UNIVERSIDADE FEDERAL DE SANTA CATARINA

CENTRO TECNOLÓGICO (CTC)

DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA (INE)

INE5416 Paradigmas de Programação

Trabalho 1

Augusto de Hollanda Vieira Guerner	**22102192**
Eduardo Gwoszdz de Lazari	**22100612**
Micael Angelo Sabadin Presotto	**22104063**

Florianópolis

2023/2

1 Introdução

Este documento descreve o que foi feito no primeiro trabalho da matéria de Paradigmas de Programação. Mas antes de falar mais sobre o que foi feito, é interessante destacar as motivações do trabalho.

O objetivo principal dele, como se verá ao longo do desenvolvimento, é exercitar o conhecimento do paradigma funcional (um dos mais importantes), utilizando a linguagem Haskell, bem como desenvolver a capacidade crítica dos alunos em resolver problemas. Para isso, foi proposto três diferentes puzzles para se fazer um solucionador e a equipe teria que escolher um.

Dado as possibilidades de escolha dos jogos, a que mais nos interessou foi o Vergleich Sudoku, onde se assemelha com o Sudoku convencional, porém implementa comparações (maior ou menor) entre células de uma mesma região. Ademais, ao discutirmos maneiras de resolver os 3 jogos, percebemos a possibilidade de aplicar conceitos aprendidos na matéria de Grafos para resolver a opção escolhida. Dessa maneira, além de exercitarmos nossa lógica, iríamos desenvolver um projeto interdisciplinar, juntando os conhecimentos obtidos durante o semestre.

Antes de prosseguir para a descrição do que foi feito, vale destacar o que será falado e em qual ordem será abordado. Primeiro, tem-se o desenvolvimento com 4 tópicos principais: sobre o código python que originou o código haskell; sobre o código python que gera a matriz de comparações; sobre o código haskell propriamente; e sobre o desempenho de cada implementação (python, haskell e diferentes sudokus).

2 Desenvolvimento

2.1 Python

Como recomendado pelo próprio professor, o grupo decidiu fazer um código em python antes de fazer o em haskell. De fato, com o código python em mãos, a implementação em haskell se tornou mais fácil, ainda mais tendo em vista que ele foi elaborado pensando em futuramente ser traduzido para haskell, ou seja, usando recursão.

No geral, o código é bem simples. Existe um ponto de entrada, neste caso a função main, que faz o setup para o algoritmo, ou seja, cria a matriz solução com zeros, pois não há nenhuma casa definida inicialmente, a matriz comparação com base num tabuleiro real disponibilizado pelo site e a matriz de possibilidades.

A matriz comparação é bem simples, cada elemento dela é uma lista com 4 elementos: o primeiro, segundo, terceiro e quarto elemento da lista são respectivamente o sinal do elemento atual para o elemento de cima, da direita, de baixo e da esquerda. O valor 1 indica maior, 0 menor e -1 nada, ou seja, não há elemento para ser comparado, podendo ser a borda de uma região ou do sudoku.

Já a matriz de possibilidades é tão simples quanto. Cada elemento dela é uma lista com todas as possíveis escolhas a serem feitas de números naquele elemento, dado o esquema de sinais do bloco. Essa lista de possibilidades é calculada usando a busca por profundidade. Dado um elemento, navega-se pelos sinais iguais até ter passado por todos os elementos possíveis. Basicamente o número de elementos que a busca passou define um limite de maior ou menor dependendo do sinal que se percorreu.

Depois do setup na função main, há o algoritmo propriamente dito que resolve o sudoku Vergleich com a chamada do resolve() dentro do for o qual percorre todas possibilidades do primeiro elemento da matriz solução. Após resolver, o mostrado na tela o sudoku resolvido com o tempo de execução logo em baixo.

A função resolve é o cerne do programa, apesar de ser bem simples sua lógica.

2.2 Gerador de matriz de comparações

Ao decorrer do trabalho, para facilitar os testes tornou-se necessário a criação de um código que automatiza a criação da matriz de comparação, como já foi comentado e explicado na seção 2.1 de como ela funciona e por que decidimos criar essa matriz, então vamos direto ao ponto de como funciona o código que a gera.

Inicialmente, colocamos na lista o seu primeiro valor, que indica a relação com a célula acima. Caso esse elemento pertencer à primeira linha do sudoku, ou for elemento da borda de cima de uma região, é inserido -1, indicando nada. Se o mesmo não satisfazer essas condições, ele está apto a fazer

comparações com a célula acima, inserindo 1 caso for maior, e 0 quando menor.

Da mesma forma, inserimos no segundo, terceiro e quarto índice, respectivamente, as comparações com as células à direita, abaixo e à esquerda, formando assim, as possíveis comparações de um elemento com as suas células adjacentes.

2.3 Haskell

Para o código Haskell foi seguida uma estrutura parecida com a feita em Python, claro que o grupo precisou realizar vários ajustes por conta dos paradigmas das linguagens serem diferentes. Inicialmente, a declaração da matriz de comparação é igual a em Python. Já a matriz de possibilidades segue a mesma ideia, são feitas duas listas de compreensão (comportamento similar ao for do Python) para o preenchimento das possibilidades de cada elemento na matriz. Além disso, vale ressaltar que o range do Haskell possui final inclusivo (diferente do Python que é exclusivo), portanto, devemos calcular 10 - o limite pelo símbolo de menor.

Além disso, por Haskell ter uma tipagem imutável, ou seja, após a declaração de uma lista por exemplo, não é possível alterá-la, por isso foi criada a função trocaElemento(). Nela, realiza-se a substituição de um elemento presente em uma posição da lista por outro, retornando uma nova lista com os valores alterados. Ademais, o grupo fez uma função for(), que simula um for de uma linguagem de paradigma imperativa ou procedural, para que os fors feitos na linguagem python sejam implementados de maneira mais intuitiva em Haskell, já que a linguagem utiliza apenas da recursão.

Por fim, para a função resolve(), a ideia implementada foi a mesma, uma vez que o grupo tentou realizar as lógicas em Python sempre de uma forma recursiva para que a tradução ficasse mais simples. Nesse caso, resolve() verifica primeiramente se o elemento já existe em uma linha, coluna ou região, através da comparação com o retorno das funções obtemLinha(), obtemColuna() e obtemRegiao(). Após isso, aplica a função compara(), que verifica se o elemento pode ser inserido em uma posição, dada as suas comparações de maior e menor com os valores adjacentes.

Assim, o elemento (que pertence às possibilidades) é inserido na matriz e o algoritmo continua inserindo as possibilidades a cada célula. Se em dado momento não houver como inserir um número na célula atual, é aplicada a lógica de backtracking, ou seja, a função retorna para a célula anterior e troca o valor que foi previamente escolhido. Dessa maneira, percebe-se que o backtracking precisa repetidamente retroceder, para poder avançar a solução. Logo, percebe-se a importância do DFS no código.

2.4 Desempenho

Inicialmente, sem usar a ideia do DFS para aumentar o desempenho da solução, o código estava levando em média 8 a 10 minutos para terminar de ser executado em nossas máquinas. Contudo, concluímos que seria interessante aplicar algum algoritmo para reduzir as falhas do backtracking, surgindo então, a busca em profundidade.

Após implementar o DFS da forma que já foi explicado seu funcionamento na seção 2.1, conseguimos obter resultados na faixa de 8 a 10 segundos, ou seja, o código se tornou aproximadamente 60x mais rápido.

É importante ressaltar também que o algoritmo de busca em largura obteria o mesmo resultado, porém a implementação recursiva da busca em profundidade se tornou mais adequada para a linguagem Haskell.

3 Conclusão

Em conclusão, este documento descreveu o desenvolvimento do primeiro trabalho da disciplina de Paradigmas de Programação, que teve como principal objetivo exercitar o conhecimento do paradigma funcional usando a linguagem Haskell e desenvolver a capacidade crítica dos alunos na resolução de problemas. O projeto escolhido foi o Vergleich Sudoku, uma variação do Sudoku convencional que envolve comparações entre células de uma mesma região.

O desenvolvimento do projeto foi dividido em várias etapas, começando com a implementação em Python para facilitar a transição para Haskell. O código em Python tratava da criação de matrizes de solução, comparação e possibilidades, além de conter a lógica para resolver o Sudoku Vergleich. Em Haskell, a estrutura geral foi mantida, mas houve ajustes para atender aos paradigmas da linguagem, como a tipagem imutável e a substituição de loops por recursão.

Uma parte importante do projeto envolveu a criação de uma matriz de comparação automática, que simplificou os testes. Essa matriz foi gerada de acordo com as regras do jogo, considerando os sinais entre células adjacentes.

O desempenho do código teve um impacto significativo na sua evolução. Inicialmente, o programa em Python demorava um tempo consideravelmente grande para ser executado. No entanto, com a implementação do algoritmo de busca em profundidade (DFS), o tempo de execução foi reduzido em até 60x.

Em resumo, este trabalho representou um desafio interessante para os alunos, envolvendo a aplicação de conceitos de programação funcional, grafos e busca em profundidade na resolução de um quebra-cabeça único. Além disso, destacou a importância de otimizar algoritmos para melhorar o desempenho em casos complexos, demonstrando a utilidade prática dos conceitos abordados na disciplina de Paradigmas de Programação.

Referências

- https://www.janko.at/Raetsel/Sudoku/Vergleich/index.htm
- https://www.janko.at/Raetsel/Makaro/index.ht
- https://www.janko.at/Raetsel/Kojun/index.htm