Relatório Técnico: Otimização de Escalonamento de Tarefas em Sistemas Computacionais

Sistema de Gerenciamento de Tarefas

24 de Novembro de 2025

Resumo

Este documento apresenta uma análise detalhada do problema de gerenciamento de tarefas em sistemas computacionais com recursos limitados. Comparamos duas abordagens algorítmicas: uma heurística gulosa (*Weighted Job Scheduling*) e uma solução exata baseada em Programação Dinâmica (*0/1 Knapsack Problem*). O projeto destaca-se pela implementação manual de todos os algoritmos, incluindo o método de ordenação *Quick Sort*, e pela análise crítica da otimização de desempenho.

Conteúdo

1 Definição do Problema e Contextualização

1.1 Cenário: Centro de Processamento de Dados

O sistema foi projetado para operar em um Centro de Processamento de Dados (CPD) com recursos computacionais finitos. O problema central é o **agendamento de tarefas** (Task Scheduling) sob restrição de tempo. Diferente de um sistema operacional de propósito geral que busca justiça (fairness), nosso objetivo é maximizar o **valor entregue** ao negócio.

Cada tarefa i é definida pela tupla (t_i, v_i, d_i), onde:

- t_i (Tempo): Custo de processamento.
- v_i (Prioridade): Valor agregado pela conclusão da tarefa.
- d_i (Prazo): Fator de urgência que influencia a prioridade dinâmica.

O problema é selecionar um subconjunto S de tarefas tal que $\sum_{i \in S} t_i \leq T_{\text{total}}$ e $\sum_{i \in S} v_i$ seja máximo.

1.2 O Dilema da Decisão

O sistema deve decidir entre uma abordagem de resposta imediata (Guloso) ou uma garantia de otimalidade (Programação Dinâmica). A escolha impacta diretamente a eficiência operacional do CPD: algoritmos rápidos liberam o escalonador para outras funções, mas decisões subótimas desperdiçam ciclos de CPU com tarefas de menor valor.

2 Implementação e Metodologia

2.1 Implementação Manual de Algoritmos

Em conformidade com os requisitos do projeto, **todos os algoritmos principais foram implementados manualmente**, sem o uso de bibliotecas de otimização ou funções de ordenação nativas (como `Collections.sort`). Isso demonstra o domínio sobre a lógica algorítmica e permite um controle refinado sobre o comportamento do sistema.

2.2 Método de Ordenação: Quick Sort

Para a etapa de pré-processamento (essencial para o algoritmo Guloso e organizacional para o DP), implementamos o algoritmo **Quick Sort**.

Justificativa da Escolha: Optamos pelo Quick Sort em detrimento de algoritmos mais simples como Bubble Sort ou Insertion Sort ($O(n^2)$) devido à sua eficiência média de $O(n \log n)$. Em um sistema de escalonamento real, onde o número de tarefas pode ser grande, a eficiência da ordenação é crucial para não se tornar um gargalo antes mesmo do processo de seleção começar. O critério de ordenação foi a **Densidade** ($\frac{\text{Prioridade}}{\text{Tempo} \times (\text{Prazo}+1)}$), ordenando as tarefas da mais "lucrativa" para a menos lucrativa.

2.3 Interface Humano-Computador (IHC)

O sistema foi desenvolvido com uma interface baseada em terminal (CLI - Command Line Interface). **Justificativa:** A escolha por uma interface textual permite foco total na lógica algorítmica e na visualização clara dos dados de saída (tabelas de comparação), sem o overhead de processamento gráfico. A saída é organizada e explicativa, facilitando a análise imediata dos resultados pelo operador.

3 Estratégias Algorítmicas Implementadas

3.1 Abordagem Gulosa: A Heurística da Densidade

Para o algoritmo guloso, não basta ordenar por valor ou por tempo. Desenvolvemos uma métrica composta denominada **Densidade**, que equilibra valor, custo e urgência:

$$\text{Densidade}_i = \frac{v_i}{t_i \times (d_i + 1)}$$

Esta fórmula prioriza tarefas que entregam muito valor rapidamente e que possuem prazos apertados. A implementação utiliza o **Quick Sort** para ordenar as tarefas decrescentemente por esta densidade e, em seguida, itera linearmente selecionando as que cabem no tempo restante.

3.2 Abordagem Exata: Programação Dinâmica (Knapsack)

Modelamos o problema como uma variação do **Problema da Mochila 0/1**, onde a "capacidade da mochila" é o tempo total disponível T. A Programação Dinâmica (DP) constrói uma solução passo-a-passo, garantindo que nenhuma combinação de tarefas que resulte em um valor total maior seja ignorada. Ao contrário do guloso, o DP consegue "ver o futuro" e preencher lacunas de tempo com combinações perfeitas de tarefas menores.

4 Documento Analítico

4.1 Formulação do Problema

Seja um conjunto de n tarefas, onde cada tarefa i tem um peso w_i (tempo) e um valor v_i (prioridade). Seja W a capacidade total. Queremos maximizar $\sum v_i$ sujeito a $\sum w_i \leq W$.

4.2 Modelo de Solução (Programação Dinâmica)

Definimos $dp[i][w]$ como o valor máximo considerando as primeiras i tarefas com capacidade w.

$$dp[i][w] = \max(dp[i-1][w], v_i + dp[i-1][w - w_i]) \quad \text{se } w_i \leq w$$

5 Análise Assintótica e Implementação

A seguir, detalhamos a complexidade teórica corroborada por trechos do código fonte desenvolvido.

5.1 Escalonador Guloso

A complexidade é dominada pela ordenação. Utilizamos o Quick Sort, que possui caso médio $O(n \log n)$. A seleção subsequente é linear $O(n)$.

```
private int partition(List<Tarefa> tarefas, int low, int high) {
    Tarefa pivot = tarefas.get(high);
    double pivotDensity = pivot.getDensidade();
    int i = (low - 1);
    for (int j = low; j < high; j++) {
        // Ordena decrescente pela densidade calculada
        if (tarefas.get(j).getDensidade() > pivotDensity) {
            i++;
            swap(tarefas, i, j);
        }
    }
    swap(tarefas, i + 1, high);
    return i + 1;
}
```

Listing 1: Partition do Quick Sort (Ordenação por Densidade)

5.2 Escalonador DP (Knapsack)

A complexidade é pseudo-polinomial $O(n \cdot W)$, onde n é o número de tarefas e W é o tempo total disponível. Isso ocorre devido à estrutura de laços aninhados necessária para preencher a tabela de memorização.

```
// n = tarefas.size(), T = tempoTotalDisponivel
for (int i = 1; i <= n; i++) {
    int tempoTarefa = tarefas.get(i-1).getTempo();
    int valorTarefa = tarefas.get(i-1).getPrioridade();

    for (int t = 1; t <= T; t++) {
        // Opcao 1: Nao incluir
        dp[i][t] = dp[i-1][t];

        // Opcao 2: Incluir se couber
        if (tempoTarefa <= t) {
            int valorCom = dp[i-1][t-tempoTarefa] + valorTarefa;
            if (valorCom > dp[i][t]) {
                dp[i][t] = valorCom;
            }
        }
    }
}
```

Listing 2: Núcleo da Programação Dinâmica

6 Análise de Desempenho Experimental

Realizamos testes empíricos comparando as duas abordagens.

6.1 Resultados Obtidos

Tabela 1: Comparativo de Valor Total e Tempo de Execução

Cenário	Algoritmo	Valor Total	Tempo Usado	Execução (ms)
1. Favorável	Guloso	37	10	≈ 0.15
(Limite: 10)	DP	37	10	≈ 0.20
2. Guloso Falha	Guloso	11	2	≈ 0.05
(Limite: 6)	**DP**	**20**	5	≈ 0.10
3. Idênticas	Guloso	60	30	≈ 0.08
(Limite: 30)	DP	60	30	≈ 0.15
4. Aleatório	Guloso	335	98	≈ 0.16
(Limite: 100)	**DP**	**352**	100	≈ 0.45

6.2 Consistência Teórica vs. Empírica

Os dados da Tabela 1 confirmam as expectativas teóricas:

- **Guloso ($O(n \log n)$)**: O tempo de execução manteve-se extremamente baixo e estável ($\approx$ 0.05ms a 0.16ms), pois depende apenas do número de tarefas (n), que é pequeno nos testes.
- **DP ($O(n \cdot W)$)**: O tempo de execução aumentou visivelmente no Cenário 4 ($\approx$ 0.45ms). Isso ocorre porque o limite de tempo (W) aumentou para 100, forçando a matriz dp a ser maior ($n \times 100$) e exigindo mais iterações do laço interno.

A "falha" do Guloso no Cenário 2 (Valor 11 vs 20 do DP) ilustra perfeitamente a limitação teórica da heurística: ao escolher uma tarefa densa mas que consome muito tempo, ele bloqueou a inserção de múltiplas tarefas menores que, somadas, valeriam mais. O DP, explorando todas as combinações na matriz, evitou essa armadilha.

7 Análise Crítica Final

7.1 Adequação da Abordagem

Por que a abordagem escolhida foi a ideal? A escolha pela Programação Dinâmica foi ideal porque, no contexto de gerenciamento de tarefas críticas, a perda de valor (prioridade) é inaceitável. O algoritmo Guloso, apesar de rápido, falhou em maximizar o uso dos recursos em cenários de "mochila cheia", deixando lacunas de tempo que poderiam ser preenchidas por tarefas mais valiosas. O DP preencheu essas lacunas perfeitamente.

7.2 Metodologia e Trabalho em Equipe

Como o trabalho em equipe e a metodologia ágil influenciaram? O desenvolvimento seguiu uma abordagem incremental. Inicialmente, focamos na implementação da estrutura básica ('Tarefa') e do algoritmo Guloso para ter um MVP (Minimum Viable Product). Em seguida, implementamos o DP para comparação. A divisão clara

de responsabilidades (implementação de algoritmos vs. infraestrutura de testes) e o uso de testes automatizados (‘Comparador‘) permitiram identificar rapidamente as falhas do algoritmo Guloso, guiando a decisão final de recomendar o DP.

8 Conclusão

A implementação do algoritmo *Knapsack* via Programação Dinâmica, suportada por uma ordenação eficiente *Quick Sort*, provou ser a solução superior. O sistema atende a todos os requisitos de otimização, robustez e implementação manual, garantindo o melhor desempenho possível para o centro de processamento de dados.