Pesquisa Operacional

Geração de Sudoku com programação linear e com heurísticas

Lucas R. Raggi
IC - Computer Institute
Federal University of Alagoas
Maceió AL - Brazil
lrr@ic.ufal.br

Nelson G. Neto
IC - Computer Institute
Federal University of Alagoas
Maceió AL - Brazil
ngn@ic.ufal.br

Wagner S. Fontes
IC - Computer Institute
Federal University of Alagoas
Maceió AL - Brazil
wsf@ic.ufal.br

Resumo

Neste artigo são mostradas e comparadas abordagens para o problema NP-Completo de geração de quebra-cabeças Sudoku, utilizando inicialmente um modelo linear com variáveis binárias e posteriormente um processo metaheurístico, neste algumas ideias não relacionadas são reunidas e ajustadas a fim de atingir uma melhor eficiência em comparação àquela alcançada com a programação linear inteira. Tal objetivo é facilmente alcançado, tornando possível a geração de quebra-cabeças 8^2 x 8^2 em tempo hábil (10 a 20 mil segundos); o mesmo não foi atingível com o modelo linear, que ficou limitado a produção de tabuleiros 6^2 x 6^2.

Abstract

In this article we show and compare approaches to the NP-Complete problem of generating Sudoku puzzles, initially using a linear model with binary variables and later a metaheuristic process, after which some unrelated ideas are gathered and adjusted in order to achieve a better efficiency compared to that achieved with linear programming. This goal is easily achieved, making it possible to produce 8^2 x 8^2 puzzles in a timely fashion (10 to 20 thousand seconds); this was not attainable with the linear model, which was limited to the production of 6^2 x 6^2 boards.

Keywords: PO, Sudoku, Eficiência, Heurística

Introdução

Os primeiros problemas resolvidos pela inteligência artificial e pesquisa operacional foram problemas de jogos, brinquedos e enigmas. Mais recentemente Sudoku apareceu no japão e foi disseminado rapidamente, e com o passar do tempo e sua popularização, passou a despertar cada vez mais o interesse da comunidade científica.

O jogo é simples: dada uma grade $n^2 \times n^2$ dividida em quadrados distintos n × n, com alguns de seus campos já preenchidos, para vencer o jogador precisa completar cada célula restante de modo que os três critérios a seguir sejam atendidos:

1. Cada linha de células contém os inteiros 1 até n^2 exatamente uma vez.

2. Cada coluna de células contém os inteiros 1 até n^2 exatamente uma vez

3. Cada quadrado n × n contém os inteiros 1 até n^2 exatamente uma vez.

9	8	1	5	3	2	6	7	4
4	2	5	8	7	6	1	9	3
6	3	7	9	1	4	2	5	8
3	1	4	7	6	8	9	2	5
5	7	9	1	2	3	4	8	6
8	6	2	4	9	5	7	3	1
7	9	3	6	8	1	5	4	2
2	4	6	3	5	7	8	1	9
1	5	8	2	4	9	3	6	7

Figura 1: tabuleiro de Sudoku 3^2x3^2, solucionado.

Neste artigo demonstramos um modelo matemático para geração de instâncias de problemas do sudoku solucionáveis, utilizando o conceito de programação linear inteira (ILP). Introduzimos também um novo método metaheurístico para produção. Os mecanismos que mostraremos também podem ser adaptados para solucionar o Sudoku, mas este não será o foco do nosso artigo.

O processo de produção do tabuleiro assemelha-se ao de resolução, já que para assegurar um estado inicial que permita ao jogador chegar em um estado final que atenda as restrições, é preciso achar ao menos uma solução. Esta condição é o que caracteriza o problema de decisão presente no Sudoku, que foi provado estar contido na classe de problemas NP-Completo em 2006.

Solução Utilizando ILP

Um programa matemático é um conjunto de desigualdades e igualdades definidas em termos de um modelo de variáveis, uma delas definindo a variável objetivo a ser maximizada ou minimizada. Em um modelo linear, todas as restrições são lineares e não pode ser aplicada nenhuma operação não linear sobre uma variável de modelo, ou seja, não pode haver nem o produto entre duas variáveis. Um *integer linear program*(ILP) é um modelo linear com variáveis inteiras.

Baseando-se na modelagem dos artigos [1] e [5]. As variáveis para este modelo estão contidas em uma matriz binária de três dimensões (board), onde cada dimensão tem o tamanho n^2, onde n é o tamanho do Sudoku. Dessa forma, cada célula do Sudoku pode ser mapeada para um array de tamanho n^2, e assim, a escolha de qual número estará naquela célula vai se dar por qual posição do array tem o valor 1, enquanto que em todas as outras constará o valor 0.

As restrições do modelo são as seguintes:

Células: $\forall i \in linhas\ e\ \forall j \in colunas,\ \Sigma_{n \in números} board_{ijn} = 1$

Linhas: $\forall i \in linhas\ e\ \forall n \in números,\ \Sigma_{j \in colunas} board_{ijn} = 1$

Colunas: $\forall i \in colunas\ e\ \forall n \in números,\ \Sigma_{i \in linhas} board_{ijn} = 1$

Blocos: $\forall k \in blocos\ e\ \forall n \in números,\ \Sigma_{i \in (linhas \in k)} \Sigma_{j \in (colunas \in k)} board_{ijn} = 1$

A função objetivo do modelo é maximizar a soma da matriz board.

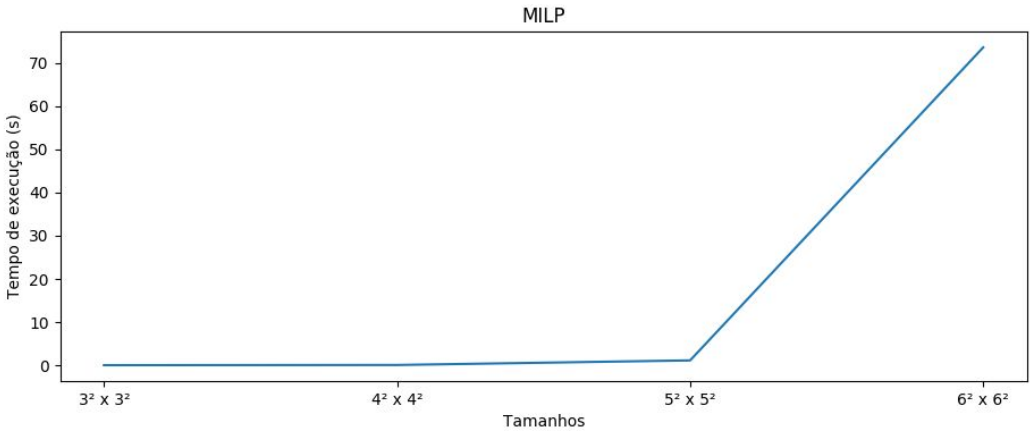

Figura 2: Desempenho do ILP.

Método Metaheurístico

Este método é a combinação de algumas ideias que mostraram bons resultados e que combinadas geram uma grande melhoria em eficiência. Para uma melhor assimilação será mostrado de forma progressiva a evolução do processo e de como os artifícios foram agregados até chegar ao modelo final.

Para evidenciar as melhorias obtidas serão geradas instâncias do problema cada vez maiores ao longo do desenvolvimento. Visto que a geração é um problema NP-Completo, é esperado que a cada incremento do tamanho do tabuleiro o tempo necessário cresça exponencialmente, mesmo sendo sempre possível produzir um resultado consideramos apenas aqueles que podem ser finalizados em um tempo hábil.

Backtracking e otimizações

Em uma solução mais ingênua, genérica e livre de qualquer análise, com a utilização da força bruta para o preenchimento das células foi possível alcançar a geração de um Sudoku 4^2 x 4^2. Visto que na escolha do número para uma célula, é avaliado no máximo todo o conjunto de números possíveis para aquela célula (de acordo com as restrições e o preenchimento atual). Assim qualquer Sudoku gerado é logicamente solucionável.

Desta forma, um possível otimização seria para cada campo do Sudoku passou a ser mantida uma lista contendo os números possíveis, levando em consideração as três regras do Sudoku. Atualizar essa lista a cada passo do algoritmo é extremamente custoso, e calculá-la apenas no início acaba por não trazer nenhuma otimização. Simples testes mostraram que o melhor era atualizar as listas a cada vez que o algoritmo chega no início de uma linha.

Durante a atualização das listas, é possível saber se já existem células que não apresentam nenhum número válido, e dessa forma, voltar na recursão, visto que aquele preenchimento certamente é inválido.

Com uma breve investigação, podemos notar que os blocos da diagonal principal são independentes entre si. Assim, é possível preencher os blocos das diagonais principais previamente, e assim reduzir o número de células que precisam ser preenchidas.

Por último, foi testado diversas ordens de preenchimento: linhas por colunas, colunas por linhas, blocos por blocos, aleatório, ordenado pela quantidade de números possíveis, espiral, etc. Por uma enorme, a melhor ordem de preenchimento foi: linhas por colunas, onde o cache do processador encaixa perfeitamente.

Algoritmo 1 Backtracking

```
1: procedure canFill(cell)
2:     if no more cells then
3:         return true
4:     end if
5:     for number in allowedNumbers do
6:         board[cell] = number
7:         if canFill(nextCell(cell)) then
8:             return true
9:         end if
10:        board[cell] = 0                                  ▷ empty
11:    end for
12:    return false
13: end procedure
14:
15: procedure main
16:    fillDiagonals()
17:    while !canFill() do
18:    end while
19: end procedure
```

Essas análises aceleram em muito o processo de geração, mesmo para o backtracking, permitindo assim que fossem gerados tabuleiros 5^2 x 5^2.

Poda

Verificando que a estratégia anterior possui ainda um grande custo computacional, pois na geração primeiro é preenchido todo tabuleiro com números respeitando a consistência do sudoku, após a geração inicial caso alguma célula seja preenchida em alguma posição que por causa dessa célula nada subsequente a ela será válido, denominaremos esta de campo inválido.

Para evitar que campos inválidos impactem de forma considerável no tempo de execução, no lugar de para cada célula, testar os todos os números possíveis (que é no máximo $\boldsymbol{n^2}$), passamos a tentar apenas $\boldsymbol{x}$ dos números possíveis escolhidos, sendo $\boldsymbol{x}$ menor que o total de números possíveis, resultando em um tempo de execução muito menor, porém não estaríamos mais tentando todas as possibilidades de preenchimento, não garantindo mais uma solução válida para uma execução do algoritmo, dessa forma, caso o algoritmo não encontre uma solução, ele deve começar de novo. Ainda assim, foi percebido que é vantajoso realizar um número menor de tentativas para cada célula, reiniciando caso não haja possibilidade de encontrar uma produção válida.

Após algumas verificações, o melhor resultado médio foi obtido com x = 2, porém mesmo ocorrendo em uma escala bem menor que n^2, com x = 2, as possibilidades permanecem

crescendo exponencialmente. Entretanto para x = 1, o preenchimento será um procedimento aleatório respeitando a consistência das regras do sudoku.

Com isso para que houvesse uma alternância na escolha do valor da variável foi designado um α que representa a probabilidade de x = 2 e (1 - α) a probabilidade de x = 1. Desta forma quanto maior o valor de α, maior a área de busca resultando em menos recomeços para achar um estado consistente, no entanto com um maior tempo de execução. De maneira oposta, quanto menor o valor menor também será o espaço de busca, resultando em mais recomeços para achar um tabuleiro válido, porém com um menor tempo de execução.

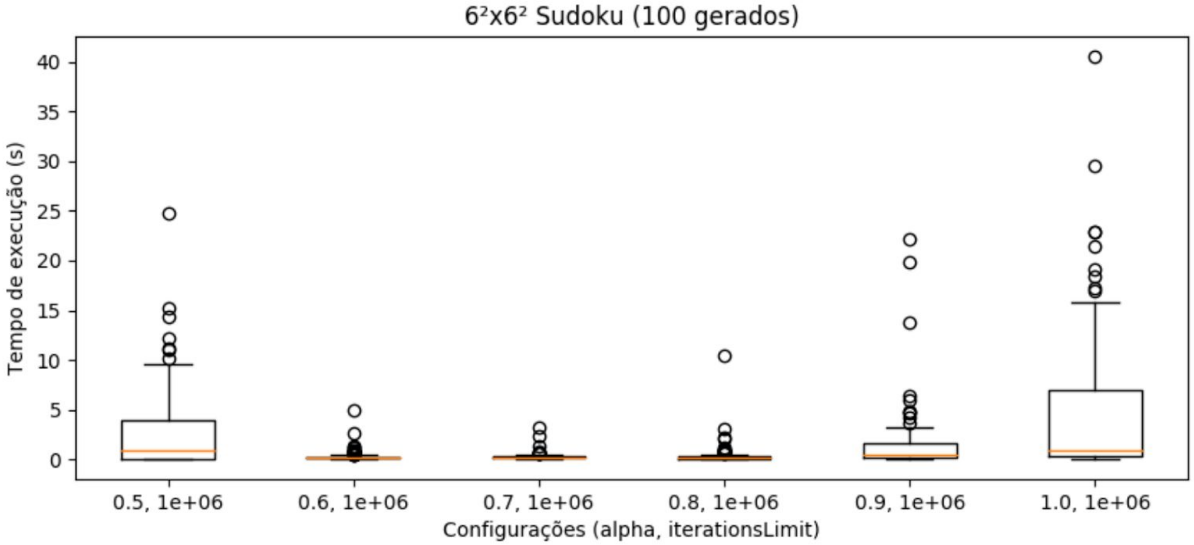

Figura 3: desempenhos obtido com 0.5 ≥ alpha ≤ 1.

Como pode ser visto na tabela acima o α obteve o melhor resultado foi 0.7. Note que os testes do α já foram feitos utilizando tabuleiros 6^2 x 6^2, o tabuleiro 5^2 x 5^2 acabou ficando rápido demais, dificultando a escolha do melhor α.

Algoritmo 2 Big prune

```
1:  procedure canFill(cell)
2:      if no more cells then
3:          return true
4:      end if
5:      if random() < alpha then
6:          testAmout = 2
7:      else
8:          testAmout = 1
9:      end if
10:     for number in allowedNumbers do
11:         board[cell] = number
12:         if canFill(nextCell(cell)) then
13:             return true
14:         end if
15:         board[cell] = 0                                  ▷ empty
16:         testAmount = testAmount - 1
17:         if testAmount == 0 then
18:             break
19:         end if
20:     end for
21:     return false
22: end procedure
```

Resultando em um tempo de execução médio de cerca de nove vezes mais rápido comparado ao backtracking, possibilitando gerar tabuleiros 6^2 x 6^2

Poda com heurística

Como visto no pseudocódigo do método de poda, foi escolhido uma constante α para servir como a probabilidade de escolher tentar dois números diferentes dentre os válidos daquela célula caso a probabilidade falhe, tentar apenas um, essa poda foi feita para que um campo inválido não puna tanto a busca.

É notável que quanto menor o α, menos células inválidas eram geradas no início do preenchimento, e quanto maior, mais possibilidades eram testadas no final, fazendo com que tivesse que recomeçar menos vezes até encontrar um preenchimento válido sem afetar tanto o tempo de execução no final. Então foi testado modificar o comportamento do alfa para ser menor no início do preenchimento e maior no final substituindo o α pelo cálculo: alfa**(filled / total)*. Onde *filled* é o número de células já completadas até o momento, *total* é o número de células no tabuleiro.

Novamente, foram feitos testes de diferentes alfas usando a nova heurística *alpha*(filled / total)* e obtivemos que:

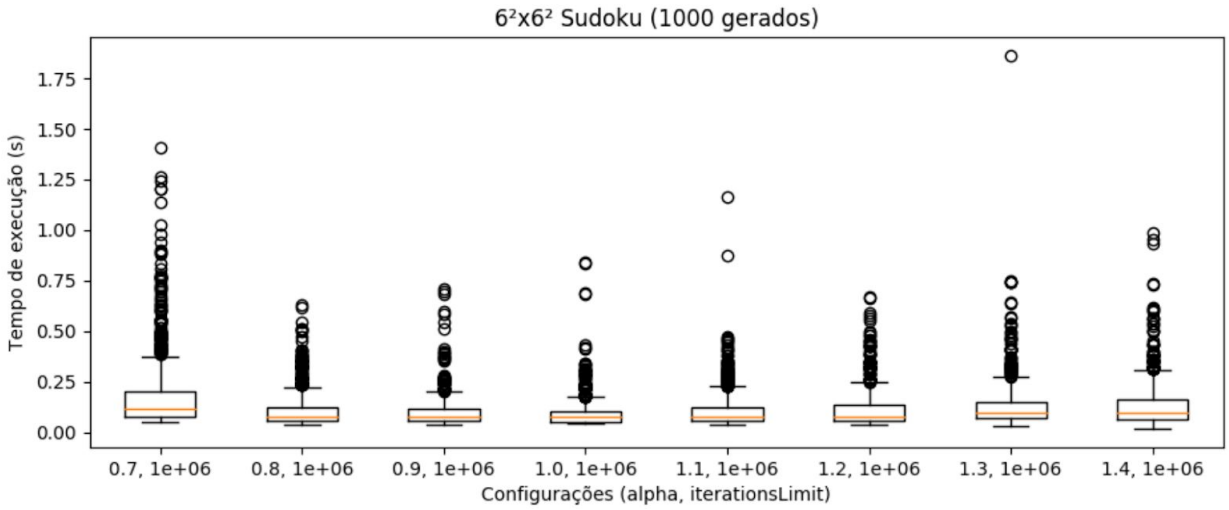

Figura 4: desempenhos obtido com 0.7 ≥ alpha ≤ 1.4

Diferente do método de poda anterior, o melhor valor para alfa foi 1, e possibilitou a geração de tabuleiros 8^2 x 8^2.

Algoritmo 3 Big weighted prune

```
1:  procedure CANFILL(cell)
2:      if no more cells then
3:          return true
4:      end if
5:      if random() < alpha*(filledCells/totalCells) then
6:          testAmount = 2
7:      else
8:          testAmount = 1
9:      end if
10:     for number in allowedNumbers do
11:         board[cell] = number
12:         if canFill(nextCell(cell)) then
13:             return true
14:         end if
15:         board[cell] = 0                                  ▷ empty
16:         testAmount = testAmount - 1
17:         if testAmount == 0 then
18:             break
19:         end if
20:     end for
21:     return false
22: end procedure
```

Por fim, foi investigado o padrão na ordem em que os campos eram preenchidos. Desde o início do processo a sequência adotada seguia a ordem das linhas, assim eram preenchidos todos os elementos da primeira linha, posteriormente a segunda, até a n-ésima linha. Resolvemos então completar de formas diferentes a fim de melhorar ainda mais os resultados. Foram experimentadas outras ordens: por coluna; das regiões mais vazias até a mais coberta; seguindo do centro da matriz até as bordas. No entanto os resultados obtidos confirmaram a superioridade do padrão que já vinha sendo seguido.

Comparação de eficiencia dos métodos utilizados

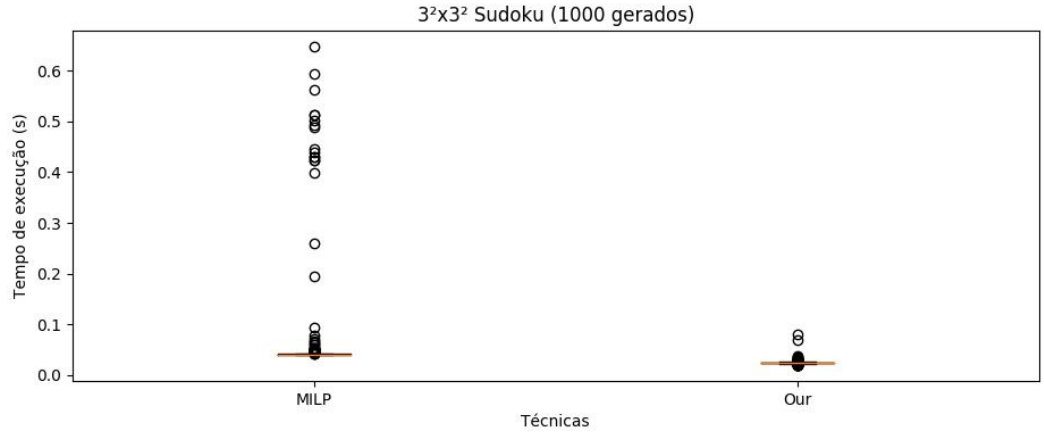

Figura 5: desempenhos do ILP e do método heurístico na geração de tabuleiros 3^2 x 3^2.

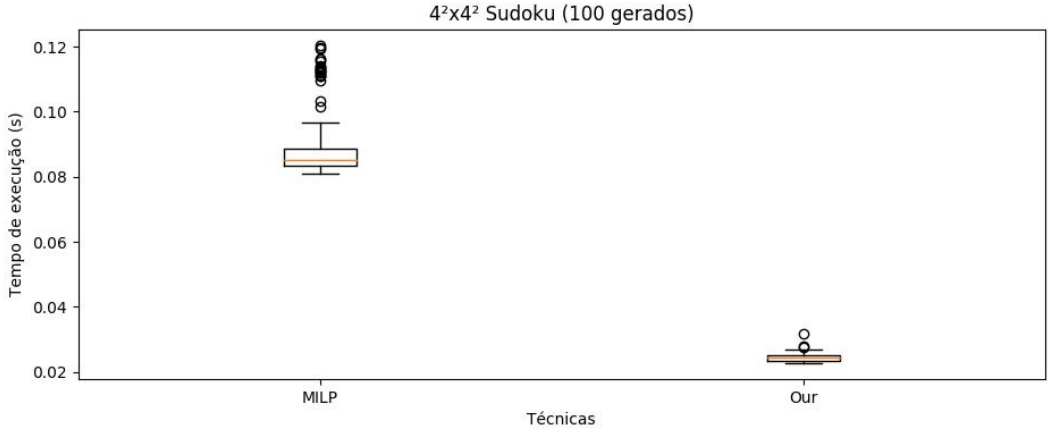

Figura 6: desempenhos do ILP e do método heurístico na geração de tabuleiros 4^2 x 4^2.

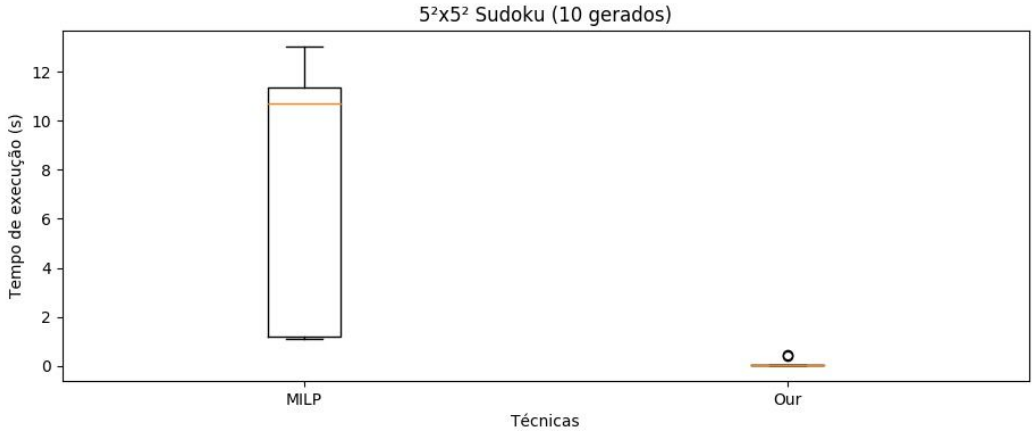

Figura 7: desempenhos do ILP e do método heurístico na geração de tabuleiros 5^2 x 5^2.

Os gráficos acima comparam a performance de tempo do modelo de programação linear com a nossa abordagem poda com heurística, percebe-se que quanto maior o tabuleiro a disparidade entre os dois modelos aumenta, para tabuleiros ainda maiores como 6x6 nossa abordagem leva 1s enquanto o ILP 200 segundos. Demonstrando que uma abordagem mais simplificada consegue ser também muito eficiente.

Outra aspecto perceptível nos gráficos além da diferença de tempo é a dispersão. Mesmo o nosso modelo não testando todas as possibilidades e algumas vezes o algoritmo é necessário reiniciar, a dispersão é mínima, normalmente convergindo para um tempo enquanto o ILP apresenta uma dispersão muito maior e uma maior ocorrência de outliers, como pode ser observado.

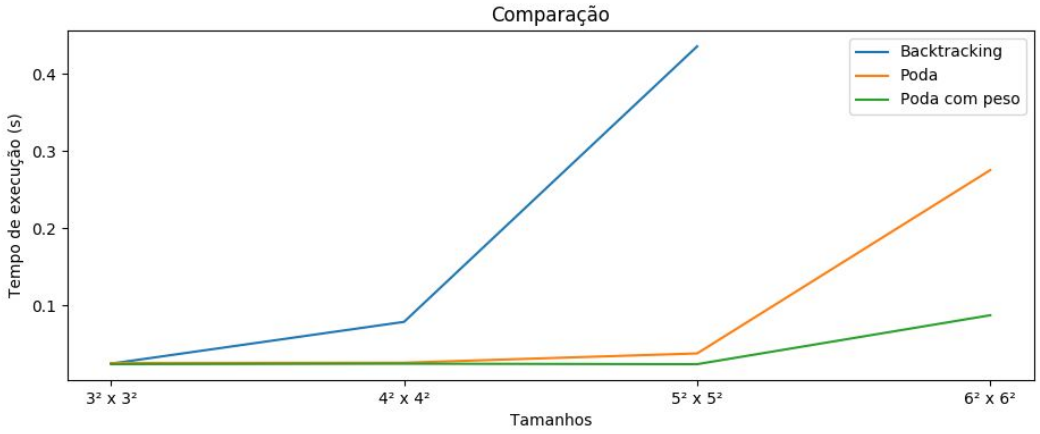

Figura 8: evolução do método heurístico na geração de tabuleiros.

Já neste gráfico comparativo, são consideradas as três etapas do método meta-heurístico apresentado. Como dito anteriormente, ele foi apresentado de forma em que ficasse evidente seu processo evolutivo, assim é razoável que haja tal diferença entre os tempos de execução. A observação mais importante a ser feita é sobre como foi possível reduzir a taxa de crescimento, provocando uma enorme diferença por entre o primeiro estágio e o último.

Conclusão

O Sudoku é um tema bastante popular e explorado no meio acadêmico, com diversas propostas para mecanismos de geração e principalmente de resolução do jogo. Entretanto, as abordagens costumam quase sempre ser mais complexas do que eficaz. Aqui mostramos uma coleção de ideias que individualmente contribuíram para eficiência, e juntas puderam superar com facilidade um modelo popular de programação linear.

Como foi demonstrado o raciocínio desenvolvido até chegar nos melhores resultados, esperamos que essas mesmas ideias sejam reutilizadas em trabalhos futuro em conjunto com outras técnicas já exploradas ou até mesmo novos procedimentos para obtenção de resultados ainda melhores nesse e em outros problemas similares, como: Hitori, Hanjie e Kakuro.

Referências

[1] Andrew C. Bartlett, Timothy P. Chartier, Amy N. Langville, Timothy D. Rankin, "An Integer Programming Model for the Sudoku Problem", The Scientific World Journal, vol. 32.

[2] R. Soto, B. Crawford, C. Galleguillos, E. Monfroy, and F. Paredes, "A hybrid ac3-tabu search algorithm for solving sudoku puzzles," Expert Syst. Appl., vol. 40, no. 15, pp. 5817–5821, 2013.

[3] R. Soto, B. Crawford, C. Galleguillos, E. Monfroy, and F. Paredes, "A Pre-filtered Cuckoo Search Algorithm with Geometric Operators for Solving Sudoku Problems," The Scientific World Journal, vol. Article ID 465359, 2014.

[4] Mantere, T., Koljonen, J., "Solving, rating and generating sudoku puzzles with GA". IEEE Congress on Evolutionary Computation. IEEE, 2007, pp. 1382–1389.

[5] Fonseca, José Barahona, "A Novel MILP Model to Solve Killer Samurai Sudoku Puzzles". IARIA 2016, ISBN: 978-1-61208-478-7.