Transactions on Environment, Energy and Earth Sciences
ISSN: 2960-1924, eISSN: 2960-2300 | Volume 4
GEMFE 2024

Optimizing Self-Assembly in Lattice Systems through Simulated Annealing: A Study of Cooling Strategies and Interaction Energies

Zexing Li

Beijing National Day School, Beijing, China

Abstract. The self-assembly is modeled by lattices containing three units of cell agents as molecules. This paper studies how various cooling strategies, including interaction parameters, act on the self-assembly of a lattice system containing three-unit cell agents of varying configurations. The self-organization is governed by attractive or repulsive forces between constituents and is a common driver of many natural processes, from the making of biological structures to that of nano-materials. This paper aims to find the energy minima using three cooling strategies: proportional, logarithmic, and exponential. The effect of interaction parameters on the system's behavior using interactions between different agent types, positive, negative, and neutral, is studied. Simulated annealing is used in simulations to find the energy reduction and clustering behavior. The results are analyzed utilizing heat maps, energy change calculations, and the overall system organization. The cluster movements ensure that local minima are avoided. The results obtained give insight into the relationship between cooling strategies and interaction parameters on the emergent properties of the lattice system.

Keywords: Agent-based simulation; simulated annealing; cooling strategy; interaction energy.

1. Introduction

The study of self-organization and self-assembly provides insights into the principles of complexity and order. Self- organization and self-assembly are foundational processes that influence various disciplines e.g., biology, material science, and even social studies. Self-organization enables the emergence of complexity, driving innovation and deepening our understanding of the natural world [1].

Self-organization and self-assembly happen without external guidance and are crucial in shaping the structures around us. In biology, molecular self-organization forms complex structures from proteins, DNA, and lipids, which are essential for life. [2] Materials science uses these principles to create advanced materials with specific properties, leading to innovations in electronics, photonics, and drug delivery [3-4]. At the molecular level, self-assembly involves molecules spontaneously forming ordered structures through interactions like hydrogen bonds, Van der Waals forces, and electrostatic interactions. These interactions are central to supra-molecular chemistry, where scientists design molecules to create complex assemblies with functions, such as molecular machines and responsive materials. [5] The ability to organize at the molecular level drives forward both biological understanding and technological advances. [6] In our study, we utilize a model to build larger molecules from smaller units. The challenge is finding the optimal configuration of these larger molecules, a problem compounded by the presence of numerous local minima in the energy landscape. To address this problem, we used simulated annealing. The physical process of annealing in metallurgy is based on heating a material to a high temperature and then gradually cooling it to remove defects and achieve a more stable structure. Simulated annealing mimics this approach by minimizing the energy of the system similarly. [7-8]

In the simulation, we minimize the energy for a given temperature and decrease the temperature gradually. This process helps us to avoid trapping in local minima and to reach the global minimum.

To check the correctness and effectiveness of our implementation of simulated annealing, we tested it using a spin glass system. A spin glass is a disordered magnetic system with randomly oriented spins, which serves as a challenging test due to its complex energy landscape. [9] By applying

simulated annealing to spin glass models, the effectiveness is demonstrated by finding low-energy states in the system. This validation confirms that our model can effectively optimize the self-assembly process in complex molecular systems. [10]

Simulated annealing can have different temperature schedules and cooling rates, which significantly impact its effectiveness. We tested various cooling strategies, including exponential cooling, logarithmic cooling, and proportional decreasing cooling, to identify the most efficient approach for our model. [11] Each strategy influences how quickly the system cools and how effectively it explores the configuration space, aiming to find a minimum. We will also investigate how the nature (e.g. interaction strength) of the building blocks affects the self-assembly process. The model assumes that molecules consist of positively charged, negatively charged, and neutral units. The interactions between these blocks—such as attraction between opposite charges, repulsion between like charges, and Van der Waals forces among neutral units—play crucial roles in stabilizing the assembled structures. By systematically varying these interactions and cooling strategies, we aim to understand the principles that govern molecular self-assembly and find global optimal configurations.

Inspired by previous studies, we will also apply techniques such as cell groups, local minima assessment, cluster moving, central incentive and unit number validation.

The paper starts with the computational approach details the simulated annealing technique, including its key com- ponents and a numerical example of a spin glass system. Simulation sections explore the effects of different cooling strategies and interaction parameters on model effectiveness. Then, results are analyzed through heat maps and energy calculations to understand the system's properties.

2. Computational Approach

2.1. Simulated Annealing

Simulated annealing is an optimization technique that relies on three fundamental components: the Boltzmann probability, a temperature schedule, and the Metropolis criterion. The Boltzmann probability describes the likelihood of a system being in a state with energy E at a given temperature T. The Boltzmann probability describes a system that can exchange energy with its surroundings. This probability is given by.

$$P(E) = e^{-\frac{E}{kT}} \tag{1}$$

Where k is the Boltzmann constant. When considering a change in the system's energy (ΔE), the probability of accepting this new configuration is governed by

$$P(\Delta E, T) = e^{-\frac{\Delta E}{kT}} \tag{2}$$

Here, ΔE represents the change in energy, and T is the system's temperature.

The temperature schedule is crucial for controlling the cooling process in simulated annealing. The system is optimized at each temperature but the resulting configurations might be local minima instead of global minimum. To move the system from a local minimum we change the temperature, but too big a change could freeze the configuration into local minima. The temperature T decreases over time according to a cooling schedule, typically following an exponential decay. This can be expressed as

$$T_{new} = \alpha \cdot T_{old}2 \tag{3}$$

Where α is a cooling rate parameter, often close to 1, which determines the rate of cooling.

The Metropolis criterion is used to decide whether to accept or reject a new configuration based on the energy change (ΔE) and the current temperature (T). If the new configuration decreases the energy, it is always accepted. If the energy increases, the new configuration can still be accepted with a certain probability, which depends on the magnitude of the energy increase and the current temperature.

By iteratively applying these approaches, simulated annealing can effectively explore the configuration space of molecules. This method allows for the finding of energetically favorable configurations

2.2. Numerical Example and Test: Spin Glass

A typical implement of simulated annealing models is a spin glass system using $4\ N \times N$ arrays representing the interaction strengths between neighboring spins: up, down, left, and right.

2.2.1. Spin Glass.

The spin glass model is a disordered magnetic material with randomly oriented magnetic moments, resembling a frozen liquid. Unlike conventional magnets, spin glasses lack long-range magnetic order due to competing interactions, leading to frustration. They exhibit complex magnetic behavior, including slow dynamics and memory effects. Spin glasses have applications in condensed matter physics, statistical mechanics, and optimization problems. [2, 6]

The energy in the system, denoted as E, is calculated as the sum of the interactions between a spin at position i,j and its neighboring spins. Mathematically, it is represented as:

$$E = 2\sigma_{i.j} \sum_{k,l} J_{k,l}\, \sigma_{k,l} \tag{4}$$

Where $\sigma_{k,l}$ represents the spin at positio i, j, with values +1 or −1; $J_{k,l}$represents the interaction strength between spins at positions k, l; The summation $\sum_{k.l}$ extends over neighboring spins; The term $\sigma_{i.j}$ accounts for the change in energy when flipping spin $\sigma_{i.j}$. In the spin glass model, s_v and s_h are pivotal in defining the system's energy and state configuration, respectively. s_v and s_h stands for the Spin Variables, which denote the individual spin states within the model.

These variables can take discrete values, often ±1 in Ising models, representing the orientation of each spin. To observe the pattern's variation with initial conditions, we fixed s_v at 1 and changed the magnitude and positivity of s_h. Energy change is calculated by subtracting the energy of a new spin configuration from an old spin configuration. For boundary conditions, see Appendix A.

2.2.2. Simulation and Discussion of Results.

When s_h and s_v are both set to 1, an island pattern emerges due to the symmetric and strong randomness introduced during initialization. This randomness allows neighboring spins to align either parallel or anti-parallel to each other, leading to regions where spins align in the same direction by chance. These regions form large clusters or islands with relatively stable configurations, resulting in the observed island pattern. Due to the random nature of spin alignment and the presence of large clusters with aligned spins, the overall magnetization of the system tends to be low. The symmetric randomness results in relatively low energy configurations within the system, as the spins have equal chances of aligning favorably with their neighbors.

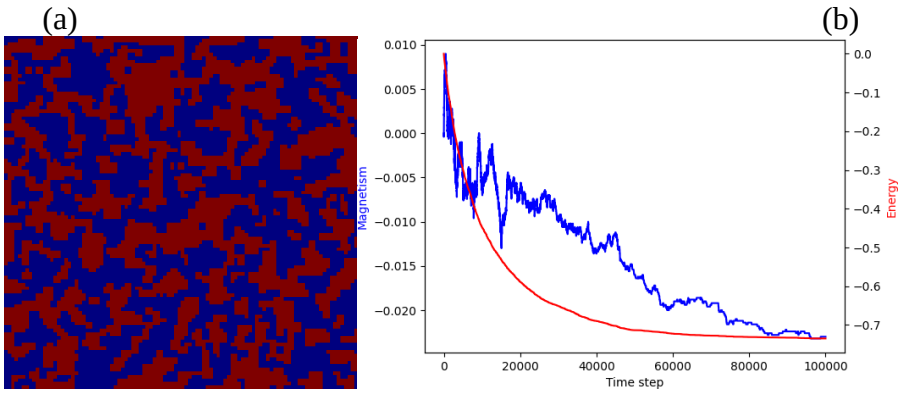


Figure 1. Properties of spin glass when $s_h = 1,\ s_v = 1$. (a) Distribution of spins (blue is up spin, red is down spin). (b) Magnetization (blue) and energy (red) as a function of iteration

Stripe patterns emerge as elongated domains of aligned spins, separated by regions where spins take on a contrasting alignment. This pattern arises from the system's efforts to reduce frustration by segregating spins into orderly domains. When $s_h = 1$ and $s_v = -1$ in the Ising model simulation, a stripe pattern emerges due to the asymmetric randomness in initialization, favoring anti-parallel alignment vertically. This pattern results in neighboring spins alternating in their orientations within the stripes. Consequently, the collective alignment of spins along the stripes leads to high magnetization, as the spins predominantly align in one direction within each stripe. The asymmetry introduced by $s_v = -1$ enhances the stability of the stripe pattern, ensuring a persistent alignment of spins and sustaining the high magnetization over time. Therefore, alongside the emergence of the stripe pattern, the Ising model simulation exhibits notable high magnetization under these conditions.

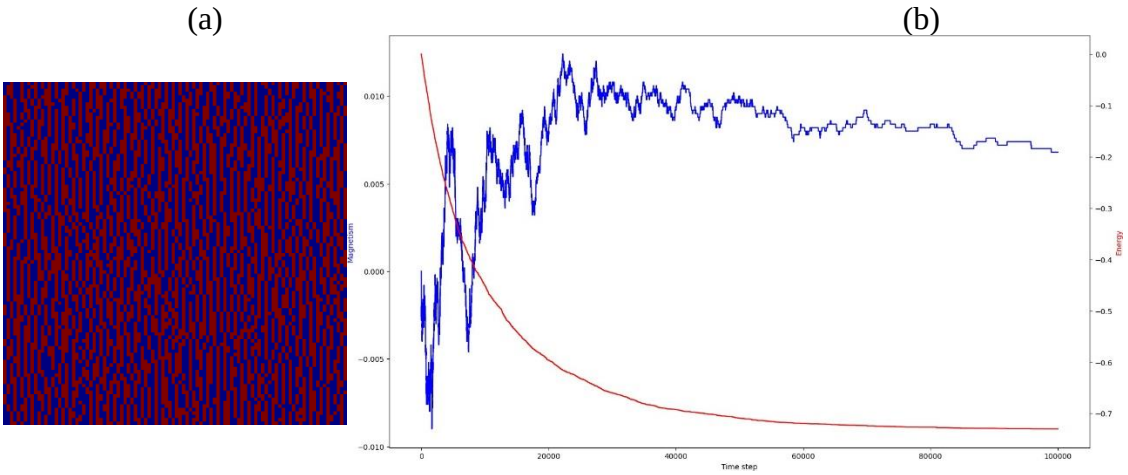


Figure 2. Properties of spin glass when $s_h = 1,\ s_v = -1$. (a) Distribution of spins (blue is up spin, red is down spin). (b) Magnetization (blue) and energy (red) as a function of iteration

The condensed phase in spin glass systems signifies a large-scale coherent alignment of spins, resulting in a macroscopic order amidst the disorder. When the magnitude of randomness is set to 0.1, the interactions are weaker, introducing more randomness into the system and potentially resulting in higher energy configurations with greater fluctuations. The energy of the condensed phase is notably lower than that of disordered configurations, as the widespread alignment of spins greatly reduces the system's overall frustration. Consequently, magnetization in the condensed phase is markedly low, reflecting the extensive coherence among spins. As shown in Fig. 1, large clusters of the same units are formed.

(a) (b)

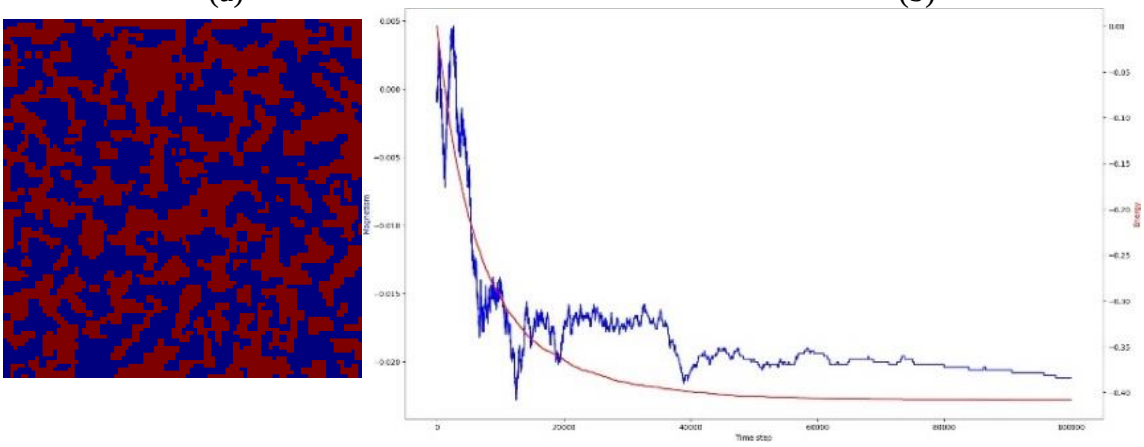


Figure 3. Properties of spin glass when $s_h = 1,\ s_v = 0.1$. (a) Distribution of spins (blue is up spin, red is down spin). (b) Magnetization (blue) and energy (red) as a function of iteration

3. Simulations of self-assembly of molecules

Self-assembly is a very difficult problem because it is hard to find the minimal energy configuration. The system consists of many local minima so we use the simulated annealing. The use of simulated annealing is not simple, because the cooling process is strongly affected by the cooling strategy.

We assume that the molecules consist of positively or negatively charged units, and neutral units. The interaction energies between units are based on their charge. The positive and negative units attract each other; and repel their same type; the neutral ones are slightly attracting every other with Van der Waals interaction. We test different interactions and show how they affect self-assembly.

To test simulated annealing, we have to check that we can reach the global minimum of energy from different random initial configurations.

The following are the rules of the simulation:

Shapes

1) Line Shape: The agent (molecular unit) can have a line shape which can be oriented either vertically or horizontally.

2) Boot Shape: The agent can have a boot shape, which can be oriented in one of four ways: top-left, bottom-left, top-right, or bottom-right.

Rotation: The agent's shape can be rotated by 90°, 180°, or 270°.

Boundary Conditions: Agents are not allowed to leave the simulation cell.

To do that, we have to make sure that molecules do not overlap during moves. In the simulated annealing process, we can have different choices: one randomly selected molecule at a time and change temperature, several randomly selected molecules at once, etc.

3.1. Effect of Cooling Strategy

3.1.1. Exponential Cooling.

The equation below represents a cooling strategy where the temperature (Temp) decreases over time based on its current value.

$$Temp_n = \frac{Temp_{n-1}}{1+Temp_{n-1}\times 0.01} \tag{5}$$

It can be interpreted as a form of exponential decay or diminishing returns, where the cooling effect becomes less pronounced as the temperature decreases.

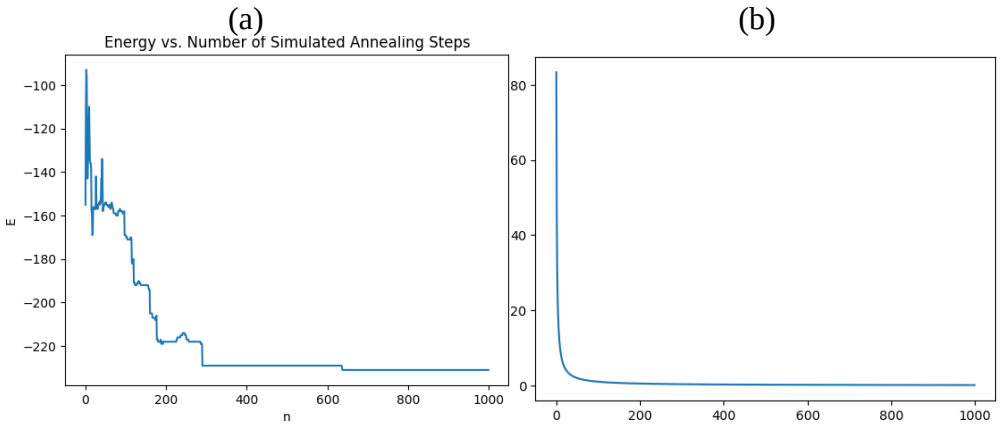


Figure 4. Change in parameters under exponential cooling. (a) The change in energy as a function of iterations. (b) The change in temperature as a function of iterations

Here, the denominator indicates that as the temperature increases, the denominator increases, which leads to a smaller value for $Temp_{new}$. The higher the initial temperature, the larger the denominator becomes, resulting in a less pronounced reduction in temperature. It is indicative of a cooling strategy where the system loses heat more efficiently at higher temperatures but slows down as it cools. As the temperature approaches lower values, the term $Temp \times 0.01$ becomes small, and the equation approaches $Temp_{new} \approx Temp$. It suggests that the cooling process slows down as the temperature gets lower.

3.1.2. Logarithmic Cooling.

$$Temp_n = \frac{Temp_{n-1}}{\ln(1\cdot(i+1))} \tag{6}$$

The term $\ln(1 \cdot (i+1))$ in the denominator indicates that the temperature decreases with the logarithm of the iteration number. As i increases, $\ln(1 \cdot (i+1))$ increases, which means the temperature decreases. However, because the logarithm function grows slowly, the rate of decrease in temperature also slows down over time.

Initially, when i is small, the temperature decreases more quickly. As i becomes larger, the logarithmic function increases more slowly, leading to a slower decrease in temperature. As i approaches very large values, $\ln(1 \cdot (i+1))$ continues to increase, but at a diminishing rate. This means the temperature approaches a lower bound asymptotically, cooling down more and more slowly.

3.1.3. Proportional Decreasing Cooling.

The equation

$$Temp_n = Temp_{n-1} \times Cooling\ Rate \tag{7}$$

represents a cooling strategy where the temperature decreases proportionally to its current value by a constant factor, Cooling Rate, at each step. This can be interpreted as a geometric progression or exponential decay, where the temperature decreases multiplicatively.

Cooling rate is a constant factor less than 1. Each time the temperature is updated, it is multiplied by this factor, resulting in a reduction of the temperature. The temperature follows a geometric progression, decreasing by the same proportion in each step. It means if the initial temperature is $Temp_0$, after n steps, the temperature will be:

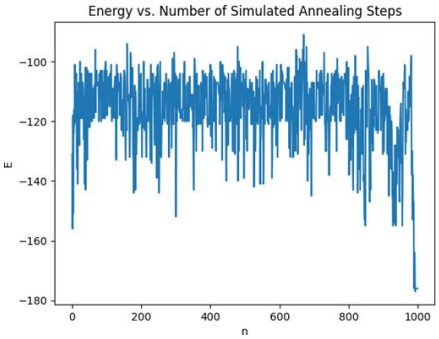


Figure 5. The change in energy under logarithmic cooling as a function of iterations

$$Temp_n = Temp_0 \times (Cooling\ Rate)^n \tag{8}$$

Which is visualized in Fig.6b below:

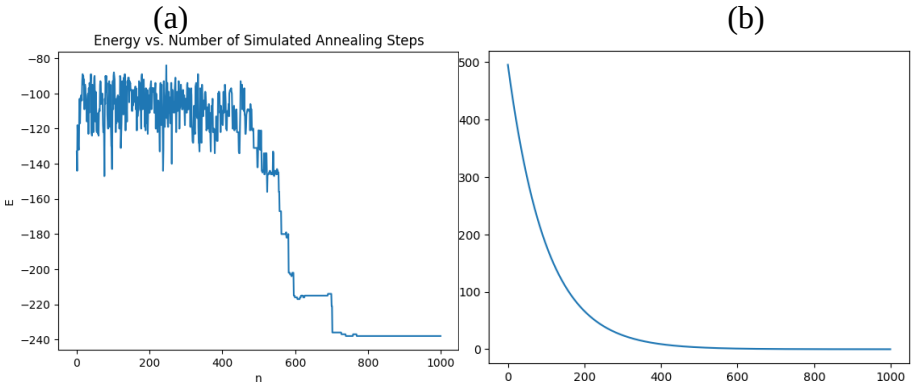


Figure 6. The change in parameters under proportional cooling. (a) The change in energy as a function of iterations. (b) The change in temperature as a function of iterations

3.1.4. Self-Assemble Effectiveness.

This analysis evaluates three different cooling strategies. It is crucial to consider the resulting energy levels and the scatters of the configurations to determine which strategy is the most effective. The interaction parameters between units are defined as follows:

Table 1. Interaction energies

	A	B	C
A	-1	-1	-1
B	-1	4	-11
C	-1	-11	4

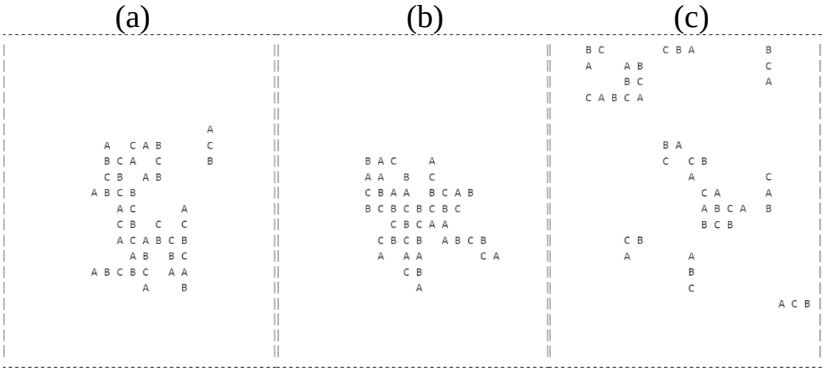


Figure 7. Cell configuration under different cooling strategies. (a) Proportional cooling strategy (b) Exponential cooling strategy (c) Logarithmic cooling strategy

By running our program using the parameters above, the optimal configuration for each cooling strategy is given in Fig.7.

The proportional cooling rate strategy (Fig.7a) resulted in a dense clustering of agents. This suggests that the proportional cooling rate effectively found a low-energy state; the tightly packed arrangement could imply that the strategy effectively reduced the energy. Meanwhile, The exponential cooling strategy also reached a dense clustering state, showing its effectiveness. In contrast, the logarithmic cooling strategy (Fig.7b) is the most dispersed among the three, suggesting that the logarithmic cooling rate (Fig.7c) might have reduced the temperature too slowly. This slower reduction in temperature could have led to an insufficient exploration of lower-energy states.

However, it is not possible to determine the efficiency of proportional and exponential cooling strategy qualitatively through this single configuration, therefore heat maps are introduced to qualify the dispersion. By simulating the self-assembly process under the same random initial configuration, we found the average energy decrease for each cooling strategy as shown in Table 2:

Table 2. The average energy change for different cooling strategies using 15 random configurations

Cooling strategy	Average energy change
Proportional cooling	191.0
Exponential cooling	213.0
Logarithmic cooling	66.3

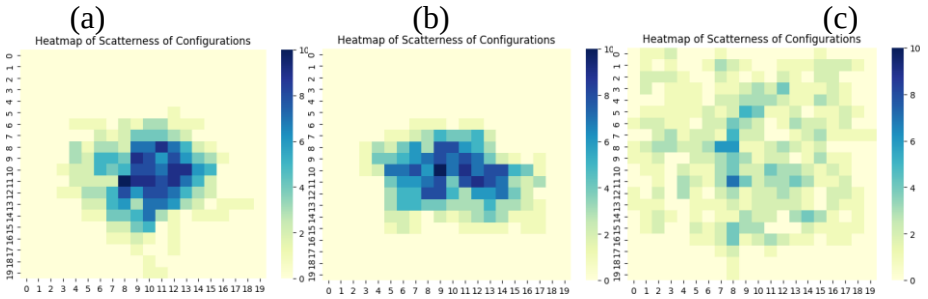


Figure 8. Heat map of cell distribution under different cooling strategies. (a) Proportional Cooling Strategy (b) Exponential Cooling Strategy (c) Logarithmic Cooling Strategy.

Heat maps of dispersion are also used to observe the overall configuration pattern. As shown in Fig.8, the exponential cooling strategy leads to the most condensed configuration with darker-colored units, indicating more occurrence. The proportion cooling strategy is also condensed but with less repeated occurrence reflecting an unstable best configuration. While merely any dark-colored units can be seen in Fig.8c, logarithmic cooling strategy is proved to be least effective. According to Table II., the number justified that the exponential cooling strategy is the most effective following the proportional cooling strategy. The logarithmic cooling strategy is proved to be most ineffective as it can only decrease 30 of the energy compared to the optimal strategy. Both dispersion maps and average change in energy indicate the priority of the exponential cooling strategy.

3.2. Effect of Interaction Between Molecules

In this part, we test how the assembly changes with different interaction parameters. The influences of parameters on simulated annealing performance are analyzed by altering the charge and magnitude of interaction energies between units. We test different combinations of interaction energies and try to find the optimal parameter for the simulated annealing process.

3.2.1. Effect of Direction of Interaction.

Different combinations of positive, negative, and neutral unit interactions are applied. The detailed interaction parameters are shown in Table 3. By running the simulated annealing process, we are able to obtain different configuration heat maps of the overall dispersion as well as the dispersion of units

A, B, and C. All configurations are cooled down by a proportional cooling strategy given its simplicity. The process, 10 larger iterations each with 1000 simulated annealing iterations, is repeated 15 times to ensure generality.

Table 3. Interaction energies for combination 1

	A	B	C
A	-5	-3	-2
B	-3	-5	-3
C	-2	-3	-5

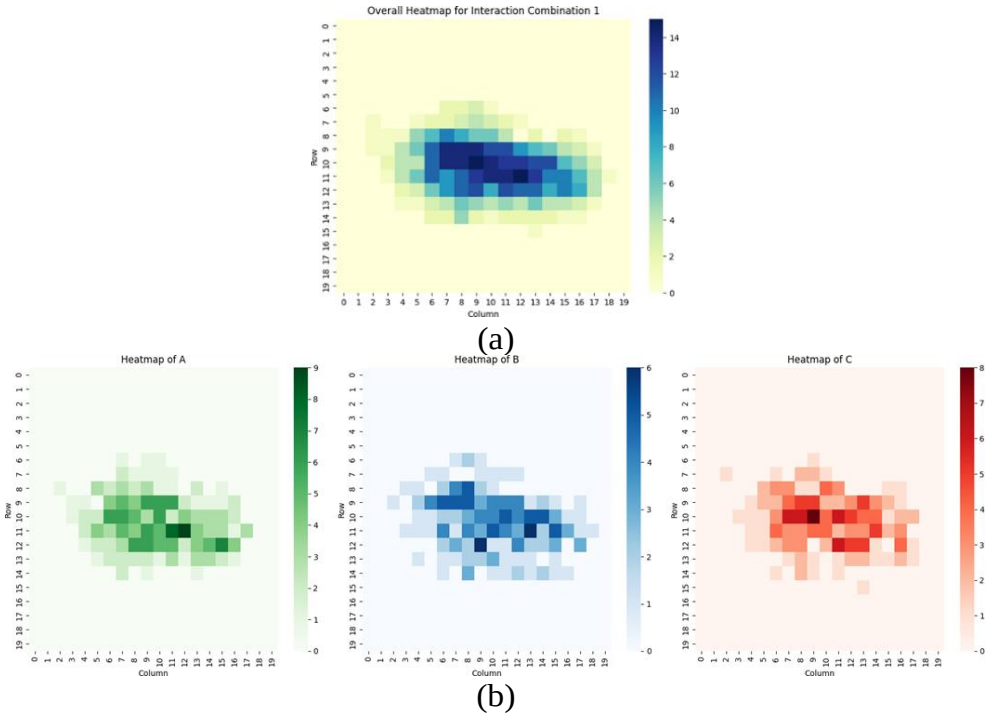


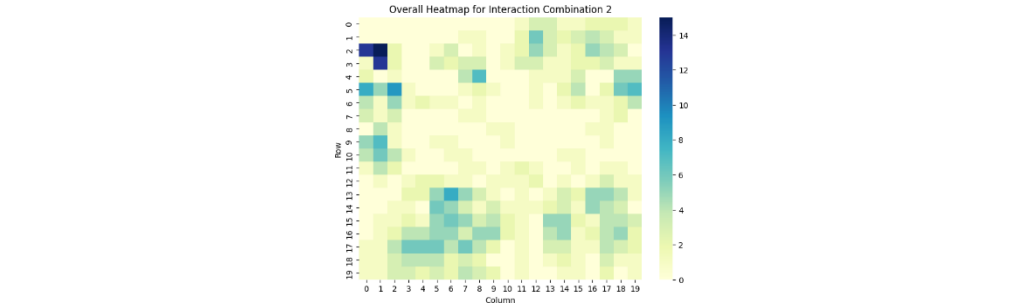
Figure 9. Configuration of interaction parameter combination 1. (a) Dispersion map of overall configuration. (b) Dispersion map of configuration of units

It can be seen that all interactions are negative meaning that the simulated annealing process, incentivized by minimizing energy, pushes units together. Accordingly, most units are optimized to the center of the map as shownin Fig.9. The dispersion of units A, B, and C, due to the equal interaction energy to each other, are approximatelyeven throughout the central space. Condensed areas for one single unit are not observed.

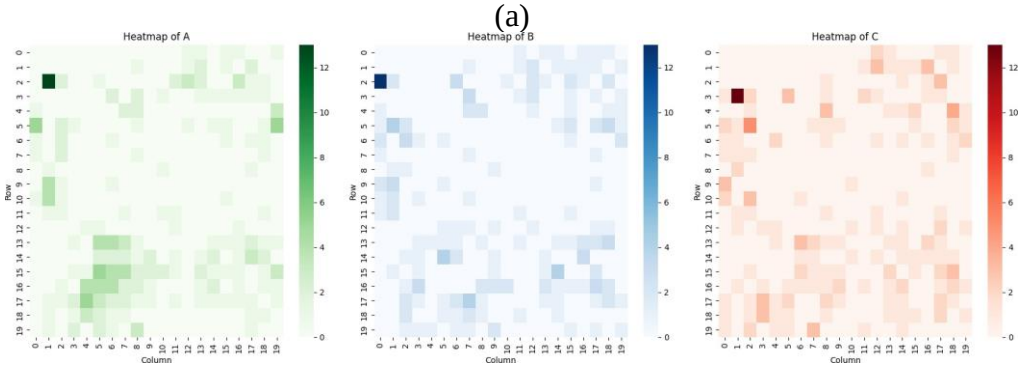
The second interaction combination is shown in Table 4. Given that it is all positive, the simulated annealing process will push units apart as the energy increases drastically when units are stuck to each other. In this case, all units are almost evenly dispersed throughout the space. However, a three-unit cell consisting of A, B, and C can always be observed on the top left of Fig.10 which is surprising. It might be due to the fact that the random initial configuration is held the same during the whole computation. Given that the program is unlikely to move agents around in space due to positive interaction energy, the initial configuration of this part of the map remained consistent throughout the whole process.

Table 4. Interaction energies for combination 2

	A	B	C
A	1	2	3
B	2	1	2
C	3	2	1



(a)



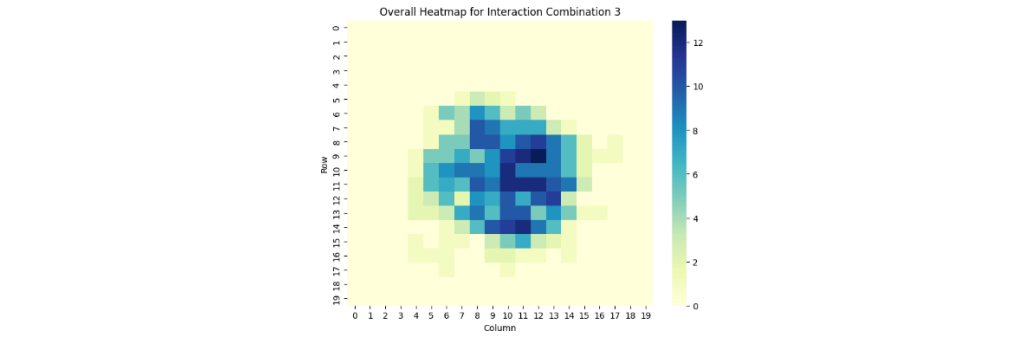
(b)

Figure 10. Configuration of interaction parameter combination 2. (a) Dispersion map of overall configuration. (b) Dispersion map of configuration of units

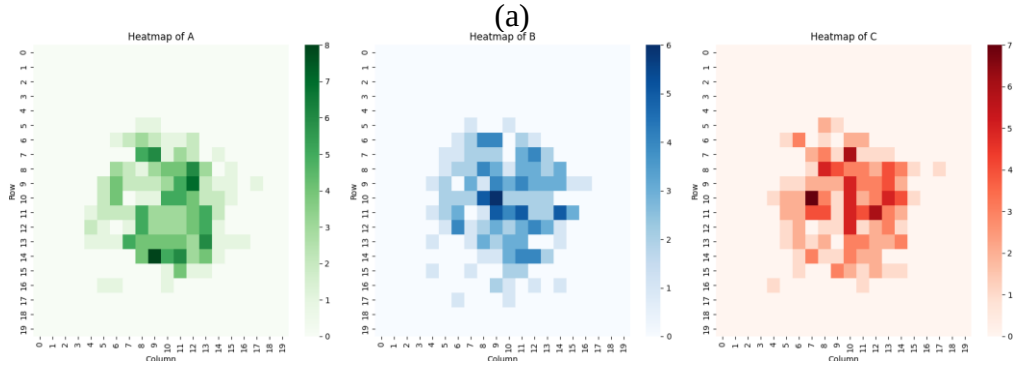
The third interaction combination is a mixture of positive and negative energies shown in Table 5.

Table 5. Interaction energies for combination 3

	A	B	C
A	-10	5	5
B	5	-10	5
C	5	5	-10



(a)



(b)

Figure 11. Configuration of interaction parameter combination 3. (a) Dispersion map of overall configuration. (b) Dispersion map of configuration of units

Under these parameters, the units' behavior will be assembled to the opposite of magnets: the same type attracts, and the opposite type repulses. For this reason, areas of the same dark color are observed in Fig.11b. It shows that the units of the same types are trying to form larger clusters on their own. The incomplete gathering has allowed the possibility of one big cluster to create though it's less condensed compared to combination 1 or 4.

The fourth combination is also all negative but with a larger magnitude of interaction energies as shown in Table 6.

Table 6. Interaction energies for combination 4

	A	B	C
A	-1	-1	-1
B	-1	-1	-1
C	-1	-1	-1

The overall configuration of Combination 4 is more condensed than Combination 1 which is easy to understand as there are more incentives for the program to crush units together. Reflected by the increased number and density of dark-colored blocks, it can be reasonably deduced that the dispersion is more condensed as the magnitude of interaction energy increases.

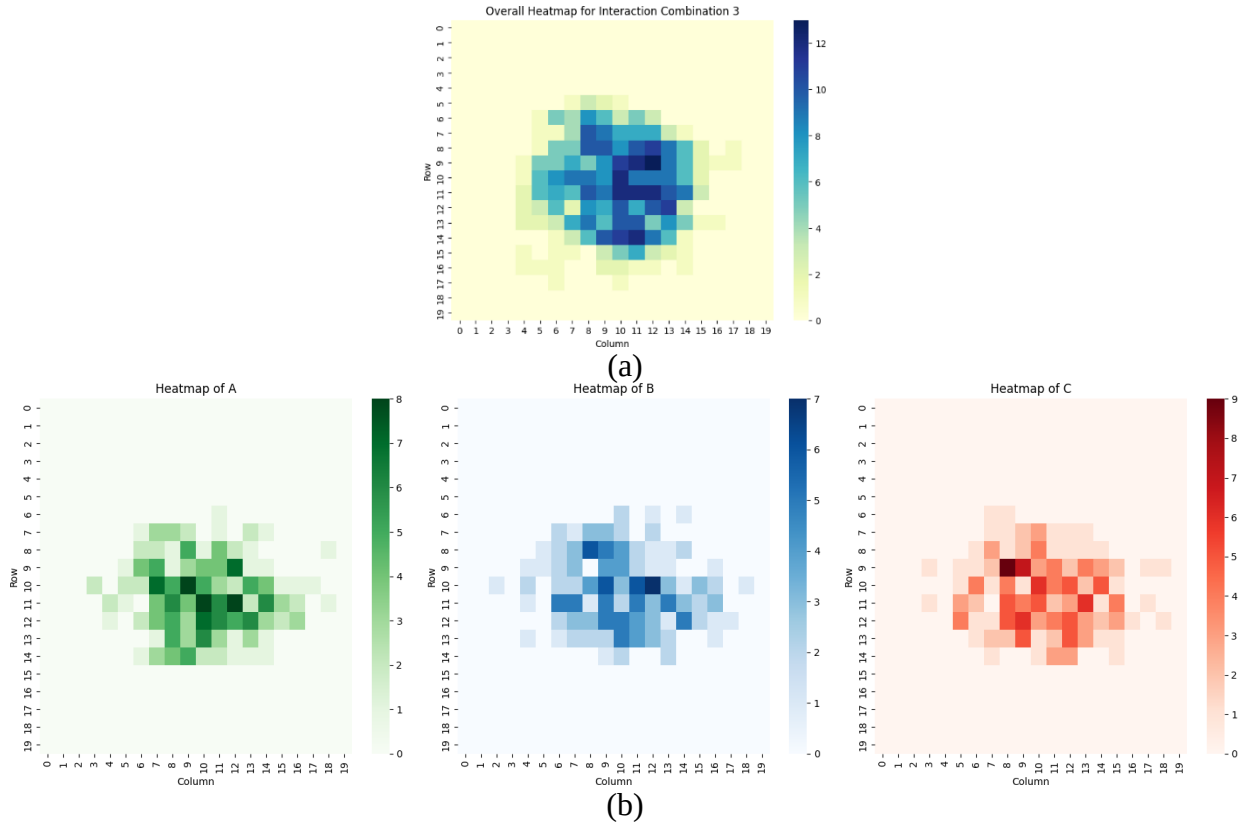


Figure 12. Configuration of interaction parameter combination 4 (a) Dispersion map of overall configuration. (b) Dispersion map of configuration of units

4. Conclusion

This study explored the self-assembly behavior of a lattice system comprising three distinct unit cell agents using simulated annealing. By employing various cooling strategies—exponential, logarithmic, and proportional—our findings reveal that both exponential and proportional cooling strategies effectively reduce energy and facilitate the formation of dense clusters. The logarithmic cooling strategy slowed down the temperature reduction, making it more successful in minimizing energy and leading to more condensed configurations obtained.

The interaction parameters between unit cells significantly influenced the self-assembly process. Negative interactions led to clustering, while positive interactions caused dispersion. Mixed interactions resulted in partial clustering, with same-type units forming larger clusters. These results underline the importance of carefully selecting cooling strategies and interaction parameters to optimize self-assembly processes.

5. APPENDIX

5.1. System Symmetries

The symmetries in the system are defined as follows:

$$up[i,j] == down[i-1,j]$$

Similarly, the interaction strength between spins to the left and right is symmetrical:

$$left[i,j] == right[i,j-1]$$

For boundary conditions, the following assumptions, indicating 3 neighbors for edges and 2 for corners, are made:

$$up[0,j] = down[N-1,j] = left[i,0] = right[i,N-1] = 0$$

5.2. Code

```
import numpy as np
import random
import matplotlib. pyplot as plt
import seaborn as sns
import pandas as pd
from tqdm import tqdm
import copy

import copy
import numpy as np
import random
from tqdm import tqdm
import seaborn as sns
import matplotlib. pyplot as plt

# Define the agent class
class agent:
 def __init__ (self, shape, position):
 self. shape = shape
 self. position = position

 # Create a deep copy of the agent
 def copy(self):
 return agent (copy. deepcopy (self. shape), copy. deepcopy (self. position))

 # Print the agent 's shape
 def print( self):
 for row in self. shape. split('\n'):
 print('  '. join (row))

 # Remove the agent from the lattice
```

```
32  def remove_from_lattice (self, this_lattice):
33  for i, row in enumerate (self. shape. split('\n')):
34  for j, unit in enumerate (row):
35  if unit!='  ':
36  if 0 <= self. position [0] + i < this_lattice. rows and 0 <= self. position [1] + j
< this_lattice. columns:
37  this_lattice. lattice_map [self. position [0] + i][self. position [1] + j] ='  '
38
39  # Place the agent in the lattice
40  def place_in_lattice (self, this_lattice):
41  for i, row in enumerate (self. shape. split('\n')):
42  for j, unit in enumerate (row):
43  if unit!='  ':
44  if 0 <= self. position [0] + i < this_lattice. rows and 0 <= self. position [1] + j
< this_lattice. columns:
45  this_lattice. lattice_map [self. position [0] + i][self. position [1] + j] = unit
46
47  # Check if a movement is valid
48  def is_valid_movement (self, this_lattice, direction):
49  directions = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}
50  curr_x, curr_y = self. position
51  dx, dy = directions. get(direction, (0, 0))
52  new_x, new_y = curr_x + dx, curr_y + dy
53  valid_placement = True
54
55  if this_lattice. is_in_bounds (self, new_x, new_y):


56  self. remove_from_lattice (this_lattice)
57  valid_placement = this_lattice. is_unoccupied (self, new_x, new_y)
58  self. place_in_lattice (this_lattice)
59  else:
60  valid_placement = False
61
62  return valid_placement
63
64  # Move the agent in the given direction
65  def move (self, this_lattice, direction):
66  directions = {'up': (-1, 0), 'down': (1, 0), 'left': (0, -1), 'right': (0, 1)}
67  curr_x, curr_y = self. position
68  dx, dy = directions. get(direction, (0, 0))
69  new_x, new_y = curr_x + dx, curr_y + dy
70  self. remove_from_lattice (this_lattice)
71  self. position = (new_x, new_y)
72  self. place_in_lattice (this_lattice)
73
74  # Rotate the agent's shape by a specified angle
75  def rotate_shape (self, angle):
76  rows = self. shape. split('\n')
77  if angle == 90:
78  rotated_rows = [''. join (row) for row in zip (* rows [:: -1])]
79  elif angle == 180:
```

```
80  rotated_rows = list (reversed (rows))
81  elif angle == 270:
82  rotated_rows = [’’. join (row) for row in zip (*(rows [:: -1]))]
83  else:
84  rotated_rows = rows
85
86  self. shape = ’\n’. join (rotated_rows)
87
88  # Check if a rotation is valid
89  def is_valid_rotation (self, this_lattice, angle):
90  tmp_agent = agent (self. shape, self. position)
91  x, y = tmp_agent. position [0], tmp_agent. position [1]
92  tmp_agent. rotate_shape (angle)
93  self. remove_from_lattice (this_lattice)
94  is_valid = False
95
96  if this_lattice. is_in_bounds (tmp_agent, x, y):
97  is_valid = this_lattice. is_unoccupied (tmp_agent, x, y)
98
99  self. place_in_lattice (this_lattice)
100
101     return is_valid
102
103     # Rotate the agent by a specified angle
104     def rotate (self, this_lattice, angle):
105     self. remove_from_lattice (this_lattice)
106     self. rotate_shape (angle)
107     self. place_in_lattice (this_lattice)
108
109     # Perform a random valid action (move or rotate )
110     def random_action (self, this_lattice, expected_units):
111     valid_actions = []
112     directions = [’up’, ’down’, ’left’, ’right’]
113     for direction in directions:
114     if self. is_valid_movement (this_lattice, direction):
115     valid_actions. append (’move’ + direction)
116
117     angles = [90, 180, 270]
118     for angle in angles:
119     if self. is_valid_rotation (this_lattice, angle):
120     valid_actions. append (’rotate’ + str(angle))
121
122     if len (valid_actions) != 0:
123     action = np. random. choice (valid_actions)
124     initial_units = sum (validate_lattice (this_lattice). values ())
125


126     if action. startswith (’move’):
127     self. move (this_lattice, action. split () [1])
128     else :
129     self. rotate (this_lattice, int(action. split () [1]))
```

```
130
131     current_units = sum (validate_lattice (this_lattice). values ())
132
133     if current_units != expected_units:
134     if action. startswith ('move'):
135     self.                    move                    (this_lattice,
{'up': 'down', 'down': 'up', 'left': 'right', 'right': 'left'}[action. split () [1]])
136     else:
137     self. rotate (this_lattice, (360 - int(action. split () [1])) % 360)
138     return 'no valid action'
139
140     return action
141
142     return 'no valid action'
143
144 # Function to create a random agent
145 def random_agent ():
146     configurations = ['AAA', 'ABC', 'ACB', 'CAB', 'CBA', 'BAC', 'BCA']
147     constituents = random. choice (configurations)
148     shapes = ['line', 'boot']
149     form = random. choice (shapes)
150
151     if form == 'line':
152     angles = ['vertical', 'horizontal']
153     orientation = random. choice (angles)
154     if orientation == 'vertical':
155     shape = constituents [0] + '\n' + constituents [1] + '\n' + constituents [2]
156     if orientation == 'horizontal':
157     shape = constituents [0] + constituents [1] + constituents [2]
158
159     if form == 'boot':
160     angles = ['top - left', 'bottom - left', 'top - right', 'bottom - right']
161     orientation = random. choice (angles)
162     if orientation == 'top - left':
163     shape = constituents [0] + constituents [1] + '\n' + constituents [2]
164     if orientation == 'bottom - left':
165     shape = constituents [0] + '\n' + constituents [1] + constituents [2]
166     if orientation == 'top - right':
167     shape = constituents [0] + constituents [1] + '\n' + '     ' + constituents [2]
168     if orientation == 'bottom - right':
169     shape = ' ' + constituents [0] + '\n' + constituents [1] + constituents [2]

170
171     return agent (shape, (0, 0))
172
173     # Function to create a list of random agents
174 def random_agent_list (num_agents):
175     return [random_agent () for _ in range (num_agents)]
176
177 # Define interaction energy values between different types of units
178 interaction_energy = {
179     ('A',   'A'):   -1,
```

```
180     ('A',   'B'):   -1,
181     ('A',   'C'):   -1,
182     ('A',   ' '):    0,
183     ('B',   'A'):   -1,
184     ('B',   'B'):   4,
185     ('B',   'C'):   -11,
186     ('B',   ' '):    0,
187     ('C',   'A'):   -1,
188     ('C',   'B'):   -11,
189     ('C',   'C'):   4,
190     ('C',   ' '):    0,
191     (' ',   ' '):    0
192 }
193
194   # Define the lattice class


195 class lattice:
196     def __init__ (self, rows, columns, agent_list, interaction_energy):
197     self. rows = rows
198     self. columns = columns
199     self. agent_list = agent_list
200     self. interaction_energy = interaction_energy
201     self. initialize ()
202
203     # Create a deep copy of the lattice
204     def copy (self):
205     new_agent_list = [agent. copy () for agent in self. agent_list]
206     new_lattice = lattice (self. rows, self. columns, new_agent_list, self.
interaction_energy)
207     new_lattice. lattice_map = [copy. deepcopy (row) for row in self.
lattice_map]
208     for i in range ( en (new_lattice. agent_list)):
209     new_lattice. agent_list [i]. position = copy. deepcopy (self. agent_list [i].
position)
210     return new_lattice
211
212     # Check if the agent is within the bounds of the lattice
213     def is_in_bounds (self, this_agent, x, y):
214     return (0 <= x < self. rows and 0 <= y < self. columns and
215     0 <= x + len (this_agent. shape. split('\n')) <= self. rows and
216     0 <= y + len (max( this_agent. shape. split('\n'), key = len)) <= self.
columns)
217
218     # Check if the position is unoccupied
219     def is_unoccupied (self, this_agent, x, y):
220     for i, row in enumerate (this_agent. shape. split('\n')):
221     for j, unit in enumerate (row):
222     if unit != '    ' and self. lattice_map [x + i][y + j] != '    ':
223     return False
224     return True
```

```
225
226     # Initialize the lattice and place agents
227     def initialize (self):
228     self. lattice_map = [[’     ’ for _ in range (self. columns)] for _ in range
(self. rows)]
229     for this_agent in self. agent_list:
230     placed = False
231     attempts = 0
232     while not placed and attempts < 100:
233     x = random. randint (0, self. rows - 1)
234     y = random. randint (0, self. columns - 1)
235     if (x + len (this_agent. shape. split(’\n’)) <= self. rows and
236     y + len (max (this_agent. shape. Split (’\n’), key=len)) <= self. columns):
237     if self. is_unoccupied (this_agent, x, y):
238     this_agent. position = (x, y)
239     this_agent. place_in_lattice (self)
240     placed = True
241     attempts += 1
242     if not placed:
243     print ("Warning: Unable to place agent after 100 attempts.")
244
245     # Calculate the energy of the lattice
246     def calculate_energy (self):
247     energy = 0
248     directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
249     for i in range (self. rows):
250     for j in range (self. columns):
251     for dx, dy in directions:
252     if 0 <= i + dx < self. rows and 0 <= j + dy < self. columns:
253     unit_1, unit_2 = self. lattice_map [i][j], self. lattice_map [i + dx][j + dy]
254     energy += self. interaction_energy. get (( unit_1, unit_2), 0)
255     return energy / 2
256
257     # Print   te lattice with edges
258     def print (self):
259     top_bottom_border = ’-’ * (2 * self. columns + 3)
260     print (top_bottom_border)
261     for row in self. lattice_map:
262     print (f’| {"     ". join (row)} |’)
263     print (top_bottom_border)
264


265 # Different strategies for changing temperature
266 def temperature_change_strategy1 (temp, i):
267     return temp / (1. + temp * 0.01)
268
269 def temperature_change_strategy2 (temp, i):
270     return temp * 0.99
271
272 def temperature_change_strategy3 (temp, i):
273     return temp / np. log (1. * (i + 1))
```

```
274
275 # Check if the system is stuck in a local minima
276 def is_stuck_in_local_minima (energies, tolerance =1 e -2, window = 3):
277     if len (energies) < window:
278     return False
279     recent_energies = energies [-window:]
280     return max (recent_energie ) - min (recent_energies) < tolerance
281
282 # Check if two agents are adjacent
283 def is_adjacent (agent1, agent2):
284     for i, row1 in enumerate (agent1. shape. Split ('\n')):
285     for j, unit1 in enumerate (row1):
286     if unit1 != '     ':
287     x1, y1 = agent1. position [0] + i, agent1. position [1] + j
288     for k, row2 in enumerate (agent2. shape. Split ('\n')):
289     for l, unit2 in enumerate (row2):
290     if unit2 != '     ':
291     x2, y2 = agent2. position [0] + k, agent2. position [1] + l
292     if abs (x1 - x2) <= 1 and abs (y1 - y2) <= 1:
293     return True
294     return False
295
296 # Calculate the center of a cluster of agents
297 def calculate_cluster_center (cluster):
298     x_coords = []
299     y_coords = []
300     for agent in cluster:
301     for i, row in enumerate (agent. shape. Split ('\n')):
302     for j, unit in enumerate (row):
303     if unit != '     ':
304     x_coords. append (agent. position [0] + i)
305     y_coords. append (agent. position [1] + j)
306     center_x = sum (x_coords) // len (x_coords)
307     center_y = sum (y_coords) // len (y_coords)
308     return center_x, center_y
309
310 # Determine the direction to move towards a target point
311 def get_direction_to_target (current, target):
312     directions = ['up', 'down', 'left', 'right']
313     dx, dy = target [0] - current [0], target [1] - current [1]
314     if abs (dx) > abs (dy):
315     return 'down' if dx > 0 else 'up'
316     else:
317     return 'right' if dy > 0 else 'left'
318
319 # Mo
320 def
321     ve clusters of agents towards a target point move_clusters (this_lattice,
target_point): clusters = []
322     visited = set ()
323     for agent in this_lattice. agent_list:
324     if agent. position not in visited:
```

```
    cluster = [agent]
    queue = [agent]
    visited. add (agent. position)
    while queue:
    current = queue. pop (0)
    for other in this_lattice. agent_list:
    if other. position not in visited and is_adjacent (current, other):
    queue. append (other)
    cluster. append (other)
    visited. add (other. position)


    clusters. append (cluster)

    for cluster in clusters:
    cluster_center = calculate_cluster_center (cluster)
    direction = get_direction_to_target (cluster_center, target_point)
    for agent in cluster:
    if agent. is_valid_movement (this_lattice, direction):
    agent. move (this_lattice, direction)

# Validate the lattice configuration
def validate_lattice (this_lattice):
    unit_counts = {'A': 0, 'B': 0, 'C': 0}
    for i in range (this_lattice. rows):
    for j in range (this_lattice. columns):
    unit = this_lattice. lattice_map [i][j]
    if unit in unit_counts:
    unit_counts [ unit] += 1
    return unit_counts

# Perform simulated annealing with cluster movement
def simulated_annealing_with_cluster_movement (initial_configuration, temperature, cooling_rate, max_steps, expected_units, temp_strategy):
    current_configuration = initial_configuration. copy ()
    best_configuration = initial_configuration. copy ()
    energies = []
    last_action = None

    target_point = (initial_configuration. rows // 2, initial_configuration. columns // 2)
    temperatures = []

   for i in tqdm (range (max_steps), desc = "Processing", unit= "iteration"):
    temperature = temp_strategy (temperature, i)
    temperatures. append (temperature)
    for j in range (10):
    new_configuration = current_configuration. copy ()
    my_agent = new_configuration. agent_list [np. random. randint (0, len (new_configuration. agent_list))]
    last_action = my_agent. random_action (new_configuration, expected_units)
```

```
371
372     unit_counts = validate_lattice (new_configuration)
373     total_units = sum (unit_counts. values ())
374     if total_units != expected_units:
375     continue
376
377     current_energy = current_configuration. calculate_energy ()
378     new_energy = new_configuration. calculate_energy ()
379
380     if new_energy < current_energy or np. random. rand () < np. exp
((current_energy - new_energy) / temperature):
381     current_configuration = new_configuration. copy ()
382
383     if new_energy < best_configuration. calculate_energy ():
384     best_configuration = new_configuration. copy ()
385
386     energies. append (current_energy)
387
388     if is_stuck_in_local_minima (energies):
389     move_clusters (current_configuration, target_point)
390
391     unit_counts = validate_lattice (best_configuration)
392     print (f "End of Annealing Process, Units: {unit_counts}")
393
394     return best_configuration, energies, temperatures, True
395
396 # Calculate the entropy and heatmap matrix from final lattice maps
397 def calculate_heatmap_entropy (final_lattice_maps, n_rows, n_columns):
398     heatmap_matrix = np. zeros ((n_rows, n_columns))
399
400     #Accumulate the presence counts in the heatmap matrix
401     for lattice_map in final_lattice_maps:


402     for i in range (n_rows):
403     for j in range (n_columns):
404     if lattice_map [i][j] != ’     ’:
405     heatmap_matrix [i][j] += 1
406
407     # Normalize the heatmap matrix to get a probability distribution
408     total_counts = np. sum (heatmap_matrix)
409     probability_matrix = heatmap_matrix / total_counts
410
411     # Calculate the entropy of the probability distribution
412     entropy = -np. sum (probability_matrix * np. log2 (probability_matrix +
1e -9)) # Add small value to avoid log (0)
413
414     return entropy, heatmap_matrix
415
416 # Plot separate heatmaps for each unit type
417 def plot_separate_heatmaps (final_lattice_maps, n_rows, n_columns):
418     a_matrix = np. zeros ((n_rows, n_columns))
```

```
419     b_matrix = np. zeros ((n_rows, n_columns))
420     c_matrix = np. zeros ((n_rows, n_columns))
421
422     # Accumulate the presence counts in the matrices
423     for lattice_map in final_lattice_maps:
424     for i in range (n_rows):
425     for j in range (n_columns):
426     if lattice_map [i][j] == 'A':
427     a_matrix [i][j] += 1
428     elif lattice_map [i][j] == 'B':
429     b_matrix [i][j] += 1
430     elif lattice_map [i][j] == 'C':
431     c_matrix[i][j] += 1
432
433     # Determine the common vmin and vmax for all heatmaps
434     vmin = min (a_matrix. min (), b_matrix. min (), c_matrix. min ())
435     vmax = max(a_matrix. max (), b_matrix. max (), c_matrix. max ())
436
437     # Plot the heatmaps
438     fig, axs = plt. subplots (1, 3, figsize = (18, 6))
439
440     sns. heatmap (a_matrix, annot = False, fmt = ".1 f", cmap = "Greens",
ax = axs [0], vmin =vmin, vmax = vmax)
441     axs [0]. set_title (' Heatmap of A')
442     axs [0]. set_xlabel ('Column')
443     axs [0]. set_ylabel ('Row')
444
445     sns. heatmap (b_matrix, annot = False, fmt = ".1 f", cmap = "Blues", ax
= axs [1], vmin = vmin, vmax = vmax)
446     axs [1]. set_title ('Heatmap of B')
447     axs [1]. set_xlabel ('Column')
448     axs [1]. set_ylabel ('Row')
449
450     sns. heatmap (c_matrix, annot = False, fmt = ".1 f", cmap = "Reds", ax
= axs [2], vmin = vmin, vmax = vmax)
451     axs [2]. set_title (' Heatmap of C')
452     axs [2]. set_xlabel ('Column')
453     axs [2]. set_ylabel ('Row')
454
455     plt. tight_layout ()
456     plt. show ()
457
458 # set parameters
459 n_agents = 15
460 n_rows = 20
461 n_columns = 20
462 initial_temp = 500
463 cooling_rate = 0.99
464 max_steps = 1000
465 expected_units = n_agents * 3
466 num_combinations = 30
467
```

```
# Generate a random initial configuration to use for all strategies
agent_list = random_agent_list (n_agents)
interaction_energy = interaction_energy


initial_lattice = lattice (n_rows, n_columns, agent_list, interaction_energy)

# Test the three cooling strategies
strategies = [temperature_change_strategy1, temperature_change_strategy2, temperature_change_strategy3]
strategy_names = ["Strategy 1", "Strategy 2", "Strategy 3"]

    for strategy, strategy_name in zip (strategies, strategy_names):
    energy_changes = []
    final_lattice_maps = []
    all_temperatures = []

    for _ in range (num_combinations):
    temperature = initial_temp
    my_lattice = initial_lattice. copy ()

    initial_energy = my_lattice. calculate_energy ()
    print(f 'Initial energy: {initial_energy}')

    best_configuration, energies, temperatures, valid = simulated_annealing_with_cluster_movement (
    my_lattice, initial_temp, cooling_rate, max_steps, expected_units, strategy
    )

    if not valid:
    print ("Simulation stopped due to unit count mismatch.")
    continue

    final_energy = best_configuration. calculate_energy ()
    energy_change = initial_energy - final_energy
    energy_changes. append (energy_change)
    final_lattice_maps. append (best_configuration. lattice_map)
    all_temperatures. append (temperatures)

    print (f 'Final energy: {final_energy}')
    print (f 'Energy change: {energy_change}')
    print ('\n\n')

    # Calculate and plot the entropy of the heatmap
    entropy, heatmap_matrix = calculate_heatmap_entropy (final_lattice_maps, n_rows, n_columns)
    print (f "Entropy of the heat map for {strategy_name}: {entropy}")

    # Plot the heatmap
    sns. heatmap (heatmap_matrix, annot=False, fmt=".1 f", cmap = "YlGn Bu")
```

```
plt. title (f 'Heatmap of Scatterness of Configurations for {strategy_name}')
plt. show ()

# Plot separate heatmaps for A, B, and C
plot_separate_heatmaps (final_lattice_maps, n_rows, n_columns)

# Plot the energy changes of each combination
plt. Plot (range (1, num_combinations + 1), energy_changes, marker='o')
plt. title (f 'Energy Changes for Each Random Parameter Combination ({strategy_name})')
plt. Xlabel ('Combination')
plt. Ylabel ('Energy Change')
plt. grid (True)
plt. show ()

# Average energy change
avg_energy_change = np. mean (energy_changes)
print (f 'Average energy change for {strategy_name}: {avg_energy_change}')

# Plot the temperature change in one iteration
plt. Plot (all_temperatures [0], marker='o')
plt. title (f 'Temperature Change in One Iteration ({strategy_name})')
plt. Xlabel ('Iteration')
plt. Ylabel ('Temperature')
plt. grid (True)
plt. show ()

# Print the best configuration
print (f 'Best configuration for {strategy_name}:')
best_configuration. print ()
```

References

[1] Barz, B., Urbanc, B. Minimal Model of Self-Assembly: Emergence of diversity and complexity. the Journal of Physical Chemistry. B, 2004, 118 (14): 3761–3770.

[2] Troisi A, Wong V, Ratner MA. An agent-based approach for modeling molecular self-organization. Proceedings of the National Academy of Sciences. 2005, 102 (2): 255-60.

[3] Israelachvili JN, Mitchell DJ, Ninham BW. Theory of self-assembly of hydrocarbon amphiphiles into micelles and bilayers. Journal of the Chemical Society, Faraday Transactions 2: Molecular and Chemical Physics. 1976, 72: 1525-68.

[4] Whitesides GM, Kriebel JK, Mayers BT. Self-assembly and nanostructured materials. Nanoscale Assembly: Chemical Techniques. 2005, 217-39.

[5] Kirkpatrick S. Improvement of reliabilities of regulations using a hierarchical structure in a genetic network. Science. 1983, 220: 671-80.

[6] Philp D, Stoddart JF. Self-assembly in natural and unnatural systems. Angewandte Chemie International Edition in English. 1996, 35 (11): 1154-96.

[7] Whitesides GM, Grzybowski B. Self-assembly at all scales. Science. 2002, 295 (5564): 2418-21.

[8] Fortuna S, Troisi A. An artificial intelligence approach for modeling molecular self-assembly: agent-based simulations of rigid molecules. The Journal of Physical Chemistry B. 2009, 113 (29): 9877-85.

[9] Mézard M, Parisi G, Virasoro MA. Spin glass theory and beyond: An Introduction to the Replica Method and Its Applications. World Scientific Publishing Company; 1987 Nov 1.

[10] Isakov SV, Zintchenko IN, Rønnow TF, Troyer M. Optimised simulated annealing for Ising spin glasses. Computer Physics Communications. 2015, 192: 265-71.

[11] Kirkpatrick S. Improvement of reliabilities of regulations using a hierarchical structure in a genetic network. Science. 1983, 220: 671-80.