Universidade de Évora

Disciplina de Programação Declarativa

Jogo Linetris

Autores:
Ricardo Fusco, 29263
Ruben Borrega, 30005
André Almeida, 31112

Professor:
Vitor Nogueira

January 3, 2015

Índice

Lista de Figuras

1 Introdução

No âmbito da unidade curricular de Programação Declarativa pretende-se usar a matéria abordada ao longo das aulas no que diz respeito a Prolog como, por exemplo, estruturas completas e incompletas, algoritmos de processamento de listas, functores, entre outros conceitos importantes relacionados com a linguagem, para criar um jogo chamado Linetris, que consiste basicamente nos mesmos princípios do jogo do 4 em Linha com uma regra adicional proveniente do jogo Tetris, quando cheia, a última linha desaparece e o tabuleiro desce uma linha. O trabalho está dividido em 2 fases, o modo Humano vs Humano e o modo Humano vs AI. O tabuleiro de jogo será um tabuleiro dinâmico, ou seja, o tamanho (número de linhas e número de colunas) poderá ser escolhido pelo utilizador, e as peças de cada jogador serão, por defeito, as peças pretas para o Jogador 1 e peças vermelhas para o Jogador 2 e AI. Para este trabalho é utilizado o swi-prolog como tem sido feito nas aulas práticas.

2 Desenvolvimento

2.1 Escolha da estrutura

No começo do desenvolvimento do trabalho o objectivo inicial era a escolha de uma estrutura viável para o tabuleiro de jogo. Numa primeira fase começámos com functores, à semelhança da estrutura usada numa das práticas para representar matrizes, um functor de functores ——> matriz(linha(...),linha(...))). Mas como não estávamos a conseguir manipular a estrutura para fazer certas coisas que queríamos e acabava por se tornar mais complicado fazer certas operações, para não estarmos a perder mais tempo parados com o mesmo problema, mudámos a estrutura para uma lista de listas. Esta estrutura foi muito mais simples de manipular para resolver os problemas com que nos deparámos com os functores.

2.2 Inicialização da estrutura

Após escolhida a estrutura procedemos então à implementação dos predicados necessários para inicializar uma lista de listas de tamanho variável com elementos não instanciados com o intuito de ter um tabuleiro dinâmico que se possa parametrizar com número de linhas e número de colunas escolhidos pelo utilizador (**inicializar/3**).

Para inicializar a estrutura é usado um predicado auxiliar do swi-prolog, **length/2**, que, além de devolver o tamanho de uma lista, nos permite criar uma lista de aridade N passando como argumento a aridade. É necessário passar como argumentos o número de linhas e o número de colunas no predicado inicializar, para criar uma lista de aridade igual ao número de linhas e de seguida percorrer cada elemento dessa lista e fazer o mesmo, criar uma sub-lista de aridade igual ao número de colunas em cada posição da lista, ficando no final uma matriz com as dimensões desejadas com elementos/variáveis não instanciadas. A criação das linhas é feita usando o predicado do swi-prolog **maplist/2** mapeando o predicado **cria_linha/2** a cada elemento da lista da matriz passando-lhe como argumento o número de colunas.

```
?- inicializar(4,4,Matriz).
Matriz = [[_G903, _G906, _G909, _G912], [_G915, _G918, _G921, _G924],
          [_G927, _G930, _G933, _G936], [_G939, _G942, _G945, _G948]] .
```

Figure 1: Inicialização Matriz

2.3 Inserção numa posição (X,Y) da matriz

O predicado para inserir uma peça na posição (X,Y) da matriz (**posicao (X, Y, Elemento, Matriz)**) foi relativamente simples de fazer utilizando como auxiliar o predicado **nth1/3** do swi-prolog que nos permite não só inserir numa posição da matriz, caso nessa posição esteja uma variável não instanciada ele unifica o valor a inserir com a variável, como também obter o valor atómico que esteja nessa posição.

O predicado **nth1/3** foi usado para obter a linha desejada (X), e é utilizado outra vez para obter o elemento ou unificar um valor a inserir com o elemento dessa linha na coluna desejada (Y).

2.4 Print do tabuleiro

Após fazer os predicados anteriormente descritos sentimos que seria mais simples fazer certos testes com estes predicados se conseguíssemos fazer um print do tabuleiro um pouco mais perceptível e mais organizado do que ver listas de listas enormes no terminal. Para tal foram feitos os predicados **print_linhas/1** e **print_elementos/1**. O primeiro predicado percorre cada sub-lista (linha) da matriz do tabuleiro e para processar cada uma dessas sub-listas é usado o predicado **print_elementos/1** que percorre cada elemento dessa linha e, caso seja atómico, escreve esse elemento no std output, caso seja uma variável não instanciada escreve um '-'. O predicado **print_tabuleiro/1** apenas faz um "new line" antes e depois de processar todas as linhas do tabuleiro. O aspecto do tabuleiro imprimido no std output poderá ser observado na figura 2.

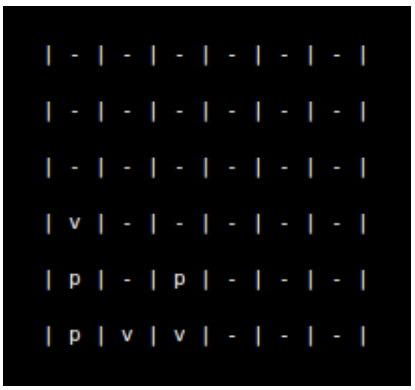

Figure 2: Tabuleiro de Jogo

2.5 Inserção por coluna

Com a inserção por coordenadas (X,Y) e o print do tabuleiro concluidos a próxima etapa foi conseguir inserir por coluna (**insere/4**) utilizando a inserção já feita como auxiliar. O predicado **insere/4** recebe a peça e a coluna onde se pretende inserir, recebe a matriz onde inserir e devolve as coordenadas da peça inserida na forma de um functor, p(X,Y) sendo X a linha e Y a coluna. Para saber qual a linha onde inserir a peça é usado o predicado **findall/3** para "coleccionar" todos os elementos não instanciados (não atómicos) da coluna pretendida numa lista e o tamanho dessa lista resultante será a linha exacta onde a peça será inserida. Por exemplo, dando-se o caso de o número de linhas ser 5, se a lista de todos os elementos da coluna onde se pretender inserir for [_ , _ , _ , _ , b], a linha onde a peça será inserida será a linha 4 pois há 4 variáveis não instanciadas na lista. Caso todos os elementos nessa coluna sejam atómicos o predicado falha e esse caso será tratado no predicado do loop principal do jogo (**ciclo/4**).

2.6 Menus

Tendo já a base para o trabalho construída, a seguinte etapa foi fazer o loop para o menu principal e sub-menus e o loop principal do jogo. A parte do menu principal e dos sub-menus não foi complicada, consistiu apenas em escrever no std output o nome do jogo e as opções disponíveis, e, de seguida, pedir input ao utilizador (**menu/0**) e tratar o input com o predicado **menuOpt/1**.

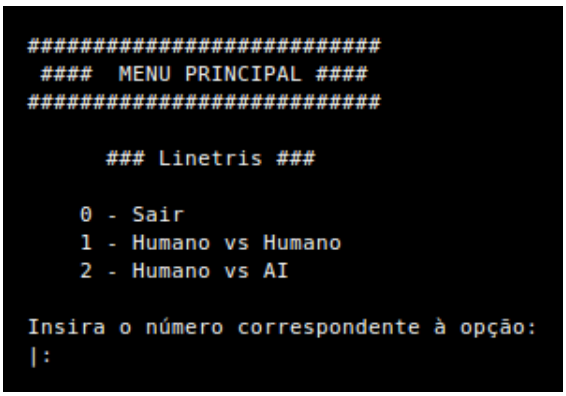

Figure 3: Menu principal

Caso o utilizador escolha a opção 1 (Humano vs Humano), são-lhe então pedidas as dimensões do tabuleiro, número de linhas (N) e número de colunas (M), é depois verificado se o input é válido (Se for inserido um número inteiro que seja pelo menos 4), caso o seja inicializa a matriz e entra no loop principal do jogo onde é alternado entre o jogador 1 e jogador 2, caso o input seja inválido é pedido outra vez input ao utilizador chamando o mesmo predicado recursivamente.

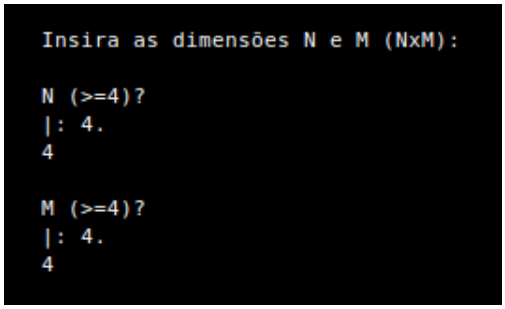

Figure 4: Sub-menu (Humano vs Humano)

Voltando novamente às opções do menu principal, caso o utilizador escolha a opção 2 (Humano vs AI), é mostrado um sub-menu igual ao da opção 1 em que são pedidas as dimensões do tabuleiro e depois de validado o input é mostrado outro sub-menu onde é pedido ao utilizador que escolha quem joga primeiro, se o Jogador 1 ou a AI (fig. 5).

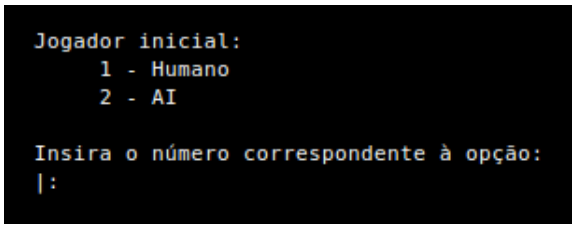

Figure 5: Sub-menu (Humano vs AI)

Caso o utilizador escolha a opção 0 é quebrado o loop do menu, saindo assim da execução do programa. Caso o utilizador escolha qualquer opção que não seja 0, 1 ou 2, e novamente pedido input ao utilizador chamando o predicado **menu/0** novamente.

2.7 Verificação terminal do Jogo

Antes de começarmos a elaborar o loop principal do jogo, ainda construímos os predicados para verificar se o jogo está em condições de terminar e, caso esteja, qual dos jogadores ganhou. A razão do predicado **insere/4** devolver as coordenadas da peça que foi inserida é estas serem necessárias para verificar o terminal do jogo (**verifica_terminal/3**) no sentido em que apenas iremos verificar a linha e a coluna onde a última peça foi inserida em vez de verificar todas as linhas e todas as colunas o que acaba por tornar esta verificação mais eficiente.

```
%% verifica_terminal(+Matriz,+UltPecaJogada,?QuemGanhou)
%
verifica_terminal(Matriz,UltPecaJog,Ganhou):-
        verifica_linha(Matriz,UltPecaJog,Ganhou);
        verifica_coluna(Matriz,UltPecaJog,Ganhou);
        verifica_diagonais(Matriz,UltPecaJog,Ganhou).
```

Figure 6: Verifica terminal

No **verifica_linha/3**, basta apenas aceder ao índice da linha onde a ultima peça foi inserida com o **nth1/3** e chamar o predicado **ver_pecas/4** para a lista de elementos dessa linha.

O predicado **ver_pecas/4** recebe uma lista, o número de peças pretas e vermelhas e caso o predicado suceda devolve qual o jogador que ganhou (1 ou 2). Este predicado é chamado com um número de peças pretas e vermelhas igual a 0, este vai percorrendo a lista incrementando o número de peças (v ou p) caso encontre elementos atómicos, quando encontra um elemento não atómico (var não instanciada) chama este predicado recursivamente fazendo um reset no número de peças vermelhas e pretas voltando a 0 novamente. Caso chegue ao fim da lista e não haja 4 peças seguidas pretas ou vermelhas o predicado falha. Caso encontre 4 seguidas sucede e devolve o jogador que ganhou.

Voltando ao predicado **verifica_terminal/3**, caso o predicado **verifica_linha/3** falhe, é chamado o predicado **verifica_coluna/3**, este predicado faz um findall de todos os elementos da coluna onde a última peça foi inserida e é feita a verificação das peças com o **ver_pecas/4**.

Caso falhe a verificação da coluna é feita a verificação das diagonais. No que diz respeito às diagonais não conseguimos fazer com que fossem apenas verificadas as 2 diag-

onais que passam no ponto onde a peça foi inserida. No predicado **verifica_diagonais/3** é usado o predicado **check_diags/2**. Este predicado faz um findall de todas as peças do tabuleiro e coloca-as na lista resultante na forma de um termo p(X,Y,Peça), em que X é a linha, Y a coluna e o 3 argumento do termo é a peça nessa posição, de seguida é passada essa lista e a ultima peça que foi jogada (v ou b) como argumento ao predicado **encontra_diag/2**. Esse predicado vai procurar 4 peças seguidas numa diagonal, no primeiro **member/2** que é chamado são unificadas as coordenadas X e Y com os primeiros valores que encontrar e irá ver se as seguintes 3 posições na diagonal têm a mesma peça, usando novamente o predicado **member/2**, a partir da peça nas coordenadas X,Y. Assim que encontrar 4 peças seguidas numa diagonal sucede, caso não encontre falha. Basicamente vai indo de peça em peça fazendo backtracking e testando as peças seguintes numa das duas diagonais.

Caso o **check_diags/2** suceda, é então usado o predicado **player/4** para saber qual o vencedor (jog 1 ou 2) de acordo com a peça (p ou v).

2.8 Verificação da última linha do tabuleiro

Após ter a verificação do terminal do jogo concluída, a próxima e última fase antes de construir o predicado do loop principal foi a verificação da última linha do tabuleiro. A regra do jogo Linetris que faltava implementar era caso a última linha estivesse preenchida essa linha era removida e o tabuleiro descia uma linha. Para verificar a última linha fizemos o predicado **check_lastLine/1** que recebe como argumento a matriz do tabuleiro de jogo e vai usar o predicado **length/2** para saber o tamanho da matriz, ou seja, o número de linhas e esse número de linhas irá ser o índice da última linha na matriz usando depois o **nth1/3** para obter a lista da última linha e passá-la como argumento ao predicado **all_atomic/1**. O predicado **all_atomic/1** por sua vez irá usar um predicado do swi-prolog, **maplist/2**, que irá suceder se um goal, neste caso o goal **atomic/1**, for aplicável a todos os elementos da lista que receber como argumento.

Depois de verificada a última linha, caso esta tenha apenas elementos atómicos significa que está preenchida e o tabuleiro precisa de descer uma linha. Para descer o tabuleiro foi feito o predicado **desce_tabuleiro/2** que recebe como argumento a matriz e devolve a nova matriz já sem a última linha e com uma nova linha de variáveis não instanciadas na primeira posição da matriz. Primeiro é usado o length para determinar o tamanho da matriz e, consequentemente, o índice da última linha na matriz, e é depois utilizado o predicado **nth1/4** para remover a última linha. Este **nth1** de aridade 4 possui uma particularidade interessante que o de aridade 3 não possui, este pode ser utilizado tanto para remover um elemento duma lista e obter a lista resultante como para adicionar um elemento num determinado índice duma lista e obter a lista resultante. Depois de remover a última linha é criada uma nova linha com vars não instanciadas usando o comprimento da lista da última linha. Por fim é utilizado novamente o **nth1/4** desta vez para adicionar a nova lista no início da matriz, no índice 1.

2.9 Loop principal do jogo e heurística

O predicado do loop principal do jogo, **ciclo/4**, recebe como argumentos o jogador, 1 ou 2, a matriz do tabuleiro, a última peça jogada e recebe o modo de jogo, Humano vs Humano (hvsh) ou Humano vs AI (hvsc). A primeira coisa a fazer no início do ciclo, independentemente de qual for o jogador a jogar, é verificar o terminal do jogo e, caso se verifiquem condições de vitória, interromper o loop e escrever no std output o tabuleiro e qual o vencedor.

```
%% ciclo(+Jogador, +Matriz,+UltPecaJogada, +ModoJogo)
%
% primeira coisa no ciclo é verificar o terminal do jogo (se alguém ganhou)
ciclo(_,Matriz,UltPecaJog,Modo):-
    verifica_terminal(Matriz,UltPecaJog,Ganhou), !, print_tabuleiro(Matriz),
    write('O vencedor é: '), player(Ganhou,_,J,Modo),write(J).
```

Figure 7: Ciclo - Terminal

É utilizado novamente o predicado **player/4** neste caso para determinar o que imprimir de acordo com o número do jogador vencedor (1 ou 2) e de acordo com o modo de jogo.

Caso não se verifiquem condições de vitória o próximo passo antes de qualquer jogador fazer a sua jogada é verificar a última linha, e, se estiver preenchida na totalidade, descer o tabuleiro.

Caso as duas verificações anteriores falhem, se considerarmos o modo humano vs humano, irá ser pedida a coluna como input ao jogador, depois de inserida a coluna onde jogar é feita a inserção da peça consoante o jogador (1-p ou 2-v) caso a inserção suceda é chamado o predicado ciclo/4 recursivamente passando como argumento o próximo jogador e a peça que foi jogada e as suas coordenadas (p(X,Y,Peça)), caso a inserção falhe (quando o jogador tenta jogar numa coluna já cheia) é chamado o ciclo recursivamente para o mesmo jogador outra vez pedindo-lhe novamente input para a coluna.

Na porção do modo Humano vs AI do ciclo , a parte relativamente ao jogador 1 é exactamente igual à do modo Humano vs Humano como foi referido anteriormente. Na parte do jogador 2 (AI) a primeira coisa a fazer é gerar uma lista com as colunas onde se pode inserir usando o predicado **geraNumsI_J/3** passando como argumento o mínimo (1) e o máximo (Número de colunas) e devolvendo a lista com os números entre o mínimo e o máximo inclusivé, o que este predicado faz é apenas um findall de todos os números entre I (min) e J(máx) e devolve a lista resultante. Depois é feito um findall para obter uma lista com todos os pares Valor-Coluna usando o goal **simula_jogs/4**.

O predicado **simula_jogs/4** recebe como argumento a matriz do tabuleiro e a lista com as colunas onde é possível inserir e devolve a coluna e o valor associado a essa coluna. Para obter este valor foi criada uma função de avaliação (heurística) para verificar o tabuleiro e devolver um valor de acordo com o estado do tabuleiro.

Esta função de avaliação consiste em contar quantas peças isoladas o jogador tem, por isoladas entenda-se sem qualquer peça igual num dos pontos à volta da peça a considerar, conta o número de 2 peças iguais numa linha, coluna ou diagonal, conta o número de 3 peças iguais e o número de 4 peças iguais e soma esses 4 valores, depois irá fazer a

mesma verificação para o adversário e soma também os 4 valores. Nesses 4 valores é dado um peso maior a quem tiver 4, 3 ou 2 peças seguidas, de seguida subtrai a soma das do adversário à soma das do jogador para o qual se está a fazer a avaliação. Caso o jogador tiver 4 peças seguidas é somado um valor grande para garantir que depois essa coluna seja escolhida dando a vitória ao jogador. Os predicados que encontram as peças são similares ao predicado **encontra_diag/2** referido anteriormente no **verifica_diagonais/3**.

```
%% func_aval(+Tuplo_MatrizPeca,?ValorAvaliacao)
%
% dados a matriz(M) e a peça(P) que vai jogar, returna em Val o valor da avaliacao
func_aval((Matriz,Peca), Val):-
  %encontra todos os elementos atómicos e as suas coordenadas
  findall(p(X,Y,E), (posicao(X,Y,E,Matriz),atomic(E)), L),
  find_all_1peca(L, Peca, V1), find_all_2pecas(L, Peca, V2),
  find_all_3pecas(L, Peca, V3), find_all_4pecas(L, Peca, V7),
  invertePeca(Peca,Peca1),
  find_all_1peca(L, Peca1, V4), find_all_2pecas(L, Peca1, V5),
  find_all_3pecas(L, Peca1, V6), find_all_4pecas(L, Peca1, V8),
  Val1 is (V1+3*V2+6*V3+9*V7)-(V4+3*V5+6*V6+9*V8),
  ((V7 >= 1,Val is Val1+1000); Val = Val1).
```

Figure 8: Função de avaliação

Antes de determinar os valores descritos anteriormente é feito um findall para obter uma lista com todos os elementos atómicos da matriz guardando também as coordenadas de cada elemento - "**findall(p(X,Y,E), (posicao(X,Y,E,Matriz),atomic(E)), L)**".

Tomemos como exemplo o predicado **find_all_2pecas/3** para explicar os outros predicados que usam a mesma lógica como o **find_all_3pecas/3** e **find_all_4pecas/3**. Este predicado recebe como argumentos a lista de obtida a partir do findall referido anteriormente, a peça que vai verificar (p ou v) e devolve o número de 2 peças. Este predicado faz um findall de todas as 2 peças usando o predicado **find_2pecas/2**. O predicado **find_2pecas/2** verifica se há duas peças seguidas numa linha, numa coluna ou numa das duas diagonais unificando as coordenadas X,Y com o primeiro valor que encontrar usando o member e verificando se a próxima peça é igual e se no 3º ponto estiver uma variável não instanciada. No fim basta apenas usar o length para saber quantas "2 peças seguidas" o jogador tem. A mesma lógica aplica-se aos restantes predicados do mesmo género.

```
%% find_all_2pecas(+Lista, +Peca, ?Num2pecas)
%
% conta o numero total de 2p's ou 2v's
find_all_2pecas(L, Peca, N):-
  findall(Peca, find_2pecas(L, Peca), L2),length(L2, N).

%% find_2pecas(+Lista, +Peca)
%
% encontrar 2p's ou 2v's na linha
find_2pecas(L, Peca):-
  member(p(X,Y,Peca), L),
  Y1 is Y-1,Y2 is Y-2,
  member(p(X,Y1,Peca), L), member(p(X,Y2,Peca1), L),
  \+ atomic(Peca1).
```

Figure 9: Predicados find_all_2pecas e find_2pecas numa linha

Voltando ao predicado **simula_jogs/4**, a primeira coisa que é feito é um member da lista das colunas para simular a jogada da AI, depois de simulada a inserção é feita a

avaliação do tabuleiro, de seguida é verificada a regra da ultima linha, após a verificação é feito outro member agora para simular a coluna onde o jogador 1 poderia inserir uma peça. Apóes simular a inserção do jogador 1 é avaliado o tabuleiro mais uma vez com a função de avaliação desta vez passando como argumento a peça do jogador 1. De seguida são tratados os valores obtidos, caso o valor da primeira avaliação seja maior que o valor da segunda, a coluna a inserir será a primeira coluna onde foi feita a primeira simulação e o valor devolvido pelo predicado será o valor da primeira avaliação. O mesmo acontence caso o valor da segunda avaliação seja maior que o da primeira a coluna a inserir e o valor serão os da segunda simulação onde o jogador 1 iria jogar. Caso os 2 valores sejam iguais o valor a devolver será a subtração do valor da segunda avaliação ao da primeira e a coluna a inserir será a coluna da primeira simulação.

Depois de obtida a lista dos pares Valor-Coluna no findall na parte do AI no loop principal (**ciclo/4**), é encontrado o par com o Valor maior usando o predicado **max_member/2**, a seguir irá ser inserida a peça do AI na coluna do par com o maior valor e muda para o jogador 1 de novo. Acontece também haver casos em que não há nenhuma jogada possível viável, ou seja, quando a lista do findall dos pares Valor-Coluna fica vazia nesse caso vai ser escolhida uma coluna ao acaso dentro das colunas onde é possível jogar.

Com isto ficou então terminado o trabalho, para correr o jogo basta executar o predicado go. no prompt do swi-prolog. Encontra-se também um script bash na pasta src que o professor nos ajudou a fazer que corre o jogo para cada ficheiro de input na pasta inputs e escreve o resultado como output na pasta outputs.

3 Conclusão

Olhando em retrospectiva para o trabalho que elaborámos podemos certamente concluir que fomos bem sucedidos no cumprimento dos objectivos necessários para desenvolver este jogo. Conseguimos fazer ambos os modos de jogo, Humano vs Humano e Humano vs AI, conseguimos fazer a verificação do terminal o mais optimizada possível verificando apenas o que é necessário, à excepção talvez da verificação das diagonais que poderia ter sido mais optimizada ainda. Foram verificados e tratados os casos extra como, por exemplo, os casos em que o utilizador não coloca inputs válidos, ou os casos em que um utilizador tenta inserir uma peça numa coluna que já se encontra cheia, etc.

No que diz respeito à heurística utilizada para a AI também falhámos um pouco no que diz respeito à optimização ficando um conjunto de predicados bastante extenso e verboso, mas apesar disso conseguimos com que a AI fizesse quase sempre a melhor jogada simulando 2 jogadas à frente, seguindo um método que em nada é aleatório que era justamente o que era pedido de nós. Nos testes que fizemos até agora só ainda conseguimos ganhar em 2 ou 3 situações, por exemplo, se forçarmos uma situação em que temos um jogo parecido ao que se vê na figura 10. Basicamente em situações em que a AI é forçada a cortar a jogada do jogador e após esse corte o jogador fica em condições de poder ganhar. Apesar destes pormenores a AI permanece um bom adversário para quem jogar.

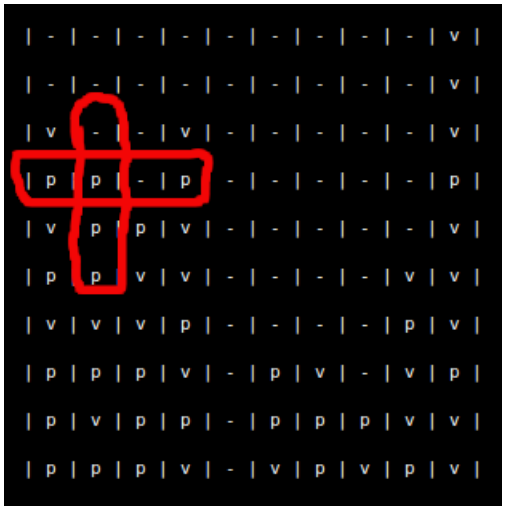

Figure 10: Situação vitória Jogador

Tirando pontualmente algumas optimizações que poderíamos ter feito para enriquecer o código e melhorar o funcionamento do jogo, podemos concluir que o balanço deste trabalho foi bastante positivo. Conseguimos também, com este trabalho, enriquecer muito mais os nossos conhecimentos de Prolog solidificando um pouco mais as nossas bases no que diz respeito a esta linguagem e às metodologias utilizadas em prol desta.