

# Avaliação do desempenho computacional do Projeto Zigue-Zague

Michel Ferracini<sup>1</sup>, André Calisto Souza Medeiros Guedes<sup>1</sup>

<sup>1</sup>Departamento de Ensino – Instituto Federal de Educação, Ciência e Tecnologia de Mato Grosso (IFMT)

Av. Dom Aquino, 1500 – 78.850-000 – Primavera do Leste – MT – Brasil

michel.ferracini@estudante.ifmt.edu.br, andre.calisto@ifmt.edu.br

**Abstract.** Write ...

**Resumo.** Escrever ...

*O resumo (e o abstract) não devem ultrapassar 10 linhas cada, sendo que ambos devem estar na primeira página do artigo.*

## 1. Introdução

Zigue-Zague é um jogo de tabuleiro utilizado em aulas de Matemática para ensinar e treinar as operações básicas, sendo composto pelos seguintes materiais:

- Tabuleiro, conforme mostra a Figura 1 - como apresentado em [Silva and Kodama 2007].
- Três dados de seis faces, numeradas de 1 a 6.
- Um marcador para cada jogador.

Chegada									
2	9	7	4	6	8	7	5	9	
5	4	3	8	9	1	2	5	4	
8	7	6	3	5	4	9	2	7	
6	2	5	7	8	7	6	4	3	
8	7	3	6	4	1	2	5	1	
2	4	8	5	9	7	6	8	5	
7	3	2	1	5	4	5	7	3	
5	8	7	2	8	7	6	9	8	
7	3	2	1	5	4	5	7	3	
2	8	1	8	10	7	9	4	5	
7	5	6	9	4	2	8	1	3	
Partida									

Figura 1. Tabuleiro do Zigue-Zague.

O objetivo do jogo é alcançar a linha de chegada resolvendo expressões numéricas que envolvem adição e subtração. As regras utilizadas, como descritas em [Silva and Kodama 2007] e em [UNESP/IBILCE 2022], são as seguintes:

1. Todos os marcadores são colocados na linha de partida.
2. Os jogadores se revezam (em alguma ordem) lançando os três dados.
3. Em cada lançamento, o jogador deve realizar o cálculo de uma expressão numérica, envolvendo os números que foram obtidos no lançamento e as operações de adição e subtração, em qualquer ordem. Cada jogador deve comunicar as operações realizadas e o resultado aos demais jogadores.
4. No primeiro movimento o jogador deverá colocar seu marcador em uma casa vaga da primeira linha, que contenha o resultado de sua expressão numérica.
5. Nas demais jogadas, o jogador deverá deslocar seu marcador para a próxima linha, de acordo com o resultado de sua expressão numérica, desde que o valor obtido esteja em casa desocupada e vizinha (adjacente) à sua, na diagonal, horizontal ou vertical.
6. Caso não seja possível movimentar seu marcador ou haja erro de cálculo (normalmente detectado pelo adversário), o jogador passa a vez.
7. Vence o jogo quem primeiro alcançar a linha de chegada.

O Projeto Zigue-Zague, por sua vez, se trata de uma pesquisa realizada pela Professora Evelize Aparecida dos Santos Ferracini (coordenadora do projeto), tendo os seguintes participantes: Professora Anne Raphaela Ledesma Cerqueira, Professor Valdiego Siqueira Melo e pelo estudante Michel Ferracini (autor deste artigo), todos do Instituto Federal de Educação, Ciência e Tecnologia de Mato Grosso, Campus Primavera do Leste. O projeto foi submetido e aprovado pelo EDITAL 09/2021 - PROIC - IFMT - PDL - FLUXO CONTÍNUO.

O objetivo geral do projeto é o estabelecimento de uma modelagem matemática (essencialmente probabilística) e computacional (implementada com a linguagem de programação *Python*) do Zigue-Zague, para responder os seguintes questionamentos:

1. Qual o total de caminhos no tabuleiro? (Onde “caminho” é qualquer sequência de 11 (onze) casas do tabuleiro, da primeira até a última linha, que se encadeiam de acordo com as regras do jogo.)
2. Todos os caminhos possuem mesma chance de vitória?
3. Existe alguma estratégia geral vencedora? Isto é, existe alguma estratégia baseada nos caminhos possíveis e na forma como se distribuem pelo tabuleiro que potencializa as chances de vitória?

A pesquisa supracitada obteve os seguintes resultados:

1. Implementou o código para calcular todos os possíveis caminhos no tabuleiro, obtendo como resultado 339 699 caminhos distintos.
2. Definiu um experimento aleatório e um espaço amostral para o jogo, juntamente com o código para mostrar todos os distintos resultados do experimento aleatório e do espaço amostral.
3. Construiu um espaço de probabilidades para o Zigue-Zague e o código para calcular a probabilidade de cada evento de interesse para o jogo, ou seja, a saída dos números que deve-se obter para efetuar uma jogada no tabuleiro.
4. Obteve um modelo para os caminhos no tabuleiro (utilizando variáveis e vetores aleatórios) e uma definição consistente para realizar o cálculo das probabilidades dos caminhos (consistindo, em síntese, na multiplicação das probabilidades de cada casa pertencente a cada caminho).

5. Implementou o código para atribuir probabilidades a cada casa do tabuleiro, bem como a construção de um tipo de “mapa” do tabuleiro.
6. Realizou análise do tabuleiro, de acordo com a probabilidade de vitória de cada caminho, culminando em uma estratégia vencedora (isto é, obteve uma delimitação no tabuleiro de jogo, indicando o conjunto de casas que potencializa as chances de vitória).

O relatório base do Projeto Zigue-Zague foi escrito em *Jupyter Notebook*<sup>1</sup> e está disponível em [https://github.com/michel-ferracini/zigue\\_zague](https://github.com/michel-ferracini/zigue_zague).

### 1.1. Problema de Pesquisa

Apesar do trabalho bem sucedido do grupo de pesquisa, notou-se uma certa lentidão nos códigos desenvolvidos para descobrir uma estratégia vencedora para o jogo.

A lentidão no processamento, mesmo não inviabilizando o estudo proposto, impede uma interação mais dinâmica com os dados obtidos, de modo que a busca por uma otimização do processamento, visando reduzir o tempo de execução dos códigos, torna-se desejável.

### 1.2. Objetivos

Para resolver o problema observado, constitui-se os seguintes objetivos.

- **Geral:** Avaliar o desempenho computacional dos códigos *Python* utilizados no Projeto Zigue-Zague, visando reduzir o tempo de processamento.
- **Específicos:**
  1. Realizar medições de tempo de execução nos códigos utilizados no Projeto Zigue-Zague.
  2. Encontrar hipóteses que expliquem o atual desempenho.
  3. Aplicar possíveis soluções de otimização de desempenho.
  4. Comparar os resultados de desempenho do projeto original com os novos resultados.

### 1.3. Justificativa

O Projeto Zigue-Zague cumpriu os objetivos propostos, porém novos estudos podem ainda ser realizados, como apontado nas considerações finais do mesmo.

Além da existência de novas possibilidades ou interpretações no modelo de tabuleiro estudado, eventualmente utilizando diferentes bases teóricas (tanto na modelagem quanto na análise), há também uma certa variação de modelos de tabuleiros utilizados em aulas de Matemática, de modo que outros estudos baseados no Projeto Zigue-Zague podem seguir estratégias que requeiram um bom desempenho computacional para obter suas conclusões.

Em particular, sempre que for necessário gerar infográficos e realizar simulações ao estudar um tabuleiro de jogo, a questão do desempenho computacional inevitavelmente será colocada em discussão, pois a experiência e testes preliminares sugerem ser estes

---

<sup>1</sup>*Jupyter Notebook* é um tipo de documento interativo para código (em várias linguagens de programação), texto (com ou sem marcação - *HTML* ou *Markdown*), visualizações de dados e outras saídas.

os processos mais lentos no estudo. Usando a função “mágica” `%%time` do *IPython*<sup>2</sup> em cada célula do *notebook* do projeto e variando os dados para observação e análise, os resultados de tempo de processamento variaram de milissegundos até 15 minutos, sugerindo que cargas mais intensas de trabalho podem tornar a obtenção de resultados mais trabalhosa, quando não inviável.

Mesmo para outros estudos, que apenas façam uso das técnicas apresentadas no projeto, mas não necessariamente tratem do jogo Zigue-Zague, torna-se relevante a busca por melhor desempenho computacional.

#### 1.4. Metodologia

Os procedimentos adotados neste trabalho são empíricos, especificamente guiados pela medição da variável tempo.

Usa-se o comando “mágico” `%timeit` do *IPython* e os programas *gprof2dot* e *SnakeViz* para averiguar o tempo de execução em cada iteração de toda função do código, antes e após a implementação das seguintes estratégias:

1. Reestruturação (ou refatoração) de códigos a partir de possíveis alterações em sua lógica.
2. Utilização da opção de computação paralela, em iterações onde seja necessário percorrer todos os itens de um *array*.
3. Criação de um arquivo de persistência das variáveis utilizadas no Zigue-Zague.

O item 1 advém da percepção, apontada na Subseção 1.3, de que procedimentos cujos retornos sejam impressões em tela geram grande demanda de tempo de processamento. Assim, entende-se que a busca por redundâncias no processo e impressões desnecessárias que possam ser substituídas por cálculos ou reduzidas por análise estatística, sejam caminhos viáveis para atingir o objetivo proposto.

A natureza dos códigos do projeto, sempre envolvendo muitas interações com laços *for*, indica que o item 2 é uma alternativa viável, como fundamentado na Seção 2.

Já o item 3, decorre da observação de que a maioria das variáveis calculadas no projeto não sofrem alterações quando geradas novamente, de modo que persisti-las na memória é uma forma de dinamizar a execução do projeto.

Para guiar este estudo, tomou-se como princípio as seguintes duas citações sobre otimização (traduzidas livremente), encontradas em [Hegde 2004], um artigo sobre geração de perfis de programas:

- “Mais pecados de computação são cometidos em nome da eficiência (sem necessariamente alcançá-la) do que por qualquer outro motivo - incluindo estupidez cega.” - - *William A. Wulf*<sup>3</sup>
- “Devemos esquecer as pequenas eficiências, digamos cerca de 97% das vezes: a otimização prematura é a raiz de todos os males.” - - *Donald E. Knuth*<sup>4</sup>

---

<sup>2</sup>*IPython* é um interpretador interativo para várias linguagens de programação, mas especialmente focado em *Python*.

<sup>3</sup>More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason - including blind stupidity - - *William A. Wulf*

<sup>4</sup>We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. - - *Donald E. Knuth*

Buscou-se, neste estudo, seguir a linha de raciocínio expressa nas frases acima, de modo que não aplicou-se estratégias para melhoria de desempenho computacional em todo o código. Utilizou-se comandos mágicos do *IPython* (`%%time` e `%timeit`) e os programas *gprof2dot* e *SnakeViz* para encontrar os gargalos no código e apenas em tais pontos executou-se as estratégias listadas no início desta seção.

## 2. Fundamentação Teórica

Nesta seção, realiza-se um detalhamento do ferramental utilizado neste estudo, como apresentado na Subseção 1.4. Assim, mostra-se a viabilidade teórica do trabalho de avaliação e melhoria de desempenho computacional propostos, indicando as bases tomadas como parâmetro para o mesmo.

### 2.1. Comandos Mágicos do Interpretador IPython

A fonte do texto desta seção, exceto por pequenas adaptações de escrita, é [McKinney 2018].

O *IPython* possui comandos especiais conhecidos como comandos “mágicos”, que servem para facilitar tarefas comuns e permitem controlar facilmente o comportamento do sistema *IPython*. Um comando mágico é qualquer comando prefixado com o símbolo de porcentagem (%), podendo ser visto como um programa de linha de comando para ser executado no sistema *IPython*.

Por exemplo, para conferir o tempo de execução de qualquer instrução *Python*, como uma multiplicação de matrizes, é possível utilizar a função mágica `%timeit`.

A instrução `%timeit`, executa uma instrução várias vezes para calcular um tempo médio de execução do conjunto; é útil para medir o tempo de um código cujo tempo de execução é muito rápido.

### 2.2. Programa gprof2dot

O programa *gprof2dot*, como descrito no *README.md* do projeto no *GitHub* [Fonseca 2022], foi desenvolvido para analisar o código de um programa, indicando quanto tempo cada parte do código precisa para ser executada. Também gera uma série de estatísticas relacionadas às chamadas de função no código.

O programa sintetiza em um grafo as informações de desempenho do código em análise. Cada nó no grafo de saída representa uma função (ou método) e apresenta as seguintes informações: nome da função; porcentagem do tempo de execução gasto na função e em todos os seus filhos; porcentagem do tempo de execução gasto somente na função (entre parêntesis); total de vezes que a função foi chamada (incluindo chamadas recursivas). Cada aresta do grafo representa as chamadas entre duas funções e apresenta as seguintes informações: porcentagem do tempo de execução transferido dos filhos para o pai (se disponível); número de chamadas (de filhos) que a função pai realizou.

### 2.3. Programa SnakeViz

O programa *SnakeViz*, como descrito em [SnakeViz 2022], é um visualizador gráfico baseado em navegador *web* para a saída do módulo *cProfile* do *Python* e uma alternativa

ao uso do módulo *pstats* da biblioteca padrão<sup>5</sup>. O *SnakeViz* fornece informações sobre cada função do sistema em análise, indicando o tempo de execução acumulado para cada uma, o arquivo e a linha do arquivo onde se encontra a função. Além disso, é possível aplicar *zoom* em cada componente do sistema para visualizar as chamadas internas para outras funções.

*SnakeViz* tem dois estilos de visualização, *icicle* (o padrão) e *sunburst*. Em ambos a fração de tempo gasto em uma função é representada pela extensão de um elemento de visualização, seja a largura de um retângulo ou a extensão angular de um arco.

As funções não gastam apenas tempo chamando outras funções, elas também têm seu próprio tempo interno; o *SnakeViz* mostra isso colocando um filho especial em cada nó que representa o tempo interno. Somente funções que chamam outras funções terão isso, funções sem chamadas são totalmente internas.

No estilo de visualização do *icicle*, as funções são representadas por retângulos. Uma função raiz é o retângulo mais alto, com as funções que ela chama abaixo dele, depois as funções que elas chamam abaixo deles e assim por diante. A quantidade de tempo gasto dentro de uma função é representada pela largura do retângulo. Um retângulo que se estende pela maior parte da visualização representa uma função que está ocupando a maior parte do tempo de sua função de chamada, enquanto um retângulo fino representa uma função que quase não está sendo usada.

## 2.4. Módulo *shelve*

O módulo *shelve* faz parte da biblioteca padrão do *Python*. De acordo com a documentação oficial do *Python* [Python 2022], *shelve* possui subclasses e métodos embutidos desenvolvidos para manipular objetos *shelf*. Ainda de acordo com a documentação oficial, *shelf* é um objeto persistente, semelhante a um dicionário. Diferentemente de um banco de dados relacional, os valores (não as chaves) em um objeto *shelf* podem ser essencialmente qualquer objeto *Python*, desde que possa ser manipulado pelo módulo padrão *pickle*.

Essencialmente, o módulo deve ser importado, depois utilizado para abrir um arquivo *shelf* binário, ou criar o arquivo caso o nome chamado ainda não exista. Depois de aberto ou criado, variáveis podem ser gravadas ou lidas do arquivo *shelf* utilizando uma chave do tipo *string*, como em um dicionário.

## 2.5. Computação Paralela

O conceito de computação paralela advém da necessidade existente no mundo moderno de processar grandes quantidades de dados. Segundo [Perkovic 2016],

---

<sup>5</sup>Segundo a documentação oficial do *Python* [Python 2022], *cProfile* fornece perfis determinísticos de programas *Python*, onde perfil é um conjunto de estatísticas que descreve com que frequência e por quanto tempo várias partes do programa são executadas. Essas estatísticas podem ser formatadas em relatórios por meio do módulo *pstats*. O módulo *cProfile* é uma extensão C com sobrecarga razoável que a torna adequada para criação de perfil de programas de longa duração.

“Por várias décadas e até meados da década de 2000, os microprocessadores na maioria dos computadores pessoais tinham um único núcleo (ou seja, unidade de processamento). Isso significava que apenas um programa poderia ser executado de uma só vez nessas máquinas. A partir de meados daquela década, os principais fabricantes de microprocessador, como Intel e AMD, começaram a vender microprocessadores com várias unidades de processamento, normalmente chamados de núcleos (ou cores). Quase todos os computadores pessoais vendidos atualmente e muitos dispositivos sem fio possuem microprocessadores com dois ou mais núcleos.”

Para que o conceito de processamento paralelo seja entendido, é necessário primeiro esclarecer o conceito de processo. Em síntese, um processo é um “programa em execução”. Segundo [Perkovic 2016],

“Quando um programa é executado em um computador, ele é executado em um ‘ambiente’ que registra todas as instruções de programa, variáveis, pilha de programa, o estado da CPU e assim por diante. Esse ‘ambiente’ é criado pelo sistema operacional subjacente para dar suporte à execução do programa. Esse ‘ambiente’ é aquilo que nos referimos como um processo.”

E continua explicando que

“Os computadores modernos realizam multiprocessamento, o que significa que eles podem executar vários programas ou, mais precisamente, múltiplos processos simultaneamente. O termo simultaneamente não significa realmente ‘ao mesmo tempo’. Em uma arquitetura de computador com um microprocessador de único núcleo, somente um processo pode estar realmente sendo executado em determinado momento. O que ele realmente significa neste caso é que, em qualquer ponto no tempo, existem vários processos (programas em execução), um dos quais está realmente usando a CPU e fazendo progresso; os outros processos são interrompidos, esperando que a CPU seja alocada a eles pelo sistema operacional. Em uma arquitetura de computador ‘multicore’, a situação é diferente: vários processos podem verdadeiramente ser executados ao mesmo tempo, em núcleos diferentes.”

Usando a linguagem *Python* é possível dividir a execução de um programa em várias tarefas. Uma possibilidade é utilizar o módulo *multiprocessing* da Biblioteca Padrão para executar as diversas tarefas em paralelo por diferentes núcleos.

### 3. Resultados

Para alcançar os objetivos propostos neste estudo, inicia-se na Subseção 3.1 a avaliação do desempenho computacional do Projeto Zigue-Zague com uma descrição e análise da estrutura dos códigos gerados no projeto.

Em seguida, visando alcançar o primeiro objetivo (realizar medições de tempo de execução nos códigos utilizados no Projeto Zigue-Zague - ver Subseção 1.2), é gerado na Subseção 3.2 um grafo de todo o programa e, também, usa-se o programa *SnakeViz* para gerar uma visualização detalhada de cada parte do mesmo.

Na Subseção 3.3 busca-se encontrar hipóteses que expliquem o atual desempenho dos códigos do projeto, partindo das medições de tempo de execução obtidas na seção anterior.

No decorrer da Subseção 3.4, são aplicadas soluções de otimização de desempenho, de acordo com as hipóteses formuladas na Subseção 3.3.

Finalmente, na Subseção 3.5, os resultados de desempenho do projeto original são comparados com os novos resultados.

### 3.1. Detalhamento do Código

O Projeto Zigue-Zague possui três fases bem delimitadas:

1. modelagem matemática e computacional do tabuleiro de jogo;
2. geração de visualizações, que associam os caminhos de mais altas probabilidades e os de mais baixas com determinadas regiões do tabuleiro, culminando na observação de uma possível estratégia vencedora;
3. confirmação de uma estratégia vencedora a partir de simulações e análise estatística.

Em seguida, detalhes de cada uma destas fases são apresentados.

#### 3.1.1. Modelagem Matemática e Computacional

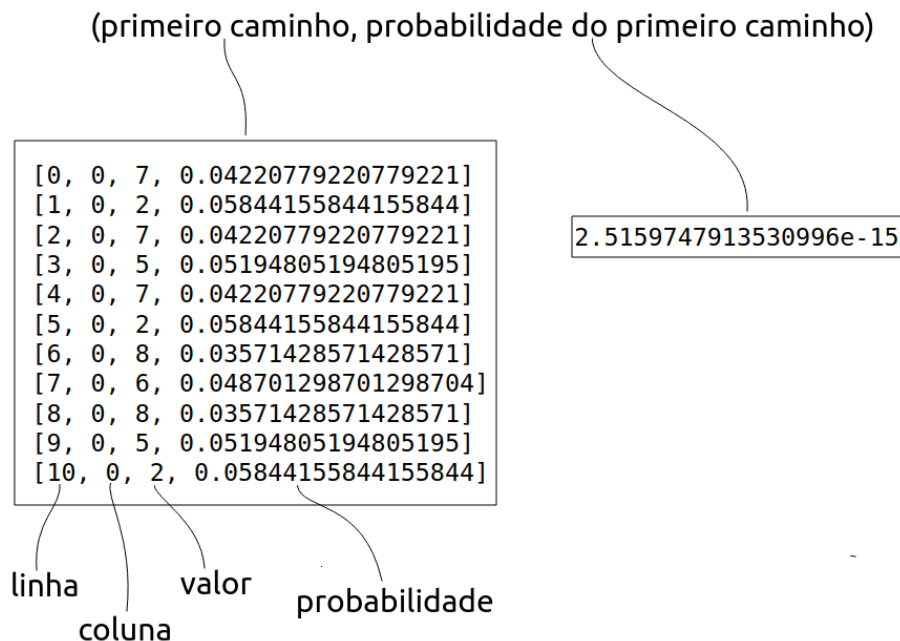
A modelagem matemática e computacional, gerada na fase 1, é sintetizada em uma função (denominada **mapaTabuleiro**) que retorna uma lista (denominada **mapa\_tabuleiro**) contendo todos os 339 699 caminhos do tabuleiro, juntamente com as probabilidades de cada um destes caminhos. Cada elemento da lista é formado por um par - tupla - (**caminho, probabilidade do caminho**). O primeiro elemento de **mapa\_tabuleiro** é o seguinte par:

```
([[0, 0, 7, 0.04220779220779221], [1, 0, 2, 0.05844155844155844],  
 [2, 0, 7, 0.04220779220779221], [3, 0, 5, 0.05194805194805195],  
 [4, 0, 7, 0.04220779220779221], [5, 0, 2, 0.05844155844155844],  
 [6, 0, 8, 0.03571428571428571], [7, 0, 6, 0.048701298701298704],  
 [8, 0, 8, 0.03571428571428571], [9, 0, 5, 0.05194805194805195],  
 [10, 0, 2, 0.05844155844155844]], 2.5159747913530996e-15)
```

Melhor compreensão do elemento acima ilustrado, pode ser obtida na Figura 2. O quadro da esquerda mostra os elementos do primeiro caminho, onde cada elemento é composto por um número entre 0 (zero) e 10 que indica a linha do tabuleiro, um número que pode variar de 0 (zero) a 8 e indica a coluna, um número que pode variar de 1 a

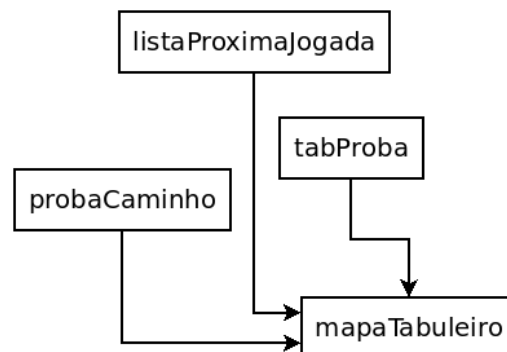


10 e indica o valor referente a casa do tabuleiro (que possui como coordenadas a linha e coluna marcadas nas posições anteriores) e um número entre 0 (zero) e 1 que indica a probabilidade do valor da casa ocorrer em um lançamento de dados. Já o quadro da direita mostra um número entre 0 (zero) e 1 que indica a probabilidade do caminho ocorrer.



**Figura 2. Detalhamento do primeiro elemento do mapeamento do tabuleiro.**

A construção da função **mapaTabuleiro** utiliza diretamente outras três funções: **listaProximaJogada**, **tabProba** e **probaCaminho**, conforme ilustra a Figura 3.



**Figura 3. Construção da função mapaTabuleiro.**

A função **listaProximaJogada** é construída para atender uma particularidade das regras do jogo: quando o jogador se encontra em uma determinada posição no tabuleiro, suas possibilidades de avançar ficam restritas às casas adjacentes da linha superior. Já a função **tabProba** fornece à função **mapaTabuleiro** os valores das probabilidades de cada casa, de acordo com o valor da mesma. Por último, a função **probaCaminho** é utilizada para calcular o valor da probabilidade de cada possível caminho no tabuleiro.

A função **tabProba** também faz uso de funções auxiliares: **probabilidade** e **resultadoExpressoes**. De modo geral, o código para obtenção do mapeamento do tabuleiro faz uso de uma série de funções auxiliares, como mostra a Figura 4, onde **produtoFiltrado** retorna todos os resultados possíveis no lançamento de três dados, **resultadoExpressoes** calcula e retorna os resultados de todas as possíveis expressões numéricas formadas com as possíveis configurações dos dados obtidas (com **produtoFiltrado**), **cardinalidade** retorna a cardinalidade de um evento simples do espaço amostral e **probabilidade** que retorna a chance de um evento ocorrer.

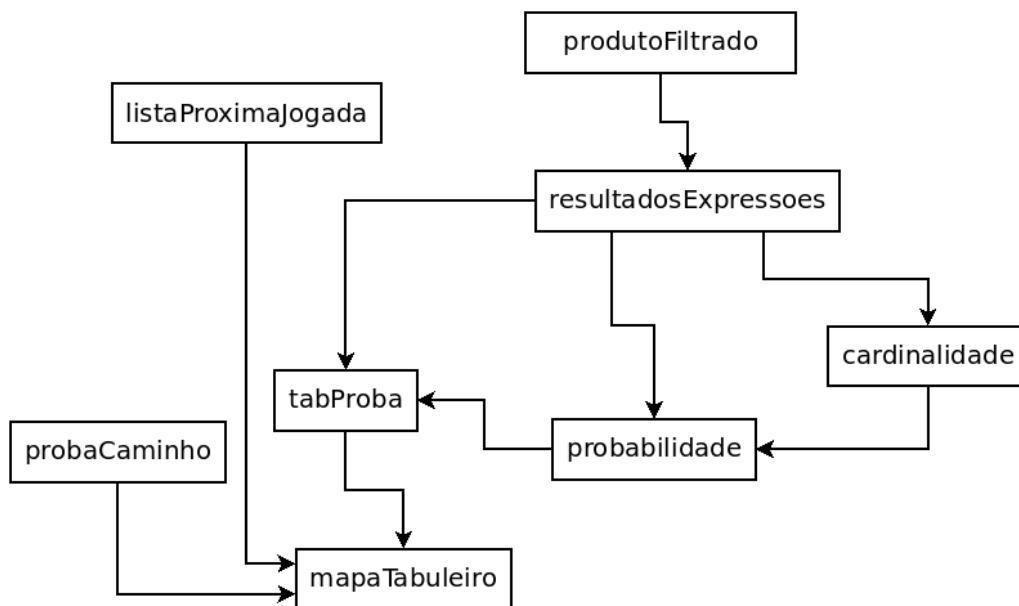
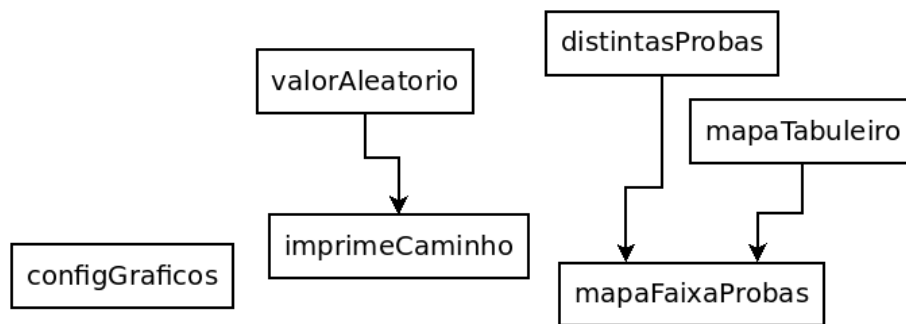


Figura 4. Diagrama mostrando a relação entre as funções utilizadas na modelagem matemática e computacional do tabuleiro.

### 3.1.2. Visualizações

As visualizações são geradas no Projeto Zigue-Zague especificamente pela função **imprimeCaminho**, que é chamada repetidas vezes para imprimir na tela cada caminho contido em qualquer submapa da lista **mapa\_tabuleiro**, obtida na fase 1 (descrita acima). A Figura 5 mostra as funções utilizadas no processo e suas inter-relações.

A função **distintasProbabilidades** percorre o mapeamento do tabuleiro (**mapa\_tabuleiro**) e retorna uma lista contendo apenas os distintos valores das probabilidades dos caminhos do tabuleiro. Já a função **configGraficos** é chamada no processo para criar um esquema de grade que representa o tabuleiro de jogo. A última função da Figura 5, **mapaFaixaProbabilidades**, percorre **mapa\_tabuleiro** comparando a probabilidade de cada caminho com dois valores, sendo o primeiro um valor mínimo de probabilidade e o segundo um valor máximo; quando a probabilidade de ocorrência de um caminho está



**Figura 5. Diagramas das funções utilizadas no processo de visualização.**

entre o mínimo e o máximo informados, o caminho é anexado ao final de uma lista, que é então retornada pela função.

De modo geral, o código para gerar as visualizações utiliza as funções **mapaFaixaProbas**, **imprimeCaminho** e **configGraficos**, seguindo a seguinte estrutura:

- Primeiro a função **mapaFaixaProbas** é chamada duas vezes para criar um submapa inferior (**submapa\_inferior**) dos caminhos que possuem as 2000 menores probabilidades e, depois, para criar um submapa superior (**submapa\_superior**) dos caminhos com as 2000 maiores probabilidades:

```

submapa_inferior = mapaFaixaProbas(
    mapa_tabuleiro,
    distintas_probas,
    0,
    2000)
submapa_superior = mapaFaixaProbas(
    mapa_tabuleiro,
    distintas_probas,
    21076,
    23076)
  
```

- Na sequência um laço *for* percorre **submapa\_superior** e usa a função **imprimeCaminho** para imprimir cada caminho:

```

for caminho in submapa_superior:
    imprimeCaminho(caminho, 1, 'green')
  
```

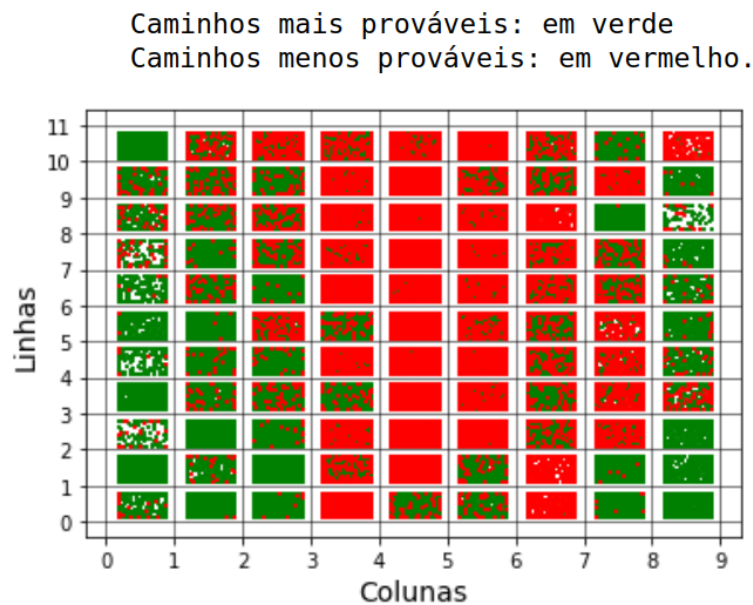
- O mesmo é feito para **submapa\_inferior**:

```

for caminho in submapa_inferior:
    imprimeCaminho(caminho, 1, 'red')
  
```

- Finalmente a função **configGraficos** é chamada para exibir as impressões na tela, como mostra a Figura 6.

A Figura 6 é um importante resultado no projeto Zigue-Zague. Além dessa visualização, uma outra também é gerada, mas apenas invertendo a ordem de impressão



**Figura 6. Primeira visualização do Projeto Zigue-Zague.**

das faixas de probabilidades. As imagens geradas servem para visualizar as regiões com as menores probabilidades de avançar no tabuleiro (indicadas com pontos vermelhos) e as regiões com as mais altas probabilidades de avançar (indicadas com pontos verdes).

### 3.1.3. Confirmação de uma Estratégia Vencedora

No texto do Projeto Zigue-Zague, é descrita a seguinte estratégia:

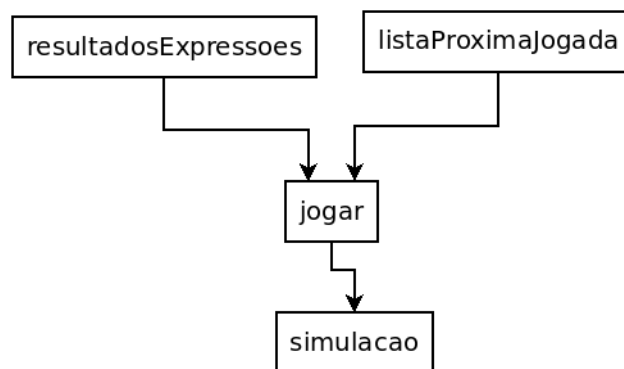
“No tabuleiro, as três colunas da esquerda (1, 2 e 3) constituem a região com maior probabilidade de avançar em um jogo; as três colunas centrais (4, 5 e 6) constituem a região com menor probabilidade; as três colunas da direita (7, 8 e 9) constituem uma região intermediária.

Enfim, jogar pela esquerda potencializa as chances de vencer no Zigue-Zague; a segunda melhor opção é avançar pela direita; a pior região é a central.”

Para confirmar que a região esquerda possui a maior chance de vitória é criada a função **jogar**, que simula as jogadas de um único jogador avançando do início até o fim em um tabuleiro, de acordo com as regras do jogo. A função **jogar** é então chamada repetidamente pela função **simulacao**. A Figura 7 mostra a inter-relação das funções utilizadas para gerar as simulações.

A função **jogar** recebe como parâmetro um tabuleiro reduzido (com 3 colunas), retornando a quantidade de jogadas em que não foi possível avançar por falta de opção. O código passa pelas seguintes etapas:

- Escolhe uma coluna (na faixa determinada para a simulação) aleatoriamente.
- Entra em um laço para percorrer cada linha do tabuleiro.



**Figura 7. Diagrama das funções utilizadas para confirmar uma estratégia vencedora.**

- “Lança” os dados.
- Calcula todas as possíveis expressões para os dados sorteados.
- Verifica as casas da próxima linha para confirmar se existem valores condizentes com as possíveis expressões.
- Se houver possibilidades de avançar, escolhe uma das casas disponíveis e realiza a jogada; caso contrário, contabiliza a tentativa, isto é, adiciona 1 à variável de retorno.

Finalmente, a função **simulacao** recebe como parâmetro o número de simulações desejadas e retorna uma terna - tupla - contendo a média do número de vezes (para cada entrada da terna) em que não houve opções de jogadas em cada região do tabuleiro (esquerda, centro e direita).

Por exemplo, para a chamada **simulacao(2000000)** um possível retorno é a terna (9.020566, 11.190417, 9.6818985), indicando que na região esquerda houve em média 9.020566 jogadas sem opções para avançar, na região central houve em média 11.190417 jogadas e na região direita 9.6818985.

No total, são realizadas 16 milhões de simulações, em etapas de 2 milhões. O código é o seguinte:

```
import numpy as np
dados2mi = np.array(simulacao(2000000))
dados4mi = (dados2mi + np.array(simulacao(2000000)))/2
dados6mi = (dados4mi + np.array(simulacao(2000000)))/2
dados8mi = (dados6mi + np.array(simulacao(2000000)))/2
dados10mi = (dados8mi + np.array(simulacao(2000000)))/2
dados12mi = (dados10mi + np.array(simulacao(2000000)))/2
dados14mi = (dados12mi + np.array(simulacao(2000000)))/2
dados16mi = (dados14mi + np.array(simulacao(2000000)))/2
```

Na segunda linha, a variável **dados2mi** recebe um *array numpy* obtido dos dados de uma simulação de 2 milhões de jogadas. Um valor típico para **dados2mi** é:

```
array([ 9.0251825, 11.186253 ,  9.677728 ])
```

A partir da terceira linha, cada variável (de **dados4mi** até **dados16mi**) consiste na média aritmética da variável da linha anterior com um *array* obtido de uma nova simulação de 2 milhões de jogadas. Em outras palavras, as variáveis acima acumulam as médias da quantidade de jogadas sem avançar em cada região do tabuleiro de jogo.

Após a realização das simulações, os dados são resumidos em um *DataFrame Pandas*<sup>6</sup>, contendo os resultados mostrados na Tabela 1.

**Tabela 1. *DataFrame Pandas* resumindo os dados das simulações.**

	<b>Esquerda</b>	<b>Centro</b>	<b>Direita</b>
<b>2 milhões</b>	9.026930	11.192411	9.676325
<b>4 milhões</b>	9.034997	11.194292	9.673800
<b>6 milhões</b>	9.028357	11.194507	9.681922
<b>8 milhões</b>	9.025284	11.188898	9.676761
<b>10 milhões</b>	9.027534	11.191456	9.673193
<b>12 milhões</b>	9.024702	11.190881	9.678636
<b>14 milhões</b>	9.033064	11.189941	9.681519
<b>16 milhões</b>	9.030639	11.182544	9.674224

A partir dos dados do *DataFrame* é então realizado um estudo estatístico básico, confirmando a estratégia vencedora descrita no início desta seção.

### 3.2. Medições no Tempo de Execução

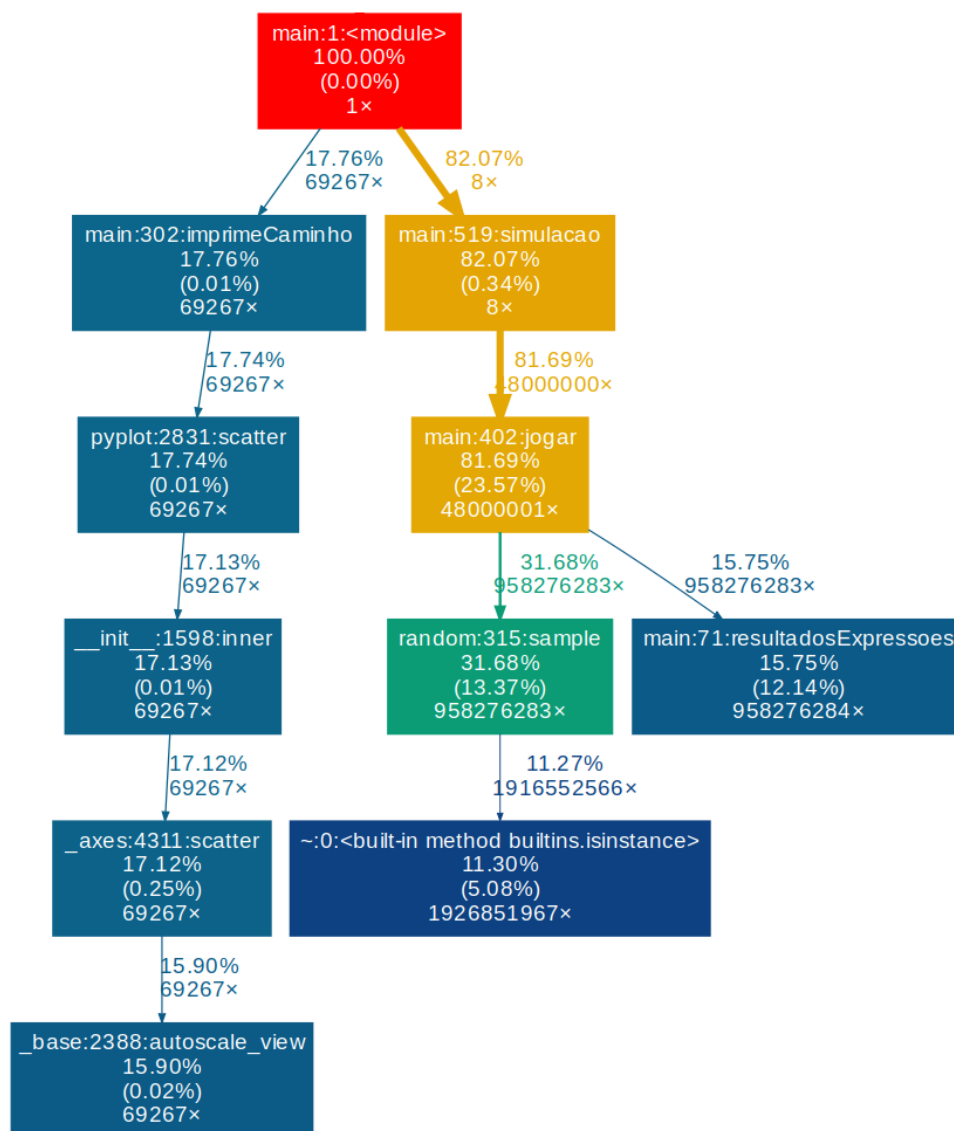
Todos os testes de desempenho computacional dos códigos do Projeto Zigue-Zague foram realizados em um computador com as seguintes características:

- Processador *Intel® Core™ i7-8565U*, 8ª geração;
- 12 GB de memória RAM;
- Armazenamento 256 GB SSD PCIe.

Na geração do grafo (utilizando o programa *gprof2dot*), foi estabelecida uma poda de todos os ramos que utilizam 10% ou menos do tempo de processamento, como ilustrado na Figura 8.

No lado esquerdo do grafo (Figura 8), a função **imprimeCaminho** possui como filho o método **scatter** do módulo *pyplot* da biblioteca *matplotlib*, sendo chamado todas as

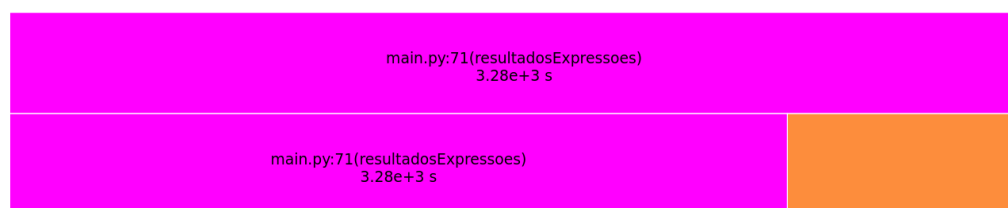
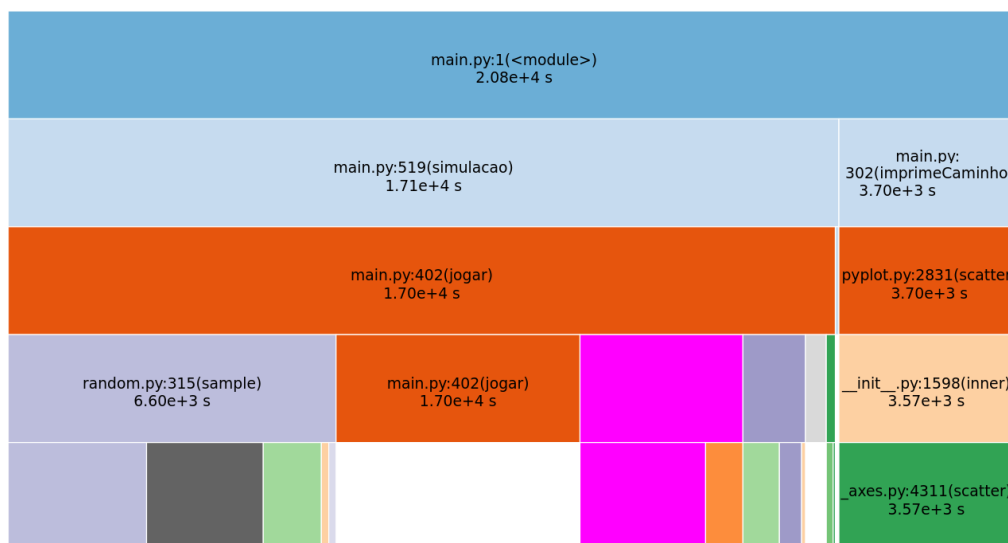
<sup>6</sup>*Pandas* (<http://pandas.pydata.org/>) é uma biblioteca *Python* que oferece estruturas de dados de alto nível e funções, projetadas para fazer com que trabalhar com dados estruturados ou tabulares seja rápido, fácil e expressivo. [McKinney 2018]



**Figura 8. Grafo do programa, com poda para 10% ou menos do tempo de processamento, gerado com o programa *gprof2dot*.**

vezes que se faz necessário imprimir um ponto na geração da visualização dos caminhos de mais alta probabilidades e também dos de mais baixas probabilidades no tabuleiro de jogo. No lado direito, a função **simulação** chama repetidamente a função filha **jogar** que, por sua vez chama repetidamente dois filhos: o método **sample** da biblioteca *random* e a função **resultadosExpressoes**.

Na visualização gerada com o *SnakeViz* (Figura 9), observa-se a mesma configuração de uso do tempo de processamento. Em particular, o tempo de uso de cada função aparece na visualização. Com relação ao grafo na Figura 8, a função **resultadosExpressoes** é a única que não aparece, mas aplicando *zoom* torna-se possível observar essa informação, como mostra a Figura 10.



O grafo foi utilizado apenas para permitir melhor visualização da relação entre as funções, enquanto os dados de tempo de processamento (e, conseqüentemente, do percentual) utilizados neste artigo são provenientes da visualização gerada pelo *SnakeViz*. O resumo destas informações do programa são mostrados na Tabela 2, onde a coluna **Função** estabelece o nome da função chamada, a coluna **Número de chamadas** mostra a quantidade de vezes que uma função foi chamada durante toda a execução do programa, a coluna **Tempo acumulado** exibe a soma de todas as chamadas de uma função e a coluna **Percentual** fornece o percentual de tempo gasto na execução de determinada função com relação ao tempo total de execução, correspondente à função **main**.

Da análise realizada nesta seção, conclui-se que as funções listadas na Tabela 2 são as mais relevantes para o desempenho dos códigos no projeto Zigue-Zague. O tempo total de execução do código foi igual a  $2,08 \cdot 10^4$  segundos, o que corresponde a 5 horas, 46 minutos e 40 segundos. Na próxima seção, esta tabela direciona fortemente a busca por hipóteses que expliquem o desempenho dos códigos do projeto.



**Tabela 2. Resumo da medição dos tempos de execução do programa.**

	Função	Número de chamadas	Tempo acumulado (segundos)	Percentual
1	<b>main</b>	1	$2,08 \cdot 10^4$	100
1.1	<b>simulacao</b>	8	$1,71 \cdot 10^4$	82,07
1.1.1	<b>jogar</b>	48000001	$1,70 \cdot 10^4$	81,69
1.1.1.1	<b>sample</b>	958276283	$6,60 \cdot 10^3$	31,68
1.1.1.2	<b>resultadosExpressoes</b>	958276284	$3,28 \cdot 10^3$	15,75
1.2	<b>imprimeCaminho</b>	69267	$3,70 \cdot 10^3$	17,76
1.2.1	<b>scatter</b>	69267	$3,70 \cdot 10^3$	17,74

### 3.3. Formulação de Hipóteses

Sugeriu-se na Subseção 1.3, a partir de testes preliminares, que os principais gargalos do código do Zigue-Zague estavam na geração de infográficos e na realização de simulações. Os dados na Tabela 2 confirmam essa percepção, já que as funções **simulacao** e **imprimeCaminho** ocupam juntas 99,83% do tempo de processamento do código. Portanto, o foco da análise do desempenho computacional é predominantemente concentrado nestas duas funções, nas subseções 3.3.1 e 3.3.2.

Sem perder o foco nos gargalos citados, uma terceira percepção é também discutida na Subseção 3.3.3: a desnecessária repetição de processamento de funções. Deve-se, portanto, ser discutida alguma estratégia destinada a eliminar estas eventuais repetições que, em um longo período de tempo, pode acumular perda de processamento.

#### 3.3.1. Função **imprimeCaminho**

Essa função possui a seguinte estrutura:

```
def imprimeCaminho(par, tamanho, cor):
    coord_lin = []
    coord_col = []
    caminho = par[0]
    for posicao in caminho:
        coord_lin.append(posicao[0] + valorAleatorio())
        coord_col.append(posicao[1] + valorAleatorio())
    plt.scatter(coord_col, coord_lin, color=cor, marker='s', s=tamanho)
```

O parâmetro **par** na assinatura da função consiste em qualquer um dos 339699 elementos - do tipo (**caminho, probabilidade do caminho**) - retornados pela função **mapaTabuleiro**, como detalhado na Subseção 3.1.1. Assim, **caminho = par[0]** é utilizado no laço *for* para criar duas listas de coordenadas de impressão, que são

utilizadas como argumentos no método **scatter** da biblioteca *pyplot* - escrita como *plt*. Os parâmetros **tamanho** e **cor** são utilizados apenas para especificações de impressão.

Voltando à Tabela 2, nota-se que cada uma das 69267 chamadas de **imprimeCaminho** correspondem a exatamente uma chamada do método **scatter**, sendo que este último consiste em 17,74% do tempo total de processamento, isto é, as chamadas de **scatter** são responsáveis pela lentidão da função analisada.

Analisando novamente a Subseção 3.1.2, nota-se que **imprimeCaminho** e, consequentemente, **scatter** são chamados repetidamente com o objetivo de gerar cada ponto da visualização da Figura 6 (e de uma outra similar omitida neste artigo).

É importante salientar que o objetivo de tal visualização (de acordo com a documentação do Projeto Zigue-Zague) consistia em estabelecer as regiões do tabuleiro de jogo com as mais baixas probabilidades (representadas pelos pontos vermelhos na visualização) e as regiões com as mais altas probabilidades (pontos verdes). De modo geral, quanto maior a razão entre os pontos verdes e vermelhos em uma casa do tabuleiro, maior seria a probabilidade daquela casa ocorrer em uma jogada. Em síntese, a chance de vitória em uma região do tabuleiro é analisada visualmente pela razão entre a quantidade de pontos de cada cor, verdes e vermelhos, em cada casa da região.

Portanto, em posse dos dados que mostram alto custo computacional na impressão dos pontos que geram cada visualização e da observação do parágrafo anterior de que a razão entre as quantidades de pontos é o parâmetro mais relevante para os objetivos do Projeto Zigue-Zague, uma proposta para otimização do desempenho do código na geração de visualizações é a troca de impressões com o método **scatter** por cálculos das razões entre os pontos em verde e em vermelho, realizando-se apenas uma impressão de pontos ao final do processo, com a formação de um “mapa de calor” do tabuleiro, onde as casas com maior razão entre pontos verdes e vermelhos são representadas por um único ponto impresso em cor “mais quente”, enquanto no extremo oposto, onde a razão entre pontos verdes e vermelhos é mais baixa, o ponto impresso possui cor “mais fria”.

### 3.3.2. Função simulacao

O gargalo mais significativo no desempenho dos códigos no Projeto Zigue-Zague é a função **simulacao**, que chama interna e repetidamente a função **jogar**. De acordo com a Tabela 2, a função **simulacao** é chamada 8 vezes com argumento 2000000 (dois milhões), ou seja, no total o Projeto Zigue-Zague realiza 16 milhões de simulações para confirmar e delimitar regiões no tabuleiro de jogo, confirmando uma estratégia vencedora.

Utilizando a função mágica *%%time* do *iPython*, isoladamente em uma dessas

simulações, obtém-se um tempo de execução de aproximadamente 15 minutos, isto é, para as 16 milhões de simulações, espera-se um tempo de pelo menos 4 horas. Segundo a Tabela 2, o tempo de processamento ocupado pelas simulações foi igual a  $1,71 \cdot 10^4$  segundos (4 horas e 45 minutos).

Assim, considerando-se a natural possibilidade de execução independente do código de cada uma das simulações, uma possível solução para melhorar o tempo de processamento consiste em usar computação paralela, como descrito na Subseção 2.5.

### 3.3.3. Repetição Desnecessária de Processamento

Avaliar o que ocorre nos gargalos, como nas subseções anteriores, é claramente uma estratégia válida, porém deve-se também pensar na natureza do projeto em execução, onde uma perda acumulada de tempo de processamento inevitavelmente ocorre. De fato, o Projeto Zigue-Zague é um típico “projeto de exploração” em *Jupyter Notebook*, de modo que células de código são repetidamente executadas.

Quando um processo de descoberta está em andamento, é natural e desejável executar repetidas vezes uma célula de código (isso faz parte da essência de um *Jupyter Notebook*). Porém, conforme um estudo dessa natureza avança e os resultados tornam-se parcialmente consolidados, a reexecução de certas células de código (mesmo que não demore mais que um minuto), pode causar o que aqui será denominado “lentidão acumulada”.

Usando a função mágica `%%timeit` do *Ipypthon*, pode-se identificar duas principais funções cujas respectivas médias de tempo de processamento sugerem potencial lentidão acumulada: **distintasProbas** (em média 17,1 segundos para ser executada) e **subMapaAleatorio** (em média 53,5 segundos para que suas execuções gerem a lista **lista\_mapas\_razoes**). No total, é necessário somente 1 minuto, 10 segundos e 6 décimos para a execução das duas funções, porém quando executadas dezenas de vezes, ocorrerá a lentidão acumulada já mencionada. Uma possível solução para esse potencial problema é, portanto, o emprego de persistência de dados, como proposto na Subseção 1.4.

### 3.4. Aplicação de Possíveis Soluções

Seguindo a discussão da seção anterior, são apresentadas nesta seção três soluções para melhorar o tempo de execução nos códigos do Projeto Zigue-Zague:

- Substituição das impressões, que utilizam a função **imprimeCaminho**, por cálculos de razões entre pontos de mais alta e de mais baixa probabilidades e geração de um “mapa de calor” utilizando as razões calculadas.

- Utilização de computação paralela na execução das chamadas da função **simulacao**.
- Criação, através do módulo *shelve*, de um objeto *shelf* para persistir as variáveis, evitando lentidão acumulada.

### 3.4.1. Cálculo de Razões e Mapa de Calor

A ideia é simplesmente criar um mapeamento do tabuleiro a partir da contagem do número de caminhos de mais altas probabilidades pelos de mais baixas probabilidades, que passam por cada casa. Entretanto, há um problema inicial: considerando-se, por exemplo, as 2000 maiores probabilidades, constata-se que há 27624 caminhos nesta faixa de valores, enquanto para as 2000 menores probabilidades há apenas 7008 caminhos. Logo, tem-se que selecionar de forma aleatória os caminhos nestas faixas e comparar as razões entre quantidades iguais de caminhos.

A função **subMapaAleatorio** abaixo resolve esse problema inicial, percorrendo uma lista de caminhos em uma faixa de probabilidades e retornando uma lista que contém uma quantidade aleatória de caminhos:

```
def subMapaAleatorio(submapa, quantidade):
    lista = []
    if len(submapa) >= quantidade:
        for caminho in rd.sample(submapa, quantidade):
            lista.append(caminho)
    return lista
```

Com a função acima, pode-se gerar um submapa inferior e um superior com quantidades iguais de caminhos, escolhida aleatoriamente. A quantidade (segundo parâmetro da função **subMapaAleatorio**) será, na prática, estabelecida pela função **total**:

```
def total(submapa_inferior, submapa_superior):
    linf = len(submapa_inferior)
    lsup = len(submapa_superior)
    if linf < lsup:
        return linf
    return lsup
```

A função **mapaRazoes** pode então ser definida como segue:

```
def mapaRazoes(submapa_inferior, submapa_superior):
    tab_razoes = [[[1, 1] for _ in range(9)] for _ in range(11)]
    razoes = [[1 for _ in range(9)] for _ in range(11)]
    for caminho_prob in submapa_inferior:
        caminho = caminho_prob[0]
```

```

for casa in caminho:
    lin = casa[0]
    col = casa[1]
    tab_razoes[lin][col][1] += 1
for caminho_prob in submapa_superior:
    caminho = caminho_prob[0]
    for casa in caminho:
        lin = casa[0]
        col = casa[1]
        tab_razoes[lin][col][0] += 1
for l in range(11):
    for c in range(9):
        razoes[l][c] = tab_razoes[l][c][0] / tab_razoes[l][c][1]
return razoes

```

No código acima, o primeiro laço *for* conta no submapa inferior, passado como parâmetro, todas as ocorrências de pontos de tal submapa em cada casa do tabuleiro. No laço *for* que percorre o submapa superior, também é realizada a contagem dos pontos. Finalmente, o mapa de razões é preenchido para cada posição no tabuleiro pelas razões, sendo esta lista de razões o retorno da função.

Todos os mapas de razões são então gerados e guardados em uma lista, bastando calcular as médias das razões por casa do tabuleiro, o que é feito com a função **mediasRazoes** a seguir:

```

def mediasRazoes(lista_mapas_razoes):
    tam = len(lista_mapas_razoes)
    medias = [[0 for _ in range(9)] for _ in range(11)]
    for mapa_razoes in lista_mapas_razoes:
        for i in range(11):
            for j in range(9):
                medias[i][j] += mapa_razoes[i][j]
    for i in range(11):
        for j in range(9):
            medias[i][j] = medias[i][j] / tam
    return medias

```

Finalmente, outras duas funções são definidas: **desviosRazoes** e **mediasComDesvio**. A primeira retorna uma lista do desvio em valor absoluto da média por casa do tabuleiro; a segunda retorna para cada casa do tabuleiro a média somada com o desvio na casa ou com o desvio subtraído da média. Os códigos destas, semelhantes ao código de **mediasRazoes**, foram aqui omitidos.

Essas funções serão implementadas para retornar quatro listas: **medias**, **desvios**, **medias\_inf** e **medias\_sup**; essas listas servem como mapas do tabuleiro e são em seguida

utilizadas para obter os já citados “mapas de calor”. A sequência a seguir mostra como as listas que mapeiam o tabuleiro são obtidas.

- Verifica qual submapa possui menor número de elementos para imprimir a mesma quantidade:

```
TOTAL = total(submapa_inferior, submapa_superior)
```

- Gera diversos mapas de razões do tabuleiro e guarda na lista de mapas de razões:

```
lista_mapas_razoes = []
for _ in range(2000):
    submapa_aleatorio_superior
        = subMapaAleatorio(submapa_superior, TOTAL)
    submapa_aleatorio_inferior
        = subMapaAleatorio(submapa_inferior, TOTAL)
    razoes = mapaRazoes(submapa_aleatorio_inferior,
                        submapa_aleatorio_superior)
    lista_mapas_razoes.append(razoes)
```

- Mapa com as médias (por casa) das razões na lista de mapas de razões:

```
medias = mediasRazoes(lista_mapas_razoes)
```

- Mapa com os desvios (por casa) das médias das razões na lista de mapas de razões:

```
desvios = desviosRazoes(lista_mapas_razoes, medias)
```

- Médias menos os desvios (indicado pelo False):

```
medias_inf = mediasComDesvio(medias, desvios, False)
```

- Médias mais os desvios (indicado pelo True):

```
medias_sup = mediasComDesvio(medias, desvios, True)
```

Finalmente, para gerar as visualizações, usa-se o método *pcolor* do pacote *pyplot* (da biblioteca *matplotlib*), que recebe uma *array* numérico e retorna um tipo de mapa de calor. Na impressão gerada, quanto maior for a razão, mais escura será a representação gráfica da casa. O código para gerar as visualizações é o seguinte:

```
fig = plt.figure()
ax1 = fig.add_subplot(1, 3, 1) # Infografico das medias inferiores.
ax2 = fig.add_subplot(1, 3, 2) # Infografico das medias.
ax3 = fig.add_subplot(1, 3, 3) # Infografico das medias superiores.

VMAX = 10 # Valor maximo para a normalizacao.

# Coloracao do infografico das medias inferiores:
im = ax1.pcolor(medias_inf, cmap="viridis_r", vmin=0, vmax=VMAX)
ax1.set_title('Medias Inferiores\n', fontsize=18)
```

```

# Coloracao do infografico das medias:
ax2.pcolor(medias, cmap="viridis_r", vmin=0, vmax=VMAX)
ax2.set_title('Medias\n', fontsize=18)

# Coloracao do infografico das medias superiores:
ax3.pcolor(medias_sup, cmap="viridis_r", vmin=0, vmax=VMAX)
ax3.set_title('Medias Superiores\n', fontsize=18)

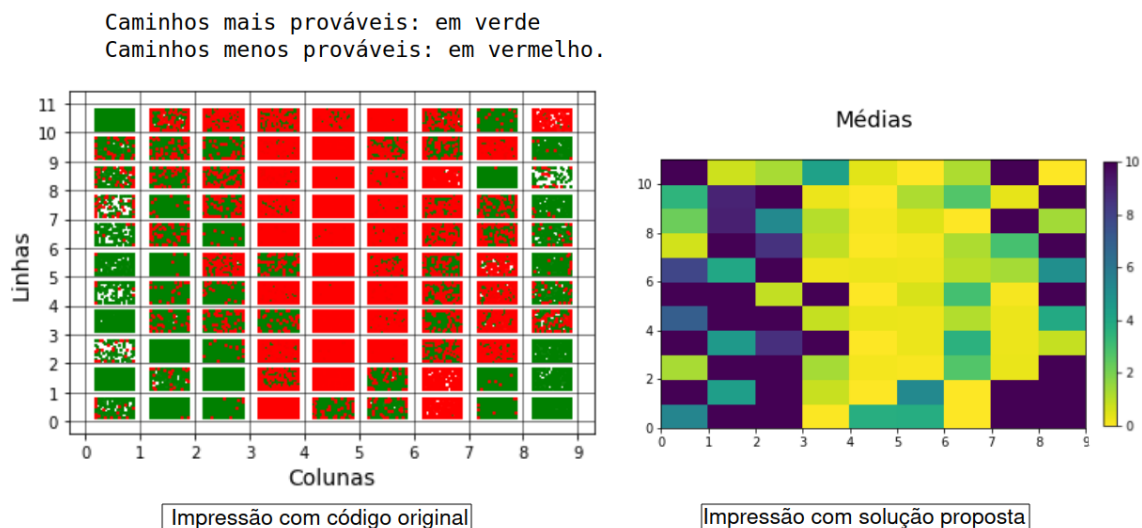
plt.colorbar(im) # Escala de cores.

plt.subplots_adjust(right=3, wspace=0.3)

plt.show()

```

A Figura 11 coloca lado a lado a visualização gerada pelo código original do projeto (do lado esquerdo) e o mapa de calor das médias, gerado pela solução apresentada nesta seção (do lado direito). Como pode-se observar, os esquemas de visualização são altamente semelhantes, mostrando que a nova solução também gera uma resposta para o problema proposto no projeto. Na Subseção 3.5, serão comparados os tempos de processamento para cada esquema de visualização, mostrando que a proposta desta seção impacta de forma significativa o desempenho.



**Figura 11. Comparação das visualizações.**

### 3.4.2. Processamento Paralelo das Simulações

Nesta seção, utiliza-se o módulo *multiprocessing* da biblioteca padrão da linguagem *Python* para processar paralelamente as chamadas da função **simulacao**.

A ideia consiste em formar um *pool* com os 8 núcleos<sup>7</sup> do processador do computador de teste. Segundo [Perkovic 2016], “Um objeto Pool representa um grupo de um ou mais processos, cada um deles capaz de executar código independentemente em um núcleo processador disponível.” O código é o seguinte:

```
from multiprocessing import Pool

pool = Pool(8)

valores = [2000000 for _ in range(8)]
dados = pool.map(simulacao, valores)
dados = np.array(dados)
df = pd.DataFrame(dados,
    index=['2 milhoes', '4 milhoes', '6 milhoes', '8 milhoes',
          '10 milhoes', '12 milhoes', '14 milhoes', '16 milhoes'],
    columns=['Esquerda', 'Centro', 'Direita'])
```

O resultado do processamento do código acima é um *DataFrame Pandas* com valores muito próximos àqueles mostrados na Tabela 1, na Subseção 3.1.3. Na Subseção 3.5, os tempos de processamento sem computação paralela e com paralelismo são confrontados, sendo então possível confirmar a vantagem obtida com os resultados apresentados nesta seção.

### 3.4.3. Redução de Lentidão Acumulada

A solução proposta para evitar a lentidão acumulada consiste na seguinte sequência.

- Verificar se existe um arquivo binário (aqui denominado **variaveis**) no diretório atual, ou caso não exista, criar o arquivo com o módulo *shelve*:

```
import os
import shelve

path = os.path.join('.', 'variaveis')

if not os.path.exists(path):
    variaveis = shelve.open(path)
    print(f'Arquivo variaveis gerado no diretorio atual: {path}')
    variaveis.close()
else:
    print(f'O arquivo variaveis ja existe no diretorio atual: \
{path}')
```

---

<sup>7</sup>No módulo *multiprocessing*, o método `cpu_count()` retorna a quantidade de núcleos do processador do computador utilizado.



- Abrir o arquivo **variaveis**:

```
variaveis = shelve.open(path)
```

- Utilizar um bloco *try - exception* para averiguar a existência de uma determinada variável no arquivo *shelf* e, caso não exista, criar a variável:

```
try:
    if variaveis['configuracoes_dados']:
        print(f'A variavel configuracoes_dados ja existe em {path}.')
except KeyError:
    variaveis['configuracoes_dados'] = produtoFiltrado()
    print(f'Variavel configuracoes_dados gravada com sucesso em \
        {path}.')
```

- Averiguar a existência ou criar as outras variáveis do código do Projeto Zigue-Zague no arquivo *shelf*; processo aqui omitido devido ser idêntico ao item anterior.
- Fechar o arquivo **variaveis**:

```
try:
    if variaveis:
        variaveis.close()
        print(f'O arquivo variaveis foi finalizado.')
except NameError:
    print(f'O arquivo variaveis ja foi finalizado.')
```

Como resultado, obteve-se a gravação das variáveis em um processo um pouco mais lento que o normal na primeira vez que os códigos acima foram executados, porém a partir da segunda vez, onde os valores das variáveis são buscados no arquivo *shelf*, o ganho de tempo de processamento torna-se evidente e será mostrado na Subseção 3.5.

### 3.5. Comparação de Desempenho

Para comparar o desempenho do projeto original com o desempenho obtido com os resultados propostos e implementados neste estudo, primeiro compara-se os principais itens da tabela 2 (com seus respectivos valores de tempo antes e após as implementações apresentadas nas Subseções 3.4.1 e 3.4.2).

A execução de todas as chamadas das funções **imprimeCaminho** e **simulacao** correspondem juntas a aproximadamente 4 horas, 51 minutos e 10 segundos. O código desenvolvido na Subseção 3.4.1 para substituir as impressões de **imprimeCaminho**, juntamente com o processamento paralelo realizado na Subseção 3.4.2, são executados em aproximadamente 27 min, 1 s e 889 ms. Em outras palavras, o ganho de tempo de processamento nos gargalos do código do Projeto Zigue-Zague é de aproximadamente

90,71%. Portanto, a estratégia adotada de fato identificou e melhorou de forma significativa os gargalos do código.

Em um segundo momento, executa-se todo o código do projeto sem a persistência de dados e compara-se com o desempenho do código original. Por último, para determinar o ganho com a persistência de dados e, também, estabelecer o resultado final, todo o código é gerado utilizando o arquivo de persistência.

Na Tabela 3, são apresentados os valores referentes a cada etapa do projeto. Na última linha, pode-se ver o tempo total de processamento para as versões do código sem e com persistência de dados. A diferença total é de aproximadamente um minuto no tempo final, sendo maior na etapa de geração do mapa de calor.

**Tabela 3. Comparação dos tempos de processamento.**

<b>Etapas</b>	<b>Com mapas de calor e processamento paralelo</b>	<b>Com mapas de calor, processamento paralelo e persistência de dados</b>
Geração do mapa de calor	1 min e 10 s	17.1 s
Criação dos métodos utilizados na simulação e formação do <i>pool</i>	889 ms	276 ms
Processamento do <i>pool</i>	25 min e 51 s	25 min e 44 s
Geração do <i>DataFrame Pandas</i>	3.61 ms	4.97 ms
Geração das médias do <i>DataFrame</i>	213 $\mu$ s	207 $\mu$ s
Cálculo do percentual no <i>DataFrame</i>	7.53 ms	11.6 ms
Média do percentual	3.07 ms	3.14 ms
Encerramento do arquivo de variáveis		178 ms
<b>Total</b>	27 min, 1 s, 903 ms e 423 $\mu$ s ( $\approx$ 27 min e 2 s)	26 min, 1 s, 573 ms, 710 $\mu$ s ( $\approx$ 26 min e 2 s)

Deve-se observar que o código responsável pela geração do mapa de calor é potencial gerador de lentidão acumulada. De fato, mapas de calor são constantemente aplicados em projetos de exploração (como o Zigue-Zague) com o objetivo de apreender algum novo fato sobre o caso em estudo, sendo comum executar (após modificar parâmetros) repetidamente as células de código de um projeto (em *Jupyter Notebook*), para observar novas perspectivas de um problema.

Enfim, o código final (já com a persistência de dados) apresentou um ganho de tempo de processamento de aproximadamente 92,5% do tempo total do código original,

de modo que as estratégias aplicadas foram bem sucedidas.

#### 4. Considerações Finais

Conforme antecipado na Seção 3.5, os objetivos propostos neste estudo foram alcançados e, assim, obteve-se resposta satisfatória para o problema de pesquisa, conforme descrito na Seção 1.1.

Conseguiu-se realizar o processamento do código em apenas 7,5% do tempo do projeto original. Além de uma mera redução de tempo, salienta-se o aspecto da dinamicidade desejada para o estudo.

De fato, os seguintes pontos chamam a atenção:

- A geração da nova visualização com mapa de calor não somente é mais rápida, como também permite explorar numericamente e sem ambiguidades as diversas regiões do tabuleiro. Por exemplo, a percepção de regiões mais favoráveis, pode ser melhor formalizada a partir da variável **medias** (referente às razões para cada casa do tabuleiro), ou seja, aquilo que antes era possível somente a partir de uma análise visual, agora pode ser matematicamente estudado.
- A persistência das variáveis...

#### Referências

- Fonseca, J. (2022). *gprof2dot*. Disponível em: <https://github.com/jrfonseca/gprof2dot>. Acesso em: 22 ago. 2022.
- Hegde, V. (2004). *Programmer's Toolkit: Profiling programs using gprof*. In: *Linux Gazette - March 2004 (#100)*. Disponível em: <https://linuxgazette.net/100/vinayak.html>. Acesso em: 16 mai. 2022.
- McKinney, W. (2018). *Python para Análise de Dados: tratamento de dados com Pandas, Numpy e Ipython*. trad. Lúcia A. Kinoshita. 1. ed. 2. reimpressão. São Paulo, Novatec.
- Perkovic, L. (2016). *Introdução à computação usando Python: um foco no desenvolvimento de aplicações*. trad. Daniel Vieira. 1. ed. Rio de Janeiro, LTC.
- Python (2022). *Library Reference*. Disponível em: <https://docs.python.org>. Acesso em: 24 out. 2022.
- Silva, A. F. and Kodama, H. M. Y. (2007). *Variações de um mesmo tema: Zigue-Zague e as expressões numéricas*. Núcleos de ensino da UNESP artigos 2007, São Paulo, p. 747 - 759.
- SnakeViz (2022). *SnakeViz*. Disponível em: <https://jiffyclub.github.io/snakeviz/#snakeviz>. Acesso em: 22 ago. 2022.

UNESP/IBILCE (2022). *Jogos no Ensino Fundamental II: 6º ao 9º Ano*. Departamento de Matemática. Laboratório de Matemática. Disponível em: <https://www.ibilce.unesp.br/#!/departamentos/matematica/extensao/lab-mat/jogos-no-ensino-de-matematica/6-ao-9-ano/>. Acesso em: 28 fev. 2022.