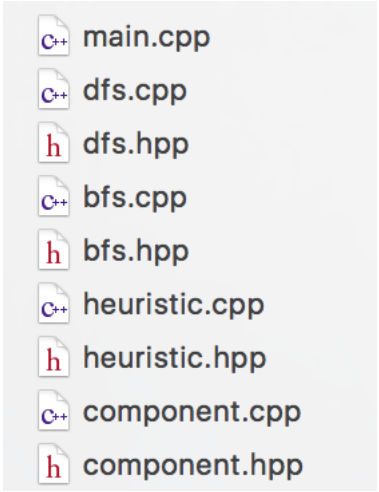
实验报告

1. 实验环境

MacOS, C++

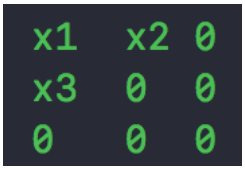
2. 代码结构说明

代码结构如下图所示, dfs.cpp, bfs.cpp, heuristic.cpp 三个文件分别实现了三种策略, component.cpp 中实现了实验需要的组件, main.cpp 中测试了三种方法的结果。



3. 问题求解

通过对题目的分析，我发现其实只需要确定九宫格中的三个值，就可以确定整个九宫格，所以我们只需要搜索这三个值即可。在实验中，这三个值我选定为下图所示的 x1, x2, x3。最终的搜索路径也可以用这三个值来表示。



然后我定义了一个结构体来表示每个搜索状态：

```
struct State{
    //储存了x1, x2, x3
    int state[4];
    //搜索的层数
    int layer;
    //搜索长度
    int searchLength;
    //记录1-9是否使用过
    bool visit[10];
    State(){
        memset(state, 0, 4 * sizeof(int));
        layer = 0;
        searchLength = 0;
        memset(visit, 0, 10 * sizeof(bool));
    }
};
```

对于具体代码来说，三个策略的框架完全一样，不同的是 dfs 使用栈，bfs 使用队列，heuristic 使用最小优先队列。代码意思的说明可以参考我的注释。

启发式搜索设计：
我定义了两个距离：
dis 是一个八元组
(第一行和，第二行和，第三行和，第一列和，第二列和，第三列和，左上到右下，右上到左下)
L1 距离是与 U = (15, 15, 15, 15, 15, 15, 15, 15)的曼哈顿距离，L2 距离是与 U 的距离的平方和。

```
//启发式策略：距离使用曼哈顿距离
int L1(int *dis){
    int len = 0;
    for (int i = 0; i < 8; i++){
        len += fabs(dis[i] - 15);
    }
    return len;
}

//启发式策略：距离使用平方和
int L2(int *dis){
    int len = 0;
    for (int i = 0; i < 8; i++){
        int t = dis[i] - 15;
        len += fabs(t * t);
    }
    return len;
}
```

然后为了使用最小优先队列，我重载了< 操作符

```
//最小优先队列，定义了两个状态哪个离最终状态最近
bool operator < (State a, State b){
    //函数指针，可以是L1或L2
    int (*ptr)(int *);
    if (isL1 == true){
        ptr = L1;
    }else{
        ptr = L2;
    }
    restore(a.state);
    computeDis(dis);
    int len1 = (*ptr)(dis);
    restore(b.state);
    computeDis(dis);
    int len2 = (*ptr)(dis);
    return len1 > len2;
}
```

当 x1, x2, x3 均不为 0 时，restore 函数会将 map 还原，当三个值中有任意一个为零时，restore 函数不会将 map 还原，即其他值依然为 0。然后计算得出当前 dis, 通过 L1 或 L2 计算当前距离。最小优先队列以这个距离为指标排列。

4. 实验结果

具体搜索路径和所有结果记录在 result.txt 中。

- dfs:

最快搜索距离：114

- bfs

最快搜索距离：179

- heuristic

L1: 最快搜索距离：15

L2: 最快搜索距离：15

5. 效率分析

由结果可以看出 heuristic 的两种距离定义得到的最快路径长度都是 15， dfs 114 次之，bfs179 最慢。