Todos já conhecem sudoku – um passatempo que consiste em completar uma coleção de números seguindo regras simples. Neste artigo veremos como, apesar de o passatempo ter regras simples,c é complicado escrever programas para resolvê-los. Veremos também como é possível usar técnicas e APIs de Programação Restrita (Constraint Programming) para resolver este tipo de problemas com um esforço muito menor.

Rafael Santos
(rafael.santos@lac.inpe.br) é doutor em Inteligência Artificial pelo Instituto Tecnológico de Kyushu, Japão. É tecnologista do Instituto Nacional de Pesquisas Espaciais, atuando em pesquisa e desenvolvimento de aplicações e técnicas de mineração de dados, processamento de imagens e inteligência artificial aplicada, é pesquisador visitante da Universidade Johns Hopkins, nos Estados Unidos. É autor do livro "Introdução à Programação Orientada a Objetos usando Java" e de várias palestras e tutoriais sobre Java.

Entendendo Programação com Restrições

Através da Resolução de Sudokus

Técnicas e APIs de Programação Restrita (Constraint Programming) para resolver sudokus e outros passatempos semelhantes.

Sudoku é um tipo de passatempo lógico que consiste no posicionamento de números em uma grade de células seguindo um conjunto de regras simples. Na sua forma mais tradicional, a grade de células tem nove linhas por nove colunas, com grades menores ou blocos de três por três células distribuídos de forma regular. Cada célula em branco deve ser preenchida por um número entre 1 e 9. Para resolver o passatempo, o leitor recebe algumas dicas na forma de células já preenchidas. As regras são simples: os números devem aparecer uma vez em cada linha, coluna ou bloco; portanto, não podem ser repetidos nas linhas, colunas e blocos.

A figura 1 mostra um exemplo de sudoku. O exemplo nesta figura é do sudoku mais comum: nove linhas por nove colunas, com cada grade menor ou sub-bloco contendo nove células em um arranjo 3x3.

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

Figura 1. Sudoku regular de 9x9 células.

Existem várias técnicas para a solução de sudokus. Uma técnica simples é procurar a única posição que um número pode ocupar em um bloco, linha ou coluna. No exemplo da figura 1, podemos observar que o bloco na

segunda linha e primeira coluna deve conter o número 5, que só pode ser posicionado na célula marcada com um círculo (a única posição que não viola as regras do passatempo). Outra técnica simples é verificar qual é o único valor que pode ocupar uma determinada célula. No exemplo mostrado na figura 1, podemos ver que a célula marcada com um quadrado só pode conter o número 9, pois todos os outros violam as regras.

Outras técnicas, algumas consideravelmente complexas, podem ser usadas para resolver qualquer sudoku considerado possível, ou seja, para o qual existe solução e somente uma solução. Para sudokus possíveis não é necessário adivinhar números: somente raciocínio lógico é necessário, embora para casos mais complicados seja necessário reavaliar várias vezes as células ainda não preenchidas para descobrir que números devem ocupá-las e usar as várias técnicas em conjunto ou alternadamente.

Outras variantes de sudoku fizeram com que o passatempo se tornasse bem popular. Duas destas variantes são mostradas nas figuras 2 e 3. O sudoku mostrado na figura 2 é conhecido também como mini-sudoku, e segue as mesmas regras do sudoku tradicional, mas contém somente quatro linhas por quatro colunas, com blocos de 2x2 células, e usa os números entre 1 e 4. A figura 2 mostra o problema original com as dicas iniciais e a solução à direita. O sudoku mostrado na figura 3 é conhecido como sudoku irregular, pois embora tenha o mesmo número de linhas e colunas, seus blocos não são quadrados ou retângulos; mas seguem as mesmas regras do sudoku regular (números não podem se repetir nas linhas, colunas e blocos). Muitas outras variantes podem ser encontradas em revistas especializadas e na internet.

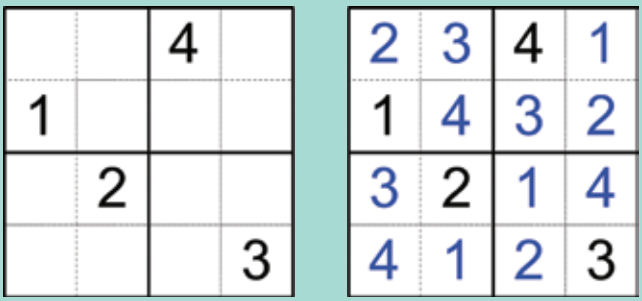

Figura 2. Mini-sudoku ou sudoku regular de 4x4 células.

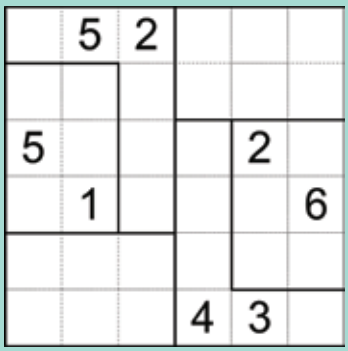

Figura 3. Sudoku irregular de 6x6 células.

O restante deste artigo descreve algumas técnicas que podem ser usadas para resolver sudokus, demonstrando a implementação de uma destas técnicas com estruturas de dados (coleções) em Java. O uso destas técnicas simples não garante a solução de qualquer sudoku, então o artigo mostra também como usar programação com restrições (usando uma API específica para isto) para resolver sudokus e outros passatempos semelhantes.

Algoritmos para a solução de sudokus

O objetivo deste artigo não é a solução manual de sudokus, mas sim a implementação de código em Java para resolver os problemas (para fãs do passatempo pode parecer um sacrilégio resolver o problema através de um programa de computador, mas desenvolvimento de algoritmos também pode ser considerado um tipo de passatempo).

Uma primeira abordagem bem ingênua seria a implementação de um algoritmo de força bruta que preenche cada célula vazia com os valores possíveis e verifica se a solução é válida. A implementação deste algoritmo seria simples, mas sua execução impraticável: para o sudoku mostrado na figura 2 (4x4 células, 4 dicas) teríamos 412 (aproximadamente 16 milhões) de soluções, das quais somente uma seria válida.

Abordagens mais inteligentes tentam eliminar soluções inválidas sem ter que calculá-las ou armazená-las, geralmente usando as próprias regras do sudoku para limitar o espaço de busca por soluções. Por exemplo, a técnica que verifica se existe um valor único para uma célula pode ser facilmente implementada, o que acelera consideravelmente a solução do problema. Esta técnica é demonstrada nas próximas listagens: primeiro na classe SudokuPorEliminacao que implementa a técnica dentro de um método main, mostrada na Listagem 1.

Listagem 1. Classe SudokuPorEliminacao, que implementa um algoritmo simples de resolução de sudokus.

```
public class SudokuPorEliminacao{
   public static void main(String[] args){
      // Cria um sudoku com as dicas mostradas na Figura 1.
      SudokuRegular s = CriaSudoku.criaSudokuFig1();
      // Cria a matriz de candidatos, uma para cada célula.
      ConjuntoInteiro[][] candidatos = new ConjuntoInteiro[9][9];
         for (int l = 0; l < 9; l++) {
            for (int c = 0; c < 9; c++) {
               candidatos[l][c] = new ConjuntoInteiro();
            }
         }
         // Laço principal. Enquanto existirem células vazias...
         while (s.contaVazios() > 0) {
            for (int l = 0; l < 9; l++) {
               for (int c = 0; c < 9; c++) {
                  // Calcula a lista de candidatos para cada célula.
                  candidatos[l][c].setTo(s.listaPossíveis(l, c));
                  // Se só houver um candidato, usa ele!
                  if (candidatos[l][c].getSize() == 1) {
                     s.põe(l, c, candidatos[l][c].getFirst());
                  }
               }
            }
         } // Fim do laço principal.
   }
}
```

A classe SudokuPorEliminacao usa algumas outras que são mostradas a seguir: a representação do sudoku e implementação de alguns métodos para verificação e manipulação dos valores é feita pela classe SudokuRegular, mostrada na Listagem 2. É importante observar que a classe SudokuRegular só permite a representação de sudokus estritamente regulares onde o tamanho de cada linha ou coluna é o quadrado do lado de cada bloco (o que acontece com sudokus regulares 4x4, 9x9, 16x16 etc.).

Listagem 2. Classe SudokuRegular, que representa um sudoku e que contém métodos para manipular seus valores.

```
import java.util.TreeSet;

// Esta classe representa um sudoku regular (com blocos quadrados).
public class SudokuRegular
  {
  private int tamanho;
  private int tamanhoBloco;
  private int[][] sudoku;
  // Inicializa campos e estruturas.
  public SudokuRegular(int t)
    {
    tamanho = t;
    tamanhoBloco = (int) Math.sqrt(tamanho);
    sudoku = new int[tamanho][tamanho];
    }

  // Põe um número em uma célula (sem verificar se pode!)
  public void põe(int linha, int coluna, int valor)
    {
    sudoku[linha][coluna] = valor;
    }

  // Recupera um número de uma célula.
  public int recupera(int linha,int coluna)
    {
    return sudoku[linha][coluna];
    }

  // Lista todos os valores que podem aparecer em uma determinada célula.
  public TreeSet<Integer> listaPossíveis(int linha, int coluna)
    {
    TreeSet<Integer> lista = new TreeSet<Integer>();
    // Adicionamos cada valor que pode ser colocado no conjunto de possíveis.
    for (int valor = 1; valor <= tamanho; valor++)
      {
      if (podePor(linha, coluna, valor))
        lista.add(valor);
      }
    return lista;
    }

  // Conta quantas células ainda estão vazias.
  public int contaVazios()
    {
    int spaces = 0;
    for (int l = 0; l < tamanho; l++)
      for (int c = 0; c < tamanho; c++)
        if (sudoku[l][c] == 0)
          spaces++;
    return spaces;
    }
// Verifica se podemos por um valor em uma célula.
  public boolean podePor(int linha, int coluna, int valor)
    {
    if (sudoku[linha][coluna] != 0)
      return false;
    // Verificamos se já existe o valor na mesma coluna.
    for (int l = 0; l < tamanho; l++)
      if (sudoku[l][coluna] == valor)
        return false;
    // Verificamos se já existe o valor na mesma linha.
    for (int c = 0; c < tamanho; c++)
      if (sudoku[linha][c] == valor)
        return false;
    // Verificamos se já existe o valor no mesmo bloco.
    int lB = (int) (linha / tamanhoBloco);
    int cB = (int) (coluna / tamanhoBloco);
    for (int l = 0; l < tamanhoBloco; l++)
      for (int c = 0; c < tamanhoBloco; c++)
        if (sudoku[l + lB * tamanhoBloco][c + cB * tamanhoBloco] == valor)
          return false;
    return true;
    }
  }
```

Outra classe necessária para a execução da classe SudokuPorEliminacao é a classe ConjuntoInteiro, que representa um conjunto de inteiros (basicamente encapsulando um TreeSet de Integers). Esta classe é mostrada na Listagem 3.

Listagem 3. Classe ConjuntoInteiro, que representa um conjunto (set) de valores inteiros.

```
import java.util.Set;
import java.util.TreeSet;

// Representa um conjunto de inteiros (um TreeSet de Integers).
public class ConjuntoInteiro
  {
  private TreeSet<Integer> conjunto;

  // Cria a instância.
  public ConjuntoInteiro()
    {
    conjunto = new TreeSet<Integer>();
    }

  // Retorna o primeiro elemento do conjunto.
  public int getFirst()
    {
    return conjunto.first();
    }

  // Inicializa com os elementos de outro conjunto.
  public void setTo(Set<Integer> s)
    {
    conjunto.clear();
    conjunto.addAll(s);
    }

  // Retorna o tamanho do conjunto.
  public int getSize()
    {
    return conjunto.size();
    }

  }
```

Finalmente, a classe CriaSudoku, mostrada na Listagem 4, possibilita a criação de um sudoku com a configuração e dicas do mostrado na figura 1.

Listagem 4. Classe CriaSudoku, que cria uma instância de SudokuRegular correspondente ao mostrado na figura 1.

```
public class CriaSudoku
  {
  // Cria o sudoku mostrado na Figura 1.
  public static SudokuRegular criaSudokuFig1()
    {
    SudokuRegular s = new SudokuRegular(9);
    s.põe(0, 0, 1); s.põe(0, 2, 3); s.põe(0, 3, 6);
    s.põe(1, 0, 7); s.põe(1, 1, 8); s.põe(1, 8, 6);
    s.põe(2, 0, 4); s.põe(2, 1, 2); s.põe(2, 2, 6);
    s.põe(2, 3, 9); s.põe(2, 5, 3); s.põe(2, 6, 7);
    s.põe(2, 8, 1); s.põe(3, 0, 9); s.põe(3, 3, 7);
    s.põe(3, 7, 5); s.põe(4, 1, 3); s.põe(4, 3, 1);
    s.põe(4, 5, 5); s.põe(4, 7, 6); s.põe(5, 1, 4);
    s.põe(5, 5, 2); s.põe(5, 8, 7); s.põe(6, 0, 5);
    s.põe(6, 2, 2); s.põe(6, 3, 8); s.põe(6, 5, 9);
    s.põe(6, 6, 1); s.põe(6, 7, 7); s.põe(6, 8, 4);
    s.põe(7, 0, 8); s.põe(7, 7, 2); s.põe(7, 8, 3);
    s.põe(8, 5, 4); s.põe(8, 6, 8); s.põe(8, 8, 5);
    return s;
    }
  }
```

O algoritmo implementado pela classe SudokuPorEliminacao (Listagem 1) consegue resolver o sudoku mostrado na figura 1 em apenas cinco iterações do laço principal, mas não é uma solução universal: para sudokus mais complexos, como o mostrado na figura 4 (conhecido como star burst leo), o algoritmo é incapaz de preencher qualquer célula vazia. Como a implementação da classe SudokuPorEliminacao é deliberadamente simples, o programa simplesmente entra em um laço eterno, por não poder chegar ao ponto de ter zero células vazias no sudoku.

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

Figura 4. Sudoku regular de 9x9 células de resolução mais complicada (star burst leo).

Poderíamos implementar outras técnicas simples para tentar preencher as células do sudoku, como, por exemplo, a técnica simples que procura determinar para uma linha, coluna ou bloco quais são as posições válidas para determinado valor. Várias outras técnicas envolvendo heurísticas, geração de soluções parciais para testes e backtracking existem, mas a implementação de seus algoritmos é bem mais complexa.

Uma alternativa para o uso de algoritmos tradicionais é tratar o problema da solução do sudoku como um problema de soluções de restrições: o problema é modelado como um conjunto de variáveis que podem assumir valores conhecidos e conjuntos de regras e condições – assim em vez de termos passos para a descoberta da solução temos a lista de propriedades da solução (ou soluções) desejada. De forma transparente para o programador, o modelo é então resolvido e as soluções (zero, uma ou várias) são disponibilizadas para o programador que usa a API.

Existem várias APIs que implementam classes e métodos para programação com restrições (constraint programming). Neste artigo usaremos a API Choco como exemplo. Instruções para a instalação da API Choco para uso com a IDE Eclipse estão no box Instalando a API Choco.

Usando a API Choco

Para dar um exemplo simples de programação com restrições e do uso da API Choco tentemos resolver um problema simples de programação com restrições. Consideremos o problema de achar todas as soluções para a equação x2+y2=z2, para x, y e z inteiros entre 1 e 100 e para x maior que y. A solução para este problema sem usar programação com restrições é trivial; basta executar um laço para cada valor de x aninhado com outro para cada valor de y maior que x e ver se o resultado de x2+y2 é um número quadrado – se for, guardamos os valores como uma possível solução.

A solução usando programação com restrições é feita de forma diferente: primeiro definimos um modelo para representar as variáveis e restrições; depois criamos um mecanismo de solução deste modelo; e finalmente percorremos as soluções encontradas por este mecanismo de solução. A Listagem 5 mostra como estes passos são implementados, usando as classes da API Choco.

Na Listagem 5 variáveis inteiras são representadas por instâncias da classe IntegerVariable, e devem ter um nome simbólico e uma faixa de valores aceitáveis. Restrições são criadas com métodos estáticos da classe Choco, e podem ser restrições de igualdade, diferença, comparação etc., criadas como instâncias de IntegerExpressionVariable. Variáveis e restrições são adicionadas ao modelo com os métodos addVariable e addConstraint.

Com o modelo criado podemos elaborar uma instância de CPSolver para resolver o problema, e com as soluções (podemos ter nenhuma, uma ou várias) podemos listar os valores que as variáveis podem assumir. Uma grande vantagem da API Choco é que o método para solução do problema de programação com restrições, assim como as estruturas de dados necessárias para implementar este método e para alcançar as soluções é definido pela própria API, embora seja possível incluir, no modelo, informações que fazem com que técnicas específicas sejam usadas pela instância de CPSolver.

Listagem 5. Classe ResolveEquacao, que demonstra algumas classes e métodos da API Choco e resolve uma equação simples.

```
import choco.Choco;
import choco.cp.model.*;
import choco.kernel.model.constraints.Constraint;
import choco.kernel.model.variables.integer.*;

public class ResolveEquacao
  {
  public static void main(String[] args)
    {
    // Criamos o modelo para o problema de programação com restrições.
    CPModel modelo = new CPModel();
    // Criamos as variáveis e seus limites, e as adicionamos ao modelo.
    IntegerVariable x = new IntegerVariable("x", 1, 100);
    IntegerVariable y = new IntegerVariable("y", 1, 100);
    IntegerVariable z = new IntegerVariable("z", 1, 100);
    modelo.addVariable(x);
    modelo.addVariable(y);
    modelo.addVariable(z);

    // Criamos as restrições através de operadores da API.
    IntegerExpressionVariable esq = Choco.plus(Choco.power(x, 2),
        Choco.power(y, 2));
    IntegerExpressionVariable dir = Choco.power(z, 2);
    // Primeira restrição: x^2+y^2 deve ser igual a z^2.
    Constraint c1 = Choco.eq(esq, dir);
    modelo.addConstraint(c1);
    // Segunda restrição: x deve ser maior que y.
    Constraint c2 = Choco.gt(x, y);
    modelo.addConstraint(c2);

    // Resolvemos o problema criando um solver.
    CPSolver solver = new CPSolver();
    solver.read(modelo);
    solver.solve();

    // Mostramos todas as soluções.
    int sol = 0;
    do
      {
      sol++;
      System.out.println("-Solução " + sol + " ----------");
      System.out.println(solver.getVar(x).getVal() + "^2+" +
                         solver.getVar(y).getVal() + "^2=" +
                         solver.getVar(z).getVal() + "^2");
    } while (solver.nextSolution());
  }
}
```

Após a solução do problema podemos recuperar os valores das variáveis a partir da instância de CPSolver, usando os nomes simbólicos das variáveis e usando um laço controlado pelo método nextSolution da classe CPSolver. A cada passo deste laço as variáveis recebem um novo conjunto de valores que satisfaz as restrições dadas pelo modelo.

A aplicação apresentada na Listagem 5 mostra todas as 52 soluções para o problema de encontrar as soluções para a equação x2+y2=z2. É interessante ressaltar que o problema pode ser facilmente resolvido com laços, como descrito anteriormente, mas se fosse preciso ter outras restrições (por exemplo, outras condições para x e y ou mesmo o uso de outras variáveis) o código que usa laços poderia se tornar complexo, com vários condicionais dentro dos laços; enquanto as modificações no código usando a API Choco seriam bem mais simples.

Resolvendo sudokus com programação com restrições

Vejamos agora como resolver sudokus usando programação com restrições e a API Choco. Os passos a serem seguidos são basicamente os mesmos dos usados para o exemplo de achar raízes de uma equação: criamos um modelo; criamos as variáveis informando seus limites, e as adicionamos ao modelo; criamos as restrições e as adicionamos ao modelo; usamos uma instância de CPSolver para resolver o problema e finalmente imprimimos os valores das variáveis encontradas como soluções.

A implementação destes passos é mostrada na Listagem 6. Nesta listagem vemos que é preciso criar uma variável para a API Choco para cada uma das células do sudoku, e as restrições são na forma de listas de células que devem ser todas diferentes (criadas com o método allDiferent da classe Choco). As dicas que são dadas no passatempo podem ser declaradas como restrições com o método eq da classe Choco, que informa que determinada célula deve ser igual a um valor constante.

Listagem 6. Classe ResolveSudoku, que mostra como resolver um sudoku com a API Choco.

```
import choco.Choco;
import choco.cp.model.CPModel;
import choco.cp.solver.CPSolver;
import choco.kernel.model.constraints.Constraint;
import choco.kernel.model.variables.integer.IntegerVariable;

public class ResolveSudoku
  {
  public static void main(String[] args)
    {
    // Criamos o modelo para o problema de programação com restrições.
    CPModel modelo = new CPModel();
    // Criamos as variáveis e seus limites, e as adicionamos ao modelo.
    // Para este exemplo é mais fácil usar uma matriz de IntegerVariables.
    int tamanho = 9;
    IntegerVariable[][] cel = new IntegerVariable[tamanho][tamanho];
    for (int linha = 0; linha < tamanho; linha++)
      for (int coluna = 0; coluna < tamanho; coluna++)
        {
        cel[coluna][linha] = Choco.makeIntVar("c"+coluna+","+linha,1,
            tamanho);
        modelo.addVariable(cel[coluna][linha]);
        }
    // Não podemos ter valores repetidos nas linhas.
    for (int c = 0; c < tamanho; c++)
      {
      Constraint con =
        Choco.allDifferent(cel[0][c],cel[1][c],cel[2][c],cel[3][c],cel[4][c],
                           cel[5][c],cel[6][c],cel[7][c],cel[8][c]);
      modelo.addConstraint(con);
      }
    // Não podemos ter valores repetidos nas colunas.
    for (int l = 0; l < tamanho; l++)
```

```
        {
        Constraint con =
            Choco.allDifferent(cel[l][0],cel[l][1],cel[l][2],cel[l][3],cel[l][4],
                               cel[l][5],cel[l][6],cel[l][7],cel[l][8]);
        modelo.addConstraint(con);
        }
    // Não podemos ter valores repetidos nos blocos.
    for (int bl = 0; bl < 3; bl++)
        for (int bc = 0; bc < 3; bc++)
            {
            Constraint con =
                Choco.allDifferent(
                cel[bl*3+0][bc*3+0],cel[bl*3+0][bc*3+1],cel[bl*3+0][bc*3+2],
                cel[bl*3+1][bc*3+0],cel[bl*3+1][bc*3+1],cel[bl*3+1][bc*3+2],
                cel[bl*3+2][bc*3+0],cel[bl*3+2][bc*3+1],cel[bl*3+2][bc*3+2]);
            modelo.addConstraint(con);
            }

    // Usamos o sudoku da Figura 1 para resolução. Precisamos usar as dicas
    // do passatempo como restrições.
    SudokuRegular s = CriaSudoku.criaSudokuFig1();
    for (int linha = 0; linha < tamanho; linha++)
        for (int coluna = 0; coluna < tamanho; coluna++)
            {
            int v = s.recupera(linha, coluna);
            // Valores iguais a zero são células vazias.
            if (v != 0) modelo.addConstraint(Choco.eq(cel[linha][coluna], v));
            }
    // Resolvemos o problema criando um solver.
    CPSolver solver = new CPSolver();
    solver.read(modelo);
    solver.solve();

    // Mostramos todas as soluções.
    int sol = 0;
    do
        {
        sol++;
        System.out.println("-Solução " + sol + " ----------");
        for (int linha = 0; linha < tamanho; linha++)
            {
            if (linha % 3 == 0) System.out.println("+---+---+---+");
            for (int coluna = 0; coluna < tamanho; coluna++)
                {
                if (coluna % 3 == 0) System.out.print("|");
                System.out.print(solver.getVar(cel[linha][coluna]).getVal());
                }
            System.out.println("|");
            }
        System.out.println("+---+---+---+");
        } while (solver.nextSolution());
    }
}
```

A solução implementada na Listagem 6 resolve rapidamente o sudoku mostrado na figura 1, e pode ser facilmente modificada para resolver também o sudoku mostrado na figura 4 – basta usar os valores das células daquele sudoku como restrições e a solução será rapidamente apresentada, demonstrando que é muito mais fácil e rápido usar programação restrita do que implementar os métodos de busca e heurísticas manualmente. O uso de programação com restrições também facilita consideravelmente a solução de sudokus irregulares como o mostrado na figura 3: basta modificar corretamente a declaração que informa ao modelo quais células pertencem a quais blocos.

O código mostrado na Listagem 6 pode ser adaptado para outros problemas semelhantes, por exemplo, para enumerar todos os sudokus que têm restrições, mas não têm solução única. Se, por exemplo, desconsiderarmos as dicas que aparecem na primeira linha do sudoku na figura 1 e executarmos o mesmo código com este sudoku modificado, a aplicação mostrará as duas soluções válidas. Se desconsiderarmos também as dicas que aparecem na última linha do sudoku da figura 1, a aplicação mostrará todas as sete soluções válidas. Se modificarmos o código mostrado na Listagem 6 para processar sudokus de 4x4 células como os mostrados na figura 2 (mas sem nenhuma dica) podemos até enumerar todos os 288 sudokus possíveis (resolvidos), mas não devemos tentar enumerar todos os sudokus com 9x9 células: o número total de sudokus regulares 9x9 foi calculado como sendo maior que seis sextilhões!

O código mostrado na figura 6 também pode ser modificado para criar sudokus para resolução manual, ao invés de resolvê-los. As modificações são relativamente simples: a partir de uma solução existente (com todas as células preenchidas) podemos executar um laço em que a cada passo removemos aleatoriamente uma célula e verificamos (através da solução deste novo sudoku) quantas soluções existem. Interrompemos o laço quando houver mais de uma solução para o sudoku com células removidas, e retornamos como resultado o último dos sudokus que apresenta somente uma solução. Esta técnica permite a criação de vários sudokus intermediários, alguns bem simples (com somente uma ou duas células removidas) até um com várias células removidas, mas ainda apresentando somente uma solução.

Bônus: resolvendo Kenken

Kenkens são passatempos similares ao sudoku: a sua solução também requer o posicionamento de números em uma grade de células quadrada, sendo que os números não podem se repetir na mesma linha ou na mesma coluna. A semelhança para aí: enquanto para resolver sudokus só precisamos de lógica, para resolver kenkens precisamos de lógica e da capacidade de fazer cálculos aritméticos simples, pois ao invés de regiões nas quais os dígitos não podem ser repetidos, temos regiões que apresentam dicas na forma de resultados e operações, sendo que a operação aplicada aos valores colocados naquela região será igual ao resultado. A explicação fica mais simples vendo um exemplo de kenken, como o mostrado na figura 5.

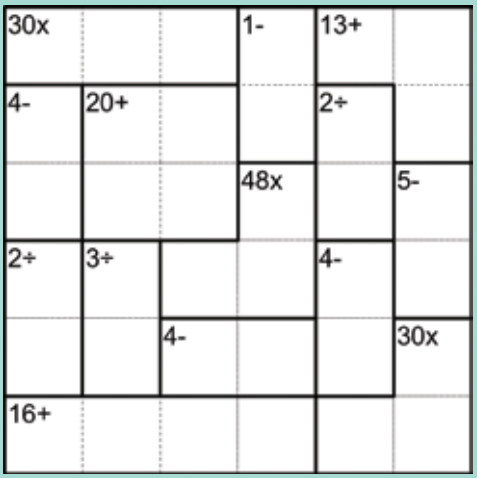

Figura 5. Exemplo de kenken.

A figura 5 mostra regiões na grade de células delimitadas por linhas mais grossas. Regiões podem ser retangulares ou irregulares, mas são sempre contíguas. Na célula mais superior e à esquerda de uma região temos um valor e uma operação, que são as dicas iniciais dadas em um kenken (excetuando as condições já conhecidas de não permitir a repetição de valores em linhas e colunas). Alguns kenkens mais simples podem dar dicas adicionais: quando uma região contém uma única célula o valor desta é colocado sem operações, assim sabemos que a célula só pode conter aquele valor. Regiões podem ter valores repetidos contanto que eles não apareçam na mesma linha ou coluna.

Assim como para o sudoku, existem variantes de kenken, algumas mais simples (usando somente soma e subtração, e grades pequenas) e mais complexas (usando as quatro operações básicas e grades maiores). Algumas variantes mais complexas (por exemplo, as chamadas Tomtom) podem usar séries não-consecutivas de valores ou mesmo restrições adicionais. Para o exemplo mostrado neste artigo, veremos somente as restrições e regras de um kenken tradicional.

A solução de kenkens é feita por fatoração e eliminação de alternativas considerando todas as restrições simultaneamente. Para dar um exemplo inicial consideremos a região na parte inferior esquerda do kenken mostrado na figura 5, e que contém quatro valores cuja soma é 16. As duas únicas opções para esta região são os conjuntos [6,5,3,2] e [6,5,4,1] (podemos rapidamente escrever uma aplicação com a API Choco que lista estas alternativas!). Por outro lado, a região irregular na parte inferior direita deve conter três valores cuja multiplicação conjunta resulta em 30: as únicas alternativas são os conjuntos [6,5,1] e [5,3,2]. Ao considerarmos estas duas regiões em conjunto podemos concluir que a primeira não pode conter o conjunto [6,5,3,2], pois seria impossível posicionar qualquer uma das duas soluções para a região cuja multiplicação dos valores é igual a 30 – podemos concluir então que os valores da primeira região (de soma 16) só podem ser [6,5,4,1], e que os dois valores à direita na última linha só podem ser [3,2] e que o valor na última coluna, penúltima linha só pode ser 5.

Achamos o primeiro valor, mas ainda existem muitas outras regras para serem avaliadas – é importante entender que descobrimos os valores que devem aparecer na região cuja soma é 16, mas não a ordem dos mesmos! Esta será descoberta avaliando outras regras e condicionais.

A implementação de um algoritmo com regras para resolver um kenken seria bem mais complexa do que para um sudoku, em parte porque cada kenken pode ter regiões com operações e formas diferentes, além de poder apresentar regiões irregulares – isto seria traduzido em uma série de condicionais no código. Se usarmos programação com restrições, a implementação será bem mais simples: parte do código será bem semelhante ao já usado para resolver sudokus (para declarar as variáveis e as restrições de valores iguais nas linhas e colunas, por exemplo); e parte seria semelhante à mostrada para a resolução de equações simples. O código para resolver o kenken da figura 5 é mostrado na Listagem 7.

Listagem 7. Classe ResolveKenken, que mostra como resolver um kenken com a API Choco.

```
import choco.Choco;
import choco.cp.model.CPModel;
import choco.cp.solver.CPSolver;
import choco.kernel.model.constraints.Constraint;
import choco.kernel.model.variables.integer.*;

public class ResolveKenken
  {
  public static void main(String[] args)
    {
    // Criamos o modelo para o problema de programação com restrições.
    CPModel modelo = new CPModel();
    // Criamos as variáveis e seus limites, e as adicionamos ao modelo.
    // Para este exemplo é mais fácil usar uma matriz de IntegerVariables.
    int tamanho = 6;
    IntegerVariable[][] cel = new IntegerVariable[tamanho][tamanho];
    for (int linha = 0; linha < tamanho; linha++)
      for (int coluna = 0; coluna < tamanho; coluna++)
        {
        cel[coluna][linha] = Choco.makeIntVar("c"+coluna+","+linha,1,
            tamanho);
        modelo.addVariable(cel[coluna][linha]);
        }
    // Não podemos ter valores repetidos nas linhas.
    for (int c = 0; c < tamanho; c++)
      {
      Constraint con =
        Choco.allDifferent(cel[0][c],cel[1][c],cel[2][c],cel[3][c],cel[4][c],cel[5][c]);
      modelo.addConstraint(con);
      }
    // Não podemos ter valores repetidos nas colunas.
    for (int l = 0; l < tamanho; l++)
      {
      Constraint con =
        Choco.allDifferent(cel[l][0],cel[l][1],cel[l][2],cel[l][3],cel[l][4],cel[l][5]);
      modelo.addConstraint(con);
      }
    // Blocos devem ser especificados de forma diferente do Sudoku, com
    // expressões e operações.
    IntegerExpressionVariable iev1,iev2;
    iev1 = Choco.mult(cel[0][0],Choco.mult(cel[0][1],cel[0][2]));
    modelo.addConstraint(Choco.eq(iev1,30)); // 30x
      iev1 = Choco.minus(cel[0][3],cel[1][3]);
    modelo.addConstraint(Choco.eq(Choco.abs(iev1),1)); // 1- (usamos abs!)
    iev1 = Choco.sum(cel[0][4],cel[0][5],cel[1][5]);
    modelo.addConstraint(Choco.eq(iev1,13)); // 13+
    iev1 = Choco.minus(cel[1][0],cel[2][0]);
    modelo.addConstraint(Choco.eq(Choco.abs(iev1),4)); // 4- (usamos abs!)
    iev1 = Choco.sum(cel[1][1],cel[1][2],cel[2][1],cel[2][2]);
    modelo.addConstraint(Choco.eq(iev1,20)); // 20+
    iev1 = Choco.div(cel[1][4],cel[2][4]);
    iev2 = Choco.div(cel[2][4],cel[1][4]);
    modelo.addConstraint(Choco.or(Choco.eq(iev1,2),Choco.eq(iev2,2))); // 2÷
    iev1 = Choco.mult(cel[2][3],Choco.mult(cel[3][2],cel[3][3]));
    modelo.addConstraint(Choco.eq(iev1,48)); // 48x
    iev1 = Choco.minus(cel[2][5],cel[3][5]);
    modelo.addConstraint(Choco.eq(Choco.abs(iev1),5)); // 5- (usamos abs!)
    iev1 = Choco.div(cel[3][0],cel[4][0]);
    iev2 = Choco.div(cel[4][0],cel[3][0]);
    modelo.addConstraint(Choco.or(Choco.eq(iev1,2),Choco.eq(iev2,2))); // 2÷
    iev1 = Choco.div(cel[3][1],cel[4][1]);
    iev2 = Choco.div(cel[4][1],cel[3][1]);
    modelo.addConstraint(Choco.or(Choco.eq(iev1,3),Choco.eq(iev2,3))); // 3÷
    iev1 = Choco.minus(cel[3][4],cel[4][4]);
    modelo.addConstraint(Choco.eq(Choco.abs(iev1),4)); // 4- (usamos abs!)
    iev1 = Choco.minus(cel[4][2],cel[4][3]);
    modelo.addConstraint(Choco.eq(Choco.abs(iev1),4)); // 4- (usamos abs!)
    iev1 = Choco.mult(cel[5][4],Choco.mult(cel[4][5],cel[5][5]));
    modelo.addConstraint(Choco.eq(iev1,30)); // 30x
    iev1 = Choco.sum(cel[5][0],cel[5][1],cel[5][2],cel[5][3]);
    modelo.addConstraint(Choco.eq(iev1,16)); // 16+
    // Resolvemos o problema criando um solver.
    CPSolver solver = new CPSolver();
    solver.read(modelo);
    solver.solve();


    // Mostramos todas as soluções.
    int sol = 0;
    do
      {
      sol++;
      System.out.println("-Solução " + sol + " ----------");
      System.out.println("+------+");
      for (int linha = 0; linha < tamanho; linha++)
        {
        System.out.print("|");
        for (int coluna = 0; coluna < tamanho; coluna++)
          {
            System.out.print(solver.getVar(cel[linha][coluna]).getVal());
          }
          System.out.println("|");
        }
      System.out.println("+------+");
      } while (solver.nextSolution());
    }
  }
```

A parte mais complexa da Listagem 7 é a que define as restrições que correspondem às regiões: cada restrição contém as variáveis da região e o valor esperado quando aplicamos a operação da região nos valores. Para algumas operações, como soma e multiplicação, a declaração da restrição é trivial: por exemplo, para declarar a restrição correspondente à região no canto superior esquerdo da figura 6 podemos escrever algo como iev1 = Choco.mult(cel[0][0],Choco.mult(cel[0][1],cel[0][2])); modelo.addConstraint(Choco.eq(iev1,30));. Operações não comutativas exigem um pouco mais de cuidado: a região que mostra a dica 5- nos leva a assumir que os valores da região são [6,1] ou [1,6], mas como declaramos os valores da célula sem saber qual é o maior ou qual é o menor devemos considerar o valor absoluto da diferença entre as células, assim a restrição é adicionada ao modelo como Choco.eq(Choco.abs(iev1),4));

Para regiões cuja dica é uma divisão, a restrição deve ser declarada de forma diferente: como não temos um operador para o valor absoluto da divisão (isto é, que retorna valores iguais para 1/3 ou 3/1) teremos que combinar as restrições com um operador lógico. Usamos esta abordagem para declarar as restrições que representam a região cuja dica é 3÷, com o trecho de código a seguir.

```
iev1 = Choco.div(cel[3][1],cel[4][1]);

iev2 = Choco.div(cel[4][1],cel[3][1]);

modelo.addConstraint(Choco.or(Choco.
eq(iev1,3),Choco.eq(iev2,3)));
```

O código mostrado na Listagem 7 resolve o kenken apresentado na figura 6 quase instantaneamente, e mostra a única solução correta.

Instalando a API Choco

A API Choco permite o uso de classes e métodos de programação com restrições em código em Java de forma simples e eficiente. Arquivos relacionados com esta API (.jar, tutoriais, documentação) podem ser baixados de http://www.emn.fr/z-info/choco-solver/. Para que possamos usar a API Choco em aplicações em Java devemos referenciar seu arquivo .jar (a versão mais recente em maio de 2011 é choco-2.1.1). No Eclipse isto pode ser feito clicando no nome do projeto com o botão direito do mouse, selecionando o menu Build Path / Add External Archives e escolhendo o nome do arquivo .jar no diálogo de seleção.

Considerações finais

Em muitas situações podemos tentar resolver problemas através de busca exaustivas com algumas restrições para evitar a busca em um espaço muito grande de soluções: passatempos como sudoku e kenken são somente alguns exemplos mais triviais e divertidos, outros são relacionados com alocação de recursos limitados e/ou restritos, como, por exemplo, criação de grades de horários e organização de pessoas em times ou empacotamento de objetos em espaços restritos.

Programação com restrições pode ser feita de forma complexa e trabalhosa usando laços, condicionais e estruturas de dados especializadas, mas este esforço pode ser consideravelmente simplificado com uma API de programação com restrições, como a API Choco, demonstrada neste artigo.

Referências

- ***Sudokus para a solução "manual" (com caneta ou lápis) podem ser encontrados em vários jornais e revistas específicas. No Brasil ainda não temos publicações regulares de kenken, mas livros só com passatempos deste tipo podem ser comprados em livrarias no exterior como Amazon.com – em particular recomendo os do autor Will Shortz, que tem coleções de kenkens desde iniciantes até os mais complexas; ou o livro KenKen for Dummies, de Tetsuya Miyamoto, inventor do passatempo. Para os que gostam de desafios ainda mais complexos, sugiro o livro TomTom Puzzles, de Thomas Snyder.***
- ***O site da editora Conceptis (www.conceptispuzzles.com) mostra vários tipos de sudokus e kenkens (chamados pela editora de calcudokus) que podem ser resolvidos on-line.***
- ***Um único livro contém abordagens algorítmicas (de implementação direta, ao invés de usando outras técnicas como programação com restrições) para a solução de sudokus: Programming Sudoku, de Mei-Weng Lee. Este livro não contém todos os algoritmos e heurísticas para solução dos sudokus, mas mostra como implementar as mais básicas. Mais informações sobre técnicas para resolver sudokus estão na página http://en.wikipedia.org/wiki/Sudoku_algorithms.***
- ***Muitos exemplos de uso da API Choco (inclusive outras formas de resolver sudokus) estão contidos nas páginas do projeto em http://www.emn.fr/z-info/choco-solver/index.html .***