Utilizando Simulated Annealing para Resolver Sudoku

Lyncoln Sousa de Oliveira

O que é Sudoku?

O Sudoku é um dos jogos do tipo de quebra-cabeças mais conhecidos do mundo, tendo suas raízes na cultura japonesa. A palavra “sudoku” é a junção de duas palavras japonesas, onde “su” significa número e doku “único”. O Sudoku pode ser traduzido como “números únicos”, o seu próprio nome já trás a ideia principal de como deve ser resolvido.

O Sudoku é um jogo baseado na colocação lógica de números. Pode ser encontrado em uma variedade de configurações, sendo o clássico representado por uma matriz 9x9 que é divida em 9 submatrizes 3x3. O objetivo do jogo é a colocação de números de 1 a 9 em cada uma das células vazias da matriz. O quebra-cabeça contém algumas pistas iniciais, que são números inseridos em algumas células, de maneira a permitir uma indução ou dedução dos números em células que estejam vazias. Resolver o problema requer apenas raciocínio lógico e algum tempo. O Sudoku possui apenas 3 regras, que são:

1. Cada linha da matriz 9x9 deve possuir somente números diferentes de 1 a 9.
2. Cada coluna da matriz 9x9 deve possuir somente números diferentes de 1 a 9.
3. Cada submatriz 3x3 deve possuir somente números diferentes de 1 a 9.

Um exemplo clássico de uma configuração do quebra cabeça sudoku é dado pela Figura a seguir, onde os números em verde não podem ser alterados para resolução do quebra-cabeça.

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

Sua solução é dada pela seguinte Figura, onde os números em vermelho são atribuídos pelo jogador.

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

Geralmente o nível de dificuldade do Sudoku pode ser medido pela quantidade de espaços em branco na matriz 9x9, onde uma quantidade maior de espaços vazios representam um conjunto maior de possibilidades de posições de números, implicando assim em uma maior dificuldade de resolução por

parte do jogador. Para garantir uma solução única para um Sudoku, é mostrado em (MCGUIRE; TUGEMANN; CIVARIO, 2012) que é necessário pelo menos 17 números preenchidos originalmente no tabuleiro.

O que é Simulated Annealing?

O Simulated Annealing é um algoritmo de otimização empregado na resolução de problemas complexos de otimização combinatória. Sua inspiração surge do processo físico do recozimento, em que um sólido é aquecido a uma determinada temperatura e, em seguida, gradualmente resfriado. Esse ciclo de aquecimento e resfriamento permite que o sólido, agora derretido, atinja um estado de baixa energia, minimizando sua energia interna (LAARHOVEN P. J. M.; L, 1987).

Ao aplicar esse conceito ao domínio da otimização, o algoritmo de Simulated Annealing opera semelhante ao processo físico. Ele emprega uma estratégia baseada no algoritmo de Metropolis-Hastings para gerar uma sequência de soluções candidatas. Essa sequência é análoga aos diferentes estados físicos do sistema, enquanto uma função de custo avalia a energia associada a cada estado, permitindo a exploração de soluções que não necessariamente levam a uma melhoria imediata, mas são essenciais para escapar de mínimos locais e buscar a solução ótima global do problema (KIRKPATRICK; GELATT; VECCHI, 1983).

O método do Simulated Annealing destaca-se pela sua capacidade de explorar o espaço de soluções de forma mais abrangente, permitindo transições entre estados que podem até mesmo piorar temporariamente a função de custo, mas têm a possibilidade de levar a uma solução ótima ou próxima disso, especialmente em problemas onde a busca por soluções ótimas é dificultada por muitos mínimos ou máximos locais.

O algoritmo de Metropolis-Hastings utilizado para gerar amostras no Simulated Annealing é uma técnica fundamental em MCMC (Monte Carlo e Cadeias de Markov), frequentemente empregada na geração de amostras de uma distribuição de probabilidade em cenários complicados. Este método se destaca ao lidar com distribuições difíceis de se realizar amostragem, utilizando uma estratégia de aceitação/rejeição baseada em simulação para produzir amostras da distribuição desejada. Ao utilizar o Metropolis-Hastings, o processo consiste em iterativamente gerar propostas de novos estados em uma cadeia de Markov com base em estados anteriores, permitindo que a cadeia de Markov explore o espaço das soluções. A cada passo, são feitas propostas de estados, que são aceitos ou rejeitados com base em probabilidade.

No Simulated Annealing, essas transições são regidas pela distribuição estacionária dada pela distribuição de Boltzmann, onde a probabilidade de aceitar ou rejeitar uma mudança de estado é calculada usando o conceito de resfriamento com a temperatura. Essa integração entre Metropolis-Hastings e a distribuição de Boltzmann orienta as transições entre estados, permitindo a exploração probabilística do espaço de soluções em busca de melhores resultados para função de custo. Denote s como estado atual, s’ como um novo estado candidato, f como a função de custo, P como a probabilidade de transição entre o estado s para o s’ e T a temperatura. Utilizando o algoritmo de

Simulated Annealing, temos que gerar cadeias com Metropolis-Hasting utilizando a distribuição de Boltzmann, assim temos para problemas de maximização a probabilidade de transição dada por:

$$P_{s,s'} = \begin{cases} 1 & \text{se } f(s') > f(s) \\ e^{\frac{f(s')-f(s)}{T}} & \text{caso contrário} \end{cases}$$

Como usar Simulated Annealing no Sudoku?

Antes de começar a desenvolver o algoritmo de Simulated Annealing para o Sudoku, vamos pensar em quantas possibilidades de estados existem para a matriz 9x9 apresentada no início do relatório. A primeira submatriz possui 4 espaços em branco, isso significa dizer que eu tenho 4! = 24 maneiras diferentes de só preencher essa matriz. Agora vamos levar em conta também preencher a segunda matriz, pelo princípio fundamental da contagem, temos que para preencher as duas matrizes 4!*5! = 2880 possibilidades. Agora levando em conta todas as submatrizes, temos 4, 5, 8, 6, 5, 6, 8, 5, 4 espaços em branco, seguindo a mesma ideia, vamos ter aproximadamente 8.388*10^23 possíveis soluções. Dentre todas esses estados, somente um representa a solução real, assim torna-se proibitivo utilizar algoritmos puramente aleatórios para resolução.

Para usar o Simulated Annealing no Sudoku primeiro temos que representar uma possível configuração do quebra cabeça como um estado em uma cadeia de Markov. Também temos que pensar em como esse estado pode ser alterado para gerar um novo estado candidato que será avaliado por uma função de custo. Definimos nosso espaço amostral de estados como todos os estados possíveis em que todas as 9 submatrizes 3x3 possuem números de 1 a 9 diferentes entre si. Observe que apesar das submatrizes possuirem números diferentes, as linhas e as colunas da matriz 9x9 principal podem possuir números repetidos.

Também precisamos definir uma função de custo que será máximizada (Apesar do algoritmo do Simulated Annealing ter sido desenvolvido originalmente para minimização, com algumas modificações podemos também resolver problemas de maximização). Definimos nossa função de custo como a soma dos elementos diferentes em cada linha e em cada coluna. Assim, para resolver um sudoku, precisamos que nossa função de custo possua valor 9*9 referentes aos elementos diferentes das linhas, e mais 9*9 referentes aos elementos da coluna, somando um máximo de 162. A Figura a seguir mostra um possível estado da nossa cadeia de markov, que possui função de custo 114.

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

Para o bom funcionamento do algoritmo do Simulated Annealing, é necessário que a cadeia de Markov associada possua a propriedade de irredutibilidade. Uma cadeia de Markov é considerada irreduzível se for possível, a partir de qualquer estado, chegar a qualquer outro estado dentro do sistema com uma probabilidade maior que zero em um número finito de etapas. Em outras palavras, não há “blocos isolados” dentro da cadeia onde você não possa alcançar certos estados a partir de outros. Essa propriedade garante que todos os estados possíveis do espaço amostral vão ser visitados em um número finito de iterações.

Podemos pensar em diversas maneiras de transitar entre estados na nossa cadeia de Markov. Vamos definir a nossa transição de estados da seguinte maneira:

1. Sortei um número inteiro uniforme entre 1 e 9.
2. Selecione a submatriz referente ao número sorteado do passo anterior.
3. Com a submatriz selecionada, sorteie mais dois números inteiros uniformes entre 1 e 9.
4. Se as posições da submatriz relacionadas aos números sorteados no passo anterior puderem ser trocadas, realize a troca. Caso contrário, retorne ao passo 3.

Dessa maneira, é intuitivo pensar que a cadeia de Markov formada por essa transição é irredutível. Definidos o espaço amostral de estados, função de custo e transição de estados, agora podemos utilizar o algoritmo do Simulated Annealing para resolver o Sudoku. Observe o pseudocódigo a seguir.

```
1. atual <- gera_estado_sudoku(tabuleiro)
2. maior_funcao_custo <- funcao_custo(atual)
3. contador  <- 0
```

```
4. maximo_iterações <- valor
5. while(contador < maximo_iterações){
6.   candidato <- transita_novo_estado(atual)
7.   funcao_custo_atual <- funcao_custo(atual)
8.   funcao_custo_candidato <- funcao_custo(candidato)
9.   delta_funcao_custo = funcao_custo_candidato - funcao_custo_atual
10.  if(delta_funcao_custo > 0){
11.    atual <- candidato
12.  } else { if(runif(1) < exp(delta_funcao_custo/temperatura)){
13.      atual <- candidato
14.    }
15.  }
16.  if(funcao_custo(atual) == 162) return(atual)
17.  temperatura = temperatura*0.99999
18.  contador = contador + 1
19. }
```

Na linha 1, é gerado um estado inicial para a cadeia de Markov, representando uma configuração inicial para o problema.

Na linha 2, é avaliada a função de custo desse estado inicial, que determina o quão boa é essa configuração inicial.

As linhas 3 e 4 definem as variáveis de controle do número de iterações para o processo de Simulated Annealing.

Na linha 5, um loop é iniciado para iterar até o valor estipulado pelo usuário.

Na linha 6, um novo estado candidato é gerado a partir do estado atual, representando uma possível solução alternativa.

Nas linhas 7 e 8, são calculadas as funções de custo tanto do estado atual quanto do estado candidato, avaliando a qualidade das soluções.

Na linha 9, é calculada a diferença entre a função de custo do estado candidato e a função de custo do estado atual.

Na linha 10 e 11, é aplicada uma condição: se a função de custo do estado candidato for maior que a do estado atual, o novo estado atual é definido como sendo o estado candidato.

Nas linhas 12 e 13, se a função de custo do estado candidato não for maior que a do estado atual, mas a exponencial da diferença entre elas dividida pela temperatura for maior que um número aleatório uniforme entre 0 e 1, o estado atual também é definido como o estado candidato.

Na linha 16 é verificado se a função de custo alcançou o valor de 162, que seria o máximo, o que significa que o tabuleiro do Sudoku foi resolvido.

Na linha 17, a temperatura é reduzida, e o loop continua. O processo de resfriamento permite ao algoritmo explorar soluções mesmo quando a função de custo é pior, ajudando a evitar mínimos locais durante a busca pela solução ótima.

O algoritmo de aceitação/rejeição de Metropolis-Hastings utilizando a distribuição de Boltzmann mencionado está presente nas linhas 12 e 13, ele assegura a propriedade de estacionariedade da cadeia de Markov. Em outras palavras, após um número suficiente de iterações, a distribuição das probabilidades dos estados convergirá para os mesmos valores, independentemente do número de iterações ou do estado inicial. Essa propriedade de estacionariedade é importante para os algoritmos de Simulated Annealing.

Por fim, temos um algoritmo funcional para resolver quebra-cabeças Sudoku. Vale ressaltar que resolver Sudoku utilizando Simulated Annealing pode não ser a melhor solução dentre os algoritmos de otimização, porém ele oferece uma abordagem interessante para explorar soluções em um espaço de busca utilizando conceitos de probabilidade como cadeias de Markov. Pode acontecer de que em um número finito de passos o algoritmo não consiga encontrar a solução final do Sudoku, e sim uma aproximação.

Avaliação do algoritmo

Agora implementamos o pseudocódigo com a linguagem de programação R (R CORE TEAM, 2023). Vamos verificar se o Simulated Annealing consegue encontrar a solução do Sudoku apresentada no início desse relatório. Escolhemos um número máximo de 100000 iterações e uma temperatura inicial de 0.5. Para implementação real, temos que adicionar alguns detalhes, como por exemplo: garantir que o algoritmo não mude os números inicias dados pelo Sudoku, que no caso chamamos de originais; gerar um estado inicial da nossa cadeia de markov; salvar o estado que resultou em uma melhor função de custo. O Código a seguir apresenta a função principal para a implementação do Simulated Annealing. O código completo, com todas as funções auxiliares utilizadas, está disponível em https://github.com/lyncoln/sa_sudoku.

```
resolve_sudoku <- function(sudoku_matrix, temperatura, max_iter){
    originais <- retorna_originais(sudoku_matrix) # Separa os números inicias dado pelo
        Sudoku
   sudoku_matrix <- preenche_zeros(sudoku_matrix) # Gera o estado inicial da cadeia
   atual <- sudoku_matrix
    metrica_melhor <- funcao_avaliacao(atual) # Calcula a função de custo para o estado
        inicial
   iter_cont <- 0
   while(iter_cont < max_iter){
      candidato <- troca(atual, originais) # Realiza a transição do estado atual para o
        candidato
     atual <- aceitar(atual,candidato,temperatura) # Aceita ou rejeita a transição
     metrica_estado <- funcao_avaliacao(atual) # Calcula a função de custo do estado atual
     iter_cont <- iter_cont + 1
```

```
      if(metrica_estado >= metrica_melhor){
        # Se a função de custo do estado atual for melhor do que a melhor observada, salva o
          melhor estado
        metrica_melhor <- metrica_estado
        melhor_estado <- atual
        if(metrica_melhor==81*2){
          # Se a função de custo for igual a 162, o sudoku foi resolvido
          print(paste0("Sudoku resolvido em ",iter_cont," passos!"))
          replot(melhor_estado,originais) # Plota a matriz 9x9 do sudoku
          return(iter_cont)
        }
      }
      if(iter_cont%%1000 == 0){
        # Printa a função de custo do estado atual, a do melhor estado e temperatura a cada
          1000 iterações
         print(paste0("Melhor métrica: ",metrica_melhor," Metrica atual: ",metrica_estado,"
          Iteração: ",iter_cont, " Temperatura ", temperatura))

      }
      temperatura <- temperatura*0.99999 # Reduz a temperatura
    }
    if(metrica_melhor!=81*2){
      # Se não encontrar a solução, mostra o melhor resultado obtido
        print(paste0("Não foi possível resolver o sudoku em ", max_iter, "passos, melhor
          resultado para métrica obtido foi ",metrica_melhor))
      replot(melhor_estado,originais) # Plota a matriz 9x9 do sudoku
      return(0)
    }

}
vetor <- c(5,3,0,0,7,0,0,0,0,
6,0,0,1,9,5,0,0,0,
0,9,8,0,0,0,0,6,0,
8,0,0,0,6,0,0,0,3,
4,0,0,8,0,3,0,0,1,
7,0,0,0,2,0,0,0,6,
0,6,0,0,0,0,2,8,0,
0,0,0,4,1,9,0,0,5,
0,0,0,0,8,0,0,7,9)

sudoku_matrix <- matrix(vetor, nrow = 9, byrow = TRUE)
resolve_sudoku(sudoku_matrix, 0.5, 100000)
```

```
## [1] "Melhor métrica: 158 Metrica atual: 156 Iteração: 1000 Temperatura 0.495029842421621"
## [1] "Melhor métrica: 160 Metrica atual: 160 Iteração: 2000 Temperatura 0.49010418868505"
## [1] "Melhor métrica: 160 Metrica atual: 154 Iteração: 3000 Temperatura 0.485227546265887"
```

```
## [1] "Melhor métrica: 160 Metrica atual: 154 Iteração: 4000 Temperatura 0.480399427490947"
## [1] "Melhor métrica: 160 Metrica atual: 156 Iteração: 5000 Temperatura 0.475619349539502"
## [1] "Melhor métrica: 160 Metrica atual: 156 Iteração: 6000 Temperatura 0.470886834394994"
## [1] "Melhor métrica: 160 Metrica atual: 150 Iteração: 7000 Temperatura 0.46620140879723"
## [1] "Melhor métrica: 160 Metrica atual: 148 Iteração: 8000 Temperatura 0.461562604195062"
## [1] "Melhor métrica: 160 Metrica atual: 153 Iteração: 9000 Temperatura 0.456969956699524"
## [1] "Melhor métrica: 160 Metrica atual: 154 Iteração: 10000 Temperatura 0.452423007037447"
## [1] "Melhor métrica: 160 Metrica atual: 156 Iteração: 11000 Temperatura 0.447921300505529"
## [1] "Melhor métrica: 160 Metrica atual: 158 Iteração: 12000 Temperatura 0.443464386924863"
## [1] "Melhor métrica: 160 Metrica atual: 154 Iteração: 13000 Temperatura 0.43905182059592"
## [1] "Melhor métrica: 160 Metrica atual: 154 Iteração: 14000 Temperatura 0.434683160253976"
## [1] "Melhor métrica: 160 Metrica atual: 155 Iteração: 15000 Temperatura 0.430357969024989"
## [1] "Melhor métrica: 160 Metrica atual: 157 Iteração: 16000 Temperatura 0.426075814381906"
## [1] "Sudoku resolvido em 16609 passos!"
```

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

```
## [1] 16609
```

Para avaliar o quão bem o algoritmo desenvolvido resolve Sudoku, vamos utilizar uma função auxiliar desenvolvida em R do pacote com nome “sudoku” do repositório oficial da linguagem. Esse pacote conta com uma função chamada “generateSudoku(n)” que gera tabuleiros de Sudoku de forma aleatória com n espaços em branco. Vamos gerar 50 tabuleiros diferentes, com n =

10,15,20,25,30,35,40,45,50,55 e 60 espaços em branco e aplicar o algoritmo para resolução. Os resultados podem ser visto a seguir:

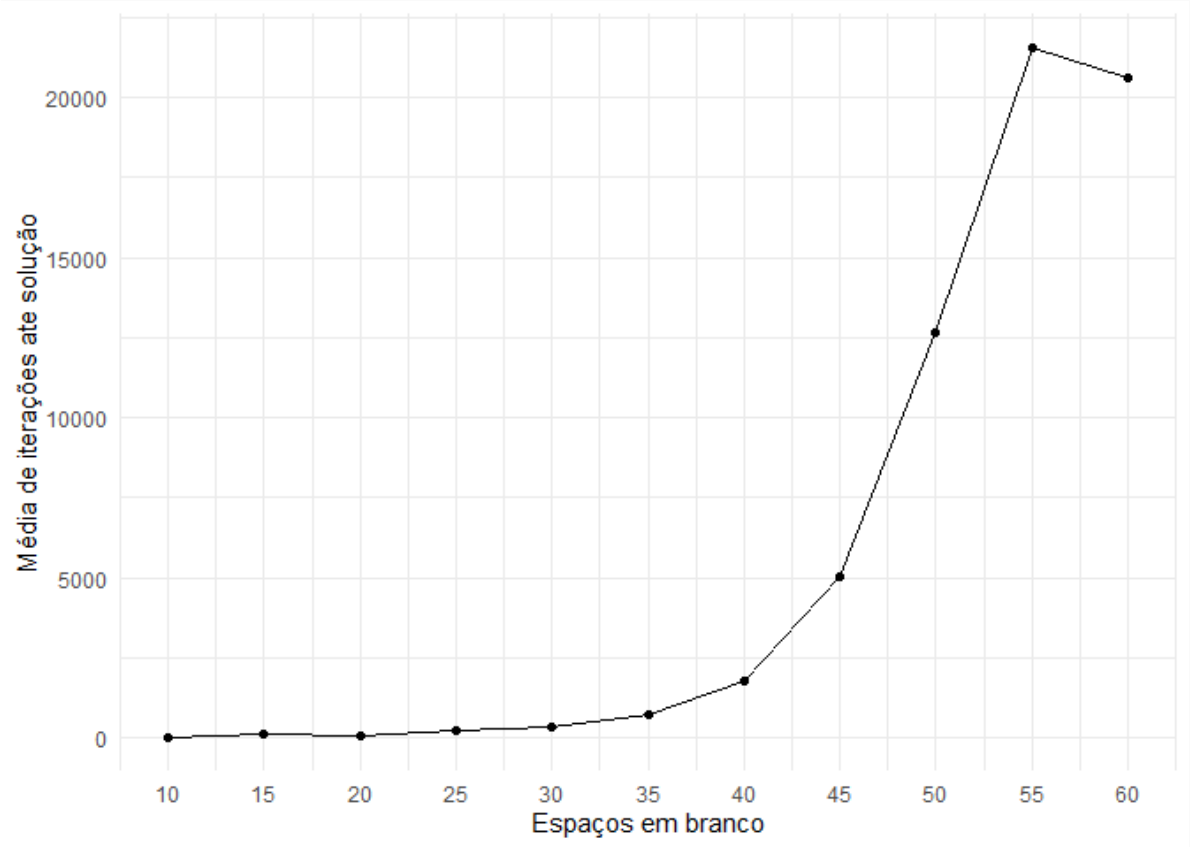
```
resultados = list()
for(n in c(10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60)){
    for(i in 1:50){
    sudoku_matrix = sudoku::generateSudoku(n)
    resolucao = resolve_sudoku(sudoku_matrix, 0.5, 100000)
    resultados[[as.character(n)]] = c(resultados[[as.character(n)]], resolucao)
  }
}
```

```
df_list <- list()
# Gerando a tabela de métricas
for (x in 1:length(resultados)) {
  N = as.numeric(names(resultados[x]))
  solucoes <- sum(resultados[[x]] > 0)
  media <- mean(resultados[[x]][resultados[[x]] > 0])
  desvio <- sd(resultados[[x]][resultados[[x]] > 0])

   df_aux <- data.frame(N = N, Solucoes = solucoes, `Media_Iteracoes` = media, Desvio =
         desvio)
  df_list <- c(df_list, list(df_aux))
}

df_tabela <- do.call(rbind, df_list)

#Estilizando a tabela
styled_table <- df_tabela |>
  kableExtra::kbl() |>
  kableExtra::kable_paper()

styled_table
```

N	Solucoes	Media_Iteracoes	Desvio
10	50	44.50	34.14809
15	50	119.62	98.23605
20	50	94.58	63.82536
25	50	213.90	177.56565
30	50	321.38	243.91224

N	Solucoes	Media_Iteracoes	Desvio
35	50	742.18	507.00182
40	50	1774.78	1724.45329
45	50	5037.28	5569.46390
50	48	12687.90	10795.98206
55	50	21539.78	17225.95508
60	50	20630.64	11442.17284

```
library(ggplot2)

ggplot(df_tabela, aes(x = N, y = `Media_Iteracoes`)) +
  geom_point() +
  geom_line() +
  scale_x_continuous(breaks = seq(10,60,5), labels = seq(10,60,5))+
    labs(x = "Espa\u00E7os em branco", y = "M\u00E9dia de itera\u00E7\u00F5es ate
         solu\u00E7\u00E3o")+
  theme_minimal()
```



Conclusões

Observamos que ao aumentar a quantidade de espaços em branco, a média de iterações necessárias para resolver o quebra-cabeça tende a aumentar consideravelmente. No entanto, essa tendência não se verifica para todos os valores de N. Por exemplo, ao passarmos de 55 para 60 espaços em branco, notamos uma queda na média de iterações necessárias. Além disso, notamos que o algoritmo apresentou dificuldade em apenas 2 quebra-cabeças quando a quantidade de espaços em branco foi de 50. Esse dado ressalta a eficiencia geral do Simulated Annealing na resolução de quebra-cabeças Sudoku. Para melhorar o desempenho do algoritmo, uma alternativa seria explorar técnicas de resfriamento mais sofisticadas dentro do Simulated Annealing ou modificar a estratégia de transição de estados.

Referências

KIRKPATRICK, S.; GELATT, C. D.; VECCHI, M. P. Optimization by Simulated Annealing. **Science**, v. 220, n. 4598, p. 671–680, 1983. Disponível em: <https://science.sciencemag.org/content/220/4598/671>.

LAARHOVEN P. J. M., van; L, A. E. H. **Simulated Annealing: Theory and Applications**. [s.l.] Springer, 1987.

MCGUIRE, G.; TUGEMANN, B.; CIVARIO, G. There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem. **CoRR**, v. abs/1201.0749, 2012. Disponível em: <http://arxiv.org/abs/1201.0749>.

R CORE TEAM. **R: A Language and Environment for Statistical Computing**. Vienna, Austria: R Foundation for Statistical Computing, 2023.