



# Maratona de Filmes

Aluno: Jamesson Leandro Paiva Santos

## O Problema

Inspirado nos problemas do caixeiro viajante e da mochila binária, o presente relatório explora conceitos e fundamentos de soluções de alto desempenho para buscar soluções para o seguinte problema:

*Você quer passar um final de semana assistindo ao máximo de filmes possível, mas há restrições quanto aos horários disponíveis e ao número de títulos que podem ser vistos em cada categoria (comédia, drama, ação, etc). Qual é o número máximo de filmes que podem ser assistidos de acordo com as restrições de horários e número máximo por categoria? Conseguimos preencher as 24h?*

A entrada de dados do algoritmo consiste em arquivos de texto que seguem a seguinte formatação:

```
10 4
1 3 1 2
11 13 3
14 15 3
10 16 2
10 14 1
...
```

- A primeira linha da entrada contém o número  $N$  de filmes e o número  $k$  de categorias.
- A segunda linha contém o número máximo de filmes por categoria que podem ser assistidos.
- Da terceira linha em diante, segue os filmes com a hora de início, hora de término e a categoria a qual pertence.

Esse problema será explorado aqui com a heurística **gulosa** e a **aleatoriedade**.

## Considerações

Há um gerador de inputs que foi disponibilizado pelos professores da disciplina para criar os arquivos de texto de entrada. A única adaptação feita na geração foi de desconsiderar os casos em que o filme tem hora de início e fim iguais. Isso foi feito porque, para um filme que começa às 17 e termina às 18, por exemplo, considera-se que ele "preenche" o slot das 17, então outro filme que comece às 18 pode ser selecionado. Assim, considerar os filmes com horário de início e fim iguais causaria uma ambiguidade e problemas na implementação do algoritmo.

Além disso, os filmes que começam em um dia e terminam em outro foram considerados para o algoritmo. A restrição imposta é que o tempo total de tela (soma das durações dos filmes selecionados) não ultrapasse 24 e que um não se sobreponha ao outro (por exemplo, é permitido que um filme termine às 10 e outro inicie nessa mesma hora, mas não é permitido que um que comece às 10 e termine às 12 seja inserido se já existe um que começa às 7 e termina às 11, pois o horário de 10 já está preenchido).

## Sobre o código

Na raiz do projeto, encontram-se todos os arquivos `.cpp` e `.py` para a execução do projeto. Os arquivos `gulosa.cpp` e `aleatorio.cpp` contém o \*core\* do trabalho, e o arquivo `functions.cpp` contém funções que são compartilhadas por ambos. Ao serem executados, os arquivos geram saídas de texto (para *logs*) e um arquivo `.json` usado para apresentar os resultados.

Formato dos arquivos `.json`:

```
{  
    "exec_time": 87,  
    "num_categories": 10,  
    "num_movies": 200,  
    "screen_time": 18  
}
```

Formato dos arquivos `.txt`:

```
TEMPO DE EXECUÇÃO: 75  
TEMPO DE TELA: 18  
FILMES SELECIONADOS COM ALEATORIEDADE  
ID: 85, categoria: 13, starts at: 10, ends at: 12  
ID: 12, categoria: 9, starts at: 1, ends at: 4  
ID: 11, categoria: 10, starts at: 16, ends at: 19  
ID: 33, categoria: 14, starts at: 6, ends at: 8  
ID: 88, categoria: 13, starts at: 21, ends at: 23  
ID: 90, categoria: 7, starts at: 12, ends at: 15  
ID: 67, categoria: 8, starts at: 9, ends at: 10  
ID: 31, categoria: 1, starts at: 19, ends at: 21
```

O executável python faz as seguintes tarefas:

1. Gera os arquivos de input.
2. Gera as saídas das heurísticas usando os arquivos de input gerados.

3. Gera gráficos que mostram tempo de execução, número de filmes e categorias.

A seguir, contém o trecho inicial do código python:

```
class Knapsack():
    def __init__(self) -> None:
        self.fix_movie = 100      # Número de filmes no teste 1
        self.fix_category = 20    # Número de categorias no teste 2

        self.n_movies = np.arange(20, 1020, 20) # array com número de filmes
        self.n_cats = np.arange(2, 30, 1) # array com número de categorias

        self.results_t1_gulosa = None # resultados experimento t1 para gulosa
        self.results_t2_gulosa = None # resultados experimento t2 para gulosa

        self.results_t1_aleatoria = None # resultados experimento t1 para aleatória
        self.results_t2_aleatoria = None # resultados experimento t2 para aleatória
```

Aqui serão trabalhados 2 experimentos, o experimento T1 (varia-se o número de categorias, mantendo fixo o número de filmes) e o T2 (varia-se o número de filmes e mantém constante o número de categorias). A ideia dos experimentos é analisar como se comporta o tempo de execução dos algoritmos conforme variamos uma das métricas, além de comparar a *performance* dos algoritmos entre si.



Para evitar ficar gerando novos inputs a cada execução, vá na função `run` do arquivo python e comente a linha `self.generate_input_files()`.

## Heurística Gulosa

A implementação da heurística gulosa foi feita tomando a lista de filmes e ordenando-a em ordem crescente de tempo de fim. De posse da lista ordenada, o algoritmo percorre os filmes e popula a lista dos selecionados, levando em conta se o horário está disponível e se ainda pode ser inserido o filme para a respectiva categoria.

O trecho de código a seguir mostra a lógica principal do algoritmo:

```
for (auto &movie : movies)
{
    // Tempo máximo de 24h
    if (total_time >= 24)
    {
        return;
    }

    if (movie.begin >= last_selected.end &&
        max_cat[movie.category] > 0 &&
        has_time_available(times_available, movie))
    {
        solution.push_back(movie);
        update_availability_list(times_available, movie);

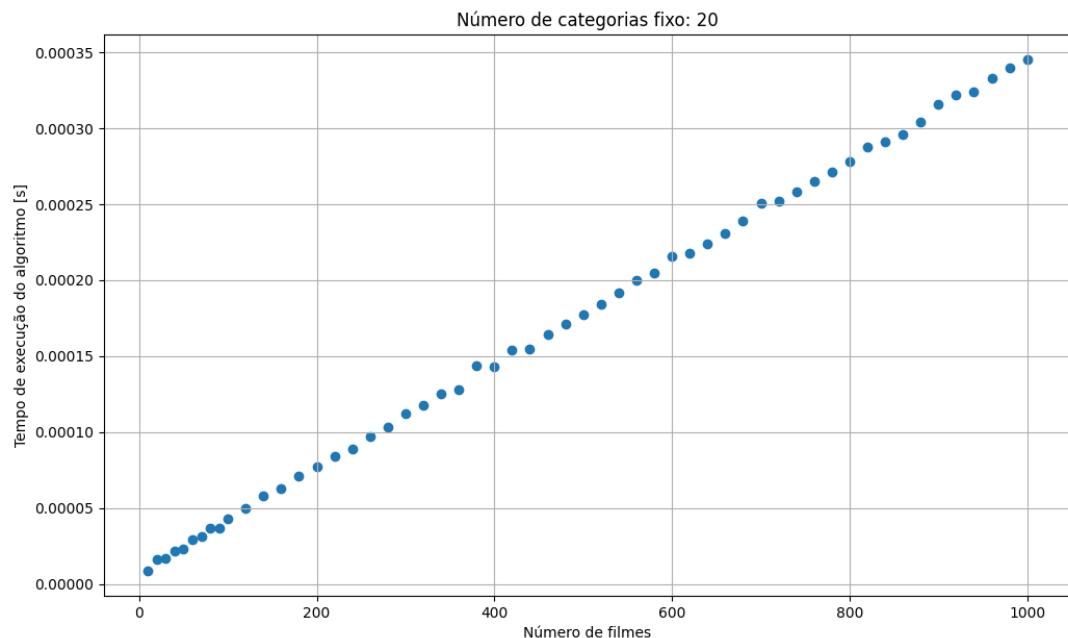
        max_cat[movie.category]--;
        total_time += movie.duration;
        last_selected = movie;
    }
}
```

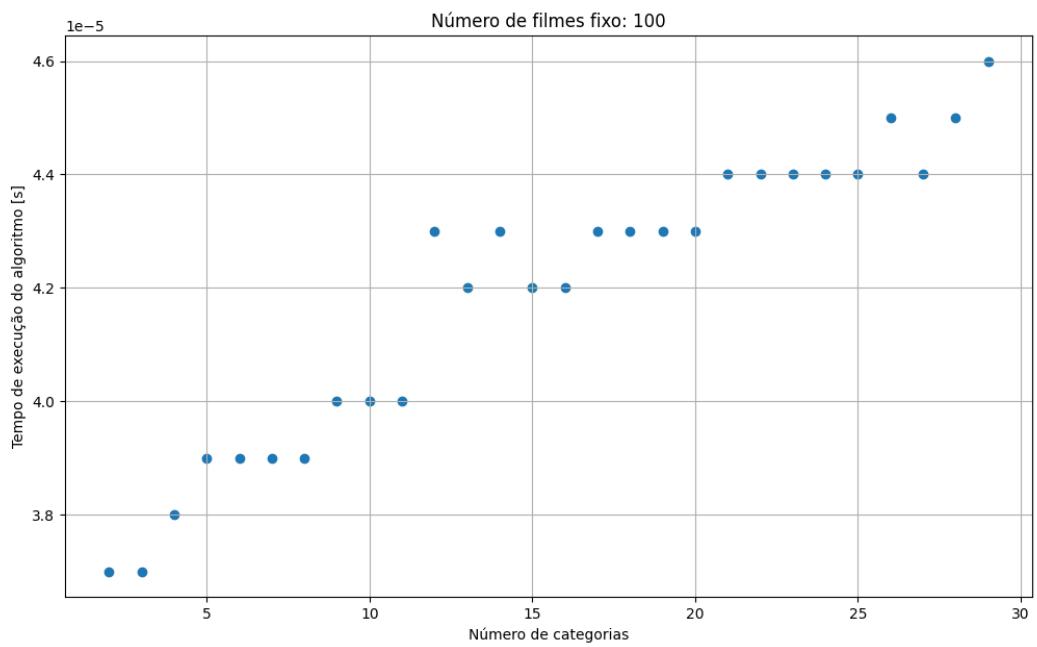
```
    }  
}
```

Após *inputar* os dados e ordenar o vetor de filmes, esse vetor é percorrido e, se o filme começa a partir do início do anterior, ainda há limite disponível para sua categoria e sua duração não *\*cruza\** com nenhum outro filme já selecionado, então esse filme entra para a lista dos selecionados.

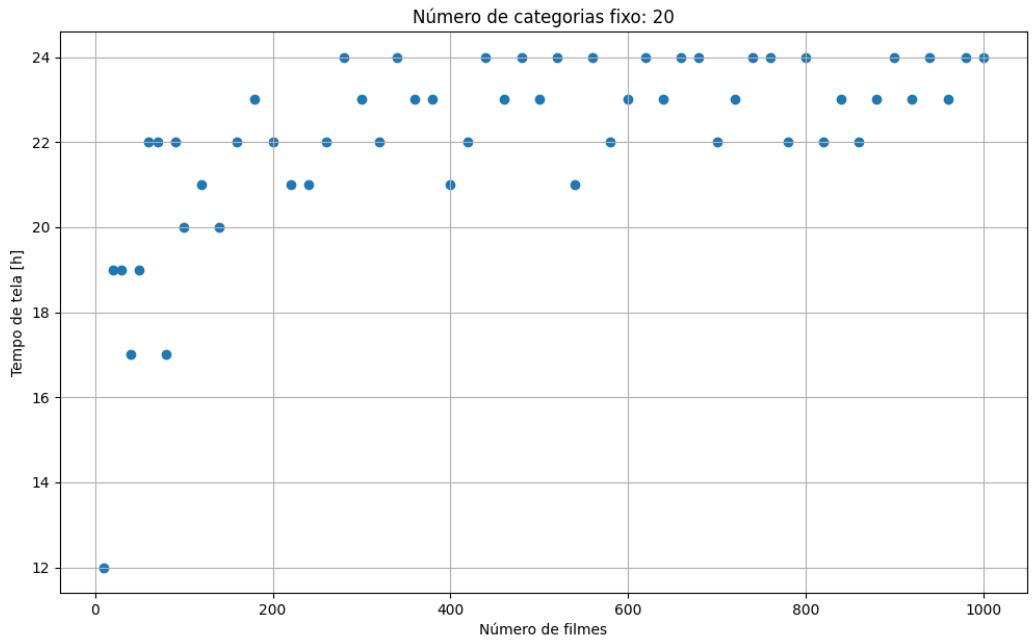
## Resultados

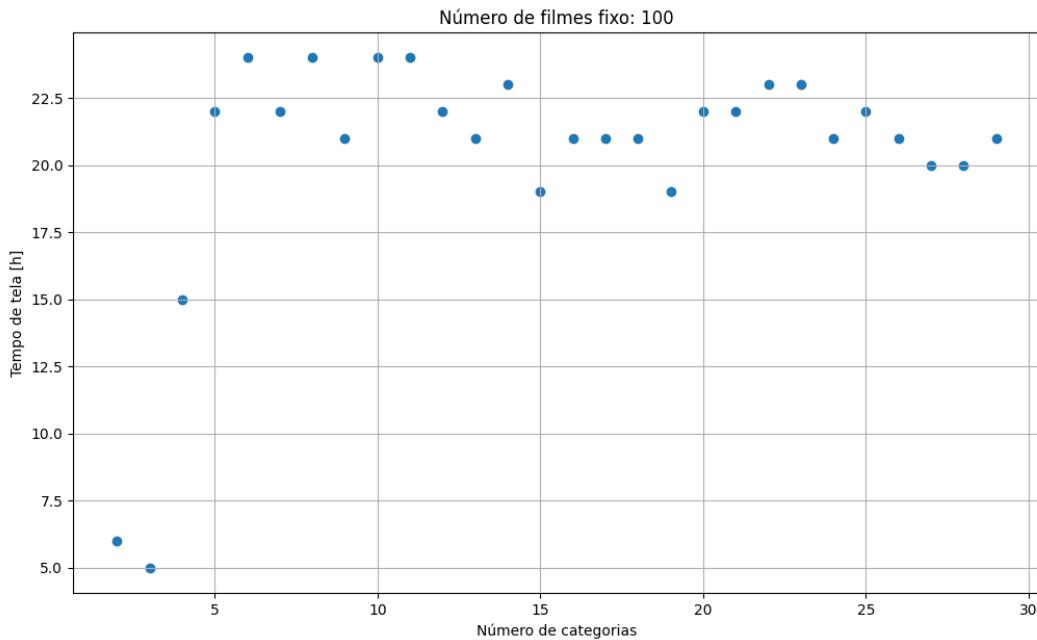
Os resultados a seguir foram obtidos executando o código para as entradas descritas anteriormente.





O tempo de execução do algoritmo quando o número de categorias é fixo é praticamente linear com o número de filmes. Já quando se fixa o número de filmes, o tempo total de execução parece crescer em pequenos degraus, a cada 4 categorias aproximadamente.





Quanto ao tempo de tela, parece que os resultados foram melhores quando se fixou o número de categorias. Com mais filmes, obteve-se mais vezes o tempo de tela em 24h.

## Análise do Valgrind

Ao executar o `valgrind` com o executável `gulosa`, o trecho do resultado (arquivo `callgrind.out.17790`) obtido chama a atenção:

```
1,014,826 (40.03%) ./elf./elf/dl-lookup.c:_dl_lookup_symbol_x [/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2]
 592,804 (23.38%) ./elf./elf/dl-lookup.c:do_lookup_x [/usr/lib/x86_64-linux-gnu/ld-linux-x86-64.so.2]
 264,358 (10.43%) ./elf.../sysdeps/x86_64/dl-machine.h:_dl_relocate_object
```

As funções apontadas no resultado do *valgrind* são funções do sistema operacional. Por exemplo, a função `_dl_lookup_symbol_x` está ligada ao carregamento de bibliotecas compartilhadas, enquanto que a `do_lookup_x` cuida da resolução de símbolos em chamadas de função ou variável de bibliotecas compartilhadas.

## Aleatoriedade

A implementação da aleatoriedade é feita a partir da heurística gulosa, mas agora considerando que, em cada iteração na lista de filmes, há 25% de pegar outro filme qualquer que satisfaça a condição de horário e disponibilidade por categoria.

O trecho abaixo contém a lógica principal do algoritmo de aleatoriedade:

```

for (int i = 0; i < num_movies; i++)
{
    // Tempo total atingido
    if (total_time >= 24)
        return;

    double random_prob = distribution_real(engine);

    // Se P > 25%, escolhemos um filme ao acaso
    if (random_prob > prob)
    {
        uniform_int_distribution<int> distribution_int(i, num_movies - 1);
        int random_index = distribution_int(engine); // sorteia um índice

        // Se ainda há vaga por categoria e horário, adiciona
        if (max_cat[movies[random_index].category] > 0 &&
            has_time_available(times_available, movies[random_index]))
        {
            solution.push_back(movies[random_index]);
            update_availability_list(times_available, movies[random_index]);

            max_cat[movies[random_index].category]--;
            total_time += movies[random_index].duration;
            last_selected = movies[random_index];
            i--;
        }
    }

    // Se P < 25%, prossegue com a heurística
    else
    {
        // Se há vaga na categoria e tempo, adiciona
        if (has_time_available(times_available, movies[i]) &&
            max_cat[movies[i].category] > 0)
        {
            solution.push_back(movies[i]);
            update_availability_list(times_available, movies[i]);

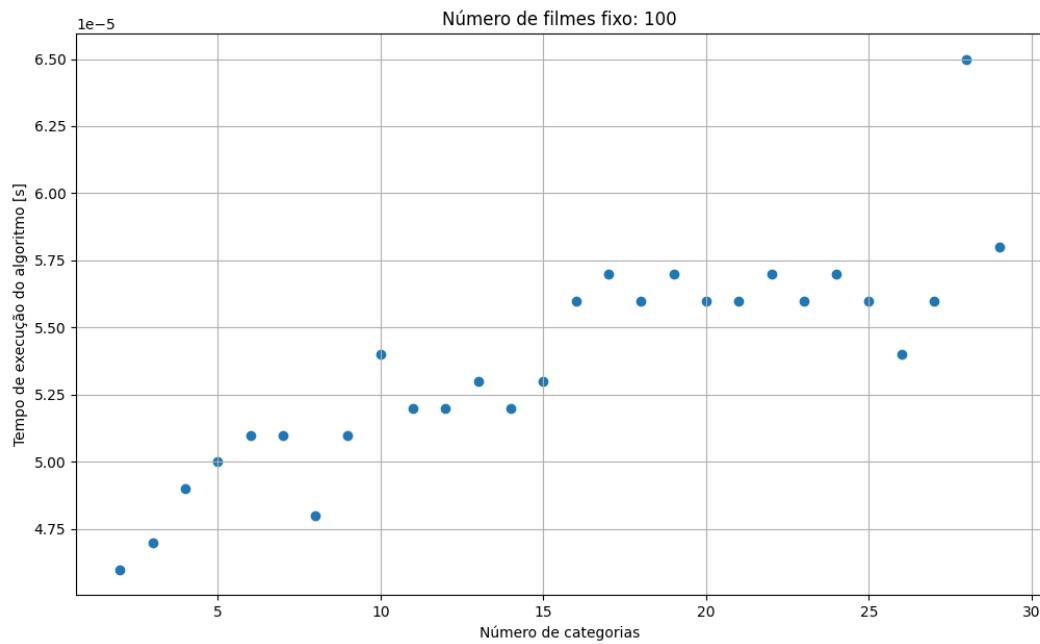
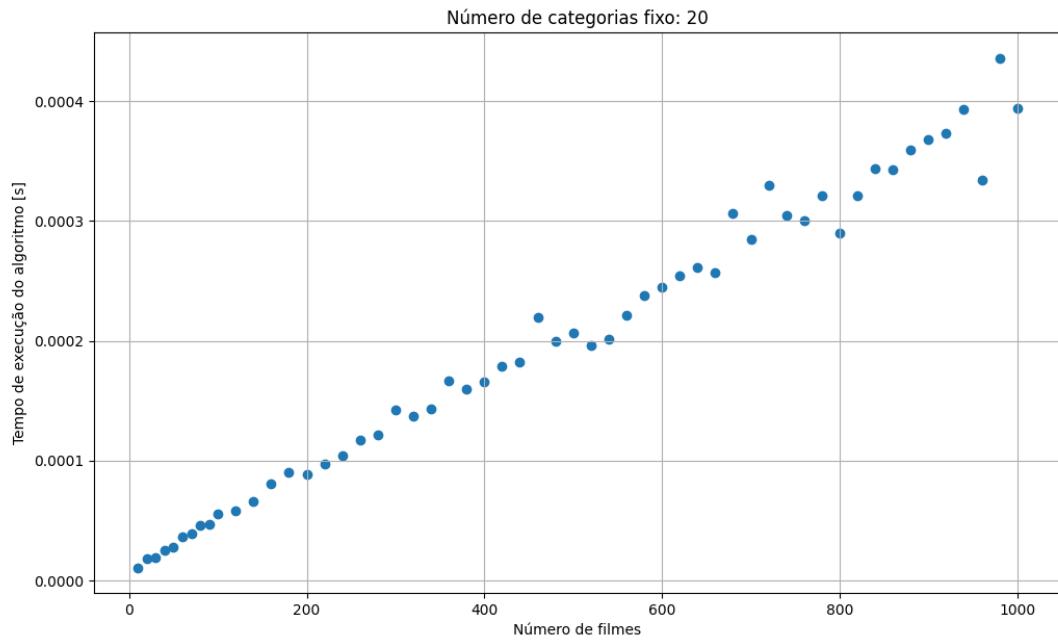
            max_cat[movies[i].category]--;
            total_time += movies[i].duration;
            last_selected = movies[i];
        }
    }
}

```

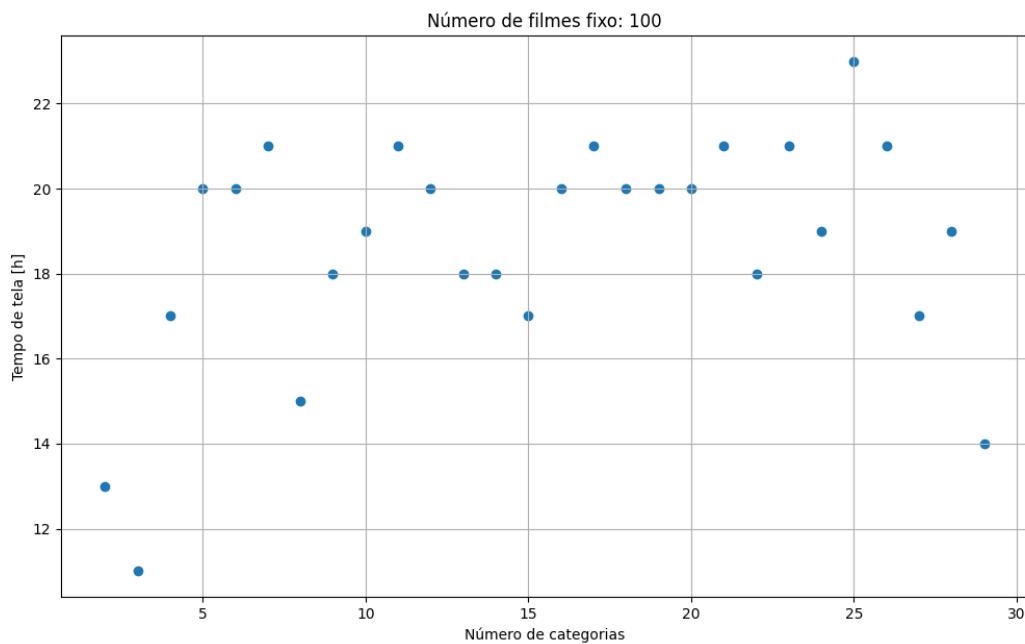
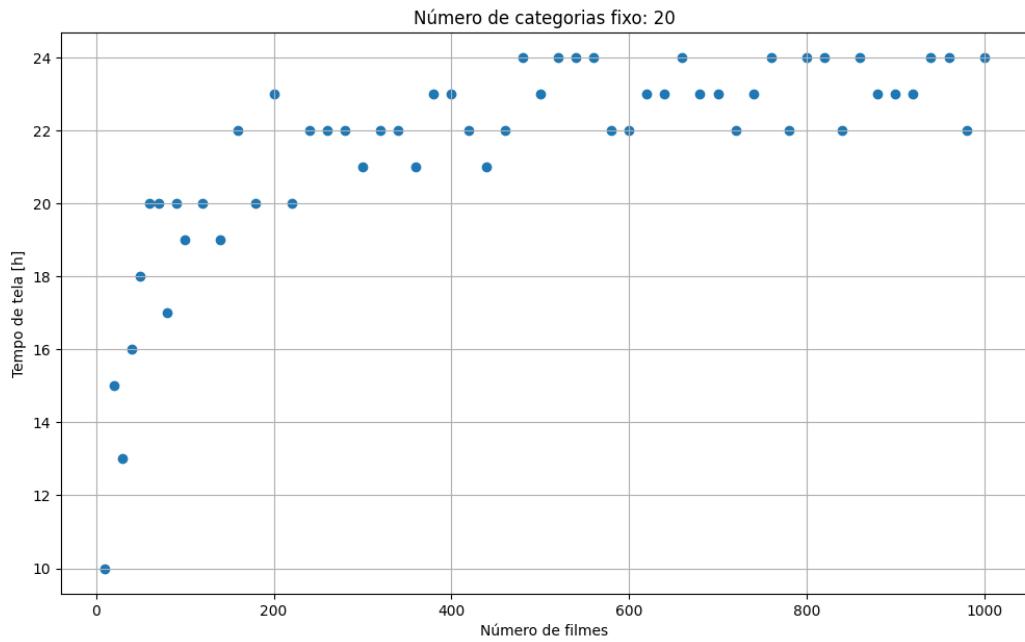
No código acima, se o valor gerado for superior a 0.25, escolhe-se um filme ao acaso e o insere na lista de selecionados, desde que ele cumpra os requisitos já definidos. Caso contrário, segue-se a heurística gulosa.

## Resultados

Os resultados a seguir foram obtidos executando o código para as entradas descritas anteriormente.



Os resultados obtidos para aleatoriedade são semelhantes aos da heurística gulosa, principalmente para o caso em que se fixou o número de categorias. Quando se ficou o número de filmes, não houve grandes mudanças no tempo de execução entre 15 e 30 categorias.



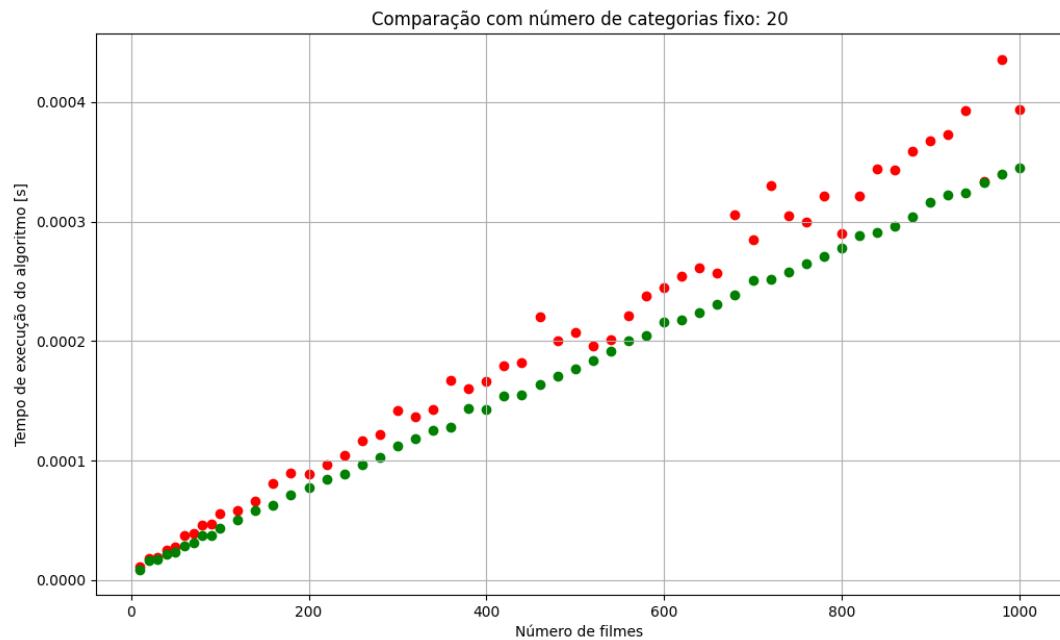
Aqui, o tempo de tela também foi melhor fixando o número de categorias, enquanto que, no caso do número de filmes constante, o tempo total praticamente não chega a 24.

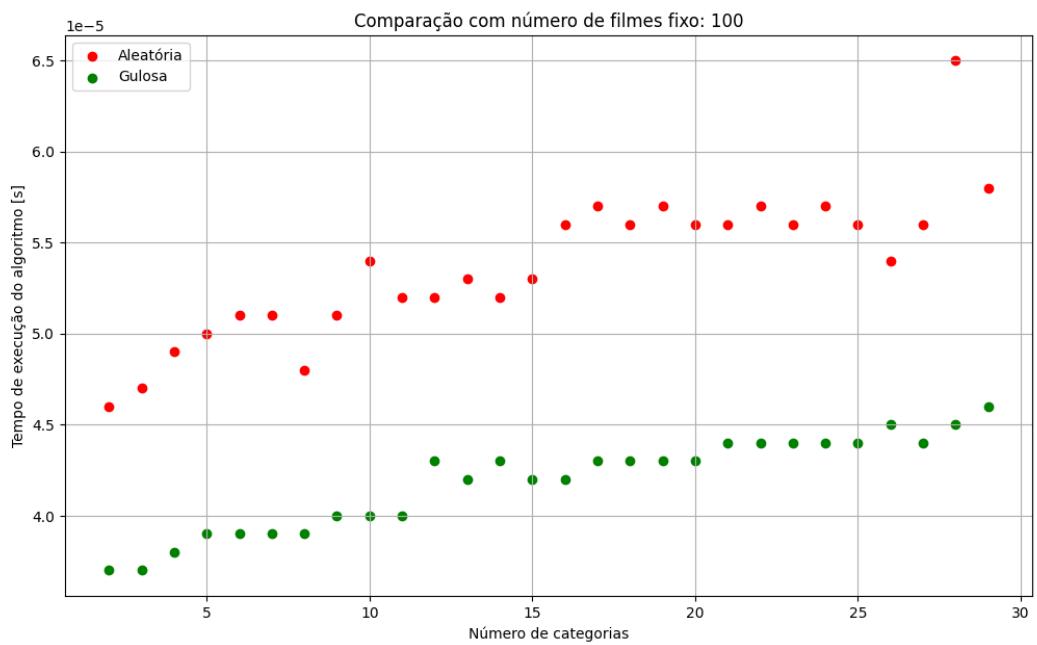
## Análise do Valgrind

Curiosamente, obteve-se o mesmo resultado obtido para a heurística gulosa aqui, dentre os casos com maior número de execuções. Isso indica que os algoritmos estão consumindo bastante recursos e tempo de execução na parte de carregamento de bibliotecas compartilhadas.

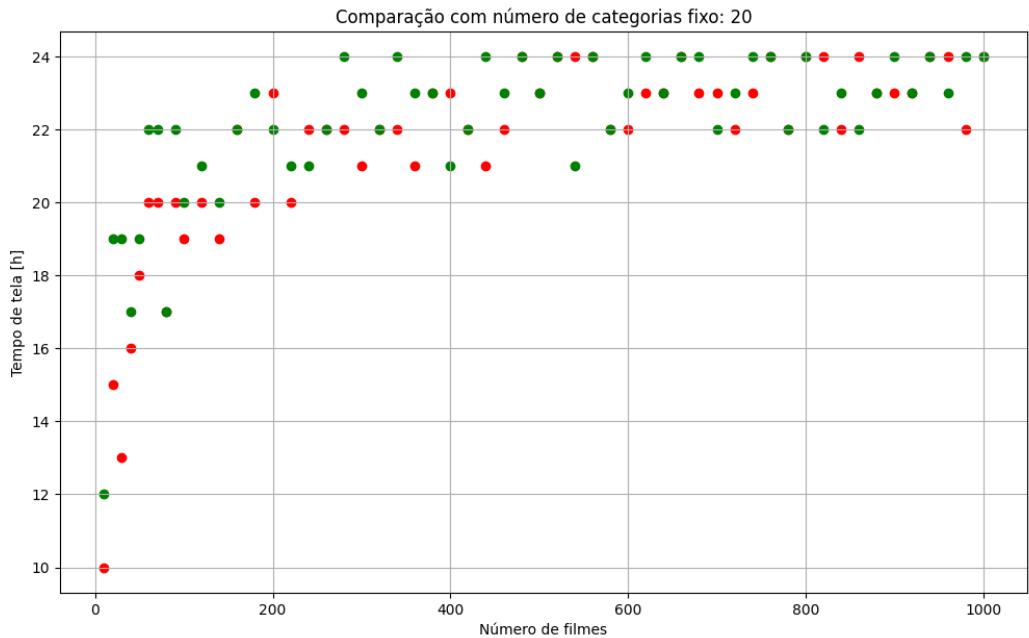
## Comparação

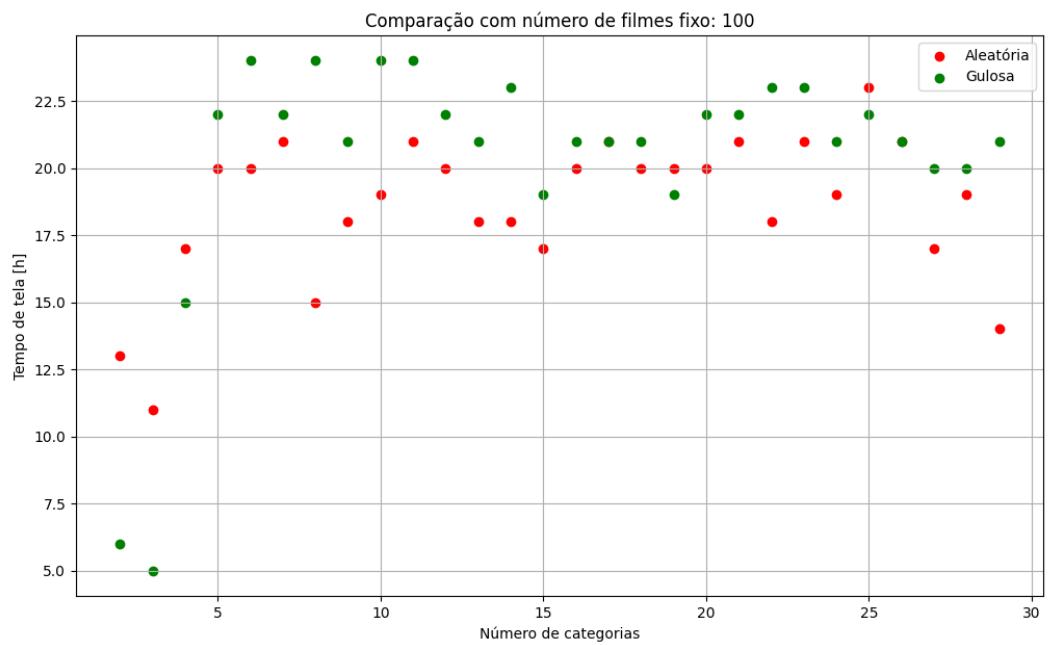
Para visualizar melhor os gráficos anteriores com fins comparativos, os gráficos abaixo são os mesmos anteriores, apenas unindo os dois casos tratados:



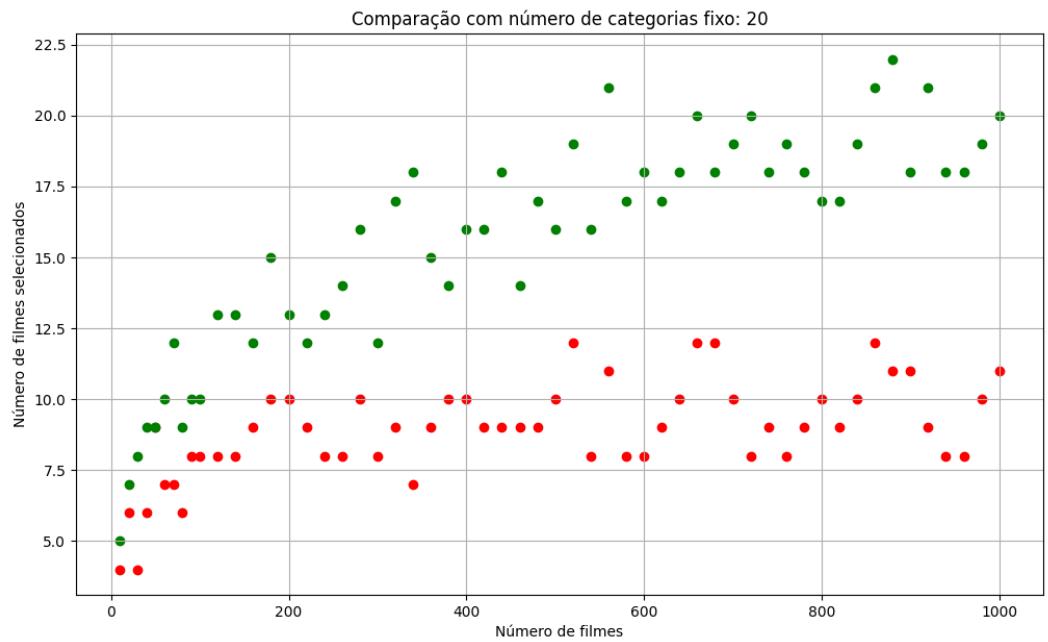


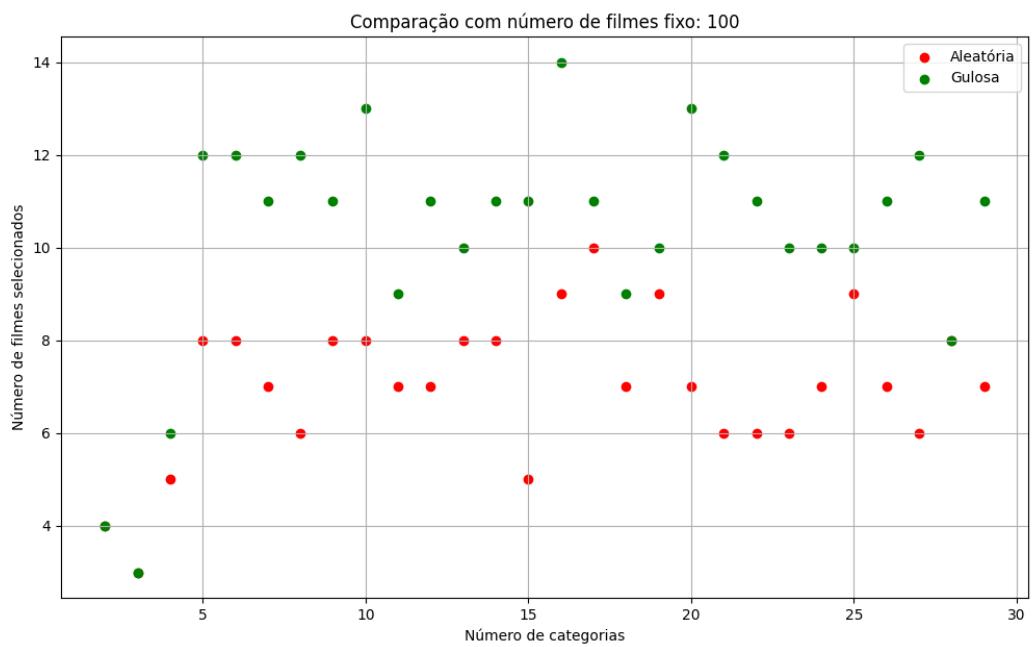
Na primeira imagem, o tempo de execução é muito semelhante para ambos, notando algumas pequenas “perturbações” no caso aleatório, o que é, de certa forma, esperado. Já na segunda figura nota-se uma diferença mais clara entre as duas: o tempo de execução para a heurística gulosa é menor.





Em termos de tempo de tela, ambas são muito próximas, mas nota-se que, no caso em que o número de filmes é fixo, a heurística gulosa alcança mais vezes o tempo de tela máximo (24 horas).





Por fim, comparando o número de filmes selecionados em ambos os casos, vemos que a heurística gulosa contemplou melhores resultados, especialmente quando se varia apenas o número de filmes.