Universidade do Minho

Mestrado Integrado em Engenharia Informática

Verificação Formal

SMT Solvers

Gonçalo Esteves
.(A85731).

18 de março de 2021

1 Matriz

Todas as resoluções desta primeira parte encontram-se dentro da pasta *Matriz*.

1.1 Exercício 1

O programa escrito encontra-se no ficheiro *matrix.c*, onde se pode verificar a decomposição dos ciclos nas atribuições necessárias.

1.2 Exercício 2

Toda a codificação lógica do programa encontra-se no ficheiro *matriz.smt2*, mais precisamente, nas primeiras 20 linhas do mesmo.

1.3 Exercício 3

Por forma a aferir a validade das afirmações que foram colocadas utilizou-se o ficheiro *matriz.smt2*, onde lhe foram adicionadas as restrições necessárias para responder a cada uma das alíneas. Por forma a garantir que as restrições de uma alínea anterior não interferiam com os resultados da seguinte, utilizaram-se as funcionalidades *push* e *pop*.

Tendo em conta que, para todas as alíneas, queríamos verificar se as afirmações eram sempre verdadeiras, foi necessário determinar se havia alguma instância em que o oposto do que as afirmações diziam se verificava. Assim, em todos os casos, adicionaram-se restrições à codificação lógica do programa por forma a verificar a satisfiabilidade da negação das afirmações.

1.3.1 Alínea a)

A intenção lógica desta afirmação é garantir que $i = j \implies \neg(M[i][j] = 3)$ se verifica sempre.

Ao determinar a satisfiabilidade do oposto da afirmação, verificamos que o modelo é *sat* e, como tal, a afirmação é refutável. Se analisarmos os modelos obtidos como exemplo, rapidamente percebemos que estes só são obtidos uma vez que não foram impostas restrições sobre o valor de i e j. Isto é, no caso de i e j serem iguais e superiores a 3, pode acontecer que M[i][j] seja igual a 3, uma vez que as modificações na matriz apenas foram aplicadas para valores de i e j compreendidos entre 1 e 3.

Se adicionarmos restrições sobre os valores que i e j podem tomar, forçando a que estes estejam entre 1 e 3, o SMT Solver dir-nos-á que o modelo é *unsat*, tornando a afirmação verdadeira.

1.3.2 Alínea b)

A intenção lógica desta afirmação é garantir que $M[i][j] = M[j][i]$ se verifica sempre que i e j estão entre 1 e 3.

Se tentarmos obter um modelo em que tal não se verifique, ou seja, M[i][j] é diferente de M[j][i] e i e j estão entre 1 e 3, o SMT Solver diz-nos que estamos perante um problema *unsat*, e como tal, a afirmação original é verdadeira.

1.3.3 Alínea c)

A intenção lógica desta afirmação é garantir que $i < j \implies M[i][j] < 6$ se verifica sempre que i e j estão entre 1 e 3.

Tal como na alínea anterior, se tentarmos obter um modelo em que tal não se verifica, o SMT Solver torna-nos a indicar que o problema é *unsat* e, portanto, a afirmação é verdadeira.

1.3.4 Alínea d)

A intenção lógica desta afirmação é garantir que $a > b \implies M[i][a] > M[i][b]$ se verifica sempre que i, a e b estão entre 1 e 3.

Mais uma vez, ou verificar a satisfiabilidade do problema quando se pede que esta afirmação não se verifique, deparamo-nos com a resposta de que o mesmo é *unsat* sendo, por isso, esta afirmação também verdadeira.

1.3.5 Alínea e)

A intenção lógica desta afirmação é garantir que $M[i][j] + M[i+1][j+1] = M[i+1][j] + M[i][j+1]$ se verifica sempre que i e j estão entre 1 e 3.

Ao verificarmos a existência de algum modelo em que tal não se verifica, o SMT Solver indica-nos que o problema é *sat*, tornando a afirmação refutável. Analisando o modelo calculado, percebemos porque: uma vez que as restrições sobre i e j indicam que estes devem estar entre 1 e 3, inclusive, quando pelo menos um deles é 3, iremos aceder a posições de memória que não foram alteradas (por exemplo, quando i é igual a 3, iremos aceder ao valor na matriz correspondente a M[4][3], que não foi modificado pelo programa). Uma vez que estas posições da matriz não foram modificadas no nosso programa, torna-se impossível garantir que a afirmação é sempre verdadeira.

No entanto, se restringirmos os valores de i e j a serem 1 ou 2, deixamos de aceder a posições não modificadas da matriz, o que faz com que a negação da afirmação gere um problema *unsat*, validando a assertividade da afirmação.

2 Puzzle Solver

Tendo em vista o desenvolvimento desta parte da ficha, foi criado um programa para resolver *puzzles* do género **MathemaGrids**. O dito programa é um *Jupyter Notebook*, que recorre à biblioteca *Z3* por forma a resolver os problemas colocados.

O programa desenvolvido, bem como alguns exemplos, encontram-se na pasta *Puzzle Solver*.

2.1 Funcionamento do Programa

Numa primeira instância, o programa recebe o nome do ficheiro onde se encontra o *puzzle* que pretendemos que seja resolvido. De modo a garantir o correto funcionamento do programa, o ficheiro deverá ser semelhante ao exemplo que se segue:

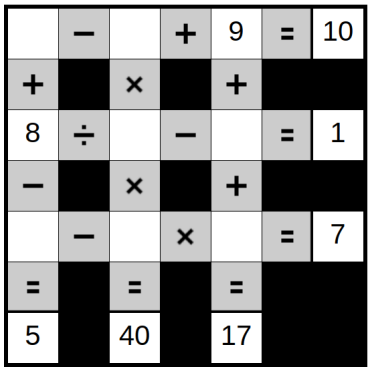

Figura 1: Exemplo de um *puzzle* cuja solução pretendemos obter

```
1  #-#+9 10
2  + * +
3  8/#-# 1
4  - * +
5  #-#*# 7
6  5 40 17
```

Figura 2: Codificação do *puzzle* num ficheiro de texto

Em seguida, o programa processa o ficheiro linha a linha, fazendo uma divisão da mesma tendo em conta os espaços existentes. Para além disto e, considerando a posição da linha que está a processar, guarda a informação nela contida de diferentes formas:

- No caso das linhas pares (consideremos a primeira linha como linha 0), a primeira *sub-string* (obtida após a separação da linha tendo em conta os espaços) irá ser percorrida a caractér a caractér, uma vez que os caracteres em posições pares (considerando o primeiro caractér como caractér 0) serão possíveis pistas (ou simplesmente valores a descobrir), enquanto que aqueles em posições ímpares serão os operadores a utilizar na expressão matemática dessa linha; a segunda *sub-string* representa o valor esperado após o cálculo da expressão;
- No caso das linhas ímpares, cada uma das *sub-strings* originadas pela divisão da linhas nas posições onde ocorrem espaços irá representar um operador a utilizar numa expressão matemática (as expressões verticais);

- No caso particular da última linha, sabe-se que cada *sub-string* irá representar os resultado final de cada uma das expressões matemáticas verticais.

Após o processamento do ficheiro, procede-se à geração do *Solver* que irá tentar descobrir um modelo que valide o problema.

Para isso, o programa começa por criar 9 variáveis, uma para cada um dos valores a descobrir, e impõem-se logo três tipos de restrições sobre estas: primeiro, o valor de todas elas deverá estar compreendido entre 1 e 9, inclusive; segundo, no caso de ser dada uma pista para uma dada variável, assume-se logo o valor da variável como sendo igual ao da pista dada; terceiro, os valores das variáveis deverão ser distintos uns dos outros.

Estando definidas estas restrições, que são gerais a todos os *puzzles* deste tipo, procede-se para a definição das restrições específicas a cada *puzzle*, as expressões matemáticas. Recorrendo a um dicionário auxiliar, que à representação de uma operação em formato *string* ('+', '-', '*' e '/') faz corresponder a implementação da operação em *Python*, torna-se fácil definir as expressões matemáticas, uma vez que a biblioteca *Z3* permite o uso das mesmas na definição de restrições para o *Solver*.

Por último, definem-se também as restrições que não permitem a ocorrência de multiplicações e divisões por 1, nem a existência de valores negativos ou fracionários nos cálculos intermédios.

No fim, e no caso de haver uma solução admissível, o programa gera um ficheiro semelhante ao passado como *input*, contendo a solução para o *puzzle*. Em seguida pode-se ver a solução obtida para o exemplo acima observado.

```
1   3-2+9 10
2   + * +
3   8/4-1 1
4   - * +
5   6-5*7 7
6   5 40 17
```

Figura 3: Solução do *puzzle* demonstrado como exemplo anteriormente