《数据结构》课程综合设计

1. 设计目的

在计算机科学中，数据结构是一般程序设计的基础。通过综合设计，使学生学会分析研究数据结构的特征，以便为应用涉及的数据选择适当的逻辑结构、存储结构及相应的算法，掌握算法的时间复杂度分析技术。另一方面，综合设计也是复杂程序设计的训练过程，要求学生编写的程序结构符合软件工程规范，培养他们的数据抽象能力、建模能力和算法设计能力，提高复杂问题的解决能力，为后续课程的学习和应用奠定基础。

2. 任务与要求

要求学生以 5 人一组，自由结合，从给定的综合设计题目中进行选择。本次设计题目是设计内容不固定的题目，这就要求学生自己先确定本组设计题目，然后确定具体内容和设计目标，从而为数据结构设计、数据建模和算法设计奠定基础。提交的资料包含综合设计纸质版、电子版及源程序电子版。

3. 迷宫探路系统

3.1 系统（问题）描述

用图形界面显示迷宫的生成及寻路过程。 为简化问题，规定迷宫为矩形完美迷宫，即两个节点之间只有唯一一条路径。迷宫单元为带边框的填
充正方形，以边框表示墙。迷宫的终点与起点为矩形的左上角与右下角。系统中的迷宫大小为 45*45

3.2 任务分配

3.3 系统需求

3.3.1 系统描述：

系统界面由两部分组成：迷宫展示界面与信息界面。 迷宫展示界面用来展示迷宫，以图形形式演示迷宫算法。 信息界面用来显示当前展示迷宫用到的算法的名称，以及当展示结束时，显示退出按钮。

3.3.2 应实现的功能：

迷宫生成功能包含：DFS、Kruskal、Prim、递归分割。 迷宫寻路功能包含：DFS、BFS、A*。 其他还包括迷宫信息的显示以及退出系统功能。

迷宫演示为实时计算，用动画展示，每次运行结果不同。并以图片形式保存每个算法的结果。 开发环境：VS2017，EasyX 20190529(beta)为使用的图形库。

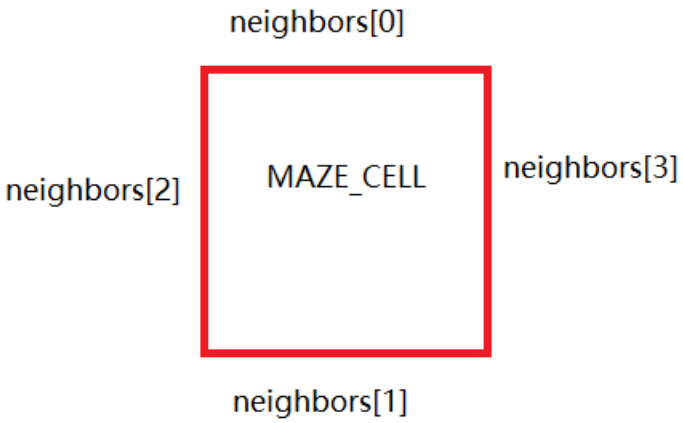
3.4 设计思想

3.4.1 数据结构的选择和设计

（1） 迷宫及迷宫单元格 用图来描述迷宫，所生成的迷宫为完美迷宫，即一棵包含所有节点的生成树，任意两点之间只有唯一路径可走，生成迷宫的算法即为生成树算法。则所解决的问题可归结为在一棵以起点为根节点的树中搜索终点节点并显示路径的问题。

本题中的数据结构设计的关键在定义迷宫的存储结构。传统的图的的存储结构不太适合描述本题迷宫。原因是每个迷宫单元格之间存在明显的几何关系，决定了单元格之间的逻辑关系。位置相邻的单元格之间才可能连通。对于一个迷宫，需要存储其每个单元格与上下左右邻居的连通情况，即这个位置的墙是否打通，并且便于使用位置坐标来得到每个单元格的位置。

设计如下的迷宫单元表示法。定义该结构的二维矩阵表示整个迷宫，也方便迷宫的绘制。




```
typedef struct {                    //迷宫单元格类
      int is_visited;               //辅助变量，可以用来表示该单元格是否访问
     int neighbors[4];              //邻居单元的连通情况,0 为该墙未访问，-1 为不能打通（生成时为迷宫边界），1 为连通，在生成完毕后将所有的 0 变为-1，因为显然生成时所有的墙都不能打通
}maze_cell,**MAZE_T;
```

为了方便表示方向，将上、下、左、右与 0、1、2、3 映射关系描述成枚举类型，并用全局数组得到对应的方向向量（POINT 定义在使用的图形库中）。

```
enum vec    {VEC_UP, VEC_DOWN, VEC_LEFT, VEC_RIGHT};    //step[n] 得到的是对应方向的向量
const POINT MAZE_STEP[4] = { {0,-1},{0,1},{-1,0},{1,0} };//全局数组
```

（2） 用于 Kruskal 算法的数据结构

在 Kruskal 算法生成迷宫的过程中使用了并查集。并查集是一种树型的数据结构，用于处理一些不相交集合的合并及查询问题。在使用中常常以森林来表示。一个并查集节点中包含其根节点的信息，以及该节点子孙节点的数目，用于启发式合并，防止退化。该并查集实现了初始化并查集，合并两个并查集，判断两个节点是否属于同一个并查集以及销毁并查集操作。对应迷宫的矩阵表示，同样用节点的二维矩阵来表示整个并查集。

```
typedef struct {                //并查集，用于 kruskal 算法
```

```
    POINT parent;                   //双亲结点
    int rank;                       //作为根结点时的度
   }ufset_node, **maze_ufset;
```

（3） 用于A*算法的数据结构

在 A*算法寻找路径的过程中使用到了优先队列（最小的元素先出队），它用堆来实现（即用数组表示完全二叉树，且父节点大于孩子节点，兄弟节点之间无序）。令孩子节点下标为 m，其父节点下标为 n，利用 n=m/2 的原理，在堆中插入删除节点的同时，堆顶的元素为最小的。

其他算法用到的常见数据结构如栈，队列**等**，不在此赘述。

3.4.2 算法设计

一、迷宫生成部分

1、深度优先 生成迷宫时，随机选择初始位置，压入栈底，标记为已遍历。将当前点设为初始位置，然后开始移动。

移动时判断周围的连通情况，随机选择一个可以前进的方向。判断的依据是：不能走出迷宫，且不能走到已经遍历过的点上（会形成环），将每一个走过的节点压栈。当前位置无路可走时，退栈，原路返回，直到该位置有至少一个可以走的点为止。

循环至少执行一次。当迷宫长度为一时（退回到初始位置时）结束循环。

算法描述：

```
void generate_maze_dfs(MAZE MAZE) {//DFS 生成迷宫
    //储存来路的栈
    init_maze_stack(&MAZE_STACK);

    //随机选择起始点
    POINT cur = { rand() % MAZE_LENGTH, rand() % MAZE_HEIGHT};
    //起始点标记为已遍历
    MAZE[i][j].is_visited = 1;
    //起始点入栈
    maze_stack_push(&MAZE_STACK, cur);
    do {
        //移动当前位置
        move_cur_generate_dfs(MAZE, &cur, &MAZE_STACK);

    } while (MAZE_STACK.len != 1);//当回到原点时，说明所有点都走完了
    //删除辅助数据
    destory_maze_stack(&MAZE_STACK);
    return;
}

void move_cur_generate_dfs(MAZE MAZE,POINT *cur,maze_stack *MAZE_STACK) {//移动当前位置

    //记录可以走的方向数目
```

```
    num = get_moveable_choice(MAZE,cur);

    //若无路，就退回到最近的可走的点
    if (num == 0) move_back_generate_dfs(MAZE,cur,MAZE_STACK);
    else {
    //如果有选择一个方向移动，并将两边的邻居设置为 1
        vec direction = get_a_rand_direction(MAZE,cur);
        MAZE[cur->x][cur->y].neighbors[direction] = 1;

        //改变当前坐标
        cur += MAZE_STEP[direction];
        MAZE[cur->x][cur->y].is_visited = 1;

        //互相连通,得到之前的方向
        switch (direction) {
            case 1:
                direction = (vec)0;
                break;
            case 0:
                direction = (vec)1;
                break;
            case 2:
                direction = (vec)3;
                break;
            case 3:
                direction = (vec)2;
                break;
        }
        MAZE[cur->x][cur->y].neighbors[direction] = 1;

        //新节点入栈
        maze_stack_push(MAZE_STACK,*cur);
    }
    return;
}
```

2、Prim

思路是图的最小生成树，因为每一个路的节点是一个叶子节点，节点的权值均相等。 如果某一个墙节点左右或上下的路只有一个连通，则打通，并将新打通的叶子节点的墙加入队列 墙的选取为随机，对应权值的平均

当队列中墙的数量为空时循环结束

算法描述：

```
void generate_maze_prim(MAZE MAZE) {
```

```
    //建立边表
    init_maze_edge_list(&edges);
    //随机选择一个点
    POINT cur = { rand() % MAZE_LENGTH, rand() % MAZE_HEIGHT};
    //记录访问的边数
    cnt = 1;

    do {
        //访问当前点
        MAZE[cur.x][cur.y].is_visited = 1;
        //若这条路是连通而且未访问过的
        for (int i = 0; i < 4; i++) {
            if (MAZE[cur.x][cur.y].neighbors[i] == 0) {
                //ngh 记录当前查看的邻居
                ngh = { cur.x + MAZE_STEP[i].x,cur.y + MAZE_STEP[i].y};
                //就把这条边加入表中
                if (MAZE[ngh.x][ngh.y].is_visited != 1) {
                    e = { cur,ngh };
                    maze_edge_list_push(&edges, e);
                }
            }
        }
        do
        {
            //随机弹出边，直到该边没有访问为止
            maze_edge_list_pop(&edges, &e);

        } while (MAZE[e.p1.x][e.p1.y].is_visited == 1 && MAZE[e.p2.x][e.p2.y].is_visited == 1);
        //让两边的邻居相互连通
        MAZE[e.p1.x][e.p1.y].neighbors[POINT_TO_VEC(e.p1, e.p2)] = 1;
        MAZE[e.p2.x][e.p2.y].neighbors[POINT_TO_VEC(e.p2, e.p1)] = 1;

        //移动到下一个未访问的点
        cur = MAZE[e.p1.x][e.p1.y].is_visited == 1 ? e.p2 : e.p1;

        ++cnt;//记录访问顶点数
    } while (cnt < MAZE_LENGTH*MAZE_HEIGHT);// 访问所有顶点时退出

    //删除辅助数据
    destory_maze_edge_list(edges);
    return;
}
```

3、Kruskal

在无权图中，该算法较在网中简单了不少。

建立含有所有墙的线性表，从线性表中随机弹出一个墙，判断该边两端的节点是否属于同一个并查集。若是，则不做操作；若不是则将该墙移除，连通两个节点，将节点所属的并查集合并。直到所有的边都遍历过，生成完毕。

算法描述：

```
void generate_maze_krus(MAZE MAZE) {//krukal 生成迷宫
    //建立并查集
    init_maze_ufset(&set);
    //建立边表
    init_maze_edge_list(&edges);
    //表中加入所有边
    for (int i = 0; i < MAZE_HEIGHT; i++) {
        for (int j = 0; j < MAZE_LENGTH - 1; j++) {
            e.p1 = { j,i };
            e.p2 = { j + 1,i };
            maze_edge_list_push(&edges, e);
        }
    }

    for (int i = 0; i < MAZE_LENGTH; i++) {
        for (int j = 0; j < MAZE_HEIGHT - 1; j++) {
            e.p1 = { i,j };
            e.p2 = { i,j + 1 };
            maze_edge_list_push(&edges, e);
        }
    }

    //当该表不为空
    while (!maze_edge_list_empty(edges)) {

        //取出一个边
        maze_edge_list_pop(&edges, &e);

        //记录边所对应的点的根节点
        r1 = get_maze_ufset_root(set, e.p1);
        r2 = get_maze_ufset_root(set, e.p2);

        //如果根节点不同（不属于一个并查集）
        if (r1.x != r2.x || r1.y != r2.y) {

            //合并两个并查集
            joint_maze_ufset(set, r1, r2);

            //连通两个点
```

```
            MAZE[e.p1.x][e.p1.y].neighbors[POINT_TO_VEC(e.p1, e.p2)] = 1;
            MAZE[e.p2.x][e.p2.y].neighbors[POINT_TO_VEC(e.p2, e.p1)] = 1;
        }
    }
    //删除辅助数据
    destory_maze_ufset(set);
    destory_maze_edge_list(edges);
    return;
}
```

4、递归分割

主要是在空白空间随机生成十字墙壁，将空间分为四个子室，再在三面墙上各自选择一个随机点挖洞，保证联通，之后再对子空间进行分割，直至最后的子空间为单元格。分割时，如果分割区域长或宽为 1，则直接打通内部的所有墙。

算法描述：

```
//递归分割生成迷宫
void generate_maze_recur(MAZE MAZE) {
    //进行分割
    devide_maze(MAZE, 0, MAZE_HEIGHT, 0, MAZE_LENGTH);
    return;
}

//分割迷宫
void devide_maze(MAZE MAZE, int border_top, int border_bottom, int border_left, int border_right) {

    //当这是一个宽或高为 1 的子区块时，其内部的边全部打通，退化为一个单链表
    if (border_bottom - border_top == 1) {
        //所有单元有相同的纵坐标
        y = border_top;
        for (int i = border_left; i < border_right - 1; i++)
        {
            //从左到右依次打通
            MAZE[i][y].neighbors[3] = 1;
            MAZE[i + 1][y].neighbors[2] = 1;
        }
        return;
    }

    if (border_right - border_left == 1) {
        //所有单元有相同的纵坐标
        x = border_left;
        for (int i = border_top; i < border_bottom - 1; i++)
        {
            //从上到下依次打通
```

```
            MAZE[x][i].neighbors[1] = 1;
            MAZE[x][i + 1].neighbors[0] = 1;
        }
        return;
}

////////////////////////////////////////////////
//下面是一般情况
////////////////////////////////////////////////

//随机取十字交叉点的坐标
crossover_x = rand() % (border_right - border_left - 1)+1+border_left,
crossover_y = rand() % (border_bottom - border_top - 1)+1+border_top;

// 随机选择一个不打通的方向
v = rand() % 4;

 //如果不是上方
if (v != VEC_UP)

{    // p1 在 p2 右边，在分界线上随机取点，打通两个区域
     p1 = { crossover_x, border_top + rand() % (crossover_y - border_top)};
     p2 = { p1.x - 1, p1.y };

     MAZE[p1.x][p1.y].neighbors[2] = 1;
     MAZE[p2.x][p2.y].neighbors[3] = 1;
}

//如果不是下方
if (v != VEC_DOWN)

{    // p1 在 p2 右边
     p1 = { crossover_x, crossover_y + rand() % (border_bottom - crossover_y)};
     p2 = { p1.x - 1, p1.y };

     MAZE[p1.x][p1.y].neighbors[2] = 1;
     MAZE[p2.x][p2.y].neighbors[3] = 1;
}
 //如果不是左边
if (v != VEC_LEFT)

{    // p1 在 p2 下边
     p1 = {border_left + rand() % (crossover_x - border_left), crossover_y};
     p2 = { p1.x, p1.y - 1};
```

```
        MAZE[p1.x][p1.y].neighbors[0] = 1;
        MAZE[p2.x][p2.y].neighbors[1] = 1;
    }

    //如果不是右边
    if (v != VEC_RIGHT)

    {   // p1 在 p2 下边
        p1 = {crossover_x + rand() % (border_right - crossover_x), crossover_y};
        p2 = { p1.x, p1.y - 1};

        MAZE[p1.x][p1.y].neighbors[0] = 1;
        MAZE[p2.x][p2.y].neighbors[1] = 1;
    }

    //递归四片子区域
    devide_maze(MAZE, border_top, crossover_y, border_left, crossover_x);
    devide_maze(MAZE, border_top, crossover_y, crossover_x, border_right);
    devide_maze(MAZE, crossover_y, border_bottom, crossover_x, border_right);
    devide_maze(MAZE, crossover_y, border_bottom, border_left, crossover_x);

    return;
}
```

二、迷宫寻路部分

1、深度优先

类似生成算法。设定当前位置的初值为入口位置

算法描述：

```
void explore_maze_dfs(MAZE MAZE) {
    //当前点为起点
    POINT cur = MAZE_START;
    MAZE[cur.x][cur.y].is_visited = 1;

    //起点入栈
    init_maze_stack(&stack);
    maze_stack_push(&stack, cur);

    while (cur.x != MAZE_GOAL.x || cur.y != MAZE_GOAL.y) {
        //移动至下一格
        move_cur_explore_dfs(MAZE, &cur, &stack);
    }
```

```
    //删除辅助数据
    destory_maze_stack(&stack);
}

void move_cur_explore_dfs(MAZE MAZE, POINT *cur, maze_stack *MAZE_STACK){//移动当前位置（DFS 走迷宫）
    //获取当前可选择的方向数量
    choices = get_moveable_choice(MAZE, cur);
    //如果没有可走的路，退回最近的可走的位置
    if (choices == 0) move_back_explore_dfs(cur, MAZE_STACK);
    else {
        //随机选取方向
        vec direction = get_a_rand_direction(MAZE, cur);
        //改变坐标
        cur += MAZE_STEP[direction];
        //当前点入栈
        maze_stack_push(MAZE_STACK, *cur);
        MAZE[cur->x][cur->y].is_visited = 1;
    }
}
```

2、广度优先

思路：从迷宫入口开始，搜索一步可到达的所有位置作为第一层，然后在上一层的基础上再走一步，并且不走重复的格子，作为下一层。这样一层一层的搜索，直至达到终点时搜索停止。

搜索时记录来路，用于绘制路线

算法描述：

```
 void explore_maze_bfs(MAZE MAZE) {

    //储存每个节点的父节点
    POINT **father;
    father = (POINT**)malloc(sizeof(POINT)*MAZE_LENGTH*MAZE_HEIGHT);
    //内存分配失败
    if (!father) exit(-1);
    for (int i = 0; i < MAZE_LENGTH; i++) {
        father[i] = (POINT*)malloc(sizeof(POINT)*MAZE_HEIGHT);
        //内存分配失败

        if (!father[i]) exit(-1);
        memset(father[i], 0, sizeof(POINT)*MAZE_HEIGHT);
    }

    init_maze_queue(&queue);

    POINT cur = MAZE_START;
    //当前点入队
```

```
    maze_enqueue(&queue, cur);

    while (cur.x != MAZE_GOAL.x || cur.y != MAZE_GOAL.y) {
        //移动至下一格
        move_cur_explore_bfs(MAZE, &cur, &queue, father);
    }

    //删除辅助数据
    for (int i = 0; i < MAZE_LENGTH; i++) {
        free(father[i]);
    }
    free(father);
    destory_maze_queue(&queue);
}

//移动当前点
void move_cur_explore_bfs(MAZE MAZE, POINT *cur, maze_queue *queue, POINT** father) {
    //将队头取出，赋给当前位置
    maze_dequeue(queue, cur);
    //若未访问
    if (MAZE[cur->x][cur->y].is_visited == 0) {
        MAZE[cur->x][cur->y].is_visited = 1;

        for (int v = VEC_UP; v <= VEC_RIGHT; v++) {
            //如果该方向可以走
            if (MAZE[cur->x][cur->y].neighbors[(vec)v] == 1) {
                //p 记录当前查看的邻居
                p = cur + MAZE_STEP[v];
                if (MAZE[p.x][p.y].is_visited == 0) {
                    //将未访问的邻居入队
                    maze_enqueue(queue, p);
                    father[p.x][p.y] = *cur;
                }
            }

        }
    }
    return;
}
```

3、A*算法

该算法为启发式算法。在原地图节点的基础上，需要另外三个变量，dis_from（起点到当前点的真实长度），dis_to（表示当前点距终点估计距离，使用当前点与终点的欧几里得距离来估计），ideal_dis= dis_from+ dis_to（理想路径长度）。

在执行算法的过程中，需要两个按 ideal_dis 排序的小根堆，一个称之为 open（用来存放待访问的点），

另一个称之为 close（存放从 open 取出的点）。用 cur 来表示当前点，并把 cur 周围满足条件的点（即该点可到达，且不在 open 或 close 中）填入 open 中，然后把 cur 从 open 中删除并添加到 close 中，并令 open 的第一个点为新的 cur 点，一直循环，直到到达终点。

算法描述：

```
void explore_maze_astar(MAZE MAZE) {

    //存放 A*算法所用数据的结构
    MAZE_TA MAZEA;
    MAZEA = (MAZE_CELLA**)malloc(sizeof(MAZE_CELLA*)*MAZE_LENGTH);
    init_mazea(MAZE, MAZEA);
    //建立两个按估计路径长度排序的小根堆
    maze_min_heap *open, *close;
    open = (maze_min_heap*)malloc(sizeof(maze_min_heap));
    close = (maze_min_heap*)malloc(sizeof(maze_min_heap));
    init_maze_heap(open);
    init_maze_heap(close);

    //初始点为起点，插入 open 堆中
    POINT cur = MAZE_START;
    maze_heap_insert(open, &MAZEA[cur.x][cur.y]);
    MAZEA[cur.x][cur.y].parent = NULL;

    while (cur.x != MAZE_GOAL.x || cur.y != MAZE_GOAL.y) {
        //移动当前位置，直到走到终点为止
        move_cur_astar(MAZEA, &cur,open,close);
    }

    return;
}

void move_cur_astar(MAZE_TA MAZEA, POINT* cur, maze_min_heap *open, maze_min_heap *close) {
    //查看周围邻居
    for (int i = 0; i < 4; i++) {
        neightbor = { cur->x + MAZE_STEP[i].x ,cur->y + MAZE_STEP[i].y };
        //判断邻居是否连通并且没有访问过
        if (MAZEA[cur->x][cur->y].maze_copy.neighbors[i] == 1 &&!maze_heap_search(close,
MAZEA[neighbor.x][ neighbor.y])&& !maze_heap_search(open, neighbor.x][ neighbor.y])) {

            dis_from = &(MAZEA[neightbor.x][ neightbor.y].dis_from);
            dis_to = &(MAZEA[neightbor.x][ neightbor.y].dis_to);
            ideal_dis = &(MAZEA[neightbor.x][ neightbor.y].ideal_dis);
            //记录来路
            MAZEA[neightbor.x][ neightbor.y].parent =&(MAZEA[cur->x][cur->y]);
```

```
            //从终点到当前点的实际距离
            *dis_from = MAZEA[cur->x][cur->y].dis_from + 1;
            //到终点的估计距离
            *dis_to =sqrt(pow(MAZE_GOAL.x - (neightbor.x),2)+pow(MAZE_GOAL.y -(neightbor.y),2));
            //估计的总路程
            *ideal_dis = *dis_from + *dis_to;
            //将该邻居插入堆中
            maze_heap_insert(open, &MAZEA[neightbor.x][ neightbor.y]);

        }
    }
    //将当前点取出，移入 close 中
    maze_heap_erase(open);
    maze_heap_insert(close, &MAZEA[cur->x][cur->y]);
    //设置当前点为估计距离最小的点 (open 堆顶)
    *cur = { open->data[1]->point.x, open->data[1]->point.y };
}
```

三、迷宫绘制部分绘制的具体实现依赖于库函数，由于我们关心的是方法而不是接口的调用，所以在这里对绘图库中的涉及 的概念不做详细描述。

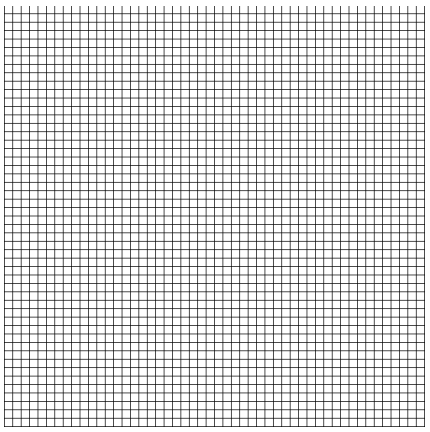
1、基本思想

根据每个迷宫单元的坐标点以及单元格的像素大小来计算单元的物理坐标（以像素计，边框宽 1px），根据物理坐标在窗口中用不同的颜色绘制每个单元格以及墙壁。

每次在生成前，默认每个迷宫单元格并未打通，所以将文件中储存的贴图加载到全局画板上（类型为 IMAGE，定义在库函数中），再将全局画板显示在迷宫迷宫展示界面上（点的绘制，窗体的创建等由库函数实现）该功能在 DRAW_RAW_MAZE()中实现。

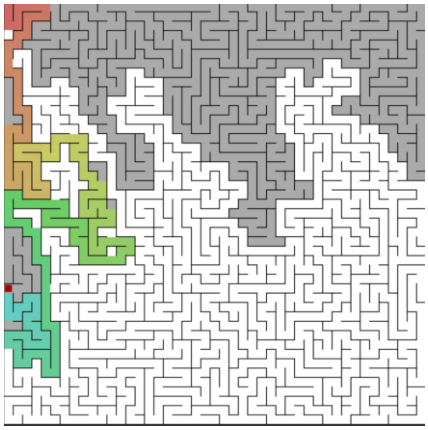
之后每一帧的绘制全全局画板上操作（除了表示当前位置的小方块）。 以一定的延迟将每一帧的图像显示在窗口的迷宫展示界面上，即形成动画效果。一个算法展示动画播放完成后将所得的结果以 png 格式保存，在文件名中记录迷宫的信息方便后期处理。

下图为迷宫画布：



2、颜色说明

为了绘制迷宫，需规定表示迷宫不同部分的颜色。 在路径的绘制中，为了强化视觉效果，设置一全局变量来记录当前路径的长度，并利用 HSL 模型来改变路径颜色的色相。效果如图。



```
#define  BKCOLOR       WHITE          //背景色为白色
#define  BLOCKCOLOR    0x0000AA       //小方块的颜色为红色
#define  VISITEDCOLOR      LIGHTGRAY     //表示搜索范围的颜色，为浅灰色
#define  DEGREE_CYCLE      360           //颜色变化的周期，对应 HSL 中色相的周期
#define  TRACECOLOR        (HSLtoRGB((float)PATH_COLOR_DEGREE,0.5,0.6f))
                                         //路径的颜色，为彩虹色
```

3、绘制操作

为了方便描述，定义了枚举类型表示绘制时的类型

```
enum DRAW_MODE { PATH_MODE, VISITED_MODE, CONNECT_MODE };
```

//PATH_MODE（路径模式，绘制颜色为 TRACECOLOR），VISITED_MODE（已访问模式，表示该块已访问，绘制颜色为 VISITEDCOLOR），CONNECT_MODE（连通模式，在生成迷宫时打通墙壁 绘制颜色为 BKCOLOR） 主要操作可归类为以下三种类型：

（1）绘制矩形

```
void DRAW_CELL          (POINT cur);                    //绘制一个单元格，颜色由所调用的函数决定
void DRAW_BLOCK         (POINT cur);                    //绘制一个显示当前位置的小方块，便于展示
```

（2）绘制墙（即边：由两个边组成的结构体类型）

```
void DRAW_CONNECTED_WALL(MAZE_EDGE edge);         //去除墙 e,用于生成迷宫
void DRAW_PATH          (POINT cur, VEC vec, DRAW_MODE m);  //在 cur 位置的 vec 方向的墙上以 m 模式绘制
```

（3）其他

```
void DRAW_VISITED (POINT cur, VEC vec,DRAW_MODE m); //以 m 模式绘制当前位置，以及在 vec 方向的墙上， 表示从 cur 位置向 vec 方向移动，
void DRAW_POSITION(POINT cur);                      //用表示访问的颜色覆盖当前点位置（不含墙）
```

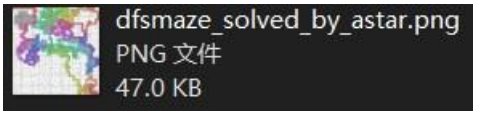
4、算法信息显示及图片保存 定义枚举类型表示当前正在进行的算法，为简化操作定义两个全局变量来记录信息。

```
enum GEN_MODE {NO_GEN,KRUSKAL, PRIM, DFS_GEN, RECUR};  //当前生成迷宫用到的算法标签
enum EXP_MODE {NO_EXP,DFS_EXP,BFS,ASTAR};              //当前解决迷宫用到的算法标签
```

每次调用不同的算法同时改变全局变量，根据全局变量的值，将不同的算法标签贴图显示在全局信息画板上（类似迷宫画板）再将信息画板显示在信息界面上。

保存图片时，生成迷宫的结果直接以算法名来命名。解决迷宫的的图片需要根据全局变量来确定文件名，存入文件夹 pic 中。并通过字符串操作来得到文件名。形式为生成算法名+寻路算法名，存入 pic 下的子 文件夹 solved_maze 中。

下图是两个例子，左为生成算法，右为解决迷宫的算法。



3.5 实现细节说明

3.5.1 主函数说明

```
int main() {

    MAZE MAZE;
    init_maze_system(&MAZE);                        //初始化迷宫

    generate_maze_recur(MAZE);                  //使用递归分割生成迷宫
    explore_maze_astar(MAZE);                   //寻路
    explore_maze_bfs(MAZE);
    explore_maze_dfs(MAZE);

    generate_maze_krus(MAZE);                   //使用 Kruskal 生成迷宫
    explore_maze_astar(MAZE);                   //寻路
    explore_maze_bfs(MAZE);
    explore_maze_dfs(MAZE);

    generate_maze_prim(MAZE);                   //使用 Prim 生成迷宫
    explore_maze_astar(MAZE);                   //寻路
    explore_maze_bfs(MAZE);
    explore_maze_dfs(MAZE);

    generate_maze_dfs(MAZE);                    //使用 DFS 生成迷宫
    explore_maze_astar(MAZE);                   //寻路
    explore_maze_bfs(MAZE);
    explore_maze_dfs(MAZE);

    end_maze_system(MAZE);                      //结束迷宫系统，关闭窗口，回收内存
    return 0;
```

}

声明一个迷宫类型对象 MAZE，调用接口初始化该迷宫，对其分配内存（由于需要改变指针的值，所以传递的是 MAZE 的地址，实质上是一个 MAZE_CELL 的三级指针）。初始化图形界面，**并将迷宫边界的墙全部设置为－1 表示不可打通**。

然后通过传值的方式来生成迷宫，将迷宫数据储存在 MAZE 中，调用不同的寻路算法来显示寻路过程。最后调用结束系统函数来关闭窗口，并回收分配给迷宫的内存。

3.5.2 Kruskal 算法生成迷宫中并查集的实现

1、初始化并查集

由函数 init_ufset 完成，由于并查集是以二维数组的形式来存储，所以在分配内存时需要先分配行指针，再分配列指针。将每个节点的根节点设为本书，并记子孙节点数目为零。具体实现如下

```
//分配行指针
    (*set) =(ufset_node**)malloc(sizeof(ufset_node*)*MAZE_LENGTH);
    if (!*set) exit(-1);
    for (int i = 0; i < MAZE_LENGTH; i++) {
        //分配列指针
        (*set)[i] =(ufset_node*)malloc(sizeof(ufset_node)*MAZE_HEIGHT);
        if (!(*set)[i]) exit(-1);
        for (int j = 0; j < MAZE_HEIGHT; j++) {
            (*set)[i][j].parent = { i,j };
            (*set)[i][j].rank = 0;
        }
    }
```

2、递归求根节点

基本思想：根节点的根节点为其本身（由初始化决定），给出如下的递归实现。

```
POINT get_maze_ufset_root(const maze_ufset set, POINT p) {
    if (set[p.x][p.y].parent.x == p.x && set[p.x][p.y].parent.y == p.y) return p;
    else return get_maze_ufset_root(set, set[p.x][p.y].parent);
}
```

3、合并两个并查集

基本思想：修改其中一个根节点的 parent 值指向另一个根节点。为了防止并查集退化为线型结构，利用每个根节点的子孙节点树来进行启发式合并，即将子孙节点少的根节点并入。

```
//合并两个等价类
void joint_maze_ufset(const maze_ufset set, POINT r1, POINT r2) {
    if (r1.x == r2.x && r1.y == r2.y) return;
    if (set[r1.x][r1.y].rank > set[r2.x][r2.y].rank) {
        set[r2.x][r2.y].parent = r1;
        set[r1.x][r1.y].rank += set[r2.x][r2.y].rank;
    }
    else {
        set[r1.x][r1.y].parent = r2;
        set[r2.x][r2.y].rank += set[r1.x][r1.y].rank;
    }
```

}

4、实现界面

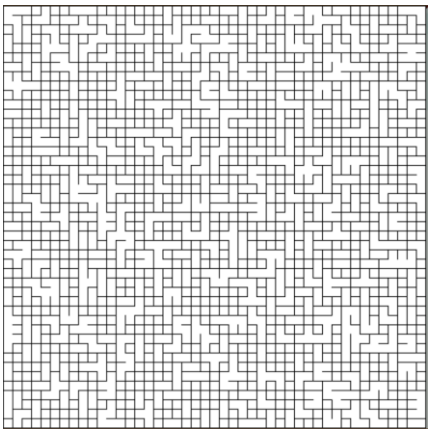
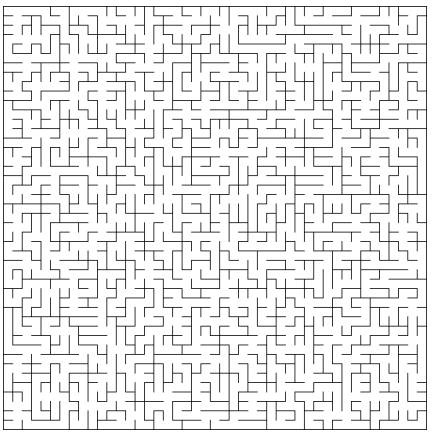


图 1：正在生成



图二：生成完毕

3.5.3 A*算法寻路中堆的实现

1.实现堆的插入：

该功能用 void insert()函数实现，其基本结构如下

```
void insert(MH *mh, MAZE_CELLA *ma) {
    mh->data[++(mh->count)] = ma;
    AdjustUp(mh, mh->count);
}
```

其中 count 为当前堆中的节点个数，AdjustUp（）使节点上移到合适的位置，其基本实现如下：

```
void  AdjustUp(MH *mh, int k) {
    while (1 < k && *(mh->data[k / 2]) > *(mh->data[k])) {
    MAZE_CELLA *temp = mh->data[k / 2];
    mh->data[k / 2] = mh->data[k];
    mh->data[k] = temp;
    k /= 2;
    }
```

```
}
```

2.实现堆的删除：

该功能用 MAZE_CELLA* erase()函数实现，其基本结构如下

```
void erase(MH *mh) {
    mh->data[1] = mh->data[(mh->count)--];
    AdjustDown(mh, 1);
}
```

其中 AdjustDown()函数使删除的节点下移至 count+1 的位置，并使新的节点替代原来的节点，其基本实现如下

```
void AdjustDown(MH *mh, int k) {
while (k * 2 <= mh->count) {
    int j = k * 2;
    //找当前层最小的元素
    if (j + 1 <= mh->count&&*(mh->data[j + 1]) < *(mh->data[j]))
        ++j;
    if (*(mh->data[k]) > *(mh->data[j])) {
        MAZE_CELLA *temp = mh->data[k];
        mh->data[k] = mh->data[j];
        mh->data[j] = temp;
    }
    k = j;
    }
}
```

3.查找堆中有无指定节点，作为 A*函数中把节点插入 open 中的判定条件

该功能用 bool count()函数实现，其基本结构如下

```
bool count(MH *mh, MAZE_CELLA ma) {
    for (int i = 1; i <= mh->count; i++) {
        if (ma == *(mh->data[i]))
            return 1;
    }
    return 0;
}
```

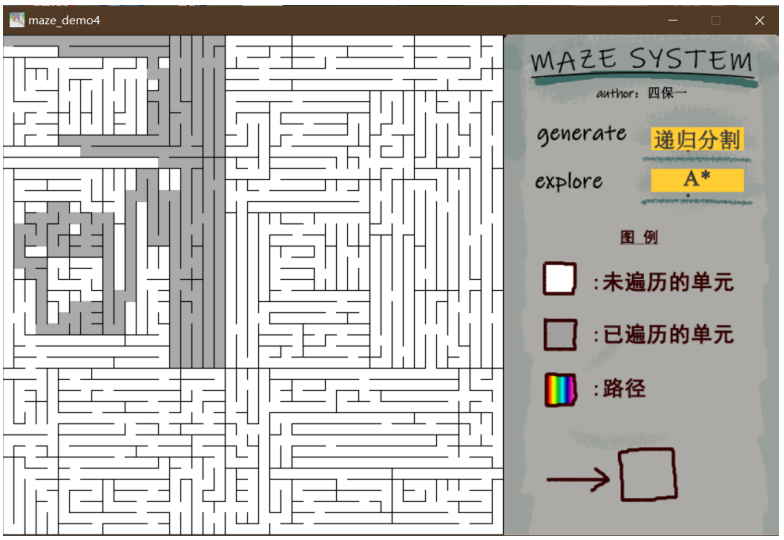
查找到了返回 1，失败则返回 0

3.6 测试数据及测试结果

测试数据：无

测试结果：如图

完整界面



每种算法的结果附在报告最后（图片难以展示完整的过程，如果允许，烦请老师移步 maze_system.exe）

4. 总结

4.1 系统特点及创新

4.1.1 系统特点

使用动画形式来呈现整个迷宫的生成以及求解过程，算法可视化，形式上直观明了。

4.2 遇到的问题及解决思路

4.2.1 动画播放时，图片无法稳定刷新

原因：控制动画播放的函数起初使用的是 Sleep 函数，延迟间隔随电脑性能的改变而改变。

解决：使用标准库 time.h，写出如下的延迟函数，问题解决。

```
void MAZE_DELAY(int fps) {      //根据帧率延迟

    clock_t last, present;      //初始化时钟
    last = clock();
    double interval = (double)1 / fps;

    present = clock();          //通过延时来控制刷新率
    while ((present - last) < (clock_t)CLOCKS_PER_SEC*(interval)) present = clock();
    last = present;
}
```

5. 本作的创新点

1.我们的迷宫寻路系统没有仅限于单种的生成、寻路算法，根据课本以及网上查阅的资料，我们设计了 4 种生成算法，3 种寻路算法，组合起来共有 12 种方案。在程序设计的过程中，设计了丰富的函数接口，

使我们程序的拓展性较强，可以容纳各种算法。

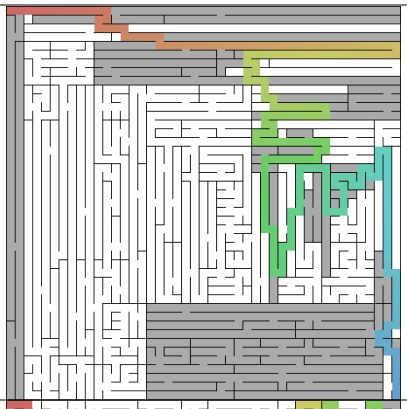
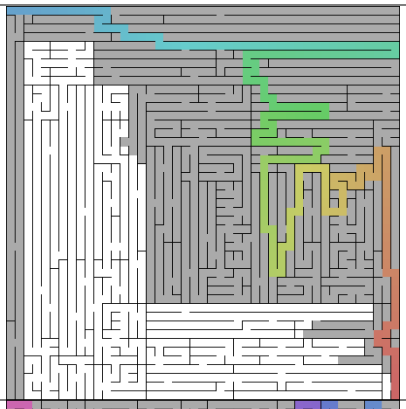
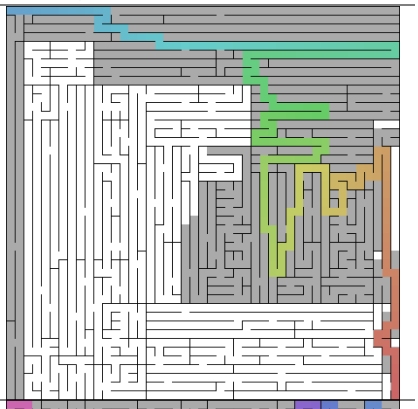
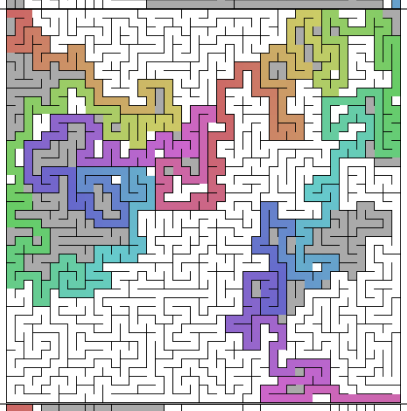
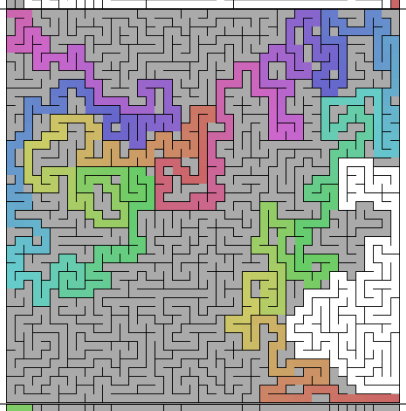
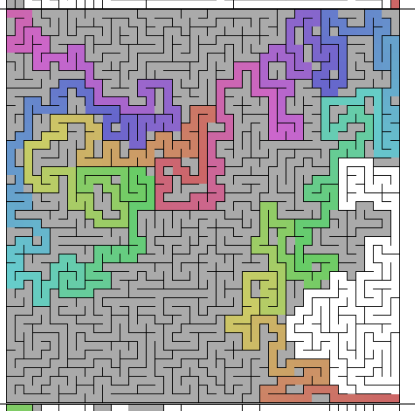
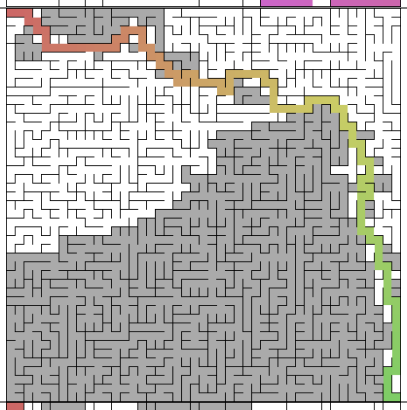
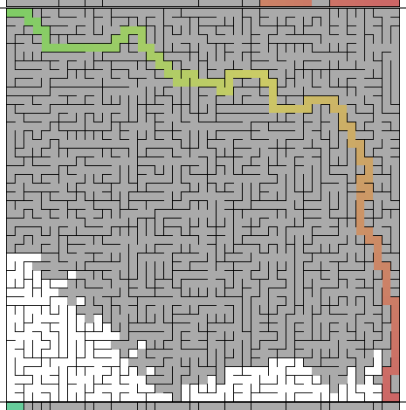
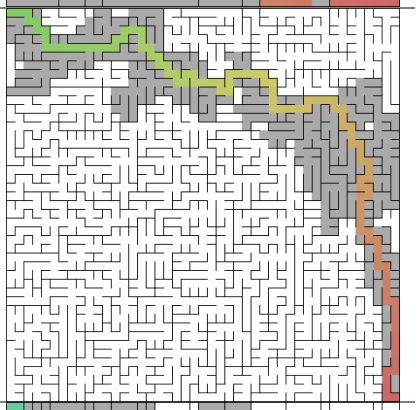
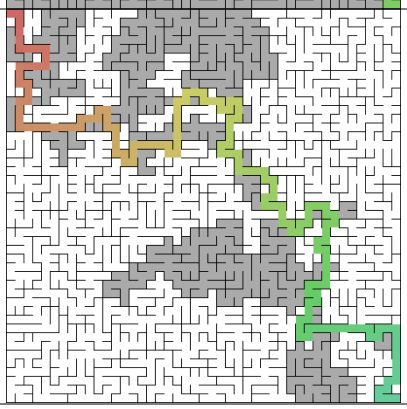
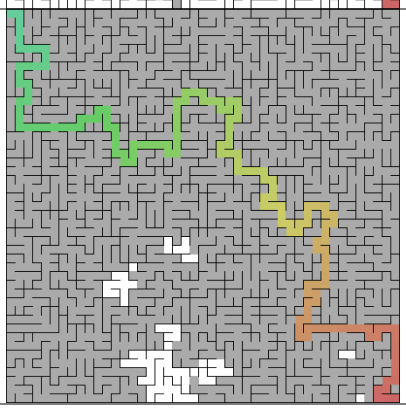
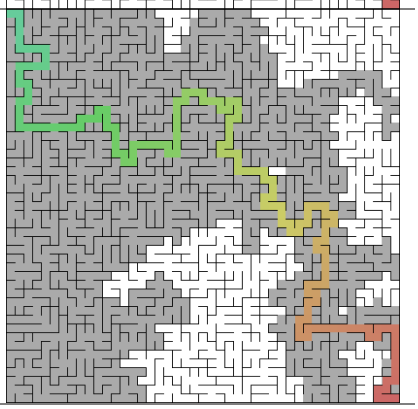
2.在观察程序运行结果后，我们发现 prim 算法所生成的迷宫路径相对于其他迷宫， 与起点与终点的连线重合度很高，针对这一特性，我们尝试基于现有算法进行改进，尝试用更少的遍历次 数来找到路线。经过简单试验我们发现，可以在 DFS 算法中，令算法尽可能向下向右走，尽可能拟合对角 线来调整（即优先选择下方与右边来前进）。

对已有的 DFS 算法进行微调，循环调用函数用 prim 来生成迷宫，使用调整前与调整后的算法来寻路，记录每次遍历次数的总和，求改进后所用总遍历次数与 改进前的比值，结果如下表。

实验次数	10	100	1000
遍历方格数比例（改进后：改进前）	0.511	0.721	0.708

可以看出，平均情况下，改进后的遍历次数要略小于改进前的次数。这说明我们的改进是有效的。当然仅仅是依靠优先向下向右走来拟合起点与终点的连线是不够的。时间所迫，未能作出更多有效的改进。

附：运行结果

	DFS 寻路	BFS 寻路	A*寻路
递归分割生成			
DFS 生成			
Prims 生成			
Kruskal 生成			

分析

表格中的图片很好的反映出了每个算法的特点。基于现有的结果，我们可以从不同角度分析，简单得出以下结论

纵向对比

1、基于深度优先的迷宫，路线最为曲折，分支最少，且有一条明显的主路。

2、两种 MST 算法的迷宫，路线最不曲折，与起点终点连线重合度最高，但分支极　　多，迷宫难度很大。

3、递归分割生成的迷宫，有大量的直线路线，分支较少，路线曲折程度一般。

横向对比

综合来看，A*算法与 DFS 算法表现最好。在面对 DFS 生成的迷宫时，A*算法的 表现不佳，甚至有时会退化成 BFS 算法；而 DFS 寻路算法在这类迷宫上的表现明显　　优于 A*算法。

在其他的迷宫中，A*算法的综合表现要略优于深度优先搜索（只考虑遍历的单元数，即图中的灰色部分）。广度优先搜索遍历的单元最多，与预期一致，且在 Prim 算法与 kruskal 算法生成的迷宫中，常常要遍历完所有的单元才能找到路线（因为这两种生成树生长的最为均匀）。