Introdução

O problema do caixeiro viajante é representado por uma pessoa que deseja passar por várias cidades, visitando cada uma delas apenas uma vez, e retornando à cidade inicial. A solução do problema é o caminho de menor distância que pode ser percorrido por esse vendendor. É um problema de otimização NP-difícil e pode ser resolvido por diversas abordagens, sendo representadas nesse trabalho as seguintes:

1. Força bruta, em que são gerados todos os caminhos que podem ser percorridos, calculados seus valores, e por fim retornado o menor deles.

2. Branch and bound, bem parecida com a abordagem de força bruta, apenas com a diferença de que é parado o cálculo de um caminho caso seja menor que o menor caminho encontrado até o momento.

3. Programação dinâmica, em que para encontrar o menor caminho partimos de caminhos de tamanhos menores e incrementamos a cada iteração.

4. Algoritmo genético, em que é feita uma tentativa de simular a evolução, atribuindo um valor que determina o desempenho de uma quantidade fixa de caminhos, e os melhores são utilizados como base para os elementos da próxima iteração.

Implementação

Execução

Apenas compilar e executar normalmente. Na pasta com o algoritmo deve ter um arquivo com o nome input.txt, que possui na primeira linha a quantidade de cidades que terão suas posições lidas, e nas linhas seguintes as posições. Para alterar a quantidade de cidades de uma execução, basta alterar a quantidade presente na primeira linha.

Entrada e saída de dados

Os dados de entrada foram lidos do arquivo input.txt, que contém números entre 0 e 1000. Para alterar a quantidade de cidades do problema atual, basta alterar a primeira linha do arquivo. Ao fim da execução, são gravados em um arquivo com o mesmo nome do programa, porém em extensão txt, a menor distância, o menor caminho encontrado e o tempo de execução em milissegundos no seguinte formato:

Menor caminho: 1 2 3 4 5

Menor distancia: 1000

10.000 ms

Ambiente computacional

Sistema Operacional: Windows 10

Editor de texto: Visual Studio Code

Compilador: GCC

Processador: Intel Core i5-7400 @ 3.00 GHz

Comum a todos algoritmos

Em todos os algoritmos, foi utilizada uma matriz n por 2, com a primeira linha contendo a posição x e a segunda a posição y das cidades lidas. Além dessa, também foi utilizada uma outra matriz para armazenar as distancias calculadas a partir das posições das cidades. Algumas variáveis foram utilizadas para armazenar informações importantes, como o menor caminho e a menor distância. Ao final das execuções, todos os algoritmos terminam de calcular o tempo total gasto e armazenam em um arquivo as informações sobre o menor caminho, distância e tempo de execução.

Força Bruta

Foram geradas todas as permutações possíveis com as n cidades utilizando o método next_permutation da biblioteca <algorithms>. Para cada uma dessas permutações, foi calculada a distância total do caminho a ser percorrido, e comparado com a menor distância encontrada até o momento para determinar qual é a maior distância possível. Fora as matrizes comuns a todos os algoritmos, foi utilizado um vetor cidades contendo a ordem em que as cidades foram percorridas pelo caminho de menor distância para ser impresso ao fim da execução.

Branch and Bound

Foram feitas alterações no algoritmo de força bruta para ir para a próxima permutação caso a distância calculada da permutação atual até o momento já fosse maior que a menor distância encontrada, já que seriam cálculos desnecessários. Assim, se já tivesse sido encontrado um caminho de distância 3000 por exemplo, e a distância percorrida pela permutação atual chegasse a 3200, o cálculo da permutação atual seria interrompido, e iniciaria a próxima.

Programação dinâmica

Foi utilizada uma matriz n por 2^{n-1}, em que as linhas representam uma cidade, e as colunas os subconjuntos aos quais as cidades serão conectadas. Para avaliar os subconjuntos, foram observados os bits da coluna. Assim, a célula [1][6] por exemplo, representa o custo de conectar a cidade 1 até o subconjunto que contém as cidades 2 e 3, que são representadas pelos bits nas posições 2 e 3 (6 = 110). Essas células foram preenchidas em ordem sequencial, considerando o menor custo para acessar as cidades. A primeira linha foi ignorada até a o momento da última iteração, já que apenas o valor da última coluna será alterado para representar o custo de conectar a primeira cidade ao subconjunto que contém todas as outras. A primeira iteração preencheu apenas a primeira coluna, que contém o custo para ir até as cidades iniciais. Depois disso, foram feitas várias iterações, em que é calculado o menor custo para as cidades que não pertencem a um subconjunto se conectarem a ele. Assim, apenas as linhas que não possuem seu bit igual a 1 são alteradas. A terceira coluna por exemplo, representada pelos bits 11, contém como elementos do subconjunto as cidades 1 e 2, e então serão calculadas nessa iteração os valores das células que maiores que 0, e diferentes de 1 e 2. Esse processo é repetido diversas vezes, até chegar a última coluna, quando é feita uma outra iteração para calcular o custo de conectar o resto das cidades de volta à cidade inicial. Foi utilizada outra matriz, também de tamanho n por 2^{n-1}, que contém o número da cidade que foi adicionada ao subconjunto naquela célula, para ser possível obter o

menor caminho a partir da matriz dinâmica posteriormente. Para armazenar os dados nas matrizes foram utilizados dados do tipo short, que possuem apenas 16 bits, tendo em vista que o tempo necessário para executar o algoritmo de força bruta, com quem será comparado, passa a ser inviável após aproximadamente 13 cidades. Caso houvesse a necessidade de fazer testes com uma quantidade de maior de cidades, seria necessário trocar o tipo de dado utilizado por um que possua mais bits.

Algoritmo genético

Para a realização do algoritmo genético foi criado um tipo indivíduo, que contém informações sobre o caminho e a distância percorrida. Inicialmente, é criado um vetor com 10 indivíduos, que possuem as posições iniciais e finais como a cidade inicial, e o resto gerado aleatoriamente. Depois disso, são realizadas 2n iterações, em que há uma chance de ocorrer crossing over, em que parte das últimas cidades percorridas pelo melhor indivíduo são transmitidas para o indivíduo atual, ou mutações, em que ocorre a troca de posição das cidades, dependendo do valor definido no início do arquivo para as variáveis taxaMutação e taxaCrossingOver. Após cada iteração, é armazenado o índice do indivíduo que possui o menor caminho, para que não seja alterado. Após o crossing over e mutações, a distância percorrida pelos indivíduos que foram alterados e recalculada.

Análise de Complexidade

Força Bruta

Melhor Caso

Considerando como operação relevante as operações aritméticas feitas, a complexidade é de O(n!*(n-1)), já que para cada uma das n! permutações são realizadas n-1 somas para calcular a distância percorrida. No melhor caso, o menor caminho está na primeira posição, pois assim não serão feitas alterações no menor caminho.

Pior caso

No pior caso, as permutações geradas estão em ordem decrescente. Dessa forma, além das n-1 adições, cada permutação também realizará n somas (concatenações), para atualizar o menor caminho para a permutação atual, já que da forma como foi implementado para atualizar o menor caminho é preciso concatenar os caracteres da permutação um a um. Assim, a complexidade passa para O (n!*(2n-1)).

Branch and Bound

Melhor caso

As permutações geradas possuem os menores caminhos em ordem crescente de distância. Assim, as próximas iterações executariam menos cálculos por serem interrompidas ao superarem o valor da menor distância encontrada até o momento. Considerando que os valores dos caminhos sejam muito grandes, e superem o valor total da distância da primeira permutação apenas com 1 caminho, com exceção da primeira permutação, que executaria as n adições, as outras fariam apenas uma. Assim, a complexidade passa a ser O(n-1! + n).

Pior caso

No pior caso as permutações sempre possuem distância menor ou muito semelhante à menor distância encontrada até o momento, logo realizam todas as adições como no algoritmo de força bruta. Portanto, a complexidade é de O(n!*n).

Programação dinâmica

É necessário preencher uma matriz de tamanho n por 2^{n-1}. logo a complexidade depende da quantidade de operações realizadas para preencher cada célula. A primeira iteração conta com n-1 operações, já que todas as cidades terão a distância calculada em relação a cidade inicial. As quantidade de operações das iterações intermediárias depende do tamanho do subconjunto, e a última iteração, que conecta o subconjunto das cidades de volta à primeira realiza n-1 cálculos para calcular as distâncias de cada subconjunto possível, portanto, a complexidade é de $O(2^n*n^3)$.

Algoritmo genético

Melhor caso: Os números gerados aleatoriamente sempre são úteis e não ocorrem mutações nem crossing over, assim a quantidade de operações que serão executadas será menor. Por serem executadas n*2 iterações com 10 indivíduos, temos como complexidade O (20n).

Pior caso

No pior caso a execução nunca terminaria por os números gerados aleatoriamente deixarem a execução presa em algum ponto em que existem restrições, como por exemplo depois de iniciar a mutação continuar gerando 0, que não pode ser substituído por ser a cidade inicial.

Testes

Por exigir muito tempo e para poder ser comparado aos outros, a quantidade de cidades utilizadas nos testes foram limitadas pelo algoritmo de força bruta. Além disso, o único que gera saídas diferentes quanto à distância e caminho percorrido é o algoritmo genético. Por isso, foi montado também um gráfico com a média das distâncias em comparação aos outros algoritmos.

Tabelas e gráfico de tempo em milisegundos

Força Bruta

	4	6	8	10	11	12
1	1.703	5429	6617	27538	267475	3073250
2	1.030	2492	1168	26195	290695	3055080
3	563	1053	1074	27348	274296	3223820
4	864	1938	1777	26819	347653	3123800
5	807	545	1120	27435	281834	2971670
6	719	1002	1080	26809	280125	3199000
7	731	4643	2713	26599	285344	2953410
8	1.016	1116	1099	25931	281922	3014670

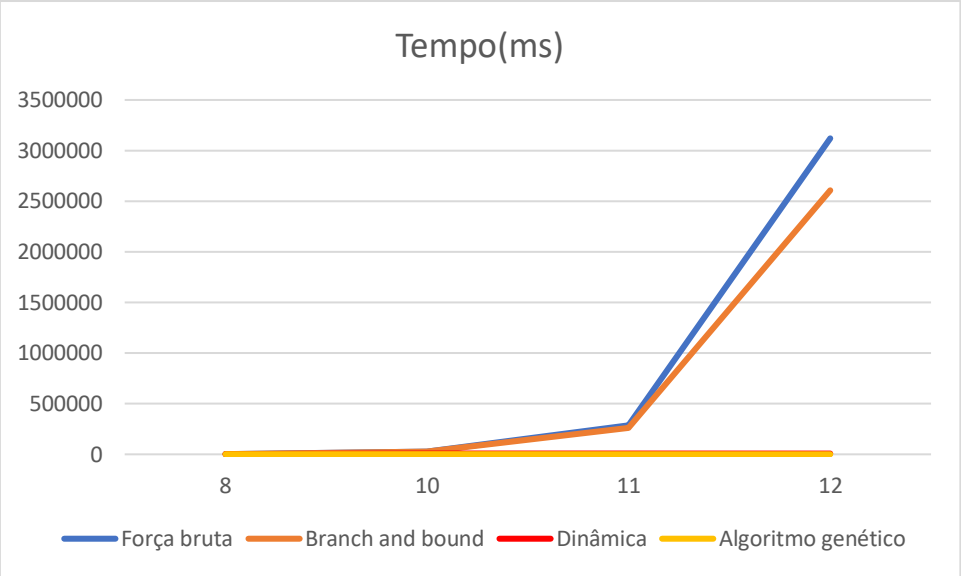
9	791	711	1930	25780	285226	2961670
10	1.298	568	1487	27582	274857	3629760
Média	952	1949	2006,5	26803,6	286942,7	3120613

Branch and Bound

	4	6	8	10	11	12
1	549	2661	1798	24426	378602	2457260
2	824	6836	1191	23898	241730	2538120
3	2084	698	1870	24323	238461	2442390
4	1720	1041	1139	23933	241314	2446660
5	970	1114	3825	24781	239361	2448750
6	762	863	1054	26746	240783	3002440
7	737	2335	818	24096	253946	3135580
8	618	3370	1444	23492	263206	2505040
9	823	894	1119	24259	248243	2446140
10	753	971	1829	25096	244208	2651760
Média	984	2078,3	1608,7	24505	258985,4	2607414

Algoritmo Genético

	4	6	8	10	11	12
1	338	1057	1210	1370	1462	471
2	376	383	378	356	370	361
3	323	321	443	404	341	329
4	312	298	347	318	322	352
5	332	318	560	326	533	351
6	324	489	811	562	356	336
7	325	286	330	464	484	336
8	323	342	300	323	344	327
9	318	432	458	1154	1174	390
10	339	359	445	331	358	331
Média	331	428,5	528,2	560,8	574,4	358,4



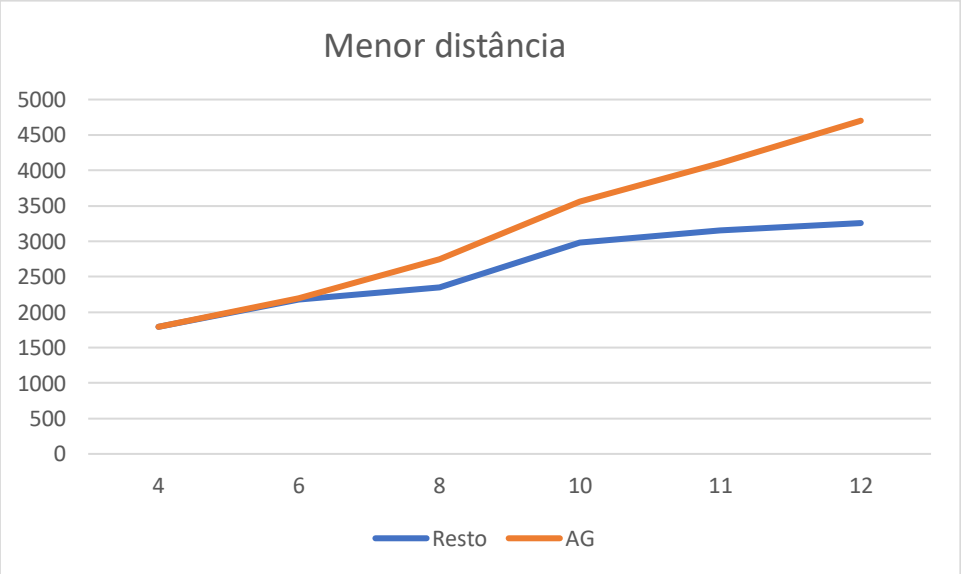
Distâncias e caminhos (Algoritmo genético)

A coluna abaixo do número de cidades representa a distância, e a coluna à direita o caminho percorrido.

	4		6		8	
1	1793	1 2 4 3	2175	1 6 3 4 5 2	2823	1 4 3 7 6 5 8 2
2	1793	1 2 4 3	2175	1 6 3 4 5 2	2847	1 2 8 4 3 7 5 6
3	1793	1 3 4 2	2175	1 2 5 4 3 6	2788	1 5 2 8 6 3 7 4
4	1793	1 3 4 2	2175	1 2 5 4 3 6	2830	1 3 7 4 8 2 5 6
5	1793	1 3 4 2	2175	1 6 3 4 5 2	2658	1 5 8 2 6 3 7 4
6	1793	1 2 4 3	2175	1 6 3 4 5 2	2788	1 5 2 8 6 3 7 4
7	1793	1 2 4 3	2289	1 3 4 5 2 6	2524	1 6 7 3 4 5 2 8
8	1793	1 2 4 3	2289	1 3 4 5 2 6	2524	1 6 2 8 5 4 3 7
9	1793	1 2 4 3	2175	1 2 5 4 3 6	2898	1 4 3 7 6 5 2 8
10	1793	1 2 4 3	2175	1 2 5 4 3 6	2781	1 6 2 8 4 7 3 5

	10		11		12	
1	3482	1 6 2 8 9 5 3 4 10 7	4248	1 11 9 4 10 3 7 6 5 2 8	4969	1 10 7 3 6 5 4 9 12 2 8 11

2	3806	1 6 3 7 9 5 4 10 2 8	4323	1 11 8 2 9 7 10 5 6 3 4	4607	1 2 3 7 10 5 4 12 9 11 8 6
3	3386	1 6 2 8 9 7 10 3 4 5	3900	1 8 2 11 5 10 4 3 7 9 6	4641	1 4 12 10 3 7 6 11 8 9 5 2
4	3511	1 8 2 9 7 10 3 4 5 6	3877	1 3 10 9 11 8 2 5 7 4 6	4725	1 12 10 9 7 3 6 8 11 2 5 4
5	3528	1 9 4 3 10 7 5 8 2 6	3939	1 4 9 11 2 8 5 6 3 7 10	4795	1 8 10 12 9 4 3 7 2 11 5 6
6	3749	1 6 5 2 8 9 7 4 10 3	4805	1 8 10 9 5 11 4 3 7 6 2	4882	1 9 12 10 6 3 4 5 11 2 8 7
7	3397	1 6 3 7 10 4 5 8 2 9	3760	1 2 8 5 11 4 9 10 7 3 6	4577	1 6 8 5 3 7 10 12 11 2 9 4
8	3837	1 3 10 9 5 8 2 6 4 7	3866	1 6 8 11 2 5 9 3 4 7 10	4630	1 6 9 4 12 10 7 3 8 5 2 11
9	3486	1 2 8 5 9 10 7 6 3 4	4024	1 4 11 9 10 7 3 6 2 8 5	4821	1 9 7 6 8 11 2 5 12 10 3 4
10	3456	1 5 9 8 2 6 3 10 7 4	4272	1 4 9 5 6 7 10 3 11 8 2	4363	1 3 4 7 10 9 8 2 11 5 2 12 6



Conclusão

Os algoritmos de força bruta e branch and bound exigem muito tempo para executarem, os tornando inviáveis para a maioria dos casos. Ainda assim, a implementação

utilizando programação dinâmica foi bem mais difícil, tanto que falhei miseravelmente nas duas primeiras tentativas de implementa-lo, e exigiu bem mais pesquisa e tempo do que os outros. Não considero que o algoritmo genético tenha sido tão complexo de implementar comparado ao de programação dinâmica, mas para se obter um bom desempenho é preciso estudar muito e testar muitos ajustes, tanto que não considero o desempenho do algoritmo genético que apresentei muito satisfatório. Assim, na maioria dos casos considerado ser melhor utilizar heurísticas e outras técnicas mais simples que trazem bons resultados em grande partes das vezes, como a do vizinho mais próximo, ao invés de gastar tanto recurso para conseguir uma solução ótima.

Bibliografia

https://en.wikipedia.org/wiki/Held%E2%80%93Karp_algorithm

https://www.geeksforgeeks.org/travelling-salesman-problem-set-1/

https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/?ref=rp