CFPT-INFORMATIQUE

Travaux de Diplômes 2017

Prisoner's Dilemma Cellular Automaton

Code source

Julien SEEMULLER

Supervisé par :

Mme. Terrier

T.IS-E2B

12 juin 2017

Table des matières

1 Vues

1.1 AboutView.cs

```
1   /*
2    Class           :   AboutView.cs
3    Description     :   Gives general information about the project
4    Author          :   SEEMULLER Julien
5    Date            :   10.04.2017
6   */
7
8   using System;
9   using System.Diagnostics;
10  using System.Windows.Forms;
11
12  namespace PrisonersDilemmaCA
13  {
14      public partial class AboutView : Form
15      {
16          public AboutView()
17          {
18              InitializeComponent();
19          }
20
21          /// <summary>
22          /// Close the form
23          /// </summary>
24          /// <param name="sender"></param>
25          /// <param name="e"></param>
26          private void btnClose_Click(object sender, EventArgs e)
27          {
28              this.Close();
29          }
30
31          /// <summary>
32          /// Open the wiki page
33          /// </summary>
34          /// <param name="sender"></param>
35          /// <param name="e"></param>
36          private void linkLabel1_LinkClicked(object sender, ↩
                LinkLabelLinkClickedEventArgs e)
37          {
38              Process.Start(lblWikiLink.Text);
39          }
40
41          /// <summary>
42          /// Open the github page
43          /// </summary>
44          /// <param name="sender"></param>
45          /// <param name="e"></param>
46          private void lblGithubLink_LinkClicked(object sender, ↩
                LinkLabelLinkClickedEventArgs e)
47          {
48              Process.Start(lblGithubLink.Text);
49          }
50      }
51  }
```

1.2 GenerateHelpView.cs

```
1   /*
2    Class           :   GenerateHelpView.cs
3    Description     :   Gives help on generating grid of cells
4    Author          :   SEEMULLER Julien
5    Date            :   10.04.2017
6   */
7
8   using System;
9   using System.Windows.Forms;
10
11  namespace PrisonersDilemmaCA
12  {
13      public partial class GenerateHelpView : Form
14      {
15          public GenerateHelpView()
16          {
17              InitializeComponent();
```

```
        }

        /// <summary>
        /// Closes the form
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void btnClose_Click(object sender, EventArgs e)
        {
            this.Close();
        }
    }
}
```

1.3 GenerateView.cs

```
   /*
    Class           :   GenerateView.cs
    Description     :   Allows the user to generate grids of cells with various ↩
        strategies
    Author          :   SEEMULLER Julien
    Date            :   10.04.2017
*/

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Drawing;
using System.Linq;
using System.Windows.Forms;

namespace PrisonersDilemmaCA
{
    public partial class GenerateView : Form
    {
        public Grid currentGrid { get; set; }
        public List<Strategy> strategies { get; set; }

        int nbOfStrategies;
        int heightOfComponents = 40;
        int widthOfComponents = 150;
        int spacing;
        int formHeight;
        int formWidth;
        List<TrackBar> trackbars;
        List<Label> trackbarLabels;
        List<int> lastTrackbarValues;

        public GenerateView()
        {
            InitializeComponent();
        }

        /// <summary>
        /// Generates the GUI dynamially on load
        /// </summary>
        /// <param name="sender"></param>
        /// <param name="e"></param>
        private void GenerateView_Load(object sender, EventArgs e)
        {
            // Store the number of available strategies we have
            nbOfStrategies = strategies.Count;

            // Define some values to create our view dynamically
            spacing = heightOfComponents;
            formHeight = 0;
            formWidth = 0;

            // Initialize our list of trackbars
            trackbars = new List<TrackBar>();
            trackbarLabels = new List<Label>();
            lastTrackbarValues = new List<int>();

            // Generate the interface dynamically
            for (int i = 1; i <= nbOfStrategies; i++)
            {
                Label tmpLabel = new Label();
                int x = spacing;
                int y = i * (heightOfComponents / 3 + spacing);

```

```
64                  // Set the location of the label
65                  tmpLabel.Location = new Point(x, y);
66                  tmpLabel.Width = widthOfComponents;
67                  tmpLabel.Height = heightOfComponents;
68
69                  // Set the label font
70                  tmpLabel.Font = new Font(FontFamily.GenericSansSerif, 11);
71
72                  // Add the label content
73                  string strategyName = strategies[i - 1].ToString();
74                  tmpLabel.Text = strategyName;
75
76                  // Create a trackbar
77                  TrackBar tmpTrackbar = new TrackBar();
78                  tmpTrackbar.Location = new Point(x + tmpLabel.Width + spacing, y);
79                  tmpTrackbar.Size = new Size(lblTitle.Width / 2 - x / 2, ↩
                        heightOfComponents);
80                  tmpTrackbar.Anchor = (AnchorStyles.Right | AnchorStyles.Top | ↩
                        AnchorStyles.Left);
81
82                  // Set the trackbar's parameters
83                  tmpTrackbar.Minimum = 0;
84                  tmpTrackbar.Maximum = 100;
85                  tmpTrackbar.Value = 0;
86                  tmpTrackbar.TickFrequency = 10;
87
88                  // Add an event handler to automatically refresh the interface
89                  tmpTrackbar.ValueChanged += new EventHandler(UpdateValues);
90
91                  // Add the trackbar to the list
92                  trackbars.Add(tmpTrackbar);
93                  lastTrackbarValues.Add(tmpTrackbar.Value);
94
95                  // Add a label for each trackbar
96                  Label tmpTrackbarLabel = new Label();
97
98                  // Set the location of the label (next to the trackbar)
99                  tmpTrackbarLabel.Location = new Point(tmpTrackbar.Left + ↩
                        tmpTrackbar.Width + spacing, y);
100                 tmpTrackbarLabel.Width = widthOfComponents;
101                 tmpTrackbarLabel.Height = heightOfComponents;
102
103                 // Anchor it
104                 tmpTrackbarLabel.Anchor = (AnchorStyles.Right | AnchorStyles.Top);
105
106                 // Set the label font
107                 tmpTrackbarLabel.Font = new Font(FontFamily.GenericSansSerif, 12);
108
109                 // Add it to the list
110                 trackbarLabels.Add(tmpTrackbarLabel);
111
112                 // Add the components to the form
113                 this.Controls.Add(tmpLabel);
114                 this.Controls.Add(tmpTrackbar);
115                 this.Controls.Add(tmpTrackbarLabel);
116
117                 // Set the form width and height
118                 formHeight += heightOfComponents + spacing - 5;
119             }
120             formWidth = lblTitle.Width + spacing * 3;
121
122             // Set the form's dimensions
123             this.MinimumSize = new Size(formWidth, formHeight);
124
125             // Center the form on screen
126             Rectangle screenSize = Screen.PrimaryScreen.Bounds;
127             int newX = screenSize.Width / 2 - this.Width / 2;
128             int newY = screenSize.Height / 2 - this.Height / 2;
129
130             this.Left = newX;
131             this.Top = newY;
132
133             // Update the controls
134             UpdateValues(null, null);
135         }
136
137         /// <summary>
138         /// Refreshes the interface and prevents the user from inputing incorrect values
139         /// </summary>
140         /// <param name="sender"></param>
141         /// <param name="e"></param>
142         public void UpdateValues(object sender, EventArgs e)
143         {
144             // Get the current total percentage
```

```
145             int sum = trackbars.Sum(item => item.Value);
146
147             // Set the new max value of the trackbars
148             for (int i = 0; i < trackbars.Count; i++)
149             {
150                 // If we are at 100 percent, prevent the user from incrementing even more
151                 if (sum > 100)
152                 {
153                     if (trackbars[i].Value > lastTrackbarValues[i])
154                     {
155                         // Restore from the last value
156                         trackbars[i].Value = lastTrackbarValues[i];
157                     }
158                 }
159                 // Store the last value
160                 lastTrackbarValues[i] = trackbars[i].Value;
161
162                 // Refresh the percentage of each of the trackbar's label
163                 trackbarLabels[i].Text = String.Format("{0}%", trackbars[i].Value);
164             }
165
166             // The percentage is equal to the sum of each of the trackbar's value
167             sum = (sum > 100) ? 100 : sum;
168             pbPercentage.Value = sum;
169             lblPercentage.Text = String.Format("{0}%", sum);
170
171             // Enable the button if we have 100% progress
172             btnApply.Enabled = (sum >= 100) ? true : false;
173         }
174
175         /// <summary>
176         /// Closes the form
177         /// </summary>
178         /// <param name="sender"></param>
179         /// <param name="e"></param>
180         private void btnCancel_Click(object sender, EventArgs e)
181         {
182             this.Close();
183         }
184
185         /// <summary>
186         /// Generates a new board with random cells
187         /// </summary>
188         /// <param name="sender"></param>
189         /// <param name="e"></param>
190         private void btnApply_Click(object sender, EventArgs e)
191         {
192             // Create a new random number generator
193             Random rng = new Random();
194
195             Dictionary<Strategy, int> stratAndPercent = new Dictionary<Strategy, int>();
196             List<Strategy> toRemove = new List<Strategy>();
197
198             // Filter out the unused strategies
199             for (int i = 0; i < nbOfStrategies; i++)
200             {
201                 if (!(trackbars[i].Value <= 0))
202                 {
203                     // Store the percentage of the remaining strategies
204                     stratAndPercent.Add(strategies[i], trackbars[i].Value);
205                 }
206             }
207
208             Grid tmpGrid = new Grid(currentGrid.Width, currentGrid.Height, ↩
                 currentGrid.NbLines, currentGrid.NbCols, currentGrid.PayoffMatrix);
209
210             // Generate a new board
211             tmpGrid.generate(stratAndPercent);
212             currentGrid.Cells = tmpGrid.Cells;
213
214             // Close the form
215             this.Close();
216         }
217
218         /// <summary>
219         /// Open the help form
220         /// </summary>
221         /// <param name="sender"></param>
222         /// <param name="e"></param>
223         private void GenerateView_HelpButtonClicked(object sender, CancelEventArgs e)
224         {
225             GenerateHelpView helpView = new GenerateHelpView();
226
227             if (helpView.ShowDialog() == DialogResult.OK)
```

```
228             {
229                 // User pressed the close button
230             }
231         }
232     }
233 }
```

1.4 MainView.cs

```
1    /*
2     Class           :   MainView.cs
3     Description     :   Main view of the application.
4     Author          :   SEEMULLER Julien
5     Date            :   10.04.2017
6 */
7
8  using LiveCharts;
9  using LiveCharts.Wpf;
10 using System;
11 using System.Collections.Generic;
12 using System.Windows.Forms;
13
14 namespace PrisonersDilemmaCA
15 {
16     public partial class MainView : Form
17     {
18
19         /*************************************************************************
20          *                              GLOBAL VARIABLES                         *
21          *************************************************************************/
22         Grid mainGrid;
23         PayoffMatrix payoffMatrix;
24         List<Strategy> availableStrategies;
25         bool isClickingOnGrid = false;
26         bool isAutoplaying = false;
27         int mouseX = 0;
28         int mouseY = 0;
29         int generation = 0;
30         WrapMode gridWrappingMode = WrapMode.Default;
31
32         const int MAX_NB_ELEMENTS_IN_CHART = 10;
33         const int DEFAULT_NORMAL_VIEW_WIDTH = 610;
34         const int DEFAULT_EXTENDED_VIEW_WIDTH = 1100;
35         /*************************************************************************
36          *                                EVENTS                                 *
37          *************************************************************************/
38         /// <summary>
39         /// Default constructor
40         /// </summary>
41         public MainView()
42         {
43             InitializeComponent();
44         }
45
46         /// <summary>
47         /// Set up the form after it is loaded
48         /// </summary>
49         /// <param name="sender"></param>
50         /// <param name="e"></param>
51         private void MainView_Load(object sender, EventArgs e)
52         {
53             // Make a list of all our available strategies
54             availableStrategies = new List<Strategy>();
55
56             // To add more strategies, add them to the list
57             availableStrategies.Add(new StratRandom());
58             availableStrategies.Add(new StratTitForTat());
59             availableStrategies.Add(new StratBlinker());
60             availableStrategies.Add(new StratAlwaysCooperate());
61             availableStrategies.Add(new StratAlwaysDefect());
62             availableStrategies.Add(new StratTitForTwoTats());
63             availableStrategies.Add(new StratGrimTrigger());
64             availableStrategies.Add(new StratFortress());
65             availableStrategies.Add(new StratAdaptativePavlov());
66             availableStrategies.Add(new StratSuspiciousTitForTat());
67
68             // Sort the list
69             availableStrategies.Sort();
70
71             // Initialize the payoff matrix with default values
```

```
72              payoffMatrix = new PayoffMatrix();
73
74              // Initialize our grid of cells
75              mainGrid = new Grid(pbGrid.Width, pbGrid.Height, tbLines.Value, ↩
                    tbColumns.Value, payoffMatrix, gridWrappingMode);
76
77              // Initialise the combobox with strategies and colors
78              cbStrategies.AddStrategies(availableStrategies);
79
80              // Select the first element by default
81              cbStrategies.SelectedIndex = 0;
82
83              // Set the user help text
84              lblUserHelp.Text = "Click on a cell to change its strategy."
85                  + Environment.NewLine + "The default strategy is "
86                  + Cell.DEFAULT_STRATEGY.ToString();
87
88              // Update the other labels
89              updateLabels();
90
91              // CHARTS
92              // Pie chart
93              pieStrategy.InnerRadius = 50;
94              pieStrategy.LegendLocation = LegendLocation.Right;
95              pieStrategy.DisableAnimations = true;
96              pieStrategy.Series = new SeriesCollection();
97
98              foreach (Strategy strategy in availableStrategies)
99              {
100                 // Get the color from the strategy
101                 System.Windows.Media.BrushConverter converter = new ↩
                        System.Windows.Media.BrushConverter();
102                 System.Windows.Media.Brush brush = ↩
                        (System.Windows.Media.Brush)converter.ConvertFromString(strategy.getColor().ToHex()
103
104                 // Create an object for storing values on the pie chart
105                 PieSeries stratToAdd = new PieSeries
106                 {
107                     Title = strategy.ToString(),
108                     Values = new ChartValues<double>
109                         {
110                             mainGrid.findCountOfStrategy(strategy)
111                         },
112                     DataLabels = true,
113                     Fill = brush
114                 };
115
116                 stratToAdd.Visibility = System.Windows.Visibility.Hidden;
117
118
119                 // Add the values to the pie chart
120                 pieStrategy.Series.Add(stratToAdd);
121             }
122
123             // Cartesian
124             cartesianStrategy.LegendLocation = LegendLocation.Right;
125             cartesianStrategy.AxisX.Add(new Axis
126             {
127                 Title = "Current Generation",
128                 LabelFormatter = value => value.ToString()
129             });
130
131             cartesianStrategy.AxisY.Add(new Axis
132             {
133                 Title = "Number of Days in Prison",
134                 LabelFormatter = value => value.ToString(),
135
136             });
137
138             // Initialize the cartesian chart
139             initializeChart();
140             updateDonutChart();
141         }
142
143         /// <summary>
144         /// Force refresh the form each tick of the timer
145         /// Default tickrate : 16ms -> 60fps
146         /// </summary>
147         /// <param name="sender"></param>
148         /// <param name="e"></param>
149         private void MainTimer_Tick(object sender, EventArgs e)
150         {
151             Refresh();
152         }
```

```
153
154         private void pbGrid_Paint(object sender, PaintEventArgs e)
155         {
156             // Draw code here
157             mainGrid.draw(e.Graphics);
158         }
159
160         /// <summary>
161         /// Updates when changing the number of cells horizontally
162         /// </summary>
163         /// <param name="sender"></param>
164         /// <param name="e"></param>
165         private void trackBar1_Scroll(object sender, EventArgs e)
166         {
167             updateGrid();
168         }
169
170         /// <summary>
171         /// Updates when changing the number of cells vertically
172         /// </summary>
173         /// <param name="sender"></param>
174         /// <param name="e"></param>
175         private void trackBar2_Scroll(object sender, EventArgs e)
176         {
177             updateGrid();
178         }
179
180         /// <summary>
181         /// Open the generation form
182         /// </summary>
183         /// <param name="sender"></param>
184         /// <param name="e"></param>
185         private void generateNewBoardToolStripMenuItem_Click(object sender, EventArgs e)
186         {
187             interruptTimer();
188
189             // Pass the grid and list of strategies to the form and open them
190             GenerateView generateView = new GenerateView();
191             generateView.currentGrid = this.mainGrid;
192             generateView.strategies = this.availableStrategies;
193
194             if (generateView.ShowDialog() == DialogResult.OK)
195             {
196                 // The user has validated his input
197                 // Reset the generation count
198                 generation = 0;
199
200                 // Update the GUI
201                 updateLabels();
202                 updateDonutChart();
203                 initializeChart();
204                 mainGrid.setColorMode(ColorMode.Strategy);
205             }
206         }
207
208         /// <summary>
209         /// Open the payoff matrix parameters
210         /// </summary>
211         /// <param name="sender"></param>
212         /// <param name="e"></param>
213         private void payoffMatrixToolStripMenuItem_Click(object sender, EventArgs e)
214         {
215             interruptTimer();
216
217             // Pass the PayoffMatrix object as parameter to the form and open it
218             PayoffMatrixView matrixView = new PayoffMatrixView();
219             matrixView.currentMatrix = this.payoffMatrix;
220
221             if (matrixView.ShowDialog() == DialogResult.Yes)
222             {
223                 // The user has validated his input
224             }
225         }
226
227         /// <summary>
228         /// Open the strategy benchmark window
229         /// </summary>
230         /// <param name="sender"></param>
231         /// <param name="e"></param>
232         private void benchmarkStrategiesToolStripMenuItem_Click(object sender, ↩
                EventArgs e)
233         {
234             interruptTimer();
235
```

```
236             // Pass the PayoffMatrix object as parameter to the form and open it
237             BenchmarkView benchmarkView = new BenchmarkView();
238 
239             // Pass some values for the view
240             benchmarkView.strategies = availableStrategies;
241             benchmarkView.matrix = payoffMatrix;
242 
243             if (benchmarkView.ShowDialog() == DialogResult.OK)
244             {
245 
246             }
247         }
248 
249         /// <summary>
250         /// Open the about window
251         /// </summary>
252         /// <param name="sender"></param>
253         /// <param name="e"></param>
254         private void helpToolStripMenuItem_Click(object sender, EventArgs e)
255         {
256             interruptTimer();
257             AboutView view = new AboutView();
258 
259             if (view.ShowDialog() == DialogResult.OK)
260             {
261                 // The user has validated his input
262             }
263         }
264 
265         /// <summary>
266         /// Update a flag when we click on the grid
267         /// </summary>
268         /// <param name="sender"></param>
269         /// <param name="e"></param>
270         private void pbGrid_MouseDown(object sender, MouseEventArgs e)
271         {
272             isClickingOnGrid = true;
273             updateCellState();
274         }
275 
276         /// <summary>
277         /// Update a flag when we release our click on the grid
278         /// </summary>
279         /// <param name="sender"></param>
280         /// <param name="e"></param>
281         private void pbGrid_MouseUp(object sender, MouseEventArgs e)
282         {
283             isClickingOnGrid = false;
284             updateCellState();
285             updateDonutChart();
286         }
287 
288         /// <summary>
289         /// Updates the clicked cell with its new strategy
290         /// </summary>
291         /// <param name="sender"></param>
292         /// <param name="e"></param>
293         private void pbGrid_MouseMove(object sender, MouseEventArgs e)
294         {
295             mouseX = e.X;
296             mouseY = e.Y;
297             updateCellState();
298         }
299 
300         /// <summary>
301         ///
302         /// </summary>
303         /// <param name="sender"></param>
304         /// <param name="e"></param>
305         private void btnPlayPause_Click(object sender, EventArgs e)
306         {
307             // Change the button's text and launch the timer
308             switchPlayPauseState();
309         }
310 
311         /// <summary>
312         /// Manually steps forward (click)
313         /// </summary>
314         /// <param name="sender"></param>
315         /// <param name="e"></param>
316         private void btnStep_Click(object sender, EventArgs e)
317         {
318             stepForward();
319         }
```

```
320
321         /// <summary>
322         /// Automatically steps forwards
323         /// </summary>
324         /// <param name="sender"></param>
325         /// <param name="e"></param>
326         private void StepTimer_Tick(object sender, EventArgs e)
327         {
328             stepForward();
329         }
330
331         /// <summary>
332         /// Change the autostep speed
333         /// </summary>
334         /// <param name="sender"></param>
335         /// <param name="e"></param>
336         private void tbTimerSpeed_Scroll(object sender, EventArgs e)
337         {
338             StepTimer.Interval = tbTimerSpeed.Value;
339             updateLabels();
340         }
341
342
343         /// <summary>
344         /// Switch back to strategy color mode when we click on the strategy combo box
345         /// </summary>
346         /// <param name="sender"></param>
347         /// <param name="e"></param>
348         private void cbStrategies_Click(object sender, EventArgs e)
349         {
350             // Interrupt the autoplay if it is running
351             interruptTimer();
352             mainGrid.setColorMode(ColorMode.Strategy);
353             updateLabels();
354             Refresh();
355         }
356
357         // Clears the board and fills it with the default cell
358         private void btnClear_Click(object sender, EventArgs e)
359         {
360             updateGrid();
361         }
362
363
364         /// <summary>
365         /// Alternates between normal and extended view
366         /// </summary>
367         /// <param name="sender"></param>
368         /// <param name="e"></param>
369         private void tsExtendedView_CheckedChanged(object sender, EventArgs e)
370         {
371             if (tsExtendedView.Checked)
372             {
373                 // Switch to extended view
374                 this.Width = DEFAULT_EXTENDED_VIEW_WIDTH;
375             }
376             else
377             {
378                 // Switch to normal view
379                 this.Width = DEFAULT_NORMAL_VIEW_WIDTH;
380             }
381         }
382
383         /// <summary>
384         /// Alternates between default and torus wrapping mode
385         /// </summary>
386         /// <param name="sender"></param>
387         /// <param name="e"></param>
388         private void tsWrapMode_CheckedChanged(object sender, EventArgs e)
389         {
390             if (tsWrapMode.Checked)
391             {
392                 gridWrappingMode = WrapMode.Torus;
393             }
394             else
395             {
396                 gridWrappingMode = WrapMode.Default;
397             }
398
399             // Reset the grid to regenerate the neighbors lists
400             updateGrid();
401         }
402
403         /*************************************************************************
```

```
404          *                                  FUNCTIONS                                  *
405          ***************************************************************************/
406         /// <summary>
407         /// Switch the states between play and pause
408         /// </summary>
409         public void switchPlayPauseState()
410         {
411             if (isAutoplaying)
412             {
413                 btnPlayPause.Text = "4";
414                 StepTimer.Stop();
415             }
416             else
417             {
418                 btnPlayPause.Text = ";";
419                 StepTimer.Start();
420             }
421
422             // Invert the state
423             isAutoplaying = !isAutoplaying;
424         }
425
426
427         /// <summary>
428         /// Steps forward in time
429         /// </summary>
430         private void stepForward()
431         {
432             // Steps forward
433             mainGrid.step();
434             mainGrid.setColorMode(ColorMode.Playing);
435
436             // Increment the generation count
437             generation++;
438
439             // Update the GUI
440             updateLabels();
441             updateDonutChart();
442             addDataToChart();
443         }
444
445
446
447         /// <summary>
448         /// Pause the "autostep" timer if it is running
449         /// </summary>
450         public void interruptTimer()
451         {
452             if (isAutoplaying)
453             {
454                 switchPlayPauseState();
455             }
456         }
457
458         /// <summary>
459         /// Updates the labels with new information
460         /// </summary>
461         private void updateLabels()
462         {
463             // Trackbar labels
464             lblLines.Text = String.Format("Rows : {0}", tbLines.Value);
465             lblCols.Text = String.Format("Columns : {0}", tbColumns.Value);
466
467             // Grid label
468             lblGridInfo.Text = String.Format("{0}x{1} Grid - Mode : {2} - Generation ↩
                 {3}", tbLines.Value, tbColumns.Value, mainGrid.ColorMode.ToString(), ↩
                 generation);
469
470             // Speed labels
471             lblSpeedValue.Text = "automatically steps every " + tbTimerSpeed.Value + " ↩
                 [ms]";
472         }
473
474         /// <summary>
475         /// If the user is clicking on the grid, update the cell under the user's cursor
476         /// </summary>
477         public void updateCellState()
478         {
479             if (isClickingOnGrid)
480             {
481                 Strategy selectedStrategy = ↩
                     availableStrategies[cbStrategies.SelectedIndex];
482                 this.mainGrid.onClick(mouseX, mouseY, selectedStrategy);
483
```

```
484                 // Interrupt the autoplay if it is running
485                 interruptTimer();
486 
487                 // Change the color mode
488                 mainGrid.setColorMode(ColorMode.Strategy);
489                 updateLabels();
490                 Refresh();
491             }
492         }
493 
494         /// <summary>
495         /// Updates the grid with new values (Re-create the grid)
496         /// </summary>
497         private void updateGrid()
498         {
499             // Interrupt the autoplay if it is running
500             interruptTimer();
501             mainGrid = new Grid(pbGrid.Width, pbGrid.Height, tbLines.Value, ↩
                    tbColumns.Value, payoffMatrix, gridWrappingMode);
502 
503             // Reset the generation count
504             generation = 0;
505 
506             // Update the labels and chart
507             updateLabels();
508             updateDonutChart();
509             initializeChart();
510         }
511 
512         /// <summary>
513         /// Updates the donut chart on the main view
514         /// </summary>
515         private void updateDonutChart()
516         {
517             // Update the donut chart
518             int count = 0;
519             foreach (Series serie in pieStrategy.Series)
520             {
521                 if (mainGrid.findCountOfStrategy(availableStrategies[count]) > 0)
522                 {
523                     serie.Visibility = System.Windows.Visibility.Visible;
524                     serie.Values = new ChartValues<double> { ↩
                            mainGrid.findCountOfStrategy(availableStrategies[count]) };
525                 }
526                 else
527                 {
528                     serie.Visibility = System.Windows.Visibility.Hidden;
529                 }
530 
531                 count++;
532             }
533         }
534 
535         /// <summary>
536         /// Initialize the cartesian chart with the base values
537         /// </summary>
538         public void initializeChart()
539         {
540             cartesianStrategy.Series = new SeriesCollection();
541             foreach (Strategy strategy in availableStrategies)
542             {
543                 // Get the color from the strategy
544                 System.Windows.Media.BrushConverter converter = new ↩
                        System.Windows.Media.BrushConverter();
545                 System.Windows.Media.Brush brush = ↩
                        (System.Windows.Media.Brush)converter.ConvertFromString(strategy.getColor().ToHex(60
546                 System.Windows.Media.Brush stroke = ↩
                        (System.Windows.Media.Brush)converter.ConvertFromString(strategy.getColor().ToHex()
547 
548                 // Create an object for storing values on the line chart
549                 LineSeries stratToAdd = new LineSeries
550                 {
551                     Title = strategy.ToString(),
552                     Values = new ChartValues<double> { 0 },
553                     PointGeometry = DefaultGeometries.None,
554                     PointGeometrySize = 15,
555                     Fill = brush,
556                     Stroke = stroke
557                 };
558 
559                 // Hide the unused strategies
560                 if (mainGrid.findCountOfStrategy(strategy) <= 0)
561                 {
562                     stratToAdd.Visibility = System.Windows.Visibility.Hidden;
```

```
563                 }
564
565                 cartesianStrategy.AxisX[0].MinValue = 0;
566                 cartesianStrategy.AxisX[0].MaxValue = MAX_NB_ELEMENTS_IN_CHART;
567
568                 // Add the values to the pie chart
569                 cartesianStrategy.Series.Add(stratToAdd);
570             }
571         }
572
573         /// <summary>
574         /// Adds data to the cartesian chart
575         /// </summary>
576         private void addDataToChart()
577         {
578             int count = 0;
579
580             // Readjust the X axis
581             cartesianStrategy.AxisX[0].MaxValue = generation;
582             if (generation > MAX_NB_ELEMENTS_IN_CHART)
583             {
584                 cartesianStrategy.AxisX[0].MinValue = generation - ↩
                        MAX_NB_ELEMENTS_IN_CHART;
585             }
586
587             foreach (Series serie in cartesianStrategy.Series)
588             {
589
590                 // Check the currently used strategies
591                 if (mainGrid.findCountOfStrategy(availableStrategies[count]) > 0)
592                 {
593                     // Add the average score of each used strategy
594                     serie.Values.Add(mainGrid.findAvgScoreOfStrategy(availableStrategies[count]));
595                     serie.Visibility = System.Windows.Visibility.Visible;
596                 }
597                 else
598                 {
599                     // Add 0 to the unused values (allows the graph to stay synced)
600                     serie.Values.Add((double)0);
601                     serie.Visibility = System.Windows.Visibility.Hidden;
602                 }
603
604                 count++;
605             }
606         }
607
608         /// <summary>
609         /// Save the current grid in a serialized format
610         /// </summary>
611         /// <param name="sender"></param>
612         /// <param name="e"></param>
613         private void saveGridToolStripMenuItem_Click(object sender, EventArgs e)
614         {
615             // Open a file dialog for the user to save the file
616             SaveFileDialog sfd = new SaveFileDialog();
617             sfd.Filter = "XML files|*.xml";
618
619             if (sfd.ShowDialog() == DialogResult.OK)
620             {
621                 // Save the data to the path
622                 mainGrid.saveData(sfd.FileName);
623
624                 // Notify the user
625                 MessageBox.Show("Grid exported successfully");
626             }
627         }
628
629         /// <summary>
630         /// Load the current grid from a serialized file
631         /// </summary>
632         /// <param name="sender"></param>
633         /// <param name="e"></param>
634         private void loadGridToolStripMenuItem_Click(object sender, EventArgs e)
635         {
636             // Open a file dialog for the user to load the file
637             OpenFileDialog ofd = new OpenFileDialog();
638             ofd.Filter = "XML files|*.xml";
639
640             if (ofd.ShowDialog() == DialogResult.OK)
641             {
642                 // Load the data from the path
643                 mainGrid.loadData(ofd.FileName);
644
645                 // Update the trackbars manually
```

```
646                 tbLines.Value = mainGrid.NbLines;
647                 tbColumns.Value = mainGrid.NbCols;
648                 this.updateLabels();
649 
650                 // Notify the user
651                 MessageBox.Show("Grid loaded successfully");
652             }
653         }
654     }
655 }
```

1.5 PayoffMatrixHelpView.cs

```
1     /*
2      Class            :   PayoffMatrixHelpView.cs
3      Description      :   Gives help to the user on payoff matrixes
4      Author           :   SEEMULLER Julien
5      Date             :   10.04.2017
6 */
7 
8 using System;
9 using System.Windows.Forms;
10 
11 namespace PrisonersDilemmaCA
12 {
13     public partial class PayoffMatrixHelpView : Form
14     {
15         public PayoffMatrixHelpView()
16         {
17             InitializeComponent();
18         }
19 
20         /// <summary>
21         /// Quit the help form
22         /// </summary>
23         /// <param name="sender"></param>
24         /// <param name="e"></param>
25         private void btnOk_Click(object sender, EventArgs e)
26         {
27             this.Close();
28         }
29     }
30 }
```

1.6 PayoffMatrixView.cs

```
1     /*
2      Class            :   PayoffMatrixView.cs
3      Description      :   Allows the user to interract with the payoff matrix
4      Author           :   SEEMULLER Julien
5      Date             :   10.04.2017
6 */
7 
8 using System;
9 using System.ComponentModel;
10 using System.Windows.Forms;
11 
12 namespace PrisonersDilemmaCA
13 {
14     public partial class PayoffMatrixView : Form
15     {
16         public PayoffMatrix currentMatrix { get; set; }
17 
18         public PayoffMatrixView()
19         {
20             InitializeComponent();
21         }
22 
23         private void PayoffMatrixView_Load(object sender, EventArgs e)
24         {
25             // Initialize our textboxes with the matrix data
26             rtbReward.Text = currentMatrix.Reward.ToString();
27             rtbSucker.Text = currentMatrix.Sucker.ToString();
28             rtbTemptation.Text = currentMatrix.Temptation.ToString();
29             rtbPunishment.Text = currentMatrix.Punishment.ToString();
```

```
30          }
31
32          // Apply the changes and quit
33          private void btnOk_Click(object sender, EventArgs e)
34          {
35              // Store the contents of the textboxes as integers
36              int t = Convert.ToInt32(rtbTemptation.Text);
37              int r = Convert.ToInt32(rtbReward.Text);
38              int p = Convert.ToInt32(rtbPunishment.Text);
39              int s = Convert.ToInt32(rtbSucker.Text);
40
41              // Check for matrix validity
42              // T < R < P < S
43              if (!(PayoffMatrix.isValid(t, r, p, s)))
44              {
45                  // If it is not valid we abort and tell the user
46                  MessageBox.Show(
47                          "The selected matrix is not valid."
48                          + Environment.NewLine
49                          + "Rules :"
50                          + Environment.NewLine
51                          + "[Temptation < Reward < Punishment < Sucker]"
52                          + Environment.NewLine
53                          + "[2 * Reward < Temptation + Sucker]"
54                      );
55              }
56              else
57              {
58                  // Else, we apply the changes
59                  currentMatrix.Reward = r;
60                  currentMatrix.Sucker = s;
61                  currentMatrix.Temptation = t;
62                  currentMatrix.Punishment = p;
63
64                  // Close the form
65                  this.Close();
66              }
67          }
68
69          // Cancel and quit
70          private void btnCancel_Click(object sender, EventArgs e)
71          {
72              this.Close();
73          }
74
75          private void PayoffMatrixView_HelpButtonClicked(object sender, CancelEventArgs e)
76          {
77              PayoffMatrixHelpView helpForm = new PayoffMatrixHelpView();
78
79              if (helpForm.ShowDialog() == DialogResult.OK)
80              {
81                  // User clicked on ok
82              }
83          }
84
85      }
86  }
```

2 Classes

2.1 Cell.cs

```
1    /*
2     Class           :   Cell.cs
3     Description     :   Main class, represents one "player" of the prisoner's dilemma
4     Author          :   SEEMULLER Julien
5     Date            :   10.04.2017
6 */
7
8 using System;
9 using System.Collections.Generic;
10 using System.Drawing;
11 using System.Linq;
12 using System.Xml;
13 using System.Xml.Schema;
14 using System.Xml.Serialization;
15
16 namespace PrisonersDilemmaCA
17 {
18     public class Cell : IXmlSerializable
19     {
20         #region fields
21         #region consts
22         public static readonly Strategy DEFAULT_STRATEGY = new StratTitForTat();
23         public const int DEFAULT_BORDER_WIDTH = 1;
24         private const int DEFAULT_X = 0;
25         private const int DEFAULT_Y = 0;
26         #endregion
27
28         private int _x;                      // X position in the grid (should be ↩
               multiplied by width if used for graphics)
29         private int _y;                      // Y position in the grid (should be ↩
               multiplied by height if used for graphics)
30         private int _width;                  // Width of the cell (dependent on the grid)
31         private int _height;                 // Height of the cell (dependent on the grid)
32         private int _score;                  // Represents the number of days in prison
33         private Strategy _strategy;          // The strategy used by the cell (ex : Tit ↩
               for Tat)
34         private Color _color;                // The current color of the cell
35         private List<Cell> _neighbors;       // A list of references to the cells ↩
               neighbors
36         private PayoffMatrix _payoffMatrix; // The payoff matrix used by the cell
37         private Move _choice;                  // What the cell intends to do this turn ↩
               (ex : Defect)
38         private Stack<Move> _history;        // Complete history of the cell's actions ↩
               (ex : C, C, C, D, C, D, etc...)
39         #endregion
40
41         #region properties
42         public int X
43         {
44             get { return _x; }
45             set { _x = value; }
46         }
47
48         public int Y
49         {
50             get { return _y; }
51             set { _y = value; }
52         }
53
54         public int Width
55         {
56             get { return _width; }
57             set { _width = value; }
58         }
59
60         public int Height
61         {
62             get { return _height; }
63             set { _height = value; }
64         }
65
66         public int Score
67         {
68             get { return _score; }
69             set { _score = value; }
70         }
71
```

```
72          public Strategy Strategy
73          {
74              get { return _strategy; }
75              set
76              {
77                  // Make sure it is a new instance of the strategy
78                  _strategy = (Strategy)Activator.CreateInstance(value.GetType());
79                  // Set the color when we change the strategy
80                  this.Color = this.Strategy.getColor();
81              }
82          }
83
84          public PayoffMatrix PayoffMatrix
85          {
86              get { return _payoffMatrix; }
87              set { _payoffMatrix = value; }
88          }
89
90          public Color Color
91          {
92              get { return _color; }
93              set { _color = value; }
94          }
95
96          public List<Cell> Neighbors
97          {
98              get { return _neighbors; }
99              set { _neighbors = value; }
100         }
101
102         private Move Choice
103         {
104             get { return _choice; }
105             set { _choice = value; }
106         }
107
108         public Stack<Move> History
109         {
110             get { return _history; }
111             set { _history = value; }
112         }
113         #endregion
114
115         #region constructors
116         /// <summary>
117         /// Designated constructor
118         /// </summary>
119         /// <param name="x">X coordinate of the cell on the grid</param>
120         /// <param name="y">Y coordinate of the cell on the grid</param>
121         /// <param name="strategy">Current strategy of the cell</param>
122         /// <param name="matrix">Payoff matrix used to determine the score of each ↩
                cell</param>
123         public Cell(int x, int y, Strategy strategy, PayoffMatrix matrix)
124         {
125             this.X = x;
126             this.Y = y;
127             this.Strategy = strategy;
128             this.PayoffMatrix = matrix;
129             this.Score = 0;
130
131             this.Neighbors = new List<Cell>();
132             this.History = new Stack<Move>();
133
134             // Get the color of the cell from the current strategy
135             this.setColorFromStrategy();
136
137             // Starts with a move relevent to the strategy
138             this.chooseNextMove();
139         }
140
141         /// <summary>
142         /// Conveniance constructor
143         /// </summary>
144         /// <param name="x"></param>
145         /// <param name="y"></param>
146         public Cell(int x, int y, PayoffMatrix matrix)
147             : this(x, y, DEFAULT_STRATEGY, matrix)
148         {
149             // No code
150         }
151
152         /// <summary>
153         /// Default constructor
154         /// </summary>
```

```
155         public Cell()
156             : this(DEFAULT_X, DEFAULT_Y, new PayoffMatrix())
157         {
158 
159         }
160         #endregion
161 
162         #region methods
163         /// <summary>
164         /// Plays a game of the prisoners dilemma with the cell's neighbors using the ↩
                cell's current strategy
165         /// </summary>
166         public void step()
167         {
168             // Go and play with each of our neighbors
169             List<int> scores = new List<int>();
170             foreach (Cell neighbor in this.Neighbors)
171             {
172                 // Play a game and store the result
173                 scores.Add(PayoffMatrix.returnPayoff(this.Choice, neighbor.Choice));
174             }
175 
176             // We get the best score of the cell
177             this.Score = scores.Min();
178 
179             // Update the color of the cell
180             this.setColorFromMove();
181         }
182 
183         /// <summary>
184         /// Choose the next move using our strategy and neighbors
185         /// </summary>
186         public void chooseNextMove()
187         {
188             this.Choice = this.Strategy.chooseMove(this, this.Neighbors);
189         }
190 
191         /// <summary>
192         /// Updates the last move of the cell
193         /// </summary>
194         public void updateLastMove()
195         {
196             this.History.Push(this.Choice);
197         }
198 
199         /// <summary>
200         /// Function used to draw the cell
201         /// </summary>
202         /// <param name="g">The graphical element we use to draw</param>
203         public void draw(Graphics g)
204         {
205             // Color of the cell
206             SolidBrush cellColor = new SolidBrush(this.Color);
207 
208             // Border parameters (color, width)
209             Pen borderColor = new Pen(Color.Black, DEFAULT_BORDER_WIDTH);
210 
211             // Draw the cell
212             g.FillRectangle(cellColor, this); // Implicitly converted as a rectangle
213             g.DrawRectangle(borderColor, this);
214         }
215 
216 
217         /// <summary>
218         /// Implicit conversion to rectangle to simplify other functions
219         /// </summary>
220         /// <param name="cell">The cell used for conversion</param>
221         /// <returns></returns>
222         public static implicit operator Rectangle(Cell cell)
223         {
224             return new Rectangle(cell.X * cell.Width, cell.Y * cell.Height, ↩
                    cell.Width, cell.Height);
225         }
226 
227         /// <summary>
228         /// On click, we update the cell's strategy with a new one
229         /// </summary>
230         /// <param name="x">The x coordinate in pixels</param>
231         /// <param name="y">The y coordinate in pixels</param>
232         public void onClick(int x, int y, Strategy strat)
233         {
234             Rectangle hitbox = this;
235 
236             // If we are the cell that is hit, update our strategy and clear it's history
```

```
237             if (hitbox.Contains(x, y))
238             {
239                 updateStrategy(strat);
240             }
241         }
242 
243         /// <summary>
244         /// Updates the strategy of the cell
245         /// </summary>
246         /// <param name="strat">The strategy to update the cell with</param>
247         public void updateStrategy(Strategy strat)
248         {
249             // Change the strategy
250             this.Strategy = strat;
251 
252             // Updates the cell's move with the new strategy
253             this.History.Clear();
254 
255             // We play a game with our neighbors to sync with the current game
256             this.chooseNextMove();
257             this.updateLastMove();
258             this.step();
259         }
260 
261         /// <summary>
262         /// Set the color of the cell according to its next move
263         /// </summary>
264         public void setColorFromMove()
265         {
266             switch (this.Choice)
267             {
268                 case Move.Cooperate:
269                     if (this.History.First() == Move.Defect)
270                     {
271                         this.Color = Color.FromArgb(230, 126, 34); // ORANGE
272                     }
273                     else
274                     {
275                         this.Color = Color.FromArgb(46, 204, 113); // GREEN
276                     }
277                     break;
278 
279 
280                 case Move.Defect:
281                     if (this.History.First() == Move.Cooperate)
282                     {
283                         this.Color = Color.FromArgb(241, 196, 15); // YELLOW
284                     }
285                     else
286                     {
287                         this.Color = Color.FromArgb(192, 57, 43); // RED
288                     }
289 
290                     break;
291             }
292         }
293 
294         /// <summary>
295         /// Set the color of the cell according to its strategy
296         /// </summary>
297         public void setColorFromStrategy()
298         {
299             this.Color = this.Strategy.getColor();
300         }
301 
302 
303 
304 
305 
306         //****************************//
307         // INTERFACE IXMLSERIALIZABLE //
308         //****************************//
309 
310         /// <summary>
311         /// Unused, see MSDN documentation :
312         /// "This method is reserved and should not be used. It should always return a ↩
                null value"
313         /// </summary>
314         /// <returns></returns>
315         public XmlSchema GetSchema()
316         {
317             return null;
318         }
319 
```

```
320
321         /// <summary>
322         /// Reads through a serialized XML file to get the values for a cell
323         /// </summary>
324         /// <param name="reader">The XML reader attached to the serialized file</param>
325         public void ReadXml(XmlReader reader)
326         {
327
328             int R = -1;
329             int G = -1;
330             int B = -1;
331
332             reader.Read(); // Skip the beggining tab
333             if (reader.Name == "X")
334             {
335                 reader.Read(); // Read past the name tag
336                 this.X = int.Parse(reader.Value);
337                 reader.Read(); // Read past the value
338
339             }
340             reader.Read(); // Read past the closing tag
341
342             // repeat this process for every value...
343
344             if (reader.Name == "Y")
345             {
346                 reader.Read();
347                 this.Y = int.Parse(reader.Value);
348                 reader.Read();
349             }
350             reader.Read();
351
352             if (reader.Name == "Width")
353             {
354                 reader.Read();
355                 this.Width = int.Parse(reader.Value);
356                 reader.Read();
357             }
358             reader.Read();
359
360             if (reader.Name == "Height")
361             {
362                 reader.Read();
363                 this.Height = int.Parse(reader.Value);
364                 reader.Read();
365             }
366             reader.Read();
367
368             if (reader.Name == "Strategy")
369             {
370                 reader.Read();
371                 // Create a new instance of the strategy
372                 Type elementType = Type.GetType(reader.Value);
373                 this.Strategy = (Strategy)Activator.CreateInstance(elementType);
374                 reader.Read();
375             }
376             reader.Read();
377
378
379             if (reader.Name == "R")
380             {
381                 reader.Read();
382                 R = int.Parse(reader.Value);
383                 reader.Read();
384             }
385             reader.Read();
386
387             if (reader.Name == "G")
388             {
389                 reader.Read();
390                 G = int.Parse(reader.Value);
391                 reader.Read();
392             }
393             reader.Read();
394
395             if (reader.Name == "B")
396             {
397                 reader.Read();
398                 B = int.Parse(reader.Value);
399                 reader.Read();
400             }
401             reader.Read();
402
403             // Check if the RGB values are assigned
```

```
404             if (R > 0 && G > 0 && B > 0)
405             {
406                 // Create a color
407                 this.Color = Color.FromArgb(R, G, B);
408 
409                 // Reset the color
410                 R = -1;
411                 G = -1;
412                 B = -1;
413             }
414 
415             if (reader.Name == "Score")
416             {
417                 reader.Read();
418                 // Tries to parse the reader value as a "Move" enum
419                 this.Score = int.Parse(reader.Value);
420                 reader.Read();
421             }
422             reader.Read();
423             reader.Read(); // Skip ending tag
424         }
425 
426 
427         /// <summary>
428         /// Write the cell's value to a serialized XML file
429         /// </summary>
430         /// <param name="writer">The XML reader attached to the serialized file</param>
431         public void WriteXml(XmlWriter writer)
432         {
433             // Set color from strategy before continuing
434             setColorFromStrategy();
435 
436             // Write the content of the cell to xml format
437             writer.WriteStartElement("X");
438             writer.WriteString(this.X.ToString());
439             writer.WriteEndElement();
440 
441             writer.WriteStartElement("Y");
442             writer.WriteString(this.Y.ToString());
443             writer.WriteEndElement();
444 
445             writer.WriteStartElement("Width");
446             writer.WriteString(this.Width.ToString());
447             writer.WriteEndElement();
448 
449             writer.WriteStartElement("Height");
450             writer.WriteString(this.Height.ToString());
451             writer.WriteEndElement();
452 
453             writer.WriteStartElement("Strategy");
454             writer.WriteString(this.Strategy.GetType().ToString());
455             writer.WriteEndElement();
456 
457             writer.WriteStartElement("R");
458             writer.WriteString(this.Color.R.ToString());
459             writer.WriteEndElement();
460 
461             writer.WriteStartElement("G");
462             writer.WriteString(this.Color.G.ToString());
463             writer.WriteEndElement();
464 
465             writer.WriteStartElement("B");
466             writer.WriteString(this.Color.B.ToString());
467             writer.WriteEndElement();
468 
469             writer.WriteStartElement("Score");
470             writer.WriteString(this.Score.ToString());
471             writer.WriteEndElement();
472 
473             /* HISTORY - UNUSED, INCREASED SIZE OF FILE EXPONENTIALLY WITH EACH ↩
                    GENERATION
474             writer.WriteStartElement("History");
475             foreach (Move choice in this.History)
476             {
477                 writer.WriteStartElement("Choice");
478                 writer.WriteString(choice.ToString());
479                 writer.WriteEndElement();
480             }
481             writer.WriteEndElement();
482             */
483         }
484         #endregion
485     }
486 }
```

2.2 Grid.cs

```
   /*
    Class           :   Grid.cs
    Description     :   Stores the cells of the cellular automaton,
                        main model of the cellular automaton
    Author          :   SEEMULLER Julien
    Date            :   10.04.2017
*/

using System;
using System.Collections.Generic;
using System.Drawing;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Xml.Serialization;

namespace PrisonersDilemmaCA
{
    public class Grid
    {
        #region fields

        #region consts
        public const int NEAREST_NEIGHBOR_RANGE = 1;    // Change the "radius" at which we consider cells neighbors
        private const int DEFAULT_HEIGHT = 100;
        private const int DEFAULT_WIDTH = 100;
        private const int DEFAULT_NB_COLS = 10;
        private const int DEFAULT_NB_LINES = 10;
        private const string DEFAULT_DATA_FILEPATH = "xml/grid.xml";

        public const WrapMode DEFAULT_WRAP_MODE = WrapMode.Torus;
        #endregion

        private Cell[,] _cells;                         // 2D array containing the cells
        private int _width;                             // Width of the grid in pixels
        private int _height;                            // Height of the grid in pixels
        private int _nbLines;                           // Number of lines in the grid (y)
        private int _nbCols;                            // Number of columns in the grid (x)
        private PayoffMatrix _payoffMatrix;             // Payoff matrix to be distributed to cells
        private ColorMode _colorMode;                   // The current color mode of the grid (cf. ColorMode enum)
        private WrapMode _wrapMode;                     // The current wrapping mode of the grid (cf. WrapMode enum)
        private List<Cell> _serializableCells;          // Since [,] is not serializable, we make a list of cell before serializing.
        #endregion

        #region properties
        [XmlIgnore]
        public Cell[,] Cells
        {
            get { return _cells; }
            set { _cells = value; }
        }

        public List<Cell> SerializableCells
        {
            get { return _serializableCells; }
            set { _serializableCells = value; }
        }

        public int Width
        {
            get { return _width; }
            set { _width = value; }
        }

        public int Height
        {
            get { return _height; }
            set { _height = value; }
```

```
70          }
71
72          public int NbCols
73          {
74              get { return _nbCols; }
75              set { _nbCols = value; }
76          }
77
78          public int NbLines
79          {
80              get { return _nbLines; }
81              set { _nbLines = value; }
82          }
83
84          public PayoffMatrix PayoffMatrix
85          {
86              get { return _payoffMatrix; }
87              set { _payoffMatrix = value; }
88          }
89
90          public ColorMode ColorMode
91          {
92              get { return _colorMode; }
93              set { _colorMode = value; }
94          }
95
96          public WrapMode WrapMode
97          {
98              get { return _wrapMode; }
99              set { _wrapMode = value; }
100         }
101         #endregion
102
103         #region constructors
104         /// <summary>
105         /// Designated constructor
106         /// </summary>
107         /// <param name="width">The width of the grid in pixels</param>
108         /// <param name="height">The height of the grid in pixels</param>
109         /// <param name="nbCols">The number of columns of the grid</param>
110         /// <param name="nbLines">The number of lines of the grid</param>
111         public Grid(int width, int height, int nbLines, int nbCols, PayoffMatrix ↩
                matrix, WrapMode wrapmode, Strategy strategy)
112         {
113             this.Width = width;
114             this.Height = height;
115             this.NbLines = nbLines;
116             this.NbCols = nbCols;
117             this.PayoffMatrix = matrix;
118             this.ColorMode = ColorMode.Strategy;
119             this.WrapMode = wrapmode;
120
121             // Initialize our list of cells
122             this.Cells = new Cell[nbLines, nbCols];
123
124             // Calculate the width and the height of a cell
125             int cellWidth = this.Width / nbCols;
126             int cellHeight = this.Height / nbLines;
127
128             // Go through each possible slot in the grid
129             for (int y = 0; y < this.NbLines; y++)
130             {
131                 for (int x = 0; x < this.NbCols; x++)
132                 {
133                     // Create a temporary cell with the default strategy
134                     Cell tmpCell = new Cell(x, y, strategy, this.PayoffMatrix);
135
136                     // Set the cell's height according to the grid's need
137                     tmpCell.Width = cellWidth;
138                     tmpCell.Height = cellHeight;
139
140                     // Add the cell to the list
141                     this.Cells[y, x] = tmpCell;
142                 }
143             }
144
145             foreach (Cell cell in this.Cells)
146             {
147                 // Make each cell aware of its neighbors
148                 cell.Neighbors = findCellNeighbors(cell);
149             }
150         }
151
152         /// <summary>
```

```
153         /// Conveniance constructor
154         /// </summary>
155         public Grid(int width, int height, int nbLines, int nbCols, PayoffMatrix ↩
                matrix, WrapMode wrapmode)
156             : this(width, height, nbLines, nbCols, matrix, wrapmode, ↩
                    Cell.DEFAULT_STRATEGY)
157         {
158             // No code
159         }
160 
161         /// <summary>
162         /// Conveniance constructor 2
163         /// </summary>
164         public Grid(int width, int height, int nbLines, int nbCols, PayoffMatrix matrix)
165             : this(width, height, nbLines, nbCols, matrix, DEFAULT_WRAP_MODE, ↩
                    Cell.DEFAULT_STRATEGY)
166         {
167             // No code
168         }
169 
170         /// <summary>
171         /// Conveniance constructor 3
172         /// </summary>
173         public Grid(int width, int height, int nbLines, int nbCols)
174             : this(width, height, nbLines, nbCols, new PayoffMatrix(), ↩
                    DEFAULT_WRAP_MODE, Cell.DEFAULT_STRATEGY)
175         {
176             // No code
177         }
178 
179         /// <summary>
180         /// Default constructor
181         /// (Required for serialization)
182         /// </summary>
183         public Grid()
184             : this(DEFAULT_WIDTH, DEFAULT_HEIGHT, DEFAULT_NB_LINES, DEFAULT_NB_COLS, ↩
                    new PayoffMatrix())
185         {
186             // No code
187         }
188         #endregion
189 
190         #region methods
191 
192         /// <summary>
193         /// Steps forward in time
194         /// </summary>
195         public void step()
196         {
197             // Store each of the cell's last move
198             foreach (Cell cell in this.Cells)
199             {
200                 cell.updateLastMove();
201             }
202 
203             // Choose each of the cell's next move
204             foreach (Cell cell in this.Cells)
205             {
206                 cell.chooseNextMove();
207             }
208 
209             // Step forward (play the game)
210             foreach (Cell cell in this.Cells)
211             {
212                 cell.step();
213             }
214         }
215 
216         /// <summary>
217         /// Draw every cell on the board and the grid around them
218         /// </summary>
219         /// <param name="g">The graphics element we draw on</param>
220         public void draw(Graphics g)
221         {
222             // Draw each cell
223             foreach (Cell cell in this.Cells)
224             {
225                 cell.draw(g);
226             }
227 
228             // Avoid drawing errors due to rounding
229             Pen borderColor = new Pen(Color.Black, Cell.DEFAULT_BORDER_WIDTH * 2);
230             g.DrawLine(borderColor, 0, this.Height, this.Width, this.Height);
231             g.DrawLine(borderColor, this.Width, 0, this.Width, this.Height);
```

```
232         }
233
234         /// <summary>
235         /// Generates a board of cell from a dictionary of strategy and percentages
236         /// </summary>
237         /// <param name="strategyAndPercentages">Dictionary countaining the strategies ↩
                and their percentage of appearence</param>
238         public void generate(Dictionary<Strategy, int> strategyAndPercentages)
239         {
240             // Create a new random number generator
241             Random rng = new Random();
242
243             // Create a list of a hundred elements representing the repartition of ↩
                    strategies
244             List<Strategy> strategyPopulation = new List<Strategy>();
245
246             // Go through each possible strategy and percentage
247             foreach (var strat in strategyAndPercentages)
248             {
249                 // Fill the list with the current strategy the same number of times as ↩
                        the percentage
250                 for (int i = 0; i < strat.Value; i++)
251                 {
252                     strategyPopulation.Add(strat.Key);
253                 }
254             }
255
256             // Go through each cell in the grid
257             foreach (Cell cell in this.Cells)
258             {
259                 // Choose a random strategy in the list and apply it to the current cell
260                 int rnd = rng.Next(strategyPopulation.Count);
261                 cell.updateStrategy(strategyPopulation[rnd]);
262             }
263         }
264
265         /// <summary>
266         /// Generates a board of cell from a list of strategy and percentages
267         /// </summary>
268         /// <param name="strats">List of strategies</param>
269         /// <param name="percentages">List of percentages</param>
270         public void generate(List<Strategy> strats, List<int> percentages)
271         {
272             // Fill a dictionary with strategies and percentages
273             Dictionary<Strategy, int> stratAndPercentage = new Dictionary<Strategy, ↩
                    int>();
274
275             int counter = 0;
276             foreach (var strategy in strats)
277             {
278                 stratAndPercentage.Add(strategy, percentages[counter]);
279             }
280
281             // Generate the board
282             this.generate(stratAndPercentage);
283         }
284
285         /// <summary>
286         /// Gets the cell at the given position in a toroidal fashion
287         /// </summary>
288         /// <param name="x">The x coordinate of the cell (on the board)</param>
289         /// <param name="y">The y coordinate of the cell (on the board)</param>
290         /// <returns></returns>
291         public Cell getCell(int x, int y)
292         {
293             // Find the corrisponding point in a toroidal fashion if we go out of bounds
294             Point point = getPointClampedInGrid(x, y);
295             int newX = point.X;
296             int newY = point.Y;
297
298             // Return the correct cell
299             return this.Cells[newY, newX];
300         }
301
302         /// <summary>
303         /// Gets a point and wraps around in a toroidal fashion if the point is out of ↩
                bounds.
304         /// The coordinates are in grid format (see nbLines, nbCols)
305         /// </summary>
306         /// <param name="x">The x coordinate of a point on the grid</param>
307         /// <param name="y">The y coordinate of a point on the grid</param>
308         /// <returns></returns>
309         public Point getPointClampedInGrid(int x, int y)
310         {
```

```
311             int newX = x;
312             int newY = y;
313 
314             // Check if we are out of bounds width-wise
315             if (newX >= this.NbCols)
316             {
317                 // ex : 20 -> 20 - width
318                 newX = newX - Convert.ToInt32(this.NbCols);
319             }
320 
321             if (newX < 0)
322             {
323                 // ex : -2 -> width - 2
324                 newX = Convert.ToInt32(this.NbCols) + newX;
325             }
326 
327             // Check if we are out of bounds height-wise
328             if (newY >= this.NbLines)
329             {
330                 // ex : 20 -> 20 - height
331                 newY = newY - Convert.ToInt32(this.NbLines);
332             }
333 
334             if (newY < 0)
335             {
336                 // ex : -2 -> height - 2
337                 newY = Convert.ToInt32(this.NbLines) + newY;
338             }
339 
340             return new Point(newX, newY);
341         }
342 
343 
344         /// <summary>
345         /// Find the current cell's nearest neighbors (default 8 per cell)
346         /// </summary>
347         /// <param name="cell">The cell used to search for neighbors</param>
348         /// <returns></returns>
349         public List<Cell> findCellNeighbors(Cell cell)
350         {
351             List<Cell> neighbors = new List<Cell>();
352 
353             // Go all around the cell to find its neighbors
354             for (int y = cell.Y - NEAREST_NEIGHBOR_RANGE; y <= cell.Y + ↩
                    NEAREST_NEIGHBOR_RANGE; y++)
355             {
356                 for (int x = cell.X - NEAREST_NEIGHBOR_RANGE; x <= cell.X + ↩
                        NEAREST_NEIGHBOR_RANGE; x++)
357                 {
358                     // Avoid our own cell
359                     if (!((x == cell.X) && (y == cell.Y)))
360                     {
361                         // Add the neighbor depending on the mode
362                         switch (this.WrapMode)
363                         {
364                             case WrapMode.Default:
365                                 // In default mode, check if we are inside the grid
366                                 if ((x >= 0) && (y >= 0) && (x < this.NbCols) && (y < ↩
                                        this.NbLines))
367                                 {
368                                     neighbors.Add(this.getCell(x, y));
369                                 }
370                                 break;
371 
372                             case WrapMode.Torus:
373                                 neighbors.Add(this.getCell(x, y));
374                                 break;
375                         }
376 
377                     }
378                 }
379             }
380 
381             return neighbors;
382         }
383 
384 
385         /// <summary>
386         /// Update the strategy of the cell that has been hit by the cursor
387         /// </summary>
388         /// <param name="x">The x coordinate in pixels</param>
389         /// <param name="y">The y coordinate in pixels</param>
390         /// <param name="strat">The strategy to apply to the cell if it is hit</param>
391         public void onClick(int x, int y, Strategy strat)
```

```
392         {
393             foreach (Cell cell in this.Cells)
394             {
395                 cell.onClick(x, y, strat);
396             }
397         }
398 
399         /// <summary>
400         /// Sets a strategy for a cell according to grid coordinates
401         /// </summary>
402         /// <param name="x"></param>
403         /// <param name="y"></param>
404         /// <param name="strat"></param>
405         public void setStrategy(int x, int y, Strategy strat)
406         {
407             this.Cells[y, x].updateStrategy(strat);
408         }
409 
410         /// <summary>
411         /// Set the color depending on the mode
412         ///
413         /// When in strategy color mode  :  The color of the strategy is shown.
414         /// When in move color mode      :  The color of the last move is shown.
415         ///
416         /// </summary>
417         /// <param name="mode">The color mode to use</param>
418         public void setColorMode(ColorMode mode)
419         {
420             // Switch according to the mode
421             switch (mode)
422             {
423                 case ColorMode.Strategy:
424                     this.setColorFromStrategy();
425                     break;
426                 case ColorMode.Playing:
427                     this.setColorFromMove();
428                     break;
429             }
430 
431             this.ColorMode = mode;
432         }
433 
434         /// <summary>
435         /// Sets the cell's colors from thier strategy
436         /// </summary>
437         private void setColorFromStrategy()
438         {
439             foreach (Cell cell in this.Cells)
440             {
441                 cell.setColorFromStrategy();
442             }
443         }
444 
445         /// <summary>
446         /// Sets the cell's colors from thier last move
447         /// </summary>
448         private void setColorFromMove()
449         {
450             foreach (Cell cell in this.Cells)
451             {
452                 cell.setColorFromMove();
453             }
454         }
455 
456 
457         /// <summary>
458         /// Finds the number of times the given strategy appears on the board
459         /// </summary>
460         /// <param name="strategy">The strategy to look for</param>
461         /// <returns></returns>
462         public int findCountOfStrategy(Strategy strategy)
463         {
464             int count = 0;
465 
466             foreach (Cell cell in this.Cells)
467             {
468                 // Find every cell that has the same type as the current strategy
469                 if (strategy.GetType() == cell.Strategy.GetType())
470                 {
471                     count++;
472                 }
473             }
474 
475             // Return the result rounded down to two decimal places
```

```
476             return count;
477         }
478 
479         /// <summary>
480         /// Returns the average score of a strategy on the board
481         /// </summary>
482         /// <param name="strategy">The strategy to look for</param>
483         /// <returns></returns>
484         public double findAvgScoreOfStrategy(Strategy strategy)
485         {
486             double count = 0;
487             int i = 0;
488 
489             foreach (Cell cell in this.Cells)
490             {
491                 // Find every cell that has the same type as the current strategy
492                 if (strategy.GetType() == cell.Strategy.GetType())
493                 {
494                     // Increment the total score and the count
495                     count += cell.Score;
496                     i++;
497                 }
498             }
499 
500             // Find the percentage from the count
501             count = (count / i);
502 
503             // Return the result rounded down to two decimal places
504             return Math.Round(count, 2);
505         }
506 
507 
508         /// <summary>
509         /// Serializes and saves grid data to a path
510         /// </summary>
511         /// <param name="path">Where to save the file on the user's disk</param>
512         public void saveData(string path)
513         {
514             this.SerializableCells = this.Cells.asList();
515             FileStream fs = new FileStream(path, FileMode.Create);
516             XmlSerializer xs = new XmlSerializer(typeof(Grid));
517             xs.Serialize(fs, this);
518             fs.Close();
519         }
520 
521 
522         /// <summary>
523         /// Serialize and saves grid data to the default location
524         /// </summary>
525         public void saveData()
526         {
527             this.saveData(DEFAULT_DATA_FILEPATH);
528         }
529 
530 
531         /// <summary>
532         /// Load serialized data from a path
533         /// </summary>
534         /// <param name="path">Where to load the file on the user's disk</param>
535         public void loadData(string path)
536         {
537             Grid newGrid;
538 
539             XmlSerializer xs = new XmlSerializer(typeof(Grid));
540             using (StreamReader rd = new StreamReader(path))
541             {
542                 newGrid = xs.Deserialize(rd) as Grid;
543             }
544 
545             // rebuild the neighbors
546             newGrid.Cells = newGrid.SerializableCells.asArrayOfArray(newGrid.NbLines, ↩
                    newGrid.NbCols);
547             foreach (var cell in newGrid.Cells)
548             {
549                 cell.Neighbors = newGrid.findCellNeighbors(cell);
550             }
551 
552             // Set each of the values from the serialized data
553             this.Width = newGrid.Width;
554             this.Height = newGrid.Height;
555             this.NbCols = newGrid.NbCols;
556             this.NbLines = newGrid.NbLines;
557             this.Cells = newGrid.Cells;
558             this.PayoffMatrix = newGrid.PayoffMatrix;
```

```
559             this.WrapMode = newGrid.WrapMode;
560         }
561 
562         /// <summary>
563         /// Loads the serialized data from the default location
564         /// </summary>
565         public void loadData()
566         {
567             this.loadData(DEFAULT_DATA_FILEPATH);
568         }
569         #endregion
570     }
571 }
```

2.3 PayoffMatrix.cs

```
1     /*
2      Class            :   PayoffMatrix.cs
3      Description      :   Class used to modelize the prisoner's dilemma payoff matrix
4      Author           :   SEEMULLER Julien
5      Date             :   10.04.2017
6 */
7 
8 using System;
9 using System.Collections.Generic;
10 using System.Linq;
11 using System.Text;
12 using System.Threading.Tasks;
13 
14 namespace PrisonersDilemmaCA
15 {
16     public class PayoffMatrix
17     {
18         #region fields
19         #region consts
20         private const int DEFAULT_TEMPTATION_PAYOFF = 0;
21         private const int DEFAULT_REWARD_PAYOFF = 1;
22         private const int DEFAULT_PUNISHMENT_PAYOFF = 3;
23         private const int DEFAULT_SUCKER_PAYOFF = 5;
24         #endregion
25 
26         private int _reward;        // Reward payoff
27         private int _sucker;        // Sucker's payoff
28         private int _temptation;    // Tempatation payoff
29         private int _punishment;    // Punishment payoff
30         #endregion
31 
32         #region properties
33         public int Reward
34         {
35             get { return _reward; }
36             set { _reward = value; }
37         }
38 
39         public int Sucker
40         {
41             get { return _sucker; }
42             set { _sucker = value; }
43         }
44 
45         public int Temptation
46         {
47             get { return _temptation; }
48             set { _temptation = value; }
49         }
50 
51         public int Punishment
52         {
53             get { return _punishment; }
54             set { _punishment = value; }
55         }
56         #endregion
57 
58         #region constructors
59         /// <summary>
60         /// Designated constructor
61         ///
62         /// Rules :
63         /// T better than R better than P better than S
64         /// </summary>
```

```
65          /// <param name="t">Temptation payoff</param>
66          /// <param name="r">Reward payoff</param>
67          /// <param name="p">Punishment payoff</param>
68          /// <param name="s">Sucker's payoff</param>
69          public PayoffMatrix(int t, int r, int p, int s)
70          {
71              this.Temptation = t;
72              this.Reward = r;
73              this.Punishment = p;
74              this.Sucker = s;
75          }
76
77          /// <summary>
78          /// Default constructor
79          /// </summary>
80          public PayoffMatrix() : this(DEFAULT_TEMPTATION_PAYOFF, DEFAULT_REWARD_PAYOFF, ↩
                DEFAULT_PUNISHMENT_PAYOFF, DEFAULT_SUCKER_PAYOFF)
81          {
82              // No code
83          }
84          #endregion
85
86          #region methods
87          /// <summary>
88          /// Returns player1's payoff of a match.
89          /// </summary>
90          /// <param name="playerOneChoice"></param>
91          /// <param name="playerTwoChoice"></param>
92          /// <returns></returns>
93          public int returnPayoff(Move playerOneChoice, Move playerTwoChoice)
94          {
95              int payoff = 0;
96
97              switch (playerOneChoice)
98              {
99                  case Move.Cooperate:
100                     // Player 1 cooperates, Player 2 cooperates = Reward payoff
101                     if (playerTwoChoice == Move.Cooperate)
102                     {
103                         payoff = this.Reward;
104                     }
105                     // Player 1 cooperates, Player 2 defects = Sucker's payoff
106                     if (playerTwoChoice == Move.Defect)
107                     {
108                         payoff = this.Sucker;
109                     }
110                     break;
111
112                 case Move.Defect:
113                     // Player 1 defects, Player 2 cooperates = Temptation payoff
114                     if (playerTwoChoice == Move.Cooperate)
115                     {
116                         payoff = this.Temptation;
117                     }
118
119                     // Player 2 defects, Player 2 defects = Punishment payoff
120                     if (playerTwoChoice == Move.Defect)
121                     {
122                         payoff = this.Punishment;
123                     }
124                     break;
125             }
126
127             return payoff;
128         }
129
130         /// <summary>
131         /// Checks the validity of the matrix according to the rules :
132         /// T better than R better than P better than S
133         /// </summary>
134         /// <param name="t">Temptation payoff</param>
135         /// <param name="r">Reward payoff</param>
136         /// <param name="p">Punishment payoff</param>
137         /// <param name="s">Sucker's payoff</param>
138         /// <returns>True if the matrix is valid, false if it is not</returns>
139         public static bool isValid(int t, int r, int p, int s)
140         {
141             bool result = false;
142
143             // First condition of validity
144             if ((t < r) && (r < p) && (p < s))
145             {
146                 if (2 * r < t + s)
147                 {
```

```
148                     result = true;
149                 }
150             }
151
152             return result;
153         }
154
155         /// Overloaded function that allows isValid to be used on the current object
156         /// <summary>
157         /// Checks the validity of the matrix according to the rules :
158         /// T better than R better than P better than S
159         /// </summary>
160         /// <returns>True if the matrix is valid, false if it is not</returns>
161         public bool isValid()
162         {
163             return PayoffMatrix.isValid(this.Temptation, this.Reward, this.Punishment, ↩
                    this.Sucker);
164         }
165
166         #endregion
167     }
168 }
```

3 Classes d'extensions

3.1 ArrayExtensions.cs

```
1   /*
2    Class            :   ArrayExtensions.cs
3    Description      :   Allows the conversion of multidimensional arrays and lists
4    Author           :   SEEMULLER Julien
5    Date             :   16.05.2017
6   */
7
8   using System;
9   using System.Collections.Generic;
10  using System.Linq;
11  using System.Text;
12  using System.Threading.Tasks;
13
14  namespace PrisonersDilemmaCA
15  {
16      public static class ArrayExtensions
17      {
18          /// <summary>
19          /// Converts the current array ([,]) to a list
20          /// </summary>
21          /// <param name="inputArray">The 2d array to convert</param>
22          /// <returns></returns>
23          public static List<Cell> asList(this Cell[,] inputArray)
24          {
25              List<Cell> output = new List<Cell>();
26
27              for (int i = 0; i < inputArray.GetLength(0); i++)
28              {
29                  for (int j = 0; j < inputArray.GetLength(1); j++)
30                  {
31                      output.Add(inputArray[i, j]);
32                  }
33              }
34
35              return output;
36          }
37
38          /// <summary>
39          /// Converts a list to a 2D array
40          /// </summary>
41          /// <param name="inputList">The list to convert</param>
42          /// <param name="nbLines">The number of lines of the outputted 2d array</param>
43          /// <param name="nbCols">The number of columns of the outputted 2d array</param>
44          /// <returns></returns>
45          public static Cell[,] asArrayOfArray(this List<Cell> inputList, int nbLines, ↩
                int nbCols)
46          {
47              Cell[,] output = new Cell[nbLines, nbCols];
48
49              // Check if the input is valid (check if the number of elements is ↩
                    superior or equal
50              // to the number of lines times the number of columns
51              if (inputList.Count >= nbLines * nbCols)
52              {
53                  int i = 0;
54
55                  for (int y = 0; y < nbLines; y++)
56                  {
57                      for (int x = 0; x < nbCols; x++)
58                      {
59                          output[y, x] = inputList[i];
60                          i++;
61                      }
62                  }
63              }
64              else
65              {
66                  // Else we throw the user an error
67                  throw new System.ArgumentException("The number of elements is inferior ↩
                        to the size of the outputted 2d array", "original");
68              }
69
70              return output;
71          }
72      }
73  }
```

3.2 ColorExtensions.cs

```
   /*
    Class           :   ColorExtensions.cs
    Description     :   Allows the conversion between Color and string format
    Author          :   Ari ROTH
                        http://stackoverflow.com/questions/2395438/convert-system-drawing-color-to-rgb-a
    Date            :   07.03.2010
    Changes         :   Adapted for use with transparency, changed to an extension format (this Color) - SEEMULLER Julien - 28.04.2017
*/

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrisonersDilemmaCA
{
    public static class ColorExtensions
    {
        /// <summary>
        /// Converts a color to Hex format
        /// </summary>
        /// <param name="c">The color to convert</param>
        /// <returns></returns>
        public static string ToHex(this Color c)
        {
            return "#" + c.R.ToString("X2") + c.G.ToString("X2") + c.B.ToString("X2");
        }

        /// <summary>
        /// Converts a color to Hex format with transparency
        /// </summary>
        /// <param name="c">The color to convert</param>
        /// <param name="transparency">The transparency level to apply to the color</param>
        /// <returns></returns>
        public static string ToHex(this Color c, byte transparency)
        {
            return "#" + transparency.ToString("X2") + c.R.ToString("X2") + c.G.ToString("X2") + c.B.ToString("X2");
        }

        /// <summary>
        /// Converts a color to RGB format
        /// </summary>
        /// <param name="c">The color to convert</param>
        /// <returns></returns>
        public static string ToRGB(this Color c)
        {
            return "RGB(" + c.R.ToString() + "," + c.G.ToString() + "," + c.B.ToString() + ")";
        }
    }
}
```

3.3 ComboBoxExtensions.cs

```
   /*
    Class           :   ComboBoxExtensions.cs
    Description     :   Allows the use of colors inside combo boxes
    Author          :   STEPHENS Rod
                        http://csharphelper.com/blog/2016/03/make-a-combobox-display-colors-or-images-in
    Date            :   29.03.2016
    Changes         :   24.04.2017, Adapted for use with strategies - SEEMULLER Julien
*/

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Drawing.Text;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
18 using System.Windows.Forms;
19
20 namespace PrisonersDilemmaCA
21 {
22     public static class ComboBoxExtensions
23     {
24         // Margins around owner drawn ComboBoxes.
25         private const int MarginWidth = 6;
26         private const int MarginHeight = 2;
27
28         /// <summary>
29         /// Draw a ComboBox item that is displaying a strategy and its color
30         /// </summary>
31         /// <param name="sender"></param>
32         /// <param name="e"></param>
33         private static void DrawItem(object sender, DrawItemEventArgs e)
34         {
35             if (e.Index < 0) return;
36
37             // Clear the background appropriately.
38             e.DrawBackground();
39
40             // Draw the color sample.
41             int height = e.Bounds.Height - 2 * MarginHeight;
42             Rectangle rectangle = new Rectangle(e.Bounds.X + MarginWidth, e.Bounds.Y + ↩
                   MarginHeight, height, height);
43             ComboBox comboBox = sender as ComboBox;
44             Color color = (comboBox.Items[e.Index] as Strategy).getColor();
45
46             using (SolidBrush brush = new SolidBrush(color))
47             {
48                 e.Graphics.FillRectangle(brush, rectangle);
49             }
50
51             // Outline the sample in black.
52             e.Graphics.DrawRectangle(Pens.Black, rectangle);
53
54             // Draw the color's name to the right.
55             using (Font font = new Font(comboBox.Font.FontFamily, comboBox.Font.Size * ↩
                   0.95f, FontStyle.Regular))
56             {
57                 using (StringFormat sf = new StringFormat())
58                 {
59                     sf.Alignment = StringAlignment.Near;
60                     sf.LineAlignment = StringAlignment.Center;
61                     int x = height + 2 * MarginWidth;
62                     int y = e.Bounds.Y + e.Bounds.Height / 2;
63                     e.Graphics.TextRenderingHint = TextRenderingHint.AntiAliasGridFit;
64                     e.Graphics.DrawString(comboBox.Items[e.Index].ToString(), font, ↩
                           Brushes.Black, x, y, sf);
65                 }
66             }
67
68             // Draw the focus rectangle if appropriate.
69             e.DrawFocusRectangle();
70         }
71
72
73         /// <summary>
74         /// Add a list of strategy to a combobox
75         /// </summary>
76         /// <param name="comboBox">The combobox we apply the function to</param>
77         /// <param name="strats">The strategies to add to the combobox</param>
78         public static void AddStrategies(this ComboBox comboBox, List<Strategy> strats)
79         {
80             // Make the ComboBox owner-drawn.
81             comboBox.DrawMode = DrawMode.OwnerDrawFixed;
82
83             // Add the strategies to the ComboBox's items.
84             foreach (Strategy strat in strats)
85             {
86                 comboBox.Items.Add(strat);
87             }
88
89             // Subscribe to the DrawItem event.
90             comboBox.DrawItem += DrawItem;
91         }
92     }
93 }
```

4 Stratégies

4.1 Strategy.cs

```
   /*
    Class           :   Strategy.cs
    Description     :   Strategy abstract class, Cf. Strategy design pattern.
                        Used to model other strategies.
    Author          :   SEEMULLER Julien
    Date            :   10.04.2017
*/

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Text.RegularExpressions;
using System.Threading.Tasks;

namespace PrisonersDilemmaCA
{
    public abstract class Strategy : IComparable
    {
        #region methods
        /// <summary>
        /// Returns the next move of the cell based on its neighbors
        /// </summary>
        /// <param name="cell">The cell using this function</param>
        /// <param name="neighbors">The neighbors of the cell using this function</param>
        /// <returns></returns>
        public abstract Move chooseMove(Cell cell, List<Cell> neighbors);

        /// <summary>
        /// Returns the color associated with the strategy
        /// </summary>
        /// <returns></returns>
        public abstract Color getColor();

        /// <summary>
        /// Returns the name of the strategy if it follows the naming convention loosely
        /// The name of the strategy is taken from the filename
        /// ex : "StratTitForTat.cs" -> "Tit for tat"
        /// </summary>
        /// <returns></returns>
        public override string ToString()
        {
            // Get the name of the current class
            string strategyName = this.GetType().Name;

            // Filter the name (remove "Strat" and use spaces insted of CamelCase)
            strategyName = Regex.Replace(strategyName, "(Strat)", "");
            strategyName = Regex.Replace(strategyName, "([a-z])([A-Z])", "$1 $2");

            return strategyName;
        }

        /// <summary>
        /// Used for sorting, alphanumerical sorting according to the name of the ↩
            strategy
        /// </summary>
        /// <param name="obj"></param>
        /// <returns></returns>
        public int CompareTo(object obj)
        {
            return this.ToString().CompareTo((obj as Strategy).ToString());
        }
        #endregion
    }
}
```

4.2 StratAdaptativePavlov.cs

```
   /*
    Class           :   StratPavlov.cs
    Description     :   Identifies an opponents according to his moves and counters them
                        http://www.prisoners-dilemma.com/strategies.html
```

```
5
6       Author          :   SEEMULLER Julien
7       Date            :   10.04.2017
8   */
9
10  using System;
11  using System.Collections.Generic;
12  using System.Drawing;
13  using System.Linq;
14  using System.Text;
15  using System.Threading.Tasks;
16
17  namespace PrisonersDilemmaCA
18  {
19      public class StratAdaptativePavlov : Strategy
20      {
21          #region fields
22          #region consts
23          private const int DEFAULT_NB_OF_ANALYSING_TURNS = 7;
24          #endregion
25
26          private StratTitForTat _tft;
27          private StratTitForTwoTats _tftt;
28          private StratAlwaysDefect _ad;
29          private Strategy _currentStrategy;
30
31          private int _defectCount;
32          #endregion
33
34          #region properties
35          public StratTitForTat Tft
36          {
37              get
38              {
39                  return _tft;
40              }
41
42              set
43              {
44                  _tft = value;
45              }
46          }
47
48          public StratTitForTwoTats Tftt
49          {
50              get
51              {
52                  return _tftt;
53              }
54
55              set
56              {
57                  _tftt = value;
58              }
59          }
60
61          public StratAlwaysDefect Ad
62          {
63              get
64              {
65                  return _ad;
66              }
67
68              set
69              {
70                  _ad = value;
71              }
72          }
73
74          public Strategy CurrentStrategy
75          {
76              get
77              {
78                  return _currentStrategy;
79              }
80
81              set
82              {
83                  _currentStrategy = value;
84              }
85          }
86
87          public int DefectCount
88          {
```

```
89              get
90              {
91                  return _defectCount;
92              }
93
94              set
95              {
96                  _defectCount = value;
97              }
98          }
99          #endregion
100
101         #region constructors
102         public StratAdaptativePavlov()
103         {
104             this.Tft = new StratTitForTat();
105             this.Tftt = new StratTitForTwoTats();
106             this.Ad = new StratAlwaysDefect();
107             this.CurrentStrategy = this.Tft;
108
109             this.DefectCount = 0;
110         }
111         #endregion
112
113         #region methods
114         public override Move chooseMove(Cell cell, List<Cell> neighbors)
115         {
116             // Count the number of defectors before proceeding
117             foreach (var neighbor in neighbors)
118             {
119                 if (neighbor.History.Count > 0)
120                 {
121                     if (neighbor.History.First() == Move.Defect)
122                     {
123                         this.DefectCount++;
124                     }
125                 }
126             }
127
128             // We analyse other cells while playing tit for tat before we reach the ↩
                    threshold
129             if (cell.History.Count < DEFAULT_NB_OF_ANALYSING_TURNS)
130             {
131                 this.CurrentStrategy = this.Tft;
132             }
133             else
134             {
135                 // Change our move only every x rounds
136                 if (cell.History.Count % DEFAULT_NB_OF_ANALYSING_TURNS == 0)
137                 {
138                     // Find the average defect count over the number of analysing ↩
                            turns turns
139                     this.DefectCount /= neighbors.Count;
140
141                     // Choose a move according to the defect count
142                     if (this.DefectCount > 4)
143                     {
144                         // Opponent always defects, we play always defect
145                         this.CurrentStrategy = this.Ad;
146                     }
147                     else if (this.DefectCount == 3)
148                     {
149                         // Opponent is STFT, we play TFTT
150                         this.CurrentStrategy = this.Tftt;
151                     }
152                     else if (this.DefectCount == 0)
153                     {
154                         // Opponent cooperates, we play TFT
155                         this.CurrentStrategy = this.Tft;
156                     }
157                     else
158                     {
159                         // Classified as random strategy, we always defect
160                         this.CurrentStrategy = this.Ad;
161                     }
162
163                     // When we are done analysing, we reset the counter
164                     this.DefectCount = 0;
165                 }
166             }
167
168             // Return our current choice according to our strategy
169             return this.CurrentStrategy.chooseMove(cell, neighbors);
170         }
```

```
171
172         public override Color getColor()
173         {
174             return Color.FromArgb(165, 214, 167);
175         }
176         #endregion
177     }
178 }
```

4.3 StratAlwaysCooperate.cs

```
1     /*
2      Class            :   StratAlwaysCooperate.cs
3      Description      :   Always cooperate strategy
4      Author           :   SEEMULLER Julien
5      Date             :   10.04.2017
6  */
7
8  using System;
9  using System.Collections.Generic;
10 using System.Drawing;
11 using System.Linq;
12 using System.Text;
13 using System.Threading.Tasks;
14
15 namespace PrisonersDilemmaCA
16 {
17     public class StratAlwaysCooperate : Strategy
18     {
19         #region fields
20         #endregion
21
22         #region properties
23         #endregion
24
25         #region constructors
26         #endregion
27
28         #region methods
29         public override Move chooseMove(Cell cell, List<Cell> neighbors)
30         {
31             return Move.Cooperate;
32         }
33
34         public override Color getColor()
35         {
36             return Color.FromArgb(46, 204, 113);
37         }
38         #endregion
39     }
40 }
```

4.4 StratAlwaysDefect.cs

```
1     /*
2      Class            :   StratAlwaysDefect.cs
3      Description      :   Always defects strategy
4      Author           :   SEEMULLER Julien
5      Date             :   10.04.2017
6  */
7
8  using System;
9  using System.Collections.Generic;
10 using System.Drawing;
11 using System.Linq;
12 using System.Text;
13 using System.Threading.Tasks;
14
15 namespace PrisonersDilemmaCA
16 {
17     public class StratAlwaysDefect : Strategy
18     {
19         #region fields
20         #endregion
21
```

```
22          #region properties
23          #endregion
24
25          #region constructors
26          #endregion
27
28          #region methods
29          public override Move chooseMove(Cell cell, List<Cell> neighbors)
30          {
31              return Move.Defect;
32          }
33
34          public override Color getColor()
35          {
36              return Color.FromArgb(192, 57, 43);
37          }
38          #endregion
39      }
40  }
```

4.5 StratBlinker.cs

```
1     /*
2      Class            :   StratBlinker.cs
3      Description      :   Blinker strategy, alternates between "defect" and "cooperate"
4      Author           :   SEEMULLER Julien
5      Date             :   10.04.2017
6  */
7
8  using System;
9  using System.Collections.Generic;
10 using System.Drawing;
11 using System.Linq;
12 using System.Text;
13 using System.Threading.Tasks;
14
15 namespace PrisonersDilemmaCA
16 {
17     public class StratBlinker : Strategy
18     {
19         #region fields
20         #endregion
21
22         #region properties
23         #endregion
24
25         #region constructors
26         #endregion
27
28         #region methods
29         public override Move chooseMove(Cell cell, List<Cell> neighbors)
30         {
31             Move result;
32
33             if (cell.History.Count % 2 == 0)
34             {
35                 result = Move.Cooperate;
36             }
37             else
38             {
39                 result = Move.Defect;
40             }
41
42
43             return result;
44         }
45
46         public override Color getColor()
47         {
48             return Color.FromArgb(155, 89, 182);
49         }
50         #endregion
51     }
52 }
```

4.6 StratFortress.cs

```
   /*
    Class           :   StratFortress.cs
    Description     :   Fortress strategy, tries to find neighbors using fortress
                        and cooperates with them.
    Author          :   SEEMULLER Julien
    Date            :   10.04.2017
*/

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrisonersDilemmaCA
{
    public class StratFortress : Strategy
    {
        #region fields
        private bool _hasFoundPartners;
        #endregion

        #region properties
        public bool HasFoundPartners
        {
            get { return _hasFoundPartners; }
            set { _hasFoundPartners = value; }
        }
        #endregion

        #region constructors
        public StratFortress()
        {
            this.HasFoundPartners = false;
        }
        #endregion

        #region methods
        public override Move chooseMove(Cell cell, List<Cell> neighbors)
        {
            Move result = Move.Defect;

            // Strats by playing the sequence "defect, defect, cooperate"
            switch (cell.History.Count)
            {
                case 0:
                    result = Move.Defect;
                    break;
                case 1:
                    result = Move.Defect;
                    break;
                case 2:
                    result = Move.Defect;
                    break;
                case 3:
                    result = Move.Cooperate;
                    break;
                // On the fourth and following turns, we look at our neighbors and see ↩
                    if there are
                // other "Fortress" players
                default:

                    foreach (Cell neighbor in neighbors)
                    {
                        if (neighbor.History.Count >= 3)
                        {
                            if (neighbor.History.ElementAt(0) == Move.Cooperate)
                            {
                                if (neighbor.History.ElementAt(1) == Move.Defect)
                                {
                                    if (neighbor.History.ElementAt(2) == Move.Defect)
                                    {
                                        this.HasFoundPartners = true;
                                    }
                                }
                            }
                        }
                    }

                    // If we have found other fortress players, we cooperate, else we ↩
                        always defect
                    if (HasFoundPartners)
```

```
82                      {
83                          result = Move.Cooperate;
84                      }
85                      else
86                      {
87                          result = Move.Defect;
88                      }
89
90                      break;
91              }
92
93
94              return result;
95          }
96
97          public override Color getColor()
98          {
99              return Color.FromArgb(230, 126, 34);
100         }
101         #endregion
102     }
103 }
```

4.7 StratFortress.cs

```
1     /*
2      Class            :    StratFortress.cs
3      Description      :    Fortress strategy, tries to find neighbors using fortress
4                             and cooperates with them.
5      Author           :    SEEMULLER Julien
6      Date             :    10.04.2017
7  */
8
9  using System;
10 using System.Collections.Generic;
11 using System.Drawing;
12 using System.Linq;
13 using System.Text;
14 using System.Threading.Tasks;
15
16 namespace PrisonersDilemmaCA
17 {
18     public class StratFortress : Strategy
19     {
20         #region fields
21         private bool _hasFoundPartners;
22         #endregion
23
24         #region properties
25         public bool HasFoundPartners
26         {
27             get { return _hasFoundPartners; }
28             set { _hasFoundPartners = value; }
29         }
30         #endregion
31
32         #region constructors
33         public StratFortress()
34         {
35             this.HasFoundPartners = false;
36         }
37         #endregion
38
39         #region methods
40         public override Move chooseMove(Cell cell, List<Cell> neighbors)
41         {
42             Move result = Move.Defect;
43
44             // Strats by playing the sequence "defect, defect, cooperate"
45             switch (cell.History.Count)
46             {
47                 case 0:
48                     result = Move.Defect;
49                     break;
50                 case 1:
51                     result = Move.Defect;
52                     break;
53                 case 2:
54                     result = Move.Defect;
55                     break;
```

```
56                  case 3:
57                      result = Move.Cooperate;
58                      break;
59                  // On the fourth and following turns, we look at our neighbors and see ↩
                        if there are
60                  // other "Fortress" players
61                  default:
62
63                      foreach (Cell neighbor in neighbors)
64                      {
65                          if (neighbor.History.Count >= 3)
66                          {
67                              if (neighbor.History.ElementAt(0) == Move.Cooperate)
68                              {
69                                  if (neighbor.History.ElementAt(1) == Move.Defect)
70                                  {
71                                      if (neighbor.History.ElementAt(2) == Move.Defect)
72                                      {
73                                          this.HasFoundPartners = true;
74                                      }
75                                  }
76                              }
77                          }
78
79                      // If we have found other fortress players, we cooperate, else we ↩
                            always defect
80                      if (HasFoundPartners)
81                      {
82                          result = Move.Cooperate;
83                      }
84                      else
85                      {
86                          result = Move.Defect;
87                      }
88
89                      break;
90              }
91
92
93              return result;
94          }
95
96          public override Color getColor()
97          {
98              return Color.FromArgb(230, 126, 34);
99          }
100         #endregion
101     }
102 }
```

4.8 StratGrimTrigger.cs

```
1     /*
2      Class            :   StratGrimTrigger.cs
3      Description      :   Grim trigger strategy, cooperates until some neighbor
4      Author           :   SEEMULLER Julien
5      Date             :   10.04.2017
6  */
7
8  using System;
9  using System.Collections.Generic;
10 using System.Drawing;
11 using System.Linq;
12 using System.Text;
13 using System.Threading.Tasks;
14
15 namespace PrisonersDilemmaCA
16 {
17     public class StratGrimTrigger : Strategy
18     {
19         #region fields
20         private bool _wasBetrayed;
21         #endregion
22
23         #region properties
24         public bool WasBetrayed
25         {
26             get { return _wasBetrayed; }
27             set { _wasBetrayed = value; }
```

```
28          }
29          #endregion
30
31          #region constructors
32          public StratGrimTrigger()
33          {
34              this.WasBetrayed = false;
35          }
36          #endregion
37
38          #region methods
39          public override Move chooseMove(Cell cell, List<Cell> neighbors)
40          {
41              // Starts by cooperating
42              Move result = Move.Cooperate;
43
44              // Check if we were betrayed in the past
45              if (WasBetrayed)
46              {
47                  result = Move.Defect;
48              }
49              else
50              {
51                  // If we didn't get betrayed yet, we look at our neighbors
52                  if (cell.History.Count > 1)
53                  {
54                      // Look if we got betrayed by a neighbor after our first move
55                      foreach (Cell neighbor in neighbors)
56                      {
57                          if (neighbor.History.First() == Move.Defect)
58                          {
59                              // If we are betrayed, we switch to a "Always Defect" ↩
                                    strategy
60                              this.WasBetrayed = true;
61                              result = Move.Defect;
62                              break;
63                          }
64                      }
65                  }
66              }
67
68              return result;
69          }
70
71          public override Color getColor()
72          {
73              return Color.FromArgb(52, 73, 94);
74          }
75          #endregion
76      }
77  }
```

4.9 StratRandom.cs

```
1     /*
2      Class            :   StratRandom.cs
3      Description      :   Random strategy.
4      Author           :   SEEMULLER Julien
5      Date             :   10.04.2017
6  */
7
8  using System;
9  using System.Collections.Generic;
10 using System.Drawing;
11 using System.Linq;
12 using System.Text;
13 using System.Threading.Tasks;
14
15 namespace PrisonersDilemmaCA
16 {
17     public class StratRandom : Strategy
18     {
19         #region fields
20         #endregion
21
22         #region properties
23         #endregion
24
25         #region constructors
26         #endregion
```

```
27
28          #region methods
29          public override Move chooseMove(Cell cell, List<Cell> neighbors)
30          {
31              // Make a new unique random number generator
32              Random rng = new Random(Guid.NewGuid().GetHashCode());
33
34              // Make a list with the possible moves
35              List<Move> availableMoves = new List<Move>();
36              availableMoves.Add(Move.Cooperate);
37              availableMoves.Add(Move.Defect);
38
39              // Return a random element in the list
40              return availableMoves[rng.Next(availableMoves.Count)];
41          }
42
43          public override Color getColor()
44          {
45              return Color.FromArgb(41, 128, 185);
46          }
47          #endregion
48      }
49  }
```

4.10 StratSuspiciousTitForTat.cs

```
1     /*
2      Class            :    StratTitForTat.cs
3      Description      :    Same as Tit-for-tat strategy, but defects first
4                             http://www.investopedia.com/terms/t/tit-for-tat.asp
5
6      Author           :    SEEMULLER Julien
7      Date             :    10.04.2017
8  */
9
10 using System;
11 using System.Collections.Generic;
12 using System.Drawing;
13 using System.Linq;
14 using System.Text;
15 using System.Threading.Tasks;
16
17 namespace PrisonersDilemmaCA
18 {
19     public class StratSuspiciousTitForTat : Strategy
20     {
21         #region fields
22         #endregion
23
24         #region properties
25         #endregion
26
27         #region constructors
28         #endregion
29
30         #region methods
31         public override Move chooseMove(Cell cell, List<Cell> neighbors)
32         {
33             // Cooperates on first move, then copies his best openent
34             Move result = Move.Defect;
35
36
37             // If this wasn't our first round, we look at our neighbors
38             if (cell.History.Count > 2)
39             {
40                 // We initialise our variables with the first neighbor in the list
41                 result = neighbors[0].History.First();
42                 int min = neighbors[0].Score;
43
44                 foreach (Cell neighbor in neighbors)
45                 {
46                     if (min > neighbor.Score)
47                     {
48                         min = neighbor.Score;
49                         result = neighbor.History.First();
50                     }
51                 }
52             }
53
54             return result;
```

```
55          }
56
57          public override Color getColor()
58          {
59              return Color.FromArgb(239, 154, 154);
60          }
61          #endregion
62      }
63  }
```

4.11 StratTitForTat.cs

```
1     /*
2      Class            :    StratTitForTat.cs
3      Description      :    Tit-for-tat strategy
4                             http://www.investopedia.com/terms/t/tit-for-tat.asp
5
6      Author           :    SEEMULLER Julien
7      Date             :    10.04.2017
8   */
9
10  using System;
11  using System.Collections.Generic;
12  using System.Drawing;
13  using System.Linq;
14  using System.Text;
15  using System.Threading.Tasks;
16
17  namespace PrisonersDilemmaCA
18  {
19      public class StratTitForTat : Strategy
20      {
21          #region fields
22          #endregion
23
24          #region properties
25          #endregion
26
27          #region constructors
28          #endregion
29
30          #region methods
31          public override Move chooseMove(Cell cell, List<Cell> neighbors)
32          {
33              // Cooperates on first move, then copies his best openent
34              Move result = Move.Cooperate;
35
36
37              // If this wasn't our first round, we look at our neighbors
38              if (cell.History.Count > 1)
39              {
40                  // We initialise our variables with the first neighbor in the list
41                  result = neighbors[0].History.First();
42                  int min = neighbors[0].Score;
43
44                  foreach (Cell neighbor in neighbors)
45                  {
46                      if (min > neighbor.Score)
47                      {
48                          min = neighbor.Score;
49                          result = neighbor.History.First();
50                      }
51                  }
52              }
53
54              return result;
55          }
56
57          public override Color getColor()
58          {
59              return Color.FromArgb(200, 200, 200);
60          }
61          #endregion
62      }
63  }
```

4.12 StratTitForTwoTats.cs

```
   /*
    Class           :   StratTitForTwoTat.cs
    Description     :   Tit-for-two-tats strategy, copies a neighbors if
                        he plays the same move twice in a row.

    Author          :   SEEMULLER Julien
    Date            :   10.04.2017
*/

using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace PrisonersDilemmaCA
{
    public class StratTitForTwoTats : Strategy
    {
        #region fields
        #endregion

        #region properties
        #endregion

        #region constructors
        #endregion

        #region methods
        public override Move chooseMove(Cell cell, List<Cell> neighbors)
        {
            Move result;
            bool hasToDefect = false;

            // If this wasn't our first round, we look at our neighbors, else we cooperate
            if (cell.History.Count > 1)
            {
                // If one of our neighbors defects twice in a row, we
                foreach (Cell neighbor in neighbors)
                {
                    // Check if our neighbor has played at least 2 turns before proceeding
                    if (neighbor.History.Count >= 2)
                    {
                        if (neighbor.History.ElementAt(0) == Move.Defect && neighbor.History.ElementAt(1) == Move.Defect)
                        {
                            hasToDefect = true;
                            break;
                        }
                    }
                }
            }

            // Send back the correct result
            if (hasToDefect)
            {
                result = Move.Defect;
            }
            else
            {
                result = Move.Cooperate;
            }


            return result;
        }

        public override Color getColor()
        {
            return Color.FromArgb(100, 100, 100);
        }
        #endregion
    }
}
```

5 Enums

5.1 Enums.cs

```
1    /*
2     Class           :   Enum.cs
3     Description     :   Replaces values with a more verbose alternative
4     Author          :   SEEMULLER Julien
5     Date            :   10.04.2017
6 */
7 using System;
8 using System.Collections.Generic;
9 using System.Linq;
10 using System.Text;
11 using System.Threading.Tasks;
12
13 namespace PrisonersDilemmaCA
14 {
15     // The different moves a cell can play
16     public enum Move { Cooperate, Defect }
17
18     // Defines if the color of the cell is from its actions or strategy
19     public enum ColorMode { Strategy, Playing }
20
21     // Defines if we wrap around the board to find neighbors (like a torus)
22     public enum WrapMode { Default, Torus }
23 }
```