Campus de Florianópolis

Relatório do Trabalho III de Paradigmas de Programação - INE5416

João Paulo A. Bonomo (21100133)
Gabriel Lima Jacinto (21202111)
Rodrigo Santos de Carvalho (21100139)

Florianópolis - SC

Junho, 2023

Relatório do Trabalho III de Paradigmas de Programação - INE5416

Professor Maicon Rafael Zatelli

Resumo

Este relatório é referente ao terceiro trabalho da disciplina de **Paradigmas de Programação - INE5416**, do trio João Paulo Bonomo, Gabriel Lima Jacinto e Rodrigo Santos de Carvalho. O trabalho em questão é um resolvedor do puzzle Vergleichssudoku, usando a linguagem de programação Prolog, que faz uso do paradigma lógico.

Sumário

1 Introdução ao Problema Proposto

1.1 Introdução

No arquivo de descrição do Trabalho III , o professor deixou a cargo dos grupos que escolhessem dentre três opções de puzzle:

1. Kojun;
2. Makaro;
3. **Vergleichssudoku**.

Nosso grupo optou pela terceira opção por uma questão de maior proximidade com o sudoku tradicional, e também pela maior simplicidade das regras (o que julgamos que simplificaria nossa implementação em Prolog, uma linguagem que ambos os 3 tínhamos pouco domínio).

1.2 Regras e Descrição do Jogo

Sucintamente, as regras do **Vergleichssudoku** são praticamente as mesmas do Sudoku tradicional, sendo acrescentada a presença de sinais de comparação ('>' e '<') entre as células do tabuleiro, o que torna o **Vergleichssudoku** mais fácil do que o próprio Sudoku, uma vez que diminui os números possíveis para cada uma das células no tabuleiro.

Um exemplo de tabuleiro 9x9 inicial de jogo pode ser visualizado na Figura 1, e a solução para esse exemplo pode ser vista na Figura 2.

Todos os tabuleiros que usamos para aprender a jogar, e também como "inputs"para testar nosso código foram retirados do site janko.at. Portanto, nosso trabalho consiste em resolver tabuleiros de tamanho 4x4, 6x6 e 9x9.

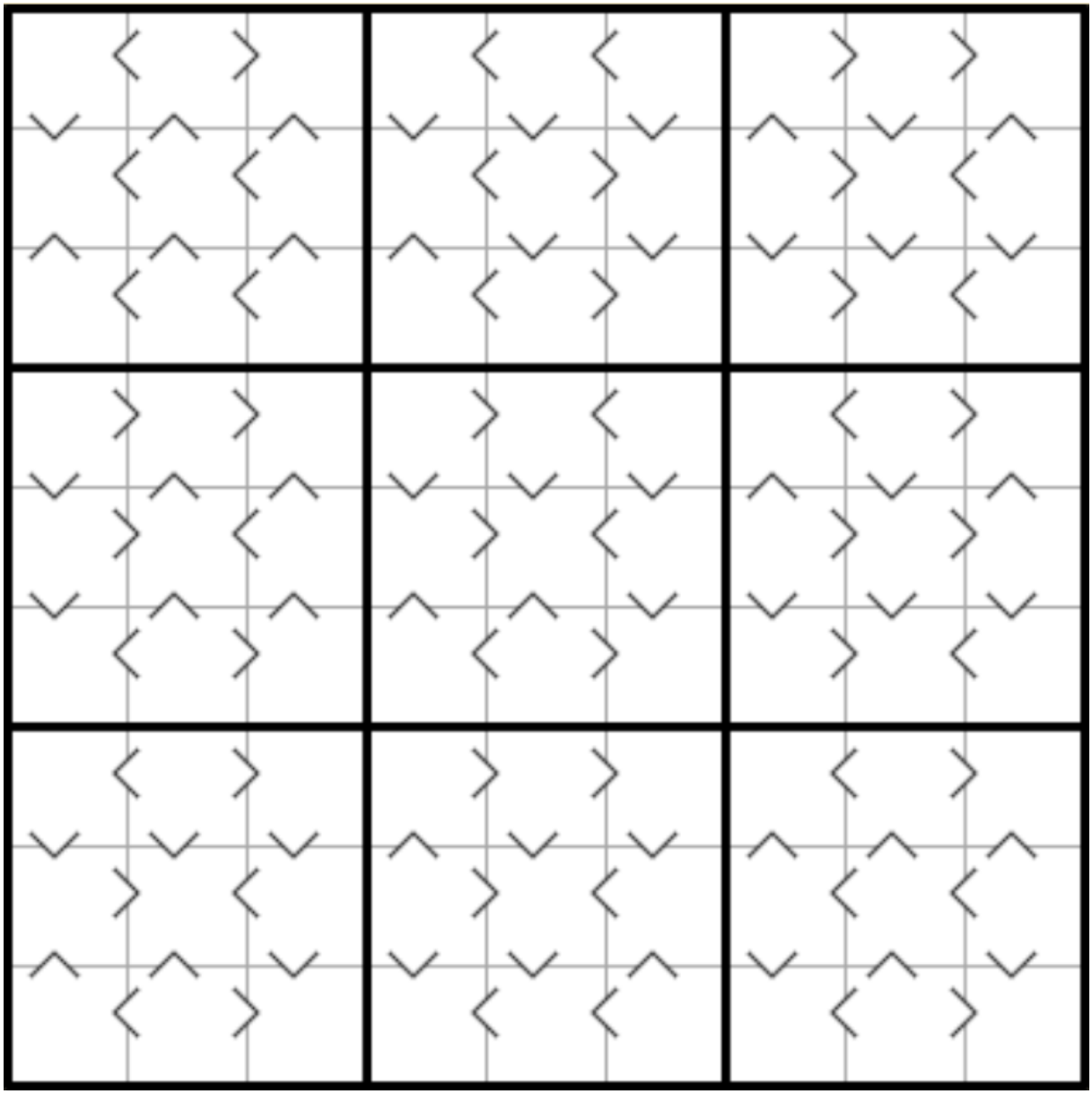

Figura 1 – Exemplo de tabuleiro 9x9 - retirado de janko.at/Raetsel/Sudoku/Vergleich/011.

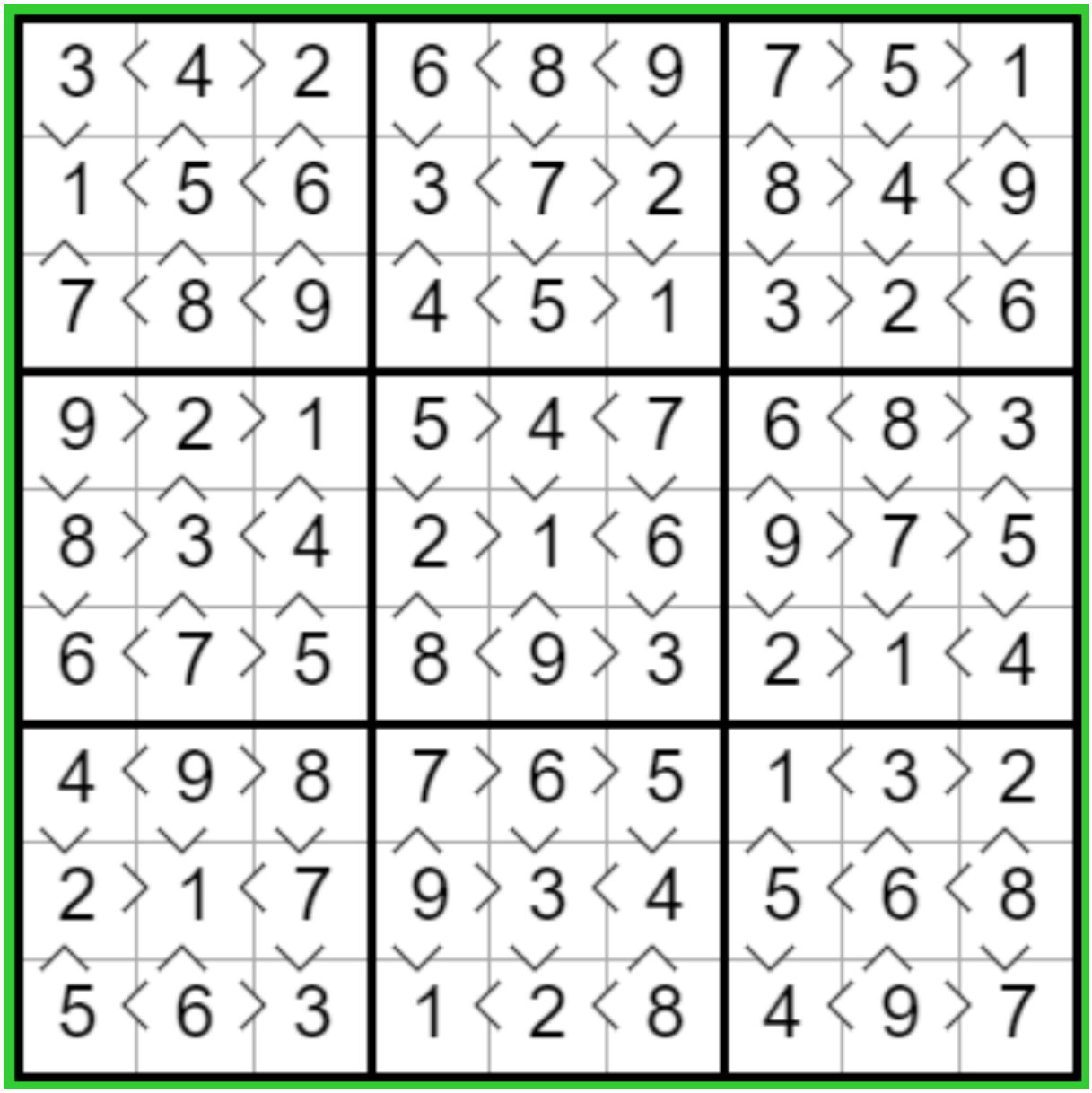

Figura 2 – Resolução da Figura 1 - retirado de janko.at/Raetsel/Sudoku/Vergleich/011.

2 Descrição Detalhada da Solução

2.1 Desenvolvimento da solução

Para o desenvolvimento de nossa solução, nos baseamos em soluções do *Sudoku* tradicional em ***Prolog***, bem como nossas soluções anteriores para o ***Vergleichssudoku***, que foram feitas em *Haskell* e *Elixir*.

2.2 Solução

Essa seção se destina a descrever nossa solução para o *puzzle* em tabuleiro ***9x9***. Como esse tabuleiro é o de maior tamanho e maior complexidade computacional, optamos por omitir no relatório o código que resolve os tabuleiros ***4x4*** e ***6x6***.

A Figura 3 contém as primeiras 37 linhas do arquivo solvedor. Nela, podemos observar os seguintes predicados:

- A primeira linha do código importa a biblioteca clpfd, que fornece restrições de programação em lógica com domínios finitos;
- O predicado *valid/1* verifica se um número está entre 1 e 9;
- Já o predicado *validate/1* verifica se uma lista não contém elementos repetidos;
- Ainda, há os predicados *bigger/2*, *smaller/2*, *biggerBigger/3*, *biggerSmaller/3*, *smallerBigger/3* e *smallerSmaller/3* estabelecem relações entre números válidos.

Já as Figuras 4, 5 e 6 ilustram o restante do código, em que:

- O predicado *blocks/2* divide uma lista em blocos de 3x3;
- O predicado *printRow/1* imprime uma linha do tabuleiro do ***Vergleichssudoku***;
- O predicado *solve/1* é o principal e realiza a resolução do Sudoku. Ele define a estrutura do tabuleiro com variáveis e usa restrições do *clpfd* para garantir que todas as regras do *puzzle* sejam atendidas. As restrições incluem, por exemplo, que cada linha, coluna e bloco contenha apenas números distintos;
- O predicado *flatten/2* é usado para transformar a matriz bidimensional do tabuleiro em uma lista unidimensional de variáveis;
- O predicado *transpose/2* é usado para transpor as linhas e colunas do tabuleiro;

- O predicado *label/1* atribui um valor a cada variável do tabuleiro;
- Por fim, o predicado *maplist/2* é usado para aplicar a impressão de cada linha do tabuleiro.

ninexnine.pl

code > ninexnine.pl

```
1  :- use_module(library(clpfd)).
2
3  % Command for swipl repl: solve(Sudoku), append(Sudoku, Vs), labeling([ff], Vs).
4
5  valid(A) :-
6    member(A, [1, 2, 3, 4, 5, 6, 7, 8, 9]).
7
8  validate([]).
9  validate([H|T]) :-
10   not(member(H, T)),
11   validate(T).
12
13 bigger(A, B) :-
14   valid(A),
15   valid(B),
16   A > B.
17
18 smaller(A, B) :-
19   valid(A),
20   valid(B),
21   A < B.
22
23 biggerBigger(A, B, C) :-
24   bigger(A, B),
25   bigger(B, C).
26
27 biggerSmaller(A, B, C) :-
28   bigger(A, B),
29   smaller(B, C).
30
31 smallerBigger(A, B, C) :-
32   smaller(A, B),
33   bigger(B, C).
34
35 smallerSmaller(A, B, C) :-
36   smaller(A, B),
37   smaller(B, C).
```

Figura 3 – Linhas de 1 a 37 do arquivo ninexnine.pl

2.3 Dificuldades no desenvolvimento da Solução

Como já tinhamos um certo domínio sobre o *puzzle* ***Vergleichssudoku***, nossas principais dificuldades foram principalmente com a linguagem ***Prolog*** em si.

Desenvolver um resolvedor de puzzle em ***Prolog*** foi um desafio estimulante para nós, mas também trouxe algumas dificuldades ao longo do processo. Durante o desenvolvimento de tal resolvedor, encontramos alguns obstáculos que exigiram tempo e esforço para superar.

Uma das principais dificuldades que enfrentamos foi a modelagem correta do problema em termos de fatos e regras lógicas. Especificamente, a representação adequada das restrições do *puzzle* em ***Prolog*** pode exigir um pensamento lógico abstrato e uma compreensão profunda das implicações das regras. A tradução das regras do *puzzle* para a lógica de programação do ***Prolog*** pode ser um processo complexo e requer um entendimento detalhado do domínio do problema.

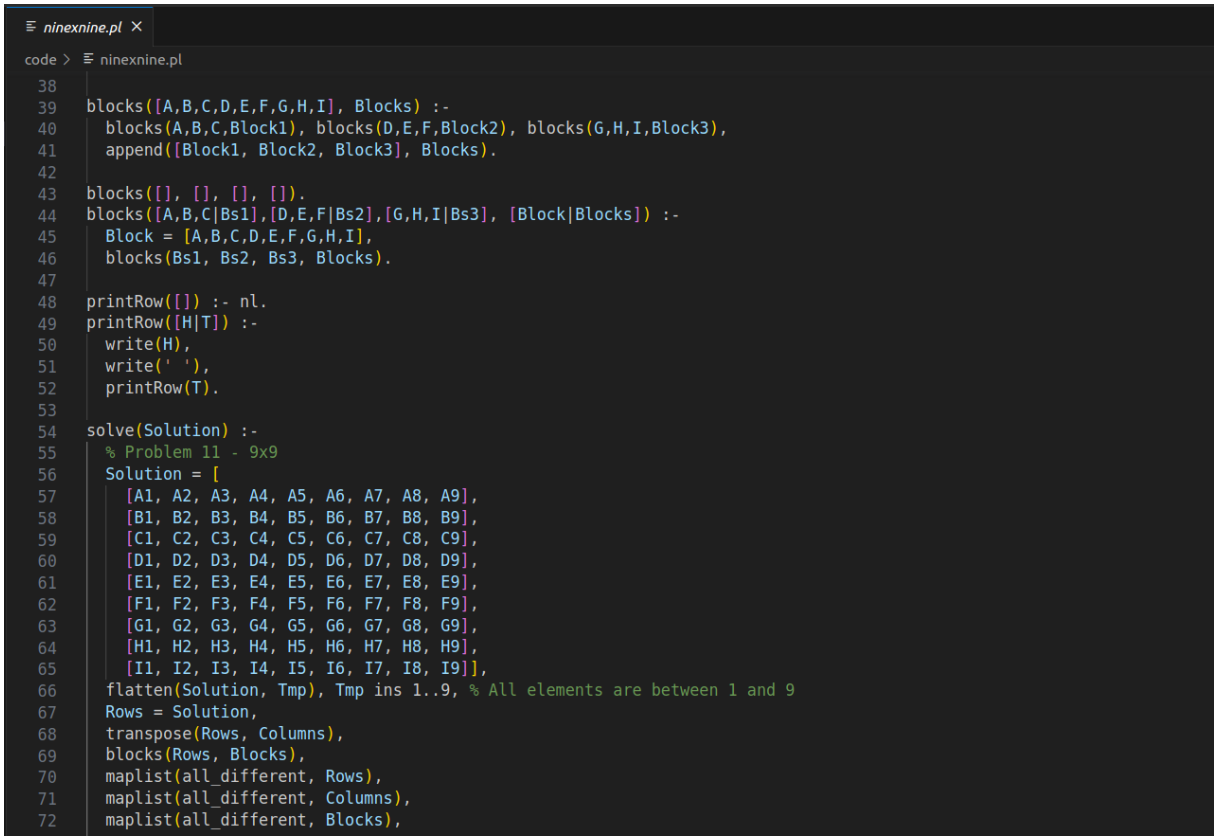

Figura 4 – Linhas de 38 a 72 do arquivo ninexnine.pl

Além disso, depurar o código em ***Prolog*** também foi desafiador. Ao contrário de outras linguagens de programação que usamos na disciplina, como *Haskell* e *Elixir*, onde a execução é passo a passo e seguimos um fluxo linear, em ***Prolog***, a execução ocorre por meio de inferências e unificações. Isso tornou a depuração mais abstrata e exigiu uma abordagem diferente para rastrear e identificar erros. Compreender o mecanismo de inferência lógica do ***Prolog*** e utilizar ferramentas de depuração apropriadas como algumas extensões do *VSCode* nos ajudou a superar essa dificuldade.

Por fim, a curva de aprendizado para dominar efetivamente a programação em ***Prolog*** também pode ser uma dificuldade a ser superada. A natureza declarativa e não procedural do ***Prolog*** exige uma mudança de mentalidade em relação à programação convencional. A compreensão dos conceitos fundamentais da programação lógica, como unificação, recursividade e resolução de metas, pode levar tempo e prática para se tornar familiar.

Além das dificuldades com a linguagem de programação utilizada, a administração do tempo para realizar o trabalho e dividí-lo entre os integrantes do grupo foi complexa, dada a tensão comum de final de semestre.

```
ninexnine.pl
code > ninexnine.pl
73
74  % Solution is presented in blocks (like the ones in sudoku), starting at the top left corner
75  % Block 1
76  smallerBigger(A1, A2, A3), % line 1
77  smallerSmaller(B1, B2, B3), % line 2
78  smallerSmaller(C1, C2, C3), % line 3
79  biggerSmaller(A1, B1, C1), % column 1
80  smallerSmaller(A2, B2, C2), % column 2
81  smallerSmaller(A3, B3, C3), % column 3
82
83  % Block 2
84  smallerSmaller(A4, A5, A6), % line 1
85  smallerBigger(B4, B5, B6), % line 2
86  smallerBigger(C4, C5, C6), % line 3
87  biggerSmaller(A4, B4, C4), % column 4
88  biggerBigger(A5, B5, C5), % column 5
89  biggerBigger(A6, B6, C6), % column 6
90
91
92  % Block 3
93  biggerBigger(A7, A8, A9), % line 1
94  biggerSmaller(B7, B8, B9), % line 2
95  biggerSmaller(C7, C8, C9), % line 3
96  smallerBigger(A7, B7, C7), % column 7
97  biggerBigger(A8, B8, C8), % column 8
98  smallerBigger(A9, B9, C9), % column 9
99
100 % Block 4
101 biggerBigger(D1, D2, D3), % line 4
102 biggerSmaller(E1, E2, E3), % line 5
103 smallerBigger(F1, F2, F3), % line 6
104 biggerBigger(D1, E1, F1), % column 1
105 smallerSmaller(D2, E2, F2), % column 2
106 smallerSmaller(D3, E3, F3), % column 3
107
```

Figura 5 – Linhas de 73 a 107 do arquivo ninexnine.pl

```
ninexnine.pl
code > ninexnine.pl
107
108 % Block 5
109 biggerSmaller(D4, D5, D6), % line 4
110 biggerSmaller(E4, E5, E6), % line 5
111 smallerBigger(F4, F5, F6), % line 6
112 biggerSmaller(D4, E4, F4), % column 4
113 biggerSmaller(D5, E5, F5), % column 5
114 biggerBigger(D6, E6, F6), % column 6
115
116 % Block 6
117 smallerBigger(D7, D8, D9), % line 4
118 biggerBigger(E7, E8, E9), % line 5
119 biggerSmaller(F7, F8, F9), % line 6
120 smallerBigger(D7, E7, F7), % column 7
121 biggerBigger(D8, E8, F8), % column 8
122 smallerBigger(D9, E9, F9), % column 9
123
124 % Block 7
125 smallerBigger(G1, G2, G3), % line 7
126 biggerSmaller(H1, H2, H3), % line 8
127 smallerBigger(I1, I2, I3), % line 9
128 biggerSmaller(G1, H1, I1), % column 1
129 biggerSmaller(G2, H2, I2), % column 2
130 biggerBigger(G3, H3, I3), % column 3
131
132 % Block 8
133 biggerBigger(G4, G5, G6), % line 7
134 biggerSmaller(H4, H5, H6), % line 8
135 smallerSmaller(I4, I5, I6), % line 9
136 smallerBigger(G4, H4, I4), % column 4
137 biggerBigger(G5, H5, I5), % column 5
138 biggerSmaller(G6, H6, I6), % column 6
139
140 % Block 9
141 smallerBigger(G7, G8, G9), % line 7
142 smallerSmaller(H7, H8, H9), % line 8
143 smallerBigger(I7, I8, I9), % line 9
144 smallerBigger(G7, H7, I7), % column 7
145 smallerSmaller(G8, H8, I8), % column 8
146 smallerBigger(G9, H9, I9), % column 9
147
148 maplist(label, Solution),
149 maplist(printRow, Solution), !.
```

Figura 6 – Linhas de 107 a 149 do arquivo ninexnine.pl

3 Paradigma Funcional vs. Paradigma Lógico

Ao longo desse semestre, vimos diferentes paradigmas de programação na disciplina, mas dentre eles, se destacaram 2: o paradigma funcional e o paradigma lógico. Dois trabalhos foram feitos usando o paradigma funcional para resolver um *puzzle* do tipo ***Vergleichssudoku*** (o primeiro em Haskell e o segundo em Elixir). Esse terceiro trabalho é referente ao paradigma lógico, usando a linguagem Prolog para resolver o mesmo *puzzle*. As seções abaixo são destinada a descrever algumas vantagens e desvantagens que encontramos nos dois paradigmas para a resolução do *puzzle*.

3.1 Paradigma Funcional

3.1.1 Vantagens

Imutabilidade: No paradigma funcional, os dados são imutáveis, o que significa que as funções não alteram o estado dos dados de entrada. Isso é benéfico para resolver puzzles, pois permite evitar efeitos colaterais e torna o código mais fácil de entender e raciocinar.

Funções puras: As funções no paradigma funcional são puras, ou seja, sempre retornam o mesmo resultado para a mesma entrada, sem efeitos colaterais. Isso facilita o teste e a depuração do código.

Composição de funções: No paradigma funcional, é comum compor várias funções pequenas para formar funções mais complexas. Isso permite uma abordagem modular e reutilizável para resolver puzzles, onde cada função pode lidar com uma parte específica do problema.

3.1.2 Desvantagens

Complexidade: O paradigma funcional pode ser mais complexo de entender e aplicar, especialmente para desenvolvedores acostumados com o paradigma imperativo. A lógica de programação funcional pode exigir um pensamento abstrato e uma mudança de mentalidade.

Performance: Em alguns casos, o paradigma funcional pode levar a uma sobrecarga de desempenho devido à imutabilidade e à criação de muitos objetos temporários. Isso pode ser um problema para implementações de resolvedores de puzzles que exigem um processamento intensivo.

Curva de aprendizado: Se a equipe de desenvolvimento não estiver familiarizada com o paradigma funcional, pode ser necessário investir tempo e esforço para aprender e adotar as melhores práticas. Isso pode ser uma desvantagem se o projeto tiver restrições de tempo ou recursos limitados.

3.2 Paradigma Lógico

3.2.1 Vantagens

Declaração de restrições: O paradigma lógico permite declarar restrições e relações entre as entidades envolvidas no puzzle. Isso simplifica a especificação do problema e permite que o resolvedor de puzzles encontre automaticamente a solução, explorando a lógica subjacente.

Inferência automática: O paradigma lógico possui mecanismos de inferência embutidos que podem ser usados para deduzir automaticamente informações adicionais com base nas restrições declaradas. Isso pode ser útil para resolver puzzles complexos, onde é difícil deduzir manualmente todas as implicações lógicas.

Expressividade: O paradigma lógico oferece uma maneira expressiva de especificar problemas e suas soluções, usando regras lógicas e fatos. Isso torna mais fácil traduzir as regras do puzzle para código e facilita a compreensão do raciocínio lógico por trás da solução.

3.2.2 Desvantagens

Eficiência: A execução de programas lógicos pode ser menos eficiente do que em outros paradigmas, devido à necessidade de explorar todas as possíveis soluções por meio de busca exaustiva. Isso pode ser um problema para puzzles com espaços de busca grandes, levando a um aumento no tempo de execução.

Complexidade da lógica: Às vezes, a lógica necessária para especificar um problema em um paradigma lógico pode ser complexa e exigir um entendimento profundo das regras e restrições envolvidas. Isso pode dificultar a implementação e manutenção do código.

Dificuldade na depuração: Quando um problema ocorre em um programa lógico, pode ser desafiador rastrear e depurar o código devido à natureza não procedural do paradigma lógico. Isso pode exigir habilidades adicionais de depuração e uma compreensão profunda dos mecanismos de inferência lógica.

4 Organização do Grupo

4.1 Comunicação do Grupo

Para organização do trabalho, o grupo utilizou de algumas ferramentas de comunicação como:

- canal de voz do **Discord**;
- grupo de **Whatsapp**;
- aba de mensagens do **Moodle**.

4.2 Gerenciamento do Código

Para o gerenciamento do código do trabalho, utilizamos apenas o **GitHub**. Mantendo um repositório privado com apenas os 3 membros como colaboradores.

4.3 Ambiente de Desenvolvimento

A IDE (Integrated Development Environment) escolhida para o desenvolvimento do trabalho foi o **VSCode**, com o auxílio de algumas extensões próprias para melhorar o ambiente para programação em ***Prolog***.