Documentação do Trabalho Prático 3
Alimentação Saudável

Caíque Bruno Fortunato

15 de Dezembro de 2015

1 Introdução

Esse trabalho tem como principal objetivo fixar os conceitos aprendidos no módulo de Programação Paralela e NP-Completo na disciplina Algoritmos e Estruturas de DADOS III praticando os conceitos na modelagem de um problema chamado Alimentação Saudável.

O problema consiste em ajudar o Departamento de Nutrição que está promovendo um evento sobre alimentação saudável. Centenas de pessoas irão participar do evento e o DN irá criar planos alimentares para cada uma dessas pessoas. Para otimizar o tempo e os cálculos, um programa que realiza o processo em questão seria muito bem-vindo, útil e prático. Com a ajuda da computação o departamento poderia atender e ajudar em menor tempo mais pessoas.

O objetivo do problema é definir se é possível ou não criar uma dieta balanceada entre as opções dos valores calóricos dos alimentos disponíveis dada o número exato de calorias que devem ser consumidas. Caso seja possível é gerada uma saída "sim" e caso contrário uma saída "não". Podem existir valores repetidos na entrada, no entanto não será possível escolher o mesmo elemento mais de uma vez.

O problema da Alimentação Saudável tem como base principal o algoritmo de enumeração, isso porque no pior caso serão listadas todas os subconjuntos e suas somas para encontrar a soma desejada. Ou, ainda, os subconjuntos são gerados até encontrar a soma necessária, utilizando sua vez uma restrição no algoritmo em questão.

A enumeração é muito importante em diversas áreas, como a matemática, estatística, computação e até mesmo na nutrição, como é o exemplo que será estudado. Isso porque em muitos casos, para encontrar o resultado desejado, caso exista, é preciso enumerar todas as opções e ter um algoritmo que resolva isso é de grande ajuda.

No entanto, infelizmente algoritmos de enumeração costumam consumir uma grande quantidade de tempo, principalmente se o número de objetos for muito grande, o que pode levar uma eternidade para obter o resultado desejado. Veremos então o custo do problema da Alimentação Saudável e o tempo que ele pode levar para ser executado, que pode ser exponencial em alguns casos.

A fim de tentar otimizar o tempo é possível realizar algumas podas no cálculo e realizar a paralelização, que serão apresentadas nesse trabalho. Mas, mesmo com os recursos para melhorar o tempo, nem todas as entradas podem ter resultados em tempo polinomial, o que será provado, ou seja, que o problema em questão é NP-Completo.

2 Modelagem do Problema

A força bruta, ou busca exaustiva, é uma técnica de algoritmo utilizada para resolver problemas ao enumerar todos os candidatos possíveis da entrada fornecida a fim de checar cada candidato para saber se ele satisfaz a solução desejada. Logo, o Força Bruta é um algoritmo possível para a modelagem de nosso problema.

Dado um vetor com os conjuntos dos valores calóricos dos alimentos disponíveis, serão testados exaustivamente os conjuntos da soma dessas calorias até que seja obtido o valor da soma da caloria desejada, caso exista. Ou seja, caso seja encontrada a soma, o algoritmo irá retornar "sim" e caso contrário, "não", utilizando o algoritmo de força bruta através da recursão.

Como um exemplo simples para entendimento: suponhamos que queremos encontrar a soma 22 no conjunto 3 e 18. Então, será gerada a seguinte árvore vinda da recursão:

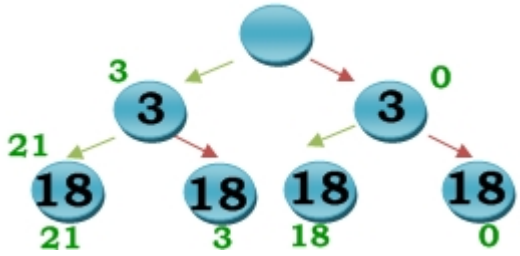

Figure 1: Árvore de recursão

Como nenhum dos resultados obtiveram resposta 22, não existe nenhum subconjunto com a soma 22 entre os valores calóricos dos alimentos disponíveis. O algoritmo então retornará "não". De maneira análoga, caso houvesse 22 como resposta o problema teria solução e retornaria "sim".

É interessante pensar que caso a soma for maior do que o desejado o algoritmo não precisa continuar realizando o cálculo e se o resultado for encontrado os cálculos também podem ser interrompidos, otimizando o tempo ao mesmo tempo que existe a poda. Vale ressaltar que a realização da poda é muito importante e crucial para a economia de tempo em problemas de decisão sim/não e, consequentemente, NP-Completo, já que testando todas as possibilidades possíveis desnecessariamente o tempo a ler levado é ainda maior para a execução da mesma solução apenas com o tempo preciso.

Para realizar o procedimento, foi realizada uma função recursiva que possui como cálculo principal as duas possibilidades existentes na árvore: somar ou não somar (seta verde e seta vermelha), respectivamente.

Caso seja somado, é calculado:

enumerationalgorithm(index - 1, partialsum + vector[index]);

Caso não seja somado, é calculado:

enumerationalgorithm(index - 1, partialsum);

As funções acima foram utilizadas como um esboço para o entendimento de como o cálculo é realizado. No código foram realizadas as alterações necessárias para que a paralelização fosse possível, como será discutido em seguida.

Vale ressaltar que o funcionamento do algoritmo recursivo está também com as restrições já discutidas anteriormente. Na figura 1, temos um exemplo de pior caso, onde todas os cálculos da soma serão realizados, nenhuma soma parcial extrapola a soma desejada e o resultado não é encontrado. Além disso, o vetor também foi ordenado, otimizando também os cálculos que serão realizados na recursão a fim de diminuir o número de contas e, consequentemente, de verificações.

Com apenas esses passos, o problema é resolvido para pequenos casos em tempo polinomial, em contrapartida possui resposta exponencial para casos maiores. Para tentar otimizar um pouco mais, podemos aplicar a programação paralela, com o uso das threads.

2.1 Estratégia de paralelização

A primeira ideia de paralelização foi criar as threads dentro da função recursiva de força bruta, o que não deu certo devido a concorrência que foi gerada e o nível de abstração e dificuldade. Embora tenha passado para alguns casos de testes, em algumas tentativas acontecia falha de segmentação e o uso da paralelização ficou quase sequencial e um pouco complexa.

Assim, foi criado uma função própria de paralização, chamada Parallel(). Nela usamos uma função para a criação das threads e também a pthread_join que suspenderá a execução da thread até que a outra termine. Caso achemos o valor da soma desejado o algoritmo será finalizado, retornando a resposta afirmativa.

Nessa função temos ainda um ciclo de repetição que irá ocorrer desde j até o tamanho do vetor (quantidade de números do conjunto) somado com a quanidade de threads fornecida pelo usuário. A incrementação irá ocorrer somado com o número de threads.

for(j = 0 to j menor que vectorlen + quant)
if (acha o resultado) para;
limit = j é menor que o vectorlen?

Se sim limit = j % quant
se não limit = quant

tendo então o valor de limit, calcularemos outra estrutura de repetição que realizará o procedimento criar as threads necessárias após inicializar os valores dos índices e da soma.

Feito isso, o algoritmo de força bruta já explicado é executado com a paralização desejada, dado o número de threads. Para que isso ocorra, é chamada uma função auxiliar que vai atribuir um valor de parâmetro para a função de força bruta funcionar corretamente com a paralização, isto é, rezlizando as restrições restantes e fornecendo os valores de parâmetro.

Assim, o algoritmo é paralelizado de acordo com o número de threads fornecidas e desejadas pelo usuário em tempo relativamente menor para casos de testes com tamanho não muito grande.

3 Prova de que o problema é NP-Completo

Como já foi discutido, é impossível encontrar uma solução em tempo polinomial com grandes entradas para esse problema. Ou seja, dependendo da entrada, o programa poderá ser executado por exaustivos minutos, horas, dias, meses, anos, décadas e assim sucessivamente. A explicação para esse fenômeno é explicado pelo motivo do problema da enumeração estar na classe de NP-Completo.

A percepção é clara, já que o problema é caracterizado por resultado da computação de fornecer uma resposta sim ou não. Logo, inicialmente precisamos mostrar que o "Problema da Alimentação Saudável" está na classe NP, ou seja, possui uma solução que pode ser verificada facilmente em tempo polinomial.

Vale lembrar que temos então uma tomada de decisão, isto é, se existe ou não um subconjunto com a soma desejada. Temos uma solução de 2^n para o número de n entradas contendo todas as soluções possíveis.

Para isso, será apresentado um algoritmo determinista polinomial que verifica Alimentação Saudável:

```
verificaDAlimentacaoSaudavel(vetorsolucao, soma, tam)
if (vetorsolucao == 0 e soma != 0) return FALSE;
for(i = 0 to —tam—)
if(soma == vetorsolucao[i]) return TRUE;
return FALSE;
```

Como o algoritmo apresentado tem complexidade O(n), onde n é o tamanho do vetor, é possível observar que ele é executado em tempo polinomial verificando se existe solução. Ou seja, a função verifica recebe o vetor das $(2)^n$ combinações de resultados possíveis e verifica se existe ou não a solução desejada, ou seja, a soma a ser encontrada.

Agora, é preciso escolher um problema conhecido Y que é NP-Completo. Sendo X o problema da Alimentação Saudável, temos que reduzir $Y \leq X$. A

redução precisa considerar a transformação de qualquer entrada do problema Y em alguma entrada do problema X.

Consideramos que Y é o problema do 3CNF SAT, que é um problema NP-Completo conhecido. O problema da satisfazibilidade booleana é o problema de determinar se existe uma determinada valoração para as variáveis de uma determinada fórmula booleana tal que essa valoração satisfaça esta fórmula em questão.

Voltando ao problema Alimentação Saudável, seja uma instância genérica dada por: S = C_1 * C_2 * ... * C_n, onde $C_i = (I_1^i + I_2^i + I_3^i)$ onde cada C seja a opção de somar ou não dados a partir de três números diferentes. Ou seja, se considerarmos C_1 ou C_2, por exemplo, sendo que C_1 é somar $I_1^i + I_2^i$...

Agora, usamos o problema do 3CNF SAT para encontrar uma determinada valoração para as variáveis de uma determinada fórmula booleana tal que essa valoração satisfaça a soma que queremos encontrar. A soma encontrada, se existente, é o Alimentação Saudável.

4 Análise de complexidade

4.1 Complexidade de tempo

O problema da Alimentação Saudável possui complexidade $O(2)^n$, sendo n as opções dos valores calóricos dos alimentos disponíveis, ou seja, os elementos do conjunto inicial no qual é procurada a soma desejada.

A análise da complexidade de tempo pode ser melhor explicada e detalhada levando em consideração que no pior caso o algoritmo irá calcular todos os subconjuntos de n. Assim, serão geradas todas as enumerações possíveis do conjunto inicial de tamanho n. Consequentemente, o número de subsequências aumentada com n dobrando toda vez que n aumenta de uma unidade.

A visualização da complexidade de tempo pode ser explicada também dado a seguinte equação de recorrência:

$$\mathrm{T(K)} = \begin{cases} k = n, retorna \\ 2T(K+1), seK < n \end{cases}$$

considere k $\in Z$, k = { 0, 1, ..., n}

A complexidade exponencial apresentada determina que quanto maior o número n maior será a quantidade de subconjuntos gerados. Consequentemente, haverá maior demora do algoritmo em computar a solução possível, mesmo com as podas e a paralelização.

As outras funções presentes no algoritmo não ultrapassam $O(2)^n$.

4.2 Complexidade de espaço

A complexidade espacial é $O(2)^n$ já que no pior caso serão necessários acesso espacial de $(2)^n$ no sistema, sendo n as opções dos valores calóricos dos alimentos

disponíveis, ou seja, os elementos do conjunto inicial no qual é procurada a soma desejada. Por fim, todas as respostas geradas ocuparão um espaço exponencial, explicando a complexidade de espaço.

Como a função recursiva cria pilhas de acesso na memória, no pior caso seria necessário alocar espaço para $(2)^n$ soluções multiplicado pelo número de instâncias desejadas.

5 Análise experimental

Para a análise experimental foral realizados dois testes para comparação, um com o algoritmo ainda não paralelizado e outro com o algoritmo aplicado à programação paralela. Assim, além de comprovar a análise de complexidade da seção anterior poderemos também ver a eficiência da aplicação do paralelismo, ou seja, o quanto ele ajudou na otimização do tempo do programa.

No entanto, não será exibida na documentação um análogo entre os testes com paralelismo e o somente com o algoritmo de força bruta. A razão é que somente com a recursividade o tempo gasto para a execução passou dos minutos, enquanto no algoritmo paralelo a execução não passa de milésimos de segundos.

Os testes foram realizados em uma máquina com processador Intel Core 2 Duo com 3GB de memória RAM e arquitetura x86. O sistema operacional em que os testes foram realizados foi o Ubuntu em sua última versão, 15.10 e o compilador usado foi o GCC, nativo do sistema.

Pada cada entrada, houveram 10 execuções diferentes e o tempo foi calculado através do comando time ./lista ./[Nome do arquivo de teste] através do uso da variável real, que calcula o tempo que o programa gastou para calcular as entradas disponibilizadas, ignorando o tempo demorado para que o usuário entre com os dados. Os dados das 10 execuções de cada arquivo foram guardadas para que fosse calculada a média e que o resultado final fosse obtido.

A medida de comparação, foram gerados diferentes casos de testes com tamanhos de n variáveis. O valor n gerado varia entre 0 e 300, que justamente foi o limite dado na especificação, sendo que próximo do 300 foi gerado um caso de teste intermediário. Os dados gerados permitem ter uma ideia clara sobre o tempo x entrada nas duas situações testadas.

Já os casos de testes foram fixados em 3 diferentes casos para o mesmo tamanho de n e o número de threads foi fixada em 10.

5.1 Análise experimental com paralelismo

Já com o paralelismo, os resultados se mostraram bem melhores e mais rápidos, como é demonstrado na tabela a seguir:

Para que possamos compreender melhor os resultados obtidos na tabela acima, foi gerado um gráfico do tempo x entrada. Assim, será possível uma melhor visualização da variação e crescimento do tempo a medida que o valor de n cresce.

Table 1: Análise experimental com paralelismo

Teste	Tamanho n	Tempo(s)
1	50	.005
2	100	.011
3	150	.015
4	200	.023
5	250	.027
6	270	.031
7	300	.035

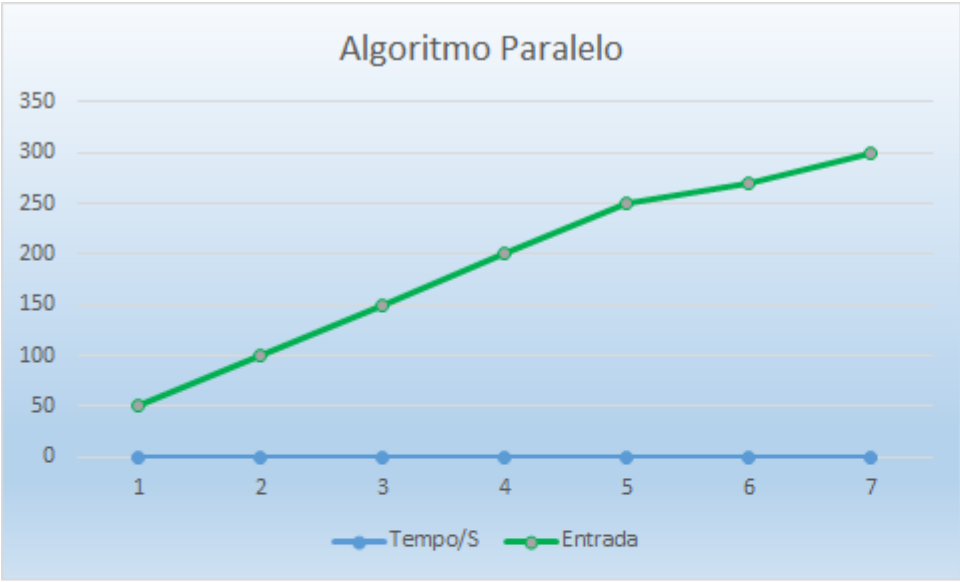

Figure 2: Gráfico gerado

5.2 Análise experimental sem paralelismo

Já sem a paralelização o algoritmo demorou muito tempo para ser executado, sendo que para o primeiro caso de teste o tempo de execução ultrapassou um

minuto. Assim, não será gerado uma tabela com os valores, como foi feito acima, já que para n = 50, a função recursiva de força bruta ultrapassa 60 segundos, já paralelizando o tempo cai drasticamente para 0,005 segundos.

Assim, é perceptível a conclusão que, mesmo sendo um algoritmo que executa em tempo exponencial para casos grandes a paralização ajuda muito na otimização do tempo gasto na execução, gerando resultados precisos mais rápido e de forma eficiente.

6 Conclusão

O problema Alimentação Saudável foi resolvido com sucesso através da modelagem do problema com o algoritmo recursivo de força bruta e com a aplicação da programação paralela, o que otimizou o tempo de execução. Contudo, ainda sim para casos com entradas maiores, ou seja, com vários valores calóricos dos alimentos disponíveis, o problema não pode ser resolvido em tempo polinomial e sim exponencial.

A justificativa para que o problema não possa ser resolvido em tempo polinomial é que Alimentação Saudável é um problema contido na classe NP-Completo. É mais fácil pensarmos que o conjunto de enumerar todos os valores possíveis de um conjunto de tamanho n também é NP-Completo, o que será realizado no pior caso.

A implementação do algoritmo de solução do problema ocorreu com algumas dificuldades, entre elas na parte do paralelismo, já que o conteúdo é um pouco abstrato e não é familiarizado. Enquanto outros conceitos de programação eram aprendidos e trabalhados desde a disciplina Algoritmos e Estruturas de Dados 1, a programação paralela veio como algo novo, diferente que precisou ser melhor estudado para a compreensão de sua funcionamento e implementação.

Além disso, não foi possível criar uma solução de programação paralela ótima, tendo alguns empecilhos. Inicialmente foi pensado a resolução do problema através de um algoritmo dinâmico, mas paraleliza-lo não seria simples, logo foi utilizado o força bruta por sua simplicidade de implementação. No entanto, a forma da solução foi a encontrada para a entrega e funcionamento do trabalho prático.

Embora tenha métodos mais eficientes e rápidos para a modelagem do problema da Alimentação Saudável, a força bruta apresentou bons resultados sendo também de fácil implementação, o que explica a escolha de seu uso. Foi preciso apenas tomar cuidado com fatores como o acesso concorrente das threads e fazer com que a função se adaptasse à função de criação de threads, que pode receber somente um parâmetro.

Com o trabalho proposto foi possível treinar conceitos de programação paralela, paradigmas de programação e também a teoria por trás da classe NP-Completo, que é muito importante na computação para que seja possível provar que um algoritmo não possui solução em tempo polinomial. Assim, toda a matéria lecionada nos módulos correspondentes na disciplina puderam ser melhor estudados e compreendidos.