Projeto 1: SAT em Z3

João Miguel Faria Rui Breda Perdigoto

5 de novembro de 2021

1 Sudoku

1.1 Descrição do problema

Este exercício do projeto consistiu na modelação e redução do problema do sudoku para SAT. Este problema consiste no seguinte: dado um tabuleiro inicial 9x9, preencher todas as células do tabuleiro com números de 1 a 9 tal que nenhum número se repita na mesma linha, coluna e região (quadrado 3x3).

1.2 Símbolos Proposicionais

Uma abordagem à resolução deste problema através do SAT-solver z3 implica que este seja descrito como fórmulas de símbolos proposicionais em CNF (Conjunctive Normal Form). Fórmulas estas que serão testadas face diversas atribuições aos símbolos proposicionais.

Uma vez que se pretende explicitar que nenhuma célula pode ter um número já presente na mesma linha, coluna e região, o número de símbolos proposicionais não poderia ser inferior a 9×9. Como o domínio onde trabalhamos é booleano, cada número que as células podem ter terá que ser um símbolo proposicional, com o valor verdadeiro se o número se encontra na célula e falso caso contrário.

Portanto, haverá no total um símbolo por cada célula e por cada número possível, resultando em $9{\times}9{\times}9$ símbolos. Assim, os símbolos terão a forma $p_{i,j,n}$, $0 \leq i, j \leq 8$ e $1 \leq n \leq 9$ com i e j os índices da célula no tabuleiro e n um número possível de 1 a 9.

1.3 Condições e restrições

Estabelecidos os símbolos proposicionais, é necessário descrever o problema em termos de condições e restrições às quais este será sujeito. Como foi referido anteriormente, cada célula necessita de ter um e um só número. Esta condição pode ser dividida em duas:

- Cada célula tem pelo menos um número, ou seja, os símbolos da respetiva célula e dos diversos números não podem ser todos simultaneamente falsos.

$$\bigwedge_{0\leq i,j<9} \bigvee_{1\leq n\leq 9} p_{i,j,n} \tag{1}$$

- Cada célula tem no máximo um número. A presença desta condição é justificada pelo facto de, se houver mais do que um número por célula, os restantes números da linha, coluna e região serão influenciados.

$$\bigwedge_{0\leq i,j<9} \bigwedge_{1\leq k<m<9} \neg p_{i,j,k} \vee \neg p_{i,j,m} \tag{2}$$

- Feitas as condições para as células, é necessário implementar as restrições que ditam as regras do puzzle. As seguintes restrições indicam que uma linha e uma coluna não podem ter um número repetido, com a única diferença entre as duas ser os índices de p.

$$\bigwedge_{0\leq i<9} \bigwedge_{1\leq k\leq 9} \bigwedge_{0\leq j<m<9} \neg p_{i,j,k} \vee \neg p_{i,m,k} \tag{3}$$

$$\bigwedge_{0\leq i<9} \bigwedge_{1\leq k<9} \bigwedge_{0\leq j<m<9} \neg p_{j,i,k} \wedge \neg p_{m,i,k} \tag{4}$$

- Adicionalmente a não haver repetições de números por linha e coluna, é preciso garantir que nenhum número se repete por região. Portanto, dado um determinado $p_{i,j,k}$, os símbolos da mesma região que não estão na mesma linha e coluna (não foram abrangidos pelas condições anteriores) não poderão ter o mesmo número. Nesta expressão, c é o índice da primeira coluna da região e $b = i + (2 - i \bmod 3)$, que representa o índice da última linha da região.

$$\bigwedge_{0\leq i<9} \bigwedge_{1\leq k\leq 9} \bigwedge_{0\leq j<9} \bigwedge_{i+1\leq n<b} \bigwedge_{c\leq l<c+3} \neg p_{i,j,k} \vee \neg p_{n,l,k} \tag{5}$$

- Por fim, é necessário adicionar como condição os números já presentes no tabuleiro inicial que se pretende resolver. Assim, dado um conjunto S com os símbolos proposicionais correspondentes aos números das células do tabuleiro inicial:

$$\bigwedge_{p \in S} p \tag{6}$$

- Em função de se poder "eliminar"soluções específicas, como é o caso da função `well_posed(P)`, estabelece-se a seguinte restrição: dado um conjunto M com os símbolos de uma solução (ou seja com valor verdadeiro):

$$\bigvee_{p \in M} \neg p \tag{7}$$

1.4 Funções

A implementação recorreu a três funções principais: `sudoku`, `well_posed` e `generate`.

A função `sudoku` gera os símbolos proposicionais, aplica as restrições indicadas e verifica se existe solução.

A função `well_posed(P)` corre `sudoku(P)` para resolver o tabuleiro P e gera o conjunto de símbolos proposicionais da resposta. De seguida, corre `sudoku(P)` outra vez, que agora aplica a restrição mencionada anteriormente. Caso a verificação resulte em unsat, então o tabuleiro possui apenas uma solução, pelo que é `well_posed`.

A função `generate(S, pat)` remove os elementos de S de acordo com o padrão e chama a função `well_posed` com o resultado.

1.5 Execução do programa

Apresentamos agora exemplos da execução das funções mencionadas com o propósito de mostrar a sua correção.

Ao correr a função `well_posed` com o tabuleiro inicial do exemplo anterior leva a `unsat`, o que indica que este é `well_posed`, ou seja, possui apenas uma solução. Correndo `well_posed` com um tabuleiro inicial mais benevolente como o que se apresenta leva a que este possua múltiplas soluções, pelo que não é `well_posed`.

Ao correr a função `generate` com o tabuleiro inicial do exemplo do sudoku e com o padrão `pat1` declarado na função `main` leva a um puzzle que não é `well_posed`. Correndo com `pat2` leva um problema `well_posed`.

Figura 1: Tabuleiro inicial

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Figura 2: Puzzle resolvido após `sudoku(P)`

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figura 3: Tabuleiro inicial de `well_posed`

1						7		
		2						
					9			
				2				
		4	3					
							6	
	9				8			

A execução de qualquer função é realizada após a sua chamada na função `main` onde podem ser passados como argumentos os tabuleiros e padrões declarados na mesma. Por defeito a função `main` chama uma função de cada tipo com o tabuleiro do exemplo do `sudoku(P1)`. Para testar com outros valores basta chamar a função no `main` com os argumentos pretendidos.

1.6 Variantes

Nas variantes aplicamos pura e simplesmente as restrições usuais, adicionando condições sobre coisas próximas, já que reparámos em paralelos entre as variantes todas — em geral, exigem que as células vizinhas (determinadas por `deltas_dist`) sejam diferentes da atual após aplicar `deltas_vals`.

Como descrevem os comentários no código mais pormenorizadamente, para `deltas_vals=[0]` isto é só verificar que as células vizinhas são `diferentes` da atual.

Notação 1.1. Aqui consideramos sempre $i, j, n \in \mathbb{N} \cap [1, 9]$. Para clareza, em vez de $p_{i,j,k}$ escrevemos $c_{i,j} = k$. Como temos as restrições usuais em vigor,

$$c_{i,j} = k \wedge c_{i,j} = k' \implies k = k' \tag{8}$$

por exemplo. Ou seja, neste contexto o símbolo de igualdade é uma notação útil que já não nos induzirá em erro.

Definição 1.2.

$$\texttt{diferentes}(\Delta, V) := \bigwedge_{i,j,n} \bigwedge_{\substack{c' \in \Delta +' c_{i,j} \\ \nu \in V}} (c_{i,j} = n \implies c' \neq n + \nu) \tag{9}$$

onde

$$\delta = (\delta_1, \delta_2) \in \Delta \tag{10}$$

$$\Delta +' c = \{c_{(i,j)+\delta} : \delta \in \Delta\} \cap \{1, \cdots, 9\}^2 \tag{11}$$

para garantir que não saímos do tabuleiro.

Isto é suficiente para resolver o problema proposto, como se pode verificar nos exemplos dados no código.

2 SUMS

2.1 Descrição do problema

Queremos reduzir a questão de, dado $R \subseteq \mathbb{N}$ e $t \in \mathbb{N}$, decidir se $\exists S \subseteq R : \sum S = t$ (para R e t fixos mas arbitrários) para um problema de satisfação booleana. Mas este caso é mais bicudo do que o sudoku...

2.2 Observações sobre a Complexidade

Para qualquer um dos problemas, há três complexidades que nos interessam: a original, a final e a da redução. `SUMS` está claramente em **NP**. Como `SAT` é **NP**-completo, também sabemos que existe uma redução *polinomial* de `SUMS` para `SAT`. O nosso problema aqui, problema esse que no Sudoku não surge, é que há duas reduções óbvias — uma com muitos ($2^{\#R}$) e outra com poucos ($\#R$) símbolos proposicionais, mas ambas exponenciais:

- p_r para cada $r \in R$. Neste caso, encontrar $S = \{r_{i_1}, \cdots r_{i_n}\}$ é o mesmo que encontrar uma valoração de $p_{r_{i_j}}$ tal que $\sum_j r_{i_j} = t$. Mas para definir assim este tipo de restrições é, tanto quanto sabemos, necessário encontrar essa valoração, como nos mostra `sums_red_exp`.

- p_S para cada $S \subseteq R$, em que p_S é verdade *sse* S satisfaz o pedido. Aqui a explosão exponencial acontece logo no número de proposições a serem analisadas.

Dado que o melhor algoritmo que temos para `SAT` é (no geral) exponencial, não queremos encontrar uma codificação de `SUMS` em `SAT` que seja exponencial também, porque à partida isso dá uma exponencial dupla. Há também uma questão mais relevante, que é não respeitar o espírito da redução de problemas: queremos que a redução seja polinomial.

2.3 Símbolos proposicionais e restrições

Notação 2.1. Nesta secção tomamos sempre

$$u, w \in R \tag{12}$$

$$r, s \in \mathbb{N} \cap [0, t] \tag{13}$$

Surge então a ideia de fazer $p_{u,r}$ em que u é o último dígito lido e r representa quanto falta para chegarmos a t. Isto dá-nos $\#R \times (t+1)$ símbolos proposicionais. Temos que ter cuidado se a notação indica que o u já foi contabilizado em r ou não — dizendo não, obtemos:[1]

$$\bigwedge_{r \neq 0} \bigwedge_{u} \left(p_{u,r} \implies \bigvee_{w} p_{w,r-w} \right) \tag{14}$$

significando que qualquer resto r que tenhamos, há de haver um símbolo w que podemos ler para nos aproximarmos de t por mais w unidades (pelo que o novo resto será $r - w$).

[1] O cuidado $r \neq 0$ é necessário: lembremo-nos que $\bigvee \varnothing = \bot$, e certamente não queremos que $p_{u,0} \implies \bot$ para qualquer u.

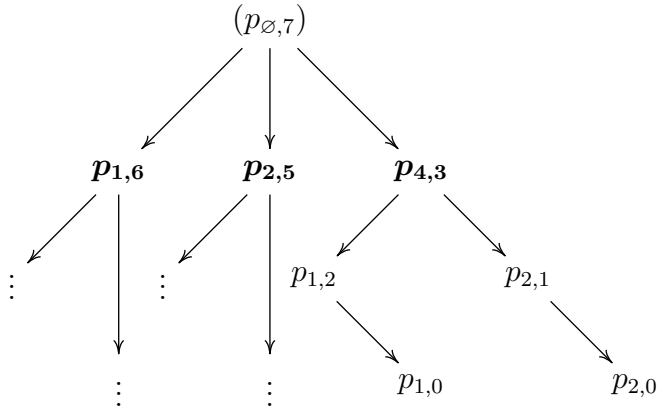

Figura 4: Exemplo de uma árvore a partir de $R = \{1, 2, 4\}$ e $t = 7$.

Esta primeira restrição dá-nos uma noção visual das proposições organizadas numa árvore, em que começamos sem ter lido nada e faltando tudo, gradualmente adicionando ramos para cada símbolo possível a ler. Poderíamos ter uma proposição extra ($p_{\varnothing,r}$); em vez disso temos:

$$\bigvee_{u} p_{u,t} \tag{15}$$

Queremos podar ao máximo a árvore: afinal de contas, queremos uma única solução, um caminho de algum $p_{u,t}$ até algum $p_{w,0}$. Isto resulta em certas condição de unicidade, até certo ponto. Primeiro exigimos existência de solução:

$$\bigvee_{u} p_{u,0} \tag{16}$$

Poderíamos incluir $\bigwedge_{u \neq w} p_{u,0} \to \neg p_{w,0}$, mas a condição seguinte já basta. Se nos falta r para alcançar t com u como último símbolo lido, certamente não nos falta outro valor com o mesmo símbolo lido, nem tão pouco falta o mesmo valor com outro último símbolo lido (isto proíbe permutações, por exemplo).

$$\bigwedge_{u,r} \left(p_{u,r} \implies \bigwedge_{w \neq u} \neg p_{w,r} \wedge \bigwedge_{s \neq r} \neg p_{u,s} \right) \tag{17}$$

Também é verdade que não podemos exceder t, embora esta condição não seja estritamente necessária.

$$\bigwedge_{u+r>t} \neg p_{u,r} \tag{18}$$

2.4 Um exemplo

Para o caso

$$R = \{0, 1, 2, 3, 4, 5, 8, 16, 32, 64, 500\}$$
$$t = 74$$

`sums(R,t)` resulta em:

```
[p_2_0, p_4_70, p_3_67, p_64_2, p_1_66]
[2, 4, 3, 64, 1]
```