Teste Dinâmico de Software

Mestrado em Engenharia Informática
Qualidade e Confiabilidade de Software 2023/2024

Versão do Documento: 1.0

Bruno Sequeira 2020235721, brunosequeira@student.dei.uc.pt
Rui Santos 2020225542, rpsantos@student.dei.uc.pt

Universidade de Coimbra

Contents

List of Figures

List of Tables

1 Introdução

Este projeto consiste no desenvolvimento de um plano de teste de software de modo a testar, avaliar e testar um software implementado por outros desenvolvedores.

O principal objetivo é identificar e selecionar corretamente abordagens de testes dinâmicoss, tendo em conta testes de White Box e o que este conceitos pode concluir ao executar um plano de teste de software.

Tal como referido acima, escolhemos dois produtos de software, que são os seguintes:

- **Jogo do Sudoku**: Este código iré tentar resolver uma tela 9x9 com as regras do sudoku, sendo que a função que irá abordar essa implementação é a **Solve()**, está irá percorrer posição a posição, usando um algoritmo de back-tracking para no final retornar um boleano com o resultado, sendo que se for true então foi possível a sua realização, sendo possível visualizar o seu resultado. O input é um array bi-dimensional, e o output é a solução deste array.
- **Algoritmo de Dijkstra**: Este software consiste em encontrar os caminhos mais curtos entre vértices consoante as ligações entre eles. O input é a criação de um grafo e escolhendo um vértice como origem. O resultado final é um array com as distâncias entre o ponto de origem com todos os outros vértices.

Na secção 2 serão apresentados os aspetos de risco de software a ser testado e potenciais riscos. Na secção 3 serão apresentados os elementos e funcionalidades a serem testados e na secção 4 serão descritos os elementos e funcionalidades que não serão testados, assumindo que estes pontos esão corretos e bem implementados.
Na secção 5 será apresentado os planos de testes, na secção 6 serão definidos os critérios de realização do plano de testes. Na secção 7 serão identificados todos os elementos que foram entregues como parte e consequência do plano de testes. Na secção 8 apresentaremos as necessidades específicas para execução dos testes. Na secção 9 a distribuição do trabalho por elemento. E por fim, a secção 10, será apresentado as conclusões para os resultados dos testes, descrevendo os defeitos identificados.

2 Aspetos de risco do software

Neste trabalho, a seleção criteriosa e a qualidade dos casos de teste assumem um papel crucial, visto que o código em questão foi desenvolvido para fins académicos e pessoais, sem documentação oficial além do código-fonte fornecido. Essa falta de documentação pode dificultar a compreensão inicial do fluxo do programa.

Além disso, a possibilidade de bus no código levanta a necessidade de testes mais rigorosos para garantir a validade dos resultados.

Por outro lado, visto que é apenas um ficheiro Java, a simplicidade do software elimina a preocupação com dependências externas que possam afetar o seu funcionamento.

3 Elementos e funcionalidades a serem testadas

Para o software encontrado, iremo-nos focar na função **Solve** e **Dijkstra**, sendo que estas irão ser analisadas e serão realizados testes do tipo White Box separadamente em cada função, iremos abordar o control flow e o data flow.

3.1 Elementos

Da função solve iremos também testar as suas variáveis para o data flow. As variáveis são as seguintes:

- board - (global)
- i
- j
- num

Da função Dijkstra iremos também testar as suas variáveis para o data flow. As variáveis são as seguintes:

- graph - (global)
- visited
- numVertices
- priorityQueue
- newDistance
- distance
- source - (global)
- currentVertex
- edges
- edge
- vertices
- i

3.2 Funcionalidades

- Funcionalidade básica: Testes para garantir que o algoritmo funcione corretamente em situações ideais.
- Testes para verificar se o algoritmo lida corretamente com casos especiais, como grafos com ciclos negativos, grafos não conectados, ou casos em que o vértice de destino não é alcançável a partir do vértice de origem.
- Verificar se as variáveis são manipuladas e transformadas corretamente em cada módulo.

4 Elementos e funcionalidades a não serem testadas

Baseado nas nossas funções, temos alguns elementos e funcionalidades que não terão o nosso foco.

4.1 Elementos

Os elementos da função dijkstra são:

- printSolution() - Baixa complexidade
- distancesV() - Baixa complexidade
- addEdge() - 2 linhas de código
- getEdges() - 1 linha de código.

O elementos da função Solve é:

- isPossible() - Baixa Complexidade

Em relação às variáveis, abordaremos todas, pois as que são utilizadas nas funções são usadas nas funções externas.

4.2 Funcionalidades

Existem algumas funcionalidades que não iremos focar, que são as seguintes.

- A complexidade do código.
- O tempo que leva para o software ser executado.
- E as funcionalidades das funções referidas acima.

5 Abordagem dos testes

5.1 Testes White Box

Nesta secção estão presentes os testes de White Box para as funções **Dijkstra()**, do projeto *Algoritmo de Dijkstra*, e **Solve()** do projeto *Sudoku*.

As técnicas de White Box analisam as estruturas internas, as estruturas de dados usadas, o design interno, a estrutura do código e o funcionamento do software, em vez de apenas a funcionalidade como no teste de caixa preta.

Esta abordagem faz a testagem por partes, ou seja, aos detalhes na implementação do código em análise e foca-se nos testes de *Control-Flow*, onde foram analisados todos os caminhos independentes e implementados casos de testes para cada um.

5.1.1 Control Flow

O fluxo de controlo é fundamental no contexto do teste de software. Este se refere à maneira como o controlo é direcionado ou manipulado dentro de um programa ou sistema durante a execução dos testes. O objetivo principal do fluxo de controlo nos testes de software é garantir que todas as partes do código sejam testadas de maneira adequada e eficiente, identificando falhas e garantindo a qualidade do software.

Para realização deste método, iremos apresentar passo a passo, a implementação deste. Projetando o grafo de Control Flow de cada função, determinamos a complexidade ciclomática, analisamos os caminho independentes possíveis e verificamos quais deles são possíveis de serem executados. Com isso, iremos determinar casos de teste para cada um dos caminhos encontrados.

- **Função Dijkstra()**

  1. **Grafo de Control Flow**

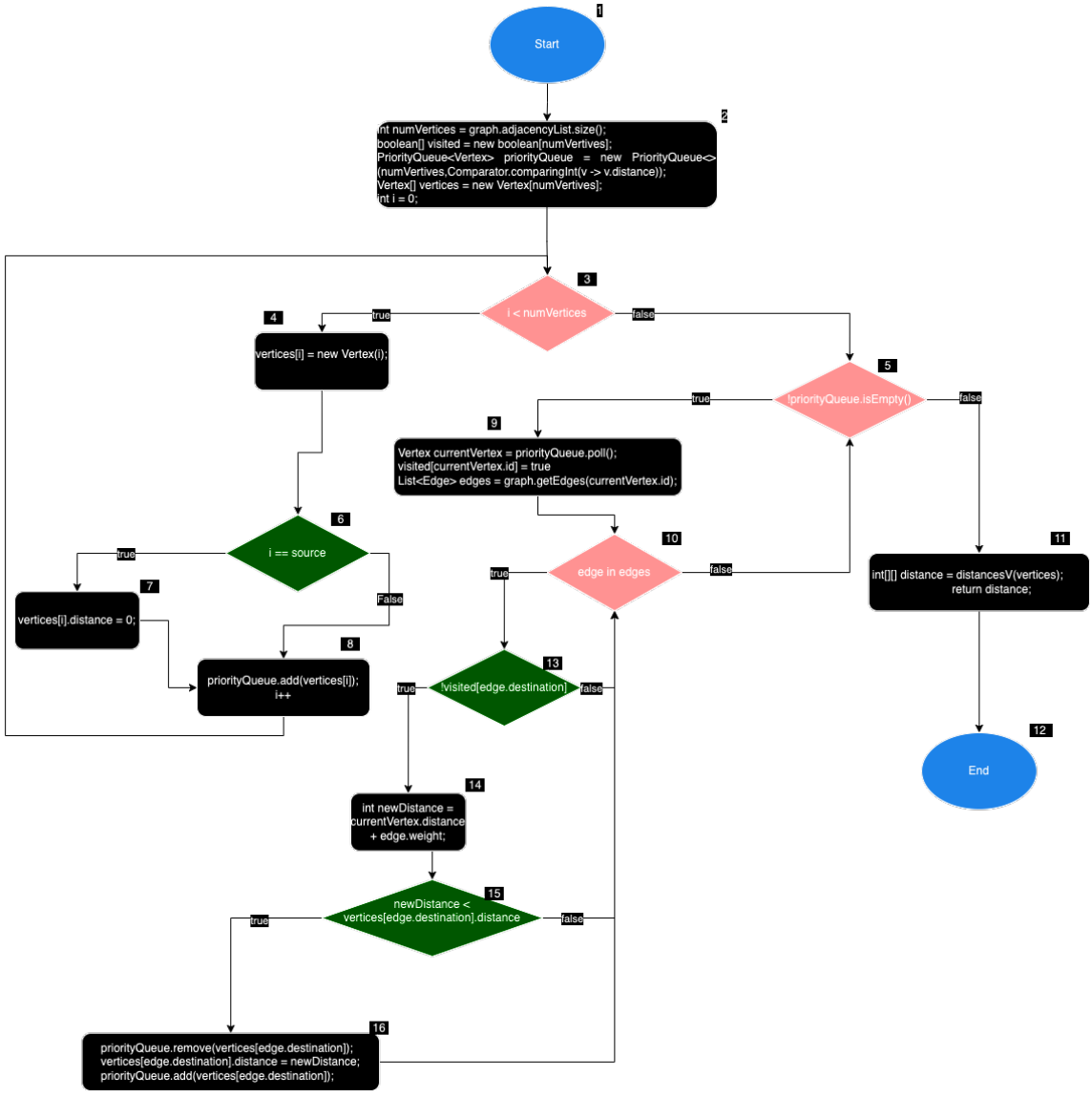

Figure 1: Control Flow - função Dijkstra.

  2. **Complexidade Ciclomática V(G)** A complexidade ciclomática é utilizada para determinar o número máximo de caminhos independentes do programa.
  A fórmula usada é: $\mathbf{V(G) = P + 1}$.

O **P** corresponde a nós predicativos.

Nós predicativos são os que têm vários arcos de saída, ou seja, corresponde a condições do programa, tais como, if's, while's e for's.

Logo, ao analisar a figura de cima, podemos verificar que existem 6 nós predicativos, send o 3 de l es if's (cor verde) e 3 de l es while's ou for's (cor rosa).

Portanto a complexidade ciclomática é de 6 + 1 = 7. O que implica que existem no máximo 7 caminhos independentes.

3. **Caminhos linearmente independentes**

Um caminho linearmente independente é uma sequência de estados de um programa que não pode ser formada combinando outras sequências de estados já testadas, ou seja, é uma sequência única de instruções que representa uma linha de execução distinta no programa. Testar caminhos linearmente independentes é importante para garantir uma cobertura abrangente do código.

Os próximos pontos são os 7 caminhos independentes que encontramos para o código Dijkstra.

- **P1** = Start, 2, 3, 5, 11, End
- **P2** = Start, 2, 3, 4, 6, 8, 3, 5, 11, End
- **P3** = Start, 2, 3, 4, 6, 8, 3, 5, 9, 10, 5, 11, End
- **P4** = Start, 2, 3, 4, 6, 7, 8, 3, 5, 9, 10, 5, 11, End
- **P5** = Start, 2, 3, 4, 6, 7, 8, 3, 5, 9, 10, 13, 10, 5, 11, End
- **P6** = Start, 2, 3, 4, 6, 7, 8, 3, 5, 9, 10, 13, 14, 15, 10, 5, 11, End
- **P7** = Start, 2, 3, 4, 6, 7, 8, 3, 5, 9, 10, 13, 14, 15, 16, 10, 5, 11, End

NOTA: Start e End correspondem aos estados 1 e 12, respetivamente.

Destes 7 caminhos linearmente independentes, 2 deles não são executáveis, estes 2 são **P1** e **P2**.

Apresentamos agora as razões pelas quais estes não são executáveis são as seguintes:

- **P1** = Neste caso, seria necessário que o grafo não contivesse nenhum elemento, mas para que a funcionalidade da função resulte seria necessário um vértice source, logo o grafo teria de ter pelo menos 1 elemento, só que este iria passar do estado 3 para o estado 4, o que não é o que este caminho deseja. Não sendo possível ser executado.

- **P2** = Neste caminho, se o grafo tem de ter pelo menos um elemento, então a fila de vértices é sempre pelo menos um elemento, portanto é impossível a condição do estado 5 ser falsa, visto que no estado 8 é adicionado o elemento na fila.

4. **Casos de teste**

Para cada caminho executável iremos apresentar os casos de usos, com o input e o output desejado. O input são várias variáveis, o *numVertices* corresponde ao número de vértices

que estão presentes no grafo, o *graph* é o grafo que foi criado com o número de vértices anteriormente apresentada. E o *distance* é o que vai conter o output da função chamada que tem como parâmetros o grafo e o vértice de origem para calcular as distâncias.

O output é um array bi-dimensional que contém as distâncias entre o vértice origem com todos o outros vértices que estão presentes no grafo anteriormente criado.

Caminho	**Input**	**Output Esperado**

P3	int numVertices = 1;
	Graph graph = new Graph(numVertices);
	int[][] distance = dijkstra(graph, 1);
		0 - 2147483647

P4	int numVertices = 1;
	Graph graph = new Graph(numVertices);
	int[][] distance = dijkstra(graph, 0);
		0 - 0

P5	int numVertices = 5;
	Graph graph = new Graph(numVertices);
	graph.addEdge(1,2,1);
	graph.addEdge(2,4,5);
	graph.addEdge(2,3,10);
	graph.addEdge(3,4,3);
	int[][] distance = dijkstra(graph, 0);
		0 - 0
		1 - 2147483647
		2 - 2147483647
		3 - 2147483647
		4 - 2147483647

P6	int numVertices = 2;
	Graph graph = new Graph(numVertices);
	graph.addEdge(0, 1, Integer.MAX_VALUE);
	int[][] distance = dijkstra(graph, 0);
		0 - 0
		1 - 2147483647

P7	int numVertices = 5;
	Graph graph = new Graph(numVertices);
	graph.addEdge(0,2,1);
	graph.addEdge(0,1,10);
	graph.addEdge(2,4,5);
	graph.addEdge(2,3,10);
	graph.addEdge(3,4,3);
	int[][] distance = dijkstra(graph, 0);
		0 - 0
		1 - 10
		2 - 1
		3 - 11
		4 - 6

Table 1: Casos de teste para os caminhos independentes do Dijkstra.

- **Sudoku**

1. **Grafo de Control Flow**

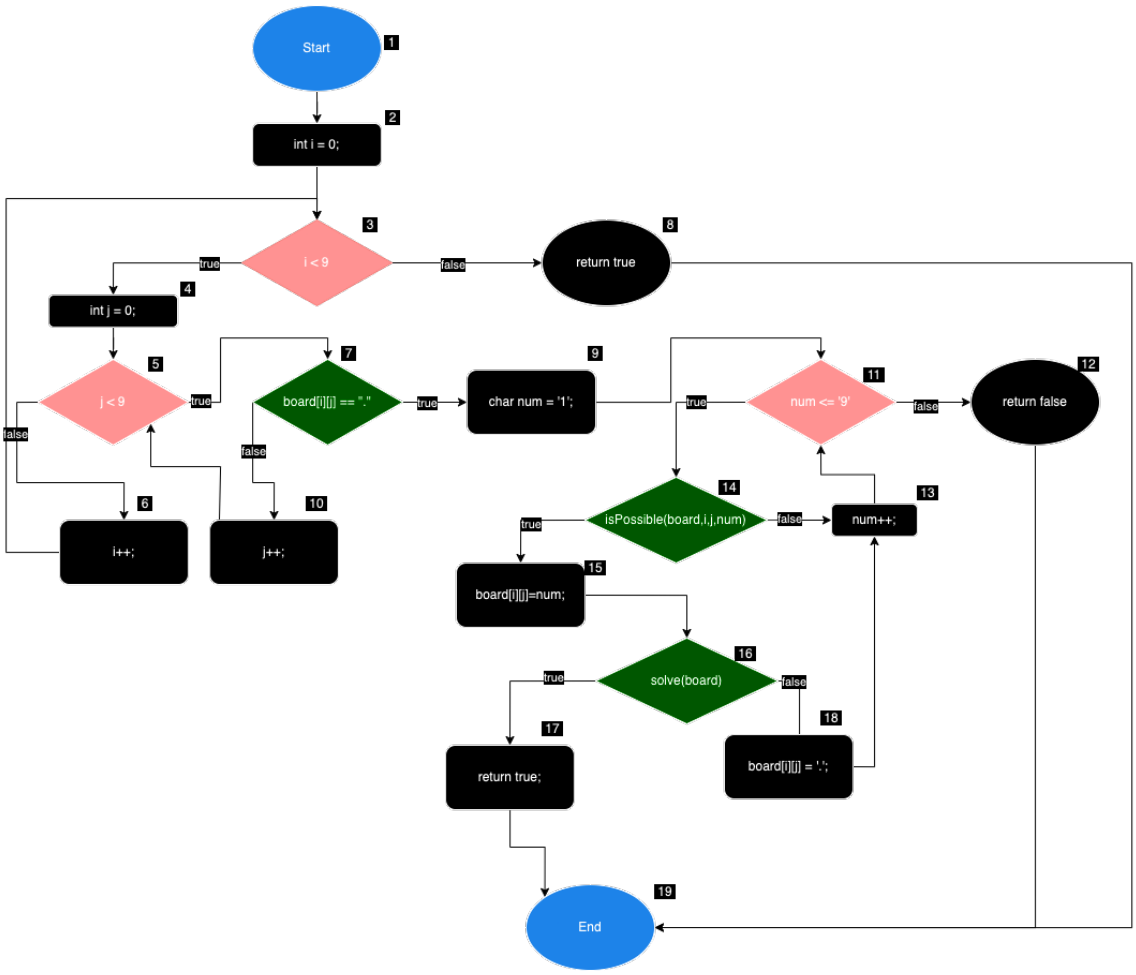

Figure 2: Control Flow - função Solve.

2. **Complexidade Ciclomática V(G)**

A complexidade ciclomática é utilizada para determinar o número máximo de caminhos independentes do programa.
A fórmula usada é: $\mathbf{V(G) = P + 1}$.

O **P** corresponde a nós predicativos.

Nós predicativos são os que têm vários arcos de saída, ou seja, corresponde a condições do programa, tais como, if's, while's e for's.

Logo, ao analisar a figura de cima, podemos verificar que existem 6 nós predicativos, send o 3 de l es if's (cor verde) e 3 de l es while's ou for's (cor rosa).

Portanto a complexidade ciclomática é de 6 + 1 = 7. O que implica que existem no máximo 7 caminhos independentes.

3. **Caminhos linearmente independentes**

Testar caminhos linearmente independentes é importante para garantir uma cobertura abrangente do código.

Os próximos pontos são os 6 caminhos independentes que encontramos para o código Solve do código Sudoku.

- **P1** = Start, 2, 3, 5, 8, End
- **P2** = Start, 2, 3, 4, 5, 7, 10, 5, 6, 3, 8, End
- **P3** = Start, 2, 3, 4, 5, 7, 9, 11, 14, 15, 16, 17, End
- **P4** = Start, 2, 3, 4, 5, 7, 9, 11, 14, 13, 11, 12, End
- **P5** = Start, 2, 3, 4, 5, 7, 9, 11, 14, 13, 11, 14, 15, 16, 18, 13, 11, 14, 13, 11, 12, End
- **P6** = Start, 2, 3, 4, 5, 7, 10, 5, 6, 3, 4, 5, 7, 10, 5, 7, 9, 11, 14, 13, 11, 14, 15, 16, 17, End

Destes 6 caminhos linearmente independentes, 1 deles não é executáveis, o **P1**.

Apresentamos agora as razões pelas quais estes não são executáveis são as seguintes:

- **P1** = Neste caso, não encontramos nenhum caso de teste para realização deste caminho, visto que a variável **i** é inicializada por 0, e a condição seguinte é sempre verdadeira, o que torna assim um caminho **unfeasable**.

4. **Casos de teste**

Para cada caminho executável iremos apresentar os casos de usos, com o input e o output desejado. Apresentamos o número do caminho linearmente independente, com o input neste caso o input é o conteúdo da variável **board** do código que foi submetido do Sudoku, o output é o resultado esperado. Sendo que sempre que existe uma solução este algoritmo altera o board, mas se não encontrar nenhuma solução este irá apresentar o mesmo board que foi submetido.

Caminho	Input - Este input é correspondente à variável board.	Output Esperado
P2	'5', '3', '4', '6', '7', '8', '9', '1', '2', '6', '7', '2', '1', '9', '5', '3', '4', '8', '1', '9', '8', '3', '4', '2', '5', '6', '7', '8', '5', '9', '7', '6', '1', '4', '2', '3', '4', '2', '6', '8', '5', '3', '7', '9', '1', '7', '1', '3', '9', '2', '4', '8', '5', '6', '9', '6', '1', '5', '3', '7', '2', '8', '4', '2', '8', '7', '4', '1', '9', '6', '3', '5', '3', '4', '5', '2', '8', '6', '1', '7', '9';	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9
P3	'.', '.';	1 2 3 4 5 6 7 8 9 4 5 6 7 8 9 1 2 3 7 8 9 1 2 3 4 5 6 2 1 4 3 6 5 8 9 7 3 6 5 8 9 7 2 1 4 8 9 7 2 1 4 3 6 5 5 3 1 6 4 2 9 7 8 6 4 2 9 7 8 5 3 1 9 7 8 5 3 1 6 4 2
P4	'.', '3', '4', '5', '7', '8', '9', '1', '2', '6', '7', '2', '1', '9', '5', '3', '4', '8', '1', '9', '8', '3', '4', '2', '5', '6', '7', '8', '5', '9', '7', '6', '1', '4', '2', '3', '4', '2', '6', '8', '5', '3', '7', '9', '1', '7', '1', '3', '9', '2', '4', '8', '5', '6', '9', '6', '1', '5', '3', '7', '2', '8', '4', '2', '8', '7', '4', '1', '9', '6', '3', '5', '3', '4', '5', '2', '8', '6', '1', '7', '9';	. 3 4 5 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9
P5	'.', '.', '4', '6', '7', '8', '9', '1', '2', '.', '7', '2', '1', '9', '5', '3', '4', '8', '1', '9', '8', '3', '4', '.', '5', '6', '7', '8', '5', '.', '7', '6', '1', '4', '2', '3', '2', '2', '6', '8', '5', '3', '.', '.', '1', '7', '1', '3', '9', '2', '4', '8', '5', '6', '9', '.', '1', '.', '3', '.', '2', '8', '4', '.', '8', '7', '4', '1', '9', '6', '3', '5', '3', '4', '5', '2', '.', '6', '1', '7', '9';	. . 4 6 7 8 9 1 2 . 7 2 1 9 5 3 4 8 1 9 8 3 4 . 5 6 7 8 5 . 7 6 1 4 2 3 2 2 6 8 5 3 . . 1 7 1 3 9 2 4 8 5 6 9 . 1 . 3 . 2 8 4 . 8 7 4 1 9 6 3 5 3 4 5 2 . 6 1 7 9
P6	'5', '3', '4', '6', '7', '8', '9', '1', '2', '6', '7', '2', '1', '9', '5', '3', '4', '8', '1', '9', '8', '3', '4', '2', '5', '6', '7', '8', '5', '9', '7', '6', '1', '4', '2', '3', '4', '2', '6', '8', '5', '3', '7', '9', '1', '7', '1', '3', '9', '2', '4', '8', '5', '6', '9', '6', '1', '5', '3', '7', '2', '8', '4', '2', '8', '7', '4', '1', '9', '6', '3', '5', '3', '4', '5', '2', '8', '6', '1', '7', '.';	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9

Table 2: Casos de teste para os caminhos independentes do Sudoku.

5.1.2 Data Flow

O teste de Data flow é um processo de coleta de informações sobre como as variáveis fluem os dados no programa. Ele tenta obter informações específicas de cada ponto específico no processo. O teste de fluo de dados tem um grupo de estratégias de teste para examinar o fluxo de controlo de programas, a fim de explorar a sequência de variáveis de acordo com a sequência de eventos. Ele se concentra principalmente nos pontos em que os valores são atribuídos às váriaves e o ponto em que estes são usados.

Vantagem:
O teste de fluxo de dados é usado para encontrar os seguintes problemas:

- Encontrar uma variável usada, mas nunca definida.
- Encontrar uma variável definida, mas nunca usada.
- Encontrar uma variável definida várias vezes antes de ser usada.

Desvantagem:
O processo é demorado.

Existem vários tipos de teste de fluxo de dados, tais como All def, All use, All-Du-Paths, sendo que este último é o tipo de teste que iremo-nos focar mais, pois esta técnica apesar de ser mais complexa, é a melhor pois analisa todos os caminhos possíveis de uma definição de variáveis.

- **Função Dijkstra()**
    1. **Grafo de Data Flow**

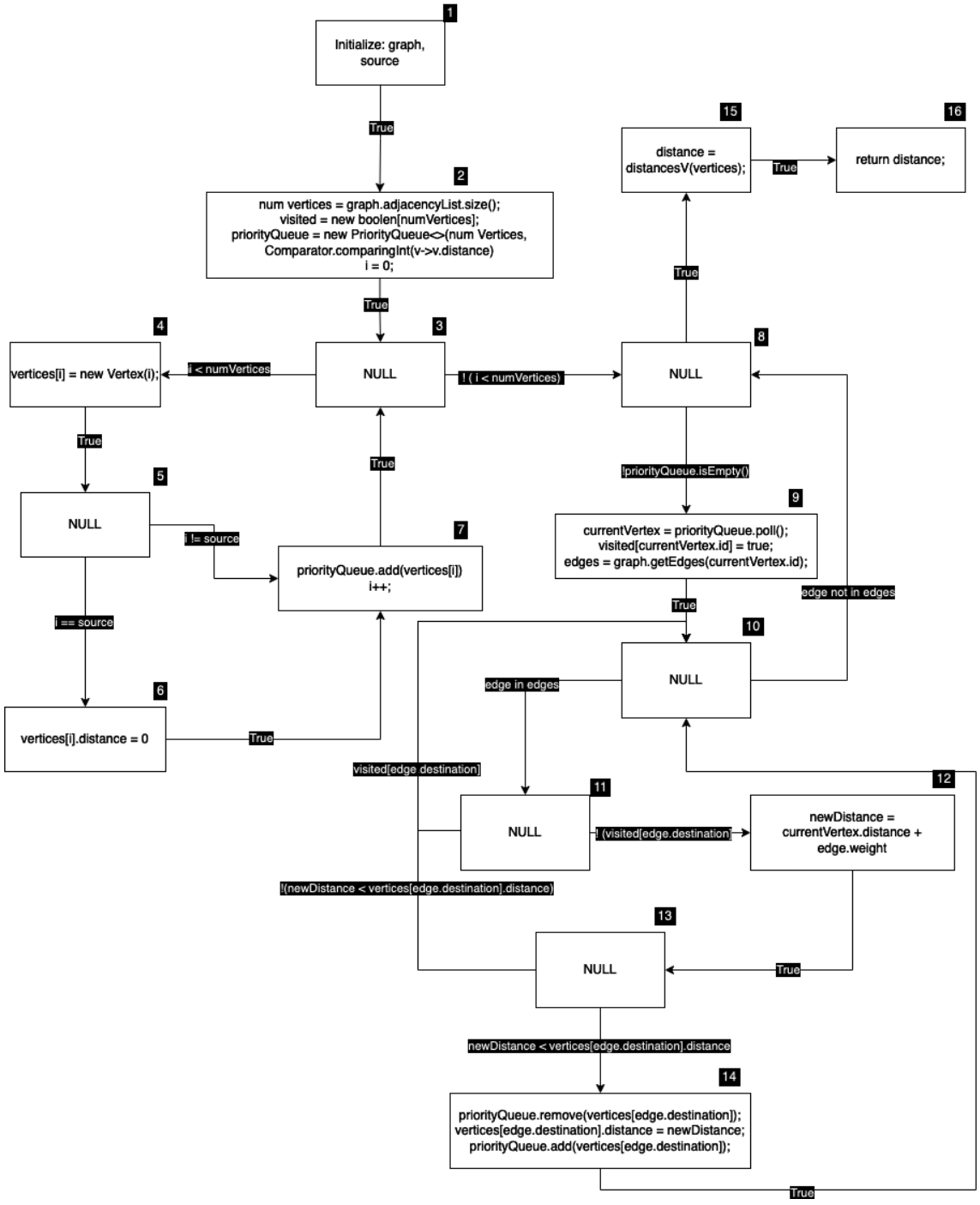

Figure 3: Data Flow - função Dijkstra.

2. **Caminhos ADUP**

Para cada variável, iremos identificar caminhos no gráfico de fluxo de dados que satisfaçam os critérios de seleção (todos os caminhos du para cada variável).
O nó **1** corresponde ao Start, e o nó **16** corresponde ao End. Nesta função, as variáveis são as seguintes:
graph, *visited*, *numVertices*, *priorityQueue*, *newDistance*, *distance*, *source*, *currentVertex*, *edges*, *vertices* e *i*.

- **graph - global**
  - ADUP1 - 1,2
  - ADUP2 - 1, 2, 3, 8, 9
  - ADUP3 - 1, 2, 3, 4, 5, 7, 3, 8, 9
  - ADUP4 - 1, 2, 3, 4, 5, 6, 7, 3, 8, 9
  - Caminho completo: 1, 2, 3, 4, 5, 6, 7, 3, 9, 10, 11, 12, 13, 14, 10, 15, 16
- **visited**
  - ADUP1 - 2, 3, 8, 9
  - ADUP2 - 2, 3, 4, 5, 7, 3, 8, 9
  - ADUP3 - 2, 3, 4, 5, 6, 7, 3, 8, 9
  - ADUP4 - 9, 10, 11, 10
  - ADUP5 - 9, 10, 11, 12
  - Caminho completo: 1, 2, 3, 4, 5, 6, 7, 3, 9, 10, 11, 12, 13, 14, 10, 15, 16
- **numVertices**
  - ADUP1 - 2, 3, 4
  - ADUP2 - 2, 3, 8
  - Caminho completo: 1, 2, 3, 4, 5, 7, 3, 8, 9, 10, 15, 16
- **priorityQueue**
  - ADUP1 - 2, 3, 4, 5, 7
  - ADUP2 - 2, 3, 4, 5, 6, 7
  - ADUP3 - 7, 3, 8, 15
  - ADUP4 - 7, 3, 8, 9
  - ADUP5 - 9, 10, 11, 12, 13, 14
  - Caminho completo: 1, 2, 3, 4, 5, 6, 7, 3, 9, 10, 11, 12, 13, 14, 10, 15, 16

  NOTA: priorityQueue.poll() é uma definição pois remove o primeiro elemento da fila.
- **newDistance**
  - ADUP1 - 12, 13, 14
  - ADUP2 - 12, 13, 10
  - Caminho completo: 1, 2, 3, 4, 5, 6, 7, 3, 9, 10, 11, 12, 13, 14, 10, 15, 16
- **distance**
  - ADUP1 - 15, 16
  - Caminho completo: 1, 2, 3, 4, 5, 6, 7, 3, 9, 10, 11, 12, 13, 14, 10, 15, 16

- **source - global**
  - ADUP1 - 1, 2, 3, 4, 5, 7
  - ADUP2 - 1, 2, 3, 4, 5, 6
  - Caminho completo: 1, 2, 3, 4, 5, 6, 7, 3, 9, 10, 11, 12, 13, 14, 10, 15, 16
- **currentVertex**
  - ADUP1 - 9, 10, 11, 12
  - Caminho completo: 1, 2, 3, 4, 5, 6, 7, 3, 9, 10, 11, 12, 13, 14, 10, 15, 16
- **edge**
  - ADUP1 - 9, 10, 8
  - ADUP2 - 9, 10, 11
  - Caminho completo: 1, 2, 3, 4, 5, 6, 7, 3, 9, 10, 11, 12, 13, 14, 10, 15, 16
- **edge**
  - ADUP1 - 9, 10, 11
  - ADUP2 - 9, 10, 8
  - ADUP3 - 9, 10, 11, 12
  - ADUP4 - 9, 10, 11, 10
  - ADUP5 - 9, 10, 11, 12, 13, 14
  - ADUP6 - 9, 10, 11, 12, 13, 10
  - Caminho completo: 1, 2, 3, 4, 5, 6, 7, 3, 9, 10, 11, 12, 13, 14, 10, 15, 16
- **vertices**
  - ADUP1 - 2, 3, 4
  - ADUP2 - 4, 5, 6
  - ADUP3 - 4, 5, 7
  - ADUP4 - 6, 7
  - ADUP5 - 4, 5, 7, 3, 8, 15
  - ADUP6 - 4, 5, 7, 3, 8, 9, 10, 11, 12, 13, 10
  - ADUP7 - 4, 5, 7, 3, 8, 9, 10, 11, 12, 13, 14
  - ADUP8 - 6, 7, 3, 8, 15
  - ADUP9 - 6, 7, 3, 8, 9, 10, 11, 12, 13, 10
  - ADUP10 - 6, 7, 3, 8, 9, 10, 11, 12, 13, 14
  - Caminho completo: 1, 2, 3, 4, 5, 6, 7, 3, 9, 10, 11, 12, 13, 14, 10, 15, 16
- **i**
  - ADUP1 - 2, 3
  - ADUP2 - 2, 3, 4
  - ADUP3 - 2, 3, 4, 5, 7
  - ADUP4 - 2, 3, 4, 5, 6, 7
  - ADUP5 - 7, 3
  - ADUP6 - 7, 3, 4, 5
  - ADUP7 - 7, 3, 4, 5, 6
  - Caminho completo: 1, 2, 3, 4, 5, 6, 7, 3, 9, 10, 11, 12, 13, 14, 10, 15, 16

Iremos apresentar de seguida os casos de teste para os caminhos completos acima descritos.

3. **Casos de teste - Data Flow**
   Para a tabela ficar mais visível iremos apresentar os dois caminhos que conseguem passar pelos casos de teste.

   - **PATH1** - 1, 2, 3, 4, 5, 6, 7, 3, 9, 10, 11, 12, 13, 14, 10, 15, 16
   - **PATH2** - 1, 2, 3, 4, 5, 7, 3, 8, 9, 10, 15, 16

   Os inputs são:

   - **INPUT1**:

     ```
     int numVertices = 5;
     Graph graph = new Graph(numVertices);
     graph.addEdge(0,2,1);
     graph.addEdge(0,1,10);
     graph.addEdge(2,4,5);
     graph.addEdge(2,3,10);
     graph.addEdge(3,4,3);
     int[][] distance = dijkstra(graph, 0);
     ```

   - **INPUT2**:

     ```
     int numVertices = 1;
     Graph graph = new Graph(numVertices);
     int[][] distance = dijkstra(graph, 1);
     ```

variável	**Caso de teste**	**Caminho**	**Graph**	**Source**
graph	CT01	PATH1	INPUT1	0
visited	CT02	PATH1	INPUT1	0
numVertices	CT03	PATH2	INPUT2	1
priorityQueue	CT04	PATH1	INPUT1	0
newDistance	CT05	PATH1	INPUT1	0
distance	CT06	PATH1	INPUT1	0
source	CT07	PATH1	INPUT1	0
currentVertex	CT08	PATH1	INPUT1	0
edges	CT09	PATH1	INPUT1	0
vertices	CT10	PATH1	INPUT1	0
i	CT11	PATH1	INPUT1	0

Table 3: Casos de teste para os caminhos ADUP do Dijkstra.

- **Função Solve()**

    1. **Grafo de Data Flow**

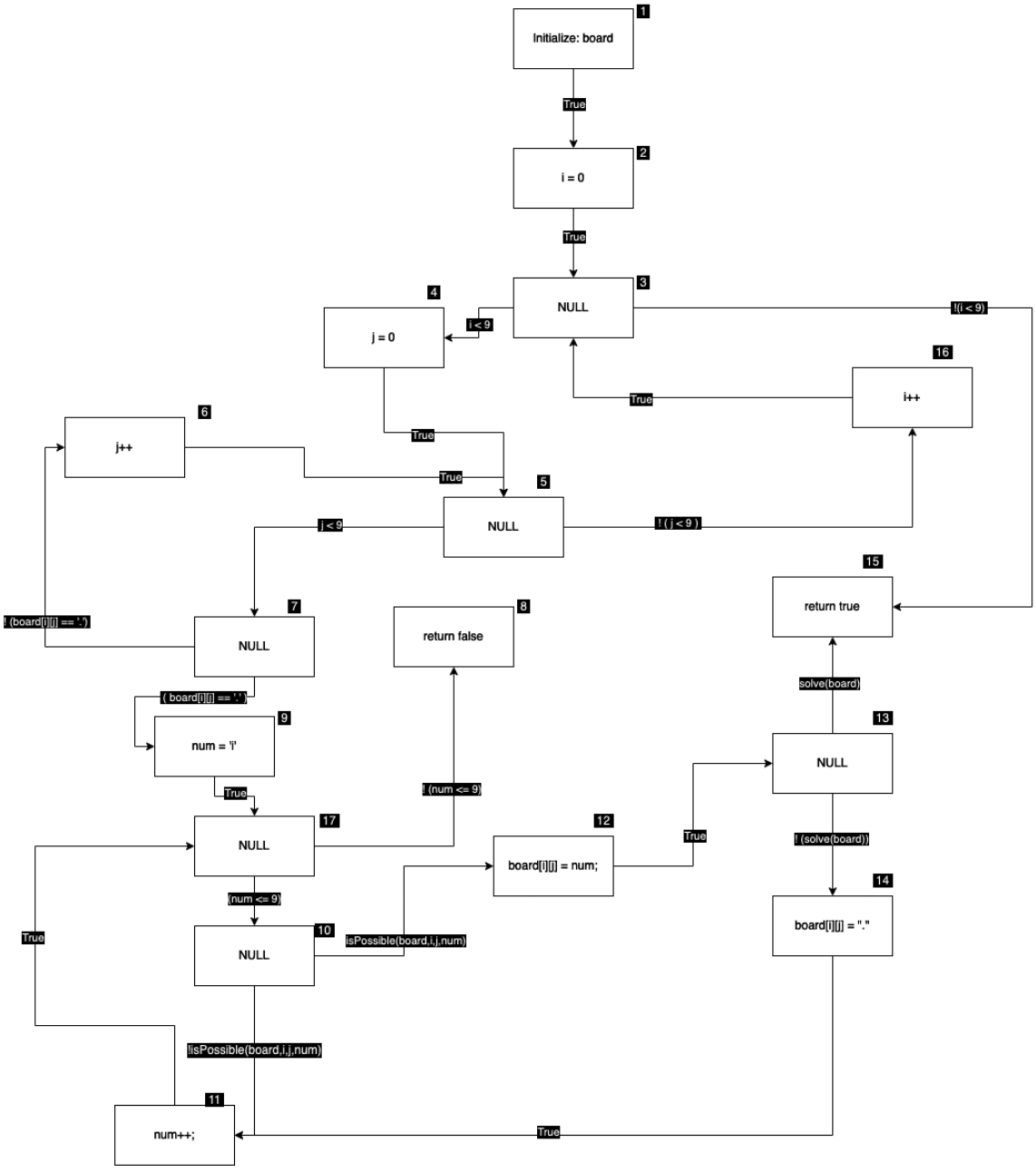

Figure 4: Data Flow - função Solve.

2. **Caminhos ADUP**

Para cada variável, iremos identificar caminhos no gráfico de fluxo de dados que satisfaçam os critérios de seleção (todos os caminhos du para cada variável). O nó **1** corresponde ao Start, e os nós **8**, **15** correspondem ao End. Nesta função, as variáveis são as seguinte: *board*, *i*, *j*, *num*.

- **board - global**
  - ADUP1 - 1, 2, 4, 3, 5, 7, 6
  - ADUP2 - 1, 2, 4, 3, 5, 7, 9
  - ADUP2 - 1, 2, 3, 5, 7, 9, 17, 10, 12
  - ADUP3 - 1, 2, 3, 5, 7, 9, 17, 10, 11
  - ADUP4 - 12, 13, 15
  - ADUP5 - 12, 13, 14, 11, 17, 10, 11
  - ADUP6 - 12, 13, 14, 11, 17, 10, 12
  - Caminho completo: 1, 2, 3, 4, 5, 7, 9, 17, 10, 11, 17, 10, 12, 13, 14, 11, 17, 8. e 1, 2, 3, 4, 5, 7, 6, 5, 16, 3, 4, 5, 7, 6, 5, 7, 9, 10, 11, 10, 12, 13, 15
    NOTA: São necessários 2 caminhos para completar todos os nós.

- **i**
  - ADUP1 - 2, 3,15
  - ADUP2 - 2, 3, 4
  - ADUP3 - 2, 3, 4, 5, 7, 6
  - ADUP4 - 2, 3, 4, 5, 7, 9
  - ADUP5 - 2, 3, 4, 5, 16
  - ADUP6 - 2, 3, 4, 5, 7, 9, 17, 8
  - ADUP7 - 2, 3, 4, 5, 7, 9, 17, 10, 11
  - ADUP8 - 2, 3, 4, 5, 7, 9, 17, 10, 12
  - ADUP9 - 2, 3, 4, 5, 7, 9, 17, 10, 13, 14
  - ADUP10 - 16, 3, 4
  - ADUP11 - 16, 3, 15
  - ADUP12 - 16, 3, 4, 5, 7, 6
  - ADUP13 - 16, 3, 4, 5, 7, 9
  - ADUP14 - 16, 3, 4, 5, 16
  - ADUP15 - 16, 3, 4, 5, 7, 9, 17, 8
  - ADUP16 - 16, 3, 4, 5, 7, 9, 17, 10, 11
  - ADUP17 - 16, 3, 4, 5, 7, 9, 17, 10, 12
  - Caminho completo: 1, 2, 3, 4, 5, 7, 9, 17, 10, 11, 17, 10, 12, 13, 14, 11, 17, 8. e 1, 2, 3, 4, 5, 7, 6, 5, 16, 3, 4, 5, 7, 6, 5, 7, 9, 10, 11, 10, 12, 13, 15
    NOTA: São necessários 2 caminhos para completar todos os nós.

- **j**

* ADUP1 - 4, 5, 16, 3,(4)
* ADUP2 - 4, 5, 7, 6
* ADUP3 - 4, 5, 7, 9, 17, 8
* ADUP4 - 4, 5, 7, 9, 17, 10, 11
* ADUP5 - 4, 5, 7, 9, 17, 10, 12
* ADUP6 - 6, 5, 16, 3, 4
* ADUP7 - 6, 5, 7, (6)
* ADUP9 - 6, 5, 7, 9, 17, 10, 12
* ADUP10 - 6, 5, 7, 9, 17, 10, 12, 13, 14
* ADUP11 - 6, 5, 7, 9, 17, 10, 11
* Caminho completo: 1, 2, 3, 4, 5, 7, 9, 17, 10, 11, 17, 10, 12, 13, 14, 11, 17, 8. e 1, 2, 3, 4, 5, 7, 6, 5, 16, 3, 4, 5, 7, 6, 5, 7, 9, 10, 11, 10, 12, 13, 15
  NOTA: São necessários 2 caminhos para completar todos os nós.

– **num**
* ADUP1 - 9, 17, 8
* ADUP2 - 9, 17, 10
* ADUP3 - 9, 17, 10, 11
* ADUP4 - 9, 17, 10, 12
* ADUP5 - 11, 17, 10
* ADUP6- 11, 17, 8
* ADUP7 - 11, 17, 10, 12
* ADUP8 - 11, 17, 10, (11)
* Caminho completo: 1, 2, 3, 4, 5, 7, 9, 17, 10, 11, 17, 10, 12, 13, 14, 11, 17, 8. e 1, 2, 3, 4, 5, 7, 6, 5, 16, 3, 4, 5, 7, 6, 5, 7, 9, 10, 11, 10, 12, 13, 15
  NOTA: São necessários 2 caminhos para completar todos os nós.

3. **Casos de teste - Data Flow**

Para a tabela ficar mais visível iremos apresentar os dois caminhos que conseguem passar pelos casos de teste.

– **PATH1** - 1, 2, 3, 4, 5, 7, 9, 17, 10, 11, 17, 10, 12, 13, 14, 11, 17, 8
– **PATH2** - 1, 2, 3, 4, 5, 7, 6, 5, 16, 3, 4, 5, 7, 6, 5, 7, 9, 10, 11, 10, 12, 13, 15

Os inputs são:

– **INPUT1**:

'.', '.', '4', '6', '7', '8', '9', '1', '2',
'.', '7', '2', '1', '9', '5', '3', '4', '8',
'1', '9', '8', '3', '4', '.', '5', '6', '7',
'8', '5', '.', '7', '6', '1', '4', '2', '3',

'2', '2', '6', '8', '5', '3', '.', '.', '1',
'7', '1', '3', '9', '2', '4', '8', '5', '6',
'9', '.', '1', '.', '3', '.', '2', '8', '4',
'.', '8', '7', '4', '1', '9', '6', '3', '5',
'3', '4', '5', '2', '.', '6', '1', '7', '9'

– **INPUT2**:

'5', '3', '4', '6', '7', '8', '9', '1', '2',
'6', '7', '2', '1', '9', '5', '3', '4', '8',
'1', '9', '8', '3', '4', '2', '5', '6', '7',
'8', '5', '9', '7', '6', '1', '4', '2', '3',
'4', '2', '6', '8', '5', '3', '7', '9', '1',
'7', '1', '3', '9', '2', '4', '8', '5', '6',
'9', '6', '1', '5', '3', '7', '2', '8', '4',
'2', '8', '7', '4', '1', '9', '6', '3', '5',
'3', '4', '5', '2', '8', '6', '1', '7', '.'

O caminho **PATH1** e **PATH2** correspondem respetivamente ao **INPUT1** e **INPUT2**.

Variável	Caso de teste	Caminho	Input
board	CT01	PATH1 e PATH2	INPUT1 e INPUT2
i	CT02	PATH1 e PATH2	INPUT1 e INPUT2
j	CT03	PATH1 e PATH2	INPUT1 e INPUT2
num	CT04	PATH1 e PATH2	INPUT1 e INPUT2

Table 4: Casos de teste para os caminhos ADUP do Sudoku.

6 Critérios de Pass/Fail

Testes unitários são uma prática essencial no desenvolvimento de software que tem como objetivo garantir a qualidade e a robustez do código. Estes consistem na verificação individual e isolada de unidades de código tais como funções, garantindo que cada unidade funcione conforme o esperado.

Portanto, para este tipo de teste, só é considerado bem sucedido se o output corresponder ao output esperado.

Se todos os testes forem sucedidos com sucesso, podemos dizer que o software funciona conforme planejado.

7 Entregáveis de testes

- problem.txt - Apresenta a funcionalidade do software escolhido para testar.
- Diagramas: as pastas DataFlow e ControlFlow contêm os seus diagramas correspondentes de cada técnica.

- Código:
  - **code.java** - contêm o código da resolução do Sudoku. Que é o código fonte do software que testamos.
  - **dijkstra.java** - contêm o código da resolução do Sudoku. Que é o código fonte do software que testamos.
- Outputs - Esta pasta contém todos os outputs esperados e os outputs finais.
- Relatório:
  - Plano de testes.
  - Casos de teste.
  - Ambiente necessário para o teste de white box.
  - Critérios de teste.
  - Conclusão de teste.

8 Necessidades do ambiente

É necessário usar um IDE, foi utilizado o VS code.

O uso da ferramenta diff é recomendado para que se possa comparar os outputs esprados com os finais.

Para testagem de inputs, é necessário ter o Java instalado.

Para executar é necessário, ter a terminologia " > ", por exemplo, na linha de comando:

/openjdk-21.0.2/Contents/Home/bin/java Code.Dijkstra > Outputs/ControlFlow/Output/Dijkstra/P5.txt

Isto significa, executar o software e o output ir para o file P5.txt. O input tem de ser colocado previamente.

9 Divisão de tarefas

Cada elemento do grupo ficou responsável por uma função, neste caso o Bruno ficou responsável pela função *Dijkstra*, e o Rui pela função *Solve*.

Sendo que ao longo do desenvolvimento, fomos nos ajudando um ao outro, realizando o debate dos planos de tstes, a implementação do caminhos, a realização do data flow.

10 Relatório de Conclusão dos Testes

Nesta secção apresentamos os resultados dos testes unitários de acordo com o critério de Pass/Fail definidos na secção 6.

10.1 Control Flow Testing

Teste	Output Esperado	Output	Comentário	Passou/Falhou
P3	0 - 2147483647	0 - 2147483647	Ok	
P4	0 - 0	0 - 0	Ok	
P5	0 - 0 1 - 2147483647 2 - 2147483647 3 - 2147483647 4 - 2147483647	0 - 0 1 - 2147483647 2 - 2147483647 3 - 2147483647 4 - 2147483647	Ok	
P6	0 - 0 1 - 2147483647	0 - 0 1 - 2147483647	Ok	
P7	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	Ok	

Table 5: Resultados para os casos de teste da função de Dijkstra.

De acordo com a seção 6, o software atende aos critérios. (100% sucesso).

Teste	Output Esperado	Output	Comentário	Passou/Falhou
P2	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	Ok	
P3	4 5 6 7 8 9 1 2 3 7 8 9 1 2 3 4 5 6 2 1 4 3 6 5 8 9 7 3 6 5 8 9 7 2 1 4 8 9 7 2 1 4 3 6 5 5 3 1 6 4 2 9 7 8 6 4 2 9 7 8 5 3 1 9 7 8 5 3 1 6 4 2	1 2 3 4 5 6 7 8 9 4 5 6 7 8 9 1 2 3 7 8 9 1 2 3 4 5 6 2 1 4 3 6 5 8 9 7 3 6 5 8 9 7 2 1 4 8 9 7 2 1 4 3 6 5 5 3 1 6 4 2 9 7 8 6 4 2 9 7 8 5 3 1 9 7 8 5 3 1 6 4 2	Ok	
P4	. 3 4 5 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	. 3 4 5 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	Ok	
P5	. . 4 6 7 8 9 1 2 . 7 2 1 9 5 3 4 8 1 9 8 3 4 . 5 6 7 8 5 . 7 6 1 4 2 3 2 2 6 8 5 3 . . 1 7 1 3 9 2 4 8 5 6 9 . 1 . 3 . 2 8 4 . 8 7 4 1 9 6 3 5 3 4 5 2 . 6 1 7 9	. . 4 6 7 8 9 1 2 . 7 2 1 9 5 3 4 8 1 9 8 3 4 . 5 6 7 8 5 . 7 6 1 4 2 3 2 2 6 8 5 3 . . 1 7 1 3 9 2 4 8 5 6 9 . 1 . 3 . 2 8 4 . 8 7 4 1 9 6 3 5 3 4 5 2 . 6 1 7 9	Ok	
P6	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	Ok	

Table 6: Resultados para os casos de teste da função de Solve.

10.2 Data Flow Testing

Teste	Output Esperado	Output	Comentário	Passou/Falhou
CT01	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	Ok	
CT02	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	Ok	
CT03	0 - 2147483647	0 - 2147483647	Ok	
CT04	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	Ok	
CT05	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	Ok	
CT06	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	Ok	
CT07	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	Ok	
CT08	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	Ok	
CT09	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	Ok	
CT10	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	Ok	
CT11	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	0 - 0 1 - 10 2 - 1 3 - 11 4 - 6	Ok	

Table 7: Resultados para os casos de teste da função de Dijkstra.

Nesta tabela, temos 2 inputs para cada teste, visto que para que o data flow fosse concluído com sucesso, era necessário passar por todos os caminhos, sendo que é não existe um caminho que passe por todos. Iremos seguir a ordem **INPUT1** e **INPUT2**.

Teste	**Output Esperado**	**Output**	**Comentário**	**Passou/Falhou**
CT01	. . 4 6 7 8 9 1 2 . 7 2 1 9 5 3 4 8 1 9 8 3 4 . 5 6 7 8 5 . 7 6 1 4 2 3 2 2 6 8 5 3 . . 1 7 1 3 9 2 4 8 5 6 9 . 1 . 3 . 2 8 4 . 8 7 4 1 9 6 3 5 3 4 5 2 . 6 1 7 9	. . 4 6 7 8 9 1 2 . 7 2 1 9 5 3 4 8 1 9 8 3 4 . 5 6 7 8 5 . 7 6 1 4 2 3 2 2 6 8 5 3 . . 1 7 1 3 9 2 4 8 5 6 9 . 1 . 3 . 2 8 4 . 8 7 4 1 9 6 3 5 3 4 5 2 . 6 1 7 9	Ok	
CT02	. . 4 6 7 8 9 1 2 . 7 2 1 9 5 3 4 8 1 9 8 3 4 . 5 6 7 8 5 . 7 6 1 4 2 3 2 2 6 8 5 3 . . 1 7 1 3 9 2 4 8 5 6 9 . 1 . 3 . 2 8 4 . 8 7 4 1 9 6 3 5 3 4 5 2 . 6 1 7 9	. . 4 6 7 8 9 1 2 . 7 2 1 9 5 3 4 8 1 9 8 3 4 . 5 6 7 8 5 . 7 6 1 4 2 3 2 2 6 8 5 3 . . 1 7 1 3 9 2 4 8 5 6 9 . 1 . 3 . 2 8 4 . 8 7 4 1 9 6 3 5 3 4 5 2 . 6 1 7 9	Ok	
CT03	. . 4 6 7 8 9 1 2 . 7 2 1 9 5 3 4 8 1 9 8 3 4 . 5 6 7 8 5 . 7 6 1 4 2 3 2 2 6 8 5 3 . . 1 7 1 3 9 2 4 8 5 6 9 . 1 . 3 . 2 8 4 . 8 7 4 1 9 6 3 5 3 4 5 2 . 6 1 7 9	. . 4 6 7 8 9 1 2 . 7 2 1 9 5 3 4 8 1 9 8 3 4 . 5 6 7 8 5 . 7 6 1 4 2 3 2 2 6 8 5 3 . . 1 7 1 3 9 2 4 8 5 6 9 . 1 . 3 . 2 8 4 . 8 7 4 1 9 6 3 5 3 4 5 2 . 6 1 7 9	Ok	
CT04	. . 4 6 7 8 9 1 2 . 7 2 1 9 5 3 4 8 1 9 8 3 4 . 5 6 7 8 5 . 7 6 1 4 2 3 2 2 6 8 5 3 . . 1 7 1 3 9 2 4 8 5 6 9 . 1 . 3 . 2 8 4 . 8 7 4 1 9 6 3 5 3 4 5 2 . 6 1 7 9	. . 4 6 7 8 9 1 2 . 7 2 1 9 5 3 4 8 1 9 8 3 4 . 5 6 7 8 5 . 7 6 1 4 2 3 2 2 6 8 5 3 . . 1 7 1 3 9 2 4 8 5 6 9 . 1 . 3 . 2 8 4 . 8 7 4 1 9 6 3 5 3 4 5 2 . 6 1 7 9	Ok	

Table 8: Resultados para o INPUT1 dataflow dos casos de teste da função de Solve.

Teste	Output Esperado	Output	Comentário	Passou/Falhou
CT01	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	Ok	
CT02	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	Ok	
CT03	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	Ok	
CT04	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	5 3 4 6 7 8 9 1 2 6 7 2 1 9 5 3 4 8 1 9 8 3 4 2 5 6 7 8 5 9 7 6 1 4 2 3 4 2 6 8 5 3 7 9 1 7 1 3 9 2 4 8 5 6 9 6 1 5 3 7 2 8 4 2 8 7 4 1 9 6 3 5 3 4 5 2 8 6 1 7 9	Ok	

Table 9: Resultados para o INPUT2 dataflow dos casos de teste da função de Solve.

10.3 Análise de resultados

Após visualização das tabelas podemos tirar boas conclusões, apartindo dos critérios apresentados na secção 6, ambos os softwares passam com 100% de sucesso.

Todos os outputs recebidos eram os outputs esperados.

Analisamos e verificamos se com todos os testes realmente passavam pelos nós, e tiramos novos

bons resultados. A cobertura do código é alta, tendo alguns testes com 100% de cobertura. Estes resultados estão na pasta *CodeCoverage*, em que apresetamos o code coverage para cada teste criado.

Os resultados obtidos nos testes unitários do DataFlow eram já esperados, visto que os testes são exatamente iguais aos do control flow.

11 Conclusão

Durante este projeto, exploramos as técnicas de controlo de fluxo e fluxo de dados nos nossos testes unitários, focando especificamente na função *Dijkstra* e *Solve*.

Ao aplicar o controlo de fluxo, pudemos examinar meticulosamente o comportamento do algoritmo em diferentes caminhos de execução, identificando áreas de código que exigiam atenção especial e garantindo uma cobertura abrangente dos casos de teste.

Da mesma forma, ao analisar o fluxo de dados, fomos capazes de rastrear como os dados eram manipulados e propagados ao longo do algoritmo.

Os resultados de nossa abordagem foram extremamente positivos.

Em suma, o controlo de fluxo e o fluxo de dados revelaram-se como ferramentas indispensáveis no nosso arsenal de testes unitários. Estamos confiantes de que continuaremos a aplicar e aprimorar essas técnicas nos nossos projetos futuros, garantindo assim a entrega de software de alta qualidade e confiabilidade.