Documentação do TP3 de Algoritmos

Lucas Resende Pellegrinelli Machado

Outubro 2019

1 Introdução

O trabalho proposto diz respeito ao famosos jogo de sudoku. O objetivo do jogo é preencher o tabuleiro (que pode vir de diversos tamanhos) completando as células que ainda não possuem números de forma a não deixar que na mesma linha, coluna e quadrante existam duas células com o mesmo número. No trabalho devemos resolver os tabuleiros usando heurísticas de forma a aproximar polinomialmente uma solução que toma complexidade exponencial caso seja implementada usando o backtracking (algoritmo que sempre encontra uma solução caso exista uma).

2 Modelagem

Foi indicado resolver o problema o transformando em um problema de coloração de grafos, onde cada nó representa uma célula do tabuleiro e tem com sí o seu valor (caso exista um) e uma lista de adjacências onde todos os outros nós que correspondem a células que ficam na mesma linha, coluna ou quadrante da célula atual ficam armazenados. Dessa forma, caso duas células não podem ter o mesmo número em sí, elas estariam ligadas e assim o problema de coloração de grafos não deixaria que elas fossem coloridas da mesma "cor" (que no nosso caso corresponde a um dos números possíveis em cada célula). Caso exista solução para o problema da coloração, todos os nós em uma mesma linha, coluna e quadrante tem números diferentes.

3 Implementação

A implementação utilizada foi bastante simples e utilizou uma heurística que consegue passar em diversos dos testes, porém como esperado, não em todos. Primeiro resolvemos as células "óbvias" (quando digo "óbivas", me refiro a células que só possuem um número possível baseado no resto do tabuleiro) iterativamente, ou seja, procuramos uma célula óbvia, caso achamos, resolvemos ela e depois com o tabuleiro atualizado, procuramos outra célula óbiva, até que não existam mais. Quando chegamos no caso de não existirem células

óbvias no tabuleiro, temos que escolher uma das células e de alguma maneira decidir algum número para entrar nela e assim repetir o processo. Porém é impossível ter certeza de qual número colocar em uma célula não óbvia sem testar todas as opções (que é o caso da solução de complexidade exponencial para o problema). Para minimizar a chance de erro dessa escolha, ao invés de escolher qualquer célula para preencher, procuramos a que possuí o menor número de opções, visto que a probabilidade de escolhermos a errada vai cair. Assim que escolhemos essa célula, preenchemos com um valor possível aleatório (na implementação foi utilizado o menor valor por conveniência) e voltamos para o primeiro passo buscando novas células óbvias que se originaram com esse novo valor preenchido.

Para checar se o resultado correto foi obtido no final da execução, simplesmente passamos por cada uma das células e testamos se a quantidade de opções de números para aquela célula é igual a 1 e se essa única opção é o valor que a célula assumiu. Caso esses dois testes passem para cada uma das células, então todas as células estão corretas e então nosso tabuleiro está correto.

O algoritmo quando identifica que ele tomou alguma decisão errada na hora de preencher uma célula não óbvia (ou seja, alguma célula deu conflito), como ele não tem o histórico de modificações (por ser uma heurística e não o backtracking) ele para de tentar resolver e retorna a entrada como a saída.

3.1 Backtracking limitado opcional

Também foi implementado com fins de aprendizado o algoritmo de backtracking, que consegue resolver qualquer sudoku porém em um tempo exponencial nas entradas. Isso foi feito de uma forma a não ser apenas uma opção (usar ou não o backtracking) mas como uma parâmetro da heurística. Foi definida uma variável que diz respeito ao número de "correções" que podem ser feitas durante a resolução de um tabuleiro, então enquanto o número de vezes que o sudoku conflitou após alguma decisão for menor que esse parâmetro, o algoritmo tem a capacidade de voltar, tentar uma outra opção naquela célula e continuar. Caso o limite seja atingido, um erro não será mais corrigido e o programa parará. Isso foi feito para poder controlar o tempo máximo de execução do programa de certa forma, visto que se o usuário tem tempo disponível, ele pode aumentar o valor desse parâmetro e potencialmente esperar mais tempo para que o programa dê uma resposta (que ainda sim, pode ser inválida caso o sudoku faça o backtracking errar muito e assim ultrapassar o limite de correções estabelecido).

Uma estratégia utilizada para que isso funcionasse foi o *iterative deepening* visto que caso isso não fosse definido, o seguinte problema poderia acontecer:

Suponha que você se encontra no momento da primeira decisão não óbvia a ser tomada durante a resolução e você toma inicialmente a primeira opção. Caso o sudoku resultante dessa escolha seja muito complexo, o número de erros dentro dessa "linha" que foi escolhida no momento da decisão pode ser muito alto e assim superar a quantidade de erros permitida pelo programa. Dessa forma, mesmo que o problema fosse muito simples e a solução fosse trivial caso

outra opção tivesse sido escolhida naquele momento, o programa retornará que não encontrou resposta pois ficou preso na linha onde ele já tinha escolhido a opção errada na primeira tomada de decisão.

O *iterative deepening* resolve esse problema primeiro testando todas as linhas da primeira decisão a ser tomada, depois ele irá percorrer as primeiras decisões que apareceram depois de ter escolhido cada uma das opções da última decisão. Uma boa analogia é que esse método é como se fosse um Breadth-First-Search comparando com a recursão comum que seria como o Depth-First-Search.

4 Análise de Complexidade

Da forma com que foi implementado, a complexidade de espaço se torna bem simples de ser calculada visto que todos os tensores utilizados foram declarados direto no construtor da classe que resolve o Sudoku. Existem cinco tensores de dimensão N^2 e dois de dimensão N^3, logo ocupamos o espaço de $5N^2 + 2N^3 = O(N^3)$. Dentro do resto do código nenhum outro tensor foi criado.

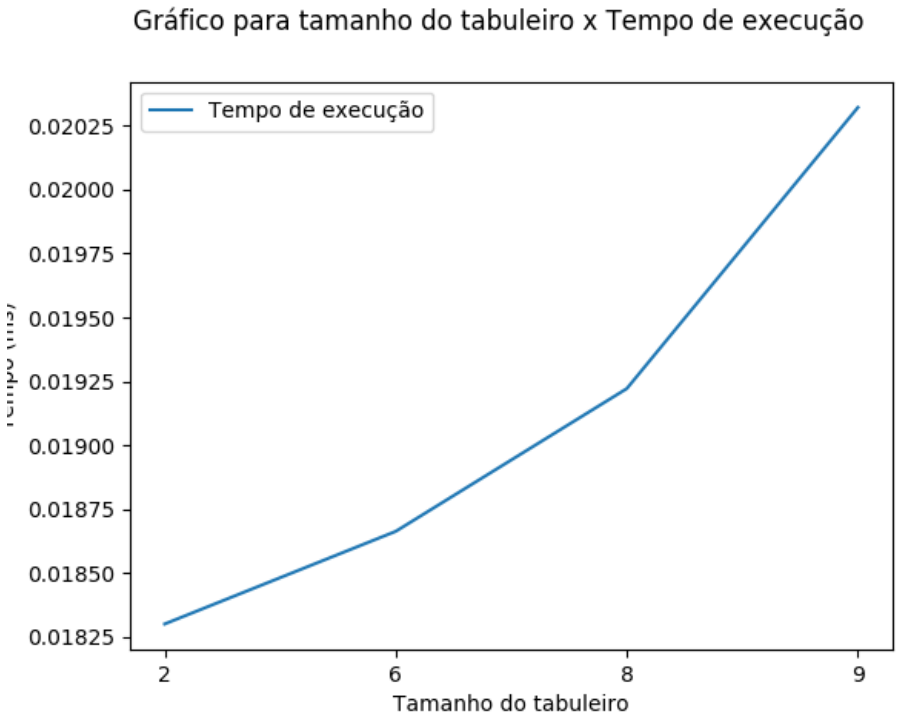
Para a complexidade de tempo, primeiro vamos desconsiderar a adição do backtracking (como se o limite de backtracking fosse 0) pois sabemos que como o backtracking é um algoritmo exponencial, com o limite do backtracking tendendo a infinito, nosso algoritmo vira exponencial, assim vamos analisar apenas a complexidade da heurística. Analisando o método `bool solve_depth(int depth)`, que é o responsável por resolver o sudoku (assumindo `int depth = 0` pois desconsideramos o backtracking), de cara encontramos um `while` que quebra quando o número de células com valor definido é igual ao número de células no tabuleiro. Sabendo que em cada iteração desse `while` obrigatóriamente adicionamos pelo menos uma célula nesse total de células já com valor definido, o pior caso é que esse loop seja executado N^2 vezes. Logo a seguir encontramos um `for` que percorre as células que tiveram seus valores definidos na última iteração, ou seja, mais um potêncial de N iterações. Logo mais encontramos um outro `for` aninhado que passa por cada um dos nós adjacentes ao nó recentemente adicionado à lista de nós com valor conhecido, adicionando mais um outro potêncial de N iterações. Voltando no aninhamento até ficarmos apenas aninhados ao `while` temos um `if(depth == 0 || branch_count > MAX_BRANCH_COUNT)`, que será o que vai ser executado com o backtracking removido, e dentro desse `if` existe apenas um `for` que acha o nó com menos opções de valores e assim adiciona N iterações. Assim dentro do `while` temos $N^2 + N = O(N^2)$ iterações, juntando com o `while`, ficamos com uma complexidade de tempo de $O(N^4)$.

5 Análise de Experimental

O gráfico de tempo de execução por tamanho do tabuleiro vem a seguir. Foram feitas 1000 medições em cada um dos casos de teste disponibilizados e para cada tamanho de tabuleiro foi computado a média do tempo gasto para resolver.

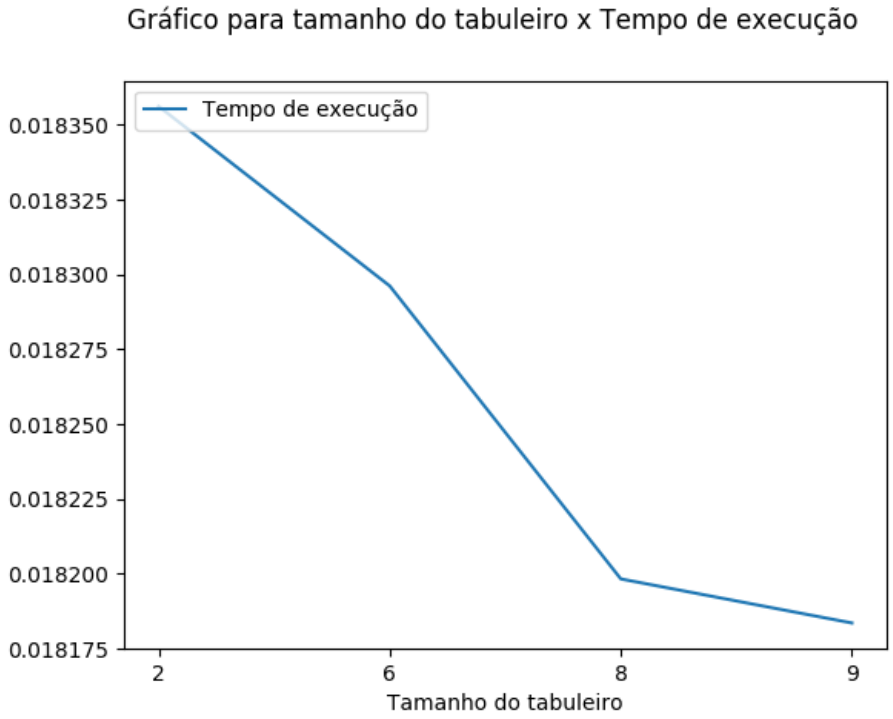
Um ponto importante é que nesse teste, a funcionalidade extra de backtracking foi completamente desabilitada.

5.1 Gráfico do tempo médio de execução por tamanho



Esse gráfico é como o esperado visto que temos uma complexidade de $O(n^4)$ ele cresceria muito rápido a medida que a entrada aumenta

5.2 Gráfico do desvio padrão do tempo de execução por tamanho



O gráfico decresce pois quanto maior a entrada, menos inconstante é o número de chutes que são necessários para resolver o sudoku e como o chute toma mais tempo, o gráfico fica dessa forma.

6 Conclusão

O problema foi modelado e implementado com sucesso e os testes representaram o que era esperado quando analisamos o algoritmo de forma teórica

O trabalho compriu o propósito de nos fazer praticar o uso de redução de um problema à outro, tanto na parte de modelagem quanto na implementação, que é algo bem diferente do que estamos acostumados a lidar no curso até então no curso.

Também me dei a liberdade de por gostar do tema, procurar sobre temas um pouco mais complexos como iterative deepening e backtracking.