chapter 6: search strategy

You Me

date: Yesterday

目录

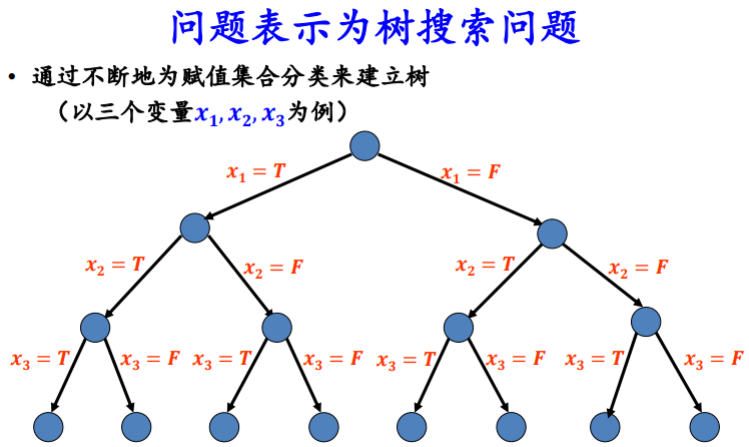

图 1: tu1

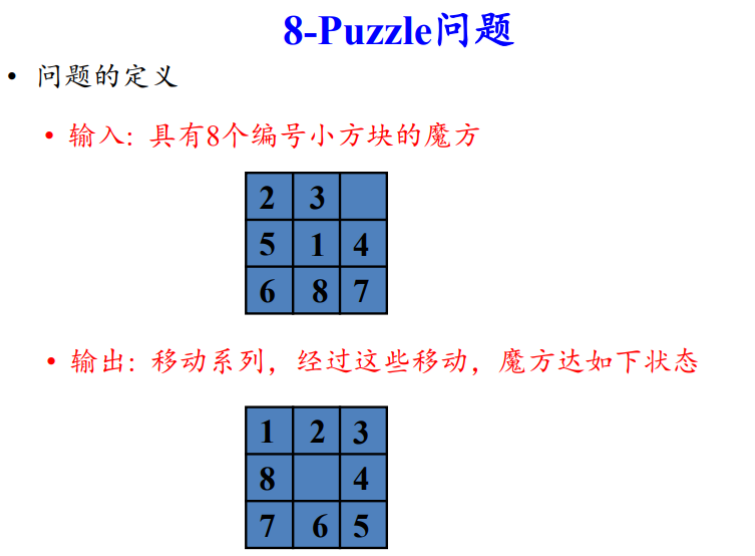

图 2: tu2

1 搜索漫谈

搜索问题无处不在, 我是说, 应该吧. 总之很多东西都能够表达为一个棵树, 此后我们在这棵树上进行一个搜索. 搜索到某些节点, 而这些节点可能就是解. 比如说, 任意的一个决策问题, 都能够转换为一个决策树的搜索问题.

Example 1. 一个 *proposition*, 是否存在一个指派使得这个 *proposition* 的值为真.

就是验证其中的那些 *propositional letters* 的各种取值, 比如说, 对于第一个 *propositional letter*, 我们作出一个决策, 即, 这个 *letter* 有两种取值, 我们有两种选择. 见图 *1*.

Example 2 (8-puzzle problem)**.** 就是那个, 不知道叫什么名字的那个. 见 *Figure 2*. 我们这里进行的任何一个决策都能够表示为树. 你看, 这个空位在右上角, 那么我们若是以这个为初始状态, 那么我们有两种选择, *3* 向右移动, 或者是 *4* 向上移动. 见 *Figure 3*

Example 3 (Hamiltonian 环问题)**.** 让我们先定义一下, 我们有一个 n 节点的连通图 $G = (V, E)$. 我们需要知道, G 中是否有 *Hamiltonian* 环.

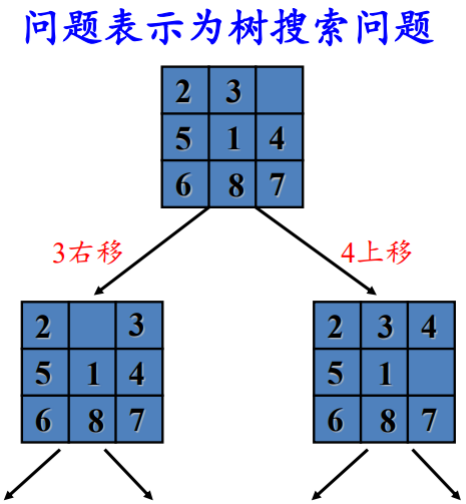

图 3: tu3

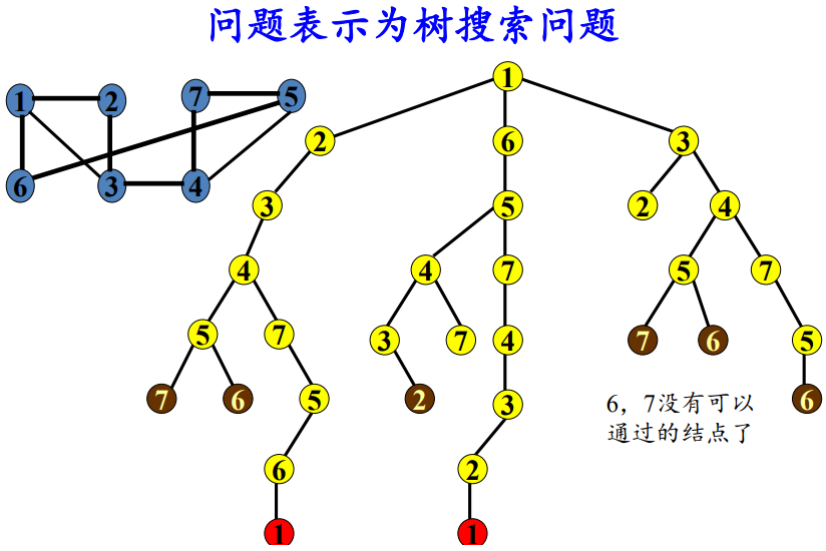

图 4: tu4

类似的，我们知道，每一个决策都能表示，那么我们还可以表示环路. 比如说我们从某一点出发，那么‘下一个点’有很多选择，那些没有经过的、相邻的点就行. 见 *Figure 4*

2 广度优先搜索和深度优先搜索

我们以前接触过这两位，是在学习树的时候.
与其说是两种策略，不如说是使用了两种数据结构. 深度优先是栈，而广度优先是队列.

2.1 BFS

下面给出 BFS 的框架:

```
// 构造由根组成的队列 Q
if Q 中的第一个元素 x 是目标节点
    stop
else
    从 Q 中删除 x, 将 x 的所有子节点放进 Q 中.
if Q == NULL
    return failed;
else goto 2
```

虽然使用了 goto 但也算是简洁了, 其实用 while 也很简洁. 我们说这里的关键就是队列, 其特点就是先进先出. 对于节点 x , 我们将其子节点放入队列, 这些子节点都搞定之后才会继续搞其他的子节点.

2.2 DFS

区别在于使用了栈.

就是说, 我们一直走一直走, 然后走到终点, 才往回退, 看看另一个选择, 总之就是先一条路走到黑. 使用栈的观点, 会看的更清晰. 每次访问一个节点, 就将其子节点塞到栈里, 然后取出站内的一个节点, 访问这个节点, 进行类似的操作. 和广度优先遍历是完全类似的. 是不是非常神奇.

```
// 构造由根组成的栈 S
if S 中的第一个元素 x 是目标节点
    stop
else
    Pop (S) ;
    Push (s); // 将 S 的子节点压进栈.
if S == NULL
    return failed;
else goto 2
```

3 搜索的优化

3.1 爬山法

基本思想: 在 DFS 的过程中, 使用贪心法确定搜索的方向. 爬山策略使用 “启发式函数” 来排序节点优先程度.

我们通过 f 来测量某一个节点, 我们说, 越接近某一值 a 则该节点距离正确答案就越近, 可以按照这一标准来排序节点.

将当前节点 x 的子节点压进栈的时候, 按照上面的 “启发式函数” 给出的排列顺序压进栈. 实际上是一种贪心算法的思路.

Example 4. *在 eight-puzzle 问题之中, 我们使用爬山法.*

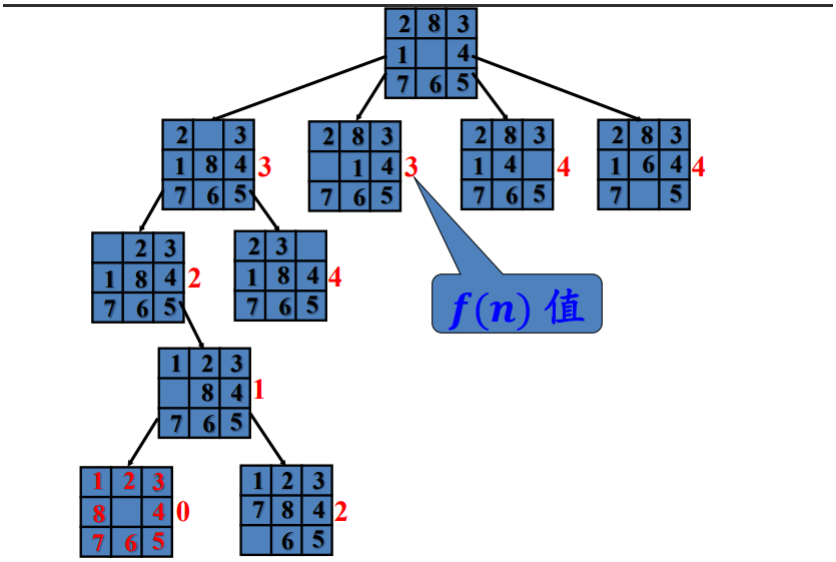

图 5: climb mountain in 8-puzzle

我们如下定义“启发式函数” $f : V \to \mathbb{R}$, 其中, 我说, V 是一个决策树上的节点的集合. 并且 $f(n)$ 是当前情况之下, 处于正常位置的格子的个数. 见 *Figure 5.*

于是乎, 我们的算法思路就是这样:

```
1  // 构造由根组成的栈 S
2  if S 中的第一个元素 x 是目标节点
3      stop
4  else
5      Pop (S) ;
6      Push (s); // 将 x 的子节点按照给出的排列顺序压入栈. 注意栈的性质, 我们应当将那些值比较高的最后塞
7  if S == NULL
8      return failed;
9  else goto 2
```

我们能够明显地看出, 这个搜索策略并不一定是最优的策略[1], 因为这只是一个贪心选择, 我们的问题不一定满足贪心选择性.

接下来我们要修改的话, 就是考虑到全局的节点.

3.2 Best-First 搜索

类似的, 我们有给出一个函数 f 来考察节点的优先程度.

基本思想: 将上面的这些数据结构换为 heap.

- 结合深度优先和广度优先的优点
- 根据函数 f 在目前产生的所有节点中选择具有最优的节点进行扩展 [2]
- 选择了全局最优解, 而爬山只是局部最优.

[1] 最优的策略应该很难找到吧

[2] 扩展又是什么意思? 我不是说故意挑刺, 但是你不能总是突然冒出一个自己编纂的词语, 然后又不解释是什么意思

```
依照 f 的值为比较的key, 构造出一个堆, 先是构造只有起点 r 的堆
if H 的根 x 是目标节点 then stop;
delete_max(H);  将 x 的子节点插入heap;
if H == NULL then failed;
else goto 2
```

3.3 分支界限

基本思想:

- 上述方法很难用于求解优化问题 [3]
- 分支界限策略可以有效地求解组合优化问题.
- 发现优化解的一个界限[4], 缩小解空间, 提高求解的效率 [5]

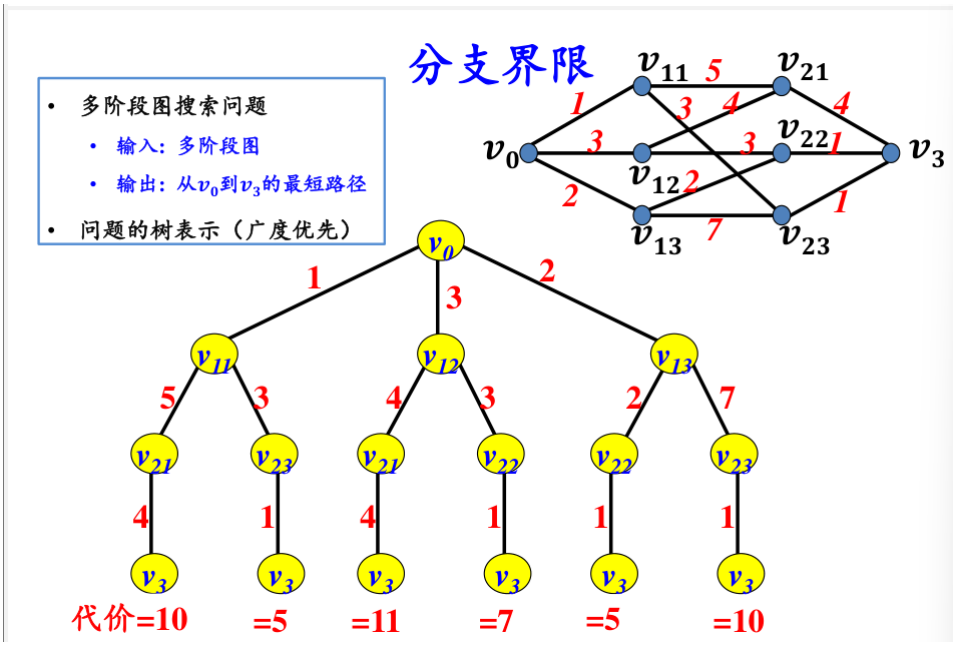

图 6: 例子

Example 5. *1. 图见 Figure 6. 使用爬山策略, 找出一个路径 (选择权重最小的边), 1 3 1 这条路*

2. 根据这个解, 进行剪枝. 已知这个路径的长度是 5 , 我们将上一层节点中大于 5 的节点剪去. 这就是缩小了解的范围.

应该是剪完了之后遍历吧, 挺无语的. 总之分支界限的方法是:

1. 产生分支的机制

[3]为什么
[4]就是硬求一个
[5]你在说什么

例. 给定$P=\{P_1,P_2,P_3\}$，$J=\{J_1,J_2,J_3\}$，$J_1\leq J_3$，$J_2\leq J_3$。

$P_1\to J_1$、$P_2\to J_2$、$P_3\to J_3$是可行解。

$P_1\to J_1$、$P_2\to J_3$、$P_3\to J_2$不可能是解。

图 7: 一个例子

2. 产生一个界限

3. 进行一个分支界限搜索, 剪除那些不可能产生优化解的分支.

4 剪枝方法与人员安排问题

4.1 问题初步处理

4.1.1 问题描述

input : 人员安排的问题之定义实际上是这样, 我们有一个工作的集合:

Definition 1. $J=\{J_i\}_{i\in\mathbb{N}}$, 他是一个偏序集, 我们将其进行一个拓扑排序 [6]

比如说我们有一个排序, $\{J_{i_k}\}$ J_{i_k} 是排序中的第 k 个工作, 意思就是这个工作由第 k 个人来担任.

Definition 2. 同时给出了一个矩阵 $C=(c_{ij})$, c_{ij} 的意思是, 第 i 个人分配了第 j 个工作所需要的时间.

Comment 1. 符号写得非常差劲捏.

output: 矩阵 $X=(x_{ij}),x_{ij}\in\{0,1\}$ 满足

$$\sum_{i,j}c_{ij}x_{ij}$$

viz. 工作所需要的代价最小. 我们这里有一个拓扑排序的例子, 见 Figure 7

4.1.2 求解思路

于是说, 我们的目的就是要遍历所有的拓扑排序, 从中找到一个最优解, 总体来说这是一个遍历的过程. 我们可以回想以前是怎么找拓扑排序的: 每次选择一个 “没有前序元素的” 的元素, 作为当前根节点的子节点. 对于这些获得的子节点也是使用这样方法, 也就是递归地处理子节点. 能够看出这其中的这个决策就是 “选取这些没有前序元素的元素”, 于是我们能够画出这个决策树, 将每一个 topo 排序列出来, 进行一个搜索.

[6] 什么是拓扑排序来着

4.1.3 topo 排序

1. 生成空树根
2. 选择偏序集中没有前序元素的元素, 作为当前根节点的子节点
3. For root 的每一个子节点, do
4. $S = S - \{v\}$
5. 将 v 作为根, 递归地处理 S

我们这就生成了一个拓扑排序对应的树, 这个树上, 从根节点到叶子的一条路就是一个拓扑排序.

我们接下来的目的是对这个方法进行一个优化.

4.2 算法的优化: 针对代价矩阵做出的优化

Proposition 1. 将代价矩阵的一行或者一列, 减去同一个数字, 不影响优化解的求解.[7]

- 代价矩阵的每行减去同一个数, 使得每一行以及每一列是找有一个零, 其余元素非负.
- 减数的和是解的一个下界.

显然, 这个下界没什么用, 不能剪枝啊. 能有什么用? 但是这并不能直接忽略了, 我们将其放在根节点上, 这样靠谱吧.

Example 6. 我们有一个代价矩阵:

$$\begin{bmatrix} 29 & 19 & 17 & 12 \\ 32 & 30 & 26 & 28 \\ 3 & 21 & 7 & 9 \\ 18 & 13 & 10 & 15 \end{bmatrix}$$

第一行 -12 , 第二行 -26 , 第三行 -3 , 第四行 -10, 有

$$\begin{bmatrix} 17 & 7 & 5 & 0 \\ 6 & 4 & 0 & 2 \\ 0 & 18 & 4 & 6 \\ 8 & 3 & 0 & 5 \end{bmatrix}$$

第二行能够 -3, 于是就有

$$\begin{bmatrix} 17 & 4 & 5 & 0 \\ 6 & 1 & 0 & 2 \\ 0 & 15 & 4 & 6 \\ 8 & 0 & 0 & 5 \end{bmatrix}$$

我们根据两个矩阵还有那个偏序集, 画出决策树. 见 *Figure 8* 以及 *Figure 9*. 能够看出, 进行优化之前的树剪了个寂寞, 就剪了一个, 没啥用. 但是优化之后就比较猛了. 我们选到的是 70 作为一个解的上界, 左边那个子树能够剪掉很多分支. 见 *Figure 10*

[7] 这该怎么证明?

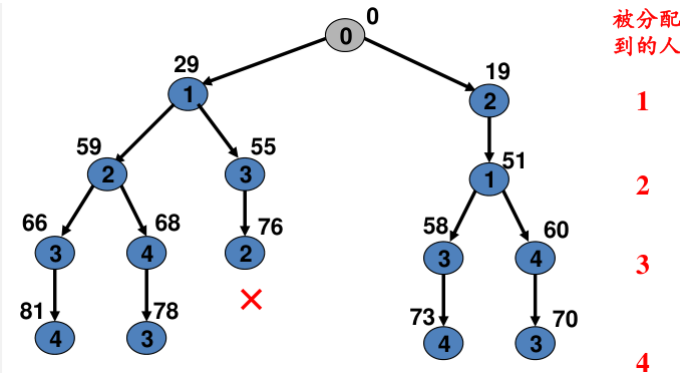

图 8: 优化前的树

- 解空间的加权树表示

$$\begin{bmatrix} 17 & 4 & 5 & 0 \\ 6 & 1 & 0 & 2 \\ 0 & 15 & 4 & 6 \\ 8 & 0 & 0 & 5 \end{bmatrix}$$

优化代价矩阵

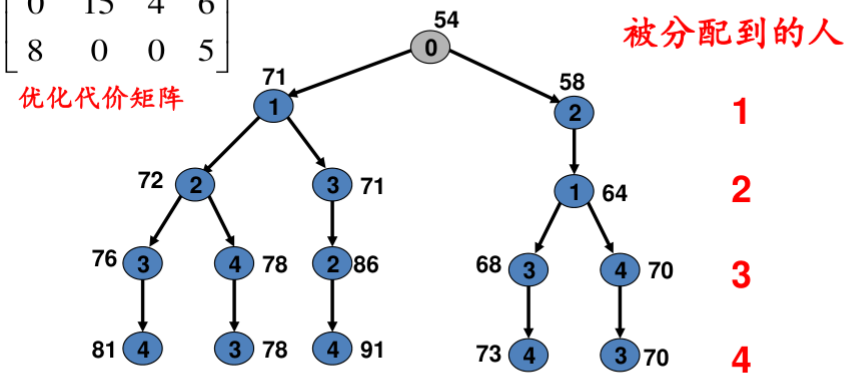

图 9: 优化后的树

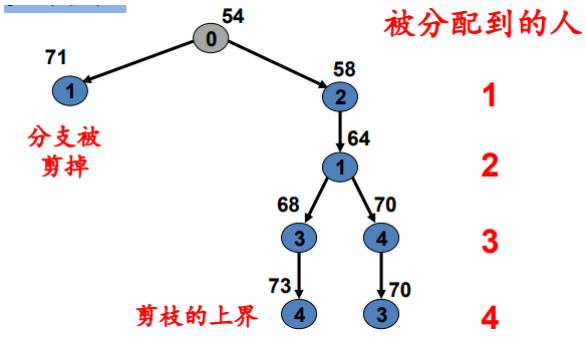

图 10: 优化后剪枝

5 旅行商问题

5.1 问题描述

在一个有向图中, 找到一个路径最短的 Hamiltonian 回路.

5.2 求解思路

这个玩意的求解思路有一些复杂, 我们现在阐述一下. 我们仍然使用决策树, 根节点放着的是所有可行解的集合. 使用爬山方划分空间, 得到一个二叉树. 按照 ppt 上的说法是这样的, 并且划分方法是如下:

1. 选定一个边 (u, v)
2. 包含了 (u, v) 的解作为左子树, 不包含 (u, v) 的解, 作为右边子树.
3. 我们根据这种信息, 更新左右两边的代价下界.
4. (u, v) 的选择原则是, 优化后的代价矩阵中 (u, v) 的值等于 0 , 并且, 右子树代价下界直接增加值[8]最大.

5.3 如何更新左子树的下界

简单来说, 就是使用了这个 (u, v) 那么, 有一些边就可以直接不选了, viz. 以 u 为起点, 以及以 v 为终点那些边都不选了. Moreover, (v, u) 也不能选了.

我们画个图就明白了, 比如说我们选取了 $(4, 6)$

$$\begin{bmatrix} \infty & 0 & 83 & 9 & 30 & 6 & 50 \\ 0 & \infty & 66 & 37 & 17 & 12 & 26 \\ 29 & 1 & \infty & 19 & 0 & 12 & 5 \\ 32 & 83 & 66 & \infty & 49 & 0 & 80 \\ 3 & 21 & 56 & 7 & \infty & 0 & 28 \\ 0 & 85 & 8 & 42 & 89 & \infty & 0 \\ 18 & 0 & 0 & 0 & 58 & 13 & \infty \end{bmatrix} \rightarrow \begin{bmatrix} \infty & 0 & 83 & 9 & 30 & \text{foo} & 50 \\ 0 & \infty & 66 & 37 & 17 & \text{foo} & 26 \\ 29 & 1 & \infty & 19 & 0 & \text{foo} & 5 \\ \text{foo} & \text{foo} & \text{foo} & \text{foo} & \text{foo} & \text{foo} & \text{foo} \\ 3 & 21 & 56 & 7 & \infty & \text{foo} & 28 \\ 0 & 85 & 8 & \text{foo} & 89 & \text{foo} & 0 \\ 18 & 0 & 0 & 0 & 58 & \text{foo} & \infty \end{bmatrix}$$

这个时候, 我们还可以进行矩阵的优化, 就是说在某一些行里面减去一些东西, 使得其代价的下界增加了. 比如说这里就是三

5.4 如何更新右子树的下界

我们说, 如果说我们不再选择这个 (u, v) 那么一定会有一个边是以 u 为起点, 而另一个边是 v 为终点的. 注意这里的措辞, 这里是两个边, 你想想, 对于每一个点, 实际上都有一进一出的边, 这里的重点在于, 现在是两个边或者以上, 用来连接 (u, v).

[8] 什么是直接增加值

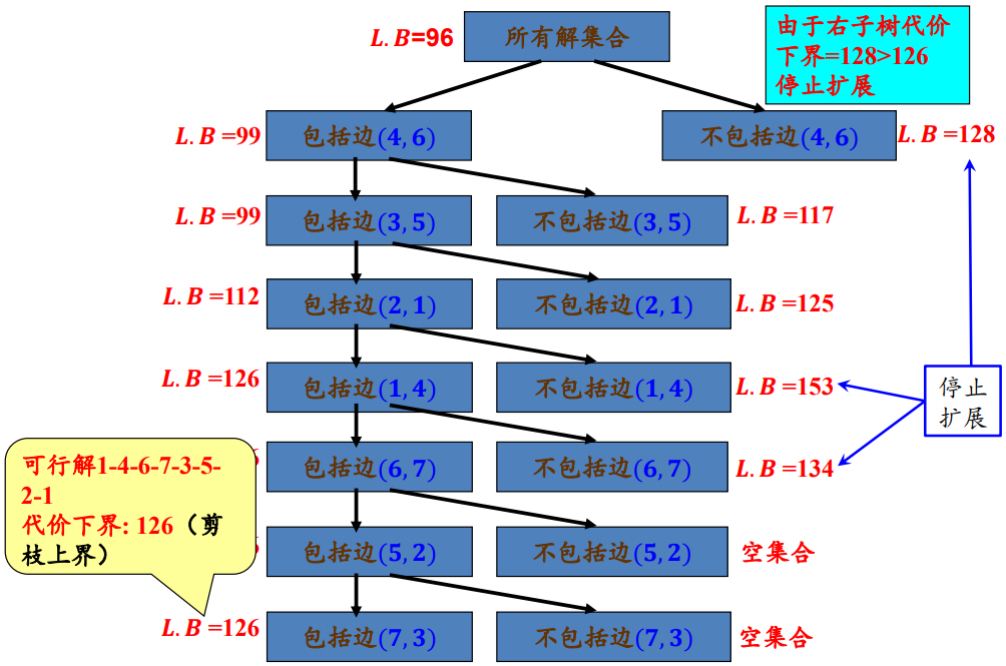

图 11: 实例

与上面的有些相反, 我们要找到 u 为起点的最短的那个边, 以 v 为终点的那个边.

$$\begin{bmatrix} \infty & 0 & 83 & 9 & 30 & 6 & 50 \\ 0 & \infty & 66 & 37 & 17 & 12 & 26 \\ 29 & 1 & \infty & 19 & 0 & 12 & 5 \\ 32 & 83 & 66 & \infty & 49 & 0 & 80 \\ 3 & 21 & 56 & 7 & \infty & 0 & 28 \\ 0 & 85 & 8 & 42 & 89 & \infty & 0 \\ 18 & 0 & 0 & 0 & 58 & 13 & \infty \end{bmatrix}$$

以这个为例子, 能够找到第四行中, 最小的是 32. 第 6 列中, 最小的是 0. 那么这个代价下界就增加了 $32+0=32$ 这样的增加就是直接增加.

并且我们说, 还有另一种表述方法, 不能使用 (u,v) 其实相当于这个边的权重变为 ∞ .

5.5 实例

下面是一个实例. L.B. 的意思是 Lower Bound

6 A* 算法

6.1 代价函数 $f*$

在这里, 我们以图最短路径为例, 我们说 $g(n)$ 是 n 节点到目标节点的优化路径的代价. $f^*(n)=g(n)+h^*(n)$ 是节点 n 的代价.

但是我们不知道, $h^*(n)$ 的值, 因为我们不知道 $h^*(n)$ 的值. 我们还是能够使用某些方法进行这个 $h^*(n)$ 的估计. 比如说, 一个点, 他不是终点, 那么他到终点, 至少还有要走一个边, 那么, $h^*(n)$ 的值一定大于, 这个相邻的边之中最小的那个. 这就是 h^* 的一个估计.

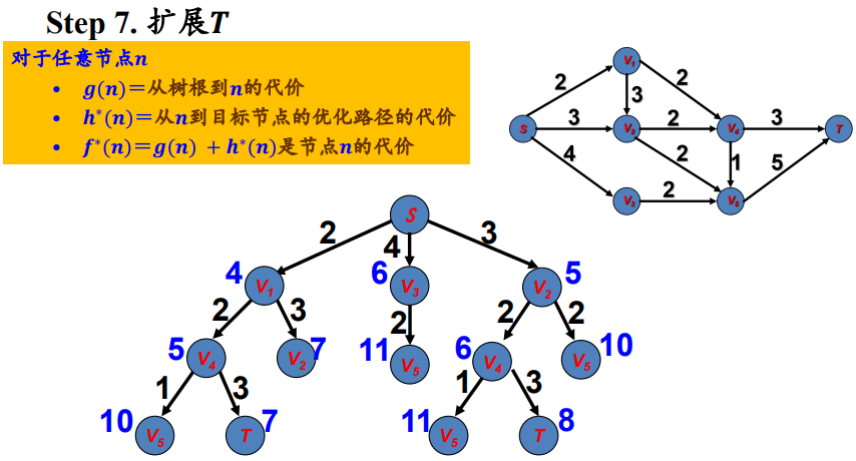

图 12: 图解

说实话非常粗糙.

6.2 图解过程

我们说, 这个 A^* 算法其实不是很难, 但是我们要熟悉这个概念, 并且熟悉这个操作. Figure 12 是一个实例. 我们只需要记住, 这个步骤其实非常简单, 就是 best-first 方法加上那个什么 f^* 函数, 一点点小优化而已.

6.3 算法正确性证明

Theorem 1. 使用 *Best-first* 策略搜索树, 如果说 A^* 选择的是目标节点, 那么该节点表示的解是优化解.

证明. 设 T 是目标节点, T for target. 我们需要证明 $f(T)$ 是最优的.

1. A* 使用的是 Best-first, 那么说对于当前任意的, 还没有被扩展的节点 v 来说, $f(T) \leq f(v)$
2. 因为对于任意的顶点 v 来说, $f(v) \leq f^*(v)$, 所以说, $\forall v \in V, f(T) \leq f^*(v)$
3. $\{f^*(v)\}_{v \in V}$ 中一定有一个是最优的[9], 设其为 s , 对应的代价为 $f^*(s)$
4. 明显 $g(T) = f(T)$, 这是因为 $h(T) = 0$ 所以说
$$g(T) \leq f^*(s)$$
5. 因为 $f^*(s)$ 是最优的, 那么说, 所有到 T 的路径长度都小于 $f^*(s)$, i.e. $g(T) \geq f^*(s)$ 综合一下就有
$$g(T) = f^*(s) \quad \square$$

另一个证明. 设 A* 算法找到了路径 P, 这个路径来到了终点 T , 然后我说, 这个路径实际上不是最小的.

然后说, 有另一个更小的路径 Q . 既然说这两个路径是不一样的, 那么说 Q 一定存在节点 v 不在路径 P 之中 (are you sure)

[9] are you sure

设从 v 开始, 两个路径开始不同, 有:

$$f(v) \leq g_Q(T) < g_P(T) \leq f(T)$$

于是就有, $f(v) \leq f(T)$. 这和什么矛盾呢? 这就和 Best-first 方法矛盾了. □