清華大學化学工程系
Department of Chemical Engineering，Tsinghua University

清华大学防疫物资网点布局优化研究

目录

01 研究背景

01 研究背景

问题描述

- 在疫情常态化的大背景下，为了防止防疫物资不足的情况发生，学校设立数个防疫物资网点方便师生购置防疫物资。
- 现在需要建设几个物资网点，需要综合考虑师生便利程度、建设成本与校医院物资配送成本

假设

- 每个人都需要购置防疫物资
- 每个站点防疫物资完全相同
- 每个人倾向于选择距离更近的网点采购防疫物资
- 每个站点的防疫物资只有校医院提供且不能流通

02 模型建立

02 模型建立


图例
4号线 圆明园站
4号线 北大东门站
15号线 清华东路西口站
13号线 五道口站

结构化


图例
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
4号线 圆明园站
4号线 北大东门站

02 模型建立

➢ 优化目标：

- 校园内居民的困难程度 $\boldsymbol{H}(\vec{n},\ \vec{m})$
- 防疫物资站点建设的总成本 $\boldsymbol{S}(\vec{n},\ \vec{m})$
- 防疫物资日常配送的总成本 $\boldsymbol{D}(\vec{n},\ \vec{m})$

➢ 主要常量：

- 校内居民点 $\vec{\boldsymbol{n}} = \{n_i\},\ i = 1, 2, \ldots, 33$

➢ 主要变量：

- 校内防疫物资点 $\vec{\boldsymbol{m}} = \{m_i\},\ i = 1, 2, \ldots, \mathrm{n}$

共 33 个居民点，记为 $n_i = (x_i,\ y_i,\ p_i), \qquad i = 1,\ 2,\ \cdots,\ 33$

共 n 个物资点，记为 $m_i = (\widetilde{x}_i,\ \widetilde{y}_i,\ q_i), \qquad i = 1,\ 2,\ \cdots,\ \mathrm{n}$

校医院记为 $m_h = (\widetilde{x}_h,\ \widetilde{y}_h)$

师生觉得到物资点还算“便利”的距离最大值记为 l_{max}

物资点到校医院距离的归一化系数记为 l_{min}

$C_s,\ C_d,\ k_s,\ k_d$ 为经验系数

02 模型建立

$$\textbf{min} \quad \begin{bmatrix} u & v & w \end{bmatrix} \begin{bmatrix} \boldsymbol{H} \\ \boldsymbol{S} \\ \boldsymbol{D} \end{bmatrix} (\vec{n},\ \vec{m})$$

$$\textbf{s.t.} \quad \vec{n} = (n_1,\ n_2,\ \cdots,\ n_{33}), \qquad n_i = (x_i,\ y_i,\ p_i)$$

$$\vec{m} = (m_1,\ m_2,\ \cdots,\ m_5), \qquad m_i = (\widetilde{x}_i,\ \widetilde{y}_i), \quad m_i \in \text{Restricting Area}$$

$$\boldsymbol{H}(\vec{n},\ \vec{m}) = \sum_{i=1}^{33} \boldsymbol{H}(n_i)$$

$$\boldsymbol{H}(n_i) = \begin{cases} p_i h_{min}, & |n_i - m^*_{n_i}| \leqslant l_{max} \\ p_i h_{min} \exp\left(\frac{|n_i - m^*_{n_i}|}{l_{max}}\right), & |n_i - m^*_{n_i}| > l_{max} \end{cases}, \qquad m^*_{n_i} = \underset{m_i}{\textbf{arg min}}\ |n_i - m_i|$$

$$\boldsymbol{S}(\vec{n},\ \vec{m}) = \sum_{i=1}^{\mathrm{n}} \boldsymbol{S}(m_i)$$

$$\boldsymbol{S}(m_i) = C_s + k_s \times \left(\frac{q_i}{q_{min}}\right)^{\frac{2}{3}}, \qquad q_i = \sum_k \left\{ 1.2p_k \;\middle|\; m_i = \underset{m_i}{\textbf{arg min}}\ |n_k - m_i| \right\}$$

$$\boldsymbol{D}(\vec{n},\ \vec{m}) = \sum_{i=1}^{\mathrm{n}} \boldsymbol{D}(m_i)$$

$$\boldsymbol{D}(m_i) = C_d + k_d \times \left(\frac{l_i}{l_{min}}\right), \qquad l_i = |m_i - m_h|$$

$$u + v + w = 1$$

03 优化方法

03 优化方法

➢ 优化问题：

- **混合整数问题**

模型并没有规定具体多少个物资点，即问题具有一个单整数变量。

- **非线性问题**

模型的目标函数与约束条件中存在指数函数与分段函数，问题不是简单线性函数

MINLP问题

- 分枝定界法
- 外部近似法
- GBD法

问题整数变量比较少
整数变量有1个
正实数变量有2n个

分枝定界法

NLP问题

03 优化方法

1. 序贯二次规划(SQP)

将一个非线性优化问题转化为一系列二次规划子问题来求解

基本思想：

在某个近似解处将原非线性规划问题简化为一个二次规划问题，用近似解代替构成一个新的二次规划问题，继续迭代

2. 线性近似约束优化(COBYLA)

使用线性逼近方法的鲍威尔非线性无导数约束优化的实现。

基本思想：

该算法是一种顺序信任区算法，它对目标函数和约束函数采用线性近似，其中近似是通过在变量空间中的n + 1个点处通过线性插值法形成的，并尝试在迭代过程中保持规则形状的单纯形。

3. 灰狼算法(Grey Wolf Optimization)，2014

元启发式优化算法，基于自然界中狼群狩猎行为的优化搜索方法

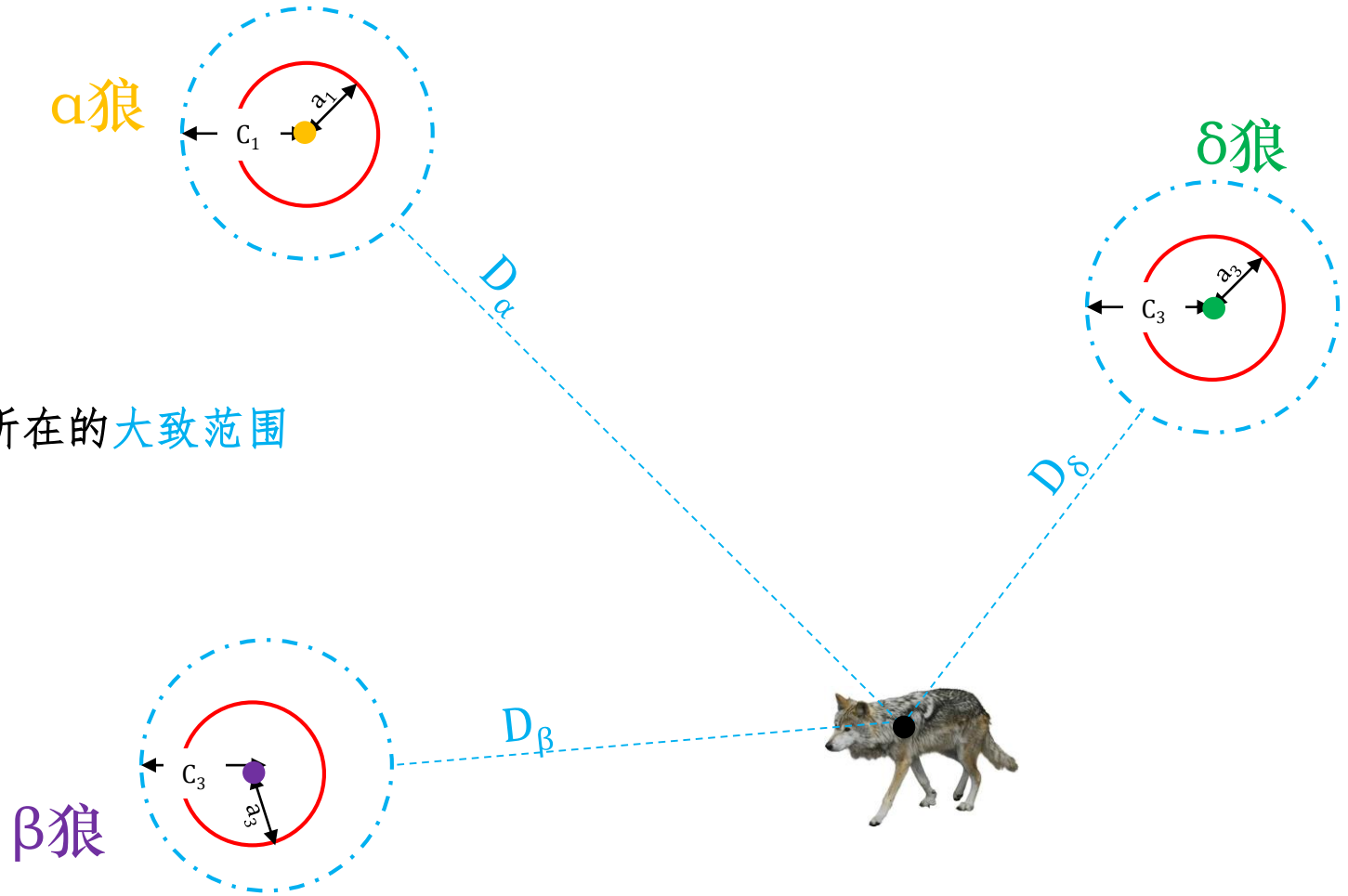
基本思想：

利用狼群具有识别潜在猎物(最优解)的能力，模拟跟踪、包围、围攻猎物等步骤。迭代范围内，头狼的平均位置指引剩下狼群的行动方向

03 灰狼优化算法(GWO)

元启发式优化算法，基于自然界中狼群狩猎行为的优化搜索方法

- 观察头狼们的位置：

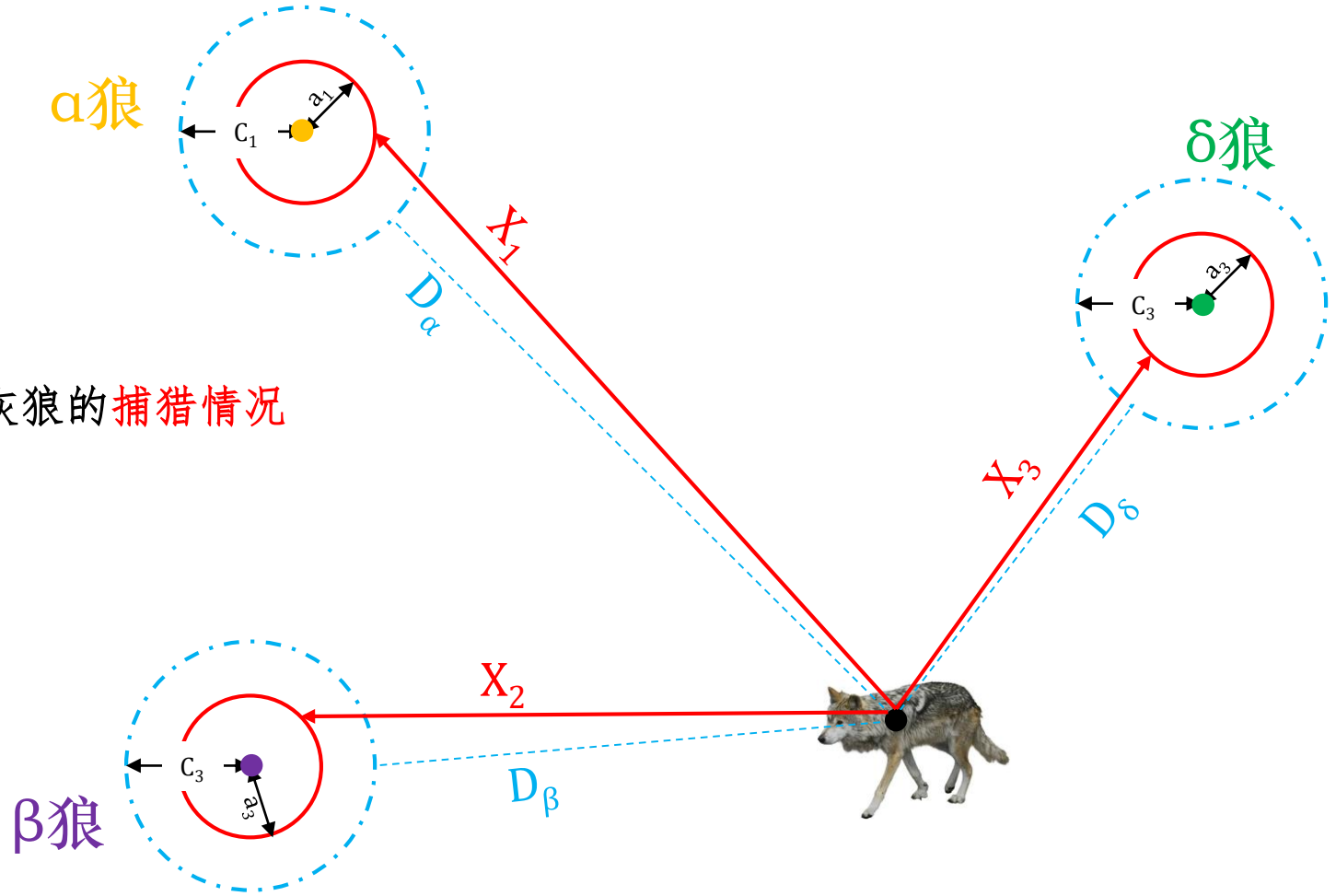


C_i是摇摆因子，决定了猎物所在的大致范围

03 灰狼优化算法(GWO)

元启发式优化算法，基于自然界中狼群狩猎行为的优化搜索方法

- 追寻头狼们的方向：



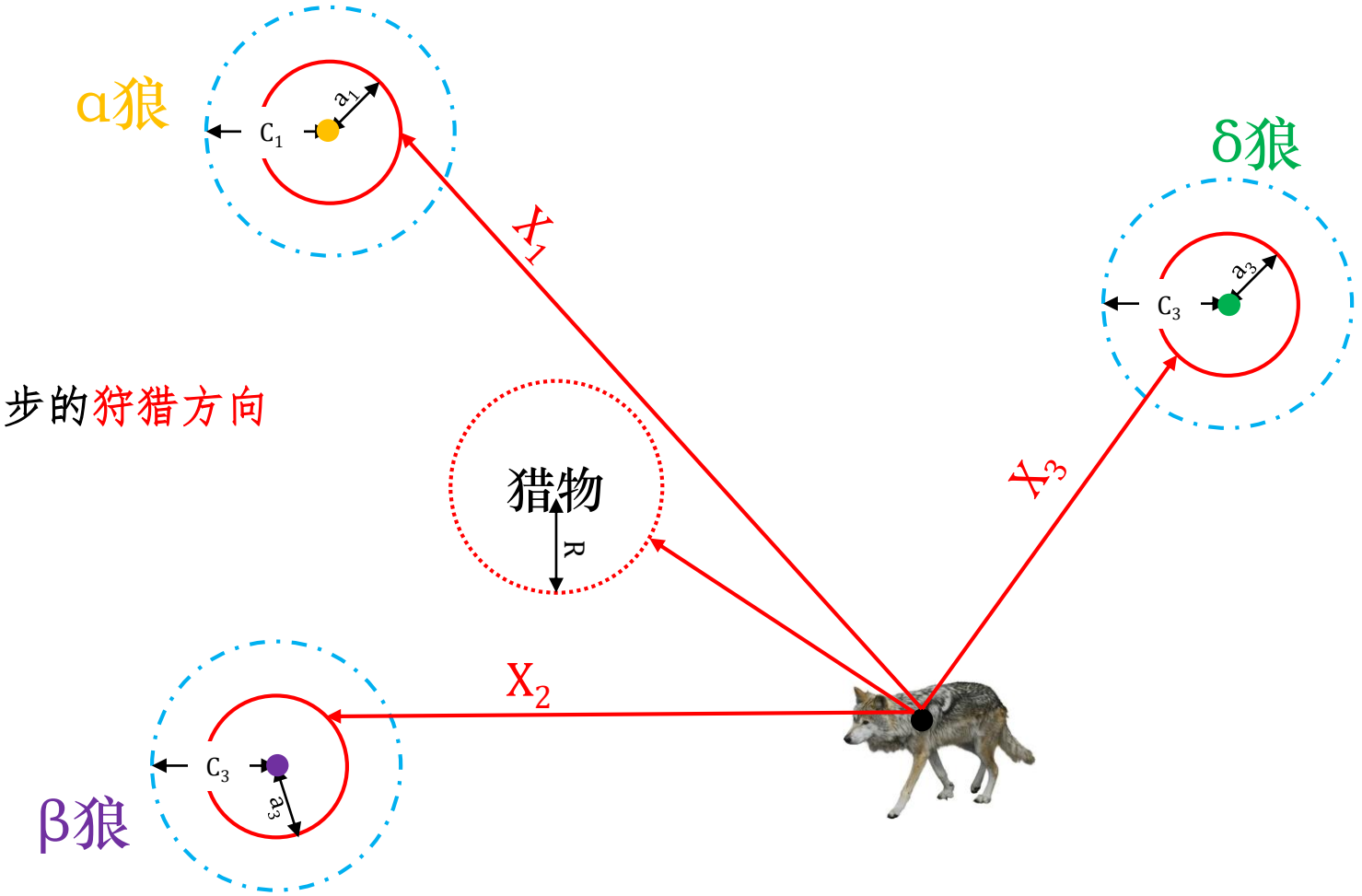
A_i是收敛因子，决定了该头灰狼的捕猎情况

03 灰狼优化算法(GWO)

元启发式优化算法，基于自然界中狼群狩猎行为的优化搜索方法

- 下一步狩猎迭代：

X_i求和，决定了该种狼下一步的狩猎方向



03 灰狼优化算法(GWO)

元启发式优化算法，基于自然界中狼群狩猎行为的优化搜索方法

➢一点细节：

A：是一个在区间[-a, a]上的随机向量
其中 a ∈[0, 2]，在迭代过程中呈**非线性下降**。

当 |A| > 1时，狼群散去，全局搜索
当 |A| < 1时，对应于局部搜索，狼群开始攻击，
下一时刻位置可以在当前灰狼与猎物之间的任何位置上

C：是一个在区间[0, 2]上的随机数，
此系数为猎物提供了随机权重，以便增加(|C|>1)或减少(|C|<1)

这有助于在优化过程中展示出**随机搜索行为**,以避免陷入局部最优。
值得注意的是，C并不是线性下降的，
C在迭代过程中是随机值，该系数有利于算法跳出局部，
特别是在迭代的后期显得尤为重要

```python
def initialize_pack(size=100,
                    lower_bounds=[],
                    upper_bounds=[],
                    target_function=None):
    """Initialize the original wolf pack for searching using tent map.

    Pack size by default is 100.
    """
    if len(lower_bounds) != len(upper_bounds):
        raise RuntimeError("Unmatched bound dimensions!")

    dimension = len(lower_bounds)
    half_bounds = [(j - i) / 2 for i, j in zip(lower_bounds, upper_bounds)]

    wolf_positions = np.zeros((size, dimension + 1))

    for i in range(0, dimension):
        wolf_positions[0, i] = random.uniform(lower_bounds[i], upper_bounds[i])

    wolf_positions[0, -1] = target_function(wolf_positions[0, 0:dimension])

    for i in range(1, size):
        for j in range(0, dimension):
            diff = wolf_positions[i - 1, j] - lower_bounds[j]
            if diff == half_bounds[j]:
                wolf_positions[i, j] = random.uniform(
                    lower_bounds[j], upper_bounds[j])
            else:
                new_diff = 2 * diff if diff < half_bounds[j]\
                    else 2 * (2 * half_bounds[j] - diff)
                wolf_positions[i, j] = lower_bounds[j] + new_diff

        wolf_positions[i, -1] = target_function(wolf_positions[i, 0:dimension])

    return wolf_positions


def initialize_top3(target_function=None,
                    lower_bounds=[]):
    """Initialize the top 3 wolves: Alpha, Beta & Delta."""
    return np.append(np.asarray(lower_bounds), target_function(lower_bounds))


def calculate_direction(control_factor, target, now):
    """Calculate which direction the wolf should go next."""
    r1 = random.uniform(0, 1)
    r2 = random.uniform(0, 1)
    C1 = 2 * r2
    D = abs(C1 * target - now)
    A1 = control_factor * (2 * r1 - 1)
    return target - A1 * D
```

```python
class Wolfpack:
    def __init__(self, size, lower_bounds, upper_bounds, target_function):
        self.pack = initialize_pack(
            size, lower_bounds, upper_bounds, target_function)
        self.size = size
        self.dimension = len(lower_bounds)
        self.lower_bounds = lower_bounds
        self.upper_bounds = upper_bounds
        self.target_function = target_function
        self.alpha = initialize_top3(target_function, lower_bounds)
        self.beta = initialize_top3(target_function, lower_bounds)
        self.delta = initialize_top3(target_function, lower_bounds)

    def update_top3(self):
        for i in range(0, self.size):
            if self.pack[i, -1] < self.alpha[-1]:
                self.alpha = np.copy(self.pack[i, :])
            elif self.pack[i, -1] < self.beta[-1]:
                self.beta = np.copy(self.pack[i, :])
            elif self.pack[i, -1] < self.delta[-1]:
                self.delta = np.copy(self.pack[i, :])

    def update_pack(self, control_factor=2):
        for i in range(0, self.size):
            for j in range(0, self.dimension):
                X1 = calculate_direction(
                    control_factor, self.alpha[j], self.pack[i, j])
                X2 = calculate_direction(
                    control_factor, self.beta[j], self.pack[i, j])
                X3 = calculate_direction(
                    control_factor, self.delta[j], self.pack[i, j])
                self.pack[i, j] = np.clip(
                    ((X1 + X2 + X3) / 3), self.lower_bounds[j], self.upper_bounds[j])
            self.pack[i, -
                      1] = self.target_function(self.pack[i, 0:self.dimension])

    def optimize(self, max_iter=100, show=True):
        count = 0
        self.update_top3()

        while count < max_iter:
            if show:
                print('Iteration {}: f(x)_min = {}.'.format(
                    count, self.alpha[-1]))
            control_factor = 2 * math.sqrt(1 - (count / max_iter))
            self.update_top3()
            self.update_pack(control_factor)
            count += 1

        print('\nIteration complete.')
        print('x = {}.'.format(self.alpha[0:self.dimension]))
        print('f(x)_min = {}'.format(self.alpha[-1]))
        return self.alpha


def optimize(target_function, lower_bounds, upper_bounds, size=100, max_iter=100, show=True):
    """Do grey wolf optimization."""
    if not isinstance(size, int) or not isinstance(max_iter, int):
        raise TypeError("Size and max iteration times must be integers.")
    if len(lower_bounds) != len(upper_bounds):
        raise RuntimeError("Unmatched lower and upper bound dimensions.")
    if len(lower_bounds) == 0:
        raise RuntimeError("No variable bounds input.")

    my_pack = Wolfpack(size, lower_bounds, upper_bounds, target_function)
    return my_pack.optimize(max_iter, show)
```

04 优化结果

04 优化结果

➢ 优化算法的比较：

- **序贯二次规划**

由于问题本身不能化成一个二次型故不能通过序贯二次规划的方法得到该问题最优解。

- **COBYLA方法**

在相同的条件下，得到相对更大的优化值；但是由于不需要迭代，计算的时间更短，t=0.15s。

$$f_{min} = 48.69$$

- **灰狼算法**

在同样的条件下，得到了相对更小的优化值；但是由于需要迭代，计算的时间相对更长，t=101s。

$$f_{min} = 45.74$$

➢ 参数设计与说明：

师生觉得到物资点还算“便利”的距离最大值记为 l_{max}
物资点到校医院距离的归一化系数记为 l_{min}
C_s, C_d, k_s, k_d 为经验系数

```
l_max = 5.8
l_min = 1

h_min = 1/1000
q_min = 100

C_s = 4
C_d = 0.9
k_s = 0.1
k_d = 0.04

u, v, w = 0.7, 0.2, 0.1
```

困难度归一化衡量系数，1000人的最小困难度为1

存储量归一化衡量系数，每平方米可以储存100份物资

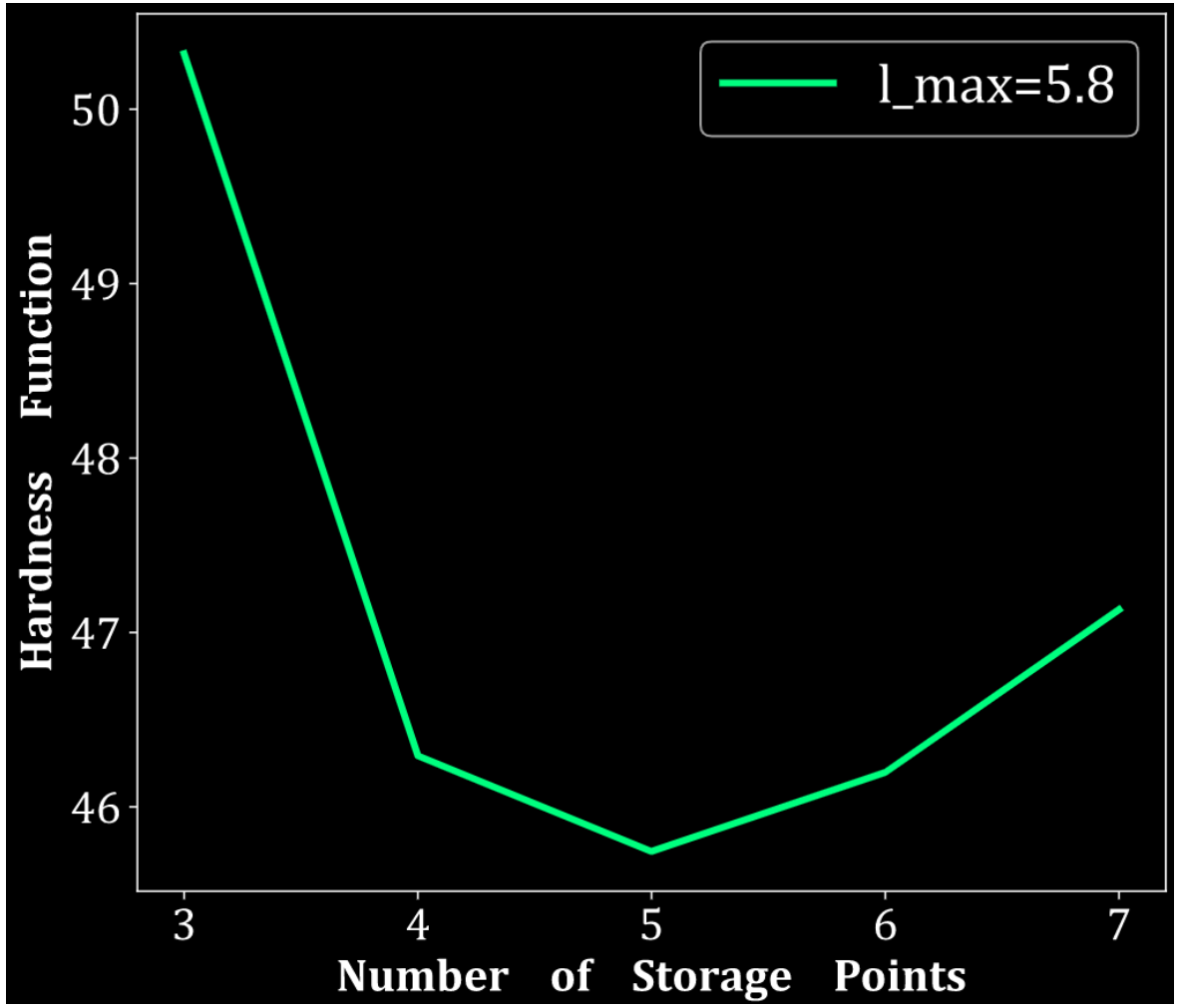
以5年为单位的配送费用

04 优化结果

➢ 灰狼算法

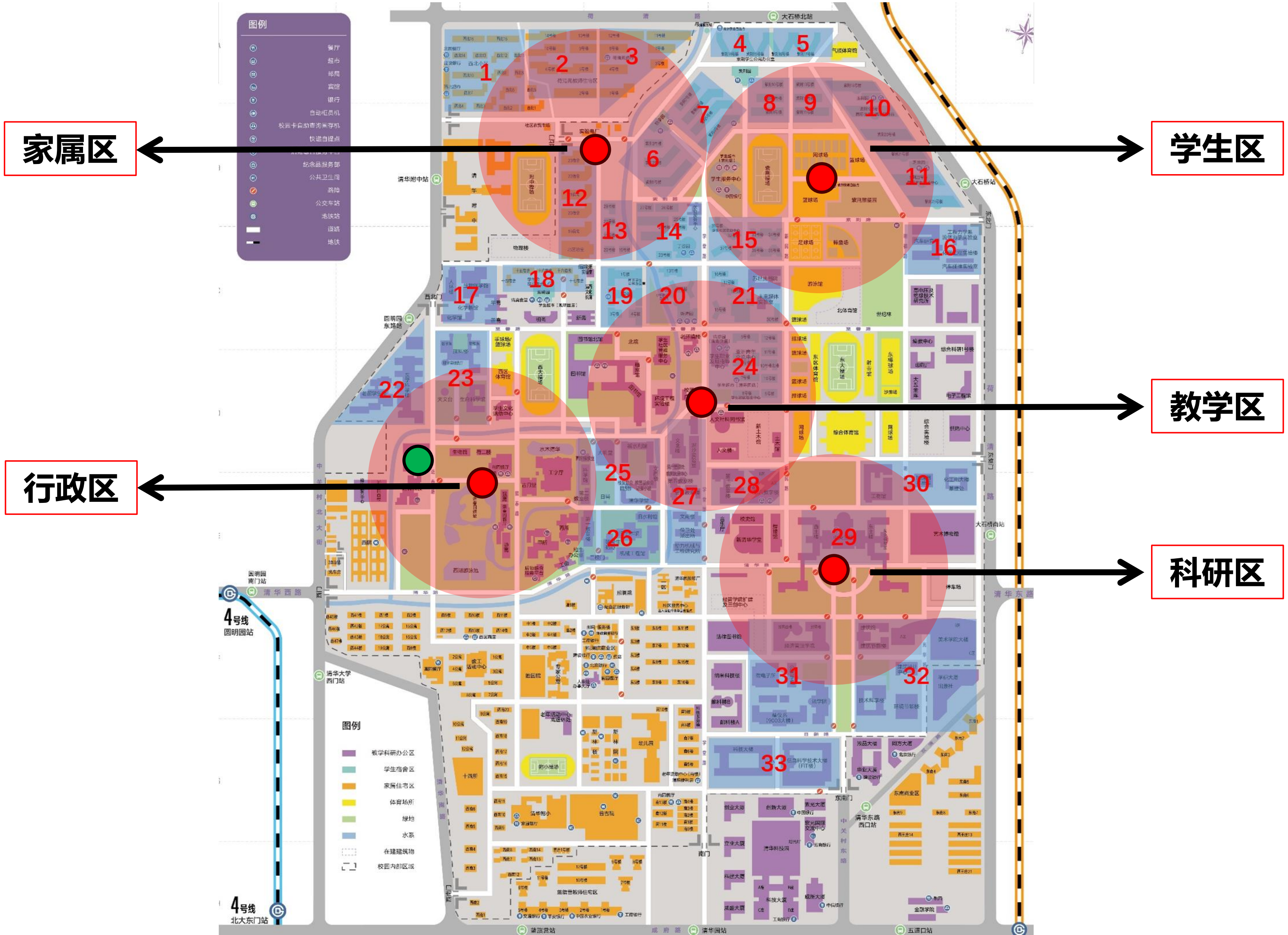
通过灰狼算法计算得到不同物资点时问题最优解

物资点个数	最优解
3	50.32
4	46.29
5	45.74
6	46.20
7	47.13



04 优化结果

➢ 结果示例

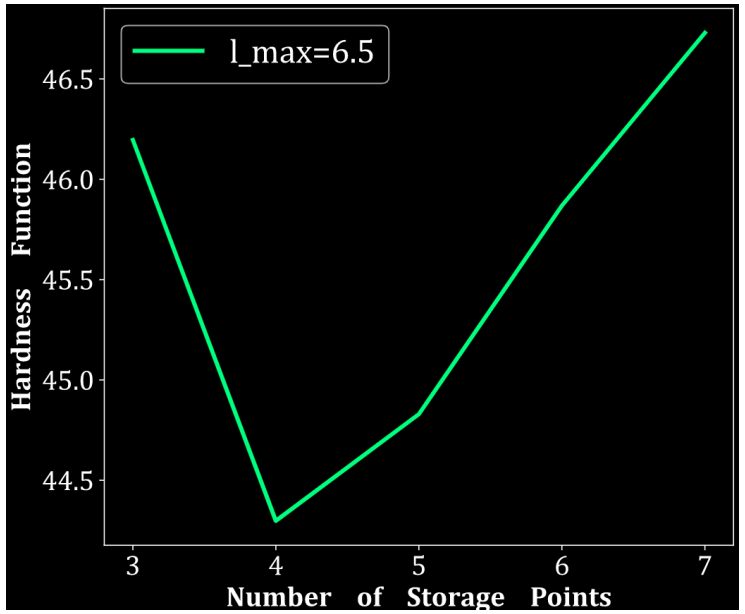

家属区
学生区
教学区
行政区
科研区
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
图例

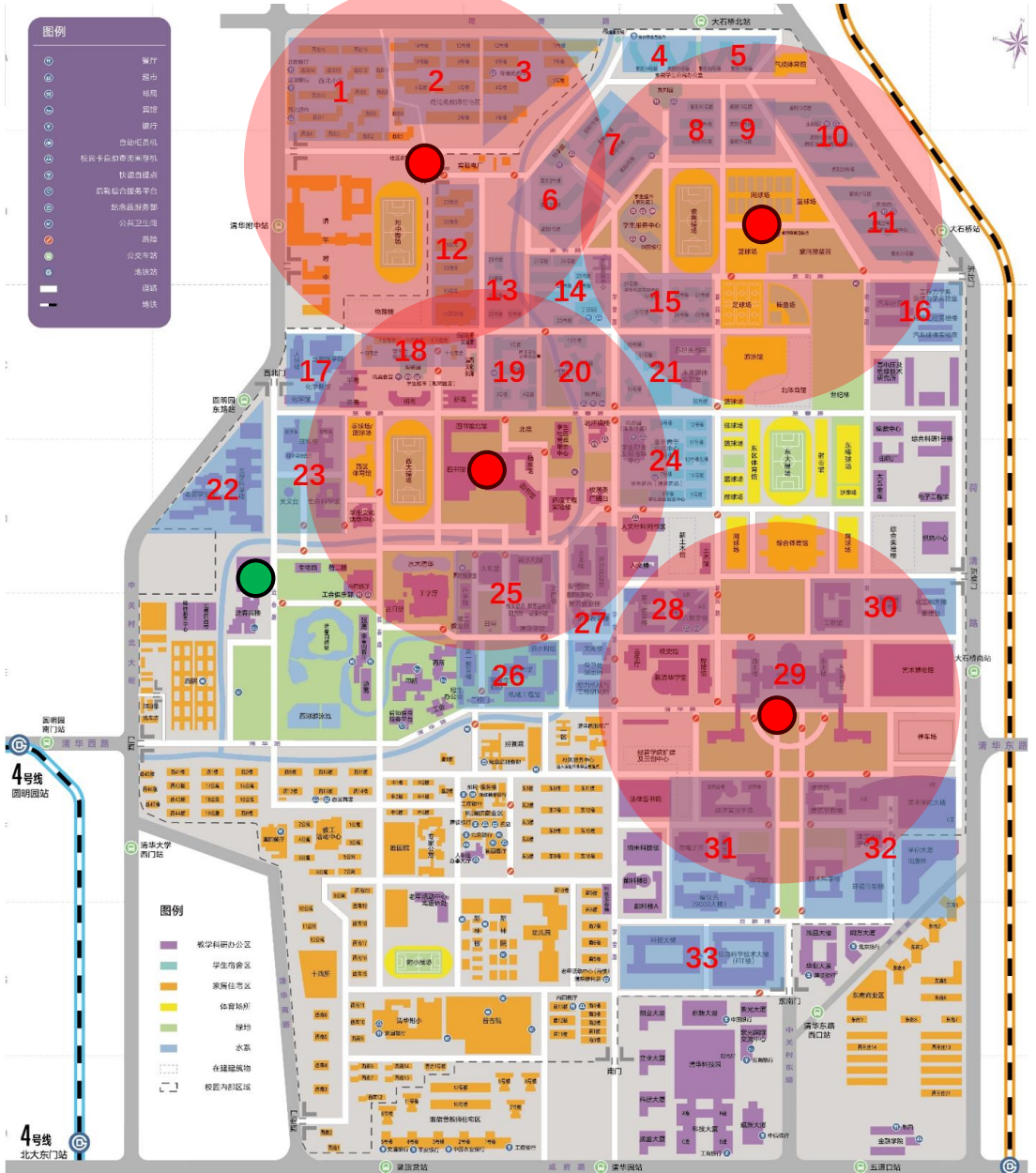
04 优化结果

➢ 参数调整

l_max = 6.5

f_min = 44.30



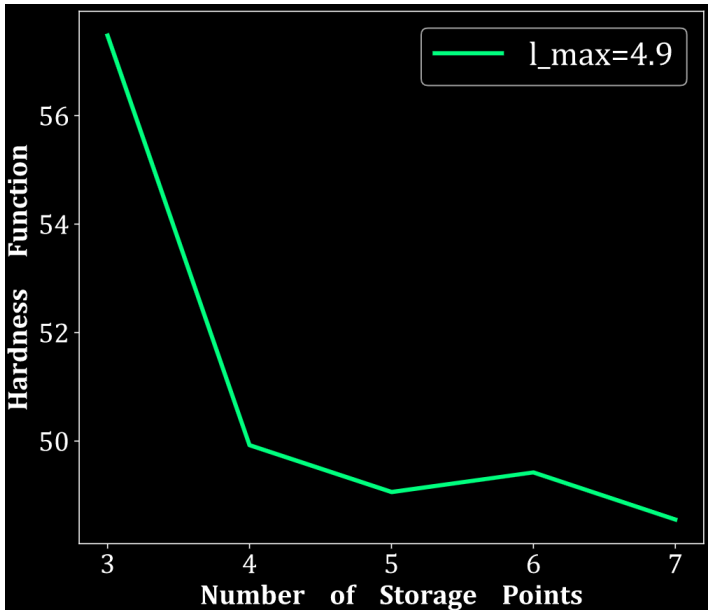


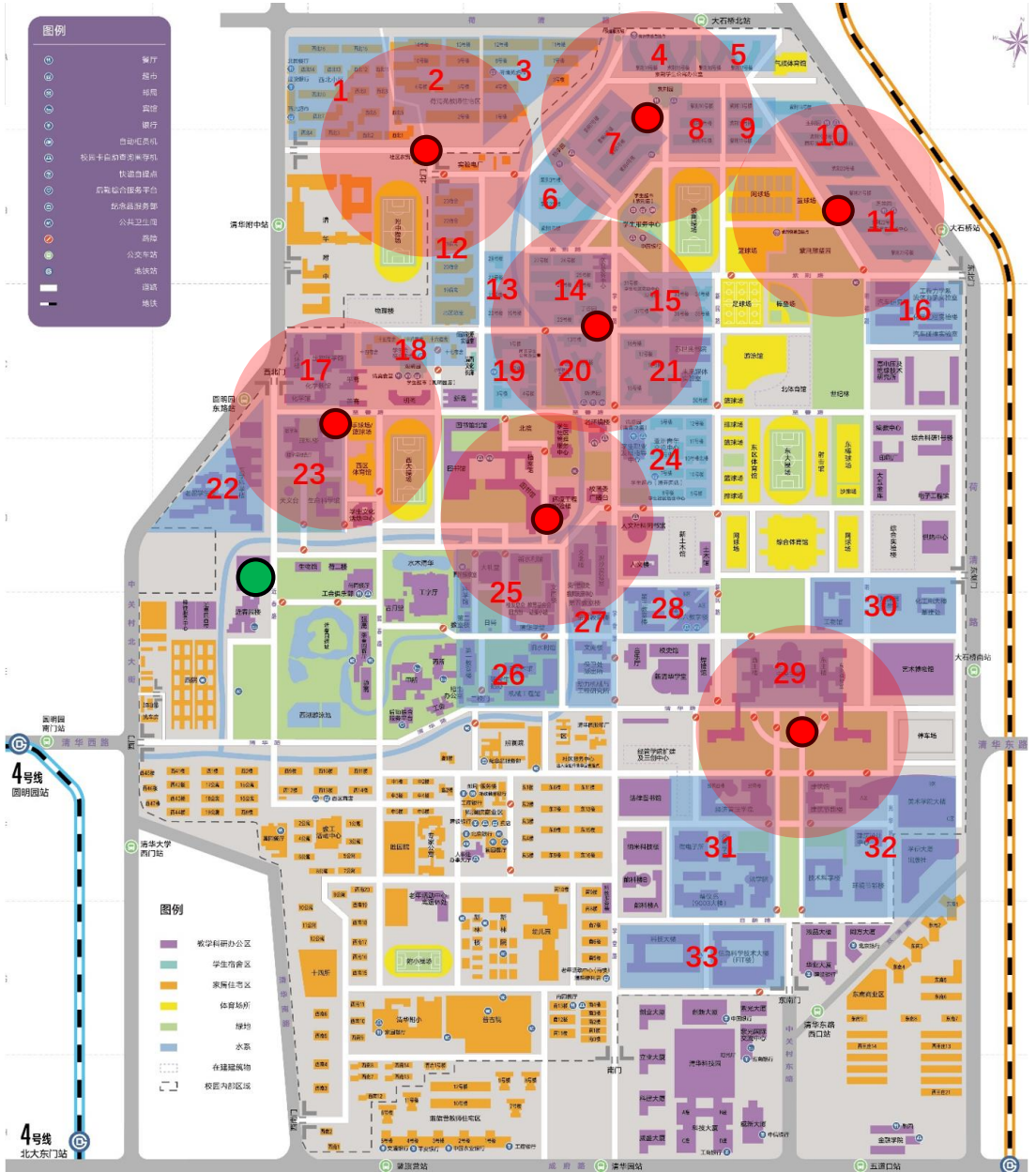
04 优化结果

➢ 参数调整

l_max = 4.9

f_min = 48.55





04 优化结果

➢ 参数调整

l_max = 5.8

q_min = 1000

f_min = 43.33