基于搜索的算法设计策略

高级算法设计与分析

Teacher：张明卫
Email：zhangmw@swc.neu.edu
Office: 东北大学浑南校区信息楼B434

搜索策略

- 利用计算机的高性能来有目的的穷举一个问题的部分或所有的可能情况，从而求出问题的解的一种方法。
- 搜索过程实际上是根据初始条件和扩展规则构造一棵解答树并寻找符合目标状态的结点的过程。
- 分为深度优先搜索、广度优秀搜索和优化的搜索策略。

CONTENTS

PART 01 回溯法

1.1 回溯算法思想

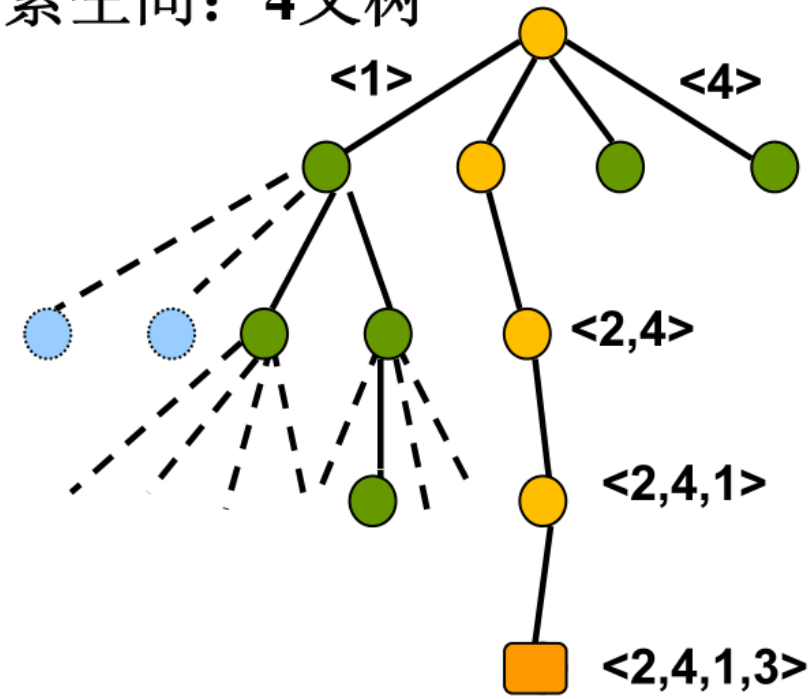
- 有许多问题，当需要找出它的解集或者要求回答什么解是满足某些约束条件的最佳解时，往往要使用回溯法。
- 回溯法的基本做法是搜索，或是一种组织得井井有条的，能避免不必要搜索的穷举式搜索法。这种方法适用于解一些组合数相当大的问题。
- 回溯法在问题的解空间树中，按深度优先策略，从根结点出发搜索解空间树。算法搜索至解空间树的任意一点时，先判断该结点是否包含问题的解。如果肯定不包含，则跳过对该结点为根的子树的搜索，逐层向其祖先结点回溯；否则，进入该子树，继续按深度优先策略搜索。

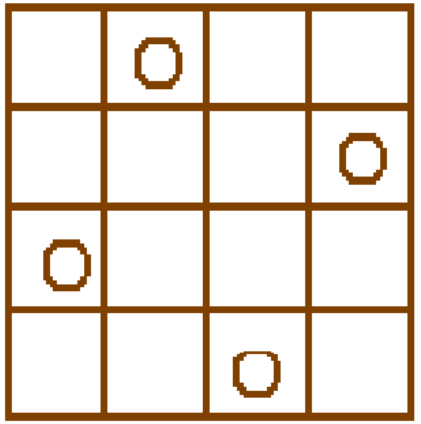
1.1 回溯算法思想

■ 示例1：n后问题：在一个n×n的棋盘上放置n个皇后，要求每两个之间都不能相互“攻击”，即使得任两个皇后都不在同一行、同一列及同一条斜线上。

4后问题：解是一个4维向量，$<x_1, x_2, x_3, x_4>$（放置列号）

搜索空间：4叉树

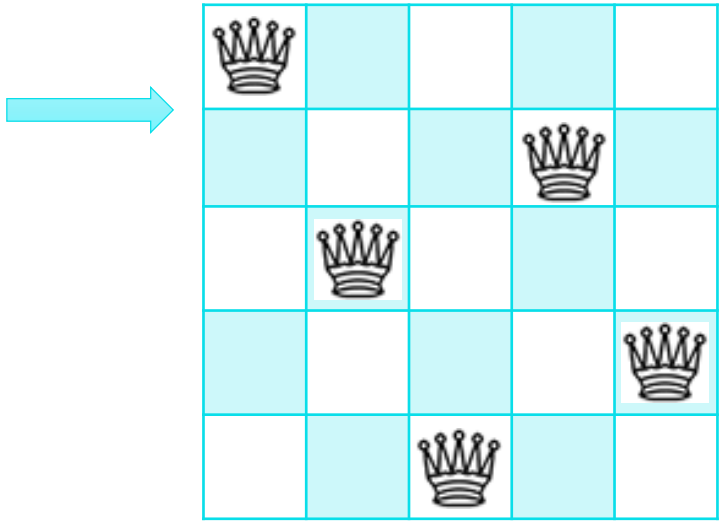




1.1 回溯算法思想

■ 5后示例

输入样例	输出样例
5	1 3 5 2 4 1 4 2 5 3 2 4 1 3 5 2 5 3 1 4 3 1 4 2 5 3 5 2 4 1 4 1 3 5 2 4 2 5 3 1 5 2 4 1 3 5 3 1 4 2 Total= 10



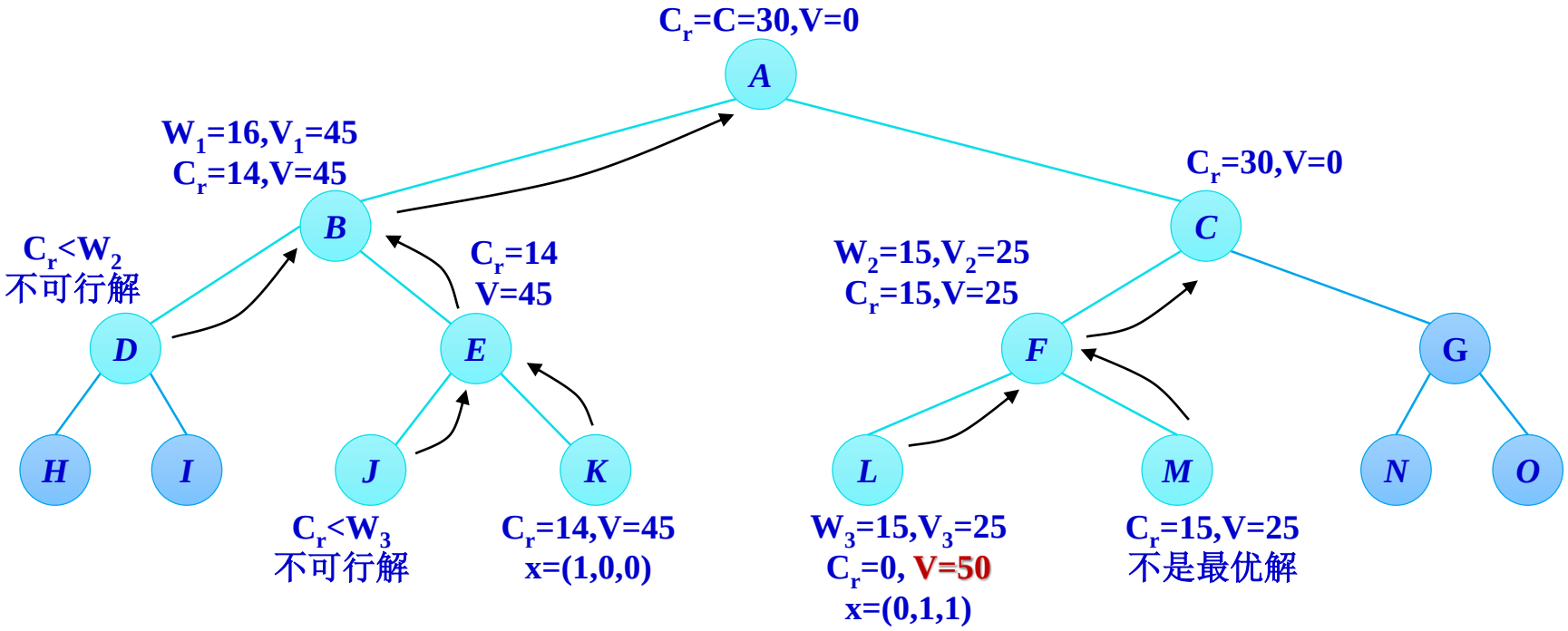
■ 8 后问题：解是一个8 维向量，$<x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8>$

◆搜索空间：8叉树 ，共92种解，一 个解为：<1,3,5,2,4,6,8,7>

1.1 回溯算法思想

■ 示例2：0—1背包问题

假设背包容量C＝30，w＝{16，15，15}，v＝{45，25，25}。



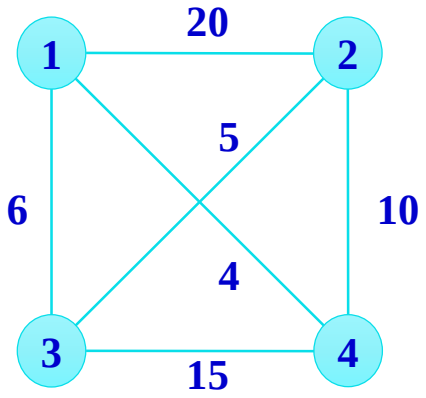
1.1 回溯算法思想

■ 示例2：0—1背包问题的解空间

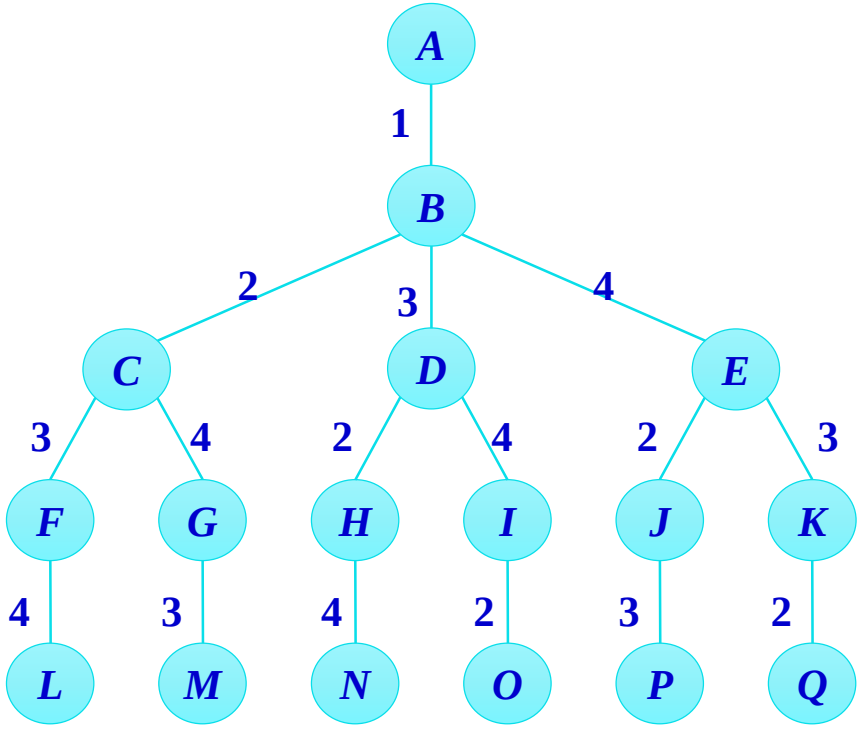
- 有时问题是要从一个集合的所有子集中搜索一个集合，作为问题的解。回溯算法可以很方便地遍历一个集合的所有子集。
- 当问题是要计算n个元素的子集，以便达到某种优化目标时，可以把这个解空间组织成一棵子集树。
- 例如，n个物品的0-1背包问题相应的解空间树就是一棵子集树。
- 这类子集树通常有2^n个叶结点，结点总数为$2^{n+1}-1$。
- 遍历子集树的算法，其计算时间复杂度是$\Omega(2^n)$。

1.1 回溯算法思想

■ 示例3：旅行商问题：回溯法找最小费用周游路线的主要过程：



（1，3，2，4，1）的总权值为25，为最优回路。



■ 示例3：旅行商问题的解空间

- 当所给的问题是确定n个元素满足某种性质的排列时，可以把这个解空间组织成一棵排列树。
- 例如，n个地点的旅行商问题相应的解空间树就是一棵排列树。
- 这类子集树通常有(n-1)!个叶结点。因此遍历排列树时，其计算时间复杂度是$\Omega(n!)$。

1.1 回溯算法思想

■适用问题：求解搜索问题和优化问题；

■回溯法的概要步骤

1. 针对所给问题，定义问题的解空间；
2. 确定易于搜索的解空间结构；
3. 以深度优先方式搜索解空间，并在搜索过程中用剪枝函数避免无效搜索。

1.1 回溯算法思想

- 问题的解向量：回溯法希望一个问题的解能够表示成一个n元式(x_1, x_2, …, x_n)的形式。
- 显约束：对分量x_i的取值限定。
- 隐约束：为满足问题的解而对不同分量之间施加的约束。
- 解空间：对于问题的一个实例，解向量满足显式约束条件的所有多元组，构成了该实例的一个解空间。
  - ◆表示为树，结点对应部分解向量，树叶对应可行解；
  - ◆例如，对于有n种可选择物品的0—1背包问题，其解空间由长度为n的0—1向量组成，该解空间包含了对变量的所有可能的0—1赋值。
- 注意：同一个问题可以有多种表示，有些表示方法更简单，所需表示的状态空间更小（存储量少，搜索方法简单）。

1.1 回溯算法思想

■在生成解空间树时，定义以下几个相关概念：

◆活结点：如果已生成一个结点而它的所有儿子结点还没有全部生成，则这个结点叫做活结点。

◆扩展结点：当前正在生成其儿子结点的活结点叫扩展结点（正扩展的结点）。

◆死结点：不再进一步扩展或者其儿子结点已全部生成的结点就是死结点。

1.1 回溯算法思想

■在确定了解空间的组织结构后，回溯从开始结点（根结点）出发，以深度优先的方式搜索整个解空间。

◆这个开始结点成为一个活结点，同时成为当前的扩展结点。在当前的扩展结点，搜索沿深度方向进入一个新的结点。这个新结点成为一个新的活结点，并成为当前的扩展结点。

◆若在当前扩展结点处不能再向深度方向移动，则当前的扩展结点成为死结点，即该活结点成为死结点。此时回溯到最近的一个活结点处，并使得这个活结点成为当前的扩展结点。

◆回溯法以这样的方式递归搜索整个解空间（树），直至满足中止条件。

1.1 回溯算法思想

■在回溯法搜索解空间树时，通常采用两种策略（剪枝函数）避免无效搜索以提高回溯法的搜索效率：

1. 用约束函数在扩展结点处减去不满足约束条件的子树；
2. 用限界函数减去不能得到最优解的子树。

◆解0—1背包问题的回溯法用剪枝函数剪去导致不可行解的子树。

◆解旅行商问题的回溯算法中，如果从根结点到当前扩展结点的部分周游路线的费用已超过当前找到的最好周游路线费用，则以该结点为根的子树中不包括最优解，就可以剪枝。

1.1 回溯算法思想

- **回溯法的显著特征：**在搜索过程中动态产生问题的解空间。在任何时刻，算法只保存从根结点到当前扩展结点的路径。如果解空间树中从根结点到叶结点的最长路径的长度为h(n)，则回溯法所需的计算空间通常为O(h(n))。而显式地存储整个解空间则需要$O(2^{h(n)})$或O(h(n)!)的内存空间。

1.1 回溯算法思想

回溯算法的适用条件

设 $P(x_1, x_2, \ldots, x_i)$ 为真表示向量$<x_1, x_2, \ldots, x_i>$满足某个性质 (n后问题中i 个皇后放置在彼此不能攻击的位置）

多米诺性质：

$$P(x_1, x_2, \ldots, x_{k+1}) \rightarrow P(x_1, x_2, \ldots, x_k) \qquad 0 < k < n$$

例4　求不等式的整数解

$$5x_1+4x_2-x_3 \le 10, \quad 1 \le x_i \le 3, \ i=1,2,3$$

$P(x_1, \ldots, x_k)$：意味将$x_1, x_2, \ldots, x_k$代入原不等式的相应部分使得左边小于等于10

不满足多米诺性质

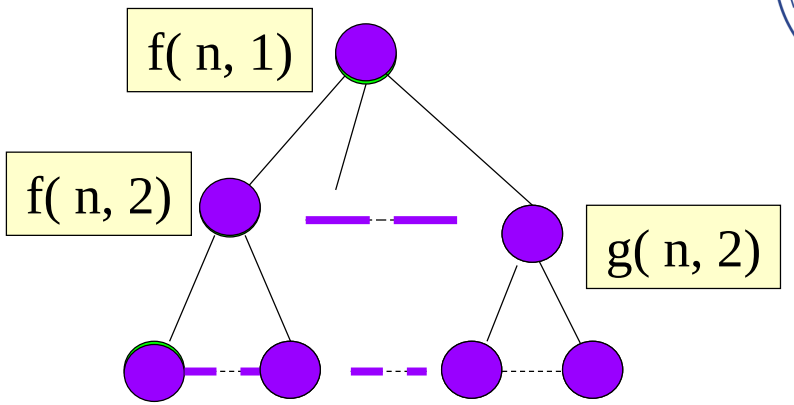
变换：　令 $x_3 = 3 - x_3'$，

$5x_1 + 4x_2 + x_3' \le 13 \ 1 \le x_1, x_2 \le 3 \ \ 0 \le x_3' \le 2$

1.1 回溯算法思想

- 回溯算法的递归实现：回溯法对解空间作深度优先搜索，因此，在一般情况下用递归方法实现回溯法。

```
void backtrack (int t)
{
    if (t>n) output(x);
    else
      for (int i=f(n,t); i<=g(n,t); i++) {
        x[t]=h(i);
        if (constraint(t)&&bound(t))
            backtrack(t+1);
        }
}
```



t-递归深度，即当前扩展结点在解空间树中的深度。
n-用来控制递归深度，即解空间树的高度。当t>n时，算法已搜索到一个叶结点
output(x)对得到的可行解x进行记录或输出处理。
f(n,t)和g(n,t)分别表示在当前扩展结点处搜索过的子树的起始编号和终止编号。
h(i)表示在当前扩展结点处x[t]的第i个可选值。
constraint(t)和bound(t)表示在当前扩展结点处的约束函数和限界函数。

1.1 回溯算法思想

- 回溯算法的迭代实现：采用树的非递归深度优先遍历算法，可将回溯法表示为一个非递归迭代过程。

```
void iterativeBacktrack () {
    int t=1;
    while (t>0) {
        if (f(n,t)<=g(n,t))
            for (int i=f(n,t);i<=g(n,t);i++) {
                x[t]=h(i);
                if (constraint(t)&&bound(t)) {
                    if (solution(t)) output(x);
                    else t++;
                }
            }
        else   t--;
    }
}
```

- **Solution(t)**判断当前扩展结点处是否已到问题的一个可行解。返回值为**true**表示在当前可扩展结点处**x[1:t]**是问题的一个可行解。若返回值为**false**则表示在当前扩展结点处**x[1:t]**只是问题的一个部分解，还需要向纵深方向继续搜索。
- **f(n,t)**和**g(n,t)**分别表示在当前扩展结点处未搜索过的子树的起始编号和终止编号。**h(i)**表示在当前扩展结点处**x[t]**的第**i**个可选值。

1.2 装载问题

■问题描述：

有一批共n个集装箱要装上2艘载重量分别为c_1和c_2的轮船，其中集装箱i 的重量为w_i，且

$$\sum_{i=1}^{n} w_i \le c_1 + c_2$$

装载问题要求确定是否有一个合理的装载方案可将这个集装箱装上这2艘轮船。如果有，找出一种装载方案。

例如，当$n = 3, c_1 = c_2 = 50$，且$w = [\ 10, 40, 40\ ]$，则可以将集装箱1和2装到第一艘轮船上，而将集装箱3装到第二艘轮船上；如果$w = [\ 20, 40, 40]$，则无法将这3个集装箱都装上轮船。

1.2 装载问题

■求解思路：

如果一个给定装载问题有解，则采用下面的策略可得到最优装载方案。

(1) 首先将第一艘轮船尽可能装满；

(2) 将剩余的集装箱装上第二艘轮船。

将第一艘轮船尽可能装满等价于选取全体集装箱的一个子集，使该子集中集装箱重量之和最接近。由此可知，装载问题等价于以下特殊的0-1背包问题。

$$\max \sum_{i=1}^{n} w_i x_i$$

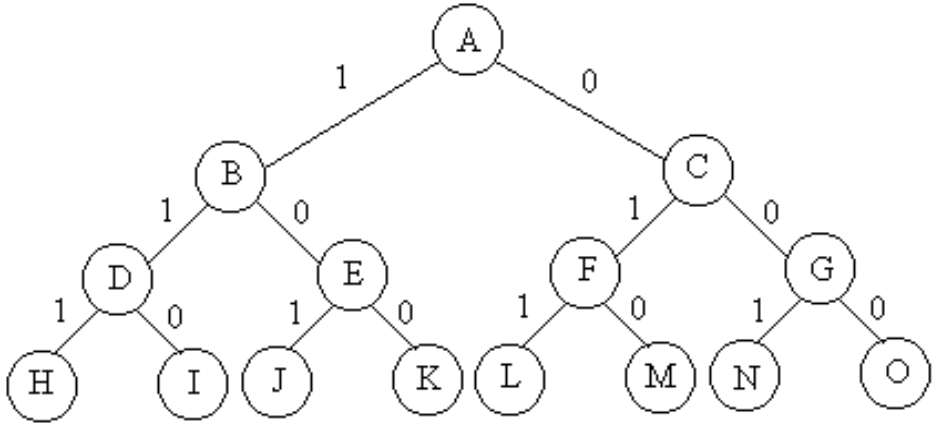
$$\text{s.t.} \sum_{i=1}^{n} w_i x_i \le c_1$$

$$x_i \in \{0,1\}, 1 \le i \le n$$

1.2 装载问题

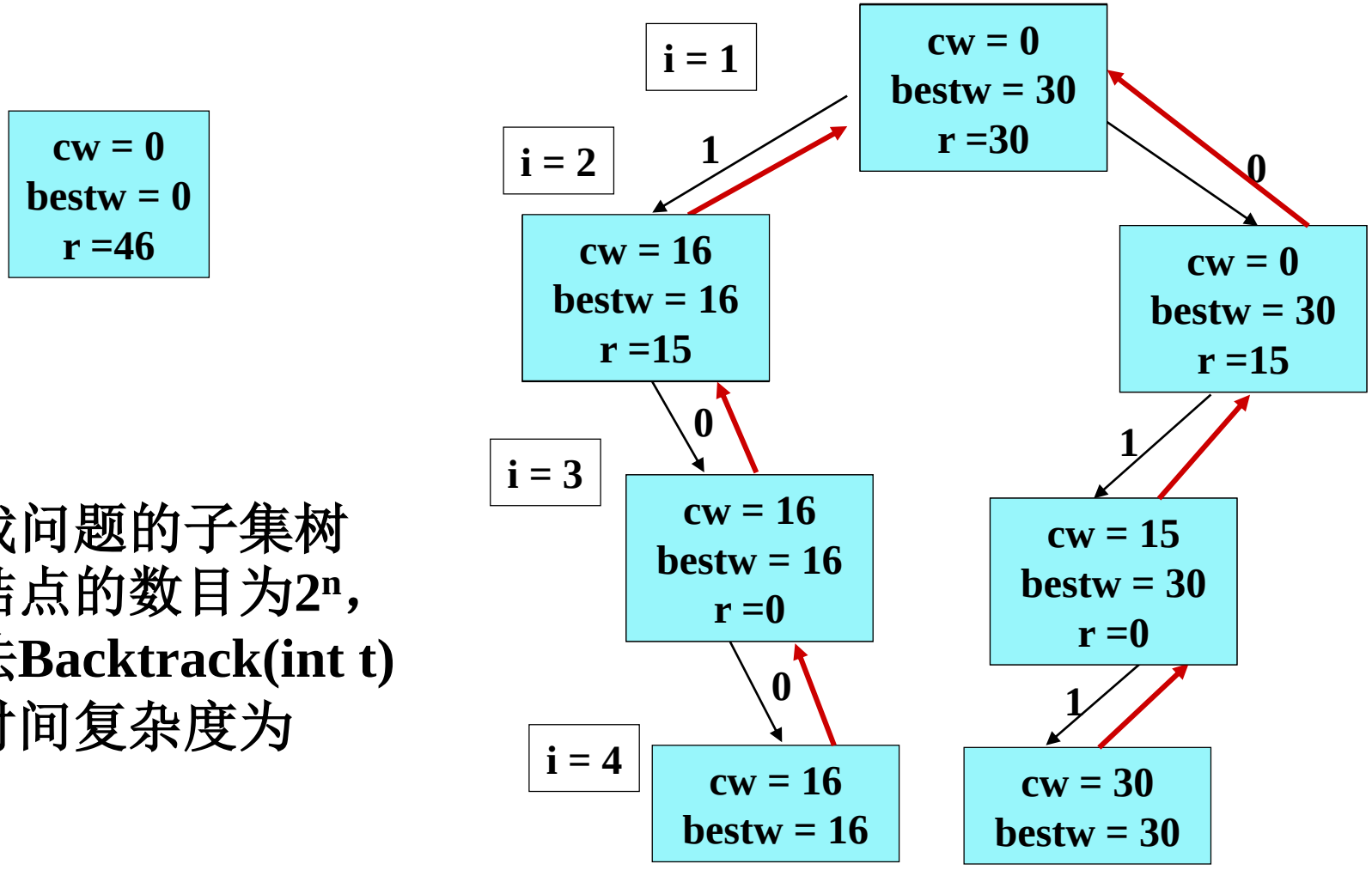
- 解空间：子集树
- 可行性约束函数(选择当前元素)：$\sum_{i=1}^{n} w_i x_i \le c_1$
- 上界函数(不选择当前元素)：
  - 当前载重量cw+剩余集装箱的重量r ≤ 当前最优载重量bestw;

```
private static void backtrack (int i)
  { // 搜索第i层结点
    if (i > n)  // 到达叶结点
                //更新最优解bestx,bestw;return;
    r -= w[i];
    if (cw + w[i] <= c) { // 搜索左子树
      x[i] = 1;
      cw += w[i];
      backtrack(i + 1);
      cw -= w[i];     }
    if (cw + r > bestw)  {
      x[i] = 0;  // 搜索右子树
      backtrack(i + 1);     }
    r += w[i];
  }
```



1.2 装载问题

- 运行实例：w={ 16, 15, 15}, c = 30;



- 由于装载问题的子集树中叶子结点的数目为2^n，因此算法Backtrack(int t)的计算时间复杂度为$O(2^n)$

1.3 图的m着色问题

- 问题：给定无向连通图G=(*V*, *E*)和*m*种不同的颜色，用这些颜色为图*G*的各顶点着色，每个顶点着一种颜色。是否有一种着色法使*G*中相邻的两个顶点有不同的颜色?
- 图着色的判定问题与优化问题：这个问题是图的*m*可着色判定问题。若一个图最少需要*m*种颜色才能使图中每条边连接的两个顶点着不同颜色，则称这个数*m*为该图的色数。求一个图的色数*m*的问题称为图的*m*可着色优化问题。
- 编程计算：给定图G=(*V*, *E*)和*m*种不同的颜色，找出所有不同的着色法和着色总数。

1.3 图的m着色问题

- 输入

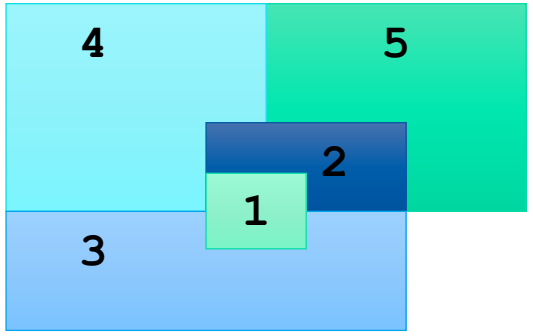
  第一行是顶点的个数n（$2 \le n \le 10$），颜色数m（$1 \le m \le n$）。

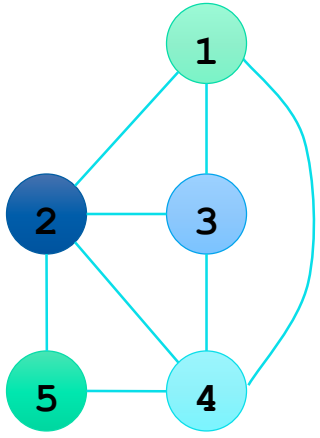
  接下来是顶点之间的相互关系：$a\ b$

  表示a和b相邻。当a，b同时为0时表示输入结束。

- 输出

  输出所有的着色方案，表示某个顶点涂某种颜色号，每个数字的后面有一个空格。最后一行是着色方案总数。

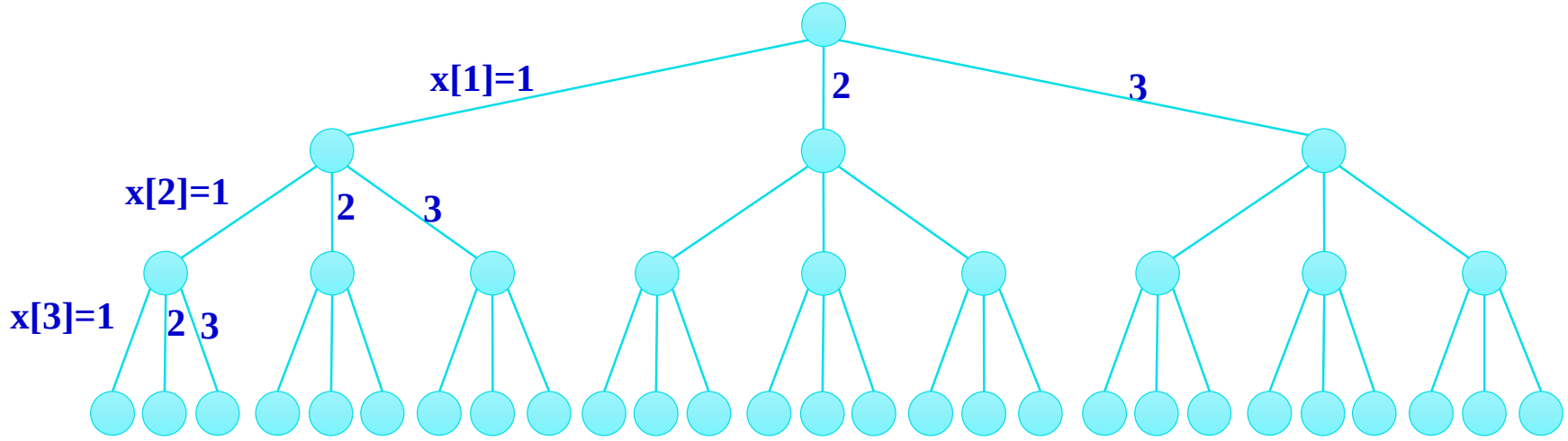




输入样例	输出样例
5 4 1 3 1 2 1 4 2 3 2 4 2 5 3 4 4 5 0 0	1 2 3 4 1 1 2 3 4 3 1 2 4 3 1 1 2 4 3 4 1 3 2 4 1 1 3 2 4 2 1 3 4 2 1 … 4 3 2 1 4 Total=48

1.3 图的m着色问题

- 对m种颜色编号为1，2，…，m，由于每个顶点可从m种颜色中选择一种颜色着色，如果无向连通图G=(V, E)的顶点数为n，则解空间的大小为m^n种，解空间是非常庞大的，它是一棵m叉树。
- 当n＝3，m＝3时的解空间树。



1.3 图的m着色问题

- 图的*m*着色问题的约束函数是相邻的两个顶点需要着不同的颜色，但是没有限界函数。
- 假设无向连通图G=(*V*, *E*)的邻接矩阵为*a*，如果顶点*i*和*j*之间有边，则a[*i*][*j*]=1，否则a[*i*][*j*]=0。
- 设问题的解向量为X ($x_1, x_2, \ldots, x_n$)，其中$x_i \in \{1, 2, \ldots, m\}$，表示顶点*i*所着的颜色是x[*i*]，即解空间的每个结点都有*m*个儿子。
- 数据结构的定义：

```
#define NUM 100
int n;                  //图的顶点数量
int m;                  //可用颜色数量
int a[NUM][NUM];        //图的邻接矩阵
int x[NUM];             //当前的解向量
int sum ;               //已经找到的可m着色的方案数量
```

1.3 图的m着色问题

■ 图的*m*着色回溯算法的实现：

```
void BackTrack(int t ){    //形参t是回溯的深度，从1开始
      int i;
      if( t > n ){       //到达叶子结点，获得一个着色方案
            sum ++ ;
            for(i=1; i<=n ;i++)
                  printf("%d ",x[i]);
            printf("\n");
      }
      else
        for(i=1; i<=m; i++ ){     //搜索当前扩展结点的m个孩子
                  x[t] = i;
                  if( NoSame(t) )    BackTrack(t+1);
                  x[t] = 0;
            }
}
```

1.3 图的m着色问题

- 检查与相邻结点的着色是否一样的约束函数：

```
//形参t是回溯的深度
bool NoSame(int t)
{
    int i;
    for(i=1; i<=t-1; i++ )
        if( (a[t][i] == 1) && (x[i] == x[t]))
            return false;
    return true;
}
```

1.3 图的m着色问题

- 算法BackTrack(int t)中，对每个内部结点，其子结点的一种着色是否可行，需要判断子结点的着色与相邻的n个顶点的着色是否相同，因此共需要耗时O(mn)，而整个解空间树的内部结点数是：

$$\sum_{i=0}^{n-1} m^i$$

- 所以算法BackTrack(int t)的时间复杂度是：

$$\sum_{i=0}^{n-1}\{m^i(mn)\} = nm\frac{m^n-1}{m-1} = O(nm^n)$$

1.3 图的m着色问题

■ 图着色问题的应用：

◆会场分配问题：

- 有n项活动需要安排，对于活动i,j，如果i,j时间冲突，就说i和j不相容。如何分配这些活动，使得每个会场的活动相容且占用会场数最小？
- 建模：活动作为图的顶点，如果i,j不相容，则在i和j之间加一条边，会场标号作为颜色标号。求图的一种着色方案，使得使用的颜色数最少。

◆四色猜想：1852 年，英国地图制图师弗朗西斯·古特里（Francis Guthrie）在观察地图时提出了一个“给地图着色”的问题。他发现只需要四种颜色就可以对地图进行着色，使得相邻的国家颜色不同。那这个猜想是否正确呢？经历了漫长的 120 多年，数学家们才将“四色猜想”变成“四色定理”。

回溯算法小结

- 回溯法（backtracking）是一种组织搜索的一般技术，基本做法是搜索，或是一种组织得井井有条的，能避免不必要搜索的穷举式搜索法，有“通用的解题法”之称，用它可以系统地搜索一个问题的所有解或任一解，既有系统性又有跳跃性。
- 为实现回溯，首先需要定义一个解空间（solution space），然后以易于搜索的方式组织解空间，最后用深度优先的方法搜索解空间，获得问题的解。
- 回溯法搜索解空间树时，通常采用两种策略避免无效搜索。其一，用约束函数在扩展结点出剪去不满足约束的子树；其二，用限界函数剪去得不到最优解的子树。这两类函数通称为剪枝函数。

PART 02　分支限界法

2.1 分支限界法的思想

与回溯法的异同

■相同点

◆与回溯法一样，分支限界也是搜索一个解空间，而这个解空间通常组织成一棵树（常见的树结构：子集树和排列树）。

◆ 解题步骤都基本一样的：定义解空间、确定解空间结构、深度/广度优先搜索 + 剪枝

■ 不同点

◆ 搜索策略：回溯以深度优先搜索树，而分支限界常常以广度优先或最小耗费(最大效益)优先的方法搜索问题的解空间树。

◆ 求解目标：回溯法的求解目标一般是找出解空间树中满足约束条件的所有解。分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出在某种意义下的最优解。

2.1 分支限界法的思想

分支界限法的要素 —— 限界函数

- 设立限界函数，其函数值是以该结点为根的搜索树中的所有可行解的目标函数值的上/下界（天花板）。限界函数一方面计算越简单越好，另一方法应该和最优解越接近越好。
- 记录已得的最优可行解，其值是当时已经得到的可行解的目标函数的最大/小值。
- 搜索中停止分支的依据：如果某个结点不满足约束条件或者其限界函数的得出的上/下界小/大于当时的最优可行解，则不再扩展该结点。
- 最优可行解的更新：如果目标函数值为正数，初值可以设为0。在搜索中如得到一个可行解，计算可行解的目标函数值，如果这个值优于当时记录的最优可行解，就将这个值记录作为新的解。

2.1 分支限界法的思想

分支界限法的要素 —— 活结点表

- 不同于回溯法，在分支限界法中，每一个活结点只有一次机会成为扩展结点。活结点一旦成为扩展结点，就一次性产生其所有儿子结点。在这些儿子结点中，通过剪枝函数将导致不可行解或导致非最优解的儿子结点舍弃，其余儿子结点被加入活结点表中。
- 此后，从活结点表中取下一结点成为当前扩展结点，并重复上述结点扩展过程。这个过程一直持续到找到所需的解或活结点表为空时为止。

2.1 分支限界法的思想

活结点表中下一个扩展结点的选择

■从活结点表中选择下一个活结点作为新的扩展结点，分支限界算法通常可以分为两种形式：

◆FIFO（First In First Out）分支限界算法：按照先进先出（FIFO）原则选择下一个活结点作为扩展结点，即从活结点表中取出结点的顺序与加入结点的顺序相同。

◆最小耗费或最大收益分支限界算法

- 在这种情况下，每个结点都有一个耗费或收益；
- 根据问题的需要，可能是要查找一个具有最小耗费的解，或者是查找一个具有最大收益的解；
- 在选择扩展结点时根据活结点的优先权来选择。

2.1 分支限界法的思想

分支界限法的特点

- 分支限界法适用于组合优化中求最优解的问题
- 时间性能：最坏的情况要搜索整个解空间，复杂度是指数型。但如果启发式信息强且剪枝处理得当，平均性能往往很好。
- 空间性能：优先队列往往需要较大的空间开销。

2.1 分支限界法的思想

分支界限法的步骤

- 针对所给问题，定义问题的解空间；
- 对解空间进行组织，确定易于搜索的解空间结构（树或图）
- 设计合适的限界函数
- 选定节点扩展策略，组织活结点表，使用限界函数来避免那些不能得到解的子空间。

2.2 装载问题

- 集装箱装载问题：要求确定在不超过轮船载重量的前提下，将总重量尽可能大的集装箱装上轮船。具体问题定义见1.2节。
- 解空间：子集树
- 可行性约束函数：$\sum_{i=1}^{n} w_i x_i \le c_1$
- 上界函数：**ub**=扩展结点的当前载重量**cw**+剩余集装箱的重量**r**
  - 当**ub** ≤ 当前最优载重量**bestw** 时即可剪枝。
- 输入样例

  80 4

  18 7 25 36
- 输出样例

  79

2.2 装载问题

- 定义一个先进先出（FIFO）队列*Q*，初始化队列时，在尾部增加一个-1标记。
- 这是一个分层的标志，当一层结束时，在队列尾部增加一个－1标志。
- 定义扩展结点的当前载重量为*cw*，剩余集装箱的重量为*r*，当前最优载重量为*bestw*，轮船的载重量为*c*＝80。

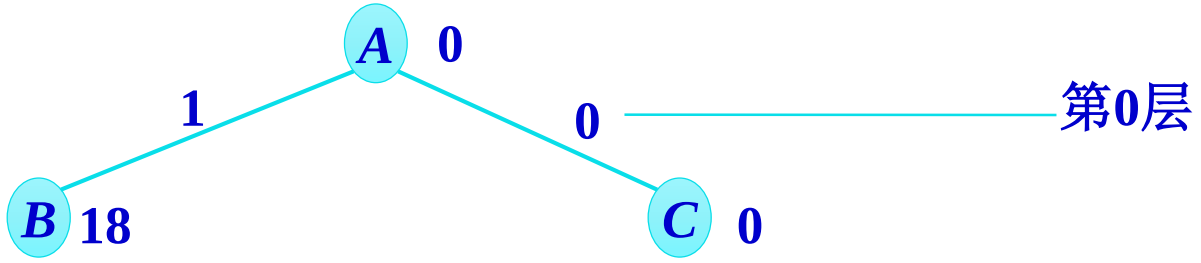
Q.front()←→Q.back()						
-1						

2.2 装载问题

■算法从子集树的第0层开始展开。

◆第0层即集装箱0的重量$w[0]=18$，是否装入轮船的两种状态。

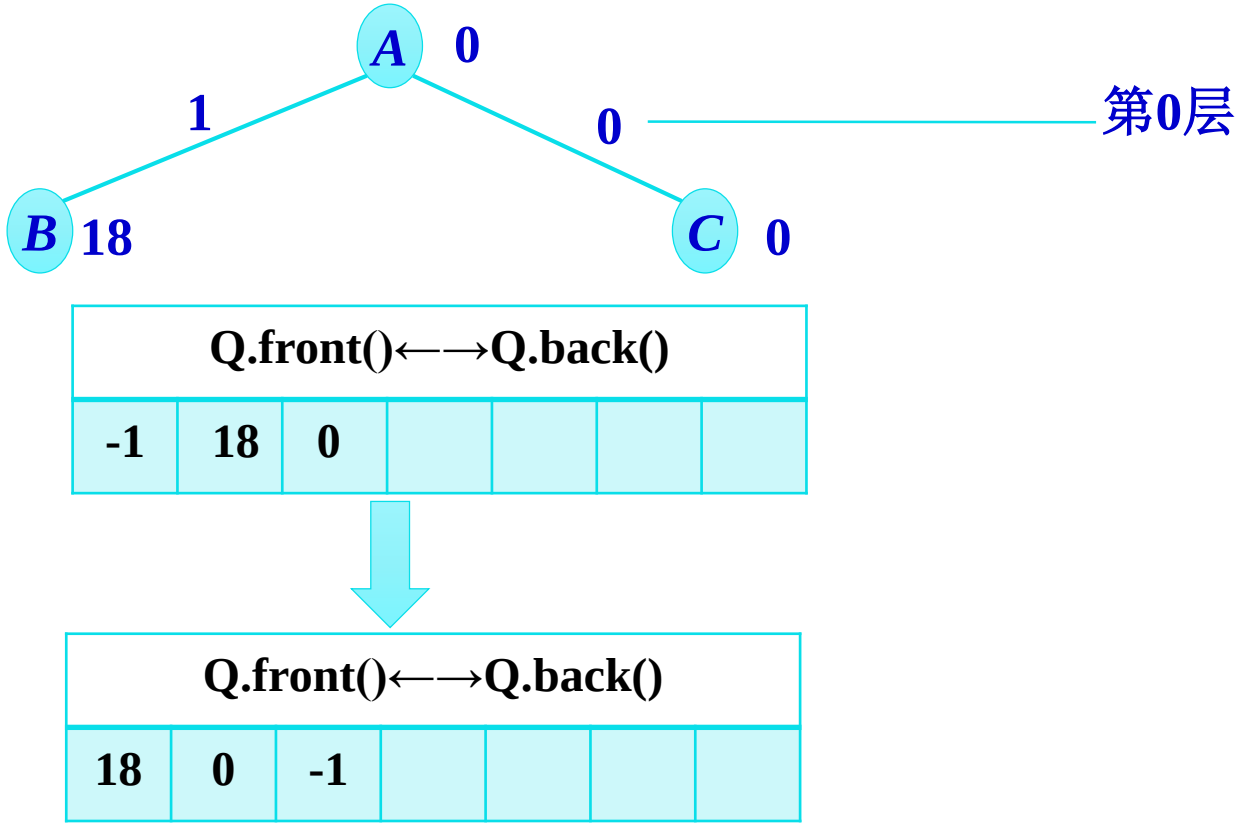
◆在第0层，$cw=0$，$bestw=0$，$r=w[1]+w[2]+w[3]=68$，$cw+w[0]<c$，$cw+r>bestw$，结点B和C依次进入队列。



Q.front()←→Q.back()						
-1	18	0				

2.2 装载问题

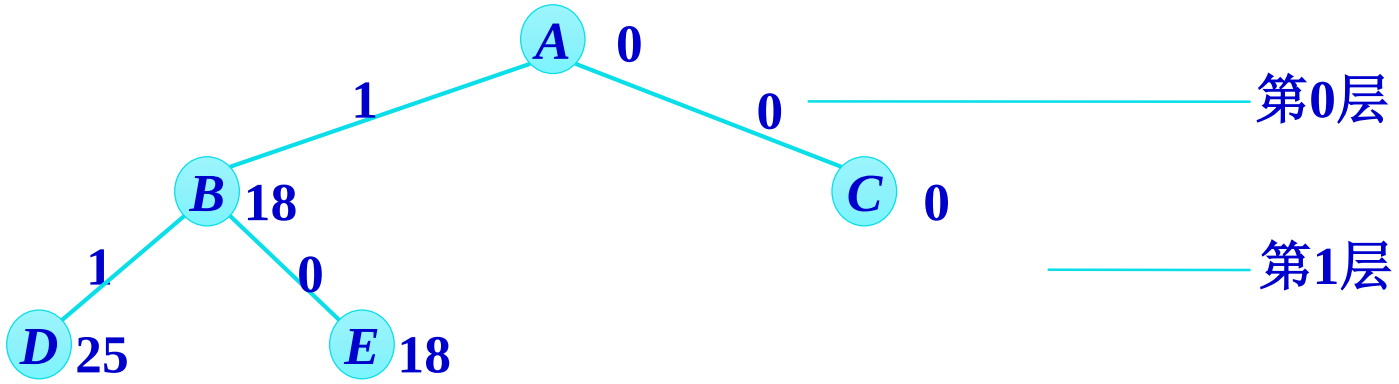
■从队列中取出活结点-1，由于队列不为空，表示当前层结束，新的一层开始，在队列尾部增加一个-1标记。



2.2 装载问题

■从队列中取出活结点$cw=18$，即结点B。

◆第1层即集装箱1的重量$w[1]=7$，$bestw=18$，$r=w[2]+w[3]=61$，$cw+w[1]=25<c$，$cw+r=79>bestw$，结点D和E依次进入队列。



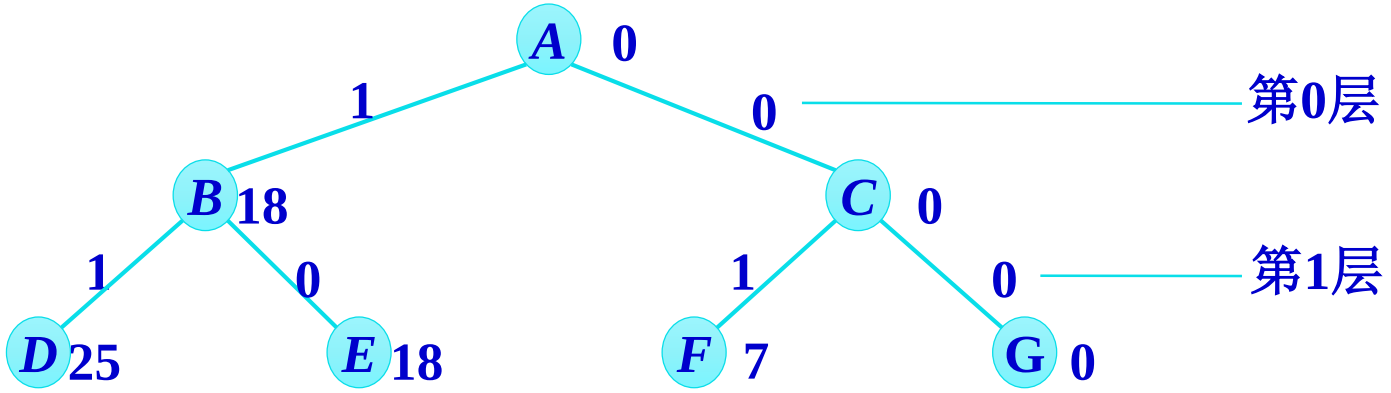
Q.front()←→Q.back()						
18	0	-1				

Q.front()←→Q.back()						
0	-1	25	18			

2.2 装载问题

■从队列中取出活结点$cw=0$，即结点C。

◆$bestw=25$，$r=w[2]+w[3]=61$，由于$cw+w[1]=7<c$，$cw+r=61>bestw$，结点F和G依次进入队列。

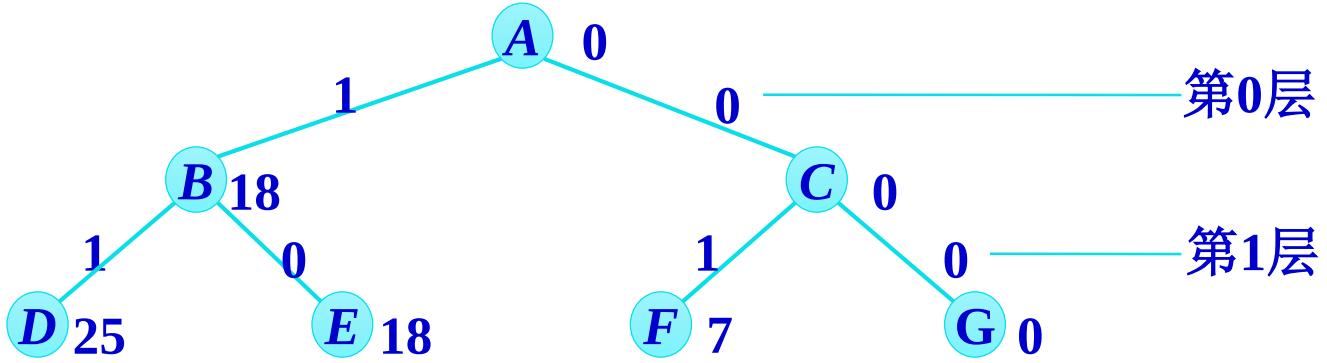


Q.front()←→Q.back()						
0	-1	25	18			

Q.front()←→Q.back()						
-1	25	18	7	0		

2.2 装载问题

■从队列中取出活结点-1，由于队列不为空，表示当前层结束，新的一层开始，在队列尾部增加一个-1标记。

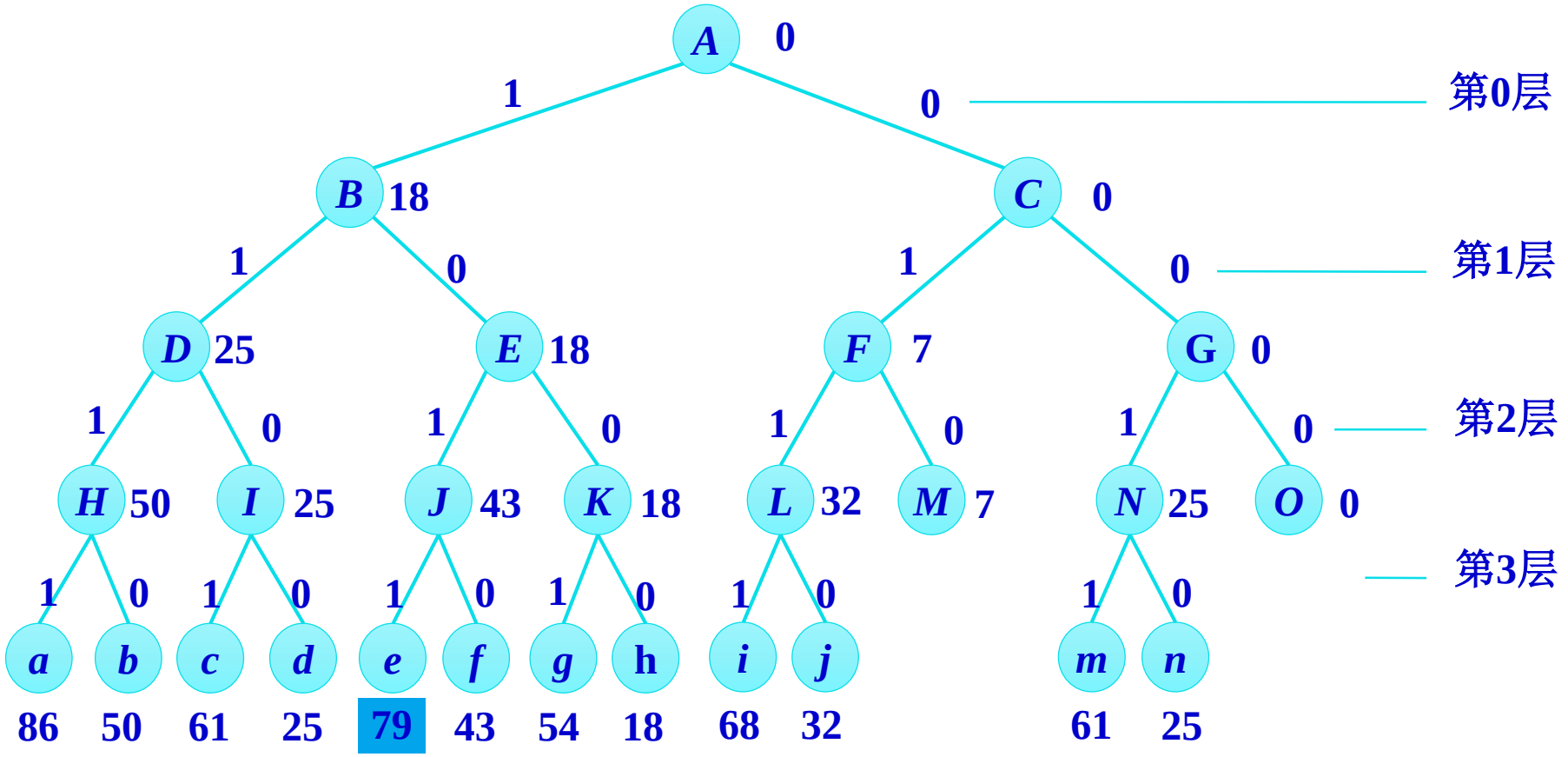


Q.front()←→Q.back()						
-1	25	18	7	0		

Q.front()←→Q.back()						
25	18	7	0	-1		

2.2 装载问题

■基于先进先出队列求出的最终解为：



2.2 装载问题

算法描述

■装载问题分支限界算法的数据结构

```
#define NUM 100
int n;                    //集装箱的数量
int c;                    //轮船的载重量
int w[NUM];               //集装箱的重量数组
```

输入样例	输出样例
80 4 18 7 25 36	79

2.2 装载问题

算法描述

```
int MaxLoading(){
    queue<int> Q;
    Q.push(-1);
    int i = 0;
    int Ew = 0;
    int bestw = 0;
    int r = 0;
    for(int j=1; j<n; j++)
        r += w[j];
    //搜索子空间树
    while (true) {
        //检查左子树
        int wt = Ew+w[i];
        if (wt<=c)     //检查约束条件
        {
            if (wt>bestw) bestw = wt;
            //加入活结点队列
            if (i<n-1) Q.push(wt);
        }
        //检查右子树
        //检查上界条件
        if (Ew+r>bestw && i<n-1)
            Q.push(Ew);
        //从队列中取出活结点
        Ew = Q.front();
        Q.pop();
        if (Ew==-1) {    //判断同层的尾部
            if (Q.empty()) return bestw;
            //同层结点尾部标志
            Q.push(-1);
            //从队列中取出活结点
            Ew = Q.front();
            Q.pop();
            i++;
            r -= w[i];
        }
    }
    return bestw;
}
```

2.2 装载问题

采用优先队列的装载问题的算法优化

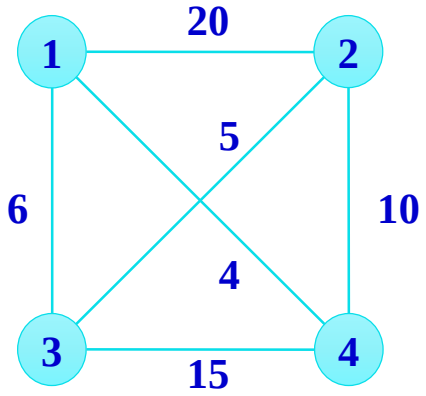
■采用优先队列的装载问题求解：

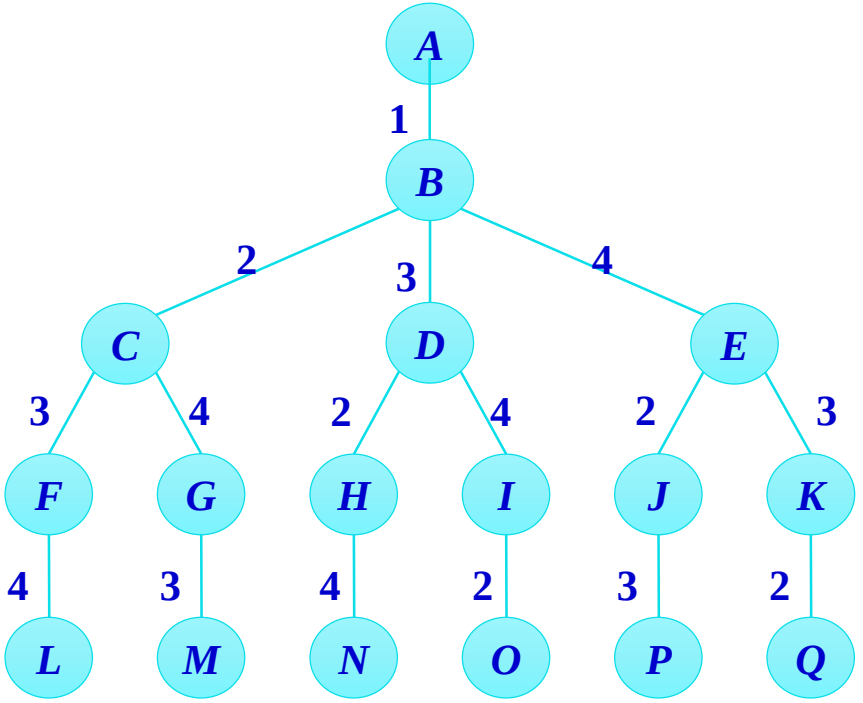
◆将求得的上界函数值作为活跃结点的优先级，采用优先队列存储各活跃结点；

◆每次选取当前优先级最高的结点作为扩展结点，直到队列为空。

2.3 旅行商问题

- **问题描述：**某售货员要到若干城市去推销商品，已知各城市之间的路程(或旅费)。他要选定一条从驻地出发，经过每个城市一次，最后回到驻地的路线，使总的路程(或总旅费)最小。
- 旅行售货员问题的解空间可以组织成一棵排列树，从树的根结点到任一叶结点的路径定义了图的一条周游路线。旅行售货员问题要在图G中找出费用最小的周游路线。





2.3 旅行商问题

无向图旅行商问题求解思路之一

■限界函数的设定方法：

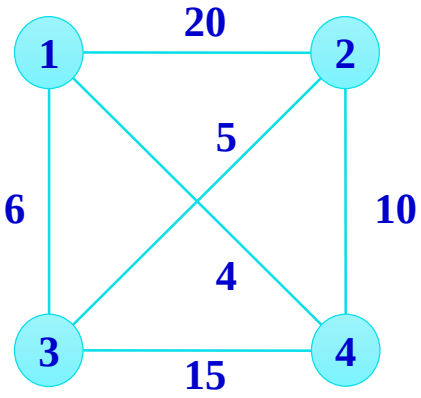
设顶点 c_i 出发的最短边长度为 l_i，d_j 为选定巡回路线中第 j 段的长度

$$L=\sum_{j=1}^{k} d_j + l_{i_k} + \sum_{i_j \notin B} l_{i_j}$$

已走过路径长

剩余长度下界

■将限界函数值作为优先队列中结点的优先级，按此顺序依次扩展各结点，直到队列为空。



（1，3，2，4，1）或（1，4，2，3，1）的总权值为25，为最优回路。

2.3 旅行商问题

普通有向图旅行商问题求解思路之一

■在计算限界函数时，令扩展结点*i*的当前费用为：

$$cc(i)=\sum_{j=2}^{i}a[x[j-1],x[j]]$$

■从每个剩余结点出发的最小出边费用的总和为：

$$rcost(i)=\sum_{j=i+1}^{n}\min_{i<k<n,\,k\neq j}\{a[x[j],x[k]]\}$$

■则扩展结点*i*的限界函数为：

$$B(i)=cc(i)+rcost(i)$$

■令*bestc*表示目前为止找到的最佳回路费用和，如果 $B(i)\geq bestc$，则不用把x[*i*]放入活结点表中，否则放入。具有最小费用*B*(*i*)的活结点优先扩展。

2.3 旅行商问题

算法描述（数据结构的定义）

- 旅行商问题优先队列式分支限界算法的数据结构

```
#define inf 1000000    //∞
#define NUM 100
int n;          //图G的顶点数
int a[NUM][NUM];//图G的邻接矩阵
int NoEdge = inf;//图G的无边标志
int cc;         //当前费用
int bestc;      //当前的最小费用
```

```
struct node
{
    //优先队列以lcost为优先级参数
    friend bool operator < (const node& a,const node& b)
    {
        if(a.lcost > b.lcost)  return true;
        else return false;
    }
    int lcost;      //子树费用的下界
    int rcost;      //从x[s]～x[n-1]顶点的最小出边和
    int cc;         //当前费用
    int s;          //当前结点的编号
    int x[NUM];     //搜索的路径
};
```

2.3 旅行商问题

算法描述（堆的建立与初始化）

■ 定义优先队列

```
priority_queue <node> H;
int minOut[NUM]; //各个顶点的最小出边费用
int minSum = 0;    //最小出边费用之和
```

■ 计算各个顶点的最小出边费用

```
int i, j;
for(i=1; i<=n; i++){
        int Min = NoEdge;
        for(j=1; j<=n; j++)
                if( a[i][j]!=NoEdge && (a[i][j]<Min ||
Min==NoEdge))
                        Min = a[i][j];
        if (Min==NoEdge)  return NoEdge; //无回路
        minOut[i] = Min;
        minSum += Min;
}
```

■ 初始化

```
node E;
for(i=1; i<=n; i++)
        E.x[i] = i;
E.s = 1;
E.cc = 0;
E.rcost = minSum;
int bestc = NoEdge;
```

■ 算法开始时创建一个最小堆，用于表示活结点优先队列。堆中每个结点的子树费用的下界lcost值是优先队列的优先级。接着算法计算出图中每个顶点的最小费用出边并用minout记录。如果所给的有向图中某个顶点没有出边，则该图不可能有回路，算法即告结束。如果每个顶点都有出边，则根据计算出的minout作算法初始化。

2.3 旅行商问题

算法描述（搜索排列树）

■ 搜索排列树

```
while (E.s<n){    //非叶结点
   //当前扩展结点是叶结点的父结点
   if (E.s==n-1){
      //再加2条边构成回路
      //所构成的回路是否优于当前最优解
      if (a[E.x[n-1]][E.x[n]]!=NoEdge && a[E.x[n]][1]!=NoEdge
          && (E.cc+a[E.x[n-1]][E.x[n]]+a[E.x[n]][1]<bestc
          || bestc==NoEdge))    {
          //费用更小的回路
          bestc = E.cc+a[E.x[n-1]][E.x[n]]+a[E.x[n]][1];
          E.cc = bestc;
          E.lcost = bestc;
          E.s++;
          H.push(E);
      }
      else delete[ ] E.x;      //舍弃扩展结点
   }
```

■ 算法的while循环体完成对排列树内部结点的扩展。对于当前扩展结点，算法分2种情况进行处理：

(1) 首先考虑s=n-2的情形，此时当前扩展结点是排列树中某个叶结点的父结点。如果该叶结点相应一条可行回路且费用小于当前最小费用，则将该叶结点插入到优先队列中，否则舍去该叶结点。

2.3 旅行商问题

算法描述（搜索排列树）

```
else{   //搜索树的内部结点
  //产生当前扩展结点的儿子结点
  for (i=E.s+1; i<=n; i++)
    if (a[E.x[E.s]][E.x[i]]!=NoEdge)   {
      //可行儿子结点
      int cc = E.cc+a[E.x[E.s]][E.x[i]];
      int rcost = E.rcost-minOut[E.x[E.s]];
      //限界函数B(i)
      int B = cc+rcost;
      //子树可能包含最优解
      if(B<bestc || bestc==NoEdge){
        //结点E插入优先队列
        ……
      }
    }
    //完成结点扩展
    delete [] E.x;
}
```

```
node N;
for(j=1; j<=n; j++)
   N.x[j] = E.x[j];
N.x[E.s+1] = E.x[i];
N.x[i] = E.x[E.s+1];
N.cc = cc;
N.s = E.s+1;
N.lcost = B;
N.rcost = rcost;
H.push(N);
```

(2) 当s<n-2时，算法依次产生当前扩展结点的所有儿子结点。由于当前扩展结点所相应的路径是x[0:s]，其可行儿子结点是从剩余顶点x[s+1:n-1]中选取的顶点x[i]，且(x[s]，x[i])是所给有向图G中的一条边。对于当前扩展结点的每一个可行儿子结点，计算出其前缀(x[0:s]，x[i])的费用cc和相应的下界lcost。当lcost<bestc时，将这个可行儿子结点插入到活结点优先队列中。

2.3 旅行商问题

算法描述（搜索排列树）

```
    //队列为空时，搜索结束
    if(H.empty()) break;
    else
    {
        E=H.top();  //取下一个扩展结点
        H.pop();
    }
}

if (bestc==NoEdge) return NoEdge; //表示无回路

//输出最优解
for(i=1; i<=n; i++)
    printf("%d ", E.x[i]);
printf("\n");

//清空剩余的队列元素
while (!H.empty()) H.pop();
return bestc;       //返回最优解
```

- 算法中while循环的终止条件是排列树的一个叶结点成为当前扩展结点。当s=n-1时，已找到的回路前缀是x[0:n-1]，它已包含图G的所有n个顶点。因此，当s=n-1时，相应的扩展结点表示一个叶结点。此时该叶结点所相应的回路的费用等于cc和lcost的值。剩余的活结点的lcost值不小于已找到的回路的费用。它们都不可能导致费用更小的回路。因此已找到的叶结点所相应的回路是一个最小费用旅行售货员回路，算法可以结束。
- 算法结束时返回找到的最小费用，相应的最优解由数组x给出。

2.3 旅行商问题

■ 分支限界的旅行商问题求解优化：

◆先使用贪心算法求解一个旅行商回路的近似解作为bestc，以避免开始时扩展不必要的结点；

◆限界函数的优化：比如旅行商回路长度小于每个结点关联的两条最短的边之和的二分之一，使用各结点关联的两条最短的边形成更精准的下界函数。

分支限界法小结

分支界限法的特点

- 分支限界法是一个用途十分广泛的算法，适用于组合优化中求最优解的问题；运用这种算法的技巧性很强，不同类型的问题解法也各不相同。
- 分支限界法的基本思想是对有约束条件的最优化问题的所有可行解（数目有限）空间进行搜索。
  - 该算法在具体执行时，把全部可行的解空间不断分割为越来越小的子集（称为分支），并为每个子集内的解的值计算一个下界或上界（称为限界）。
  - 在每次分支后，对凡是界限超出（劣于）已知可行解值的那些子集不再做进一步分支。这样，解的许多子集（即搜索树上的许多结点）就可以不予考虑，从而缩小了搜索范围。
  - 这一过程一直进行到找出可行解为止，该可行解的值优于任何子集的界限。
- 时间性能：最坏的情况要搜索整个解空间，复杂度是指数型。但如果启发式信息强且剪枝处理得当，平均性能往往很好。
- 空间性能：优先队列往往需要较大的空间开销。

分支限界法小结

- 分支限界法类似于回溯法，是一种在问题的解空间树T上搜索问题解的算法。
- 一般情况下，分支限界法与回溯法的求解目标不同。回溯法的求解目标是找出T中满足约束条件的所有解，而分支限界法的求解目标则是找出满足约束条件的一个解，或是在满足约束条件的解中找出使某一目标函数值达到极大或极小的解，即在某种意义下的最优解。

方法	回溯法	分支限界法
解空间树的搜索方式	深度优先搜索	广度优先或最小耗费优先搜索
存储结点的数据结构	搜索过程中动态产生问题的解空间	队列、优先队列
结点存储特性	只保存从根结点到当前扩展结点的路径	每个结点只有一次成为活结点的机会
应用范围	能够找出满足约束条件的所有解	找出满足约束条件的一个解或特定意义下的最优解

PART 03 启发式搜索法A*

3.1启发式搜索A*的思想

启发式搜索

- 启发式搜索(Heuristic Search)又称为有信息搜索(Informed Search)，它是利用问题拥有的启发信息来引导搜索向最有希望的方向前进，降低了复杂性，达到减少搜索范围、降低问题复杂度的目的，这种利用启发信息的搜索过程称为启发式搜索。
- 在解决问题的过程中，启发式策略仅仅是下一步将要采取措施的一个猜想，常常根据经验和直觉来判断。由于启发式搜索只有有限的信息(比如当前状态的描述)，要想预测进一步搜索过程中状态空间的具体行为则很难。因而，一个启发式搜索可能得到一个次最佳解，这是启发式搜索固有的局限性。
- 一般说来，启发信息越强，扩展的无用节点就越少。引入强的启发信息，有可能大大降低搜索工作量，但不能保证找到最小耗散值的解路径(最佳路径)。因此，在实际应用中，最好能引入降低搜索工作量的启发信息而不牺牲找到最佳路径的保证。

3.1启发式搜索A*的思想

估价函数

■用于评价结点重要性的函数称为估价函数，其一般形式为：

$$f(x)=g(x)+h(x)$$

◆g(x)为从初始结点到结点x付出的实际代价；

◆h(x)为从结点x到目标结点的最优路径的估计代价。启发性信息主要体现在h(x)中，其形式要根据问题的特性来确定。

■虽然启发式搜索有望能够很快到达目标结点，但需要花费一些时间来对新生结点进行评价。因此，在启发式搜索中，估计函数的定义是十分重要的。如定义不当，虽多花费时间进行结点评价，也不保证能找到问题的最优解、甚至是解。

3.1启发式搜索A*的思想

有序搜索算法

- 有序搜索（Ordered Search）又称之为最佳优先搜索（Best First Search），是一种启发式搜索算法，我们也可以将它看作广度优先搜索算法的一种改进；最佳优先搜索算法在广度优先搜索的基础上，用启发估价函数对将要被遍历到的点进行估价，然后选择代价小的进行遍历，直到找到目标节点或者遍历完所有的点，算法结束。
- 如果取估价函数等于节点深度，则它将退化为广度优先搜索。

有序搜索算法步骤如下：

①将初始节点S_0放入Open表中；

②如Open表为空，则搜索失败，退出；

③把Open表的第一个节点取出，放入到Closed表中，并把该节点记为节点n；

④如果节点n是目标节点，则搜索成功，求得一个解，退出；

⑤扩展节点n，生成一组子节点，对既不在Open表中也不在Closed表中的子节点，计算出相应的估价函数值；

⑥把节点n的子节点放到Open表中；

⑦对Open表中的各节点按估价函数值从小到大排列；

⑧转到②。

3.1启发式搜索A*的思想

A*算法

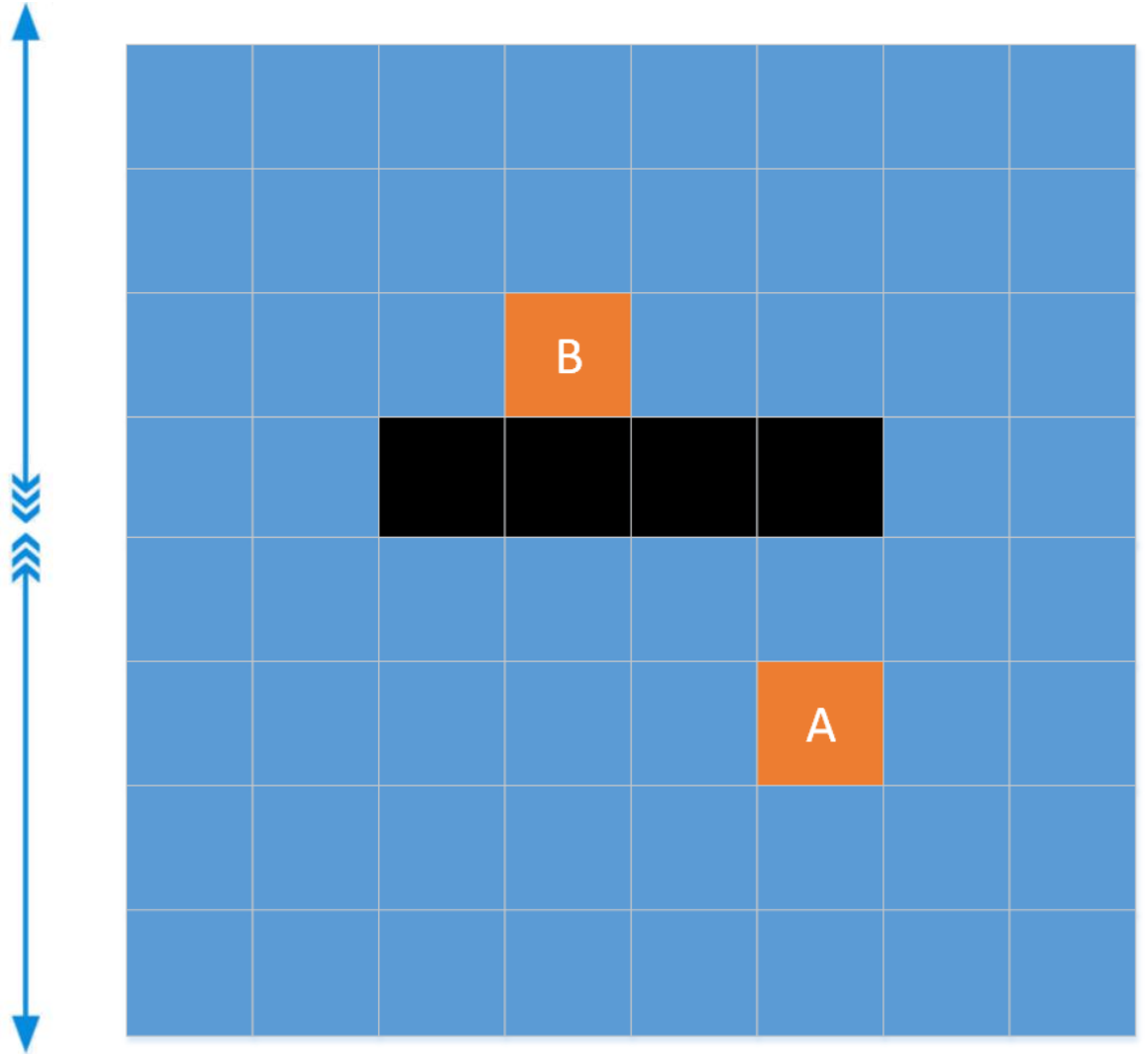
- 启发性信息用一个特别的估价函数f*来表示：f*(x)=g*(x)+h*(x)
  - g*(x)为从初始结点到结点x的最佳路径所付出的代价；
  - h*(x)是从x到目标结点的最佳路径所付出的代价；
  - f*(x)是从初始结点出发通过结点x到达目标结点的最佳路径的总代价。
- 基于上述g*(x)和h*(x)的定义，对启发式搜索算法中的g(x)和h(x)做如下限制：
  - g(x)是对g*(x)的估计，且g(x)>0；
  - h(x)是h*(x)的下界，即对任意节点x均有h(x)≤h*(x)。
  - 在满足上述条件情况下的有序搜索算法称为A*算法。
- 对于某一搜索算法，当最佳路径存在时，就一定能找到它，则称此算法是可纳的。可以证明，A*算法是可纳算法。也就是说，对于有序搜索算法，当满足h(x)≤h*(x)条件时，只要最佳路径存在，就一定能找出这条路径。

3.2 路径规划问题

- 路径规划是非常常见的一类问题，例如移动机器人从A点移动到B点，游戏中的人物从A点移动到B点，以及自动驾驶中，汽车从A点到B点。
- 这类问题中，都有两个关键问题需要解决：一是找到最短路径；二是避开障碍物。
- 解决这类问题，不得不提的一个经典的算法就是A*算法。 A*算法的提出是想要解决移动机器人路径规划问题，也就是要在地图上找到一条从起点到终点的最短路径。A*算法是一种基于采样搜索的粗略路径规划算法，由stanford研究院的Peter Hart，Nils Nilsson以及Bertram Raphael发表于1968年。

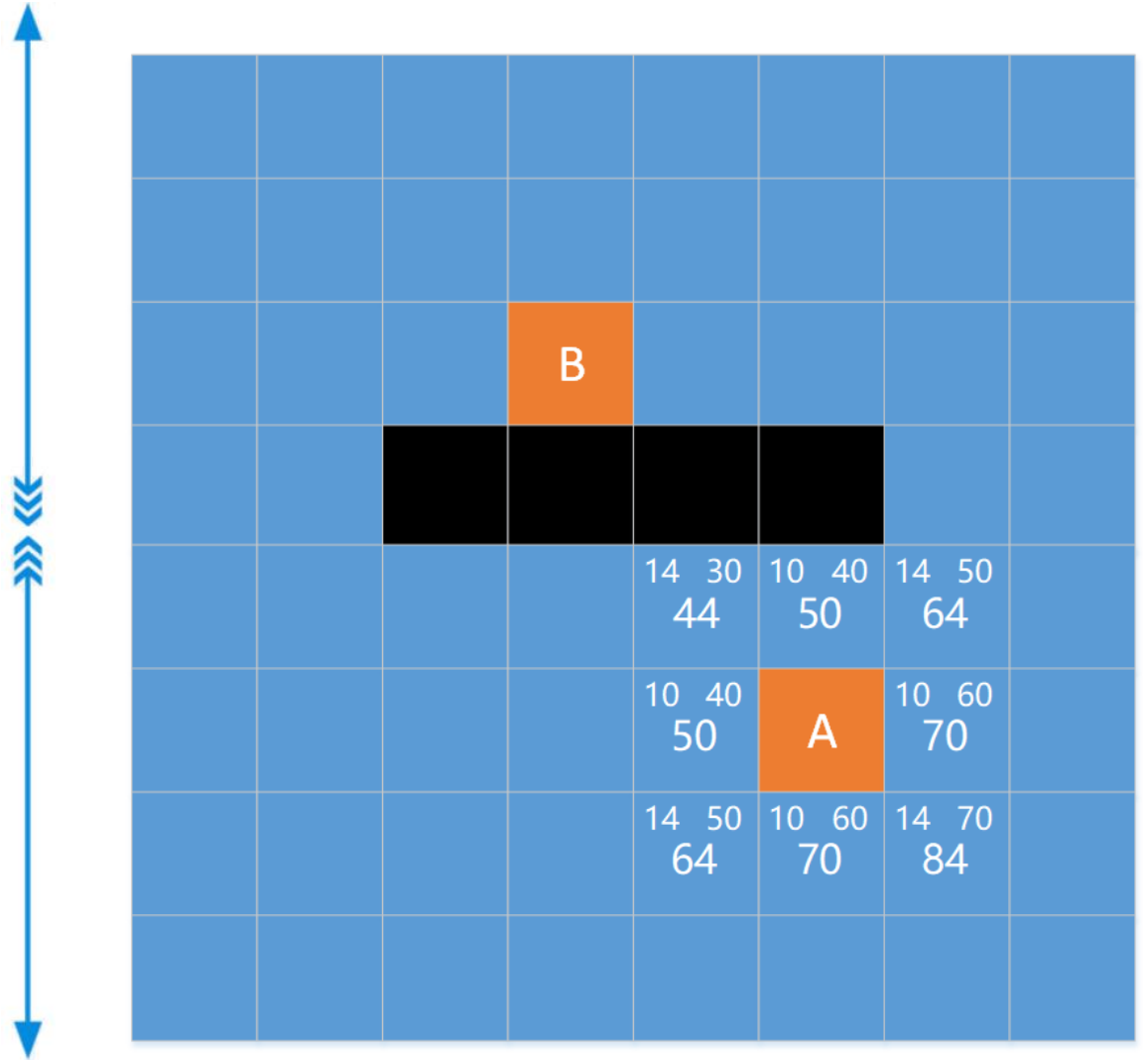
3.2 路径规划问题

- 设有这么一个场景：某人想从图中A点移动到B点，但是这两点之间被一堵墙隔开。图中黑色为墙壁，即不可移动区域，蓝色为可移动区域。
- 问：如何求得从起始点A点开始，到终点B点的最短距离？
- 要注意的几点：
  - 当前的区域已被简化成由一个个小正方形组成的简化区域。类似于一个二维数组，其状态分为可通过和不可通过。
  - 方格之间的距离按照其中心点之间的距离计算得出。
  - 区域的简化，不仅可以被划分为一个个小正方形，还可以依照精细程度，将区域划分为一个个三角形，六边形等。



3.2 路径规划问题

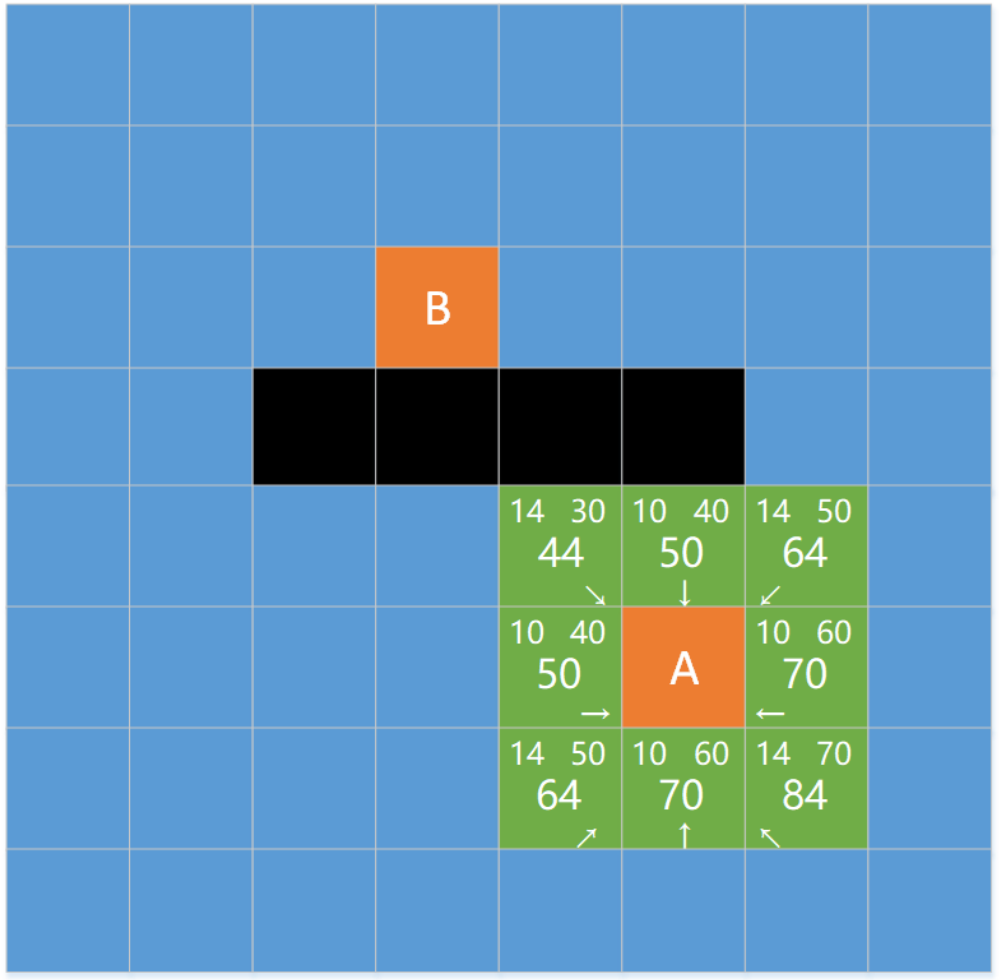
- 问题中A*（A-Star）算法的估价函数f(x)=g(x)+h(x)的设计：
  - g(x)=当前方格x的中心点与起始点A的中心点的距离。该距离需要根据当前已确定的最短路径来计算（注：按照每个小方格边长为10计算，A点的上下左右四个方向的方块G取值都为10；四个斜边方向G取值都为√2*10≈14）。
  - h(x)=当前方格中心点与目标点B中心点的曼哈顿距离，即为当前点与终点的横向和纵向距离之和。
- 现对A点周围各点进行评估：
  - f,g和h的评分被写在每个方格里。如图所示，中间数字为f，g在左上角，h则在右上角。



3.2 路径规划问题

■开始搜索：

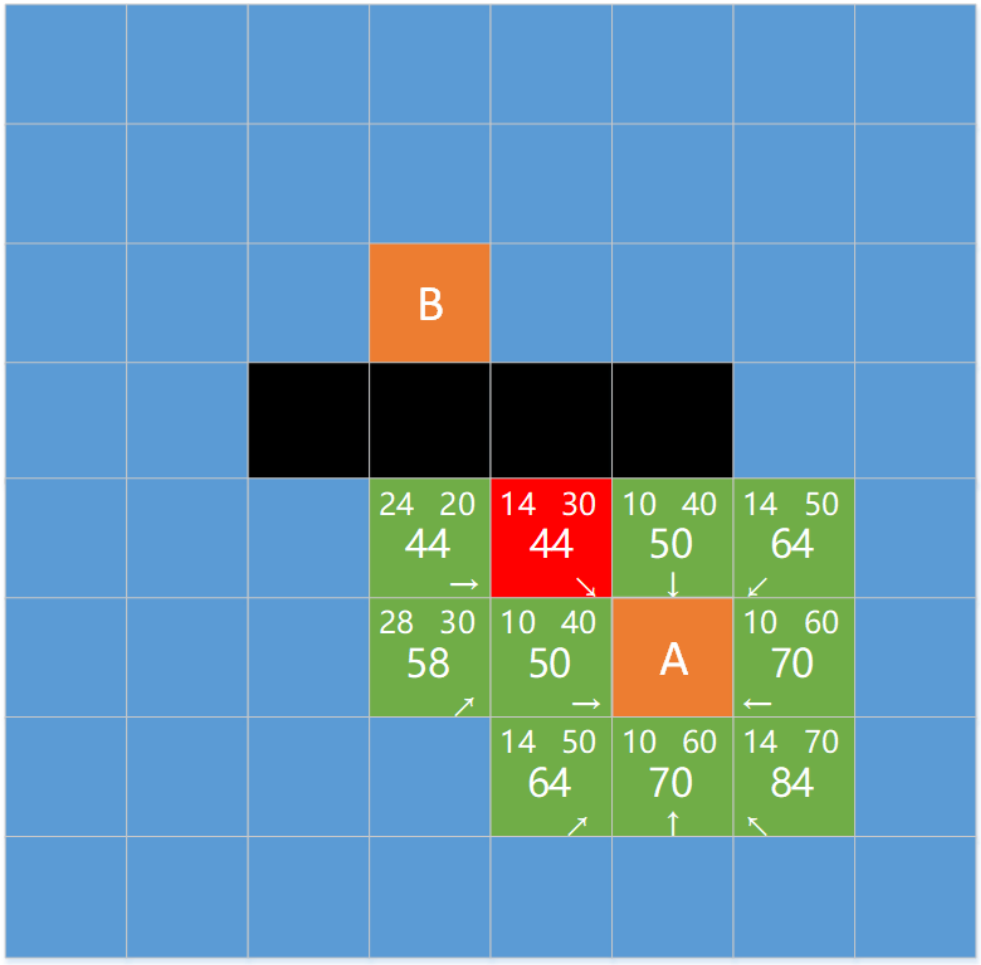
1. 把起始点添加到open list。
2. 查找起始点周围所有可到达或者可通过的方格，把他们加入 open list。将起始点设置为新加入的方格的父节点，用箭头表示如图。
3. 从open list中删除起始点，把它加入到close list，列表中保存所有不需要再次检查的方格。



3.2 路径规划问题

■继续搜索：

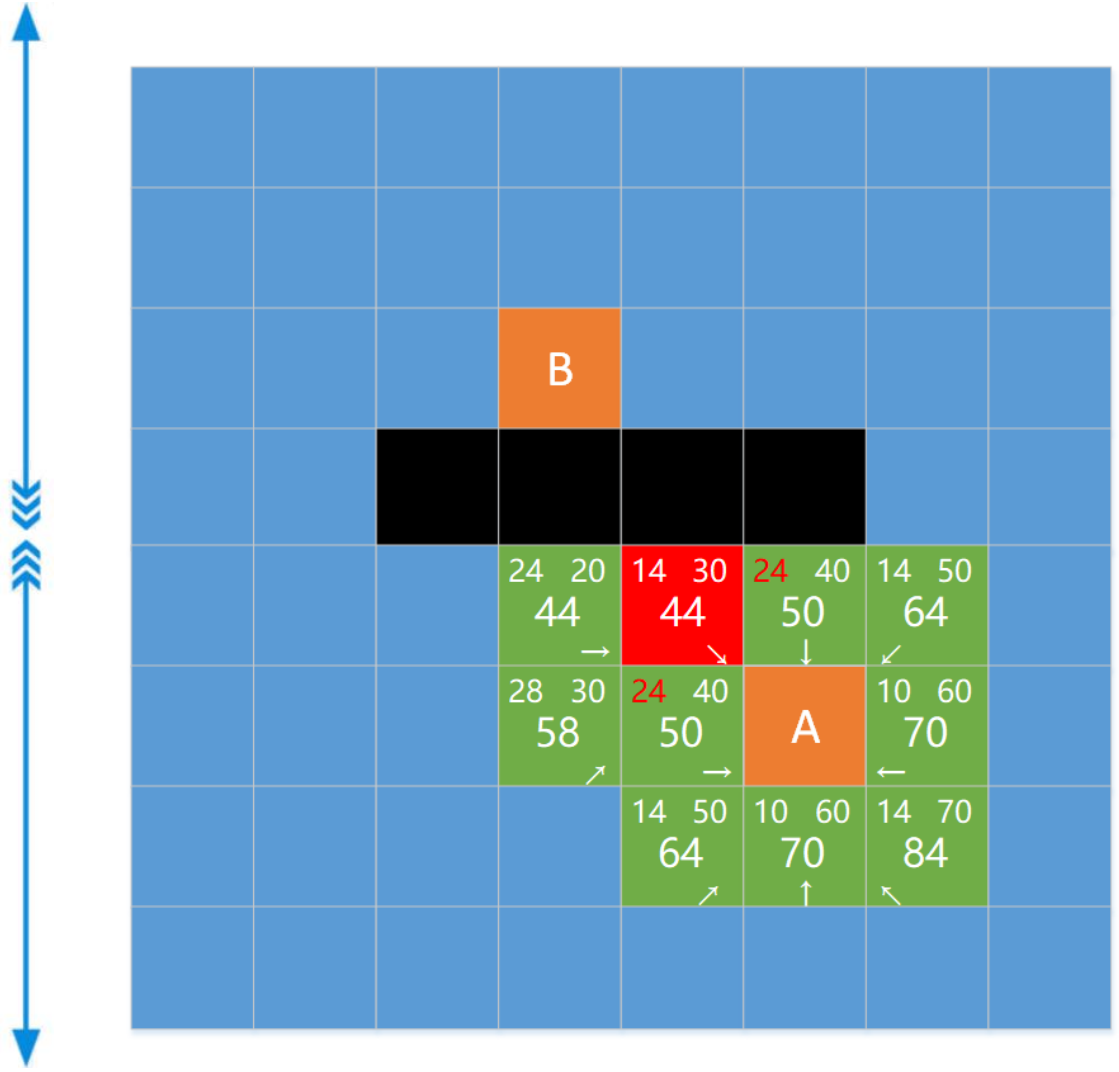
1. 遍历open list，找到其中f值最小的方格。称其为选定方格。
2. 对该方格做如下操作：
   - 从open list中取出放到close list中；
   - 检查与该方格相邻的每个格子，忽略其中已在close list或者不可通过的格子；
   - 如果检查到的格子不在open list中，则将其加入到open list。并且将选定方格设定为他们的父节点；
   - 新加入的方格如右图所示。



3.2 路径规划问题

■继续搜索：

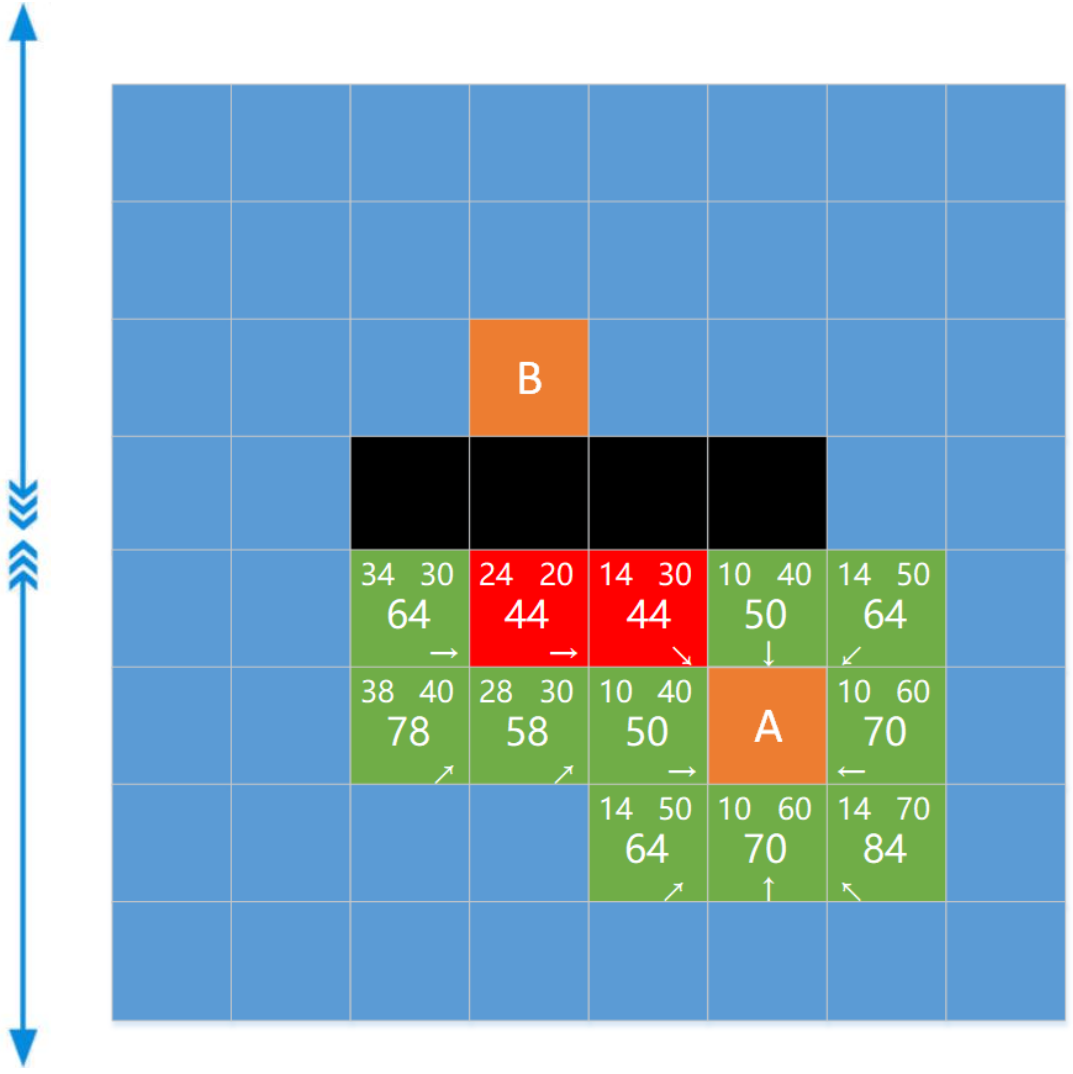
- 如果检查到的格子在open list中，则检查经过选定方格到达该方格，是否有更小的g值。
  - 如果没有，不做操作。
  - 如果有，将该方格的父节点设置为选定方格，并重新计算其f值和g值。
- 如右图所示，检查后的g值为24，比原来10大，所以不做任何操作。



3.2 路径规划问题

■再次搜索：

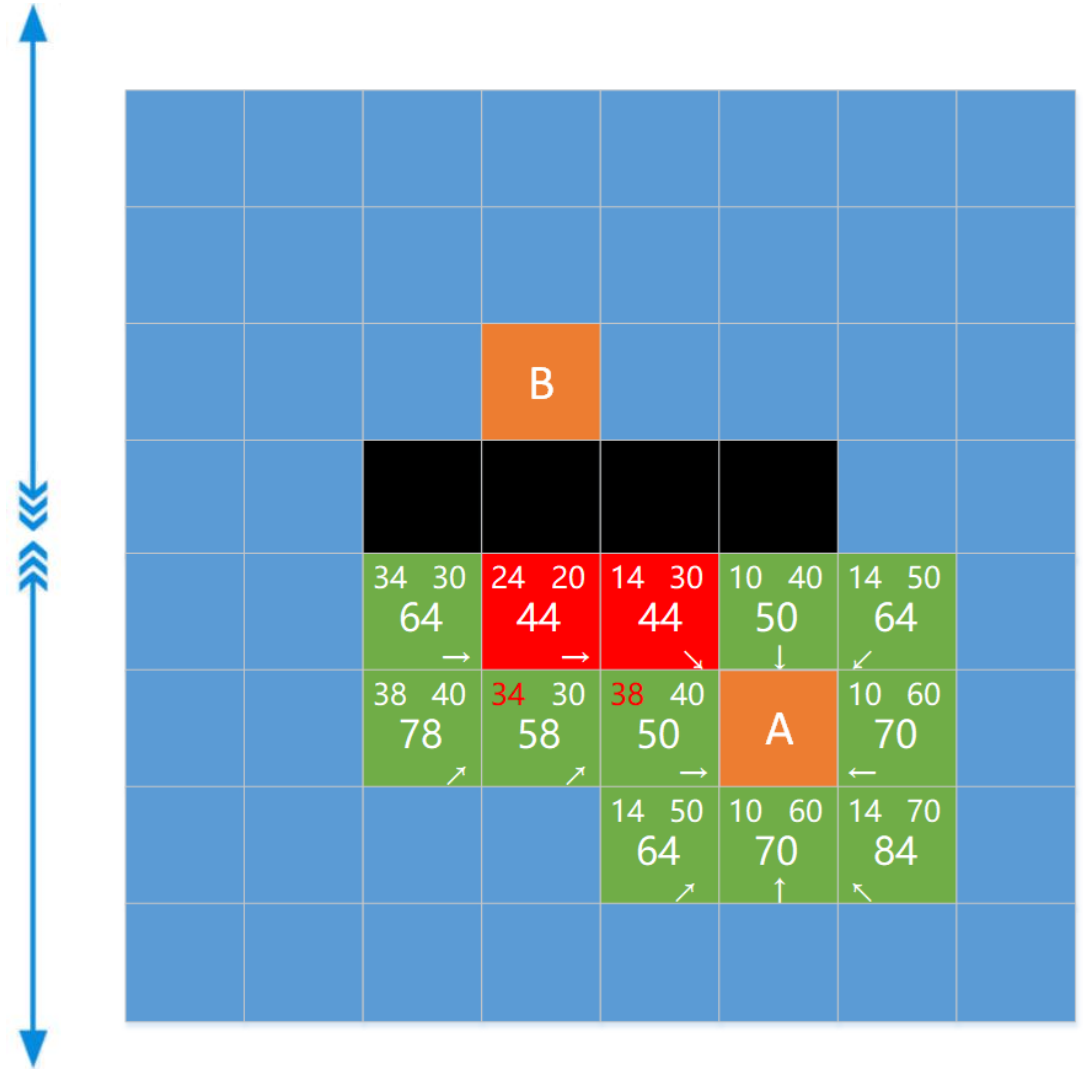
1. 再次遍历open list，找到其中f值最小的方格作为为选定方格。
2. 对该方格做如下操作：
   - 从open list中取出放到close list中；
   - 检查与该方格相邻的每个格子，忽略其中已在close list或者不可通过的格子；
   - 如果检查到的格子不在open list中，则将其加入到open list。并且将选定方格设定为他们的父节点；
   - 新加入的方格如右图所示。



3.2 路径规划问题

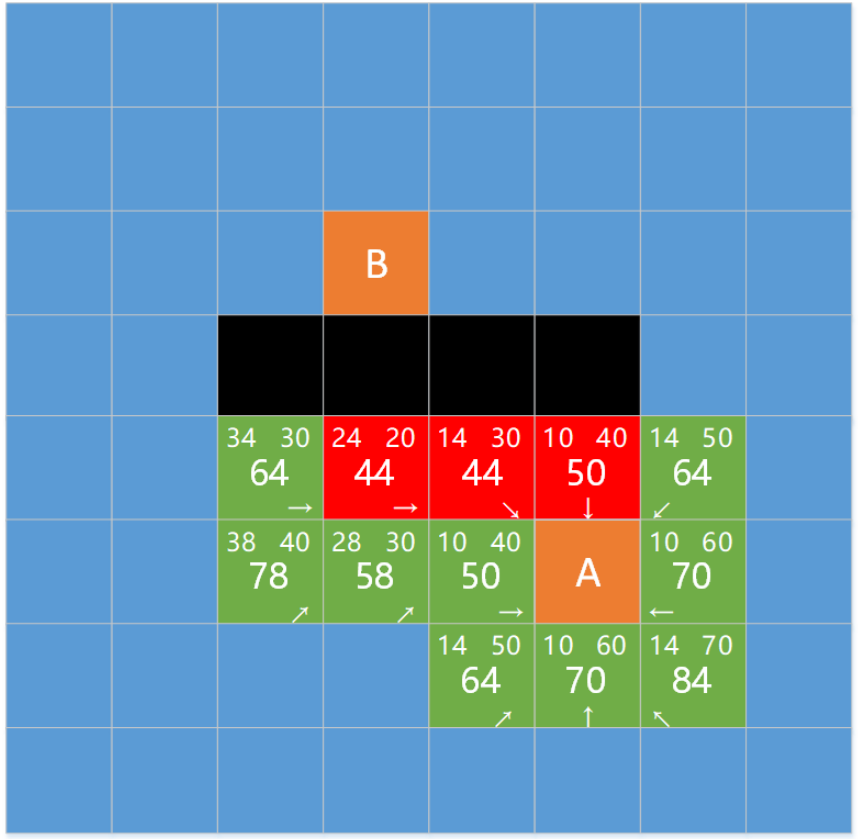
■再次搜索：

- 如果检查到的格子在open list中，则检查经过选定方格到达该方格，是否有更小的g值。
  - 如果没有，不做操作。
  - 如果有，将该方格的父节点设置为选定方格，并重新计算其f值和g值。
- 如图所示，检查后的g值为34和38，比原来28,24大，所以不做任何操作。

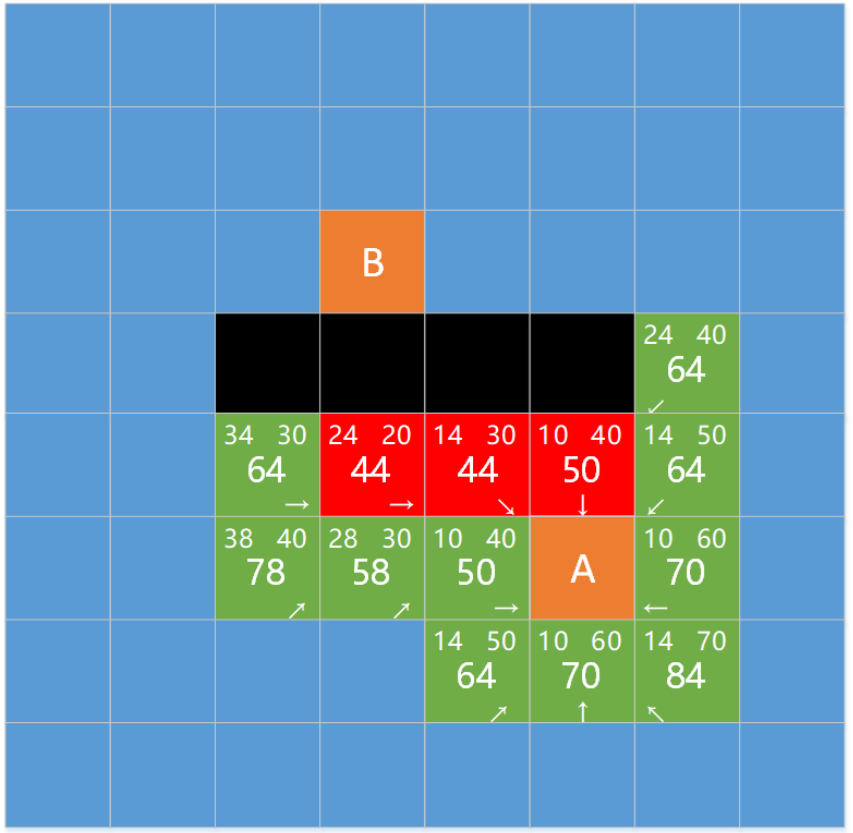


3.2 路径规划问题

■第3轮搜索的选定方格

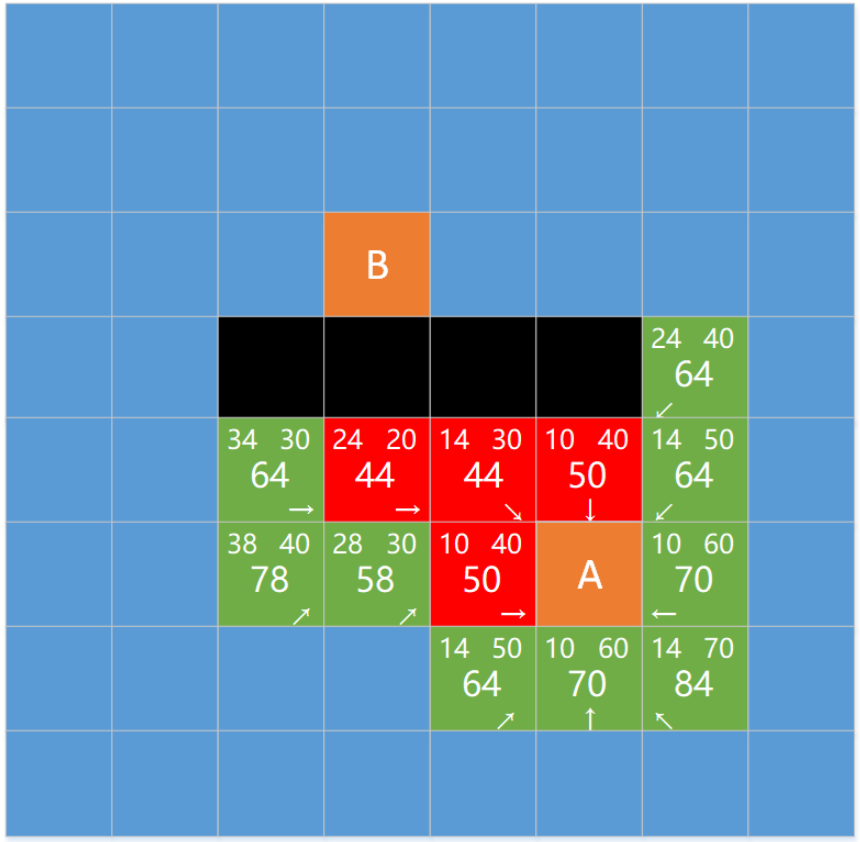


■第3轮搜索的扩展与更新方格

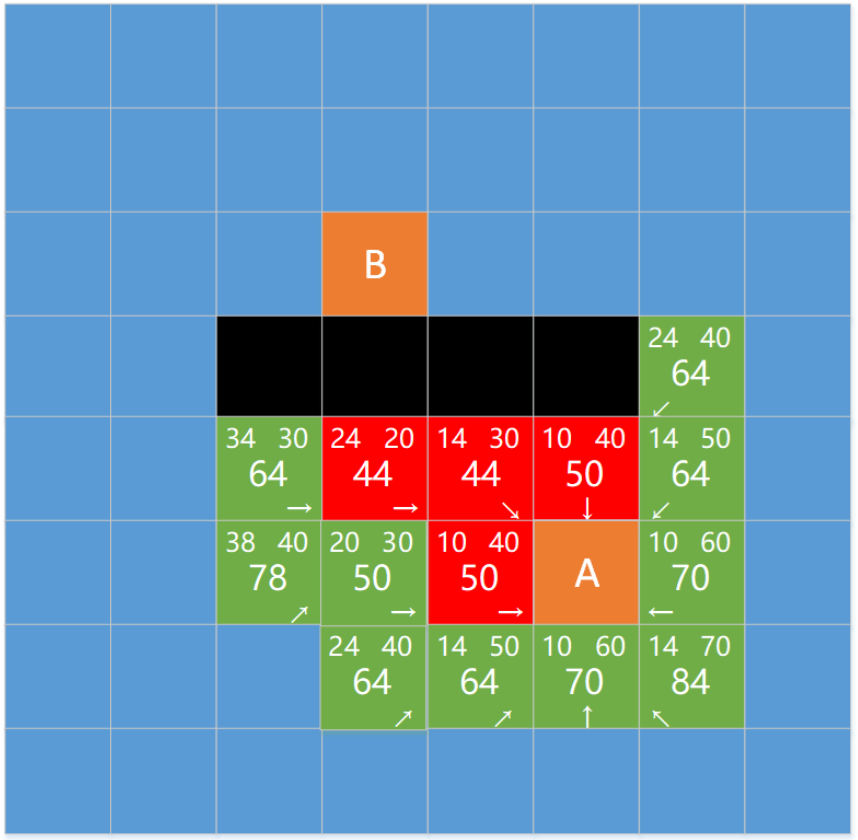


3.2 路径规划问题

■第4轮搜索的选定方格

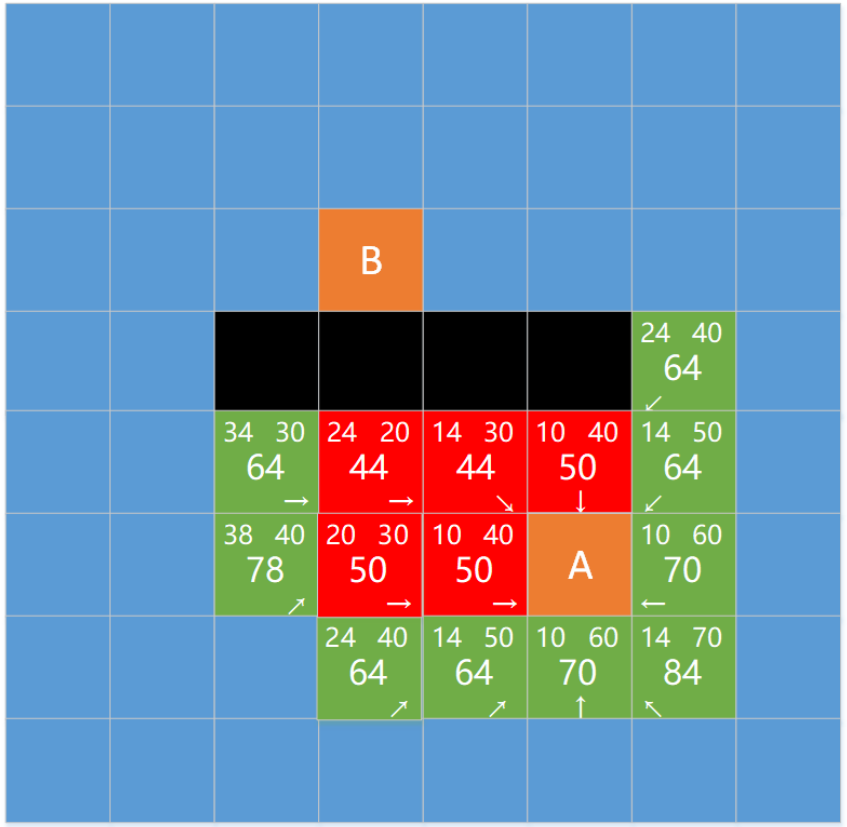


■第4轮搜索的扩展与更新方格

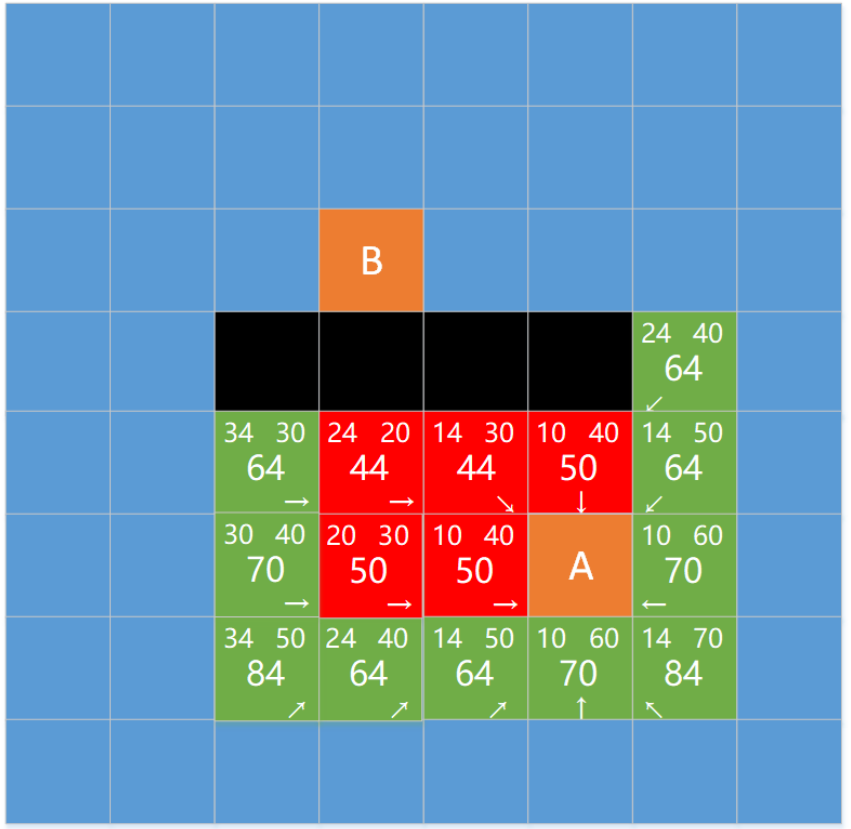


3.2 路径规划问题

■第5轮搜索的选定方格

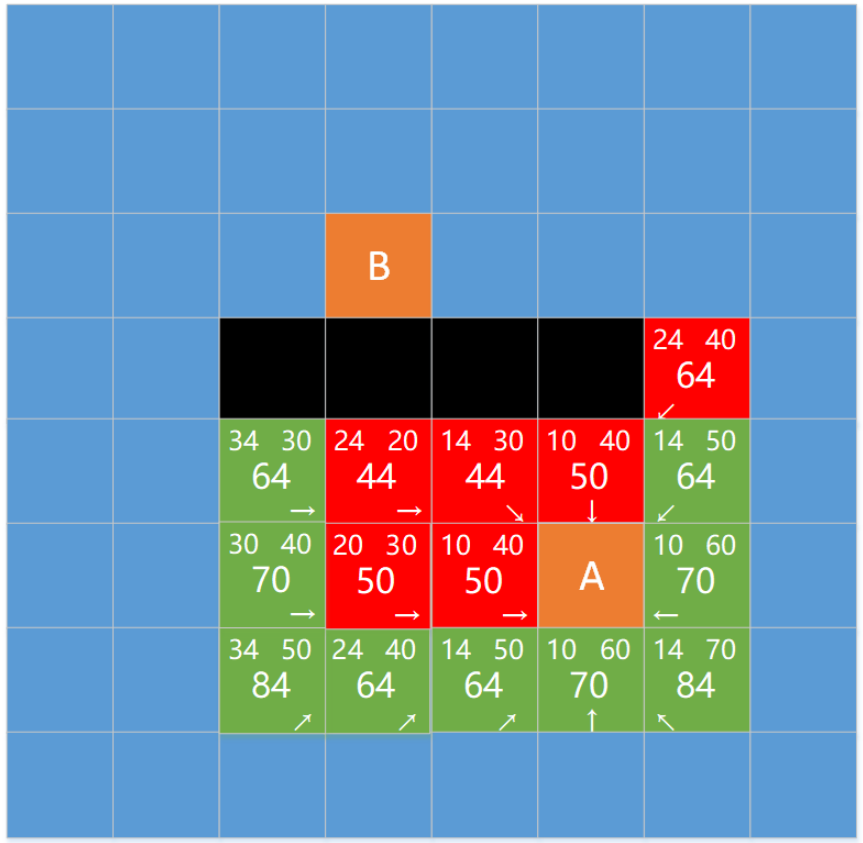


■第5轮搜索的扩展与更新方格

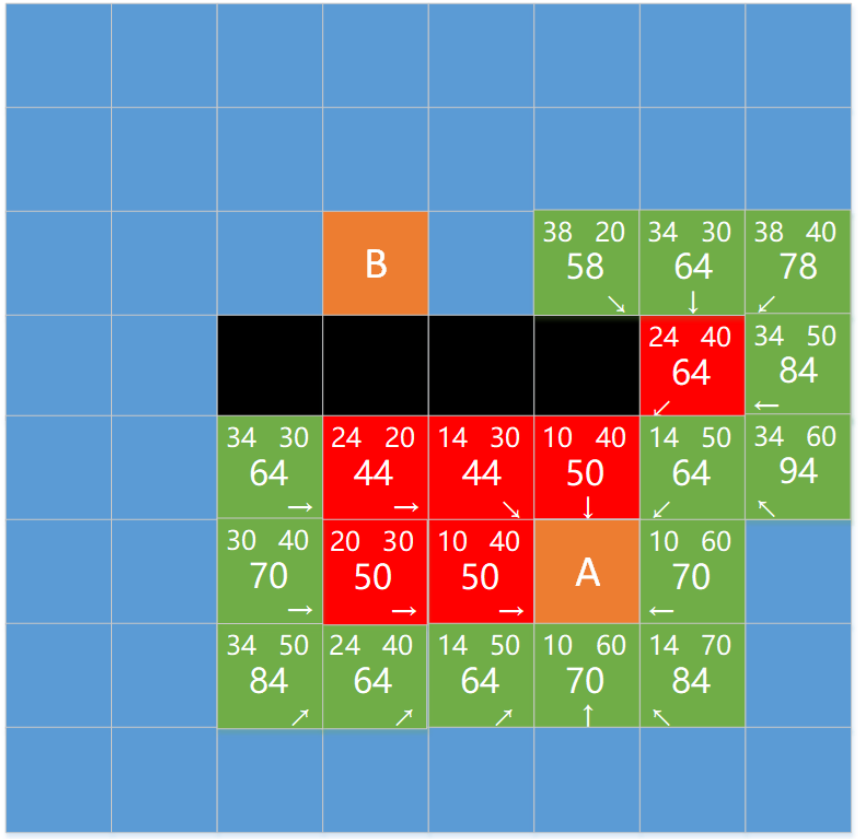


3.2 路径规划问题

■第6轮搜索的选定方格

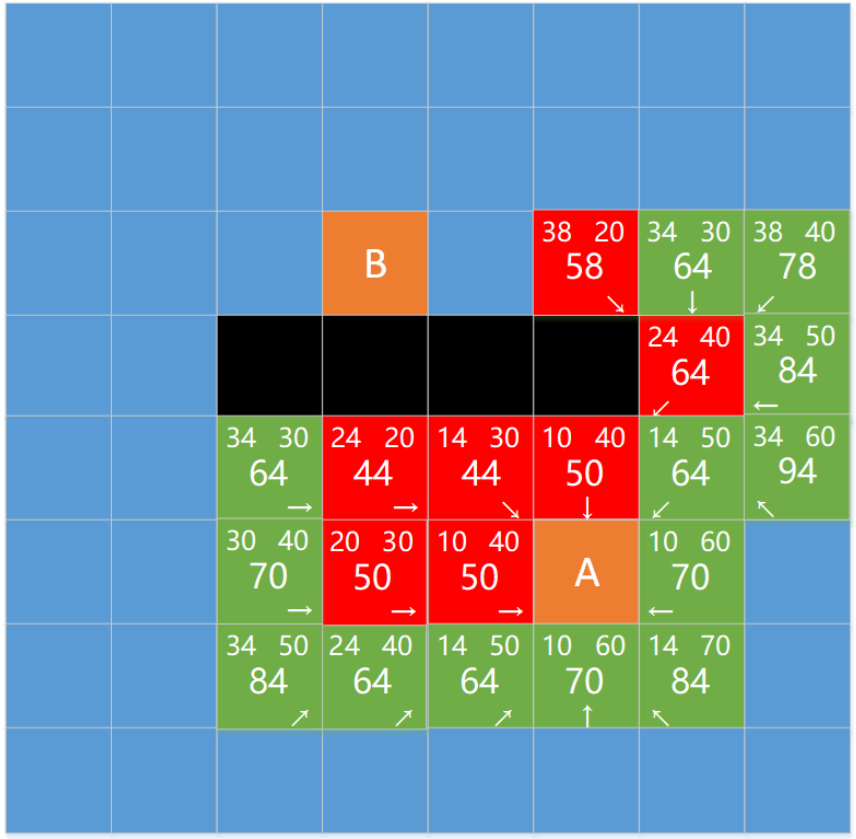


■第6轮搜索的扩展与更新方格

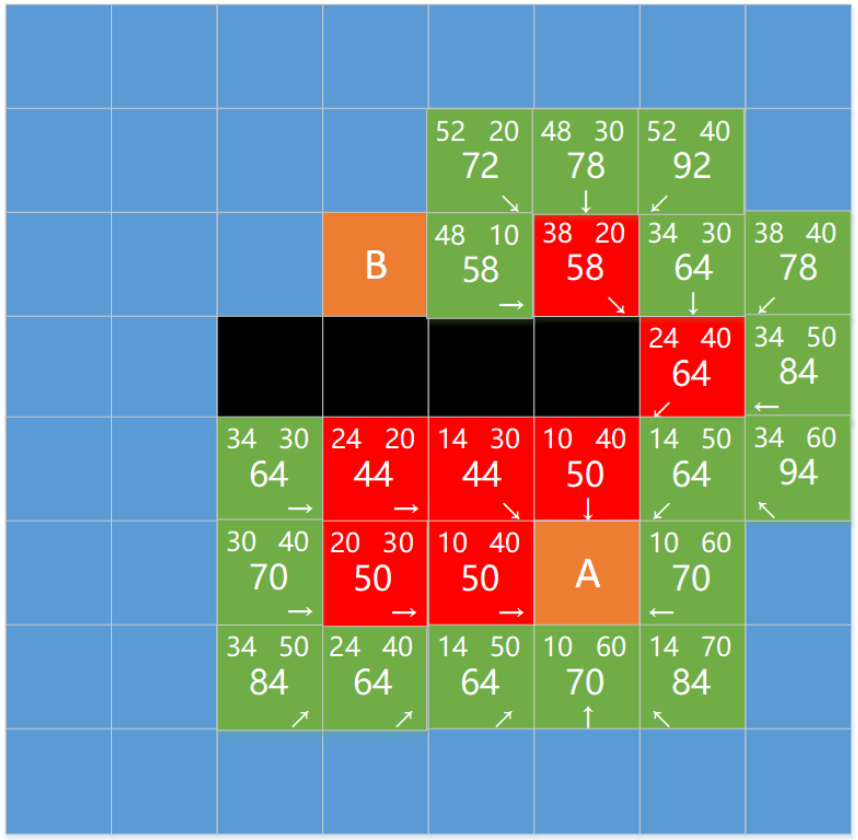


3.2 路径规划问题

■第7轮搜索的选定方格

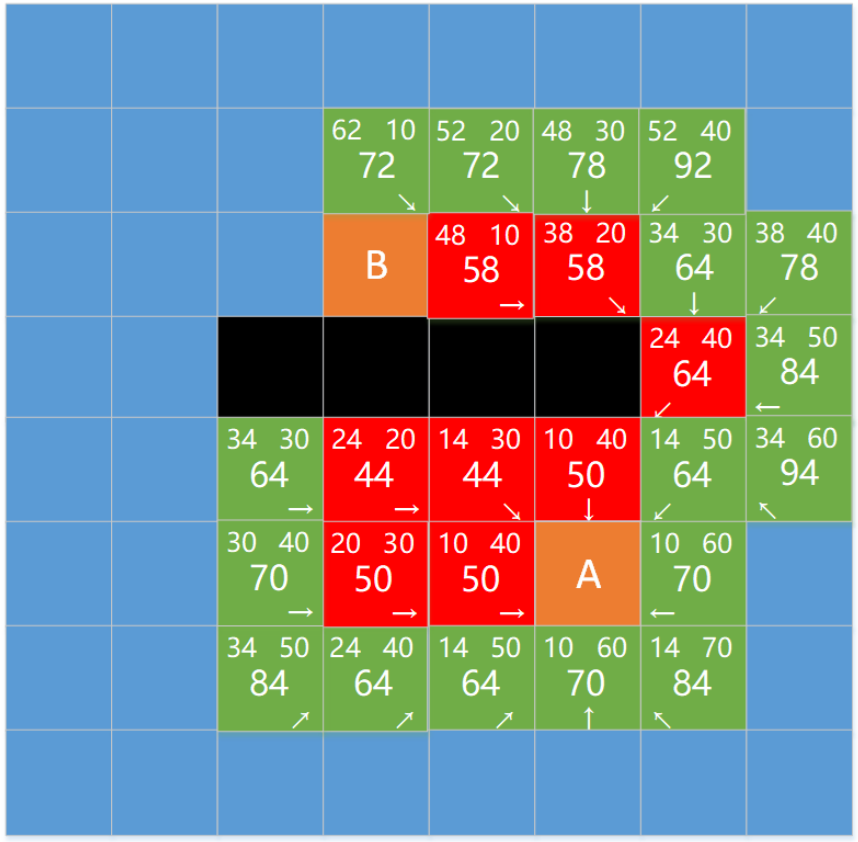


■第7轮搜索的扩展与更新方格

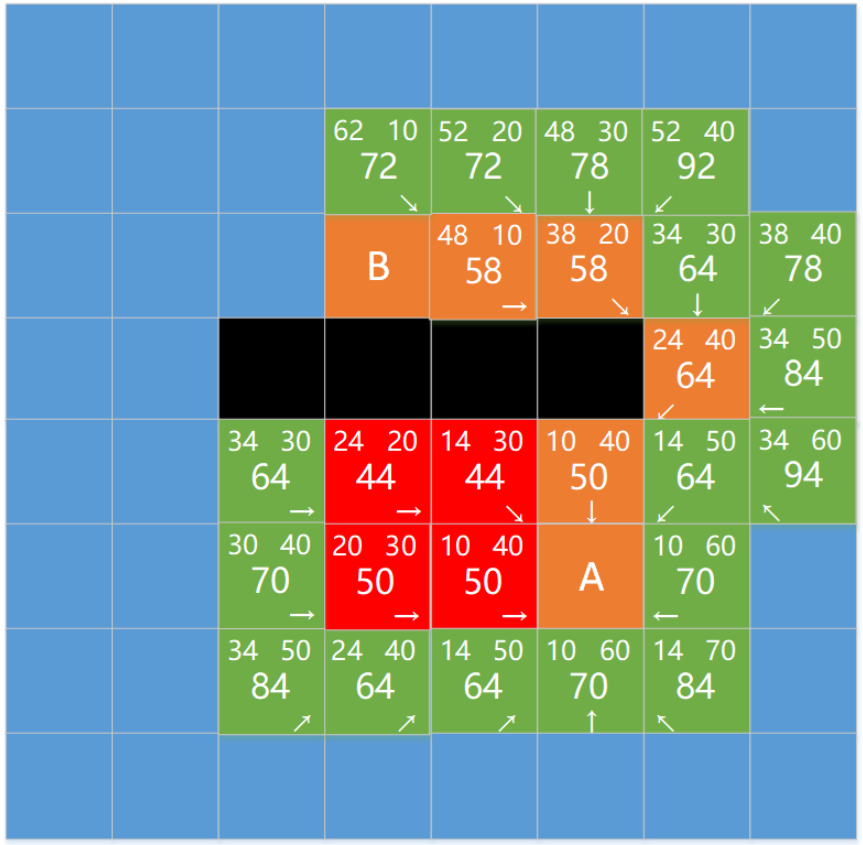


3.2 路径规划问题

■第8轮搜索



■问题的解



3.2 路径规划问题

算法总结

1. 把起始点加入 open list;
2. 查找起始点周围所有可到达或者可通过的方格，把他们加入 open list。将起始点设置为新加入的方格的父结点。
3. 将起始点从open list中删除，加入close list中。
4. 重复如下步骤：

①遍历open list，找到其中f值最小的方格。称其为选定方格。

②对选定方格做如下操作：

a. 从open list中取出放到close list中。

b. 检查与该方格相邻的每个格子，忽略其中已在close list或者不可通过的格子。

- 如果检查到的格子不在open list中，则将其加入到open list。并且将选定方格设定为他们的父结点。
- 如果检查到的格子在open list中，则检查经过选定方格到达该方格，是否有更小的G值。如果没有，不做操作。如果有，将该方格的父节点设置为选定方格，并重新计算其f值和g值。

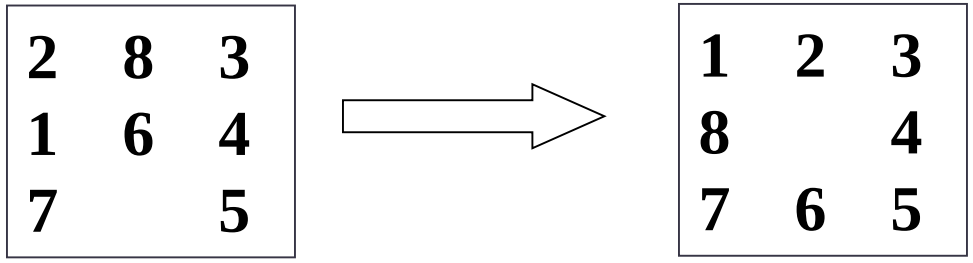
5.停止重复步骤，当：

① 把终点加入到了open list中，此时路径已找到。

② 查找路径失败，open list为空，此时没有路径。

3.3 8-数码问题

■问题定义：是指在3x3的矩阵中，其中有8个格子放置成1-8，剩下一个格子是空格。能够移动和空格相邻的格子到空格，直到这个矩阵满足每一行依次从左到右读取是有序，得到最后得到1-8有序，最后一个格子是空格。下图展示了一个案例：



3.3 8-数码问题

■定义估价函数：

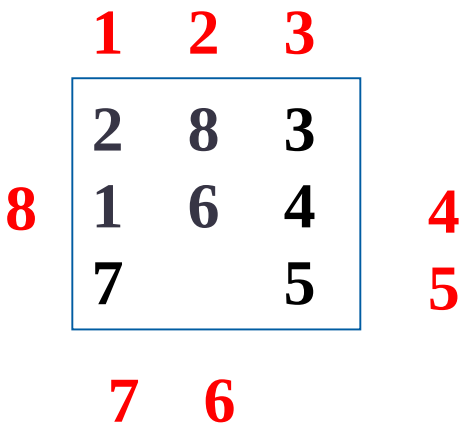
f(n) = g(n) + h(n)

g(n)为从初始节点到当前节点的步数

h(n)为当前节点“不在位”的方块数

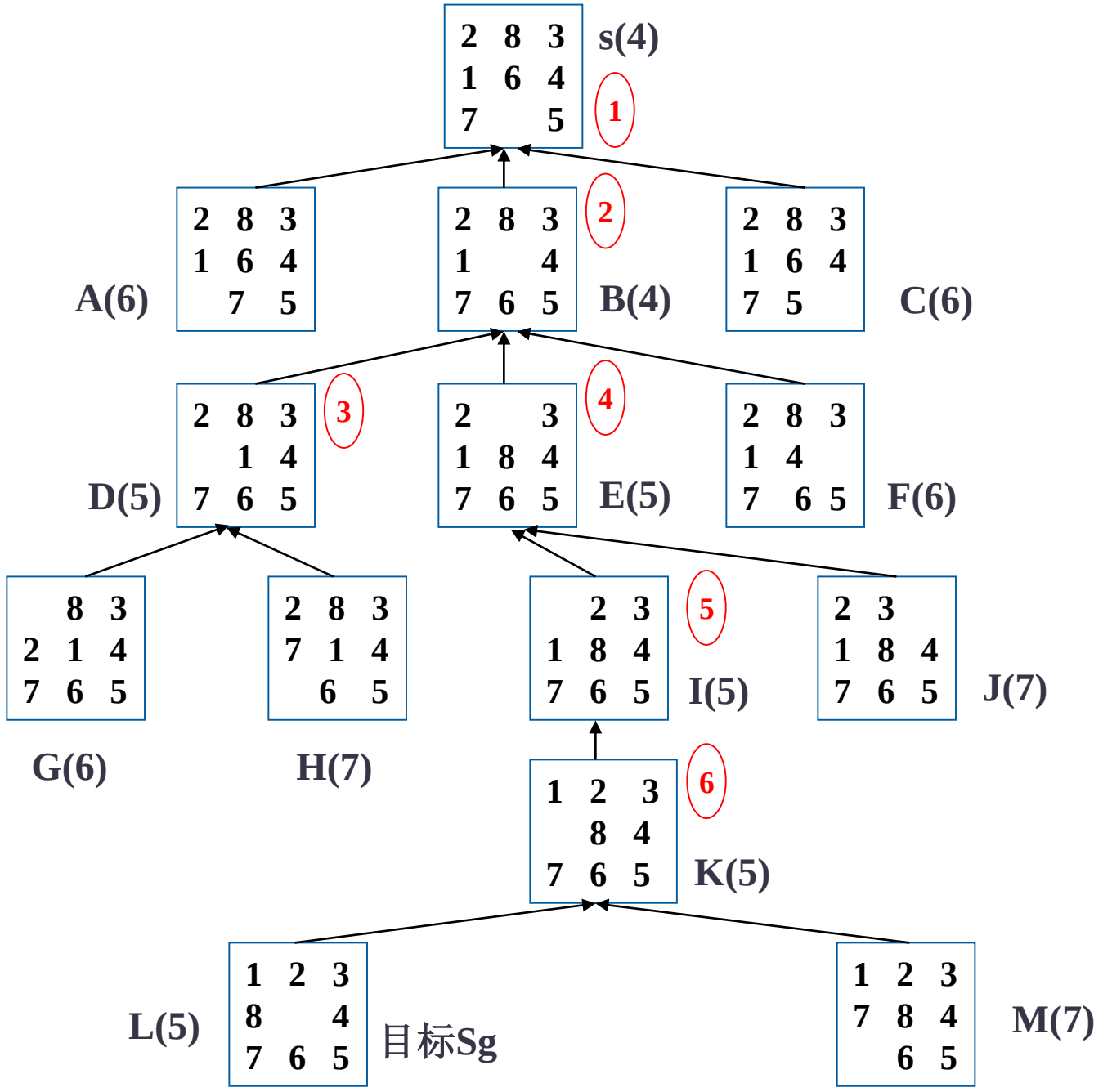
■例：右图

h(n)=4



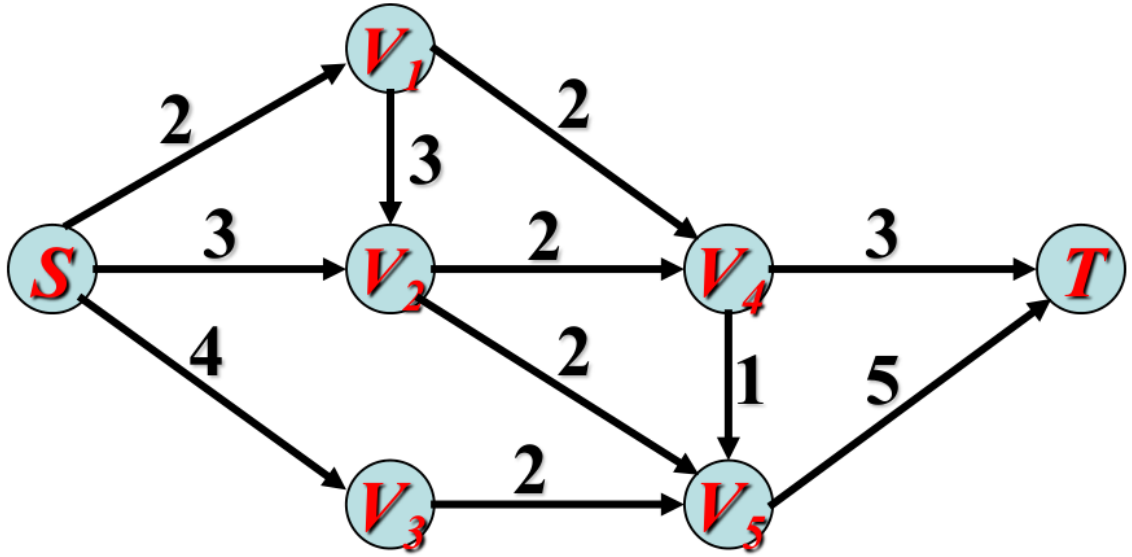
3.3 8-数码问题


EXAMPLE


2 8 3
1 6 4
7 5
s(4)
1
A(6)
2 8 3
1 6 4
7 5
2 8 3
1 4
7 6 5
2
B(4)
2 8 3
1 6 4
7 5
C(6)
2 8 3
1 4
7 6 5
3
D(5)
2 3
1 8 4
7 6 5
4
E(5)
2 8 3
1 4
7 6 5
F(6)
8 3
2 1 4
7 6 5
2 8 3
7 1 4
6 5
2 3
1 8 4
7 6 5
5
I(5)
2 3
1 8 4
7 6 5
J(7)
G(6)
H(7)
1 2 3
8 4
7 6 5
6
K(5)
1 2 3
8 4
7 6 5
目标Sg
L(5)
1 2 3
7 8 4
6 5
M(7)

3.4 多段图的最短路径问题

- 多段图：设图G=(V,E)是一个带权有向连通图，如果把顶点集合V划分成k个互不相交的子集V_i($2 \le k \le n$，$1 \le i \le k$)，使得E中的任何一条边(u,v)，必有u属于V_i，v属于V_{i+m} ($1 \le i < k, 2 \le i+m \le k$)，则称图G为多段图，称$s \in V_1$为源点，$t \in V_k$为终点。
- 多段图的最短路径问题是求从源点到终点的最小代价路径。下图展示了一个案例：

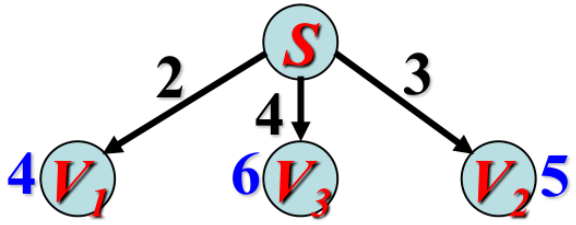


3.4 多段图的最短路径问题

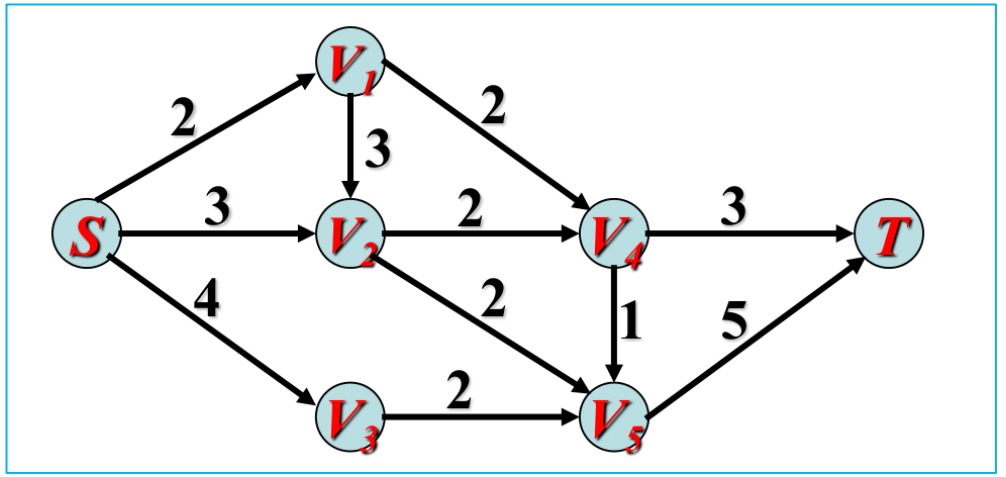
■最短路径问题中A*（A-Star）算法的估价函数f(n)=g(n)+h(n)的设计：

- ◆ g(n)是从源点到结点n的实际路径长度；
- ◆ h(n)是从结点n 到目标结点的最优路径的估计代价，这里取：由结点n发出的边的长度的最小值。

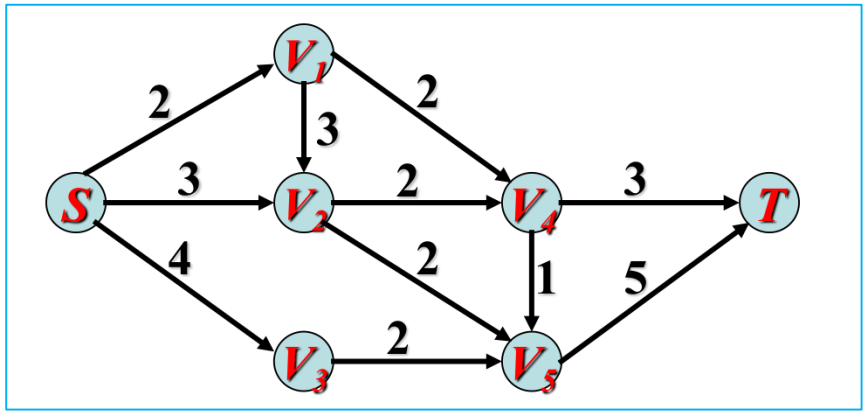
■例：算法初始运行（Step1），计算各结点的f值如下：



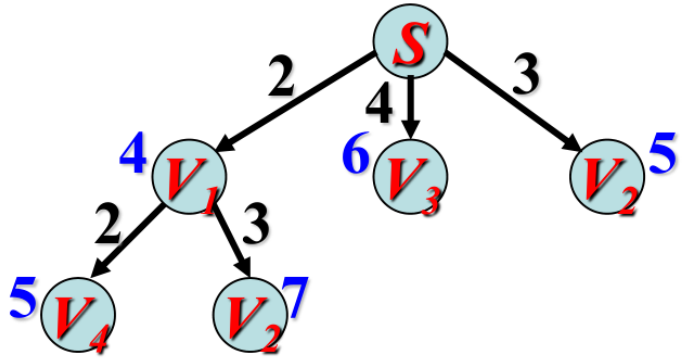
$g(V_1)=2 \quad h(V_1)=\min\{2,3\}=2 \quad f(V_1)=2+2=4$
$g(V_2)=3 \quad h(V_2)=\min\{2,2\}=2 \quad f(V_2)=3+2=5$
$g(V_3)=4 \quad h(V_3)=\min\{2\}=2 \quad f(V_3)=4+2=6$



3.4 多段图的最短路径问题

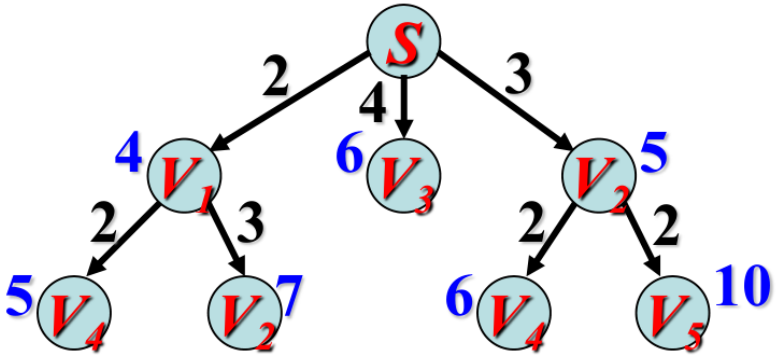


Step2：扩展结点V_1



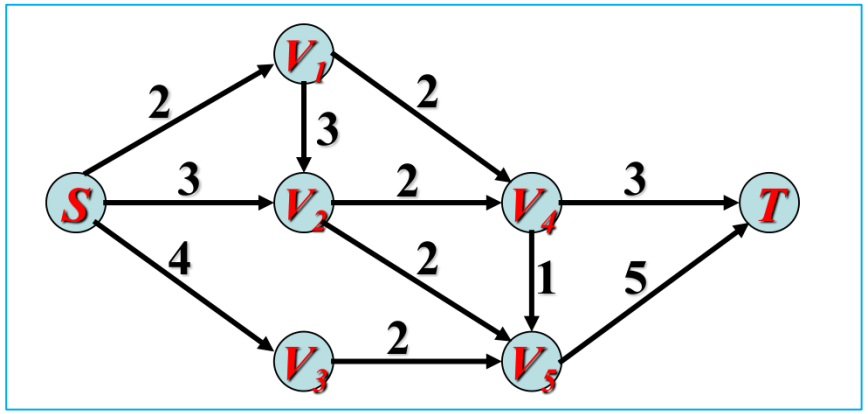
$g(V_4)=2+2=4$ $h(V_4)=\min\{1,3\}=1$ $f(V_4)=4+1=5$
$g(V_2)=2+3=5$ $h(V_2)=\min\{2,2\}=2$ $f(V_2)=5+2=7$

Step3：扩展结点V_2

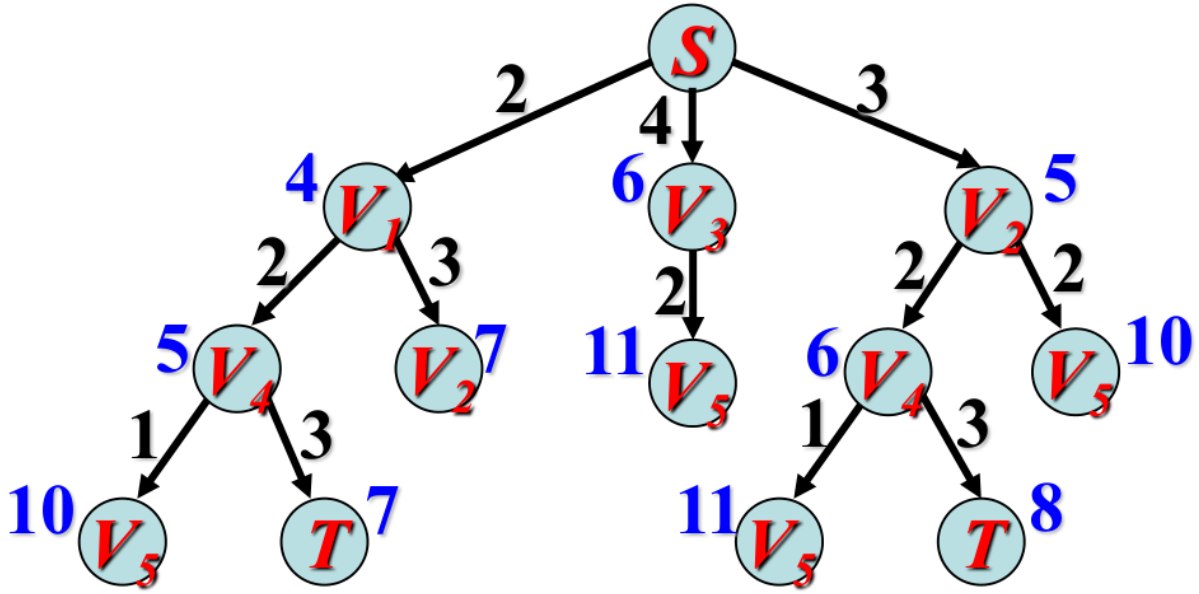


$g(V_4)=3+2=5$ $h(V_4)=\min\{1,3\}=1$ $f(V_4)=5+1=6$
$g(V_5)=3+2=5$ $h(V_5)=\min\{5\}=5$ $f(V_5)=5+5=10$

3.4 多段图的最短路径问题



■ 依次迭代，最终扩展的搜索树为：



对应的最短路径为：

$S \rightarrow V_1 \rightarrow V_4 \rightarrow T$

■ 该算法可以通过定义更合适的估计函数来改进。比如另h(n)为结点n发出边的最小值加上后续分段图各段间最小边的和。

启发式搜索A*法小结

■A* 算法使用一个合理的启发函数， 采用Best-first 策略求解优化问题，试图尽早地发现优化解。

■与分支界限策略的比较

◆A* 算法的核心是启发式函数的设计，在该类问题中，有一个明确的目标状态，启发式函数让我们尽快地发现到达目标的最优解。

◆分支界限策略是通过剪掉不能达到最优解的分支进行优化，分支界限策略的关键是 限界函数的设计。该类问题未必有明确的目标状态，而是仅求满足一般约束条件的一个最优解。

启发式搜索A*法拓展

■ 推荐文献阅读：**Empowering A* Algorithm With Neuralized Variational Heuristics for Fastest Route Recommendation. IEEE Trans. Knowl. Data Eng. 35(10): 10011-10023 (2023).Minrui Xu; Jiajie Xu; Rui Zhou; Jianxin Li; Kai Zheng; Pengpeng Zhao; Chengfei Liu.**

Abstract:

Fastest route recommendation (FRR) is crucial for intelligent transportation systems. The existing methods treat it as a path finding problem on dynamic graphs, and extend A* algorithm with neuralized travel time estimators as cost functions. However, they fail to provide effective heuristic cost due to the neglect of its admissibility and the utilization of noise path information, resulting in sub-optimal results and inefficiency. Besides, path sequentiality is also ignored, affecting algorithm accuracy as well. In this paper, we propose a variational inference based fastest route recommendation method, which follows the framework of A* algorithm and provides effective costs for routing. Specifically, we first adopt a sequential estimator to accurately estimate the travel time of a specific path. More importantly, we design a variational inference based estimator, which models the distribution of travel time between two nodes and provides an effective heuristic cost with high probability of being admissible. We further take advantage of adversarial learning to enrich the fastest path information. To the best of our knowledge, we are the first to use variational estimator to consider the admissibility of heuristics in FRR. Extensive experiments are conducted on two real-world datasets. The results verify the performance advantage of our proposed method.

THANK YOU FOR YOUR WATCHING

Teacher：张明卫
@东北大学 软件学院