Relatório de Inteligência Artificial

Problema do Caxeiro Viajante - TSP

PUC
RIO

Fernando Homem da Costa - 1211971
João Pedro Garcia - 1621161
Júlia Aleixo - 1411397
Rodrigo Leite - 1413150

Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro
INF1771 - Inteligência Artificial
30 de Setembro de 2017

Conteúdo

1. Introdução

Este projeto faz parte da disciplina de Inteligência Artificial (INF1771) do segundo semestre de 2017, ministrada pela professora Renatha Capua. O principal objetivo é abordar um dos problemas clássicos de otimização de combinatória, o problema do caixeiro-viajante. Para isso, utilizaremos dos conceitos aprendidos em sala de aula. A nossa análise é feita através da implementação dos algoritmos, Hill Climbing e Simulated Annealing, e a comparação dos resultados obtidos.

2. Definição do Problema

O problema do caixeiro-viajante é exemplo clássico de problema de otimização combinatória. Esse problema do caixeiro-viajante consiste em encontrar o menor caminho para percorrer uma coleção de cidades e retornar à cidade inicial, visitando cada cidade somente uma vez. Por ser um NP-complexo, a primeira coisa que podemos pensar para resolver esse tipo de problema é reduzí-lo a um problema de enumeração. Suas aplicações abrangem logística, astronomia, fabricação de microchips e sequenciamento de DNA (se modificado ligeiramente).

2.1 Representação das Cidades

Decidimos representar as cidades pelos seus números em um array, e as distâncias entre elas em uma matriz (que, em Python, é uma lista de listas). Apesar de não ser incluída no array da solução, a cidade inicial entra no cálculo da distância do percurso.

2.2 Geração de vizinhança

2.2.1 Função Swap

A partir de um vizinho, troca duas cidades adjacentes de lugar para gerar um novo vizinho. Para criar uma vizinhança inteira, faremos isso ao longo do vetor, trocando suas posições 0 e 1, 1 e 2, e assim por diante até n-1 até n. Essa função foi utilizada para o hillclimbing, pois gera uma quantidade de vizinhos igual ao da dimensão do problema. Assim, avaliar todos os vizinhos e escolher o melhor deles é um processo rápido.

2.2.2 Função Reallocate

Como no simulated annealing não precisamos gerar toda vizinhança, a função seleciona duas cidades aleatórias e troca suas posições. Assim, conseguimos gerar soluções mais diversas do que com a função swap, sem precisar fazer todas as permutações das cidades.

2.3 Avaliação dos vizinhos

A função que avalia os vizinhos calcula a distância de seus percursos de forma circular (depois da ultima cidade ele volta para a cidade inicial para também incluir essa distância) e seleciona o vizinho com a menor distância de percurso.

3. Busca

Decidimos optar por uma busca local, a hill climbing, e uma meta-heurística, simulated annealing. Assim, poderíamos comparar dois tipos de busca que seriam mais prováveis de nos darem resultados satisfatórios, uma vez que as buscas cegas não contém informação alguma sobre o problema, é feito na força bruta, e as buscas heurísticas não se adequam a esse tipo de problema.

3.1 Hill Climbing

Essa busca local começa com uma possível solução, e a partir dela uma vizinhança é gerada. Os vizinhos gerados são então avaliados por uma função que calcula a distância do percurso de sua solução. O vizinho com a menor função de avaliação é escolhido e comparado com a solução inicial: caso tenha uma distância menor, ele substitui a solução inicial e o processo se repete, caso contrario, a busca termina. O problema com esse método são os mínimos locais, dos quais o algoritmo não escapa.

3.2 Hill Climbing com Random Starts

Essa é uma versão modificada do Hill Climbing que tenta evitar máximos/mínimos locais, executando o algoritmo um número definido de vezes. Caso o algoritmo encontre uma solução mas esteja em uma iteração menor que a determinada, ele gera uma nova solução a partir do Hill Climbing, e a compara com a atual. Caso tenha uma distância menor, a nova solução passa a ser a melhor, e esse processo se repete até o número de iterações ser igual ao desejado.

3.3 Simulated Annealing

Essa busca meta-heurística se baseia em probabilidade, e é fundamentada em uma analogia com a termodinâmica. Annealing é um processo térmico usado na metalurgia para obter estados de baixa energia em um sólido. Esse algoritmo funciona de forma análoga a esse processo: ele substitui a solução atual por uma solução de sua vizinhança, escolhida de acordo com uma função objetivo e de uma variável T (de temperatura, que no algoritmo simboliza o tempo). Quanto maior a T, maior a componente aleatória que será incluída na próxima solução escolhida. Conforme a progressão do algoritmo, o valor de T diminui, e mais perto ele está da solução ótima. A vantagem desse algoritmo é que, como ele permite algumas "pioras" na escolha de soluções, ótimos locais são evitados.

4. Metodologia

A linguagem escolhida para implementar as buscas foi Python, pois é uma das linguagens mais usadas para Data Science nos dias de hoje. Além disso, todos no grupo tinham alguma familiaridade com a linguagem. Python tem retornos múltiplos e isso nos permitiu não usar referencias (ponteiros), facilitando a interpretação do nosso código e evitando problema relacionados a vazamento de memória. Ainda, o vetor em Python é implementado como uma lista, e a matriz como uma lista de listas. Assim, não teríamos que nos preocupar com alocação de memória e acesso a ela, e ganhamos em performance. Por isso e a facilidade de transformar o arquivo de entrada em uma matriz, a escolhemos para indicar as distâncias entre as cidades.

4.1 Funções em comum das buscas:

4.1.1 Create Initial Solution

create_initial_solution(dimension): Cria uma solução inicial aleatória com a dimensão do array (número de cidades).

4.1.2 Route distance

route_distance(distance_matrix,cities): Calcula a distância do percurso. Obs: apesar da cidade inicial não ser repetida no final do array de cidades, a distância da última cidade até ela entra nesse cálculo.

4.1.3 Select Traveling Salesman Problem

selectTSP(cwd, filename): Seleciona instância do TSP que vai ser utilizada.

4.1.4 Read Traveling Salesman Problem

readTSP(selected Traveling Salesman Problem): Lê o arquivo do TSP e salva as distâncias em uma matriz.

4.1.5 Write Traveling Salesman Problem

writeTSP(currentWorkingDirectory, selectedTSP, cost, route): Escreve no arquivo de saída a quantidade de cidades, a melhor solução encontrada e sua distância de percurso

4.2 Hill Climbing com Random Starts

Iniciamos nosso projeto com o intuito de aplicar o hill climbing para o problema do caixeiro viajante. Porém, após implementá-lo, vimos que a solução ainda era muito distante da solução ótima. Além disso, o número de iterações feitas para encontrar nossa solução final era muito baixa. Logo, decidimos mudar nosso algoritmo para hill climbing com random starts, em que esse processo seria repetido por um número fixo de vezes. Dessa forma, quando ficássemos presos em um mínimo local, gerávamos uma nova solução inicial aleatória, que aumentava as chances de acharmos a solução ótima.

4.2.1 Evaluate Neighbours

evaluate_neighbours(distance_matrix,neighbours): Seleciona o melhor vizinho, aquele com a menor distância de percurso.

4.2.2 Swap

swap(solution,first_city_number,second_city_number): Troca duas cidades de uma solução.

4.2.3 Create Neighbourhood

create_neighbourhood(initial_solution): Gera a vizinhança de uma solução incial, usando a função swap repetidas vezes.

4.2.4 Hill Climbing

hill_climbing(matrix,dimension): Faz o processo de hill climbing: gera uma solução inicial e partir dela sua vizinhança. Depois, avalia seus vizinhos, escolhendo o melhor (com menor distância), e o compara com a solução inicial. Se for melhor, essa nova solução passa a ser a solução inicial. Enquanto a melhor solução não for a inicial, esse processo se repete.

4.2.5 Single Hill Climbing

single_hill_climbing(matrix,dimension): Configura a semente do hill climbing normal.

4.2.6 Random Starts Hill Climbing

random_starts_hill_climbing(matrix,dimension): Seleciona uma nova solução pelo hillclimbing e a compara com a atual melhor solução. Repete esse processo **1000/(dimensão da matriz)** vezes.

4.2.7 Run For Multiple Seeds Hill Climb

runForMultipleSeedsHillClimb(chosen_tsp) : Configura a semente, executa hill climbing com random starts e retorna tempo de execução, melhor solução encontrada, sua distância e a matriz de distâncias.

4.3 Simulated Annealing

Nós pretendíamos executar o simulated annealing com a mesma função de gerar vizinhança, utilizando a função de swap. Porém, o swap não gerava uma vizinhança grande o suficiente, e o processo entrava em um loop interminável, pois as opções não esgotavam o critério de exaustão. Por isso, substituímos a função swap pelo reallocate, mas realocando somente 2 cidades aleatórias em uma solução. Assim, as possibilidades aumentaram e a diversidade das alternativas também. Consequentemente, nossos resultados foram bem melhores.

Os parâmetros da função (temperatura inicial, α e critério de exaustão) foram testados com vários valores diferentes. Os mais adequados foram: temperatura incial = 10, $\alpha = 0.9995$ e critério de exaustão = 800.

4.3.1 Read Annealing Params

read_AnnealingParams(paramsfile): Lê os parâmetros do simulated annealing.

4.3.2 Kirkpatrick Cooling

kirkpatrick_cooling(start_temp, α): Retorna um modelo de temperatura = $\alpha \times$ tempo, onde $0 < \alpha < 1$ (grau de queda de temperatura), que faz uso do operador yield de python.

4.3.3 P

p(delta,temp): Retorna uma probabilidade P dada por exp(- Δ / temp), $0 < P \leq 1$.

4.3.4 Reallocate

Reallocate(cities): muda duas cidades aleatórias de posição no array.

4.3.5 Get Random Neighbour

getRandomNeighbour(solution): chama o reallocate e retorna o vizinho novo gerado.

4.3.6 Simulated Annealing

simulated_annealing(start_temp, α , exaustion_criteria, matrix, dimension): Faz o processo do simulated annealing: a cada temperatura, visita-se vizinhos ate trocar com um deles. Caso o vizinho for melhor que a solução inicial, ele o substitui. Se não for melhor, é calculado o Δ , que é a diferença entre a solução inicial e a nova. Esse delta e a temperatura são avaliados pela função p, e se o resultado for maior que um número escolhido aleatoriamente, essa solução também é aceita (A chance de aceitar um vizinho pior diminui a cada solução que se aceita). Isso é repetido até que o número de comparações for maior ou igual ao exaustion criteria.

4.3.7 Run Simulated Anneling

runSimulatedAnneling(chosen_tsp, chosen_params): Configura seed, roda annealing e retorna tempo de execução, melhor solução encontrada, sua distância, a matriz de distâncias, temperatura inicial, final e α.

5. Resultados

5.1 Distância

	Hill Climbing	Hill Climbing com Random Starts	Simulated Annealing
17 Cidades	3381	2287	2085
21 Cidades	6196	4963	2987
24 Cidades	2469	2096	1320
48 Cidades	39045	31861	16745
175 Cidades	42323	40882	27181

Tabela 1: Distância x Algoritmo

5.2 Tempo

	Hill Climbing	Hill Climbing com Random Starts	Simulated Annealing
17 Cidades	0.00072 seg	0.02962 seg	1.18098 seg
21 Cidades	0.00116 seg	0.04240 seg	0.06087 seg
24 Cidades	0.00198 seg	0.05687 seg	0.44131 seg
48 Cidades	0.00911 seg	0.19203 seg	0.39301 seg
175 Cidades	0.33427 seg	1.59568 seg	10.18501 seg

Tabela 2: Tempo x Algoritmo

6. Conclusão

A busca que nos trouxe os melhores resultados para todas as instâncias foi o simulated annealing, porém foi mais lento para todas elas. Isso era esperado, pois as buscas meta-heurísticas conseguem chegar a soluções melhores, escapando de ótimos locais. Aceitando pioras baseados em probabilidade, consegue diversificar as soluções no espaço de busca.

Também é interessante notar que poderíamos aumentar o número de starts para o hill climbing, mas não significa que teríamos um resultado melhor. A solução final depende de quão boa é a solução inicial gerada. Existe uma probabilidade muito pequena de, com uma iteração, encontrarmos a solução ótima. Logo, esses resultados precisam ser avaliados repetidamente para conseguirmos julgar o desempenho das buscas.

Além disso, quanto maior o número de cidades, pior nossa solução do Hill Climbing com Random Starts. Isso se deve ao fato de que a quantidade de iterações definidas na função é inversamente proporcional ao número de cidades.