Proposta Trabalho Final Reconhecimentos de Padrões

Paulo Cirino Ribeiro Neto

15 de dezembro de 2015

1 Introdução

O Departamento de Nutrição pretende promover um evento sobre alimentação saudável e precisa da ajuda do DCC para automatizar o processo de criar uma refeição balanceada. Assim o DCC ficou a cargo de criar um programa que recebe uma lista de valores calóricos de alimentos, e o valor calórico total de uma refeição, e dizer se é ou não possível criar uma combinação desses alimentos de forma a fazer uma refeição saudável.

1.1 Exemplo de Entrada

Lais precisa consumir 500 Calorias no café da manhã, e ela em sua casa têm os seguintes alimentos :

- Banana - 200 Calorias
- Copo de Leite - 150 Calorias
- Bacon - 490 Calorias
- Maça - 80 Calorias
- Bolacha - 70 Calorias
- Suco - 50 Calorias

Nesse caso o programa receberia como entrada o valor ideal do café da manhã e o valor de cada alimento, e retornaria que é "sim"possível combinar esses alimentos, de forma a selecionar a Banana, o Leite, a Maça e o Suco, completando exatamente 500 Calorias que é valor calórico desejado. No caso se o valor total de calorias do café da manhã fossem 450, o programa deveria retornar que "não"é possível combinar esses elementos de forma a encontrar a refeição ideal.

2 Formalização de Problema de Decisão e Prova NP-Completo

Temos que esse problema é exatamento o problema classico **Soma dos Sub-Conjuntos** ou em inglês *Subset Sum*, onde que o nosso subconjunto $S = \{s1, s2, s3, ..., sn\}$ é compostos pelo conjunto dos valores calóricos dos alimentos de entrada, e queremos um conjunto $S_K = \{s_k1, s_k2, ..., s_km\}$ com $m \leq n$ onde a soma de todos os elementos de S_K é igual a um valor fornecido K. Assim se tal conjunto existe, a decisão é sim, e se não existe a resposta é não.

2.1 Prova que o problema é NP-Completo

Para provar que um problema é **NP-Completo**, iniciamos provando que ele é **NP**.

2.1.1 Prova que é NP

Para provar que o nosso problema é **NP**, apresentamos abaixo um algorítimo não-determinista que pode solucionar o problema em uma máquina teórica de Turing em tempo polinomial:

Algoritmo 1: Resolve *Subset Sum* não-determinista

```
Entrada: S, K
Saída: Sim ou Não
inicio
    S_K = EscolheSolucao(S, K)
    if soma(S_K) == K then
        Return Sim
    end
    else
        Return Nao
    end
    ,
fin
```

2.1.2 Redução do 3-SAT para SubSet-Sum

Agora, para provarmos que o problema é de fato **NP-Completo**, temos que reduzir um outro problema que já sabemos que é pertencente a classe **NP-Completo** para o problema que queremos provar.

Alem disso iremos utilizar a metodologia padrão da prova que o **SubSet Sum** é NP completo, mais especificadamente iremos utilizar o método utilizado por Pilu Crescenzi and Viggo Kann, professores da University of Florence, em suas notas de aula de 2011.

Para isso iremos utilizar um problema que é o **3-SAT** onde já sabemos de suas propriedades para fazer essa prova.

Consideramos inicialmente a formula 3CNF com as variáveis $x_1, ..., x_n$ e as clausulas $c_1, ..., c_m$. Além disso consideramos também que nenhuma clausula possui uma variáveis x_i e sua negação $\neg x_1$, e que cada variável x_i deve aparecer em pelo menos uma clausula.

Temos que nossa redução criará 2 números x_i e y_i em uma tabela, para cada uma das clausulas c_j, existirá também 2 números f_j e t_j.

Temos que os valores f_j e t_j para posição j é sempre 1, e que para $n+1 \leq j \leq n+m$, o elemento de posição j de t é igual a 1 se $x_i \in c_(j-n)$. E para os mesmos valores o elemento de posição j de f é igual a 1 se $\neg x_i \in c_(j-n)$. Todos os outros valores são iguais a 0.

Ja para a construção dos valores x_i e y_i de cada clausula c_j, fazemos todo valor x_i e y_i igual a 1 para as posições $(n+j)$, e todos os outros elementos como 0.

Finalmente construímos então o somatório S de $(n+m)$ digitos de forma que, para todos os elementos $1 \neq j \neq n$ o digito de posição j de S é igual a 1, já para todos os outros elementos o digito de posição j é igual a 3.

Após a contrução de todas as informações fazemos então a seleção delas de forma que para todo x_i que é verdadeiro escolhemos o valor t_i e para todo valor x_i que é falso escolhemos o valor f_i. Agora em relação a escolha dos valores $x_i e y_i$ escolhemos y_i se a quantidade de verdadeiros em c_j é igual a 1, caso for maior que 1 escolhemos x_i.

Para demonstrar satisfazibilidade da redução fazemos x_1 verdadeiro se t_i está no subconjunto, f_i se não. Se exatamente 1 número por variável está no subconjunto a condição está satisfeita.

Consideramos então o exemplo abaixo : *3-SAT* $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee \neg x_3) \wedge (x_1 \vee \neg x_2 \vee x_3)$

A partir desse exemplo podemos construir a seguinte tabela de valores f_j e t_j:

	i	i	i	j	j	j	j
Numeros	1	2	3	1	2	3	4
t1	1	0	0	1	0	0	1
f1	1	0	0	0	1	1	0
t2	0	1	0	1	0	1	0
f2	0	1	0	0	1	0	1
t3	0	0	1	1	1	0	1
f3	0	0	1	0	0	1	0

Construímos então a tabela de valores x_i e y_i :

	i	i	i	j	j	j	j
Numeros	1	2	3	1	2	3	4
x1	0	0	0	1	0	0	0
y1	0	0	0	1	0	0	0
x2	0	0	0	0	1	0	0
y2	0	0	0	0	1	0	0
x3	0	0	0	0	0	1	0
y3	0	0	0	0	0	1	0
x4	0	0	0	0	0	0	1
y4	0	0	0	0	0	0	1
s	1	1	1	3	3	3	3

Fando agora a condição de satisfazibilidade para construção onde todas as variáveis são verdadeiras :

	i	i	i	j	j	j	j
Numeros	1	2	3	1	2	3	4
t1	1	0	0	1	0	0	1
t1	0	1	0	1	0	1	0
t2	0	0	1	1	1	0	1
x2	0	0	0	0	1	0	0
y2	0	0	0	0	1	0	0
x3	0	0	0	0	0	1	0
y3	0	0	0	0	0	1	0
x4	0	0	0	0	0	0	1
s	1	1	1	3	3	3	3

Temos então que a condição está satisfeita.

3 Modelagem e solução do Problema

O problema foi modelado baseado na ideia de que para cada alimento na entrada existem 2 possibilidades, incluir ou não incluir ele na dieta. Assim o trabalho foi resolvido de forma que a cada nova entrada, para cada uma das instâncias existentes até o momento são criadas 2 novas instâncias, uma onde alimento analisado é escolhido e outra onde ele não é escolhido, assim como representa a imagem abaixo:

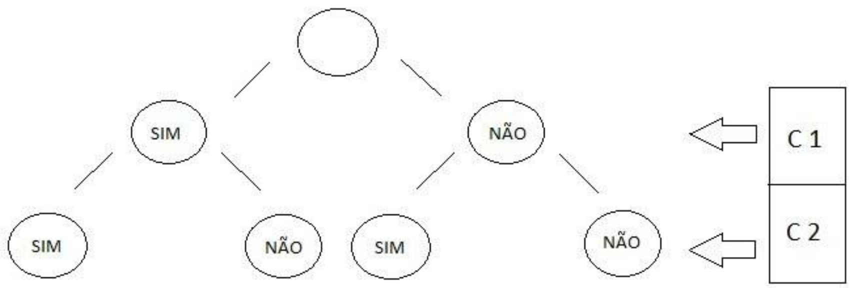

Figura 1: Modelo 01 - Modelagem do Problema

Essa imagem referencia a ideia do funcionamento do programa de forma que, c_1 e c_2 são o primeiro e segundo alimento da lista de alimentos, e cada circulo é uma possibilidade de combinação dos mesmos, de forma geral as folhas da árvore representam todos as combinações de valores.

Essa árvore é construída recursivamente, e a oportunidade de paralelização é dado nó da arvore, para os 2 caminhos que ele pode seguir, escolher ou não o próximo alimento, é criado uma nova thread que segue em uma das possibilidades a thread pai continua em um dos outros caminhos.

O Pseudo códigos abaixo representam o funcionamento dos algorítimos utilizados na resolução do problema:

Algoritmo 2: VOID SPAWNTREE

Entrada: $Arvore- > No, numThreads, maxNumThreads$
inicio
    void GrowTree(Arvore-¿no);
    if $numThreads < maxNumThreads$ **then**
        numThreads ++; void GrowTree($Arvore- > no$);
        NewThread($SpwanTree(Arvore- > no- > sim)$) ;
        SpawnTree($Arvore- > No- > nao$);
    end
    else
        SpwanTree($Arvore- > no- > sim$) ;
        SpawnTree($Arvore- > No- > nao$) ;
    end
fin

Algoritmo 3: VOID GROWTREE

Entrada: $Arvore->No, VetComidas$

inicio

$No->sim->totalCalorias = No->totalCalorias + VetComidas[No->pos+1]$;

$No->sim->pos = No->pos+1$;

$No->nao->totalCalorias = No->totalCalorias$;

$No->nao->pos = No->pos+1$;

fin

Assim o funcionamento do programa é basicamente a chamada da função **void SpawnTree()** até alguma condição de parada, que alem do termino do programa pode ser entrar em uma instância impossível.

4 Análise Teórica de Custo Assintótico

Assim como todos os problemas **NP-Completo**, a complexidade da solução do problema é exponencial no pior caso.

Temos em particular que para a solução apresentada a complexidade tanto espacial quanto temporal são da ordem $O(2^n)$ no pior caso. No caso médio que não sera demonstrado aqui essa complexidade melhora bastante, mas mantendo a proporção espacial e temporal igual.

4.1 Análise Teórica do Custo Assintótico de Tempo

Abaixo é mostrado na tabela 1 todas a funções do programa com suas devidas complexidades assintóticas de pior caso:

	Função	Complexidade
1	void growTree()	O(n)
2	leaf addNode()	O(1)
3	tree initTree()	O(n)
4	void LiberaFolhas()	O(2^n)
5	void liberaArvore()	O(2^n)
6	void * NewSpawnTree()	O(2^n)
7	void computate()	O(2^n)

Tabela 1: Tabela 1 - Complexidade Assintótica de Tempo

Temos que as funções *void * NewSpawnTree()* e *void LiberaFolhas()* são funções com complexidade exponencial, o motivo delas terem tal complexidade é o fato de que passam, no pior caso, em todos 2^n nós que representam as 2^n possíveis combinações de sub-conjuntos.

As demais funções que têm complexidade exponencial são porque chamam essas funções.

4.2 Análise Teórica do Custo Assintótico de Espaço

Temos que na solução proposta, para cada uma das instâncias testadas do problema é criada uma nova folha da árvore, assim como existem 2^n instâncias, a complexidade teórica é $O(2^n)$. Existem para o mesmo algorítimo uma solução mais elegante, que necessita apenas de um número de folhas igual ao número de threads em funcionamento, só que tal solução requer um nível superior de sincronização, que ainda não tenho. Essa outra solução funciona de forma que quando se passa por uma instância, ao sair dela ela é liberada.

5 Análise de Experimentos

Foram feitos vários experimentos no sentido de saber testar o funcionamento do programa, o que foi descoberto é que existe instabilidade quando o programa entra no pior caso com $N \neg 20$, o motivo disso é devido a grande quantidade de memória que o programa necessita. Fora essa situação o programa funciona como o esperado, dando em todos os casos testado a resposta correta. Alem disso o algorítimo não necessariamente entra no pior caso sempre, isso ocorre muito dificilmente no caso real, assim a estabilidade do algorítimo se estende muito alem de $N = 20$. Os testes que decidi mostrar são fundamentados em 2 importantes perguntas :

- Qual é a proporção entre o tempo gasto pelo programa e o número de entradas?
- Qual é a melhora no tempo gasto quando aumentamos o número de threads?

A resposta a essas 2 perguntas rederam 2 gráficos que seguem abaixo.

5.1 Testando a relação entre o tamanho da entrada e o tempo de processamento gasto

Esse teste foi feito baseando-se na primeira pergunta. A forma que encontrei de responde-la foi fixando um valor máximo de threads a uma constante, e variando o tamanho da entrada. No teste representado pelo gráfico abaixo em específico, o número de threads máximo que o programa pode utilizar é de 256, e o tamanho das entradas é o conjunto $N = \{2, 4, 6, 8, 10, 12, 14, 16, 18\}$.

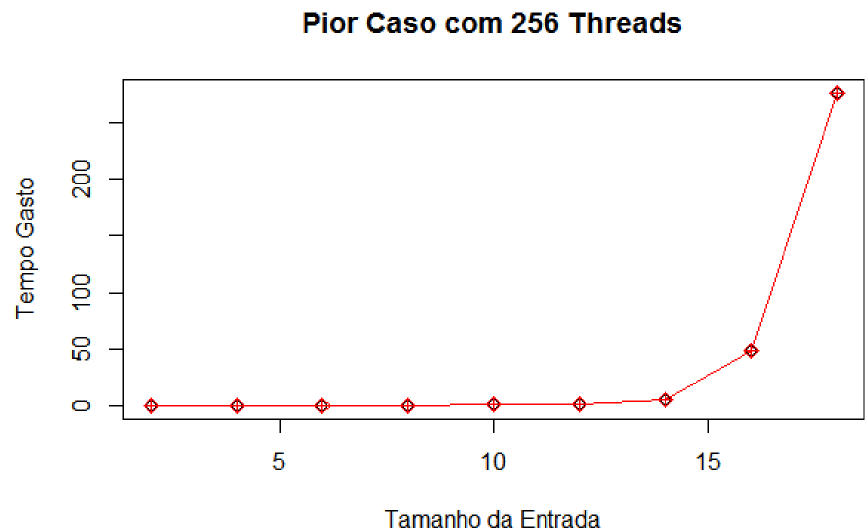

Figura 2: Plot 02 - Pior Caso

Por meio desse gráfico fica fácil perceber, principalmente com $N = \{14, 16, 18\}$ a característica exponencial do problema, onde que um aumento de 4 em N fez com que o tempo gasto se elevasse em mais de 10 vezes. Pelo gráfico parece que o tempo é linear até $N = 14$, mas isso não é verdade, oque realmente aconte é que a taxa de crescimento aumenta tanto para valores maiores de N, que a "impressão "é que o aumento de tempo para valores pequenos é irrisório.

5.2 Testando a relação entre o número de threads e o tempo de processamento gasto

Esse teste foi desenvolvido para responder a segunda pergunta, que é, colocada de uma outra forma, "O que o programa ganha com a paralelização ?".

Para responder essa pergunta metodologia de teste foi escolher apenas uma entrada, e para a mesma entrada testar quantidade de threads diferentes.

Segue abaixo o gráfico obtido com esses testes:

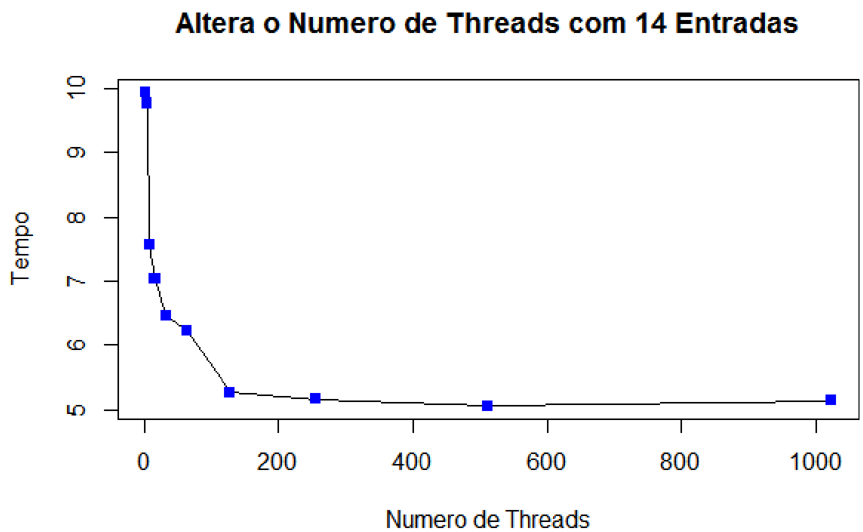

Figura 3: Plot 01 - Altera Número de Plots

Por esse gráfico fica evidente a vantagem da paralelização, ele mostra que o tempo gasto pelo programa quando aumentamos o número de threads é realmente menor que o tempo gasta com poucas. De forma geral o que pude perceber é que o limite do tempo de execução ocorre quando número de threads é próximo a $T >= N^2$ e têm como limite superior valores $T <= 2^N$. Esses valores são completamente empíricos, e baseado apenas nos testes feitos por mim. Não houveram cálculos para dar suporte a esses valores, apenas mera observação.

6 Conclusão

Conclui que problemas **NP-Completo** são difíceis de lidar, isso ocorre porque não importa a forma que tentamos,a solução sempre será exponencial. O que pode ser feito são métodos que restringem todas as instâncias impossíveis, isso é, se existe uma instância de um problema que torna a solução impossível, todas as instâncias que dependem dela não são calculadas e já são automaticamente descartadas.

O problema dessa abordagem, que é necessária para tornar a solução mais palpável, é que existem demasiadas restrições ao problema, gerando assim grande instabilidade e complexidade de programação.

Alem disso foi possível perceber através desse trabalho que a ferramenta de paralelização é extremamente poderosa desde que usada com bom senso e precisão. Em muitas situações, como a que me ocorreu ao longo do trabalho, é muito difícil lidar com todas as peculiaridades de sincronização.

Ou seja, threads são úteis mas, assim como outras ferramentas na programação têm que serem utilizadas com precisão cirúrgica para funcionarem a todo potencial.