湖南科技大学课程教案
（章节、专题首页）

授课教师：王志喜　　职称：副教授　　单位：计算机科学与工程学院

课程名称	算法设计与分析
章节、专题	难问题的近似算法
教学目标及基本要求	熟悉近似算法的性能比和多项式时间近似方案等相关概念，掌握各类近似算法的一般设计方法。
教学重点	近似算法的性能比和多项式时间近似方案等相关概念
教学难点	概率上的好算法，完全多项式时间近似方案
教学内容与时间分配	（1）相关基本概念（0.6课时） （2）常数性能比的近似算法（0.4课时） （3）概率上的好算法（0.75课时） （4）完全多项式时间近似方案（0.75课时） （5）带随机因素的近似算法（0.5课时） 共计3课时。
习题	第10.6.1节（基础知识题），第10.6.2节（实验题）。

第10章　难问题的近似算法

本章首先介绍近似算法的性能比和多项式时间近似方案等相关概念，然后通过实例说明各类近似算法的设计方法，包括常数性能比的近似算法（顶点覆盖问题）、概率上的好算法（顶点覆盖问题和最大团问题）、完全多项式时间近似方案（子集和问题），最后考虑将随机因素引入近似算法（团问题和着色问题）。

- 相关基本概念
- 常数性能比的近似算法
- 概率上的好算法
- 完全多项式时间近似方案
- 带随机因素的近似算法

10.1 相关基本概念

10.1.1 关于难问题

1. 难问题的含义

有很多问题到目前为止都没有找到多项式时间的求解算法，一般将这些问题看作难处理的问题。例如，旅行商问题目前最好的求解算法的计算时间为 $O(n^2\ 2^n)$，最大团问题目前最好的求解算法的计算时间为 $O(2^{n/3})$，0/1背包问题目前最好的求解算法的计算时间为 $O(2^{n/2})$，都不是多项式时间。

为方便描述，本章将难处理的问题简称为难问题。

2. 一些典型的难问题

（1）团问题。在无向图中寻找最大团。

（2）顶点覆盖问题。在无向图中寻找最小顶点覆盖。

（3）哈密尔顿回路问题。在无向图中寻找哈密尔顿回路。

（4）旅行商问题。在边权值非负的无向加权图中寻找费用最小的旅行。

（5）着色问题。为无向图选取颜色数目最少的着色方案。

（6）子集和问题。在正整数集合S中寻找和为t的子集。

（7）0-1背包问题。给定正数数组$\{v_i\}_{i=0}^{n-1}$和$\{w_i\}_{i=0}^{n-1}$及正数c，寻找满足$\sum\{w_i x_i\}_{i=0}^{n-1} \le c$且使得$\sum\{v_i x_i\}_{i=0}^{n-1}$最大的的0/1数组$\{x_i\}_{i=0}^{n-1}$。

（8）多机调度问题。已知n个独立作业各自需要的处理时间，寻找一种调度方案使得这些作业能够在最短的时间内由m台相同的机器处理完毕。

3. 难问题的常用解决办法

到目前为止，所有难问题都没有找到多项式时间求解算法。然而，有很多难问题具有很重要的实际意义，实际应用中经常遇到。对于这类问题，通常采用以下几种解决办法。

（1）只针对特殊情形求解。遇到一个难问题时，应该仔细考察是否必须在一般意义下求解，实际应用中很可能只需要针对某种特殊情形求解。难问题的特殊情形通常有高效算法，例如，判断一个无向图是否可2着色和从顶点度数不超过3的无向图中寻找最大团等。

（2）用动态规划方法、回溯方法或分枝限界方法等方法求解。虽然不能构造出有效算法，但是，在许多情况下，这些方法比穷举搜索要有效得多。

（3）用概率方法求解。有时可通过概率分析证明某个难问题的“难”实例非常少，因此可以使用概率算法来解这类问题，设计出在平均情况下的高效算法。

（4）只求近似解。实际应用中遇到的难问题不一定需要精确解，可能只需要在一定误差范围内的近似解。许多难问题的近似算法可以用很短的时间得到一个精度很高的近似解。

本章专门讨论解决难问题的近似算法。

10.1.2 关于近似算法

1. 近似算法的用途

在实际应用中遇到的难问题通常表现为最优化问题，即表现为寻找使某一个目标函数达到最大值或最小值的解。近似算法通常用于求解最优化问题，近似算法获得的近似解可能与问题的最优解存在一定的差距。

2. 近似算法的基本要求

对一个规模为n的最优化问题，它的近似算法应该满足下列两条基本要求。

- 近似算法必须在关于n的多项式时间内完成。
- 近似算法得到的近似解必须达到一定的精度。

3. 近似算法的精度

近似算法的精度可以使用性能比或误差来度量。

为了方便描述，设问题P是一个优化问题，算法A是求解问题P的一个近似算法。

（1）性能比。对于问题P的一个实例，若最优解对应的目标函数值（最优值）为c^*，使用近似算法A获得的近似解对应的目标函数值（近似值）为c，则称$\eta=\max\{c/c^*,\ c^*/c\}$为该近似解的性能比。设$\rho(n)$是问题输入规模$n$的一个函数，如果问题$P$的任何一个实例都满足$\max\{c/c^*,\ c^*/c\}\le\rho(n)$，则称$\rho(n)$为该近似算法的性能比。

（2）误差。近似解的绝对误差定义为$|c-c^*|$，近似解的相对误差定义为$\lambda=|(c-c^*)/c^*|$。设$\varepsilon(n)$是问题输入规模n的一个函数，如果问题P的任何一个实例都满足$|(c-c^*)/c^*|\le\varepsilon(n)$，则称$\varepsilon(n)$为该近似算法的相对误差界。

（3）性能比与误差的关系。近似算法的性能比$\rho(n)$与相对误差界$\varepsilon(n)$之间的关系为$\varepsilon(n)\le\rho(n)-1$。

通常，若要使近似解达到较高的精度，则近似算法需要较大的计算量。

10.1.3 关于近似方案

通常，若要使近似解达到较高的精度，则近似算法需要较大的计算量。

一个最优化问题的近似方案是指具有相对误差界 ε $(\varepsilon>0)$ 的一类近似算法 $A(\varepsilon)$。对固定的 $\varepsilon>0$，若问题的一个近似方案需要的计算时间 $T(\varepsilon,n)$ 以关于问题规模 n 的一个多项式 $p(n)$ 为上界，则称该近似方案为多项式时间近似方案。若问题的一个近似方案需要的计算时间 $T(\varepsilon,n)$ 以关于 $1/\varepsilon$ 和问题规模 n 的一个多项式 $p(1/\varepsilon,n)$ 为上界，则称该近似方案为完全多项式时间近似方案。

下面针对一些常见的难问题来研究有效近似算法的设计与分析方法。

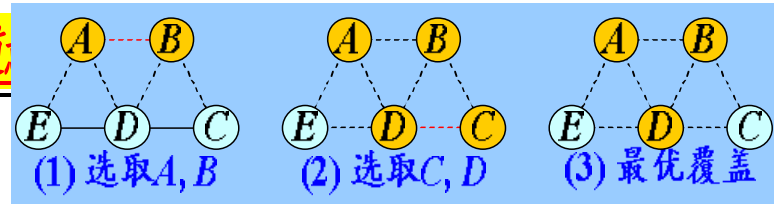
10.2 常数性能比的近似算法

以顶点覆盖问题的近似算法（最大匹配算法）为例说明。

10.2.1 问题描述

【问题】求一个无向图的最小顶点覆盖。

【说明】在后续描述中，顶点 u 覆盖边 e 是指 e 与 u 关联，顶点集 V 覆盖边集 E 是指 E 中的每一条边都至少有一个端点在 V 中。



10.2.2 顶点覆盖问题的最大匹配算法

1. 算法描述

```
#include "algorithm.h"
auto VertexCover(const Matrix<bool> &G) // 邻接矩阵
{   int n = G.rows(); // 顶点数
    vector<bool> X(n, 0); // 当前覆盖, 初始为空集
    for(int u = 0; u < n; ++u) // 选取顶点
        for(int v = u + 1; v < n; ++v) // 无向图只需考虑邻接矩阵的上三角
        {   if(G(u, v) == 0) continue; // (u, v)不是边
            if(X[u] == 0 and X[v] == 0) // 若X未覆盖边(u, v)
                X[u] = X[v] = 1; // 则选取u, v
        }
    return X;
} // 耗时O(n^2)
```

X存储顶点覆盖中的各顶点，初始为空。不断地从G中选取边(u, v)，若X未覆盖(u, v)，则将u和v加入X，直到所有边都考虑完毕。

容易看出，该近似算法可在$O(n^2)$时间内完成。

图10-1说明了最大匹配算法的运行过程。可以看出，该算法产生的近似最优顶点覆盖为$\{A, B, C, D\}$，而图G的一个最小顶点覆盖为$\{A, B, D\}$，只有3个顶点。

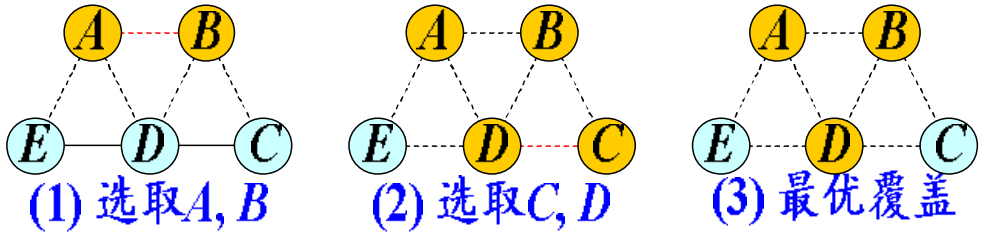


图10-1　顶点覆盖问题最大匹配算法的运行过程

```
auto VertexCover(const Matrix<bool> &G)
{   int n = G.rows();
    vector<bool> X(n, 0);
    for(int u = 0; u < n; ++u)
        for(int v = u + 1; v < n; ++v)
        {   if(G(u, v) == 0) continue;
            if(X[u] == 0 and X[v] == 0)
                X[u] = X[v] = 1;
        }
    return X;
}
```

2. 性能比

用M表示算法选出的边的集合，X表示算法选出的顶点覆盖，X^*表示最小顶点覆盖。首先，M中任何两条边都没有公共端点，这是因为每次选取的边都是不能被X覆盖的边，将其端点加入X后，不会再选取与该边有公共端点的边，所以算法终止时有$|X|=2|M|$。其次，因为M中各边都至少有一个端点在X^*中，所以$|M|\leq|X^*|$。由此可知，$|X|\leq 2|X^*|$，即该算法的性能比不超过2。

但是，经试验，该方法得到好近似的机会不多。

```
auto VertexCover(const Matrix<bool> &G)
{   int n = G.rows();
    vector<bool> X(n, 0);
    for(int u = 0; u < n; ++u)
        for(int v = u + 1; v < n; ++v)
        {   if(G(u, v) == 0) continue;
            if(X[u] == 0 and X[v] == 0)
                X[u] = X[v] = 1;
        }
    return X;
}
```

10.3 概率上的好算法

以顶点覆盖问题和最大团的近似算法为例说明。首先通过顶点覆盖问题的一种非常数性能比的近似算法说明概率上的好算法的含义，然后介绍最大团的两种概率上的好算法。

10.3.1 顶点覆盖问题的大度数优先算法

1. 设计思路

使用贪心方法。用X存储顶点覆盖中的顶点，初始为空。不断从顶点集中选取度数最大的顶点u，将u加入X中，并删除u覆盖的边，直至X已覆盖所有边，即边集合为空。

图10-2说明了顶点覆盖问题大度数优先算法的运行过程。可以看出，该算法产生的近似最优顶点覆盖为$\{A, B, D\}$，正好是图G的一个最小顶点覆盖。

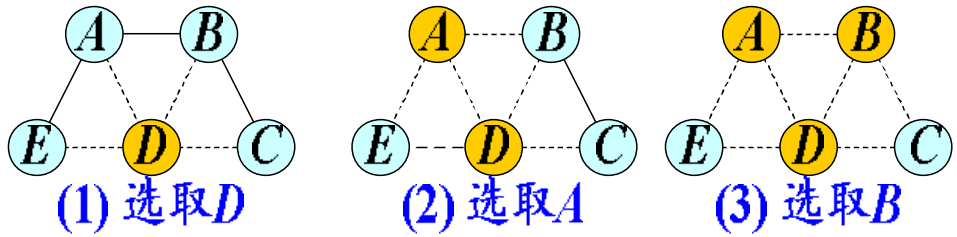


图10-2 顶点覆盖问题大度数优先算法的运行过程

2. 算法描述

实际上，如果将各顶点的度数保存在一个数组中，则删除边可以通过减少两端点的度数来实现，边集合为空可以用最大度数不大于0判断。

```
#include "Degree.h" // degrees, max_vertex, etc.
auto VertexCover2(const Matrix<bool> &G) // 邻接矩阵
{   int n = G.rows(); // 顶点数
    vector<bool> X(n, 0); // 当前覆盖, 初始为空集
    auto deg = degrees(G); // 计算各顶点的度数, 耗时O(n^2)
    for(;;) // 选取顶点, 共O(n)次
    {   int u = max_vertex(deg); // 找到度数最大的顶点u, 耗时O(n)
        if(deg[u] <= 0) break; // 若deg[u]<=0, 则边集为空
        X[u] = 1; // 选取u
        for(int v = 0; v < n; ++v) // 删除u关联的所有边, 耗时O(n)
            if(G(u, v) == 1) --deg[u], --deg[v];
    }
    return X;
}
```

因为计算各顶点度数耗时$O(n^2)$，选取顶点耗时$O(n)$，选取次数为$O(n)$，删除的边数为$O(n^2)$，所以该算法的计算时间为$O(n^2)$。

3. 性能比

设 $n=|E|$，$k=|C^*|$，n_t 是迭代 t 次以后剩余的边数（显然 $n_0=n$）。因为 C^* 覆盖所有剩余边，这意味着 $\sum\{d(v):v\in C^*\}\geq n_t$，所以，存在 $v\in C^*$ 使得 $d(v)\geq n_t/k$。由此可知，第 $t+1$ 次迭代选取的 v 满足 $d(v)\geq n_t/k$，从而

$$n_{t+1}=n_t-d(v)\leq n_t-n_t/k=n_t(1-1/k)$$

由 $1-x<e^{-x}\,(x\neq 0)$ 和 $1/k\neq 0$ 可得 $n_{t+1}<n_te^{-1/k}$。由此可得，当 $t\geq 0$ 时，$n_t\leq n_0e^{-t/k}=ne^{-t/k}$。

当 $t\geq k\ln n$ 时，$n_t<ne^{-t/k}\leq ne^{(-k\ln n)/k}=ne^{-\ln n}=1$，即 $E=\varnothing$，也就是所有顶点的度数都为 0。由此可知，$|C|\leq k\ln n+1$，从而 $|C|/|C^*|\leq(k\ln n+1)/k\leq\ln n+1$。这说明该算法的性能比不超过 $\ln|E|+1$，不是常数性能比，可能超过2。

4. 辅助程序☼

```
// Degree.h
#pragma once
#include "algorithm.h"
```

```
// 计算各顶点的度数
auto degrees(const Matrix<bool> &G) // 邻接矩阵
{   int n = G.rows(); // 顶点数
    vector<int> deg(n, 0); // 各顶点度数的初始值为0
    for(int u = 0; u < n; ++u) // 计算度数
        for(int v = 0; v < n; ++v)
            if(G(u, v) == 1) ++deg[u];
    return deg;
}
```

```
// 选取度数最大的顶点
int max_vertex(const vector<int> &X) // 度数数组
{   // 返回最大值序号
    return max_element(begin(X), end(X)) - begin(X);
}
```

10.3.2 概率上的好算法的含义

上述顶点覆盖问题贪心算法的性能比不超过$\ln|E|+1$，不是常数性能比，可能超过2。但是，该算法有一个很有意思的性质，它几乎总是产生好近似（生成最优解或近似值与最优值非常接近的解）。例如，在上述示例中已经获得了最优解。这样的算法从概率的角度来看是好算法，称为概率上的好算法。如果随机选取问题的一个实例，这样的算法产生好近似的概率非常高。

10.3.3 团问题的近似算法※

【问题】在无向图 G 中寻找最大团。

1. 根据定义构造近似算法

可以直接根据团和极大团的定义给出如下近似算法（极大团算法）。容易看出，该算法耗时 $O(n^2)$。

```
#include "Clique.h"
auto Clique(const Matrix<bool> &G) // 邻接矩阵
{   int n = G.rows(); // 顶点数
    vector<bool> X(n, 0); // 当前团，初始为空集
    for(int u = 0; u < n; ++u) // 选取顶点
    {   if(Connected(G, X, u)) // 若u与X中的各顶点都相邻
            X[u] = 1; // 则选取顶点u
    }
    return X;
} // 耗时O(n^2)
```

虽然该算法看起来感觉不怎么样，也很容易找到该算法产生不好近似的例子（请读者自己构造），但是该算法确实几乎处处产生好近似。例如，对于图10-3所示的实例，该算法得到的结果为$\{A, B, D\}$，是一个最大团。

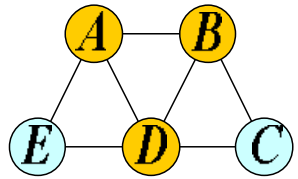
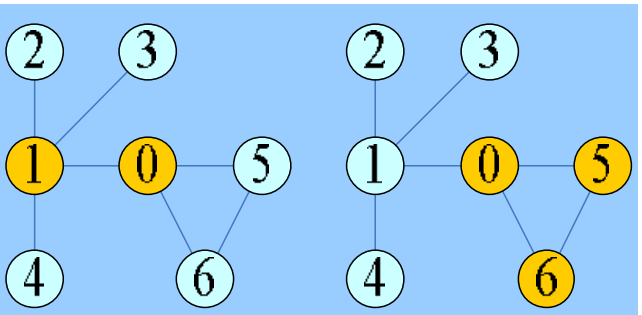


图10-3　最大团

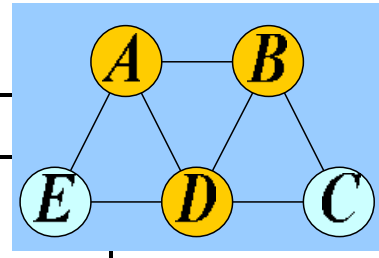



```
auto Clique(const Matrix<bool> &G)
{   int n = G.rows();
    vector<bool> X(n, 0);
    for(int u = 0; u < n; ++u)
    {   if(Connected(G, X, u))
            X[u] = 1;
    }
    return X;
}
```

2. 大度数优先算法

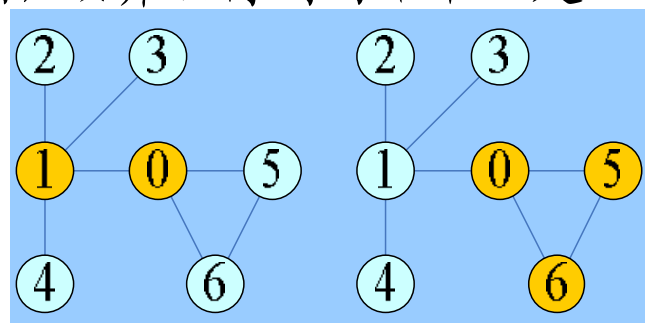
可以使用贪心方法对上述算法进行改进。用X存储最大团中的顶点，初始为空。首先从顶点集中选取度数最大的顶点u，并从顶点集中删除u（注意，不是从图中删除u），然后检查u与X中各顶点是否都相邻，若是，则将u加入X中。重复该过程直至顶点集为空。

实际上，如果将各顶点的度数保存在一个数组中，则从顶点集中删除顶点u可以通过将u的度数改为-1来实现，顶点集为空可以用最大度数小于0判断（可以优化为最大度数不大于0，因为孤立点不可能在最大团中）。




```
#include "Clique.h"
#include "Degree.h" // degrees, max_vertex, etc.
auto Clique(const Matrix<bool> &G) // 邻接矩阵
{   int n = G.rows(); // 顶点数
    auto deg = degrees(G); // 计算各顶点度数, 耗时O(n^2)
    vector<bool> X(n, 0); // 当前团, 初始为空集
    for(;;) // 选取顶点, 共O(n)次
    {   int u = max_vertex(deg); // 找到度数最大的顶点u, 耗时O(n)
        if(deg[u] <= 0) break; // 若顶点集为空或只剩孤立点, 则不再选取
        deg[u] = -1; // 从顶点集中删除顶点u
        if(Connected(G, X, u)) // 若u与X中的各顶点都相邻
            X[u] = 1; // 则选取顶点u
    }
    return X;
} // 耗时O(n^2)
```

容易看出，该算法耗时$O(n^2)$。对于图10-3所示的实例，该算法得到的结果也是$\{A, B, D\}$。



10.4 完全多项式时间近似方案

以子集和问题的近似算法为例说明。

10.4.1 最优化形式的子集和问题

【问题】设 $S=\{x_1,x_2,\ldots,x_n\}$ 是一个正整数的集合，t 是一个正整数，要求判定是否存在 S 的一个子集 S'，使得 $\sum S'=t$。在实际应用中经常遇到的是最优化形式的子集和问题。在这种情况下，要找出 S 的一个子集 S'，使得其和不超过 t，但又尽可能地接近于 t。

下面先介绍一个解最优化形式的子集和问题的指数时间算法，然后对这个算法作适当修改，使它成为解子集和问题的一个完全多项式时间的近似方案。

10.4.2 指数时间精确算法

1. 获得最优值

（1）算法描述。

```
#include "algorithm.h"
set<int> operator+(const set<int> &X, int v); // 辅助程序
// 计算X中每个元素都加上v以后得到的新表, 耗时O(n)
int SetSum(const set<int> &S, int t)
{   set<int> L = {0};
    for(int x : S)
    {   L.merge(L + x);
        L.erase(L.upper_bound(t), end(L)); // 从L中删除超过t的元素
    }
    return *rbegin(L); // max(L);
}
```

其中，L是一个由正整数组成的有序表，$L+x$表示L中每个元素都加上x以后得到的新表。L_1.merge(L_2)表示将有序表L_1和L_2合并为一个新的有序表，计算时间为$O(|L_1|+|L_2|)$（详见分治方法一章）。

当S={101, 102, 104, 201}, t=308时，该算法得到的结果是307。

（2）辅助程序。

```
set<int> operator+(const set<int> &X, int v)
{   // 辅助程序，计算X中每个元素都加上v以后得到的新表
    set<int> W;
    for(int u : X) W.insert(end(W), u + v);
    return W;
} // O(|X|)
```

2. 正确性

用 P_i 表示 $\{x_1, x_2, \ldots, x_i\}$ 的所有可能的子集和，即 P_i 中的每一个元素都是 $\{x_1, x_2, \ldots, x_i\}$ 的一个子集和。约定空子集的和为0，且 $P_0 = \{0\}$ 。不难证明 $P_i = P_{i-1} \cup (P_{i-1} + x_i)$，$i = 1, 2, \ldots, n$ 。由此易知，算法中的表 L_i 是一个由 P_i 中所有不超过 t 的元素组成的有序表。因此，L_n 中的最大元素就是 S 中不超过 t 的最大子集和。

3. 复杂性

由于 P_i 是 $\{x_1, x_2, \ldots, x_i\}$ 的所有可能的子集和的全体，故在不出现重复子集和的情况下，$|P_i| = 2^i$ 。在最坏情况下，L_i 可能与 P_i 相同。因此，在最坏情况下 $|L_i| = 2^i$ 。所以该算法是一个指数时间算法。

10.4.3 一种完全多项式时间近似方案

基于指数时间精确算法，通过对有序表 L_i 作适当修整建立一个子集和问题的完全多项式时间近似方案。

1. 关于修整

（1）修整的方法。在对有序表 L_i 进行修整时，需要用到一个修整参数 $0<\delta<1$。用参数 δ 修整一个有序表 L 是指从 L 中删去尽可能多的元素，使得每一个从 L 中删去的元素 y，在修整后的新表中都有一个元素 z 满足 $(1-\delta)y \le z \le y$。可以将 z 看作被删去元素 y 在新表中的代表。由此可知，对当前元素 y，若 $z<(1-\delta)y$，则需要将 y 作为新的 z 保留到新表中。

（2）举例说明。若 $\delta=0.1$，$L=\{10,11,12,15,20,21,22,23,24,29\}$，则用 δ 对 L 进行修整后得到 $W=\{10,12,15,20,23,29\}$。其中被删除的数11由10代表，21和22由20代表，24由23代表。

$10:\ 0.9\times\cancel{11}=0.99\le 10, 0.9\times 12=10.8>10$

$12:\ 0.9\times 15=13.5>12$

$15:\ 0.9\times 20=18>15$

$20:\ 0.9\times\cancel{21}=18.9<20, 0.9\times\cancel{22}=19.8<20, 0.9\times 23=20.7>20$

$23:\ 0.9\times\cancel{24}=21.6<23, 0.9\times 29=26.1>23$

29

$(1-\delta)\cancel{y}\le z\le\cancel{y}$

$z<(1-\delta)y$

（3）算法描述。

```
set<int> Trim(const set<int> &L, double del)
{   set<int> W = {0};
    int z = 0;
    for(int y : L)
        if(z < (1 - del) * y)
            W.insert(end(W), y), z = y;
    return W;
}
```

2. 近似方案

（1）算法描述。该近似算法的输入是 n 个正整数组成的集合 S 、目标正整数 t 和一个控制近似程度的参数 $0 \leq \varepsilon \leq 1$ 。

```
int SetSum(const set<int> &S, int t, double eps)
{   int n = S.size();
    set <int> L = {0};
    for(int x : S)
    {   L.merge(L + x);
        L = Trim(L, eps / n); // 修整
        L.erase(L.upper_bound(t), end(L)); // 从L中删除超过t的元素
    }
    return *rbegin(L); // max(L);
}
```

（2）正确性。因为在对 L_i 进行修整，并删除超过 t 的元素以后，仍然有 $L_i \subseteq P_i$ ，所以，算法的返回值是 P_n 的元素，当然是 S 的一个子集和。这说明该算法是正确的。

$(1-\delta)y \le z \le y$
$z < (1-\delta)y$

3. 性能比

设问题的最优值为 c^*，算法的返回值为 c。下面分2步分析该近似方案的性能。

（1）证明 L_i 被修整以后，对于 P_i 中任何不超过 t 的元素 y，都存在 $z \in L_i$，使得 $(1-\delta)^i y \le z \le y$，其中 $\delta = \varepsilon / n$，这时可以将 z 看作 y 在 L_i 中的代表。

用数学归纳法证明。显然，L_1 被修整以后，对于 P_1 中任何不超过 t 的元素 y，都存在 $z \in L_1$，使得 $(1-\delta)y \le z \le y$，即结论对 L_1 成立。假设结论对 L_i 成立，下面证明结论对 L_{i+1} 也成立。

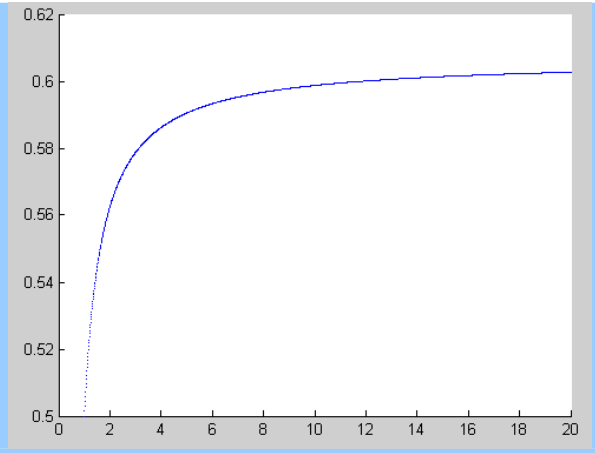
L_{i+1} 被修整以后，对于 P_{i+1} 中任何不超过 t 的元素 y，有

- 当 $y \in P_i$ 时，存在 $z \in L_i$，使得 $(1-\delta)^i y \le z \le y$。若 $z \in L_{i+1}$，则 $(1-\delta)^{i+1} y \le (1-\delta)^i y \le z \le y$；否则，存在 $z' \in L_{i+1}$，使得 $(1-\delta)z \le z' \le z$，从而 $(1-\delta)^{i+1} y \le (1-\delta)z \le z' \le y$。
- 当 $y \notin P_i$ 时，存在 $z \in L_{i+1}$，使得 $(1-\delta)y \le z \le y$，从而 $(1-\delta)^{i+1} y \le (1-\delta)y \le z \le y$。

$(1-\delta)y \le z \le y$
$z < (1-\delta)y$

（2）证明 c 与 c^* 的相对误差 $(c^*-c)/c^* \le \varepsilon$。

因为 $c^* \in P_n$，且 $c^* \le t$，由（1）可知，存在 $z \in L_n$，使得 $(1-\delta)^n c^* \le z \le c^*$。由 c 是 L_n 的最大元素可知，$z \le c \le c^*$。因此，$(1-\delta)^n c^* \le c \le c^*$，即 $(1-\varepsilon/n)^n c^* \le c \le c^*$。因为 $(1-\varepsilon) \le (1-\varepsilon/n)^n$，所以 $(1-\varepsilon)c^* \le c \le c^*$。由此可知，$c$ 与 c^* 的相对误差 $(c^*-c)/c^* \le \varepsilon$。



$f(x) = (1-\varepsilon/x)^x$

$(1-\delta)y \le z \le y$
$z < (1-\delta)y$

4. 复杂性分析

实例特征为 $m = \max(n, \text{bits}(t))$。显然，$\ln t \le \log_2 t \le m$。从算法的循环体容易看出，对有序表 L_i 进行合并、修整和删除元素等操作需要耗时 $O(|L_i|)$，且 $|L_i|$ 关于 i 递增。因此，整个算法耗时 $O(n|L_n|) = O(m|L_n|)$。

对表 L_i 进行修整后，表中的相邻元素 z 和 y 满足

$$y/z > 1/(1-\varepsilon/n) = 1+\varepsilon/(n-\varepsilon) > 1+\varepsilon/m$$

即相邻元素间至少相差比例因子 $1+\varepsilon/m$。因为 L_i 中的数都不超过 t，且可能含有0和1，所以，对 L_i 进行合并、修整和删除元素等操作后，只要 $t \ge 4$，就有

$$|L_i| \le \log_{(1+\varepsilon/m)} t + 2 \le 2\ln t / \ln(1+\varepsilon/m)$$

因为当 $x \ge 0$ 时，$x/(1+x) \le \ln(1+x)$，即 $1/\ln(1+x) \le 1+1/x$，所以由 $0 \le \varepsilon/m \le 1$ 可得 $|L_n| \le 2m(1+m/\varepsilon) \le 4m^2/\varepsilon$。

由此可知，整个算法耗时 $O(m^3/\varepsilon) = O(m^3(1/\varepsilon))$，是一个完全多项式时间近似方案。

5. 测试程序☼

```
int main()
{   set<int> S = {105, 112, 128, 177, 243, 266};
    int t = 702;
    cout << "源集合: " << S << endl;
    cout << "最优值: " << SetSum(S, t) << endl;
    cout << "近似值: " << SetSum(S, t, 0.05) << endl;
    // 近似值: 683, 近似解: 112 128 177 266
    // 最优值: 686, 最优解: 177 243 266
}
```

10.5 带随机因素的近似算法※

以团问题和着色问题为例说明。

10.5.1 团问题

【问题】求一个无向图的最大团。

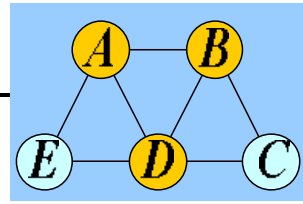
1. 算法描述

（1）一次执行。由下述算法容易看出，一次执行耗时$O(n^2)$。

```
#include "Clique.h"
int Clique(const Matrix<bool> &G, vector<bool> &X)
{   // 邻接矩阵, 当前团
    int n = G.rows(), V[n]; // 顶点数, 顶点集
    iota(V, V + n, 0); // 顶点集初始化为自然排列
    random_shuffle(V, V + n); // 顶点集洗牌
    X.assign(n, 0); // 当前团初始化为空集(必须)
    for(auto u : V) // 选取顶点, 共O(n)次
        if(Connected(G, X, u)) X[u] = 1; // u可行, 选取u, 耗时O(n)
    return count(begin(X), end(X), 1); // 当前团顶点数
} // 耗时O(n^2)
```

（2）多次执行。

```
auto Clique(const Matrix<bool> &G) // 邻接矩阵
{   int n = G.rows(); // 顶点数
    vector<bool> X, BX; // 当前团, 最优团
    int bn = 0; // 最优团大小
    for(n *= 2; n > 0; n /= 2) // 执行log(n)+1次
    {   int cn = Clique(G, X);
        if(cn > bn) BX = X, bn = cn; // 更优, 修改结果
    }
    return BX;
} // 耗时O(n^2*log(n))
```



2. 测试☼

使用如图10-3所示的图。

```
int main()
{   Matrix<bool> G = // 邻接矩阵
    {   {0, 1, 0, 1, 1},
        {1, 0, 1, 1, 0},
        {0, 1, 0, 1, 0},
        {1, 1, 1, 0, 1},
        {1, 0, 0, 1, 0}
    };
    cout << Clique(G) << endl; // 1 0 0 1 1
}
```

10.5.2 着色问题

【问题】为一个需要着色的无向图选取一种颜色数目最少的着色方案。

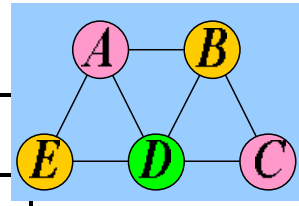
这里首先使用贪心方法求解，然后在此基础上构造蒙特卡洛算法。

1. 贪心算法

（1）贪新准则。尽量使用已用颜色为新顶点着色。

（2）算法描述。

```
#include "Chromatic.h" // legal
auto Chromatic(const Matrix<bool> &G) // 邻接矩阵
{   int n = G.rows(); // 顶点数
    vector<int> X(n, -1); // 各顶点颜色(1, 2, ...), 负数表示未着色
    for(int t = 0; t < n; ++t) // 为各顶点着色(尽量使用已用颜色)
        for(X[t] = 1; not legal(G, X, t); ++X[t]) {}
    return X;
}
```



（3）测试。使用如图10-3所示的图。

```
int main()
{   Matrix<bool> G = // 邻接矩阵
    {   {0, 1, 0, 1, 1},
        {1, 0, 1, 1, 0},
        {0, 1, 0, 1, 0},
        {1, 1, 1, 0, 1},
        {1, 0, 0, 1, 0}
    };
    auto X = Chromatic(G);
    cout << X << endl; // 1 2 1 3 2
}
```

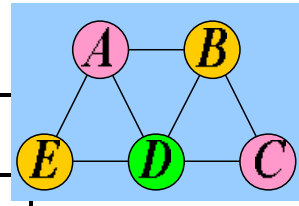
2. 蒙特卡洛算法

（1）一次执行。由下述算法容易看出，一次执行耗时$O(n^3)$。

```
#include "Chromatic.h" // legal
auto Chromatic(const Matrix<bool> &G, vector<int> &X)
{   // 邻接矩阵, 当前方案(负数表示未着色)
    int n = G.rows(), V[n]; // 顶点数, 顶点集
    iota(V, V + n, 0); // 顶点集初始化为自然排列
    random_shuffle(V, V + n); // 顶点集洗牌
    X.assign(n, -1); // 当前方案初始化为未着色
    for(auto t : V) // 为各顶点着色(尽量使用已用颜色)
        for(X[t] = 1; not legal(G, X, t); ++X[t]) {}
    return *max_element(begin(X), end(X)); // 返回已用颜色数
} // 耗时O(n^3)
```

（2）多次执行。

```
auto Chromatic(const Matrix<bool> &G)
{   int n = G.rows(); // 顶点数
    vector<int> X, W; // 当前方案, 最优方案
    int m = n + 1; // 最优颜色数
    for(n *= 2; n > 0; n /= 2) // 执行log(n)+1次
    {   int cm = Chromatic(G, X);
        if(cm < m) W = X, m = cm; // 更优, 修改结果
    }
    return W;
} // 耗时O(n^3×log(n))
```



（3）测试。使用如图10-3所示的图。

```
int main()
{   Matrix<bool> G = // 邻接矩阵
    {   {0, 1, 0, 1, 1},
        {1, 0, 1, 1, 0},
        {0, 1, 0, 1, 0},
        {1, 1, 1, 0, 1},
        {1, 0, 0, 1, 0}
    };
    auto X = Chromatic(G);
    cout << X << endl; // 3 1 3 2 1 (1 2 1 3 2)
}
```

10.6 练习题

10.6.1 基础知识题

1. 请使用一种程序设计语言为顶点覆盖问题构造一个具有常数性能比的近似算法。
2. 请使用一种程序设计语言为顶点覆盖问题构造一个概率上的好算法。
3. 请使用一种程序设计语言为最大团问题构造一个概率上的好算法。
4. 请使用一种程序设计语言为子集和问题构造一个完全多项式时间近似方案。
5. 请构造一个求最大团的近似算法产生不好近似的例子。

10.6.2 实验题

1. 将求给定无向图的近似最大团的一种概率上的好算法改写成蒙特卡洛算法，以达到在多项式时间内尽最大可能获得最大团的目的。

2. 为0/1背包问题构造一个近似算法，并将该近似算法改写成蒙特卡洛算法，以达到在多项式时间内尽最大可能获得最优解的目的。

3. 对于顶点覆盖问题，为了能够在保证常数性能比的基础上提高获得好近似的机会，可以考虑将顶点覆盖问题的最大匹配算法和贪心算法结合使用。请使用一种程序设计语言完成该任务。

10.6.3 进阶训练题

1. 子集和问题的近似解。对于子集和问题，大多数文献给出的近似算法只给出了近似值，没有给出获得近似解的方法。请修改在“完全多项式时间近似方案”一节中给出的近似算法，使得修改后的近似算法能够获得近似值和近似解，并且是一种完全多项式时间近似方案（必须分析性能和耗时）。

2. 子集和问题的近似方案。设S是一个正整数集合，t是一个正整数，寻找S的满足子集和不小于t且使得子集和最小的子集。为该问题设计一种完全多项式时间近似方案以获得近似最优值（必须分析性能和耗时）。

3. 顶点覆盖问题的蒙特卡洛算法。将求给定无向图的近似最优顶点覆盖的一种概率上的好算法改写成蒙特卡洛算法，以达到在多项式时间内尽最大可能获得最优顶点覆盖的目的。

4. 集合覆盖问题的蒙特卡洛算法。参照顶点覆盖问题的近似算法为集合覆盖问题设计一种近似算法（要求分析性能和耗时），并将该算法改写成蒙特卡洛算法，以达到在多项式时间内尽最大可能获得最优覆盖的目的。

5. 多机调度问题的蒙特卡洛算法。将求解多机调度问题的贪心算法改写成蒙特卡洛算法，以达到在多项式时间内尽最大可能获得最优调度的目的。