Implementação da heurística $LR(x)$ para o problema de escalonamento $F \mid \circ \mid C_{sum}$ utilizando *Data-Oriented Design*

Emerson Ferreira Silva

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2019

Emerson Ferreira Silva

Implementação da heurística $LR(x)$ para o problema de escalonamento $F \mid \circ \mid C_{sum}$ utilizando *Data-Oriented Design*

Monografia apresentada ao curso Ciência da Computação
do Centro de Informática, da Universidade Federal da Paraíba,
como requisito para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Lucídio dos Anjos Formiga Cabral

Outubro de 2019

Catalogação na publicação
Seção de Catalogação e Classificação

S586i Silva, Emerson Ferreira.
Implementação da heurística LR(x) para o problema de escalonamento F | | C_sum utilizando Data-Oriented Design / Emerson Ferreira Silva. - João Pessoa, 2019.
40 f. : il.

Orientação: Lucídio dos Anjos Formiga Cabral.
Monografia (Graduação) - UFPB/CI.

1. Escalonamento. 2. Heurística LR(x). 3. Data-Oriented Design. I. Cabral, Lucídio dos Anjos Formiga. II. Título.

UFPB/CI

CENTRO DE INFORMÁTICA
UNIVERSIDADE FEDERAL DA PARAÍBA

Trabalho de Conclusão de Curso de Ciência da Computação intitulado ***Implementação da heurística*** $LR(x)$ ***para o problema de escalonamento*** $F \mid \circ \mid C_{sum}$ ***utilizando Data-Oriented Design*** de autoria de Emerson Ferreira Silva, aprovada pela banca examinadora constituída pelos seguintes professores:

Prof. Dr. Lucídio dos Anjos Formiga Cabral
Universidade Federal da Paraíba — UFPB

Prof. Dr. Gilberto Farias de Sousa Filho
Universidade Federal da Paraíba — UFPB

Prof. Dr. Fernando Menezes Matos
Universidade Federal da Paraíba — UFPB

Prof. Dr. Gustavo Henrique Matos Bezerra Motta
Coordenador do curso de Ciência da Computação
CI/UFPB

João Pessoa, 3 de outubro de 2019

Centro de Informática, Universidade Federal da Paraíba
Rua dos Escoteiros, Mangabeira VII, João Pessoa, Paraíba, Brasil CEP: 58058-600
Fone: +55 (83) 3216 7093 / Fax: +55 (83) 3216 7117

RESUMO

Design orientado a dados é uma nova metodologia de software, que foca na compreensão e exploração dos dados que acompanham um problema a ser resolvido. A área que mais se beneficia atualmente dessa metodologia é a indústria de jogos; porém, os conceitos trazidos podem beneficiar outros setores de software. Este trabalho propôs a implementação da heurística $LR(x)$ utilizando o *design* orientado a dados, objetivando tanto uma implementação eficiente do algoritmo quanto o estudo dos benefícios dessa metodologia. De fato, a análise profunda do algoritmo e dos dados disponibilizados revelaram otimizações que não seriam possíveis com o uso de outras metodologias, como o paradigma orientado a objetos. Os resultados mostram um aumento de 34 vezes no desempenho para o menor conjunto de testes.

Palavras-chave: Escalonamento. Heurística $LR(x)$. *Design* orientado a dados.

ABSTRACT

Data-oriented design is a new software methodology that focuses on understanding and exploring the data that accompanies a problem to be solved. The area that currently benefits most from this methodology is the games industry; however, the concepts brought may benefit other software sectors. This work proposed the implementation of the $LR(x)$ heuristic using data-oriented design, aiming at both an efficient implementation of the algorithm and the study of the benefits of this methodology. Indeed, in-depth analysis of the algorithm and available data revealed optimizations that would not be possible using other methodologies, such as the object-oriented paradigm. Results show a performance increase by a factor of 34 for the smallest set of tests.

Keywords: Scheduling. $LR(x)$ heuristic. Data-oriented design.

SUMÁRIO

1 INTRODUÇÃO

O aumento da velocidade do hardware parece não ter sido acompanhado da mesma forma pelo software. Camadas e mais camadas de abstração tornam o software mais pesado; programas que aparentam ser simples utilizam a CPU no seu limite máximo. Um dos responsáveis por esse problema são as metodologias de desenvolvimento software, principalmente o paradigma de orientação a objetos: em nome de uma suposta "manutenibilidade" e "elegância", ignoram detalhes sobre o problema, sobre os dados que acompanham o problema, sobre o hardware que executa o programa que resolve o problema. O resultado disso são milhares e milhares de *watts* desperdiçados.

Uma resposta à essa observação sobre o *status quo* apareceu há pouco mais de dez anos, primeiramente na indústria de jogos e, posteriormente, sendo espalhada para outros setores de software: o *design* orientado a dados. Nele, o desenvolvedor busca entender os dados que vêm com o problema que ele está tentando resolver; procura padrões, formas eficientes de guardar e processar os dados.

Nesse trabalho, pretendemos testar a eficácia do *design* orientado a dados na implementação de uma heurística para resolução de um problema cuja demanda computacional é considerável: o problema de escalonamento.

1.1 Definição do problema

Problemas de escalonamento são extremamente difíceis de resolver; de fato, eles encontram-se na classe $\mathcal{NP}$-difícil. Heurísticas são elaboradas para gerar soluções suficientemente boas; porém, a codificação sem o devido cuidado pode gerar aplicações ineficientes.

1.2 Premissas e hipóteses

Partimos da premissa de que uma estratégia baseada no *design* orientado a dados pode levar a otimizações a nível de projeto que podem melhorar o desempenho da implementação de uma heurística.

1.3 Objetivo principal

Experimentar o *design* orientado a dados e mostrar que os resultados obtidos com ele são superiores a outras metodologias de desenvolvimento.

1.4 Objetivos específicos

Desenvolver um programa que implemente a heurística para o problema de escalonamento, aplicando as otimizações baseadas no *design* orientado a dados.

1.5 Estrutura da monografia

No capítulo 2, apresentamos um apanhado de conceitos sobre o problema de escalonamento e a metodologia de *design* orientado a dados. No capítulo 3, aplicamos o processo de *design* orientado a dados para extrair possíveis otimizações. Por fim, no capítulo 4, mostramos os resultados da aplicação das otimizações obtidas, concluindo no 5 com observações gerais sobre o trabalho e perspectivas para trabalhos futuros.

2 REFERENCIAL TEÓRICO

Nesse capítulo, serão apresentados os conceitos gerais sobre o problema de escalonamento, tipos específicos desse problema, heúristicas para resolvê-lo e metodologias para implementar essas heurísticas.

2.1 Problema de escalonamento

MacCarthy e Liu (1993, p. 60, tradução nossa) definem o problema de escalonamento da seguinte forma:

> Existem m máquinas $M = \{M_1, \ldots, M_m\}$. n tarefas $J = \{J_1, \ldots, J_n\}$ devem ser processadas. Um subconjunto das máquinas é necessário para completar o processamento de cada tarefa. Um [...] fluxo de processamento [...] pode ou não ser estabelecido para algumas ou todas as tarefas. O ato da máquina M_i processar a tarefa J_j é chamado de operação, e é denotado por O_{ij}. Para cada O_{ij}, existe um tempo de processamento p_{ij}. Além disso, cada tarefa pode ter um prazo de liberação r_j, a partir da qual J_j pode ser processada, e/ou um prazo máximo de processamento d_j. Nesse contexto, o escalonamento é a alocação de tarefas para máquinas ao longo do tempo. O problema de escalonamento consiste em encontrar um escalonamento que otimiza alguma medida de desempenho.

O problema de escalonamento é um dos mais conhecidos e estudados na literatura de otimização combinatória (MACCARTHY; LIU, 1993); o começo do interesse na área data da década de 1950, com o artigo pioneiro de Johnson (1954) sobre escalonamentos ótimos em duas e três máquinas. O interesse em escalonamentos eficientes surgiu primeiramente no setor industrial, pois, para uma companhia:

a) utilizar os recursos existentes de forma eficiente;

b) responder rapidamente a demandas; e

c) obedecer rigorosamente a deadlines

significaria permanecer competitiva frente à concorrência (BAKER, 1974 apud MACCARTHY; LIU, 1993, p. 61–62). Por conta dessa origem, a comunidade científica convencionou utilizar os termos “máquina” e “tarefa” para designar as entidades envolvidas no problema. Porém, esses termos não implicam que as soluções propostas apenas resolvam problemas de nível industrial envolvendo escalonamento; pelo contrário, eles podem assumir diferentes significados, dependendo do contexto.

Tome, como exemplo, o TSP (do inglês *Travelling Salesman Problem*, ou Problema do Caixeiro Viajante). Nesse problema, há uma lista de cidades pelas quais o caixeiro

deve passar; há também um mapa que mostra as distâncias entre as cidades. Então, partindo de uma delas, o caixeiro deve passar por todas elas e voltar à cidade de onde partiu *através da menor rota possível* (KORTE; VYGEN, 2012, p. 405). Podemos reduzir o TSP a um problema de escalonamento se assumirmos que o caixeiro é uma tarefa, as cidades pelas quais ele deve passar são máquinas e a medida de desempenho é o tamanho da rota. Ou seja, o TSP é um caso especial do problema de escalonamento, com m máquinas e 1 tarefa.

Essa redução diz-nos uma característica importante do problema de escalonamento: o TSP é um problema $\mathcal{NP}$-difícil (KORTE; VYGEN, 2012, p. 405); dado que ele é polinomialmente reduzível a um problema de escalonamento, então esse também é $\mathcal{NP}$-difícil. Isso significa que, exceto em casos especiais como o escalonamento utilizando duas máquinas que pode ser resolvido utilizando a regra de Johnson (JOHNSON, 1954, p. 16–17), encontrar soluções ótimas para esse problema é um processo computacionalmente dispendioso.

Um problema de escalonamento é caracterizado por detalhes como: características das máquinas; características das tarefas; padrões dos fluxos de processamento; medidas de desempenho; e ambiente de escalonamento (ROJAS, 2015, p. 28). Essas características podem ser descritas na notação $\alpha|\beta|\gamma$, proposta por Graham et al. (1979, p. 288); nessa notação, α representa o ambiente das máquinas, β representa as restrições das tarefas, e γ determina a medida de desempenho.

Dentre esses problemas, temos o problema de escalonamento do tipo *flow shop* (do inglês, *Flow Shop Scheduling Problem*, ou FSSP). Esse, mais especificamente a variação *flow shop* de permutação com o tempo total de conclusão C_{sum} como medida de desempenho, denotada por $F||C_{sum}$, será o nosso problema alvo. A seção 2.2 apresenta sua formulação.

2.2 Escalonamento do tipo *flow shop* de permutação

Liu e Reeves (2001, p. 440, tradução nossa) fornecem a seguinte definição para o problema de escalonamento do tipo *flow shop* de permutação (do inglês, *Permutation Flow Shop Scheduling Problem*, ou PFSSP):

> n tarefas $J = \{J_1, \ldots, J_n\}$ devem ser sequencialmente processadas em uma série de m máquinas $M = \{M_1, \ldots, M_m\}$. O tempo de processamento de J_j em M_i é dado por p_{ij}. Cada máquina pode processar no máximo uma tarefa por vez, e cada tarefa pode ser processada em no máximo uma máquina por vez. Não é permitida a preempção. Para uma dada sequência [...] de tarefas $\pi = (\pi_1, \pi_2, \ldots, \pi_n)$, o tempo de conclusão das tarefas nas máquinas é calculado de forma recursiva:

$$C(i, \pi_j) = \begin{cases} \max\left\{C(i-1, \pi_j), C(i, \pi_{j-1})\right\} + p_{i,\pi_j}, & i \in [m], j \in [n] \\ 0, & i = 0 \\ 0, & j = 0 \end{cases}. \tag{2.1}$$

> O tempo de [...] conclusão de uma tarefa é definido como sendo o seu tempo de conclusão na última máquina. O problema é achar uma sequência que minimiza a soma dos tempos de conclusão de todas as n tarefas,
>
> $$C_{sum} = \sum_{j=1}^{n} C(m, j). \tag{2.2}$$

A restrição adicionada pelo PFSSP — de que o fluxo de processamento das tarefas é prefixado (MACCARTHY; LIU, 1993, p. 61) — reduz o espaço de busca de $(n!)^m$ para $n!$. No entanto, mesmo com essa redução, encontrar escalonamentos ótimos para esse problema é, como dito na seção 2.1, absurdamente difícil. Uma saída é procurar por escalonamentos *suficientemente bons*. Essas soluções podem ser encontradas através da utilização de *heurísticas*, critérios simples para encontrar boas soluções em um tempo razoável (PEARL, 1984, p. 3). O quão "suficientemente boa" é uma solução é definida pelo contexto do problema a ser resolvido; esse contexto influencia a formulação da heurística, para torná-la mais rígida ou mais relaxada quanto à qualidade da solução (BENTLEY, 1982, p. 11).

Para esse problema, utilizaremos a heurística $LR(x)$, descrita na seção 2.3.

2.3 Heurística $LR(x)$

$LR(x)$ é uma heurística apresentada por Liu e Reeves (2001) para solucionar o problema $F||C_{sum}$. Ela é uma heurística construtiva, isto é, começa com uma sequência de escalonamento vazia e, enquanto não a preencher completamente, escolhe e anexa, dentre as tarefas não escalonadas, a que é considerada a melhor da rodada segundo um dado critério. O fato de fazer escolhas localmente ótimas na esperança de chegar a uma solução globalmente ótima define $LR(x)$ como sendo um algoritmo guloso (CORMEN et al., 2009, p. 414).

O critério de escolha é definido pela função de índice $\xi(j, k)$; ela verifica, dentro do conjunto de tarefas não escalonadas U, qual delas é a mais adequada para ser anexada à sequência de escalonamento π, sequência essa que contém k tarefas já escalonadas. Ganha a tarefa que exibir o *menor valor* para $\xi(j, k)$.

$\xi(j, k)$ é definida por

$$\xi(j, k) = (n - k - 2)IT(j, k) + AT(j, k), \tag{2.3}$$

onde $IT(j, k)$ estima o tempo ocioso induzido pelo escalonamento de J_j na posição $k + 1$ e $AT(j, k)$ estima o impacto desse escalonamento nos tempos de conclusão das tarefas em U. Destrincharemos cada um desses termos, começando por $IT(j, k)$.

Sejam

$$C(i, j) = \begin{cases} \max\{C(i-1, j), C(i, \pi_k)\} + p_{ij}, & i \in [2, m] \\ C(1, \pi_k) + p_{1,j} & i = 1 \end{cases} \tag{2.4}$$

o tempo de conclusão da tarefa J_j em M_i caso J_j fosse escalonada após π_k; e

$$\max\{C(i-1, j) - C(i, \pi_k), 0\} \tag{2.5}$$

o tempo ocioso da máquina M_i, entre o fim do processamento de π_k e o início do processamento de J_j. Então o tempo ocioso total ponderado $IT(j, k)$ entre os processamentos das tarefas π_k e J_j é dado por

$$IT(j, k) = \sum_{i=2}^{m} w(i, k) \max\{C(i-1, j) - C(i, \pi_k), 0\}, \tag{2.6}$$

onde

$$w(i, k) = \frac{m}{i + k(m-i)/(n-2)}. \tag{2.7}$$

O tempo ocioso das máquinas iniciais, comparado ao das máquinas finais, podem resultar em maiores atrasos nos tempos de conclusão das tarefas em U. $w(i, k)$, portanto, apresenta penalidades maiores para as primeiras máquinas. Conforme as tarefas são escalonadas, $w(i, k)$ diminui essas penalidades.

Agora é a vez de $AT(j, k)$. Considere uma tarefa artificial J_a. Sejam

$$p_{i,a} = \sum_{\substack{q \in U \\ q \neq j}} p_{i,q}/(n-k-1) \tag{2.8}$$

o tempo de processamento de J_a, dado pela média dos tempos de processamento de todas as tarefas em U, exceto por J_j; e

$$C_a(i) = \begin{cases} \max\{C_a(i-1), C(i, j)\} + p_{i,a}, & i \in [2, m] \\ C(1, j) + p_{1,a} & i = 1 \end{cases} \tag{2.9}$$

o tempo de conclusão de J_a, caso fosse escalonada após J_j. Então o tempo total de conclusão artificial é dado por

$$AT(j, k) = C(j, m) + C_a(m). \tag{2.10}$$

Ao somar esses dois termos, $\xi(j, k)$ dá maior peso ao resultado de $IT(j, k)$ nos estágios iniciais, pois, por haver muitas tarefas não escalonadas, $AT(j, k)$ não exibe resultados acurados. Conforme as tarefas são escalonadas, a acurácia de $AT(j, k)$ aumenta; consequentemente, o peso de $IT(j, k)$ diminui.

A figura 1 mostra o algoritmo da heurística $LR(x)$. Sua complexidade é $\mathcal{O}(kn^3m)$.

Até a criação da heurística $FF(x)$ (FERNANDEZ-VIAGAS; FRAMINAN, 2015), essa heurística era considerada o estado da arte na resolução do problema $F||C_{sum}$.

Algoritmo 1 Heurística $LR(x)$

```
function LR(x)
    π_o ← (π_o1, π_o2, ... π_on) ordenados de forma ascendente pelo valor de ξ(j, k)
    π = ()
    for π_oj, j ∈ [x] do
        π' = (π_oj)
        U = J − {π_oj}
        while |U| > 0 do
            Escolha a tarefa J_j' em U para o qual ξ(j, k) retorna o menor valor
            Remova J_j' de U e anexe a π'
        end while
        if C_sum(π') < C_sum(π) then
            π ← π'
        end if
    end for
    return π
end function
```

2.4 *Design* orientado a dados

Segundo Acton (2008), a cultura de desenvolvimento é atualmente centrada nos três seguintes princípios:

a) o software é a plataforma;

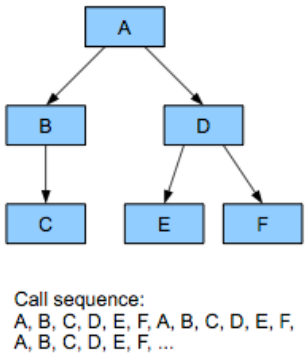
b) o código de um programa deve ser projetado como um modelo do mundo; e

c) o código é a parte mais importante de um programa.

O primeiro princípio refere-se ao ato de utilizar um produto de software como ambiente de execução para outro produto de software; isso é o resultado de ciclos e mais ciclos de abstração e generalização ao longo dos anos, ciclos esses guiados pelo propósito de evitar ao máximo lidar diretamente com os detalhes ligados ao hardware e/ou ao sistema operacional.

O segundo princípio está intimamente ligado à predominância do paradigma de orientação a objetos. Nesse paradigma, o desenvolvedor lida com *objetos*, entidades que agregam dados, na forma de *campos*, e código, na forma de *procedimentos*. Em conjunto, os campos designam o *estado interno* do objeto. Os procedimentos ligados a um objeto podem *acessar* e *modificar* seu estado interno. Um objeto *interage* com outro ao chamar um de seus procedimentos. Programas construídos com o paradigma de orientação a objetos são, portanto, compostos de objetos que interagem uns com os outros (WILL, 2019). O esquema de interações entre objetos é compreensível à mente humana, pois vários processos existentes ocorrem dessa forma; portanto, é extremamente apelativo estruturar programas segundo essa forma.

Figura 1 – Sequência de chamadas no paradigma orientado a objetos



Fonte: Llopis (2009)

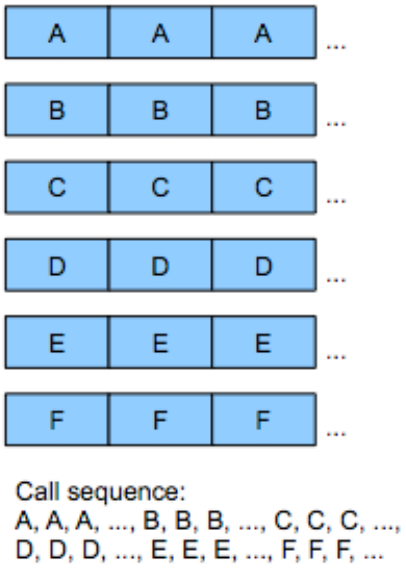
No terceiro princípio, profissionais da área dão ao código um status elevado dentro de um produto de software; isso revela-se na exploração de áreas como a escolha do paradigma a ser utilizado na construção de um programa, a criação de guias de boas práticas (KERNIGHAN; PLAUGER, 1978), diagramas de modelagem (JACOBSON; BOOCH; RUMBAUGH, 1999) e padrões de projeto (GAMMA et al., 1994). Muitas vezes, essa importância é elevada em detrimento de outros pontos, como uma pesquisa profunda sobre o domínio do problema a ser resolvido.

Acton (2008) chama esses princípios de "três grandes mentiras". O primeiro princípio é considerado uma mentira porque *o hardware é a plataforma*. Não se pode idealizar o hardware e desprezar os efeitos causados pelo seu (mau) uso na execução do software; o software não é executado no "vácuo". Quando tiramos o hardware da equação, não conseguimos raciocinar sobre o custo de resolver um problema (ACTON, 2008; MEYERS, 2014).

O segundo princípio é considerado uma mentira porque gera dois problema: o primeiro deles, e o mais óbvio, é que *o hardware não trabalha em termos de um modelo do mundo*. Quando pensamos em objetos, pensamos em árvores de objetos; naturalmente, arranjamos os dados na forma de árvores. Aplicar uma operação em um objeto gera uma cascata de operações diferentes nos objetos relacionados; essa situação piora se a operação é aplicada em um conjunto de objetos, como mostra a figura 1. No entanto, o hardware lida melhor com estruturas lineares e homogêneas que são processadas sequencialmente (LLOPIS, 2009; CARRUTH, 2014; MEYERS, 2014). Essa forma de estruturar os dados leva em consideração que, "onde há um, provavelmente há mais de um" (ACTON, 2008).

O segundo problema afeta tanto o hardware quanto o desenvolvedor. Dados não carregam significado, mas podem ser dotados de significado para criar informação; porém, enquanto, em alguns contextos, atribuir um significado resulta em uma melhor

Figura 2 – Sequência de chamadas no *design* orientado a dados



Fonte: Llopis (2009)

compreensão dos dados, em outros, ela bloqueia o ato de pensar nos dados na sua forma mais pura, o que pode impedir *insights* importantes para criar soluções para os problemas que os desenvolvedores estão tentando resolver (FABIAN, 2018). Ao dar significado aos dados, agregá-los em objetos e atrelar comportamentos a esses objetos, o programa torna-se um sistema rígido, praticamente impenetrável a questionamentos sobre os dados, mascarando oportunidades para otimização (ACTON, 2014).

Por fim, o terceiro princípio é considerado uma mentira porque *o verdadeiro valor está nos dados que acompanham o problema*. Segundo Llopis (2009, tradução nossa), "programação, por definição, é sobre transformar dados: é o ato de criar uma sequência de instruções de máquina descrevendo como processar dados de entrada e criar dados de saída". Sendo assim, o propósito de todo programa e de todas as partes do programa, é resolver problemas através de transformações de dados; o *único* propósito do código é expressar essas transformações (ACTON, 2014; WIRTH, 1976). Portanto, o foco do desenvolvedor de software deve ser resolver problemas, não escrever código, pois esse não possui valor intrínseco nenhum.

As refutações aos três princípios têm um ponto em comum, que é o princípio norteador do chamado *design orientado a dados*: os dados que rodeiam o problema são a sua parte mais importante (FABIAN, 2018). Segundo Llopis (2009, tradução nossa):

> [...] Note que o foco principal [do paradigma de orientação a objetos] é o código: [...] [ele preocupa-se com comportamentos] associados a algum estado interno [...] . *Design* orientado a dados muda o foco [...] dos objetos para os dados em si: o tipo dos dados, como eles são dispostos na memória, e como eles serão lidos e processados [...] .

3 METODOLOGIA

Nesse capítulo, os dados disponibilizados pelo problema serão analisados e otimizações serão propostas para implementação. A avaliação da eficiência e eficácia dessas otimizações será feita com base no seu impacto no hardware — tempo de CPU e consumo de memória.

3.1 Unindo a sequência π e o conjunto U

Na seção 2.3, a heurística $LR(x)$ é apresentada. Duas entidades são centrais nesse algoritmo: a sequência de escalonamento π e o conjunto de tarefas não escalonadas U. Uma invariante do algoritmo é $|\pi| + |U| = n$. Essa invariante é mantida pelo fato de que, a cada iteração, uma tarefa é retirada de U e colocada como o último elemento de π.

Com essa invariante, podemos colocar π e U em uma única sequência P, cujo tamanho é n. Se o número de tarefas já escalonadas é dado por k, $\pi = (P_j)_{1 \leq j \leq k}$ e $U = (P_j)_{k < j \leq n}$. Consequentemente, $|\pi| = k$ e $|U| = n - k$.

Nós sabemos que uma tarefa possui o menor valor de $\xi_{j,k}$ dentro do conjunto U porque comparamos esse valor com os de todas as outras tarefas. A palavra-chave aqui é "comparação": se analisarmos bem, o algoritmo tem uma estrutura semelhante ao *algoritmo de ordenação por seleção* (do inglês, *selection sort*): a cada iteração, a tarefa que possui o menor valor para $\xi_{j,k}$ é escolhida; essa tarefa troca de lugar com a primeira tarefa da sequência de tarefas não escalonadas, P_{k+1}, e k é incrementado.

O algoritmo resultante está disposto em 2.

Vamos analisar qual o impacto dessa manipulação na implementação em código. Se π é implementada como um *array* dinâmico, há o risco de, durante o escalonamento de uma tarefa, sua capacidade ser estourada, provocando uma expansão em memória e uma cópia das tarefas escalonadas para o novo local (CORMEN et al., 2009, p. 464). Além disso, se houve alguma expansão em algum momento do seu ciclo de vida, há a possibilidade da capacidade de π ser maior que n, o que significa que uma porção da memória está sendo inutilizada. Como o tamanho de π é conhecido, assumimos a partir de agora que π é alocado de antemão.

U é um conjunto; portanto, a ordem dos seus elementos é irrelevante. U pode ser implementado como um conjunto propriamente dito ou como um *array*. Suponhamos que U é implementado como um *array*; ao escalonar tarefas, podemos removê-las de U ou marcá-las com um valor sentinela. Ambos os casos são ineficientes, pois remover elementos de um *array* implica em mover a última tarefa em U para a lacuna deixada

Algoritmo 2 Heurística $LR(x)$, com estrutura similar ao *selection sort*

```
function LR(x)
    π_o ← (π_o1, π_o2, ... π_on) ordenados de forma ascendente pelo valor de ξ(j, k)
    π = ()
    for π_oj, j ∈ [x] do
        P = (P_1, P_2, ..., P_n)
        Encontre onde π_oj está em P e troque-o de lugar com P_1
        for k ← 2, n − 1 do
            U = (P_j)_{k<j≤n}
            Escolha a tarefa J_j' em U para o qual ξ(j, k) retorna o menor valor
            Troque J_j' de lugar com U_1
        end for
        if C_sum(P) < C_sum(π) then
            π ← π'
        end if
    end for
    return π
end function
```

pela tarefa removida; combinado com o movimento da tarefa escalonada para P, temos duas escritas na memória.

Já marcar tarefas com um valor setinela cria a necessidade de, durante uma iteração em U, verificar se a tarefa da iteração atual já foi escalonada; essa verificação, implementada pela estrutura condicional `if`, cria um problema chamado *branch misprediction*: em arquiteturas com *pipeline* de instruções, um *branch predictor* tenta "adivinhar" qual será a próxima instrução a ser executada; se essa "adivinhação" estiver errada, todas as instruções executadas de forma especulativa são *descartadas*, o processador espera o *pipeline* esvaziar e, só então, as instruções corretas são executada (KAPOOR, 2009). Isso resulta em perda de desempenho. Suponhamos, então, que U é implementado como um conjunto de fato.

Juntos, π e U têm tamanho $2n$; porém, apenas n tarefas existem para ser escaladas. Isso significa que há memória sendo inutilizada. Embora, conceitualmente, π cresça e U diminua de tamanho, mantendo a invariante $|\pi| + |U| = n$, essas repectivas expansão e contração são operações custosas em hardware. Na versão baseada no *selection sort*, como π e U são implementados em um mesmo *array*, é possível simular, de forma eficiente, suas respectivas expansão e contração. Portanto, essa versão resulta numa implementação mais eficiente.

3.2 Transformando $C(i, \pi_j)$ em um algoritmo iterativo

$C(i, \pi_j)$ é definida na equação 2.1 como sendo o tempo de conclusão da tarefa π_j na máquina M_i. Mostraremos que o custo de computar $C(i, \pi_j)$ é exponencial.

Tabela 1 – Resultados de $T(i, j)$ para $i \in [1, 10]$, $j \in [1, 10]$

1	1	1	1	1	1	1	1	1	1
1	4	7	10	13	16	19	22	25	28
1	7	16	28	43	61	82	106	133	163
1	10	28	58	103	166	250	358	493	658
1	13	43	103	208	376	628	988	1483	2143
1	16	61	166	376	754	1384	2374	3859	6004
1	19	82	250	628	1384	2770	5146	9007	15013
1	22	106	358	988	2374	5146	10294	19303	34318
1	25	133	493	1483	3859	9007	19303	38608	72928
1	28	163	658	2143	6004	15013	34318	72928	145858

Fonte: o autor

A quantidade de operações realizadas por $C(i, \pi_j)$ é dada pela seguinte relação de recorrência:

$$T(i, j) = \begin{cases} T(i-1, j) + T(i, j-1) + 2, \, i \in [m], \, j \in [n] \\ 1, \, i = 0 \\ 1, \, j = 0 \end{cases} \tag{3.1}$$

A tabela 1 apresenta os resultados de $T(i, j)$, para $i \in [1, 10]$, $j \in [1, 10]$. Os elementos plotados podem ser facilmente reconhecidos como coeficientes binomiais. De fato, a matriz gerada por $T(i, j)$ tem relação com o triângulo de Pascal. A diagonal $T(i, i)$ sugere que (ADAMSON, 2007):

$$T(i, j) = 3\binom{i+j}{i} - 2. \tag{3.2}$$

Provaremos essa afirmação por indução.

Problema 3.1. Prove que, para todo $i \geq 0$, $j \geq 0$, $T(i, j) = 3\binom{i+j}{i} - 2$.

Solução. Para qualquer par (i, j), $i \geq 0$, $j \geq 0$, suponhamos que $S(i, j)$ denota a afirmação

$$S(i, j) : T(i, j) = 3\binom{i+j}{i} - 2.$$

Caso base: $i = 0$, $j = 0$. $S(0, 0)$ é verdadeira, pois:

$$\begin{aligned} 3\binom{0+0}{0} - 2 &= 3\frac{(0+0)!}{0!0!} - 2 \\ &= 3 - 2 \\ &= 1. \end{aligned}$$

Passo indutivo: $S(k, 0) \longrightarrow S(k+1, 0)$. Seja $k \geq 0$. Assumindo que $S(k, 0)$ é verdadeira, provemos que $S(k+1, 0)$ também é verdadeira. Começando pelo lado esquerdo

de $S(k+1,\,0)$:

$$\begin{aligned}
T(k+1,\,0) &= 1 && \text{(pela definição de } T(i,\,j)\text{)}\\
&= 3\frac{k!}{k!} - 2 && \text{(pela hipótese indutiva)}\\
&= 3\frac{(k+1)k!}{(k+1)k!} - 2\\
&= 3\frac{(k+1)!}{(k+1)!} - 2\\
&= 3\binom{k+1}{k+1} - 2,
\end{aligned}$$

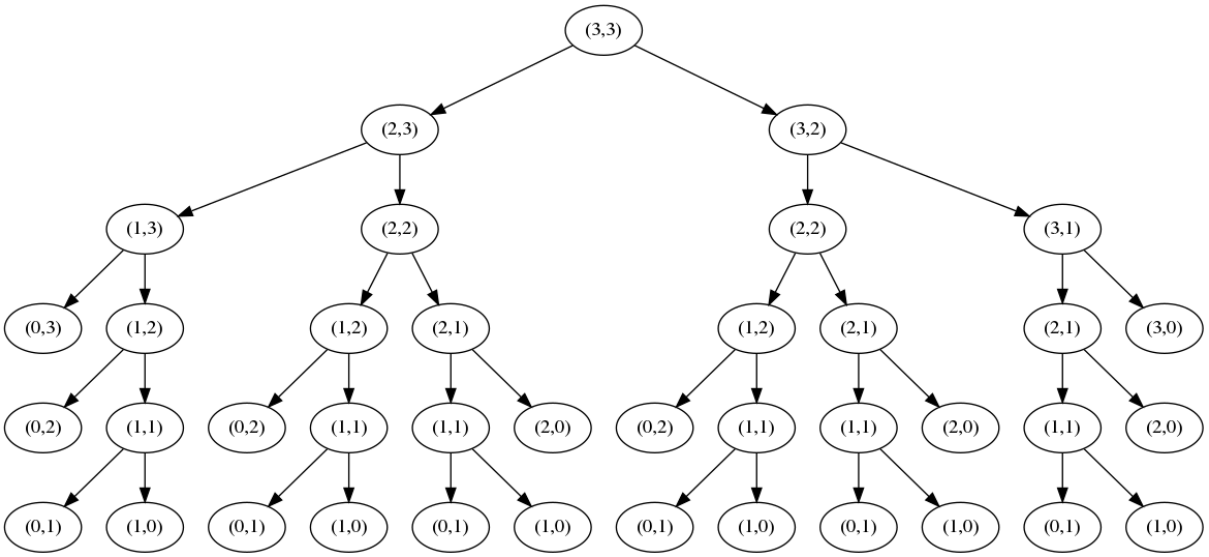
chegamos ao lado direito de $S(k+1,\,0)$, portanto mostrando que $S(k+1,\,0)$ também é verdadeira, completando o passo indutivo.

Passo indutivo: $S(h,\,k) \rightarrow S(h,\,k+1)$. Seja $k \geq 0$ e $h \geq 0$ uma constante. Assumindo que $S(h,\,k)$ é verdadeira, provemos que $S(h,\,k+1)$ também é verdadeira. Começando pelo lado esquedo de $S(h,\,k+1)$:

$$\begin{aligned}
T(h,\,k+1) &= T(h-1,\,k+1) + T(h,\,k) + 2 && \text{(pela definição de } T(i,\,j)\text{)}\\
&= 3\frac{(h-1+k+1)!}{(h-1)!(k+1)!} - 2 + 3\frac{(h+k)!}{h!k!} - 2 + 2 && \text{(pela hipótese indutiva)}\\
&= 3\left[\frac{(h+k)!}{h!k!} + \frac{(h+k)!}{(h-1)!(k+1)!}\right] - 2\\
&= 3\left[\frac{h!k!(h+k)! + (h-1)!(k+1)!(h+k)!}{h!k!(h-1)!(k+1)!}\right] - 2\\
&= 3\left[(h+k)!\frac{h!k! + (h-1)!(k+1)!}{h!k!(h-1)!(k+1)!}\right] - 2\\
&= 3\left[(h+k)!\frac{h!k! + (h-1)!k!(k+1)}{h!k!(h-1)!(k+1)!}\right] - 2\\
&= 3\left[(h+k)!\frac{h! + (h-1)!(k+1)}{h!(h-1)!(k+1)!}\right] - 2\\
&= 3\left[(h+k)!\frac{(h-1)!h + (h-1)!(k+1)}{h!(h-1)!(k+1)!}\right] - 2\\
&= 3\left[(h+k)!\frac{h+k+1}{h!(k+1)!}\right] - 2\\
&= 3\frac{(h+k+1)!}{h!(k+1)!} - 2\\
&= 3\binom{h+k+1}{h} - 2,
\end{aligned}$$

chegamos ao lado direito de $S(h,\,k+1)$, portanto mostrando que $S(h,\,k+1)$ também é verdadeira, completando o passo indutivo.

Figura 3 – Árvore de recursão para $C(3, \pi_3)$



Fonte: o autor

Conclusão: Está, então, provado por indução que, para todo $i \geq 0, j \geq 0$, a afirmação $S(i, j)$ é verdadeira. ■

Sabendo que $T(i, j) = 3\binom{i+j}{i} - 2$, vejamos qual o comportamento de $T(i, j)$. Assumindo, por simplicidade, que $j = i$, isto é, olhando apenas para o comportamento na diagonal, utilizamos a aproximação de Stirling:

$$\binom{2i}{i} \sim \frac{4^i}{\sqrt{\pi i}}. \tag{3.3}$$

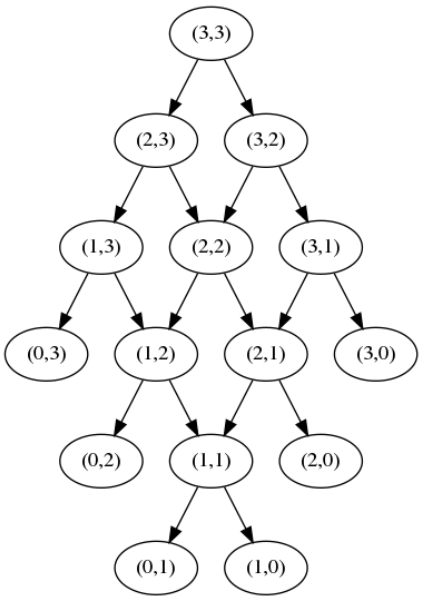
Temos, com isso, que o custo para calcular $C(i, \pi_j)$ é exponencial.

Ao analisar a árvore de recursão de $C(i, \pi_j)$, podemos ver um padrão emergir. A figura 3 mostra a árvore de recursão para $C(3, \pi_3)$. Note que alguma subárvores repetem-se; isso significa que há resultados sendo calculados mais de uma vez. De fato, essa árvore de recursão é melhor representada por um grafo, como mostra a figura 4.

Note que, pela definição de $C(i, \pi_j)$ em 2.1, ela apresenta uma *subestrutura ótima*, isto é, a solução ótima para $C(i, \pi_j)$ incorpora as soluções ótimas para $C(i - 1, \pi_j)$ e $C(i, \pi_{j-1})$. Além disso, o fato de que a "árvore" de recursão de $C(i, \pi_j)$ é um grafo indica a presença de *subproblemas sobrepostos*. Esses dois atributos indicam que existe um algoritmo de programação dinâmica que resolve $C(i, \pi_j)$ (CORMEN et al., 2009, p. 378–389). De fato, ele existe; o algoritmo 3 computa o valor de $C(i, \pi_j)$ utilizando a estratégia *bottom-up* (Para propósito de simplicidade, assuma que $c_{0,i} = 0$ e $c_{j,0} = 0$).

A figura 5 mostra a computação de $C(3, \pi_3)$, dessa vez com base em programação dinâmica. Note que cada célula da matriz consulta seu vizinho de baixo e da direita

Figura 4 – Grafo correspondente à árvore de recursão para $C(3, \pi_3)$

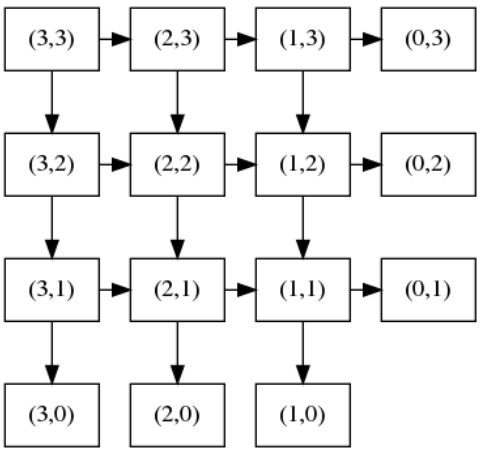


Fonte: o autor

Algoritmo 3 $C(i, \pi_j)$, modificado para utilizar programação dinâmica

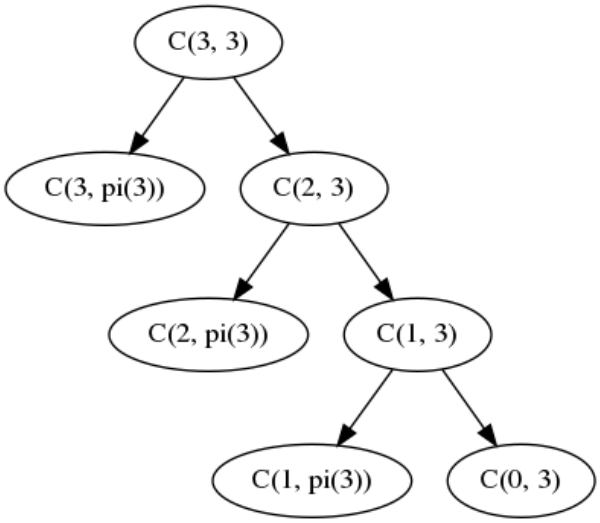
function C(i, π_j)
  $c = 0_{j \times i}$
  for $j' \leftarrow 1$, j **do**
    for $i' \leftarrow 1$, i **do**
      $c_{j',i'} = \max\left\{c_{j',i'-1},\, c_{j'-1,i'}\right\} + p_{i'j'}$
    end for
  end for
  return $c_{j,i}$
end function

Figura 5 – Padrão de acesso às celulas da matrix de memoização para $C(3, \pi_3)$



Fonte: o autor

Figura 6 – Árvore de recursão para $C(3, 3)$, com π contendo 3 tarefas escalonadas



Fonte: o autor

para saber quem é o maior deles; depois, soma o resultado com o respectivo tempo de processamento.

Esse algoritmo é da ordem de $\mathcal{O}(mn)$, pois dois laços, parametrizados, respectivamente, pelo número de tarefas e máquinas, preenche uma matriz de ordem $n \times m$.

Analisando outras funções que são construídas com base em $C(i, \pi_j)$, é fácil mostrar que a matriz de memoização pode ser reusada. Tome, como exemplo, $C(i, j)$, definida na equação 2.4. A figura 3.2 mostra que $C(i, \pi_k)$ é chamada i vezes, com i decrescendo a cada chamada. Isso significa que $C(i, \pi_k)$ está reconstruindo partes de uma matriz já existente, matriz essa computada em sua primeira chamada. Como estamos simulando que J_j é a próxima a ser escalonada, isso significa que, tendo a tabela gerada por $C(i, \pi_k)$, para calcular $C(i, j)$ basta computar os valores da linha $k + 1$ da matriz.

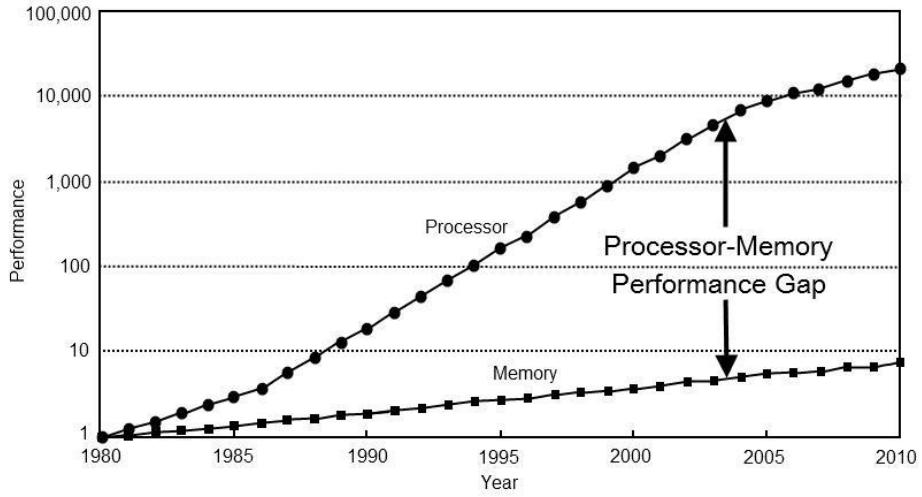
O mesmo raciocínio pode ser aplicado a $C_a(i)$, definida na equação 2.9: se J_j é alocada na posição $k + 1$ e, após isso, simulamos que a tarefa artificial J_a é a próxima a ser escalonada, basta computar os valores da linha $k + 2$ da matriz.

3.3 Dispondo os tempos de processamento p_{ij} na memória

Para entender como dispor os tempos de processamento p_{ij} na memória, precisamos falar sobre *caches* do processador.

Por conta da grande diferença de velocidade entre a memória principal e o processador, o processo de acesso a memória principal é extremamente lento. Para mascarar

Figura 7 – Diferença de desempenho entre o processador e a memória ao longo dos anos



Fonte: Hennessy e Patterson (2012)

essa lentidão, o processador utiliza uma *cache*. *Cache* é uma pequena quantidade de memória, que tem o propósito de diminuir o custo médio de acesso aos dados na memória principal. É extremamente rápida, e costuma guardar conteúdo frequentemente utilizado pelo processador (MEYERS, 2014).

Há, geralmente, três tipos de *cache*:

a) *cache* de dados: acelera a busca e armazenamento de dados;

b) *cache* de instruções: acelera a busca de instruções;

c) *translation lookaside buffer*, ou TLB: acelera a tradução de endereços de memória virtuais para reais.

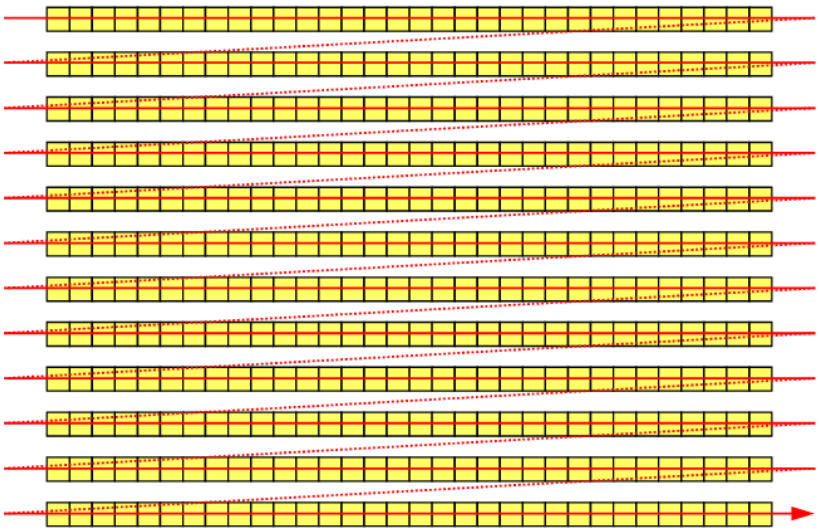
Arquiteturas modernas geralmente possuem uma *hierarquia* de caches, composta de dois ou mais níveis; cada nível é identificado por "Lx", onde "x" indica o nível do *cache*. Quando maior é o nível, maior é a quantidade de memória e menor é o desempenho.

Dados são transferidos entre *caches* e a memória principal em blocos de tamanho fixo[1], chamados de *linhas de cache* (MEYERS, 2014). Se o processador precisa recuperar ou armazenar algum conteúdo na memória, ele primeiramente verifica se o conteúdo está presente na *cache*. Se o conteúdo estiver presente, temos um *cache hit*; senão, temos um *cache miss*, e o processador precisará recorrer à memória principal. Grandes quantidades de *cache miss* resultam em baixo desempenho do programa.

Arquiteturas modernas são projetadas para detectar padrões de acesso; isso é conhecido como *pré-busca especulativa*. Se o programa itera sobre a linha n, o processador pode pré-buscar a linha $n + 1$; quando o programa chegar na linha $n + 1$, ela já estará

[1] Em processadores da Intel e da AMD, esse tamanho é de 64 bytes.

Figura 8 – Iteração em matriz, do tipo *row-major*



Fonte: Meyers (2014)

na *cache*. O mesmo vale para iterações reversas: se o programa itera sobre a linha n de forma reversa, o processador pode pré-buscar a linha $n - 1$; quando o programa chegar na linha $n - 1$, ela já estará na *cache*.

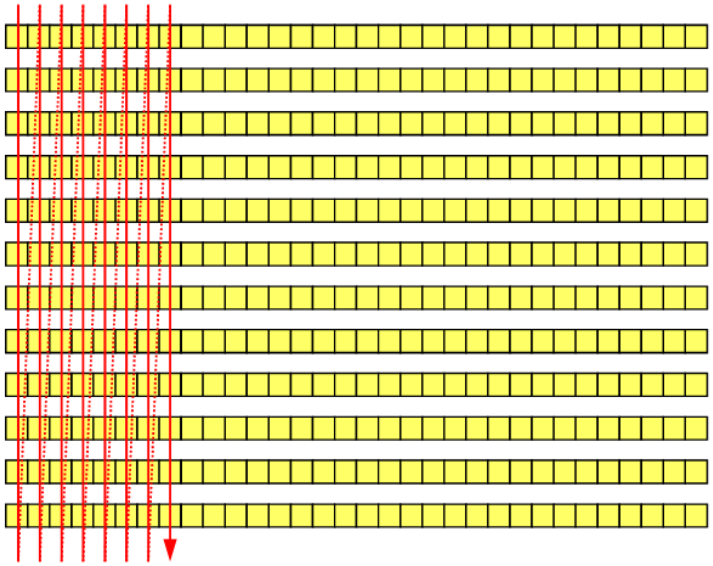
A consequência de todos esses pontos é que acessos com padrão previsível a elementos próximos uns dos outros é a estratégia *ideal* para garantir a utilização (quase) ótima das *caches*.

Sabendo disso, agora vamos dispor os elementos de p na memória. Como todas as máquinas M_i processam todas as tarefas J_j, há um tempo de processamento p_{ij} para cada operação O_{ij}; então podemos naturalmente dispor esses elementos em uma matriz. Porém, escolher se as tarefas ficam dispostas em colunas ou em linhas gera um impacto no desempenho do programa. Para saber qual escolha é a mais assertiva, analisemos os padrões de acesso a p.

Considere o algoritmo da heurística $LR(x)$. Note que acessos a p são feitos apenas no cálculo dos tempos de conclusão $C(i, \pi_j)$. Note, também, o padrão de acesso à matriz de memoização em $C(i, \pi_j)$: sua construção é feita linha por linha[2]. p, portanto, é também acessada linha por linha. Cada linha na matriz de memoização corresponde a uma tarefa; portanto, a melhor escolha para p é designar as linhas como tarefas e as colunas como máquinas. Dessa forma, os tempos de processamento de uma tarefa estão contíguos em memória, estando na mesma linha de *cache* ou em linhas vizinhas, de forma que o acesso aos elementos é extremamente rápido.

[2] Essa forma de iteração numa matriz é conhecida como *row-major traversal* (MEYERS, 2014).

Figura 9 – Iteração em matriz, do tipo *column-major*



Fonte: Meyers (2014)

3.4 Implementando a solução

Para implementar a heurística e as otimizações propostas, utilizaremos a linguagem Python. Apesar de ser lenta(WELLONS, 2019), ela é ótima para prototipação e *profiling* rápidos; além disso, os fatos aqui apresentados apresentam efeitos positivos independentemente da linguagem escolhida, ainda que esses efeitos sejam menores em linguagens interpretadas.

4 RESULTADOS E DISCUSSÃO

Esse capítulo, por fim, apresenta os resultados da implementação da heurística $LR(x)$ e das otimizações propostas no capítulo 3.

As instâncias utilizadas são as de Taillard (1993), mais especificamente as instâncias `ta001`, com 20 tarefas e 5 máquinas, `ta011`, com 20 tarefas e 10 máquinas, e `ta021`, com 20 tarefas e 20 máquinas. Foram rodadas versões da heurística com $x = 1$ e $x = 20$. Como, para verificar se a modificação do projeto do algoritmo resulta em mudanças nos tempos de execução do código, devemos deixar todos os outros parâmetros estáticos, selecionamos a instância `ta001` para os testes em cada etapa da implementação.

Os testes foram realizados em um notebook Dell Inspiron 5437, com processador Intel i5-4200U (*clock* de 1.6 GHz, cache de 3 MB) e 6 GB de RAM.

4.1 Implementação *ipsis litteris*

A primeira implementação foi uma transcrição direta do algoritmo descrito por Liu e Reeves (2001).

Para a instância `ta001`, $LR(1)$ gera a sequência

$$\pi = (2, 16, 8, 14, 13, 15, 5, 18, 12, 6, 11, 10, 7, 1, 0, 19, 3, 9, 4, 17), \tag{4.1}$$

cujo tempo total de conclusão $C_{sum} = 29315$. O menor tempo de execução em 10 rodadas da heurística foi de, aproximadamente, 3.64s.

$LR(20)$ gera a sequência

$$\pi = (3, 2, 16, 14, 12, 11, 10, 18, 8, 13, 7, 1, 5, 0, 15, 6, 9, 4, 17, 19), \tag{4.2}$$

cujo tempo total de conclusão $C_{sum} = 28425$. O menor tempo de execução em 10 rodadas de $LR(20)$ foi de, aproximadamente, 73.02s.

4.2 Estrutura similar ao *selection sort*

A união de π e U deu-se por meio de *slices*, isto é, trechos de um *array*. Eles funcionam como uma espécie de janelas, permitindo ao código ver apenas a parte que lhe é pertinente.

O menor tempo de execução em 10 rodadas de $LR(1)$ para a instância `ta001` foi de, aproximadamente, 2.81s. O menor tempo de execução em 10 rodadas de $LR(20)$ para a instância `ta001` foi de, aproximadamente, 56.60s.

Figura 10 – Resultados do *profiling* de $LR(x)$, na versão *ipsis litteris*

```
>>> cProfile.run('LR(ta001)')
    # ...
8757686/7206    7.061    0.000    9.050    0.001 lr.py:29(C_J)
8480/2110    0.013    0.000    7.383    0.003 lr.py:40(C)
      836    0.001    0.000    0.001    0.000 lr.py:53(w)
      209    0.000    0.000    2.258    0.011 lr.py:57(IT)
      836    0.002    0.000    2.257    0.003 lr.py:62(f)
     1045    0.001    0.000    0.008    0.000 lr.py:68(t_p)
    14345    0.003    0.000    0.003    0.000 lr.py:69(f)
 1254/209    0.002    0.000    2.261    0.011 lr.py:75(C_p)
      209    0.000    0.000    3.948    0.019 lr.py:83(AT)
      209    0.001    0.000    6.206    0.030 lr.py:88(xi)
        1    0.000    0.000    2.875    2.875 lr.py:95(C_sum)
    # ...
```

Fonte: o autor

Figura 11 – Resultados do *profiling* de $LR(x)$, na versão com programação dinâmica

```
>>> cProfile.run('LR(ta001)')
    # ...
     7206    0.196    0.000    0.224    0.000 lr.py:30(C_J)
8480/2110    0.015    0.000    0.209    0.000 lr.py:59(C)
      836    0.001    0.000    0.001    0.000 lr.py:72(w)
      209    0.000    0.000    0.098    0.000 lr.py:76(IT)
      836    0.002    0.000    0.097    0.000 lr.py:81(f)
     1045    0.001    0.000    0.008    0.000 lr.py:87(t_p)
    14345    0.003    0.000    0.003    0.000 lr.py:88(f)
 1254/209    0.002    0.000    0.109    0.001 lr.py:94(C_p)
    # ...
```

Fonte: o autor

4.3 Programação dinâmica, parte 1

Ao executar um *profiler* do código, temos o resultado mostrado na figura 10.

Note que boa parte do tempo gasto pela heurística, cerca de 9s, é gasto na função `C_J`, que implementa a equação 2.1.

Ao fazer a modificação para o algoritmo de programação dinâmica e novamente executar o *profiler*, temos o resultado mostrado na figura 11.

Note que o tempo de execução baixou e muito, para menos de 200ms. O menor

tempo de execução em 10 rodadas de $LR(20)$ para a instância `ta001` foi de, aproximadamente, 3.97s.

4.4 Programação dinâmica, parte 2

A função `C()`, que implementa a equação 2.4, também está sofrendo com recursões. Aplicando o mesmo algoritmo de programação dinâmica que aplicamos em `C_J()` e, em seguida, executando o *profiler*, temos o resultado apresentado na figura 12. O menor tempo de execução em 10 rodadas de $LR(20)$ para a instância `ta001` foi de, aproximadamente, 2.14s.

Figura 12 – Resultados do *profiling* de $LR(x)$, na 2ª versão com programação dinâmica

```
>>> cProfile.run('LR(t)')
    # ...
      836    0.030    0.000    0.034    0.000 lr.py:29(C_J)
     2110    0.077    0.000    0.090    0.000 lr.py:44(C)
      836    0.001    0.000    0.001    0.000 lr.py:68(w)
      209    0.000    0.000    0.069    0.000 lr.py:72(IT)
      836    0.002    0.000    0.068    0.000 lr.py:77(f)
     1045    0.001    0.000    0.008    0.000 lr.py:83(t_p)
    14345    0.004    0.000    0.004    0.000 lr.py:84(f)
 1254/209    0.002    0.000    0.054    0.000 lr.py:91(C_p)
      209    0.000    0.000    0.068    0.000 lr.py:99(AT)
    # ...
```

Fonte: o autor

4.5 Instâncias `ta011` e `ta021`

As instâncias `ta011` e `ta021` foram utilizadas nas execuções da versão *ipsis litteris* e da versão final.

Com a versão *ipsis litteris*, O menor tempo de execução em 10 rodadas de $LR(1)$ para a instância `ta011` foi de, aproximadamente, 1578s; enquanto que o menor tempo de execução em 10 rodadas de $LR(20)$ foi de, aproximadamente, 31335s.

Já com a última versão, o menor tempo de execução em 10 rodadas de $LR(1)$ foi de, aproximadamente, 0.39s; enquanto que o menor tempo de execução em 10 rodadas de $LR(20)$ foi de, aproximadamente, 7.23s.

Para a instância `ta021`, com a última versão, o menor tempo de execução em 10 rodadas de $LR(1)$ foi de, aproximadamente, 1.42s; enquanto que o menor tempo de execução em 10 rodadas de $LR(20)$ foi de, aproximadamente, 27.87s.

4.6 Discussão

De todas as otimizações propostas, a otimização baseada em programação dinâmica foi a que retornou resultados mais expressivos, em termos de tempo de execução.

Por exemplo, na instância `ta001` para $x = 20$, comparando a versão *ipsis litteris* com a última versão, tivemos um aumento no desempenho de aproximadamente 34 vezes. Já para a instância `ta011`, tivemos um aumento de 4334 vezes.

Ainda que a otimização envolvendo programação dinâmica aloque uma matriz, houve, de fato, uma redução no consumo de memória, dado que, para fazer uma recursão, o *runtime* da linguagem precisa salvar o conteúdo da pilha da função. Dado que há milhares de recursões, há um custo absurdo de consumo de memória; perto desse consumo, a alocação de uma matriz é quase insignificante.

5 CONSIDERAÇÕES FINAIS

Nesse trabalho, objetivamos implementar, de forma eficiente e eficaz, a heurística $LR(x)$ para o problema do escalonamento do tipo $flowshop$ de permutação utilizando conhecimentos do *design* orientado a dados. Com base nas argumentações e resultados mostrados nos capítulos anteriores, concluímos que tivemos êxito no nosso objetivo, tendo em vista que as otimizações realizadas a nível de projeto realmente impactaram de forma positiva o desempenho do código.

Para trabalhos futuros, temos várias oportunidades de otimização. A primeira delas é a escolha da linguagem. Python é uma ferramenta ótima para prototipação rápida e experimentação; porém, é extremamente lenta, principalmente pelo fato do bytecode gerado pelo CPython, a principal implementação da linguagem, ser praticamente uma transcrição do código original (WELLONS, 2019). Além disso, o fato de CPython possuir um *lock* global (BEAZLEY, 2010) não nos permite paralelizar a construção de soluções. Essa peculiaridade não é exclusiva de Python; outras implementações de linguagens interpretadas, como Ruby e JavaScript, são *single-threaded*. A forma mais simples de resolver o problema em questão é escrever o código em uma linguagem que dê ao usuário o poder de controlar os recursos da máquina, como C, C++, Rust ou Zig.

A nível de implementação, existem duas possibilidades de otimização, uma de desempenho e outra de memória, que não exploramos por falta de tempo hábil. Segue a primeira possibilidade: para computar os tempos de conclusão de $\pi(k)$, precisamos saber apenas os tempos de conclusão de $\pi(k-1)$. (Essa definição é recursiva.) Isso significa que podemos construir a matriz de tempos de conclusão C de forma iterativa, compondo-a com uma nova linha, ao invés de recalculá-la, a cada iteração.

A segunda otimização, de memória, segue da primeira. Os detalhes dos tempos de conclusão Considere o seguinte: depois que uma tarefa $\pi(k)$ é escalonada, ela permanece no mesma posição. Os detalhes sobre os tempos de conclusão das tarefa $\pi(k-1)$ estão codificados nos tempos de conclusão da tarefa ($\pi(k)$). (Essa definição também é recursiva.) Sendo assim, podemos guardar apenas com os tempos de conclusão da tarefa escalonada na iteração anterior. Isso reduziria o consumo de memória de $\mathcal{O}(mn)$ para $\mathcal{O}(m)$.

Por fim, Fernandez-Viagas e Framinan (2015) fornecem uma nova heurística, $FF(x)$, que contém uma função de índice $\xi(j, k)$ mais acurada, permitindo o descarte da computação do tempo de conclusão J_a, computação essa de complexidade $\mathcal{O}(mn)$ (ROJAS, 2015, p. 50). Com isso, conseguem reduzir a complexidade computacional de $LR(x)$ de $\mathcal{O}(xn^3m)$ para $\mathcal{O}(xn^2m)$, gerando resultados ainda melhores. A heurística $FF(x)$ é, agora,

considerada o estado-da-arte. Um trabalho futuro é substituir e/ou remover as “peças” necessárias para transformar a heurística $LR(x)$ na $FF(x)$.

BIBLIOGRAFIA

ACTON, Mike. Data-oriented design and C++. In: CppCon, Washington. Disponível em: <https://www.youtube.com/watch?v=rX0ItVEVjHc>. Acesso em: 26 set. 2019.

_____. *Three big lies*. 2008. Disponível em: <https://cellperformance.beyond3d.com/articles/2008/03/three-big-lies.htm>. Acesso em: 26 set. 2019.

ADAMSON, Gary W. *The On-Line Encyclopedia of Integer Sequences*. 2007. Disponível em: <http://oeis.org/A134762>. Acesso em: 26 set. 2019.

BAKER, Kenneth R. *Introduction to sequencing and scheduling*. 1. ed. New York: Wiley, 1974. ISBN: 978-0471045557.

BEAZLEY, David. Understanding the Python GIL. In: PyCon, 2010, Atlanta. Disponível em: <http://dabeaz.com/python/UnderstandingGIL.pdf>. Acesso em: 26 set. 2019.

BENTLEY, Jon Louis. *Writing efficient programs*. Englewood Cliffs: Prentice-Hall, 1982. ISBN: 978-0139702440.

CARRUTH, Chandler. Efficiency with algorithms, performance with data structures. In: CppCon, Washington. Disponível em: <https://www.youtube.com/watch?v=fHNmRkzxHWs>. Acesso em: 26 set. 2019.

CORMEN, Thomas H. et al. *Introduction to algorithms*. 3. ed. Cambridge: The MIT Press, 2009. ISBN: 978-0262033848.

FABIAN, Richard. *Data-oriented design*: software engineering for limited resources and short schedules. [S.l.: s.n.], 2018. ISBN: 978-1916478701. Disponível em: <http://www.dataorienteddesign.com/dodbook>. Acesso em: 29 set. 2019.

FERNANDEZ-VIAGAS, Victor; FRAMINAN, Jose M. A new set of high-performing heuristics to minimise flowtime in permutation flowshops. *Computers & Operations Research*, v. 53, p. 68–80, 2015. DOI: 10.1016/j.cor.2014.08.004.

GAMMA, Erich et al. *Design patterns*: elements of reusable object-oriented software. 1. ed. Reading: Addison-Wesley, 1994. ISBN: 978-0201633610.

GRAHAM, R. L. et al. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, v. 5, n. 2, p. 287–326, 1979. DOI: 10.1016/S0167-5060(08)70356-X.

HENNESSY, John L.; PATTERSON, David A. *Computer architecture*: a quantitative approach. 5. ed. Waltham: Morgan Kaufmann, 2012. ISBN: 978-0-12-383872-8.

JACOBSON, Ivar; BOOCH, Grady; RUMBAUGH, James. *The unified software development process*. 1. ed. Reading: Addison-Wesley, 1999. ISBN: 978-0321822000.

JOHNSON, S. M. Optimal two- and three-state production schedules with setup times included. *Naval research logistics quarterly*, v. 1, n. 1, p. 61–68, 1954. DOI: 10.1002/nav.3800010110.

KAPOOR, Rajiv. *Avoiding the cost of branch misprediction*. 2009. Disponível em: <https://software.intel.com/en-us/articles/avoiding-the-cost-of-branch-misprediction/>. Acesso em: 30 set. 2019.

KERNIGHAN, Brian W.; PLAUGER, P. J. *The elements of programming style*. 2. ed. New York: McGraw-Hill, 1978. ISBN: 978-0070342071.

KORTE, Bernhard; VYGEN, Jens. *Combinatorial optimization*: theory and algorithms. 5. ed. Heidelberg: Springer-Verlag, 2012. ISBN: 978-3642244872.

LIU, Jiyin; REEVES, Colin R. Constructive and composite heuristic solutions to the $P//\sum C_i$ scheduling problem. *European Journal of Operational Research*, v. 132, n. 2, p. 439–452, 2001. DOI: 10.1016/s0377-2217(00)00137-5.

LLOPIS, Noel. *Data-Oriented Design (Or Why You Might Be Shooting Yourself in The Foot With OOP)*. 2009. Disponível em: <http://gamesfromwithin.com/data-oriented-design>. Acesso em: 29 set. 2019.

MACCARTHY, B. L.; LIU, Jiyin. locationing the gap in scheduling research: a review of optimization and heuristic methods in production scheduling. *International journal of production research*, v. 31, n. 1, p. 59–79, 1993. DOI: 10.1080/00207549308956713.

MEYERS, Scott. CPU caches and why you care. In: code::dive, 2014, Wrocław. Disponível em: <https://www.aristeia.com/TalkNotes/codedive-CPUCachesHandouts.pdf>. Acesso em: 26 set. 2019.

PEARL, Judea. *Heuristics*: intelligent search strategies for computer problem solving. Reading: Addison-Wesley, 1984. ISBN: 978-0201055948.

ROJAS, Alexander Javier Benavides. *Heuristics for flow shop scheduling*: considering non-permutation schedules and a heterogeneous workforce. Tese (Doutorado em Ciência da Computação) – Universidade Federal do Rio Grande do Sul, Porto Alegre, 2015.

TAILLARD, Éric. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, v. 64, n. 2, p. 278–285, 1993. DOI: 10.1016/0377-2217(93)90182-m.

WELLONS, Chris. *The CPython Bytecode Compiler is Dumb*. 2019. Disponível em: <http://nullprogram.com/blog/2019/02/24/>. Acesso em: 26 set. 2019.

WILL, Brian. *Object-oriented programming is bad*. 2019. Disponível em: <https://www.youtube.com/watch?v=QM1iUe6IofM>. Acesso em: 28 set. 2019.

WIRTH, Niklaus. *Algorithms + data structures = programs*. 1. ed. Upper Saddle River: Prentice-Hall, 1976. ISBN: 978-0130224187.