Experiment No : 1

Aim : Write a C/C++/Java program to solve missionaries and cannibal problem using Breadth First Search Technique.

Theory :

Breadth First Search

Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighboring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

The algorithm of breadth first search is given below. The algorithm starts with examining the node A and all of its neighbors. In the next step, the neighbors of the nearest node of A are explored and process continues in the further steps. The algorithm explores all neighbors of all the nodes and ensures that each node is visited exactly once and no node is visited twice.

State Space Search

State space search is a process used in the field of computer science, including artificial intelligence (AI), in which successive configurations or states of an instance are considered, with the intention of finding a goal state with a desired property.

Problems are often modeled as a state space, a set of states that a problem can be in. The set of states forms a graph where two states are connected if there is an operation that can be performed to transform the first state into the second.

State space search often differs from traditional computer science search methods because the state space is implicit: the typical state space graph is much too large to generate and store in memory. Instead, nodes are generated as they are explored, and typically discarded thereafter. A solution to a combinatorial search instance may consist of the goal state itself, or of a path from some initial state to the goal state.

Algorithm :

1. Initialize queue Q with the initial state
2. While queue Q is not empty
    - 2.1 Delete the element from Q and call it S.
    - 2.2 If S is goal state then exit.
    - 2.3 else get a list of rules applicable to S and store it in T.
    - 2.4 If T is empty then return failure.
    - 2.5 for each rule in T do
        - **I.** Apply that rule to S and get resultant R.
        - **II.** If T is goal state then exit.
        - **III.** If T is not already seen then insert T into Q.
3. Exit Failure.

Code :

```
#include <iostream>
#include<list>
#include<map>
#include<string>
using namespace std;

bool DEBUG=false;
list<struct node> queue;
map<string,bool> visited;
int M=3,C=3,B=2;


struct node{
int m;
int c;
char side;
};


bool seen(int m, int n, char side){
// Checks if the node has already been encountered
string temp=to_string(m)+","+to_string(n)+side;
if(visited.count(temp)>0){
return true;
}
visited[temp]=true;
return false;
}
bool moreCannibals(int m, int n){
// Checks if in this state(m,n), either side of the
// river has more cannibals that missionaries
if(m!=0 && m<n){
return true;
}
m=M-m;
n=C-n;
if(m!=0 && m<n){
return true;
}
return false;
}

bool isGoal(int m,int n ,char c){
// Check if goal state is reached
if(m==0 && n==0){
return true;
}
return false;
}
char sideComplement(char side){
```

```
//Returns the complement of current side
if(side=='R')
return 'L';
return 'R';
}

bool isValid(int m, int c,int i,int j, char side){
// Checks if a particular new state is valid
// Is the number of passengers within the capacity of
// the boat and greater than 0?
int numPassengers=0;
if(DEBUG){cout<<i<<" "<<j<<" "<<endl;}
numPassengers=(m-i)+(c-j);
if(numPassengers>B || numPassengers==0){
if(DEBUG){cout<<"not boat"<<endl;}
return false;
}

// more cannibals?
if(moreCannibals(i,j)){
if(DEBUG){cout<<"more cannibals"<<endl;}
return false;
}

//already seen?
if(seen(i,j,sideComplement(side))){
if(DEBUG){cout<<"Already seen"<<endl;}
return false;
}
return true;
}

bool mutate(int m, int c, char side){
// Checks all possible values for number of missionaries
// and cannibals and if the state is valid it pushes it to the
// back of the queue
int i,j;
cout<<"Mutating "<<m<<" "<<c<<" "<<side<<endl;
if(side=='L'){
// Going to the right
for(i=m;i>=m-B && i>=0;i--){
for(j=c;j>=c-B && j>=0;j--){
if(isValid(m,c,i,j,side)){
// is goal ?
if(isGoal(i,j,sideComplement(side))){
cout<<"Reached goal";
return true;
}
struct node temp={i,j,sideComplement(side)};
cout<<"Pushing "<<i<<" "<<j<<" "<<sideComplement(side)<<endl;
queue.push_back(temp);
```

```
}
}
}
}
else{
//Going left
for(i=m;i<=m+B && i<=M;i++){
for(j=c;j<=c+B&& j<=C;j++){
if(isValid(m,c,i,j,side)){
// is goal ?
if(isGoal(i,j,sideComplement(side))){
cout<<"Reached goal"<<endl;
return true;
}
struct node temp={i,j,sideComplement(side)};
cout<<"Pushing "<<i<<" "<<j<<" "<<sideComplement(side)<<endl;
queue.push_back(temp);
}
}
}

}
return false;
}
int main(){
struct node init={M,C,'L'};
queue.push_back(init);

//Add initial node to visited
string temp=to_string(M)+","+to_string(C)+'L';
visited[temp]=true;

bool goal=false;
while(!queue.empty() && !goal){
struct node s=queue.front();
queue.pop_front();
goal=mutate(s.m,s.c,s.side);
}
}
```

Output :
Mutating 3 3 L
Pushing 3 2 R
Pushing 3 1 R
Pushing 2 2 R
Mutating 3 2 R
Mutating 3 1 R
Pushing 3 2 L
Mutating 2 2 R
Mutating 3 2 L
Pushing 3 0 R
Mutating 3 0 R
Pushing 3 1 L
Mutating 3 1 L
Pushing 1 1 R
Mutating 1 1 R
Pushing 2 2 L
Mutating 2 2 L
Pushing 0 2 R
Mutating 0 2 R
Pushing 0 3 L
Mutating 0 3 L
Pushing 0 1 R
Mutating 0 1 R
Pushing 0 2 L
Pushing 1 1 L
Mutating 0 2 L

Conclusion : This Experiment was successfully executed and the output was verified.

Deepraj Bhosale
181105016