1. O Problema

Não vou me prolongar A base da lógica do Sudoku é preencher uma matriz de 9x9 “casas” com os números de 1 a 9, de forma que o número não se repita numa mesma linha ou coluna. Além disso, essa matriz está dividida em 9 pequenos quadrantes de 3x3 casas, e os números não podem também se repetir dentro destes quadrantes. A tábua do Sudoku em geral já vem com alguns números preenchidos de forma a limitar as possibilidades de solução. O problema em solucionar o sudoku consiste basicamente em tentar colocar os números nas casas vazias verificando as violações das regras a cada tentativa e, caso chegar em um “beco sem saída” voltar e tentar novas configurações. Esse comportamento sugere uma óbvia solução utilizando-se *backtracking* porém admite diversas heurísticas adicionais que sugerem melhores formas de tentar colocar os números nas casas de forma menos aleatória.

2. Implementação

1. Estruturas de Dados

Optei por utilizar alocamento dinâmico, acessando os dados como num vetor de 81 posições. Essa implementação se mostrou bem mais trabalhosa do que a intuitiva repesentação usando uma matriz de 9x9 posições, mas dei peferência ao fato de acessar a memória linearmente, mantendo a integridade na referência dos dados e evitando assim problemas de referenciamento. Essa estratégia se mostrou altamente eficiente para evitar os já conhecidos “segmentation fault”, já que acessando a memória através de um ponteiro único a possibilidade de erros é bem menor, e quando acontece, é bem mais fácil tratá-los.

Para utilizar esta representação porém era preciso transformar as referências clássicas da tabela 9x9 para o formato dos índices do vetor, que variavem de 0 a 80, essa transformação foi feita através de fórmulas específicas que tive que deduzir. A idéia era receber o número de 0 a 80 relativo à casa em que se estava colocando o número, e a partir daí obter todas as casas da sua linha, coluna e do seu quadrante, de forma a poder varrê-las em busca de repetições.

Linhas:

Para varrer as linhas virtuais da matriz 9x9, a transformação foi a seguinte:

$J_k = q \times L + i$, com:

J_k = casas J na mesma linha que k
L = número total de linhas do sudoku
q = K / L (apenas parte inteira)
i = um número variando de 0 a L-1

Colunas:

Para varrer as colunas virtuais:

$J_k = i \times L + r$, com:

J_k = casas J na mesma coluna que k
L = número total de linhas do sudoku
r = K % L (apenas parte inteira)
i = um número variando de 0 a L-1

Quadrantes:

Já para varrer os quadrantes 3x3 foi preciso uma transformação mais sofisticada:

$$J_k = L \times (i + p \times l) + (l * c) + j, \text{ com:}$$

```
Jk = casas J no mesmo quadrante que k
L = número total de linhas do sudoku
l = raiz quadrada de L
p = ( k / L ) / l (somente parte inteira)
c = ( k % L ) / l (somente parte inteira)
i e j = números variando de 0 a l
```

Esta fórmula reduz o número k extraindo suas linha e coluna através das funções anteriores, em seguida reduz novamente, obtendo assim em que quadrante o número está na horizontal e na vertical. Em seguida, com alguma manipulação algébrica, gera os seus vizinhos naquele quadrante. A fórmula foi exaustivamente testada e, muito embora não tenha criado uma prova formal, acredito que funcione para qualquer k, desde que k possua uma raiz quadrada inteira.

2. Estratégias

A melhor estratégia encontrada para a solução do problema foi baseada em *backtracking,* é sabido que uma estratégia gulosa aliada a esse *backtracking* diminuiria consideravelmente a quantidade de tentativas necessárias, e dessa forma o tempo de execução do programa, mas não foi implementada. O backtracking foi implementado através de uma recursão simples, o retorno de cada recursão é responsável pela decisão se o algoritmo continua ou volta atrás. Além disso, várias heurísticas são possíveis de ser implementádas de forma a reduzir o número de tentativas introduzindo alguma inteligência na forma como os números são testados.

Na implementação escolhida nenhuma heurística específica ao problema foi utilizada, o algoritmo resume-se à estratégia simples de *backtracking* sem maiores otimizações.

3. Entrada

A entrada é feita através do standard in e consiste simplesmente de 81 números inteiros entre 0 e 9, em sequência. Sendo que o 0 significa uma casa vazia e um número entre 1 e 9 representa uma casa que já vem preenchia com aquele número. Os números são interpretados como procedendo em linhas, ou seja, as posições da tabela são preenchidas de forma que o primeiro elemento seja A_{00} o segundo A_{01}, A_{02} e assim por diante. Ao se completar uma linha, passa-se a usar a de baixo. Essa é apenas uma representação virtual, visto que na realidade a estrutura de dados do algoritmo armazena todos os números num vetor único de 81 espaços. Por conta dessa forma de entrada, também é possível entrar com um arquivo, através do redirecionamento da entrada padrão. Esse arquivo deve conter 81 números inteiros separados por espaços ou quebras de linha. Exemplos de entrada encontram-se na área de testes.

4. Saída

A saída do programa consiste numa tábua de sudoku completa, sem divisões. A tábua consiste de nove linhas, cada uma contendo nove inteiros separados por espaços, de forma que o inteiro j da linha i representa o inteiro na posição ij da matriz. Exemplos de saídas encontram-se na área de testes.

5. Funções

1. main.c

- int main (void)

Esta é a função principal da aplicação. Não recebe parâmetro algum e, como padrão do C, retorna um inteiro. 0 para o caso de sucesso e 1 para o caso de erro em execução.

A função possui 3 constantes, ROWS, TABS e MINI, respectivamente representando o número se linhas/colunas da tábua de sudoku, a quantidade total de casas e o tamanho dos quadrantes internos. Alterando essas constantes podemos portar o programa para diversas variações do sudoku em tamanho. É possível modificar o programa para aceitar esses tamanhos como parâmetros, o que seria interessante mas foge das especificações dadas.

Nesta função é alocado dinamicamente o espaço de memória que irá abrigar a matriz e é instanciada uma variável contador, que apenas nos dirá quantas vezes a função teve que realizar um *backtracking.* Em seguida a função chama findNextZero para descobrir a primeira casa vazia do sudoku e chama a função solveRec nessa casa vazia. Caso o algoritmo retorne com sucesso, a matriz preenchida é impressa na tela, seguida do contador de tentativas. Caso constrário é impressa a mensagem “Solução Impossível” logo abaixo do número de tentativas feitas antes de se chegar a esse resultado.

2. logic.c

- int findNextZero(int *mtx, int atual)

Esta função recebe, além do ponteiro para a estrutura da matriz, um inteiro que diz em que ponto da matriz o algoritmo está trabalhando. Ela varre a matriz a partir desse ponto em busca da próxima casa nula, onde o algoritmo deverá ser aplicado. A função retorna o número indicativo da casa vazia em caso de sucesso, ou o flag arbitrário 999 em caso de não haver mais casas vazias na matriz.

- int colocaNum(int *mtx, int off, int cand)

Esta função recebe, além da estrutura base da matriz, um inteiro *off* que representa a casa atual do tabuleiro em que se está aplicando o algoritmo e um inteiro *cand* que representa o número a ser testado na quela casa.

Em seguida a função instancia variáveis e inicializa *q, e, ll e cc* com os números correspondentes aos índices de linhas e colunas a que *off* pertence, na tabela grande (9x9) e no quadrante pequeno (3x3). Isso é feito utilizando uma álgebra bem simples para converter a representação linear [0~80] na representação matricial [9x9] e [3x3]. O próximo passo é varrer a linha e a coluna daquela casa a procura de um número repetido, caso encontre, o método retorna erro. Em seguida, é necessário varrer o quadrante daquela casa usando a fórmula descrita no item “Estruturas de Dados”, novamente, se houver algum número repetido a função retorna um erro.

Caso o candidato passe por todos os testes, a função retorna 0, o que significa que o número pode ser colocado naquela posição.

- int solveRec (int *matriz, int offset)

A função recebe como parâmetro a estrutura base da matriz e um número inteiro que representa o deslocamento a partir dessa base – o índice a ser verificado naquele momento. A idéia do método é ser chamado apenas uma vez, e a partir daí varrer toda a matriz usando a técnica de backtracking para tentar colocar números em todas as posições, obedecendo às regras do sudoku. O primeiro passo é tentar colocar qualquer número de 1 a 9 na casa atual. Nenhuma heurística em especial é usada para definir a ordem de tentativas, simplesmente tenta-se comećando do 1, e chama-se a função colocaNum para verificar se o movimento não viola as regras do sudoku. Ao conseguir colocar um número, a função findNextZero é chamada a partir da base da matriz para achar a próxima casa vazia. Se for encontrada tal casa, o método solveRec é chamado recursivamente sobre aquela casa. Se ele retornar um sucesso, o método pai retorna sucesso, se retornar um insucesso, o método pai zera a casa atual e retorna um insucesso também, tentando assim colocar um número diferente daquele que causou o problema.

3. io.c

As funções no módulo io.c não precisam de descrições detalhadas, tudo o que fazem é ler as entradas do *Standard In* e imprime as entradas no *Standard Out*.

6. Complexidade

É difícil analisar a complexidade deste programa já que a entrada é sempre fixa, o que varia é a dificuldade da mesma, o que envolve não apenas a quantidade de valores pré-preenchidos, mas também as posições dos mesmos. Podemos fazer uma estimativa bastante rudimentar se considerarmos como N o tamanho da dimensão da matriz, ou seja, N=9. Dessa forma, podemos dizer que a possibilidade de soluções é de N^{N^2}, afinal para cada casa existem exatamente N possibilidades [1~9], para cada uma das N^2 casas do tabuleiro. Como temos o *backtracking*, essas N^{N^2} possibilidades podem ser testadas N^2 vezes. Esse seria o pior caso se o *backtracking* voltasse em todas as suas tentativas, só conseguindo na última, assim a complexidade ficaria da seguinte forma:

$$O(N^2) \times O(N^{N^2}) = \boldsymbol{O(N^{N^2})}$$

3. Testes Realizados

Foram realizados três testes baseados em entradas fornecidas. Como nenhuma heurística em especial foi utilizada, fiz duas baterias de testes apenas para efeito estatístico: uma que testa os números a partir de 1 até 9, e outra que testa os números a partir de 9 até 1.

Obs.: Nos códigos anexos, para facilitar a leitura, omiti o código exclusivo para a contagem do tempo, estes códigos foram adicionados apenas para efeito de teste e não fazem parte da estrutura final do programa.

- Primeiramente seguem os resultados dos testes na ordem decrescente:

Entrada	Saída	Número de Tentativas	Tempo de CPU (s)	Tempo de Usuário (s)	Tempo Total (s)
9 4 0 1 0 2 0 5 8 6 0 0 0 5 0 0 0 4 0 0 2 4 0 3 1 0 0 0 2 0 0 0 0 0 6 0 5 0 8 0 2 0 4 0 1 0 6 0 0 0 0 0 8 0 0 0 1 6 0 8 7 0 0 7 0 0 0 4 0 0 0 3 4 3 0 5 0 9 0 1 2	9 4 7 1 6 2 3 5 8 6 1 3 8 5 7 9 2 4 8 5 2 4 9 3 1 7 6 1 2 9 3 8 4 5 6 7 5 7 8 9 2 6 4 3 1 3 6 4 7 1 5 2 8 9 2 9 1 6 3 8 7 4 5 7 8 5 2 4 1 6 9 3 4 3 6 5 7 9 8 1 2	128	0.00000000	0.00000000	0.00043800
3 4 9 0 0 2 1 0 0 0 0 0 1 5 4 9 0 0 0 0 0 0 0 3 7 4 2 1 5 0 0 9 0 0 7 0 9 2 0 0 3 0 8 6 0 8 0 0 4 2 0 5 1 0 0 0 1 2 0 9 0 0 5 5 9 8 0 0 0 0 2 7 0 0 6 8 4 5 0 0 1	3 4 9 7 6 2 1 5 8 7 8 2 1 5 4 9 3 6 6 1 5 9 8 3 7 4 2 1 5 4 6 9 8 2 7 3 9 2 7 5 3 1 8 6 4 8 6 3 4 2 7 5 1 9 4 3 1 2 7 9 6 8 5 5 9 8 3 1 6 4 2 7 2 7 6 8 4 5 3 9 1	79	0.00000000	0.00000000	0.00038500
0 0 0 1 6 0 0 0 0 1 2 4 0 5 0 0 0 0 0 0 6 0 0 0 0 4 0 0 6 0 9 0 0 0 0 2 0 5 0 4 0 1 0 0 0 0 0 0 0 0 0 0 9 3 5 0 0 0 7 3 0 0 0 8 0 1 0 0 6 0 0 0 0 0 0 0 0 0 0 7 5	9 3 5 1 6 4 2 8 7 1 2 4 7 5 8 9 3 6 7 8 6 3 9 2 5 4 1 4 6 8 9 3 7 1 5 2 3 5 9 4 2 1 7 6 8 2 1 7 6 8 5 4 9 3 5 9 2 8 7 3 6 1 4 8 7 1 5 4 6 3 2 9 6 4 3 2 1 9 8 7 5	15.565	0.02000100	0.00400000	0.01986500
0 0 0 0 0 0 0 1 0 4 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 5 0 4 0 7 0 0 8 0 0 0 3 0 0 0 0 1 0 9 0 0 0 0 3 0 0 4 0 0 2 0 0 0 5 0 1 0 0 0 0 0 0 0 0 8 0 6 0 0 0	6 9 3 7 8 4 5 1 2 4 8 7 5 1 2 9 3 6 1 2 5 9 6 3 8 7 4 9 3 2 6 5 1 4 8 7 5 6 8 2 4 7 3 9 1 7 4 1 3 9 8 6 2 5 3 1 9 4 7 5 2 6 8 8 5 6 1 2 9 7 4 3 2 7 4 8 3 6 1 5 9	57.769.116	75.47671700	0.02400100	75.49921600

- Agora os resultados dos testes feitos na ordem crescente:

Entrada	Saída	Número de Tentativas	Tempo de CPU (s)	Tempo de Usuário (s)	Tempo de Relógio (s)
9 4 0 1 0 2 0 5 8 6 0 0 0 5 0 0 0 4 0 0 2 4 0 3 1 0 0 0 2 0 0 0 0 0 6 0 5 0 8 0 2 0 4 0 1 0 6 0 0 0 0 0 8 0 0 0 1 6 0 8 7 0 0 7 0 0 0 4 0 0 0 3 4 3 0 5 0 9 0 1 2	9 4 7 1 6 2 3 5 8 6 1 3 8 5 7 9 2 4 8 5 2 4 9 3 1 7 6 1 2 9 3 8 4 5 6 7 5 7 8 9 2 6 4 3 1 3 6 4 7 1 5 2 8 9 2 9 1 6 3 8 7 4 5 7 8 5 2 4 1 6 9 3 4 3 6 5 7 9 8 1 2	229	0.00000000	0.00000000	0.00055800
0 0 0 1 6 0 0 0 0 1 2 4 0 5 0 0 0 0 0 0 6 0 0 0 0 4 0 0 6 0 9 0 0 0 0 2 0 5 0 4 0 1 0 0 0 0 0 0 0 0 0 0 9 3 5 0 0 0 7 3 0 0 0 8 0 1 0 0 6 0 0 0 0 0 0 0 0 0 0 7 5	9 3 5 1 6 4 2 8 7 1 2 4 7 5 8 9 3 6 7 8 6 3 9 2 5 4 1 4 6 8 9 3 7 1 5 2 3 5 9 4 2 1 7 6 8 2 1 7 6 8 5 4 9 3 5 9 2 8 7 3 6 1 4 8 7 1 5 4 6 3 2 9 6 4 3 2 1 9 8 7 5	143	0.00000000	0.00000000	0.00046300
0 0 0 1 6 0 0 0 0 1 2 4 0 5 0 0 0 0 0 0 6 0 0 0 0 4 0 0 6 0 9 0 0 0 0 2 0 5 0 4 0 1 0 0 0 0 0 0 0 0 0 0 9 3 5 0 0 0 7 3 0 0 0 8 0 1 0 0 6 0 0 0 0 0 0 0 0 0 0 7 5	6 9 3 7 8 4 5 1 2 4 8 7 5 1 2 9 3 6 1 2 5 9 6 3 8 7 4 9 3 2 6 5 1 4 8 7 5 6 8 2 4 7 3 9 1 7 4 1 3 9 8 6 2 5 3 1 9 4 7 5 2 6 8 8 5 6 1 2 9 7 4 3 2 7 4 8 3 6 1 5 9	25.887	0.03600200	0.00000000	0.03299900
0 0 0 0 0 0 0 1 0 4 0 0 0 0 0 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 5 0 4 0 7 0 0 8 0 0 0 3 0 0 0 0 1 0 9 0 0 0 0 3 0 0 4 0 0 2 0 0 0 5 0 1 0 0 0 0 0 0 0 0 8 0 6 0 0 0	6 9 3 7 8 4 5 1 2 4 8 7 5 1 2 9 3 6 1 2 5 9 6 3 8 7 4 9 3 2 6 5 1 4 8 7 5 6 8 2 4 7 3 9 1 7 4 1 3 9 8 6 2 5 3 1 9 4 7 5 2 6 8 8 5 6 1 2 9 7 4 3 2 7 4 8 3 6 1 5 9	26.590.293	34.5662	0.0040	34.5686