第三次上机

1. Fractional Knapsack (分数背包)

value($US)	20	30	65	40	60
weight(Lbs)	10	20	30	40	50
value/weight	2	1.5	2.1	1	1.2

解决分数背包问题使用贪心法十分快捷。只需要不断选取单位质量价值最高的物品就能在满足质量约束的条件下，最大化背包中物品的价值。

使用快速排序的情况下，算法的时间复杂度为$\Theta(n \log n)$

核心代码

代码十分简单：

```
// 对物品根据价值比排序, items为排序后所有物品的数组, 背包容量为100
int volume = 100;
// begin greedy method
for (auto it = items.begin(); volume > 0 && it != items.end(); ++ it) {
    // 当背包容量不足以容纳全部质量的物品时，只取背包能够容纳的部分
    int w = volume >= it->weight? it->weight: volume;
    volume -= w;
    weight += w;
    value += (double)it->value / it->weight * w;
}
// 最终总价值为value, weight=100

```

测试结果

```
3 1 2 5
Total value: 163.000000, Total weight: 100
```

2. 01 Knapsack (01背包)

01背包问题可以使用动态规划方法解决。

每个物品只有放入背包与不放入背包两种可能。对于物品集合L, 背包容量w的情况，如果物品集合变为$L-l_i$，就产生了一个子问题，对物品集合$L-l_i$需要满足背包中物品的质量不超过$w-w_i$。如果子问题的最优解是(V_{l_i}, W_{l_i}), 原问题的最优解为：

$$V=\begin{cases} V_{l_i}+v_i, & if \quad w-W_{l_i}>=w_i \\ //背包容量为w,物品集合为L-l_i子问题的最优解, & if \quad w-W_{l_i}<w_i \end{cases} \tag{1}$$

根据这个最优子结构可以构造动态规划方法，$V(i,w)$表示物品集合为$l_1, l_2, \cdots, l_i$背包容量为w的子问题的最优解。

递推公式为：

$$V(i,w)=\begin{cases} max\{V(i-1,w-w_i)+v_i, V(i-1,w)\}, & if \quad i>0 \wedge w>=w_i \\ V(i-1,w), & if \quad i>0 \wedge w<w_i \\ 0, & if i=0 \vee w=0 \end{cases} \tag{2}$$

这个动态规划算法的时间复杂度为$\Theta(nw)$，物品数量n（$|L|$），背包容量w。

构造最优解

当$V(i,w)$取值为$V(i-1,w-W-i)+v_i$时，说明物品i放入背包。只要利用另一个二维数组U来记录物品i是否放入背包：

$$U(i,w)=\begin{cases} 1, & if \quad i>0 \wedge w>=w_i \wedge V(i-1,w-w_i)+v_i>V(i-1,w) \\ 0, & otherwise \end{cases} \tag{3}$$

核心代码

```
//初始化二维数组dp为全0
// dp记录总价值，dp2记录物品是否放入背包
/*
** V[i,w] = max(V(i-1, w-weights[i])+v[i], V(i-1, w)), if i>0 && w>=weights[i]
**          V(i-1, w), if i>0 && w<weights[i]
**          0, otherwise (i=0 || w=0)
*/
for (int i = 1; i <= n; ++ i) {
    for (int w = 1; w <= volume; ++ w) {
        if (w >= weights[i]) {
            if (dp[i-1][w-weights[i]]+v[i] > dp[i-1][w]) {
                // item i is taken
                dp[i][w] = dp[i-1][w-weights[i]]+v[i];
                dp2[i][w] = 1;
            }
            else {
                dp[i][w] = dp[i-1][w];
                dp2[i][w] = 0;
            }
        }
    }
}

// 构造最优解
int i = n, w = volume;
int value = 0, weight = 0;
vector<int> items;
while (i > 0) {
    if (dp2[i][w]) {
        // item i is taken
        value += v[i-1];
        weight += weights[i-1];
        w -= weights[i-1];
        items.push_back(i);
    }
    -- i;
}
```

测试结果

```
1 2 5
Total value: 110, Total weight: 80
```

3. Scheduling Problem (任务调度问题)

> A simple scheduling problem. We are given jobs *j1, j2... jn,* all with known running times *t1, t2... tn*, respectively. We have a single processor. What is the best way to schedule these jobs in order to minimize the average completion time. Assume that it is a nonpreemptive scheduling: once a job is started, it must run to completion. The following is an instance.
>
> (j1, j2, j3, j4) : (15，8，3，10)

单处理器非抢占式调度, 目标是最小化完成每个任务的平均等待时间（**the average completion time**）。

而最小化平均完成时间也就是最小化所有任务的完成时间之和$S = \sum_{i=1}^{n} c_i$，其中c_i表示从处理器开始执行第一个任务到完成任务i的完成时间。在S的求和中越排在前面的任务被重复计入的次数越多，所以需要使越先调度的任务花费时间越短（**Shortest Job First,SJF**调度）。

算法流程

对所有任务根据花费的时间从短到长排序得到任务调度序列。

平均等待时间为:

$$W = \sum_{i=1}^{n} \sum_{j=1}^{i} c_j / n \tag{4}$$

核心代码

```cpp
// 排序获取调度序列
sort(C.begin(), C.end(), [](const Job& x, const Job& y){
    return x.cost < y.cost;
});

// 计算平均等待时间
double avg = 0;
printf("Schedueled tasks ");
for (int i = 0; i < n; ++ i) {
    printf("%d ", C[i].id);
    for (int j = 0; j <= i; ++ j) {
        avg += C[j].cost;
    }
}
printf("with average completion time = %lf\n", avg/n);
```

测试结果

```
Scheduled tasks 3 2 4 1 with average completion time = 17.750000

--------------------------------
Process exited after 0.04857 seconds with return value 0
请按任意键继续. . .
```

4. Single Source Shortest Path

The adjacent matrix:

$$\begin{bmatrix} & A & B & C & D & E \\ A & \infty & -1 & 3 & \infty & \infty \\ B & \infty & \infty & 3 & 2 & 2 \\ C & \infty & \infty & \infty & \infty & \infty \\ D & \infty & 1 & 5 & \infty & \infty \\ E & \infty & \infty & \infty & -3 & \infty \end{bmatrix}$$

Dijistra Algorithm

Dijistra算法基于贪心的思想，每次都选择距离源节点最近的点作为下一轮更新的前缀点，更新距离$d[j] = min\{d[i] + w_{ij}, d[i]\}$。算法的时间复杂度为 $\Theta(n^2)$。

以示例输入为例，初始设置距离为$d = \{0, \infty, \infty, \infty, \infty\}$,
$visited = \{false, false, false, false, false\}$, $prev = \{-, -, -, -, -\}$。

Iteration	prefix	d(B)/Path	d(C)/Path	d(D)/Path	d(E)/Path	prev	visited
1	A	-1/A-B	3/A-C	∞/-	∞/-	{-,A,A,-,-}	{true,false,false,false,false}
2	B	-1/A-B	2/A-C	1/A-B-D	1/A-B-E	{-,A,A,B,B}	{true,true,false,false,false}
3	D	-1/A-B	2/A-C	1/A-B-D	1/A-B-E	{-,A,A,B,B}	{true,true,false,true,false}
4	E	-1/A-B	2/A-C	1/A-B-E-D	1/A-B-E	{-,A,A,E,B}	{true,true,false,true,true}
5	C	-1/A-B	2/A-C	1/A-B-E-D	1/A-B-E	{-,A,A,E,B}	{true,true,false,true,true}

$prev$数组用于构造最短路。以A到D的最短路为例，A-B-E-D=prev(B)-prev(E)-prev(D)-D。

核心代码

```
// dijistra, 邻接矩阵aj
for (int i = 0; visited[i] != true;) {
    visited[i] = true;
    for (int j = 0; j < n; ++ j) {
        // INF = 0x7fffffff
        if (aj[i][j] == INF) {
            continue;
        }
        // d[v] = min{d[u]+w[u][v], d[v]}
        if (d[j] > d[i] + aj[i][j]) {
            d[j] = d[i] + aj[i][j];
            // prev[j] = i, 用于构造最短路径
        }
    }
// i = argmin{d[i]}
// ...
}
```

```
18 // 用prev构造最短路径
```

测试结果

```
distance from A to A: 0
A
distance from A to B: -1
A->B
distance from A to C: 2
A->B->C
distance from A to D: -2
A->B->E->D
distance from A to E: 1
A->B->E
```

5. All Pairs Shortest Path

Floyd Algorithm

所有节点对的最短路径计算得到一个距离矩阵d和一个用于构造最短路的前缀矩阵$prev$，矩阵的每一行是单元最短路的情况。初始化d的对角线元素为0，其余为∞。

Floyd算法对每一个节点，每一轮更新以一个节点k作为前缀节点，更新$d[i][j] = min\{d[i][k] + w_{kj}, d[i][j]\}$, 同时维护前缀数组$prev$。如果$d[i][j] = d[i][k] + w_{kj}$，设置$prev[i][j] = k$。

节点i和j之间的最短路径上，目标节点的前一个节点为$prev[i][j]$。利用这个前缀数组即可逆序构造出最短路。

核心代码

计算最短路

```
1  // floyd
2  // each row of d are singlesource shortest pathes
3  vector<vector<int>> d(n, vector<int>(n, INF));
4  vector<vector<int>> prev(n, vector<int>(n, INF));
5  for (int i = 0; i < n; ++ i) {
6      d[i][i] = 0;
7  }
8
9  for (int k = 0; k < n; ++ k) {
10     for (int i = 0; i < n; ++ i) {
11         for (int j = 0; j < n; ++ j) {
12             // kj节点之间不通或ik目前不可达
13             if (aj[k][j] == INF || d[i][k] == INF || i == j) {
14                 continue;
15             }
16             if (d[i][k]+aj[k][j] < d[i][j]) {
17                 d[i][j] = d[i][k]+aj[k][j];
18                 prev[i][j] = k;
19             }
20         }
21     }
22 }
```

构造最短路

```
// output
for (int i = 0; i < n; ++ i) {
    for (int j = 0; j < n; ++ j) {
        if (i == j) {
            continue;
        }
        if (d[i][j] == INF) {
            printf("No path between %s and %s\n", nodes[i].c_str(), nodes[j].c_str());
            continue;
        }
        // 输出节点i和j之间的最短距离
        vector<int> path;
        int k = prev[i][j];
        while (k != INF) {
            path.push_back(k);
            k = prev[i][k];
        }
        // 逆序输出最短路
    }
}
```

测试结果

```
Distance between A and B: -1
A->B
Distance between A and C: 2
A->B->C
Distance between A and D: -2
A->B->E->D
Distance between A and E: 1
A->B->E
No path between B and A
Distance between B and C: 3
B->C
Distance between B and D: -1
B->E->D
Distance between B and E: 2
B->E
No path between C and A
No path between C and B
No path between C and D
No path between C and E
No path between D and A
Distance between D and B: 1
D->B
Distance between D and C: 5
D->C
No path between D and E
No path between E and A
No path between E and B
No path between E and C
Distance between E and D: -3
E->D
```