Problema de particionamento em k subconjuntos balanceados

Marcela Ribeiro de Oliveira
GRR20157372
Ciência da Computação
Universidade Federal do Paraná
Email: mro15@inf.ufpr.br

I. Descrição do Problema

Nesse trabalho utilizamos a técnica de *Branch & Bound* aplicada na resolução do problema de particionar um conjunto em k subconjuntos balanceados.

Dados um conjunto S de itens, uma função que $w : S \rightarrow \mathbb{N}$ e um inteiro k. O problema consiste em encontrar uma partição de S em k subconjuntos de tal forma que $max\{\sum_{x \varepsilon S_i} w(x) | 1 \leq i \leq k\}$ é mínimo.

Além de explicar como o algoritmo foi implementado, esse trabalho também apresenta uma análise experimental com diferentes estratégias de ordenação.

II. Execução do Programa

Para compilar basta executar o comando *make*.

Para executar o programa:

```
./subconjuntos <n> <k> <arquivo> <ordem>
```

Onde o parâmetro ordem pode assumir os seguintes valores:

- 1: elementos são enumerados na ordem do arquivo de entrada.
- 2: elementos são enumerados na ordem crescente do arquivo de entrada.

III. Recorrência de Enumeração

Para enumerar os subconjuntos a recorrência utilizada foi:

$$\text{enumera(v, k, s, i)=} \begin{cases} \emptyset & \text{se } v = \emptyset. \\ v & \text{se } k = 1. \\ enumera(v - s, k - 1, \emptyset, 0) & \text{se } i = n. \\ enumera(v, k, s \cup v[i], i + 1) \cup & \\ enumera(v, k, s - v[i], i + 1) & \text{caso contrário.} \end{cases}$$

Onde:

- v é um vetor que representa o conjunto S, dessa forma, no início da recorrência, v contém o peso de todos os elementos de S;
- k representa o número de partições que se quer encontrar do conjunto S;
- s representa uma possível partição do conjunto S, assim, s começa vazio e vai sendo preenchido com elementos no decorrer da execução;
- i marca o deslocamento dentro do conjunto v e começa com o valor 0.

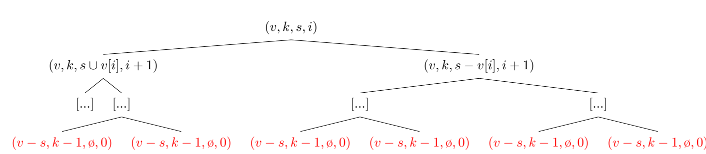
Essa recorrência enumera os subconjuntos na forma de uma árvore binária. A cada chamada, duas novas chamada são criadas. Uma acrescentando no subconjunto S o elemento do conjunto v na posição i (deslocamento no vetor), e outra com o subconjunto S sem esse elemento.

A cada vez que o deslocamento (i) chega em n, ou seja, o vetor foi todo percorrido, quer dizer uma partição foi encontrada. Assim, uma nova chamada recursiva é chamada, só que dessa vez todos os elementos de v que já estão contidas na partição S são removidos. Assim não há como ter subconjuntos repetidos, pois todos os elementos já utilizados em uma partição não ficam mais disponíveis para serem enumerados novamente (são apagados de v). E como uma nova partição foi encontrada, essa nova chamada é feita com k subtraído de 1, pois agora uma partição a menos precisa ser enumerada.

O caso base é quando $k = 1$, ou seja, falta enumerar apenas a última partição. E quando a recorrência chega na base, apenas é retornado o vetor v, pois a última partição é formada por todos os elementos que não estão em nenhuma das partições anteriores.

Quando essa recorrência desce nos ramos mais a esquerda, antes de enumerar todas as k partições já terão acabado os elementos de v. Por isso, um outro caso base que pode ocorrer é quando v fica vazio. Nesse caso, apenas retorna-se o conjunto vazio.

A árvore de enumeração fica da seguinte maneira:



Para cada partição k, essa árvore tem altura n. Portanto, se ela fosse uma árvore completa teria $k2^n$ nodos. Mas como pode-se observar a mesma não é uma árvore binária completa.

IV. Algoritmo

O algoritmo funciona da seguinte maneira:

Na primeira chamada passa-se n, k, o conjunto de pesos ($v\{1, \ldots$ n$\}$), o $offset$ começando em 0, um conjunto $subsets$ que inicia vazio, e duas variáveis do tipo solution s

e $best$ que servem para armazenar a solução atual e a melhor solução já encontrada, onde ambas inicialmente estão vazia.

A struct solution é a seguinte:

```
typedef struct solution{
std::vector<int >subset;
float value;
}solution;
```

O vetor $subset$ guarda todos os elementos contidos na partição e $value$ é a soma dos pesos de todos os elementos dessa partição. Ao utilizar um vetor de tamanho k do tipo solution, é possível guardar todas as k partições.

A cada chamada da recursão são feitas 2 verificações, a primeira verifica se k é igual a 1 ou se v está vazio. Se k é igual a 1, significa que estamos na última partição, essa partição é composta por todos os elementos que sobraram em v. Então após adicionar todos esses elementos na solution $s[k]$ chama-se a função de corte. Se o corte retornar 1, é porque essa partição encontrada não é boa, então incrementa-se o contador de cortes e retorna. Já se v está vazio, significa que todos os seus elementos foram utilizados e há pelo menos uma partição que ficou sem nenhum elemento· Se isso acontecer a função de corte retornará 1 e essa partição é descartada.

Se a partição $k = 1$ não é descartada então a função evaluate_solution é chamada, ela irá comparar se as k (o k original da entrada) partições contidas em s são melhores que as contidas em $best$. Se sim, $best$ passa a ser s e retorna-se 0 pois terminou-se de enumerar k partições.

A segunda verificação checa se o $offset$ (deslocamento) é igual a n, se isso for verdade então todos os elementos de v foram percorridos, independente se todos eles foram escolhidos para fazer parte da partição atual ou não. Se o vetor subsets está vazio, retorna-se 0. Senão, adicionamos a solution $s[k]$ todos os elementos de $subset$, que será a partição k atual. Já que uma partição foi enumerada, é necessário apagar de v todos os elementos que estão nessa partição. Então apaga-se de v todos os elementos que estão em $s[k]$. Feito isso chama-se a função de corte para essa nova partição. Se o corte retornar 1, essa chamada recursiva retorna com 0. Senão, declaramos a variável $subset2$, que irá guardar os elementos da próxima enumeração. E é chamada novamente a função enumera, porém agora com o v modificado, n sendo o tamanho de v (quantos elementos ainda faltam entrar em alguma partição), $k-1$ pois agora falta menos uma partição, s e $best$.

Se nenhuma dessas duas verificações for verdadeira, então prossegue-se a execução do algoritmo em 2 chamadas recursivas, o que significa que nenhuma partição completa ainda foi enumerada. Antes da primeira chamada é adicionado no fim do subconjunto $subsets$ o elemento de v na posição $offset$, e então chama-se recursivamente a função enumera com o $offset$ incrementado de 1. Antes da segunda chamada é removido do subconjunto $subsets$ o elemento correspondente a v na posição $offset$, e então chama-se recursivamente a função enumera com o $offset$ incrementado de 1.

Assim sendo, a cada nova chamada serão criadas duas novas chamadas, onde na primeira o elemento do conjunto está contido no subconjunto e na segunda o mesmo elemento do conjunto não está contido no subconjunto. E quando uma das duas verificações acima citadas são verdadeiras é porque uma partição terminou de ser enumerada.

Algumas variáveis que aparecem no algoritmo não fazem parte da entrada, abaixo pode-se conferir o que cada uma delas representa:

- n_s é um inteiro que guarda o valor de k original; a
- n_cut é um contador, ele é incrementado em um a cada vez que uma subárvore (raiz k) é cortada.

Algoritmo 1: Enumera

```
Entrada: vetor v, int n, int k, int offset,
         vetor subsets, solution s[], solution best[]
Saída: O particionamento do vetor v em k
       subconjuntos balanceados
início
    se k == 1 ou v == ∅ então
        clear_solution(s, k)
        para i = v.begin ate v.end faça
            s[k].subset := s[k].subset ∪v[i]
        fim
        se cut(s, k, v) então
            n_cut := n_cut + 1
            retorna 0
        fim
        evaluate_solution(s, best, n_s)
        retorna 0
    fim
    se offset == n então
        se subsets == ∅ então
            retorna 0
        fim
        clear_solution(s, k)
        para i = subsets.begin ate subsets.end faça
            s[k].subset := s[k].subset ∪subsets[i]
            v := v - subsets[i]
        fim
        se cut(s, k, v) então
            n_cut := n_cut + 1
            retorna 0
        fim
        vetor subsets2;
        enumera(v, v.size(), k − 1, 0, subsets2, s, best)
        retorna 0
    fim
    subsets := subsets ∪v[offset]
    enumera(v, n, k, offset + 1, subsets, s, best)
    subsets := subsets - v[offset]
    enumera(v, n, k, offset + 1, subsets, s, best)
fim
```

V. Função de bound

A função de bound utilizada nesse algoritmo é a seguinte:

- Seja m a média de todos os pesos do conjunto S original;
- Seja $f1$ a soma de todos os pesos da partição k encontrada;

- Se o vetor v está vazio e $k>1$, então corta, pois acabou os elementos do vetor antes da última partição, indicando que há uma ou mais partições vazias;
- Seja $f2$ a média de todos os pesos de v, ou seja, valores que ainda vão entrar nas próximas $k-1$ partições;
- Seja f o mínimo entre m e $f2$;
- Se $f1<f$, então corta, caso contrário, não corta.

A. *Estratégias de Ordenação*

Juntamente com esse bound, duas estratégias de ordenação foram utilizadas. A primeira, enumera as partições conforme a ordem recebida na entrada. A segunda, ordena a entrada na ordem crescente antes de fazer a enumeração.

VI. Análise Experimental

Para a análise experimental, como dependendo do valor de k o tempo de execução é diferente para a mesma instância, diferentes tamanhos de instâncias foram utilizas para diferentes tamanhos de k. A tabela a seguir mostra o tamanho das instâncias utilizadas nos testes para cada valor de k.

I. Testes

Valor de k	Tamanho das instâncias
2	5, 10, 15, 18, 25
3	5, 10, 12, 15, 18
5	5, 8, 10, 12, 13

Para cada valor valor de n (5, 8, 10, 12, 13, 15, 18), foram geradas aleatoriamente 10 instâncias. Como há 2 formas de ordenação diferentes, para cada k e n, 20 testes foram realizados, sendo 10 para cada forma de ordenação. As tabelas a seguir mostram a média de tempo de execução e a média de subárvores (partições k) da árvore que foram cortados durante a enumeração.

A. *Resultados obtidos para k=2*

II. Resultados k=2

n	ordem	cortes	tempo
5	1	23	0.001s
10	1	712	0.003s
15	1	19022	0.104s
18	1	160028	0.950s
25	1	1936928	2m36s
5	2	23	0.001s
10	2	712	0.003s
15	2	19022	0.104s
18	2	160028	0.950s
25	2	1936628	2m39s

B. *Resultados obtidos para k=3*

III. Resultados k=3

n	ordem	cortes	tempo
5	1	113	0.001s
10	1	32283	0.072s
12	1	295512	0.717s
15	1	8001162	23.4s
18	1	215344805	12m24s
5	2	113	0.001s
10	2	32283	0.079s
12	2	295512	23.6s
15	2	160028	0.950s
18	2	215344805	12m25s

C. *Resultados obtidos para k=5*

IV. Resultados k=5

n	ordem	cortes	tempo
5	1	285	0.001s
8	1	48423	0.08s
10	1	1439212	2s
12	1	40207246	1m11s
13	1	215197247	6m3s
5	2	285	0.001s
8	2	48423	0.08s
10	2	1439212	2s
12	2	40207246	1m11s
13	2	215197247	6m22s

Como é possível observar, apesar de diferentes as duas estratégias de ordenação resultam em um mesmo número de subárvores cortadas. E também o tempo de execução entre as duas diferencia-se por segundos.

Outra análise interessante é o aumento do tempo de execução conforme o n e o k crescem. Por exemplo, para $k=2$ e $n=18$, o tempo de execução em média não passa de 1 segundo. Já para o mesmo n com $k=3$, o tempo de execução ultrapassa 12 minutos.

VII. Referências

Stack Overflow, *https://stackoverflow.com/questions/32791033/how-to-divide-a-set-into-k-subsets-such-that-the-sum-of-the-elements-in-the-subs*

Stack Exchange, *https://cs.stackexchange.com/questions/19181/partition-array-into-k-subsets-each-with-balanced-sum*