Documentação TP3

Matheus Ibrahim Fonseca Magalhães
Matrícula: 2018046912

1. Introdução

O problema apresentado requisita um algoritmo capaz de resolver um Sudoku, recebendo uma matriz de tamanho NxN(N ∈ {4,6,8,9}), e a quantidade de linhas e colunas em cada quadrante da matriz. O Sudoku consiste de um jogo com uma matriz NxN com N quadrantes semipreenchida, onde o jogador deve completar a matriz com números de 1 a N, sendo que cada linha, coluna e quadrante possua todos os números do intervalo sem repetição. Para isso, o algoritmo foi desenhado baseando-se em coloração de grafos.

2. Código

2.1) Linguagem e ambiente

O programa foi escrito na linguagem de programação C++ em uma máquina virtual utilizando o sistema operacional Linux e compilado por um arquivo *makefile*(incluído no envio).

2.2) Arquivos

O programa possui três arquivos: *main.cpp, Sudoku.hpp* e *Sudoku.cpp*. O arquivo *main* é responsável pela leitura dos dados do arquivo de entrada e realiza chamadas para funções da classe *Sudoku*, que possui todas as funções e estruturas necessárias para resolução do problema. O arquivo *Sudoku.hpp* estabelece uma *struct* chamada *Position* e define o contrato da classe *Sudoku*. A classe é, então, implementada no arquivo *Sudoku.cpp*.

2.3) Detalhamento das estruturas e classes

A estrutura Position é utilizada para definir cada célula da tabela Sudoku, possuindo os atributos posição(cada célula é enumerada como no exemplo abaixo), valor(o valor de 1 a N que a célula recebe) e um booleano colored(se a célula já possui um valor atribuído ou ainda deve receber).

A classe Sudoku define uma matriz e um grafo(implementado via lista de adjacência) da estrutura Position, que representam a tabela do Sudoku. Cada vértice do grafo faz uma referência a uma posição da matriz, de forma a simplificar as operações, podendo alterar o valor em uma estrutura com a alteração se propagando para a outra. A classe também possui todas as funções necessárias para resolver o problema:

- Sudoku(int square, int line, int column, int** table): o construtor da classe, cria a matriz do sudoku com base numa matriz parâmetro e chama a função responsável pela construção do grafo.
- BuildGraph(int square, int line, int column): considerando cada posição da matriz como um vértice, define quais pares de célula possuem adjacência entre si(não podem ter o mesmo valor)
- PrintGraph(): função que imprime a lista de adjacência de cada vértice do grafo.
- FindSolution(): Encontra qual o vértice do grafo está mais saturado(menos valores disponíveis) e atribui um valor que não cause conflito a ele até a matriz ser preenchida ou ocorrer alguma falha(vértice sem valor disponível).
- NextNodeToColor(bool*): Função auxiliar que encontra o vértice com maior saturação do grafo.
- AssignValue(bool*, int): Função auxiliar que atribui valor a um determinado vértice.
- PrintSolution(): Imprime se uma solução foi encontrada e imprime ela na forma de matriz.
- checkSolved(): Verifica se uma solução válida foi encontrada.
- addEdge(): Cria uma aresta no grafo
- existsEdge(): Verifica se uma aresta já existe no grafo.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Tabela 1. Exemplo de enumeração das células e dividindo os quadrantes numa tabela 4x4.

2.4) Detalhamento das principais funções do programa

Os dois processos mais importantes do programa são a criação do grafo, devendo determinar quais vértices são adjacentes, e a busca pela solução, que, no caso esperado, encontra uma tabela completa. Portanto, é importante compreender a lógica utilizada nas funções *BuildGraph* e *FindSolution*, e suas auxiliares *NextNodeToColor* e *AssignValue*.

BuildGraph: A função percorre a matriz criada e adiciona os vértices que devem existir(mesma linha, mesma coluna, mesmo quadrante). A mesma linha e a mesma coluna são simples e podem ser feitas em três laços for, onde os dois primeiros percorrem a tabela em cada posição [i,j] e o terceiro adiciona adjacências com as posições [i,k] e [j,k]. Para verificar se dois vértices estão no mesmo quadrante, é utilizado um quarto laço for, e a verificação pode ser feita pela seguinte lógica:

Dadas duas células da tabela com n linhas e m colunas nas posições $[i,j]$ *e* $[k,l]$:
$Se\ i/n = k/n\ e\ j/m = l/m$, *as duas células estão no mesmo quadrante*

Uma observação importante é que as divisões devem ser truncadas (não verificar parte decimal).

NextNodeToColor: É uma função auxiliar para encontrar a solução, e encontra o vértice mais saturado do grafo. Para isso, é utilizado um vetor de booleanos *available*, que possui os valores de 1 a N como *true*, se esse valor pode ser atribuído ao vértice, e *false* caso contrário. Depois disso, a função visita cada vértice, e, caso ele não possua valor, percorre sua lista de adjacência verificando quais dos seus vizinhos já possuem e quantos valores diferentes eles possuem. Ao final, a função retorna o índice do vértice com menor quantidade de valores disponíveis.

AssignValue: Essa função é chamada quando já se sabe qual é o próximo vértice a receber um valor, e, mais uma vez utilizando um vetor de booleanos, é encontrado o primeiro valor disponível para o vértice e tal valor é atribuído a ele.

FindSolution: Apoia-se nas duas funções auxiliares explicadas para encontrar a solução final. Realiza no máximo N^2 iterações e atribui um valor a um vértice em cada iteração(portanto utiliza o máximo de iterações se receber uma tabela vazia). Caso o mesmo vértice seja selecionado duas vezes seguidas, a função é finalizada porque ou a solução foi encontrada ou ocorreu um conflito e não será encontrada solução.

2.5) Fluxo do programa

O programa começa recebendo um parâmetro, o nome do arquivo que possui as entradas para a execução do programa. Esse arquivo é armazenado em uma variável do tipo *ifstream*, e, a partir dele, as entradas dos programas são lidas e armazenadas em variáveis. Após a leitura das entradas é criado o Sudoku e o grafo, e, em seguida, é chamada a função FindSolution. Por fim, é impressa a mensagem “Solução” ou ”Sem solução”, dependendo de uma solução ter ou não sido encontrada, e a tabela final em forma de matriz.

2.6) Decisões de implementação

a) **Linguagem C++**

A linguagem C++ permite, ao contrário de C, uso de técnicas de Programação Orientada a Objetos, além de possuir uma construção intuitiva para leitura de entradas via arquivos, especialmente com a biblioteca *fstream* e o tipo *ifstream*. Por isso, a linguagem C++ oferece mais suporte a forma como a implementação foi montada.

b) Matriz e grafo

Como o grafo possui o mesmo vértice em diversas listas de adjacência, a decisão foi implementar o grafo com cada vértice sendo uma referência a uma posição na matriz. Dessa forma, cada alteração pode ser feita diretamente na matriz e propagada para o grafo.

Com essa estratégia, o grafo é utilizada para pesquisas(como o grau de saturação dos vértices) e a matriz é utilizada para alterar valor/propriedades das células.

c) Instruções de compilação e execução

No envio já foi incluído um arquivo executável chamado **tp3**, portanto não é necessário compilar o programa, e os casos de teste foram enviados na pasta dataset3 e dentro. De qualquer forma, também está incluído no envio um arquivo *makefile* que, se executado com o comando *make*, irá gerar outro arquivo executável chamado **tp3**. Para executar o programa, basta utilizar, em um terminal do linux, o comando ./tp3 dataset3/CasoDeTeste.txt.

3. Análise de complexidade

Assumindo tabela NxN, grafo com V vértices($V = N^2$) e E arestas.

a) Tempo

Sudoku: Percorrer a tabela NxN e chamar a função construtora do grafo. Portanto, custo **$O(N^2)$ + BuildGraph()**

BuildGraph: A construção utiliza quatro laços for com N repetições em cascata, mais o custo O(E) de verificar se já existe uma aresta entre dois vértices ou de adicionar uma aresta entre dois vértices. Portanto, custo **$O(N^4 * E)$**

PrintGraph: Custo de percorrer uma lista de adjacências, portanto **O(V + E)**

NextNodeToColor: No pior caso(nenhum vértice colorido), percorre a lista de adjacências inteira, portanto **O(V + E)**

AssignValue: Simplesmente percorre um vetor com os valores de 1 a N, portanto **O(N)**

FindSolution: chama as funções NextNodeToColor e AssignValue até V vezes, portanto **O(V) + NextNodeToColor() + AssignValue()**

checkSolved: Percorre a tabela verificando a solução, portanto custo **$O(N^2)$**

PrintSolution: Percorrer a matriz, portanto **O(N^2)**

Custo total do programa: O(N^4 * E)

b) Espaço

A matriz e o grafo são atributos da classe, então o custo não foi considerado nas funções que não criam esses atributos.

Sudoku: Cria a tabela e chama o construtor do grafo, portanto **O(N^2) + BuildGraph()**

BuildGraph: A construção cria a lista de adjacências do grafo, portanto **O(V + E)**

PrintGraph: Simplesmente acessa uma lista de adjacência já criada e armazenada, portanto **O(1)**

NextNodeToColor: O parâmetro recebido *available*, que é um vetor de tamanho N, portanto **O(N)**

AssignValue: O parâmetro recebido *available*, que é um vetor de tamanho N, portanto **O(N)**

FindSolution: Cria o vetor de tamanho N, portanto **O(N) + NextNodeToColor() + AssignValue()**

checkSolved: Percorre uma tabela já armazenada na memória e não cria outras estruturas, portanto **O(1)**

PrintSolution: Percorre uma tabela já armazenada na memória e não cria outras estruturas, portanto **O(1)**

Custo total do programa: Matriz + Grafo, O(N^2) + O(V + E)

c) Conclusões sobre a análise

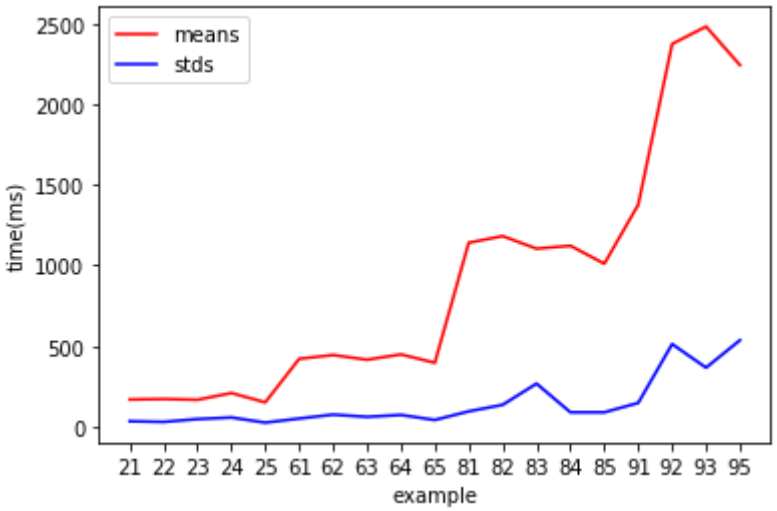
O tempo gasto para o programa certamente não é baixo, mas o custo é especialmente alto na construção do problema, e não na solução em si(apesar de a solução também possuir custo alto). Com relação ao espaço, o gasto foi razoável, pois a manutenção do grafo e da matriz como atributos permitem mantê-los na memória ao longo da execução do programa e não ter que criá-los diversas vezes para funções diferentes.

No geral, parte do custo teve de ser sacrificada para apresentar um desempenho satisfatório na maioria dos casos de teste.

4. Avaliação experimental

a) Análise do tempo

Para a análise de tempo, o programa foi executado, para cada caso de teste, 10 vezes. O gráfico a seguir demonstra os resultados encontrados.



O gráfico apresenta um resultado previsível, com 4 claros níveis na média dos tempos, para as tabelas de diferentes tamanhos, e o desvio padrão mantém-se razoavelmente constante. Os exemplos no eixo X seguem os mesmo nomes dos txts do dataset para ajudar a compreensão, mas não duplica o primeiro dígito(221.txt contra 21 no gráfico, por exemplo).

b) Saídas dos casos de teste

Os casos de teste foram sucedidos em um bom nível, encontrando soluções em 16 dos 19 casos de teste. Todas as tabelas 4x4 e 6x6 foram resolvidas, 3 de 5 das tabelas 8x8 foram resolvidos e 3 de 4 das tabelas 9x9 foram resolvidas(994.txt possui 9 linhas por 8 colunas, não é possível encontrar a solução).

5. Conclusão

A resolução de um problema NP-completo foi relativamente complexa, especialmente porque existem poucas referências na Internet para se basear na lógica do programa. Contudo, a aplicação de coloração de grafos em Sudoku é certamente condizente com o material visto em sala e uma ótima prática para a resolução de problemas dessa classe.

A maior dificuldade do problema era certamente descobrir o próximo vértice a colorir, pois a ideia da saturação não é simplista. Mesmo assim, tirando essa dificuldade, o geral do problema é relativamente simples.

6. Referências

C plus plus. Disponível em: <http://www.cplusplus.com/reference/>. Acesso em: 12 nov. 2019.

Sudoku as a Constraint Problem. SIMONIS, Helmut. Disponível em: <https://www.inf.tu-dresden.de/content/institutes/ki/cl/study/winter06/fcp/fcp/sudoku.pdf>. Acesso em: 14 nov. 2019.

Solving Every Sudoku Puzzle. Disponível em: <https://norvig.com/sudoku.html> Acessp em: 13 nov. 2019.