

**ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA
INFORMÁTICA**

INGENIERO EN INFORMÁTICA

**DISEÑO E IMPLEMENTACIÓN DE UNA ESTRATEGIA PARA UN
JUEGO DE CONEXIÓN**

Realizado por

DAVID DANIEL ALBARRACÍN MOLINA

Dirigido por

FRANCISCO J. VICO VELA

Departamento

LENGUAJES Y CIENCIAS DE LA COMPUTACIÓN

UNIVERSIDAD DE MÁLAGA

MÁLAGA, Marzo de 2010.

UNIVERSIDAD DE MÁLAGA
ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA
INGENIERO EN INFORMÁTICA

Reunido el tribunal examinador en el día de la fecha, constituido por:

Presidente/a Dº/Dª. _____

Secretario/a Dº/Dª. _____

Vocal Dº/Dª. _____

para juzgar el proyecto Fin de Carrera titulado:

Diseño e implementación de una estrategia para un juego de conexión

realizado por Dº. David Daniel Albarracín Molina

tutorizado por Dº. Francisco J. Vico Vela

ACORDÓ POR _____ OTORGAR LA
CALIFICACIÓN DE _____

Y PARA QUE CONSTE, SE EXTIENDE FIRMADA POR LOS COMPARECIENTES
DEL TRIBUNAL, LA PRESENTE DILIGENCIA.

Málaga a 18 de Marzo de 2010

El Presidente _____ El Secretario _____ El Vocal _____

Fdo: _____ Fdo: _____ Fdo: _____

Design and implementation of a strategy for a connection game

ABSTRACT

The purpose of this project is developing an automatic strategy for playing *Selfo*, a connection game whose goal is to get arranged all the friendly pieces into a single connected group. The idea is to copy some kind of swarm conduct, like the observed in an ant colony or in a flock of birds, to play this game. The swarm behaviors consist on few simple inborn rules. But these simple rules applied in group usually bring complex dynamics. The ultimate goal is to check whether a computer player based on these principles can compete against a thinking trained human.

We first built a simple computer version of *Selfo* in order to deeply examine the dynamics of the game by running human-human and human-dummycomputer games. From this experience we proposed and developed a swarm behavior based strategy.

We built the program over the Zillions of Games platform. Zillions is a popular website where people can download and play a multitude of board games. This is a great tool to get the players experiences. And it also allowed us to easily define the user interface and game rules, letting us focusing on the game engine.

To evaluate the quality of the proposed strategy, we run several simulations of computer-computer games and played human-computer games. We noted that the computer opponent can defeat many times an average player and sometimes beat an experienced player. We also learned that it is very important the initial distribution of pieces and sometimes the strength of the first move. Finally we could observe the speed of the computer player, despite the computational complexity of the game.

We conclude that we have achieved the initial aims: building an easy to use computer version of *Selfo* with an on-line game mode and developing a fast strategy inspired by nature which could face up a human player.

INDEX

Chapter 1: Introduction.....	10
1.1 Motivation	10
1.2 Aims.....	10
1.3 Chapter overview	11
Chapter 2: Introduction to connection games.....	13
2.1 Definition.....	13
2.2 Board design.....	13
2.3 Scale.....	14
2.4 Rules of play	15
2.5 Clarity.....	15
2.6 First move advantage	15
2.7 More players	16
2.8 Defensive play	16
2.9 Classification.....	16
2.10 Examples.....	17
Chapter 3: Description of <i>Selfo</i>	20
3.1 Introduction	20
3.2 Definition.....	21
3.2.1 Board.....	21
3.2.2 Initial board position	22
3.2.3 Density	22
3.2.4 Winning condition	22
3.2.5 Number of players.....	23
3.2.6 Move length.....	23
3.2.7 What a <i>Selfo</i> game is not	23
3.2.8 Summarized rule set	24
3.3 Initial board position	25
3.3.1 Regular positions.....	25
3.3.2 Irregular positions.....	26
3.4 Self-organized dynamics.....	28
3.4.1 An example of self-organized play	29
3.5 Discussion	30

Chapter 4: Related technologies	32
4.1 Introduction	32
4.2 Zillions of Games	33
4.2.1 Introduction	33
4.2.2 Web.....	34
4.2.3 Zillions of Games program	36
4.3 Building a game in Zillions	38
4.3.1 ZRL Language	39
4.3.2 DLL Interface.....	41
Chapter 5: Proposal of a game strategy.....	45
5.1 Initial study.....	45
5.2 An approach	46
5.3 Definition of the strategy	47
Chapter 6: Simulation results	52
6.1 Introduction	52
6.2 Human-Computer results.....	52
6.3 Computer-Computer simulations	53
6.3.1 Setup 1	54
6.3.2 Setup 2.....	56
6.3.3 Setup 3.....	58
6.3.4 Setup 4.....	60
6.3.5 Setup 5.....	62
6.4 General observations.....	64
Chapter 7: Conclusions	65
References	67
Appendix: Implementation	68
Diagrams	68
DLL Interface	68
Structures.....	68
Methods	69
Custom Strategy	70
Structures.....	70
Functions.....	70

Chapter 1: Introduction

1.1 Motivation

It has been shown that nature is a powerful source of inspiration in different fields of engineering. We have taken ideas which has bring to us inventions like planes, diving fins, radar, sonar, Velcro, computer viruses, self-cleaning surfaces and countless other gadgets. Particularly, in computer science we can see many ideas inspired by nature helping us to solve several kinds of problems, ideas like artificial neural networks, fuzzy logic, evolutionary computation or many techniques in computer vision.

Over the years, game theory has provided to computer science the possibility of applying and testing different strategies, behaviors or algorithm. MINIMAX strategies, application of artificial neural networks or the use of pattern databases are among the most common methods to face the problem of building computer intelligences for playing games.

Our proposal is to develop a game engine for a connection game, a kind of board game where the goal is to get certain configuration between the friendly pieces. To achieve that purpose we will think about bio-inspired strategies which will treat each piece like an independent individual following simple rules. That will bring a general and a more complex emergent behavior, similar to how the ant colonies find the shortest path to food simply dropping and following pheromones.

1.2 Aims

This project focuses in three main goals: building a computer version of the connection game *Selfo*, developing a bio-inspired computer play engine taking into account the dynamics of the game; and checking whether the developed strategy is capable of defeating a human player.

According to these objectives, we will first build a preliminary computer version of *Selfo* containing the following features:

- An on-line mode to provide another way to play. So people can play against an offline human player, an online human player and a computer player,
- it should let people send us their save games in order to analyze them.
- a friendly and easy to use interface,

- different game variant,
- the computer game engine should be fast, so the people do not get bored waiting,
- *Selfo* should be placed on the Internet to increase the people who have access to it.

After studying executed games and in-depth understanding the dynamics of the game, we will design an advanced game strategy. The strategy will be bio-inspired and the execution of the game engine will remain fast, in order to keep the game fun.

The last phase will include a series of tests consisting of computer-computer and human-computer games to see if the new designed strategy has reached the desired level. The goal is that the new engine is able to defeat an experienced human player.

1.3 Chapter overview

This document has been structured as follows:

- **Chapter 2: Introduction to connection games:**

Here we provide a description of what a connection game is, what exactly characterizes a board game as such. We discuss features like board setups, rules or usual developments of a connection game and finally we give some examples of them.

- **Chapter 3: Description of *Selfo*:**

In this chapter we focus on the connection game named *Selfo*. It is given a precise definition of boards, kind of moves, goals and many other features. In addition we discuss some important issues related to *Selfo*, like its dynamic or the initial configurations of pieces.

- **Chapter 4: Related technologies:**

Through this chapter, we discuss the discarded and chosen technologies. It is given a description of the *Zillions of Games*

platform, the ZRL language and the DLL interface to engage a custom game engine to *Zillions*.

- **Chapter 5: Proposal of a game strategy:**

In this section we propose an artificial strategy to play *Selfo*. We give the definition of the strategy according to our experiences playing *Selfo* and we explain its functioning and some aspect related to the implementation.

- **Chapter 6: Simulation results:**

Here we show simulation results and experiences playing *Selfo* using the new developed strategy. We give some statistic data and graphics; and we expose our conclusion in relation to the kind and level of play developed by our game engine.

- **Chapter 7: Conclusions:**

In this chapter we reflect on the obtained result. We comment the level of play achieved by the proposed strategy and think about the accomplished goals. Finally we expose some additional experiences and future lines of work.

Chapter 2: Introduction to connection games

2.1 Definition

A connection game is a board game in which players have to develop or complete a specific type of connection with their pieces. This might involve forming a path between two or more goals, completing a closed loop, or gathering all pieces together into a single connected group. In all cases, the size and shape of the connection do not matter; it is the fact of connection that counts.

Most board games feature at least some aspect of connection. This could be as fundamental as the adjacency of squares on a *Chess* board, or the fact that a winning pattern in *Tic-Tac-Toe* forms a connected line. But in both cases what really matters is not the fact of connection. Consider the game of *Tic-Tac-Toe*, we can set up a board position that connects opposite sides of the board or connects as many pieces as we want without either player winning. It is the pattern's size and shape that is important here (three pieces in a line), the fact of connection is an irrelevant by-product of its formation.

2.2 Board design

As board geometry is a central element in most connection games, any singularities in board design can have profound effects upon the game. Players should take advantage of powerful cells and stay away from weaker cells. Powerful cells are those with greater connective potential or at which connective flow converges. The central point is usually the strongest point on the board.

The right-most design, a hexagon tiled with hexagons called the *hex hex* board, is of particular interest. The absence of acute corners means that edge cells are more uniformly distant from the center of the board, resulting in a more even distribution of power over all cells.

The cells of a game board are generally adjacent to those cells with which they share an edge.

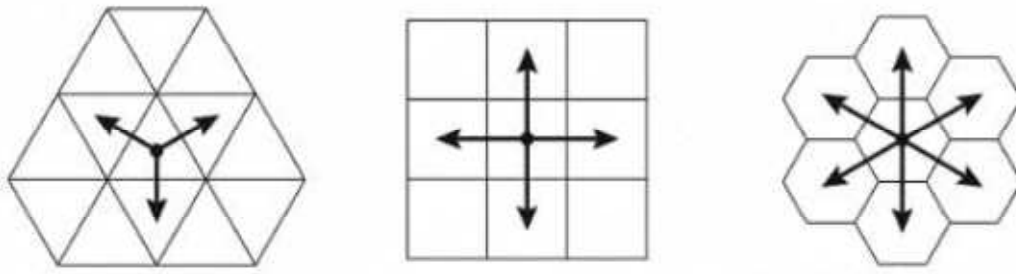


Figure 2.1 Adjacencies on the triangular, square and hexagonal grids. Taken from Browne [2005].

Some games are played on grid intersections rather than cell interior. In addition, the rules of some games specify connective adjacencies between cells or board points that are not physical neighbors; for instance, points in *Twixt* are not connected to their immediate neighbors but to those a knight's move away.

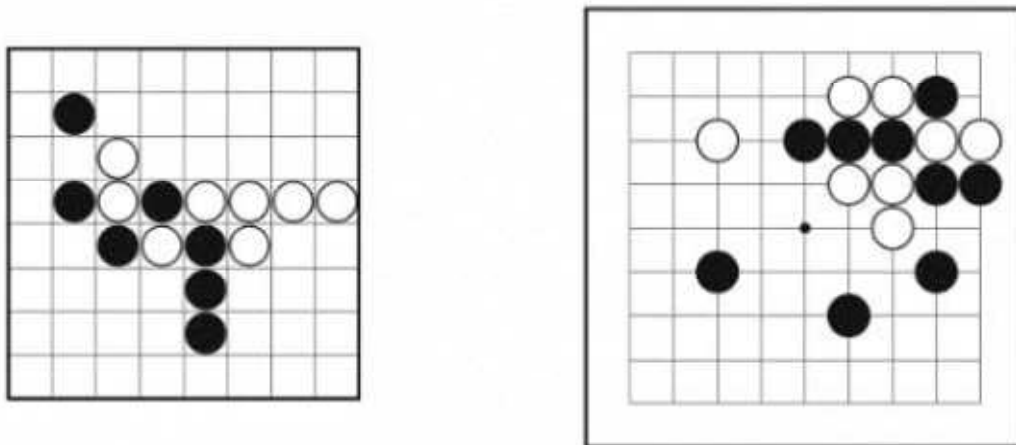


Figure 2.2 Games played on cell interior and on grid intersections. Taken from Browne [2005].

2.3 Scale

Most connection games tend to scale up well. For instance, *Hex* (see 2.10) can be played on a 20 x 20 board just as well as on a 10 x 10 board because the connections involved are independent of size. However, connection games do not tend to scale down so well. Smaller boards degenerate into limited tactical battles. Smaller boards offer fewer lines of play, are more readily analyzed, and tend to be overshadowed by combinatorial edge tactics that reward rote learning of book positions rather than quick thinking over the board.

Larger games are generally more satisfying, but can take much longer to complete. Players will generally find a board size that balances their skill and depth of interest in a game with their patience to play it.

2.4 Rules of play

Rules of play govern the interaction between pieces on the board, and shape the stages of play as the game progresses.

2.5 Clarity

Clarity is the ease with which a player can understand what is going on. Games like *Hex* with transparently simple rules and goals, and no special conditions or hidden complexities to distract the mind have excellent clarity. The clarity of a game determines how far you can see down the game's strategy tree. Games with overly complex move mechanics or excessive piece movement tend to have poor clarity and hence limited depth.

2.6 First move advantage

Most Pure Connection (see 2.9) games suffer from a severe first-move advantage as the opening player can win with perfect play. Move transformers are a balancing mechanism used in some games to counteract the first-move advantage. For instance, the move transformer 12333 means that the first move is a single move, the second move is a double move, and all subsequent moves are triple moves. It can speed up play but can also introduce unnecessary complexity, reducing clarity and making it extremely difficult to anticipate future moves and formulate strategies. The swap option is a more elegant way to address any first-move advantage. The opening player makes a move, and then the opponent has the option of either making a move in reply or swapping colors. The swap option discourages the first player from making an overly strong opening move.

2.7 More players

Even though the Cut/Join nature of connection games is ideally suited to two-player games, games involving three or more players can be implemented successfully. However, care must be taken because the possibility of deadlocks could be increased.

2.8 Defensive play

A good rule of thumb for complementary-task connection games is that defense equals attack. Blocking the opponent's connection by definition implies a win for the player. A strongly defended position is a good one, and players should resist the urge to play excessively aggressive moves that overreach and lead to disaster. An active form of defense is to exploit weak points of overlap in the opponent's potential connections. It is often important to block a group's progress across the board. Such positional defense is usually best done from a distance.

2.9 Classification

Connective Goal

Connective Goal games are those that end as soon as a specified connection, independent of size or shape, is achieved; connection is paramount in deciding the winner. Such games can be described as involving connection at a global or strategic level.

Connective Play

Connective Play games are those that feature at least some connective aspect and no non-connective aspects during general play; connection between pieces is paramount during play. Such games can be described as involving connection at a local or tactical level.

Non-connective aspects include unconstrained piece movements, jumps, or flips. Similarly, the movement, rotation, or removal of tiles to arbitrarily change connections disqualifies a game from this category. Moves must be primarily dictated by connection. The placement of pieces on the board from an outside pile does not exclude games from this category. Piece capture is allowed, as

long as it is strictly connection-based (as in *Go*) and does not involve size or shape constraints.

Pure Connection

Pure Connection games are those with both strictly connective play and strictly connective goals. These can be described as games involving connection at both the local and global levels.

2.10 Examples

Hex

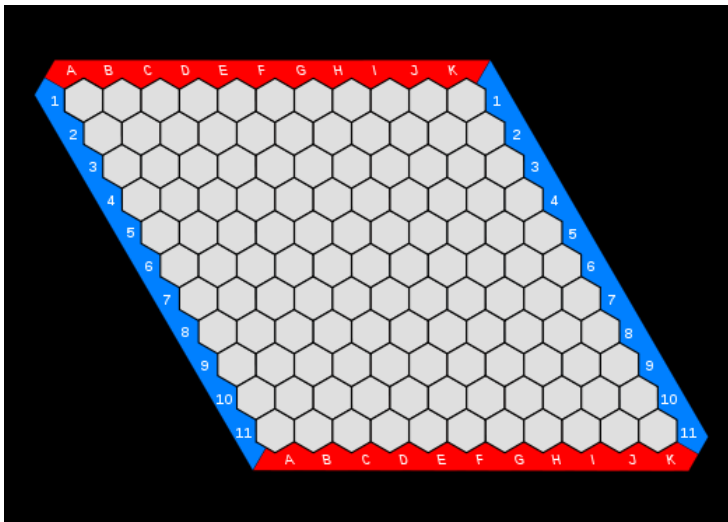


Figure 2.3 *Hex* board. Taken from Browne [2005].

Hex is the game that kick-started the connection game genre in the middle of the twentieth century. It has extraordinarily simple rules yet remains one of the most difficult and interesting of all connection games. It is included in Pure Connection class.

Hex is played on a rhombus of hexagons, typically 11 x 11, which is initially empty. Two players, Black and White, own alternating sides of the board that bear their color. Players take turns placing a piece of their color on an empty cell.

The game is won by the player who connects his two sides with a chain of his pieces. Exactly one player must win. The first player has a huge (winning)

advantage, especially if allowed to open near the center of the board. It is recommended that a single-move swap option be used.

Go



Figure 2.4 Go game. Taken from Browne [2005].

Go is one of the most influential of all abstract board games. It is played on the intersections of a square grid, typically 19 x 19, which is initially empty. Two players, Black and White, take turns placing pieces of their color on an empty intersection. All enemy groups with no remaining liberties (orthogonally connected empty points) are then captured and removed from the board.

Players may not place a piece that would commit suicide, that is, any piece placed on the board must have at least one liberty or become part of a group that has at least one liberty. In addition, players cannot make a move that would result in a repeated board position.

Players may pass in lieu of making a move. If both players pass in succession then the game ends, and player scores are calculated based on the amount of territory under or surrounded by their pieces. Captured stones may also contribute to player's scores, depending upon which version of the rules is used.

Go is widely regarded as one of the deepest games in existence. It is estimated to be 3,000 years old and boasts many textbooks, clubs, and professional players who spend their lifetimes studying the game.

Although strongly based on the concepts of war and territory, Go has a substantial connective aspect. In fact, capturing moves can be seen as the formation of orthogonally and diagonally connected cycles around maximal orthogonally connected enemy groups.

Lines of action

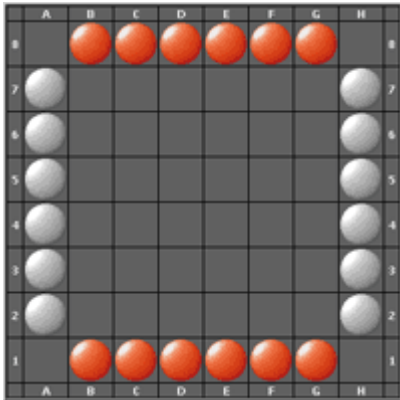


Figure 2.5 Initial configuration of Lines of action. Taken from Browne [2005].

Lines of Action, the classic game of convergence, is played on an 8 x 8 square grid. Two players, Black and White, start with 12 pieces each, setup as shown in Figure 2.5

Players take turns moving one of their pieces in an orthogonal or diagonal line. The piece must move exactly the same number of squares as there are pieces of either color along that line. The piece may jump over friendly pieces but not enemy pieces, and may not leave the board. The piece may land on an enemy piece to capture it.

The game is won by the first player to move all of his remaining pieces into a single connected group. Connections within the group may be orthogonal or diagonal. If a capturing move creates single connected groups for both players simultaneously, then the mover wins.

Lines of Action is widely regarded as a game of the highest quality, achieving deep tactical and strategic possibilities with a simple and interesting move mechanism. It remained something of a cult game until the 1990s when it started enjoying a wider audience (much like *Hex*).

One of the fascinating features of *Lines of Action* is that capturing an opponent's piece can harm a player's chances as much as improve them, as the opponent then has one less piece to connect to achieve his goal.

Chapter 3: Description of *Selfo*

3.1 Introduction

Selfo is defined as a class of abstract strategy board games subscribed to the category of connection games. Its name derives from the phenomenon of self-organization (i.e. the increase in a system's organization without external guidance), since during the game the sets of pieces might flow in a coordinated way as they step on the board. Despite its very simple definition ("group all your pieces by moving in turns to adjacent cells") complex self-organization processes takes place under concrete circumstances (a balanced distribution of pieces and similar levels of expertise in the players), and are the result of abrupt and deep changes in the tactics. Since a big number of variants have been found to meet the conditions for self-organization, the particular values given to the traditional parameters that define a game (i.e., board tiling, size and initial position, or number of pieces and players) are not so relevant. The *Selfo* class of connection games is defined, instead, by the interrelations among parameters in order to favor self-organization.

Since the invention of *Go*, traditional connection games have been widely played and studied. In the last decades this genre of games has proliferated, and now they constitute a significant contribution to strategy games. An ambitious compilation has been published by C. Browne [2005].

Some connection games are well-known and have become popular as board games: *Lightning*, the first connection game by several decades [Polczynski 2001]; *Hex*, devised by the mathematician and Nobel laureate John F. Nash, and whose publication [Gardner 1957] raised the connection game genre; *Y*, from which *Hex* is a special case, was proposed in the early 1950s by C. Shannon, the father of the Communication Theory; or *Twixt*, a game that has been marketed by six different companies since 1961.

Far from pure connection games, Browne classifies under the category "convergent connective goal" those connection games whose winning condition implies amalgamating a set of pieces into a single connected group. A number of games have been proposed under this convention, like *Lines of Action* (invented by C. Soucie [Sackson 1969]), or *Groups* (proposed in 1998 by R. Hutnik [Browne 2005]).

The *Selfo* class of games defined in this report subscribes to the convergent family, since the ultimate goal is to knit together the pieces. Despite their apparent similarity, *Lines of Action* and *Groups* do not belong to this class; some rules are added to the definition, and they have been designed for fast

games (usually under 10 turns in *Groups*). On the contrary, the fun of playing *Selfo* will be more in going through balanced positions (like in *Tetris*); with a tempo that switches frequently among evenly matched players.

3.2 Definition

This section describes the main rules that apply to a game in this class.

After setting the board's grid and size, move length, and the initial board position, each player is assigned a set of pieces, and a random order of play. Players move their pieces in turns, and after the first player's turn, a swap option is given to the rest of the player on their first move. The game ends when a set of pieces gets arranged into a single connected group, or when all players decide a draw (e.g. if some pieces get isolated).

3.2.1 Board

Selfo does not impose any restriction on the particular board topology. This aspect of the game is indeed critical, since the adjacency graph (based on a triangular, square, hexagonal, or even irregular grid) strongly influences the particular dynamics of the game. But, as said before, self-organized dynamics does emerge on any particular grid if the connectivity is balanced with other parameters. For simplicity, in this report we will constraint ourselves to board surfaces with hexagonal tessellations, where board points will be cell interiors.

With respect to board size, in principle *Selfo* can be played on a theoretically infinite board, where pieces are not confined, and can wander around without limits. This option reduces dramatically the possibilities of self-organized play, which easily turns into a race where any initial advantage cannot be neutralized by the opponents.

Limiting the number of cells introduces a major difference in the development of the game. Boundary effects clearly favor the use of tactics for isolating competing pieces over densely occupied finite boards. This balancing mechanism strongly favors the self-organization during the game, or, said in another way; it expands the range of the parameter space where self-organization takes place.

3.2.2 Initial board position

Some constraints apply to the initial position: the minimal number of moves necessary for a set of pieces to reach the winning condition (keeping the opponents' pieces on their initial cells) must be high (proportional to the number of pieces) and similar for all sets. Also, subsets of pieces cannot be isolated by competing chains of pieces, neither in the initial position, nor after the first moves (i.e. there must be a chance for any piece to avoid isolation).

All the pieces are placed on the board before starting the game (players cannot place pieces, as in *Go*). This can be done according to a fixed arrangement, or by an algorithm that, respecting the previous constraints, randomly assigns pieces to empty cells. Section 3.3 develops some examples of fixed initial positions, and algorithms for randomly sorting the pieces.

In order to avoid first-move advantage, a swap option is offered to each player. Every player can (only on their first turn) either make a move, meaning that they keep the set, or swap sets with any other player. This determines the final assignment of sets of pieces to the players.

3.2.3 Density

Defined only for finite boards, the density of a *Selfo* game is the relation, as a percentage, between the overall number of pieces and the number of cells of the board (i.e. a ratio of empty vs. non-empty cells).

The density has to be high enough to provide a strong interaction among the sets of pieces. But a too dense game will raise the probability of a deadlocked game. Densities in the range 35-40% have demonstrated to favor self-organized play.

3.2.4 Winning condition

The winning condition of any *Selfo* game is to form a single group connecting all the pieces of a player. The connectivity in this group will be assumed to be that of the adjacency graph of the board, i.e. two pieces are connected if they are in adjacent cells. For example, on a square board where pieces could only move orthogonally, the winning condition would be to form a single orthogonally connected group.

Players can also resign. Resigning must be announced on a player's turn, and the effect is like the player passing on the following turns: the pieces do not

move any more, they stay on the current cells, remaining as non-empty cells for the rest of the game.

3.2.5 Number of players

Selfo can be played by two or more players that are assigned the same number of pieces. Since the density of pieces has been defined as a very influencing parameter, and must be kept, the size of the sets of pieces will be the total number of pieces divided by the number of players. This size must be bigger than one in order to make groups, but less than four pieces per player is not recommended.

3.2.6 Move length

This parameter is defined as the maximum number of moves through empty adjacent cells that a player can perform with a single piece in a turn. The length of the move has a lower limit of zero (meaning that the player can pass the turn on). This is the parameter that influences the depth of the game more significantly, since the collection of possible movements increases exponentially with higher values of the maximum length allowed. For this reason the class will be divided into subclasses according to the maximum move length: *Selfo-1* being the simplest subclass, where any piece can perform single moves by stepping onto one empty adjacent cell (and players can pass on turns), and *Selfo-n* being the class where a piece can make a sequence of up to n single moves in a turn.

3.2.7 What a *Selfo* game is not

Possible refinements and extensions of the class can be considered, but in order to keep the simplicity of its definition a number of restrictions should always be met:

- all pieces move according to the same rules,
- players cannot influence the initial position,
- the winning condition should not be altered.

With respect to the first condition, having pieces with different behaviors would introduce a significant complexity in the definition. In such a case it would be necessary to specify a distinctive shape for each type of piece, and their role in group formation.

The second condition is also important, since it affects the simplicity of the strategy: moving pieces on a densely populated board is one thing, and placing pieces on an empty board according to some rules is another. To reach a balanced initial position would mean that the players match also in their skills to select an advantageous constellation of cells.

The proposed winning condition is a standard one (see the Introduction section), and any deviation from this simple goal will increase the solutions, giving less chance for a balanced play.

3.2.8 Summarized rule set

All the above definition can be condensed on the following rules for the simplest variant of the class, and two players:

- A board with a given topology and size, and a number of pieces are chosen,
- all black and white pieces are distributed on the board with a proper algorithm,
- colors are randomly assigned to players,
- starting with blacks, players move in turns, and a swap option is given to whites on the first movement,
- in each turn, a player can either pass on the turn, or move one piece to an adjacent empty cell,
- the player that first arranges all her/his pieces in a single connected group wins the game,
- players can decide a draw if the game lasts for too long or some pieces get isolated.

Or even more concisely:

Starting with an initial board position, players take turns moving a piece of their color an adjacent empty cell. The first player to connect all of their pieces into a single group wins.

3.3 Initial board position

Given that the expertise of the players match (or has been balanced somehow), the main point for a *Selfo* game to develop self-organized dynamics is to start with a board position where the sets of pieces are distributed in a way that do not favor the grouping of one nor the other sets. Some initializations are proposed based on regular and irregular distributions of pieces.

3.3.1 Regular positions

Figure 3.1 shows a number of fixed initial positions of hex hex boards of size 6 (for clarity, hexagonal cells are painted as circles), for two players (first row) and three players (last row).

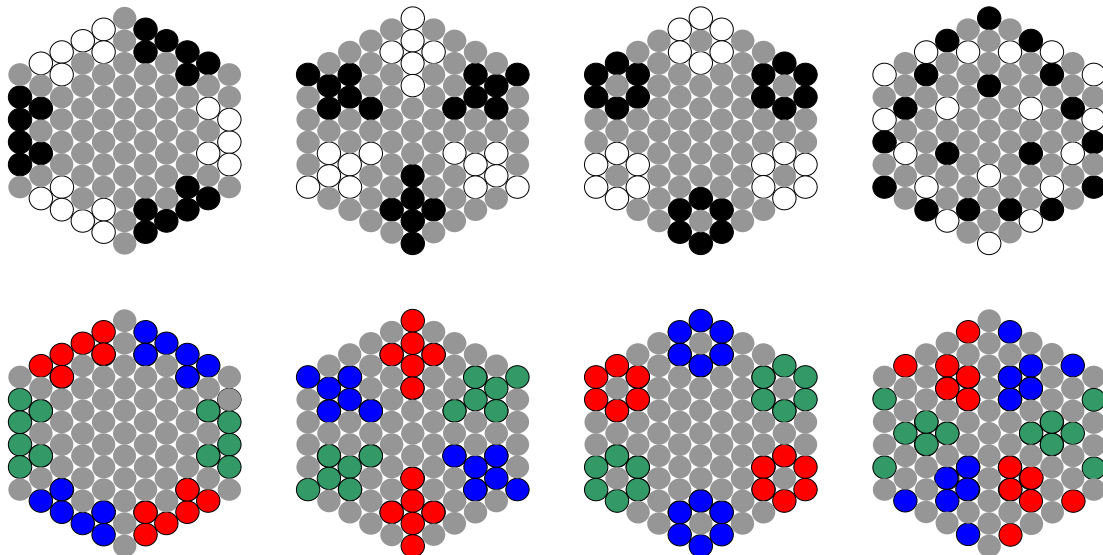


Figure 3.1 Regular initial positions for a hex hex board of size 6. Colors identify the different sets. Taken from Vico [2007].

Similar positions can be obtained for a higher number of players by re-coloring the pieces in a way that keeps the symmetry. All these board positions contain 36 pieces ($\approx 40\%$ density).

A number of regular positions for two players (can be extended to more players) derive from defining a pattern of pieces on one of the six triangles that form the hex hex board (see Figure -left), and copying it after reversing colors and rotating the pattern to fit the neighboring triangles. Figure shows an example of initial position after reproducing the pattern on the left. A variation would be to

define the pattern on two or three adjacent triangles, and copying and inverting colors three or two times, respectively.

This method is simple in its definition, and ensures a good distribution of the pieces on a hex hex board, whatever its size.

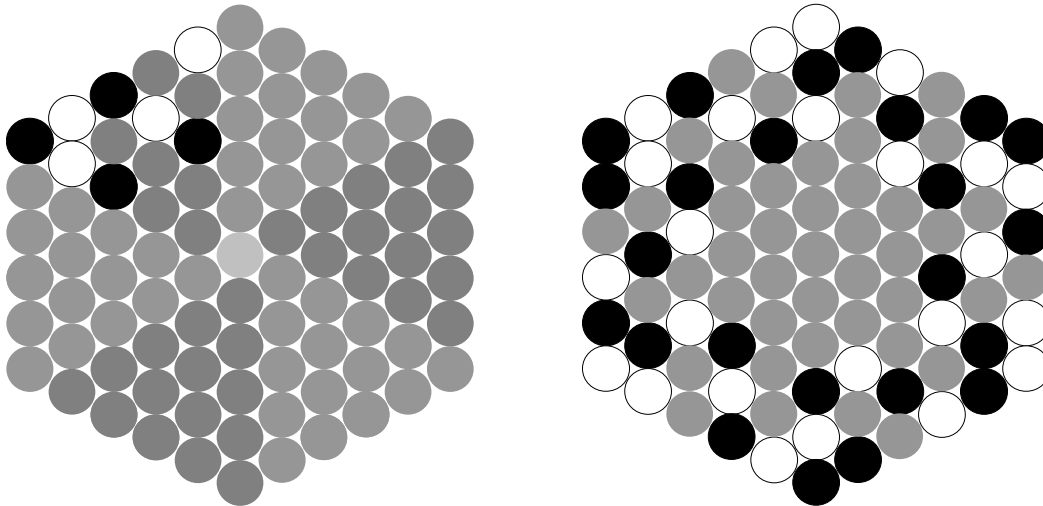


Figure 3.2 Shown in two differentiable gray tones are the six triangles that, arranged around the central cell, form a hex hex board; and a pattern of black and white pieces (left). The resulting position of 36 pieces (right). Taken from Vico [2007].

3.3.2 Irregular positions

Irregular initial positions of the board can be obtained by different algorithms. The main restriction is that the sets of pieces are balanced (see section 3.2.2). Two main strategies are proposed to determine an initial position: perturbing a regular position, and selecting random cells for each piece after indexing the board.

The first option starts with a regular distribution, and all pieces are randomly numbered. Starting from the first piece, a random number from 1 to 6 is selected that indicates a direction according to the scheme in Figure 3.2, the piece then steps onto the corresponding cell if it exists and is empty. This procedure applies to all the pieces on the board, and can be easily implemented on a physical board with a standard die. The result is a random rearranging of the sets of pieces that generally keeps a good distribution of pieces. Figure 3.3 shows an example starting with one of the positions proposed in the previous section.

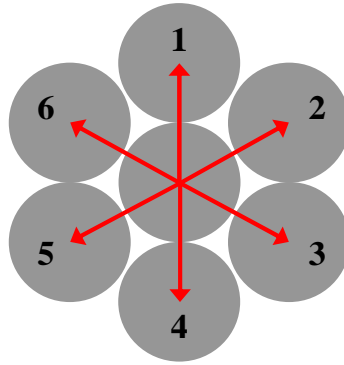


Figure 3.2 The six possible directions of movement from a given cell. Taken from Vico [2007].

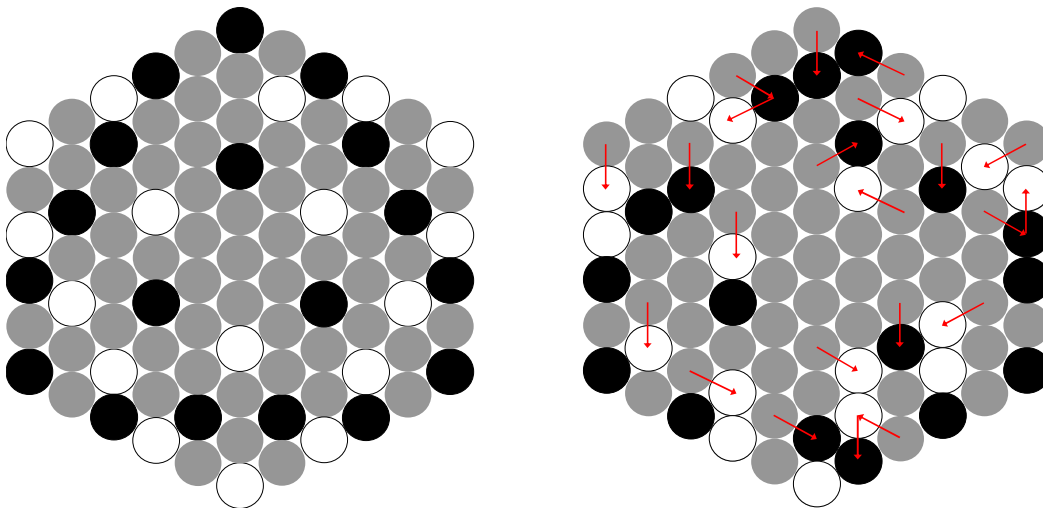


Figure 3.3 An irregular position derived from a regular one. Taken from Vico [2007].

The second strategy places the pieces following a spiral course that starts from the central cell and goes clockwise. In this case a random number is selected, and a piece of the first player is placed on the corresponding cell; counting from the central cell. A piece of the next player is placed after moving a random number of steps forward from the last piece. The process goes on until the last piece of the last player is placed on the board. This procedure can also be implemented with a die, renumbering the six faces accordingly (for example, on a size 6 hex hex board, numbers from 1 to six would be reinterpreted as {1, 2, 2, 3, 3, 4} for a 36% mean density). In case some of the last pieces cannot be placed on the board, the whole procedure should be repeated.

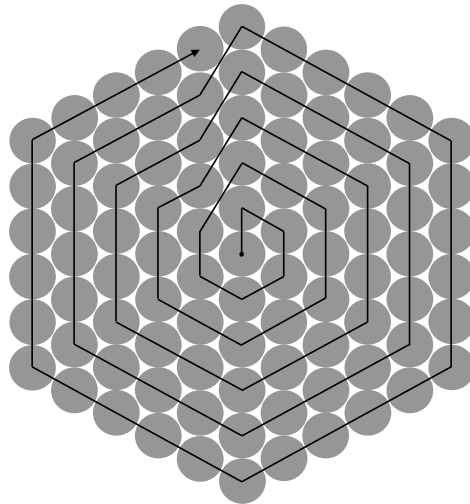


Figure 3.4 Trajectory followed on the hex hex board to place the pieces randomly. Taken from Vico [2007].

The proposed methods give an initial position that warranties a balanced distribution of the pieces. But it could also happen that a piece gets isolated (or can be isolated with little effort) in this initial setting. In such a case, the procedure should be applied again.

3.4 Self-organized dynamics

As explained in previous sections, the development of the match in a *Selfo* game starts with a fixed board position. After the opening movement, and the consideration of the swap option by the rest of the players, pieces start to occupy strategic cells, and to form small groups. Each set of pieces will converge to a configuration that minimizes the overall distance inside a set, while obstructing the opponents' options to group their pieces first. Given the conditions of equilibrium in the initial distribution of the pieces and similar experience among the players, the tightly coupled position reached during the opening should develop into a phase of self-organization.

But, what does it mean to say that the sets of pieces self-organize during the game? Self-organization is a process widely studied in the field of Complex Systems. Models of cellular automata are good examples of self-organized behavior, where structure (order) emerges from disordered initial states, after the iterative updating of the cells values with a local rule (e.g. Wolfram's 1D cellular automata [Wolfram], and the 2D automata based on the Belousov-Zhabotinsky reaction [Dewney 1988]). Self-organization takes place similarly in *Selfo* games, once a player's set of pieces is distributed on the board, they start to move according to local rules (i.e. the strategy of the player) to try to find an ordered configuration. The fact that adversary pieces try to do the same, while

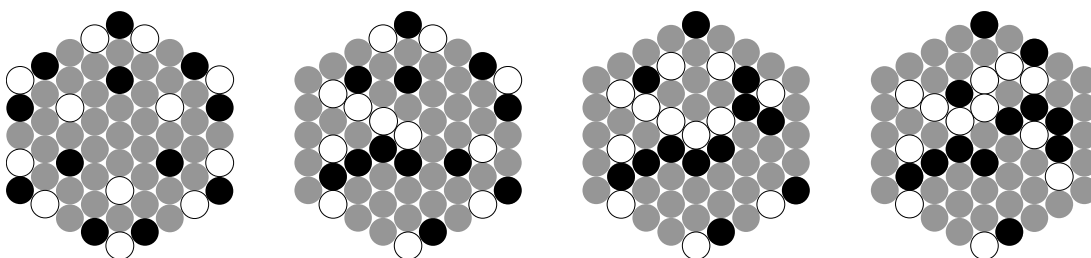
preventing others from grouping their sets, allows a balance of forces that makes the pieces flow on the board in a self-organized fashion.

Self-organization occurs only over long lasting matches. It is a direct consequence of rapid changes in the tactics (alternating offensive and defensive movements), forced by the opponents' recent actions, and it is characterized by global long-range displacements of the pieces on the board. This phenomenon can be measured in different ways. A simple indicator is the "mean number of turns since last movement" applied to a player's set of pieces. A value of this mean fluctuating around the size of the set of pieces reveals a mobilization of the whole set, On the other hand, values significantly higher than the number of pieces are representative of the set having found a stable configuration (pieces that do not change positions, while a small number of them wander around).

3.4.1 An example of self-organized play

Figure 3.5 shows successive positions of a hex hex board of size 5 during a *Selfo-1* game, played by players of similar experience. The initial board position is a regular one that soon shows the formation of some groups derived from the first black piece's movement. After 28 turns, the two players compensate their movements by establishing two ladders aligned horizontally (position after 18 and 28 moves). But in deciding who will be first connecting both sides of the board, tension grows on the upper-right corner (position after 38 moves). At this point the initial strategy changes, when the whites break the ladder, allowing some of the black pieces to cross it (positions after 42 and 49 turns), and converge to new diagonal ladders (position after turn 57) that finally align vertically. This option finally forces a draw when both sets of pieces reach a configuration where some pieces would become isolated before any whole set can get arranged into a single group (final position after 77 turns).

This simple example shows how self-organization takes place: the sets of pieces tend to balance configurations, and continue in this direction until the tension fractures the patterns, pushing the sets towards unstable sequences of positions that rapidly generate new patterns.



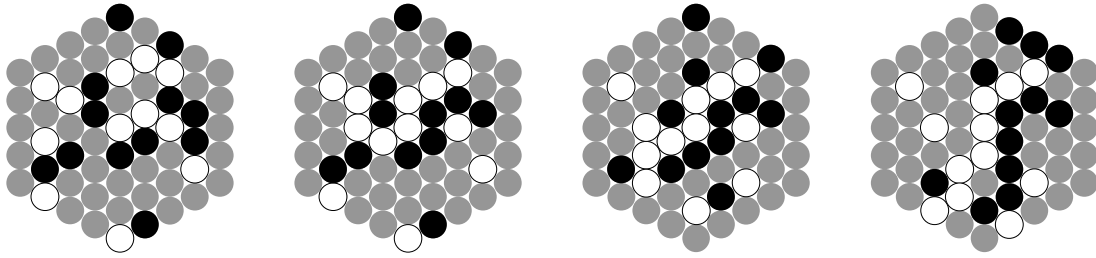


Figure 3.5 Evolution of the board position after 0, 18, 28, 38, 42, 49, 57 and 77 turns (from left to right and from top to bottom). Taken from Vico [2007].

3.5 Discussion

We are living in a time (the Internet era) where less and less time can be dedicated to learning from reading and mastering particular games. The success of new games is certainly influenced by this constraint. Traditional games, like backgammon or chess, could be categorized as complex to start playing, in the scope of the emerging family of online games and game consoles, operated by very simple rules and commands. Board strategy games will benefit much from the new Internet-based infrastructure, but simplicity will be a cutting factor for a game to become popular.

Taking this constraint as one of the main factors in game design, *Selfo* has been conceived in the spirit of E. de Bono's *L-game* [de Bono 1968]: simplest possible definition, a considerable depth, and indecisiveness of the game (playing all players perfectly, the winning condition would never be met). The result is a game where the only difficulty is in establishing the initial board position, but this handicap vanishes when the game is played on the computer, since the methods proposed in section 3.2.2 can be programmed. For the rest, a child starts playing *Selfo* correctly within minutes.

Interesting extensions of the *Selfo* class can be derived from assigning unequal forces to the players: variable number of pieces, or different lengths of move. In principle, move length looks like an adequate way to balance unpaired players with different degrees of expertise, but the fact is that a player with a slightly longer move, has in practice too much advantage. In general, the depth of a *Selfo* game is influenced by move length (more significantly), the number of pieces of a player, and the number of players.

Another variant might consider special pieces with a bigger (or unlimited) move length that help to block the winning strategy of an opponent. This proposal challenges the first condition imposed to the class in section 3.2.7, but it looks like the simplest variation when departing from a homogeneous set of pieces.

An important feature of *Selfo* is that it can be played with household stuff, by using the conventional chess (or draughts) board and pieces. For example, using the 16 pawns yields a 25% density, that works fine on an 8-neighbours-based topology. Initial board positions can be obtained from regular distributions or by iterating a pattern (4 times a square one, or 8 times a triangular one). Hexagonal boards can also be handcraft easily with a round cutting board.

Chapter 4: Related technologies

4.1 Introduction

As discussed before, it was needed a programming support to build the program's user interface and implement the game engine. It was also required a website to publish *Selfo* to provide greater disclosure and elicit feedback with players.

One option may have been: choosing a programming language from scratch and deploy the application; locate and reserve a space on the internet where to place it; and make a small campaign to publicize the game and make it known. But we knew the existence of a platform called *Zillions of Games* which basically consists of a website where people can publish their board games to share them with others and a program that can be downloaded from the web to run those games.

Zillions of Games is a well-known platform among the people who likes board games. Publish *Selfo* in this place would greatly help us to disclose it. We could play games with known people, against the computer, make simulations between non-human players; and we would also have a host of players willing to spend time in our game and willing to share their experiences and saved games with us.

On the other hand, making use of *Zillions* will allow us do the auxiliary job, like providing a powerful and easy to use user interface, in a quick and simple way. The *Zillions's* game interface includes all features you could ask for a programmed board game, such as choosing the color of pieces, choosing the type of players (human or computer), choosing between different variants of the game, providing user support, etc.

The integration of a game in *Zillions of Games* is a relatively simple and well documented process. It consists of three basic phases: definition of the game, game engine implementation and publication.

The definition of the game is done over a programming language called *ZRL* (Zillions Rule Language), created by *Zillions of Games* for that purpose. The syntax of that language is very similar to *LISP*. It has a set of sentences, types and structures to define the dimensions of the board, its topology, the number and kind of pieces, or the number of players who can participate.

Zillions of Games provides three choices to have an artificial player for custom games. The first option is using the universal engine provided by *Zillions*. The second option is implementing the engine using a tool called *Axiom*, created by an expert user of *Zillions* to assist people in creating their own games. And there is a third option which is implementing the play engine directly through a DLL, following a predefined interface.

In order to publish a game inside *Zillions of Games* it is necessary to make a package with a predetermined structure and containing files with the definition of the game, the images used on it and optionally the game engine. It is also necessary to write some information about the game to be included in the download section of the game page in *Zillions*. The package is sent by email to the administration of the platform and no later than one week is obtained confirmation of the publication.

4.2 Zillions of Games

4.2.1 Introduction

Zillions of Games is a platform that consists of two main elements: the website and the program *Zillions*.

The program *Zillions* has been developed to manage and run a series of default games and games created by users. This program provides an interface where the user can configure the application, run a game and interact with it. *Zillions* also includes a generic game engine ready to play any game defined over the platform and it also has the ability to interact with a possible custom play engine.

The website is designed to facilitate access to the latest version of *Zillions*, to games developed by individual users and obtain the games development kit. It is also a place where users can discuss their programming and playing experiences, and to get games-related information.

4.2.2 Web

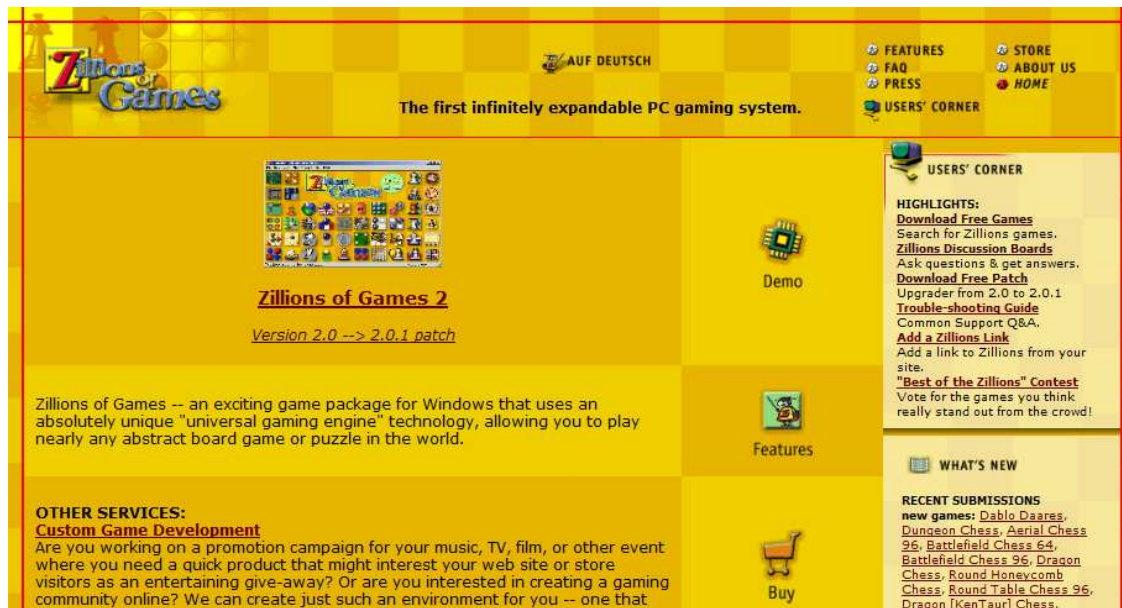


Figure 4.1 *Zillions of Games* homepage.

The *Zillions of Games* website is composed of a *main* section which allows the downloading of a demo version of program *Zillions*, purchase the full version and get updates. The *main* section also reports the existence of other services such as custom game development, free games downloading, the game development tools, an overview of all sections of the website and the latest published games.

In *Features* section it is explained in detail the features of the latest version of *Zillions* and it is offered the possibility to download the demo or purchase the full version on CD or through an unlocking code for the demo.

The *FAQ* section contains a helpful list of resolved questions about the acquisition and characteristics of *Zillions* programming platform.

In *Press* section, the references that have been made of *Zillions* in media and press dedicated to the world of computers and games are shown and discussed.

In *Store*, *Zillions* full version can be bought and demo can be downloaded and in *About Us* section it is provided some information about the *Zillions* platform.

And finally in the *User's Corner* section it is given access to those areas and most important tools for a *Zillions* user and/or developer. In the game download area, you can locate and get, close to 2000 different games based on the *Zillions* system. In other areas you can get updates of the program, the

development tools necessary for creating games, all sorts of information about board games, technical support, programming guides and a forum for sharing information between users and game developers.

Zillions of Games Here are some great games you can play with the Zillions of Games Interface!

User's Corner

- [Free Games](#) | [Patches](#) | [Discussion Board](#) | [Discoveries](#)
- [Installation Instructions](#) | [Share your Game](#)

You'll need the full version of Zillions to run these games, which you can get at the [Zillions Store](#).

Connection games

<name of the game> find Game 1 - 10 of 36 all Connection

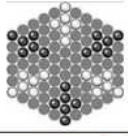
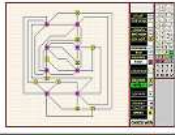
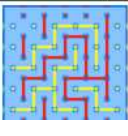
	<p>F.J. Vico's Selfo by D.D. Albarracín, 2009-11-28 (Updated 2009-12-05)</p> <p>Arrange all your pieces into a single connected group.</p>	<p>Connection 2-player download 245 K</p>
	<p>Roadmaps by Karl Scherer, 2008-03-08 (Updated 2008-03-22)</p> <p>Connect cities with roads.</p>	<p>Connection Solitaire 43 variants download 258 K</p>
	<p>Claude Shannon's Shannon by Greg Schmidt, 2008-03-01 (Updated 2008-04-19)</p> <p>Place bridges to connect the sides of your color.</p>	<p>Connection 2-player 4 variants download 94 K</p>

Figure 4.2 Connection games zone inside the Download Section of Zillions of Games website.

4.2.3 Zillions of Games program



Figure 4.3 *Zillions of Games* main window.

Zillions of Games program has a free trial version that allows access to 48 of the over 350 game variants that are installed; and disables features like opening and storing saved games, opening game rules files (games created by other users of *Zillions*), board editing and network / Internet play. The complete version can be obtained through an unlocking code from the demo for a current price of \$ 24.98 or on CD sent by mail with a current total cost of \$ 34.98. Currently the software is at version 2.0.1.

If we run the program, we can see a very friendly interface. It has a status bar at the bottom. The central part is occupied by a set of icons representing default games included in *Zillions*. At the top is the menu bar that allows us, among other things: load a different game to those that appear by default, load save games from any game, configure general options (sound, music, animations, graphical look ...), configure LAN parties, set the parameters of the artificial game engine and get help and access to the official website. Another way to start the program is making it through the *ZRF* file of the chosen game, in which case *Zillions* will be executed with the desired game loaded and ready to play.

When a game is loaded, the program interface adjusts itself to that situation. The bottom status bar now provides information that gets from the game; the

menu bar has enabled several options like saving games, printing the list of moves, choosing piece color, moving forward or backward along the states reached during the game, accessing to different game variants and getting some help and information about the current game. We also note that a new toolbar with buttons which provide access to the most frequent actions during the game has appeared. Finally, in the central area we can see the game board with the pieces placed according to the reached game state and a list of all performed moves at the right.

Handling the game interface is very intuitive. For instance, to move pieces over the board you just need to drag them using the mouse from the starting position to a valid final position. There are two important function that provides the interface which have not been mentioned: the *Start Thinking* button, used to indicate the engine may begin searching moves to perform (e.g. when established both players as non-human and want to start the simulation); and the *Move Now* button, used when the search engine is consuming too much time to find a proper move, so the engine will return best move found so far.

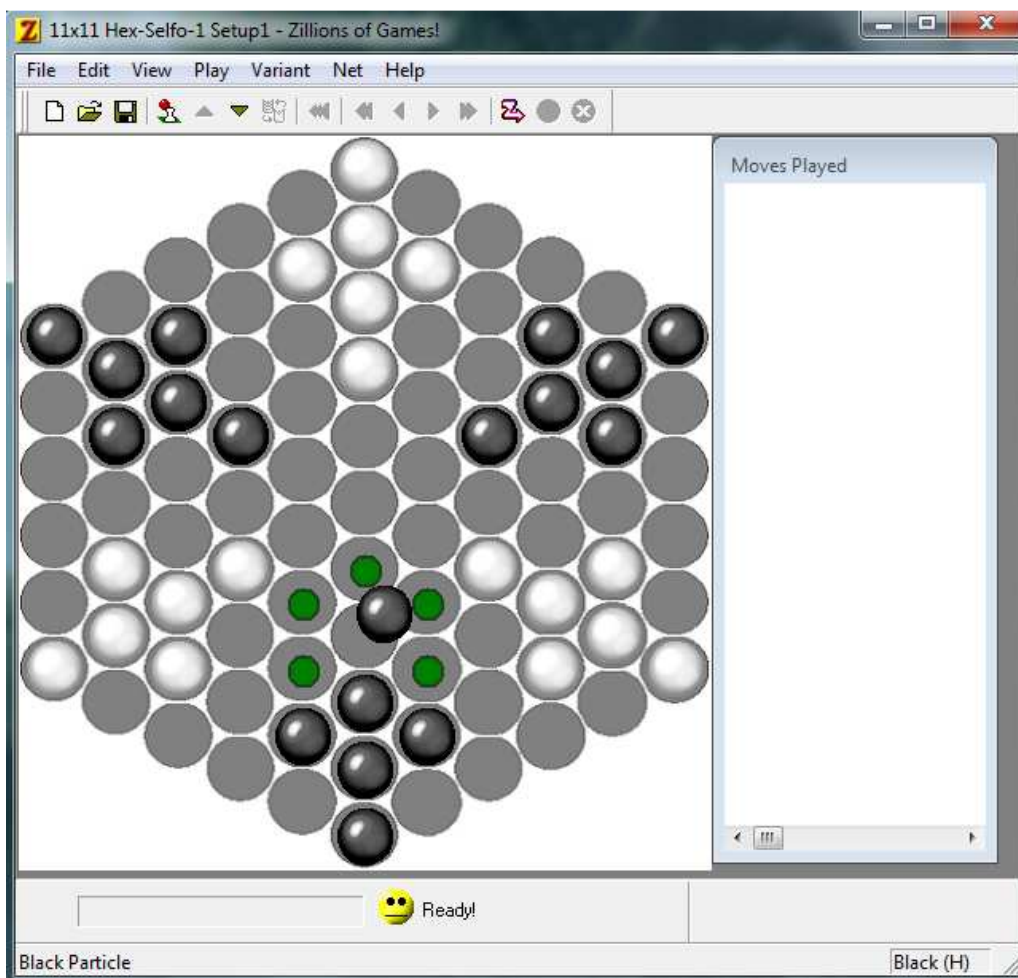


Figure 4.4 Playing *Selfo* inside program *Zillions*.

4.3 Building a game in Zillions

As already mentioned, a game created for *Zillions of Games* consists of a *ZRF* file which defines the game rules and refers to other resources.

Board and pieces images (BMP files) constitute the additional resources compulsory required. Optional resources could be sounds (win/loss/draw sound, background music...) or custom game engines.

For correctly integration of a custom game, it must be packaged according to the following structure: The main folder will be named like the game, inside this folder there will be the resources folders named with the resource type (audio, images, engines, include...), the *ZRF* file and an optional readme file. Inside each resource folder should be a folder with the name of the game and inside it there will be the applicable resources.

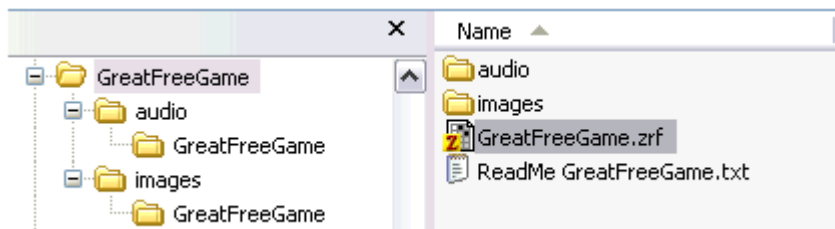


Figure 4.5 Game packaging.

Zillions of Games comes with a universal game engine which can run any game developed games inside the platform. This engine avoids having to create an independent game engine for simple games because it is capable of playing well such games, or if the game was designed to play between humans, for example, and the option of playing against the computer is a mere accessory. The default *Zillions* engine is generally very limited to play more complex games. In the web it is mentioned that this engine can behave properly in games similar to *Checkers* or *Chess* but it is really bad playing games whose win conditions are more complex to express in *ZRL* language, like connection games, where it behave like a random engine, fact that we have been able to test.

One of the options to implement a custom game engine is to make use of the *Axiom development tool*, created by an expert user of *Zillions*. *Axiom* is provided as a development kit designed to provide an interface between the programmer; and the *ZRF* file with the definition of the game and the DLL with the custom game engine. The game logic is defined in *Axiom* language, created for this purpose and based on *Forth* language.

Another option to deploy the engine is doing by directly creating a DLL that must implement a specific interface to which the program *Zillions* accesses

when needed. Because *Axiom* language provided no advantage in terms of versatility in the implementation of the engine and because it would have been necessary spending some time in learning that language, we decided to build the engine directly into a DLL using the *C* language.

4.3.1 ZRL Language

The game definition in *Zillions of Games* is made in the so-called *Zillions Rule Language (ZRL)*, whose syntax is based on *S-expressions*, in the same way as LISP language.

Those rules which define the game are stored in a *Zillions Rules File (.zrf)*. When one is selected, *Zillions* loads the *ZRF*, and uses it to find out how to run the game. *ZRF* files have four main parts: the board, the pieces, the goals of the game, and additional information like help and strategy.

Main parts:

```
(game
  ;...players, help and extra information
  ;...board definition
  ;...piece definition
  ;...goals of game)
```

Players:

```
(players X O)
```

This line tells *Zillions* the names of the players. In this case, there are two players named *X* and *O*.

Board:

```
(board
  (image "images\TicTacToe\TTTBoard.bmp")
  (grid
    (start-rectangle 16 16 112 112) ; top-left position
    (dimensions ;3x3
      ("a/b/c" (0 112)) ; rows
      ("1/2/3" (112 0))) ; columns
    (directions (n -1 0) (e 0 1) (nw -1 -1) (ne -1 1))
  )
)
```

The *image* statement tells *Zillions* what bitmap file to use to display the board. In this case, the file in *images\TicTacToe\TTTBoard.bmp* will be used. The *grid* statement makes possible to specify a regularly spaced set of positions. The *start-rectangle* tells *Zillions* rectangle of the upper left screen position. The

dimensions section has information about the placement and name of the positions. The *directions* statement indicates the directions linking each position (*n* for north, *e* for east and so on). The numbers after these names indicated which way to step for that direction. When Zillions reads this *grid* statement, then it will combine all this information to make a three by three grid, with names of positions like *a1* and *c3*.

Pieces:

```
(piece
  (name man)
  (help "Man: drops on any empty square")
  (image X "images\TicTacToe\TTTX.bmp"
    O "images\TicTacToe\TTTO.bmp")
  (drops ((verify empty?) add))
)
```

The *name* section gives this piece a name: *man*. The *help* section gives the text which Zillions will automatically display in the Status bar when the user points to the piece. The *image* section gives the bitmap names for Zillions to use for each piece and for each player. The *drops* section tells Zillions that this piece is dropped onto the board when it moves. The *(verify empty?)* section tells Zillions to make sure a position is empty before adding it to the board.

Board Setup:

```
(board-setup
  (X (man off 5))
  (O (man off 5))
)
```

The *board-setup* section tells Zillions that there are 5 men for each player off the board at the start of the game.

Goals:

```
(draw-condition (X O) stalemated)
(win-condition (X O)
  (or (relative-config man n man n man)
    (relative-config man e man e man)
    (relative-config man ne man ne man)
    (relative-config man nw man nw man)
  )
)
```

The *draw-condition* statement tells us that if any side is *stalemated* (has no legal moves left), then the game is a draw.

After *(win-condition* we see *(X O)*, the names of the players which this condition applies to. Next comes an *(or* and the *relative-config* statements tell Zillions that any position where a *man* piece is north of another *man* piece which is north of

a third *man* piece indicates a win. This line is repeated for the other three directions.

Creating Variants:

Zillions let defining game variants using the *variant* statement, were only the changing sections in new variant are included.

```
(variant
  (title "Same game"))
```

There exist many other statements which let us, for example, defining more complex move rules or more sophisticated ending conditions.

4.3.2 DLL Interface

In order to indicate the use of a custom play engine to Zillions, the statement *engine* is used in the *ZRF* file.

```
(engine "Engines\myEngine.dll")
```

To make communication possible between Zillions and the play engine, it is necessary the implementation of a predefined interface inside de DLL. This interface consists of four compulsory functions and two optional functions. To illustrate it we will use C language.

The engine returns a *DLL_Result* constant back to the Zillions, which should be *DLL_OK* under normal circumstances. If the engine returns a negative error code, *Zillions of Games* will report this to the user and then unload the engine plug-in. If an engine plug-in is unloaded, either for this reason or because it returned a move that *Zillions* did not recognize as valid, *Zillions* will revert to using its built-in, universal engine.

DLL_Result:

```
typedef enum {
  DLL_OK = 0,
  DLL_OK_DONT_SEND_SETUP = 1,

  DLL_GENERIC_ERROR = -1,
  DLL_OUT_OF_MEMORY_ERROR = -2,
  DLL_UNKNOWN_VARIANT_ERROR = -3,
  DLL_UNKNOWN_PLAYER_ERROR = -4,
  DLL_UNKNOWN_PIECE_ERROR = -5,
  DLL_WRONG_SIDE_TO_MOVE_ERROR = -6,
  DLL_INVALID_POSITION_ERROR = -7,
  DLL_NO_MOVES = -8
} DLL_Result;
```

Functions:

DLL_Search:

This function will be called from *Zillions* to obtain the best movement to perform in a given game situation.

```
typedef DLL_Result (FAR PASCAL *SEARCH)(long lSearchTime, long lDepthLimit, long lVariety,  
    const Search_Status *pSearchStatus, LPSTR bestMove, LPSTR currentMove,  
    long *plNodes, long *plScore, long *plDepth);
```

If it returns *DLL_OK* it should also return the best move found in *bestMove*, however, it should not make the move internally. A separate call to *MakeAMove* will follow to make move the engine returns. It can also return a negative error code.

lSearchTime: Target search time in milliseconds.

lDepthLimit: Maximum moves deep the engine should search.

lVariety: Variety setting for engine. 0 = no variety, 10 = most variety.

pSearchStatus: Pointer to variable where *Zillions* will report search status.

bestMove: Pointer to a string where engine can report the best move found so far.

currentMove: Pointer to a string where engine can report the move being searched.

plNodes: Pointer to a long where engine can report number of positions searched so far.

plScore: Pointer to a long where engine can report current best score in search.

plDepth: Pointer to a long where engine can report current search depth.

MakeAMove:

```
typedef DLL_Result (FAR PASCAL *MAKEAMOVE)(LPCSTR move);
```

This function executes inside the engine the move given by the parameter *move*.

StartNewGame:

```
typedef DLL_Result (FAR PASCAL *STARTNEWGAME)(LPCSTR variant);
```

It should prepare the game to play a new game. It can be influenced by the current *variant*.

CleanUp:

```
typedef DLL_Result (FAR PASCAL *CLEANUP)(void);
```

It must free memory and prepare the game unload.

Optional functions:

IsGameOver:

```
typedef DLL_Result (FAR PASCAL *ISGAMEOVER)(long *lResult, LPSTR zcomment);
```

This optional routine is called by *Zillions* to see if a game is over. If not present, *Zillions* uses the goal specified in the *ZRF* to decide the winner.

lResult: Pointer to the game result which the DLL should fill in when called. If the game is over the routine should fill in *WIN_SCORE*, *DRAW_SCORE*, or *LOSS_SCORE*. Otherwise the routine should fill in *UNKNOWN_SCORE*.

zcomment: Pointer to a 500-char string in *Zillions* which the DLL can optionally fill in, to make an announcement about why the game is over, such as "*Draw by third repetition*". The DLL should not modify this string if there is nothing to report.

GenerateMoves:

```
typedef DLL_Result (FAR PASCAL *GENERATEMOVES)(LPCSTR moveBuffer);
```

It can be used in the DLL to tell *Zillions* the legal moves for any position in the game.

moveBuffer: Pointer to a 1024-char string which the DLL should fill in when called. Initial call should be with *moveBuffer* set to "". Each call to *GenerateMoves* should fill in the next available move from the current position, with a final "" when no more moves are available. All moves must be in valid *Zillions* move string format.

The engine does not call *Zillions of Games*. However, during a search it can find out from *Zillions* whether it should continue searching. When *DLL_Search* is called, the engine should store away the argument *pSearchStatus* and then refer to it during the search. If the user requests that the program move now or the time has expired, the value will change to *kSTOPSOON*. In this case the engine should return a result as soon as possible. If *Zillions of Games* needs to abort the search prematurely, e.g. the user has chosen to exit the program; the value will change to *kSTOPNOW*. In this case the engine should return as soon as possible, whether or not a good result is available. The engine should not change the value of *pSearchStatus* itself.

During a search the engine should continually report its own search status by updating the values of *plNodes*, *plScore*, and *plDepth*. *Zillions* uses these values to display feedback on progress to the user. In order to display this feedback during the search, the engine needs to periodically give *Zillions* a chance to process Windows messages.

Moves are passed back and forth as move strings. These move strings are the same as those written out to a saved game, such as "*Pawn e2 - e4*".

For most games these strings are also the same as the move strings displayed in the move list, there are two exceptions: the moves involving setting piece attributes and partial moves.

After calling *DLL_StartNewGame* *Zillions of Games* will always pass down a series of board edits to place all the initial pieces on the board.

Chapter 5: Proposal of a game strategy

5.1 Initial study

After examining the platform functioning and the related technologies that we will use to build *Selfo* and its game strategy, we built an initial version comprising the game interface, all the necessary logic to define the game rules and a simple game engine. This engine was made in order to implement the *Zillions* imposed interface and to be used like an interface to the future and more advanced game strategy. The first version of the engine only consisted of a few simple rules to move the friend pieces to a certain area of the board. It was a very easy to defeat “intelligence” but it helped us to make the pieces move automatically with some criteria, so that sometimes avoided the need of a human player.

Tests using this first version allowed us to understand several aspects of the dynamics of *Selfo*:

We realize that thinking for a good move in *Selfo* do not takes a long time for a person. A move choosing is more an exercise in understanding the situation on the board than making a deep exploration into the possible moves that could happen in the future.

Another important and related issue is the fact that the goodness of a movement is not critical in this game, i.e. in general, there are not great moves or really bad moves; it has more to do with a good or bad concatenation of them. This usually leads to development of long games between two experienced players. To win in *Selfo*, a long-term strategy is needed more than execute a few brilliant moves. It should looks for improving the situation of pieces gradually, exploiting holes and blocking the opponent.

We also realized the importance of the initial distribution of pieces over the game board. On the one hand inequitable distribution can decant a game to one player from the beginning and on the other hand, an initial distribution where pieces are dispersed requires a higher concentration for a human player to understand the board state, is more difficult to perceive if the situation is favorable or not.

The study of the complexity of the game gave us some useful information as well:

Being N the number of squares on the board, taking into account that the optimal occupancy consist of around 40% of covered cells (20% for each color) and 60% of empty cells, considering the usually board in Selfo (hexagonal, 6-connective) and considering that between occupied neighbor cells and the limits of the board; on average, pieces has 3 possible movements. So that the branching factor of the search tree is around $(N / 5) * 3$. In the standard Selfo board: $(90 / 5) * 3 = 54$.

Complexity is relatively high compared to most connection games (except for Go) and because of nature of movement and the game goal, it's hard to develop a simple heuristic in order to indicate the goodness of the current board. Therefore, might be a good idea for the automatic strategy being similar to human in this sense, i.e. look for a method not based on a deep search but being sophisticated to deciding whether a situation is favorable.

5.2 An approach

Apart from traditional approaches to address the problem of creating an automatic playing strategy as the use of MINIMAX algorithms, the use of databases of patterns or learning making use of artificial neural networks, we wanted to base our game engine in the definition of a set of simple rules. The idea was to implement these simple rules on each piece, which were considered separate entities with certain autonomy of action. This mechanism combined with a few making centralized decisions should provide emergent behavior that would fit our problem.

The emergence in a given system consists on the appearance of complex behaviors or patterns from some simple interactions among its component entities. This concept, widely studied in fields such as philosophy, art or science, there has been present in many biological systems and help them in solving problems related to their adaptation to the environment. The human mind is considered an example emergency such originated in the neuronal interactions. This phenomenon is also present in the formation of flocks of birds or schools of fish or in the way ants organize to find the shortest path to the food.

The stigmergy is a related concept, which refers to the way in which many living organisms communicate with others in an indirect way by making use of the

environment, usually via pheromones expulsion and monitoring. Ants, for example, use this mechanism in its search process for the shortest path.

Our intention is the designing of a game strategy based on these ideas. And see if this behavior is sophisticated enough to compete with a person who knows how the game works and is capable of comprehending the board to decide a good move.

5.3 Definition of the strategy

Taking into account all the discussed ideas for designing the strategy and *Selfo* dynamics and goal, the fundamental idea, apart from many other calculations and further considerations is the need to construct a greedy algorithm to obtain a measure of the distance between particles, or rather between the groups of particles; and try to approach them.

The designed algorithm works as follows:

Game pieces act as entities that transmit signals across the board to report their presence and as receptors that capture the signals emitted by other particles. This signal is spread from one cell to its neighbors, decreasing the power at each step, so that a near particle receives a strong signal and a farther particle will capture a less intense signal.

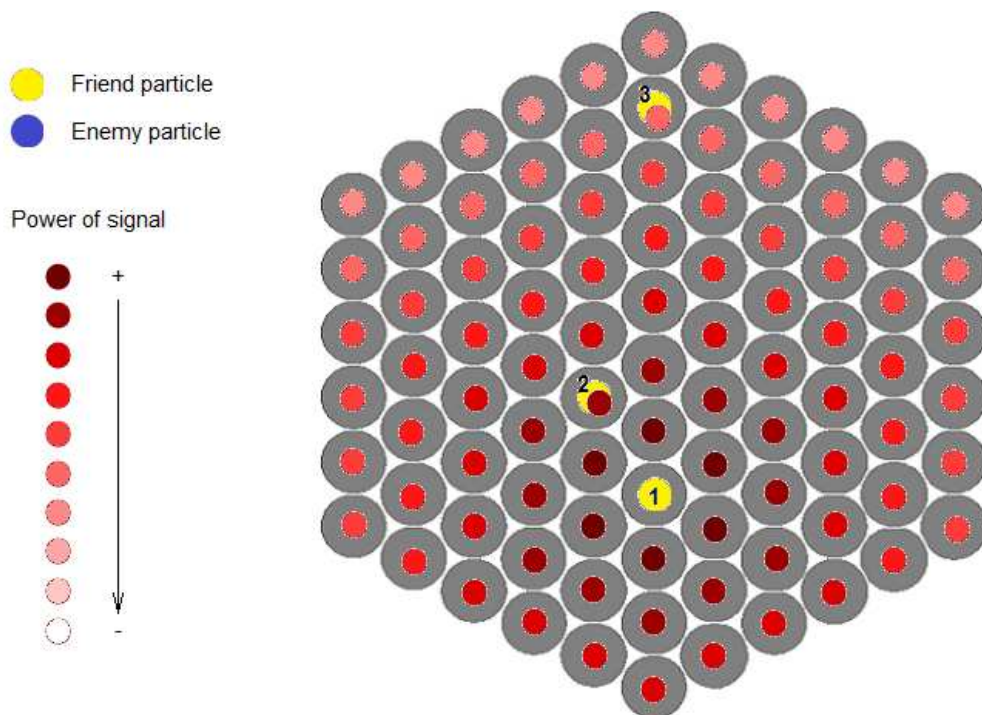


Figure 5.1 Shows the signal expansion for particle 1 and different power of signal captured by 2 and 3.

Besides, if two particles are connected, they will throw a joint signal (same signal identifier) with a higher intensity (at source, will be twice the intensity that gives a single particle) and so increasing the number of particles connected to the group, increases the intensity of the signal emitted by the group.

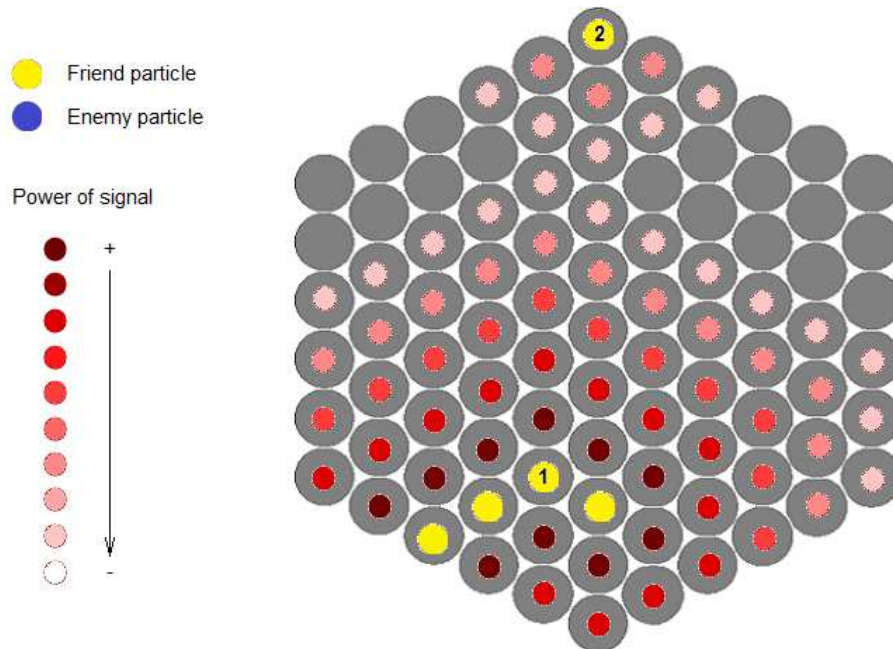


Figure 5.2 Shows the higher signal power for group 1.

A signal will spread along neighboring empty cells and neighboring cells occupied by friendly pieces belonging to another group and the signal propagation will stop when the board limit is reached and also when reaching an enemy particle, which will be able to capture the signal but prevent its spread. If a group of pieces is behind a group of enemy pieces, the signal must go around the group and will arrive with less intensity.

A particle at a distance s whose group has a size of gS receives the signal with intensity:

$$power = (InitialParticlePower * gS)^s$$

Thus, at a given time or state of the game, the game environment will be constituted by the board, the pieces placed according to the state and all signals propagated across the ground and generated by each group of particles, enemy or friends particles.

Once generated the gaming environment for a given state, the next step is locating the area where there is a greater concentration of signals from groups of friend particles. This will be the most important zone for the team and any proposed movement should take that into account.

Next task is locating the group that has more presence on that are. This will be the most important group and all moves must be oriented to favor connection of the other particles with it.

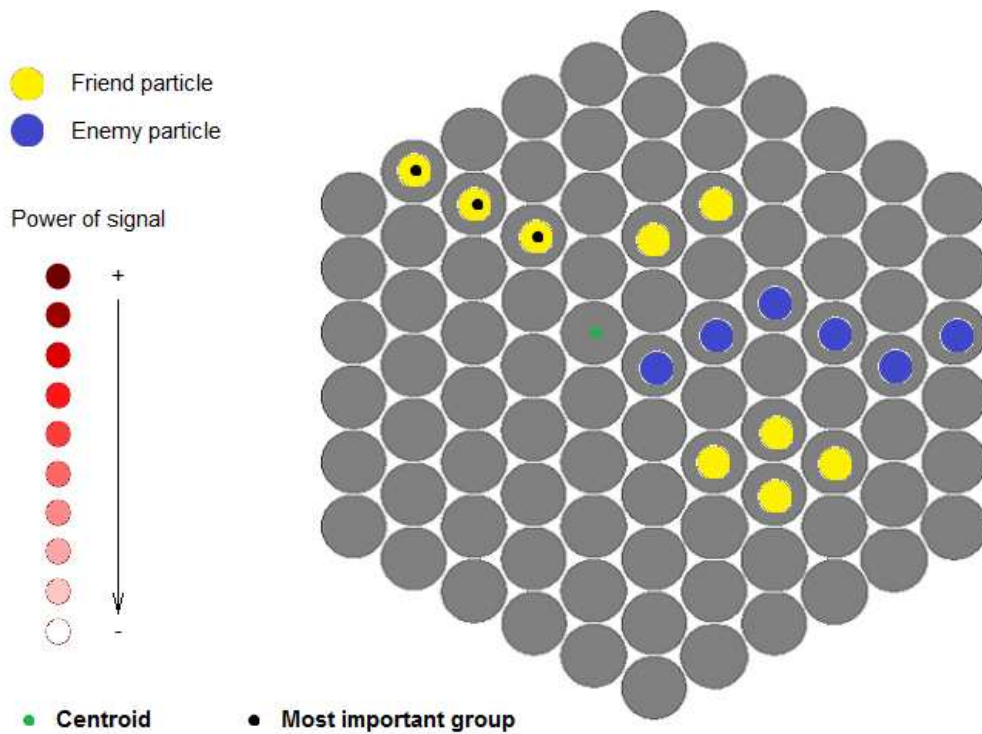


Figure 5.3 Shows where the centroid probably is, and which would be the most important group.

The information obtained through these steps will be combined to calculate a value that is indicative of good or bad arrangement of friendly particles to achieve total connection. The above process is repeated for enemy pieces and both values will be combined to obtain a score representing the overall goodness of the current situation for the given team.

The above algorithm is used in the final version of the engine as a heuristic helping in guiding a search process through possible future moves. The strategy game will perform a search beginning from a state and every time it needs to know the goodness of one situation, it will call the signal propagation algorithm. The details of implementation are discussed in the appendix.

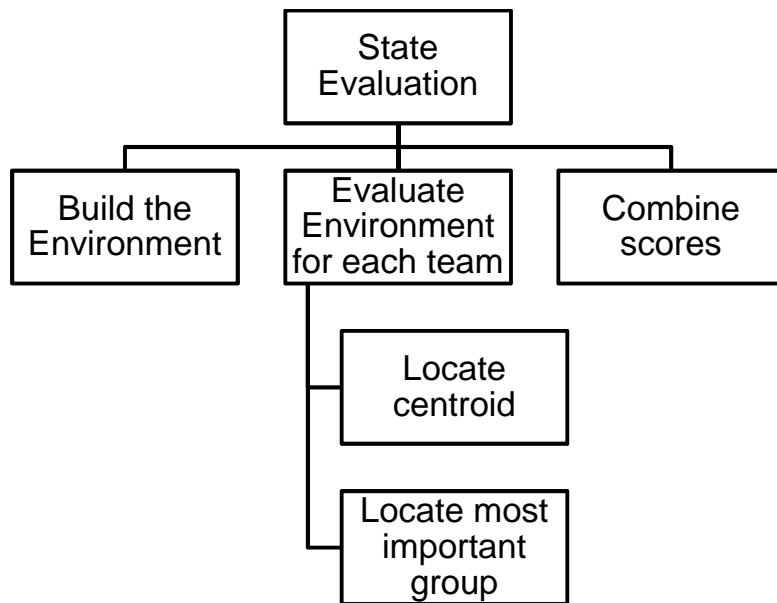


Figure 5.4 General scheme of the algorithm.

The algorithm developed to achieve connection of particles in Selfo has the same guidelines that the observed behavior of a species of amoeba known as *Dictyostelium Discoideum*. This species has a life cycle that lasts about 8 to 10 hours. One of the first phases of the cycle consists of consolidation of many of these spores in order to achieve the mature form of *Dictyostelium*.

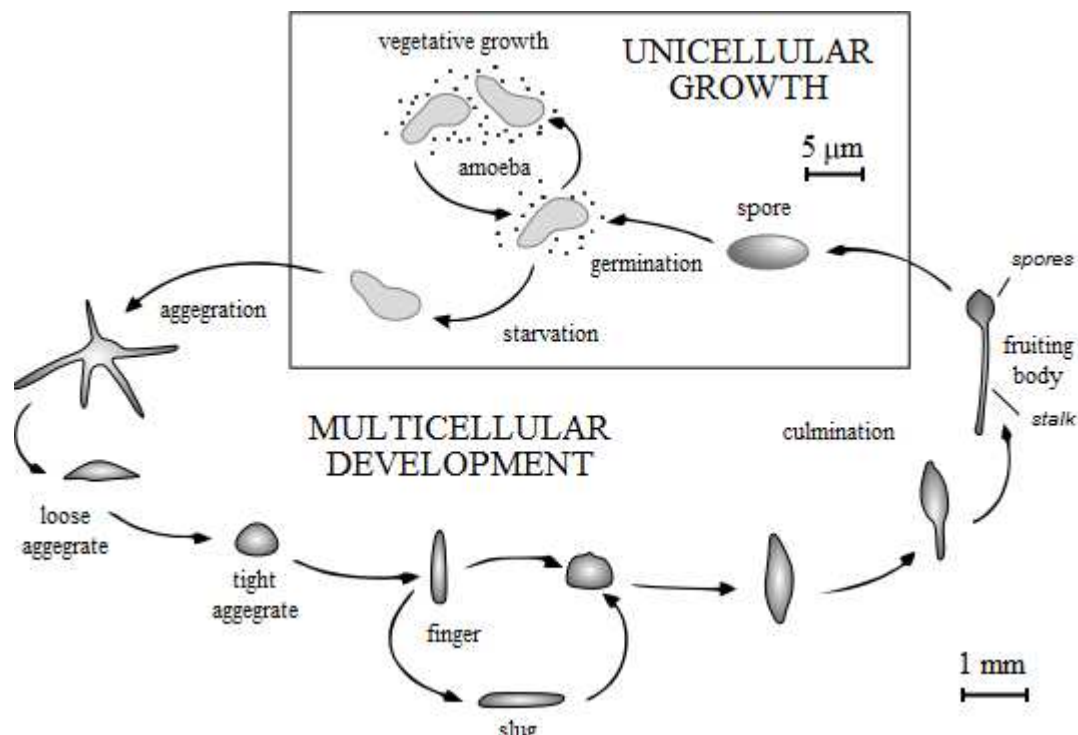


Figure 5.5 Shows the life cycle of the *Dictyostelium Discoideum*. Taken from http://en.wikipedia.org/wiki/Dictyostelium_discoideum.

In the process of grouping, a substance called cyclic adenosine monophosphate is secreted, which acts attracting external spores to converge

toward the central amoeba. This behavior consisting on following a chemical signal detected in the environment, through the gradient direction until the source of the signal is called *chemotaxis*. A previous version of the proposed algorithm worked in a more similar way to *Chemotaxis* of *Dictyostelium*, tracking the gradient of the stronger signal. But we had to make some modifications to fit the competition with the opponent and for possible inclusion in a MINIMAX strategy.



Figure 5.6 Dictyostelium Discoideum exhibiting chemotaxis. Taken from http://en.wikipedia.org/wiki/Dictyostelium_discoideum.

Chapter 6: Simulation results

6.1 Introduction

In order to check the level of play achieved by the proposed strategy, we published the game in *Zillions* again; we made local games between players whose level of play was known and finally we made several computer-computer games for collecting data like spent times, move trends, influence of the first move, initial distribution of particles, etc.

6.2 Human-Computer results

We made some games between the engine and human players. We classified human players into two categories: *Inexperienced players* and *Experienced players*. The division was somewhat subjective but in general we considered a person like an *Inexperienced player* if he/she has played *Selfo* less than 10 times, regardless the initial setup. The number of games for each setup depended on the interest (from previous experiences) using that initial configuration.

	Inexperienced player				Experienced player			
	LOSS	WIN	DRAW	TOTAL	LOSS	WIN	DRAW	TOTAL
Setup 1	14	1	6	20	3	7	10	20
Setup 2	3	1	2	5	0	3	2	5
Setup 3	4	0	1	5	0	3	2	5
Setup 4	16	0	4	20	4	5	11	20
Setup 5	5	0	5	10	1	5	14	20

Figure 6.1 Data from Human-Computer games.

In relation to the initial distribution we see that *Inexperienced players* have more chances to win using setups where pieces are already grouped, like in 1 and 2. When particles are scattered they can only force a blockade. *Experienced players* are able to win in any situation, although they are also better when pieces are grouped from the beginning. We realized that *Setup 5* tended to blockade; since there are more particles than in other cases and they are found near the board limits.

In general, games between *Inexperienced players* and the artificial strategy ended in victory for the latter in almost all cases. Thus the computer player was perceived as a program capable of connecting its particles using a lower number of moves, easily blocking and delaying the union of particles belonging to this inexperienced human player.

On the other hand, the result of games between a player with more experience and the proposed algorithm was more unpredictable. The experienced human player defeated the engine more times than he lost, but most of times game ended with draw caused by mutual blocking.

6.3 Computer-Computer simulations

The simulations have been performed using a laptop with an Intel Celeron 1.6 GHz processor.

6.3.1 Setup 1

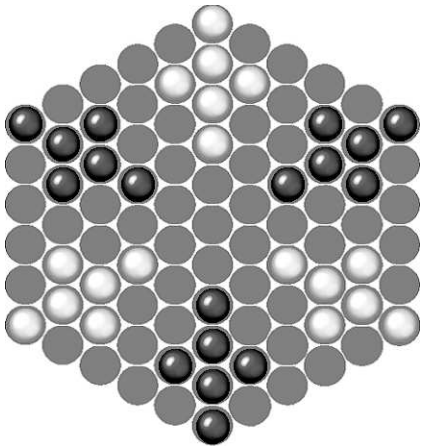


Figure 6.2 Setup 1.

Number of simulations: 20
Black Wins: 9
White Wins: 0
Draw (Stalemate): 11
Average move time: 0.54s
Average game time: 22.83s

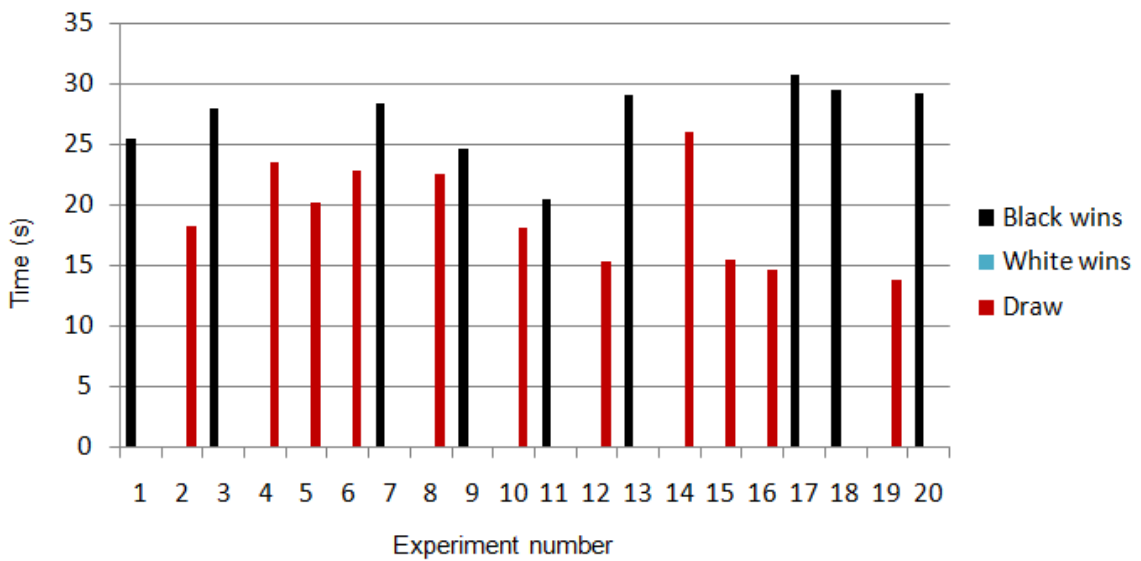


Figure 6.3 Simulation times for Setup 1.

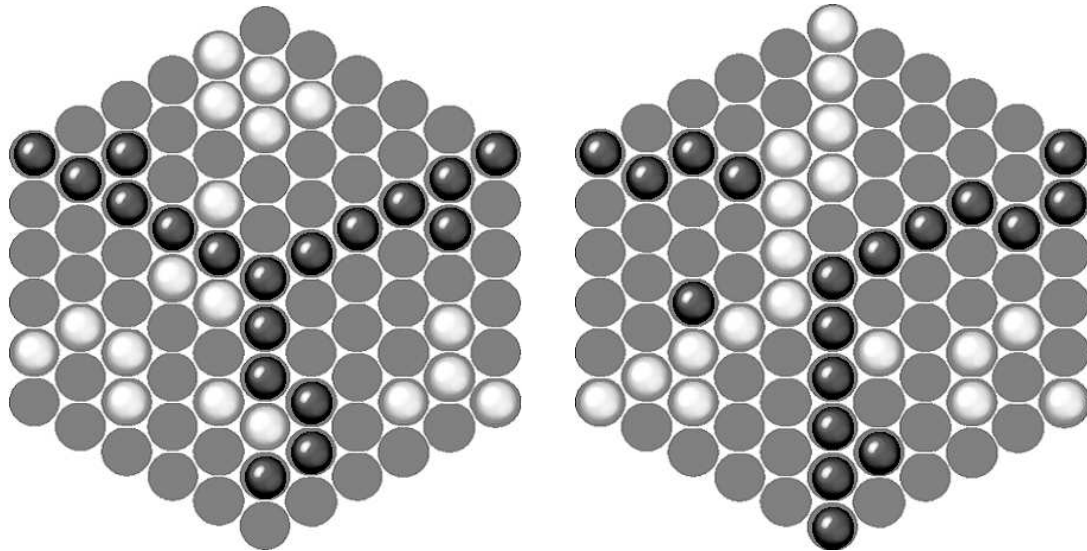


Figure 6.4 Some end situations for Setup 1.

Given the initial situation in this variant, both artificial players tend to get blocked quickly. If the black player gets a rather favorable situation at the beginning, the game is a little longer because the white player tries to delay his almost inevitable defeat.

The average time to perform a move is relatively low throughout the game. This is because particles are well organized from the beginning and there are not many moves to consider.

The games had a relatively rapid and predictable result because the pieces are organized from the beginning and particle clusters are very close, leading to rapid blocking or rapid victory of the player with advantage.

6.3.2 Setup 2

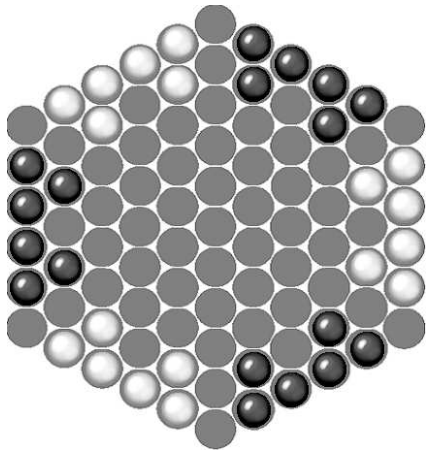


Figure 6.5 Setup 2.

Number of simulations: 15
Black Wins: 1
White Wins: 1
Draw (Stalemate): 13
Average move time: 0.49s
Average game time: 47s

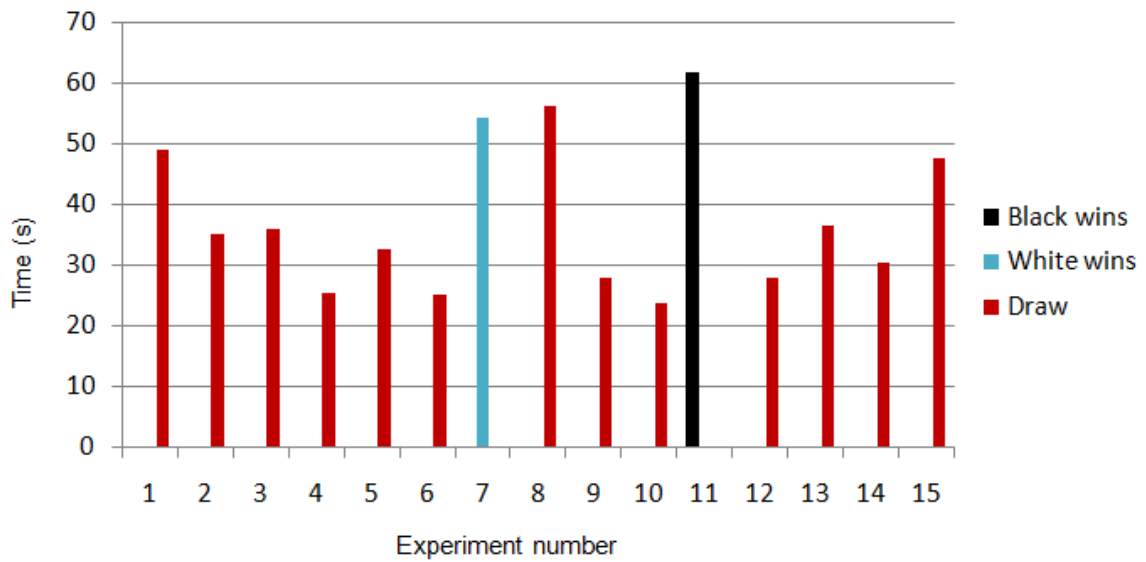


Figure 6.6 Simulation times for Setup 2.

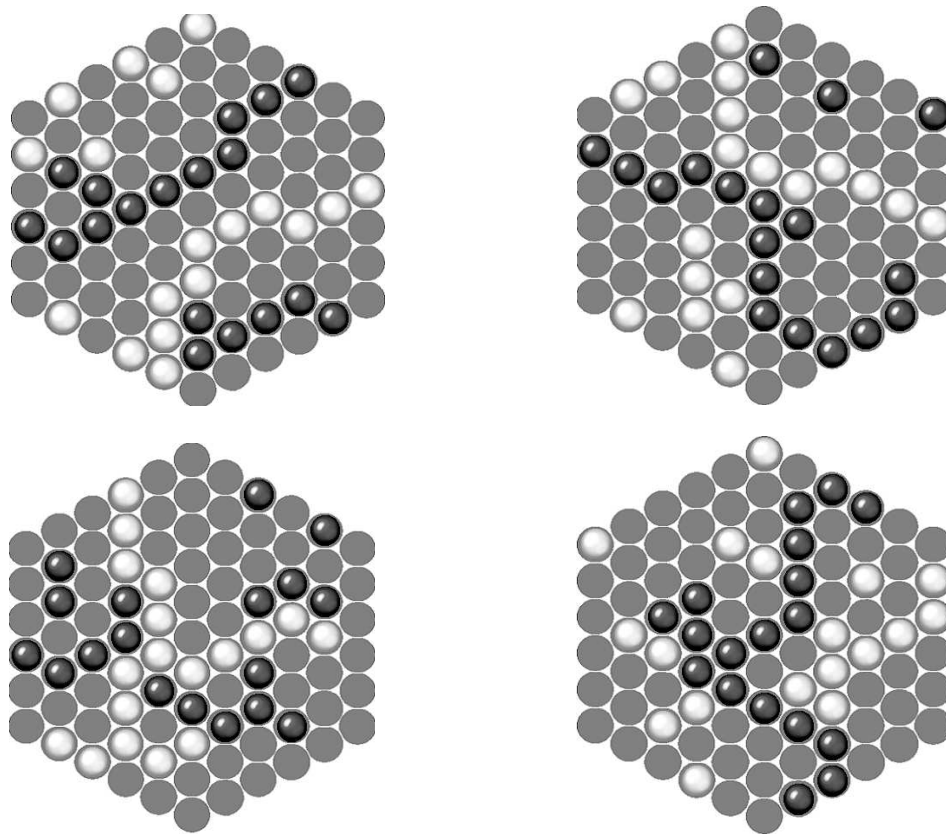


Figure 6.7 Some end situations for Setup 2.

Most games ended with mutual blocking. On a few occasions, games were finished with the victory of either player.

In this setup pieces are relatively clustered and set far apart from other groups. This caused the approximation group process tended to formation of a long chain of particles which tried to minimize the possibility of joining the pieces of the opposing team. This behavior usually led to a mutual blockade. Sometimes one of two players got enough advantage to prevent the formation of the enemy side-to-side chain and won the game.

The game took longer than using first setup, because the pieces are farther from each other. Average time of move was also lower and it was almost constant throughout the game.

6.3.3 Setup 3

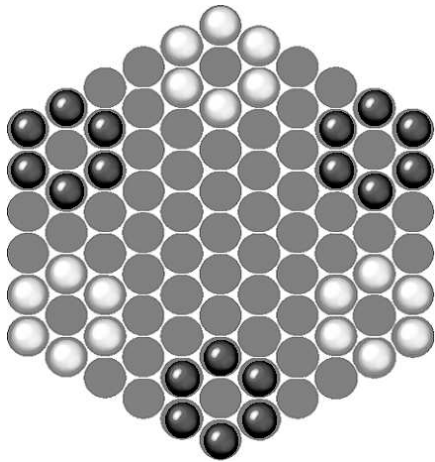


Figure 6.8 Setup 3.

Number of simulations: 10
Black Wins: 1
White Wins: 2
Draw (Stalemate): 7
Average move time: 0.63s
Average game time: 37s

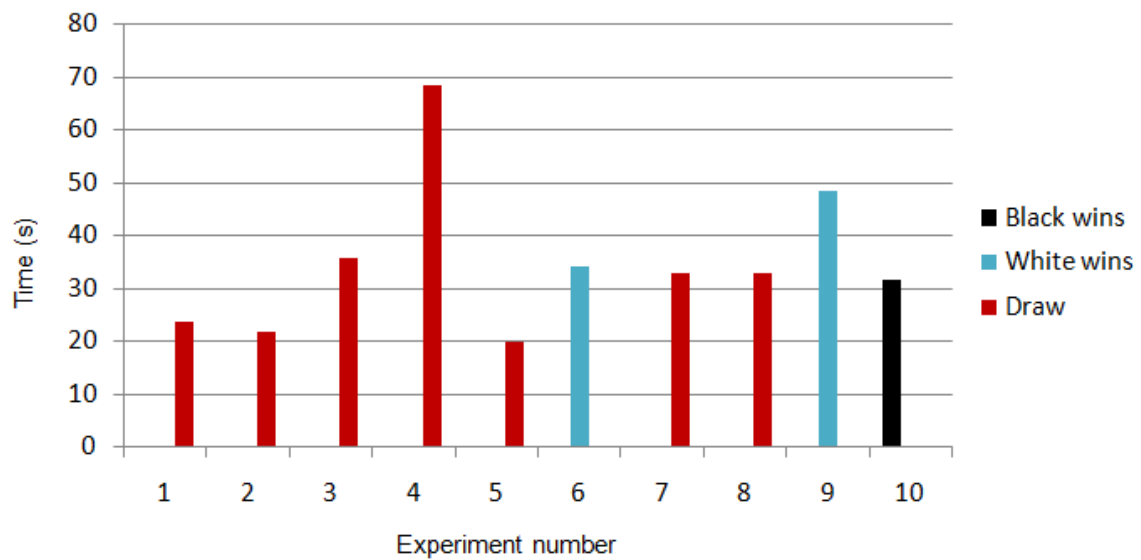


Figure 6.9 Simulation times for Setup 3.

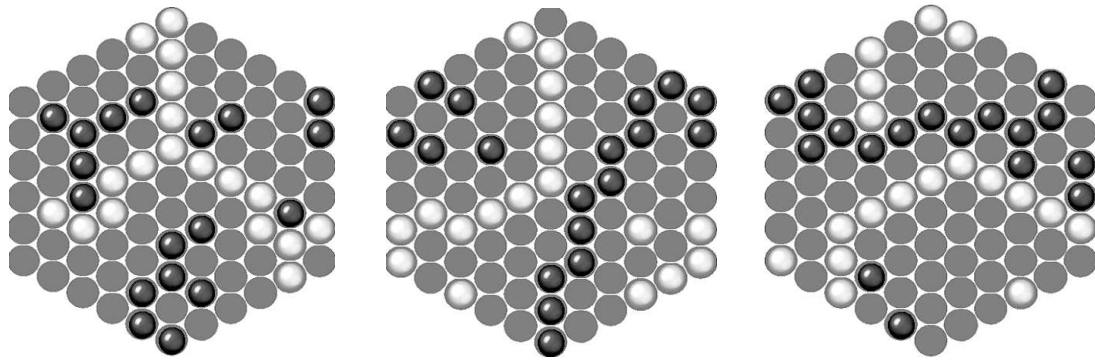


Figure 6.10 Some end situations for Setup 3.

Pieces are grouped at the beginning of the game and there exists considerable distance between clusters of the same team, though not as much as in Setup 2. These causes less mutual blocking occur here, although it remains the most frequent result by far.

Game time is similar as before, although a little lower. Move time is observed slightly higher, partly because there are more possible moves in the initial state.

6.3.4 Setup 4

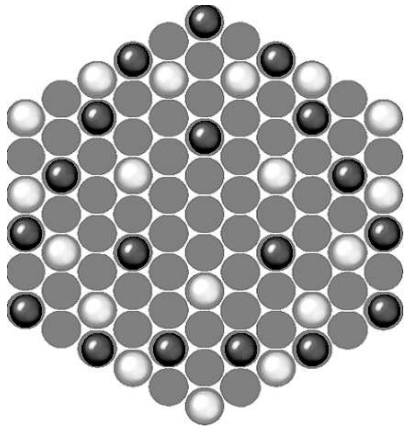


Figure 6.11 Setup 4.

Number of simulations: 20
Black Wins: 3
White Wins: 9
Draw (Stalemate): 8
Average move time: 0.76s
Average game time: 1:07.56

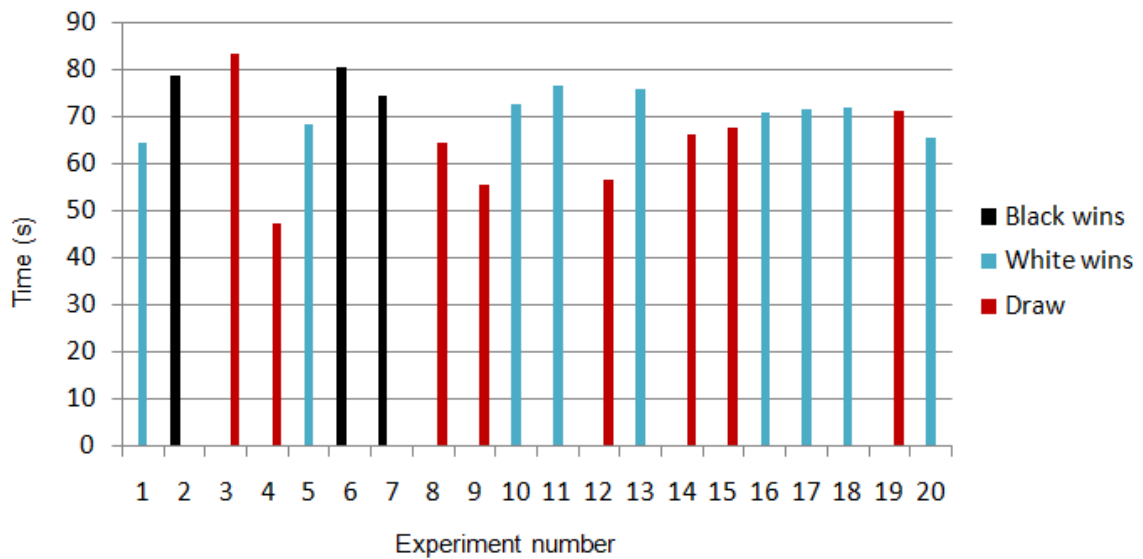


Figure 6.12 Simulation times for Setup 4.

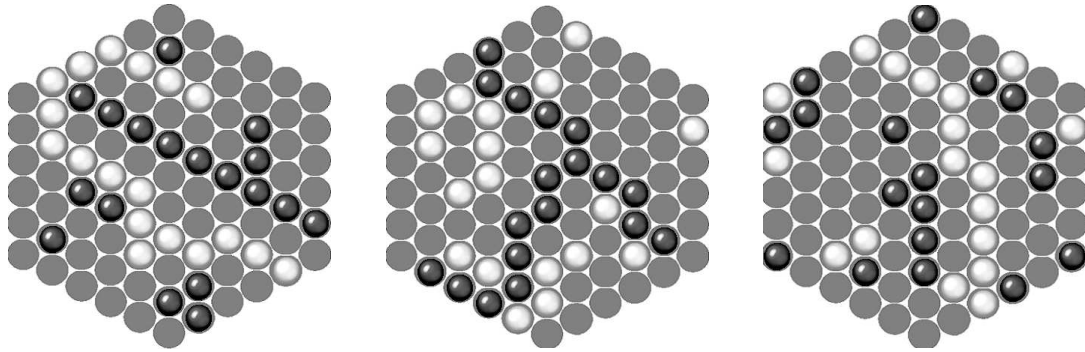


Figure 6.13 Some end situations for Setup 4.

In this occasion, we have an initial situation where pieces are widely scattered. Because of this, the end of the game is much more unpredictable than in previous cases. In fact, we note in simulations carried out that one player gets an advantage over the other very frequently and this advantage often leads to victory.

At the beginning of the game, time to make a move is significantly higher than in previous settings. This is because the particles are widely dispersed and there are a bigger number of possible moves. As the game progresses the move time will equate to other setups.

The average game time also increases considerably. On the one hand because of the higher move time. On the other and because games here are developed with a greater number of moves. Stabilization in a mutual blockade or victory situation always takes longer than in previous cases.

6.3.5 Setup 5

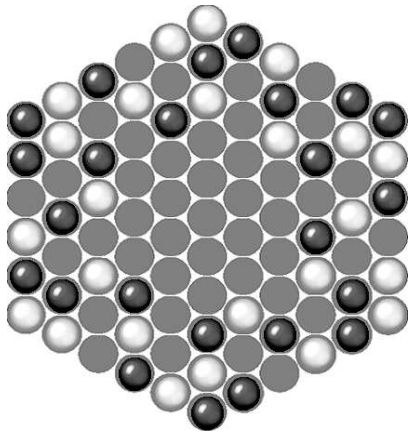


Figure 6.14 Setup 5.

Number of simulations: 10
Black Wins: 0
White Wins: 0
Draw (Stalemate): 10
Average move time: 0.62s
Average game time: 24.87s

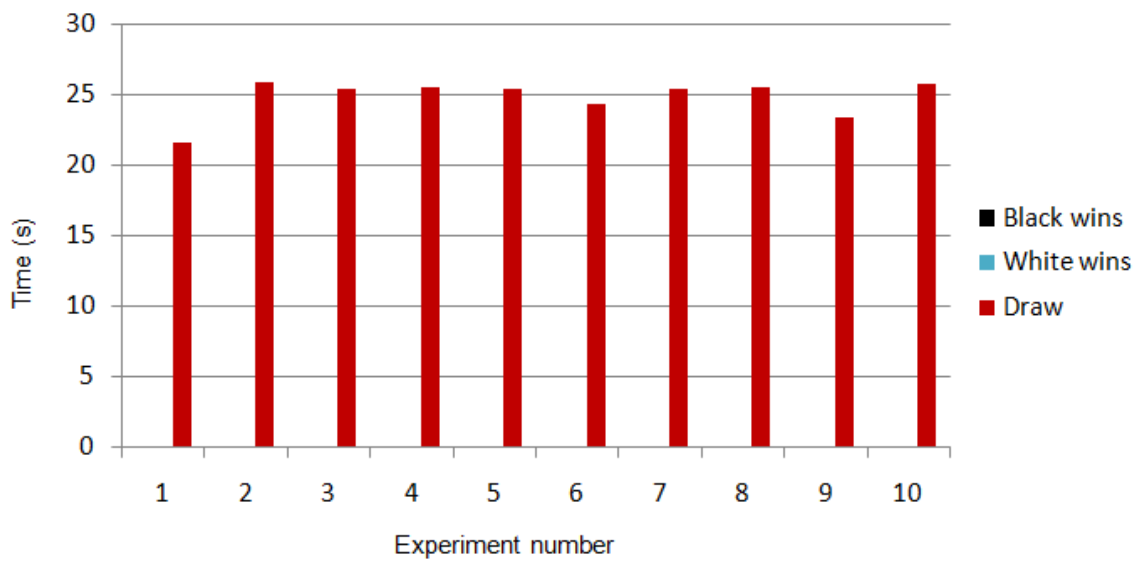


Figure 6.15 Simulation times for Setup 5.

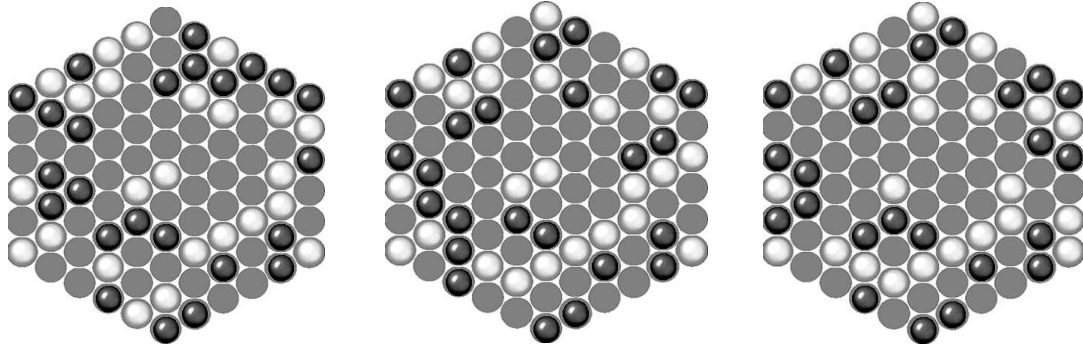


Figure 6.16 Some end situations for Setup 5.

In this configuration there are more pieces than in previous. The number of possible moves is large but not as much as in Setup 4, which is reflected in a high average move time.

Because of the number of pieces in play and because the pieces are all scattered in the area near the limits of the board, the game is ended in very few moves (no more than 20). The end is always a mutual blockade of very few pieces.

6.4 General observations

In relation to the advantage of making the first move, we have observed that is a very important factor in cases like the first configuration, a key factor in computer-computer games. But also see that, although beneficial, a strong movement is not as crucial when the pieces are not so close together, i.e. when the duration of the game is potentially higher.

Another aspect observed is the kind of game that makes a human player and the developed strategy, according to the initial position of particles. Usually, when the pieces are grouped at the beginning, a human player tries to connect their groups and tend to notice the enemy blockades. However, when the pieces are scattered, the human player does not realizes blocking situations. On these occasions, the blockades usually involve a small number of particles. An experienced player usually perceives the situation and makes a blockade too, in order to force draw. However, a not skilled player does not usually aware and continues connecting the pieces as having a chance to win. When he is aware of the situation may be too late or he may cannot be able to execute a successful blockade.

For a long and interesting development of a game in *Selfo* from the point of view of the phenomenon of self-organization, the initial location of particles is crucial. The number of pieces on the board and their distribution should be studied jointly and carefully. Distribute the pieces, so that friend particles are relatively dispersed at the beginning and not in some large groups, is often a good option, especially when using the developed game engine.

Chapter 7: Conclusions

The realization of the project has been a good experience, through which I have acquired a wealth of new knowledge and skills and allowed me to apply those acquired during the career: I have read and learned many ideas and technologies related to bio-inspiration, hitherto unknown to me; I have improved my ability to research and also my ability in drafting and project management have increased; I had the opportunity to exercise the use of English through the drafting of this document, the research and also the interaction with the administration staff of *Zillions*, via email; and I have acquired knowledge related to software development, like use and creation of dynamic libraries, handling of new development environments and adapting software created by me to interact with some commercial software. Finally, in relation to what I have learned during the career, making this project has let me applying knowledge about programming in Lisp, C and C++ languages; knowledge related to artificial intelligence, software design, analysis and design of algorithms; and all my skills in programming.

About the *Zillions of Game* platform, we noted that the needing of paying for a full version which permitted the execution of custom board games is a limiting factor to the disclosure of a game. At the end, *Selfo* was only tested by regular users of *Zillions* and by those known people, who were provided by us with a full version of the program. However, after overcoming this problem, the implementation of a game is not very difficult and getting access and executing a game developed by a user is extremely simple. The support received in the forum and by the administration of *Zillions* has been also very successful. They are people know very well the world of board games and most of them have high knowledge of programming.

Learning to define a game using the *ZRL* language is not difficult; the development kit downloadable on the website provides a good documentation. Programming inside the *Zillions* platform let us focus only in the rules and other aspects related to our particular game, avoiding the tedious work of defining the entire user interface, which should be substantially similar to any board game. On the other hand, the implementation of the game strategy had a problem, since it was found no documentation about the DLL interface to be implemented, so we had to learn about it by examining another user custom engine.

Regarding the proposed algorithm, we can say that we have found a game strategy with a medium level of play. It is able to beat a human player in some cases, to force a lock on many occasions and usually only defeated if exploited its weakness or the initial distribution is not favorable to our proposal algorithm.

In the various tests performed by human-computer games, we note that the artificial engine is generally faster than a human in grouping its pieces when they are relatively sparse (Setup 4) but not so when they have some level of grouping at the beginning and groups are considerably far (Setup 2). However, the major weakness of this strategy is the difficulty in predicting "total blockade" (a piece or group of them that is completely isolated from other friends particles). The strategy is not usually able to avoid enemy blockades, although usually makes them well.

In general, the strategy found is very fast in relation to the quality of provided move. Its execution consumes less than a second in a game with a high number of possible moves and under a lower-performance computer. This allows not changing the pace of play. And also give us a large margin to enhance the algorithm in the future.

The achieved execution fits perfectly with the fast dynamic but prolonged by the number of moves that characterizes *Selfo* and many other connection games. Another important point is that we have been able to extract one behavior from nature, whose application in computer science did not exist or was unknown, and adapt it to solve a different problem but similar in its principle. And we have done this with a certain level of success.

In order to improve the game engine, we designed a simple MINIMAX algorithm without any pruning and we combined with the original proposed strategy. It was configured to make a 2-depth search, but we did not see any improvement in the quality of given moves and time consumption rose to 5 - 20 seconds. We configured the algorithm for a 3-depth search. Moves obtained could not be considered better and time consumption increased to achieve several minutes. Accordingly we believe that a good strategy for *Selfo* should not be based on the search depth, but in a quality and fast evaluation of the current state.

The current algorithm is relatively efficient in grouping particles and executing partial blockages to the enemy player. So improvements should focus on expanding its capacity to detect and avoid or execute total blockades. In this sense we propose algorithms based on *stigmery*, to detect a critical path between clusters of particles and calculate features like its width. It might also be useful to locate on the board certain blocking patterns previously investigated and defined.

An alternative strategy, away from the idea of swarm intelligence, although being also bio-inspired and fitting into the concept of emergent behaviors, may be based on the use of artificial neural networks, each neuron may correspond to a square, connections to adjacencies and the activation state to the presence of a friend or enemy particle or the absence of one piece.

References

- Browne, C. (2005). *Connections games: variations on a theme*. A K Peters, Ltd.
- Francisco J. Vico (2007). Selfo: A class of self-organizing connection games. Technical Report ITI 07-7.
- Ilachinski Andrew (2002). *Cellular automata: a discrete universe*. River Edge, NJ: World Scientific.
- Kennedy, J. and Eberhart, R.C. (2001). *Swarm Intelligence*. Morgan Kaufmann
- Sackson, S. (1969). *A Gamut of Games*. Random House, Inc.
- José Santos Reyes (2007). *Vida artificial: Realizaciones computacionales*. Universidad da Coruña, Servizo de Publicacións.
- Polczynski, J. (2001). Lightning: A connection game from the 1890s. *Abstract Games*, **5**: 8-9.
- Wolfram, S. (1983). Statistical mechanics of cellular automata. *Reviews of Modern Physics*, **55**(3): 601-644.
- Gardner, M. (1957). Concerning the game of Hex, which may be played on the tiles of the bathroom floor. *Scientific American*, **197**(1): 145-150.
- Dewdney, A. (1988) The hodge-podge machine makes waves. *Scientific American*, Computer Recreations. **Aug**: 104-107.
- de Bono, E. (1968) *The five day course in thinking*. Penguin Books.
- <http://www.zillions-of-games.com/>
- http://en.wikipedia.org/wiki/Dictyostelium_discoideum
- <http://en.wikipedia.org/wiki/Stigmergy>
- <http://en.wikipedia.org/wiki/Chemotaxis>
- <http://en.wikipedia.org/wiki/Emergence>
- <http://en.wikipedia.org/wiki/Emergent>
- <http://en.wikipedia.org/wiki/Self-organization>

Appendix: Implementation

Diagrams

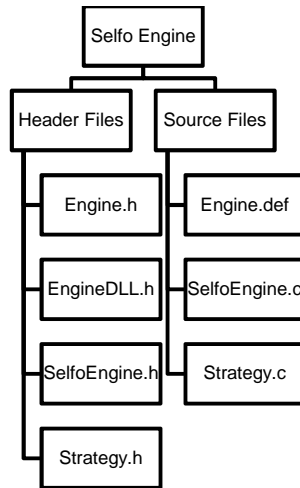


Figure A.1 Package structure.

SelfoEngine.c is used as an interface between Zillions and the search strategy implemented in *Strategy.c*.

DLL Interface

Structures

Definition of a square in the game board:

```
enum {  
    WHITE = 0,  
    BLACK = 1,  
    EMPTY = 2  
};
```

Values for search status:

```
typedef enum {  
    kKEEPSEARCHING = 0,  
    kSTOPSOON = 1,  
    kSTOPNOW = 2  
} Search_Status;
```

Relevant scores during the game:

```
enum {
    UNKNOWN_SCORE = -2140000000L,
    LOSS_SCORE = -2130000000L,
    DRAW_SCORE = 0,
    WIN_SCORE = 2130000000L
};
```

Global variables used in *SelfoEngine.c*:

```
static int iBoard[SIZE*SIZE];
static char *(gstrSideNames[2]) = {"White", "Black"};
static int gColor;
static int gBoard[NUM_POSITIONS];
static int auxBoard[NUM_POSITIONS];
static int inspectionBoard[NUM_POSITIONS];
```

Methods

DLL_Search is called by *Zillions* and it makes the call to the search strategy implemented separately:

```
DLL_Result FAR PASCAL DLL_Search(long lSearchTime, long lDepthLimit, long lVariety,
    Search_Status *pSearchStatus, LPSTR bestMove, LPSTR currentMove,
    long *plNodes, long *plScore, long *plDepth) {

    return SearchStrategy(lSearchTime, lDepthLimit, lVariety, pSearchStatus, bestMove, currentMove,
        plNodes, plScore, plDepth, gBoard, gColor);
}
```

These functions allow translation between *Zillions* representation of a square, in form of a string (e.g. "j9") and internal representation:

```
int StringToPosition(char *s);
void PositionToString(int position, char *s);
```

These functions make the string which represents a move in *Zillions*:

```
void DropToString(int position, int color, char *s);
void MoveToString(int position1, int position2, char *s);
```

GetNeighbor gives the *n*th neighbor to position *pos* in the game board. It has to deal with the topology of the game board:

```
int GetNeighbor(int pos, int n);
```

This function returns *WIN_SCORE* or *LOSS_SCORE* if the current player or his opponent has won the game, and *UNKNOWN* score in other case:

```
int AreAllConnected(int color, int *board);
```

Custom Strategy

Structures

groundBoard represents the relevant information of a given state in the game board. It contains information about groups and their signals throw to the environment:

```
static square groundBoard[NUM_POSITIONS][MAX_FRONTS+1];
```

Information contained in *groundBoard* for each group and cell:

```
typedef struct {  
    int front_id;  
    double front_power;  
    int color;  
} square;
```

Representation of a move inside the engine:

```
typedef struct {  
    int o;  
    int d;  
} child;
```

Auxiliary boards to explore a certain game state:

```
static int auxBoard[NUM_POSITIONS];  
static int inspectionBoard[NUM_POSITIONS];
```

Functions

SearchStrategy can call to an auxiliary search strategy, like MINIMAX. But it can also perform a simple search by itself:

```
DLL_Result SearchStrategy(long lSearchTime, long lDepthLimit, long lVariety, Search_Status *pSearchStatus,  
    LPSTR bestMove, LPSTR currentMove, long *plNodes, long *plScore, long *plDepth, int *gBoard, int gColor) {
```

When an evaluation of a state is required, *CheckEnvironment* is called to put into *groundBoard* the environment's information corresponding to the state:

```
int CheckEnvironment(int nPlayers, Search_Status *pSearchStatus);
```

When the environment has been built, *EvaluateEnvironment* is called to obtain the score which will tell how good the given state is:

```
void EvaluateEnvironment(int cColor, long *score, int nPlayers, int nGroups, Search_Status *pSearchStatus);
```

These functions help *CheckEnvironment* to locate connected groups and expand their signal of attraction along the environment (*groundBoard*):

```
int MarkGroups();  
int MarkGroup(int position, int id, int color);  
void ThrowSignal(int g, int nGroups, Search_Status *pSearchStatus);  
int ExpandSquare(int position, int idG, double power);
```

These functions help *EvaluateEnvironment* to locate the most attractive square in the environment (the centroid of attraction) and the score of the situation which will depend on the relative situation of particles to centroid:

```
int LocateMostPopulous(int color, int nGroups);  
long CalculateScore(int color, int centroid, int nGroups, int nPlayers);  
void CalculateDistanceToCentroid(int color, int centroid);
```

Often is necessary to inform *Zillions* about current search status:

```
void Report();
```