Simutech Mid Term Evaluation

By Madhav Lata

**Reading Material ➖**

What I understood on reading the research paper is that it seems Plasma Chemical Reactions are really useful for different things like treating biomass and working with reactors. But there's a big problem: it is hard to control these reactions when the pressure is normal. This makes it tough to choose the right reaction path.

Scientists are trying to fix this by using things like plasma catalysis and special waveforms. But understanding these reactions is tough as there are so many different reactions and substances involved and is like trying to figure out the most important roads on a map when there are lots of roads to choose from.

They're using graph theory to help. By representing chemical reactions and species interactions as nodes and edges in a graph, we can visualize and analyze the intricate network of reactions. This helps them see and understand all the different reactions and how they're connected. It's like drawing a map of all the roads to see which ones are most important.

**Limitations of Dijkstra's Algorithm in Plasma Chemical Reaction Engineering**

Graph theory is really helpful for understanding plasma chemical reactions, but it has some downsides, especially when it comes to using algorithms like Dijkstra's shortest path one. In plasma reactions, things can get pretty complicated because they change a lot and are affected by stuff like energy, pressure and temperature. So, Dijkstra's algorithm might not show the whole picture accurately.

For example, Dijkstra's algorithm finds the shortest path between two points based on how fast reactions happen. But it only looks at the fastest pathways and overlooks all other pathways, which might not always be the most important ones. It also doesn't do well at considering how conditions change their effects on reaction rates.

It may provide misleading results when applied to systems where reaction kinetics are strongly influenced by changing conditions.

Also it relies on rate coefficients, which might not fully capture how different reactions and species interact and behave that too in really dynamic plasma environments.

Dijkstra's algorithm is efficient for small to moderately sized graphs but it may encounter computational challenges while scaling to large-scale networks with hundreds or thousands of nodes and edges.

Problems ➖

1. Stone game -

class Solution {

public:

string stoneGameIII(vector<int>& stoneValue) {

int n = stoneValue.size();

int a = 0;

int b = 0;

int c = 0;

for(int i = n-1; i>=0; i--) {

int result = INT\_MIN;

result = max(result, stoneValue[i] - b);

if(i+2 <= n)

result = max(result, stoneValue[i] + stoneValue[i+1] - b);

if(i+3 <= n)

result = max(result, stoneValue[i] + stoneValue[i+1] + stoneValue[i+2] - c);

c = b;

b = a;

a = result;

}

int diff = a;

if(diff < 0)

return "Bob";

else if(diff > 0)

return "Alice";

return "Tie";

}

};

1. Snakes and Ladders

class Solution {

public:

int n ;

map<int,int> go ;

void handle(int sum , int op , queue<array<int,2>> &q){

if(go.find(sum) != end(go)) q.push({go[sum],op + 1}) ;

else q.push({sum,op + 1}) ;

}

int bfs(int pos){

queue<array<int,2>> q ;

q.push({pos,0}) ;

vector<int> visited(n\*n + 1,0) ;

while(q.size()){

auto [num,op] = q.front() ; q.pop() ;

if(num == n\*n) return op ;

if(visited[num]) continue ;

visited[num] = 1 ;

if(num + 1 <= n\*n ) handle(num + 1 , op , q) ;

if(num + 2 <= n\*n ) handle(num + 2 , op , q) ;

if(num + 3 <= n\*n ) handle(num + 3 , op , q) ;

if(num + 4 <= n\*n ) handle(num + 4 , op , q) ;

if(num + 5 <= n\*n ) handle(num + 5 , op , q) ;

if(num + 6 <= n\*n ) handle(num + 6 , op , q) ;

}

return -1 ;

}

int snakesAndLadders(vector<vector<int>>& grid) {

n = grid.size() ;

bool leftToRight = true ;

for(int i = n - 1 ; i >= 0 ; --i ){

for(int j = 0 ; j < n ; ++j ){

if(grid[i][j] == -1) continue ;

int spot = 0 , row = n - i - 1 ;

if(leftToRight) spot = row\*n + j + 1 , go[spot] = grid[i][j] ;

else spot = row\*n + n - j , go[spot] = grid[i][j] ;

}

leftToRight = !leftToRight ;

}

return bfs(1);

}

};

1. Binary Tree to Linked List

class Solution {

public:

void link(TreeNode\* root,TreeNode\* &end){

if(!root) return;

TreeNode\* temp = root->right;

end = root;

if(root->left){

root->right = root->left;

root->left = NULL;

link(root->right,end);

end->right = temp;

}

link(temp,end);

}

void flatten(TreeNode\* root) {

TreeNode\* end=NULL;

link(root,end);

}

};

1. Vertical Order Traversal -

class Solution {

public:

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

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

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

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

while(!q.empty()){

auto it=q.front();

q.pop();

TreeNode\* p=it.first;

int x=it.second.first;

int y=it.second.second;

mpp[x][y].insert(p->val);

if(p->left){

q.push({p->left,{x-1,y+1}});

}

if(p->right){

q.push({p->right,{x+1,y+1}});

}

}

vector<vector<int>>ans;

for(auto it:mpp){

vector<int>col;

for(auto ab:it.second){

col.insert(col.end(),ab.second.begin(),ab.second.end());

}

ans.push\_back(col);

}

return ans;

}

};