Universidade Federal
de São João del-Rei

PROJETO E ANÁLISE DE ALGORITMOS

Trabalho prático 1 - Kit BOOM

Guilherme Francis Ramon Coelho

Outubro de 2024

1 Introdução

O problema em questão tem como objetivo principal otimizar o embalamento do Kit BOOM, um conjunto de explosivos desenvolvido pela empresa fictícia Explosivos Jepslon. Esse kit é composto por explosivos de quatro cores distintas (azul, amarelo, vermelho e verde), cada um com um comprimento entre 1 e 3 centímetros e uma largura fixa de 1 centímetro.

Os explosivos precisam ser acomodados em uma caixa de 6x6 centímetros, respeitando rigorosas condições de segurança para evitar explosões acidentais. Para garantir a segurança, o programa verifica três critérios essenciais para uma configuração válida:

- Posicionamento Seguro: Explosivos da mesma cor não podem estar em posições adjacentes, tanto na horizontal quanto na vertical.
- Uso Eficiente do Espaço: A configuração dos explosivos deve respeitar a área total da caixa, que é limitada a 36 cm^2.
- Correspondência com o Kit: A configuração proposta deve corresponder ao kit em termos de quantidade, tamanho e cor dos explosivos, garantindo que cada configuração utilize apenas os materiais disponíveis no kit inicial.

O programa recebe os dados necessários para essas verificações por meio de dois arquivos de texto (um para o kit e outro para a configuração), fornecidos como parâmetros durante a execução. Após a leitura, esses dados são processados e validados por meio de uma série de funções que examinam a consistência e completude dos parâmetros, o cumprimento das condições de segurança e o correto uso dos recursos.

Para avaliar a eficiência do programa, algumas configurações serão analisadas, utilizando as funções getrusage e gettimeofday para medir os tempos de execução. Dessa forma, é possível distinguir entre os tempos de computação e os tempos de entrada e saída, analisando tanto os tempos de usuário (tempo gasto em operações de processamento de dados pelo programa) quanto os tempos de sistema (tempo utilizado pelo sistema operacional para operações auxiliares, como entrada e saída de dados e alocação de memória). Essa distinção permite compreender melhor a relação entre o tempo de relógio (tempo total) e os tempos computacionais, possibilitando ajustes para otimizar o desempenho geral do programa.

A implementação utiliza leitura dinâmica dos arquivos e verifica os critérios por meio de funções específicas, como a detecção de adjacência entre explosivos de mesma cor e a verificação dos limites de área. A estrutura modular do programa facilita a manutenção e permite ajustes futuros na lógica de embalamento e segurança, garantindo que o kit esteja configurado de maneira segura e eficiente e assegurando o cumprimento de todas as regras de segurança antes do uso.

2 Desenvolvimento

2.1 Estruturas de Dados

Para resolver o problema proposto, foram definidos dois tipos abstratos de dados para melhor

- `Bomba`: representa uma bomba configurada com os atributos `cor`, `tam`, `x_inicial`, `y_inicial`, `x_final` e `y_final`. Estes atributos representam a cor, o tamanho e as posições iniciais e finais da bomba, usados para validar as posições dentro da área da caixa.
- `KitExplosivo`: representa cada explosivo no kit, com atributos `cor`, `tam` e `quantidade`, permitindo comparar a configuração desejada com a disponibilidade do kit.

2.2 Modularização

O código foi dividido em três módulos principais para facilitar a organização e a clareza da solução: o módulo de entrada e saída, o módulo de configuração do kit, e o módulo principal. Abaixo, descrevemos as funções de cada módulo, suas responsabilidades e como interagem entre si.

2.2.1 Módulo de Entrada

Este módulo é responsável pelo processamento dos parâmetros de entrada e pela leitura dos arquivos de configuração. Suas principais funções incluem:

- **processar_entrada**: Recebe e processa os argumentos passados por linha de comando, verificando se os arquivos de kit e configuração foram especificados. Em caso de erro, exibe uma mensagem e encerra o programa. Essa função espera como argumentos os nomes dos arquivos de entrada para o kit e para a configuração, precedidos pelos parâmetros -k e -c, respectivamente.
- **ler_kit**: Lê o arquivo que contém as informações do kit de explosivos, incluindo a cor, tamanho e quantidade de cada explosivo. Aloca dinamicamente o espaço necessário para armazenar esses dados e realoca o espaço conforme necessário.
- **ler_config**: Lê o arquivo de configuração com os detalhes de cada explosivo posicionado, incluindo coordenadas iniciais e finais, tamanho e cor. Aloca o espaço necessário e realoca conforme a quantidade de explosivos.

Essas funções trabalham juntas para garantir que os dados de entrada estejam corretos e prontos para serem usados no restante do programa.

2.2.2 Módulo de Configuração do Kit

Este módulo contém a lógica de verificação e validação do kit e da configuração, incluindo restrições como limites e correspondências de quantidade e características dos explosivos. As principais funções são:

- **validar_configuracao**: Verifica se cada explosivo na configuração está dentro dos limites da caixa, se não há explosivos adjacentes de mesma cor, e se a área total dos explosivos não excede o limite permitido.
- **verificar_correspondencia**: Confere se a quantidade e as características dos explosivos na configuração correspondem ao kit. Utiliza as informações de cor e tamanho para garantir que cada explosivo da configuração esteja presente no kit com a quantidade correta.
- **adjacencia**: Auxiliar de `validar_configuracao`, verifica se dois explosivos estão em posições adjacentes.
- **dentro_dos_limites**: Outra função auxiliar, garante que um explosivo está posicionado dentro dos limites da caixa (6x6).
- **calcular_tamanho**: Calcula a área de cada explosivo com base em suas coordenadas iniciais e finais.

Essas funções garantem que a configuração obedeça às restrições do problema, assegurando a validade e a segurança do posicionamento dos explosivos.

2.2.3 Módulo Principal

O módulo principal integra todas as funções para execução do programa. Suas etapas incluem:

- Inicializar variáveis e estruturas para armazenar os dados do kit e da configuração.
- **start_timer** e **stop_timer**: Inicia e encerra o cronômetro para medir o tempo de execução do programa, usado para análise de desempenho.
- **processar_entrada**: Chama a função do módulo de entrada e saída para processar os parâmetros e garantir que os arquivos de entrada estejam definidos.
- **ler_kit** e **ler_config**: Carrega os dados do kit e da configuração, respectivamente.
- **verificar_correspondencia** e **validar_configuracao**: Realiza as validações necessárias entre o kit e a configuração.
- Finaliza o programa, liberando a memória alocada.

Esse módulo coordena o fluxo de execução do programa, chamando as funções necessárias na sequência apropriada para atingir o objetivo do código.

2.2.4 Medição de desempenho

Esse módulo coleta dados sobre o tempo de execução total do programa e o uso de CPU. Ele permite fazer a distinção entre o tempo gasto efetivamente no processamento de dados e o tempo dedicado a operações auxiliares do sistema.

- **start_timer()**: Inicializa a contagem do tempo de execução, registrando o horário atual no início do programa ou de uma operação específica.
- **stop_timer()**: Registra o horário atual no fim do programa ou da operação. Essa função permite calcular o intervalo de tempo até o final da execução.
- **print_timer()**: Calcula e exibe o tempo total de execução em segundos desde o **start_timer**. Também utiliza a função **getrusage** para mostrar o tempo de CPU usado:

2.3 Fluxo de Execução do Código

Abaixo está o diagrama de fluxo que representa as etapas principais do programa. Este diagrama ilustra o processo desde o recebimento dos parâmetros de entrada até a finalização do programa, passando pelas etapas de leitura dos arquivos, validações e verificação de correspondência entre o kit e a configuração.

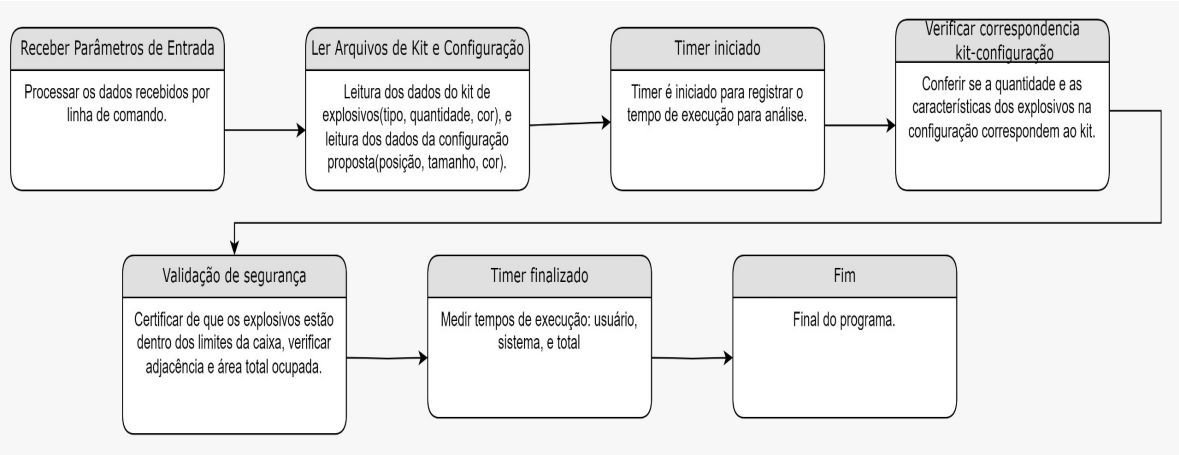

Figura 1: Diagrama de fluxo de execução do código

2.4 Função verificar_correspondencia

A função `verificar_correspondencia` é responsável por verificar se os explosivos especificados na configuração correspondem ao kit, em termos de cor, tamanho e quantidade. Esta função percorre todos os explosivos listados na configuração e, para cada explosivo, verifica se ele existe no kit. Se um explosivo não é encontrado, o programa emite um erro e interrompe a execução.

```
        for (int i = 0; i < num_config; i++) {
            int encontrado = 0;
            for (int j = 0; j < num_kit; j++) {
                if (strcmp(kit[j].cor, config[i].cor) == 0 && kit[j].
                    tam == config[i].tam) {
                    quantidade_explosivos[j]++;
                    encontrado = 1;
                    break;
                }
            }

            if (!encontrado) {
                printf("Erro: explosivo %d%s na configura  o n o
                    existe no kit!\n", config[i].tam, config[i].cor);
                return 0;
            }
        }
```

Listing 1: Trecho da função verificar_correspondencia

Neste trecho, o primeiro `for` itera sobre os explosivos na configuração. Em cada iteração, ele procura o explosivo correspondente no kit (segundo `for`). Se uma correspondência é encontrada, a quantidade de explosivos é incrementada; caso contrário, o programa emite uma mensagem de erro e interrompe a execução.

Isso mostra que as funções dependem umas das outras, e a função **validar_configuracao** é a mais complexa. A análise indica que, conforme o número de bombas e tipos de explosivos aumenta, o tempo de execução do módulo pode crescer, especialmente quando é preciso validar e comparar configurações.

2.5 Exemplos de entrada e saída

Para teste do programa, algumas entradas foram testadas separadamente e a partir delas, avaliado o seu desempenho, levando em conta o tempo tempo total de execução e o tempo de CPU.

3 Análise de complexidade

A análise de complexidade computacional permite compreender a eficiência e os limites de desempenho do código. Ao avaliar os melhores, médios e piores cenários, podemos prever o comportamento do programa em diferentes condições, especialmente considerando o número de itens em config e kit assim como mostra a Tabela 1, onde n representa o número de explosivos em config e m o número de itens em kit .

CASO	TOTAL DE OPERAÇÕES	COMPLEXIDADE
Melhor	$n + m$	$O(n + m)$
Médio	$n^2 + n \cdot m$	$O(n^2 + n \cdot m)$
Pior	$n^2 + n \cdot m$	$O(n^2 + n \cdot m)$

Tabela 1: Tabela de Complexidade do Código

3.1 Melhor Caso

No cenário de melhor caso, as funções **verificar_correspondencia** e **validar_configuracao** executam o menor número possível de operações internas. Isso ocorre quando a correspondência de cada explosivo em config com os itens de kit é encontrada rapidamente, sem necessidade de percorrer todo o array para cada item. Com isso temos o número total de operações sendo

$$\sum_{i=1}^{n} 1 + \sum_{j=1}^{m} 1$$

A complexidade para o melhor caso é reduzida para $O(\mathrm{n} + \mathrm{m})$.

3.2 Caso Médio

No caso médio, assumimos que o programa realiza a maioria das operações sem que todas as verificações adicionais sejam necessárias, mas ainda assim, algumas operações de comparação e validação serão executadas para uma fração significativa dos elementos em config e kit. Assim, temos o somatorio sendo

$$\sum_{i=1}^{n} \sum_{j=1}^{m} 1 + \sum_{i=1}^{n} \sum_{k=i+1}^{n} 1$$

simplificando esse somátorio temos

$$n \cdot m + \frac{n(n-1)}{2}$$

Portanto, a complexidade permanece em $O(\mathrm{n}^2 + \mathrm{n} \times \mathrm{m})$. O tempo de execução tende a crescer proporcionalmente ao aumento do número de elementos em config e kit, embora ainda seja menos custoso do que o pior caso.

3.3 Pior Caso

O pior cenário ocorre quando o programa precisa executar todas as operações internas de cada função, o que acontece quando cada explosivo em config requer uma verificação completa contra todos os itens em kit e todas as verificações de adjacência

precisam ser processadas. Isso significa que, em **validar_ configuracao**, todas as comparações de adjacência são realizadas, enquanto em **verificar_correspondencia**, cada explosivo em config é comparado com todos os kits em kit. Deixando a complexidade também de $O(\mathrm{n}^2 + \mathrm{n} \times \mathrm{m})$. Esse cenário representa o limite superior da complexidade computacional que o programa pode enfrentar, com cada interação resultando em um aumento no tempo de execução à medida que o tamanho dos dados cresce. O número de operações é o mesmo que no caso médio.

4 Resultados e testes

4.1 Configuração Inválida

Na primeira rodada de testes usamos a configuração inválida de acordo com a tabela 2.

Configuração				**Kit**		
Início (X, Y)	**Fim (X, Y)**	**Tamanho**	**Cor**	**Quantidade**	**Tamanho**	**Cor**
(1,1)	(3,1)	3	Am	2	3	Az
(4,1)	(6,1)	3	Am	2	2	Az
(1,2)	(3,2)	3	Vd	3	1	Az
(4,2)	(4,2)	1	Az	2	2	Vm
(5,2)	(6,2)	2	Vd	2	1	Vm
(1,3)	(3,3)	3	Vd	2	3	Am
(4,3)	(4,3)	1	Am	1	1	Am
(5,3)	(5,3)	1	Vm	2	2	Vd
(6,3)	(6,3)	1	Vm	2	3	Vd
(1,4)	(1,5)	2	Vm			
(2,4)	(2,5)	2	Vm			
(3,4)	(3,5)	2	Az			
(4,4)	(4,5)	2	Vd			
(5,4)	(5,5)	2	Az			
(6,4)	(6,4)	1	Az			
(6,5)	(6,5)	1	Az			
(1,6)	(3,6)	3	Az			
(4,6)	(6,6)	3	Az			

Tabela 2: Configuração das Bombas e Kit Disponível

Analisando a tabela 2, já da para notar incongruências, pois a quantidade de explosivos em algumas cores e tamanhos excede o que está disponível no kit, como nos casos dos explosivos de cor azul (tamanhos 3 e 2) e verde (tamanho 3). Além disso, temos também explosivos adjacentes assim como podemos visualizar na figura 2 (Configuração Inválida).

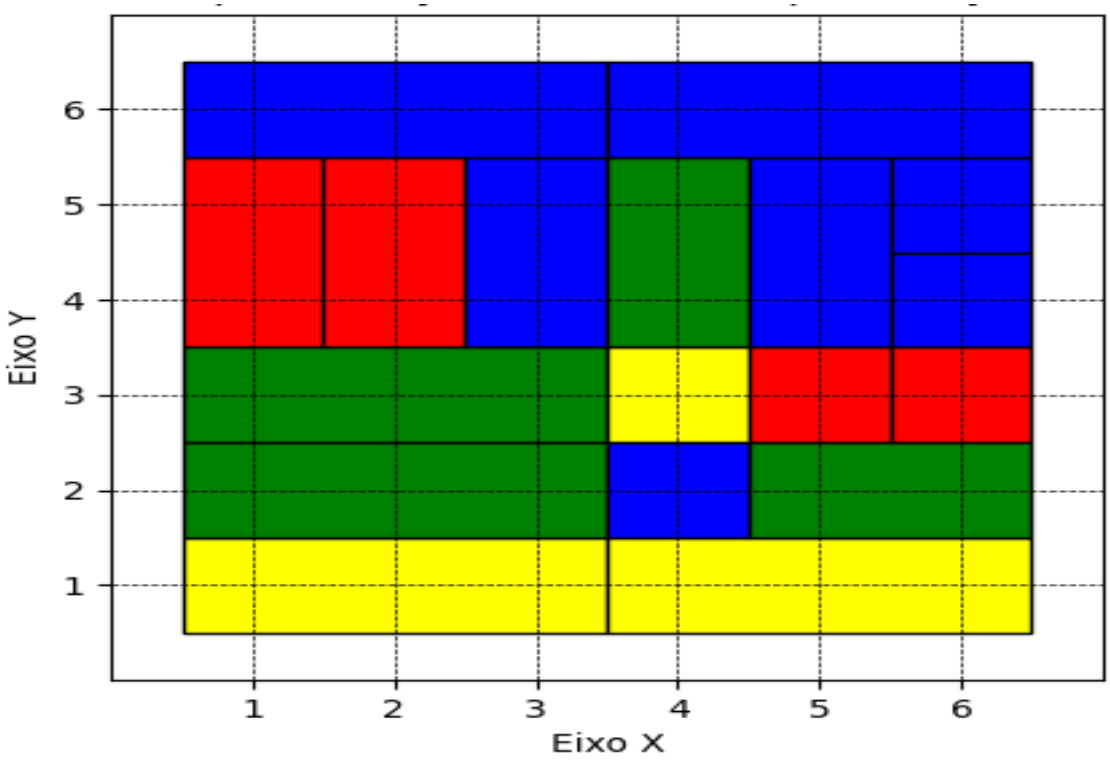

Figura 2: Configuração Inválida

Utilizando esse exemplo, obtivemos uma média de 0.0006125 segundos usando gettimeofday da bliblioteca $< sys/time.h >$.

4.2 Configuração Válida

De forma análoga, para a configuração válida utilizamos os seguintes dados, observados na tabela 3. Após a leitura da configuração e o kit, fizemos um gráfico

Configuração				**Kit**		
Início (X, Y)	**Fim (X, Y)**	**Tamanho**	**Cor**	**Quantidade**	**Tamanho**	**Cor**
(1,1)	(3,1)	3	Am	3	3	Az
(4,1)	(6,1)	3	Vd	1	2	Az
(1,2)	(3,2)	3	Az	2	3	Vm
(4,2)	(6,2)	3	Vm	1	1	Vm
(1,3)	(1,3)	1	Vd	2	3	Am
(2,3)	(2,3)	1	Am	1	2	Am
(3,3)	(3,3)	1	Vm	1	2	Vd
(4,3)	(4,3)	1	Vd	1	3	Vd
(5,3)	(6,3)	2	Vm	2	1	Vd
(1,4)	(2,4)	2	Az	1	1	Am
(3,4)	(4,4)	2	Am	1	2	Vm
(5,4)	(6,4)	2	Vd			
(1,5)	(3,5)	3	Vm			
(4,5)	(6,5)	3	Az			
(1,6)	(3,6)	3	Az			
(4,6)	(6,6)	3	Am			

Tabela 3: Configuração das Bombas e Kit Disponível

mostrando a vizualização em um plano cartesiano da verificação válida, demonstrada na figura 3.

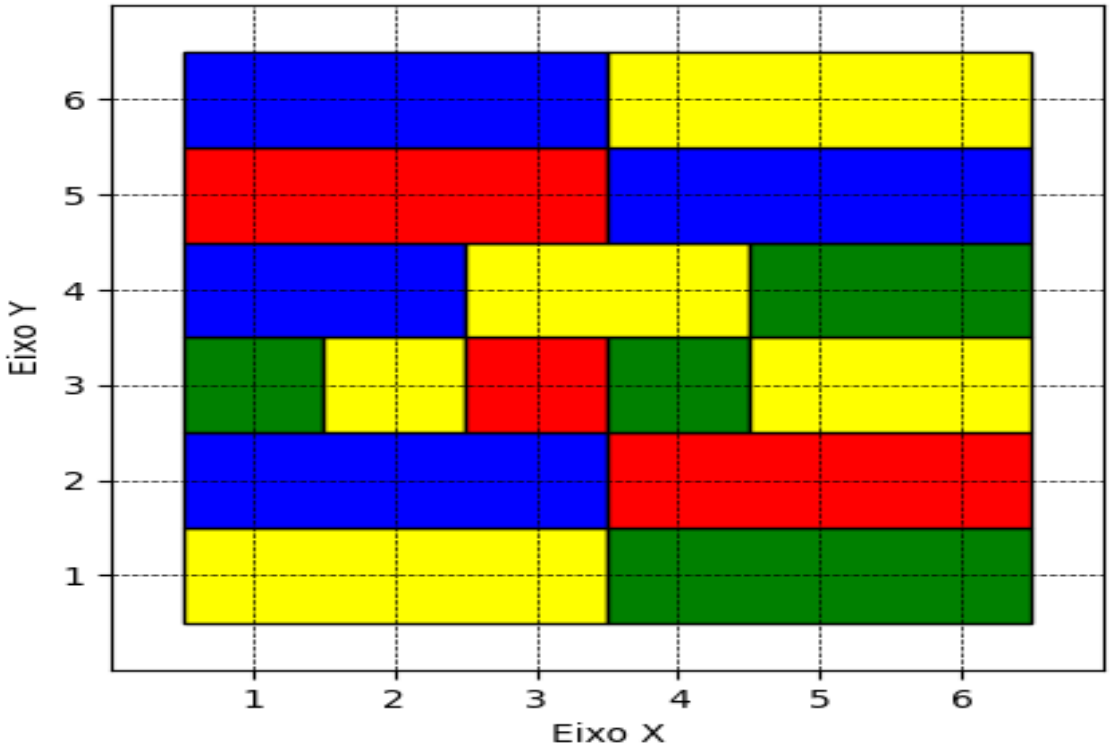

Figura 3: Configuração Válida

Com essa configuração obtivemos uma média de tempo de 0.000563 segundos usando gettimeofday da bliblioteca $< sys/time.h >$.

5 Conclusão

A análise de complexidade feita neste estudo ajudou a entender melhor o desempenho do código e como ele reage ao aumento no número de bombas e tipos de explosivos dos kits e suas respectivas configurações. Podemos observar que, no pior caso e no caso médio, o tempo de execução pode crescer muito, pois há muitas comparações e verificações a serem feitas, especialmente quando os dados aumentam.

A complexidade de $O(n^2 + n \times m)$ nos casos médio e pior mostra que o código pode ter problemas de desempenho quando trabalha com grandes conjuntos de dados. No melhor caso, com complexidade $O(n+m)$, o código é mais eficiente, especialmente se as correspondências são encontradas rapidamente, o que é ideal para conjuntos menores de dados ou condições mais favoráveis.

Em resumo, podemos destacar os pontos fortes do programa em gerenciar configurações e kits explosivos, mas também é possível identificar limitações em seu desempenho quando o número de itens aumenta. A complexidade $O(n^2 + n \times m)$ para casos médios e piores casos indica que o tempo de execução pode crescer significativamente, especialmente com verificações de adjacência e comparações de todos os itens

de configuração com seus kits. Como possível melhoria, o programa poderia otimizar as verificações de adjacência para reduzir a necessidade de loops aninhados. Revisar essas áreas ajudaria a melhorar o desempenho e a escalabilidade do código em futuras versões.

6 Bibliografia

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. *Introduction to Algorithms*. 3rd edition, MIT Press, 2009.