O Jogo dos Diamantes - Algoritmos 1

Matheus Aquino Motta[1]

[1]Bacharelado em Matemática Computacional
DCC - Universidade Federal de Minas Gerais

matheusmotta@dcc.ufmg.br

Abstract. *In this documentation we will explain in detail the approach used to solve the Diamond's Game, where the main problem have had been reduced into a Knapsack 0-1 optimization program, and give efficient Dynamic Programming solution for it, given the statement constraints. Furthermore, explaining from mathematical aspects to implementation decisions while comparing our results with a naive Brute Force solution, from small inputs to feasible large ones.*

Resumo. *Nessa documentação iremos explicar detalhadamente a abordagem utilizada para solução do problema O Jogo dos Diamantes, onde o problema central foi reduzido a um problema de otimização equivalente a Mochila 0-1, e apresentamos um algoritmo eficiciente de Programação Dinâmica, dadas as restrições de entrada do problema. Assim, explicaremos a abordagem desde aspectos matemáticos à decisões de implementação e comparando nossos resultados com uma solução ingênua alternativa de força bruta, para entradas pequenas e grandes entradas viáveis.*

1. Introdução

O problema proposto nesse trabalho consiste em ajudar Adriana a resolver *O Jogo dos Diamantes*. O objetivo do Jogo dos Diamantes é bem simples: dado um malote M com $n \leq 128$, diamantes e seus respectivos pesos $0 \leq w_i \leq 128$, para $0 \leq i \leq n-1$, queremos determinar o mínimo valor que pode ser obtido a partir do seguinte processo:

Enquanto o número de diamantes no malote $|M| > 1$, tomamos dois diamantes i, j, destruimos os diamantes um contra um outro de modo que

- Se $|w_i - w_j| = 0$, então os diamantes são completamente destruídos.
- Caso contrário, sobra um diamante k, tal que $w_k = |w_i - w_j|$, no malote.

Assim, queremos que ao final do execução do algoritmo nossas decisões tenham sido ótimas e o peso do diamante $w_f > 0$ restante no malote seja mínimo, ou que por sua vez, não existam mais pedras, isto é, $w_f = 0$.

Desse modo, é perceptível que *O jogo dos Diamantes* consiste em um problema de otimização, onde queremos encontrar o menor inteiro $w_f \geq 0$ possível como resultado, ou seja, a escolha dos diamantes dois a dois deve ser ótima.

Podemos observar que devido às dimensões da entrada do problema, abordagens ingênuas de força bruta sobre todas as possibilidades não irão garantir uma solução escalável em tempo de execução e memória para o problema. Portanto, iremos utilizar técnicas de programação dinâmica para atacar esse problema de decisão, para alcançar resultados ótimos de forma mais eficiente e robusta.

Destarte, a abordagem utilizada para resolução do problema consiste na observação de que podemos reduzi-lo à uma Mochila 0-1, onde a capacidade C da nossa mochila é dada em função da soma dos pesos dos diamantes, tal que

$$C = \frac{1}{2}\sum_{i=0}^{n-1} w_i$$

Com isso, iremos demonstrar as decisões teóricas para tal abordagem e múltiplas das opções de implementação existentes para solucionar o problema.

2. Fundamentos Básicos

Para entender as decisões de implementação utilizadas no trabalho prático, precisamos discutir qual o que é o problema da Mochila 0-1 e como podemos reduzir O jogo dos Diamantes em um problema análogo.

2.1. Mochila 0-1

Um dos mais famosos problemas em otimização combinatória, conhecido como o o problema da mochila, que por sua vez é um problema *NP-Hard*, que pode ser classificado como uma programação linear inteira.

Assim, ilustramos o problema da mochila como um cenário no qual dado um conjunto S de n elementos $e_0, e_1, \ldots, e_{n-1}$, onde cada elemento i, tal que $0 \leq i \leq n-1$, possui a ele associado um peso w_i e um valor v_i. Desse modo, dada uma mochila de capacidade limitada C, queremos preencher nossa mochila a partir de um subconjunto de elementos $H \subseteq S$, tal que, $\sum_{i \in H} v_i$ seja máximo e $\sum_{i \in H} w_i \leq C$. Reescrevendo, temos o seguinte problema de otimização,

$$\begin{aligned} \text{maximize} \sum_{i=0}^{n-1} v_i x_i \\ \text{subject to} \sum_{i=0}^{n-1} x_i w_i \leq C \\ x \in \{0,1\}^n \end{aligned}$$

A partir dessa formulação dada pela Programação Linear Inteira, podemos extrair um algoritmo simples para computar a solução ótima do problema da mochila, de complexidade $O(n2^n)$. Onde basta computar para cada string $x \in \{0,1\}^n$, a soma obtida em relação ao peso e valor, e armazenar o $\arg\max_x \sum_{i=0}^{n-1} v_i x_i$, tal que o peso esteja dentro das restrições estabelecidas.

É possível melhorar essa complexidade a partir de uma abordagem recursiva, derivada da expansão do problema em uma equação de recorrência, na qual para cada elemento $0 \leq i \leq n-1$, tomamos a decisão de adicionar ou não o i-ésimo elemento à mochila.

$$\mathrm{T}(i, C) = \begin{cases} \mathrm{T}(i+1, C), & \text{Se } w_i > C \\ \max(\mathrm{T}(i+1), \mathrm{T}(i+1, C - w_i) + v_i), & \text{Caso contrário} \end{cases}$$

Em uma complexidade da ordem de $O(2^n)$.

Entretanto, podemos melhorar esse algoritmo a partir da observação de que essa recorrência pode apresentar sobreposição de subproblemas $T(i, c)$, onde o estado da recorrência (i, c) pode aparecer múltiplas vezes em nossa árvore de recursão, o que resultaria em computações redundantes para as quais já obtemos o resultado anteriormente.

Assim, utilizaremos técnicas de Programação Dinâmica para otimizar a complexidade do algoritmo para resolver o nosso problema, a partir do armazenamento do resultado para os subproblemas $T(i, c)$. Dessa forma, temos uma complexidade de tempo da ordem de $O(nC)$, haja vista que cada peso C possível terá sua árvore de recorrência pesquisada apenas uma vez, para cada um dos n elementos.

Iremos utilizar esse problema de programação para encontrar a solução ótima para O Jogo dos Diamantes, onde serão apresentadas 3 soluções alternativas para o problema: Brute Force, Programação Dinâmica *Top Down* e Programação Dinâmica *Bottom Up*.

2.2. Divisão em conjuntos

Nesse sentido, agora precisamos entender como utilizar o Problema da Mochila 0-1 para resolver O Jogo dos Diamantes, haja vista que, esta redução não é explícita no problema.

Seja $S = w_0, w_1, \ldots, w_{n-1}$ um conjunto arbitrário de diamantes, assumindo que para cada par (i, j) temos que $i \geq j$,

$$
\begin{aligned}
S &= w_0, w_1, \ldots, w_{n-1}, \\
&= w_0, w_1, \ldots, w_{n-3}, w_{k_0}, && \text{Para } w_{k_0} = w_{n-2} - w_{n-1}, \\
&= w_0, w_1, \ldots, w_{n-4}, w_{k_1}, && \text{Para } w_{k_1} = w_{n-3} - w_{k_0}, \\
&\vdots \\
&= w_0, w_{w_{n-3}}, && \text{Para } w_{k_{n-3}} = w_1 - w_{k_{n-4}}, \\
&= w_{k_{n-2}}, && \text{Para } w_{k_{n-2}} = w_0 - w_{k_{n-3}}
\end{aligned}
$$

Por fim, $w_{k_{n-2}}$ nos da o valor do diamante final após $n - 1$ destruições de diamantes. Assim, tomando $D = w_{k_{n-2}}$ expandindo de forma reversa temos que,

$$
\begin{aligned}
D &= w_0 - w_{k_{n-3}} \\
&= w_0 - w_1 + w_{k_{n-4}} \\
&= w_0 - w_1 + w_2 - w_{k_{n-5}} \\
&\vdots \\
&= w_0 - w_1 + w_2 - w_3 + w_4 - \cdots \pm w_{n-3} \pm w_{k_0} \\
&= w_0 - w_1 + w_2 - w_3 + w_4 - \cdots \pm w_{n-1} \\
&= \sum_{i=0}^{n-1} (-1)^n w_i \\
&= \sum_{i=0}^{n-1} a_i w_i - \sum_{i=0}^{n-1} b_i w_i \\
&= A - B
\end{aligned}
$$

Para $a_i = \frac{(-1)^i + 1}{2}$ e $b_i = \frac{(-1)^{i+1} + 1}{2}$.

Destarte, seja $T = \sum_{i=0}^{n-1} w_i$ a partir de substituições apropriadas podemos derivar a seguinte relação

$$\begin{aligned} T &= A + B \\ D &= A - B \\ &= T - 2B \end{aligned}$$

Logo, temos que D será mínimo quando B for máximo, onde

$$B \leq \frac{1}{2} \sum_{i=0}^{n-1} w_i$$

Portanto, podemos reecrever o problema do Jogo dos Diamantes, como uma mochila 0-1 definida pela seguinte programação, onde $C = \frac{1}{2} \sum_{i=0}^{n-1} w_i$.

$$\begin{aligned} \text{maximize} \sum_{i=0}^{n-1} w_i x_i \\ \text{subject to} \sum_{i=0}^{n-1} x_i w_i \leq C \\ x \in \{0,1\}^n \end{aligned}$$

Então, seja $R = \arg\max_x \sum_{i=0}^{n-1} w_i x_i$, sob as respectivas restrições da programação linear inteira acima, a resposta ótima será dada por

$$D = T - 2R$$

E sabendo que R é obtido a partir de um algoritmo que garante o máximo da programação linear inteira, então essa abordagem para O Jogo Dos Diamantes é ótima.

3. Implementação

Agora que discutimos em detalhe os fundamentos teóricos que basearam os algoritmos implementados, iremos descrever brevemente as soluções apresentadas, estruturas dedos utilizadas e complexidade de tempo e espaço das diferentes abordagens.

3.1. Abordagem por força bruta

Como já discutido na Sessão 2.1, a solução por força bruta pode ser extraída diretamente da programação linear inteira do nosso problema com complexidade de tempo da ordem de $O(n2^n)$.

Entretanto, utilizando a equação de recorrência e definindo a recursão a partir de casos bases e casos de parada apropriados, como exibido pelo *Algorithm 1* abaixo.

$$\mathrm{T}(i, C) = \begin{cases} \mathrm{T}(i+1, C), & \text{Se } w_i > C \\ \max(\mathrm{T}(i+1, C), \mathrm{T}(i+1, C - w_i) + v_i), & \text{Caso contrário} \end{cases}$$

Temos que o nosso resultado pode ser obtido em uma complexidade de tempo da ordem de $O(2^n)$, e de espaço auxiliar $O(1)$, a menos da pilha de recursão. Assim, essa abordagem seria ineficiente para entradas suficientemente grandes.

3.2. Abordagem com Programação de Dinâmica

Como já discutido, sabemos que a abordagem por força bruta é ineficiente devido à existência de sobreposição de subproblemas. Assim, para melhorar a eficiência do nosso algoritmo utilizaremos técnicas de memoization para armazenar a solução de subproblemas já resolvidos.

3.2.1. *Top Down*

A abordagem *Top Down* consiste em uma abordagem similar a solução por força bruta, entretando iremos criar uma matriz de memoization memo[][] de dimensões $N \times M$. Onde N é o número de diamantes dado como entrada e $M = \frac{1}{2}\sum_{i=0}^{n-1} w_i$, peso máximo que pode ser carregado na mochila.

Assim, a partir da equação de recorrência já conhecida, obtemos o seguinte o algoritmo, que possui como resultado a variável $result$.

Algorithm 1 Abordagem Top Down - O Jogo dos Diamantes

```
n ← input
w_i ← input, para todo i, tal que 0 ≤ i ≤ n − 1
memo[N][M] ← −1, para toda entrada i, j da matriz
procedure DP(i, cap)
    if cap < 0 then
        return −INF
    end if
    if i == n then
        return 0
    end if

    curr ← &memo[i][cap]
    if curr != −1 then
        return curr
    end if

    return curr ← max(DP(i + 1, c), DP(i + 1, c − w[i]) + w[i])
end procedure

S ← Σ_{i=0}^{n−1} w_i, soma dos diamantes
result ← S − 2 DP(0, S/2)
```

Seguindo a ideia da equação de recorrência, para cada elemento i do nosso vetor de diamantes, dada uma capacidade cap, verificamos dois casos de parada na execução:

1. Se cap está de acordo com as restrições de peso da mochila, caso contrário, retonamos um valor infinitamente pequeno para os propóstos do problema que será descartado pelo algoritmo no *Backtracking*.
2. Além disso, caso não existam mais elementos a serem pesquisados, i.e, $i = n$, então temos uma resposta vivável, e retornamos um valor nulo que não afetará o resultado do caminho da árvore de recursão do algoritmo.

Caso o estado da nossa recursão não defina uma parada para o algoritmo verificamos se já computamos a resposta para o subproblema pegando por referência a entrada da matriz definida por DP(i, cap), se sim, retornarmos o valor da entrada da matriz associada ao estado da recorrência, caso contrário, realizamos a chamada da equação de recorrência dada por

$$\text{DP}(i, cap) = \max(\text{DP}(i+1, cap), \text{DP}(i+1, cap - w_i) + w_i)$$

Ao final, como já discutido anteriormente, basta tomarmos o resultado do algoritmo principal R, e obtermos o resultado ótimo para o problema dado por $D = S - 2R$

O algoritmo de força bruta é trivialmente obtido a partir dessa mesma ideia ao remover o código associado a matriz de memoização memo[][].

Assim, seja n o número de elementos e $C = \frac{1}{2}\sum_{i=0}^{n-1} w_i$ obtemos a solução do problema utilizando uma complexidade de tempo da ordem de $O(nC)$ no pior caso e complexidade auxiliar de memória da ordem de $O(nC)$, dada pelas dimensões da matriz de memoização, a menos da pilha de execução oriunda da recursão do algoritmo utilizado.

3.2.2. *Bottom Up*

Podemos observar que o algoritmo *Top Down* funciona de modo que o estado definido por DP(i, cap) depende exclusivamente dos estados DP$(i+1, cap)$ e DP$(i+1, cap - w_i)$. Analisando essa dependência de outra forma, podemos extrair que de modo análogo, o estado DP(i, cap), depende exclusivamente dos estados DP$(i-1, cap)$ e DP$(i-1, cap - w_i)$.

Desse modo, percebemos que um estado pode ser obtido a partir de informações contidas na matriz de memoização de forma iterativa, inicializando a matriz de memoização para o primeiro elemento $i = 0$ e todas as capacidades possíveis $0 \leq j \leq C$ com aquele elemento, de forma apropriada.

Entretanto, ainda podemos observar que o estado (i, j) depende apenas de estados da linha $i - 1$ da matriz, portanto precisamos apenas de um vetor para realizar a nossa memoização e sobrescreve-lo apropriadamente. Assim, o valor R será computado pelo seguinte algoritmo.

Algorithm 2 Abordagem Bottom Up - O Jogo dos Diamantes

```
n ← input
w_i ← input, para todo i, tal que 0 ≤ i ≤ n − 1
S ← Σ_{i=0}^{n−1} w_i, soma dos diamantes
memo[S] ← 0, para todo j, tal que 0 ≤ j ≤ M
for 0 ≤ i ≤ n − 1, i do
    for S/2 ≥ j − w_i ≥ 0 do
        memo[j] ← max(memo[j], memo[j − w_i] + w_i)
    end for
end for
result ← S − (2 × memo[S/2])
```

A partir da noção de dependência de estados do algoritmo o vetor foi inicializado com todas as entradas memo[j] $= 0$, de modo que, para elemento nenhum, é possível carregar nenhum peso na mochila.

Com isso, para cada elemento $0 \leq i \leq n-1$, percorremos as capacidades $0 \leq j \leq C$, nas quais é possível adicionar o i-ésimo elemento à mochila, i.e, $j - w_i \geq 0$ de modo que, assim como na versão *Top Down*, o algoritmo toma a decisão a partir da equação de recorrência, e sobrescrevemos o vetor da seguinte forma

$$\text{memo}[j] = \max(\text{memo}[j], \text{memo}[j - w_i] + w_i)$$

Onde o primeiro fator é dado por não adicionar o elemento na mochila, i.e, quanto eu já era capaz de carregar sem o elemento w_i com capacidade j, memo[j]. Enquanto o segundo é dado por adicionar o elemento na mochila, i.e, dado por quanto eu era capaz de carregar com a capacidade memo[$j - w_i$], mais o novo valor que estou adicionando w_i.

Assim, obtemos o resultado para *O Jogo do Pulo* por meio de uma programação dinâmica *Bottom Up*, com complexidade de tempo da ordem de $O(nC)$ e de espaço auxiliar da ordem de $O(C)$. Enquanto utilizando a matriz similarmente utilizariamos um espaço auxiliar da ordem de $O(nC)$.

A abordagem *Bottom Up* torna-se mais relevante quando comparada a *Top down* devido ao fato de que já sabemos quais estados precisam ser computados e não utilizaremos a pilha de recursão, o que resultará em um tempo de execução inferior e uso limitado de memória do *stack*.

No que tange estritamente a implementação *Bottom up*, ambas implementações usando memoização com matriz e vetor foram disponibilizadas. A opção de usar vetores torna-se relevante devido à economia de memória e simplicidadade do algoritmo. Entretanto, sob uma circunstancial necessidade de recuperação da resposta, i.e, descobrir quais diamantes foram adicionados à mochila, teriamos um problema de resolução não trivial, enquanto utilizando a matriz conseguiriamos realizar esse processo facilmente.

4. Análise de Complexidade

Agora que entendemos as diferentes abordagens para resolução do problema. Iremos expor brevemente detalhes das expostas análises de complexidade.

Para a resolução por força bruta temos diretamente que a resposta é extraída por um algoritmo de complexidade de tempo da ordem $f(n) = O(2^n)$ e espaço auxiliar $O(1)$, haja vista que visitamos toda a árvore de recursão do algoritmo, e para cada i-ésimo diamante $0 \leq i \leq n-1$ temos a opção de adiciona-lo ou não à mochila, sobre as restrições de capacidade.

Enquanto isso para as resoluções por Programação Dinâmica temos uma complexidade de tempo dar ordem de $f(n) = O(nC)$,seja para abordagem *Bottom Up*, seja para a *Top Down*, a variar pelo tempo de execução da pilha de recursão para o método *Top Down*. Essa complexidade é evidente devido pelos *loops* de iteração do método *Bottom Up* onde visitamos exatamente uma vez cada um dos estados definidos pela matriz. Enquanto para o método *Top Down*, caso já tenhamos resolvido um sub-problema, apenas retornamos o valor já calculado. e obtemos a complexidade de tempo esperada.

A complexidade de espaço auxiliar para o método *Top Down* é da ordem de $O(nC)$, dada pelas dimensões da matriz de memoização. Enquanto para a abordagem *Bottom Up* temos uma complexidade de espaço auxiliar da ordem de $O(nC)$ ou $O(C)$, a depender da decisão de implementação, memoização por matriz ou vetor, respectivamente.

5. Análise Experimental

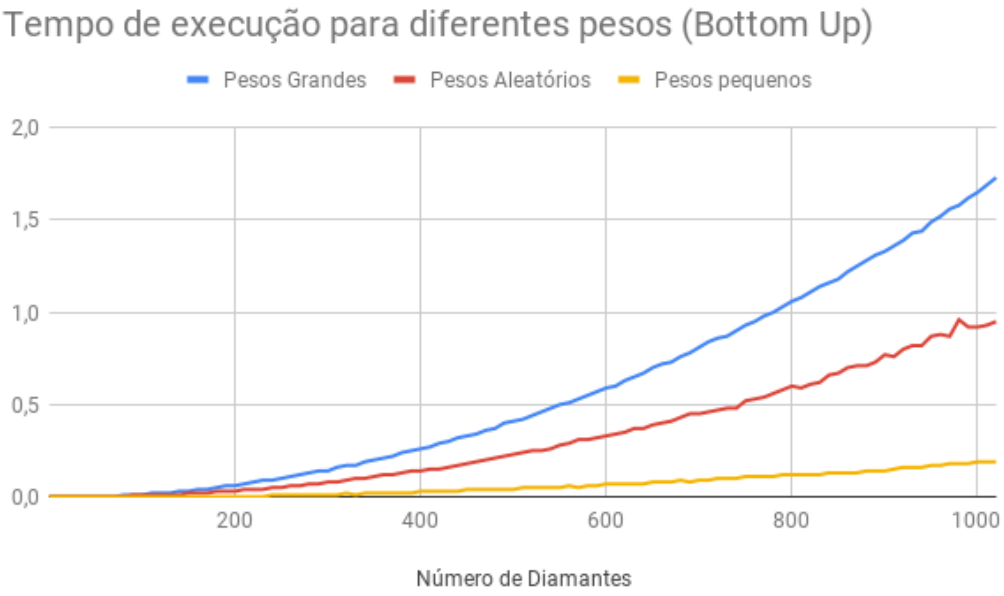

Figure 1. Tempo de execução do algoritmo com Programação Dinâmica para diferentes pesos.

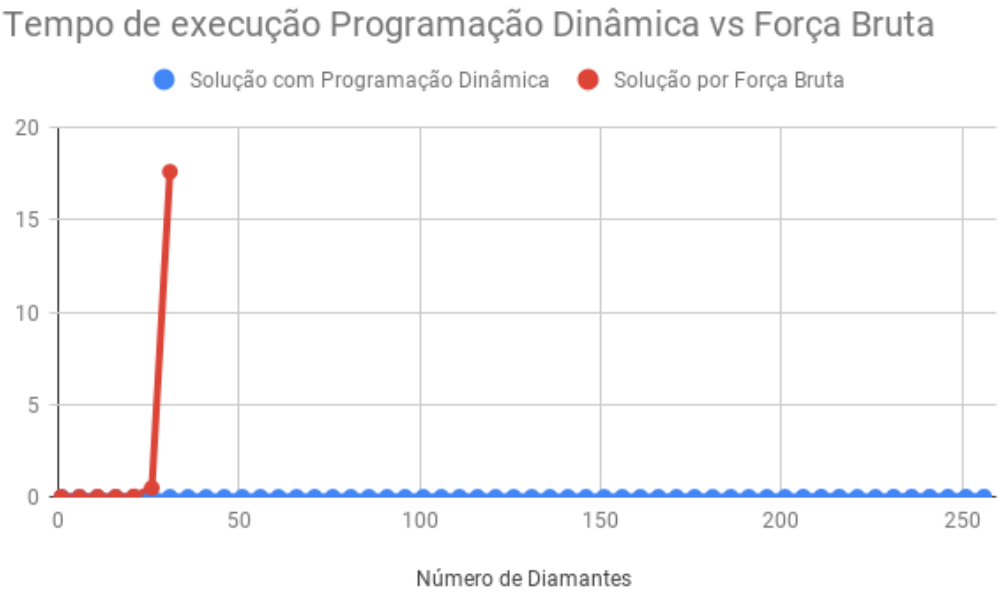

Figure 2. Comparação do tempo de execução do algoritmo com Programação Dinâmica vs Força Bruta.

Podemos verificar que o Desvio Padrão aumenta quando o valor de n aumenta, mas mais importante, podemos visualizar o comportamento da função ao aumentar o número de diferentes valores para o peso C, o que corrobora às nossas assumpções teóricas acerca da complexidade do algoritmo implementado.

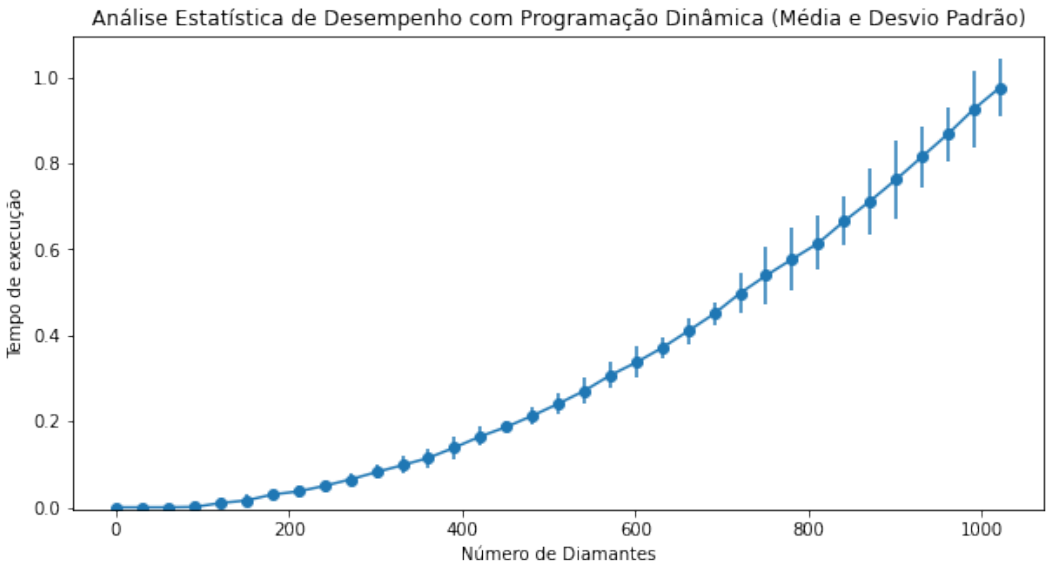

Figure 3. Desvio padrão e média do tempo(s) de execução do algoritmo.

Então, podemos concluir que análise experimental possui resultados assim como o esperado pela análise de complexidade teórica.

Os testes foram realizados em um Intel(R) Core(TM) i5-6400 CPU @ 2.70GHz 8GB RAM.

6. Instruções de compilação e execução

Para a execução do programa basta a utilização do *Makefile* presente no arquivo e a execução do comando *make* no diretório. Versões alternativas do algoritmo também foram disponibilizadas.
O algoritmo foi implementado e testado em um ambiente linux Ubuntu

Distributor ID: Ubuntu
Description: Ubuntu 18.04.2 LTS
Release: 18.04
Codename: bionic

Utilizando o compilador *g++ -std=c++11*.

7. Conclusões

A partir da resolução desse trabalho foi possível utilizar técnicas de Programação Dinâmica e suas variações para a resolução de um problema da mochila implícito em um cenário inesperado. E assim, entender como utilizar técnicas de programação dinâmica para problemas de otimização e redução de problemas para encontrar soluções ótimas para problemas não triviais de forma indireta.