```
// This AI follows a hard-coded strategy based on the programmer's
experience.
// It is not a trained AI.

GoalType = { UNDEFINED: -1, BUILD: 0, SHIFT: 1, MOVE: 2 };

Goal = function() {
};
Goal.prototype = {
  type: GoalType.UNDEFINED
};

SmartAI = function(game) {
  this.game = game;
};

SmartAI.prototype.nextMove = function() {
  // Determine the best move given the current game state
  // Use a couple of stragtegies:
  // 1. Goals:
  //   - Determine the main goal given the current board state
  //     (e.g. if the highest number is 64, build it up to 128)
  //   - Determine sub-goals that must happen to achieve that goal
  // 2. Planning ahead
  //   - in some cases, the AI should plan ahead a certain number of
  //     moves to determine the effect of each possible move it can
make.
  //     If it sees that a certain move will put the board in a bad
state
  //     (i.e. forced to push the wrong direction), then it will
avoid that
  //     move. Alternatively, if it sees a sequence of moves that
will
  //     accomplish a goal, it will do those moves.
  /*var goal = this.determineGoal(this.game.grid);
  // Keep looking at sub-goals until we find a sub-goal that is a
simple movement.
  while (goal.type != GoalType.MOVE) {
    goal = this.determineSubGoal(this.game.grid, goal);
  }

  if (goal.directions) {
    // Move in the most optimal legal direction.
    for (var i = 0; i < goal.directions.length; i++) {
      if (this.game.moveAvailable(goal.directions[i])) {
        return goal.directions[i];
      }
    }
  }*/

  // Plan ahead a few moves in every direction and analyze the board
state.
  // Go for moves that put the board in a better state.
  var originalQuality = this.gridQuality(this.game.grid);
  var results = this.planAhead(this.game.grid, 3, originalQuality);
  // Choose the best result
  var bestResult = this.chooseBestMove(results, originalQuality);

  return bestResult.direction;
};

// Plans a few moves ahead and returns the worst-case scenario grid
quality,
// and the probability of that occurring, for each move
SmartAI.prototype.planAhead = function(grid, numMoves,
originalQuality) {
  var results = new Array(4);

  // Try each move and see what happens.
  for (var d = 0; d < 4; d++) {
    // Work with a clone so we don't modify the original grid.
    var testGrid = grid.clone();
    var testGame = new GameController(testGrid);
    var moved = testGame.moveTiles(d);
    if (!moved) {
      results[d] = null;
      continue;
    }
    // Spawn a 2 in all possible locations.
    var result = {
      quality: -1,    // Quality of the grid
      probability: 1, // Probability that the above quality will
happen
      qualityLoss: 0, // Sum of the amount that the quality will
have decreased multiplied by the probability of the decrease
      direction: d
    };
    var availableCells = testGrid.availableCells();
    for (var i = 0; i < availableCells.length; i++) {
      // Assume that the worst spawn location is adjacent to an
existing tile,
      // and only test cells that are adjacent to a tile.
      var hasAdjacentTile = false;
      for (var d2 = 0; d2 < 4; d2++) {
        var vector = testGame.getVector(d2);
        var adjCell = {
          x: availableCells[i].x + vector.x,
          y: availableCells[i].y + vector.y,
        };
        if (testGrid.cellContent(adjCell)) {
          hasAdjacentTile = true;
          break;
        }
      }
      if (!hasAdjacentTile)
        continue;

      var testGrid2 = testGrid.clone();
      var testGame2 = new GameController(testGrid2);
      testGame2.addTile(new Tile(availableCells[i], 2));
      var tileResult;
      if (numMoves > 1) {
        var subResults = this.planAhead(testGrid2, numMoves - 1,
originalQuality);
        // Choose the sub-result with the BEST quality since that is
the direction
        // that would be chosen in that case.
        tileResult = this.chooseBestMove(subResults,
originalQuality);
      } else {
        var tileQuality = this.gridQuality(testGrid2);
        tileResult = {
          quality: tileQuality,
          probability: 1,
          qualityLoss: Math.max(originalQuality - tileQuality, 0)
        };
      }
      // Compare this grid quality to the grid quality for other
tile spawn locations.
      // Take the WORST quality since we have no control over where
the tile spawns,
      // so assume the worst case scenario.
      if (result.quality == -1 || tileResult.quality <
result.quality) {
        result.quality = tileResult.quality;
        result.probability = tileResult.probability /
availableCells.length;
      } else if (tileResult.quality == result.quality) {
        result.probability += tileResult.probability /
availableCells.length;
      }
      result.qualityLoss += tileResult.qualityLoss /
availableCells.length;
    }
    results[d] = result;
  }
  return results;
}

SmartAI.prototype.chooseBestMove = function(results,
originalQuality) {
  // Choose the move with the least probability of decreasing the
grid quality.
  // If multiple results have the same probability, choose the one
with the best quality.
  var bestResult;
  for (i = 0; i < results.length; i++) {
    if (results[i] == null)
      continue;
    if (!bestResult ||
        results[i].qualityLoss < bestResult.qualityLoss ||
        (results[i].qualityLoss == bestResult.qualityLoss &&
results[i].quality > bestResult.quality) ||
        (results[i].qualityLoss == bestResult.qualityLoss &&
results[i].quality == bestResult.quality && results[i].probability <
bestResult.probability)) {
      bestResult = results[i];
    }
  }
  if (!bestResult) {
    bestResult = {
      quality: -1,
      probability: 1,
      qualityLoss: originalQuality,
      direction: 0
    };
  }
  return bestResult;
}

// Gets the quality of the current state of the grid
SmartAI.prototype.gridQuality = function(grid) {
  /* Look at monotonicity of each row and column and sum up the
scores.
   * (monoticity = the amount to which a row/column is constantly
increasing or decreasing)
   *
   * How monoticity is scored (may be subject to modification):
   *   score += current_tile_value
   *   -> If a tile goes againt the monoticity direction:
   *      score -= max(current_tile_value, prev_tile_value)
   *
   * Examples:
   *   2    128    64   32
   *  +2     +0   +64  +32

   *  32     64   128    2
   * +32    +64  +128 -126
   *
   *   128   64   32   32
   *  +128  +64  +32  +32
   *
   *   ___  128   64   32
   *    +0   +0  +64  +32
   *
   *   128   64   32  ___
   *  +128  +64  +32
   *
   *   ___  128  ___  ___
   *         +0
   *
   *   ___  128  ___  32
   *         +0      +32
   */
  var monoScore = 0; // monoticity score
  var traversals = this.game.buildTraversals({x: -1, y:  0});
  var prevValue = -1;
  var incScore = 0, decScore = 0;

  var scoreCell = function(cell) {
    var tile = grid.cellContent(cell);
    var tileValue = (tile ? tile.value : 0);
    incScore += tileValue;
    if (tileValue <= prevValue || prevValue == -1) {
      decScore += tileValue;
      if (tileValue < prevValue) {
        incScore -= prevValue;
      }
    }
    prevValue = tileValue;
  };

  // Traverse each column
  traversals.x.forEach(function (x) {
    prevValue = -1;
    incScore = 0;
    decScore = 0;
    traversals.y.forEach(function (y) {
      scoreCell({ x: x, y: y });
    });
    monoScore += Math.max(incScore, decScore);
  });
  // Traverse each row
  traversals.y.forEach(function (y) {
    prevValue = -1;
    incScore = 0;
    decScore = 0;
    traversals.x.forEach(function (x) {
      scoreCell({ x: x, y: y });
    });
    monoScore += Math.max(incScore, decScore);
  });

  // Now look at number of empty cells. More empty cells = better.
  var availableCells = grid.availableCells();
  var emptyCellWeight = 8;
  var emptyScore = availableCells.length * emptyCellWeight;

  var score = monoScore + emptyScore;
  return score;
}

// Determine the main (highest level) goal to accomplish for the
current grid
SmartAI.prototype.determineGoal = function(grid) {
  var goal = new Goal();
  // Find the highest tile on the board.
  var maxValue = 0;
  var maxCells = [];
  grid.eachCell(function(x, y, tile) {
    if (tile && tile.value >= maxValue) {
      if (tile.value > maxValue) {
        maxCells = [];
        maxValue = tile.value;
      }
      maxCells.push({x: x, y: y});
    }
  });
  var maxCell;
  if (maxCells.length == 1) {
    maxCell = maxCells[0];
  } else {
    // If there are multiple cells with the highest value, choose
the one closest to the corner
    var minDist = grid.size;
    for (var i = 0; i < maxCells.length; i++) {
      dist = Math.min(maxCells[i].x, grid.size - maxCells[i].x - 1)
           + Math.min(maxCells[i].y, grid.size - maxCells[i].y - 1);
      if (dist < minDist) {
        minDist = dist;
        maxCell = maxCells[i];
      }
    }
  }
  // Find the distance of the max tile from the corner
  dist = Math.min(maxCell.x, grid.size - maxCell.x - 1)
       + Math.min(maxCell.y, grid.size - maxCell.y - 1);
  if (dist == 0) {
    // Great! The tile is in a corner.
    // In this case, the goal is to double that tile's value.
    goal.type = GoalType.BUILD;
    goal.cell = maxCell;
    goal.value = maxValue * 2;
    return goal;
  }
  // Shoot, the highest tile is not in the corner.
  // Find a way to get it in the corner.
  if (dist == 1) {
    if (maxValue <= 512) {
      // Option 1: build up the corner tile to have the same value
as the max tile
      // This is only reasonable if the tile value is not greater
than 512.
      goal.type = GoalType.BUILD;
      goal.cell = { x: (maxCell.x < grid.size / 2 ? 0 : grid.size -
1),
                    y: (maxCell.y < grid.size / 2 ? 0 : grid.size -
1) };
      goal.value = maxValue;
      return goal;
    }
    // Things are looking pretty rough.
    // Option 2: do some fancy moves to try and shift the max tile
into a different corner.
```

```
    // This is only reasonable if there are enough open cells.
    var availableCells = game.grid.availableCells();
    if (availableCells.length > 4) {
      // TODO: if the target cell is empty, just move! (don't shift)
      goal.type = GoalType.SHIFT;
      goal.fromCell = maxCell;
      if (maxCell.x == 0 || maxCell.x == game.size - 1) {
        goal.cell = { x: maxCell.x,
                      y: (maxCell.y < grid.size / 2 ? grid.size - 1:
0) };
      } else {
        goal.cell = { x: (maxCell.x < grid.size / 2 ? grid.size -
1 : 0),
                      y: maxCell.y };
      }
      return goal;
    }
    // Now things are looking REALLY rough.
    // Option 3: Our best bet is to try and build up the max tile to
clear up room on the board.
    goal.type = GoalType.BUILD;
    goal.cell = maxCell;
    goal.value = maxValue * 2;
    return goal;
  }
  // dist > 1
  // The cell is really far from a corner, which sucks.
  // Do some fancy moves to try and shift the max tile into a
different corner.
  var availableCells = game.grid.availableCells();
  goal.type = GoalType.SHIFT;
  goal.fromCell = maxCell;
  goal.cell = { x: maxCell.x,
                y: (maxCell.y < grid.size / 2 ? grid.size - 1: 0) };
  return goal;
};

// Determine the sub-goal required to achieve the current goal.
SmartAI.prototype.determineSubGoal = function(grid, goal) {
  var subgoal = new Goal();
  if (goal.type == GoalType.BUILD) {
    var tile = grid.cellContent(goal.cell);

    if (!tile) {
      // Cell is empty. This is easy; just move a tile into that
cell.
      goal.type = GoalType.MOVE;
      vector = { x: goal.cell.x - grid.size / 2,
                 y: goal.cell.y - grid.size / 2 };
      goal.directions = this.getDirections(vector);
      return goal;
    }

    // See if any adjacent cells have equal value.
    var adjacentCells = new Array(4);
    for (i = 0; i < 4; i++) {
      var vector = this.game.getVector(i);
      var adjCell = {x: goal.cell.x + vector.x, y: goal.cell.y +
vector.y };
      var adjTile = grid.cellContent(adjCell);
      if (adjTile && adjTile.value == tile.value) {
        // Adjacent tiles with equal value. Combine the tiles.
        // Flip the vector and use that as the direction.
        vector.x = -vector.x;
        vector.y = -vector.y;
        goal.type = GoalType.MOVE;
        goal.directions = this.getDirections(vector);
        return goal;
      }
    }

    // No tiles to combine. Start building an adjacent tile.

  } else if (goal.type == GoalType.MOVE) {
  }
  return subgoal;
};

// Gets the direction of movement priority order given a vector
SmartAI.prototype.getDirections = function(vector) {
  directions = [0, 3, -1, -1];
  if (vector.x > 0) {
    directions[0] = 2;
  }
  if (vector.y > 0) {
    directions[1] = 1;
  }
  if (Math.abs(vector.x) > Math.abs(vector.y)) {
    var temp = directions[0];
    directions[0] = directions[1];
    directions[1] = temp;
  }
  directions[2] = (directions[1] + 2) % 4;
  directions[3] = (directions[0] + 2) % 4;
}
```