

Programação em Lógica

### PLOG – 3MIEIC321



TP1 – Relatório Intercalar – Grupo Cannon 4

João Álvaro Cardoso Soares Ferreira – [up201605592](mailto:up201605592@fe.up.pt)

[João Augusto dos Santos Lima](https://sigarra.up.pt/feup/pt/fest_geral.cursos_list?pv_num_unico=201605314) – [up201605314](mailto:up201605314@fe.up.pt)

Índice

[Introdução 3](#_Toc530240156)

[O Jogo 4](#_Toc530240157)

[Introdução ao Jogo 4](#_Toc530240158)

[Como Jogar 4](#_Toc530240159)

[Lógica do Jogo 6](#_Toc530240160)

[Representação do Estado do Jogo: 6](#_Toc530240161)

[Visualização do Tabuleiro 7](#_Toc530240162)

[Lista de Jogadas Válidas 8](#_Toc530240163)

[Execução de Jogadas 8](#_Toc530240164)

[Final do Jogo 8](#_Toc530240165)

[Avaliação do Tabuleiro 8](#_Toc530240166)

[Jogada do Computador 8](#_Toc530240167)

[Conclusões 8](#_Toc530240168)

[Bibliografia 8](#_Toc530240169)

# Introdução

Este projeto foi desenvolvido no âmbito da unidade curricular de PLOG e tem como requerimento a implementação de um jogo de tabuleiro em Prolog.

O objetivo do trabalho é, portanto, aumentar o nosso conhecimento e experiência em programação declarativa, desenvolvendo as capacidades para raciocínio abstrato e de resolução de problemas através de representações declarativas. Isto implica utilizar ferramentas a que não tenhamos tido tanto contacto ao aprender linguagens com focos diferentes (por exemplo programação orientada a objetos), como o uso constante de recursividade, backtracking e aproveitamento da ordem das regras e goals, levando a uma maior eficiência em problemas de decisão.

O jogo escolhido pelo grupo, Cannon, é semelhante a Damas e a nossa implementação pode ser jogada por dois jogadores, por um jogador contra a inteligência artificial ou por duas inteligências artificias uma contra a outra, de modo a demonstrar as capacidades do programa.



# O Jogo

## Introdução ao Jogo

Cannon é um jogo de tabuleiro desenvolvido por David E. Whitcher e publicado pela NestorGames. É um jogo abstrato de guerra por turnos para dois jogadores onde cada jogador jogador tenta capturar a cidade-base do adversário ou destruí-la com um canhão. Um canhão é um arranjo específico de 3 peças onde, estando alinhadas, é-lhes permitido habilidades especiais.

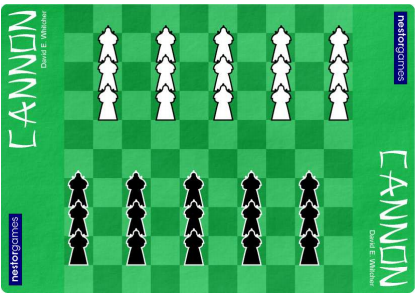
### Componentes

* Um tabuleiro de 10x10
* 2x15 peças básicas (soldados) – pretas e brancas
* 2 peças de cidade

## Como Jogar

### Montagem

Inicialmente, atribuir um jogador a cada lado (superior e inferior) e montar as peças da seguinte maneira:



Após isso, cada jogador coloca uma peça de cidade na linha horizontal mais próxima de si, por trás das suas peças “soldado”.

### Objetivos do Jogo

As cidades são o objetivo central do jogo – cada jogador tenta capturar ou destruir a cidade do adversário. Um sub-objetivo é capturar/destruir as peças do adversário.

### Movimento das Peças

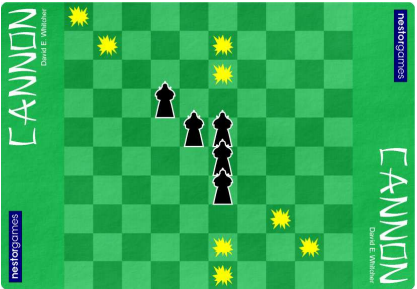
Todos os turnos é obrigatório o respetivo jogador mover uma peça. As peças “cidade” não têm qualquer tipo de movimento. As peças soldados podem mover-se individualmente ou, se estiverem alinhadas num canhão, mover-se em conjunto com as restantes componentes do respetivo canhão.

No movimento individual, podem:

* Mover-se para uma quadrícula adjacente frontal (em frente ou na diagonal) que esteja vazia.
* Capturar um soldado inimigo que lhes esteja adjacente, tanto à frente como nos lados (tomando o seu lugar ao fazê-lo).
* Fugir para a retaguarda duas quadrículas (para trás diretamente ou na diagonal) se estiverem adjacentes a uma peça inimiga.

Quanto formando um canhão (uma linha ortogonal ou diagonal de 3 soldados) podem também:

* Mover ao longo do canhão sem capturar nenhuma peça adjacente.
* Capturar peças a dois ou três quadrículas de distância que estejam na direção do canhão (exemplificado em baixo) desde que não hajam peças diretamente em frente ao canhão.



Como já foi mencionado, o jogo termina aquando da captura de uma das cidades.

# Lógica do Jogo

## Representação do Estado do Jogo:

32 – Espaço Vazio; 49 – Soldado Branco, 50 – Soldado Preto; 51 – Cidade Branca; 52 – Cidade Preta

### Estado Inicial (Exemplo):

tabuleiroInicial([[32,32,32,32,32,32,32,32,32,32],

[32,49,32,49,32,49,32,49,32,49],

[32,49,32,49,32,49,32,49,32,49],

[32,49,32,49,32,49,32,49,32,49],

[32,32,32,32,32,32,32,32,32,32],

[32,32,32,32,32,32,32,32,32,32],

[50,32,50,32,50,32,50,32,50,32],

[50,32,50,32,50,32,50,32,50,32],

[50,32,50,32,50,32,50,32,50,32],

[32,32,32,32,32,32,32,32,32,32]])

### Estado Intermédio (Exemplo):

tabuleiroMédio([[32,32,32,51,32,32,32,32,32,32],

[32,32,32,49,32,32,32,32,32,32],

[32,49,32,32,32,49,32,32,32,32],

[32,49,32,50,50,50,49,32,32,32],

[32,49,50,32,32,32,32,49,50,32],

[32,32,32,50,49,32,32,32,32,32],

[32,50,32,32,50,49,32,32,32,32],

[32,32,32,32,32,32,50,32,32,32],

[32,32,32,32,32,32,50,32,50,32],

[32,32,32,32,32,32,32,32,32,52]]).

### Estado Final (Exemplo):

tabuleiroFinal([[32,32,32,51,32,32,32,32,32,32],

[32,32,32,32,32,32,32,32,32,32],

[32,50,32,32,32,49,32,32,32,32],

[32,32,32,50,32,32,32,32,32,32],

[32,32,32,32,32,32,32,49,50,32],

[32,32,32,32,32,32,32,32,32,32],

[32,50,32,32,32,49,49,32,32,32],

[32,32,32,32,49,32,50,49,32,32],

[32,32,32,32,32,32,50,32,32,32],

[32,32,32,32,32,32,32,32,32,52]]).

Visualização do Tabuleiro

### Estado Inicial (Exemplo):

Uma imagem com palavras cruzadas, texto

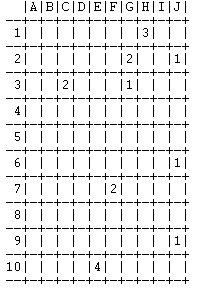
Descrição gerada automaticamente

### Estado Intermédio (Exemplo):

Uma imagem com texto

Descrição gerada automaticamente

### Estado Final (Exemplo):



Este é o código utilizado para efetuar a visualização do tabuleiro:

writeNumber(Number):-

((Number >= 10) ->

format('~d',Number);

format(' ~d',Number)).

imprimirLine([]):-

write('|'),

write('\n'),

write('--+-+-+-+-+-+-+-+-+-+-+\n').

imprimirLine([Head|Tail]):-

write('|'),

format('~c',Head),

imprimirLine(Tail).

imprimir([],\_).

imprimir([Head|Tabuleiro],Linha) :-

writeNumber(Linha),

imprimirLine(Head),

NewNumber is Linha + 1,

imprimir(Tabuleiro,NewNumber).

imprimirTabuleiro(Tabuleiro) :-

write(' |A|B|C|D|E|F|G|H|I|J|\n'),

write('--+-+-+-+-+-+-+-+-+-+-+\n'),

imprimir(Tabuleiro,1).

## Lista de Jogadas Válidas

Uma peça pode capturar, avançar ou recuar, podendo também um canhão ser disparado. Durante o decorrer do jogo, para mover uma peça primeiro selecionamos a sua posição. Quando isto ocorre, o jogo apresenta uma versão do tabuleiro com todas as jogadas válidas a partir dessa peça (neste caso, a peça selecionada foi a que ocupa a cela C9):

Uma imagem com texto, palavras cruzadas

Descrição gerada automaticamente

Para tal, verifica-se as posições disponíveis para movimento individual da peça (diagonal ou ortogonal na direção do adversário) estão vazias, verifica-se se a peça faz parte de um canhão e verifica-se se o movimento deste é possível (não estando uma peça diretamente à frente ou atrás deste).

Para captura verificam-se as posições adjacentes à exceção de diagonais traseiras e a célula diretamente atrás estão ocupadas com uma peça inimiga, enquanto que para captura do canhão verifica-se se há peças inimigas capturáveis a 2/3 celulas da frente ou retaguarda do canhão, verificando também se a célula diretamente em frente ao canhão está ocupada (o que impossibilitaria o disparo.

**Os predicados utilizados são:**

getCanonDisparos(Player,EstouLinha,EstouColuna,Tabuleiro,NovoMoves) – retorna a lista NovoMoves com os disparos possíveis de um canhão de que a peça indicada faça parte, verificando um predicado individual para cada direção.

getMovesPecaPlayer(1,EstouLinha,EstouColuna,Tabuleiro,NovoMoves) - retorna a lista NovoMoves com os movimentos possíveis, verificando um predicado individual para cada direção (inclui movimentos caso esteja num canhão).

getMovesPecaPlayer(2,EstouLinha,EstouColuna,Tabuleiro,NovoMoves) - retorna a lista NovoMoves com os movimentos possíveis, verificando um predicado individual para cada direção (inclui movimentos caso esteja num canhão).

getCaptures(1,EstouLinha,EstouColuna,Tabuleiro,NovoMoves) - retorna a lista NovoMoves com as capturas possíveis da peça individual, verificando um predicado individual para cada direção.

getCaptures(2,EstouLinha,EstouColuna,Tabuleiro,NovoMoves) - retorna a lista NovoMoves com as capturas possíveis da peça individual, verificando um predicado individual para cada direção.

## Execução de Jogadas

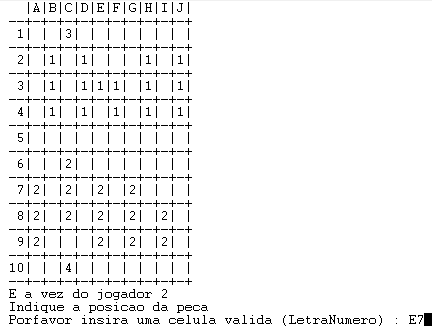
O jogo é iniciado com um menu onde são determinados os parâmetros para o início do jogo – o modo de jogo, as peças do jogador caso seja jogador vs PC (podendo escolher as brancas ou as negras) e o nível de capacidade do PC caso este seja um dos jogadores. É chamado startGame, onde os jogadores escolhem a posição da sua cidade respetiva, começando após isso o ciclo de jogo.

Este ciclo é denominado de play e chama-se recursivamente. Neste o primeiro jogador faz a sua jogada e depois é verificado se o jogo chegou a um estado final. Se tivermos um vencedor, o jogo acaba. No fim de cada jogada, o jogador atual é trocado pelo seu adversário, pelo que quando play é chamado novamente a jogada a fazer será a do adversário.

Durante a jogada, é pedido ao jogador que indique a peça que quer utilizar, sendo verificado se esta peça é válida. Se for, é retornado um tabuleiro alternativo com os movimentos e capturas possíveis com a peça ou um canhão onde esta esteja inserida. Quando é uma jogada por parte do computador, este processo ocorre de igual forma mas são inseridos predicados que fazem análise das jogadas possíveis e oferecem uma resposta. Esta resposta é de maior ou menor qualidade dependendo o nível de dificuldade e, portanto, o nível de profundidade da pesquisa. Este processo é explicado mais a fundo na secção “Avaliação do Tabuleiro”.

Esta é a interface durante a execução das jogadas:

**Turno do Jogador:**



**Turno do Computador:**

Uma imagem com palavras cruzadas, texto

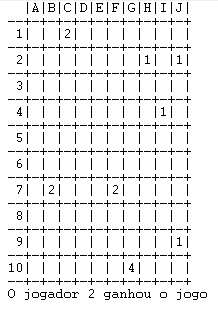
Descrição gerada automaticamente

Os principais predicados responsáveis pela execução são play, playTurn, play\_human, play\_robot, lerUserCelula, validar\_inicial, move e make\_move.

## Final do Jogo

Como já foi mencionado, a verificação do fim do jogo é feita uma vez por ciclo de jogo no predicado play. Isto é efetuado com uma chamada ao método game\_over que, através de análise do tabuleiro, verifica se alguma das duas cidades foi capturada ou se um dos jogadores ficou sem peças no tabuleiro, ambas condições de vitória para o adversário. Não introduzimos nenhum sistema para obter um empate já que não existem regras para “stalemate” no Cannon (ou outra forma de chegar a um empate), apesar de estas existirem em jogos similares como as Damas ou o Xadrez.

A interface no caso de fim do jogo (neste caso, o jogador 2 capturou a cidade do jogador 1, que estava na célula C1):



O predicado game\_over e o sua auxiliar, game\_over2:

game\_over2([Primeira,\_,\_,\_,\_,\_,\_,\_,\_,Decimo],All,Winner):-

check\_linha\_primeira(Primeira,Winner1),

check\_linha\_ultima(Decimo,Winner2),

((find\_board\_linha(All,49))->

Winner3 is 2

;

((find\_board\_linha(All,50))->

Winner3 is 1

;

Winner3 is 0

)

),

check\_winner(Winner1,Winner2,Winner3,Winner).

game\_over(Board,Winner):-

game\_over2(Board,Board,Winner).

Check\_linha\_primeira e check\_linha\_ultima verificam se as cidades em ambos os lados do tabuleiro ainda se encontram no tabuleiro (sendo a sua captura condição de vitória para o adversário. As condições após isso verificam se os jogadores estão sem peças ou não, e atribuem um vencedor de acordo com o resultado.

Por último, check\_winner processa os valores obtidos em Winner1, Winner2 e Winner3 e retorna no parâmetro Winner o vencedor.

## Avaliação do Tabuleiro

Para gerar num computador um adversário que estivesse ao nível da dificuldade indicada, foi necessário efetuar uma avaliação eficiente do tabuleiro e respetivas jogadas possíveis.

O nosso predicado value analisa o tabuleiro para determinar quanto valor possível total ainda existe nos movimentos das peças neste. Para isto serve-se dos predicados valid\_pecas\_linhas, que obtem as pecas do board de um determinado jogador e as retorna numa lista, lista essa que é percorrida por percorrer\_pecas, sendo que este predicado retorna o valor total das possíveis jogadas na lista.

O valor de uma jogada depende do que esta implica para o estado do jogo. Um movimento sem captura tem um valor de 0, formar um canhão novo tem um valor de 1, capturar um soldado inimigo tem um valor de 2 e capturar a cidade inimiga tem um valor de 10 (sendo assim superior a qualquer combinação de jogadas até uma pesquisa em profundidade de 5 jogadas, algo que não fazemos).

Durante o decorrer do jogo fazemos pesquisas em profundidade da lista de jogadas, ou seja repetimos a pesquisa para o estado do tabuleiro após cada sucessiva jogada, atribuindo então uma pontuação à soma final. Desta forma, uma jogada para capturar a cidade que implique duas jogadas anteriores sem captura (de valor 0) será escolhida em vez de uma jogada que capture um soldado inimigo mas que, após isso, não faça mais nada de significativo.

## Jogada do Computador

Para simular a dificuldade quando se joga contra um computador, associamos a dificuldade a um diferente nível de profundidade na pesquisa do valor dos estados do tabuleiro. “Fácil” é apenas de nível 1, ou seja pesquisa apenas uma vez e executa a jogada de maior valor, enquanto que “Normal” e “Difícil” são de nível 2 e 3 (ou seja, repetem a pesquisa mais uma e duas vezes, respetivamente). Isto é feito através do predicado “make\_move” que está responsável pela escolha da jogada durante o turno do computador.

O predicado make\_move atualiza o jogador, faz a valorização das jogadas percorrendo as peças de cada linha uma a uma, faz o sort desses movimentos com base no valor a eles atribuído e escolhe o movimento com maior valor. Após isso, atualiza o tabuleiro com a jogada executada e passa para o turno seguinte.

Uma imagem com palavras cruzadas, texto

Descrição gerada automaticamente

# Conclusões

O software desenvolvido é uma implementação de “Cannon” de acordo com os parâmetros do projeto que julgamos ser bem sucedida, sendo eficiente na sua execução e atingindo resultados satisfatórios.

Quanto a dificuldades, os maiores entraves ao desenvolvimento deste projeto foram a adaptação a uma linguagem e tipo de programação com que não estávamos muito familiarizados. No entanto, após o período de adaptação inicial, esses problemas foram resolvidos.

Quanto a melhoramentos ao trabalho, seria desejável poder atingir a pesquisa em profundidade com níveis superiores, mas esse processo revela-se muito demorado por muito que tenhamos optimizado o programa. Algo seria possível se decidíssemos filtrar a pesquisa, por exemplo, mas isso poderia levar a erros como ignorar jogadas simples que não tivessem recompensa imediata, e foi algo que decidimos não explorar.

Concluindo, podemos dizer que o principal objetivo do projeto, o de aumentar os nossos conhecimentos relativamente ao conteúdo da unidade curricular de Programação em Lógica, foi cumprido.

## Bibliografia

### Fontes sobre o jogo:

<https://nestorgames.com/rulebooks/CANNON_EN.pdf>

<https://nestorgames.com/#cannon_detail>

### Fontes sobre PROLOG:

<https://sicstus.sics.se/sicstus/docs/3.7.1/html/sicstus_toc.html>