Optimization Competition

—Based on *Greedy Algorithm* and connectivity penalty

Jia-Xin Wang

China-UK Low Carbon College, Shanghai Jiao Tong University, Shanghai, China

June 6, 2025

Shanghai Jiao Tong University

Overview

1. Optimization objectives and constraints
2. Algorithm
3. Result

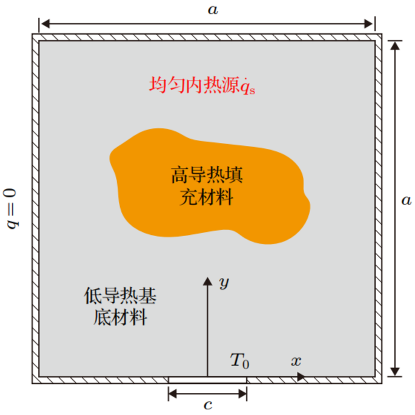


图 1　体点导热问题示意图

Fig. 1. The schematic diagram of the VP problem.

Find the minimum value of a function

$$T^* = \frac{k_0(T - T_0)}{L_x^2 q}$$

$$\overline{T^*} = \frac{1}{N}\sum_{i=1}^{N} T_i^*$$

Constraints

$$\sum_{i=1}^{N} \mathbb{I}(k_i > 250) \leq 0.15N$$

```
penalty_coeff*(num_components-1)
```

Optimization objectives and constraints

Mathematical Formulation of the Complete Optimization Problem

$$\min_{k} \left(\overline{T^*} + \lambda \cdot (\text{num\_components} - 1) \right)$$

$$\text{subject to: } \sum_{i=1}^{N} \mathbb{I}(k_i = k_1) \leq 0.15N$$

λ: Penalty coefficient (dynamically adjusted, see `penalty_coeff` in the code)
$k_i \in \{k_0, k_1\}$: Unit thermal conductivity (binary distribution, $k_0 = 1.0$, $k_1 = 500.0$)

Algorithm

Greedy Algorithm

Local Optimality: At each step, only the best current choice is considered, without backtracking or global consideration of future impacts.
Example: When making change, always use the largest denomination coin first.

No Aftereffect: Current choices do not affect the structure of subsequent subproblems (i.e., subproblems are independent).
Example: When selecting paths, the current path choice doesn't change weights of subsequent paths.

High Efficiency: Typically has low time complexity (e.g., $O(n \log n)$), making it suitable for large-scale problems.

No Global Optimality Guarantee: Since only local optima are considered, the final result may be an approximate solution (though optimal for certain specific problems).

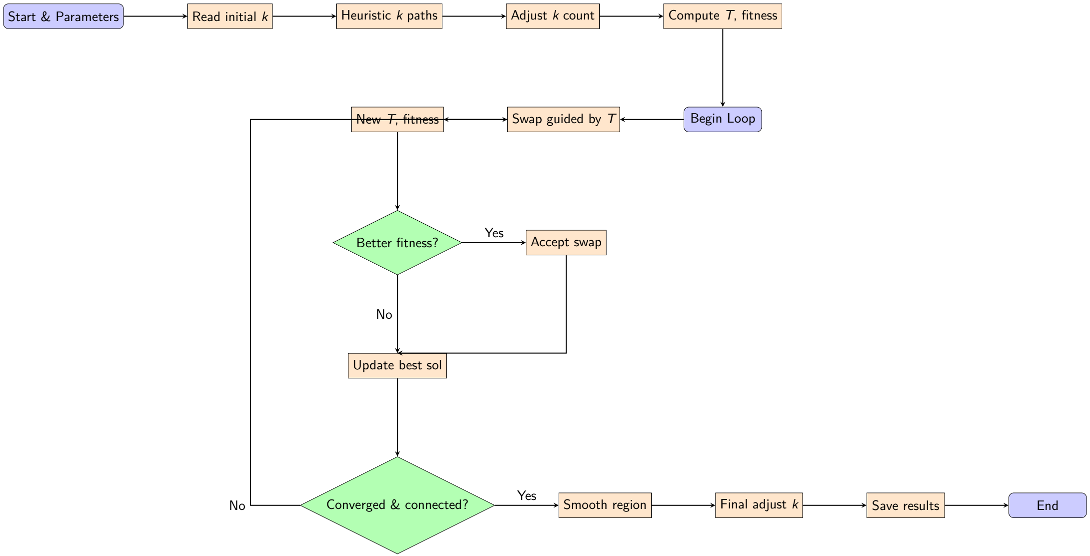
Algorithm

Optimization problem characteristics:

- **Discrete decision space** (Binary thermal conductivity distribution $k_i \in \{k_0, k_1\}$)
- **Decomposable local effects** (Changes in single unit's conductivity mainly affect neighboring region's temperature)
- **Submodularity property**: Diminishing marginal returns when adding high-conductivity material:

$$\Delta T^*(S \cup \{i\}) - \Delta T^*(S) \geq \Delta T^*(T \cup \{i\}) - \Delta T^*(T), \quad \forall S \subseteq T$$

Algorithm flow chart



Algorithm code snap

Greedy Algorithm Settings

```
max_iter_greedy = 5000;
num_swaps = 20;
best_solution_greedy = current_solution;
best_fitness_greedy = current_fitness;
fitness_history_greedy = zeros(max_iter_greedy, 1);

temp_sensitivity = T_flipped / max(T_flipped(:));

for iter = 1:max_iter_greedy
    idx1 = find(current_solution == 1);
    idx0 = find(current_solution == 0);
    best_swap_fitness = current_fitness;
    best_swap_solution = current_solution;

    for s = 1:num_swaps
        if rand() < 0.7
            [~, sorted_idx] = sort(T_flipped(:), 'descend');
            candidate_pos = intersect(sorted_idx(1:round(0.2*nx*ny)),
                idx0);
            if ~isempty(candidate_pos)
                swap0 = candidate_pos(randi(length(candidate_pos)));
                swap1 = idx1(randi(length(idx1)));
            else
                swap1 = idx1(randi(length(idx1)));
                swap0 = idx0(randi(length(idx0)));
            end
        else
            swap1 = idx1(randi(length(idx1)));
            swap0 = idx0(randi(length(idx0)));
        end

        new_solution = current_solution;
        new_solution(swap1) = 0;
        new_solution(swap0) = 1;

        k_opt = reshape(new_solution, [nx, ny]) * (k1 - k0) + k0;
    end
end
```

Listing 1: Greedy optimization strategy with temperature-based swapping

Algorithm code snap

New function: Calculate fitness with penalty term

```
function fitness = calculate_fitness_with_penalty(T_flipped, k_opt)
    % Compute average temperature
    mean_temp = mean(T_flipped(:));

    % Compute connectivity penalty term
    k_binary = k_opt > 250;
    cc = bwconncomp(k_binary, 8); % Find 8-connected components
    num_components = cc.NumObjects;

    % Dynamic penalty coefficient based on number of components
    if num_components <= 2
        penalty_coeff = 0.0005; % Mild penalty
    else
        penalty_coeff = 0.002;  % Stronger penalty
    end

    % Total fitness = average temperature + connectivity penalty
    fitness = mean_temp + penalty_coeff * (num_components - 1);
end
```

Listing 2: Fitness function with connectivity penalty

Post-processing - Improved smoothing

```
% Smoothing process
k_opt = reshape(best_solution_greedy, [nx, ny]) * (k1 - k0) + k0;
fprintf("Number of high-conductivity cells after optimization: %d\n", sum(sum(
    k_opt > 250)));

% Extract the largest connected region of high conductivity
k_binary = k_opt > 250;
cc = bwconncomp(k_binary, 8);
numPixels = cellfun(@numel, cc.PixelIdxList);
[~, idx] = max(numPixels);
k_smooth = false(size(k_binary));
k_smooth(cc.PixelIdxList{idx}) = true;

% Expand smooth region to sensitive cells if neighbors are connected
temp_sensitivity_binary = temp_sensitivity > 0.7;
for i = 2:nx-1
    for j = 2:ny-1
        if ~k_smooth(i,j) && k_binary(i,j) && temp_sensitivity_binary(i,j)
            neighbors = k_smooth(i-1:i+1, j-1:j+1);
            if sum(neighbors(:)) > 0
                k_smooth(i,j) = true;
            end
        end
    end
end

% Adjust to maintain the fixed number of high-conductivity cells
k_smooth = adjust_high_k(k_smooth(:)', num_high_k);
k_smooth = reshape(k_smooth, [nx, ny]);
k_opt = k_smooth * (k1 - k0) + k0;
```

Listing 3: Post-optimization smoothing of high-conductivity region

Result

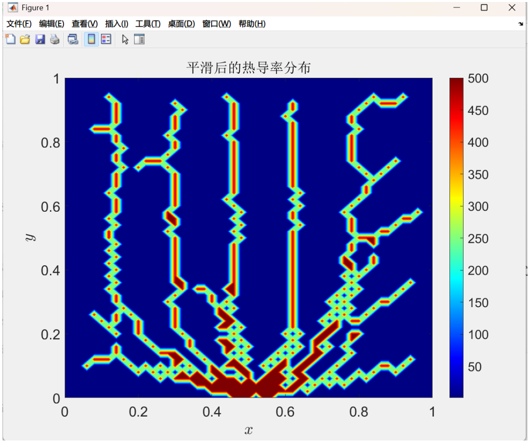


Figure: Thermal conductivity distribution after 3000 iterations

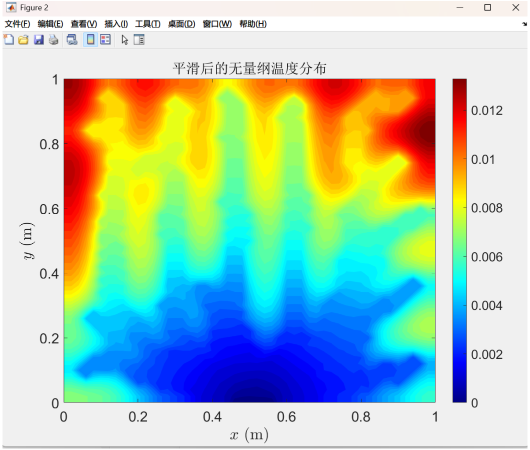


Figure: Temperature distribution after 3000 iterations (**0.006544**)

Python version

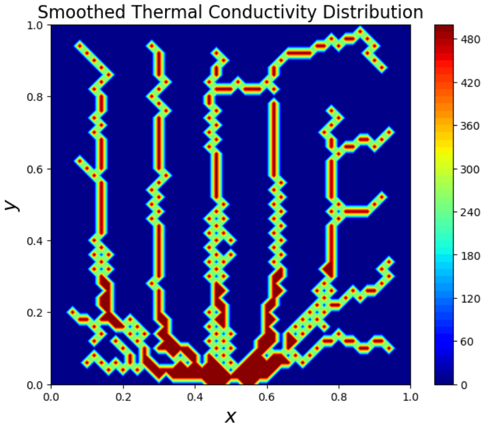


Figure: Thermal conductivity distribution after 5000 iterations (**0.006358**)

Accelerate:

- from **numba** import **njit**
- from **multiprocessing** import **Pool** (**Hints:** PSO and GA can be calculated in parallel, but other algorithms cannot.)

Grammatical Differences:

- (nx,ny) and (ny,nx)
- Function definition order

Python Version

```python
@njit
def compute_directional_averages(matrix):
    rows, cols = matrix.shape
    averages = np.zeros((rows, cols, 4), dtype=float)

    for i in range(rows):
        for j in range(cols):
            # Left
            averages[i, j, 0] = matrix[i, j] / 2 if i == 0 else (matrix
                [i, j] + matrix[i-1, j]) / 2

            # Right
            averages[i, j, 1] = matrix[i, j] / 2 if i == rows - 1 else
                (matrix[i, j] + matrix[i+1, j]) / 2

            # Down
            averages[i, j, 2] = matrix[i, j] / 2 if j == 0 else (matrix
                [i, j] + matrix[i, j-1]) / 2

            # Up
            averages[i, j, 3] = matrix[i, j] / 2 if j == cols - 1 else
                (matrix[i, j] + matrix[i, j+1]) / 2

    return averages

mesh_x = given_parameters['mesh_x']
mesh_y = given_parameters['mesh_y']
```

Listing 1: Compute directional averages using Numba JIT

Python version

pymoo (Suitable for Multi-Objective Optimization)

- Example import: `from pymoo.algorithms.moo.nsga2 import NSGA2`, `from pymoo.optimize import minimize`
- Supports genetic algorithms, particle swarm, and multi-objective optimization
- Suitable for research involving Pareto front and trade-off analysis

pyswarm (Suitable for Particle Swarm Optimization, PSO)

- Example import: `from pyswarm import pso`
- Simple and easy to use, best for single-objective continuous optimization

My code in Github:
`https://github.com/wjx0209/Optimization_Competition.git`

The End