Universidade Federal de São Paulo – Campus São José dos Campos

Instituto de Ciência e Tecnologia

Modelagem Computacional

Simulação 5

O Problema da Mochila

DANILO AUGUSTO DIAS CARVALHO

EDUARD ERIC SCHARDIJN

RENAN ARANTES BERNARDES VIEIRA

Turma: B

Bacharelado em Ciência e Tecnologia

São José dos Campos, SP

Maio de 2012

Objetivos

O objetivo desta simulação é implementar três códigos diferentes que de formas distintas tentem otimizar o problema da mochila.

Os códigos deverão ser comparados quanto ao tempo de execução e eficiência na otimização.

Modelo Teórico

Na teoria da complexidade computacional, NP é o acrônimo em inglês para "Tempo polinomial não determinístico" (Non-Deterministic Polynomial time) que denota o conjunto de problemas que são decidíveis em tempo polinomial por uma máquina de Turing não-determinística.[1]

Usaremos nesse caso uma subclasse dos NP, pertencendo à classe de complexidade NP-completo de tal modo que todo problema em NP se pode reduzir, com uma redução de tempo polinomial, a um dos problemas NP-completo. Pode-se dizer que os problemas de NP-completo são os problemas mais difíceis de NP e muito provavelmente não formem parte da classe de complexidade P. A razão é que se se conseguisse encontrar uma maneira de resolver qualquer problema NP-completo rapidamente (em tempo polinomial, então poderiam ser utilizados algoritmos para resolver todos problemas NP rapidamente.[2]

É fácil de verificar se uma resposta é correta, mas não se conhece uma solução significativamente mais rápida para resolver este problema do que testar todos os subconjuntos possíveis, até encontrar um que cumpra com a condição, esse método é conhecido como método de "*força bruta*".

Na prática, o conhecimento de NP-completude pode evitar que se desperdice tempo tentando encontrar um algoritmo de tempo polinomial para resolver um problema quando esse algoritmo não existe.

Outro método muito usado para a otimização de problemas é o método determinístico, método que resolve a maioria dos métodos clássicos, na qual é gerada uma seqüência determinística de possíveis soluções requerendo, na maioria das vezes, o uso de pelo menos a primeira derivada da função objetivo em relação às variáveis de projeto. Nestes métodos, a função objetivo e as restrições são dadas como funções matemáticas e relações funcionais. Além disso, a função objetivo deve ser contínua e diferenciável no espaço de busca. [3]

Um exemplo desse método é o algoritmo guloso que, por sua vez, é "míope": ele toma decisões com base nas informações disponíveis na iteração corrente, sem olhar as consequências que essas decisões terão no futuro. Um algoritmo guloso jamais se arrepende ou volta atrás: as escolhas que faz em cada iteração são definitivas.[4] isso torna o processo de escolha muito mais rápido e eficiente porém abre margem para erros sendo sua analise de correção em muitos casos um tanto quanto complicada, por esse motivo são poucos os problemas que aceitam uma resolução desse tipo de forma totalmente correta.

Três abordagens serão feitas para resolver o problema de otimização combinatória ou problema da mochila, onde buscamos otimizar o valor agregado de objetos que podem ser colocados em uma mochila (espaço limitante), levando em consideração seu tamanho e valor.

Referências:

[1] http://pt.wikipedia.org/wiki/NP_(complexidade)

[2] http://pt.wikipedia.org/wiki/NP-completo

[3] http://www.maxwell.lambda.ele.puc-rio.br/7603/7603_4.PDF

[4] http://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/guloso.html

Modelagem Computacional

Adotou-se para os três códigos a metodologia de criação randômica de objetos, cujos valores máximos de peso e valor eram discretos e estavam no intervalo fechado de um a nove, com o peso máximo suporte da mochila de vinte. Salvo estes, citemos os códigos implementados:

1. *"Força Bruta" Randômica*;
2. *Algoritmo Guloso*;
3. *Método* n^2.

Comecemos uma análise particular para cada um destes.

Força Bruta Randômica

Inicialmente fora proposto que se implementasse o método de força bruta que consiste na tentativa de inserir todos os objetos em todas as ordens possíveis. Tal método, por abranger todo espaço amostral de possibilidades de combinações, terá necessariamente a solução ótima do problema. Entretanto, à medida que o número de

objetos aumenta, o tempo de execução do código torna-se demasiadamente grande a ponto de impossibilitar as simulações.

Este método é da ordem n! em que n é o número de objetos a serem analisados para estocagem na mochila. Devido a dificuldade encontrada em implementar tal método, foi proposto e implementado um alternativo. Este consiste no método de força bruta randômica, em que são gerados n! combinações possíveis entre os objetos, porém com a possibilidade de repetições. Logo este método não abrange todo espaço amostral e não satisfaz de forma ótima o força bruta (existe a possibilidade de em uma simulação, não agregar a solução ótima), porém para âmbito de tempo de simulação (que se trata da mesma ordem do força bruta) este código torna-se válido.

Código

O principal trecho deste código é:

```
//Gera o vetor de posicoes aleatorias
      for(i=0 ; i<objetos ; i++){
         flag = 0;
         while(flag==0){
            posicao = rand()%objetos;
            if(posicoes[i]==-1 && nao_existe(posicoes,posicao,objetos)){
               posicoes[i] = posicao;
               flag=1;
            }
         }
      }
```

Onde a função auxiliar nao_existe(), segue:

```
int nao_existe(int posicoes[], int valor, int obj){
   int i;
   for(i=0 ; i<obj ; i++){
      if(posicoes[i]==valor)
         return (0);
   }
   return(1);
}
```

No vetor posicoes[] são geradas as combinações randômicas que são utilizadas para inserção de objetos respeitando a ordem deste vetor. Vale salientar que este laço está contido em outro que executa estas permutações randômicas n! vezes.

Algoritmo Guloso

Nosso algoritmo guloso basicamente varre os objetos atribuindo uma relação de proporção entre valor/peso em que valores maiores correspondem a melhores objetos. Estes são ordenados e adicionados até que a mochila encha.

Código

Serão aqui descritos os principais trechos do algoritmo guloso implementado.

```
//O programa utiliza o flag para assegurar que todos os objetos tivessem a possibilidade de serem adicionados na mochila
while(flag){
    // Esse laço varre todos os objetos, comparando a relação valor/peso, atribuindo a variável a melhor posição do objeto com melhor relação valor/peso.
    for(i = 0; i < n_objetos; i++){
        if(w[i] !=0 && va < v[i]/w[i]){
            melhor = i;
            va = v[i]/w[i];
        }
    }

    //Esse laço adiciona o objeto selecionado verificando se não foi passado o peso limite da mochila.
    //Caso o objeto venha a ser adicionado ele recebe 0, sinalizando que este foi adicionado,  não podendo ser possível adicioná-lo novamente.
    if(peso_acumulado + w[melhor] <= pl){
        peso_acumulado+=w[melhor];
        valora+=v[melhor];
        adicionados[j] = melhor;
        j++;
        v[melhor] = 0;
        w[melhor] = 0;
        melhor = 0;
        va = 0;
    }

    //função do else: Caso o objeto não seja adicionado por que excedeu o peso limite, ele é zerado, notificando que já foi tentado ser adicionado na mochila.
    else{
        v[melhor] = 0;
        w[melhor] = 0;
        melhor = 0;
        va = 0;
    }
    flag = 0;
```

O laço de “for” varre todos os objetos verificando se estes foram zerados (objeto já foi tentado ser adicionado), sinalizando o flag.

```
    for(i = 0; i < n_objetos && !flag; i++){
```

```
        if(w[i] == 0)
            flag = 0;
        else
            flag = 1;
    }
}
```

Método n^2

Como proposta de uma outra abordagem ao problema da mochila, desenvolveu-se o método n^2 que consiste em uma lista encadeada em que cada nó representa um objeto contendo o valor, peso e uma variável lógica que se for 1, indica que o objeto está na mochila, e zero caso contrário.

O programa consiste em varrer todos os objetos de forma seqüencial adicionando um por um até que o suporte da mochila seja atingido. Quando atingido outra varredura começa com todos os objetos, entretanto começando com o próximo nó e assim por diante.

Este algoritmo executa n^2 passagens pelos nós, em que n é o número de objetos. Este executa apenas uma fração de todas as combinações possíveis, mas não deixa de ser uma alternativa para n muito grande, devido à força bruta ser inviável nestes casos.

Código

Uma função crucial para este código é o calculo do peso atual da mochila. Verifica-se se o peso atual mais o peso que se deseja inserir é menor ou igual ao peso suporte da mochila. Se sim, o objeto é inserido. Esta função segue:

```
//calcula e retorna o peso atual da mochila
float peso_mochila(Plista_circ l){
    float peso_parcial = 0;
    int i;
    Plista_circ p=l;
    do{
        if(p->flag!=0){
            peso_parcial += p->peso;
        }
        p = p->prox;
    }while(p!=l);
    return peso_parcial;
}
```

Outra função importante se trata da que esvazia a mochila, ou seja, desativa os flags para análises posteriores. Desta, segue:

```
//Esvazia a mochila (desativa os flags)
void esvaziar_mochila(Plista_circ l){
```

```
   int i;
   Plista_circ p=l;
   do{
      if(p->flag!=0){
         p->flag = 0;
      }
      p = p->prox;
   }while(p!=l);
}
```

No trecho seguinte é possível notar a chamada da função no método principal, bem como a dinâmica de rotação entre os nós-objetos.

```
      do{
         if(lista!=NULL)
            peso_parcial = peso_mochila(lista);
         if(peso_parcial + p->peso <= W) //se o peso que ja esta na mochila + o peso da
vez <= peso suporte
            p->flag = 1; //coloca o peso na mochila
      p = p->prox;
      }while(p != q);
(...)
      p = p->prox;
      q = p;
```

Simulações

As simulações tiveram o âmbito de comparar o tempo de execução dos códigos bem como sua eficiência de otimizar os valores da mochila, dado uma restrição de peso limitante. Para melhor visualização destas dinâmicas, para simulações arbitrárias foram coletados os dados referentes ao tempo e valores máximos obtidos e a partir destes foram plotados gráficos.

Vale ressaltar que devido a aleatoriedade de números para pesos e valores de cada objeto em simulações diferentes implica diretamente na forma dos gráficos aqui apresentados e estes visam inferir hipóteses quanto as execuções. Não necessariamente estas sendo verdadeiras, pois o espaço amostral para cada simulação é diferente.

Para o primeiro gráfico temos a relação entre o número de objetos e o tempo de execução em segundos do código para o algoritmo de força bruta randômica. Devido ao laço ser limitado por um fatorial, a medida que os objetos aumentam linearmente é notável o aumento do tempo de execução, que para n>20 por exemplo torna a simulação impraticável. A figura 2 ilustra os valores máximos obtidos na mochila para o algoritmo força bruta randômica. Observa-se grande flutuação do valor agregado e um baixo valor ótimo relativo associado. Isto se justifica por este método não ser exaustivo e por ser

executado para uma quantidade baixa de objetos, o que acarreta na baixa densidade de objetos ótimos (leves e caros).

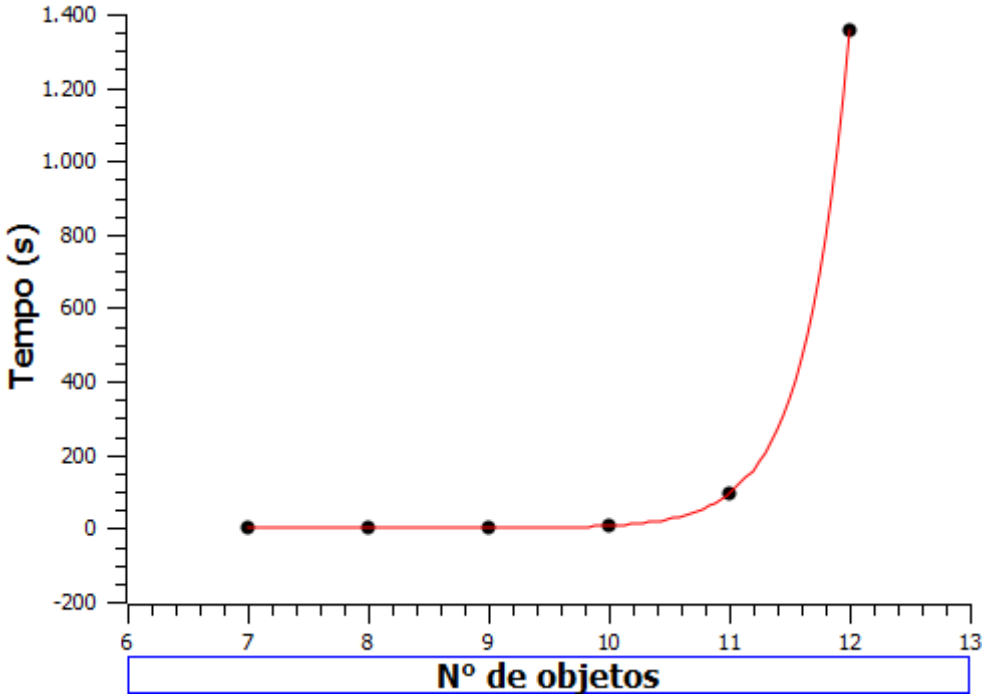

Figura 1 – Tempo de execução do código força bruta randômica.

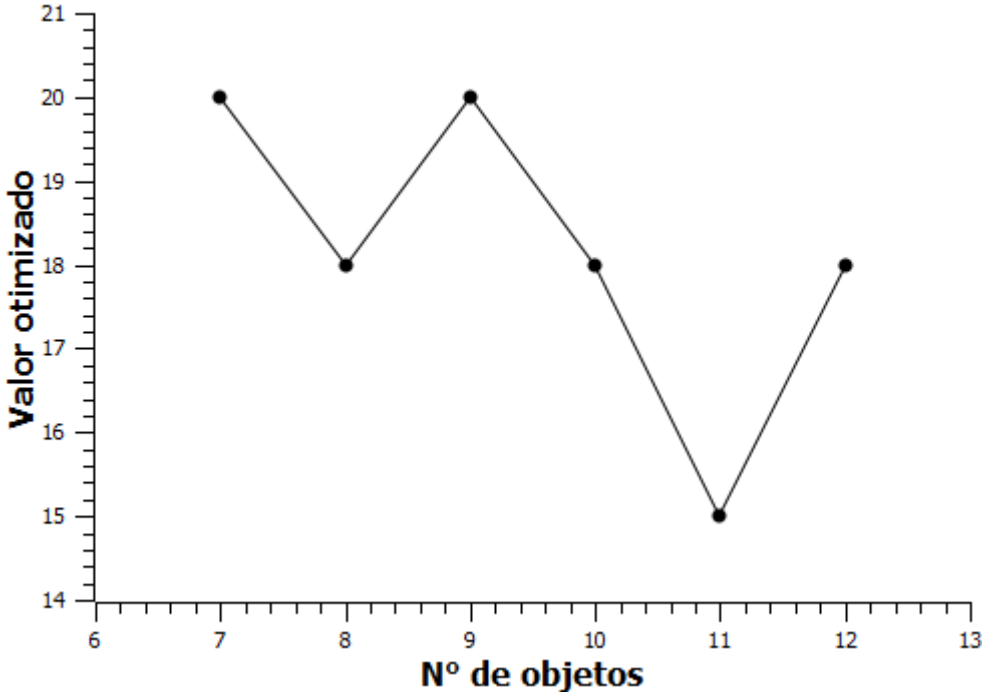

Figura 2 – Otimização do valor máximo agregado do código força bruta randômica.

Analogamente um estudo semelhante é feito com o algoritmo guloso. A figura 3 expressa a curva do tempo de execução em função do numero de objetos.

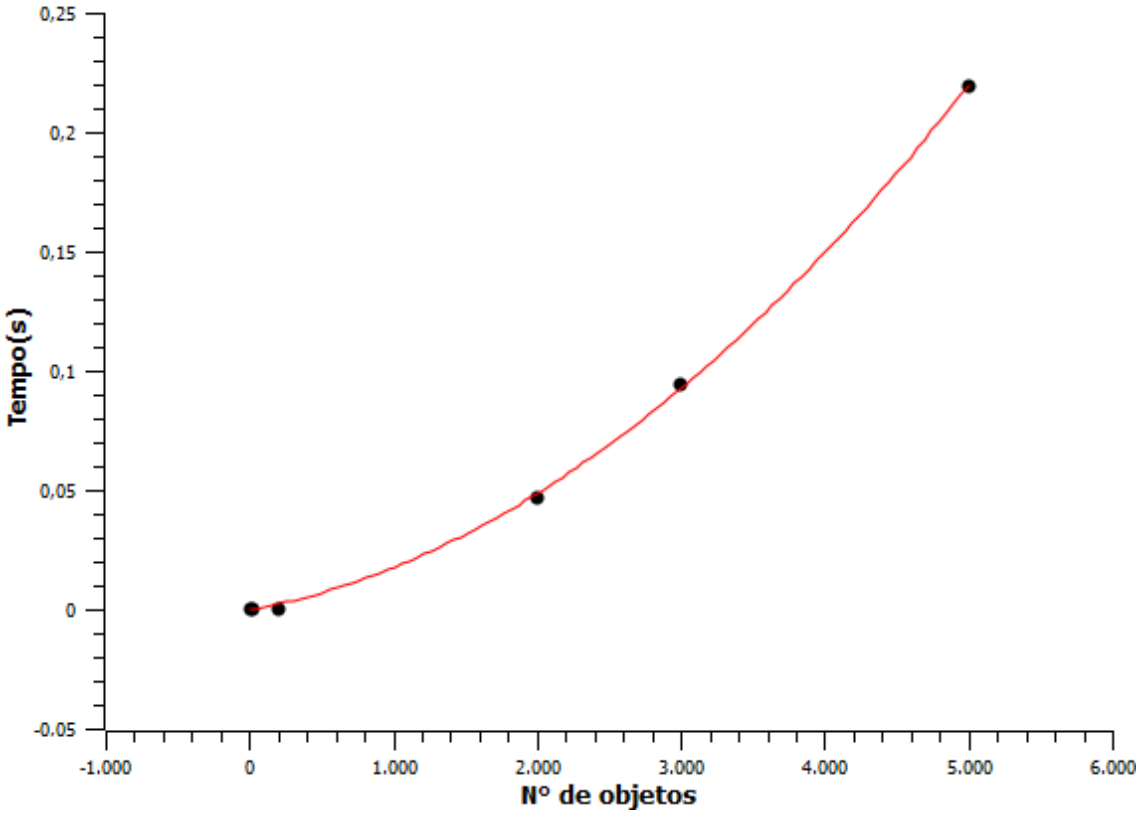

Figura 3 – Tempo de execução do código guloso.

Observa-se que o tempo aumenta a medida que o numero de objetos aumenta. Vale notar a escala referente ao tempo neste gráfico, onde execuções de milhares de objetos são feitas em menos de um segundo.

O gráfico referente a otimização do valor do algoritmo guloso apresenta um aumento de valor agregado com o aumento do numero de objetos. Isto se deve a maior possibilidade de que se tenham objetos com baixo peso e alto custo. Assim o algoritmo os seleciona e os incorpora à mochila.

Cabe ressaltar que a solução ótima do problema da mochila nos parâmetros de simulações impostas, ou seja, com valores e pesos de 1 a 9 e peso máximo da mochila de 20, temos que a solução ótima, seriam vinte objetos com peso um e valor nove, ou seja, o maior valor agregado se trata de $20 \cdot 9 = 180$. Isto é obtido para $n>3000$. Logo o algoritmo guloso para uma grande variedade de objetos é capaz de otimizar a mochila.

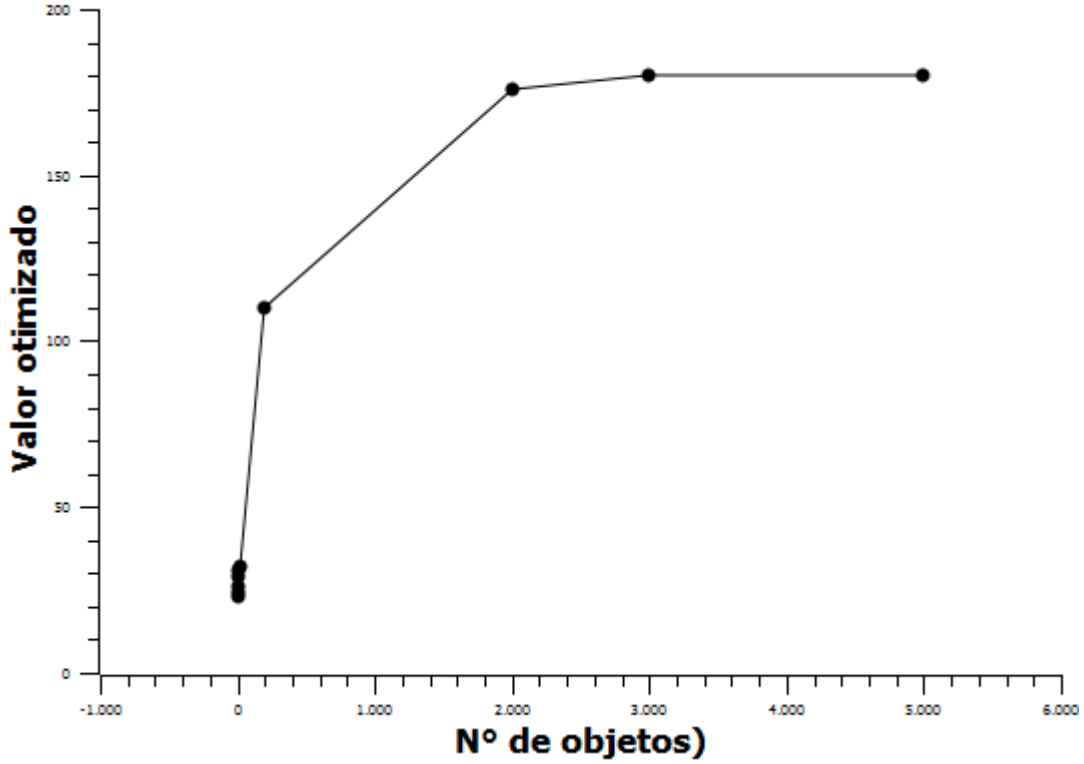

Figura 4 – Otimização do valor máximo agregado do código guloso.

O último método analisado se trata do n^2 que se mostrou um método intermediário ao força bruta randômica e o guloso. A figura 5 ilustra o gráfico referente ao tempo de execução deste código em função do numero de objetos.

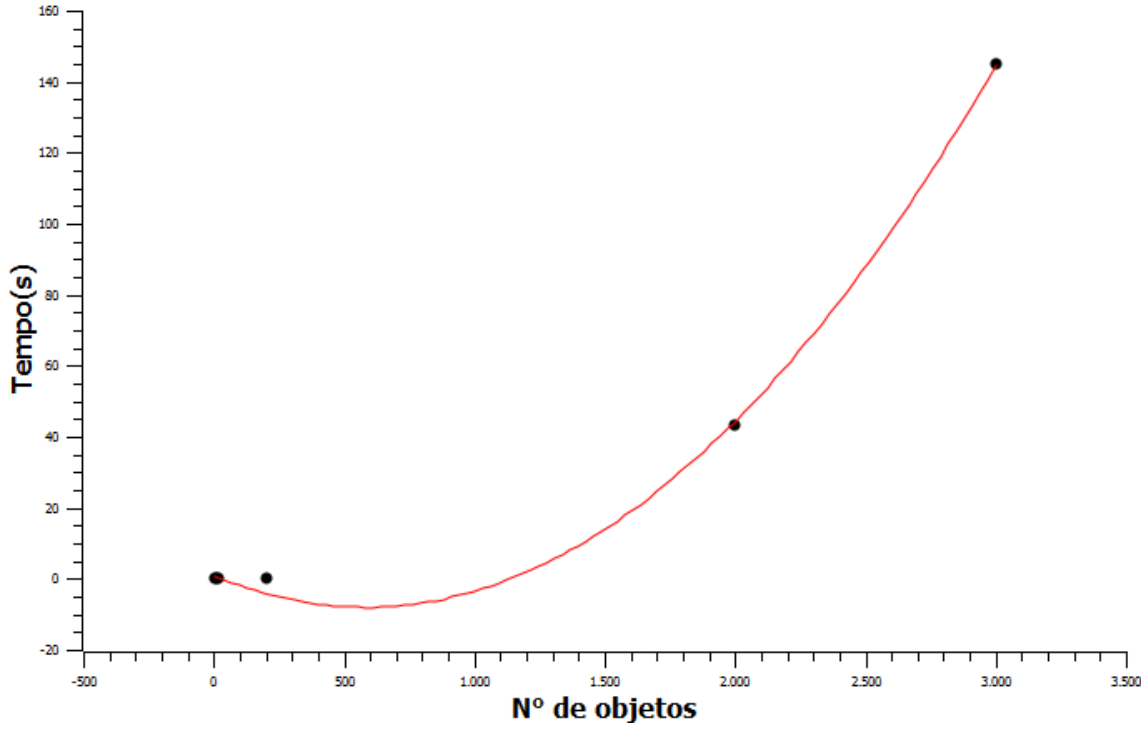

Figura 5 – Tempo de execução do código n^2.

A figura 5 ilustra a evolução do tempo de execução a medida que o numero de objetos aumenta. É notável que para um tempo factível, na ordem de minutos, são simulados milhares de verificações de objetos. Este modelo não abrange todo espaço amostral, porém atende a simulação em um tempo aceitável.

A figura 6 ilustra a relação do método n^2 em função do máximo valor agregado obtido.

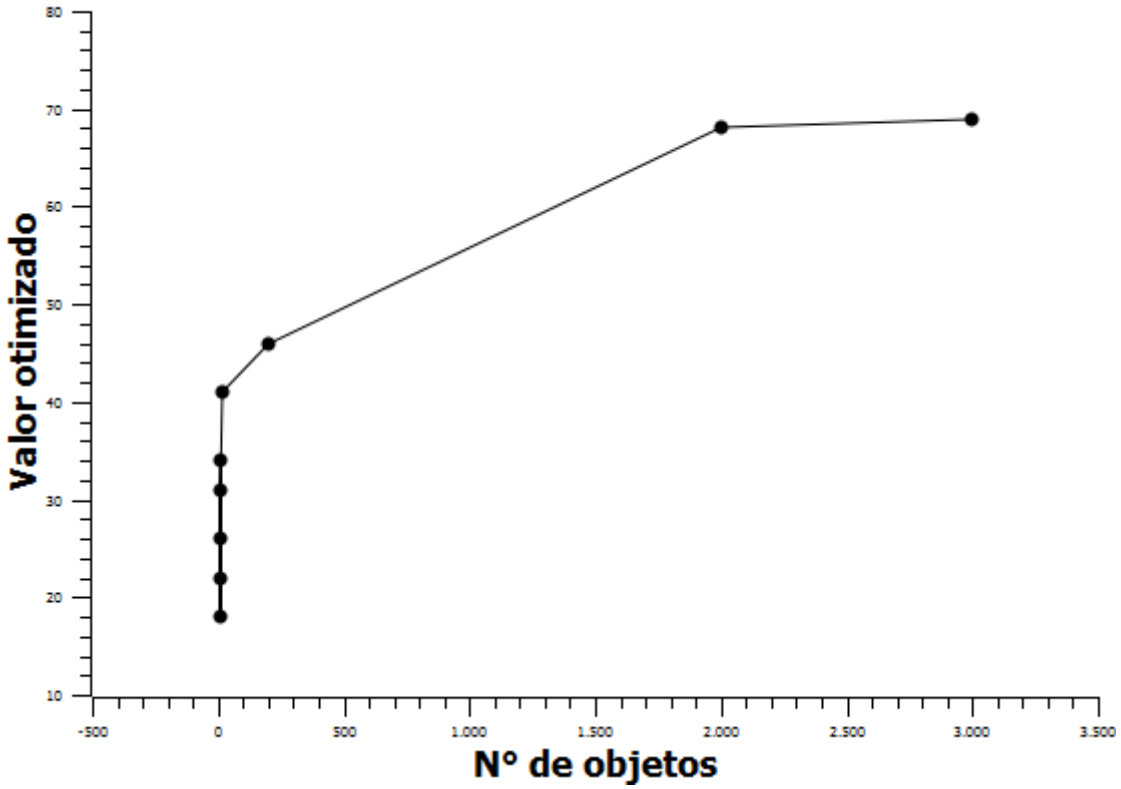

Figura 6 – Otimização do valor máximo agregado do código n^2.

Como era de se esperar o valor agregado aumenta em função do numero de objetos, pelos mesmos motivos já explicitados. Nota-se que para n pequeno, $n<100$, este método se demonstrou muito eficaz.

Uma última discussão acerca dos dados obtidos consiste na ineficiência do algoritmo força bruta randômica. Para tanto confeccionou-se o gráfico 7 que explicita os três códigos para os mesmos números de objetos. Neste gráfico observa-se a flutuação de otimização de valor para os três códigos e isto se deve ao baixo número de objetos disponíveis e a distribuição randômica. Assim, para poucos objetos, aumentando a quantidade destes não necessariamente o valor agregado irá aumentar, pois são outros objetos com valores distintos e não necessariamente melhores que os anteriores.

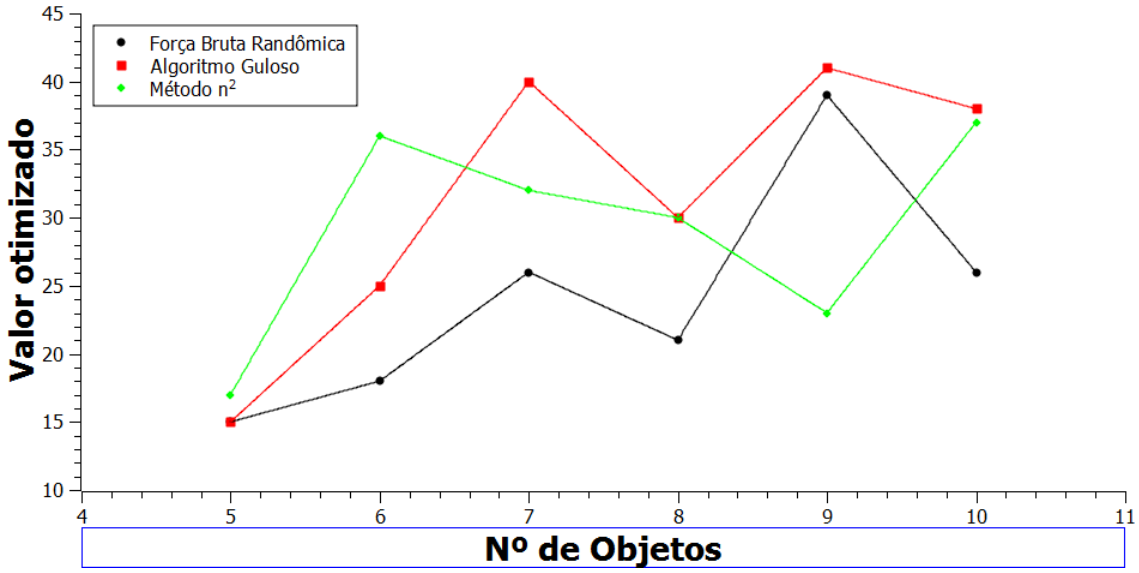

Figura 7 – Valor máximo para os três códigos com o mesmo valor de n.

Para o código de força bruta randômica, o tempo de simulação torna-se extremamente alto com o aumento do número de objetos, sendo doze o máximo de objetos testados. Logo não é possível comparar este método com os demais diretamente quanto a otimização de valores, quando os outros códigos são executados com milhares de objetos.

A figura 7 tenta compará-los com a mesma quantidade de objetos para simulações arbitrárias com objetos arbitrários. Acredita-se que o força bruta randômica possa conter a solução ótima por aleatoriedade, mas demonstra-se ineficaz se comparado aos outros métodos, para $n \leq 10$ como visto no gráfico anterior.

Conclusões

A partir do estudo feito no presente relatório, foi possível verificar a complexidade do Problema da Mochila que pode ser estendida a outros problemas semelhantes como o do caixeiro viajante por exemplo.

Notou-se a inviabilidade de execução de códigos n! para n grande devido ao tempo de simulação e fez-se necessário outras abordagens do problema mesmo que a solução fosse sub-ótima.

Por fim, com os códigos desenvolvidos e análises destes, pode-se afirmar que o algoritmo guloso demonstrou-se o melhor, devido ao baixo tempo de execução e pela obtenção da solução ótima do problema.