C/C++从入门到精通-高级程序员之路

【 数据结构 】

图及其企业级应用

第 1 节 图的故事导入

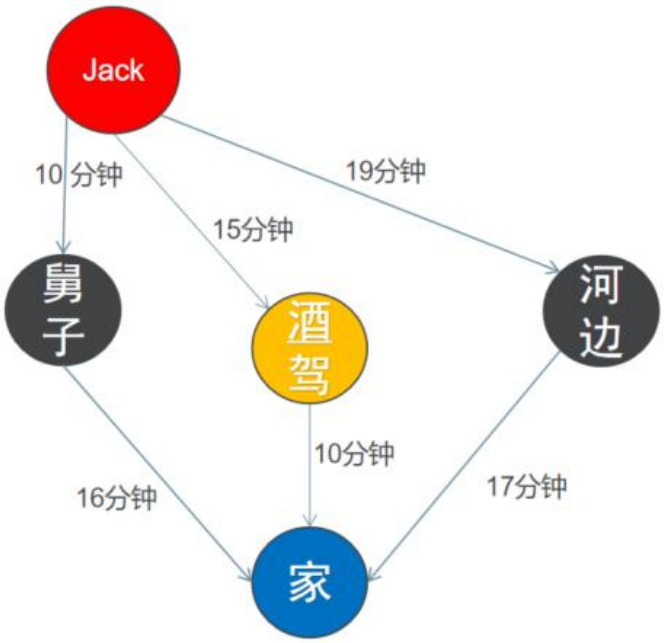
故事情节

Jack 自从买车后，交通出行方便了，心自然就野了！身边的各种朋友自然就多了起来！

有一天晚上，一个年轻漂亮的女同事生日，Jack 受邀请准时爽约！Jack 亲睐此女已久，只是介于家里的 LD 不敢越雷池一步，但是，一有机会，Jack 都会和此女接近，经常在一起，所以可以说是就当个哥们吧！

Jack 当晚开心的和女同事喝酒、聊天，忘记了家的存在，不知不觉时间一下子指向了 10 点！此时，正沉浸在幻想中的 Jack 被老婆电话惊醒！不用接都知道，老婆的“圣旨”又来了！

这下麻烦来了，老婆电话一到，必须半小时内到家，否则，皮都可能被母老虎拔了，但现在自己渴了酒，叫代驾又需要很久的时间，这一想, Jack 觉得自己麻烦来了！



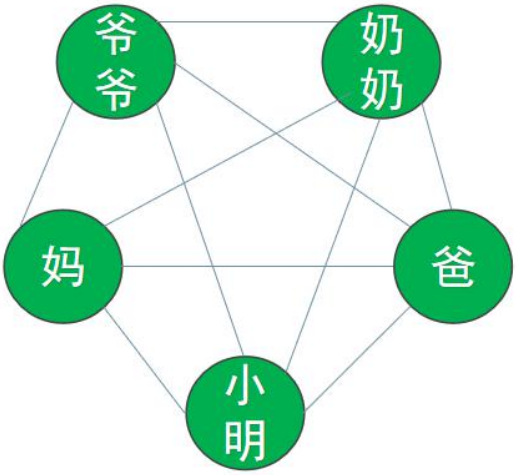
<Jack 回家的三条路线>

做为程序员，Jack 肯定会选择第 3 条，那么我们如何用程序来做选择呢？

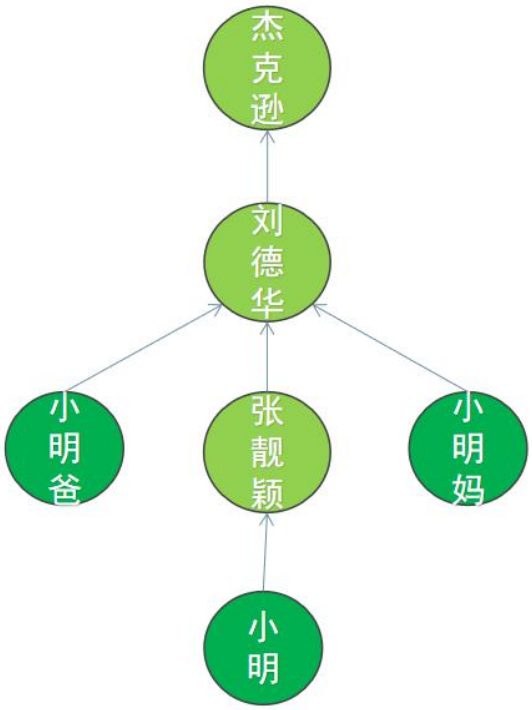
第 2 节 图的原理精讲

在计算机科学中，一个图就是一些顶点的集合，这些顶点通过一系列边结对（连接）。顶点用圆圈表示，边就是这些圆圈之间的连线。顶点之间通过边连接。注意：顶点有时也称为节点或者交点，边有时也称为链接。

社交网络，每一个人就是一个顶点，互相认识的人之间通过边联系在一起，边表示彼此的关系。这种关系可以是单向的，也可以是双向的！



（家庭关系）

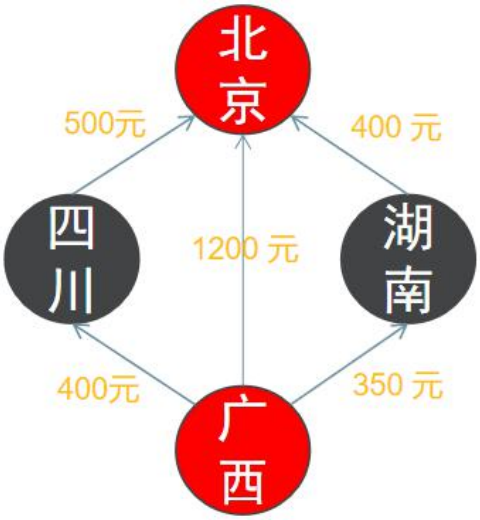


（粉丝关系）

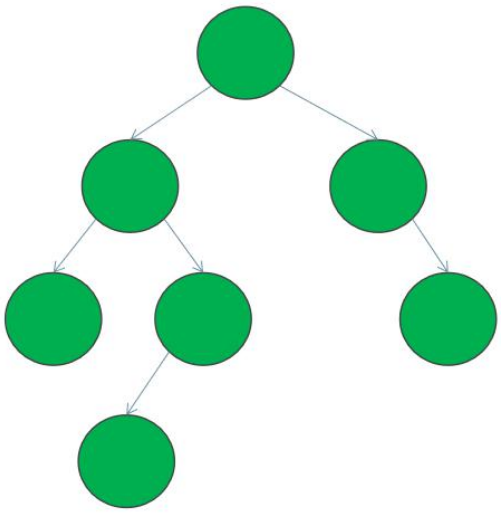
同时，边可以是双向的，也可以是单向的！

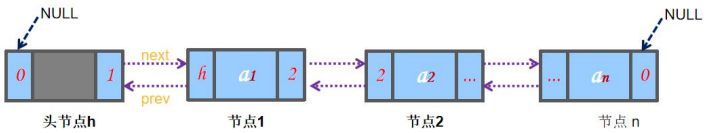
地图导航 - 起点、终点和分叉口（包括十字路口·、T 字路口等）都是顶点，导航经过的两顶点的路径就是边！

如上图所示，我们导航从一个点到另外一个点可以有条路径，路径不同，路况就不同，拥堵程度不同，所以导致不同路径所花的时间也不一样，这种不同我们可以使用边的**权重**来表示，即根据每条边的实际情况给每一条边分配一个正数或者负数值。考虑如下机票图，各个城市就是顶点，航线就是边。那么权重就是机票价格。



我们前面讲解的树和链表都是图的特例！



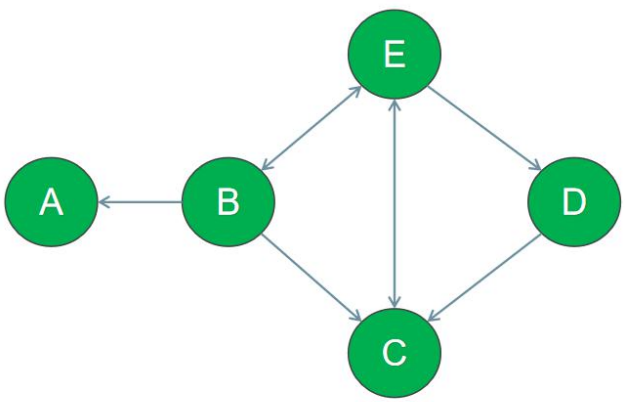


如果我们有一个编程问题可以通过顶点和边表示，那么我们就可以将你的问题用图画出来，然后使用相应的图算法来找到解决方案。

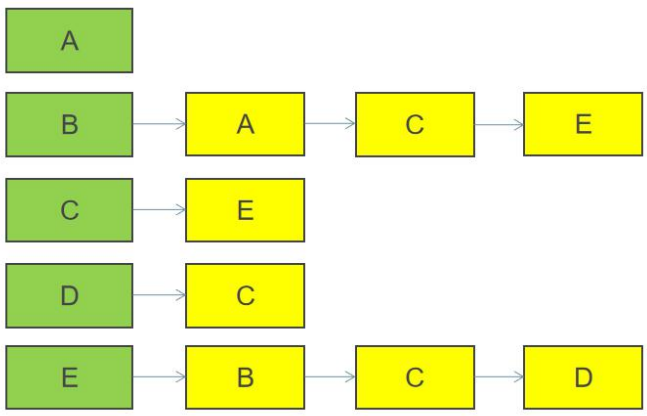
图的表示

邻接列表

在邻接列表实现中，每一个顶点会存储一个从它这里开始的相邻边的列表。比如，如果顶点 B 有一条边到 A、C 和 E，那么 A 的列表中会有 3 条边



（图）



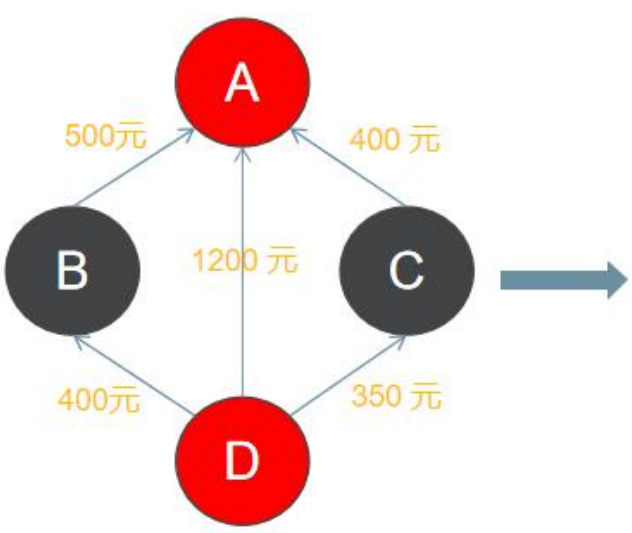
（邻接列表）

邻接列表只描述指向外部的边。B 有一条边到 A，但是 A 没有边到 B，所以 B 没有出现在 A 的邻接列表中。

查找两个顶点之间的边或者权重会比较费时，因为遍历邻接列表直到找到为止。

邻接矩阵

由二维数组对应的行和列都表示顶点，由两个顶点所决定的矩阵对应元素数值表示这里两个顶点是否相连（如，0 表示不相连，非 0 表示相连和权值）、如果相连这个值表示的是相连边的权重。例如，广西到北京的机票，我们用邻接矩阵表示：



图

	A	B	C	D
A	0			
B	500	0		
C	400		0	
D	1200	400	350	0

邻接矩阵

往这个图中添加顶点的成本非常昂贵，因为新的矩阵结果必须重新按照新的行/列创建，然后将已有的数据复制到新的矩阵中。

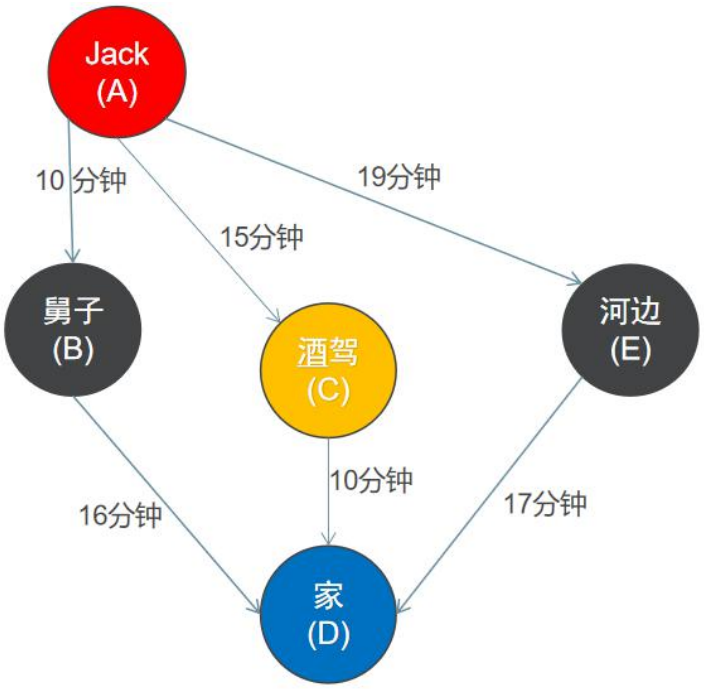
所以使用哪一个呢？我们先来看看下表

操 作	邻接列表	邻接矩阵
存储空间	O(V + E)	O(V ^ 2)
添加顶点	O(1)	O(V ^ 2)
添加边	O(1)	O(1)
检查相邻性	O(V)	O(1)

注意: ***V*表示图中顶点的个数，*E*表示边的个数。**

结论：大多数时候，选择邻接列表是正确的。（*在图比较稀疏的情况下，每一个顶点都只会和少数几个顶点相连，这种情况下邻接列表是最佳选择。如果这个图比较密集，每一个顶点都和大多数其他顶点相连，那么邻接矩阵更合适。*）

第 3 节　图的算法实现



邻接表结构的定义

```
#define MaxSize 1024

typedef struct _EdgeNode {//与节点连接的边的定义
    int adjvex;  //邻接的顶点
    int weight;  //权重
    struct _EdgeNode *next; //下一条边
}EdgeNode;

typedef struct _VertexNode{//顶点节点
    char data;  //节点数据
    struct _EdgeNode *first;//指向邻接第一条边
}VertexNode, AdjList;

typedef struct _AdjListGraph {
    AdjList *adjlist;
    int vex;  //顶点数
    int edge; //边数
}AdjListGraph;
```

邻接表的初始化

```
/*图的初始化*/
void Init(AdjListGraph &G){
    G.adjlist = new AdjList[MaxSize];
    G.edge = 0;
    G.vex = 0;
}
```

邻接表的创建

```
/*图的创建*/
void Create(AdjListGraph &G){
    cout << "请输入该图的顶点数以及边数：" << endl;
    cin>> G.vex >> G.edge;

    cout<<"请输入相关顶点："<< endl;
    for(int i=0; i<G.vex; i++){
        cin >> G.adjlist[i].data;
        G.adjlist[i].first = NULL;
    }

    char v1=0, v2=0;//保存输入的顶点的字符
    int i1, i2;    //保存顶点在数组中的下标

    cout<<"请输入想关联边的顶点："<< endl;
    for(int i=0; i<G.edge; i++){
        cin >>v1 >>v2;
        i1 = Location(G, v1);
        i2 = Location(G, v2);

        if(i1!=-1 && i2!=-1){//寻找到位置
            EdgeNode *temp = new EdgeNode;
            temp->adjvex = i2;
            temp->next = G.adjlist[i1].first;
            G.adjlist[i1].first = temp;
        }

    }

}

/*通过顶点对应的字符寻找顶点在图中的邻接点*/
int  Location(AdjListGraph &G, char c){
```

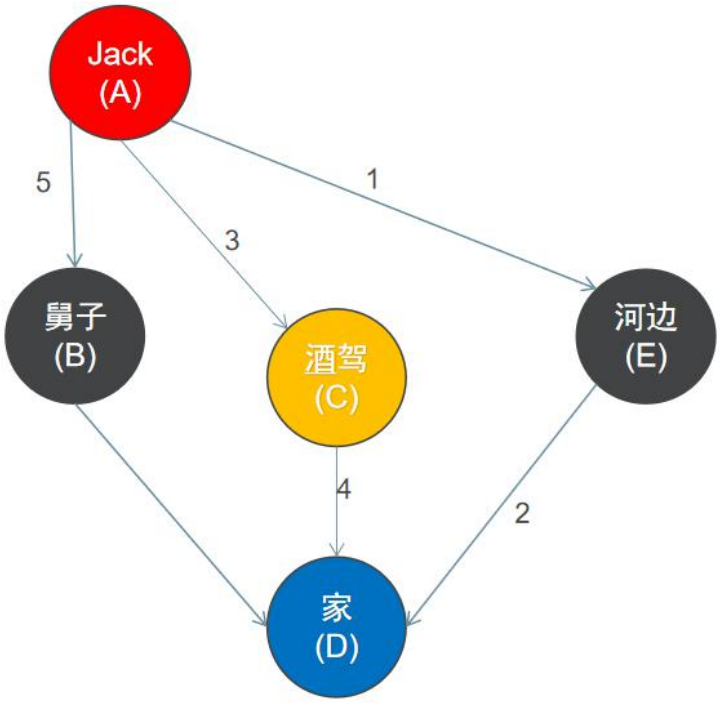
```
    for(int i=0; i<G.vex; i++){
        if(G.adjlist[i].data == c) {
            return i;
        }
    }

    return -1;
}
```

QQ 交流群：875300321

邻接表的深度遍历

深度优先遍历思想：

- 首先以一个未被访问过的顶点作为起始顶点，沿当前顶点的边走到未访问过的顶点；
- 当没有未访问过的顶点时，则回到上一个顶点，继续试探别的顶点，直到所有的顶点都被访问过。



使用深度优先搜索来遍历这个图的具体过程是：

1. 首先从一个未走到过的顶点作为起始顶点，比如 **A** 顶点作为起点。
2. 沿 **A** 顶点的边去尝试访问其它未走到过的顶点，首先发现 E 号顶点还没有走到过，于是访问 **E** 顶点。
3. 再以 **E** 顶点作为出发点继续尝试访问其它未走到过的顶点，接下来访问 **D** 顶点。
4. 再尝试以 **D** 顶点作为出发点继续尝试访问其它未走到过的顶点。
5. 但是，此时沿 **D** 顶点的边，已经不能访问到其它未走到过的顶点，接下来返回到 **E** 顶点。
6. 返回到 **E** 顶点后，发现沿 **E** 顶点的边也不能再访问到其它未走到过的顶点。此时又回到顶点 **A**（**D**->**E**->**A**），再以 **A** 顶点作为出发点继续访问其它未走到过的顶点，于是接下来访问 **C** 顶点。
7. 。。。。。。。。。。
8. 最终访问的结果是 **A** -> **E** -> **D** -> **C** -> **B**

```
#include <iostream>

using namespace std;

#define MaxSize 1024

typedef struct _EdgeNode {//与节点连接的边的定义
    int adjvex;  //邻接的顶点
    int weight;  //权重
    struct _EdgeNode *next; //下一条边
}EdgeNode;

typedef struct _VertexNode{//顶点节点
    char data;  //节点数据
    struct _EdgeNode *first;//指向邻接第一条边
}VertexNode, AdjList;

typedef struct _AdjListGraph {
    AdjList *adjlist;
    int vex;  //顶点数
    int edge; //边数
}AdjListGraph;

bool visited[MaxSize];  //全局数组，用来记录节点是否已被访问

int  Location(AdjListGraph &G, char c);

/*图的初始化*/
void Init(AdjListGraph &G){
    G.adjlist = new AdjList[MaxSize];
    G.edge = 0;
    G.vex = 0;

    for (int i = 0; i < MaxSize; i++) {
        visited[i] = false;
    }
}

/*图的创建*/
void Create(AdjListGraph &G){
    cout << "请输入该图的顶点数以及边数：" << endl;
    cin>> G.vex >> G.edge;

    cout<<"请输入相关顶点："<< endl;
    for(int i=0; i<G.vex; i++){
```

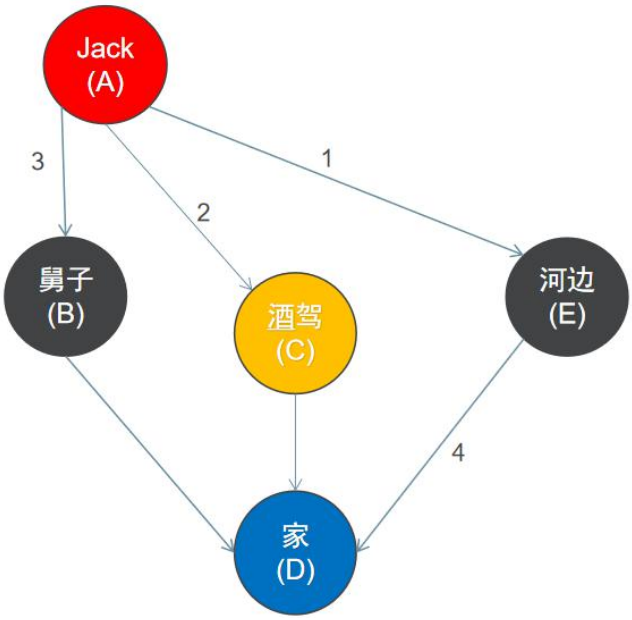
```
        cin >> G.adjlist[i].data;
        G.adjlist[i].first = NULL;
    }

    char v1=0, v2=0;//保存输入的顶点的字符
    int i1, i2;    //保存顶点在数组中的下标

    cout<<"请输入想关联边的顶点："<< endl;
    for(int i=0; i<G.edge; i++){
        cin >>v1 >>v2;
        i1 = Location(G, v1);
        i2 = Location(G, v2);

        if(i1!=-1 && i2!=-1){//寻找到位置
            EdgeNode *temp = new EdgeNode;
            temp->adjvex = i2;
            temp->next = G.adjlist[i1].first;
            G.adjlist[i1].first = temp;
        }

    }

}

/*通过顶点对应的字符寻找顶点在图中的邻接点*/
int  Location(AdjListGraph &G, char c){
    for(int i=0; i<G.vex; i++){
        if(G.adjlist[i].data == c) {
            return i;
        }
    }

    return -1;
}

/*对图上的顶点进行深度遍历*/
void DFS(AdjListGraph &G,int v){
    int next = -1;

    if(visited[v]) return;

    cout<<G.adjlist[v].data<<" ";
    visited[v] = true; //设置为已访问

    EdgeNode *temp = G.adjlist[v].first;

    while(temp){
```

```
        next = temp->adjvex;
        temp = temp->next;
        if(visited[next] == false){
            DFS(G, next);
        }
    }
}

/*对所有顶点进行深度遍历*/
void DFS_Main(AdjListGraph &G){
    for(int i=0; i<G.vex; i++){
        if(visited[i] == false){
            DFS(G, i);
        }
    }
}


int  main(){
    AdjListGraph G;

    //初始化
    Init(G);

    //创建图
    Create(G);

    //深度遍历
    DFS_Main(G);

    system("pause");
}
```

邻接表的广度遍历

广度优先遍历思想

- 首先以一个未被访问过的顶点作为起始顶点，访问其所有相邻的顶点;
- 然后对每个相邻的顶点，再访问它们相邻的未被访问过的顶点，直到所有顶点都被访问过，遍历结束。

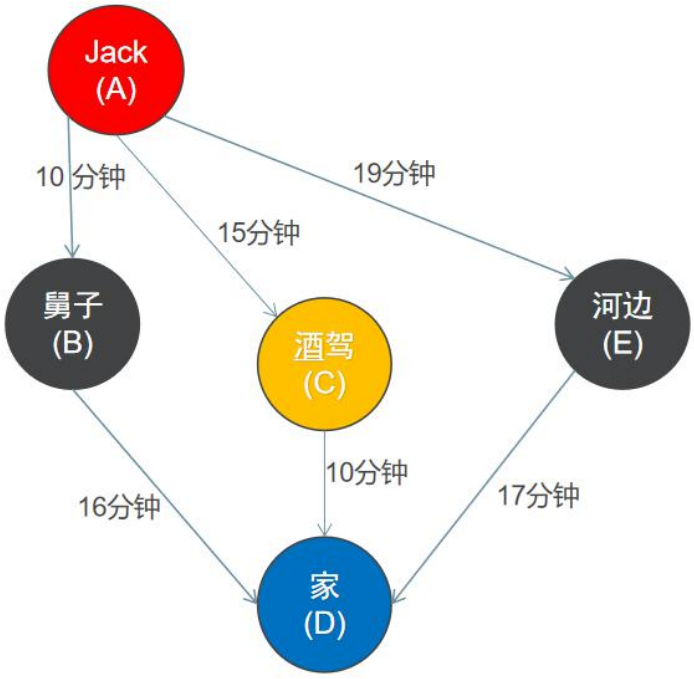



```
/*对图上的顶点进行广度遍历*/
void BFS(AdjListGraph &G,int v) {
    queue <int> q;
    int cur;
    int next;

    q.push(v);

    while (!q.empty()) {  //队列非空
        cur = q.front();  //取队头元素
        if (visited[cur] == false) {  //当前顶点还没有被访问
            cout << G.adjlist[cur].data << " ";
            visited[cur] = true;  //设置为已访问
        }
        q.pop();  //出队列

        EdgeNode *tp = G.adjlist[cur].first;
        while (tp != NULL) {
            next = tp->adjvex;
            tp = tp->next;
            q.push(next);  //将第一个邻接点入队
        }
```

```
    }
}

 /*对所有顶点进行广度遍历*/
void BFS_Main(AdjListGraph &G) {
    for (int i = 0; i < G.vex; i++) {
        if (visited[i] == false) {
            BFS(G,i);
        }
    }
}
```

图的导航-最短路径算法

从起点开始访问所有路径，则到达终点节点的路径有多条，其中路径权值最短的一条则为最短路径。最短路径算法有深度优先遍历、广度优先遍历、Bellman-Ford 算法、弗洛伊德算法、 SPFA(Shortest Path Faster Algorithm)算法和迪杰斯特拉算法等。



源码实现

```
#include <iostream>

using namespace std;

#define MaxSize 1024

typedef struct _EdgeNode {//与节点连接的边的定义
    int adjvex;  //邻接的顶点
    int weight;  //权重
    struct _EdgeNode *next; //下一条边
}EdgeNode;

typedef struct _VertexNode{//顶点节点
    char data;  //节点数据
    struct _EdgeNode *first;//指向邻接第一条边
}VertexNode, AdjList;

typedef struct _AdjListGraph {
    AdjList *adjlist;
    int vex;  //顶点数
    int edge; //边数
}AdjListGraph;
```

```
bool visited[MaxSize]={0};  //全局数组，用来记录节点是否已被访问

int  Location(AdjListGraph &G, char c);

/*图的初始化*/
void Init(AdjListGraph &G){
    G.adjlist = new AdjList[MaxSize];
    G.edge = 0;
    G.vex = 0;

    for (int i = 0; i < MaxSize; i++) {
        visited[i] = false;
    }
}

/*图的创建*/
void Create(AdjListGraph &G){
    cout << "请输入该图的顶点数以及边数：" << endl;
    cin>> G.vex >> G.edge;

    cout<<"请输入相关顶点："<< endl;
    for(int i=0; i<G.vex; i++){
        cin >> G.adjlist[i].data;
        G.adjlist[i].first = NULL;
    }

    char v1 = 0, v2 = 0;//保存输入的顶点的字符
    int i1, i2;    //保存顶点在数组中的下标
    int weight = 0;

    cout<<"请输入想关联边的顶点及权重："<< endl;
    for(int i=0; i<G.edge; i++){
        cin >>v1 >>v2 >>weight;
        i1 = Location(G, v1);
        i2 = Location(G, v2);


        if(i1!=-1 && i2!=-1){//寻找到位置
            EdgeNode *temp = new EdgeNode;
            temp->adjvex = i2;
            temp->next = G.adjlist[i1].first;
            temp->weight = weight;
            G.adjlist[i1].first = temp;
        }
```

```
    }

}

/*通过顶点对应的字符寻找顶点在图中的邻接点*/
int  Location(AdjListGraph &G, char c){
    for(int i=0; i<G.vex; i++){
        if(G.adjlist[i].data == c) {
            return i;
        }
    }

    return -1;
}

int min_weights = 0x7FFFFFFF; //最大的整数 2 的 32 次方-1
int steps = 0;
int path[MaxSize] = {0};//保存走过的路径
int shortest_path[MaxSize] = {0}; //保存最短的路径


/*对图上的顶点进行深度遍历*/
void DFS(AdjListGraph &G,int start, int end, int weights){
    int cur = -1;
    //if(visited[start]) return;

    cur = start;

    if(cur == end){//已找到终点，不用继续遍历
            //打印所经过的所有路径
        for (int i = 0; i < steps; i++)      /// 输出所有可能的路径
         {
            cout<<G.adjlist[path[i]].data<<" "; //输出路径
         }
         printf("\t\t 该路径对应的长度是：%d\n", weights);//输出对应路
径长度

         if (min_weights > weights)  //更新最小路径
         {
             min_weights = weights;
            memcpy(shortest_path, path, steps*sizeof(int));
         }
    }


    visited[start] = true; //设置为已访问
```

```
    EdgeNode *temp = G.adjlist[start].first;

    while(temp){
        int weight = temp->weight;
        cur = temp->adjvex;
        if(visited[cur] == false){
            visited[cur] = true;   //标记城市 i 已经在路径中
            path[steps++] = cur;//保存路径到 path 数组中
            DFS(G, cur, end, weights + weight);
            visited[cur] = false;// 之前一步探索完毕后,取消对城市 i 的
标记以便另一条路径选择顶点
            path[--steps] = 0;
        }
        temp = temp->next;
    }
}


int  main(){
    AdjListGraph G;

    //初始化
    Init(G);

    //创建图
    Create(G);

    char start, end;
    cout<<"请输入要查询的最短路径的起点和终点："<<endl;
    cin >>start >>end;


    //求两点间的最短路径
    DFS(G, Location(G, start), Location(G, end), 0);

    cout<<"最小路径长度为："<<min_weights<<endl;
    cout<<"路径：";
    int i=0;
    while(i<MaxSize && shortest_path[i]>0){
        cout<<G.adjlist[shortest_path[i]].data<<" ";
        i++;
    }
    cout<<endl;

    system("pause");
}
```

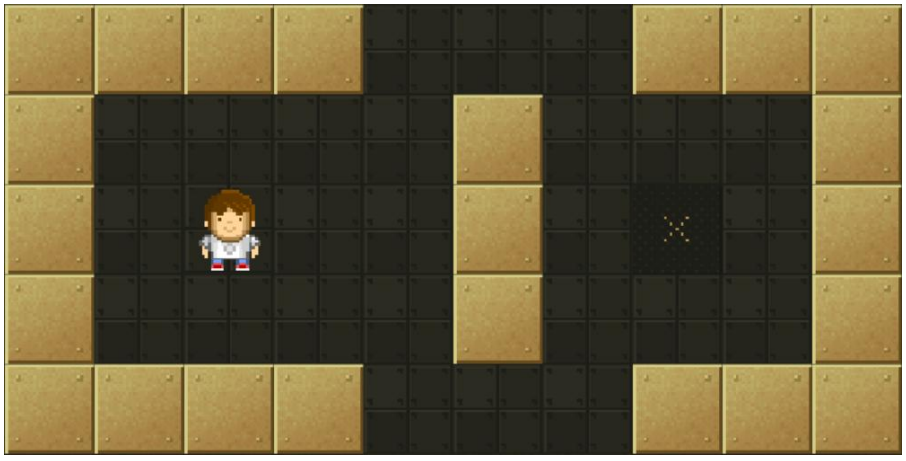
第 4 节 图的企业级应用案例

4.1 AI 游戏中的自动寻路-A*算法

随着 3D 游戏的日趋流行,在复杂的 3D 游戏环境中如何能使非玩家控制角色准确实现自动寻路功能成为了 3D 游戏开发技术中一大研究热点。其中 A*算法得到了大量的运用，A*算法较之传统的路径规划算法,实时性更高、灵活性更强,寻路结果更加接近人工选择的路径结果．A*寻路算法并不是找到最优路径，只是找到相对近的路径，因为找最优要把所有可行路径都找出来进行对比，消耗性能太大，寻路效果只要相对近路径就行了。

A* 算法的原理

我们假设在推箱子游戏中人要从站里的地方移动到右侧的箱子目的地，但是这两点之间被一堵墙隔开。

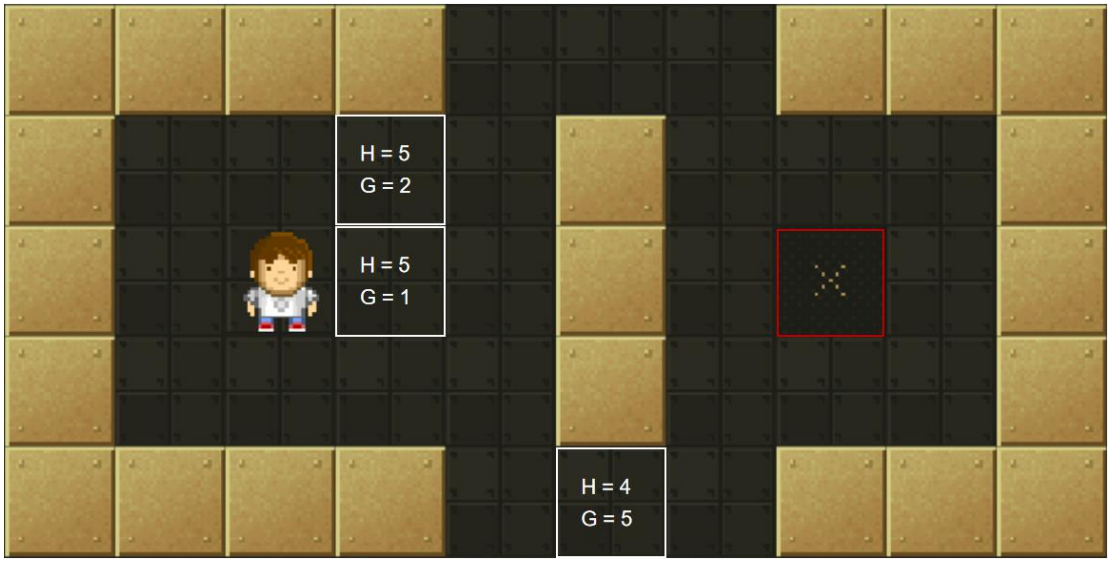


我们下一步要做的便是查找最短路径。既然是 AI 算法， A* 算法和人寻找路径的做法十分类似，当我们离目标较远时，我们的目标方向是朝向目的点直线移动，但是在近距离上因为各种障碍需要绕行（走弯路）！而且已走过的地方就无须再次尝试。


为了简化问题，我们把要搜寻的区域划分成了正方形的格子。这是寻路的第一步，简化搜索区域，就像推箱子游戏一样。这样就把我们的搜索区域简化为了 2 维数组。数组的每一项代表一个格子，它的状态就是可走（walkalbe）和不可走(unwalkable) 。通过计算出从起点到终点需要走过哪些方格，就找到了路径。一旦路径找到了，人物便从一个方格的中心移动到另一个方格的中心，直至到达目的地。

简化搜索区域以后，如何定义小人当前走要走的格子离终点是近是远呢？我们需要两个指标来表示：

- G 表示从起点移动到网格上指定方格的移动距离（暂时不考虑沿斜向移动，只考虑上下左右移动)。
- H 表示从指定的方格移动到终点的预计移动距离，只计算直线距离（H 有很多计算方法，这里我们设定只可以上下左右移动，即该点与终点的直线距离)。



令 F = G + H ，F 即表示从起点经过此点预计到终点的总移动距离

接下来我们从起点开始，按照以下寻路步骤，直至找到目标。

寻路步骤

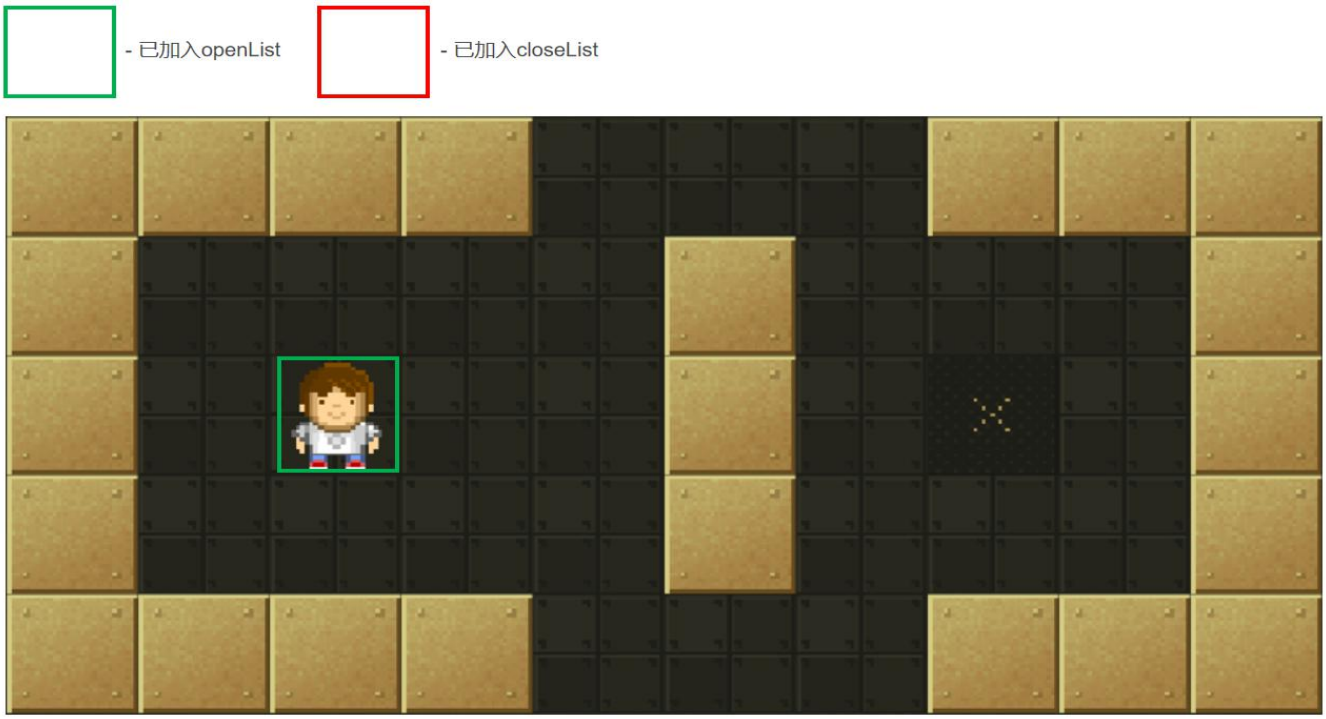
1. 从起点开始，把它作为待处理的方格存入一个预测可达的节点列表，简称 openList，即把起点放入“预测可达节点列表”，可达节点列表 openList 就是一个等待检查方格的列表。

2. 寻找 openList 中 F 值最小的点 min（一开始只有起点）周围可以到达的方格（可到达的意思是其不是障碍物，也不存在关闭列表中的方格,即不是已走过的方格）。计算 min 周围可到达的方格的 F 值。将还没在 openList 中点放入其中，并设置它们的"父方格"为点 min,表示他们的上一步是经过 min 到达的。如果 min 下一步某个可到达的方格已经在 openList 列表那么并且经 min 点它的 F 值更优,则修改 F 值并把其"父方格"设为点 min。

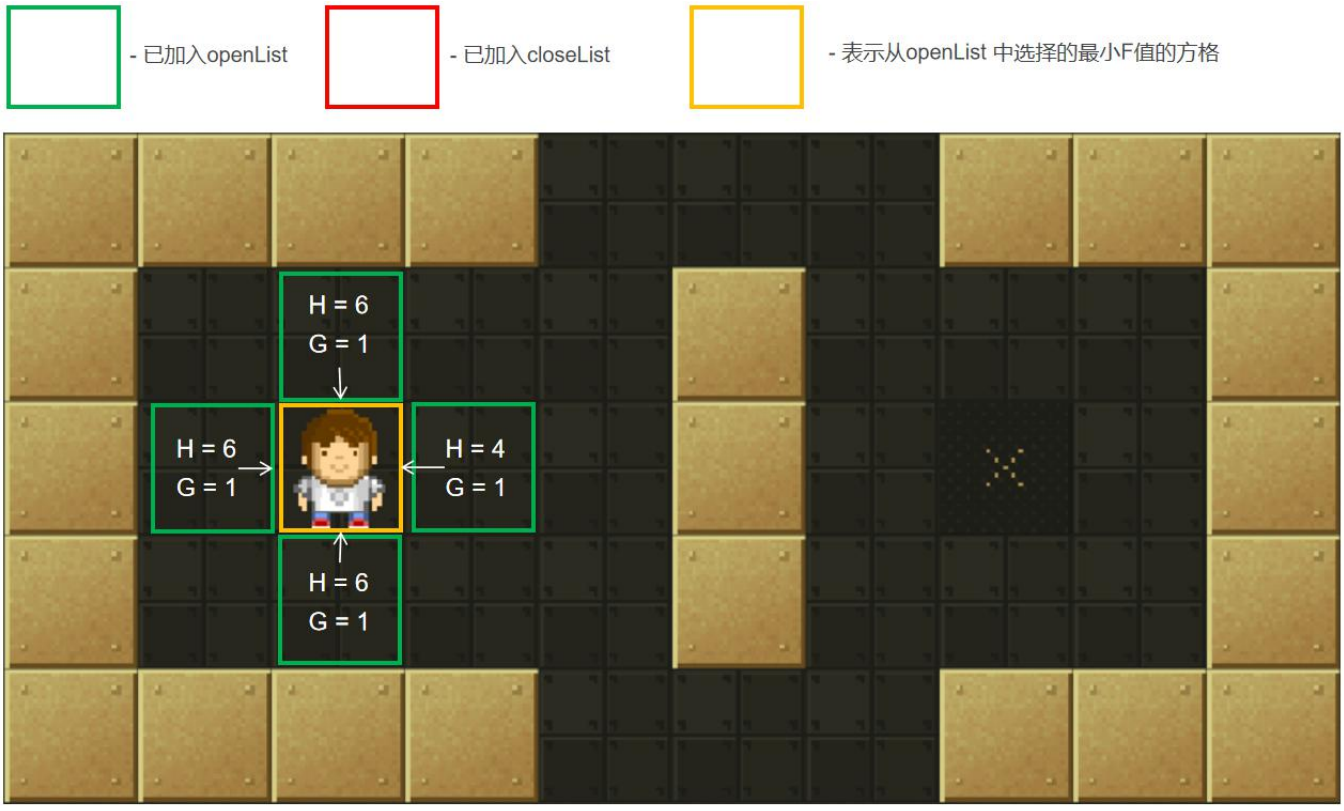
3. 把 2 中的点 min 从"开启列表"中删除并存入"关闭列表"closeList 中, closeList 中存放的都是不需要再次检查的方格。如果 2 中点 min 不是终点并且开启列表的数量大于零，那么继续从第 2 步开始。如果是终点执行第 4 步，如果 openList 列表数量为零，那么就是找不到有效路径。

4.如果第 3 步中 min 是终点，则结束查找，直接追溯父节点到起点的路径即为所选路径。

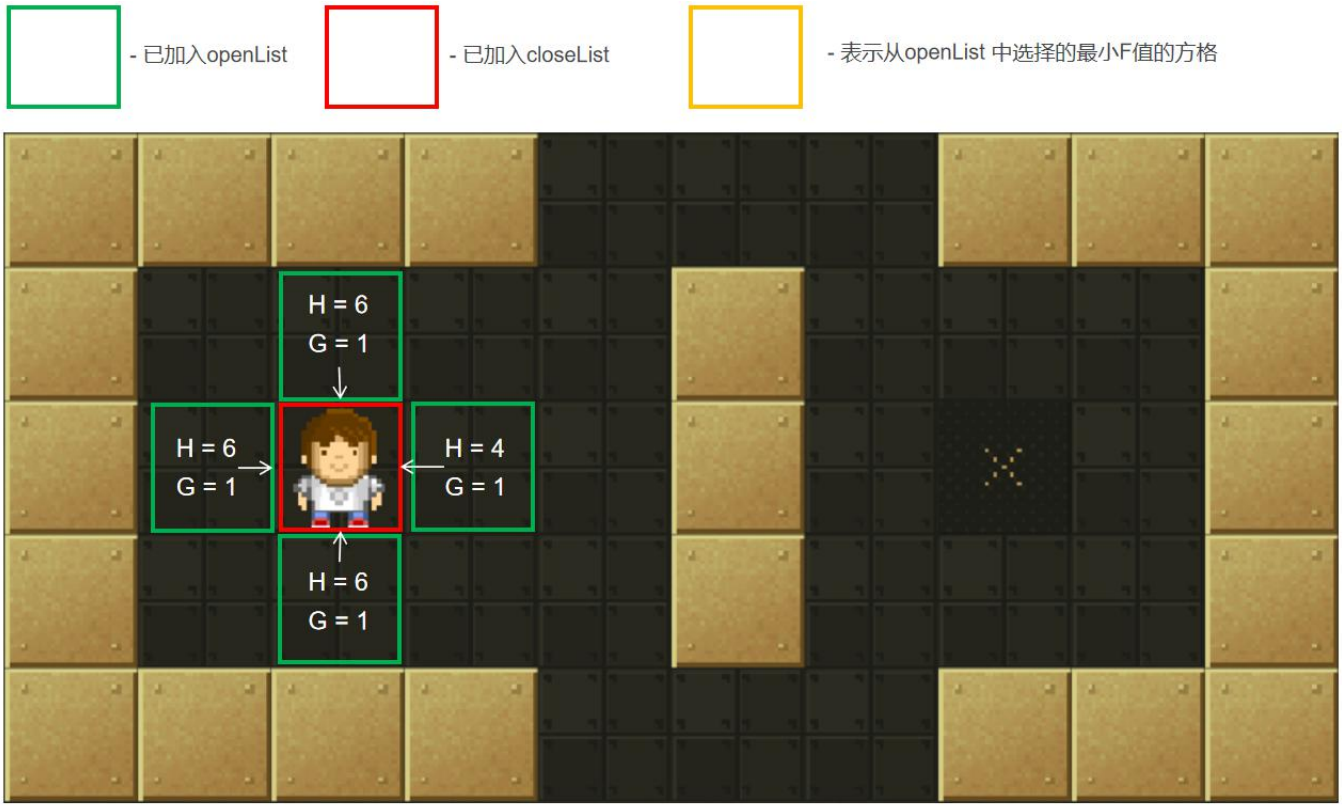
具体寻路步步骤如下所示：



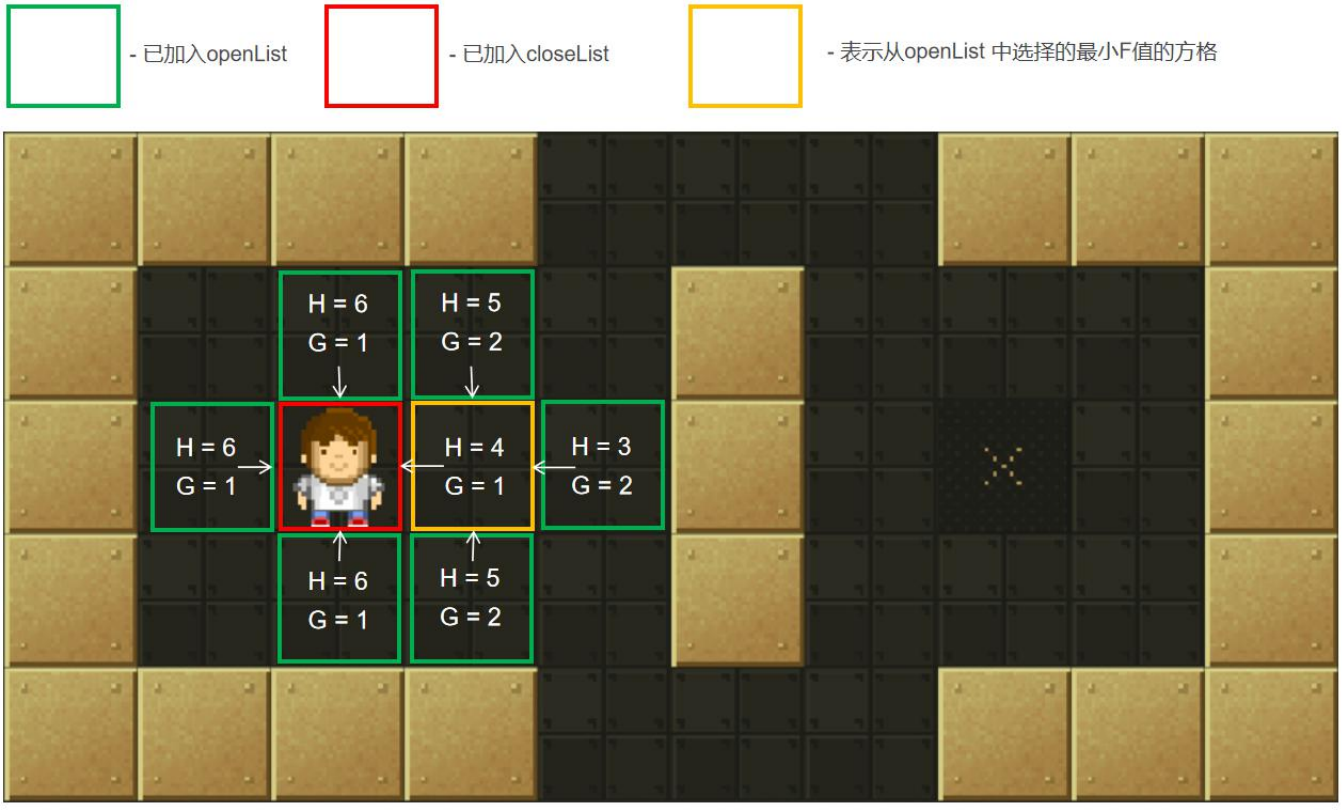
第一步: 把起点加入 openList



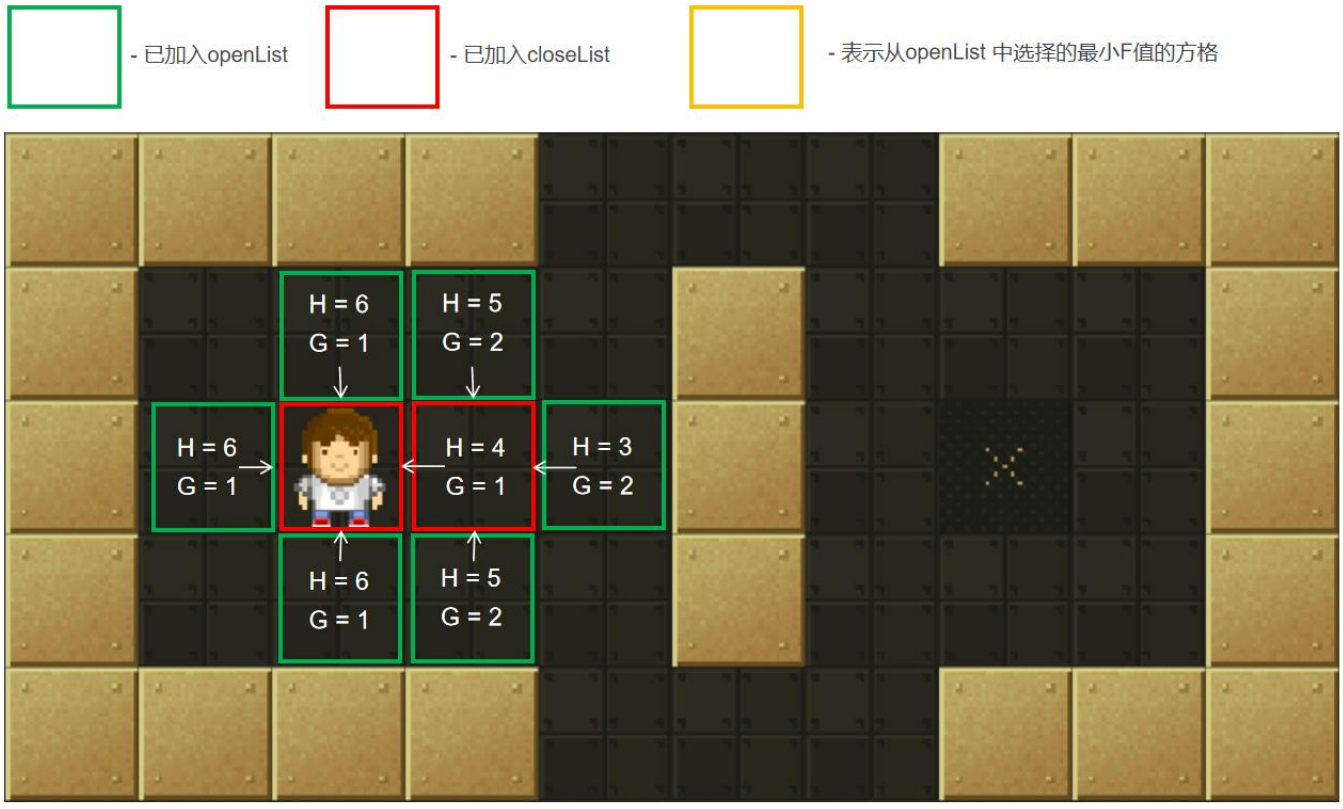
第二步: 从openList中选择最小的F值，此时只有起点一个节点,计算它周边可达方格的G、H和F值，标记这些节点的父节点为当前选择节点(起点)，并把它们加入openList中



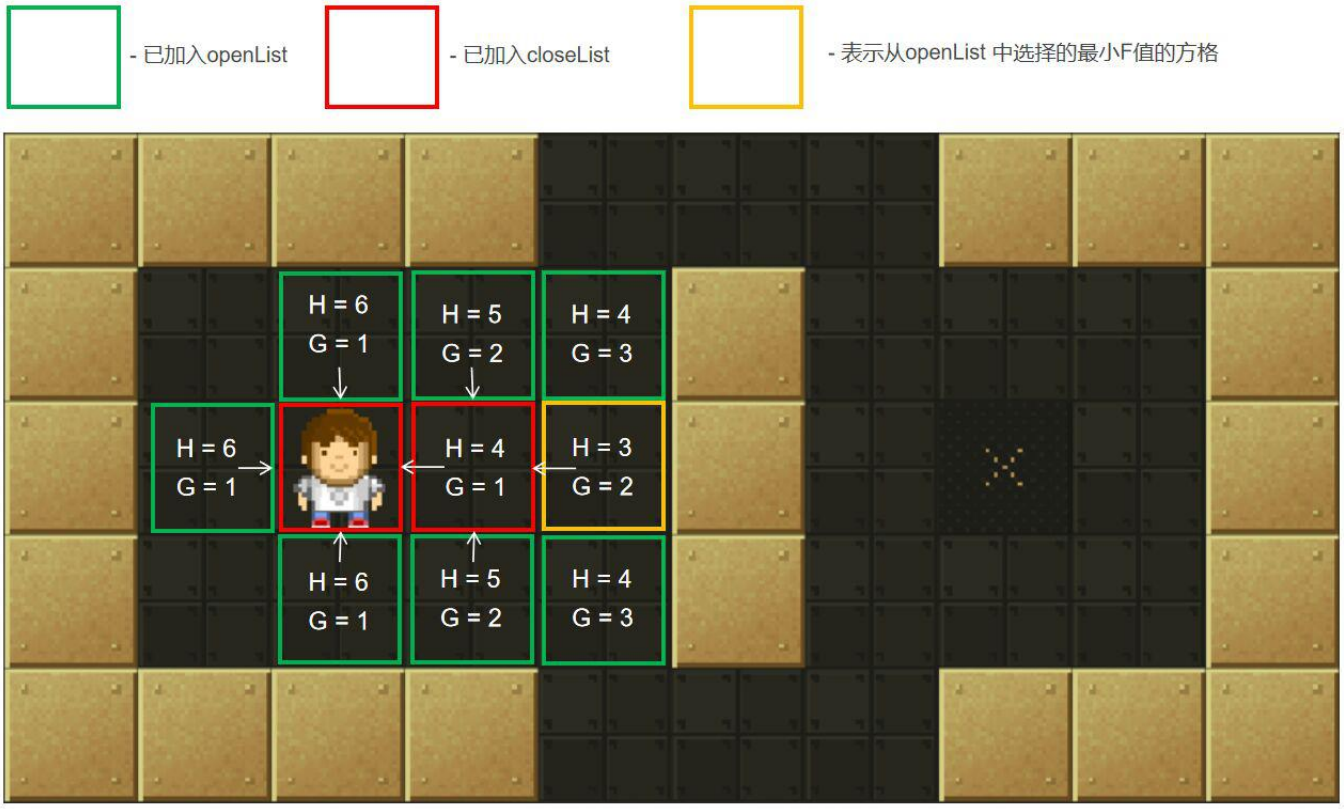
第三步: 把起点加入关闭列表closeList中



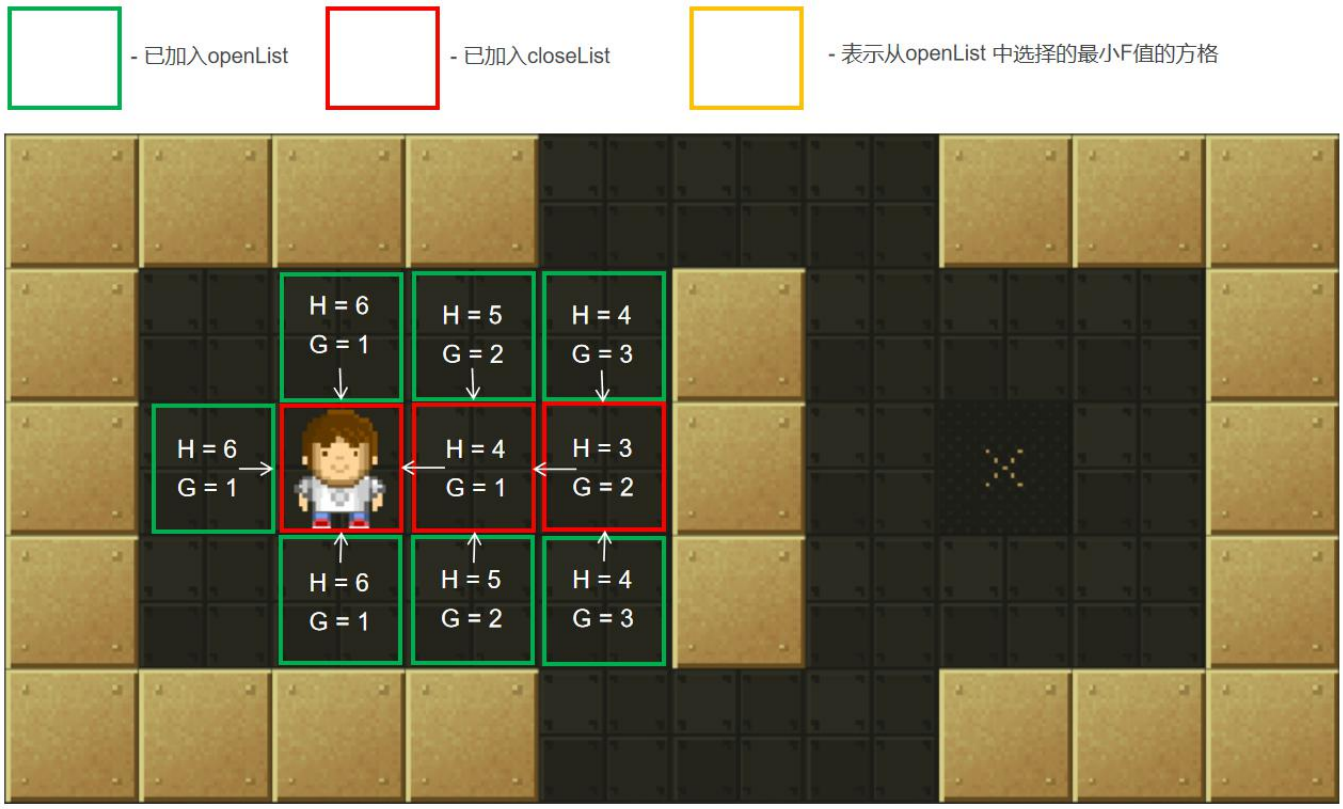
第四步: 从openList中选择最小F值的节点,如黄色所示,计算它周边可达方格的G、H和F值，标记这些节点的父节点为当前选择节点(起点)，并把它们加入openList中



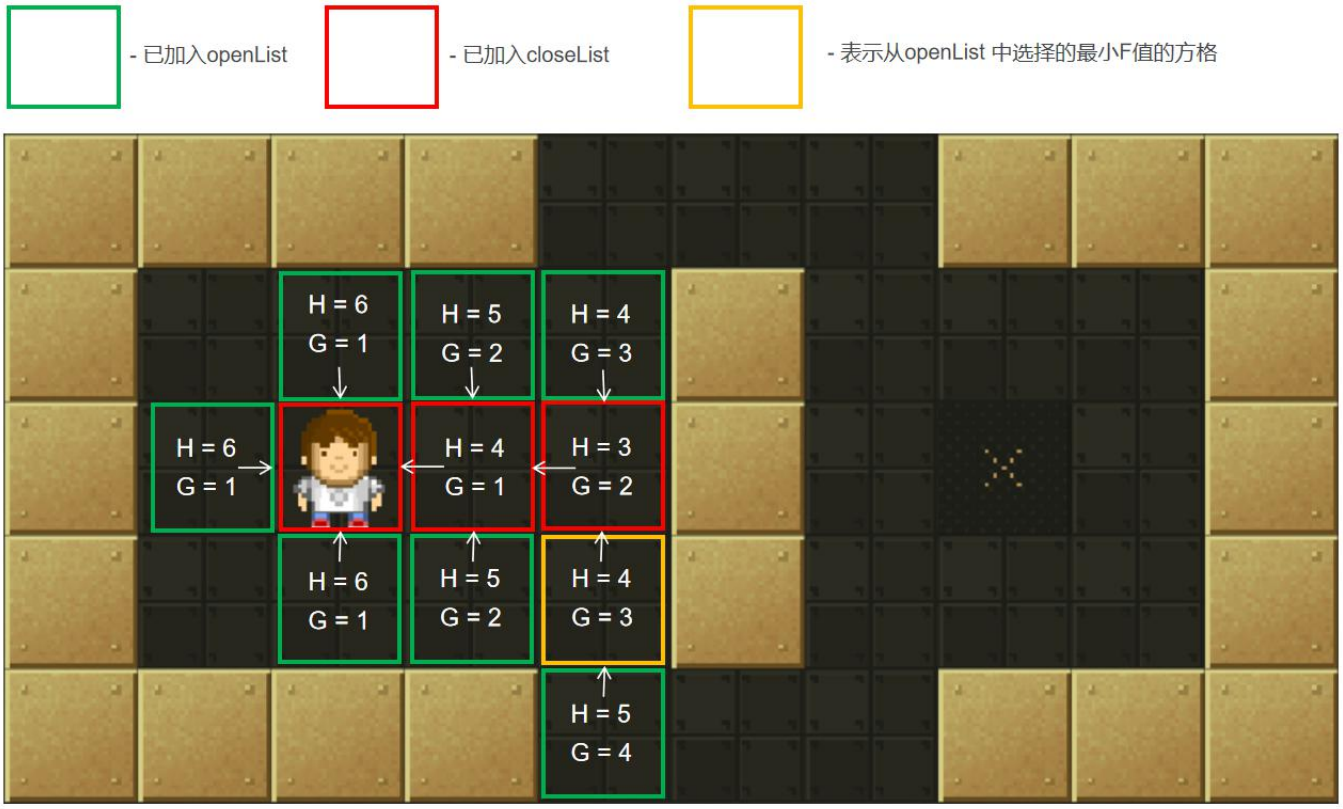
第五步: 把当前节点（上图黄色）加入关闭列表closeList中



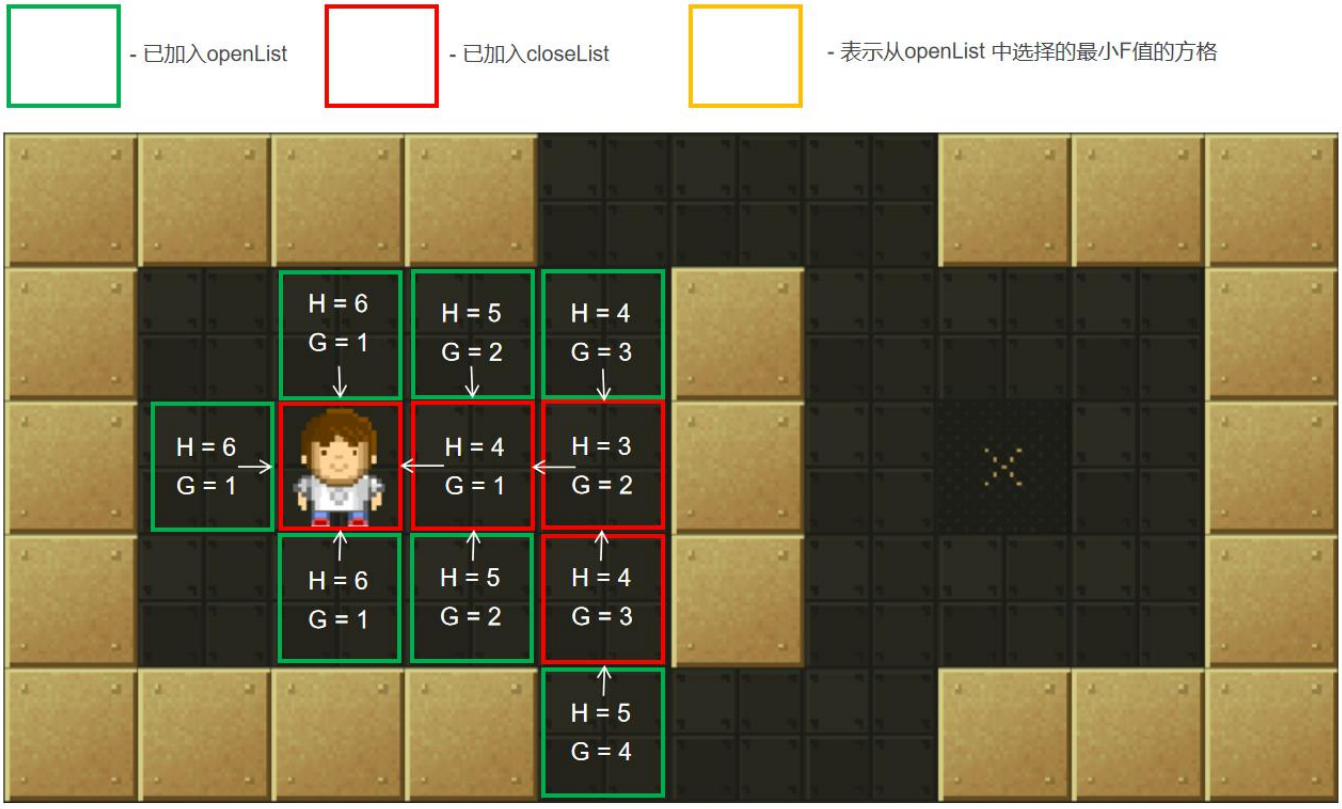
第六步: 从openList中选择最小F值的节点,如黄色所示,计算它周边可达方格的G、H和F值，标记这些节点的父节点为当前选择节点(起点)，并把它们加入openList中



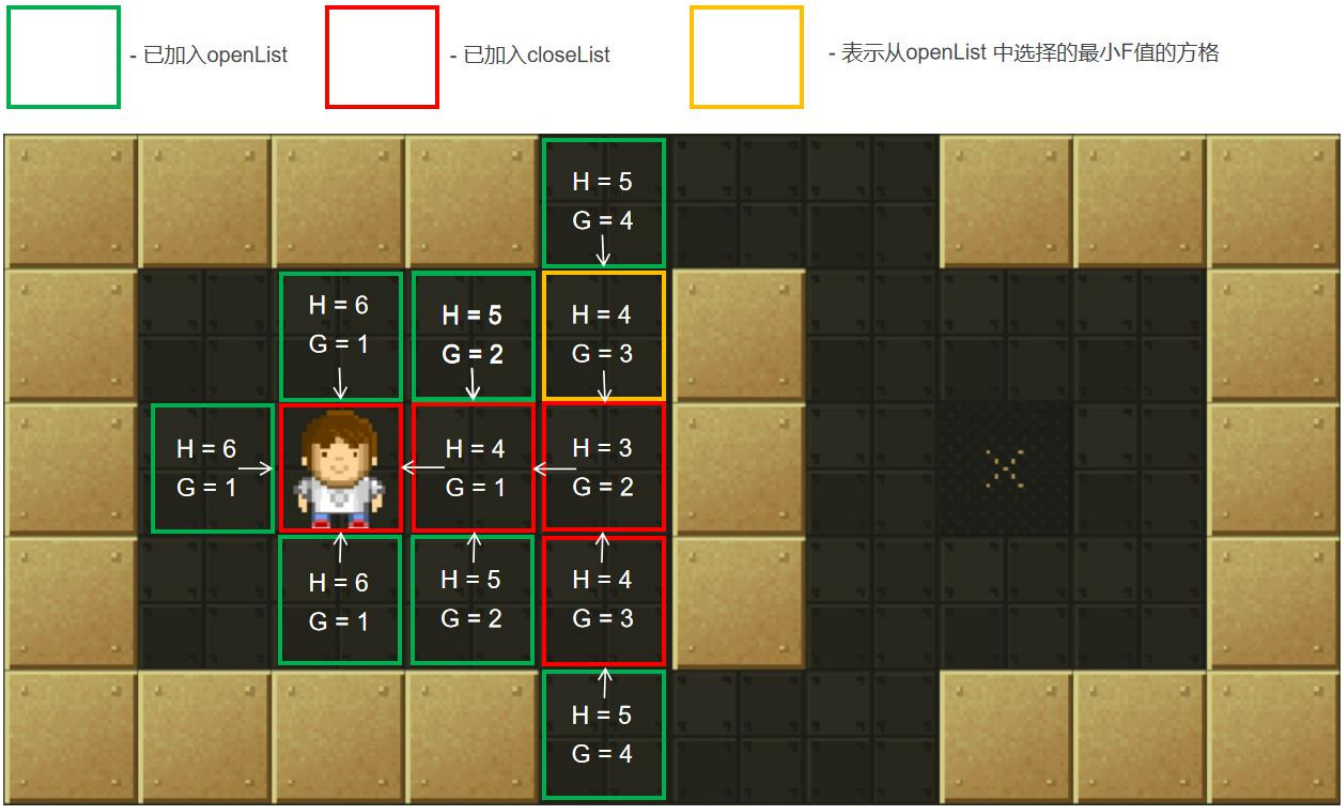
第七步: 把当前节点（上图黄色）加入关闭列表closeList中



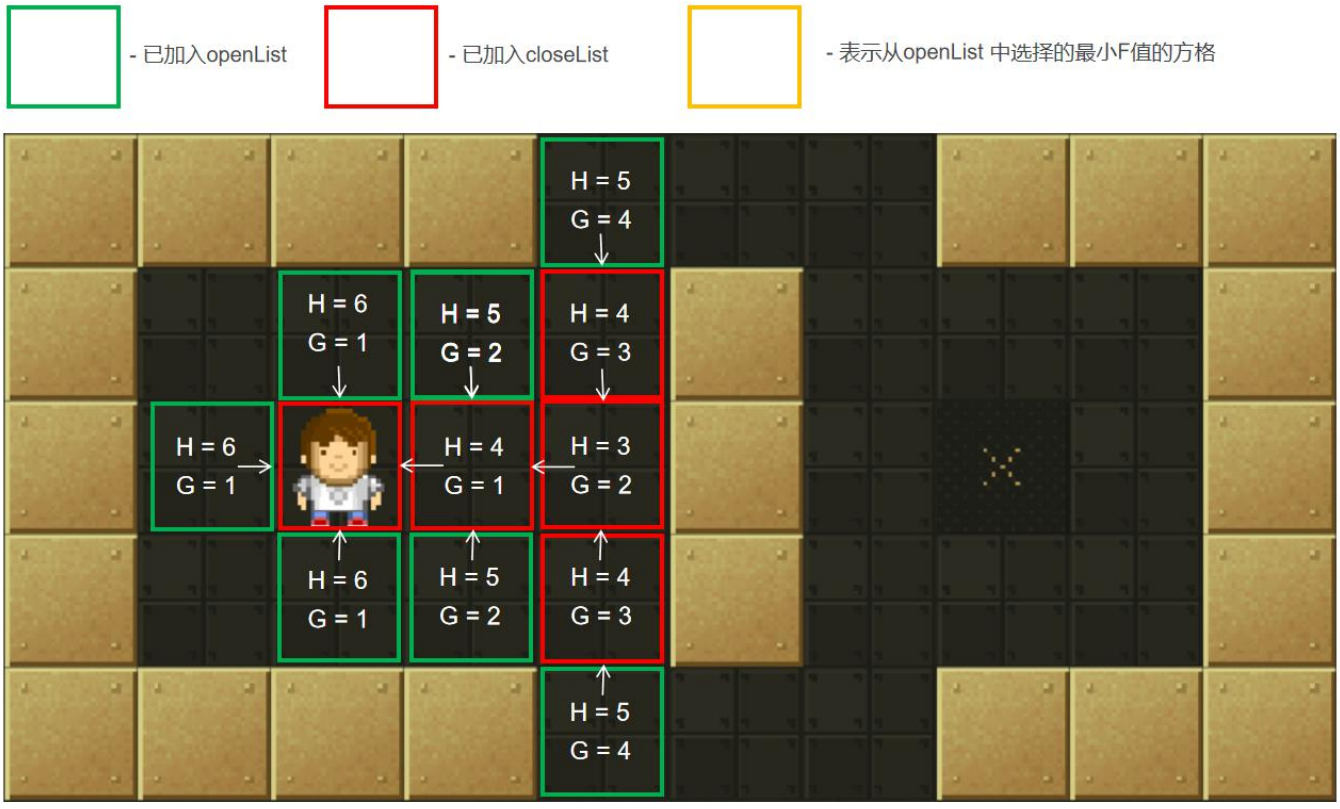
第八步: 从openList中选择最小F值的节点,如黄色所示,计算它周边可达方格的G、H和F值，标记这些节点的父节点为当前选择节点(起点)，并把它们加入openList中



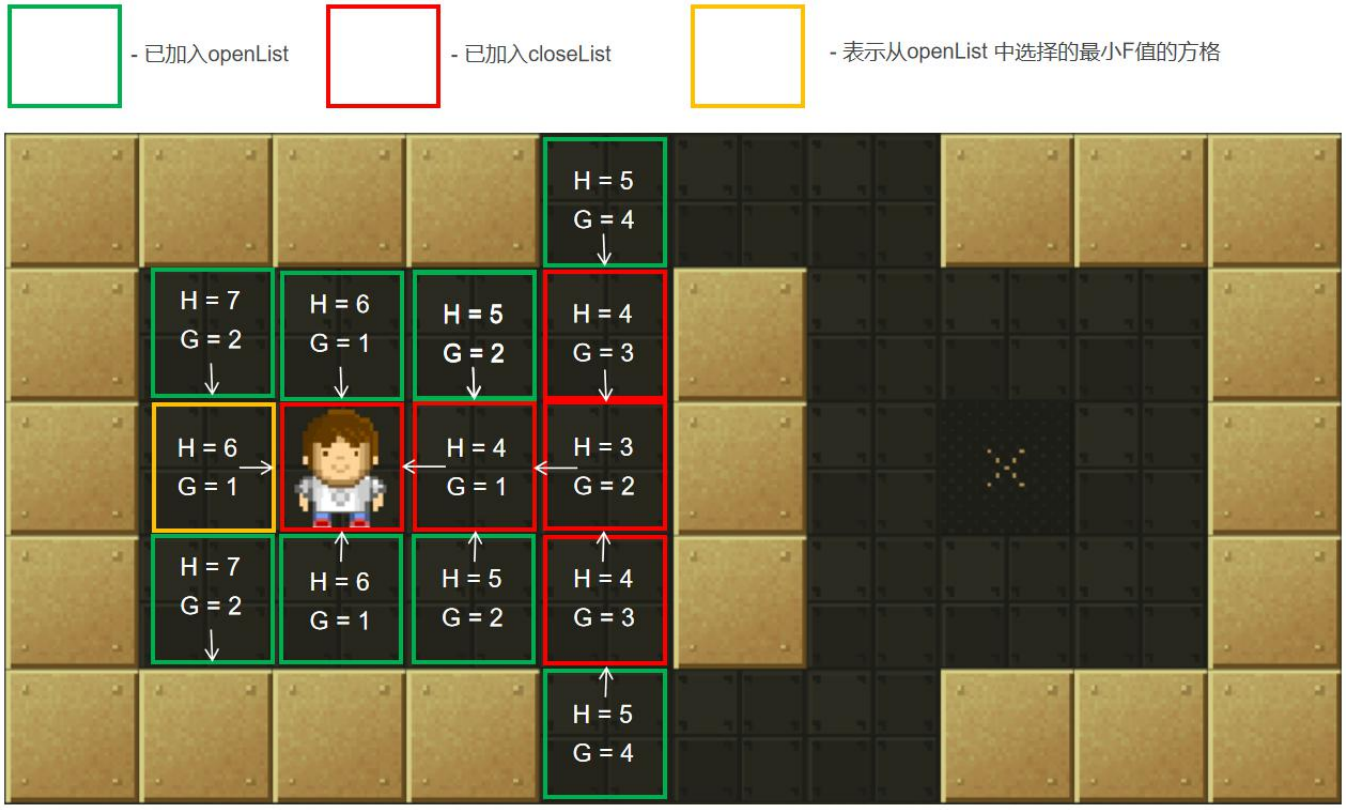
第九步: 把当前节点（上图黄色）加入关闭列表closeList中



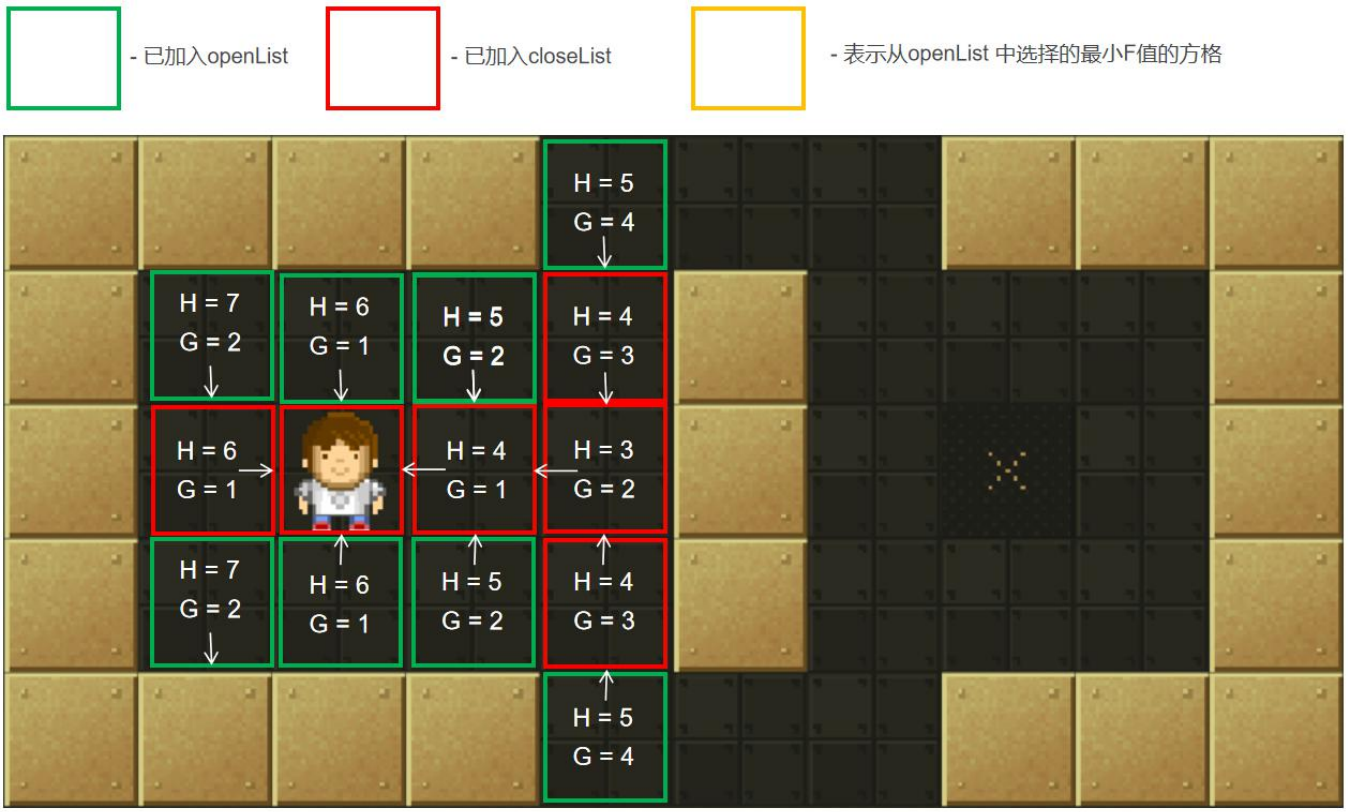
第十步: 从openList中选择最小F值的节点,如黄色所示,计算它周边可达方格的G、H和F值，标记这些节点的父节点为当前选择节点(起点)，并把它们加入openList中



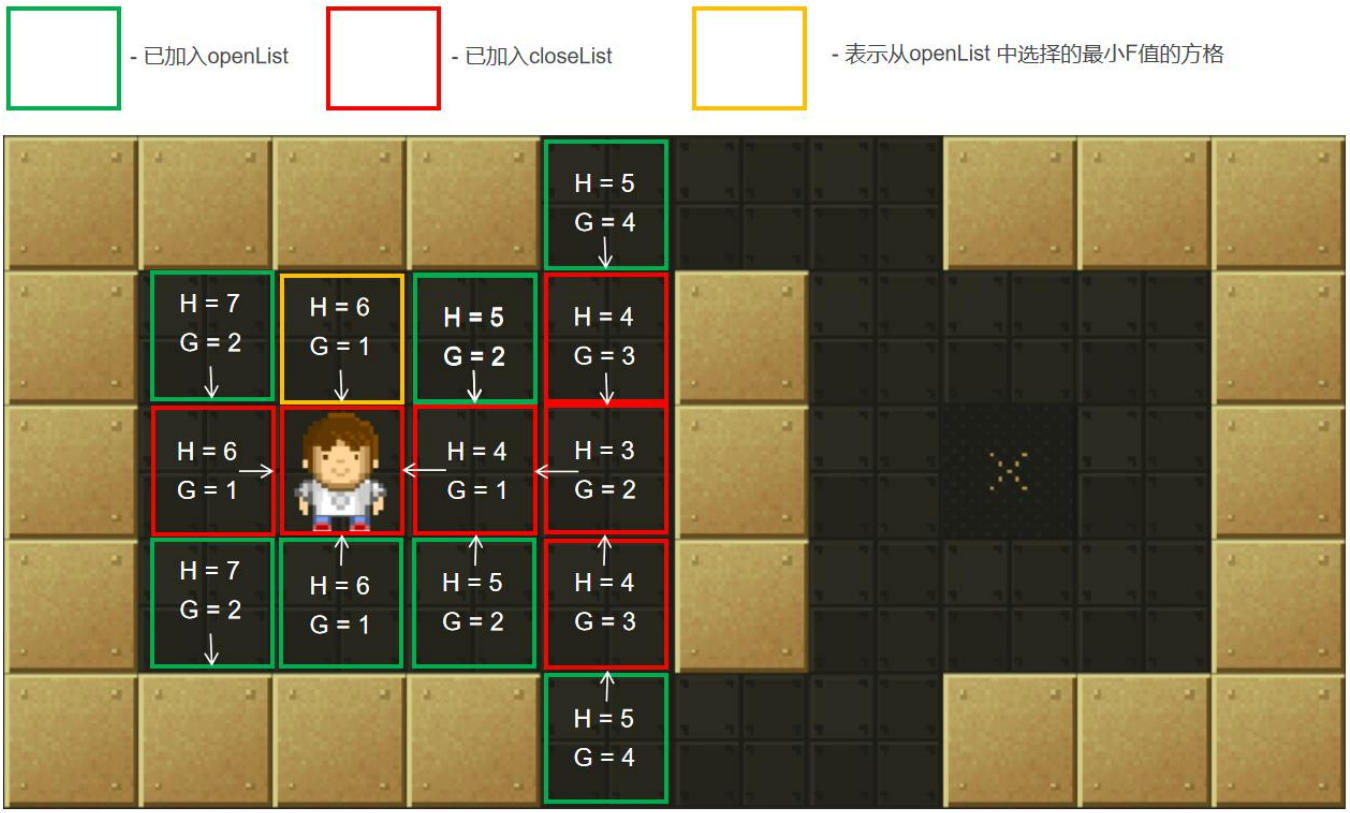
第十一步: 把当前节点（上图黄色）加入关闭列表closeList中



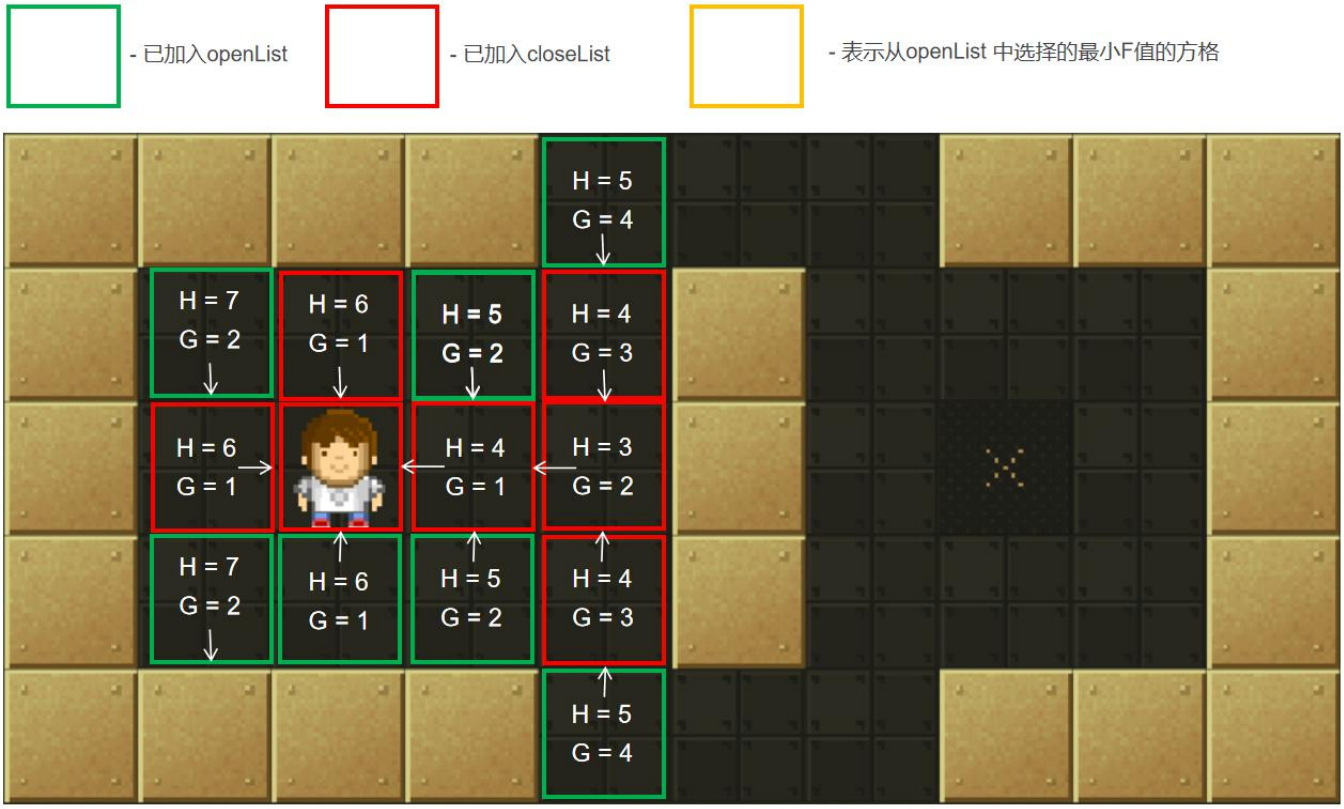
第十二步: 从openList中选择最小F值的节点,如黄色所示,计算它周边可达方格的G、H和F值，标记这些节点的父节点为当前选择节点(起点)，并把它们加入openList中



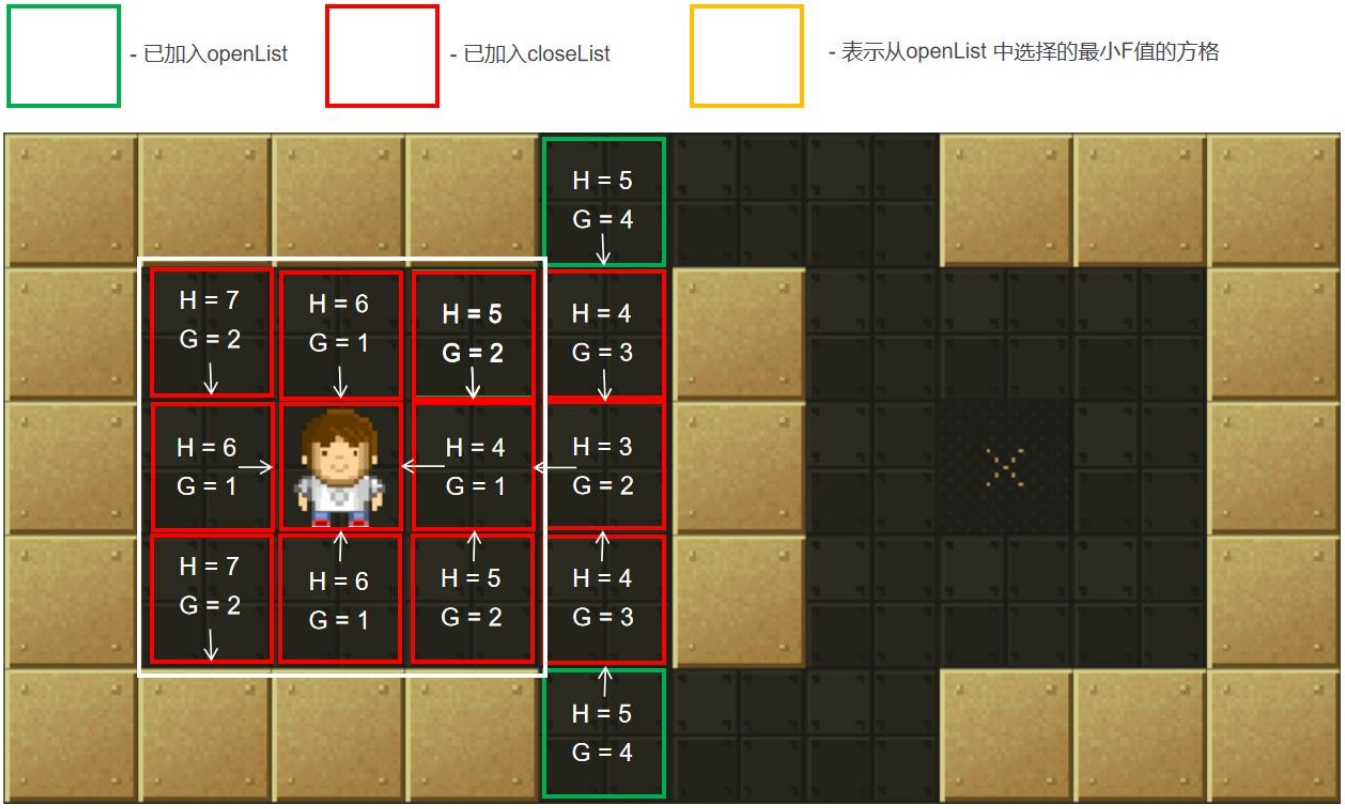
第十三步: 把当前节点（上图黄色）加入关闭列表closeList中



第十四步: 从openList中选择最小F值的节点,如黄色所示,此时， 它身边的可达的点都在closeList 中或是在openList 中，closeList 的直接排除， openList 中的相邻节点重新计算F值， 但他们均不小于原值， 保持现状不变。

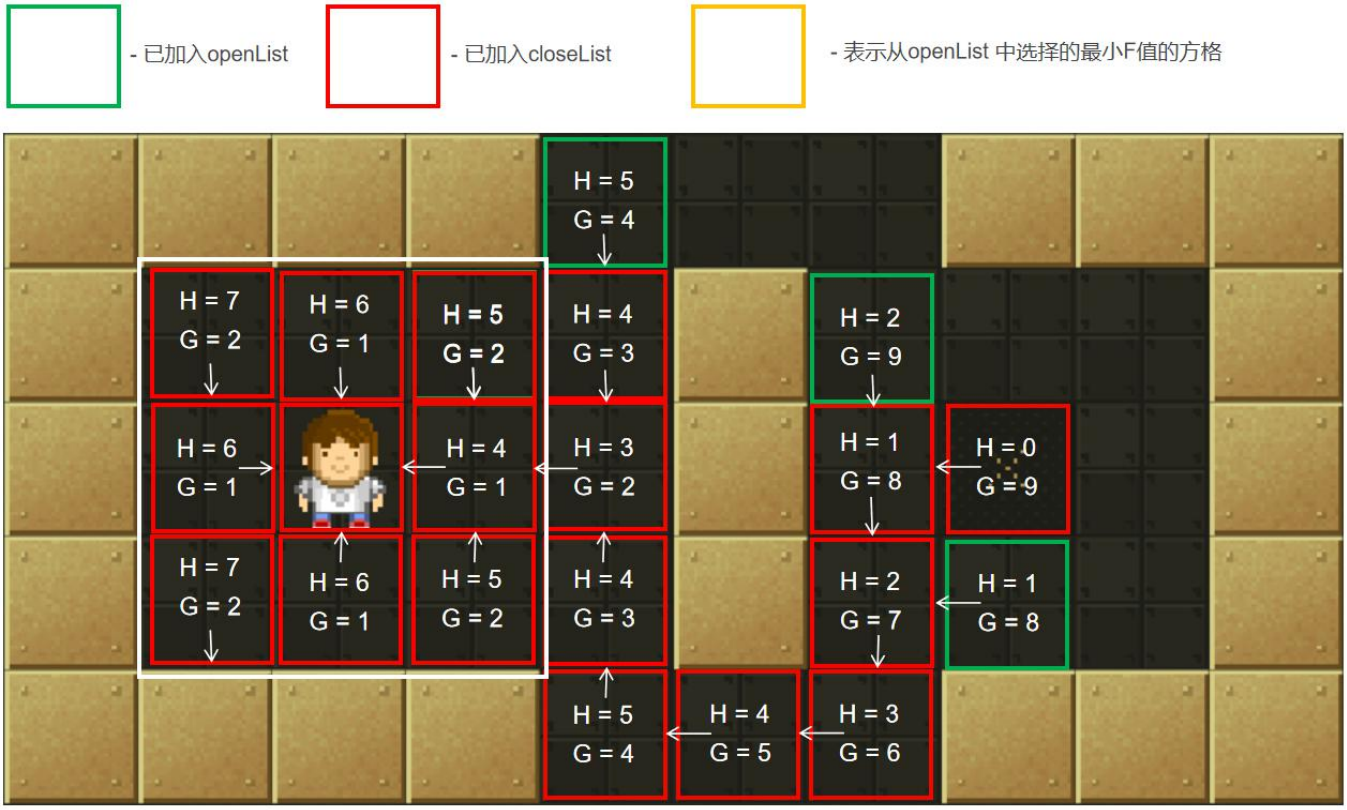


第十五步: 把当前节点（上图黄色）加入关闭列表closeList中



第十六步: 按照第十四步和第十五步的步骤，白色区域中的绿色格子最终都会变成红色

第十七步: 继续从openList中选择最小F值的节点,如黄色所示,继续按照前面的重复步骤寻路，最终当终点也变成红色后，寻路结束。



第十八步: 最终结果。

算法实现

Astar.h

```
#pragma once

#include <vector>
#include <list>

const int kCost1=10; //直移一格消耗
const int kCost2=14; //斜移一格消耗

typedef struct _Point
{
    int x,y; //点坐标，这里为了方便按照 C++的数组来计算，x 代表横排，y 代表竖列
    int F,G,H; //F=G+H
    struct _Point *parent; //parent 的坐标
}Point;



//    void InitAstar(int *_maze, int _lines, int _colums);
Point* AllocPoint(int x, int y);

void InitAstarMaze(int *_maze, int _lines, int _colums);
std::list<Point *> GetPath(Point *startPoint,Point *endPoint);
void ClearAstarMaze();
```

Astar.cpp

```
#include <math.h>
#include "Astar.h"
#include <iostream>

static int * maze; //迷宫对应的二维数组，使用一级指针表示
static int cols;    //二维数组对应的列数
static int lines;   //二维数组对应的行数

static std::list<Point *> openList;  //开放列表
static std::list<Point *> closeList; //关闭列表



static  Point *findPath(Point &startPoint,Point &endPoint,bool
isIgnoreCorner);
static    std::vector<Point *> getSurroundPoints(const Point *point) ;
static    bool isCanreach(const Point *point,const Point *target) ; //
判断某点是否可以用于下一步判断
static    Point *isInList(const std::list<Point *> &list,const Point
*point) ; //判断开启/关闭列表中是否包含某点
static    Point *getLeastFpoint(); //从开启列表中返回 F 值最小的节点

//计算 FGH 值
static    int calcG(Point *temp_start,Point *point);
static    int calcH(Point *point,Point *end);
static    int calcF(Point *point);

/*分配节点*/
Point* AllocPoint(int x, int y){
    Point *temp = new Point;
    memset(temp, 0, sizeof(Point));//初始值清零

    temp->x = x;
    temp->y = y;
    return temp;
}

/*初始化 A*搜索的地图*/
void InitAstarMaze(int *_maze, int _lines, int _colums)
{
    maze =  _maze;
    lines = _lines;
    cols = _colums;
}
```

```
/*清理地图，并释放资源*/
void ClearAstarMaze(){
    maze =  NULL;
    lines = 0;
    cols = 0;

    std::list<Point *>::iterator itor;

    //清除 openList 中的元素
    for(itor = openList.begin(); itor!=openList.end();){
        delete *itor;
        itor = openList.erase(itor);
    }

    //清理 closeList 中的元素
    for(itor = closeList.begin(); itor!=closeList.end();){
        delete *itor;
        itor = closeList.erase(itor);
    }
}

/*计算节点的 G 值*/
static int calcG(Point *temp_start, Point *point)
{
    int
extraG=(abs(point->x-temp_start->x)+abs(point->y-temp_start->y))==1?k
Cost1:kCost2;
    int parentG=(point->parent==NULL? NULL:point->parent->G); //如果是
初始节点，则其父节点是空
    return parentG+extraG;
}

static int calcH(Point *point,Point *end)
{
    //用简单的欧几里得距离计算 H，可以用多中方式实现
    return
(int)sqrt((double)(end->x-point->x)*(double)(end->x-point->x)+(double
)(end->y-point->y)*(double)(end->y-point->y))*kCost1;
}

/*计算节点的 F 值*/
static int calcF(Point *point)
{
    return point->G+point->H;
}

/*从开放列表中查找最小 F 值的节点*/
```

```
static Point* getLeastFpoint()
{
    if(!openList.empty())
    {
        auto resPoint=openList.front();

        std::list<Point *>::const_iterator itor;

        for(itor = openList.begin(); itor!=openList.end();itor++){
            if((*itor)->F<resPoint->F)
                resPoint=*itor;
        }
        return resPoint;
    }
    return NULL;
}

/*搜索从起点到终点的最佳路径*/
Point* findPath(Point *startPoint,Point *endPoint)
{
    openList.push_back( AllocPoint(startPoint->x, startPoint->y)); //
置入起点,拷贝开辟一个节点，内外隔离
    while(!openList.empty())
    {
        auto curPoint=getLeastFpoint(); //找到 F 值最小的点
        //std::cout<<"--找到最小的点: G="<<curPoint->G<<"
F="<<curPoint->F<<" "<<curPoint->H<<std::endl;
        //system("pause");
        openList.remove(curPoint); //从开启列表中删除
        closeList.push_back(curPoint); //放到关闭列表
        //1. 找到当前周围八个格中可以通过的格子
        auto surroundPoints=getSurroundPoints(curPoint);
        //std::cout<<"Point 数目："<<surroundPoints.size()<<std::endl;
        std::vector<Point *>::const_iterator iter;
        for(iter=surroundPoints.begin();iter!=surroundPoints.end();)
        {
            Point * target = *iter;
            //std::cout<<"周围的点：  x="<<target->x<<"
y="<<target->y<<std::endl;

            //2,对某一个格子，如果它不在开放列表中，加入到开启列表，设
置当前格为其父节点，计算 F G H
            Point *exist = isInList(openList,target);
            if(!exist)
            {
                target->parent=curPoint;
```

```
                target->G=calcG(curPoint,target);
                target->H=calcH(target,endPoint);
                target->F=calcF(target);

                openList.push_back(target);
                iter++;
            }
            //3，对某一个格子，它在开放列表中，计算 G 值， 如果比原来的大，
就什么都不做， 否则设置它的父节点为当前点,并更新 G 和 F
            else
            {
                int tempG=calcG(curPoint, target);
                if(tempG<target->G)
                {
                    exist->parent=curPoint;

                    exist->G=tempG;
                    exist->F=calcF(target);
                }
                delete *iter;
                iter = surroundPoints.erase(iter);


            }//end else

        }
        //end for
        surroundPoints.clear();

        Point *resPoint=isInList(openList, endPoint);
        if(resPoint){
            return resPoint; //返回列表里的节点指针，不要用原来传入的
endpoint 指针，因为发生了深拷贝
        }
    }

    return NULL;
}

std::list<Point *> GetPath(Point *startPoint,Point *endPoint)
{
    Point *result=findPath(startPoint, endPoint);
    std::list<Point *> path;
    //返回路径，如果没找到路径，返回空链表
    while(result)
    {
        path.push_front(result);
```

```
        result=result->parent;
    }
    return path;
}

static Point* isInList(const std::list<Point *> &list,const Point
*point)
{
    //判断某个节点是否在列表中，这里不能比较指针，因为每次加入列表是新
开辟的节点，只能比较坐标
    std::list<Point *>::const_iterator itor;

    for(itor = list.begin(); itor!=list.end();itor++){
        if((*itor)->x==point->x&&(*itor)->y==point->y) {
            return *itor;
        }
    }
    return NULL;
}

bool isCanreach(const Point *point, const Point *target)
{
    if(target->x<0||target->x>(lines-1)
        ||target->y<0||target->y>(cols-1)
        ||(maze[target->x * cols + target->y]==1 )
        ||maze[target->x * cols + target->y]==2
        ||(target->x==point->x&&target->y==point->y)
        ||isInList(closeList,target)) //如果点与当前节点重合、超出地图、
是障碍物、或者在关闭列表中，返回 false
        return false;
    else
    {
        if(abs(point->x-target->x)+abs(point->y-target->y)==1){ //非斜
角可以
            //std::cout<<"--点可达: x="<<point->x<<"
y="<<point->y<<std::endl;
            return true;
        }
        else
        {
            return false;
        }
    }
}

/*获取当前点周边可走通的节点*/
std::vector<Point *> getSurroundPoints(const Point *point)
```

```
{
    std::vector<Point *> surroundPoints;

    for(int x=point->x-1;x<=point->x+1;x++) {
        for(int y=point->y-1;y<=point->y+1;y++){
            Point *temp = AllocPoint(x,y);
            if(isCanreach(point, temp)) {
                surroundPoints.push_back( temp);
            }
        }
    }
    return surroundPoints;
}
```

QQ 交流群：875300321