Instituto de Computação

# *Organizando um Baile: Solução do Problema da Organização de uma festa*

Erick Grilo
Max Fratane
Matheus Prado
Vitor Santos

# 1 Introdução

O problema consiste em que o professor Stewart foi contradado para prestar um serviço de consultoria para o presidente de uma determinada empresa que deseja realizar uma festa e nós vamos ajudá-lo. A empresa possui uma hierarquia tal que a relação de hierarquia forma uma árvore cuja raiz é o presidente da empresa. O departamento do RH classificou cada um dos empregados com uma classificação de convivialidade (que é um número real). Para a festa ser proveitosa o máximo para todos que forem, o presidente não quer que ambos um funcionário e seu supervisor vão simultaneamente.

Ao professor Stewart, é dada uma árvore onde cada nó da árvore é um funcionário da companhia, que possui um nome e um valor de convivialidade, onde o pai desse nó é o seu supervisor imediato. O objetivo é criar um algoritmo que cria uma lista de convidados que maximiza a soma dos valores de convivialidade dos convidados.

# 2 O Algoritmo

A ideia do algoritmo consiste em solucionar o problema para a árvore de funcionários, com dois casos: caso a árvore tenha somente um nó, e caso a árvore tenha mais de um nó. Dessa forna, a partir do enunciado, temos:

$$Baile(tree) = \begin{cases} \text{convivialidade}(tree), \text{caso tree não tenha filhos} \\ \text{MAX} \begin{cases} \sum \text{Baile}(t), \forall t \in \text{filho}(tree) \\ \text{convivialidade}(raiz) + \sum Baile(t), \forall t \in \text{neto}(tree) \end{cases} \end{cases} \quad (1)$$

Onde o caso base é o caso de *tree* ser uma árvore com um só nó e a relação de recorrência consiste no caso de tree ter filhos: temos que avaliar então, dentre todos os filhos da árvore de entrada, quais serão os nós da árvore que maximizarão o somatório de convivialidade, tomando a precaução de não permitir que um empregado e seu supervisor imediato possam ir ao baile. Nesse caso, na primeira linha da parte do MAX, é avaliado o caso dos filhos de tree (caso o nó da árvore atual vá à festa), e na segunda linha, a convivialidade do nó imediatamente acima de tree e os filhos dos seus filhos, ou seja, o funcionário do funcionário da empresa. Dessa forma, é evitado que um funcionário e seu supervisor imediato possam ir ao baile. A função funcionário retorna o funcionário de um determinado nó, enquanto a função convivi-abiidade retorna o valor de convivialidade de um determinado nó.

Da relação de recorrência apresentada acima, temos o seguinte pseudo-código, que ilustra como o algoritmo funciona:

**Entrada:** tree: uma árvore com os funcionários da empresa
**Saída:** A lista de convidados tais que a soma dos valores de convivialidade de seus elementos seja a máxima possível

```
Resolver(tree):
início
    Baile(tree,lista)
    retorna lista
fim
```

**Algoritmo 1:** RESOLVER

Onde, a função *resolver* retorna a lista desejada, enquanto a função principal responsável por procurar a sequência de valores que maximizam a soma das convivialidades (*baile*) é chamada no seu escopo. Tal função tem seu comportamento descrito a seguir:

**Entrada:** tree: uma árvore com os funcionários da empresa, Lista: lista de convidados, inicialmente vazia
**Saída:** O maior valor de convivialidade, de acordo com o caso
Baile(tree,lista):
**início**
    **se** *tree já foi visitado* **então**
        *lista* ← append(lista,lista(tree))//junta a lista de entrada com a lista armazenada do nó
        **retorna** o valor armazenado em tree
    $visitada(tree) \leftarrow True$//o nó tree é marcado como visitado
    **se** *tree não possui filhos* **então**
        *lista* ← adicionar funcionario(tree)//funcionário(tree): retorna o funcionário desse nó da árvore.
        $listaArmazenada \leftarrow insert(listaArmazenada, lista(tree))$//armazena lista de tree
        $crArmazenado \leftarrow insert(crArmazenado, convivialidade(tree))$
        **retorna** convivialidade(tree)//convivialidade(tree): retorna o valor de convivialidade desse nó da árvore.
    **senão**
        $A \leftarrow 0.0$
        $B \leftarrow$ convivialidade(tree)
        **para cada** *filho* $x \in$ *filho(tree)* **faça**
            $A \leftarrow A + Baile(x, listaA)$//listaA: lista auxiliar
            **para cada** *filho* $y \in$ *filho(x)* **faça**
                $B \leftarrow B + Baile(y, listaB)$//listaB: lista auxiliar
            **fim**
        **fim**
        **se** $B > A$ **então**
            $listaB \leftarrow insert(listaB, funcionário(tree))$
            $listaArmazenada \leftarrow insert(listaArmazenada, listaB)$
            $crArmazenado \leftarrow insert(crArmazenado, B)$
            $lista \leftarrow append(lista, listaB)$
            **retorna** B
        **fim**
        $listaArmazenada \leftarrow insert(listaArmazenada, listaA)$
        $lista \leftarrow append(lista, listaA)$
        $crArmazenado \leftarrow insert(crArmazenado, A)$
        **retorna** A
    **fim**
**fim**

**Algoritmo 2:** Baile

# 3 Exemplo

Suponta que a empresa em questão seja uma pequena empresa de informática, e como estamos em junho, A empresa pediu para usarmos o algoritmo para resolvermos o problema em questão referente à uma festa junina. Dessa forma, se um funcionário for, nem seu subordinado imediato nem o seu superior imediato poderão ir. A árvore abaixo ilustra a organização dos funcionários dessa empresa:

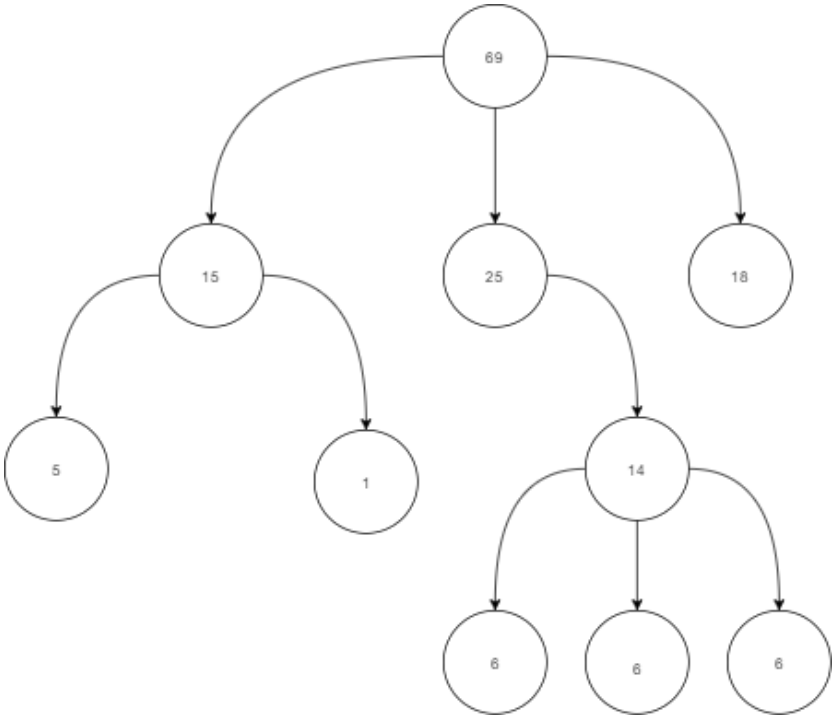


Figura 1: Exemplo de uma árvore que representa funcionários de uma empresa.

Ao se executar o algoritmo, temos que o mesmo verificará que a árvore de entrada não é uma árvore sem filhos. Logo, ele segue o segundo caso descrito na relação de recorrência, que verifica o máximo dos valores entre todos os filhos de *tree* ou o valor de convivialidade da raiz com o dos netos (assim, respeitando a restrição).

Ao entrar no caso de *tree* (a raiz) ter filhos, para cada filho de *tree*, a função *baile* é chamada recursivamente a fim de armazenar o valor de convivialidade de cada um dos filhos de *tree* em uma variável A qualquer. Em seguida, o mesmo ocorre, só que agora para os netos de *tree*, armazenando a soma dos valores em B (que já recebera o valor de convivialidade de *tree*). Ou seja, Quando $Baile(69)$ é chamada, temos que avaliar o máximo entre ($Baile(15)$ + ($Baile(21)$ + $Baile(18)$ e (69 + $Baile(5)$ + $Baile(11)$ $baile(14)$), que são respectivamente o somatório dos valores do filho da raiz com o seu valor mais o valor de seus netos.

O algoritmo então verifica primeiro para cada um dos filhos da raiz: chama recursivamente $Baile(15)$, que por sua vez chama recursivamente *baile* para seus filhos, $baile(5)$ e $baile(1)$. Como $baile(1)$ e $baile(5)$ são folhas, o seu valor de convivialidade é retornado. Dessa chamada recursiva, é retornado o máximo entre 15 (o valor da raiz da chamada) e a soma de seus filhos (5+1). O valor retornado é 15

Em seguida, o algoritmo chama *baile* para o outro filho da raiz, o nó 25, que por sua vez chama recursivamente *baile* para seu único filho, 14, que por sua vez, chama *baile* para cada um de seus filhos (6,6 e 6). Como esses três nós são folhas, os seus valores de convivialidade são retornados. Dessa chamada recursiva, é retornado o máximo entre 14 e 18 (6+6+6). Logo, o valor 18 é retornado. Dessa forma, para

a chamada do nó 25, retorna o valor da raiz da chamada (25) somado com o somatório dos seus netos (18), que é justamente o segundo caso da relação de recorrência. porém, o valor da raiz da árvore é maior que 25. Logo, o valor final dessa chamada acaba por ser 69+ 18.

Por último, o último filho da raiz é visitado: *baile*(18) é chamado recursivamente, que, como não possui filhos, acaba por retornar o valor de convivialidade desse nó (o próprio 18).Logo, temos:
n1 = 15+25+18 e n2 = 5 + 1 + 69 + 6+6+6. *baile*(69) = MAX(58,93). = 93. Logo, os nós que vão compor a lista de convidados são os funcionários que são representados por esses nós, cuja soma maximizou o valor de convivialidade respeitando a restrição de que, se um convidado vai, seu supervisor imediato não vai (e vice-versa).

# 4 Prova da Corretude do Algoritmo

A função que resolve o problema é dada pelo pseudocódigo definido em 1 (*resolver*). Porém, a função que efetivamente resolve e processa os dados para solucionar o problema é a 2 (*Baile)*. Dessa forma, ao provarmos a corretude da função *Baile*, provamos a corretude de toda o algortimo que soluciona o problema.

Seja *Baile(T)*: A chamada da função *Baile* tal como descrita na relação de recorrência na seção 2 ou seja, *Baile*(T) retorna o valor máximo do somatório das convivialidades dos seus nós. A prova será feita por indução forte estrutural na árvore de entrada para o problema.

**(i) Base da indução:** *Baile*($T_0$) é verdadeira?
Seja *Baile*($T_0$) : O valor máximo da soma de convivialidade da árvore que possui apenas um nó (sem filhos). Como só existe um único nó a ser avaliado, temos que o seu valor de convivialidade (por definição) é o que maximiza o somatório dos valores de convivialidade da árvore. Portanto, a lista retornada por *Resolver* só possui o funcionário representado por esse nó como um convidado. Logo, temos que *Baile*($T_0$) vale.

**(ii) Hipótese de Indução:** Sejam $T_1$,$T_2$,$T_3$,...,$T_n$ sub-árvores formadas a partir da raiz (árvore de entrada para o problema) cujas raízes estão no k-ésimo nível, para k = 1, 2, ..., n. Logo, supor que *Baile*($T_k$) vale, para k = 1, 2, ..., n. Por causa dessa suposição, temos que a relação de recorrência também passa a valer para sub-árvores de $T_k$, ou seja, árvores $T_{k_q}$ que são sub-árvores de uma $T_k$, com q = 1,2,...,m.

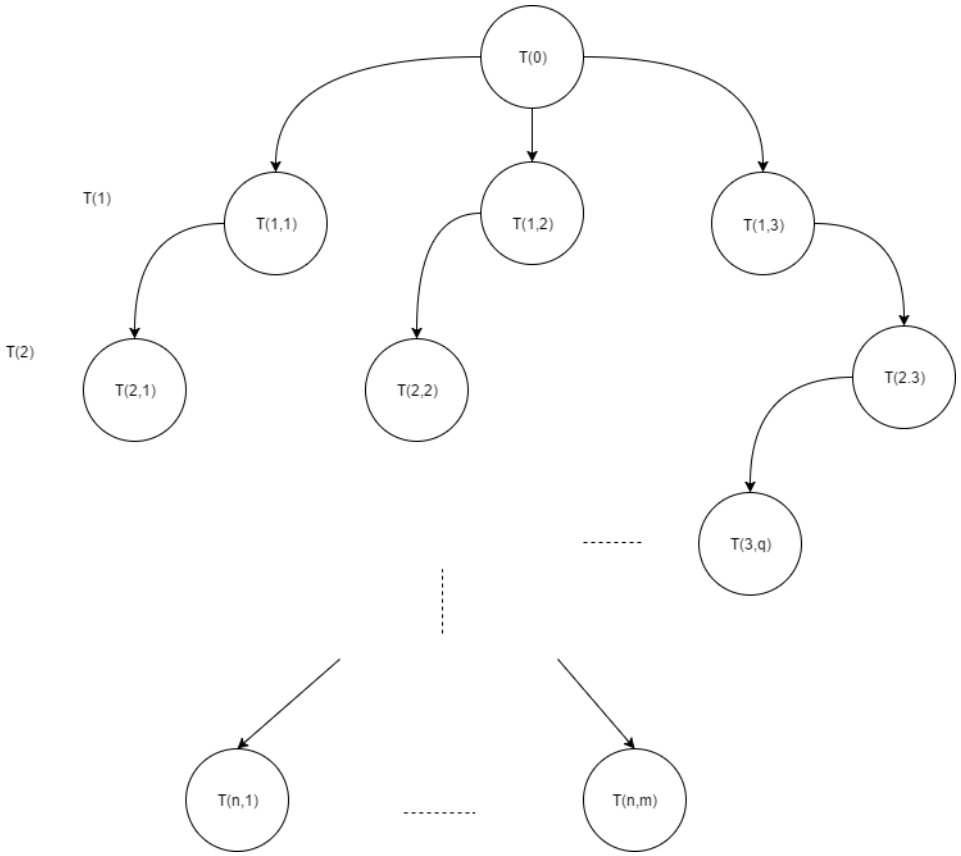


Figura 2: Exemplo de uma árvore com até n níveis e m filhos, onde o limite de m pode variar de nível em nível, uma vez que a árvore é n-ária

**(iii) Passo Indutivo:** Provar válido para qualquer árvore do nível *Baile*($T_{n+1}$), uma $T_{n+1_q}$ qualquer, q = 1,2,...,m.
Ao entrarmos nesse passo, fixemos para uma $T_{n+1_q}$ qualquer tal como na árvore acima, sem perda de

generalidade. A partir daí, temos: pela hipótese de indução, *Baile*($T_k$) é o valor máximo da convivialidade para uma sub-árvore do nível k, $T_k$, para k = k+1,k+2,...,n. Dessa forma, temos que *Baile*($T_k$) é o valor máximo do somatório do valor de convivialidade de $T_k$. Ao considerarmos que *Baile*($T_n$), vale (pela hipótese de indução), se *Baile*($T_n$) possui filhos, chegamos em *Baile*($T_{n+1}$) recursivamente. Logo, temos que analisar dois casos distintos:

**I)** Uma árvore qualquer do nível $T_{n+1}$, a $T_{n+1_q}$ que estamos analisando não possui filhos : pela base da indução, nota-se que o valor de convivalidade máximo dessa árvore, é o seu próprio valor de convivialidade.

**II)**Uma árvore qualquer do nível $T_{n+1}$ possui filhos:

Seja $n_1 = \sum Baile(t), \forall t \in$ filho($T_{n+1_q}$). Pela hipótese de indução, temos que *Baile*($T_k$) vale, para k = 1,2,..., n. Como extendemos isso para qualquer filho de $T_k$ (logo, para qualquer filho de $T_n$), vale também para o caso de $T_{n+1_q}$ (que é alcançado recursivamente na chamada *Baile*($T_n$), onde $T_{n+1}$ é filha de $T_n$. Portanto, n1 é o somatório dos valores dos filhos de uma árvore qualquer do nível $T_{n+1}$, a $T_{n+1_q}$ genérica introduzida no passo indutivo.

Agora, seja $n_2$ = convivialidade(raiz) + $\sum Baile(t), \forall t \in$ neto($T_{n+1}$). Para o caso de $T_{n+1}$, faz-se a raiz sendo a própria $T_{n+1}$ que é raiz da $T_{n+1_q}$. Como no parágrafo acima, vimos que vale para um filho de uma árvore no nível de $T_{n+1}$ (por valer para qualquer neto de qualquer árvore de $T_n$, pela hipótese), vale para a árvore que fixamos, $T_{n+1_q}$. Logo, se a raiz nesse caso é a árvore pai da árvore do nível $T_{n+1}$ fixada, a $T_{n+1_q}$, n2 é o valor do somatório da convivialidade de $T_{n+1_q}$ com os seus netos.

Logo, por $n_1$ e $n_2$, o valor de convivialidade máximo da árvore original que foi dada como entrada para o problema (T), pois como *Baile*(T) vale para qualquer sub-árvore até o n-ésimo nível (hipótese de indução), pode ser encontrada pela seguinte fórmula:
*Baile(T) = Baile($T_n$) + Baile($T_{n+1}$)*, onde *Baile($T_n$)* é o baile para todas as sub-árvores de T (a árvore original), que pela hipótese de indução,vale.

Porém, podemos reescrever *Baile($T_{n+1}$)* como sendo:

$$\textbf{\textit{Baile(}}\boldsymbol{T_{n+1}}\textbf{\textit{)}} = \textbf{convivialidadeMaxima}(\mathbf{T_{n+1_q}}) = \textbf{MAX(n1,n2)},$$
$\forall T_{n+1_q}$ filhas de qualquer árvore do nível $T_{n+1}$.

Temos que a função acima culmina justamente na relação de recorrência que foi encontrada. Ao provarmos, sem perda de generalidade, que *Baile* vale para uma *Baile($T_{n+1_q}$* qualquer, generalizamos acima novamente. Como MAX retorna o valor máximo entre dois valores, temos que convivialidadeMaxima retorna o valor máximo dentre os possíveis valores máximos que $T_{n+1_q}$ pode ter, não retornando um resultado inválido para a solução do problema da árvore de entrada T. Logo, para qualquer sub-árvore de $T_{n+1_q}$,*Baile($T_{n+1}$)* vale.

# 5 Prova da Complexidade do algoritmo

Após uma série de testes, a primeira versão do algoritmo (sem o uso da noção de programação dinâmica, nesse caso, verificando se o algoritmo já avaliou o valor de convivialidade do nó), havíamos chegado em uma suposição de que o algoritmo parecia estar com a complexidade em $O(n^2)$. Após uma segunda versão do algoritmo ter sido gerada (com a noção de marcar o nó que já foi visitado), chegamos, através de testes e medição de quantidade de nós visitados à complexidade de $O(n)$.

Pelo método da suposição e verificação, temos que a complexidade de *Baile* está em $O(n)$. A prova será feita por indução em n. Queremos provar que a complexidade de Baile é $O(n)$.

**(i) Base da indução:** *Baile* com a árvore de de tamanho 1 é $O(1)$ ?
Seja P(n): Baile($T_n$) possui complexidade n. Como só existe um único nó a ser avaliado, temos que o número de iterações que o algoritmo faz é apenas 1. Portanto, a compexidade do algoritmo nesse caso é $O(1)$, com n = 1.

**(ii) Hipótese de Indução:** Supor válido P(k), para $1 \leq k \leq n$ (indução forte).

**(iii) Passo Indutivo:** Provar válido P(n+1); ou seja, para uma árvore de entrada que possui n + 1 nós, a complexidade de *Baile*($T_{n+1}$) é $O(n+1) = O(n)$.
A partir do algoritmo, é possível deduzir:
1 chamada para *Baile(filho(n+1))*;
p chamadas de *Baile(neto(n+1))*. Logo, como *Baile(filho(n+1))* visita recursivamente *Baile(neto(n+1))*,temos: *Baile(filho(n+1))* $\in O(p)$ e *Baile(neto(n+1))* $\in O(n)$. Logo, como $(O(p) + O(n)) = O(n)$ (pois n ¿ p),*Baile(n+1)* $\in O(n)$.