Problema do Caixeiro Viajante com Prioridade

André Figueiredo de Almeida

RA 164047

1. O problema

São dados como entrada:

- Um grafo direcionado $D = (V, A)$;
- Nó $s \in V$, chamado de origem;
- Prioridades p_v para cada $v \in V$ ($p_s = 0$);
- Tempos t_a para todo arco $a \in A$;

O objetivo é encontrar um ciclo hamiltoniano C que começa em s, passa por todos os vértices exatamente uma vez e termina em s, tal que $p(C)$ é mínimo. A função $p(C)$ é definida como: $p(C) = \sum_{v \in V} p_v * t_c(s, v)$, onde $t_C(s,v)$ é a soma dos tempos nas arestas do caminho de s até v no ciclo C.

Todos os resultados apresentados foram executados em uma máquina com processador Intel Core i7-3537U CPU @ 2.00GHz, rodando com GNU/Linux 4.15. O compilador usado foi o GCC 7.3.0 e o Gurobi 7.5.2 como o solver do PLI, usando as *flags* de compilação do Makefile disponibilizado. Para auxiliar na execução dos testes, implementei um *shell script*, que roda todos os testes disponíveis, verifica a qualidade e o tempo das soluções nos quatro modos e exporta no formato CSV[1].

2. Soluções propostas

Para resolver este problema da melhor forma possível, foram estudados quatro métodos, cada um com uma relação distinta entre otimalidade e custo computacional.

Os métodos propostos foram:

- Uma heurística construtiva rápida, mas que sacrifica a otimalidade;
- Uma meta-heurística, no caso deste trabalho, o *simulated annealing*;
- Usar o BRKGA, uma implementação otimizada para algoritmos genéticos;
- Uma formulação de Programação Linear Inteira (o único método exato).

As implementações e resultados de cada método serão discutidos nas próximas seções. Todas as implementações possuem um mecanismo que verifica se o tempo de execução é igual ou superior que o tempo limite estabelecido para execução do programa. Se for, ele para o algoritmo e retorna a melhor solução encontrada até o momento.

3. Heurística construtiva

A heurística construtiva deve retornar um caminho possível de forma muito rápida (mesmo que isto comprometa a otimalidade) e com base em alguma característica do problema em que seja possível inferir qual um provável caminho melhor. Esta heurística também pode ser reutilizada pelas outras implementações como uma solução inicial.

3.1. Implementação

A implementação foi baseada no método *naive* que foi fornecida como um modelo

inicial.

1. Cria um vetor *V* de índice com todos os vértices que ainda não foram incluídos no caminho (no início, isso significa todos os vértices com exceção do inicial).
2. Enquanto *V* não estiver vazio, faça:
3. Pega o último vértice inserido, v_k e remove ele de *V*
4. Para cada vértice ainda disponível em *V*, chamamos de v_d, faça:
5. Calcula a inserção de v_d, onde uma inserção é dada pelo valor *peso*(v_d) * *peso do arco*(v_k,v_d) e atualiza a variável de melhor inserção global, caso a encontrada seja menor que a global
6. Insere no caminho o vértice com inserção mais barata encontrado no passo 4
7. Volta ao passo 2
8. Retorna a solução construída.

3.2. Resultados

Todos os testes rodaram com menos de 1 segundo, então esse dado não foi incluído na tabela.

Tabela 1. Resultados da heurística construtiva

Teste	Resposta	Ótimo	Precisão
teste10_1.txt	4813	2943	61,15%
teste10_2.txt	5033	4036	80,19%
teste11_1.txt	3654	3647	99,81%
teste11_2.txt	9003	4170	46,32%
teste12_1.txt	7036	3091	43,93%
teste12_2.txt	4819	3610	74,91%
teste13_1.txt	10048	5584	55,57%
teste13_2.txt	14054	9081	64,62%
teste14_1.txt	19387	11369	58,64%
teste14_2.txt	20357	3085	15,15%
teste15_1.txt	33851	9957	29,41%
teste15_2.txt	23108	8953	38,74%
teste50_1.txt	1,14E+06	-	-
teste50_2.txt	1,47E+06	-	-
teste100_1.txt	103729	-	-
teste100_2.txt	120606	-	-
teste100_3.txt	116625	-	-
teste100_4.txt	102579	-	-

Na tabela, “resposta” é o resultado retornado pelo código, “ótimo” é o custo esperado (a saída do problema) e precisão é a divisão entre o custo esperado e o custo obtido (esse padrão é mantido em todo o relatório).

Os resultados no geral não ficaram muito perto do resultado esperado, contudo, dado que nossa heurística deve ser rápida e retornar apenas um valor inicial para as outras heurísticas, atingimos nosso objetivo.

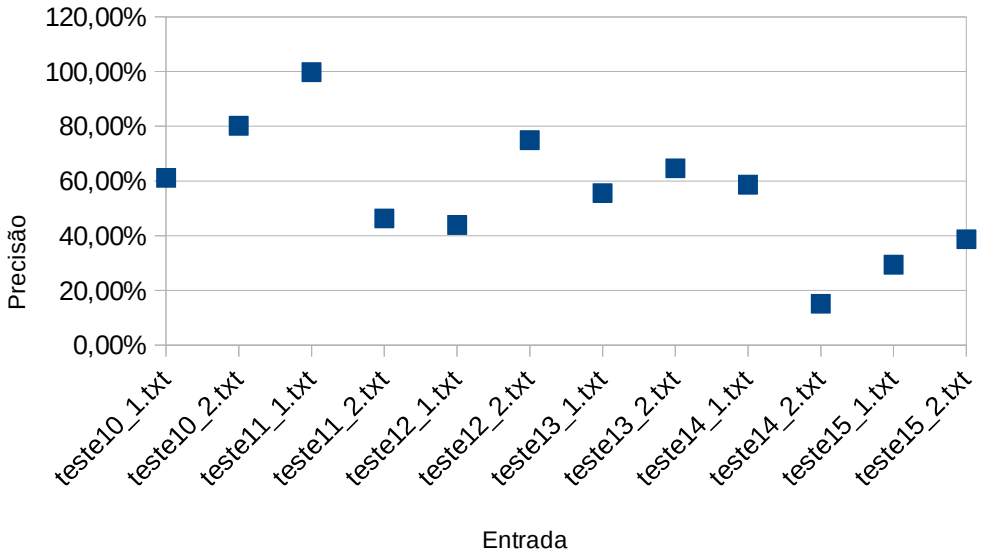

Figura 1. Precisão x Entrada

Apesar da dispersão dos resultados, é possível observar que no geral a qualidade é melhor quando as entradas são menores.

4. Meta-Heurística (*Simulated Annealing)*

O Simulated Annealing[2] é uma meta-heurística que tem como inspiração o processo de recozimento da metalurgia, onde, para se obter um sólido com estrutura cristaligráfica boa, deve-se esfriá-lo com um leve decaimento na temperatura, de forma que não fique em temperatura muito alto por muito tempo nem esfrie abruptamente. Esse algormito une esse controle de temperatura com o algoritmo *Metropolis*[2]. Foi utilizada a solução da heurística construtiva como solução inicial.

4.1. Implementação

O algoritmo foi implementado da seguinte maneira, com k sendo uma constante multiplicativa da temperatura e α é uma constante de decaimento geométrico da temperatura:

1. Cria um caminho inicial $S(0)$, tal que $S(0)$ é o resultado da heurística construtiva para a entrada fornecida.
2. Inicializa S^+, a melhor solução global, com $S(0)$, e a temperatura T com o valor da temperatura inicial;
3. Enquanto $T >$ *temperatura mínima*:
4. Para $i = 0$, enquanto $i <$ *limite do loop interno*, i++:
5. S' = VizinhoAleatório(S)
6. $\Delta S = c(S') - c(S)$
7. se $\Delta S \leq 0$:
8. $S = S'$

9. Se $c(S) < c(S^+)$:
10. $S^+ = S$ // atualiza melhor global
11. Finaliza o loop interno // break
12. senão:
13. r = número aleatório entre (0, 1)
14. se $r < e^{-\Delta S/kT}$:
15. $S = S'$
16. $T = T * \alpha$
17. Retorna S^+

VizinhoAleatório(S) é uma rotina que inverte dois vértice i e j escolhidos aleatoriamente, desde que i não seja igual a j e nenhum dos dois seja o vértice inicial. Uma alternativa de implementação de busca local foi feita com o 2-OPT[2]. 2-OPT(S) é um algoritmo de busca local que encontra a melhor solução vizinha de uma solução inicial S tal que troque apenas duas arestas. Pela característica de verificar todas as trocas de duas arestas, a busca é $O(n^2)$ e retorna um vizinho melhor ou igual.

O fator de troca por uma solução pior na linha 15 é acionado com uma probabilidade tal que, quanto mais longe da solução ótima atual e menor a temperatura, mais improvável será a troca. Ela é realizada de forma aleatória para evitar possíveis mínimos locais, mas sem comprometer a solução final quando o algoritmo está próximo de se encerrar.

O "esfriamento" feito na linha 16 é realizado multiplicando T por uma constante tal que $0 < \alpha < 1$ de forma geométrica, ou seja, começa caindo rapidamente, mas vai caindo mais lentamente conforme se aproxima de zero. Isso evita trocas bruscas na linha 15 conforme o algoritmo se aproxima do final da execução.

4.2. Resultados

Tabela 2. Resultados finais da meta-heurística com as duas buscas locais

	Vizinho Aleatório				2-OPT			
Teste	**Tempo (s)**	**Resposta**	**Ótimo**	**Precisão**	**Tempo (s)**	**Resposta**	**Ótimo**	**Precisão**
teste10_1.txt	0	2943	2943	100,00%	3	4603	2943	63,94%
teste10_2.txt	0	4036	4036	100,00%	3	4042	4036	99,85%
teste11_1.txt	0	3647	3647	100,00%	4	3654	3647	99,81%
teste11_2.txt	0	4179	4170	99,78%	4	7587	4170	54,96%
teste12_1.txt	0	3091	3091	100,00%	6	6586	3091	46,93%
teste12_2.txt	0	3940	3610	91,62%	6	4819	3610	74,91%
teste13_1.txt	0	6915	5584	80,75%	11	9288	5584	60,12%
teste13_2.txt	0	10248	9081	88,61%	9	10608	9081	85,61%
teste14_1.txt	0	11921	11369	95,37%	10	18526	11369	61,37%
teste14_2.txt	0	3191	3085	96,68%	10	20357	3085	15,15%
teste15_1.txt	0	10793	9957	92,25%	13	16958	9957	58,72%
teste15_2.txt	0	10530	8953	85,02%	13	20311	8953	44,08%
teste50_1.txt	1	914174	-	-	299	1,14E+06	-	-
teste50_2.txt	1	919546	-	-	299	1,14E+06	-	-
teste100_1.txt	4	103729	-	-	299	101757	-	-
teste100_2.txt	4	111492	-	-	299	120606	-	-
teste100_3.txt	5	110119	-	-	299	116625	-	-
teste100_4.txt	5	102579	-	-	299	102579	-	-

Os testes tinham como tempo máximo 300 segundos. A implementação usando 2-OPT demorava até duas ordens de grandeza de tempo a mais para executar e alcançava resultados piores. O tempo superior se deve ao fato da busca feita pelo 2-OPT ocorrer em $O(n^2)$ e os resultados inferiores provavelmente se devem a falta de entropia das

respostas pelas quais o algoritmo percorreu.

Para atingir esses resultados, os parâmetros definidos para o *Simulated Annealing* foram:

- Temperatura inicial: 2000
- Temperatura mínima: 10
- Limite do loop interno: 300
- Parâmetro alfa do decaimento exponencial: 0.96
- Parâmetro *k* de multiplicação da temperatura: 2

Outras combinações foram experimentadas, mas essas forma as que obtiveram os resultados mais satisfatórios com a busca de vizinhança aleatória. O tempo de execução se mostrou praticamente linear na implementação com Vizinho Aleatório e polinomial no 2-OPT, com 300 segundos de tempo máximo de execução (as quatro últimas entradas estouraram o limite de tempo e no 2-OPT).

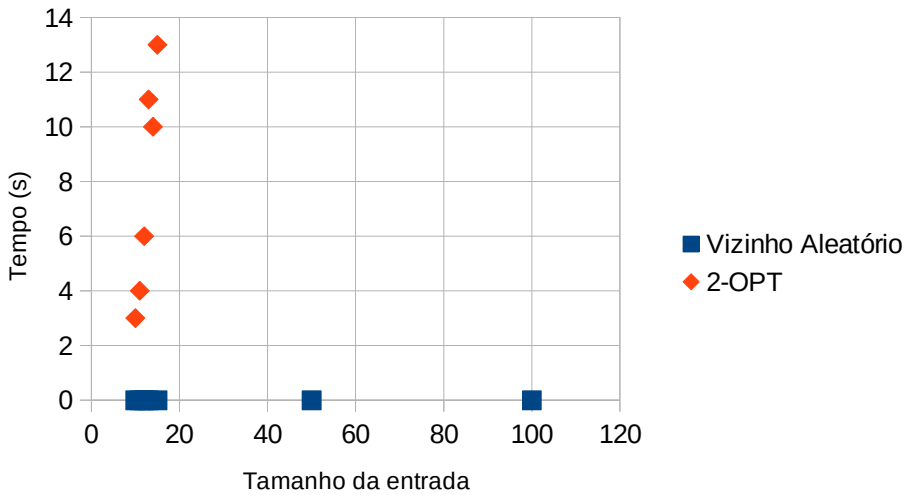

Figura 2. Tempo (s) x Tamanho da entrada

Quanto a qualidade das soluções pequenas, usando 2-OPT não houve um padrão bem definido, com as respostas em média obtendo 63,79% de precisão. Já no Vizinho Aleatório, a precisão média foi de 94,17%. As soluções nos casos grandes foram parecidos porque ambas implementações não conseguiram “caminhar para muito longe” do valor e do caminho inicial retornado pela heurística construtiva. Podemos observar que a heurística construtiva cria um limite inferior de qualidade (e superior de valor) em relação as respostas da meta-heurística, e ele o 2-OPT fica próximo a esse limite para a maioria das entradas.

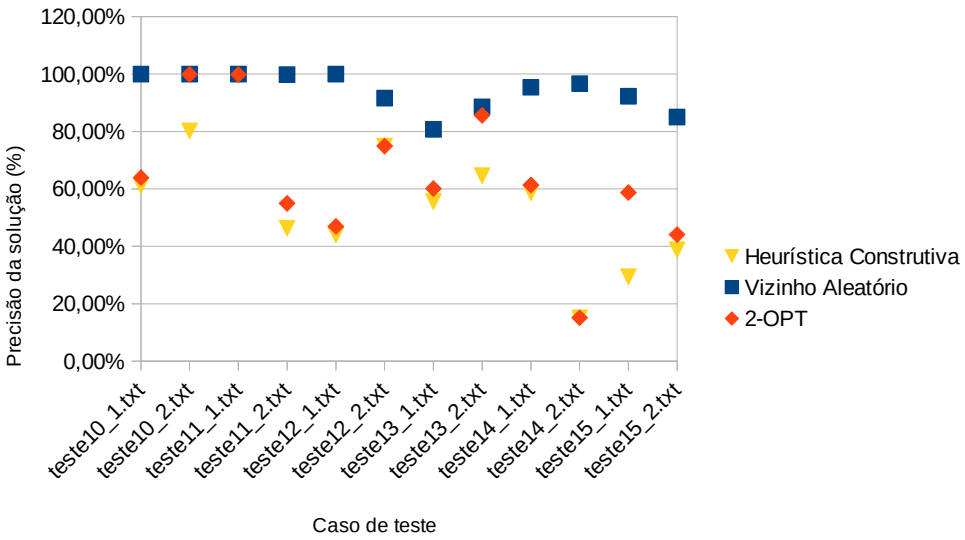

Figura 3. Precisão da solução (s) x Entrada

Em ambos os quesitos (qualidade da solução e tempo de execução) a implementação usando a vizinhança aleatória foi melhor em todos os casos testes. Isso pode ser um reflexo da alta variabilidade das soluções que o *Simulated Annealing* alcança em pouco tempo, facilitando encontrar soluções mais próximas das ideais.

5. BRKGA

O BRKGA (*Biased Random-Key Genetic Algorithm* ou algoritmo genético de chave randômica enviesada) é uma meta-heurística baseada em algoritmos genéticos. Um algoritmo genético é uma técnica para encontrar uma solução ideal ou próxima à ideal para um problema computacional, com inspirações nas teorias darwinistas de evolução dos seres vivos. As iterações do algoritmo são sobre as gerações, onde uma geração é um conjunto de indivíduos e indivíduos são decodificados como entradas. No nosso problema, um indivíduo possuiu uma sequência de cromossomos da mesma cardinalidade do conjunto de vértices menos um (o vértice inicial é fixo e não muda de posição) e cada cromossomo possui um vértice correspondente. Então, os cromossomos são ordenados crescentemente pelo seu valor-chave e tem-se um caminho válido gerado pelo algoritmo. A iteração do algoritmo genético do BRKGA segue os seguintes passos:

1. Os parâmetros dos indivíduos da primeira geração são gerados randomicamente;
2. Os indivíduos da população são decodificados como descrito anteriormente;
3. Verifica se o critério de parada foi satisfeito (se foram geradas todas as n gerações, com n sendo uma constante pré-definida);
4. Se sim, retorna o cromossomo do indivíduo mais bem-sucedido (caminho com menor custo encontrado);
5. Se não, calcula o custo do caminho dos indivíduos gerados e ordena de forma decrescente;
6. Alguma porcentagem dos melhores indivíduos (segundo seus resultados do item anterior) é escolhida e são definidos como “elite”;
7. Replica os indivíduos elite para a próxima geração. A partir deles, combinações

e mutações gerarão os outros indivíduos da nova geração.

8. Causa mutações (distúrbios nos cromossomos com cardinalidade e probabilidades aleatórias) na próxima população;
9. Combina indivíduos elite e não-elite e adiciona seus filhos na próxima população;
10. Retorna ao passo 2.

5.1. Implementação

A implementação foi majoritariamente baseada no código livre disponível em [3], feito por Rodrigo Franco Toso e Mauricio G. C. Resende (um dos autores do BRKGA). O algoritmo faz chamadas à API do BRKGA de forma a evoluir uma população inicial. O código foi implementado da seguinte forma (as chamadas à API aparecem com *):

1. São definidas as contantes do problema (tamanho da população, fração da população que será elite, fração a ser substituída por mutantes, probabilidade de herdar genes da elite, número de população independente, número máximo de gerações, número do topo a serem trocados), inicializadas variáveis, o gerador de números aleatórios e uma instância da API do BRKGA.
2. Uma variável *bestChromosome* guarda o melhor cromossomo global encontrado até o momento.
3. De 1 até a constante *Número Máximo de Gerações:*
4. Evolui* a população atual;
5. Se o cromossomo tiver custo menor que o melhor global, atualiza ele;
6. Se a condição de reiniciar geração for verdadeira (*número da geração atual – constante de geração relevante > constante)*, reinicia* a população atual;
7. Substitui* indivíduos da elite por indivíduos não-elite, de acordo com as constantes definidas no passo 1.
8. Decodifica o cromossomo do melhor indivíduo e retorna o caminho.

5.2. Resultados

Tabela 3. Resultados do BRKGA com parâmetros originais (tempo máximo = 300 s)

Teste	Tempo (s)	Resposta	Ótimo	Precisão
teste10_1.txt	3	2943	2943	100,00%
teste10_2.txt	3	4036	4036	100,00%
teste11_1.txt	3	3647	3647	100,00%
teste11_2.txt	3	4170	4170	100,00%
teste12_1.txt	4	3091	3091	100,00%
teste12_2.txt	4	3633	3610	99,37%
teste13_1.txt	4	5584	5584	100,00%
teste13_2.txt	4	9081	9081	100,00%
teste14_1.txt	5	11369	11369	100,00%
teste14_2.txt	5	3085	3085	100,00%
teste15_1.txt	5	10413	9957	95,62%
teste15_2.txt	5	9110	8953	98,28%
teste50_1.txt	32	1.111.570	-	-
teste50_2.txt	34	922950	-	-
teste100_1.txt	106	192843	-	-
teste100_2.txt	108	201700	-	-
teste100_3.txt	105	174930	-	-
teste100_4.txt	104	171098	-	-

Foram feitas duas baterias de testes. Uma primeira com os parâmetros mantidos como encontrados em [3], já que era uma implementação otimizada para o TSP.

Nos casos pequenos, a média da qualidade foi de 99.44%, obtendo um resultado em um tempo razoável. Nos casos grandes, o BRKGA fica mais lento muito rapidamente, mostrando um comportamento polinomial em relação ao tamanho da entrada.

Na segunda bateria de testes os parâmetros foram alterados na tentativa de melhorar o tempo nos casos grandes, que passava de 100 segundos. Os resultados obtidos estão na próxima tabela.

Tabela 4. Resultados do BRKGA com parâmetros novos (tempo máximo = 300 s)

Teste	Tempo (s)	Resposta	Ótimo	Precisão
teste10_1.txt	0	2943	2943	100,00%
teste10_2.txt	0	4036	4036	100,00%
teste11_1.txt	1	3647	3647	100,00%
teste11_2.txt	1	4170	4170	100,00%
teste12_1.txt	1	3091	3091	100,00%
teste12_2.txt	1	3633	3610	99,37%
teste13_1.txt	1	5584	5584	100,00%
teste13_2.txt	1	9749	9081	93,15%
teste14_1.txt	1	11369	11369	100,00%
teste14_2.txt	1	3085	3085	100,00%
teste15_1.txt	1	9957	9957	100,00%
teste15_2.txt	1	9110	8953	98,28%
teste50_1.txt	9	1,37E+06	-	-
teste50_2.txt	9	885625	-	-
teste100_1.txt	29	252451	-	-
teste100_2.txt	29	243780	-	-
teste100_3.txt	28	244705	-	-
teste100_4.txt	29	213990	-	-

A qualidade dos resultados pequenos foi praticamente mantida (99,23% na média) e todos os tempos desceram para mais da metade do original. A razão média entre os resultados anteriores com os novos para os casos grandes é de 88%, ou seja, os resultados foram em média 22% piores, mas com cerca de 1/3 do tempo.

Tabela 5. Comparação entre os parâmetros

Parâmetros	Original	Nova
Tamanho da população	256	256
Fração da população a ser elite	**0,1**	**0,2**
Fração da população a ser substituída por mutantes	**0,1**	**0,2**
Probabilidade dos filhos herdarem alelos da elite	0,7	0,7
Número de populações independentes	3	3
Gerações até a troca dos melhores indivíduos	100	100
Quantos indivíduos serão trocados	2	2
Número máximo de gerações	**1000**	**300**
Número de gerações até reiniciar	200	200

Pelos resultados, os parâmetros novos representam uma opção melhor de modelo.

6. Programação Linear Inteira

Para a modelagem desse problema, foi criada uma variável inteira x_{ij}, indexada em *i,j* na matriz de adjacência dos arcos do grafo. Ou seja, se $x_{ij} = 1$, o arco entre os vértices *i* e *j* foi utilizado e 0 caso contrário. Essa variável é sujeita as seguintes restrições:

- Para limitar que cada vértice seja utilizado exatamente uma vez, a soma das arestas que saem e entrem dele deve ser igual a 2. Definimos essa restrição por $\sum_{j \in V} x_{ij} = 2 \, \forall \, i \in V$.
- A aresta que sai e entra no mesmo vértice não deve ser utilizada, já que o caminho é hamiltoniano. Limitamos isso por $\sum_{i \in V} x_{ii} = 0$.

Para verificar o tempo de cada vértice, criamos uma variável v_i, indexada em *i* pelos índices dos vértices. Ou seja, o valor em v_i guarda o momento em que o vértice *i* foi acessado. Para fazer com que o valor em *v* de um vértice que é visitado mais tarde seja maior que os que são visitados antes, restringimos *v* com essa fórmula: $v_j \geq v_i + t_{ij} - (1 - x_{ij}) * M$, onde t_{ij} é o valor do arco entre *i* e *j* e *M* é um valor suficientemente grande. No nosso caso, definimos *M* como a soma de todos os arcos.

A última restrição a ser aplicada era de que fosse um caminho hamiltoniano. Para isso, definimos que todo componente deve ser conexo, ou seja, que em todo corte deve existir no mínimo duas arestas. Podemos definir isso garantindo que para cada subconjunto *S* não vazio de vértices, devem existir pelo menos $|S|$ - 1 arestas. Escrevemos isso da forma $\sum_{i, j \in S, i \neq j} x_{ij} \leq |S| - 1, \forall \, S \subset V, s \neq \emptyset$.

Por fim, queremos minimizar $p(C)$, tal que $p(C) = \sum_{v \in V} p_v * t_c(s, v)$.

6.1. Implementação

A implementação foi baseada nos conceitos teóricos definidos em [4] e nos conceitos técnicos disponíveis em [5] e [6], incluindo código-fonte. Essas referências são dos próprios autores da biblioteca de solução de PLI usada nessa implementação, o Gurobi, conferindo credibilidade à fonte.

O código primeiramente utiliza-se da heurística construtiva para criar uma solução inicial e realizar um plano de corte, diminuindo o número de soluções viáveis. O código implementa as restrições definidas acima, com exceção da última restrição. Como esta exige um número exponencial de restrições, o PLI se resolveria muito lentamente e ocuparia muita memória. Em vez disto, a cada solução, nós verificamos se nela existe algum *subtour* desconexo, através de um *callback* do Gurobi. Se houve, o Gurobi faz restrições para que este caminho seja inválido. Se não houver, a solução é válida. Como ele retorna os resultados mais minimizado possível, a primeira solução válida será a solução válida de menor custo. A implementação da classe que executa essa função foi obtida em [6].

Na prática, a função de minimização foi implementada de forma que utilizamos um vetor *time*[*i*] com os valores das prioridades de cada vértice de acordo com a sua ordem e multiplicando pelo valor da soma dos pesos dos vértices até o momento.

6.2. Resultados

Como de esperado, o tempo de resolução do PLI cresce muito rápido em relação ao tamanho da entrada. Podemos observar isso na tabela 6, onde teste com 10 vértices roda em menos de 1 segundos, já o com 11 leva aproximadamente 40 segundos rodar. Todas

as entradas pequenas conseguiram obter 100% de precisão (como era esperado, já que se trata de um método exato) em menos de 60 segundos (com exceção da entrada 13_2, que levou 80 segundos). Para as entradas grandes, foi fornecido 600 segundos de tempo limite.

Tabela 6. Resultados do PLI

Teste	Tempo (s)	Reposta
teste10_1.txt	0	2943
teste10_2.txt	1	4036
teste11_1.txt	33	3647
teste11_2.txt	48	4170
teste12_1.txt	15	3091
teste12_2.txt	60	3610
teste13_1.txt	60	5584
teste13_2.txt	80	9081
teste14_1.txt	60	11369
teste14_2.txt	5	3085
teste15_1.txt	60	9957
teste15_2.txt	60	8953
teste50_1.txt	600	436582
teste50_2.txt	600	365290
teste100_1.txt	600	63714
teste100_2.txt	600	53413
teste100_3.txt	600	49971
teste100_4.txt	600	44771

7. Conclusão

Embora a heurística construtiva não venha a retornar resultados úteis na prática para os casos pequenos, ela foi muito útil para melhorar o desempenho dos outros métodos. Nas entradas grandes (com exceção das com 50 vértices) foi um limite superior justo para o *Simulated Annealing* e retornou resultados melhores que o BRKGA.

Com os dois métodos de solução rápida, a meta-heurística e o BRKGA, os resultados para os casos pequenos foram próximos. Enquanto o *Simulated Annealing* retornou uma média de 94,17% com 0 segundos para os casos pequenos, o BRKGA retornou resultados com uma média de 99,23% com tempos entre 0 e 1 para os mesmos casos. Contudo, nos casos grandes, os resultados do *Simulated Annealing* foram em média 50% menores que os do BRKGA e 7 vezes mais rápido na média. Portanto, no caso geral, o *Simulated Annealing* deve ser mais adequado na prática.

A solução exata mostrou resultados muito bons em termo de qualidade de solução, atingindo 100% em todos os casos pequenos e os menores valores para os casos grandes. Contudo, a solução exata apresenta pouca aplicação prática (salvo casos extremos em que é necessário se assegurar, com certeza, da otimalidade da solução), já que o *Simulated Annealing* conseguiu atingir resultados próximos à 95% em menos de 1 segundo e o PLI levou entre uma e duas ordens a mais de tempo para chegar no resultado.

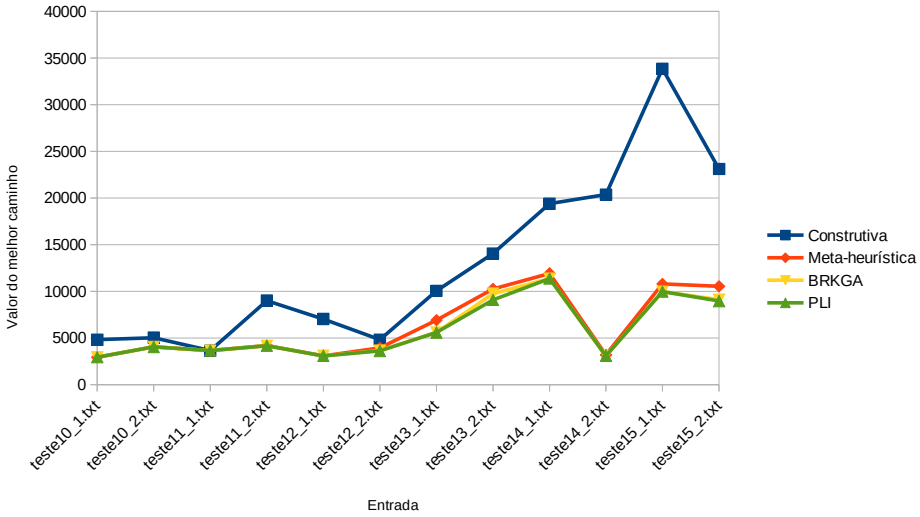

Figura 4. Resultados dos melhores custos encontrados nas entradas pequenas

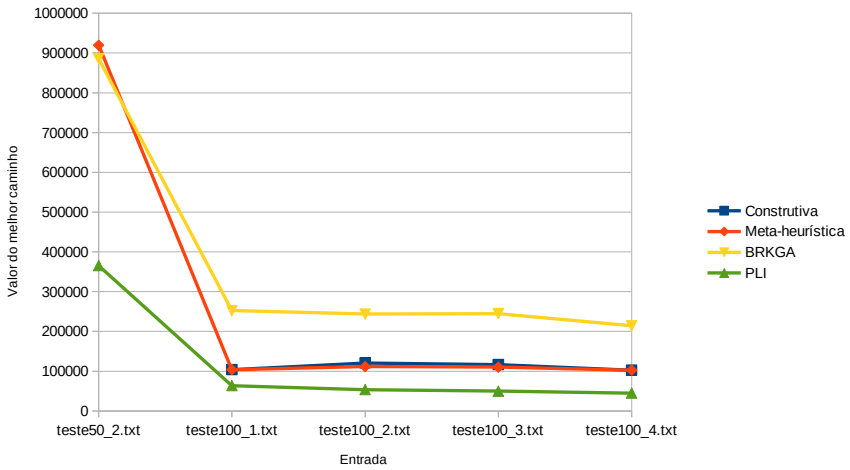

Figura 5. Resultados dos melhores custos encontrados nas entradas grandes

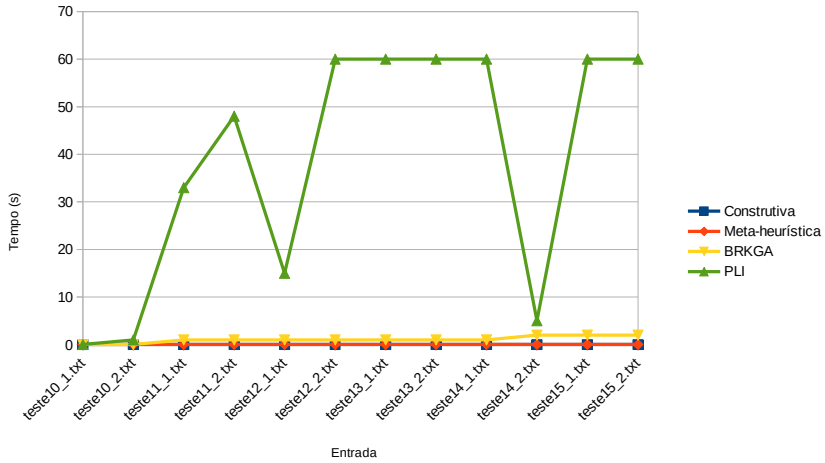

Figura 6. Tempo de execução das entradas pequenas (tempo máximo = 60 s)

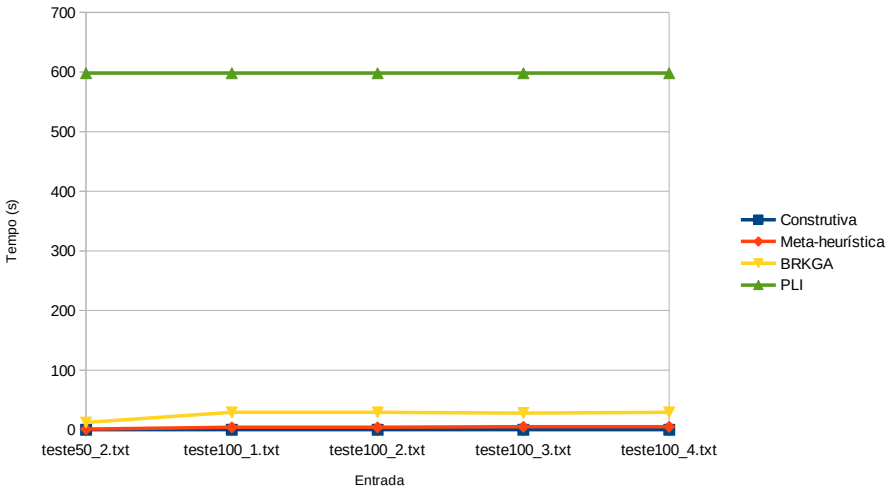

Figura 7. Tempo de execução das entradas pequenas (tempo máximo = 600 s)

Referências

[1] TSCript: A script to run TSP-P tests and gather data: https://github.com/andrealmeid/tscript

[2] Heurísticas e Metaheurísticas 2: http://www.ic.unicamp.br/~fkm/lectures/heuristicas.pdf

[3] BRKGA for the travelling salesman problem: a simple algorithm to the symmetric TSP: https://github.com/rfrancotoso/brkgaAPI/tree/master/examples/brkga-tsp

[4] The Traveling Salesman Problem with integer programming and Gurobi: http://examples.gurobi.com/traveling-salesman-problem/

[5]Some Connectivity Problems solved via Integer Linear Programming: Traveling Salesman Problem, Steiner Tree and Hamiltonian Path. Using LEMON/COIN-OR and the integer programming solver GUROBI: http://www.ic.unicamp.br/~fkm/codes/connectivity.tgz

[6] Gurobi Examples – tsp_c++.cpp: http://www.gurobi.com/documentation/7.5/examples/tsp_cpp_cpp.html