Projeto IA - Takuzu

Diogo Gaspar (99207), João Rocha (99256)

Grupo 005 LEIC-A

Descrição do Problema e da Solução

Foi proposta a elaboração de um programa, em `Python`, que resolvesse, de forma eficiente o *puzzle* binário *takuzu*.

Este *puzzle* pede-nos para encontrar um tabuleiro, totalmente preenchido com 0's e 1's, partindo de uma configuração inicial, que satisfaça as seguintes restrições:

- Não podem haver 3 símbolos (0 ou 1) iguais consecutivos;
- A diferença entre o número de 1's e 0's numa dada linha ou coluna deve ser no máximo 1 (isto é, deve ser 0 em tabuleiros de tamanho par e 1 nos de tamanho ímpar);
- Todas as linhas devem ser diferentes entre si;
- Todas as colunas devem ser diferentes entre si.

No contexto da matéria de Inteligência Artificial, isto consiste em encontrar um estado **completo** e **consistente**, em que:

- Um estado é completo se não tiver células vazias;
- Um estado é consistente se satisfizer as 4 regras acima.

Chegar a um estado completo é bastante fácil, basta preencher todas as células do tabuleiro. Para garantir que atingimos um estado completo **e** consistente, basta então garantir que nunca fazemos uma jogada que nos leve a um estado inconsistente.

Chamamos uma jogada de **impossível** se a execução desta levar a um estado inconsistente (o estado quebra alguma das regras supramencionadas). Destas regras, a única cuja verificação merece alguma explicação é a da identificação entre linhas/colunas. Para garantir que isto nunca acontece, guardamos a qualquer momento (na classe `Board`) dois *sets*, cada uma guardando *strings* binárias que representam, respetivamente, as linhas e colunas que já estão totalmente preenchidas no tabuleiro. Esta solução com *strings* binárias permite aumentar consideravelmente a eficiência da verificação de igualdade entre linhas/colunas.

As jogadas podem ainda ser **possíveis** (corresponde apenas a não ser impossível) ou **obrigatórias** (se forem possíveis e o seu conjugado não for). Aqui, o conjugado de uma jogada corresponde à jogada que atua sobre a mesma célula, mas com o valor **conjugado** (isto é, se uma jogada coloca 0 na posição (1,3), a sua conjugada coloca 1 nessa mesma posição).

Em cada estado, verificamos sempre se há alguma célula vazia em que ambas as jogadas sejam impossíveis. Neste caso, qualquer jogada nessa célula levaria a um estado em que o tabuleiro é inconsistente e não vale a pena prosseguir neste ramo da árvore de procura.

Sempre que haja ações obrigatórias por realizar, num dado estado, realizamo-las. Isto é apenas lógico, visto que, por definição de obrigatoriedade, vamos ter de as executar para chegar a qualquer tabuleiro solução. Assim, antecipando a sua execução, reduzimos o número de nós da nossa árvore de procura.

Sempre que isto não é possível, escolhemos o par de ações possíveis para qualquer célula vazia do tabuleiro (note-se que uma vez que não há jogadas obrigatórias - células em que apenas uma jogada é possível - nem células em que não seja possível jogar, é necessariamente verdade que em qualquer célula vazia podemos colocar tanto um 0 como um 1).

Esta decisão de devolver sempre no máximo duas ações traduz-se em que o *branching factor* da nossa árvore é 2. Como vamos ver na análise experimental, isto é fundamental para a execução em tempo eficiente da nossa solução.

Como a nossa procura é feita de forma a nunca alcançar estados inconsistentes, basta que o nosso `goal_test` verifique se está num estado completo - um estado sem células vazias (nesse caso será uma solução).

Análise Experimental