Comparação de desempenho entre algoritmos

Leonardo D. Secchin

Tópicos em PO, 2021/1E

UFES

Julho de 2021

Considere dois algoritmos A e B aplicados aos problemas P1,...,Pn. Após a execução, o seguinte panorama foi observado:

- o Algoritmo A resolveu 65% dos problemas
- o Algoritmo B resolveu 72% dos problemas
- o tempo de execução do Algoritmo B foi, em média, 2 vezes o tempo do Algoritmo A.

Pergunta: Qual algoritmo foi melhor? Qual você indicaria?

Considere dois algoritmos A e B aplicados aos problemas P1,...,Pn. Após a execução, o seguinte panorama foi observado:

- o Algoritmo A resolveu 65% dos problemas
- o Algoritmo B resolveu 72% dos problemas
- o tempo de execução do Algoritmo B foi, em média, 2 vezes o tempo do Algoritmo A.

Pergunta: Qual algoritmo foi melhor? Qual você indicaria?

A resposta possivelmente dependa da situação, do tamanho dos problemas, do condicionamento dos dados, de paralelismo etc etc etc...

Considere dois algoritmos A e B aplicados aos problemas P1,...,Pn. Após a execução, o seguinte panorama foi observado:

- o Algoritmo A resolveu 65% dos problemas
- o Algoritmo B resolveu 72% dos problemas
- o tempo de execução do Algoritmo B foi, em média, 2 vezes o tempo do Algoritmo A.

Pergunta: Qual algoritmo foi melhor? Qual você indicaria?

A resposta possivelmente dependa da situação, do tamanho dos problemas, do condicionamento dos dados, de paralelismo etc etc etc...

ok, mas há uma forma de responder "balanceando" tempo *vs* eficácia?

Agora, tente dar uma "resposta média" olhando a tabela... !!!!! :(

Row	Prob	n	m	st	it	f	viab	KKT	tempo	metodo
1	lp_adlittle	138	56	1	8	NaN	4.59223e12	5.6862e8	0.00625257	afimescala
2	lp_adlittle	138	56	0	13	225495.0	1.38656e-10	2.17674e-10	0.00219731	pred-corr_p2
3	lp_adlittle	138	56	0	13	225495.0	1.38062e-9	4.6669e-12	0.00212062	pred-corr_p3
4	lp_adlittle	138	56	1	20	NaN	1.24738e-10	2.84205e-8	0.00350535	seguidor
5	lp_afiro	51	27	4	18	-464.754	10.071	0.00554879	0.00123545	afimescala
6	lp_afiro	51	27	0	8	-464.753	6.56722e-12	7.32249e-10	0.000450609	pred-corr_p2
7	lp_afiro	51	27	0	8	-464.753	9.48886e-12	1.60566e-10	0.000536535	pred-corr_p3
8	lp_afiro	51	27	0	16	-464.753	5.10003e-12	2.49791e-9	0.000865001	seguidor
9	lp_agg	615	488	1	33	NaN	3.52612e15	6.39858e7	0.0930459	afimescala
10	lp_agg	615	488	0	27	-3.59918e7	2.7054e-5	1.92828e-9	0.0961403	pred-corr_p2
11	lp_agg	615	488	0	39	-3.59918e7	1.0664e-5	6.31478e-13	0.142016	pred-corr_p3
12	lp_agg	615	488	0	57	-3.59918e7	0.0658189	1.56728e-9	0.143155	seguidor
13	lp_agg2	758	516	4	61	-1.99724e7	137955.0	0.00798686	0.203749	afimescala
14	lp_agg2	758	516	0	20	-2.02393e7	2.11389e-7	7.04435e-12	0.0931601	pred-corr_p2
15	lp_agg2	758	516	0	19	-2.02393e7	2.02984e-5	1.72002e-10	0.0913567	pred-corr_p3
16	lp_agg2	758	516	0	30	-2.02393e7	0.00172784	6.43798e-9	0.0974315	seguidor
17	lp_agg3	758	516	1	50	NaN	5.01421e15	2.92043e8	0.163052	afimescala
18	lp_agg3	758	516	0	20	1.03121e7	3.48558e-7	1.30753e-9	0.0950648	pred-corr_p2
19	lp_agg3	758	516	0	18	1.03121e7	9.58804e-6	9.13078e-11	0.0866596	pred-corr_p3
20	lp_agg3	758	516	0	32	1.03121e7	0.000617788	1.45328e-9	0.109657	seguidor
21	lp_bandm	472	305	4	16	-139.181	25.8847	0.370008	0.0234934	afimescala
22	lp_bandm	472	305	0	19	-158.628	6.21731e-10	5.5994e-9	0.0289009	pred-corr_p2
23	lp_bandm	472	305	0	19	-158.628	3.3434e-10	1.92558e-12	0.0266341	pred-corr_p3
24	lp_bandm	472	305	0	31	-158.628	3.01558e-10	6.49026e-9	0.0348959	seguidor
25	lp_beaconfd	295	173	4	16	33574.4	3230.33	0.219422	0.0131723	afimescala
26	lp_beaconfd	295	173	0	11	33592.5	1.33872e-5	6.59554e-9	0.0102533	pred-corr_p2
27	lp_beaconfd	295	173	0	10	33592.5	1.61167e-8	6.77704e-12	0.00898379	pred-corr_p3
28	lp_beaconfd	295	173	0	18	33592.5	2.93402e-9	4.69255e-9	0.0125896	seguidor
29	lp_blend	114	74	4	20	-29.8934	7.46157	0.0660842	0.00431388	afimescala
30	lp_blend	114	74	0	14	-30.8121	1.47306e-11	9.00176e-10	0.00327916	pred-corr_p2
31	lp_blend	114	74	0	13	-30.8121	3.29287e-11	1.70695e-11	0.00308425	pred-corr_p3
32	lp_blend	114	74	0	20	-30.8121	2.25261e-11	9.91278e-9	0.00369068	seguidor
33	lp_bnl2	4486	2324	1	45	NaN	1.10967e15	3.02111e10	1.10452	afimescala
34	lp_bnl2	4486	2324	4	40	1811.24	0.00125769	3.42411e-8	1.74602	pred-corr_p2
35	lp_bnl2	4486	2324	4	36	1811.24	0.000585816	1.59491e-8	1.58145	pred-corr_p3
36	lp_bnl2	4486	2324	0	68	1811.24	5.68037e-7	6.48521e-10	1.65283	seguidor
37	lp_d2q06c	5831	2171	4	13	2.52971e5	2.17288e5	1.49644	0.412703	afimescala
38	lp_d2q06c	5831	2171	4	26	1.22829e5	8.90209e-5	0.000328237	1.29015	pred-corr_p2
39	lp_d2q06c	5831	2171	0	30	1.22784e5	3.41739e-5	2.07375e-9	1.47965	pred-corr_p3
40	lp_d2q06c	5831	2171	0	62	1.22784e5	1.17896e-5	1.63516e-9	1.80164	seguidor

Algoritmos podem ser comparados por diversos ângulos, a depender da situação. Algumas medidas comuns:

Algoritmos podem ser comparados por diversos ângulos, a depender da situação. Algumas medidas comuns:

- **Tempo de execução:** útil para comparar algoritmos de diferentes naturezas. *Veremos mais a frente alguns cuidados ao contabilizar tempo de CPU.*
- **Número de iterações:** útil quando os algoritmos têm iterações com custo computacional parecido. Exemplos:

Algoritmos podem ser comparados por diversos ângulos, a depender da situação. Algumas medidas comuns:

- **Tempo de execução:** útil para comparar algoritmos de diferentes naturezas. *Veremos mais a frente alguns cuidados ao contabilizar tempo de CPU.*
- **Número de iterações:** útil quando os algoritmos têm iterações com custo computacional parecido. Exemplos:
  - Pontos interiores primal dual afim escala, seguidor de caminhos e preditor-corretor – custo por iteração parecido

Algoritmos podem ser comparados por diversos ângulos, a depender da situação. Algumas medidas comuns:

- **Tempo de execução:** útil para comparar algoritmos de diferentes naturezas. *Veremos mais a frente alguns cuidados ao contabilizar tempo de CPU.*

- **Número de iterações:** útil quando os algoritmos têm iterações com custo computacional parecido. Exemplos:
  - Pontos interiores primal dual afim escala, seguidor de caminhos e preditor-corretor – custo por iteração parecido
  - Método do gradiente e Newton – custo por iteração muito diferente. O primeiro necessita apenas de ∇f; o segundo, resolver um sistema linear com $\nabla^2 f$.
    $\Longrightarrow$ Não faz sentido comparar por número de iterações (gradiente pode dar mais iterações e mesmo assim ser muito mais rápido)

Algoritmos podem ser comparados por diversos ângulos, a depender da situação. Algumas medidas comuns:

- **Tempo de execução:** útil para comparar algoritmos de diferentes naturezas. *Veremos mais a frente alguns cuidados ao contabilizar tempo de CPU.*

- **Número de iterações:** útil quando os algoritmos têm iterações com custo computacional parecido. Exemplos:
  - Pontos interiores primal dual afim escala, seguidor de caminhos e preditor-corretor – custo por iteração parecido
  - Método do gradiente e Newton – custo por iteração muito diferente. O primeiro necessita apenas de ∇f; o segundo, resolver um sistema linear com $\nabla^2 f$.
    $\Longrightarrow$ Não faz sentido comparar por número de iterações (gradiente pode dar mais iterações e mesmo assim ser muito mais rápido)

- **Número de avaliações de funções e/ou gradientes:** útil quando avaliar funções e seus gradientes é caro (problemas de grande porte, funções resultantes de processos iterativos)

OK, independentemente da medida de comparação, queremos uma forma fácil de comparar o desempenho entre dois algoritmos.

OK, independentemente da medida de comparação, queremos uma forma fácil de comparar o desempenho entre dois algoritmos.

Como comparar de forma mais fácil?

Perfis de desempenho: uma forma visual para comparar algoritmos.

OK, independentemente da medida de comparação, queremos uma forma fácil de comparar o desempenho entre dois algoritmos.

Como comparar de forma mais fácil?

Perfis de desempenho: uma forma visual para comparar algoritmos.

Os perfis de desempenho que veremos foram idealizados por Elizabeth D. Dolan e Jorge J. Moré.

Dolan, Elizabeth D.; Moré, Jorge J. Benchmarking optimization software with performance profiles. Math. Program., Ser. A 91: 201-213 (2002)

Eles são muito utilizados pelos pesquisadores em otimização.

Construção dos perfis de desempenho

Vamos fixar **tempo de execução** como medida (as outras serão a mesma coisa).

Construção dos perfis de desempenho

Vamos fixar **tempo de execução** como medida
(as outras serão a mesma coisa).

Sejam

- $\mathcal{P}$ o conjunto dos problemas-teste (instâncias);
- $\mathcal{A}$ o conjunto dos algoritmos aplicados à cada problema $p \in \mathcal{P}$;
- $t_{p,a}$ o tempo de execução do algoritmo $a \in \mathcal{A}$ para resolver $p \in \mathcal{P}$.

  Definimos $t_{p,a} = \infty$ caso a **não** resolveu p.

Construção dos perfis de desempenho

O índice de desempenho do algoritmo a no problema p é a razão

$$r_{p,a} = \frac{t_{p,a}}{\min\{t_{p,\tilde{a}} \mid \tilde{a} \in \mathcal{A}\}} \geq 1$$

Observe que

- $r_{p,a} = 1 \implies$ algoritmo a resolveu p mais rápido;
- quanto maior $r_{p,a}$, mais lento foi a em p (talvez nem resolveu)
- convencionamos $r_{p,a} = \infty$ caso $t_{p,a} = \infty$ (note que o denominador pode ser ∞ caso nenhum algoritmo tenha resolvido p).

Construção dos perfis de desempenho

Definimos a **função de desempenho** (ou **perfil de desempenho**) do algoritmo $a \in \mathcal{A}$ por

$$\rho_a : [1, \infty) \to [0, 1], \qquad \rho_a(\tau) = \frac{1}{\#\mathcal{P}} \operatorname{card} \{p \in \mathcal{P} \mid r_{p,a} \leq \tau\}.$$

- $\rho_a(1)$ é a proporção de problemas que a resolve no menor tempo;
- $\rho_a(\tau)$ é a proporção de problemas que a resolve **em até τ vezes** o tempo do algoritmo mais rápido.

Construção dos perfis de desempenho

Definimos a **função de desempenho** (ou **perfil de desempenho**) do algoritmo $a \in \mathcal{A}$ por

$$\rho_a : [1, \infty) \to [0, 1], \qquad \rho_a(\tau) = \frac{1}{\#\mathcal{P}} \operatorname{card} \{p \in \mathcal{P} \mid r_{p,a} \leq \tau\}.$$

- $\rho_a(1)$ é a proporção de problemas que a resolve no menor tempo;
- $\rho_a(\tau)$ é a proporção de problemas que a resolve **em até τ vezes** o tempo do algoritmo mais rápido.
- a função ρ_a balanceia **tempo** e **eficácia**.
  - qual o algoritmo mais rápido, desconsiderando eficácia? (isto é, sem olhar para a % de problemas resolvidos)
  - qual o algoritmo é mais rápido, considerando uma taxa de resolução de 70% dos problemas?

Construção dos perfis de desempenho

A maneira fácil de visualizar o comparativo entre algoritmos é traçando os gráficos de $\rho_a(\tau)$ em função de τ, para os vários algoritmos.

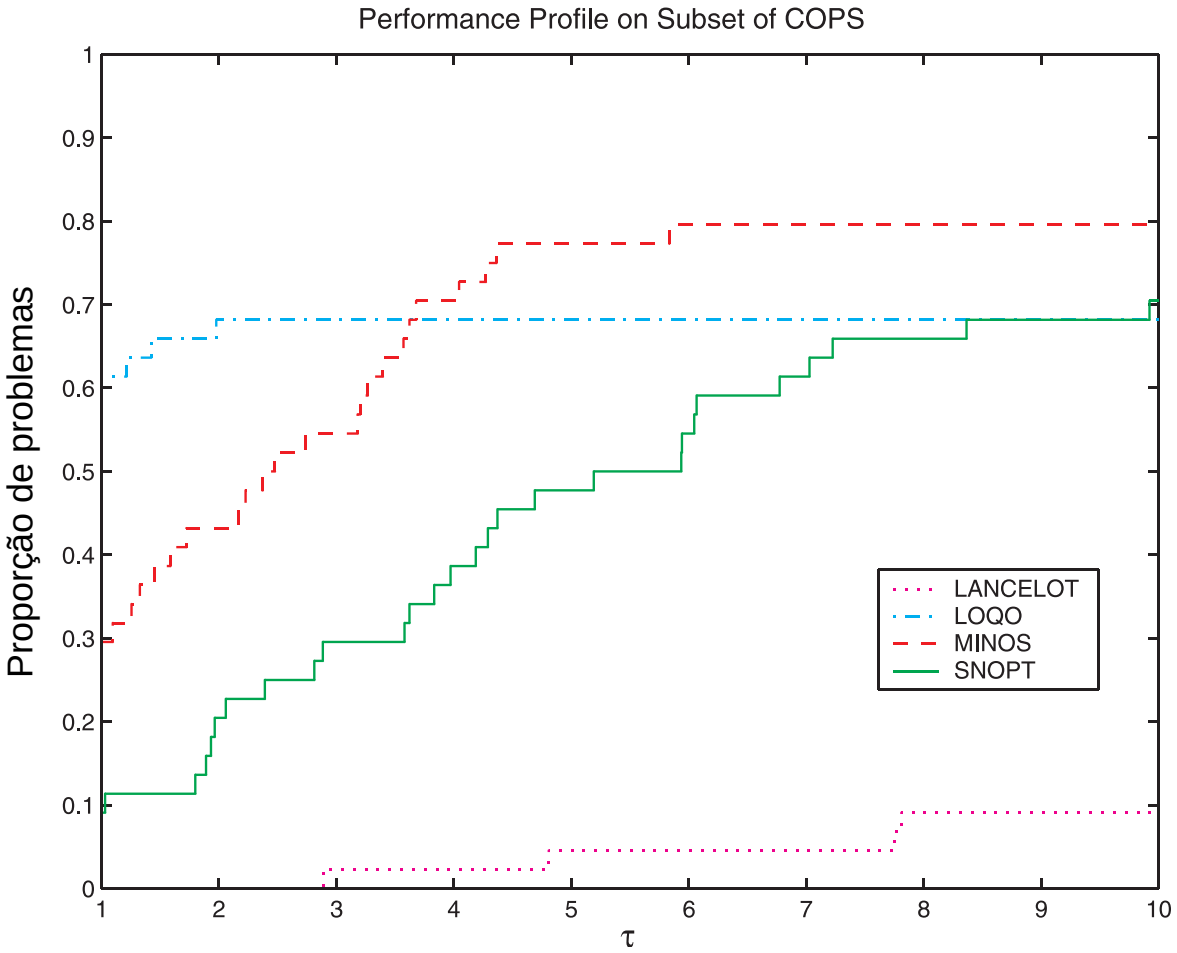
É bastante utilizada pelos pesquisadores para tirar conclusões entre diferentes algoritmos.

Os perfis ρ_a podem ser definimos para qualquer medida de comparação (iterações, avaliações de função etc) de forma inteiramente análoga.

Para tanto, basta definir $t_{p,a}$ como a medida de interesse, mantendo $t_{p,a} = \infty$ quando a não resolve p.

Exemplo 1

Comparativo para até 10 vezes o tempo do algoritmo mais rápido ($\tau \in [1, 10]$)



(figura adaptada do artigo de Dolan e Moré (2002))

LOQO é mais rápido em $\approx 60\%$ dos problemas, mas não é o mais eficaz

MINOS é o mais eficaz se esperamos até 10x o tempo do alg. mais rápido

LOQO e SNOPT têm eficácia similar, porém LOQO é muito mais rápido

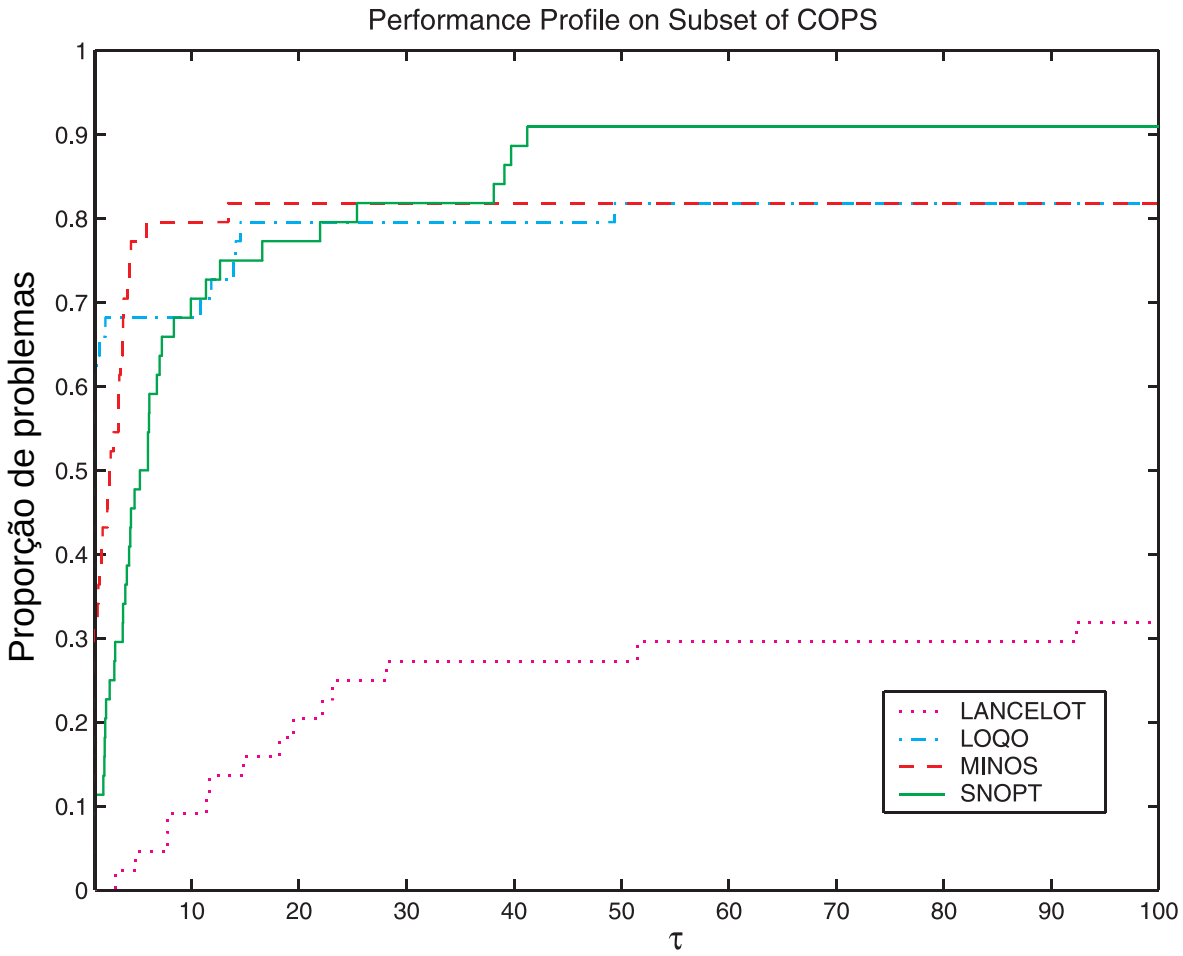
A eficácia entre LOQO e SNOPT se equiparam somente se esperarmos até 8,5x o tempo do alg. mais rápido

MINOS tem eficácia superior se esperamos até cerca de 4x do algoritmo mais rápido

LANCELOT é pior em eficácia e tempo

Exemplo 2

Comparativo para até 100 vezes o tempo do algoritmo mais rápido ($\tau \in [1, 100]$)



(figura adaptada do artigo de Dolan e Moré (2002))

SNOPT é o mais eficaz, porém para resolver $\approx$ 90% dos problemas devemos esperar mais de 40x o tempo do algoritmo mais rápido

LANCELOT aumenta sua eficácia se esperamos mais tempo

LOQO resolve apenas poucos problemas a mais para $\tau \geq 50$

MINOS não é mais eficaz se esperamos mais tempo

Escala logarítmica

As vezes o gráfico fica ilegível próximo de $\tau = 1$. Podemos usar uma escala logarítmica de base 2 (proposto por Dolan e Moré) no eixo τ para melhor visualização.

Na verdade, a proposta é plotar

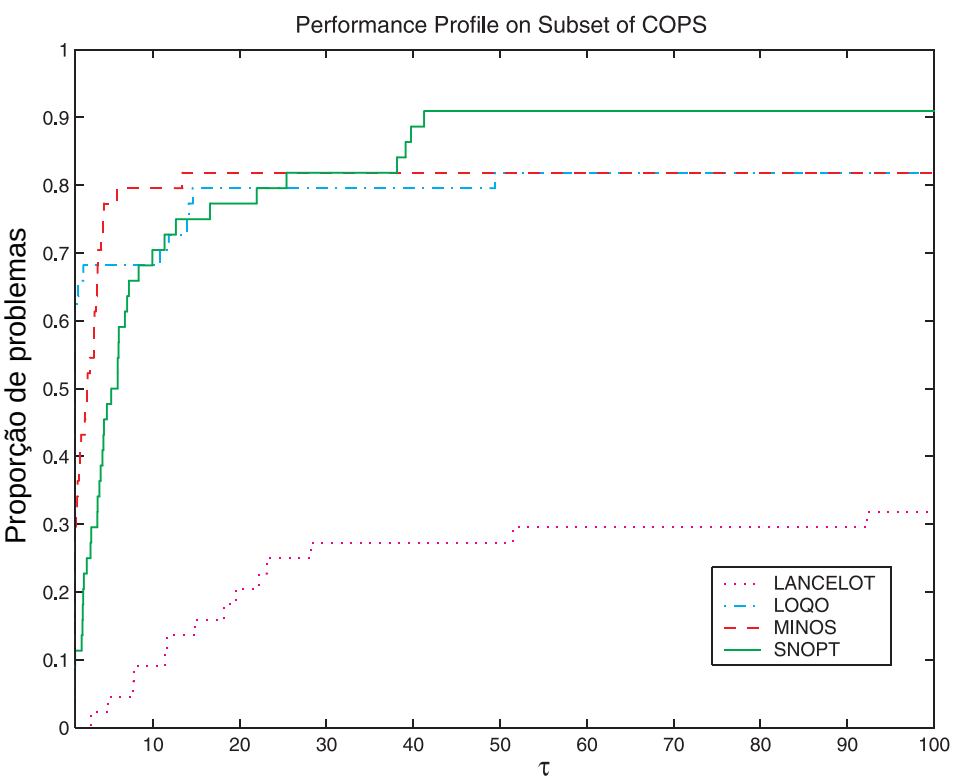
$$\rho_a : [1, \infty) \to [0, 1], \qquad \rho_a(\tau) = \frac{1}{\#\mathcal{P}} \operatorname{card} \{p \in \mathcal{P} \mid \log_2(r_{p,a}) \leq \tau\}.$$

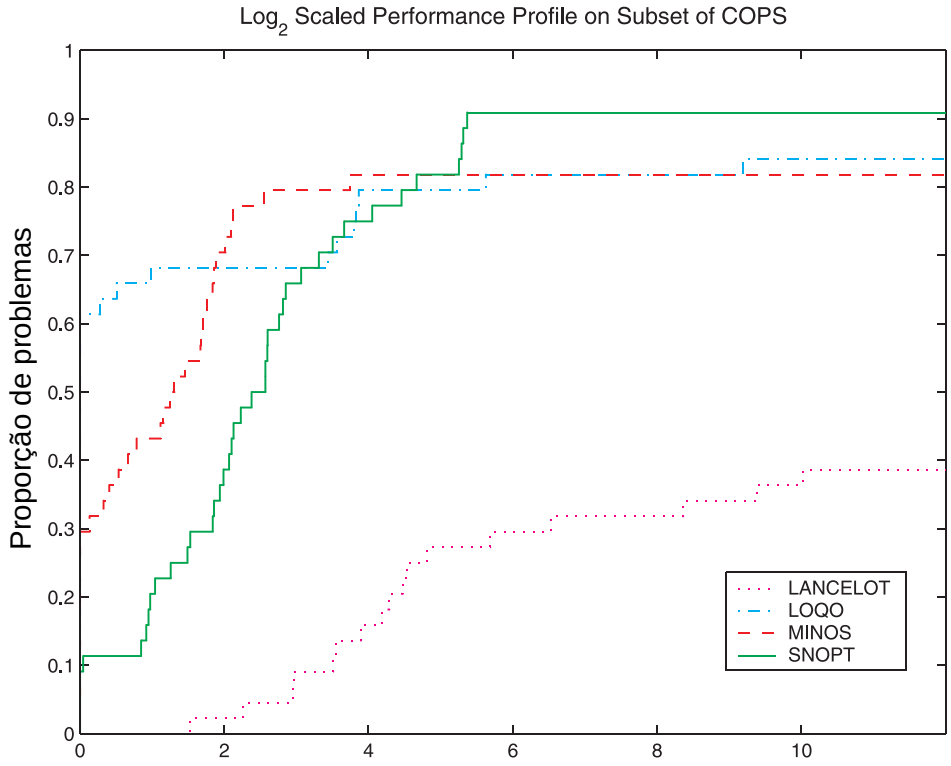
A escala logarítmica serve para dar mais ênfase aos valores de τ menores

$$\text{“estica o eixo” para } \tau \approx 1 \text{ e “comprimi o eixo” para } \tau \gg 1.$$

“Problema”: a leitura fica menos intuitiva...

Exemplo





(figuras adaptadas do artigo de Dolan e Moré (2002))

Eixo horizontal: à esquerda, $\tau \geq 1$; à direita, $\log_2(\tau) \geq 0$.

Obs: $\log_2(40) \approx 5,322$, $\log_2(100) \approx 9,966$

O pacote Julia `BenchmarkProfiles.jl`

O pacote `BenchmarkProfiles.jl` gera facilmente perfis de desempenho a partir da tabela dos dados.

O pacote Julia `BenchmarkProfiles.jl`

O pacote `BenchmarkProfiles.jl` gera facilmente perfis de desempenho a partir da tabela dos dados.

Exercício:

1. Gere uma tabela 25×3 fictícia para 3 algoritmos e 25 problemas:

   ```
   julia> T = 10 * rand(25,3);
   ```

2. Insira alguns tempos infinitos nos dados:

   ```
   julia> T[1:7:end] .= Inf;
   ```

3. Gere um perfil dos dados T com a legenda "Alg 1", "Alg 2", "Alg 3":

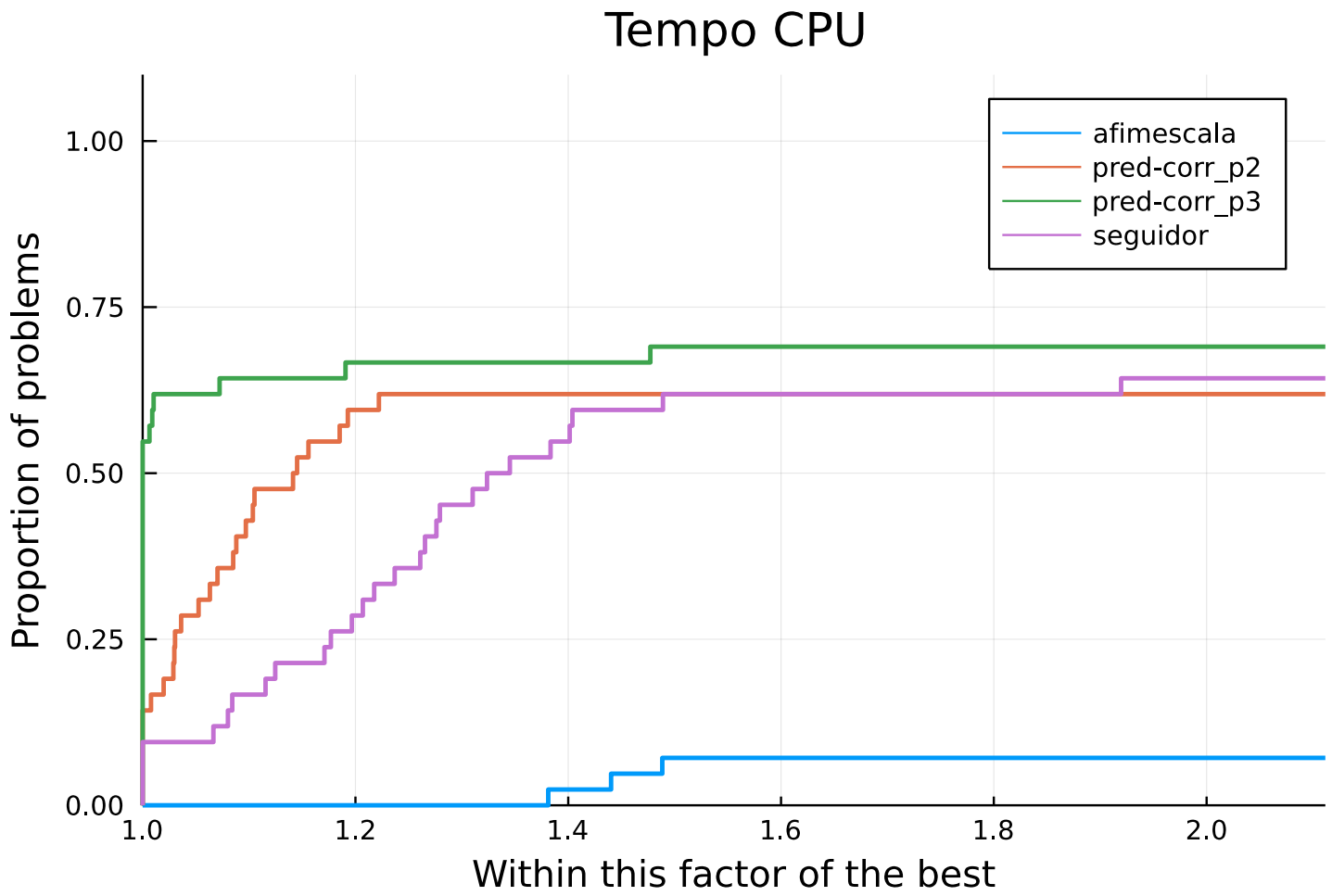
   ```
   julia> fig = performance_profile(ArrayFloat64(T), ["Alg 1","Alg
   2","Alg 3"], title="Tempo CPU", logscale=false)
   ```

4. Salve a figura em PDF (carregue `Plots` antes):

   ```
   julia> savefig(fig, "fig.pdf")
   ```

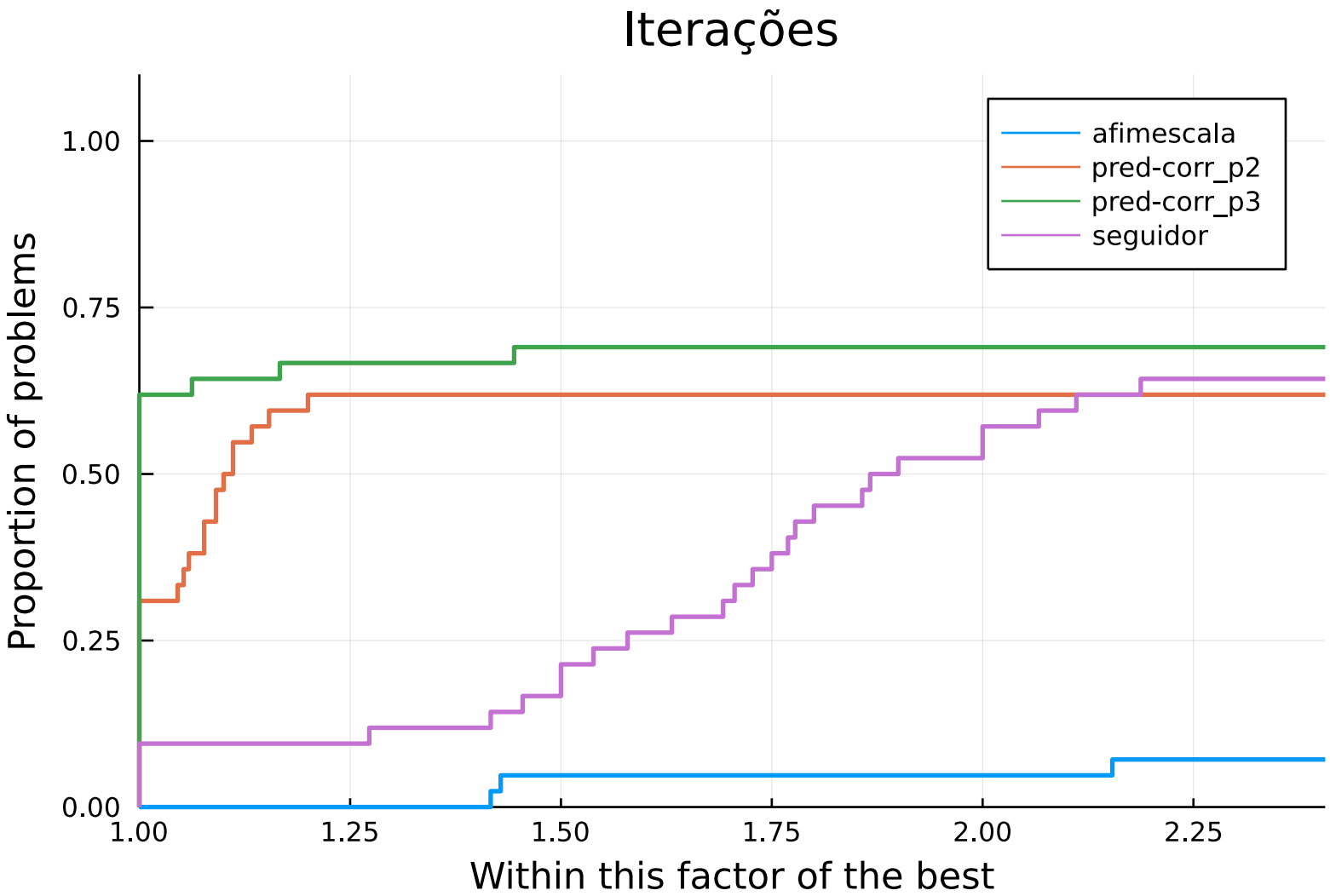
5. Gere o mesmo perfil com escala logarítmica (padrão) e salve a figura.

Exemplo – pontos interiores (Tempo CPU)



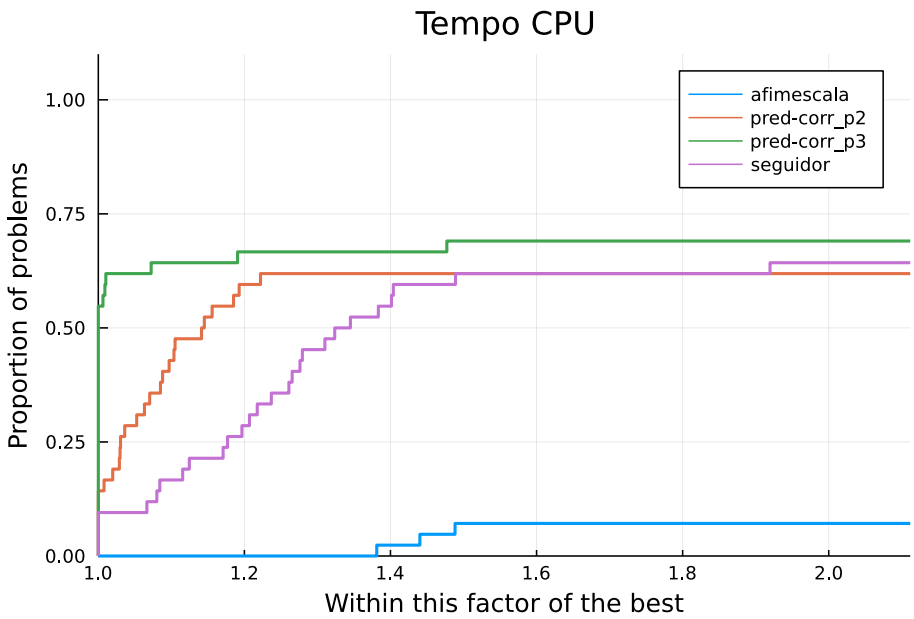
afimescala é **pior** em tempo e eficácia; *pre-corr_p3* é **melhor** em tempo e eficácia; *seguidor* é **mais lento** que *pre-corr_p3* porém, **se esperamos** $\approx 1,9$ **vezes o tempo do mais rápido** (*pre-corr_p3*), ele é mais eficaz que *pre-corr_p2*.

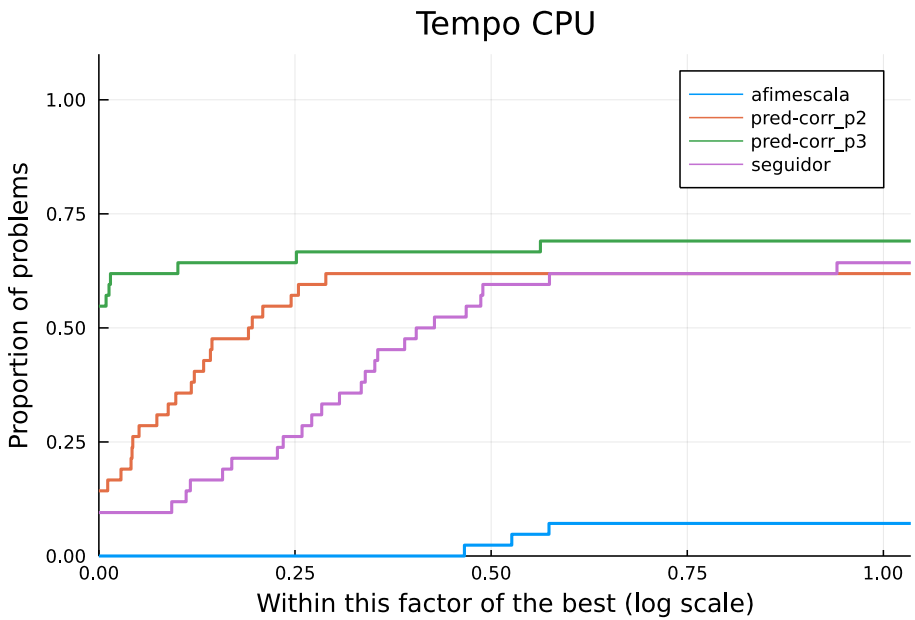
Exemplo – pontos interiores (Número de iterações)



As conclusões são parecidas com o perfil de tempo, pois todos os métodos têm iterações com custo similar.

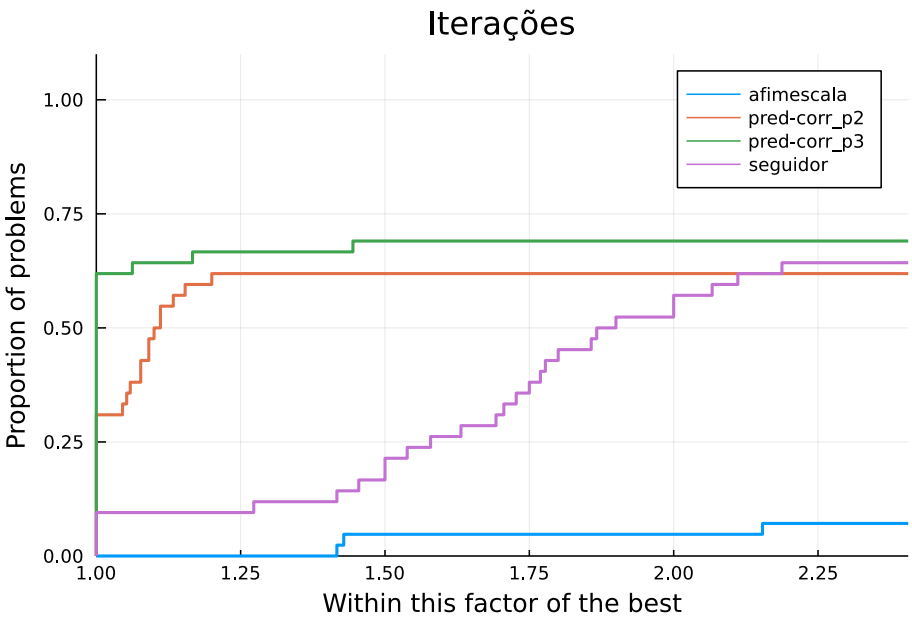
Exemplo – pontos interiores (Tempo CPU)

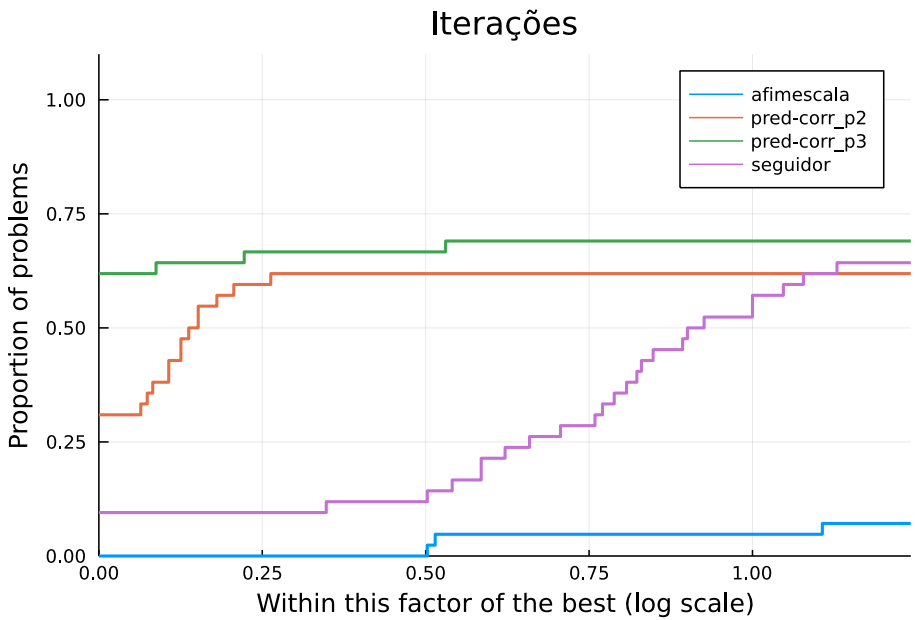




Eixo horizontal: à esquerda, $\tau \geq 1$; à direita, $\log_2(\tau) \geq 0$.

Exemplo – pontos interiores (Número de iterações)





Eixo horizontal: à esquerda, $\tau \geq 1$; à direita, $\log_2(\tau) \geq 0$.

Dicas no Julia

- É recomendável usar a estrutura *DataFrames* para gerenciar tabelas.
  - Pacote `DataFrames.jl`
  - Similar ao *dataframes* do Python
  - Veja https://leonardosecchin.github.io/juliaopt_ex12
- Para evitar erros, sempre converta os dados para `Float64` (esse tipo aceita o "número" `Inf`)
- É recomendável salvar os resultados em arquivos. Recomendo arquivos binários pois preservam a estrutura dos objetos.

  Veja https://leonardosecchin.github.io/juliaopt_ex11/
- Você pode configurar o perfil de desempenho como uma figura `Plots`.

  Veja a seção *"Configurando gráficos"* em https://leonardosecchin.github.io/juliaopt_ex5

Sobre contagem de tempo de execução

A captura do tempo de execução pode sofrer oscilações de processos do sistema, processos concorrentes, paralelismo ou alterações na frequência do processador.

Dicas para minimizá-las:

- não trabalhe na máquina em que está executando os testes;
- fique atento à *swaps*. Eles destroem qualquer contagem de tempo! Se seu computador não possui memória RAM suficiente, não insista;
- se possível, "trave" o processador na sua frequência base, evitando *TurboBoost* ou similares. Talvez requeira acesso como *root* (Linux);

Sobre contagem de tempo de execução

- se sua aplicação não requer paralelismo, pode ser bom executar cada problema com 1 *thread* apenas. Lembre-se que mesmo um programa *serial* pode ter trechos em paralelo pois vários resolvedores de sistemas lineares / pacotes de álgebra linear rodam em paralelo.

  Geralmente você pode definir o número de *threads* por variáveis de ambiente (Linux). Algumas delas:

  - `OPENBLAS_NUM_THREADS=1`, `GOTO_NUM_THREADS=1` : OpenBlas
  - `OMP_NUM_THREADS=1` : *threads* para códigos com trechos OpenMP
  - `MKL_NUM_THREADS=1` : Intel© Math Kernel Library© (BLAS da Intel para seus processadores)

  Consulte a documentação do *software* que está utilizando.

Sobre contagem de tempo de execução

Inconsistências surgem ainda de oscilações naturais na contagem

(rode um algoritmo várias vezes, você verá que o tempo é diferente a cada execução).

- Se seu algoritmo leva "muito" tempo para resolver um problema (p. ex. > 15 seg), então essas oscilações naturais não devem representar grandes variações percentuais...

- Porém, se seu algoritmo é muito rápido num dado problema (p. ex., < 1 seg), **pequenas oscilações naturais no tempo representam grandes variações percentuais.**

  Isso compromete a comparação entre algoritmos em tais problemas.

Sobre contagem de tempo de execução

Como minimizar as variações naturais?

- Utilize sempre as melhores maneiras para contar tempo (aliás, essa dica vale para problemas demorados também).

  Por exemplo, no Julia use os comandos `@time` ou `@elapsed`
  $\rightarrow$ consulte
  https://leonardosecchin.github.io/juliaopt_ex14

- Rode cada problema N vezes até somar um tempo "confortável", por exemplo, 15 seg. O tempo de execução do problema será a média aritmética

$$\frac{\text{tempo total das execuções}}{N}.$$

  Isso deve ser implementado!
  Isso minimiza **muito** o efeito de oscilações nos problemas pequenos.

- Você pode simplesmente descartar os problemas cuja execução é muito rápida **caso não sejam relevantes para sua análise.**

Exercícios, códigos e referências

Veja

https://leonardosecchin.github.io/topicospo