Sinhgad Technical Education Society's

RMD SINHGAD SCHOOL OF ENGINEERING, PUNE

Department of Information Technology

Academic Year: 2023-24

Third Year (Semester: VI)

Laboratory Practice-II: 314458 (Artificial Intelligence)

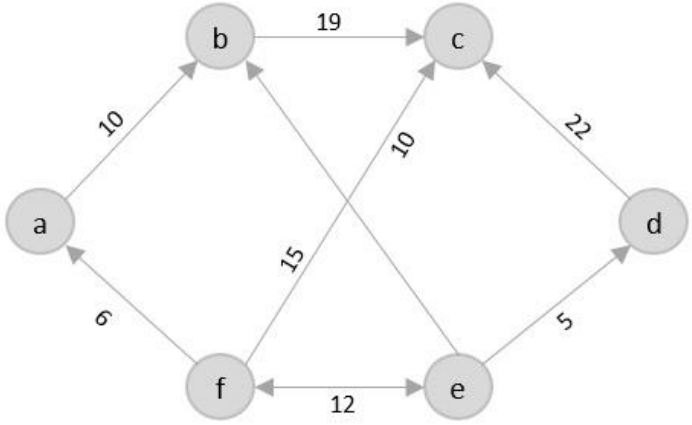
Identify and Implement heuristic and search strategy for Travelling Salesperson Problem

The traveling salesman problems abide by a salesman and a set of cities. The salesman has to visit every one of the cities starting from a certain one (e.g., the hometown) and to return to the same city. The challenge of the problem is that the traveling salesman needs to minimize the total length of the trip.

Suppose the cities are x_1 x_2..... x_n where cost c_{ij} denotes the cost of travelling from city x_i to x_j. The travelling salesperson problem is to find a route starting and ending at x_1 that will take in all cities with the minimum cost.

Example: A newspaper agent daily drops the newspaper to the area assigned in such a manner that he has to cover all the houses in the respective area with minimum travel cost. Compute the minimum travel cost.

If you look at the graph below, considering that the salesman starts from the vertex 'a', they need to travel through all the remaining vertices b, c, d, e, f and get back to 'a' while making sure that the cost taken is minimum.



Laboratory Practice-II: 314458 (Artificial Intelligence)

There are various approaches to find the solution to the travelling salesman problem: naive approach, greedy approach, dynamic programming approach, etc. In this tutorial we will be learning about solving travelling salesman problem using greedy approach.

Travelling Salesperson Algorithm

As the definition for greedy approach states, we need to find the best optimal solution locally to figure out the global optimal solution. The inputs taken by the algorithm are the graph G {V, E}, where V is the set of vertices and E is the set of edges. The shortest path of graph G starting from one vertex returning to the same vertex is obtained as the output.

Algorithm

- Travelling salesman problem takes a graph G {V, E} as an input and declare another graph as the output (say G') which will record the path the salesman is going to take from one node to another.
- The algorithm begins by sorting all the edges in the input graph G from the least distance to the largest distance.
- The first edge selected is the edge with least distance, and one of the two vertices (say A and B) being the origin node (say A).
- Then among the adjacent edges of the node other than the origin node (B), find the least cost edge and add it onto the output graph.
- Continue the process with further nodes making sure there are no cycles in the output graph and the path reaches back to the origin node A.
- However, if the origin is mentioned in the given problem, then the solution must always start from that node only. Let us look at some example problems to understand this better.

The Traveling Salesperson Problem (TSP) is a classic optimization problem where the goal is to find the shortest possible tour that visits a given set of cities and returns to the starting city. Due to its NP-hard nature, exact algorithms become impractical for large instances. Heuristic and search strategies are often employed to find near-optimal solutions efficiently. One common heuristic approach is the nearest neighbor algorithm, and a popular search strategy is the simulated annealing algorithm. Here's a brief overview and implementation in Python for both:

1. Nearest Neighbor Algorithm (Heuristic):

The nearest neighbor algorithm starts from a given city and repeatedly selects the nearest unvisited city until all cities are visited forming a tour

2. Simulated Annealing (Search Strategy):

Simulated Annealing is a probabilistic optimization algorithm inspired by the annealing process in metallurgy. It allows the algorithm to explore the solution space and escape local optima

Note: You need to implement the **generate_neighbor** function according to the specific neighborhood generation strategy you choose (e.g., swapping two cities). Adjust the initial solution, temperature, and cooling rate based on your problem size and characteristics.

3. Iterative steps:

Randomly perturb the current solution to generate a neighboring solution.

Calculate the change in objective function value (e.g., total distance) between the current and neighboring solutions.

If the neighboring solution is better, accept it as the new current solution.

If the neighboring solution is worse, accept it with a certain probability determined by the temperature and the change in objective function value.

Repeat for the specified number of iterations or until convergence.

4. Completion:

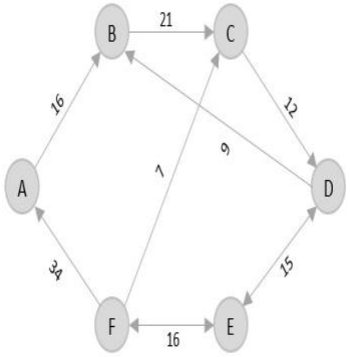
Return the best solution found.

This algorithm combines a greedy heuristic (nearest neighbor) for constructing an initial solution and a meta heuristic (simulated annealing) for further exploration and optimization. Simulated annealing allows the algorithm to escape local minima by accepting worse solutions probabilistically, especially at the early stages when the temperature is high

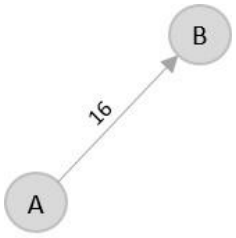
Example:

Consider the following graph with six cities and the distances between them −

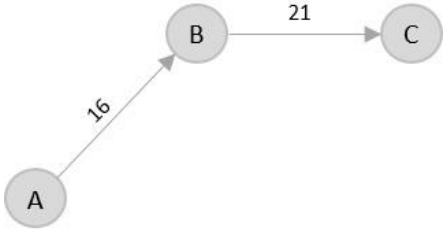
Laboratory Practice-II: 314458 (Artificial Intelligence)



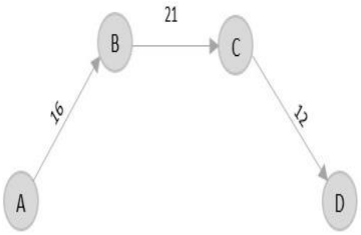
From the given graph, since the origin is already mentioned, the solution must always start from that node. Among the edges leading from A, A → B has the shortest distance.



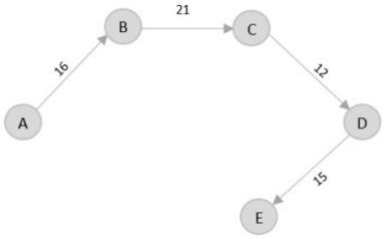
Then, B → C has the shortest and only edge between, therefore it is included in the output graph.



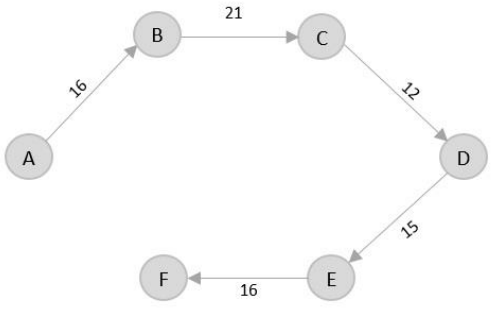
There's only one edge between C → D, therefore it is added to the output graph.



There's two outward edges from D. Even though, D → B has lower distance than D → E, B is already visited once and it would form a cycle if added to the output graph. Therefore, D → E is added into the output graph

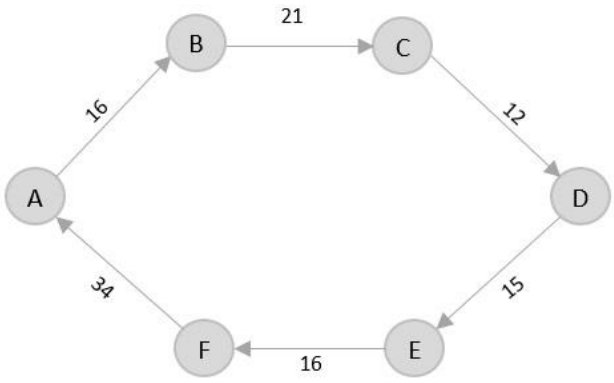


There's only one edge from e, that is E → F. Therefore, it is added into the output graph.



Again, even though F → C has lower distance than F → A, F → A is added into the output graph in order to avoid the cycle that would form and C is already visited once.

Laboratory Practice-II: 314458 (Artificial Intelligence)



The shortest path that originates and ends at A is A → B → C → D → E → F → A

The cost of the path is: 16 + 21 + 12 + 15 + 16 + 34 = 114.

Even though, the cost of path could be decreased if it originates from other nodes but the question is not raised with respect to that.

Sinhgad Technical Education Society's

RMD SINHGAD SCHOOL OF ENGINEERING, PUNE

Department of Information Technology

Academic Year: 2023-24

Third Year (Semester: VI)

Laboratory Practice-II: 314458 (Artificial Intelligence)

Implement n-queens problem using Hill-climbing / simulated annealing / A* algorithm etc. Write a program for Water jug problem / Towers of Hanoi

- While there are algorithms like Backtracking to solve N Queen problem, let's take an AI approach in solving the problem.
- It's obvious that AI does not guarantee a globally correct solution all the time but it has quite a good success rate of about 97% which is not bad.
- A description of the notions of all terminologies used in the problem will be given and are as follows:-
    - **Notion of a State** – A state here in this context is any configuration of the N queens on the N X N board. Also, in order to reduce the search space let's add an additional constraint that there can only be a single queen in a particular column. A state in the program is implemented using an array of length N, such that if **state[i]=j** then there is a queen at column index **i** and row index **j**.
    - **Notion of Neighbours** – Neighbours of a state are other states with board configuration that differ from the current state's board configuration with respect to the position of only a single queen. This queen that differs a state from its neighbour may be displaced anywhere in the same column.
    - **Optimisation function or Objective function** – We know that local search is an optimization algorithm that searches the local space to optimize a function that takes the state as input and gives some value as an output. The value of the objective function of a state here in this context is the number of pairs of queens attacking each other. Our goal here is to find a state with the minimum objective value. This function has a maximum value of NC2 and a minimum value of 0.

Algorithm:

1. Start with a random state(i.e, a random configuration of the board).
2. Scan through all possible neighbour's of the current state and jump to the neighbour with the highest objective value, if found any. If there does not exist, a neighbour, with objective strictly higher than the current state but there exists one with equal then jump to any random neighbour(escaping shoulder and/or local optimum).
3. Repeat step 2, until a state whose objective is strictly higher than all it's neighbour's objectives, is found and then go to step 4.
4. The state thus found after the local search is either the local optimum or the global optimum. There is no way of escaping local optima but adding a random neighbour or a random restart each time a local optimum is encountered increases the chances of achieving global optimum(the solution to our problem).
5. Output the state and return.

- It is easily visible that the global optimum in our case is 0 since it is the minimum number of pairs of queens that can attack each other. Also, the random restart has a higher chance of achieving global optimum but we still use random neighbour because our problem of N queens does not has a high number of local optima and random neighbour is faster than random restart.
- **Conclusion:**
  1. **Random Neighbour** escapes shoulders but only has a little chance of escaping local optima.
  2. **Random Restart** both escapes shoulders and has a high chance of escaping local optima.

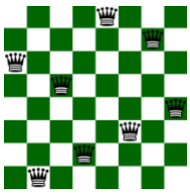
Complexity Analysis

- The time complexity for this algorithm can be divided into three parts:
  1. **Calculating Objective** – The calculation of objective involves iterating through all queens on board and checking the no. of attacking queens, which is done by our *calculate Objective* function in **O(N2)** time.

  2. **Neighbour Selection and Number of neighbours** – The description of neighbours in our problem gives a total of **N(N-1)** neighbours for the current state. The selection procedure is *best fit* and therefore requires iterating through all neighbours, which is again **O(N2)**.

  3. **Search Space** – Search space of our problem consists of a total of **NN** states, corresponding to all possible configurations of the **N** Queens on board. Note that this is after taking into account the additional constraint of one queen per column.

The **N** Queen is the problem of placing **N** chess queens on an **N×N** chessboard so that no two queens attack each other. For example, the following is a solution for **8** Queen problem.



Input: *N = 4*
Output:
0 1 0 0

0 0 0 1
1 0 0 0
0 0 1 0

Explanation:

The Position of queens are:
1 – {1, 2}
2 – {2, 4}
3 – {3, 1}
4 – {4, 3}
As we can see that we have placed all 4 queens
in a way that no two queens are attacking each other.
So, the output is correct

Input: *N = 8*
Output:
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 1 0 0 0

The N-Queens problem is a classic combinatorial optimization problem where the goal is to place N chess queens on an N×N chessboard in such a way that no two queens threaten each other. In other words, no two queens should share the same row, column, or diagonal

Simulated Annealing is a probabilistic optimization algorithm inspired by the annealing process in metallurgy. It is particularly useful for solving combinatorial optimization problems like the N-Queens problem. Simulated Annealing allows the algorithm to explore the solution space by accepting worse solutions with a certain probability, which decreases over time. This stochastic acceptance of worse solutions helps the algorithm avoid getting stuck in local optima.

Here's a high-level explanation of how Simulated Annealing can be applied to the N-Queens problem:

- Representation of State:

A state in this context represents a particular arrangement of queens on the chessboard.

- Objective Function:

The objective function measures the quality of a solution. In the case of N-Queens, it could be the number of pairs of queens that threaten each other.

- Initial States:

Generate an initial state randomly or using some heuristic.

- Neighbour Generation:

Define a way to generate neighboring states. In the context of N-Queens, a neighboring state can be generated by moving one queen to a different position.

Define a function to calculate the probability of accepting a worse solution. The probability decreases over time and is affected by the magnitude of the change in the objective function and a temperature parameter.

- Termination:

The algorithm terminates when a stopping criterion is met, such as reaching a maximum number of iterations or achieving a satisfactory solution.

This probabilistic nature makes Simulated Annealing a versatile and effective algorithm for solving a variety of optimization problems, including combinatorial problems like the N-Queens.

The N-Queens problem can be approached using the A* (A-star) algorithm, which is a popular search algorithm used in path finding and graph traversal problems. In the context of the N-Queens problem, the state space is represented by different configurations of queens on the chessboard.

Here's an outline of how the A* algorithm can be applied to the N-Queens problem:

- Sate Representation:

A state represents a particular configuration of queens on the chessboard.

- Initial State:

The initial state is a configuration with no queens placed.

- Goal Sate:

The goal state is a configuration where all N queens are placed on the board, and no two queens threaten each other.

- Actions:

The actions are the possible moves to transition from one state to another. In the context of N-Queens, an action involves placing a queen in a valid position.

- Transition Model:

Define a transition model that describes how the state changes when an action is applied

- Cost Function:

Design a heuristic function that estimates the cost from the current state to the goal. For the N-Queens problem, this could be the number of conflicting pairs of queens.

Start with the initial state and initialize the open list with it. While the open list is not empty:

Water Jug Problem

The Water Jug Problem involves two water jugs of different capacities and the goal is to measure out a specific quantity of water using these jugs. The problem is usually framed with the following constraints:

You have two jugs with capacities x and y liters, where x and y are positive integers.

The jugs do not have any markings on them

You have access to a water source (e.g., a tap).

The operations allowed are:

1. Fill a jug completely.
2. Empty a jug completely.

3. Pour water from one jug into another until the pouring jug is empty or the receiving jug is full.

The objective is to measure out a specific quantity of water using these operations. The problem might ask for the sequence of steps to achieve the desired quantity.

The Water Jug Problem can be solved using various algorithms, and one common approach is to use a depth-first search. Here's a simple algorithm for solving the Water Jug Problem:

```
Algorithm: WaterJugProblemDFS(x_capacity, y_capacity, target)
Input: Capacities of two jugs (x_capacity, y_capacity) and the target amount of water
to be measured.

1. Call DFS with initial state (0, 0) and an empty sequence of operations.

2. Function DFS(state, path):
   a. If state is the target state, return success and the sequence of operations (path).
   b. Mark the current state as visited.
   c. Generate possible next states by performing valid operations (filling, emptying,
pouring).
   d. For each next state:
      i. If the state is not visited:
         - Recursively call DFS with the next state and the updated path (path +
[operation]).
         - If the recursive call returns success, propagate the success.

3. Function to Generate Next States:
   a. Fill jug x: (x_capacity, y)
   b. Fill jug y: (x, y_capacity)
   c. Empty jug x: (0, y)
   d. Empty jug y: (x, 0)
   e. Pour water from x to y until y is full or x is empty: (max(0, x - (y_capacity -
y)), min(y + x, y_capacity))
   f. Pour water from y to x until x is full or y is empty: (min(x + y, x_capacity),
max(0, y - (x_capacity - x)))

4. Return failure if DFS completes without finding a solution.
```

Tower of Hanoi

The Towers of Hanoi is a classic problem that involves three pegs and a set of disks of different sizes. The disks are initially stacked in increasing size on one peg. The goal is to move the entire stack to another peg, obeying the following rules.

1. Only one disk can be moved at a time
2. A disk can only be moved to the top of another peg if it is smaller than the disk already on that peg or if the peg is empty.
3. Only the top disk of a peg can be moved.

The problem can be stated more formally

- You have three pegs (A, B, and C).
- There are **n** disks of different sizes.
- The disks are initially stacked in increasing order of size on peg A

The objective is to move the entire stack from peg A to peg C, following the rules mentioned above. The intermediary peg (B) can be used to facilitate the moves. Solving both problems often involves recursion and careful consideration of the allowable operations to reach the desired state. These problems are commonly used in algorithmic courses to teach recursion, problem-solving strategies, and algorithm analysis.

Here's an algorithm for solving the Tower of Hanoi problem using recursion:

```
Algorithm: TowerOfHanoi(n, source, auxiliary, target)
Input: Number of disks (n), source peg, auxiliary peg, and target peg.

1. If n == 1:
   a. Move the top disk from the source peg to the target peg.
   b. Return.

2. Recursively solve the TowerOfHanoi problem for n-1 disks:
   a. Move n-1 disks from the source peg to the auxiliary peg using the target peg as
the auxiliary peg.

3. Move the nth disk from the source peg to the target peg.

4. Recursively solve the TowerOfHanoi problem for n-1 disks:
   a. Move n-1 disks from the auxiliary peg to the target peg using the source peg as
the auxiliary peg.
```

This algorithm effectively breaks down the problem into sub problems, solving them recursively. The base case (n == 1) handles the situation where there is only one disk to move, and the recursive steps ensure that the Tower of Hanoi problem is solved for n-1 disks, allowing the movement of the nth disk, and then solving the sub problem again for the remaining disks

Write a program for sorting algorithms using appropriate knowledge representation and reasoning techniques

Sinhgad Technical Education Society's

RMD SINHGAD SCHOOL OF ENGINEERING, PUNE

Department of Information Technology

Academic Year: 2023-24

Third Year (Semester: VI)

Laboratory Practice-II: 314458 (Artificial Intelligence)

Sorting algorithms are typically implemented using traditional programming languages, and AI techniques are not commonly applied to the implementation details of sorting algorithms. However, knowledge representation and reasoning techniques in AI are often used in the design and analysis of algorithms, including sorting algorithms

Knowledge representation and reasoning (KRR) techniques are commonly associated with symbolic AI and are often used in problem-solving, knowledge-based systems, and expert systems. While sorting algorithms typically don't directly involve KRR techniques, understanding the principles of KRR can be beneficial when designing and analyzing algorithms. Here's an overview of how you might apply KRR concepts to sorting algorithms:

1. Problem modeling

- Knowledge representation: Use symbolic representations to model the sorting problem. For example, you can represent the input array as a set of logical statements or use formal notations to describe the properties of the elements.
- Reasoning: Apply logical reasoning to understand the relationships between elements in the array. For instance, if you know that one element is greater than another, use logical deduction to infer properties about their positions in the sorted order.

2. Algorithm Design:

- Knowledge representation: Represent the structure and operations of the sorting algorithm using formal notations. Use symbols and expressions to describe how the algorithm transforms the input data.
- Reasoning: Apply deductive reasoning to prove the correctness of the sorting algorithm. Use mathematical induction or other formal reasoning techniques to demonstrate that the algorithm produces the correct output for any valid input.

3. Complexity Analysis:

- Knowledge representation: Represent the time and space complexity of the sorting algorithm using Big-O notation or other formal complexity measures.
- Reasoning: Use deductive reasoning to analyze the efficiency of the algorithm in terms of time and space complexity. Infer how the algorithm's performance scales with the size of the input.

4. Heuristic and adaptive Techniques:

- Use symbolic representations to model heuristics or adaptive strategies that can be applied during the sorting process. Represent domain-specific knowledge that can guide the algorithm in making decisions.

- Apply reasoning techniques to dynamically adjust the sorting strategy based on the characteristics of the input data. For example, if the data is nearly sorted, adapt the algorithm to take advantage of this property.

5. Knowledge-Based Optimization:

- Represent optimization criteria and constraints using symbolic notations. Describe the desired properties of the sorted output
- Apply reasoning techniques to optimize the sorting algorithm based on the specified criteria. For example, if stability in sorting is crucial, modify the algorithm to ensure stable sorting.

While traditional sorting algorithms don't heavily leverage symbolic reasoning, the application of KRR techniques becomes more evident in more complex AI applications, such as knowledge-based systems or expert systems where sorting is just one component of a larger problem-solving process. The principles of KRR provide a foundation for designing algorithms that can incorporate domain knowledge and reason about the properties of the input data.

Write a program for the Information Retrieval System using appropriate NLP tools (such as NLTK, Open NLP, …)

a. Text tokenization
b. Count word frequency
c. Remove stop words
d. POS tagging

Laboratory Practice-II: 314458 (Artificial Intelligence)

a. Text tokenization is the process of breaking down a sequence of text into individual units, known as tokens. Tokens can be as small as individual characters or as large as words or phrases, depending on the specific requirements of the analysis. Tokenization is a fundamental step in natural language processing (NLP) and plays a crucial role in various language-related tasks. Here's an explanation of text tokenization using the NLTK (Natural Language Toolkit) library in Python:

NLTK's **word_tokenize** function takes a text input and breaks it into a list of words. It considers punctuation and whitespace to identify word boundaries.

```
words = word_tokenize(text)
```

Example Output

```
['This', 'is', 'an', 'example', 'sentence', '.', 'Tokenization', 'breaks', 'it', 'into', 'words', '.']
```

NLTK's **sent_tokenize** function is used to split a piece of text into sentences

```
sentences = sent_tokenize(text)
```

Example Output

```
['This is an example sentence.', 'Tokenization breaks it into words.']
```

Tokenization Challenges

Tokenization can be challenging, especially in languages with complex structures or in tasks involving social media data where text may not follow standard grammatical rules, By breaking down text into smaller units, tokenization facilitates subsequent NLP tasks, including part-of-speech tagging, named entity recognition, and sentiment analysis. It is an essential step in transforming raw text data into a format suitable for analysis and machine learning applications.

b. To count word frequency using natural language processing (NLP) tools in Python, you can use popular libraries such as NLTK (Natural Language Toolkit) or spaCy. Here's a simple example using NLTK:

Counting word frequency using natural language processing (NLP) tools involves analyzing a given text to determine how often each word appears. This process is essential for various NLP applications, including text summarization, sentiment analysis, and information retrieval. Here's

an explanation of how to count word frequency using appropriate NLP tools, focusing on the NLTK (Natural Language Toolkit) library in Python:

1. Tokenization:
- Breaks the text into individual words or tokens, making it easier to analyze.
2. Stop word Removal:
- Eliminates common words that may not provide significant information, focusing on meaningful words for analysis.
3. Frequency Distribution:
- The **FreqDist** class from NLTK helps count the occurrences of each word in the text.
4. Normalization:
- Converting words to lowercase (**text.lower()**) ensures case-insensitive counting.
5. Filtering:
- The list comprehension (**[word for word in words if word.isalpha() and word not in stop_words]**) filters out non-alphabetic words and stopwords.

c. In **natural language processing (NLP)**, **stopwords** are frequently filtered out to enhance text analysis and computational efficiency. Eliminating stopwords can improve the accuracy and relevance of NLP tasks by drawing attention to the more important words, or content words. The article aims to explore stopwords

Certain words, like "the," "and," and "is," are thought to be ineffective for communicating important information. The objective of eliminating words that add little or nothing to the comprehension of the text is to expedite text processing, even though the list of stopwords may differ.

What are Stop words?

A stop word is a commonly used word (such as "the", "a", "an", or "in") that a search engine has been programmed to ignore, both when indexing entries for searching and when retrieving them as the result of a search query.

We would not want these words to take up space in our database or take up valuable processing time. For this, we can remove them easily, by storing a list of words that you consider to stop words.

Need to remove the Stopwords

The necessity of removing stopwords in NLP is contingent upon the specific task at hand. For text classification tasks, where the objective is to categorize text into distinct groups,

excluding stopwords is common practice. This is done to channel more attention towards words that truly convey the essence of the text. As illustrated earlier, certain words like "there," "book," and "table" contribute significantly to the text's meaning, unlike less informative words such as "is" and "on."

Conversely, for tasks like machine translation and text summarization, the removal of stopwords is not recommended. In these scenarios, every word plays a pivotal role in preserving the original meaning of the content.

Types of Stopwords

Stopwords are frequently occurring words in a language that are frequently omitted from natural language processing (NLP) tasks due to their low significance for deciphering textual meaning. The particular list of stopwords can change based on the language being studied and the context. The following is a broad list of stopword categories:

- **Common Stopwords:** These are the most frequently occurring words in a language and are often removed during text preprocessing. Examples include "the," "is," "in," "for," "where," "when," "to," "at," etc.
- **Custom Stopwords:** Depending on the specific task or domain, additional words may be considered as stopwords. These could be domain-specific terms that don't contribute much to the overall meaning. For example, in a medical context, words like "patient" or "treatment" might be considered as custom stopwords.
- **Numerical Stopwords:** Numbers and numeric characters may be treated as stopwords in certain cases, especially when the analysis is focused on the meaning of the text rather than specific numerical values.
- **Single-Character Stopwords:** Single characters, such as "a," "I," "s," or "x," may be considered stopwords, particularly in cases where they don't convey much meaning on their own.
- **Contextual Stopwords:** Words that are stopwords in one context but meaningful in another may be considered as contextual stopwords. For instance, the word "will" might be a stopword in the context of general language processing but could be important in predicting future events.

Checking English Stopwords List

An English stopwords list typically includes common words that carry little semantic meaning and are often excluded during text analysis. Examples of these words are "the," "and," "is," "in," "for," and "it." These stopwords are frequently removed to focus on more meaningful terms when processing text data in natural language processing tasks such as text classification or sentiment analysis.

d. Part-of-speech (POS) tagging, also known as grammatical tagging, is the process of assigning a part-of-speech category (such as noun, verb, adjective, etc.) to each word in a given text. This task is crucial for natural language processing (NLP) applications, as it helps in understanding the

syntactic structure of sentences and is a key component in many downstream tasks like information extraction, sentiment analysis, and machine translation.

Tokenization

NLTK's **word_tokenize** function is used to break the input text into a list of words.

```
words = word_tokenize(text)
```

Example Output

```
['POS', 'tagging', 'helps', 'in', 'understanding', 'the', 'syntactic', 'structure', 'of', 'sentences', '.']
```

POS Tagging

NLTK's **pos_tag** function assigns a part-of-speech tag to each word in the list.

```
pos_tags = pos_tag(words)
```

Example Output:

```
[('POS', 'NN'), ('tagging', 'VBG'), ('helps', 'VBZ'), ('in', 'IN'), ('understanding', 'VBG'), ('the', 'DT'), ('syntactic', 'JJ'), ('structure', 'NN'), ('of', 'IN'), ('sentences', 'NNS'), ('.', '.')]
```

Each tuple in the result contains a word and its corresponding POS tag. For example, ('POS', 'NN') indicates that 'POS' is a noun, and ('tagging', 'VBG') indicates that 'tagging' is a verb in its gerund form

POS Tagset

POS tags follow a specific tagset. In the example, 'NN' represents a singular noun, 'VBG' is a verb in its gerund form, 'VBZ' is a third person singular present verb, 'JJ' is an adjective, 'IN' is a preposition, and 'NNS' is a plural noun.

Custom Tagsets

Some POS taggers allow for custom tagsets to suit specific applications or linguistic analysis requirements.

Sinhgad Technical Education Society's

RMD SINHGAD SCHOOL OF ENGINEERING, PUNE

Department of Information Technology

Academic Year: 2023-24

Third Year (Semester: VI)

Laboratory Practice-II: 314458 (Artificial Intelligence)

POS tagging is a crucial preprocessing step in many NLP applications, providing valuable information about the grammatical structure of the text. Different POS taggers may use different models or algorithms, and the choice of tagger may depend on the specific requirements of your application.

Write a program for the Tic-Tac-Toe game.

Laboratory Practice-II: 314458 (Artificial Intelligence)

Tic-Tac-Toe serves as an introductory platform for understanding and implementing various AI concepts, ranging from classical algorithms like Minimax to more modern approaches involving machine learning and neural networks. The simplicity of the game allows for a clear demonstration of these concepts before applying them to more complex scenarios.

Rules of the Game

- The game is to be played between two people (in this program between HUMAN and COMPUTER).
- One of the player chooses 'O' and the other 'X' to mark their respective cells.
- The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character ('O' or 'X').

0	X	0
0	X	X
X	0	X

- If no one wins, then the game is said to be draw. **Implementation** In our program the moves taken by the computer and the human are chosen randomly. We use rand() function for this. **What more can be done in the program?** The program is in not played optimally by both sides because the moves are chosen randomly. The program can be easily modified so that both players play optimally (which will fall under the category of Artificial Intelligence). Also the program can be modified such that the user himself gives the input (using scanf() or cin). The above changes are left as an exercise to the readers. **Winning Strategy – An Interesting Fact** If both the players play optimally then it is destined that you will never lose ("although the match can still be drawn"). It doesn't matter whether you play first or second.In another ways – " Two expert players will always draw ". Isn't this interesting ?

In the context of Artificial Intelligence (AI) and Tic-Tac-Toe, several theoretical concepts and techniques are relevant. Here's an overview:

6. State Space Representation:

- Representing the state of the Tic-Tac-Toe board is crucial. The game state is the primary input to AI algorithms. The board can be represented as an array, matrix, or another data structure.

7. Game Tree and Decision Tree:

- Tic-Tac-Toe can be modeled as a game tree, where nodes represent game states, and edges represent possible moves. Decision trees, derived from the game tree, help in determining the best move at each state

8. Min Max Algorithm:

- The Minimax algorithm is a fundamental concept in AI for games. It's a decision-making algorithm used to minimize the possible loss for a worst-case scenario. In Tic-Tac-Toe, Minimax evaluates all possible moves and selects the one that maximizes the player's chances of winning or minimizes the chances of losing

9. Alpha Beta Pruning:

- To optimize the Minimax algorithm and reduce the number of evaluated nodes, Alpha-Beta Pruning is often applied. This technique eliminates branches of the game tree that are guaranteed not to influence the final decision.

10. Heuristic Evaluation:

- While Minimax ensures optimal play, in practice, Tic-Tac-Toe's simplicity allows for exact computation. However, heuristic evaluation functions might be used in more complex games. These functions provide an approximate measure of the desirability of a particular game state

11. Symmetry Exploitation:

- Due to the symmetry of the Tic-Tac-Toe board, AI algorithms can take advantage of symmetrical positions to reduce the number of unique states to evaluate.

12. Q learning and Reinforcement Learning:

- In the context of machine learning, reinforcement learning techniques, such as Q-learning, can be applied to train agents to play Tic-Tac-Toe. The agent learns by interacting with the environment and receiving rewards based on the outcomes of its moves.

13. Neural Networks:

- While not strictly necessary for Tic-Tac-Toe, neural networks can be applied to learn optimal strategies or approximate value functions. Deep Q-learning or deep reinforcement learning methods might be explored for more complex games.

14. Monte Carlo Tree Search(MCTS):

- MCTS is a search algorithm used in decision processes to optimize decision-making in a way that balances exploration and exploitation. While Tic-Tac-Toe is simple enough for exhaustive search, MCTS becomes crucial in more complex games.

15. **Dynamic Programming:**

- Dynamic programming techniques may be applied for optimal policy computation in Tic-Tac-Toe, particularly when considering its small state space.

16. **Adversarial Search:**

- Tic-Tac-Toe falls under the category of adversarial search problems, where two players are in competition. Concepts like the minimax algorithm and alpha-beta pruning are part of adversarial search strategies.

step-by-step algorithm for a simple Tic-Tac-Toe game:

1. **Initialize the Board:**

- Initialize each cell with an empty space (' ') to indicate an available position.

2. **Print the Board:**

- Display the current state of the board after each move.

3. **Take Player Input**

- Alternate between players (X and O).For each player's turn: Prompt the player to enter the row and column where they want to make a move.
- Validate the input to ensure it's within the valid range (0 to 2) and the chosen cell is not already occupied.

4. **Update the Board:**

- Update the board with the player's symbol (X or O) at the specified row and column

5. **Check for Winner:**

- Check for a winner after each move
- Check each row to see if all elements are the same.
- Check each column to see if all elements are the same.
- Check both diagonals to see if all elements are the same.

6. **Check for Tie:**

- If the board is full and no winner is found, declare a tie.

7. **End the Game:**

- If there is a winner or a tie, display the final board and the result.

- If the game continues, go back to step 2.

8. Repeat or End:

- Ask players if they want to play again.

  If yes, go back to step 1.

  If no, end the game.