]'

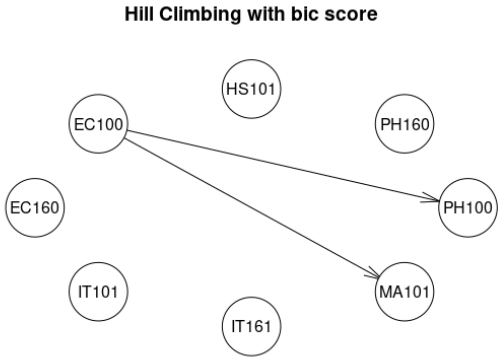

Fig. 19: Hill climbing Bayesian network using the bic score

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.

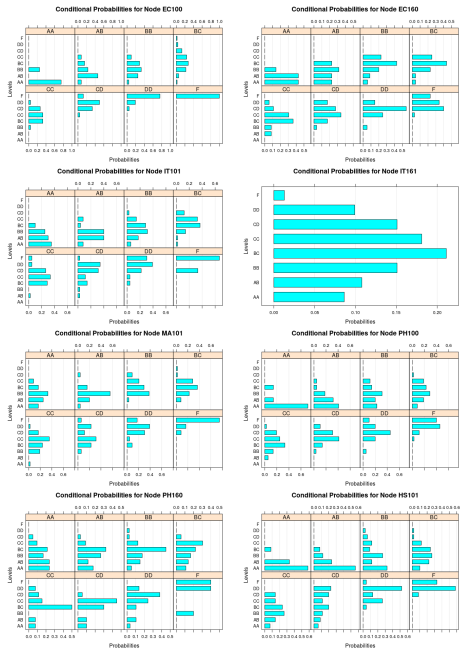

Fig. 20: Conditional probability distributions made form the CPTs of the learned data

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.

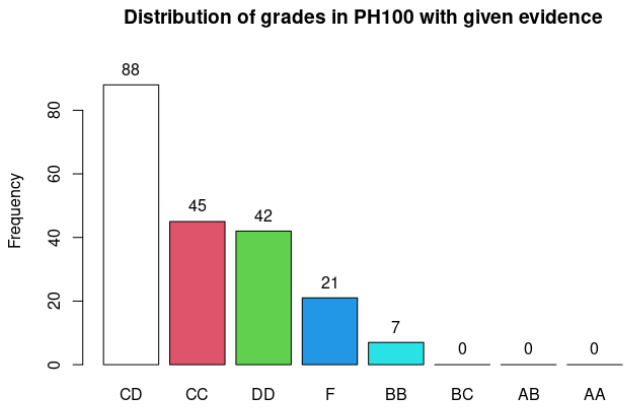

Fig. 21: Probability distribution of getting a grade in PH100 as per the given evidence

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.

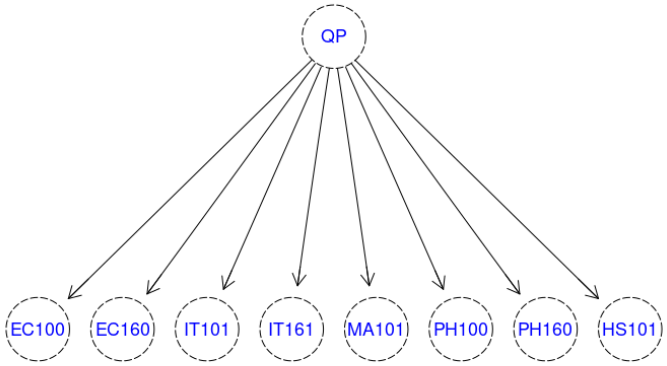

Fig. 22: Naive Bayes Classifier for independent data

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.

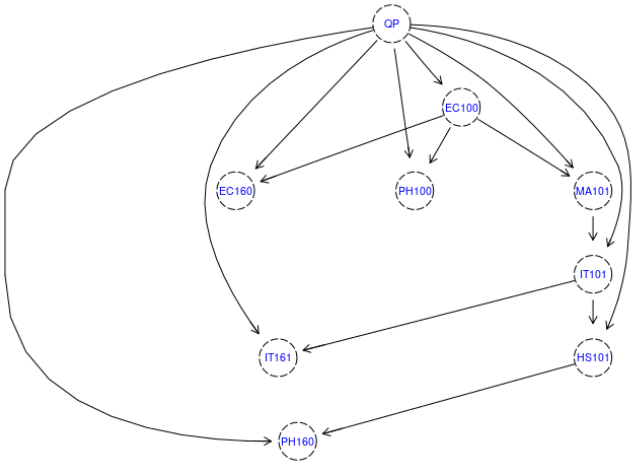

Fig. 23: Naive Bayes Classifier for dependent data

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. 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.

Lastly 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.

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