数据结构课程设计

项目说明文档

勇闯迷宫游戏

作者姓名: 汪明杰

学　　号: 1851055

指导教师: 张　颖

学院专业: 软件学院 软件工程

同济大学

Tongji University

目录

1 项目分析

1.1 项目背景

迷宫只有两个门，一个门叫入口，另一个门叫出口。一个骑士骑马从入口进入迷宫，迷宫设置很多障碍，骑士需要在迷宫中寻找通路以到达出口。

事实上，对于迷宫问题进行求解在现实生活中有着许多的实际应用场景。例如：日常使用的百度地图就需要通过寻路算法来为我们指引正确的道路。而迷宫问题正是其问题的一个简化，能够快速有效的解决迷宫问题能够为我们未来研究更加深入的算法提供最为基础的理论依据。

1.2 项目需求分析

对于迷宫寻路算法这个问题，我们在实现的时候首先考虑到降低时间复杂度，即在一个迷宫中寻路的时候应当尽可能快速的找出合适的路径。此外，为了方便测试，还需要设计一个算法来实现快速的生成迷宫，即迷宫生成算法。

- ✓ **功能完善**

  本系统应当整体完成度比较高，即同时完成了生成迷宫算法和迷宫寻路算法。

- ✓ **执行效率高**

  当用户所输入的迷宫数据比较大的情况下，该系统也应当能够在比较短的时间内生成出相对复杂度比较高的迷宫。此外，也需要能够在比较短的时间内找出所要求解两点间的路径。

- ✓ **健壮性**

  当用户输入的数据不合理时，系统应当给予相应的提示而非直接报错。

- ✓ **迷宫可视化**

  该系统应当将迷宫和寻找到的路径通过字符展示出来，以便直观的显示寻找到的路径。

2 项目设计

2.1 数据结构设计

本项目中使用了多种算法，在算法的使用过程中涉及到了多种数据结构，包括：向量、栈、优先级队列等。同时，本项目也实现了这些数据结构，在此对这些数据结构类型加以介绍。

2.1.1 向量类（Vector）

向量（Vector）是一个封装了动态大小数组的顺序容器（Sequence Container）。跟任意其它类型容器一样，它能够存放各种类型的对象。可以简单的认为，向量是一个能够存放任意类型的动态数组。需要注意到的是，向量具有以下两个特性：

（1） 顺序序列：

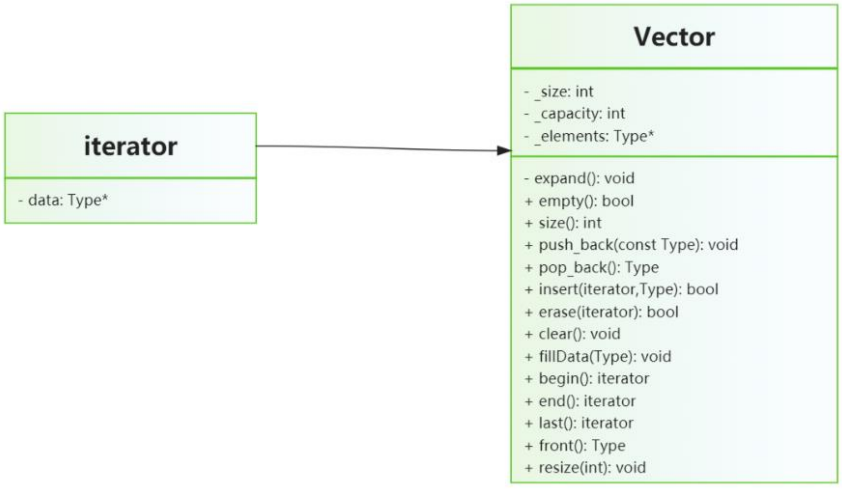
顺序容器中的元素按照严格的线性顺序排序，可以通过元素在序列中的位置访问对应的元素。本类中即通过重载取下标运算符[]实现了此功能。

（2） 动态数组：

支持对序列中的任意元素进行快速直接访问，甚至可以通过指针算术进行该操作，提供了在序列末尾相对快速地添加或者删除元素的操作。

本项目为了实现向量类，在内部定义了一个动态指针以及当前数组的大小。同时，封装了较多的函数以便使用。此外，为了便于对向量进行遍历、插入、删除、查找等操作，增加了一个 iterator 类。iterator 类内部存储一个数组元素指针，可以直接对向量的每个元素进行操作。同时，通过运算符重载相应的自增、自减、判等等操作。

该类和其内部的 iterator 类的 UML 图如下所示：



本类中的主要函数如下所示：

◆ `inline int size()const`

返回向量中元素的个数，也即_size 的大小。

◆ `inline bool empty()const`

判断向量是否为空，也即_size 是否为 0。

◆ `void push_back(const Type data)`

在向量尾端加入一个元素。

◆ `Type pop_back()`

删除向量最后一个元素，并且加以返回。

◆ `bool insert(const Vector<Type>::iterator place, Type item)`

在指定迭代器的位置插入元素，返回是否插入成功。

◆ `bool erase(const Vector<Type>::iterator place)`

删除指定迭代器位置的元素，返回是否删除成功。

◆ `void clear()`

清空向量中所有元素。

◆ `void fillData(const Type data)`

将向量中的元素统一赋值为同一个值。

◆ `iterator begin()`

返回向量首元素的迭代器。

◆ `iterator end()`

返回向量尾元素的下一个位置的迭代器。

◆ `iterator last()`

返回向量尾元素的迭代器。

◆ `Type& front()const`

返回向量的首个元素值。

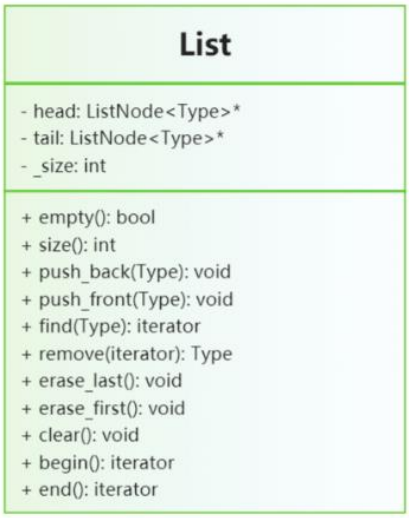
◆ `void resize(int sz)`

将向量重新设定大小，如果变小则删除多余元素。

2.1.2 双向链表类（List）

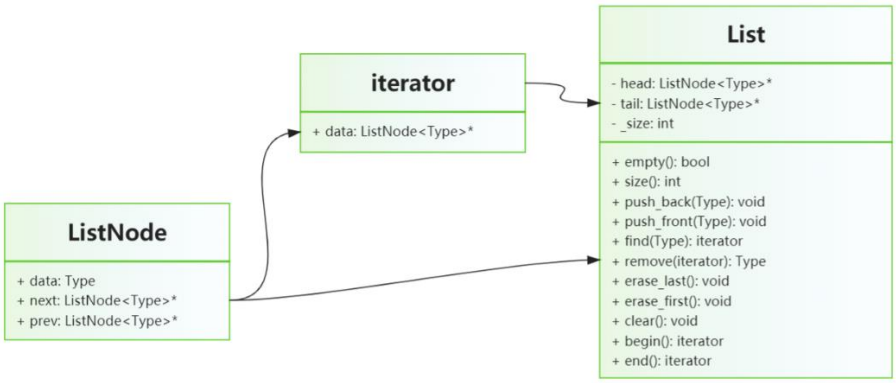
链表的实现原理大同小异，不同之处在于：是否带头结点、是否带尾结点、每一个结点是否带前驱结点等。为了使得链表中各种操作的时间复杂度都尽可能低，因此这里选择了带头结点和尾结点的双向链表来实现链表中的各种操作。也即：选择了牺牲空间来达到降低时间复杂度的效果。

链表的 UML 图如下所示：



为了便于对链表进行遍历、插入、删除、查找等操作，增加了一个 iterator 类。iterator 类内部存储一个链表节点指针，同时，通过运算符重载相应的自增、自减、判等等操作。

iterator 类、ListNode 类和 List 类的关系如下图所示：



其中，List 类中的主要函数如下所示：

◆ `inline int size()const`

返回链表中结点的个数，不包括头结点。

◆ `inline bool empty()const`

判断链表是否为空，也即链表中结点的个数是否为 0。

◆ `void push_back(Type data)`

在链表尾部插入新的数据，也即新增一个结点并且加入到链表末端。

◆ `void push_front(Type data)`

在链表头部插入新的数据，也即新增一个结点并且加入到链表的头部。

◆ `iterator find(const Type& data)const`

在链表中查找值为 data 的元素是否存在，返回该位置的迭代器，若查找失败返回空指针对应的迭代器。

◆ `Type remove(iterator index)`

移除迭代器所处位置的元素，返回移除位置元素的值。

◆ `void erase_last()`

移除链表末端的元素，即最后的结点。

◆ `void erase_first()`

移除链表首端的元素，即第一个结点。

◆ `void clear()`

清空链表，即删除链表中所有的结点。

◆ `iterator begin()`

返回链表第一个元素所在位置的迭代器。

◆ `iterator end()`

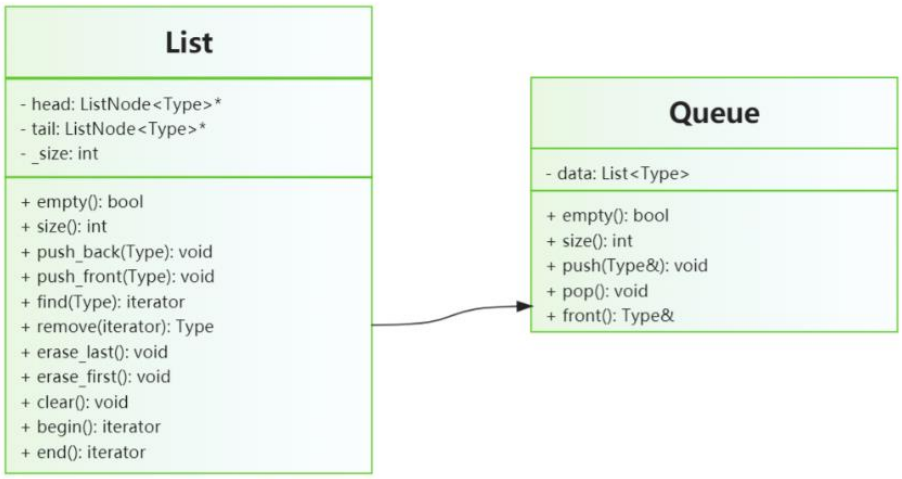
返回链表尾结点后的空结点的迭代器。

2.1.3 队列类（Queue）

队列是计算机程序中常用的数据结构，常常用于计算机模拟现实事务，如：排队等。队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，因此是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头。队列中没有元素时，被称为空队列。

队列的数据元素又称为队列元素。在队列中插入一个队列元素称为入队，从队列中删除一个队列元素称为出队。因为队列只允许在一端插入，在另一端删除，所以只有最早进入队列的元素才能最先从队列中删除，故队列又称为先进先出（FIFO—first in first out）线性表。

队列的包括了顺序队列和循环队列，实现存储的底层数据可以通过向量或者链表完成。本项目中的队列以链表作为底层数据结构，实现了顺序队列。其 UNL 图如下所示：



队列中主要函数如下所示：

◆ `inline bool empty()const`

判断队列是否为空，也即队列内部链表是否为空。

◆ `inline int size()const`

返回队列中链表节点的个数。

◆ `void push(const Type& i)`

在队列尾部加入一个元素，也即入队。

◆ `void pop()`

删除队列头部的元素，也即出队。

◆ `const Type& front()const`

获取队首的元素值。

通过上述函数操作，即完成了一个队列所需要的最基本操作。

2.1.4 堆（Heap）

堆（Heap）是一类特殊数据结构的简称，通常是一个可以被看做一棵树的的数组对象。堆满足下述两个性质：

（1） 堆中某个节点的值总是不大于或者不小于其父节点的值；

（2） 堆总是一棵完全二叉树。

将根节点最大的堆叫做最大堆或者大根堆，根节点最小的堆叫做最小堆小根堆。常见的堆有二叉堆、斐波拉契堆等。

本项目中堆的底层数据结构使用向量来实现，通过定义建堆、调整堆来完成了一个堆。主要函数如下所示：

◆ **建堆函数**

将[start,end)内的元素调整为堆

```
template <class Iter>
void makeHeap(Iter start, Iter end)
{
    typedef typename Iter::value_type value_type;
    ptrdiff_t len = end - start;
    if (len < 2) return;
    ptrdiff_t hole = (len - 2) / 2;
    while (hole >= 0) {
        //调整 hole 节点
        sinkFix(start, len, hole, *(start + hole), Less<value_type>());
        --hole;
    }
}
```

◆ **向上调整**

向上调整 hole 的位置，找到合适的位置将 value 存入

```
template <class Iterator, typename T, class Compare>
void siftFix(Iterator first,
```

```
    ptrdiff_t top,
    ptrdiff_t hole,
    T value,
    Compare comp)
{
    ptrdiff_t parent = (hole - 1) / 2;
    while (parent >= top && comp(*(first + parent), value)) {
        *(first + hole) = *(first + parent);
        hole = parent;
        parent = (hole + 1) / 2 - 1;
    }
    *(first + hole) = value;
}
```

◆ 向下调整

向下调整 hole 的位置，找到合适的位置将 value 存入

```
template <class Iterator, typename T, class Compare>
void sinkFix(Iterator first,
    ptrdiff_t len,
    ptrdiff_t hole,
    T value,
    Compare comp)
{
    /* 1.将hole 下移到叶子处 */
    ptrdiff_t bigChild = 2 * hole + 2;  //从右孩子开始考虑
    while (bigChild < len) {
        if (comp(*(first + bigChild), *(first + bigChild - 1)))
            bigChild--;  //左孩子大于右孩子
        *(first + hole) = *(first + bigChild);
        hole = bigChild;
        bigChild = 2 * hole + 2;
    }
    if (len % 2 == 0 &&
        bigChild == len) {  //特殊情况：最后一个节点为左节点，没有兄弟节点
        *(first + hole) = *(first + bigChild - 1);
        hole = bigChild - 1;
    }
    /* 2.转化为向上调整问题 */
    siftFix(first, 0, hole, value, comp);
}
```

◆ 入堆

将*(end-1)调整到合适的位置，使得[start,end)变成堆

```
template <class Iter>
void pushHeap(Iter start, Iter end)
{
    typedef typename Iter::value_type value_type;
    siftFix(start, 0, end - start - 1, *(end - 1), Less<value_type>());
}
```

◆ **出堆**

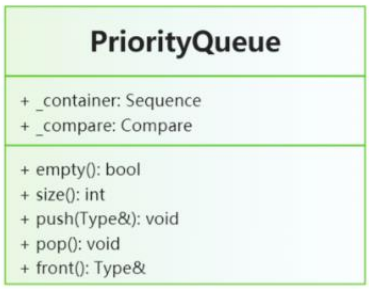
将*(start)交换到*(end-1)处(以便删去), [start,end-1)变成堆

```
template <class Iter>
void popHeap(Iter start, Iter end)
{
    typedef typename Iter::value_type value_type;
    value_type value = *(end - 1);  //将原来的末尾值保存起来
    *(end - 1) = *(start);  //将*(start)移动到*(end-1),*(start)处出现
hole
    sinkFix(start, end - start - 1, 0, value, Less<value_type>());
}
```

2.1.5 优先级队列类（PriorityQueue）

普通的队列是一种先进先出的数据结构，元素在队列尾追加，而从队列头删除。在优先级队列（Priority Queue）中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。优先级队列具有最高级先出（First In, Largest Out）的行为特征。

本项目中的优先级队列类为外部提供了底层数据结构，同时默认使用向量作为数据结构，UML 图如下所示：



优先级队列中主要操作函数如下所示：

◆ `inline bool empty()const`

判断优先级队列是否为空，也即内部存储结构是否为空。

◆ `inline int size()const`

返回优先级队列中元素的个数。

◆ `void push(const Type& i)`

在优先级队列加入一个数据。

◆ `void pop()`

删除优先级队列中优先级最高的元素。

◆ `const Type& front()const`

获取优先级队列中优先级最高的元素。

2.2 算法设计

通过查阅相关资料，我们可以了解到迷宫生成算法和迷宫寻路算法都主要有三种：迷宫生成算法有**随机 Prim 算法**，**递归回溯法**，**递归分割法**；迷宫寻路算法有 **DFS 算法**，**BFS 算法**和 **A-star 算法**。各种算法的优势如下表所示：

算法类型	算法名称	特点
迷宫生成算法	随机 Prim 法	随机性好，难度较大
	递归回溯法	主路特征明显
	递归分割法	生成的迷宫整体方正
迷宫寻路算法	DFS 算法	随机搜索
	BFS 算法	随机搜索
	A-Star 算法	效率较高

本项目中依次实现了三种迷宫生成算法和 DFS 算法，并且分别利用随机数对生成的迷宫进行了测试。此外，还将迷宫内容可视化了出来，整体项目比较完善。

2.3 迷宫生成算法

2.3.1 随机 Prim 算法

随机 Prim 算法的基本思想是将整个迷宫视为一张图，生成迷宫就是在此图上生成一棵（最小）生成树，我们使用 Prim 算法，将整个迷宫图划分成已加入最小生成树的部分和未加入最小生成树的部分，每次迭代在这两部分之间随机选择一处路径打通，直至整个迷宫都加入最小生成树。

2.3.2 递归回溯法

递归回溯算法的基本思想与 DFS 寻路算法类似，只是找路的过程变成了打通路径的过程，当在一条路径上无法再前进时，从栈中弹出上一个分支路口，继续打通路径。直至栈空，算法结束。

2.3.3 递归分割法

递归分割算法的思想基于分治思想：首先在迷宫中间随机取一点将迷宫分为左上、左下、右上、右下四个部分，相邻部分用墙相隔，一共得到四面墙。然后在这四面墙中随机取三面各打通一个路口，这样这四部分就一定能够彼此连通。再对四个小区域重复以上过程，直至每个小区域都足够小。

2.4 迷宫寻路算法（DFS 算法）

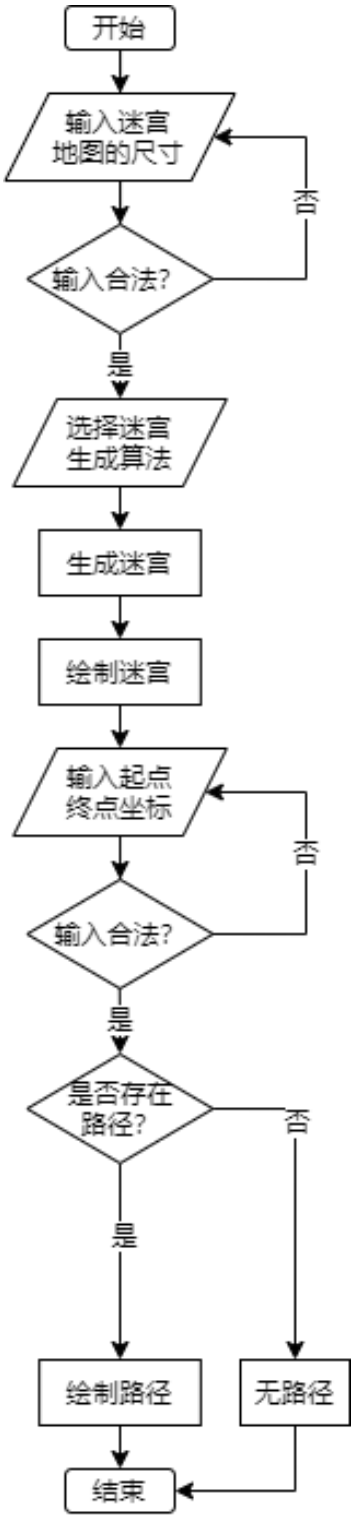
深度优先搜索算法（DFS）的思想即回溯法（Backtrack），即在一定的约束条件下试探地搜索前进，若前进中受阻，则及时回头纠正错误另择通路继续搜索的方法。从入口出发，按某一方向向前探索，若能走通，即某处可达，则到达新点，否则探索下一个方向；若所有的方向均没有通路，则沿原路返回前一点，换下一个方向再继续试探，直到所有可能的道路都探索到，或找到一条通路，或无路可走又返回入口点。

在求解过程中，为了保证在达到某一个点搜索失败时，能正确返回前一个点以便向下一个方向试探，则需要在试探过程中保存所能够达到的每个点的下标以及该点前进的方向，当找到出口时试探过程就结束了。

3 项目实现

3.1 程序整体功能的实现

3.1.1 程序整体功能流程图



3.1.2 程序整体功能代码

为了实现程序整体的功能，本项目在主函数中利用分支语句完成上述功能：

```
int main()
{
    //随机生成种子
    srand(time(NULL));
    char ch;

    while (true)
    {
        cout << "请输入地图的行数和列数，以空格分隔: ";
        cin >> mapX >> mapY;
        if (cin.fail())
            printf("输入有误\n");
        else if (mapX <= 0)
            printf("行数有误\n");
        else if (mapY <= 0)
            printf("列数有误\n");
        else if (mapX > 100)
            printf("行数过大\n");
        else if (mapY > 100)
            printf("列数过大\n");
        else
            break;
        cin.clear();             //重置错误标志位
        cin.ignore(4096, '\n');  //跳过后续的输入
    }

    printf("迷宫大小: %d x %d\n\n", mapX, mapY);
    blockMap = new BLOCK_TYPE *[mapX];
    for (int i = 0; i < mapX; i++)
        blockMap[i] = new BLOCK_TYPE[mapY];

    cout << "迷宫生成方式: " << endl;
    cout << "0. 随机 Prim 算法 (默认)" << endl;
    cout << "1. 递归回溯法" << endl;
    cout << "2. 递归分割法" << endl;
    cout << "请选择迷宫生成方式: ";
    cin >> ch;
    switch (ch)
    {
    case '1':
```

```
            recursiveBacktrack(Position(0, 0));
            break;
        case '2':
            recursiveDivision();
            break;
        default:
            randomizedPrim(Position(0, 0));
            break;
        }
        cout << endl;
        printMap();
        Position start, end;
        while (true)
        {
            cout << "请输入起点坐标和终点坐标(行 列): ";
            cin >> start.first >> start.second >> end.first >> end.second;
            if (cin.fail())
                printf("输入有误\n");
            else if (isUnreachable(start))
                printf("起点有误, 请重新输入\n");
            else if (isUnreachable(end))
                printf("终点有误, 请重新输入\n");
            else
                break;
            cin.clear();
            cin.ignore(4096, '\n');
        }
        printf("迷宫起点为(%d, %d), 终点为
(%d, %d)\n\n", start.first, start.second,
            end.first, end.second);

        Vector<Position> route;
        cout << "开始求解..." << endl;
        dfs(route, start, end);

        //打印结果
        if (route.empty())
            cout << "找不到路径,请检查起点和终点坐标" << endl;
        else
        {
            cout << endl;
            printMap(&route);
            cout << endl;
            //打印路径
```

```
        for (int i = 0; i < route.size(); i++)
        {
            if (i != 0) cout << " ---> ";
            printf("(%d, %d)", route[i].first, route[i].second);
        }
    }

    cin.ignore(4096, '\n');
    cout << endl << "按回车键退出程序" << endl;
    cin >> ch;
    return 0;
}
```

3.2 迷宫生成算法

3.2.1 随机 Prim 算法

```
void randomizedPrim(Position seed)
{
    Vector<Position> stack;
    stack.push_back(seed);

    initMap();
    while (!stack.empty())
    {
        int randd = rand() % stack.size();
        Position current = stack[randd];
        stack.erase(stack.begin() + randd);

        int road_count = 0;
        for (int i = 0; i < 4; i++)
        {
            Position nextPlace = current + step[i];
            if (isRoad(nextPlace))
                road_count++;
        }
        if (road_count <= 1)
        {
            blockMap[current.first][current.second] = BLANK;
            for (int i = 0; i < 4; i++)
            {
                Position nextPlace = current + step[i];
                if (isWall(nextPlace))
                    stack.push_back(nextPlace);
```

```
            }
        }
    }
}
```

3.2.2 递归回溯法

```
void recursiveBacktrack(Position seed)
{
    Vector<Position> stack;
    stack.push_back(seed);

    initMap();
    while (!stack.empty())
    {
        Position current = *stack.last(); stack.pop_back();

        int road_count = 0;
        for (int i = 0; i < 4; i++)
        {
            Position nextPlace = current + step[i];
            if (isRoad(nextPlace))
                road_count++;
        }
        if (road_count <= 1)
        {
            blockMap[current.first][current.second] = BLANK;
            Vector<Position> nextPlaces;
            for (int i = 0; i < 4; i++)
            {
                Position nextPlace = current + step[i];
                if (isWall(nextPlace))
                    nextPlaces.push_back(nextPlace);
            }
            mySTL::random_shuffle(nextPlaces.begin(), nextPlaces.end())
;
            for (int i = 0; i < nextPlaces.size(); i++)
                stack.push_back(nextPlaces[i]);
        }
    }
}
```

3.2.3 递归分割法

```
void recursiveDivision(Position leftup, Position rightdown)
{
    if (rightdown.first - leftup.first < 3 ||
        rightdown.second - leftup.second < 3)
        return;
    //分割点
    Position divisionPoint(RAND_ODD_RANGE(leftup.first, rightdown.first
),
        RAND_ODD_RANGE(leftup.second, rightdown.second));

    for (int i = leftup.first; i < rightdown.first; i++)
        blockMap[i][divisionPoint.second] = WALL;
    for (int i = leftup.second; i < rightdown.second; i++)
        blockMap[divisionPoint.first][i] = WALL;

    int randomNumber = rand() % 4;
    if (randomNumber != 0)
    {
        blockMap[RAND_EVEN_RANGE(leftup.first, divisionPoint.first)][di
visionPoint.second] =
            BLANK;
    }
    if (randomNumber != 1)
    {
        blockMap[divisionPoint.first]
            [RAND_EVEN_RANGE(divisionPoint.second + 1, rightdown.second
)] = BLANK;
    }
    if (randomNumber != 2)
    {
        blockMap[(RAND_EVEN_RANGE(divisionPoint.first + 1, rightdown.fi
rst))]
            [divisionPoint.second] = BLANK;
    }
    if (randomNumber != 3)
    {
        blockMap[divisionPoint.first][RAND_EVEN_RANGE(leftup.second, di
visionPoint.second)] =
            BLANK;
    }

    //递归
```

```
    recursiveDivision(leftup, divisionPoint);
    recursiveDivision(Position(leftup.first, divisionPoint.second + 1),
        Position(divisionPoint.first, rightdown.second));
    recursiveDivision(Position(divisionPoint.first + 1, leftup.second),
        Position(rightdown.first, divisionPoint.second));
    recursiveDivision(
        Position(divisionPoint.first + 1, divisionPoint.second + 1), ri
ghtdown);
}

void recursiveDivision()
{
    initMap(BLANK);
    for (int i = 0; i < mapX; i++)
        blockMap[i][mapY - 1] = WALL;
    for (int j = 0; j < mapY; j++)
        blockMap[mapX - 1][j] = WALL;
    Position rightdown(TO_ODD(mapX), TO_ODD(mapY));
    recursiveDivision(Position(0, 0), rightdown);
}
```

3.3 迷宫寻路算法（DFS 算法）

```
bool dfs(Vector<Position> &path, Position start, Position end)
{
    bool **visited = new bool*[mapX];
    for (int i = 0; i < mapX; i++)
    {
        visited[i] = new bool[mapY];
        memset(visited[i], 0, mapY * sizeof(bool));
    }

    path.clear();
    path.push_back(start);  //用参数path作栈
    visited[start.first][start.second] = true;

    Position current;
    while (!path.empty())
    {
        current = *path.last();
        if (current == end)
            break;
        bool no_way = true;
        for (int i = 0; i < 4; i++)
```

```
        {
            Position next = current + step[i];
            if (!isUnreachable(next) && !visited[next.first][next.secon
d])
            {
                path.push_back(next);
                visited[next.first][next.second] = true;
                no_way = false;
                break;
            }
        }
        if (no_way)
            path.pop_back();
    }

    for (int i = 0; i < mapX; i++)
        delete[] visited[i];
    delete[] visited;

    if (path.empty())
        return false;
    return true;
}
```

4 项目测试

4.1 随机 Prim 算法测试

测试用例：10 10 0

预期输出：10X10 的随机迷宫

实验结果：

4.2 递归回溯法测试

测试用例：10 10 1

预期输出：10X10 的随机迷宫

测试结果：

4.3 递归分割法测试

测试用例：10 10 2

预期输出：10X10 的随机魔攻

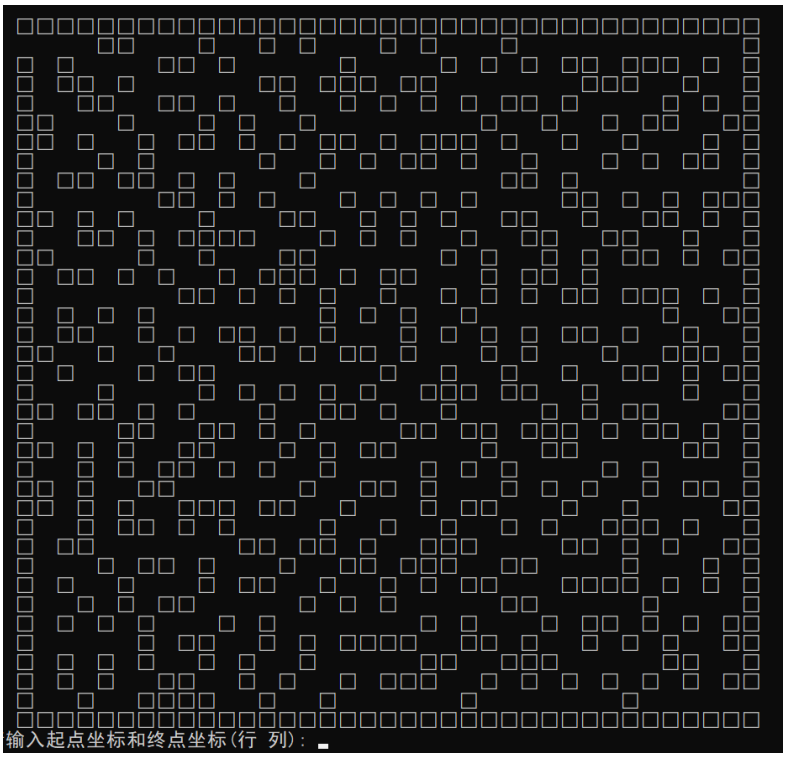
实验结果：

4.4 大型迷宫生成测试

输入用例：35 35 0

预期输出：35X35 的随机迷宫

实验结果：

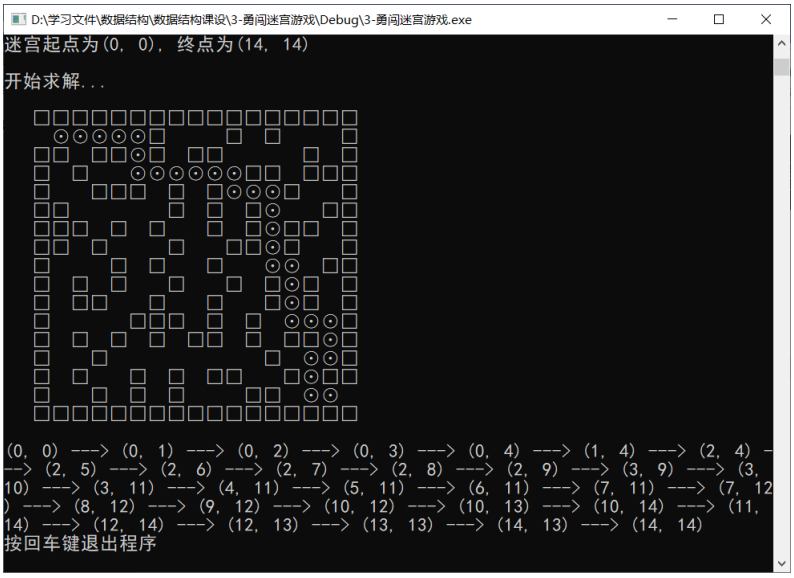


4.5 小型迷宫寻路测试

输入用例：15 15 0 0 0 14 14

预期输出：15X15 的迷宫中从（0,0）到（14,14）的路径

实验结果：

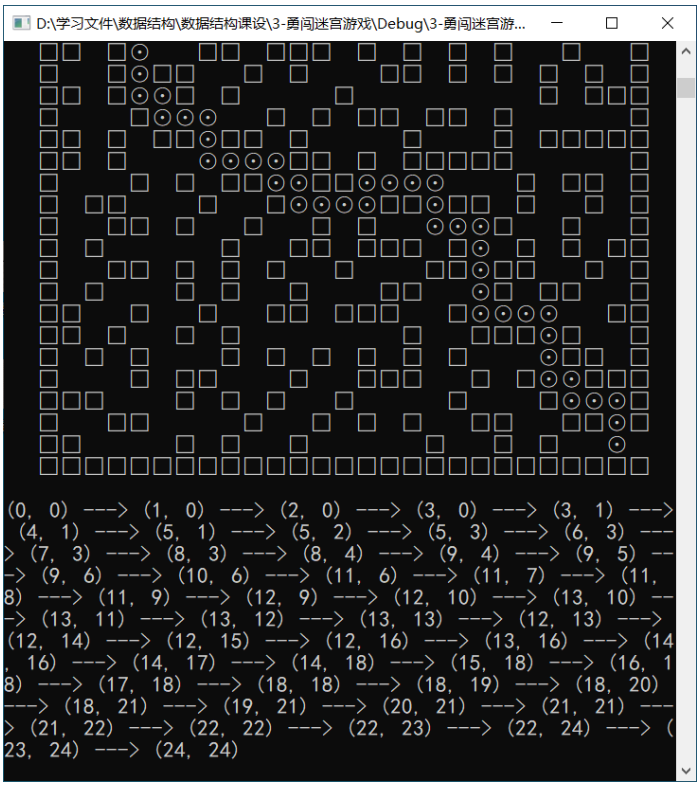


4.6 大型迷宫寻路测试

输入用例：25 25 0 0 0 24 24

预期结果：25X25 的迷宫中从（0,0）到（24,24）的路径

实验结果：

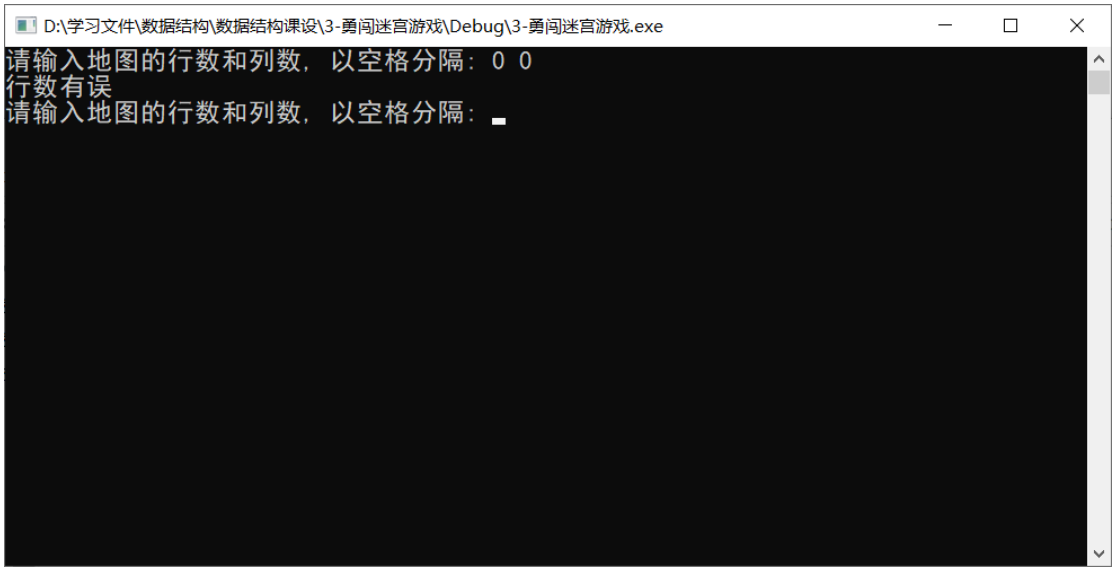


4.7 非法数据测试

测试用例：0 0

预期输出：输入不正确

实验结果：



4.8 边界情况测试

测试用例：1 1 0 0 0 0 0

预期输出：只有一个路径的输出

实验结果：