Artificial Intelligence Coursework 1

Search Methods

Student Name: Zhao Qi

Student Id:30571626

Email:qz1y18@soton.ac.uk

Content

1. APPROACH

I choose C++ to complete this assighment. Firstly, I build a class to act as a node. It has many elements. Such as the statement of the board, the depth of the node, the last step of the node and the location of the node etc.

The Breadth-First Search is written in the Function **breadthFirst**().The input of it is a root node, and the output is the goal node. In the function, I use a queue to store the node which will be checked. Every turn I will check whether the node on queue front is the goal node. If it is, return it. If not, I will create its child nodes and push them into the queue rear, and then pop the node which was checked. Repeat this process until the goal node is found.

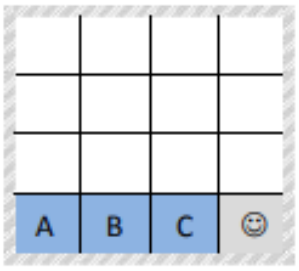
The Depth First Search is written in the Function **depthFirst()**.The input of it is a root node, and the output is the goal node. I use recursive the function to complete it. Firstly, I check the whether the input node is the goal node. If it is, return it. If not, I will generate its child nodes and use the function **depthFirst()** to search its child nodes. I record its last history step to avoid the direction of this step as opposed to the previous step. I found this function does not work because it will fall into an infinite loop. So I write the Depth limited search in **depthlimited()**.In this function I limit the node's depth. Then the function can successfully find a solution to get the goal state.

The Iterative Deepening Search is written in the Function **iterativeDeepening()**. Most of the process of it is as same as **depthlimited**(), but this is a loop enfolding **depthlimited()**, and the depth limitation will add one in every circle until find the goal node.

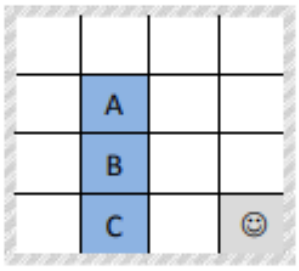
The A* heuristic search is written in the Function **Astar()**. I define the cost function as the node's depth + Manhattan distance. And I write the cost function in **costValue().**In Function **Astar()**, I use a queue to store the all of the node generated. And in this section, I add a parameter valid in the node class. Valid is 1 means this node has not child node (its color is 'white'). Valid is 0 means the node has child node (its color is 'black'). And when compare the lowest cost value, there is no need for it to compare. At the beginning of the function, I find the node in queue who has the lowest cost value and its valid is 1, then generate its child nodes and push these nodes into the queue rear. Then I set the lowest cost node's valid to 0. Repeat this process until the goal node is found.

2. EVIDENCE

In this section, I use these four methods to solve the problem on coursework assignment.



Start state.



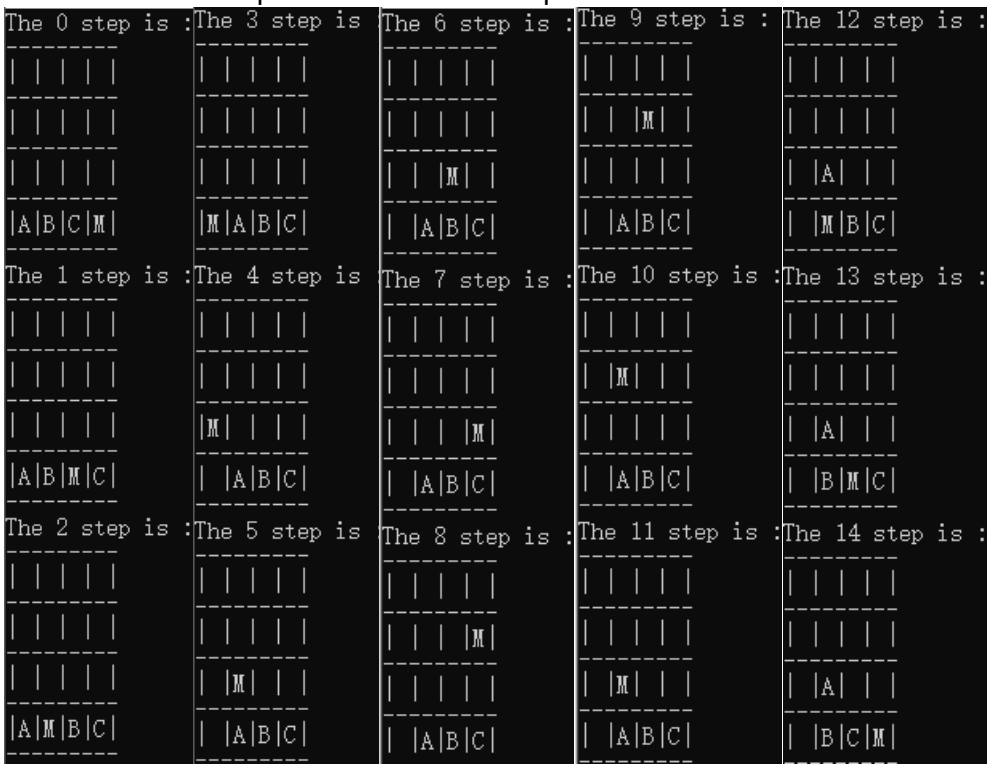
Goal state.

2.1 Breadth-First Search

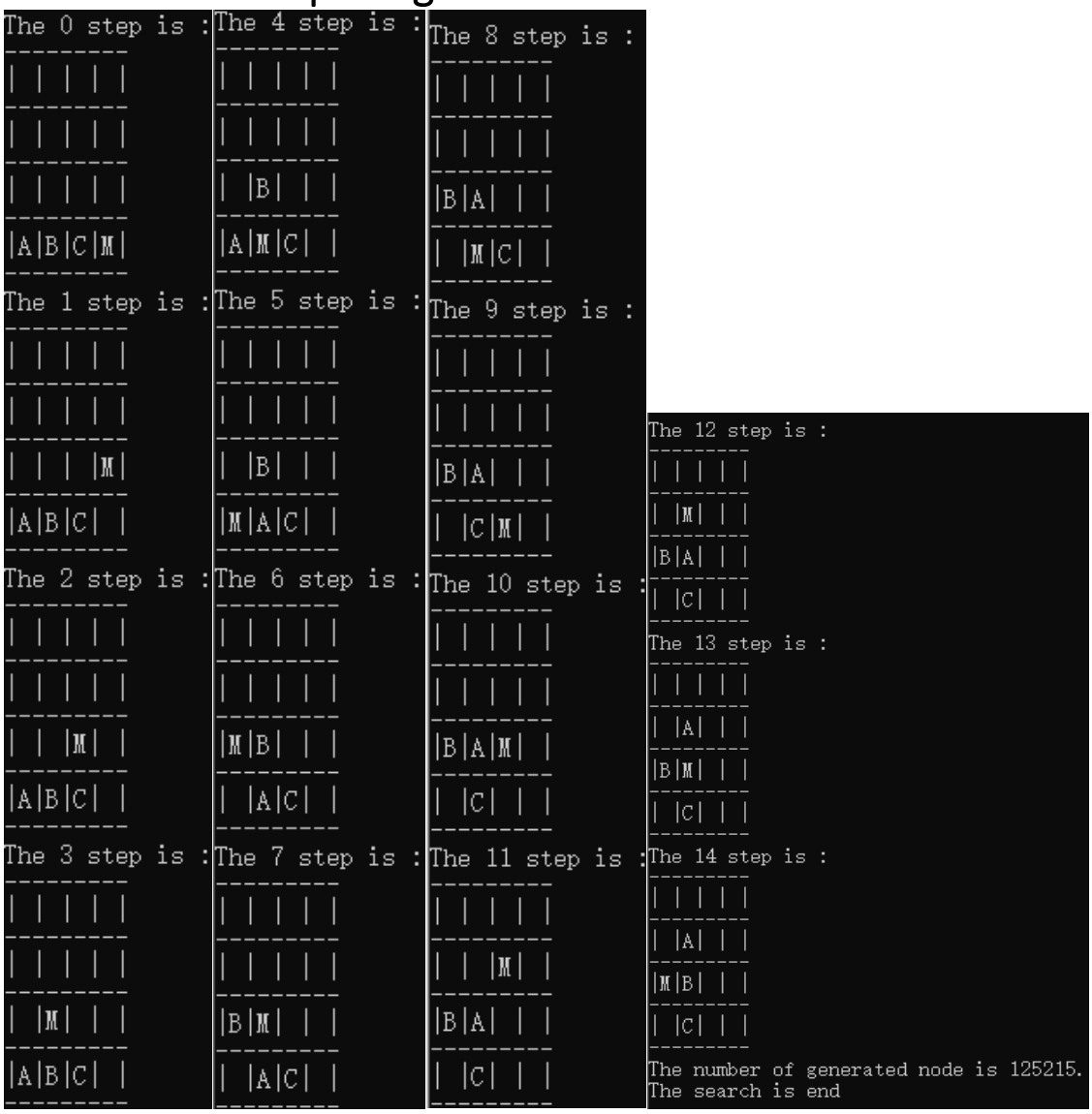
```
The 0 step is :
---------
| | | | |
---------
| | | | |
---------
| | | | |
---------
|A|B|C|M|
---------
The 1 step is :
---------
| | | | |
---------
| | | | |
---------
| | | |M|
---------
|A|B|C| |
---------
The 2 step is :
---------
| | | | |
---------
| | | | |
---------
| | |M| |
---------
|A|B|C| |
---------
The 3 step is :
---------
| | | | |
---------
| | | | |
---------
| |M| | |
---------
|A|B|C| |
---------
The 4 step is :
---------
| | | | |
---------
| | | | |
---------
| |B| | |
---------
|A|M|C| |
---------
The 5 step is :
---------
| | | | |
---------
| | | | |
---------
| |B| | |
---------
|M|A|C| |
---------
The 6 step is :
---------
| | | | |
---------
| | | | |
---------
|M|B| | |
---------
| |A|C| |
---------
The 7 step is :
---------
| | | | |
---------
| | | | |
---------
|B|M| | |
---------
| |A|C| |
---------
The 8 step is :
---------
| | | | |
---------
| | | | |
---------
|B|A| | |
---------
| |M|C| |
---------
The 9 step is :
---------
| | | | |
---------
| | | | |
---------
|B|A| | |
---------
| |C|M| |
---------
The 10 step is :
---------
| | | | |
---------
| | | | |
---------
|B|A|M| |
---------
| |C| | |
---------
The 11 step is :
---------
| | | | |
---------
| | | | |
---------
| | |M| |
---------
|B|A| | |
---------
| |C| | |
---------
The 12 step is :
---------
| | | | |
---------
| |M| | |
---------
|B|A| | |
---------
| |C| | |
---------
The 13 step is :
---------
| | | | |
---------
| |A| | |
---------
|B|M| | |
---------
| |C| | |
---------
The 14 step is :
---------
| | | | |
---------
| |A| | |
---------
|M|B| | |
---------
| |C| | |
---------
The number of generated node is 141699.
The search is end
```

2.2 Deep First Search & Deep Limited Search

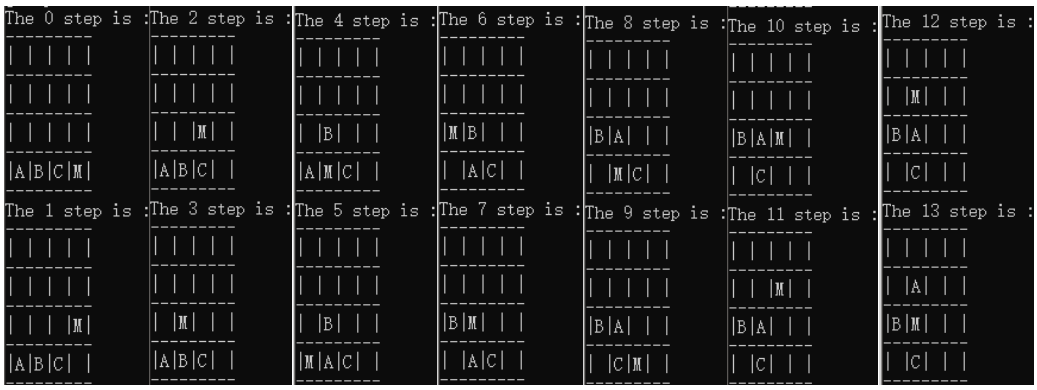
Because I found deep first always fall into an infinite loop. So I write the Depth limited search to solve the problem. I set the deep limitation as 20.

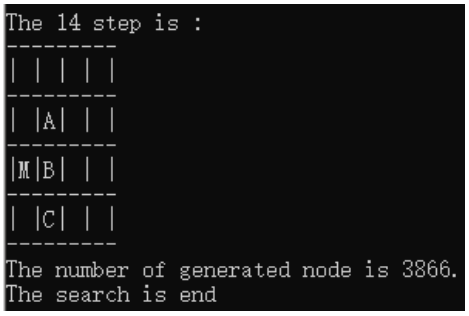


2.3 Iterative Deepening Search



2.4 A* heuristic search





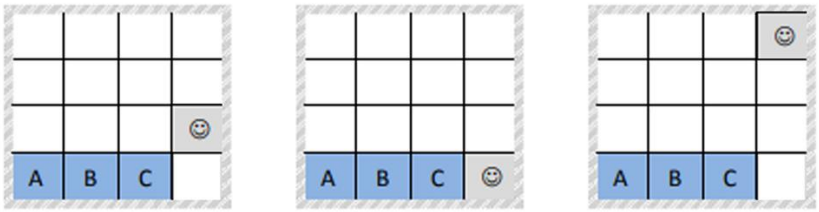
Through the result shown below, we can gain a queue about the number of generated nodes to solve a same problem:
A* heuristic < Deep Limited < Iterative Deepening < Breadth-First .

3. SCALABILITY STUDY

In this section, I change the difficulty of the game. I control the difficult in terms 3 aspects: the start position of man, the number of blocks and the number of grids.

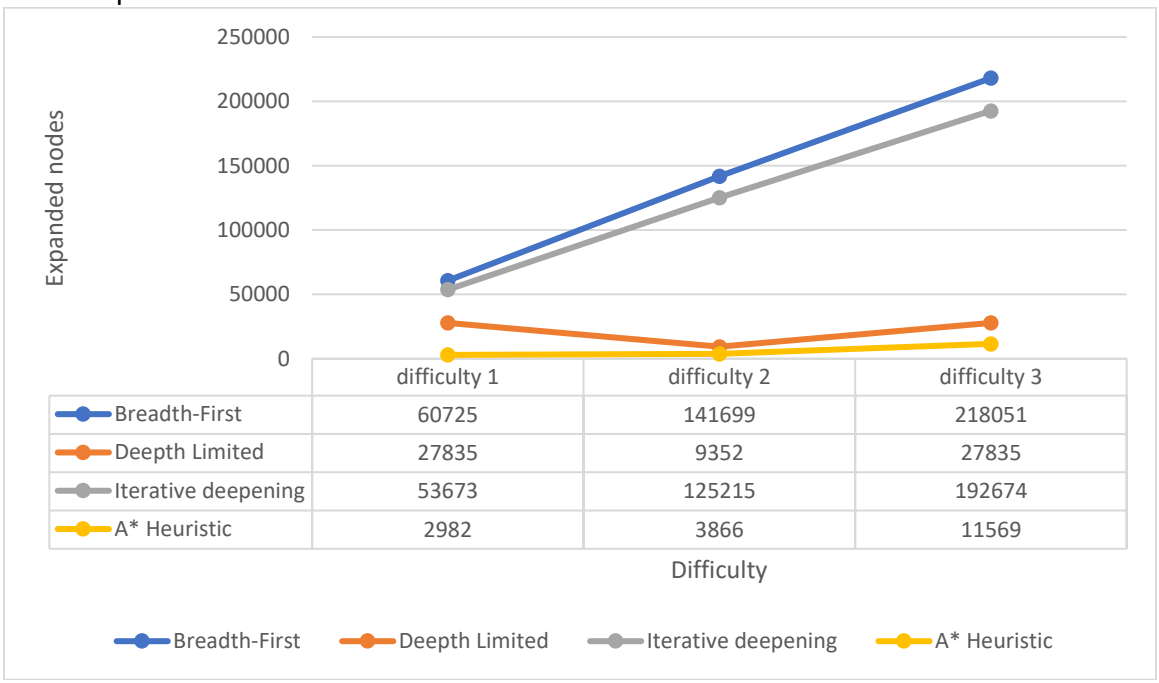
3.1 Change the start position of man

I set the difficulty of the problem depend on the distance between man and the center position of blocks. The examples are shown below:



diffculty 1 diffculty 2 diffculty 3

The compare result is shown below:

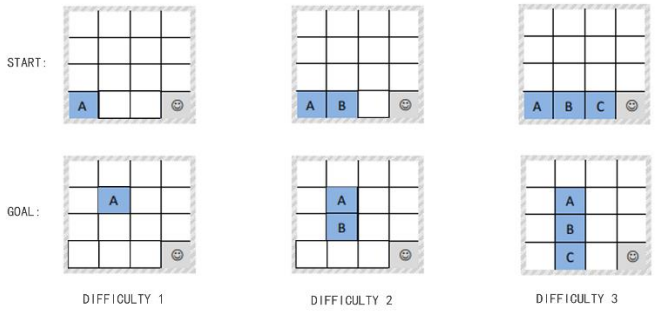


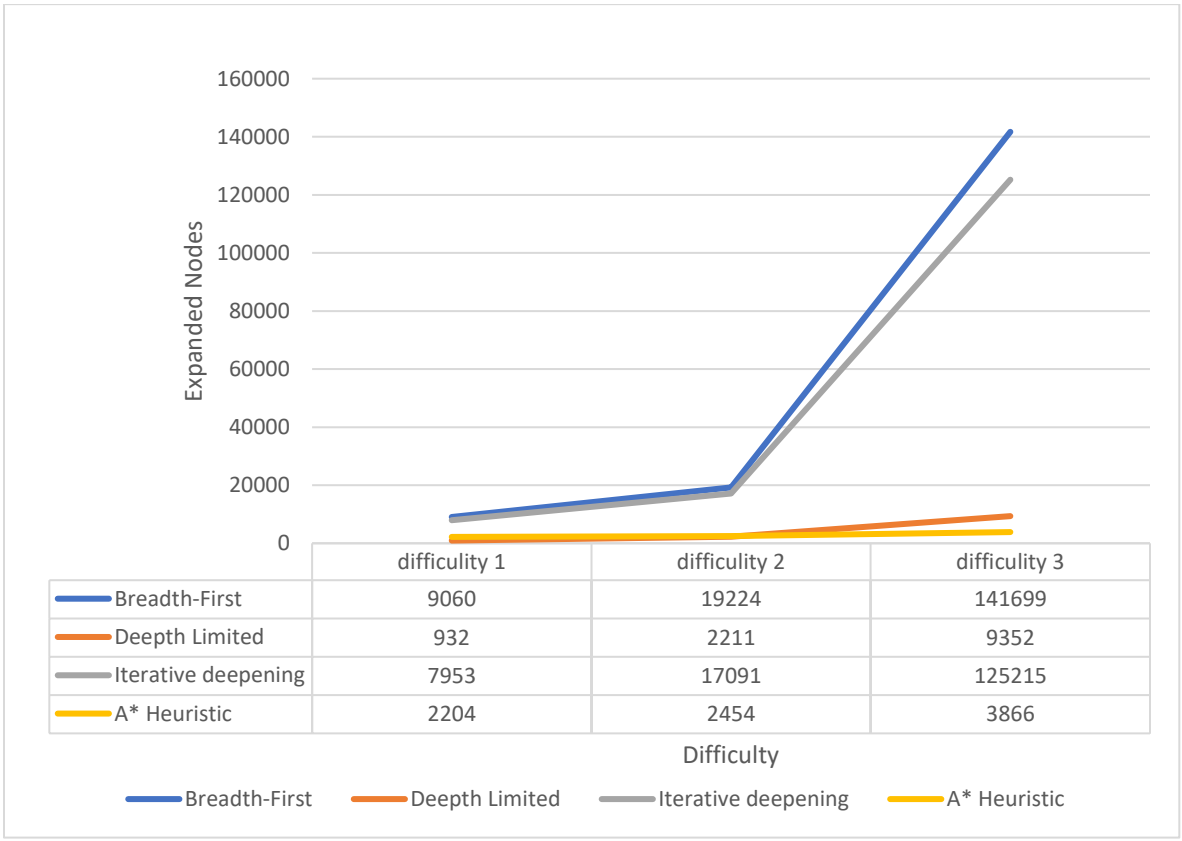
	difficulty 1	difficulty 2	difficulty 3
Breadth-First	60725	141699	218051
Deepth Limited	27835	9352	27835
Iterative deepening	53673	125215	192674
A* Heuristic	2982	3866	11569

Through the picture above we can see that with the distance between the center position of blocks (B's position) and Man's position, the expanded nodes mostly increase. With the distance increase, Breadth-First and Iterative deepening rise greatly, while A* Heuristic increase slowly. And what is strange is that Depth Limited decreases firstly and increases then. I think the reason is that comparing difficulty2, the difficulty 1 and difficulty 3's goal state node are on a more to the left tree. In addition, the number of expanded nodes of A* Heuristic and Depth Limited are much lower than that of Breadth First and Iterative deepening.

3.2 Change the number of blocks

In this section, I set difficulty depend on the number of blocks, the examples are shown below:



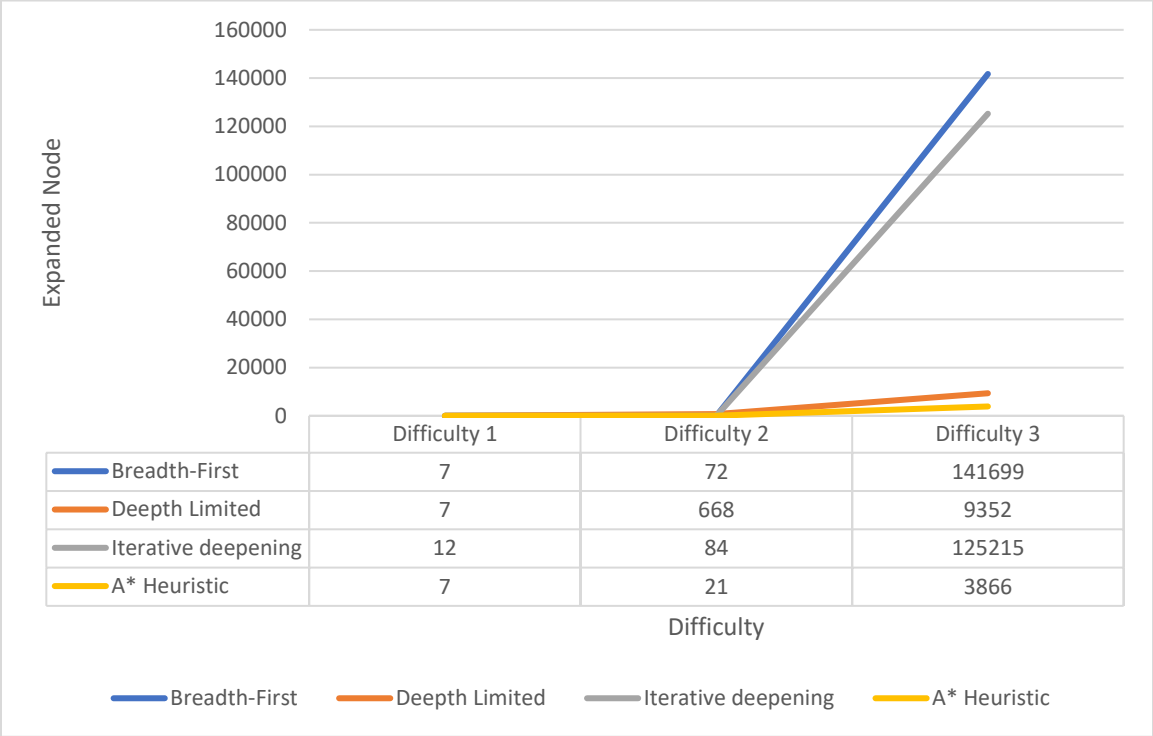


	difficulty 1	difficulty 2	difficulty 3
Breadth-First	9060	19224	141699
Deepth Limited	932	2211	9352
Iterative deepening	7953	17091	125215
A* Heuristic	2204	2454	3866

Through the picture above we can see with difficulty increasing, for these four methods, the number of expanded nodes rise. And Breadth-First and Iterative deepening rise exponentially, while Depth Limited and A* Heuristic increase stably.

3.3 Change the number of grids

In this section, I set difficulty depend on the number of grids, the examples are shown below:



	Difficulty 1	Difficulty 2	Difficulty 3
Breadth-First	7	72	141699
Deepth Limited	7	668	9352
Iterative deepening	12	84	125215
A* Heuristic	7	21	3866

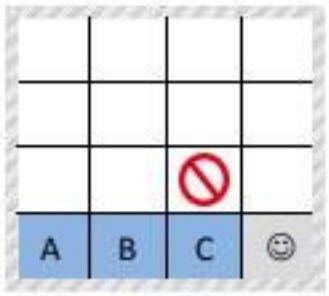
The result showed in the picture above is similar with the result in last section. Breadth-First and Iterative deepening rise exponentially, while Depth Limited and A* Heuristic increase stably.

4. EXTRAS AND LIMITATION

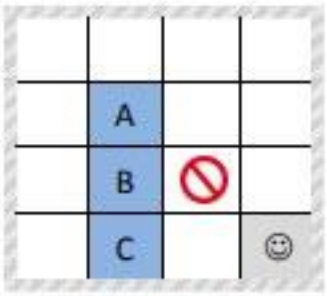
4.1 Extras

4.1.1

I put a obstacle block into the board. And I compare the difference of expanded nodes before and after the change in these 4 methods.

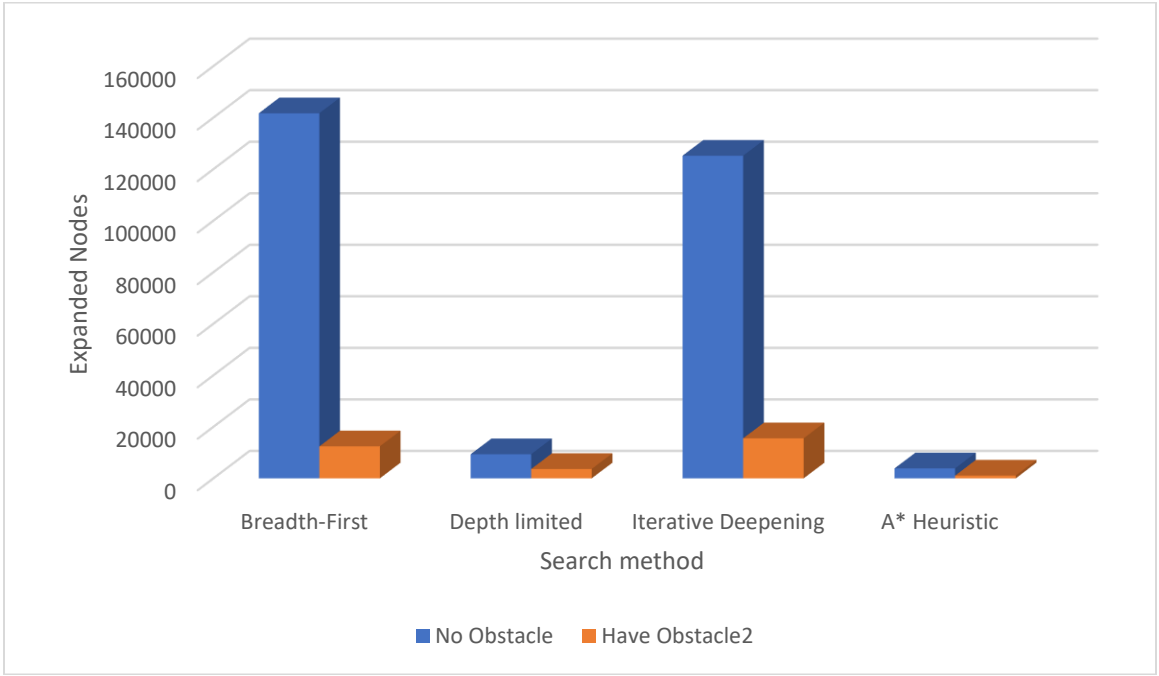


Start state.



Goal state.

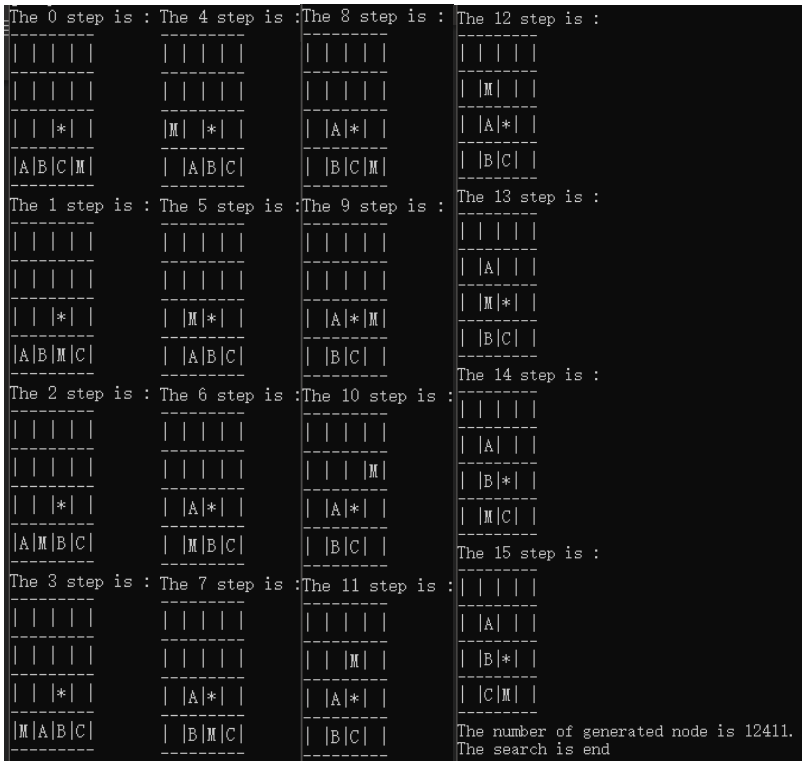
4.1.2 Result



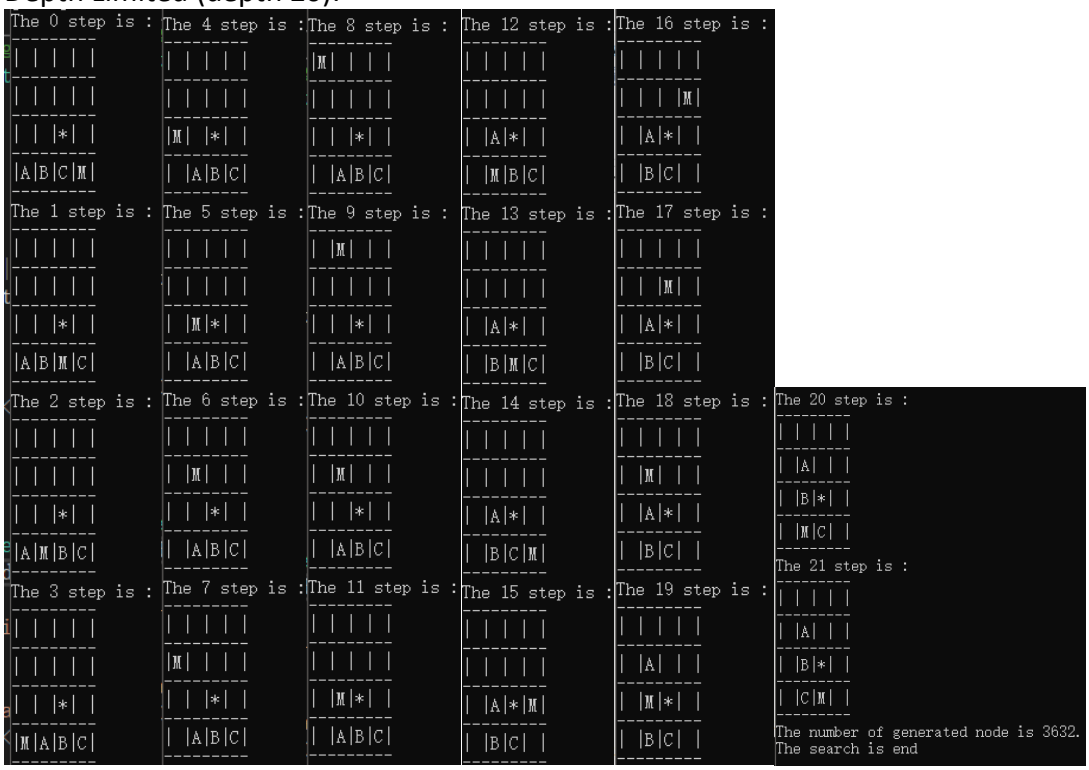
The result shows that for these four search methods, the exist of obstacle makes the number of expanded nodes decrease. I think the reason is that because obstacle pose some limitation to actions, the exist of obstacle makes the branch of the step tree decrease, So the number of expanded nodes to find a solution decreases.

4.1.3 Evidence

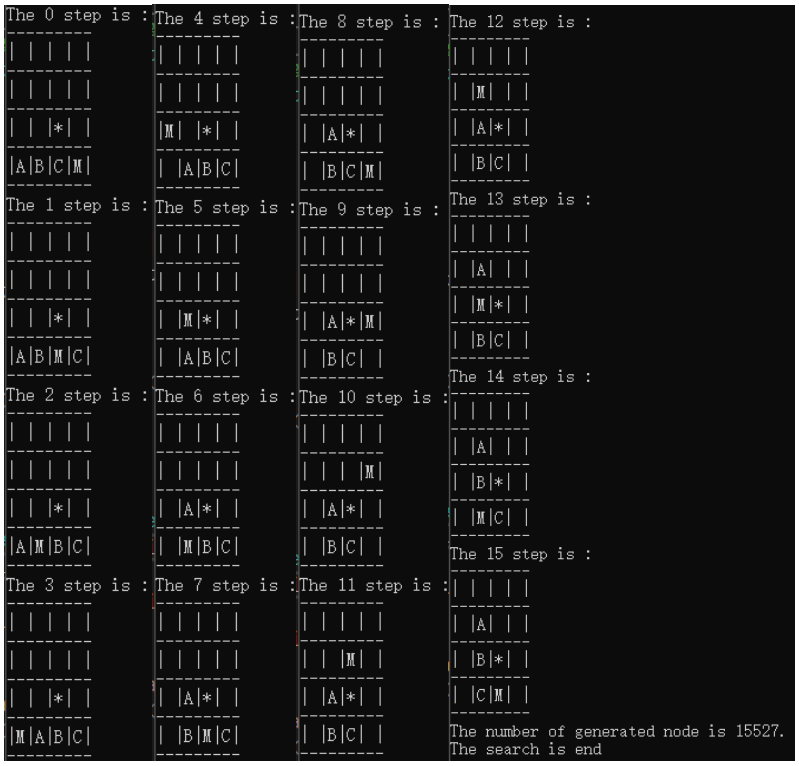
Breadth-First :



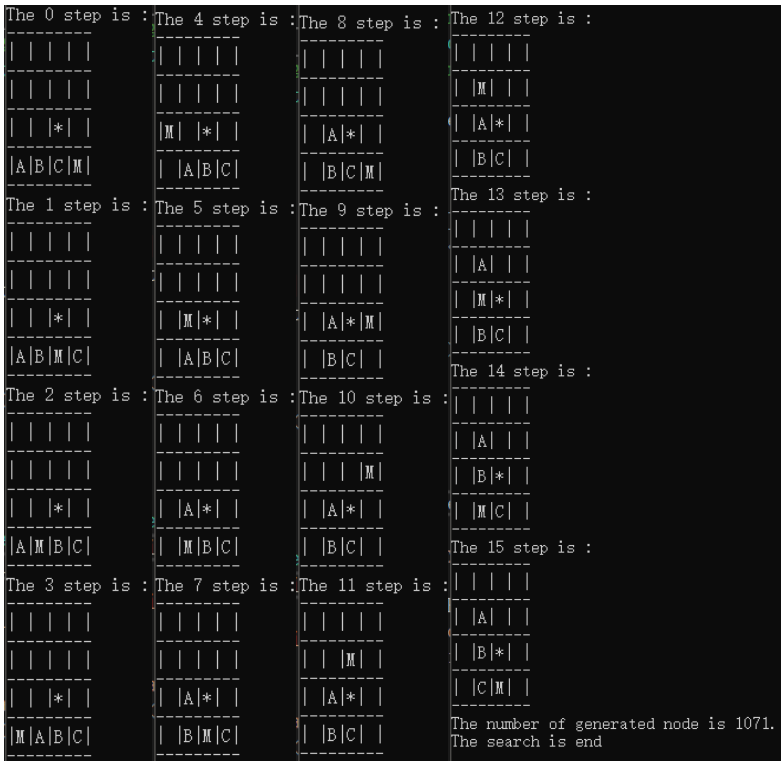
Depth Limited (depth 20):



Iterative Deepening:


The 0 step is :
		*	
A	B	C	M
The 1 step is :
		*	
A	B	M	C
The 2 step is :
		*	
A	M	B	C
The 3 step is :
		*	
M	A	B	C
The 4 step is :
M		*	
	A	B	C
The 5 step is :
	M	*	
	A	B	C
The 6 step is :
	A	*	
	M	B	C
The 7 step is :
	A	*	
	B	M	C
The 8 step is :
	A	*	
	B	C	M
The 9 step is :
	A	*	M
	B	C	
The 10 step is :
			M
	A	*	
	B	C	
The 11 step is :
		M	
	A	*	
	B	C	
The 12 step is :
	M		
	A	*	
	B	C	
The 13 step is :
	A		
	M	*	
	B	C	
The 14 step is :
	A		
	B	*	
	M	C	
The 15 step is :
	A		
	B	*	
	C	M	
The number of generated node is 15527.
The search is end

A* Heuristic:


The 0 step is :
		*	
A	B	C	M
The 1 step is :
		*	
A	B	M	C
The 2 step is :
		*	
A	M	B	C
The 3 step is :
		*	
M	A	B	C
The 4 step is :
M		*	
	A	B	C
The 5 step is :
	M	*	
	A	B	C
The 6 step is :
	A	*	
	M	B	C
The 7 step is :
	A	*	
	B	M	C
The 8 step is :
	A	*	
	B	C	M
The 9 step is :
	A	*	M
	B	C	
The 10 step is :
			M
	A	*	
	B	C	
The 11 step is :
		M	
	A	*	
	B	C	
The 12 step is :
	M		
	A	*	
	B	C	
The 13 step is :
	A		
	M	*	
	B	C	
The 14 step is :
	A		
	B	*	
	M	C	
The 15 step is :
	A		
	B	*	
	C	M	
The number of generated node is 1071.
The search is end

4.2 Limitation & Self Evaluation

One the limitation of this coursework I think is my computer's CPU speed and memory size. I planned to achieve the algorithm in 5*5 size block or higher size, but because of the limitation of my computer, I give up it.

The second limitation I think is that this problem can be solved by deep first method. Because when I run the deep first method that I wrote, the algorithm fall into an infinite circle. When I checked the step history I recorded, I found that its path is a circle. So I think may be put some 'obstacles' on its circle path can solve the problem.

About the drawbacks of my coursework, one of it is that I should use more useful data structure into my code, I think it can help me save times and computer memory.

And for A* method, my heuristic is Manhattan distance. I think if it can be changed by a better function, the A* method can be improved.

5. Code:

```
#include "pch.h"
#include <iostream>
#include<cmath>
using namespace std;
#define N 4 //the size of a board
#define goalstate TNode->state[1][1] == 1 && TNode->state[2][1] == 2 && TNode->state[3][1] ==
3
#define okleft (TNode->location[1] - 1 >= 0 && TNode->his != 3 && !(TNode->location[1] - 1 ==
obstacleLocation[1]&&TNode->location[0] == obstacleLocation[0]))
#define okright (TNode->location[1] + 1 < N && TNode->his != 4 && !(TNode->location[1] + 1 ==
obstacleLocation[1]&&TNode->location[0] == obstacleLocation[0]))
#define okup (TNode->location[0] - 1 >= 0 && TNode->his != 1 && !(TNode->location[0] - 1 ==
obstacleLocation[0]&&TNode->location[1] == obstacleLocation[1]))
#define okdown (TNode->location[0] + 1 < N && TNode->his != 2 && !(TNode->location[0] + 1 ==
obstacleLocation[0]&&TNode->location[1] == obstacleLocation[1]))
int NodeNumber = 0;//record how many nodes are generated during a search
int obstacleLocation[2] = { 2,2 };//define the location of obstacles
//Node Class
class pathnode {
public:
          int depth = 0;//The depth of the node
          int state[N][N] = { {0,0,0,0},{0,0,0,0},{0,0,5,0},{1,2,3,4} };//the board
statement,1 means A,2 means B,3 means C,4 means man,5 means obstacle.
          int location[2] = { 3,3 };//the location of man
          int his = 0;//Record the history step
          int vaild = 1;//useful in Astar reasearch,to make node be 'dark'
          int hvalue = 1000;//useful in Astar research
          pathnode* father = NULL;//father pointer
          pathnode* left = NULL;//left child pointer
          pathnode* right = NULL;
          pathnode* up = NULL;
          pathnode* down = NULL;
          pathnode* stepnext = NULL;//useful in printing the complete route
          pathnode* quenext = NULL;//useful in breadfirst research and Astar research
};
//Copy the board statement
void copystate(pathnode* A, pathnode* B) {
          for (int i = 0; i < N; i++) {
                    for (int j = 0; j < N; j++) {
                              B->state[i][j] = A->state[i][j];
                    }
          }
}
//Generate Child Node.
//For the parameter operate, 1 means down, 2 means up, 3 means right, 4 means left.
void updateChildNode(pathnode* TNode, int operate) {
          //if the operate is go down
          if (operate == 1) {
                    //generate new node
                    TNode->down = new pathnode();
                    NodeNumber += 1;
                    TNode->down->father = TNode;
```

```
                    //Assign the value to a new node
                    //Record the previous step
                    TNode->down->his = 1;
                    //Record the depth of the node
                    TNode->down->depth = TNode->depth + 1;
                    cout << "go down and " << TNode->depth + 1 << endl;
                    //Change the location of man
                    TNode->down->location[0] = TNode->location[0] + 1;
                    TNode->down->location[1] = TNode->location[1];
                    //Change the board statement
                    copystate(TNode, TNode->down);
                    int Tem = TNode->down->state[TNode->location[0]][TNode->location[1]];
                    TNode->down->state[TNode->location[0]][TNode->location[1]] =
TNode->down->state[TNode->location[0] + 1][TNode->location[1]];
                    TNode->down->state[TNode->location[0] + 1][TNode->location[1]] = Tem;
          }
          //if the operate is go up
          if (operate == 2) {
                    TNode->up = new pathnode();
                    NodeNumber += 1;
                    TNode->up->father = TNode;
                    TNode->up->depth = TNode->depth + 1;
                    cout << "go up and " << TNode->depth + 1 << endl;
                    TNode->up->his = 2;
                    TNode->up->location[0] = TNode->location[0] - 1;
                    TNode->up->location[1] = TNode->location[1];
                    copystate(TNode, TNode->up);
                    int Tem = TNode->state[TNode->location[0]][TNode->location[1]];
                    TNode->up->state[TNode->location[0]][TNode->location[1]] =
TNode->up->state[TNode->location[0] - 1][TNode->location[1]];
                    TNode->up->state[TNode->location[0] - 1][TNode->location[1]] = Tem;
          }
          //if the operate is go right
          if (operate == 3) {
                    TNode->right = new pathnode();
                    NodeNumber += 1;
                    TNode->right->father = TNode;
                    TNode->right->his = 3;
                    TNode->right->depth = TNode->depth + 1;
                    cout << "go right and  " << TNode->depth + 1 << endl;
                    TNode->right->location[1] = TNode->location[1] + 1;
                    TNode->right->location[0] = TNode->location[0];
                    copystate(TNode, TNode->right);
                    int Tem = TNode->right->state[TNode->location[0]][TNode->location[1]];
                    TNode->right->state[TNode->location[0]][TNode->location[1]] =
TNode->right->state[TNode->location[0]][TNode->location[1] + 1];
                    TNode->right->state[TNode->location[0]][TNode->location[1] + 1] = Tem;
          }
          //if the operate is go left
          if (operate == 4) {
                    TNode->left = new pathnode();
                    NodeNumber += 1;
                    TNode->left->father = TNode;
                    TNode->left->his = 4;
                    TNode->left->depth = TNode->depth + 1;
```

```
                    cout << "go left and " << TNode->depth + 1 << endl;
                    copystate(TNode, TNode->left);
                    TNode->left->location[1] = TNode->location[1] - 1;
                    TNode->left->location[0] = TNode->location[0];
                    int Tem = TNode->left->state[TNode->location[0]][TNode->location[1]];
                    TNode->left->state[TNode->location[0]][TNode->location[1]] =
TNode->left->state[TNode->location[0]][TNode->location[1] - 1];
                    TNode->left->state[TNode->location[0]][TNode->location[1] - 1] = Tem;
          }
}
//Compute the Cost of path: node depth+Manhattan distance
int costValue(pathnode* TNode) {
          int AlocationX,AlocationY,BlocationX,BlocationY,ClocationX,ClocationY,score = 0;
          for (int i = 0; i < N; i++) {
                    for (int j = 0; j < N; j++) {
                              //Find A's position
                              if (TNode->state[i][j] == 1) {
                                        AlocationX = i;
                                        AlocationY = j;
                              }
                              //Find B's positon
                              if (TNode->state[i][j] == 2) {
                                        BlocationX = i;
                                        BlocationY = j;
                              }
                              //Find C's position
                              if (TNode->state[i][j] == 3) {
                                        ClocationX = i;
                                        ClocationY = j;
                              }
                    }
          }
          //compute the cost function
          score = abs(AlocationX - 1) + abs(AlocationY - 1) + abs(BlocationX - 2) +
abs(BlocationY - 1) + abs(ClocationX - 3) + abs(ClocationY - 1)+TNode->depth;
          return score;
}

//Deep First
pathnode* deepfirst(pathnode* TNode) {
          if (goalstate)
                    return TNode;
          else {
                    //Determine if i can go left
                    if (okleft) {
                              //generate child node of going left
                              updateChildNode(TNode, 4);
                              //determine whether there is a resolution
                              pathnode* L=deepfirst(TNode->left);
                              if (L != NULL) {
                                        return L;
                              }
                    }
                    //generate the right step child node
                    if (okright) {
```

```
                                updateChildNode(TNode, 3);
                                pathnode* R = deepfirst(TNode->right);
                                if (R != NULL) {
                                        return R;
                                }
                        }
                        //generate the up step child node
                        if (okup) {
                                updateChildNode(TNode, 2);
                                pathnode* U=deepfirst(TNode->up);
                                if (U != NULL) {
                                        return U;
                                }
                        }
                        //generate the down step child node
                        if (okdown) {
                                updateChildNode(TNode, 1);
                                pathnode* D=deepfirst(TNode->down);
                                if (D != NULL) {
                                        return D;
                                }
                        }
                        //release node
                        delete TNode;
                        return NULL;
                }
}
//Breadth First
pathnode* breadthFirst(pathnode* TNode) {
                pathnode* start = TNode;
                pathnode* end = TNode;
                if (TNode == NULL)
                        return NULL;
                if (goalstate)
                        return TNode;
                while(true) {
                        //determine whether the node is the aim node
                        if (goalstate) {
                                break;
                        }
                        //generate the left step child node
                        if (okleft) {
                                updateChildNode(TNode, 4);
                                //put the node into a queue
                                end->quenext = TNode->left;
                                end = TNode->left;

                        }
                        //generate the right step child node
                        if (okright) {
                                updateChildNode(TNode, 3);
                                end->quenext = TNode->right;
                                end = TNode->right;

                        }
```

```
                    //generate the up step child node
                    if (okup) {
                              updateChildNode(TNode, 2);
                              end->quenext = TNode->up;
                              end = TNode->up;

                    }
                    //generate the down step child node
                    if (okdown) {
                              updateChildNode(TNode, 1);
                              end->quenext = TNode->down;
                              end = TNode->down;
                    }
                    //Chang the queue head pointer
                    TNode = TNode->quenext;
                    start = TNode;
          }
          return TNode;
}
pathnode* Astar(pathnode* TNode) {
          int min = 1000;
          pathnode* start = TNode;
          pathnode* head = TNode;
          pathnode* end = TNode;
          //point the node which has the lowest cost value
          pathnode* minNode = NULL;
          //point the node which is the aim node
          pathnode* resultnode = NULL;
          if (goalstate)
                    return TNode;
          while (true) {
                    if (goalstate) {
                              break;
                    }
                    min = 1000;
                    start = head;
                    //make the vaild bit of the node to zero (Make the node be 'Dark')
                    TNode->vaild = 0;
                    //generate the left step child node
                    if (okleft) {
                              updateChildNode(TNode, 4);
                              //compute the cost value of the node
                              TNode->left->hvalue = costValue(TNode->left);
                              //put the node into a queue
                              end->quenext = TNode->left;
                              end = TNode->left;

                    }
                    //generate the right step child node
                    if (okright) {
                              updateChildNode(TNode, 3);
                              TNode->right->hvalue = costValue(TNode->right);
                              end->quenext = TNode->right;
                              end = TNode->right;
```

```
                    }
                    //generate the up step child node
                    if (okup) {
                              updateChildNode(TNode, 2);
                              TNode->up->hvalue = costValue(TNode->up);
                              end->quenext = TNode->up;
                              end = TNode->up;

                    }
                    //generate the down step child node
                    if (okdown) {
                              updateChildNode(TNode, 1);
                              TNode->down->hvalue=costValue(TNode->down);
                              end->quenext = TNode->down;
                              end = TNode->down;
                    }
                    //find the node which has the lowest cost value
                    while (start != end) {
                              if (start->vaild == 1) {
                                        if (min > start->hvalue) {
                                                  min = start->hvalue;
                                                  minNode = start;
                                        }
                              }
                              start = start->quenext;
                    }
                    TNode = minNode;
          }
          return TNode;
}
pathnode* deeplimited(pathnode* TNode, int k) {
          if (goalstate)
                    return TNode;
          else {
                    if (TNode->depth > k)
                              return NULL;
                    if (okleft) {
                              updateChildNode(TNode, 4);
                              pathnode* L = deeplimited(TNode->left, k);
                              if (L != NULL) {
                                        return L;
                              }
                    }
                    //generate the right step child node
                    if (okright) {
                              updateChildNode(TNode, 3);
                              pathnode* R = deeplimited(TNode->right, k);
                              if (R != NULL) {
                                        return R;
                              }
                    }
                    //generate the up step child node
                    if (okup) {
                              updateChildNode(TNode,2);
                              pathnode* U = deeplimited(TNode->up, k);
```

```
                                if (U != NULL) {
                                        return U;
                                }
                        }
                        //generate the down step child node
                        if (okdown) {
                                updateChildNode(TNode, 1);
                                pathnode* D = deeplimited(TNode->down, k);
                                if (D != NULL) {
                                        return D;
                                }
                        }
                        return NULL;
                }
}
pathnode* iterativeDeepening(pathnode* TNode) {
                pathnode* result = NULL;
                for (int m = 0;; m++) {
                        result = deeplimited(TNode, m);
                        if (result != NULL)
                                break;
                }
                return result;
}
//print a node's board
void printstate(pathnode* TNode) {
                cout << "---------" << endl;
                //print different sign
                for (int i = 0; i < N; i++) {
                        for (int j = 0; j < N; j++) {
                                if (TNode->state[i][j] == 0)
                                        cout << "| ";
                                if (TNode->state[i][j] == 1)
                                        cout << "|A";
                                if (TNode->state[i][j] == 2)
                                        cout << "|B";
                                if (TNode->state[i][j] == 3)
                                        cout << "|C";
                                if (TNode->state[i][j] == 4)
                                        cout << "|M";
                                if (TNode->state[i][j] == 5) {
                                        cout << "|*";
                                }
                                if ((j + 1) % N == 0) {
                                        cout <<"|"<<endl;
                                        cout << "---------" << endl;
                                }
                        }
                }
}
//print all the steps through a goal node
void printprocess(pathnode* A,pathnode* Head) {
                //Find the complete route to reach the aim node
                while (A != Head) {
                        A->father->stepnext = A;
```

```
                A = A->father;
        }
        //Print the route
        while (A->state[1][1] != 1 || A->state[2][1] != 2 || A->state[3][1] != 3) {
                cout << "The " << A->depth << " step is :" << endl;
                printstate(A);
                A = A->stepnext;
        }
        cout << "The " << A->depth << " step is :" << endl;
        printstate(A);
}
int main()
{
        pathnode* Head = new pathnode();
        pathnode*A = Astar(Head);//choose search method
        if (A == NULL)
                cout << "this way is in infinited loop" << endl;
        else
                printprocess(A, Head);
        cout << "The number of generated node is " << NodeNumber << "." << endl;
        cout << "The search is end"<<endl;
}
```