

算法设计与分析实验报告



实验题目： 实验五 蒙图版钻石矿工算法的设计与分析

姓名： 陈俊卉

学号： 2020212256

日期： 2022. 11. 7

一、实验环境

(列出实验的操作环境，如操作系统，编程语言版本等，更多信息可视各自实际情况填写)

- ① 操作系统: windows 10
- ② 编程语言: c++
- ③ 编程工具: vscode 及其组件

二、实验内容

具体要求请参照实验指导书中各实验的“实验内容及要求”部分。

(示例: 1. 描述你对实验算法的设计思路; 2. 给出算法关键部分的说明以及代码实现截图; 3. 对测试数据、测试程序(没有要求则非必须)进行说明, 如测试覆盖程度, 最好最坏平均三种情况等等, 并给出测试结果截图等信息)

1. 算法的设计与实现

(1) 直接贪心策略

① 实验代码:

```
#include<iostream>
#include<ctime>
#include <cstdlib>
#include<chrono>
using namespace std;
using namespace std::chrono;

// 定义深度
const int n = 10000;
// 定义探测器的深度 (x < n) 只有蒙图版需要
// const int x = 3;
// 定义矿石数组
int a[n + 1][n + 1];
// 保存贪婪结果
int greed_res = 0;

// 设置矿石数组
void set_up_pyramid(){
```

```

// 是否设置随机种子
srand(int(time(0)));
for (int i = 1; i <= n; i++){
    for (int j = 1; j <= i; j++){
        a[i][j] = rand() % 10;
        // cout << a[i][j] << '\t';
    }
    // cout << endl;
}
}

void greedy_max_gold(int line_index, int column_index){
    // 如果在顶部
    if (line_index == 1) {
        greed_res += a[1][1];
    }
    // 如果达到底部
    else if (line_index == n) {
        return ;
    }

    // 比较下面两个矿石 走向更大的一边
    if (a[line_index + 1][column_index] >= a[line_index + 1][column_index + 1]){
        greed_res += a[line_index + 1][column_index];
        // cout << greed_res <<endl;
        greedy_max_gold(line_index + 1, column_index);
    }
    else {
        greed_res += a[line_index + 1][column_index + 1];
        // cout << greed_res <<endl;
        greedy_max_gold(line_index + 1, column_index + 1);
    }
}

int main(){
    set_up_pyramid();
    system_clock::time_point start2 = system_clock::now();
    greedy_max_gold(1, 1);
    system_clock::time_point end2 = system_clock::now();
    cout << "max_gold:" << greed_res << endl;
    cout << "running time:" << double(duration_cast<microseconds>(end2 -
start2).count()) * chrono::microseconds::period::num /

```

```
chrono::microseconds::period::den << "s" << endl;
}
```

② 原理:

直接贪心策略并不需要对全局进行把握,只需要自顶向下地考虑当前矿工所处位置下面一层的左右两个方块是否有钻石。这导致了所求得解并不一定是最优解。

(2) 全局动态规划

① 实验代码:

```
#include<iostream>
#include<ctime>
#include <cstdlib>
#include<chrono>
using namespace std;
using namespace std::chrono;

// 定义深度
const int n = 10000;
// 定义探测器的深度 (x < n) 只有蒙图版需要
// const int x = 3;

// 定义矿石数组
int a[n + 1][n + 1];
// 定义从第 i 层第 j 块出发到出口能获得的最大钻石价值
int A[n + 1][n + 1];
// 定义路径数组
int r[n + 1][n + 1];

// 设置矿石数组
void set_up_pyramid(){
    // 是否设置随机种子
    // srand(int(time(0)));
    for (int i = 1; i <= n; i++){
        for (int j = 1; j <= i; j++){
            a[i][j] = rand() % 10;
            // cout << a[i][j] << '\t';
        }
    }
}
```

```

        // cout << endl;
    }
}

// dp
int dp_max_gold(){
    // 从下向上计算
    for (int i = n; i >= 1; i--){
        // 从左到右计算
        for (int j = 1; j <= i ; j++){
            // 初始化最后一行
            if (i == n){
                // 赋值
                A[i][j] = a[i][j];
                r[i][j] = 0;
            }

            // 确定往左下走或往右下走
            else {
                // 往左下走
                if (A[i + 1][j] >= A[i + 1][j + 1]){
                    A[i][j] = a[i][j] + A[i + 1][j];
                    r[i][j] = j;
                }

                // 往右下走
                else if (A[i + 1][j] < A[i + 1][j + 1]){
                    A[i][j] = a[i][j] + A[i + 1][j + 1];
                    r[i][j] = j + 1;
                }
            }
        }
    }

    return A[1][1];
}

// 输出最佳路径
void get_best_way(){
    cout << "best way: " << a[1][1];
    for (int i = 1, j = 1; r[i][j] != 0; i++){
        j = r[i][j];
        cout << "-" << a[i + 1][j];
    }
    cout << endl;
}

```

```

}

int main(){
    set_up_pyramid();
    system_clock::time_point start2 = system_clock::now();
    int res = dp_max_gold();
    system_clock::time_point end2 = system_clock::now();
    // get_best_way();
    cout << "max_gold:" << res << endl;
    cout << "running time:" << double(duration_cast<microseconds>(end2 -
start2).count()) * chrono::microseconds::period::num /
chrono::microseconds::period::den << "s" << endl;
}

```

② 原理:

全局动态规划通过自底向上地求解子问题从而最终得到问题的解: 我们记:

$a[i][j]$ 为相应位置的钻石价值;

$A[i][j]$ 为自底向上到 $[i, j]$ 位置的最大数值之和;

$r[i][j]$ 为 $[i, j]$ 到达的下一层的列坐标(其实就是选择了左边还是右边)。

通过自底向上地求解每一个根为 $A[i][j]$ 的子金字塔的最优解(可以利用刚刚求得的子问题的解), 最终推出整个金字塔的最优值。

递归式如下所示:

$$A[i][j] = \begin{cases} a[i][j], & i = n \\ a[i][j] + \max(A[i+1][j], A[i+1][j+1]), & i < n \end{cases}$$

显然这个方法是一定能够求得最优解的。

(3) 蒙图版

① 实验代码:

```

#include<iostream>
#include<ctime>
#include <cstdlib>
#include<chrono>
using namespace std;
using namespace std::chrono;

// 定义深度

```

```

const int n = 10;
// 定义探测器的深度 (x < n) 只有蒙图版需要
const int x = 2;

// 定义矿石数组
int a[n + 1][n + 1];
// 定义路径数组
int r[n + 1][n + 1];
// 定义从第 i 层第 j 块出发到出口能获得的最大钻石价值
int A[n + 1][n + 1];

// 保存贪婪结果
int greed_res = 0;

// 设置矿石数组
void set_up_pyramid(){
    // 是否设置随机种子
    // srand(int(time(0)));
    for (int i = 1; i <= n; i++){
        for (int j = 1; j <= i; j++){
            a[i][j] = rand() % 10;
            cout << a[i][j] << '\t';
        }
        cout << endl;
    }
}

void cover_graph_max_gold(int line_index, int column_index){
    if (line_index == 1){
        greed_res += a[1][1];
        cout << greed_res << endl;
    }
    // 如果达到底部
    else if (line_index >= n) {
        return ;
    }

    // 有可能剩下的深度小于 x
    int size;
    if (line_index + x <= n) size = x;
    else size = n - line_index;

```

```

for(int i = line_index + size; i >= line_index; i--){
    for(int j = column_index; j <= column_index + size; j++){
        // 初始化最后一行
        if (i == n){
            // 赋值
            A[i][j] = a[i][j];
            r[i][j] = 0;
        }

        // 确定往左下走或往右下走
        else {
            // 往左下走
            if (A[i + 1][j] >= A[i + 1][j + 1]){
                if(i != line_index)
                    A[i][j] = a[i][j] + A[i + 1][j];
                else
                    A[i][j] = A[i + 1][j];
                r[i][j] = j;
            }

            // 往右下走
            else if (A[i + 1][j] < A[i + 1][j + 1]){
                if(i != line_index)
                    A[i][j] = a[i][j] + A[i + 1][j + 1];
                else
                    A[i][j] = A[i + 1][j + 1];
                r[i][j] = j + 1;
            }
        }
    }
}

cout << "best way: " << a[line_index][column_index];
for (int i = line_index, j = column_index; i < line_index + size; i++){
    j = r[i][j];
    cout << "-" << a[i + 1][j];
}

cout << endl;

```



```

greed_res += A[line_index][column_index];
cout << A[line_index][column_index] << endl;

// 寻找 next_column_index
int next_column_index = column_index;
for (int i = line_index; r[i][next_column_index] != 0; i++){
    next_column_index = r[i][next_column_index];
}
cout << "next_line_index:" << line_index + size << endl;
cout << "next_column_index:" << next_column_index << endl;
cover_graph_max_gold(line_index + size, next_column_index);
}

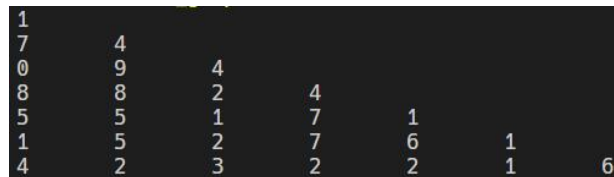
int main() {
    set_up_pyramid();
    system_clock::time_point start2 = system_clock::now();
    cover_graph_max_gold(1, 1);
    system_clock::time_point end2 = system_clock::now();
    // cout << endl;
    // for (int i = 1; i <= n; i++){
    //     for (int j = 1; j <= i; j++){
    //         cout << A[i][j] << '\t';
    //     }
    //     cout << endl;
    // }
    cout << "max_gold:" << greed_res << endl;
    cout << "running time:" << double(duration_cast<microseconds>(end2 -
start2).count()) * chrono::microseconds::period::num /
chrono::microseconds::period::den << "s" << endl;
}

```

② 原理：

蒙图版的钻石金字塔问题的解法是动态规划和贪心的结果。由于矿工只能探测出 x 层的钻石价值，所以不可能通过全局信息求得一个最优解。我们只能通过对每一次探测得到的局部信息求得局部最优解，通过多个局部最优解试图逼近真正整体的最优解。而求局部最优解对应着局部的动态规划算法，而局部最优解之间直接相加则对应着贪心算法。

但为了局部之间的衔接，逻辑发生了些许的变化，下面用一个例子讲解。



1						
7	4					
0	9	4				
8	8	2	4			
5	5	1	7	1		
1	5	2	7	6	1	
4	2	3	2	2	1	6

生成的矿石金字塔如上图所示。我们假设探测器能探测的深度为 $x=2$ 。

同时因为第一层的方块矿工无论如何都需要挖开，所以我们的探测是从第一层开始，探测第二层和第三层，而不是从第零层开始探测第一层和第二层。

我们先将第一层的价值累加到 `greed_res`：

```
38 void cover_graph_max_gold(int line_index, int column_index){
39     if (line_index == 1){
40         greed_res += a[1][1];
41         cout << greed_res << endl;
42     }
```

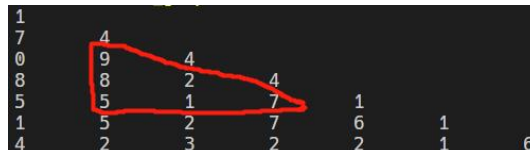
然后位于 `[1, 1]` 对下面两层进行探测。其实就是对第一层到第三层的子金字塔：



1		
7	4	
0	9	4

求全局动态规划。求得该局部最优解和矿工的落点 `[3, 2]`，也就是价值为 9 的地方，将得到的局部最优解 16 累加到 `greed_res`。（而不是 17，因为刚刚处理了第一层的价值，下面也同理，我们不对矿工在的地方的价值进行累加，因为这个地方的价值已经在上一轮的局部求解中计算过了）

继续在此处向下探测，也就是求该部分的局部最优解：



1						
7	4					
0	9	4				
8	8	2	4			
5	5	1	7	1		
1	5	2	7	6	1	
4	2	3	2	2	1	6

求得局部最优解为 13。以此类推，我们就能够求得最优解。

当然，存在一个特殊情况：当剩下的层数比能够探测到的深度 x 小时，我们需要调整下一次的局部大小 `size` 为剩下的层数：

```

48 // 有可能剩下的深度小于x
49 int size;
50 if (line_index + x <= n) size = x;
51 else size = n - line_index;

```

2. 算法测试

(1) 正确性

我们使用 `rand()` 和 `srand()` 函数随机生成每层的钻石价值。在验证正确性时，我们使用较小的规模方便判断正误；使用默认的随机种子，确保每次生成的价值分布是一样的，以进行对比。

① 直接贪心策略

```

1
7 4
0 9 4
8 8 2 4
5 5 1 7 1
1 5 2 7 6 1
4 2 3 2 2 1 6
8 5 7 6 1 8 9 2
7 9 5 4 3 1 2 3 3
4 1 1 3 8 7 4 2 7 7
8
17
25
30
35
38
45
50
53
max_gold:53

```

观察每次选取左右的较大价值并累加得到的数字 8、17、25、...、53，并进行推演，可以得知过程没有错误。

② 全局动态规划

```

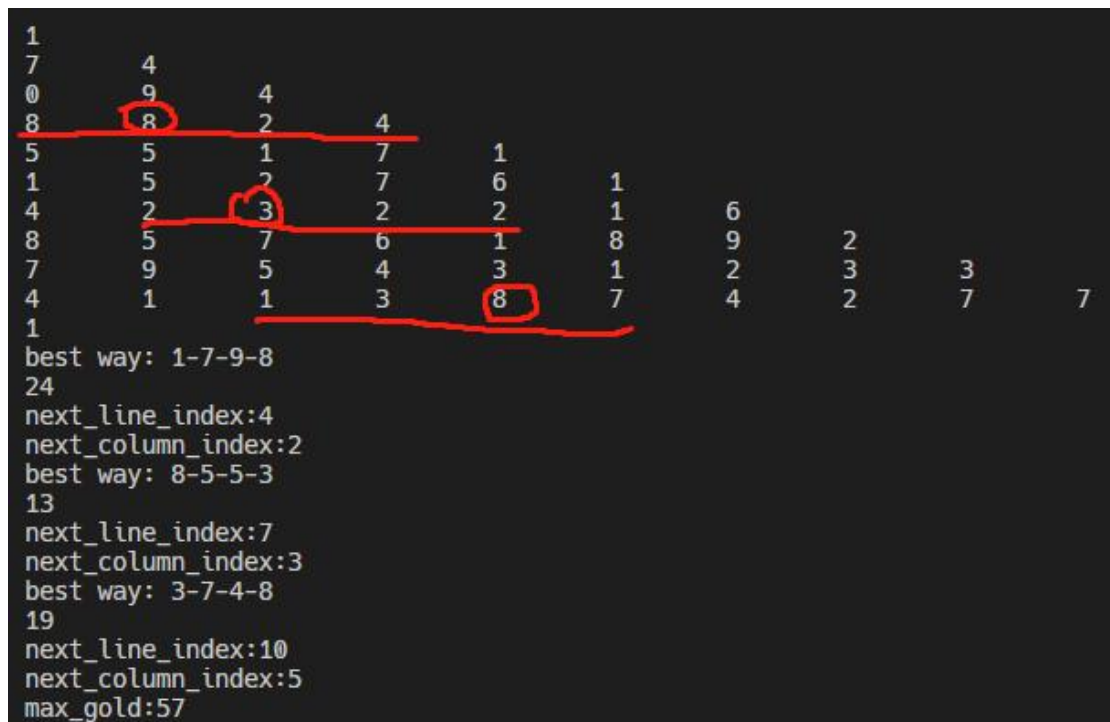
1
7 4
0 9 4
8 8 2 4
5 5 1 7 1
1 5 2 7 6 1
4 2 3 2 2 1 6
8 5 7 6 1 8 9 2
7 9 5 4 3 1 2 3 3
4 1 1 3 8 7 4 2 7 7
best way: 1-7-9-8-5-5-3-7-4-8
max_gold:57

```

经手算验证，答案是正确的。

③ 蒙图版

验证时使用的探测器最大探测深度为 $x=3$.



上图金字塔是生成的钻石金字塔，红线标出的是每一个子问题求解的区域。第一个区域路径为 1-7-9-8，上文说到矿工所处的位置的价值不计入累加，因为该位置的价值会被上一个子问题累计（第一层单独累加），所以第一个子问题的答案为 $7+9+8=24$ ，在下面的输出也有体现。然后确定矿工所处的位置[4, 2]，也就是价值为 8 的地方，进行下一个子问题的求解，以此类推求得解。（路径通过 r 数组记录）

事实上，我们可以在每一次求解子问题的时候都重新生成一个规模为 $x \times x$ 的数组用作记录（因为最大规模的子问题就为 $x \times x$ ），甚至可以不使用 A 数组进行记录，直接在钻石金字塔的价值数组 a 中进行修改，因为在 dp 的过程中使用过的信息不会再被使用第二次，所以完全可以进行信息的覆盖。但考虑到展示的直观性，笔者还是使用了这种直观的方式写出代码。

(2) 性能分析（去掉所有 log 输出）

1) 时间复杂度

本部分我们使用 C++ 中的 `srand(int(time(0)))` 设置随机种子，并对三个算法在不同规模问题上进行分析。假设钻石价值在 10 以内（这并不会对时间复杂度造成巨大的影响）。每个规模测试五组求得平均时间。

① 直接贪心策略

规模	N=100	N=1000	N=10000
平均时间	0s	0s	0.000997s

② 全局动态规划

规模	N=100	N=1000	N=10000
平均时间	0s	0.01001s	0.812635s

③ 蒙图法

x = 10:

规模	N=100	N=1000	N=10000
平均时间	0.001s	0.004989s	0.035s

x = 50:

规模	N=100	N=1000	N=10000
平均时间	0.000963s	0.003962s	0.041891s

x = 100:

规模	N=100	N=1000	N=10000
平均时间	0s	0.005983s	0.047873s

x = 500:

规模	N=1000	N=10000
平均时间	0.009977s	0.110726s

可以看出：

直接贪心策略的平均时间均很短，这是因为每一行只需做一次判断与累加，所以是 $O(n)$ ；
全局动态规划基本存在一个二次的关系曲线；

而对于蒙图法，当 x 增加时，虽然子问题数目减少了，但子问题规模增加，且子问题是 $O(x^2)$ ，而 x 增大带来的数目减少只是一次的，所以 x 增大，会带来平均时间的增加。

具体的时间复杂度分析会在下文阐述。

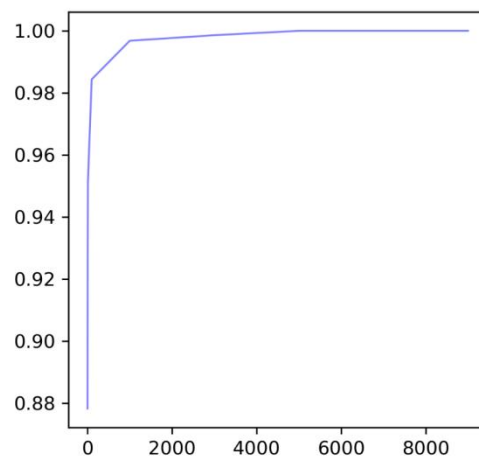
2) 蒙图版的准确率趋势

采用默认的随机种子。固定 n 规模为 10000， x 规模变化。

随机出的数组的最大获取价值为 **69696**。

规模	X=1	X=5	X=10	X=100	X=1000	X=3000	X=5000	X=7000	X=9000
解	61206	64653	66294	68606	69470	69598	69696	69696	69696
占比	0.8782	0.92764	0.9512	0.98436	0.9968	0.99859	1	1	1

使用 python 绘图如下：



可以看出，在 $n=10000$ 规模，当 x 增大到 2000 之后，实际上得到的解与最优解相差无几。且下面将会介绍到，蒙图版的效率是比全局动态规划的效率高的。实际上，在工程应用时，我们更多时候会使用蒙图版这种办法去大大减少计算量，即使全局的数据我们是已知的。

(3) 复杂度分析

① 直接贪心策略

时间复杂度：很显然，直接贪心策略对于每一层只需要做一次判断与一次累加。

所以时间复杂度为 $O(n)$ 。

空间复杂度：除了存储钻石价值的数组外，我们没有用到额外的数组进行存储，只用了 `greed_res` 变量进行累计。

所以空间复杂度为 $O(1)$ 。

② 全局动态规划

时间复杂度：我们需要从下往上对每一层进行计算；对于每一层，我们需要对每一个方块进行 dp 比较操作（也就是判断和赋值）。所以时间复杂度与方块的个数和每个方块的计算次数有关。而每个方块的计算次数是常数，方块个数为 $(1+n)*n/2$ 。

所以时间复杂度为 $O(n^2)$ 。

空间复杂度：上文提到，我们为了输出的清晰，使用了 A 数组来存储 dp 的过程，但实际上可以不使用 A 数组进行记录，直接在钻石金字塔的价值数组 a 中进行修改，因为在 dp 的过程中使用过的信息不会再被使用第二次，所以完全可以进行信息的覆盖。所以我们在不需要记录路径时并不需要额外的数组。空间复杂度为 $O(1)$ 。

但需要记录路径时，因为需要原来的钻石价值来为每个方块标记，所以 A 数组和 r 数组都是需要的。此时空间复杂度为 $O(n^2)$ 。

③ 蒙图法

时间复杂度：问题分解为 $\text{ceil}(n/x)$ 个子问题。每个子问题的规模为 x（除最后一个子问题可能小于 x）。对于规模为 x 的子问题，使用动态规划，时间复杂度为 $O(x^2)$ 。

所以总时间复杂度为 $O(\frac{n}{x}x^2) = O(xn)$

空间复杂度：

第一种记录方式：建立一个与原金字塔规模相同的数组，在对应位置记录子问题的 A 与 r。空间复杂度为 $O(n^2)$

第二种记录方式：每一次子问题都新建规模为 x 的数组记录子问题的 A 和 r。空间复杂度为 $O(x^2)$

第二种记录方式：不记录路径，直接在原金字塔进行 dp。空间复杂度为 $O(1)$ 。

三、出现问题及解决

（列出你在实验中遇到了哪些问题以及是如何解决的）

问题一：蒙图法是否能确保得到最优解。

事实上，没有全局信息注定不能够确保最优解的产生。因为最优解势必要同时利用到所有层的所有信息。

四、总结

（对所实现算法的总结评价，如时间复杂度，空间复杂度，是否有能够进一步提

升的空间，不同实现之间的比较，不同情况下的效率，通过实验对此算法的认识与理解等等)

本次实验，我对直接贪心算法、全局动态规划和蒙图动态规划有了深刻的认识。我不仅更加熟悉了贪心和动态规划两种算法的过程，而且对蒙图法的方法研究和在工程上的使用有了更加深刻的理解。

事实上，许多工程相关的问题，尤其是人工智能、神经网络相关的模型调参问题，基本是不可能得到一个最优解的，因为当前的算力还不支持。但通过观察蒙图法的占比图像可以管中窥豹：

很多时候，去达到一个比较接近最优解的解所需要的算力其实相对而言很小，只是在趋近于最优时，算力增大对于趋近最优解的收益很小。就如这个采矿问题一样，研发一个 $x=100$ 的探测器和一个 $x=1000$ 的探测器，成本可能天差地别，但得到的收益其实并没有想象中的这么大。这启发我们在工程中我们不一定非要求得一个最优解，而是要兼顾投入和产出，得到一个最佳的收益。