# Algoritmos e Estrutura de Dados III (2019-1)

Trabalho Prático 2

Heitor Lourenço Werneck
heitorwerneck@hotmail.com

2 de Maio de 2019

## 1 Introdução

Um dos problemas dos pesquisadores de saúde é a análise de imagens oriundas da placenta de bebês, a partir da análise dessas imagens e identificação de certos elementos é possível prevenir uma enorme quantidade de doenças que o bebê poderia ter ao longo de sua vida. O problema pode ser modelado e simplificado como uma matriz $(a_{ij})_{n\times n}$ tal que $a_{ij} \in \{0(\text{Material líquido}), 1(\text{Material sólido})\}$, o objetivo é encontrar a maior submatriz $(s_{ij})_{m\times m}$ com $m \leq n$ sendo $s_{ij} = 1$ para $i = 1, ..., m \wedge j = 1, ..., m$.

A solução é apresentada com força bruta(tentativa e erro), estratégia gulosa, programação dinâmica e *branch and bound*.

A entrada e saída do programa segue o seguinte padrão:

```
Entrada    Saída

           a b c
1 1 1      a   1 1
0 1 1  >>  b   1 1
0 0 0      c
```

## 2 Estruturas de Dados

Deve existir uma variável para guardar o problema, essa variável será definida como:

```
int8_t **m;
```

Foi feito uma estrutura de dados utilizada para auxiliar na solução do problema que descreve o resultado, ou seja uma matriz $(s_{ij})_{m\times m}$ onde $s_{ij} = 1$ para $i = 1, ..., m \wedge j = 1, ..., m$. Sua definição é a seguinte:

```
    struct square{
      int order,i,j;
    };
typedef struct square Square;
```

O padrão de saída de todos paradigmas será essa estrutura.(figura 1)

Como a estrutura não contem os dados de quais cantos englobam a solução, então é utilizado implicitamente a informação de que a solução engloba a partir do ponto i e j da submatriz quadrada até i+order e j+order. Veja na figura 2.

## 3 Solução

A primeira solução a ser abordada será a de força bruta com tentativa e erro que irá explorar todas possibilidades exaustivamente e garante uma solução ótima.

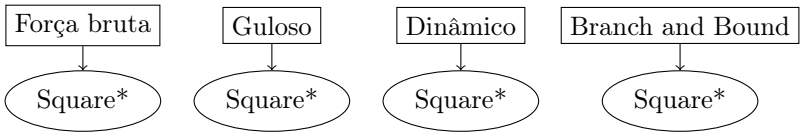



**Figura 1:** Padronização da saída.

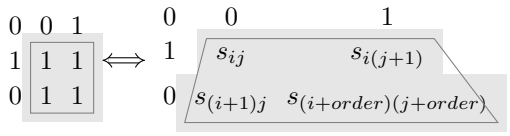



**Figura 2:** Representação dos dados da estrutura Square.

## 3.1 Força bruta (tentativa e erro)

A proposta de força bruta deve explorar todas possibilidades de submatrizes com 1. Como também foi feito com tentativa e erro quando uma tentativa não levar a solução final o passo feito é apagado(encontrar algum 0 em uma submatriz proposta) e vai para próxima tentativa.

Para a verificação se há uma submatriz de ordem m em um dado ponto é necessário o algoritmo 1.

**Algoritmo 1** Checar existencia de submatriz.

**Input:** matrix :: refToInteger, y, x, area

```
1: procedure checkSubSquare
2:     for i = y to y + area − 1 do
3:         for j = x to x + area − 1 do
4:             if matrix[i][j] == 0 then
5:                 return false
6:     return true
```

Porém quando se está expandindo a partir de um ponto só é preciso verificar os cantos logo um outro algoritmo pode ser criado somente para a verificação de 1's no canto para evitar redundância de verificação.(algoritmo 2)

**Algoritmo 2** Checar existencia de submatriz.

**Input:** matrix :: refToInteger, y, x, area, corner

```
1: enum Moves{LeftBottom=0,LeftTop,RightTop,RightBottom}
2: procedure checkSubSquareCorner                                   ▷ O(area)
3:     if corner == LeftBottom ∨ corner == RightBottom then yAdd = area − 1 else yAdd = 0
4:     if corner == RightTop ∨ corner == RightBottom then xAdd = area − 1 else xAdd = 0
5:     for i = x to x + area do
6:         if matrix[y + yAdd][i] == 0 then
7:             return false
8:     for i = y to y + area do
9:         if matrix[i][x + xAdd] == 0 then
10:            return false
11:    return true
```

Então o algoritmo que testa todas possibilidades com auxilio do algoritmo 2 será o algoritmo 3.

O algoritmo 3 começa zerando os campos da matriz resultado, depois dois laços aninhados percorrem a matriz nos índices i e j. Na linha 5 é passado por todas ordens possíveis da matriz até que alguma propriedade não seja valida que é checado na linha 6(no código em c há otimizações), se o tamanho da submatriz a ser testada violar o tamanho da matriz ou se os cantos da matriz não estiverem preenchidos por 1 então começa a testar a próxima célula caso exista e durante esse processo se a ordem obtida for maior que a atual então guarda essa submatriz.

## 3.2 Branch and bound

O algoritmo força bruta por tentativa e erro pode ainda ser melhorado utilizando branch and bound [6]. O objetivo do algoritmo é eliminar verificações de áreas que não serão obtidas uma ordem maior de submatriz que a atual, desse modo cortando grande parte do problema. Isso pode ser feito facilmente somente trocando o ponto de parada do primeiro laço (linha 3) de $n - 1$ para $n - maxSqr.order - 1$ e a mesma coisa para o segundo laço (linha 4).

## 3.3 Estratégia gulosa

Para esse problema é possível dar diversas estratégias gulosas a partir do critério global de otimalidade que é: a sub matriz quadrada de 1 deve ser a maior possível. [4]

Para maximizar a obtenção de uma solução ótima é importante ater-se a boas estratégias gulosas.

Na busca por uma estratégia gulosa é interessante explorar as propriedades de uma solução ótima, a solução ótima possui uma alta densidade de 1's em um só local.

**Algoritmo 3** Acha a maior submatriz com força bruta.

**Input:** matrix :: refToInteger, n

```
1: procedure maxSubSquareBruteForce
2:     maxSqr = {0,0,0}
3:     for i = 0 to n − 1 do
4:         for j = 0 to n − 1 do
5:             for area = 1 to n do
6:                 if i <= n − area ∧ j <= n − area ∧ checkSubSquareCorner(m, i, j, area, RightBottom) then
7:                     if maxSqr.order < area then
8:                         maxSqr.order = area
9:                         maxSqr.i = i
10:                        maxSqr.j = j
11:                else
12:                    break
13:    return maxSqr
```

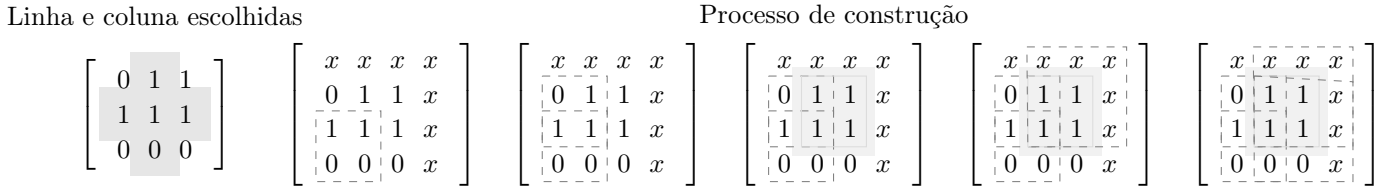



**Figura 3:** Processo guloso.

Um critério de otimalidade local: o ponto escolhido para se expandir deve ter a linha com maior quantidade de 1's em sequência(um ponto que possivelmente está dentro uma submatriz ótima). E esse mesmo pensamento pode ser derivado para colunas e diagonais paralelas a diagonal principal e secundária.

E adicionando todos esses critérios e formulando uma estratégia que engloba os 4 poderia ser feito um algoritmo que cria 4 matrizes, que nelas estão somados 1's em sequência em suas respectivas direções e apos a criação soma as 4 e acha a maior célula, essa célula seria uma célula com alta concentração de 1's que pode-se dizer que é bastante robusto, porém a complexidade de espaço deixaria o algoritmo pouco interessante por conta de sua performance visto que o algoritmo de programação dinâmica teria a mesma complexidade de espaço como será possível observar mais a frente.

Logo a estratégia a ser utilizada para achar um ponto que possivelmente está dentro de uma submatriz ótima irá se ater a propriedade simples de linha com maior quantidade de 1's em sequência e coluna com maior quantidade de 1's em sequência.

É importante notar que para uma matriz da forma

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 0 | 1 | 1 | | 0 | 1 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 | | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | | 1 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | | 1 | 0 | 1 | 1 | 1 |

**Tabela 1:** Casos de solução ruim pela estratégia gulosa.

a estratégia gulosa não irá apresentar uma solução ótima.

O algoritmo *maxSubSquareGreedy* (4) é dependente da função *maxSubSquarePoint* como pode ser visto, ou seja, é necessário outro processo além do de escolha do ponto, é necessário expandir por esse ponto para obter a maior matriz nesse ponto (etapa de construção da solução).

Para expandir por um ponto é necessário definir os sentidos de tentativa de crescimento. Por exemplo: cresce o máximo possível no canto inferior esquerdo depois esquerdo superior, direito superior e direito inferior. Esse processo é ilustrado na figura 3 (matriz quadrada de ordem 3).

Na matriz da figura 3 o primeiro movimento falha, o segundo também e no terceiro já é possível expandir a matriz(movimento para a direita superior) e incrementado no conjunto solução, o segundo movimento para essa mesma direção já não é possível pois acessa posições invalidas logo tenta o outro movimento que também é invalido e então termina e retorna o conjunto solução que é uma matriz 2x2. Veja o algoritmo 5.

Seria possível obter a maior matriz que engloba um ponto dado fazendo todas combinações de ordem movimentos porém necessitaria de uma avaliação de qual a solução maior e isso pode ser avaliado como um processo não guloso, logo no algoritmo é utilizado uma ordem arbitrária.

Outra proposta para se achar a melhor solução de maneira gulosa seria a execução de todas combinações de

**Algoritmo 4** Acha a maior submatriz com uma estratégia gulosa.

**Input:** matrix :: refToInteger, n

```
1: procedure maxSubSquareGreedy
2:     i,j,bestLine=0,bestColumn=0,bestLineQnt=0,bestColumnQnt=0,tempQnt=0
3:     for i = 0 to n do
4:         tempQnt = 0
5:         for j = 0 to n do
6:             tempQnt+ = matrix[i][j]
7:             if matrix[i][j] == 0 then
8:                 if tempQnt > bestLineQnt then
9:                     bestLineQnt = tempQnt
10:                    bestLine = i
11:                tempQnt = 0
12:        if tempQnt > bestLineQnt then
13:            bestLineQnt = tempQnt
14:            bestLine = i
15:        tempQnt = 0
16:        for j = 0 to n do
17:            tempQnt+ = matrix[i][j]
18:            if matrix[i][j] == 0 then
19:                if tempQnt > bestColumnQnt then
20:                    bestLineQnt = tempQnt
21:                    bestLine = i
22:                tempQnt = 0
23:        if tempQnt > bestLineQnt then
24:            bestLineQnt = tempQnt
25:            bestLine = i
26:    maxSqr.order = m[bestLine][bestColumn]
27:    maxSqr.i = bestLine
28:    maxSqr.j = bestColumn
29:    subSquarePoint(m, n, maxSqr)
```

**Algoritmo 5** Expande a partir de um ponto.

**Input:** matrix :: refToInteger, n,maxSqr

```
1: procedure subSquarePoint
2:     bestLine = maxSqr.i, bestColumn = maxSqr.j, right = bestColumn, left = bestColumn, top = bestLine, bottom =
       bestLine, moved
3:     corner = LeftBottom
4:     while corner <= RightBottom do
5:         Faz um movimento
6:         if Se foi um movimento valido then
7:             if checkSubSquare(matriz,top,left,right-left+1,corner) then
8:                 Adiciona o pedaço valido descoberto ao conjunto solução
9:             else
10:                Desfaz movimento
11:        else
12:            Proximo sentido de movimento
```

estratégias gulosas e checar qual a estratégia da a maior submatriz, desse modo poderia-se chegar a soluções possivelmente melhores. A figura 4 ilustra esse processo.

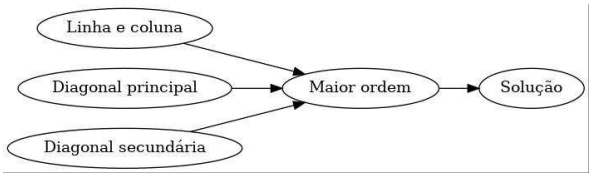



**Figura 4:** Combinação de estratégias gulosas.

### 3.4 Programação dinâmica

O paradigma de programação para ser aplicado nesse problema, mais rápido e eficiente que o força bruta e que, ao contrário da estratégia gulosa apresentada, sempre alcança solução ótima é o paradigma de programação dinâmica.

Visto que esse problema apresenta sobreposição dos subproblemas o algoritmo de força bruta trabalha mais que o necessário. O algoritmo de programação dinâmica a ser apresentado não irá apresentar esse problema.

Resolvendo o problema recursivamente [5] e guardando os valores das submatrizes em uma tabela é exibido a propriedade de subestrutura ótima [3] desse problema.

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | $\Longrightarrow$ | | | | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 2 | 2 | |
| 1 | 1 | 1 | | | | | 1 | 2 | | 1 | 2 | 2 | 1 | 2 | 2 | 1 | 2 | 2 | 1 | 2 |
| 1 | 1 | 1 | | | | | 1 | | | 1 | | | 1 | 2 | | 1 | 2 | 3 | | |

É possível notar que os problemas maiores dependem dos problemas menores para serem resolvidos. Então o segundo passo é descrever o valor de uma solução ótima:

$$m[i,j] = \begin{cases} 0 & \text{if } m[i,j] = 0 \wedge (i = 0 \vee j = 0) \\ 1 & \text{if } m[i,j] = 1 \wedge (i = 0 \vee j = 0) \\ 0 & \text{if } m[i,j] = 0 \wedge i \neq 0 \wedge j \neq 0 \\ 1 + min(m[i-1,j], m[i-1,j-1], m[i,j-1]) & \text{if } m[i,j] = 1 \wedge i \neq 0 \wedge j \neq 0 \end{cases} \tag{1}$$

$$\begin{aligned} m &= max(m[i,j]) \\ resolver(A_{n \times n}) &= (s_{ij})_{m \times m} \end{aligned} \tag{2}$$

Essa é toda a base necessária para escrever o algoritmo que resolve o problema.

O algoritmo com programação dinâmica (6) começara criando um auxiliar que guardará a maior submatriz encontrada até o momento e uma tabela de ordem $n \times n$ para guardar as soluções, após isso é necessário primeiro completar a tabela na sua primeira coluna e linha, pois são problemas elementares, para então começar a análise dos problemas que sobrepõem os outros problemas.

O próximo passo do algoritmo é completar a tabela passando em cada linha por todas suas colunas começando por uma linha a baixo e uma coluna a frente em todas linhas, e basta utilizar a lógica dos dois últimos casos da equação (1) para aplicar valores em cada célula $(i, j)$. Para guardar a maior ordem de uma submatriz basta verificar se um valor de uma solução é maior que a submatriz encontrada até então ou não (linha 21), se for guarda a sua ordem e índice, visto que a solução nesse algoritmo é construída de cima para baixo na tabela então o índice é respectivo ao canto inferior direito, mas nesse trabalho esta sendo usado o padrão de que o índice se refere ao canto superior esquerdo logo é feito uma normalização nas linhas 23 e 24.

### 3.5 Programa principal

O programa principal é feito em função dos algoritmos obtidos que resolvem o problema de diferentes formas. A figura 5 ilustra o programa principal.

## 4 Análise de complexidade

### 4.1 Complexidade de tempo

Antes da análise de complexidade é importante definir a variável que irá dar a complexidade do algoritmo.

$$\underbrace{n}_{\substack{\text{quantidade de colunas} \\ \text{ou linhas da matriz}}} \in \mathbb{N} \tag{3}$$

**Algoritmo 6** Acha a maior submatriz com programação dinâmica.

**Input:** matrix :: refToInteger, n

```
1:  procedure maxSubSquareDynamic
2:      maxSqr.order = 0; maxSqr.i = 0; maxSqr.j = 0
3:      table[n][n]
4:      for i = 0 to n do
5:          table[i][0] = matrix[i][0]
6:          table[0][i] = matrix[0][i]
7:          if table[i][0] == 1 then
8:              maxSqr.order = 1
9:              maxSqr.i = i
10:             maxSqr.j = 0
11:         if table[i][0] == 1 then
12:             maxSqr.order = 1
13:             maxSqr.i = 0
14:             maxSqr.j = i
15:     for i = 1 to n do
16:         for j = 1 to n do
17:             if matrix[i][j] == 0 then
18:                 table[i][j] = 0
19:             else
20:                 table[i][j] = 1 + min(table[i − 1][j], table[i − 1][j − 1], table[i][j − 1])
21:                 if table[i][j] > maxSqr.order then
22:                     maxSqr.order = table[i][j]
23:                     maxSqr.i = i − maxSqr.order + 1
24:                     maxSqr.j = j − maxSqr.order + 1
25:     return maxSqr
```

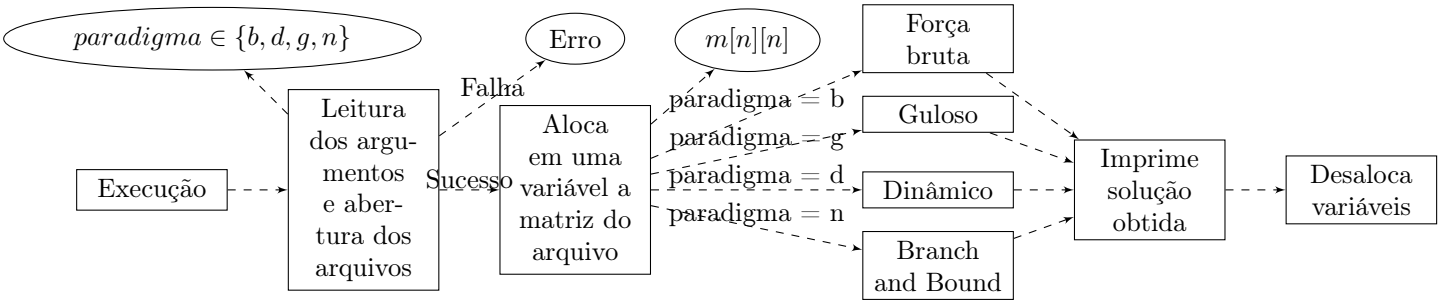



**Figura 5:** Programa principal.

### 4.1.1 Força bruta

Como o algoritmo 3 depende do algoritmo de checar os cantos (algoritmo 2), então é necessário sua análise inicialmente.

$$checkSubSquareCorner(area) \in \sum_{i=1}^{area} O(1) + \sum_{i=1}^{area} O(1) = area \cdot O(1) + area \cdot O(1) = area \cdot O(1) \\ \therefore checkSubSquareCorner(area) \in O(area) \tag{4}$$

$$makSubSquareBruteForce(n) \in \sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{area=1}^{n} O(area) = O(\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{area=1}^{n} area) \\ O(\sum_{i=1}^{n}\sum_{j=1}^{n} \frac{n\cdot(n+1)}{2}) = O(\frac{n^3\cdot(n+1)}{2}) \\ \therefore maxSubSquareBruteForce(n) \in O(n^4) \tag{5}$$

### 4.1.2 Branch and bound

A abordagem em *branch and bound* é complexa, para fazer sua análise detalhada é difícil pois os limites do laço podem mudar ou não a cada iteração. Mas de qualquer maneira foi provado que sua complexidade é $O(n^4)$ porém na pratica seu desempenho deve ser muito melhor.

$$maxSubSquareBranchnBound(n) \\ \sum_{i=1}^{n-Sqr.order}\sum_{j=1}^{n-Sqr.order}\sum_{area=1}^{n} O(area) = O(\sum_{i=1}^{n-Sqr.order}\sum_{j=1}^{n-Sqr.order}\sum_{area=1}^{n} area) \\ O(\sum_{i=1}^{n-Sqr.order}\sum_{j=1}^{n-Sqr.order} \frac{n\cdot(n+1)}{2}); Sqr.order \in \{x : x \in \mathbb{N}^* \wedge x \leq n\} \\ n - Sqr.order; \exists(c \in \mathbb{N}^*)\exists(m \in \mathbb{N}^*)\forall(n \geq m) : 0 \leq n - Sqr.order \leq c \cdot n \\ m = 1; c = 1 \implies 0 \leq n - Sqr.order \leq n \implies n - Sqr.order \in O(n) \\ O(\frac{(n-Sqr.order)^2\cdot n\cdot(n+1)}{2}) \in O(\frac{n^3\cdot(n+1)}{2}) \therefore maxSubSquareBranchnBound(n) \in O(n^4) \tag{6}$$

### 4.1.3 Algoritmo guloso

O pior caso do algoritmo *subSquarePoint* (5) ocorrerá quando ele continua adicionando pedaços validos e no máximo pode existir $O(n^2)$ pedaços, então o algoritmo *maxSubSquareGreedy* é $O(n^2)$ como é mostrado.

$$\therefore subSquarePoint(n) \in O(n^2) \tag{7}$$

$$maxSubSquareGreedy(n) \in O(max(\sum_{i=0}^{n}\sum_{j=0}^{n} 1, n^2)) \\ \therefore maxSubSquareGreedy(n) \in O(n^2) \tag{8}$$

### 4.1.4 Programação dinâmica

$$maxSubSquareDynamic(n) \in \sum_{i=0}^{n} O(1) + \sum_{i=0}^{n}\sum_{j=0}^{n} O(1) = O(max(\sum_{i=0}^{n} 1, \sum_{i=1}^{n}\sum_{j=1}^{n} 1)) = O(n^2) \\ \therefore maxSubSquareDynamic(n) \in O(n^2) \tag{9}$$

### 4.1.5 Programa principal

A análise do programa principal será feita com base na figura 5. É intuitivo pensar que o passo de alocar em uma variável a matriz que foi entrada no arquivo tem complexidade $O(n^2)$, apos isso um dos algoritmos será executado, considerando o pior caso então será $O(max(n^2, n^4)) \implies O(n^4)$, logo a complexidade do programa principal será $O(n^4)$.

## 4.2 Complexidade de espaço

Antes de se fazer a análise de complexidade de espaço é preciso evidenciar que os algoritmos que possuem em sua lista de parâmetros uma referência para um vetor não apresentará complexidade de espaço relacionada com o espaço ocupado por esse valor referenciado(Ex. $O(n^c), c \in \mathbb{N}$) pois o parâmetro guarda uma constante(referência para o vetor) ou seja a complexidade de espaço é constante.[1]

### 4.2.1 Força bruta

O algoritmo força bruta utiliza a função **checkSubSquareCorner** logo é necessário analisar sua complexidade de espaço, essa função não cria um vetor ou algo parecido, somente usa um espaço constante de espaço logo sua complexidade é O(1). O algoritmo força bruta apresenta complexidade de espaço $O(1)$ da mesma forma pois utiliza uma quantidade de espaço constante, e como já comentado só há uma referência a uma matriz $n \times n$ e essa referência é $O(1)$.

### 4.2.2 Branch and bound

A abordagem em *branch and bound* é bem similar ao força bruta, é fácil identificar que esse algoritmo também tem complexidade de espaço $O(1)$.

### 4.2.3 Algoritmo guloso

A função **subSquarePoint** utiliza a função **checkSubSquareCorner** que tem complexidade de espaço $O(1)$ e não há nenhum crescimento de espaço na função **subSquarePoint** logo sua complexidade também é $O(1)$. Como a função **subSquarePoint** é $O(1)$ então a complexidade de espaço da função **maxSubSquareGreedy** é $O(1)$ também pois não há nenhuma variavel com tamanho dependente da entrada.

### 4.2.4 Programação dinâmica

O algoritmo dinâmico apresenta complexidade de espaço $O(n^2)$ pois na linha 3 há a criação de uma tabela auxiliar com tamanho dependente da entrada de ordem $n^2$.

### 4.2.5 Programa principal

Com base na figura 5 o passo de alocar uma variável que tem tamanho $n^2$ implicará em uma complexidade de espaço de $O(n^2)$, junto com o resto do programa $O(max(n^2, 1, 1, n^2, 1)) = O(n^2)$. Logo a complexidade de espaço do programa principal é $O(n^2)$.

## 4.3 Análise Geral

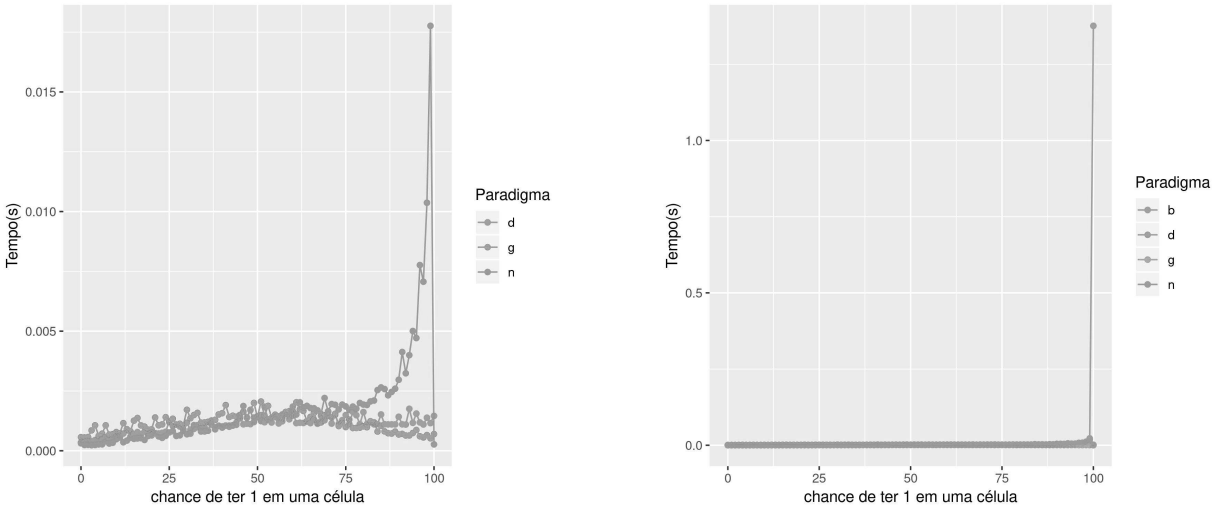
A tabela 2 mostra todas complexidades obtidas, com ela pode-se ver que o melhor algoritmo é o dinâmico porém seu custo de espaço é dependente da entrada o que o torna indesejável caso haja pouco espaço de memória. Porém independente do algoritmo todos eles utilizam $O(n^2)$ de espaço em um aspecto geral contando com o programa principal pois é preciso guardar a matriz dada na memória principal, mas o algoritmo dinâmico utiliza pelo menos o dobro desse espaço pois usa uma tabela de ordem $n \times n$ a mais.

| Complexidade | Tempo | Espaço |
|---|---|---|
| Força bruta | $O(n^4)$ | $O(1)$ |
| Guloso | $O(n^2)$ | $O(1)$ |
| Dinâmico | $O(n^2)$ | $O(n^2)$ |
| Branch and bound | $O(n^4)$ | $O(1)$ |
| Principal | $O(n^4)$ | $O(n^2)$ |

**Tabela 2:** Complexidades.

# 5 Resultados

A maquina utilizada para os experimentos possui as seguintes especificações: Intel(R) Core(TM) i3-4005U CPU @ 1.70GHz e 4GiB de memória RAM.




**(a)** Todos algoritmos exceto força bruta.

**(b)** Todos algoritmos.

**Figura 6:** Tempo por chance de ter um 1 em uma célula.(matriz $200 \times 200$)

## 5.1 Tempo

Antes da análise de resultados é importante refletir sobre quais fatores influenciam no tempo de execução. Não é só o tamanho da matriz de entrada que influencia no tempo, mas também a quantidade de submatrizes e suas ordens dentro da matriz. A figura 6 mostra como o algoritmo se comporta de acordo com a mudança de seu conteúdo. O conteúdo traz efeitos diferentes para cada algoritmo, o força bruta aumenta seu tempo com o aumento da chance de ter 1; o dinâmico também porém de uma maneira mais suave; já o algoritmo guloso aumenta seu tempo quando se aproxima de 50% de chance e diminui longe de 50%; o *branch and bound* aumenta quando a porcentagem aumenta porém ao chegar em 100% seu custo cai drasticamente, sendo até mesmo um de seus melhores casos, por causa do corte de problemas ou seja, seu pior caso está em uma matriz quase completa de 1's.

Primeiramente para checar se as complexidades obtidas que estão na tabela 2 são corretas é preciso gerar entradas que exploram o pior caso para obter o pior tempo. Com base no pior tipo de conteudo em uma entrada de cada algoritmo basta variar a ordem da matriz.

A figura 7 mostra a regressão polinomial executando o algoritmo força bruta no seu pior caso(matriz completa de 1's). A equação polinomial obtida consegue predizer os valores corretamente, uma equação cubica nesse caso pode obter um resultado bem próximo porém ela falha nos valores iniciais. Pela limitação de obter dados para ordens maiores de matrizes, pois o gráfico já chega mais que 60 segundos por entrada, dificulta mostrar uma falha mais forte em uma possivel regressão cubica. Ou seja na prática o crescimento do tempo é um pouco melhor.

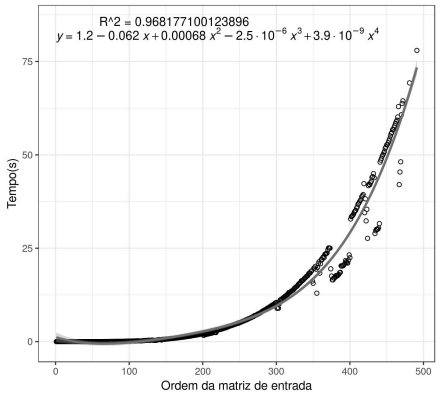
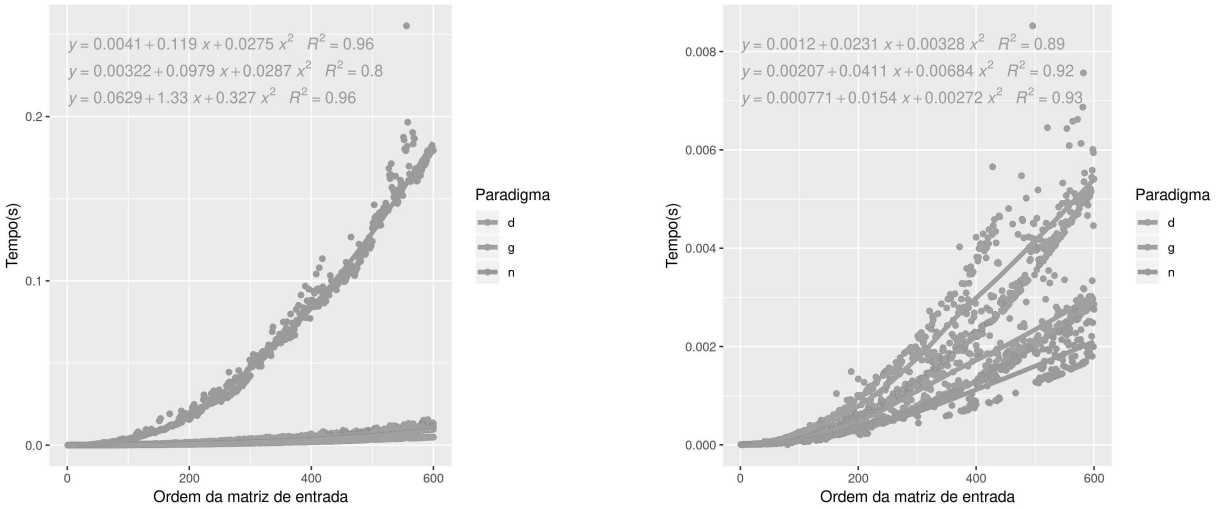



**Figura 7:** Tempo força bruta pior caso.

A figura 8a mostra o comportamento do tempo de acordo com o tamanho da entrada para os outros algoritmos, o tempo dos algoritmos tem valores de tempo totalmente discrepantes do algoritmo de força bruta que chega na casa dos minutos.




**(a)** Pior caso.

**(b)** Melhor caso.

**Figura 8:** Tempo dos paradigmas rápidos.

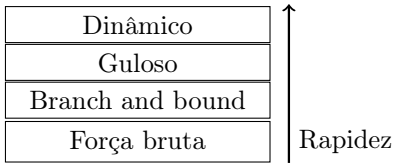



**Figura 9:** Hierarquia de tempo empiríco.

Devido ao $R^2$ ter um valor alto nessas regressões de polinômio quadrado pode se concluir que empiricamente foi comprovado a complexidade desses algoritmos. Na prática o *branch and bound* foi muito melhor do que a complexidade obtida na secção de análise de complexidade ($O(n^4)$), um polinômio quadrado representou o custo da função com um erro pequeno, ou seja na prática o algoritmo teve uma complexidade perto de $O(n^2)$.

Também é importante notar que o algoritmo com programação dinâmica se mantém na mesma ordem de tempo independente dos valores de uma matriz porém seus fatores podem ser mudados se a operação mais custosa não for efetuada($a_{ij} = 0$). A figura 8b mostra alguns outros casos de entrada melhores para cada algoritmo e é obtido que o algoritmo de *branch and bound* consegue ter o menor tempo no seu melhor caso de entrada.

Então com o tempo interpretado na prática pode-se ver uma hierarquia de melhores algoritmos em questão de tempo no pior caso, a figura 9 mostra essa hierarquia.

## 5.2 Espaço

Para provar o tempo do algoritmo principal em relação a memória basta monitorar as alocações feitas pelo algoritmo para cada tamanho de entrada. O monitoramento será feito somente da memória *heap* [2] que é a parte da memória onde a alocação dinâmica é feita e o espaço pode-se variar dinamicamente permitindo-se assim obter uma função de complexidade correta.

Pela figura 10 o espaço utilizado pelos 3 algoritmos que não o dinâmico utilizam o mesmo espaço e até mesmo tem a mesma regressão polinomial que provém do algoritmo principal na alocação de memória para a matriz de entrada. Já o algoritmo dinâmico tem a complexidade $O(n^2)$ do algoritmo principal mais $O(n^2)$ na matriz auxiliar logo seu custo de espaço é maior porém continua tendo complexidade assintótica $O(n^2)$ assim como todos outros paradigmas considerando o programa principal.

Então é necessário ser cauteloso com a utilização do algoritmo dinâmico para grandes matrizes pois pode ser visto que o custo de espaço tem fatores em sua função que o fazem crescer de forma muito maior que os outros algoritmos.

Pelas funções de complexidade obtidas com um $R^2 = 1$ é possível predizer o quanto de espaço o algoritmo irá utilizar para cada ordem de entrada o que pode ser uma coisa útil para aplicação do algoritmo na prática. Um exemplo de aplicação dessa função prediria se o algoritmo gasta mais memória do que o computador atual possui disponível então se sim usa outro algoritmo que seja possível, esse processo é ilustrado na figura 11. É importante lembrar que a função de complexidade obtida varia de arquitetura para arquitetura e é um cuidado que deve ser tomado pois os tipos de dados primitivos podem ter tamanhos diferentes.

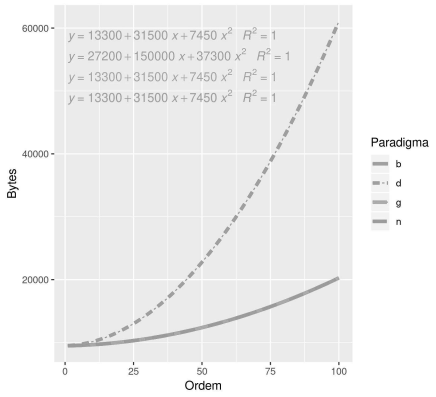



**Figura 10:** Espaço usado pelos paradigmas.

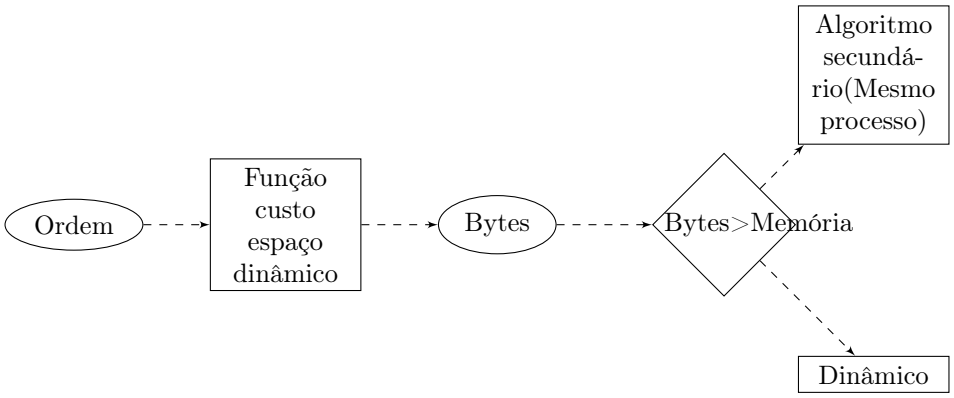



**Figura 11:** Utilização da função de custo de espaço.

### 5.3 Estratégia gulosa

A taxa de acerto do algoritmo guloso como é mostrado na figura 12 é muito baixa, ele somente obtém uma solução ótima quando o tamanho da matriz é muito baixo. Porém é importante se atentar ao fato de que os dados de entrada foram com matrizes preenchidas de forma aleatória e o problema a ser resolvido vem da realidade que tem padrões e não possui toda essa estrutura aleatória que faz com que a estrategia seja muito ineficiente.

Com entradas com mais ordem e algo mais parecido com placentas na realidade o algoritmo possui soluções melhores, isso pode ser visto comparando os diagramas de caixa na figura 13.

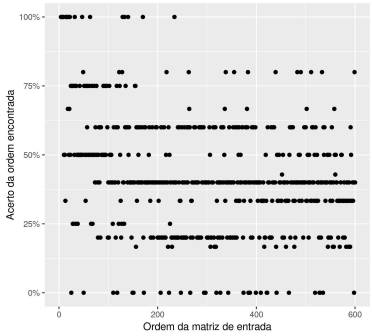



**Figura 12:** Taxa de acerto com estratégia gulosa.

## 6 Conclusão

Com o trabalho foi possível observar que o problema possui um algoritmo que resolve-o com complexidade de tempo de $O(n^2)$ e com custo de espaço $O(n^2)$, diferentemente do força bruta com tentativa e erro que tem complexidade assintótica de tempo de $O(n^4)$ e se mostrou ineficiente, garantindo solução ótima por meio de programação dinâmica


**(a)** Entradas realistas.(10 amostras)

**(b)** Entradas aleatórias.

**Figura 13:** Taxa de acerto da estratégia gulosa.

e se mostrando o algoritmo mais estável por meio dos resultados observados, porém por ser estável em alguns casos os algoritmos que utilizam *branch and bound* e estratégia gulosa apresentam tempos melhores porém como o pior caso deve ser sempre o considerado é melhor utilizar o algoritmo com programação dinâmica e além disso a estratégia gulosa não garante uma solução ótima e nos testes o seu erro foi bem grande, logo não é nem um pouco ideal ser utilizada, somente caso seja uma matriz extremamente grande e não seja possível aplicar o *branch and bound* ou o dinâmico.

É bem difícil que para a aplicação proposta uma matriz de entrada esteja quase toda completa com 1's, logo o *branch and bound* pode se sair melhor ainda com uma certa garantia devido a esse possivel padrão de problema de entrada. Se em um trabalho futuro for verificado que a placenta tenha em média 75% para menos de material sólido o algoritmo *branch and bound* se torna uma alternativa mais viável para ser utilizado ao invés do guloso, isso dito com base nos gráficos mostrados.

Também foi possível obter uma função de complexidade de espaço para os algoritmos empiricamente e com isso foi proposto uma utilização para esse dado que é a predição de estouro da memória para o algoritmo dinâmico e forçar a utilização de algum dos outros algoritmos caracterizado como o melhor. Por meio dos resultados empíricos o *branch and bound* se mostrou o segundo algoritmo mais viável para utilização caso não seja possível a utilização do dinâmico, garante uma solução ótima e também tem um tempo bem similar ao guloso, em alguns casos sendo mais rápido. O guloso nos testes teve um erro grande para entradas aleatórias porém com entradas com uma forma mais similar a uma placenta o algoritmo da soluções melhores.

# Referências

[1] Space complexity. `https://www.cs.northwestern.edu/academics/courses/311/html/space-complexity.html`. Accessed: 2019-04-23.

[2] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 2018.

[3] Thomas Cormen. *Introduction to algorithms*. MIT Press, Cambridge, Mass, 2009.

[4] S.A. Curtis. The classification of greedy algorithms. *Science of Computer Programming*, 49(1-3):125–157, 2003.

[5] Antti Laaksonen. *Introduction*, pages 64–65. Undergraduate Topics in Computer Science. Springer International Publishing, 2017.

[6] E. L. Lawler and D. E. Wood. Branch-and-bound methods: A survey. *Operations Research*, 14(4):699–719, August 1966.