Trabalho Pratico 3 – Algoritmos I

Luiz Felipe M. D. Nery

Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG

luiznery@dcc.ufmg.br

1. Introdução

O seguinte trabalho apresenta uma heurística para resolver sudokus de ordem $n \times m$ transformando o puzzle em um problema de cobertura exata (exact cover) e usando a técnica Dancing Links e uma implementação própria do Algorithm X, ambos propostos por Knuth.

1.1. Descrição do Problema

Dado um sudoku de ordem n, ou seja, com uma grade $n^2 \times n^2$ e com quadrantes de tamanho $i \times j$, podendo já haver células preenchidas (pistas), deseja-se preencher a grade respeitando as regras do puzzle, deve-se utilizar os números $\{1,2,\dots,n\}$ sem poder repetir um número na própria célula, ou seja, alocando somente um numero por célula, também não é permitida a repetição de números na mesma linha, na mesma coluna e dentro do mesmo quadrante $i \times j$.

Um caso específico do sudoku que será tratado nesse trabalho, é o sudoku próprio. Um sudoku próprio possui somente uma solução e é possível ser resolvido seguindo somente passos lógicos.

Foi pedido que se implementasse uma heurística capaz de resolver o puzzle.

2. Implementação e análise de complexidade

Para resolver o problema, primeiramente transformou-se o sudoku em um problema de cobertura exata (exact cover), que será explicado mais a frente, e em seguida foi utilizado o algorithm X associado a técnica de dancing links (DLX), proposto por Donald Knuth, para resolver a instancia do problema de cobertura exata criada.

Essa abordagem foi escolhida pois é extremamente eficiente do ponto de vista temporal e utiliza uma abordagem bem elegante e interessante do ponto de vista da transformação do sudoku em um problema de cobertura exata.

A solução aqui proposta pode ser dividida em três grandes partes, a transformação do sudoku em um problema de cobertura exata, a criação da estrutura de dados que suporta DLX e o algoritmo responsável por resolver a instancia do problema de cobertura exata a partir da estrutura de dados criada. Cada uma delas serão descritas a seguir.

Para implementar o programa foi escolhida a linguagem C++ e utilizado o compilador g++. Para compilar o programa foi utilizado um arquivo makefile, que será entregue com o código.

A seguir é descrito os formatos de entrada, saída, a transformação em um problema de cobertura exata, a estrutura de dados implementada, e os algoritmos utilizados.

2.1. Transformação do Soduku em Problema de Cobertura Exata

Um problema de cobertura exata consiste em dada uma coleção S de subconjuntos de um conjunto U, uma cobertura exata é uma subcoleção S' de S tal que cada elemento de U está contido em exatamente um subconjunto de S'. O problema de cobertura exata é um problema NP-Completo. A seguir podemos ver um exemplo de um problema de cobertura exata e sua solução.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Figura 1: Problema de cobertura exata representado por uma matriz. Uma solução é escolher as linhas 1,4 e 5 – contando a partir de 1. **Fonte:** Retirado de Knuth, 2000.

Podemos representar o problema de cobertura exata através de uma matriz, em que cada linha representa um subconjunto de S e cada coluna representa um elemento de U. Caso um elemento de U pertença a um subconjunto de S, o campo na linha referente a esse subconjunto e na coluna referente ao elemento será igual a 1, caso contrário igual a 0. Dessa forma, para resolver o problema basta escolher as linhas que quando somadas resulte em uma linha de somente 1's, ou seja, representando que a coleção de subconjuntos escolhida tenha exatamente um elemento de cada do conjunto U.

O primeiro passo do trabalho foi fazer a transformação do sudoku em uma instancia do problema de cobertura exata. Para tanto, o sudoku foi modelado da seguinte forma. Criamos uma matriz M, $n^3 \times 4n^2$, na qual modelaremos o sudoku a partir das suas regras.

Nessa matriz temos n^3 linhas, cada linha representará cada célula do sudoku com cada um dos n valores do conjunto $N = \{1,2, \dots, n\}$ de possibilidades de valores para uma célula. Como temos n^2 celulas e para cada uma dessas células, n possíveis valores temos n^3 linhas. Isso implica que escolher uma linha da matriz M significa escolher um valor específico para uma célula especifica do sudoku.

As colunas da matriz M representarão as restrições, isto é, as quatro regras do sudoku, (1) não se pode ter mais de um elemento na mesma célula, (2) não se pode ter o mesmo elemento mais de uma vez na mesma linha, (3) na mesma coluna e (4) no mesmo quadrante. Temos $4n^2$ colunas pois cada coluna representa uma dessas restrições para cada célula do sudoku. Então na solução não basta escolher somente n^2 linhas de M, deve-se escolhê-las obedecendo as restrições impostas.

Uma grande parte do trabalho consistiu em descobrir como modelar a matriz M. Separamos a matriz M em 4 partes, uma para cada regra, as primeiras n^2 colunas de M representam a regra (1) para cada n^2 células, as próximas n^2 colunas representam a regra (2) para cada uma das n^2 células e assim por diante. A seguir vamos explicar a modelagem de cada uma dessas regras e usar um sudoku 2×2 como exemplo. Nota-se que cada célula é representada por um valor c_i que vai de 0 a $n^2 - 1$, a seguir temos a representação dos valores de cada célula.

0	1
2	3

Figura 2: Representação dos valores c_i para um sudoku 2×2, com quadrantes 1×2. **Fonte:** Elaborado pelo autor.

A primeira regra, não poder haver mais de um número por célula foi modelada da seguinte forma, na mesma coluna, para todas as linhas que representam aquela célula para todos os valores representados há um 1. Dessa forma, garantimos que não será possível escolher linhas que representam as mesmas células com dois valores diferentes.

Célula	Valor	Regra 1				Regra 2				Regra 3				Regra 4			
0	1	1															
0	2	1															
1	1		1														
1	2		1														
2	1			1													
2	2			1													
3	1				1												
3	2				1												

Figura 3: Regra 1 para um sudoku 2×2 modelada na Matriz M. **Fonte:** Elaborado pelo autor.

Para representar a segunda regra, cada elemento que compartilha a mesma linha e o mesmo valor tem um 1 na mesma coluna. Nota-se que cada n colunas representa uma linha. Dessa forma nunca poderemos escolher valores iguais para duas células diferentes na mesma linha.

Célula	Valor	Regra 1				Regra 2				Regra 3				Regra 4			
0	1	1				1											
0	2	1					1										
1	1		1			1											
1	2		1				1										
2	1			1				1									
2	2			1					1								
3	1				1			1									
3	2				1				1								

Figura 4: Regra 2 para um sudoku 2×2 modelada na Matriz M. **Fonte:** Elaborado pelo autor.

Similarmente a regra 2, a regra 3 é modelada na matriz M de forma que células na mesma coluna do sudoku e com mesmo valor tenham 1's em uma mesma coluna. Assim, não é possível escolher duas linhas de M que representem elementos da mesma coluna do sudoku e com o mesmo valor.

Célula	Valor	Regra 1				Regra 2				Regra 3				Regra 4			
0	1	1				1				1							
0	2	1					1				1						
1	1		1			1						1					
1	2		1				1						1				
2	1			1				1		1							
2	2			1					1		1						
3	1				1			1				1					
3	2				1				1				1				

Figura 5: Regra 3 para um sudoku 2×2 modelada na Matriz M. **Fonte:** Elaborado pelo autor.

Para satisfazer a regra 4, devemos modelar as colunas da matriz M referentes a essa regra de forma que não seja possível escolher dentro do mesmo quadrante, valores iguais para duas células diferentes. Então para cada n colunas de M dentro do espaço destinado a regra 4, cada

uma dessas colunas representa um valor e um quadrante, dessa forma, quando duas células pertencem a um mesmo quadrante e tem o mesmo valor, ambas devem ter 1's nessa coluna.

Célula	Valor	Regra 1				Regra 2				Regra 3				Regra 4			
0	1	1				1				1				1			
0	2	1					1				1				1		
1	1		1			1						1		1			
1	2		1				1						1		1		
2	1			1				1		1						1	
2	2			1					1		1						1
3	1				1			1				1				1	
3	2				1				1				1				1

Figura 6: Regra 4 para um sudoku 2 × 2 modelada na Matriz M. **Fonte:** Elaborado pelo autor.

Como podem ser passadas grades de sudoku com pistas, para cada célula que já está preenchida devemos deixar somente a linha referente a aquele valor e apagar todas as outras, de forma que nunca teremos a opção de escolher um valor diferente para uma célula que está originalmente preenchida. Assim garantimos que as pistas estarão na solução. O exemplo abaixo considera uma matriz M já com as linhas referentes a células preenchidas com pistas retiradas.

Sudoku

1	

Célula	Valor	Regra 1				Regra 2				Regra 3				Regra 4			
0	1	1				1				1				1			
1	1		1			1						1		1			
1	2		1				1						1		1		
2	1			1				1		1						1	
2	2			1					1		1						1
3	1				1			1				1				1	
3	2				1				1				1				1

Figura 7: Sudoku 2 × 2 (1 × 2) com uma dica e a matriz M construída em cima dele. A única solução possível é escolher as linhas (0,1), (1,2), (2,2), (3,1) da matriz M. **Fonte:** Elaborado pelo autor.

2.2. DLX

Um simples algoritmo de backtracking poderia ser usado para resolver a instancia do problema descrita acima. Seja A a matriz do problema de cobertura exata e supondo que a escolha de coluna aconteça sempre escolhendo a que tem menos 1's.

If A is empty, the problem is solved; terminate successfully.
Otherwise choose a column, c (deterministically).
Choose a row, r, such that $A[r, c] = 1$ (nondeterministically).
Include r in the partial solution.
For each j such that $A[r, j] = 1$,
    delete column j from matrix A;
    for each i such that $A[i, j] = 1$,
        delete row i from matrix A.
Repeat this algorithm recursively on the reduced matrix A.

Figura 8: Algoritmo para resolver o problema de cobertura exata. **Fonte:** Retirado de Knuth, 2000.

Por exemplo, primeiro escolheríamos a linha (0, 1) e retiraríamos as linhas (1,1) e (2,1).

Célula	Valor	Regra 1				Regra 2				Regra 3				Regra 4				
0	1	1				1				1				1				Solução
1	1		1			1						1		1				Removido
1	2		1				1						1		1			
2	1			1				1		1						1		Removido
2	2			1					1		1						1	
3	1				1			1				1				1		
3	2				1				1				1				1	

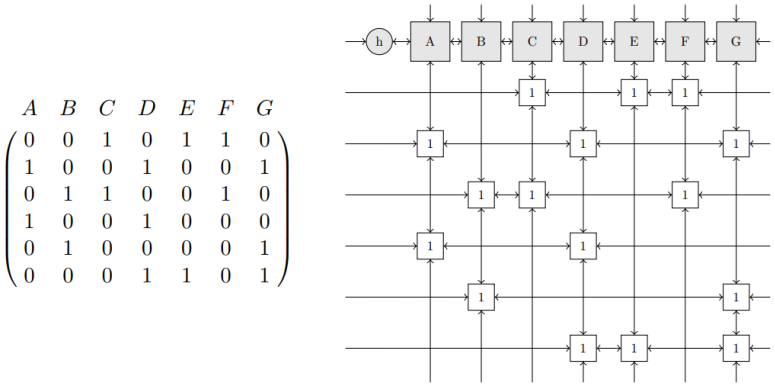
Figura 9: Escolhe $(0, 1)$ e retira as linhas (1,1) e (2,1). **Fonte:** Elaborado pelo autor.

Em seguida escolheríamos a linha (1,2) e retiraríamos (3,2). Seguindo esses passos chegamos que as duas linhas que sobram, (2,2) e (3,1), também farão parte da solução.

Célula	Valor	Regra 1				Regra 2				Regra 3				Regra 4				
0	1	1				1				1				1				Solução
1	1		1			1						1		1				Removido
1	2		1				1						1		1			Solução
2	1			1				1		1						1		Removido
2	2			1					1		1						1	Solução
3	1				1			1				1				1		Solução
3	2				1				1				1				1	Removido

Figura 10: Escolhe $(1, 2)$ e retira alinha (3,2) e adicionamos (2,2) e (3,1) a solução. **Fonte:** Elaborado pelo autor.

Contudo, esse processo de remoção de linhas e percorrer coluna a coluna e linha por linha pode ser muito custoso, especialmente para uma matriz tão grande. Dai vem a ideia de representar usando de acordo com a técnica DLX que consiste em representar cada 1 como um objeto Node. Cada Node, nessa implementação guardará os campos $L[x], R[x], U[x], D[x], C[x]$ e $lin[x]$.



$$\begin{matrix} A & B & C & D & E & F & G \end{matrix}$$
$$\begin{pmatrix} 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 \end{pmatrix}$$

Figura 11: Representação da matriz C, do lado direito, com DLX. **Fonte:** Harryson, 2014.

$L[x]$ irá guardar o Node a esquerda, $R[x]$ guarda o Node a direita, $U[x]$ guarda o Node acima, $D[x]$ guarda o Node abaixo, $C[x]$ guarda o Node do cabeçalho que indica a coluna original e $lin[x]$ guarda qual linha da matriz M aquele Node representa. Nota-se que as conexões de cada Node com seu cabeçalho foram suprimidas. Quando Node for um cabeçalho também teremos as variáveis $size[x]$, que guardará o número de Nodes naquela coluna. Também é necessário criar um header h, que será nossa conexão com essa estrutura de dados criada.

Dessa forma as operações de retirar todas as linhas que compartilham na mesma coluna células com 1's, apagar linhas e depois as reposicionar, operação que será explicada mais afrente, são feitas de forma muito eficiente.

2.3. Algorithm X

Para resolver o sudoku usamos, então, o seguinte algoritmo, em que h é o header que aponta para a estrutura de dados criada, k é um inteiro que indica em qual passo da recursão estamos e s é a solução.

```
1  search(h, k, s) =
2      if R[h] = h then
3          print_solution(s)
4          return
5      else
6          c ← choose_column_object(h)
7          r ← D[c]
8          while r ≠ c
9              s ← s + [r]
10             j ← R[r]
11             while j ≠ r
12                 cover(C[j])
13                 j ← R[j]
14             search(h, k+1, s)
15             // Pop data object
16             r ← s_k
17             c ← C[r]
18             j ← L[r]
19             while j ≠ r
20                 uncover(C[j])
21                 j ← L[j]
22             r ← D[r]
23         uncover(c)
24         return
```

```
1  cover(c) =
2      L[R[c]] ← L[c]
3      R[L[c]] ← R[c]
4      i ← D[c]
5      while i ≠ c
6          j ← R[i]
7          while j ≠ i
8              U[D[j]] ← U[j]
9              D[U[j]] ← D[j]
10             S[C[j]] ← S[C[j]] − 1
11             j ← R[j]
12         i ← D[i]
```

```
1  uncover(c) =
2      i ← U[c]
3      while i ≠ c
4          j ← L[i]
5          while j ≠ i
6              S[C[j]] ← S[C[j]] − 1
7              U[D[j]] ← j
8              D[U[j]] ← j
9              j ← L[j]
10         i ← U[i]
11     L[R[c]] ← c
12     R[L[c]] ← c
```

Figura 12: AlgorthmX. **Fonte:** Retirado de Harryson, 2014.

Algumas modificações foram feitas em cima desse algoritmo, como por exemplo, nossa solução é composta por linhas diretamente. E sempre escolhemos a coluna como sendo aquela que tem menor valor $size[x]$.

Um outro ponto a ser observado é que se por tratar de um heurística o se o algoritmo implementado chega ao fim de uma arvore de recursão, ou seja, quando a matriz ficar vazia, o que acontece quando ou todas as linhas já foram escolhidas ou foram excluídas, o algoritmo retorna. Portanto, garantidamente, resolve sudokus próprios com solução logica em que não há nenhuma derivação.

2.4. Complexidade Temporal e Espacial

A complexidade de tempo dada por Knuth para seu algoritmo é exponencial, uma vez que ele testa todas as possibilidades de solução. Contudo como limitamos ele a testar somente uma arvore de recursão, podemos dizer que a complexidade é polinomial, uma vez que serão feitas somente n^2 chamas recursivas, uma para cada célula, e para cada uma dessas um número polinomial de operações é feito.

Em relação a complexidade de tempo e espaço da transformação do sudoku para um problema de cobertura exata, podemos dizer ser $O(n^5)$, uma vez que devemos criar uma matriz de tamanho $n^3 \times 4n^2$. A complexidade de tempo para a criação da estrutura de dados proposta também será $O(n^5)$ pois também faremos o processo uma vez para cada célula da matriz M.

Contudo, sem dúvidas esse processo pode ser otimizado, de forma que é possível reduzir a complexidade de espaço para $O(n^2)$, uma vez que podemos criar cada linha da estrutura de dados proposta a medida que calculamos cada linha da matriz M, sem a necessidade de armazenar todas as linhas de M, mas sim, somente a última criada.

Nota-se que do ponto de vista que analisa a complexidade pelo número de células $c = n^2$, a complexidade da versão algorithmX implementado é $O(c)$.

2.5. Entrada

A primeira linha da entrada do programa é composta por três inteiros, o primeiro representa o número de linhas e colunas do sudoku, n, o segundo o número de colunas do quadrante j e o terceiro o número de linhas do quadrante i. As n próximas linhas contêm as linhas de uma matriz $n \times n$ que representa o sudoku que deve ser resolvido. Nota-se que 0 representa uma célula do sudoku vazia.

2.6. Saída

A saída do programa consiste em, caso o algoritmo tenha encontrado solução, ele escreve na tela "solução" nas próximas linhas imprime a matriz que representa o sudoku resolvido, caso não encontre solução imprime "sem solução" e nas próximas linhas imprime a matriz original, uma vez que caso o algoritmo aqui proposto não encontre solução ele não retorna nada.

3. Avaliação Experimental

A análise experimental do programa foi feita medindo o tempo de execução individual de cada algoritmo implementado usando a biblioteca crono do c++, esse código não é incluído nos entregáveis. Para isso, foram usadas como entrada o dataset dado e também foi criado um dataset com 10 casos de teste para sudokus de $i \times j$ igual a 2×2, 2×3, 2×4 e 3×3. Para o primeiro dataset, o programa foi rodado 100 vezes para cada arquivo dado, enquanto para o dataset próprio foi rodado 10 para cada arquivo, ou seja, como são 10 arquivos de cada tamanho $i \times j$ especificado, para cada tamanho $i \times j$ de sudoku rodamos 100 vezes e tiramos nossas métricas em cima disso.

Nota-se que o programa não encontrou solução somente para as entradas 885.txt e 994.txt do dataset dado. Para o dataset criado o algoritmo passou em todos os 40 testes.

Por uma questão de simplicidade a complexidade temporal em si, será analisada a partir do dataset criado, uma vez que temos um número maior de instancias e para todas elas o algoritmo encontrou solução. Contudo, a seguir vemos alguns dados referentes aos testes feitos com o dataset dado.

Como na maioria dos casos a complexidade da criação da estrutura de dados domina a complexidade do algoritmo de solução em si, analisaremos dois casos, o caso que considera o tempo de ambos os processos e o caso que analisa o tempo do algoritmo de solução separadamente.

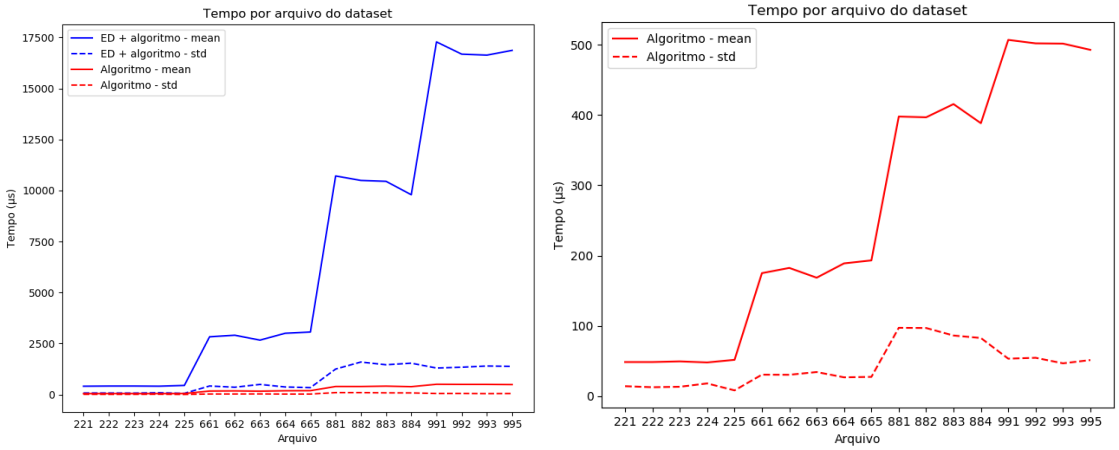
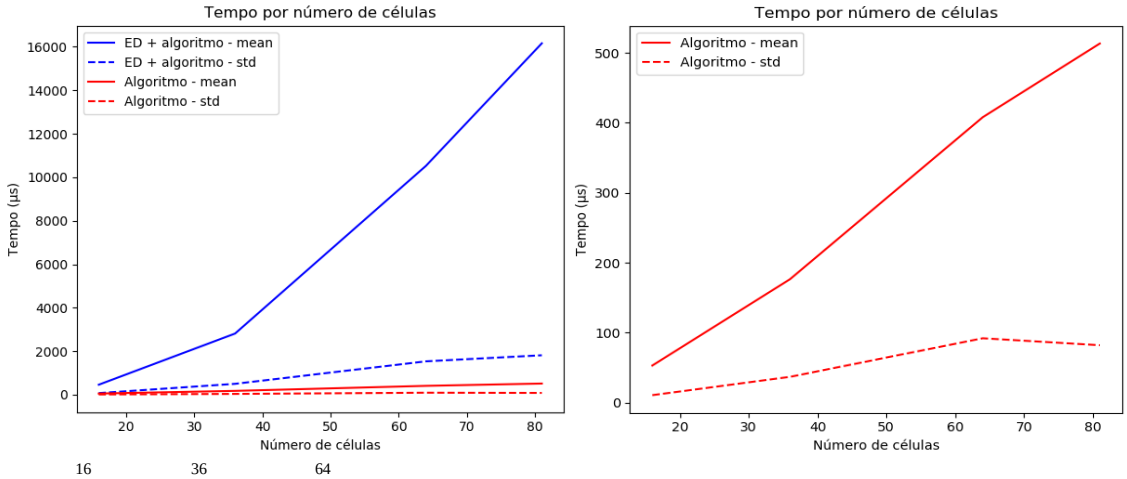


Figura 13: Gráficos de tempo por arquivo do dataset dado. **Fonte:** Elaborado pelo autor.

A princípio, nota-se uma grande diferença entre o tempo da criação da estrutura de dados e o tempo de execução do algoritmo. Pode-se dizer que para um mesmo tamanho de sudoku o tempo gasto para se resolver não variou muito, pois como as instâncias do dataset são todas de sudokus próprios, é possível chegar nas soluções seguindo somente passos lógicos. Nesse gráfico não é completamente, mas é possível verificar através dos dados que o comportamento do algoritmo de solução é de fato linear como previsto.



16 36 64

Figura 14: Gráficos de tempo por número de células por arquivo do dataset criado. **Fonte:** Elaborado pelo autor.

Podemos perceber a tendência linear no número de células, como previsto. Nota-se também uma diferença muito grande entre os tempos para a criação das estruturas de dados e transformação do problema. Fica claro que para melhorar a implementação é preciso efetuar a otimização proposta.

Percebemos também que o desvio padrão cresce muito pouco comparado a sua respectiva média. Podemos concluir a partir disso que para mais células temos menos variação nos tempos medidos, isso quer dizer que as medições são confiáveis, uma vez que o desvio padrão não é muito relevante, principalmente para um número células maiores.

Para executar esses casos de teste e gerar os dados foi criado um programa em Python.

Sabendo que o algoritmo acertou 58 dos 59 casos de teste validos propostos, podemos concluir que a heurística implementada, além de extremamente eficiente acerta mais de 98% dos casos de testes, sendo eles sudokus próprios.

Um teste extra foi feito com o algoritmo, foi dada uma instancia do problema com nenhuma pista e como esperado o algoritmo conseguiu resolver o problema, tirando a restrição de retornar ao fim da arvore de recursão.

Uma outra observação feita acerca da análise de complexidade do algoritmo, é que dado seu comportamento polinomial e dominada pelo algoritmo responsável pela transformação do problema, o pior caso do algoritmo é quando ele deve resolver um sudoku sem nenhuma pista.

3.1. Pergunta: Quais os formatos de tabela do Sudoku (4x4, 9x9, etc.) a heurística adotada obteve melhores soluções?

É interessante notar, que como o algoritmo teve uma taxa de acerto muito alta e somente não conseguiu resolver um problema, dado que o 994.txt está na formatação errada, então não podemos usar essa como uma boa métrica, dessa forma, analisaremos pelo tempo.

Para o algoritmo escolhido e dada a forma que ele foi implementado, obteve-se melhores soluções do ponto de vista de tempo gasto para os sudokus de tamanho 4x4, uma vez que estes são os com menor número de células e o algoritmo apresenta comportamento polinomial.

4. Conclusão

Após a implementação, testes e análise do algoritmo, concluímos que o objetivo do trabalho foi cumprido, um algoritmo muito eficiente foi implementado que resolveu a grande maioria dos casos de testes.

5. Referências Bibliográficas

Ponti, M. Projeto de Algoritmos: paradigmas. Disponível em: < http://wiki.icmc.usp.br/images/f/fe/ICC2_15.ProjetodeAlgoritmos.pdf>. Acesso em 27 de outubro de 2019.

Ziviani, N. Projeto de Algoritmos. 3ª Ed. Cengage, 2004.

Documentação da biblioteca std::list de c++. Disponível em: < http://www.cplusplus.com/reference/list/list/>. Acesso em 27 de outubro de 2019.

KNUTH, Donald E. Dancing links. arXiv preprint cs/0011047, 2000.

HARRYSSON, Mattias; LAESTANDER, Hjalmar. Solving Sudoku efficiently with Dancing Links. 2014.

Sites consultados:

1. https://en.wikipedia.org/wiki/Dancing_Links
2. https://en.wikipedia.org/wiki/Knuth%27s_Algorithm_X
3. http://pi.math.cornell.edu/~mec/Summer2009/Mahmood/More.html
4. https://pt.wikipedia.org/wiki/Cobertura_exata