CURSO: ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

DISCIPLINA: ALGORITMOS E ESTRUTURAS DE DADOS

ZERUNS

Diogo Martins Alves Nº 86980 (diogo.m.alves@tecnico.ulisboa.pt)

Xavier Dias Nº 87136 (xavier.dias@tecnico.ulisboa.pt)

Data: 13/12/2017

ÍNDICE

PROBLEMA A RESOLVER

O problema que nos foi proposto resolver consistiu em construir um programa que, dado um ficheiro de extensão ".puz", contendo puzzles, produzisse um ficheiro de saída com extensão ".sol" com uma resolução de cada puzzle existente no ficheiro de entrada (se esta existisse) ou a informação de que o puzzle não tinha solução.

Os puzzles a resolver consistiam em matrizes quadradas de números inteiros (1, 0 ou 9), e o objetivo era substituir todos os 9 com 1 ou 0, tendo em conta as seguintes regras:

1. Não podem existir mais que dois uns ou dois zeros consecutivos em qualquer linha ou coluna;
2. Cada linha e cada coluna, depois de preenchida, possui exatamente o mesmo número de zeros e de uns;
3. Nenhuma linha pode ser repetida, assim como nenhuma coluna pode ser repetida.

Cada puzzle no ficheiro de entrada teria uma linha inicial, com dois números inteiros, em que o primeiro representa a dimensão da matriz (número de linhas e colunas) e o segundo, que podia ser 1 ou 2, representa a forma como o puzzle deveria ser resolvido: se fosse 1 apenas era necessário cumprir as regras 1 e 2; se fosse 2 era necessário cumprir as 3 regras.

COMO RESOLVEMOS O PROBLEMA

Para resolvermos o problema, começámos por observar que existiam soluções "triviais", isto é, valores que podíamos inserir no puzzle e que tínhamos a certeza estar corretos, simplesmente por aplicação das regras, por exemplo, se encontrássemos dois 1 seguidos na mesma linha, sabíamos imediatamente que ao

lado tinha que haver um zero. Ao preenchermos estes valores no puzzle, iriam surgir possivelmente mais soluções triviais. No entanto, na maior parte das vezes, isto não é suficiente para resolver completamente o puzzle, uma vez que chegamos a situações em que não existem mais soluções triviais. Nestas situações, o que tínhamos a fazer era guardar uma cópia da configuração do puzzle naquele momento e tomar uma decisão: inserir um 0 ou um 1 (no nosso programa preenchemos em primeiro lugar sempre 1) numa casa onde existisse um 9 e continuar com o algoritmo anterior, preenchendo as soluções triviais. Este processo para quando encontramos uma impossibilidade, isto é, quando encontramos uma posição que, de acordo com as regras, não pode ser ocupada por um 0 nem por um 1. Neste caso o que temos a fazer é voltar à última configuração guardada e preencher, onde tínhamos preenchido anteriormente com 1, agora com 0.

Seguindo sucessivamente estes passos, o programa pode terminar de duas formas: encontrando uma configuração do puzzle em que todas as entradas se encontram preenchidas com 0 e 1, o que corresponde a uma solução válida do puzzle ou chegando a uma situação em que a configuração atual é impossível e não existem mais configurações guardadas, o que significa que o puzzle não tem solução.

ARQUITETURA DO PROGRAMA

No programa que desenvolvemos, foi criado o módulo *files.c*, com o objetivo de manipular os ficheiros de entrada e saída. Neste está definida a função que permite abrir o ficheiro de entrada e criar o ficheiro de saída (*openFiles()*), a função que permite ler um puzzle do ficheiro de entrada (*readDataFromFile()*), e a que permite escrever no ficheiro de saída a solução de um puzzle (*writeOutputFile()*).

Desenvolvemos também o módulo *puzzlemngt.c*, cujo objetivo é estabelecer todas as funções necessárias à resolução de um puzzle, fazendo proveito de uma

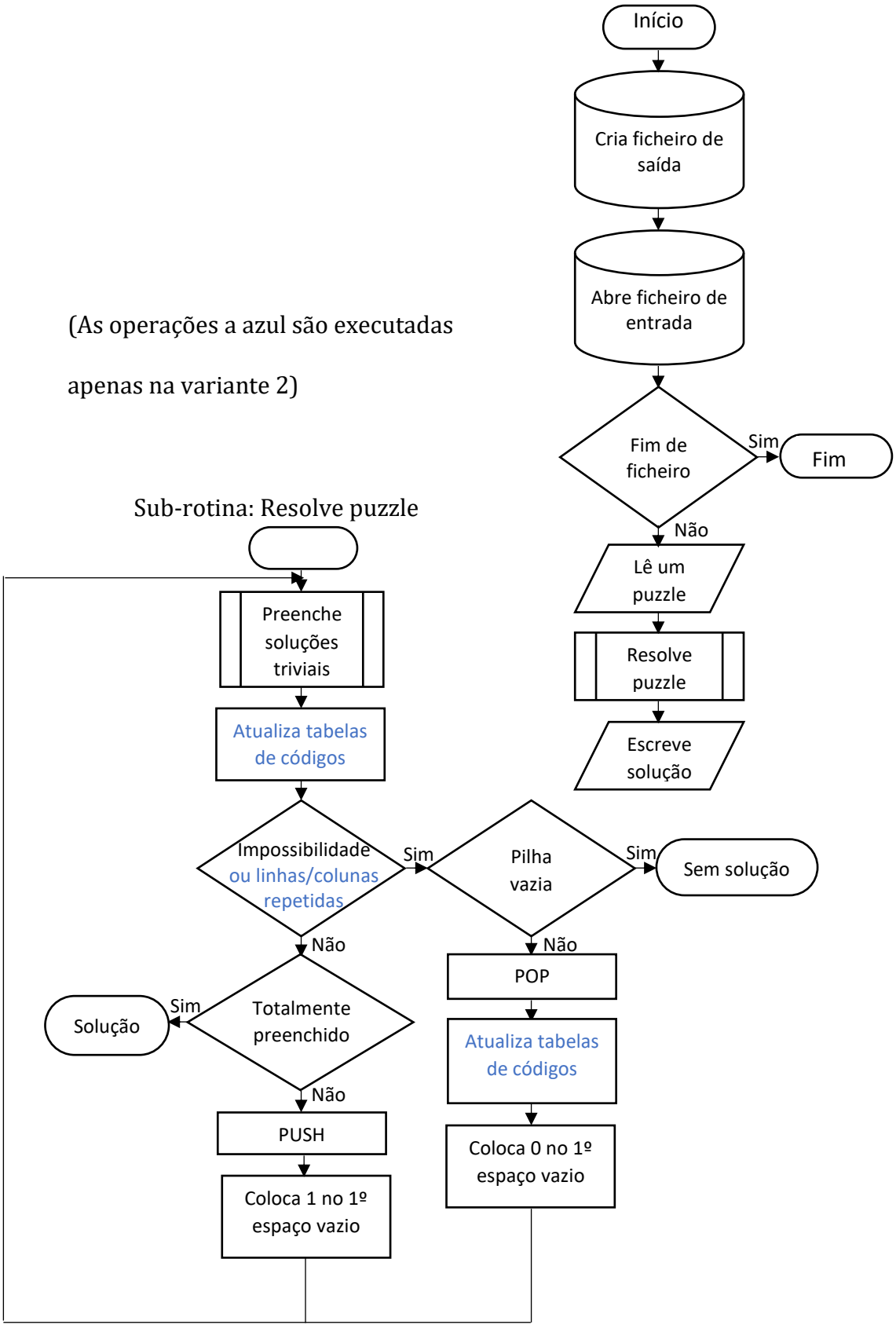
pilha (definida em *stack.h*) e de um tipo de dados denominado inteiro de tamanho arbitrário (definido em *bigInt.h*). Neste módulo estão definidas, entre outras, a função que permite preencher um puzzle com as suas soluções triviais (*fillTrivialSolutions()*), a função que verifica se existem linhas ou colunas repetidas (*checkRepeatedValues()*), as funções que atualizam a informação que permite determinar se existem linhas e colunas repetidas (*UpdateColmnCodeArray()* e *UpdateColmnCodeArray()*) e a função que permite resolver o puzzle, fazendo uso das funções anteriores (*solvePuzzle()*). As restantes funções presentes neste módulo servem de auxilio a estas (*checkAdjacent()*, *isNotSolution()*, *firstEmptySpace()*, *columnCode()*, *lineCode()* e *createCodeArray()*).

Por fim, desenvolvemos o módulo *zeruns.c*, onde se encontra a função *main()* que faz proveito dos módulos anteriores para ler, resolver e escrever a solução de um ou mais puzzles.

Em seguida apresenta-se o fluxograma do nosso programa:

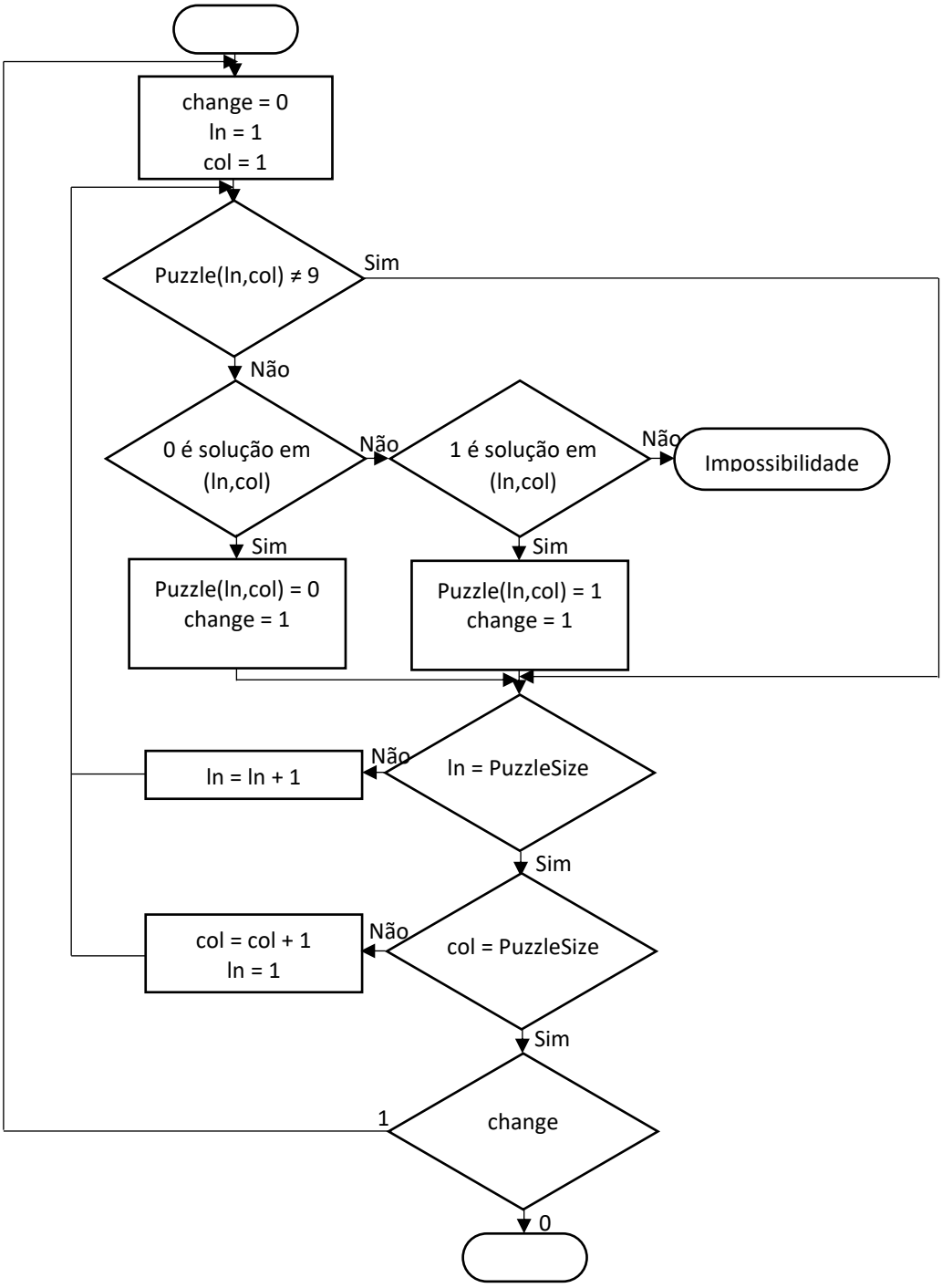
zeruns

(As operações a azul são executadas apenas na variante 2)



Sub-rotina: Resolve puzzle

Sub-rotina: Preenche soluções triviais



TIPOS DE DADOS UTILIZADOS

No nosso programa os tipos de dados utilizados foram pilhas e tabelas.

A pilha (definida em *stack.h* e implementada em *stack.c*) foi utilizada para guardar as sucessivas configurações do puzzle, sendo guardada uma cópia da configuração do puzzle na pilha antes de preenchermos com '1' uma posição onde não tínhamos a certeza ser essa a solução e sendo retirada da pilha a última cópia guardada sempre que descobrimos que seguimos pelo caminho errado. Desta forma, a pilha funciona como um suporte para o varrimento de uma árvore implícita, árvore esta que seria impossível de representar em memória devido à enorme quantidade de memória associada.

São utilizadas duas tabelas para guardar inteiros de tamanho arbitrário (bigInt), em que cada entrada representa a codificação de uma linha/coluna que esteja completamente preenchida. A partir desta informação é possível determinarmos se existem linhas ou colunas repetidas através da comparação dos elementos da tabela. Optámos por utilizar uma tabela, neste caso, uma vez que tínhamos que fazer bastantes leituras, comparações e escritas, mas não remoções visto que as entradas correspondentes a linhas/colunas não completamente preenchidas ficam simplesmente com o valor *NULL*.

ALGORITMOS UTILIZADOS

Dos algoritmos dados na disciplina, utilizámos o algoritmo de procura em profundidade (*Depth First Search*) para visitar os vértices da árvore implícita onde estão contidas as várias combinações legais de puzzles possíveis a partir do puzzle inicial, cujas soluções triviais se encontram preenchidas: partindo de um vértice inicial, seguimos em primeiro lugar para o vértice que tem um 1 na primeira posição vazia e continuamos desta forma até encontrarmos uma folha, que corresponde ou a uma solução possível do puzzle ou uma configuração que não seja possível

resolver. Neste último caso, temos que voltar atrás na árvore e optar pelo caminho que tem um 0 na primeira posição vazia e continuar o processo a partir daí. Esta procura termina quando encontramos uma folha que corresponda a uma solução possível do puzzle inicial.

SUBSISTEMAS FUNCIONAIS

Neste programa estão presentes três subsistemas funcionais:

Stack:

O objetivo deste subsistema é estabelecer uma interface para criar e manipular uma pilha (LIFO) de elementos do tipo abstrato *Item* (definido pelo utilizador).

Os tipos de dados abstratos utilizados neste subsistema estão contidos nos módulos *stack.h* (nó da pilha) e *item.h* (payload de cada nó da pilha).

As funções deste subsistema estão presentes no ficheiro *stack.c*:

- *Stack* StackInit()* - Função que inicializa uma pilha;
- *void Push(Stack **s, Item i)* - Função que insere um elemento numa pilha;
- *Item Pop(Stack **s)* - Função que retira da pilha o primeiro elemento;
- *int IsEmpty(Stack *s)* - Função que indica se uma pilha está ou não vazia.

Puzzle:

Neste subsistema pretende criar-se uma interface que permita ao utilizador criar e manipular elementos do tipo puzzle, que representa um puzzle lido de um ficheiro ".puz".

O tipo de dados abstrato utilizado neste subsistema está definido no ficheiro *puzzle.h* .

As funções deste subsistema estão presentes no ficheiro *puzzle.c*:

- *void freePuzzle(puzzle** puz)* - Função para libertar um elemento do tipo puzzle;
- *puzzle* initPuzzle(int size, int** grid)* - Função para inicializar um novo puzzle;
- *puzzle* copyPuzzle(puzzle* puz)* - Função para efetuar uma cópia de um puzzle;
- *int getElement(puzzle *puz, int ln, int col)* - Função para obter determinado elemento de um puzzle;
- *int insertElement(puzzle *puz, int ln, int col, int val)* - Função para inserir em determinadas coordenadas de um puzzle um elemento (val);
- *int puzzleSize(puzzle* puz)* - Função para obter a dimensão do puzzle;
- *int** puzzleGrid(puzzle* puz)* - Função para obter a matriz do puzzle;
- *int numberOfXInLine(puzzle* puz, int x, int ln)* - Função para obter a quantidade de vezes que um certo número aparece em determinada linha;
- *int numberOfXInColmn(puzzle* puz, int x, int col)* - Função para obter a quantidade de vezes que um certo número aparece em determinada coluna.

<u>bigInts:</u>

Neste subsistema pretende criar-se uma interface que permita ao utilizador criar e manipular elementos do tipo bigInt, que representa um número inteiro de tamanho arbitrário.

O tipo de dados abstrato utilizado neste subsistema está definido no ficheiro *bigInts.h* .

As funções deste subsistema estão presentes no ficheiro *bigInts.c*:

- *bigInt* newBigInt(int num_bits)* - Função para inicializar um inteiro de tamanho arbitrário a 0;
- *void addBitRight(bigInt* b_int, int bit)* - Função para adicionar um bit à direita de um inteiro de tamanho arbitrário;

- *int cmpBigInts(bigInt* first, bigInt* second)* - Função que informa se dois inteiros de tamanho arbitrário são iguais ou diferentes;
- *void freeBigInt(bigInt** b_int)* - Função que liberta um interio de tamanho arbitrário.

ANÁLISE DOS REQUISITOS COMPUTACIONAIS DO PROGRAMA

COMPLEXIDADE

A função que irá determinar a complexidade do programa é a função *solvePuzzle()*, bem como as funções invocadas por esta:

Sendo N a dimensão do puzzle a resolver e V o número de espaços vazios que este tem (tem-se $V \leq N^2$).

Na tabela seguinte encontram-se as complexidades das funções invocadas diretamente pelo ciclo principal da função *solvePuzzle()* (a sublinhado), bem como das funções invocadas por estas:

numberOfXInColmn() *numberOfXInLine()*	O(N)
lineCode() *columnCode()*	O(N)
UpdateLineCodeArray() *UpdateColmnCodeArray()*	$O(N^2)$ em pior caso
cmpBigInts()	O(N/32), que se pode considerar O(1) nesta situação
checkRepeatedValues()	$O(N^2)$
isEmpty()	O(1)
Pop()	O(1)

firstEmptySpace()	O(N^2/V) em caso médio
insertElement()	O(1)
emptySpaces()	O(N^2)
Push()	O(1)
isNotSolution()	O(N)
fillTrivialSolutions()	entre O(V*N) e O(V^2 * N)

O ciclo principal da função *solvePuzzle()* é executado entre 1 e 2^V vezes.

Como estas funções são invocadas um número constante de vezes em cada ciclo, é a função *fillTrivialSolutions()* que vai determinar a complexidade do programa.

Temos duas situações extremas: quando a função *fillTrivialSolutions()* resolve o puzzle por completo, a sua complexidade é no pior caso O(V^2 * N), mas o ciclo principal da função *solvePuzzle()* é executado apenas uma vez, logo a complexidade do programa é **O(V^2 * N)** (para V^2>N, caso contrário a complexidade é O(N^2)). Este é o caso em que a complexidade é menor.

A outra situação extrema é quando a função *fillTrivialSolutions()* nunca resolve nenhuma solução. Neste caso a sua complexidade é O(V*N), mas o ciclo principal de *solvePuzzle()* pode ser executado até 2^V vezes (que corresponde ao varrimento completo da árvore implícita). Neste caso a complexidade do programa será **O(V * N * 2^V)**. Esta situação corresponde ao pior caso em termos de complexidade.

Se assumirmos, por hipótese, que em média, por cada solução dada por *solvePuzzle()*, a função *fillTrivialSolutions()* é capaz de encontrar uma solução trivial, isto significa que metade das soluções são preenchidas pela função *fillTrivialSolutions()* e a outra metade pela função *solvePuzzle()*, então a complexidade de *fillTrivialSolutions()* será O(V*N), e o ciclo principal é executado no máximo $2^{(V/2)}$ vezes, logo a complexidade do programa será **O($2^{(V/2)}$ * V * N)**.

MEMÓRIA

Da mesma forma, a função que irá determinar os requisitos de memória do programa é a função *solvePuzzle()* e todas as funções que forem invocadas por esta.

Na tabela seguinte encontram-se os requisitos de memória das funções invocadas pela função *solvePuzzle()*:

createCodeArray()	O(N)
newBigInt()	O(N)
lineCode() *columnCode()*	O(N)
UpdateLineCodeArray() *UpdateColmnCodeArray()*	$O(N^2)$ em pior caso
Push()	O(1)
copyPuzzle()	$O(N^2)$

Tal como anteriormente temos dois casos extremos: quando o ciclo principal de *solvePuzzle()* é executado apenas uma vez, dá-se o melhor caso, em que os requisitos de memória do programa são de apenas $\mathbf{O(N^2)}$.

Na situação correspondente ao pior caso, em que a árvore implícita é varrida completamente, o ciclo principal de *solvePuzzle()* é executado no máximo 2^V vezes e, portanto os requisitos de memória do sistema são no máximo $\mathbf{O(N^2 * 2^V)}$.

Considerando o caso médio anterior, os requisitos de memória nessa situação são no máximo $\mathbf{O(N^2 * 2^{V/2})}$.

ANÁLISE CRÍTICA DO FUNCIONAMENTO DO PROGRAMA

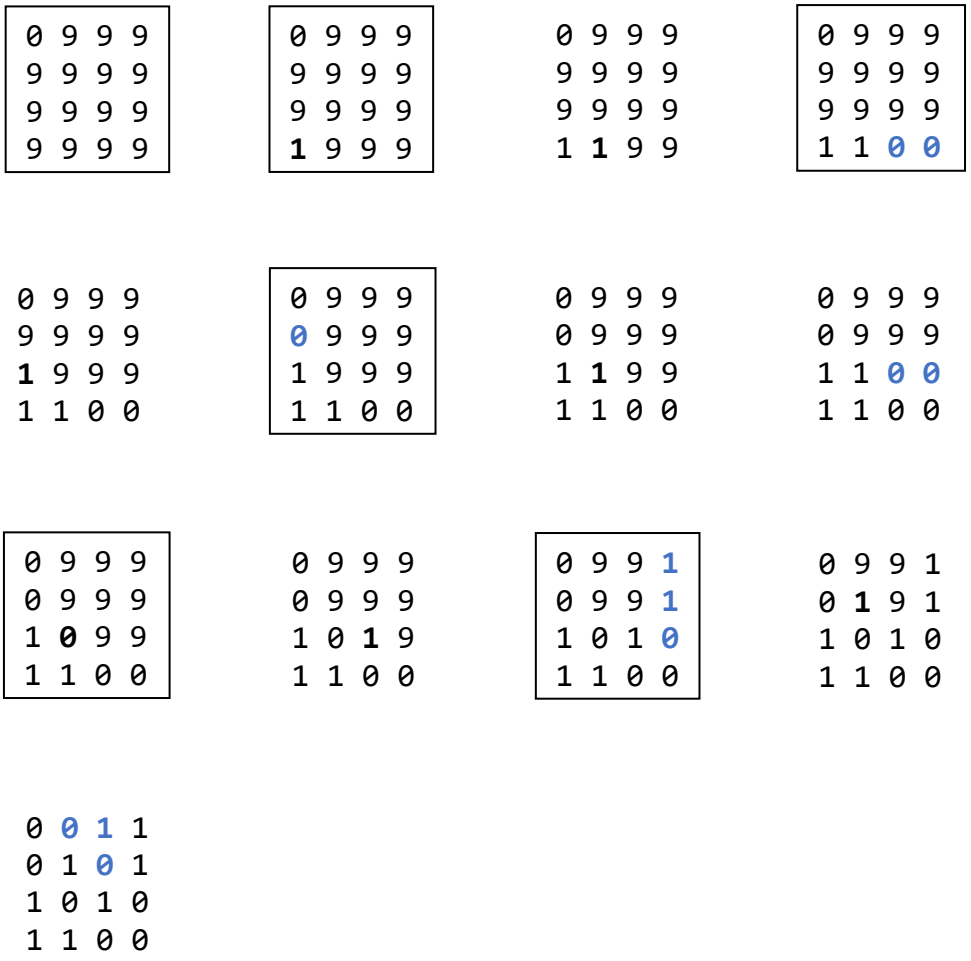
O programa que desenvolvemos deve ser capaz de, dado um ficheiro com puzzles, ser capaz de os resolver ou informar que não existe solução em qualquer situação. A questão que se põe é se o faz em tempo útil. Pelos nossos testes, o nosso programa consegue resolver puzzles de tamanho 14, maioritariamente vazios, em poucos segundos. Puzzles de tamanho 16 já demoram alguns minutos a serem resolvidos. Esta discrepância é consequência de a complexidade crescer com um fator proporcional a 2^V, em pior caso, em que V é o número de espaços vazios. É importante realçar que este problema tem que ter este tipo de complexidade obrigatoriamente, uma vez que não existe nenhuma maneira sistemática de resolver puzzles sem ser varrendo a árvore implícita onde estes estão contidos.

EXEMPLO DE APLICAÇÃO

Para simplificar, iremos mostrar o funcionamento do programa para o seguinte puzzle de dimensão 4, na variante 2:

```
4 2
1 9 9 9
9 9 9 9
9 9 9 9
9 9 9 9
```

Seguidamente apresentam-se as sucessivas configurações do puzzle inicial produzido pelo nosso programa até chegar à solução. A azul estão as soluções triviais produzidas por *fillTrivialSolutions()*, a negrito as hipóteses de solução produzidas por *solvePuzzle()* e contornados a preto estão os puzzles que são carregados para a pilha:



Nota: no passo 8 é detetado que existem duas linhas iguais e é carregado da pilha o último puzzle guardado, que é preenchido com 0 na primeira posição vazia (passo 9).