

Desenvolvimento de um agente autónomo para o jogo **Tetris**

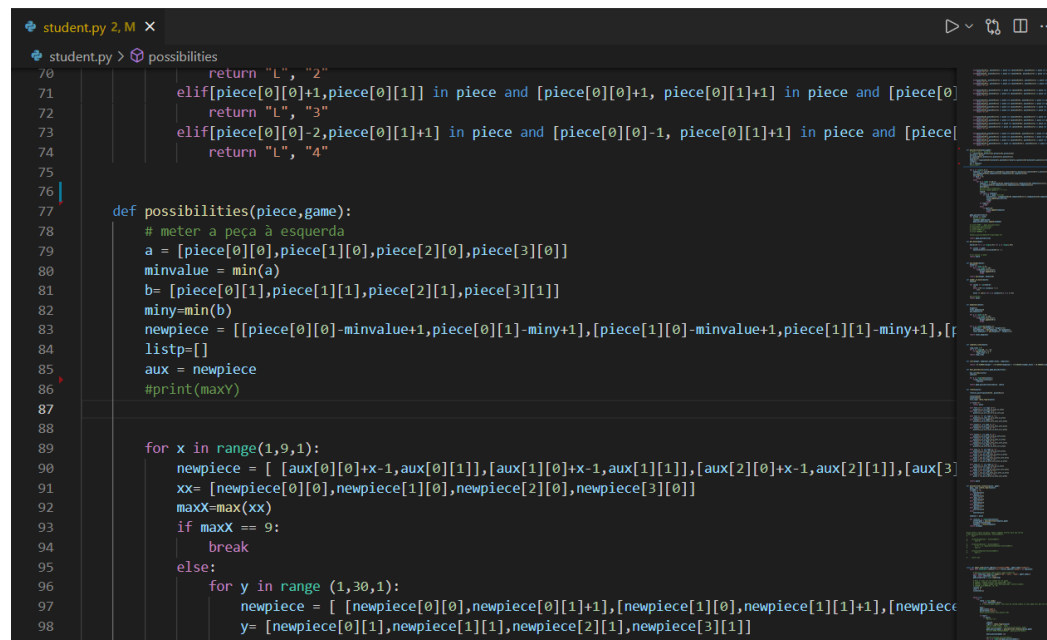
Relatório



Introdução

O projeto em questão consiste em desenvolver um agente inteligente, em python, capaz de resolver o tetris.

O funcionamento deste agente foi criado exclusivamente no ficheiro **student.py**.



```
student.py 2, M x
student.py > possibilities
70         return "L", "2"
71     elif[piece[0][0]+1,piece[0][1]] in piece and [piece[0][0]+1, piece[0][1]+1] in piece and [piece[0]
72         return "L", "3"
73     elif[piece[0][0]-2,piece[0][1]+1] in piece and [piece[0][0]-1, piece[0][1]+1] in piece and [piece
74         return "L", "4"
75
76
77 def possibilities(piece,game):
78     # meter a peça à esquerda
79     a = [piece[0][0],piece[1][0],piece[2][0],piece[3][0]]
80     minvalue = min(a)
81     b= [piece[0][1],piece[1][1],piece[2][1],piece[3][1]]
82     miny=min(b)
83     newpiece = [[piece[0][0]-minvalue+1,piece[0][1]-miny+1],[piece[1][0]-minvalue+1,piece[1][1]-miny+1],[
84     listp=[]
85     aux = newpiece
86     #print(maxY)
87
88
89     for x in range(1,9,1):
90         newpiece = [ [aux[0][0]+x-1,aux[0][1]], [aux[1][0]+x-1,aux[1][1]], [aux[2][0]+x-1,aux[2][1]], [aux[3]
91         xx= [newpiece[0][0],newpiece[1][0],newpiece[2][0],newpiece[3][0]]
92         maxx=max(xx)
93         if maxx == 9:
94             break
95         else:
96             for y in range (1,30,1):
97                 newpiece = [ [newpiece[0][0],newpiece[0][1]+1],[newpiece[1][0],newpiece[1][1]+1],[newpiec
98                 y= [newpiece[0][1],newpiece[1][1],newpiece[2][1],newpiece[3][1]]
99                 maxY=max(y)
```

Organização de ideias

Numa primeira fase começamos por entender como é que o jogo estava criado e desenvolvido.

A grelha do tetris tem 8 células de largura e 29 células de altura. Temos 7 peças em jogo, "O","I","S","Z","J","T","L". Todas elas são identificadas por uma lista de coordenadas ('piece') e possuem vários estados, mediante o seu estado de rotação. Também possuímos de uma lista onde mostra todas as coordenadas das peças que estão em jogo('game').

Depois de avaliar as bases do jogo, o nosso objetivo foi conseguir descobrir o melhor movimento possível para cada peça que estava a cair. Para atingir esse objetivo o nosso agente teve que ser capaz de calcular um custo para cada movimento possível e selecionar a jogada com o melhor deste custo. Estes custos baseia-se em 4 heurísticas: altura máxima, numero de buracos, linhas completas e diferenças de altura entre peças.

Algoritmo

Inicialmente desenvolvemos uma função que conseguisse calcular todas as possibilidades para a peça atual e para todas as peças rodadas e que retornasse uma lista com todas as possibilidades de coordenadas. Para além disso fizemos um função de reconhecimento de peças. Em que, dada as coordenada de uma peça, essa função

De seguida começamos a desenvolver as funções das heurísticas ('aggregate_height', 'number_of_holes', 'bumpiness' e 'complete_lines') onde cada uma retornava o seu valor. Com isto foi-nos possível calcular as heurísticas para todas as possibilidades e calcularmos um certo custo. O nosso custo foi calculado desta forma:

$$a*(\text{max_height}) + b*(\text{number_of_holes}) + c*(\text{bumpiness}) + d*(\text{complete_lines})$$

Como queremos minimizar a altura máxima, o numero de buracos e o bumpiness pusemos estas contantes (a,b e c) a negativo. Para maximizar o número de linhas completas a contante d teria de ficar a positivo.

A possibilidade escolhida era aquela que tinha o maior custo. Mediante isto, selecionamos essa hipótese e mandamos para essas coordenadas as peças com o auxilio das nossas keys ('w', 'd', 'a' e 's').

Heurísticas

- Aggregate height: minimizando o valor total da altura jogamos mais peças na grid de forma a não atingir tão rapidamente o topo.
- Complete lines: maximizando o numero de linhas completas atingimos o objetivo do jogo e limpamos as linhas de forma a dar mais espaço para as próximas peças.
- Number_of_holes: Sendo os buracos difíceis de eliminar temos que minimizar o aparecimentos deles.
- Bumpiness: O topo do grid deve ser o mais monótono possível. Para isso não deve haver muita variação das alturas entre colunas. Calculámos esta variação (bumpiness) da seguinte forma:

$$\text{Bumpiness} = |x_0 - x_1| + |x_1 - x_2| + |x_2 - x_3| \dots |x_7 - x_8|$$

Possibilidades de jogo e a melhor possibilidade

Desenvolvemos uma função que devolve uma lista com todas os games de possibilidades. Para isso simulámos todas as possíveis posições para onde a peça pode cair e guardamos essas coordenadas numa lista.

De seguida também desenvolvemos a função onde adicionamos à lista das possibilidades, as possibilidades da peça rodada.

Decidimos criar uma função onde convertemos um game numa board de 0's e 1's, onde os '1's representam os sítios onde há peça.

Com isto já nos foi mais fácil calcular as heurísticas para todas os games de possibilidades da lista de possibilidades.

A melhor possibilidade foi escolhida a partir de uma função onde analisamos todos os custos de cada game. O que tiver menor custo é o game escolhido.