Lab 2 实验报告

PB20000296 郑滕飞

动态规划：

1、基本原理

动态规划算法通过储存重复子问题的解来达到减小时间复杂度的效果，实质是用空间换时间的策略。于是，能用动态规划求解的问题需要包含两个要素：最优子结构与子问题覆盖。前者意味着达到全局最优时存在某种局部最优，而后者意味着通过局部最优可以导出全局最优(也就是某种递推关系)。

对于复杂度，动态规划问题若直接用递归求解，往往是问题规模的指数层级，而通过保存子问题的解，能达到多项式级别的复杂度。

解决动态规划问题的一般步骤：刻画最优解的结构特征、递归定义最优解值、自底向上计算最优解值、通过计算信息构造最优解。

2、例：最长公共子序列

[相关代码与结果见 largest_subsequence 文件夹]

子序列定义：从串中左到右取出的一列未必连续的元素(也即类似数列中的子列)。

问题：输入两序列 X、Y(实际操作中以字符串表示)，要求找到它们的最长公共子序列 Z 与它的长度。

a. 最优子结构特征

考虑 X、Y 最后一位，若它们相同，则找到的最优公共子序列一定相当于各自去掉最后一位后的最优公共子序列加上这位。

否则，最优公共子序列相当于分别去掉最后一位后得到的最优公共子序列中取长的(这是由于最优公共子序列的最后一位一定至多与 X、Y 最后一位中的一个相同)。

b. 递归定义最优解

记 e(a,b)代表 X 前 a 位与 Y 前 b 位的最长公共子序列长度，当 a 或 b 为 0 时初始为 0，满足递推关系：

$$e(a,b)=\begin{cases} e(i-1,j-1)+1 & x_a=y_b \\ \max\{e(a-1,b),e(a,b-1)\} & x_a\neq y_b \end{cases}$$

c. 自底向上计算最优解

容易发现，按照上方的递推，只需要按先行后列的顺序逐个计算 e(a,b)的每个元素就可以了，由此建立矩阵后可算出最优解。

d. 通过计算信息构造最优解

为了获取最长公共子序列，需要注意递推的过程。每当发生 X[a]=Y[b]时，需要将这一位放入子序列中，而如果是发生下面的情况，代表与其中较大的那个共用子序列。由此，只需要记录每次递推的选择，再反向寻找，即可找到最长公共子序列。

3、算法设计

算法的核心部分如下：

```
for (int i = 0; i < m; i++) for (int j = 0; j < n; j++) {
    if (a[i] == b[j]) {
        c[i+1][j+1] = c[i][j] + 1;
```

```
            t[i][j] = 'd';
        }
        else if (c[i][j+1] > c[i+1][j]) {
            c[i+1][j+1] = c[i][j+1];
            t[i][j] = 'u';
        }
        else {
            c[i+1][j+1] = c[i+1][j];
            t[i][j] = 'l';
        }
    }
```

其中，c 记录最优结果，t 记录每次的选择。在知道每次选择后，可以打印子序列：

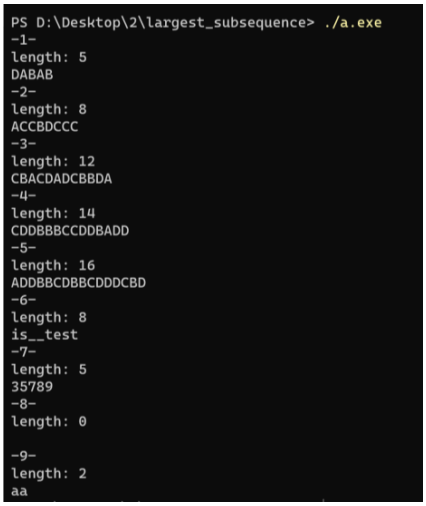
```
    string d = "";
    while (m >= 0 && n >= 0) {
        if (t[m][n] == 'd') {
            d = a[m] + d;
            m--;
            n--;
        }
        else if (t[m][n] == 'u') m--;
        else n--;
    }
```

这两部分结合，就基本得到了完整算法。由过程循环可以看出，时空复杂度均为两字符串长度乘积。由于需要具体构造最优解，这个空间复杂度不能再优化。

4、结果展示

in.txt 中保存着输入数据，每组输入数据三行，第一行编号，下面两行表示字符串 X 与 Y。

直接编译、运行文件即会在控制台打印输出，每组输出三行，分别是编号、最长公共子序列长度、最长公共子序列，输出如下：



对比 in.txt 可以看到，对一般例子与特殊的边界情况都能达到正确输出。

多重背包问题：

[相关代码与结果见 multi_bag 文件夹]

在刚才的简单例子后，这部分里，我们求解多重背包问题。

1、0-1 背包问题

首先，先解决问题的基础形式：0-1 背包。具体来说，有一个容量为 n 的背包，与若干件物品(假设共 t 件)，每件的重量 w[i]，价值 c[i]，要求找到背包内放入的总价值尽量高的方案，且总重量不能超过 n，物品不能拆分。

a. 最优子结构特征

假设最优的背包里有一件重量为 w、价值为 c 的物品，则背包里剩下的物品一定是容量 n-w 的背包在去除这件物品后的最佳方案。

b. 递归定义最优解

记 f[i][a]为最大重量为 a 的背包在放置物品中的前 i 件(i 不超过 t)时的最优方案，则有递推：

$$f[i][a] = \max\{f[i-1][a] \ , \ f[i-1][a - w[i]] + c[i]\}$$

左右分别是不选取第 i 件物品与选取了第 i 件物品的情况。

对边界情况，i 为 0 时 f[i][a]均为 0，而当 a 小于 w[i]时，只会出现第一种选择方式。

c. 自底向上计算最优解

按照先 a 后 i(即先行后列)的顺序逐步更新 f[i][a]，最终的 f[t][n]即为最优解。

d. 递归构造最优解

值得注意的是，如果不要求最优解的具体内容，上方的递归过程只需要数组 f[a]就可以做到，达到 O(n)的空间复杂度。不过，需要最优解时，必须用额外的数组记录每次的选择，这就导致空间复杂度必须在 O(tn)。

记录选择后，往回寻找就能得到最优解。

2、0-1 背包代码实现

核心的计算过程为：

```
for (int i = 1; i <= t; i++)
    for (int j = 1; j <= n; j++) {
        int t = -1;
        if (j >= w[i-1]) t = f[i-1][j-w[i-1]] + c[i-1];
        if (f[i-1][j] >= t) {
            f[i][j] = f[i-1][j];
            choice[i][j] = 0;
        }
        else {
            f[i][j] = t;
            choice[i][j] = 1;
        }
    }
```

而输出最优解的过程为：

```
    for (int i = t; i > 0; i--)
        if (choice[i][a]) {
            cout << i << endl;
            a -= w[i-1];
        }
```

组合得到整体代码。

3、0-1 背包结果展示

以 01 开头的 txt 中保存着输入数据，每组输入数据第一行表示背包大小与物品件数，第二行开始对应每件物品的重量与价值。

编译 01_bag.cpp 并在控制台运行，第二个输入参数为文件名，结果示例如下：

```
PS D:\Desktop\2\multi_bag> ./a.exe 01_3.txt
best value: 202
choice:
10
9
8
5
2
```

输出的最优解代表选取物品的序号(从 0 开始)，输入为：

```
100 10
12 25
23 49
14 26
89 181
23 47
45 89
12 25
34 70
13 26
7 10
```

标红的是选中的物品，可以验证满足要求。更多测试样例见文件夹中。

4、多重背包问题

问题描述：有一个容量为 n 的背包，与若干种物品(假设共 t 种)，每种的重量 w[i]，价值 c[i]，个数 m[i]，要求找到背包内放入的总价值尽量高的方案，且总重量不能超过 n，物品不能拆分。

一个基本的思路是，将问题直接当作 0-1 背包，逐个物品计算求解。但是，这样做会导致时空复杂度很高。为了降低复杂度，可以采取二进制拆分的方案。

举例来说，假设某个物品有 11 件，由于 11 的二进制为 1101，共 4 位，小于 11 的任何数一定可以通过二进制数 1、10、100 与 1101 减它们剩下的 101 累加出来。这样，假设一个物品有 n 件，事实上可以通过 log n 的量级表示。

将这 11 件物品合为 4 件物品，对应的重量与价格分别是 1、2、4、5 倍，即可以当作 0-1 背包进行求解，求解后再合并得到结果即可。

5、代码实现

拆分过程的代码实现为：

```
for (int i = 0; i < v; i++) {
    int temp = 1;
    while (m[i] > temp) {
        m[i] -= temp;
        seq.push_back(i);
        num.push_back(temp);
        temp = temp << 1;
    }
    seq.push_back(i);
    num.push_back(m[i]);
}
```

seq 记录这个拆分后的“大物品”对应物品的序号，num 则记录件数，这样，需要知道重量与价值时只需：

```
int w0 = w[seq[i-1]] * num[i-1];
int c0 = c[seq[i-1]] * num[i-1];
```

最后，打印结果时合并即可。

6、结果展示

直接数字命名的 txt 中保存着输入数据，每组输入数据第一行表示背包大小与物品件数，第二行开始对应每件物品的重量、价值与个数。

编译 multi_bag.cpp 并在控制台运行，第二个输入参数为文件名，对于示例输入：

```
100 3
5 4 100
2 3 39
1 1 20
```

输出如下：

```
PS D:\Desktop\2\multi_bag> ./a.exe 3.txt
best value: 138
choice:
1
39
17
```

这里 choice 后每行代表对应种类物品的选取个数，可验证符合要求。更多测试样例见文件夹中。

分析过程可知，时空复杂度为$O(n\log\prod_{i=1}^{t}m_i)$，其中$m_i$代表每件物品的个数。比起直接当作 0-1 背包的$O(n\sum_{i=1}^{t}m_i)$，这个结果有不小的优化。

总结：

动态规划思想意味着只要问题具有最优子结构和子问题覆盖性质，就能通过保存子问题的解来降低整体时间复杂度。这样的保存可以是显式的(如实验报告中涉及的三个问题)，也可以是隐式的(如 Dijkstra 算法进行最短路径求解中每一步保存的 d)。一般来说，如果需要知道最优解的构造，会导致中途保存的选择信息更多，于是会有更高的空间复杂度，否则，可以以都较低的时空复杂度解决问题。

附录-文件列表

report.pdf 实验报告
largest_subsequence 最长公共子序列
-- largest_subsequence.cpp 代码
-- in.txt 测试样例
multi_bag 背包问题
-- 01_1.txt
-- 01_2.txt
-- 01_3.txt 零一背包问题样例
-- 01_bag.cpp 零一背包问题代码
-- 1.txt
-- 2.txt
-- 3.txt 多重背包问题样例
-- multi_bag.cpp 多重背包问题代码

Lab 3 实验报告

PB20000296 郑滕飞

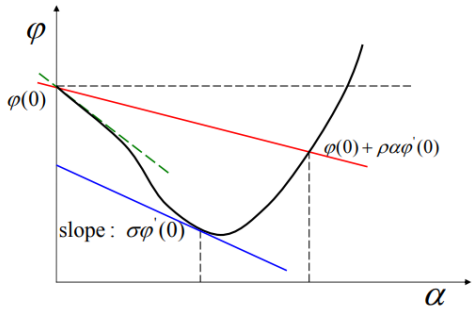
实验目的：通过各种迭代法结合非精确一维搜索对全空间函数最小值进行求解。

非精确一维搜索：

1、基本原理

非精确一维搜索希望在某个方向上找到“极小点附近的某一段”中的点。结合不同迭代法对方向的选取，它可以在优化效率的情况下保证收敛于极小点。

如来自讲义的下图，Wolfe-Powell 准则的斜率条件(或更强的斜率绝对值条件，本实验中不采用)保证了比起点更接近最优，而函数值条件则是同时限制了不会选到过远处。



根据准则的示例也可看出，能用此准则进行一维搜索有两个基本要求：取到的方向是下降方向，且局部极小值存在。否则，可能根本不存在满足对应要求的点。

2、算法准备工作

为了实现 wp 算法与之后的迭代，先定义了一些向量基本操作，如点乘、线性组合、无穷范数，例如线性组合实现如下：

```
vector<double> bi_com(double a, vector<double> x, double b, vector<double>
y) {
    for (int i = 0; i < x.size(); i++) x[i] = a * x[i] + b * y[i];
    return x;
}
```

之后的算法中，线性组合函数 bi_com 在出现两个向量参数时为加法，数与向量参数时为数乘，向量、数、向量时为 x+by。

除此之外，还实现了点乘函数 dot 与无穷范数函数 inf_norm，方便之后的算法里直接使用。

关于 WP 一维搜索中的参数，由于不会对正确性产生太大影响，直接采取了宏定义的方式：

```
#define RHO 0.3
#define SIGMA 0.5
#define MAX_ITER 500
#define TOL 1e-5
```

这里的 MAX_ITER 表示过程中某些部分的最大迭代次数，而 TOL 表示算法终止的梯度无穷范数上限。

3、搜索过程

WP 搜索的过程实质上是一个二分的过程，每次调整都是将上限或下限调整为旧的分点后取新的分点。不过，调整的过程比起取中点要复杂很多：本质是用二次函数拟合后取二次函数的最小点。

主要迭代过程前，还有两个重要的判断步骤：

```
const double phip0 = dot(d, df(x)), phi0 = f(x);
if (phip0 >= 0) {
    cout << "warning: direction choosed not a decrement" << endl;
    return 0;
}
double begin = 0, end = 2;
int count = 0;
double phip1 = phip0, phi1 = phi0;
while (dot(d, df(bi_com(x, end, d))) < 0) {
    end *= 2;
    count++;
    if (count == 10) {
        cout << "warning: perhaps d too small" << endl;
        return end;
    }
}
```

若方向根本不是下降方向，直接退出报 warning。此外，为了决定初始的搜索区域，在 d 方向不断跳跃，直到找到正在增加的位置，若不存在，则也退出，报告 d 可能过小，并返回最后跳跃到的位置。

如果正确找到了初始区间，就可以开始搜索过程：

```
double now = end / 2;
count = 0;
while (count < MAX_ITER) {
    count++;
    vector<double> x_now = bi_com(x, now, d);
    double phi = f(x_now);
    if (phi > phi0 + RHO * now * phip0) {
        end = now;
        now = begin + (begin - now) * (begin - now) *
            phip1 / (phi1 - phi + (now - begin) * phip1) / 2;
        continue;
    }
    double phip = dot(d, df(x_now));
    if (phip < SIGMA * phip0) {
        double temp = now;
        now += (now - begin) * phip / (phip1 - phip);
        begin = temp;
        phi1 = phi;
```

```
            phip1 = phip;
            continue;
        }
        return now;
    }
    cout << "warning: too many searches" << endl;
    return now;
```

具体的迭代方式来自讲义，如果迭代次数过多会退出并报告过多搜索次数。否则，找到结果后退出并返回结果，也即这个方向的最佳步长因子。

迭代法：

实现非精确一维搜索后，需要配合迭代法，达到完整的求解无约束最优化过程。

1、梯度、共轭梯度

直接梯度方法最为简单，直接选取梯度方向下降：

```
    while (count < MAX_ITER) {
        count++;
        vector<double> g = df(x0);
        if (inf_norm(g) < TOL) return count;
        double a = wp_search(df, f, x0, bi_com(-1, g));
        x0 = bi_com(x0, -a, g);
    }
    cout << "reached max iter" << endl;
    return count;
```

最终下降到的结果存储在 x0 中，返回值是迭代次数，若达到最大迭代次数会报告。

对共轭梯度法，每次选取适当的下降方向，保证与之前的方向共轭，从而加快搜索效率。不过，由于共轭的要求，每当达到维数的迭代次数时需要进行重启动，从负梯度方向重新开始，核心迭代如下：

```
    while (count < MAX_ITER) {
        count++;
        double beta = dot(g, g);
        g = df(x0);
        if (inf_norm(g) < TOL) return count;
        if (count % dim == 1) d = bi_com(-1, g);
        else {
            beta = dot(g, g) / beta;
            d = bi_com(-1, g, beta, d);
        }
        double a = wp_search(df, f, x0, d);
        x0 = bi_com(x0, a, d);
    }
```

这里的 beta 采取 Fletcher–Reeves 的算法，以新的梯度模长平方除以旧的梯度模长平方。

2、拟牛顿方法

拟牛顿迭代需要保存一个矩阵作为 Hesse 阵的近似，初始为单位阵，每次以 SR1 或 SR2 的方式修正更新。

算法主体过程如下：

```
vector<vector<double>> H(dim, vector<double>(dim));
for (int i = 0; i < dim; i++) H[i][i] = 1;
while (count < MAX_ITER) {
    count++;
    vector<double> g = df(x0);
    if (inf_norm(g) < TOL) return count;
    vector<double> d = count_d(H, g);
    double a = wp_search(df, f, x0, d);
    x0 = bi_com(x0, a, d);
    vector<double> s = bi_com(a, d);
    vector<double> y = bi_com(df(x0), -1, g);
    if (sr == 2) count_H2(s, y, H);
    else count_H1(s, y, H);
}
```

count_d 为计算-Hg 的过程，通过当前的 H 与梯度方向计算下降方向，而 count_H1 与 count_H2 则是秩 1、秩 2 修正计算更新的 H 的算法。sr 为拟牛顿法的参数，决定修正方式。计算 H 的代码如下：

```
void count_H2(vector<double> s, vector<double> y, vector<vector<double>>&
H) {
    const int dim = s.size();
    vector<double> Hy(dim);
    for (int i = 0; i < dim; i++)
        for (int j = 0; j < dim; j++)
            Hy[i] += H[i][j] * y[j];
    double a = 1 / dot(s, y), b = 1 / dot(Hy, y);
    for (int i = 0; i < dim; i++)
        for (int j = 0; j < dim; j++) {
            H[i][j] += s[i] * s[j] * a;
            H[i][j] -= Hy[i] * Hy[j] * b;
        }
}

void count_H1(vector<double> s, vector<double> y, vector<vector<double>>&
H) {
    const int dim = s.size();
    vector<double> Hy(dim);
    for (int i = 0; i < dim; i++)
        for (int j = 0; j < dim; j++)
            Hy[i] += H[i][j] * y[j];
```

```
    s = bi_com(s, -1, Hy);
    double a = 1 / dot(s, y);
    for (int i = 0; i < dim; i++)
        for (int j = 0; j < dim; j++) {
            H[i][j] += s[i] * s[j] * a;
        }
}
```

由此即得到两种拟牛顿法的迭代过程。

展示与对比：

1、测试函数

算法的测试共分为四个函数，分别是：

$$f_1(x) = \sum_{i=0}^{n-1} x_i^4$$

$$\mathrm{f_2(x) = (x_0 - 1)^2 + \sum_{i=1}^{n-1} (x_i - ix_{i-1})^2}$$

$$f_3(x) = |x_0 - 3| + \sum_{i=1}^{n-1} |x_{i-1} + 2x_i|$$

$$f_4(x) = \sum_{i=0}^{n-2} \left(100\left(x_{i+1} - x_i^2\right)^2 + (x_i - 1)^2\right)^2$$

前三个函数可以对任何维度构建，第四个函数要求至少二维。算法同时要求传入每个函数的梯度，此处对绝对值函数求导的结果为：在绝对值超过 1e-7 时为其符号，否则为 0，也即有容差的符号函数。

第一个函数为普通四次函数，最小值为 0，当且仅当各分量均为 0 时取到，有凸性，测试增长速度很快时的偏差。

第二个函数为二次函数，最小值为 0，当且仅当$x_i = i!$时取到，有凸性，测试基本的收敛情况。

第三个函数为绝对值函数，最小值为 0，当且仅当$x_i = \frac{3}{(-2)^i}$时取到，有凸性，测试只有一阶光滑性，无二阶光滑性时各算法的情况。

第四个函数为 Rosenbrock 函数，最小值为 0，当且仅当各分量均为 1 时取到，无凸性，测试对一般的无约束问题效果。

2、测试方案

测试分为三个部分：

第一部分，将四个函数代入运用 WP 非精确一维搜索的四个迭代法进行基础测试，初值为{1,-3,10,-4,-5,50}。

第二部分，仍然是上述四个函数与相同初值，不使用 WP 非精确一维搜索，而是直接将步长因子设为 0.2，对四种迭代法进行测试。

第三部分，对 f1 与 f4 进行更好、更坏初值的测试进行对比。(f2 作为二次函数，初值并不存在太大影响，而 f3 绝对值函数并不属于合适迭代法的二阶光滑情况。)

三个部分采取的都是六维的初值，可以表现一般情况的性能。

值得注意的是，这里实现的算法都是需要输入精确的梯度的，这也是现实中常遇到的情况，即通过表达式可以直接理论计算出精确的梯度函数。如果不能做到，就需要结合其他算法估计梯度，不在本实验的范围内。

3、测试结果

将 iteration 文件夹中的解决方案直接打开运行，即可在控制台看到结果。为了方便，将控制台输出复制到了 out.txt 当中。

每组每个测试的输出包括：一维搜索中的 warning 与迭代过程中达最大次数的报告、算法名与迭代次数、找到的最小 x、找到的 x 对应的函数值，例如：

```
warning: perhaps d too small
sr2 quasi newton iter:360
1 1 1 1 1 1
f(x)= 3.25676e-14
```

第一行为迭代过程中的警告(可能没有也可能有多行)，第二行表示这是对称秩 2 修正的拟牛顿法，迭代次数为 360 次，第三行为找到的最小 x，第四行为对应的函数值。下面叙述的测试结果基于 out.txt，细节详见 txt 文档。

第一组测试：

对 f1，良好的四次函数，直接梯度法与共轭梯度法都在较少次数直接找到了较精确的最小值，对称秩 2 修正的拟牛顿法在到达最大迭代次数后找到了相对梯度方法精确性差些的最小值，而对称秩 1 仅 24 次迭代后就失败，输出无效数字 nan(以下此类输出称为未输出结果)。

对 f2，二次函数，两种拟牛顿法都在接近维度次数的迭代后找到了到达机器精度的最小点，而共轭梯度法历经 54 次迭代，最后找到了较精确的最小点。直接梯度到达了最大迭代次数，仍然有明显的误差。

对 f3，不具有二阶光滑性时，只有直接梯度法输出了结果，但也误差极大。

对 f4，直接梯度法输出了误差极大的结果，而对称秩 2 修正的拟牛顿法在 360 次迭代找到了精确的最小值(即上方示例输出)，其他两方法没能输出结果。

第二组测试：

在采用特定步长而不进行搜索的情况下，各个算法的效果基本显著不如 WP 搜索时。

对 f1，只有对称秩 1 修正的拟牛顿法在高迭代次数后输出了较精确的结果，而其他三者都没有输出。

对 f2，对称秩 1 修正拟牛顿法仍能以接近维度次数的迭代找到机器精度附近最小点，但对称秩 2 就到达了最大迭代次数，且仍有明显误差，梯度法均不能输出。

对 f3，拟牛顿法均无输出，而梯度法与共轭梯度都到达了最大迭代次数，共轭梯度输出的最小值好于梯度法，但仍然都有明显误差。

对 f4，四个方法都没能输出。

第三组测试：

对 f1，更好的初值与更差的初值直接影响到了秩 2 修正拟牛顿法的精度，初值较差时达到最大迭代次数也仍有明显误差，而初值较好时较少迭代就可以找到更精确结果。初值对梯度方法收敛速度的影响存在，但并不明显，而结果精度都较好。即使是较好的初值，秩 1 修正拟牛顿法也没能输出结果。

对 f4，初值较好、较差时直接梯度法都在最大迭代次数后输出了有一定误差的结果，秩 1 修正拟牛顿法都没能输出结果。而共轭梯度法与秩 2 修正拟牛顿法则对初值有较显著的差别，初值较好时都能找到精确结果，初值较差时都没能输出。对比第一组测试时，初值好坏程度一般时秩 2 修正拟牛顿能输出，共轭梯度不能。此外，还做了一组额外测试，f4 哪怕在初值较好时，若不采用非精确一维搜索，直接给定步长，没有迭代法输出有意义的结果。

4、分析总结

结合上方的测试结果，可以做出以下分析:

直接梯度方法一般情况下收敛速度远不如其他方法，但优点在于稳定。对凸函数，只要步长足够小，一定能找到最小值的点。而对于非凸的函数，随机性就较大了，初值偏离程度的影响不如初值点选取大。不过，由于其易于实现，实际写代码中用到的时候不少，常采取较小的固定步长因子。

共轭梯度法是直接梯度法的优化。作为梯度方法，它的显著特点是不直接依赖二阶光滑性条件。对于二阶导函数无意义的绝对值函数(也是现实常见的 L1 范数优化)，拟牛顿法的原理注定了无法得出结果，而共轭梯度法优化搜索方向的思路则对一阶光滑也有意义。此外，重启动保证了某种意义上的稳定性，会修复此前造成的误差，这也让共轭梯度的使用范围更广。

值得一提的是，W-P 迭代虽然过程中只用到了一阶导函数，但在不具有二阶光滑性的时候，也可能因为限制条件的难以达成而导致结果失去意义。不过，从过程中也可以看出，二阶光滑性时加入 W-P 搜索可以显著提升收敛速度，尤其非凸的情况下，这样的搜索格外重要，可以避免越过最优位置太多。

牛顿法利用了二阶条件，因此在具有二阶光滑性时收敛比梯度法快。拟牛顿类方法试图逼近二阶条件，在有二阶条件时结果一般也好于梯度类方法。不过，正如讲义所写，秩 1 修正的拟牛顿可能破坏正定性，因此使用条件苛刻。实际测试里，对四次函数基本没有成功输出，而二次函数时达到了很精确的结果。

总体来看，具有二阶光滑性时，秩 2 修正的拟牛顿法表现最好。哪怕是 Rosenbrock 函数，也可以以较稳定的表现找到最小值，除非人为构造的极端误差。按照实验中的结果，个人感觉可以用一定程度的重启动来优化秩 2 修正优化拟牛顿法(例如提升新加入的部分权重)，避免误差的累计。

初值选取对各个方法的收敛速度都会产生影响，不过在凸函数的情况下基本不会影响最终的收敛。但对非凸函数，过差的初值就有导致迭代失败的可能。

附录-文件列表

report.pdf 实验报告
out.txt 控制台输出
iteration 程序文件
-- function.cpp 迭代法的主要内容
-- function.h 迭代法的头文件
-- iteration.cpp 测试函数与主函数
-- iteration.sln 解决方案文件
-- iteration.vcxproj
-- iteration.vcxproj.filters
-- iteration.vcxproj.user