**RESEARCH PAPER**

**Graph Theory Applied to Plasma Chemical Reaction**

**Engineering**

**Thomas D. Holmes , Rachael H. Rothman, William B. Zimmerman1**

**1.My Understanding of the Research Paper:**

According to the research article, the authors investigate the use of graph theory in plasma chemical reaction engineering, with a specific focus on atmospheric pressure plasmas. They talk about creating a weighted directional graph with reaction data for air, visualizing the reaction network, and using graph algorithms like Dijkstra's shortest path algorithm. They also offer a unique technique, Optimal Condition Approaching via Reaction-In-Network Analysis (OCARINA), for determining the best circumstances for targeted chemical production.

One of the paper's important results is that graph theory can provide insights into complex plasma chemical reaction systems. By portraying reactions and species as nodes and edges in a graph, researchers can evaluate the connectivity between species, discover key reaction routes, and evaluate the possibilities for tailored chemical creation. The graph visualization provides a thorough picture of the reaction network by showing links between species and reactions.

**2.Limitations:**

However, despite the prospective possibilities, the research article highlights some limitations and problems. Here are some of the main constraints as follows:

**1.The Complexity of Plasma Chemistry:**

Plasma chemical reaction systems are extremely complicated, including multiple reactions and species. While graph theory provides a valuable framework for analysis, analysing and comprehending the data can be difficult due to the vast amount of information.

**2.Data Availability and Quality:**

The success of graph-based analysis is strongly dependent on the availability and quality of reaction data. Incomplete or faulty data might produce biased results and untrustworthy conclusions. Furthermore, experimental data on plasma chemical reactions may be scarce, making it difficult to validate theoretical predictions.

**3.Model Assumption Sensitivity:**

The techniques and models employed in graph-based analysis make specific assumptions about reaction kinetics and system behaviour. These assumptions may not always be valid in real-world plasma systems, resulting in inconsistencies between theoretical predictions and actual results.

**4.Scalability:**

As the size and complexity of plasma chemical reaction networks grow, the scalability of graph-based analysis becomes an issue. Large networks with hundreds of nodes and edges may necessitate large computational resources and time to analyse, limiting the applicability of the method.

**5.Interpretation of Results:**

Although graph-based analysis can offer insightful information about plasma chemistry, it may be necessary to know both graph theory and plasma physics to interpret the data and turn them into useful suggestions for process optimization or experimental design.

Despite these limitations, the study shows that graph theory has great potential as a useful tool for plasma chemical reaction engineering. Subsequent investigations may concentrate on overcoming these constraints by enhancing strategies for gathering data, honing modelling approaches, and creating more effective algorithms for graph-based analysis. All things considered, the work advances our knowledge of plasma chemistry and establishes the framework for further research in the area.

**NAME: BHAVISHYA GUPTA**

**ROLL NO: 220295**

**MID-TERM EVALUATION PROBLEMS**

**1. LEETCODE 1406 : STONE GAME III –**

class Solution {

public:

/\*

int game(vector<int>& stoneValue,int i)

{

int m=stoneValue.size();

if (i>=m) return 0;

int resvalue=stoneValue[i]-game(stoneValue,i+1);

if(i+1<m) resvalue=max(resvalue,stoneValue[i]+stoneValue[i+1]-game(stoneValue,i+2));

if(i+2<m) resvalue=max(resvalue,stoneValue[i]+stoneValue[i+1]+stoneValue[i+2]-game(stoneValue,i+3));

return resvalue;

}

\*/

string stoneGameIII(vector<int>& stoneValue)

{

int m = stoneValue.size();

vector<int> dp(m+1);

for(int i=m-1; i>=0; i--)

{

dp[i] = stoneValue[i]-dp[i+1];

if (i+2 <= m) dp[i] = max(dp[i],stoneValue[i]+stoneValue[i+1]-dp[i+2]);

if (i+3 <= m) dp[i] = max(dp[i],stoneValue[i]+stoneValue[i+1]+stoneValue[i+2]-dp[i+3]);

}

int diff=dp[0];

if (diff<0) return "Bob";

else if (diff>0) return "Alice";

return "Tie";

}

};

**2. LEETCODE 987 : VERTICAL ORDER TRAVERSAL OF A BINARY TREE-**

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode() : val(0), left(nullptr), right(nullptr) {}

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

\* };

\*/

class Solution {

public:

vector<vector<int>> verticalTraversal(TreeNode\* root)

{

map<int, map<int, multiset<int>>> nodes;

queue<pair<TreeNode\*, pair<int, int>>> todo;

todo.push({root, {0, 0}});

while(!todo.empty())

{

auto p = todo.front();

todo.pop();

TreeNode\* temp = p.first;

int x = p.second.first;

int y = p.second.second;

nodes[x][y].insert(temp->val);

if(temp->left) todo.push({temp->left, {x-1, y+1}});

if(temp->right) todo.push({temp->right,{x+1, y+1}});

}

vector <vector<int>> result;

for(auto p: nodes){

vector<int> col;

for(auto q: p.second) col.insert(col.end(),q.second.begin(),q.second.end());

result.push\_back(col);

}

return result;

}

};

**3. LEETCODE 114 ; FLATTEN BINARY TREE TO LINKED LIST –**

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode() : val(0), left(nullptr), right(nullptr) {}

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

\* };

\*/

struct TreeNode \* newNode(int val)

{

struct TreeNode \* TreeNode = (struct TreeNode \* ) malloc(sizeof(struct TreeNode));

TreeNode -> val = val;

TreeNode -> left = NULL;

TreeNode -> right = NULL;

return TreeNode;

}

class Solution {

TreeNode \* prev = NULL;

public:

void flatten(TreeNode \* root)

{

if (root == NULL) return;

flatten(root -> right);

flatten(root -> left);

root -> right = prev;

root -> left = NULL;

prev = root;

}

};

**4. LEETCODE 909 : SNAKES AND LADDERS –**

class Solution {

public:

int n;

pair<int, int> getCoord(int s) {

int row = n-1-(s-1)/n;

int col = (s-1)%n;

if((n%2==1 && row%2==1)||(n%2==0 && row%2==0))

col = n-1-col;

return make\_pair(row, col);

}

int snakesAndLadders(vector<vector<int>>& board)

{

n = board.size();

int steps = 0;

queue<int> que;

vector<vector<bool>> visited(n, vector<bool>(n, false));

visited[n-1][0] = true;

que.push(1);

vector<bool> seen(n\*n+1,false);

while(!que.empty())

{

int N = que.size();

while(N--)

{

int x = que.front();

que.pop();

if(x == n\*n)

return steps;

for(int k = 1; k<=6; k++)

{

if(x+k > n\*n)

break;

pair<int, int> coord = getCoord(x+k);

int r = coord.first;

int c = coord.second;

if(visited[r][c] == true)

continue;

visited[r][c] = true;

if(board[r][c] == -1)

que.push(k+x);

else

{

que.push(board[r][c]);

}

}

}

steps++;

}

return -1;

}

};