前言

重排九宫就是重新排列九宫图的意思。这是根据当时盛行研究的数学游戏——纵横图（也叫幻方或魔方阵）发展来的，九宫游戏的起源，更可追溯到我国远古神话历史时代的河图、洛书。重排九宫则属滑块类游戏。在使用算术的同时，还必须推动方块使其到相对应的位置。其玩法是在3×3方格盘上，放有1-8八个数，剩下一格为空，每一空格其周围的数字可移至空格。先设定初始排列数字，然后开始思考如何以最少的移动次数来达到目的排列状态。

目前，针对该问题的一般解法为广度优先搜索。对于某一个确定的初始状态和目标状态，通过广度优先的方式，遍历所有的可能性，最终找到一条从初始状态到目标状态的路径，或者无解。

此外，由于广度优先是盲目搜索的，也可以结合启发式算法，如A*算法，在广度优先搜索的基础上，加入一个启发式函数来优化搜索过程，最终常常可以减少找到解时所生成的节点数量，增强算法的性能。

摘要

本文结合广度优先搜索的思想，编写C++代码成功实现了重排九宫问题的计算机求解。对于任意的初始状态及目的状态，如果有解，代码会输出从初始状态到目标状态最短路径。

此外，在尝试使用多个示例时，本文也发现使用广度优先搜索的重排九宫算法对某些目标状态的求解产生了大量不必要的节点。在查阅相关资料后，本文在广度优先搜索的基础上，结合了A*算法等启发式思想，成功改进了相关算法，能够在大部分情况下，更好的求解重排九宫问题。

算法描述

广度优先搜索

算法

将九宫格的每一种可能的摆放方式定义为九宫格的一种状态。初始时，程序读入用户输入的初始状态与目标状态，分别存储在head和target结构中。之后，对于每一次程序执行，定义队列wait表示等待处理的状态，定义集合used表示已经处理过的状态，防止重复处理。在算法正式执行前，将head节点置入wait队列，used清零。

之后，从wait队列中取出状态，如果该状态为目标状态，算法结束，找到路径；如果该状态不是目标状态，将经由该状态可以抵达、且不再used集合中的状态加入wait队列及used集合。然后，重复从wait队列取出状态进行判断，直到找到目标状态或者wait队列为空时，算法结束。

程序结构

项目文件结构如下：

📦root

┣ 📂include

┃ ┣ 📜BFS.h

┃ ┗ 📜Node.h

┣ 📂src

┃ ┣ 📜BFS.cpp

┃ ┣ 📜Node.cpp

┃ ┗ 📜main.cpp

┣ 📜CMakeLists.txt

┣ 📜LICENSE

┗📜README.md

其中，在Node.h文件中定义了表示九宫格状态的结构体struct Node,代码如下：

```
struct Node {
    int content[ROW][COL]; // 数据成员
    int r;
    int c;
    Node* parent;

    // 构造函数声明
    Node();

    // 成员函数声明
    void print();
    void set_value(int row, int col, int value, Node* p=nullptr);
    Node* up();
    Node* left();
    Node* right();
    Node* down();

    bool operator==(const Node& other) const;
    bool operator!=(const Node& other) const;
    bool operator<(const Node& other) const;

};
```

content表示该状态内容，r、c表示空格所在位置，parent是指向生成该状态的父状态。对于==、！=、<则定义了忽略父节点指针的重载运算。

在BFS.h文件中，定义了BFS类，包含了使用广度优先搜索求解问题所需要的所有变量和函数，代码如下：

```
class BFS
{
private:
    Node head;
    Node target;
    vector<Node> path;
    set<Node> used;

public:
    BFS(/* args */);

    void FindPath();
    void Print();
    bool NotUsed(Node& node);
};
```

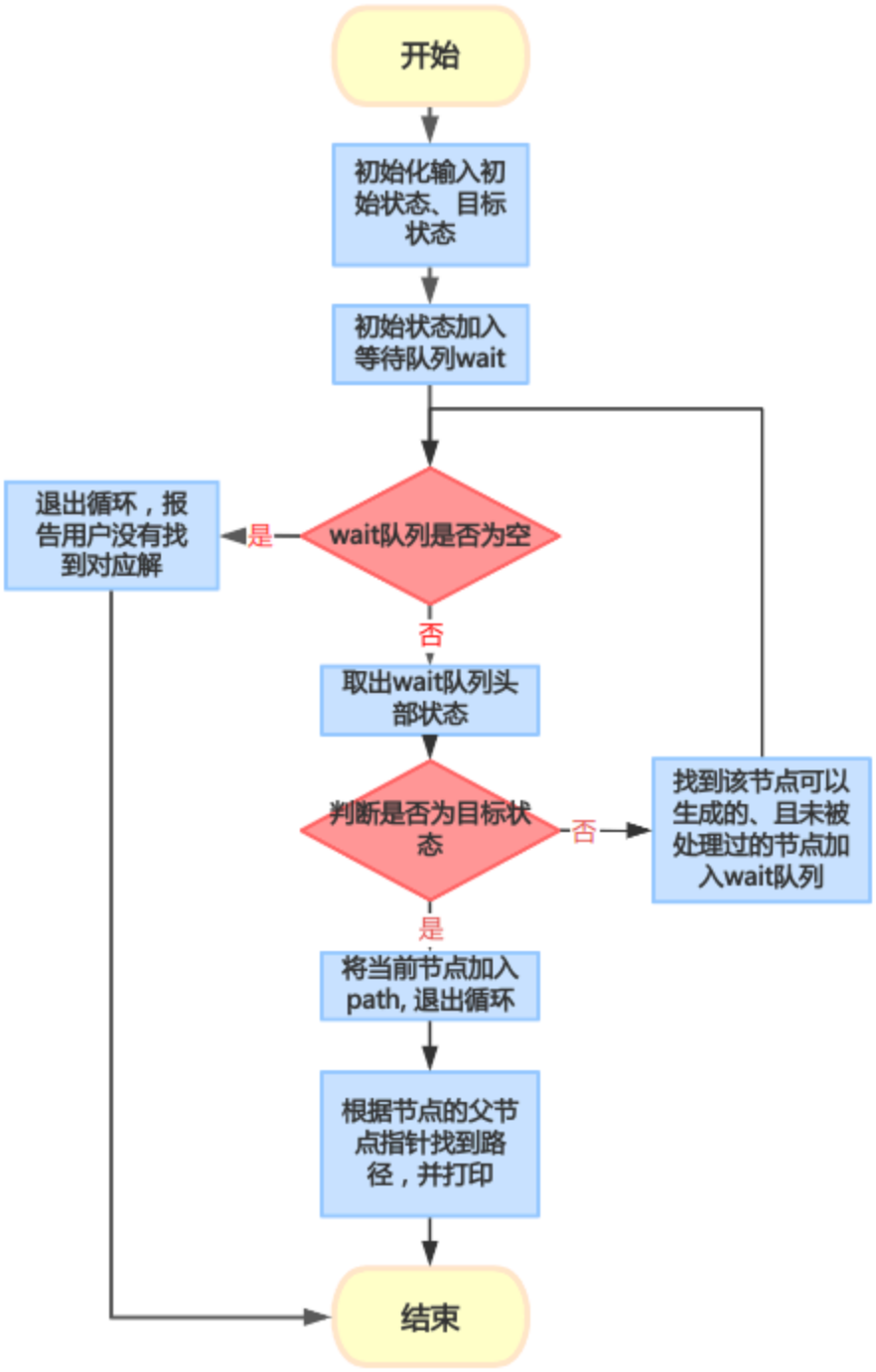
其中，head为初始状态，target为目标状态，path为最终的路径，used表示已经处理过的状态集合。函数FindPath用来寻找该具体问题的解，代码如下：

```
void BFS::FindPath()
{
    queue<Node*> wait;
    wait.emplace(&head);
    used.insert(head);
    while (!wait.empty())
    {
        Node* curr = wait.front();
        wait.pop();

        if (*curr == target)
        {
            path.emplace_back(*curr);
            while (path.back().operator!=(head))
            {
                path.emplace_back(*(path.back().parent));
            }
            break;
        }
        Node* temp = curr->up();
        if (temp && used.count(*temp) == 0)
        {
            wait.emplace(temp);
            used.insert(*temp);
        }
        temp = curr->left();
        if (temp && used.count(*temp) == 0)
        {
            used.insert(*temp);
            wait.emplace(temp);
        }
        temp = curr->right();
        if (temp && used.count(*temp) == 0)
        {
            used.insert(*temp);
            wait.emplace(temp);
        }
        temp = curr->down();
        if (temp && used.count(*temp) == 0)
```

```
        {
            used.insert(*temp);
            wait.emplace(temp);
        }
    }
}
```

流程图

算法流程图如下：



多次运行结果

示例1

初始状态：

2 8 3

1 -1 4

7 6 5

目标状态：
1 2 3
8 -1 4
7 6 5
运行结果：

```
2 8 3
1   4
7 6 5
 ↓
2   3
1 8 4
7 6 5
 ↓
  2 3
1 8 4
7 6 5
 ↓
1 2 3
  8 4
7 6 5
 ↓
1 2 3
8   4
7 6 5
```

示例2
初始状态：
2 8 3
1 -1 4
7 6 5
目标状态：
2 4 3
1 8 -1
7 6 5
运行结果：

```
请输入初始节点状态
2 8 3
1 -1 4
7 6 5
2 4 3
1 8 -1
7 6 5请输入目标节点状态

没有从初始节点到目标节点的路径
```

示例3

初始状态：

1 2 3

4 5 6

7 8 -1

目标状态：

1 2 3

-1 4 6

7 5 8

运行结果：

```
1 2 3
4 5 6
7 8
 ↓
1 2 3
4 5 6
7   8
 ↓
1 2 3
4   6
7 5 8
 ↓
1 2 3
  4 6
7 5 8
```

启发式搜索

算法

考虑到广度优先搜索中，节点的添加与处理是盲目的，常常会处理很多不必要的节点。为改善该问题，等待队列中的每一个待处理状态增添一个有启发式函数得到的预估从初始节点经由该节点到目标节点的代价。计算方法如下：

f(n)=g(n)+h(n)

- g(n) 是从初始状态到当前状态的实际代价（即已移动步数）。
- h(n) 是当前状态到目标状态的估计代价。

这里采用曼哈顿距离作为启发式函数，即对于每个数字 ，计算它当前位置（x1, y1）和目标位置（x2, y2）的曼哈顿距离（|x1 - x2|+|y1 - y2|），再将这些曼哈顿距离求和得到估计代价。

之后，在从wait等待队列中取节点时，选择代价最小的节点而不是头部节点。

程序结构

定义关键数据结构，不仅包含每个状态的Node节点，还包含该状态的生成层级与估计的从初始状态经由该状态到目标状态的代价。代码如下：

```
struct pNode
{
    Node* node;
    int depth;
    int cost;
    pNode(): node(nullptr), depth(0), cost(0) {}
};
```

此外，为便于计算曼哈顿距离，为Astar类增添数据positions，记录target每个数字所在的位置。修改后Astar类代码如下：

```
class Astar: public BFS
{
private:
    vector<pair<int, int>> positions;
public:
    Astar(/* args */);
    void FindPath() override;
    int ManhattanDistance(Node& node);
};
```

其中，ManhattanDistance函数用于计算当前状态的曼哈顿距离。重写后的FindPath函数代码如下：

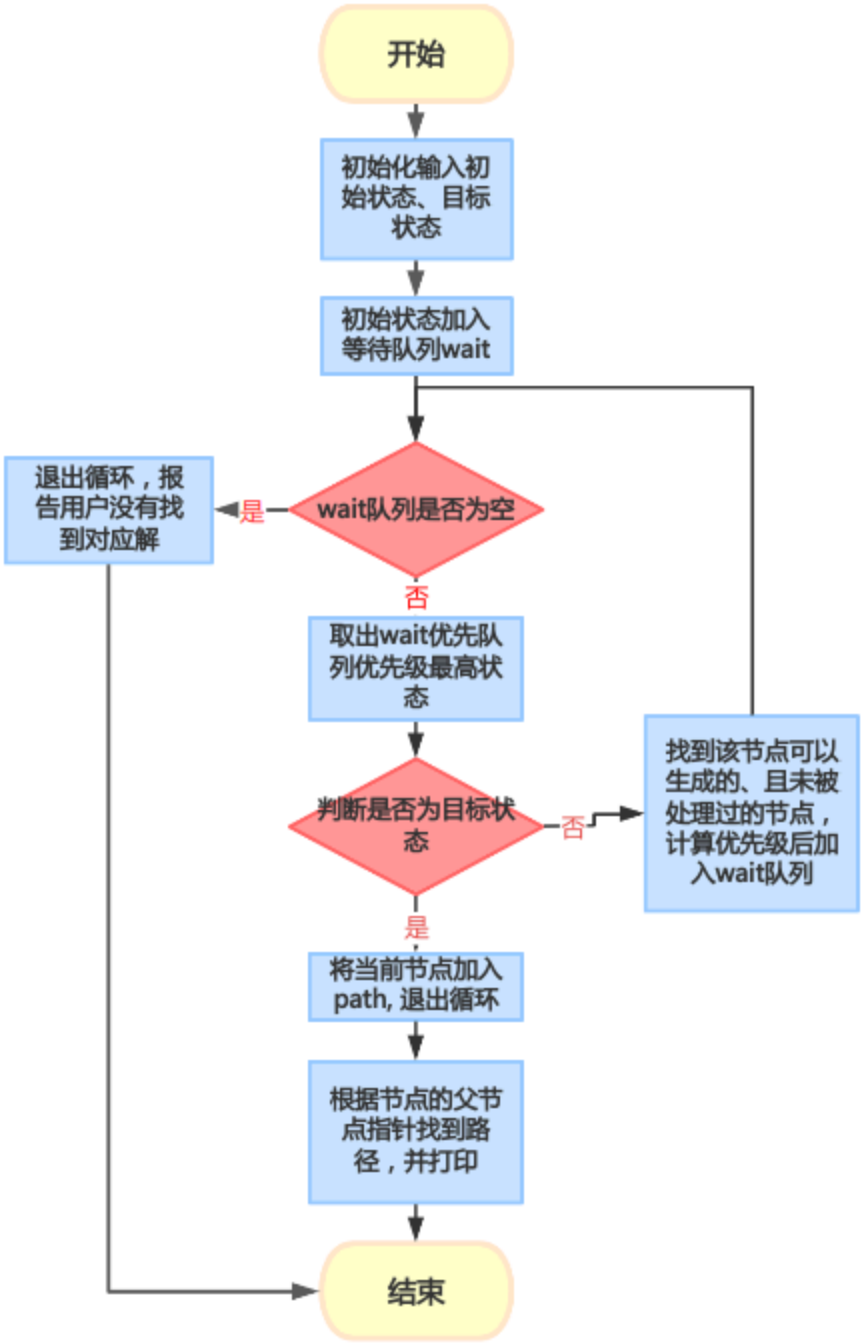
```
void Astar::FindPath()
{
    priority_queue<pNode, vector<pNode>, ComparePNode> wait;
    pNode h;
    h.node = &head;
    h.depth = 0;
    h.cost = ManhattanDistance(head) + h.depth;
    wait.emplace(h);
    used.insert(head);
    while (!wait.empty())
    {
        pNode c = wait.top();
        Node* curr = c.node;
        wait.pop();

        if (*curr == target)
        {
            path.emplace_back(*curr);
            while (path.back().operator!=(head))
            {
                path.emplace_back(*(path.back().parent));
            }
            break;
        }
        Node* temp = curr->up();
        pNode newpNode;
        if (temp && used.count(*temp) == 0)
        {
            newpNode.depth = c.depth+1;
            newpNode.cost = ManhattanDistance(*temp) + newpNode.depth;
            newpNode.node = temp;
            wait.emplace(newpNode);
            used.insert(*temp);
        }
        temp = curr->left();
        if (temp && used.count(*temp) == 0)
        {
            newpNode.depth = c.depth+1;
            newpNode.cost = ManhattanDistance(*temp) + newpNode.depth;
```

```
            newpNode.node = temp;
            wait.emplace(newpNode);
            used.insert(*temp);
        }
        temp = curr->right();
        if (temp && used.count(*temp) == 0)
        {
            newpNode.depth = c.depth+1;
            newpNode.cost = ManhattanDistance(*temp) + newpNode.depth;
            newpNode.node = temp;
            wait.emplace(newpNode);
            used.insert(*temp);
        }
        temp = curr->down();
        if (temp && used.count(*temp) == 0)
        {
            newpNode.depth = c.depth+1;
            newpNode.cost = ManhattanDistance(*temp) + newpNode.depth;
            newpNode.node = temp;
            wait.emplace(newpNode);
            used.insert(*temp);
        }
    }
}
```

将wait队列修改为使用自定义规则的优先队列存储，每次输出优先级最高（即预估代价最小）的节点进行处理。

流程图

使用启发式搜索的重排九宫算法修改了原先队列的数据结构，修改后程序的流程图如下：



比较

选用多个示例，比较广度优先搜索与启发式搜索算法的结果，将两种算法处理节点数目与具体示例列为表格，表格中-1表示空格，如下：

初始状态	目标状态	广度优先搜索处理节点数目	启发式搜索处理节点数目
2 8 3 1 -1 4 7 6 5	1 2 3 8 -1 4 7 6 5	37	10
1 2 3 4 5 6 7 8 -1	1 2 3 -1 4 6 7 5 8	25	8
2 8 3 1 6 4 7 -1 5	1 2 3 8 -1 4 7 6 5	61	12

可见，在采用了启发式搜索后，搜索到解时所处理的节点大大减少，说明启发式搜索算法提供了更强的性能。

总结

本文实现了两种算法：广度优先搜索与启发式搜索，求解重排九宫问题。每种算法都通过多个示例验证了代码的可行性。此外，也通过一些示例的比较，发现采用广度优先搜索算法是盲目搜索的，会处理许多不必要的状态，影响算法的性能。而采用曼哈顿距离作为启发式函数的启发式搜索会优先选择更可能抵达目标节点的状态，减少了冗余状态的处理，有效改善算法的性能。

可能的改进方向

本文实现了广度优先搜索算法与启发式搜索算法，在这个基础上可以有两种易于实现的改进方式：

增添问题无解的判定

在本文的两种算法中，对于没有解的实例，算法仍然会遍历所有可能的状态（总计181440个），最终依据wait队列为空判断无解，严重影响了算法的效率。
对于重排九宫问题，可直接通过以下方式判断是否有解：

- 计算逆序数：对于数组中的每一对数字 ◇◇和 ◇◇，如果 ◇<◇且 ◇◇>◇◇，则该对元素形成一个逆序。逆序数就是这样所有逆序对的总数。
- 判断是否有解：
    - 如果逆序数是偶数，则该排列是可解的；
    - 如果逆序数是奇数，则该排列是不可解的。

使用更优越的启发式函数

在某些情况下，曼哈顿距离可能低估了某些状态之间的实际距离，特别是当数字移动的过程中可能会有较多的交换时。可以考虑使用以下启发式函数，可能会有更好的性能：

1. 错位数（Misplaced Tiles）
   错位数启发式计算的是当前状态与目标状态中不同的数字的数量。也就是说，计算当前状态中与目标状态不一致的数字个数。
   错位数的定义：
   错位数启发式的值是当前状态与目标状态不一致的数字个数，通常是通过比较两个状态中的数字来计算。
    - 优点：
      计算非常简便，适合快速实现。
    - 缺点：
      错位数不能很好地反映某些状态之间的实际“距离”，因此可能不如曼哈顿距离有效，尤其是在需要考虑多个数字交换的情况时。
2. 线性冲突（Linear Conflict）
   线性冲突是一种在曼哈顿距离的基础上进一步优化的启发式函数。线性冲突发生在两个数字处于同一行或列，并且它们的目标位置是互相对调的。在这种情况下，这两个数字必须交换位置，这增加了额外的代价。
   线性冲突的定义：
    - 线性冲突是在同一行或同一列上的两个数字，它们的位置与目标状态中的位置互相对调。
    - 每个线性冲突会为启发式函数增加 2 的值（因为每个冲突的解决需要两步）。

源代码

/include/Astar.h

```
#ifndef ASTAR_H
#define ASTAR_H

#include "./Node.h"
#include "BFS.h"

struct pNode
{
    Node* node;
    int depth;
    int cost;
    pNode(): node(nullptr), depth(0), cost(0) {}
};

struct ComparePNode
{
    bool operator()(const pNode& p1, const pNode& p2) const
    {
        return p1.cost > p2.cost;
    }
};


class Astar: public BFS
{
private:
    vector<pair<int, int>> positions;
public:
    Astar(/* args */);
    void FindPath() override;
    int ManhattanDistance(Node& node);
};


#endif
```

/include/BFS.h

```
#ifndef BFS_H
#define BFS_H

#include "./Node.h"
#include <vector>
#include <set>
using namespace std;

class BFS
{
protected:
    Node head;
    Node target;
    vector<Node> path;
    set<Node> used;

public:
    BFS(/* args */);

    virtual void FindPath();
    void Print();
    bool NotUsed(Node& node);
};
#endif
```

/include/Node.h

```
#ifndef NODE_H
#define NODE_H

#include <iostream>  // 包含输入输出流
#include <optional>

#define ROW 3
#define COL 3

struct Node {
    int content[ROW][COL]; // 数据成员
    int r;
    int c;
    Node* parent;

    // 构造函数声明
    Node();

    // 成员函数声明
    void print();
    void set_value(int row, int col, int value, Node* p=nullptr);
    Node* up();
    Node* left();
    Node* right();
    Node* down();

    bool operator==(const Node& other) const;
    bool operator!=(const Node& other) const;
    bool operator<(const Node& other) const;
};
#endif // NODE_H
```

/src/Astar.cpp

```cpp
#include "Astar.h"
#include <queue>
#include <utility>
#include <cmath>

Astar::Astar()
{
    positions.resize(ROW * COL);
    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0;j < COL;j++)
        {
            if (target.content[i][j] == -1)
            {
                positions[0] = make_pair(i, j);
                continue;
            }
            positions[target.content[i][j]] = make_pair(i, j);
        }
    }
}

void Astar::FindPath()
{
    priority_queue<pNode, vector<pNode>, ComparePNode> wait;
    pNode h;
    h.node = &head;
    h.depth = 0;
    h.cost = ManhattanDistance(head) + h.depth;
    wait.emplace(h);
    used.insert(head);
    while (!wait.empty())
    {
        pNode c = wait.top();
        Node* curr = c.node;
        wait.pop();

        if (*curr == target)
        {
```

```
            path.emplace_back(*curr);
            while (path.back().operator!=(head))
            {
                path.emplace_back(*(path.back().parent));
            }
            break;
        }
        Node* temp = curr->up();
        pNode newpNode;
        if (temp && used.count(*temp) == 0)
        {
            newpNode.depth = c.depth+1;
            newpNode.cost = ManhattanDistance(*temp) + newpNode.depth;
            newpNode.node = temp;
            wait.emplace(newpNode);
            used.insert(*temp);
        }
        temp = curr->left();
        if (temp && used.count(*temp) == 0)
        {
            newpNode.depth = c.depth+1;
            newpNode.cost = ManhattanDistance(*temp) + newpNode.depth;
            newpNode.node = temp;
            wait.emplace(newpNode);
            used.insert(*temp);
        }
        temp = curr->right();
        if (temp && used.count(*temp) == 0)
        {
            newpNode.depth = c.depth+1;
            newpNode.cost = ManhattanDistance(*temp) + newpNode.depth;
            newpNode.node = temp;
            wait.emplace(newpNode);
            used.insert(*temp);
        }
        temp = curr->down();
        if (temp && used.count(*temp) == 0)
        {
            newpNode.depth = c.depth+1;
```

```
            newpNode.cost = ManhattanDistance(*temp) + newpNode.depth;
            newpNode.node = temp;
            wait.emplace(newpNode);
            used.insert(*temp);
        }
    }
}

int Astar::ManhattanDistance(Node &node)
{
    int cost = 0;
    for (int i = 0; i < ROW; i++)
    {
        for(int j=0;j<COL;j++)
        {
            int curr = node.content[i][j];
            if (curr == -1)
            {
                continue;
            }

            cost += abs(i - positions[curr].first) + abs(j - positions[curr].second);
        }
    }
    return cost;
}
```

/src/BFS.cpp

```cpp
#include "BFS.h"
#include <queue>
#include <optional>

BFS::BFS()
{
    cout<<"请输入初始节点状态"<<endl;
    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0; j < COL; j++)
        {
            int value;
            cin>>value;
            head.set_value(i, j, value);
        }

    }
    cout<<"请输入目标节点状态"<<endl;
    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0; j < COL; j++)
        {
            int value;
            cin>>value;
            target.set_value(i, j, value);
        }

    }

}

void BFS::FindPath()
{
    queue<Node*> wait;
    wait.emplace(&head);
    used.insert(head);
    while (!wait.empty())
    {
        Node* curr = wait.front();
```

```
        wait.pop();

        if (*curr == target)
        {
            path.emplace_back(*curr);
            while (path.back().operator!=(head))
            {
                path.emplace_back(*(path.back().parent));
            }
            break;
        }
        Node* temp = curr->up();
        if (temp && used.count(*temp) == 0)
        {
            wait.emplace(temp);
            used.insert(*temp);
        }
        temp = curr->left();
        if (temp && used.count(*temp) == 0)
        {
            used.insert(*temp);
            wait.emplace(temp);
        }
        temp = curr->right();
        if (temp && used.count(*temp) == 0)
        {
            used.insert(*temp);
            wait.emplace(temp);
        }
        temp = curr->down();
        if (temp && used.count(*temp) == 0)
        {
            used.insert(*temp);
            wait.emplace(temp);
        }
    }
}

void BFS::Print()
```

```
{
    cout<<"处理节点数目: "<<used.size()<<endl;
    if (path.empty())
    {
        cout<<"没有从初始节点到目标节点的路径"<<endl;
        return;
    }

    cout<<endl;

    for (int i = path.size()-1; i >= 0; i--)
    {
        path[i].print();
        if (i == 0)
        {
            break;
        }

        cout<<" \u2193"<<endl;
    }


}

bool BFS::NotUsed(Node &node)
{
    return false;
}
```

/src/main.cpp

```cpp
#include "../include/BFS.h"
#include "../include/Node.h"
#include "Astar.h"
int main()
{
    BFS solution;
    solution.FindPath();
    solution.Print();
    Astar solution2;
    solution2.FindPath();
    solution2.Print();
    return 0;
}
```

/src/Node.cpp

```cpp
#include "../include/Node.h"
#include <cstring>  // 用于 memset
#include "Node.h"

// 构造函数定义
Node::Node()
{
    memset(content, 0, sizeof(content));
    parent = nullptr;
}

// print 函数定义
void Node::print() {
    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0; j < COL; j++)
        {
            if (content[i][j] == -1)
            {
                std::cout << " " << " ";
                continue;
            }

            std::cout << content[i][j] << " ";
        }
        std::cout << std::endl;
    }
    // std::cout << std::endl;
}

// set_value 函数定义
void Node::set_value(int row, int col, int value, Node* p)
{
    if (row < ROW && row >= 0 && col < COL && col >= 0)
    {
        if (value == -1)
        {
            r = row;
            c = col;
```

```
        }

        content[row][col] = value;
    }
    else
    {
        std::cerr << "索引超出界限" << std::endl;
    }

    parent = p;

}

Node* Node::up()
{
    if (r-1 >= 0)
    {
        if (this->parent && this->parent->r == r-1)
        {
            return nullptr;
        }

        Node* newNode = new Node(*this);
        newNode->content[r][c] = newNode->content[r-1][c];
        newNode->content[r-1][c] = -1;
        newNode->r = r-1;
        newNode->c = c;
        newNode->parent = this;
        return newNode;
    }
    return nullptr;
}

Node* Node::left()
{
    if (c-1 >= 0)
    {
        if (this->parent && this->parent->c == c-1)
        {
```

```
            return nullptr;
        }
        Node* newNode = new Node(*this);
        newNode->content[r][c] = newNode->content[r][c-1];
        newNode->content[r][c-1] = -1;
        newNode->r = r;
        newNode->c = c-1;
        newNode->parent = this;
        return newNode;
    }
    return nullptr;
}

Node* Node::right()
{
    if (c+1 < COL)
    {
        if (this->parent && this->parent->c == c+1)
        {
            return nullptr;
        }
        Node* newNode = new Node(*this);
        newNode->content[r][c] = newNode->content[r][c+1];
        newNode->content[r][c+1] = -1;
        newNode->r = r;
        newNode->c = c+1;
        newNode->parent = this;
        return newNode;
    }
    return nullptr;
}

Node* Node::down()
{
    if (r+1 < ROW)
    {
        if (this->parent && this->parent->r == r+1)
        {
            return nullptr;
```

```
        }
        Node* newNode = new Node(*this);
        newNode->content[r][c] = newNode->content[r+1][c];
        newNode->content[r+1][c] = -1;
        newNode->r = r+1;
        newNode->c = c;
        newNode->parent = this;
        return newNode;
    }
    return nullptr;
}

bool Node::operator==(const Node &other) const
{
    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0; j < COL; j++)
        {
            if (content[i][j] != other.content[i][j])
            {
                return false;
            }

        }

    }

    return true;
}

bool Node::operator!=(const Node &other) const
{
    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0; j < COL; j++)
        {
            if (content[i][j] != other.content[i][j])
            {
                return true;
```

```
            }

        }

    }

    return false;
}

bool Node::operator<(const Node& other) const
{
    for (int i = 0; i < ROW; i++)
    {
        for (int j = 0; j < COL; j++)
        {
            if (content[i][j] != other.content[i][j])
            {
                return content[i][j] < other.content[i][j];
            }
        }

    }

    return false;
}
```

/CMakeLists.txt

```
# 设置CMake的最低版本要求
cmake_minimum_required(VERSION 3.10)

# 设置项目名称
project(MyProject VERSION 1.0 LANGUAGES CXX)

# 设置C++标准
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# 设置生成的二进制文件目录
set(CMAKE_RUNTIME_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/bin)

# 设置中间目标文件目录
set(CMAKE_ARCHIVE_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)
set(CMAKE_LIBRARY_OUTPUT_DIRECTORY ${CMAKE_BINARY_DIR}/lib)

# 包含头文件目录
include_directories(${CMAKE_SOURCE_DIR}/include)

# 如果有第三方库，可以指定库路径
# link_directories(${CMAKE_SOURCE_DIR}/lib)

# 定义源文件
set(SOURCE_FILES
    src/main.cpp
    src/BFS.cpp
    src/Node.cpp
    src/Astar.cpp
)

set(HEADERS
    include/BFS.h
    include/Node.h
    include/Astar.h
)

# 定义可执行文件
add_executable(MyProject ${SOURCE_FILES} ${HEADERS})
```