北京郵電大學

实验报告

蒙图版钻石矿工算法的设计与分析

学院：计算机学院（国家示范性软件学院）

专业：计算机科学与技术

班级：2022211305

成员 1：2022211683 张晨阳

成员 2：2022211124 梁维熙

2024 年 12 月 12 号

目录

1.实验概述

1.1. 实验目的

- 理解动态规划算法的策略，掌握 DP 算法避免重复计算的方法；
- 掌握基于最优子结构递推分解原问题和子问题的基本方法
- 掌握自底向上的 DP 算法的实现方法；
- 理解基于全局动态规划的 DP 算法在实际应用中的局限性，掌握基于局部动态规划和贪心策略相结合的 DP 算法的设计方法；

1.2. 实验内容及要求

1. 算法的设计与实现

- 问题描述

经典的钻石矿工问题描述如下：

有一座金字塔，金字塔的每块石头上都镶有对应的钻石，钻石可以被取下来，不同的钻石有着不同的价值，例如图 1 所示，你的任务是从金字塔的顶端向金字塔的底端收集钻石，并且尽可能收集价值高的钻石，但是只能从一块砖斜向左下或斜向右下走到另一块砖上，如图 1 从用红色 A 标记的砖走向用蓝色 B 标记的砖上。请找到一个收集最高价值钻石的路线，并给出能够获得的最大钻石总价值？

课堂上，我们基于动态规划方法给出了该问题的基本求解算法。此时，我们能够实用动态规划算法解决该题，是因为我们实现得到了整个金字塔的钻石价值分布，因为可以通过动态规划算法求解——全局动态规划算法。

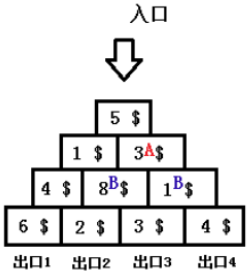


图 1 钻石金字塔

- 蒙图版的钻石金字塔问题

如图 2 所示，在实际应用中，矿工事先往往无法提前知道金字塔的钻石分布。通常，他们只能估计面前两个方块内的钻石数，或者租用探测器来获得面前 x 步（$x < n$，n 为金字塔的层数）内钻石的分布。又或者，假设他有一张残破的地图。在上述这些情况下的信息量和矿工的收益有怎样的关系呢？请设计并实现能够获得最大价值算法，包括找寻最佳路径及最大价值。

图 2 蒙图版钻石金字塔

2. 实验内容

① 数据生成：通过随机或高斯等随机方法生成矿产的模拟分布图。

② 算法实现：分别基于 X 步（ x<n，n 为金字塔的层数）已知，及蒙图版（残缺地图版）的挖矿算法。

1.3. 实验环境

- Visual Studio Code
- C++ 17
- Python 3.12.3

2.算法设计与实现

2.1. 地图生成算法

2.1.1.随机数填充

```
vector<vector<int>> pyramid;  // 存储金字塔的二维数组

// 使用随机数值填写金字塔
void pyramid_random_fill(int n) {
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n - i; j++) {
            pyramid[i][j] = rand() % 100;
        }
    }
}
```

在本程序中，我们使用二维向量模拟矩阵的形式进行数据存储，其中矩阵的左上部分用于存储数据，右下部分作为冗余进行矩阵填充并防止越界访问时导致段错误。

函数通过二重循环的方式进行金字塔的遍历，并使用 rand 方法生成随机数进行填充。

通过 rand 函数生成的随机数不具有规律性，其分布在生成样本量极大时能够呈现一定规律性，但是我们生成的金字塔层数过小，无法达到相对应的数量级。

我们可以认为 rand 函数生成的随机数在改数量级下无法呈现规律性，其生成的金字塔没有参考价值，故不选择该方式进行数据生成。

2.1.2.正态分布填充

```
// 使用正态模拟的方式填充金字塔
void pyramid_normal_fill(int n, int k) {
    // 定义正态分布参数
    double mean = 50.0;    // 均值
    double stddev = 10.0;  // 标准差

    // 创建一个正态分布的随机数生成器
    default_random_engine generator;
    normal_distribution<double> distribution(mean, stddev);
    uniform_int_distribution<int> point_distribution(0, n - 1);
    pyramid.assign(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n - i; ++j) {
            double value = distribution(generator);
            int clampedValue = static_cast<int>(round(value));
            if (clampedValue < 0)
                clampedValue = 0;
            if (clampedValue > 100)
                clampedValue = 100;
            pyramid[i][j] = clampedValue;
        }
    }
    cout << "Pyramid has been filled with values:" << endl;
}
```

正态分布相较于使用 rand 函数进行随机数生成，我们可以通过指定平均值与标准差，对生成的随机数进行生成概率控制，使得填充的金字塔更加具有规律性。

该函数中的正态分布通过调用 std 方法中的 default_random_engine 与 normal_distribution 两个类实现。

通过使用正态分布能够较好的弥补使用 rand 函数在改数量级下无法呈现规律性的问题，但是此方法生成的数据产生数据聚集块的情况极少，基本可以认为其数据不聚块，这与金字塔中拥有矿脉的情况相悖。

故我们也排除该生成方式。

2.1.3.高斯函数填充

```
double gaussian(double x, double y, double sigma) {
    return exp(-(x * x + y * y) / (2 * sigma * sigma));
}

double random_noise(double scale) {
    return (static_cast<double>(rand()) / RAND_MAX - 0.5) * scale;
}
// 使用高斯函数进行填充（最优填充方案）
void pyramid_gaussian_fill(int n, int k) {
    srand(time(0));
    pyramid.resize(n, vector<int>(n, 0));
    // 确定聚集块中心点的数量和位置
    int numClusters = k;
    vector<pair<double, double>> clusterCenters;
    // 随机选择 k 个中心点
    for (int i = 0; i < numClusters; ++i) {
        double centerX = static_cast<double>(rand()) / RAND_MAX * n / 2;
        double centerY = static_cast<double>(rand()) / RAND_MAX * (n - centerX);
        clusterCenters.push_back({centerX, centerY});
    }
    // 使用高斯分布函数填充聚集块中心点附近的值
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            int value = 0;
            for (const auto& center : clusterCenters) {
                double g = gaussian(i - center.first, j - center.second,
3.0);  // 调整 σ 值以控制聚集块的形状和强度
                value += static_cast<int>(g * 100);   // 范围在 0-100 之间
            }
            if (value > 100)
                value = 100;
            pyramid[i][j] = value;
        }
    }
    // 引入噪声进行空白位置填充
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            pyramid[i][j] += static_cast<int>(random_noise(65));
            if (pyramid[i][j] < 0)
                pyramid[i][j] = 0;
            if (pyramid[i][j] > 100)
```

```
                pyramid[i][j] = 100;
        }
    }
}

```

gaussian 函数实现使用二维高斯函数运算位置(x, y)处的点对应的高斯函数值，其实现为下式：

$$G(x,y) = exp(-\frac{x^2 + y^2}{2\sigma^2})$$

函数 random_noise 实现从范围（scale,，scale）中生成双浮点精度的随机数。

函数 pyramid_gaussian_fill 通过使用 rand 方法生成 k 个数值聚集块的中心点，而后使用 gaussian 函数计算中心点附近点与中心点的距离关系以生成对应的二维高斯函数值。通过这种方式，我们得以实现矿脉聚集的现象。在完成矿脉的生成后，我们使用 random_noise 函数在还未进行填充的位置填充随机数值，让生成的矿脉图具有更强的随机性。

通过这种方式，我们能够生成具有聚块的钻石金字塔，同时能够保证数据的出现有一定规律性，当算法进行路径选择时能够有更明显的趋向，更便于我们对于路径优劣进行判断。

2.2. 贪心算法

```cpp
// 贪心算法
int miner_greedy(int n) {
    int x = 0, y = 0;
    int total_value = 0;
    while (x + y < n - 1) {
        if (x + 1 < n - y && y + 1 < n - x) {
            if (pyramid[x + 1][y] > pyramid[x][y + 1]) {
                total_value += pyramid[x + 1][y];
                path.push_back(make_pair(x + 1, y));
                x++;
            } else {
                total_value += pyramid[x][y + 1];
                path.push_back(make_pair(x, y + 1));
                y++;
            }
        }
    }
    cout << "path find successfully" << endl;
    return total_value;
}
```

贪心算法是每个人生来就会的算法，他与我们的直觉最为接近。由于没有探测器，每个矿工只能够看到眼前两个位置包含的矿的价值，我们通过选取两个矿中价值最高的即可完成普通矿工的选择。

2.3. 全局动态规划算法

```
1. // 动态规划求解最优路径
2. int dp_all(int n) {
3.     vector<vector<int>> values(n, vector<int>(n, 0));
4.     vector<vector<vector<pair<int, int>>>> path_all(n, vector<vector<pair<int,
int>>>(n, vector<pair<int, int>>()));
5.
6.     for (int i = 0; i < n; i++)
7.         for (int j = 0; j < n - i; j++) {
8.             if (i == 0 && j == 0) {
9.                 values[i][j] = pyramid[i][j];
10.                path_all[i][j].push_back({i, j});
11.            } else if (i == 0) {
12.                values[i][j] = values[i][j - 1] + pyramid[i][j];
13.                path_all[i][j] = path_all[i][j - 1];
14.                path_all[i][j].push_back({i, j});
15.            } else if (j == 0) {
16.                values[i][j] = values[i - 1][j] + pyramid[i][j];
17.                path_all[i][j] = path_all[i - 1][j];
18.                path_all[i][j].push_back({i, j});
19.            } else {
20.                values[i][j] = max(values[i - 1][j], values[i][j - 1]) +
pyramid[i][j];
21.                if (values[i - 1][j] > values[i][j - 1]) {
22.                    path_all[i][j] = path_all[i - 1][j];
23.                    path_all[i][j].push_back({i, j});
24.                } else {
25.                    path_all[i][j] = path_all[i][j - 1];
26.                    path_all[i][j].push_back({i, j});
27.                }
28.            }
29.        }
30.
31.    int ret = values[0][n - 1];
32.    for (int i = 1; i < n; i++)
33.        for (int j = n - 2; j >= 0; j--)
34.            if (values[i][j] > ret) {
35.                ret = values[i][j];
36.                path = path_all[i][j];
37.            }
38.    return ret;
39. }
```

全局动态规划可以理解为上帝视角/探测距离无限大，矿工可以知道地图的全部情况。那么我们通过划分子问题，可以得到：

矿工走到每一个点的最大值=该点的值+该点前一步的位置的最大累计价值

写为状态转移方程即为：

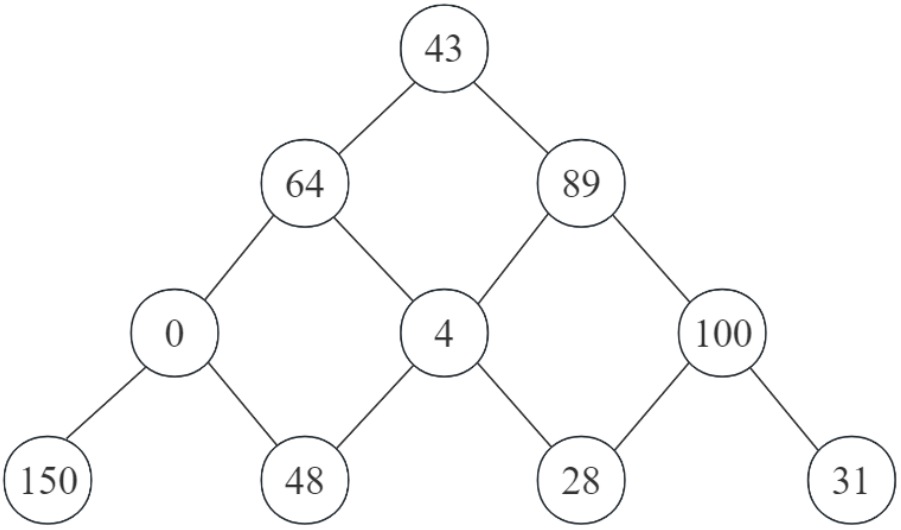
$$values[i][j] = max(values[i-1][j], values[i][j-1]) + pyramid[i][j]$$

对于边界情况：

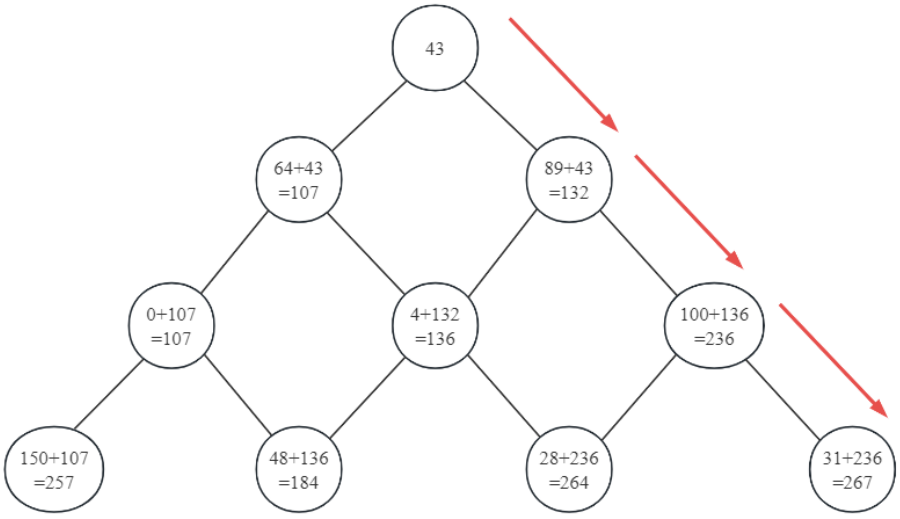
1. 当矿工处于起点时，values 的值即为起点的值；
2. 当矿工处于金字塔的两条边时，values 只能由上一个点（确定）+该点的值。

举例说明：

假设下图为我们的金字塔：



下图为全局动态规划之后得到的，到达每个点时可获得的最大值：



对于这个四层金字塔，我们很容易可以得到路径：

$$(0,0) \rightarrow (0,1) \rightarrow (0,2) \rightarrow (0,3)$$

我们只需要查看最下层的最大值，即可知道最后一步走到了哪个点。

再向上看到倒数第二层，即为倒数第二步的点。

……

依次类推，直到回到顶点。

在我们的程序实现中，在更新每一个点的 values 的值的时候，也同时记录起点到达该点所需的路径，不断向下更新，最终得到所有点的“最优路径”path_all。选取最大值的点即为最优路径 path。

2.4. 蒙图版动态规划

```
1. // 探测范围为 x 的动态规划算法
2. // 传入起始点的坐标
3. int dp_x(int start_i, int start_j, int n, int x) {
4.     // 先判断最大探测范围还剩多少
5.     if (start_i + start_j + x > n - 1)
6.         x = n - 1 - start_i - start_j;
7.
8.     int ret = 0;
9.     vector<vector<int>> values(n, vector<int>(n, 0));
10.
11.    for (int i = start_i; i < start_i + x + 1; i++)
12.        for (int j = start_j; j < start_j + x + 1; j++) {
13.            if (i == start_i && j == start_j) {
14.                values[i][j] = pyramid[i][j];
15.            } else if (i == start_i) {
16.                values[i][j] = values[i][j - 1] + pyramid[i][j];
17.            } else if (j == start_j) {
18.                values[i][j] = values[i - 1][j] + pyramid[i][j];
19.            } else {
20.                values[i][j] = max(values[i - 1][j], values[i][j - 1]) +
pyramid[i][j];
21.            }
22.        }
23.
24.    for (int i = start_i; i < start_i + x + 1; i++)
25.        for (int j = start_j + x; j >= start_j; j--)
26.            if (values[i][j] > ret) {
27.                ret = values[i][j];
28.            }
29.
30.    return ret;
31. }
32.
33. // 矿工前进过程
34. int step(int n, int x) {
35.     int ret = pyramid[0][0];
36.
37.     // 起始位置
38.     int current_i = 0;
39.     int current_j = 0;
40.
```

```
    while (current_i + current_j < n) {
        int left_value = -1;
        int right_value = -1;
        // 先处理右边的局部最优
        if (current_i < n && current_j + 1 < n) {
            right_value = dp_x(current_i, current_j + 1, n, x - 1);
        }
        // 再处理左边的局部最优
        if (current_i + 1 < n && current_j < n) {
            left_value = dp_x(current_i + 1, current_j, n, x - 1);
        }

        // 说明到达金字塔底部
        if (right_value == -1 && left_value == -1)
            break;

        // 右边的局部最优比左边大，向右下方走一步
        if (right_value >= left_value) {
            current_j += 1;                          // 向右走一步
            ret += pyramid[current_i][current_j];    // 累加这一步的价值
            path.push_back({current_i, current_j});  // 记录这一步的路径
        } else {
            current_i += 1;                          // 向左走一步
            ret += pyramid[current_i][current_j];    // 累加这一步的价值
            path.push_back({current_i, current_j});  // 记录这一步的路径
        }
    }

    return ret;
}
```

对于不同的探测范围 x，前面两种算法可以理解为 x 的两种特殊情况：

- 贪心：$x = 1$
- 全局动态规划：$x = \infty$

那么对于其他的 x 值，我们的算法逻辑如下：

因为我们已知每一步都只能向左下或右下走，所以对于当前位置只需要考虑两种情况：

1. 向左走获得的最大值；
2. 向右走获得的最大值；

而因为我们的 x 不再是无限大，所以只能看到当前层数$i + x$层的值。而对于

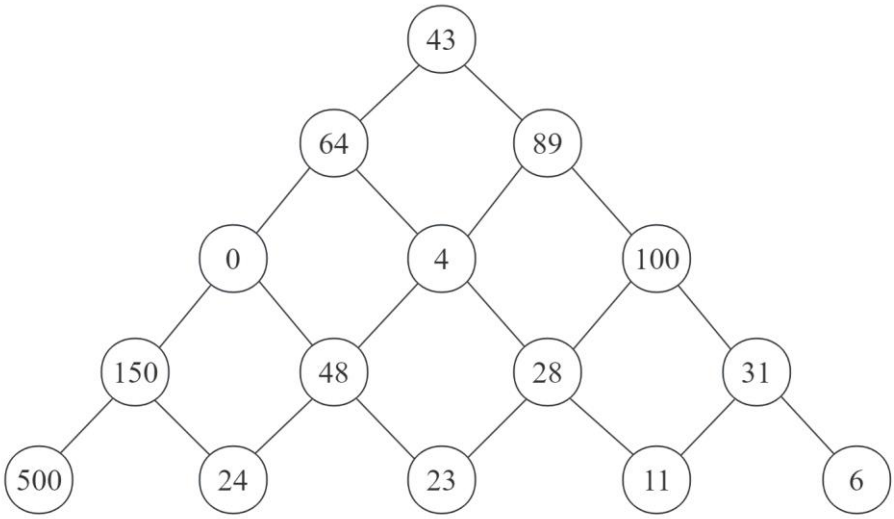
下一步（即下一层），我们只能看到距离为$x-1$的范围。

在这个限制下，我们的算法逻辑为：

1. 从当前位置出发，对于左下和右下，分别考虑各自探测范围为$x-1$（下一层也算在探测范围中）的局部金字塔，进行局部的动态规划；
2. 得到两个情况的最大值，决定下一步是向左还是向右；
3. 将下一步作为新的当前位置，从第 1 步开始直到到达最底层。

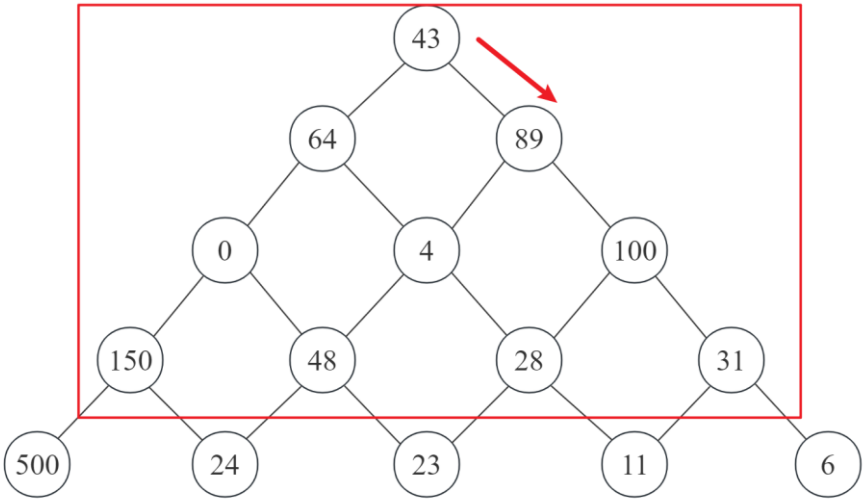
举例说明：

假设在上一个算法的金字塔基础上，我们又添加了一层如下：

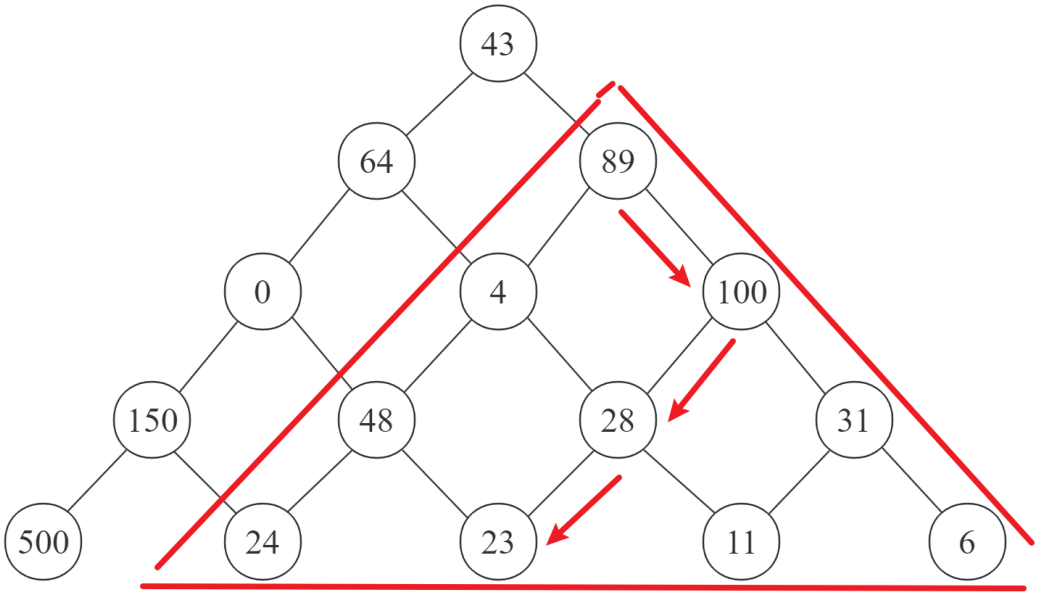


很明显，最左下角的路线一定是最大的，因为 500 远大于其他点的值。

但假设我们的矿工的的探测范围是 3，则他一开始看到的地图为：



由上一节我们知道，对于这个局部金字塔，局部最优解是靠右走，那么矿工会走到 89，之后矿工会继续探测，范围为 3，如下：



与上一节的路线不同，矿工在 89 的位置，继续进行左右两边局部的动态规划，得到下一步为 100，继续向下。这时发现：$28 + 23 > 31 + 11$（左右两步的各自最大值），得到新的路线：

$$(0,0) \rightarrow (0,1) \rightarrow (0,2) \rightarrow (1,2) \rightarrow (2,2)$$

最终完成挖矿。

很明显，矿工由于探测范围的限制，根本没有机会了解到全面的地图，所以会在他探测到的局部金字塔中，计算局部最优的下一步。这也是我们将在后面分析结果时提到的探测范围 m 会陷入局部最优的问题。

除此之外，在具体代码实现过程中，边界情况需要特殊考虑，以实现在代码中并添加了注释，在此不多赘述。

3.测试程序与可视化

3.1. 三种算法结果输出

```
vector<pair<int, int>> path;
// 打印路径至文件中
void print_path_to_file(int n) {
    ofstream outfile("../path.txt");
    // cout<<"1"<<endl;
    if (!outfile.is_open()) {
        cerr << "Failed to open file 'path.txt'" << endl;
        return;
    }
    // outfile << n << endl;
    for (int i = 0; i < path.size(); i++) {
        outfile << path[i].first << " " << path[i].second << endl;
    }
    cout << "path has been printed to file." << endl;
    outfile.close();
}
```

为了将路径可视化，我们选择将每个算法计算出的路径输出至文件中，同时使用 python 中的 matplotlib 模块对输出的路径进行可视化。

首先，我们使用一个一维向量进行路径的记录，其中每个路径选择点使用一个 pair<int,int>进行记录，这样我们就能够将路径通过坐标点的形式分解并记录。在函数 print_path_to_file 中，我们选择将路径以每行两个数字，分别代表每个路径点的横纵坐标进行输出，输出得到的文件结果如下：

```
0 1
0 2
…
…
…
29 69
30 69
```

文件中共有 n 行，记录着算法规划的路径。

3.2. 三种算法路线可视化

1. **数据读取：**

代码首先从名为 pyramid_map.txt 的文本文件中读取金字塔的数据，每一行包含一个层级的整数数据，以逗号分隔。同时，它将这些数据转换成整数列表，并去除可能存在的换行符。代码创建一个 NumPy 数组 matrix，其大小足以容纳所有行数据的最大宽度，并填充这个矩阵。

2. **矩阵填充：**

代码遍历每一行数据，将它们填充到 matrix 数组中，确保每行的数据都被正确地放置。然后，代码尝试从 path_dp_x_{k}.txt 文件中读取路径数据，这些数据由一系列坐标点组成。对于每个路径点，如果它们在矩阵的范围内，代码将这些点的值设置为 110，以标记路径。

3. **图像绘制：**

通过使用 Matplotlib 库，程序绘制一个热力图以显示金字塔的数据，并且特别标记其中的路径点。图像使用红色调的色图（Reds）和最近邻插值（nearest）来显示。图像被赋予标题、轴标签，并添加了颜色条。最后，代码确保保存路径的目录存在，并将绘制的热力图保存为 PNG 文件，文件名包含 k 的值，以区分不同的图像。

这个函数中我们传入三个参数，分别为路径文件路径，金字塔数据文件路径，输出图像路径。

3.3. 不同探测度结果曲线可视化

1. 读取数据文件：

使用 with 语句打开存储有不同探测范围对应的路径总价值的文件，这样可以确保文件在操作完成后自动关闭。通过循环读取文件的每一行，并将每行的值转换为浮点数后添加到 x_values 列表中。同时，将当前行号（从 0 开始）添加到 count 列表中，作为每个值的索引或范围。

2. 绘制图表：

使用 plt.plot() 函数绘制 count 列表（作为 x 轴）和 x_values 列表（作为 y 轴）之间的关系图。设置图表的标签为 ranges vs. values，表示图表显示的是范围与值的关系。设置 x 轴标签为 ranges，y 轴标签为 values。设置图表标题为 ranges vs. values。为图表添加图例，用于标注图表中的曲线的表示内容。

3. 保存图表：

使用 plt.savefig() 函数将图表保存至指定路径中。

3.4. 探究局部最优原因可视化

我们通过将所有探测范围对应的图表进行打印，通过观察图表分析不同探测范围中路径的选择问题与其路径选择优劣。

4.算法效率与结果分析

4.1. 三种算法效率对比

假设金字塔的层数为 n。

4.1.1.贪心算法

1. 贪心算法逻辑

- 在每一步，代码比较两个可能的路径：向下$(x+1,y)$和向右$(x,y+1)$，选择值较大的路径。
- 这一过程通过逐步逼近底部，形成一条从顶部到底部的路径。

2. 循环执行次数

- 该算法从顶点 $(0,0)$ 开始，每次执行一次比较，直到到达金字塔底层。
- 金字塔底层的索引为$n-1$，因此一共需要移动 $n-1$ 步。

3. 时间复杂度

- 每一步选择时，算法进行一次比较操作，这需要 O(1) 的时间。
- 总共移动 $n-1$ 步，因此总时间复杂度为：

$$O(n)$$

4.1.2.全局动态规划

1. 全局动态规划算法逻辑

- 创建一个二维数组 values 来存储从起点到每个点的最大值。
- 使用另一个数据结构 path_all 来存储到达每个点的路径。
- 动态规划递推公式：

$$values[i][j] = max(values[i-1][j], values[i][j-1]) + pyramid[i][j]$$

2. 时间复杂度分析

- **动态规划计算 values**
  - 外层双重循环遍历二维数组，每个点的计算复杂度为 $O(1)$。
  - 金字塔是一个下三角矩阵，点的总数为：

$$T = n \times (n+1)/2 \approx O(n^2)$$

  - 因此，动态规划的时间复杂度为： $O(n^2)$
- **路径回溯**
  - 回溯路径时，通过访问 path_all，每条路径最多存储 $O(n)$ 个点。
  - 假设需要输出路径的次数为 k，那么路径回溯的总时间复杂度为：

$$O(k \cdot n)$$

- **总体时间复杂度**
  - 通常情况下，路径数量 k 为常数，回溯路径的时间复杂度为 $O(n)$。
  - 因此，总体时间复杂度为：

$$O(n^2) + O(n) = O(n^2)$$

4.1.3.蒙图版动态规划算法

1. 算法核心逻辑

- 使用动态规划计算从某个起点出发，在限定探测范围 x 内可以获得的最大路径值。
- 利用一个辅助函数 dp_x，在范围内计算局部最优路径值。
- 在主函数 step 中，每次移动都调用 dp_x，在当前位置的下方和右方选择下一步方向，从而构成全局路径。

2. 时间复杂度分析

- **dp_x 的时间复杂度**
  - 探测范围为 x，dp_x 遍历范围内的所有点。
  - 探测范围是一个矩形（右下部分认为是 0），点的数量为：

$$T = (x+1)^2 = O(x^2)$$

  - 因此，dp_x 的时间复杂度为：

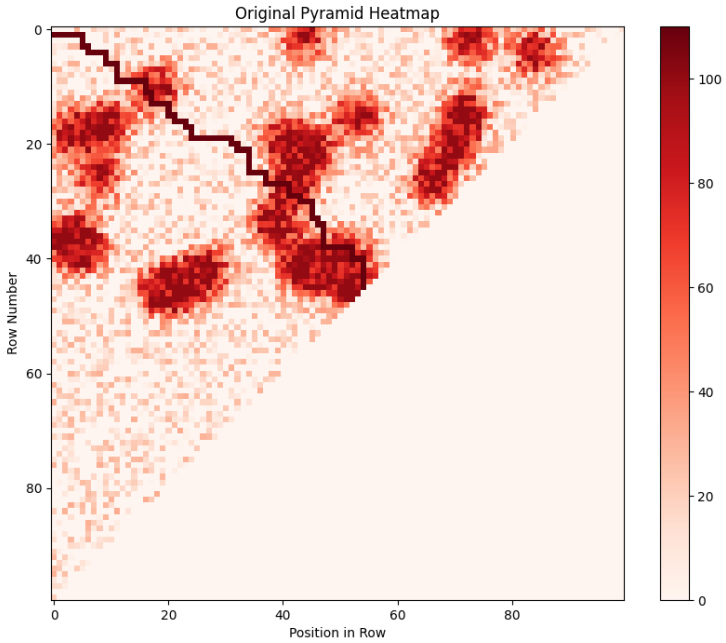
$$O(x^2)$$

- **step 的时间复杂度**
  - 每次调用 dp_x，分别计算右侧和下方的局部最优路径，最多调用 2n 次（沿着对角线移动）。
  - 每次调用的 dp_x 的复杂度为 $O(x^2)$。
  - 总时间复杂度为：

$$O(2n \cdot x^2) = O(n \cdot x^2)$$

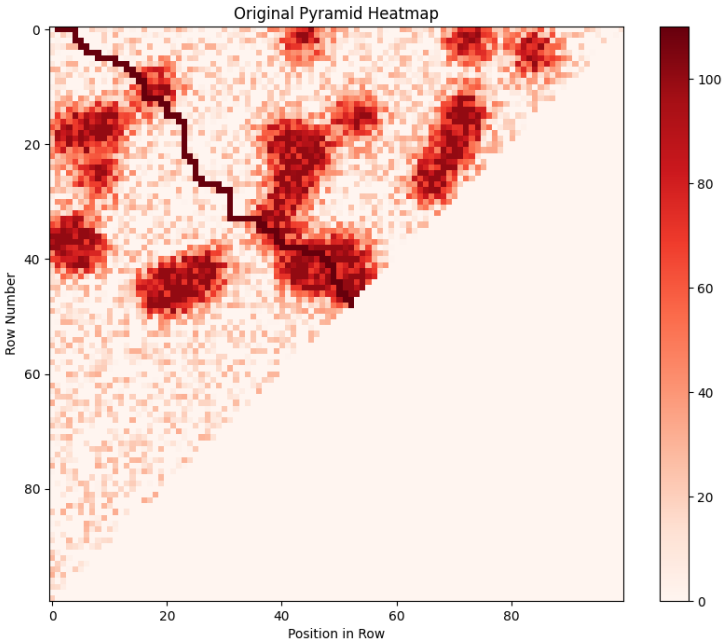
4.2. 三种算法路线对比

1) 贪心算法



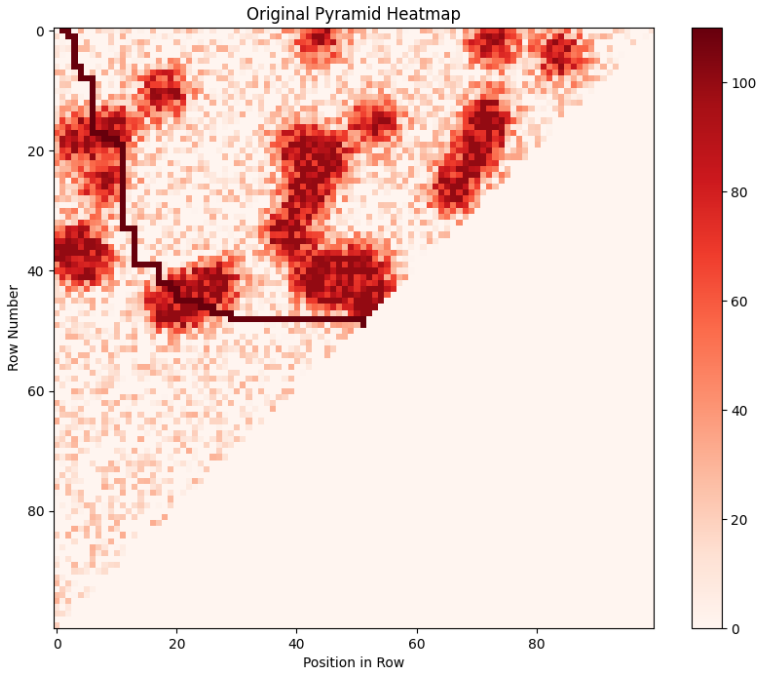
贪心算法最终结果为：4448

2) 探测范围为 2



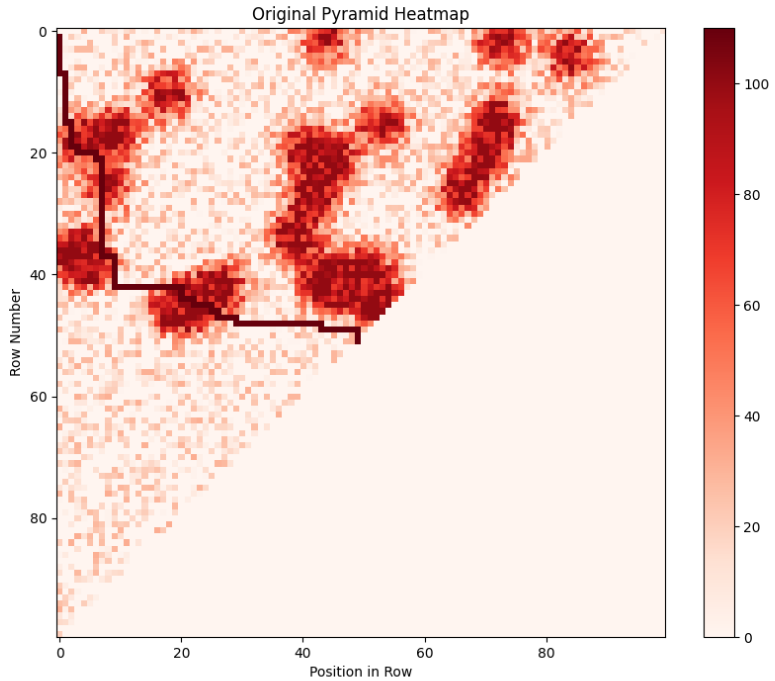
探测范围为 2 最终结果为：4834

3) 探测范围为 4



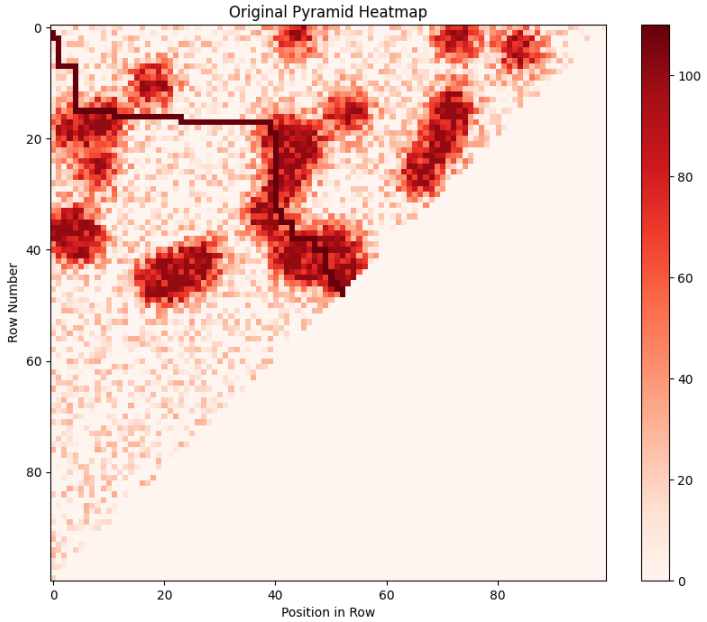
探测范围为 4 最终结果为：4164

4) 探测范围为 12



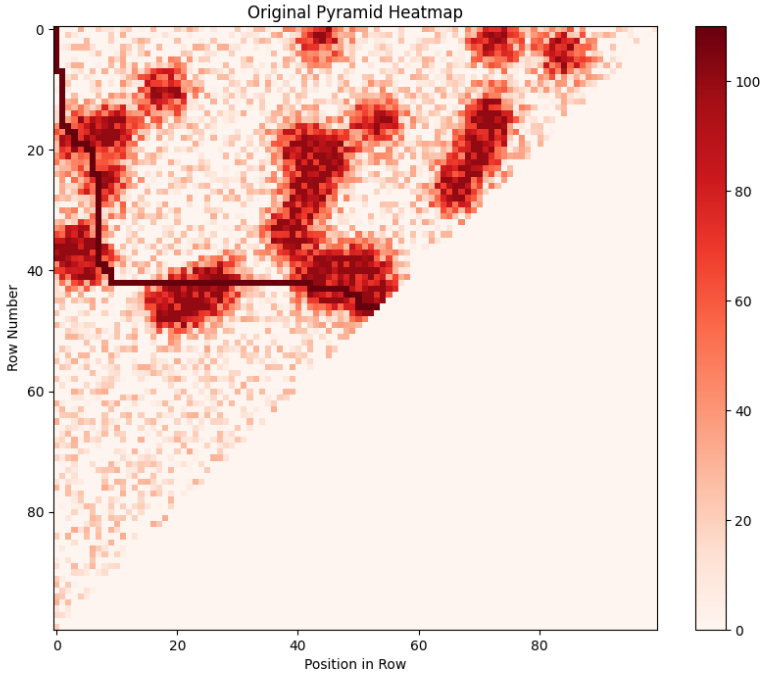
探测范围为 12 最终结果为：5087

5) 探测范围为 42



探测范围为 42 最终结果为：5790

6) 全局动态规划



最优路线的结果为：6220

4.3. 局部最优情况分析

由上图 6 全局动态规划情况可知：最优路线可以描述为先向下，再向右，经过 5 个主要聚集快。

4.3.1.探测范围很小

分析探测范围很小时，路线基本为直线（对角线），主要原因如下

(1) 探测范围小，无法观察到远处（非金字塔正下方）的聚集块；

(2) 生成的金字塔地图在中心轴两边的点分布较均匀。

4.3.2.探测范围较小

分析探测范围增大一些，路线大致正确，但最终结果不够大，主要原因为：

(1) 可以探测到附近的聚集块，故大致方向正确；

(2) 在聚集块中，因为无法检测到更远的聚集块，更倾向于在聚集块内收集足够多的钻石，导致错过下一个聚集块；

(3) 原因（2）可参考探测范围 4/12，在收集倒数第二个聚集块时，过于关注局部信息，错过了最后一个大聚集块。

4.3.3.探测范围较大+路线完全相反

分析探测范围较大，结果较大，路线却完全相反，主要原因为：

(1) 可以探测到多个聚集块，趋向局部更大的聚集块；

(2) 由(1)的原因，错过了最大的聚集块路线；

(3) 因为趋向了局部较大的聚集块，故最终结果并不小。

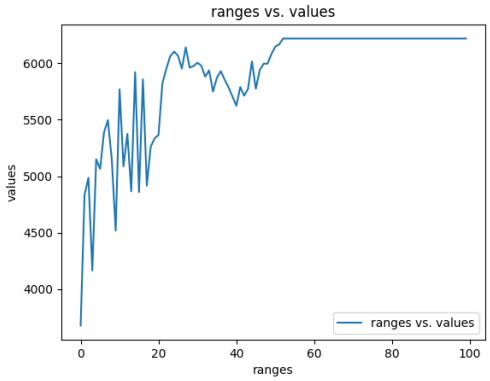
(4) 参考探测范围为 42 的情况

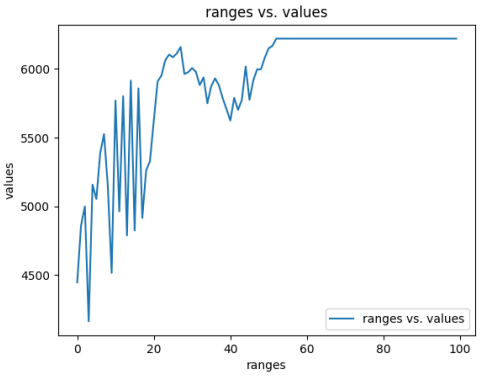
4.4. 最终 val 与探测范围的关系

我们生成了探测范围从 1 到 100 的结果曲线图如下。

因为涉及当左右两步相等时，去向不同可能导致的结果浮动，我们将两种做法都图形化展示了出来。

左图为相等时向左下走，右图为相等时向右下走：





分析发现：

(1) 整体趋向都是随着探测范围的增大，最终结果也增大；

(2) 当探测范围超过金字塔尺寸的一半时，结果/路线慢慢逼近最优解；

(3) 上下波动的部分已在上一节局部最优的分析中给出原因；

(4) 两种操作在某个探测范围上差异较大（如两张图的起点）：

a) 这种情况多发生在探测范围较小时，此时更容易陷入局部最优且不会纠正到局部的聚集块；

b) 我们的地图的左右/上下的轴对称差异很大，导致向左走/向右走的局部地图的总价值存在差异。

5.心得总结

完成了蒙图版钻石矿工算法设计与分析的实验后，我们获得了宝贵的实践经验和深刻的认识。这个实验不仅让我们深入理解了动态规划算法的策略，还让我们掌握了如何避免重复计算，以及如何基于最优子结构递推分解原问题和子问题。

在实验过程中，我们学会了如何从全局动态规划的角度来解决问题，同时也意识到了在实际应用中，由于信息的不完整性，全局动态规划的局限性。这促使我们探索并实现了基于局部动态规划和贪心策略相结合的算法设计方法。通过模拟钻石矿工问题，我们不仅锻炼了编程技能，还提高了解决实际问题的能力。

实验中的蒙图版钻石金字塔问题特别具有挑战性，因为它要求我们在信息不完全的情况下做出最优决策。这让我们意识到，在现实世界的问题中，我们往往需要在有限的信息下做出决策，并且这些决策会对结果产生重大影响。通过这个实验，我们学会了如何在不确定性中寻找最佳解决方案，并且理解了算法设计中的权衡和决策过程。

这个实验不仅提升了我们的算法设计能力，还增强了我们对实际问题解决策略的理解。深刻体会到了理论与实践相结合的重要性，以及在复杂问题面前，如何灵活运用算法知识来寻找解决方案。