O Problema do Caixeiro Viajante:

Solução por Algoritmo Genético

O problema do caixeiro viajante é um problema clássico em ciência da computação e na área de otimizações em geral. Isso se deve, em parte, por sua importância prática, por sua formulação simples e por sua solução exata ser difícil de ser obtida, já que é um problema classificado como NP-difícil, o que significa que uma solução para o problema não pode ser verificada em tempo polinomial e que este está entre os mais difíceis problemas não polinomiais. O problema pode ser formulado da seguinte forma: “Dada uma lista de cidades e as distâncias entre cada par de cidades, qual é a menor rota possível para se visitar todas as cidades apenas uma vez e retornar à cidade inicial?”.

Dada a dificuldade em se verificar se uma solução para o problema é a ideal, uma vez que o tempo envolvido nessa operação cresce com o tamanho de cidades de forma pelo menos superpolinomial, diversos métodos que buscam soluções aproximadas para o problema foram propostos e são usados frequentemente. Aqui, exploraremos o uso do Algoritmo Genético (AG) para procurar soluções para o problema. O Algoritmo Genético é um método para resolver problemas de otimização que é baseado no processo de seleção natural, o processo que impulsiona a evolução biológica. Para tanto, o AG modifica repetidamente o “código genético” uma população de soluções possíveis para o problema. Em cada etapa, o os indivíduos com maior “aptidão” da população atual serão selecionados para serem pais e os usa para produzir os filhos para a próxima geração. Ao longo de gerações sucessivas, a população "evolui" em direção a uma solução ótima. O AG pode ser aplicado na solução de uma variedade de problemas de otimização que não são adequados para algoritmos de otimização padrão, incluindo problemas nos quais a função objetivo é descontínua, não diferenciável, estocástica ou altamente não linear.

A utilização do AG passa pela identificação de uma codificação adequada do problema de forma que cada solução possível possa ser codificada em um genótipo. Dado o genótipo (“código genético”) de um indivíduo, devemos ser capazes de determinar sua aptidão para aplicar regras evolutivas baseados nesta. Assim, ao gerarmos uma próxima geração de indivíduos, os mais aptos poderão ser clonados e terão ainda prioridade para serem selecionados como pais dos indivíduos da próxima geração. Assim como na reprodução natural, os indivíduos da nova geração, tem seu código genético formado através de combinações (*crossover*) dos códigos genéticos de seus genitores, somado a processos de mutações. As mutações consistem em mudanças localizadas no código genético que foi gerado através do *crossover* dos genótipos dos pais.

Pelo exposto, fica clara a imensa variedade de implementações possíveis e de parâmetros envolvidos no processo. De fato, há inúmeras formas diferentes de selecionarmos os genitores, de combinar seus genótipos, de introduzir mutações e assim por diante. Ademais, o tamanho da população, a forma como a aptidão será considerada e o número de gerações que serão avaliadas terão fortes impactos sobre os resultados encontrados. Assim, para uma compreensão maior do método e de suas potencialidades, explorar essas diversas possibilidades é um excelente exercício.

Implementação:

Iremos considerar N cidades distribuídas num plano cartesiano bidimensional. As coordenadas de cada cidade serão escolhidas aleatoriamente no intervalo $[0,1)$. Assim, sugiro que sejam definidos os vetores x e y, cada um contendo N elementos, com valores no intervalo citado. O caminho a ser percorrido será, então, uma sequência de tamanho N e também será representado por um vetor de tamanho N que chamarei de cam. A sequência mais simples de se definir seria $0, 1, 2, \ldots, N-1$ (já considerando índices dos vetores iniciando em 0), onde o caixeiro percorreria as cidades na ordem definida pela lista inicial e, após chegar à última cidade ($N-1$) o caixeiro retornaria para primeira cidade (0). Usando o pacote numpy, há a opção de embaralhar esses valores para gerar um novo caminho aleatório através do comando *np.random.shuffle()*.

De fato, há outras formas de se codificar o problema, mas a mais simples para nossos objetivos é essa, que na literatura é chamada de *path*. Fazendo desta forma, podemos identificar o próprio vetor *cam* como sendo o genótipo de um indivíduo. A partir deste genótipo, podemos determinar a aptidão de cada indivíduo calculando a distância total que ele percorre.

Para determinar a distância total percorrida, nossa função custo, que desempenhará o papel da aptidão no algoritmo genético, iremos considerar a distância euclidiana entre os pontos. Para facilitar os cálculos, sugiro que seja construído um array (tabela) $N \times N$, que chamarei de $dist$, que contenha a distância entre cada par de cidades. A distância entre a cidade i, localizada nas coordenadas (x_i, y_i), e a cidade j, localizada em (x_j, y_j) será:

$$\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}.$$

Assim, o elemento (i, j) do referido array será igual ao elemento (j, i) e conterá a distância entre as cidades i e j. Perceba que fazendo dessa forma poderíamos trocar a distância entre duas cidades definida acima por outras quantidades que envolvam, por exemplo, detalhes do custo de combustível, a distância realmente percorrida considerando ruas e estradas, etc. De posse dessa tabela, a distância percorrida pelo indivíduo *k* (*apt[k]*), caracterizado pelo genótipo *cam[k]*, que é um vetor, será calculada por:

```
apt[k] = 0
for i in range(N-1):
        apt[k] += dist[cam[k][i],cam[k][i+1]]
apt[k] += dist[cam[k][0],cam[k][N-1]]
```

De posse das distâncias que cada indivíduo percorre, devemos escolher a forma como a próxima geração será formada. Para isso, mecanismos de recombinação e mutação devem ser definidos com base na aptidão dos indivíduos. FIQUEM ATENTOS!!! Da forma que coloquei aqui, quanto menor o valor da variável apt, maior será a aptidão!

Os tipos clássicos de crossover que consideraremos são:

1 - Order Operator (OX)

Proposto por Davis. Constrói um descendente escolhendo uma subsequência do caminho percorrido por um dos pais e preservando a ordem relativa das cidades do outro pai. Para tanto, dois pontos de

corte são escolhidos aleatoriamente no intervalo do tamanho total da caminhada. No exemplo abaixo, os pontos de corte estão representados pelas linhas verticais.

Exemplo:

p1 = (1 2 3 | 4 5 6 7 | 8 9)
p2 = (4 5 2 | 1 8 7 6 | 9 3)

mantém os segmentos selecionados

o1 = (x x x | 4 5 6 7 | x x)
o2 = (x x x | 1 8 7 6 | x x)

A seguir, partindo do ponto do segundo corte de um dos pais, as cidades do outro pai são copiadas na mesma ordem, **omitindo-se** as cidades que estão entre os pontos de corte. Ao se atingir o final do vetor continua-se no início do mesmo. Esse tipo de crossover gera o seguinte resultado.

o1 = (2 1 8 | 4 5 6 7 | 9 3)
o2 = (3 4 5 | 18 7 6 | 9 2)

2 - Partially-Mapped Operator (PMX)

Proposto por Goldberg e Lingle. Constrói um descendente escolhendo uma subsequência do caminho percorrido por um dos pais e preservando a **ordem e a posição** do outro pai de quantas cidades forem possíveis.

Exemplo:

p1 = (1 2 3 | 4 5 6 7 | 8 9)
p2 = (4 5 2 | 1 8 7 6 | 9 3)

- troca os segmentos selecionados

o1 = (x x x | 1 8 7 6 | x x)
o2 = (x x x | 4 5 6 7 | x x)

- define o mapeamento (1-4,8-5,7-6,6-7)

Copia os genes remanescentes em p1 para o1 usando o mapeamento definido:

o1 = (1-4 2 3 | 1 8 7 6 | 8-5 9) = (4 2 3 | 1 8 7 6 | 5 9)

Constrói o2 da mesma forma:

o2 = (4-1 5-8 2 | 4 5 6 7 | 9 3) = (1 8 2 | 4 5 6 7 | 9 3)

3 - Cycle Crossover (CX)

Proposto por Oliver, I.M., D.J. Smith e J. R. C. Holland. Este operador gera os filhos de forma que cada cidade e sua posição venha de um dos pais.

Exemplo:

p1 = (1 2 3 4 5 6 7 8 9)
p2 = (4 1 2 8 7 6 9 3 5)

a partir do primeiro pai, pega-se a primeira cidade e coloca no filho 1.

f1 = (1 x x x x x x x)

a próxima cidade escolhida será a correspondente da primeira posição do pai 2, mas na posição do pai 1, nesse caso é a cidade 4, e ficará na posição (4).

f1 = (1 x x 4 x x x x)

a próxima será a cidade 8, e assim por diante até que teremos

f1 = (1 2 3 4 x x x 8 x)

onde não temos mais escolhas, portanto completamos com as cidades do pai 2

f1 = (1 2 3 4 7 6 9 8 5)

Similarmente,

f2 = (4 1 2 8 5 6 7 3 9)

Os tipos clássicos de mutação que consideraremos são:

1 - Reciprocal Exchange

A mutação de Troca Recíproca sorteia duas cidades e inverte sua posição.

Exemplo:

(1 2 3 |4| 5 6 |7| 8 9) → (1 2 3 |7| 5 6 |4| 8 9)

2 – Inversion

A mutação é feita invertendo-se a ordem dos segmentos selecionados.

Exemplo:

(1 2 3 | 4 5 6 7 | 8 9) => (1 2 3 | 7 6 5 4 | 8 9)

Esse método pode ser implementado da seguinte forma:
Dois índices são escolhidos aleatoriamente, tomado cuidado para não escolher índices iguais, o que não alteraria o caminho. Uma sugestão de código para gerar mutação é[1]:

```
ncam = np.zeros(N,dtype=np.int16)

i=np.random.randint(N)
j=i
while j==i:                          # escolhe j de forma que j ≠ i
   j=np.random.randint(N)
if i>j:
   ini = j
   fim = i
else:
   ini = i
   fim = j

for k in range(N):
   if k >= ini and k <= fim:
      ncam[k] = cam[fin-k+ini]
   else:
      ncam[k] = cam[k]
cam=ncam
```

Tarefa

Vocês devem implementar o algoritmo genético para resolver o problema do caixeiro viajante. Definam, e deixem claro no relatório, quais foram os critérios utilizados para se gerar a próxima geração. Como sugestão e ponto de partida, considerem algo em torno de 20 indivíduos em cada geração. Classifiquem os indivíduos pela ordem de aptidão. No presente caso, quanto menor a distância percorrida, maior será a aptidão. Sugiro que o indivíduo mais apto sempre seja clonado e que alguns indivíduos da próxima geração sejam mutações dele. Além disso, um critério interessante é escolher os pais dos indivíduos da próxima geração com base numa probabilidade diretamente proporcional à sua aptidão.

Sugiro que considerem o número de cidades, $N \in [10,150]$. Mostrem a evolução da menor distância percorrida para um dado número de gerações. Definam critérios de parada baseados tanto na modificação da distância percorrida em um conjunto de gerações quanto no total de gerações consideradas. Ao final, principalmente para as instâncias com menor número de cidades, vocês devem chegar na solução exata do problema, o que algumas vezes pode ser verificado até mesmo de forma visual.

Apresentem suas impressões e conclusões sobre o método em um relatório que deve analisar pelo menos três problemas (conjuntos de cidades) de tamanhos diferentes.

[1] Obviamente devem existir formas muito mais inteligentes, elegantes e eficazes de se fazer isso em Python, mas com meu histórico em Fortran e meu pouco conhecimento de Python essa foi a melhor forma que eu encontrei.

Referências

[1] Larrañaga, P., Kuijpers, C., Murga, R. *et al.* Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators. *Artificial Intelligence Review* **13**, 129–170 (1999).