INSTITUTO SUPERIOR TÉCNICO

Departamento de Engenharia Electrotécnica e de Computadores

Projeto de AED - ZERUNS

Grupo 75

Realizado por:

Pedro Fernandes 84168, email: pedrocoelhofernandes@gmail.com

Ivan Andrushka 86291, email: ivanandrushka@gmail.com

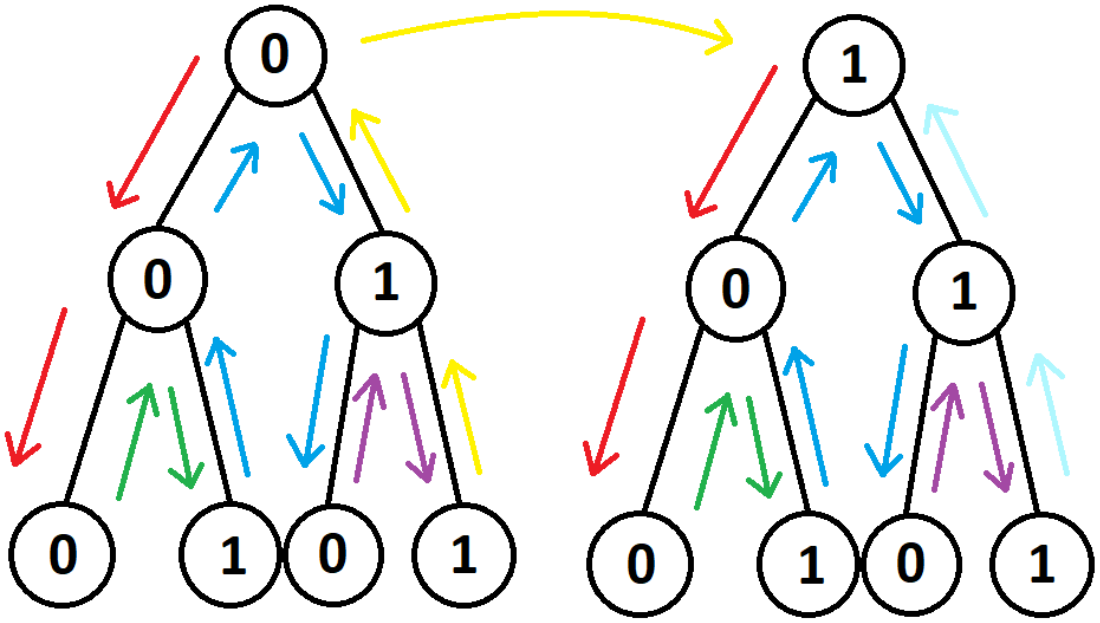
Conteúdo

Problema

O problema a resolver é um jogo de sudoku binário, nomeadamente, a implementação de um programa que resolve puzzles binários dada uma disposição inicial num campo cujo tamanho é NxN com N par (i.e. N= 4, 6, 8, etc). Ou seja, tendo um puzzle inicial que está parcialmente resolvido (ou não), o objectivo é resolver estes puzzles preenchendo os espaços vazios (representados pelo algarismo 9) com 0s e 1s até que, ou se chegue à solução final (puzzle totalmente preenchido) ou que se conclua que o puzzle não tem solução. É óbvio que o preenchimento dos puzzles não é aleatório, mas segue um conjunto de regras. As regras separam-se em 2 conjuntos: variante 1 e variante 2. Para a variante 1 não podem haver mais de 2 0s ou 1s seguidos e o número de 0s e 1s numa dada linha ou coluna não pode ultrapassar N/2. A variante 2 faz uso das mesmas restrições, impondo ainda a restrição de não poder ter linhas ou colunas repetidas.

Solução encontrada

Tendo em conta que os campos de jogo iniciais dos puzzles dados vão, em geral, estar parcialmente preenchidos, irão, em princípio, existir casas que podem ser preenchidas por aplicação direta das restrições. Quando deixam de existir soluções imediatas, passa-se a aplicar um algoritmo que funciona de forma semelhante ao algoritmo de procura em profundidade em árvores, sendo que no nosso caso escolheu-se utilizar uma lista simplesmente ligada para simular o funcionamento de uma árvore. No nosso caso, escolhemos inserir sempre 0s nas casas onde se tem de fazer uma decisão e testar se a matriz continua a ser possível de resolver. Quando a inserção de um 0 torna o jogo impossível, testa-se um 1 na mesma posição. Se este também falhar, regressa-se na árvore até se encontrar o último zero que se inseriu e muda-se este para 1 e retoma-se o processo de inserção de 0s. É de notar que sempre que se insere um valor no campo é necessário fazer o teste das restrições, tanto para verificar se o jogo não se tornou impossível como para preencher quaisquer valores diretos novos.



Esta figura representa o funcionamento do algoritmo de procura em profundidade que implementámos para um jogo impossível em que só há 2 decisões para se tomar. O algoritmo vai seguir as linhas vermelhas que partem da raiz com o valor 0. Sendo o 3º 0 impossível, altera-se para 1, que também dá impossível. Assim, é necessário voltar ao 2º 0 e alterá-lo para 1, e assim sucessivamente. Quando há necessidade de voltar ao 1º 0 inserido e alterá-lo para 1, visto que esse nó representa a raiz, facilita pensar que existem, na verdade, 2 árvores: uma cuja raiz é 0 e outra com a raiz a 1.

Este é o funcionamento básico do problema de inserção e procura que se pretendeu resolver, sendo mais à frente explicado como isto foi implementado em mais detalhe.

Arquitetura básica do programa

Nesta secção, adotou-se um formato um pouco diferente do normal. Decidimos copiar a função main() do código e explicar a função de cada um dos elementos que esta contém

```
int main ( int argc, char **argv )

{

    Definição de variáveis;

    max = readFile(nome do ficheiro);
```

A variável max guarda o valor máximo do tamanho. Este valor é obtido como retorno da função readFile(char *) que percorre o ficheiro que lhe é passado como argumento de entrada e calcula o tamanho máximo entre todas as matrizes do ficheiro. (fluxograma em anexo 1)

```
    Board = Allocate_Table(max);
```

Tendo calculado o tamanho máximo das matrizes presentes no ficheiro, aloca a memória para a maior matriz. A vantagem disto é que se pode utilizar essa matriz para resolver qualquer outro jogo mais pequeno. (fluxograma em anexo 2)

```
    f = OpenFile(nome do ficheiro, "r");
```

Abre o ficheiro em modo de leitura para que se possa processar a informação e resolver os jogos ao longo do tempo.

```
    Mutli_tables(nome do ficheiro,  f, Board);
```

Resolve os jogos do ficheiro 1 a 1 até que se chegue ao final do ficheiro. Para tal, chamam-se os seguintes algoritmos:

- Se os parâmetros do jogo forem válidos (i.e. variante igual a 1 ou 2 e tamanho do campo divisível por 2)
    1. **Fill_Table (ficheiro, tamanho, Board)** : Lê uma matriz do ficheiro e preenche a matriz alocada no array bidimensional Board
    2. **Allocate_Vector (tamanho, variante, Board, vetores**): Alocam-se vectores que serão utilizados para a contagem de 0s e 1s numa dada linha ou coluna e, no caso da variante 2, alocam-se vectores que vão guardar um número decimal associado às linhas ou colunas já preenchidas, para que se possam comparar linhas e colunas iguais
    3. **Run_test_check_ini (Board, tamanho, variante, vetores):** Aplica as restrições do jogo e preenche a matriz com os valores imediatos (fluxograma da função **Run_test_check(…)** em anexo 2. O funcionamento é análogo à exceção do facto desta não alocar e não inserir elementos na lista de alterações imediatas). Para

calcular o valor a ser preenchidos pelas restrições recorre-se à função **test_check()** cujo fluxograma se encontra no anexo 3.

4. **Solve_game (Board, tamanho, variante, vetores):** Resolve o jogo aplicando a lógica que foi explicada na secção "Solução encontrada" (fluxograma em anexo 1)
5. **Free_vector (variante, vetores):** No final de cada jogo desaloca-se toda a memória alocada para os vectores auxiliares, para que se possa proceder ao jogo seguinte

- Caso o jogo tenha parâmetros inválidos:
    1. Passa a matriz à frente lendo-a do ficheiro com fscanf e guardando-a numa variável auxiliar que faz, nada mais, do que descartar a informação

```
fclose(f);
```

Fecha o ficheiro.

```
Free(Board, max);
```

Liberta a memória alocada para a matriz.

```
return (0);
```

```
}
```

Estruturas de dados

Para a representação do jogo utilizou-se, por razões óbvias, uma matriz (array bidimensional). Foram consideradas 2 opções no que toca à alocação da matriz: podia-se percorrer o ficheiro 1 vez para se calcular o tamanho da matriz máxima no ficheiro e utilizar essa matriz de tamanho máximo para resolver todos os jogos, ou então, alocar uma matriz no início de cada jogo com o tamanho adequado, e, sempre que um jogo terminava, desalocava-se a matriz. Como visto na secção anterior, escolheu-se a 1ª opção. Isto deve-se ao facto da leitura do ficheiro ser praticamente instantânea pelo que a desvantagem é o facto de a utilização de memória ser mais alta em média. O problema associado à 2ª opção é que a alocação contínua de memória torna o programa mais lento, pois a alocação dinâmica de memória não é, verdadeiramente, uma operação elementar, visto que é necessário percorrer o stack de memória até que seja encontrado um bloco de tamanho adequado livre.

Para o problema da procura decidiu-se utilizar uma lista, que simula o funcionamento de uma árvore. Ou seja, aplica-se o algoritmo de procura em profundidade, mas apenas se alocam os nós cujos valores são determinados como possíveis, até que se prove o contrário (devido a violação das restrições), sendo que neste caso o nó é desalocado. Havia também a possibilidade de utilizar um vetor cujo tamanho seria o número de 9s presentes na matriz inicial. No entanto o consumo de memória será superior do que no caso da lista. O fator tempo não teve grande influência na escolha, visto que para a lista faz-se inserção e remoção no início da lista, levando a uma complexidade temporal de O(1). Para o vetor, o mesmo se faz mas para o final, pelo que a complexidade será a mesma. O mesmo raciocínio se aplica para a lista secundária, utilizada para guardar as alterações feitas na matriz por aplicação direta das restrições.

Recorreu-se também à utilização de vectores para guardar o número de 0s e 1s por linha e por coluna. Visto que o número de linhas e colunas das matrizes é sempre conhecido, a escolha de vectores para esta situação é óbvia, pois, sabendo o índice onde está contida a informação, o acesso à informação é feito com complexidade O(1).

Para a implementação da variante 2, decidiu-se guardar o número decimal associado à concatenação de todos os elementos de uma dada linha ou coluna em vectores. Inicialmente, o pretendido era uma implementação de uma hash table. Para tal, recorreu-se a 2 funções de hash distintas: resto da divisão por 2^N e o método multiplicativo de Knuth. No caso da primeira função, era necessário ter um vector de tamanho 2^N, o que não é exactamente eficiente para a memória. No segundo caso, ocorria um número significativo de colisões, levando-nos a descartar o método. Por fim, foi feita a implementação com vectores. Esta implementação, embora não seja muito eficiente em termos de tempo, é bastante eficiente em memória e bastante simples de implementar.

Descrição dos subsistemas funcionais

main.c

solve_game()

Returns: inteiro que é igual a 1 se a matriz tiver solução ou -1 se a matriz não tiver solução.
Descrição: esta função percorre a matriz, já preenchida com alguns elementos imediatos, à procura de um 9 (elemento por preencher). De seguida, preenche esse elemento com 0 e testa as suas imediatas e guarda-as numa lista. Se o programa verificar que 0 não é elegível para essa posição, apaga as imediatas da lista e troca 0 por 1. Se 1 também não for elegível para essa posição, as suas imediatas são apagadas e volta-se ao elemento inserido anteriormente. Se o mesmo for 1, as suas imediatas são apagadas e continua-se à procura do elemento anterior, se for 0 apagam-se as suas imediatas e altera-se para 1. O programa acaba quando não houver mais 9's por preencher ou se todos os valores forem impossíveis.

OpenFile()

Returns: ponteiro para ficheiro aberto.
Descrição: função que abre o ficheiro com o nome recebido como argumento de entrada.

readFile()

Returns: inteiro com o tamanho da maior matriz lida do ficheiro.
Descrição: função que percorre o ficheiro de entrada à procura do tamanho da maior matriz.

Fill_Table()

Descrição: função que lê uma matriz no ficheiro de entrada e guarda-a numa matriz (Board) alocada previamente.

check_board()

Returns: função que retorna inteiro que indica se a matriz preenchida obedece a todas as regras, 1 se obedece ou 0 caso contrário.
Descrição: função que percorre a matriz Board já preenchida e verifica se todos os valores obedecem às regras.

Multi_Tables()

Descrição: função que detecta uma matriz no ficheiro de entrada, resolve-a e escreve no ficheiro de saída a sua solução. Esta ação ocorre repetidamente até ao final do ficheiro de saída.

Allocate_Table()

Returns: duplo ponteiro para a matriz Board.
Descrição: função que aloca na memória uma matriz com o tamanho da maior matriz obtido através da função **readFile**.

Free()
Descrição: função que liberta a memória da matriz Board.

WriteFile()
Descrição: função que escreve no ficheiro de saída a solução do problema, consoante os dados obtidos a partir da função **solve_game**.

struct.c

add_front()
Returns: ponteiro para cabeça da lista de elementos inseridos.
Descrição: função que cria um novo nó e insere-o na lista de elementos inseridos, inicializando, também, o seu conteúdo.

add_front_coord()
Returns: ponteiro para cabeça da lista de imediatas.
Descrição: função que cria um novo nó e insere-o na lista de imediatas, inicializando, também, o seu conteúdo.

run_test_check_ini()
Descrição: função que percorre a matriz inicial e preenche-a com elementos imediatos que sejam possíveis obter sem a inserção de novos elementos.

run_test_check()
Returns: inteiro que indica se o teste é possível. Com valor 0 se for impossível ou 1 se for possível, ou seja, a matriz é preenchida.
Descrição: função que procura um elemento vazio na matriz e verifica se é possível preenchê-lo com 0 ou 1. Esta ação é executada até não haver mais alterações na matriz.

clean_Board()
Returns: ponteiro para cabeça da lista de imediatas.
Descrição: função que apaga todos os nós da lista de imediatas e repõe esses elementos a 9 na Board.

Reset()
Returns: ponteiro para cabeça da lista de elementos inseridos.

Descrição: função que apaga a cabeça da lista de elementos inseridos. Apaga também a lista de imediatas associada a esse elemento, através da função clean_board.

test0()

Returns: inteiro que indica se o teste é possível. Com valor 0 se for impossível ou 1 se for possível.
Descrição: função que insere 0 na posição escolhida e verifica se, segundo as regras, esse valor pode estar nessa mesma posição.

test1()

Returns: inteiro que indica se o teste é possível. Com valor 0 se for impossível ou 1 se for possível.
Descrição: função que insere 1 na posição escolhida e verifica se, segundo as regras, esse valor pode estar nessa mesma posição.

free_list()

Descrição: função que percorre a lista de elementos inseridos, apagando a lista de imediatas correspondente a cada nó e o próprio nó.

test.c

Allocate_Vector()

Descrição: Os seguintes vetores são alocados e inicializados:
***vec_cols_1:** número de 1's presente numa certa coluna;
***vec_cols_0:** número de 0's presente numa certa coluna;
***vec_lines_1:** número de 1's presente numa certa linha;
***vec_lines_0:** número de 0's presente numa certa linha.

free_vector()

Descrição: função que liberta a memória dos vectores auxiliares alocados anteriormente.

test_inicial()

Returns: inteiro que indica o valor que deve estar nessa posição, 0 ou 1, ou -1 caso não haja informação suficiente para selecionar 0 ou 1.
Descrição: função que retorna o valor lido diretamente da matriz.

test_pares_line_esquerdo()

Returns: inteiro que indica o valor que deve estar nessa posição, 0 ou 1, ou -1 caso não haja informação suficiente para selecionar 0 ou 1.

Descrição: função que verifica se existem dois 0's ou dois 1's à esquerda. Pelas regras do jogo, caso existam dois 1's, retorna-se 0. Caso existam dois 0's retorna-se 1.

test_pares_line_direito()

Returns: inteiro que indica o valor que deve estar nessa posição, 0 ou 1, ou -1 caso não haja informação suficiente para selecionar 0 ou 1.
Descrição: função que verifica se existem dois 0's ou dois 1's à direita. Pelas regras do jogo, caso existam dois 1's, retorna-se 0. Caso existam dois 0's retorna-se 1.

test_pare_column_baixo()

Returns: inteiro que indica o valor que deve estar nessa posição, 0 ou 1, ou -1 caso não haja informação suficiente para selecionar 0 ou 1.
Descrição: função que verifica se existem dois 0's ou dois 1's em baixo. Pelas regras do jogo, caso existam dois 1's, retorna-se 0. Caso existam dois 0's retorna-se 1.

test_pare_column_cima()

Returns: inteiro que indica o valor que deve estar nessa posição, 0 ou 1, ou -1 caso não haja informação suficiente para selecionar 0 ou 1.
Descrição: função que verifica se existem dois 0's ou dois 1's em cima. Pelas regras do jogo, caso existam dois 1's, retorna-se 0. Caso existam dois 0's retorna-se 1.

test_trios_line()

Returns: inteiro que indica o valor que deve estar nessa posição, 0 ou 1, ou -1 caso não haja informação suficiente para selecionar 0 ou 1.
Descrição: função que verifica se existem dois 0's ou dois 1's à esquerda e à direita. Pelas regras do jogo, caso existam dois 1's, retorna-se 0. Caso existam dois 0's retorna-se 1.

test_trios_column()

Returns: inteiro que indica o valor que deve estar nessa posição, 0 ou 1, ou -1 caso não haja informação suficiente para selecionar 0 ou 1.
Descrição: função que verifica se existem dois 0's ou dois 1's em cima e em baixo. Pelas regras do jogo, caso existam dois 1's, retorna-se 0. Caso existam dois 0's retorna-se 1.

test_complete_line()

Returns: inteiro que indica o valor que deve estar nessa posição, 0 ou 1, ou -1 caso não haja informação suficiente para selecionar 0 ou 1.
Descrição: função que verifica se o número de 0's ou de 1's na linha já atingiu o seu valor máximo, ou seja metade do tamanho da matriz.

test_complete_column()

Returns: inteiro que indica o valor que deve estar nessa posição, 0 ou 1, ou -1 caso não haja informação suficiente para selecionar 0 ou 1.

Descrição: função que verifica se o número de 0's ou de 1's na coluna já atingiu o seu valor máximo, ou seja metade do tamanho da matriz.

test_line()

Returns: inteiro que indica se existem linhas repetidas. -1 se não houver "colisões" ou 2 se houver "colisões".
Descrição: função que verifica se o valor decimal da linha já existe no vetor **repeat_lines**.

test_column()

Returns: inteiro que indica se existem colunas repetidas. -1 se não houver "colisões" ou 2 se houver "colisões".
Descrição: função que verifica se o valor decimal da linha já existe no vetor **repeat_cols**.

test_check()

Returns: inteiro que indica o teste é possível segundo as regras. 1 se for possivel 0 se for impossivel.
Descrição: função que executa todos os testes e verifica se há impossibilidades ou incoerências nos mesmos.

Definição de estruturas

Node - Lista de imediatas
struct Node* next; - Ponteiro para próximo nó na lista.
int x; - Inteiro que guarda a linha onde essa imediata se encontra.
int y; - Inteiro que guarda a coluna onde essa imediata se encontra.

Node_Binary - Lista de elementos inseridos na matriz
struct Node_Binary* next; - Ponteiro para o próximo valor da lista.
int test_0; - Inteiro que contém informação relativa ao teste com valor 0. Este inteiro tem valor 0 (inicializado), 10 (se o teste no 0 for impossivel) ou 11 (se o teste for possivel).
int test_1; - Inteiro que contém informação relativa ao teste com valor 1. Este inteiro tem valor 0 (inicializado), 10 (se o teste no 1 for impossivel) ou 11 (se o teste for possivel).
Node* change_ptr; - Ponteiro para a lista de imediatas.
int linha; - Inteiro que guarda o valor da linha onde o valor inserido se encontra.
int coluna; - Inteiro que guarda o valor da coluna onde o valor inserido se encontra.

Análise da complexidade

Complexidade temporal teórica

Variante 1

A função principal do programa é a função multitables, visto que é a função que resolve os puzzles. Como tal, esta função é a única função cuja complexidade vale a pena calcular.

Como a função multitables resolve todos os puzzles de um ficheiro, se existirem M puzzles num ficheiro de dimensão NxN, a complexidade desta função será O(M*O(Solve_game)). A função Solve_game consiste de um loop até que o jogo termine (seja considerado impossível ou a matriz esteja completamente preenchida). Visto que a complexidade deste loop depende, não só do tamanho da matriz, mas também da dificuldade do jogo e da disposição inicial do puzzle, pode ser difícil calcular exactamente a complexidade. Por exemplo, disposição inicial do puzzle pode ser bastante desfavorável, dependendo da implementação do algoritmo de procura em profundidade.

Considerando k elementos do puzzle já preenchidos, o pior caso para a procura será O(|V|-k), sendo V o número de nós. No limite o número de nós da nossa pseudo-árvore será o número 9 no puzzle inicial, N^2. Após a inserção de cada nó, pode ser necessário percorrer uma dada matriz várias vezes para o cálculo dos valores imediatos. Isto leva a uma complexidade de $O(i * N^2)$, onde i representa o número de vezes que é necessário percorrer a matriz.

Assim, a complexidade total da função multitables será, com |V|>>k e $\lim_{i \to N} i = N$:

$$O(|V|) * O(N^3) = O(N^5)$$

Pelo que a complexidade da função multitables, e assim, do programa é:

$$O(M) * O(N^5) = O(MN^5)$$

Variante 2

Para a variante 2, a análise será semelhante mas a função solve_game agora também corre o teste de verificação de linhas e colunas iguais, que consiste em percorrer 2 vetores de dimensão N. Assim, a complexidade para a variante 2 será:

$$O(M) * O(N^2) * O(2N * N^3) = O(M * 2N^6) = O(M * N^6)$$

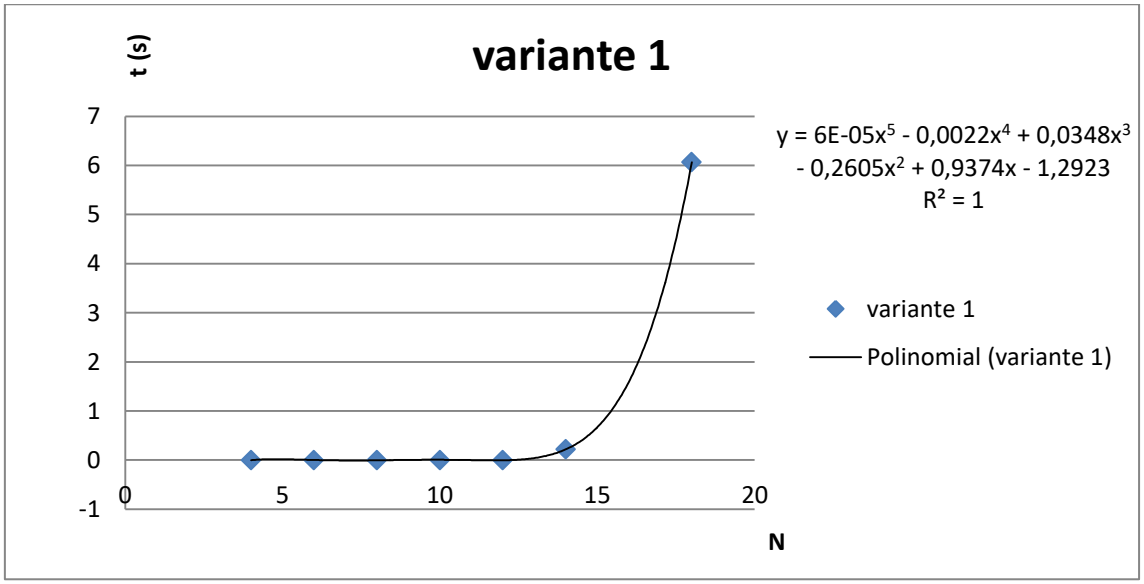
Complexidade temporal experimental

Com recurso a excel, fez-se uma análise experimental da complexidade temporal do programa. Para tal, recorreu-se à biblioteca time.h para se obter uma boa estimativa do tempo de execução. Foram utilizadas matrizes vazias (só com 9s) devido à dificuldade em criar puzzles específicos para os quais o programa teria de fazer mais iterações e só foi testado com 1 matriz, visto que a adição de mais matrizes no ficheiro implica um aumento linear na complexidade, pelo que é informação pouco exemplificativa da complexidade do algoritmo de resolução puzzles. Infelizmente, isto implica que a quantidade de testes que se podem fazer é limitada.

Variante 1

N	t (s)
4	0,000122
6	0,000179
8	0,000305
10	0,000305
12	0,000599
14	0,218281
18	6,065874

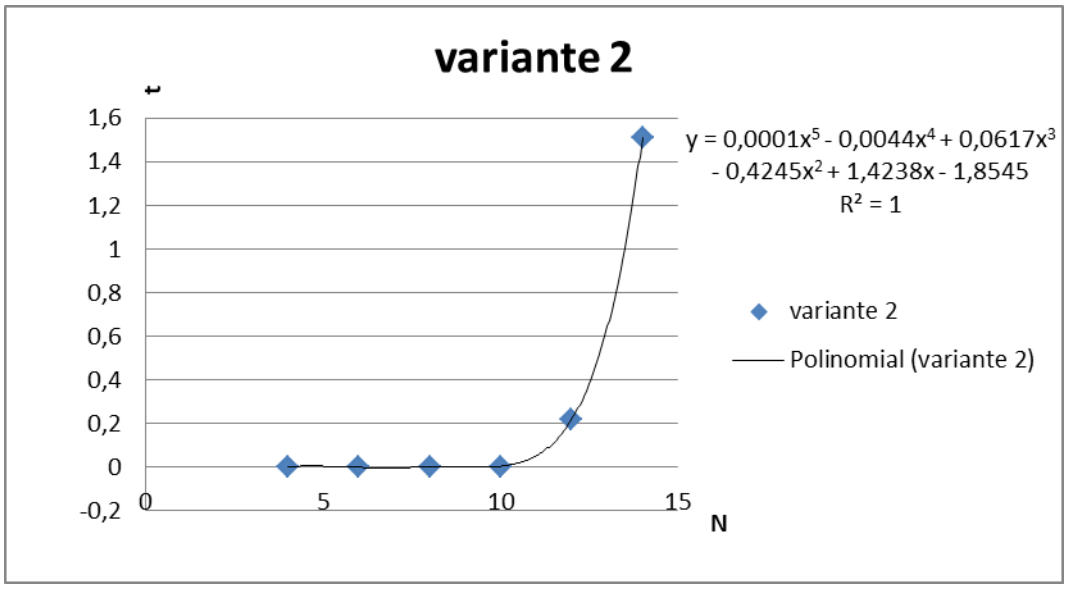
Para a variante 1 foi utilizada esta tabela de pontos para se traçar um gráfico t(N). Recorrendo à ferramenta de regressão do excel pode-se observar que os resultados experimentais não diferem fortemente dos resultados teóricos, sendo a complexidade experimental $O(N^5)$.



Variante 2

N2	t (s)
4	0,000191
6	0,000214
8	0,000339
10	0,005505
12	0,218944
14	1,513298

A mesma análise se fez para a variante 2. Os resultados obtidos revelam que a complexidade experimental deste algoritmo é $O(N^5)$. No entanto, é claramente visível que a variante 2 consome, como era esperado, bastante mais tempo, pelo que acredita-se que com o aumento de N, os resultados experimentais tendam para os resultados teóricos. Infelizmente não é possível analisar a complexidade experimental para valores de N maiores, pois o programa deixa de ser capaz de analisar matrizes de tamanho N>14 vazias, e, como foi dito anteriormente, utilizar matrizes com disposições iniciais diferentes leva a resultados pouco fiáveis, visto que a procura pode variar imenso.



Complexidade de memória teórica

Variante 1

Para um dado ficheiro com M matrizes, vai-se alocar a matriz de maior dimensão NxN. A resolução do puzzle requer a utilização de 4 vetores auxiliares, utilizados para guardar o número de 0s e 1s em cada linha e coluna, pelo que se reserva espaço proporcional a 4N. Para o algoritmo principal de procura tem-se uma lista de listas. No entanto, visto que um puzzle tem, no máximo, apenas NxN casas por preencher, esta lista nunca pode exceder uma dimensão de N^2, pois uma grande quantidade de valores imediatos implica poucos valores de decisão e vice-versa. Assim podemos concluir que a complexidade de memória será:

$$O(N^2) + O(4N) + O(N^2) = O(N^2 logN)$$

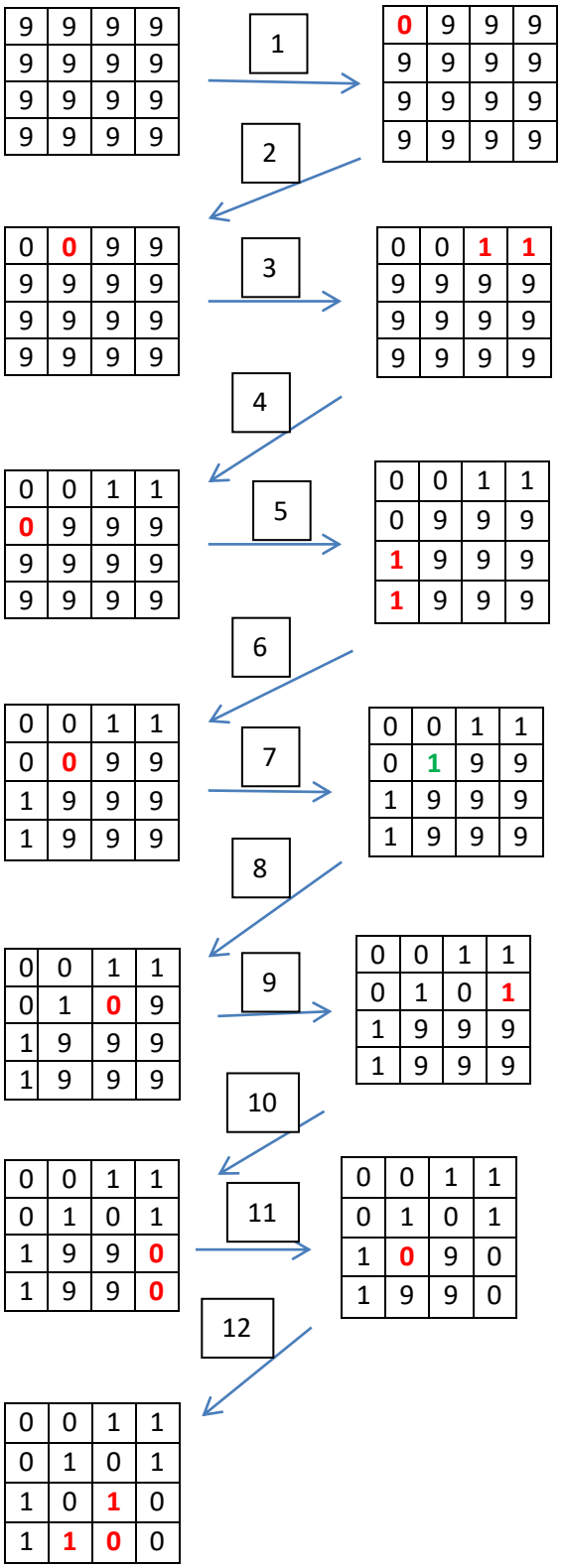
Variante 2

Para a variante faz-se mais do mesmo, mas agora também é necessário contabilizar os 2 vetores utilizados para guardar os valores associados às linhas e colunas, pelo que a complexidade de memória neste caso é:

$$(N^2) + O(6N) + O(N^2) = O(N^2 logN)$$

Exemplo do funcionamento do programa

Por simplicidade, será resolvido um puzzle 4x4 que demonstra o funcionamento do programa para a variante 2. O funcionamento para a variante 1 é análogo, mas com mais graus de liberdade.



0->1: Insere-se um 0 na posição (0,0)

1->2: Insere-se um 0 na posição (0,1)

2->3: Por aplicação da regra da paridade (número de 0s e 1s no máximo N/2) preenchem-se as posições (0,2) e (0,3) da matriz. A posição (0,2) também é preenchida pela regra dos duos (dois 0s ou 1s de seguida no máximo)

3->4: Insere-se um 0 na posição (1,0)

4->5: Pela regra dos duos e da paridade preenche-se a posição (2,0) e pela regra da paridade preenche-se a posição (3,0)

5->6: Insere-se um 0 na posição (1,1)

6->7: Por aplicação da regra da paridade as posições (1,2) e (1,3) seriam preenchidas com 1s, no entanto isso viola a restrição das linhas iguais. Assim, volta-se à última posição na qual se inseriu um 0 e testa-se um 1 na mesma posição

7->8: Insere-se um 0 na posição (1,2)

8->9: Pela regra da paridade preenche-se a posição (1,3) com um 1

9->10: Pela regra da paridade preenchem-se as posições (2,3) e (3,3) com 0s

10->11: Insere-se um 0 na posição (2,1)

11->12: Por aplicação das regras preenchem-se as restantes posições

Comentários e críticas

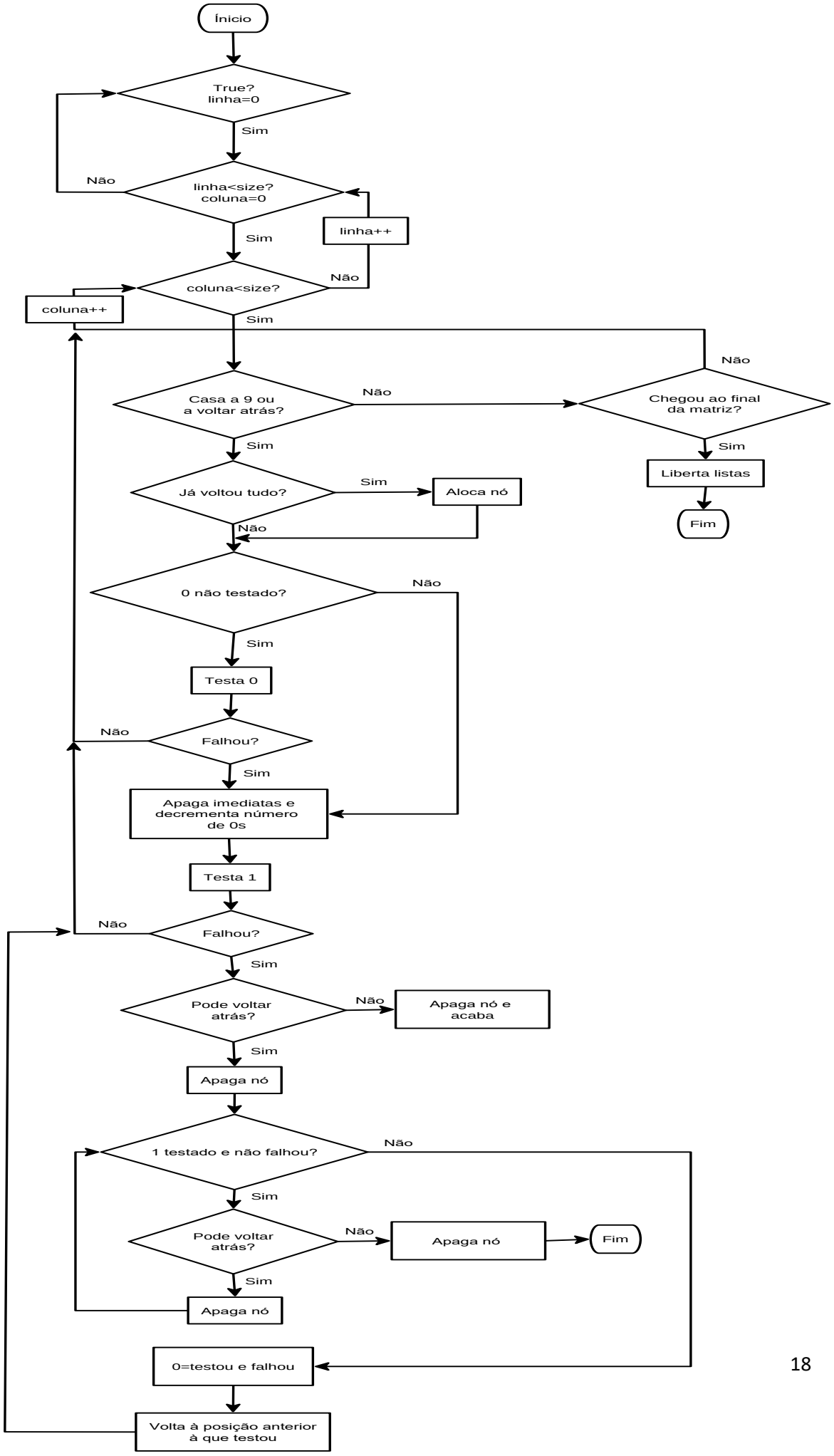
É também importante reparar num detalhe. Apesar de terem sido realizados todos os 20 testes da plataforma *Mooshak*, foi detectado um *bug* no nosso programa.
Numa das nossas submissões foram concluídos apenas 18 testes, onde recebemos uma indicação em que o nosso programa apresentava soluções para matrizes sem solução.
De modo a corrigir este problema implementou-se a função **check_board**. Ao executar esta função em todas as casas da matriz podemos verificar se existe algum erro na mesma. Este método é apenas uma maneira de contornar o *bug*, pelo que apenas solucionou parte do problema.
Reparou-se também que, em certas matrizes, o programa não apresenta solução para uma matriz onde era esperado haver solução.
Como por exemplo na matriz,

14 2

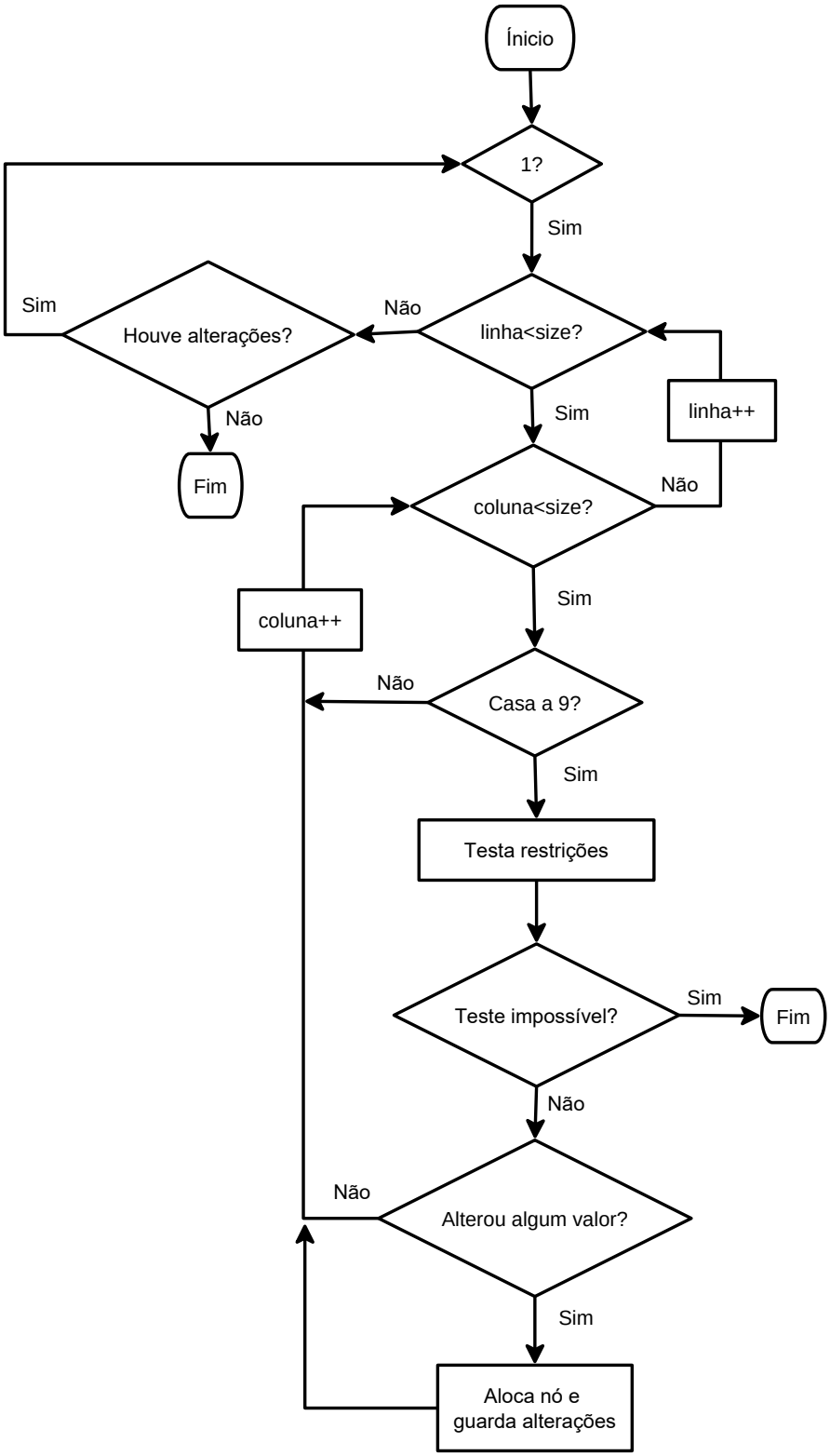
9	9	0	9	9	9	9	9	9	1	9	9	9	9
9	1	9	9	9	0	9	9	9	0	0	9	9	9
9	9	0	9	9	0	9	9	0	9	9	1	9	9
9	9	9	9	9	9	9	9	9	9	9	9	9	9
9	9	9	9	9	0	9	9	9	9	9	9	9	0
1	1	9	9	9	9	9	9	9	9	9	9	1	9
1	9	0	0	9	9	9	9	9	9	9	9	9	0
9	9	9	9	9	9	9	9	9	9	9	0	9	0
9	0	9	9	9	9	9	9	9	1	1	9	9	9
9	9	9	1	1	9	9	9	0	9	9	9	9	1
9	1	9	1	9	9	9	9	9	9	9	1	9	9
9	9	9	9	9	9	0	0	9	9	9	9	9	9
0	9	9	9	9	9	9	9	9	9	9	9	9	9
9	9	1	9	9	9	9	9	0	9	9	1	1	9

Após várias tentativas, não nos foi possível descobrir a causa do *bug*, pelo que achámos que devíamos reportar o mesmo.

Anexo 1 (fluxograma da função Solve_Game())



Anexo 2 (fluxograma da função run_test_check())



Anexo 3 (fluxograma da função test_check())