Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - USP

GUIA DE CÓDIGO

Futoshiki 不等式

SCC0218 - Algoritmos Avançados e Aplicações

Prof.: Gustavo Enrique de Almeida Prado Alves Batista

Felipe Scrochio Custódio, 9442688
Gabriel Henrique Campos Scalici, 9292970

São Carlos
Setembro de 2016

Futoshiki 不等式

Implementação em C
- Makefile
- Estrutura de dados
- Funções auxiliares:
- Leitura dos dados:
- Saída de dados:
- Definição de tipos e otimização de acesso:
- Backtracking Simples:
- Forward checking :
- Forward checking + MVR:

Análise de Desempenho

Implementação em C

- *Makefile*
- *Estrutura de dados*
- *Funções auxiliares*
- *Leitura dos dados*
- *Saída de dados*
- *Definição de tipos e otimização de acessos*
- *Backtracking simples*
- *Forward checking*
- *Forward checking + MVR*

- Makefile

Foi criado um Makefile para simplificar os comandos tanto na hora de compilar quando na hora de realizar testes com demais tabuleiros ou até mesmo executar o programa.

No Makefile também há a opção de escolher o arquivo de teste para rodar no programa.

- Estrutura de dados

```
typedef struct board {

    // board size
    int n;
    CELL** matrix;

} BOARD;
```

Nossa lógica utiliza "*structs*" para armazenar as informações principais para o funcionamento do jogo futoshiki, isso é, nosso tabuleiro está armazenado em uma struct chamada "*BOARD*" que armazena o tamanho do tabuleiro e uma matriz de células, que por sua vez é outra struct chamada "*CELL*", que representa todas as posições do tabuleiro onde irão se colocar os números.

Como o tabuleiro do jogo possui restrições (Regras do próprio jogo futoshiki), em "*CELL*" também são armazenadas as restrições para essa célula, isso é, se ela deve ser menor que outra célula (Tais restrições são passadas na leitura do tabuleiro), que serão futuramente analisadas na hora de colocar os números permitidos no tabuleiro.

```
typedef struct cell {

    // number of restrictions
    int r;
    // restriction coordinates
    int** restrictions;
    int value;
    bool filled; // determines if cell is pre-filled

    // possible values for use with heuristics
    LIST* fw;

} CELL;
```

Outra informação importante sobre nossa estrutura de dados, é que mesmo não utilizando no backtracking simples (Sem

heurísticas), há uma struct chamada "*LIST*" que corresponde às possibilidades de valores para uma determinada célula, sendo assim, há uma variável do tipo "*LIST*" dentro das células. Tal estrutura é importante para o uso das heurísticas, como o "forward checking" que remove das listas de valores possíveis para células de mesma linha e coluna, verificando se tal número foi ou não uma boa escolha de maneira mais eficiente.

```
typedef struct list {

    // possible values
    int* vector;
    // number of possible values
    int size;
    // vector size is always board size

} LIST;
```

- Funções auxiliares:

Na implementação do projeto foram utilizadas funções específicas para cada struct, deixando seu uso ao longo do programa mais simples e legível.

Dentre as funções, temos de iniciar determinada struct, como "*initBoard*", "*initList*", para iniciar um tabuleiro e uma lista para cada célula, respectivamente.

```
list = (LIST*)malloc(sizeof(LIST));
// list has n members
list->vector = (int*)malloc(sizeof(int) * n);
// add possible values to list
for (i = 0; i < n; i++) {
    list->vector[i] = i + 1;
}
list->size = n;
return list;
```

O "*initList*" aloca dinamicamente um vetor com "*n*" posições, pois há "*n*" valores possíveis que podem ser adicionados em determinada célula e armazena os valores respectivos a cada espaço livre do vetor, ou seja, na posição "0" do vetor terá o número "1" pois tal célula pode assumir esse valor.

Há funções de limpar determinados dados da memória heap como "*destroyBoard*" para desalocar todo o espaço utilizado para o tabuleiro e tudo que o envolve, como as células e listas. Seguindo a estratégia usada na alocação, por meio de um laço "*for*" é dado "*free()*"em todas as colunas alocadas da matriz, depois é desalocado o vetor de linhas que foi utilizado.

Para "*LIST*" temos funções específicas como "*listAdd*" e "*listRemove*", que respectivamente adicionam e removem um valor possível que está anotado na lista de determinada célula fazendo um laço "*for*" que percorre a lista de valores possíveis até encontrar o correto para exclusão.

Tornando mais legível o processo de gerenciar as listas no momento da implementação.

Uma função importante na nossa forma de implementação, é a função "*isValid*", que após adicionar um número em uma célula, analisa, no backtracking simples, se esse valor é válido, testando por meio de laços "*for*" se na linha ou coluna daquele número exista um que entre em conflito, e ainda caso passe por essa verificação, ainda

```
// check if value already exists
for (i = 0; i < board->n; i++) {
}

// check if position has constraint
if (board->matrix[x][y].restrictions != NULL) {
}

// all conditions were valid
return TRUE;
```

checa se as restrições iniciais do tabuleiro foram atendidas, por meio de outro laço "*for*" dessa vez com base nas restrições de cada célula. Retornando "*FALSE*" caso durante algum passo seja encontrado incoerências e "*TRUE*" caso chegue no final da função sem ter encontrado nenhum conflito.

- Leitura dos dados:

Os dados são lidos dos casos de teste passados via stdin e são identificados pelo programa aloca os tabuleiro (criando uma lista de tabuleiros) com seu respectivo tamanho, aloca a quantidade correta de células dentro dos tabuleiros e também as listas de restrições pela função "*readBoard*" que pega os dados e vai percorrendo o tabuleiro, anotando os números iniciais na células correspondentes, anotando as restrições de "menor que" outra célula, criando a lista de restrições e já removendo alguns valores possíveis de tal lista com base nos valores inciais do tabuleiro.

```
b = (BOARD**)malloc(sizeof(BOARD*) * n);
for (i = 0; i < n; i++) {
    // get size of board
    scanf("%d", &d);
    // allocate memory for new board
    b[i] = initBoard(d);
    // get number of restrictions
    scanf("%d", &r);

    // read board
    for (j = 0; j < d; j++) {...
    }

    // read restrictions
    for (j = 0; j < r; j++) {...
    }
}
// all boards read successfully
return b;
```

Em tal função foram usados laços "*for*" para percorrer o tabuleiro, anotando os valores iniciais nas posições que foram lidas, ainda há outro laço "*for*" para anotar quantas restrições tem a célula em questão, já anotando quais são elas. Retornando para a main o ponteiro que indica onde foi alocado, para que possa ser usado.
Então ao final da leitura, temos alocado todos os tabuleiros, células e listas de restrições de cada tabuleiro no vetor (lista de restrições) de tabuleiros. Vale lembrar que tal lista não é usada na implementação sem heurística do backtracking, mesmo assim ela é criada em todos os casos, e usada apenas nos que necessitam de tal auxílio. Tornando possível utilizar os métodos de força bruta para a resolução dos mesmos.

- Saída de dados:

- Definição de tipos e otimização de acesso:

Em um arquivo header separado, implementamos duas estruturas de dados, uma para o tabuleiro e outra para a célula (posição).
Em um tabuleiro, armazenamos uma matriz de células e suas dimensões.
Em uma célula, armazenamos as coordenadas de restrição e seu valor. Como podemos ter mais de uma restrição por célula, temos um vetor de coordenadas X,Y. Isso deixa o acesso para validar uma restrição muito mais rápido do que se percorrêssemos, por exemplo, uma matriz de restrições.

- Backtracking Simples:

Nosso algoritmo de backtracking simples sem heurísticas é basicamente um algoritmo recursivo que começa verificando se o número máximo de atribuições foram atingidas (Exigido na descrição do projeto como no máximo 10^6 atribuições) retornando "*FALSE*", caso ainda não tenha sido atingido o número máximo, é verificado se a análise não chegou no fim do tabuleiro, o que indica que todas as atribuições foram feitas sem conflitos, retornando "*TRUE*" caso tenha percorrido todo o tabuleiro, sendo essa a nossa análise para verificar se o futoshiki foi resolvido corretamente.

```
// check if recursive calls reached overflow
if (*calls >= OVERFLOW) {
    return FALSE;
}
// check if has reached end of board
if (x >= (*b)->n || y >= (*b)->n) {
    return TRUE;
}
```

Após as duas verificações de parada, o algoritmo analisa todas as células do tabuleiro colocando os valores, caso não tenha um valor inicial (Valor já recebido na leitura do tabuleiro), e verificando se são válidos com a chamada da nossa função "*isValid*", já explicado anteriormente que verifica se não há o número na mesma linha, na mesma coluna e se as restrições do tabuleiro foram respeitadas, caso seja válido, o valor em questão é adicionado, e de forma recursiva, é chamado novamente o algoritmo, porém agora para a célula da próxima coluna do tabuleiro, e caso seja a última célula de coluna, é chamado para a primeira célula da próxima linha.

```
    if (futoshiki_simple(b, x+1, 0, calls)) {
        // placement was successful
        return TRUE;
    }
} else {
    // goes to next column
    if (futoshiki_simple(b, x, y+1, calls)) {
        // placement was sucessful
        return TRUE;
    }
```

- Forward checking :

Como já citado anteriormente, na implementação dessa heurística, iremos utilizar as listas para fazer a verificação adiante (Característica desse método de poda no backtracking).

A implementação, é recursiva, de modo que é necessário algum ponto de parada, que assim como no simples deve ser se exceder o número máximo de "*calls*" que são as chamadas.

A lógica funciona basicamente como a implementação sem heurística, a diferença é que agora são usados os valores possíveis armazenados nas listas antes de adicionar um valor, ou testá-lo naquela posição.

Além dessa função, de cortar os valores que estão inválidos, as listas servem para fazer a verificação adiante no código, antes de chegar até a célula que está com

```
if ((*b)->matrix[x][y].fw->vector[i] != 0) {
    // don't overwrite prefilled values
    if (!((*b)->matrix[x][y].filled)) {
        (*b)->matrix[x][y].value = (*b)->matrix[x][y].fw->vector[i];
        (*calls)++;
        // update forward checking lists (remove current value) of line and column
        updateLists(b, x, y, 0, (*b)->matrix[x][y].value);
    }
```

conflito, isso ocorre, pois quando é adicionado um valor, é percorrido todas as listas da mesma coluna e linha, removendo os valor que foi adicionado (Atualizando as listas de todas as células afetadas com "*updateList*"), dessa forma, ao remover quando fazemos uma escolha, é possível analisar antes de chegar em determinada célula, que ela não possui mais valores válidos. Caso após a atribuição de algum número, alguma célula fique sem valores possíveis, retorna "*FALSE*" e poda as possibilidades de backtracking.

Caso o valor colocado naquela célula não seja válido pois excluiu todos os valores da lista de outras células, esse valor é ressetado.

```
// reset value
if (!((*b)->matrix[x][y].filled)) {
    // reset lists
    updateLists(b, x, y, 1, (*b)->matrix[x][y].value);
    // reset value
    (*b)->matrix[x][y].value = 0;
}
```

Isso ajuda muito na resolução pois, não precisa percorrer várias células antes de chegar em uma com conflitos, reduzindo assim, o número de execuções feitas pela função, tornando mais eficiente o backtracking.

A forma de percorrer o tabuleiro (linhas e colunas) é similar ao sem heurística, já explicado anteriormente.

- Forward checking + MVR:

O algoritmo implementado com a junção das duas heurísticas, usou todo o código do algoritmo que tem somente Forward checking, com algumas pequenas modificações para que seja possível a implementação do MVR.

Tais modificações são basicamente, ao remover um dos valores possíveis da coluna e linha ao adicionar um valor, analisa qual das células possui a menor quantidade de valores possíveis (armazenados em "*size*"), escolhendo-a para análise e colocar um valor (heurística MVR).