Universidade do Minho
Escola de Engenharia

Ganda Galo

Trabalho realizado por:

Lara Vaz A88362

Mariana Carvalho A88360

Margarida Gonçalves A88830

Docentes:

Filipa Ferraz

Vítor Alves

Introdução

De forma a demonstrar os conhecimentos adquiridos ao longo do semestre no âmbito da unidade curricular "Introdução aos Paradigmas da Programação", foi-nos proposta a realização de um trabalho de grupo com o objetivo de criar um jogo, o "Ganda Galo". Este trabalho, desenvolvido no ambiente integrado *Spyder*, tem como principal objetivo a realização de um programa funcional, sem erros e eficiente. A finalidade do "Ganda Galo" é a de o utilizador conseguir completar um dado tabuleiro, em que existem peças bloqueadas (representadas por '#'), com 'X's e 'O's sem que existam mais do que dois símbolos iguais consecutivos. Como grupo, queremos consolidar os nossos conhecimentos em *Python* e sermos capazes de elaborar um programa de forma autónoma e livre.

Especificação do problema

De modo à concretização eficaz do trabalho proposto, é necessária a especificação do problema, bem como a sua compreensão. Pretende-se assim que o grupo elabore um programa com os seguintes componentes:

Mostrar: comando mostrar que leva como parâmetro o nome de um ficheiro;

Abrir: comando carregar que leva como parâmetro o nome de um ficheiro;

Gravar: comando gravar que leva como parâmetro o nome de um ficheiro;

Jogar: comando jogar que leva como parâmetro o caractere referente à peça a ser jogada ('X' ou 'O') e dois inteiros que indicam o número da linha e o número da coluna, respetivamente, onde jogar;

Validar: comando validar que testa a consistência do puzzle e verifica se o tabuleiro está válido;

Ajuda: comando ajuda que indica a próxima casa lógica a ser jogada (sem indicar a peça a ser colocada);

Undo: comando para anular movimentos (retroceder no jogo);

Resolver: comando para resolver o puzzle;

Ancora: comando âncora que deve guardar o ponto em que está o jogo para permitir mais tarde voltar a este ponto através do comando undoancora. Funciona como um estado temporário;

Undoancora: comando undo para voltar à última ancora registada;

Gerar: comando gerar que gera puzzles com solução única e leva três números inteiros como parâmetros: o nível de dificuldade (1 para 'fácil' e 2 para 'difícil'), o número de linhas e o número de colunas do puzzle;

Ver: comando para visualizar o puzzle em ambiente gráfico;

Sair: comando para sair do jogo.

Codificação

Comando mostrar: este comando já tinha sido implementado pelo professor.

Comando abrir: baseamo-nos no comando mostrar, carregando apenas o puzzle, sem apresentar o print do puzzle.

Comando gravar: No *Engine* criamos um método "escrever_tabuleiro_ficheiro", que abre o ficheiro que se pretende guardar, em modo de escrita. Este método percorre todo o tabuleiro e guarda as informações presentes neste ficheiro. No fim, o ficheiro é fechado. Na Shell, apenas invocamos o método acima referido, que leva como parâmetro, 1 argumento, o nome do ficheiro.

Comando jogar: No *Engine* definimos o método "jogar", que verifica se o símbolo que o utilizador inseriu é valido (se é um 'X' ou um 'O') e se a quadricula onde o utilizador pretende jogar se encontra livre. Pra isso implementamos o método "quadricula_ocupada" que averigua se uma quadrícula está ocupada. Se tudo isto se verificar então, a casa selecionada pelo jogador é preenchida com o símbolo digitado pelo mesmo. O tabuleiro é uma matriz, que começa na linha 0 e coluna 0. Porém, para o utilizador, isso corresponde as linha e coluna 1,1, logo no tabuleiro, a casa em que será colocada o símbolo é na casa [linha-1][coluna-1]. Cada jogada é colocada numa *Stack*, pois iremos utilizar isso para o método "undo". Se o tabuleiro estiver cheio, aparece um print, que informa o utilizador dessa situação. Para verificar se o tabuleiro está cheio, implementamos a função "tabuleiro cheio" para percorrer todo o tabuleiro, e ver se não existe nenhuma casa disponível. Na Shell, 'chamamos' o método jogar, que toma 3 argumentos, o símbolo, a linha e a coluna.

Comando validar: Na *Engine*, em cada uma das linhas do método "validar" verificamos uma série de condições, que pretendem verificar se existem 3 símbolos consecutivos iguais.

Na 1ª linha, começamos por delimitar que i-1> 0 e i+1 <nº linhas do tabuleiro, ou seja se as casas que pretendemos verificar fazem parte do tabuleiro. Posteriormente, afirmamos que essas casas tem de ser diferentes de '#' e de '.'. Verificando-se todas estas condições, verificamos se na diagonal [i-i][j-1], [i][j], [i+1][j+1] existem 3 símbolos iguais. Se tal acontecer o método retorna False. Na 2º linha, aplicamos o mesmo raciocínio para a outra diagonal do tabuleiro, na 3ª para as linhas, na 4ª para as colunas.

Após termos feito estas condições, reparamos que, mesmo três casas juntas terem símbolos diferentes, isso não implicaria que o tabuleiro fosse válido. Por exemplo, na 4ª linha verificamos se [i-1][j], [i][j],[i+1][j] tinham símbolos diferentes. Mesmo isso se verificando, [i-1][j] e [i][j] podiam ter o mesmo símbolo. Assim, concluímos que não é suficiente verificar isso, sendo necessário verificar também se [i-2][j], [i-1][j],[i][j] tem símbolos diferentes. Isto foi o que fizemos na 6ª linha. Na 5ª linha, aplicamos o mesmo raciocínio para os casas [i][j], [i+1][j],[i+2][j]. Nas 8ª e 7ª linhas verificamos condições semelhantes das que verificamos nas 6ª e 5ª linhas, mas para as linhas. Nas restantes linhas, fizemos o mesmo mas para as duas diagonais. Se todas estas condições não se verificarem, então o método retorna *True*. Na *Shell*, com o retorno de *True*, há um print que afirma que o puzzle é válido, e se o retorno for False, há um print que diz que o puzzle é inválido.

Comando ajuda: Neste comando, no *Engine*, usamos a estratégia que utilizamos para o "validar". Porém em vez de verificarmos se temos 3 casas consecutivas com símbolos iguais, verificamos se entre 3 casas consecutivas, em que as das extremidades tem o mesmo símbolo, a do meio está vazia. Se isso se verificar, o comando "ajuda" vai dar essa casa como palpite ao utilizador. Na Shell, criamos duas cópias do tabuleiro, 'aux1' e 'aux2'. A cópia 'aux1' corresponde a uma cópia relativa a jogada de 'X' na casa [i][j] e a 'aux2' é relativa a uma jogada de 'O' na casa [i][j]. Se apenas uma das cópias for válida, então há uma sugestão de jogada na casa [i][j]. Se isto não ocorrer, então 'aux1' é relativa a uma jogada de 'X' na casa [i+1][j+1] e 'aux2' é relativa a uma jogada de 'O' na casa [i+1][j+1]. Se apenas uma das cópias for válida, então há uma sugestão de jogada na casa [i+1][j+1]. Porém se as duas cópias forem válidas, isto é, se houver simultaneamente risco de haverem 3 'O's e 3'X's consecutivos, não há sugestão de jogada. Isto porque, como não sabemos qual o algoritmo que permite resolver os tabuleiros, esta foi a forma que arranjamos de corrigir o "ajuda" sem ter implementado o "resolver".

Comando undo: No *Engine*, se o tamanho da *Stack* em que guardamos as jogadas for maior que 0, então fazemos o *pop* da última jogada. Assim, o utilizador pode retroceder uma jogada. Se o tamanho for menor ou igual a 0, então há um print a dizer que não existem jogadas armazenadas. Na Shell, apenas invocamos o método "undo".

Comando resolver: Começamos por invocar o método "create_ancora" para criar uma lista com o estado do jogo num dado momento e depois adicionamos à *Stack* ancora, para guardar o estado do jogo antes de ver a resoluçao. Posteriormente, utilizamos o método "traduz_tabuleiro_lista" para percorrer a matriz tabuleiro, criamos uma lista com esta e contamos as casas vazias. Com o auxílio dos métodos *product* e *repeat* da biblioteca *itertools*, conseguimos preencher o suposto tabuleiro de todas as maneiras possíveis e em conjunto com o método "validar" retorna a única solução possível do puzzle. Depois disto, convertemos a solução para uma matriz tabuleiro com o método "carrega_lista_tabuleiro". No fim invocamos o "undo_ancora" para regressar ao jogo onde se ficamos.

Comando ancora: Invocamos na Shell o método "create_ancora" anteriormente explicado no resolver.

Comando undoancora: Invocamos na Shell o método "undo_ancora" anteriormente explicado no resolver.

Comando gerar: No Engine**,** avaliamos a dificuldade do puzzle. Se for 1 consideramos 50% das casas ocupadas no tabuleiro, se a dificuldade for 2 , 20% das casas ficam ocupadas. De seguida invocamos o método "iniciar_tabuleiro_vazio", cria um tabuleiro vazio. Voltamos a utilizar o

método product para ver as possíveis maneiras de preencher o puzzle e com o método random , escolhemos um tabuleiro aleatório dos possíveis resultantes das permutações feitas.

Na Shell, 'chamamos' o método "gera_forca_bruta", que toma 3 argumentos, a dificuldade, a linha e a coluna.

Comando ver: este comando implementado na *Shell*, é muito semelhante ao "mostrar", não tendo porém nenhum argumento, pois permite visualizar um determinado puzzle que já se contra aberto.

Comando sair: foi implementado pelo professor na *Shell*.

Verificação

Foram realizadas, à medida que íamos elaborando o trabalho, várias verificações para que nos pudéssemos certificar que os diferentes métodos estavam a funcionar como pretendíamos. Devido às verificações, conseguimos também detetar diversos erros, nomeadamente do "validar" e no "ajudar", em que numa fase inicial nos faltavam condições.

Conclusão

Neste trabalho conseguimos enriquecer o nosso conhecimento quer a nível da linguagem *Python*, quer também na resolução de diversos algoritmos.

Apesar do balanço geral ser positivo, tivemos bastantes dificuldades no "resolver" e no "gerar", devido ao grau de complexidade associado aos algoritmos que permitem resolver e implementar estes métodos. Nas nossas pesquisas, apercebemo-nos que a resolução dos mesmos, implicaria um conhecimento mais abrangente sobre Teoria de Jogos e Inteligência Artificial. Como tal, devemos aprofundar ambos os temas.

Para concluir, este trabalho foi bastante importante, pois ajudou-nos a consolidar diversos conhecimentos e estimulou o nosso raciocínio, parta desenvolvermos os diferentes algoritmos!