UNIVERSIDADE FEDERAL DE MINAS GERAIS

ALGORITMOS 1
DOCUMENTAÇÃO TP3

MATHEUS TIAGO PIMENTA DE SOUZA - 2018054893
PROFESSORA JUSSARA M. ALMEIDA

Objetivo:

Este trabalho da disciplina de Algoritmos 1 tem como objetivo fazer com que os alunos apliquem os conceitos de "NP-Completude" e de aplicação de heurísticas vistos em aula, além de praticarem e aperfeiçoarem técnicas de programação. Dessa forma, dado o problema de encontrar a solução de um dado Sudoku, deve-se implementar uma heurística que tente resolvê-lo.

Ambiente de desenvolvimento, compilação e execução:

O projeto foi desenvolvido no ambiente de C++ e compilado nas máquinas do DCC utilizando o compilador G++. Para compilar, rodar *make* no diretório onde estão os arquivos. Em seguida, execute o executável gerado, passando por linha de comando o arquivo texto que será usado.

O problema: Sudoku:

Dado um quadrante $N \times N$ com algumas células preenchidas e as dimensões I e J dos subquadrantes, preencher as células restantes com números de 1 a N obedecendo às seguintes restrições:

- Não pode haver dois números iguais na mesma linha;
- Não pode haver dois números iguais na mesma coluna;
- Não pode haver dois números iguais no mesmo subquadrante.

Para tal feito, foi recomendado mapear o Sudoku em um grafo, e transformar o problema para coloração de vértices. Qualquer heurística que não fosse por força bruta poderia ser aplicada.

Análise do algoritmo:

Nesse algoritmo o Sudoku foi mapeado para um grafo a partir de matriz de adjacências, pois apesar do custo maior para construí-la a consulta na mesma é O(1), e serão feitas consultas constantemente.

A primeira etapa é montar a matriz, que foi feita da seguinte forma (representando o pseudo-código como função, mas a mesma é feito diretamente no *main*):

Montagem_matriz(N,I,J){:

Gerar matriz M[N][N];

Para cada célula da matriz:

Marcar como 1 cada adjacência entre a célula e todas na mesma linha;

Marcar como 1 cada adjacência entre a célula e todas na mesma coluna;

Marcar como 1 cada adjacência entre a célula todas no mesmo quadrante;

End for;

End função;

É fácil gerá-la uma vez que temos as dimensões do sudoku e de cada subquadrante. Por exemplo, para marcar as linhas basta achar o primeiro vértice da linha da célula analisada, e marcar a adjacência dessa célula analisada com o primeiro vértice, adicionar N ao primeiro vértice e repetir o processo.

A heurística utilizada foi um meio termo entre uma estratégia gulosa e *backtracking*, baseada no slide disponível na página do moodle. A parte gulosa diz respeito a analisar o vértice escolhido, para ver se o mesmo tem cor ou não, e tentar achar uma solução válida para o mesmo. Assim, se tal solução existir, ao final da execução para esse vértice o tabuleiro vai ter a cor desse vértice (ou uma cor possível para ele, se inicialmente ele não tinha cor) totalmente preenchida, e essa solução não é alterada, pois o algoritmo julga essa ser a coloração certa na solução final (característica base de estratégias gulosas, onde fazer a melhor escolha local acreditando gerar a melhor solução global).

Já a parte do backtraking corresponde ao fato de que, se o vértice analisado no momento não tem cor, tentar todas as cores possíveis para o mesmo, e se alguma der errado voltar e tentar a próxima. A heurística não analisa todos os vértices do grafo para escolher qual cor tentar, mas sim apenas as cores adjacentes da célula em questão. É assim que o backtracking funciona, tentar todas as cores possíveis para cada vértice. De fato, se tirássemos a condição de não alterarmos uma solução de uma cor, caso a próxima tentativa desse errado, teríamos praticamente implementado um backtracking, com algumas análises para tentar otimizar.

O algoritmo tem três funções para resolver o sudoku, sendo elas:

- *possible_colors*:

Uma função que escreve em um vetor todas as cores que o vértice analisado tem adjacência, de forma que no final o vetor contém todas as cores possíveis para o vértice.

- *check_color*:

Dada a cor e o vértice que se desejam analisar, o algoritmo retorna se essa cor é possível para esse vértice (pode ser a cor dele) ou não.

- s*olve*:

Dados vários parâmetros, como quadranet, flag se houve remoção de vértice, etc., a função tenta achar uma solução para certo vértice e certa cor dada.

A função *Solve* faz uso de uma memória, que consiste em uma pilha, que foi uma estrutura implementada com apenas três funções (tirando o construtor): adicionar, remover o primeiro e pegar o primeiro elemento. Tal estrutura é uma lista encadeada simples, onde cada Nó guarda o endereço do próximo, além de parâmetros importantes dos vértices, como coordenadas e o vértice passado. Segue um diagrama de uma pilha:

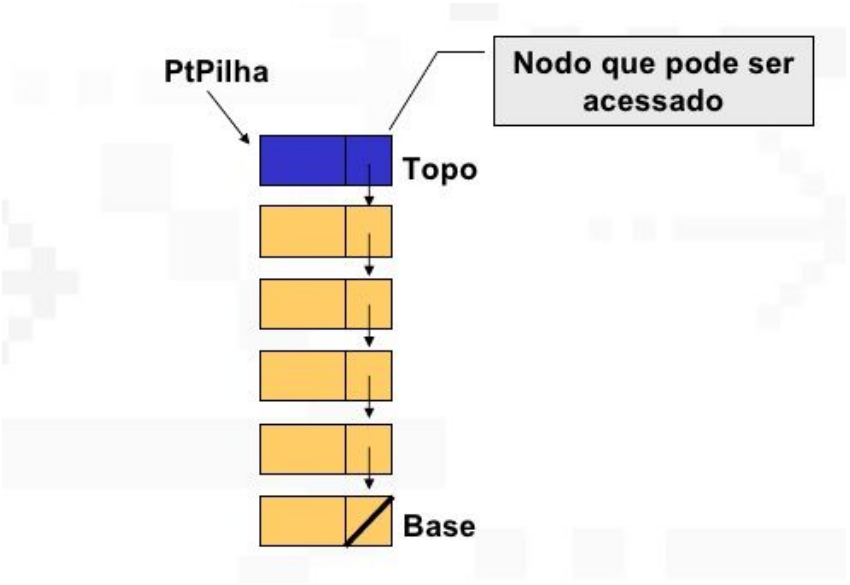


Diagrama de uma pilha, disponível em <https://pt.slideshare.net/CriatividadeZeroDocs/pilhas-e-filas>

Abaixo, o pseudocódigo da função solve:

Solve(v,ant,...): //v = vertice atual; ant = vertice da iteracao passada.
Setar variáveis result e aux = false;
Para cada vértice dentro do quadrante analisado:
Se ant não for adjacente à v:
Se a cor(v) == cor(ant):
aux = true e adcionar esse vértice a memória;
Senão se a cor desse vértice for 0 e não houve remoção na iteração passada:
aux = check_colors(v,cor);
Se aux==true:
Escrever essa cor em v e adicioná-lo à memória;
Senão:
Se chegamos no vértice que foi removido:
Setar a flag como falso;
End if;
Se aux==true:
Se estivermos no último subquadrante:
result = true e retornar result;
Senão se não existirem mais quadrantes a direita:
result = chamar Solve para o quadrante abaixo mais a esquerda;
Senão
result = chamar Solve para o quadrante a direita;
End if;
End loop;
Se result == false:
mem = pegar o primeiro elemento da pilha de memória;
Se a cor(mem) == cor(v)
Remover mem da memória;
Se mem não era cor fixa
cor(mem) = o;
Setar a flag de removido para true;
result = chamar Solve para o quadrante anterior;
End if;
End if;
End Solve;

Vamos analisar melhor a parte de remoção: caso não seja possível adicionar a cor em nenhum vértice do quadrante atual, se tenta voltar na última adição na memória, que consiste no último vértice colorido, e remover essa coloração e passar para o próximo. Caso chegue

no ponto onde volte no primeiro vértice, essa cor é descartada e o algoritmo tenta a próxima cor disponível (essa parte é tratada no main). Caso o vértice anterior seja um vértice de cor fixa (já veio do arquivo), essa cor não é deletada e o algoritmo vai voltar no quadrante anterior.

No *main*, nós chamamos a função Solve para cada vértice do primeiro subquadrante, passando uma cor possível e as coordenadas do segundo subquadrante.

Complexidade do algoritmo:

- Complexidade de tempo: Na montagem da matriz existem 3 loops aninhados, onde o primeiro vai executar *N* vezes e os outros dois vão executar em um quadrante, que são frações de *N*, gerando assim uma complexidade da ordem de $O(N^3)$. Já a função solve é recursiva, e sua equação de recorrência para um caso onde não haja remoção de vértices da memória é da seguinte forma:

$$T(n) = n^2 + T(n-1).$$
$$T(n) = n^3$$

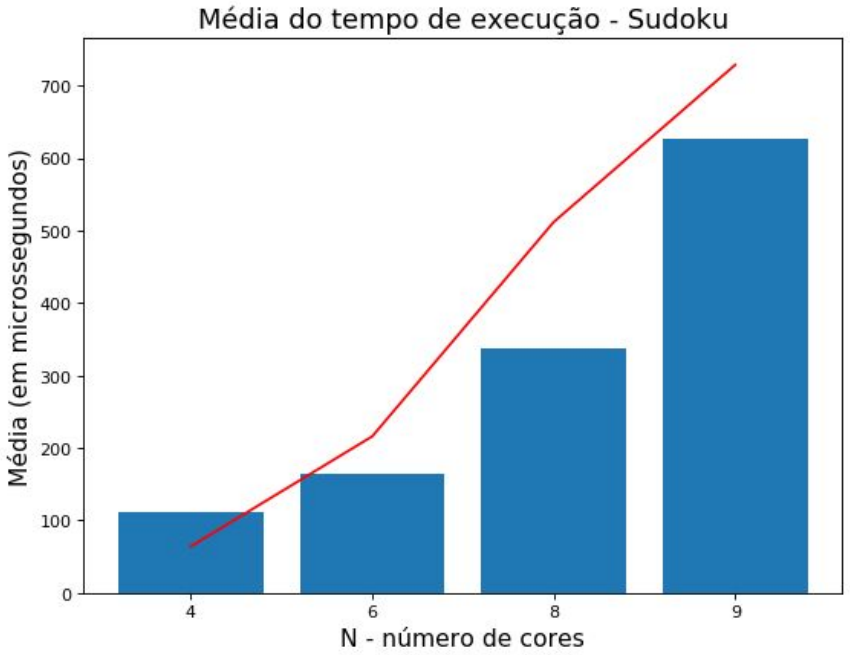
Como chamamos Solve N vezes, temos que nosso algoritmo tem complexidade da ordem de $O(N^4)$.

O pior caso seria um sudoku onde se testassem todas as cores e no final não fosse possível adicioná-las, sendo necessário apagá-las. Note que poderíamos testar a mesma cor no quadrante mais de uma vez, e dar errado de todas as formas. Nesse caso nossa função pode até ser $T(n) = n^2 + c*T(n+1)$ em alguns momentos, onde c = cte, mas isso não mudaria a ordem de complexidade, supondo sudokus resolvíveis, pois casos onde sempre se adiciona e depois se remove infinitamente não acontecem nesses sudokus. Portanto, complexidade da ordem de $O(cN^4)$.

- Complexidade de espaço: O maior custo de memória está com a geração da matriz, e portanto complexidade da ordem de $O(N^2)$. A memória nunca vai chegar a N^2, seŕa sempre uma fração da mesma. No pior caso não faria muita diferença, uma vez que apagamos a memória.

Avaliação experimental:

Para realizar a avaliação experimental, pelo fato de ser difícil gerar casos válidos de sudoku, foram compilados todos os exemplos do dataset. Utilizado python para gerar os gráficos. Cada teste foi repetido 20 vezes Seguem os resultados:



Em azul temos a média dos tempos de execução, enquanto a curva em vermelho é $O(N^3)$. Podemos perceber que o tempo aumenta à medida que N aumenta, o que era esperado para complexidade da ordem de N^3. Também, percebemos que os dados estão abaixo da curva de $O(N^3)$, o que comprova a análise de que o algoritmo tem essa complexidade assintótica.

Apesar da variação, ela é muito pequena se comparado com os valores da média (no máximo 5%), e não apresenta um padrão tal qual a média. Podemos inferir que o algorítmo é bem regular, para N<=9.

Analisando tanto as saídas quanto os tempos de execuções, o formato que mais obteve sucesso foi o 6x6, seguido do 4x4. Isso ocorreu provavelmente porque nesses tamanhos a chance de fazer uma escrita que, apesar de dar certo inicialmente, impede a solução completa é menor, pelo fato de serem menores. Nos maiores com muitos vértices sem cor a chance é bem mais alta.

Conclusões:

Com esse trabalho foi possível aprimorar os conhecimentos de "NP-Completude" e heurísticas de coloração de grafos, entendendo seus conceitos e porque coloração de grafos é um problema NP-Completo. Também foi possível compreender mais sobre testes e análise de complexidade de algoritmos.