第12章　*NP*完全问题的近似算法*

本章首先介绍近似算法的性能比和多项式时间近似算法等相关概念，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）、非常数性能比的近似算法（顶点覆盖问题）、概率上的好算法（最大团问题）、完全多项式时间近似方案（子集和问题），并以旅行商问题的近似算法为例来说明近似算法与 $P = NP$ 的关系。

- 求解NP完全问题的常用办法
- 近似算法的性能比
- 常数性能比的近似算法
- 非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
- 近似算法与$P = NP$

12.1 求解*NP*完全问题的常用办法

到目前为止，所有 *NP* 完全问题都没有找到多项式时间算法。然而有许多 *NP* 完全问题具有很重要的实际意义，经常遇到。对于这类问题，通常有以下几种办法。

① 只对特殊情形求解。遇到一个 *NP* 完全问题时，应仔细考察是否必须在最一般的意义下求解。也许只要针对某种特殊情形求解就够了。*NP* 完全问题的特殊情形常常有高效的算法。例如，2-SAT、2-可着色、顶点度数不超过3的图的团问题。

② 用动态规划方法、回溯方法或分枝限界方法等方法求解。虽然不能构造有效算法，但是，在许多情况下，它们比穷举搜索方法要有效得多。

③ 用概率方法求解。有时可通过概率分析证明某个 *NP* 完全问题的“难”实例非常少，因此可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。实际中遇到的 *NP* 完全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 *NP* 完全问题的近似算法可以在很少的时间内得到一个精度很高的近似解。

本章只讨论解决 *NP* 完全问题的近似算法。

12.2 近似算法的性能比

在实际应用中遇到的 NP 完全问题多表现为最优化问题，即表现为要求使某一个目标函数达到最大值或最小值的解。

对一个规模为 n 的 NP 完全的最优化问题，它的近似算法应该满足下列两条基本要求。

① 算法在关于 n 的多项式时间内完成。

② 算法得到的近似解达到一定的精度。

近似解的精度可以用性能比来度量。

1. 性能比

若一个最优化问题的最优值为 c^*，求解该问题的一个近似算法求得的近似最优解相应的目标函数值为 c，则将该近似算法的性能比定义为 $\eta = \max\{c/c^*,\ c^*/c\}$。在通常情况下，该性能比是问题输入规模 n 的一个函数 $\rho(n)$（通常直接将 $\rho(n)$ 称为性能比），即 $\max\{c/c^*,\ c^*/c\} \le \rho(n)$。该近似算法的绝对误差定义为 $|c-c^*|$，相对误差定义为 $\lambda = |(c-c^*)/c^*|$。若对问题的输入规模 n，有一函数 $\varepsilon(n)$ 使得 $|(c-c^*)/c^*| \le \varepsilon(n)$，则称 $\varepsilon(n)$ 为该近似算法的相对误差界。近似算法的性能比 $\rho(n)$ 与相对误差界 $\varepsilon(n)$ 之间的关系为 $\varepsilon(n) \le \rho(n)-1$。

2. 多项式时间近似方案

一般说来，若要使近似解达到较高的精度，则近似算法需要有较大的计算量。

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 $\varepsilon>0$，若近似方案需要的时间 $T(\varepsilon,n)$ 与问题规模的一个多项式同阶，则称该近似方案为多项式时间近似方案。若一个问题的近似方案的计算时间 $T(\varepsilon,n)$ 是关于 $1/\varepsilon$ 和问题规模的多项式（ $p(1/\varepsilon,n)$ ），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 NP 完全问题来研究有效近似算法的设计与分析方法。

12.3 常数性能比的近似算法

以顶点覆盖问题的近似算法为例说明。

12.3.1 问题描述

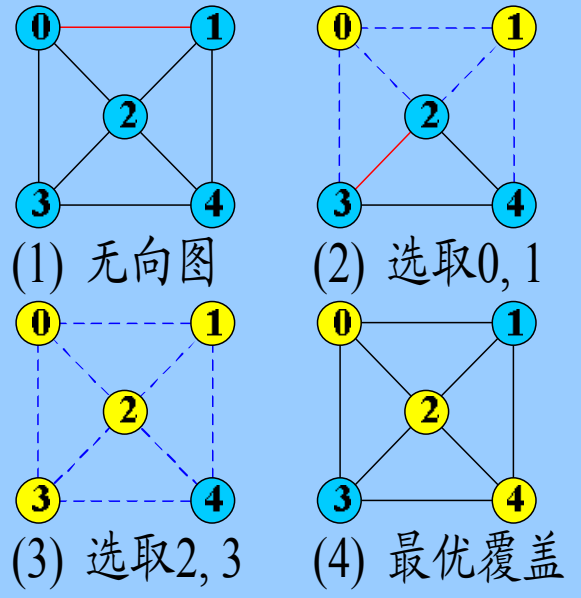
【问题】求一个无向图的最小顶点覆盖。

【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。

12.3.2 顶点覆盖问题的近似算法 I

1. 算法描述

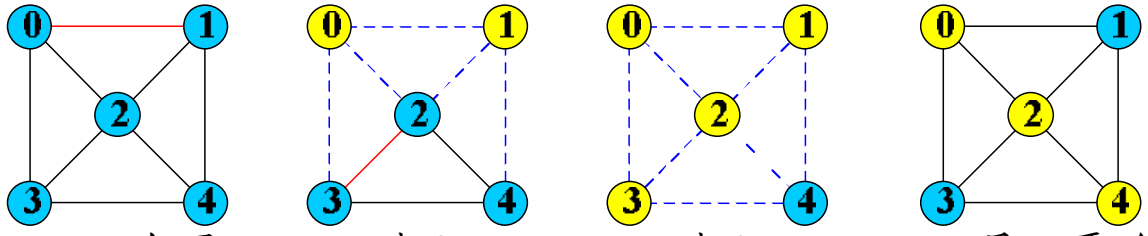
```
function ApproxVertexCover(G)
{   C = {};
    for([u, v] in G.E)
    {   if(u not in C and v not in C) // 可在O(1)时间内完成
            C.insert(u, v); // u, v加入C, 可在O(1)时间内完成
    }
    return C;
} // 可在O(|E|)时间内完成
```



C 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 C 未覆盖 (u,v)，则将端点 u 和 v 加入 C，直到所有边都考虑完毕。经试验，该方法得到好近似的机会不多。

容易看出，该近似算法可在 $\Theta(|G.E|)=O(|G.V|^2)$ 时间内完成。

图12-1说明了算法 I 的运行过程及结果。可以看出，算法产生的近似最优顶点覆盖由顶点 $0,1,2,3$ 组成，而图 G 的一个最小顶点覆盖只有3个顶点 $0,2,4$。



(1) 无向图　(2) 选取0, 1　(3) 选取2, 3　(4) 最优覆盖

图12-1　算法 I 的运行过程及结果

2. 性能比

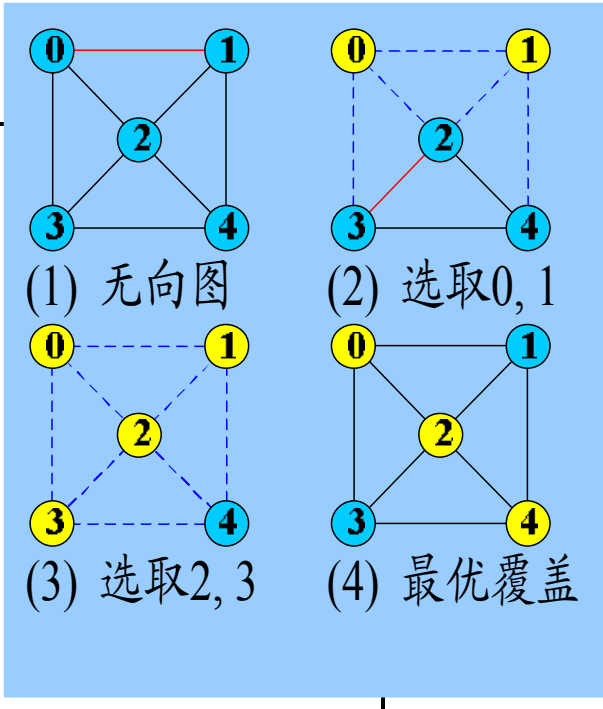
用 A 表示算法选出的边的集合，C 表示算法选出的顶点覆盖，C^* 表示最小顶点覆盖。首先，A 中任何两条边都没有公共端点，这是因为每次选择的边都是不能被 C 覆盖的边，将其端点加入 C 后，不会再选取 E 中与该边有公共端点的边，所以算法终止时有 $|C|=2|A|$。其次，因为 A 中各边都至少有一个端点在 C^* 中，所以 $|A|\leq|C^*|$。由此可知，$|C|\leq 2|C^*|$，即该算法的性能比不超过2。

3. C++程序*

```
#include "algorithm.h"
auto ApproxVertexCover(const Matrix<bool> &G)
{   int n = G.Rows();
    vector<bool> X(n, 0); // 初始化
    for(int u = 0; u < n; ++u)
        for(int v = u + 1; v < n; ++v)
        {   // 因为是无向图，只需考虑邻接矩阵的上三角
            if(G[u][v] != 0 && X[u] == 0 && X[v] == 0)
                X[u] = X[v] = 1; // X未覆盖边(u, v)，将u, v加入X
        }
    return X;
}
```

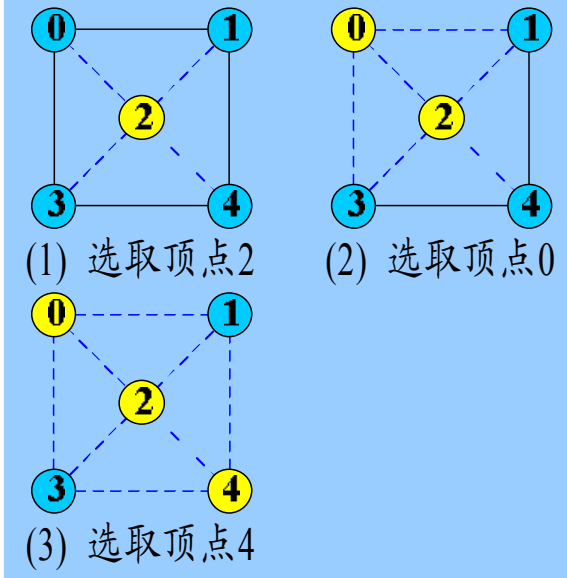
```
int main()
{   Matrix<bool> G =
    {   {0, 1, 1, 1, 0},
        {1, 0, 1, 0, 1},
        {1, 1, 0, 1, 1},
        {1, 0, 1, 0, 1},
        {0, 1, 1, 1, 0},
    };
    cout << ApproxVertexCover(G) << endl;
}
```

```
1 1 1 1 0
```



12.5 非常数性能比的近似算法

以顶点覆盖覆盖问题的另一种近似算法为例说明。

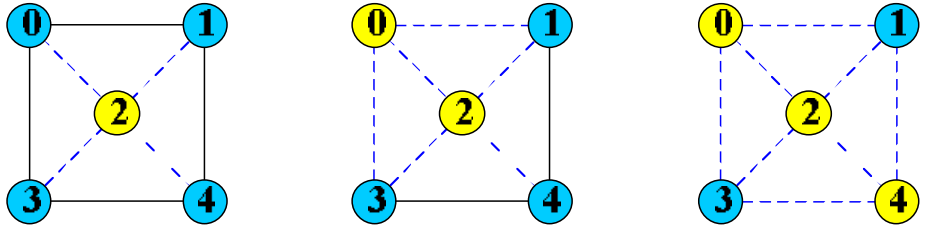


12.5.1 顶点覆盖问题的近似算法Ⅱ

1. 设计思路

使用贪心方法。用 C 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 C 中，并将 u 覆盖的边删去，直至 C 已覆盖所有边，即边集合为空。

图12-2说明了算法Ⅱ的运行过程及结果。可以看出，算法产生的近似最优顶点覆盖由顶点 $0,2,4$ 组成，正好是图G的一个最小顶点覆盖。



(1) 选取顶点2　(2) 选取顶点0　(3) 选取顶点4

图12-2　算法Ⅱ的运行过程及结果

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。

```
function ApproxVertexCover2(G)
{   C = {};
    deg[] = '各顶点度数'; // 计算各顶点度数
    for(u = max(deg); deg[u] > 0; u = max(deg))
    {   // 选取度数最大的顶点u, 若deg[u]<=0, 则边集为空
        C.insert(u); // 将u加入C
        for(v in V) // 删除u覆盖的所有边
            if([u, v] in E) --deg[u], --deg[v];
    }
    return C;
}
```

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

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 $\mathrm{sum}\{d(v):v\in C^*\}\geq n_t$，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。

12.5.2 近似算法Ⅱ的C++描述*

1. 计算各顶点度数

保存在文件Degree.h中。

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

```
auto degrees(const Matrix<bool> &G) // 计算各顶点的度数
{   int n = G.Rows();
    vector<int>deg(n, 0); // 各顶点度数初始化为0
    for(int u = 0; u < n; ++u)
        for(int v = 0; v < n; ++v)
            if(G[u][v] == 1) ++deg[u];
    return deg;
}
```

```
int max(const vector<int> &deg) // 数组中最大值的下标
{   auto F = begin(deg), L = end(deg);
    return max_element(F, L) - F;
}
```

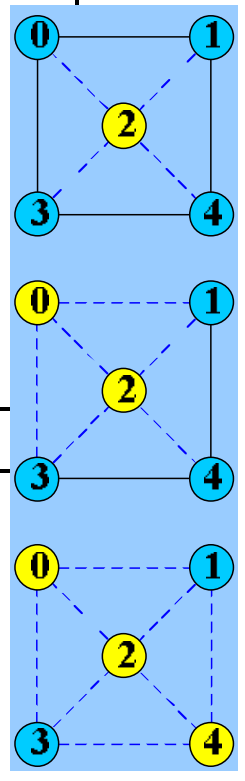
2. 算法描述

```
#include "Degree.h"
auto ApproxVertexCover2(const Matrix<bool> &G)
{   int n = G.Rows();
    vector<bool> X(n, 0); // 解向量初始化为空集
    auto deg = degrees(G); // 计算各顶点的度数
    for(int u = max(deg); deg[u] > 0; u = max(deg))
    {   // 选取度数最大的顶点u, 若deg[u]<=0, 则边集为空
        X[u] = 1; // 将u加入X
        for(int v = 0; v < n; ++v) // 删除u关联的所有边
            if(G[u][v] == 1) --deg[u], --deg[v];
    }
    return X;
}
```

3. 测试

```
int main()
{   Matrix<bool> G = // 邻接矩阵
    {   {0, 1, 1, 1, 0},
        {1, 0, 1, 0, 1},
        {1, 1, 0, 1, 1},
        {1, 0, 1, 0, 1},
        {0, 1, 1, 1, 0},
    };
    cout << ApproxVertexCover2(G) << endl;
}
```

```
1 0 1 0 1
```



12.6 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。

12.6.1 概率上的好算法的含义

上述顶点覆盖问题的近似算法Ⅱ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，它几乎处处产生好近似（例如，在上述例子中已经获得了最优解），即产生好近似的概率为1，这类算法通常称为概率上的好算法。

【注】要准确说明“几乎处处产生好近似”的含义，需要测度论和概率论的知识，这里略过，可以粗略地理解为“能够对几乎所有实例都产生好近似”。

12.6.2 最大团的近似算法

【问题】在无向图 G 中寻找最大团。

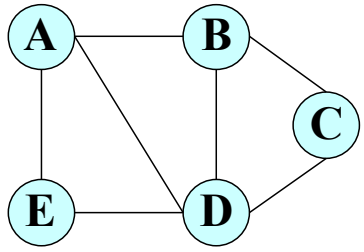


图12-3 一个最大团实例

1. 根据定义构造近似算法

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

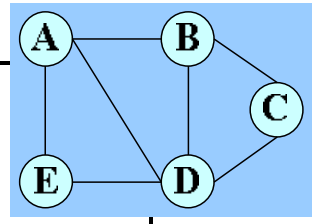
```
function ApproxClique(G)
{   X = {};
    for(u in V)
        if(Connected(X, u)) // u与X中的各顶点都相邻
            X.insert(u); // 将顶点u加入X
    return X;
}
```

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12-3所示的实例，该算法得到的结果为 $\{A, B, D\}$，是一个最大团。

2. 贪心算法

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，并从顶点集中将 u 删去（不需要从图中删除），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可以通过将顶点度数改为-1来实现，顶点集合为空可以用最大度数小于0判断（可优化为最大度数不大于0，因为孤立的顶点不可能在最大团中）。

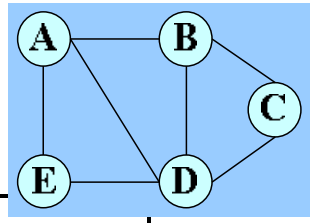



```
function ApproxClique2(G)
{   X = {};
    deg[] = '各顶点度数'; // 计算各顶点度数
    for(;;)
    {   u = max(deg); // 选取度数最大的顶点u
        if(deg[u] <= 0) break; // 若deg[u]<=0, 则顶点集为空
        deg[u] = -1; // 从顶点集中删除顶点u
        if(Connected(X, u)) // u与X中的各顶点都相邻
            X.insert(u); // 将顶点u加入X
    }
    return C;
}
```

容易看出，该算法耗时 $O(n^2)$。对于图12-3所示的实例，该算法得到的结果也是 $\{A, B, D\}$。

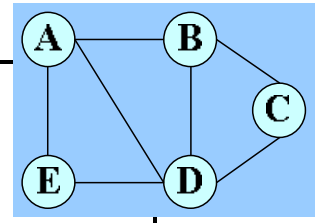
3. 算法的C++描述及测试*

使用如图12-3所示的图。




```
#include "Clique.h"
#include "Degree.h"

auto ApproxClique(Matrix<bool> &G)
{   int n = G.Rows(); // 顶点个数
    auto deg = degrees(G); // 计算各顶点度数, 保存在数组deg中
    vector<bool> X(n, 0); // 解向量, 初始为空集
    for(;;)
    {   int u = max(deg); // 选取度数最大的顶点u
        if(deg[u] <= 0) break; // 若deg[u]<=0, 则顶点集为空
        deg[u] = -1; // 从顶点集中删除顶点u
        if(Connected(G, X, u)) X[u] = 1; // 顶点u加入X
    }
    return X;
}
```




```
int main()
{   Matrix<bool> G = // 图的邻接矩阵
    {   {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 << ApproxClique(G) << endl;
}
```

```
1 1 0 1 0
```

12.7 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。

12.7.1 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 $\sum_{x\in S'}x=t$。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。

12.7.2 指数时间精确算法

1. 获得最优值

```
function ExactSetSum(S, t)
{   L = {0};
    for(x in S)
    {   L = L + (L + x);
        upper_remove(L, t); // 从L中删除超过t的元素
    }
    return max(L);
}
```

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+L_2 表示将有序表 L_1 和 L_2 合并为一个新有序表，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. 正确性与复杂性

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。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。

12.7.3 关于修整

基于指数时间精确算法算法，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. 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，都有一个修整后的有序表 L_1 中的元素 z 满足 $(1-\delta)y \le z \le y$。可以将 z 看作是被删去元素 y 在修整后的新表 L_1 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。

【举例】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 $W=\{10,12,15,20,23,29\}$。其中被删去的数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

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

$z<(1-\delta)y$

2. 算法描述

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

```
function Trim(L, del)
{   W = {0};
    z = 0;
    for(y in L)
        if(z < (1 - del) * y)
            W.insert(y), z = y;
    return W;
}
```

12.7.4 一种完全多项式时间近似方案

1. 近似方案

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

```
function ApproxSetSum(S, t, eps)
{   L = {0};
    for(x in S)
    {   L = L + (L + x); // Merge(L, L + x)
        L = Trim(L, eps / S.size()); // 修整
        upper_remove(L, t); // 从L中删除超过t的元素
    }
    return max(L);
}
```

2. 正确性

因为在对 L_i 进行修整，并删去超过 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} 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

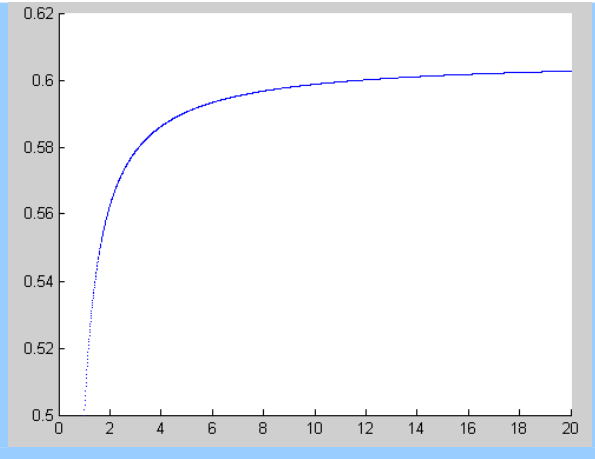
- 当 $y \in P_i$ 时，存在 $z \in L_i$，使得 $(1-\delta)^i y \le z \le y$。若 $z \in L_{i+1}$，则 $(1-\delta)^{i+1} y \le (1-\delta)^i y \le z \le y$；否则，存在 $z' \in L_{i+1}$，使得 $(1-\delta)z \le z' \le z$，从而 $(1-\delta)^{i+1} y \le (1-\delta)z \le z' \le y$
- 当 $y \notin P_i$ 时，存在 $z \in L_{i+1}$，使得 $(1-\delta)y \le z \le y$，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 $m = \max(n, \log_2 t)$。显然，$\ln t \le m$。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。

12.7.5 算法的C++描述及测试*

1. 支持程序

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

```
int max(const set<int> &X) // 最大元素
{   return *rbegin(X);
} // O(1)
```

```
auto operator+(const set<int> &X, const set<int> &Y) // 集合的并
{   set<int> W = X, S = Y;
    W.merge(S);
    return W;
} // O(|X|+|Y|)
```

```
auto operator+(const set<int> &X, int v) // 每个元素都加上指定值
{   set<int> W;
    for(int u : X) W.insert(u + v);
    return W;
} // O(|X|)
```

```
auto &upper_remove(set<int> &X, int v) // 删除大于指定值的元素
{   X.erase(X.upper_bound(v), end(X));
    return X;
} // O(log|X|)
```

2. 修整

```
#include "SetSum.h"
auto 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(y), z = y;
    return W;
}
```

3. 近似方案

```
int ApproxSetSum(const set<int> &S, int t, double e)
{   set<int> L = {0};
    for(int x : S)
    {   L = L + (L + x); // Merge(L, L + x)
        L = Trim(L, e / S.size()); // 修整
        upper_remove(L, t); // 从L中删除超过t的元素
    }
    return max(L);
}
```

4. 测试

```
int main()
{   set<int> S = {105, 112, 128, 177, 243, 266};
    int t = 702;
    cout << "源集合: " << S << endl;
    cout << "近似值: " << ApproxSetSum(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
}
```

12.8 近似算法与$P=NP$

以旅行商问题的近似算法为例说明。

12.8.1 问题描述

【问题】给定一个边权值非负的无向图G和一个非负数t，从G中找出一个费用最小的旅行。

12.8.2 近似算法

1. 算法描述

对于给定的无向图，可以利用最小生成树算法设计寻找近似最优旅行的算法[①]。

① 在图中任意选取一个起始顶点[②]。

② 用Prim算法找出一棵以起始顶点为根的最小生成树。

③ 先根遍历最小生成树，得到一个顶点表。

④ 将起始顶点加到该顶点表末尾，按顶点表中各顶点的次序组成一条回路。该回路就是近似最优的旅行。

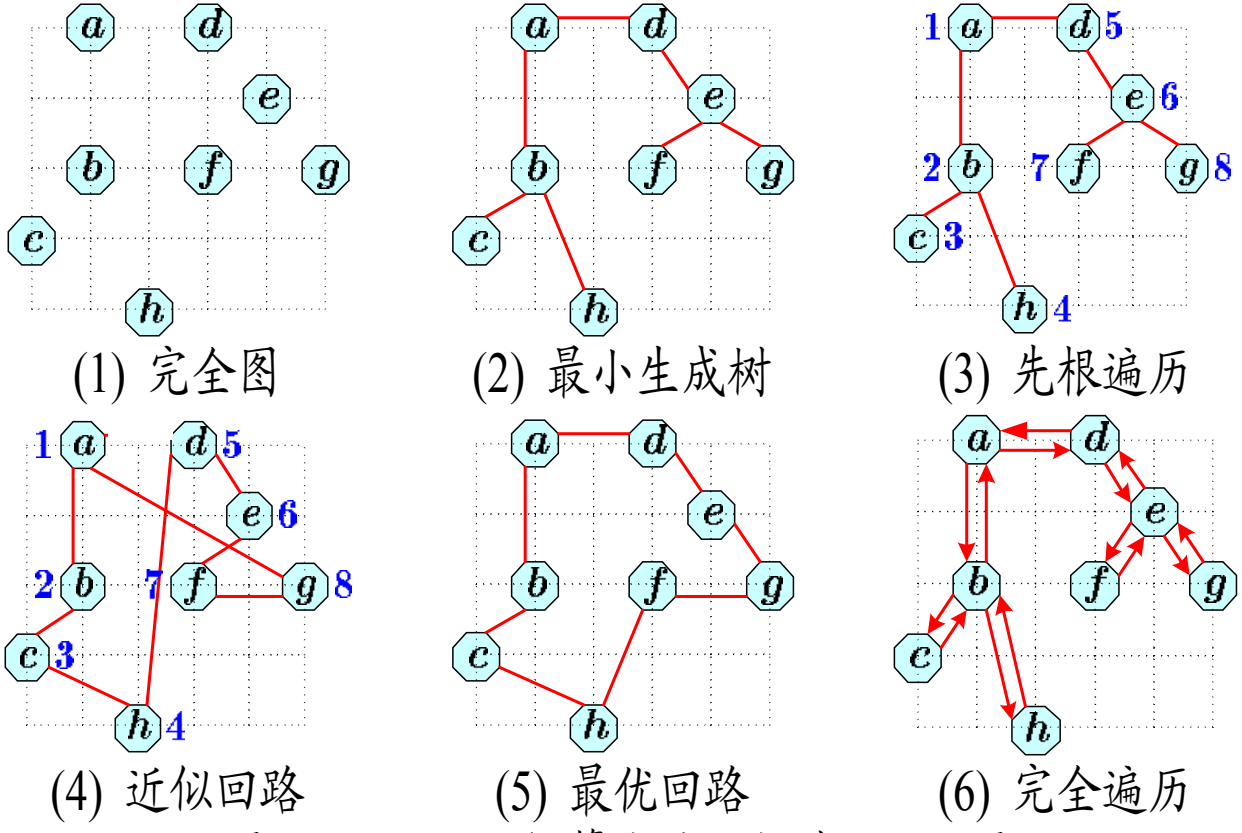
---

① 要使用该方法，必须保证给定图是一个完全图。

② 使用不同的起始顶点多次执行该算法，可能找到更优甚至最优的回路。

2. 举例说明

图12-4说明了上述TSP近似算法的运行过程及结果。



(1) 完全图　(2) 最小生成树　(3) 先根遍历

(4) 近似回路　(5) 最优回路　(6) 完全遍历

图12-4　TSP近似算法的运行过程及结果

3. 复杂性分析

由于给定的图是一个完全图，所以生成最小生成树的计算时间为 $\Theta(|V|^2)$。遍历最小生成树的计算时间为 $\Theta(|V|)$，其余计算时间 $\Theta(1)$。由此可知，上述算法的计算时间为 $\Theta(|V|^2)$。

12.8.3 近似算法的性能分析

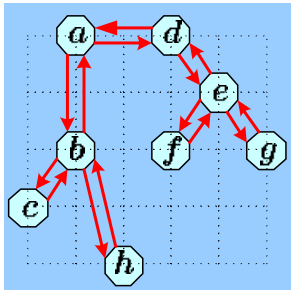
1. 满足三角不等式的情况

从实际应用中抽象出来的旅行商问题常具有一些特殊性质。比如，费用函数 c 往往具有三角不等式性质，即对任何3个顶点 $u,v,w\in V$，有：$c(u,w)\le c(u,v)+c(v,w)$。当图 G 中的顶点就是平面上的点，任意2顶点间的费用就是这2点间的欧氏距离时，费用函数 c 就满足三角不等式性质。

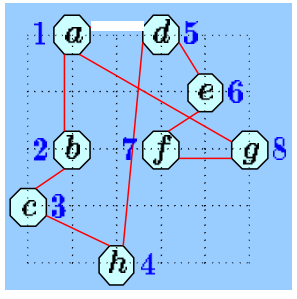
当费用函数满足三角不等式时，可以证明该算法的性能比不大于2。换句话说，若用 H^* 表示图G的最优旅行，用 H 表示该算法得到的旅行，则 $C(H)\le 2C(H^*)$，其中 $C(A)=\sum_{(u,v)\in A} c(u,v)$。

下面证明这一结论。

设 T 是图 G 的一棵最小生成树，T' 是 H^* 去掉一条边后形成的一棵生成树，则 $C(T)\le C(T')\le C(H^*)$。从 T 的根出发绕着 T 的外缘逆时针遍历 T 再回到根可以得到 G 的一条回路 W（这种遍历称为 T 的完全遍历，如图 12-4(6) 所示）。在本例中，$W=abcbhbadefegeda$。因为 W 经过 T 的每一条边恰好 2 次，所以 $C(W)=2C(T)\le 2C(H^*)$。

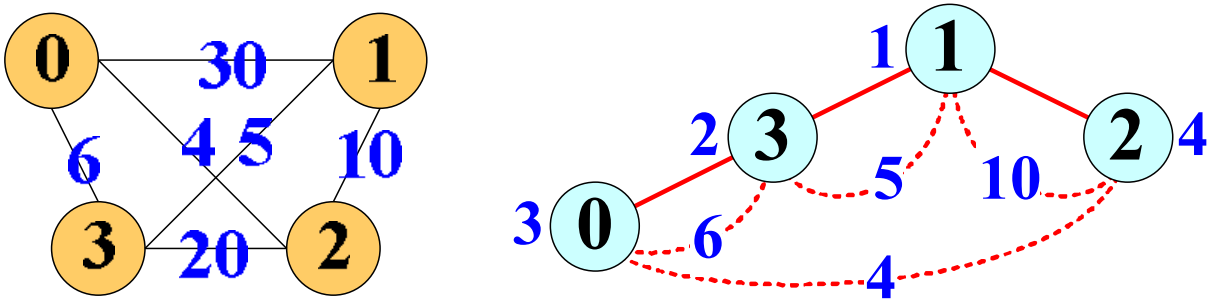


另外，近似回路 H 的顶点序列正好是 W 的顶点序列按顺序去掉重复出现的顶点(末尾顶点除外)以后保留下来的顶点(如图12-4(4)所示)。在本例中，$W=abcbhbadefegeda$，$H=abchdefga$。由费用函数的三角不等式性质可知 $C(H)\le C(W)$，从而 $C(H)\le 2C(H^*)$。



2. 一般的旅行商问题

上述近似算法没有用到费用函数的三角不等式性质。因此，该算法也适用于一般的旅行商问题。例如，对于图12-5(1)所示的网络，如果选取的起始顶点是1，则得到如图12-5(2)所示的最小生成树，由此得到的近似回路为1, 3, 0, 2, 1，等价于回路0, 2, 1, 3, 0，正好是最优旅行。



(1) 一个网络　　(2) 最小生成树及近似回路

图12-5　一个TSP实例

在一般情况下，当 $P \neq NP$ 时，对任何 $\rho > 1$，都不存在性能比为 ρ 的解旅行商问题的多项式时间近似算法。下面证明这一结论。

【证明】假设有一个性能比为 ρ 的解旅行商问题的多项式时间近似算法 A。对图 $G=(V,E)$，令 $n=|V|$，构造加权图 $G'=(V',E')$，其中 $V'=V$，$E'=\{(u,v)\,|\,u,v\in V',u\neq v\}$，且

$$w(u,v)=\begin{cases}1 & (u,v)\in E\\ \rho n & (u,v)\notin E\end{cases}$$

该构造过程显然可在多项式时间内完成。

设 G' 中旅行的最优值为 t^*，显然，$t^*\geq n$，且 G 有Hamilton回路当且仅当 $t^*=n$。若用算法 A 求解 G' 获得的近似值为 t，则 $t^*=n$ 当且仅当 $t\leq \rho n$。下面证明这一事实。若 $t^*=n$，则 $t\leq \rho t^*=\rho n$；若 $t^*>n$，则 $t>t^*>n-1+\rho n>\rho n$。

综上可得，图 G 有Hamilton回路当且仅当 $t\leq \rho n$。由此，得到Hamilton回路问题的一个多项式时间算法，从而 $P=NP$。

12.8.4 近似算法的C++描述*

1. 先根遍历最小生成树

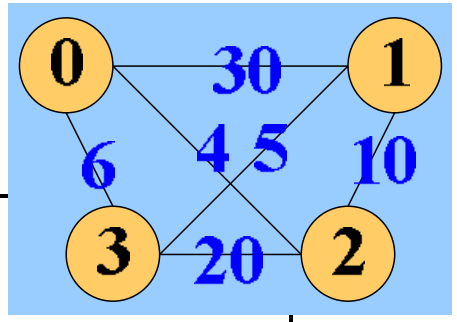
```
#include "Prim.h" // 使用Prim算法获得生成树
void PreOrder(const vector<int> &Prev, int v, vector<int> &X)
{   // 先根遍历树Prev(数组Prev保存各顶点的父亲)
    int n = Prev.size();
    X.push_back(v); // 访问根v
    for(int w = 0; w < n; ++w) // 遍历子树
        if(w != v && Prev[w] == v) // v是w的父亲
            PreOrder(Prev, w, X); // 访问子树
}
```

2. 算法描述

```
auto ApproxTSP(const Matrix<double> &G, int v)
{   vector<int> Prev;
    Prim(G, v, Prev); // 使用Prim算法获得生成树
    vector<int> X;
    PreOrder(Prev, v, X); // 先根遍历树Prev
    return X;
}
```

3. 测试程序

使用如图12-5(1)所示的网络。




```
int main()
{   Matrix<double> G = // 邻接矩阵
{   {0, 30, 4, 6},
        {30, 0, 10, 5},
        {4, 10, 0, 20},
        {6, 5, 20, 0}
    };
    auto X = ApproxTSP(G, 0);
    cout << "Root = 0: " << X << endl;
    X = ApproxTSP(G, 1);
    cout << "Root = 1: " << X << endl;
    cout << "Optimal: 0 2 1 3" << endl;
}
```

```
Root = 0: 0 2 3 1 // 59
Root = 1: 1 3 0 2 // 25
Optimal: 0 2 1 3 // 25
```

12.9 练习题

1、请使用一种程序设计语言或伪代码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或伪代码为顶点覆盖问题和最大团问题分别构造一个概率上的好算法。

3、请使用一种程序设计语言或伪代码为旅行商问题构造一个近似算法。

4、请使用一种程序设计语言或伪代码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。