Universidade Federal de São João del-Rei
Algoritmo e Estrutura de Dados III
Trabalho Prático 2

Felipe Francisco Rios de Melo
Thales Mrad Leijoto

1. Introdução

A programação pode auxiliar em diversas áreas. Um dos ramos mais beneficiados pela informatização é a área da saúde, principalmente no que se remete a área de pesquisas na prevenção, diagnóstico e tratamento de doenças.

Segundo pesquisas recentes, a partir da análise de imagens oriundas da placenta de bebês recém nascidos e da identificação de certos elementos na placenta, é possível prevenir uma enorme quantidade de doenças que o bebê poderia ter ao longo de sua vida.

Desta forma, o problema tratado neste trabalho refere-se à implementação de diferentes tipos de algoritmos (força bruta, guloso e programação dinâmica) para auxiliar na identificação de materiais sólidos em placenta de recém nascidos, por meio da análise da imagem da placenta.

2. Modelagem e solução do problema proposto

2.1. Problema proposto

O problema proposto por este trabalho prático é, dado uma imagem binária — apenas dois valores possível para cada pixel, 0 ou 1 — proveniente de uma placenta de bebês, identificar a maior imagem com material sólido possível, ou seja, determinar a maior sub-matriz S de dimensões m x m, m <= n, onde todos os elementos de S são 1. Em outras palavras, encontrar a maior sub-matriz composta unicamente por 1's.

Por exemplo:

Temos a seguinte matriz 5x5:

0	0	0	0	0
0	0	1	0	0
0	0	0	1	1
0	0	0	1	1
0	0	0	1	0

A maior submatriz é uma matriz 2x2:

0	0	0	0	0
0	0	1	0	0
0	0	0	**1**	**1**
0	0	0	**1**	**1**
0	0	0	1	0

2.2. Modelagem

Intuitivamente, para modelar uma imagem binária bidimensional quadrada (comprimento e largura idênticos), se pensa em uma matriz bidimensional n x n. Foi exatamente desta forma a implementação da imagem neste presente trabalho, onde que n é dado pela variável *tamanhoMatriz.*

Foi usado também uma estrutura de dados que recebe as coordenadas cartesianas do ponto inicial da submatriz, utilizados para auxiliar na lógica de impressão da submatriz resultante de cada paradigma.

2.3. Formato de entrada e saída

Como já mencionado, a entrada de dados é sequência de 0's e 1's, modelada como uma matriz quadrada pelo algoritmo. Essa sequência está armazenada em um arquivo de texto no seguinte formato:

De modo que as colunas são separadas por um espaço e as linhas separadas por uma quebra de linha.

Lido todo o conteúdo do arquivo de entrada e armazenado na matriz, o usuário deve escolher para qual estratégia de algoritmo ele quer ver a saída (pode-se optar por ver todas). O próximo passo executado pelo programa é buscar a maior sub-matriz composta apenas por 1. Para isso foram utilizados três projetos de algoritmos diferentes para encontrar a solução do problema (seção 2.4).

A saída do programa é imprimida em um arquivo de texto, e é composta pela solução encontrada pelo algoritmo escolhido ou a solução de todos os três paradigmas caso esta seja a opção escolhida pelo usuário.

2.4. Solução proposta

O problema foi implementado, segundo três paradigmas de projeto de algoritmos distintos, são eles: força bruta (tentativa e erro), algoritmo guloso e programação dinâmica.

2.4.1. Força bruta (tentativa e erro)

Força bruta (ou busca exaustiva) é uma algoritmo trivial mas de uso muito geral que consiste em enumerar todos os possíveis candidatos de uma solução e verificar se cada um satisfaz o problema.

Esse algoritmo possui uma implementação muito simples, e sempre encontrará uma solução se ela existir. Entretanto, seu custo computacional é proporcional ao número de candidatos a solução.

A implementação do algoritmo por força bruta para este problema, consiste em pegar a primeira posição da matriz, considerá-la uma sub-matriz de dimensão 1x1 desta matriz, e analisar se o seu conteúdo é 1 ou 0, caso seja 1, aumentamos a dimensão desta sub-matriz em uma unidade (sempre obedecendo o principio da matriz quadrada). Mais uma vez é analisado todo o conteúdo da sub-matriz, caso todos os elementos sejam 1, aumenta-se a tamanho da sub-matriz, e assim o processo se repete até que exista um 0 no conjunto ou chegue até a última coluna ou linha da matriz. Este processo é feito para todos os elementos da matriz. Mas diferentemente de um algoritmo de busca exaustiva propriamente dito, este algoritmo apresentado possui uma refinamento. É passado sempre, pra função responsável pela “expansão” da sub-matriz a partir de uma posição da matriz, a dimensão da maior sub-matriz encontrada até o momento. Desta maneira, o algoritmo passará a buscar somente sub-matrizes que são maiores que a maior já identificada, assim evitando cálculos desnecessários que não levarão a uma solução ótima.

Exemplo de execução do força bruta (tentativa e erro):

Posição [0][0] - tamanho sub-matriz: 1
sub-matriz contém somente 1’s? Sim.
Aumenta tamanho sub-matriz.

1	0	1	1
0	0	1	1
0	0	0	0
0	0	0	0

Posição [0][0] - tamanho sub-matriz: 2
sub-matriz contém somente 1’s? Não.
Vá para proxima posição. MAIOR = 1.

1	**0**	1	1
0	**0**	1	1
0	0	0	0
0	0	0	0

Posição [0][1] - tamanho sub-matriz: 1
sub-matriz contém somente 1’s? Não.
Vá para proxima posição.

1	**0**	1	1
0	0	1	1
0	0	0	0
0	0	0	0

Posição [0][2] - tamanho sub-matriz: 1
sub-matriz contém somente 1’s? Sim.
Aumenta tamanho sub-matriz.

1	0	**1**	1
0	0	1	1
0	0	0	0
0	0	0	0

Posição [0][2] - tamanho sub-matriz: 2
sub-matriz contém somente 1’s? Sim.
Aumenta tamanho sub-matriz.

1	0	**1**	**1**
0	0	**1**	**1**
0	0	0	0
0	0	0	0

Posição [0][2] - tamanho sub-matriz: 2
Chegou até o a última coluna.
Vá para próxima posição. MAIOR = 2.

1	0	**1**	**1**
0	0	**1**	**1**
0	0	0	0
0	0	0	0

2.4.2. Algoritmo guloso

Algoritmo guloso é técnica de projeto de algoritmos que tenta resolver o problema fazendo a escolha localmente ótima em cada fase com a esperança de encontrar um ótimo global.

Contudo nem sempre produz a melhor solução (depende da quantidade de informação fornecida), quanto mais informações, maior a chance de produzir uma solução melhor.

A implementação usando algoritmo guloso foi usado como parâmetro a escolha da maior sequência diagonal de 1, acreditando que ela trará a melhor solução possível. Escolhido a maior diagonal, a função checa o conteúdo da sub-matriz originada pela diagonal, se ela for composta apenas por valores iguais a 1, é retornado o tamanho desta sub-matriz. Caso exista ao menos um 0, são testadas as possibilidades de matrizes com o tamanho da sub-matriz - 1, caso não ache uma solução, vai diminuindo o tamanho até encontrar a maior submatriz composta unicamente por 1's e retorna seu tamanho.

Tamanho maior diagonal: 3

0	0	0	1
1	**1**	1	1
1	0	**1**	1
1	0	0	**1**

Tamanho sub-matriz = tamanho maior diagonal = 3
sub-matriz contém somente 1's? Não.
sub-matriz = sub-matriz - 1

0	0	0	1
1	**1**	**1**	**1**
1	**0**	**1**	**1**
1	**0**	**0**	**1**

Tamanho sub-matriz: 2
sub-matriz contém somente 1's? Não.
Analisa a próxima sub-matriz de mesmo tamanho dentro do escopo da maior diagonal

0	0	0	1
1	**1**	**1**	1
1	**0**	**1**	1
1	0	0	1

Tamanho sub-matriz: 2
sub-matriz contém somente 1's? Sim.
MAIOR = 2

0	0	0	1
1	1	**1**	**1**
1	0	**1**	**1**
1	0	0	1

2.4.3. Programação dinâmica

A estratégia da programação dinâmica consiste em dividir o problema em problemas menores, então calcula-se a solução dos menores subproblemas até os maiores, armazenando em uma tabela os resultados. Se algum subproblema precisar da solução de outro subproblema menor, ele buscará na tabela, evitando assim cálculos repetitivos.

Para o problema em questão, cria-se uma outra matriz com o tamanho da matriz de entrada mais 1, e é definido como valor 0 todos os elementos da primeira linha e da primeira coluna. Essa matriz armazenará os resultados dos subproblemas, salvando em cada posição, a maior submatriz possível que termina naquele ponto. Cada ponto coordenado da matriz de entrada é verificado, se seu valor for 0, é salvo 0 no ponto correspondente na matriz de resultados (matAux[i][j]), já se seu valor for igual a 1, verifica-se os valores salvos na matriz das soluções do ponto acima, a esquerda e acima (sentido noroeste). Pega o mínimo entre esses valores, soma 1 e salva no ponto correspondente.

Estes passos seguem o modelo matemático mostrado abaixo:

Se M(i - 1, j - 1) = 0 → aux(i, j) = 0
Se M(i - 1, j - 1) = 1 → aux(i, j) = 1 + min(aux(i - 1, j - 1), aux(i - 1, j), aux(i - 1, j))

Ao ser salvo, o valor é verificado e comparado com o antigo maior (que é inicializado com 0), se for maior, o valor maior é atualizado e é salvo o ponto mais acima e a esquerda da matriz, mantendo o padrão das outras funções.

Matriz de entrada

0	0	1	0
1	1	1	0
1	1	1	0
1	1	1	1

Matriz de soluções

0	0	0	0	0
0	0	0	1	0
0	1	1	1	0
0	1	2	2	0
0	1	2	3	1

O fundamento da programação dinâmica é aplicável ao problema em questão, pois ao verificar os 3 valores próximos do ponto na tabela de soluções, o algoritmo está verificando o tamanho da matriz que existe até aquele ponto, os subproblemas resolvidos anteriormente. Como estamos interessados apenas em submatrizes quadradas, elas crescem diagonalmente da mesma forma. O ponto que está em análise é um possível ponto inferior e a direita de uma submatriz em questão.

3. Análise de Complexidade

Nesta seção faremos uma análise da complexidade de tempo e espaço das funções mais importantes do algoritmo. Cada função será analisada individualmente e em seguida calcularemos a complexidade total a partir da análise da função do programa principal. Ressaltamos que “n” é o tamanho da matriz. Cálculos e atribuições serão consideradas constantes, dessa forma, complexidade O(1).

3.1. CalculaTamanhoMatriz()

Tempo:
A função percorre o arquivo de entrada até encontrar um ‘\n’, como a entrada é sempre uma matriz quadrada de tamanho n, o custo para encontrar a quebra de linha é O(n).

Espaço: a função não possui alocação de memória, O(1).

3.2. leituraArqEntrada()

Tempo:
Possui uma função que aloca a matriz, que tem uma complexidade O(n), ao alocar a matriz é lido o conteúdo do arquivo de entrada e gravado na matriz, esse processo tem complexidade $O(n^2)$. Então, a complexidade geral da função é $O(\max(n, n^2)) = O(n^2)$.

Espaço:
A função que aloca a matriz (implicita na *leituraArqEntrada()*), por fazer a alocação dinâmica de uma matriz bidimensional de tamanho n, contém uma complexidade espacial de $O(n^2)$. No restante da função não há nenhuma outra alocação de memória, logo a complexidade espacial resultante da função é $O(n^2)$.

3.4. imprimeMaiorSubMatriz()

Tempo: Segue a explicação da função imprimeMatrizCompleta(), $O(n^2)$.

Espaço: a função não possui alocação de memória, O(1).

3.5. liberaMatriz()

Tempo:
Por ter que percorrer a matriz para desalocar os ponteiros, possui um laço que vai de 0 até n. Logo, tem complexidade O(n).

Espaço: a função não possui alocação de memória, O(1).

3.6. liberaArquivos()

Tempo: Apenas fecha os arquivos e libera a struct, O(1).

Espaço: a função não possui alocação de memória, O(1).

3.7. forcaBruta()

Tempo:
O algoritmo de força bruta contém inicialmente dois laços aninhados que vão de 0 a n, $O(n^2)$, no qual no laço mais interno há uma estrutura condicional que chama outra função, a *submatrizPossivel()*.

A função *submatrizPossivel()*, possui um laço que vai no pior caso, irá de 1 até n, ou seja, executará n-1 iterações, complexidade O(n). Dentro deste laço há uma função que checa o conteúdo da submatriz em análise, a implementação desta função é feita através de dois laços aninhados que percorrem toda a matriz checando se o conteúdo do elemento é 1 ou 0, $O(n^2)$. Tiramos como conclusão que a complexidade total da função *submatrizPossivel()* é $O(n * n^2)$, que é igual a $O(n^3)$ pela regra do produto para notação O.

Como dito anteriormente, a função forcaBruta() contém dois laços aninhados ($O(n^2)$), que chamam a função subMatrizPossivel(), esta por sua vez com complexidade $O(n^3)$. Por fim, então, podemos assumir que no pior caso o algoritmo de força bruta tem complexidade assintótica da ordem de $O(n^5)$.

Espaço: a função não possui alocação de memória, O(1).

3.8. guloso()

Tempo:
O primeiro passo do algoritmo guloso apresentado neste trabalho é achar a maior subdiagonal composta unicamente de 1s. O programa testa todas as diagonais possíveis, esse teste tem pior caso $O(n^3)$. Achado o valor da maior diagonal, cujo pior caso geral para o algoritmo seria quando a diagonal principal fosse inteiramente composta 1's, o programa irá verificar todas as submatrizes possíveis, desde o tamanho da maior diagonal até 1, o ato de verificar se uma submatriz é composta unicamente de 1's pode custar $O(n^2)$, e percorrer todas as submatrizes pode custar $O(n^3)$.

Logo, a complexidade geral do algoritmo guloso é, no pior caso, $O(\max(n^3, n^2 \times n^3) = O(n^5)$, conforme a regra do produto e da soma para a notação O.

Espaço: a função não possui alocação de memória, O(1).

3.9. dinamica()

Tempo:
A função começa criando uma matriz (matriz das soluções) de tamanho igual ao tamanho da matriz de entrada + 1, depois em um laço preenche a primeira linha e a primeira coluna dessa matriz. Esse laço tem complexidade O(n+1). Na segunda parte da função, são usados dois laços aninhados para percorrer a matriz das soluções a partir da segunda linha e da segunda coluna, tendo complexidade assintótica $O(n^2)$.

Então, temos que a complexidade do algoritmo de programação dinâmica é $O(\max(n, n^2)) = O(n^2)$ em todos os casos, independente da entrada.

Espaço: Por ser alocado uma matriz auxiliar de tamanho n x n, o algoritmo de programação dinâmica possui complexidade $O(n^2)$.

3.10. Programa principal

Tempo:
A função *main* é a responsável por chamar todas estas funções citadas acimas, além de algumas outras que não entramos no mérito, por possuir complexidade O(1). Logo, concluímos que a complexidade assintótica do programa no pior caso é dado por $O(\max(n^5, n^2, n^2, 1))$, que é igual a $O(n^5)$.

Observamos, então, que a função de força bruta e gulosa dominam assintoticamente o método de programação dinâmica no pior dos casos.

Espaço:
Quanto a complexidade espacial geral do programa, temos apenas a alocação da matriz bidimensional, logo a complexidade em qualquer caso é $O(n^2)$.

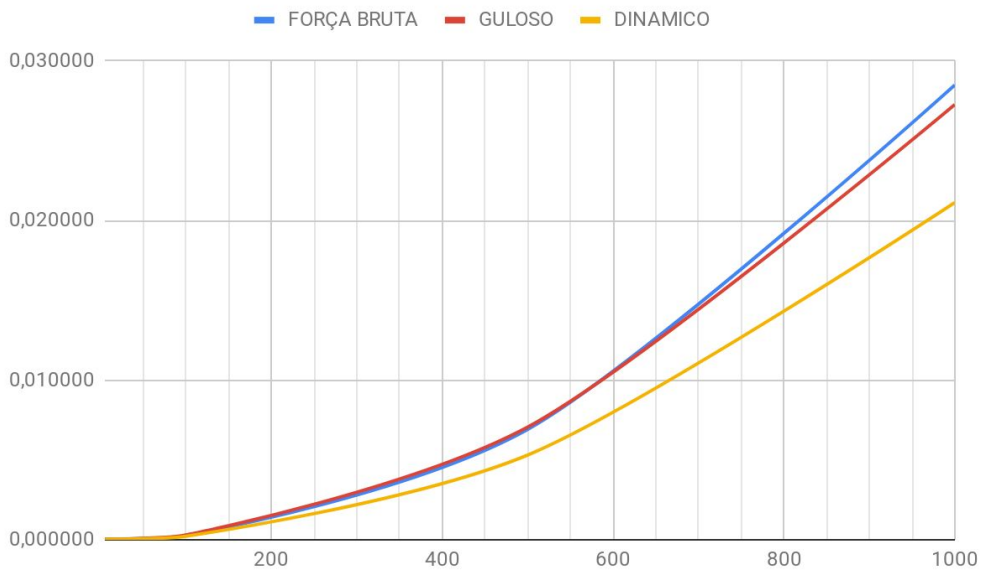
4. Experimentos e análise de resultados

Os testes foram realizados em um Intel Core i5 7ª geração i5-7200U, com 8gb de memória RAM.

4.1. Tempo

Para analisar o crescimento do tempo gasto de execução em relação a dimensão da matriz de entrada, foi realizado testes com diferentes tamanhos de entrada, no qual o programa foi executado 10 vezes para cada parâmetro e a média aritmética do tempo gasto foi calculada. Segue na tabela abaixo este estudo:

GRÁFICO 1. Tempo gasto em razão da dimensão da matriz de entrada com chances iguais de conter 0 ou 1 em cada posição.



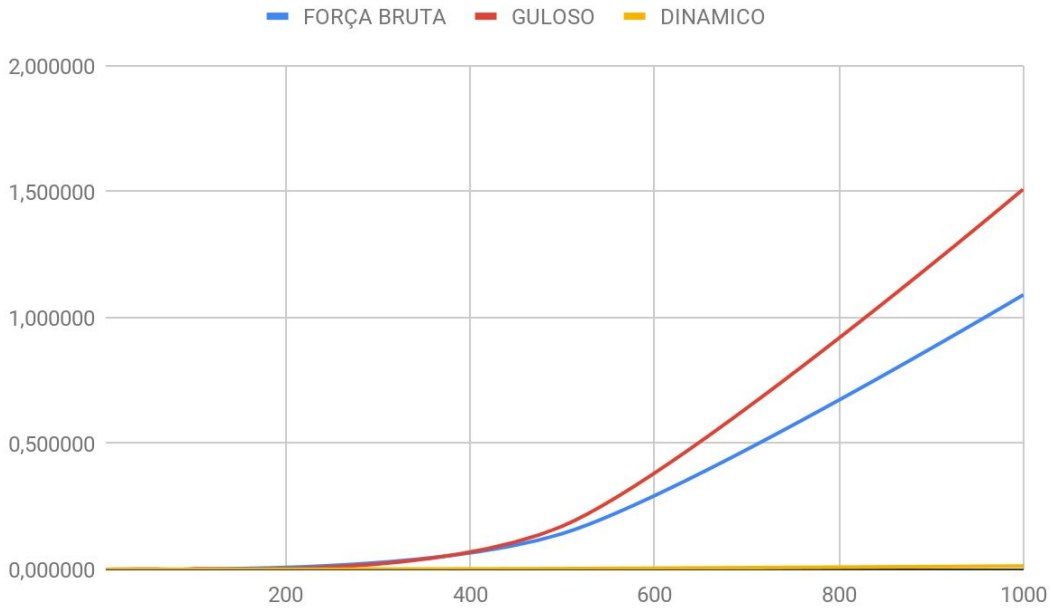
Através deste gráfico é possível observar que a estratégia dinâmica tem um crescimento assintótico nitidamente menor que as outras, ou seja é dominada assintoticamente pelos algoritmos de força bruta e guloso.

Porém foi demonstrado na análise de complexidade (seção 3) que o algoritmo dinâmico possuía complexidade $O(n^2)$, enquanto que os outros dois possuíam $O(n^5)$, e por este fato, a diferença de curva entre os algoritmos deveria ser mais discrepante e, de acordo com o

gráfico, não é. Isso ocorre pois a ordem de complexidade do método de programação dinâmica é estático, independe do tamanho ou configuração da entrada, sempre terá complexidade quadrática, enquanto que o força bruta e o guloso possui complexidade $O(n^5)$ apenas no pior dos piores caso. Como foi executado um número limitado de testes para cada tamanho de entrada, com a configuração de 0's e 1's dispostas aleatoriamente e com iguais chances de ocorrência de ambas, a possibilidade de ocorrer o pior caso é mínima tanto do algoritmo de exaustão, quanto o algoritmo guloso. Além do fato, também, de existir várias podas no algoritmo de força bruta que o deixa eficiente na maioria dos casos.

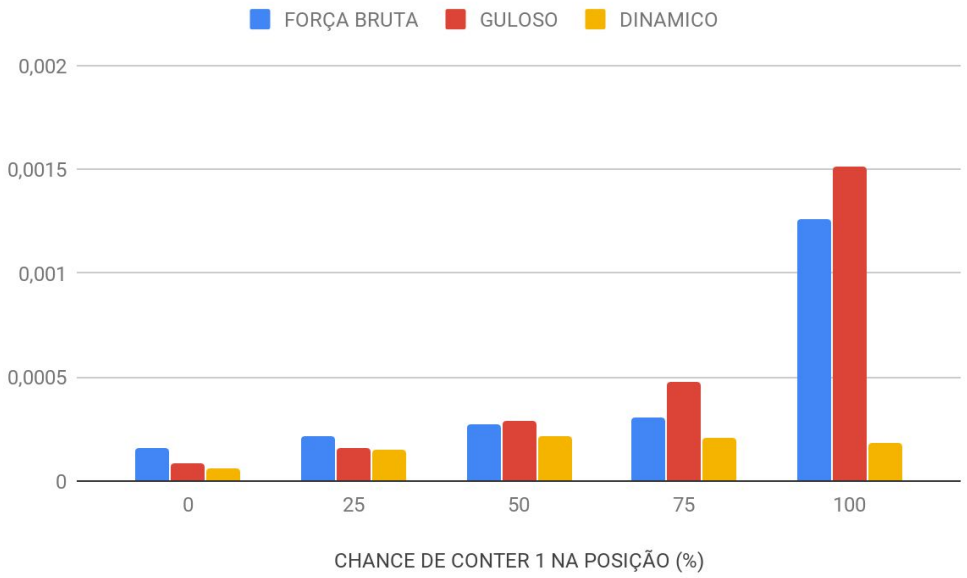
Contudo, foi feito o mesmo teste variando o tamanho da entrada, mas desta vez, fazendo os algoritmos trabalharem o máximo possível, isto é, executá-los sempre no pior caso, ou o mais próximo possível dele. Obteve-se o seguinte gráfico, a partir dos dados de tempos coletados:

GRÁFICO 2. Tempo gasto em razão da dimensão da entrada explorando o pior caso dos algoritmos.



Diferentemente do *gráfico 1*, visto anteriormente, temos agora uma diferença notável de tempo de execução entre os algoritmos. As curvas de crescimento, refletem a complexidade no pior caso de cada paradigma, $O(n^2)$ para o método da programação dinâmica, e $O(n^5)$ para os restantes, que é uma complexidade consideravelmente superior à dinâmica. Apesar de possuírem a mesma ordem de complexidade, o algoritmo guloso possui um crescimento um pouco maior devido a sua função de complexidade, no qual o n^5 é somado a outros polinômios de menor ordem, aumentando assim o seu tempo.
Após a análise acima em que foram testados os algoritmos em seus piores casos, ficam alguns questionamentos como: que casos são estes? Como eles foram encontrados?
O gráfico e a análise do resultado abaixo discutem estas questões:

GRÁFICO 3. Tempo gasto em razão da configuração do conteúdo da matriz de entrada (100x100).



O gráfico acima demonstra o tempo dos algoritmos apresentados anteriormente, em função da quantidade de 1's, (representa o elemento sólido da placenta) que a matriz de entrada possui. As matrizes foram geradas através de um software que permite a escolha da porcentagem de chance conter o número 1 em cada ponto da matriz, essa chance pode ser visualizada no eixo horizontal do gráfico.

A presença de 0's na matriz de entrada, auxilia os algoritmos, pois eles evitam processos. É isso que acontece no algoritmo força bruta e guloso, quanto mais 1's possui a matriz, mais profundas são as checagens de conteúdo dentro da matriz, o que resulta em um custo computacional muito alto a cada iteração.

A estratégia dinâmica é a mais estável das três, pois ela sempre faz um número parecido de procedimentos. Ainda sim, quanto mais 0 tiver melhor para ela, pois, como explicado no tópico 2.4.3, quando o valor de um ponto da matriz de entrada é 0, é atribuído 0 na matriz de solução e não são necessários mais cálculos.

Ou seja, podemos dizer que em ambos os paradigmas, quanto maior a quantidade de 1's na matriz, maior será o custo de tempo para o programa ser executado.

4.2. Espaço

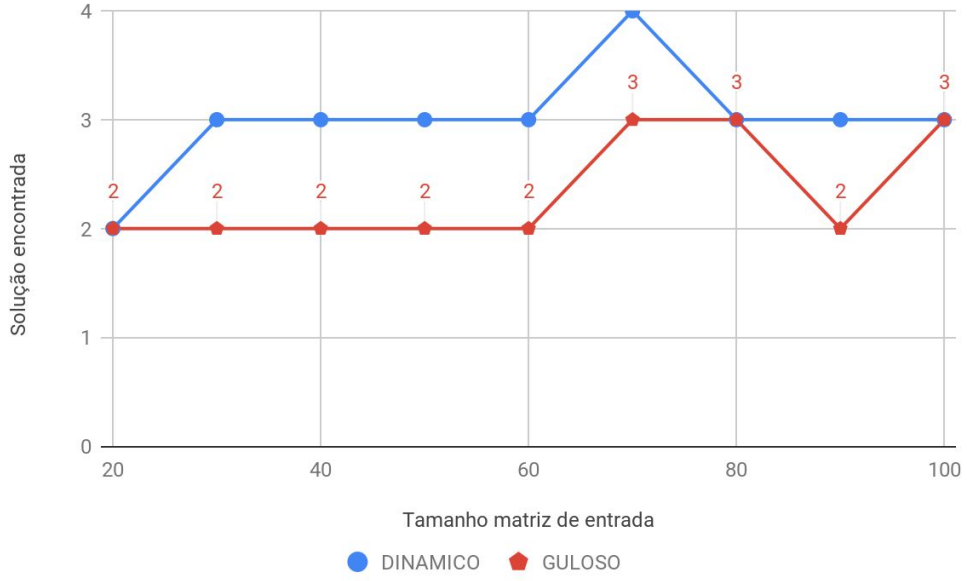
Quanto à análise de memória, utilizando o programa valgrind, foram feitos testes com situações adversas, para verificar a alocação de memória. Em todos, toda a memória alocada foi desalocada. (O resultado de um dos testes está no apêndice - seção 6).

4.3. Análise empírica da solução gulosa

A fim de obter uma análise mais particular dos resultados obtidos pelo método guloso, foram feitos testes com o algoritmo, pegando a solução dada por ele (maior submatriz encontrada) e comparando com os resultados da estratégia dinâmica, pois ela sempre acha a solução ótima. Este teste, teve como objetivo conhecer o grau de aproximação da solução gulosa em comparação com a solução ótima do problema.

A comparação pode ser vista no gráfico a seguir:

GRÁFICO 4. Solução encontrada em razão da estratégia de algoritmo usado variando o tamanho da matriz de entrada.



Através deste gráfico, vemos empiricamente que, a solução dada pelo guloso flutua entre a solução ótima do problema e uma solução próxima da ótima. Variando o tamanho da entrada, em 30% das ocasiões obtivemos a solução gulosa igual a solução ótima e nas outras 70% das ocasiões a solução foi abaixo da ótima, de qualquer forma, não foi tão discrepante o valor.

5. Conclusão

O intuito do trabalho prático de implementar o paradigma da força bruta (tentativa e erro), guloso e um terceiro paradigma, que se encaixasse ao problema proposto - a programação dinâmica - foi satisfeito.

Foi possível testá-los com diferentes entradas, realizar a análise de complexidade de cada um deles e por fim tirar as seguintes conclusões:

- Os projetos de algoritmos baseados em programação dinâmica, e tentativa e erro resultam sempre em uma solução ótima.

- A programação dinâmica é o mais eficiente deles em relação a complexidade de tempo, porém não em relação a complexidade de espaço.

- O método guloso não é o mais apropriado para este problema, pois não garante uma solução ótima, além de não ser tão simples encontrar a melhor escolha local para este problema. Algoritmos gulosos são mais indicados para encontrar soluções aproximadas de problemas de otimização, que não é o caso deste problema.

- Por natureza, algoritmos de força bruta são os mais custosos. Mas como foi dito (seção 4.1), a forma como foi implementado o algoritmo guloso, neste problema, consumiu mais tempo de processamento, no pior caso, do que o algoritmo de força bruta. Aliado, também, ao fato de que o algoritmo de tentativa e erro possui um refinamento que evita muitos cálculos desnecessários, dessa forma reduzindo consideravelmente o tempo em muitos casos.

6. Apêndice

6.1. Memória

Saída valgrind com uma matriz de entrada com tamanho 10x10:

```
==6157== Memcheck, a memory error detector
==6157== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==6157== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==6157== Command: ./tp2 -i entrada.txt -o a.txt
==6157==
==6157==
==6157== HEAP SUMMARY:
==6157==             in use at exit: 0 bytes in 0 blocks
==6157==   total heap usage: 16 allocs, 16 frees, 9,800 bytes allocated
==6157==
==6157== All heap blocks were freed -- no leaks are possible
==6157==
==6157== For counts of detected and suppressed errors, rerun with: -v
==6157== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

7. Referências bibliográficas

Algoritmos gulosos: definições e aplicações. Disponível em: <https://www.ic.unicamp.br/~rocha/msc/complex/algoritmosGulososFinal.pdf> Acesso em 21 de abr de 2019.

Programação Dinamica. Disponível em: <https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/dynamic-programming.html> Acesso em 18 de abr de 2019.

ZIVIANI, Nívio. Projeto de Algoritmos. 3ª Edição. São Paulo, 2011.