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 (o mesmo para as colunas).

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 e **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. Na **Figura 1** pode-se observar um exemplo da transição entre uma configuração inicial e um tabuleiro completo e consistente.

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, não valendo a pena, portanto, 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, 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).

Função Heurística

Na nossa escolha de função heurística, insistimos na ideia do LCV. Como a procura A* escolhe nós por ordem crescentes da função $f(n) = g(n) + h(n)$, aos valores mais constringentes devem estar atribuídos valores da heurística maiores. Para isto, calculamos um "peso" que corresponde à média de duas componentes:

- O constrangimento ao longo da linha/coluna da posição onde foi executada a última jogada, causado por essa jogada. Idealizamos que uma jogada é tão mais constringente sobre uma linha quão mais perto deixar essa linha de estar saturada do valor que acabou de ser introduzido. Assim, calculamos o constrangimento sobre uma linha tirando a proporção de valores preenchidos que estão preenchidos com o valor que acabou de ser inserido. O valor final desta componente é a média entre o constrangimento sobre a linha e o constrangimento sobre a coluna.

- O constrangimento na "vizinhança" da posição onde foi executada a última jogada, causado por essa jogada. Uma jogada é tão mais constringente sobre a sua vizinhança quantas mais jogadas se tornarem obrigatórias para prevenir 3 símbolos idênticos consecutivos. Esta componente calcula então o número média de tais jogadas originadas.

O valor final da heurística é então o peso - que corresponde à média das duas componentes acima - multiplicado pelo número de células vazias. Ao multiplicarmos pelo número de células vazias fazemos com que a heurística seja **mais dominante**. De facto, se usássemos apenas o peso, a nossa procura A* não seria muito diferente da DFS.

Observe-se que, como o valor do peso está entre 0 e 1, a nossa heurística é **admissível**, visto que a distância verdadeira ao objetivo é o número de células vazias, se estivermos num ramo que leva ao objetivo, e infinito caso contrário. Note-se no entanto que não é **consistente**. Desta forma, e uma vez que a função `astar_search` implementa uma procura em grafo, a nossa heurística não garante uma procura **ótima** - irrelevante, dado que todas as soluções estão à mesma distância do estado inicial.

Na verdade, este facto torna as procuras informadas pouco mais úteis que as procuras cegas. A procura informada acaba por servir apenas para fazer escolhas mais "educadas". Porém, como observaremos na análise experimental, isto nem sequer leva a melhores resultados.

Análise Experimental

Tendo em conta a implementação proposta, foram obtidos os resultados experimentais (para os testes públicos fornecidos pela docência) descritos na **Tabela 1**. Note-se que a coluna `Tempo de Execução (ms)` corresponde à media de tempo de execução de cada teste, calculada recorrendo à ferramenta `hyperfine` (com 250 execuções por teste, por procura).

Teoricamente, o facto de escolhermos **uma variável por nível** ajudaria de forma drástica quanto à eficiência do problema. Porém, verificamos quanto aos testes usados que, dos 13, apenas os testes 3, 5 e 6 requerem alguma decisão no caminho para uma solução. Mesmo os testes 3 e 5 exigem um número bastante limitado de decisões. Sendo assim, com a aplicação da heurística LCV, a diferença entre as várias procuras é nula em quase todos os testes e limitada em todos menos um. De facto, observa-se que nos testes 3 e 5, a não escolha de uma variável por nível leva a um aumento ligeiro no tempo e no número de nós gerados e expandidos quando se usa procuras que não a DFS. Claramente, é no entanto no teste 6 que se observa a única alteração significativa, em que o teste só executa em tempo útil (isto é, em menos de 2 minutos) quando usamos uma DFS.

A heurística **LCV** surtiu, contrastando com a utilização de 1 variável por nível, uma diferença drástica em comparação com a sua não-utilização: apenas os dois primeiros testes corriam em tempo útil, para qualquer procura, considerando também a ausência da lógica de 1 variável por nível. Estando essa lógica incluída, os resultados experimentais são os descritos na **Tabela 3**.

Conclusão

A nossa solução é, para todas as procuras, completa, visto que, na pior das hipóteses, analisa todos os tabuleiros possíveis para o tabuleiro inicial. Pode-se ainda dizer que é ótima, também para todas as procuras, já que a distância ao objetivo é predeterminada.

A heurística escolhida não leva a melhorias significativas (na verdade é pior que uma DFS), visto que este problema, por natureza não benefícia de procuras informadas. Isto verifica-se uma vez que a distância ao objetivo é predeterminada e conhecida à partida.

Na solução submetida para avaliação automática, via *Mooshak*, optámos por utilizar a versão original, com 1 variável por nível, utilizando uma procura em profundidade primeiro (por ser a mais eficiente).

Anexos

	1		0
		0	
	0		
1	1		0

0	1	1	0
1	0	0	1
0	0	1	1
1	1	0	0

Figure 1: Estados inicial e final de um possível *puzzle* takuzu consistente

	Tempo de Execução (ms)				Nós Gerados				Nós Expandidos			
Teste	BFS	DFS	A*	Gananciosa	BFS	DFS	A*	Gananciosa	BFS	DFS	A*	Gananciosa
01	73.128	72.515	73.633	73.186	7	7	7	7	7	7	7	7
02	72.843	73.416	73.337	73.053	7	7	7	7	7	7	7	7
03	81.326	81.748	82.538	82.823	43	43	43	43	43	42	43	43
04	77.715	75.694	77.806	77.450	32	32	32	32	32	32	32	32
05	85.999	85.665	86.816	86.653	59	59	59	59	59	58	59	59
06	111.490	110.929	113.014	112.789	85	82	85	85	85	81	85	85
07	91.672	91.307	92.653	92.878	69	69	69	69	69	69	69	69
08	74.727	74.672	75.366	75.339	19	19	19	19	19	19	19	19
09	116.292	115.688	118.930	118.580	139	139	139	139	139	139	139	139
10	154.169	151.870	156.289	156.253	184	184	184	184	184	184	184	184
11	129.147	129.426	133.098	133.109	180	180	180	180	180	180	180	180
12	98.991	99.399	102.951	102.772	166	166	166	166	166	166	166	166
13	102.463	102.535	107.096	107.023	180	180	180	180	180	180	180	180

Table 1: Resultados Experimentais, 1 variável por nível.

	Tempo de Execução (ms)				Nós Gerados				Nós Expandidos			
Teste	BFS	DFS	A*	Gananciosa	BFS	DFS	A*	Gananciosa	BFS	DFS	A*	Gananciosa
01	72.970	73.094	73.127	73.297	7	7	7	7	7	7	7	7
02	73.474	73.014	73.396	73.446	7	7	7	7	7	7	7	7
03	82.725	81.771	82.583	82.932	60	49	51	51	57	42	44	44
04	76.408	76.588	77.628	77.514	32	32	32	32	32	32	32	32
05	95.799	86.219	93.721	92.753	209	80	149	140	189	57	114	104
06	-	905.868	-	-	-	9537	-	-	-	9430	-	-
07	92.043	92.060	93.145	92.655	69	69	69	69	69	69	69	69
08	74.632	74.724	74.974	75.032	19	19	19	19	19	19	19	19
09	115.589	116.284	118.702	118.603	139	139	139	139	139	139	139	139
10	152.918	151.920	156.084	156.218	184	184	184	184	184	184	184	184
11	128.591	129.511	133.295	133.087	180	180	180	180	180	180	180	180
12	98.737	99.028	102.885	102.364	166	166	166	166	166	166	166	166
13	102.898	102.658	106.931	107.385	180	180	180	180	180	180	180	180

Table 2: Resultados Experimentais, todas as ações possíveis por nível.

	Tempo de Execução (ms)				Nós Gerados				Nós Expandidos			
Teste	BFS	DFS	A*	Gananciosa	BFS	DFS	A*	Gananciosa	BFS	DFS	A*	Gananciosa
01	76.747	75.894	75.294	73.152	132	14	54	14	85	7	40	7
02	74.491	74.594	74.677	76.325	42	14	30	14	27	9	17	7
03	-	88.132	9088.461	148.721	-	108	13750	464	-	76	9726	320
04	-	82.285	32412.935	91.285	-	64	23928	134	-	43	14157	83
05	-	93.511	-	143.983	-	110	-	412	-	66	-	258
06	-	130.922	-	-	-	242	-	-	-	179	-	-
07	-	604.342	-	4702.732	-	1800	-	8874	-	1758	-	6717
08	-	81.294	-	84.485	-	38	-	80	-	21	-	48
09	-	-	-	-	-	-	-	-	-	-	-	-
10	-	-	-	-	-	-	-	-	-	-	-	-
11	-	-	-	-	-	-	-	-	-	-	-	-
12	-	-	-	-	-	-	-	-	-	-	-	-
13	-	-	-	-	-	-	-	-	-	-	-	-

Table 3: Resultados Experimentais, 1 variável por nível, sem LCV.