Avaliação do desempenho computacional do Projeto Zigue-Zague

Michel Ferracini[1], **André Calisto Souza Medeiros Guedes**[1]

[1]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*[1] e está disponível em `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

[1] *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*[2] 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 10 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. Substituição de códigos usando a biblioteca *Python NumPy*, comumente utilizada em computação científica.
3. Utilização dos decoradores da biblioteca *Python Numba* nos gargalos encontrados com *%timeit* e com os programas *gprof2dot* e *SnakeViz*.
4. Utilização da opção de computação paralela, em iterações onde seja necessário percorrer todos os itens de um *array*.

O item 1 advém da percepção, apontada na Seção 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.

Já os itens 2, 3 e 4 são propostos pela natureza dos códigos do projeto, sempre envolvendo muitas interações com laços *for*, de modo que cada uma das propostas sugeridas são viáveis, como fundamentado na Seção 2.

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*[3]
- "Devemos esquecer as pequenas eficiências, digamos cerca de 97% das vezes: a otimização prematura é a raiz de todos os males." - - *Donald E. Knuth*[4]

[2] *IPython* é um interpretador interativo para várias linguagens de programação, mas especialmente focado em *Python*.

[3] 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*

[4] 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 o comando mágico *%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 Seção 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 subseção, 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[5]. 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. Biblioteca *NumPy*

A fonte do texto a seguir, exceto por pequenas adaptações de escrita, novamente é [McKinney 2018].

NumPy é uma abreviatura de *Numerical Python* (ou Python Numérico), sendo um dos pacotes básicos mais importantes para processamento numérico em *Python*. A maioria dos pacotes de processamento com funcionalidades científicas utiliza objetos *array* do *NumPy* como a "língua franca" para troca de dados.

Alguns dos recursos disponibilizados pelo *NumPy* são:

- ndarray: *array* multidimensional eficaz, que oferece operações aritméticas rápidas, orientadas a *arrays* e recursos flexíveis de *broadcasting*.
- Funções matemáticas para operações rápidas em *arrays* de dados inteiros, sem que seja necessário escrever laços.
- Ferramentas para ler ou escrever dados de *array* em disco e trabalhar com arquivos mapeados em memória.

Um dos motivos para o *NumPy* ser tão importante para processamentos numéricos em *Python* é o fato de ele ter sido projetado para ser eficaz em grandes *arrays* de dados. Há uma série de motivos para isso:

- Internamente, o *NumPy* armazena dados em um bloco contíguo de memória, independentemente de outros objetos *Python* embutidos. A biblioteca do *NumPy*

[5]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.

de algoritmos escritos na linguagem C é capaz de atuar nessa memória sem qualquer verificação de tipo ou outro *overhead*. Os *arrays NumPy* também utilizam muito menos memória que as sequências embutidas de *Python*.
- As operações do *NumPy* realizam processamentos complexos em *arrays* inteiros sem a necessidade de laços *for* de *Python*.

É importante notar que os algoritmos baseados no *NumPy* geralmente são de 10 a 100 vezes mais rápidos do que suas contrapartidas em *Python* puro, além de utilizarem significativamente menos memória.

2.5. Biblioteca Numba

O texto a seguir consiste em uma tradução livre e adaptada de parte da documentação oficial do *Numba*, sendo [Numba 2022] a fonte original.

Numba é um compilador *just-in-time* para *Python* que funciona melhor em código que usa matrizes e funções *NumPy* e *loops*. A maneira mais comum de usar o *Numba* é através de sua coleção de decoradores que podem ser aplicados às suas funções para instruir o *Numba* a compilá-las. Quando uma chamada é feita para uma função decorada com *Numba*, ela é compilada para o código de máquina "*just-in-time*" para execução e todo (ou parte do) seu código pode ser executado na velocidade do código de máquina nativa.

A utilização do *Numba* é simples, não sendo necessário substituir o interpretador *Python*, executar uma etapa de compilação separada ou até mesmo ter um compilador C/C++ instalado. Basta aplicar um dos decoradores do *Numba* à função Python desejada e a biblioteca *Numba* executa o processo automaticamente.

O *Numba* será considerado uma boa escolha para melhorar o desempenho de um *software* quando o código for orientado numericamente (faz muita matemática), usa muito o *NumPy* ou tem muitos *loops*.

2.6. Computação Paralela

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

> "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 multiprocessador 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.

Outra possibilidade, em *Python*, é a utilização do decorador para processamento paralelo disponível no interpretador *Numba*, apresentado na Seção 2.5.

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.

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 da estratégia vencedora a partir de simulações e análise estatística.

A modelagem matemática e computacional gerada na fase 1, é sintetizada em uma lista contendo todos os 339 699 caminhos do tabuleiro, juntamente com as probabilidades de cada um destes caminhos. O diagrama apresentado na Figura ilustra as funções criadas para gerar esta lista, com seus respectivos inter-relacionamentos.

3.2. Medições no Tempo de Execução

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 2.

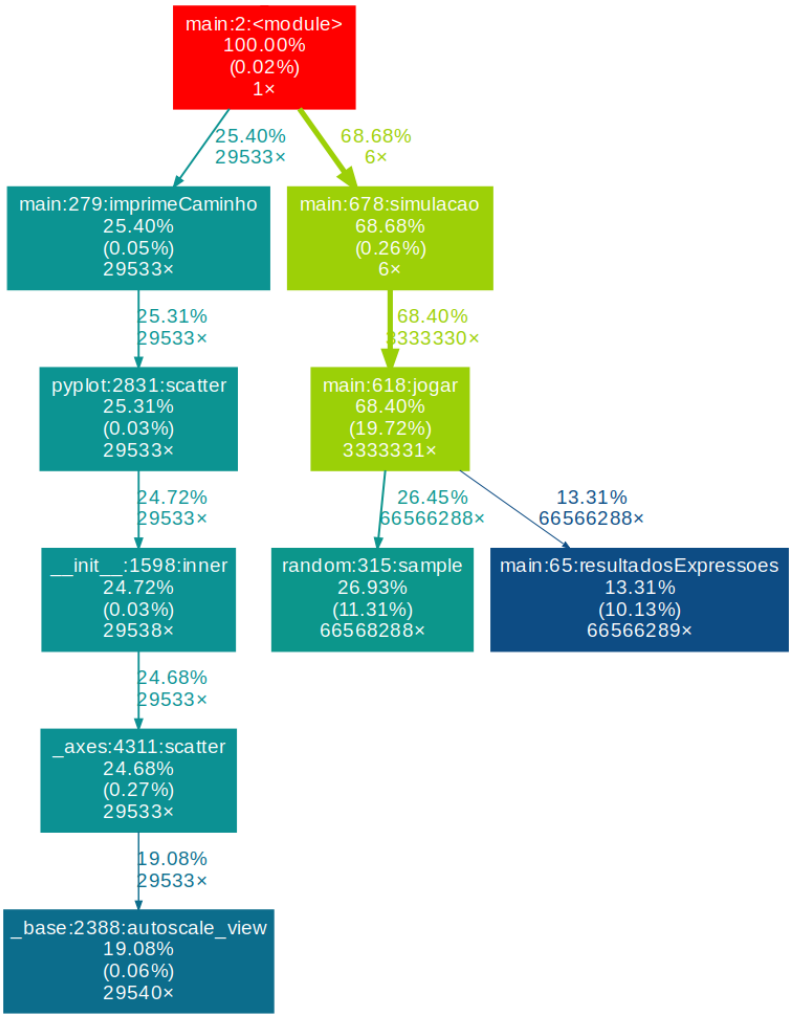

Figura 2. Grafo do programa, com poda para 10% ou menos do tempo de processamento.

No lado esquerdo do grafo (Figura 2), a função **imprimeCaminho** possui como filho o método **scatter** do módulo *pyplot* da biblioteca *matplotlib*, sendo chamado todas as 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 3), 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 2, a função **resultadosExpressoes** é a única que não aparece, mas aplicando *zoom* torna-se possível observar essa informação, como mostra a Figura 4.

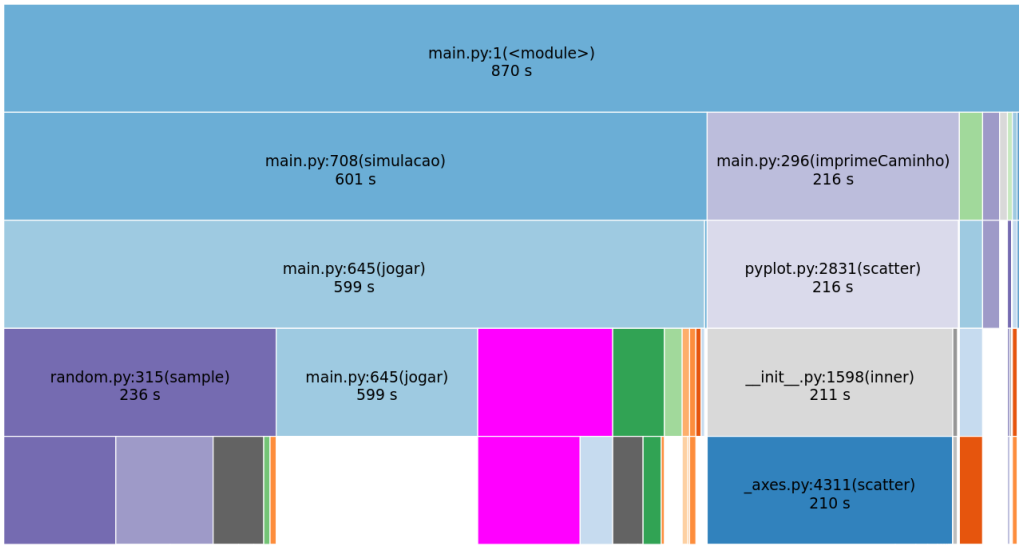

Figura 3. Visualização gerada com o *SnakeViz*.

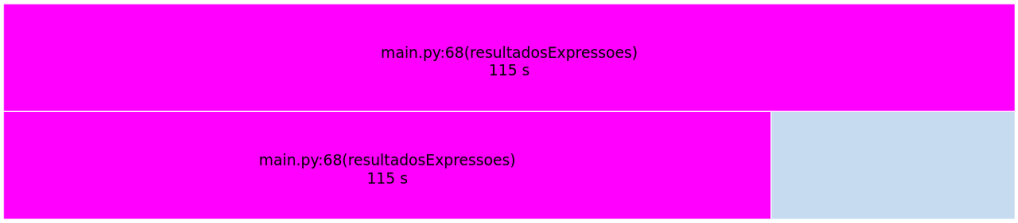

Figura 4. *Zoom* para visualizar a função resultadosExpressoes.

Apesar de utilizarem como base o mesmo arquivo de saída do módulo *pstats*, as visualizações geradas com o *gprof2dot* e com o *SnakeViz* apresentam resultados ligeiramente discrepantes. Por exemplo, o grafo (Figura 2) mostra que as funções **imprimeCaminho** e **simulacao** ocupam juntas 94,08% do processamento, enquanto a visualização gerada pelo *SnakeViz*, apresenta o resultado 93,91%. A discrepância ocorre devido divergências em critérios de arredondamento (fato constatado na tabela estática gerada pelo *SnakeViz*).

O grafo foi utilizado apenas para permitir melhor visualização da relação entre as funções, mas os dados de tempo de processamento (e, consequentemente, do percentual) utilizados neste artigo são provenientes da visualização gerada pelo *SnakeViz*. O resumo destas informações do programa são mostrados na Tabela 1, 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 a porcentagem do tempo gasto na execução de determinada função com relação ao tempo total de execução, correspondente à função **main**.

Tabela 1. Resumo da medição dos tempos de execução do programa.

	Função	**Número de chamadas**	**Tempo acumulado (segundos)**	**Percentual**
1	**main**	1	870	100
1.1	**simulacao**	6	601	69,07
1.1.1	**jogar**	3333331	599	68,79
1.1.1.1	**sample**	66566289	236	27,15
1.1.1.2	**resultadosExpressoes**	66566289	115	13,25
1.2	**imprimeCaminho**	29533	216	24,86
1.2.1	**scatter**	29533	210	24,16

Da análise realizada nesta subseção, conclui-se que as funções listadas na Tabela 1 são as mais relevantes para o desempenho dos códigos no projeto Zigue-Zague. Portanto, na próxima subseção, a busca por hipóteses que expliquem o desempenho dos códigos do projeto são relacionadas à esta tabela.

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 Gazzette - 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.

Numba (2022). *Numba Documentation*. Disponível em: https://numba.readthedocs.io/en/stable/user/index.html. Acesso em: 16 mai. 2022.

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). *The Python Profilers. Disponível em: https://docs.python.org/3/library/profile.html. Acesso em: 23 ago. 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.