TP2 - Alg 2

Victor Prates

10/12/2023

1 Introduction

O Problema do Caixeiro Viajante (PCV) é um desafio clássico em otimização combinatória e teoria dos grafos. Ele se resume da seguinte forma: dado um conjunto de cidades e as distâncias entre cada par de cidades, o objetivo é encontrar o caminho mais curto que visite cada cidade exatamente uma vez e retorne à cidade de origem.

Em termos mais simples, o caixeiro viajante precisa encontrar a rota mais eficiente para percorrer todas as cidades, minimizando a distância total percorrida. Este problema é conhecido por ser NP-difícil, o que significa que não se conhece um algoritmo eficiente para resolvê-lo em tempo polinomial para instâncias grandes, a menos que P seja igual a NP (um problema não resolvido na teoria da computação). Como resultado, muitas abordagens para o PCV envolvem a busca por soluções aproximadas em vez de soluções exatas.

Tendo isso em vista, nesse trabalho iremos analisar os diferentes algoritmos ligados ao PCV, sendo um de solução exata e dois de solução aproximada.

2 Desenvolvimento

Demos início ao trabalho definindo a linguagem que usariamos, como o problema é NP-Difícil e o Branch And Bound poderia no pior caso ter $O(2^{n+1} - 1)$ iterações necessitavamos que nosso programa executasse o mais rápido possível para conseguirmos percorrer mais nós antes que o limite de 30 minutos por problema fosse atingido. Tendo isso em mente realizamos todo o projeto em C++, essa linguagem por ser compilada nos concede um ganho computacional muito grande em relação ao Python, a outra opção permitida.

Primeiramente, iniciamos o projeto pela representação do mapa especificado em cada Dataset como grafos não orientados, além disso era necessário o calculo da distância Euclidiana. Essa tarefa foi feita por meio da implementação de uma classe City, que possui o id da cidade, suas coordenadas e como calcular a distância para outra City. Com essa implementada criamos outra classe Map, que possui uma matriz N x N, sendo N o número de cidades, que possui a distância entre cada vértice e a distância __INT64_MAX__ quando i = j, para evitar que haja a escolha de loopings entre si mesmo nos algoritmos.

2.1 Branch And Bound

Demos início a implementação dos algoritmos pelo algoritmo de Branch And Bound. Esse é um método de otimização utilizado para resolver problemas de programação inteira, nos quais as variáveis de decisão devem assumir valores inteiros. Ele divide o espaço de busca em subproblemas menores, explorando sistematicamente diferentes soluções candidatas e utilizando técnicas de eliminação para evitar a exploração desnecessária de certas regiões do espaço de busca.

O processo básico do Branch and Bound envolve a criação de uma árvore de busca, onde cada nó representa um subproblema. À medida que a busca avança, alguns ramos da árvore são eliminados com base em limites inferiores e superiores, conhecidos como "bounds". Isso permite uma exploração mais eficiente do espaço de busca, reduzindo o número de soluções candidatas a serem avaliadas.

Para ele criamos a classe Node que abstraisse os nós de uma árvore, ela possui como atributos a profundidade do nó, o valor do bound desse nó, o custo que esse nó já possui e uma lista que abstrai o caminho feito na solução, além disso sobrecarregamos o operador ¡ para podermos compará-los, priorizando sempre o mais profundo e como desempate o que possui menor bound.

Em sequência demos início a implementação do algritmo começando pela função de Bound, para isso criamos um arranjo de arranjos que armazena as duas menores arestas que cada vertice, visando reduzir as buscas. Na função de estimativa verificamos se já definimos alguma aresta à esse vértice, caso contrário usamos os menores vértices.

Com a estimativa já implementada implementamos a função de Branch And Bound seguindo fielmente a abordagem vista em aula. Na nossa versão implementamos a busca em profundidade.

2.2 Twice Around The Tree

Seguimos para a implementação do Twice Around The Tree, essa heurística consiste em duplicar as arestas de uma árvore geradora mínima (Minimum Spanning Tree - MST) para formar um ciclo euleriano, e depois aprimorar essa solução através da aplicação de um algoritmo de refinamento.

Em termos simples, o TATT começa construindo uma árvore geradora mínima no grafo das cidades. Em seguida, duplica as arestas dessa árvore para criar um ciclo euleriano (um ciclo que passa por cada aresta exatamente uma vez). Por fim, utiliza técnicas de otimização, para aprimorar a solução, eliminando cruzamentos entre as arestas e melhorando a eficiência do percurso.

Demos início a implementação pelo algoritmo de geração da AGM, usando o algoritmo de Prim, retornando uma lista em que cada no erá representado por um uma posição e seu valor representava seu pai, sendo que a raiz sempre era o vértice zero. A partir desse, criamos um vetor de vetores de inteiros que cada vertice era representado por uma posição e vetor de inteiros armazenavam seus filhos. Com essa estrutura em mãos era necessário saber como percorre-la

e para isso criamos uma função recursiva que removia todo nó recém visitado e sempre que ela encontrava uma folha era se ligava que fosse filho de algum de seus antecessores. Assim garantindo a poda dos vértices repetidos uma vez que sempre passamos por destinos inéditos.

2.3 Christofides

Finalmente, implementamos o algoritmo de Cristofides, esse é uma heurística projetada para resolver o Problema do Caixeiro Viajante (PCV), que busca encontrar o caminho mais curto que visita todas as cidades exatamente uma vez. Proposto por Nicos Christofides em 1976, o algoritmo oferece soluções aproximadas com garantias teóricas.

O processo inicia com a construção de uma Árvore Geradora Mínima (MST) no grafo das cidades, conectando todas as cidades com o mínimo custo possível. Em seguida, são identificadas as arestas de grau ímpar na MST, e são adicionadas arestas extras para formar um emparelhamento perfeito mínimo, garantindo que todos os vértices tenham grau par.

Um ciclo euleriano é formado percorrendo cada aresta da MST exatamente uma vez, incluindo as arestas adicionadas anteriormente. Esse ciclo é então convertido em um circuito hamiltoniano, que é o objetivo final do PCV, eliminando repetições de vértices.

Com o algoritmo de Prim já implementado para o método anterior, pudemos focar no matching perfeito de pesos mínimos. Usamos o algoritmo Hungaro para realiza-la, uma vez que esse é capaz de unir os pares de vértice que minimizam a soma de pesos. Ele nos retorna então uma matriz de na qual há apenas zeros exceto quando os indices do vertice na matriz tiverem um matching.

Com os vertices ja coletados adicionamos os pares de vertices no vetor de filhos. Usamos o mesmo algoritmo que usamos para percorrer a AGM no problema anterior e temos nossa estimativa.

3 Análise de Resultados

Dataset	Estimativa	Limiar	Aproximação
a280	3562.330000	2579	1.381283
berlin52	8251.760000	7542	1.094108
bier127	141998.000000	118282	1.200504
brd14051	NA	468942	NA
ch130	7772.550000	6110	1.272103
ch150	8161.890000	6528	1.250290
d1291	62923.600000	50801	1.238629
d15112	NA	1564590	NA
d1655	NA	62128	NA
d18512	NA	644650	NA

d198	19061.200000	15780	1.207934
d2103	NA	79952	NA
d493	40612.400000	35002	1.160288
d657	60818.800000	48912	1.243433
eil101	835.568000	629	1.328407
eil51	528.078000	426	1.239620
eil76	640.579000	538	1.190667
fl1400	26158.500000	20127	1.299672
fl1577	NA	22204	NA
fl3795	NA	28723	NA
fl417	14592.500000	11861	1.230293
fnl4461	NA	182566	NA
gil262	3061.940000	2378	1.287611
kroA100	27243.900000	21282	1.280138
kroA150	31627.100000	26524	1.192396
kroA200	38697.000000	29368	1.317659
kroB100	28552.400000	22141	1.289571
kroB150	33559.600000	26130	1.284332
kroD100	24980.200000	21294	1.173110
kroE100	25257.300000	22068	1.144521
lin105	17540.500000	14379	1.219869
lin318	51286.600000	42029	1.220267
linhp318	51286.600000	41345	1.240455
nrw1379	68531.300000	56638	1.209988
p654	41224.100000	34643	1.189969
pcb1173	71248.300000	56892	1.252343
pcb3038	NA	137694	NA
pcb442	60765.000000	50778	1.196680
pr1002	328279.000000	259045	1.267266
pr107	45990.300000	44303	1.038085
pr124	66585.600000	59030	1.127996
pr136	120307.000000	96772	1.243201
pr144	62573.500000	58537	1.068956
pr152	85907.300000	73682	1.165920
pr226	95913.500000	80369	1.193414
pr2392	NA	378032	NA
pr264	58288.600000	49135	1.186295
pr299	64433.800000	48191	1.337050
pr439	129889.000000	107217	1.211459
pr76	NA	108159	NA
pr76	128935.000000	108159	1.192088
rat195	2586.970000	2323	1.113633
rat575	8212.320000	6773	1.212508
rat783	10849.100000	8806	1.232012
rat99	1566.230000	1211	1.293336
rd100	9906.020000	7910	1.252341

rd400	18453.300000	15281	1.207598
rl11849	NA	920847	NA
rl1304	316555.000000	252948	1.251463
rl1323	350844.000000	270199	1.298465
rl1889	NA	316536	NA
rl5915	NA	565040	NA
rl5934	NA	554070	NA
st70	841.921000	675	1.247290
ts225	152494.000000	126643	1.204125
tsp225	4986.680000	3919	1.272437
u1060	284019.000000	224094	1.267410
u1432	NA	152970	NA
u159	49021.800000	42080	1.164967
u1817	NA	57201	NA
u2152	NA	64253	NA
u2319	NA	234256	NA
u574	46219.400000	36905	1.252389
u724	51202.400000	41910	1.221723
usa13509	NA	19947008	NA
vm1084	301564.000000	239297	1.260208
vm1748	NA	336556	NA

Table 1: Branch And Bound

Como vemos a cima o Branch And Bound por buscar a solução exata apresentou grande dificuldade no cálculo do custo mínimo para problemas maiores. Entretanto, nos 30 minutos permitidos para a execução notamos que nas instâncias que houveram resultados parciais a aproximação estava bem próxima do resultado ótimo. Isso é perceptível devido ao valor que mais se distanciou da resposta correta ser apenas 30% maior.

Dataset	Estimativa	Limiar	Aproximação
a280	3484.850000	2579	1.351241
berlin52	10403.900000	7542	1.379462
bier127	155481.000000	118282	1.314494
brd14051	645836.000000	468942	1.377219
ch130	8279.760000	6110	1.355116
ch150	9000.970000	6528	1.378825
d1291	74472.200000	50801	1.465959
d15112	2215050.000000	1564590	1.415738
d1655	82869.600000	62128	1.333853
d18512	889766.000000	644650	1.380231
d198	19503.500000	15780	1.235963
d2103	124582.000000	79952	1.558210

d493	45866.300000	35002	1.310391
d657	65936.400000	48912	1.348062
eil101	860.865000	629	1.368625
eil51	615.352000	426	1.444488
eil76	749.275000	538	1.392704
fl1400	28662.600000	20127	1.424087
fl1577	30288.800000	22204	1.364115
fl3795	39783.300000	28723	1.385068
fl417	16068.600000	11861	1.354742
fnl4461	250755.000000	182566	1.373503
gil262	3343.980000	2378	1.406215
kroA100	30516.900000	21282	1.433930
kroA150	38762.600000	26524	1.461416
kroA200	40235.300000	29368	1.370039
kroB100	28803.500000	22141	1.300912
kroB150	35292.400000	26130	1.350647
kroB200	40619.900000	29437	1.379893
kroC100	27632.800000	20749	1.331765
kroD100	28599.100000	21294	1.343059
kroE100	30978.600000	22068	1.403779
lin105	21170.800000	14379	1.472342
lin318	60989.100000	42029	1.451119
linhp318	60989.100000	41345	1.475126
nrw1379	77169.400000	56638	1.362502
p654	49628.400000	34643	1.432566
pcb1173	81895.200000	56892	1.439485
pcb3038	195796.000000	137694	1.421965
pcb442	71280.800000	50778	1.403773
pr1002	351431.000000	259045	1.356641
pr107	55954.600000	44303	1.262998
pr124	82762.800000	59030	1.402046
pr136	146986.000000	96772	1.518890
pr144	77703.900000	58537	1.327432
pr152	90290.600000	73682	1.225409
pr226	114700.000000	80369	1.427167
pr2392	528148.000000	378032	1.397099
pr264	66408.400000	49135	1.351550
pr299	66098.900000	48191	1.371603
pr439	144156.000000	107217	1.344526
pr76	147670.000000	108159	1.365305
rat195	3265.080000	2323	1.405545
rat575	9468.820000	6773	1.398025
rat783	12019.000000	8806	1.364865
rat99	1671.060000	1211	1.379901
rd100	10463.500000	7910	1.322819
rd400	20894.000000	15281	1.367319

rl11849	1382900.000000	920847	1.501770
rl1304	376119.000000	252948	1.486942
rl1323	390557.000000	270199	1.445442
rl1889	452653.000000	316536	1.430021
rl5915	850261.000000	565040	1.504780
rl5934	853172.000000	554070	1.539827
st70	861.765000	675	1.276689
ts225	188010.000000	126643	1.484567
tsp225	5352.670000	3919	1.365825
u1060	302532.000000	224094	1.350023
u1432	214203.000000	152970	1.400294
u159	54081.000000	42080	1.285195
u1817	82316.300000	57201	1.439071
u2152	93425.700000	64253	1.454029
u2319	320148.000000	234256	1.366659
u574	50331.900000	36905	1.363823
u724	59358.400000	41910	1.416330
usa13509	NA	19947008	NA
vm1084	348762.000000	239297	1.457444
vm1748	483168.000000	336556	1.435624

Table 2: Twice Around The Tree

No nosso primeiro algoritmo aproximativo notamos resultados muito bons, que seguiram se mantiveram dentro do que o algoritmo se propõe. Embora ele tenha apresentado resultados piores na maioria dos casos o algoritmo cumpre seu papel de gerar soluções aproximadas em tempo polinomial.

Dataset	Estimativa	Limiar	Aproximação
a280	3599.370000	2579	1.395646
berlin52	10373.800000	7542	1.375471
bier127	157396.000000	118282	1.330684
brd14051	NA	468942	NA
ch130	8721.920000	6110	1.427483
ch150	9402.820000	6528	1.440383
d1291	77070.500000	50801	1.517106
d15112	NA	1564590	NA
d1655	86207.100000	62128	1.387572
d18512	NA	644650	NA
d198	19830.700000	15780	1.256698
d2103	124426.000000	79952	1.556259
d493	47758.800000	35002	1.364459
d657	69124.000000	48912	1.413232
eil101	889.113000	629	1.413534

eil51	614.242000	426	1.441883
eil76	757.287000	538	1.407597
fl1400	29145.500000	20127	1.448080
fl1577	30667.200000	22204	1.381157
fl3795	40748.600000	28723	1.418675
fl417	15899.900000	11861	1.340519
fnl4461	259789.000000	182566	1.422987
gil262	3513.040000	2378	1.477309
kroA100	31059.400000	21282	1.459421
kroA150	39299.200000	26524	1.481647
kroA200	41055.100000	29368	1.397954
kroB100	30448.400000	22141	1.375204
kroB150	36467.700000	26130	1.395626
kroB200	42886.100000	29437	1.456877
kroC100	29760.900000	20749	1.434329
kroD100	29125.900000	21294	1.367798
kroE100	32205.400000	22068	1.459371
lin105	21482.100000	14379	1.493991
lin318	64239.400000	42029	1.528454
linhp318	64239.400000	41345	1.553740
nrw1379	80393.000000	56638	1.419418
p654	49599.300000	34643	1.431726
pcb1173	84684.400000	56892	1.488512
pcb3038	201806.000000	137694	1.465612
pcb442	72762.500000	50778	1.432953
pr1002	370052.000000	259045	1.428524
pr107	57924.400000	44303	1.307460
pr124	83510.900000	59030	1.414720
pr136	140267.000000	96772	1.449459
pr144	82268.200000	58537	1.405405
pr152	100318.000000	73682	1.361499
pr226	126364.000000	80369	1.572298
pr2392	554955.000000	378032	1.468011
pr264	66703.600000	49135	1.357558
pr299	66870.600000	48191	1.387616
pr439	152948.000000	107217	1.426528
pr76	157185.000000	108159	1.453277
rat195	3289.710000	2323	1.416147
rat575	9679.410000	6773	1.429117
rat783	12467.700000	8806	1.415819
rat99	1710.880000	1211	1.412783
rd100	11338.400000	7910	1.433426
rd400	21645.800000	15281	1.416517
rl11849	NA	920847	NA
rl1304	391781.000000	252948	1.548860
rl1323	410893.000000	270199	1.520705

rl1889	473690.000000	316536	1.496481
rl5915	891041.000000	565040	1.576952
rl5934	883399.000000	554070	1.594382
st70	891.492000	675	1.320729
ts225	188288.000000	126643	1.486762
tsp225	5332.960000	3919	1.360796
u1060	311174.000000	224094	1.388587
u1432	220709.000000	152970	1.442825
u159	56328.700000	42080	1.338610
u1817	85406.000000	57201	1.493086
u2152	96694.200000	64253	1.504898
u2319	327354.000000	234256	1.397420
u574	52143.200000	36905	1.412903
u724	61100.800000	41910	1.457905
usa13509	NA	19947008	NA
vm1084	360326.000000	239297	1.505769
vm1748	499187.000000	336556	1.483221

Table 3: Algoritmo de Christofides

Por fim, temos o algoritmo de Christofides que podemos ver pelos próprios resultados que houve algum erro durante a implementação desse, uma vez que, temos estimativas piores do que 1,5. Isso pode ter sido causada pela forma com que escolhemos caminhar na árvore uma vez que o algoritmo de matching perfeito mínimo funciona de maneira correta.

4 Conclusão

Como podemos notar o algoritmo que retornou os melhores resultados foi o de Branch And Bound, entretanto ele tem limitações de espaço e tempo. Dos aproximativos o Twice Around The Tree se mostrou muito eficiente por retornar soluções próximas das reais em muito menos tempo e ocupando bem menos espaço. No final, para resolver um problema NP-Difícil deve existir um trade-off entre qualidade de solução e performance, caso deseje a solução exata use o Branch And Bound, caso deseje uma solução próxima da ideial utilize algoritmos aproximativos.