Relatório Técnico – Trabalho A(TA) – Algoritmos Gulosos e Programação Dinâmica

Mateus S. Ribeiro[1], Sergio A. Cezar[1]

[1]Departamento de Informática – Universidade Estadual de Maringá (UEM)

ra128459@uem.br, ra134680@uem.br

Resumo. *Este trabalho aborda a resolução de dois problemas clássicos de otimização, utilizando técnicas de algoritmos gulosos e programação dinâmica.*

O primeiro problema, Arrumando Tarefas, tem como objetivo selecionar quais tarefas devem ser executadas a fim de minimizar o valor total perdido, considerando restrições de prazo e capacidade limitada de execução. Como resultado, foi possível garantir que, em todos os casos de teste, o valor perdido fosse minimizado de forma ótima, com tempo de execução adequado para instâncias grandes.

O segundo problema, Efeito Dominó, consiste em determinar o menor número de peças de dominó que devem ser movidas para que a distância entre todas as peças consecutivas não exceda um limite especificado. A solução implementada foi capaz de computar corretamente o número mínimo de movimentos ou identificar situações inviáveis.

1. Problema Arrumando Tarefas

1.1. Problema

Dado um conjunto de n tarefas {1,2,..,n}, cada tarefa i associada a um valor monetário Vi e um prazo máximo de execução Ti, determinar o subconjunto a ser processado de forma a minimizar a soma dos valores perdidos, considerando que apenas uma tarefa pode ser executada por unidade de tempo e que há um número limitado de horas disponíveis.

Técnica utilizada: Algoritmo guloso.

1.2. Resolução

O problema foi resolvido por meio de uma abordagem gulosa que prioriza as tarefas de maior valor. Optamos por adotar a estratégia que consiste em ordenar todas as tarefas em ordem decrescente de valor monetário e, para cada uma, buscar o horário mais próximo do prazo máximo em que ainda seja possível alocá-la. Para gerenciar os horários disponíveis de maneira eficiente, adotamos uma estrutura de conjuntos disjuntos com compressão de caminho (buscamos garantir tempo de execução eficiente em instâncias com grande número de tarefas, pois em outras tentativas o desempenho foi insatisfatório), que funciona da seguinte forma: o algoritmo utiliza o vetor proximo_livre para representar os tempos disponíveis, usando a estrutura Union-Find. A cada tarefa, ele executa a operação de busca (find) para encontrar o último tempo livre antes do prazo, seguindo os ponteiros em proximo_livre até chegar a um tempo que aponta para si mesmo, ou seja, quando proximo_livre[x] == x. Durante essa busca, aplica-se a técnica de compressão de caminho (path compression), que atualiza todos os ponteiros do caminho para apontarem diretamente para a raiz, reduzindo significativamente o custo de buscas futuras. Caso um tempo livre seja encontrado, o

algoritmo agenda a tarefa e realiza a união (union), fazendo com que o tempo ocupado passe a apontar para o tempo anterior disponível, caso contrário, a tarefa não é agendada. Assim, o Union-Find permite gerenciar os tempos de forma eficiente, sem percorrer todos manualmente.

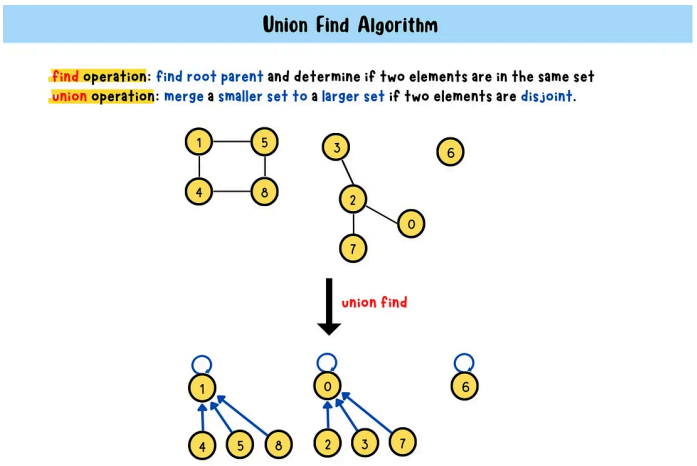

Figura 1. Funcionamento do union-find
Fonte: Claire Lee, Medium (25 out. 2022).

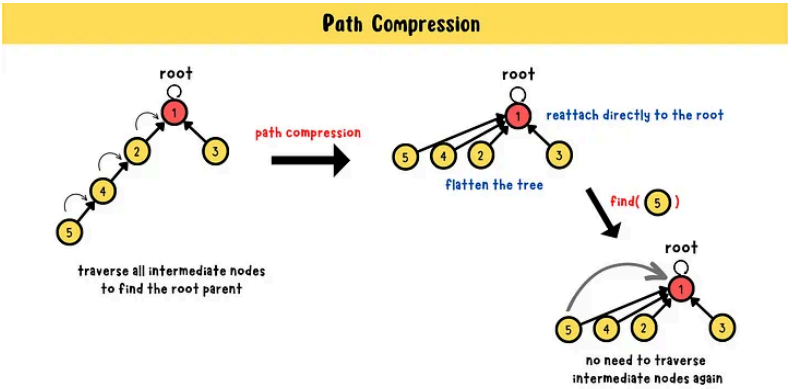

Figura 2. Funcionamento do Path Compression
Fonte: Claire Lee, Medium (25 out. 2022).

Demonstração da propriedade da escolha gulosa: Para obter o maior lucro possível (ou minimizar o prejuízo), a tarefa escolhida será a que rende o maior lucro entre as disponíveis. Após essa escolha, ela será agendada no último momento em que poderia ser feita, de forma a liberar mais espaço para tarefas urgentes, visto que o momento em que a tarefa é realizada não interfere no lucro. Assim, se a solução ótima incluir uma tarefa de menor valor em uma hora h, poderíamos trocá-la pela tarefa de maior valor que coubesse naquele horário, resultando em solução de valor não inferior. Portanto, qualquer solução ótima pode ser transformada para seguir a escolha gulosa sem perda de qualidade. Essa demonstração pode ser vista como uma forma do argumento de troca (exchange argument), conforme descrito na Seção 16.2 de Cormen et al. (2009). Ou seja, todas as tarefas que resultam no maior lucro foram escolhidas.
Subproblema: Dado um conjunto de tarefas ainda não agendadas e um conjunto de

horários disponíveis, escolher o próximo horário de execução para a tarefa de maior valor, ou seja, devemos encontrar o maior horário livre ≤ Ti.
Subestrutura ótima: O problema pode ser decomposto em subproblemas de alocação de horários para tarefas restantes, e a solução ótima é composta pela união das soluções ótimas dos subproblemas. Portanto, é possível afirmar que a solução ótima dos subproblemas é feita seguindo a estratégia gulosa.

Algoritmo em pseudocódigo:

```
# Cada tarefa é representada por uma tupla que contém lucro e prazo.

função melhor_escolha(tempo_total, lista_de_tarefas)
    '''
    Segue uma ideia gulosa para agendar as tarefas:
    As tarefas que rendem os maiores lucros serão agendadas no último momento possível.
    '''
    ordenar lista_de_tarefas por lucro decrescente
    lucro_max ← soma de todos os lucros das tarefas
    lucro_obtido ← 0

    // Vetor que indica o próximo tempo livre para agendamento
    criar vetor proximo_livre de 0 até tempo_total
    para cada tarefa na lista_de_tarefas:
        lucro ← tarefa.lucro
        prazo ← tarefa.prazo

        // Busca o tempo mais próximo e disponível antes do prazo
        tempo_disponivel ← onde_agendar(proximo_livre, mínimo entre prazo e tempo_total)

        se tempo_disponivel > 0 então:
            // Marca esse tempo como ocupado e atualiza o próximo disponível
            agendar(proximo_livre, tempo_disponivel, tempo_disponivel - 1)
            lucro_obtido ← lucro_obtido + lucro

    // Retorna o lucro perdido
    retornar lucro_max - lucro_obtido
```

```
função onde_agendar(proximo_livre, tempo):
    // Busca o tempo livre mais próximo. Caso o tempo esteja livre, a tarefa será agendada, senão o vetor é atualizado
    // e é necessário buscar qual é o tempo livre.
    se proximo_livre[tempo] ≠ tempo então:
        proximo_livre[tempo] ← onde_agendar(proximo_livre, proximo_livre[tempo])
    retornar proximo_livre[tempo]

função agendar(proximo_livre, tempo_ocupado, novo_disponivel):
    // Atualiza o vetor para indicar qual o tempo deve ser ocupado.
    // Tarefa agendada em "tempo_ocupado". A partir de agora, esse tempo aponta para o "novo_disponivel"
    proximo_livre[onde_agendar(proximo_livre, tempo_ocupado)] ← onde_agendar(proximo_livre, novo_disponivel)
```

Análise da complexidade:

- Ordenação das tarefas: O(N log N)
- Para cada tarefa: busca e atualização de horários com union-find em O(α(N)), onde α é a inversa da função de Ackermann. A função de Ackermann (consulte o apêndice A) tem um crescimento extremamente rápido, portanto, sua inversa tem um crescimento extremamente lento:
    - Na prática α(n) ≤5 para qualquer valor usável de n, ou seja, é praticamente constante.
- Complexidade total: O(N log N), onde N é o número de tarefas.

1.3. Exemplo

Entrada:

5 3

5 1

20 1

8 2

15 3

25 2

Passo a passo:

- Ordenar por valor:
  (25,2), (20,1), (15,3), (8,2), (5,1).
- Primeira tarefa (lucro = 25): tenta agendar no tempo 2 → sucesso.
- Segunda tarefa (lucro = 20): tenta agendar no tempo 1 → sucesso.
- Terceira tarefa (lucro = 15): tenta agendar no tempo 3 → sucesso.
- Quarta tarefa (lucro = 8): tenta agendar no tempo 2 → tempo ocupado → tenta agendar no tempo 1 → ocupado → não há espaço.
- Quinta tarefa (lucro = 5): tenta agendar no tempo 1 → tempo ocupado → não há espaço
  Perda total: 8 + 5 = 13.

  Saída esperada: 13

1.4. Resultados e conclusões

Os testes com entradas diversas mostraram que a implementação foi capaz de minimizar o valor perdido em todas as instâncias de forma eficiente. A utilização da estrutura union-find contribuiu para um bom desempenho, mesmo em casos com grande número de tarefas. A primeira tentativa não utilizava a estrutura union-find, resultando em uma busca nas posições para agendar as tarefas, a qual não era tão eficiente. Implementar a estrutura foi inicialmente complexo, tendo em vista que era necessário atualizar os ponteiros corretamente após agendar cada tarefa. Essa estrutura foi utilizada para gerenciar os tempos disponíveis de forma eficiente, permitindo encontrar rapidamente o último instante livre antes do prazo de cada tarefa, sem precisar percorrer manualmente toda a agenda.

2. Problema Efeito Dominó

2.1. Problema

Dada uma sequência de peças de dominó alinhadas e numeradas de 1 a N, cada uma com uma altura máxima que limita a distância entre peças consecutivas, determinar o menor número de peças que devem ser movidas (excetuando a primeira e a última) para garantir que todas as distâncias estejam dentro do limite permitido. Caso seja

impossível, reportar -1.
Técnica utilizada: Programação dinâmica: Bottom-up

2.2. Resolução

Optamos por adotar a abordagem de programação dinâmica no estilo bottom-up, sem recursão, principalmente devido à clareza no controle das iterações e à facilidade para reconstruir os índices das peças mantidas na posição original.
Caracterização da subestrutura ótima: A ideia central é que a decisão de manter uma peça na posição atual depende apenas das decisões anteriores. Em outras palavras, para determinar a solução ótima até a peça i, é suficiente conhecer qual foi a última peça mantida e quantas peças foram mantidas até ela. Isso garante que uma solução ótima para o prefixo de peças até i pode ser construída a partir de soluções ótimas para prefixos menores (até $j < i$).
Se temos uma sequência ótima até a peça j, podemos decidir se a peça i pode ser mantida no local original verificando se a distância entre a posição cumulativa de i e a de j não viola o limite imposto: $xi - xj \leq (i-j) \cdot H$
Se isso ocorrer, mantemos a peça i sem custo adicional. Caso contrário, ela precisa ser movida (portanto, não contribuímos com +1 ao total de peças mantidas).

Cálculo recursivo do valor ótimo:
opt(i) = número máximo de peças mantidas no lugar entre a primeira e a peça i.
Casos base: opt(1) = 1
Relação de recorrência: para cada $i > 1$:

- $opt(i) = \max\{\{opt(j)+1 \mid j < i \text{ e } xi - xj \leq (i - j) \cdot H\}\}$

Se não existe nenhum j viável, então opt(i) é inválido (marcado como $-\infty$).
O valor final será: opt(N) e o número mínimo de peças movidas será: N − opt(N) desde que $opt(N) \geq 2$. Caso contrário, a configuração é impossível.

Subproblemas sobrepostos:

- Para cada peça i, verificamos todos os índices anteriores j.
- Para cada par (j, i), recalculamos xi−xj e consultamos opt(j).
- Exemplo:
  Para N = 5, H = 2 e D = [1, 1, 3, 1], temos x = [0, 1, 2, 5, 6].
  Para calcular opt(4), precisamos verificar opt(0), opt(1), opt(2) e opt(3).
  Mas esses mesmos valores já foram utilizados anteriormente:
- opt(3) verifica opt(0), opt(1), opt(2). opt(2) verifica opt(0), opt(1), opt(1) verifica opt(0)

De maneira genérica, os subproblemas são:
$\{opt(k) \mid k=1\ldots N\}$ e cada opt(k) é consultado por todos os índices posteriores.

Algoritmo:
Estratégia: Bottom-up, sem recursão.

```
função efeito_domino(N, H, D)
    criar vetor x de tamanho N com zeros

    para i de 1 até N - 1:
        x[i] ← x[i - 1] + D[i - 1]

    // Inicializa o vetor de DP com -∞
    criar vetor dp de tamanho N com valores -∞
    dp[0] ← 1  // A primeira peça é sempre fixa

    // Verifica a melhor forma de manter peças no lugar até a posição i
    para i de 1 até N - 1:
        para j de 0 até i - 1:
            // Se o trecho entre j e i respeita a inclinação máxima
            se x[i] - x[j] ≤ (i - j) x H então:
                // Atualiza a melhor quantidade de peças fixas até i
                se dp[j] + 1 > dp[i] então:
                    dp[i] ← dp[j] + 1

        // Se não foi possível alcançar i mantendo nenhuma peça
        se dp[i] < 1 então:
            dp[i] ← -∞

    // Verifica se é possível manter a primeira e a última peças fixas
    se dp[N - 1] < 2 então:
        retornar -1
    senão:
        retornar N - dp[N - 1]  // Número mínimo de peças a serem movidas
```

1. Inicializar dp[0] = 1.
2. Para cada peça i = 1…N−1:
    - Inicializar dp[i] como inválido.
    - Para cada j=0…i−1:
        - Se $x[i] - x[j] \leq (i - j) \cdot H$, considerar dp[j]+1 como candidato.
    - Guardar o máximo válido.
3. Se dp[N-1] < 2, imprimir -1.
4. Senão, imprimir N − dp[N − 1].

<u>Recuperação da solução completa:</u>

Caso fosse necessário reconstruir a solução ótima, poderíamos manter um vetor prev[] que armazena, para cada posição, o índice anterior responsável pelo valor máximo. O procedimento seria:

- Inicializar prev[i] ← −1 para todos i.
- Sempre que dp[i] for atualizado com dp[j]+1, fazer prev[i] ← j.
- Após o término do preenchimento de dp[], reconstruir a sequência percorrendo prev[] de forma retroativa:

  s ← N−1

  Enquanto s ≠ −1:

  marcar peça s como mantida

  s ← prev[s]

Análise da complexidade:

- Para cada i, verifica todos os j < i: O(N²).

2.3. Exemplos

Entrada:

5 3

2 4 2 3

Explicação:

- Distâncias entre peças:
    - (2) ok, (4) inválido, (2) ok, (3) ok.
- Precisamos mover a peça 3 (a que cria distância 4).

Saída esperada: 1

2.4. Resultados e conclusões

Os casos de teste mostraram que o algoritmo foi capaz de identificar corretamente situações sem solução e computar o número mínimo de peças movidas nas demais. A principal dificuldade foi garantir a atualização correta do vetor de prefixos das distâncias, para que as verificações fossem feitas com precisão.

3. Referências

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). *Algoritmos: Teoria e Prática*. 3ª Edição. Rio de Janeiro: Elsevier.

Documentação oficial do Beecrowd: https://www.beecrowd.com.br

Problema 1704 – Arrumando Tarefas: https://judge.beecrowd.com/pt/problems/view/1704

Problema 2036 – Efeito Dominó: https://judge.beecrowd.com/pt/problems/view/2036

Lee, C. (2022, 25 de outubro). **Union Find Algorithm**. Medium. Disponível em: https://yuminlee2.medium.com/union-find-algorithm-ffa9cd7d2dba.

GEEKSFORGEEKS. *Introduction to Disjoint Set Data Structure or Union-Find Algorithm*. [S. l.], [s. n.], 18 abr. 2023. Disponível em: https://www.geeksforgeeks.org/dsa/introduction-to-disjoint-set-data-structure-or-union-find-algorithm/. Acesso em: 16 jul. 2025.

GEEKSFORGEEKS**.** *Inverse Ackermann Function*. GeeksforGeeks, 2022. Disponível em: https://www.geeksforgeeks.org/dsa/inverse-ackermann-function/. Acesso em: 16 jul. 2025.

Apêndice A – Função de Ackermann:

$$A(m,n) = \begin{cases} n+1 & if\, m = 0 \\ A(m-1,1) & if\, m > 0\ and\ n = 0 \\ A(m-1, A(m,n-1)) & if\, m > 0\ and\ n > 0. \end{cases}$$