Projeto de Escalonamento

Cecilia Fernandes
Lucas Cavalcanti dos Santos

Prof. Alfredo Goldman

17 de dezembro de 2009

Sumário

1 Introdução

Um escalonamento é uma forma de dividir a execução de tarefas em diferentes recursos.

É um conceito que, no contexto de linhas de produção fabris, é usado para diminuir custos de matéria-prima ou para aumentar a velocidade ou capacidade de produção. Dadas tarefas que máquinas sabem executar, como dividi-las de forma que essas restrições sejam atendidas de maneira eficiente.

Em computação, o escalonamento de tarefas é usualmente associado à divisão do tempo de um processador ou mesmo na divisão de processos em diferentes *cores* de máquinas com mais de um processador. Ainda há a extensa e vital utilização de escalonamento na paralelização de tarefas em diferentes máquinas.

Cada problema de escalonamento tem características específicas e, portanto, heurísticas e algoritmos diferentes que se aplicam melhor a cada um.

Outros trabalhos também desenvolvidos nessa matéria estudaram as diversas heurísticas, meta-heurísticas e restrições aplicáveis a problemas de escalonamento. Nesse trabalho, criamos a infraestrutura para que seja fácil implementar tais heurísticas e vê-la funcionando na prática.

De modo lúdico, com uma motivação para o problema, abordaremos as formas de escalonar programas de televisão de um canal de forma a atender às necessidades ou desejos de uma família tida como "modelo", apresentada ao leitor na seção de Motivação.

Em seguida, mostraremos as Heurísticas implementadas explicando a idéia do algoritmo e mostrando *screenshots* dos escalonamentos gerados por essas heurísticas para uma determinada entrada do problema.

Depois, vêm as seções de Desenvolvimento, onde explicamos como a ferramenta desenvolvida é facilmente estensível para implementação de novos escalonamentos, e de Outras Restrições que poderiam ser adicionadas ao problema de modo a deixá-lo mais realista e interessante.

Finalmente, os Passos Futuros em direção aos quais o projeto pode caminhar são mencionados e a conclusão que tiramos com relação às heurísticas implementadas e ao problema abordado.

2 Motivação

Como já havia muitos trabalhos teóricos e achamos importante e motivante (além de gostarmos de código) implementar os algoritmos e vê-los rodando na prática, optamos por criar um sistema simples que conseguisse ler uma entrada e mostrar o escalonamento de forma simples e clara.

Contudo, antes de começar o código, sentimos a necessidade de focar o projeto em um problema menor e mais definido que, embora provenha uma boa guia para a implementação, não limita o projeto à resolução específica desse cenário.

De forma a tornar ainda mais fácil o entendimento do problema, criamos personagens, cada um com suas preferências de horários e programação.

2.1 Os personagens

Temos uma família de três pessoas com perfis bem específicos:

- uma criança que assiste TV de manhã, preferencialmente desenhos e programas infantis, estuda à tarde e dorme cedo;
- uma esposa que, de manhã, faz ginástica e trabalho voluntário e de tarde assiste TV, principalmente programas femininos e seriados;
- um marido que trabalha o dia inteiro, chega a noite em casa e gosta de assistir notícias e esportes, principalmente futebol.

Esses personagens têm uma televisão e alguns problemas de escalonamento surgem de seus desejos ou para sua conveniência.

Problema: Montagem da grade de canais

"Com as preferências de programa e horário, os canais de TV organizam seus programas, de modo que melhor atendam às necessidades da família."

Modelagem: $(P_\infty|p_j, r_j, d_j|\sum_j U_j)$

Todos os programas têm *release date* (r_j) e *due date* (d_j) dentro de períodos bem definidos, de acordo com sua categoria:

- programas infantis, desenhos: $r_j = 6h$ e $d_j = 12h$
- programas femininos: $r_j = 12h$ e $d_j = 18h$
- séries, notícias e esportes: $r_j = 18h$ e $d_j = 24h$

Os programas têm tamanhos variados, informados na entrada, e cada máquina representa um dia da semana/mês. Com um escalonamento, podemos gerar a grade de programação de vários dias.

Nesse problema, não há preempção dos programas, isto é, eles não podem começar em um dia e terminar em outro ou mesmo começar em um período e terminar em outro. Assim, sobra tempo ocioso da emissora no final de um dia.

A fim de evitar essa sobra dispendiosa de tempo em um dia, em vez de adicionar a preempção, que não traria nenhum desafio e cujo algoritmo trivial dá solução ótima, decidimos por espalhar, na programação, intervalos comerciais.

Problema: Montagem da grade de canais com intervalos comerciais

> *"Dada essa grade, em vez de deixar tempo sobrando, quero ganhar dinheiro com comerciais espalhados pela programação. Se não tiver nada pra passar num período inteiro, quero vender esse período para programação especializada de vendas."*

A modelagem desse problema é exatamente a mesma do problema anterior, mas a resolução exige uma nova heurística que considere essa restrição. A forma como isso foi implementado, ela pode ser usada com qualquer outro escalonamento que se crie.

A restrição está acessível, no *software*, como segundo ítem do menu superior e, no gráfico exibido, pode-se descobrir o que é comercial e o que é programa colocando o *mouse* sobre o intervalo de uma cor: ele mostrará a legenda que indica se é um programa ou um comercial.

Via de regra, o comercial é indentificável por seu tamanho, menor do que o programa – faz-se exceção aos programas extremamente curtos.

3 Heurísticas

O problema de escalonar programas em uma grade de programação de uma emissora de TV a cabo é um caso particular de um problema bastante conhecido de teoria da combinatória, e que também pode ser visto como um problema de escalonamento: o *Bin-packing*.

Esse problema é conhecidamente NP-difícil, então não é possível conseguir a solução ótima em um tempo razoável. Por causa disso, ao invés de tentarmos descobrir a solução ótima, temos que ir atrás de uma solução que é apenas satisfatória, uma aproximação. Para conseguir uma aproximação precisamos usar heurísticas.

Heurísticas são algoritmos de aproximação, ou seja, algoritmos que não buscam encontrar a solução ótima, mas uma solução que consiga se aproximar da ótima segundo algum fator. Esse fator é a razão entre a solução aproximada e a ótima, significando o quão boa é a aproximação.

Uma característica importante de heurísticas é que elas tomam decisões específicas quando chegam em determinado ponto do algoritmo. Por exemplo escolher o maior item, inverter a ordem de itens da solução, etc.

Para conseguir uma solução aproximada para o problema *Bin-packing* podemos utilizar várias heurísticas. Algumas delas foram implementadas e rodadas com a mesma entrada.

3.1 First Fit

O *First Fit* coloca o item no primeiro bin em que ele cabe. No caso do problema da grade de programação, o programa é colocando no primeiro dia em que ele cabe. Caso o programa não caiba em nenhum dos dias, é criado um novo dia.

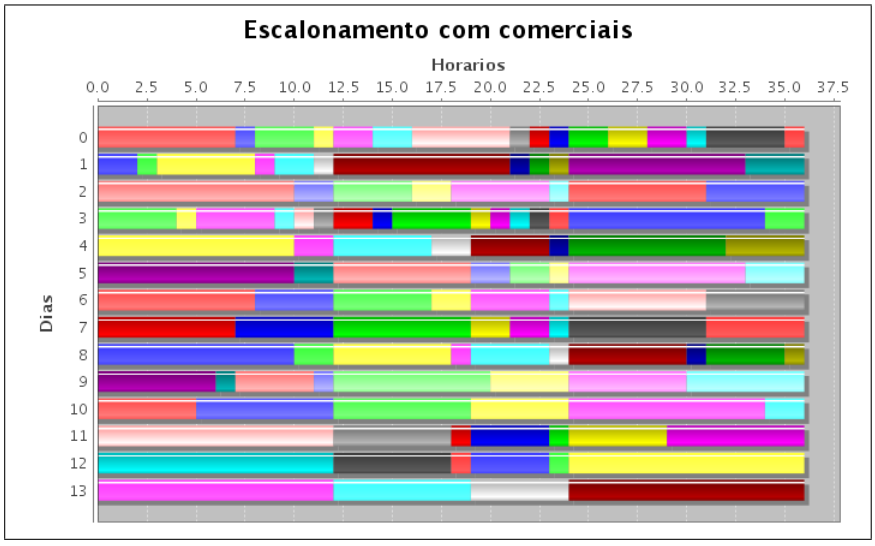

Figura 1: Resolução de um problema usando o First Fit

3.2 First Fit Decreasing

O *First Fit Decreasing* é uma variação do First Fit em que os itens são ordenados em ordem decrescente de valor antes de rodar o algoritmo. Essa heurística só pode ser usada se todos os itens forem conhecidos de antemão.

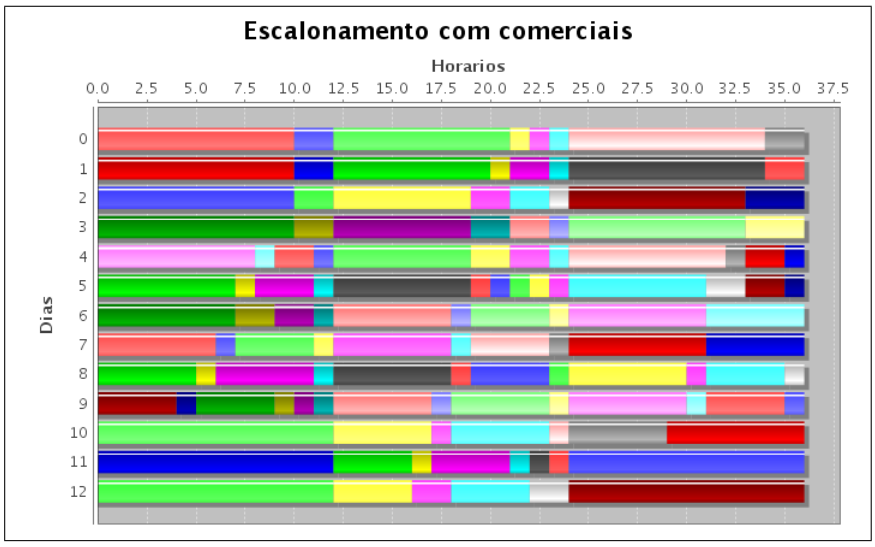

Figura 2: Resolução de um problema usando o First Fit Decreasing

3.3 Next Fit

O *Next Fit* tenta preencher um bin usando o primeiro item que caiba, se nenhum programa couber, um novo bin é criado. A principal diferença entre o *Next Fit* e o *First Fit* é que no primeiro é preenchido um bin por vez e no segundo tenta-se colocar o item no primeiro bin que ele caiba.

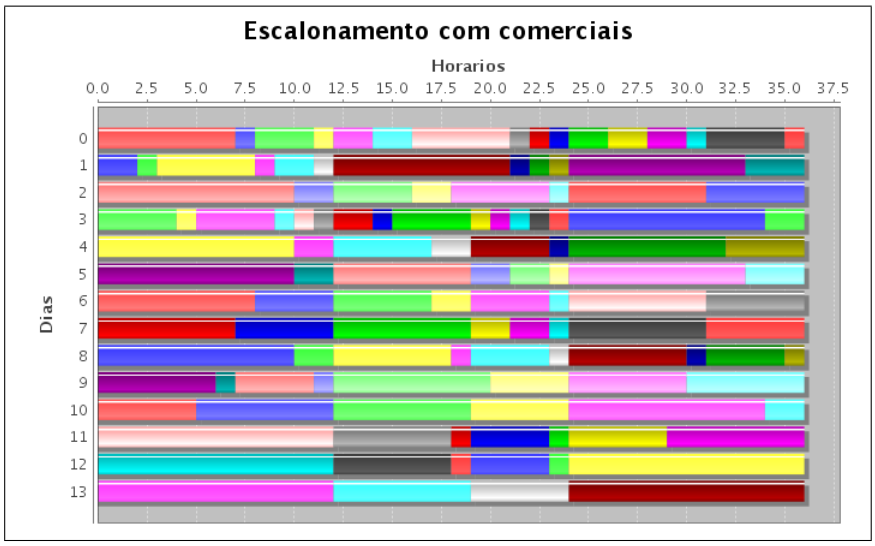

Figura 3: Resolução de um problema usando o Next Fit

3.4 Best Fit

O *Best Fit* tenta colocar o item no bin em que, após colocar o item, sobre o mínimo de espaço restante no bin. Caso não caiba em nenhum bin, é criado um novo.

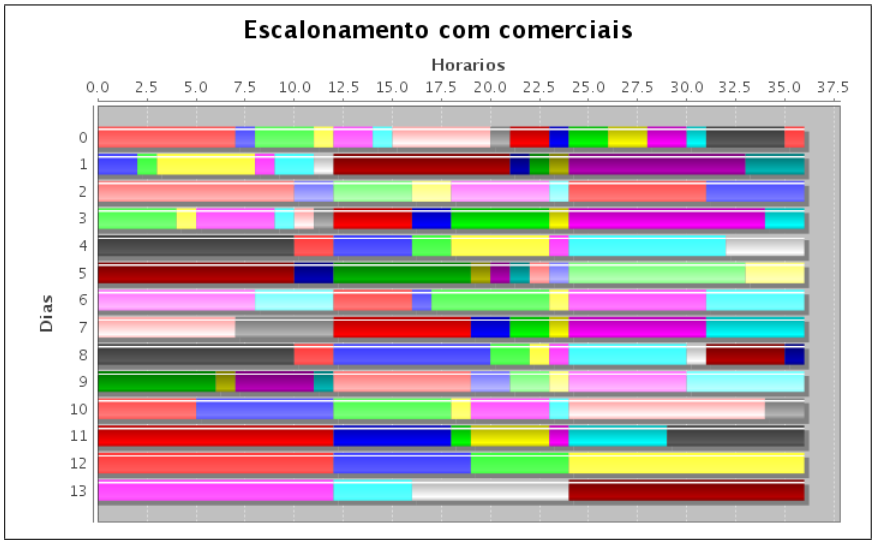

Figura 4: Resolução de um problema usando o Best Fit

3.5 Best Fit Decreasing

O *Best Fit Decreasing* é uma variação do *Best Fit* em que os itens são ordenados em ordem decrescente de valor antes de rodar o Algoritmo.

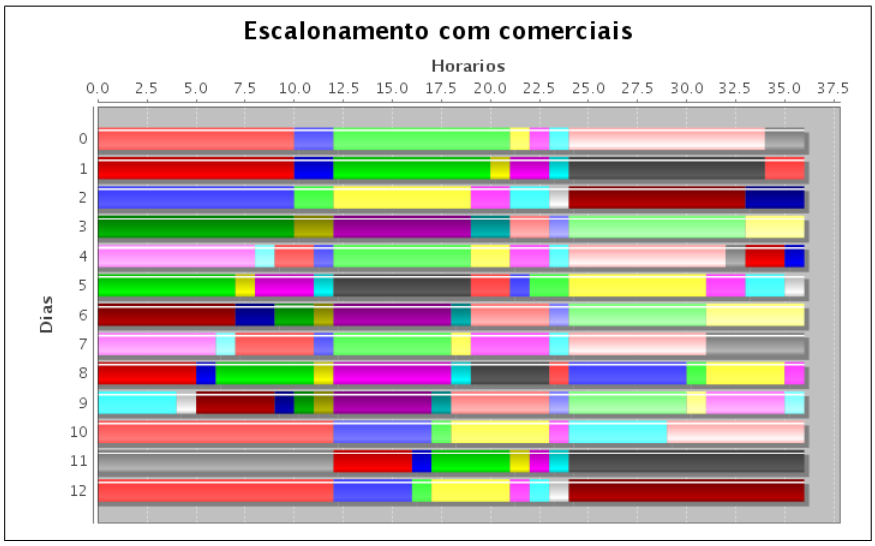

Figura 5: Resolução de um problema usando o Best Fit Decreasing

3.6 Worst Fit

O *Worst Fit* tenta colocar o item no bin em que, após colocar o item, sobre o máximo de espaço restante no bin. Caso não caiba em nenhum bin, é criado um novo.

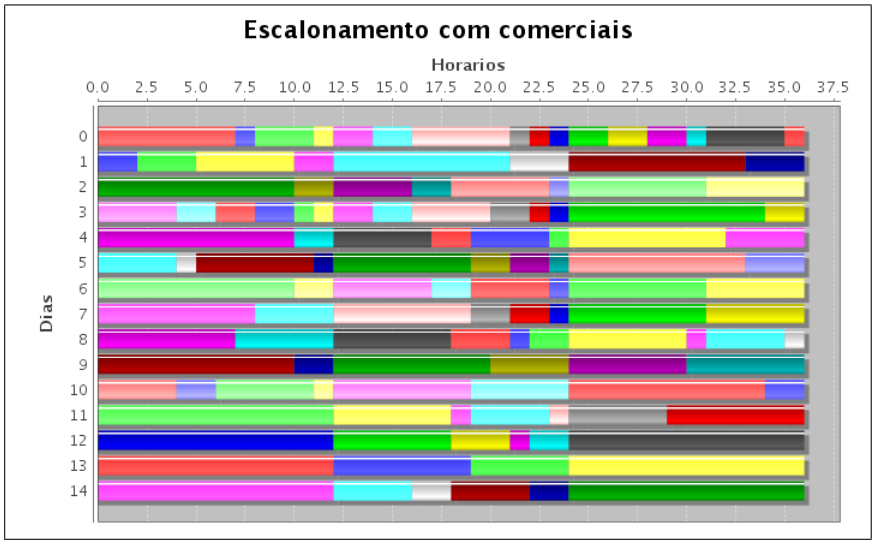

Figura 6: Resolução de um problema usando o Worst Fit

3.7 Almost Worst Fit

O *Almost Worst Fit* é semelhante ao *Worst Fit*, mas escolhe-se o bin que tem o segundo maior espaço restante. Caso caiba em apenas um bin o item é colocado nele, e caso não caiba em nenhum, um novo bin é criado.

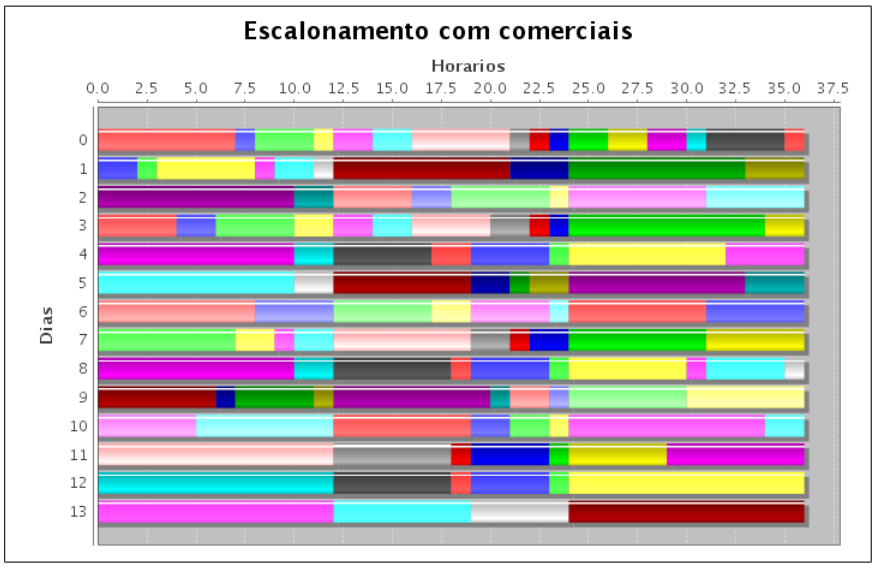

Figura 7: Resolução de um problema usando o Almost Worst Fit

4 O programa

Para que a implementação das heurísticas vá além do problema proposto e alcance diversos outros problemas da área de escalonamento, tomamos algumas decisões que deixaram o código particularmente fácil de estender.

4.1 Tecnologias usadas

Para o desenvolvimento desse programa, foi escolhida a linguagem de programação Java, amplamente difundida no mercado e também no meio acadêmico e, para a interface gráfica, foi usado Swing.

Na geração de gráficos para mostrar os escalonamentos, utilizamos a biblioteca *open source* JFreeChart, que gera gráficos de forma bastante simples e provê vantagens como *zoom* e legendas já por padrão.

4.2 Gerador de entradas aleatórias

O sistema conta com uma classe chamada `Gerador` que, quando rodada, cria no arquivo `testeAleatorio.txt` uma entrada (quase) aleatória para o problema proposto.

Isso nos ajudou a verificar semelhanças e diferenças entre heurísticas variadas e notar que, para algumas instâncias, as heurísticas chegam a um mesmo resultado apenas por coincidência. Nos poupou bastante tempo de correção de supostos *bugs* que, na verdade, não existiam.

4.3 Flexibilidade

Criar um novo escalonador, isto é, passar para código o algoritmo relacionado à uma nova heurística, meta-heurística ou restrição, é bastante simples nesse sistema.

Basta que o desenvolvedor implemente a interface `Escalonador` e adicione ao menu correto, a opção que o chama.

Compondo heurísticas, uma nova delas pode ser usada com qualquer outro escalonamento que se crie. Ela é apenas um Decorador [1] que recebe um outro escalonador base.

Cada escalonador, por sua vez, é uma Estratégia [2] para lidar com as listas de programas por períodos.

Usando esses padrões de projeto, a arquitetura do sistema se manteve simples, mas ganhou uma enorme flexibildade, excelente para implementar variações simples de um algoritmo e expor a diferença na forma de gráficos coloridos.

4.4 Exemplo prático

Se quiséssemos implementar a meta-heurística *Simulated Annealing* no sistema, bastaria que criássemos uma classe na seguinte forma:

[1] Decorator – Design Patterns (GOF)
[2] Strategy – Design Patterns (GOF)

```
/**
 * Meta-heuristica: que simula processos industriais de metalurgia
 *          usados para rearranjar particulas
 */
public class SimulatedAnnealing implements Escalonador {

  public Escalonamento escalona(List<Programa> periodo) {
    //
    // faz o algoritmo aqui...
    //
    return new Escalonamento(escalonado);
  }
}
```

E um ítem a mais no menu de heurísticas:

```
  private Escalonador escolheEscalonador() {

    // outros escalonadores...

    if(simulatedAnnealing.isSelected()) {
      return new SimulatedAnnealing();
    }
    return new FirstFit();
  }
```

Rodando a aplicação, o novo ítem apareceria no menu e, ao selecioná-lo e escolher o arquivo de entrada, o escalonamento utilizando *Simulated Annealing* apareceria na tela em poucos momentos.

5 Mais restrições e heurísticas

Reservas e outras restrições ou heurísticas que podiam ser implementadas

6 O futuro

Se continuarmos o projeto, ou se alguém mais continuar o projeto, enxergamos alguns pontos de potencial melhoria que podem ser considerados:

6.1 Granularidade do tempo

Na sequência, o planejamento para esse projeto envolve melhorar a granularidade dos tempos de duração de cada tarefa. No momento, a unidade de tempo vale meia hora – o que ainda não é realista para duração de comerciais.

Esse aumento na granularidade pode causar um crescimento inaceitável no tempo de execução das heurísticas mais ingênuas e, dessa forma, é esperado que novas necessidades apareçam – e que, portanto, novas heurísticas ou meta-heurísticas iterativas sejam implementadas.

6.2 Formato da entrada

Outra mellhoria potencial seria mudar o formato da entrada para que ele contenha também informações sobre o programa, como seu nome de exibição e idade recomendada.

Mudando a entrada, será necessário melhorar as legendas do gráfico para que, com o *mouse* sobre um programa, msotre-se essas novas informações, em vez no número do programa e outros caracteres, como o atual faz.

Talvez, também, dispender um tempo alterando a legenda do eixo horizontal para que ela represente, no formato em que estamos acostumados a ver, o horário em que cada programa começará e terminará.

6.3 Novos escalonadores

Implementar os escalonadores propostos na seção anterior, Outras Restrições, também seria uma tarefa interessante e bastante mais ligada à área de escalonamento.

Talvez até implementar alguma meta-heurística, simples a princípio, como a Busca Tabu, para compará-la com heurísticas puras e simples e decidir qual nos traria um mlehor resultado.

7 Conclusão

Ao implementar várias heurísticas, conseguimos ver que heurísticas diferentes levavam a escalonamentos diferentes, para uma mesma entrada. Apesar disso, duas heurísticas estão gerando o mesmo resultado, para várias das entradas testadas.

Essas heurísticas problemáticas são a *First Fit* e a *Next Fit*. Elas são bastante parecidas, mas uma foca em tentar preencher um bin por vez, colocando o primeiro item que cabe, e a outra em colocar um item por vez, no primeiro bin que cabe. Uma explicação possível é que, pela característica do problema e dos limites para o tamanho do item, no máximo dois bins ficam abertos, o que faz com que essas duas heurísticas ficam equivalentes. Uma outra explicação é que a nossa implementação esteja incorreta, mas não conseguimos encontrar algo que indique isso – o algoritmo parece estar de acordo com a especificação da heurística.

Ao analisar os resultados das heurísticas, a *Best Fit* pareceu apresentar melhores resultados. Melhor no sentido que minimizou o tempo gasto com comerciais, levando em conta que pelo menos um tempo de comercial deveria acontecer entre dois programas. Uma explicação razoável para esse fato é que como esse algoritmo tenta deixar o menor espaço vago possível no período em que ele coloca um programa, o período tende a ter menos programas que quando utilizamos outras heurísticas. Como o fato do comercial ser obrigatório entre dois programas, é bom que tenha menos programas em um período, então o *Best Fit* parece ser a melhor opção de heurística para o problema.