Trabalho de Análise de Algoritmos

Problema da Mochila

Felipe R. S. Barbosa, Flávio R. J. de Sousa, Leonardo dos Reis Vilela

Faculdade de Ciência da Computação – Universidade Federal de Uberlândia (UFU)
Uberlândia – MG – Brasil

felipe@comp.ufu.br, janones@comp.ufu.br, leo@comp.ufu.br

Resumo. *Este artigo apresenta três soluções para o problema de encontrar um conjunto de itens nos quais a soma de seus pesos não exceda um valor dado e que a soma da suas utilidades seja a maior possível. O primeiro algoritmo é através da força bruta, o segundo algoritmo utiliza a técnica de programação dinâmica [CLR02] e o terceiro é um algoritmo de aproximação [CLR02] que utiliza a técnica de algoritmos gulosos [CLR02]. No decorrer do artigo discutiremos cada um destes.*

1 Introdução

O problema da mochila consiste em dado um conjunto C_n de n itens, representados por $C_n = \{1, 2, ..., n\}$, cada item $i \in C_n$ tem um peso p_i e utilidade u_i ($p_i > 0$ e $u_i > 0$). Determinar um subconjunto $S \subseteq C_n$ tal que a soma dos pesos dos elementos de S seja a maior possível.

Neste artigo iremos apresentar três soluções com algoritmos distintos para o problema da mochila.

O primeiro algoritmo apresenta a resolução mais óbvia, o qual é implementado utilizando a força bruta, ou seja, compara todas as possibilidades até encontrar a melhor. Este algoritmo devolve a resposta correta, mas quando consideramos um conjunto relativamente grande de dados, o tempo gasto para obtermos a resposta é exponencialmente difícil de ser encontrada, logo o tempo deste algoritmo é exponencial.

O segundo algoritmo é o que utiliza o método da programação dinâmica. Este algoritmo também encontra a resposta correta com um tempo melhor que o anterior, porém se aumentarmos consideravelmente a capacidade da mochila, este algoritmo gastará um tempo muito maior do que o esperado. Assim dizemos que este algoritmo possui um tempo pseudo-polinomial, mas na sua essência ele é exponencial.

O terceiro e último algoritmo utiliza o método de aproximação ou heurística. Este algoritmo pode ou não devolver um resultado correto, dependendo de sua execução, para isso existe uma taxa de aproximação à qual o algoritmo possui. Este algoritmo é baseado na programação gulosa.

Estes algoritmos serão discutidos no decorrer deste artigo.

2 Algoritmo de Força Bruta

Um algoritmo que implementa a solução de um problema através da força bruta é aquele que compara todas as possibilidades possíveis de resposta para o problema e devolve a melhor solução, ou a solução mais correta.

Para o problema da mochila, o algoritmo da força bruta compara todas as possibilidades de preenchimento da mochila que não ultrapassem o peso máximo estipulado. Durante este teste, o algoritmo guarda numa variável a maior utilidade conseguida e ao final de todas as comparações, o resultado do algoritmo está armazenado nesta variável.

Não utiliza-se nenhuma estrutura de dados especial , apenas utiliza-se uma variável inteira para armazenar a maior utilidade encontrada até aquele momento da comparação.

O algoritmo da força bruta pode ser visto a seguir:

```
Mochila-B-Rec(p, u, n, M)

Se n = 0
     Então devolva 0
Senão  A ? Mochila-B-Rec(p, u, n-1, M)
        Se p[n] > M
              Então devolva A
              Senão B ? u[n] + Mochila-B-Rec(p, u, n-1, M-p[n])
                    Devolva max(A, B)
```

1 – Algoritmo Força Bruta para o problema da mochila

Se denotarmos por T(n) o consumo de tempo no pior caso podemos dizer que:

$T(0) = 1$
$T(n) = T(n-1) + T(n-1) + \Theta(1)$

A solução desta recorrência tem a forma $2^0 + 2^1 + ... + 2^n$. Assim $T(n) = \Theta(2^n)$.

Este algoritmo é ineficiente pelo fato de que ele refaz várias vezes a solução de vários subproblemas já calculados.

3 Algoritmo com programação dinâmica

Um algoritmo que implementa a programação dinâmica, utiliza como artifício o armazenamento de dados já calculados para que estes possam ser usados na resolução dos problemas seguintes.

Utilizamos como estrutura especial uma matriz para guardar as soluções intermediárias entre i e j, afim de evitar recálculos de soluções já anteriormente recalculadas.

Para a resolução do problema basta construir uma tabela Matriz_Acc onde Matriz_Acc(i, j) é o valor máximo de um subconjunto de {1, ..., i} cujo peso é no máximo j. Daí temos que:

Matriz_Acc (i, j) = $\max\{u(S) : S \subseteq \{1, . . . , i\}, p(S) = j\}$, sendo 0 <= i <= n e 0 <= j <= M.

Dados dois vetores x[1..n] e p[1.. n], denotamos por x * p o produto escalar
p[1]x[1] + p[2]x[2] + ... + p[n]x[n].

Suponha dado um número inteiro não-negativo M e vetores não-negativos p[1..n] e u[1..n]. Uma mochila é qualquer vetor x[1..n] tal que x * p <= M e 0 <= x[i] <= 1 para todo i. O valor de uma mochila é o número x * v. Uma mochila é ótima se tem valor máximo.

Um mochila x[1..n] tal que x[i] = 0 ou x[i] = 1 para todo i é dita booleana, pois ou o elemento será adicionado à mochila ou não será.

Suponha que x[1..n] é mochila boolena ótima para o problema (p; u; n; M).

Se x[n] = 1 então x[1..n-1] é mochila boolena ótima para (p; u; (n -1); M - p[n]) senão x[1..n] é mochila boolena ótima para (p; u; n-1; M)

O algoritmo com programação dinâmica pode ser visto a seguir :

```
Mochila_Dinamica (p, u, n, M)

1.   Para i ? 0 até n faça
2.       Matriz_Acc[0][i] ? 0
3.   Para i ? 1 até M faça
4.       Matriz_Acc[i][0] ? 0
5.       Para j ? 1 até n faça
6.              Valor1 ? Matriz_Acc[i][j-1]
7.              Se p[i-1] > i
8.                     Então valor2 = 0
9.              Senão
10.                    valor2 = Matriz_Acc[(i-p[j-1])][j-1] + u[j-1]
11.      Se Valor1 > Valor2
12.             Então Matriz_Acc[i][j] ? Valor1
13.      Senão Matriz_Acc[i][j] ? Valor2
```

2 – Algoritmo em Programação Dinâmica para o problema da mochila

A execução do algorimo Mochina_Dinamica, o qual utiliza a programação dinâmica, é Θ (nM), onde n é a quantidade de itens e M é a capacidade total da mochila. Analisando o algoritmo, temos que a linha 3 é responsável por um laço M e na linha 5

temos um outro laço incorporando n repetições, como este último é um laço aninhado sob um laço M, logo tempos um consumo de tempo em Θ (nM).

Esta taxa de aproximação não é satisfatória, imagine que uma mudança trivial na escala de medidas de pesos multiplica por 5000 os números p[1], ..., p[n] e M, o problema continua essencialmente o mesmo mas o tempo consumido pelo algoritmo é 5000 vezes maior. Assim, diz-se que este algoritmo é essencialmente exponencial pois:

$$\Theta(nM) = \Theta(n\ 2^{\ln(M)})$$

e ln(M) é a medida certa do tamanho do número M.

4 Algoritmo por Aproximação

Um algoritmo que utiliza o método de aproximação tem por objetivo encontrar uma solução aproximada para o problema com tempo na ordem de n ou n^2. Além disso, temos uma taxa de aproximação representada por

$$MAX(C^*/C, C/C^*)$$

onde C^* é o custo ótimo e C é custo obtido. Esta taxa de aproximação deve ser menor ou igual a 2.

Para a resolução do problema da mochila, utilizamos o método de aproximação baseado em algoritmos gulosos.

Este algoritmo ordena a entrada com relação à taxa obtida da divisão do valor pelo peso, essa ordenação é feita do maior para o menor. O método de ordenação utilizado é o QuickSort o qual possui um tempo de ordenação na ordem de nlog(n).

Depois de realizada a ordenação, a solução do problema é obtida simplesmente pegando os primeiros elementos da ordenação até que a capacidade da mochila seja atingida, se existir um elemento que ao ser colocado na mochila ultrapasse a sua capacidade, este elemento é desprezado e passa-se para o próximo.

O algoritmo pelo método de aproximação pode ser visto a seguir:

```
Mochila_Aproximacao(p,u,M)
     maior = max {u[i] : 1 ≤ i ≤ n}    // Admite-se p[i] ≤ M, 1 ≤ i ≤ n
     return max {maior, Mochila_Guloso(p,u,M)}

Mochila_Guloso(p,u,M)
     Ordenar p e u tal que u[i]/p[i] ≥ u[j]/p[j] para 1 ≤ i ≤ j ≤ n
     peso = 0; valor = 0
     for i = 1 to n
         if peso + p[i] ≤ M
               valor += u[i]
               peso += p[i]
     return valor
```

A execução do algoritmo para resolver o problema da mochila que utiliza o método de aproximação possui um tempo na ordem de nlog(n). Este tempo é devido ao método utilizado para a ordenação da entrada que é o QuickSort. Ademais, gasta-se um tempo na

ordem de n para construir a solução ótima, já que visitaremos as posições do vetor de relação u[i]/p[i] no máximo uma vez cada.

5 Discussão sobre os limites e apresentação da base de entrada

Os limites que serão discutidos aqui foram baseados e executados no seguinte ambiente:

Material: Intel Pentium IV 2 GHz (clock real de 2000.777) Single, cache 512 KB, 512 MB de RAM.
Sistema Operacional: GNU Red Hat Fedora 2, kernel 2.6.5.
Os fontes de todos os algoritmos foram compilados com gcc (GCC) 3.2.2 20030222 (Red Hat Linux 3.2.2-5).

Os tempos de execução foram recuperados através do *time* de David Keppel.
A base de entrada para a discussão dos limites dos três algoritmos que serão apresentadas possui 5 milhões de pares de números (peso e utilidade respectivamente) . Todos os testes apresentados foram executados utilizando uma mochila com capacidade máxima de 1000.

5.1 Limite por Força Bruta

O algoritmo por força bruta apresentou um comportamento exponencial quando fora executado para os seguintes dados:

Qtde Itens	Tempo (seg)	Qtde Itens	Tempo (seg)
10	0,002	110	2,53
20	0,002	120	3,965
30	0,003	130	12,323
40	0,004	140	63,082
50	0,031	150	89,01
60	0,195	160	93,88
70	0,407	170	225,3
80	0,717	180	257,1
90	0,985	190	523,67
100	0,977	200	3240,37

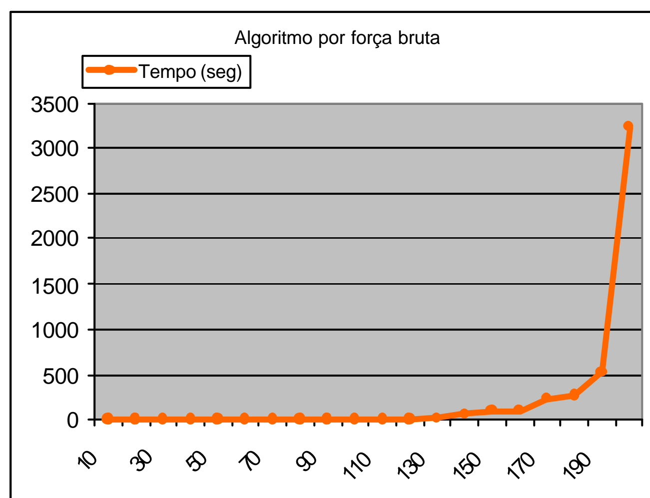

Tabela 1 e Gráfico 1: resultados do algoritmo por força bruta

O algoritmo não foi possível de ser executado para uma quantidade de 220 itens, pois após 90 minutos do lançamento ainda não havíamos encontrado o resultado, logo consideramos para a quantidade de 220 itens o resultado de *timeout*. Portanto, o limite para este algoritmo fora de 200 itens. Indubitavelmente este algoritmo possui ordem exponencial.

5.2 Limite por Programação Dinâmica

O algoritmo por programação dinâmica apresentou um comportamento exponencial quando fora executado para os seguintes dados:

Qtde Itens	Tempo (seg)	Qtde Itens	Tempo (seg)
10000	0,951	80000	8,938
20000	1,947	90000	11,707
30000	3,004	100000	12
40000	5,194	110000	13,54
50000	5,798	120000	31,43
60000	6,696	130000	300,1
70000	8		

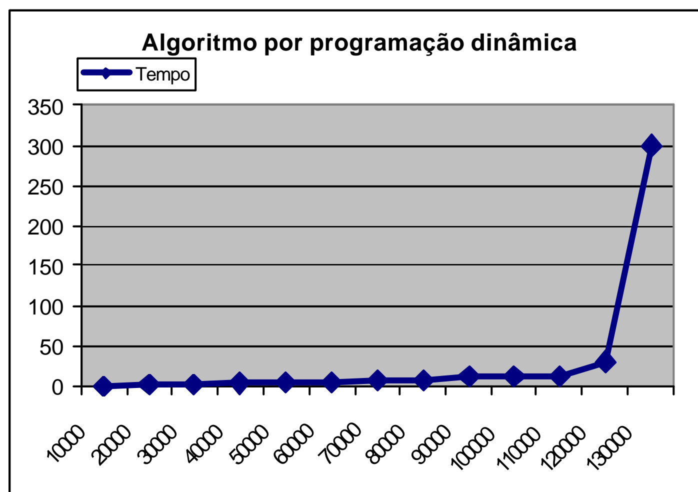

Tabela 2 e Gráfico 2: resultados do algoritmo por programação dinâmica

O algoritmo não foi possível de ser executado para uma quantidade de 150 mil itens, pois após 70 minutos do lançamento ainda não havíamos o resultado, logo consideramos para a quantidade de 130 mil itens o resultado de *timeout*. Portanto, o limite para este algoritmo fora de 130 mil itens.

5.3 Limite por Aproximação

O algoritmo por aproximação apresentou os seguintes resultados:

Qtde Itens	Tempo (seg)
500000	0,977
1000000	2,04
1500000	3,154
2000000	4,28
2500000	5,536
3000000	6,654
3500000	7,926
4000000	9,183
4500000	10,501
5000000	11,829

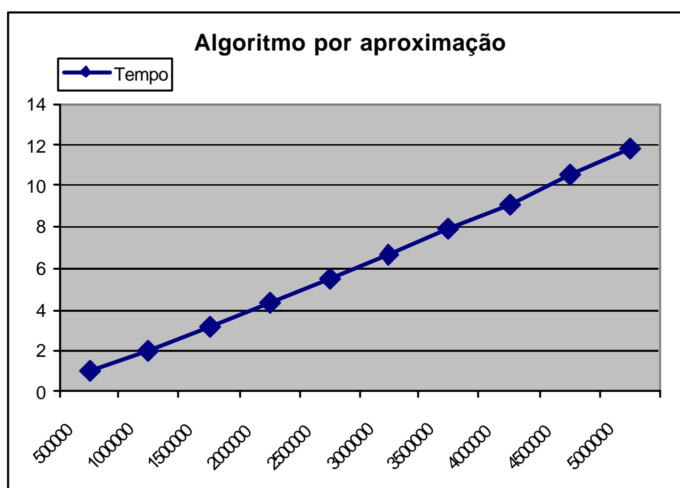

Tabela 3 e Gráfico 3: resultados do algoritmo por aproximação

Executamos o algoritmo por aproximação para uma base infinitamente superior aos testes efetuados para os algoritmos por força bruta e por programação dinâmica. De fato, a técnica por aproximação permitiu reduzir um algoritmo de ordem exponencial para ordem polinomial.

Ademais, não encontramos um limite por aproximação, cabendo este ser limitado por razões técnicas como: arquitetura, memória etc.

6 Benchmark

O benchmark fora realizado comparando os três algoritmos com entradas de tamanhos variados e um arquivo com 10 milhões de números entre 1 e 1000.

A geração da base de testes foi criada utilizando *rand – random number generator*, o qual está disponível na biblioteca *stdlib.h*.

itens	Tempo Bruto	Tempo Dinâmico	Tempo Aproximação	itens	Tempo Bruto	Tempo Dinâmico	Tempo Aproximação
10	0,002	0,002	0,002	60000		6,719	0,105
50	0,031	0,003	0,002	70000		7,727	0,126
100	0,977	0,005	0,002	80000		9,013	0,143
110	2,53	0,006	0,002	90000		11,593	0,163
120	3,965	0,006	0,002	100000		11,900	0,177
130	12,323	0,007	0,002	110000		13,173	0,195
140	63,082	0,007	0,002	120000		13,700	0,22
150	89,01	0,008	0,002	125000		13,940	0,231
160	93,88	0,009	0,002	126000		16,010	0,232
170	225,3	0,009	0,002	127000		16,070	0,236
180	257,1	0,010	0,002	128000		45,302	0,236
200	3240,37	0,012	0,002	129000		136,600	0,236
300		0,020	0,002	130000		300,930	0,236
400		0,029	0,003	150000			0,277
1000		0,085	0,003	1000000			2,04
4000		0,368	0,008	2000000			4,226
10000		0,957	0,017	3000000			6,639
24000		2,343	0,041	4000000			9,154
30000		2,989	0,051	5000000			11,745
50000		5,573	0,087				

Tabela 4: resultados comparativos entre os três algoritmos para uma base gerada aleatoriamente

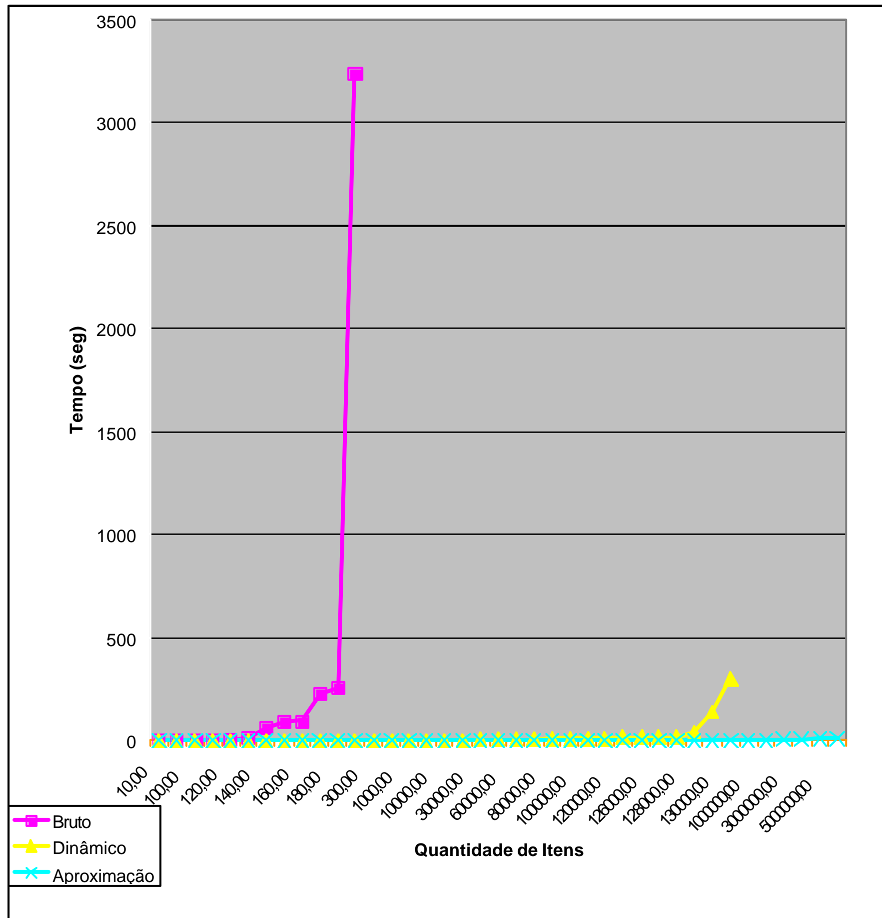

Gráfico 4: Benchmark entre os 3 algoritmos

Através do gráfico da página anterior verificamos novamente que o algoritmo por força bruta como o algoritmo por programação dinâmica são exponenciais, enquanto o algoritmo por aproximação é polinomial.

7 Trade-off entre complexidade computacional e qualidade do resultado

Entre quaisquer duas ou mais soluções não dominadas verifica-se que uma melhoria em um objetivo sempre está associado ao sacrifício de um outro objetivo. Isto é, verifica-se sempre uma compensação ("*trade-off*") entre objetivos no conjunto de soluções. No caso dos algoritmos apresentados, sempre temos que a complexidade computacional proporciona um ganho na qualidade do resultado.

Quando analisamos o *trade-off* para o **algoritmo por força bruta** vemos que as críticas que são alvo neste algoritmo devem-se ao elevado esforço temporal necessário para o cálculo exaustivo (recálculos) até se encontrar a solução ótima. De fato não existe um elevado esforço computacional para a sua resolução e nem mesmo se produz um resultado em tempo ótimo. Outrossim, este algoritmo possui um limite reduzido não sendo capaz de

suportar uma grande entrada de dados, entretanto com entradas pequenas ele sempre fornece uma solução ótima.

No **algoritmo por programação dinâmica** conseguimos sempre encontrar a solução ótima sem a execução de cálculos exaustivos (recálculos), entretanto para evitar os recálculos somos obrigados a utilizar uma matriz para estocagem dos resultados intermediários, aumentando assim a complexidade do algoritmo. Portanto, temos uma melhoria na qualidade do resultado, pois minimizamos a quantidade de recálculos e conseguimos reduzir o tempo para encontrar a solução ótima (quando se comparado com o algoritmo por força bruta). Contudo se a quantidade de dados for imensamente grande teremos problemas na execução deste algoritmo, podendo ocorrer *timeout*, pois como já vimos, ele é exponencial. Análogo ao por força bruta, ele sempre fornece uma solução ótima.

Finalmente, apresentamos o **algoritmo por aproximação**, neste acrescentamos uma pré-ordenação de dados antes da busca da solução. A complexidade deste algoritmo se concentra principalmente na ordenação dos dados. Entretanto é justamente através da ordenação de dados que conseguimos reduzir o tempo para se encontrar a solução, em contra-partida a solução encontrada pode não ser ótima, e sim aproximada.

Em conclusão, temos que o aumento da complexidade computacional nem sempre proporciona uma redução de tempo, mas indubitavelmente a qualidade do resultado é relativamente proporcional à complexidade computacional.

8 Curva n X Taxa de aproximação

Abaixo apresentaremos os dados obtidos em testes realizados para comparação de programação dinâmica e por aproximação

itens	util. (dinâmico)	util. (aproximação)	Taxa de Aproximação
5000	60978	60844	0,9978
10000	85102	85034	0,9992
15000	102580	102536	0,9996
20000	116793	116680	0,9990
25000	134422	134314	0,9992
30000	145159	145072	0,9994
35000	153843	153698	0,9991
40000	163124	163031	0,9994
45000	171496	171385	0,9994
50000	177832	177732	0,9994

Tabela 5: Taxa de aproximação (aproximação / dinâmico)

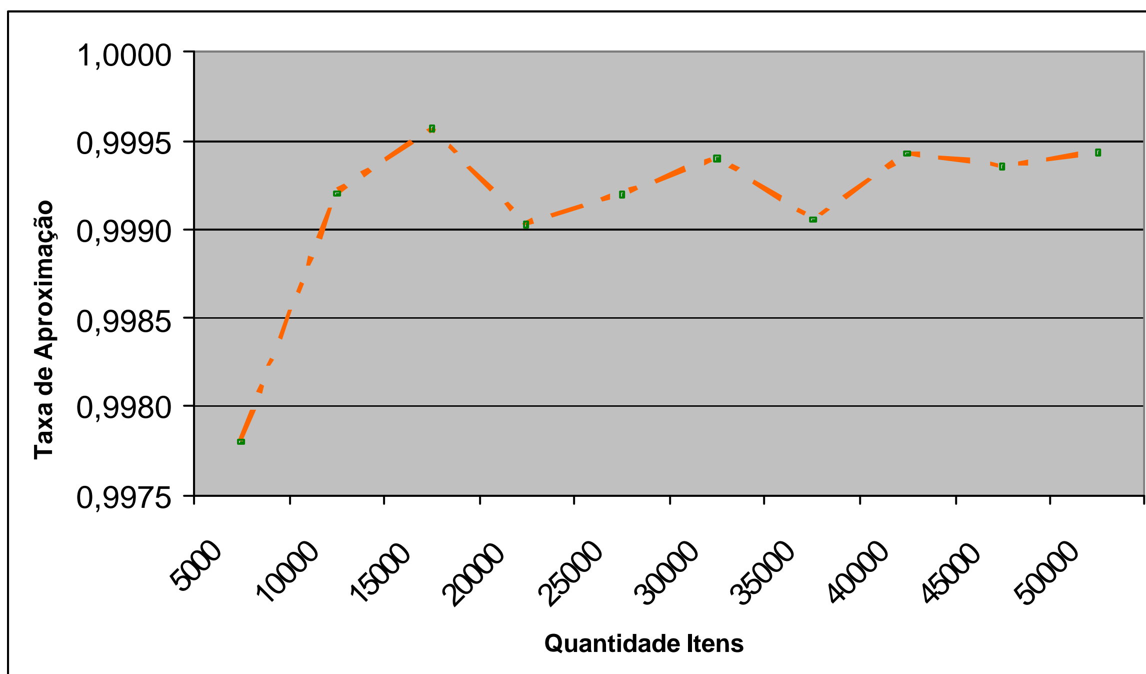

Gráfico 5: Gráfico entre taxa de aproximação

Observamos que a taxa de aproximação média é 0,9991 (99,91%), ou seja, o algoritmo de aproximação apresenta um bom desempenho com resultados bem próximos ao ótimo.

9 Conclusão

O método de aproximação é uma ferramenta que visa a otimização de algoritmos para a resolução de problemas, construindo algoritmos polinomiais para problemas que normalmente são exponenciais, entretanto podendo não nos devolver a solução correta, mas sim uma solução aproximada, daí o nome do método.

Este artigo apresentou três algoritmos para a resolução do problema da mochila. O primeiro algoritmo é através da força bruta, o segundo algoritmo utiliza a técnica de programação dinâmica e o terceiro é um algoritmo de aproximação que utiliza a técnica de algoritmos gulosos.

Podemos notar que através do algoritmo por aproximação conseguimos primeiramente aumentar o limite superior da entrada, fazendo com que este tenda para um número extremamente grande, e em segundo reduzimos consideravelmente o tempo para se encontrar uma solução aproximada.

Portanto, conclui-se que o método de aproximação é útil para minimizar o tempo de algoritmos complexos.

10 Bibliografia

[CLR02] Cormen, Thomas H. ; Leiserson, E. Charles; Rivest, L. Ronald; Stein, Clifford R. and Renault; (2002) **"Algoritmos - Teoria e Prática ",** Tradução da segunda versão americana . Editora campos