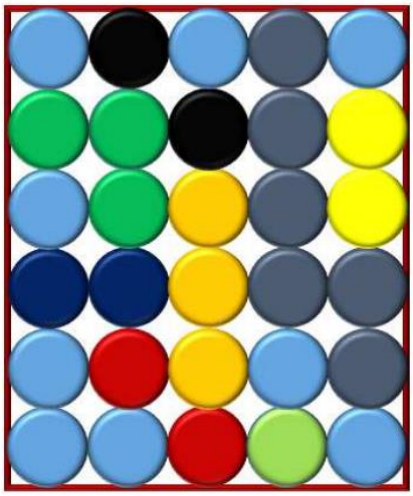
TÉCNICO LISBOA

MESTRADO INTEGRADO EM ENGENHARIA ELETROTÉCNICA E DE COMPUTADORES

ALGORITMOS E ESTRUTURAS DE DADOS

Relatório do Projeto



BURSTER

Identificação dos Alunos:

Nome: Luís Afonso Lopes <luis.afonso.lopes@tecnico.pt> **Número:** 84116

Nome: Miguel Freire <miguel.freire@tecnico.pt> **Número:** 84145

Docentes:

Prof. Carlos Bispo

Prof. Pedro Miraldo

2º Semestre 2016/2017

CONTEÚDO

1. DESCRIÇÃO DO PROBLEMA

O programa desenvolvido tem por objetivo resolver puzzles constituídos por diversas cores. Grupos de elementos da mesma cor (juntos não-diagonalmente) formam clusters, que devem ser removidos para resolver com sucesso o puzzle.

Neste projeto havia 3 variantes de problema:

- **Variante 1** – Remoção simples de clusters
- **Variante 2** – Procura da sequência de jogadas que leva a atingir um limiar mínimo de pontuação
- **Variante 3** – Procura da sequência de jogadas que leva a atingir o máximo de pontuação

As especificações do programa, tal como entendidas por nós, são as seguintes:

- O programa deverá ser chamado burster e ser invocado na linha de comandos como **aed$ ./burster <problema>.puz**

  Onde **<problema>** é o nome do ficheiro com o puzzle a ser resolvido. Este ficheiro tem obrigatoriamente de ter a extensão **".puz"**. A sua primeira linha é na forma L C V, onde L é o número de linhas, C é o número de colunas e V a variante do problema a resolver. As linhas seguintes especificam a estrutura do puzzle.

- O programa deve criar um ficheiro de solução com o mesmo nome que o ficheiro de entrada mas com extensão **".moves"**. A primeira linha deste ficheiro deve ser igual à primeira linha do ficheiro de entrada, e segunda linha deve apresentar o número de lances realizados e a pontuação total obtida. As seguintes linhas devem imprimir os lances realizados para resolver aquele problema. Caso não exista solução a segunda será "0 -1".

2. FORMA DE ABORDAGEM AO PROBLEMA

De uma forma abstrata o problema é análogo ao de percorrer uma árvore.

Neste caso, os nós da árvore são cada puzzle gerado por cada lance e as arestas são esses mesmos lances.

Assim para resolver problemas de variante:

1) É apenas necessário fazer jogadas até não existirem mais jogadas possíveis, ou seja, percorrer um ramo da árvore até ao fim. (Depth First Search com restrição de paragem)

2) É necessário fazer jogadas até atingir ou ultrapassar uma certa pontuação. É o mesmo que percorrer a árvore até satisfazer uma certa condição (pontuação ≥ pontuação de limiar). Caso se chegue a um puzzle sem mais jogadas possíveis e não foi atingida a pontuação mínima é necessário voltar à matriz anterior e realizar um novo lance. (Depth First Search com backtracking)

3) É necessário fazer todas as jogadas possíveis e guardar o conjunto de jogadas que chega à pontuação máxima, que é o mesmo que percorrer toda a árvore e guardar o caminho que leva à pontuação máxima. Assim e como o tamanho de uma árvore deste género cresce exponencialmente o problema pode tornar-se um pesadelo a nível de memoria (NP-HARD) sem forma de poder ser resolvido em tempo polinomial.

   É necessário então implementar uma nova condição *a-priori* para ignorar certos ramos da árvore que de certeza não nos levam à solução desejada.
   Essa condição para este problema é a seguinte:

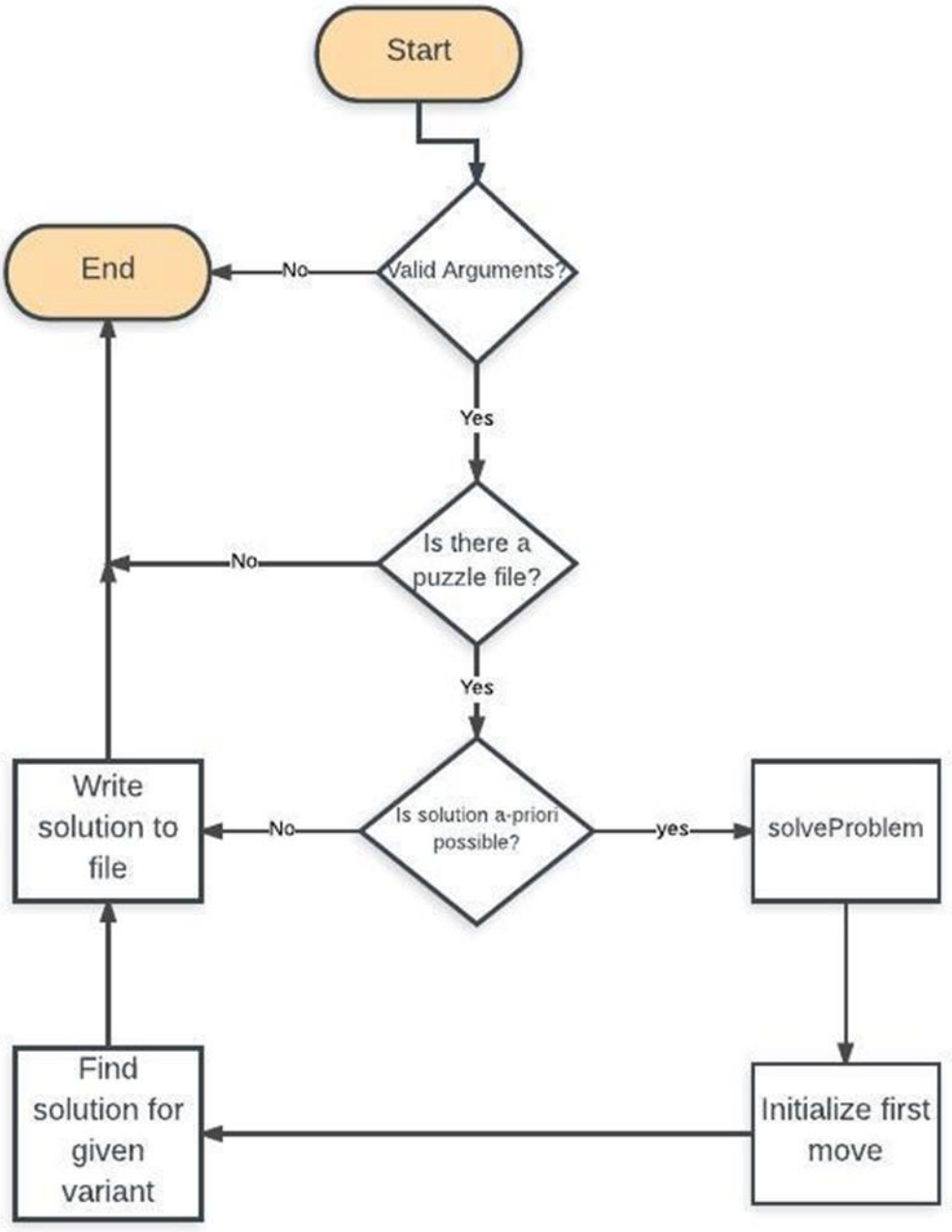
$$Pa + Pp > Pmo$$

   - Pa → Pontuação atual
   - Pp → Pontuação máxima que é possível obter através da realização daquele lance. É dada pelo cálculo da remoção de todos os clusters se todos os elementos da mesma cores estivessem juntos.
   - Pm → Pontuação máxima obtida até aquele lance

É necessário então um algoritmo de Depth First Search com restrição de paragem, backtracking e branch-and-bound.

3. DESCRIÇÃO DA ARQUITETURA DO PROGRAMA

3.1. FLUXOGRAMA



3.2. ESTRUTURAS DE DADOS

1. Jogada (Move)

Nos ficheiros Move.h e Move.C está a interface e implementação da estrutura de dados principal do programa. É através desta estrutura que é possível guardar as informações de cada jogada realizada, tais como: o puzzle gerado pela jogada, a pontuação acumulada com a jogada, o número de lances disponíveis e a pontuação máxima possível de obter em cada puzzle.

É usado um vector de Moves para especificar uma árvore, onde cada índice do vector indica o nível da árvore que estamos a trabalhar. É depois através da aplicação de um algoritmo de DFS com backtrack e branch-and-bound que é possível resolver o problema.

2. Matriz (Matrix)

Matrix é uma estrutura de dados que implementa uma tabela bidimensional de inteiros. É usada para guardar a representação numérica dos puzzles e outra informação necessária à sua manipulação tal como o número de linhas, colunas e um vector de estados que indica se existe alguma coluna sem elementos. A sua especificação e implementação estão feitas nos ficheiros matrix.h e matrix.c respectivamente.

3. Cluster

Cluster é uma lista ligada, onde cada elemento da lista é um Cluster diferente.

Pode ser vista como o todos os lances disponíveis. Nela também são guardadas informações sobre o cluster como o seu tamanho, a cor do elemento que constitui o cluster e uma outra lista ligada de ClusterNode's que listam a localização na matriz de todos os elementos pertencentes a um dado cluster.

É através da remoção deles elementos da matriz que se realizam jogadas e geram-se novos elementos do vector Move.

4. Path

Path é uma estrutura de dados que apenas guarda um par de coordenadas (2 inteiros). Neste problema é usando um array deste tipo para guardar os lances realizados para satisfazer a variante do problema.

3.3. ALGORITMOS UTILIZADOS

1. **Depth First Search com backtracking**

O principal algoritmo do programa, diretamente responsável por resolver o problema é um algoritmo de procura em profundidade recursivo com backtracking. Não foi possível programar a parte do código de variante 3 com branch-and-bound, assim, em baixo, só está especificado o código que resolve variante 1 e alguns problemas de variante 2.

A sua implementação é feita no ficheiro Move.C com o nome findSolution().

De uma forma geral o algoritmo faz uma procura em profundidade à árvore do problema até satisfazer uma certa condição de acordo com a variante pedida. Caso chegue a um extremo da árvore e a condição não foi satisfeita este volta atrás (backtracking) até à jogada anterior com lances disponíveis e volta a fazer uma jogada (chama recursivamente esta função na nova jogada gerada). Este processo é realizado até a condição dada pela variante do problema ser satisfeita.

Seja M o vector de jogadas disponível:

```
Procedure FindSolution(M[k])
  If variant 2 condition is satisfied
        Set Found true
  If there are no available move's return
  Else:
    While there are move's available:
       Do a new move
       FindSolution(M[k+1]);
       If variant is -1 or 0 return
       If found is true return

end
```

2. Algoritmo de localização de clusters numa matriz

O algoritmo usado para a localização de clusters numa matriz está implementado no ficheiro Clusters.C com o nome getCluster().

É um algoritmo recursivo que varre a matriz à procura de elementos não-diagonalmente juntos. Este varre cada elemento da matriz uma só vez, guardando a indicação que já visitou aquele elemento num vector de estados.

Os elementos iguais são então guardados numa lista ligada de ClusterNode's formando assim um Cluster.

```
Procedure getCluster(x,y)
    If (x,y) was already visited return
    Set (x,y) visited
    Create new ClusterNode(M[x][y])
    Insert ClusterNode in Cluster
   If element on top of (x,y) is equal
      getCluster(x-1,y)
   If element on the right of (x,y) is equal
      getCluster(x,y+1)
   If element on bottom of (x,y) is equal
      getCluster(x-1,y)
   If element on the left of (x,y) is equal
      getCluster(x, y-1)

end
```

3.4. DESCRIÇÃO DOS SUBSISTEMAS

1. Burster

Burster é o principal módulo do programa, é aqui que a execução do programa é realizada e é onde ocorre o processamento e validação dos argumentos passados para o programa.

O módulo é constituído pelo ficheiro burster.c onde está a função main() do programa e duas outras funções: uma para resolver o problema pedido e outra para imprimir a solução para ficheiro.

void printSolution(FILE *outFile, int rows, int cols, int objective, int numMoves, int score, Path *path)

Imprime a solução para um ficheiro.

void solveProblem(int rows, int cols, int variant, FILE *puzzleFile, FILE *solutionFile)

Função responsável por inicializar todas as variáveis necessárias para aplicar o algoritmo de procura de solução e chamar o mesmo.

2. Cluster

Este módulo é responsável pela criação, manipulação e remoção de Clusters de puzzles. A sua interface está presente no ficheiro cluster.h e a sua implementação no ficheiro cluster.c.

É constituída por 12 funções: 2 de inserção, 2 de procura, 1 de remoção, 2 de libertação de memória e 2 funções auxiliares para aplicar arranjos na matriz após ser feita uma jogada.

ClusterNode *newClusterNode(int x, int y):

Cria um novo elemento do cluster com coordenadas (x,y).

Cluster *newCluster():

Cria um novo cluster.

void getCluster(Matrix *M, int x, int y, int element, int *visited, Cluster *C, int elements[MAX_NUM_ELEMENTS]);

Encontra todos os elementos iguais não-diagonalmente juntos de (x,y) e cria um cluster.

Cluster *getPuzzleClusters(Matrix *matrix, Cluster *clusters, int *n, int *maxScore);

Encontra todos os clusters presentes numa matriz.

Cluster *removeCluster(Matrix *M, Cluster *C, Path *path, int *score)

Remove o cluster C da matriz M.

void freeCluster(Cluster *C);

Liberta a memória alocada para o cluster C (lista ligada).

void freeClusterGroup(Cluster *C);

Liberta a memória alocada para o grupo de clusters C (lista ligada).

void fixPuzzle(Matrix *M);

Aplica o efeito gravítico à matriz, todos os elementos -1 vão para o topo dos outros elementos.

void shiftMatrix(Matrix *M, int y);

Empurra todas as colunas vazias para a esquerda.

3. Matrix

Módulo responsável pela criação, manipulação e remoção de matrizes de inteiros.

Matrix *newMatrix(int rows, int cols);

Cria uma nova matriz de tamanho (rows*cols) de inteiros.

int getNumRows(Matrix *M);

Retorna o número de linhas da matriz.

int getNumCols(Matrix *M);

Retorna o número de colunas na matriz.

void insertElement(int element, Matrix *M, int row, int col);

Insere um inteiro na linha row e na coluna col.

void cloneMatrix(Matrix *A, Matrix *B);

Clona a matriz A para a Matriz B, i.e copia o conteúdo de A para B.

void freeMatrix(Matrix *M);

Liberta a memória alocada para a matriz.

```
void readMatrixFromFile(Matrix *matrix, FILE *file, int rows, int cols);
```

Lê uma matriz de um ficheiro e guarda-a na matriz matrix.

4. Moves

Este módulo é responsável pela manipulação do tipo de dados Move e pela execução do algoritmo de procura de solução. A sua interface está no ficheiro move.h e a sua implementação no ficheiro move.c.

```
void initMove(Move *firstMove, FILE *puzzleFile, int rows, int cols);
```
Inicializa o move inicial, correspondente ao puzzle inicial.

```
Move *newMove(int rows, int cols);
```
Cria um novo move.

```
void freeMoves(Move **moves, int numMoves);
```
Liberta a memória alocada pelo move.

```
void findSolution(Move **moves, int k, int objective, int *numMoves,
Path *path, int *found)
```
Procura por uma solução ao problema utilizando o algoritmo descrito em 3.3.1.

5. Util

Módulo com algumas funções úteis que ajudaram no desenvolvimento do programa. A sua interface está presente no ficheiro util.h e a sua implementação no ficheiro util.c.

```
void *smalloc(const size_t size);
```
Aloca memória dinâmica de forma segura usando o malloc.

```
void *scalloc(const size_t n, const size_t size);
```
Aloca memória dinâmica de forma segura usando o calloc.

```
void *srealloc(void *pointer, const size_t newSize);
```
Realoca memória de forma segura usando o realloc.

```
FILE *sfopen(const char *fileName, const char *mode);
```
Abre um ficheiro de forma segura, i.e. faz a verificação da sua existência.

```
int hasExtension(const char *fileName, const char *extension);
```

Verifica se um ficheiro tem uma certa extensão.

char *changeFileExtension(const char *fileName, const char *newExtension);
Muda a extensão de um ficheiro.

void resetArray(int *array, int size);
Coloca todos os elementos de um array de tamanho size a 0.

int convert(int x, int y, int n);
Converte um par de coordenadas 2D (x,y) para o seu respectiva representação em 1D numa matriz com n linhas.

4. Análise dos Requisitos Computacionais do Programa

A nível temporal o programa demora pouco tempo a procurar cada solução já que utiliza algoritmos recursivos que melhoram bastante a complexidade temporal neste tipo de problemas de varrimento de matrizes e de árvores. Tem no entanto algumas funções com complexidade $O(n^2)$ e $O(n^3)$ como é o caso das funções para corrigir o puzzle após ter sido realizada uma jogada. Assim o tempo irá aumentar de forma polinomial com o aumento da dimensão da matriz para os algoritmos de manipulação desta, e em tempo exponencial- $O(2^n)$ para o algoritmo de procura de solução. O uso de branch-and-bound melhora o tempo necessário para encontrar o problema.

A nível de memória o programa é pesado. O programa aloca inicialmente um vector de Moves com o tamanho máximo possível (Nº Linhas * Nº Colunas / 2). Cada vector contém uma matriz de tamanho (L*C), o que leva a um uso muito grande de memória. Na submissão eletrónica o nosso programa tinha um uso de memoria dos maiores presentes na tabela de submissão, com um pico de memória na ordem dos 100MB.

5. EXEMPLO DE EXECUÇÃO

Abaixo encontra-se um exemplo da execução do programa para a variante 1.

Seja exemple.puz um ficheiro de problemas com o seguinte conteúdo

```
3 3 -1
2 2 1
2 2 1
1 1 2
```

O programa é chamado para resolver este problema:

```
$ ./burster exemple.puz
```

1. O programa começa por validar os argumentos (verificar se o ficheiro existe e tem a extensão correta);
2. Inicializa as variáveis do problema (linhas, colunas, variante);
3. Chama a função solveProblem para aquele puzzle;
4. Inicializa as variáveis para a resolução do problema (moves, path);
5. Inicializa a primeira jogada correspondente ao puzzle inicial (moves[0]);
6. Como a variante é -1 chama a função findSolution(moves[0]);
7. Enquanto for possível realizar jogadas a função findSolution(moves[k+1]) é chamada de forma recursiva, guardando sempre a jogada no vetor path em cada chamada. Quando o programa chegar a um puzzle sem mais jogadas possíveis i.e. sem nenhum cluster disponível para ser removido ele termina;
    1. Remove o cluster (1,1) → 2 pontos;

```
-1-1 1
2 2 1
2 2 2
```

    2. Remove o cluster (1,1) → 20 pontos;

```
-1-1-1
-1-1  1
-1-1  1
```

    3. Remove o cluster (1,3) → 2 pontos;

```
-1-1-1
-1-1-1
-1-1-1
```

    4. Não é possível fazer mais lances->Termina (24 pontos);

8. Escreve a solução no ficheiro de saída;

```
3 3 -1
3 24
1 1
1 1
1 3
```

9. Liberta a restante memória alocada;
10. Caso existam mais problemas a serem resolvidos volta ao ponto 2. Caso não existam mais problemas o programa termina;

6. CRITICA AO PROGRAMA

O programa implementado resolve bem problemas de variante 1. No entanto, quando testado para problemas da variante 2, onde é necessário atingir uma determinada pontuação mínima, este só funciona corretamente caso consiga atingir essa pontuação no primeiro varrimento da árvore. Caso contrário, quando é necessário voltar um nível atrás para procurar novos clusters, a jogada seguinte não é executada corretamente, sendo esse o erro detetado no mooshack (o programa não produz lances em problema que os tem). A variante 3 não está a funcionar uma vez que não se encontra implementada.