图 3.11: 建图。(a) 最小转弯问题构图。(b) 最小转弯问题的改进构图。(c) 带钥匙的迷宫问题构图。

可以在权上作文章。在种子填充中，图的每条边长度都为1，表示从一个格子走到相邻格子将使路径长度增加1。而在本题中，从一个格子走到相邻格子**不一定**会让转弯次数增加1：和当前方向相同则不变，不同则转弯次数增加1。这提示我们把一个格子拆成四个，代表四个方向，如图 3.11(b)。灰色的四个格子对应于原图中的同一个格子。

其中灰色边的权为0，黑色边的权为1。点数变为了原来的四倍，而每个格子的出度仅为3（没有180度转弯），总边数为$12mn$，比方法一少了一个数量级。可新的问题产生了：这个图的权有的为1有的为0，无法直接进行宽度优先搜索。解决方法是设计两个队列，读者可以试一试。可以证明：结点扩展是按照路径长度从小到大的顺序，和原始的宽度优先搜索一样。

小测验

1. 为什么说种子填充是特殊的图遍历？
2. 如何把前向星表示转化成邻接矩阵？
3. 如果只允许右转，如何求出两点间的最少转弯次数？

3.2 常用数据结构

本节介绍一些常用数据结构，包括二叉堆、并查集、哈希表和排序二叉树。这些数据结构代码简单、使用方便且时间效率高，被广泛的应用到问题求解中。希望读者熟练掌握每一种数据结构的实现，并体会其思想。很多高级数据结构都是从它们出发变化而来的。

3.2.1 二叉堆

二叉堆(binary heap) 是本节介绍的第一个高级数据结构，它的思想简单，代码短，然而用处非常大。二叉堆一般被用来实现**优先队列(priority queue)**，如图3.12所示。

换句话说，优先队列应该提供以下两种操作：

.

图 3.12: 优先队列的功能

Insert(S, x)：把元素x插入到优先队列中。

deleteMin(S, x)：从优先队列中取出最小元素x并从队列中删除。

优先队列经常被用来实现贪心算法（见第四章）和离散时间模拟系统，是一个基础数据结构。STL专门提供了 priority_queue 容器适配器，读者可以在学习本节后比较它和自己的二叉堆实现法的效率。

二叉堆是一棵完全二叉树，即所有叶子在同一层，且集中在左边。二叉堆满足**堆性质(heap property)**：根的值在整棵树中是最小的。不仅如此，根的两棵子树分别构成堆。图 3.13是一个堆。

.

图 3.13: 一棵有14个结点的堆

用 heap 数组来表示各个元素，则根是 heap[1]，最后一个元素是 heap[n]（heap[0] 不使用）。k 的左儿子是 $2k$，右儿子是 $2k+1$，父亲是 $\lfloor k/2 \rfloor$。

删除最小值(deleteMin) 先用最后一个元素代替根，如图 3.14(a)。

.

图 3.14: 堆的删除最小值操作

由于这一步会导致根的元素比儿子大，因此需要向下调整。向下调整的方法很简单，就是把根和较小儿子做比较，如果根比较大，则交换它和儿子，如图 3.14(b)。算法好比石沉大海，因此有的书把它称为sink过程。代码如下：

```
void sink(int p)
{
    int q=p<<1, a = heap[p];
    while(q<=hs)
    {
        if(q<hs && heap[q+1]<heap[q]) q++;
        if(heap[q]>=a) break;

        heap[p]=heap[q];
        p=q;
        q=p<<1;
    }
    heap[p] = a;
```

```
}

int deleteMin()
{
    int r=heap[1];
    heap[1]=heap[hs--];
    sink(1);
    return r;
}
```

插入(insert) 插入是类似的，先插入到最后，然后向上调整。向上调整只需要比较结点和父亲，比向下调整更容易。算法好比从海底游回水面，因此称为swim过程。如图 3.15。

图 3.15: 堆的插入操作

代码如下：

```
void swim(int p)
{
    int q = p>>1, a = heap[p];
    while(q && a<heap[q])
    {
        heap[p]=heap[q];
        p=q;
        q=p>>1;
    }
    heap[p] = a;
}

void insert(int a)
{
    heap[++hs]=a;
    swim(hs);
}
```

删除任意元素(delete) 删除任意元素类似于删除最小元素，先用最后一个元素代替被删除元素，再进行调整。不过值得注意的是，最后一个元素被交换后可能要向下调整，也有可能要向上调整。具体实现可参照以上两段代码。

建立(build) 给n个整数，如何构造一个二叉堆？可以一个一个插入，但是有更好的方法：从下往上一层一层向下调整。由于叶子无需调整，因此只需要从 [hs/2] 开始递减到1。由数学归纳法可以证明：循环变量为i时，以$i+1$, $i+2$,..., n为根的树都是堆。代码如下：

```
void build()
{
    for(int i=hs/2;i>0;i--)
        sink(i);
}
```

时间复杂度 由于堆是完全二叉树，因此高度是O($\log n$)的。插入和删除最多从根到叶子，因此时间复杂度为O($\log n$)。建立操作计算起来稍微麻烦一些。高度为h的结点有$n/2^{h+1}$个，时间为(h)，对$0 \ldots \log n$的h求和，得O(n)。故：建立的时间复杂度为O(n)。

小测验

1. 描述二叉堆的数组表示法，证明树的高度为$\log n$
2. 为什么插入和删除只需要调整一条从根到叶子的路径而不会影响到其他？
3. 如果减小了一个结点的值（称为decreaseKey操作），需要对此结点进行怎样的调整，才能重新得到一个堆？

3.2.2 并查集

并查集(union-find set) 维护一些不相交的集合，它是一个集合的集合。每个元素恰好属于一个集合，好比每条鱼装在一个鱼缸里。每个集合S有一个元素作为“集合代表”rep[S]，好比每个鱼缸选出一条”鱼王”。并查集提供三种操作：

MakeSet(x)：建立一个新集合x。x应该不在现有的任何一个集合中出现。
Find(S, x)：返回x所在集合的代表元素。
Union(x, y)：把x所在的集合和y所在的集合合并。

森林表示法 可以用一棵森林表示并查集，森林里的每棵树表示一个集合，树根就是集合的代表元素。一个集合还可以用很多种树表示，只要树中的结点不变，表示的都是同一个集合。

合并操作 只需要将一棵树的根设为另一棵即可。这一步显然是常数的。一个优化是：把小的树合并到大树中，这样会让深度不太大。这个优化称为启发式合并。