Linhas de Montagem

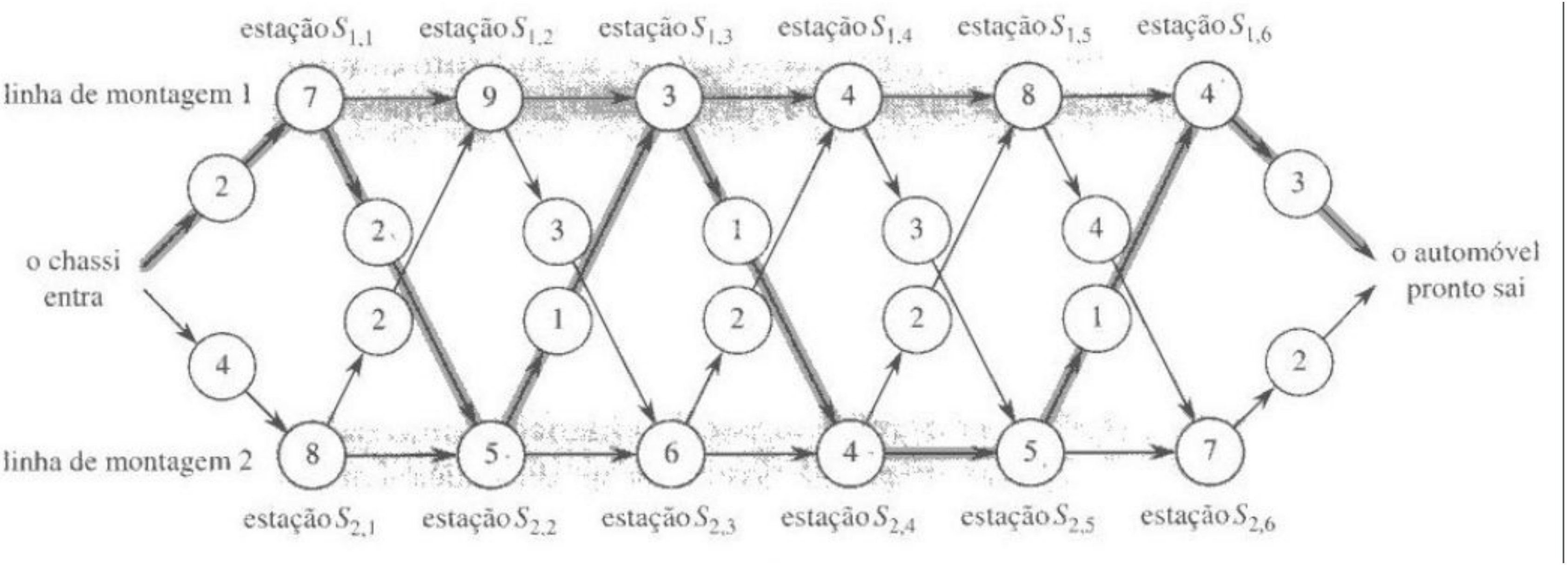
Trabalho 2 - Análise de Algoritmos

Vitor Hugo Honorato Tiago

Definição do Problema

- Duas linhas de montagem
- Cada linha possui n estações
- Cada estação exige um custo de processamento (tempo)
- Existe um custo para entrada em cada linha de montagem
- Existe um custo para saída em cada linha de montagem
- O produto pode ser transferido entre as linhas de montagem e essa transferência também tem um custo
- Qual o caminho com menor custo (tempo) dentro das linhas de montagem?

Definição do Problema



Etapa 1 - Subestrutura ótima

- A primeira estação j = 1, não aceita transferência
- Existem duas maneiras de chegar até uma estação j, quando j > 1
    - Continuando na mesma linha de produção
    - Vindo da outra linha de produção por transferência
- O menor tempo até uma estação j (j > 1) é a soma do menor tempo da estação anterior com o menor entre:
    - tempo de execução da estação na mesma linha.
    - tempo de transferência para outra linha + tempo de execução da estação na outra linha

$$se\ j\ =\ 1,\ só\ temos\ uma\ opção$$
$$se\ j\ >\ 1,\ min(S_{1,j-1},\ S_{2,j-1}\ +\ t_{2,j-1})$$

Etapa 2 - Solução recursiva

```
def assemblyLineBruteForce(S, t, e, x):
    n = len(S[0])

    def firstLine(station):
        if station == 0:
            return e[0] + S[0][0]
        keepLineCost = firstLine(station - 1) + S[0][station]
        switchLineCost = secondLine(station - 1) + S[0][station] + t[1][station-1]

        if keepLineCost < switchLineCost:
            return keepLineCost
        return switchLineCost

    def secondLine(station):
        if station == 0:
            return e[1] + S[1][0]
        keepLineCost = secondLine(station - 1) + S[1][station]
        switchLineCost = firstLine(station - 1) + S[1][station] + t[0][station-1]

        if keepLineCost < switchLineCost:
            return keepLineCost
        return switchLineCost

    finalFirstLine = x[0] + firstLine(n-1)
    finalSecondLine = x[1] + secondLine(n-1)

    minCost = min(finalFirstLine, finalSecondLine)

    return minCost
```

O algoritmo recursivo começa no fim das linhas de montagem e vai chamando recursivamente a soma até o início da linha

O início da linha é nosso caso base, pois se a estação for a primeiro seu custo é o custo de entrada da linha mais o custo da estação zero

Temos duas funções *firstLine* e *secondLine*. Cada uma delas faz duas chamadas recursivas, uma a si mesmo e uma a outra função.

Com isso, vamos construir a árvore de recorrência, a fórmula da recursão e calcular o custo do algoritmo

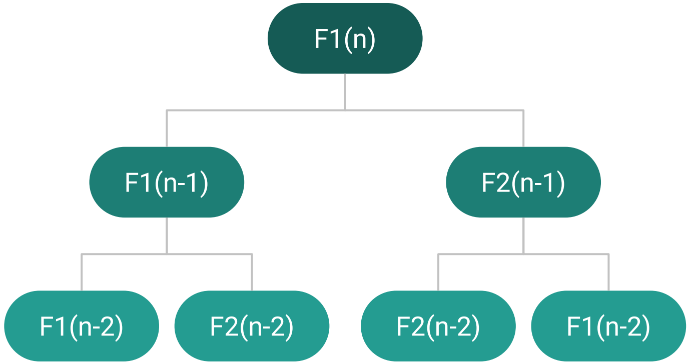
Etapa 2 - Solução recursiva

Nível 0 2^0

Nível 1 2^1

Nível 2 2^2

Nível i 2^i



Árvore de chamadas recursivas

A cada nível temos 1 retirado de n, logo a árvore terá altura n.

Cada nível i tem 2^i chamadas recursivas, assim o último nível da árvore terá 2^{n-1} folhas

São feitas duas chamadas a essa árvore no início do algoritmo, uma iniciando em F1 e outra em F2

Etapa 2 - Solução recursiva

Resolvendo pelo método da expansão:

$$T(n) = \begin{cases} \theta(1) & \text{se } j = 1 \\ 2T(n-1) + \theta(1) & \text{se } j > 1 \end{cases}$$

$$T(n) = 2T(n-1) + 1$$

$$T(n-1) = 2T(n-2) + 1$$

$$T(n-2) = 2T(n-3) + 1$$

$$T(n) = 2^{n-1}T(1) + \Theta(1)\sum_{i=0}^{n-2} 2^i$$

$$T(n) = \sum_{i=0}^{n-1} 2^i = 2^n - 1$$

Logo, a complexidade do algoritmo é $O(2^n)$

Etapa 3 - Solução iterativa

```
def assemblyLinesDynamicProgrammming(S, t, e, x):
    n = len(S[0])

    F1 = [0] * n
    F2 = [0] * n
    L1 = [0] * n
    L2 = [0] * n
    L1[n-1] = 0
    L2[n-1] = 1

    F1[0] = e[0] + S[0][0]
    F2[0] = e[1] + S[1][0]
```

O algoritmo iterativo é bottom up. Começa nos menores subproblemas (início das linhas) e vai até o problema completa, o custo no fim das linhas.

O custo de tempo do algoritmo é linear, visto que só existe um for variando de 1 até n. Logo a complexidade de tempo é O(n).

Entretanto, utilizamos duas estruturas (arrays) adicionais F1 e F2, cada um deles de tamanho n para armazenar os subproblemas ótimos encontrados. Logo o algoritmo tem complexidade de espaço O(n).

```
for j in range(1, n):
    keepFirstLineCost = F1[j-1] + S[0][j]
    changeFromSecondLineCost = F2[j-1] + t[1][j-1] + S[0][j]
    if keepFirstLineCost <= changeFromSecondLineCost:
        F1[j] = keepFirstLineCost
        L1[j-1] = 0
    else:
        F1[j] = changeFromSecondLineCost
        L1[j-1] = 1

    keepSecondLineCost = F2[j-1] + S[1][j]
    changeFromFirstLineCost = F1[j-1] + t[0][j-1] + S[1][j]
    if keepSecondLineCost <= changeFromFirstLineCost:
        F2[j] = keepSecondLineCost
        L2[j-1] = 1
    else:
        F2[j] = changeFromFirstLineCost
        L2[j-1] = 0
```

```
if F1[n-1] + x[0] <= F2[n-1] + x[1]:
    finalCost = F1[n-1] + x[0]
    finalL = 0
else:
    finalCost = F2[n-1] + x[1]
    finalL = 1

path = L1 if finalL == 0 else L2

return finalCost, path
```

Etapa 4 - Construção do caminho

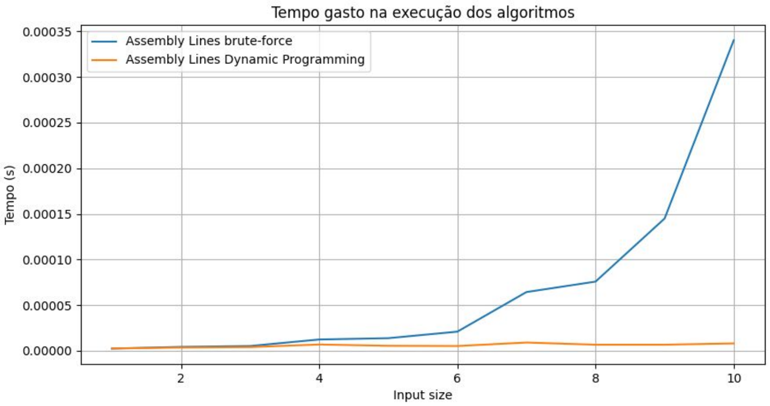
Para construir o caminho vamos salvar em dois arrays quais foram as decisões tomadas em cada passo.

No array L1 teremos as decisões a partir do caminho que terminou na linha 1 e em L2 a partir do caminho que terminou na linha 2

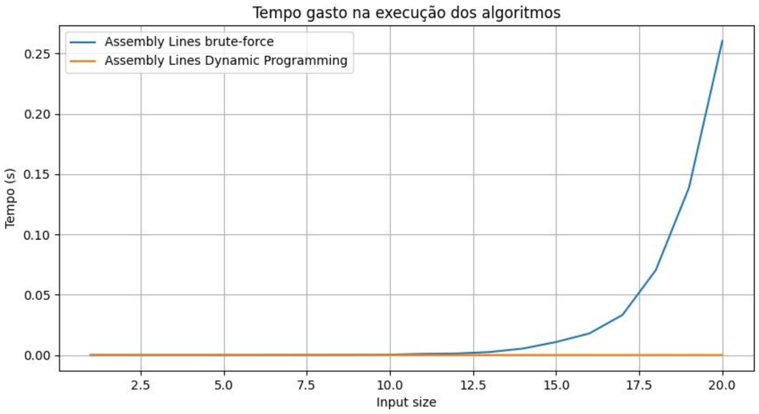
Nos arrays L teremos uma lista de 0 e 1, onde 0 no índice j indica que é melhor realizar a estação j na primeira linha de montagem enquanto 1 indica que é melhor realizar a estação j na segunda linha de montagem.

No fim do algoritmo decidimos entre qual array usar como caminho ótimo dependendo dos valores finais de F1 e F2.

Execuções



10 estações



20 estações

Execuções

35 estações

Estações	Tempo O(n)	Tempo $O(2^n)$
1	2,38E-06	2,38E-06
10	7,39E-06	0,0002937
15	1,40E-05	0,0099918
20	1,53E-05	0,1402859
25	2,31E-05	8,9618437
30	2,24E-05	300,9050838
35	2,1E-05	9980,8314960

Conclusões

- O algoritmo na força bruta com recursividade tem complexidade 2^n, isso acontece pelo recálculo dos subproblemas.
- O algoritmo na força bruta com recursividade rapidamente se torna inviável de ser executado devido a seu alto tempo de execução
- Utilizando a técnica de programação dinâmica temos um algoritmo de complexidade temporal linear
- Entretanto para utilizar a técnica de programação dinâmica aumentamos a quantidade de espaço gasta para salvar os subproblemas que já foram calculados, logo o algoritmo tem complexidade de espaço linear

Referências

- Cormen, T., Leiserson, C., Rivest, R., Stein, C. Algoritmos (Teoria e Prática) - Tradução da 2a edição Americana. 2002;

- https://edisciplinas.usp.br/pluginfile.php/2234590/mod_resource/content/1/ProgDinamica.pdf