# 第6章　贪心方法

本章首先介绍贪心方法的基本思想和抽象流程，然后给出几个实例，包括装载问题和背包问题的贪心算法、几种常见作业调度问题的贪心算法和两个图论优化问题的贪心算法。

- 贪心方法的基本思想和抽象流程
- 装载问题和背包问题的贪心算法
- 几种常见作业调度问题的贪心算法（活动安排问题、限期作业安排问题、多机调度问题）*
- 最优生成树的贪心算法（Prim算法和Kruskal算法）
- 单源最短路径的贪心算法（Dijkstra算法）

# 6.1 贪心方法的基本思想和抽象流程

## 6.1.1 贪心方法的基本思想

贪心方法在决策中总是做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，它所做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、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、单源最短路径问题、最小生成树问题、作业排序问题等。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。

## 6.1.2 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 $w_1, w_2, \ldots, w_n$，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物。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 GreedyLoad(double W[], int n, double M)
{   // W是重量数组, M是货船容量
    sort(W, W + n); // 升序排列重量数组W
    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 && 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 $O(n \log n)$ 。

## 3. 测试程序*

```
int main()
{   int n = 5; // 货箱个数
    double W[] = {3, 2, 6, 5, 4}, M = 10; // 重量数组, 货船容量
    auto X = GreedyLoad(W, n, M); // 求解
    cout << "重量: " << to_string(W, n) << endl;
    cout << "答案: " << to_string(X, n) << endl;
}
```

```
重量: 2 3 4 5 6
答案: 1 1 1 0 0
```

## 6.1.3 背包问题

### 1. 问题描述

已知容量为 $c$ 的背包和 $n$ 件物品。第 $i$ 件物品的重量为 $w_i$，价值是 $v_i$。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。问题是怎样装包使所获得的价值最大，也就是求使得 $\sum_{i=1}^{n} v_i x_i$ 最大的 $x_1, x_2, \ldots, x_n$，其中 $0 \le x_i \le 1, w_i > 0, v_i > 0$，$\sum_{i=1}^{n} w_i x_i \le c$。

**2. 贪心准则**

采用贪心方法，有下列几种原则可循。

① 每次捡最轻的物品装。

② 每次捡价值最大的装。

③ 每次装包时既考虑物品的重量又考虑物品的价值，也就是说每次捡单位价值 $v_i / w_i$ 最大的装。

按原则①来装只考虑到多装些物品，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，但同时也可能占用了较大的空间，装的物品少，未必能够达到总价值最大。比较合理的原则是③。事实上，按照原则③来装，确实能够达到总价值最大。

## 3. 算法描述

```
#include "KNAP.h" // 辅助程序，将物品按单价降序排列，etc.
auto GreedyKNAP(double V[], double W[], int n, double M)
{   // 价值数组V，重量数组W，背包容量M
    Sort(V, W, n); // 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 && 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. 辅助程序及测试*

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>
using namespace std;
```

```
void Sort(double V[], double W[], int n) // 将物品按单价降序排列
{   struct oType { double v, w; }; // 物品类型
    auto comp = [](oType p, oType q) // 物品的比较方法
    {   return p.v / p.w > q.v / q.w;
    };
    oType X[n]; // 临时物品数组
    for(int i = 0; i < n; ++i) // 将两个数组复制到临时数组
        X[i] = {V[i], W[i]};
    sort(X, X + n, comp); // 将临时物品数组排序
    for(int i = 0; i < n; ++i) // 将临时物品数组复制回原数组
        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。

```
#include "ios.hpp" // to_string, io等, 第1章介绍
```

```
void PrintSolution(double V[], double W[], const auto &X, int n)
{   cout << "效益: " << to_string(V, n) << endl;
    cout << "重量: " << to_string(W, n) << endl;
    cout << "答案: " << to_string(X, n) << endl;
}
```

③ 测试程序。测试程序和测试结果如下。

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

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

### 6.1.4 贪心准则的选取和贪心方法的抽象流程

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，函数 $\sum_{i=1}^{n} v_i x_i$ 是目标函数，而 $\sum_{i=1}^{n} w_i x_i \leq c$ 是约束条件，优化目标是使目标函数取得最大值。

**1. 贪心准则的选取**

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，每一步都是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，而装载问题的贪心方法选取的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。如背包问题的三种贪心准则中只有“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(A, n) // A[1 .. n]代表输入
{   Solution = {}; // 解向量初始化为空集
    for(i = 1 to n)
    {   x = Select(A);
        if(Feasible(Solution, x))
            Solution = Union(Solution, x);
    }
    return Solution;
}
```

这里Select(A)是按照贪心准则选取A中的输入项；Feasible(solution, x)是判断已知解的部分solution与新选取的x的结合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。

# 6.2 作业调度问题*

## 6.2.1 活动安排问题

### 1. 问题描述

已知 $n$ 个活动 $E=\{1,2,\ldots,n\}$ 要求使用同一资源，第 $k$ 个活动的开始和结束时间分别是 $s_k$ 和 $f_k$，其中 $s_k<f_k$，$k=1,2,\ldots,n$。活动 $k$ 与活动 $j$ 称为相容是指 $s_k>f_j$ 或者 $s_j>f_k$。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非减排序。

### 3. 例子

设待安排的11个活动的开始时间和结束时间按结束时间的非减次序排列如表6-1所示，解集合为{1, 4, 8, 11}。

表6-1　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"
void Sort(double S[], double F[], int n); // 将活动按照结束时间升序排列
auto GreedyAction(double S[], double F[], int n)
{   // S是开始时间数组, F是结束时间数组
    Sort(S, F, n); // 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. 辅助程序及测试*

① 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
void Sort(double S[], double F[], int n)
{   struct oType { double s, f; }; // 活动类型
    auto comp = [](oType p, oType q) // 活动的比较方法
    {   return p.f < q.f;
    };
    oType X[n]; // 临时数组
    for(int i = 0; i < n; ++i) // 将时间数组复制到临时数组
        X[i] = {S[i], F[i]};
    sort(X, X + n, comp); // 将临时数组排序
    for(int i = 0; i < n; ++i) // 将临时数组复制回原数组
        S[i] = X[i].s, F[i] = X[i].f;
}
```

② 测试程序。

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

```
开始: 1 3 0 5 3 5 6 8 8 2 12
结束: 4 5 6 7 8 9 10 11 12 13 14
答案: 1 0 0 1 0 0 0 1 0 0 1
```

## 6.2.2 带期限的作业安排问题

### 1. 问题描述

带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为：

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 $J$。$J$ 中的每个作业都能按期完成。目标是找一个子集 $J$，$J$ 中的每个作业都能按期完成，并且使得作业的效益和 $\sum_{i\in J} p_i$ 最大。

## 2. 一个例子

待求解的问题实例如表6-2所示。

表6-2　一个限期作业安排问题

| 作业 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| 效益$P$ | 100 | 10 | 15 | 20 |
| 期限$D$ | 2 | 1 | 2 | 1 |
| 优先级 | 1 | 4 | 3 | 2 |

这个问题的可行解如表6-3所示，最优解为第7个，所允许的处理顺序是：先处理作业4，在处理作业1，在时间0开始处理作业4而在时间2完成对作业1的处理。

表6-3　表6-2的可行解

| | 可行解 | 处理顺序 | 效益值 |
|---|---|---|---|
| 1 | {1} | 1 | 100 |
| 2 | {2} | 2 | 10 |
| 3 | {3} | 3 | 15 |
| 4 | {4} | 4 | 20 |
| 5 | {1, 2} | 2, 1 | 110 |
| 6 | {1, 3} | 1, 3或3, 1 | 115 |
| 7 | {1, 4} | 4, 1 | 120 |
| 8 | {2, 3} | 2, 3 | 25 |
| 9 | {3, 4} | 4, 3 | 35 |

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $P$ | 100 | 10 | 15 | 20 |
| $D$ | 2 | 1 | 2 | 1 |
| | 1 | 4 | 3 | 2 |

### 3. 贪心准则

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。对于上例，作业根据效益值排序后为作业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 $J=\{1,4\}$ 是一个可行的解。然后考虑作业3，但是作业3的期限值是2，计入 $J$ 后就没有办法保证 $J$ 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。

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| $P$ | 100 | 10 | 15 | 20 |
| $D$ | 2 | 1 | 2 | 1 |
| | 1 | 4 | 3 | 2 |

## 4. 算法的概略描述及说明

```
// 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 GreedyJob(D, n)
{   J = {1};
    for(i = 2 to n)
        if('Union(J, i)中的作业都能按期完成')
            J = Union(J, i);
    return J;
}
```

GreedyJob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~~ | |

**5. 所有作业能否按期完成的判断**

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，肯定能按期内完成。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，寻找作业 $i$ 的待插入位置，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 | |

## 6. 带限期作业安排问题的贪心算法

用数组D[0 .. n]存放作业期限值，按效益降序排列，其中作业0是虚构的，且D[0] = 0，便于将作业插入位置1；用数组J[0 .. n]存放成为可行解的作业；用m表示放入J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，即当前时刻。

```
int GreedyJob(int D[], int n, int J[])
{   J[0] = 0, J[1] = 1; // 把虚构作业0和实际作业1放入J中
    int m = 1; // J中的作业数m=1
    for(int i = 2; i <= n; ++i)
    {   // 从J的最后位置m开始寻找作业i的待插入位置r
        int r;
        for(r = m; D[J[r]] > D[i] && D[J[r]] > r; --r) {}
        if(D[J[r]] <= D[i] && D[i] > r)
        {   for(int k = m; k > r; --k) J[k + 1] = J[k]; // 后移r后的作业
            J[r + 1] = i, ++m; // 作业i放入r+1位置, J中的作业数加1
        }
    }
    return m; // 返回插入作业个数
}
```

## 7. 复杂性算法分析

对于GreedyJob，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 $O(sn)$。由于 $s \leq n$，因此该算法的总时间是 $O(n^2)$。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，所以该算法的最坏情况时间为 $\Theta(n^2)$。

## 8. 测试程序*

```
#include "algorithm.h"
int main()
{   int n = 7; // 实际作业数
    int D[n + 1] = {0, 4, 2, 4, 3, 4, 8, 3}; // 必须有D[0]=0
    int J[n + 1];
    int m = GreedyJob(D, n, J);
    cout << '\t' << "作业" << '\t' << "期限" << endl;
    for(int i = 1; i <= m; ++i)
        cout << i << '\t' << J[i] << '\t' << D[J[i]] << endl;
}
```

```
        作业    期限
1       2       2
2       4       3
3       1       4
4       3       4
5       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~~ | |

## 6.2.3 多机调度问题

### 1. 问题说明

有 $n$ 项独立作业 $\{1,2,\ldots,n\}$，由 $m$ 台相同的机器处理。作业 $i$ 需要的处理时间为 $t_i$。约定每项作业都可以由任何一台机器处理，但未完成前不能中断，任何作业都不能拆分更小的子作业。

多机调度问题要求给出一种调度方案，使 $n$ 个作业在尽可能短的时间内由 $m$ 台机器处理完。该问题到目前为止还没有一个有效的解法，利用贪心方法可以设计出较好的近似解。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6-1所示。

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$M_3$。

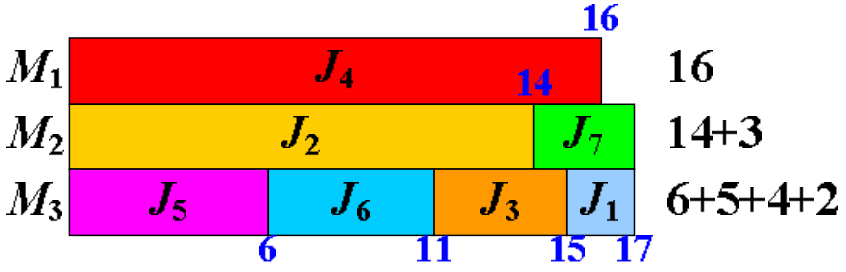


图6-1　一个多机调度的例子

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. 作业类和机器类

作业包含作业号ID和所需处理时间time两个属性，机器包含机器号ID和开始空闲时刻avail两个属性。

```
struct Job
{   int ID; // 作业号
    double time; // 所需处理时间
};
```

```
struct Machine
{   int ID; // 机器号
    double avail; // 开始空闲时刻
};
```

```
bool operator<(const Job &x, const Job &y)
{   return x.time < y.time; // 比较所需处理时间
}
```

```
bool operator<(const Machine &x, const Machine &y)
{   return x.avail < y.avail; // 比较开始空闲时刻
}
```

## 4. 算法描述

```
#include "algorithm.h"
void greedyLPT(Job J[], int n, int m) // n是作业数, m是机器数, n>m
{   minheap<Machine> H; // 最小堆, 存放可用机器
    for(int i = 1; i <= m; ++i) H.push({i, 0}); // 初始化机器和最小堆
    sort(J, J + n); // 作业按照所需处理时间升序排列
    for(int i = n - 1; i >= 0; --i) // 从处理时间最长的作业开始
    {   Machine M = H.top(); H.pop(); // 找到最早空闲的机器
        cout << "作业" << J[i].ID << "安排到机器" << M.ID << endl;
        M.avail += J[i].time, H.push(M); // 修改空闲时刻, 入堆
    }
}
```

## 5. 测试程序*

```
int main()
{   Job J[] = {{1, 2}, {2, 14}, {3, 4}, {4, 16}, {5, 6}, {6, 5}, {7, 3}};
    greedyLPT(J, 7, 3);
}
```

```
作业4安排到机器1
作业2安排到机器2
作业5安排到机器3
作业6安排到机器3
作业3安排到机器3
作业7安排到机器2
作业1安排到机器3
```

# 6.3 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。

## 6.3.1 Prim算法

### 1. 基本思想

如图6-2所示。

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 $T+\{e\}$ 仍是一棵子树。

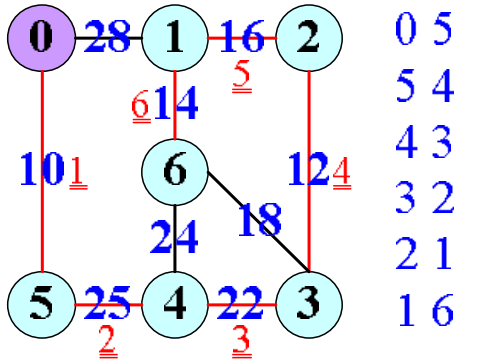



图6-2 Prim算法的基本思想

## 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。

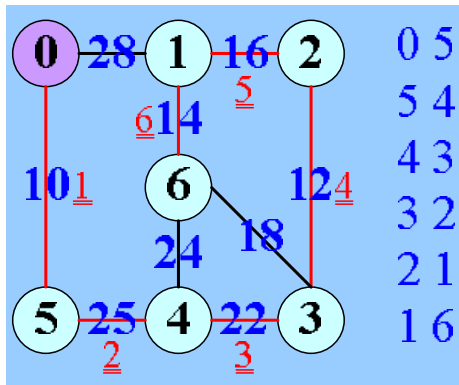
数组S用于标志顶点是否已经选进当前子树中。如果v已经在当前子树中，则S[v]=1，否则，S[w]=0。初始值为0。

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6-2所示的图，从顶点0开始执行的执行过程演示如图6-3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6-3　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)
{   int n = G.Rows();// G是加权图，顶点数为n
    prev.assign(n, v); // prev保存各顶点的父亲，初始为根v
    vector<bool> S(n, 0); // S标志顶点是否已选，初始为0
    S[v] = 1; //  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(S[w] == 0 && G(prev[w], w) < min)
                min = G(prev[w], w), v = w;
        if(isinf(min)) return false; // 不连通
        S[v] = 1; // 选取v
        for(int w = 0; w < n; ++w) // 更新未选顶点的父亲
            if(S[w] == 0 && 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 $O(n^2)$。

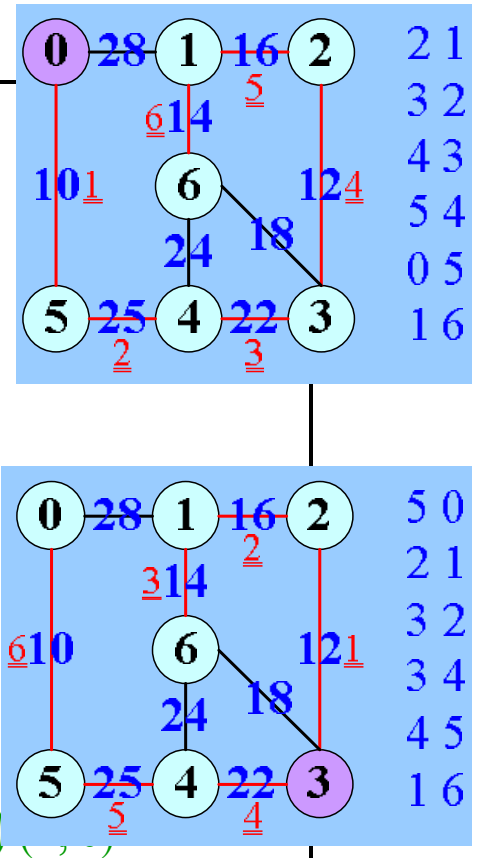
## 5. 辅助程序及测试*

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();
    for(int v = 0; v < n; ++v)
        if(prev[v] != v) // 根满足prev[v] == v
            cout << "(" << prev[v] << ", " << v << ") ";
}
```

② 测试程序。使用图6-2所示的图。

```
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); // (2, 1) (3, 2) (4, 3) (5, 4) (0, 5)
    else cout << "不是连通图";
}
```



# 6.3.2 合并查找结构

Kruskal算法的C++描述需要使用合并查找结构，这里首先给出合并查找结构的C++程序（保存在文件UnionFind.hpp中），然后再用C++描述Kruskal算法。

## 1. 数据成员与初始化

```
// UnionFind.hpp
#pragma once
#include <algorithm> // fill_n, etc.
using namespace std;
```

```
class UnionFind
{   int *parent; // 双亲数组，负数代表根和深度
```

```
public:
    UnionFind(int n = 10) // 每个集合(类别，树)一个元素
    {   parent = new int[n];
        fill_n(parent, n, -1); // 双亲数组初始值为-1
    }
```

```
    ~UnionFind()
    {   delete []parent;
    }
```

## 2. 查找与合并

```
    int Find(int e) const // e所在树的根
    {   if(parent[e] < 0) return e; // e是根
        else return Find(parent[e]); // e不是根
    }
    void Union(int i, int j) //  合并i和j所在的树
    {   i = Find(i), j = Find(j); // i和j所在树的根
        if(i == j) return; //  无需合并
        if(-parent[i] < -parent[j]) // i深度较小, i成为j的子树
            parent[i] = j;
        else if(-parent[j] < -parent[i]) // j深度较小, j成为i的子树
            parent[j] = i;
        else // j成为i的子树, i深度增加
            parent[j] = i, --parent[i];
    }
};
```

**3. 复杂性分析**

容易看出，UnionFind初始化耗时 $O(n)$。

对于查找/合并操作，设 $d$ 为合并算法产生的有 $n$ 个顶点的树的深度。容易看出，查找/合并操作耗时 $O(d)$，下列引理说明 $d \le \log n + 1$，所以查找/合并操作耗时 $O(\log n)$。

【引理】若 $T$ 是一棵由合并算法产生的树，且 $T$ 的顶点数为 $n$，则 $T$ 的深度 $d \le \log n + 1$。

【证明】当 $n = 1$ 时引理显然成立。假设当 $n < m$ 时引理成立。

当 $n = m$ 时，设 $T$ 是一棵由合并算法产生的树，且 $T$ 的顶点数为 $m$。考虑最后一次合并操作。若合并前两棵子树的深度不相同，则 $T$ 的深度与深度较大的子树相同，引理成立。否则，$T$ 的深度 $d$ 比子树的深度 $h$ 多1。对于顶点较少的子树，其顶点数 $v \le m/2$，所以 $d = h + 1 \le (\log v + 1) + 1 = \log(2v) + 1 \le \log m + 1$。引理成立。

# 6.3.3 Kruskal算法

## 1. Kruskal算法的基本思想

如图6-4所示。

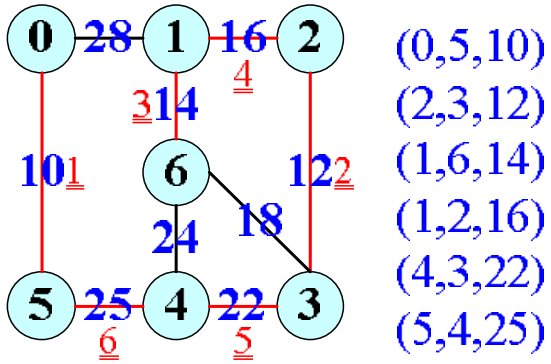



图6-4 Kruskal算法的基本思想

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 $L+\{e\}$ 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
    UnionFind U(n); // 合并/查找结构, 初始时表示n个不同分枝
    sort(begin(Ed), end(Ed)); // 将图的边集合按照边成本升序排列
    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, 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 $e<n-1$ 时无需使用该算法，所以该算法总耗时 $O(e\log e)$。

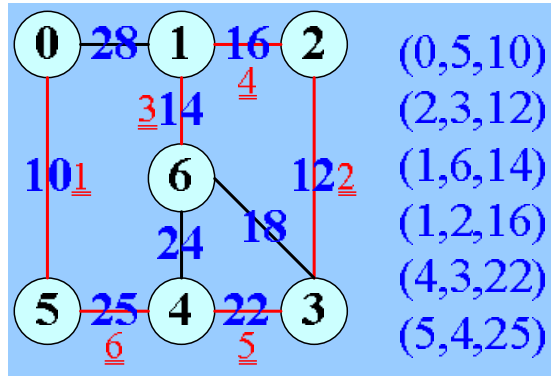
## 5. 测试程序*

使用如图6-4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;
}
```

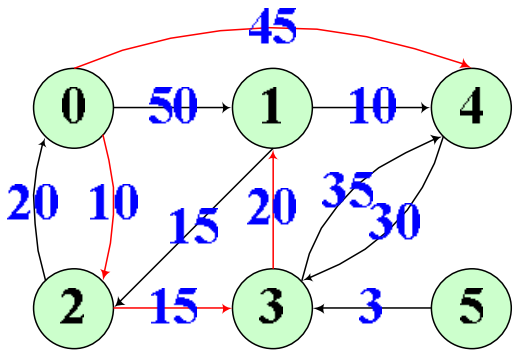
```
(0, 5, 10) (2, 3, 12) (1, 6, 14) (1, 2, 16) (3, 4, 22) (4, 5, 25)
```



# 6.4 单源最短路径问题

## 6.4.1 问题介绍

问题：已知一个有向加权图 $G=(V,E,w)$，求由$G$中某个指定的顶点 $v_0$ 到其他各个顶点的最短路径。例如，对于图6-5左侧所示的有向加权图，从顶点0到其他各顶点的最短路径如图6-5右侧所示。




| 路径 | 长度 |
|---|---|
| 0-2 | 10 |
| 0-2-3 | 25 |
| 0-2-3-1 | 45 |
| 0-4 | 45 |

图6-5　从顶点0到其他各顶点的最短路径

## 6.4.2 贪心准则和算法思路

采用逐条构造最短路径的办法，用“已生成的所有路径长度之和最小”为贪心准则，为此，每一条单独的路径都必须具有最小长度。生成从 $v_0$ 到其他顶点的最短路径的贪心算法按照路径长度递增的顺序来生成这些路径。为了按照递增顺序生成这些路径，需要确定与 $v_0$ 生成最短路径的下一个顶点，以及 $v_0$ 到这一顶点的最短路径。集合 $S$ 包含所有已经构造好最短路径的顶点（包含 $v_0$，简称为已选顶点）。若 $V-S\neq\varnothing$，则从 $v_0$ 到 $V-S$ 中各顶点（简称为未选顶点）的最短路径中应该有一条最短的，设该路径是 $v_0$ 到 $v_s$ 的最短路径 $v_0v_1\cdots v_{s-1}v_s$。显然，$v_0v_1\cdots v_{s-1}$ 应是 $v_0$ 到 $v_{s-1}$ 最短路径。根据 $S$ 的定义和贪心准则，$v_{s-1}$ 应属于 $S$。同理，路径 $v_0v_1\cdots v_{s-1}v_s$ 上的其他顶点 $v_i$ 也都在 $S$ 中。所以，新的最短路径一定是某个已有的最短路径向前延伸一步。

用 $\mathrm{dist}(v)$ 记从 $v_0$ 到 $S$ 中顶点 $v$ 的最短路径的长度，图中的顶点 $v$ 依其是否已选分别记为 $S(u)=1$ 或 $S(u)=0$。从 $v_0$ 出发，新的最短路径的长度应该是

$$\min_{v\in S}\left\{\min_{w\notin S}\{\mathrm{dist}(v)+(v,w)\}\right\}$$

选取满足该条件的顶点 $v$（通过 $S(v)=1$ 实现），新的最短路径就是从 $v_0$ 出发到 $v$ 的最短路径，同时，更新未选顶点的 dist 值以实现 $\min_{w\notin S}\{\mathrm{dist}(v)+(v,w)\}$，方法是对每个未选顶点 $w$，令 $\mathrm{dist}(w)=\min\{\mathrm{dist}(w),\mathrm{dist}(v)+(v,w)\}$。

上述算法思想是Dijkstra提出的。

# 6.4.3 Dijkstra最短路径算法

$$\mathrm{dist}(w)=\min\{\mathrm{dist}(w),\mathrm{dist}(v)+(v,w)\}$$

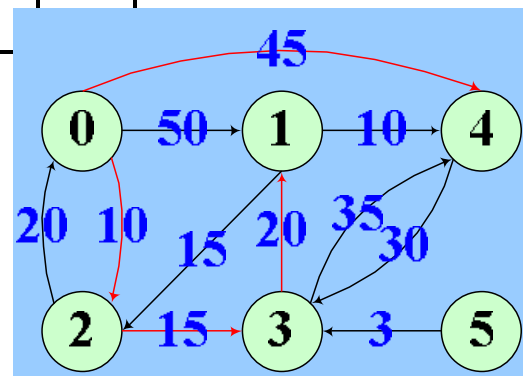
## 1. 算法说明

该算法计算出从源点到其他各顶点最短路径的长度（用数组dist保存，初始为源点到其他顶点的边长），并记录最短路径中各顶点的前一顶点（用数组prev保存，初始为源点，可以用于生成各最短路径）。方法是在未选顶点中寻找使dist(v)最小的顶点v，然后将v选中，同时更新未选顶点的dist值和prev值。

例如，对于图6-5所示的有向加权图，Dijkstra算法的执行过程如图6-6所示。

| $S$ | Dist : Prev | | | | | $(v,w)$ | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | 5 |
| 0 | 50:0 | 10:0 | ∞:0 | 45:0 | ∞:0 | ∞ | √ | 15 | ∞ | ∞ |
| 0, 2 | 50:0 | √ | 25:2 | 45:0 | ∞:0 | 20 | √ | √ | 35 | ∞ |
| 0, 2, 3 | 45:3 | √ | √ | 45:0 | ∞:0 | √ | √ | √ | 10 | ∞ |
| 0, 2, 3, 1 | √ | √ | √ | 45:0 | ∞:0 | √ | √ | √ | √ | ∞ |
| 0, 2, 3, 1, 4 | √ | √ | √ | √ | ∞:0 | | | | | |

图6-6 Dijkstra算法的执行过程演示



## 2. 算法描述

```
#include "algorithm.h"
auto Dijkstra(const Matrix<double> &G, int v) // G是邻接矩阵, v是指定顶点
{   int n = G.Rows(); // 顶点数
    vector<int> prev(n, v); // 路径中各顶点的前趋, 初始为v
    vector<double> dist(n); // 各路径的耗费, 初始为边(v, w)的耗费
    for(int w = 0; w < n; ++w) dist[w] = G[v][w];
    vector<bool> S(n, 0); // 标志顶点是否已选, 初始为0
    S[v] = 1; // 选取v
```

$$\min_{v \in S}\left\{\min_{w \notin S}\{\mathrm{dist}(v)+(v,w)\}\right\}$$

```
    for(int i = 0; i < n - 1; ++i)
    {   // 从未选顶点中寻找使dist[v]最小的v
        double min = inf; // min初始为无穷大
        for(int w = 0; w < n; ++w)
            if(S[w] == 0 && dist[w] < min) min = dist[w], v = w;
        if(isinf(min)) break; // 无需再选
        S[v] = 1; // 选取v
        for(int w = 0; w < n; ++w) // 更新未选顶点的dist和prev
        {   double t = dist[v] + G[v][w];
            if(S[w] == 0 && t < dist[w]) dist[w] = t, prev[w] = v;
        }
    }
    return prev;
}
```

$$\mathrm{dist}(w)=\min\{\mathrm{dist}(w),\mathrm{dist}(v)+(v,w)\}$$

**3. 算法复杂性**

初始化耗时 $O(n)$。在for循环的循环体中，从未选顶点中寻找使Dist(v)最小的v耗时 $O(n)$，更新未选顶点的dist值和prev值耗时 $O(n)$，执行次数不超过 $n-1$，因而整个循环共耗时 $O(n^2)$。所以，该算法的时间复杂度为 $O(n^2)$。

## 4. 构造最短路径

根据上述算法获得的prev数组计算从源点到顶点v的最短路径，用数组X依次记录该路径中的每个顶点。

```
auto Path(const vector<int> &prev, int v) // 从源点到顶点v的最短路径
{   vector<int> X; // 用数组X记录最短路径中的顶点
    while(prev[v] != v) // 逆向记录除源点外的其他顶点
        X.push_back(v), v = prev[v];
    X.push_back(v); // 记录源点v
    reverse(begin(X), end(X)); // 反转数组
    return X;
}
```

容易知道，该算法的时间复杂度为 $O(n)$ 。

## 5. 测试*

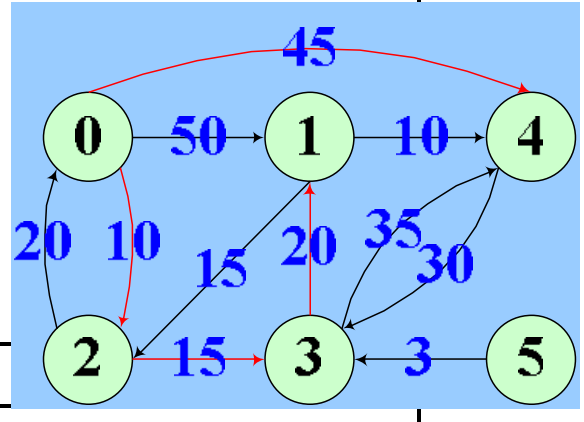
① 输出路径。输出所有最短路径。

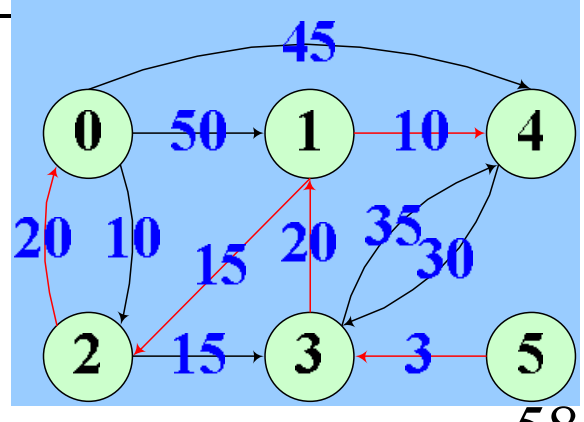
```
void Path(const vector<int> &prev) // 输出所有最短路径
{   int n = prev.size();
    for(int v = 0; v < n; ++v)
    {   auto X = Path(prev, v); // 从源点到顶点v的最短路径
        cout << to_string(begin(X), end(X), "-") << " ";
    }
    cout << endl;
}
```

② 测试程序。使用图6-5左侧所示的有向加权图。

```
int main()
{   Matrix<double> G = // 邻接矩阵
    {   {0, 50, 10, inf, 45, inf},
        {inf, 0, 15, inf, 10, inf},
        {20, inf, 0, 15, inf, inf},
        {inf, 20, inf, 0, 35, inf},
        {inf, inf, inf, 30, 0, inf},
        {inf, inf, inf, 3, inf, 0}
    };
    Path(Dijkstra(G, 0));
    Path(Dijkstra(G, 5));
}
```

```
0 0-2-3-1 0-2 0-2-3 0-4 0-5
5-3-1-2-0 5-3-1 5-3-1-2 5-3 5-3-1-4 5
```



# 6.5 练习题

1、请使用一种程序设计语言或伪代码描述Dijkstra最短路径算法。已知：$G$ 是有 $n$ 个顶点的有向加权图（用邻接矩阵表示），$v$ 是指定的源顶点。

2、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。

3、使用一种程序设计语言描述最小生成树的Prim算法和Kruskal算法。已知：$G$ 是具有 $n$ 个顶点的无向加权图，$E$ 是 $G$ 中边的集合，并且已经按照权值排序，其中矩阵类型Matrix和边类型Edge已经定义。

4、请使用一种程序设计语言描述使用贪心方法解决下列问题的算法。

装载问题，背包问题，活动安排问题，带限期作业安排问题，多机调度问题。

5、求最大团的近似算法。

（1）用贪心方法设计一种求最大团的近似算法（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。