Universidade Federal de Ouro Preto - UFOP
Instituto de Ciências Exatas e Biológicas - ICEB
Departamento de Computação - DECOM
Ciência da Computação

Nonogramas

BCC202 - Estrutura de dados I

Thayllon Ryan Bragança
Thiago Martins Zanete

Professor: Pedro Henrique Lopes Silva

Ouro Preto
21 de janeiro de 2025

Sumário

1 Introdução

Para este trabalho é necessário entregar o código em C e um relatório referente ao que foi desenvolvido. O algoritmo a ser desenvolvido deve ser capaz de resolver um nonograma, utilizando-se das dicas fornecidas nos inputs.

A codificação deve ser feita em C, usando somente a biblioteca padrão da GNU, sem o uso de bibliotecas adicionais. Além disso, deve-se usar um dos padrões: ANSI C 89 ou ANSI C 99.

1.1 Especificações do problema

Este trabalho prático aborda a resolução de um problema relacionado a nonogramas, um tipo de quebra-cabeça de tabuleiro. O objetivo do jogador é preencher as células do tabuleiro de acordo com as dicas fornecidas para cada linha e coluna.

Para solucionar o problema de forma eficiente, foi necessário desenvolver um algoritmo que utiliza conceitos fundamentais de programação. O tabuleiro foi representado como uma matriz alocada dinamicamente, enquanto as dicas foram estruturadas em dois vetores distintos, um para as colunas e outro para as linhas.

A abordagem adotada incluiu a definição de Tipos Abstratos de Dados (TADs) para organizar as dicas e o tabuleiro. Funções específicas foram criadas para alocar, preencher e imprimir essas estruturas. A lógica principal do algoritmo foi implementada em várias funções que juntas são responsáveis pela resolução automática do quebra-cabeça.

1.2 Considerações iniciais

Algumas ferramentas foram utilizadas durante a criação deste projeto:

- Ambientes de desenvolvimento do código fonte: Replit, VSCode e CLion.
- Linguagem utilizada: C.
- Ambiente de desenvolvimento da documentação: Overleaf (LaTeX).

1.3 Ferramentas utilizadas

Algumas ferramentas foram utilizadas para testar a implementação, como:

- GNU Compiler Collection (GCC): Compilador C.
- Valgrind: Ferramenta de análise da memória dinâmica do código.
- Git: Repositório local.
- GitHub: Repositório remoto.
- GitHub: `corrector.py` (ferramenta disponibilizada pelo professor).

1.4 Especificações da máquina

A máquina onde o desenvolvimento e os testes foram realizados possui a seguinte configuração:

- Processador: Intel Core i5 8350u .
- Memória RAM: 16 Gb.
- Sistema Operacional: Linux (UBuntu).

1.5 Instruções de compilação e execução

Para a compilação do projeto, basta digitar:

Compilando o projeto

```
gcc main.c nonogram.c -o exe -std=c99 -Wall -g
```

Usou-se para a compilação as seguintes opções:

- *-std=99*: para usar-se o padrão ANSI C 99, conforme exigido.
- *-g*: para compilar com informação de depuração e ser usado pelo Valgrind.
- *-Wall*: para mostrar todos os possível *warnings* do código.

Para a execução do programa basta digitar:

```
./exe caminho_até_o_arquivo_de_entrada opcao
```

Onde "opcao" pode ser: "p" para fazer a, "d" para fazer b e "n" para fazer c.

2 Implementação

Para a implementação do algoritmo que resolve um nonograma utilizamos do pensamento computacional para separar o programa em partes, facilitando assim a sua compreensão e resolução.

2.1 Modularização

Visando melhorar a legibilidade e manutenibilidade do código, decidimos separá-lo em três módulos:

- `nonogramm.c`: Onde a lógica do programa foi implementada, ou seja, a ação de cada função está neste módulo.
- `nonogramm.h`: Apenas os cabeçalhos das funções são declarados aqui.
- `main.c`: A parte onde as funções são chamadas, podendo assim executar o programa e verificar sua funcionalidade.

2.2 Funções complementares

A fim de garantir o funcionamento do algoritmo de resolução de nonogramas, várias funções foram implementadas. São essas as funções complementares, pois não atuam diretamente na lógica de resolução do quebra-cabeças.

Funções para alocar e desalocar a memória utilizada durante a execução do programa foram devidamente implementadas a fim de otimizar espaço e eficiência do algoritmo.

Com a capacidade de ler as dicas, preencher o tabuleiro vazio e imprimir o jogo, algumas outras funções foram feitas, garantindo assim a funcionalidade correta e uma melhor experiência para o usuário.

Além disso, durante o desenvolvimento e depuração do código alguns outros métodos foram desenvolvidos principalmente para ajudar na correção de possíveis erros gerados. No entanto, essa parte não está contida na versão final, pois assim é possível entregar uma resolução mais organizada e legível.

2.3 Funções principais

Para implementar a parte responsável por de fato resolver o problema, foram implementadas algumas funções, estando essas condicionadas a uma função principal onde a recursão estava realmente aplicada.

Para checar em cada linha e coluna se as dicas estavam sendo respeitadas, duas funções foram desenvolvidas, ambas verificavam a cada nova inserção se o preenchimento estava sendo respeitado, ou seja, se ultrapassou ou não alcançou o limite estipulado pela TAD de dicas. Com base nisso, outra função mostrou-se ser necessária. Com o objetivo de checar se a linha já está completa, podendo assim com base no resultado, avançar a resolução. Dessa forma, checar a cada preenchimento trabalhava lado a lado com a verificação de uma linha ou coluna, proporcionando assim, maior organização do código e melhor eficiência ao longo da resolução do nonograma.

Para cada nova inserção na matriz, um novo procedimento foi criado, tendo como principal objetivo unir as verificações anteriores. Desse modo, antes de alterar qualquer coisa no tabuleiro essa função era chamada, que por sua vez chamava outras duas verificando célula por célula.

A função responsável pela recursão e backtracking, intitulada "NonogrammPlay", tinha como objetivo percorrer e preencher todo o quebra-cabeças, utilizando para isso, as verificações anteriores antes de avançar. Em caso de inconsistência, o algoritmo utiliza-se do backtracking retornando preenchimentos até o último estado válido, e então sempre tentando continuar de uma forma diferente da anterior. Em caso satisfatório, quando um nononograma chegasse ao final da resolução passando por todas estas etapas, o jogo é impresso, e o programa continua a explorar outras soluções para o mesmo tabuleiro.

Além disso, em caso do algoritmo testar de todas as formas possíveis e não conseguir progredir, pode-se considerar que o nonograma não tem solução, o que é mostrado ao usuário por uma pequena função, que também em casos positivos mostra o total de resultados possíveis para determinado jogo.

2.3.1 NonogrammPlay

Inicialmente essa função foi desenvolvida usando a lógica de preencher cada linha de uma coluna, dessa forma, quando essa função era chamada, começava a preencher da linha inicial e coluna inicial, e preenchia sistematicamente, considerando as restrições, linha por linha dá uma coluna, quando chegava ao limite daquela coluna, o algoritmo vai para a próxima, mas na linha inicial novamente.

Essa solução apresentou um resultado satisfatório para todos os testes, exceto o 3.in, que invertia a ordem da solução número 4 com a número 5. Mais tarde, mudou-se o preenchimento das células, usando a lógica de preenchimento de todas as linhas, ao contrário do que era feito antes.

O código final preenchia a linha/coluna atual, se era válido, chamava o NonogrammPlay adicionando uma unidade ao índice da coluna, para tentar preencher o novo item da linha. Caso o preenchimento naquela linha não seja válido, vai desfazer a ação, indo para a próxima célula repetindo até chegar no limite da linha, onde preenchemos todas as linhas e colunas possíveis, printando aquela solução.

Após isso, o algoritmo para a primeira inserção, desfazendo, e tentando preencher a próxima linha. Dessa forma, o backtracking garante a exploração de todo o espaço de solução.

2.3.2 CheckLine/CheckCol

Essas funções são responsáveis por checar toda a linha do início até a posição que foi inserida, a fim de verificar se aquela inserção está válida em comparação com as dicas fornecidas.

O algoritmo trabalha da seguinte forma: temos uma iteração iniciando da posição 0 até a posição atual da coluna, ou seja, matriz[lin][i], sendo "lin" a posição linha da matriz e "i" cada coluna naquela linha.

Para cada iteração, é testado se naquela posição tem uma célula preenchida, se sim, adicionamos ao contador, se na próxima iteração, o próximo bloco não estiver preenchido, e se o contador for maior que zero, nesse ponto, se a contagem for diferente da quantidade de dica quer dizer que o bloco tá incompleto ou errado, se não, vamos para a próxima dica naquela linha e redefinimos o contador para 0, mas caso não exista próxima dica, quer dizer que ultrapassamos o limite de dicas permitido.

Para cada vez que entrar em contador ¿ 0, quer dizer que estamos em um bloco não preenchido, ou seja, indo de uma dica, para a próxima.

O mesmo ocorre na função CheckCol, mas com algumas alterações, por exemplo, é checado do início da coluna até a o ponto que foi inserido e também é trabalho com a matriz no formato matriz[i][col], sendo "col", a posição coluna da matriz e "i", cada linha naquela coluna.

2.3.3 CheckIsCompletedLine

Como o algoritmo preenche o tabuleiro linha por linha, e durante essa ação era testado somente se aquela coluna/linha podia ser inserida ou não, mas em alguns casos, quando a próxima linha seria preenchida, a anterior ainda estava incompleta.

Dessa forma, foi criado um método que contava a quantidade total permitida pelas dicas naquela linha e quantos blocos estão preenchidos naquela linha, assim, quando o algoritmo for preencher a atual, ele testa se a anterior está completa, caso não esteja, vai retornar até a última pré-solução válida.

2.4 Considerações

No decorrer do desenvolvimento do programa, algumas funções mostraram-se desafiadoras no quesito implementação, principalmente as checagens de linhas colunas. Chegar na lógica de resolução de um nonograma revelou ser uma tarefa um tanto complexa, por serem muitas etapas e checagens que devem ser executadas de forma sequencial e estarem completamente alinhadas com o objetivo final. Devido a isso, e com base em muita análise do problema, várias formas de resolução foram propostas, até chegar aos resultados apresentados em **2.1** e **2.2**, onde foi possível equilibrar robustez e entendimento do código.

3 Testes

Os testes essenciais durante o desenvolvimento do código, pois assim foi possível verificar a cada novo trecho escrito se o caminho seguido continuava promissor, ou seja, se manter tal abordagem levaria a resolução completa do problema.

3.1 Testes iniciais

A cada nova função implementada, sua eficácia era testada tanto individualmente, quanto em conjunto com outras, focando principalmente nas funções complementares, que foram mencionadas no tópico **2.2**.

Nessa etapa foram submetidas a testes,principalmente funções de alocar e desalocar. Utilizando o Valgrind, a fim de detectar vazamentos de memória, foi possível detectar pequenos erros que impediam a liberação completa de memória, erros esses que foram devidamente corrigidos assim que eram encontrados.

Além disso, funções responsáveis por ler as dicas fornecidas, imprimir o jogo na tela, e iniciar o tabuleiro em branco foram postas à prova, com testes manuais no próprio terminal, usando como base os arquivos de entradas fornecidos pelo professor, foi possível garantir a eficácia destas. Desse modo, estas funções foram, de certa forma, aprovadas para permanecerem sem grandes alterações até o final do código, estabelecendo assim, com o auxílio do git e GitHub uma espécie de checkpoint no código.

3.2 Testes de lógica geral

Com o objetivo de testar principalmente as funções que de fato resolviam o nonograma, como mencionado em 2.3, uma série de testes foram feitos. A princípio, antes de serem desenvolvidas, estas funções passavam por pequenos questionamentos resolvidos entre os membros do grupo, para assim começarem a ser implementadas. Como mencionado em 2, primeiramente o algoritmo deveria ser capaz de resolver um exemplo de nonograma o mais simples possível, ou seja, onde todas as dicas são 1, pois assim o resultado seria uma diagonal completa. Essa etapa foi criada rapidamente, passando no teste sem muitas dificuldades. Para completar o programa foi preciso implementar as funções principais **(2.3)**, que de certa forma desafiaram as capacidades cognitivas dos membros. Ao serem implementadas eram imediatamente submetidas aos casos de testes propostos pelo professor, que em grande parte das vezes falhavam, provavelmente por lógica incorreta ou incompleta. Durante o desenvolvimento, essas funções apresentavam problemas que eram corrigidos, e criavam outros. Com base em muito estudo e análise, foi possível corrigir erros a cada teste, até finalmente serem capazes de passar em todos os casos, estando aptas a resolver corretamente o nonograma.

3.3 Testes finais

Nesta etapa, apenas erros de formatação foram testados e corrigidos, para se adequar ao formato de saídas proposto.

Mensagens ao usuário em caso de não encontrar uma solução para o jogo, quantidade de soluções encontradas e formatação de quebra de linha na saída do programa foram cuidadosamente analisadas e corrigidas, a fim de se adequarem ao que foi pedido no PDF do trabalho e nos resultados dos casos de testes.

Além disso, alguns comentários além dos existentes foram adicionados no código, facilitando assim a compreensão e manutenção futura deste.

4 Análise

Essa etapa é importante para avaliar o código, sua legibilidade, manutenibilidade e complexidade, pois assim, pode-se determinar possíveis alterações que melhorem a eficiência e velocidade do código.

4.1 Análise geral

Após a finalização do projeto, uma análise geral do que foi escrito foi necessária. A fim de avaliar o entendimento do código, uma busca por possíveis simplificações foi feita, onde pequenas, mas importantes, alterações foram realizadas, como, mudar o nome de algumas variáveis, deixando seu uso mais explícito e exclusão de trechos de comentários desnecessários futuramente. Ademais, uma rápida conferida na indentação geral mostrou-se bastante útil, garantindo assim, maior organização do projeto.

Uma verificação da memória e do tempo gastos, para a execução de cada teste foi realizada, a fim de avaliar a eficiência do programa, conforme especificado na tabela abaixo.

Teste	Tempo (s)	Memória (bytes)
1	0,004	5.649
2	0,004	5.417
3	0,004	5.649
4	0,004	5.417
5	0,004	6.444
6	0,004	5.417

4.2 Análise assintótica

Conforme foi pedido, a análise de tempo gasto para resolver o problema foi avaliada para cada função principal **(2.3)**, conforme listado abaixo:

Função CheckLine

```
1     int checkLine(Nonograma *nonograma, int lin, int col, int position) {
2     int count = 0;
3
4     for (int i = 0; i <= col; i++) {
5         if (nonograma->mat[lin][i] == '*') {
6             count++;
7             if (count > nonograma->dicas[nonograma->n + lin].linhas[position])
                  {
8                 return 0; // Excedeu o permitido pela dica atual
9             }
10        } else if (count > 0) {
11            // Separamos um bloco, verificamos se o bloco corresponde a dica
12            if (count != nonograma->dicas[nonograma->n + lin].linhas[position
                  ]) {
13                return 0; // Bloco incompleto ou incorreto
```

```
            }
            position++;
            count = 0;

            if (position >= nonograma->dicas[nonograma->n + lin].lqnt) {
                return 0; // Excedeu o numero de dicas permitidas
            }
        }
    }

    return 1;
}
```

Função de complexidade:
* Col = n
* f(n) = n + 1
* Notação Assintótica:
* O(n + 1) = O(n)

Função CheckCol

```
    int checkCol(Nonograma *nonograma, int lin, int col, int position) {
    int count = 0;


    for (int i = 0; i <= lin; i++) {
        if (nonograma->mat[i][col] == '*') {
            count++;
            if (count > nonograma->dicas[col].colunas[position]) {
                return 0; // Excedeu o permitido pela dica atual
            }
        } else if (count > 0) {
            // Separamos um bloco, verificamos se o bloco corresponde a dica
            if (count != nonograma->dicas[col].colunas[position]) {
                return 0; // Bloco incompleto ou incorreto
            }
            position++;
            count = 0;

            if (position >= nonograma->dicas[col].cqnt) {
                return 0; // Excedeu o numero de dicas permitidas
            }
        }
    }

    return 1;
}
```

Função de complexidade:
* Lin = n
* f(n) = n + 1
* Notação Assintótica:
* O(n + 1) = O(n)

Função CheckInsertion

```
    int checkInsertion(Nonograma *nonograma, int lin, int col) {

    int condition = 1;
```

```
    condition = checkCol(nonograma, lin, col, 0);

    if(condition){
        condition = checkLine(nonograma, lin, col, 0);
    }

    if (nonograma->dicas[col].cqnt == 0 ||
        nonograma->dicas[nonograma->n + lin].lqnt == 0) {
        condition = 0;
    }

    return condition;
}
```

Função de complexidade:
* CheckCol = n, CheckLin = n
No pior caso, analisando os últimos blocos da linha/coluna
* Logo
* F(n + m) = 2n + 1
* Notação Assintótica:
* O(2n + 1) = O(n)

Função CheckIsCompletedLine

```
    int checkIsCompletedLine(Nonograma *nonograma, int lin, int col) {
    int totalInLine = 0;
    for (int i = 0; i < nonograma->dicas[nonograma->n + lin].lqnt; i++) {
        totalInLine += nonograma->dicas[nonograma->n + lin].linhas[i]; // 1
    }

    int count = 0;
    for (int i = 0; i <= col; i++) {
        if (nonograma->mat[lin][i] == '*') { // 1
            count++;
        }
    }

    return count == totalInLine;
}
```

* Função de Complexidade:
* Assim como na CheckInsertion, a função será dada da por:
* f(n + m) = n + m
* Já que no pior caso, Columns totais sempre sera maior que a quantidade totais de dicas.
* Notação Assintótica:
* O(n + m) = O(n)

Função NonogrammPlay

```
    void NonogrammPlay(Nonograma *nonograma, int lin, int col, int *solutions)
        {

    if (lin == nonograma->n) {
        *solutions+=1;
        printf("SOLUTION %d:\n", *solutions);
        NonogrammPrint(nonograma); // Imprime a solucao
        printf("\n"); // Adiciona uma linha em branco entre solucoes
        return; // Continua explorando outras solucoes
```

```
    }

    if (col == nonograma->n) {
        if (!checkIsCompletedLine(nonograma, lin, col-1)) {
            return; // Linha incompleta
        }
        NonogrammPlay(nonograma, lin+1, 0, solutions); // Proxima Coluna
        return;
    }

    nonograma->mat[lin][col] = '*';
    if (checkInsertion(nonograma, lin, col)) {
        NonogrammPlay(nonograma, lin, col+1, solutions); // Avanca para a
            proxima coluna
    }

    nonograma->mat[lin][col] = '.';
    NonogrammPlay(nonograma, lin, col+1, solutions);
}
```

Função de Complexidade:

Para cada chamada de `NonogrammPlay`, temos mais 2 dela mesma.

A equação seria:

$$T(n) = 2 \cdot T(n-1)$$
$$T(n) = T(0) = 1$$

$T(n-1)$ é a quantidade restante para resolver o restante das células.

Já no pior caso, de visitar todas as células, temos $n = n \cdot n$:

$$T(n \cdot n) = 2 \cdot T(n \cdot n - 1) + O(n)$$

Expandindo:

$$T(n \cdot n) = 2 \cdot T(n \cdot n - 1) + O(n)$$
$$T(n \cdot n - 1) = 2 \cdot (2 \cdot T(n \cdot n - 2)) + O(n)$$
$$T(n \cdot n - 2) = 2 \cdot (2 \cdot (2 \cdot T(n \cdot n - 3))) + O(n)$$
$$T(n \cdot n - 3) = 2 \cdot (2 \cdot (2 \cdot (2 \cdot T(n \cdot n - 4)))) + O(n)$$

$$\dots$$
$$T(n \cdot n - k) = 2^k \cdot T(0)$$

$$T(0) = 1$$

Prova:

$$n = 2$$

$$T(4) = 2^1 \cdot T(3) + O(n)$$
$$T(3) = 2^2 \cdot T(2) + O(n)$$
$$T(2) = 2^3 \cdot T(1) + O(n)$$
$$T(1) = 2^4 \cdot (2 \cdot T(0)) + O(n)$$

Elevamos o 2, k vezes, $k = 4$.

$$T(k+1) = 1 \Rightarrow T(0) = 1$$
$$k+1 = n^2 \Rightarrow k = n^2 - 1$$

Logo:

$$k = n^2 - 1$$
$$T(n) = 2^{(k-1)} + 1 + O(n)$$

Logo, $O(2^{n^2-1} + n + 1) = O(2^{n^2})$.

5 Conclusão

Após finalizar o código, é importante mencionar uma conclusão, onde será explorado as principais dificuldades durante a execução da tarefa, além de uma visão geral sobre o processo de aprendizado e desenvolvimento do projeto.

5.1 Conclusão final

O processo de criação de um algoritmo responsável por resolver um nonograma corretamente revelou-se um tanto complexo, tanto para pensar em uma possível forma de resolução, quanto para traduzir esse pensamento em uma linguagem de programação.

As funções de verificar cada célula, linha e coluna, desafiaram o pensamento, visto que, haviam muitas partes para percorrer e uma interdependência crucial entre cada alteração entre elas. Um número considerável de variáveis mostraram-se fundamentais, pois foi preciso contar células preenchidas, células vazias e blocos de células preenchidas, o que demandou um raciocínio diferente para cada situação. Somado a isso, a implementação do backtracking também foi desafiadora, pois, por explorar uma funcionalidade da recursão na qual nem todos os membros dominavam perfeitamente foi preciso aprimorar o conhecimento através de vídeos e textos, o que de certa forma ocasionou em uma maior aumento do tempo de desenvolvimento do projeto.

A proposta do trabalho prático revelou-se um pouco laboriosa, principalmente por estar relacionada intrinsecamente com recursividade, um tópico novo para osmembros. Entretanto, a escolha deste tema , foi bastante coerente com o que foi ensinado em sala de aula, obrigando, de certa forma, aos alunos buscarem se aperfeiçoarem nas técnicas ensinadas, favorecendo assim, um maior aprendizado e fixação do conteúdo teórico. Por outro lado, a análise assintótica também requereu uma certa atenção, por estar relacionada com uma função recursiva com várias chamadas, sua análise precisou de um maior domínio em relação a complexidade , o que foi buscado através de estudo e colaboração entre os membros.

Conclui-se finalmente que o trabalho prático foi importante para fixação dos conhecimentos, contribuindo para uma melhor compreensão na prática de como as coisas realmente funcionam. Ademais, durante o desenvolvimento foi proporcionado, em grande parte no Replit, um ambiente de trabalho em equipe, onde cada membro apresentava suas ideias e as desenvolvia, elevando de forma satisfatória as experiências dos integrantes em relação a isso.

6 Bibliografia

Referências utilizadas ao longo do desenvolvimento do trabalho.

- Slide das aulas de estruturas de dados I, disponibilizados pelo professor: Slides
- Stack Overflow: stackoverflow.com
- Nonograms: Nonograms.com
- Geek for Geeks na linguagem C: geekforgeeks
- Medium (N-Queens problems): mediumNquenns