第5章 搜索算法

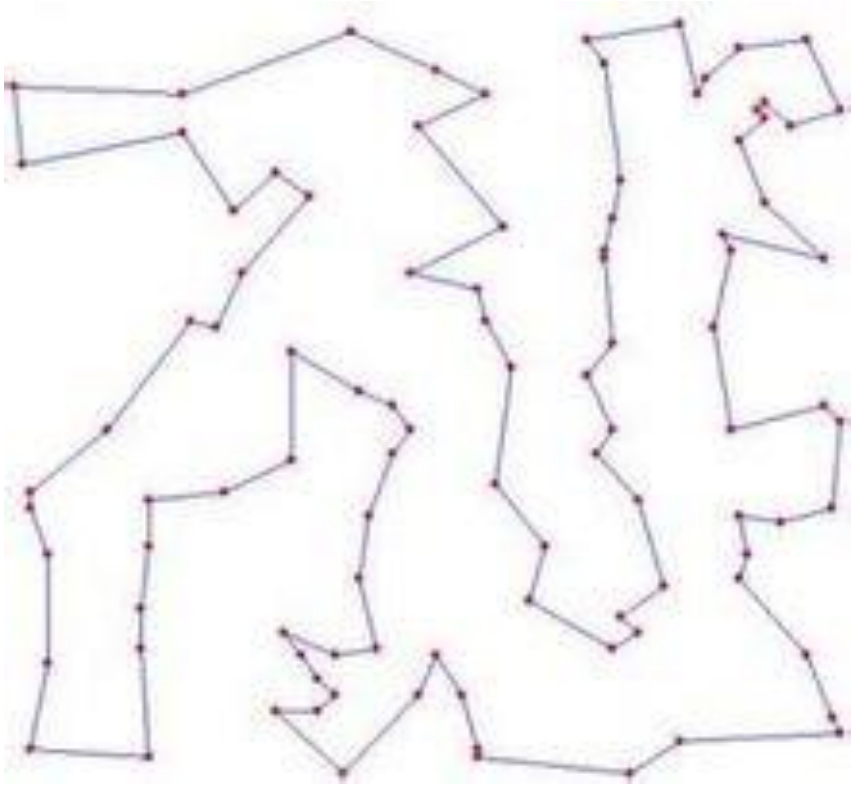
旅行商问题

内容

- 旅行商问题
  - 深度优先搜索
  - 宽度优先搜索
  - 优先队列搜索
- 搜索算法应用
  - 倒水问题
  - 八数码问题
  - 传教士野人过河问题

旅行商问题

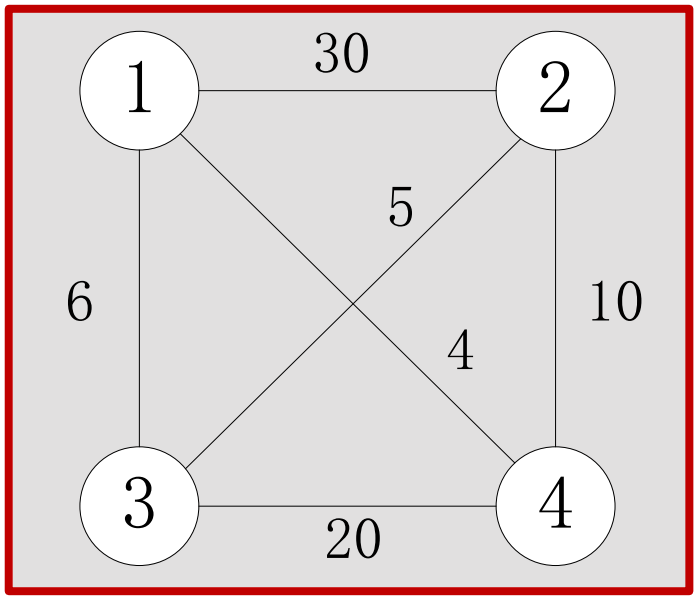
- 旅行商（TSP）问题是在寻求单一旅行者由起点出发，通过所有给定的顶点之后，最后再回到原点的最小路径成本。



旅行商问题

- 设G=(V, E)是一个带权无向图，图中各边的费用（权）为正数。图中一条周游路线是包括V中的每个顶点在内的一条回路。周游路线的费用是这条路线上所有边的费用之和。TSP问题要在图G中找出费用最小的周游路线。

例子



4顶点带权图

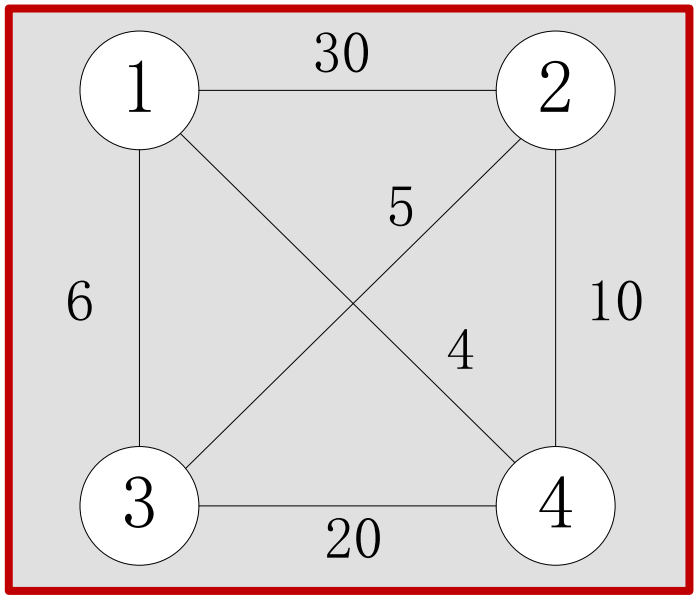
邻接矩阵

	1	2	3	4
1	0	30	6	4
2	30	0	5	10
3	6	5	0	20
4	4	10	20	0

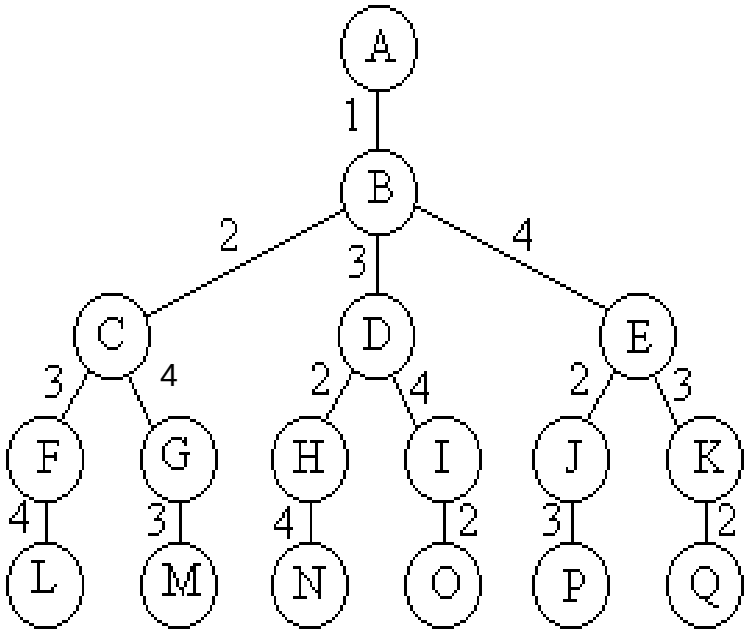
邻接表

1	{2, 3, 4}
2	{1, 3, 4}
3	{1, 2, 4}
4	{1, 2, 3}

例子



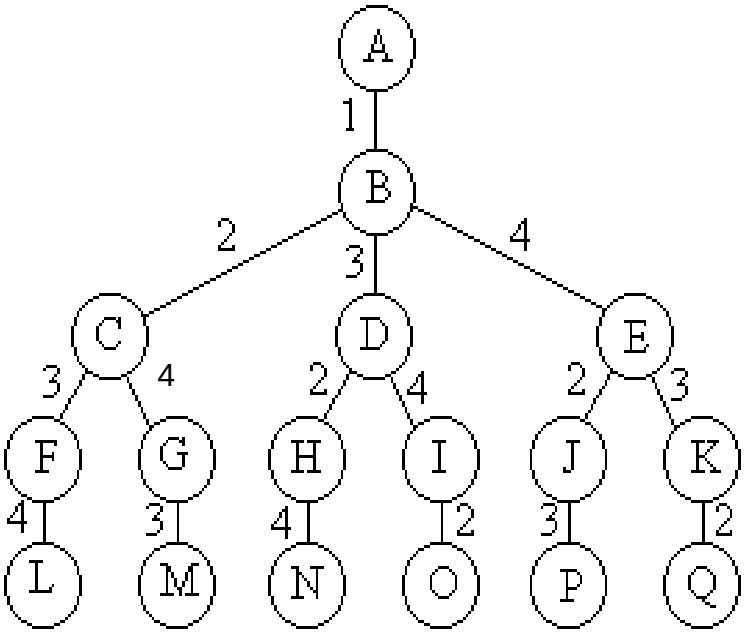
4顶点带权图



TSP问题的解空间树

TSP问题的解空间树

- 从树的根结点到任一叶结点的路径定义了图G的一条周游路线。
- 解空间树中叶结点个数为(n-1)!。



穷举法

- 穷举所有可能的路线，求出每条路线的成本，得到一个最小的路径成本路线。

$$1 \to 2 \to 3 \to 4$$

$$1 \to 2 \to 4 \to 3$$

$$1 \to 3 \to 2 \to 4$$

$$1 \to 3 \to 4 \to 2$$

$$1 \to 4 \to 2 \to 3$$

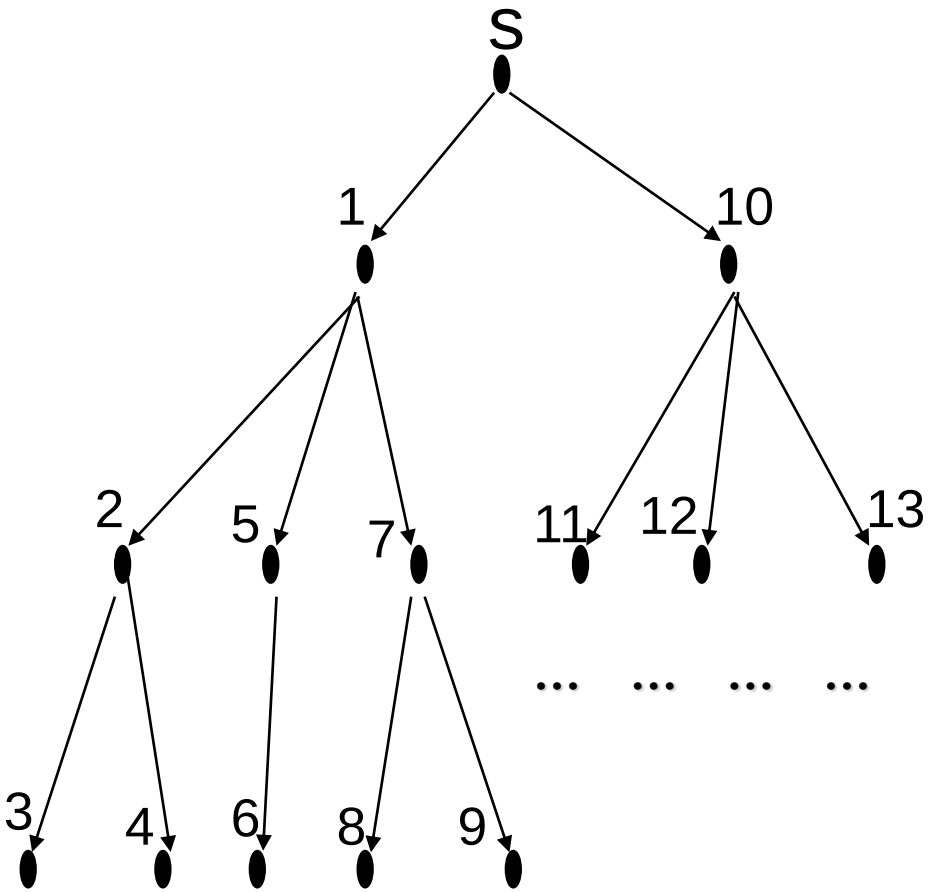
$$1 \to 4 \to 3 \to 2$$

- 算法时间复杂度：$T(n) = (n-1)!$

内容

- 旅行商问题
  - 深度优先搜索
  - 宽度优先搜索
  - 优先队列搜索
- 搜索算法应用
  - 倒水问题
  - 八数码问题
  - 传教士野人过河问题

深度优先搜索

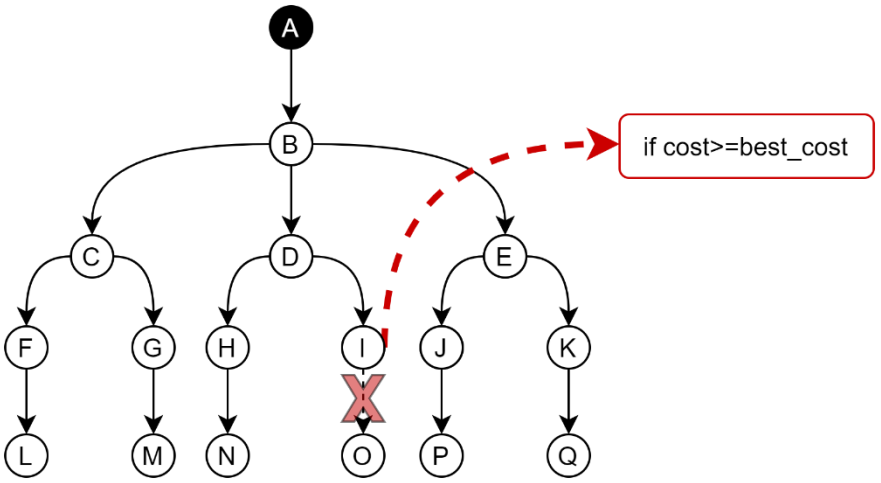


深度优先搜索

- 为了表示解空间搜索的轨迹，使用open表和close表
- open表是一个**堆栈**，即后进先出（LIFO）数据结构，记录的是待扩展的活结点，即其子状态未被搜索的结点。
- open表中结点的排列次序就是搜索的次序。
- close表记录了已扩展过的结点。

限界函数-剪枝操作

- 用限界函数剪去得不到最优解的子树。
- 在解TSP问题的搜索算法中，如果从根结点到当前扩展结点处的部分周游路线的费用已超过当前找到的最优的周游路线费用，则可以断定以该结点为根的子树中不含最优解，因此可将该子树剪去，即不再对该子树的后续结点遍历下去。



深度优先搜索

DEPTH_FIRST_SEARCH_TSP

best_cost = ∞

open = [root]　# 开始时，栈open中只有根结点

close = ϕ

d = 深度限制值

while open $\neq \phi$

　cur = open.pop

　close.push(cur)

　if cur.cost >= best_cost

　　continue

　if cur为叶结点

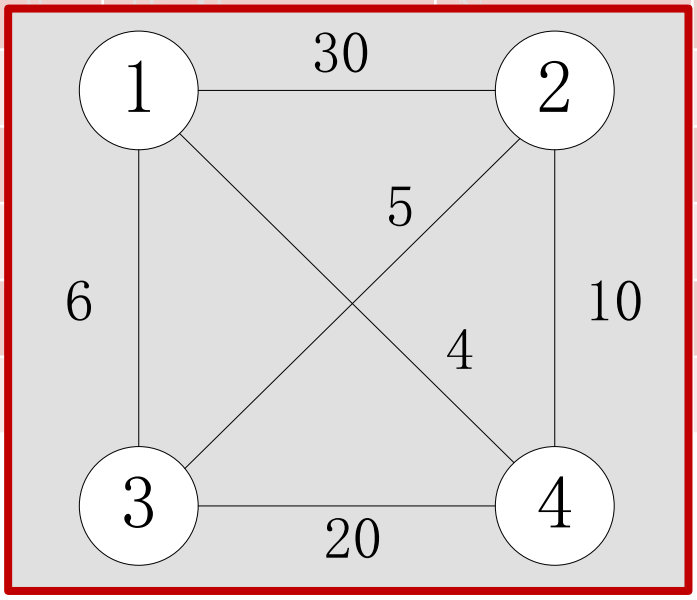
　　检测是否是更优解，若是，则修改当前最优解;

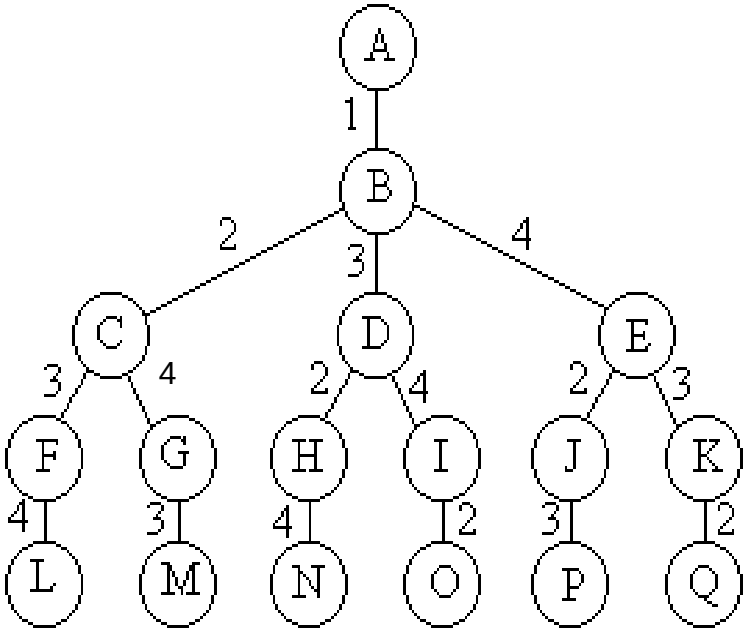
　else

　　生成cur的所有满足条件的子结点;

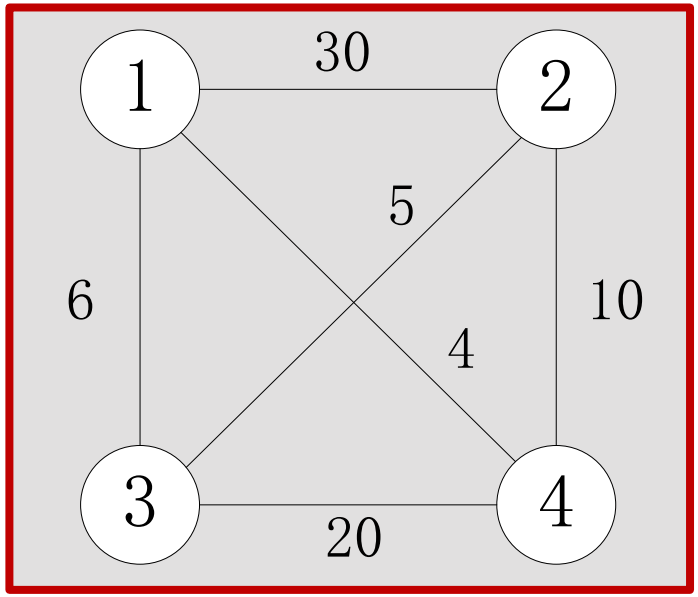
　　将子结点，按生成的次序**倒序**加入open中

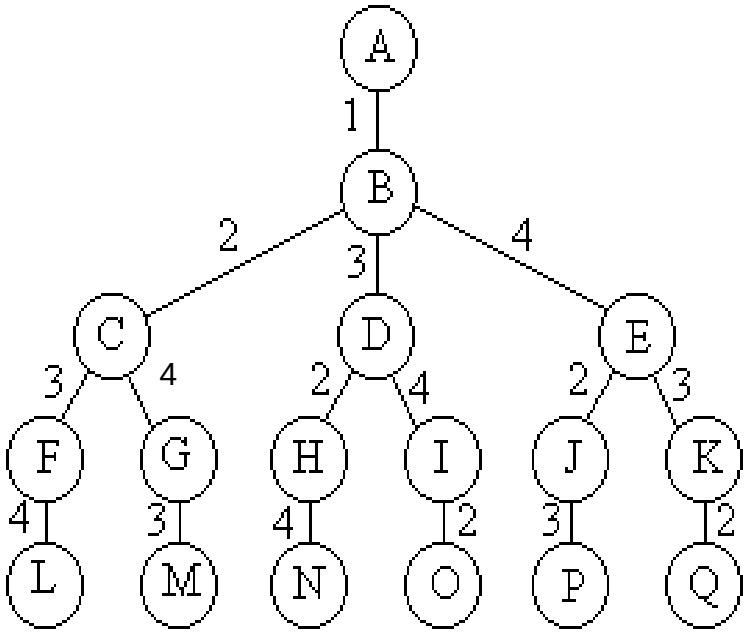
迭代	open表	当前结点	当前费用：当前路径	最优值：最优解
0	{**B**}			∞ : []
1	{E, D, **C**}	B	0: [1]	∞ : []
2	{E, D, G, **F**}	C	30: [1, 2]	∞ : []
3	{E, D, G, **L**}	F	35: [1, 2, 3]	∞ : []
4	{E, D, **G**}	L	55: [1, 2, 3, 4]	59: [1, 2, 3, 4]
5	{E, **D**}	G	40: [1, 2, 4]	59: [1, 2, 3, 4]
6	{E, I, **H**}	D	6: [1, 3]	59: [1, 2, 3, 4]
7	{E, I, **N**}	H	11: [1, 3, 2]	59: [1, 2, 3, 4]





迭代	open表	当前结点	当前费用：当前路径	最优值：最优解
8	{E, **I**}	N	21: [1, 3, 2, 4]	25: [1, 3, 2, 4]
9	{**E**}	I	26: [1, 3, 4]	25: [1, 3, 2, 4]
10	{K, **J**}	E	4: [1, 4]	25: [1, 3, 2, 4]
11	{K, **P**}	J	14: [1, 4, 2]	25: [1, 3, 2, 4]
12	{**K**}	P	19: [1, 4, 2, 3]	25: [1, 3, 2, 4]
13	{}	K	24: [1, 4, 3]	25: [1, 3, 2, 4]

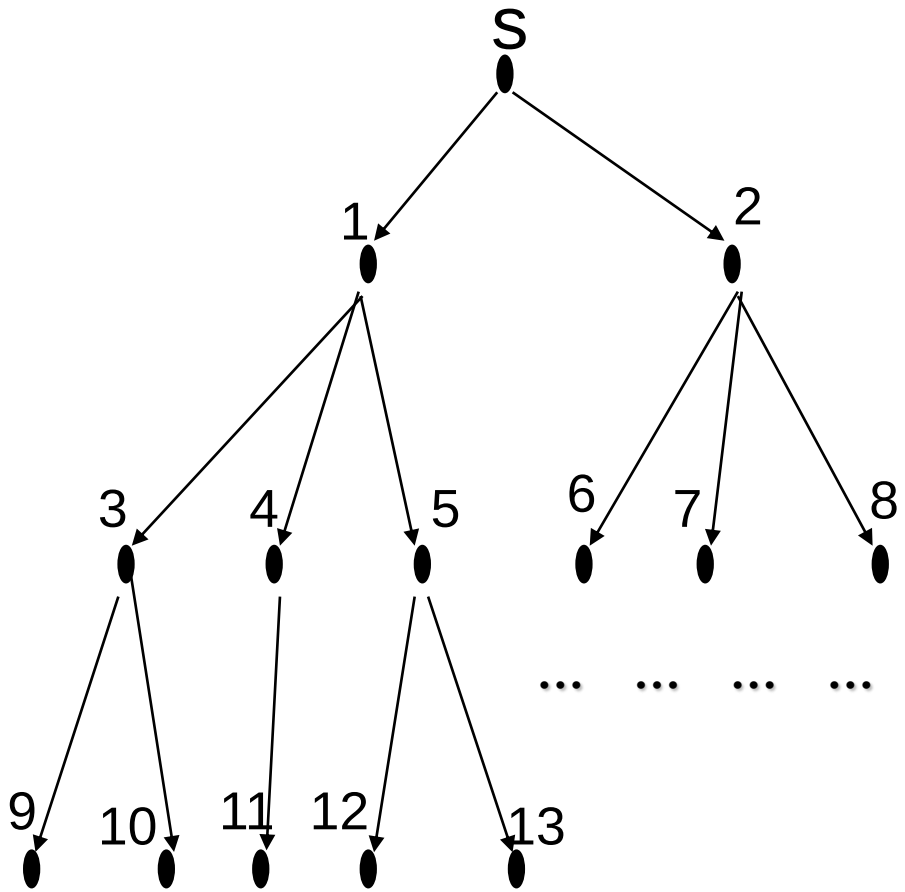




内容

- 旅行商问题
  - 深度优先搜索
  - 宽度优先搜索
  - 优先队列搜索
- 搜索算法应用
  - 倒水问题
  - 八数码问题
  - 传教士野人过河问题

宽度优先搜索



宽度优先搜索

- open表是一个队列，即先进先出（FIFO）数据结构，记录的是待扩展的活结点，即其子状态未被搜索的结点。
- open表中结点的排列次序就是搜索的次序。
- close表记录了已扩展过的结点。

宽度优先搜索

BREADTH_FIRST_SEARCH_TSP

best_cost = ∞

open = [root] # 开始时，队列open表中只有根结点

close = ϕ

while open $\neq \phi$

cur = open.get

close.put(cur)

if cur.cost >= best_cost

continue

if cur为叶结点

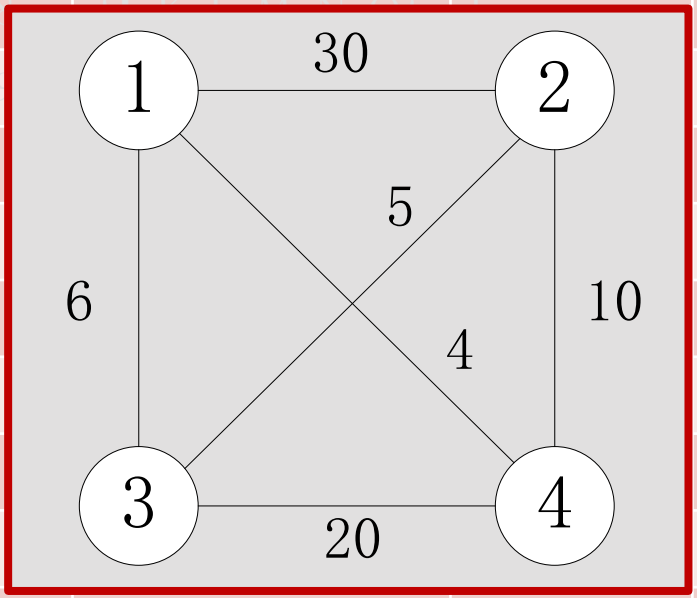
检测是否是更优解，若是，则修改当前最优解;

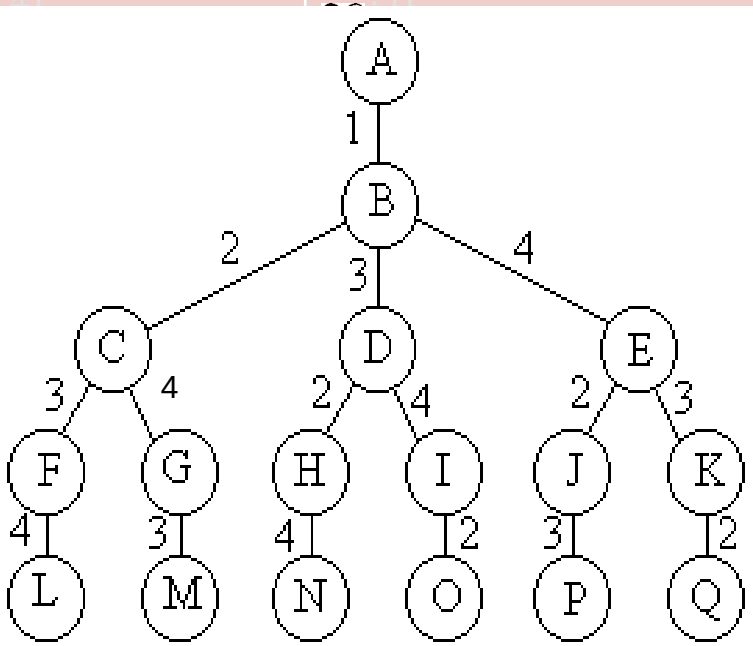
else

生成cur的所有满足条件的子结点;

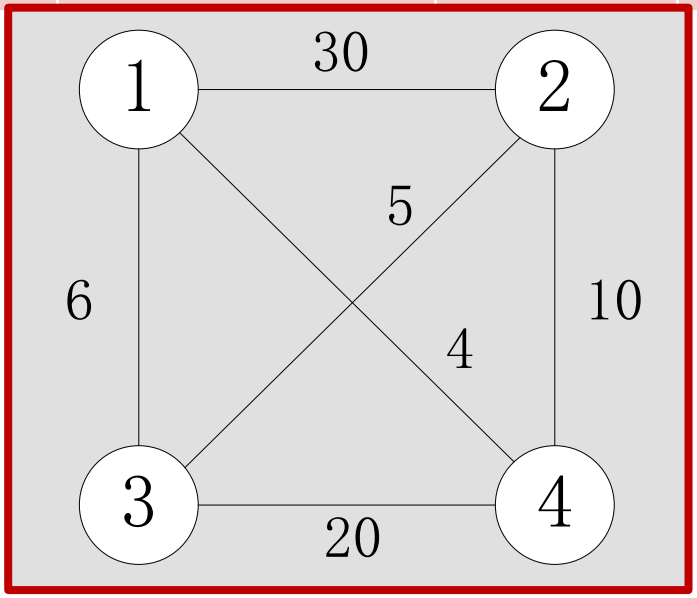
将子结点，按生成的次序加入open表中

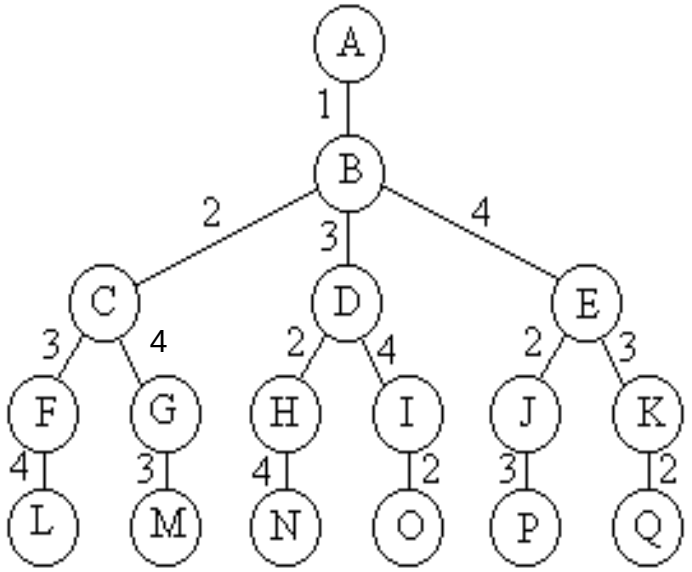
迭代	open表	当前结点	当前费用：当前路径	最优值：最优解
0	{**B**}			∞ : []
1	{**C**, D, E}	B	0: [1]	∞ : []
2	{**D**, E, F, G}	C	30: [1, 2]	∞ : []
3	{**E**, F, G, H, I}	D	6: [1, 3]	∞ : []
4	{**F**, G, H, I, J, K}	E	4: [1, 4]	∞: []
5	{**G**, H, I, J, K, L}	F	35: [1, 2, 3]	∞: []
6	{**H**, I, J, K, L, M}	G	40: [1, 2, 4]	∞: []
7	{**I**, J, K, L, M, N}	H	11: [1, 3, 2]	∞: []





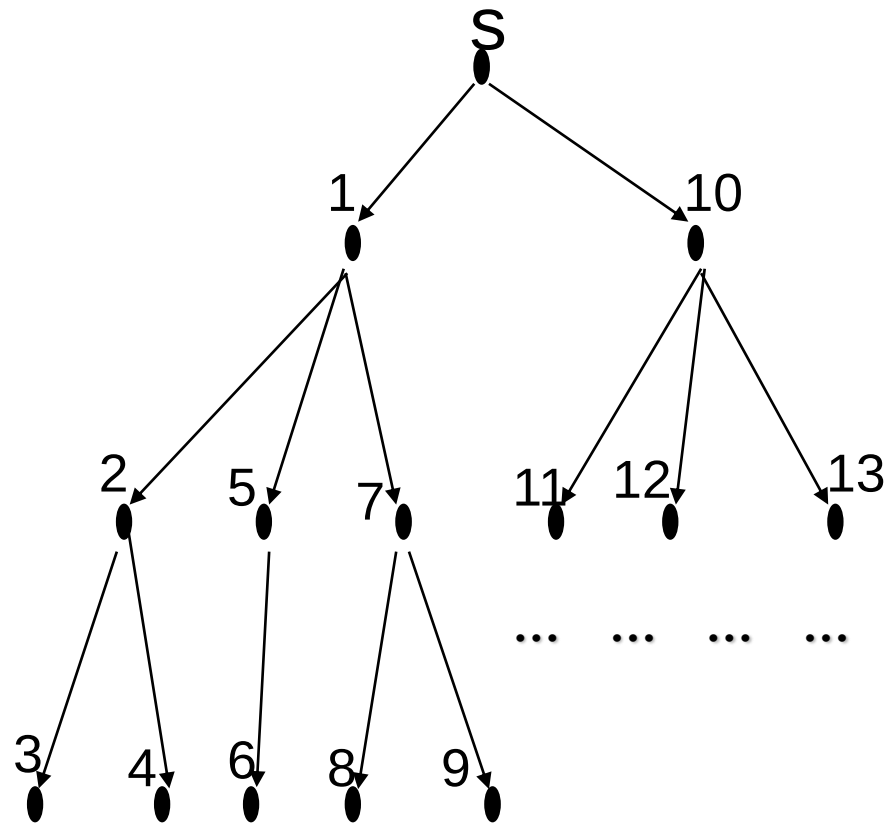
迭代	open表	当前结点	当前费用：当前路径	最优值：最优解
8	{**J**, K, L, M, N, O}	I	26: [1, 3, 4]	∞ : []
9	{**K**, L, M, N, O, P}	J	14: [1, 4, 2]	∞ : []
10	{**L**, M, N, O, P, Q}	K	24: [1, 4, 3]	∞ : []
11	{**M**, N, O, P, Q}	L	55: [1, 2, 3, 4]	59: [1, 2, 3, 4]
12	{**N**, O, P, Q}	M	60: [1, 2, 4, 3]	59: [1, 2, 3, 4]
13	{**O**, P, Q}	N	21: [1, 3, 2, 4]	25: [1, 3, 2, 4]
14	{**P**, Q}	O	36: [1, 3, 4, 2]	25: [1, 3, 2, 4]
15	{**Q**}	P	19: [1, 4, 2, 3]	25: [1, 3, 2, 4]
16	{}	Q	29: [1, 4, 3, 2]	25: [1, 3, 2, 4]

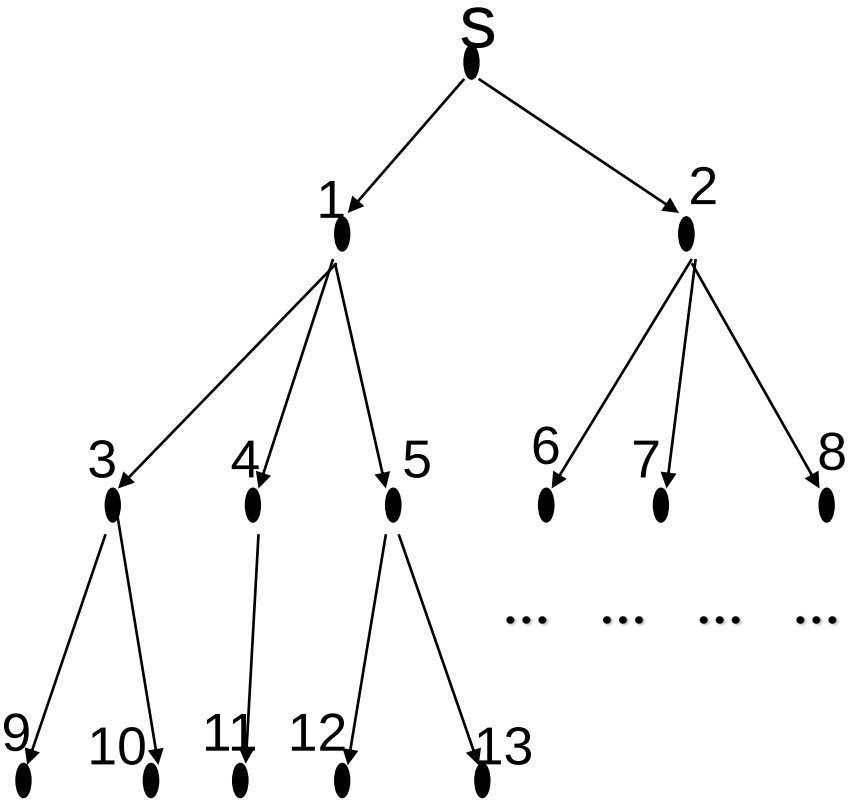




深度和宽度优先搜索比较

- 对于旅行商问题，哪种搜索策略较好？为什么？





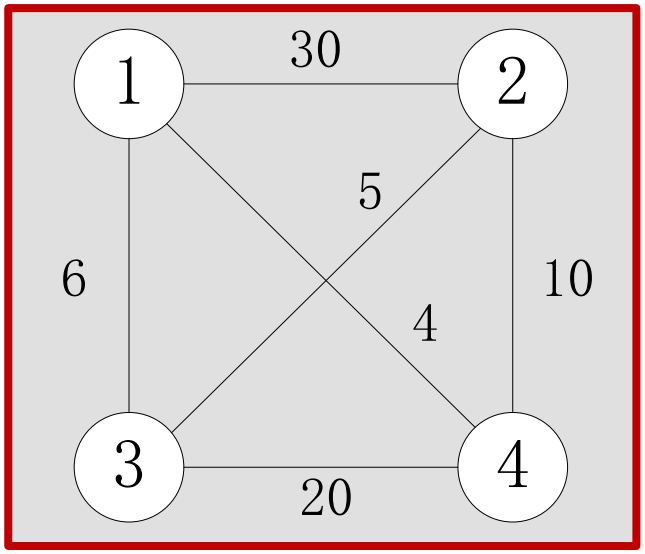
内容

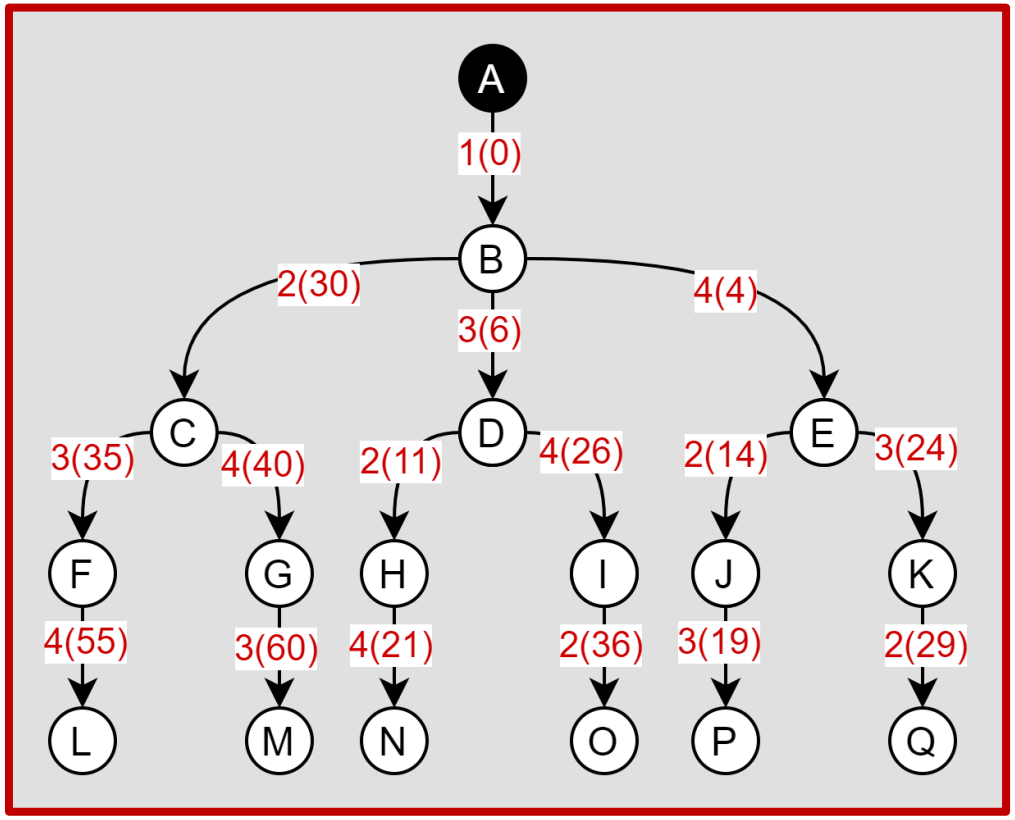
- 旅行商问题
  - 深度优先搜索
  - 宽度优先搜索
  - 优先队列搜索
- 搜索算法应用
  - 倒水问题
  - 八数码问题
  - 传教士野人过河问题

搜索策略——盲目搜索

- 不利用与特定问题相关的任何信息，按固定的步骤进行搜索；
- 由于没有可参考的信息，因此只要能匹配的操作算子都需运用，这会搜索出更多的状态，生成出较大的状态空间图。
- 例如：深度优先和宽度优先搜索

优先队列搜索


1
2
3
4
30
5
6
10
4
20

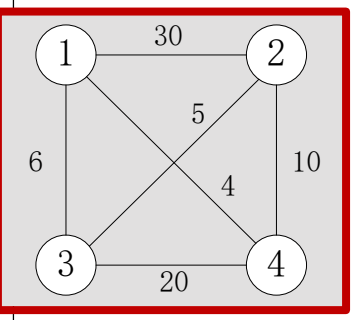

A
1(0)
B
2(30)
3(6)
4(4)
C
D
E
3(35)
4(40)
2(11)
4(26)
2(14)
3(24)
F
G
H
I
J
K
4(55)
3(60)
4(21)
2(36)
3(19)
2(29)
L
M
N
O
P
Q

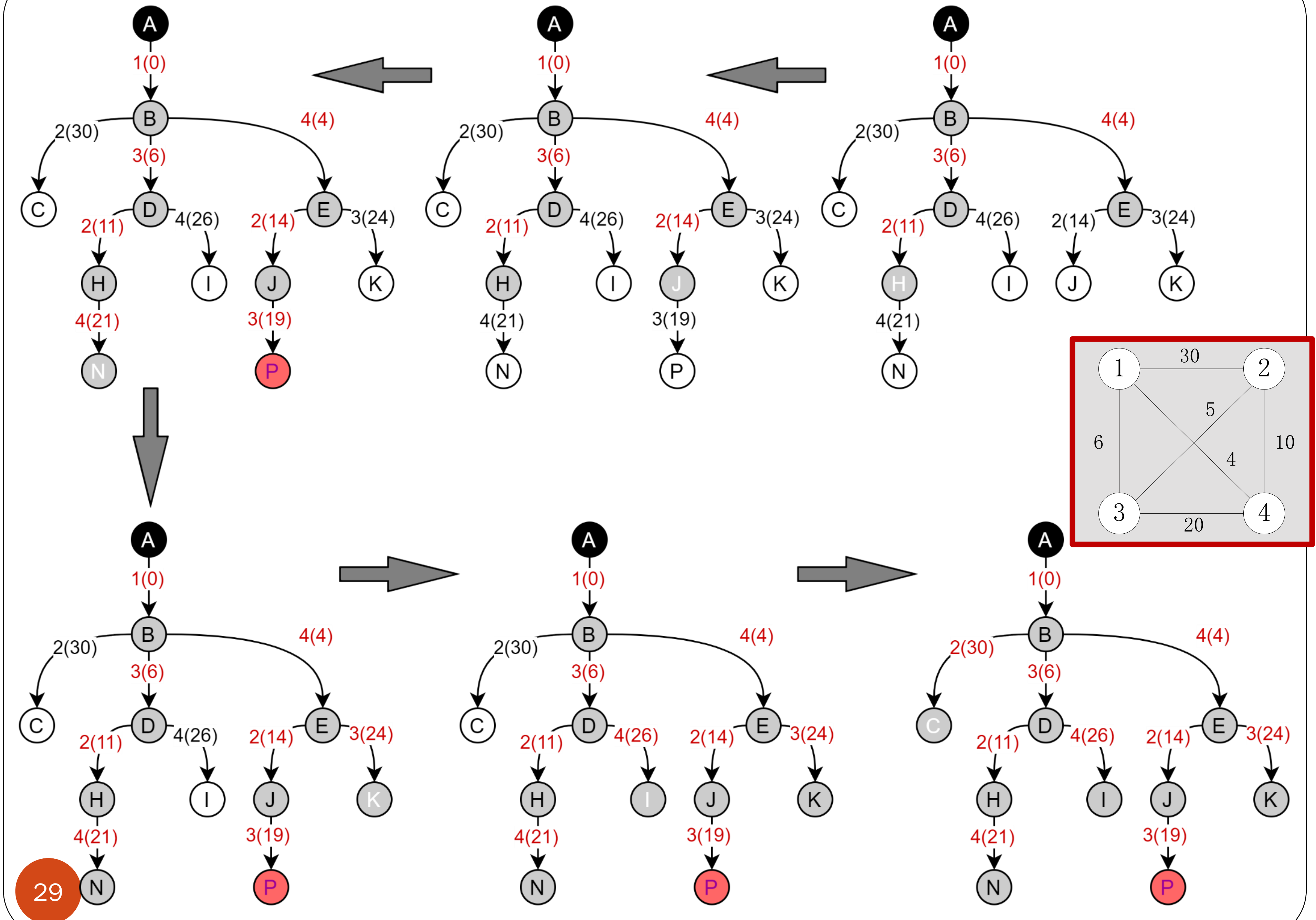
优先队列搜索

- 利用特定问题领域可应用的知识，动态地确定调用操作算子的步骤；
- 将open表中的结点组织成一个优先队列，并按优先队列中规定的结点优先级选取最高的下一个结点成为当前扩展结点。
- 优先队列搜索一般要优于盲目搜索。

优先队列搜索

- 采用优先队列搜索算法求解TSP问题时，其**优先级f(n)**是从初始结点到n结点的实际代价，即已走过路径的费用。
- open表中记录的是待扩展的活结点，是一个**最小堆**数据结构，已走过的路径的费用越小，优先级越高。


1
2
3
4
30
5
6
10
4
20



A
1(0)
B
2(30)
3(6)
4(4)
C
D
E
2(11)
4(26)
2(14)
3(24)
H
I
J
K
4(21)
3(19)
N
P
1
2
3
4
30
5
6
10
4
20

优先队列搜索

PRIORITY_SEARCH_TSP

best_cost = ∞

open = [root] # 开始时，最小堆open表中只有根结点

close = ϕ

while open $\neq \phi$

cur = **EXTRACT-MIN**(open)

close.push(cur)

if cur.cost >= best_cost

continue

if cur为叶结点

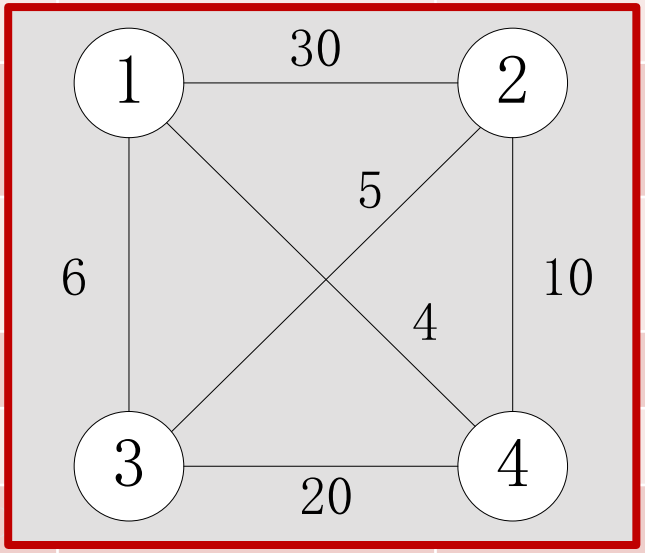
检测是否是更优解，若是，则修改当前最优解;

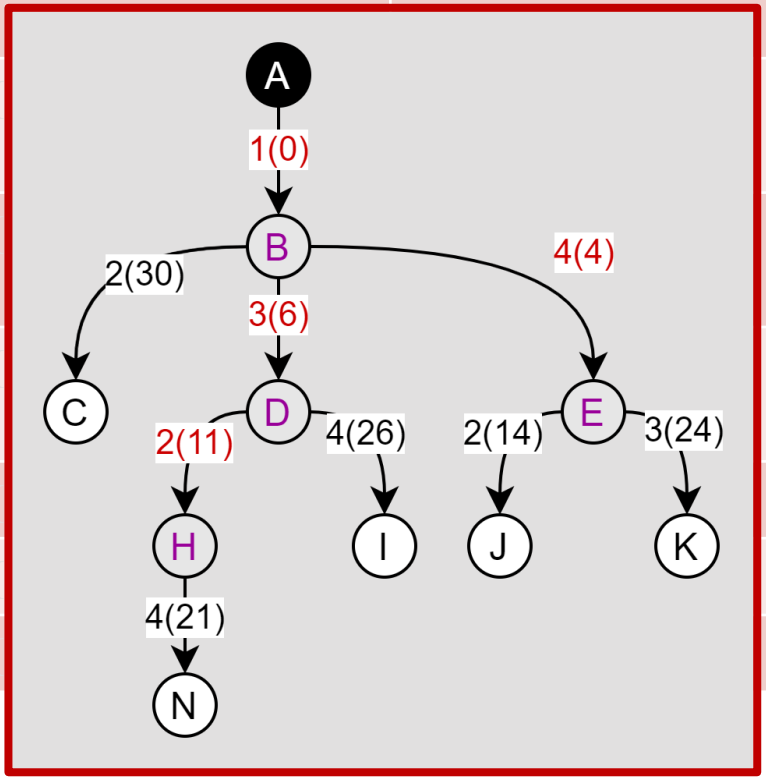
else

生成cur的所有满足条件的子结点;

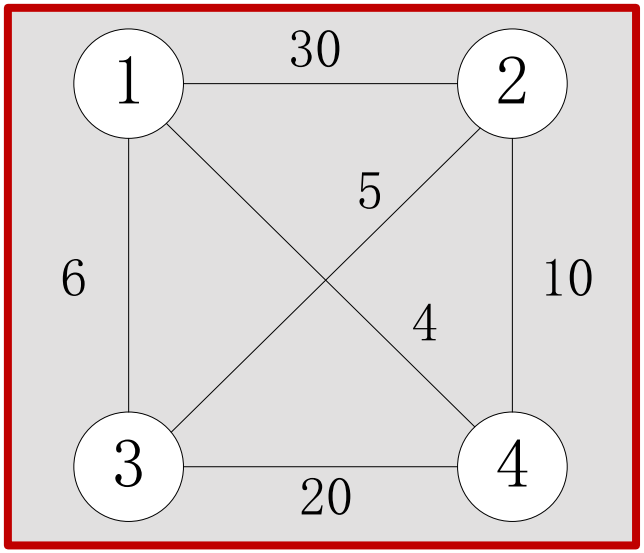
将子结点加入open表中，并根据其f(n)值维持最小堆性质

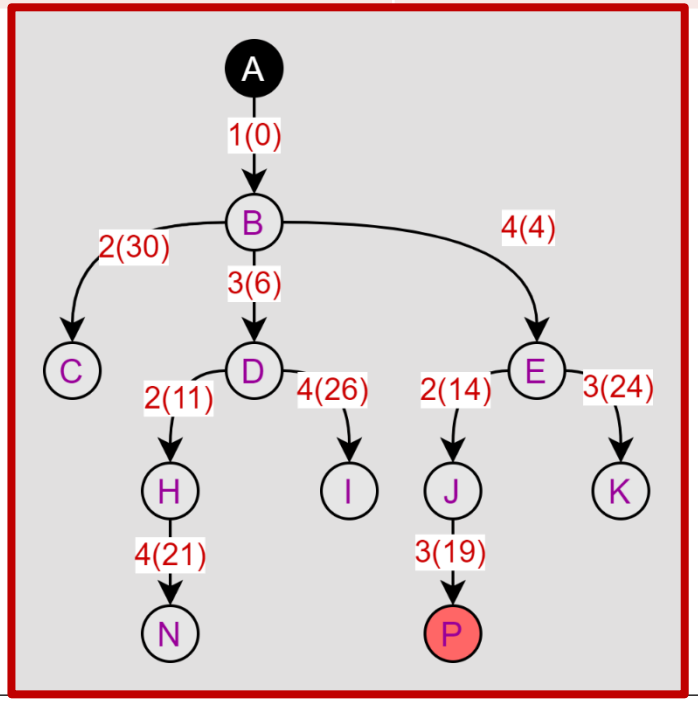
迭代	open表	当前结点	当前费用：当前路径	最优值：最优解
0	{**B(0)**}			∞ : []
1	{**E(4)**, D(6), C(30)}	B	0: [1]	∞ : []
2	{**D(6)**, J(14), K(24), C(30)}	E	4: [1, 4]	∞ : []
3	{**H(11)**, J(14), K(24), I(26), C(30)}	D	6: [1, 3]	∞ : []
4	{**J(14)**, N(21), K(24), I(26), C(30)}	H	11: [1, 3, 2]	∞: []
5	{P(19), N(21),	J	14: [	
6			19: [	
7			21: [	
8			24: [	
9			26: [	
10			30: [	





迭代	open表	当前结点	当前费用：当前路径	最优值：最优解
5	{**P(19)**, N(21), K(24), I(26), C(30)}	J	14: [1, 4, 2]	∞ : []
6	{**N(21)**, K(24), I(26), C(30)}	P	19: [1, 4, 2, 3]	25: [1, 4, 2, 3]
7	{**K(24)**, I(26), C(30)}	N	21: [1, 3, 2, 4]	25: [1, 4, 2, 3]
8	{**I(26)**, C(30)}	K	24: [1, 4, 3]	25: [1, 4, 2, 3]
9	{**C(30)**}	I	26: [1, 3, 4]	25: [1, 4, 2, 3]
10	{}	C	30: [1, 2]	25: [1, 4, 2, 3]





最大堆图例

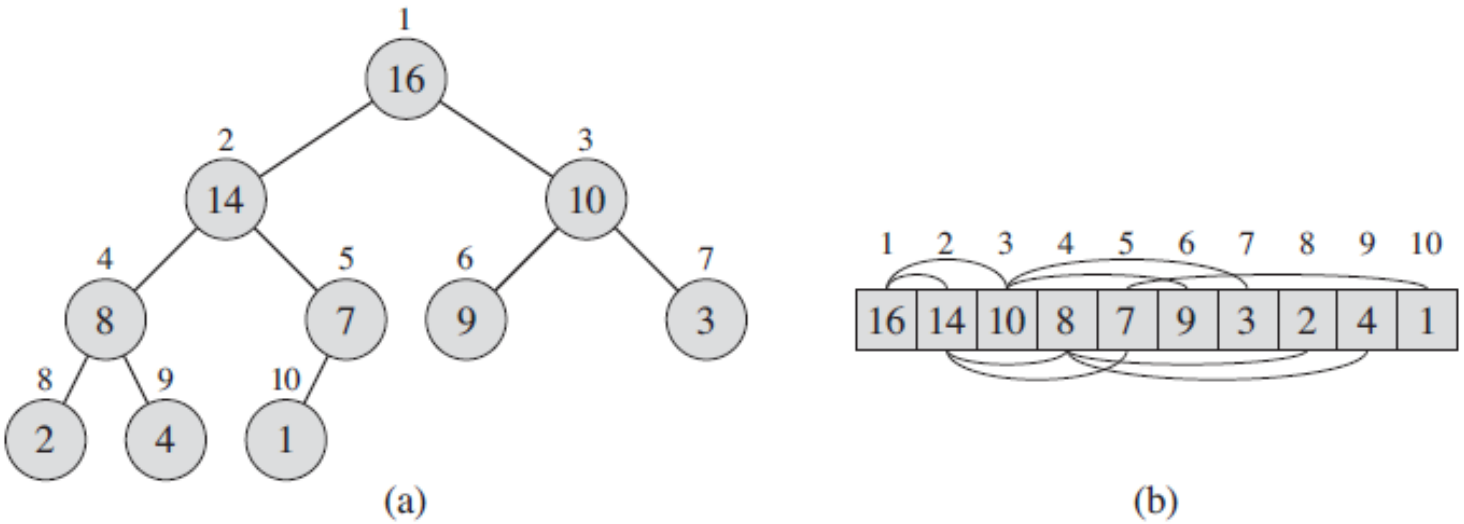


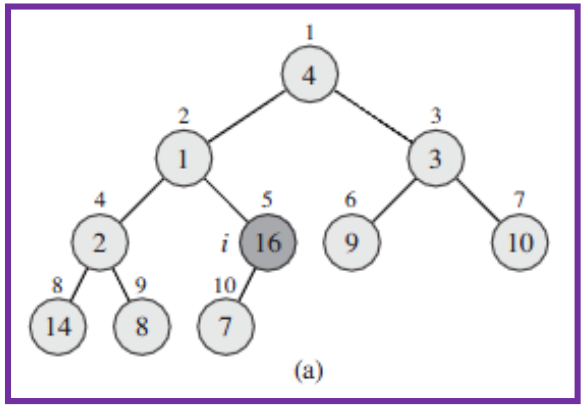
Figure 6.1 A max-heap viewed as (**a**) a binary tree and (**b**) an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

关于堆的一些基本操作(1)

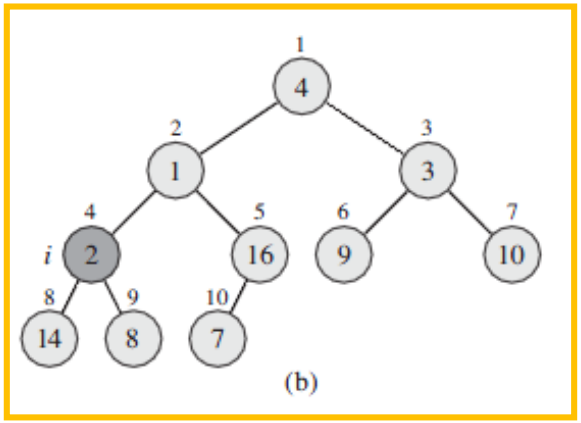
- **BUILD-MAX-HEAP (或BUILD-MIN-HEAP)**
  - 把一个无序数组构造成为一个最大堆（或最小堆）
  - 运算时间为 $O(n)$
- **EXTRACT-MAX (或EXTRACT-MIN)**
  - 去掉并返回堆中的具有最大（或最小）键字的元素，同时保持最大堆（或最小堆）的性质
  - 运算时间为 $O(\log n)$

BUILD-MAX-HEAP

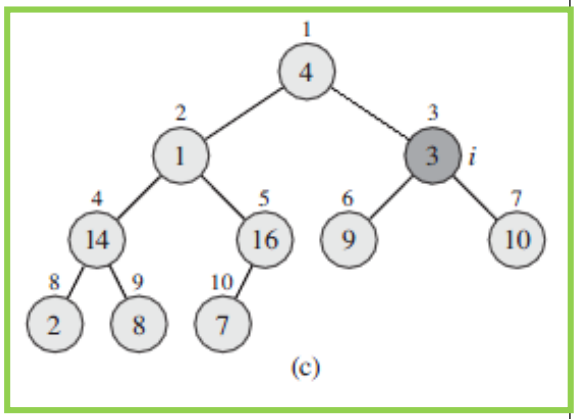
A	4	1	3	2	16	9	10	14	8	7



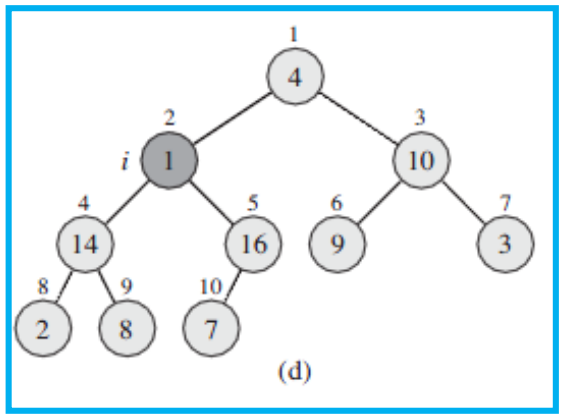
(a)



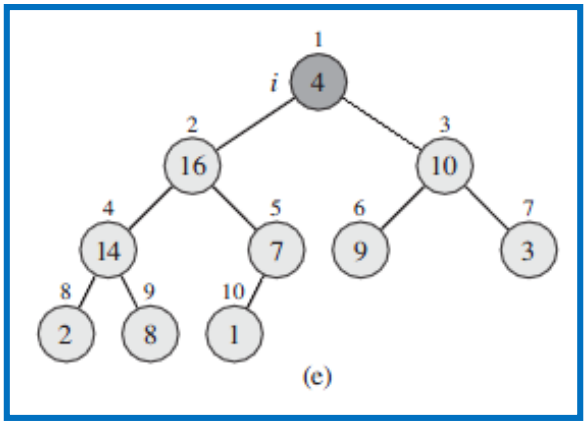
(b)



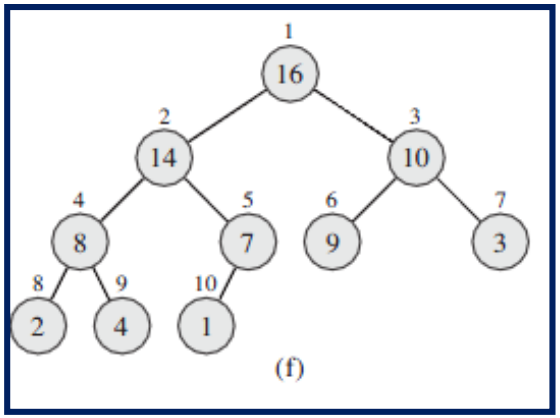
(c)



(d)



(e)



(f)

关于堆的一些基本操作(2)

- **MAX-HEAP-INSERT(或MIN-HEAP-INSERT)**
  - 在最大堆（或最小堆）中插入一个新元素，同时保持最大堆（或最小堆）的性质
  - 运算时间为 $O(\log n)$
  - **实现方法：**将新元素插入到堆的最后位置，调用：**HEAP-INCREASE-KEY**

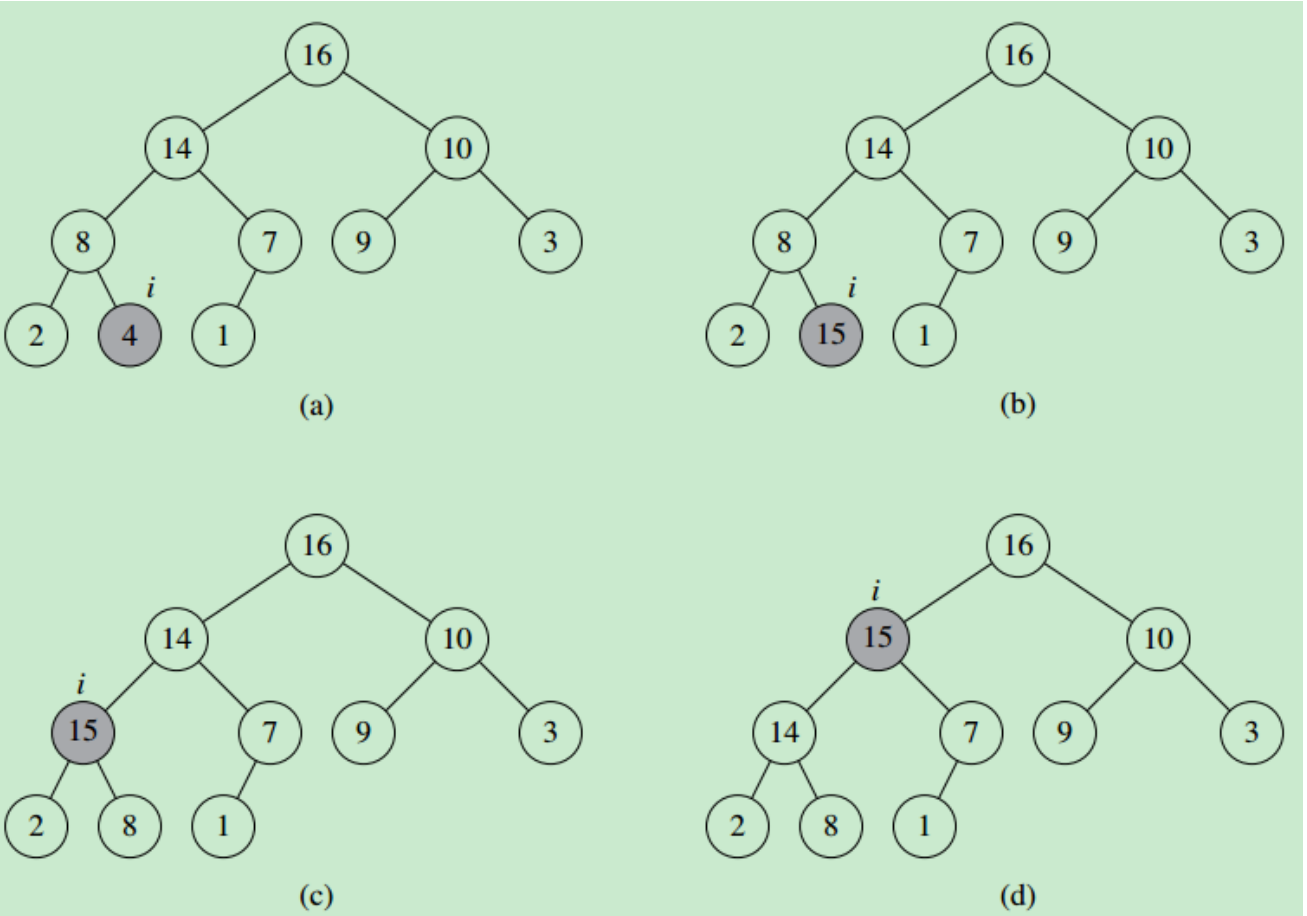


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

END