Laboratórios de Informática II

Geração de Tabuleiros com Solução Única

Nelson Miguel de O. Estevão A76434
Pedro Alexandre Gonçalves Ribeiro A85493
Rui Filipe Moreira Mendes A83712

Junho 2018

Introdução

No âmbito da Unidade Curricular *Laboratórios de Informática II* foi-nos proposto a criação de um programa capaz de gerar tabuleiros para o jogo - **GandaGalo** - desenvolvido durante esta mesma Unidade Curricular.

Os tabuleiros deste jogo podem conter três tipos de peças, tal como se pode verificar na figura seguinte:

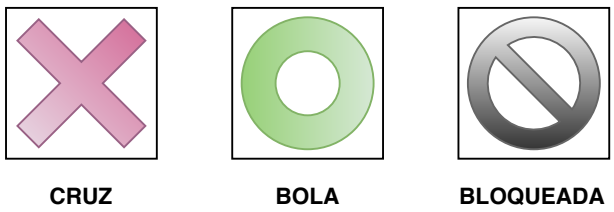

Figura 1: Tipos de Peças

Os tabuleiros são matrizes que podem ou não ser quadradas. Estes estão compreendidos entre 1 a 20 quadriculas (tanto linhas como colunas). O objetivo é completar o tabuleiro, ou seja, preencher todas as quadriculas com uma *cruz* ou com uma *bola*. No final, não podem existir mais de duas bolas nem duas cruzes seguidas em qualquer uma das direções, horizontal, vertical e diagonal.

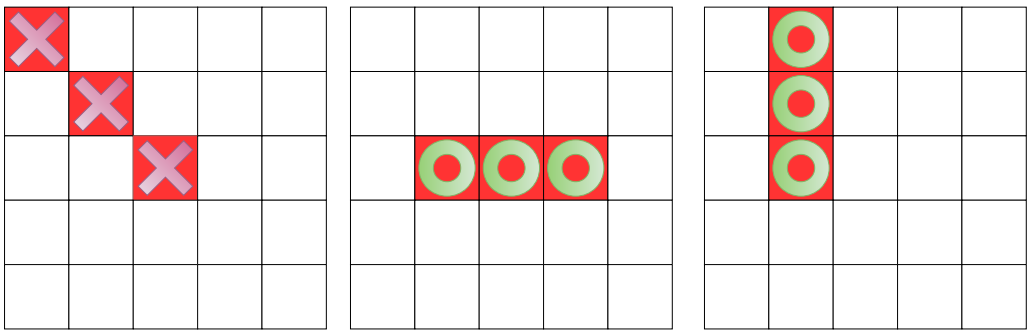

Figura 2: Exemplos de casos inválidos

Não existe qualquer restrição para as peças bloqueadas, ou seja, não existe qualquer limitação de quantas podem estar seguidas. As bloqueadas interrompem as sequências de cruzes ou bolas na sua intersecção. Contudo, não é permitido ao jogador colocar uma peça bloqueada. Estas existem num tabuleiro ou não no início e nenhuma outra pode ser acrescentada durante.

Estas devem ser colocadas estrategicamente em locais que nenhuma outra seria possível.

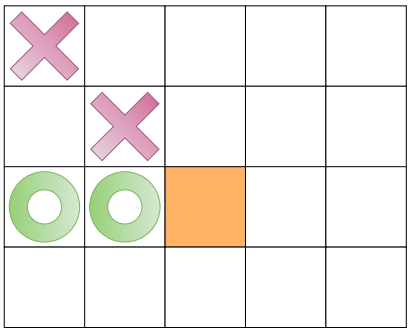

Figura 3: Tabuleiro impossível de resolver

Qualquer tabuleiro que tivesse uma configuração inicial do tipo representado na Figura 3 seria de resolução impossível, uma vez que, quer se coloque uma bola na casa laranja quer se coloque cruz passaria a estar num dos casos representados na Figura 2. Um tabuleiro não é impossível de resolver se apresentar uma situação de resolução impossível logo à partida mas sim, se a sua configuração leve a uma serie de jogadas forçadas que resultam num caso impossível. No caso em concreto, passaríamos a estar num caso válido se estivesse uma peça bloqueada nesse lugar, tal como está representado na Figura 4.

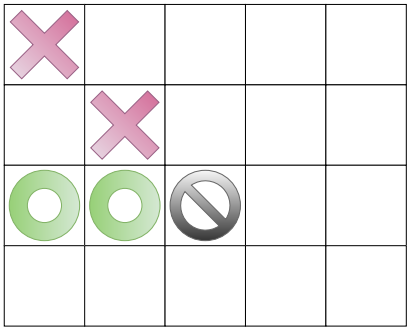

Figura 4: Correção possível para tabuleiro impossível

Contudo, um tabuleiro só é verdadeiramente válido se, para além de ser de possível resolução, tiver uma e uma só possível solução. Este relatório descreve como conseguimos criar um programa capaz de gerar tabuleiros com uma única solução recebendo apenas a dimensão desejada e a dificuldade esperada.

Implementação

O programa `gerar` recebe três argumentos à sua chamada:

- dificuldade;
- número de linhas;
- número de colunas.

A dificuldade pode variar entre fácil (1) e difícil (2). O número de linhas e o número de colunas podem variar entre 1 e 20[1]. Assim sendo, começamos por construir a nossa estrutura de dados que servirá para a construção do tabuleiro. A sua definição pode ser vista no seguinte bloco de código:

```
typedef struct puzzle {
  int num_lins, num_cols;
  int validade;
  int sizeU, sizeR, numAncs;
  LISTA undo, redo;
  char grelha[MAX_GRID][MAX_GRID];
} PUZZLE;
```

Os dois primeiros inteiros representam a dimensão do tabuleiro e são inicialmente atribuídos com os argumentos do programa. Ao inteiro `validade` é atribuído o valor `VALIDO`.

```
typedef enum {INVALIDO, IMPOSSIVEL, VALIDO} VALIDADE;
```

A lista `undo` e `redo` são definidas como listas ligadas com três inteiros, em que o `x` e o `y` são a posição da jogada e o `a` é o número da ancora em que o puzzle estava no momento da jogada.

```
typedef struct lista {
    int x, y, a;
    struct lista *next;
} *LISTA;
```

Os inteiros `sizeU`, `sizeR` e `numAncs` indicam o tamanho da lista `undo`, da lista `redo` e o número de ancoras até ao momento, respetivamente. Cada elemento da grelha tem um `VALOR` associado que pode ser descrito pelo bloco de código seguinte.

[1] O `MAX_GRID` é definido como 20.

```
typedef enum { BLOQUEADA,
               FIXO_X, FIXO_O,
               VAZIA,
               SOL_X, SOL_O
             } VALOR;
```

Cada `VALOR` está diretamente ligado a uma peça apresentada na Figura 1. A diferença entre um `FIXO_X` e um `SOL_X` é que o primeiro é uma peça do próprio tabuleiro, e a segunda uma peça jogada. Na construção do tabuleiro o `SOL_X` e o `SOL_O` serão usadas como auxiliares.

No nosso programa, após verificarmos a validade dos argumentos declaramos um tabuleiro e inicializamos as posições com o valor `VAZIA`.

```
for (i = 0; i < board->num_cols; i++)
  for (j = 0; j < board->num_lins; j++)
    board->grelha[i][j] = VAZIA;
```

Uma das primeiras características que o nosso programa precisava de ter era uma certa aleatoriedade. Não seria de todo interessante se dados os mesmos argumentos, obtivéssemos o mesmo tabuleiro. Para tal, usando a função `rand()` e uma divisão modular pelo número de colunas desejadas pelo o utilizador, escolhemos a coluna (`x`) da casa em que iremos jogar. O mesmo se aplica para a linha (`y`).

```
int x = rand() % board->num_cols;
int y = rand() % board->num_lins;
```

Para a escolha do tipo de peça que será colocada recorremos a uma estratégia semelhante.

```
int k = rand() % 2;
if (board->grelha[x][y] == VAZIA)
  board->grelha[x][y] = k == 1 ? FIXO_X : FIXO_O;
```

No caso do `if statement` der negativo, voltamos a tentar encontrar outra casa aleatoriamente usando os mesmos processos, sendo que nunca é colocada uma peça num local onde o tabuleiro ficaria invalido, ou seja, se encontrar uma peça vazia onde não poderá ser colocado um `FIXO_X` então é colocada um `FIXO_O` nessa posição e vice-versa. Graficamente, começamos com um tabuleiro vazio e fazemos uma jogada aleatória nesse tabuleiro. Isto não compromete a validade do tabuleiro porque para qualquer dimensão entre 1 e 20, com nenhuma peça, existem várias soluções possíveis. Supondo que o

tabuleiro pretendido tem dimensão 3 por 3, a Figura 5 representa o como o puzzle está e como fica após aplicada a estratégia descrita anteriormente.

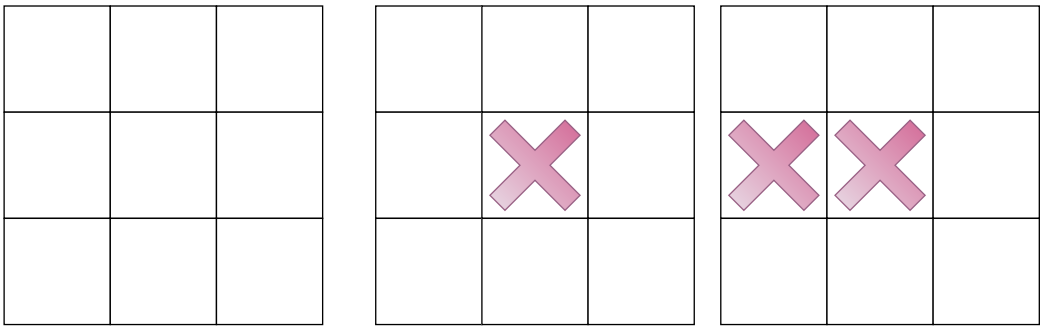

Figura 5: Jogada aleatória na posição (1,2) seguida de jogada aleatória na posição (2,1)

Nesta fase, ainda não existem peças suficientes para fazer jogadas definitivas. Seja qual for a casa escolhida, tanto uma cruz como uma bola pode levar a um tabuleiro possível de resolver. Desta forma, voltamos a fazer uma jogada aleatória usando a mesma estratégia. Suponhamos que a posição escolhida foi (2,1) com uma cruz. Nesse momento, existe uma jogada obrigatória na posição (3,2) que é uma bola. Essa jogada obrigatória é feita com um `SOL_O`[2], uma vez que é só uma peça auxiliar que não constará no tabuleiro final (nesta posição existe só uma possibilidade e por isso a ausência desta não levará a nenhuma solução dupla).

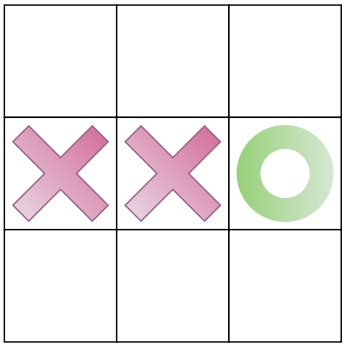

Figura 6: Jogada obrigatória feita na posição (3,2)

Nesta fase, volta a não existir uma jogada obrigatória e, por isso, é atribuída uma jogada aleatória. Começaremos com o nível de dificuldade fácil. Suponhamos que essa jogada é na posição (1,2) uma bola. Uma vez que esta não é

[2]As peças auxiliares (`SOL_X` e `SOL_O`) têm contorno branco.

invalida e não cria jogada óbvia, volta-se a repetir o procedimento. Se calhar uma bola na posição (3,3), passamos a ter um jogada obrigatória na casa (3,1) que terá o valor de `SOL_X`.

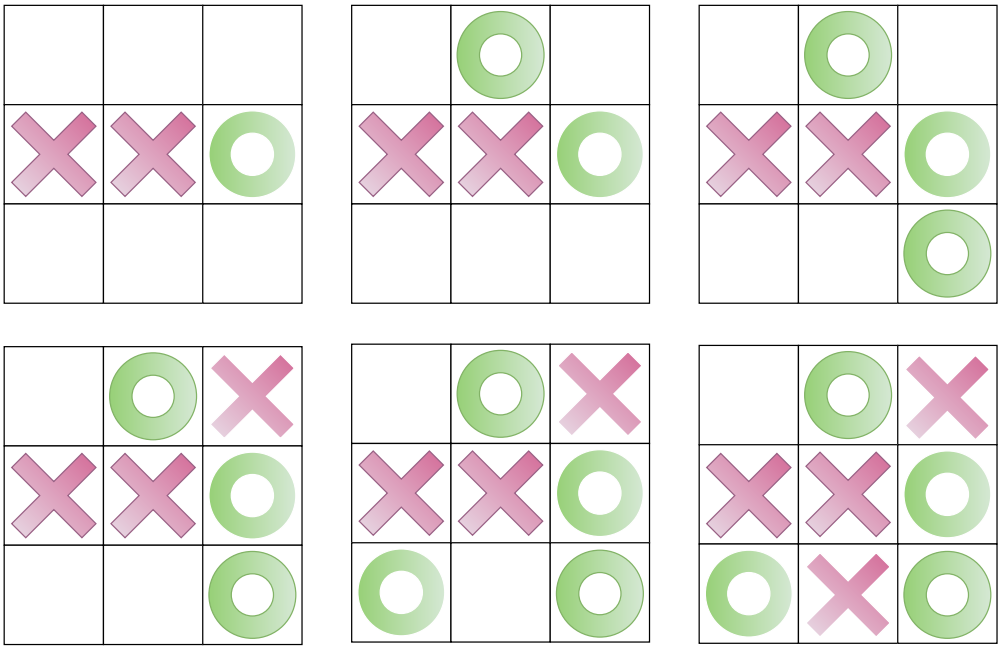

Figura 7: Etapas de construção de tabuleiro fácil

Nesta fase, o procedimento é repetido levando à situação em que na posição (1,1) é indiferente qual o tipo peça colocada.

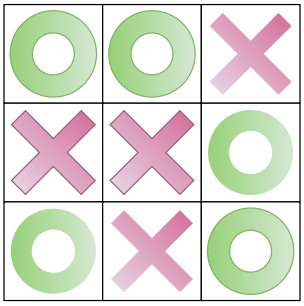

Figura 8: Tabuleiro fácil completo

Este processo é executado através da função `solver_easy`. Esta função tentará para as casas vazias colocar quer uma cruz quer uma bola e verificar

se torna o tabuleiro inválido. Caso assim seja, é porque poderá ter encontrado que naquela casa que só a peça contrária é valida. Se encontrar tal para os dois, é porque terá que ser bloqueada. A sua implementação pode ser lida no próximo bloco de código.

```
int solver_easy (PUZZLE *board, int *x, int *y)
{
  int i, j, found, foundX, found0;
  i = j = found = foundX = found0 = 0;
  for (i = 0; i < board->num_cols && !found; i++)
    for (j = 0; j < board->num_lins && !found; j++)
      if (board->grelha[i][j] == VAZIA) {
        board->grelha[i][j] = SOL_X;
        if (!validaPeca (board, i, j))
          found0 = 1;
        board->grelha[i][j] = SOL_0;
        if (!validaPeca (board, i, j))
          foundX = 1;
        if (foundX && found0) {
          found = 2; *x = i; *y = j;
          board->grelha[i][j] = BLOQUEADA;
        } else if (foundX) {
          board->grelha[i][j] = SOL_X;
          found = 1; *x = i; *y = j;
        } else if (found0) {
          board->grelha[i][j] = SOL_0;
          found = 1; *x = i; *y = j;
        } else board->grelha[i][j] = VAZIA;
      }
  return found;
}
```

Após o tabuleiro ser verificado como completo, todas as peças auxiliares são removidas do resultado final. O tabuleiro resultante está representado na Figura 9.

A diferença entre a construção de tabuleiros fáceis e difíceis é que, não só são acrescentadas peças bloqueadas e fixas aleatórias ao tabuleiro fácil sendo a quantidade dessas mesmas peças dependente da "área" do tabuleiro, como também casos que eram considerados como não tendo uma jogada óbvia no tabuleiro fácil apenas é gerado uma nova peça fixa até voltar a haver jogadas óbvias, enquanto que no difícil são procuradas peças que não são de jogada

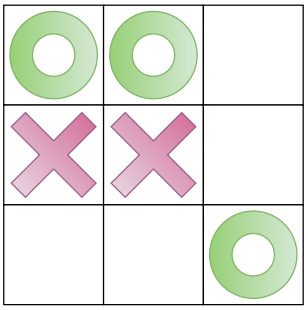

Figura 9: Tabuleiro fácil de dimensão 3 por 3

imediata mas são peças que tem se der um determinado tipo para que o mapa seja válido, acrescentando assim a necessidade de suposição aos tabuleiros mais difíceis.

Seguindo o raciocínio da Figura 10, vamos supor que na casa (3,2) colocamos um `SOL_O` e vamos verificar se isso nos leva a um caso impossível.

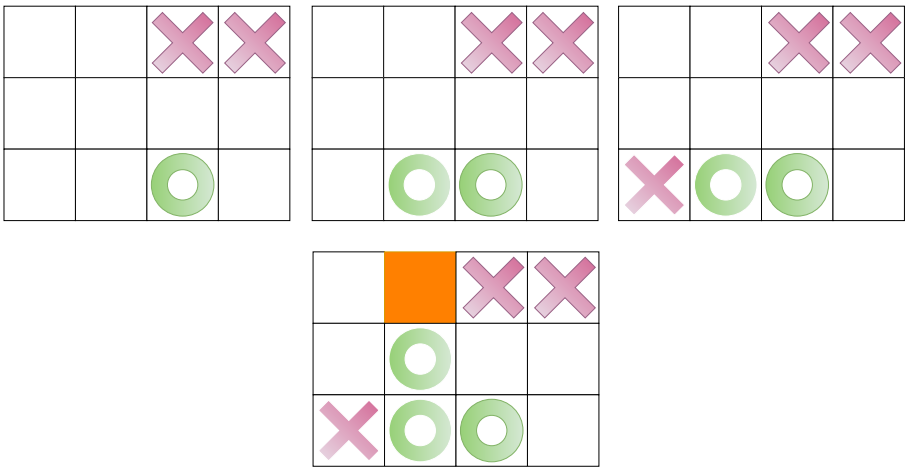

Figura 10: Suposição que leva a caso impossível

Isto quererá dizer que na casa (3,2) ou é uma cruz ou terá de ser bloqueada. Testando isso, verificamos que é possível ter uma cruz, uma vez que não nos leva a nenhuma impossibilidade.

Na Figura 11, mostra a sequência de decisões lógicas resultantes de fazer aquela suposição. No final, se fizermos a mesma suposição na casa (1,1) verificamos que tanto pode ser bola como cruz. Por isso, temos de colocar um `FIXO_X` ou um `FIXO_O`. Se colocarmos um `FIXO_O` seria mais rápido chegar a

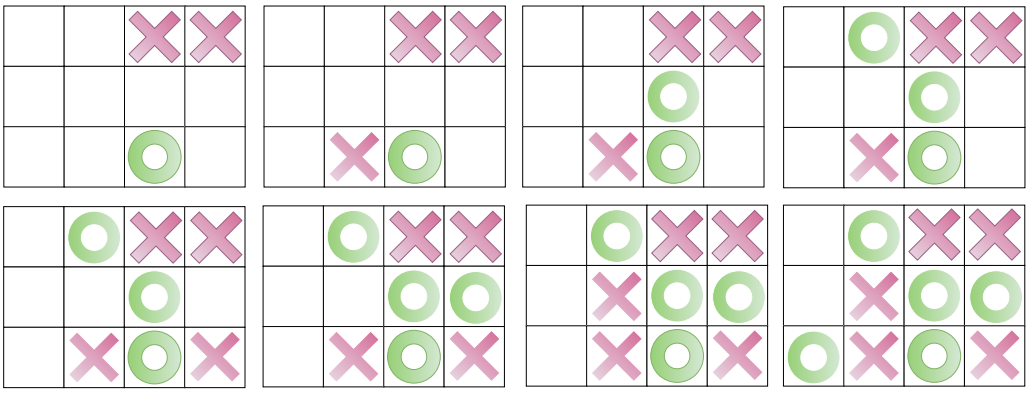

Figura 11: Suposição possível

uma conclusão. Mas a escolha poderia ser aleatória. A Figura 12, mostra o tabuleiro resultante de colocar `FIXO_O`.

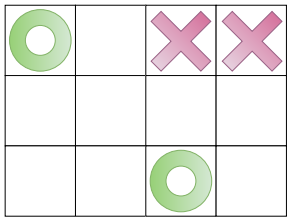

Figura 12: Resultado para um tabuleiro difícil

O seguinte bloco de código mostra como é feito este processo.

```
int solver_hard (PUZZLE *board)
{
  int i, j, found, foundX, foundO;
  found = foundX = foundO = 0;
  for (i = 0; (i < board->num_cols) && !found; i++)
    for (j = 0; (j < board->num_lins) && !found; j++)
      if (board->grelha[i][j] == VAZIA) {
        board->grelha[i][j] = SOL_O;
        found = 0; supor(board,&found);
        if (found == 2) foundX = 1;
        voltaAncora (board); board->grelha[i][j] = SOL_X;
        found = 0; supor(board,&found);
        if (found == 2) foundO = 1;
        if (foundX && foundO) {
```

```
            board->grelha[i][j] = BLOQUEADA; found=0;
          } else if (foundX) {
            board->grelha[i][j] = SOL_X; found = 1;
          } else if (foundO) {
            board->grelha[i][j] = SOL_O; found = 1;
          } else {
            board->grelha[i][j] = VAZIA; found = 0;
          }
        voltaAncora(board);
        }
    return found;
}
```

A função de suposição é uma função auxiliar da `solver_hard` e é definida no bloco de código seguinte.

```
void supor (PUZZLE *board,int *found)
{
  int x, y, tmp;
  marcaAncora(board);
  while ((*found )==0)
  {
    tmp = solver_easy (board, &x, &y);
    if (tmp == 1)
    {
      push(x, y, board->numAncs, &(board->undo));
      board->sizeU++;
    } else if (tmp == 2) {
      board->grelha[x][y] = VAZIA;
      *found = 2;
    } else if (tmp == 0) *found = 1;
  }
}
```

Em tabuleiros de dimensão pequena, os tabuleiros acabam por não se diferenciar muito em termos de dificuldade. No entanto, com maiores dimensões fica mais clara a distinção.

Conclusão

Para concluir, o algoritmo base do nosso trabalho para tabuleiros fáceis é: gerar algumas peças fixas e bloqueadas no tabuleiro, jogar o máximo número de jogadas óbvias possível, se o tabuleiro ainda não estiver completo gerar uma nova peça fixa e repetir o processo depois do 1º passo.

Já nos tabuleiros difíceis é: jogar o máximo número de jogadas óbvias possível, se o tabuleiro não estiver cheio jogar uma peça a partir de uma suposição, se nenhuma peça for jogada na suposição e o tabuleiro não estiver completo gerar uma nova peça fixa e voltar ao primeiro passo, senão voltar ao primeiro passo. No final de ambos os algoritmos as peças que não são fixas ou bloqueadas são retiradas.

Fazendo uma pequena análise relativa às competências adquiridas durante a realização deste trabalho conseguimos concluir que o mesmo foi bastante importante no que diz respeito à aquisição de conhecimentos sobre algoritmos, tendo sido a nossa primeira introdução séria a essa área.

Achamos também que alcançamos os objetivos desta fase do trabalho, ou seja, conseguimos gerar com sucesso tabuleiros resolúveis de dificuldades distintas, conforme a vontade do utilizador.

De uma maneira geral é nossa convicção que atingimos os objetivos propostos aquando da apresentação deste trabalho.