

# *Swack*

## *Relatório Final*

### Programação em Lógica

Victor Saldanha Nunes

**up201907226@fe.up.pt**

Artur José Gonçalves Zeferino

**up201504823@fe.up.pt**

# *Índice*

Instalação e Execução do Jogo	2
As Regras do Jogo	2
Representação e Visualização do Estado do Jogo	5
Lista de Jogadas Válidas	7
Execução de Jogadas	8
Final do Jogo	8
Avaliação do Tabuleiro	9
Jogada do Computador	9
O Flow do Jogo	10
Conclusão	10
Bibliografia	10

# *Instalação e Execução do Jogo*

Swack é um jogo muito simples e não requer nada adicional instalado para além do SICStus Prolog. Basta baixar os ficheiros .pl do jogo, abrir o SICStus, clicar em “File”, depois clique em “Consult”, e selecione o ficheiro chamado “main.pl”. Finalmente basta escrever “play.” e o jogo irá começar.

## *As Regras do Jogo*

Swack é um jogo que se é jogado em um tabuleiro quadrado, e seu tamanho pode ser escolhido pelos jogadores. Se se tratar de um tabuleiro de dimensão ímpar, o quadrado central terá de ser ocupado pelas peças brancas.

### **Definições**

Antes de começarmos a tratar das regras propriamente ditas, vale a pena definir certos conceitos previamente:

- Pilha: Uma pilha é um amontoado de peças sobrepostas. O tamanho da pilha é a quantidade de peças que ela possui e o dono da pilha é a cor da peça que está ao topo.

- Grupo: É um conjunto de pilhas que são ortogonalmente adjacentes entre si. O tamanho de um grupo é igual à quantidade de pilhas que o constituem.

### **Posição Inicial**

As peças das pretas e das brancas são dispostas no tabuleiro de forma alternada. Propriamente as peças correspondendo às cores dos seus respectivos quadrados.

	1	2	3	4	5	6
1	○	●	○	●	○	●
2	●	○	●	○	●	○
3	○	●	○	●	○	●
4	●	○	●	○	●	○
5	○	●	○	●	○	●
6	●	○	●	○	●	○

## Possíveis Jogadas num Turno

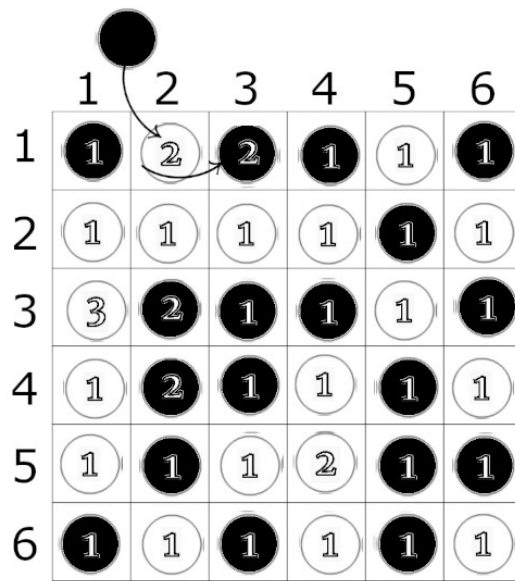
No turno de um jogador, ele tem as opções de ou passar a vez ou realizar um movimento. As pretas sempre começam o jogo.

Para realizar um movimento, certas condições devem ser atendidas:

- O movimento consiste em sua peça, estando no topo de uma pilha, capturar uma pilha inimiga ortogonalmente adjacente a ela. Sua peça então ficará acima da pilha inimiga, tornando-a sua, mas em sua pilha de origem é colocada uma peça do rival (peça esta que é trazida de fora do tabuleiro), tornando-a dele.
- Para fazer a captura, é necessário que ambas a sua pilha e a pilha do inimigo sejam do mesmo tamanho antes da jogada.

	1	2	3	4	5	6
1	1	2	2	1	1	1
2	1	1	1	1	1	1
3	3	2	1	1	1	1
4	1	2	1	1	1	1
5	1	1	1	2	1	1
6	1	1	1	1	1	1

É a vez das brancas. A peça branca da pilha 1x2 está planejando capturar a pilha das pretas de 1x3.



	1	2	3	4	5	6
1	1	2	2	1	1	1
2	1	1	1	1	1	1
3	3	2	1	1	1	1
4	1	2	1	1	1	1
5	1	1	1	2	1	1
6	1	1	1	1	1	1

A peça branca de 1x2 está capturando a pilha das pretas de 1x3. Depois disso, uma nova peça preta será posta na antiga pilha das brancas.

	1	2	3	4	5	6
1	1	2	3	1	1	1
2	1	1	1	1	1	1
3	3	2	1	1	1	1
4	1	2	1	1	1	1
5	1	1	1	2	1	1
6	1	1	1	1	1	1

Situação após a captura

## Fim

O jogo acaba quando ambos os jogadores passam seu turno em sequencia. E assim, vence o jogador que possuir o maior grupo. Em caso de empate, olha-se para o segundo maior grupo, e se persistir o empate, olha-se para o terceiro e assim sucessivamente. Se o empate persistir no final de tudo, ganha quem realizou a última jogada.

Para mais informações, basta visitar o site: <https://boardgamegeek.com/boardgame/314462/swack>

# *Representação e Visualização do Estado do Jogo*

A representação do tabuleiro em nosso jogo será feita usando uma lista de listas, sendo que cada lista-célula do tabuleiro terá dois valores. Esses valores são respectivamente: o tamanho da pilha e o dono da pilha.

Nossa função para imprimir o estado do jogo receberá como argumentos o tabuleiro, tal qual o descrevemos anteriormente, e quem é o jogador da vez.

`display_game(+Board, +Player).`

Já nossa função para gerar um tabuleiro inicial apenas “retornará” o tabuleiro, perguntando-nos o tamanho que desejamos.

`initial(-Board).`

Vejamos alguns exemplos de situações durante o jogo, no caso em um tabuleiro 6x6:

## **Situação Inicial:**

```
display_game([[[[1, white], [1, black], [1, white], [1, black], [1, white], [1, black]],
               [[1, black], [1, white], [1, black], [1, white], [1, black], [1, white]],
               [[1, white], [1, black], [1, white], [1, black], [1, white], [1, black]],
               [[1, black], [1, white], [1, black], [1, white], [1, black], [1, white]],
               [[1, white], [1, black], [1, white], [1, black], [1, white], [1, black]],
               [[1, black], [1, white], [1, black], [1, white], [1, black], [1, white]]], black).
```

## **Situação Intermédia:**

```
display_game([[[[1, black], [2, white], [2, black], [1, black], [1, white], [1, black]],
               [[1, white], [1, white], [1, white], [1, white], [1, black], [1, white]],
               [[3, white], [2, black], [1, black], [1, black], [1, white], [1, black]],
               [[1, white], [2, black], [1, black], [1, white], [1, black], [1, white]],
               [[1, white], [1, black], [1, white], [2, white], [1, black], [1, black]],
               [[1, black], [1, white], [1, black], [1, white], [1, black], [1, white]]], white).
```

**Situação Final:**

```
display_game([[[[2, white], [3 ,white], [4, black], [2, black], [5, white], [2 ,black]],  
               [[3, black], [4, white], [3, white], [2, black], [3, white], [2, black]],  
               [[2, white], [6 ,white], [4, black], [3, black], [4, white], [5, black]],  
               [[3, black], [4, white], [3, white], [5, black], [3, black], [6, white]],  
               [[1, black], [2 ,white], [4, white], [7, black], [4, white], [3 ,black]],  
               [[4, white], [2, white], [3, black], [3, black], [2, white], [4, black]]], black).
```

	1	2	3	4	5	6
1	1W	1B	1W	1B	1W	1B
2	1B	1W	1B	1W	1B	1W
3	1W	1B	1W	1B	1W	1B
4	1B	1W	1B	1W	1B	1W
5	1W	1B	1W	1B	1W	1B
6	1B	1W	1B	1W	1B	1W

**Fig. 1** Situação Inicial vista na consola

	1	2	3	4	5	6
1	1B	2W	2B	1B	1W	1B
2	1W	1W	1W	1W	1B	1W
3	3W	2B	1B	1B	1W	1B
4	1W	2B	1B	1W	1B	1W
5	1W	1B	1W	2W	1B	1B
6	1B	1W	1B	1W	1B	1W

**Fig. 2** Situação Intermédia vista na consola

	1	2	3	4	5	6
1	2W	3W	4B	2B	5W	2B
2	3B	4W	3W	2B	3W	2B
3	2W	6W	4B	3B	4W	5B
4	3B	4W	3W	5B	3B	6W
5	1B	2W	4W	7B	4W	3B
6	4W	2W	3B	3B	2W	4B

**Fig. 3** Situação Final vista na consola

(As brancas venceram, pois fizeram 12 pontos,  
enquanto as pretas fizeram 10 pontos)

## *Lista de Jogadas Válidas*

Este predicado é usado para se obter uma lista de todas as jogadas válidas que um jogador pode fazer:

`validMoves(+Board, +Player, -ListOfMoves)`

Ele é utilizado pelos bots, sendo que a lista de movimentos retornada já vem ordenada de melhor para o pior movimento. Então o bot com maior inteligência sempre escolhe o primeiro movimento da lista, enquanto o bot com menor inteligência escolhe um movimento aleatório.

Os possíveis movimentos da lista tem a seguinte estrutura:

`Value-InitialRow/InitialColumn-DestRow/DestColumn`

Onde o par **InitialRow/InitialColumn** representa as coordenadas da peça a ser movida, e o par **DestRow/DestColumn** representa as coordenadas da pilha inimiga que será capturada. Já **Value** representa a diferença entre a pontuação do jogador e a pontuação de seu adversário, e este é o parâmetro para avaliar o quão bom é uma jogada. Quanto maior o **Value**, melhor é a jogada.

Também temos uma jogada especial que é o passar a vez, representado pela estrutura `DiffPoints-999/999-pass`. O **Value** do passar a vez nada mais é do que a diferença de pontos atual entre os jogadores. Ele tem essa forma para que, segundo o algoritmo de ordenação da lista de movimentos, ele seja a primeira opção da sua categoria de **Value**.

Isto é interessante pois sempre que a opção de passar a vez estiver igualmente empatada com outras possíveis movimentações como as jogadas com maior **Value** possível, o bot inteligente escolherá o passar a vez. É melhor para nós que o bot prefira passar a vez do que fazer uma jogada inútil pois pode encurtar a duração do jogo, se o oponente igualmente passar a vez e terminar o jogo.



## *Execução de Jogadas*

Para a execução de jogadas temos o seguinte predicado:

```
move(+Board-Player, +InitialRow/InitialColumn-DestRow/DestColumn,  
-NewBoard) :-  
findStack(+Board, +InitialRow, +InitialColumn, _, -Height),  
capture(+Board, +Player, +InitialRow, +InitialColumn, +Height, +DestRow,  
+DestColumn, -NewBoard).
```

Ele é utilizado para a movimentação feita tanto pelos jogadores humanos quanto os bots, depois de já terem decidido a jogada, recebendo o tabuleiro e o jogador da vez, e também as coordenadas da peça a ser mover e da pilha que ela irá capturar, retornando finalmente o novo tabuleiro.

Sua lógica é simples, o primeiro predicado serve para determinarmos o valor da altura da pilha, já o segundo realiza a captura e retorna o novo tabuleiro.

## *Final do Jogo*

O predicado de `game_over` recebe o tabuleiro do final do jogo, junto com o último jogador a realizar uma jogada, e retorna o vencedor e seus pontos. O predicado é o seguinte:

```
game_over( +Board-+LastPlayerToPlay, -Winner/-Points) :-  
countPoints(+Board, -ListWhitePoints, -ListBlackPoints),  
winner(+ListWhitePoints, +ListBlackPoints, +LastPlayerToPlay, -Winner,  
-Points).
```

O predicado `countPoints` recebe um tabuleiro e retorna duas listas ordenadas de maneira decrescente com os tamanhos dos grupos (correspondentes aos pontos propriamente ditos) das peças brancas e a outra das peças pretas. E o predicado `winner` recebe essas listas, junto com o último jogador que fez a jogada (para desempatar no caso de um empate total) e retorna o vencedor.

## *Avaliação do Tabuleiro*

O predicado para avaliar um tabuleiro é o seguinte:

```
value(+Board, +Player, -Value) :-
```

```
    countPoints(+Board, -ListWhitePoints, -ListBlackPoints),  
    diffPoints(+Player, +ListWhitePoints, +ListBlackPoints, -Value).
```

Recebendo um tabuleiro e o jogador, ela retornará o valor de tal situação atual do tabuleiro na perspectiva do jogador em questão. Primeiramente fazemos uma contagem de pontos do tabuleiro, criando as listas (ordenadas de maneira decrescente) dos tamanhos dos grupos do jogador branco e do preto. Em seguida, calculamos o valor atrás de uma diferença entre o tamanho do maior grupo do jogador e o tamanho do maior grupo do seu oponente.

Quanto maior o Value, melhor, pois significa que há uma maior diferença entre a pontuação do jogador em relação ao seu rival. Já ter uma Value negativo é sinal de que o oponente tem mais pontos do que o jogador que estamos analisando.

## *Jogada do Computador*

O predicado `choose_move` recebe um tabuleiro, o jogador em questão e o nível do bot e retorna a jogada escolhida pelo bot:

```
choose_move(+Board, +Player, +Level, -Move)
```

Baseando-se no Level do bot, no nível mais inteligente o bot escolhe a primeira jogada da lista de `valid_moves`, pois esta lista já está ordenada da melhor para a pior jogada possível. Já no nível menos inteligente, o bot simplesmente escolheria uma jogada aleatória desta lista.

## *O Flow do Jogo*

Todo o flow principal do jogo é baseado nas chamadas das funções `turn(+Board, +Player, +NumPasses)` indicando que é o vez de tal jogador ou `turnBot(+Board, +Player, +NumPasses)` se se tratar de um bot.

Após o turno se encerrar, é chamada a função `endTurn(+Board, +Player, +NumPasses, +Mode)` que com base no `gameMode` determinado no começo do jogo (1: Humano x Humano, 2: Humano x Computador, 3: Computador x Humano, 4: Computador x Computador) chamará o turno do próximo jogador, podendo ser um bot ou um humano dependendo da situação.

Este fluxo só acaba no momento em que `NumPasses`, que é a variável que armazena o número de vezes seguidas em que foi passado a vez (por isso que após toda movimentação ele volta para zero, e após todo 'pass' ele é incrementado uma unidade), for igual a dois.

Sendo `NumPasses` igual a dois, é sinal de que ambos os jogadores passaram a vez em sequencia, encerrando assim o jogo. Ocorre então a contagem de pontos e é determinado o vencedor.

## *Conclusão*

A conclusão do trabalho é bem satisfatória, pelo visto todos os mecanismos estão a funcionar bem e como o esperado. O jogo está preparado para lidar com respostas inválidas e dar a oportunidade do jogador corrigi-las para continuar o jogo, sendo o único jeito de burlar o mecanismo é inserir múltiplas respostas separadas por espaço ao predicado **read**, que o faz levantar um erro.

Uma possível melhoria para o trabalho seria uma interface visual mais interessante para o jogo, tentando melhorar o máximo possível apesar da limitação de estar numa tela de terminal.

## *Bibliografia*

Consulta geral: <https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/>  
Implementação do insertion sort: <http://kti.ms.mff.cuni.cz/~bartak/prolog/sorting.html>  
Random permutation:  
[https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/lib\\_002drandom.html#lib\\_002drandom](https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/lib_002drandom.html#lib_002drandom)