Particle Swarm Optmization (PSO) aplicado ao problema do caixeiro viajante

Arthur Vieira Silva

[1]Departamento de Ciência da Computação
Universidade Federal de São João del Rei (UFSJ) – São João del Rei, MG – Brasil

1. Introdução

O problema do caixeiro viajante (*Travelling Salesman Problem* - TSP) figura como um dos problemas mais famosos da Teoria de Grafos, no qual o objetivo é visitar n cidades, dessa forma, o problema pode ser modelado como um grafo completo com n vértices, em que cada vértice representa uma cidade, e o caixeiro viajante deve realizar um **percurso** (ciclo hamiltoniano) visitando cada cidade apenas uma vez e retornar à cidade de origem. Contudo, o custo do percurso realizado pelo caixeiro viajante deve ser o menor possível e é obtido somando-se os custos individuais (distâncias) entre cada par de cidades.

Entretanto, o problema do caixeiro viajante faz parte da categoria de problemas NP-difícil, isto é, a sua complexidade é exponencial sendo, portanto, inviável encontrar a solução ótima, à medida que o número de cidades cresce, até mesmo pelos computadores mais rápidos. Dessa maneira, como se trata de um problema muito comum e com uma vasta gama de aplicações, há inúmeras heurísticas que podem ser aplicadas ao problema do caixeiro viajante para encontrar a solução ótima (ou uma solução aproximada) em um tempo computacionalmente viável.

Sendo assim, conforme mostrado por Cormen (2002), tomando-se um grafo não direcionado e completo $G = (V, E)$ e que possui um custo inteiro não negativo $c(u, v)$ associado a cada aresta $(u, v) \in E$, é necessário encontrar um ciclo hamiltoniano de G com custo mínimo. Dessa forma, seja $c(A)$ o custo total das arestas no subconjunto $A \subseteq E$:

$$c(A) = \sum_{(u,v)\in A} c(u, v)$$

O algoritmo de Enxame de Partículas (*Particle Swarm Optmization* - PSO) é um ótimo algoritmo para ser adotado tanto para problemas de otimização discretos quanto contínuos para encontrar soluções ótimas. Contudo, tratando-se de problemas discretos como o problema do caixeiro viajante, o PSO deverá sofrer algumas alterações para poder funcionar de maneira adequada, tais mudanças serão descritas a seguir.

2. Enxame de Partículas (PSO)

Particle Swarm Optmization (PSO) é um algoritmo bioinspirado proposto, originalmente, por Kennedy e Eberhart em 1995 e é baseado no comportamento de cardume de peixes e bando de pássaros. A ideia principal é que um indivíduo (partícula) do grupo pode se beneficiar a partir da sua própria experiência e da experiência de todos do grupo. Uma simples analogia seria a de um bando de passáros procurando por alimento em um determinado local, cada pássaro procura pela comida por conta própria mas, todos os membros podem compartilhar a sua posição para saberem o quão próximos estão do alimento.

A ideia central do algoritmo é partir de uma população de possível soluções para o problema (partículas) e movimentar essas partículas a cada iteração no espaço de busca de acordo com fórmulas matemáticas. Além disso, cada partícula possui informações sobre a sua melhor posição e sobre a melhor posição encontrada até o momento considerando todas as partículas. Desse modo, o próximo movimento de uma partícula é baseado na sua melhor posição (informação local) e na melhor posição levando em consideração todas as partículas (informação global). Cada possível solução (partícula) também possui

uma determinada velocidade que representa a sua direção no espaço de busca e deve ser atualizada a cada iteração.

Dessa forma, baseando-se em (CLERC, 2006), podemos sintetizar todas essas informações em algumas equações matemáticas simples. Para isso, algumas informações essenciais do PSO são:

- a dimensão do espaço de busca é D;
- a posição atual de uma determinada partícula nesse espaço em um dado momento t é representada por $x(t)$, com D componentes;
- A sua velocidade atual é representada por $v(t)$;
- a melhor solução encontrada até o momento é chamada de *pbest* (*personal best*) e é $p(t)$;
- a melhor solução considerando todas as partículas é chamada de *gbest* (*global best*) e é representada por $g(t)$.

A partir dessa informações, as equações que representam a atualização da velocidade e da posição de uma partícula em cada dimensão d são, respectivamente:

$$v_d \leftarrow wv_d + c_1 rand()(p_d - x_d) + c_2 rand()(g_d - x_d) \tag{1}$$

$$x_d \leftarrow x_d + v_d \tag{2}$$

Na Equação 1, temos que w é uma constante positiva que representa o peso de inércia (confiança no seu próprio movimento), c_1 e c_2 (constantes positivas) representam, respectivamente, a confiança no seu melhor desempenho (componente cognitiva - individualidade) e a confiança no desempenho do seu melhor informante (componente social - sociabilidade).

Por outro lado, na Equação 2, para atualizar a nova posição de uma determinada partícula, é necessário a posição atual da partícula e a sua velocidade que foi atualizada utilizando-se a Equação 1.

```
inicializar cada particula p

for t = 0: numero maximo de iteracoes (M)
    for i = 0: tamanho da populacao (N)
        if f(p[i]) < f(pbest[i]):
            pbest[i] = p[i]

        if f(p[i]) < f(gbest):
            gbest = p[i]

        atualiza velocidade de p[i]
        atualiza posicao de p[i]

retorne gbest
```

Acima, temos um pseudo-código para exemplificar o funcionamento do PSO. Primeiramente, todas as partículas são inicializadas e o tamanho da população é representado por N e o número de iterações do algoritmo é representado por M. Com isso, a cada

iteração uma função de adaptação (ou *fitness*) foi utilizada para avaliar cada partícula, essa função será descrita posteriormente.

No caso do problema do caixeiro viajante, a função de adaptação, quando aplicada a cada partícula, retorna o custo de um determinado percurso. Verifica-se, assim, se a solução atual de uma partícula é melhor do que a sua melhor solução até o momento e se é melhor do que a melhor solução global até o momento. Em seguida, atualiza-se a velocidade e a posição da partícula com as equações descritas anteriormente. Ao final de todas as iterações, retorna-se o valor da melhor solução encontrada levando em consideração todas as partículas.

2.1. PSO para o problema do caixeiro viajante

Como mencionado anteriormente, o algoritmo PSO é mais aplicável em problemas contínuos, para problemas discretos como o problema do caixeiro viajante, algumas adaptações precisam ser feitas.

Primeiramente, o vetor de velocidade será representado como uma lista na qual cada posição terá uma lista de pares de índices (i, j) onde cada índice representa o índice das cidades que serão trocadas na posição atual da partícula. Há uma permutação de N elementos e o comprimento da lista é $||v||$. Desse modo, quando o operador v for aplicado a uma partícula, uma nova posição será definida para essa partícula:

$$v = ((i_k, j_k)), i_k, j_k \in \{1, ..., N\}, k \uparrow_1^{||v||}$$

Por exemplo, temos uma partícula e o seu respectivo vetor de velocidade:

$$p = (1, 3, 4, 5, 2)$$

$$v = [(1, 4), (2, 5), (3, 1), (4, 2), (3, 2)]$$

Aplicando-se as trocas definidas no vetor de velocidade, teremos a seguinte nova solução para a partícula:

1. $p = (\mathbf{4}, 3, \mathbf{1}, 5, 2)$
2. $p = (4, 3, 1, \mathbf{2}, \mathbf{5})$
3. $p = (4, \mathbf{1}, \mathbf{3}, 2, 5)$
4. $p = (\mathbf{2}, 1, 3, \mathbf{4}, 5)$
5. $p = (\mathbf{3}, 1, \mathbf{2}, 4, 5)$

Além disso, a partir de uma determinada partícula $p = (v_1, v_2, ..., v_n)$, a função de adaptação, quando aplicada sobre essa partícula, será definida da seguinte forma:

$$f(x) = \sum_{i=1}^{n} w_{i,i+1}$$

Com isso, a função de adaptação para o TSP retorna o custo da solução de uma partícula e o seu mínimo global será, consequentemente, a solução ótima para o problema.

3. Implementação

Na implementação desse projeto, foi-se utilizada a linguagem de programação *Python*, pelo fato de se tratar de uma linguagem de fácil entendimento e com uma ampla gama de bibliotecas que podem ser utilizadas para a resolução do problema, ajudando a manipular a estrutura de dados do grafo e a visualizar os dados obtidos de maneira mais prática. As bibliotecas utilizadas na implementação do algoritmo foram a *math* (fornece acesso às funções matemáticas definidas pelo padrão C), *numpy* (para gerar permutações aleatórias de possíveis rotas para o TSP), *random* (para criar números aleatórios) e *itertools* (para gerar todas as permutações possíveis de um iterável).

A respeito do *hardware* usado, o programa foi criado e executado em um computador pessoal com um processador AMD Ryzen 5 3500U com Radeon RX Vega 8, com uma memória principal de 12GB e no Sistema Operacional Windows 11 de 64 bits.

Foram utilizadas 9 funções na implementação do programa que serão listadas a seguir, juntamente com uma breve descrição sobre como elas funcionam e o que elas realizam:

- **ler_arquivo()**: essa função foi utilizada apenas para leitura das linhas do arquivo de entrada fornecido que terá o nome "instancia.txt". Há também um tratamento para a exceção *FileNotFoundError* caso o arquivo de entrada não seja encontrado;
- **distancia_euclidiana(u, v)**: calcula a distância euclidiana entre duas cidades, ou seja, dois pontos no espaço Euclidiano. Essa distância é o segmento de reta que une os dois pontos e será usada para ponderar o grafo com peso nas arestas representando a distância entre cada par de cidades. Considerando duas cidades u e v, a distância euclidiana será dada pela fórmula:

$$d = \sqrt{(v_x - u_x)^2 + (v_y - u_y)^2}$$

- **obter_coordenadas(coordenadas, linhas)**: após a leitura do arquivo de entrada, as coordenadas x e y de cada cidade serão obtidas e armazenadas em uma lista. Também será armazenado o número de cidades do grafo e esse número encontra-se na primeira linha do arquivo de entrada. Por fim, retorna-se coordenadas e numero_cidades;
- **inicializa_grafo(Grafo, numero_cidades, coordenadas)**: função utilizada para inicializar a estrutura de dados do grafo, que será representada como uma matriz de adjacência de tamanho $numero\_cidades$ x $numero\_cidades$. Inicialmente, cada posição da matriz será preenchida com 0s. Em seguida, como o grafo é completo, cada posição da matriz (exceto as que o número da linha é igual ao número da coluna) receberá o valor da distância entre a cidade da linha *i* e da linha *j*, calculada utilizando-se a função **distancia_euclidiana(u, v)** descrita anteriormente;
- **inicializar_particulas(N, numero_cidades)**: inicializa-se a lista de partículas (possíveis soluções para o TSP), utilizando N (número de partículas) permutações possíveis dos valores de 0 à numero_cidades;
- **inicializar_velocidades(N, numero_cidades)**: inicializa o vetor de velocidade com possíveis permutações de tuplas com os valores de 0 à numero_cidades. Essas permutações não possuem valores repetidos e também são colocadas em ordem aleatória no vetor de velocidade usando o método *random.shuffle()*;

- **fitness(Grafo, rota, numero_cidades)**: projeta a função de adaptação (ou fitness) para cada partícula, isto é, calcula-se o custo total do ciclo hamiltoniano para uma dada solução ao chamar esta função;
- **atualizar_particula(particula, numero_cidades, velocidade, w, c1, c2, pBest, gBest)**: função utilizada para atualizar uma determinada partícula com base em trocas nos índices das cidades usando o vetor velocidade e atualizando, assim, a nova solução para a partícula. Parte do vetor de velocidade é mantido utilizando a inércia w, adiciona-se novos índices para serem trocados relacionando-se c1 e pBest e c2 e gBest. Com isso, uma solução para a partícula é obtida com base novo vetor de velocidade e ela é retornada ao final da função;
- **pso(N, M, w, c1, c2)**: Finalmente, essa função realiza chamadas das funções listadas anteriormente. Ela implementa o algoritmo PSO com base no pseudo-código visto antes e retorna, ao final de todas as iterações, o valor encontrada em gBest e o seu respectivo custo. E também o critério de parada definido para o algoritmo é apenas o de atingir o número máximo de iterações M.

4. Resultados

Foram realizados vários testes com o arquivo de entrada fornecido com 12 cidades, executando o programa 3 vezes a cada alteração nos parâmetros e obtendo o custo das soluções em cada caso. O número de iterações M e o número de partículas no enxame N foram alterados e os respectivos parâmetros do PSO também sofreram algumas mudanças.

w	c1	c2	gbest inicial (custo)	gbest final (custo)
0.4	0.7	0.7	41-44-45	39-42-40
0.7	0.7	0.7	41-43-44	41-40-40
0.7	0.8	0.9	41-45-42	41-38-35
0.5	0.75	0.9	40-41-40	40-41-39

w	c1	c2	gbest inicial (custo)	gbest final (custo)
0.4	0.7	0.7	42-44-46	37-37-36
0.7	0.7	0.7	44-43-44	37-34-37
0.7	0.8	0.9	40-44-40	38-37-32
0.5	0.75	0.9	40-45-47	37-36-35

w	c1	c2	gbest inicial (custo)	gbest final (custo)
0.4	0.7	0.7	41-41-41	32-31-32
0.7	0.7	0.7	39-42-44	33-33-34
0.7	0.8	0.9	44-44-44	32-30-29
0.5	0.75	0.9	42-43-41	32-33-31

Inicialmente, na primeira tabela, para um número de iterações e de partículas pequeno com o valor 10, note que com valores baixos para os parâmetros, o valor do gbest no início da execução do algoritmo é muito próximo do seu valor final. E isso se repete

mesmo alterando os valores dos parâmetros, visto que o número de iterações e o tamanho da população são muito pequenos. Apesar disso, a melhor solução foi com um valor alto para c2, já que a influência global prevalece sobre os outros parâmetros.

Na segunda tabela, dobrando o número de partículas no enxame e com 100 iterações, foram encontradas melhores soluções para o problema, a variação entre o gbest inicial e o gbest final já é maior do que os valores da tabela anterior. A melhor solução também foi encontrada com um valor maior para c2.

Por fim, na terceira tabela, aumentando consideravelmente a população para 40 partículas e com 1000 iterações, como o número de iterações e de partículas aumentou bastante, é possível afirmar que foram obtidos valores ainda melhores do que nos outros casos. A diferença entre o custo da melhor solução no início e no final se ampliou ainda mais e, ainda assim, as melhores soluções foram encontradas na terceira linha da tabela, como nos outros casos.

Portanto, nota-se que é melhor utilizar valores mais altos para c1 e c2 mas, optando-se por utilizar um valor maior para c2, haja vista que a sua influência para encontrar a melhor solução leva em consideração todas as partículas e não apenas uma partícula individual. Além disso, igualar os três parâmetros não é muito eficiente porque foram encontradas soluções menos promissoras e o gbest incial não se alterou tanto durante a execução do programa.

5. Conclusão

Verifica-se, portanto, que apesar do problema do caixeiro viajante se tratar de um problema NP-difícil, é possível chegar próximo ou até encontrar a solução ótima utilizando o algoritmo *Particle Swarm Optmization* (PSO), visto que ele proporciona soluções com uma ótima qualidade em um tempo computacionalmente viável. O PSO trata-se de um algoritmo de fácil implementação e que, apesar de precisar sofrer algumas adaptações para o TSP, ainda assim é um excelente mecanismo para tentar solucionar o problema do caixeiro viajante e inúmeros outros problemas de otimização.

Além disso, nota-se que a escolha dos parâmetros utilizados no PSO são fundamentais para se encontrar uma solução ótima ou cada vez mais próxima da melhor solução. É possível fazer várias variações nos valores desses parâmetros, aumentando ou diminuindo a população de partículas no enxame, o número de iterações que o algoritmo irá realizar, os parâmetros cognitivo e social e a ponderação de inércia. As melhores soluções foram encontradas com os valores $w = 0.7$, $c1 = 0.8$ e $c2 = 0.9$, ou seja, $c2$ possui um valor um pouco maior com objetivo de priorizar a informação dada pelo melhor informante de uma partícula em uma determinada iteração do algoritmo PSO.

Referências

CLERC, M. Particle Swarm Optmization. United Kingdom (UK): London.

T. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein. Algoritmos: Teoria e Prática. 3 ed. São Paulo: Elsevier, 2002.