Objetivos

O presente trabalho tem como objetivo avaliar os conhecimentos adquiridos nas aulas da Unidade Curricular de "Introdução aos Algoritmos e à Programação", nomeadamente no que diz respeito à análise de problemas, elaboração de algoritmos e programação de computadores através do desenvolvimento de uma aplicação, em Python, para resolver um tipo de puzzle chamado GandaGalo. O GandaGalo é um jogo semelhante ao jogo do galo e ao sudoku: é jogado numa grelha com 'X's e 'O's, sendo que cada puzzle tem uma solução única. Tal como nos jogos referidos anteriormente, neste também não podem existir mais do que dois símbolos iguais consecutivos em qualquer direção – horizontal, vertical ou diagonal. O objetivo do jogo é preencher o tabuleiro com 'X's e 'O's, sem deixar nenhuma casa livre.

Descrição

O trabalho é constituído por 3 Classes: Window, Engine e Shell. Estas classes contêm métodos que permitem construir uma visão gráfica do tabuleiro e puzzle, nos quais é possível ter uma interação. Assim sendo, é possível escolher a jogada, gravar e validar o estado do tabuleiro, pedir ajuda, retroceder caso seja necessário e ainda a possibilidade de resolução automática do puzzle.

Métodos

De seguida, são apresentados os métodos das classes existentes, com uma breve descrição sobre cada um.

```
def do_mostrar(self, arg):
" -  comando mostrar que leva como parâmetro o nome de um ficheiro..: mostrar
<nome_ficheiro> \n"
```

O objetivo deste método é abrir um puzzle no formato de ficheiro, convertendo-o para um formato que posso ser lido pela *package* responsável pelo *design* gráfico e que também possa ser visto na *Shell* (desenvolvido pelo docente).

```
def do_abrir(self, arg):
" - comando abrir que leva como parâmetro o nome de um ficheiro..: abrir
<nome_ficheiro>  \n"
```

A função deste método é exatamente a mesma que o anterior, no entanto, esta função apenas imprime o puzzle na *Shell* (desenvolvido pelo docente).

```
def do_gravar(self, arg):
" - comando gravar que leva como parâmetro o nome de um ficheiro..: gravar
<nome_ficheiro>  \n"
```

Permite a gravação, criando um ficheiro com o nome fornecido no argumento.

```
def do_jogar(self, arg):
" - comando jogar que leva como parâmetro o caractere referente à peça a ser jogada
('X' ou 'O') e dois inte
```

O método **do_jogar** usa o método **gettabuleiro**, existente na classe *Engine*, à qual atribui uma variável, onde a matriz que compõe o tabuleiro é alterada com o símbolo escolhido, nas posições fornecidas nos argumentos.

```
def do_validar_resolver(self, arg):
" - comando validar que testa a consistência do puzzle e verifica se o tabuleiro está
válido: validar \n"
```

Este método apenas é executado durante o resolver automático dos puzzles, as suas funcionalidades serão explicadas no método **do_validar**.

```
def do_validar(self, arg):
" - comando validar que testa a consistência do puzzle e verifica se o tabuleiro está
válido: validar \n"
```

Neste método, é verificado inicialmente se o tabuleiro tem dimensões quadradas e se não contêm carateres não permitidos no jogo, ao chamar um método definido no *Engine*; de seguida é feito um *loop* sob o tabuleiro, de modo a verificar a existência de galos, onde as condições não correm todas as posições mas apenas as necessárias para não registar galos repetidos e poupar tempo na sua leitura. Esta poupança é útil pois este método é usado intensamente durante a função que resolve o puzzle automaticamente.

```
def do_ajuda(self, arg):
" - comando ajuda que indica a próxima casa lógica a ser jogada (sem indicar a peça
a ser colocada): ajuda  \n"
```

Através deste método, o jogador pode solicitar ajuda na resolução do puzzle. Deste modo, é pesquisada uma jogada plausível para sugerir ao jogador, através da iluminação de uma casa vazia na interface gráfica. Para este objetivo ser cumprido, o método vai iterar sob o tabuleiro com o principal objetivo de procurar uma situação, que satisfaça as condições definidas na função (de possível galo), sendo a jogada sugerida com o intuito de prevenir o galo.

```
def do_undo(self, arg):
" - comando para anular movimentos (retroceder no jogo): undo \n"
```

Permite a regressão para a jogada anterior. Faz o *pop*() do último elemento colocado numa *stack* com as jogadas realizadas e elimina essa jogada do tabuleiro, fazendo uso também dom método **do_jogar**.

```
def do_resolver(self, arg):
" - comando para resolver o puzzle: resolver \n"
```

O propósito deste método é resolver o puzzle de forma automática. Assim sendo, itera sob os vários elementos do tabuleiro, fazendo uma jogada nas casas disponíveis tentando cumprir as condições, estabelecidas, linha a linha, com especificações para cada possível localização dentro da linha, de modo a não fazer galo. Este tabuleiro é validado no fim de cada jogada pelo método **do_validar_resolver**. Caso haja galo, o programa vai iterar sobre ele e tentar resolver o conflito. Se

não for possível, uma segunda tentativa será aplicada no fim do preenchimento do tabuleiro. Quando não é possível jogar sem fazer galo, a jogada é feita tendo em conta o número de 'X' e 'O' no tabuleiro, na esperança de este ser corrigido durante a correção dos galos. Esta aproximação estilo *brute force*, revelou-se um pouco exaustiva e não muito rentável para puzzles muito grandes com um grande número de combinações possíveis. Devido a ineficiências do programa, nem todos os puzzles são possíveis de resolver desta maneira.

```
def do_ancora(self, arg):
" - comando âncora que deve guardar o ponto em que está o jogo para permitir mais
tarde voltar a este ponto: ancora \n"
```

A função do método **do_ancora** cria um ficheiro que guarda o estado do tabuleiro atual, através da criação de um ficheiro. Por razões de *design*, achamos que o *save* da ancora deve ser único, sendo que a execução de uma nova âncora apaga a anterior.

```
def do_undoancora(self, arg):
" - comando undo para voltar à última ancora registada: undoancora \n"
```

Este método tem o mesmo papel do anterior, contudo este está pré-definido para abrir uma âncora criada previamente.

```
def do_gerar(self, arg):
" - 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 \n"
```

O método **do_gerar** tem o objetivo de criar um tabuleiro com determinadas dimensões e suposta dificuldade, de acordo com os argumentos fornecidos, preenchendo-o com casas bloqueadas e algumas jogadas iniciais, de modo a que este tenha uma solução única. O tabuleiro é criado através do preenchimento de um ficheiro que cumpre os requisitos para ser lido pelo programa. As dificuldades 1 e 2 distinguem-se pela razão de 'X:O:#' criados no tabuleiro com intuito de condicionar as jogadas do utilizador, no entanto não nos foi possível programar a funcionalidade de criar um puzzle com uma solução única.

```
def do_ver(self,arg):
" - Comando para visualizar graficamente o estado atual do GandaGalo caso seja válido:
VER  \n"
```

Permite ver o puzzle no estado gráfico, graças ao uso das *packages tkinter e graphics.*

```
def do_random(self,arg):
```

Através do método **do_random**, é definida a jogada através da razão entre a soma dos diferentes símbolos. Caso a quantidade destes seja igual, a *package random* é usada para escolher aleatoriamente um dos 2 símbolos 'X,O' para ser jogado.

```
def do_sair(self, arg):
"Sair do programa GandaGalo: sair"
```

O método **do_sair** termina o programa, destruindo a instância.

Conclusão

Como retrospeção, verificamos que o nosso programa não está otimizado a 100%. Existiram dificuldades na programação de alguns dos métodos, nomeadamente **do_resolver**, **do_ajuda** e **do_gerar**.

Relativamente ao primeiro, poderia estar melhor organizado para funcionar de forma correta. Assumimos também que a forma escolhida para a resolução automática do puzzle foi muito exaustiva e desnecessariamente trabalhosa. Quanto ao segundo, sabemos também que não é tão fiável em situações mais ambíguas, como por exemplo a inexistência de galo, onde acaba por indicar apenas casas disponíveis para jogar. Sobre o último, é necessária a funcionalidade de uma solução única, a qual não fomos capazes de desenvolver com a devida consistência.

Com a inserção de mais métodos na classe *Engine* seria possível tornar o código da *Shell* mais legível e compreensivo.

Como nota final, compreendemos que este trabalho foi útil para interiorizar os conceitos teóricos lecionados nas aulas, levando também a muitas horas de prática.