# Projeto LI2 - Battleships

Luís Gonçalo Epifânio Pereira
Matias Nicolau Araújo
José Pedro Veiga da Silva

31 de Maio de 2015

# Conteúdo

# 1 Introdução

Neste relatório será feita uma breve análise acerca de dois parâmetros:

**Comando R**
: A implementação do comando R, descrevendo o seu comportamento

**Código Assembly**
: Análise do código assembly gerado através da compilação do ficheiro "comp.c"disponibilizado no git, através do GCC -O2

# 2 Código

Nesta secção será analisado o código gerado nos dois pontos anteriormente referidos.

## 2.1 Comando R

Nesta etapa, foi implementado um comando R, acrónimo de Resolver, com o intuito de, através da receção de um tabuleiro, tentar a resolução do mesmo, através da repetição de execução das estratégias 1, 2 e 3 implementadas na etapa anterior.

```
void cmd_R(ESTRUTURA_BN *est_bn, stck *stack){
    int e1,e2,e3,max=0;
    stck s;
    union dados d;

    d.dadosest=newDadosEst(est_bn);
    push(stack,'R',d);

    do{
        e1=cmd_E1(est_bn,&s);
        e2=cmd_E2(est_bn,&s);
        e3=cmd_E3(est_bn,&s);
        ++max;
        if (max>=150)
            e1=0;
            e2=0;
            e3=0;
    }while (e1!=0 || e2!=0 || e3!=0);
}
```

Esta função receberia um tabuleiro sob forma de ESTRUTURA_bn, assim como uma posição da stack.

```
void cmd_R(ESTRUTURA_BN *est_bn, stck *stack)
```

De seguida, criar-se-iam três variáveis e1, e2 e e3, com o intuito de verificar qual o valor retornado pelas funções correspondentes às estratégias, ou seja, e1 corresponderia à estratégia 1, e2 corresponderia à estratégia 2 e e3 corresponderia à estratégia 3. A variável max é uma variável de controlo, para assegurar que o ciclo não é infinito.

```
int e1,e2,e3;
```

Tal como nas outras funções, cmd_R guardaria uma posição na stack, de modo a que, após execução do comando D, para desfazer, esta pudesse voltar atrás e devolver o tabuleiro anterior ao da última alteração.

```
stck s;
union dados d;

d.dadosest=newDadosEst(est_bn);
push(stack, 'R', d);
```

Por fim, a função cmd_R executaria as estratégias em ciclo, até que cada uma delas igualasse '0'. Este valor foi escolhido para retratar a existência de alterações no tabuleiro após execução da estratégia, isto é, em cada uma das estratégias foi implementado um mecanismo de controle de alterações, em que consoante existissem alterações após execução da estratégia, esta devolveria o valor 1; caso não existissem alterações, esta devolveria o valor 0. De forma a evitar ciclos infinitos, foi definida uma variável 'max', anteriormente referida, de modo a que, após os 150 ciclos sem sucesso de cada uma das estratégias, este pára a execução do comando e retorna '0', ou seja, o tabuleiro é irresolúvel.

```
    do{
        e1=cmd_E1(est_bn,&s);
        e2=cmd_E2(est_bn,&s);
        e3=cmd_E3(est_bn,&s);
        ++max;
        if (max>=150)
            e1=0;
            e2=0;
            e3=0;
    }while (e1!=0 || e2!=0 || e3!=0);
```

## 2.2 Código Assembly

Foi-nos também proposta a análise de uma porção de código gerado, após compilação de um ficheiro previamente fornecido.
Esta análise consiste, fundamentalmente, na explicação de instruções subsequentes. Para tal, são expostos pequenos fragmentos de código, seguidos de uma breve análise do mesmo.

```
Dump of assembler code for function contar_segs:
   0x0000000000400920 <+0>:   test   %dil,%dil                    #se forem iguais retorna 0 e atribui a uma flag
   0x0000000000400923 <+3>:   push   %rbx                         #vai buscar o operando em %rbx e decrementa o stack pointer
   0x0000000000400924 <+4>:   mov    0x2720(%rsp),%r10d           #copia o valor em %r10d para %rsp(0x2720)
   0x000000000040092c <+12>:  mov    0x2724(%rsp),%eax            #copia o valor em %eax para %rsp(0x2724)
   0x0000000000400933 <+19>:  je     0x400990 <contar_segs+112>   #salta caso igual para a instrução em 0x400990
   0x0000000000400935 <+21>:  sub    $0x1,%esi                    #subtrai o registo em %esi ao valor memória $0x1
   0x0000000000400938 <+24>:  xor    %ebx,%ebx                    #põe o conteúdo de %ebx a zero(ou exclusivo)
   0x000000000040093a <+26>:  mov    $0x1,%r11d                   #copia o valor em r11d para 0x1
   0x0000000000400940 <+32>:  xor    %edx,%edx                    #põe o conteúdo de edx a zero(ou exclusivo)
   0x0000000000400942 <+34>:  xor    %eax,%eax                    #põe o conteúdo de eax a zero(ou exclusivo)
   0x0000000000400944 <+36>:  test   %r10d,%r10d                  #se forem iguais retorna 0 e atribui a uma flag
   0x0000000000400947 <+39>:  jle    0x400989 <contar_segs+105>   #salta menor ou igual para a instrução 0x400989
   0x0000000000400949 <+41>:  xor    %ecx,%ecx                    #põe o conteúdo de %ecx a zero(ou exclusivo)
   0x000000000040094b <+43>:  nopl   0x0(%rax,%rax,1)             #no operation mas assume o tamanho da word e avança para a instrução
   0x0000000000400950 <+48>:  movslq %esi,%rdi                    #copia o valor em %rdi para %esi com sinal extendido
   0x0000000000400953 <+51>:  movslq %edx,%r9                     #copia o valor em %r9 para %edx com sinal extendido
   0x0000000000400956 <+54>:  lea    (%rdi,%rdi,4),%rdi           #transfere (%rdi,%rdi,4) para %rdi
   0x000000000040095a <+58>:  lea    (%rdi,%rdi,4),%r8            #transfere (%rdi,%rdi,4) para %r8
   0x000000000040095e <+62>:  lea    (%r9,%r8,4),%rdi             #transfere (%r9,%r8,4) o %rdi
   0x0000000000400962 <+66>:  movzbl 0x10(%rsp,%rdi,1),%edi       #extende o byte do sinal e copia para uma double-word
   0x0000000000400967 <+71>:  cmp    $0x2e,%dil                   #compara %dil com %0x2e
   0x000000000040096b <+75>:  je     0x40097c <contar_segs+92>    #salta para 0x40097c
   0x000000000040096d <+77>:  cmp    $0x7e,%dil                   #compara %dil com %0x7e
   0x0000000000400971 <+81>:  setne  %dil                         #define o byte no operando %dil para 1 se a flag 0 estiver livre
   0x0000000000400975 <+85>:  cmp    $0x1,%dil                    #compara %dil com %0x1
   0x0000000000400979 <+89>:  sbb    $0xffffffff,%eax             #subtrai 0xffffffff a eax e subtrai 1 se CF estiver definida
   0x000000000040097c <+92>:  add    $0x1,%ecx                    #adiciona %ecx a %0x1
   0x000000000040097f <+95>:  add    %r11d,%edx                   #adiciona %edx a %r11d
   0x0000000000400982 <+98>:  add    %ebx,%esi                    #adiciona %esi a %edx
   0x0000000000400984 <+100>: cmp    %r10d,%ecx                   #compara %ecx com %r10d
   0x0000000000400987 <+103>: jne    0x400950 <contar_segs+48>    #salta se não fôr igual
   0x0000000000400989 <+105>: pop    %rbx                         #coloca no topo da stack
   0x000000000040098a <+106>: retq                                #coloca o conteúdo do stack pointer para a base da stack
   0x000000000040098b <+107>: nopl   0x0(%rax,%rax,1)             #no operation mas assume o tamanho da word e avança para a instrução
   0x0000000000400990 <+112>: lea    -0x1(%rsi),%edx              #transfere (-0x1(%rsi) para %edx
   0x0000000000400993 <+115>: mov    %eax,%r10d                   #copia o valor em %r10d para %eax
   0x0000000000400996 <+118>: mov    $0x1,%ebx                    #copia o valor em %ebx para $0x1
   0x000000000040099b <+123>: xor    %r11d,%r11d                  #põe o conteúdo de %r11 a zero(ou exclusivo)
   0x000000000040099e <+126>: xor    %esi,%esi                    #põe o conteúdo de %esi a zero(ou exclusivo)
   0x00000000004009a0 <+128>: jmp    0x400942 <contar_segs+34>    #salta para 0x40092
```

# 3 Conclusão

Após realização desta etapa, conseguimos concluir que:

- O nosso cmd_R apenas recorre às estratégias implementadas na etapa2. Como tal, este mesmo comando não é complexo ao ponto de conseguir resolver todo o tipo de tabuleiros
- Não nos foi possível a implementação do cmd_G, cujo propósito seria o de gerar tabuleiros com diversas dificuldades