A practical guide for squeezing every drop of performance out of your Java apps

Level: Intermediate

Erwin Vervaet (erwin@ervacon.com), Senior Software Engineer, Ervacon

Maarten De Cock (maarten.decock@asq.be), Application Engineer, ASQdotCOM

June 2002

Many useful techniques exist for optimizing a Java program. Instead of focusing on one particular technique, this article considers the optimization process as a whole. Authors Erwin Vervaet and Maarten De Cock walk readers through the performance tuning of a puzzle-solving program, applying an assortment of techniques ranging from simple technical tips to more advanced algorithm optimizations. The end result is a spectacular performance increase (more than a *million* fold) between the first working implementation and the fully optimized solution.

Most Java performance-related articles focus on the many techniques that programmers can employ to speed up their programs. At one end of the spectrum you can find descriptions of relatively simple programming idioms, like the use of the `StringBuffer` class. At the other end you find discussions of more advanced techniques, like the use of object caches. Instead of adding to this list of techniques, we'll present a practical example that combines them to speed up a puzzle-solving program.

The program we will develop and optimize calculates all possible solutions for the Meteor puzzle, a brain teaser consisting of 10 puzzle pieces, each a different color made up of five hexagons (six-sided polygons with each side of equal length). The puzzle board itself is a rectangular grid of 50 hexagons laid out in a 5-by-10 pattern. You solve the puzzle by covering the entire board using the 10 available pieces. A possible solution to this puzzle is shown in Figure 1.

Figure 1. A solution for the Meteor puzzle

The Eternity puzzle

While the Java program discussed in this article solves the 10-piece Meteor puzzle, the real goal was to solve a much larger 209-piece puzzle called the Eternity puzzle, devised by Christopher Monckton and introduced in Britain in June 1999. At the same time that Eternity was released, Monckton released

Contents:

Related content:

Merlin brings nonblocking I/O to the Java platform

Improve the performance of your Java code

Subscribe to the developerWorks newsletter

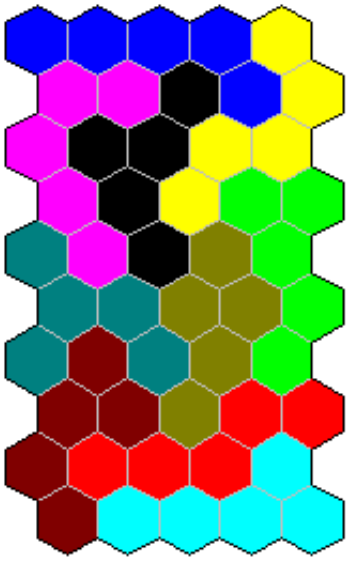
Also in the Java zone:

Tutorials

Tools and products

Code and components

Articles



As simple as finding this solution might seem, it is a non-trivial problem to implement in a computer program. Writing that program will be a refreshing change from the contrived examples you can find in many other Java performance-related articles. It allows us to illustrate a number of different optimization techniques and the ways of combining them. However, before we start optimizing, we first need to develop a working solution.

> various smaller puzzles: Meteor, Delta, and Heart. By solving any of these puzzles, a player could send off for one of various hints that showed where on the Eternity grid particular pieces were located in Monckton's solution. A £1 million award (approximately $1.5 million USD) was offered for the first person who solved the Eternity puzzle, which was finally collected by Alex Selby and Oliver Riordan on May 15, 2000. A second solution was later found by Guenter Stertenbrink. Interestingly, neither of these solutions matched the six clues given by Christopher Monckton for his solution, which remains unknown.

First, a working solution

In this section, we'll discuss an initial implementation of our puzzle-solving program. This will involve quite a few code fragments, so bear with us; once we have explained the basic algorithms involved, we will start optimizing. Source code for this initial implementation as well as the optimizations we'll discuss later in the article, is available in Resources.

The puzzle-solving algorithm

Our puzzle-solving program will calculate all possible solutions for the Meteor puzzle. This means that we will have to exhaustively search for every possible tiling of the board using the pieces. One step in accomplishing this task is to determine all the *permutations* of a piece. A permutation is a possible way of placing a piece on the board. Knowing that every piece can be flipped upside down and can be rotated around the six sides of one of its hexagons, we arrive at a total of 12 (2 x 6) possible ways to put a piece on one position of the board. With 50 board positions, the total number of possible ways to put a single piece on the board equals 600 (2 x 6 x 50).

Not all of these "possibilities" would actually work, of course. For instance, some have a piece hanging over the edge of the board, which clearly does not lead to a solution. Recursively repeating this process for all pieces brings us to a first algorithm that will find every possible solution by trying every possible tiling of the board using the pieces. Listing 1 presents the code for this algorithm. We use a simple `ArrayList` object called `pieceList` to hold all the pieces. The `board` object represents the puzzle board, which we will discuss shortly.

Listing 1. The initial puzzle-solving algorithm

```
public void solve() {
  if (!pieceList.isEmpty()) {
    // Take the first available piece
    Piece currentPiece = (Piece)pieceList.remove(0);

    for (int i = 0; i < Piece.NUMBEROFPERMUTATIONS; i++) {
      Piece permutation = currentPiece.nextPermutation();

      for (int j = 0; j < Board.NUMBEROFCELLS; j++) {
        if (board.placePiece(permutation, j)) {

          /* We have now put a piece on the board, so we have to
             continue this process with the next piece by
             recursively calling the solve() method */

          solve();

          /* We're back from the recursion and we have to continue
             searching at this level, so we remove the piece we
             just added from the board */

          board.removePiece(permutation);
        }
        // Else the permutation doesn't fit on the board
      }
    }

    // We're done with this piece
    pieceList.add(0, currentPiece);
  }
  else {

    /* All pieces have been placed on the board so we
       have found a solution! */

    puzzleSolved();
  }
}
```

Now that we have our basic algorithm set up, we need to investigate two other important issues:

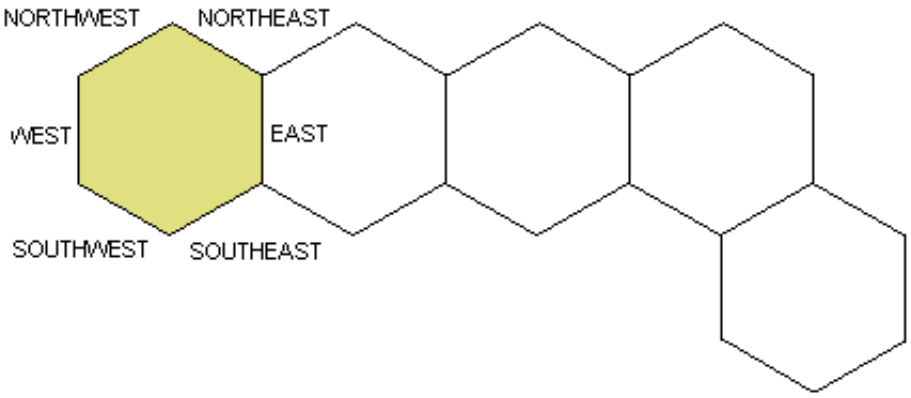
- How will we represent a piece of the puzzle?
- How will we implement the puzzle board?

In the algorithm shown in Listing 1, we used a `Piece` class and a `Board` class. Now let's take a look at the implementation of those two classes.

The Piece class

Before we start designing the `Piece` class, we need to consider what this class should represent. When you look at Figure 2, you can see that a Meteor puzzle piece consists of five connected cells. Each cell is a regular hexagon with six sides: EAST, SOUTHEAST, SOUTHWEST, WEST, NORTHWEST, and NORTHEAST. When two cells of a piece are joined at a particular side, we call these cells *neighbours*. In the end, a `Piece` object is nothing more than a set of five connected `Cell` objects. Each `Cell` object has six sides and six possible neighbouring cells.
Implementing the `Cell` class is straightforward, as shown in Listing 2. Note that we maintain a `processed` flag in a `Cell` object. We will use this flag later on to avoid infinite loops.

Figure 2. A puzzle piece and its cells



Listing 2. The Cell class

```
public class Cell {
  public static final int NUMBEROFSIDES = 6;

  // The sides of a cell
  public static final int EAST      = 0;
  public static final int SOUTHEAST = 1;
  public static final int SOUTHWEST = 2;
  public static final int WEST      = 3;
  public static final int NORTHWEST = 4;
  public static final int NORTHEAST = 5;

  private Cell[] neighbours = new Cell[NUMBEROFSIDES];

  private boolean processed = false;

  public Cell getNeighbour(int side) {
    return neighbours[side];
  }

  public void setNeighbour(int side, Cell cell) {
    neighbours[side] = cell;
  }

  public boolean isProcessed() {
    return processed;
  }

  public void setProcessed(boolean b) {
    processed = b;
  }
}
```

The `Piece` class is more interesting because we need a method to calculate the permutations of a Piece. We can find all permutations by first rotating the piece around the six sides of one of its cells, flipping it upside down, and finally rotating it again around the six sides of one of its cells. As we mentioned before, a piece consists of five adjacent cells. Flipping or rotating the piece is simply flipping or rotating all of its cells. So we need `flip()` and `rotate()` methods for `Cell` objects. Both flipping and rotating are easily accomplished by changing the neighbouring sides accordingly. These methods are provided in the `PieceCell` subclass of the `Cell` class, shown in Listing 3. A `PieceCell` object is a cell used in a `Piece` object.

Listing 3. The PieceCell subclass

```
public class PieceCell extends Cell {
  public void flip() {
    Cell buffer = getNeighbour(NORTHEAST);
    setNeighbour(NORTHEAST, getNeighbour(NORTHWEST));
    setNeighbour(NORTHWEST, buffer);
    buffer = getNeighbour(EAST);
    setNeighbour(EAST, getNeighbour(WEST));
    setNeighbour(WEST, buffer);
    buffer = getNeighbour(SOUTHEAST);
    setNeighbour(SOUTHEAST, getNeighbour(SOUTHWEST));
    setNeighbour(SOUTHWEST, buffer);
  }

  public void rotate() {
    // Clockwise rotation
    Cell eastNeighbour = getNeighbour(EAST);
    setNeighbour(EAST, getNeighbour(NORTHEAST));
    setNeighbour(NORTHEAST, getNeighbour(NORTHWEST));
    setNeighbour(NORTHWEST, getNeighbour(WEST));
    setNeighbour(WEST, getNeighbour(SOUTHWEST));
    setNeighbour(SOUTHWEST, getNeighbour(SOUTHEAST));
    setNeighbour(SOUTHEAST, eastNeighbour);
  }
}
```

Using the `PieceCell` class, we can complete the implementation of the `Piece` class. Listing 4 shows you the source code:

Listing 4. The Piece class

```
public class Piece {
  public static final int NUMBEROFCELLS = 5;
  public static final int NUMBEROFPERMUTATIONS = 12;

  private PieceCell[] pieceCells = new PieceCell[NUMBEROFCELLS];
  private int currentPermutation = 0;

  private void rotatePiece() {
    for (int i = 0; i < NUMBEROFCELLS; i++) {
      pieceCells[i].rotate();
    }
  }

  private void flipPiece() {
    for (int i = 0; i < NUMBEROFCELLS; i++) {
      pieceCells[i].flip();
    }
  }

  public Piece nextPermutation() {
    if (currentPermutation == NUMBEROFPERMUTATIONS)
      currentPermutation = 0;

    switch (currentPermutation%6) {
      case 0:
        // Flip after every 6 rotations
```

```
        flipPiece();
        break;

      default:
        rotatePiece();
        break;
    }

    currentPermutation++;

    return this;
  }

  public void resetProcessed() {
    for (int i = 0; i < NUMBEROFCELLS; i++) {
      pieceCells[i].setProcessed(false);
    }
  }

  //Getters and setters have been omitted
}
```

The Board class

Before we implement the `Board` class, we'll need to tackle two interesting problems. First we have to decide on a data structure. A Meteor puzzle board is basically a 5-by-10 grid of regular hexagons, which we can represent as an array of 50 `Cell` objects. Instead of using the `Cell` class directly, we'll use the `BoardCell` subclass, shown in Listing 5, which keeps track of the piece that occupies the cell:

Listing 5. The BoardCell subclass

```
public class BoardCell extends Cell {
  private Piece piece = null;

  public Piece getPiece() {
    return piece;
  }

  public void setPiece(Piece piece) {
    this.piece = piece;
  }
}
```

If we store all 50 board cells of the board in an array, we'll have to write some tedious initialisation code. This initialisation identifies the neighbouring board cells for each cell of the board, as illustrated in Figure 3. For instance, cell 0 has two neighbours: cell 1 in the east and cell 5 in the southeast. Listing 6 shows the `initializeBoardCell()` method that is called from the constructor of the `Board` class to do this initialisation.

Figure 3. The board represented as an array of cells

Now that we've implemented the data structure for the board, we move on to the next problem: writing a `placePiece()` method that puts a piece on the board. The hardest part of writing this method is deciding whether the piece fits on the board at the given position. One way to determine whether the piece fits is to first find all the board cells that would be occupied by the cells of the piece if it were placed on the board. After we have this set of

board cells, we can easily determine if the new piece would fit: all corresponding board cells need to be empty and the piece needs to fit completely on the board. This process is implemented by the `findOccupiedBoardCells()` method and `placePiece()` method shown in Listing 6. Note that we use the `processed` field of the `PieceCell` objects to avoid an infinite recursion in the `findOccupiedBoardCells()` method.

Listing 6. The Board class

```
public class Board {
  public static final int NUMBEROFCELLS = 50;
  public static final int NUMBEROFCELLSINROW = 5;

  private BoardCell[] boardCells = new BoardCell[NUMBEROFCELLS];

  public Board() {
    for (int i = 0; i < NUMBEROFCELLS; i++) {
      boardCells[i] = new BoardCell();
    }

    for (int i = 0; i < NUMBEROFCELLS; i++) {
      initializeBoardCell(boardCells[i], i);
    }
  }

  /**
   * Initialize the neighbours of the given boardCell at the given
   * index on the board
   */
  private void initializeBoardCell(BoardCell boardCell, int index) {
    int row = index/NUMBEROFCELLSINROW;

    // Check if cell is in last or first column
    boolean isFirst = (index%NUMBEROFCELLSINROW == 0);
    boolean isLast = ((index+1)%NUMBEROFCELLSINROW == 0);

    if (row%2 == 0) { // Even rows
      if (row != 0) {
        // Northern neighbours
        if (!isFirst) {
          boardCell.setNeighbour(Cell.NORTHWEST, boardCells[index-6]);
        }
        boardCell.setNeighbour(Cell.NORTHEAST, boardCells[index-5]);
      }
      if (row != ((NUMBEROFCELLS/NUMBEROFCELLSINROW)-1)) {
        // Southern neighbours
        if (!isFirst) {
          boardCell.setNeighbour(Cell.SOUTHWEST, boardCells[index+4]);
        }
        boardCell.setNeighbour(Cell.SOUTHEAST, boardCells[index+5]);
      }
    }
    else { // Uneven rows
      // Northern neighbours
      if (!isLast) {
        boardCell.setNeighbour(Cell.NORTHEAST, boardCells[index-4]);
      }
      boardCell.setNeighbour(Cell.NORTHWEST, boardCells[index-5]);
      // Southern neighbours
```

```
    if (row != ((NUMBEROFCELLS/NUMBEROFCELLSINROW)-1)) {
      if (!isLast) {
        boardCell.setNeighbour(Cell.SOUTHEAST, boardCells[index+6]);
      }
      boardCell.setNeighbour(Cell.SOUTHWEST, boardCells[index+5]);
    }
  }

  // Set the east and west neighbours
  if (!isFirst) {
    boardCell.setNeighbour(Cell.WEST, boardCells[index-1]);
  }
  if (!isLast) {
    boardCell.setNeighbour(Cell.EAST, boardCells[index+1]);
  }
}

public void findOccupiedBoardCells(
  ArrayList occupiedCells, PieceCell pieceCell, BoardCell boardCell) {
  if (pieceCell != null && boardCell != null && !pieceCell.isProcessed()) {
    occupiedCells.add(boardCell);

    /* Neighbouring cells can form loops, which would lead to an
       infinite recursion. Avoid this by marking the processed
       cells. */

    pieceCell.setProcessed(true);

    // Repeat for each neighbour of the piece cell
    for (int i = 0; i < Cell.NUMBEROFSIDES; i++) {
      findOccupiedBoardCells(occupiedCells,
                             (PieceCell)pieceCell.getNeighbour(i),
                             (BoardCell)boardCell.getNeighbour(i));
    }
  }
}

public boolean placePiece(Piece piece, int boardCellIdx) {
  // We will manipulate the piece using its first cell
  return placePiece(piece, 0, boardCellIdx);
}

public boolean
  placePiece(Piece piece, int pieceCellIdx, int boardCellIdx) {
  // We're going to process the piece
  piece.resetProcessed();

  // Get all the boardCells that this piece would occupy
  ArrayList occupiedBoardCells = new ArrayList();
  findOccupiedBoardCells(occupiedBoardCells,
                         piece.getPieceCell(pieceCellIdx),
                         boardCells[boardCellIdx]);

  if (occupiedBoardCells.size() != Piece.NUMBEROFCELLS) {
    // Some cells of the piece don't fall on the board
    return false;
  }
```

```
    for (int i = 0; i < occupiedBoardCells.size(); i++) {
      if (((BoardCell)occupiedBoardCells.get(i)).getPiece() != null)
        // The board cell is already occupied by another piece
        return false;
    }

    // Occupy the board cells with the piece
    for (int i = 0; i < occupiedBoardCells.size(); i++) {
      ((BoardCell)occupiedBoardCells.get(i)).setPiece(piece);
    }

    return true; // The piece fits on the board
  }

  public void removePiece(Piece piece) {
    for (int i = 0; i < NUMBEROFCELLS; i++) {
      // Piece objects are unique, so use reference equality
      if (boardCells[i].getPiece() == piece) {
        boardCells[i].setPiece(null);
      }
    }
  }
}
```

This completes the implementation of our initial solution. Let's put it to the test.

Running the program

Now that we have finished our first puzzle-solving program, we can run it to find all possible solutions for the Meteor puzzle. The source code described in the previous sections is found in the `meteor.initial` package of the source download. This package contains a `Solver` class that has a `solve()` method and a `main()` method to start the program. The constructor of the `Solver` class initializes all puzzle pieces and adds them to `pieceList`. We can launch the program using `java meteor.initial.Solver`.

The program starts searching for solutions, but as you will notice, it doesn't seem to find any. Actually, it does find all possible solutions, but you will have to be very patient. It takes several hours to find just one solution. Our test computer, an Athlon XP 1500+ with 512MB of RAM running RedHat Linux 7.2 and Java 1.4.0, finds the first solution after about eight hours. Finding all of them would take several months, if not years.

Clearly, we have a performance problem. A first candidate for optimization is the puzzle-solving algorithm. We're currently using a naive, brute-force approach to find all possible solutions. We should try to fine tune this algorithm. A second thing we can do is to cache temporary data. For instance, instead of recalculating the permutations of a piece every time, we could cache those permutations. Finally, we can try to apply some low-level optimization techniques, like avoiding unnecessary method calls. In the next sections, we'll study these optimization techniques.

Improving the algorithm

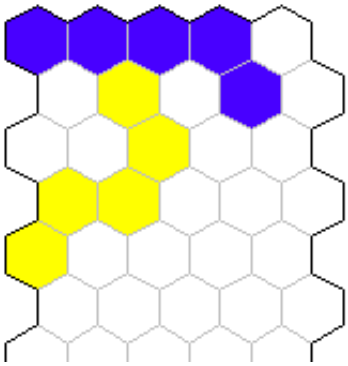
Take a look back at Listing 1 and think about how we might be able optimize our initial puzzle-solving algorithm. A good way to optimize an algorithm is to visualize it. Visualisation allows us to get a better understanding of the process being implemented and its possible downsides. The next sections discuss two inefficiencies we can discern. We leave the actual visualisation code for our puzzle-solving program to the interested reader.

Island detection pruning

The algorithm in Listing 1 fits pieces (or more precisely, the piece cells of a piece) onto every position of the board. Figure 4 shows a possible board situation at the beginning of the process. The current permutation of the blue piece has been placed on the first available board position and the current permutation of the yellow piece has moved to its second possible board position. Our algorithm then continues with the third piece, and so on. However, if we look carefully at Figure 4, it's clear that there will be no possible solutions for the puzzle with the blue and yellow pieces in these positions. The reason is those two pieces have formed an *island* of three neighbouring empty cells. Because all the puzzle pieces consist of five cells, there is no way to fill this island. All the effort that our algorithm exerts trying to fit the remaining eight pieces on the board is useless. What we need to do is cut off our algorithm if we

detect an island on the board that cannot be filled.

Figure 4. An island on the board



Text books call this process of interrupting a recursive search algorithm *pruning*. Adding a pruning function to our `Solver` class is easy. Before every recursive call to the `solve()` method, we check for islands on the board. If we find an island consisting of a number of empty cells that is not a multiple of five, we do not make the recursive call. Instead, the algorithm continues at the current level of recursion. Listings 7 and 8 show the necessary code adjustments:

Listing 7. A puzzle-solving algorithm with pruning

```
public class Solver {
  public void solve() {
    ...
            if (!prune()) solve();
    ...
  }

  private boolean prune() {
    /* We'll use the processed field of board cells to avoid
    infinite loops */
    board.resetProcessed();

    for (int i = 0; i < Board.NUMBEROFCELLS; i++) {
      if (board.getBoardCell(i).getIslandSize()%Piece.NUMBEROFCELLS != 0) {
        // We have found an unsolvable island
        return true;
      }
    }

    return false;
  }
}
```

Listing 8. The getIslandSize() method

```
public class BoardCell {
  public int getIslandSize() {
    if (!isProcessed() && isEmpty()) {
      setProcessed(true); // Avoid infinite recursion
      int numberOfCellsInIsland = 1; // this cell

      for (int i = 0; i < Cell.NUMBEROFSIDES; i++) {
        BoardCell neighbour=(BoardCell)getNeighbour(i);
        if (neighbour != null) {
          numberOfCellsInIsland += neighbour.getIslandSize();
        }
      }
      return numberOfCellsInIsland;
    }
    else {
      return 0;
    }
  }
}
```

The fill-up algorithm

A second downside of our initial algorithm is that it intrinsically generates a lot of islands. This happens because we take one permutation of a piece and move that over the board before switching to the next permutation of the piece. For instance, in Figure 5 we have moved the current permutation of the blue piece to its third possible board position. As you can see, this generates an island at the top of the board. While the island-detection pruning we added in the previous section will generate drastic performance improvements because of the large number of islands we're generating, it would be even better if we could update our algorithm to minimize the number of islands it generates in the first place.

Figure 5. Generating islands

To reduce the number of islands we generate, it would be best if our algorithm concentrated on filling empty board positions. So instead of just focusing on trying every possible way of tiling the board, we'll try to fill the board left-to-right, top-to-bottom. This new puzzle-solving algorithm is shown in Listing 9:

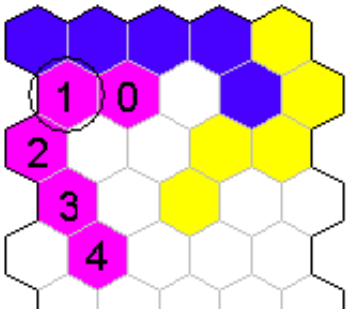
Listing 9. The fill-up puzzle-solving algorithm

```
public void solve() {
  if (!pieceList.isEmpty()) {
    // We'll try to find a piece that fits on this board cell
    int emptyBoardCellIdx = board.getFirstEmptyBoardCellIndex();

    // Try all available pieces
    for (int h = 0; h < pieceList.size(); h++) {
      Piece currentPiece = (Piece)pieceList.remove(h);

      for (int i = 0; i < Piece.NUMBEROFPERMUTATIONS; i++) {
        Piece permutation = currentPiece.nextPermutation();

        /* Instead of always using the first cell to manipulate
```

```
            the piece, we now try to fit any cell of the piece on
            the first empty board cell */

          for (int j = 0; j < Piece.NUMBEROFCELLS; j++) {
            if (board.placePiece(permutation, j, emptyBoardCellIdx)) {
              if (!prune()) solve();
              board.removePiece(permutation);
            }
          }
        }


        /* Put the piece back into the list at the position where
           we took it to maintain the order of the list */

        pieceList.add(h, currentPiece);
      }
    }
    else {
      puzzleSolved();
    }
}
```

Our new approach tries to fit any available piece on the first empty board cell. Just trying all possible permutations of all available pieces is not enough. We should also try to cover the empty board cell with any piece cell in the piece. In the initial algorithm, we silently assumed that we were manipulating the piece using its first cell. Now we have to try every cell in the piece, as illustrated in Figure 6. The current permutation of the pink piece does not fit on the board when we try to put the piece cell with index 0 on board position 5 (circled in Figure 6). However, it does fit when we use the second piece cell.

Figure 6. The cells of a piece



Running the updated program

When we ran our initial program, it failed to find any solutions in a reasonable amount of time. Let's try again with our improved algorithm and island-detection pruning. The code for this version of the program can be found in the package `meteor.algorithm`. When we launch it using `java meteor.algorithm.Solver`, we almost immediately see solutions popping up. Our test computer calculates all 2,098 possible solutions in 157 seconds. So we've made a gigantic performance improvement: from several hours per solution to less than one-tenth of a second. That's roughly 400,000 times as fast! As an aside, the initial algorithm combined with island detection pruning completes in 6,363 seconds. So the pruning optimization causes a 10,000-fold speedup, while the fill-up algorithm generates an extra 40-fold speedup. It clearly pays off to spend some time studying your algorithms and attempting to optimize them.

Caching intermediate results

The redesign of our puzzle-solving algorithm dramatically improved the execution speed of our program. For further optimizations, we'll have to look at technical performance techniques. An important issue to consider in Java programs is garbage collection. You can show the activity of the garbage collector during program execution by using the `-verbose:gc` command line switch.

```
java -verbose:gc meteor.algorithm.Solver
```

If we run our program with this switch, we see a lot of output from the garbage collector. Studying the source code tells us that the problem is the instantiation of a temporary `ArrayList` object in the `placePiece()` method of the `Board` class (see Listing 6). We use this `ArrayList` object to hold the board cells that a particular permutation of a piece would occupy. Instead of recalculating this list every time, it would be better to cache the results for later reference.

The `findOccupiedBoardCells()` method determines the cells of the puzzle board that would be occupied by a puzzle piece if a certain cell of that piece is placed on a certain board position. The results of the method are determined by three parameters: first we have the puzzle piece, or a permutation thereof; second we have the cell of the piece that we're using to manipulate the piece; and finally we have the cell of the board we'll put the piece on. To cache these results, we can associate a table with every possible piece permutation. This table holds the results of the `findOccupiedBoardCells()` method for that permutation using a specified piece cell index and board cell position. Listing 10 shows an updated version of the `Piece` class that maintains such a table:

Listing 10. Caching the results of the findOccupiedBoardCells() method

```
public class Piece {
  private Piece[] permutations = new Piece[NUMBEROFPERMUTATIONS];
  private ArrayList[][] occupiedBoardCells =
    new ArrayList[Piece.NUMBEROFCELLS][Board.NUMBEROFCELLS];

  private void generatePermutations(Board board) {
    Piece prevPermutation=this;
    for (int i = 0; i < NUMBEROFPERMUTATIONS; i++) {
      // The original nextPermutation() has been renamed
      permutations[i]=
        ((Piece)prevPermutation.clone()).nextPermutation_orig();
      prevPermutation=permutations[i];
    }

    // Calculate occupied board cells for every permutation
    for (int i = 0; i < NUMBEROFPERMUTATIONS; i++) {
      permutations[i].generateOccupiedBoardCells(board);
    }
  }

  private void generateOccupiedBoardCells(Board board) {
    for (int i = 0; i < Piece.NUMBEROFCELLS; i++) {
      for (int j = 0; j < Board.NUMBEROFCELLS; j++) {
        occupiedBoardCells[i][j]=new ArrayList();
        resetProcessed(); // We're going to process the piece
        board.findOccupiedBoardCells(occupiedBoardCells[i][j],
                                     pieceCells[i],
                                     board.getBoardCell(j));
      }
    }
  }

  public Piece nextPermutation() {
    if (currentPermutation == NUMBEROFPERMUTATIONS)
      currentPermutation = 0;

    // The new implementation of nextPermutation()
    // accesses the cache
    return permutations[currentPermutation++];
  }
```

```
  public ArrayList
    getOccupiedBoardCells(int pieceCellIdx, int boardCellIdx) {
    // Access requested data in cache
    return occupiedBoardCells[pieceCellIdx][boardCellIdx];
  }
}
```

The `generatePermutations()` method is triggered when a `Piece` object is created. It calculates every permutation of the piece and caches all possible results of the `findOccupiedBoardCells()` method for those permutations. It is clear that we'll need access to the puzzle board if we want to calculate the occupied board cells. Also note that the permutations of a piece are clones of the original `Piece` object. Cloning a `Piece` involves a deep copy of all of its cells.

The only thing left to do is to access the cache from the `placePiece()` method of the `Board` class, which is shown in Listing 11:

Listing 11. Accessing the occupied-board-cells cache

```
public class Board {
  public boolean
    placePiece(Piece piece, int pieceCellIdx, int boardCellIdx) {
    // Get all the boardCells that this piece would occupy
    ArrayList occupiedBoardCells =
      piece.getOccupiedBoardCells(pieceCellIdx, boardCellIdx);
    ...
  }
}
```

Running the program once more

The source code of this updated version of our puzzle-solving program can be found in the `meteor.caching` package. Running `java meteor.caching.Solver` shows us that we again improved the performance considerably. On our test machine all solutions are found in 25 seconds. Caching resulted in a six-fold speedup. If we use the `-verbose:gc` switch, we also see that garbage collection is no longer an issue.

The extra code we introduced to implement the cache obviously complicates the program. This is a typical downside of performance techniques that try to reduce computation time by storing intermediate results. However, in this case the performance gain seems to outweigh the added code complexity.

Programming optimizations

A final possible step in the optimization process for our puzzle-solving program is the use of low-level Java code optimization idioms. We're not manipulating any strings in our application, so applying the well-known `StringBuffer` idiom is useless. We could try to avoid the method call overhead for getters and setters by replacing those getters and setters with direct member access. However, this clearly degrades the quality of our code and tests show that this hardly generates any speedup at all. The same is true for the use of `final` methods. By declaring our methods as `final`, we avoid dynamic binding and allow the Java virtual machine to use more efficient static binding. But alas, this does not produce any noticeable speedup. Also, the use of the `-O` optimization switch of the Java compiler does not produce any real performance increase.

A slight execution speedup can still be obtained by improving the implementation of the `prune()` method. The code in Listing 7 always makes a call to the recursive `getIslandSize()` method, even if the board cell is already processed or is not empty. If we proactively do these checks before invoking `getIslandSize()`, we gain about 10 percent.

As is clear from this discussion, low-level optimizations result in very small performance increases. This, combined with the fact that some of these optimization techniques deteriorate the quality of your code, makes the use of low-level optimizations unappealing.

Conclusion

All our effort to improve the implementation of our puzzle-solving program certainly paid off. Table 1 summarizes

the different versions we created and their execution times. The overall result is an amazing estimated 2,000,000-fold speedup.

Table 1. Comparing execution times

Version	**Time (seconds)**
meteor.initial	~ 60,422,400 (about 2 years)
meteor.algorithm	157
meteor.caching	25

However impressive this optimization might be, the important question is what can we learn from this experiment? The different optimization techniques we used each have their benefits and drawbacks. Combining them into a single optimization process clarifies their use and prevents out-of-order application:

- High-level optimization techniques, like the algorithm improvements we used, have great potential. If you need to optimize a performance-critical piece of code, first try to analyse the process this code implements. Visualizing the process is an excellent way to gain a better understanding of it. Also try to tackle the problem from different angles. You might come up with a vastly better solution than the one you originally invented. An obvious difficulty with this kind of optimization is that it's hard to generalize. Every algorithm is specific to a particular application domain and as such there are few general guidelines that can be provided. It's up to the programmer to be creative.

- Once you're sure you have a good working solution in place, it's time to apply technical performance improvement techniques. The basic idea is to exchange *time complexity* for *data complexity*. Object caches are one of the most typical of such techniques. In Java programs, object caches are particularly useful because they help you avoid expensive object creation and garbage collection overhead. Remember, this kind of system adds extra infrastructure code to your programs, so don't introduce it too early. The more complex your code is, the harder it is to optimize.

- Finally, we can apply a range of low-level programming optimizations. Most Java programmers are familiar with these kinds of techniques. However, their benefit is limited in most real-world programs. Apply them where possible, but don't focus all your optimization effort on these kinds of idioms. Rather, they should be part of your programming toolset to help you avoid well-known performance traps.

The spectacular performance increases we achieved by combining different optimization techniques in our puzzle-solving program should motivate all Java programmers to take a look at their own code and see how it could be optimized.

Acknowledgments
The authors would like to acknowledge the assistance of Bieke Meeussen in reviewing this document for publication. We would also like to acknowledge the contributions of Pieter Bekaert and Kris Cardinaels. Pieter came up with the fill-up puzzle-solving algorithm and Kris developed parts of the puzzle visualisation code.

Resources

- Participate in the discussion forum on this article by clicking **Discuss** at the top or bottom of the article.

- You can download the source code for the three different versions of the Meteor puzzle-solving program we developed in this article. This zip file also contains the solution viewer that allows you to visualize the found solutions.

- The recently released Java 2 Platform, Standard Edition version 1.4 introduces several performance-oriented features. "Merlin brings nonblocking I/O to the Java platform" (*developerWorks*, March 2002) describes one of these features: the new I/O API.

- "Improve the performance of your Java code" (*developerWorks*, May 2001) is an installment of Eric Allen's *Diagnosing Java code* series. In it, he discusses tail-recursive methods and some of the issues involved in

optimizing them.

- Profiling tools can be a big help when exploring and visualizing the run-time behaviour of your program. "Jinsight: A tool for visualizing the execution of Java programs" (*developerWorks*, November 1999) describes the use of one such tool (Jinsight), which was developed at IBM's Research Division and is available for free.

- Best Practice: String Concatenation with Java explains the well-known StringBuffer idiom. Another description of this technique can be found in the March 5, 2002, issue of *JDC Tech Tips* .

- The December 22, 2000, issue of *JDC Tech Tips* explains the use of the Java garbage collector. Techniques for tracking and controlling memory allocation in Java programs are further discussed in "Heap of trouble" (*developerWorks*, September 1999).

- If you are interested in learning more about the larger 209-piece Eternity puzzle and how it was solved, take a look at Prize specimens by Mark Wainwright. It turns out that solving Eternity was quite a bit harder than solving Meteor, like we did in this article. Further information about the Eternity puzzle can also be found on the Eternity page.

- Find other Java programming resources on the *developerWorks* Java technology zone. Of particular interest are the Java performance articles you can find there.

About the authors

Erwin Vervaet is a software engineer with a keen interest in applying modern IT concepts and tools. He has been using the Java programming language since 1996, and has a master's degree in computer science from the Katholieke Universiteit Leuven in Belgium. He has been involved in IT research, e-commerce projects, open source initiatives, and industrial software systems. As an independent consultant, Erwin builds object-oriented business information systems using the Java programming language. You can contact him at erwin@ervacon.com.

Maarten De Cock is a programmer focusing on writing clean and fast Java code. After graduating from the Katholieke Hogeschool Leuven, Belgium, he started using the Java programming language in 1999. Since then he has participated in several e-commerce projects and also teaches Java programming courses on a regular basis. He currently works as a consultant for ASQdotCOM. Contact Maarten at maarten.decock@asq.be.