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's ou 1's) iguais consecutivos;
- A diferença entre o número de 0's e 1's numa dada linha ou coluna deve ser no máximo 1: 0 em tabuleiros de tamanho par, 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**, onde:

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

Chegar a um estado completo é bastante fácil, bastando para tal 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.

Para nos ajudar a resolver o problema em mãos, vamos suportar-nos em algumas definições auxiliares. Dizemos que uma jogada é **impossível** se a execução desta levar a um estado inconsistente (o estado quebra alguma das regras supra-mencionadas, portanto). Destas regras, a única cuja verificação merece alguma explicação é a da identificação linhas/colunas repetidas. Para garantir que tal nunca acontece, guardamos (em `Board`) a qualquer momento dois *sets*, cada um 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: é mais eficiente comparar *strings* que tuplos, por exemplo.

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 (considerando o ramo atual da árvore de procura, claro: uma jogada obrigatória num dado ramo poderá não o ser na solução). Assim, antecipando a sua execução, **reduzimos o número de nós da nossa árvore de procura**.

Sempre que tal não é possível, escolhemos um 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 seja 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 necessariamente uma solução).

Tendo em conta a perspetiva dos CSP's (*Constraint Satisfaction Problems*) abordada em aula, temos que na nossa solução:

- A opção de devolver **no máximo duas opções**, ambas respetivas a apenas uma posição vazia, capitaliza na ideia de escolher uma variável de cada vez, visto que todas vão ter de ser escolhidas eventualmente. Como vimos em aula, isto pode reduzir o número de nós da árvore de procura em várias ordens de grandeza.
- A opção de devolver as jogadas obrigatórias sempre que possível é uma aplicação da heurística LCV (*Least Constraining Value*). De facto, ao escolhermos um valor obrigatório para a variável não estamos a impor qualquer condição ao tabuleiro que ainda não estivesse imposta (mesmo que indiretamente).

Análise Experimental

Tendo em conta a implementação descrita na página anterior, foram obtidos os seguintes resultados experimentais (para os testes públicos fornecidos pela docência):

Note-se que a coluna `Tempo de Execução (s)` corresponde à media de tempo de execução, calculada recorrendo à ferramenta `hyperfine` (com 50 execuções por procura), que a solução implementada levava a resolver os treze testes públicos fornecidos pela docência.

Aqui, não faz particular sentido optar pela BFS (em detrimento da DFS): partindo de uma configuração inicial, com n posições vazias, é claro que uma solução terá sempre de estar no nível n da árvore de procura (não podendo estar mais acima), visto que vamos sempre ter de executar n jogadas para chegar a uma solução. Assim sendo, os possíveis ganhos de uma procura em largura primeiro não são aqui sentidos.

Análise de Heurísticas