

UNIVERSIDADE FEDERAL DO MARANHÃO - UFMA
CENTRO DE CIÊNCIAS EXATAS E SUAS TECNOLOGIAS

Bruno Araujo Muniz

Estudo de algoritmos para resolução do problema bin packing

São Luís, 2020

Bruno Araujo Muniz

Estudo de algoritmos para resolução do problema bin packing

Relatório técnico apresentado como requisito
para obtenção de aprovação na disciplina
Algoritmos II, no Curso de Ciência da Computação,
na Universidade Federal do Maranhão.

Prof. Dr. Alexandre Cesar Muniz de Oliveira

São Luís, 2010

SUMÁRIO

Introdução	4
Problema da mochila	4
Bin packing	4
Algoritmos estudados	5
Enumerativo Guloso	5
Branch and Bound Guloso	6
Algoritmo guloso	7
Random Search	7
Hill Climbing	8
Simulated Annealing	9
Algoritmo Genético	10
PROCEDIMENTOS EXPERIMENTAIS	10
Algoritmos Determinísticos	10
Análise das execuções para os algoritmos Determinísticos	11
Algoritmos estocásticos	13
Análise das execuções para os algoritmos Estocásticos	13
Algoritmo Genético VS Guloso	14

Introdução

Na computação há sempre várias maneiras de se chegar a um resultado de um determinado problema, como por exemplo algoritmos de ordenação, todos eles fazem a mesma coisa, ordenar um vetor de elementos, mas cada um desses algoritmos têm a sua própria complexidade, que define o custo computacional para realizar tal ordenação.

Então basta buscar aquele que tem a menor complexidade como o MergeSort? a resposta é depende, pois algoritmos que possuem uma complexidade ótima, exigem um tempo a mais do programador pela sua complexidade, onde em casos pequenos é mais interessante optar por um algoritmo de complexidade maior que são mais fáceis de reproduzir, e o tempo que ele levará para ordenar será aceitável para instâncias pequenas, assim fazendo com que seu tempo seja melhor aproveitado.

E é isso que iremos observar, utilizando como base o problema de minimização de mochilas (bin packing), observando o comportamento de algoritmos gulosos e estocásticos para a resolução deste problema.

Problema da mochila

O problema da mochila consiste em dado um vetor de pesos $V = \{p_1, p_2, p_3, \dots, p_n\}$, encontrar uma configuração em que o número de elementos de V dentro da mochila seja maximizado, considerando que a mochila tem uma capacidade máxima suportada que pode carregar.

Bin packing

Bin packing ou o problema do empacotamento de mochilas consiste em empacotar múltiplas mochilas, de forma que o número total de mochilas seja minimizado, ou seja dado um vetor de pesos $V\{p_1, p_2, p_3, \dots, p_n\}$, o elementos de V devem ser armazenados em mochilas de capacidade B , de modo que nenhum elemento fique de fora, outra etapa realizada é o da minimização dessas mochilas, pois não basta apenas empacotar, deve-se também garantir que o número de mochilas

utilizados é o menor número possível, esse problema é considerado complexo e de difícil implementação, considerado np-difícil.

Algoritmos estudados

Os algoritmos utilizados para a resolução do empacotamento de mochilas, estão separado em dois tipos, os que utilizam abordagens gulosas que tem como propriedade o determinismo, que significa que independente de quantas vezes o algoritmo for executado, se a entrada for a mesma a saída sempre será a mesma, e algoritmos estocásticos, que utilizam a aleatoriedade como recurso para solucionar o problema, isso quer dizer que pelo uso da aleatoriedade eles se tornam algoritmos não determinísticos, e para cada execução do algoritmo mesmo que seja a mesma entrada a saída será sempre diferente, os algoritmos a seguir são uma forma genérica dos que foram implementados.

Enumerativo Guloso

Este algoritmo utiliza enumeração para encontrar a melhor configuração para o problema da mochila, ou seja, a mochila é resolvida com enumeração, que é chamada enquanto houver resíduo, observe o seu funcionamento no trecho de código genérico na tabela 1.

```
while(Residuo != 0) solucao = Enumera(Todas as Possibilidades);

Enumera(inteiro){

    for i = 0 ; i < inteiro ; i++ {

        solucao = incremento(solucao)

        a melhor solução é definida como a que comporta mais peso, sem
        estourar
        solucao = Melhor_Solucao(solucao, solucao anterior)

    }

}

Complexidade:  $O(n \cdot 2^m)$ 
```

tabela-1

Branch and Bound Guloso

O Branch and bound guloso funciona de maneira muito parecida com o enumerativo, a diferença entre os dois acontece durante a enumeração da mochila na função enumera, que ao invés de percorrer todas as possibilidades é utilizada uma função recursiva onde o algoritmo viaja até a folha das soluções viáveis, ignorando as soluções que estouram a capacidade da mochila, e volta retornando subárvores e escolhendo a melhor entre elas, e é guloso pelo mesmo motivo do enumerativo, pois a função BaB, e chamada para empacotar itens nas mochilas até que não sobre resíduo, observe o pseudocódigo na tabela 2.

```
while(Residuo != 0) solucao = Branch_and_Bound(solucao, tamanho, peso1, peso2,
posicao);

Branch_and_Bound(solucao, tamanho, peso1, peso2, posicao){
    os pesos definem em qual recursão irá entrar, assim eliminando soluções
    inválidas no caso de estouro

    peso1 = peso1 + solucao[posicao]
    peso2 = peso2

    if tamanho > 1 {
        if peso1 < peso_Max {
            dir = (solucao, tamanho - 1, peso1, peso2, posicao+1)
        }
        if peso2 < peso_Max {
            esq = (solucao, tamanho - 1, peso1, peso2, posicao+1)
        }
    }

    return Melhor_Solucao ( dir, esq )
}
```

Complexidade:

$O(n \cdot 2^m)$ pior caso, onde **n** é o número de mochilas e **m** o número de pesos.

$O(n \cdot \log m)$ no melhor caso

tabela-2

Algoritmo guloso

O algoritmo guloso não se preocupa em minimizar mochilas, ele apenas busca o melhor resultado no momento, sem se preocupar a longo prazo. A versão implementada ordena o vetor de pesos $V\{p_1, p_2, p_3, \dots, p_n\}$ em ordem decrescente e acumula em um contador quais itens cabem na mochila sem que ela estoure, como mostra a tabela-3.

```
ShellSort( V )
    i = 0
    peso = 0

    while( Resíduo != 0){

        //para se o peso atingir o máximo ou se percorrer o vetor todo
        while( peso != peso_max && i < tamanho V){
            //verifica se o elemento já foi empacotado
            if( ! Empacotado ( V[i] ) ){
                peso = peso + Peso _V[i]
                V[i] = False
            }
            //se o elemento adicionado estoura a bolsa, ele é removido
            if (peso > peso_Max) {
                peso = peso - Peso _V[i]
                V[i] = True
            }
        }
        i++;
    }

Complexidade:  $O(m*n)$ 
```

tabela-3

Random Search

É um algoritmo de fácil implementação, pois gera suas soluções aleatoriamente, e grava a melhor solução entre essas geradas como solução do problema, como já foi dito anteriormente por usar a aleatoriedade ele é classificado como um algoritmo estocástico ou não determinista, veja sua implementação na tabela 4. A solução é gerada de forma totalmente aleatória, e o resíduo é calculado com base na quantidade de estouros nas mochilas.

```
Random_Serach(N_Interacoes, informacoes_problema){

    for i = 0 ; i < N_Iterações ; i++){

        //gera um elemento totalmente aleatório
```

```

        solucao = Gerador_Random()

        salva o resíduo em tmp
        tmp = Verifica_Residuo()

        se o resíduo atual for menor que o anterior
        if(tmp < residuo_solucao) residuo_solucao = tmp

    }return residuo_solucao
}

```

tabela-4

Hill Climbing

Este algoritmo também se utiliza de geração aleatória se solução, mas diferente do Random Search ele a utiliza somente na primeira iteração, o conceito utilizado neste algoritmo é o conceito de vizinhança, onde ele vai pegar a primeira solução aleatória e alterar alguns elementos dela com o objetivo de encontrar uma solução melhor que a anterior, esse conceito é conhecido como subida da ladeira, onde o algoritmo busca soluções melhores a partir de uma inicial.

```

Hill_Climbing(N_Interacoes, informacoes_problema){

    gera solução e salva o resíduo
    solucao = Gerador_Random()
    tmp = Verifica_Residuo()

    for i = 0 ; i < N_Interações ; i++){

        procura uma nova solução na vizinhança
        solucao1 = Vizinho(solucao)

        tmp = Verifica_Residuo()
        verifica qual das duas soluções é a melhor
        if(solucao1 < solucao) solucao = solucao1

    }return solucao
}
Complexidade: O(n)

```

tabela-5

A função Vizinho altera entre 1 e 10% dos valores da solução.

Simulated Annealing

O Simulated Annealing diferente do hill climbing se permite aceitar soluções piores do que a atual, mas isso depende de uma certa variável chamada temperatura, essa temperatura vai diminuindo com o tempo e o algoritmo deixa de aceitar soluções piores, e passa a se comportar como o hill climbing.

```
Simulated_Annealing(N_Interacoes, informacoes_problema){

    solucao = Gerador_Random()
    tmp = Verifica_Residuo()
    best = solucao

    for i = 0 ; i < N_Interacoes ; i++){
        solucao1 = Vizinho(solucao)

        tmp = Verifica_Residuo()

        if(solucao1 < solucao) {
            solucao = solucao1

            if(solucao1 < best){
                best = solucao1
            }
        }else{
            a função exp(x) é equivalente a euler^x
            if (exp((solucao-solucao1)/temperatura) > rand()){
                solucao = solucao1
            }
        }
        temperatura = temperatura * 0.98
    }return solucao
}
```

tabela-6

Algoritmo Genético

O algoritmo genético diferente dos últimos 3 algoritmos estocásticos ao invés de depender totalmente da aleatoriedade busca gerar uma população de soluções e identificar seus melhores indivíduos, com o intuito de combiná-los para conseguir encontrar melhores soluções, os melhores indivíduos ganham uma pontuação e a partir dela serão ou não selecionados para ir para a próxima geração ou para participar de cruzamentos.

```

Genetico(N_Geracoes, tamanho_populacao){
    populacao = Gerar(tamanho_populacao)

    for i = 0 ; i < N_Geracoes ; i++){
        seleciona uma porcentagem da população para a próxima geração
        Selecao(populacao)

        j = n_de_Selecionados
        while( j < tamanho_populacao){

            a e b são números aleatórios dentre os selecionados para a próxima geração
            a = rand() % n_de_Selecionados
            b = rand() % n_de_Selecionados

            aux = Crossover(populacao[ a ], populacao[ b ] )

            a taxa de mutação é de 10%
            if( rand() % 100 < 10) Mutacao(aux)
            j++
        }
    }
}

```

tabela-7

PROCEDIMENTOS EXPERIMENTAIS

Algoritmos Determinísticos

Para a análise dos algoritmos foram utilizados amostras no seguinte formato:

150 3

45

30

100

Onde o primeiro número da primeira linha corresponde ao peso de uma mochila, e o segundo a quantidade de elementos, os que vem a seguir são os pesos, mas na execução dos algoritmos estocásticos existe mais um algarismo que corresponde ao número de mochilas.

Além disso, para os algoritmos determinísticos foi utilizada uma codificação

binária, como mostra a tabela tabela-8.

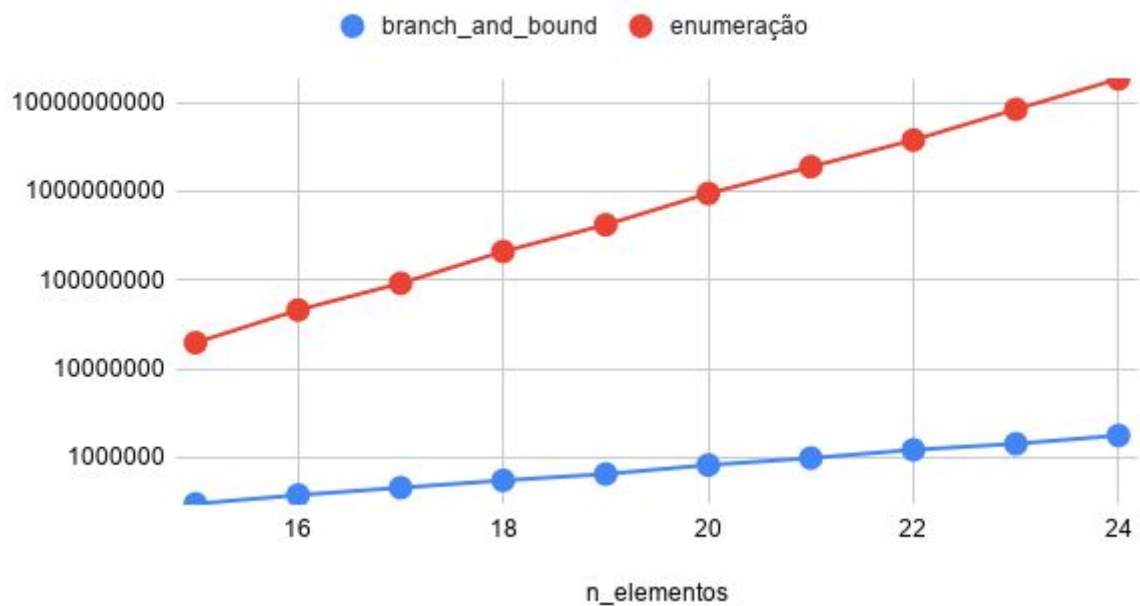
peso = 150	elementos = 12				
Elementos	Mochila 1	Mochila 2	Mochila 3	Mochila 4	Mochila 5
42	0	0	0	1	0
69	0	0	1	0	0
67	0	1	0	0	0
57	1	0	0	0	0
93	1	0	0	0	0
90	0	0	0	1	0
38	0	1	0	0	0
36	0	0	0	0	1
45	0	1	0	0	0
42	0	0	0	0	1
33	0	0	0	0	1
79	0	0	1	0	0
Distribuição de pesos	150	150	148	132	111

tabela-8 - resultado do algoritmo de enumeração

Análise das execuções para os algoritmos Determinísticos

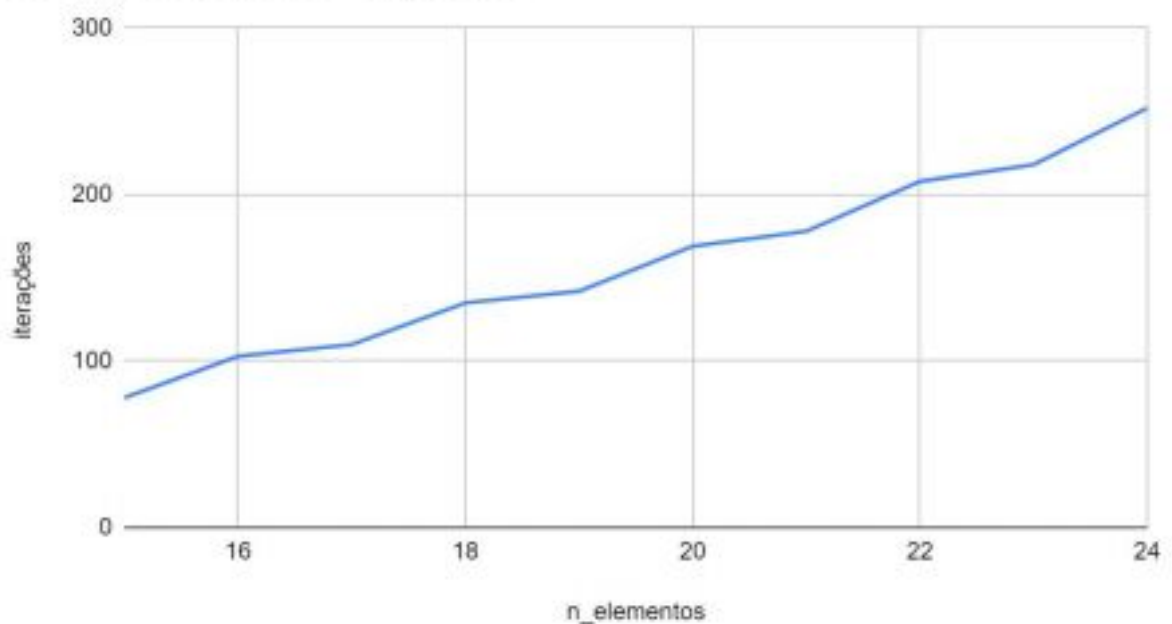
Ao realizar um total de 10 execuções com um total de 10 entradas diferentes é possível observar o comportamento dos algoritmos a cada iteração, as 10 entradas foram as mesmas para os 3 algoritmos, abaixo na figura 1 e 2 é possível ver o comportamento assintótico dos algoritmos determinísticos.

branch_and_bound e enumeração



**gráfico de interações entre
numerador e branch and bound
figura-1**

iterações versus n_elementos



**gráfico de iterações do
algoritmo guloso
figura-2**

Ao analisar os gráficos e fazer a comparação entre os 3 algoritmos, é possível perceber que o algoritmo guloso precisa fazer muito menos iterações, para chegar ao mesmo resultado que os outros dois, por tanto do lado dos algoritmos determinísticos o

algoritmo guloso levou a melhor.

Algoritmos estocásticos

Foram utilizadas as mesmas amostras para testes mostradas no tópico anterior, mas com a adição de mais um elemento à estrutura da entrada, que é o número de mochilas necessárias para empacotar os itens.

150 3 2

45

30

100

A solução foi codificada utilizando uma sequência de itens com repetição onde para contar todas as possibilidades é dada pela expressão m^n .

exemplo: dada solução $S\{3 \ 5 \ 3 \ 4 \ 1 \ 0 \ 0 \ 4 \ 5 \ 1 \ 5 \ 2 \ 3 \ 4\}$, um vetor de pesos $V\{42, 69, 67, 57, 93, 90, 38, 36, 45, 42, 33, 79, 27, 57, 44\}$

3	5	3	4	1	0	0	4	5	1	5	2	3	4	2
42	69	67	57	93	90	38	36	45	42	33	79	27	57	44

tabela-9 - resultado da execução do algoritmo genético

mochila 0 : $90 + 38 = 128$

mochila 1 : $93 + 42 = 135$

mochila 2 : $44 + 79 = 123$

mochila 3 : $42 + 67 + 27 = 136$

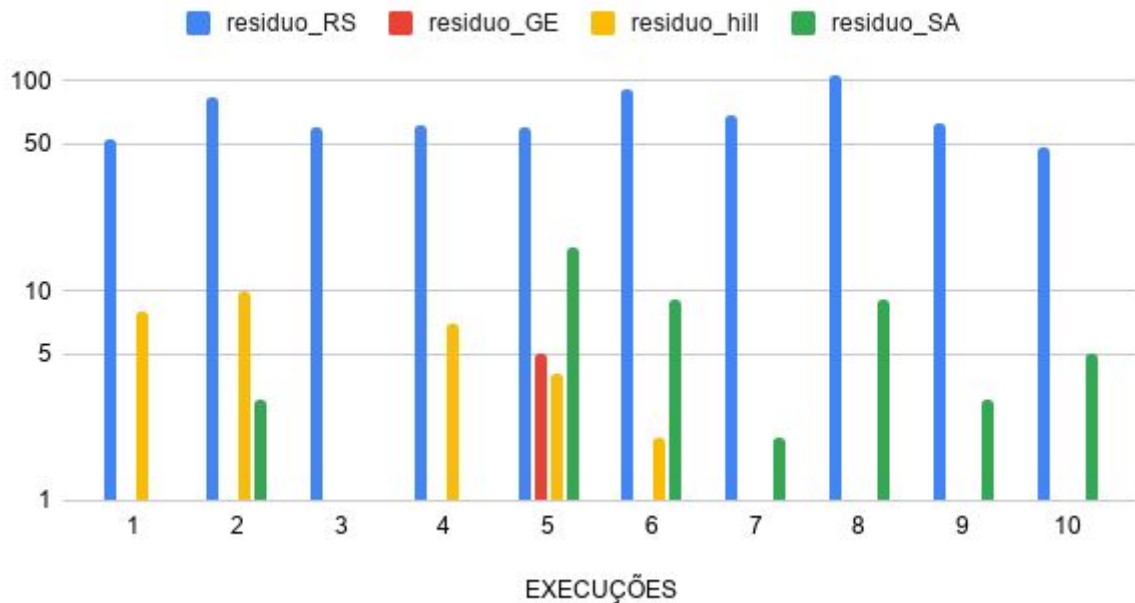
mochila 4 : $57 + 36 + 57 = 150$

mochila 5 : $69 + 45 + 33 = 147$

Análise das execuções para os algoritmos Estocásticos

Ao executar os 4 algoritmos estocásticos sobre a mesma entrada 10 vezes, foi possível obter os seguintes resultados apresentados na figura-3.

residuo_RS, residuo_GE, residuo_hill e residuo_SA



GE(genético), RS(random search), hill(hill climbing) e SA(simulate annealing)
figura-3

É possível ver que o algoritmo genético acaba por ser mais efetivo à medida que as execuções são realizadas, entre os algoritmos estocásticos venceu o genético.

Algoritmo Genético VS Guloso

Antes de pôr os algoritmos a prova é importante lembrar de seu funcionamento e de suas implementações, o algoritmo genético conta com 4 funções principais a Fitness, Crossover, Seleção e Mutação, enquanto a função gulosa necessita apenas de uma função de ordenação além de sua função principal, ou seja o custo do algoritmo genético acaba sendo muito maior, ao realizar algumas execuções é possível ver que o algoritmo guloso chega a uma solução mesmo com instâncias muito grandes, enquanto o algoritmo genético tenta convergir para uma solução.

- Experimento:

Os algoritmos foram submetidos a executar a seguinte instância mostrada na
figura-4

```
E:\Projetos\algoritmosII\Genetico\bin\Debug\Genetico.exe
informe o nome do arquivo que deseja enumerar
buffer

Conteudo do arquivo:

peso maximo: 150
qtd_elementos: 120
n_mochilas: 48
elementos :
42
69
67
57
93
90
38
```

mostra o peso da mochila, quantidade de elementos e o número de mochilas. figura-4

Ao realizar a execução dessa instância no algoritmo guloso foi possível observar como já se sabia, que ele não se preocupa com o problema de minimização das mochilas, informando um valor diferente ao dado pela instância que pode ser observado na figura-5.

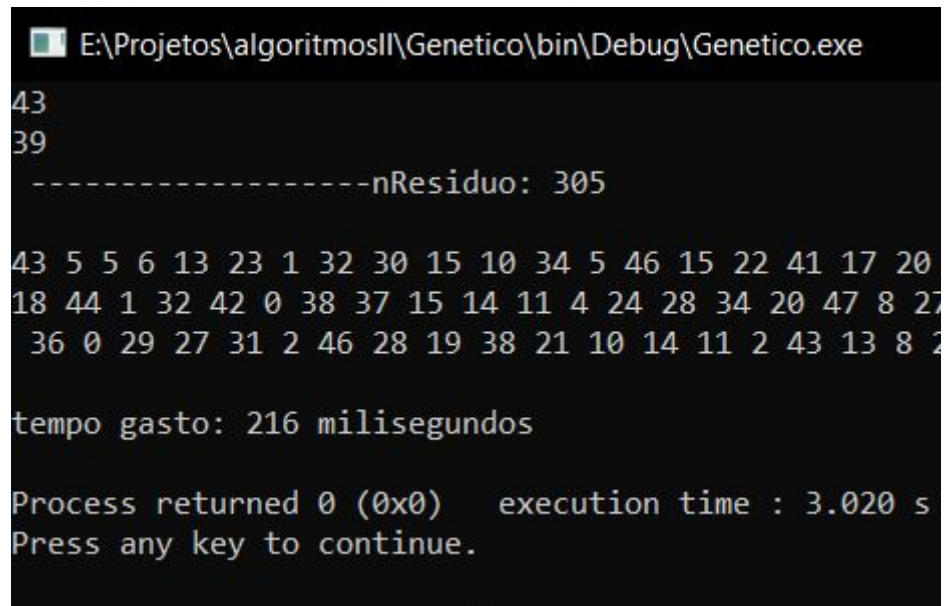
```
E:\Projetos\algoritmosII\algoritmo-guloso\guloso\bin\Debug\guloso.exe
20 23 23 24 25 25 26 27 27 28 29 30 30 30 32 32 33 33
43 43 43 44 44 45 45 46 46 47 49 49 49 50 55 55 55 57
0 71 72 73 73 73 74 74 76 78 78 78 78 79 79 80 80 8
93 94 96 96 98 98 98 o numero minimo de pacotes e:67
numero de iteracoes: 3016
tempo gasto: 2438 milisegundos

Process returned 0 (0x0)   execution time : 5.701 s
Press any key to continue.
```

informa o número de iterações o tempo e o número de mochilas usadas para empacotar os pesos. figura-5

Em contrapartida o algoritmo genético busca encontrar a configuração que se encaixe nas 48 mochilas, pois pode acontecer de o número de gerações se esgotarem e o

algoritmo não conseguir chegar na solução como mostra a figura-6.



```
E:\Projetos\algoritmosII\Genetico\bin\Debug\Genetico.exe
43
39
-----nResiduo: 305

43 5 5 6 13 23 1 32 30 15 10 34 5 46 15 22 41 17 20
18 44 1 32 42 0 38 37 15 14 11 4 24 28 34 20 47 8 27
36 0 29 27 31 2 46 28 19 38 21 10 14 11 2 43 13 8 2

tempo gasto: 216 milisegundos

Process returned 0 (0x0) execution time : 3.020 s
Press any key to continue.
```

mostra o resíduo, o tempo em milissegundos e a
solução atual. figura-6

O que nos leva a declarar como o algoritmo vencedor aquele que supre as nossas necessidades, em um contexto em que preciso de uma solução rápida e viável o algoritmo guloso é a melhor solução, mas em uma situação onde não há brecha para erros, não importando o tempo que irá levar para encontrar a solução, o algoritmo genético é a melhor escolha.

CONCLUSÕES E RECOMENDAÇÕES

O problema do empacotamento pode ser resolvido de várias maneiras possíveis, é importante ter em mente qual o seu objetivo para poder utilizar aquele algoritmo que melhor irá satisfazer suas necessidades, o problema de minimização de mochilas por outro lado não pode ser resolvido com abordagem gulosa, mas pode ser resolvido através de um algoritmo evolucionário.

REFERÊNCIAS