6 Prisoner's Dilemma

6.1 Introduction

One of the most prominently studied phenomena in Game Theory is the *Prisoner's Dilemma.*. The Prisoner's Dilemma, which was formulated by Melvin Drescher and Merrill Flood and named by Albert W. Tucker, is an example of a class of games called *non-zero-sum* games.

In *zero-sum* games, total benefit to all players add up to zero, or in other words, each player can only benefit at the expense of other players (e.g. chess, football, poker – one person can only win when the opponent loses). On the other hand, in non-zero-games, each person's benefit does not necessarily come at the expense of someone else. In many non-zero-sum situations, a person can benefit only when others benefit as well. Non-zero-sum situations exist where the supply of a resource is not fixed or limited in any way (e.g. knowledge, artwork, and trade). Prisoner's Dilemma, as a non-zero-sum game, demonstrates a conflict between rational individual behavior and the benefits of cooperation in certain situations. The classical prisoner's dilemma is as follows.

Two suspects are apprehended by the police. The police do have enough evidence to convict these two suspects. As a result, they separate the two, visit each of them, and offer both the same deal: "If you confess, and your accomplice remains silent, he goes to jail for 10 years and you can go free. If you both remain silent, only minor charges can be brought upon both of you and you guys get 6 months each. If you both confess, then each of you two gets 5 years."

Each suspect may reason as follows: "Either my partner confesses or not. If he does confess and I remain silent, I get 10 years while if I confess, I get 5 years. So, if my partner confesses, it is best that I confess and get only 5 years than 10 years in prison. If he didn't, then by confessing, I go free, whereby remaining silent, I get 6 months. Thus, if he didn't confess, it is best to confess, so that I can go free. Whether or not my partner confesses or not, it is best that I confess." In a non-iterated prisoner's dilemma, the two partners will never have to work together again. Both partners are thinking in the above manner and decide to confess. Consequently, they both receive 5 years in prison. If neither would have confessed, they would have only gotten 6 months each. The rational behavior paradoxically leads to a socially unbeneficial outcome. In Figure 6.1 you see the payoff matrix in terms of numbers of years in prison, so the lower the number the better.

	Cooperate	Defect
Cooperate	(0.5, 0.5)	(10, 0)
Defect	(0, 10)	(5, 5)

Figure 1: Payoff matrix for Prisoner's Dilemma in terms of punishment; (x,y) means x number in years in prison for the row, y for the column.

In an Iterated Prisoner's Dilemma where you have multiple rounds, the rational behavior may be different. If players retaliate defecting by defecting in the next round, it may be beneficial to cooperate and thus, in the long run, earn more points, since continually cooperating players will, together, earn the most. In the model for this assignment, it is assumed that an increase in the number of people who cooperate will increase proportionately the benefit for each cooperating player (which would be a fine assumption, for example, in the sharing of knowledge).

For those who do not cooperate, assume that their benefit is some factor (α) multiplied by the number of people who cooperate (that is, to continue the previous example, the non-cooperating players take advantage of the cooperating players). How much cooperation is incited is dependent on the factor multiple for not cooperating. See Figure 6.2 for the rewards in this game. Consequently, in an iterated prisoner's dilemma with multiple players, the dynamics of evolution in cooperation may be observed.

In this assignment, you are going to build a simulation for this type of mulit-player multi-round Prisoner's Dilemma.

6.2 The Game

The players are positioned on patches on a grid of, e.g., 50 by 50. Each player interacts with its immediate neighbours in horizontal, vertical, and diagonal direction. Patches at the edges are connected to patches at the opposite side of the grid. This way, each player interacts with 8 neighbours.

The simulation runs in cycles. At each cycle, each patch will interact with all of its 8 neighbours to determine the score for the interaction. Should a patch have cooperated, its score will be the number of neighbours that also cooperated. Should a patch defect, then the score for this patch will be the product of the Defection-Award factor α and the number of neighbours that cooperated (i.e., the patch has taken advantage of the patches that cooperated). In the subsequent round, the player will adopt the strategy of one of its neighbours that scored the highest in the previous round, including itself. When there is more than one player in its neighbourhood that scored highest, a random choice between them will be made.

	Cooperate	Defect
Cooperate	(1, 1)	(0, α)
Defect	(α, 0)	(0, 0)

Figure 2: Payoff matrix for iterated multi-player Prisoner's Dilemma in terms of reward; higher numbers are better.

6.3 The looks

The grid is displayed as coloured patches (circles or (rounded corner) squares): blue for cooperating players and red for defecting players. In an advanced version, patches that just switched from defection to cooperation are light blue and patches that switched from defection to cooperation are orange.

There is a button "Reset" that (re-)initializes the grid to random strategies. In this grid, every patch is red or blue. There is a button "Go" that starts the simulation, i.e., subsequent cycles are displayed at an interval of 1 second. The label of the button changes to "Pause". When pressed the simulation pauses and the label changes back to "Go".

In an advanced version, clicking on the patches will change its strategy from cooperation to defection and vice versa.

There is a slider that controls the defection award factor α, running from 0.0 to 3.0. Aditionally, the value of α is displayed as a number.

6.4 Approach

Write a program with at least the following classes. Three template files are provided with this assignment. Remove the line "INCOMPLETE". Add methods and variables as mentioned below, including the mentioned names and types, and use them. This is a minimum, more are allowed, of course. **Stick to the names and types mentioned here.** Momotor uses these methods and variables to test your submission. Deviations will cause us trouble and you a lower grade.

As a rule, make instance variables `private`. If you deviate from this rule, motivate your decision in comment. Do not access instance variables directly from other classes; use methods instead.

Do not use so-called magic numbers. Values such as the size of the patches, the size of the grid, etc. should not appear als literal numbers in your code, but declared as instance (or static) variables, possibly `final` or `static`, having proper names. The literal number will appear only once, at the initialization of the variable.

This does not apply if the value is calculated when needed; then no variable is needed, although a local variable could improve readability and understandability.

6.4.1 PrisonersDilemma

This is the main class where the method `main` is located and where the GUI is built. It contains, a.o., a PlayingField object, see below.

6.4.2 PlayingField

PlayingField is a JPanel that displays the evolving patches. Most of the simulation control is defined here. It contains:

- `Patch[][] grid`, the grid of patches (for the class Patch, see below);
- `Random random`, the random number generator of the application; use only one object of this class.
- `void setGrid(boolean[][] isCooperating)` sets the strategy of the patches: `true` if the patch is cooperating (C), `false` if it is defecting (D). This method is needed for our testing, it is very well possible that you do not need it in your program, although it could be convenient for your own testing as well.
- `boolean[][] getGrid()` counterpart of `setGrid`.
- `void step()` makes one step in the simulation: the scores of the patches are calculated and their strategies are updated.
- `void setAlpha(double alpha)` sets the defection award factor α.
- `double getAlpha()` returns the defection award factor α.

6.4.3 Patch

An object of this class represents one player. It contains the strategy (cooperating of defecting). Required methods:

- `boolean isCooperating()` returns whether the strategy is C (true) or D (false).
- `void setCooperating(boolean coop)` sets the strategy to C (true) or D (false).
- `void toggleStrategy()` changes the strategy form C to D and vice versa.
- `double getScore()` returns the score of this patch with the current settings of strategies.

6.4.4 Hints

- It may be convenient to store in each Patch a list of its neighbours (as an array or an ArrayList of Patches). This way, deeply nested loops can be avoided. Furthermore, it makes it easier to change the notion of neighbourhood, e.g., a 4-neighbourhood or larger neighbourhoods.
- Note that the scores of all the patches should be calculated from the current strategy, in other words, do not update the strategies too soon, or you would interfere with the score calculation. A possibility is to deal with this is to store more in a Patch than just the current strategy (and, if applicable, the list of neighbours).
- The operator `%` may come in handy when calculating the neighbours of a patch.

6.4.5 Extensions

Extensions to get a higher grade than 8.

1. Two extra colours to mark patches that just changed strategy.
2. Changing patches by clicking.
3. A simulation speed slider.
4. An alternative update rule is when a player's own score is highest in the neighbourhood (but possibly not unique) the player will not change strategy. Whether this rule applies or not can be chosen in the UI.