Relatório Fase II

André Camargo Perello - 9912403
Lucas Henrique Bahr Yau - 9763832
Rudá Lima da Floresta 9912410

- **Introdução**
  Nesta fase do projeto devemos, utilizando o código que interpreta código de cada robô fornecido pelo usuário, inserir novos comandos e criar estruturas para que a base do jogo esteja formada. Dentre elas estão os tipos de **terreno**, o **Grid Hexagonal**, uma matriz onde o jogo se passará, **chamadas de sistema**, onde o robô pede ao sistema para realizar uma ação, **criação e destruição de exércitos** de robôs e um atualizador do jogo, ou **game loop**.

- **Arena**
  A Arena é onde o jogo ocorre. Sua implementação envolve uma *struct* que contém o Grid Hexagonal, representando o tabuleiro, um vetor que armazena os exército assim como dois inteiros, um representando a última posição livre do vetor de exército e outro representando o tempo decorrido de jogo.

```
// Arena
typedef struct {
  Grid grid;
  int rows;
  int cols;
  int tempo;
  Maquina* exercitos[MAXMEM];
  int firstFree;
} Arena;
```

- **Célula**
  Uma Célula é uma struct da forma:

```
// Celula
typedef struct {
  Terreno t;
  Base b;
  Cristais c;
  Ocupacao o;
} Celula;
```

  Com esses dados, listamos o que cada célula do Grid necessita para que o jogo funcione bem. Assim como é dedutível, o *Terreno t* indica que tipo de terreno a célula possui. A *Base b* determina se a célula configura uma base e se é amiga ou inimiga. *Cristais c* é um inteiro que possui o número de cristais contidos na célula. Por último, a *Ocupação o* diz se a célula está ocupada e se quem a ocupa é amigo ou inimigo.

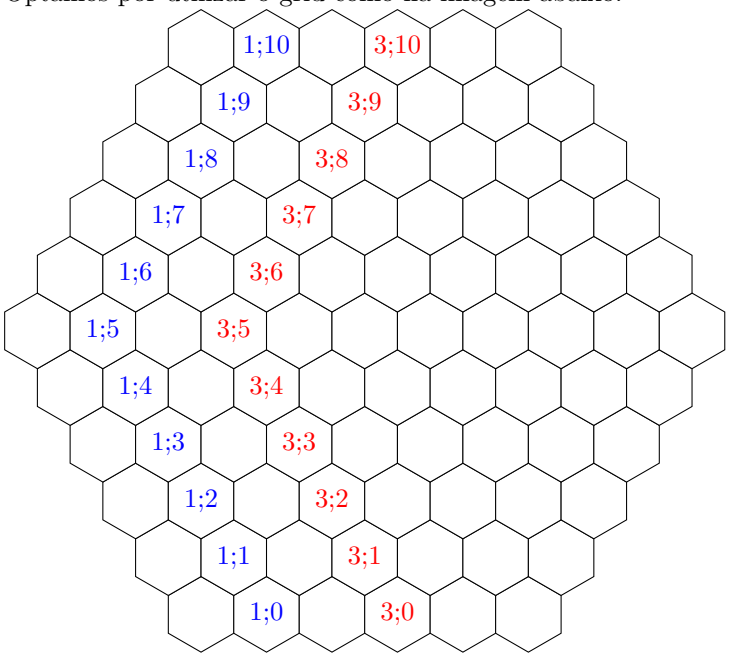
- **Grid Hexagonal**
  O Grid Hexagonal é apenas uma matriz de células onde o jogo será realizado. Ele está definido da forma:

```
typedef struct Celula** Grid;
```

Ele é inicializado da forma:

```
arena->grid = malloc(nrows * sizeof(Grid *));
for(int i = 0; i < nrows; i++) {
    arena->grid[i] = malloc(ncolumns * sizeof(Grid));
```

Além disso inicializamos o tipo do terreno e a quantidade de cristais aleatoriamente para toda célula do grid. Por simplicidade, as bases de cada time começam nos cantos superior esquerdo e inferior direito, sendo os únicos lugares onde há ocupação.
Optamos por utilizar o grid como na imagem abaixo:



Isso facilitou a implementaçao de movimentos e o acesso às celulas vizinhas.

- **Game Loop**
  Esta é funao responsável pela atualização do jogo do tempo t para o tempo $t + 1$:

```
void atualiza(Arena *arena, int ciclos);
```

Nada mais que um *loop* que percorre o vetor de robô da Arena e executa as instruções fornecidas ao robô um número *ciclos* de vezes, aumentando o tempo de jogo em 1 a cada atualização.

Dessa forma, o *game loop* é formado por uma série de chamadas desta função, utilizadas até que um dos times saia vencedor.

- **O tipo OPERANDO**
  Antes de falarmos sobre as chamadas de sistema e o método ATR, peças vitais para o funcionamento do programa nesta fase, devemos nos ater às escolhas que foram feitas para a implementação do tipo **OPERANDO**. A **struct OPERANDO** é da forma:

```
typedef struct {
    int n;
    Direction d;
} OPERANDO;
```

Esta escolha de implementação se baseou nos variados tipos de programas que um robô pode ter. Sejam calcular fatoriais, obter informações sobre uma célula do grid ou mover-se e atacar, as instruções precisam apenas de dois tipos de informações, um inteiro ou uma direção. Para ilustrar isso, considere o seguinte programa:

```
ACAO  MOVE WEST
ACAO  ATTK SWEST
INTER PUSH NEAST
NUM   ATR  0
```

Embora estes comandos não tenham sido explicados até o momento, é fácil ver que ele consiste em:

  - Mover-se na direção oeste
  - Atacar na direção sudeste
  - Inserir na pilha a célula na direção nordeste
  - Empilhar o primeiro atributo da célula inserida na pilha

  Como a pilha que executa o robô consiste em uma pilha de tipo OPERANDO, essa implementação facilitou uma série de problemas que encontramos, e poderíamos encontrar, devido a outras implementações que se baseavam em tornar o OPERANDO um tipo de célula do grid. Deste modo, lidamos apenas com inteiros e uma enumeração das direções possíveis do grid hexagonal.

- **System Calls**
  As chamadas de sistema são um conjunto de funções que, dados *ação* e *direção*, verifica se é uma ação válida e executa no robô que a pediu Elas retornam um OPERANDO que possui na sua parcela inteira um indicativo do sucesso ou fracasso da chamada. Este OPERANDO é empilhado para que o robô possa fazer verificações e estruturas mais complexas de decisão.

Dentre as ações possíveis estão: *mover, pegar, atacar, depositar*. Logo, o usuário faz uma chamada da forma:

```
ACAO <ACTION> <DIRECTION>
```

Temos as ações:

1. MOVE

   ```
   OPERANDO moveMachine(Arena *A, Maquina *m, Direction
       d);
   ```

   Esta função apenas "olha" se a posição correspondente a direção d está ocupada e atualiza a posição atual do robô caso contrário.

2. GRAB

   ```
   OPERANDO grabCrystal(Arena *A, Maquina *m, Direction
       d);
   ```

   Esta função apenas verifica se a posição correspondente a direção d possui cristais e diminui a quantidade de cristais da célula em uma unidade, aumentando a do robô na mesma proporção.

3. DEPO

   ```
   OPERANDO depositCrystal(Arena *A, Maquina *m,
       Direction d);
   ```

   Esta função retira cristal do robô e aumenta a da célula, amos em uma unidade.

4. ATTK

   ```
   OPERANDO attackMachine(Arena *A, Maquina *m, Direction
       d);
   ```

   Esta função apenas verifica se há um robô do time inimigo na posição correspondente e, por enquanto, se houver, imprime uma mensagem na saída padrão. Futuramente pretendemos implementar um sistema de pontos de vida nos robôs, além de pontos de ataque e defesa para que esta função se mostre útil.

- **O método ATR**
  Antes de chegarmos ao método ATR, vale a pena explicar como funciona para o usuário empilhar uma célula para que ele possa acessar um de seus atributos.
  Criamos um tipo novo de instrução que possui apenas uma chamada específica:

```
INTER PUSH <DIRECTION>
```

De nome **interação**, esta instrução empilha um OPERANDO com a direção especificada para que esta direção possa ser utilizada pelo método ATR através da função:

```
getPosition(Maquina *m, Direction d, int *i, int *j, int
    rows, int cols);
```

Esta função recebe a direção especificada e coloca nos ponteiros i e j a posição correspondente no grid. Assim como especificado, quando o usuário chama ATR, a seguinte parcela de código é executada:

```
case ATR:
 tmp = desempilha(pil);
 int i = -1, j = -1;
 getPosition(m, tmp.d, &i, &j, arena->rows, arena->cols);
 if(i != -1){
    switch(arg.n){
        case 0:
          tmp.n = (int)(arena->grid[i][j].t);
          empilha(pil, tmp);
          break;
       case 1:
          tmp.n = (int)(arena->grid[i][j].b.isBase);
          empilha(pil, tmp);
          break;
       case 2:
          tmp.n = (int)(arena->grid[i][j].c);
          empilha(pil, tmp);
          break;
       case 3:
          tmp.n = (int)(arena->grid[i][j].o.ocupado);
          empilha(pil, tmp);
          break;
    }
 }
 else
    printf("Celula invalida.\n");
```

Ou seja, pegamos o atributo de número arg.n, especificado pelo usuário, e empilhamos. Um detalhe vital para a simplificação da implementação consiste nos atributos *Base* e *Ocupacao*. Ambas as structs possuem duas informações, porém optamos por indicar ao usuário apenas se a célula correspondente é base ou se está ocupada.

- **Criação e destruição de exércitos**
  Essas funções são simples varreduras pelo vetor de robôs da arena. Para fins de simplificação, limitamos a quantidade de robôs da arena em 100, o que explica as constantes nas funções. No momento inserimos uma quantidade size de robôs de um determinado time e colocamos sua posição inicial na primeira posição válida do grid. Isto mudará nas fases futuras, pois estamos colocando todos os robôs, de times diferentes, na mesma localização.

```
void InsereExercito(Arena *arena, int size, INSTR *p, Time
    team) {

   if(size > 100-arena->firstFree) {
      printf("The Arena is full.\n");
      return;
   }

 for(int i = arena->firstFree; i < size + arena->firstFree;
     i++){
   Maquina *robo;
   robo = cria_maquina(p);

   robo->t = team;
   arena->exercitos[i] = robo;
   robo->crystals = 0;
   robo->alive = True;
   robo->x = 1;
   robo->y = 0;

 }
 arena->firstFree += size;

}
```

Decidimos por destruir um grupo de robôs utilizando o time que ele pertence.

```
void removeExercito(Arena *arena, Time t) {

  for(int i = 0; i < arena->firstFree; i++) {
     if(arena->exercitos[i] != NULL && arena->exercitos[i]->t
         == t) {
        arena->exercitos[i] = NULL;
     }
  }


  arena->firstFree = tapaBuraco(arena->exercitos,
      arena->firstFree);
```

```
}
```

A função *tapaBuraco* foi essencial para a solução de bugs e para a atualização da primeira posição livre do vetor de máquinas. Ela funciona similarmente à função *partition* do *quicksort* que, dado um pivô, separa todos os elementos do vetor em maiores que o pivô e menores que ele. Neste caso, separamos todos os ponteiros **NULL** pra máquina daqueles que não o são

- **O montador**
  O programa em python do montador simplesmente gera um arquivo em C a partir de um programa feito para um robô como no exemplo:

```
recursiveFactorial:

  NUM PUSH 9
  NUM CALL 4
  NUM PRN 0
  NUM END 0
  NUM ALC 1
  NUM DUP 0
  NUM STL 0
  NUM PUSH 1
  NUM EQ 0
  NUM JIF 13
  NUM PUSH 1
  NUM FRE 1
  NUM RET 0
  NUM RCE 0
  NUM PUSH 1
  NUM SUB 0
  NUM CALL 4
  NUM RCE 0
  NUM MUL 0
  NUM FRE 1
  NUM RET 0
```

A diferença principal do código da fase anterior está no vetor de instruções, prog, gerado pelo montador. Como adaptamos o tipo operando, o montador agora verifica de que tipo é o comando da linha especificada e acrescenta uma direção *default* ou um inteiro *default*:

```
INSTR prog[] = {


  {NUM, PUSH, { 9, WEST }},
  {NUM, CALL, { 4, WEST }},
  {NUM, PRN, { 0, WEST }},
```

```
    {NUM, END, { 0, WEST }},
    {NUM, ALC, { 1, WEST }},
    {NUM, DUP, { 0, WEST }},
    {NUM, STL, { 0, WEST }},
    {NUM, PUSH, { 1, WEST }},
    {NUM, EQ, { 0, WEST }},
    {NUM, JIF, { 13, WEST }},
    {NUM, PUSH, { 1, WEST }},
    {NUM, FRE, { 1, WEST }},
    {NUM, RET, { 0, WEST }},
    {NUM, RCE, { 0, WEST }},
    {NUM, PUSH, { 1, WEST }},
    {NUM, SUB, { 0, WEST }},
    {NUM, CALL, { 4, WEST }},
    {NUM, RCE, { 0, WEST }},
    {NUM, MUL, { 0, WEST }},
    {NUM, FRE, { 1, WEST }},
    {NUM, RET, { 0, WEST }},

};
```

- **Dificuldades**
  A maior dificuldade nesta fase foi com o tratamento de variáveis de diferentes tipos. Como a especificação não era muito clara, focamos em entregar um produto que possui as funcionalidades pedidas. Em especial, lidar com a struct OPERANDO foi um jogo de acertos e erros até encontrar a implementação que mais fizesse sentido e que gerasse o menor número de bugs.

- **Makefile e Testes**
  Este é o código do nosso **Makefile**:

```
    # Variveis
        ################################################
CC = gcc
CFLAGS = -std=c99
EXECS = testAttackMachine testAtualiza testDepositCrystal
    testGetPosition testGrabCrystal testHasCrystal
    testInicializaGrid testInsereExercito
    testRemoveExercito testRemoveMortos testSysCall
    testTapaBuraco testMove
OBJ = testAttackMachine.o testAtualiza.o
    testDepositCrystal.o testGetPosition.o
    testGrabCrystal.o testHasCrystal.o testInicializaGrid.o
    testInsereExercito.o testRemoveExercito.o
    testRemoveMortos.o testSysCall.o testTapaBuraco.o
    testMove.o
HEADERS = maq.h arena.h structures.h instr.h pilha.h
DEPENDS = maq.c arena.c pilha.c
```

```
# Cria os arquivos-objeto de todos os testes
    ################
%.o: %.c $(CFLAGS) $(HEADERS)
 $(CC) -c $(CFLAGS) -o $@ $<



# Compila todos os arquivos-objetos
    ########################
all: $(OBJ)
 gcc $(CFLAGS) -o testAttackMachine testAttackMachine.o
     $(DEPENDS)
 gcc $(CFLAGS) -o testAtualiza testAtualiza.o $(DEPENDS)
 gcc $(CFLAGS) -o testDepositCrystal testDepositCrystal.o
     $(DEPENDS)
 gcc $(CFLAGS) -o testGetPosition testGetPosition.o
     $(DEPENDS)
 gcc $(CFLAGS) -o testGrabCrystal testGrabCrystal.o
     $(DEPENDS)
 gcc $(CFLAGS) -o testHasCrystal testHasCrystal.o $(DEPENDS)
 gcc $(CFLAGS) -o testInicializaGrid testInicializaGrid.o
     $(DEPENDS)
 gcc $(CFLAGS) -o testInsereExercito testInsereExercito.o
     $(DEPENDS)
 gcc $(CFLAGS) -o testMove testMove.o $(DEPENDS)
 gcc $(CFLAGS) -o testRemoveExercito testRemoveExercito.o
     $(DEPENDS)
 gcc $(CFLAGS) -o testRemoveMortos testRemoveMortos.o
     $(DEPENDS)
 gcc $(CFLAGS) -o testSysCall testSysCall.o $(DEPENDS)
 gcc $(CFLAGS) -o testTapaBuraco testTapaBuraco.o $(DEPENDS)



# CLEAN
    ######################################################
clean:
 rm -f $(EXECS) *.o
```

Este file ajuda o monitor a compilar todos os testes com apenas um comando. Dessa forma, pode apenas testar cada um deles. Cada teste possui um modo de testar suas saídas que, em sua maioria, estão especificadas no file de teste. Caso o monitor queira gerar um programa para testar nosso código, basta gerar o código com o auxílio do programa *montador.py* que gera o código em C a ser executado.

- **Conclusões**
  Embora hajam funcionalidades que poderiam ser melhor lapidadas, como a introdução de um exército de um determinado time na arena, optamos por simplificar a implementação para poder lidar melhor com detalhes nas fases seguintes. Assim, todas as funcionalidades pedidas operam de acordo e podem ser reutilizadas e ampliadas a medida que o projeto se desenvolve.
  Em suma, temos um esqueleto bem estruturado, porém feito de ossos, não de diamantes.