Programa para embalamento do KIT BOOM

Allyson Rodrigues, Thiago Souto

Universidade Federal de São joão Del Rei – UFSJ

Março de 2018

1 Introdução

Em linhas gerais, na computação, algoritmos são um conjunto de procedimentos bem definidos, que constituem uma das principais ferramentas utilizadas para solucionar problemas computacionais[1]. Existem vários paradigmas de programação, que são técnicas e "mindsets"que os programadores utilizam na criação dos algoritmos, alguns deles muito conhecidos e replicados. Um desses famosos paradigmas é o do **algoritmo guloso**. Algoritmos gulosos são aqueles que sempre fazem a escolha que parece a melhor no momento particular analisado, buscando que isso leve a solução globalmente ótima. Usa-se onde não há necessidade de reconsiderar decisões já tomadas[2]. Porém não há garantia que o algorimto guloso possa chegar na solução ótima do problema. Ainda assim, esse paradigma pode dar uma solução suficiente em muitos casos, sobretudo quando os problemas são mais simples. Este trabalho propõe uma algoritmo guloso que soluciona o problema dos Explosivos Jepslon, proposto na disciplina de AEDS III pelo professor Leonardo Rocha. O problema consiste em checar a validade de uma caixa em formato de matriz 6x6, que deve ser disposta de forma que não haja proximidade direta entre barras de bomba de mesmo tipo. Consideramos como tipo apenas a cor do explosivo. Nas figuras 1 e 2 podem ser observador dois exemplos de configurações possíveis, sendo uma considerada válida e outra inválida.

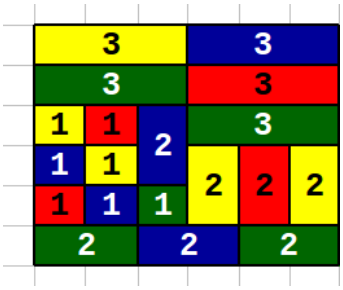

Figura 1. Exemplo de configuração válida

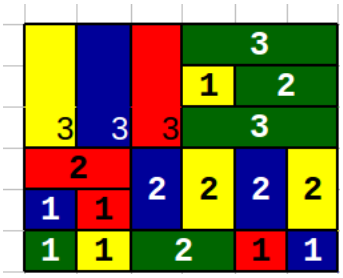

Figura 2. Exemplo de configuração inválida

Existem outras condições que invalidam a configuração da caixa além da colisão de bombas de mesmo tipo. Trataremos mais detalhadamente adiante.

No algoritmo, são realizados testes de verificação para observar se existe alguma bomba, independente do tamanho, do lado de outra com mesma cor, se existe uma bomba encima da

outra, se existe uma bomba cruzando outra bomba e se de fato a composição da caixa (quantidade exata de cada bomba) foi respeitada.

Neste trabalho, apresentaremos um algoritmo que propõe uma solução gulosa porque quando uma decisão é tomada ele joga a composição fora e segue pro próximo teste, guarda o resultado do teste no arquivo, que não pode ser alterado.

O Código foi escrito em C, compilado pelo GCC usando a ferramenta make, que automatiza a compilação.

Para executar o código siga as instruções no arquivo README.txt que está anexo ao código.

2 Solução Proposta para o Problema

Basicamente, o algoritmo faz vários testes independentes que no final se combinam para formar um resultado definitivo gerando a saída válido ou inválido para a configuração. Para isso, primeiro todas as informações dos arquivos de entrada são lidas e armazenadas em uma estrutura bem dividida e identificada. A composição da caixa fica sempre armazenada na memória, pois ela é usada para testar todas as opções de configuração. Então, é carregada um configuração individualmente, e nela são feitos todos os testes. Por fim, imprime-se o resultado no arquivo de saída e a configuração é descartada e outra configuração é chamada para testes. Assim segue até o final do arquivo. São realizados os seguintes testes:

- Verifica se a quantidade de bomba total da configuração é igual da composição;
- Verifica se a quantidade de cada tipo e tamanho de bomba tambem é igual a composição;
- Verifica se todos as bombas da configuração não extrapolam o espaço total da caixa;
- Verifica se todas as bombas tem sua posição inicial e final dentro da caixa;
- Verifica se não tem nenhuma bomba com a posições repetidas de outra;
- Verifica se existe alguma bomba de mesma cor/tipo na proximidade;
- Verifica se o meio da bomba de tamanho 3 tem alguma bomba de mesma cor/tipo encostando nela;
- Verifica se o espaço ocupado por cada bomba é o mesmo passado como parâmetro.

Assim que esses testes são executados, uma função verifica se a configuração passou por todos os testes e imprime o resultado no arquivo de saída.

3 Organização do Algoritmo

3.1 Modularização

O princípio da modularização é utilizado não apenas por requisito do trabalho prático mas, também, porque traz diversas vantagens para programação. Além de ser uma prática mais profissional de escrita de programas, a modularização pertmite a dinamização e organização do código, que é dividio em subprogramas formados por funções e procedimentos.

O código foi dividido em 6 arquivos sendo 3 códigos (.c) e 3 headers (.h) gerando uma “hierarquia” de forma que o arquivo main.c possui apenas a função main que serve simplesmente para chamar e executar várias vezes as funções do módulo ”inteligencia”. O arquivo main.h contém as estruturas de armazenamento das composições e das bombas, além de bibliotecas básicas da linguagem C.

Já o arquivo inteligencia.c funciona para fazer a parte de inteligência do código. É onde acontece a tomada de decisão. Essa parte pode tanto comunicar com funçao.c quanto chamar outra função dela mesma. O inteligencia.h contém as declarações das funções de inteligencia.c.

Por fim temos o funcoes.h e funcoes.c onde estão, respectivamente, os cabeçalhos e as funções. As funções desse módulo são trechos que são utilizadas em duas ou mais funçoes da inteligência. Então, essas operações são emcapsuladas e reaproveitadas, evitando a repetição de trechos de código idênticos.

A figura 3 representa visualmente essa comunicação e hieraqrquia do sistema.

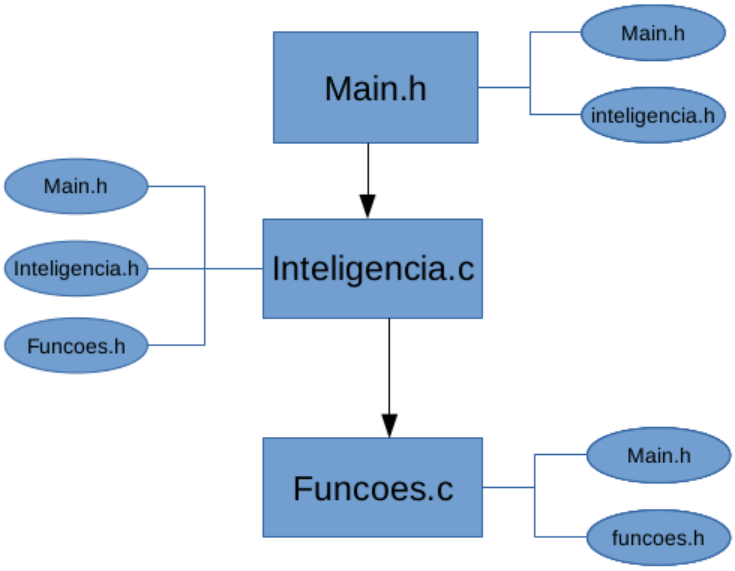

Figura 3. Fluxo de comunicação entre os módulos do programa

3.2 Entradas

O programa depende de dois arquivos no formato .txt passados como parâmetros na execução do programa. Esse arquivos tem um escopo obrigatório para o código conseguir 'entendê-lo'. Fica a cargo do usuário enviar os .txt nesse escopo pois o código não verifica se está errado. A única verificação feita no código é se existe um arquivo com os nomes fornecidos, caso não exista, o código avisará qual parâmetro está errado e pode gerar um arquivo de saída vazio. Entretanto, se os arquivos não seguirem o escopo, o algoritmo não identificará e pode gerar um erro indefinido.

Esses arquivos de entrada irão definir os parâmetros de quantidade e tipos de bombas contido em cada caixa. Este denominamos "Composição da Caixa"e o segundo irá definir como vai ser colocada cada bomba na caixa, o qual nomeamos "Configurações da Caixa".

A **Composição** tem o seguinte modelo: A primeira parte da linha é a quantidade de bombas, tem um espaço em branco e depois vem o tamanho da bomba seguido das abreviações das cores. (Am para amarelo, Vm para vermelho, Az para azul e Vd para verde) e que cada linha representa um tipo ou um tamanho diferente. Temos um exemplo de composição na figura 4.

```
2 3Az
2 2Az
3 1Az
2 2Vm
2 1Vm
2 3Am
1 1Am
2 2Vd
2 3Vd
```

Figura 4. Exemplo de composição da caixa

O arquivo de **Configuração** segue o modelo que contem várias configurações no formato ilustrado na figura 5, que contém a posição, o tamanho da bomba e seu tipo.

```
configuracao1
1 1 3 1 3Am
4 1 6 1 3Am
1 2 3 2 3Vd
4 2 4 2 1Az
5 2 6 2 2Vd
1 3 3 3 3Vd
4 3 4 3 1Am
5 3 5 3 1Vm
6 3 6 3 1Vm
1 4 1 5 2Vm
2 4 2 5 2Vm
3 4 3 5 2Az
4 4 4 5 2Vd
5 4 5 5 2Az
6 4 6 4 1Az
6 5 6 5 1Az
1 6 3 6 3Az
4 6 6 6 3Az
```

Figura 5. Exemplo de configuração da caixa

Como dito anteriormente, este segundo arquivo pode conter várias possíveis configurações, já o de composição sempre terá apenas uma composição lá dentro, que será o padrão para todas as configurações serem testadas se são ou não válidas.

3.3 Saída

Ao finalizar a execução, o programa guarda suas informações em um arquivo "saidaTP1.txt", esse arquivo é gerado no inicio do programa e caso dê algum erro para gera-lo o programa vai emitir uma mensagem e encerrar, caso exista um arquivo com esse nome ele é apagado e gerado um novo arquivo vazio. Esse arquivo é feito com o nome da configuração dois pontos e se ele é valido ou não é valido como o exemplo da figura 6

```
configuracao1:valido
configuracao2:nao-valido
configuracao3:valido
```

Figura 6. Exemplo de arquivo de saída

3.4 Código

O código foi desenvolvido em três etapas, a primeira parte é para alocar os espaços necessários, abrir e gerar os arquivos e guardar as informações na memória. Na segunda etapa são feitos os teste e gerados os resultados. E a terceira etapa é para deslocar e fechar os arquivos.

Estruturas Foram feitas 3 estruturas para guardar as informações da caixa, composições e bombas: Bomba, Composicao e Mala.

Na estrutura **Bomba** ficam armazenadas todas a informações de uma única bomba. São passados os parâmetros identificação, tipo, tamanho, x e y (posição) inicial e final. A identificação é única para cada bomba da configuração, a posição cartesiana de cada bomba tendo os valores iniciais (1,1) e a mesma lógica para posição final, colocando o tipo da bomba e seu tamanho.

Na estrutura **Composicao** fica armazenada a quantidade de bombas, em que cada tipo(Az, Am, Vd e Vm) é definido por um vetor e a quantidade fica armazenada na posição do tamanho da bomba subtraindo 1, pois nosso vetor em C começa do zero e vai ate 2, e os tamanhos vão de 1 a 3.

Por fim, na estrutura **Mala**, está a composição completa da caixa em que configuração fica. Nela fica armazenado o nome da configuração, a composição padrão, que é a composição é recebida do usuário, a composição final que é a composição extraída da configuração da bomba. As bombas, armazenadas em forma de vetor e um boolean para armazenar se a configuração é valida ou não.

Testes O código funciona fazendo vários testes para ver se é valida ou não a caixa. Assim, cada teste fica responsável por verificar uma única propriedade, e no final se todas derem válida quer dizer que a caixa á válida. Porém se apenas uma der inválida, já se descarta a caixa inteira nem executando os outros testes. A estrutura Mala tem um verificador que é setado como válido, e ao fim de cada teste, se for verificado algum erro, é alterado para inválido, e dentro de cada teste verifica se já foi alterada a validade da caixa para executar o teste. Caso sim, já sai da função. Enfim, realiza-se o teste para verificar se o espaço ocupado por cada bomba na matriz representa realmente o tamanho passado como parâmetro.

Primeiro é feito um teste para verificar quantidade total de bomba dentro da configuração é igual a quantidade total da composição que o usuário passou. Segundo, é verificado se a quantidade de cada tipo de bomba também são iguais. Terceiro, é conferido se a caixa não passa de 36 centímetros quadrados. Quarto, é feito uma verificação se nenhuma bomba extrapola o tamanho máximo da caixa que é de 6 centímetros ou se foi colocada em um espaço que não existe na caixa. Quinto, verifica se não existe uma bomba com a mesma posição de outra. Sexto, é verificado se existe alguma bomba igual na proximidade. Sétimo, é um teste exclusivo para bombas de tamanho 3 centímetros, é um teste que verifica se a posição do meio da bomba cruza alguma coisa.

Resposta Ao fim de todos os testes, é chamada uma função que vai pegar o verificador e imprimir no arquivo o nome da mala com a resposta valida ou não valida.

4 Análise de Complexidade

Complexidade das Funções A complexidade das funções em relação a quantidades de bombas (n) no modulo funcoes.c são apresentadas a seguir:

1. "lerArqSTR": A função de complexidade é $f(n) = 1$ logo é $O(1)$, pois essa função só faz uma leitura de string do arquivo que é mandado por parâmetro.
2. "arquivo_vazio": A função de complexidade é $f(n) = 6.(1)$ também é $O(1)$ porquê faz 2 atribuições, uma verificação, e desaloca uma variável.
3. "alocar_composicao": Essa função é mostrada pela função de complexidade $f(n) = 5 + 3 * 4$, pois dentro dela existe um for que faz quatro iterações. Porém esse loop só executa 3 vezes de forma estática, e independente do n, assim sendo uma função $O(1)$.
4. "desalocar_composicao": A função de complexidade é $f(n) = 5 * 1$ pois ocorre somente 5 "free" logo também sendo de complexidade de $O(1)$.
5. "ler_composicao": Esse trecho tem a $f(n) = 3$ pois ela somente faz uma soma e depois duas iterações.

As complexidades das funções de leitura na memoria os dados dos arquivos enviado pelo usuário em relação a quantidades de bombas na configuração (n) , a quantidade de configuração (m) e/ou a quantidade de composições (c) no modulo inteligencia.c são:

1. "criar_mala": Criar mala chama duas vezes a função de alocar mala que é $O(1)$ e faz um incremento, assim a função é $2 * O(1) + 1$ que seria uma função $O(1)$.
2. "destruir_mala": Destruir mala chama agora duas vezes a função de desalocar a composição que tem o comportamento também de $O(1)$ assim sendo ficaria uma função de complexidade de $2 * O(1)$ que faz essa função ser $O(1)$.
3. "criar_composicao": Essa função primeiro chama a função que verifica o arquivo que é $O(1)$ e cria algumas variáveis, depois entra em um laço que verifica todos as composições, assim essa função tem a função de complexidade de $4 + O(1) + C * 6$ ficando uma função $O(N)$.
4. "quantidade_configuracao": Essa função primeiro chama outra função de verificação e se o arquivo existe que é $O(1)$, cria 3 variáveis, desaloca um ponteiro de arquivo e entra em um laço até o fim do arquivo de configuração chamando duas funções de leitura e a de ler composição que é $O(1)$ e assim fica uma função de complexidade de $4 + O(1) + (M * N) * (5 * (O(1) + 1))$. Resultando em uma função $O(N * M)$
5. "alocar_bomba": Essa função primeiro cria uma variável e depois aloca um espaço e faz a atribuição do laço. Então, entra em um laço fazendo 7 atribuições dentro e faz a comparação, ficando com a funçao de complexidade $3 + N * 8$ assim sendo uma função $O(N)$.
6. "desalocar_bomba": A função cria uma variável, faz uma atribuição no laço e executa o laço n vezes desalocando um espaço e incrementando, e por ultimo desaloca uma bomba. Assim temos uma função $3 + N * 2$, portanto, uma complexidade de $O(N)$.

7. “quantidade_bomba”: A função cria 4 variáveis faz uma verificação chamando uma função $O(1)$, entra em um loop fazendo uma atribuição, e executa m vezes, ainda dentro do loop entra em outro laço que é executado n vezes fazendo 5 atribuições, 5 locações e 5 desalocações fechando o primeiro e segundo laço, então para terminar é feito 1 decremento e 3 desalocação assim ficamos com a função de complexidade $10 + m * (4 + n * 17)$ Assim temos uma função $O(N * M)$.
8. “criar_configuracoes”: A função cria 3 variáveis e chama 3 funções $O(1)$ então entra em um laço que é executado n vezes, e dentro do loop é chamado 6 funções de leitura que é $O(1)$ e feito 9 atribuição e uma verificação e no fim do código e fora do loop é feito uma atribuição. Assim a função de complexidade da função fica $4 + 3 * O(1) + n * (6 * O(1) + 10)$, portanto, $O(N)$.

A seguir, as funções de complexidade dos testes em função da quantidade de bombas (N) do modulo de inteligencia.c:

1. “verificar_quantidade”: Nessa função são feita apenas duas configurações e uma atribuição de modo que essa função seja $O(1)$.
2. “verificar_configuracao”: Nessa função temos uma verificação inicial, é criado uma variável e temos um loop que executa apenas 3 vezes e dentro desse loop são feitos comparação de modo que é feita apenas uma atribuição assim temos a função de complexidade $3 + 3 * 3$. Assim temos uma complexidade de $O(1)$.
3. “verificar_max_espaco”: Primeiro na função é feita uma verificação e criadas duas variáveis depois temos um loop executando N vezes e fazendo 2 incrementos e uma verificação. Fora do laço, por último é feita uma verificação fazendo a função ficar $5+N*3$ que é uma complexidade $O(N)$.
4. “verificar_tamanho”: Primeiro é feito uma verificação e depois criado uma variável, então entramos em um laço que incrementa e executa n vezes fazendo verificações e setando valores. Temos uma função de complexidade de $2 + N * 7$ fazendo a função ficar $O(N)$.
5. “verificar_tamanho_bomba”: Primeiro é feito um teste e logo após são declaradas duas variáveis então entramos em um laço que faz uma verificação e executa mais duas verificações. Assim chegamos a uma função de $3 + N * 5$ e concluímos que o código dessa função é $O(N)$.
6. “verificar_proximidade”: A função primeiro faz uma verificação e depois entra em um loop que é executado $n$vezes e dentro desse loop entramos em outro loop que é executado $n-1$ vezes e nesse loop são feitas 6 verificações e um incremento assim temos uma função $5+N*((N-1)*7)$ essa função é $O(N^2)$.
7. “verificar_posicao_repetida”: Nessa função é feita uma verificação e criadas 2 variáveis. Setamos essa variável e entramos em um loop que executa n vezes e depois em um loop que executa $n - 1$ vezes, nesse segundo loop são feitas 9 operações básicas de $O(1)$ assim temos uma função de complexidade $5 + N * ((N - 1) * 12)$ e chegamos que a função é $O(N^2)$.
8. “verificar_meio”:A função faz uma verificação depois cria 4 variáveis e entra em um loop que executa n vezes nesse loop são 3 testes e entra em outro loop que executa $n-1$ vezes e executa mais 3 teste ficando uma função de $5 + N * ((N - 1) * 9)$ assim também temos uma função $O(N^2)$.
9. “resposta”: Essa função faz apenas uma verificação e imprime e um arquivo ficando $O(1)$.

Complexidade do código Agora apresentamos a complexidade do código em relação a quantidade de bombas na configuração (N) e quantidade de configurações (M) e a quantidade de composição (C) enviada, assim iremos demonstrar a função geral do código.

Na main, primeiro é declarado as variáveis e ponteiros e estruturas somando o total de de 6. Daí são feita duas verificações, então é chamada uma função de criar mala que é $O(1)$, criar composição que é $O(N)$, depois uma função de quantidade de configuração que é $O(N * M)$. Então é alocado um espaço que é $O(1)$ e chamada a função de quantidade de bombas que tem complexidade de $O(N * M)$.

Então o código entra em um loop que é executado M vezes e nesse loop primeiro é alocado um espaço de memória que é $O(1)$. Então são chamada as funções de alocar bomba e criar configuração que tem complexidade de $O(N)$. Então são chamados os testes. Primeiro são os de verificar quantidade e configuração que são $O(1)$, então são chamados os de espaço máximo e de tamanho da bomba que são $O(N)$ por fim dos testes são os de verificar posição repetida da bomba, proximidade dela e se o meio das bombas de 3 estão próximas de bombas iguais que tem complexidade de $O(N^2)$ e por fim do loop temos a função de imprimir no arquivo que é $O(1)$, de desalocar bomba que é $O(N)$ o “free” de ponteiro e incremento de duas variáveis. E por fim do

código todo 3 “free” de ponteiro e a chama da função de destruir mala que tem complexidade de $O(1)$.

Obtemos uma função:

$$F(n) = 11 + [2 * O(1)] + [2 * O(N * M)] + \\ M * 5 + [3 * O(N)] + [3 * O(1)] + [3 * O(N)] + [3 * O(N^2)] \quad (1)$$

Assim conclui-se que o código tem complexidade assintótica de $O(M * N^2)$ como explicado anterioremte as configurações são limitadas a 36 bombas, então a quantidade de configurações que faz a função realmente ter um comportamento assintotico.

5 Resultados Obtidos com os Testes

Após analisarmos a complexidade do código, foram feitos alguns testes no algoritmo a afim de verificar seu comportamento. Para realizar os testes utilizamos apenas uma máquina, um notebook Samsung com processador I7 quinta geração, usando o sistema operacional Linux Mint 18 com compilador gcc 5.4.

Os testes foram realizados **utilizando como entradas modelos de configurações** de três tipos.

- Bombas de tamanho 1
- Bombas de tamanho 3
- Bomas de tamanhos variados (1, 2 e 3)

Dentro dessas três variações foram feitas configurações válidas e inválidas e em cada um desses formatos foram feitos arquivos contendo 1, 5, 10, 20, 30, 40, 50, 100, 500, 1000, 2500, 5000, 7500 e 10000 configurações totalizando 84 entradas diferentes. Essas entradas foram rodadas com três marcadores de tempo diferente, um pegando o **tempo de relógio**, outro pegando o **tempo do processador** e um terceiro pegando o **tempo de threads**, chegando um total 252 testes de entrada.

Desconsideramos nos testes os tempos de alocar as composições e imprimir o resultado no arquivo, pois a alocação da composição ocorre apenas uma vez, gerando um resultado irrisório no resultado efetivo de custo do algoritmo. Pelo motivo oposto, desconsideramos a escrita em arquivo, pois acesso e escrita em arquivo geraria um aumento expressivo no tempo, o que causaria um resultado distorcido da realidade.

A partir dos testes foi possível gerar os seguintes gráficos, representados nas figuras 7, 8, 9. Cada gráfico é uma relação da quantidade de entradas por tempo medido de três maneiras distintas. Reiteramos que utilizamos como quantidade de entrada o número de arquivos de configuração passados. Cada uma das curvas plotadas nos gráficos representa um tipo de entrada específica, baseada no número de arquivos de configuração passados. Como dito anteriormente, temos 6 tipos diferentes de entradas, a saber:

- Configuração Válida com Bombas de tamanho 1 (Válida - bombas de tamanho 1)
- Configuração Válida com Bombas de tamanho 3 (Inválida - bombas de tamanho 3)
- Configuração Válida com Bombas de tamanhos variados (Válida - bombas de tamanhos variados)
- Configuração Ínvalida com Bombas de tamanho 1 (Inválida - bombas de tamanho 1)
- Configuração Inválida com Bombas de tamanho 3 (Inválida - bombas de tamanho 3)
- Configuração Inválida com Bombas de tamanhos variados ((Inválida - Bombas de tamanhos variados))

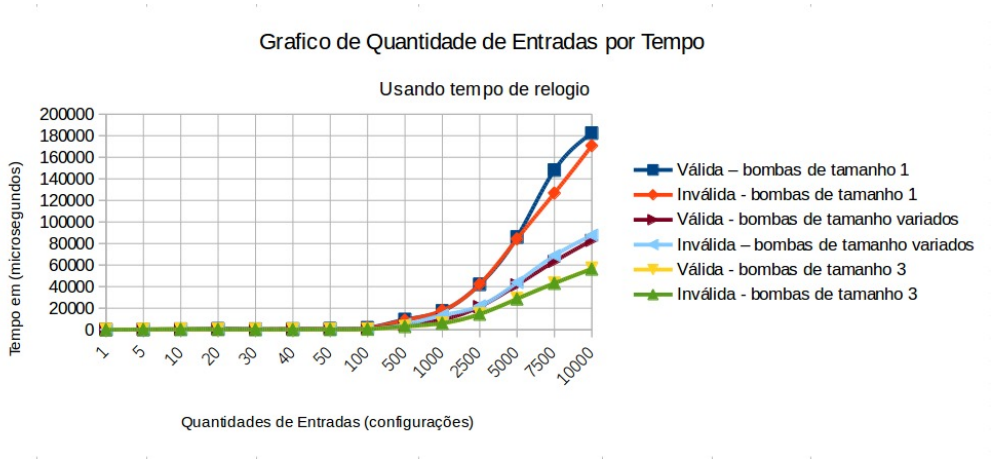

Figura 7.

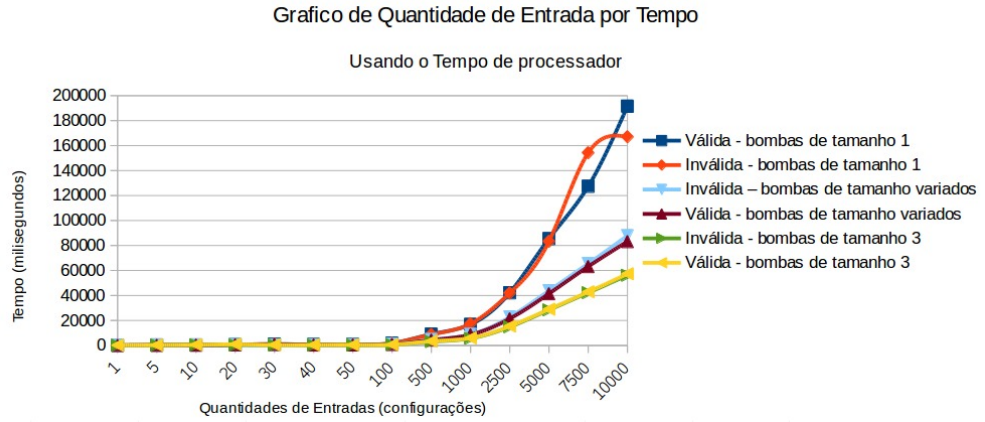

Figura 8.

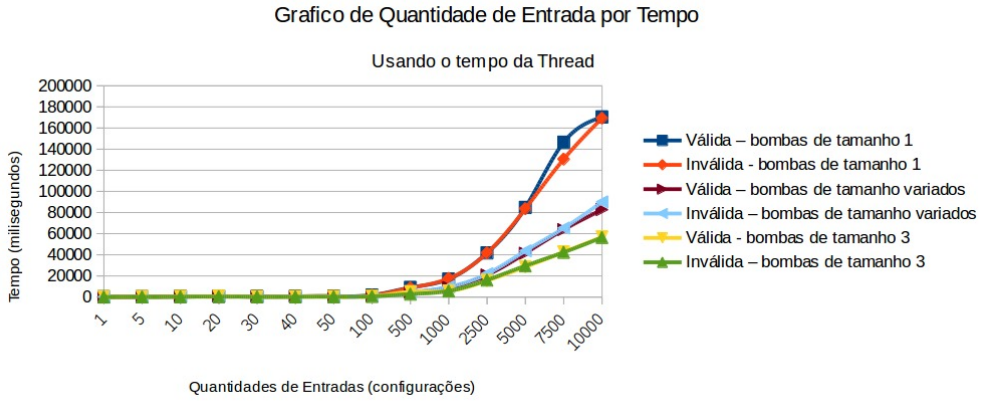

Figura 9.

A diferença entre o tempo de relógio, e tempo do computador (processador e Thread) foram muito pequenas podendo até desconsiderar nos resultados desses testes. O que não quer dizer que em testes com entradas muito maiores esses resultados sejam de fato insignificantes, pois mesmo que pequena, houve uma diferença. Porém nesse caso, foram insignificantes. Percebemos claramente que há um melhor caso, um pior caso e um caso médio, quando observamos qual tipo de entradas/configurações serão passadas. Quando temos bombas de tamanho 1 caímos o pior

caso. Em configurações que passam apenas bombas de tamanho 3 temos o melhor caso e o caso médio é quando as configurações possuem bombas de tamanhos variados. Por isso as curvas são as mais inclinadas no pior caso, as menos inclinadas no melhor caso e as do caso médio ficam entre as duas.

Como analisado na parte de análise de complexidade, concluiu-se que o algoritmo possui comportamento assiontótico descrito por $O(M * (N^2))$ em que N é a quantidade de bombas na configuração e M é a quantidade de configurações. Esse comportamento é bem descrito nos gráficos, onde todas as curvas se comportam de forma exponencial. Destaca-se que há ângulo maior da curva na medida em que o número de bombas aumenta e vemos a importancia do M que é quem faz a função não se limitar ao máximo que N pode assumir. Ao analisar as curvas das configurações que tinham bombas de tamanho 1, consequentemente contendo mais bombas já que o tamanho é limitado, vemos que é bem mais inclinada do que o gráfico das bombas de tamanho 3. Já o gráfico das configurações de bombas mistas ficou entre as outras duas curvas e constatamos que o tamanho da bomba interfere indiretamente no crecimento.

Também pode-se inferir a partir do gŕafico que crecimento está diretamente ligado com a quantidade de configurações da entrada. Isso é mostrado na análise quando multiplicamos o N^2 por M, o que faz a curva aumentar.

Com isso temos que nosso pior caso são as configurações serem completas (preencherem toda a matriz) e ter apenas bombas de tamanho 1.

6 Conclusão

A implementação do algoritmo apresentado neste trabalho foi conduzida de forma razoavelmente simples e o projeto correu dentro do planejado e os resultados foram satisfatórios. No caso específico do empacotamento das bombas da "empresa Jepslon"o paradigma guloso atendeu as necessidades e apresenta-nos uma solução eficaz para o problema em questão. Destacamos que este projeto apresentou-nos alguns desafios, sobretudo no que diz respeito à organização do código através das seus módulos e estruturas de dados. Foi possível visualizar de forma concreta os conceitos desenvolvidos em sala de aula até então, com destaque para as técnicas para analisar algoritmos. Tais conceitos revelaram-se de grande valia para projetos de algoritmos, uma vez que tal ferramental teórico torna possível uma análise precisa sobre o custo do algoritmo, possibilitando, assim, escolhas que direcionam o projeto no sentido da eficiência. Como trabalhos futuros, pretende-se aprofundar nas diferentes técnicas e paradigmas de programação, no intuito de alcançar um nível de programação cada vez mais eficiente e profissional, portanto, utilizando estratégias que envolvem menor custo de tempo e memória.

7 Referências Bibliográficas

[1] Cormen, H. [et al.]. *Algoritmos: Teoria e Prática*. Elsevier, 2012.

[2] Ziviani, Nívio. *Projeto de Algoritmos: com implementação em Pascal e C*. Cengage Learning, 2013.