



**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](#_Toc184823048)

[1.1. 实验目的 1](#_Toc184823049)

[1.2. 实验内容及要求 1](#_Toc184823050)

[1.3. 实验环境 2](#_Toc184823051)

[2. 算法设计与实现 3](#_Toc184823052)

[2.1. 地图生成算法 3](#_Toc184823053)

[2.1.1. 随机数填充 3](#_Toc184823054)

[2.1.2. 正态分布填充 3](#_Toc184823055)

[2.1.3. 高斯函数填充 3](#_Toc184823056)

[2.2. 贪心算法 4](#_Toc184823057)

[2.3. 全局动态规划算法 5](#_Toc184823058)

[2.4. 蒙图版动态规划 6](#_Toc184823059)

[3. 测试程序与可视化 7](#_Toc184823060)

[4. 算法效率与结果分析 8](#_Toc184823061)

[5. 心得总结 9](#_Toc184823062)

# 实验概述

## 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；

## 实验内容及要求

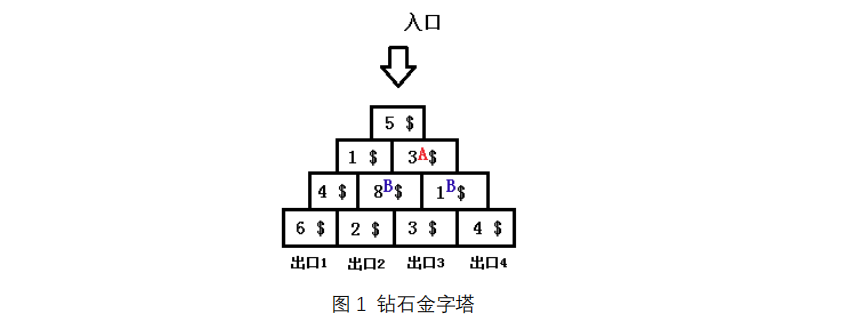
**1. 算法的设计与实现**

* 问题描述

经典的钻石矿工问题描述如下：

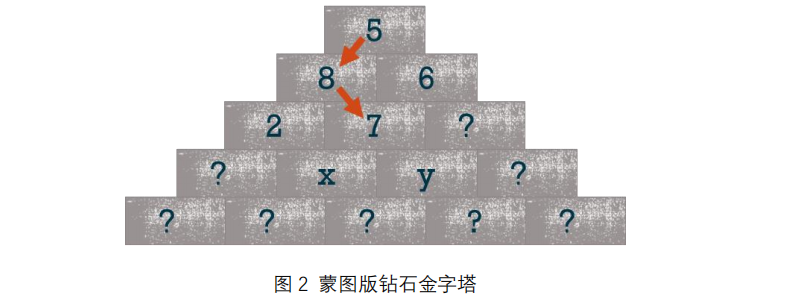
有一座金字塔，金字塔的每块石头上都镶有对应的钻石，钻石可以被取下来，不同的钻石有着不同的价值，例如图 1 所示，你的任务是从金字塔的顶端向金字塔的底端收集钻石，并且尽可能收集价值高的钻石，但是只能从一块砖斜向左下或斜向右下走到另一块砖上，如图 1 从用红色 A 标记的砖走向用蓝色 B 标记的砖上。请找到一个收集最高价值钻石的路线，并给出能够获得的最大钻石总价值？

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。



* 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（，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。



**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。

## 实验环境

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

# 算法设计与实现

## 地图生成算法

### 随机数填充

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

2.

3. // 使用随机数值填写金字塔

4. void pyramid\_random\_fill(int n) {

5.     for (int i = 0; i < n; i++) {

6.         for (int j = 0; j < n - i; j++) {

7.             pyramid[i][j] = rand() % 100;

8.         }

9.     }

10. }

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。

### 正态分布填充

1. // 使用正态模拟的方式填充金字塔

2. void pyramid\_normal\_fill(int n, int k) {

3.     // 定义正态分布参数

4.     double mean = 50.0;    // 均值

5.     double stddev = 10.0;  // 标准差

6.

7.     // 创建一个正态分布的随机数生成器

8.     default\_random\_engine generator;

9.     normal\_distribution<double> distribution(mean, stddev);

10.     uniform\_int\_distribution<int> point\_distribution(0, n - 1);

11.     pyramid.assign(n, vector<int>(n, 0));

12.     for (int i = 0; i < n; ++i) {

13.         for (int j = 0; j < n - i; ++j) {

14.             double value = distribution(generator);

15.             int clampedValue = static\_cast<int>(round(value));

16.             if (clampedValue < 0)

17.                 clampedValue = 0;

18.             if (clampedValue > 100)

19.                 clampedValue = 100;

20.             pyramid[i][j] = clampedValue;

21.         }

22.     }

23.     cout << "Pyramid has been filled with values:" << endl;

24. }

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。

### 高斯函数填充

1. double gaussian(double x, double y, double sigma) {

2.     return exp(-(x \* x + y \* y) / (2 \* sigma \* sigma));

3. }

4.

5. double random\_noise(double scale) {

6.     return (static\_cast<double>(rand()) / RAND\_MAX - 0.5) \* scale;

7. }

8. // 使用高斯函数进行填充（最优填充方案）

9. void pyramid\_gaussian\_fill(int n, int k) {

10.     srand(time(0));

11.     pyramid.resize(n, vector<int>(n, 0));

12.     // 确定聚集块中心点的数量和位置

13.     int numClusters = k;

14.     vector<pair<double, double>> clusterCenters;

15.     // 随机选择k个中心点

16.     for (int i = 0; i < numClusters; ++i) {

17.         double centerX = static\_cast<double>(rand()) / RAND\_MAX \* n / 2;

18.         double centerY = static\_cast<double>(rand()) / RAND\_MAX \* (n - centerX);

19.         clusterCenters.push\_back({centerX, centerY});

20.     }

21.     // 使用高斯分布函数填充聚集块中心点附近的值

22.     for (int i = 0; i < n; ++i) {

23.         for (int j = 0; j < n; ++j) {

24.             int value = 0;

25.             for (const auto& center : clusterCenters) {

26.                 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

27.                 value += static\_cast<int>(g \* 100);   // 范围在0-100之间

28.             }

29.             if (value > 100)

30.                 value = 100;

31.             pyramid[i][j] = value;

32.         }

33.     }

34.     // 引入噪声进行空白位置填充

35.     for (int i = 0; i < n; ++i) {

36.         for (int j = 0; j < n; ++j) {

37.             pyramid[i][j] += static\_cast<int>(random\_noise(65));

38.             if (pyramid[i][j] < 0)

39.                 pyramid[i][j] = 0;

40.             if (pyramid[i][j] > 100)

41.                 pyramid[i][j] = 100;

42.         }

43.     }

44. }

45.

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

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。

## 贪心算法

1. // 贪心算法

2. int miner\_greedy(int n) {

3.     int x = 0, y = 0;

4.     int total\_value = 0;

5.     while (x + y < n - 1) {

6.         if (x + 1 < n - y && y + 1 < n - x) {

7.             if (pyramid[x + 1][y] > pyramid[x][y + 1]) {

8.                 total\_value += pyramid[x + 1][y];

9.                 path.push\_back(make\_pair(x + 1, y));

10.                 x++;

11.             } else {

12.                 total\_value += pyramid[x][y + 1];

13.                 path.push\_back(make\_pair(x, y + 1));

14.                 y++;

15.             }

16.         }

17.     }

18.     cout << "path find successfully" << endl;

19.     return total\_value;

20. }

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。

我们在程序中使用一个path向量进行算法生成的挖掘方向的记录。

## 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

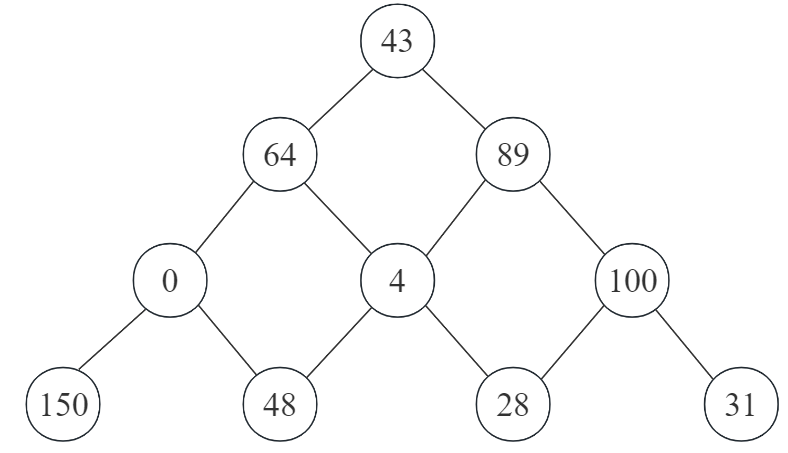
写为状态转移方程即为：

对于边界情况：

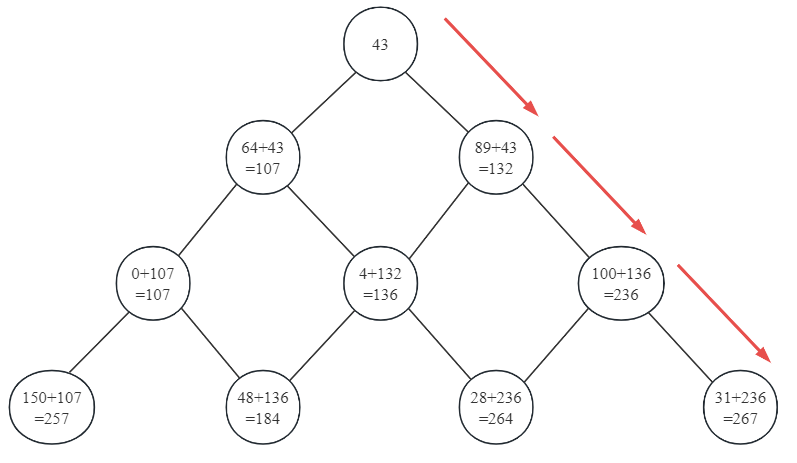
1. 当矿工处于起点时，values的值即为起点的值；
2. 当矿工处于金字塔的两条边时，values只能由上一个点（确定）+该点的值。

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：

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。

## 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.

41.     while (current\_i + current\_j < n) {

42.         int left\_value = -1;

43.         int right\_value = -1;

44.         // 先处理右边的局部最优

45.         if (current\_i < n && current\_j + 1 < n) {

46.             right\_value = dp\_x(current\_i, current\_j + 1, n, x - 1);

47.         }

48.         // 再处理左边的局部最优

49.         if (current\_i + 1 < n && current\_j < n) {

50.             left\_value = dp\_x(current\_i + 1, current\_j, n, x - 1);

51.         }

52.

53.         // 说明到达金字塔底部

54.         if (right\_value == -1 && left\_value == -1)

55.             break;

56.

57.         // 右边的局部最优比左边大，向右下方走一步

58.         if (right\_value >= left\_value) {

59.             current\_j += 1;                          // 向右走一步

60.             ret += pyramid[current\_i][current\_j];    // 累加这一步的价值

61.             path.push\_back({current\_i, current\_j});  // 记录这一步的路径

62.         } else {

63.             current\_i += 1;                          // 向左走一步

64.             ret += pyramid[current\_i][current\_j];    // 累加这一步的价值

65.             path.push\_back({current\_i, current\_j});  // 记录这一步的路径

66.         }

67.     }

68.

69.     return ret;

70. }

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：
* 全局动态规划：

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；

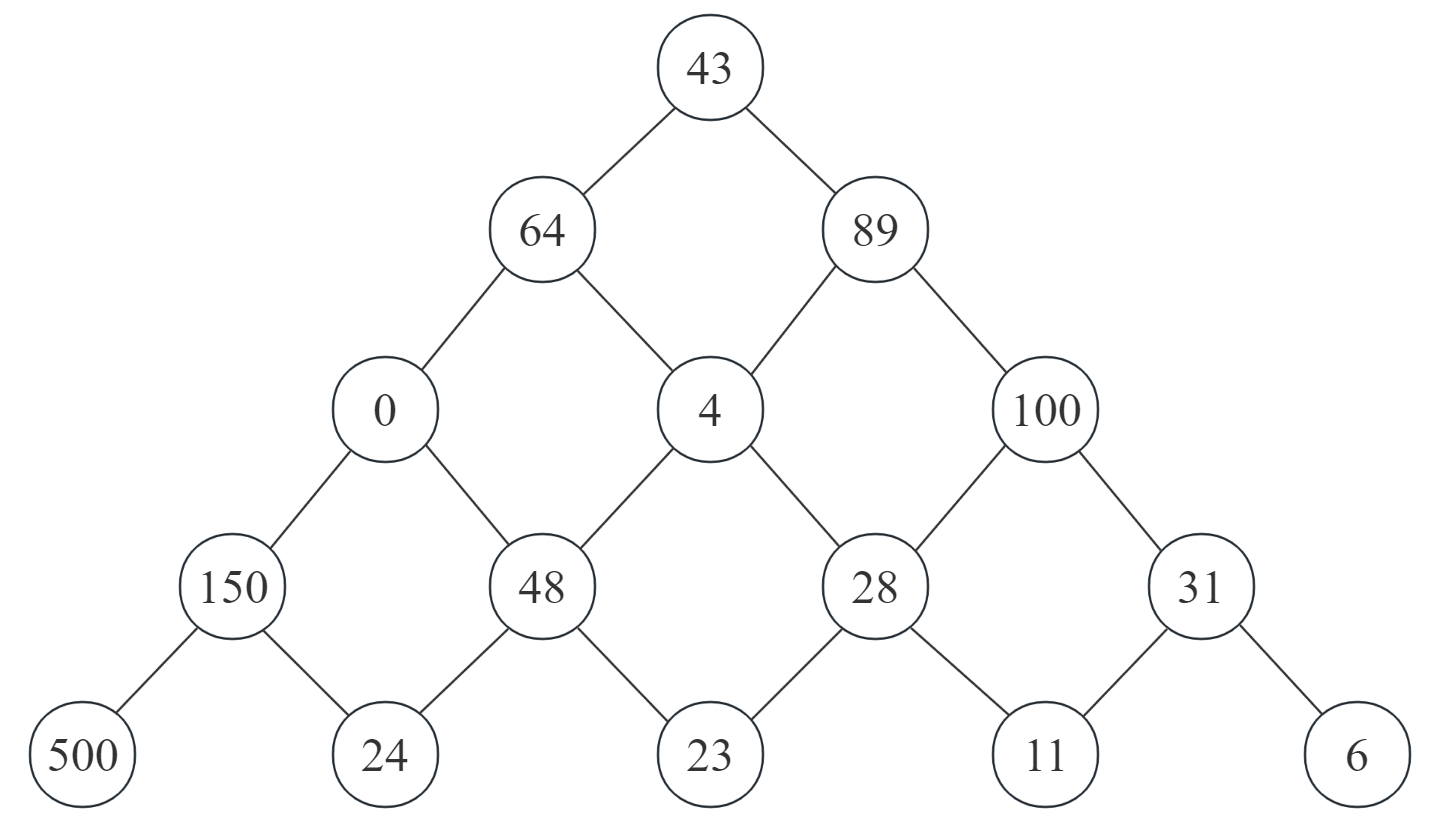
而因为我们的x不再是无限大，所以只能看到当前层数层的值。而对于下一步（即下一层），我们只能看到距离为的范围。

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（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。

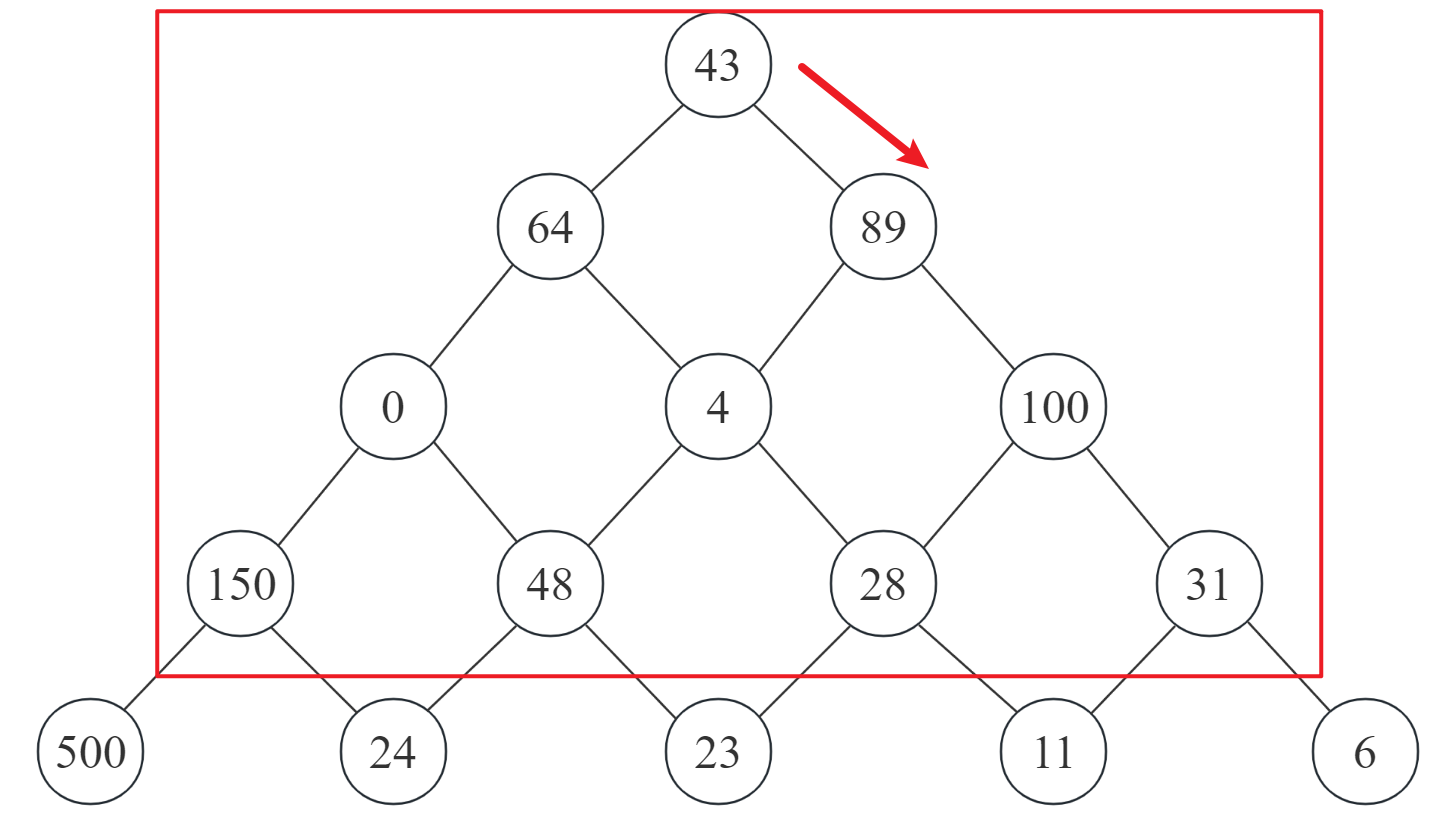
举例说明：

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：

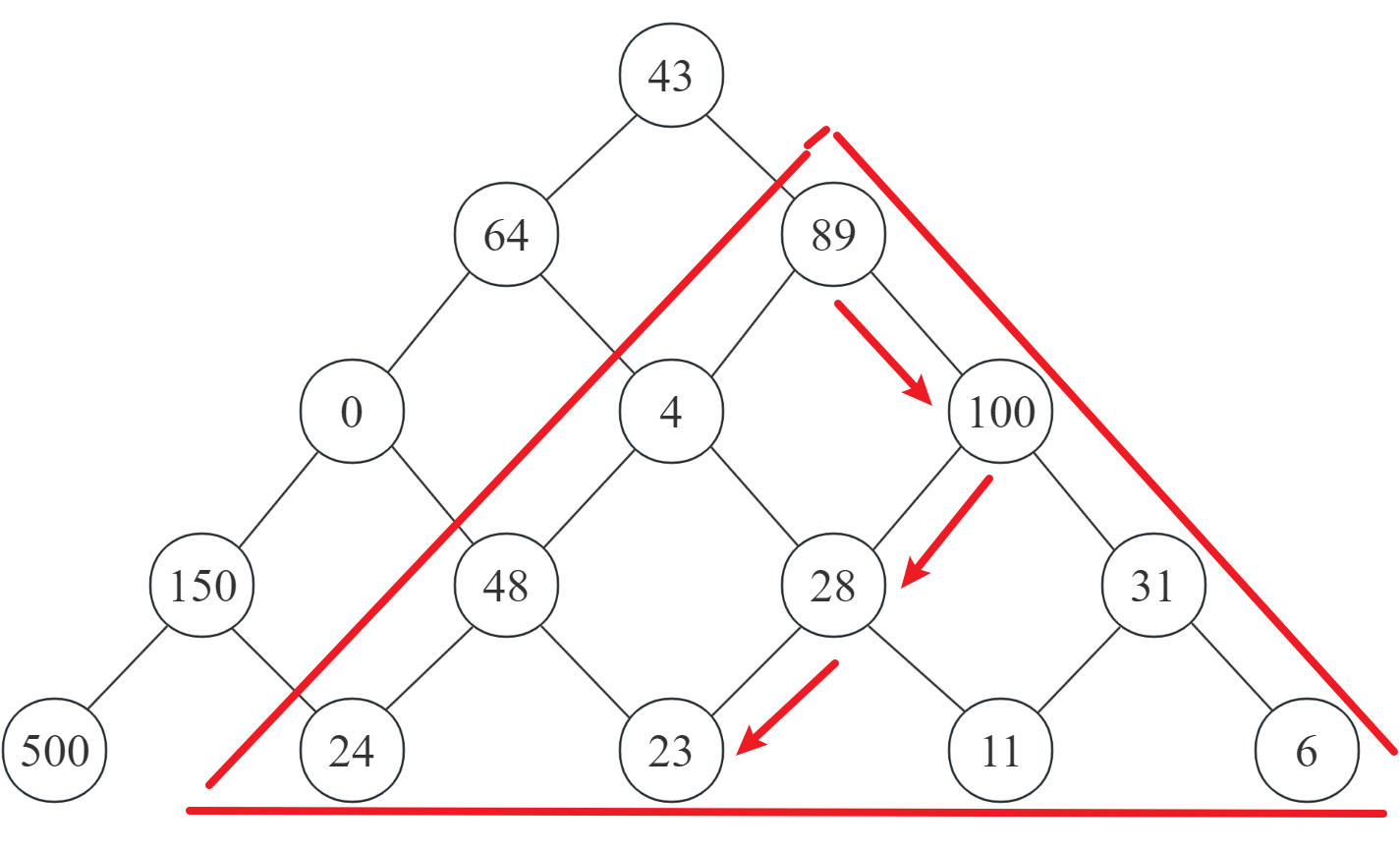


很明显，最左下角的路线一定是最大的，因为500远大于其他点的值。

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：（左右两步的各自最大值），得到新的路线：

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。

# 测试程序与可视化

## 三种算法结果输出

## 三种算法路线可视化

## 不同探测度结果曲线可视化

## 探究局部最优原因可视化

# 算法效率与结果分析

# 心得总结