

INF05010 Otimização Combinatória
Trabalho Final:
Algoritmo Memético para o Problema da Ordenação Linear

27 de novembro de 2013

1 Introdução

O objetivo deste trabalho foi implementar uma meta-heurística para resolver o problema de ordenação linear. A meta-heurística escolhida foi um algoritmo memético, cujos detalhes serão expandidos na seção 5.

O problema da ordenação linear é definido como segue: dado um digrafo completo $G = (V, A)$, deseja-se encontrar um torneio $A' \subseteq A$ tal que o grafo (V, A') não contenha ciclos direcionados, e a soma dos pesos de A' seja máxima. Além disto, para cada $i, j \in V$, exatamente um dentre (i, j) e (j, i) pode fazer parte do torneio. Este problema é conhecidamente $\mathcal{NP}$ -Difícil[5].

O problema também pode ser formulado da seguinte maneira: dada uma matriz $A_{n \times n} \in \mathbb{R}^{n \times n}$, encontrar uma permutação de linhas e colunas que maximize a soma dos valores acima da diagonal principal.

2 Formulação do programa inteiro

Variáveis: $x_{ij} \in \{0, 1\} \quad \forall (i, j) \in A$, tal que $x_{ij} = 1$ se a aresta (i, j) está no torneio, e $x_{ij} = 0$ se ela não está.

Função Objetivo:

$$\max \sum_{(i,j) \in A} x_{ij} a_{ij}$$

Restrições:

$$x_{ij} + x_{ji} = 1 \quad \forall i, j \in V, i \neq j \quad (1)$$

$$x_{ij} + x_{jk} + x_{ki} \leq 2 \quad \forall i, j, k \in V, i \neq j \neq k \quad (2)$$

$$x_{ij} \in \{0, 1\} \quad \forall i, j \in V \quad (3)$$

A restrição (1) cuida para que exista apenas uma aresta ligando cada par de vértices.

A restrição (2) utiliza o fato de que um torneio é acíclico se e somente se ele não contém um ciclo direcionado de tamanho 3 [4]. Ou seja, para cada trio de vértices, pode haver no máximo duas arestas entre as que os conectam em ordem.

É fácil de observar que a restrição genérica (1) expande para $\binom{n}{2}$ restrições, a (2) para $\binom{n}{3}$ restrições, e a (3) para n^2 restrições não triviais. Desta forma, temos $\frac{n^3}{6} + n^2 - \frac{n}{6}$ restrições ao total, e n^2 variáveis.

3 Elementos genéticos

Para o algoritmo genético, é necessário associar alguns elementos do problema a elementos do mundo real.

Cromossomo : Uma permutação $p = (p_1, p_2, \dots, p_n)$ de $1..n$.

Gene : Dado um cromossomo $p = (p_1, p_2, \dots, p_n)$, cada elemento p_i da permutação pode ser visto como um gene.

Genótipo/Fenótipo : Uma matriz $c_{n \times n} = (c_{p_1}, c_{p_2}, \dots, c_{p_n})$ onde $c_i = (a_{i,p_1}, a_{i,p_2}, \dots, a_{i,p_n})$ para algum cromossomo p e uma instância com matriz de entrada a .

População : Um conjunto de permutações de $1..n$.

4 Parâmetros

population_size : tamanho da população, inteiro entre 1 e $n!$.

crossover_rate : taxa de crossover, número real entre 0 e 1.

mutation_rate : taxa de mutação, número real entre 0 e 1.

do_local_search : booleano, *true* para realizar busca local, e *false* para não realizá-la.

max_non_improving_generations : critério de terminação, número inteiro maior do que 1, diz que o algoritmo terminará após serem avaliadas *max_non_improving_generations* gerações que não aumentam o valor da melhor solução.

max_time : tempo limite: o algoritmo executará por, no máximo, *max_time* segundos.

random_seed : semente randômica inicial, utilizada para gerar números aleatórios durante a execução. É garantido que, para uma mesma semente e mesmos parâmetros, a solução seja igual em diferentes execuções.

5 O algoritmo

Algorithm 1 Algoritmo Memético

```
1:  $população \leftarrow \text{GerarPopulaçãoInicial}()$ ;
2:  $melhor\_solução \leftarrow$  melhor elemento na população inicial.
3: while not CriteriosDeParada() do
4:    $nova\_população \leftarrow \emptyset$ 
5:   while  $\text{size}(nova\_população) \neq population\_size$  and not CriteriosDeParada() do
6:     EscolheDoisIndivíduos(), chame-os  $p1$  e  $p2$ .
7:     if  $\text{random}[0, 1) > crossover\_rate$  then
8:        $filho \leftarrow \text{Crossover}(p1, p2)$ 
9:     else
10:       $filho \leftarrow$  melhor entre  $p1$  e  $p2$ .
11:    end if
12:    if  $\text{random}[0, 1) > mutation\_rate$  then
13:       $filho \leftarrow \text{Mutação}(filho)$ 
14:    end if
15:    if  $do\_local\_search$  then
16:       $filho \leftarrow \text{BuscaLocal}(filho)$ 
17:    end if
18:    if  $filho > melhor\_solução$  then
19:       $melhor\_solução \leftarrow filho$ 
20:    end if
21:     $nova\_população \leftarrow nova\_população \cup \{filho\}$ 
22:  end while
23:   $população \leftarrow nova\_população$ 
24: end while
```

5.1 Geração da população inicial

Gera-se $population_size$ permutações randômicas, e aplica-se busca local a cada uma delas.

5.2 Escolha de indivíduos para crossover

A escolha de indivíduos é feita por torneio: seleciona-se k elementos aleatórios da população atual, e seleciona-se os dois melhores dentre eles. Na implementação testada, utilizou-se $k = \text{random}(2, 0.2 \times population_size)$. O valor 0.2 foi escolhido por ter apresentado bons resultados em avaliações empíricas.

5.3 Crossover

Utilizou-se uma variação do Ordered Crossover OX_2 , como descrito em [3]. Dados dois pais, $p1$ e $p2$, são gerados dois filhos, $f1$ e $f2$, que inicialmente são iguais aos respectivos pais. Seleciona-se então k índices aleatórios entre 1 e n , e reordena-se os genes correspondentes a estes índices em $f1$ na ordem que aparecem em $p2$, assim como os de $f2$ na ordem que aparecem em $p1$. Note que esta operação preserva a propriedade que um cromossomo é uma permutação $1..n$.

Por exemplo:

$$p1 = (1, 2, 3, 4, 5, 6)$$

$$p2 = (6, 5, 4, 3, 2, 1)$$

Seleciona-se aleatoriamente os índices 1, 2, e 4. Em $f1$, isso corresponde aos genes 1, 2 e 4, que serão

reordenados para a ordem em que aparecem em $p2$ (4, 2, 1). Em $f2$, isto corresponde aos genes 6, 4 e 3, que serão reordenados para a ordem em que aparecem em $p1$ (3, 4, 6).

Assim temos:

$$\begin{aligned} f1 &= (4, 2, 3, 1, 5, 6) \\ f2 &= (3, 5, 4, 6, 2, 1) \end{aligned}$$

Uma vez selecionados dois filhos, retorna-se o melhor entre eles. Na implementação testada, utilizou-se $k = \text{random}(1, 0.5n)$. O valor 0.5 foi escolhido por ter apresentado bons resultados em avaliações empíricas.

5.4 Mutação

Utilizou-se a técnica k-EM (Exchange Mutation)[1]: escolhe-se $2k$ índices aleatórios entre 1 e n , e executa-se k operações de *swap* entre estes itens. Note que esta operação preserva a propriedade que um cromossomo é uma permutação 1.. n . Na implementação testada, utilizou-se $k = \text{random}(1, 0.25n)$. O valor 0.25 foi escolhido por ter apresentado bons resultados em avaliações empíricas.

5.5 Busca Local

Ao fim de cada iteração, é aplicada uma busca local sobre a nova população gerada, para tentar atingir mínimos locais. Após esta operação, eventuais cromossomos duplicados na população serão removidos.

A operação de vizinhança sobre um cromossomo $p = (p_1, p_2, \dots, p_n)$ é escolher $i, j \in [n]$ e fazer o *swap* de p_i e p_j . Desta maneira, temos que $|N(p)| = \binom{n}{2} = \frac{n(n-1)}{2}$. Note que esta operação preserva a propriedade que um cromossomo é uma permutação 1.. n .

As estratégias para a busca local podem ser *FirstImprovement*, a qual realiza o passo de vizinhança para a primeira solução com valor maior, e *BestImprovement*, que verifica toda a vizinhança e realiza o passo para o vizinho que aumenta mais a função objetivo. A busca local termina quando não houver vizinho que aumenta o valor da função objetivo.

Após avaliação empírica, verificou-se que a técnica *BestImprovement* se mostrou muito mais eficiente, tanto em termos de valor da solução como tempo de processamento, que a técnica *FirstImprovement*.

5.6 Critério de terminação

O algoritmo termina quando, após um número pré-definido de gerações *max_non_improving_generations*, o valor da melhor solução continua o mesmo, ou então quando um tempo limite *max_time* for atingido.

6 Implementação

6.1 Plataforma de implementação

O trabalho foi implementado e testado em sistema operacional Windows 7 Ultimate (64-bit) SP1, com um processador Intel(R) Core(TM) i5-2450M CPU, com 2 núcleos físicos e 2 virtuais de 2.5GHz, com cache L2 de 3072KB e 4GB de memória. A linguagem de programação utilizada foi C++, compilador MSVC12, com flags */W3 /O2*.

6.2 Estruturas de dados utilizadas

Para representar a matriz de entrada em memória, utilizou-se um array singular com dimensão $n \times n$. Este tipo de representação favorece a representação da matriz na memória cache, já que todas as posições estão contíguas – o que não ocorre caso fosse utilizado um array de arrays.

Para representar uma possível permutação (cromossomo), simplesmente utilizou-se um array de tamanho n . Cada cromossomo também guarda o valor de sua solução, para evitar recálculos desnecessários.

6.3 Cálculo do fitness de um cromossomo

Uma operação muito frequente no algoritmo é executar o *swap* de duas posições i e j na permutação: ela é usada na busca local (operação dominante no custo temporal) e na mutação.

No entanto, seria extremamente custoso recalcular toda a função *fitness* para o cromossomo a cada vez que um *swap* ocorre, já é necessário $\frac{n^2}{2} - n = O(n^2)$ operações para calcular o valor do triângulo superior da matriz.

Portanto, a seguinte estratégia foi adotada: subtrai-se as contribuições das colunas e linhas i e j do valor armazenado, executa-se o *swap*, e soma-se as novas contribuições das colunas e linhas i e j , desta vez com os valores trocados. Esta técnica resulta em $4(n - 1) = O(n)$ operações aritméticas.

6.4 Paralelização

Observando-se o pseudocódigo do algoritmo, pode-se concluir sem muita dificuldade que o algoritmo é paralelizável: o segmento de código que vai da linha 6 à linha 20, e que executa as operações mais computacionalmente custosas, pode ser executado paralelamente, já que acessa elementos da população atual sem modificá-los, modificando somente variáveis locais.

No entanto, isto não preservaria a reprodutibilidade dos experimentos, razão pela qual o paralelismo não foi implementado neste trabalho. Isto ocorre pois a ordem em que as threads irão requisitar números aleatórios do gerador randômico é imprevisível.

7 Testes de parâmetros

Para os seguintes testes, variou-se cada parâmetro de entrada separadamente, para tentar determinar qual a configuração mais adequada. Cada um dos parâmetros foi testado com 100 instâncias de tamanho entre 30 e 50, geradas randomicamente. Cada teste foi executado 5 vezes, com sementes aleatórias distintas. Para cada teste, mediu-se o tempo de execução, que foi normalizado para o valor do primeiro tempo registrado, e o valor da solução obtida, que foi normalizado para o melhor valor obtido dentre todos os testes.

7.1 Teste do parâmetro *population_size*

O valor de *population_size* foi variado entre 5 a 100, com incrementos de 5. O teste foi executado sem crossover e sem mutação. A Figura 1 mostra os resultados.

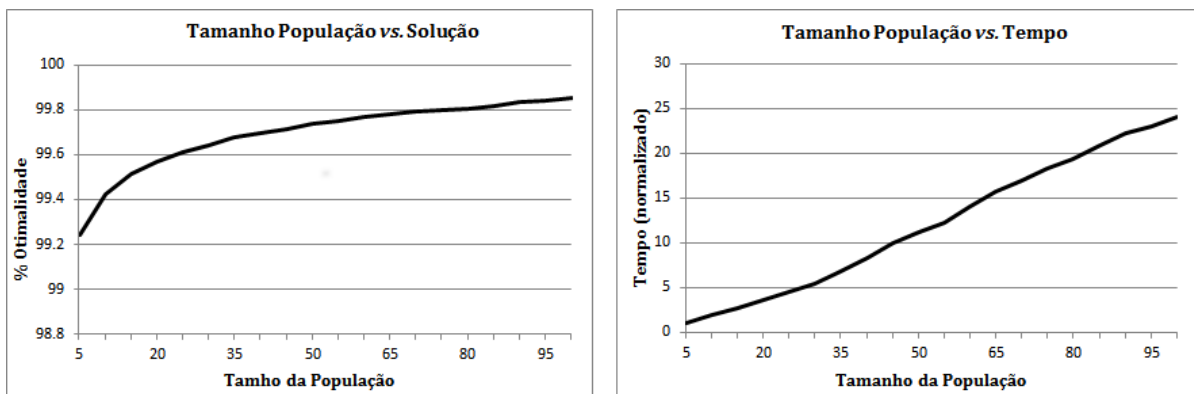


Figura 1: Variação do parâmetro *population_size*

Este é, sem dúvida, o parâmetro mais significativo do algoritmo. Analisando os resultados, observa-se que o percentual de otimalidade cresce de maneira aproximadamente logarítmica conforme cresce o tamanho da população, enquanto o tempo demandado cresce de maneira claramente linear. Conclui-se que é razoável

escolher um tamanho de população entre 30 e 50: maiores valores trarão um benefício pequeno ao comparar-se com o aumento considerável no tempo, enquanto valores menores poderiam ser melhorados com um custo temporal pequeno.

7.2 Teste do parâmetro *crossover_rate*

O valor de *crossover_rate* foi variado entre 0.0 a 1.0, com incrementos de 0.05. O teste foi executado sem mutação, com uma população de tamanho 40, e no máximo 10 gerações não-melhorantes. A Figura 2 mostra os resultados.

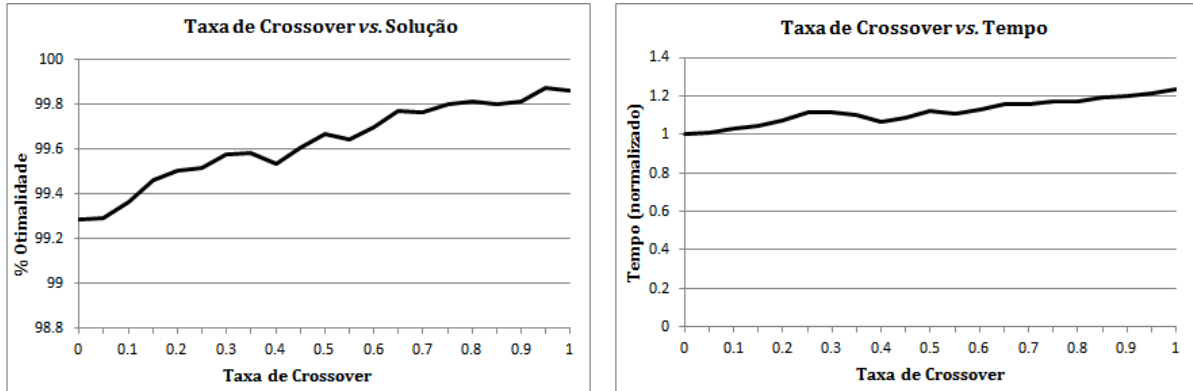


Figura 2: Variação do parâmetro *crossover_rate*

Observa-se que a taxa de otimalidade, apesar de apresentar certas variações, aumenta de maneira aproximadamente linear com o crescimento da taxa de crossover, ao passo que o tempo gasto tem um aumento aparentemente linear, porém extremamente pequeno. Desta maneira, conclui-se que, quanto maior possível o valor da taxa de crossover, melhor, já que este praticamente não influi no tempo gasto.

7.3 Teste do parâmetro *mutation_rate*

O valor de *mutation_rate* foi variado entre 0.0 a 1.0, com incrementos de 0.05. O teste foi executado sem crossover, com uma população de tamanho 40, e no máximo 10 gerações não-melhorantes. A Figura 3 mostra os resultados.

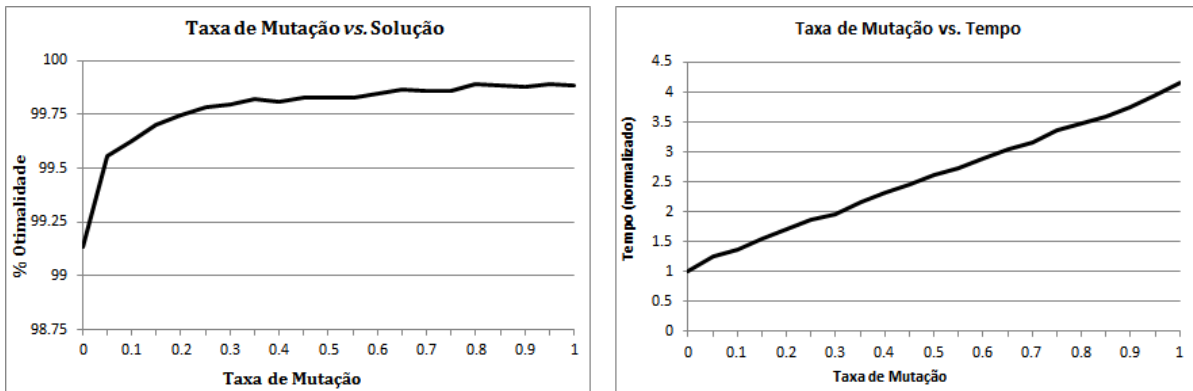


Figura 3: Variação do parâmetro *mutation_rate*

Analisando os resultados, observa-se que após um certo valor para a taxa de mutação, a variação na função objetivo é praticamente imperceptível, enquanto o crescimento do tempo utilizado é claramente linear. Portanto, conclui-se que é razoável escolher um valor para a taxa de mutação de, aproximadamente, 0.3.

7.4 Teste do parâmetro *max_non_improving_generations*

O valor de *max_non_improving_generations* foi variado entre 5 a 100, com incrementos de 5. O teste foi executado com *crossover_rate* = 1.0, *mutation_rate* = 0.25 e com uma população de tamanho 40. A Figura 4 mostra os resultados.

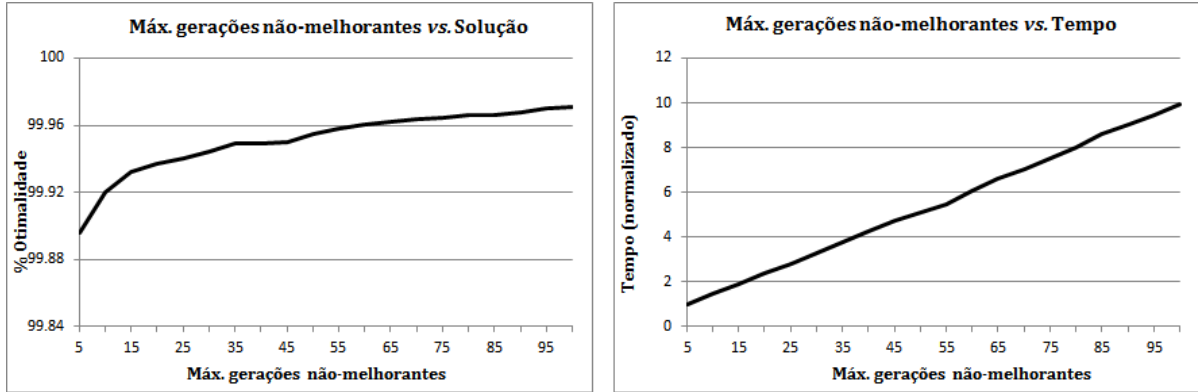


Figura 4: Variação do parâmetro *max_non_improving_generations*

Observa-se que este parâmetro possui uma influência claramente linear no tempo de execução, e uma influência aparentemente logarítmica na qualidade da solução.

Embora em muitas aplicações seja desejável executar o algoritmo por um determinado período de tempo máximo, para valores altos de máximas gerações não-melhorantes, a solução aumenta pouco enquanto o tempo aumenta muito. Portanto, acredita-se que é mais proveitoso utilizar este tempo extra para, em lugar de calcular novas gerações, executar o algoritmo diversas vezes com sementes randômicas diferentes, para que a maior diversidade de populações iniciais facilite a exploração de novos mínimos locais.

8 Testes das instâncias

Neste conjunto de testes, foram testadas as instâncias N-tiw56n54, N-p50-09, N-t1d100.01m, N-atp111, N-t65l11xx, N-t2d150.02, N-t1d150.11, N-atp163, N-t2d200.14, N-t1d500.7, e N-t1d500.25, da LOLIB, disponíveis em <http://www.optsim.es/lolib/>.

A Tabela 1 mostra algumas informações sobre as instâncias. Os valores referência (melhores valores conhecidos) são fornecidos pela LOLIB, enquanto os valores das soluções iniciais correspondem ao valor da solução trivial $p = (1, 2, \dots, n)$, para uma entrada de tamanho n .

Tabela 1: Valores referentes à cada instância.

Instância	n	Valor Referência	Solução Inicial
N-t65l11xx	44	16719	12465
N-p50-09	50	43711	32438
N-tiw56n54	56	91554	24458
N-t1d100.01	100	106852	81262
N-atp111	111	1495	871
N-t1d150.11	150	234157	188705
N-t2d150.02	150	73624	42008
N-atp163	163	2073	1283
N-t2d200.14	200	144384	80918
N-t1d500.25	500	2405718	2108274
N-t1d500.7	500	2400739	2100294

Para cada uma das instâncias, o algoritmo foi executado por exatamente 30 minutos, com três configurações diferentes. Os valores resultantes são comparados com os valores obtidos através de resolução via o solver GLPK, e com as melhores soluções conhecidas segundo a LOLIB.

8.1 Teste sem busca local

Este teste consistiu em executar cada instância por 30 minutos, com parâmetros $population_size = 40$, $crossover_rate = 1.0$, $mutation_rate = 0.25$ e $do_local_search = false$. A ideia é avaliar o quanto a busca local influi no resultado final. A Tabela 2 mostra os resultados.

Tabela 2: Resultados do teste sem busca local

Instância	Valor Referência	Valor Obtido	Desvio para Referência (%)	Desvio para Sol. Inicial (%)	Semente
N-t65l11xx	16719	16719	0.00	-34.13	1569653962
N-p50-09	43711	42186	3.49	-30.05	1874118798
N-tiw56n54	91554	91380	0.19	-273.62	4006437987
N-t1d100.01	106852	102388	4.18	-26.00	93563511
N-atp111	1495	1453	2.81	-66.82	574807814
N-t1d150.11	234157	227561	2.82	-20.59	580489168
N-t2d150.02	73624	73219	0.55	-74.30	3235512124
N-atp163	2073	2007	3.18	-56.43	2142629206
N-t2d200.14	144384	143474	0.63	-77.31	4254013499
N-t1d500.25	2405718	2309824	3.99	-9.56	1531279498
N-t1d500.7	2400739	2300565	4.17	-9.54	3090131583

8.2 Teste com população unitária

Este teste consistiu em executar cada instância por 30 minutos, com apenas um elemento na população, desta maneira degenerando o algoritmo para uma simples busca local. Os parâmetros utilizados foram $population_size = 1$, $crossover_rate = 0.0$, $mutation_rate = 0.0$ e $do_local_search = true$. A ideia é comparar quão melhor é o algoritmo com parâmetros tunados do que uma busca local simples. A Tabela 3 mostra os resultados.

Tabela 3: Resultados do teste com população unitária

Instância	Valor Referência	Valor Obtido	Desvio para Referência (%)	Desvio para Sol. Inicial (%)	Semente
N-t65l11xx	16719	16630	0.53	-33.41	1928209974
N-p50-09	43711	41250	5.63	-27.17	985574101
N-tiw56n54	91554	91259	0.32	-273.13	3093949009
N-t1d100.01	106852	102785	3.81	-26.49	3869399849
N-atp111	1495	1428	4.48	-63.95	2790418292
N-t1d150.11	234157	226609	3.22	-20.09	4197480042
N-t2d150.02	73624	73262	0.49	-74.40	730522204
N-atp163	2073	1978	4.58	-54.17	1516259335
N-t2d200.14	144384	143660	0.50	-77.54	2678964741
N-t1d500.25	2405718	2357212	2.02	-11.81	3862427152
N-t1d500.7	2400739	2353313	1.98	-12.05	894601866

8.3 Teste com parâmetros tunados

Este teste consistiu em executar cada instância por 30 minutos, com os melhores valores obtidos no teste de cada parâmetro separadamente. Os parâmetros utilizados foram $population_size = 40$, $crossover_rate = 1.0$,

$mutation_rate = 0.25$ e $do_local_search = true$. A ideia é ver como o algoritmo se comporta com a (teoricamente) melhor configuração. A Tabela 4 mostra os resultados.

Tabela 4: Resultados do teste com parâmetros tunados

Instância	Valor Referência	Valor Obtido	Desvio para Referência (%)	Desvio para Sol. Inicial (%)	Semente
N-t65l11xx	16719	16719	0.00	-34.13	3680230590
N-p50-09	43711	43634	0.18	-34.52	1974150471
N-tiw56n54	91554	91553	0.00	-274.33	1506506657
N-t1d100.01	106852	106369	0.45	-30.90	137668471
N-atp111	1495	1488	0.47	-70.84	775729445
N-t1d150.11	234157	233130	0.44	-23.54	894377495
N-t2d150.02	73624	73586	0.05	-75.17	1661648426
N-atp163	2073	2056	0.82	-60.25	2398225556
N-t2d200.14	144384	144246	0.10	-78.26	3525590560
N-t1d500.25	2405718	2387373	0.76	-13.24	3484803453
N-t1d500.7	2400739	2382813	0.75	-13.45	2061142236

8.4 Execução com o solver GLPK

Os seguintes testes consistiram em executar o programa inteiro formulado no solver GLPK, com um tempo limite de 30 minutos. A versão utilizada do solver foi 4.52, para Windows 64 bits. A ideia é comparar os resultados obtidos pela meta-heurística com os obtidos via solver. A Tabela 5 mostra os resultados. As linhas com solução zero significam que o solver não conseguiu encontrar nenhuma solução inteira factível no tempo alocado.

Tabela 5: Resultados do teste com o solver

Instância	Valor Referência	Valor Obtido	Desvio para Referência (%)	Desvio para Sol. Inicial (%)
N-t65l11xx	16719	16719	0.00	-34.13
N-p50-09	43711	42612	2.51	-31.36
N-tiw56n54	91554	91554	0.00	-274.33
N-t1d100.01	106852	0	100.00	100.00
N-atp111	1495	0	100.00	100.00
N-t1d150.11	234157	0	100.00	100.00
N-t2d150.02	73624	0	100.00	100.00
N-atp163	2073	0	100.00	100.00
N-t2d200.14	144384	0	100.00	100.00
N-t1d500.7	2400739	0	100.00	100.00
N-t1d500.25	2405718	0	100.00	100.00

Observa-se que, para a maioria das instâncias, é impraticável utilizar a solução via solver, já que o número de restrições criadas é muito grande.

8.5 Comparação dos testes das instâncias

As Figuras 8.5, 8.5, 8.5 e 8.5 mostram os resultados dos testes para cada instâncias, comparando as diferentes configurações. Observa-se que a configuração com os parâmetros devidamente tunados tem uma performance muito superior às outras. Para algumas instâncias, a execução com tamanho de população 1 é melhor do que a sem busca local, porém, a sem busca local fica em segundo lugar na maioria dos casos.

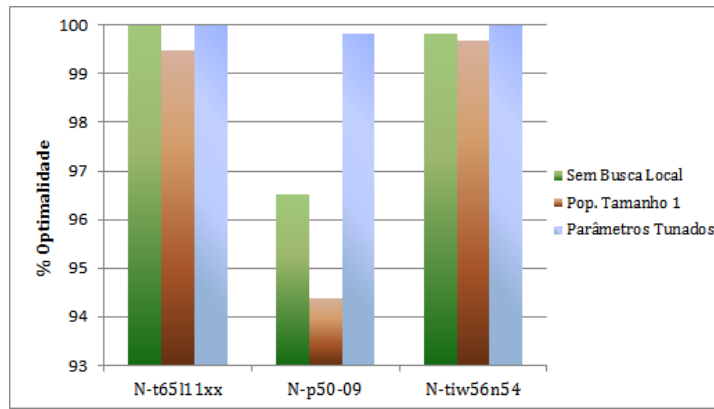


Figura 5: Comparação das configurações para N-t65l11xx, N-p50-09 e N-tiw56n54

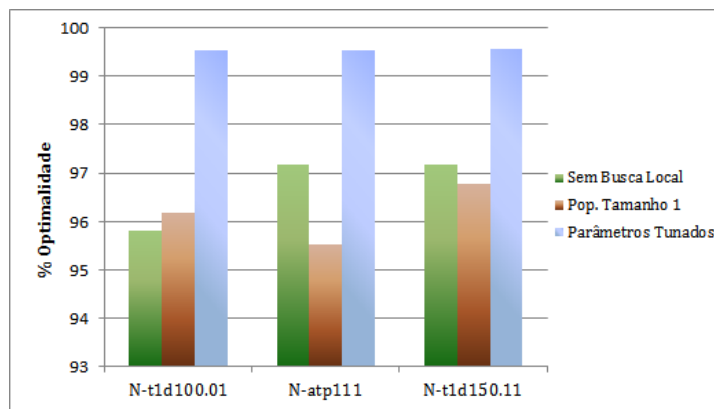


Figura 6: Comparação das configurações para N-t1d100.01, N-atp111 e N-t1d150.11

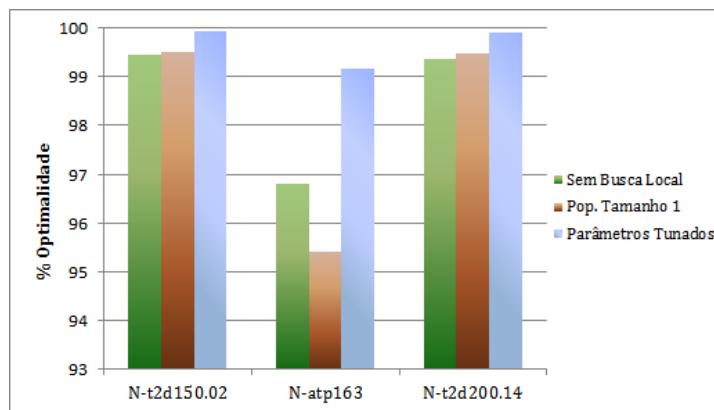


Figura 7: Comparação das configurações para N-t2d150.02, N-atp163 e N-t2d200.14

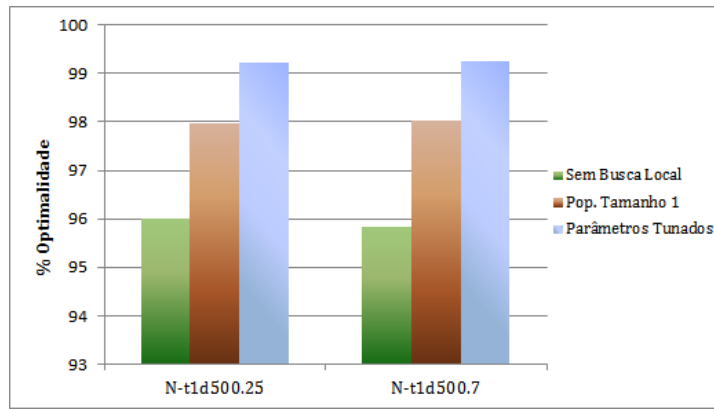


Figura 8: Comparação das configurações para N-t1d100.7 e N-t1d500.25

9 Conclusão

Considera-se que o algoritmo teve uma performance relativamente boa, chegando a um fator de otimalidade acima de 99% para todas as instâncias testadas. Conclui-se que, para entradas grandes, é muito mais vantajoso buscar uma solução aproximada via meta-heurística do que uma solução exata via solver.

Através dos testes para cada parâmetro separadamente, propõe-se uma melhor configuração para o algoritmo: *do_local_search* = true, *population_size* = 40, *crossover_rate* = 1.0, e *mutation_rate* = 0.25.

A codificação do algoritmo foi relativamente simples, e foi possível realizar diversas otimizações a nível de código. O fato de o algoritmo poder ser paralelizado também consiste em uma grande vantagem.

Se os experimentos fossem executados por maior tempo, possivelmente poderia se conseguir resultados muito melhores, chegando talvez à otimalidade em alguns casos.

Tendo em vista os pontos citados acima, considera-se que o trabalho, como um todo, foi bem-sucedido.

Referências

- [1] Huang, G., Lim, A.: Designing a Hybrid Genetic Algorithm for the Linear Ordering Problem, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2003)*, Lecture Notes in Computer Science 2723, 2003, pp. 1053-1064.
- [2] G. Syswerda: Schedule Optimization Using Genetic Algorithms, in L.Davis (Ed.). *Handbook of Genetic Algorithms*, pp. 332-349, New York, (1991)
- [3] Deep, K., Mebrahtu, H. New Variations of Order Crossover for Travelling Salesman Problem. *IJCOPI* Vol. 2, No. 1, Jan-April 2011, pp. 2-13.
- [4] Martí, R., Reinelt, G.: *The Linear Ordering Problem. Exact and Heuristic Methods in Combinatorial Optimization*. Springer, Heidelberg (2011)
- [5] R.M. Karp: Reducibility among combinatorial problems, R.E. Miller e J.W. Thatcher (Eds.), *Complexity of Computer Computations*, pp. 85-103, New York (1972)