算法分析与设计

宋承云

scyer123@163.com
Chongqing University of Technology

2019年秋季学期

课程介绍

- 课程性质：专业核心课（理论讲授32学时，实验16学时）
- 课程要求：
  - 掌握常用的算法基本思想和经典问题求解过程
  - 分析算法的性能（时间和空间复杂度）
  - 设计算法解决计算机领域复杂问题
- 学习方法：
  - 掌握算法设计的核心思想（基础）
  - 注重编程实现（提高）
- 考核方式：
  - 平时成绩10%（作业，考勤，提问等）+实验成绩30%
  - 期末考试60%（闭卷）

教材及参考书籍

- 课程教材

- 其他教材

- 入门书籍

- 课外读物

课程资源

- 公开课
  - 算法设计与分析（北京大学–屈婉玲）

    https://www.coursera.org/learn/algorithms
  - 算法导论（麻省理工）

    http://open.163.com/special/opencourse/algorithms.html
  - 算法（普林斯顿大学）

    https://www.coursera.org/learn/algorithms-part1

    https://www.coursera.org/learn/algorithms-part2
- 专项课程
  - 程序设计与算法（北京大学）

    www.coursera.org/specializations/biancheng-suanfa
  - 算法（斯坦福大学）

    www.coursera.org/specializations/algorithms

编程资源

- 对于新手、进阶的信息安全工作者来说，刷题能够让算法能力得到一个质的飞跃。
- 在线题库
  - Hackerrank: https://www.hackerrank.com/
  - Topcoder: https://www.topcoder.com/
  - Geeksforgeeks: https://www.geeksforgeeks.org/
  - Codeforces: http://codeforces.com/
  - Lintcode: https://www.lintcode.com/
  - Leetcode: https://leetcode.com/
  - POJ-北大: http://poj.org/
  - HDU-杭电: acm.hdu.edu.cn/

课程内容

1.算法概述

- 理论（4）：算法的概念，几类重要问题，算法效率分析
- 实验（2）：算法效率对比分析，算法效率与输入规模分析

2.递归与分治策略

- 理论（8）：递归分析策略，分治算法思想，合并排序，快速排序，大整数乘法，矩阵乘法，最近对与凸包问题
- 实验（4）：实现快速和归并排序，设计寻找第k小元素算法

3.贪婪技术

- 理论（4）：贪心算法思想，Prim算法，Kruskal算法
- 实验（2）：实现0-1背包的贪心算法、最小生成树算法

课程内容

4.图算法

- 理论（6）：图算法基本思想，图搜索问题，拓扑排序，最短路径问题求解，回溯算法，分支界限
- 实验（4）：实现图的深度优先、广度优先搜索算法，以及图的最短路径算法

5.动态规划

- 理论（10）：动态规划思想，最长递增子序列，编辑距离问题，背包问题，矩阵链乘计算
- 实验（4）：实现0-1背包问题的动态规划求解、矩阵链乘算法，并与分治算法和贪心法比较，分析并设计数塔算法

Part I

算法概述

阅读: 1.1–1.4 2.1–2.3 3.1–3.4 4.4.1 4.4.3

主要内容

1. 什么是算法?
   - 趣味问题
   - 算法的概念
   - 问题求解基础
2. 重要问题类型
   - 排序和查找
   - 字符串处理
   - 图和组合问题
   - 几何问题
   - 数值问题
3. 基本数据结构
   - 列表
   - 图
   - 树
   - 集合与字典
4. 算法效率分析
   - 算法分析策略
   - 渐进符号
   - 算法效率分析
   - 基本效率类型

主要内容

问题1：农夫过河

问题描述

有一个农夫带一条狼、一只羊和一筐白菜过河。如果没有农夫看管，则狼要吃羊，羊要吃白菜。但是船很小，只够农夫带一样东西过河。农夫该如何解此难题？

问题1：农夫过河

问题描述

有一个农夫带一条狼、一只羊和一筐白菜过河。如果没有农夫看管，则狼要吃羊，羊要吃白菜。但是船很小，只够农夫带一样东西过河。农夫该如何解此难题？

解决步骤

1. 带羊到对岸，返回；
2. 带菜到对岸，把羊带回；
3. 带狼到对岸，返回；
4. 带羊到对岸。

问题2：鸡兔同笼

具体描述

一群小兔一群小鸡，两群合到一群中，腿一共有48，脑袋共有17个，问一共有多少小鸡？多少小兔？

解决步骤

1. 设未知数：设有x只小鸡，y只小兔
2. 列方程组：$\begin{cases} x+y=17 \\ 2x+4y=48 \end{cases}$
3. 解方程组：$\begin{cases} x=10 \\ y=7 \end{cases}$
4. 得到实际问题的答案：小鸡10只，小兔7只

线性方程组求解

如何求解一般的线性方程组？

$$\begin{cases} a_1x + b_1y = c_1 & (1) \\ a_2x + b_2y = c_2 & (2) \end{cases}, a_1b_2 - a_2b_1 \neq 0$$

求解步骤

1. $(1) * b_2 - (2) * b_1$得到：$(a_1 * b_2 - a_2 * b_1)x = c_1b_2 - c_2b_1 \quad (3)$
2. 解(3)得到：$x = (c_1b_2 - c_2b_1)/(a_1b_2 - a_2b_1)$
3. $(1) * a_2 - (2) * a_1$得到：$(a_2 * b_1 - a_1 * b_2)x = a_2c_1 - a_1c_1 \quad (4)$
4. 解(4)得到：$y = (a_2c_1 - a_1c_2)/(a_2b_1 - a_1b_2)$
5. 得到方程组的解：$\begin{cases} x = (c_1b_2 - c_2b_1)/(a_1b_2 - a_2b_1) \\ y = (a_2c_1 - a_1c_2)/(a_2b_1 - a_1b_2) \end{cases}$

问题3：最大公约数求解

最大公约数（greatest common divider）的定义

能够同时整除（即余数是0）两个不全为0非负整数的最大正整数。

最大公约数问题

求两个不全为0的非负整数（m和n）的最大公约数（记为$gcd(m,n)$）。

例子

$$gcd(5,0) = 5$$

$$gcd(15,5) = 5$$

$$gcd(60,24) = 12$$

方法一：欧几里得算法

算法要点

- $gcd(m,n) = gcd(n, m \ mod \ n)$，其中$m \ mod \ n$为$m$除$n$后的余数。
- $gcd(m,0) = m$

计算$gcd(m,n)$的欧几里得算法

1. 如果$n = 0$，返回m的值作为结果，过程结束；否则，进入第2步；
2. m除以n，得到余数r；
3. 将n的值赋给m，将r的值赋给n，返回第1步；

计算$gcd(60,24)$

$$\begin{aligned} gcd(60,24) &= gcd(24, 60 \ mod \ 24) = gcd(24,12) \\ &= gcd(12, 24 \ mod \ 12) = gcd(12,0) = 12 \end{aligned}$$

练习：利用欧几里得算法求$gcd(31415, 14142)$

答案

$$
\begin{aligned}
gcd(31415, 14142) &= gcd(14142, 31415 \ mod \ 14142) = gcd(14142, 3131) \\
&= gcd(3131, 14142 \ mod \ 3131) = gcd(3131, 1618) \\
&= gcd(1618, 3131 \ mod \ 1618) = gcd(1618, 1513) \\
&= gcd(1513, 1618 \ mod \ 1513) = gcd(1513, 105) \\
&= gcd(105, 1513 \ mod \ 105) = gcd(105, 43) \\
&= gcd(43, 105 \ mod \ 43) = gcd(43, 19) \\
&= gcd(19, 43 \ mod \ 19) = gcd(19, 5) \\
&= gcd(5, 19 \ mod \ 5) = gcd(5, 4) \\
&= gcd(4, 5 \ mod \ 4) = gcd(4, 1) \\
&= gcd(1, 4 \ mod \ 1) = gcd(1, 0) = 1
\end{aligned}
$$

欧几里得算法伪代码

算法： 欧几里得算法计算$gcd(m,n)$

输入： 两个不全为0的非负整数m,n

输出： m和n的最大公约数

```
1: function Euclid(m, n)
2:     while n ≠ 0 do
3:         r ← m mod n
4:         m ← n
5:         n ← r
6:     end while
7:     return m
8: end function
```

方法二：连续整数检测法

算法要点

基于最大公约数的定义，m和n的最大公约数是能够整除它们的最大正整数，因此可以从m和n中较小的数逐步减1寻找最大公约数。

计算$gcd(m,n)$的连续整数检测算法

1. 将$min(m,n)$的值赋给t；
2. m除以t，如果余数为0，进入第3步，否则进入第4步；
3. n除以t，如果余数为0，返回t的值，否则进入第4步；
4. 将t的值减1，返回第2步；

注意

当算法的一个输入是0的时候，连续整数检测算法得到的结果是错误的。因此，在算法的设计中必须认真、清晰地规定算法的输入的值域。

方法三：中学的计算方法

中学时计算$gcd(m,n)$的过程

1. 找到m的所有质因数；
2. 找到n的所有质因数；
3. 找出所有的公因数；
4. 将找到的公因数相乘，结果即为最大公因数；

注意

- 利用上述过程计算$gcd(31415,14142)$有什么问题？
- 在给出具体的质因数计算过程之前，上述方法不能称为一个真正意义上的算法。
- 计算连续质数序列：埃拉托色尼筛选法（课本$Page5$）

算法的定义

数学

算法通常是指按照一定规则解决某一类问题的明确和有限的步骤。

广义

算法是完成某项工作的方法和步骤。

- 菜谱是做菜的算法
- 歌谱是一首歌曲的算法

计算机科学

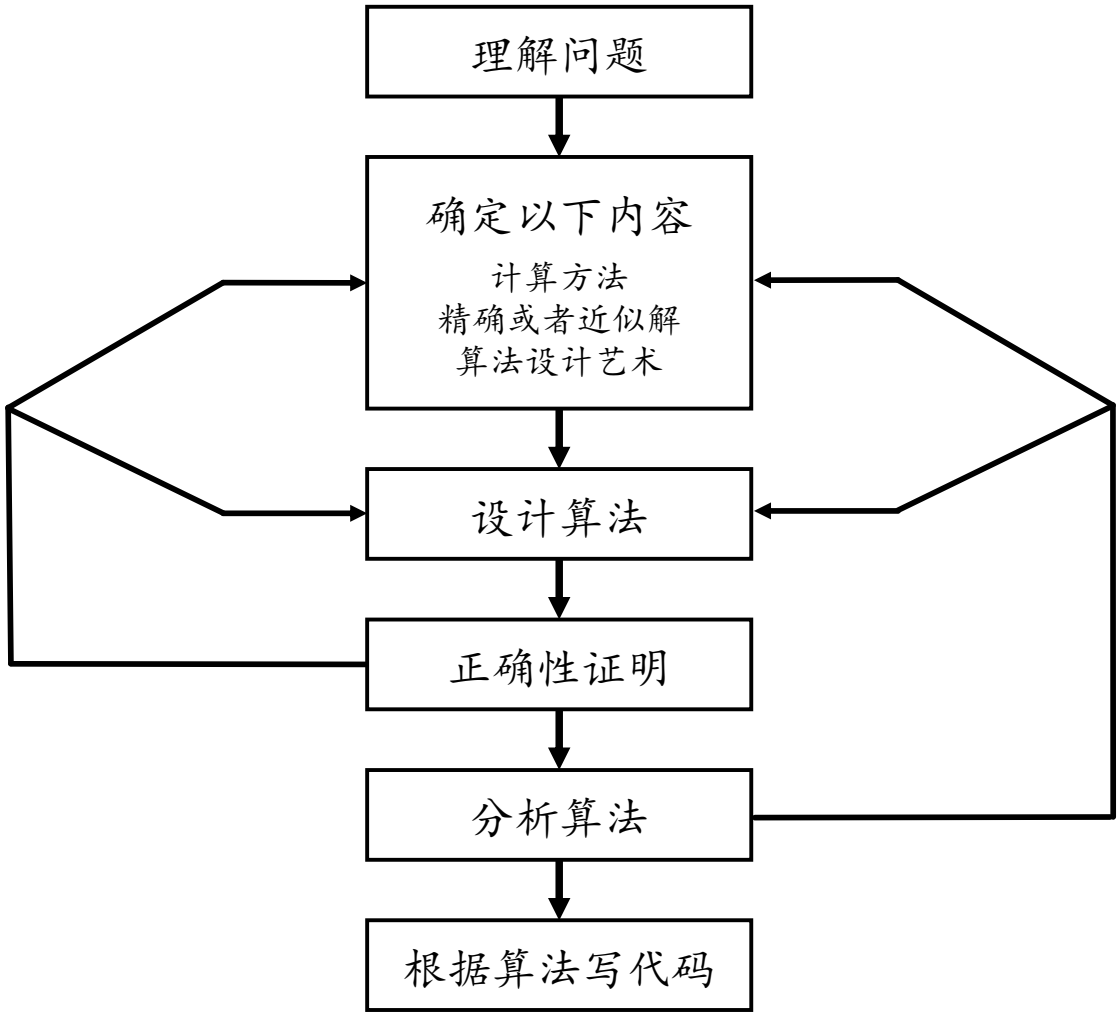
算法是一些列解决问题的明确指令，也就是说，对于符合一定规范的输入，能够在有限时间内获得要求的输出。

算法的特点

- **明确性与可行性**：算法中的每一个步骤都是确切的，且能有效地执行。
- **有限性**：算法必须在有限的步骤内完成。
- **一般性**：算法必须可以解决一类问题。
- **有序性**：算法从初始步骤开始，分为若干明确的步骤，每一步都只能有一个确定的继任者，只有执行完前一步才能进入到后一步，并且每一步都确定无误后，才能解决问题。
- **不唯一性**：求解某一个问题的解法不一定是唯一的，对于同一个问题可以有不同的解法。**但算法有优劣之分，好的算法是我们追求的目标。**

算法设计和分析过程

- 理解问题是解决问题的关键;
- 一个好的算法是不懈努力和反复修正的结果，这是一条规律;
- 发明算法是要一个非常有创造性和非常值得付出的过程;



主要内容

排序问题

问题定义

"排序"就是把一组杂乱的数据按照一定的规律排列起来。

为什么需要有序列表？

- 有序列表是所求解问题的输出，如学生成绩排序。
- 在很多其他领域的算法（如几何算法和数据压缩算法），排序被作为一个辅助步骤。

排序算法

- 选择排序，冒泡排序，插入排序，快速排序，二分归并排序，堆排序等。
- 虽然有些算法在平均效率和最差情况下的效率比其他算法都好，但是没有一种算法在任何输入下都是最优的。

选择排序算法

算法要点

扫描整个列表，找到它的最小元素，然后和第1个元素交换；然后从第2个元素开始扫描列表，找到后面$n-1$个元素的最小值，和第2个元素交换位置；如此重复。

例子

1	89	45	68	90	29	34	17
2	17	45	68	90	29	34	89
3	17	29	68	90	45	34	89
4	17	29	34	90	45	68	89
5	17	29	34	45	90	68	89
6	17	29	34	45	68	90	89
7	17	29	34	45	68	89	90

比较次数

- 第1次找最小需要$n-1$次比较
- 第2次找最小需要$n-2$次比较
- 依次类推.....
- 总次数:$n(n-1)/2$

选择排序伪代码

算法：选择排序对给定数组排序

输入： 一个杂乱的数据$A[0...n-1]$

输出： 升序排列的数据$A[0...n-1]$

```
1: function Selection_Sort(A[0...n-1])
2:     for i ← 0 → n-2 do
3:         min ← i        //min为最小元素索引
4:         for j ← i+1 → n-1 do
5:             if A[j] < A[min] then
6:                 min ← j
7:             end if
8:         end for
9:         swap A[i] and A[min]
10:    end for
11: end function
```

冒泡排序算法

算法要点

比较相邻元素，如果它们逆序就交换它们的位置，重复多次以后最大的元素就放置至最后的位置，如此重复放置次大的元素至倒数第2个位置，直到所有元素放好位置。

例子：放置最大元素

1	89	45	68	90	29	34	17
2	45	89	68	90	29	34	17
3	45	68	89	90	29	34	17
4	45	68	89	90	29	34	17
5	45	68	89	29	90	34	17
6	45	68	89	29	34	90	17
7	45	68	89	29	34	17	90

比较次数

- 找最大的元素需要$n-1$次比较
- 找次大的元素需要$n-2$次比较
- 依次类推.....
- 总次数:$n(n-1)/2$

冒泡排序伪代码

算法：冒泡排序对给定数组排序

输入： 一个杂乱的数据$A[0...n-1]$

输出： 升序排列的数据$A[0...n-1]$

```
1: function Bubble_Sort(A[0...n-1])
2:     for i ← 0 → n-2 do
3:         for j ← 0 → n-2-i do
4:             if A[j+1] < A[j] then
5:                 swap A[j] and A[j+1]
6:             end if
7:         end for
8:     end for
9: end function
```

查找

问题描述

从数据集合中找到满足某种条件的数据，返回两个结果：找不到返回空值，或找到返回搜索对象的位置。

查找算法

顺序查找，折半查找，二叉树查找等。

例子

从有序序列$3, 14, 27, 31, 39, 42, 55, 70, 74, 81, 85, 93, 98$中寻找元素$70$所在的位置。

简单的方法–顺序查找

- 序列中的元素依次和70比较，需比较8次。
- 对于n个元素的序列，最坏情况下需要比较n次。

顺序查找伪代码

算法：顺序查找

```
1: function Seq_Search(A[0...n − 1],K)
2:     i ← 0
3:     while A[i] ≠ K do
4:         i ← i + 1
5:     end while
6:     if i < n then
7:         return i        //首个值为K的元素的位置
8:     else
9:         return −1       //找不到返回−1
10:    end if
11: end function
```

折半查找

算法要点

对于有序的序列A，待查找元素K与序列最中间的元素$A[m]$比较，如果相等，算法结束；否则，如果$K < A[m]$，在序列的前半部分查找，如果$K > A[m]$，在序列的后半部分查找；如此重复。

例子：寻找$K = 70$的元素

1	3	14	27	31	39	42	55	70	74	81	85	93	98
2	3	14	27	31	39	42	55	70	74	81	85	93	98
3	3	14	27	31	39	42	55	70	74	81	85	93	98

比较次数

上述例子寻找55需要4次比较；对于n个元素的序列，最坏情况下的比较次数为$\lfloor log_2 n \rfloor + 1$。

折半查找算法伪代码

算法：非递归的实现折半查找

```
1:  function Binary_Search(A[0...n − 1], K)
2:      l ← 0,    r ← n − 1
3:      while l ⩽ r do
4:          m ← ⌊(l + r)/2⌋
5:          if K = A[m] then
6:              return m                    //值为K的元素的位置
7:          else if K < A[m] then
8:              r ← m − 1
9:          else
10:             l ← m + 1
11:         end if
12:     end while
13:     return −1                   //找不到返回-1
14: end function
```

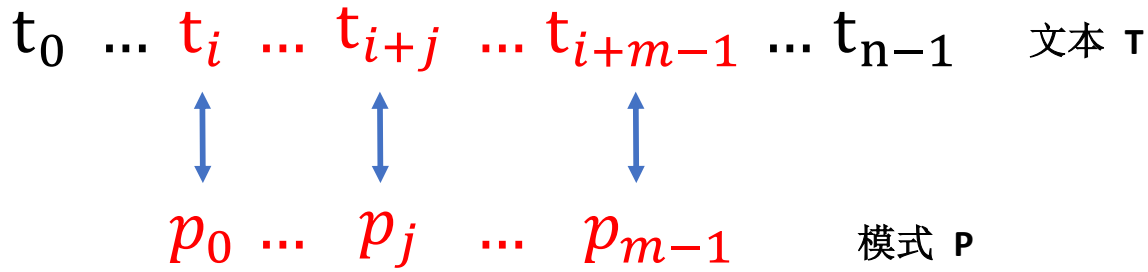
字符串处理

字符串匹配

字符串匹配问题是一种特殊的字符串处理问题，其目的是在一个大的字符串T中搜索某个字符串P的所有出现位置，其中T称为文本，P称为模式。

数学定义

给定一个n个字符组成的文本和$m(m < n)$个字符的模式，寻找i(文本中第一个匹配字符串最左边元素的下标)，使得$t_i = p_0$，...，$t_{i+j} = p_j$，$t_{i+m-1} = p_{m-1}$。

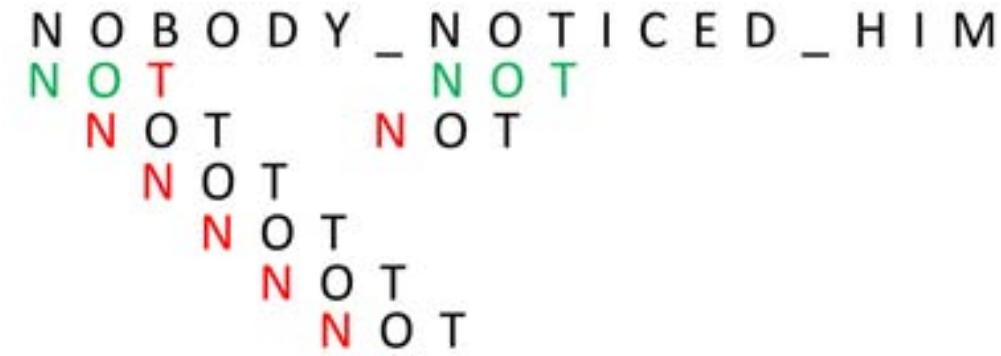


蛮力匹配

算法要点

将模式P对准文本的前m个字符，然后从左到右匹配每一个相应的字符，直到m个字符全部匹配。

例子



比较次数

最坏情况下，移动模式之前会做m次比较，$n-m+1$次移动都可能遇到这种情况，比较次数为$m(n-m+1)$。

蛮力字符串匹配伪代码

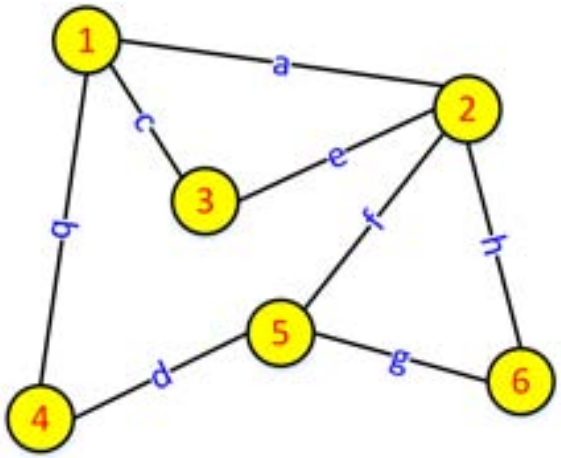
算法：蛮力匹配

```
1: function Brute_Force_String_Match(T[0...n − 1],P[0...m − 1])
2:     for i ← 0 → n − m do
3:         j ← 0
4:         while i < m and P[j] = T[i + j] do
5:             j ← j + 1
6:             if j = m then
7:                 return i //第1个匹配字符子串中第1个字符位置
8:             end if
9:         end while
10:    end for
11:    return −1        //找不到返回−1
12: end function
```

图问题

图的定义

图（Graph）是有一些称为顶点的点构成的集合，其中某些顶点由一些称为边的线段相连。



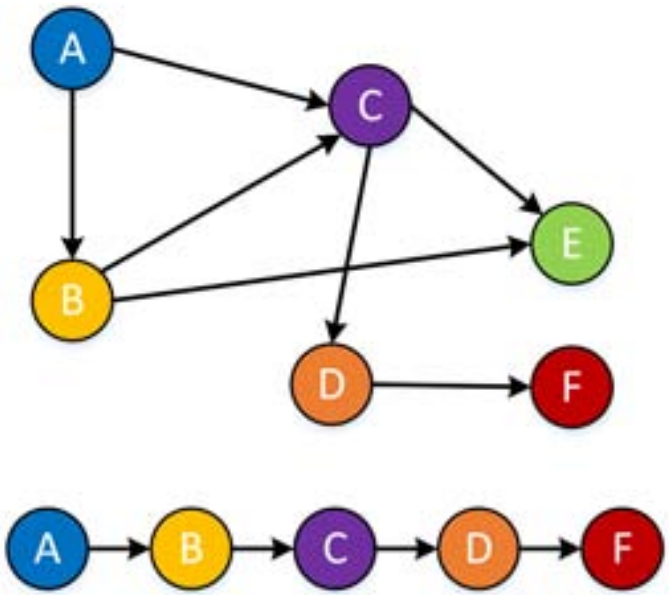
最短线路问题

两个地点之间的最佳线路是哪条？

其他图问题

拓扑排序

在**有向无环图**中，寻找满足拓扑次序的序列。



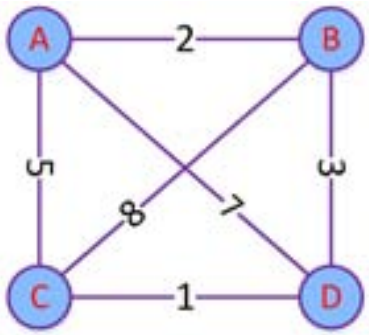
图着色

对于**无向图**，用最小数目的颜色标记各个顶点，使得任何相邻的顶点颜色不重复。

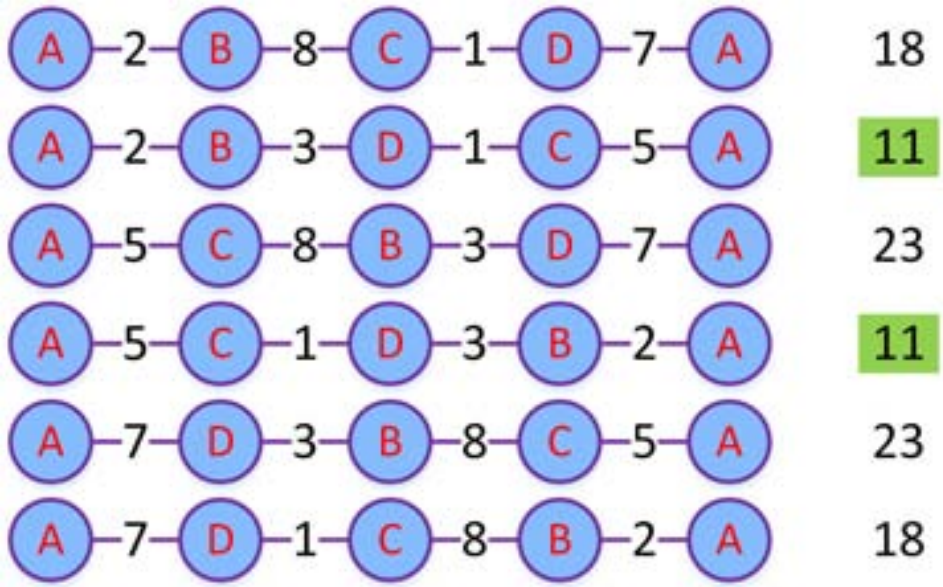
图问题：旅行商问题（TSP）

问题描述

找出一条访问每个城市的最短路径，使得回到出发城市之前，每个城市只访问一次



穷举查找求解



计算次数

需要穷举出所有路线，总排列数为$(n-1)!$

组合问题

问题描述

寻找一些组合对象（例如一个排列，一个组合或者一个子集），这些对象能够满足特定的条件并具有我们想要的特性，如价值最大化或者成本最小化。图问题中的最短路径、旅行商和图着色问题都是组合问题。

组合问题是计算机领域中最难的问题

- 随着问题规模的增大，组合对象的数量增长极快。
- 还没有一种已知的算法能在可接受的时间内，精确地解决绝大多数这类问题。
- 有些组合问题有高效的算法求解（如最短路径算法），但是那仅仅是一些幸运的例外。

背包问题

问题描述

给定n个重量为$w_1, w_2, ..., w_n$，价值为$v_1, v_2, ..., v_n$的物品，和一个承重量为W的背包，求能够装到背包中最有价值的物品子集。

实例：穷举查找求解

子集	重量	价值	子集	重量	价值
ø	0	0	A	7	42
B	3	12	C	4	40
D	5	25	AB	10	54
AC	11	Not	AD	12	Not
BC	7	52	BD	8	37
CD	9	65	ABC	14	Not
ABD	15	Not	ACD	16	Not
BCD	12	Not	ABCD	19	Not

所有子集总数：2^n

分配问题

问题描述

有n个任务需要分配给n个人执行，一个任务只能分配给一个人，一个人只能分配一个任务。将第j个任务分配给第i个人的成本是$C[i,j]$，求成本最小分配方案。

例子

人员＼任务	1	2	3	4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

穷举查找求解

方案	成本
$<1,2,3,4>$	9+4+1+4=18
$<1,2,4,3>$	9+4+8+9=30
$<1,3,2,4>$	9+3+8+4=24
$<1,3,4,2>$	9+3+8+6=26
$<1,4,2,3>$	9+7+8+9=33

……

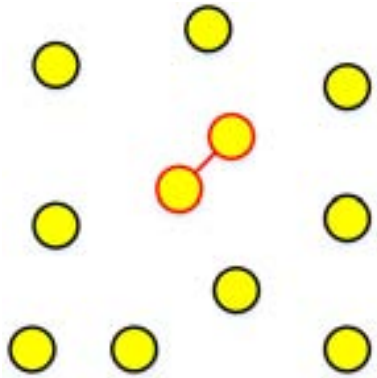
计算次数：穷举的排列总数：$(n)!$

最近对问题

问题描述

- 在一个包含n个点的集合中，找出距离最近的两个点。
- 航空交通控制人员监测最有可能发生碰撞的飞机。
- 区域邮政管理人员寻找地理位置最近的邮局。

示例



蛮力查找求解最近对问题

算法：蛮力查找

```
1: function Brute_Force_Closest_Points(P)
2:     d ← +∞
3:     for i ← 1 → n − 1 do
4:         for j ← i + 1 → n do
5:             d ← min(d, dist(p_i, p_j)) //dist(p,q)计算点p和q的距离
6:         end for
7:     end for
8:     return −1
9: end function
```

计算次数

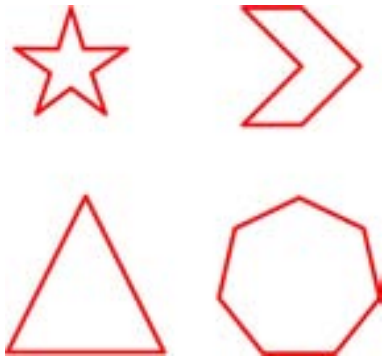
距离计算的总次数为$C_n^2 = n(n-1)/2$。

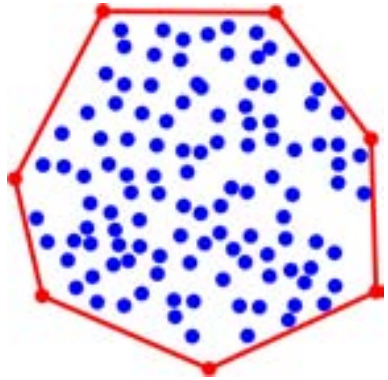
凸包问题

问题描述

- 凸集合：对于平面上的一个点的集合，如果任意两个点的连线都在集合中，就称为凸集合。
- 凸包问题：寻找包含所有点的最小凸集合。

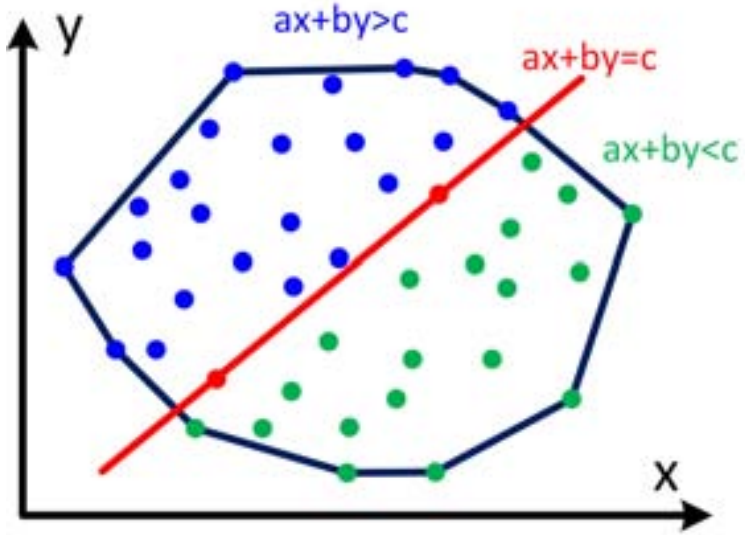
示例





蛮力法求解凸包问题

- 对于n个点集合中的两个点p_i和p_j，当且仅当该集合中的其他点都位于穿过这两个点的直线的同一边时，p_i和p_j的连线是该集合凸包边界的一部分。
- 一共有C_n^2条线，对于每一条线，需要检查其他$n-2$个点的符号，比较次数为$n(n-1)(n-2)/2$;



数值问题

典型的问题

解方程及方程组，计算定积分，求函数的值，矩阵乘法计算，矩阵链乘问题，最小公倍数，最大公约数，正整数相乘等。

问题难点

- 对于数值计算中的大多数问题，都只能近似求解；
- 数值问题要操作实数，而实数在计算机内部只能近似表示；
- 对近似数的大量算术操作可能会将大量的舍入误差叠加起来，导致一个看似可靠的算法输出严重歪曲的结果。

整数相乘：俄式乘法计算$n * m$

算法步骤

1. 如果n是偶数，$n = n/2, m = m * 2$（$n * m = n/2 * 2m$）；
2. 如果n是奇数，$n = (n-1)/2, m = m * 2$，并保存上一步m的值（$n * m = (n-1)/2 * 2m + m$）；
3. 重复步骤1和2，直到$n = 1$，计算总和。

实例

n	m	
50	65	
25	130	
12	260	(+130)
6	520	
3	1040	
1	2080	(+1040)
	2080	+(130+1040)=3250

特点

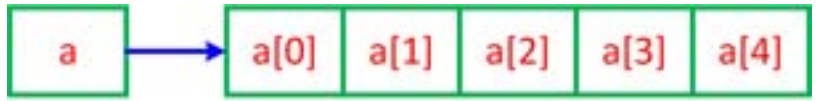
- 算法仅涉及折半，加倍和加法操作，不需乘法口诀；
- 算法计算快，计算机使用移位操作就可以实现二进制数的折半和加倍。

主要内容

列表

数组

连续存储，访问快。



链表

便于插入和删除，内存占用小。

双向链表

便于寻找父节点。

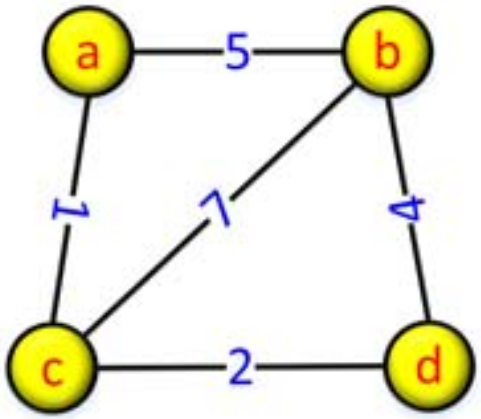
栈和队列

- 栈：先进后出，添加元素称为进栈，删除元素称为出栈；
- 队列：先进先出，添加元素称为入队，删除元素称为出队；

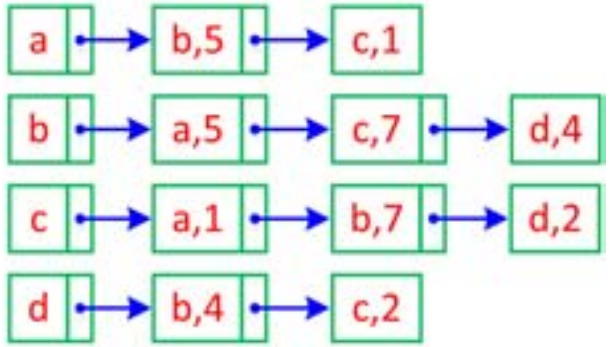
图（Graph）

图的表示

- 图$G =< V, E >$有两个集合来定义，V表示顶点，E表示边；
- 主要的图类型包括：无向图，有向图，加权图（边具有权值）；
- 图常常采用邻接矩阵或邻接链表表示；



	a	b	c	d
a	Inf	5	1	Inf
b	5	Inf	7	4
c	1	7	Inf	2
d	Inf	4	2	Inf



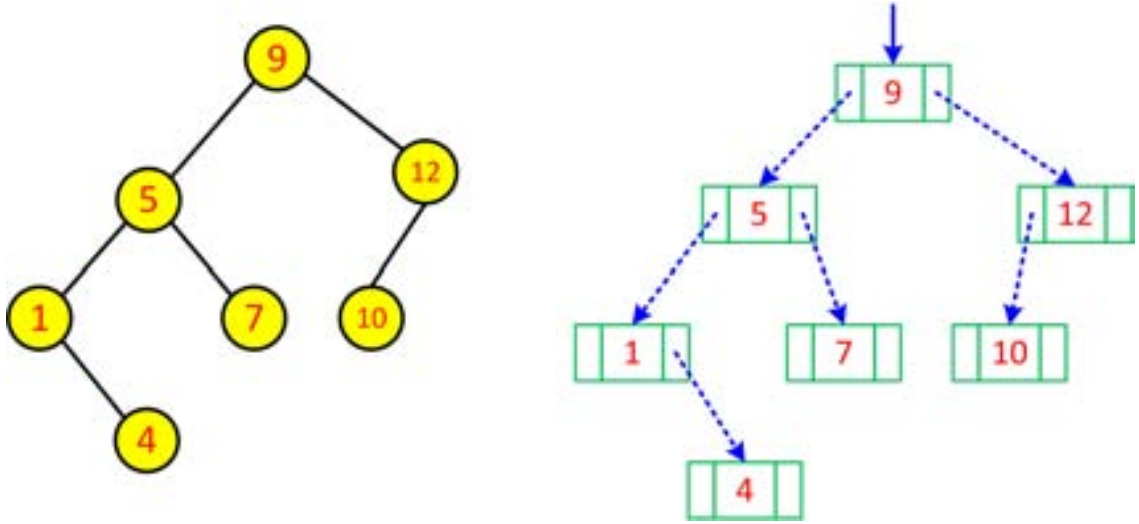
树

二叉树

二叉树的每个顶点的子女节点不超过两个，并且是有序的（分别为左子女和右子女节点）。

二叉查找树

每个父母比左子树的大，比右子树小。



集合与字典

集合

集合是互不相同元素构成的无序集合，可以用位向量或者线性列表实现。最重要的集合操作有：检查元素是不是属于集合，集合求并集和交集。

字典

字典是以(key,value)的形式储存的数据结构，可方便地进行元素查找，及求并集和交集的操作。可以用数组，散列法和平衡查找树实现。

抽象数据类型

抽象数据类型是由一个表示数据项的抽象对象集合和一系列对这些对象的操作构成。在面向对象的编程语言（C++，Java）中，可用类实现。

主要内容

算法效率

算法的效率分析包括两个方面

- 时间效率：讨论算法运行的有多快；
- 空间效率：关心算法占用的存储空间；

空间重要性降低

- 对于早期的计算机，时间和空间两种资源都是非常昂贵的。
- 随着电脑工艺的进步，计算机的存储容量已经提升了好几个数量级，降低了空间效率对算法性能的影响。
- 算法的时间效率没有得到相同程度的提高。

时间重要性

算法的时间效率分析是算法课程的主要部分。

度量因子

输入规模

- 几乎所有的算法，运行时间都随着输入规模的增大而增大；
- 算法效率是输入规模的函数，应当恰当地选择输入规模的度量；

运行时间

- 以秒等单位度量的程序运行时间依赖于计算机的性能；
- 统计每步操作（赋值及计算）的执行次数不仅困难而且没有必要；
- 基本操作（加减乘除，除法最耗时，其次是乘法）次数对算法的总运行时间贡献最大；
- 算法运行时间$T(n) \approx c_{op} * C(n)$，其中$c_{op}$为特定计算机上基本操作执行时间，$C(n)$为算法基本操作次数。

增长次数–算法效率度量

思考

- 考虑两个算法，计算复杂度分别为$T_1(n) = n$，$T_2(n) = n!$，当n等于1和2的时候，二者的运行时间相同，它们的计算效率是否相同？
- 当输入规模变大，$T_1(10) = 10$，$T_2(10) \approx 3.6 * 10^6$，是不是意味着算法1比算法2快36万倍？

结论

- 小规模的输入不足以将高效的算法和低效的算法区分出来，必须考虑较大规模的输入；
- 不仅需要知道算法需要多少时间运行完毕，还需要知道运行时间随着输入规模的增长而增加的幅度；

重要函数特性

随输入规模的增长情况（近似）

n	$log_2 n$	n	$nlog_2 n$	n^2	n^3	2^n	$n!$
10	3.3	10	$3.3 * 10$	10^2	10^3	10^3	$3.6 * 10^6$
10^2	6.6	10^2	$6.6 * 10^2$	10^4	10^6	$1.3 * 10^{30}$	$9.3 * 10^{157}$
10^3	10	10^3	$1.0 * 10^4$	10^6	10^9		
10^4	13	10^4	$1.3 * 10^5$	10^8	10^{12}		
10^5	17	10^5	$1.7 * 10^6$	10^{10}	10^{15}		
10^6	20	10^6	$2.0 * 10^7$	10^{12}	10^{18}		

注：当n增长为$2n$时，增长分别是：$log_2 n$为1，n为2倍，$nlog_2 n$为2倍多，n^2为4倍...

注意

对数函数的操作次数依赖于对数的低，由于$log_a n = log_a b * log_b n$，因此底不相同的两个对数函数只差一个乘法常量，当我们关心增长次数的时候，忽略对数的底，简单写成$logn$。

算法的三种效率

定义

- 最差效率$C_{worst}(n)$：输入规模为n时，算法在最坏输入下的效率；
- 最优效率$C_{best}(n)$：输入规模为n时，算法在最好输入下的效率；
- 平均效率$C_{avg}(n)$：输入规模为n时，算法在随机输入下的效率；

顺序查找算法从序列$A[1,2,...,n]$中寻找元素10比较次数

- $C_{best}(n)=1$，如$A=[10,2,5,6,7,8,9]$；
- $C_{worst}(n)=n$，如$A=[5,2,5,6,7,8,10]$；
- $C_{avg}(n)=p*[1*\frac{1}{n}+2*\frac{1}{n}+...+n*\frac{1}{n}]+(1-p)*n$

$$=\frac{p(n+1)}{2}+n(1-p)$$

注：p为查找成功的概率，$p=1$表示查找元素一定在序列中，$p=0$表示会查找不成功。

增长次数描述：渐进符号

上界O的定义

如果存在大于0的常数c和非负整数n_0，使得$\forall n > n_0$，$t(n) \leqslant cg(n)$，我们称函数$t(n)$包含在$O(g(n))$中，记作$t(n) \in O(g(n))$。

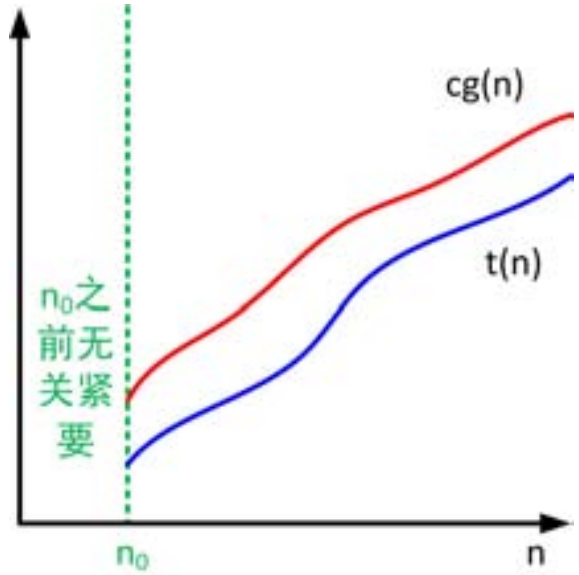
例子

$$n \in O(n), n \in O(n^2), n \in O(n^3)$$

$$100n + 5 \in O(n^2)$$

$$0.5n(n+1) \in O(n^2)$$

$$n^2 \notin O(n), n^3 \notin O(n^2)$$



增长次数描述：渐进符号

下界Ω的定义

如果存在大于0的常数c和非负整数n_0，使得$\forall n > n_0$，$t(n) \geqslant cg(n)$，我们称函数$t(n)$包含在$\Omega(g(n))$中，记作$t(n) \in \Omega(g(n))$。

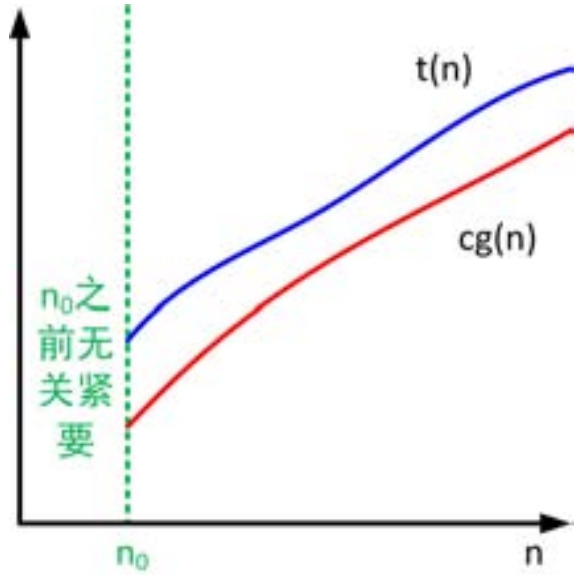
例子

$$n^2 \in \Omega(n)$$

$$0.5n(n-1) \in \Omega(n^2)$$

$$n^3 \in \Omega(n), n^3 \in \Omega(n^2)$$

$$100n + 5 \notin \Omega(n^2)$$



增长次数描述：渐进符号

Θ的定义

若存在大于0的常数c_1,c_2和非负整数n_0,$\forall n > n_0$,$c_1 g(n) \leqslant t(n) \leqslant c_2 g(n)$,
我们称函数$t(n)$包含在$\Theta(g(n))$中，记作$t(n) \in \Theta(g(n))$。

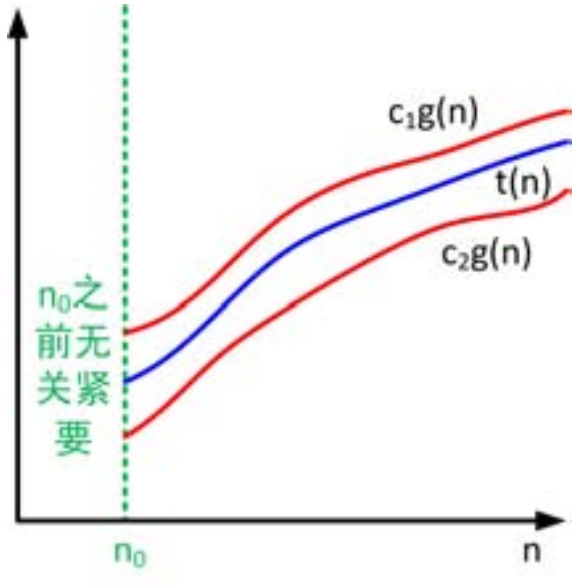
例子

$$100n^2 \in \Theta(n^2)$$

$$0.5n(n-1) \in \Theta(n^2)$$

$$0.5n(n-1) \notin \Theta(n^3)$$

$$0.5n(n-1) \notin \Theta(n)$$



渐进符号特性

定理

如果$t_1(n) \in O(g_1(n))$，并且$t_2(n) \in O(g_2(n))$，则

$$t_1(n) + t_2(n) \in O(max\{g_1(n), g_2(n)\}).$$

对于Ω和Θ符号，类似的断言也成立。

例子：检查数组中是否含有相同元素

- 算法首先对数组进行排序，然后比较相邻元素是否相等；
- 排序算法采用选择排序，比较次数为$C_n^2 \in O(n^2)$;
- 比较相邻元素步骤的比较次数为$n-1 \in O(n)$;
- 算法的整体效率为$O(max\{n^2, n\}) \in O(n^2)$
- 结论：算法的整体效率由较大增长次数（效率较差）的部分决定。

利用极限比较增长次数

定理

设f和g是定义域为自然数集合N上的非负函数，有：

$$\lim_{n\to\infty}\frac{t(n)}{g(n)}=\begin{cases}0 & : t(n)\in O(g(n))\\ c>0 & : t(n)\in O(g(n));t(n)\in\Omega(g(n));t(n)\in\Theta(g(n))\\ \infty & : t(n)\in\Omega(g(n))\end{cases}$$

例子

$$\lim_{n\to\infty}\frac{log_2n}{\sqrt{n}}=\lim_{n\to\infty}\frac{log_2n'}{\sqrt{n}'}=\lim_{n\to\infty}\frac{(log_2e)\frac{1}{n}}{\frac{1}{2\sqrt{n}}}=2log_2e\lim_{n\to\infty}\frac{1}{\sqrt{n}}=0$$

$$log_2n\in O(\sqrt{n})$$

选择排序算法效率分析

算法回顾

```
1: function Selection_Sort(A[0...n − 1])
2:     for i ← 0 → n − 2 do
3:         min ← i //min为最小元素索引
4:         for j ← i + 1 → n − 1 do
5:             if A[j] < A[min] then
6:                 min ← j
7:             end if
8:         end for
9:         swap A[i] and A[min]
10:    end for
11: end function
```

算法分析

- 算法涉及赋值和比较运算；
- 比较运算比赋值开销大，所以比较为基本操作；
- $C_{best}(n) = C_n^2$
  $C_{worst}(n) = C_n^2$
  $C_{avg}(n) = C_n^2$
  $\frac{1}{2}n(n-1) \in \Theta(n^2)$

其他算法效率分析

冒泡排序算法（比较次数）

$$C_{best}(n) = C_{worst}(n) = C_{avg}(n) = C_n^2 \in \Theta(n^2)$$

顺序查找算法（比较次数）

$$C_{best}(n) = 1,\ C_{worst}(n) = n \in \Theta(n),\ C_{avg}(n) = n/2 \in \Theta(n)$$

折半查找算法（比较次数）

$$C_{best}(n) = 1,\ C_{worst}(n) = \lfloor log_2 n \rfloor + 1 \in \Theta(logn),\ C_{avg}(n) = ?\text{（练习）}$$

蛮力字符串匹配算法（比较次数）

$$C_{best}(n) = m,\ C_{worst}(n) = m(n-m+1) \in \Theta(n),\ C_{avg}(n) = ?\text{（练习）}$$

蛮力求解最近对算法（距离中的平方计算）

$$C_{best}(n) = C_{worst}(n) = C_{avg}(n) = 2 * C_n^2 \in \Theta(n^2)\text{（二维）}$$

基本效率类型

类型	名称	注释
1	常数	为数很少的效率最高的算法，难以举例。
$logn$	对数	算法的每一次循环都会消去问题规模的一个常数因子，如有序序列中的折半查找算法。
n	线性	扫描规模为n的列表，如顺序查找算法。
$nlogn$	线性对数	许多分治算法（合并排序，快速排序）的平均效率属于这个类型。
n^2	平方	一般来说，包含两重嵌套循环算法的典型效率，选择排序和冒泡排序属于这一类型。
n^3	立方	一般来说，包含三重嵌套循环算法的典型效率。
2^n	指数	求n个元素集合的所有子集的算法，如蛮力查找求解背包问题和最近对问题。
$n!$	阶乘	求n个元素的完全排列算法，如穷举查找求解分配问题，旅行商问题。

小结

- 算法是一些列解决问题的明确指令，对于符合一定规范的输入，能够在有限时间内获得要求的输出；
- 理解问题是利用算法解决问题的首要条件，算法设计是一个不断重复的过程；
- 常见的重要问题类型有：排序，查找，字符串处理，图问题，组合问题，几何问题和数值问题；
- 算法分析主要包括时间复杂度和空间复杂度，而时间复杂度是算法课程的核心内容；
- 算法运行时间的度量单位为最影响算法性能的基本操作；
- 增长次数是衡量算法好坏的最关键的指标；
- 算法的效率包括最优效率，最差效率和平均效率；

作业

- 1.1: 4 $Page6$
- 1.2: 5 $Page14$
- 1.4: 7 $Page30$
- 2.2: 5 $Page47$
- 2.3: 5 $Page53$

Part II

递归与分治策略

阅读: 2.4 附录B 5.1–5.2 5.4–5.5 4.5.1

主要内容

1. 递归分析方法
   - 递归算法数学分析
   - 递归分析实例
2. 分治算法
   - 分治策略
   - 合并排序
   - 快速排序
   - 选择问题
   - 大整数乘法和矩阵乘法
   - 最近对问题和凸包问题

主要内容

递归的概念

引例

- 对于任意非负整数n，计算阶乘函数$F(n) = n!$的值；
- 当$n > 1$时，$n! = 1 * 2 * ... * (n-1) * n = (n-1)! * n$，并且$n! = 1$；
- 可以使用递归的方法计算$F(n) = F(n-1) * n$。

伪代码

```
1: function F(n)
2:     if n = 0 then
3:         return 1
4:     else
5:         return F(n − 1) ∗ n
6:     end if
7: end function
```

乘法执行次数$M(n)$

- $n > 0$时，$M(n) = M(n-1) + 1$
- 计算$F(n-1)$需要$M(n-1)$次乘法
- 计算$F(n-1) * n$需要1次乘法
- $n = 0$时，$M(0) = 0$，不需要乘法
- 替换法：$M(n) = M(n-i) + i = n$

递归方程求解–替换法

例1：计算$W(n)$，$W(n) = W(n-1) + n - 1, W(1) = 0$

$$
\begin{aligned}
W(n) &= W(n-1) + n - 1 = W(n-2) + (n-2) + (n-1) \\
&= W(n-3) + (n-3) + (n-2) + (n-1) = ... \\
&= W(1) + 1 + 2 + ... + (n-2) + (n-1) = 1 + 2 + ... + (n-2) + (n-1) \\
&= n(n-1)/2
\end{aligned}
$$

例2：计算$W(n)$，$W(n) = 2W(n/2) + n - 1, n = 2^k, W(1) = 0$

$$
\begin{aligned}
W(n) &= 2W(2^{k-1}) + 2^k - 1 = 2[2W(2^{k-2}) + 2^{k-1} - 1] + 2^k - 1 \\
&= 2^2 W(2^{k-2}) + 2^k - 2 + 2^k - 1 \\
&= 2^2[2W(2^{k-3}) + 2^{k-2} - 1] + 2^k - 2 + 2^k - 1 \\
&= 2^3 W(2^{k-3}) + 2^k - 2^2 + 2^k - 2 + 2^k - 1 = ... \\
&= 2^k W(1) + k2^k - (2^{k-1} + 2^{k-2} + ... + 2 + 1) = k2^k - 2^k + 1 \\
&= nlogn - n + 1
\end{aligned}
$$

递归方程求解–主定理

主定理

设$T(n)$是一个非递减函数（定义见课本P376），并且满足递推式

$$T(n) = aT(n/b) + f(n)，\ 其中n = b^k，\ k = 1, 2, ...$$

$$T(1) = c$$

其中$a \geqslant 1$，$b \geqslant 2$，$c > 0$，如果$f(n) \in \Theta(n^d)$，$d \geqslant 0$，那么

- 当$a < b^d$时，$T(n) \in \Theta(n^d)$；
- 当$a = b^d$时，$T(n) \in \Theta(n^d logn)$；　　**结论对符号$O$和$\Omega$也成立。**
- 当$a > b^d$时，$T(n) \in \Theta(n^{log_b a})$；

注释

- $T(n) = aT(n/b) = a^2T(n/b^2) = a^{log_b n}T(1) = cn^{log_b a}$（课本附录A定理）
- 本质上对比$aT(n/b)$和$f(n)$增长速度，并利用课本$P43$的定理。

主定理应用例子

求解如下递归方程

1. $T(n) = 9T(n/3) + n$
2. $T(n) = T(n/2) + n$
3. $T(n) = T(n/3) + 1$

答案

1. $a = 9,\ b = 3,\ f(n) = n \in \Theta(n),$ 即$d = 1$
   由于$a = 9 > b^d = 3^1,$ 所以$T(n) \in \Theta(n^{log_b a}) = \Theta(n^2)$
2. $a = 1,\ b = 2,\ f(n) = n \in \Theta(n),$ 即$d = 1$
   由于$a = 1 < b^d = 2^1,$ 所以$T(n) \in \Theta(n^d) = \Theta(n)$
3. $a = 1,\ b = 3,\ f(n) = 1 \in \Theta(1),$ 即$d = 0$
   由于$a = 1 = b^d = 3^0,$ 所以$T(n) \in \Theta(n^d logn) = \Theta(logn)$

常见递归类型

减一算法

- 算法利用一个规模为n的实例和规模为$n-1$的给定实例之间的关系来对问题求解（插入排序，课本4.1）。
- $T(n) = T(n-1) + f(n)$

减常因子算法

- 规模为n的实例化简为一个规模为n/b的给定实例来求解（俄式乘法）。
- $T(n) = T(n/b) + f(n)$

分治算法

- 给定实例划分为若干个较小的实例，对每个实例递归求解，然后再把较小的实例合并成给定实例的一个解（快速排序，合并排序）。
- $T(n) = aT(n/b) + f(n)$

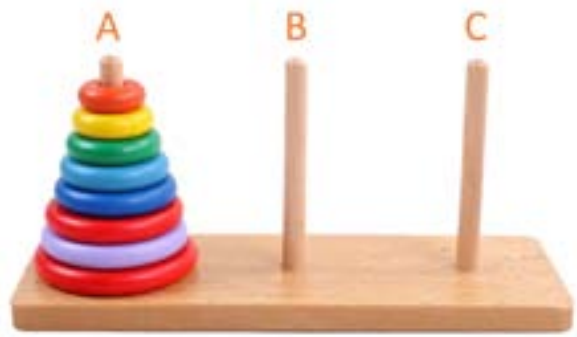
汉诺塔

汉诺塔问题

假设有A, B, C三根柱子，在A柱上放着n个圆盘，其中小圆盘放在大圆盘的上面。我们的目的是从A柱将这些圆盘移动到C柱上去，在必要的时候可借助B柱，每次只能移动一个盘子，并且不允许大盘子在小盘子上面。问需要多少次的移动？

解决办法

1. 把$n-1$个盘子递归的从A移动到B;
2. 把第n个盘子从A移动到C;
3. 把$n-1$个盘子递归的从B移动到C;
4. 如果$n=1$，直接把盘子从A移动到C;



汉诺塔问题递归分析

伪代码

```
1: function Hanoi(A, C, n)
2:     if n = 1 then
3:         move(A, C)
4:     else
5:         Hanoi(A, B, n − 1)
6:         move(A, C)
7:         Hanoi(B, C, n − 1)
8:     end if
9: end function
```

移动次数分析

- 递推关系

$$M(n) = \begin{cases} 2M(n-1)+1 & n > 1 \\ 1 & n = 1 \end{cases}$$

- 递归求解（课本P57）

$$\begin{aligned} M(n) &= 2^i M(n-i) + 2^i - 1 \\ &= 2^n - 1 \end{aligned}$$

注意

- 递归仅提供了一种分析手段，本身不具有降低算法复杂性的性质。
- 应该谨慎使用递归算法，它的简洁可能会掩盖其低效的事实。

主要内容

分治算法基本策略

分治法

1. 将一个问题划分为同一类型的若干子问题，子问题最好规模相同。
2. 对这些子问题求解（通常使用递归的方式）。
3. 如果有必要，合并这些子问题的解，以得到原问题的解。

例子

- 计算n个数字$a_1, a_2, ..., a_{n-1}$的和；
- 如果$n > 1$，可以把该问题划分为两个子问题，前一半的数字之和和后一半的数字之和；
- $a_1 + a_2 + ... + a_{n-1} + a_n = (a_0 + a_1 + ... + a_{n/2-1}) + (a_{n/2} + a_{n/2+1} + ... + a_{n-1})$
- 类似地一直划分，直到含有一个元素，直接返回该元素值；

分治法复杂度分析

主定理-回顾

设$T(n)$是一个非递减函数（定义见课本P376），并且满足递推式

$$T(n) = aT(n/b) + f(n)，\ 其中n = b^k，\ k = 1, 2, ...$$

$$T(1) = c$$

其中$a \geqslant 1$，$b \geqslant 2$，$c > 0$，如果$f(n) \in \Theta(n^d)$，$d \geqslant 0$，那么

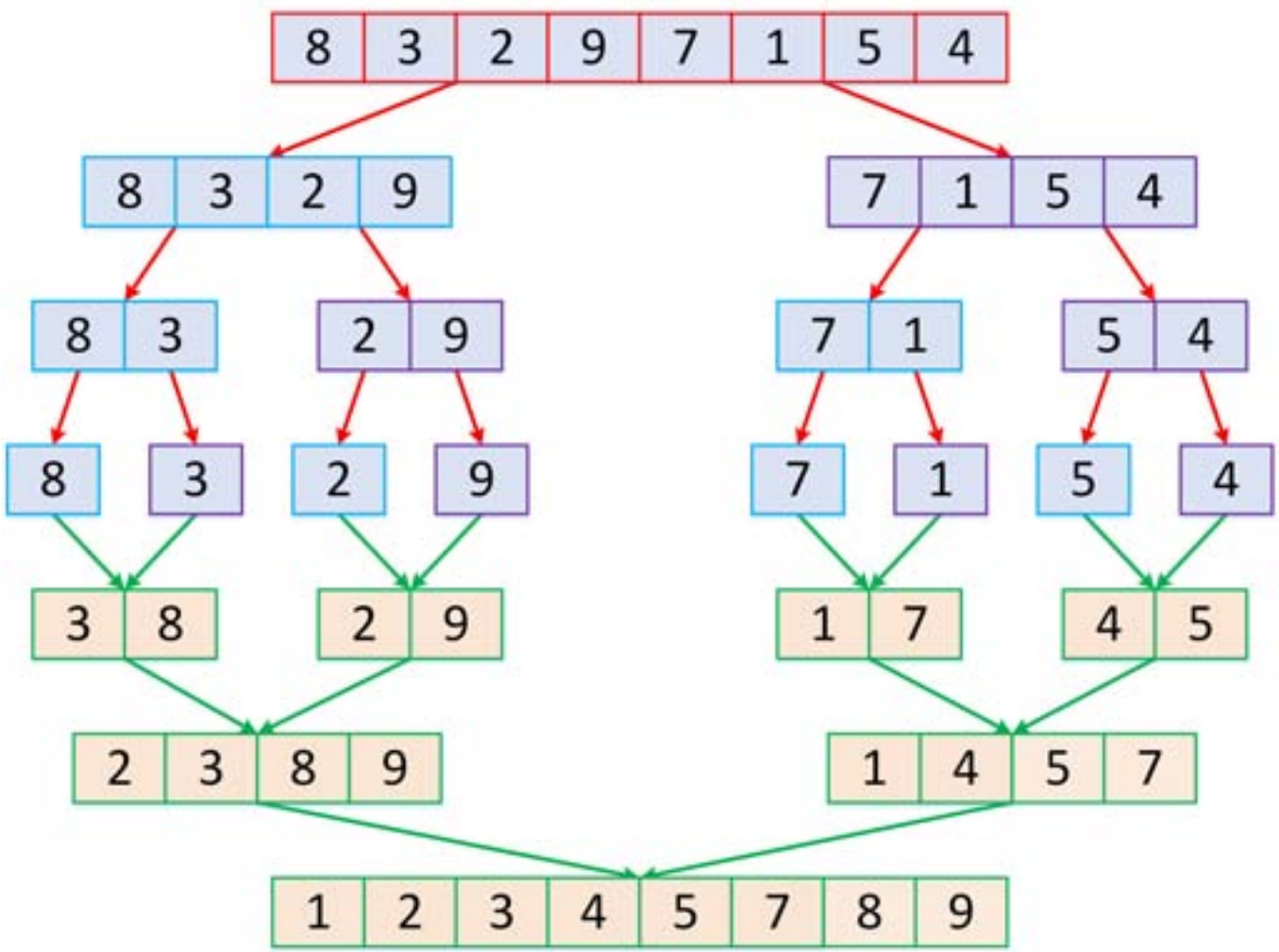
- 当$a < b^d$时，$T(n) \in \Theta(n^d)$；
- 当$a = b^d$时，$T(n) \in \Theta(n^d logn)$；　　**结论对符号$O$和$\Omega$也成立。**
- 当$a > b^d$时，$T(n) \in \Theta(n^{log_b a})$；

例子

- $T(n) = 2T(n/2) + 1$
- $a = 2, b = 2, d = 0, a = 2 > b^d = 2^0$，$T(n) \in \Theta(n^{logb_a}) = \Theta(n)$

合并排序

引例


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

伪代码

合并排序算法

1: **function** Merge_Sort($A[0...n-1]$)
2: **if** $n > 1$ **then**
3: **copy** $A[0...\lfloor n/2 \rfloor - 1] \rightarrow B[0...\lfloor n/2 \rfloor - 1]$
4: **copy** $A[\lfloor n/2 \rfloor ...n-1] \rightarrow C[0...\lceil n/2 \rceil - 1]$
5: **Merge_Sort**$(B[0...\lfloor n/2 \rfloor - 1])$
6: **Merge_Sort**$(C[0...\lceil n/2 \rceil - 1])$
7: **Merge**(B, C, A)
8: **end if**
9: **end function**

伪代码

对有序数据进行合并

```
1: function Merge(B[0...p-1], C[0...q-1], A[0...p+q-1])
2:     i ← 0; j ← 0; k ← 0;
3:     while i < p and j < q do
4:         if B[i] ⩽ C[j] then
5:             A[k] ← B[i]    //将较小的元素放入A
6:             i ← i + 1    //考查B中下一个元素
7:         else
8:             A[k] ← C[j]    //将较小的元素放入A
9:             j ← j + 1    //考查C中下一个元素
10:        end if
11:        k ← k + 1
12:    end while
13:    if i = p then    //B中元素检查完毕，直接复制C中剩余元素到A
14:        copy C[j...q-1] → A[k...p+q-1]
15:    else    //C中元素检查完毕，直接复制B中剩余元素到A
16:        copy B[j...p-1] → A[k...p+q-1]
17:    end if
18: end function
```

效率分析

- 算法基本操作：元素之间比较；
- 比较次数$C(n)$的递推关系：

$$C(n) = 2C(n/2) + C_{merge}(n)，\ C(0) = 1，\ 其中n = 2^k；$$

- 最坏情况下，$C_{merge}(n) = n - 1$；
- 最坏情况下效率

$$C_{worst}(n) = \begin{cases} 2C_{worst}(n) + n - 1 & n > 1 \\ 0 & n = 1 \end{cases}$$

- 根据主定理可得$C_{worst}(n) \in \Theta(nlogn)$
- 合并排序在最坏情况下的比较次数十分接近基于比较的排序算法的理论上能够达到的最少次数；
- 合并排序的主要缺点是需要线性的额外空间。

快速排序

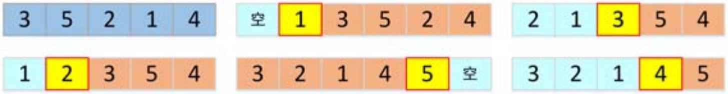
算法思想

- 合并排序按照元素的位置划分并合并的方式进行排序。划分过程很快，主要工作在合并子问题。
- 考虑如果按照元素的值进行划分子问题：

$$\underbrace{A[0]...A[s-1]}_{<A[s]} \quad \underbrace{A[s]}_{=} \quad \underbrace{A[s+1]...A[n-1]}_{>A[s]}$$

- 子问题递归排序后，合并很简单（拼接），主要工作在如何划分（如何有效地划分？），这就是快速排序算法。

子问题划分方式



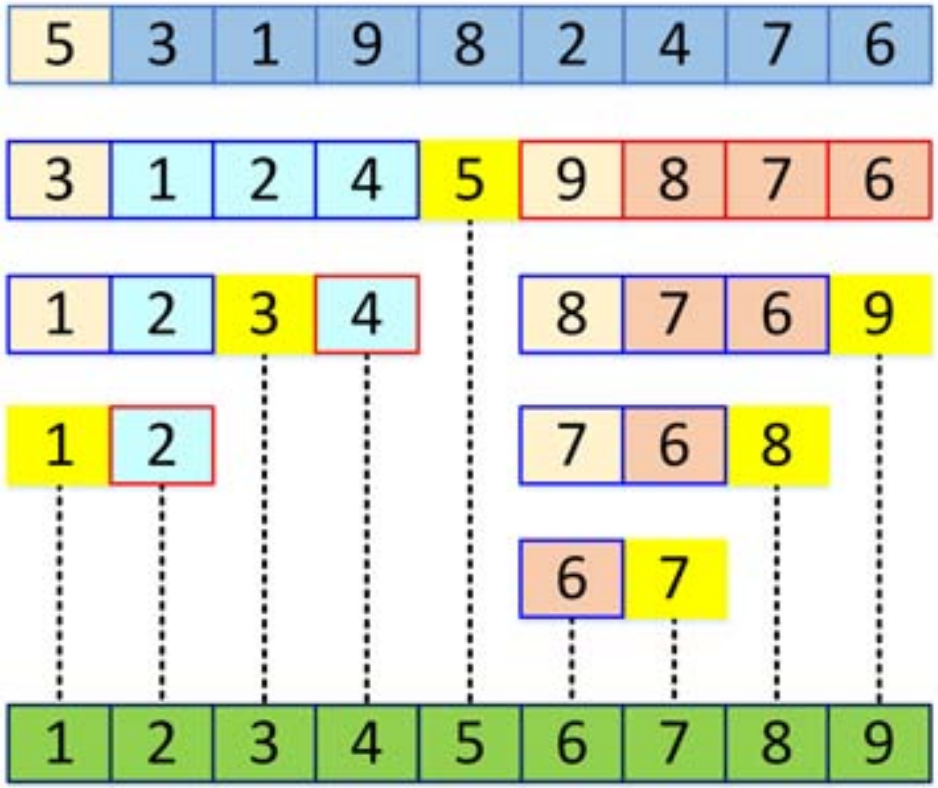
伪代码

快速排序算法

1: **function** Quick_Sort($A[l...r]$)
2: **if** $l < r$ **then**
3: $s \leftarrow$ **Partition**$(A[l...r])$ //找到分裂位置
4: **Partition**$(A[l...s-1])$
5: **Partition**$(A[s+1...r])$
6: **end if**
7: **end function**

简单的快速排序

依据首元素划分（递归调用）



霍尔划分方法

- 比划分值p小的元素位于左边，大的位于右边，简单地$p = A[0]$;
- 从左到右的扫描从第二个元素开始，直到$A[i] \geqslant p$;
- 从右到左的扫描从最后的元素开始，直到$A[j] \leqslant p$;
- 两边的扫描都停止后，会有下面三种情况：
  1. $i < j$，则$A[i] \leftrightarrow A[j]$;

  2. $i > j$，则$A[j] \leftrightarrow A[0]$;

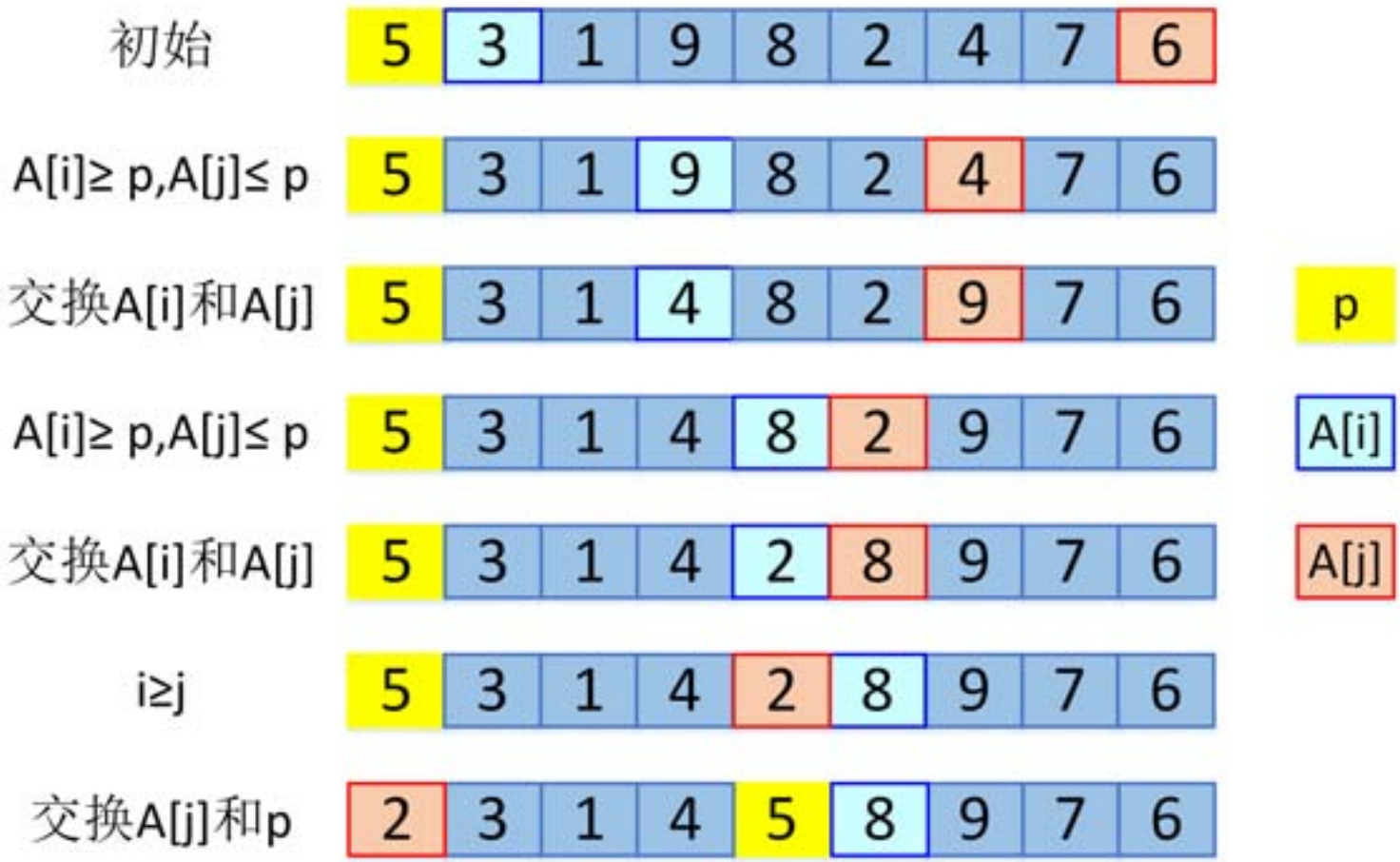
  3. $i = j$，则$A[i] = A[j] = p$;

霍尔划分过程

霍尔划分伪代码

```
1:  function Hoare_Partition(A[l...r])
2:      p ← A[l];   i ←l;   j ← r + 1
3:      repeat
4:          repeat      //找出A[i] ⩾ p的位置
5:              i ← i + 1
6:          until A[i] ⩾ p
7:          repeat      //找出A[j] ⩽ p的位置
8:              j ← j − 1
9:          until A[j] ⩽⩾ p
10:         A[i] ↔ A[j]      //i < j时，交换A[i]和A[j]
11:     until i ⩾ j
12:     A[i] ↔ A[j]      //当循环满足i ⩾ j后，执行了A[i] ↔ A[j]，需撤销
13:     A[l] ↔ A[j]
14: end function
```

实例


初始
5 3 1 9 8 2 4 7 6
A[i]≥ p,A[j]≤ p
5 3 1 9 8 2 4 7 6
交换A[i]和A[j]
5 3 1 4 8 2 9 7 6
p
A[i]≥ p,A[j]≤ p
5 3 1 4 8 2 9 7 6
A[i]
交换A[i]和A[j]
5 3 1 4 2 8 9 7 6
A[j]
i≥j
5 3 1 4 2 8 9 7 6
交换A[j]和p
2 3 1 4 5 8 9 7 6

实例


初始
2 3 1 4
8 9 7 6
A[i]≥ p,A[j]≤ p
2 3 1 4
8 9 7 6
交换A[i]和A[j]
2 1 3 4
8 6 7 9
i≥j
2 1 3 4
8 6 7 9
交换A[j]和p
1 2 3 4
7 6 8 9
初始
3 4
7 6
i≥j
3 4
7 6
交换A[j]和p
3 4
6 7
p
A[i]
A[j]

算法效率分析

最差情况

- 考虑对一个已经排好序的序列$[1,2,3,4,5,6,7,8,9.10]$进行排序；
- 从左到右扫描，停止在$A[1]$；从右到左扫描，停止在$A[0]$；
- $A[0]$和本身交换，然后继续递归排序$[2,3,4,5,6,7,8,9.10]$；
- 第1次分裂需要的比较次数为$1+n$；
- 第2次分裂需要的比较次数为n;
- 最后一次分裂比较3次（$A[i]$比较1次，$A[j]$比较2次）；
- $C_{worst}(n) = (n+1)+n+...+3 = (n+1)(n+2)/2-3 \in \Theta(n^2)$

算法效率分析

最优情况

- 最优情况下，数组序列被分为两个大小差不多规模的子序列；
- $C_{best}(n) = 2C_{best}(n/2) + \Theta(n), C_{best}(1) = 0$；
- 根据主定理可得$C_{best}(n) \in \Theta(nlog_2n)$；

平均情况

- 经过$n+1$次比较，分裂点出现在任意的位置$s(0 \leqslant s \leqslant n-1)$，划分数组为大小为$s$和$n-1-s$两个部分，每种情况出现的概率为$1/n$；
- 递推方程：

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)], C_{avg}(1) = C_{avg}(0) = 0$$

- 计算可得$C_{avg}(n) \approx 2nln2 \approx 1.39nlog_2n$；
- 算法平均情况下的效率仅比最优情况下仅仅多执行39%的计算；

选择问题

问题的一般描述

设L是n个元素的集合，从L中选择第k小的元素，当L中的元素从小到大排列后，排在第k个位置的元素。

- $k = 1$：最小元素；
- $k = n$：最大元素；
- $k = n - 1$：第二大元素；
- $k = \lceil n/2 \rceil$：中位数；

选最大和选最小：顺序比较法

- 从前到后扫描，找出最大元素或者最小元素；
- 比较次数：$W(n) = n - 1$；

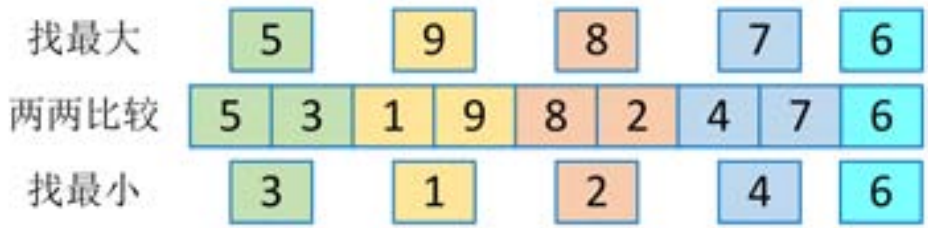
同时选最大和最小

顺序比较

- 调用顺序比较找最大和最小算法，先找出最大，然后把最大从L中删除，继续找出最小；
- 比较次数：$W(n) = n - 1 + n - 2 = 2n - 3$；

分组算法

- 分组（$\lfloor n/2 \rfloor$组）比较，再从比较结果中$\lceil n/2 \rceil$中找出最大和最小；
- 比较次数：$W(n) = \lfloor n/2 \rfloor + 2(\lceil n/2 \rceil - 1) = \lceil 3n/2 \rceil - 2$；



同时选最大和最小

分治算法

1. 将L从中间划分为L_1和L_2;
2. 递归的在$L1$中找最大max_1和最小min_1;
3. 递归的在$L2$中找最大max_2和最小min_2;
4. $max \leftarrow max(max_1, max_2)$;
5. $min \leftarrow min(min_1, min_2)$;

效率分析

- 比较次数递归方程：$W(n) = 2W(n/2) + 2, W(2) = 1, n = 2^k$;
- 递推计算可得：$W(n) = 3n/2 - 2$;

找第二大元素

顺序比较

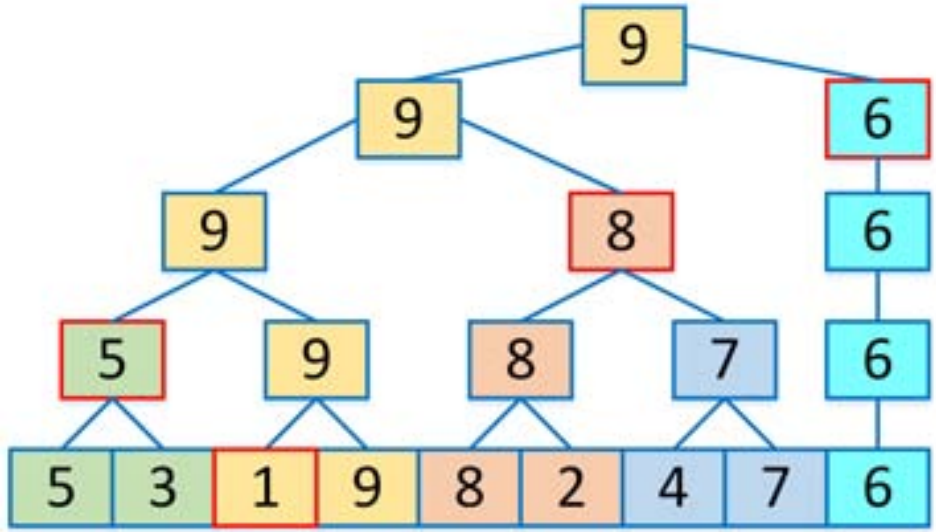
- 调用顺序比较找最大算法，先找出最大，然后把最大从L中删除，继续找出最大；
- 比较次数：$W(n) = n - 1 + n - 2 = 2n - 3$；

锦标赛算法

- 第二大数的数仅在与最大数的比较中被淘汰；
- 两两分组比较，大者进入下一轮，直到剩下1个元素max为止；
- 在每次比较中淘汰较小元素，将被淘汰元素记录在淘汰它的元素的链表上；
- 检查max链表，从中找到最大，即$second$；

锦标赛算法效率分析

- 算法两个部分：（1）找max；（2）从被max淘汰的数据中找最大；
- 第（1）步淘汰了$n-1$个元素，比较次数为$n-1$；
- 第（1）步中与max进行比较的元素数目为$\lceil logn \rceil$；
- 第（2）步从$\lceil logn \rceil$个数中找最大，比较次数为$\lceil logn \rceil - 1$；
- 总的比较次数为：$W(n) = n - 1 + \lceil logn \rceil - 1 = n + \lceil logn \rceil - 2$；



一般性选择问题（选择第k小）

分治法求解思路

- 用某个元素m^*作为比较标准将S划分成S_1与S_2；
- 如果 $k \leqslant |S_1|$，则在S_1中找第k小；
- 如果 $k = |S_1| + 1$，则在m^*是第k小；
- 如果 $k > |S_1| + 1$，则在S_2中找第$k - |S_1| - 1$小；

注意

- 寻找m^*（选择问题）的代价不能太高，假设为$T(cn)$；
- 算法在最坏的情况下总进入$S1$和$S2$中较大（规模为dn）的子集；
- 定理：当$c + d < 1$时，$T(n) = T(cn) + T(dn) + O(n) \in O(n)$；
- 在寻找m^*时应保证$c + d < 1$才能够保证算法复杂度为$O(n)$；

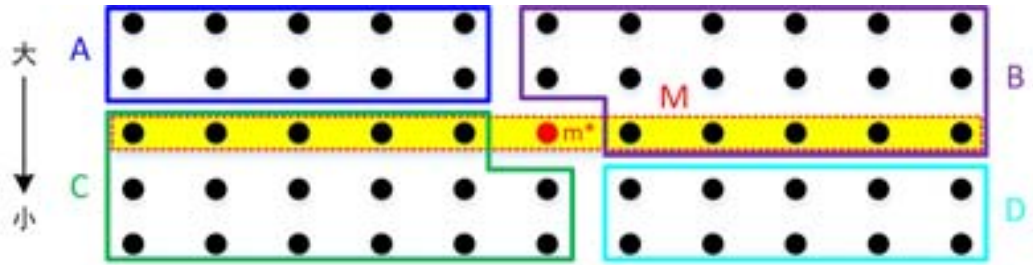
算法过程与复杂度分析

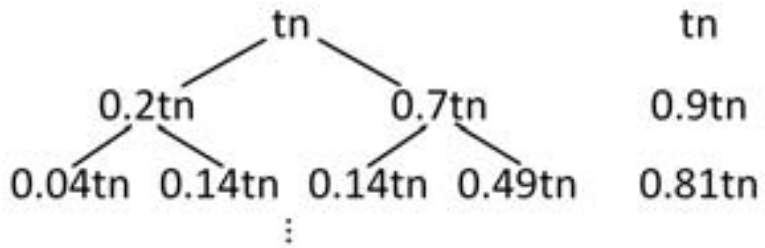
- 将S均分成$\lceil n/5 \rceil$组，每组找出中位数构成M（复杂度为$O(n)$）；
- 在M中寻找中位数做为m^*（复杂度为$T(cn), c = 0.2$的子问题）；
- A和D中小于m^*的元素加入$C(S_1)$，大的加入$B(S_2)$（复杂度为$O(n)$）；
- 假设n是5的倍数，且$n/5$为奇数，即$n/5 = 2r + 1$，则有

$$|A| = |D| = 2r, |B| = |C| = 3r + 2, n = 10r + 5$$

$$max(|S_1|, |S_2|) = |A| + |D| + |C| = 7r + 2 < 0.7n, d = 0.7$$

$$C_{worst}(n) \leqslant T(0.2n) + T(0.7n) + tn \leqslant tn + 0.9tn + 0.9^2tn + ... = O(n)$$





伪代码

选择第k小算法

1: **function** Select(S, k)
2: 将S每5个划分为一组，每组排序，中位数放到集合M中；
3: $m^* \leftarrow$ **Select** $(M, \lceil |M|/2 \rceil)$;
4: 根据m^* =划分S为A, B, C, D四个部分;
5: A和D中元素与m^*比较，小的构成S_1，大的构成S_2;
6: $S_1 \leftarrow S_1 \cup C$; $S_2 \leftarrow S_2 \cup B$;
7: **if** $k = |S1| + 1$ **then**
8: **return** m^*;
9: **else if** $k \leqslant |S_1|$ **then**
10: **Select** (S_1, k);
11: **else**
12: **Select** $(S_2, k - |S_1| - 1)$;
13: **end if**
14: **end function**

快速选择算法

算法思想

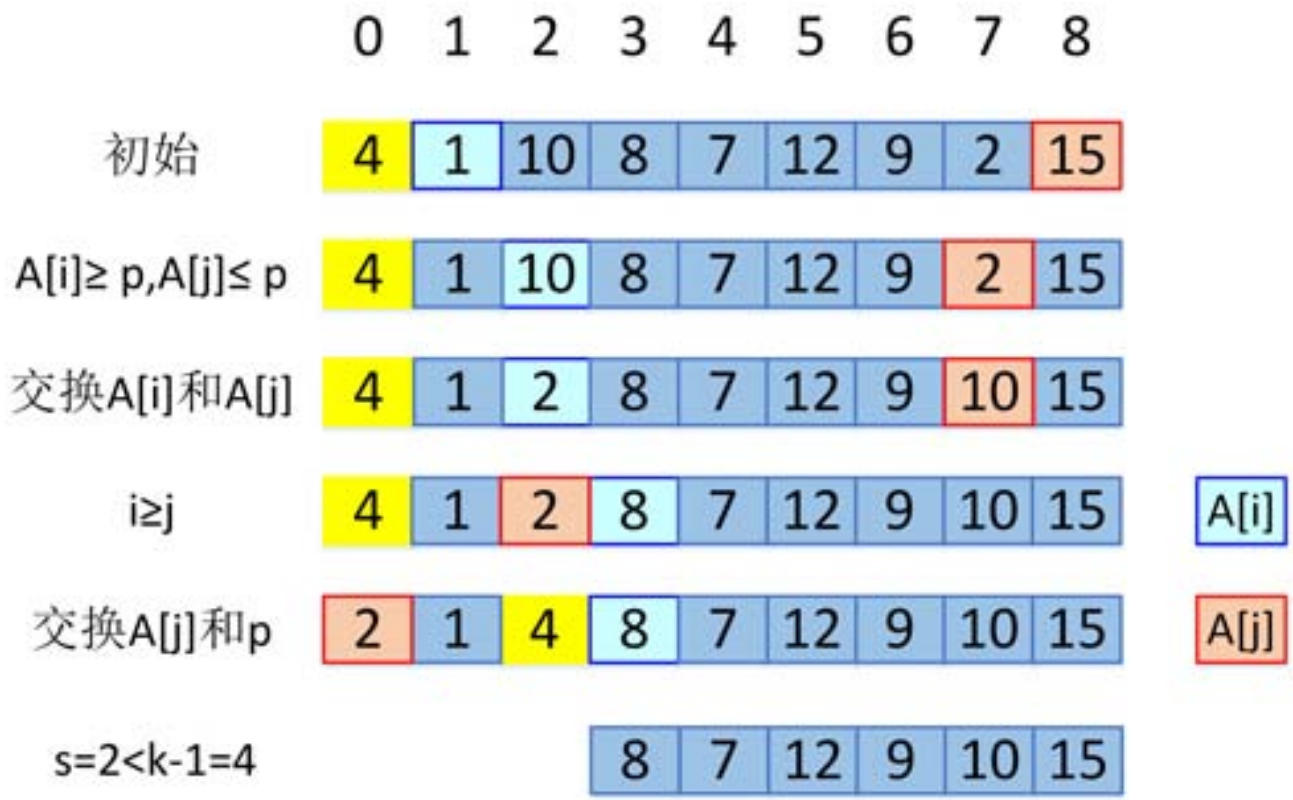
- 快速排序算法按照元素的值进行划分子问题：

$$\underbrace{A[0]...A[s-1]}_{<A[s]} \quad A[s] \quad \underbrace{A[s+1]...A[n-1]}_{>A[s]}$$

- $A[s]$是第$s+1$小的元素；
- 如果$k=s+1$，则$A[s]$是要找的元素；
- 如果$k>s+1$，则在$A[s+1]...A[n-1]$中寻找；
- 如果$k<s+1$，则在$A[0]...A[s-1]$中寻找；
- 如此重复划分过程，直到$k=s+1$；
- 划分方法：Lomuto算法（$Page123$），Hoare划分方法（$Page138$）；

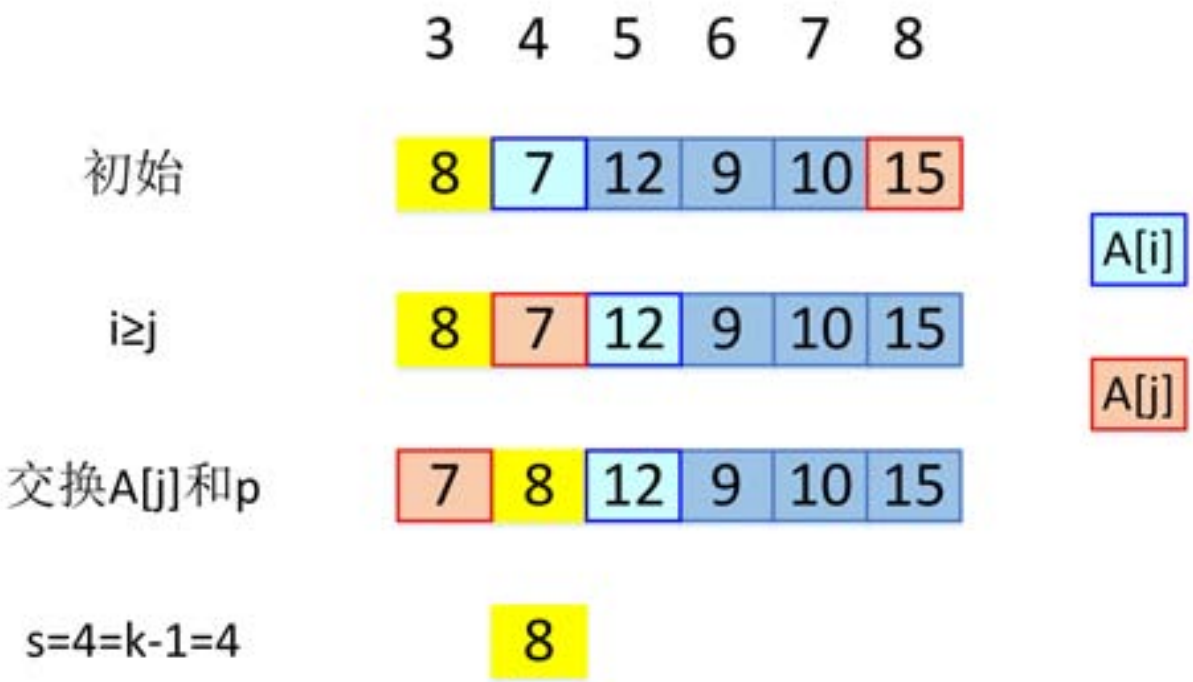
实例：寻找9个数的中位数($k = 5$)

第一次划分



实例：寻找9个数的中位数($k = 5$)

第二次划分



伪代码

快速选择算法

```
1: function Quick_Select(A[l...r], k)
2:     s ← Partition (A[l...r]);
3:     if s = l + k − 1 then
4:         return A[s];
5:     else if s > l + k − 1 then
6:         Quick_Select (A[l...s − 1], k);
7:     else
8:         Quick_Select (A[s + 1...r], l + k − 1 − s);
9:     end if
10: end function
```

快速选择算法效率分析

- 对于n个元素的数组进行划分总是需要$n-1$次比较；
- 最好情况下，首划分就找到想要的元素，$C_{best}(n) = n-1 \in \Theta(n)$；
- 最坏情况下，每次划分仅仅少一个元素（快速排序最差情况）；
- $C_{worst} \in \Theta(n^2)$（参考课件快速排序最差效率分析）；
- 快速选择算法的实用性依赖于其平均效率（线性，$C_{avg} = cn$）；

大整数乘法

大整数乘法问题

- 某些应用中，如当代的密码技术，需要计算超过上千位的二进制数的乘积;
- 假设X和Y是两个n位的二进制数，$n = 2^k$，计算XY;

解决办法

- 按照通常的做法，需要总共n^2次乘法计算;
- 考虑分治法，将X和Y分成相等的两段，每段$n/2$位，即

$$X = A2^{n/2} + B$$

$$Y = C2^{n/2} + D$$

$$XY = AC2^n + (AD + BC)2^{n/2} + BD$$

- 规模为n的原问题转换为4个规模为$n/2$的子问题;

分治法求解

复杂度分析

- 计算需要的乘法次数递归方程：

$$W(n) = \begin{cases} 4W(n/2) & n > 1 \\ 1 & n = 1 \end{cases}$$

- 根据主定理可得到$W(n) \in \Theta(n^2)$

注意

- 虽然采用了分治法，但是时间复杂度并没有降低；
- 回顾主定理，当问题规模减半$b = 2$，合并（加减法运算次数）的复杂度为$\Theta(n)$时，子问题数$a > 2$时，时间复杂度为$\Theta(n^{log_b a})$，减少子问题数可降低时间复杂度。

改进的分治法

改进思路

- $AD + BC = (A - B)(D - C) + AC + BD$
- AC和BD已知，子问题数目从原来的4个变为3个；

复杂度分析

- 乘法次数递归方程：$M(n) = 3M(n/2), n > 1; M(1) = 1.$；
- 根据主定理可得到$M(n) \in \Theta(n^{log_2 3}) \approx \Theta(n^{1.59})$；
- 合并子问题的复杂度：6次规模为n的整数加减法操作，可记为cn；
- 加减次数递归方程：$A(n) = 3A(n/2) + cn, n > 1; A(1) = 0$；
- 根据主定理可得到$A(n) \in \Theta(n^{log_2 3}) \approx \Theta(n^{1.59})$；
- 算法总的时间效率为$\Theta(n^{1.59})$，效率有明显的提升；

矩阵乘法

矩阵相乘问题

假设A和B是两个n阶的矩阵，$n = 2^k$，计算$C = AB$；

解决办法

- $C_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + ... + a_{in}b_{nj}$
- 计算C_{ij}需要n次乘法（不考虑加法），计算C需要n^3次乘法；
- 考虑分治法：

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

$$C_{11} = A_{11}B_{11} + A_{12}B_{21}, C_{12} = A_{11}B_{12} + A_{12}B_{22}$$

$$C_{21} = A_{21}B_{11} + A_{22}B_{21}, C_{22} = A_{21}B_{12} + A_{22}B_{22}$$

- 规模为n的原问题转换为8个规模为$n/2$的子问题；

分治法求解

复杂度分析

- 乘法次数递归方程：

$$M(n) = \begin{cases} 8M(n/2) & n > 1 \\ 1 & n = 1 \end{cases}$$

- 根据主定理可得到$M(n) \in \Theta(n^{log8}) = \Theta(n^3)$

注意

- 与传统方法比，时间复杂度并没有降低；
- 同样地，考虑减少子问题数以降低时间复杂度。

Strassen（施特拉森）矩阵乘法

- 分块矩阵相乘：

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

- 中间结果：

$$M_1 = A_{11}(B_{12} - B_{22}), M_2 = (A_{11} + B_{12})B_{22}$$

$$M_3 = (A_{21} + B_{22})B_{11}, M_4 = A_{22}(B_{21} - B_{11})$$

$$M_5 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$M_6 = (A_{12} - A_{22})(B_{21} + B_{22})$$

$$M_7 = (A_{11} - A_{21})(B_{11} + B_{12})$$

- 计算最终结果：

$$C_{11} = M_5 + M_4 - M_2 + M_6$$

$$C_{12} = M_1 + M_2, C_{21} = M_3 + M_4$$

$$C_{22} = M_5 + M_1 - M_3 - M_7$$

时间复杂度分析

- 合并子问题的复杂度：矩阵加法（$(n/2)^2$个元素相加）18次；
- 乘法计算总次数递归方程：

$$M(n) = \begin{cases} 7M(n/2) & n > 1 \\ 1 & n = 1 \end{cases}$$

- 根据主定理可得到$M(n) \in \Theta(n^{log_2 7}) \approx \Theta(n^{2.807})$
- 加法计算总次数递归方程：

$$A(n) = \begin{cases} 7A(n/2) + 18(n/2)^2 & n > 1 \\ 0 & n = 1 \end{cases}$$

- 根据主定理可得到$A(n) \in \Theta(n^{log_2 7}) \approx \Theta(n^{2.807})$
- 算法总的时间效率为$\Theta(n^{2.807})$，效率有明显的提升；

最近对问题

问题回顾

- 平面上有n个点，$P_1, P_2, ..., P_n$，$n > 1$，P_i的直角坐标是(x_i, x_j)，求距离最近的两个点之间的距离。P_i和P_j距离计算如下：

$$d(P_i, P_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

- 蛮力查找算法距离计算的总次数为$C_n^2 = n(n-1)/2$。

分治算法求解

- $2 < n < 3$时，可采用蛮力法求解；
- $n > 3$时，利用数据点P在x点上的中位数m，将点划分为P_L和P_R；
- 递归求解P_L和P_R中的最近点的距离，假设分别为d_L和d_R；
- $d = min(d_L, d_R)$，检查距离中线l（$x = m$）为d的窄缝里是否有小于d的点对存在，如果存在，替换d；

子问题分解

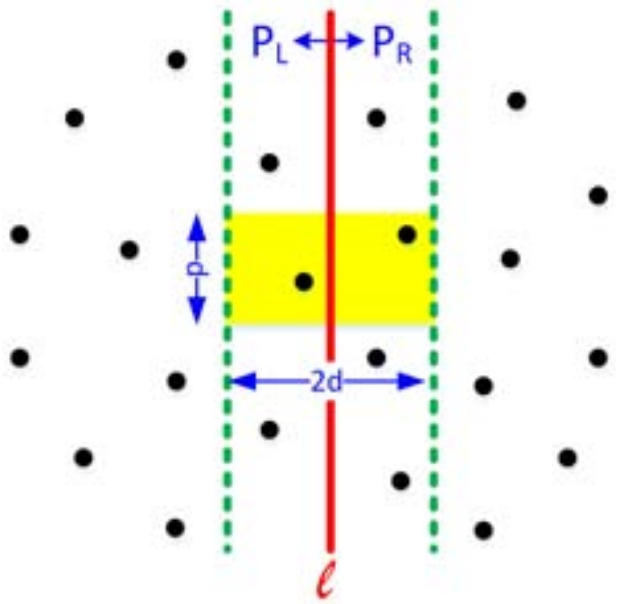
- 算法需要不断的利用中位数对子问题进行划分，因此对坐标的预排序可以有效降低算法时间复杂度；
- X表示数据点P按照x递增的顺序排好后的坐标；
- Y表示数据点P按照y递增的顺序排好后的坐标；
- 子问题调用需要P_L的有序坐标X_L和Y_L及P_R的有序坐标X_R和Y_R；

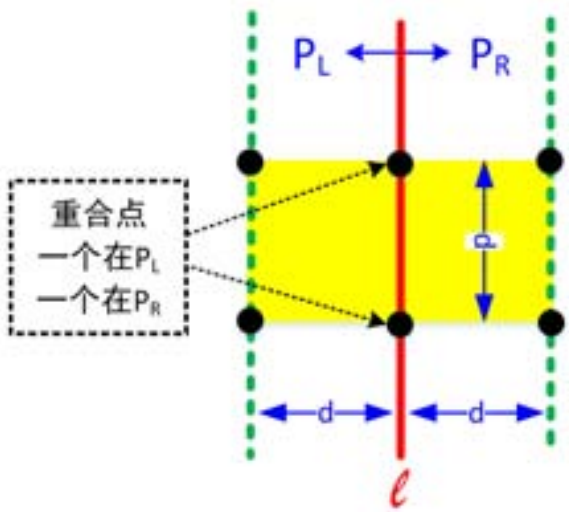
原始数据P					预排序数据X和Y				
P_x	0.5(1)	2(2)	-2(3)	1(4)	X	-2(3)	0.5(1)	1(4)	2(2)
P_y	2(1)	3(2)	4(3)	-1(4)	Y	-1(4)	2(1)	3(2)	4(3)

子数组P_L			子数组P_R		
X_L	-2(3)	0.5(1)	X_R	1(4)	2(2)
Y_L	2(1)	4(3)	Y_R	-1(4)	3(2)

合并子问题

- 子问题合并需要检查以直线l为中心，宽度为$2d$的区域内的点（记为Y'，从Y中获得，有序），是否存在距离小于d的点对；
- 对于Y'中的每个元素p，只需要检查其下方的$d*2d$的区域，该区域内最多只有8个点（包含p），因此仅检查其后的7个点；





伪代码

```
1: function Efficient_Closest_Pair(X, Y) //X和Y分别为点的已经排序的横纵坐标
2:     if n ⩽ 3 then
3:         return 由蛮力计算法求出的最小距离;
4:     else
5:         根据X的中位数m，划分得到X_L和X_R，进而从Y中抽取得到Y_L和Y_R;
6:         d_L ← Efficient_Closest_Pair (X_L, Y_L); d_R ← Efficient_Closest_Pair (X_R, Y_R);
7:         d ← min(d_L, d_R); dminsq ← d^2;
8:         从Y中去除不在距离中位线l为d的窄缝里点，得到点集Y'[1...m];
9:         for i ← 0 → m − 1 do
10:            k ← i + 1;
11:            while k ⩽ 7 and (Y'[k].y − Y'[i].y)^2 < d^2 do
12:                dminsq ← min(dminsq, (Y'[k].x − Y'[i].x)^2 + (Y'[k].y − Y'[i].y)^2);
13:                k ← k + 1;
14:            end while
15:        end for
16:        return √dminsq;
17:    end if
18: end function
```

算法复杂度分析

运行时间

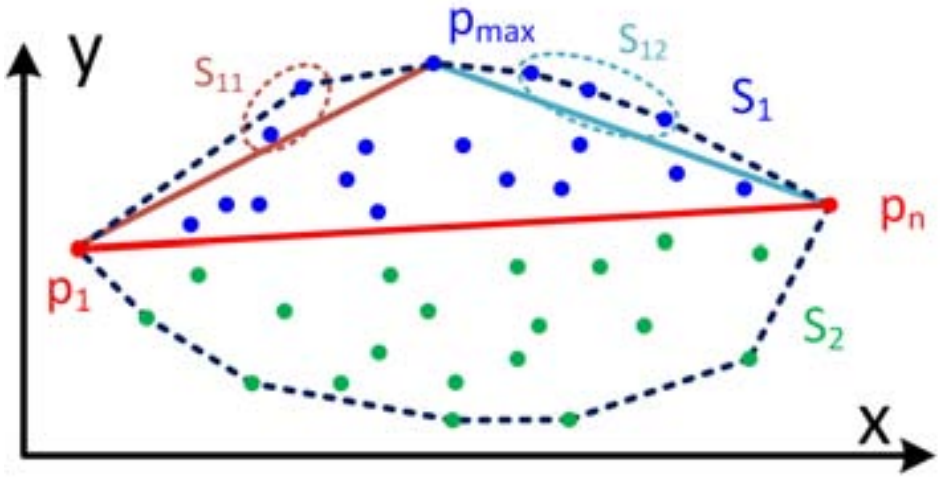
- 运行时间递归方程：$T(n) = 2T(n/2) + f(n)$，其中$f(n) \in \Theta(n)$；
- 根据主定理（$a = 2, b = 2, d = 1$），可得到$T(n) \in \Theta(nlogn)$；
- 假设采用复杂度为$\Theta(nlogn)$的排序算法预排序，总体的复杂度为$\Theta(nlogn) + \Theta(nlogn) \in \Theta(nlogn)$；

注意

- 如果不进行预排序，需要在每次的递归调用中排序，运行时间递归方程为：$T(n) = 2T(n/2) + O(nlogn)$
- 计算可得$T(n) \in O(nlog^2n)$
- 改进分治算法的另一个途径为有效地预处理数据；

凸包问题–分治法求解

- 集合S是平面上$n > 1$个点，$p_1(x_1, y_1), p_2(x_2, y_2)...p_n(x_n, y_n)$，按照$x$和$y$的大小排序，则最左边的点$p_1$和最右边的点$p_n$一定是凸包的顶点；
- p_1p_n线将S分为S_1和S_2，寻找$p_1 \cup S_1 \cup p_n$和$p_1 \cup S_2 \cup p_n$的凸包；
- 找到S_1中距离p_1p_n线最远的点p_{max}，并可得到集合S_{11}和S_{12}；
- 寻找集合$p_1 \cup S_{11} \cup p_{max}$和集合$p_{max} \cup S_{12} \cup p_n$的凸包；
- 最坏情况下，子集的规模只能少一个点，效率为$\Theta(n^2)$



小结

- 递归的方式描述算法可让问题解决方法更加清晰，但并不一定具有性能上的优势（可能循环算法更好）；
- 分治策略通过将原问题划分为若干子问题分别求解，最后合并子问题的解得到原问题的解，复杂度的分析可采用递归的方式；
- 合并排序和快速排序平均效率都为$\Theta(nlogn)$，但是实际上快速排序性能更好一点（常量小）；
- 降低子问题数可提高分治法的效率（整数乘法和矩阵乘法问题）；
- 通过预排序，最近对问题算法的效率为$\Theta(nlogn)$（蛮力法$\Theta(n^2)$）；
- 凸包问题的分治策略的复杂度为$\Theta(n^2)$（蛮力法$\Theta(n^3)$）；
- 一般性选择问题的分治算法达到了理论上最低的算法性能$O(n)$；

作业

- 2.4: 3 $Page60$
- 2.4: 4 $Page60$
- 2.4: 6 $Page60$
- 5.1: 8 $Page135$
- 5.1: 9 $Page135$
- 5.2: 11 $Page141$
- 5.4: 7 $Page148$
- 5.5: 12 $Page154$

Part III

贪婪技术

阅读: 9.1–9.2 6.4.1 9.4 11.3 12.3

主要内容

1. 贪心策略
   - 几个例子
   - 贪心算法
2. 最小生成树
   - 概念
   - Prim算法
   - Kruskal算法
3. 哈夫曼树及编码
   - 最优前缀码
   - 哈夫曼算法
4. NP困难问题近似算法
   - P和NP问题
   - 旅行商问题
   - 背包问题

主要内容

找零问题

问题描述

假设有面额为$25, 10, 5, 1$美分的硬币，用这些硬币给出48美分的找零。

解决办法

不超过总额的情况下优先选择面额最大的，$25 * 1 + 10 * 2 + 3 * 1 = 48$。

注意

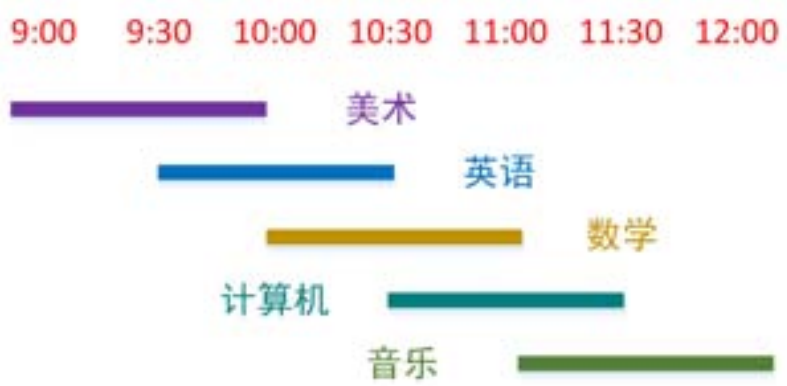
- 对于上面的实例算法给出了最优解；
- 考虑面额为$25, 10, 1$美分的硬币，找零30每分；
- 算法给出解：$25 * 1 + 5 * 1 = 30$；
- 实际最优解：$10 * 3 = 30$；

教室调度问题

问题描述

假设有如下课表，你希望尽可能多的课程安排在某间教室上。

课程	开始时间	结束时间
美术	9:00	10:00
英语	9:30	10:30
数学	10:00	11:00
计算机	10:30	11:30
音乐	11:00	12:00



解决办法

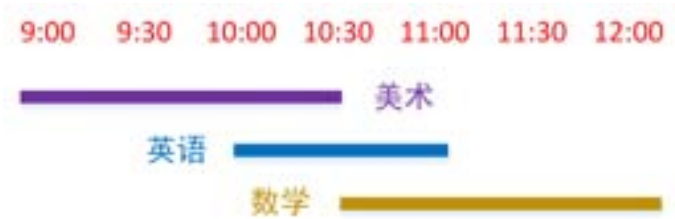
在时间不冲突的情况下，每一次都选择最早结束的课程。

美术	9:00	10:00
数学	10:00	11:00
音乐	11:00	12:00

贪心策略分析

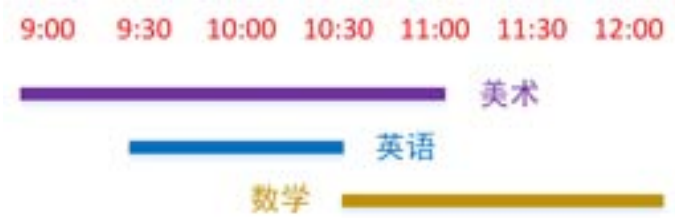
- 贪心算法每步都采取当前最优（局部最优），最终得到一个全局最优的方案；
- 选择最早结束的课程的策略可以找到教室调度问题最优解；
- 选择最小占用时间的课程得不到最优解；

美术	9:00	10:30
英语	10:00	11:00
数学	10:30	12:00



- 选择最早开始的课程得不到最优解；

美术	9:00	11:00
英语	9:30	10:30
数学	10:30	12:00



- 贪心算法不是任何情况下都有效，但是容易实现。

背包问题

问题描述

有一个贪婪的小偷，背着可以装35磅重东西的背包，可以选择盗窃如下的三种商品，小偷的目的是往背包里装入总价值最高的商品。

3000美元
30磅

2000美元
20磅

1500美元
15磅

贪心策略

- 不断地装入可装入背包的最贵商品，只能装入3000$的商品；
- 贪婪算法虽然不能得到最优解，但可以得到非常接近最优解的解；
- 如果需要找到能够大致解决问题的方案，贪心算法正好派上用场。

贪心算法

贪心算法特点

- 可行性：贪心算法每一步选择都必须满足问题的约束；
- 局部最优：算法不考虑全局最优，仅做出当前看来最优的选择；
- 不可取消：一旦做出选择，再算法后面的步骤中无法改变；

证明贪婪算法可以获得最优解的方法（本课程不涉及）

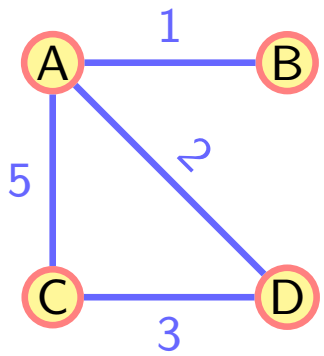
- 数学归纳法证明算法每一步获得的部分解能够扩展到全局最优解；
- 证明算法在接近目标的过程中，每一步的选择不会比其他算法差；
- 基于算法的输出，证明算法得到的解的最优性；

主要内容

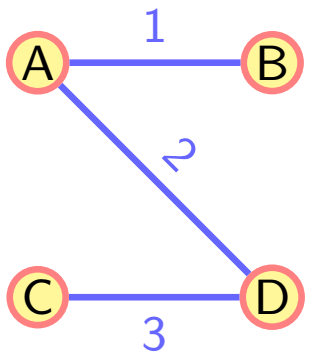
最小生成树

生成树

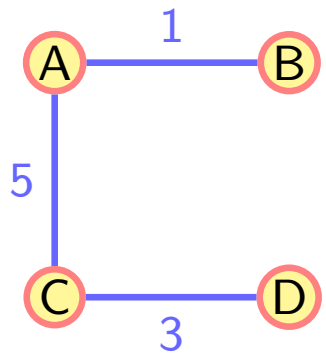
连通图的一棵生成树是包含图的所有顶点的连通无环子图（一棵树）。



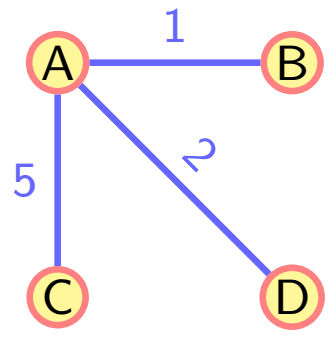
图



$W(T_1) = 6$



$W(T_2) = 9$



$W(T_3) = 8$

最小生成树

加权连通图的一棵最小生成树是图的一棵权重最小（T_1）的生成树。

Prim算法

图的定义回顾

图（Graph）是有一些称为顶点的点（V）构成的集合，其中某些顶点由一些称为边的线段（E）相连，$G =< V, E >$。

Prim算法流程

1. 任意选一点v_0，集合V被分割成两个集合$V_T = \{v_0\}$和$V - V_T$；
2. 从连通V_T和$V - V_T$的边中挑选一条权重最小的边$e^* = (v^*, u^*)$，$v^* \in V_T, u^* \in V - V_T$，将$u^*$加入$V_T$中，从$V - V_T$删除$u^*$；
3. 重复步骤2，直到集合$V - V_T$为空；

为什么Prim算法是贪心算法？

以贪婪的方式生成树，每次选择不在当前树中最短的边。

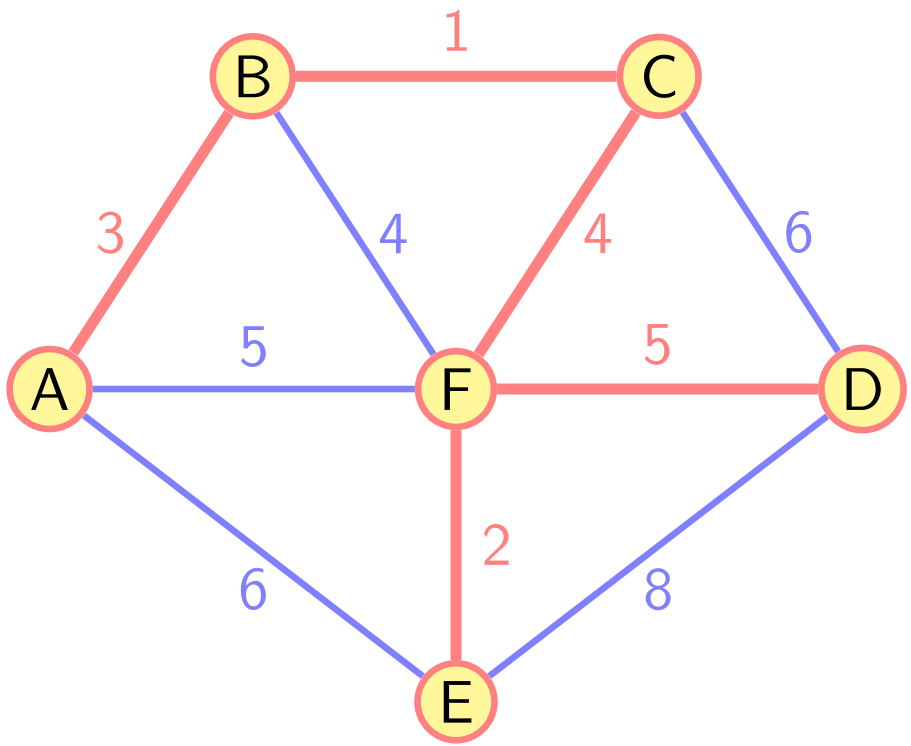
算法实例

算法核心

从连通V_T和$V - V_T$的边中挑选一条权重最小的边。

计算过程

- $V_T = \{A, B, C, F, E, D\}$
- $V - V_T = \{\varnothing\}$
- $W(T) = 15$

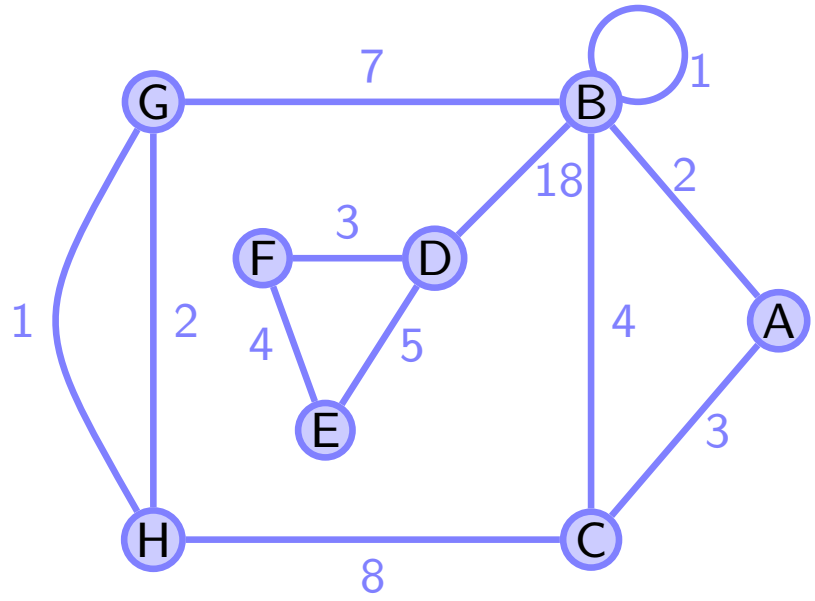


算法练习

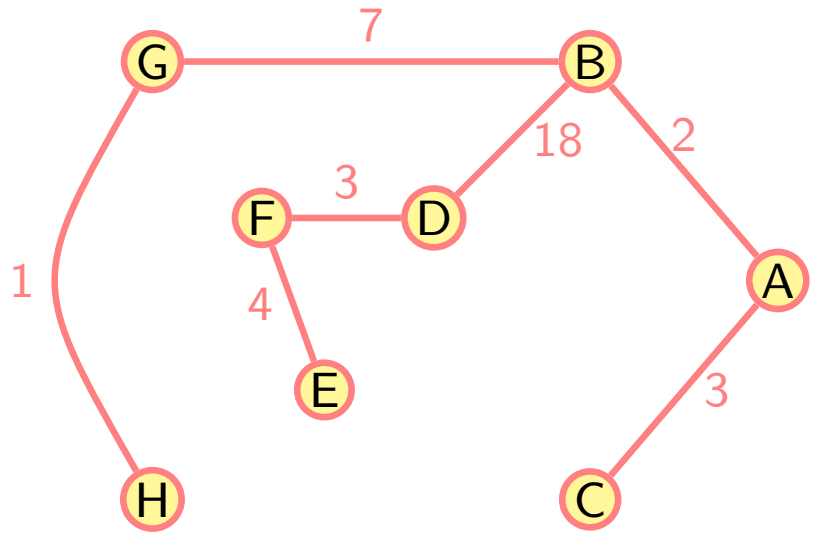
算法核心

从连通V_T和$V - V_T$的边中挑选一条权重最小的边。

图



最小生成树



伪代码

构造最小生成树的Prim算法

```
1:  function Prim(G)
2:      V_T ← v_0 //v_0为G中任一顶点
3:      E_T ← ∅
4:      for i ← 1 → |V| − 1 do
5:          for (v ∈ V_T) and (u ∈ V − V_T) do
6:              求权重最小的边e* = (v*, u*);
7:          end for
8:          V_T ← V_T ∪ {u*}
9:          E_T ← E_T ∪ {e*}
10:     end for
11:     return E_T
12: end function
```

算法效率分析–无序数组实现优先队列

计算过程中用到的数组

邻接矩阵

	A	B	C	D	E	F
A	∞	3	∞	∞	6	5
B	3	∞	1	∞	∞	4
C	∞	1	∞	6	∞	4
D	∞	∞	6	∞	8	5
E	6	∞	∞	8	∞	2
F	3	4	4	5	2	∞

V_T和$V-V_T$连接权重矩阵

序号	V_T	A	B	C	D	E	F
0	A	∞	3	∞	∞	6	5
1	A,B	∞	3	∞	∞	6	5
2	A,B	∞	∞	1	∞	6	4
3	A,B,C	∞	∞	1	∞	6	4
4	A,B,C	∞	∞	∞	6	6	4
5	...						

复杂度分析

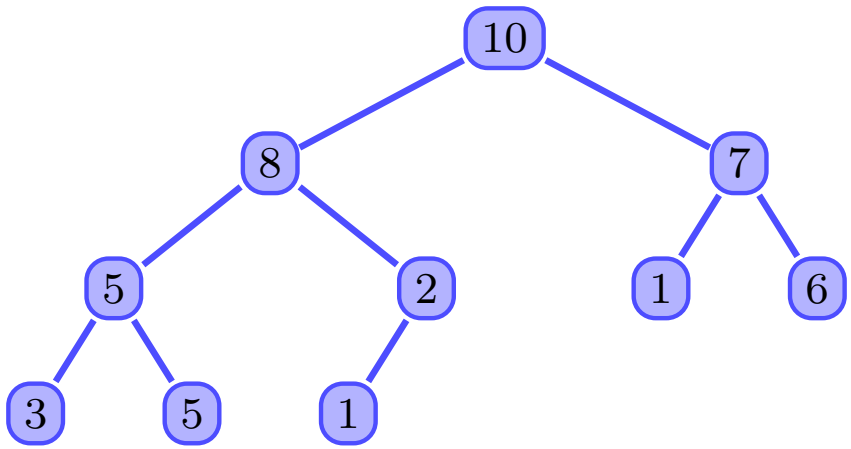
- 步骤1和3，从$Near$中选出最小的权值，复杂度为$O(|V|)$；
- 步骤2和4，根据V_T中节点，更新$Near$的权值，复杂度为$O(|V|)$；
- 一共需要向V_T中加入$|V|-1$个节点，总的复杂度为$O(|V|^2)$；

最小堆实现优先队列

堆

堆可以定义为一棵二叉树，并且满足下面的条件：

- 完备性：除了最后一层最右边的元素可能缺，其他层都是满的；
- 堆特性：每个节点的值大于等于它的子女节点的值；

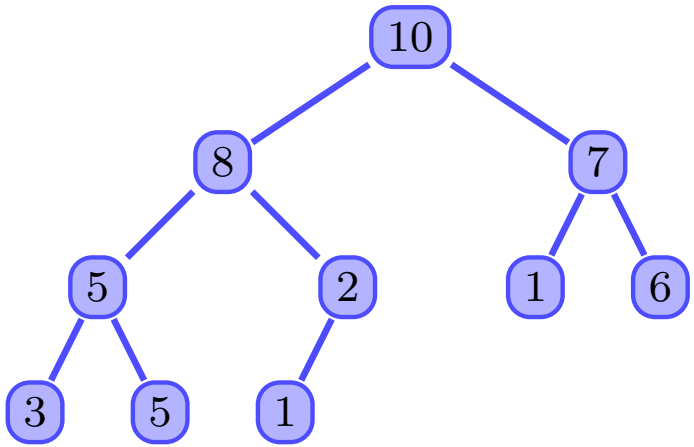


最小堆

每个节点的值都小于其子女的值，插入、删除等操作复杂度与堆相同。

堆的特性和实现

- 只存在一棵n个节点的完全二叉树，它的高度等于$log_2 n$；
- 可以用数据实现堆，从上到下，从左到右的方式记录堆的元素；
- 为了表达方便，在数组的开始位置设置为空；
- 父母节点i位于数组的前$\lfloor n/2 \rfloor$个位置，它的子女位于$2i$和$2i+1$；
- 对于一个位于$i(2 \leqslant i \leqslant n)$的建来说，它的父母将会位于$\lfloor i/2 \rfloor$。



堆的数组表示

0	1	2	3	4	5
	10	8	7	5	2

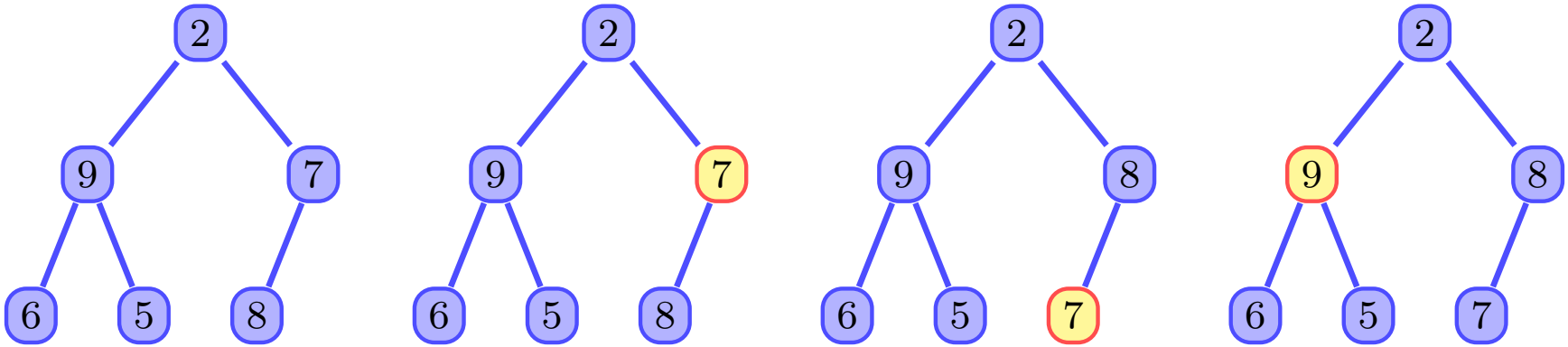
6	7	8	9	10
1	6	3	5	1

堆的构造–自底向上

方法要点

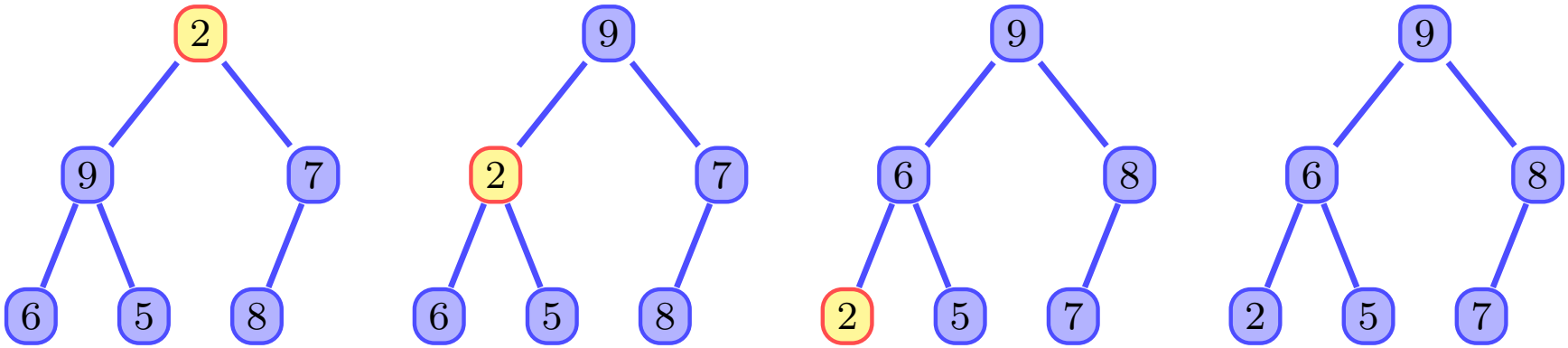
1. 按照给定的顺序放置节点；
2. 从最后的父母节点K开始，到根为止，检查节点是否满足要求；如果不满足，交换K和子女最大键值节点的位置；
3. 检查新的位置上，执行步骤2，检查键值K是不是满足要求；
4. 重复步骤2和3，直到对树的根完成操作。

实例：对列表$2, 9, 7, 6, 5, 8$构造堆



堆的构造–自底向上

实例：对列表$2, 9, 7, 6, 5, 8$构造堆



时间复杂度

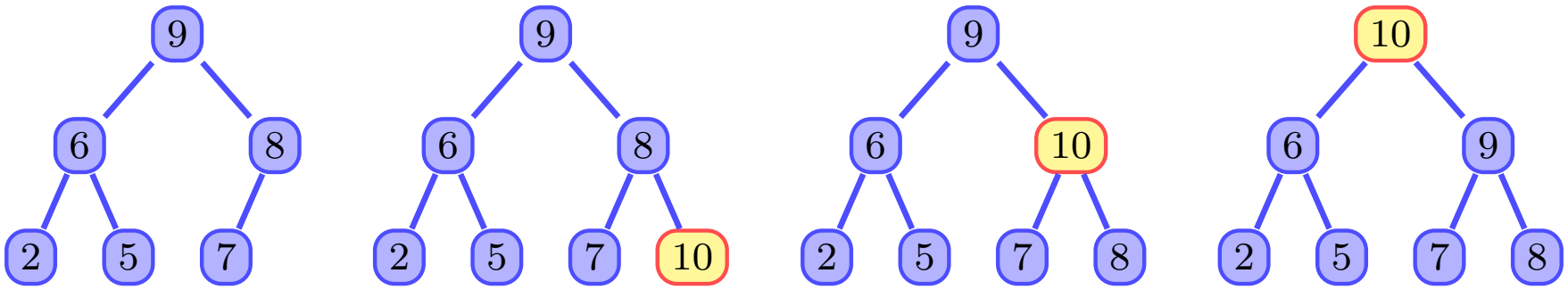
- 最坏情况下，每一层的节点的键都会交换到最底层；
- 移动一次需要两次比较：是否执行交换和找出最大的子女；
- 第i层的每个节点（一共有2^i个）需要$2(h-i)$次键值比较；
- $C_{worst}(n) = \sum_{i=0}^{log_2 n-1} 2^i * 2(log_2 n - i) = 2(n - log_2(n+1))$；

堆的构造–自顶向下

方法要点

- 通过把新的键连续插入预先构造好的堆来构造新的堆；
- 把一个键值为K的节点附加在当前堆的最后一个叶子节点后；
- 比较K和其父节点的大小，如果父节点的键值小，则交换；
- 重复上述步骤，直到遇到键值大于K的父节点或者位于根节点；

实例



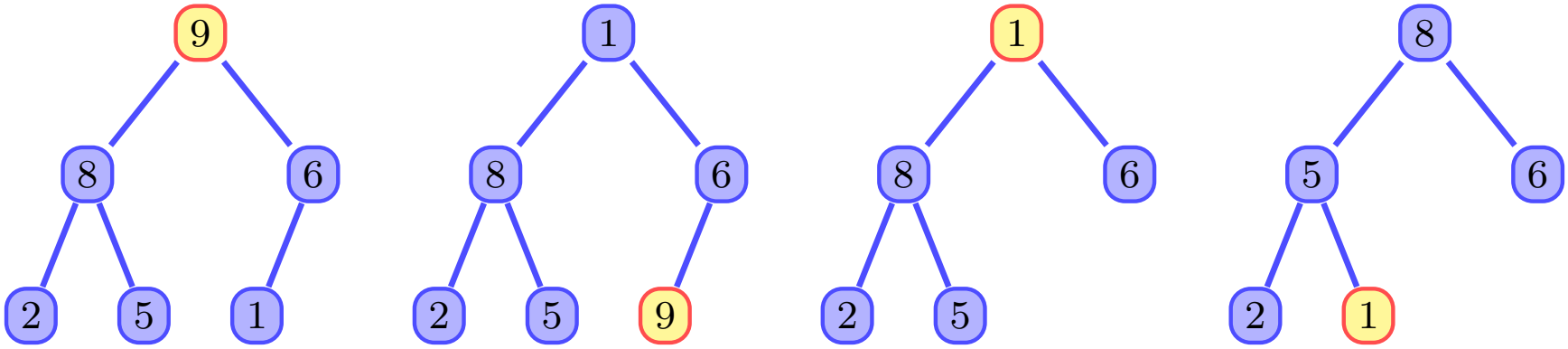
堆的操作

插入操作

与自顶向下构造堆相似，最坏情况下的复杂度为$O(log_2 n)$（树的高度）

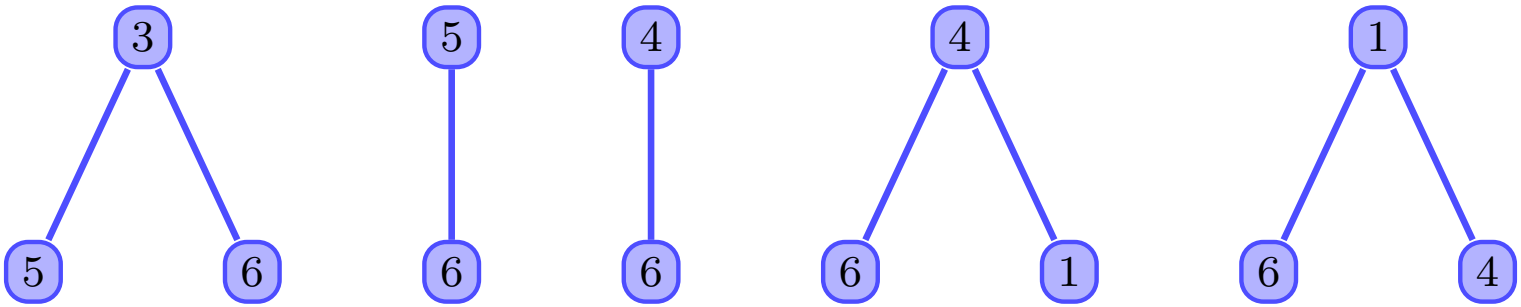
删除操作

1. 要删除的节点和堆的最后一个节点交换，然后删除；
2. “堆化”（自底向上构造堆步骤$2-3$）删除后的树；



算法效率分析–最小堆实现优先队列

过程示例



复杂度分析

- 删除最小连接边（最小堆最根元素），需要执行$|V|-1$次；
- 通过新的V_T更新最小堆（堆化和插入操作，最多执行$|E|$次）
- 堆的元素数不超过$|V|$，高度最大为$log\,|V|$，每个操作的复杂度为$O(logV)$;
- 总的复杂度为：$(|V|-1+|E|)O(log\,|V|)$;

Kruskal算法

算法思想

算法贪婪地选择最小权重的边，扩展无环的子图构造最小生成树。

算法流程

1. 按照权重非递减顺序对图中的边E进行排序；
2. 扫描以排序的列表，如果下一条边加入到当前的子图中不导致一个回路，则加入该边到子图中，否则跳过该边；
3. 重复步骤2，直到子图中有$|V|-1$条边；

算法的另一种解释

Kruskal算法可看作是对包含给定图所有顶点和某些边的一系列森林所做的连续动作。初始森林是$|V|$颗只包含一个顶点的树，通过加入边，将小数连通成大树，最终构造出一棵最小生成树。

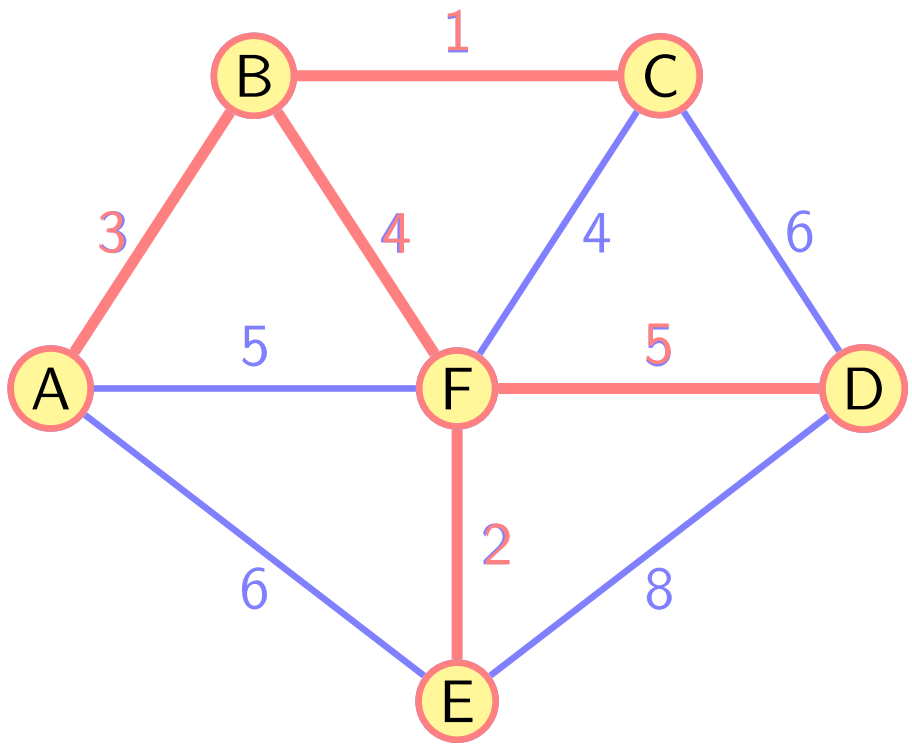
算法实例

算法核心

算法贪婪地选择最小权重的边构造生成树，所选择的边不能构成回路。

计算过程

- $(A, B) = 3$
- $(A, F) = 5$
- $(A, E) = 6$
- $(B, C) = 1$
- $(B, F) = 4$
- $(C, F) = 4$
- $(C, D) = 6$
- $(D, E) = 8$
- $(D, F) = 5$
- $(F, E) = 2$

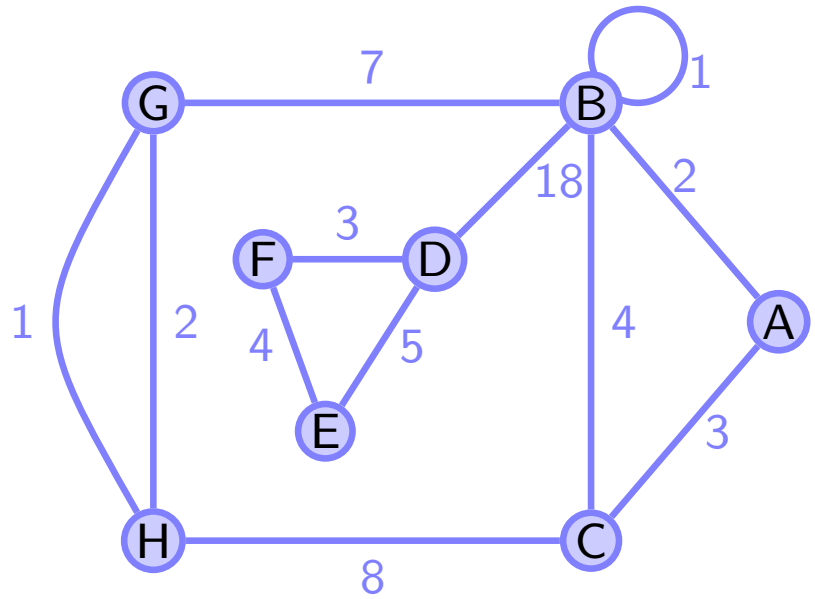


算法练习

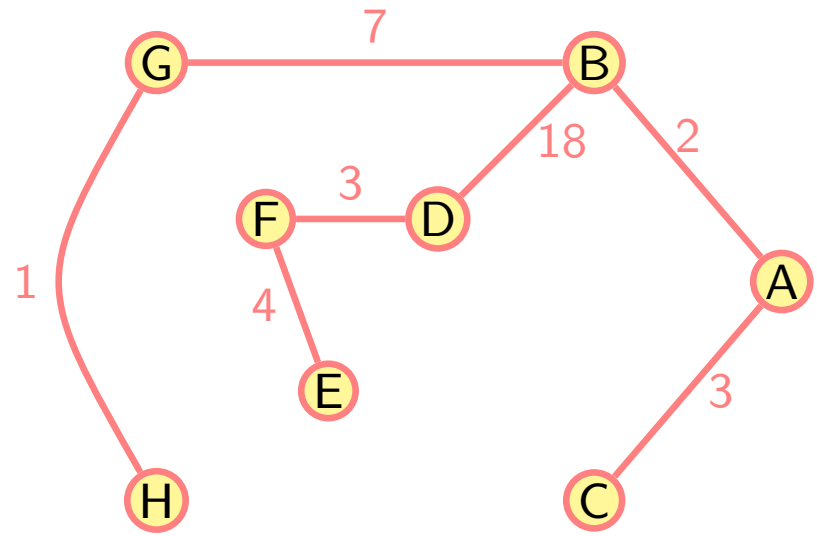
算法核心

算法贪婪地选择最小权重的边构造生成树，所选择的边不能构成回路。

图



最小生成树



伪代码

构造最小生成树的Kruskal算法

```
1:  function Kruskal(G)
2:      按照权重w(e_1) ⩽ ... ⩽ w(e_|E|)的非递增顺序对集合E排序;
3:      E_T ← ∅;
4:      ecounter ← 0;  k ← 0;
5:      while ecounter < (|V| − 1) do
6:          k ← k + 1;
7:          if E_T ∪ {e_k}无回路 then
8:              E_T ← E_T ∪ {e_k};
9:              ecounter ← ecounter + 1;
10:         end if
11:     end while
12:     return E_T
13: end function
```

难点：判断新加入的边是否构成回路

抽象数据类型

包含一些列不相交的子集和以下的操作：

- $makeset(x)$生成一个单元素集合x;
- $find(x)$返回一个包含x的子集；
- $union(x, y)$构造分别包含x和y的不相交子集的并集；

操作$makeset(x)$

集合：$\{a, b, c, d, e, f\}$

- $makeset(a) = \{a\}$
- $makeset(b) = \{b\}$
- $makeset(c) = \{c\}$
- $makeset(d) = \{d\}$
- $makeset(e) = \{e\}$
- $makeset(f) = \{f\}$

集合：$\{a\}, \{b\}, \{c\}, \{d\}, \{e\}, \{f\}$

难点：判断新加入的边是否构成回路

操作$union(x,y)$和$find(x)$

集合：$\{a\},\{b\},\{c\},\{d\},\{e\},\{f\}$

- $union(a,b)=\{a,b\}$
- $union(c,d)=\{c,d\}$
- $union(a,e)=\{a,b,e\}$
- $union(c,f)=\{c,d,f\}$

集合：$\{a,b,e\},\{c,d,f\}$

- $find(a)=\{a,b,e\}$
- $find(c)=\{c,d,f\}$
- $find(b)=\{a,b,e\}$
- $find(f)=\{c,d,f\}$

思考

如何用操作$find$和$union$判断回路？

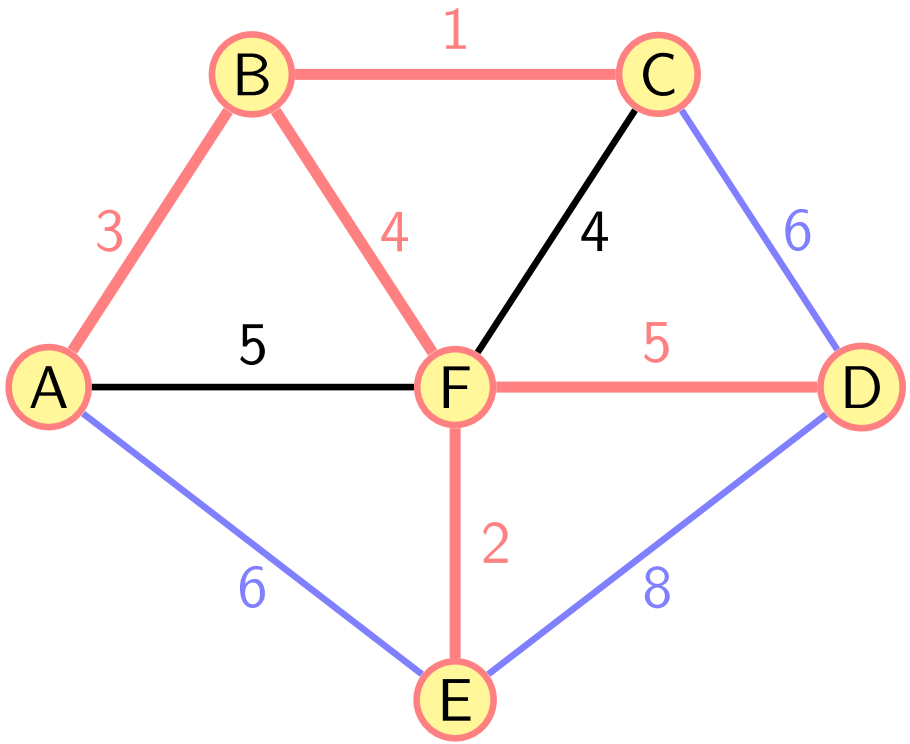
判断回路

边权重排序

1. $(B, C) = 1$
2. $(E, F) = 2$
3. $(A, B) = 3$
4. $(B, F) = 4$
5. $(C, F) = 4$
6. $(A, F) = 5$
7. $(D, F) = 5$
8. ...

操作过程

- $\{A, B, C, E, F\}, \{D\}$
- $find(D) = \{D\}$
- $find(F) = \{A, B, C, E, F\}$
- $union(D, F) = \{A, B, C, D, E, F\}$
- $\{A, B, C, D, E, F\}$



算法效率分析

算法要点

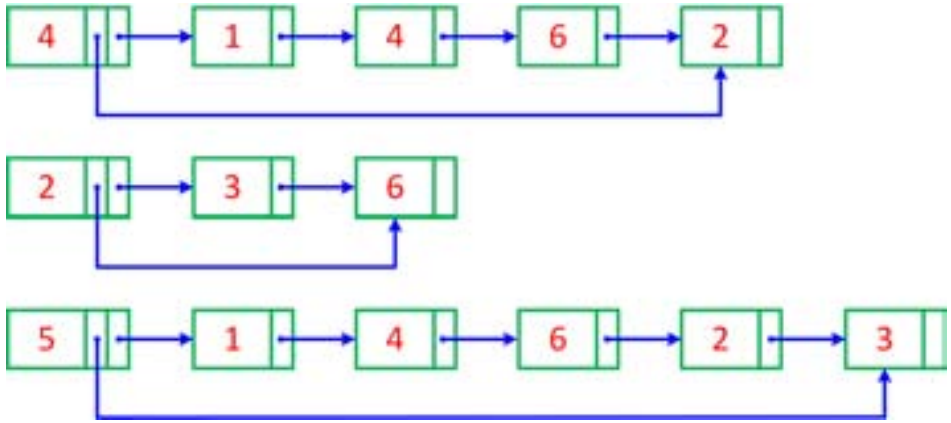
1. 按照权重$w(e_1) \leqslant ... \leqslant w(e_{|E|})$的非递增顺序对集合$E$排序;
2. 判断$find(a)$是否等于$find(b)$，如果不相等，$union(a,b)$;

效率分析

- 第1步的时间复杂度为$O(|E|\,log\,|E|)$（快速排序）;
- 第2步$find(x)$和$union(a,b)$的复杂度取决于实现方式;
- 快速查找（P253）:$T(find) \in O(1)$,$T(union) \in O(nlogn)$;
- 快速求并（P254）:$T(find) \in O(logn)$,$T(union) \in O(1)$;
- 快速查找下总的效率为$O(|E|\,log\,|E|) + O(m + |V|\,log\,|V|)$;
- 快速求并下总的效率为$O(|E|\,log\,|E|) + O(mlog\,|V| + |V|)$;

具体实现

快速查找



快速求并

主要内容

编码策略

两种编码方式

- 定长编码：对每个字符赋予固定长度的编码；
- 变长编码：不同字符的编码长度不同，希望平均长度最短；

二元前缀码

- 为正确解码，要求任何字符的代码不能作为其他字符的前缀；
- $Q = \{001, 00, 010, 01\}$不是二元前缀码，假设码字分别代表字符：

$$a : 001, \quad b : 00, \quad c : 010, \quad d : 01$$

如果接收到序列0100001，那么可能有两种译码方法：

分解为：$01, 00, 001$，译作：d, b, a

分解为：$010, 00, 01$，译作：c, b, d

最优前缀码

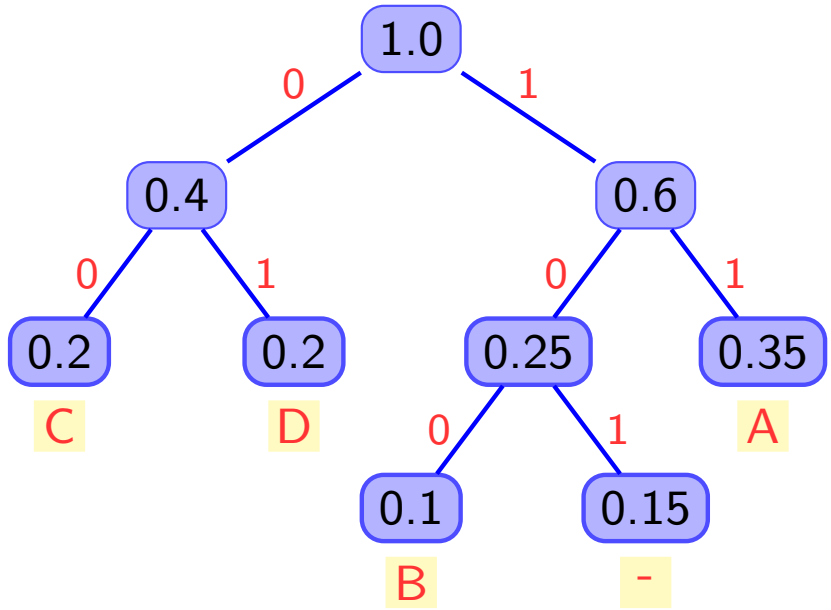
最优二元前缀码

平均使用二进制位数B最少的前缀码，$B = \sum_{i=1}^{n} p(x_i)f(x_i)$，$f(x_i)$，其中$f(x_i)$表示$x_i$的编码长度，$p(x_i)$表示$x_i$出现的概率)。

例子

字符	出现概率	代码字
A	0.35	11
B	0.1	100
C	0.2	00
D	0.2	01
-	0.15	101

每个字符平均长度$B = 2.25$



如何构造最优二元前缀码的二叉树？

哈夫曼算法

算法思想

构造一棵树，将较短编码（树较浅）配给高频字符，较长编码（树较深）配给低频字符。

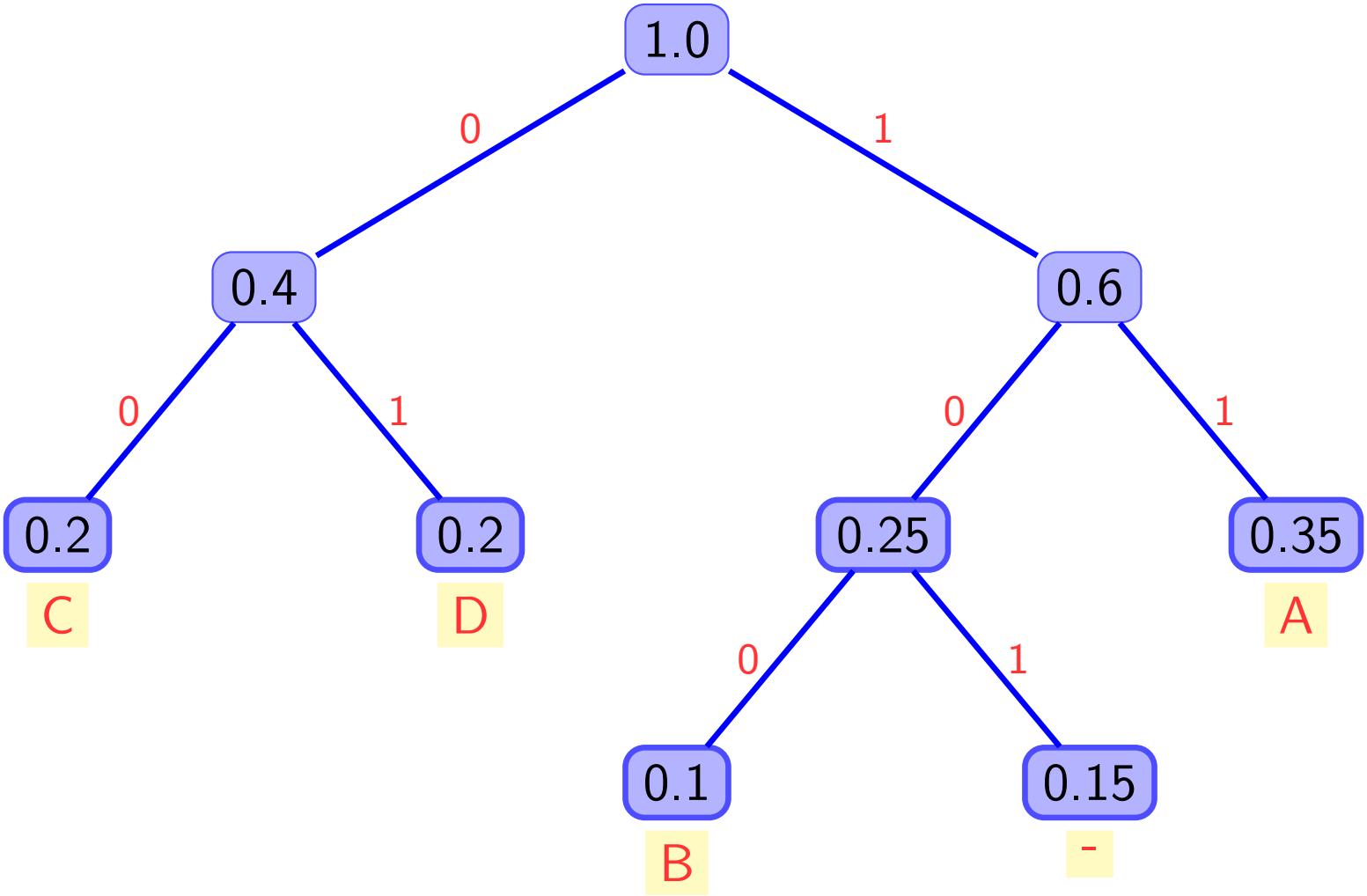
算法流程

1. 初始化n个只包含一个节点的树，每个树的权重为节点的概率；
2. 找到两颗权重最小的树，把它们作为新树的左右子树，新树的权重为左右子树的权重之和；
3. 重复步骤2，直到只剩一棵单独的树；

注

用哈夫曼算法构造的树称为哈夫曼树，所得到的编码称为哈夫曼编码。

算法过程

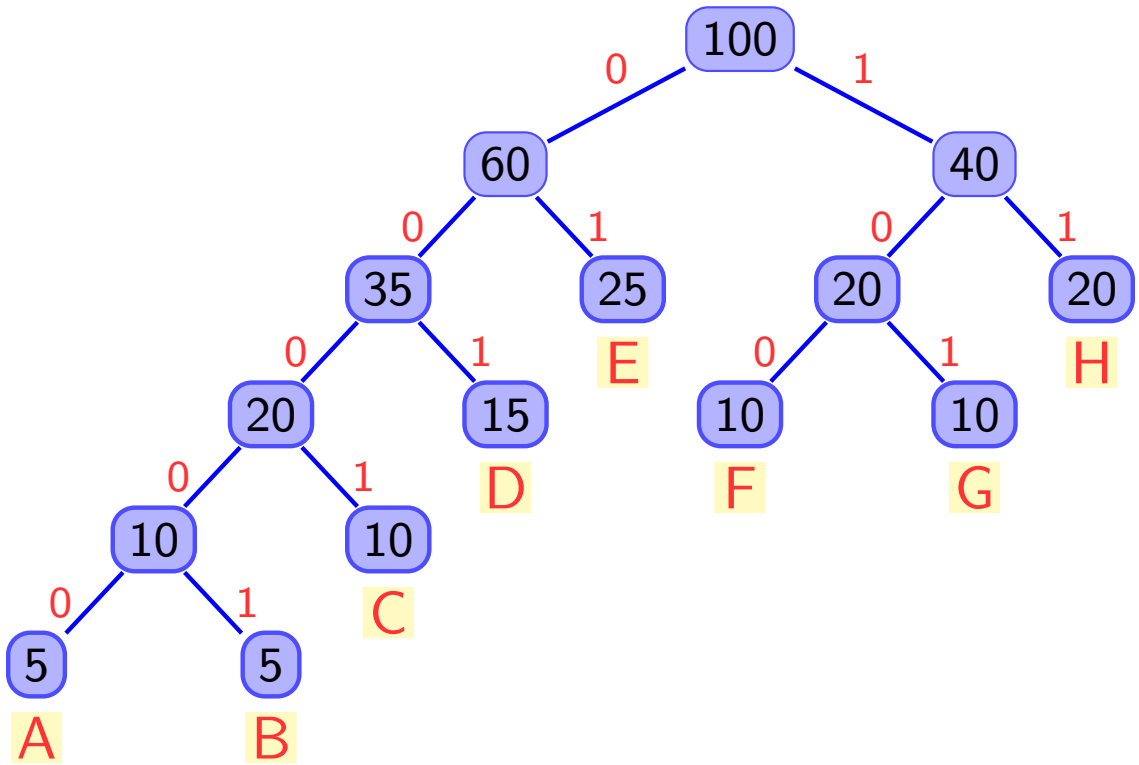


算法练习

字符及出现概率

字符	A	B	C	D	E	F	G	H
概率（%）	5	5	10	15	25	10	10	20

哈夫曼树



伪代码

构造最优二元前缀码的二叉树的哈夫曼算法

1: **function** Haffman(C) //$C = \{x_1, x_2 ... x_n\}$为字符集，每个字符频率为f_i
2: $n \leftarrow |C|$;
3: $Q \leftarrow$ **sort**(C) ; //按照频率递增构成队列Q
4: **for** $i \leftarrow 1 \rightarrow n-1$ **do**
5: $z \leftarrow AllocateNode()$; //生成节点$z$
6: $z.left \leftarrow Q$中最小元;
7: $z.right \leftarrow Q$中次小元;
8: $f(z) \leftarrow f(x) + f(y)$;
9: **Insert**(Q, z); //将z插入Q
10: **end for**
11: **return** Q
12: **end function**

效率分析

主要步骤

1: **function** Haffman(C)
2: $Q \leftarrow$ **sort**(C) ; //按照频率递增构成队列Q
3: **for** $i \leftarrow 1 \rightarrow n-1$ **do**
4: 构造节点z;
5: **Insert**(Q, z); //将z插入Q
6: **end for**
7: **end function**

效率

- 行2排序时间复杂度为$O(nlogn)$;
- 行3执行循环$n-1$次，行5插入操作的复杂度为$O(logn)$;
- 算法的时间复杂度为$O(nlogn)$;

主要内容

P和NP问题

判定问题（只有是否两种输出）

- 图着色问题：最少需要几种颜色才能使得任意两个相邻顶点不同。
- 判定问题：能否可以用不超多m种颜色对图的顶点着色。

P问题

一类能够用确定性的算法在多项式时间内求解的判定问题（算法的最差时间效率属于$O(p(n))$）。

NP问题

一类可以用不确定多项式算法求解的判定问题。

注意

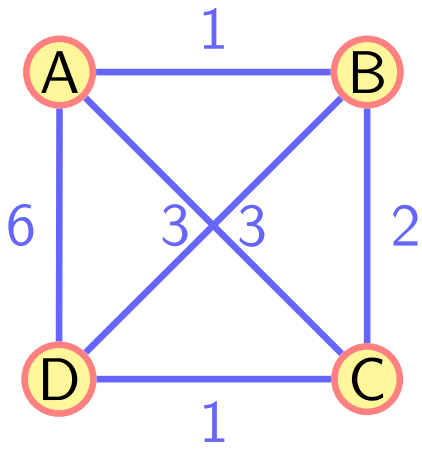
有很多问题，还没有找到它们的多项式算法，也无法证明这样的算法不存在：旅行商问题，背包问题，图着色问题，整数线性规划问题等。

最近邻居算法

算法过程

1. 任意选择一个城市作为开始；
2. 访问和最近一次访问的城市k最接近的未访问城市，如此重复。

算法实例



- 算法解为$A-B-C-D-A$，长度为10
- 最优解为$A-B-D-C-A$，长度为8

多片段启发算法

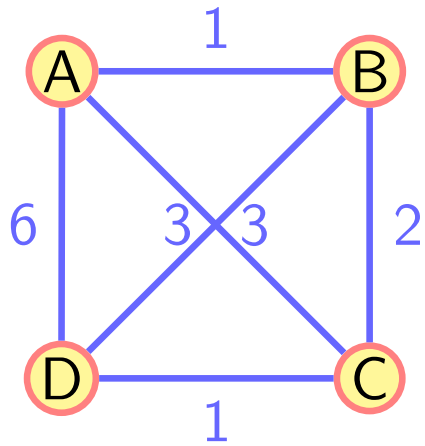
算法思路

TSP可以抽象为求一个给定加权完全图的最小权重边集合，使得所有顶点的连通度都为2（类似于Kruskal算法，不断加入最短的边）。

算法过程

1. 将边按照权重的升序排列；
2. 逐步将排序列表中的下一条边加入旅途边集合，保证本次加入不会使得某个顶点的连通度为3，也不会产生长度小于n的回路，否则忽略该边；
3. 返回旅途边集合；

算法实例



解：$\{(a,b),(c,d),(b,c),(a,d)\}$

绕树两周算法

算法思想

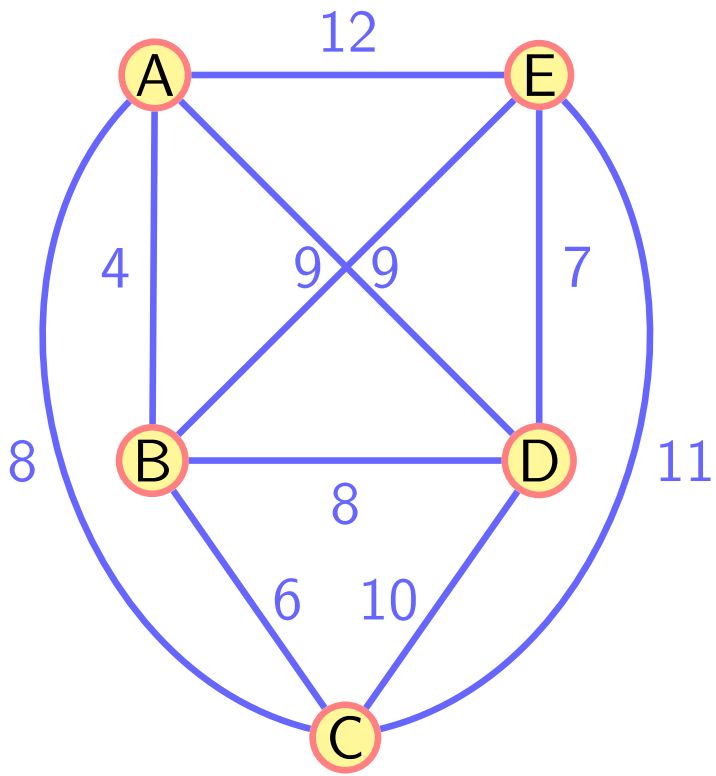
哈密顿回路中去掉一条边就能得到一棵生成树，可以考虑先构造一棵最小生成树，在此基础上构造一条近似最短路径。

算法流程

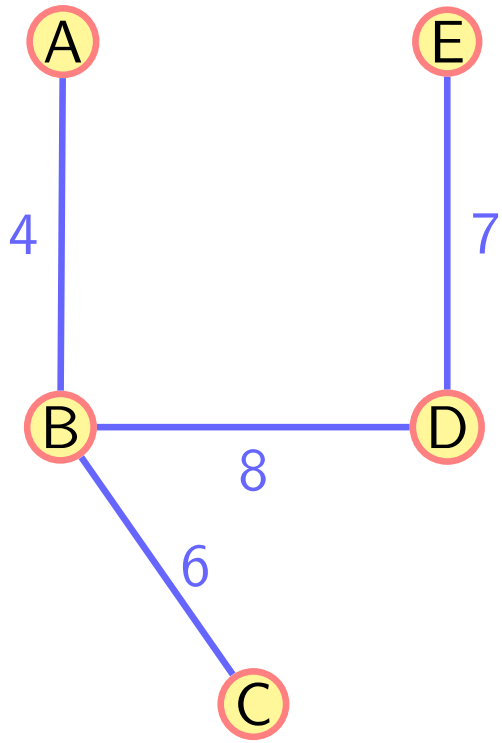
1. 对一个TSP实例，构造它相应图的最小生成树；
2. 从一个任意顶点开始，绕着这棵最小生成树散步一周，并记录下经过的顶点；
3. 扫描得到的顶点列表，消去所有中间路径中重复出现的顶点，列表中余下的顶点构成哈密顿回路；

算法实例

图

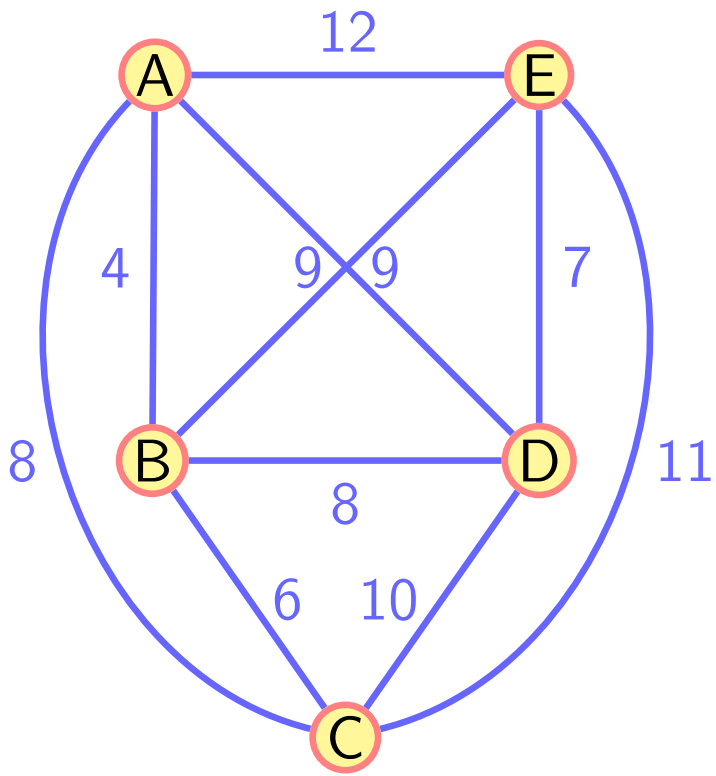


求解过程：最小生成树

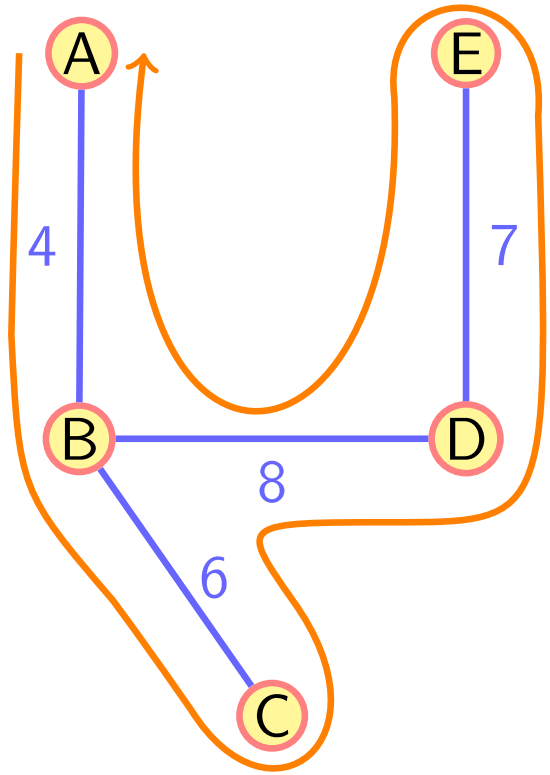


算法实例

图

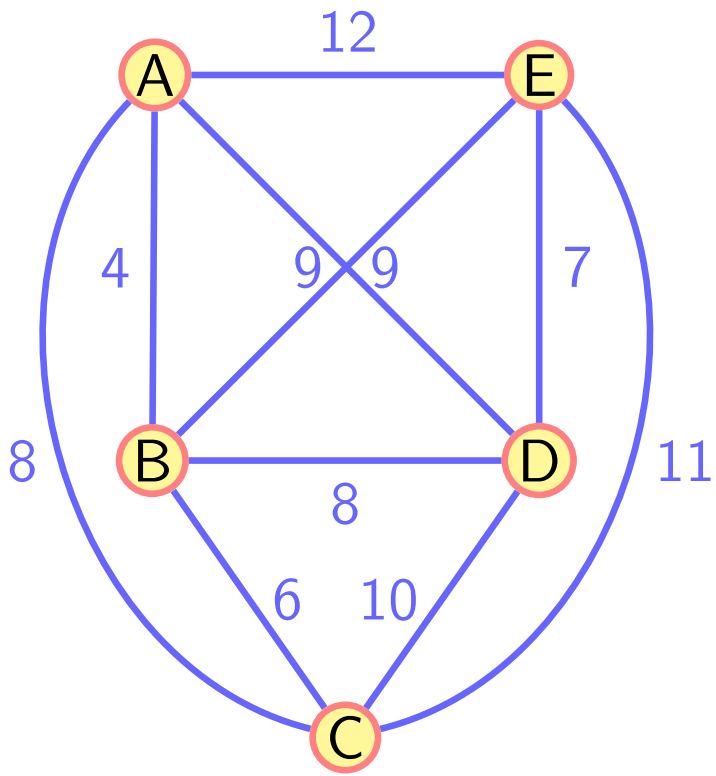


求解过程：绕树一周

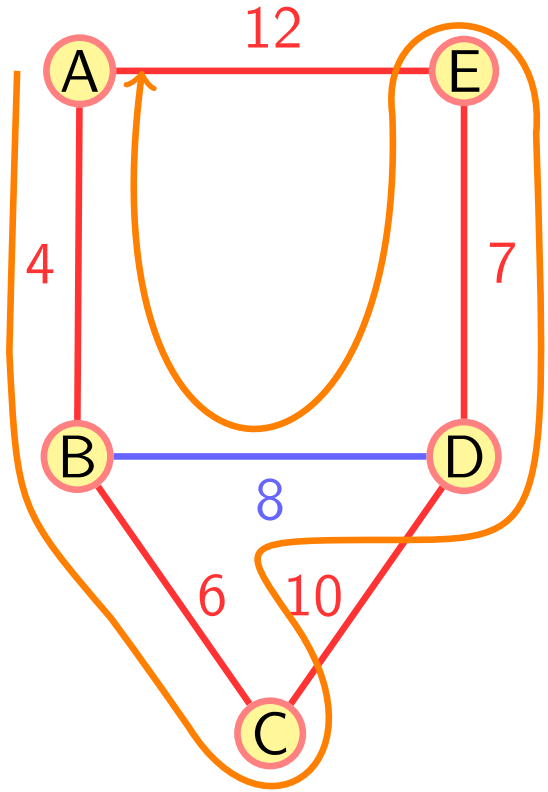


算法实例

图

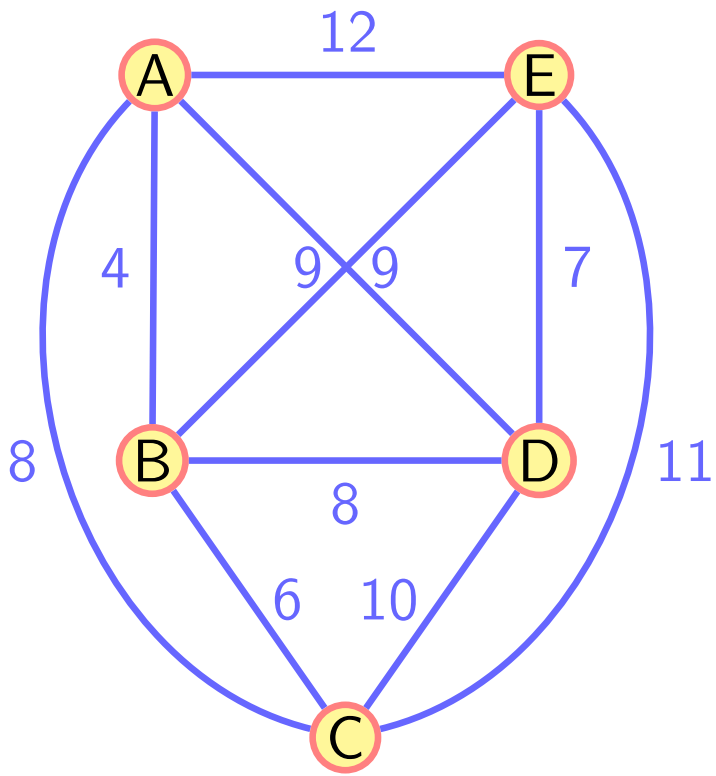


求解过程：消去重复点

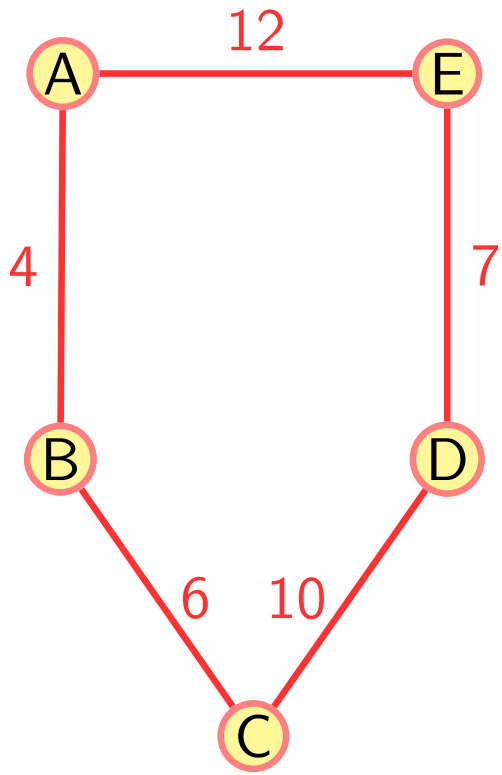


算法实例

图



求解过程：哈密顿回路



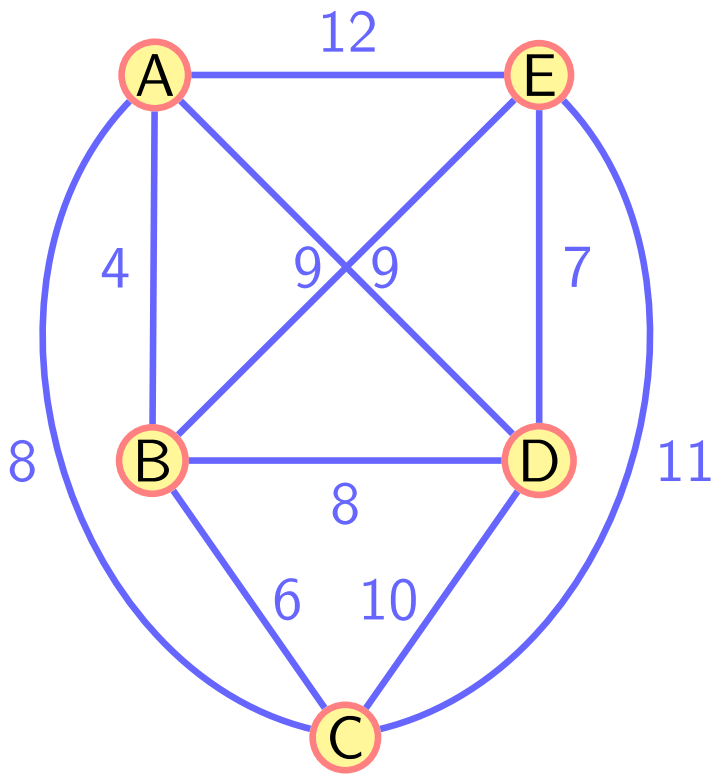
Christofides算法

算法流程

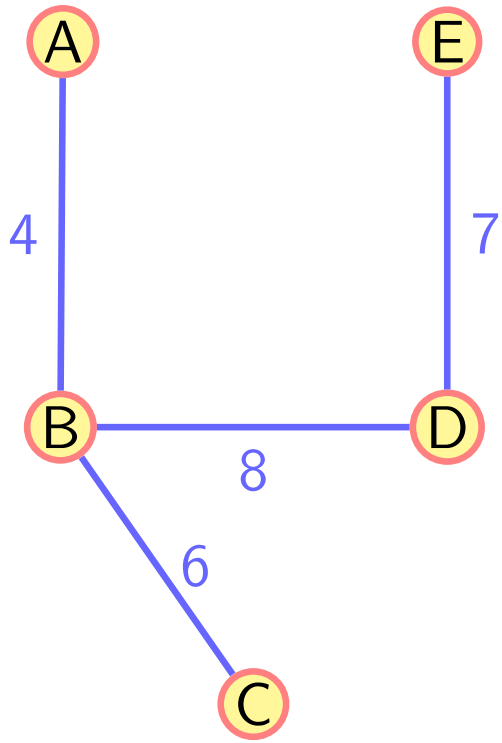
1. 对一个TSP实例，构造它相应图的最小生成树；
2. 找出生成树中所有连通度为奇数的顶点（总为偶数）；
3. 计算得到的顶点的可能的最小权重匹配边（如有a, b, c, d四个顶点，可能的匹配有(a, b)和(c, d)，(a, c)和(b, d)，(a, d)和(b, c)），加入图中；
4. 从一个任意顶点开始，绕着这棵最小生成树散步一周，并记录下经过的顶点；
5. 扫描得到的顶点列表，消去所有中间路径中重复出现的顶点，列表中余下的顶点构成哈密顿回路；

算法实例

图

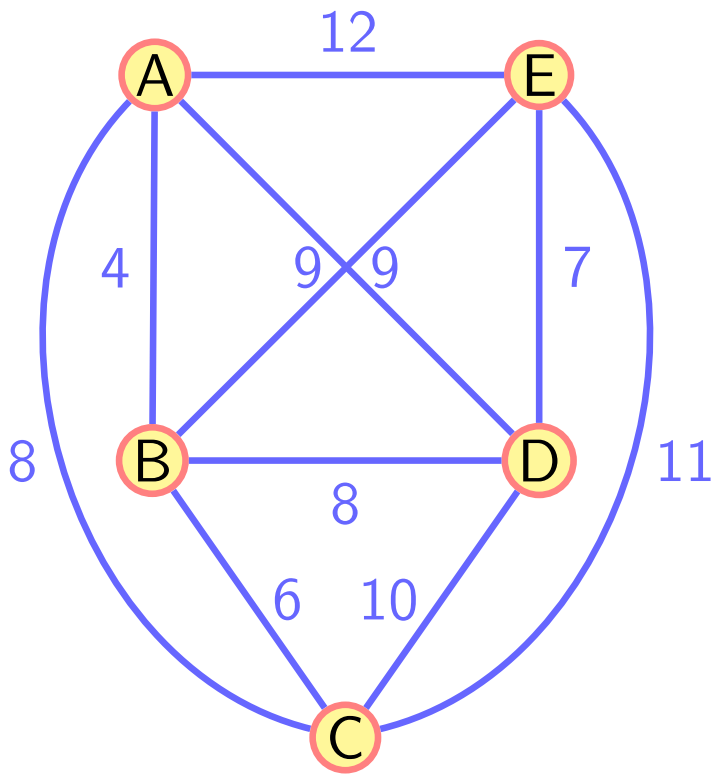


求解过程：最小生成树

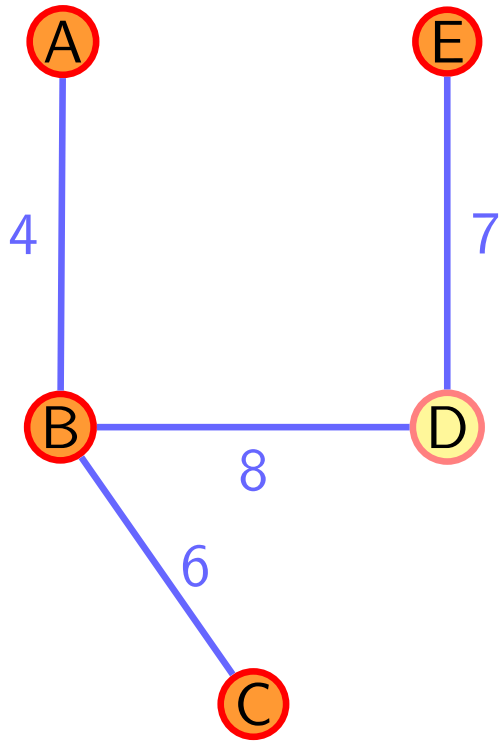


算法实例

图

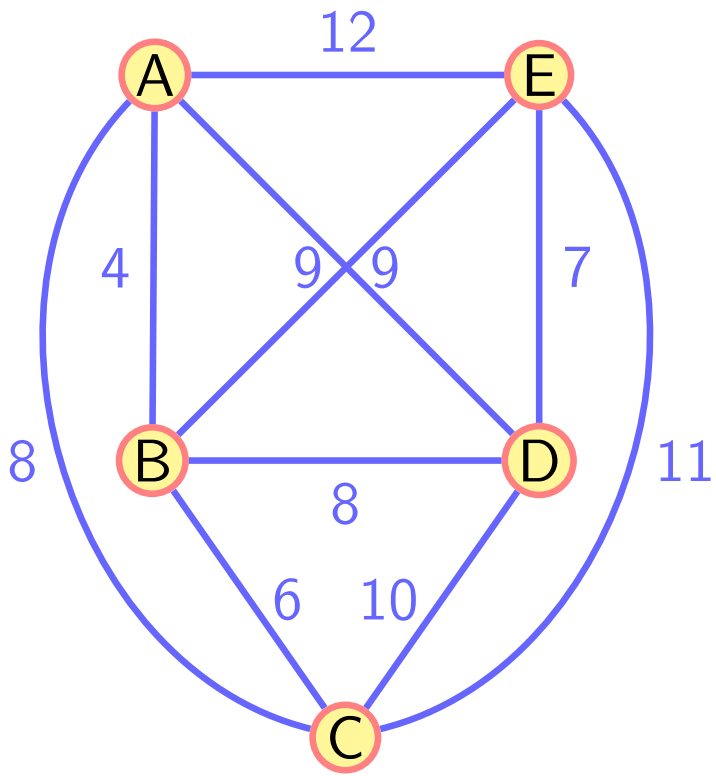


求解过程：连通度为奇数的顶点

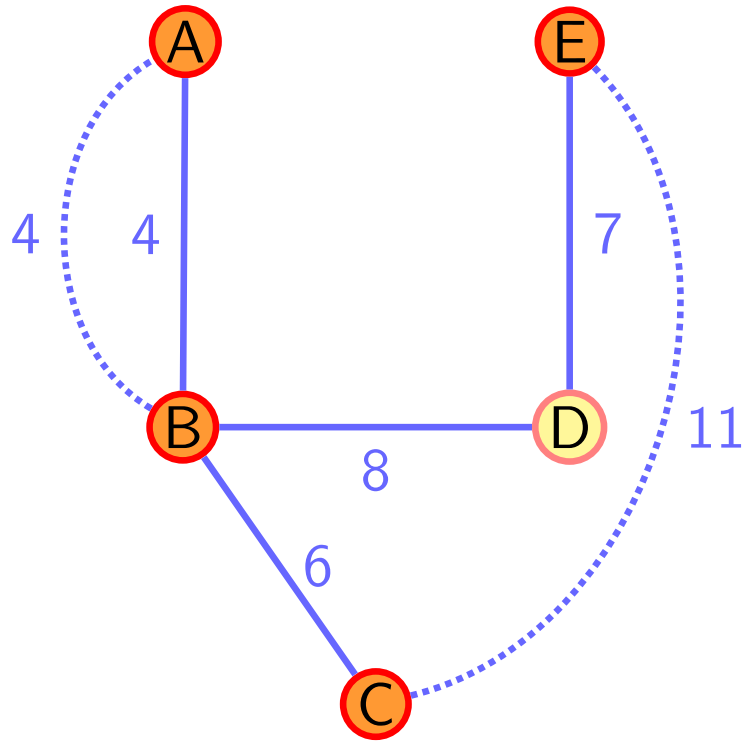


算法实例

图

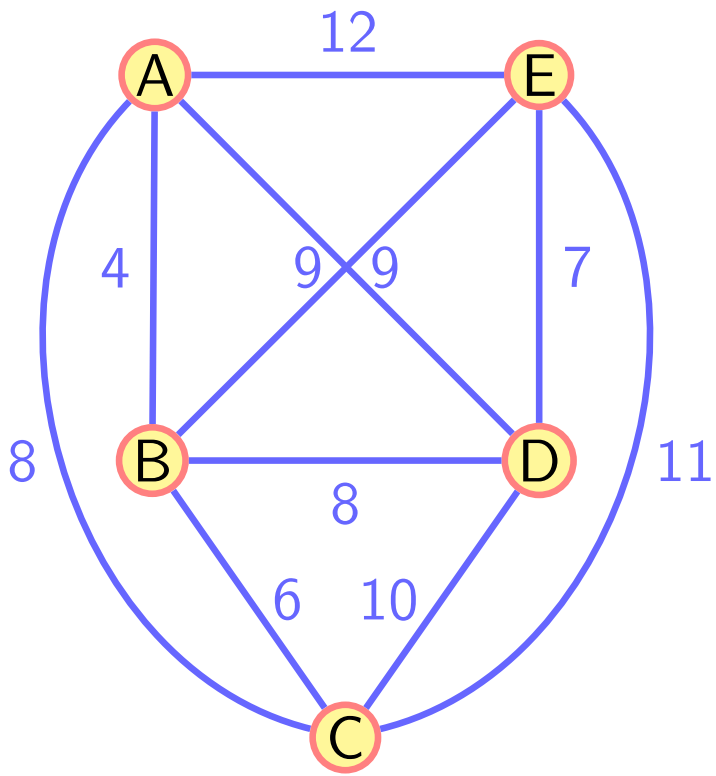


求解过程：最小权重匹配边

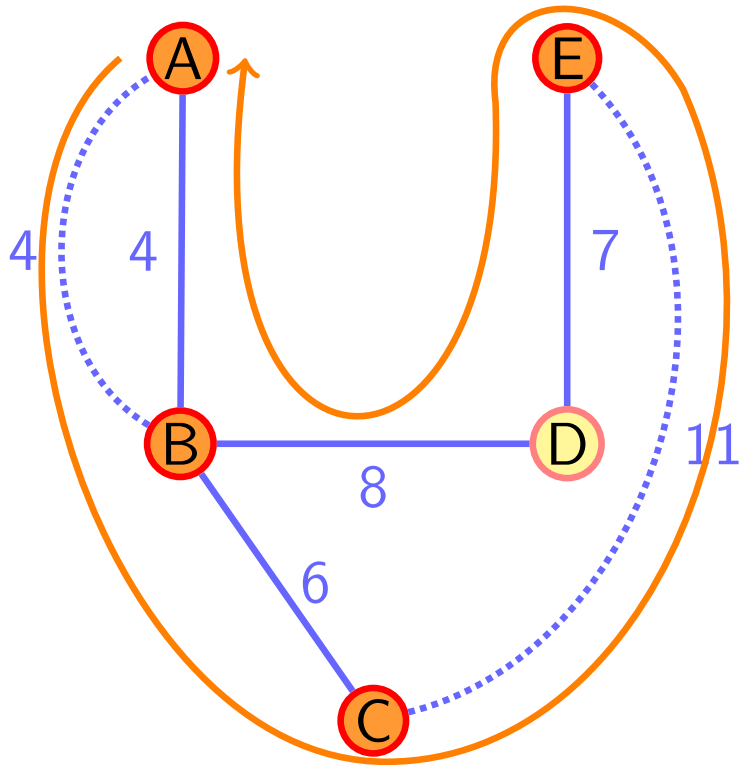


算法实例

图

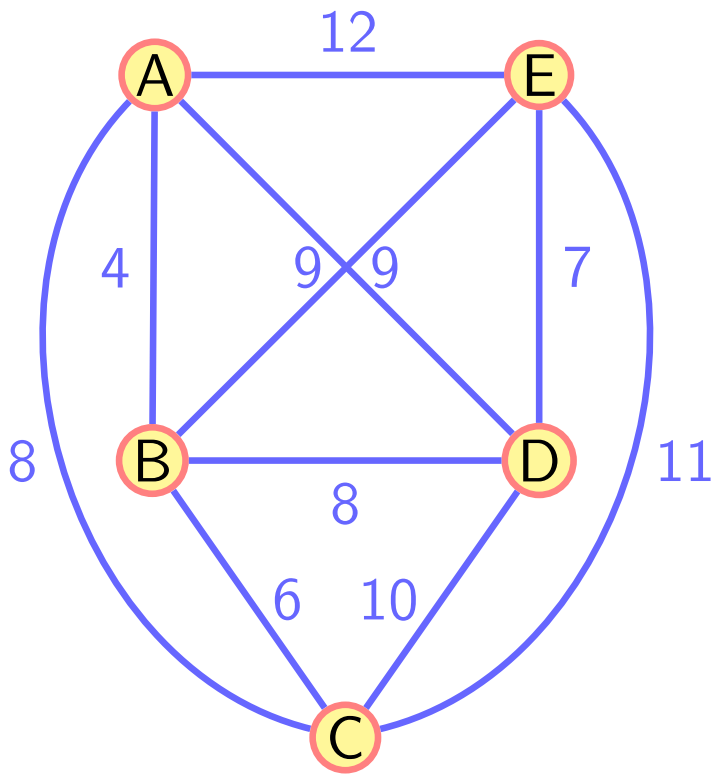


求解过程：绕树一周

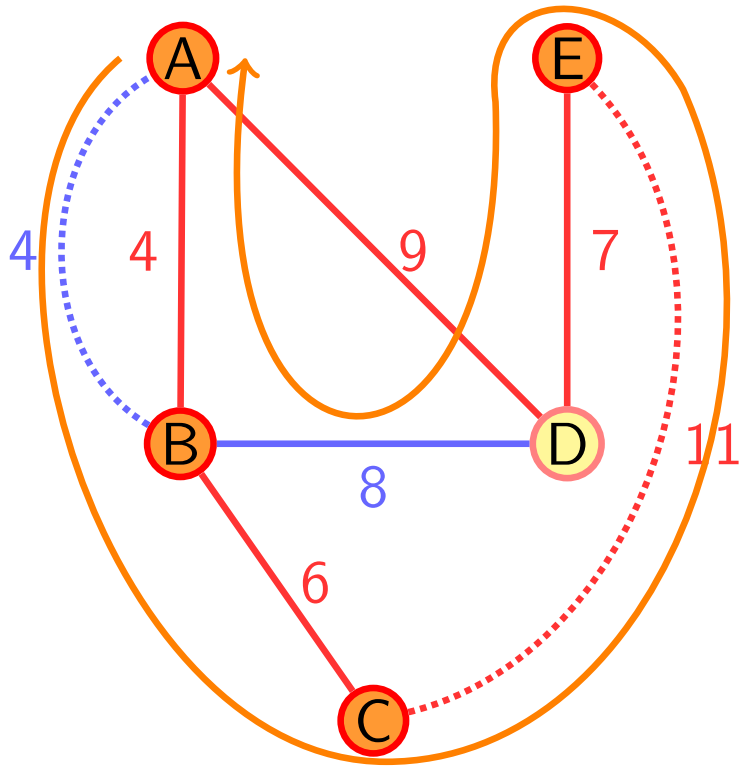


算法实例

图

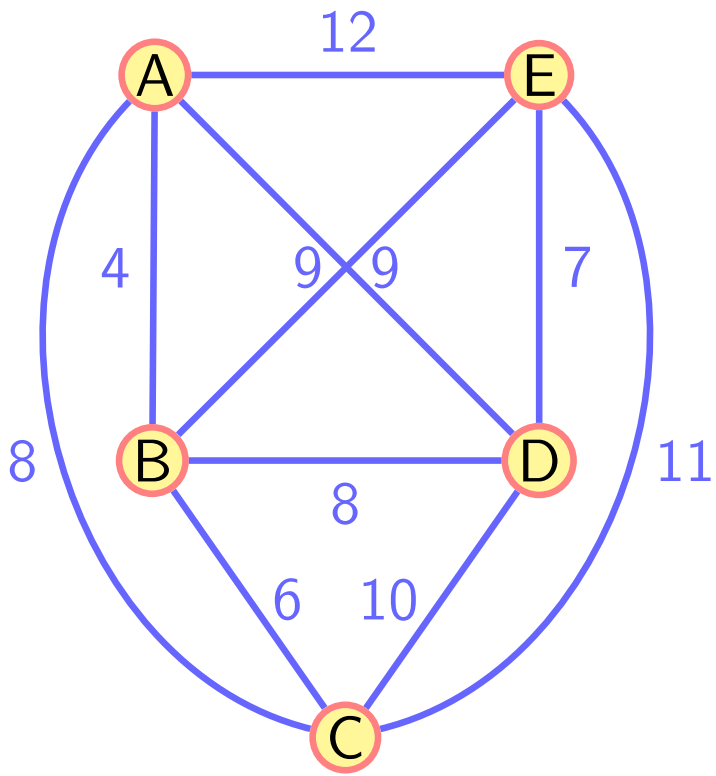


求解过程：消去重复点

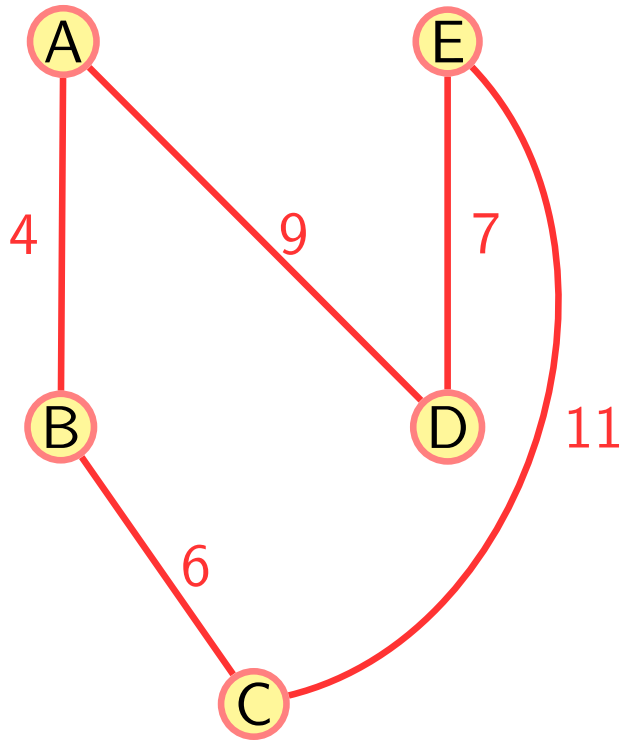


算法实例

图



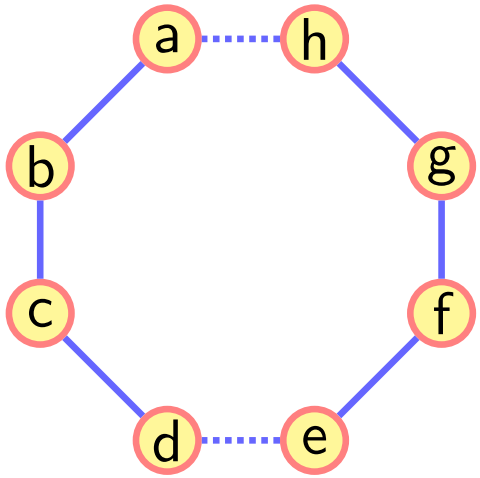
求解过程：哈密顿回路

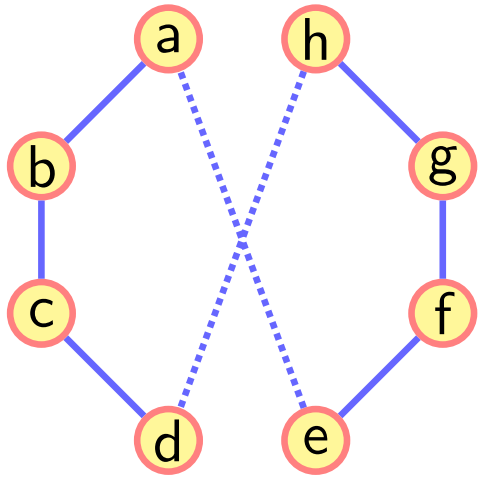


本地查找启发法（Swap-2法）

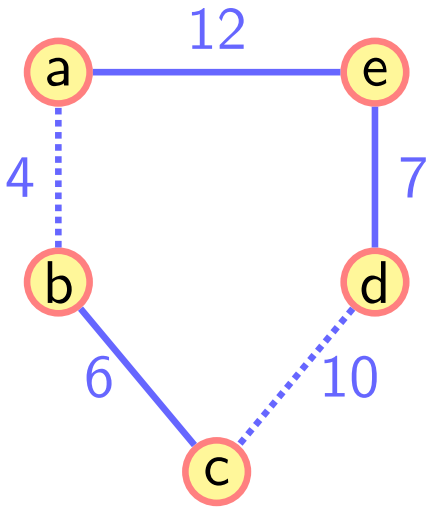
算法思想

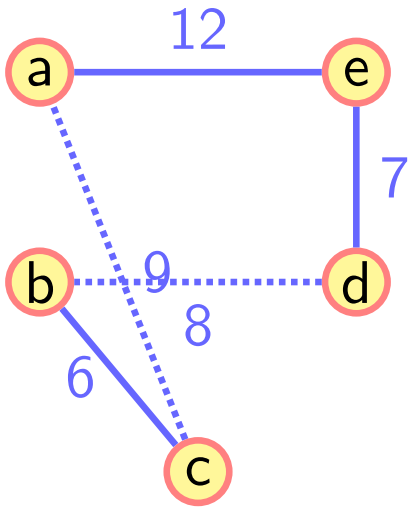
从初始旅途（可随机构造或者用一些简单的方法得到）开始，每次迭代时，把当前旅途中的一些边用其他边代替（交换两个顶点的边），生成一个更短的旅途。



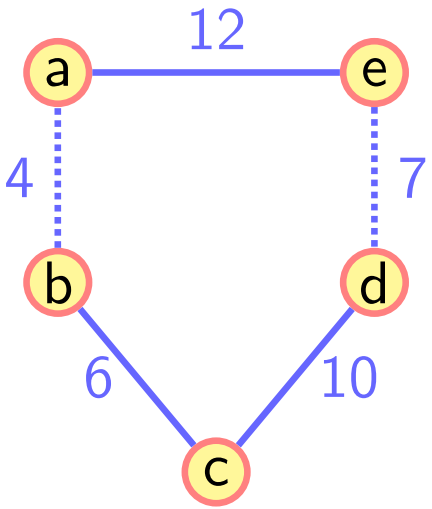


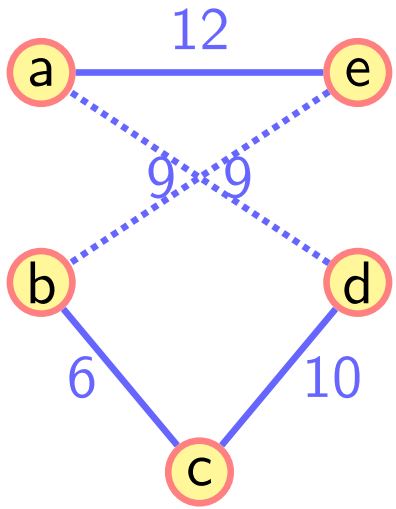
算法示例





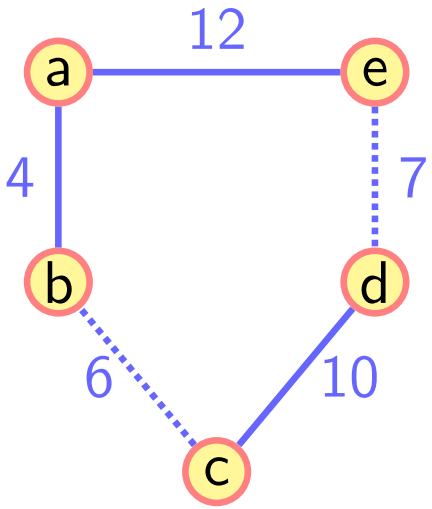
$l < l_{swap}$

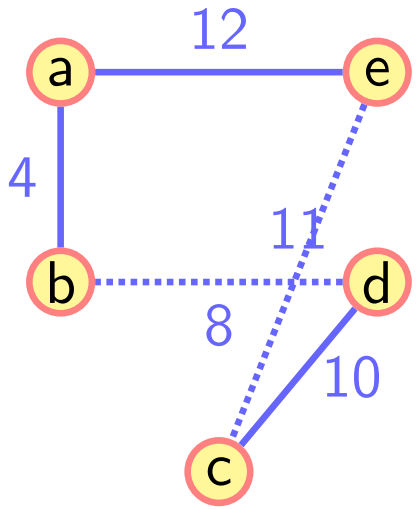




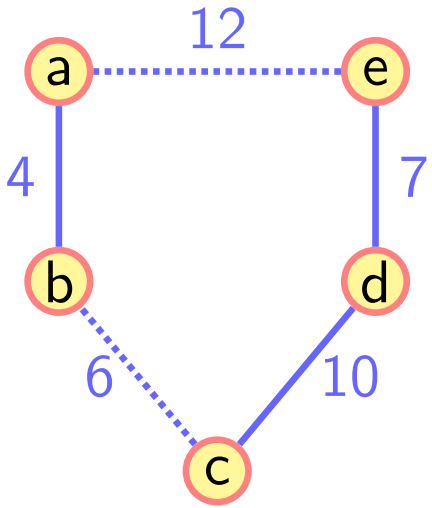
$l < l_{swap}$

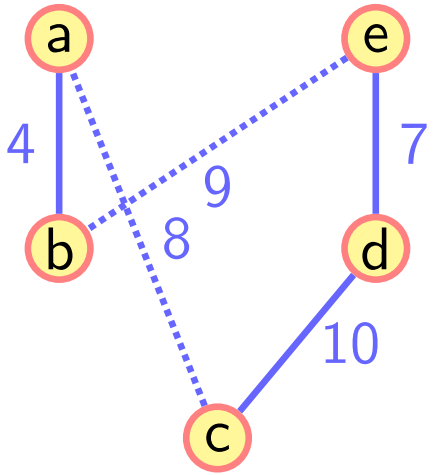
算法示例





$l < l_{swap}$





$l > l_{swap}$

算法对比

表: 包含10000个城市的示例，各种算法旅途质量和运行时间

算法	超过Held-Karp下界的/%	运行时间/秒
最近邻居法	24.79	0.28
多片段启发法	16.42	0.20
Christofides	9.81	1.04
Swap-2	4.70	1.41
Swap-3	2.88	1.50
Lin-Kernighan	2.00	2.06

- 一般来说，Held-Karp下界和最优旅途的长度非常接近；
- 一般来说，Christofides算法要明显好于绕树两周算法；
- Lin-Kernighan算法是一种变选算法，可以看作是Swap-3之后跟着一系列的Swap-2的操作；

贪婪算法

算法过程

1. 对于给定的物品，计算其价值重量比，并依此排序；
2. 如果有序列表中的当前物品可以装入背包，放入背包，否则跳过该物品，如此重复；

例子：承重量为10

物品信息		
物品	重量	价值
A	7	42
B	3	12
C	4	40
D	5	25

排序并装入背包			
物品	重量	价值	价值/重量
C	4	40	10
A	7	42	6
D	5	25	5
B	3	12	4

Sahni方法

算法思想

生成所有小于等于k个物品的子集，并像贪婪算法那样向子集中添加剩余物品（价值重量比最大的物品），返回最大价值的物品子集。

例子：承重量为10，$k = 2$

子集				剩余物品				总价值
物品	重量	价值	价值/重量	物品	重量	价值	价值/重量	
∅				A	4	40	10	69
				B	7	42	6	
				C	5	25	5	
				D	1	4	4	

Sahni方法

例子：承重量为10，$k = 2$

子集				剩余物品				总价值
物品	重量	价值	价值/重量	物品	重量	价值	价值/重量	
A	4	40	10	B	7	42	6	69
				C	5	25	5	
				D	1	4	4	
B	7	42	6	A	4	40	10	46
				C	5	25	5	
				D	1	4	4	
C	5	25	5	A	4	40	10	69
				B	7	42	6	
				D	1	4	4	
D	1	4	4	A	4	40	10	69
				B	7	42	6	
				C	5	25	5	

Sahni方法

例子：承重量为10，$k = 2$

子集				剩余物品				总价值
物品	重量	价值	价值/重量	物品	重量	价值	价值/重量	
A	4	40	10	C	5	25	5	不可行
B	7	42	6	D	1	4	4	
A	4	40	10	B	7	42	6	69
C	5	25	5	D	1	4	4	
A	4	40	10	B	7	42	6	69
D	1	4	4	C	5	25	5	
B	7	42	6	A	4	40	10	不可行
C	5	25	5	D	1	4	4	
B	7	42	6	A	4	40	10	46
D	1	4	4	C	5	25	5	
D	1	4	4	A	4	40	10	69
C	5	25	5	B	7	42	6	

小结

- 贪心算法通过一系列的步骤来构造问题的解，每一步对目前构造的部分解做一个扩展，直到获得问题的完整解为止；
- $Prim$算法不断地向已建立的子树加入最近的顶点，以构造加权连通图的最小生成树；
- $Kruskal$算法按照边权重大小依次包含进子树，并在保证包含操作不构成回路，从而将多颗子树连通为最小生成树；
- $Prim$和$Kruskal$均是贪心算法，总能产生最优解（最小生成树）；
- 哈夫曼编码是一种最优的二元前缀变长编码的方案，可用于文件压缩，归并等各个方面；
- 贪婪技术可以为旅行商问题和背包问题提供近似解；

作业

- 9.1: 1 $Page249$
- 9.1: 3 $Page249$
- 9.1: 9 $Page249$
- 9.2: 1 $Page255$
- 9.2: 2 $Page256$

Part IV

图算法

阅读: 3.5 9.3 4.2 10.2 12.1 12.2

主要内容

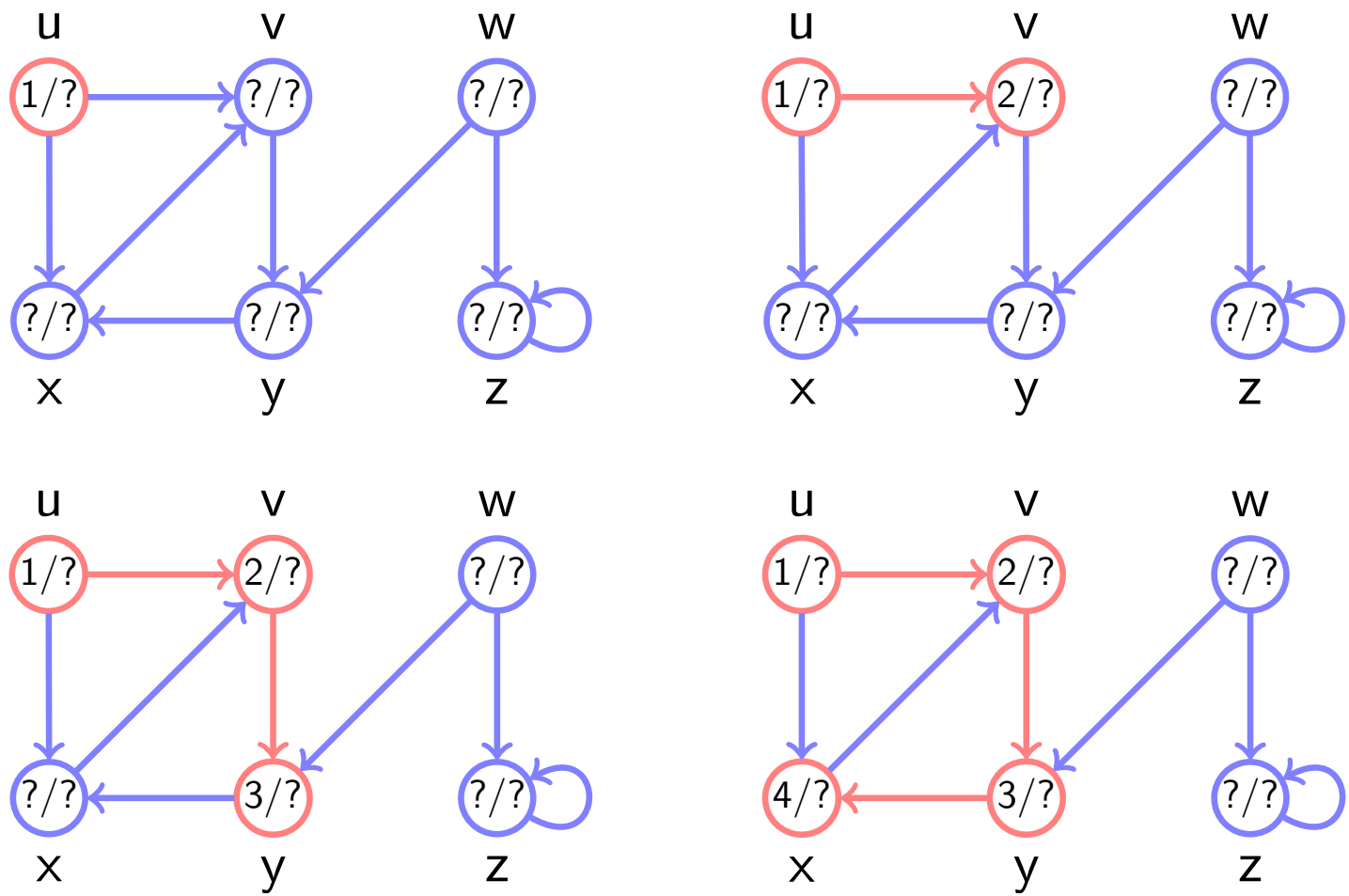
1. 图搜索策略
   - 深度优先查找
   - 广度优先查找
2. 最短路径问题
   - 无权最短路径
   - 单源最短路径
3. 拓扑排序
   - 问题描述
   - 深度优先法
   - 源删除法
4. 最大流问题
   - 流量网络
   - 最短路径增益法
5. 回溯法
   - n皇后问题
   - 哈密顿回路
   - 其他问题
6. 分支界限
   - 分配和背包问题
   - 其他问题

主要内容

深度优先查找

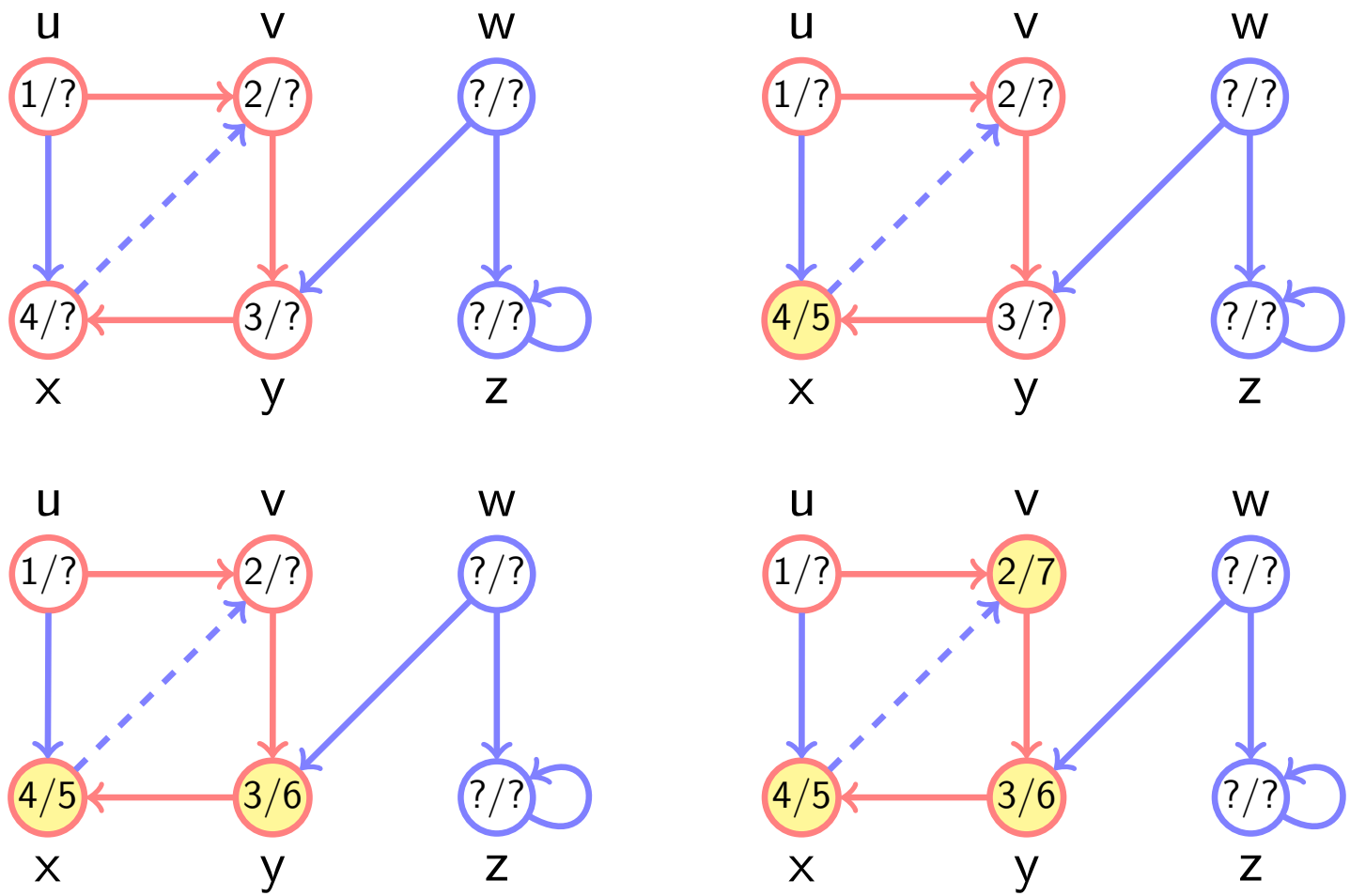
1. 从任意顶点开始访问图的顶点，然后把该顶点标记为已访问；
2. 每次迭代，算法接着处理与当前顶点邻接的未访问顶点（有若干顶点，任意选择一个）；
3. 重复步骤2，直到遇到一个终点（所有邻接顶点都已被访问）；
4. 在该终点上，算法沿着来路后退一条边，重复步骤2和3，并试着继续从那里访问未访问的顶点；
5. 重复步骤4，直到起始顶点变为一个终点。
6. 如果未访问的顶点仍然存在，继续从其中任意顶点开始，重复上述过程；

深度优先查找实例



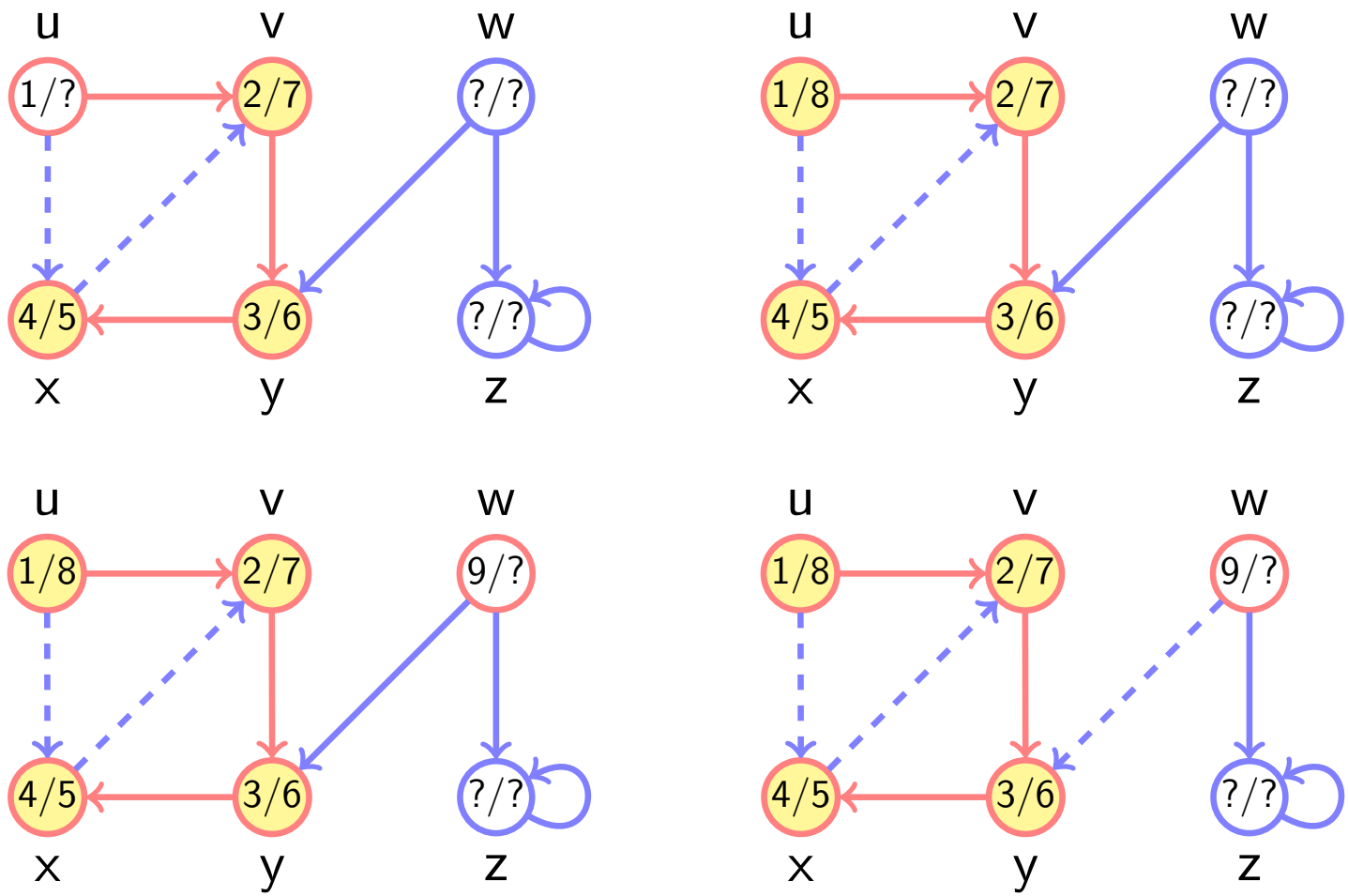
s/t 未发现节点　s/t 发现节点　s/t 完成节点　s:发现时间 t:完成时间

深度优先查找实例



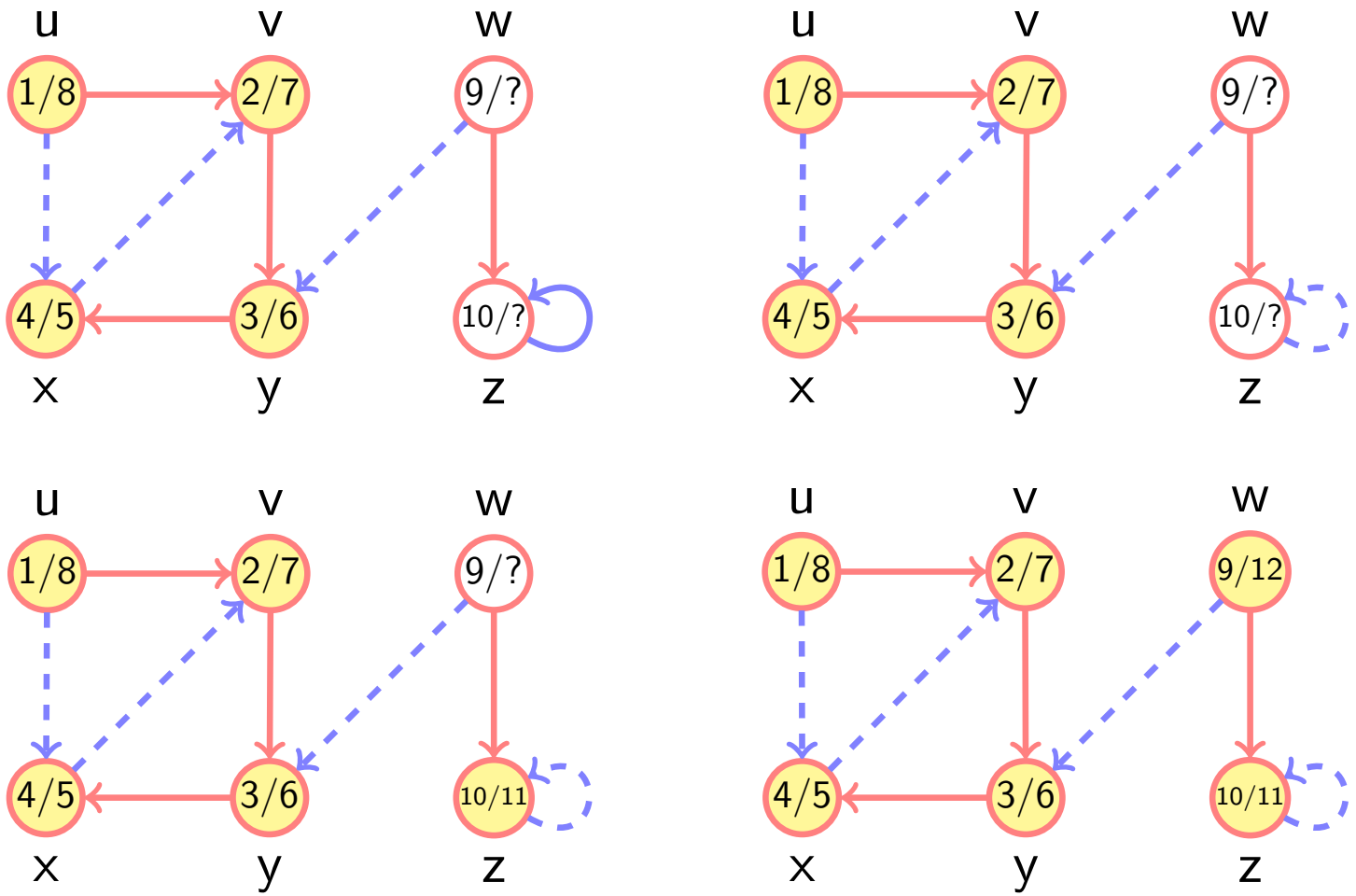
s/t 未发现节点　s/t 发现节点　s/t 完成节点　s:发现时间 t:完成时间

深度优先查找实例



s/t 未发现节点　s/t 发现节点　s/t 完成节点　s:发现时间 t:完成时间

深度优先查找实例



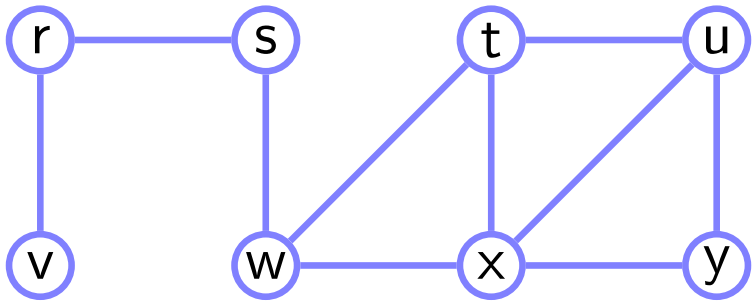
s/t 未发现节点　s/t 发现节点　s/t 完成节点　s:发现时间 t:完成时间

广度优先查找

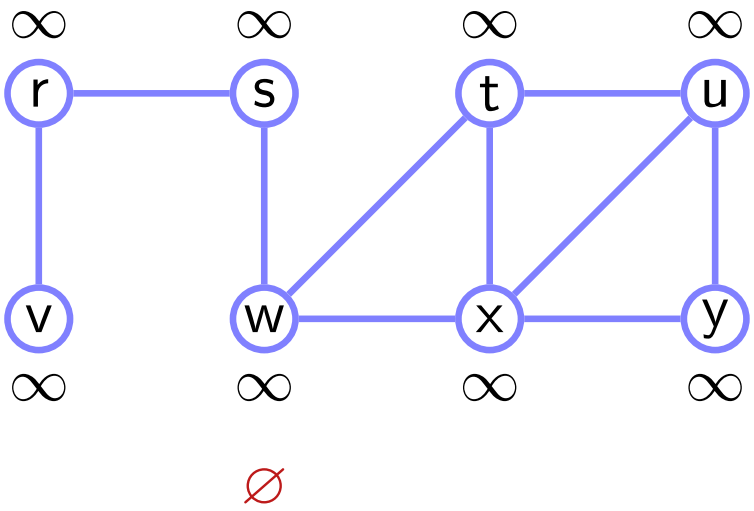
算法思想

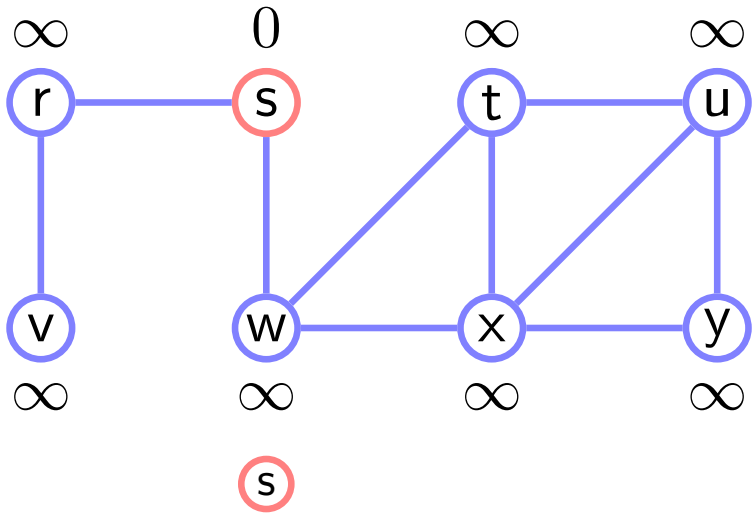
首先访问和初始顶点邻接的顶点，然后是它两条边的所有未访问顶点。以此类推，直到所有与初始顶点连通的顶点都访问为止。如果未访问的顶点仍然存在，继续从其中任意顶点开始，重复此过程；

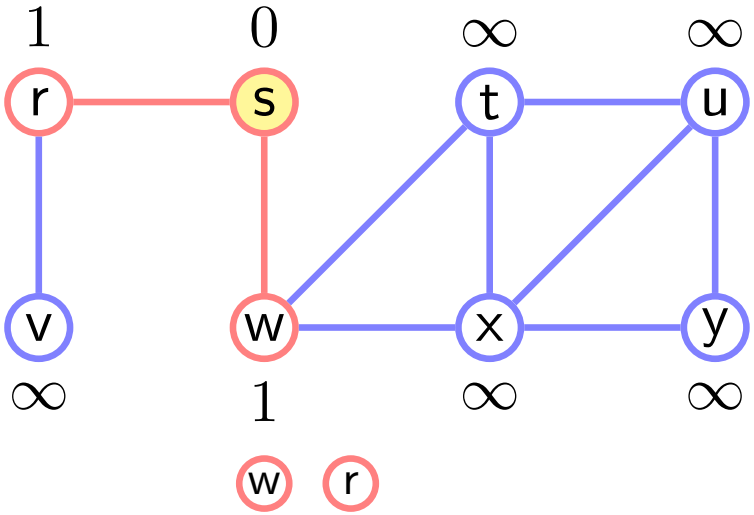
思考：如何利用广度优先搜寻如下的图？

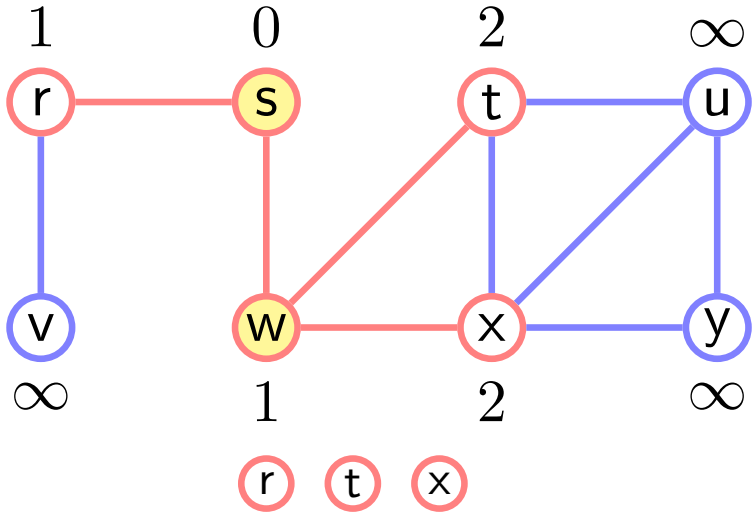


广度优先查找实例

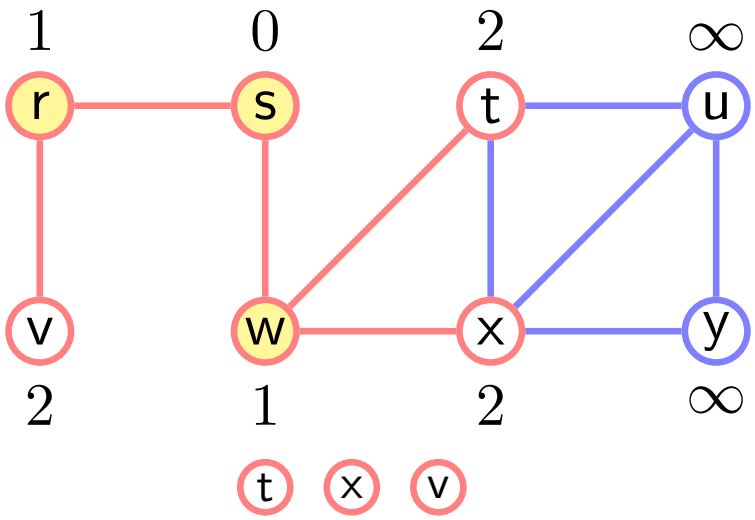


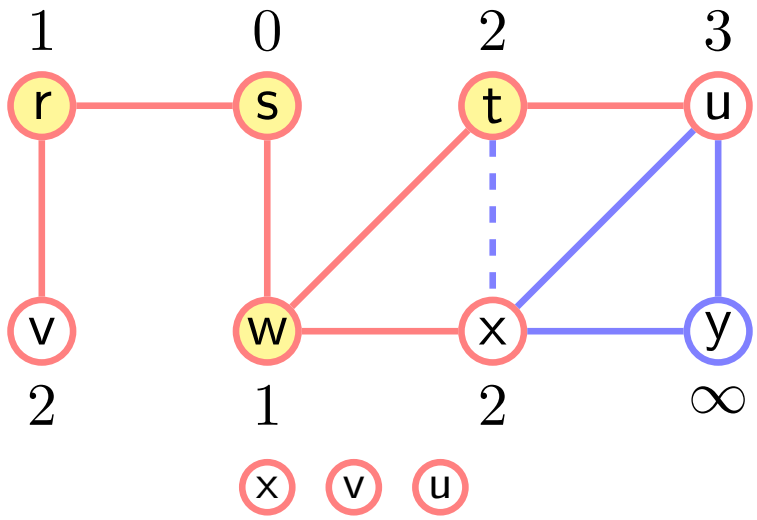


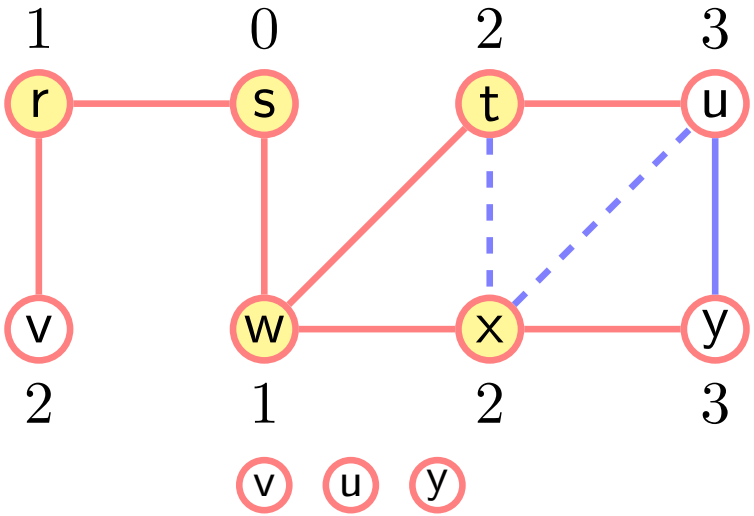


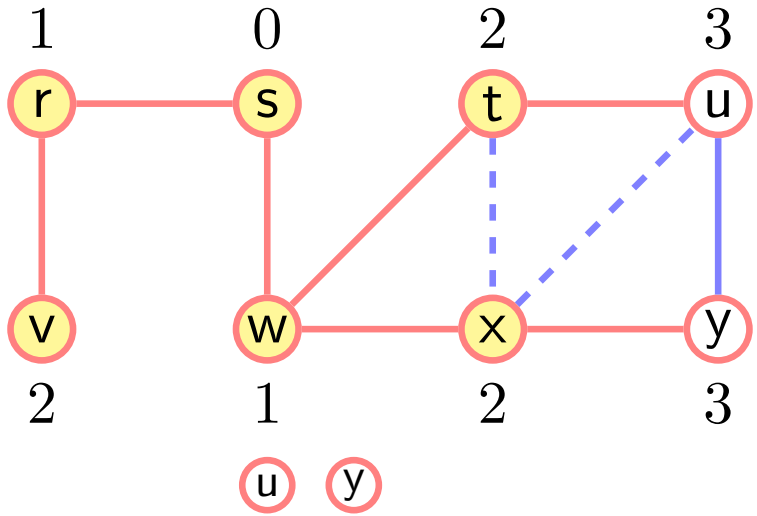


广度优先查找实例

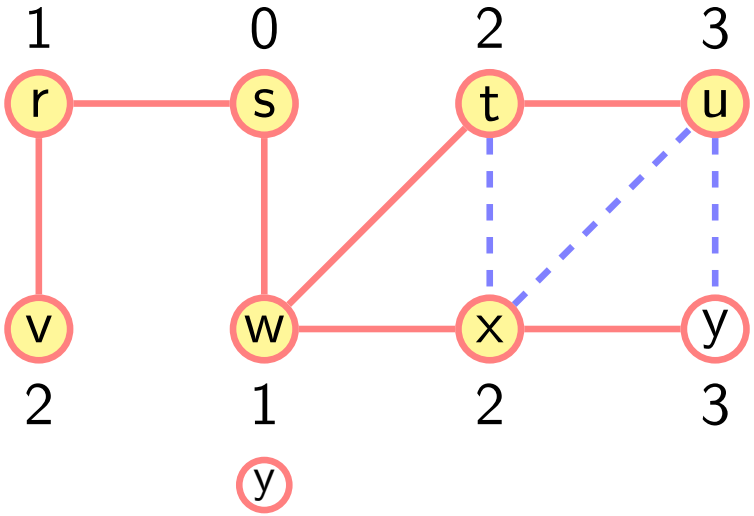

1
0
2
∞
r
s
t
u
v
w
x
y
2
1
2
∞
t
x
v

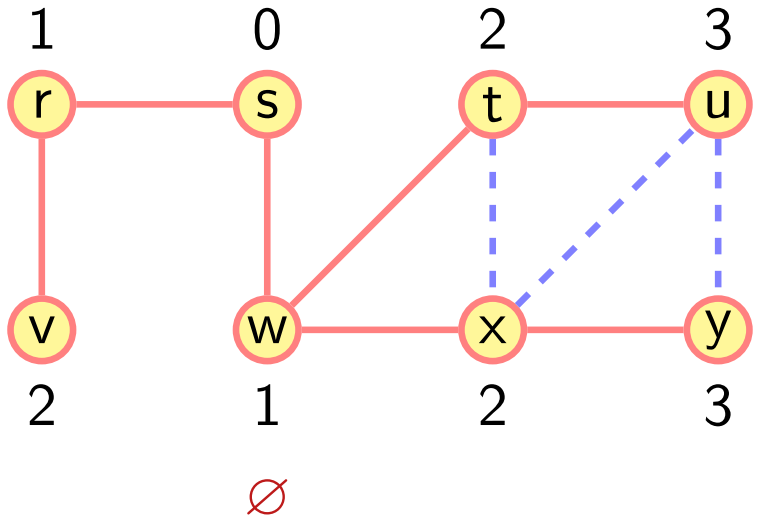

1
0
2
3
r
s
t
u
v
w
x
y
2
1
2
∞
x
v
u


1
0
2
3
r
s
t
u
v
w
x
y
2
1
2
3
v
u
y


1
0
2
3
r
s
t
u
v
w
x
y
2
1
2
3
u
y

广度优先查找实例





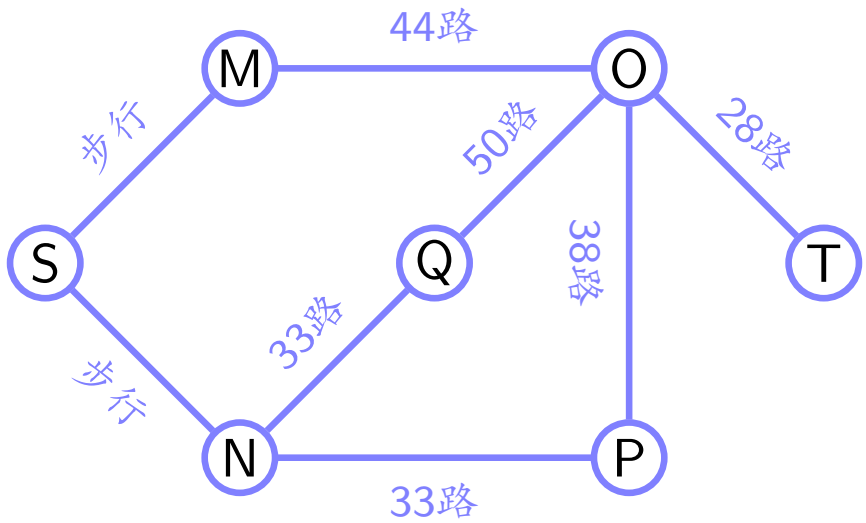
主要内容

无权最短路径

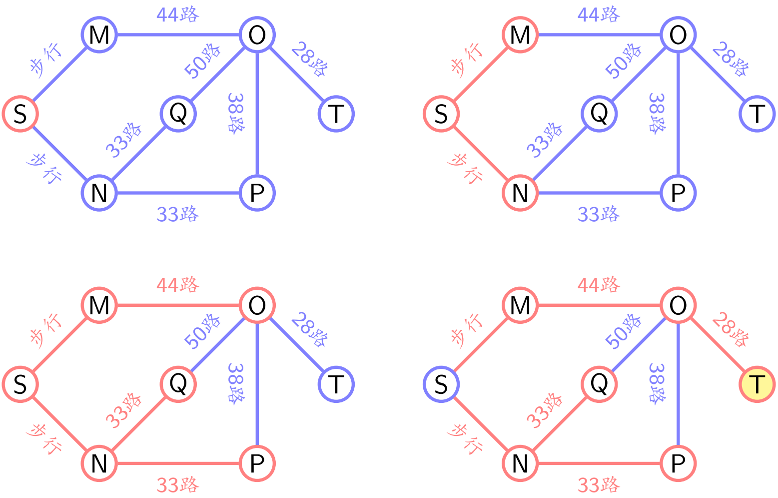
问题描述

对于一个无权图G，希望找到某个顶点到其他顶点的最短路径，也就是说只对包含在路径中的边数感兴趣。

实例：寻找从“双子峰”（S）到“金门大桥”（T）的最少换乘路径



求解过程–广度优先搜索



单源最短路径问题

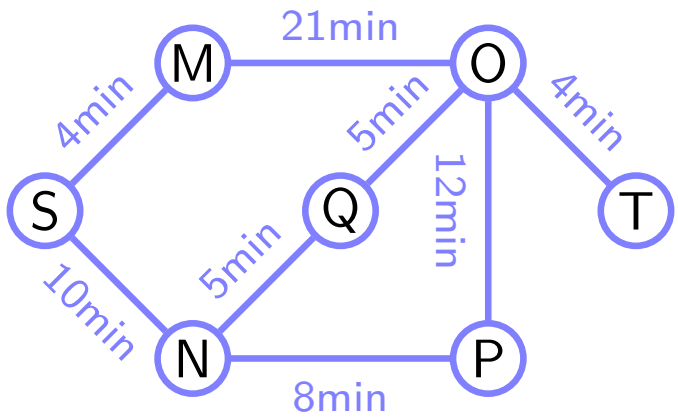
赋权路径长

对于加权图$G=(V,E)$，假设与每条边(v_i,v_j)相连的代价为$c_{i,j}$，定义一条路径$v_1,v_2,v_3,\ldots,v_N$的赋权路径长为$\sum_{i=1}^{N-1}c_{i,i+1}$。

单源最短路径问题

找出从一个特定顶点S到G中每一个其他顶点的最短赋权路径。

思考：寻找从“双子峰”（S）到“金门大桥”（T）的耗时最短路径

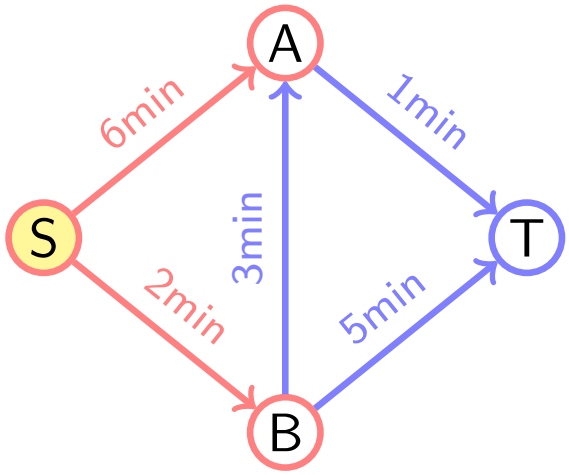


Dijkstra算法–简单的例子

算法步骤

1. 找出“最便宜”的节点
2. 更新该节点的邻居的开销
3. 重复1和2直到访问所有节点
4. 计算最终路径

例子



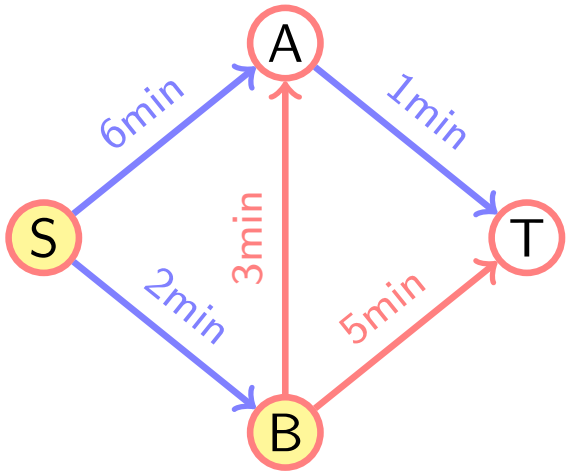
节点	耗时
A	6
B	2
T	∞

Dijkstra算法–简单的例子

算法步骤

1. 找出“最便宜”的节点
2. 更新该节点的邻居的开销
3. 重复1和2直到访问所有节点
4. 计算最终路径

例子



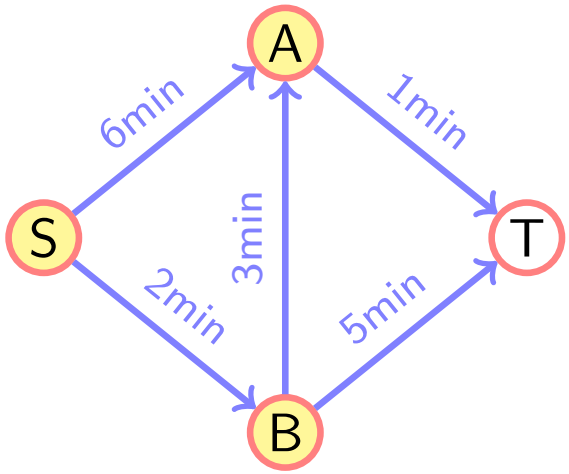
节点	耗时
A	$6 \rightarrow 5$
B	2
T	$\infty \rightarrow 7$

Dijkstra算法–简单的例子

算法步骤

1. 找出“最便宜”的节点
2. 更新该节点的邻居的开销
3. 重复1和2直到访问所有节点
4. 计算最终路径

例子



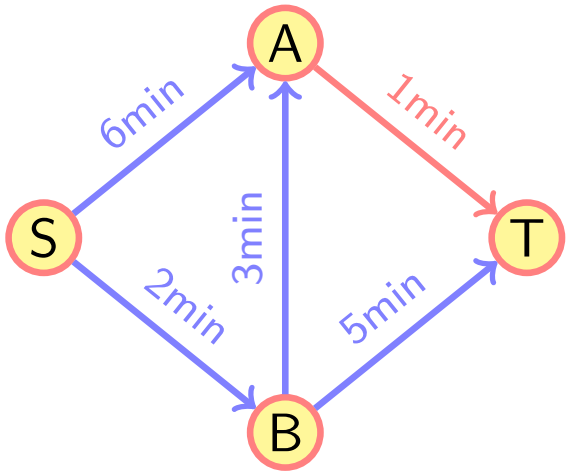
节点	耗时
A	5
B	2
T	7

Dijkstra算法–简单的例子

算法步骤

1. 找出“最便宜”的节点
2. 更新该节点的邻居的开销
3. 重复1和2直到访问所有节点
4. 计算最终路径

例子



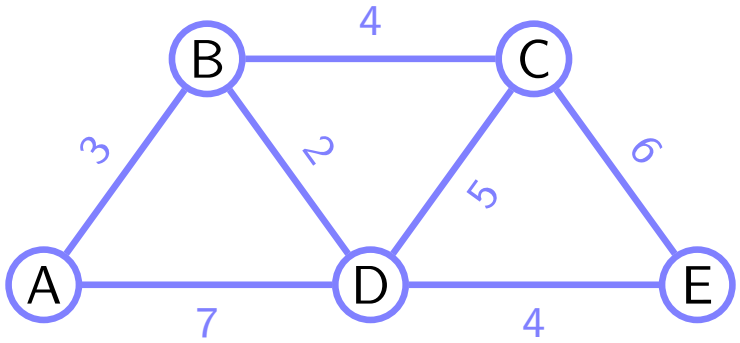
节点	耗时
A	5
B	2
T	7 → 6

Dijkstra算法详细过程

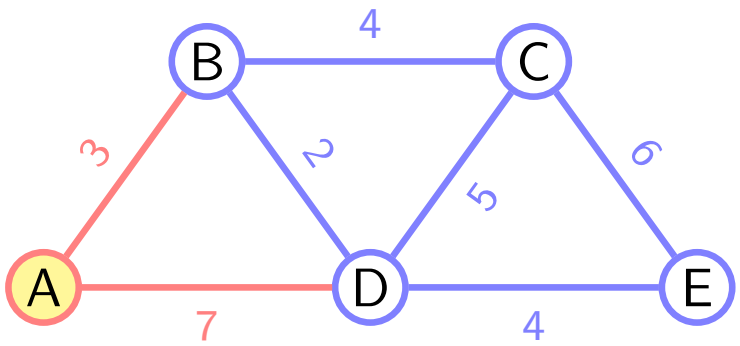
1. 根据选择的起点s，将顶点V划分为两个部分，树顶点$O = \{s\}$和边缘顶点$V - O$；
2. 给图的所有顶点V添加标记d，记录目前为止从起点s到每个顶点的最短路径长度，并记录该路径倒数第2个顶点（父节点）；
3. 从边缘顶点$V - O$中选择具有最小d值的节点u^*，把u^*加入O中；
4. 对于$V - O$中的剩余顶点u，如果通过权重为$w(u^*, u)$的边和u^*相连接，当$d_{u^*} + w(u^*, u) < d_u$时，$u$的父节点标记更新为$u^*$，最短路径长度$d_u$更新为$d_{u^*} + w(u^*, u)$；
5. 算法重复步骤3和4，逐步从$V - O$选择节点，加入到O中，直到$V = O$；

实例

如何利用Dijkstra算法寻找从点A开始的最短路径？



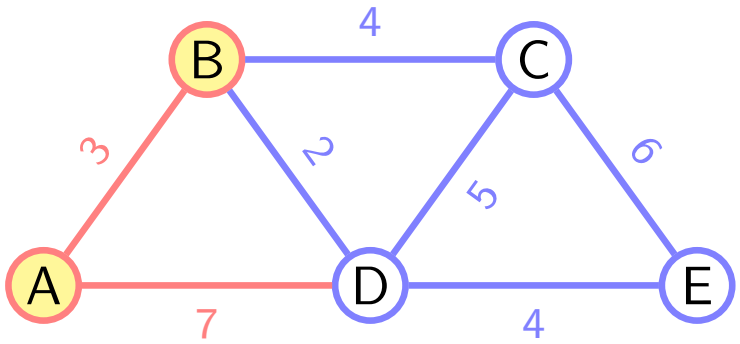
计算过程



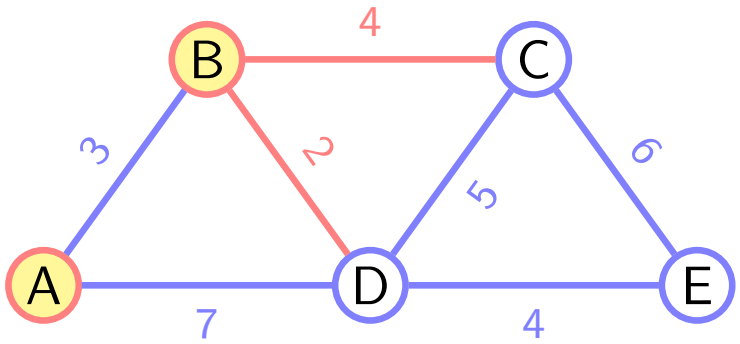
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
			B	A	3
			C	-	∞
			D	A	7
			E	-	∞

实例

计算过程



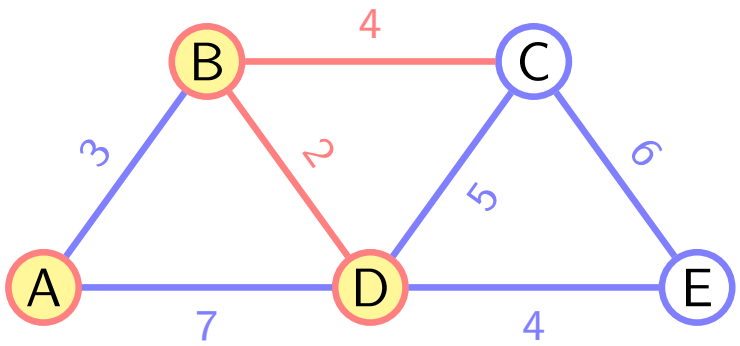
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
			C	-	∞
			D	A	7
			E	-	∞



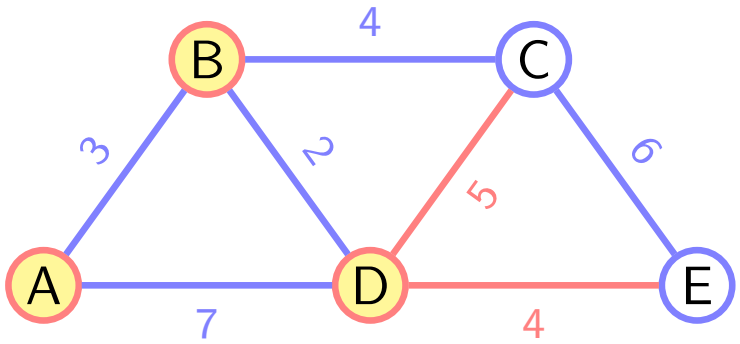
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
			C	B	3+4
			D	B	3+2
			E	-	∞

实例

计算过程



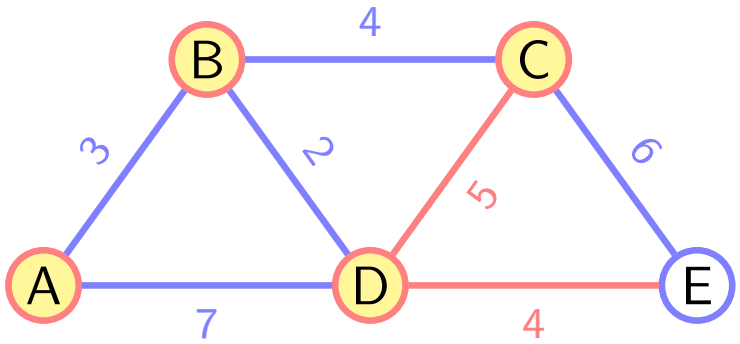
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
D	B	5			
			C	B	7
			E	-	∞



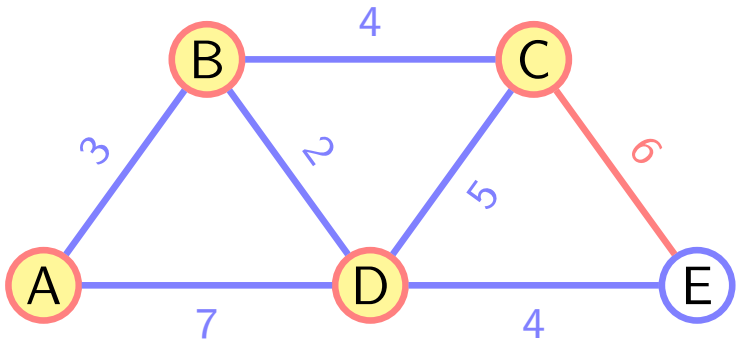
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
D	B	5			
			C	B	7
			E	D	9

实例

计算过程



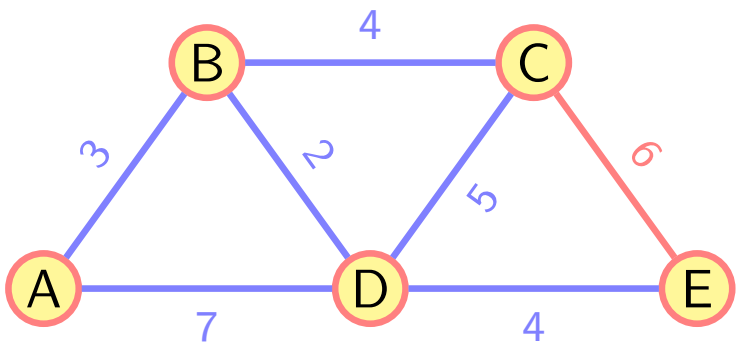
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
D	B	5			
C	B	7			
			E	D	9



树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
D	B	5			
C	B	7			
			E	D	9

实例

计算过程



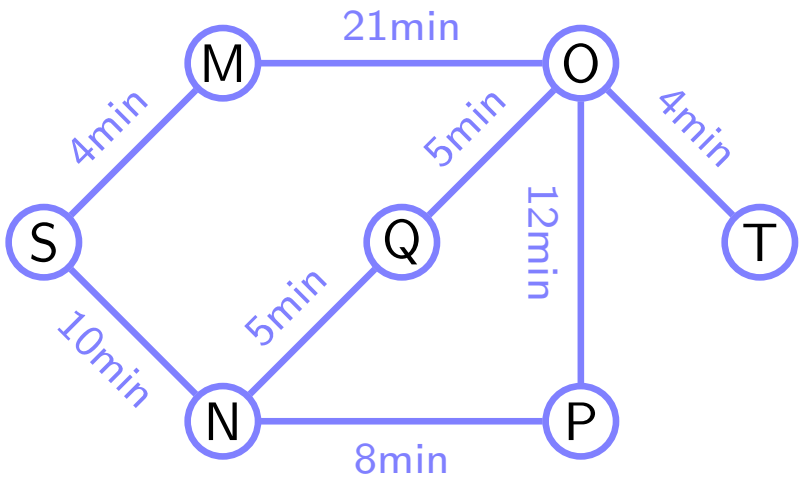
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
D	B	5			
C	B	7			
E	D	9			

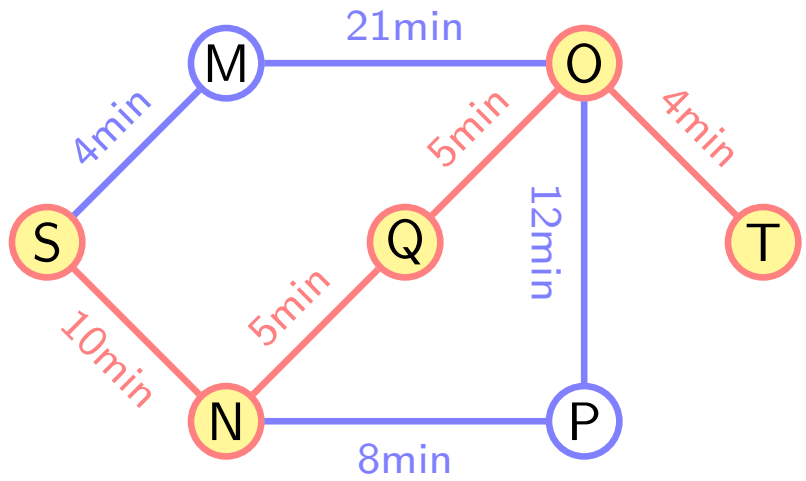
最短路径

- 从A到B：A-B，长度为3;
- 从A到C：A-B-C，长度为7;
- 从A到D：A-B-D，长度为5;
- 从A到E：A-B-D-E，长度为9;

问题回顾

寻找从“双子峰”（S）到“金门大桥”（T）的耗时最短路径





最短路径

从“双子峰”到“金门大桥”：S-N-Q-O-T，长度为24分钟

Dijkstra算法伪代码

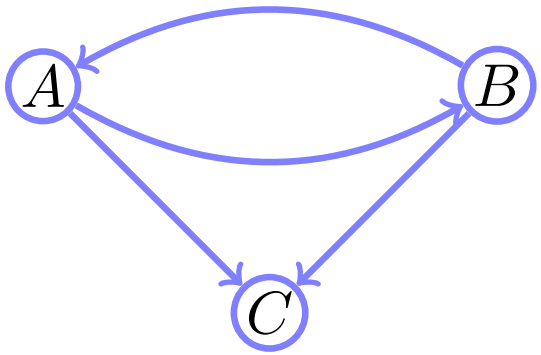
```
1:  function Dijkstra(G, s)
2:      Initialize(Q)    //将顶点优先队列初始化为空
3:      for V中每一个顶点v do
4:          d_v ← ∞; p_v ← null
5:          Insert(Q, v, d_v)    //初始化优先队列中顶点的优先级
6:      end for
7:      V_T ← ∅;  d_s ← 0;  Decrease(Q, s, d_s)    //将s的优先级更新为d_s
8:      for i ← 0 → |V| − 1 do
9:          u* ← DeleteMin(Q)    //删除优先级最小的元素
10:         V_T ← V_T ∪ {u*}
11:         for V − V_T中每一个和u*相邻的顶点u do
12:             if d_u* + w(u*, u) < d_u then
13:                 d_u ← d_u* + w(u*, u);  p_u ← u*
14:                 Decrease(Q, u, d_u)
15:             end if
16:         end for
17:     end for
18: end function
```

主要内容

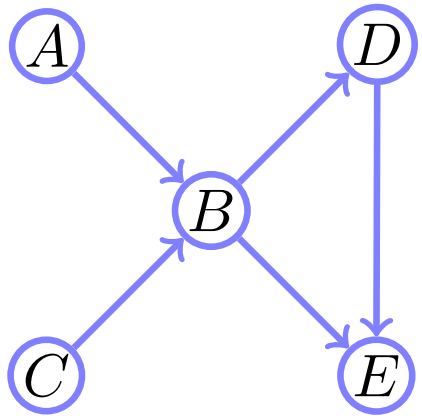
拓扑排序问题–回顾

问题描述

从有向无环图中寻找一个序列，使得图中每一条边来说，边的起始顶点总是排在边的结束顶点之前。



有向有环图



有向无环图

注意

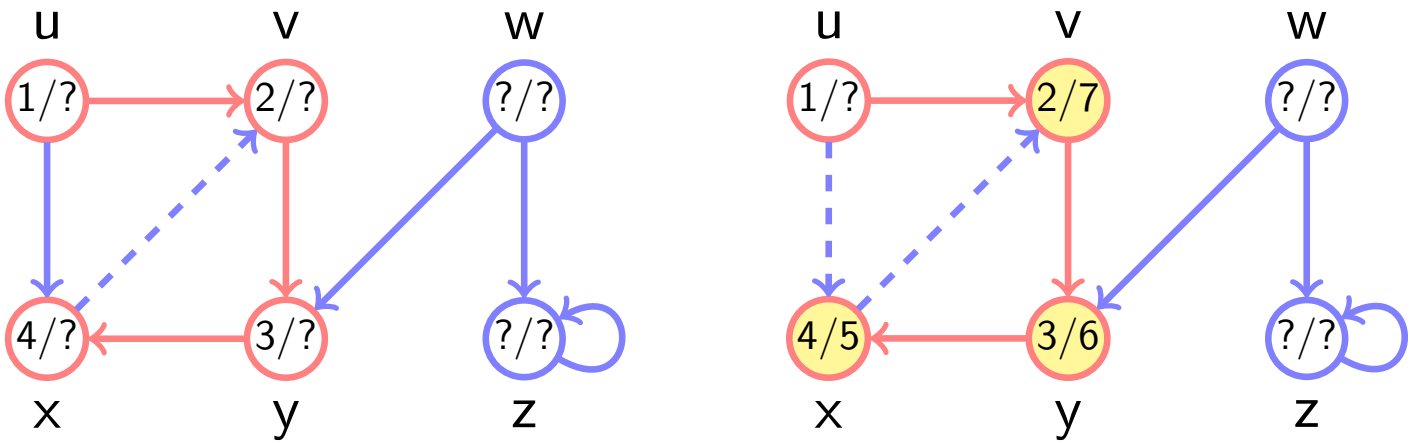
对于图中有一个回路（有向有环图），拓扑排序问题是无解的。

无环图判断方法

要点

在DFS中，如果出现一条边指向发现且未完成的节点，则该图为有环图，该边称为回边。

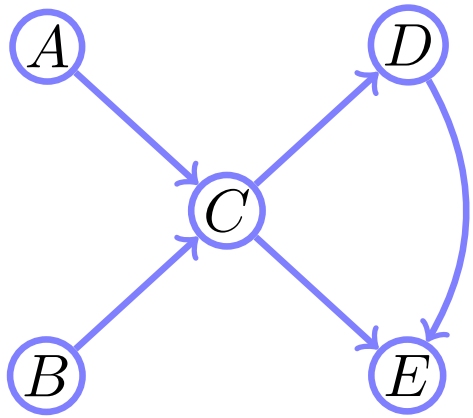
例子：(x, v)是回边，(u, x)不是回边



实例问题

问题描述

考虑5门必修课的一个集合$\{A, B, C, D, E\}$，一个在校学生必须在某个阶段修完这几门课程。学生可以按照任何次序修课，但是必须满足如下的条件，修完A和B才能修C，修完C才能修D，修完C和D才能修E。问学生该如何修课？

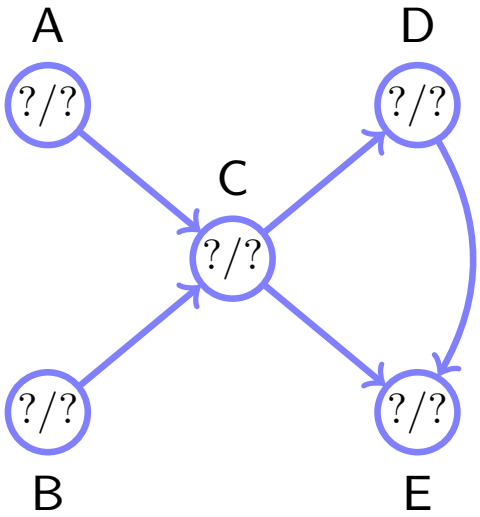


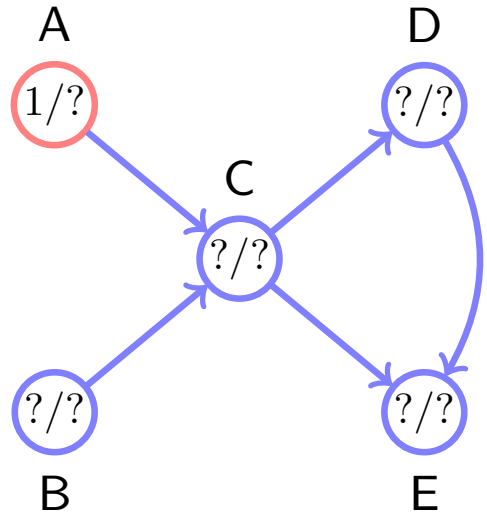
深度优先法

算法思想

执行DFS遍历搜寻，并记住顶点变成终点（完成或出栈）的顺序，将该次序反过来就得到拓扑排序的一个解。

算法过程

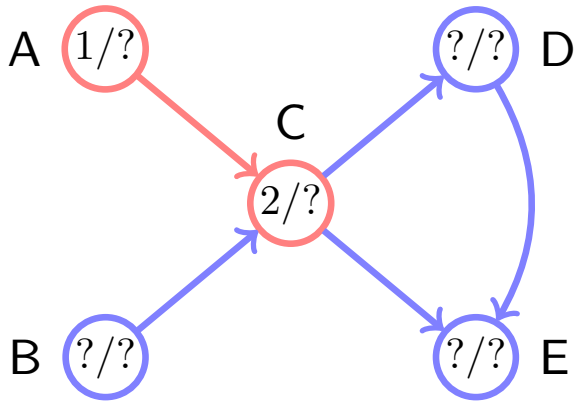


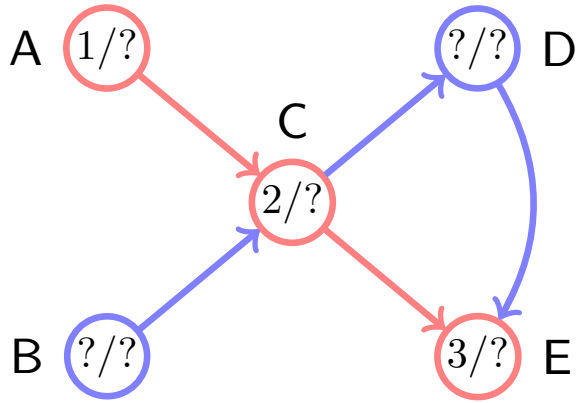


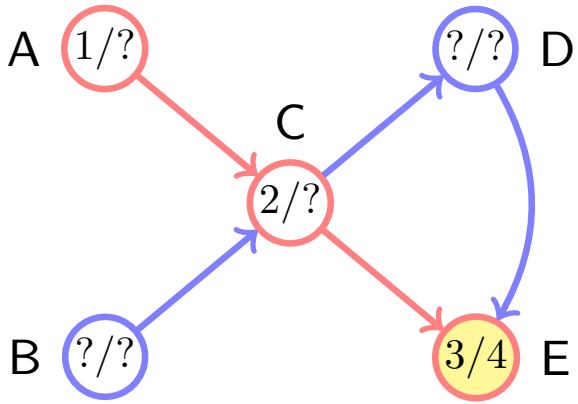
s/t 未发现节点 s/t 发现节点 s/t 完成节点 s:发现时间 t:完成时间

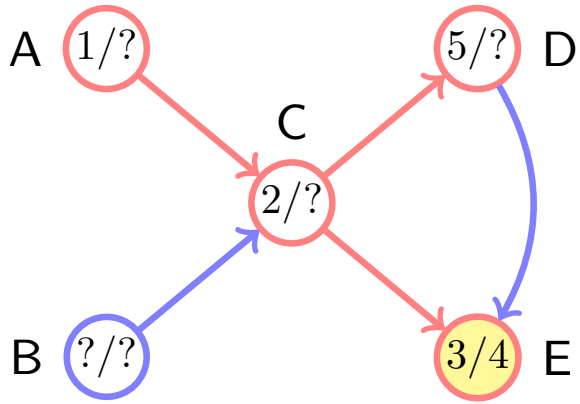
深度优先法

算法过程





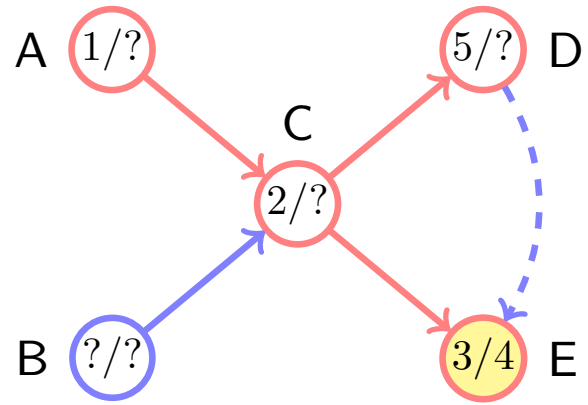


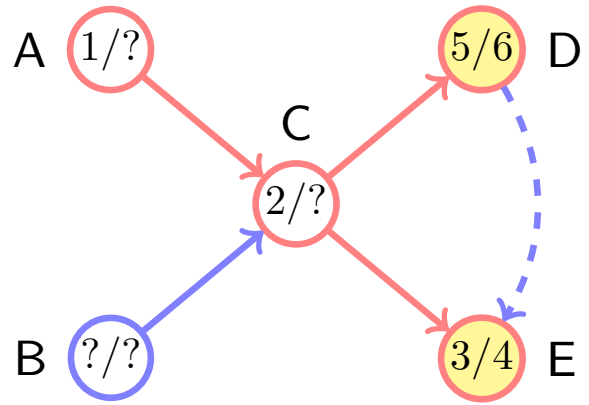


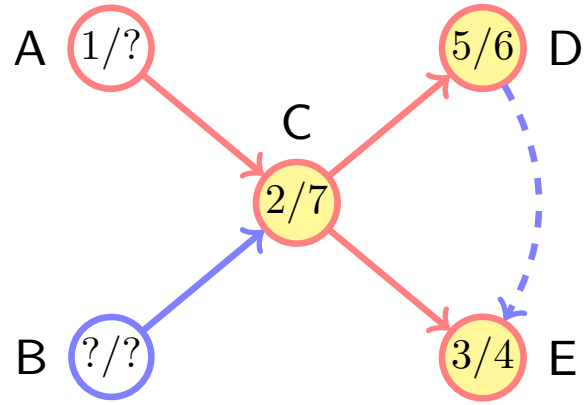
s/t 未发现节点　s/t 发现节点　s/t 完成节点　s:发现时间 t:完成时间

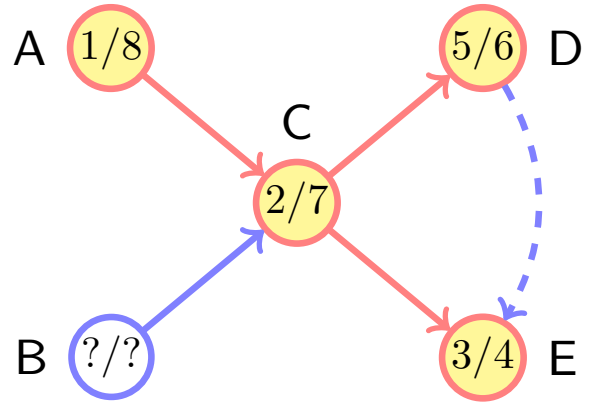
深度优先法

算法过程





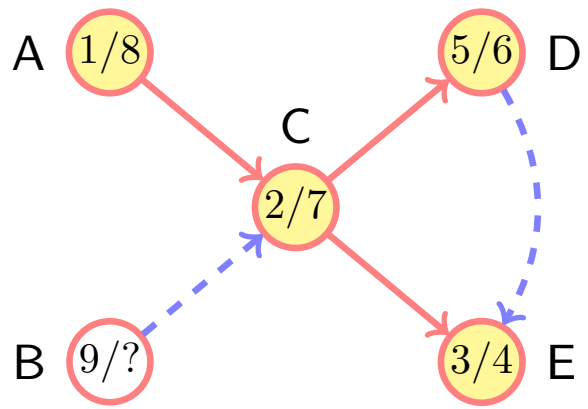


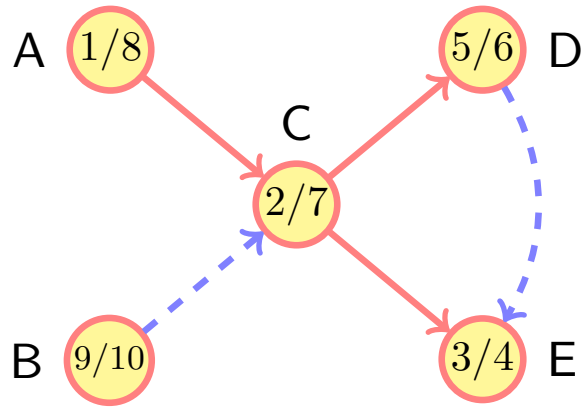


s/t 未发现节点　s/t 发现节点　s/t 完成节点　s:发现时间 t:完成时间

深度优先法

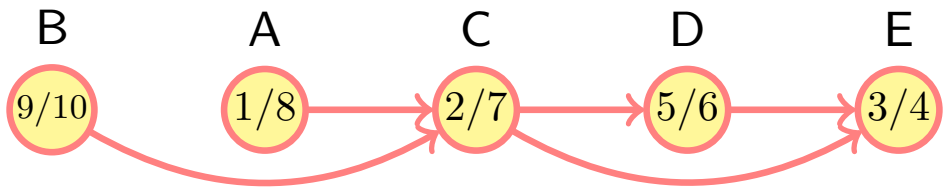
算法过程





s/t 未发现节点　s/t 发现节点　s/t 完成节点　s:发现时间 t:完成时间

拓扑顺序

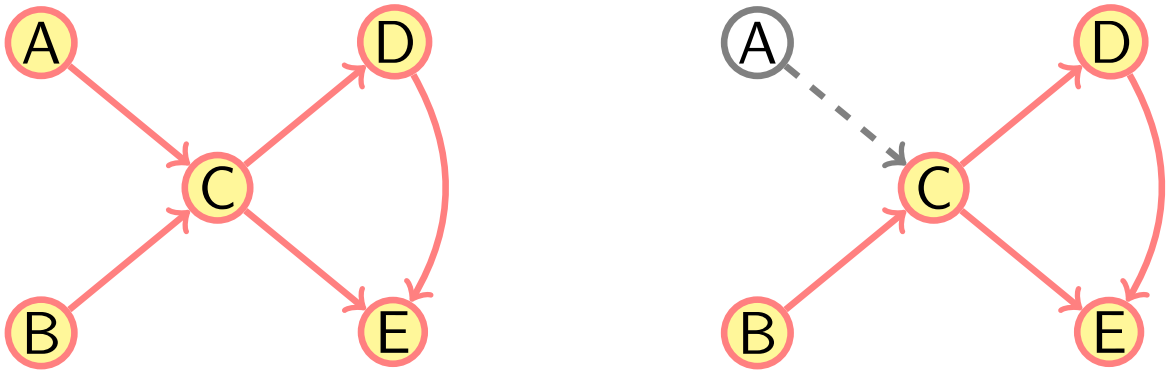


源删除法

算法思想

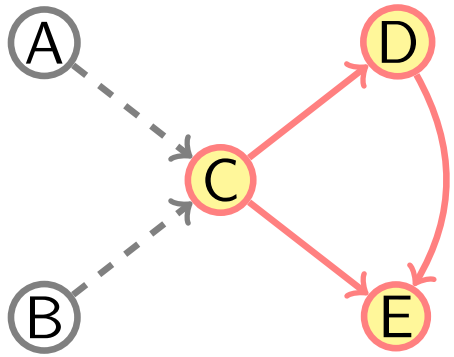
不断地在图中找出一个源点（没有输入边的顶点），然后把该点和所有从该点出发的边都删除，如此重复。顶点被删除的次序就是拓扑排序的次序。

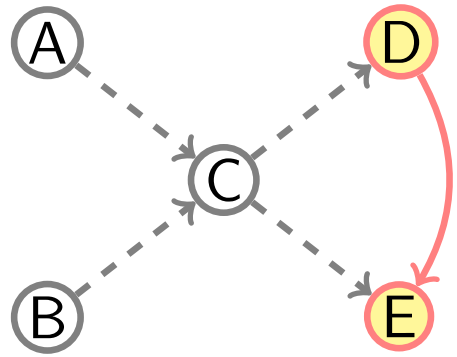
算法过程

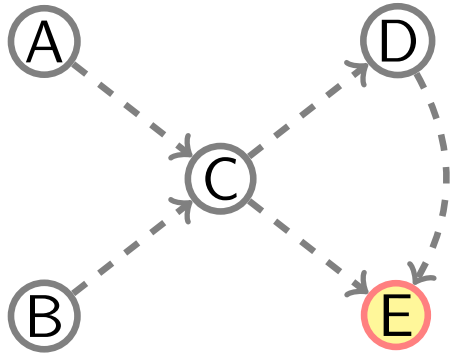


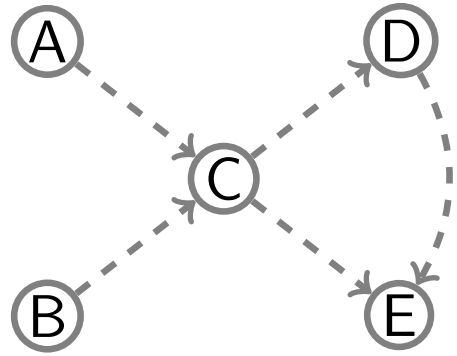
源删除法

算法过程









拓扑顺序

主要内容

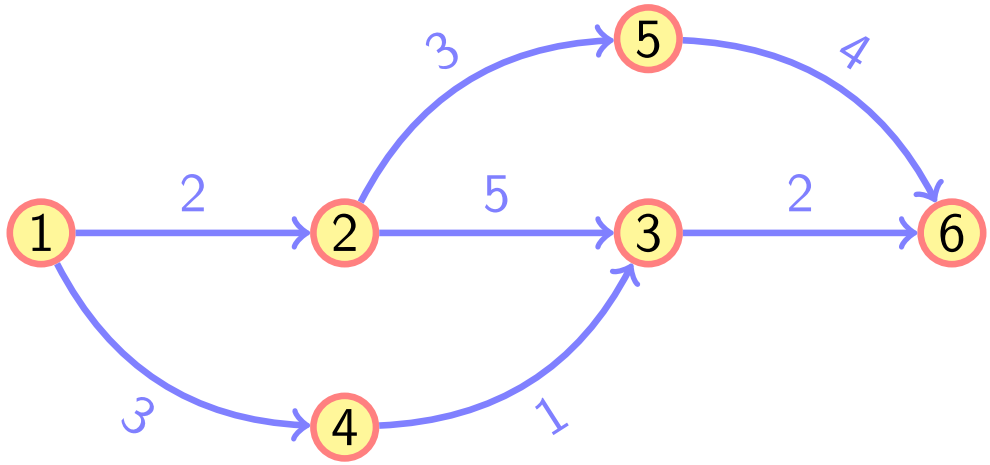
流量网络

流量网络定义

对于一个加权有向图$G = (V, E)$，具有n个顶点，如具有如下的特性：

- 包含1个没有输入边的顶点（称为源点，标识为1）；
- 包含1个没有输出边的顶点（称为汇点，标识为n）；
- 每条有向边(i, j)的权重u_{ij}是一个正整数（该边的容量）；

满足这些特性的有向图称为流量网络。



最大流量问题

流量守恒

进入任意中间顶点i的总流量等于离开的总流量：

$$\sum_{j:(j,i)\in E} x_{ji} = \sum_{j:(i,j)\in E} x_{ij}$$

最大流问题

对于每条边$(i,j) \in E$的流量x_{ij}，满足容量约束：

$$0 \leqslant x_{ij} \leqslant u_{ij}$$

并且中间节点满足流量守恒要求，最大流问题的目标是最大化v:

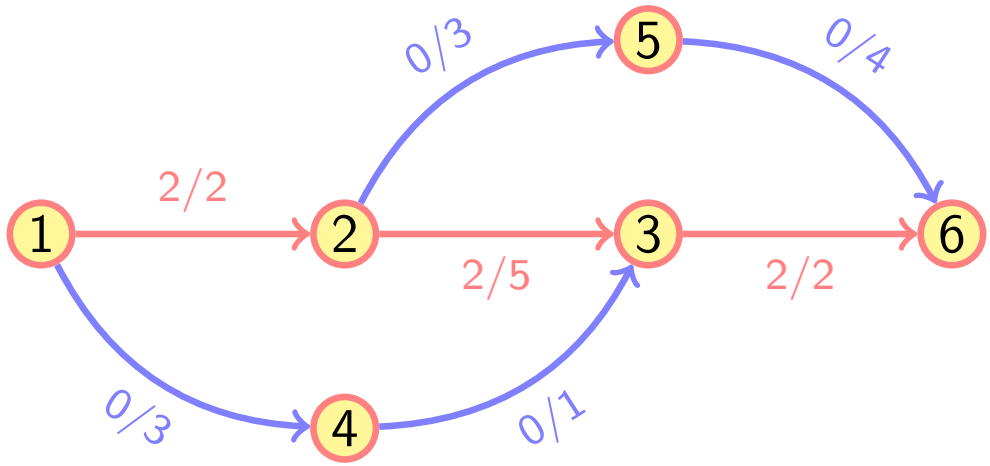
$$v = \sum_{j:(1,j)\in E} x_{1j} = \sum_{j:(j,n)\in E} x_{jn}$$

增益路径法

方法要点

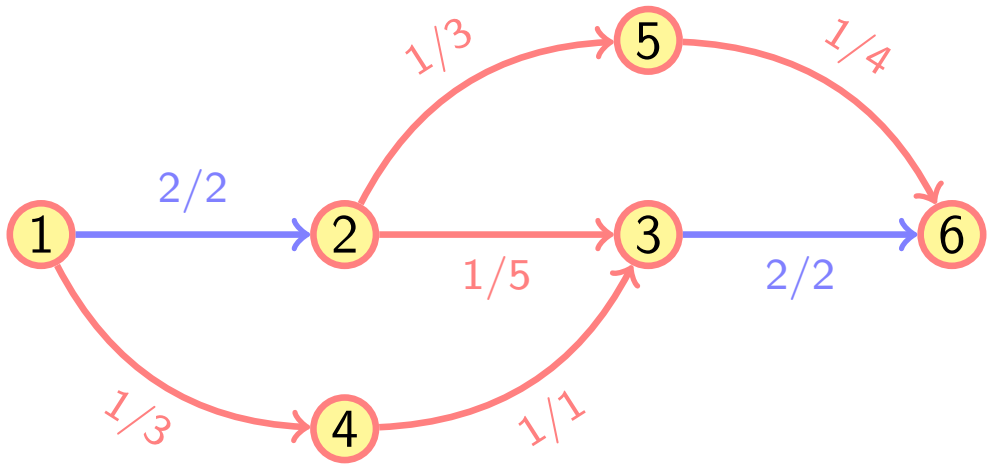
从流量为0开始，每次迭代，试着找到一条可以传输更多从源点到汇点流量的增益路径，并逐步调整各条边上的流量，直到找不到增益路径为止。

问题：流量是否是最大的？是否有其他增益路径？



增益路径法

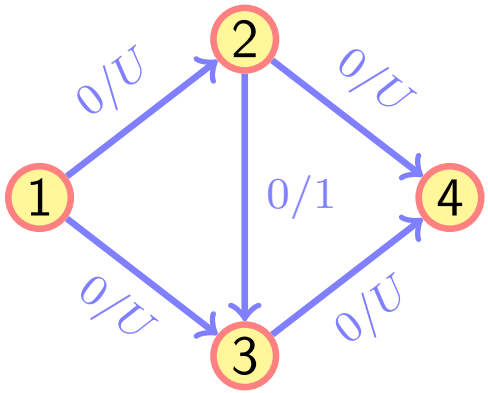
最优增益路径

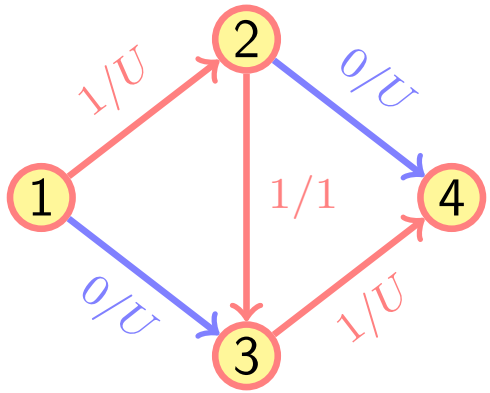


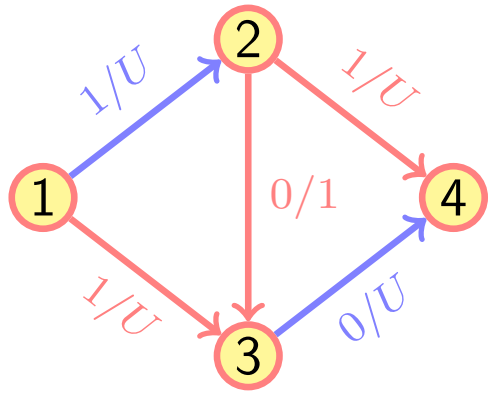
路径具有前向边和反向边时的增益方法

流量增益路径选择次序

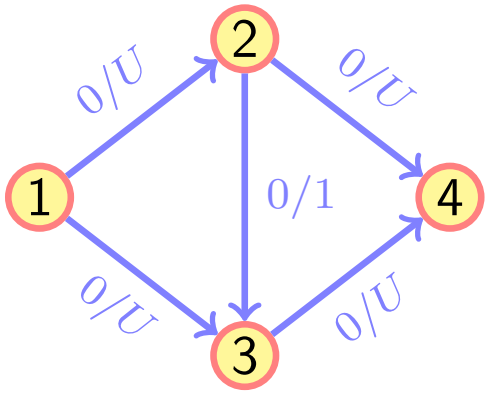
次序1：总共需要迭代$2U$次

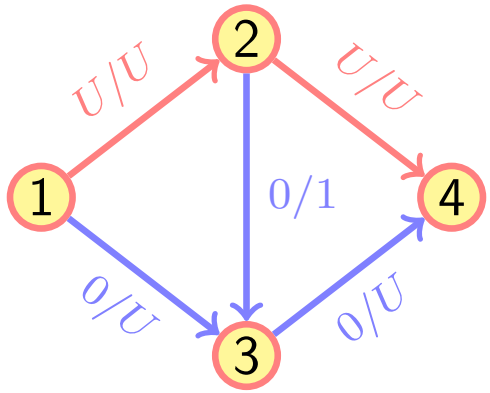


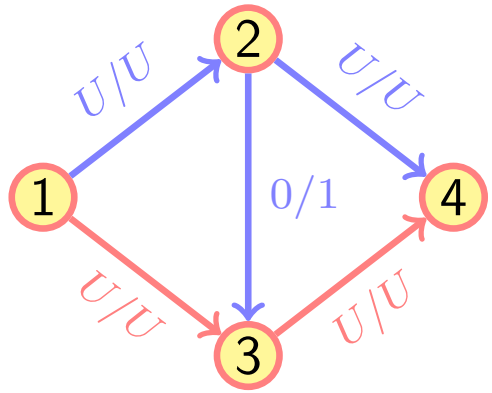




次序2：总共需要迭代2次





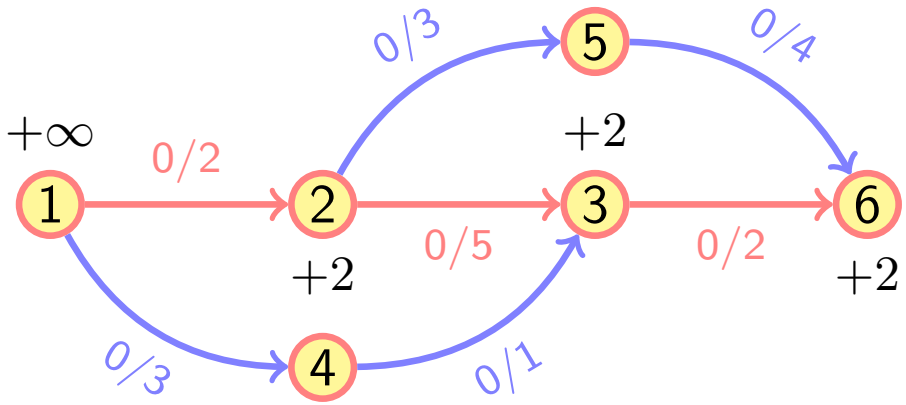


最短增益路径法

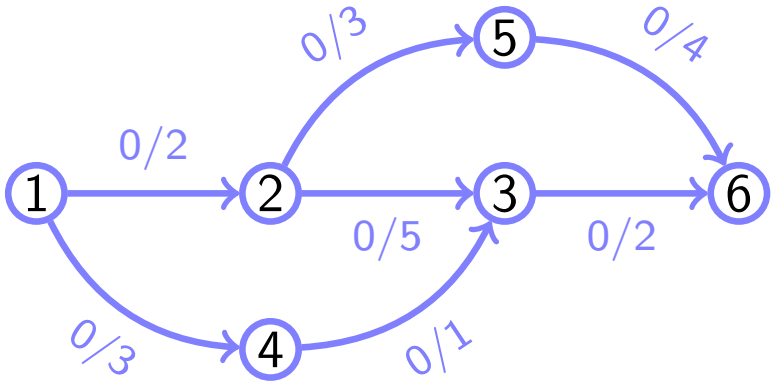
算法要点

1. 用广度优先查找法生成增益路径（边数量最少）；
2. 逐步标记当前增益路径每个顶点能够增加（正向边）或者减少（反向边）的流量大小；
3. 用汇点的流量标记修正当前增益路径每个顶点的流量；
4. 重复步骤$1-3$，直到优先队列为空（没有其他增益路径）；

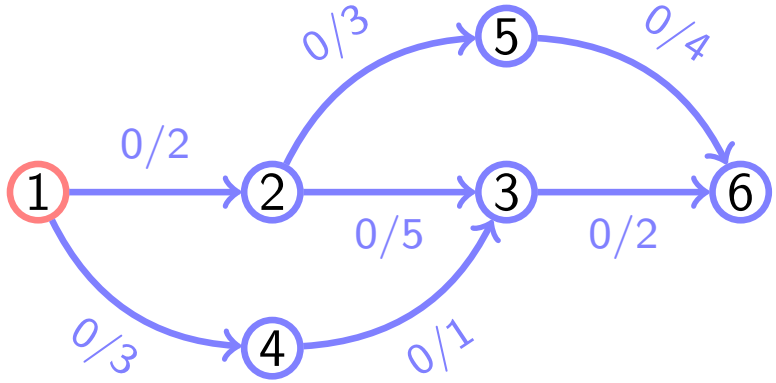
示例



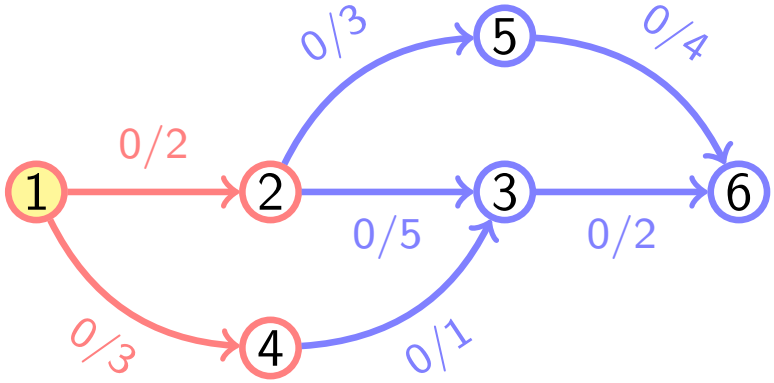
实例


1
2
3
4
5
6
0/2
0/3
0/3
0/4
0/5
0/2
0/1

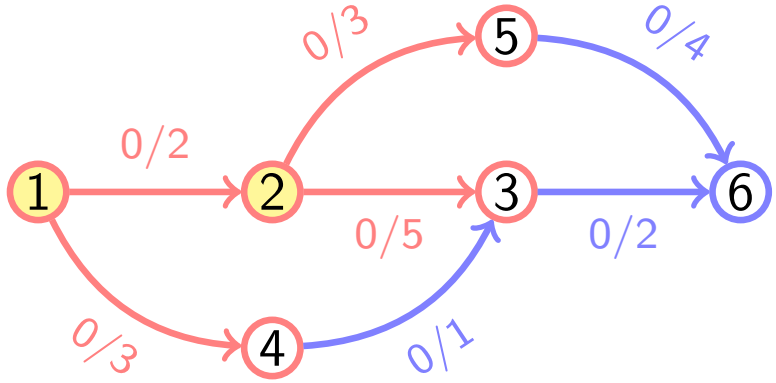
∅


1
2
3
4
5
6
0/2
0/3
0/3
0/4
0/5
0/2
0/1

1

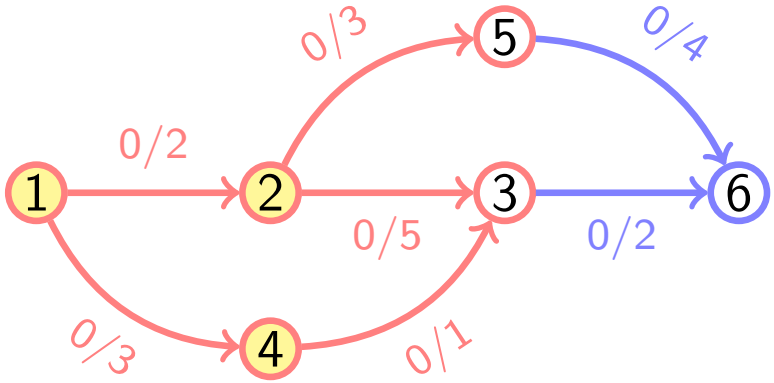

1
2
3
4
5
6
0/2
0/3
0/3
0/4
0/5
0/2
0/1

2 4

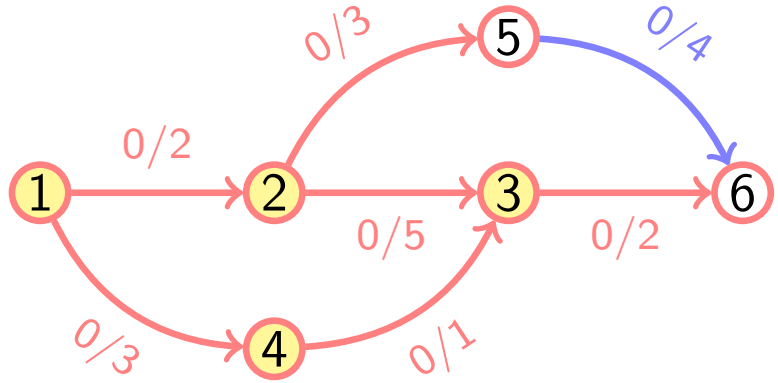

1
2
3
4
5
6
0/2
0/3
0/3
0/4
0/5
0/2
0/1

4 3 5

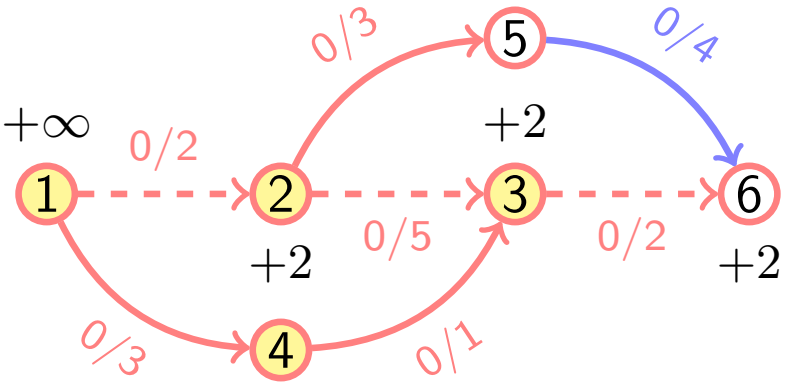
实例



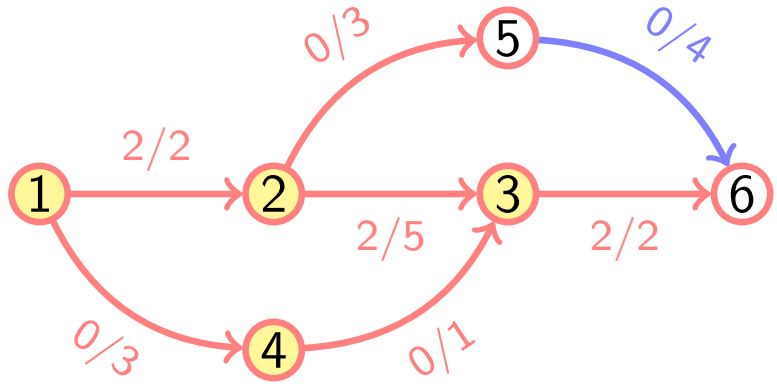
3 5 3



5 3 2 6

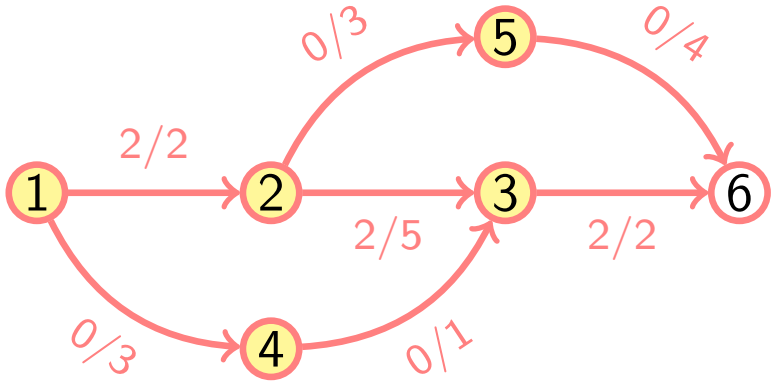


5 3 2 6

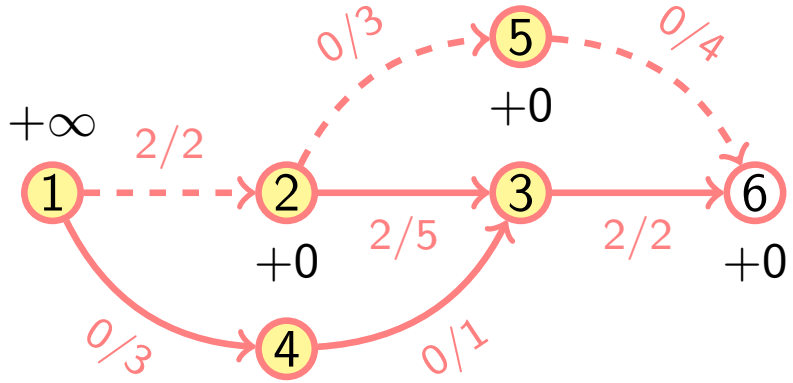


5 3 2 6

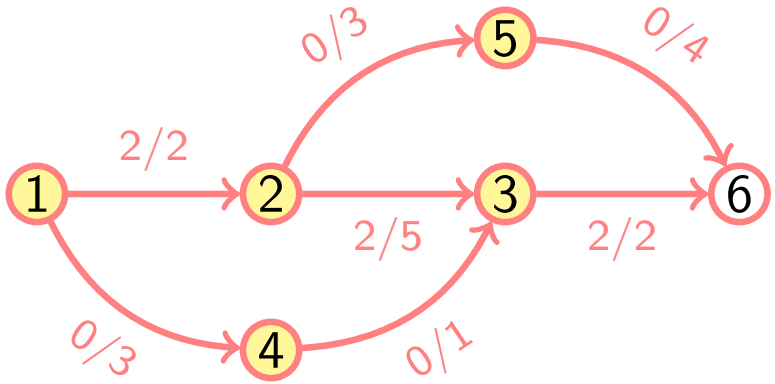
实例



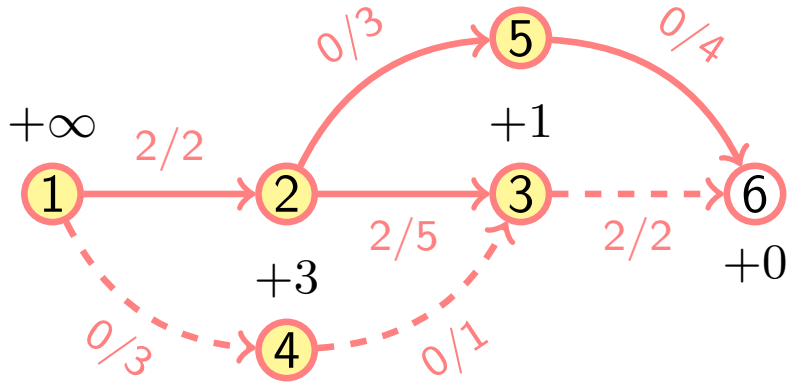
3 2 6 2 6



3 2 6 2 6

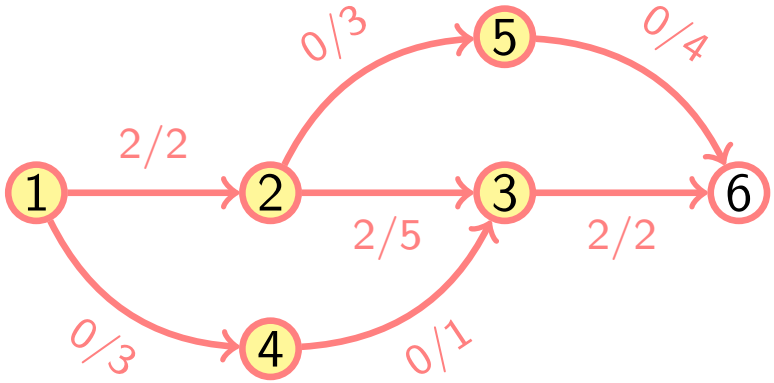


2 6 2 6 2 6

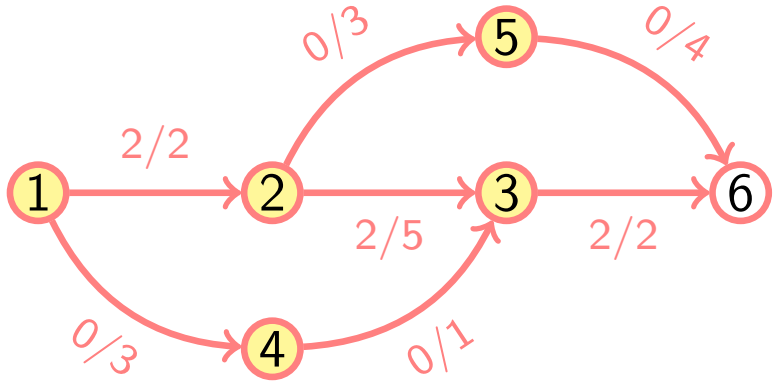


2 6 2 6 2 6

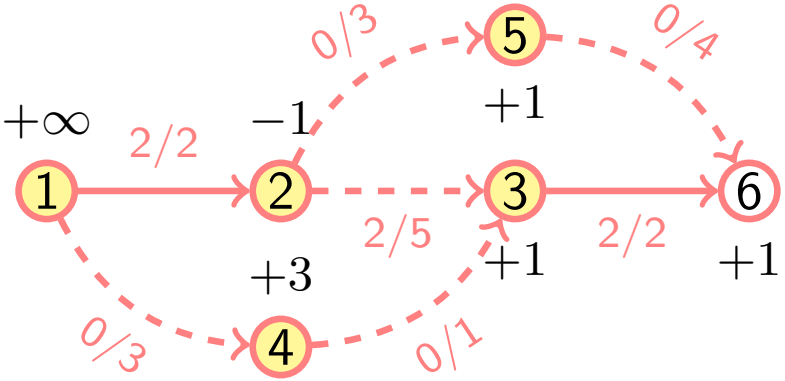
实例



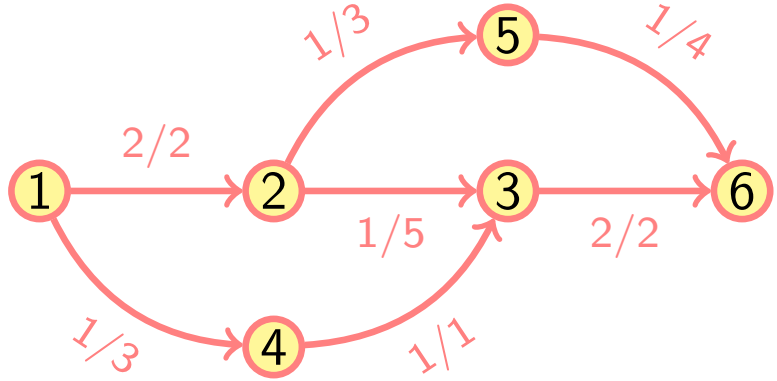
6 3 5



2 6



2 6



2 6

主要内容

回溯法

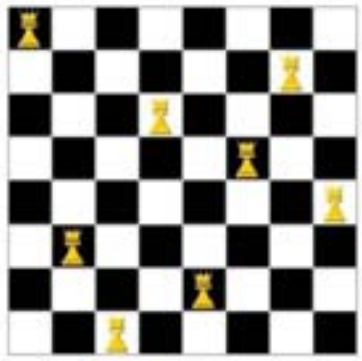
- 许多问题很难用算法求解，但是又不得不解决；
- 回溯法搜索问题的解空间，可看作是穷举查找的改进，可解决规模较大的组合难题；
- 回溯法以构造一个状态空间树为基础，树的节点代表部分构造解；
- 回溯法每次构造候选解的一个分量，然后评估这个部分构造解，若加上剩下的分量不可能求得一个解，则放弃该解；
- 回溯法通常按照深度优先、宽度优先或者宽度-深度结合等多种方法遍历树；
- 分支界限法是一种改进版的回溯法，通过剪枝等方法改善算法的运行时间；

n皇后问题

问题描述

将n个皇后放在一个$n*n$的棋盘上，使得任何两个皇后都不互相攻击。

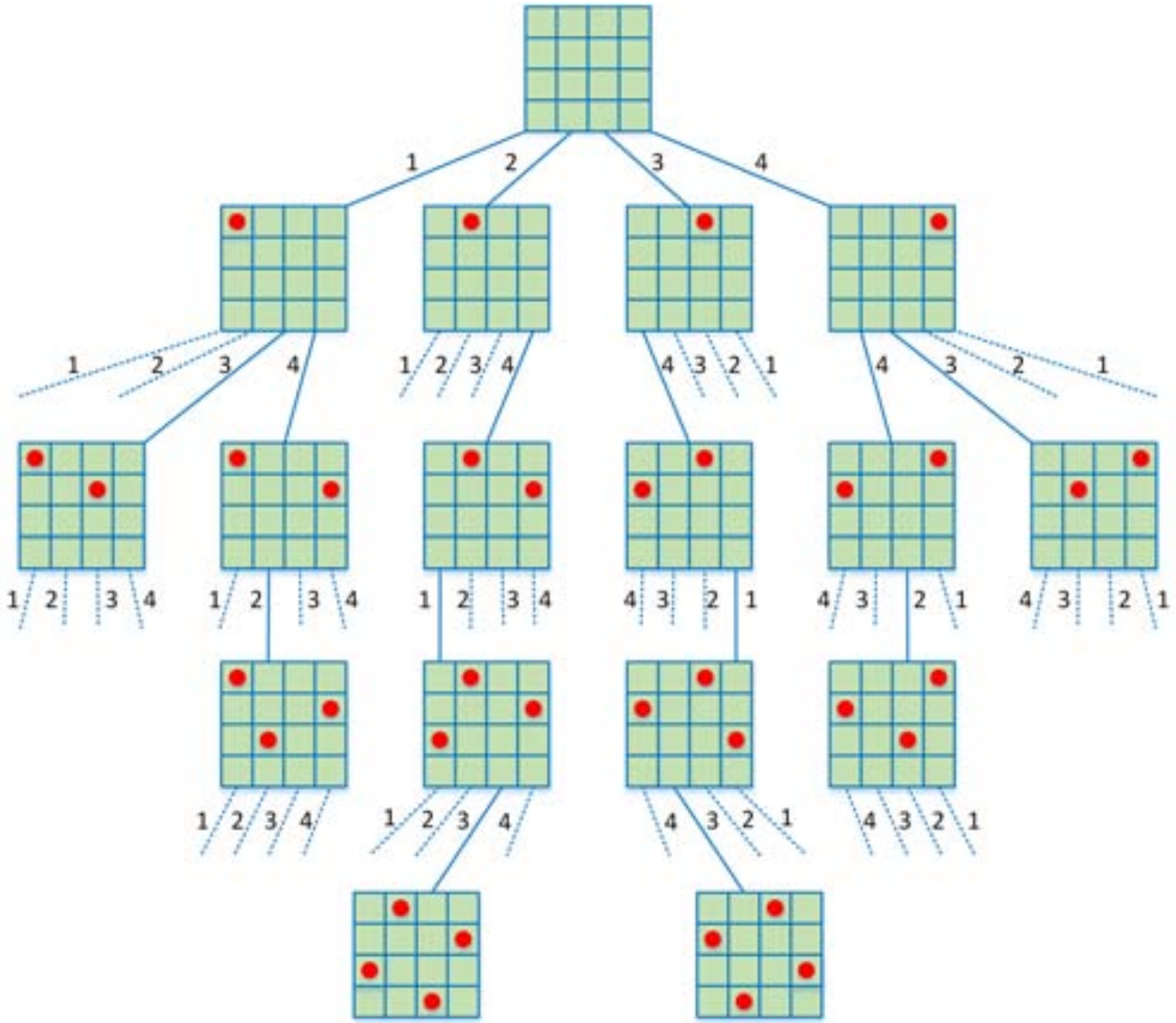
8皇后问题



问题

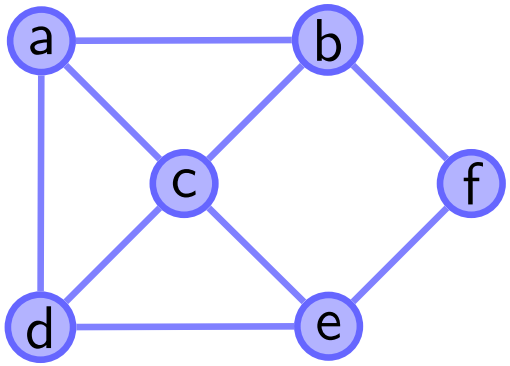
- 2皇后和3皇后问题有没有解？
- 4皇后问题有没有解？有几个解？

4皇后问题求解

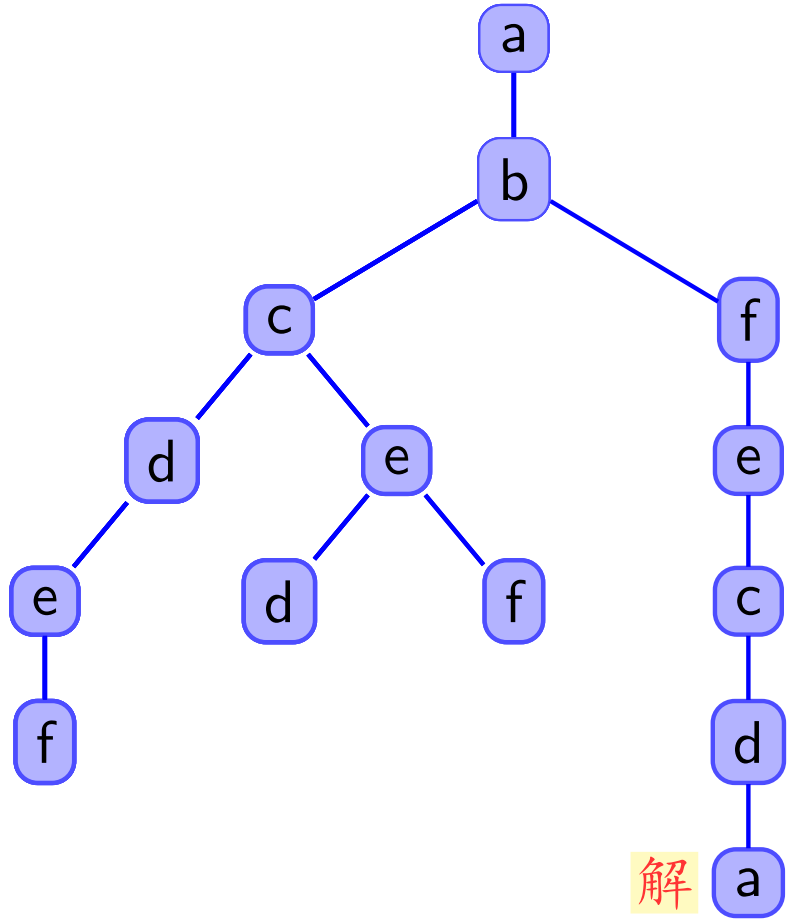

1 2 3 4
1 2 3 4
1 2 3 4
4 3 2 1
4 3 2 1
1 2 3 4
1 2 3 4
1 2 3 4
4 3 2 1
4 3 2 1
4 3 2 1
1 2 3 4
1 2 3 4
4 3 2 1
4 3 2 1

哈密顿回路

图



回溯过程

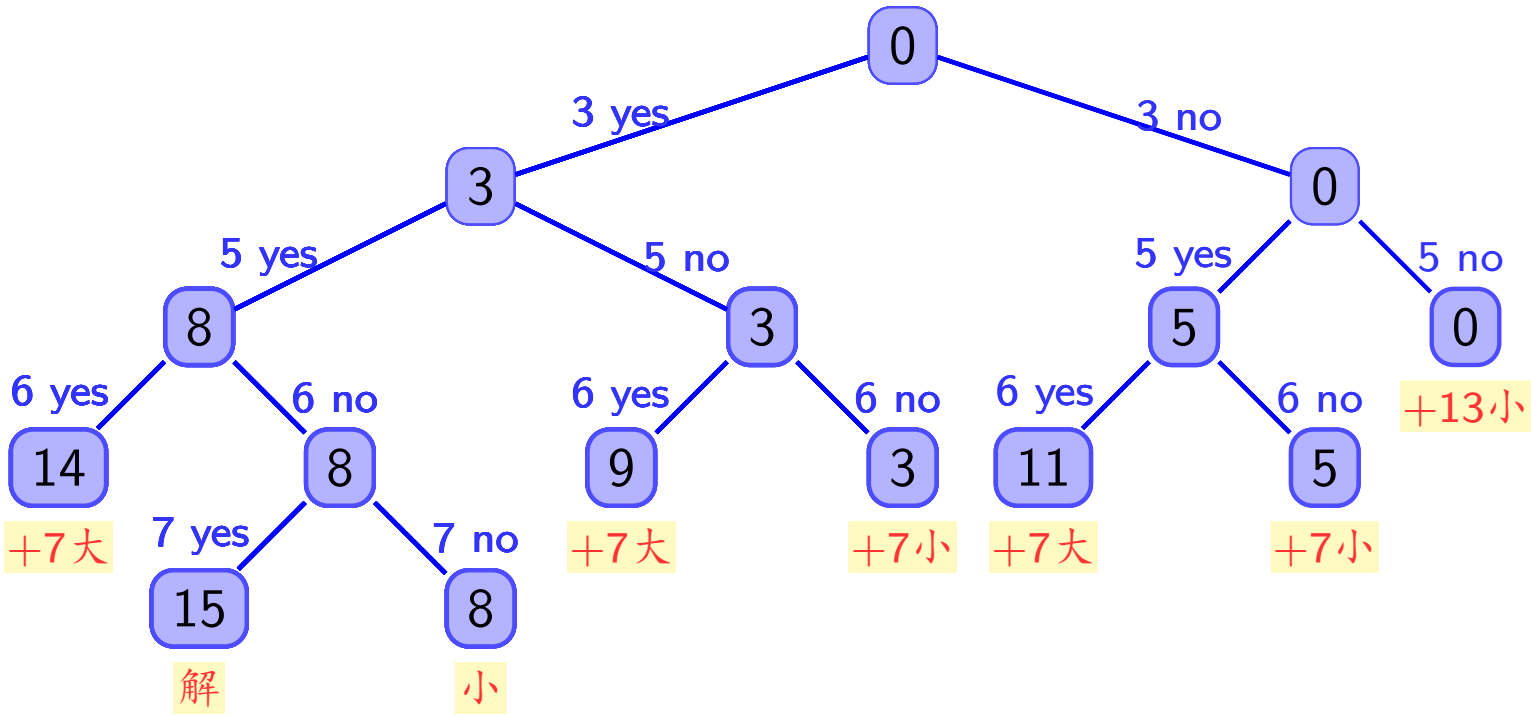


子集和问题

问题描述

求n个正整数构成的一个给定集合$A = \{a_1, .., a_n\}$的子集，子集的和要等于一个给定的正整数d。

例子：$A = \{3, 5, 6, 7\}, d = 15$



主要内容

分支界限

主要思想

用节点的边界值和目前求得的最佳解比较，如果边界值不能超过目前的最佳解，立即终止继续搜索（即从该节点上生成的解，没有一个能比目前已经得到的解更好）。

分支界限法节点终止条件

- 节点上的边界值不能超越目前最佳解的值；
- 节点无法代表任何可行解，因为它违反了问题的约束；
- 节点上的可行解子集只包含一个单独的顶点；

最佳优先分支边界

不断地选择当前树中最有希望的节点（最佳下界的节点）进行搜寻。

分配问题

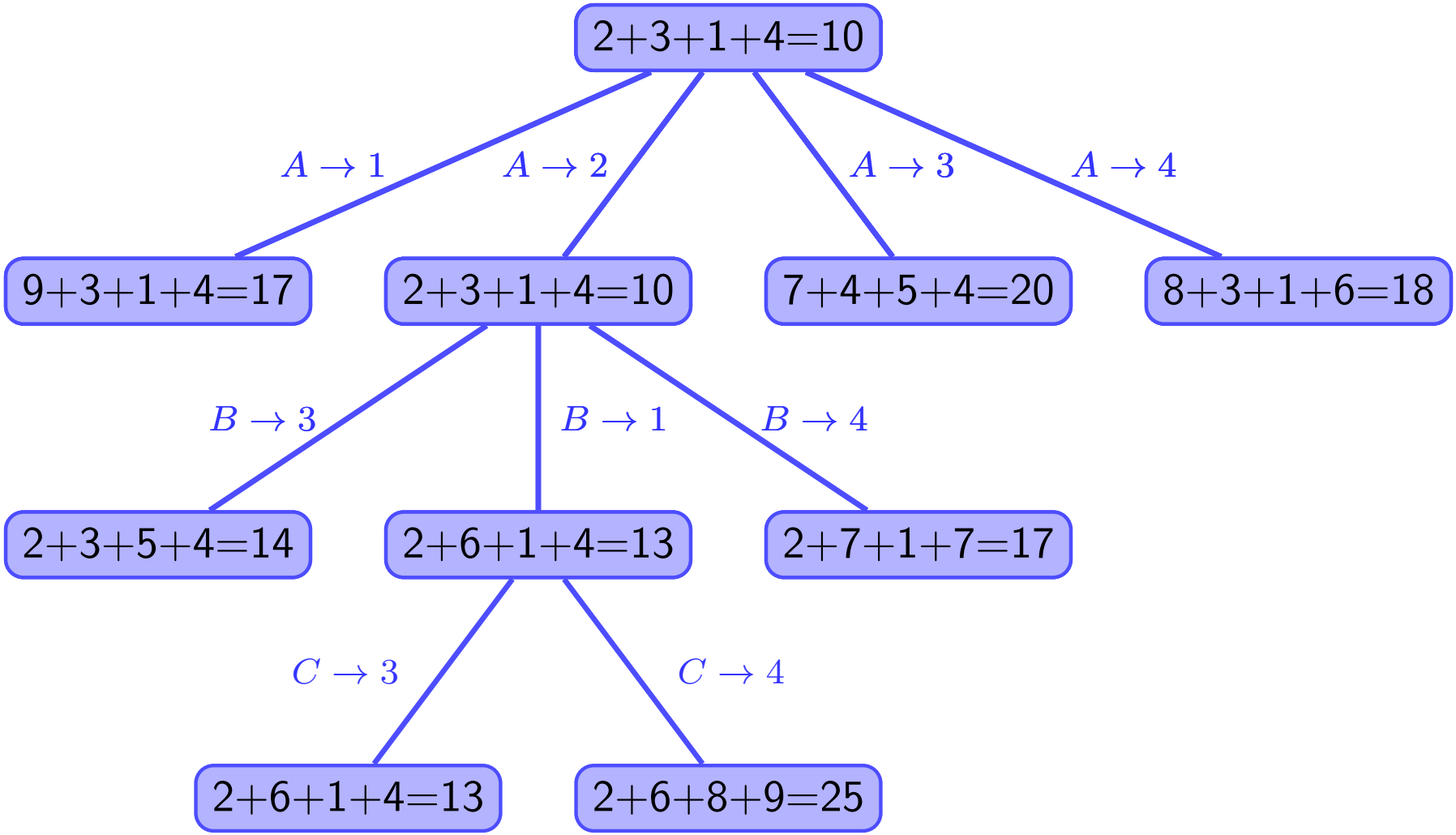
问题：寻找最小成本分配

人员＼任务	1	2	3	4
A	9	2	7	8
B	6	4	3	7
C	5	8	1	8
D	7	6	9	4

成本下界

- 任何解的成本不会小于每行最小元素的和，即$2+3+1+4=10$;
- 任务1分配给人员A的所有解的成本下界为$9+3+1+4=17$;
- 任务2分配给人员B的所有解的成本下界为$7+4+1+4=16$;
- 任务1分配给人员A，任务2分配给人员B，所有解的成本下界为$9+4+1+4=18$;

最佳优先分支边界算法求解



背包问题

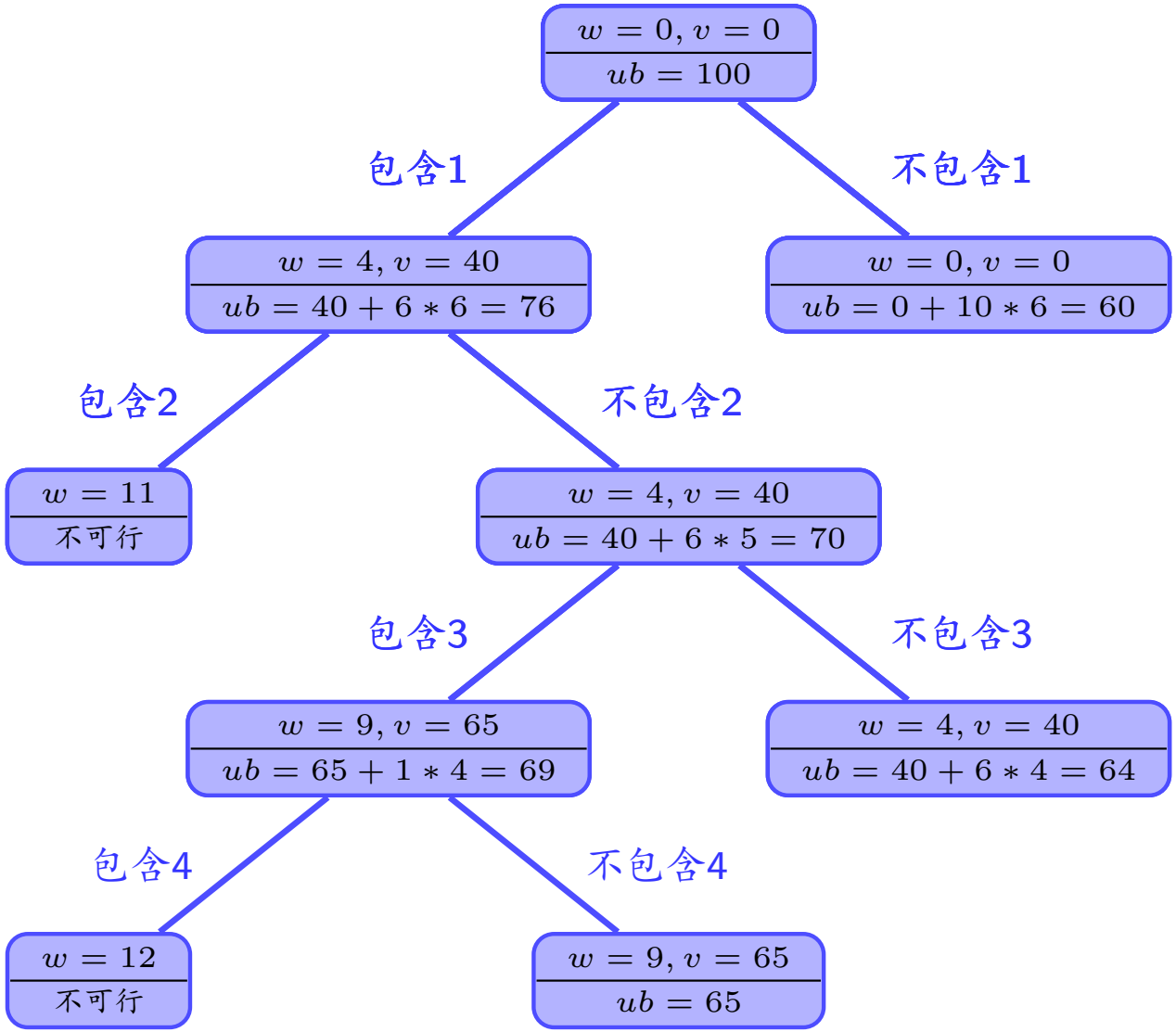
问题：装入承重量$W = 10$的背包价值最大的物品

物品	重量	价值/美元	价值/重量
1	4	40	10
2	7	42	6
3	5	25	5
4	3	12	4

价值上界

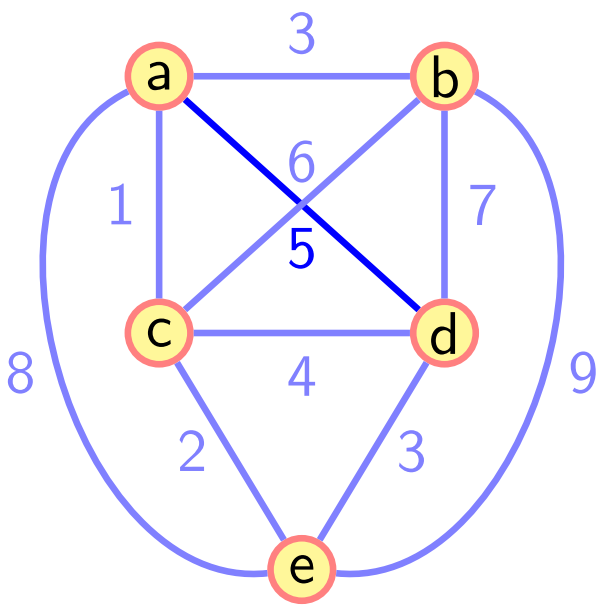
- 对于给定实例中的物品，按照降序对“价值/重量”比排序；
- 依次装入还未装入背包价值最大的物品；
- 若已经选的物品总价值为v，重量为w，后续装入物品后总价值的上界ub =背包剩余重量$(W - w)$*剩下物品最佳单位价值v_{i+1}/w_{i+1}；

最佳优先分支边界算法求解



旅行商问题

图

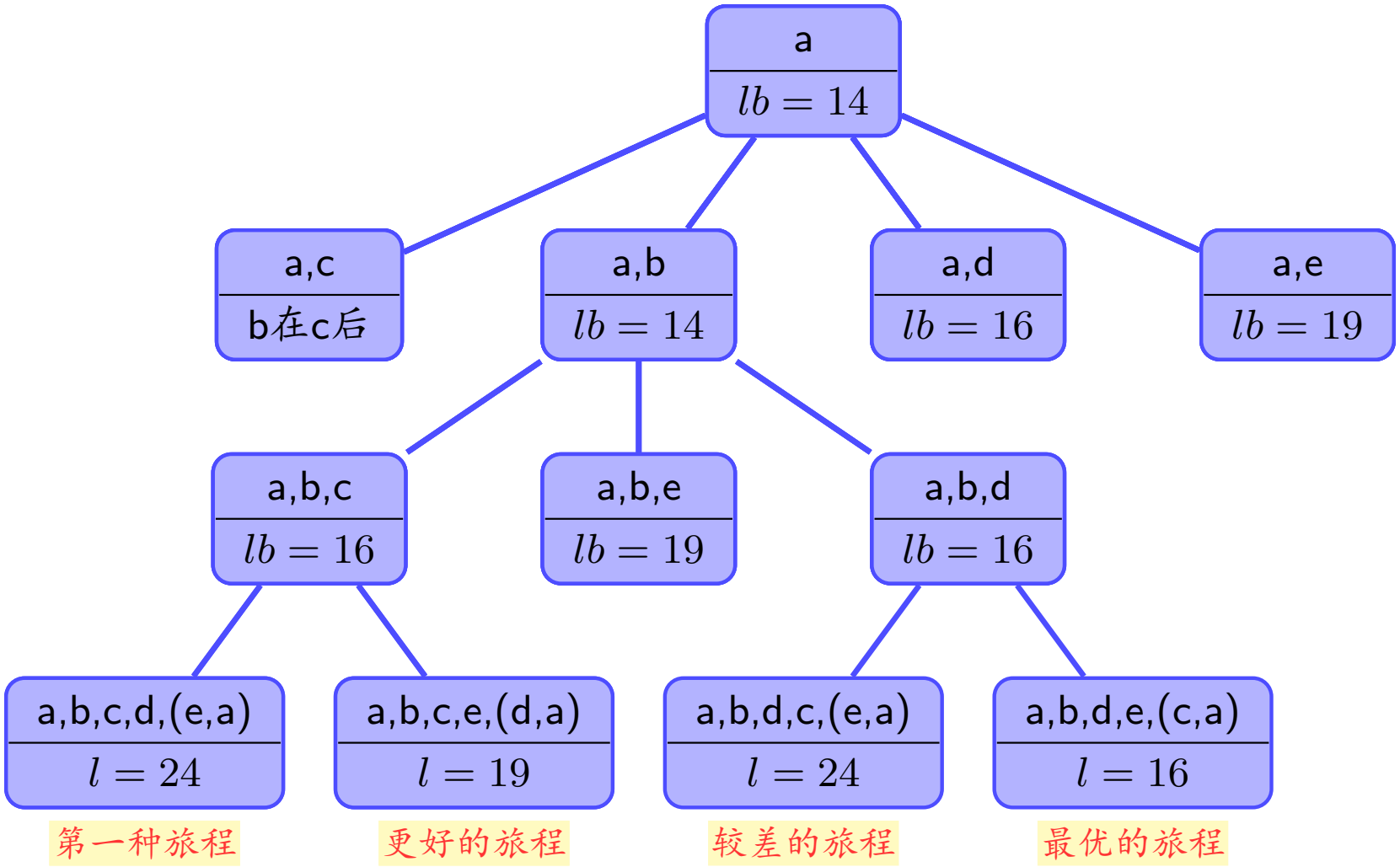


旅程下界

对于每一个城市，计算其到最近两个城市的距离之和，然后把所有城市的值加起来，除以2向上取整得到下界；

	a	b	c	d	e	下界
∅	1+3	3+6	1+2	3+4	2+3	14
a,b	1+3	3+6	1+2	3+4	2+3	14
a,d	1+5	3+6	1+2	3+5	2+3	16
a,e	1+8	3+6	1+2	3+5	2+8	19
a,b,c	1+3	3+6	1+6	3+4	2+3	16
a,b,d	1+3	3+7	1+2	3+7	2+3	16
a,b,e	1+3	3+9	1+2	3+4	2+9	19

最佳优先分支边界算法求解b在c之前的路径



小结

- 图搜索方法主要有深度优先和广度优先方法；
- Dijkstra算法贪婪地寻找代价最低的节点，并不断地更新节点邻居的代价，从而为单源最短路径问题提供最优解；
- 拓扑排序问题的两种解法：深度优先搜寻和源删除法；
- 最大流问题采用最优路径增益发求解，增益路径的生成采用广度优先搜寻的方式；
- 回溯法和分支界限法为许多很难，但是又不得不解决的问题（n皇后问题）提供了求解方法；
- 分支界限法是一种改进版的回溯法，通过评估解的上下界，剪枝搜索树，从而加快求解速度；

作业

- 9.3: 2 $Page259$
- 9.3: 4 $Page260$

Part V

动态规划

阅读: 8.1-8.4

主要内容

1. 动态规划思想
2. 基本例子
   - 币值最大化问题
   - 找零问题
   - 硬币收集问题
3. 背包问题
4. 最优二叉查找树
5. 图传递闭包问题
   - Warshall算法
6. 完全最短路径问题
   - Floyd算法
7. 矩阵链乘计算
8. 最长递增子序列
9. 编辑距离问题

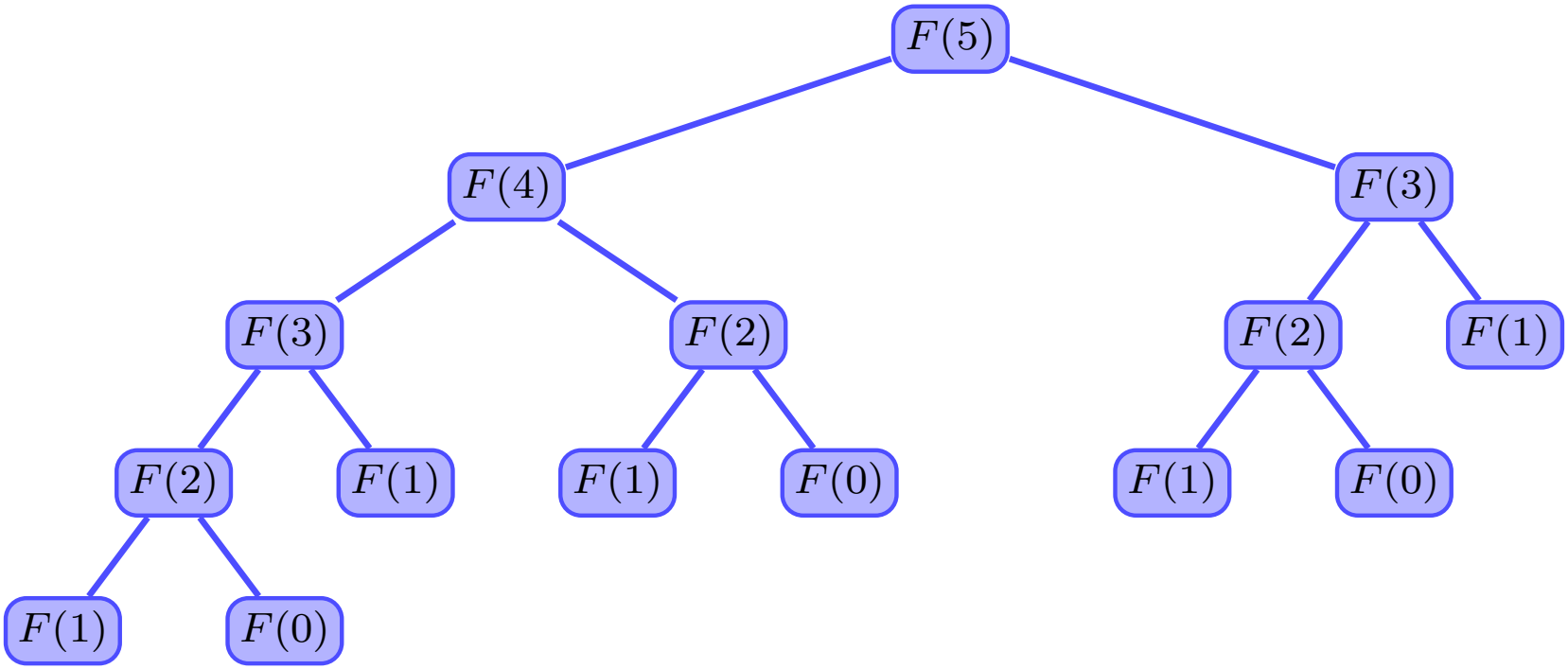
主要内容

动态规划思想

要点

如果问题是由交叠的子问题构成（递推），动态规划对每个子问题只求解一次，并把结果记录在表中，递推计算得到原始问题的解。

例子：计算Fibonacci数列



主要内容

币值最大化问题

问题描述

给定一排n个硬币，其面值均为正整数，$c_1, c_2, ..., c_n$，请问如何选择硬币，使得在其原始位置互不相邻的情况下，所选硬币的总金额最大。

递推关系

- 所有可行的计划：包括最后一枚硬币，不包含最后一枚硬币；
- 第1组中，可选硬币的最大金额为$c_n + F(n-2)$；
- 第2组中，可选硬币的最大金额为$F(n-1)$；
- $F(n) = max\{c_n + F(n-2), F(n-1)\}, n > 1; F(0) = 0, F(1) = c_1$；

计算复杂度

需要计算$n-1$个F值，每个需要2次比较，复杂度为$\Theta(n)$。

实例

计算最大化币值

$$c_1 = 5, c_2 = 1, c_3 = 2, c_4 = 10, c_5 = 6, c_6 = 2$$

计算过程

- $F(n) = max\{c_n + F(n-2), F(n-1)\}$
- $F(0) = 0;\ \ F(1) = c_1 = 5$
- $F(2) = max\{c_2 + F(0), F(1)\} = max\{1 + 0, 5\} = 5$
- $F(3) = max\{c_3 + F(1), F(2)\} = max\{2 + 5, 5\} = 7$
- $F(4) = max\{c_4 + F(2), F(3)\} = max\{10 + 5, 7\} = 15$
- $F(5) = max\{c_5 + F(3), F(4)\} = max\{6 + 7, 15\} = 15$
- $F(6) = max\{c_6 + F(4), F(5)\} = max\{2 + 15, 15\} = 17$

找零问题

问题描述

给定m个硬币，其面值分别为$d_1, d_2, ..., d_m$，$d_1 < d_2 ... < d_m$，需找零金额为n，最少需要多少硬币？

递推关系

- 获得总金额为n的只能是在总金额为$n - d_j$的一堆硬币上加入一个面值为d_j的硬币；
- 先找出最小的$F(n - d_j)$，然后加上1即可得到解；
- $F(n) = \min\limits_{j:n \geqslant d_j} \{F(n - d_j)\} + 1, n > 0; F(0) = 0;$

计算复杂度

需要计算n个F值，每个值最多需要m次比较，复杂度为$\Theta(mn)$。

实例

计算找零$n = 6$的最小硬币数

$$d_1 = 1, \qquad d_2 = 3, \qquad d_3 = 4$$

计算过程

- $F(n) = \min\limits_{j:n \geqslant d_j} \{F(n - d_j)\} + 1$
- $F(0) = 0, F(1) = \min\{F(1 - d_1)\} + 1 = 1$
- $F(2) = \min\{F(2 - d_1)\} + 1 = 2$
- $F(3) = \min\{F(3 - d_2), F(3 - d_1)\} + 1 = 1$
- $F(4) = \min\{F(4 - d_3), F(4 - d_2), F(4 - d_1)\} + 1 = 1$
- $F(5) = \min\{F(5 - d_3), F(5 - d_2), F(5 - d_1)\} + 1 = 2$
- $F(6) = \min\{F(6 - d_3), F(6 - d_2), F(6 - d_1)\} + 1 = 2$

硬币收集问题

问题描述

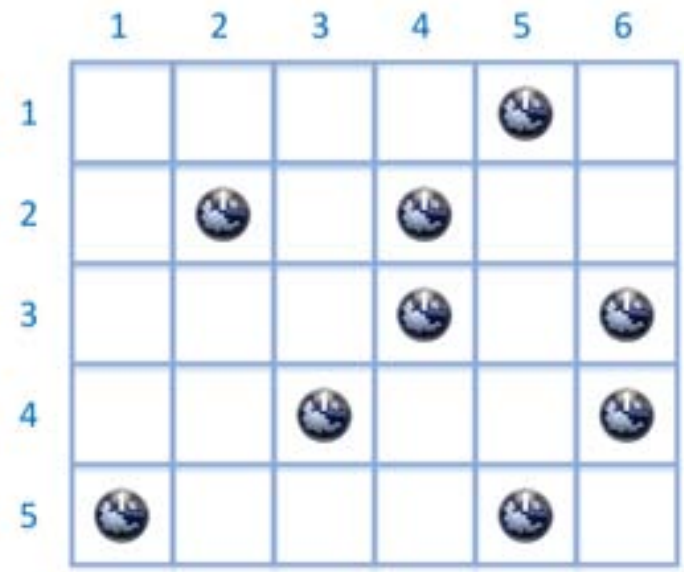
$n*m$的格木板中放有一些硬币，每格的硬币数目最多一个。机器人从木板的左上方移动到右下方（每次只能移动一个格子，并且不能后退），遇到有硬币的单元格就会收集硬币。设计算法找出机器人能够收集最大硬币数的路径。

递归关系

- $F(i,j)$为机器人截止到单元格(i,j)能够收集到的最大硬币数；
- 第一行上方和第一列左边的单元格，假定其硬币数为0；
- $F(i,j)=\max\{F(i-1,j),F(i,j-1)\}+c_{ij}, 1\leqslant i\leqslant n, 1\leqslant j\leqslant m$；
- $F(0,j)=0, 1\leqslant j\leqslant m;\quad F(i,0)=0, 1\leqslant i\leqslant n$；

实例

格木板



动态规划计算

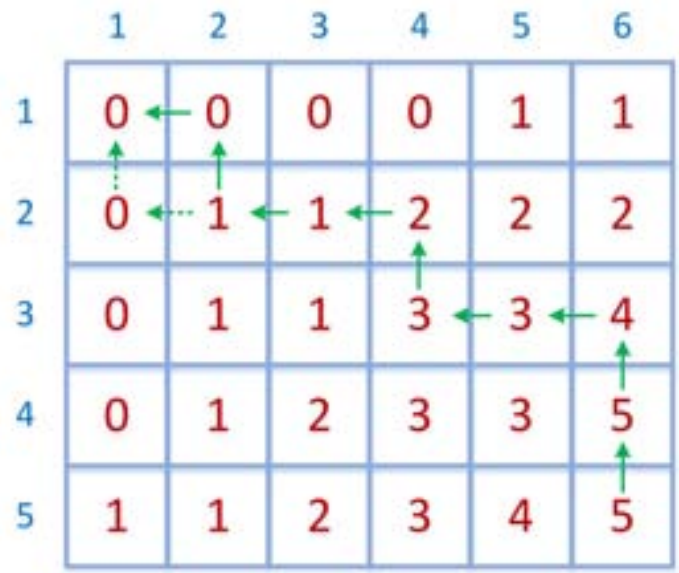
	1	2	3	4	5	6
1	0	0	0	0	1	1
2	0	1	1	2	2	2
3	0	1	1	3	3	4
4	0	1	2	3	3	5
5	1	1	2	3	4	5

计算复杂度

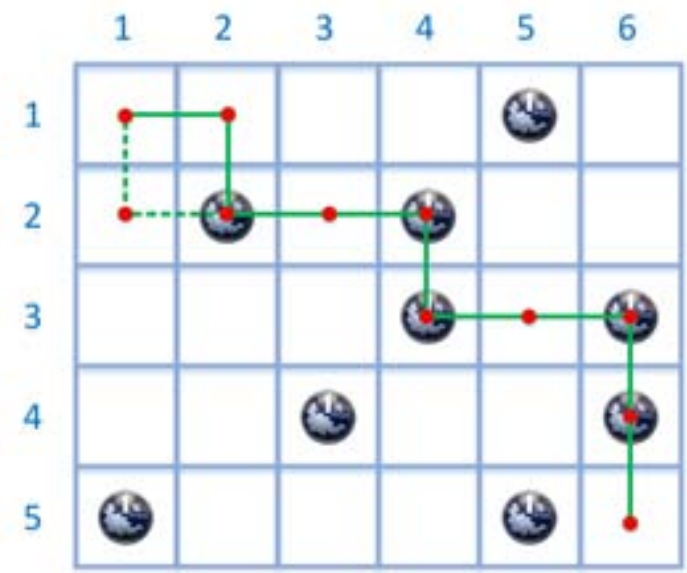
计算每个单元格需要2次比较，共$n*m$个格子，复杂度为$\Theta(nm)$。

实例

回溯过程



最优路径



回溯复杂度

路径的长度为$n+m$，复杂度为$\Theta(n+m)$。

主要内容

背包问题

问题回顾

- 求能够放入给定承重量的背包的最大价值的物品集合；
- 穷举查找法可以得到最优解，但是计算复杂度较高（2^n）；
- 贪心算法计算较快，但是只能得到近似解；

动态规划求解

- $F(i,j)$为前i个物品放进承重量为j的背包问题的最优解；
- 前i个物品构成的子集分为两类：含第i个物品和不含第i个物品；
- $$F(i,j)=\begin{cases}\max\{F(i-1,j),v_i+F(i-1,j-w_i)\}, & j-w_i\geqslant 0\\ F(i-1,j), & j-w_i<0\end{cases}$$
- $F(0,j)=0, j\geqslant 0; F(i,0)=0, i\geqslant 0$；
- 一共需要计算nW个F值，复杂度为$\Theta(n)$；

实例

递推关系

$$F(i,j)=\begin{cases}\max\{F(i-1,j),v_i+F(i-1,j-w_i)\}, & j-w_i\geqslant 0\\ F(i-1,j), & j-w_i<0\end{cases}$$

$$F(0,j)=0, j\geqslant 0; F(i,0)=0, i\geqslant 0$$

实例

物品	重量（w）	价值（v）
1	2	12
2	1	10
3	3	20
4	2	15
背包承重量$W=5$		

$F(i,j)$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

实例

实例

物品	重量（w）	价值（v）
1	2	12
2	1	10
3	3	20
4	2	15
背包承重量$W=5$		

$F(i,j)$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	10	12	22	22	22
3	0	10	12	22	30	32
4	0	10	15	25	30	37

回溯求装入背包的物品

- $F(4,5) > F(3,5)$：物品4装入背包；
- $F(3,5-2) = F(2,5-2)$：物品3不是最优子集一部分；
- $F(2,3) > F(1,3)$：物品2装入背包；
- $F(1,3-1) > F(0,3-1)$：物品1装入背包；

动态规划优化

记忆化

- 自顶向下的递归调用求解会多次求解子问题；
- 自下向上的动态规划过程会求解一些不必要的子问题；
- 结合两种方法，只求解一次必要的子问题（记忆化）；

实例

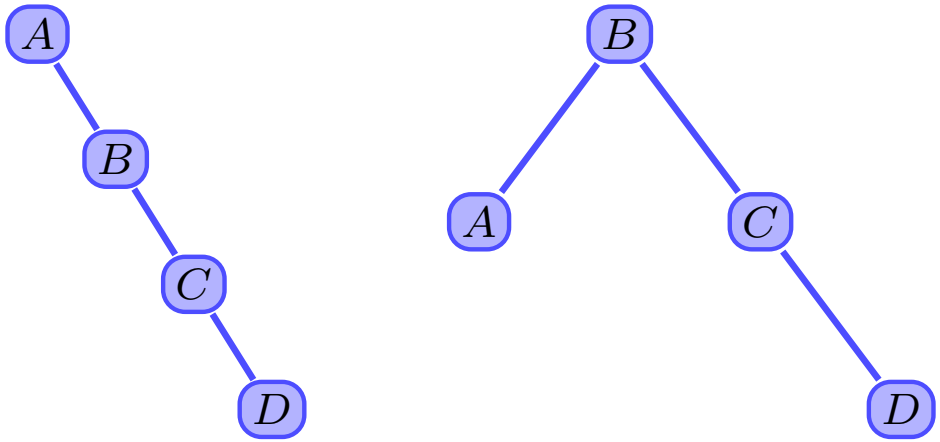
物品	重量（w）	价值（v）
1	2	12
2	1	10
3	3	20
4	2	15
背包承重量$W=5$		

$F(i,j)$	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	12	12	12	12
2	0	–	12	22	–	22
3	0	–	–	22	–	32
4	0	–	–	–	–	37

主要内容

最优二叉查找树

二叉查找树：父节点比左子树的大，比右子树小

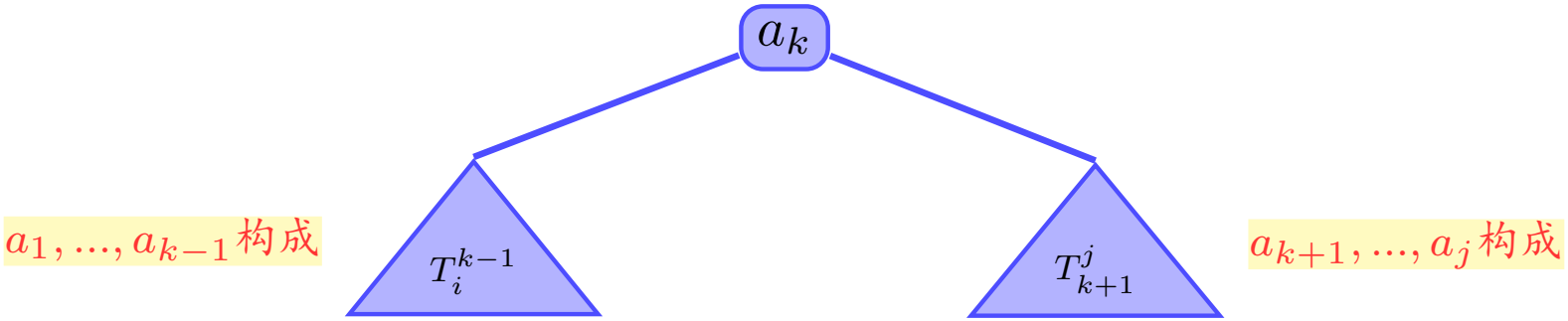


最优二叉查找树

- 假设查找A，B，C，D的概率分别为$0.1, 0.2, 0.4, 0.3$；
- 左边的树平均查找次数：$0.1 * 1 + 0.2 * 2 + 0.4 * 3 + 0.3 * 4 = 2.9$；
- 右边的树平均查找次数：$0.1 * 2 + 0.2 * 1 + 0.4 * 2 + 0.3 * 3 = 2.1$；
- 最优二叉查找树是平均键值比较次数最少的树；

递推关系

假设$a_1, ..., a_n$是从小到大排列的键，概率分别为$p_1, ..., p_n$，T_i^j是由键$a_i, ..., a_j$构成的最优二叉查找树，$C(i,j)$是其查找次数，如何选择键a_k做为T_i^j的根？



$$
\begin{aligned}
C(i,j) &= \min_{i \leqslant k \leqslant j} \{p_k * 1 + \sum_{s=i}^{k-1} p_s * (a_s在T_i^{k-1}层数 + 1) + \sum_{s=k+1}^{j} p_s * (a_s在T_{k+1}^{j}层数 + 1)\} \\
&= \min_{i \leqslant k \leqslant j} \{\sum_{s=i}^{k-1} p_s * (a_s在T_i^{k-1}层数) + \sum_{s=k+1}^{j} p_s * (a_s在T_{k+1}^{j}层数) + \sum_{s=i}^{j} p_s\} \\
&= \min_{i \leqslant k \leqslant j} \{C(i,k-1), C(k+1,j)\} + \sum_{s=i}^{j} p_s
\end{aligned}
$$

实例

递推关系

$$C(i,j) = \min_{i \leqslant k \leqslant j} \{C(i,k-1), C(k+1,j)\} + \sum_{s=i}^{j} p_s$$

$$C(i,i-1) = 0, C(i,i) = p_i$$

实例

键	概率
A(1)	0.1
B(2)	0.2
C(3)	0.4
D(4)	0.3

C	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

根	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					

实例

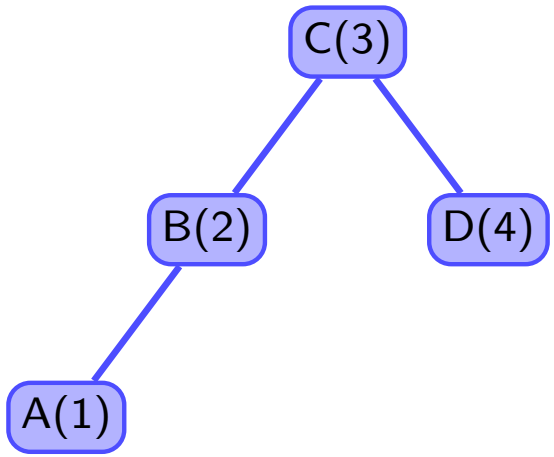
递推关系

$$C(i,j) = \min_{i \leqslant k \leqslant j} \{C(i,k-1), C(k+1,j)\} + \sum_{s=i}^{j} p_s$$

$$C(i,i-1) = 0, C(i,i) = p_i$$

实例

根	0	1	2	3	4
1		1	2	3	3
2			2	3	3
3				3	3
4					4
5					



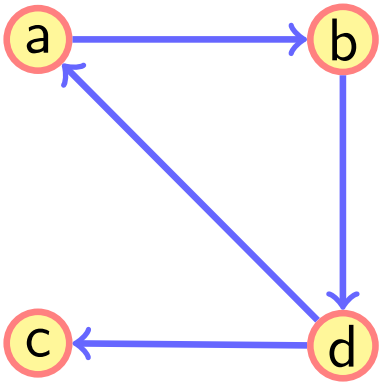
主要内容

传递闭包问题

定义

一个n个顶点的有向图的传递闭包可以定义为一个n阶矩阵$T = \{t_{ij}\}$，如果从第i个顶点到第j个顶点之间存在一条有效的有向路径，则$t_{ij} = 1$，否则为0。

示例



邻接矩阵

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 \end{bmatrix} \end{array}$$

传递闭包

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} \end{array}$$

Warshall算法

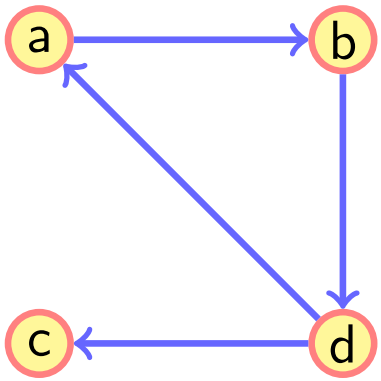
算法要点

- 顶点v_i到v_j路径上顶点的编号都不大于k，则$R^{(k)}(i,j)=1$；
- 路径$a(1)-b(2)-d(3)$：$R^{(2)}(1,3)=1, R^{(3)}(1,3)=1$；
- $R^{(k)}(i,j)=1$时，v_i到v_j路径有两种情况：
  - v_i-{编号$<k$的顶点}-v_j
  - v_i-{编号$<k$的顶点}-v_k-{编号$<k$的顶点}-v_j（假定v_k只出现1次）
- 即$\{R^{(k-1)}(i,j)=1\}$或$\{R^{(k-1)}(i,k)=1, R^{(k-1)}(k,j)=1\}$

递推关系

$R^{(k-1)}(i,j)=1$或$\{R^{(k-1)}(i,k)=1, R^{(k-1)}(k,j)=1\} \Rightarrow R^{(k)}(i,j)=1$

实例



$\mathbf{R}^{(0)}$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

$\mathbf{R}^{(1)}$

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

$\mathbf{R}^{(2)}$

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

$\mathbf{R}^{(3)}$

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

$\mathbf{R}^{(4)}$

	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

主要内容

完全最短路径问题

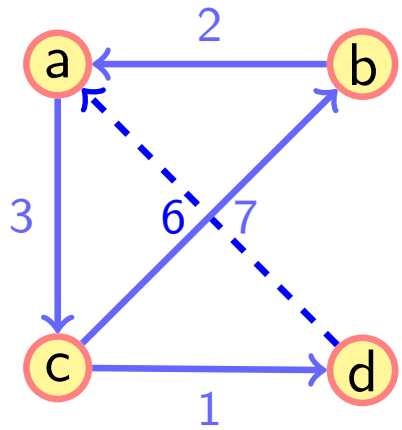
单源最短路径问题（回顾）

给定一个起点，计算起点到所有其他顶点之间的最短路径。

完全最短路径问题

找到从每个顶点到其他所有顶点之间的最短路径。

示例



邻接矩阵$\mathbf{W}$

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & - & 3 & - \\ 2 & 0 & - & - \\ - & 7 & 0 & 1 \\ 6 & - & - & 0 \end{bmatrix} \end{array}$$

距离矩阵$\mathbf{D}$

$$\begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ 7 & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

Floyd算法

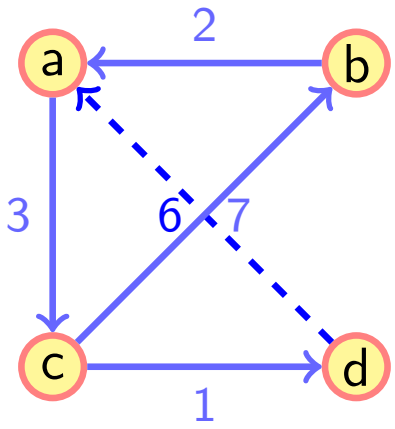
算法要点

- $D^{(k)}(i,j)$表示第i个顶点到第j个顶点的最短路径长度，并且路径的中间顶点编号不大于k;
- 第i个顶点到第j个顶点的最短路径有两种情况:
  - v_i-{编号$< k$的顶点}-v_j
  - v_i-{编号$< k$的顶点}-v_k-{编号$< k$的顶点}-v_j
- 第i个顶点到第j个顶点的最短路径为两种情况下的最小值;

递推关系

$$D^{(k)}(i,j) = min\{D^{(k-1)}(i,k) + D^{(k-1)}(k,j), D^{(k-1)}(i,j)\}$$

实例



$$\mathbf{D}^{(0)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & - & 3 & - \\ 2 & 0 & - & - \\ - & 7 & 0 & 1 \\ 6 & - & - & 0 \end{bmatrix} \end{array}$$

$$\mathbf{D}^{(1)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & - & 3 & - \\ 2 & 0 & \mathbf{5} & - \\ - & 7 & 0 & 1 \\ 6 & - & \mathbf{9} & 0 \end{bmatrix} \end{array}$$

$$\mathbf{D}^{(2)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & - & 3 & - \\ 2 & 0 & 5 & - \\ \mathbf{9} & 7 & 0 & 1 \\ 6 & - & 9 & 0 \end{bmatrix} \end{array}$$

$$\mathbf{D}^{(3)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & \mathbf{10} & 3 & \mathbf{4} \\ 2 & 0 & 5 & \mathbf{6} \\ 9 & 7 & 0 & 1 \\ 6 & \mathbf{16} & 9 & 0 \end{bmatrix} \end{array}$$

$$\mathbf{D}^{(4)} = \begin{array}{c} \\ a \\ b \\ c \\ d \end{array} \begin{array}{c} \begin{array}{cccc} a & b & c & d \end{array} \\ \begin{bmatrix} 0 & 10 & 3 & 4 \\ 2 & 0 & 5 & 6 \\ \mathbf{7} & 7 & 0 & 1 \\ 6 & 16 & 9 & 0 \end{bmatrix} \end{array}$$

主要内容

矩阵链乘计算问题

问题描述

- 设$A_1, A_2, ..., A_n$为n个矩阵的序列，其中A_i为$P_{i-1} * P_i$阶矩阵，可用向量$P =< P_0, P_1, P_2, ..., P_n >$表示矩阵链的输入规模；
- 计算$A_1, A_2, ..., A_n$的乘积，不同的计算顺序，计算量有所不同，选择计算量最少的次序计算矩阵乘积；

示例

- 若A_1, A_2分别是$i * j, j * k$的矩阵，计算A_1A_2需要$i * j * k$次乘法；
- 如有$P =< 10, 100, 5, 50 >$，计算$A_1A_2A_3$的乘法次数分别为：
  - $(A_1A_2)A_3$：$10 * 100 * 5 + 10 * 5 * 50 = 7500$
  - $A_1(A_2A_3)$：$10 * 100 * 50 + 100 * 5 * 50 = 75000$

动态规划求解

递推关系

- $A_iA_{i+1}...A_j = (A_iA_{i+1}...A_k)(A_{k+1}A_{k+2}...A_j)$;
- 假设$m[i,j]$为计算乘积$A_iA_{i+1}...A_j$所用的最少运算次数，有：
  - $i = j, m[i,j] = 0, i = j$
  - $i < j, m[i,j] = \min_{i \leqslant k < j}\{m[i,k] + m[k+1,j] + P_{i-1}P_kP_j\}$

实例：$P =< 30, 35, 15, 5, 10, 20 >$

- $A_1 : 30 * 35, A_2 : 35 * 15, A_3 : 15 * 5, A_4 : 5 * 10, A_5 : 10 * 20$
- $m[1,2] = 30 * 35 * 15 = 15750,\ \ m[2,3] = 35 * 15 * 5 = 2625$
- $m[3,4] = 15 * 5 * 10 = 750,\ \ m[4,5] = 5 * 10 * 20 = 1000$
- $m[1,3] = min\{m[1,2] + 30 * 15 * 5, m[2,3] + 30 * 35 * 5\} = 7875$
- $m[2,4] = min\{m[2,3] + 35 * 5 * 10, m[3,4] + 35 * 15 * 10\} = 4375$

动态规划求解

实例： $P =< 30, 35, 15, 5, 10, 20 >$

- $m[3,5] = min\{m[3,4] + 15 * 10 * 20, m[4,5] + 15 * 5 * 20\} = 2500$
- $m[1,4] = min\{m[2,4] + 30 * 35 * 10, m[1,2] + m[3,4] + 30 * 15 * 10, m[1,3] + 30 * 5 * 10\} = 9375$
- $m[2,5] = min\{m[3,5] + 35 * 15 * 20, m[2,3] + m[4,5] + 35 * 5 * 20, m[2,4] + 35 * 10 * 20\} = 7125$
- $m[1,5] = min\{m[2,5] + 30 * 35 * 20, m[1,2] + m[3,5] + 30 * 15 * 20, m[1,3] + m[4,5] + 30 * 5 * 20, m[1,4] + 30 * 10 * 20\} = 11875$

回溯求解

最佳计算次序：$A_1A_2A_3A_4A_5 = (A_1(A_2A_3))(A_4A_5)$

主要内容

最长递增子序列

问题描述

在一个给定的数值序列中，找到一个子序列，使得这个子序列元素的数值依次递增，并且这个子序列的长度尽可能地大。最长递增子序列中的元素在原序列中不一定是连续的。

示例

- 原始序列：$\{0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15\}$
- 最长递增子序列：$\{0, 2, 6, 9, 11, 15\}$

注意

$\{0, 2, 6, 9, 13, 15\}$,$\{0, 4, 6, 9, 11, 15\}$,$\{0, 4, 6, 9, 13, 15\}$也是问题的解。

动态规划求解思路

递推关系

- 假设$A[0,1,...,n-1]$表示输入序列，其中n表示问题序列的长度；
- L_i表示以$A[i]$结尾的最长递增子序列的长度；
- 求最长递增子序列问题：$max\{L_i\}, 0 \leqslant i \leqslant n-1$；
- 对于$\forall k$，$k < i$，$L_i = max\{L_k | A[k] < A[i]\} + 1$；
- 对于$\forall k$，$k < i$，若$A[k] \geqslant A[i]$ for all，则$L_i = 1$；
- 初始条件：$L_0 = 1$；

实例

- $A = \{0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15\}$
- $L_0 = 1, L_{15} = ?$

求解过程

实例

- $A = \{0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15\}$
- $i = 0; \Rightarrow L_0 = 1$
- $i = 1; k = 0; \Rightarrow L_1 = max\{L_k\} + 1 = 2$
- $i = 2; k = 0; \Rightarrow L_2 = max\{L_k\} + 1 = 2$
- $i = 3; k = 0, 1, 2; \Rightarrow L_3 = max\{L_k\} + 1 = 3$
- $i = 4; k = 0; \Rightarrow L_4 = max\{L_k\} + 1 = 2$
- $i = 5; k = 0, 1, 2, 4; \Rightarrow L_5 = max\{L_k\} + 1 = 3$
- $i = 6; k = 0, 2, 4; \Rightarrow L_6 = max\{L_k\} + 1 = 3$
- $i = 7; k = 0, ..., 6; \Rightarrow L_7 = max\{L_k\} + 1 = 4$

求解过程

实例

- $A = \{0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15\}$
- $i = 8; k = 0; \Rightarrow L_8 = max\{L_k\} + 1 = 2$
- $i = 9; k = 0, 1, 2, 4, 6, 8; \Rightarrow L_9 = max\{L_k\} + 1 = 4$
- $i = 10; k = 0, 2, 4, 8; \Rightarrow L_{10} = max\{L_k\} + 1 = 4$
- $i = 11; k = 0, ..., 6, 8, 9, 10; \Rightarrow L_{11} = max\{L_k\} + 1 = 5$
- $i = 12; k = 0, 4, 8; \Rightarrow L_{12} = max\{L_k\} + 1 = 3$
- $i = 13; k = 0, 1, 2, 4, 5, 6, 8, 9, 10, 12; \Rightarrow L_{13} = max\{L_k\} + 1 = 5$
- $i = 14; k = 0, 2, 4, 6, 8, 10, 12; \Rightarrow L_{14} = max\{L_k\} + 1 = 5$
- $i = 15; k = 0, ..., 14; \Rightarrow L_{15} = max\{L_k\} + 1 = 6$

求解过程

回溯求解

- $A = \{0, 8, 4, 12, 2, 10, 6, 14, 1, 9, 5, 13, 3, 11, 7, 15\}$
- $L_{15} = 6; \Rightarrow \{15\}$
- $L_{11}, L_{13}; \Rightarrow \{13, 15\}, \{11, 15\}$
- $L_{11} : L_9; \Rightarrow \{9, 13, 15\}$
- $L_{11} : L_9 : L_6; \Rightarrow \{6, 9, 13, 15\}$
- $L_{11} : L_9 : L_6 : L_2, L_4; \Rightarrow \{4, 6, 9, 13, 15\}, \{2, 6, 9, 13, 15\}$
- $L_{11} : L_9 : L_6 : L_2, L_4 : L_0; \Rightarrow \{0, 4, 6, 9, 13, 15\}, \{0, 2, 6, 9, 13, 15\}$
- $L_{13} : L_9; \Rightarrow \{9, 11, 15\}$
- $L_{13} : L_9 : L_6 : L_2, L_4 : L_0; \Rightarrow \{0, 4, 6, 9, 11, 15\}, \{0, 2, 6, 9, 11, 15\}$

主要内容

编辑距离问题

问题描述

对于序列S和T，求从S变为T最少需要几步，其中每一步只能进行以下三个操作：1，删除一个字符；2，插入一个字符；3，改变一个字符。将S变为T的最小操作计数就是两者的编辑距离。

示例

- $S = ABC$；$T = CBCD$
- S需要执行的操作：替换A为C，末尾插入D，距离为2；
- $S = ABC$；$T = DCB$
- S需要执行的操作：替换A为D，删除B，末尾插入B，距离为3；

动态规划求解思路

递推关系

- $D[i,j]$表示$S[0,..,i] \rightarrow T[0,...,j]$的编辑距离。
- $S[i] = T[j]$时，不需要操作，即$D[i,j] = D[i-1,j-1]$
- $S[i] \neq T[j]$时，可以执行如下的操作：
  - $S[0,..,i] \rightarrow T[0,...,j-1]$，$T[j]$插入到$S$末尾，$D[i][j-1]+1$
  - 删除$S[i]$，$S[0,..,i-1] \rightarrow T[0,...,j]$，$D[i-1][j]+1$
  - $S[0,..,i-1] \rightarrow T[0,...,j-1]$，替换$S[i]$为$T[j]$，$D[i-1][j-1]+1$
- $D[i][j] = min(D[i][j-1]+1, D[i-1][j]+1, D[i-1][j-1]+1)$

实例

计算$S(algorithm)$和$T(altruistic)$的编辑距离。

求解过程

$D[i,j]$矩阵

S \ T		1	2	3	4	5	6	7	8	9	10
		a	l	t	r	u	i	s	t	i	c
1	a	0	1	2	3	4	5	6	7	8	9
2	l	1	0	1	2	3	4	5	6	7	8
3	g	2	1	1	2	3	4	5	6	7	8
4	o	3	2	2	2	3	4	5	6	7	8
5	r	4	3	3	2	3	4	5	6	7	8
6	i	5	4	4	3	3	3	4	5	6	7
7	t	6	5	4	4	4	4	4	4	5	6
8	h	7	6	5	5	5	5	5	5	5	7
9	m	8	7	7	6	6	6	6	6	6	6

时间复杂度

假如S的长度为n，T的长度为m，计算$D[i,j]$的复杂度为$m*n$。

小结

- 动态规划方法是对一种具有交叠子问题进行求解的技术，与递归调用不同，动态规划对较小的子问题只求解一次；
- 动态规划方法要求最优问题满足的法则：该问题的任何实例的最优解是由该实例的子实例的最优解组成。
- 记忆功能技术试图结合自顶向下和自底向上的优势，对必要的子问题求解一次；
- 求传递闭包的Warshall算法和求最短路径问题的Floyd算法都基于同一种动态规划思想；
- 找零问题，最优二叉查找树，矩阵链乘计算，最长递增子序列，编辑距离等问题均可以采用动态规划求解；

作业

- 8.1: 1 $Page224$
- 8.1: 5 $Page224$
- 8.1: 6 $Page225$
- 8.1: 11 $Page225$

Part VI

课程总结

主要内容

1. 知识点：算法概念
2. 知识点：算法的效率分析
3. 知识点：递归分析
4. 知识点：分治策略
5. 知识点：贪心算法

主要内容

算法的概念与特点

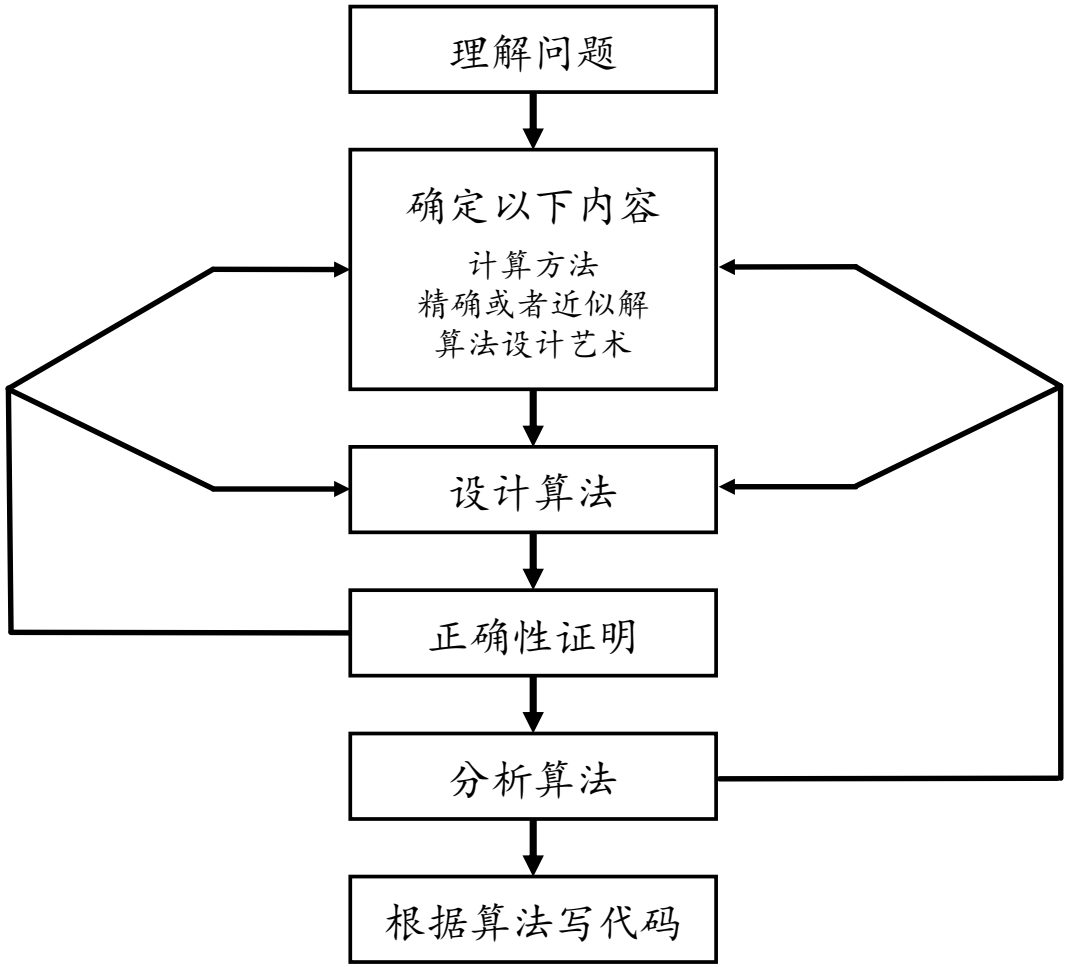
概念

算法是一些列解决问题的明确指令，也就是说，对于符合一定规范的输入，能够在有限时间内获得要求的输出。

特点

- 明确性与可行性：算法的步骤都是确切的，且能有效地执行；
- 有限性：算法必须在有限的步骤内完成；
- 一般性：算法必须可以解决一类问题；
- 有序性：算法从初始步骤开始，逐步的执行解决问题；
- 不唯一性：求解某一个问题的解法不一定是唯一的；

算法设计的一般流程



主要内容

算法的效率

算法的效率分析包括两个方面

- 时间效率：讨论算法运行的有多快（重点）；
- 空间效率：关心算法占用的存储空间；

时间效率度量因子：输入规模

- 几乎所有的算法，运行时间都随着输入规模的增大而增大；
- 算法效率是输入规模的函数，应当恰当地选择输入规模的度量；

时间效率度量因子：运行时间

- 以秒等单位度量的程序运行时间依赖于计算机的性能；
- 基本操作（加减乘除，除法最耗时，其次是乘法）次数对算法的总运行时间贡献最大；

算法的效率

增长次数

- 考虑两个算法，计算复杂度分别为$T_1(n) = n$，$T_2(n) = n!$，当n等于1和2的时候，二者的运行时间相同，它们的计算效率是否相同？
- 小规模的输入不足以将高效的算法和低效的算法区分出来，必须考虑较大规模的输入；
- 需要知道运行时间随着输入规模的增长而增加的幅度；

算法三种效率

- 最差效率$C_{worst}(n)$：输入规模为n时，算法在最坏输入下的效率；
- 最优效率$C_{best}(n)$：输入规模为n时，算法在最好输入下的效率；
- 平均效率$C_{avg}(n)$：输入规模为n时，算法在随机输入下的效率；

增长次数描述：渐进符号

上界O的定义

如果存在大于0的常数c和非负整数n_0，使得$\forall n > n_0$，$t(n) \leqslant cg(n)$，我们称函数$t(n)$包含在$O(g(n))$中，记作$t(n) \in O(g(n))$。

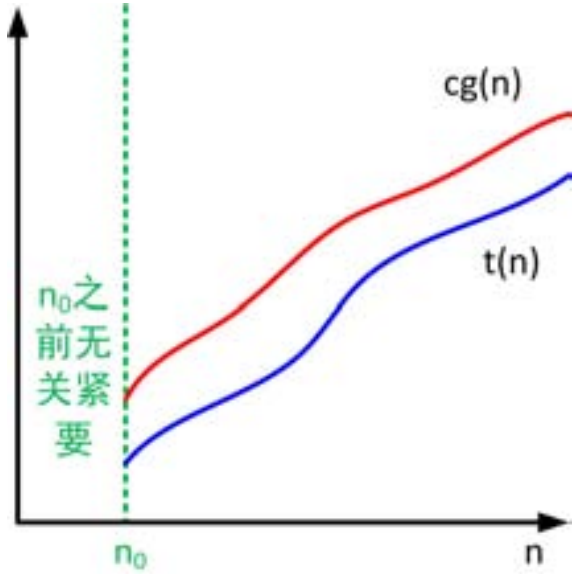
例子

$$n \in O(n), n \in O(n^2), n \in O(n^3)$$

$$100n + 5 \in O(n^2)$$

$$0.5n(n+1) \in O(n^2)$$

$$n^2 \notin O(n), n^3 \notin O(n^2)$$



增长次数描述：渐进符号

下界Ω的定义

如果存在大于0的常数c和非负整数n_0，使得$\forall n > n_0$，$t(n) \geqslant cg(n)$，我们称函数$t(n)$包含在$\Omega(g(n))$中，记作$t(n) \in \Omega(g(n))$。

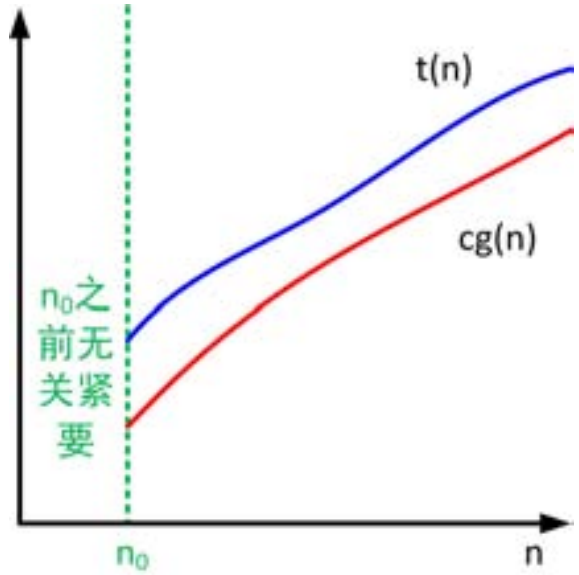
例子

$$n^2 \in \Omega(n)$$

$$0.5n(n-1) \in \Omega(n^2)$$

$$n^3 \in \Omega(n), n^3 \in \Omega(n^2)$$

$$100n + 5 \notin \Omega(n^2)$$



增长次数描述：渐进符号

Θ的定义

若存在大于0的常数c_1,c_2和非负整数n_0,$\forall n > n_0$,$c_1 g(n) \leqslant t(n) \leqslant c_2 g(n)$,
我们称函数$t(n)$包含在$\Theta(g(n))$中，记作$t(n) \in \Theta(g(n))$。

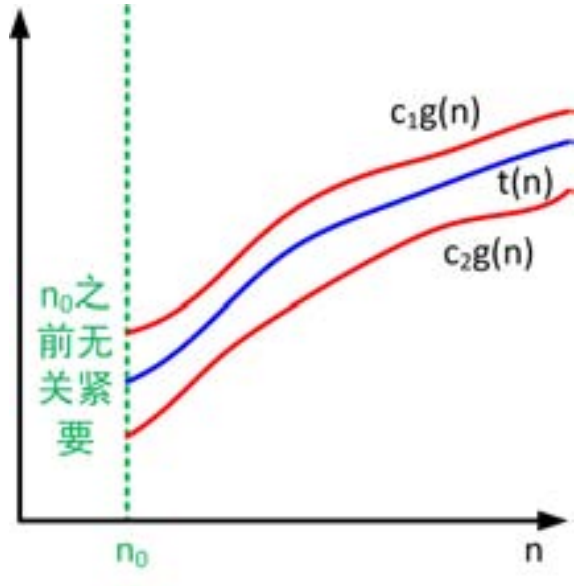
例子

$$100n^2 \in \Theta(n^2)$$

$$0.5n(n-1) \in \Theta(n^2)$$

$$0.5n(n-1) \notin \Theta(n^3)$$

$$0.5n(n-1) \notin \Theta(n)$$



增长次数分析

定理

如果$t_1(n) \in O(g_1(n))$，并且$t_2(n) \in O(g_2(n))$，则

$$t_1(n) + t_2(n) \in O(max\{g_1(n), g_2(n)\}).$$

对于Ω和Θ符号，类似的断言也成立。

定理

设f和g是定义域为自然数集合N上的非负函数，有：

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0 & : \ t(n) \in O(g(n)) \\ c > 0 & : \ t(n) \in O(g(n)); t(n) \in \Omega(g(n)); t(n) \in \Theta(g(n)) \\ \infty & : \ t(n) \in \Omega(g(n)) \end{cases}$$

基本效率类型

类型	名称	注释
1	常数	为数很少的效率最高的算法，难以举例。
$logn$	对数	算法的每一次循环都会消去问题规模的一个常数因子，如有序序列中的折半查找算法。
n	线性	扫描规模为n的列表，如顺序查找算法。
$nlogn$	线性对数	许多分治算法（合并排序，快速排序）的平均效率属于这个类型。
n^2	平方	一般来说，包含两重嵌套循环算法的典型效率，选择排序和冒泡排序属于这一类型。
n^3	立方	一般来说，包含三重嵌套循环算法的典型效率。
2^n	指数	求n个元素集合的所有子集的算法，如蛮力查找求解背包问题和最近对问题。
$n!$	阶乘	求n个元素的完全排列算法，如穷举查找求解分配问题，旅行商问题。

递归分析基础

递归的概念

- 对于任意非负整数n，计算阶乘函数$F(n) = n!$的值；
- 当$n > 1$时，$n! = 1 * 2 * ... * (n-1) * n = (n-1)! * n$，并且$n! = 1$；
- 可以使用递归的方法计算$F(n) = F(n-1) * n$。

乘法执行次数$M(n)$

- $n > 0$时，$M(n) = M(n-1) + 1$
- $n = 0$时，$M(0) = 0$

递归方程求解

- 反向替换法求解
- 主定理求解

递归方程求解–替换法

例1：计算$W(n)$，$W(n) = W(n-1) + n - 1, W(1) = 0$

$$
\begin{aligned}
W(n) &= W(n-1) + n - 1 = W(n-2) + (n-2) + (n-1) \\
&= W(n-3) + (n-3) + (n-2) + (n-1) = ... \\
&= W(1) + 1 + 2 + ... + (n-2) + (n-1) = 1 + 2 + ... + (n-2) + (n-1) \\
&= n(n-1)/2
\end{aligned}
$$

例2：计算$W(n)$，$W(n) = 2W(n/2) + n - 1, n = 2^k, W(1) = 0$

$$
\begin{aligned}
W(n) &= 2W(2^{k-1}) + 2^k - 1 = 2[2W(2^{k-2}) + 2^{k-1} - 1] + 2^k - 1 \\
&= 2^2 W(2^{k-2}) + 2^k - 2 + 2^k - 1 \\
&= 2^2[2W(2^{k-3}) + 2^{k-2} - 1] + 2^k - 2 + 2^k - 1 \\
&= 2^3 W(2^{k-3}) + 2^k - 2^2 + 2^k - 2 + 2^k - 1 = ... \\
&= 2^k W(1) + k2^k - (2^{k-1} + 2^{k-2} + ... + 2 + 1) = k2^k - 2^k + 1 \\
&= nlogn - n + 1
\end{aligned}
$$

递归方程求解–主定理

主定理

设$T(n)$是一个非递减函数（定义见课本P376），并且满足递推式

$$T(n) = aT(n/b) + f(n)，\ 其中n = b^k，\ k = 1, 2, ...$$

$$T(1) = c$$

其中$a \geqslant 1$，$b \geqslant 2$，$c > 0$，如果$f(n) \in \Theta(n^d)$，$d \geqslant 0$，那么

- 当$a < b^d$时，$T(n) \in \Theta(n^d)$；
- 当$a = b^d$时，$T(n) \in \Theta(n^d logn)$；　　**结论对符号$O$和$\Omega$也成立。**
- 当$a > b^d$时，$T(n) \in \Theta(n^{log_b a})$；

注释

主定理的推导过程见课程辅助材料（注意等比数列求和的简单分析方法）。

主要内容

分治算法

分治法

1. 将一个问题划分为同一类型的若干子问题，子问题最好规模相同。
2. 对这些子问题求解（通常使用递归的方式）。
3. 如果有必要，合并这些子问题的解，以得到原问题的解。

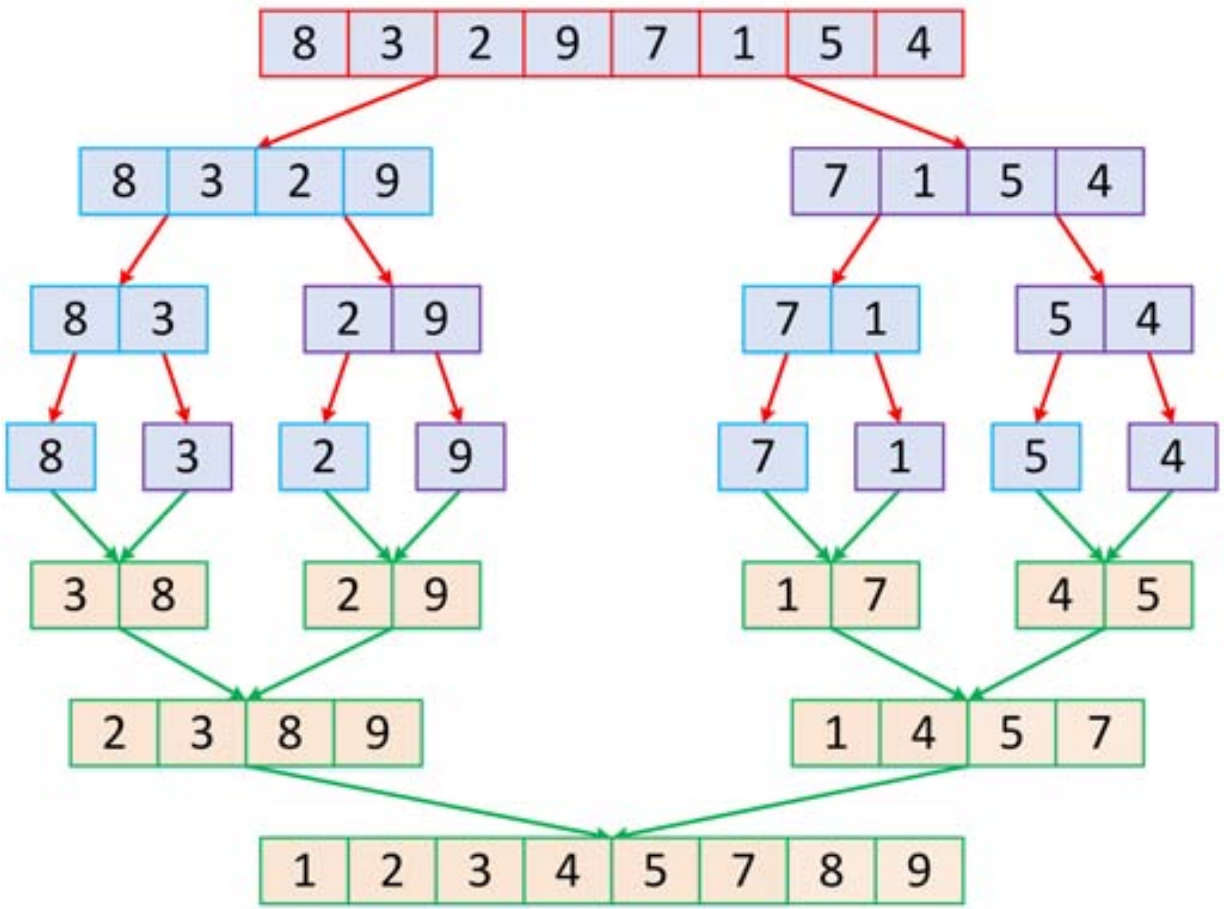
例子

$$a_1+a_2+\ldots+a_{n-1}+a_n=(a_0+a_1+\ldots+a_{n/2-1})+(a_{n/2}+a_{n/2+1}+\ldots+a_{n-1})$$

复杂度分析

- $T(n) = aT(n/b) + f(n)$
- 主定理或者反向替换法求解；

合并排序


8 3 2 9 7 1 5 4
8 3 2 9
7 1 5 4
8 3
2 9
7 1
5 4
8
3
2
9
7
1
5
4
3 8
2 9
1 7
4 5
2 3 8 9
1 4 5 7
1 2 3 4 5 7 8 9

效率分析

- 算法基本操作：元素之间比较；
- 比较次数$C(n)$的递推关系：

$$C(n) = 2C(n/2) + C_{merge}(n)，\ C(0) = 1，\ 其中n = 2^k；$$

- 最坏情况下，$C_{merge}(n) = n - 1$；
- 最好情况下，$C_{merge}(n) = n/2$；
- 根据主定理可得$C_{worst}(n) \in \Theta(nlogn)$；
- 根据主定理可得$C_{best}(n) \in \Theta(nlogn)$；
- 合并排序在最坏情况下的比较次数十分接近基于比较的排序算法的理论上能够达到的最少次数；
- 合并排序的主要缺点是需要线性的额外空间。

快速排序

算法思想

- 合并排序按照元素的位置划分并合并的方式进行排序。划分过程很快，主要工作在合并子问题。
- 考虑如果按照元素的值进行划分子问题：

$$\underbrace{A[0]...A[s-1]}_{<A[s]} \quad A[s] \quad \underbrace{A[s+1]...A[n-1]}_{>A[s]}$$

- 子问题递归排序后，合并很简单，主要工作在如何划分。

子问题划分简单方式

根据首元素划分数组（霍尔划分方法）。

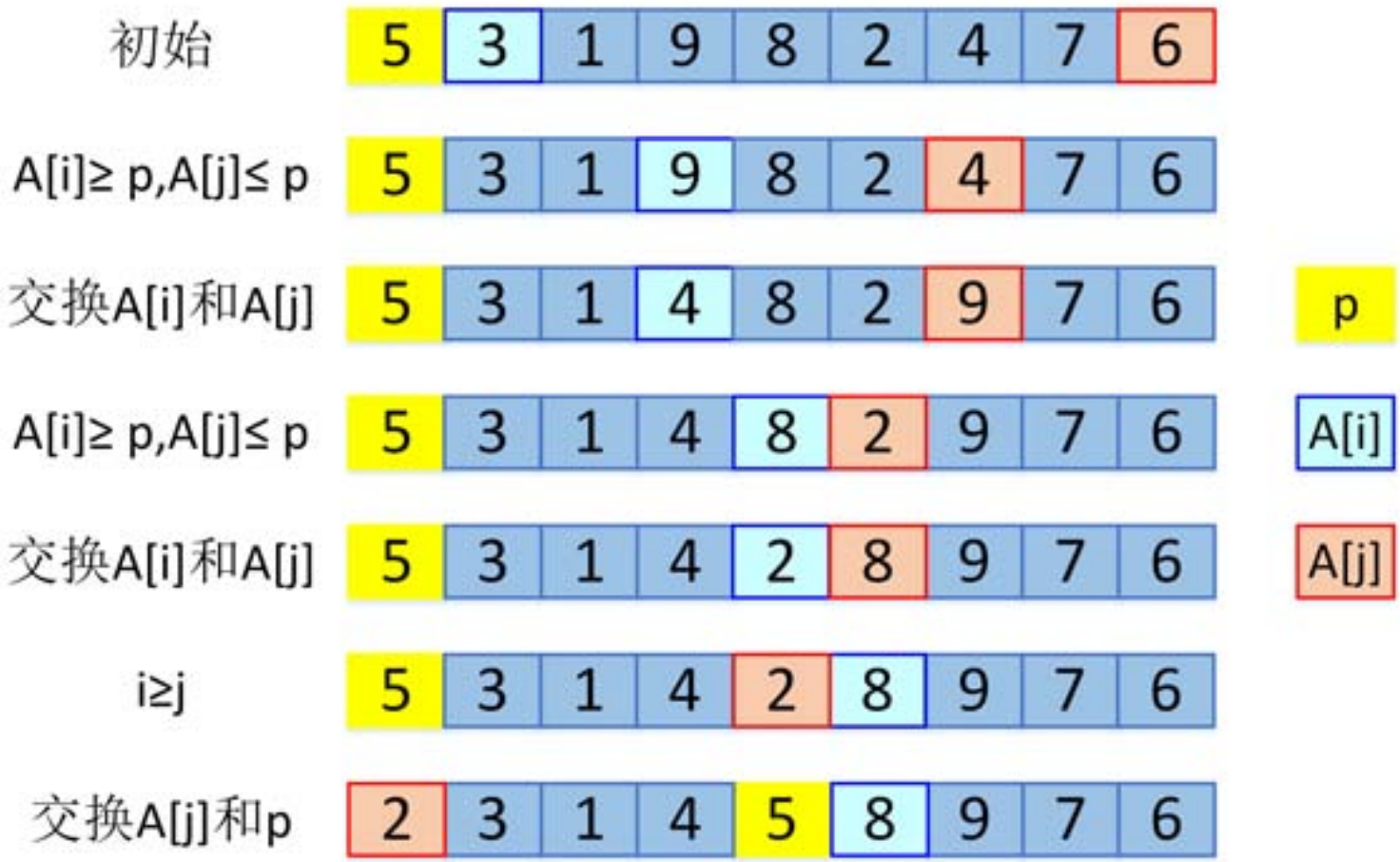
霍尔划分方法

- 比划分值p小的元素位于左边，大的位于右边，简单地$p = A[0]$;
- 从左到右的扫描从第二个元素开始，直到$A[i] \geqslant p$;
- 从右到左的扫描从最后的元素开始，直到$A[j] \leqslant p$;
- 两边的扫描都停止后，会有下面三种情况：
  1. $i < j$，则$A[i] \leftrightarrow A[j]$;

  2. $i > j$，则$A[j] \leftrightarrow A[0]$;

  3. $i = j$，则$A[i] = A[j] = p$;

例子


初始
5 3 1 9 8 2 4 7 6
A[i]≥ p,A[j]≤ p
5 3 1 9 8 2 4 7 6
交换A[i]和A[j]
5 3 1 4 8 2 9 7 6
p
A[i]≥ p,A[j]≤ p
5 3 1 4 8 2 9 7 6
A[i]
交换A[i]和A[j]
5 3 1 4 2 8 9 7 6
A[j]
i≥j
5 3 1 4 2 8 9 7 6
交换A[j]和p
2 3 1 4 5 8 9 7 6

算法效率分析

最差情况

- 考虑对一个已经排好序的序列$[1,2,3,4,5,6,7,8,9.10]$进行排序；
- 从左到右扫描，停止在$A[1]$；从右到左扫描，停止在$A[0]$；
- $A[0]$和本身交换，然后继续递归排序$[2,3,4,5,6,7,8,9.10]$；
- 第1次分裂需要的比较次数为$1+n$；
- 第2次分裂需要的比较次数为n;
- 最后一次分裂比较3次（$A[i]$比较1次，$A[j]$比较2次）；
- $C_{worst}(n) = (n+1)+n+...+3 = (n+1)(n+2)/2-3 \in \Theta(n^2)$

算法效率分析

最优情况

- 最优情况下，数组序列被分为两个大小差不多规模的子序列；
- $C_{best}(n) = 2C_{best}(n/2) + \Theta(n), C_{best}(1) = 0$；
- 根据主定理可得$C_{best}(n) \in \Theta(nlog_2n)$；

平均情况

- 经过$n+1$次比较，分裂点出现在任意的位置$s(0 \leqslant s \leqslant n-1)$，划分数组为大小为$s$和$n-1-s$两个部分，每种情况出现的概率为$1/n$；
- 递推方程：

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)], C_{avg}(1) = C_{avg}(0) = 0$$

- 计算可得$C_{avg}(n) \approx 2nln2 \approx 1.39nlog_2n$；
- 算法平均情况下的效率仅比最优情况下仅仅多执行39%的计算；

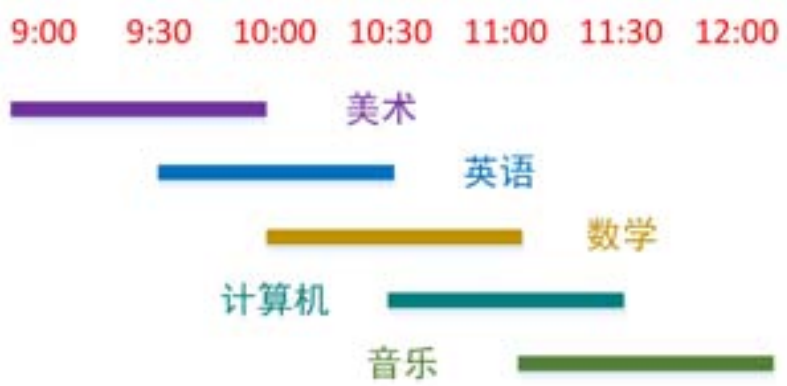
主要内容

教室调度问题

问题描述

假设有如下课表，你希望尽可能多的课程安排在某间教室上。

课程	开始时间	结束时间
美术	9:00	10:00
英语	9:30	10:30
数学	10:00	11:00
计算机	10:30	11:30
音乐	11:00	12:00



解决办法

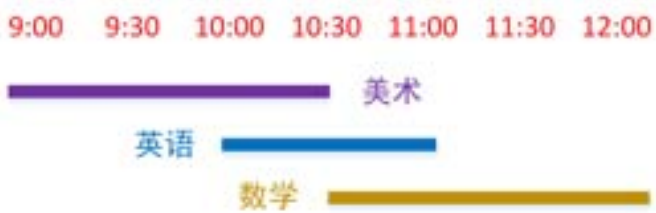
在时间不冲突的情况下，每一次都选择最早结束的课程。

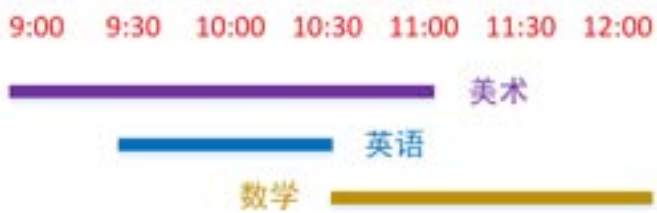
美术	9:00	10:00
数学	10:00	11:00
音乐	11:00	12:00

贪心策略分析

教室调度问题分析

- 选择最早结束的课程的策略可以找到教室调度问题最优解；
- 选择最小占用时间或者最早开始的课程的得不到最优解；





贪心算法特点

- 可行性：贪心算法每一步选择都必须满足问题的约束；
- 局部最优：算法不考虑全局最优，仅做出当前看来最优的选择；
- 不可取消：一旦做出选择，再算法后面的步骤中无法改变；

Prim算法

生成树

连通图的一棵生成树是包含图的所有顶点的连通无环子图（一棵树）。

最小生成树

加权连通图的一棵最小生成树是图的一棵权重最小（T_1）的生成树。

Prim算法流程

1. 任意选一点v_0，集合V被分割成两个集合$V_T = \{v_0\}$和$V - V_T$；
2. 从连通V_T和$V - V_T$的边中挑选一条权重最小的边$e^* = (v^*, u^*)$，$v^* \in V_T, u^* \in V - V_T$，将$u^*$加入$V_T$中，从$V - V_T$删除$u^*$；
3. 重复步骤2，直到集合$V - V_T$为空；

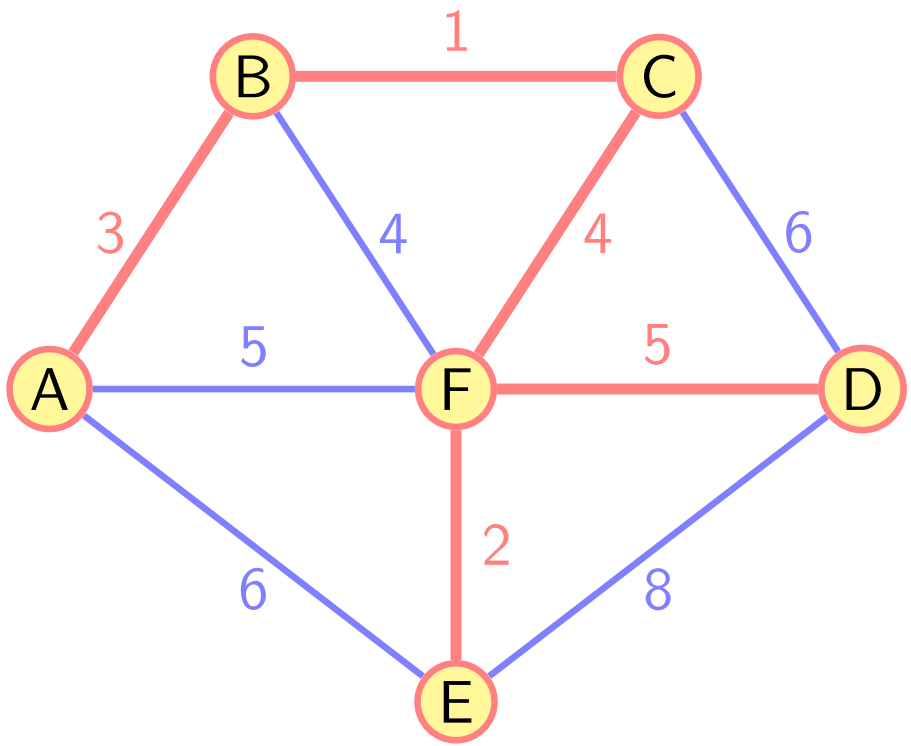
算法实例

算法核心

从连通V_T和$V - V_T$的边中挑选一条权重最小的边。

计算过程

- $V_T = \{A, B, C, F, E, D\}$
- $V - V_T = \{\varnothing\}$
- $W(T) = 15$



Kruskal算法

算法思想

算法贪婪地选择最小权重的边，扩展无环的子图构造最小生成树。

算法流程

1. 按照权重非递减顺序对图中的边E进行排序；
2. 扫描以排序的列表，如果下一条边加入到当前的子图中不导致一个回路，则加入该边到子图中，否则跳过该边；
3. 重复步骤2，直到子图中有$|V|-1$条边；

算法的另一种解释

Kruskal算法可看作是对包含给定图所有顶点和某些边的一系列森林所做的连续动作。初始森林是$|V|$颗只包含一个顶点的树，通过加入边，将小数连通成大树，最终构造出一棵最小生成树。

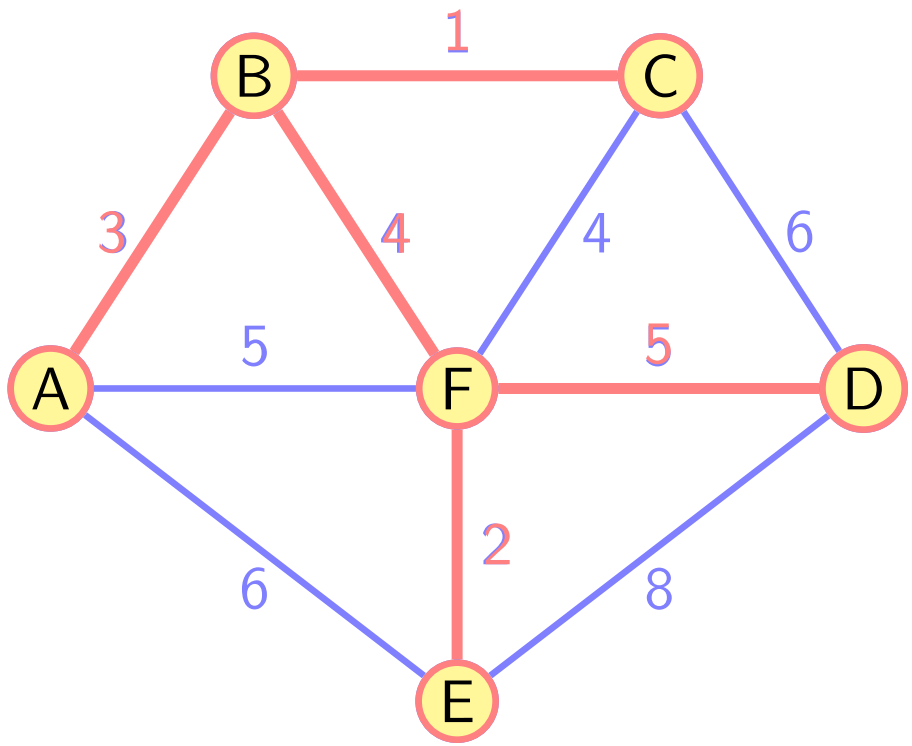
算法实例

算法核心

算法贪婪地选择最小权重的边构造生成树，所选择的边不能构成回路。

计算过程

- $(A, B) = 3$
- $(A, F) = 5$
- $(A, E) = 6$
- $(B, C) = 1$
- $(B, F) = 4$
- $(C, F) = 4$
- $(C, D) = 6$
- $(D, E) = 8$
- $(D, F) = 5$
- $(F, E) = 2$



单源最短路径问题

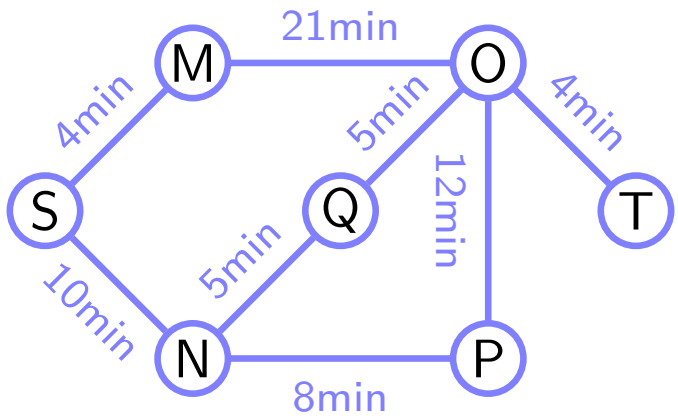
赋权路径长

对于加权图$G=(V,E)$，假设与每条边(v_i,v_j)相连的代价为$c_{i,j}$，定义一条路径$v_1,v_2,v_3,\ldots,v_N$的赋权路径长为$\sum_{i=1}^{N-1}c_{i,i+1}$。

单源最短路径问题

找出从一个特定顶点S到G中每一个其他顶点的最短赋权路径。

思考：寻找从“双子峰”（S）到“金门大桥”（T）的耗时最短路径

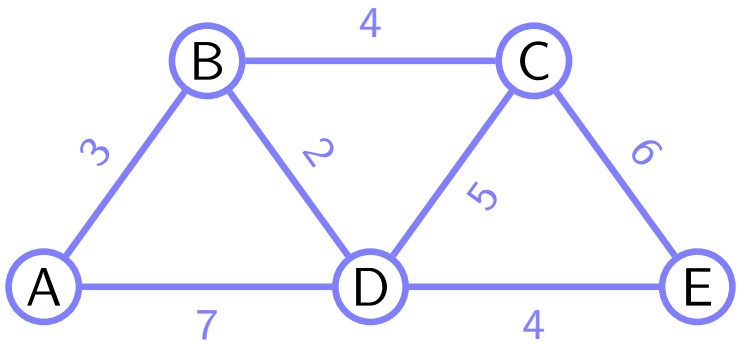


Dijkstra算法详细过程

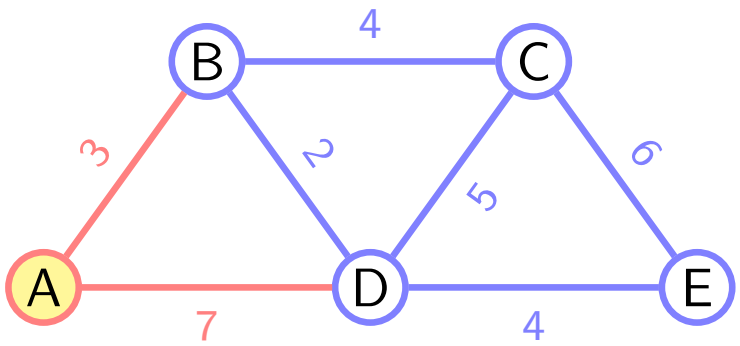
1. 根据选择的起点s，将顶点V划分为两个部分，树顶点$O = \{s\}$和边缘顶点$V - O$；
2. 给图的所有顶点V添加标记d，记录目前为止从起点s到每个顶点的最短路径长度，并记录该路径倒数第2个顶点（父节点）；
3. 从边缘顶点$V - O$中选择具有最小d值的节点u^*，把u^*加入O中；
4. 对于$V - O$中的剩余顶点u，如果通过权重为$w(u^*, u)$的边和u^*相连接，当$d_{u^*} + w(u^*, u) < d_u$时，$u$的父节点标记更新为$u^*$，最短路径长度$d_u$更新为$d_{u^*} + w(u^*, u)$；
5. 算法重复步骤3和4，逐步从$V - O$选择节点，加入到O中，直到$V = O$；

实例

利用Dijkstra算法寻找从点A开始的最短路径



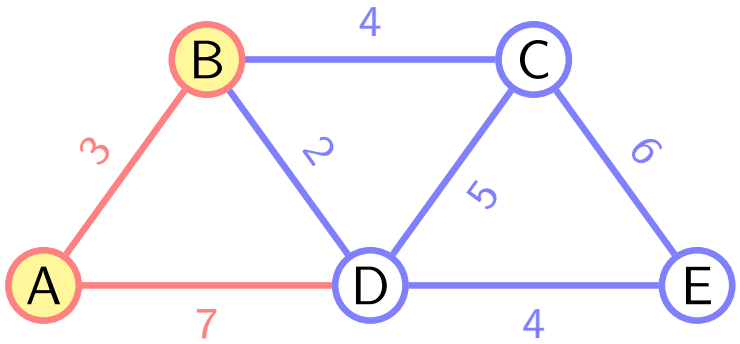
计算过程



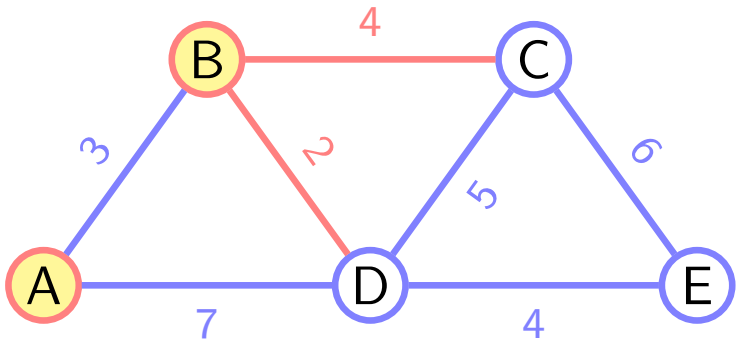
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
			B	A	3
			C	-	∞
			D	A	7
			E	-	∞

实例

计算过程



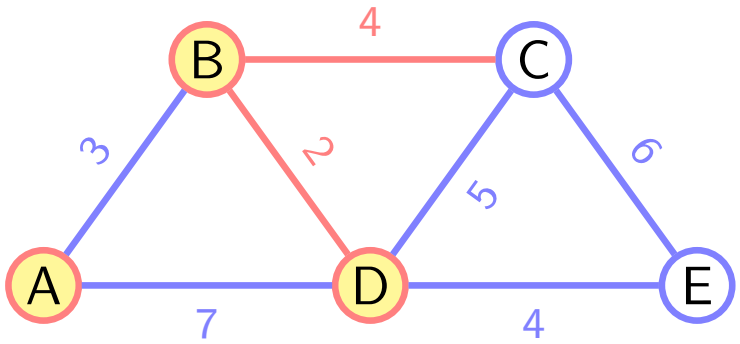
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
			C	-	∞
			D	A	7
			E	-	∞



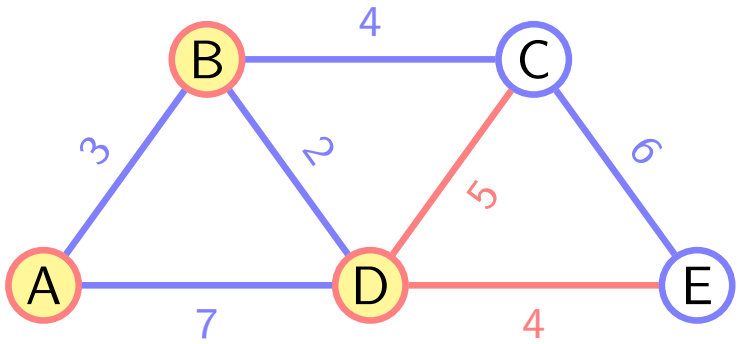
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
			C	B	3+4
			D	B	3+2
			E	-	∞

实例

计算过程



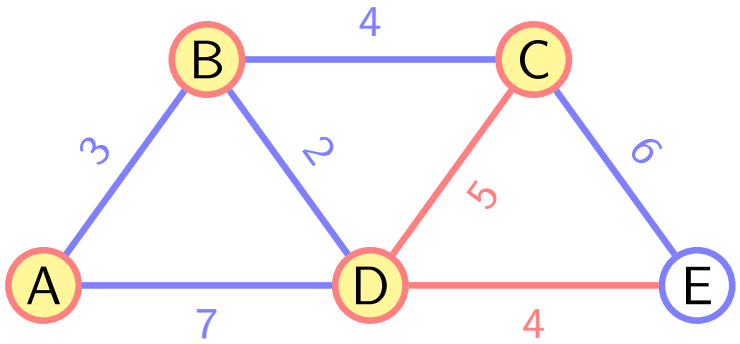
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
D	B	5			
			C	B	7
			E	-	∞



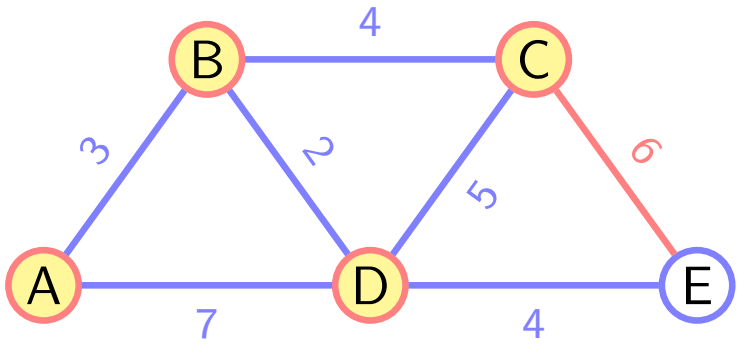
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
D	B	5			
			C	B	7
			E	D	9

实例

计算过程



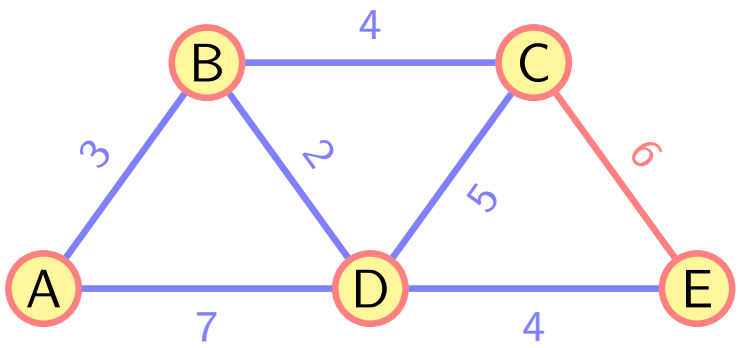
树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
D	B	5			
C	B	7			
			E	D	9



树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
D	B	5			
C	B	7			
			E	D	9

实例

计算过程



树顶点			边缘顶点		
节点	父节点	权重	节点	父节点	权重
A	-	0			
B	A	3			
D	B	5			
C	B	7			
E	D	9			

最短路径

- 从A到B：A-B，长度为3;
- 从A到C：A-B-C，长度为7;
- 从A到D：A-B-D，长度为5;
- 从A到E：A-B-D-E，长度为9;

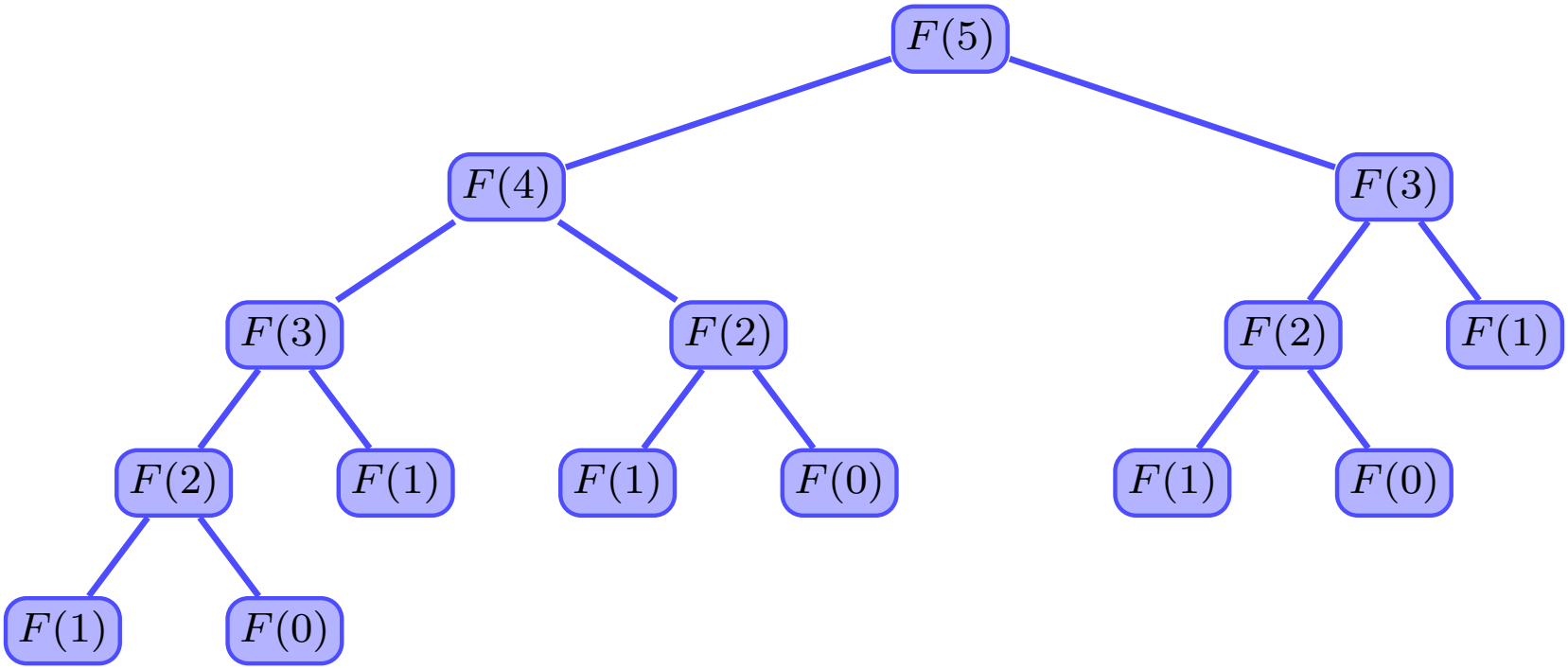
主要内容

动态规划思想

要点

如果问题是由交叠的子问题构成（递推），动态规划对每个子问题只求解一次，并把结果记录在表中，递推计算得到原始问题的解。

例子：计算Fibonacci数列



币值最大化问题

问题描述

给定一排n个硬币，其面值均为正整数，$c_1, c_2, ..., c_n$，请问如何选择硬币，使得在其原始位置互不相邻的情况下，所选硬币的总金额最大。

递推关系

- 所有可行的计划：包括最后一枚硬币，不包含最后一枚硬币；
- 第1组中，可选硬币的最大金额为$c_n + F(n-2)$；
- 第2组中，可选硬币的最大金额为$F(n-1)$；
- $F(n) = max\{c_n + F(n-2), F(n-1)\}, n > 1; F(0) = 0, F(1) = c_1$；

计算复杂度

需要计算$n-1$个F值，每个需要2次比较，复杂度为$\Theta(n)$。

找零问题

问题描述

给定m个硬币，其面值分别为$d_1, d_2, ..., d_m$，$d_1 < d_2 ... < d_m$，需找零金额为n，最少需要多少硬币？

递推关系

- 获得总金额为n的只能是在总金额为$n - d_j$的一堆硬币上加入一个面值为d_j的硬币；
- 先找出最小的$F(n - d_j)$，然后加上1即可得到解；
- $F(n) = \min_{j:n \geqslant d_j} \{F(n - d_j)\} + 1, n > 0; F(0) = 0;$

计算复杂度

需要计算n个F值，每个值最多需要m次比较，复杂度为$\Theta(mn)$。

硬币收集问题

问题描述

$n*m$的格木板中放有一些硬币，每格的硬币数目最多一个。机器人从木板的左上方移动到右下方（每次只能移动一个格子，并且不能后退），遇到有硬币的单元格就会收集硬币。设计算法找出机器人能够收集最大硬币数的路径。

递归关系

- $F(i,j)$为机器人截止到单元格(i,j)能够收集到的最大硬币数；
- 第一行上方和第一列左边的单元格，假定其硬币数为0；
- $F(i,j)=\max\{F(i-1,j),F(i,j-1)\}+c_{ij}, 1\leqslant i\leqslant n, 1\leqslant j\leqslant m$；
- $F(0,j)=0, 1\leqslant j\leqslant m;\quad F(i,0)=0, 1\leqslant i\leqslant n$；

背包问题

问题描述

- 求能够放入给定承重量的背包的最大价值的物品集合；
- 穷举查找法可以得到最优解，但是计算复杂度较高（2^n）；
- 贪心算法计算较快，但是只能得到近似解；

动态规划求解

- $F(i,j)$为前i个物品放进承重量为j的背包问题的最优解；
- 前i个物品构成的子集分为两类：含第i个物品和不含第i个物品；
- $$F(i,j)=\begin{cases}\max\{F(i-1,j),v_i+F(i-1,j-w_i)\}, & j-w_i\geqslant 0\\ F(i-1,j), & j-w_i<0\end{cases}$$
- $F(0,j)=0, j\geqslant 0; F(i,0)=0, i\geqslant 0$；
- 一共需要计算nW个F值，复杂度为$\Theta(n)$；

最后的最后

- 只有自己诚心待人，别人才有可能对自己以诚相待。

–《平凡的世界》

- 书读的越多而不加思考，你就会觉得你知道得很多；当你读书且思考得越多的时候，你就会越清楚地看到，你知道得很少。

–伏尔泰

- In the computer field, the moment of truth is a running program, all else is prophecy.
- Your attitude, not your aptitude, will determine your altitude.
- 非常感谢同学们!