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	贪心方法
教学目标及基 本 要 求	熟悉贪心方法的基本思想和控制流程，能够使用贪心方法求解背包问题、作业调度问题和最优生成树问题等常见问题。
教 学 重 点	背包问题和最优生成树的贪心算法
教 学 难 点	限期作业安排问题和最优生成树问题的贪心算法
教学内容与时 间 分 配	（1）贪心方法的基本思想和抽象流程（0.8课时） （2）几种常见作业调度问题的贪心算法（1课时） （3）最优生成树的贪心算法（1.2课时） 共计3课时。
习　　题	第5.4.1节（基础知识题），第5.4.2节（实验题）。

第5章　贪心方法

本章首先以装载问题和背包问题的贪心算法为例介绍贪心方法的基本思想和抽象流程，然后介绍最优生成树问题和几种常见作业调度问题的贪心算法。

- 贪心方法的基本思想和抽象流程（以装载问题和背包问题的贪心算法为例）
- 几种常见作业调度问题的贪心算法（活动安排问题、有期限的作业安排问题、多机调度问题）
- 最优生成树问题的贪心算法（Prim算法，Kruskal算法）

5.1 贪心方法的基本思想和抽象流程

5.1.1 贪心方法的基本思想

贪心方法在决策中总是作出在当前看来最好的选择。使用贪心方法设计的算法称为贪心算法。贪心方法并未考虑整体最优解，它所作出的选择只是在某种意义上的局部最优选择，不一定能够得到整体最优解（例如，0/1背包问题、多机调度问题和集合覆盖问题等）。但是，有相当一部分问题，使用贪心方法能够得到整体最优解（例如，装载问题、背包问题、最小生成树问题和活动安排问题等）。下面通过2个简单的例子介绍贪心方法的一般流程和贪心准则的选取。

5.1.2 装载问题※

1. 问题描述

在某货船上装载货物。假设有n个货箱，它们的体积相同，重量分别是$w_1, w_2, ..., w_n$，货船的最大载重量为c。目标是在船上装最多货箱，该怎样装？

每装一个货箱都需要想着在不超重的前提下让船装进更多的箱子，所以可以考虑尽量先装轻的货箱。如果使用x_i=1表示装第i个货箱，而x_i=0表示不装第i个货箱，rc表示货船剩余容量，则可设计出下列求解装载问题的贪心算法。

2. 算法描述

```
#include "algorithm.h"
auto Load(vector<double> &W, double M) // 重量数组, 货船容量
{   int n = W.size(); // 货箱个数
    sort(W.begin(), W.end()); // 重量数组W升序排列
    vector<bool> X(n, 0); // 解向量初始化为0
    double rc = M; // 货船剩余容量
    for(int i = 0; i < n and W[i] <= rc; ++i)
        X[i] = 1, rc -= W[i]; // 装入货箱i
    return X;
}
```

容易看出，该算法耗时 $O(n \log n)$ 。

5.1.3 背包问题

1. 问题描述

已知一个容量为c的背包和n件物品。其中，第i件物品的重量和价值分别是w_i和v_i，因而将物品i的一部分x_i（百分比）放进背包即获得$v_i x_i$的价值。问题是怎样装包（不能超重）使所获得的总价值最大，也就是怎样选取满足$\sum\{w_i x_i\}_{i=0}^{n-1} \le c$的百分比数组$\{x_i\}_{i=0}^{n-1}$，使得$\sum\{v_i x_i\}_{i=0}^{n-1}$最大。

2. 选取物品的原则

采用贪心方法，有下列几种原则可循。

（1）每次拣最轻的物品装包，可以尽量多装些物品。

（2）每次拣价值最大的物品装包，可以使得每次选择的价值最大。

（3）综合考虑物品的价值和重量，每次拣单位价值v_i/w_i最大的物品装包。

按原则（1）来装，只考虑多装些物品，但由于单位价值未必高，总价值不一定能达到最大。按原则（2）来装，每次选择的价值最大，但也可能占用了较大的空间，装的物品少，总价值未必能够达到最大。比较合理的原则是（3）。事实上，按照原则（3）来装，总价值确实能够达到最大。

3. 算法描述

```
#include "Knap.h" // 辅助程序, 将物品按单价降序排列, etc.
auto Knap(vector<double> &V, vector<double> &W, double M)
{   // 价值数组, 重量数组, 背包容量
    int n = min(V.size(), W.size()); // 物品数
    Sort(V, W); // 将物品按单价降序排列
    vector<double> X(n, 0); // 解向量初始化为零
    double rc = M; // 背包剩余容量
    int t; // 当前物品
    for(t = 0; t < n and W[t] <= rc; ++t)
        X[t] = 1, rc -= W[t]; // 可完整装包的物品完整装包
    if(t < n) X[t] = rc / W[t]; // 不能完整装包的物品部分装包
    return X;
}
```

容易看出，该算法耗时 $O(n \log n)$ 。

4. 辅助程序

（1）物品排序。将物品按单价降序排列，保存在文件Knap.h中。

```
// Knap.h
#pragma once
#include "Algorithm.h"
```

```
// 将物品按单价降序排列(价值数组, 重量数组)
void Sort(vector<double> &V, vector<double> &W)
{   struct Type // 临时的物品类型
    {   double v, w; // 价值, 重量
        // 按单价比较物品
        bool operator<(const Type &q) const { return v / w < q.v / q.w; }
    };
```

```
    int n = min(V.size(), W.size()); // 物品数
    vector<Type> X(n); // 临时数组
    for(int i = 0; i < n; ++i) // 两个数组复制到临时数组
        X[i] = {V[i], W[i]};
    sort(begin(X), end(X), greater<Type>()); // 临时数组降序排列
    for(int i = 0; i < n; ++i) // 临时数组复制回原数组
        V[i] = X[i].v, W[i] = X[i].w;
}
```

（2）输出结果。输出效益数组、重量数组和答案数组，保存在文件Knap.h中。

```
template<class T>
void PrintSolution(const vector<double> &V, const vector<double> &W,
     const vector<T> &X)
{   // 价值数组, 重量数组, 答案
     cout << "效益: " << V << endl;
     cout << "重量: " << W << endl;
     cout << "答案: " << X << endl;
}
```

5. 测试程序☼

```
int main()
{   int n = 5; // 物品数量
    vector<double> V = {6, 3, 5, 4, 6}; // 价值数组
    vector<double> W = {2, 2, 6, 5, 4}; // 重量数组
    double M = 13; // 背包容量
    auto X = Knap(V, W, M);
    PrintSolution(V, W, X);
}
```

```
效益: 6 3 6 5 4
重量: 2 2 4 6 5
答案: 1 1 1 0.833333 0
```

5.1.4 贪心准则和贪心算法的抽象流程

贪心方法主要用于处理优化问题。每个优化问题都是由目标函数和约束条件组成。满足约束条件的解称为可行解，而那些使得目标函数取极值的可行解称为最优解。如背包问题是一个优化问题，$\sum\{v_i x_i\}_{i=0}^{n-1}$ 是目标函数，$\sum\{w_i x_i\}_{i=0}^{n-1} \leq c$ 是约束条件，优化目标是使目标函数取得最大值。

1. 贪心准则

贪心方法在每一步决策中虽然没有完全顾及到问题的整体优化，但在局部择优中是朝着整体优化的方向发展的。为此，贪心方法首先要确定一个度量准则，每一步都按这个准则选取优化方案。如背包问题的贪心准则是选取单位价值最大的物品，而装载问题的贪心准则是选取最轻的货箱。

对于一个给定的问题，初看起来，往往有若干种贪心准则可选，但在实际上，大多数都不能使贪心方法达到问题的最优解。如背包问题的3种贪心准则中只有“选取单位价值最大的物品”才能使贪心方法达到问题的最优解。实践证明，选择能产生最优解的贪心准则是使用贪心方法的核心问题。

2. 贪心算法的抽象流程

下面给出贪心算法流程的伪代码。

```
function Greedy(X, n) // X[0 .. n)代表输入
{   Solution = {}; // 解向量初始化为空集
    for(i = 0; i < n; ++i)
    {   x = Select(X);
        if(Feasible(Solution, x))
            Solution = Union(Solution, x);
    }
    return Solution;
}
```

这里Select(X)是按照贪心准则选取X中的输入项；Feasible(solution, x)是判断已知解的部分solution与新选取的x的合并Union(solution, x)是否是可行解。过程Greedy描述了贪心算法的主要工作和基本控制流程。一旦给出一个特定的问题，就可将Select，Feasible和Union具体化并付诸实现。

5.2 作业调度问题

5.2.1 活动安排问题

1. 问题描述

已知n个活动$E=\{1, 2, ..., n\}$需要使用同一资源，第k个活动的开始和结束时间分别是s_k和f_k，其中$s_k<f_k$，$k = 1, 2, ..., n$。活动i与活动j相容是指$s_i>f_j$或$s_j>f_i$。活动安排问题是要在所给活动集合中选出最大的相容活动子集。

2. 贪心准则

基本思路是将结束时间早的活动尽量往前安排，给后面的活动留出更多的空间，达到安排最多活动的目标。据此，贪心准则应当是“在未安排的活动中挑选结束时间最早的活动”。在贪心算法中，首先需要将给定的一组活动按结束时间升序排列。

3. 例子

已知11个活动的开始时间和结束时间，按结束时间升序排列，如表5-1所示。容易看出，解集合为{1, 4, 8, 11}。

表5-1　11个活动的开始时间和结束时间

k	1	2	3	4	5	6	7	8	9	10	11
$s[k]$	1√	3	0	5√	3	5	6	8√	8	2	12√
$f[k]$	4	5	6	7	8	9	10	11	12	13	14

4. 算法描述

```
#include "algorithm.h"
// 将活动按照结束时间升序排列(开始时间, 结束时间)
void Sort(vector<double> &S, vector<double> &F);
```

```
// 活动安排(开始时间, 结束时间)
auto Action(vector<double> &S, vector<double> &F)
{   int n = min(S.size(), F.size()); // 活动数
    Sort(S, F); // 将活动按照结束时间升序排列
    vector<bool> X(n, 0); // 解向量初始为0
    X[0] = 1; // 将活动0加入解中
    for(int i = 1, j = 0; i < n; ++i)
        if(S[i] >= F[j]) X[i] = 1, j = i; // 将活动i加入解中
    return X;
}
```

容易看出，该算法耗时 $O(n \log n)$ 。

5. 辅助程序☼

将活动按照结束时间升序排列。

```
// 将活动按照结束时间升序排列(开始时间, 结束时间)
void Sort(vector<double> &S, vector<double> &F)
{   struct Type // 临时的活动类型
    {   double s, f; // 开始时间, 结束时间
        // 按照结束时间比较活动
        bool operator<(const Type &q) const { return f < q.f; }
    };
    int n = min(S.size(), F.size()); // 活动数
    vector<Type> X(n); // 临时数组
    for(int i = 0; i < n; ++i) // 两个数组复制到临时数组
        X[i] = {S[i], F[i]};
    sort(X.begin(), X.end()); // 临时数组升序排列
    for(int i = 0; i < n; ++i) // 临时数组复制回原数组
        S[i] = X[i].s, F[i] = X[i].f;
}
```

6. 测试程序☼

```
int main()
{   vector<double> S = {1, 3, 0, 5, 3, 5, 6, 8, 8, 2, 12};
    vector<double> F = {4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14};
    auto X = Action(S, F); // 求解
    cout << "开始: " << S << endl; // 开始: 1 3 0 5 3 5 6 8 8 2 12
    cout << "结束: " << F << endl; // 结束: 4 5 6 7 8 9 10 11 12 13 14
    cout << "答案: " << X << endl; // 答案: 1 0 0 1 0 0 0 1 0 0 1
}
```

5.2.2 有期限的作业安排问题※

1. 问题描述

有期限的作业安排问题要解决的是操作系统中单机、无资源约束且每个作业可在等量时间内完成的作业调度问题。这个问题的形式化描述如下。

（1）在一台机器上处理 n 个作业，每个作业可以在单位时间内完成。

（2）每个作业 i 都有一个期限值 d_i，$d_i>0$。

（3）作业按期完成，可以获得效益 p_i，$p_i>0$。

可行解是这 n 个作业的一个子集，该子集中的每个作业都能按期完成。目标是找一个子集 J，使得 J 中的每个作业都能按期完成，并且作业的效益和 $\sum\{p_i : i\in J\}$ 最大。

2. 贪心准则

（1）一个实例。这里通过表5-2所示的实例分析贪心准则的选取。容易看出，该实例的最优解应该正好包含2个作业，通过比较{1, 2}、{1, 3}、{1, 4}、{2, 3}和{3, 4}这5个可行解（{2, 4}不可行）可以得到该实例的最优解是{1, 4}，处理顺序是先处理作业4再处理作业1，在时间1完成作业4而在时间2完成作业1。

表5-2　一个有期限的作业安排问题

作业	1	2	3	4
效益P	100	10	15	20
期限D	2	1	2	1

（2）贪心准则的选取。目标是作业的效益和最大，因此考虑把目标函数作为度量标准。首先把作业按照效益值降序排列。对于表5-2所示的实例，作业根据效益值排序后为(1, 4, 3, 2)。求解时首先把作业1计入J，由于作业1的期限值是2，所以J={1}是一个可行解。接下来计入作业4。由于作业4的期限值是1而作业1的期限值是2，可以先完成作业4后再完成作业1，所以J={1, 4}是一个可行解。然后考虑作业3，但是作业3的期限值是2，计入J后无法保证J中的每一个作业都能按期完成，因此计入作业3后不能构成一个可行解。同理计入2后也不能构成一个可行解。由分析可知，把目标函数作为度量标准能够得到问题的最优解。

作业	1	2	3	4
效益P	100	10	15	20
期限D	2	1	2	1

3. 算法概要描述

```
// 作业按照效益降序排列，它们的期限值为D[i]
// J是能够在截止期限前完成的作业集合
function Job(D, n)
{   J = {};
    for(i = 1 to n)
        if('Union(J, i)中的作业都能按期完成')
            J = Union(J, i);
    return J;
}
```

Job描述的贪心算法对于作业排序问题总能得到最优解。在这个算法中需要解决的关键问题是如何判断作业 i 并入 J 后仍然能够保证 J 中的所有作业都能够按期完成。

	1	2	3	4	5	6	7
D	4	2	4	3	4	8	3
J_1	1						
J_2	2	1					
J_3	2	1	3				
J_4	2	4	1	3			
J_5					X		
J_6	2	4	1	3	6		
J_7			X	~~1~~	~~3~~	~~6~~	

4. 可行性判断

假设已经处理了 $i-1$ 个作业，并且有 m 个作业已经纳入 J 中，分别是 J_1，J_2，...，J_m，且 $D[J_1]\le D[J_2]\le\ldots\le D[J_m]$。现考虑作业 i 能否并入到 J 中形成一个可行解。要判断作业 i 能否并入到 J 中形成一个可行解，就要看能否在 J 中找到一个位置，使得 i 插入后 J 中的所有 $m+1$ 个作业都能按期完成，且 $D[J_1]\le D[J_2]\le\ldots\le D[J_{m+1}]$。现在的关键问题变成了如何为作业 i 找到一个合适的位置。

（1）利用插入排序的思想为作业 i 找到一个待插入位置。将作业 i 的期限值与 J 中的作业 m、作业 $m-1$、...、作业1比较，直到找到第一个使得 $D[J_r]\le D[i]$ 的位置 r，位置 $r+1$ 就是作业 i 的待插入位置。

（2）考虑作业 i 插入后 J 中的作业是否都能按期完成。位置 $r+1$ 之前的作业都没有变动位置，肯定能按期完成。从位置 $r+1$ 开始的作业都需要后移一个位置，执行时间推迟了一个单位。一个可行解应该使这些作业在推迟执行时间后还能按期完成，因此满足 $D[J_r]\le r$ 的作业不能后移。

综合上述两条，要寻找作业 i 的待插入位置，需要从 J 中已有作业的最后一个作业开始往第1个作业扫描。用变量 r 表示正在扫描的位置。如果 $D[J_r]>D[i]$ and $D[J_r]>r$ 则继续扫描；否则，要么找到了待插入位置 $r+1$，要么某个作业满足 $D[J_r]\leq r$，不能插入作业 i。

（3）确定作业 i 能否插入。找到待插入位置不能保证作业 i 可以插入。因为如果 $D[i]\leq r$，则插入后作业 i 要在第 $r+1$ 个时间才开始执行，但是作业 i 必须在 $r+1$ 时间之前完成。所以还要做第3个判断，如果 $D[i]>r$ 则作业 i 能够插入。

	1	2	3	4	5	6	7
D	4	2	4	3	4	8	3
J_1	1						
J_2	2	1					
J_3	2	1	3				
J_4	2	4	1	3			
J_5					X		
J_6	2	4	1	3	6		
J_7			X	1	3	6	

5. 算法描述

用数组D存放作业的期限值，按效益降序排列，其中作业0是虚构的，期限为0，便于将作业插入位置1（用位置表示时刻）；用数组J存放成为可行解的作业；用m表示选中的作业数；用i指示正在处理的作业；用r表示数组J的当前位置（当前时刻）。

```
#include "algorithm.h"
auto Job(const vector<int> &D) // D[0]=0是虚构作业，必需
{   int n = D.size(), m = 0; // 总作业数，已选作业数
    vector<int> J(n); // 选取的作业
    J[0] = 0, ++m; // 选取虚构作业0，已选作业数加1
    for(int i = 1; i < n; ++i)
    {   int r; // 从最后一个已选作业开始寻找作业i的待插入位置r+1
        for(r = m - 1; D[J[r]] > D[i] and D[J[r]] > r; --r) {}
        if(D[J[r]] <= D[i] and D[i] > r)
        {       for(int k = m - 1; k > r; --k) // 后移r后的作业
                    J[k + 1] = J[k];
                J[r + 1] = i, ++m; // 作业i放入r+1位置，已选作业数加1
        }
    }
    J.resize(m); // 只选取了m个作业
    return J; // 返回已选作业
}
```

用数组 D 存放作业的期限值，按效益降序排列，其中作业 0 是虚构的，期限为 0，便于将作业插入位置 1（用位置表示时刻）；用数组 J 存放成为可行解的作业；用 m 表示选中的作业数；用 i 指示正在处理的作业；用 r 表示数组 J 的当前位置（当前时刻）。

6. 复杂性分析

对于Job，有两个测量复杂度的参数，即作业数 n 和包含在解中的作业数 s 。寻找位置至多迭代 k 次，每次迭代的时间为 $O(1)$ 。若if行的条件为真，则插入作业 i ，需要 $O(k-r)$ 时间。因此，每次迭代的总时间是 $O(k)$ ，共迭代 $n-1$ 次。如果 s 是 k 的最终值，即 s 是最终解的作业数，则该算法所需要的总时间是 $O(sn)$ 。因为 $s \leq n$ ，所以该算法的总时间是 $O(n^2)$ 。当 $d_i = n-i+1$ 时，插入过程正好相当于对逆序数组进行插入排序，因此该算法的最坏情况时间为 $\Theta(n^2)$ 。

```
for(int i = 1; i < n; ++i)
{   int r; //寻找作业i的待插入位置r+1
    for(r = m - 1; D[J[r]] > D[i] and D[J[r]] > r; --r) {}
    if(D[J[r]] <= D[i] and D[i] > r)
    {     for(int k = m - 1; k > r; --k) // 后移r后的作业
            J[k + 1] = J[k];
          J[r + 1] = i, ++m; // 作业i放入r+1位置
    }
}
```

7. 测试程序

```
int main()
{   vector<int> D = {0, 4, 2, 4, 3, 4, 8, 3}; // 必须有D[0]=0
    for(auto j : Job(D))
        cout << "(" << j << ", " << D[j] << ") ";
}
```

```
(0, 0) (2, 2) (4, 3) (1, 4) (3, 4) (6, 8)
```

	1	2	3	4	5	6	7
D	4	2	4	3	4	8	3
J_1	1						
J_2	2	1					
J_3	2	1	3				
J_4	2	4	1	3			
J_5					X		
J_6	2	4	1	3	6		
J_7			X	1	3	6	

5.2.3 多机调度问题

1. 问题说明

【含义】有 n 个独立作业（每个作业各自需要的处理时间是已知的），由 m 台（ $m \geq 2$ ）相同的机器并行处理。规定每个作业都可以由任何一台机器处理，但未完成前不能中断，也不能拆分为更小的子作业。

【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这些作业能够在尽可能短的时间内由 m 台（ $m \geq 2$ ）相同的机器处理完毕。

【贪心准则】该问题到目前为止还没有一个有效的解法，利用贪心方法可以获得较好的近似解。可以采用“最长处理时间作业优先处理”的贪心准则。

2. 一个例子

有7项独立作业J_1、J_2、J_3、J_4、J_5、J_6、J_7，由三台机器M_1、M_2、M_3处理。各作业分别耗时2、14、4、16、6、5、3。将作业按耗时长短排列成J_4、J_2、J_5、J_6、J_3、J_7、J_1。采用“将耗时最长的待安排作业首先安排给能最早空闲的机器”的调度方案，如图5-1所示。

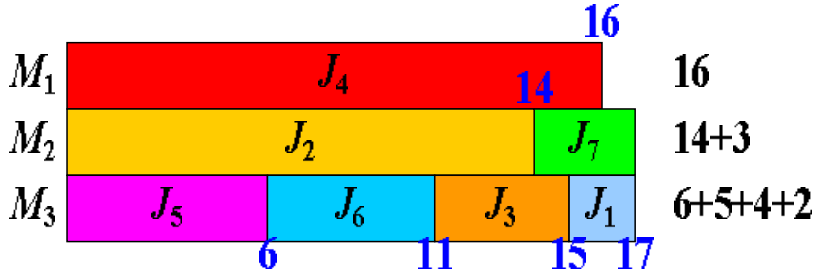


图5-1　一个多机调度的例子

（1）J_4、J_2、J_5分别安排给M_1、M_2、M_3。
（2）M_1、M_2、M_3空闲的时刻分别是16、14、6，J_6安排给M_3。
（3）M_1、M_2、M_3空闲的时刻分别是16、14、11，J_3安排给M_3。
（4）M_1、M_2、M_3空闲的时刻分别是16、14、15，J_7安排给M_2。
（5）M_1、M_2、M_3空闲的时刻分别是16、17、15，J_1安排给M_3。
全部作业安排完毕，所需时间为17（恰好得到最优解）。

3. 机器的表示

机器包含机器号*i*和已使用时间tm两个属性，优先选用最早空闲的机器。

```
#include "algorithm.h"
struct Machine { int i, tm; }; // 机器号, 使用时间
bool operator<(const Machine &x, const Machine &y)
{   return x.tm < y.tm; // 比较使用时间
}
auto InitMachine(int m) // 初始化机器和最小堆
{   minheap<Machine> H;
    for(int i = 0; i < m; ++i)
        H.push({i, 0});
    return H;
}
```

4. 算法描述

```
auto LPT(vector<int> &J, int m) // n是作业数, m是机器数, n>m
{   sort(J.begin(), J.end()); // 作业按照所需处理时间升序排列
    auto H = InitMachine(m); // 初始化机器和最小堆
    int n = J.size(); // 作业数
    vector<int> X(n); // 当前调度
    for(int t = n - 1; t >= 0; --t) // 从处理时间最长的作业开始
    {   auto [i, tm] = H.top(); H.pop(); // 选用最早空闲的机器
        X[t] = i, tm += J[t]; // 作业t安排到机器i
        H.push({i, tm});
    }
    return X;
}
```

5. 测试程序☼

（1）输出结果。

```
void Schedule(vector<int> &J, int m, vector<int> &X)
{   vector<int> tm(m, 0);
    int n = J.size();
    for(int t = 0; t < n; ++t)
        tm[X[t]] += J[t];
    cout << "J:\t" << to_string(J, n, "\t") << endl;
    cout << "M:\t" << to_string(X, n, "\t") << endl;
    cout << "T:\t" << to_string(tm, m, "\t") << endl;
    auto avg = accumulate(J.begin(), J.end(), 0) / (double)m;
    cout << "Avg:\t" << avg << endl;
}
```

（2）举例说明。使用4台机器处理9个作业。

```
int main()
{   int n = 9, m = 4; // 作业数和机器数
    vector J = {21, 7, 3, 6, 29, 20, 6, 18, 7}; // 每个作业需要的处理时间
    auto X = LPT(J, m);
    Schedule(J, m, X);
}
```

```
J:      29    21    20    18    7     7     6     6     3
M:      0     2     3     1     1     3     2     1     3
T:      29    31    27    30
Avg:    29.25
```

```
// 一种最优调度
J:      29    21    20    18    7     7     6     6     3
M:      0     1     2     3     1     2     3     3     2
T:      29    28    30    30
```

5.3 最小生成树

连通的加权图中总权值最小的生成树称为该图的最小生成树。贪心方法可以很好地求解最小生成树问题，这里介绍Prim算法和Kruskal算法。

5.3.1 Prim算法

1. 基本思想

如图5-2所示。

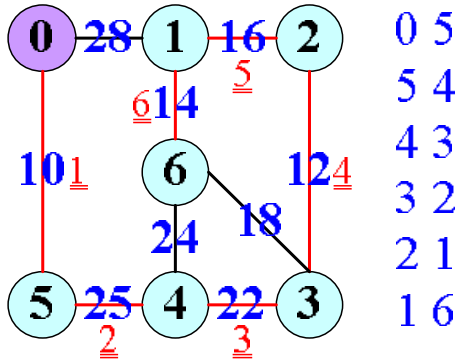


图5-2 Prim算法的基本思想

（1）从图中选取一个顶点（如0）；

（2）假设一棵子树T已经确定；

（3）从图中选取一条不在T中的权值最小的边e，使得$T \cup \{e\}$仍是一棵子树。

2. 算法说明

数组prev存放最小生成树中每个顶点的父亲，元素prev[v]为当前子树中所有与顶点v相邻且边权值最小的顶点，初始化为根顶点。

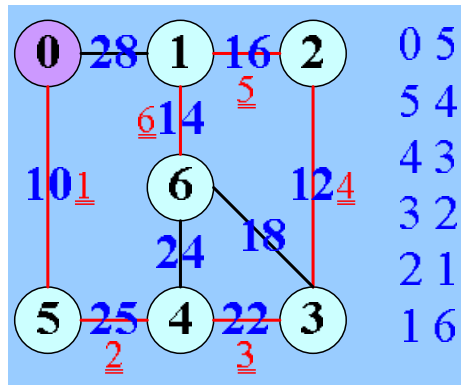
数组S用于标志顶点是否已经选进当前子树中。如果v已经在当前子树中，则$S[v]$=1，否则，$S[w]$=0。初始值为0。

方法是在树外顶点中寻找使(prev[v], v)最小的顶点v，然后将v选进生成树中，同时更新树外顶点的prev值。

例如，对于图5-2所示的加权图，从顶点0开始执行的执行过程演示如图5-3所示。

1	2	3	4	5	6
0: 28	0: ∞	0: ∞	0: ∞	0: 10	0: ∞
0: 28	0: ∞	0: ∞	5: 25	√	0: ∞
0: 28	0: ∞	4: 22	√	√	4: 24
0: 28	3: 12	√	√	√	3: 18
2: 16	√	√	√	√	3: 18
√	√	√	√	√	1: 14

图5-3　Prim算法的执行过程演示



3. 算法描述

需要多次使用，保存在文件Prim.h中。

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

```
bool Prim(const Matrix<double> &G, int v, vector<int> &prev)
{   // 邻接矩阵, 起始顶点(根), 各顶点的双亲(生成树)
    int n = G.rows(); // 顶点数
    prev.assign(n, v); // 各顶点的双亲初始化为根v
    vector<bool> S(n, false); // 各顶点是否已选, 初始时未选
    S[v] = true; // 选取v
```

```
    for(int i = 0; i < n - 1; ++i)
    {   // 寻找已选顶点到未选顶点权值最小的顶点
        double min = inf; // min初始为无穷大
        for(int w = 0; w < n; ++w)
            if(not S[w] and G(prev[w], w) < min)
                min = G(prev[w], w), v = w;
        if(isinf(min)) return false; // 不连通
        S[v] = true; // 选取v
        for(int w = 0; w < n; ++w) // 更新未选顶点的父亲
            if(not S[w] and G(v, w) < G(prev[w], w))
                prev[w] = v;
    }
    return true; // 连通
}
```

4. 时间复杂度分析

初始化耗时$O(n)$。在for循环的循环体中，寻找已选顶点到未选顶点权值最小的顶点耗时$O(n)$，更新未选顶点的父亲耗时$O(n)$，执行次数不超过n-1，因而整个循环共耗时$O(n^2)$。所以，Prim算法的时间复杂度为$O(n^2)$。

5. 测试程序☼

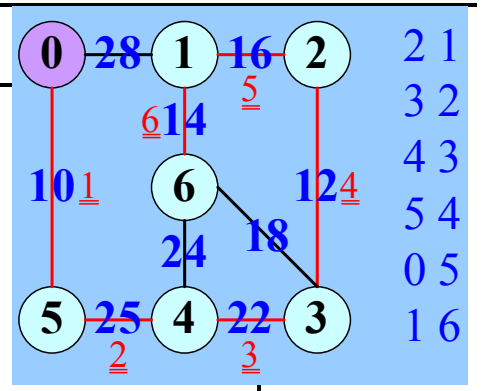
（1）获取生成树的边。因为prev[v]存放最小生成树中顶点v的父亲，(prev[v], v)正好是生成树的一条边，且根满足prev[v] == v，所以可以使用下列程序显示生成树的每条边。

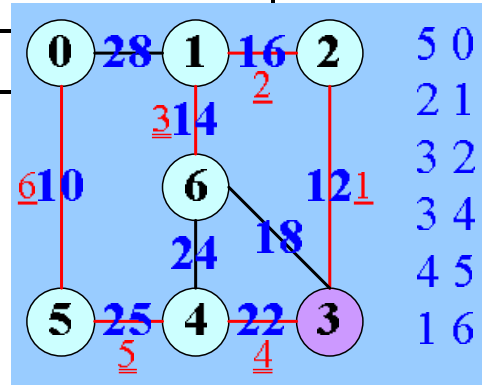
```
#include "Prim.h"
void print(const vector<int> &prev) // 显示生成树的每条边
{   int n = prev.size(); // 顶点数
    for(int v = 0; v < n; ++v)
        if(prev[v] != v) // 根满足prev[v] == v
            cout << "(" << prev[v] << ", " << v << ") ";
}
```

（2）一个实例。使用图5-2所示的加权图。

```
int main()
{   Matrix<double> G = // 邻接矩阵
    {   {0, 28, inf, inf, inf, 10, inf},
        {28, 0, 16, inf, inf, inf, 14},
        {inf, 16, 0, 12, inf, inf, inf},
        {inf, inf, 12, 0, 22, inf, 18},
        {inf, inf, inf, 22, 0, 25, 24},
        {10, inf, inf, inf, 25, 0, inf},
        {inf, 14, inf, 18, 24, inf, 0}
    };
    int n = G.rows();
    vector<int> prev; // 保存各顶点父亲
    if(Prim(G, 0, prev)) print(prev);
    else cout << "不是连通图";
}
```

(2, 1) (3, 2) (4, 3) (5, 4) (0, 5) (1, 6)





5.3.2 Kruskal算法※

1. 基本思想

如图5-4所示。

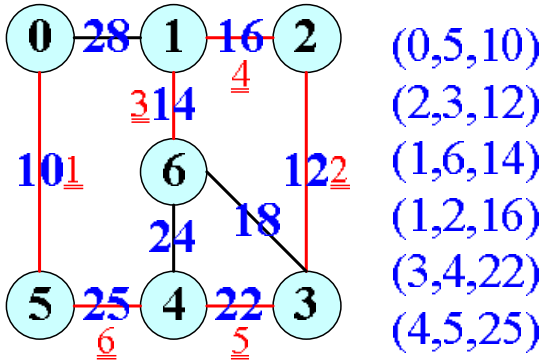


图5-4 Kruskal算法的基本思想

（1）从图中选取一条权值最小的边；

（2）假设已经选好一组边L；

（3）从图中选取一条不在L中的权值最小的边e，使得$L \cup \{e\}$形成的子图不含圈。

2. 边的表示

```
#include "algorithm.h"
#include "UnionFind.hpp" // 合并查找结构
```

```
struct Edge
{   int u, v; // 顶点
    double w; // 权值
};
```

```
bool operator<(const Edge &e, const Edge &f)
{   return e.w < f.w; // 比较边成本
}
```

3. 算法描述

Kruskal算法的特点是首先将图的边集合按照边成本升序排列，然后从边集合中依次选取边形成最小生成树。这里的关键是怎样判断新边(u, v)能否加入到树中。这可以使用合并查找结构实现，依据是如果边(u, v)的两端点在同一棵树中，则加入该边会形成圈。

```
bool Kruskal(vector<Edge> &Ed, int n, vector<Edge> &X)
{   // 图的边集合, 顶点数, 生成树的边集合
    if(Ed.size() < n - 1) return false; // 不是连通图
    sort(Ed.begin(), Ed.end()); // 将图的边集合按照边成本升序排列
    UnionFind U(n); // 合并/查找结构, 初始时表示n个不同分枝
    for(auto e : Ed) // 依次选取边
    {   int u = U.Find(e.u), v = U.Find(e.v); // 两端点所在分枝u, v
        if(u == v) continue; // u, v是同一分枝, 考虑下一边
        U.Union(u, v); // 合并分枝u, v(此时, u, v是根), 耗时O(1)
        X.push_back(e); // 选取该边
        if(X.size() >= n - 1) return true; // 生成树构造完毕
    }
    return false; // 不是连通图
}
```

4. 时间复杂性分析

UnionFind初始化耗时$O(n)$。排序耗时$O(e\log(e))$。一次查找操作耗时$O(\log(n))$，最多$2e$次，总耗时$O(e\log(n))$。一次合并操作耗时$O(1)$（因为直接使用根合并），最多n-1次，总耗时$O(n)$。因为当$e<n$-1时不需要使用该算法，所以该算法总耗时$O(e\log(e))$。

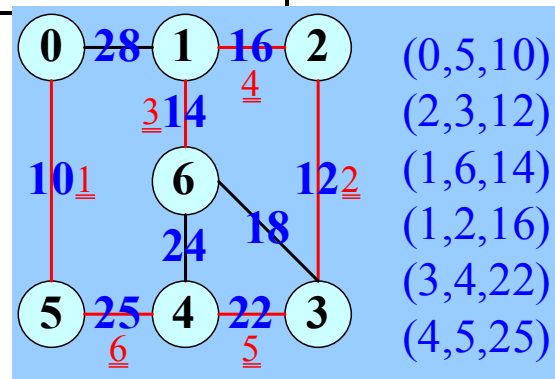
5. 测试程序

使用如图5-4所示的图。

```
ostream &operator<<(ostream &out, const Edge &e) // 输出边
{   return out << '(' << e.u << ", " << e.v << ", " << e.w << ')';
} // 输出边的3个分量。
```

```
int main()
{   int n = 7;
    vector<Edge> X, Ed =
    {   {0, 1, 28}, {0, 5, 10}, {1, 2, 16}, //
        {1, 6, 14}, {2, 3, 12}, {3, 4, 22}, //
        {3, 6, 18}, {4, 5, 25}, {4, 6, 24} //
    };
    if(Kruskal(Ed, n, X)) cout << X << endl;
    else cout << "不是连通图" << endl;
}
```

```
(0, 5, 10) (2, 3, 12) (1, 6, 14) (1, 2, 16) (3, 4, 22) (4, 5, 25)
```



5.3.3 两种算法的比较※

1. 复杂性

Prim算法的时间复杂度为$O(n^2)$，Kruskal算法的时间复杂度为$O(e\log(e))$。容易看出，若存在小于2的正数α，使得$e=O(n^{\alpha})$，则Kruskal算法的时间复杂度为$O(n\log(n))$，选用Kruskal算法可能耗时较少，否则，因为$e=O(n^2)$，Kruskal算法的时间复杂度可能达到$O(n^2\log(n))$，选用Prim算法可能耗时较少。

2. 生成树的结构

Prim算法使用数组prev记录了生成树中每个顶点的父亲，也就是记录了生成树的结构，而Kruskal算法只记录了生成树的每一条边，需要另想办法构造生成树的结构。

【注】如果不准备完整阅读“难问题的近似算法”和“问题的复杂性”两章，则可在了解相关概念后选择补充阅读下列小节并选作相关练习题。

- 10.2.2　顶点覆盖问题的最大匹配算法。
- 10.3.1　顶点覆盖问题的大度数优先算法。
- 10.3.3　最大团的近似算法。
- 11.6.1　旅行商问题的近似算法。

5.4 练习题

5.4.1 基础知识题

1. 请使用一种程序设计语言描述使用贪心方法求解装载问题的算法。
2. 请使用一种程序设计语言描述使用贪心方法求解背包问题的算法。
3. 请使用一种程序设计语言描述使用贪心方法求解活动安排问题的算法。
4. 使用一种程序设计语言描述最小生成树的Prim算法。已知：G 是具有 n 个顶点的无向加权图，Matrix是矩阵类型。
5. 使用一种程序设计语言描述最小生成树的Kruskal算法。已知：E 是无向加权图 G 的边集合，并且已经按照权值排序，Matrix和Edge分别是矩阵类型和边类型。

5.4.2 实验题

1. 请使用一种程序设计语言描述使用贪心方法求解有期限的作业安排问题的算法。

2. 请使用一种程序设计语言描述使用贪心方法求解多机调度问题的算法。

3. 已知容量为 M 的背包和 n 件物品。第 i 件物品的重量为 w_i，价值是 p_i。因而将物品 i（不能分割）放进背包即获得 p_i 的价值。问题是：怎样装包使所获得的价值最大。请使用贪心方法设计求解该问题的近似算法，并使用一种程序设计语言描述该算法。

4. 分别给出一个用下述算法能够获得最大团的例子和不能获得最大团的例子。

```
function Clique(G) // G是图的邻接矩阵
{   X = {}; // X是当前团, 初始为空集
    for(i = 0 to n - 1) // n是图的顶点数
        if(Connected(G, X, i)) // 顶点i与已有团每个顶点都相邻
            X << i; // 顶点i加入当前团
    return X; // 返回获得的团
}
```

5.4.3 进阶训练题

1. 求最大团的近似算法。

（1）使用贪心方法设计一种求最大团的多项式时间近似算法（该算法得到的结果可能不是最大团）。

（2）分别给出一个用该算法能够获得最大团的例子和不能获得最大团的例子。

（3）分析该算法的复杂性。

【提示】可参考实验题4，但有区别。

2. 旅行商问题的贪心算法。使用贪心方法在多项式时间内从一个无向加权图中选取一条总权值尽可能小（不一定最小）的Hamilton回路。

3. 子集覆盖问题的贪心算法。已知一个有限集X和X的一组子集F，如果F中所有子集的并等于X，则称F覆盖X。请使用贪心方法在多项式时间内从F中选取一组覆盖X且子集数目尽可能少（不一定最少）的子集。

4. 子集和问题的贪心算法。使用贪心方法在多项式时间内从一个正整数集W中选取一个和尽可能接近（不一定最接近）正整数t的子集。

5. 均衡划分问题的贪心算法。使用贪心方法在多项式时间内将一个正整数集分为和尽可能接近（不一定最接近）的两个子集。

6. 着色问题的贪心算法。请使用贪心方法为一个需要着色的无向图在多项式时间内选取一种颜色数目尽可能少（不一定最少）的着色方案。

7. 会议室调度问题的贪心算法。某宾馆有m个会议室，有n个会议申请在该宾馆召开。已知每个会议的开始时间和结束时间，请问如何调度才能使用尽可能少的会议室召开尽可能多的会议。请使用贪心方法求解该问题。