Exp no 1: Implementation of toy problems (Tic Tac Toe)

AIM:

To implement the two player Tic-Tac-Toe game in python

ALGORITHM:

1. We will make the board using dictionary in which keys will be the location (i.e: top-left, mid-right, etc.) and initially it's values will be empty space and then after every move we will change the value according to player's choice of move.
2. We will have to print the updated board after every move in the game and thus we will make a function in which we'll define the printBoard function so that we can easily print the board every time by calling this function.
3. Now we'll write the main function which has all the gameplay functionality.
4. Now we will check if player X or O has won, for every move after 5 moves.
5. If neither X nor O wins and the board is full, we'll declare the result as 'tie'.
6. Now we have to change the player after every move.
7. Now we will ask if player wants to restart the game or not.

Exp no 2: Developing agent programs for real world problems (8-puzzle)

AIM:

To implement the python code to display the way from the root node to the final destination node for N*N-1 puzzle algorithm by the help of Branch and Bound technique.

ALGORITHM:

1. Helper function that returns -1 for non-found index value of a seq
2. Returns list of tuples with which the free space may be swapped.

3. Get row and column of the empty piece and find which pieces can move there.
4. Performs A* search for goal state.
5. If finished state not found, return failure.
6. Heuristic template that provides the current and target position for each number and the total function.
7. Some heuristic functions, the best being the standard manhattan distance in this case, as it comes closest to maximizing the estimated distance while still being admissible.

Exp no 3: Implementation of constraint satisfaction problems (Cryptarithmetic Problem)

AIM:

To implement the CPP program for solving cryptographic puzzles

ALGORITHM:

1. Vector stores 1 corresponding to index number which is already assigned to any char, otherwise stores 0.
2. Structure to store char and its corresponding integer.
3. Function check for correct solution
4. Recursive function to check solution for all permutations.
5. Call recursive function.
6. Backtrack for all other possible solutions.

Exp no 4: Implementation and Analysis of DFS and BFS for an application

AIM:

To implement the python program for analysis of DFS and BFS for an application

ALGORITHM:

DFS:

1. Find DFS traversal from starting vertex.
2. Create memo once in top-level call.
3. Generate adjacency list for undirected graph.

BFS:

1. BFS algorithm in python
2. Dequeue a vertex from queue
3. If not visited, mark it as visited and enqueue it.

Exp no 5: Developing Best first search and A* Algorithm for real world problem

AIM:

To develop the best first search and A* algorithm for real world problem in python

ALGORITHM:

1. A node class for A* Pathfinding
2. Returns a list of tuples as a path from the given start to the given end in the given maze.
3. Create start and end node.
4. Initialize both open and closed list.
5. Add the start node.
6. Loop until you find the end.
7. Get the current node.
8. Pop current off open list, add to closed list.
9. Generate children.
10. Get node position.
11. Add the child to the open list.

Exp no 6: Implementation of minimax algorithm for an application

AIM:

To implement the minimax algorithm for an application in python

ALGORITHM:

1. Import libraries.
2. Create a new game and game state.
3. Set the number of noughts and crosses in a row that is needed to win the game.
4. Create vectors and Set lengths.
5. Set the starting player at random and get winning positions.
6. Loop the board and vectors.
7. Get the start position and vector deltas.
8. Create a counter and Loop until we are outside the board.
9. Add winning positions and Break out from the loop
10. Update the position.
11. Check if the loop should terminate.
12. Get a heuristic move at cut off.
13. Check if the game has ended, break out from the loop in that case.
14. Get a recommended move.
15. Get a heuristic move at cut off.
16. Check if the move is legal.
17. Create a heuristic dictionary.
18. Check if number is in a winning position and calculate the number of X: s and O: s
19. Get the best move from the heuristic dictionary and return the best move.
20. Check if the game has ended and return a tie.
21. Check if a player has won.
22. Loop until we are outside the board or have moved the number of steps in the goal.
23. Check if a player has a piece in the tile.

24. Check if the loop should terminate.
25. Return None if no winner is found.
26. Get a min value (O)
27. Set min value to max value if it is lower than current min value.
28. Do an alpha test and beta test.
29. Get max value (X)
30. Check if the game has ended.
31. Add a piece to the board.
32. Set max value to min value if min value is greater than current max value.
33. Adjust the max value.
34. Print the current game state.
35. Tell python to run main method.

Exp no 7: Implementation of Unification and Resolution

AIM:

To implement the unification and resolution in python

ALGORITHM:

```
Step 1: If Ψ1 or Ψ2 is a variable or constant, then:
    a. If Ψ1 or Ψ2 are identical, then return NULL.
    b. Else if Ψ1 is a variable:
            -then if Ψ1 occurs in Ψ2, then return False
            -Else return (Ψ2 / Ψ1)
    c. Else if Ψ2 is a variable:
            -then if Ψ2 occurs in Ψ1, then return False
            -Else return (Ψ1 / Ψ2)
    d. Else return False.
Step 2: If the initial Predicate symbol in Ψ1 and Ψ2 are not same, then return False.
Step 3: IF Ψ1 and Ψ2 have a different number of arguments, then return False.
```

Step 4: Create Substitution list.

Step 5: For i=1 to the number of elements in Ψ1.

a. Call Unify function with the i^{th} element of Ψ1 and i^{th} element of Ψ2, and put the result into S.
b. If S = False then returns False
c. If S ≠ Null then append to Substitution list

Step 6: Return Substitution list.

Exp no 8: Implementation of Knowledge Representation schemes

AIM:

To implement knowledge representation schemes in c

ALGORITHM:

1. Function to check if all cells are assigned or not.
2. If there is any unassigned cell, then that function will change the values of row and col accordingly.
3. Function to check if we can put a value in a particular cell or not.
4. Checking sub matrix
5. Function to solve sudoku using backtracking.
6. If all cells are assigned, then the sudoku is already solved.
7. If we can’t proceed with this solution, reassign the cell.

Exp no 9: Implementation of Uncertain Methods for an application

AIM:

To implement the uncertain methods for an application in jupyter

ALGORITHM:

# prediction

y_pred_without_dropout = model_without_dropout.predict(x_test)

```
y_pred_with_dropout = model_with_dropout.predict(x_test)

# plotting
fig, ax = plt.subplots(1,1,figsize=(10,5))
ax.scatter(x_train, y_train, s=10, label='train data')
ax.plot(x_test, x_test, ls='--', label='test data', color='green')
ax.plot(x_test, y_pred_without_dropout, label='predicted ANN - R2 {:.2f}'.format(r2_score(x_test, y_pred_without_dropout)), color='red')
ax.plot(x_test, y_pred_with_dropout, label='predicted ANN Dropout - R2 {:.2f}'.format(r2_score(x_test, y_pred_with_dropout)), color='black')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.legend()
ax.set_title('test data');
```

Exp no 10: Implementation of block world problem

AIM:

To implement block world problem in python

ALGORITHM:

1. If Block is on another block, unstack.
2. If block is on table, pick up.
3. Initially push the goal_state as compound goal onto the stack
4. Repeat until the stack is empty.
5. Get the top of the stack.
6. If Stack Top is Compound Goal, push its unsatisfied goals onto stack.
7. If Stack Top is an action and peek the operation
8. Check if any precondition is unsatisfied and push it onto program stack.

9. If all preconditions are satisfied, pop operation from stack and execute it.
10. If Stack Top is a single unsatisfied goal, Replace Unsatisfied Goal with an action that can complete it.
11. Push Precondition on the stack.

Exp no 11: Implementation of Learning algorithms

AIM:

To implement the learning algorithms in jupyter

ALGORITHM:

Import Matplotlib:

a. Use the "import" statement to import the Matplotlib library.

Prepare the data:

a. Create a dataset that you want to visualize using Matplotlib.

b. Ensure that the data is in a format that Matplotlib can use, such as a NumPy array or a Pandas DataFrame.

Create a figure:

a. Use the "plt.figure()" function to create a new figure object.

b. Specify the size and other properties of the figure if necessary.

Create a plot:

a. Use one of the many plot functions available in Matplotlib, such as "plt.plot()" or "plt.scatter()", to create a plot of the data.

b. Specify the data to be plotted, the color, the marker style, and other properties of the plot.

Add titles and labels:

a. Use the "plt.title()" function to add a title to the plot.

b. Use the "plt.xlabel()" and "plt.ylabel()" functions to add labels to the x and y axes of the plot.

Customize the plot:

a. Use various Matplotlib functions to customize the appearance of the plot, such as changing the font size or style, adjusting the axis limits, or adding a legend.

Save or show the plot:

a. Use the "plt.savefig()" function to save the plot to a file in a specified format, such as PNG, PDF, or SVG.

b. Use the "plt.show()" function to display the plot on the screen.

Exp no 12: Development of ensemble model

AIM:

To implement the ensemble model in jupyter

ALGORITHM:

1. Load libraries
2. Import train_test_split function.
3. Load data
4. Split dataset into training set and test set
5. Import Support Vector Classifier
6. Import scikit-learn metrics module for accuracy calculation.
7. Create adaboost classifer object.
8. Train Adaboost Classifer
9. Predict the response for test dataset.

Exp no 13: Natural language processing-Levels of NLP -Text Pre processing

AIM:

To implement the text pre-processing of Natural Language Processing in python

ALGORITHM:

Import the necessary libraries.

Tokenization:

a. Split the text into individual words or tokens.

b. Remove any punctuation or special characters that are not relevant to the analysis.

c. Convert all the words to lowercase for consistency.

Stop word removal:

a. Remove common words that are not relevant to the analysis, such as "the", “and” ,"a", etc.

b. Use a predefined list of stop words or create your own list based on the specific problem.

Stemming or Lemmatization:

a. Reduce each word to its root or base form to reduce the number of unique words in the text.

b. Use a stemmer or lemmatizer to perform this task.

c. Stemming is a more aggressive method that can produce non-words, while lemmatization preserves the meaning of the words.

Part-of-speech tagging:

a. Identify the part of speech (noun, verb, adjective, etc.) of each word in the text.

b. Use a part-of-speech tagger to perform this task.

Named entity recognition:

a. Identify named entities such as people, places, and organizations in the text.

b. Use a named entity recognizer to perform this task.

Parsing:

a. Analyze the grammatical structure of the text to understand the relationship between words.

b. Use a parser to perform this task.

Feature extraction:

a. Identify relevant features in the text, such as keywords or phrases, that can be used for further analysis.

b. Use techniques such as term frequency-inverse document frequency (TF-IDF) or bag-of-words to extract these features.

Exp no 14: Discriminating between ham/spam messages automatically using UCI datasets

AIM:

To discriminate between ham/spam messages automatically using UCI datasets in google colab

ALGORITHM:

1. Import libraries.
2. Load data
3. Rename names columns.
4. Join words again to form the string.
5. Remove any stopwords for message_not_punc, but first we should transform this into the list.
6. Classification Model
7. Test model

Exp no 15: Applying deep learning method for Automatic Handwriting recognition

AIM:

To apply deep learning method for automatic handwriting recognition by using google colab

ALGORITHM:

1. Import libraries.
2. Number of training epochs since start
3. Best validation character error rate
4. Number of epochs no improvement of character error rate occurred.
5. Stop training after this number of epochs without improvement.
6. Train, validate and write summary.
7. If best validation accuracy so far, save model parameters.
8. Stop training if no more improvement in the last x epochs.
9. Print validation result.
10. Set chosen CTC decoder.
11. Train or validate on IAM dataset.
12. Load training data, create TF model.
13. Save characters of model for inference mode
14. Save words contained in dataset into file.
15. Execute training or validation.
16. Infer text on test image.