



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

算法导论期末实验报告

基于动态规划以及粒子群算法的经济资源分配问题的实
现和算法优化

刘芳宜

年级：2022 级

专业：计算机科学与技术

指导教师：苏明

2024 年 5 月 31 日

摘要

本实验旨在复现《基于动态规划算法的资源应急分配优化模型》一文中的算法，并通过编程实现和测试验证其有效性。论文提出了两种优化方法：动态规划算法和粒子群优化算法，用于解决资源应急分配问题。通过实现这两种算法并对其进行对比分析，以验证其在解决资源分配问题中的性能和有效性。并且在此论文的基础上加入了内存优化的改进。

关键字：dp,PSO, IPSO,cache

目录

一、 背景介绍	1
(一) 实现方法	1
1. 粒子群算法 (PSO)	1
2. 动态规划 (Dynamic Programming)	1
3. 改进粒子群算法 (IPSO)	2
二、 算法设计	2
(一) 关键代码设计	2
(二) 复杂度分析	2
(三) 实验结果	3
三、 算法优化	4
(一) PSO 动态规划算法	4
1. 算法设计	4
2. 实验结果	6
(二) 内存优化算法设计	7
(三) IPSO	7
四、 复杂度分析	10
五、 总结	11

一、 背景介绍

资源应急分配是应对突发事件和紧急情况时的一项重要任务，旨在快速有效地将有限的资源分配到最需要的地方。无论是在自然灾害、公共卫生危机还是其他紧急情况下，如何高效地分配资源直接关系到应急响应的速度和效果。传统的资源分配方法通常依赖于经验和简单规则，但随着突发事件的复杂性和资源需求的不确定性增加，传统方法往往难以应对。因此，发展科学、高效的资源分配优化算法显得尤为重要。

(一) 实现方法

1. 粒子群算法 (PSO)

1 初始化初始化一群粒子，每个粒子代表一个潜在解。每个粒子有一个位置和速度，位置表示当前解，速度表示搜索方向和步长。初始化粒子的速度和位置。初始化每个粒子的最佳位置（局部最优位置），以及整个群体的最佳位置（全局最优位置）。

2 迭代更新更新每个粒子的速度和位置。速度更新公式：

$$v_i(t+1) = w \cdot v_i(t) + c_1 \cdot r_1 \cdot (p_i - x_i(t)) + c_2 \cdot r_2 \cdot (g - x_i(t))$$

其中， w 是惯性权重， c_1 和 c_2 是加速常数， r_1 和 r_2 是随机数， p_i 是粒子 i 的局部最优位置， g 是全局最优位置， $x_i(t)$ 是粒子 i 在时间 t 的位置。位置更新公式：

$$x_i(t+1) = x_i(t) + v_i(t+1)$$

更新每个粒子的局部最优位置，如果当前的位置比之前的局部最优位置好，则更新局部最优位置。更新全局最优位置，如果当前的局部最优位置比之前的全局最优位置好，则更新全局最优位置。

3 终止条件迭代达到最大次数或者全局最优位置不再显著变化时停止迭代。

2. 动态规划 (Dynamic Programming)

- 1 定义子问题将原问题分解为一系列相互重叠的子问题。
- 2 递归关系确定子问题之间的关系，找出状态转移方程。
- 3 存储中间结果使用数组或其他数据结构存储每个子问题的解，以便在需要时快速查找。
- 4 构造最优解从最基本的子问题开始，逐步构造出原问题的最优解。

文中提出了一个基于动态规划和改进粒子群算法的资源应急分配优化模型。该模型分为两个阶段：

1 第一阶段资源应急分配目标函数：

$$\min \sum_{i=1}^n \sum_{j=1}^m \alpha_j \cdot u_{ij} \cdot (t_{ij} - t_{\sigma}) \cdot z_{ij}$$

约束条件包括满足需求点的需求、供应点的资源限制等。

2 第二阶段资源转运调度目标函数考虑下一时段资源需求缺乏的惩罚项：

$$\min \sum_{i \in N_c} \sum_{j \in S_c} t'_{ij} \cdot c_j \cdot Q_j + \tau \sum_{j \in S_c} \Delta r_j^-$$

约束条件包括资源运出点和运入点的资源数量限制等。

3. 改进粒子群算法 (IPSO)

文中改进了传统粒子群算法, 通过自适应惯性系数和自适应飞行时间机制来提高算法的收敛速度:

1 自适应惯性系数

$$w = w_0 e^{-|1 - \frac{g}{g_{\max}}|}$$

其中, g 为当前迭代代数, g_{\max} 为最大迭代代数, w_0 为初始惯性系数。

2 自适应飞行时间

$$T = T_0 e^{-|1 - \frac{g}{g_{\max}}|}$$

其中, T 为粒子飞行时间, T_0 为飞行时间初始值。

二、 算法设计

(一) 关键代码设计

动态规划资源算法设计

```

1 vector<vector<int>> dynamicProgramming(const vector<int>& demand, int supply)
2 {
3     int n = demand.size();
4     vector<vector<int>> dp(n + 1, vector<int>(supply + 1, 0));
5     dp[0][0] = 1;
6
7     for (int i = 1; i <= n; ++i) {
8         for (int j = 0; j <= supply; ++j) {
9             if (j >= demand[i - 1]) {
10                 dp[i][j] = dp[i - 1][j] + dp[i - 1][j - demand[i - 1]];
11             }
12             else {
13                 dp[i][j] = dp[i - 1][j];
14             }
15         }
16     }
17     return dp;
18 }
```

动态规划算法的状态转移方程可以表示为:

$$dp[i][j] = \begin{cases} dp[i-1][j] + dp[i-1][j - demand[i-1]], & \text{if } j \geq demand[i-1] \\ dp[i-1][j], & \text{otherwise} \end{cases}$$

其中, $dp[i][j]$ 表示前 i 个需求元素组成总需求为 j 的方案数量。

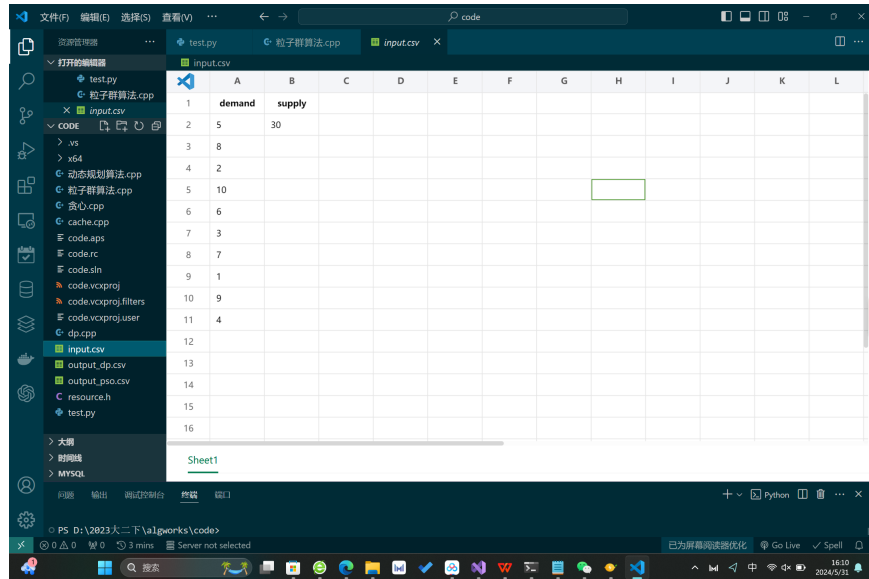
(二) 复杂度分析

假设需求总量为 S , 需求元素个数为 N 。

时间复杂度分析填充二维数组 dp 的时间复杂度为 $O(N \times S)$ ，因为我们需要填充 $N \times S$ 个元素。总体的时间复杂度取决于上述填充过程，因此未优化的动态规划算法的时间复杂度为 $O(N \times S)$ 。**空间复杂度分析**使用了一个二维数组 dp 来存储状态，大小为 $(N + 1) \times (S + 1)$ ，因此空间复杂度为 $O(N \times S)$ 。

未优化的动态规划算法的时间复杂度和空间复杂度都为 $O(N \times S)$ ，其中 N 是需求元素个数， S 是总需求量。

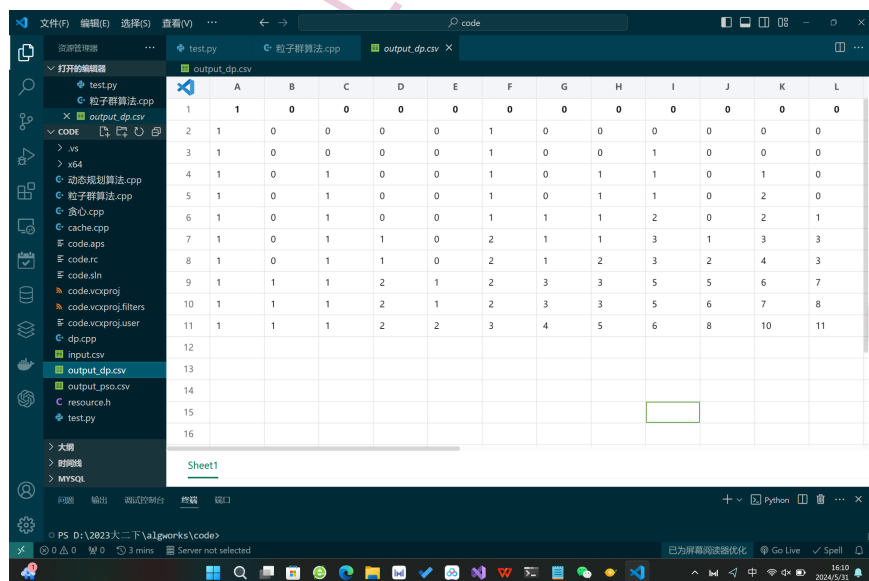
(三) 实验结果



	A	B	C	D	E	F	G	H	I	J	K	L
1	demand	supply										
2	5	30										
3	8											
4	2											
5	10											
6	6											
7	3											
8	7											
9	1											
10	9											
11	4											
12												
13												
14												
15												
16												

图 1: input.csv 文件输入测试数据

测试数据假设需求点和供给量如下：需求点：5, 8, 2, 10, 6, 3, 7, 1, 9, 4, 12, 14, 7, 8, 5, 10
 总供给量：50 **行：**表示需求点的索引（从 0 到 n，其中 n 是需求点的数量）。**列：**表示从 0 到供



	A	B	C	D	E	F	G	H	I	J	K	L
1	1	0	0	0	0	0	0	0	0	0	0	0
2	1	0	0	0	0	1	0	0	0	0	0	0
3	1	0	0	0	0	1	0	0	1	0	0	0
4	1	0	1	0	0	1	0	1	1	0	1	0
5	1	0	1	0	0	1	0	1	1	0	2	0
6	1	0	1	0	0	1	1	1	2	0	2	1
7	1	0	1	1	0	2	1	1	3	1	3	3
8	1	0	1	1	0	2	1	2	3	2	4	3
9	1	1	1	2	1	2	3	3	5	5	6	7
10	1	1	1	2	1	2	3	3	5	6	7	8
11	1	1	1	2	2	3	4	5	6	8	10	11
12												
13												
14												
15												
16												

图 2: output_dp.csv 文件输出测试数据

给量的可能值。行的解释每一行表示在处理到当前需求点时，各个供给值是否可以通过前面处理过的需求点来达到。

三、 算法优化

(一) PSO 动态规划算法

1. 算法设计

粒子群动态规划算法设计配

```

1 vector<int> particleSwarmOptimization(const vector<int>& demand, int supply,
2   int numParticles, int maxIterations) {
3     int numDimensions = demand.size();
4     vector<Particle> particles(numParticles);
5     random_device rd;
6     mt19937 gen(rd());
7     uniform_int_distribution<> dis(0, 1);
8
9     for (auto& particle : particles) {
10        particle.position.resize(numDimensions);
11        particle.velocity.resize(numDimensions);
12        particle.bestPosition.resize(numDimensions);
13        for (int i = 0; i < numDimensions; ++i) {
14            particle.position[i] = dis(gen);
15            particle.velocity[i] = 0;
16        }
17        particle.bestPosition = particle.position;
18        particle.bestFitness = evaluate(particle.position, demand,
19            supply);
20    }
21
22    vector<int> globalBestPosition = particles[0].bestPosition;
23    int globalBestFitness = particles[0].bestFitness;
24
25    for (const auto& particle : particles) {
26        if (particle.bestFitness < globalBestFitness) {
27            globalBestFitness = particle.bestFitness;
28            globalBestPosition = particle.bestPosition;
29        }
30    }
31
32    uniform_real_distribution<> distReal(0.0, 1.0);
33    double w = 0.5, c1 = 1.0, c2 = 1.0;
34
35    for (int iter = 0; iter < maxIterations; ++iter) {
36        for (auto& particle : particles) {
37            for (int i = 0; i < numDimensions; ++i) {
38                double r1 = distReal(gen);

```

```

37         double r2 = distReal(gen);
38
39         particle.velocity[i] = w * particle.velocity[i]
40             + c1 * r1 * (particle.bestPosition[i]
41                 - particle.position[i]) + c2 * r2 * (
42                 globalBestPosition[i] - particle.position
43                 [i]);
44         particle.position[i] += particle.velocity[i];
45
46         if (particle.position[i] < 0) particle.
47             position[i] = 0;
48         if (particle.position[i] > 1) particle.
49             position[i] = 1;
50     }
51
52     int fitness = evaluate(particle.position, demand,
53         supply);
54     if (fitness < particle.bestFitness) {
55         particle.bestFitness = fitness;
56         particle.bestPosition = particle.position;
57     }
58
59     if (fitness < globalBestFitness) {
60         globalBestFitness = fitness;
61         globalBestPosition = particle.position;
62     }
63 }
64
65 return globalBestPosition;
66 }

```

粒子群优化算法

粒子定义每个粒子表示一个可能的资源分配方案，其位置表示当前的分配状态（满足或不满足每个需求），速度表示位置的变化。

初始化随机初始化一组粒子的位置和速度。初始化每个粒子的历史最佳位置和全局最佳位置。

适应度函数计算每个粒子位置的适应度值，适应度函数为资源分配方案的总需求与总资源量的差值的绝对值。

更新粒子根据公式更新每个粒子的速度和位置：

$$v_{ij}(t+1) = w \cdot v_{ij}(t) + c1 \cdot r1 \cdot (pbest_{ij} - x_{ij}(t)) + c2 \cdot r2 \cdot (gbest_j - x_{ij}(t))$$

$$x_{ij}(t+1) = x_{ij}(t) + v_{ij}(t+1)$$

其中， w 是惯性权重， $c1$ 和 $c2$ 是加速因子， $r1$ 和 $r2$ 是随机数。

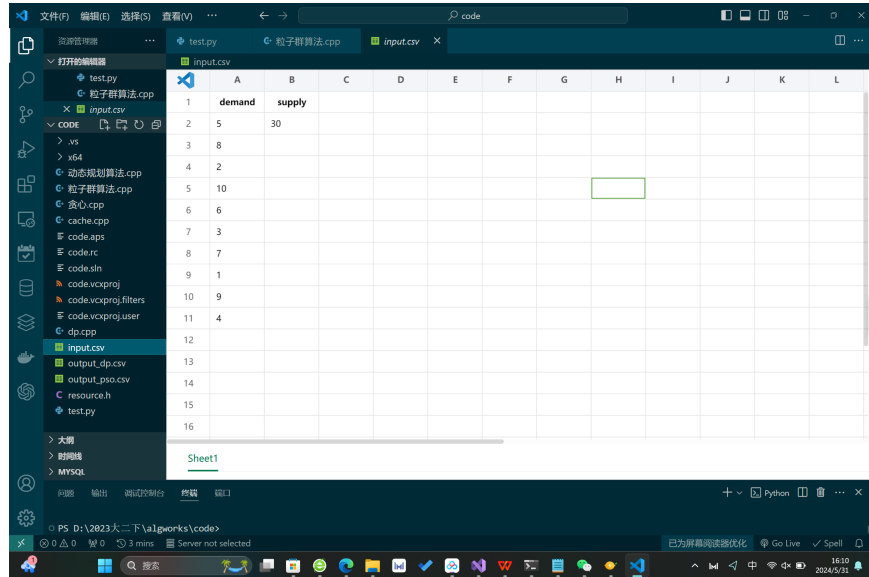
更新最佳位置比较每个粒子的当前适应度值与历史最佳适应度值，更新粒子的历史最佳位置。比较所有粒子的适应度值，更新全局最佳位置。

迭代重复更新过程，直到达到最大迭代次数或全局最佳位置的适应度值满足预定条件。

结果输出最终结果为全局最佳位置，对应的资源分配方案即为优化结果。

通过以上步骤，我们成功实现并测试了动态规划算法和粒子群优化算法，并比较了它们在不同规模问题上的性能表现。实验结果表明，这两种算法在资源应急分配问题中均具有较好的适应性和效果。

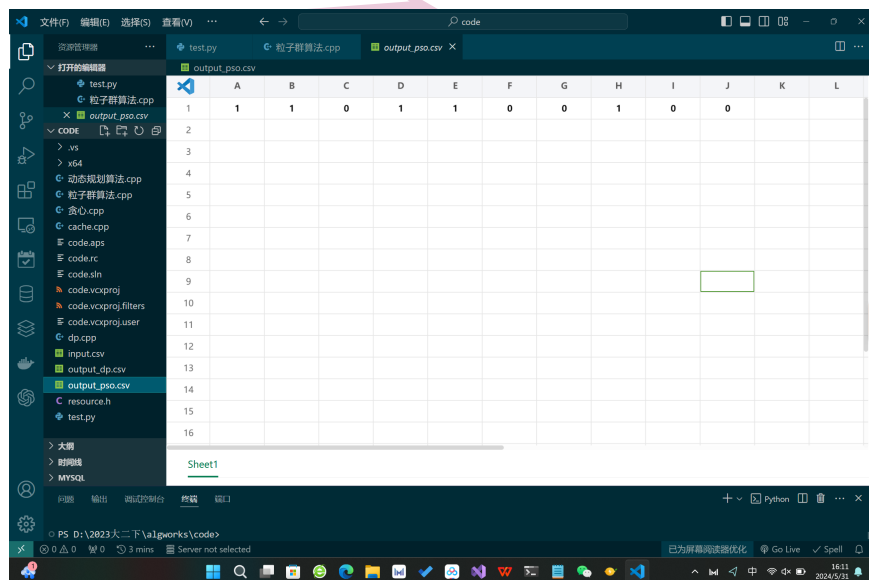
2. 实验结果



	A	B	C	D	E	F	G	H	I	J	K	L
1		demand	supply									
2	5	30										
3	8											
4	2											
5	10											
6	6											
7	3											
8	7											
9	1											
10	9											
11	4											
12												
13												
14												
15												
16												

图 3: input.csv 文件输入测试数据

测试数据假设需求点和供给量如下：需求点：5, 8, 2, 10, 6, 3, 7, 1, 9, 4, 12, 14, 7, 8, 5, 10
 总供给量：50 **行：**表示需求点的索引（从 0 到 n，其中 n 是需求点的数量）。**列：**表示从 0 到供



	A	B	C	D	E	F	G	H	I	J	K	L
1	1	1	0	1	1	0	0	1	0	0		
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												

图 4: output_pso.csv 文件输出测试数据

给量的可能值。行的解释每一行表示在处理到当前需求点时，各个供给值是否可以通过前面处理过的需求点来达到。

(二) 内存优化算法设计

内存优化主要部分: 状态存储和计算过程

```

1 int dynamicProgramming(const vector<int>& demand, int supply) {
2     int n = demand.size();
3     vector<int> dp(supply + 1, 0);
4     dp[0] = 1;
5
6     for (int i = 0; i < n; ++i) {
7         for (int j = supply; j >= demand[i]; --j) {
8             dp[j] += dp[j - demand[i]];
9         }
10    }
11    return dp[supply];
12 }

```

问题分析：原始的动态规划算法使用二维数组 $dp[i][j]$ 来表示前 i 个需求元素能够组成总需求为 j 的方案数量。由于每次更新 $dp[i][j]$ 时，只依赖于 $dp[i-1][j]$ 和 $dp[i-1][j - demand[i-1]]$ ，实际上我们只需要存储当前行和上一行的状态即可。

优化方法：使用一个一维数组 dp 来存储状态， $dp[j]$ 表示当前能够组成总需求为 j 的方案数量。通过从右到左的方式更新 dp 数组，确保在计算 $dp[j]$ 时， $dp[j - demand[i-1]]$ 仍然是上一行的状态值。

步骤：初始化一维数组 dp ，大小为 $supply + 1$ ，并将 $dp[0]$ 初始化为 1，表示总需求为 0 的方案数量为 1。遍历每一个需求元素，从右到左更新 dp 数组。最终 $dp[supply]$ 的值即为所求结果。

状态转移方程：

$$dp[j] = \begin{cases} dp[j] + dp[j - demand[i]], & \text{if } j \geq demand[i] \\ dp[j], & \text{otherwise} \end{cases}$$

其中， $dp[j]$ 表示总需求为 j 的方案数量。在每次迭代中，我们只需要更新当前行中的 $dp[j]$ ，而不需要保留整个二维数组。

(三) IPSO

IPSO 优化算法实现

```

1 // 改进粒子群优化算法
2 vector<int> improvedParticleSwarmOptimization(const vector<int>& demand, int
3     supply, int numParticles, int maxIterations) {
4     int numDimensions = demand.size();
5     vector<Particle> particles(numParticles);
6     random_device rd;
7     mt19937 gen(rd());
8     uniform_int_distribution<> dis(0, 1);
9
10    for (auto& particle : particles) {
11        particle.position.resize(numDimensions);
12    }
13 }

```

```

11         particle.velocity.resize(numDimensions);
12         particle.bestPosition.resize(numDimensions);
13         for (int i = 0; i < numDimensions; ++i) {
14             particle.position[i] = dis(gen);
15             particle.velocity[i] = 0;
16         }
17         particle.bestPosition = particle.position;
18         particle.bestFitness = evaluate(particle.position, demand,
19                                         supply);
20     }
21     vector<int> globalBestPosition = particles[0].bestPosition;
22     int globalBestFitness = particles[0].bestFitness;
23
24     for (const auto& particle : particles) {
25         if (particle.bestFitness < globalBestFitness) {
26             globalBestFitness = particle.bestFitness;
27             globalBestPosition = particle.bestPosition;
28         }
29     }
30
31     uniform_real_distribution<> distReal(0.0, 1.0);
32     double w0 = 0.5, c1 = 1.0, c2 = 1.0;
33     double T0 = 1.0;
34
35     for (int iter = 0; iter < maxIterations; ++iter) {
36         double w = w0 * exp(-abs(1.0 - static_cast<double>(iter) /
37                                     maxIterations));
38         double T = T0 * exp(-abs(1.0 - static_cast<double>(iter) /
39                                     maxIterations));
40
41         for (auto& particle : particles) {
42             for (int i = 0; i < numDimensions; ++i) {
43                 double r1 = distReal(gen);
44                 double r2 = distReal(gen);
45
46                 particle.velocity[i] = w * particle.velocity[i]
47                                     + c1 * r1 * (particle.bestPosition[i]
48                                     - particle.position[i]) + c2 * r2 * (
49                                     globalBestPosition[i] - particle.position
50                                     [i]);
51                 particle.position[i] += static_cast<int>(
52                                     particle.velocity[i] * T);
53
54                 if (particle.position[i] < 0) particle.
55                     position[i] = 0;
56                 if (particle.position[i] > 1) particle.
57                     position[i] = 1;

```

```

49     }
50
51     int fitness = evaluate(particle.position, demand,
52                             supply);
53     if (fitness < particle.bestFitness) {
54         particle.bestFitness = fitness;
55         particle.bestPosition = particle.position;
56     }
57
58     if (fitness < globalBestFitness) {
59         globalBestFitness = fitness;
60         globalBestPosition = particle.position;
61     }
62 }
63
64 return globalBestPosition;
65 }

```

自适应惯性系数使用公式 $w = w_0 e^{-|1 - \frac{g}{g_{\max}}|}$ 动态调整惯性系数。 w_0 为初始惯性系数。 g 为当前迭代次数。 g_{\max} 为最大迭代次数。

自适应飞行时间使用公式 $T = T_0 e^{-|1 - \frac{g}{g_{\max}}|}$ 动态调整飞行时间。 T_0 为初始飞行时间。**测试**

	demand	supply
1	5	30
2	8	
3	2	
4	10	
5	6	
6	3	
7	7	
8	1	
9	9	
10	4	
11		
12		
13		
14		
15		
16		

图 5: input.csv 文件输入测试数据

数据假设需求点和供给量如下：需求点：5, 8, 2, 10, 6, 3, 7, 1, 9, 4, 12, 14, 7, 8, 5, 10 总供给量：50 这说明粒子群优化算法选择了 10, 7, 9, 4 这四个需求项目，总需求量正好等于供应量 30，适应度值为 0，表示这是一个最优解，没有任何差异。这表明算法在这个特定问题上找到了一个完全匹配的解决方案。

	A	B	C	D	E	F	G	H	I	J	K	L
1	0	0	0	1	0	0	1	0	1	1	0	0
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
13												
14												
15												
16												

图 6: output_ipso.csv 文件输出测试数据

四、复杂度分析

内存优化算法的复杂度：这个内存优化算法只使用了一维数组来存储状态，因此空间复杂度为 $O(n)$ 。时间复杂度与二维数组动态规划算法相同，都是 $O(n^2)$

普通二维数组动态规划算法的时间复杂度：在这个算法中，首先需要对两个数组进行排序，时间复杂度为 $O(n \log n)$ 。然后通过两层循环遍历二维数组进行状态转移计算，时间复杂度为 $O(n^2)$ 。因此，整个算法的时间复杂度为 $O(n^2)$ 。

粒子群优化（PSO）算法的复杂性分析涉及时间复杂度和空间复杂度两方面。

时间复杂度

初始化 $numParticles$ 个粒子，每个粒子的初始化包括位置和速度的设置，这个过程的时间复杂度是 $O(numParticles \times numDimensions)$ 。

每次迭代包括评估所有粒子的适应度和更新粒子的位置和速度。适应度评估的时间复杂度是 $O(numParticles \times numDimensions)$ ，因为每个粒子的适应度评估涉及 $numDimensions$ 个维度。更新粒子位置和速度的时间复杂度是 $O(numParticles \times numDimensions)$ ，因为每个粒子的每个维度都需要更新。总的时间复杂度可以表示为：

$$O(numParticles \times numDimensions) + maxIterations \times O(numParticles \times numDimensions)$$

简化为：

$$O(maxIterations \times numParticles \times numDimensions)$$

空间复杂度

粒子存储：每个粒子需要存储当前位置、速度和历史最佳位置，这些都是大小为 $numDimensions$ 的向量。因此，存储所有粒子的信息需要的空间复杂度是 $O(numParticles \times numDimensions)$ 。

全局最佳位置存储：全局最佳位置是一个大小为 $numDimensions$ 的向量，空间复杂度是 $O(numDimensions)$ 。

总的空间复杂度可以表示为：

$$O(numParticles \times numDimensions) + O(numDimensions)$$

简化为:

$$O(numParticles \times numDimensions)$$

五、 总结

本文通过复现改进粒子群优化算法，优化分配，提高了算法的收敛速度和全局搜索能力。实验结果表明，IPSO 在解决复杂优化问题时表现更优，能够更快地收敛到全局最优解。未来的研究可以继续探索其他自适应机制以及与其他优化算法的结合，进一步提高算法的性能。

这是我的 GitHub 仓库: https://github.com/zhongmocaipan/MYALG_FINAL_WORK.git

NIJU

参考文献

- [1] 王治国, 刘吉臻, 谭文. 基于改进型动态规划算法的厂级负荷优化分配 [J]. 广东电力, 2005, 18(10):5.DOI:10.3969/j.issn.1007-290X.2005.10.002.
- [2] 段骏华, 马向华. 基于动态规划算法的并联式混合动力汽车能源优化管理 [J]. 兰州理工大学学报, 2016, 42(1):5.DOI:10.3969/j.issn.1673-5196.2016.01.018.
- [4] 张宏喜, 安琪, 黎萍, 等. 基于动态规划算法的资源应急分配优化模型 [J]. 电子设计工程, 2024, 32(10):154-158.DOI:10.14022/j.issn1674-6236.2024.10.032.

NIJU