CMC-15 - Inteligência Artificial

Relatório do Laboratório 1 - O Problema das Quatro Cores no Colorimento dos Estados do Brasil

Grupo:

Gabriel Telles Missailidis - gabriel.missailidis@ga.ita.br

João Lucas Rocha Rolim - joao.rolim@ga.ita.br

Samir Nunes da Silva - samir.silva@ga.ita.br

Professor:

Paulo Marcelo Tasinaffo

05/11/2024

Instituto Tecnológico de Aeronáutica – ITA

São José dos Campos, SP

1 Introdução

O problema das quatro cores é um problema clássico da teoria dos grafos que afirma que, para qualquer divisão plana de regiões adjacentes, é possível colorir o mapa usando no máximo quatro cores de modo que nenhuma região adjacente compartilhe a mesma cor. No contexto atual, o problema das quatro cores possui diversas aplicações práticas, como a alocação de frequências em redes de telefonia móvel e a otimização de recursos em sistemas de alocação de tarefas.

O presente trabalho propõe a aplicação de uma técnica de otimização inspirada em comportamento coletivo — a Otimização por Enxame de Partículas (PSO) — para resolver o problema de coloração de grafos aplicado ao mapa dos estados do Brasil. Para isso, o algoritmo PSO é adaptado para uma versão discret, que permite que as partículas explorem combinações de cores limitadas a quatro valores possíveis. Essa adaptação permite ao PSO lidar com a restrição de coloração do problema, onde a cada estado brasileiro deve ser atribuída uma cor entre quatro opções (vermelho, azul, verde e amarelo) sem que haja conflitos de cores entre estados adjacentes.

O algoritmo implementado é inspirado pelo trabalho de Cui et al. (2008), que propõem um método específico para coloração de grafos planares com uso de PSO discreto. Em vez de uma busca contínua típica do PSO, esta abordagem utiliza funções que probabilisticamente definem as cores em cada iteração. Assim, cada partícula representa uma configuração de cores para o grafo dos estados, e o movimento das partículas no espaço de soluções ocorre por meio de atualizações em suas posições e velocidades, representados por matrizes de adjacência.

Para resolver o problema proposto, desenvolveu-se uma implementação em Python na qual a estrutura do grafo modela as adjacências dos estados, e as partículas buscam soluções de coloração explorando e refinando a solução iterativamente. A partir da adaptação do PSO para este problema, espera-se demonstrar a eficácia do algoritmo de otimização por enxame de partículas como uma ferramenta para resolver problemas complexos e inclusive discretos, tal como o problema das quatro cores.

2 Objetivos

O objetivo do trabalho é resolver, computacionalmente, o problema das quatro cores para os estados do Brasil utilizando o algoritmo de Otimização por Enxame de Partículas (PSO), adaptado para o caso discreto (Quaternary PSO). Objetiva-se gerar uma solução na linguagem Python que resolva, em tempo hábil, o problema proposto e por meio da qual seja possível analisar os resultados de maneira clara, mostrando a eficiência do algoritmo.

3 Metodologia

3.1 Teoria

Para a resolução do problema proposto, baseou-se no artigo "Modified PSO algorithm for solving planar graph coloring problem", de Cui et al.:

- CUI, Guangzhao et al. Modified PSO algorithm for solving planar graph coloring problem. Progress in Natural Science, v. 18, n. 3, p. 353-357, 2008.

Nele, os autores propõem uma adaptação do algoritmo PSO para o caso discreto de coloração de grafos planares. Esse algoritmo adaptado se chama "Quaternary PSO".

Para a modelagem do problema das 4 cores, criou-se, em código, a representação dos nós do grafo, cada qual contendo uma cor (uma dentre 4 cores, a saber, R, B, G e Y - respectivamente - vermelho, azul, verde e amarelo) e um estado do Brasil, e as respectivas conexões entre os estados por meio de arestas, sendo que uma aresta entre dois estados indica que eles são adjacentes.

Com o grafo construído, cada partícula do enxame tem sua posição representada por uma coloração do grafo. Por sua vez, define-se as velocidades das partículas em termos das mudanças das probabilidades dos valores assumidos pelos elementos da solução. Basicamente, trata-se posição e velocidade como grafos no formato de matriz de adjacências e realiza-se as operações com base nessas matrizes.

Nesse contexto, cada nó pode ter um valor de cor: 0, 1, 2 ou 3 (respectivamente, R, B, G e Y). As operações realizadas têm como objetivo realizar mudanças nessas cores de tal forma que o grafo final seja corretamente colorido conforme o problema das 4 cores - isto é - cada estado não pode ter uma cor igual a um de seus estados adjacentes.

Conforme o artigo, as equações de atualização para a velocidade e para a posição de uma dada partícula são dadas pela Figura 2. Nela, rand() representa um número aleatório retirado de uma distribuição uniforme no intervalo [0, 1], r = 0,5, Mod(number, divisor) é a função resto, e a função f é dada pela Figura 2, em que $S(v) = \frac{1}{1+e^{-v}}$ representa a função sigmoide.

$$\begin{cases} v_{id}^{t+1} = w \times v_{id}^{t} + c_1 \times rand() \times (p_{id} - x_{id}^{t}) \\ \qquad\qquad + c_2 \times rand() \times (p_{gd} - x_{id}^{t}) \\ x_{id}^{t+1} = Mod((x_{id}^{t} + f(v_{id}^{t+1})), 4) \end{cases}$$

Figura 1: Fórmulas de atualização da posição e velocidade das partículas do enxame para o algoritmo Quaternary PSO.

$$f(v) = \begin{cases} 0, rand() > r \& rand() < S(v) \\ 1, rand() < r \& rand() < S(v) \\ 2, rand() \leqslant r \& rand() \geqslant S(v) \\ 3, rand() \geqslant r \& rand() \geqslant S(v) \end{cases}$$

Figura 2: Função f para a escolha da variação da cor de um nó do grafo que representa a posição de uma partícula.

Como parâmetros para o algoritmo, deve-se escolher:

- *n_particles*: o número de partículas;
- *w_max*: o peso máximo para a velocidade na fórmula de atualização;

- w_min: o peso mínimo para a velocidade na fórmula de atualização;
- $c1$: a constante c1 da fórmula de atualização da velocidade, que define o peso da melhor posição local obtida até então pela partícula;
- $c2$: a constante c2 da fórmula de atualização da velocidade, que define o peso da melhor posição global obtida até então pelo enxame;
- max_iter: o número máximo de iterações do algoritmo, que define uma das condições de parada.

O peso w é reduzido de w_max para w_min em uma escala linear conforme se passam as iterações do algoritmo. Isso tem como objetivo priorizar, no começo, a exploration, e, no final, a explotation.

3.2 Implementação

Os códigos listados no Apêndice B referem-se à implementação realizada para a resolução do problema proposto.

Conforme explicado anteriormente, a implementação apresentada utiliza o algoritmo Quaternary PSO para resolver o problema das quatro cores aplicado aos estados do Brasil. Ela modela o mapa dos estados como um grafo, onde cada estado é um nó (Node) e as conexões entre estados adjacentes representam as arestas. O grafo principal é definido na classe `BrazilGraph`, que especifica as relações de vizinhança para cada estado, permitindo avaliar se uma coloração dada é válida ou não.

A estrutura de dados dos estados e das cores é baseada em enums (`State` e `Color`). O enum `State` contém uma lista de estados brasileiros, enquanto o enum `Color` define quatro cores (R para vermelho, B para azul, G para verde e Y para amarelo) associadas a valores inteiros. Essas cores são atribuídas aos estados por meio de instâncias da classe `Node`, cada uma representando um estado com uma cor. A classe `BrazilGraph` instancia esses nós e define as adjacências entre eles, permitindo gerar uma matriz de adjacência com as cores atribuídas a cada estado.

O núcleo da implementação está na classe `QuaternaryPSO`, que representa o algoritmo de otimização por enxame de partículas. O PSO é uma técnica inspirada no comportamento de grupos de partículas (ou agentes) que se movem pelo espaço de solução (nesse caso, um grafo a ser colorido) em busca da melhor. No caso, cada partícula é representada pela classe `ColorParticle`, que contém a coloração atual dos estados (posição) e a velocidade da partícula. As partículas exploram o espaço de soluções movendo-se em direção a colorações com menor conflito, avaliadas pela função `fitness` definida em `fitness.py`. A função de fitness soma o número de conflitos, ou seja, pares de estados adjacentes que compartilham a mesma cor. Quanto menor o fitness, melhor a solução, sendo que a solução final é obtida quando fitness = 0.

A classe `ColorParticle` também mantém o histórico da melhor posição encontrada até o momento, armazenado no atributo `best`. Cada partícula atualiza sua posição e velocidade em cada iteração do algoritmo, levando em consideração tanto a melhor solução global encontrada até o momento quanto sua melhor posição individual. A atualização da velocidade considera os parâmetros de ajuste do PSO (pesos w, c1, e c2) e utiliza uma função sigmoide para ajustar a probabilidade de mudar de cor, definida pela função f dentro de QuaternaryPSO._update_particle. Esse ajuste de probabilidade permite uma exploração mais controlada das cores, que no caso têm quatro valores possíveis.

Um ponto importante da implementação é o uso do parâmetro w, que diminui progressivamente a cada iteração, incentivando a convergência do algoritmo ao longo do tempo. Esse

peso é calculado na função _update_w, que reduz o valor de w do máximo (w_max) ao mínimo (w_min) conforme o número de iterações aumenta, promovendo uma maior exploração inicial e uma busca local mais refinada nas últimas iterações. Isso é fundamental para o PSO, já que um peso elevado no início ajuda a explorar o espaço de soluções amplamente, enquanto um peso reduzido nas etapas finais evita que a solução se distancie de uma potencial solução ótima.

Na prática, o método `run` da classe `QuaternaryPSO` executa o algoritmo até alcançar o número máximo de iterações (max_iter) ou até encontrar uma solução sem conflitos de cor (fitness mínimo, que valor 0). Em cada iteração, o método chama `_update_particle` para atualizar a posição e velocidade de cada partícula e, em seguida, avalia se a nova posição tem um fitness melhor. Se uma partícula encontrar uma solução com menos conflitos, essa posição é registrada como a melhor posição global (atributo `_best`). Logs são gerados para monitorar o progresso do algoritmo, especialmente nas mudanças da melhor solução encontrada ao longo das iterações.

Ao final da execução, a implementação exibe a solução com a melhor coloração encontrada. A função `_plot_best` utiliza a biblioteca `networkx` para criar um grafo visual, onde os estados são representados por nós coloridos de acordo com a coloração atribuída pelo PSO, e as conexões representam adjacências entre os estados. Isso permite uma verificação visual da validade da solução e facilita a identificação de conflitos, caso existam. As cores dos nós são configuradas em um mapeamento (`color_map`), onde cada valor inteiro associado a uma cor é convertido para sua representação gráfica.

4 Resultados

Os parâmetros escolhidos para a solução do grafo dos estados do Brasil foram:

```
n_particles=20
w_max=2
w_min=0.8
c1=2
c2=1.8
max_iter=10000
```

Tais parâmetros são propostos no artigo que serviu de base para o presente trabalho, exceto pelo número de partículas, que, no geral, não afeta no resultado final se ele for pelo menos 20.

A Figura 3 mostra o grafo colorido obtido após a utilização do algoritmo Quaternary PSO para a coloração dos estados do Brasil. Deve-se notar que foram fixados os estados randômicos do programa, de tal maneira que o resultado é reprodutível mesmo em chamadas diferentes. O tempo médio do algoritmo foi de aproximadamente 2 minutos e 30 segundos.

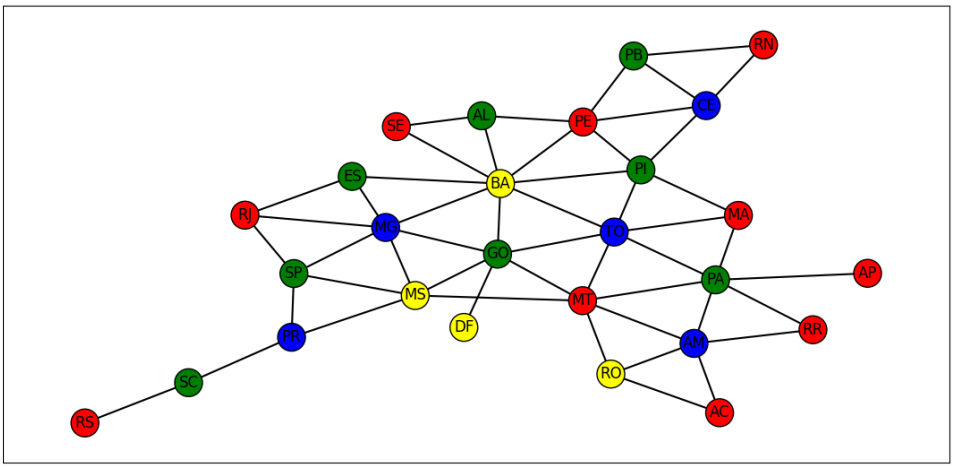

Figura 3: Grafo do mapa do Brasil colorido através do algoritmo Quaternary PSO.

A seguir, lista-se o log das iterações, indicando a melhoria do fitness até o valor chegar a 0, quando o grafo está corretamente colorido. Destaca-se que a iteração 0 possui 4 valores porque, nela, são anexados valores aleatórios de posição (grafos coloridos) para as partículas e então, dentre eles, determina-se o melhor, para então começar de fato as iterações do algoritmo, que são constituídas de atualização de posições, velocidades e melhores posições locais (para cada partícula) e global (para o enxame como um todo).

```
At iteration 0, best global fitness = 22
At iteration 0, best global fitness = 21
At iteration 0, best global fitness = 20
At iteration 0, best global fitness = 18
At iteration 2, best global fitness = 15
At iteration 33, best global fitness = 12
At iteration 48, best global fitness = 11
At iteration 112, best global fitness = 10
At iteration 116, best global fitness = 8
At iteration 189, best global fitness = 6
At iteration 2359, best global fitness = 4
At iteration 2676, best global fitness = 3
At iteration 3848, best global fitness = 2
At iteration 4508, best global fitness = 0
```

Finalmente, os pares estado-cor resultantes, observados na Figura 3, podem ser vistos abaixo na forma de dicionário.

```
{
 <State.RN: 0>: <Color.R: 0>,
 <State.PB: 1>: <Color.G: 2>,
 <State.CE: 2>: <Color.B: 1>,
 <State.PE: 3>: <Color.R: 0>,
 <State.AL: 4>: <Color.G: 2>,
 <State.SE: 5>: <Color.R: 0>,
 <State.PI: 6>: <Color.G: 2>,
```

```
 <State.BA: 7>: <Color.Y: 3>,
 <State.ES: 8>: <Color.G: 2>,
 <State.DF: 9>: <Color.Y: 3>,
 <State.MA: 10>: <Color.R: 0>,
 <State.TO: 11>: <Color.B: 1>,
 <State.GO: 12>: <Color.G: 2>,
 <State.MG: 13>: <Color.B: 1>,
 <State.RJ: 14>: <Color.R: 0>,
 <State.PA: 15>: <Color.G: 2>,
 <State.MT: 16>: <Color.R: 0>,
 <State.MS: 17>: <Color.Y: 3>,
 <State.SP: 18>: <Color.G: 2>,
 <State.RR: 19>: <Color.R: 0>,
 <State.AP: 20>: <Color.R: 0>,
 <State.PR: 21>: <Color.B: 1>,
 <State.AM: 22>: <Color.B: 1>,
 <State.RO: 23>: <Color.Y: 3>,
 <State.AC: 24>: <Color.R: 0>,
 <State.SC: 25>: <Color.G: 2>,
 <State.RS: 26>: <Color.R: 0>
}
```

5 Conclusão

A Figura 3 mostra que o algoritmo PSO adaptado para o caso discreto obteve êxito em colorir o grafo dos estados do Brasil sem conflitos e em tempo hábil. Dessa maneira, o trabalho conseguiu evidenciar que o Quaternary PSO é uma técnica robusta para problemas de coloração de grafos planares. A convergência do algoritmo ao longo das iterações demonstra sua capacidade na exploração de soluções válidas em problemas discretos, como o problema das quatro cores. Finalmente, destaca-se a aplicabilidade de algoritmos de inteligência coletiva, tal como enxame de partículas, em problemas de otimização complexos, bem como valida-se a solução proposta no artigo de Cui et al.

6 Apêndice

6.1 Apêndice A

Utilizou-se a linguagem Python para a codificação do laboratório. O código se encontra no seguinte repositório:

- ⟨https://github.com/Samirnunes/cmc-15-ia/tree/main/lab4⟩

Para rodá-lo, instale o poetry (⟨https://python-poetry.org/docs/⟩) e crie um environment por meio da instalação das dependências:

```
> poetry install
```

Em seguida, partindo da raiz, rode os comandos no terminal:

```
> cd src/lab4
> poetry run python main.py
```

6.2 Apêndice B

6.2.1 color.py

```
from enum import Enum

class Color(Enum):
    R = 0
    B = 1
    G = 2
    Y = 3
```

6.2.2 state.py

```
from enum import Enum

class State(Enum):
    RN = 0
    PB = 1
    CE = 2
    PE = 3
    AL = 4
    SE = 5
    PI = 6
    BA = 7
    ES = 8
    DF = 9
    MA = 10
    TO = 11
    GO = 12
    MG = 13
    RJ = 14
    PA = 15
    MT = 16
    MS = 17
    SP = 18
    RR = 19
    AP = 20
    PR = 21
    AM = 22
    RO = 23
    AC = 24
    SC = 25
    RS = 26

    @classmethod
    def map(cls, i: int):
        return {state.value: state.name for state in cls}[i]
```

6.2.3 graph.py

```
from typing import List
from color import Color
from state import State
import numpy as np
from abc import ABC

class Node:

    _random_state = 0

```

```
    def __init__(self, state: State) -> None:
        self.state = state
        self.color = self._color_from_state()

    def __hash__(self) -> int:
        return self.state.__hash__()

    def _color_from_state(self) -> Color:
        rand = np.random.RandomState(self.state.value + self._random_state)
        return Color(rand.randint(0, 4, 1))

    @classmethod
    def set_random_state(cls, random_state: int):
        cls._random_state = random_state

class ColorGraph(ABC):
    def __init__(self):
        self._graph: dict[Node, List[Node]] = None

    def to_array(self):
        dim = max([key.state.value for key in self._graph.keys()]) + 1
        array = np.full((dim, dim), np.nan)
        for key, nodes in self._graph.items():
            array[key.state.value][key.state.value] = key.color.value
            for node in nodes:
                array[key.state.value][node.state.value] = node.color.value
        return array


class BrazilGraph(ColorGraph):
    def __init__(self, random_state: int):
        Node.set_random_state(random_state)
        self._graph = {
            Node(State.RN): [Node(State.PB), Node(State.CE)],
            Node(State.PB): [Node(State.RN), Node(State.CE), Node(State.PE)],
            Node(State.CE): [Node(State.RN), Node(State.PB), Node(State.PE), Node(State.PI)],
            Node(State.PE): [Node(State.PB), Node(State.CE), Node(State.PI), Node(State.BA), Node(State.AL)],
            Node(State.AL): [Node(State.PE), Node(State.SE), Node(State.BA)],
            Node(State.SE): [Node(State.AL), Node(State.BA)],
            Node(State.PI): [Node(State.CE), Node(State.PE), Node(State.MA), Node(State.TO), Node(State.BA)],
            Node(State.BA): [Node(State.SE), Node(State.AL), Node(State.PE), Node(State.PI), Node(State.TO), Node(State.GO), Node(State.ES)],
            Node(State.ES): [Node(State.BA), Node(State.MG), Node(State.RJ)],
            Node(State.DF): [Node(State.GO)],
            Node(State.MA): [Node(State.PI), Node(State.PA), Node(State.TO)],
            Node(State.TO): [Node(State.PI), Node(State.MA), Node(State.PA), Node(State.MT), Node(State.GO), Node(State.BA)],
            Node(State.GO): [Node(State.BA), Node(State.TO), Node(State.MT), Node(State.MS), Node(State.MG), Node(State.DF)],
            Node(State.MG): [Node(State.BA), Node(State.ES), Node(State.RJ), Node(State.SP), Node(State.MS), Node(State.GO)],
            Node(State.RJ): [Node(State.ES), Node(State.MG), Node(State.SP)],
```

```
            Node(State.PA): [Node(State.MA), Node(State.TO), Node(State.MT),
    Node(State.AM), Node(State.RR), Node(State.AP)],
            Node(State.MT): [Node(State.TO), Node(State.GO), Node(State.MS),
    Node(State.RO), Node(State.AM), Node(State.PA)],
            Node(State.MS): [Node(State.MG), Node(State.GO), Node(State.MT),
    Node(State.SP), Node(State.PR)],
            Node(State.SP): [Node(State.MG), Node(State.RJ), Node(State.MS),
    Node(State.PR)],
            Node(State.RR): [Node(State.PA), Node(State.AM)],
            Node(State.AP): [Node(State.PA)],
            Node(State.PR): [Node(State.MS), Node(State.SP), Node(State.SC)
    ],
            Node(State.AM): [Node(State.PA), Node(State.RR), Node(State.AC),
    Node(State.RO), Node(State.MT)],
            Node(State.RO): [Node(State.MT), Node(State.AM), Node(State.AC)
    ],
            Node(State.AC): [Node(State.AM), Node(State.RO)],
            Node(State.SC): [Node(State.PR), Node(State.RS)],
            Node(State.RS): [Node(State.SC)],
        }
```

6.2.4 fitness.py

```
import numpy as np

MIN_FITNESS = 0

def fitness(array: np.ndarray):
    fitness = MIN_FITNESS
    for i in range(array.shape[0]):
        color = array[i][i]
        for j in range(array.shape[1]):
            if array[i][j] == color and j != i:
                fitness += 1
    return fitness
```

6.2.5 particle.py

```
import numpy as np
from graph import ColorGraph
from fitness import fitness
from dataclasses import dataclass


@dataclass
class Best:
    position: np.ndarray
    fitness: int


class ColorParticle:
    _random_state = 0

    def __init__(self, position: np.ndarray, velocity: np.ndarray):
        self.position = position
        self.velocity = velocity
        self.best = Best(self.position, fitness(self.position))

    def update(self, new_position: np.ndarray, new_velocity: np.ndarray):
        self.position = new_position
        self.velocity = new_velocity
```

```
        self._update_best()

    def _update_best(self):
        current = fitness(self.position)
        if current < self.best.fitness:
            self.best = Best(self.position, current)

    @classmethod
    def from_graph(cls, graph: ColorGraph):
        position = graph.to_array()
        return cls(position, cls._rand_velocity(position))

    @classmethod
    def _rand_velocity(cls, position: np.ndarray):
        rand = np.random.RandomState(cls._random_state)
        cls._random_state += 1
        return np.where(position is not np.nan, rand.uniform(-1, 1, 1), np.
    nan)
```

6.2.6 pso.py

```
import numpy as np
from particle import ColorParticle
from graph import BrazilGraph
from state import State
from color import Color
from fitness import MIN_FITNESS
from logging import getLogger
from logging import StreamHandler
from logging import INFO
import matplotlib.pyplot as plt
import networkx as nx
import sys


class QuaternaryPSO:
    _random_state = 0

    def __init__(
        self,
        n_particles: int = 20,
        w_max: float = 2.0,
        w_min: float = 0.8,
        c1: float = 2.0,
        c2: float = 1.8,
        max_iter: int = 10000,
    ):
        assert w_min < w_max

        self._logger = getLogger(self.__class__.__name__)
        self._logger.setLevel(INFO)
        self._logger.addHandler(StreamHandler(sys.stdout))

        self._particles = [
            ColorParticle.from_graph(BrazilGraph(i)) for i in range(
    n_particles)
        ]
        self._best = self._particles[0].best
        for particle in self._particles:
            self._update_best(particle, 0)
        self._w = w_max
```

```
        self._w_max = w_max
        self._w_min = w_min
        self._c1 = c1
        self._c2 = c2
        self._max_iter = max_iter
        self._r = 0.5

    def run(self):
        iter_count = 0
        while iter_count < self._max_iter and self._best.fitness > MIN_FITNESS:
            self._update_w(iter_count)
            for i in range(len(self._particles)):
                self._update_particle(self._particles[i])
                self._update_best(self._particles[i], iter_count)
            iter_count += 1
        self._plot_best()
        return self._get_result()

    def _get_result(self):
        result: dict[State, Color] = {}
        for i in range(self._best.position.shape[0]):
            result[State(i)] = Color(self._best.position[i][i])
        return result, self._best.fitness

    def _plot_best(self):
        matrix = self._best.position
        color_map = {-1: "gray", 0: "red", 1: "blue", 2: "green", 3: "yellow"}
        G = nx.Graph()
        node_colors = []
        for i in range(matrix.shape[0]):
            node_color = color_map[matrix[i, i]]
            node_colors.append(node_color)
            G.add_node(State.map(i), color=node_color)
        for i in range(matrix.shape[0]):
            for j in range(matrix.shape[1]):
                if matrix[i, j] >= 0 and j != i:
                    G.add_edge(State.map(i), State.map(j))
        plt.figure(figsize=(20, 12))
        pos = nx.kamada_kawai_layout(G)
        nx.draw_networkx_nodes(
            G, pos, node_color=node_colors, node_size=500, edgecolors="black"
        )
        nx.draw_networkx_edges(G, pos, width=1.5)
        nx.draw_networkx_labels(G, pos, font_color="black")
        plt.show()

    def _update_particle(self, particle: ColorParticle):
        def f(velocity: np.float64, rand: np.float64):
            def sigmoid(v: np.float64):
                return 1 / (1 + np.exp(-v))

            sig_velocity = sigmoid(velocity)
            if rand > self._r and rand < sig_velocity:
                return 0
            if rand < self._r and rand < sig_velocity:
                return 1
            if rand <= self._r and rand >= sig_velocity:
```

```
                return 2
            return 3

        rand = self._rand_const()
        new_velocity = np.where(
            particle.position != np.nan,
            np.clip(
                (
                    self._w * particle.velocity
                    + self._c1 * rand * (particle.best.position - particle.position)
                    + self._c2 * rand * (self._best.position - particle.position)
                ),
                -3,
                3,
            ),
            np.nan,
        )

        new_position = np.where(
            particle.position is not np.nan,
            (particle.position + np.vectorize(f)(new_velocity, rand)) % 4,
            np.nan,
        )

        particle.update(new_position, new_velocity)

    def _update_best(self, particle: ColorParticle, iter_count: int):
        if particle.best.fitness < self._best.fitness:
            self._best = particle.best
            self._logger.info(
                f"At iteration {iter_count}, best global fitness = {self._best.fitness}"
            )

    def _update_w(self, iter_count: int):
        self._w = (
            self._w_max - iter_count * (self._w_max - self._w_min) / self._max_iter
        )

    @classmethod
    def _rand_const(cls):
        rand = np.random.RandomState(cls._random_state)
        cls._random_state += 1
        return rand.uniform(0, 1, 1)
```

6.2.7 main.py

```
from pso import QuaternaryPSO

if __name__ == "__main__":
    pso = QuaternaryPSO(
        n_particles=20,
        w_max=2,
        w_min=0.8,
        c1=2,
        c2=1.8,
        max_iter=10000
    )
```

```
12      result = pso.run()
13      print(result)
```