Name:	Navin Kumar Adhikari
Email:	navinadhikari@nevada.unr.edu
Title	Assignment 0 (CS776)

State Space Representation of Machineries and Cannibals Problem

The state space representation of Machineries and Cannibals Problem is ordered pair of Machineries and Cannibals with corresponding state ID as indicated below in figure.

M	C	ID

Fig: typical Node of State Space Diagram

Where, M= Number of Machineries
C=Number of Cannibals
ID = State ID of that particular Node

Case I: Initial (M, C) = (3, 3), and boat capacity of up to 2 persons

The number of valid actions from left side of the river to right side of the river are:
CCR = two cannibals to right
MMR = two mercenaries to right
MCR = one mercenary and one cannibal to right
MR = one mercenary to right
CR = one cannibal to right
The are many other actions that can be applied to the particular state. These actions are not considered for the clarity of state space diagram as these moves do not generate a new state in the state space graph, thus, do not affect the optimal solution.

Case II: Initial (M, C) = (3, 3), and boat capacity of up to 3 persons

The number of valid actions from left side of the river to right side of the river are:
CCCR = three cannibals to right
MMMR = three mercenaries to right
CMMR = one cannibal and two mercenaries to right
CCMR = one mercenaries and two cannibals to right
CCR = two cannibals to right
MMR = two mercenaries to right
MCR = one mercenary and one cannibal to right
MR = one mercenary to right
CR = one cannibal to right
The are many other actions that can be applied to the particular state. These actions are not considered for the clarity of state space diagram as these moves do not generate a new state in the state space graph, thus, do not affect the optimal solution.

Solution to the questions:

Q. Is it a good idea to check for repeated states?
Answer: In my opinion, it is, obviously, not a good idea to check for repeated states as it costs more space as well as computation time to end up with a solution. As passing through the same state we are consuming unnecessary time to reach the destination. Furthermore, we have to save number of repeated

state to traverse which take greater computers memory. This is not a good idea. As such, we have to eliminate repeated state before applying any algorithm to save space as well as computational time.

Q. Why do you think people have a hard time solving this puzzle given that the state space is so simple?
Answer: The biggest challenge one can find in this puzzle is the boat capacity to take Machinery and Cannibal to another side of the river as well as the limited options for the combinations of Machineries and Cannibals as one cannot the move Machineries and Cannibals haphazardly. The moves should be carefully scrutinized to avoid the condition of Machinery being outnumbered by Cannibals. As we see, when we increased the boat size from 2 to 3, the solution became quite easier.

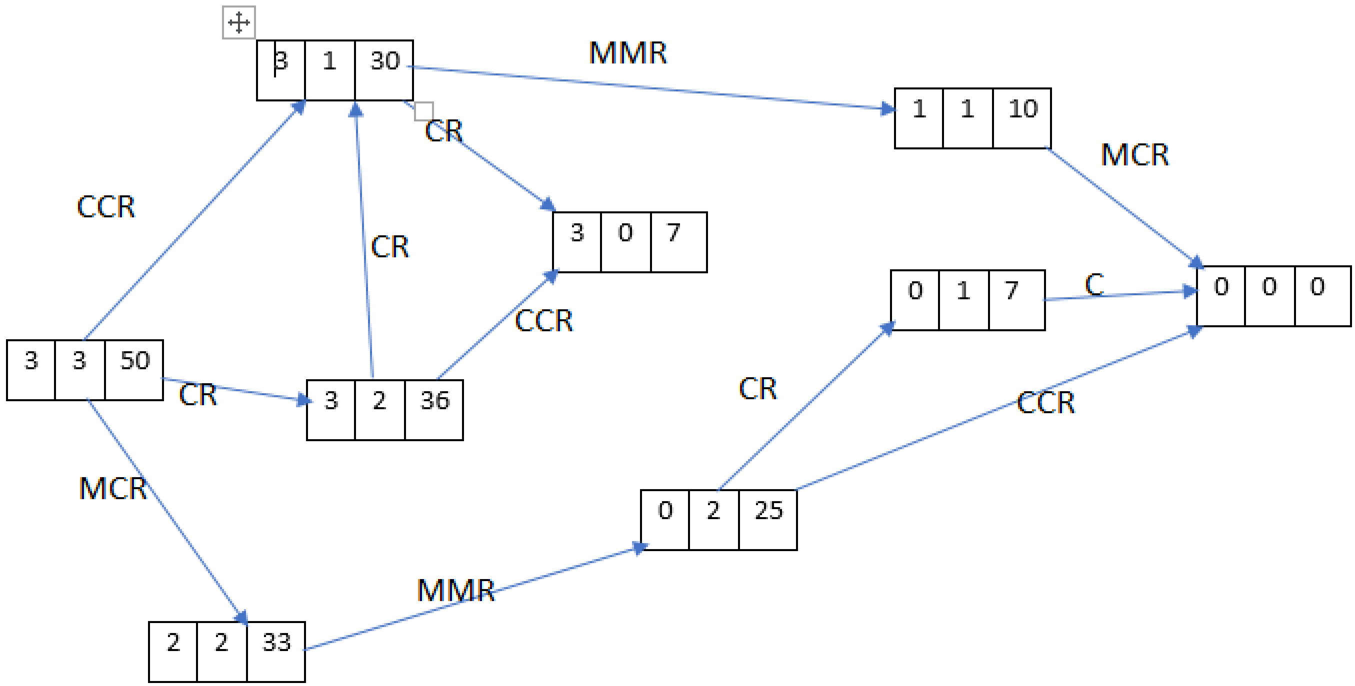

Fig: State Space Diagram for machinaries and Cannibals with Boat capacity 2

M	C	ID

Where,

M=Number of Machinaries

C=Number of Cannibals

ID=State ID

R= Towards right side of river

Code for Case I: 3M, 3C with Boat Capacity=2

```
#include<iostream>
#include<cmath>
#include<cstdlib>
using namespace std;
//Number of possbile valid actions from left part of the river to the right part.
#define CCR 0
#define MMR 1
#define CR  2
#define MR  3
#define MCR 4
#define NODE_LIMIT  51 //Total possible nodes with including nodes.
#define NUM_OF_MOVE 5

struct Node{
 int nodeNum;
 int mNum;
 int cNum;
 Node *next;
};
/*
 A linked list fucntion that joins childs nodes to theier corresponding parent nodes.
*/
struct Node *addadjnode (Node *currentNode, int newNodeNum, int newcNum, int newmNum)
{
   Node *newNode = new Node();
   newNode->nodeNum = newNodeNum;
   newNode->mNum = newmNum;
   newNode->cNum = newcNum;
   newNode->next   = currentNode;
   return newNode;

}
/*
  This function checks if the Action is valid or not.
*/
int IsValidMove(  int c, int m, int cOnboat, int mOnboat)
{
   int cL, mL, cR, mR;
   cL = c - cOnboat;
   mL = m - mOnboat;
   cR = 3 - c + cOnboat;
   mR = 3 - m + mOnboat;

   if(((cL <= mL) || (cL == 0) || (mL == 0)) && ((cR <= mR) || (cR == 0) || (mR == 0)))
   {
      return (1);
   }
   else
   {
      return (0);
   }
}
/*
  A function that returns node number
*/
int  assignnodenum (int c, int m)
{
   if((c == 3) && (m == 3)){
      return(50);
   }
```

```
    else if((c == 3) && (m == 2)){
        return(20); //
    }
    else if((c == 3) && (m == 1)){
        return(18);
    }
    else if((c == 3) && (m == 0)){
        return(41);
    }
    else if((c == 2) && (m == 3)){
        return(36);
    }
    else if((c == 2) && (m == 2)){
        return(33);
    }
    else if((c == 2) && (m == 1)){
        return(42);
    }
    else if((c == 2) && (m == 0)){
        return(25);
    }
    else if((c == 1) && (m == 3)){
        return(30);
    }
    else if((c == 1) && (m == 2)){
        return(17);
    }
    else if((c == 1) && (m == 1)){
        return(10);
    }
    else if((c == 1) && (m == 0)){
        return(5);
    }
    else if((c == 0) && (m == 3)){
        return(7);
    }
    else if((c == 0) && (m == 2)){
        return(13);
    }
    else if((c == 0) && (m == 1)){
        return(16);
    }
    else if((c == 0) && (m == 0)){
        return(0);
    }
    else{
        return(9);
    }
}

/*
  This function takes the # cannibals (c) and mercenaries (m) along with the action.
  Based on the value of (c, m) and move next state ID is generated.
*/
int GenState(int *c, int *m, int action)
{
    int can, mer;
    can = *c;
    mer = *m;
    switch(action){
        case 0: //CCR
                if(IsValidMove(can, mer, 2, 0)){
```

```
            can = *c - 2;
            mer = *m;
            *c = can;
            *m = mer;
            return(assignnodenum(can, mer));
        }
        else{
            return (255);
        }
        break;
    case 1: //MMR
        if(IsValidMove(can, mer, 0, 2)){
            can = *c;
            mer = *m - 2;
            *c = can;
            *m = mer;
            return(assignnodenum(can, mer));
        }
        else{
            return (255);
        }
        break;
    case 2: //CR
        if(IsValidMove(can, mer, 1, 0)){
            can = *c - 1;
            mer = *m;
            *c = can;
            *m = mer;
            return(assignnodenum(can, mer));
        }
        else{
            return (255);
        }
        break;
    case 3: //MR
        if(IsValidMove(can, mer, 0, 1)){
            can = *c;
            mer = *m - 1;
            *c = can;
            *m = mer;
            return(assignnodenum(can, mer));
        }
        else{
            return (255);
        }
        break;
    case 4: //MCR
        if(IsValidMove(can, mer, 1, 1)){
            can = *c - 1;
            mer = *m - 1;
            *c = can;
            *m = mer;
            return(assignnodenum(can, mer));
        }
        else{
            return (255);
        }
        break;
    default:
        return (255);
        break;
}
```

```
}

/*
  This function removes the repeated nodes in the currentPool
  to avoid the multiple execution of same node during the
  state space generation algorithm.

*/
void FilterCurrentPool(int currentPool[])
{
    int i, j, k;
    //check for the repeated one in the currentPool
    for(j = 0; currentPool[j] != -1; j++)
    {
     for(i = j+1; currentPool[i] != -1; i++)
     {
      if(currentPool[j] == currentPool[i])
      {
       for(k = j; k < NODE_LIMIT-1; k++)
       {
        currentPool[k] = currentPool[k+1];
       }
       currentPool[NODE_LIMIT-1] = -1;
      }
     }
    }
}

/*
  This function is used in state space (or graph) generation algorithm.
  This function checks if the nextPool of the graph nodes is empty or not.
  If the nextPool is empty, then the state space generation is completed.
*/
int IsEmpty(int nextPool[])
{
 int i;
 for(i = 0; i < NODE_LIMIT; i++)
 {
  if(nextPool[i] > 0)
  {
   return (0);
  }
 }
 return (1);
}
void BreadthFirstSearch( Node * adjacencyList[],int parent[],int level[])
{
    Node * traverse;
    int i, par, lev, flag = 1;
    // 'lev' represents the level to be assigned
    // 'par' represents the parent to be assigned
    // 'flag' used to indicate if graph is exhausted
    lev = 0;
    level[NODE_LIMIT-1] = lev;
    // We start at startVertex

    while (flag) {
        flag = 0;
        for (i = 0; i < NODE_LIMIT; ++i) {
            if (level[i] == lev) {
                flag = 1;
                traverse = adjacencyList[i];
```

```
            par = i;

            while (traverse != NULL) {
               if (level[traverse->nodeNum] != -1) {
                  traverse = traverse->next;
                  continue;
               }
               level[traverse->nodeNum] = lev + 1;
               parent[traverse->nodeNum] = par;
               traverse = traverse->next;
            }
         }
      }
      ++lev;
   }
}

int main(void)
{
   int  i  = 0, j  = 0,k  = 0, v1 = 0, v2 = 0,c=3,m=3,tempC=0,tempM=0;
   int  flag = 0,currentPool[NODE_LIMIT],nextPool[NODE_LIMIT],path[6];
   int  move[7] = {CCR, MMR, CR, MR, MCR};
   int cValue[NODE_LIMIT], mValue[NODE_LIMIT],ath[NODE_LIMIT];
   int p, levelCount,parent[NODE_LIMIT], level[NODE_LIMIT];
   Node *adjList[NODE_LIMIT];
   Node *traverse;
   //Initialization
   for(i = 0; i < NODE_LIMIT; i++){
      adjList[i]= NULL, currentPool[i] = -1;
      nextPool[i]= -1,parent[i]= 50,level[i]= -1;
   }
    //set up thethe first parent Node
    v1 = assignnodenum(c, m);
    for(i = 0; i < NUM_OF_MOVE; i++){
       tempC = c;
       tempM = m;
       v2 = GenState(&tempC, &tempM, move[i]);
       if(v2 != 0b11111111)
       {
          adjList[v1] = addadjnode(adjList[v1], v2, tempC, tempM);
       }
    }
    currentPool[0] = v1;
    while(1)
    {
     k = 0;
     for(j = 0; currentPool[j] != -1; j++)
     {
      v1 = currentPool[j];
      traverse = adjList[v1];
      while(traverse != NULL)
      {
       v1 = traverse->nodeNum;
        //check for whether the node is already build.
       for(i = 0; i < NODE_LIMIT; i++)
       {
        if((v1 == currentPool[i]) || (v1 == nextPool[i]))
        {
         flag = 1;
         break;
        }
        else
```

```
      {
       flag = 0;
      }
     }
     if(flag == 1)
     {
      traverse = traverse->next;
      continue;
     }

     nextPool[k++] = v1;

     c = traverse->cNum;
     m = traverse->mNum;
     for(i = 0; i < NUM_OF_MOVE; i++)
      {
        tempC = c;
        tempM = m;
        v2 = GenState(&tempC, &tempM, move[i]);
        if(v2 != 0b11111111)
        {
           adjList[v1] = addadjnode(adjList[v1], v2, tempC, tempM);
        }
     }
     traverse = traverse->next;
    } //while(traverse)
   } //for(j)
   //is nextPool empty
   if(IsEmpty(nextPool))
   {
    break;
   }
   //copy nextPool into currentPool
   for(i = 0; i < NODE_LIMIT; i++ )
   {
    if(i<k)
    {
     currentPool[i] = nextPool[i];
     nextPool[i]    = -1;
    }
    else
    {
     currentPool[i] = -1;
     nextPool[i]    = -1;
    }
   }
   FilterCurrentPool(currentPool);

  } //while

  flag = 1;
  //Printing adjancency list
  for(i = 50; i >= 0; i--){
      traverse = adjList[i];
      if(traverse != NULL)
      {
         cout<<"Node"<<i<<"--->";
      }
      flag = 0;
      while(traverse != NULL){
         cout<<"Node"<<"["<<traverse->nodeNum<<"]"<<"c="<<traverse->cNum<<","<<"m="<< traverse->mNum<<"  ---
> ";
```

```
        traverse = traverse->next;
        flag = 1;
      }
      if(flag == 1)
      {
        cout<<"NULL\n\n";
      }
   }
  BreadthFirstSearch(adjList, parent, level);
  levelCount = level[0];
  path[0] = 0;
  for(i = 1; i <= levelCount; i++)
  {
     path[i] = parent[path[i-1]];
  }
  int cn[5],me[5];
  cout<<"\nThe States Forming Optimal Path in State Space Diagram  -\n";
  for (i = 0; i <= levelCount; i++) {
      switch(path[i])
      {
      case 0:
             cn[i]=0;
             me[i]=0;

             break;
      case 10:
            cn[i]=1;
            me[i]=1;
            break;
      case 15:
            cn[i]=1;
            me[i]=3;
            break;
      case 50:
            cn[i]=3;
            me[i]=3;
            break;
      }
     cout<<"stateID= "<<path[i]<<" and (c,m)= "<<cn[i]<<","<<me[i]<<"\t";

  }
    cout<<"\n\n";
    cout<<"\n~~~~~~~~~The optimal solution for missionary and cannibals problem is~~~~~~~~~\n";
    cout<<"\nState(c="<<cn[levelCount]<<","<<" m="<<me[levelCount]<<")";
    for (int k = levelCount; k > 0; k--) {
     c = abs(cn[k] - cn[k-1]);
     m = abs(me[k] - me[k-1]);
     cout<<"-->Action(c="<<c<<","<<"m="<<m<<")"<<"-->State(c="<<cn[k-1]<<","<<"m="<<me[k-1]<<")";
  }
  return 0;
}
```

navin@navin-Aspire-F5-573G: ~/Desktop/CS 776/Assignment 1

```
navin@navin-Aspire-F5-573G:~/Desktop/CS 776/Assignment 1$ g++ 1.cpp
navin@navin-Aspire-F5-573G:~/Desktop/CS 776/Assignment 1$ ./a.out
Node50--->Node[33]c=2,m=2 ---> Node[36]c=2,m=3 ---> Node[30]c=1,m=3 ---> NULL

Node36--->Node[33]c=2,m=2 ---> Node[30]c=1,m=3 ---> Node[7]c=0,m=3 ---> NULL

Node33--->Node[10]c=1,m=1 ---> Node[25]c=2,m=0 ---> NULL

Node30--->Node[7]c=0,m=3 ---> Node[10]c=1,m=1 ---> NULL

Node25--->Node[5]c=1,m=0 ---> Node[0]c=0,m=0 ---> NULL

Node10--->Node[0]c=0,m=0 ---> Node[5]c=1,m=0 ---> NULL

Node5--->Node[0]c=0,m=0 ---> NULL


The States Forming Optimal Path in State Space Diagram  -
stateID= 0 and (c,m)= 0,0       stateID= 10 and (c,m)= 1,1      stateID= 30 and (c,m)= -1452763752,-1498281944  stateID= 50 and (c,m)= 3,3


~~~~~~~~~The optimal solution for missionary and cannibals problem is~~~~~~~~~

State(c=3, m=3)-->Action(c=1452763755,m=1498281947)-->State(c=-1452763752,m=-1498281944)-->Action(c=1452763753,m=1498281945)-->State(c=1,m=1)-->Action(c=1,m=1)-->State(c=0,m=0)navin@navin-Aspire-F5-573G:~/Desktop/CS 776/Assignment 1$
```

Code for Case II: 3M, 3C with Boat Capacity=3

```
#include<iostream>
#include<cmath>
#include<cstdlib>
using namespace std;
//Number of possbile valid actions from left part of the river to the right part.
 #define CCCR   0
 #define MMMR   1
 #define CMMR   2
 #define CCMR   3
 #define CMR    4
 #define MMR    5
 #define CCR    6
 #define CR     7
 #define MR     8
#define NODE_LIMIT  51 //Total possible nodes with including nodes.
#define NUM_OF_MOVE 9

struct Node{
 int nodeNum;
 int mNum;
 int cNum;
 Node *next;
};
/*
 A linked list fucntion that joins childs nodes to theier corresponding parent nodes.
*/
struct Node *addadjnode (Node *currentNode, int newNodeNum, int newcNum, int newmNum)
{
    Node *newNode = new Node();
    newNode->nodeNum = newNodeNum;
    newNode->mNum = newmNum;
    newNode->cNum = newcNum;
    newNode->next   = currentNode;
    return newNode;

}
/*
  This function checks if the Action is valid or not.
*/
int IsValidMove(  int c, int m, int cOnboat, int mOnboat)
{
    int cL, mL, cR, mR;
    cL = c - cOnboat;
    mL = m - mOnboat;
    cR = 3 - c + cOnboat;
    mR = 3 - m + mOnboat;

    if(((cL > 3)||(mL > 3)||(cR > 3)||(mR > 3)) ||
       ((cL < 0)||(mL < 0)||(cR < 0)||(mR < 0)))
     {
        return (0);
     }
     else
     {
        if(((cL <= mL) || (cL == 0) || (mL == 0)) &&
          ((cR <= mR) || (cR == 0) || (mR == 0)))
        {
           return (1);
        }
        else
```

```
        {
           return (0);
        }
    }
}
/*
  A function that returns node number
*/
int  assignnodenum (int c, int m)
{
    if((c == 3) && (m == 3))
    {
       return(50);
    }
    else if((c == 3) && (m == 2))
    {
       return(49);
    }
    else if((c == 3) && (m == 1))
    {
       return(47);
    }
    else if((c == 3) && (m == 0))
    {
       return(46);
    }
    else if((c == 2) && (m == 3))
    {
       return(31);
    }
    else if((c == 2) && (m == 2))
    {
       return(27);
    }
    else if((c == 2) && (m == 1))
    {
       return(24);
    }
    else if((c == 2) && (m == 0))
    {
       return(22);
    }
    else if((c == 1) && (m == 3))
    {
       return(16);
    }
    else if((c == 1) && (m == 2))
    {
       return(10);
    }
    else if((c == 1) && (m == 1))
    {
       return(7);
    }
    else if((c == 1) && (m == 0))
    {
       return(6);
    }
    else if((c == 0) && (m == 3))
    {
       return(5);
    }
```

```
    else if((c == 0) && (m == 2))
    {
       return(3);
    }
    else if((c == 0) && (m == 1))
    {
       return(1);
    }
    else if((c == 0) && (m == 0))
    {
       return(0);
    }
    else
    {
       return(19);
    }
}

/*
  This function takes the # cannibals (c) and mercenaries (m) along with the action.
  Based on the value of (c, m) and move next state ID is generated.
*/
int GenState(int *c, int *m, int action)
{
   int can, mer;
    can = *c;
    mer = *m;
    switch(action)
    {
       case 0: //CCCR
             if(IsValidMove(can, mer, 3, 0))
             {
                can = *c - 3;
                mer = *m;
                *c  = can;
                *m  = mer;
                return(assignnodenum(can, mer));
             }
             else
             {
                return (255);
             }
             break;
       case 1: //MMMR
             if(IsValidMove(can, mer, 0, 3))
             {
                can = *c;
                mer = *m - 3;
                *c = can;
                *m = mer;
                  return(assignnodenum(can, mer));
             }
             else
             {
                return (255);
             }
             break;
       case 2: //CMMR
             if(IsValidMove(can, mer, 1, 2))
             {
                can = *c - 1;
                mer = *m - 2;
```

```
         *c = can;
         *m = mer;
          return(assignnodenum(can, mer));
      }
      else
      {
         return (255);
      }
      break;
case 3: //CCMR
      if(IsValidMove(can, mer, 2, 1))
      {
         can = *c - 2;
         mer = *m - 1;
         *c = can;
         *m = mer;
          return(assignnodenum(can, mer));
      }
      else
      {
         return (255);
      }
      break;
case 4: //CMR
      if(IsValidMove(can, mer, 1, 1))
      {
         can = *c - 1;
         mer = *m - 1;
         *c = can;
         *m = mer;
          return(assignnodenum(can, mer));
      }
      else
      {
         return (255);
      }
      break;
case 5: //MMR
      if(IsValidMove(can, mer, 0, 2))
      {
         can = *c;
         mer = *m - 2;
         *c = can;
         *m = mer;
          return(assignnodenum(can, mer));
      }
      else
      {
         return (255);
      }
      break;
case 6: //CCR
      if(IsValidMove(can, mer, 2, 0))
      {
         can = *c - 2;
         mer = *m;
         *c = can;
         *m = mer;
          return(assignnodenum(can, mer));
      }
      else
      {
```

```
            return (255);
        }
        break;
    case 7: //CR
        if(IsValidMove(can, mer, 1, 0))
        {
            can = *c - 1;
            mer = *m;
            *c = can;
            *m = mer;
             return(assignnodenum(can, mer));
        }
        else
        {
            return (255);
        }
        break;
    case 8: //MR
        if(IsValidMove(can, mer, 0, 1))
        {
            can = *c;
            mer = *m - 1;
            *c = can;
            *m = mer;
             return(assignnodenum(can, mer));
        }
        else
        {
            return (255);
        }
        break;
    default:
        return (255);
        break;
  }
}

/*
 This function removes the repeated nodes in the currentPool
 to avoid the multiple execution of same node during the
 state space generation algorithm.

*/
void FilterCurrentPool(int currentPool[])
{
  int i, j, k;
  //check for the repeated one in the currentPool
  for(j = 0; currentPool[j] != -1; j++)
  {
   for(i = j+1; currentPool[i] != -1; i++)
   {
    if(currentPool[j] == currentPool[i])
    {
     for(k = j; k < NODE_LIMIT-1; k++)
     {
      currentPool[k] = currentPool[k+1];
     }
     currentPool[NODE_LIMIT-1] = -1;
    }
   }
  }
}
```

```
/*
  This function is used in state space (or graph) generation algorithm.
  This function checks if the nextPool of the graph nodes is empty or not.
  If the nextPool is empty, then the state space generation is completed.
*/
int IsEmpty(int nextPool[])
{
 int i;
 for(i = 0; i < NODE_LIMIT; i++)
 {
  if(nextPool[i] > 0)
  {
   return (0);
  }
 }
 return (1);
}
void BreadthFirstSearch( Node * adjacencyList[],int parent[],int level[])
{
    Node * traverse;
    int i, par, lev, flag = 1;
    // 'lev' represents the level to be assigned
    // 'par' represents the parent to be assigned
    // 'flag' used to indicate if graph is exhausted
    lev = 0;
    level[NODE_LIMIT-1] = lev;
    // We start at startVertex

    while (flag) {
        flag = 0;
        for (i = 0; i < NODE_LIMIT; ++i) {
            if (level[i] == lev) {
                flag = 1;
                traverse = adjacencyList[i];
                par = i;

                while (traverse != NULL) {
                    if (level[traverse->nodeNum] != -1) {
                        traverse = traverse->next;
                        continue;
                    }
                    level[traverse->nodeNum] = lev + 1;
                    parent[traverse->nodeNum] = par;
                    traverse = traverse->next;
                }
            }
        }
        ++lev;
    }
}

int main(void)
{
    int  i  = 0, j  = 0,k  = 0, v1 = 0, v2 = 0,c=3,m=3,tempC=0,tempM=0;
    int  flag = 0,currentPool[NODE_LIMIT],nextPool[NODE_LIMIT],path[6];
    int  move[NUM_OF_MOVE] = {CCCR, MMMR, CMMR, CCMR, CMR, MMR, CCR, CR, MR};
    int cValue[NODE_LIMIT], mValue[NODE_LIMIT],ath[NODE_LIMIT];
    int p, levelCount,parent[NODE_LIMIT], level[NODE_LIMIT];
    Node *adjList[NODE_LIMIT];
    Node *traverse;
    //Initialization
```

```
for(i = 0; i < NODE_LIMIT; i++){
   adjList[i]= NULL, currentPool[i] = -1;
   nextPool[i]= -1,parent[i]= 50,level[i]= -1;
}
 //set up thethe first parent Node
 v1 = assignnodenum(c, m);
 for(i = 0; i < NUM_OF_MOVE; i++){
    tempC = c;
    tempM = m;
    v2 = GenState(&tempC, &tempM, move[i]);
    if(v2 != 0b11111111)
    {
       adjList[v1] = addadjnode(adjList[v1], v2, tempC, tempM);
    }
 }
 currentPool[0] = v1;
 while(1)
 {
  k = 0;
  for(j = 0; currentPool[j] != -1; j++)
  {
   v1 = currentPool[j];
   traverse = adjList[v1];
   while(traverse != NULL)
   {
    v1 = traverse->nodeNum;
     //check for whether the node is already build.
    for(i = 0; i < NODE_LIMIT; i++)
    {
     if((v1 == currentPool[i]) || (v1 == nextPool[i]))
     {
      flag = 1;
      break;
     }
     else
     {
      flag = 0;
     }
    }
    if(flag == 1)
    {
     traverse = traverse->next;
     continue;
    }

    nextPool[k++] = v1;

    c = traverse->cNum;
    m = traverse->mNum;
    for(i = 0; i < NUM_OF_MOVE; i++)
     {
       tempC = c;
       tempM = m;
       v2 = GenState(&tempC, &tempM, move[i]);
       if(v2 != 0b11111111)
       {
          adjList[v1] = addadjnode(adjList[v1], v2, tempC, tempM);
       }
    }
    traverse = traverse->next;
   } //while(traverse)
  } //for(j)
```

```
      //is nextPool empty
      if(IsEmpty(nextPool))
      {
       break;
      }
      //copy nextPool into currentPool
      for(i = 0; i < NODE_LIMIT; i++ )
      {
       if(i<k)
       {
        currentPool[i] = nextPool[i];
        nextPool[i]    = -1;
       }
       else
       {
        currentPool[i] = -1;
        nextPool[i]    = -1;
       }
      }
      FilterCurrentPool(currentPool);

     } //while

     flag = 1;
     //Printing adjancency list
     for(i = 50; i >= 0; i--){
         traverse = adjList[i];
         if(traverse != NULL)
         {
             cout<<"Node"<<i<<"--->";
         }
         flag = 0;
         while(traverse != NULL){
             cout<<"Node"<<"["<<traverse->nodeNum<<"]"<<"c="<<traverse->cNum<<","<<"m="<< traverse->mNum<<" ---
> ";
             traverse = traverse->next;
             flag = 1;
         }
         if(flag == 1)
         {
             cout<<"NULL\n\n";
         }
     }
    BreadthFirstSearch(adjList, parent, level);
    levelCount = level[0];
    path[0] = 0;
    for(i = 1; i <= levelCount; i++)
    {
       path[i] = parent[path[i-1]];
    }
    int cn[5],me[5];
    cout<<"\nThe States Forming Optimal Path in State Space Diagram  -\n";
    for (i = 0; i <= levelCount; i++) {
         switch(path[i])
         {
         case 0:
                 cn[i]=0;
                 me[i]=0;

                 break;
         case 5:
                cn[i]=0;
```

```
            me[i]=3;
            break;
      case 6:
            cn[i]=1;
            me[i]=0;
            break;
      case 50:
            cn[i]=3;
            me[i]=3;
            break;
      }
     cout<<"stateID= "<<path[i]<<" and (c,m)= "<<cn[i]<<","<<me[i]<<"\t";

  }
    cout<<"\n\n";
    cout<<"\n~~~~~~~~~The optimal solution for missionary and cannibals problem is~~~~~~~~~\n";
    cout<<"\nState(c="<<cn[levelCount]<<","<<" m="<<me[levelCount]<<")";
    for (int k = levelCount; k > 0; k--) {
     c = abs(cn[k] - cn[k-1]);
     m = abs(me[k] - me[k-1]);
     cout<<"-->Action(c="<<c<<","<<"m="<<m<<")"<<"-->State(c="<<cn[k-1]<<","<<"m="<<me[k-1]<<")";
  }
  return 0;
}
```

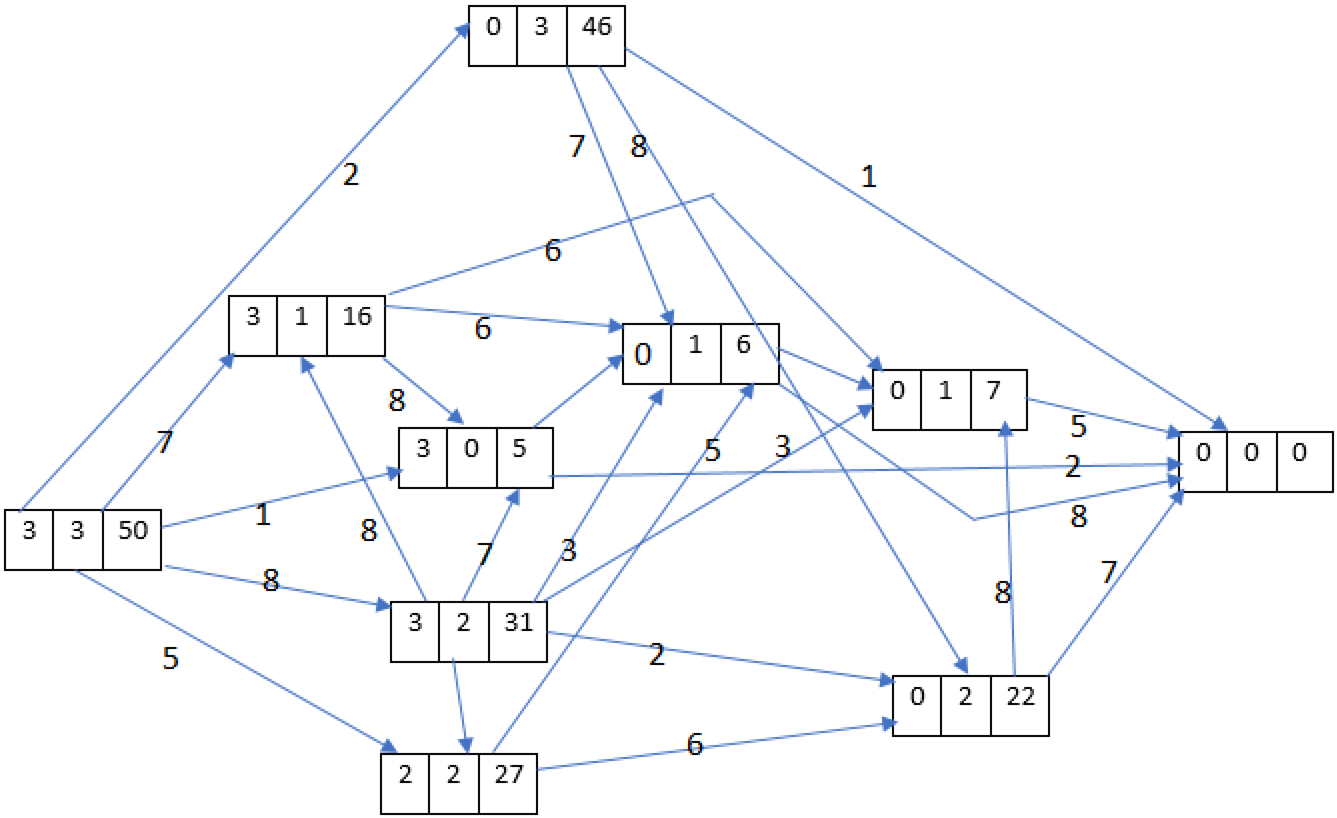

Fig: State Space Diagram for Machineries and Cannibals problem with Boat Capacity 3

1=CCCR, 2=MMMR, 3=CMMR, 4=CCMR, 5=CMR, 6=MMR, 7=CCR, 8=CR, 9=MR

M	C	ID

Where,

M=Number of Machineries

C= Number of Cannibals

ID=Space ID

R= Boat action toward right side of river navin@navin-Aspire-F5-573G: ~/Desktop/CS 776/Assignment 1

```
navin@navin-Aspire-F5-573G:~/Desktop/CS 776/Assignment 1$ g++ 1.cpp
navin@navin-Aspire-F5-573G:~/Desktop/CS 776/Assignment 1$ ./a.out
Node50--->Node[33]c=2,m=2 ---> Node[36]c=2,m=3 ---> Node[30]c=1,m=3 ---> NULL

Node36--->Node[33]c=2,m=2 ---> Node[30]c=1,m=3 ---> Node[7]c=0,m=3 ---> NULL

Node33--->Node[10]c=1,m=1 ---> Node[25]c=2,m=0 ---> NULL

Node30--->Node[7]c=0,m=3 ---> Node[10]c=1,m=1 ---> Node[9]c=-1,m=3 ---> NULL

Node25--->Node[5]c=1,m=0 ---> Node[0]c=0,m=0 ---> NULL

Node10--->Node[0]c=0,m=0 ---> Node[5]c=1,m=0 ---> NULL

Node9--->Node[9]c=-2,m=-2 ---> Node[9]c=-2,m=3 ---> Node[9]c=-3,m=3 ---> NULL

Node7--->Node[9]c=-1,m=3 ---> Node[9]c=-2,m=3 ---> NULL

Node5--->Node[9]c=0,m=-1 ---> Node[0]c=0,m=0 ---> NULL

Node0--->Node[9]c=-1,m=-1 ---> Node[9]c=0,m=-1 ---> Node[9]c=0,m=-2 ---> NULL


The States Forming Optimal Path in State Space Diagram  -
stateID= 0 and (c,m)= 0,0       stateID= 10 and (c,m)= 1,1      stateID= 30 and (c,m)= -599522600,-926717912    stateID= 50 and (c,m)= 3,3


~~~~~~~~~The optimal solution for missionary and cannibals problem is~~~~~~~~~

State(c=3, m=3)-->Action(c=599522603,m=926717915)-->State(c=-599522600,m=-926717912)-->Action(c=599522601,m=926717913)-->State(c=1,m=1)-->Action(c=1,m=1)-->State(c=0,m=0)navin@navin-Aspire-F5-573G:~/Desktop/CS 776/Assignment 1$
```

1:12 PM