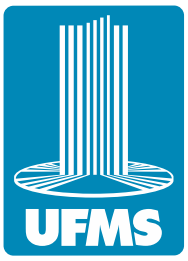

UNIVERSIDADE FEDERAL DE MATO GROSSO DO SUL
FACULDADE DE COMPUTAÇÃO

ALGORITMOS E PROGRAMAÇÃO II

Anderson Bessa da Costa

LUIS MIGUEL TABORDA FIALHO
MAYKON KAZUHIRO FALCÃO TAMANAHA

TRABALHO PRÁTICO SUDOKU

CAMPO GRANDE - MS
13/11/2024

FACOM - UFMS

LUIS MIGUEL TABORDA FIALHO
MAYKON KAZUHIRO FALCÃO TAMANAHA

TRABALHO PRÁTICO SUDOKU

Relatório de prática experimental apresentado como requisito parcial de avaliação da disciplina ALGORITMOS E PROGRAMAÇÃO II, sob orientação do Prof. Anderson Bessa da Costa.

CAMPO GRANDE - MS
2024

FACOM - UFMS

1 INTRODUÇÃO

No âmbito da disciplina Algoritmos e Programação II, os alunos foram desafiados a realizar um trabalho prático que demonstrasse a aplicação dos conceitos fundamentais de lógica de programação, além de sua capacidade de colaboração em grupo para o desenvolvimento de um projeto comum. Como tema, foi escolhido o popular jogo de Sudoku, um quebra-cabeça numérico que ganhou popularidade global a partir de 2005, especialmente no Brasil. Esse jogo envolve uma matriz 9x9 onde o jogador deve completar as células vazias, obedecendo a regras específicas que impedem a repetição de números na mesma linha, coluna ou região 3x3 (ASCENCIO; CAMPOS, 2012).

Além disso, foi solicitado que os alunos resolvessem o desafio de forma iterativa, em contraste com o método de backtracking, amplamente utilizado para solucionar Sudokus de qualquer nível de dificuldade. A escolha da abordagem iterativa teve como objetivo fortalecer a lógica de programação e a compreensão das estruturas de repetição, que são fundamentais para a resolução desse tipo de problema. Nesse contexto, foram trabalhados apenas Sudokus de nível fácil, o que permitiu aos alunos praticarem a lógica iterativa de forma mais acessível e didática (PIVA *et al.*, 2012).

Para facilitar o desenvolvimento, a estrutura básica do código foi fornecida aos alunos, incluindo a definição das principais funções. Contudo, apesar de as funções estarem organizadas, foram deixadas propositalmente sem implementação, dando aos estudantes a oportunidade de aplicar seus conhecimentos em lógica de programação na resolução e validação das jogadas do Sudoku, bem como no carregamento e salvamento de arquivos, entre outras operações programadas (DEITEL; DEITEL, 2011).

2 OBJETIVOS

Desenvolver um programa em C/C++ para resolver Sudoku em nível fácil de forma iterativa, utilizando uma base de código previamente estruturada. Este desafio buscou demonstrar a compreensão dos alunos sobre fundamentos de lógica de programação, aplicados em um quebra-cabeça lógico clássico e amplamente conhecido.

- Fuções ”TO DO”:

1. `carregue`;
2. `carregue_novo_jogo`;
3. `carregue_continue_jogo`;
4. `crie_arquivo_binario`;

5. `válido_na_coluna`;
6. `válido_na_linha`;
7. `válido_no_quadrante`;
8. `resolve_um_passo`;
9. `salve_jogada_em_binario`.

3 MATERIAIS E MÉTODOS

3.1 MATERIAIS

- Computador (Sistema operacional: Windows);
- Site compilador online de C++ - GDB online Debugger (Compilador: g++ versão 11.4.0, padrão -std=c++11).

3.2 MÉTODOS

Inicialmente, todo o código e as funções a serem implementadas foram analisados. As funções `determine_quadrante`, `fim_x`, `fim_y`, `ini_x` e `ini_y` não foram necessárias durante o programa e, por isso, foram removidas. Na função principal, `main`, foi configurada a função `srand(time(NULL))` para gerar uma semente aleatória logo no início do programa.

```
27  /* MAIN */
28  int main() {
29      srand(time(NULL)); // Cria uma semente aleatória para o rand.
30      jogue();
31
32      return 0;
33  }
```

Figura 1 – Função principal (main) com srand configurada

A função principal do jogo, `jogue`, possui inicialmente a função `carregue`, responsável por invocar duas outras funções: `carregue_novo_jogo`, para abrir um arquivo de texto com um novo Sudoku de nível fácil, e `carregue_continue_jogo`, para carregar um jogo salvo previamente em um arquivo binário.

```c
FILE* carregue(char quadro[9][9]) {
    char nome_arquivo[10];
    FILE* arquivo = NULL;
    int opcao;

    menu_arquivo();
    opcao = leia_opcao();

    switch (opcao) {

        // carregar novo sudoku
    case 1:
        printf("Nome do arquivo (Adicione '.txt' ao final): ");
        scanf("%s", nome_arquivo);

        carregue_novo_jogo(quadro, nome_arquivo);
        arquivo = crie_arquivo_binario(quadro);
        break;

        // continuar jogo
    case 2:
        printf("Nome do arquivo (Adicione '.bin' ao final): ");
        scanf("%s", nome_arquivo);

        arquivo = carregue_continue_jogo(quadro, nome_arquivo);
        break;

        // retornar ao menu anterior
    case 9:
        break;

    default:
        printf("Opção inválida!\n");
        break;
    }

    return arquivo;
}
```

Figura 2 – Função carregue

Na função `carregue_novo_jogo`, é solicitado o nome do arquivo de texto do Sudoku, que é aberto e lido posição por posição, após isso a função `crie_arquivo_binario` é chamada, retornando um ponteiro para o arquivo binário criado. A função `carregue_continue_jogo`, por outro lado, recebe o nome de um arquivo binário salvo, abre-o e lê a quantidade de jogadas realizadas, armazenada nas primeiras posições do arquivo. Em seguida, busca o último estado salvo do quadro do jogo, calculando a posição correta com base no tamanho do quadro e no número de jogadas, a partir do começo do arquivo, e finalmente retornando o ponteiro para o arquivo aberto.

```c
void carregue_novo_jogo(char quadro[9][9], char* nome_arquivo) {
    FILE* arquivo = fopen(nome_arquivo, "r");

    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo: %s.\n", nome_arquivo);
        return;
    }

    rewind(arquivo);

    // Ler o estado inicial do jogo
    for (int i = 0; i < 9; i++) {
        for (int j = 0; j < 9; j++) {
            fscanf(arquivo, " %c", &quadro[i][j]);
        }
    }

    printf("Novo jogo carregado com sucesso do arquivo: %s!\n", nome_arquivo);
    fclose(arquivo);
}
```

Figura 3 – Função carregue um novo jogo

```c
FILE* carregue_continue_jogo(char quadro[9][9], char* nome_arquivo) {
    FILE* arquivo = fopen(nome_arquivo, "rb+");
    int jogada;

    if (arquivo == NULL) {
        printf("Erro ao abrir o arquivo: %s.\n", nome_arquivo);
        return NULL;
    }

    rewind(arquivo);
    fread(&jogada, sizeof(int), 1, arquivo);

    fseek(arquivo, sizeof(int) + (sizeof(char) * 81 * jogada), SEEK_SET);

    fread(quadro, sizeof(char), 81, arquivo);

    return arquivo;
}
```

Figura 4 – Função carregue e continue jogo

Após a execução da função `carregue_novo_jogo`, a função `crie_arquivo_binario` é chamada para criar um arquivo em binário, que possui um nome aleatório gerado com auxílio da função `gen_random`. Após ser criado, inicialmente armazena-se a quantidade de jogadas (0) e o estado inicial do quadro (81 elementos com valor zero).

```c
FILE* crie_arquivo_binario(char quadro[9][9]) {
    char nome_arquivo[10];
    FILE* arquivo;
    int jogada = 0;

    gen_random(nome_arquivo, 5);

    arquivo = fopen(nome_arquivo, "wb+");

    if (arquivo == NULL) {
        printf("Erro ao criar o arquivo: %s.\n", nome_arquivo);
        return NULL;
    }

    printf("\nJogada: %d.\n", jogada);

    // Escreve a matriz 9x9 diretamente no arquivo binário
    rewind(arquivo);
    fwrite(&jogada, sizeof(int), 1, arquivo);
    fwrite(quadro, sizeof(char), 81, arquivo);

    return arquivo;
}
```

Figura 5 – Função crie um arquivo binário

A função `salve_jogada_em_binario` foi implementada para salvar o progresso do jogo. Ela recebe o ponteiro do arquivo e o quadro do jogo, lê a quantidade atual de jogadas no arquivo, incrementa em uma unidade e regrava no arquivo. Em seguida, acessa o final do arquivo para salvar o estado atualizado do quadro.

```c
void salve_jogada_em_binario(FILE* fb, const char quadro[9][9]) {
    int jogada;

    if (fb == NULL) {
        printf("O arquivo binário não está aberto!\n");
        return;
    }

    rewind(fb);
    fread(&jogada, sizeof(int), 1, fb);

    jogada++;

    printf("\nJogada: %d.\n", jogada);

    rewind(fb);
    fwrite(&jogada, sizeof(int), 1, fb);

    fseek(fb, 0, SEEK_END);
    fwrite(quadro, sizeof(char), 81, fb);

    printf("Estado do jogo salvo com sucesso!\n");
}
```

Figura 6 – Função salve jogada em binário

A função `valido` foi implementada com três funções de verificação (`valido_na_coluna`,

`valido_na_linha` e `valido_no_quadrante`) para garantir a integridade dos valores inseridos no Sudoku.

```
138 int valido(const char quadro[9][9], int x, int y, int valor) {
139     // verifica as três condições:
140     if (!valido_na_coluna(quadro, y, valor))
141         return false;
142     if (!valido_na_linha(quadro, x, valor))
143         return false;
144     if (!valido_no_quadrante(quadro, x, y, valor))
145         return false;
146 
147     return true;
148 }
```

Figura 7 – Funções de validação

A função `valido_na_coluna` verifica se o valor está presente na coluna. Caso positivo, retorna `false`; caso contrário, `true`. A função `valido_na_linha` segue a mesma lógica, mas verifica a linha.

```
150 int valido_na_coluna(const char quadro[9][9], int y, int valor) {
151     int i;
152 
153     for (i = 0; i < 9; i++) {
154         if (quadro[i][y] == (valor + '0')) return false;
155     }
156 
157     return true;
158 }
159 
160 int valido_na_linha(const char quadro[9][9], int x, int valor) {
161     int j;
162 
163     for (j = 0; j < 9; j++) {
164         if (quadro[x][j] == (valor + '0')) return false;
165     }
166 
167     return true;
168 }
```

Figura 8 – Funções de validação

A função `valido_no_quadrante` recebe a linha (x), coluna (y) e o valor a ser posicionado; calcula o quadrante com as expressões "$(x/3)*3$"e "$(y/3)*3$", e usa dois laços de repetição para verificar todos os números do quadrante. Retornando false caso o número já esteja presente, e retornando true no caso contrário.

```c
int valido_no_quadrante(const char quadro[9][9], int x, int y, int valor) {
    int i, j, ini_x, ini_y;

    ini_x = (x / 3) * 3;
    ini_y = (y / 3) * 3;

    for (i = ini_x; i < ini_x + 3; i++) {
        for (j = ini_y; j < ini_y + 3; j++) {
            if (quadro[i][j] == (valor + '0')) {
                return false;
            }
        }
    }

    return true;
}
```

Figura 9 – Funções de validação

Por fim, a função `resolve_um_passo` foi concluída. Ela utiliza dois laços de repetição para percorrer todo o quadro em busca de posições vazias. Quando encontra uma posição vazia, testa os valores de 1 a 9 e, se houver apenas uma opção válida, insere o valor nessa posição. Caso contrário, se todas as posições vazias tiverem duas ou mais opções válidas, a função passa a analisar cada quadrante individualmente, verificando quais valores de 1 a 9 estão faltando. Se em um quadrante específico houver apenas um valor ausente, ele é inserido na posição vazia. Essa abordagem é viável para resolver Sudokus de nível fácil.

```c
void resolve_um_passo(char quadro[9][9]) {
    int i, j, z, cont, ultima_resposta, quebra = 0;
    int quadrante_i, quadrante_j;
    int presente[9] = { 0 };
    int linha_vazia = -1, coluna_vazia = -1;
    int num_faltante;
    char valor;

    // Primeiro, percorre todas as posições
    for (i = 0; i < 9; i++) {
        for (j = 0; j < 9; j++) {
            if (quadro[i][j] == '0') {
                cont = 0; // Contador de quantas respostas válidas

                for (z = 1; z < 10; z++) {
                    if (valido(quadro, i, j, z)) {
                        ultima_resposta = z;
                        cont++;
                    }

                    if (cont > 1) break;
                }

                // Se houver exatamente uma resposta válida
                if (cont == 1) {
                    quadro[i][j] = ultima_resposta + '0';
                    printf("Um passo resolvido!\n");
                    return;
                }
            }
        }
    }
```

Figura 10 – Funções de resolver um passo - primeira parte

```
366     // Se nenhum for resolvido
367     for (quadrante_i = 0; quadrante_i < 9; quadrante_i += 3) {
368         for (quadrante_j = 0; quadrante_j < 9; quadrante_j += 3) {
369 
370             for (z = 0; z < 9; z++) {
371                 presente[z] = 0;
372             }
373 
374             linha_vazia = -1;
375             coluna_vazia = -1;
376             cont = 0;
377             quebra = 0;
378 
379             // Verificar o quadrante 3x3
380             for (i = quadrante_i; i < quadrante_i + 3 && !quebra; i++) {
381                 for (j = quadrante_j; j < quadrante_j + 3 && !quebra; j++) {
382                     valor = quadro[i][j];
383                     if (valor != '0') {
384                         presente[valor - '1'] = 1;  // Marcar o número como presente
385                     } else {
386                         linha_vazia = i;
387                         coluna_vazia = j;
388                         cont++;
389 
390                         if (cont >= 2) quebra = 1;
391                     }
392                 }
393             }
394 
395             // Se houver exatamente uma célula vazia, preencha-a com o número faltante
396             if (cont == 1) {
397                 for (num_faltante = 0; num_faltante < 9; num_faltante++) {
398                     if (!presente[num_faltante]) {
399                         quadro[linha_vazia][coluna_vazia] = (num_faltante + 1) + '0';
400                         printf("Um passo resolvido pelo quadrante!\n");
401                         return;
402                     }
403                 }
404             }
405         }
406     }
407 
408     printf("Nenhum passo resolvido.\n");
409 }
```

Figura 11 – Funções de resolver um passo - segunda parte

4 RESULTADOS E DISCUSSÕES

A implementação do programa de Sudoku enfrentou desafios, especialmente nas funções `salve_jogada_em_binario`, `carregue_continue_jogo` e `resolve_um_passo`. Inicialmente, houve dificuldade em salvar o estado do jogo em binário conforme o PDF do projeto, mas, após ajustes no ponteiro do arquivo para configurar os quatro primeiros bytes (número de jogadas) e salvar o tabuleiro, a função de salvamento foi corrigida. Isso também permitiu resolver os problemas na função de carregamento.

A função `resolve_um_passo` foi a mais complexa. Além de localizar e preencher células vazias usando um laço iterativo para verificar valores de 1 a 9, houve um desafio ao identificar casos em que mais de um valor era válido para a mesma célula. A solução

envolveu um contador que apenas permitia preencher células com uma única opção válida. Em casos sem progresso, um segundo laço verificava quadrante a quadrante, completando células com apenas uma posição viável para o número faltante.

Os resultados indicam que o programa conseguiu gerenciar o jogo em nível fácil, controlando partidas, salvando e carregando progresso. As funções de verificação (`valido_na_coluna`, `valido_na_linha` e `valido_no_quadrante`) foram eficazes para evitar duplicações em linhas, colunas e quadrantes, respeitando as regras do Sudoku e sem comprometer o desempenho.

Contudo, a abordagem iterativa de `resolve_um_passo` não é suficiente para níveis médio ou difícil, pois nestes casos é comum haver múltiplas opções em cada célula vazia. Para esses níveis, seria necessária uma técnica de tentativa e erro (como *backtracking*), capaz de explorar e ”corrigir”escolhas para encontrar uma solução.

Em suma, a implementação atingiu os objetivos propostos, e o programa demonstrou consistência em validação e resolução para níveis fáceis, respondendo adequadamente aos cenários testados.

5 CONCLUSÕES

A implementação do programa de Sudoku cumpriu seus objetivos ao demonstrar a aplicação prática de conceitos fundamentais de lógica de programação e manipulação de arquivos em C++. Através de uma estrutura organizada e das funções de verificação, o projeto garantiu a precisão nas jogadas e o cumprimento das regras do jogo, atendendo ao nível fácil de Sudoku de forma satisfatória. Essa experiência permitiu consolidar o aprendizado em programação estruturada e destacou potenciais melhorias para suportar níveis mais complexos, indicando caminhos para o aprofundamento no desenvolvimento de algoritmos eficientes em resoluções de problemas lógicos.

REFERÊNCIAS

ASCENCIO, A. F. G.; CAMPOS, E. A. V. **Fundamentos da programação de computadores: algoritmos, Pascal, C/C++ e Java**. 3rd. ed. São Paulo: Pearson, 2012.

DEITEL, P.; DEITEL, H. **Como Programar**. 6th. ed. São Paulo: Pearson, 2011.

PIVA, D. *et al.* **Algoritmos e programação de computadores**. Rio de Janeiro: Elsevier, 2012.