O Problema do Caixeiro Viajante:

Solução por Simulated Annealing

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 deste tipo. 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 simulated annealing (recozimento simulado) para procurar soluções para o problema. A ideia do simulated annealing tem origem na metalurgia onde metais são aquecidos a temperaturas muito elevadas e lentamente resfriados gerando, ao final, estruturas cristalinas mais bem organizadas que aquelas obtidas com resfriamentos mais rápidos. A ideia por trás do método reside na observação que em muitos problemas de otimização existem muitos mínimos locais separados por grandes barreiras. Assim, métodos como gradiente descendente tendem a ficar aprisionados aos mínimos locais. No simulated annealing, ao considerarmos inicialmente temperaturas altas, teríamos acesso a todo o espaço de soluções. Esperamos, então, que ao reduzir lentamente a temperatura o sistema convirja para a região do espaço de configurações que contém o mínimo global. Obviamente não temos garantia que isso ocorrerá e há a chance de o sistema ficar preso em um mínimo local, mas em geral o método fornece boas soluções mesmo quando não atinge o mínimo global.

Para realizar o simulated annealing, faremos uma simulação de Monte Carlo utilizando o algoritmo de Metropolis reduzindo a temperatura lentamente. Em problemas de otimização, como o aqui proposto, trocamos a energia pela grandeza a ser minimizada. Perceba que se quisermos maximizar uma grandeza, podemos trocar o sinal da grandeza para torná-lo um problema de minimização e assim usar esse método. Neste ponto, já deve estar ficando evidente para vocês um dos pontos mais delicados do uso do simulated annealing, a grande quantidade de parâmetros que devem ser escolhidos numa simulação. Apenas para ilustrar alguns dos pontos mais relevantes vamos elencar:

1) Temperatura inicial;
2) Temperatura final;
3) Protocolo de redução da temperatura (aqui, diversos parâmetros podem estar envolvidos);
4) Número de novas configurações consideradas em cada temperatura;
5) Utilização de reinicializações ou outros métodos auxiliares.

De fato, o simulated annealing pode ser uma ferramenta extremamente eficiente, mas também pode ser extremamente ineficiente. A tendência geral é que esse método é mais adequado para problemas extremamente complexos. No caso do problema do caixeiro viajante, por exemplo, isso indicaria uma geometria complicada e um número muito grande de cidades. Aqui, não nos preocuparemos com a

eficiência do método ou de sua implementação frente a outras formas de se resolver o problema, sendo o objetivo principal apresentá-lo e discutir seus principais aspectos.

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().*

Para determinar a distância total percorrida, nossa função custo, que desempenhará o papel da energia no algoritmo de Metropolis, 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, nossa função custo, $ener$, será calculada por:

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

Assim como foi feito na simulação do modelo de Ising, devemos propor uma nova configuração para o sistema, no caso, um novo caminho a ser percorrido. Enquanto no modelo de Ising trocávamos o sinal de um dos spins da rede, aqui iremos inverter a ordem em que o caixeiro percorre um determinado trecho do caminho. Assim, a tabela abaixo ilustra este processo. Para tanto, dois índices são escolhidos aleatoriamente, $ini = 2$ e $fim = 6$, no exemplo abaixo, com a condição $fim > ini$. Cuidado especial deve ser tomado para não escolher índices iguais, o que não alteraria o caminho.

índice	caminho inicial	caminho final
0	5	5
1	6	6
2	8	2
3	1	9
4	3	3

5	9	1
6	2	8
7	4	4
8	7	7
9	0	0

Uma sugestão de código para gerar essa nova caminhada 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]
```

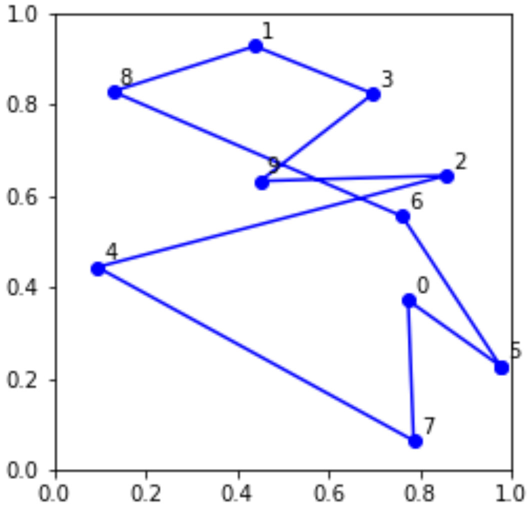
Fazendo dessa forma, a diferença na distância percorrida considerando-se o novo caminho, $ncam$, em relação ao caminho original, cam, depende apenas das modificações nas extremidades do intervalo selecionado. No exemplo dado, dependeria apenas das diferenças relacionadas à ini, fim, e suas vizinhas fora do intervalo, já que os demais trechos permanecem inalterados. Assim, a diferença no tamanho do caminho percorrido pode ser calculada da seguinte forma:

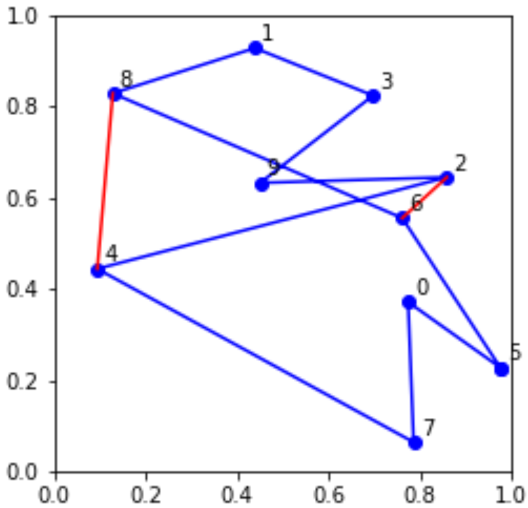
$$esq = ini - 1$$
$$if\ esq < 0: esq = N - 1$$
$$dir = fin + 1$$
$$if\ dir > N - 1: dir = 0$$
$$de = -dist[cam[esq], cam[ini]] - dist[cam[dir], cam[fin]] + dist[ncam[esq], ncam[ini]] + dist[ncam[dir], ncam[fin]]$$

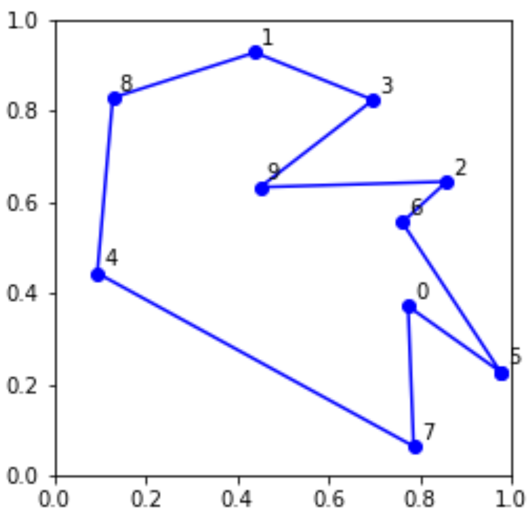
onde $ncam$ é o array que representa novo caminho esq e dir são variáveis auxiliares que indicam as cidades por onde o caixeiro passa antes e após o trecho modificado, respectivamente. No exemplo fornecido teremos $ncam(esq) = cam(esq) = 6$ e $ncam(dir) = cam(dir) = 4$.

[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.

Essa proposta de modificação no caminho está ilustrada na sequência de imagens abaixo







Agora que definimos como os novos caminhos são propostos e como calcular a diferença na distância percorrida, podemos implementar o algoritmo de Metropolis. Um novo caminho deve ser aceito se a distância percorrida diminuir, $de < 0$, ou, se a distância aumentar, o novo caminho deve ser aceito com probabilidade

$$\exp\left(-\frac{de}{T}\right),$$

onde T é a temperatura. Ou seja, se um número aleatório uniformemente escolhido no intervalo [0,1) for menor que a exponencial acima, o novo caminho será aceito.

Como comentado, existem diversos protocolos de redução de temperatura que podemos adotar. Aqui, iremos utilizar um protocolo exponencial, mas fiquem à vontade para experimentar outros. Suponha que a temperatura inicial seja $T_i = 1$. Na forma proposta, iremos realizar 100 passos de Monte Carlo em cada temperatura. Assim, após os 100 primeiros passos de Monte Carlo na temperatura T_i, a temperatura será reduzida multiplicando-a por um fator constante menor que 1, dt, i.e., $T = T * dt$ e mais 100 passos de Monte Carlo serão realizados. Iremos fazer isso até a temperatura atingir um limite inferior T_f. Sugiro que valores nos intervalos $T_i \in [1{,}10]$, $dt \in [0.8{,}0.99], T_f \in [0.005{,}0.0001]$ sejam utilizados. Como ressaltado, o objetivo aqui não é encontrar o melhor conjunto de parâmetros para este problema nem ter um método extremamente eficiente. Ao final desse processo, o caminho deve convergir para o caminho ótimo ou algo próximo a isso.

O que desejamos que aconteça nesse processo é que inicialmente a temperatura seja alta o suficiente para permitir que o caminhante percorra todos os caminhos possíveis. Neste início, o processo seria quase que completamente aleatório. À medida que a temperatura é reduzida, o caráter estocástico deste passeio pelo espaço de configurações vai perdendo força e passamos para um regime de otimização estocástica, onde a energia (distância) vai sendo reduzida em média, mas aumentos na energia ainda são permitidos com probabilidade significante, permitindo acessar outras regiões do espaço de configurações. Ao final do processo, esperamos que apenas uma exploração local da região do espaço de configurações onde o caminhante se encontra ocorra, de forma semelhante a um processo de otimização por gradiente descendente.

Perceba que da forma proposta não temos garantia de encontrar a solução ótima. Uma forma que temos de obter mais confiança na qualidade da solução encontrada é repetir todo o processo, partindo de um novo caminho inicial à temperatura alta e chegando a um caminho "ideal" à temperatura baixa. Pensamos, assim, em realizar várias amostragens do processo, que percorrerá caminhos diferentes devido ao caráter estocástico do processo. Sugiro que considerem algo entre 10 e 50 amostras diferentes para cada problema que propuserem.

A atividade em si:

Uma implementação do método descrito para caminhadas envolvendo N cidades. Está sendo fornecida. Sugiro que considerem $N \in [10{,}150]$. Acompanhem a evolução da distância (energia) no decorrer do processo e veja se está de acordo com o que você esperaria para o método. Trace gráficos de alguns caminhos amostrados durante o processo para ilustrar o caminho percorrido no espaço de configurações. Ao final, determine o melhor caminho a ser percorrido, determinando a distância total percorrida e faça um gráfico mostrando esse caminho para verificar se está condizente com o que você espera. Vocês têm total liberdade na escolha de parâmetros e situações a serem exploradas. Ao final, apresentem suas impressões e conclusões sobre o método.

Além do relatório, vocês devem enviar um programa (em Python para eliminarmos diferenças no tempo de execução devido à linguagem) que lê de um arquivo de texto como o exemplo enviado as posições de um conjunto de cidades e que faça uma estimativa do melhor caminho. O programa deve retornar a distância total do caminho, um gráfico mostrando o caminho e o tempo total de execução. A instância a ser analisada tem entre 100 e 150 cidades com as coordenadas escolhidas no intervalo [0,1), como proposto. Faremos uma espécie de competição onde o grupo (ou grupos) que chegar mais próximo da solução ideal no menor tempo de computação ganhará 3 pontos extra! Sejam criativos na busca por protocolos e formas diferentes de propor novos caminhos.

Uma curiosidade para os mais interessados, algoritmos genéticos são "concorrentes" do método de simulated annealing e também podem ser aplicados a este e outros problemas de otimização com resultados semelhantes.

Algumas referências:

https://pt.wikipedia.org/wiki/Problema_do_caixeiro-viajante
https://en.wikipedia.org/wiki/Travelling_salesman_problem
https://nathanrooy.github.io/posts/2020-05-14/simulated-annealing-with-python/
https://codecapsule.com/2010/04/06/simulated-annealing-traveling-salesman/