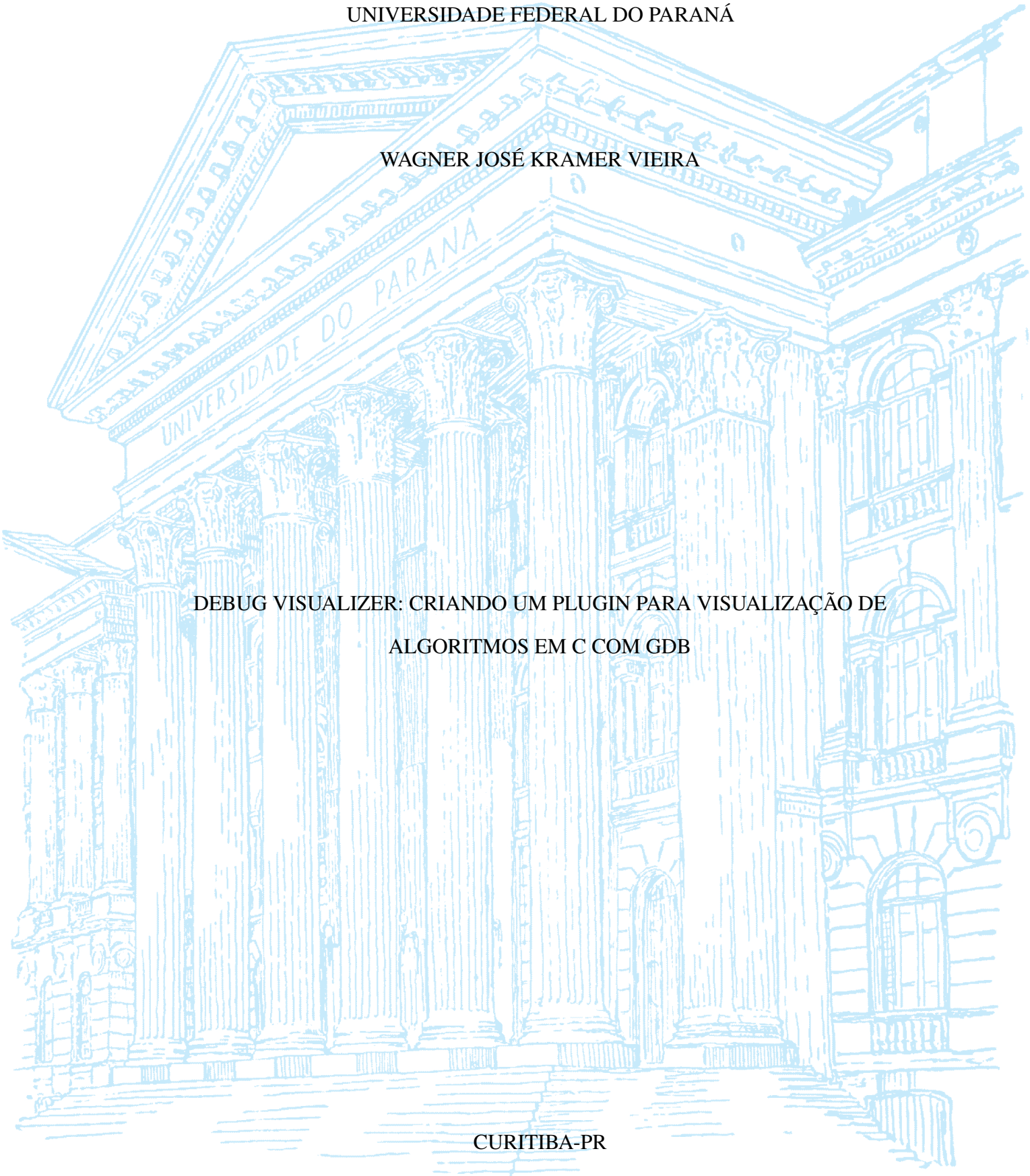
UNIVERSIDADE FEDERAL DO PARANÁ

WAGNER JOSÉ KRAMER VIEIRA




# DEBUG VISUALIZER: CRIANDO UM PLUGIN PARA VISUALIZAÇÃO DE ALGORITMOS EM C COM GDB

CURITIBA-PR

2023

WAGNER JOSÉ KRAMER VIEIRA

# DEBUG VISUALIZER: CRIANDO UM PLUGIN PARA VISUALIZAÇÃO DE ALGORITMOS EM C COM GDB

Trabalho apresentado como requisito parcial à conclusão do Curso de Bacharelado em Ciência da Computação, Setor de Ciências Exatas, da Universidade Federal do Paraná.

Área de concentração: *Ciência da Computação*.

Orientador: Carlos Alberto Maziero.

CURITIBA-PR

2023

## AGRADECIMENTOS

À minha mãe, Denise, que lutou para eu poder alcançar o que tenho hoje. A todos os que estiveram ao meu lado durante esse período de sobrecarga, que me ajudaram a focar e lutar para eu chegar aqui. Ao meu orientador, Prof. Carlos Maziero, que me auxiliou e não desistiu de mim. Ao meu chefe e amigo, Jackson Machado, que me incentivou a lutar e que foi o grande responsável pelo meu sucesso na área que eu amo.

## RESUMO

Este estudo aprimora o plugin Debug Visualizer, uma ferramenta de visualização de algoritmos disponível para o editor de texto Visual Studio Code. O objetivo é converter os dados da memória de algoritmos em uma visualização que mostre claramente as estruturas de dados neles presentes. São apresentados, inicialmente, os conceitos de dados, informação e conhecimento, bem como o funcionamento e a utilização do GDB. Em seguida, apresentam-se os objetivos deste trabalho, o que se planeja alcançar ao expandir o Debug Visualizer, incluindo uma explicação de como funciona a depuração no Visual Studio Code e a arquitetura do Debug Visualizer. A seguir, mostra-se como é feita a introspecção de memória com o GDB, a criação de grafos que representam a memória do algoritmo em execução e as maneiras de converter estes grafos em uma visualização útil. Os objetivos alcançados, as visualizações, a comparação com outras ferramentas e as utilidades descobertas são demonstradas na sequência. Apresentam-se, por fim, as considerações finais, as contribuições deste trabalho e sugestões de trabalhos futuros.



Palavras-chave: Debug Visualizer; Visualização de Algoritmos; GDB.

## ABSTRACT

This study improves the Debug Visualizer plugin, an algorithm visualization tool available for the Visual Studio Code text editor. The goal is to convert algorithms memory data into a visualization that clearly shows the data structures present in them. Initially, the concepts of data, information, and knowledge are presented, as well as the operation and use of the GDB. Then, the objectives of this work are presented, what is planned to be achieved by expanding the Debug Visualizer, including an explanation of how debugging works in Visual Studio Code and the architecture of the Debug Visualizer. Next, it is shown how memory introspection is done with GDB, the creation of graphs that represent the memory of the running algorithm and ways to convert these graphs into a useful visualization. The objectives reached, the visualizations, the comparison with other tools and the utilities discovered are shown below. Finally, the final considerations, the contributions of this work and suggestions for future work are presented.



Keywords: Debug Visualizer; Algorithm Visualization; GDB.

## LISTA DE FIGURAS

## LISTA DE TABELAS

## LISTA DE ACRÔNIMOS

| | |
|---|---|
| DINF | Departamento de Informática |
| UFPR | Universidade Federal do Paraná |
| GDB | GNU Debugger |
| VSCode | Visual Studio Code |
| DDD | Data Display Debugger |
| BST | Binary Search Tree |

**SUMÁRIO**

# 1 INTRODUÇÃO

Este capítulo apresenta a justificativa para o desenvolvimento desta pesquisa, a proposta criada e os desafios enfrentados. Por fim, apresenta-se a organização dos demais capítulos deste documento.

## 1.1 MOTIVAÇÃO

Em Ciência da Computação, Algoritmos e Estruturas de Dados são considerados um dos tópicos mais relevantes. A base para codificação vem desta área, sendo fundamental para qualquer estudante não somente compreender, mas também dominar tal conhecimento. Segundo Santos e Costa (2006), o iniciante deve ter sua base bem sedimentada, para ficar apto a prosseguir de maneira positiva durante os seus estudos.

As disciplinas de Algoritmos e Estruturas de Dados do Departamento de Informática da Universidade Federal do Paraná (DINF) usam diagramas e representações visuais para demonstrar as estruturas de dados estudadas. Sendo algo bastante recorrente também em outras instituições de ensino, essas formas de representação tornam mais simples a compreensão dos estudantes, mostrando de uma maneira didática a memória do computador ao executar certos algoritmos.

Essas representações são vistas em sala de aula com instâncias específicas das estruturas de dados, o que pode deixar dúvidas que surgem somente durante a implementação dos algoritmos. Além disso, os alunos apresentam dificuldades em compreender problemas que surgem durante a criação de tais algoritmos, uma vez que não sabem como utilizar a memória do programa para identificar e corrigir erros.

Diante disso, é importante ensinar, de maneira clara, como depurar código e visualizar algoritmos, o que supriria uma lacuna existente fora da sala de aula.

## 1.2 PROPOSTA

O objetivo deste trabalho é aprimorar a ferramenta Debug Visualizer, permitindo que os algoritmos sejam visualizados de forma dinâmica, assim como acontece em sala de aula. Além disso, será apresentado como utilizar a ferramenta, outras ferramentas equivalentes e maneiras de integrar o GDB com o Visual Studio Code, para facilitar o desenvolvimento de algoritmos.

## 1.3 DESAFIOS

Os maiores desafios dessa pesquisa são determinar formas de representar estruturas de dados, permitindo a generalização e a representação de diferentes algoritmos, e realizar a introspecção de memória para gerar um grafo que armazene o estado atual da execução do algoritmo em pontos determinados pelo programador. Além disso, definir os limites da ferramenta e ampliar as suas maneiras de visualização serão etapas importantes para atingir representações úteis dos algoritmos.

## 1.4 ORGANIZAÇÃO DO DOCUMENTO

Após esta seção introdutória, o documento será estruturado da seguinte forma.

O capítulo 2 contém a fundamentação teórica necessária para compreender este documento. Neste tópico, serão apresentados os conceitos de dados, informação e conhecimento, tanto do ponto de vista computacional, quanto do contexto psicológico. Mais adiante, será apresentado como ocorre a visualização de dados, bem como se dá a depuração de programas e a introspecção de memória.

No capítulo 3 serão apresentadas ferramentas de visualização de algoritmos e uma consideração sobre o porquê elas não seriam ótimas para um ambiente acadêmico.

Em seguida, no capítulo 4, será apresentada a proposta de pesquisa, bem como a depuração com o Visual Studio Code e a ferramenta Debug Visualizer.

A seguir, no capítulo 5, serão apresentados os detalhes da implementação da proposta, incluindo todo o processo de coleta de dados do algoritmo, a construção de um grafo de memória e a transformação deste grafo em uma visualização.

No capítulo 6, serão apresentados os resultados obtidos, algumas visualizações interessantes, considerações sobre a facilidade de uso e as limitações da ferramenta.

Enfim, no capítulo 7, serão apresentadas as considerações finais, inclusive uma conclusão do que foi desenvolvido, quais os resultados alcançados, e como baixar a ferramenta.

# 2 FUNDAMENTAÇÃO TEÓRICA

Atualmente, assistimos ao surgimento de novas ferramentas que auxiliam no ensino da computação. Podemos compreender estas ferramentas através de um modelo que utiliza dados para gerar informação e levar ao conhecimento.

Tal modelo é conhecido como a hierarquia de Dados-Informação-Conhecimento-Sabedoria (DIKW) (Bavoil et al., 2005), a partir dele podemos definir o funcionamento do processo necessário para a geração de conhecimento.

Sobre o modelo DIKW, segundo Chen et al. (2009), podemos nos focar apenas no subconjunto de dados, informação e conhecimento, dado que:

> "Seja $\mathbb{P}$ o conjunto de toda a memória humana possível, explícita e implícita. O primeiro abrange a memória de eventos, fatos e conceitos, e a compreensão dos significados, contextos e associações. O último abrange todas as formas não conscientes de memória, como respostas emocionais, habilidades, hábitos e assim por diante.Podemos nos focar em três subconjuntos da memória, $\mathbb{P}_{data} \subset \mathbb{P}$, $\mathbb{P}_{info} \subset \mathbb{P}$ e $\mathbb{P}_{know} \subset \mathbb{P}$, onde $\mathbb{P}_{data}$, $\mathbb{P}_{info}$ e $\mathbb{P}_{know}$ são o conjunto de toda a memória explícita e implícita sobre dados, informação e conhecimento, respectivamente." (Chen et al. (2009), tradução nossa).

Tabela 2.1: Definição de dados, informação e conhecimento no espaço computacional. (Adaptado de Chen et al. (2009))

| *Categoria* | *Definição* |
|---|---|
| `Dados` | Representação computacional dos modelos e atributos de entidades reais ou simuladas. |
| `Informação` | Dados que representam o resultado de um processo computacional, tais como análises estatísticas, para atribuir significado aos dados, ou às transcrições de alguns significados atribuídos por seres humanos. |
| `Conhecimento` | Dados que representam o resultado de um processo cognitivo, como a percepção, aprendizagem, associação e raciocínio, ou às transcrições de algum conhecimento adquirido por seres humanos. |

## 2.1 PROCESSO DE VISUALIZAÇÃO

Nosso objetivo com o processo de visualização, dadas as definições de Chen et al. (2009), é utilizar os dados de um determinado algoritmo para gerar informação sobre suas estruturas de dados, permitindo ao programador obter conhecimento.

O trabalho de Card et al. (1999) é considerado a base para os estudos sobre visualização de algoritmos. O livro se propõe a definir o campo emergente da visualização de informação, e foca no uso de visualização para descobrir relações. Além disso, o livro reúne uma série de artigos sobre assunto escritos nos anos 80 e 90, bem como análises sobre os mesmos.

Inicialmente, Card et al. (1999) propôs um modelo onde a pessoa interagindo com os dados de um sistema era responsável por ajustar todos os elementos da visualização da informação. A figura 2.1, apresentada no trabalho de Santos (2013) e adaptada de Card et al. (1999), mostra o modelo de referência para visualização.

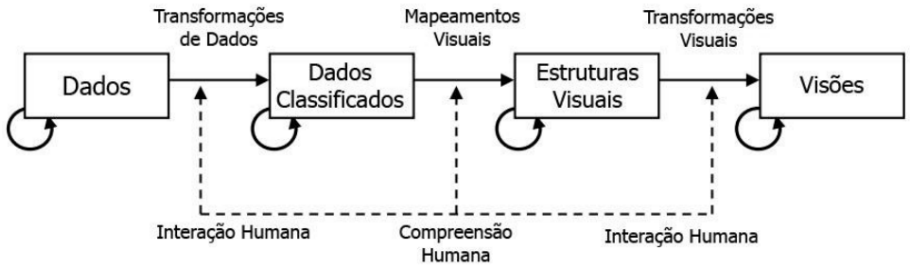



Figura 2.1: Modelo de referência para visualização. (Card et al., 1999)

Posteriormente, alguns autores subdividiram o processo de visualização de informação em modelos específicos, como o modelo de visualização assistindo por informações, o modelo de visualização assistido pelo conhecimento com representações adquiridas pelo conhecimento, e o modelo de visualização assistido pelo conhecimento com processamento cognitivo simulado (Chen et al., 2009).

## 2.2 VISUALIZAÇÃO ASSISTIDA POR INFORMAÇÃO

Dada a necessidade de visualizar informação a partir de dados existentes, sem a interação direta do usuário com a customização da visualização, surgiu o modelo de visualização assistida por informação. Juntamente com um processo de visualização típico, existe um pipeline secundário que processa as informações dos dados e apresenta uma visualização paralela destes após certo processamento. Este modelo pode ser representado pela figura 2.2.

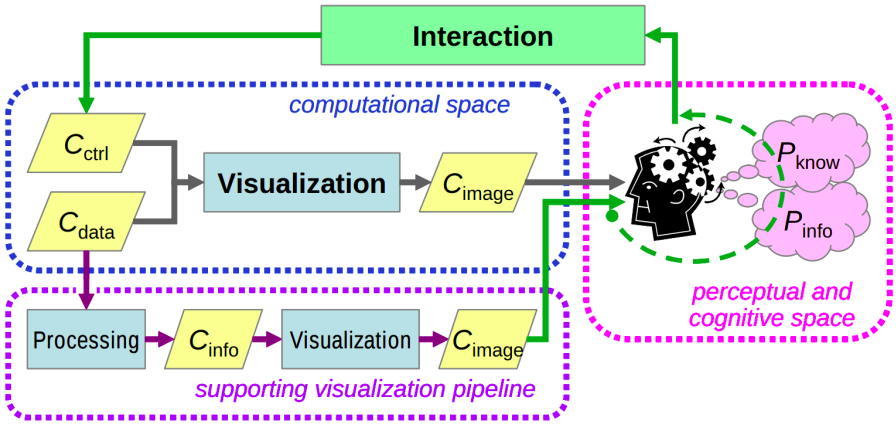



Figura 2.2: Modelo de visualização assistida por informação. (Chen et al., 2009)

Este pipeline secundário pode apresentar informações sobre os dados de entrada, porém também pode mostrar características do processo de visualização, propriedades dos dados ou características do processo de percepção do usuário (Chen et al., 2009).

## 2.3 DEPURAÇÃO DE PROGRAMAS E INTROSPECÇÃO DE MEMÓRIA COM GDB

O propósito de um depurador como o GDB é permitir a visualização do que está acontecendo “dentro” de outro programa enquanto ele é executado (ou o que outro programa estava fazendo quando ele teve sua execução interrompida) (Stallman et al., 2002).

Sendo um depurador de código aberto protegido pela licença GPL (GNU General Public Licence), o GDB pode ser usado para depurar código escrito em C, C++ e outras linguagens de programação, mas seu maior foco é nas duas primeiras.

Essa seção oferecerá uma breve introdução ao uso do GDB, sem se alongar nos detalhes de sua estrutura. A versão 13.1 do GDB deve ser considerada, utilizando o GCC como compilador de código.

O GDB possibilita que o utilizador visualize o código durante a execução, utilizando pontos de parada (*breakpoints*). Nesses pontos, o usuário pode executar comandos que permitem observar variáveis, endereços de memória, registradores, informações do código, entre outros.

### 2.3.1 Executando o GDB com um algoritmo de exemplo.

Para entender como o GDB pode ser utilizado, executaremos um exemplo simples, uma implementação do algoritmo *Bubble Sort*.

É importante saber que a execução do GDB usando diretamente o ambiente *Visual Studio Code* (VSCode) e diretamente pela linha de comando tem algumas diferenças. Isso se deve ao fato do VSCode utilizar o GDB com a MIEngine, uma ferramenta interna do VSCode responsável pela comunicação entre o editor de código e o depurador. Conforme forem aparecendo diferenças entre o GDB usado na linha de comando e o do VSCode, ambas as formas de utilização serão descritas.

O primeiro passo para utilizar o GDB se dá durante a compilação de código. É importante que durante ela, utilize-se a flag `-g`, desta forma o GCC irá incluir as informações necessárias para o GDB depurar o código. Além disso, recomenda-se não utilizar nenhuma flag de otimização de código.

Após isso, basta iniciar o GDB. Considerando o arquivo `bubble_sort.c`, para compilar e executar ele dentro do GDB, utilizaremos os seguintes comandos:

```
$ gcc -o bubble_sort -g bubble_sort.c
$ gdb bubble_sort
```

Para os próximos passos, será considerada a seguinte implementação do *Bubble Sort*:

```
1  #include <stdio.h>
2
3  int *bubble_sort(int *values, int arr_len)
4  {
5     int aux = 0;
6
7     for (int i = 0; i < arr_len; i++)
8     {
9        for (int j = 0; j < arr_len - i - 1; j++)
10       {
11          if (values[j] > values[j + 1])
12          {
13             aux = values[j];
14             values[j] = values[j + 1];
15             values[j + 1] = aux;
16          }
17       }
18    }
19
20    return values;
21 }
22
23 void print_arr(int *values, int arr_len)
24 {
25    for(int i = 0; i < arr_len; i++) {
26       printf("%d ", values[i]);
27    }
28
29    printf("\n");
30 }
31
32 int main()
33 {
34    int values[5] = {0, 2, 4, 3, 1};
35
36    bubble_sort(values, 5);
37
38    return values[0];
39 }
```

O algoritmo é dividido em três partes. A primeira parte é a função `main`, que inicializa um vetor de tamanho 5, ordena este vetor e retorna o primeiro elemento dele. A segunda parte é a função `bubble_sort`, que simplesmente ordena um vetor utilizando o algoritmo *Bubble Sort*. A terceira parte é a função `print_arr`, que simplesmente imprime um vetor.

Executemos dois passos, um antes e um depois da ordenação do vetor `values`. Vamos então adicionar dois *breakpoints* no nosso código, na linha 36 e na linha 38. No ambiente VSCode, basta clicar na parte esquerda da linha desejada, no GDB pelo terminal, basta executar os seguintes comandos:

```
break 36
break 38
```

Após a adição dos breakpoints, pode-se executar o código utilizando o comando `run` (ou iniciando a depuração pelo VSCode). Parando no primeiro breakpoint, temos duas formas de visualizar os valores do vetor, a primeira delas é simplesmente inspecionando os valores do vetor. Para visualizar o valor de uma variável executa-se o comando `print <nome da variável>`,

por exemplo, `print values[1]`. Ao executarmos isto, veremos que o valor de `values[1]` é 2.

A segunda forma é utilizando a função escrita no próprio código, para realizar uma chamada de função, utiliza-se o comando `call <chamada da função>`, para este exemplo, utiliza-se `call print_arr(values, 5)`. Então a função será executada e irá imprimir o vetor.

Para ir para o próximo breakpoint, utiliza-se o comando `continue`. Após executar ele, o GDB irá parar na linha 38. Se imprimir novamente o vetor, ele estará ordenado.

Por fim, executando-se novamente o comando `continue`, como não há mais breakpoints, o código irá executar até o fim e encerrar sua execução.

```
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from bubble_sort...
(gdb) break 36
Ponto de parada 1 at 0x131d: file bubble_sort.c, line 36.
(gdb) break 38
Ponto de parada 2 at 0x1332: file bubble_sort.c, line 38.
(gdb) run
Starting program: /home/kramer/TCC/bubble_sort
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at bubble_sort.c:36
36          int *ordered_values = bubble_sort(values, 5);
(gdb) print values[1]
$1 = 2
(gdb) call print_arr(values, 5)
0 2 4 3 1
(gdb) continue
Continuing.

Breakpoint 2, main () at bubble_sort.c:38
38          return ordered_values[0];
(gdb) call print_arr(values, 5)
0 1 2 3 4
(gdb) continue
```

Figura 2.3: Execução do *Bubble Sort* com o GDB.

Além disso, existem algumas operações interessantes para se executar. Pode-se, por exemplo, observar o tipo de uma variável utilizando-se o comando `ptype <variável>`, existe a possibilidade de realização de *casting*, acesso a um endereço de memória, acesso direto e indireto com *structs*, entre outras possibilidades.

# 3 FERRAMENTAS DE VISUALIZAÇÃO DE ALGORITMOS

Existem várias ferramentas de visualização de algoritmos online que podem ser usadas atualmente. Não existe um único recurso considerado a melhor opção, mas alguns são interessantes.

Estas foram escolhidas por serem as mais conhecidas ou utilizadas pelos professores ministrando as disciplinas de Algoritmos e Estruturas de dados na UFPR:

- O Algorithm Visualizer (Park, 2016) foi escolhido pela quantidade de visualizações e algoritmos disponíveis.
- O Python Tutor (Guo, 2013) foi uma ferramenta bastante difundida e serviu de base para outras ferramentas semelhantes.
- O Visualgo (HALIM, 2015) e o Data Structure Visualizations (Gales, 2012) são plataformas que costumam ser apresentadas aos alunos das disciplinas de Algoritmos e Estruturas de dados na UFPR.

## 3.1 ALGORITHM VISUALIZER

O Algorithm Visualizer (Park, 2016) é um projeto de código aberto que oferece ao usuário a possibilidade de visualizar vários algoritmos pré-estabelecidos. A plataforma virtual apresenta algoritmos para as mais variadas finalidades, incluindo recursão, ordenação, força bruta, dentre outros. Além disso, o usuário pode criar e visualizar o seu próprio código.

O Algorithm Visualizer apresenta os seus algoritmos nas linguagens Java e JavaScript, permitindo que estes sejam modificados. Além disso, mostra a execução dos algoritmos sem breakpoints, utilizando passos adicionados sempre que se encontram funções específicas, podendo avançar e retroceder no código.

Na figura 3.1, é possível visualizar a estrutura de edição do Algorithm Visualizer durante a execução do Quicksort. À esquerda, tem-se duas representações dos dados a serem ordenados. À direita, temos acesso ao código em execução e podemos fazer alterações.


Figura 3.1: Gráfico de barras, array e algoritmo dentro do Algorithm Visualizer.

## 3.2 PYTHON TUTOR

O projeto Python Tutor (Guo, 2013) tem como foco demonstrar a memória de um fragmento de código. Esta plataforma não possui algoritmos pré-definidos, mas é compatível com 5 linguagens de programação distintas.

O Python Tutor tem uma interface intuitiva. O usuário seleciona entre as linguagens Python, Java, JavaScript, C e C++, insere o código e pode visualizar como a memória se comporta durante a execução. A visualização passo a passo permite que o usuário avance e retroceda linha a linha no código, porém isso impede que algoritmos muito grandes sejam suportados, podendo levar alguns minutos para compilar.

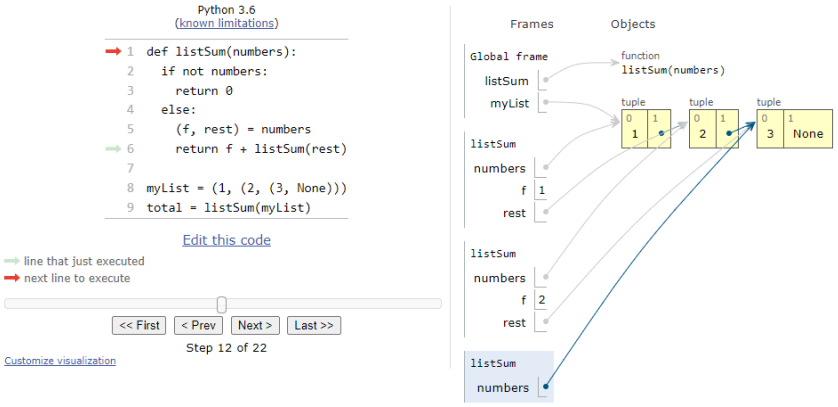



Figura 3.2: Visualização lista encadeada dentro do Python Tutor.

Na figura 3.2, temos a estrutura de visualização do Python Tutor. À esquerda temos o código sendo executado, neste caso um algoritmo que faz uma soma por meio de uma lista encadeada. À direita temos a visualização da pilha de chamadas, bem como os objetos gerados naquele momento.

## 3.3 VISUALGO E DATA STRUCTURE VISUALIZATIONS

O Visualgo (HALIM, 2015) e o Data Structure Visualizations (Gales, 2012) são plataformas mais simples para representar a evolução de algoritmos ao longo do tempo. As plataforma contém uma vasta gama de algoritmos prontos, no entanto não permitem a edição do código em si, somente sua visualização.

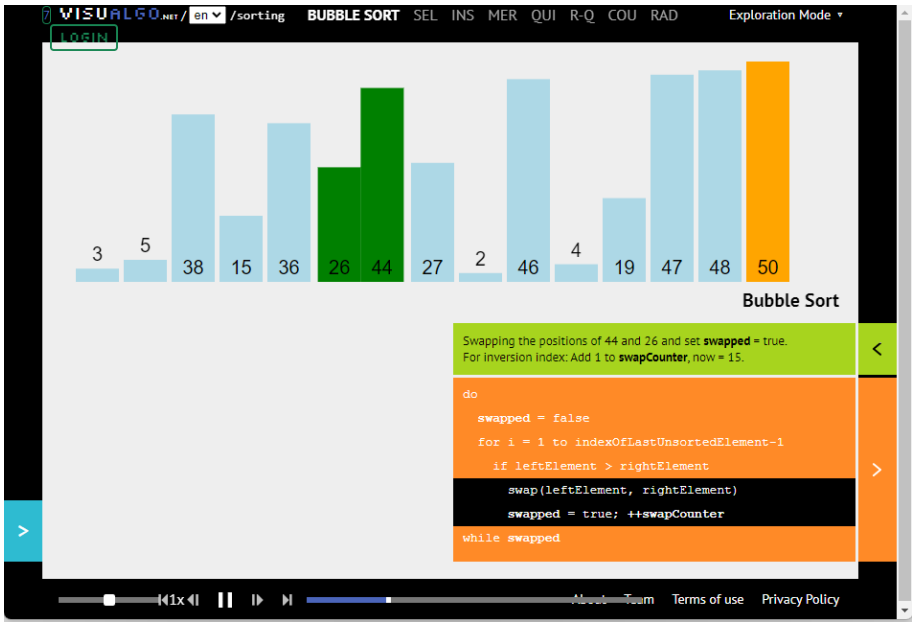



Figura 3.3: Visualização de algoritmo de ordenação dentro do Visualgo.

Na figura 3.3, temos a interface do Visualgo durante a visualização de algoritmos de ordenação. Na parte superior temos a seleção do algoritmo utilizado para realizar a ordenação, no centro temos o array sendo ordenado e abaixo temos o algoritmo em execução.

## 3.4 DDD - DATA DISPLAY DEBUGGER

O DDD (Zeller e Lütkehaus, 1995) é um front-end para um depurador inferior (Zeller, 2004) como, por exemplo, o GDB. As partes mais antigas do DDD foram escritas em 1990 quando Andreas Zeller projetou o VSL, uma linguagem de estrutura visual baseada em caixas para visualizar dados e estruturas de um programa (Zeller, 2004).

Sendo uma das primeiras ferramentas com as funcionalidades oferecidas, o DDD teve diversas atualizações e atualmente suporta diversas linguagens como Python e Perl.

O DDD contém um editor de código que se integra com o depurador, permitindo ao usuário avaliar como a memória se comporta durante a execução com o auxílio de breakpoints. Para algoritmos mais complexos, o DDD também fornece uma janela de visualização de estruturas de dados.

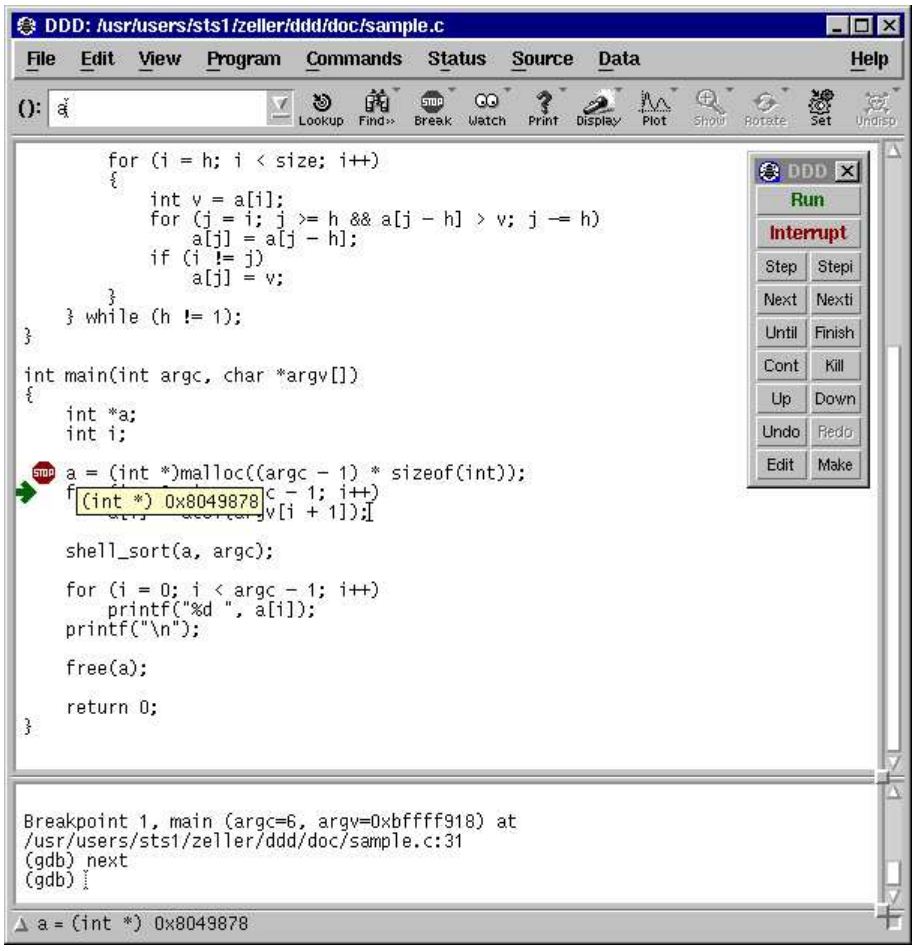



Figura 3.4: Janela de edição de código do DDD durante a depuração de código. (Zeller, 2004)

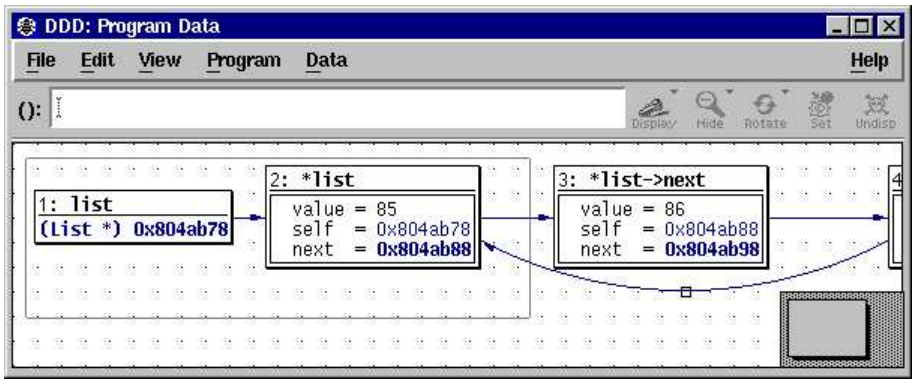



Figura 3.5: Janela de visualização de estruturas de dados do DDD. (Zeller, 2004)

Nas figuras acima, temos a visualização do DDD durante a depuração de um programa escrito na linguagem C. A figura 3.4 mostra o editor de texto, um terminal integrado e uma janela de comandos de depuração. Além disso, mostra o usuário com o mouse em cima da variável `a`, mostrando seu valor, um ponteiro com o endereço `0x8049878`.

Na figura 3.5, podemos ver um grafo direcionado criado pelo DDD que mostra uma lista encadeada cíclica.

## 3.5 CONSIDERAÇÕES

Uma vez tendo sido apresentadas estas ferramentas, é possível notar que a maioria delas tem como problema a execução de código na linguagem C. A única capaz de executar código nessa linguagem é o DDD, que acaba sendo uma ferramenta extremamente complexa e ultrapassada. Além disso, essas ferramentas necessitam de seu ambiente específico para serem executadas, seja o navegador ou o próprio editor de texto, limitando assim a integração com outras ferramentas.

Diante dessas reflexões, conclui-se que nenhuma dessas ferramentas seria simples de usar, completa e efetivamente proveitosa em um ambiente de aprendizado e programação dinâmico, como na UFPR.

# 4 PROPOSTA DE PESQUISA

O Departamento de Informática (DINF) da Universidade Federal do Paraná (UFPR) ensina Algoritmos e Estruturas tendo como base o livro "Algoritmos: Teoria e prática" (Cormen, 2012). Este livro apresenta uma variedade de algoritmos, muitos deles com uma representação usando grafos direcionados e outros diagramas. O propósito desta pesquisa é melhorar a extensão "Debug Visualizer", disponível no editor de texto "Visual Studio Code", para poder representar algoritmos escritos na linguagem C de maneira similar ao que é mostrado no livro e no DDD (Zeller e Lütkehaus, 1995).

Por exemplo, dado o seguinte pseudo-código para um algoritmo de rotação:

**Algoritmo 1** $LEFT-ROTATE(T;x)$ (Cormen, 2012)

```
 1: y = x.direita
 2: x.direita = y.esquerda
 3: if y.esquerda ≠ T.nil then
 4:     y.esquerda.p = x
 5: end if
 6: y.p = x.p
 7: if x.p == T.nil then
 8:     T.raiz = y
 9: else if x == x.p.esquerda then
10:     x.p.esquerda = y
11: else
12:     x.p.direita = y
13: end if
14: y.esquerda = x
15: x.p = y
```

Se estiver implementado corretamente na linguagem C, deve ser possível inserir pontos de interrupção antes, durante e depois da chamada da função, de forma que possa ser visualizado em forma de grafo o estado anterior à chamada da função, cada uma das transições e o estado posterior a ela. Deverá ser possível, então, visualizar um grafo análogo ao da figura 4.1.

A visualização deve ser simplificada, intuitiva e interativa, sendo criada somente com base na memória do programa, sem a necessidade de qualquer tipo de configuração pelo programador, exceto a definição da variável que será usada como ponto inicial para a visualização.

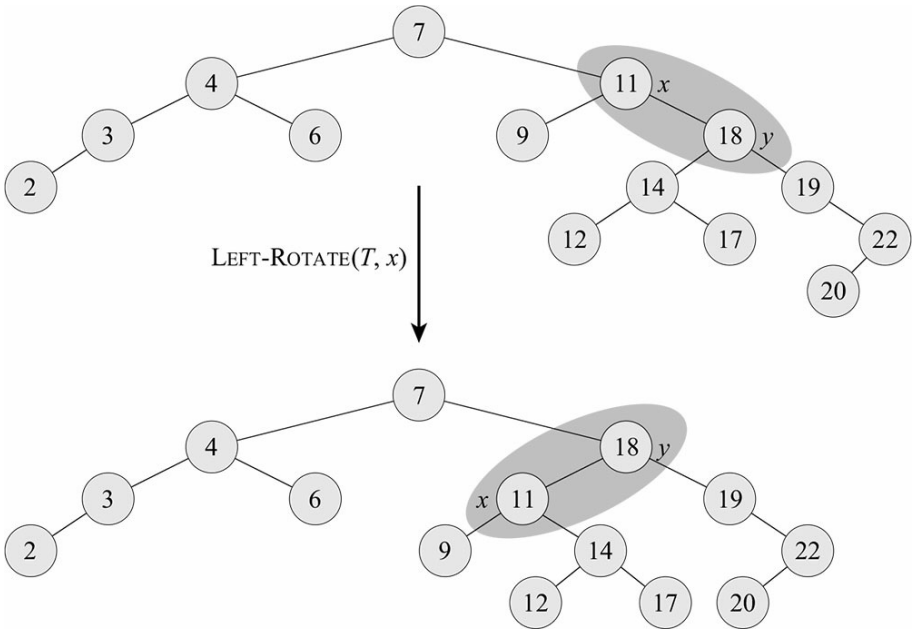



Figura 4.1: Rotação à esquerda em uma árvore de busca. (Cormen, 2012)

Além das estruturas de dados, deve ser possível visualizar variáveis individuais, ponteiros, vetores e matrizes, semelhantemente ao que é mostrado tanto no livro "Algoritmos: Teoria e prática" quanto no DDD, como exemplos.

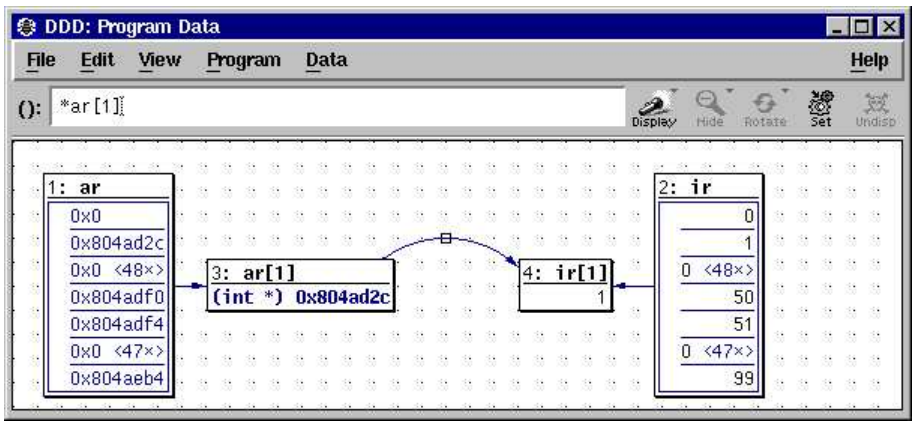



Figura 4.2: Visualização de vetor, variável simples e ponteiro no DDD. (Zeller, 2004)

| $A$ | 4 | 1 | 3 | 2 | 16 | 9 | 10 | 14 | 8 | 7 |
|---|---|---|---|---|---|---|---|---|---|---|

Figura 4.3: Um exemplo de vetor. (Cormen, 2012)

Tais visualizações devem conseguir demonstrar como funciona a memória do programa fidedignamente. Mostrando, por exemplo, que um ponteiro armazena somente um endereço de memória. Para este caso, a visualização deve conter um vértice $a$ com o endereço de memória, um vértice $b$ com outro endereço e uma aresta direcionada partindo de $a$ e chegando em $b$. Na figura 4.2, podemos visualizar isto considerando o vértice $a$ = vértice 3($ar[1]$) e o vértice $b$ = vértice 4($ir[1]$).

Enfim, a extensão deve conseguir representar:

| | | | |
|---|---|---|---|
| 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 |

Figura 4.4: Um exemplo de visualização de matriz identidade.

- Variáveis simples,
- Ponteiros, ponteiros para ponteiros,
- Structs,
- Matrizes,
- Vetores.

## 4.1 DEPURAÇÃO COM VISUAL STUDIO CODE

Desenvolvido pela Microsoft, o Visual Studio Code (VSCode) é um editor de texto multiplataforma utilizado amplamente por programadores ao redor do mundo. Entre diversas funcionalidades, esta ferramenta permite ao programador depurar seu código em tempo de execução, conectando-se a um depurador semelhantemente ao que o DDD faz.

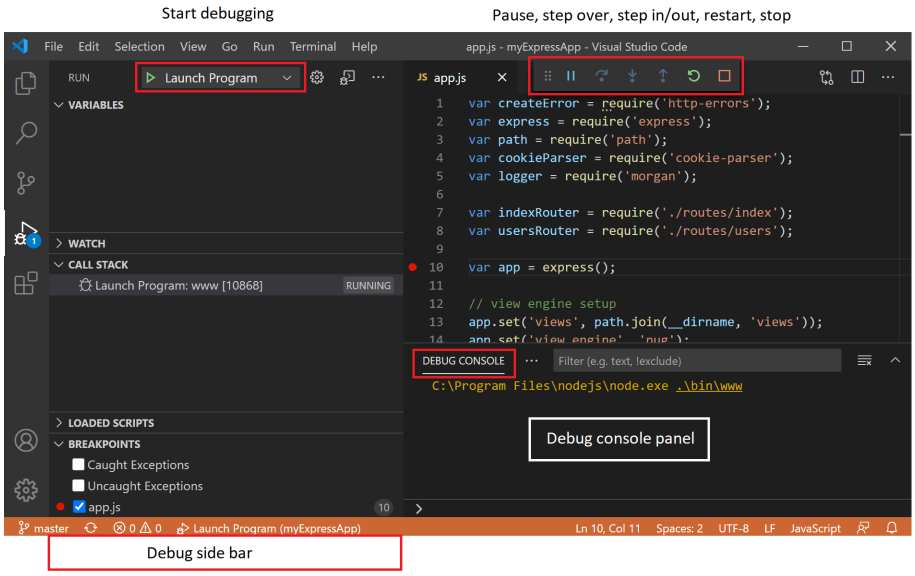



Figura 4.5: Janela de visualização do VSCode. (Microsoft, 2023)

A extensão “C/C++” [1] implementa o adaptador de depuração CPPDBG, capaz de fazer a comunicação entre o VSCode e o GDB. A extensão permite ao programador utilizar a estrutura de depuração do VSCode, mostrada na figura 4.5.

[1] disponível em https://marketplace.visualstudio.com/items?itemName=ms-vscode.cpptools

A depuração do VSCode é feita em três passos, o primeiro é definir o arquivo *tasks.json*. Este arquivo consiste em um JSON especificado pelo tipo de depurador utilizado e normalmente inclui um nome para a opção de depuração, um tipo de depuração, um tipo de requisição, entre outros parâmetros.

A figura 4.6 mostra um exemplo de configuração utilizado para depurar o arquivo *a.out* gerado pelo compilador.

```
{
  "name": "C++ Launch",
  "type": "cppdbg",
  "request": "launch",
  "program": "${workspaceFolder}/a.out",
  "args": ["arg1", "arg2"],
  "environment": [{ "name": "config", "value": "Debug" }],
  "cwd": "${workspaceFolder}"
}
```

Figura 4.6: Exemplo de arquivo *tasks.json* usado para depurar com o CPPDBG. (Microsoft, 2023)

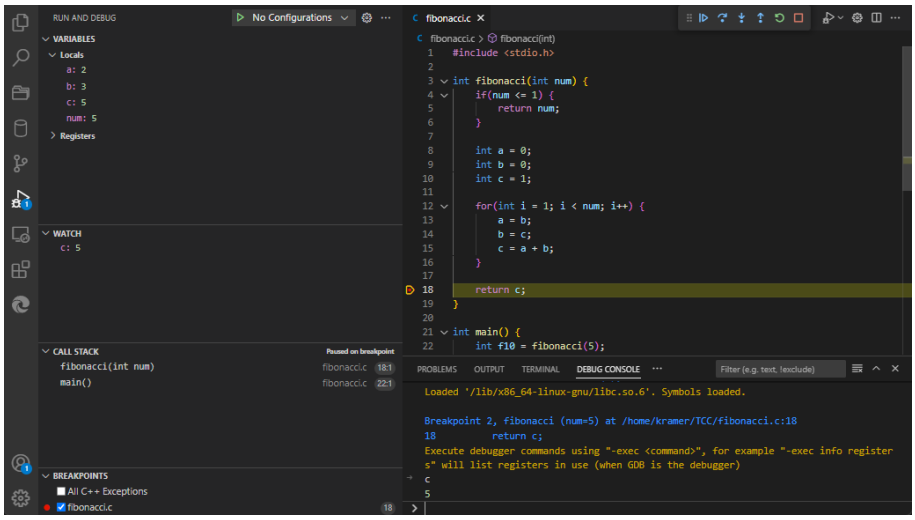



Figura 4.7: Exemplo de depuração do algoritmo de Fibonacci.

Após ter o arquivo *tasks.json* definido, vamos ao segundo ponto passo que consiste em adicionar pontos de parada no código. Basta clicar na esquerda da linha de interesse e durante a execução do código, o depurador irá parar neste ponto. Pode-se ver um breakpoint na linha 10 da figura 4.5 e um na linha 18 da figura 4.7.

Por fim, basta iniciar a execução pela aba de depuração ou simplesmente pressionando a tecla F5. É possível interagir diretamente com o depurador em tempo real, no caso da figura 4.7 o GDB. Pode-se então observar o conteúdo da memória do programa em execução nos locais onde os breakpoints estão situados.

Pode-se observar uma lista de informações que temos acesso neste ponto na figura 4.7:

- A aba de variáveis com as variáveis locais e registradores,
- Aba de variáveis observadas, monitorando a variável c,
- Pilha de chamadas,
- Lista de breakpoints.

Este mecanismo será de extrema importância para nós durante a introspecção de memória e com ele será possível ter informações o suficiente para gerar informação sobre o código em execução.

## 4.2 DEBUG VISUALIZER

Debug Visualizer (Dieterichs, 2019) é uma extensão disponível para o Visual Studio Code capaz de gerar diversas representações visuais a partir de algoritmos durante sua depuração. A ferramenta foi desenvolvida inicialmente para dar suporte a algoritmos implementados em JavaScript, mas ao longo do tempo foi expandindo para mais linguagens de programação.

Permitindo diversas visualizações, a extensão funciona interpretando um objeto JSON armazenado em alguma variável ou execução de função. Na figura 4.8, vemos ela em execução no painel da direita mostrando uma lista encadeada.

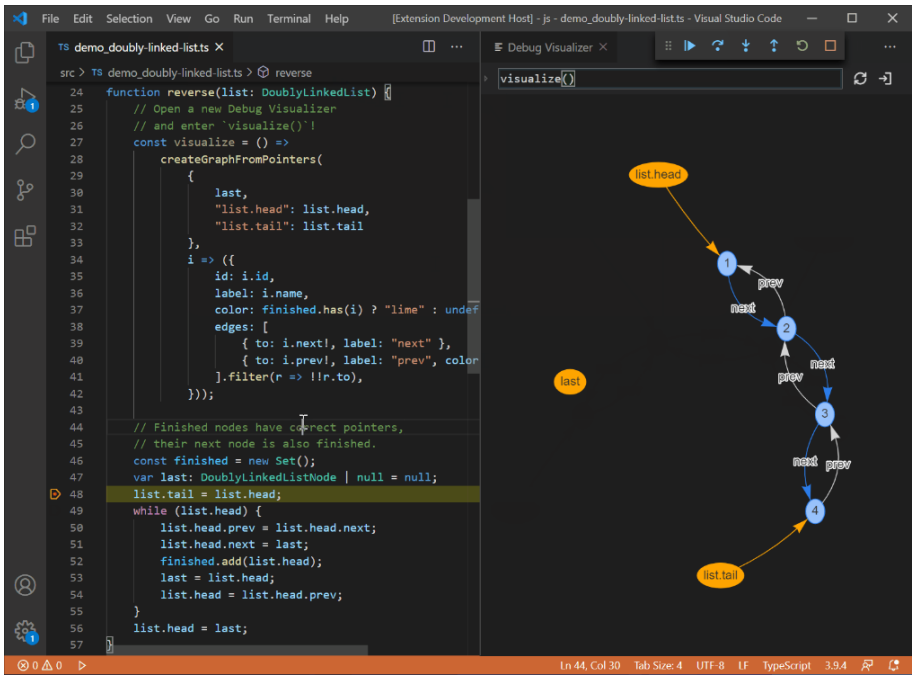



Figura 4.8: Visualização de uma lista encadeada com o Debug Visualizer. (Dieterichs, 2019)

Para utilizar a extensão, basta usar o comando "Debug Visualizer: New View" dentro do VSCode. O Debug Visualizer será então apresentado em uma janela dividida em duas partes: uma entrada de texto com um comando que será executado pelo depurador a cada breakpoint e uma visualização do que foi interpretado. Na figura 4.8, a cada pausa do depurador, o comando `visualize()` é interpretado e executa uma função de mesmo nome que retorna um objeto JSON contendo a representação de um grafo usado para gerar a visualização da lista.

A extensão ainda permite diversas outras visualizações além de grafos direcionados, por exemplo, gráficos, tabelas, código-fonte, imagens, entre outros.

### 4.2.1 Arquitetura do Debug Visualizer

A extensão consiste em 4 módulos que se comunicam internamente (figura 4.9).

O módulo da extensão tem a função principal nessa arquitetura, sendo responsável por controlar a extensão e inicializar o Webview, além de estabelecer a comunicação entre o módulo de extração de dados e o Webview.

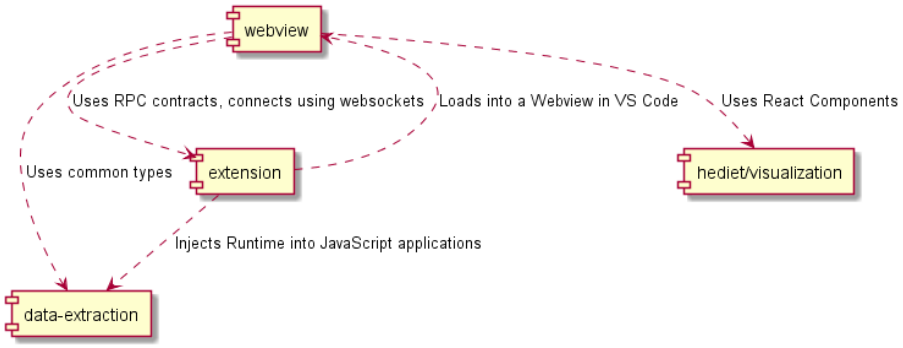



Figura 4.9: Arquitetura do Debug Visualizer. (Dieterichs, 2019)

O Webview, sendo o responsável pela estrutura de entrada e saída do sistema, começa lendo o comando que será interpretado pelo depurador. Envia-se o comando para a extensão, que o direciona para o módulo de extração de dados, que, nesse ponto, já está inicializado.

O módulo de extração de dados é iniciado logo após a depuração começar. Ele é uma classe que, por uma cadeia de responsabilidade, transmite o comando dado pelo usuário para um sub-módulo capaz de interpretá-lo usando o depurador específico configurado pelo usuário.

Após o módulo de extração de dados interpretar o comando, o resultado é enviado para a extensão, que o transmite para o Webview. Por fim, o Webview utiliza elementos desenvolvidos em React para exibir a visualização.

É importante lembrar que nem toda entrada pode ser processada. Os visualizadores consomem dados em formato JSON específicos (Dieterichs, 2019).

# 5 DETALHAMENTO

Esta seção irá apresentar os detalhes de implementação da expansão para o Debug Visualizer. Seguindo um fluxo desde a extração de dados até a representação visual dos mesmos, veremos como foi implementada a introspecção de memória com o GDB, como é construído um grafo de memória que represente estes dados e por fim, como este grafo é transformado em uma visualização interativa.

## 5.1 INTROSPECÇÃO DE MEMÓRIA USANDO GDB NO VSCODE

Como visto na seção 2.3, o GDB consegue observar o que está "dentro" de um programa em execução (Stallman et al., 2002). Podemos utilizar suas funcionalidades para extrair dados do programa recursivamente.

Dada a infraestrutura do Debug Visualizer, foi implementada a classe *CVisualizationBackend* que extende a classe *VisualizationBackendBase* responsável por interagir com o depurador e gerar um grafo de memória a partir dessa interação.

A classe é inicializada com três parâmetros. O primeiro é um proxy para a sessão atual de depuração, responsável pela interação com o GDB, ele recebe uma string que representa um comando a ser interpretado e devolve o resultado do depurador. O segundo é um proxy para a visualização do depurador, capaz de determinar o escopo léxico atual. O terceiro é a configuração atual do usuário, com o que deve ser avaliado, bem como configuração do tipo de visualização requisitado.

A introspecção de memória com o GDB se dá por meio de uma busca em largura, para cada endereço de memória avaliado itera-se sobre cada um dos seus "filhos". As funções necessárias para isso são as seguintes:

- *evaluate(expressão: string)*: a classe *CVisualizationBackend* requisita ao proxy da sessão atual de depuração que faça a avaliação da expressão utilizando diretamente ela como comando para o depurador. Isto difere do funcionamento padrão do GDB sem o MIDebugger e equivale ao usuário digitando `print <expressão>` diretamente para realizar a mesma operação.

- *getVariableAddress(variableName: string)*: a classe *CVisualizationBackend* requisita ao proxy da sessão atual de depuração que encontre o endereço de memória da variável solicitada utilizando o comando `&<variableName>`. Isto equivale ao usuário digitando `print &<variableName>` diretamente no GDB.

- *getVariableType(variableName: string)*: a classe *CVisualizationBackend* requisita ao proxy da sessão atual de depuração que encontre o tipo da variável solicitada utilizando o comando `-exec ptype <variableName>`. Isto equivale ao usuário digitando `ptype <variableName>` diretamente no GDB.

Com estas três funções, é possível recursivamente criar um grafo que representa a memória do programa em execução.

## 5.2 CONSTRUÇÃO DE UM GRAFO DE MEMÓRIA

A construção de um grafo de memória é o passo intermediário entre a coleta de dados e a geração da visualização do programa. O usuário pode digitar uma lista de variáveis, que serão utilizadas como ponto de partida para fazer a introspecção de memória.

Utilizando uma estratégia de busca em profundidade, é possível gerar um grafo que avalie toda a memória requisitada pelo usuário. O algoritmo para a execução desta busca consiste no que segue:

**Algoritmo 2** $GERA{-}GRAFO(comandos, comandos\_avaliados, vertices, arestas, matrizes)$

```
 1: for all comando em comandos do
 2:     if comando está em comandos_avaliados then
 3:         proximo comando
 4:     end if
 5:     if comando é NULL then
 6:         proximo comando
 7:     end if
 8:     if comando é primitivo then
 9:         ADICIONA-VERTICE(vertices, comando)
10:     else if comando é ponteiro then
11:         ADICIONA-VERTICE(vertices, comando)
12:         arestas.push({de : comando, para : *comando})
13:         GERA-GRAFO([*comando], comandos_avaliados, vertices, arestas, matrizes)
14:     else if comando é matriz then
15:         ADICIONA-VERTICE(vertices, comando)
16:         m = novamatriz
17:         for all elemento de comando do
18:             GERA-GRAFO([elemento], comandos_avaliados,
19:                                    vertices, arestas, matrizes)
20:             m.push(novo_vertice)
21:         end for
22:         matrizes.push(matriz)
23:     else
24:         # caso de structs
25:         ADICIONA-VERTICE(vertices, comando)
26:         for all variavel dentro da struct comando do
27:             GERA-GRAFO([comando → variavel], comandos_avaliados,
28:             arestas.push({de : comando, para : comando → variavel})
29:                                    vertices, arestas, matrizes)
30:         end for
31:     end if
32: end for
```

O algoritmo 3 é responsável por adicionar um novo vértice ao vetor de vértices que serão visualizados. Ele utiliza as funções definidas previamente para realizar a inserção.

Para melhor compreensão do algoritmo, utilizaremos um exemplo de árvore para a geração do grafo de memória. A figura 5.1 representa esta árvore. O comando que será executado no Debug Visualizer será `i, sub_tree, root_pointer`.

Inicialmente é avaliada a variável `i`, como ela é uma variável primitiva, cai no primeiro caso do algoritmo 2 e retorna. Depois disso é avaliada a variável `sub_tree`, como é um

**Algoritmo 3** $ADICIONA-VERTICE(vertices, comando)$

```
1: id = getVariableAddress(comando)
2: tipo = getVariableType(comando)
3: valor = evaluate(comando)
4: vertices.push({
5:     id : id,
6:     label : tipo + comando + " = " + valor
7: })
```

ponteiro, cai no segundo caso. O ponteiro é adicionado à lista de variáveis e uma aresta entre o ponteiro e seu valor é adicionado. Recursivamente o algoritmo GERA-GRAFO é chamado para avaliar o ponteiro armazenado por ela (`*sub_tree`).

Nessa chamada recursiva, é avaliado o nó `node_t 2`, cai então no último caso, onde cada propriedade é avaliada recursivamente novamente. Como seus filhos são variáveis nulas, não faz nada com eles. Recursivamente avalia-se a propriedade valor, adicionando à lista de vértices e criando a devida aresta.

Por fim, avalia-se a variável `root_pointer`. Sendo um ponteiro, entra no segundo caso, recursivamente avalia-se o endereço armazenado. Novamente, entra no último caso do `if`, pois a variável `node_t 2` é uma estrutura. Neste ponto, não faz nada com o filho da esquerda, pois é nulo, adiciona o conteúdo da propriedade valor. O filho da direita já foi avaliado , portanto, apenas adiciona-se o arco entre `node_t 1` e `node_t 2`.

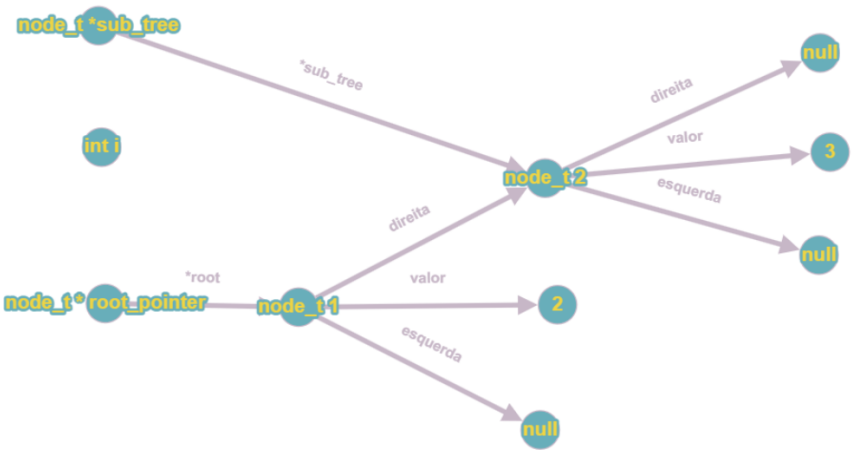



Figura 5.1: Exemplo de um grafo gerado a partir da memória de um programa.

## 5.3 TRANSFORMAÇÃO DO GRAFO DE MEMÓRIA EM UMA VISUALIZAÇÃO

Tendo a definição dos vértices, arestas e matrizes, basta compilá-los em um objeto JSON. A partir deste momento entra o visualizador que renderiza os dados em um diagrama.

Neste ponto, o Webview interpreta o arquivo JSON e utilizando uma cadeia de responsabilidade, repassa o JSON para um renderizador.

Atualmente existem dois renderizadores, um que gera grafos do Graphviz[1] e outro que gera uma visualização do Vis.JS[2]. Na interface do Debug Visualizer, o usuário consegue escolher qual irá utilizar.

No Graphviz, para adicionarmos vértices e arestas, basta utilizar a estrutura nativa dele. O Graphviz interpreta texto na linguagem DOT[3] e gera um grafo a partir disso. O Graphviz suporta a todas as estruturas necessárias, vértices, arestas e matrizes.

Um exemplo de grafo escrito com o Graphviz é o seguinte, responsável por gerar a visualização do grafo da figura 5.2:

```
digraph G {
   "0x0001" [label = "int i = 0";];
   "0x0002" [label = "node_t *sub_tree";];
   "0x0003" [label = "node_t *root_pointer";];
   "0x0004" [label = "node_t 1";];
   "0x0005" [label = "node_t 2";];
   "0x0006" [label = "2";];
   "0x0007" [label = "3";];

   "0x0002" -> "0x0005" [label = "*sub_tree";];

   "0x0005" -> "null_2" [label = "esquerda";];
   "0x0005" -> "0x0007" [label = "valor";];
   "0x0005" -> "null_3" [label = "direita";];

   "0x0004" -> "0x0006" [label = "valor";];
   "0x0004" -> "0x0005" [label = "direita";];
   "0x0004" -> "null_1" [label = "esquerda";];

   "0x0003" -> "0x0004" [label = "*root_pointer";];

   "null_1" [label = "null";];
   "null_2" [label = "null";];
   "null_3" [label = "null";];
}
```

Nativamente o Vis.JS consegue interpretar o JSON gerado pelo passo anterior. No entanto, não consegue interpretar matrizes e vetores. Sendo uma estrutura dinâmica que contém algoritmos gravitacionais, é necessário criar formas de implementar isto.

A forma encontrada foi desativar a gravidade para cada integrante da matriz/vetor e gerar um vértice maior que encapsula os elementos. Assim, quando o vértice ao redor se move, todos os elementos internos também se movem junto. Simultaneamente, se algum elemento da matriz/vetor se move, todos os outros elementos e o vértice ao redor também se movem.

Exemplos de visualização com o GraphViz serão apresentadas na etapa de validação da expansão.

[1] https://graphviz.org/

[2] https://visjs.org/

[3] https://graphviz.org/doc/info/lang.html

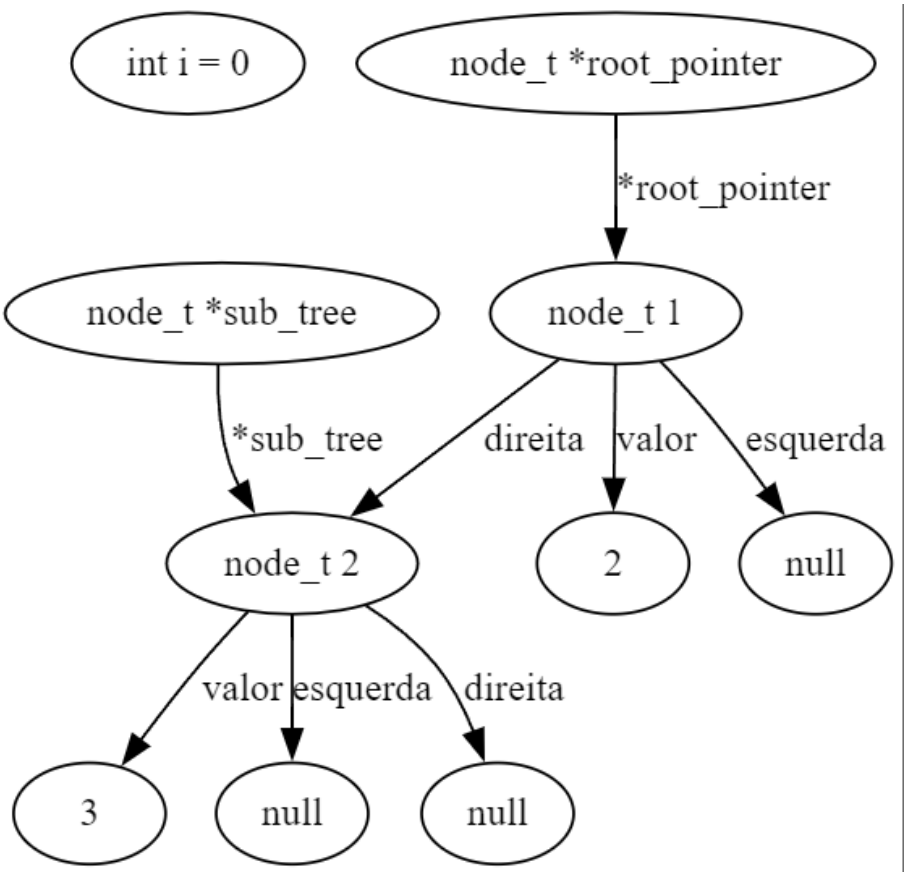



Figura 5.2: Visualização de memória com GraphViz.

# 6 RESULTADOS OBTIDOS

Após completar o desenvolvimento, foi possível visualizar diversos algoritmos, esta seção irá apresentar a visualização de alguns algoritmos conhecidos no meio acadêmico, bem como mostrar as limitações e formas corretas de implementação para alcançar os melhores resultados possíveis.

## 6.1 BST - BINARY SEARCH TREE

Para uma primeira validação, foi implementado o algoritmo visto na seção de proposta (rotação à esquerda). Durante a depuração do código, os resultados obtidos foram satisfatórios e conseguiram representar a árvore utilizada, da maneira como foi proposta. Para a execução do algoritmo, utilizou-se a seguinte struct:

```
1 struct node_t {
2    int value;
3    struct node_t *left;
4    struct node_t *right;
5 };
```

A seguir estão os resultados alcançados antes e depois das rotações, tanto no formato do GraphViz quanto do Vis.JS.

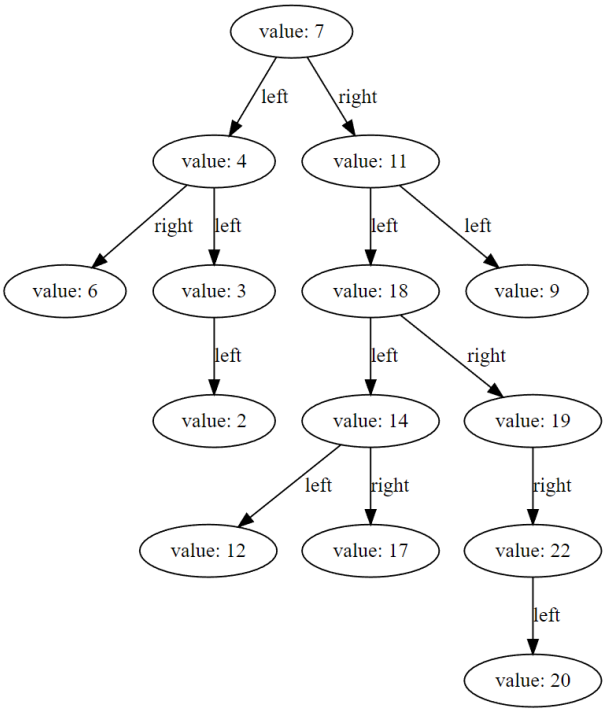



Figura 6.1: Árvore antes da rotação visualizada com o GraphViz.

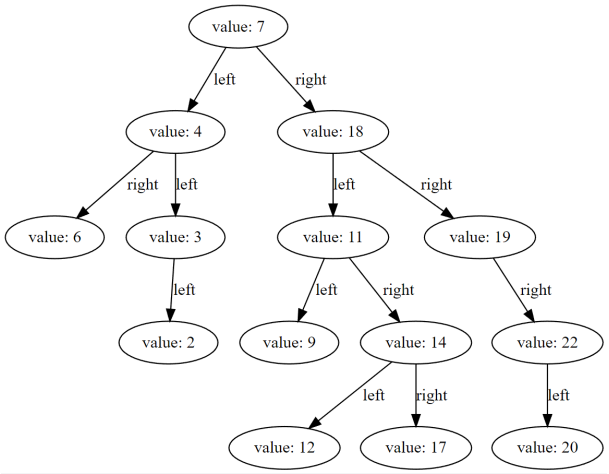



Figura 6.2: Árvore após a rotação visualizada com o GraphViz.

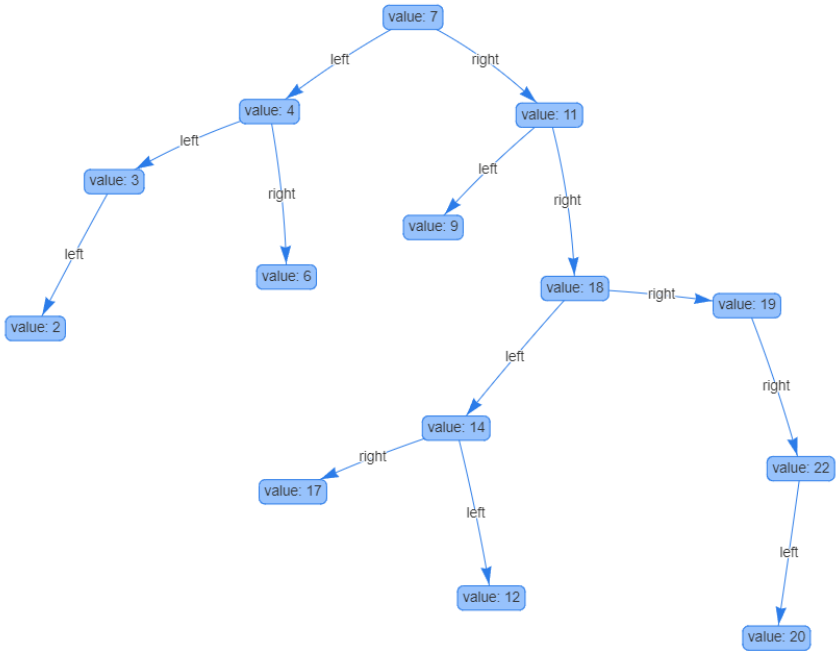



Figura 6.3: Árvore antes da rotação visualizada com o Vis.JS.

É importante lembrar que o algoritmo não tem como saber o que é esquerda e direita. Por este motivo, algumas vezes as visualizações invertem os lados.

Dado o nível de semelhança com o diagrama apresentado no livro “Algoritmos: teoria e prática” (Cormen, 2012), podemos concluir que a implementação conseguiu alcançar o resultado almejado.

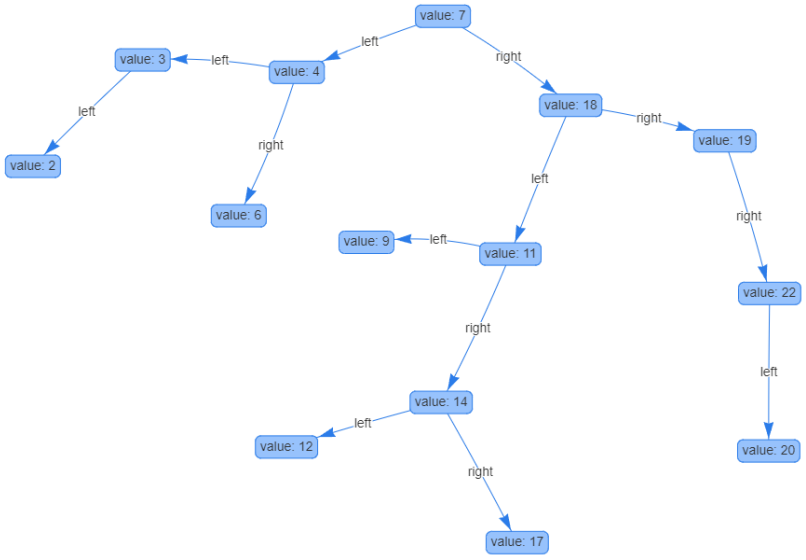



Figura 6.4: Árvore após a rotação visualizada com o Vis.JS.

## 6.2 VISUALIZAÇÃO DE PONTEIROS, VETORES E MATRIZES

Diversas vezes o aluno não consegue compreender como funcionam ponteiros. A percepção de que o ponteiro é um endereço de memória por vezes não é fácil de se entender em um primeiro momento. Por este motivo, a visualização gerada pelo Debug Visualizer consegue elucidar como os ponteiros funcionam. Podemos observar na figura 6.5 o comportamento de um ponteiro, por ela compreendemos que um ponteiro guarda um endereço de memória que pode ser seguido.

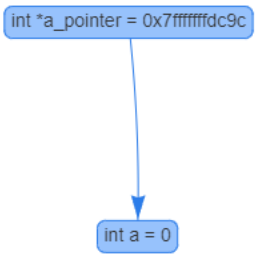



Figura 6.5: O ponteiro *a_pointer* aponta para a variável *a*.

A visualização de vetores e matrizes, apresenta duas possíveis visualizações. Quando o dado armazenado neles é considerado um tipo primitivo e não é uma string, então a apresentação dos dados é realizada diretamente nas células da representação. Na figura 6.6, podemos ver a matriz `int identity[3][3]`, preenchida com os valores de uma matriz identidade.

No caso de um vetor ou matriz com estruturas de dados complexas, ou endereços de memória, a visualização é feita por meio de arestas com a indexação do dado armazenado. A figura 6.7 mostra a visualização da struct abaixo, onde a raiz, do tipo `employee_t` tem três filhos.

```
1 struct person_t {
```

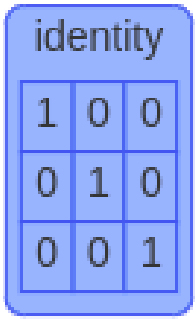



Figura 6.6: Representação da matriz identidade *int identity[3][3]*.

```
2     int age;
3 }
4
5 struct employee_t {
6     person_t children[3];
7 }
```

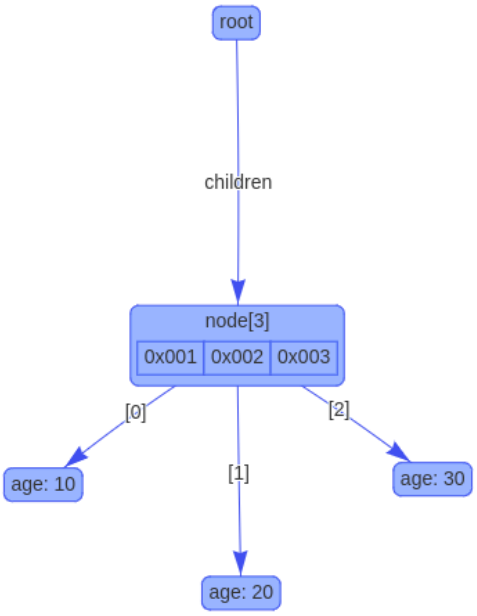



Figura 6.7: Representação de uma estrutura de dados contendo um vetor.

Dado que a variável do vetor armazena o endereço de memória do primeiro elemento, podemos então representá-lo separadamente da struct, facilitando a visualização e a compreensão.

## 6.3 VISUALIZAÇÃO DE FALHAS DE SEGMENTAÇÃO

Durante os testes, uma nova funcionalidade apareceu. A extensão apresentou a capacidade de trazer informações quando existem falhas de segmentação, as quais são tentativas de acesso a um endereço de memória não permitido.

Em certa implementação de uma árvore de busca binária, verificou-se que durante a remoção de nós o programa em execução apresentava falha de segmentação. Utilizando-se o Debug Visualizer no momento em que ocorreu a falha, apesar da memória corrompida, observou-se a figura 6.8.

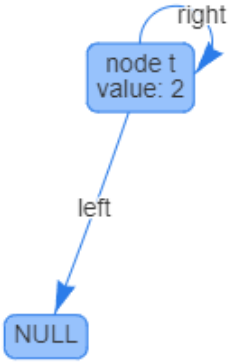



Figura 6.8: Visualização de memória corrompida.

Podemos observar que o ponteiro da direita está apontando para o próprio nó, o que não deveria acontecer. Devido a isto, constatou-se que em algum ponto da remoção de nós, o ponteiro da direita apontava para o lugar errado. Tendo esta informação, permitiu-se limitar as possibilidades da origem do erro simplesmente para a função que removia o nó e corrigia os ponteiros.

É importante frisar que durante uma falha de segmentação, a memória do programa está geralmente corrompida e, apesar de poder apresentar uma boa representação em alguns casos, isso não será necessariamente verdade para toda falha de segmentação.

## 6.4 FACILIDADE DE USO E LIMITAÇÕES

Por fim, foi possível tornar o Debug Visualizer mais simples e intuitivo para a proposta deste trabalho, bastando selecionar a variável a ser observada e adicionar os breakpoints no código. Porém, devido ao comportamento gravitacional intrínseco das animações do Vis.JS, a visualização normalmente não é apresentada da melhor forma possível, sendo necessário ao usuário mover os vértices para compreender o que acontece com as estruturas de dados. Isso pode ser ilustrado na figura 6.9, que apresenta o estado inicial da figura 6.3. Para chegar na visualização final vista anteriormente, foi necessário, através da ação de clicar e arrastar, ajustar a posição dos nós da árvore.

Apesar da alta capacidade de generalização da extensão, visualizações específicas não são suportadas por esta implementação. Por exemplo, no caso de uma árvore red-black, não é possível representar os nós com cores distintas utilizando esta estrutura.

Além disso, algumas estruturas de representação não foram implementadas, como é o caso da iteração de vetores. Considere o seguinte algoritmo para a impressão de um vetor apresentado na seção 2.3.1.

```
1 void print_arr(int *values, int arr_len)
2 {
3     for(int i = 0; i < arr_len; i++) {
```

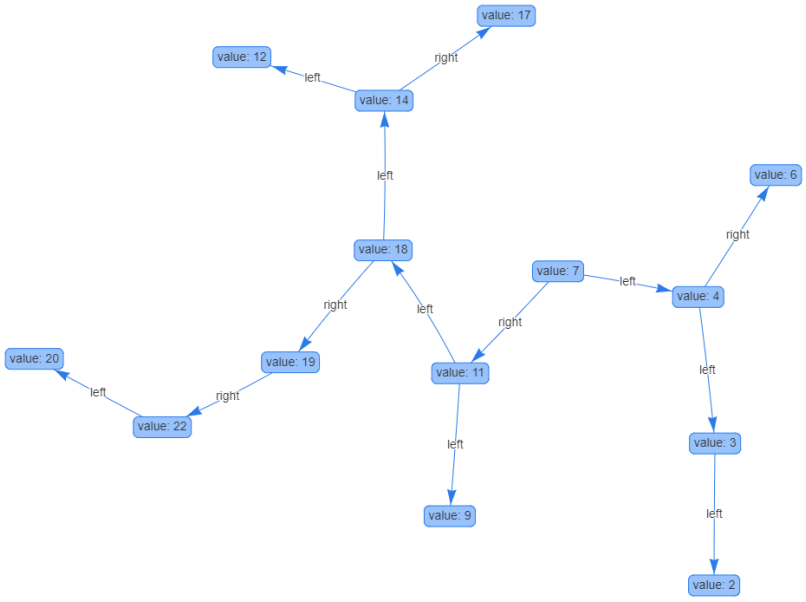



Figura 6.9: Visualização inicial da figura 6.3.

```
4        printf("%d ", values[i]);
5    }
6
7    printf("\n");
8 }
```

Com a implementação feita, não é possível ter uma boa visualização do vetor com a variável `i` apontando para a posição atual no vetor.

## 6.5 CONSIDERAÇÕES

Após a análise dos resultados obtidos, podemos concluir que foi feita uma ferramenta capaz de superar aquelas apresentadas no capítulo 3.

A vantagem do Debug Visualizer neste ponto, além da possibilidade de realizar diversas visualizações, é que se trata de uma ferramenta integrada a um ambiente de desenvolvimento popular e moderno, o Visual Studio Code. Desta forma, o aluno utilizando ela pode conhecer outras extensões que podem facilitar e melhorar seu aprendizado e produtividade.

Além disso, a integração com o GDB abre um leque de possibilidades de expansão para outras linguagens de programação. Mais ainda, por ser uma ferramenta fácil de usar com uma arquitetura simples, apresenta alta capacidade de generalização.

A visualização dinâmica, mesmo em face da dificuldade de ajustá-la algumas vezes, mostra-se uma ótima ferramenta de desenvolvimento. O programador não tem apenas o depurador ao seu lado, mas também um feedback visual sobre o que está fazendo no seu código, auxiliando na tomada de decisões e facilitando o desenvolvimento e depuração.

# 7 CONCLUSÃO

Podemos ver por meio deste trabalho que é possível gerar conhecimento a partir de dados armazenados na memória de programas em execução. Podemos entender como é feito o processo de coleta de dados de algoritmos e como mostrá-los de forma didática e útil.

Foram vistas algumas ferramentas de visualização, tais como o DDD, que foi a principal inspiração para esta pesquisa. Além de como pode-se utilizar o Visual Studio Code com o GDB para extrair informações úteis de programas em execução.

Enfim, apresentou-se a proposta de expandir o plugin Debug Visualizer, para gerar representações a partir de dados coletados pelo GDB e assim foi feito.

Através da coleta de dados utilizando o adaptador CPPDBG pode-se construir um grafo que represente a memória do programa recursivamente. Então, com este grafo, é possível utilizar tanto o GraphViz quanto o Vis.JS para criar visualizações dos algoritmos.

Com isto, foi possível analisar como podemos utilizar essas visualizações para mostrar informações úteis sobre o algoritmo sendo executado como, por exemplo, suas estruturas de dados, eventuais problemas relacionados a falhas de segmentação, entre outros.

## 7.1 CONTRIBUIÇÕES

A extensão Debug Visualizer é disponibilizada na loja de extensões do Visual Studio Code[1]. A partir deste documento e do código implementado, será feita uma requisição para integrar os resultados obtidos na extensão, então será possível utilizar estas funcionalidades diretamente.

Além disso, pode-se acessar o código implementado neste projeto no GitHub[2].

## 7.2 TRABALHOS FUTUROS

Dado que a extensão é de código-fonte aberto e o criador da extensão, Henning Dieterichs, aceita ativamente novas contribuições, ficam como sugestões de trabalhos futuros:

- A implementação de visualizações específicas, como de árvores red-black.
- Novas expansões para a extensão, como adição de suporte a mais linguagens de programação.
- A criação de uma ferramenta semelhante ao Debug Visualizer do zero.

[1] https://marketplace.visualstudio.com/items?itemName=hediet.debug-visualizer
[2] https://github.com/kramer2005/vscode-debug-visualizer

# REFERÊNCIAS


Bavoil, L., Callahan, S., Crossno, P., Freire, J., Scheidegger, C., Silva, C. e Vo, H. (2005). Vistrails: enabling interactive multiple-view visualizations. Em *VIS 05. IEEE Visualization, 2005.*, páginas 135–142.

Card, S., Card, M., Mackinlay, J. e Shneiderman, B. (1999). *Readings in Information Visualization: Using Vision to Think*. Interactive Technologies. Elsevier Science.

Chen, M., Ebert, D., Hagen, H., Laramee, R. S., van Liere, R., Ma, K.-L., Ribarsky, W., Scheuermann, G. e Silver, D. (2009). Data, information, and knowledge in visualization. *IEEE Computer Graphics and Applications*, 29(1):12–19.

Cormen, T. (2012). *Algoritmos: teoria e prática*. Campus.

Dieterichs, H. (2019). Debug visualizer. `https://marketplace.visualstudio.com/items?itemName=hediet.debug-visualizer`. Acessado em 19/02/2023.

Gales, D. (2012). Data structure visualizations. https://www.cs.usfca.edu/ galles/visualization/Algorithms.html. Acessado em 2022-08-09.

Guo, P. (2013). Online python tutor: Embeddable web-based program visualization for cs education. Em *Proceedings of the ACM Technical Symposium on Computer Science Education (SIGCSE)*.

HALIM, S. (2015). Visualgo – visualising data structures and algorithms through animation. *OLYMPIADS IN INFORMATICS*, 9:243–245.

Microsoft (2023). *Debugging*. Microsoft Corporation. `https://code.visualstudio.com/docs/editor/debugging`. Acessado em 26/02/2023.

Park, J. (2016). Algorithm visualizer. https://algorithm-visualizer.org/. Acessado em 2022-08-09.

Santos, R. d. e Costa, H. (2006). TBC-AED e TBC-AED/WEB: Um desafio no ensino de algoritmos, estruturas de dados e programação.

Santos, S. R. d. (2013). Visualização de memória utilizando java reflection. Dissertação de Mestrado, Mestrado em Ciência da Computação - Universidade do Estado do Rio Grande do Norte, Mossoró - RN.

Stallman, R., Pesch, R., Shebs, S. e Free Software Foundation (Cambridge, M. (2002). *Debugging with GDB: The GNU Source-level Debugger*. A GNU manual. Free Software Foundation.

Zeller, A. (2004). *Debugging with DDD*.

Zeller, A. e Lütkehaus, D. (1995). Data display debugger (gnu ddd). https://www.gnu.org/software/ddd/. Acessado em 2022-08-09.