Universitat Autònoma de Barcelona

MASTER IN COMPUTER VISION AND ARTIFICIAL INTELLIGENCE
REPORT OF THE RESEARCH PROJECT
ARTIFICIAL INTELLIGENCE

Optimizing a Diplomacy Bot Using Genetic Algorithms

Author: Dave de Jonge
Date: 26 August 2010
Advisor: Carles Sierra

Optimizing a Diplomacy Bot Using Genetic Algorithms

Dave de Jonge dave_de_jonge@hotmail.com
IIIA-CSIC
Campus de la UAB
08193 Bellaterra, Catalonia (Spain)

Editor: Carles Sierra

Abstract

We give a short explanation of the game Diplomacy and an overview of the currently existing projects regarding artificial intelligence applied to Diplomacy. Then we make an analysis of the DumbBot, which is one of the simplest existing bots that can play Diplomacy at a reasonable level; we analyze its strategy and its parameters. Our goal is to optimize the parameters of the DumbBot, and therefore we repeat the basics of optimizing functions of many variables and show when and why one can use a genetic algorithm to do this. Also we propose a new alternative variant of the genetic algorithm that might lead to better results in our specific optimization problem. This alternative increases the exploration of lower order schemas, at the expense of a decrease of the exploitation of higher order schemas. We argue that this decrease is not a loss, because it is impossible for us to explore higher order schemas sufficiently in the first place. Finally we optimize the parameters using three different methods: hill climbing, a standard genetic algorithm and our alternative genetic algorithm.

1. Introduction

Diplomacy is a strategy game that distinguishes itself from other strategy games in that players continuously need to negotiate with the other players in order to be successful. To play well, a player needs to be an expert not only in strategy, but also in negotiation and argumentation and should be able to estimate whether its negotiation partners can be trusted or not. These typical human skills make it especially challenging for research in artificial intelligence. Moreover, Diplomacy is a game without random moves in which all players take their turn simultaneously, which make it an ideal environment for researchers.

Another advantage is that it has been played for decades, by many people and therefore there is a lot of data and expertise available. Many of these players meet in online communities where they share their knowledge and there is even an online community for Diplomacy bot developers (DAIDE) where many examples of Diplomacy bots are already available. This all makes it very easy to compare your results with other bots and human players. Therefore, the IIIA-CSIC in Barcelona has developed a framework called DipGame for the development and testing of Diplomacy bots, which serves as testbed for multiagent systems.

In order for users of this framework to have something to compare their results with, we need an example of a very simple bot that is able to play the game at a reasonable level. One such a bot already exists from the DAIDE community, which is called the DumbBot.

In order to make it compatible with DipGame we use a copy of the DumbBot that is based upon the DipGame framework. The parameters of this bot are however chosen purely by intuition; no attempts have been made by its creator to find the best values[1]. So our goal is to improve this bot by optimizing them, using genetic algorithms.

Genetic algorithms form a technique that is inspired directly from nature. They imitate the natural selection process that has successfully been applied by nature for millions of years with highly optimized creatures as a result. Genetic algorithms imitate this process by considering arrays of parameters as chromosomes. Many versions (individuals) of the same program are initialized with different values for their parameters and then the performance of each of these programs is evaluated. This performance (or 'fitness') is used to apply a form of natural selection. Also, chromosomes from different individuals are cut into pieces and pasted together, in order to create new individuals that have so inherited their features from several 'parent' programs. Comparable to the way in which in nature genetic information is passed on and recombined from generation to generation.

2. The Main Rules of Diplomacy

Diplomacy is a board game that appeared during the 50's. It is a game without random choices and in which all players take their turn simultaneously. It distinguishes itself from other strategy games in the fact that it is essential to negotiate with your opponents, making it a very "human" game. The game can be played in many variants, but throughout this thesis we will only consider the standard variant.

2.1 The Map and the Units

We will only explain the most basic rules here. For a complete documentation of the rules we refer the reader to the Diplomacy Rulebook (Ava, 2000). Diplomacy is situated in Europe at the beginning of the 20th century. The play board is a map of Europe, which is divided into seventy-five provinces. The game is played by seven players, that each represent one of the seven great powers of that time: England, France, Turkey, Russia, Italy, Germany, and Austria. Each player begins with three units (except Russia, who begins with four units) that can be moved through the map. A unit always occupies one of the provinces, and each province can be occupied by maximally one unit. During a turn, a player can try to move a unit from one province to another if both provinces are adjacent to each other on the map. For example: a unit in Holland could only try to move to Kiel, Ruhr, Belgium, North Sea, or Helgoland Bight, because these are the provinces adjacent to Holland (see Figure 2).

Thirty-four of the provinces contain a so called *supply center*. A supply center is said to be *owned* by a certain power if the last unit that occupied it after a fall turn (explained below) was a unit of this power. The goal of the game is to own at least eighteen of these supply centers. The location of each supply center and their initial distribution among the powers is fixed. The supply centers that a power occupies at the beginning of the game are referred to as its *home supply centers*.

1. The source code of the DumbBot contains the following comments regarding the parameters: *"These are just values that seemed right at the time - there has been no attempt to optimise them for best play."* The source code can be downloaded from David Norman's Diplomacy AI page.

Figure 1: The Diplomacy map. The colored circles are the units at their initial positions.

Figure 2: Holland has five neighbors.

The units are divided in two types: armies and fleets, while the provinces are divided in three types: inland provinces, sea provinces, and coastal provinces. An inland province can only be occupied by an army and a sea province can only be occupied by a fleet. A coastal province can be occupied by either type of unit. A coastal province has one or more coasts. When a fleet is located in a coastal province, it is explicitly located at one of its coasts. This is because a fleet can only move from one coast to another if these two coasts are adjacent, rather than if the two provinces they belong to are adjacent. In Figure 1 we see for instance that Spain has two coasts: the north coast, bordering to the Mid-Atlantic Ocean, and the south-east coast, bordering to the Western Mediterranean and the Gulf of Lyon. A fleet at the south-east coast can move for instance to Marseille, but not to Gascony, while a fleet at the north coast of Spain could move to Gascony, but not to Marseille.

To simplify the language, instead of talking a about coasts we will talk about *regions*. A region refers to either an inland province, a sea province, a coast of a coastal province, or the inner territory of a coastal province. Therefore each unit is located at a certain region, rather than at a certain province or coast.

Figure 3: The home supply centers of England.

The game begins in the spring of the year 1901. Every year is divided into five phases: *spring*, *summer*, *fall*, *autumn* and *winter*. The players can move their units during spring and fall and they can retreat their units during summer and autumn (see below). During the winter phase powers can build new units (see below).

2.2 Movement of Units

During spring and fall (the movement phases), each player secretly writes down movement orders for his units and then all orders are revealed simultaneously. For example, England might want to move his fleet from London to Wales and his fleet in Edinburgh to the North Sea, while he wants his army in Liverpool to stay where it is (see Fig. 3). If a unit is ordered to stay where it is, it is called a *hold order*.

An order might or might not succeed, depending on the moves that the opponents have ordered for their own units. If a unit is ordered to move into an empty province and no other unit is ordered to go into the same province, then the move will succeed. If however two units are ordered to move into the same province, the strongest unit will succeed and will indeed move to that province, while the weakest unit will stay where it was. Although each unit is in principle equally strong, a unit's strength is increased if it is supported by another unit, which we will explain below. If two equally strong units are trying to move into the same province they both fail.

2.3 Support

Instead of ordering a unit to move from one region to another, a player can also order his unit to support another unit. This means that the supporting unit will stay in the region where it is, while the supported unit will become stronger. The strength of a unit is simply given by the number of units that support it. A unit a can only give support to a unit b if b is ordered to move into a region that is adjacent to the region where a is located.

For example: England can order his army in Liverpool to move to Wales and can order his fleet in London to give support to this army. However, if England orders his army in Liverpool to Clyde, this army cannot be supported by the fleet in London, because Clyde is not adjacent to London (see Fig. 3).

A unit can be supported by an unlimited amount of other units, but a unit can only give support to one other unit. The supported unit and the supporting unit do not have to belong to the same power.

2.4 Retreat

As said before, when a unit does not succeed in moving to another region, it will stay in its original region. However, it might happen that during the same turn this region is invaded by another unit. In that case, the unit that failed can not stay in its original location and it has to retreat to another one. This happens during the summer or autumn phase following the movement phase. During these phases each unit that has to be retreated is ordered by its owner to move to some available region, adjacent to its original location. If there is no region available to go, the unit will be *disbanded*, which means that it is removed from the game.

2.5 Build

In the last phase of the year, winter, the players get the opportunity to build new units. The number of units a power can own after the winter phase is maximally the number of supply centers it owns, so a power can only build new units if it currently owns more supply centers than units. A power can only build a new unit in one of its home supply centers and only if he currently owns this supply center, but does not occupy it (because a province can only be occupied by one unit at the same time).

For example: suppose England currently has only five units, but owns seven supply centers. This means it can build two new units. England's home supply centers are London, Edinburgh and Liverpool, so each new unit can be build only in one of these provinces. If for instance London is occupied however, it can only build one unit in Edinburgh and one unit in Liverpool (assuming that these supply centers are still owned by England).

If a power owns less supply centers than units, during the winter phase it has to disband a number of its units, such that it again owns as many supply centers as units. If a unit is built in a coastal province then the power can choose whether the new unit will be an army or a fleet. When a power builds a new fleet in a province with more than one coast it has to specify at which coast the new fleet will be located.

3. Existing Projects

Diplomacy has been used before by many other people to do experiments in the field of AI Here follows a short overview of the existing projects.

3.1 DAIDE

The main community on internet that focusses on writing AI bots that play Diplomacy is the *Diplomacy AI Development Environment* (DAIDE). They have developed a game server and several bots that are capable of playing the game, with various skills of negotiation. You can let these bots play against each other by starting the game server and letting the bots connect to this server via a TCP/IP connection. Furthermore they provide several development kits to start building your own bot. Also they have developed a language

for communication between the bots. This language is built up of 15 layers, of which the complexity increases with each level. Level 0, is the *no press* level, in which there is no communication between the players, while level 14 is the free text level, in which the players communicate with each other in natural language. Most currently existing bots are able to communicate in level 0 language exclusively and only a few bots are capable of communicating at level 1.

3.2 The DumbBot

One of the bots provided by the DAIDE community is the DumbBot by David Norman. This is a very simple bot that applies a basic strategy, but nevertheless performs quite well against human opponents. It is not able to communicate with other players, so it is a level 0 bot. One could consider the DumbBot as the simplest non-trivial bot.

3.3 DipGame

DipGame is an open source project from the Artificial Intelligence Research Institute of the Spanish Scientific Research Council (IIIA-CSIC) in Barcelona. It is meant as a scientific testbed for multiagent systems and has its own infrastructure, but is compatible with DAIDE. A big difference is that DipGame intends to make everything available for all platforms, while the tools of DAIDE are almost exclusively developed for windows.

Currently the main component of the DipGame project is a framework, written in java, that makes it easy to develop new bots. Also some tools are under development to test your bots and to analyze their progress and results.

4. Analysis of the DumbBot

In this section we explain how the DumbBot of David Norman works. This analysis is based only on the source code that David Norman has published on his website (Norman). We will explain the strategy that the DumbBot uses and analyze its parameters. Finally, we will give some suggestions on how it could be improved.

4.1 Strategy of the DumbBot

The strategy that the DumbBot uses is based on the following principle: *It is better to steal supply centers from strong enemies than from weak enemies.* The reason for this is that the more supply centers a power owns, the easier it is for it to gain even more of them. Therefore it is vital to stop strong powers as soon as possible. So, whenever the DumbBot has to choose which supply center to attack, it favors the one which is owned by the strongest power, analogously, whenever the DumbBot has to choose which supply center it has to defend, it favors the one which might be attacked by the strongest power. In the ideal case the supply centers that are not owned by us are homogeneously divided among our opponents.

4.1.1 Power Size

The DumbBot needs a way to measure the strength of a power. This measure is called the *power size*. The power size S_p of a power p is determined by the number of supply centers it owns, which we shall denote by x_p.

$$x_p = \text{number of supply centers owned by power } p$$

$$x_p \in [0, 17]$$

According to the basic strategy defined above, we want to express that when a power captures a new supply center, it gains more strength if it is already very strong then when it is weak. Therefore, the power size is defined as a quadratic function of the number of supply center the power owns:

$$S_p \quad = \quad A \cdot x_p^2 + B \cdot x_p + C \tag{1}$$

where A, B, and C are real-valued constants that have the following values in the source code: $A = 1$, $B = 4$, $C = 16$. The fact that power size should be quadratic in the number of units can be seen in the following way: we can consider the negative of the sum of the power sizes of our six opponents as a form of utility (the stronger our opponents are, the lower our utility).

$$u \quad = \quad -\sum_{p=1}^{6} S_p \quad \simeq \quad -\sum_{p=1}^{6} x_p^2$$

If we then steal a supply center from power number 1, the increase δu of our utility would be:

$$\delta u \quad \simeq \quad -((x_1 - 1)^2 - x_1^2) \quad = \quad 2x_1 - 1 \ .$$

So we see that our gain of utility is increasing with the size of the power from which we steal the supply center. In other words: our utility increases more if we steal from a strong power than if we steal from a weak power.

It is easy to see that if power size would be linear in the number of supply centers, it wouldn't make any difference whether this enemy were a strong or a weak power, so our preference for equally strong opponents wouldn't be reflected in the utility.

4.1.2 Attack- and Defense- Value

At the beginning of each turn the DumbBot has to choose which regions to attack. Therefore, we need to assign a value to each region that determines how much a region is worth to attack. This value is called the *attack value* and it is defined such that the province which is the most valuable to attack has the highest attack value. As stated above, a supply center is more valuable if it is owned by a stronger power. Also a province that contains a supply center is more valuable than a province that does not contain a supply center. Therefore we define the attack value av_r of a region r as follows:

$$av_r = \begin{cases} 0, & \text{if region } r \text{ does not have a supply center.} \\ 0, & \text{if region } r \text{ has a supply center, but we already occupy it.} \\ S_p, & \text{if region } r \text{ has a supply center and is occupied by opponent } p. \end{cases}$$

At the same time, we also need to determine how much the regions that we occupy are worth to defend. According to the same principle as for attacking, we prefer to defend the regions that could be attacked by strong powers. So to evaluate how much a region r is worth to defend we define its *defense value* as follows:

$$dv_r = \begin{cases} 0, & \text{if region } r \text{ does not contain a supply center.} \\ max_{p \in att\{r\}}\ S_p, & \text{if region } r \text{ contains a supply center and is occupied by us.} \\ 0, & \text{if we do not occupy region } r. \end{cases}$$

where $att\{r\}$ is defined as the set of powers that are able to attack region r. In other words: the defense value of a region that contains a supply center and is occupied by us, is equal to the power size of the strongest power that might attack it.

4.1.3 Proximity Map

These definitions assume that the regions that do not contain a supply center are worthless. Of course, they indeed do not directly contribute to a victory, since only the number of supply centers owned determines who wins. However, they do have value in an indirect manner, since conquering such a region might give you a strategic advantage. That is: a region might border to other regions that do contain supply centers. Therefore, in order to determine how valuable a region is, we must not only take into account the attack- and defense value of the region itself, but also the attack- and defense values of its neighbors. Moreover, a region might also be valuable because its neighbors have neighbors that have supply centers, etc.

For this reason the value of a region is a weighted sum of its own attack- and defense values and those of all nearby regions. The contribution from its own attack value and defense value is called the *zeroth order proximity map*, pm_r^0 and is defined as follows:

$$pm_r^0 \quad = \quad aw \cdot av_r + dw \cdot dv_r$$

Here, aw and dw are constants that we respectively call the *attack weight* and the *defense weight*. Notice that av_r and dv_r are never both non-zero at the same time, so we can equivalently write:

$$pm_r^0 = \begin{cases} 0, & r \text{ does not contain a s.c.} \\ aw \cdot av_r, & r \text{ contains a s.c. and is not occupied by us.} \\ dw \cdot dv_r, & r \text{ contains a s.c. and is occupied by us.} \end{cases}$$

or:

$$pm_r^0 = \begin{cases} 0, & r \text{ does not contain a s.c.} \\ aw \cdot S_p, & r \text{ contains a s.c. and is occupied by opponent } p. \\ dw \cdot max_{p \in att\{r\}}\ S_p, & r \text{ contains a s.c. and is occupied by us.} \end{cases} \quad (2)$$

The contribution to the value of a region from the attack- and defense values of the region's neighbors is called the *first order proximity map*, pm_r^1. It is defined as the average of the

zeroth order proximity maps of the region itself and its neighbors:

$$pm_r^1 \quad = \quad \frac{1}{5} \cdot (pm_r^0 \; + \sum_{q \in adj\{r\}} pm_q^0)$$

where $adj\{r\}$ is the set of all neighbors of region r. And in general, the *i-th order proximity map* pm_r^i is defined as:

$$pm_r^i \quad = \quad \frac{1}{5} \cdot (pm_r^{i-1} \; + \sum_{q \in adj\{r\}} pm_q^{i-1}) \; .$$

The factor $\frac{1}{5}$ in these formulas is included to take the average value over all neighbors of r and r itself, so in fact it should actually be $1/(|adj\{r\}|+1)$, but to simplify the calculations David Norman chose to simply always divide by 5, because the average number of neighbors a region has is approximately 4.

4.1.4 Competition- and Strength Value

Furthermore, we should also take into account how difficult it is to defend or attack a region. For example: region a might be more valuable than region b, but if we have a much bigger chance to succeed in capturing region b than to succeed in capturing region a, it could be better to try to attack region b. Therefore, we define the *strength value* and the *competition value*. The strength value sv_r is the number of units that we own in regions adjacent to region r:

$$sv_r \quad = \quad |occ \cap adj\{r\}| \quad \in [0, 7]$$

where occ is the set of all regions that are currently occupied by us. So the strength value is the maximum amount of units that we could use to attack region r.

The competition value cv_r is the maximum number of units that any opponent has adjacent to region r:

$$cv_r \quad = \quad max_p \; \{ \; |occ_p \cap adj\{r\}| \; \} \quad \in [0, 7]$$

where occ_p is the set of regions currently occupied by opponent p In other words: the competition value is the maximum number of units that any opponent could use to attack region r.

4.1.5 The Destination Value

Now, to determine which regions we should attack and which we should defend, we need a formula based on the proximity map values and the competition- and strength- values. We take a weighted sum of all these values and this is then called the *destination value* of region r:

$$DV_r \quad = \quad \sum_{i=0}^{n} pw^i \cdot pm_r^i + sw \cdot sv_r - cw \cdot cv_r \tag{3}$$

with pw^i, sw and cw the weights that determine how important each variable is. They have separate values for spring turns and for fall turns. The value n (let's call it the *proximity*

depth) determines how far ahead we look. If this value is set too high the calculations take too much time while the contribution of high order proximity values is relatively small. If it is too small however, we would neglect the strategic value of a region r due its proximity to other regions that contain a supply center. In the source code of the DumbBot this value is set to $n = 9$ without any justification.

To get a bit of an idea of what range of values the destination values can take on and of the way in which they are influenced by the values of the weights we can make an approximation of equation (3). If we define x as the average number of units any power has, and y as the average number of units that any power has adjacent to region r. Then we have:

$$\begin{aligned} S_p &\simeq x^2 \\ pm_r^0 &\simeq \frac{1}{2}(aw + dw)x^2 \\ pm_r^i &\simeq pm_r^0 \end{aligned}$$

such that Equation (3) is approximated as:

$$DV_r \simeq 5 \cdot pw^i \cdot (aw + dw) \cdot x^2 + (sw - cw) \cdot y$$

with:

$$x \in [0, 17], \quad y \in [0, 7]$$

4.2 Determining Moves

After calculating the destination value for each region, every unit tries to move into its best adjacent region, but with a random chance of moving into the second best, third best, and so on. It is possible that the chosen destination is the region in which the unit is already located, in which case the unit holds. If the chosen destination is already occupied by one of our own units or if another of our units is moving there, it either supports that unit or moves elsewhere, depending on wether the other unit is guaranteed to succeed or not.

4.2.1 Ordering the Next Move

The procedure that chooses which move a unit is ordered to make works as follows. At first, we make a list of all adjacent regions that our unit could move into, including the region where it currently is. We select the region with the highest destination value from this list as our destination. However, with a certain probability we will replace this choice by the next best region. The chance of picking the next best destination is calculated with:

$$next\_province\_chance = \frac{(DV_a - DV_b)}{DV_a} \cdot 500 \tag{4}$$

Where DV_a is the value assigned to the first destination (the destination with highest value) and DV_b the value assigned to the next best destination.

A random number between 0 and 100 is generated. If this number is higher than the threshold *play_alternative* (which is set to 50 by default) or lower than *next_province_chance*,

the currently selected destination will be the definitive destination for the current unit. Otherwise, we will pick the second best destination, but with a certain chance of replacing it by the third best destination. The probability of this is again calculated with the same formula (4), only this time with DV_a and DV_b referring to the second best and third best destination values respectively. This process is repeated until we have a definitive destination.

4.2.2 Build, Remove, Retreat and Support

For the removal of units (if necessary) the bot follows the same recipe as for moving units: for every unit the value of its location is calculated (the *winter destination value*) and then one or more units are randomly chosen to be removed, but with the probability of being chosen decreasing with the value of the location (the higher the value, the lower the chance of being removed). The *winter destination value* for a region i is given by:

$$WDV_i = \sum_{i=0}^{n} pw^i \cdot pm_r^i + dw \cdot dv_r$$

with special values for pw and dw that are specific for the removal phase ("winter values"). The weights that are involved in calculating pm_r^i however, have their "spring value".

Choosing a location to build a unit, or a location to retreat to, works exactly the same, only with the chance of a location being chosen getting higher, as the destination value gets higher.

If the algorithm described in Section 4.2.1 determines that one of our units should hold, the bot checks whether this unit can give support to one of the other units. To do this, it first checks to which units it could give support and whether these units need it. Then, from all units that can get support and need support it chooses the one that tries to move into the region with the highest destination value.

4.3 Analysis of the Values of the Weights

We will first give the values that are given to the weights in the source code, and then we will discuss them. The default values of the weights are:

weight	**spring**	**fall**	**winter**
strength weight (sw)	1000	1000	-
competition weight (cw)	1000	1000	-
attack weight (aw)	700	600	-
defense weight (dw)	300	400	1000

For the fall and winter values of the proximity weights we have:

pw^0	=	1000	pw^5	=	5
pw^1	=	100	pw^6	=	4
pw^2	=	30	pw^7	=	3
pw^3	=	10	pw^8	=	2
pw^4	=	6	pw^9	=	1

The spring values of the proximity weights are exactly the same, except that the values of the first two weights are interchanged: $pw^0 = 100$ and $pw^1 = 1000$. Finally, the values of the constants A, B and C that define power size in (1) have the following values:

$$A = 1 \quad B = 4 \quad C = 16$$

4.3.1 Competition- and Strength Value vs. Proximity Map

One interesting observation is that the competition weight, the strength weight, and the highest proximity weight all have the same value, namely 1000. This means that in equation (3) the two terms concerning strength- and competition value are relatively small compared to the proximity map, because, if x denotes the average number of supply centers owned by a power, the proximity map variables are proportional to power size, which is proportional to x^2, while the strength and competition values are proportional to x. Moreover, because of the minus sign in (3) the competition- and strength values tend to cancel each other out, making their contribution even smaller. This is striking, because it means that the contributions of the strength- and competition value are negligible.

Let's illustrate this with the following example. Suppose we want to attack a certain supply center. We assume that we have only one unit next to it, while an enemy has 4 units next to it. We have:

$$pw^0 \cdot pm_r^0 \quad = \quad pw^0 \cdot aw \cdot av_r \quad = \quad 1000 \cdot 600 \cdot S_p \quad \simeq \quad 600,000 \cdot x_p^2$$

while

$$sw \cdot sv_r - cw \cdot cv_r \quad = \quad 1000 \cdot 1 - 1000 \cdot 4 \quad = \quad -3000$$

We see here that the last two terms of (3) are very small indeed compared to the proximity map. Now if the province we'd like to attack is unoccupied we'd have that pm_r^0 would be zero, but even in that case the higher order values of the proximity map would be very big compared to the strength and competition values.

4.3.2 Spring Proximity Weights

The fact that the first two proximity weights for spring are interchanged with respect to their fall and winter counterparts might seem strange at first but it does make some sense. This is probably because you can only build new units if you own new supply centers after a fall turn. So during the fall turn it is important to obtain new supply centers, while during the spring turn it is more important to have a strategic position so that you have a better chance of obtaining new supply centers during the following fall turn. The value of a region during spring is therefore more determined by the supply centers it borders to, than by its own supply center.

4.3.3 Attack- and Defence Weights

Another interesting observation is that the attack weights are higher than the defense weights, and that the difference between them is smaller during fall than during spring. This means that apparently the bot considers attacking new supply centers more important than defending currently owned supply centers. One reason might be that, by just defending

your own supply centers you might retain them, but you will never increase your number of supply centers. However, by attacking a province you might simultaneously achieve the goals of gaining a new supply center and defending the one you already posses, because if you succeed, the attacked province will not be able to attack you. To say it simpler: attack is also a form of defense, while defense is never a form of attack.

The fact that the difference between attack weight and defense weight is smaller during fall, makes sense. After all, losing a supply center during spring is not so bad, because you might be able to re-capture it during fall without any consequences. While if you loose a supply center during fall, you will be able to build one unit less.

4.4 Discussion

In this subsection we will discuss our findings with regard to the DumbBot and make some suggestions on how it might be improved.

4.4.1 Power Size Reviewed

In Section 4.1.1 we argued that power size should be a quadratic function of the number of supply centers owned. However, there is a subtle point that we would like to point out about this. One should notice that, if we would neglect the competition- and strength- values and the higher order values of the proximity map, it would not matter whether the power size were quadratic or linear. Since in that case, the region with the highest destination value would always be the one which were occupied by the power with the highest number of supply centers, regardless of the question whether the power size were quadratic or linear in the number of supply centers. Mathematically: if S_p is any monotonically increasing function of x_p, then we have:

$$arg\ max_p\{S_p(x_p)\} \quad = \quad arg\ max_p\{x_p\}$$

(notice that we take the *arg max* with respect to p, rather than x_p). Since we do not use the *actual* value of the destination value, but only the *arg max* of the destination value, it would not matter whether the power size were quadratic or linear.

The fact that the power size is quadratic does play a role however, when calculating the higher order proximity map values. Remember that if we want to capture a supply center, then pm_r^0 is proportional to the attack value of the province, which is the power size of the power that owns it (see Equation (2)), so $pm_r^0 \propto S_p$ for some power p. Then for pm_r^1 we have:

$$pm_r^1 \quad \propto \quad S_p + \sum_{j \in adj\{r\}} S_j$$

With S_j standing for the power size of the power that owns region j. It is because of the summation in this equation that non-linearity of the power size matters, because a linear power size would give different results than a quadratic power size, even after taking the *arg max*. In general we have:

$$arg\ max_r\ \{x_r^2 + \sum_{j \in adj\{r\}} x_j^2\} \quad \neq \quad arg\ max_r\ \{x_r + \sum_{j \in adj\{r\}} x_j\}$$

with x_r standing for the number of units belonging to the power that occupies region r.

4.4.2 Suggestions for Improvement

The values of the weights that the DumbBot uses have been chosen merely by intuition. David Norman has not tried to optimize them in any way. We have seen for instance that the values of the strength weight and the competition weight seem to be much smaller than one would expect them to be. Therefore the DumbBot might be able to perform better if we could optimize the parameters. This is exactly what the rest of this thesis is devoted to.

The DumbBot was just intended as a very simple bot that applies some kind of basic strategy. Therefore, there are a lot of things that the DumbBot does not take into account, which a seriously competitive bot would need to.

For instance, the bot does not make any predictions of what its opponents will do. Its strategy is purely based on what move is the best assuming that each move would succeed and that the opponents' positions remain the same. Also, the bot does not have any long term strategic plan. Every turn is treated separately and therefore the bot ignores any future consequences of a move.

Furthermore, except from the concept of "being near to a supply center" the DumbBot ignores the fact that some provinces have strategic locations. A province can for instance be valuable because by occupying it, you might block the way for your opponents to move to other provinces.

Finally, one important ability that the DumbBot lacks is the ability to coordinate the moves between its units. Every unit is treated individually. For this reason the DumbBot is unable to perform one very important move, which is the *self standoff*. This is one of the basic moves that every novel player learns. The self standoff is a move in which two units of the same power try to move into the same province with the goal of defending three provinces at the same time with only two units.

5. Searching With Genetic Algorithms

The goal of our work is to optimize the parameters of the DumbBot, which we will try to do using a genetic algorithm. Therefore, in this section we first describe how a general genetic algorithm (GA) works. Then, to compare it with simpler algorithms we will look at technique that we will call naïve optimization. We will then derive a mathematical foundation for the GA. Also we argue when and why a genetic algorithm is a good search algorithm. Finally, in the last subsection we explain why the advantages of the GA mentioned before are not very useful for our purposes, and therefore we propose a new variant of the genetic algorithm that might be better suited to our needs.

5.1 The Standard Genetic Algorithm

The goal of a genetic algorithm is to search for the optimum of a function in a high-dimensional space. A point in such a space is called an *instance* or an *individual*. The function that we want to optimize is called the *fitness function*. We suppose that we have an encoding scheme with which we can encode each point in the search space as a row vector. Such a vector is called a *chromosome*. Each entry of the vector is called a *gene* and a specific value at a specific entry is called an *allele*.

So for example we have an instance x_0 of our search space that can be encoded as a vector in a 4-dimensional space:

$$x_0 = (1, 2, 3, 4)$$

The row vector $(1, 2, 3, 4)$ is the chromosome, and since it is a 4-dimensional vector, it consists of four genes. If f is the fitness function, than $f(x_0)$ is the fitness of x_0 and the goal of the algorithm is to find $arg\ max_x\{f(x)\}$, or at least an instance x_1 for which $f(x_1)$ is close to the global maximum of f.

The idea is that after having evaluated a certain set of individuals (*the initial population*), we pick the ones with the highest fitness and let them interchange their genes, so that hopefully we combine the best features of one individual with the best features of another individual. The newly created individuals will then partly replace the initial population. The process of exchanging genes is called *cross-over*. The individuals that will be crossed over are selected randomly, but the probability for an individual to be selected is increasing with its fitness (usually it is proportional to the fitness, but not necessarily. For example, it might be necessary to re-scale the fitness function to make sense as a probability distribution).

Crossing over works as follows: we have two individuals (the *parents*) and we cut both their chromosomes into two pieces (both are cut in exactly the same place, for example: between the second and the third gene). The place where we cut the chromosomes is chosen randomly. Then the first part of the first chromosome is pasted together with the second part of the second chromosome and vice versa. This way we have created two new individuals that have inherited genes from both their parents. For example, suppose the parents are

$$(1, 2, 3, 4) \quad \text{and} \quad (5, 6, 7, 8)$$

and we cut both chromosomes exactly in the middle, then we obtain the following two children:

$$(1, 2, 7, 8) \quad \text{and} \quad (3, 4, 7, 8)$$

(This is called single point cross-over, because we cut the chromosomes at one point. Alternative variants of GA might use multi point cross-over.)

After crossing over, a few of the new individuals are randomly selected for *mutation*. That is: for each selected individual one of its genes is selected randomly and its value (its allele) is replaced by some random other value. The selection of the individuals, the selection of the gene to mutate and the selection of its new value are all made with a uniform probability distribution. By applying mutation we make sure that alleles that have disappeared from the population have a chance to come back, or that alleles that weren't present in the original population still might appear into it.

If we have already found some good solutions in the initial generation, we want to keep them. Therefore, of each generation the best few individuals will be copied into the next generation. Making copies of individuals into the next generation is called *reproduction*. In the special case that these individuals are the best of the current generation it is known as *elitism*.

So finally the initial population is replaced by the new individuals that were produced by cross-over and mutation, plus the best few individuals of the initial population. This

process is then repeated several times, such that the population will contain better and better individuals.

Summarized, a genetic algorithm works as follows:

1. We choose a random population of individuals.
2. We evaluate their performances and assign a 'fitness' to each instance, which describes how well it has performed.
3. We randomly choose a set of individuals from the population to be crossed-over. The chance for an individual to be selected for cross-over is increasing with its fitness.
4. We randomly choose some of the newly created individuals to be mutated.
5. The new individuals that were created by cross-over and mutation form a new generation, together with the best individuals of the previous generation.
6. We then we repeat steps 2-5 for a certain amount of time and finally we pick the individual with the highest fitness of the last generation and consider it as the solution of our optimization problem.

5.2 Naïve Optimization

Our goal is to show that the problem of finding an optimum can indeed be solved by applying a genetic algorithm, under certain assumptions. Therefore we will first look at a much simpler way of optimizing a function, which is, what we will call here, *naïve optimization*. The analysis that we make here will be relevant for the following sections.

By naïve optimization we mean: first find the best value for the first variable while holding all the other variables fixed, and then continue by optimizing with respect to the second variable, while holding the others fixed, etc. In some very simple cases, after having optimized all variables separately, one might have found the global optimum of the function. However, in general this is not true, since we have only optimized all variables *under the condition that the other variables had a certain value*. In reality the optimal value of one variable might be highly dependent on the values of other variables.

So when does naïve optimization work? This is easy to answer if we can describe the evaluation function f as a polynomial of several variables, for instance:

$$f(x, y) = x + y + x^2 + y^2$$

In this case we can write

$$f(x, y) = f_1(x) + f_2(y)$$

with

$$f_1(x) = x + x^2$$

$$f_2(y) = y + y^2 \quad .$$

We can separate the function f here into two functions: one that is purely dependent on x and one that is purely dependent on y, because f does not contain any mixed terms:

each term only contains either x or y but not both of them. This implies that if $x_0 = arg\ max\{f_1(x)\}$ and $y_0 = arg\ max\{f_2(y)\}$ then we also have $(x_0, y_0) = arg\ max\{f(x, y)\}$.

On the other hand, we can show that if f does contain mixed terms, the optimum of f with respect to x depends on the value of y. This can easily be seen by differentiating.

$$g(x, y) = x + xy + y$$

$$\frac{\partial g}{\partial x} = 1 + y$$

We see that the mixed term causes the fact that the derivative of g with respect to x depends on y. And since the optimum can be found by setting the derivative to zero, the maximum value of g with respect to variable x depends on the value of y. In general we can state that a function f depending on variables x and y can be optimized by naïve optimization if $\frac{\partial^2 f}{\partial x \partial y} = 0$. If this holds we say that x and y are *uncorrelated*.

Definition 1 *Two variables x and y are uncorrelated if $\frac{\partial^2 f}{\partial x \partial y} = 0$. If $|\frac{\partial^2 f}{\partial x \partial y}|$ is high we say that their correlation is high, and if it is low, we say their correlation is low.*

It is however important to realize that we might be able to choose our variables such that they have low correlation. Take for instance the function $f = xy$. According to our previous analysis we can not apply naïve optimization here. However, if we change our variables to α and β with $x = \alpha + \beta$ and $y = \alpha - \beta$ then we see: $f = \alpha^2 - \beta^2$ and therefore it is possible to optimize f by naïve optimization, as long as we do this with respect to the variables α and β instead of x and y.

Of course, in reality we don't know the function f, because if we would know it, it would be much easier to find its optimum by setting the derivative to zero, than to apply naïve optimization. However, this is just to show quantitatively how and when naïve optimization can be used.

The fact that naïve optimization works only when x and y are uncorrelated can be overcome, by repeating it many times. So after all parameters have been optimized the algorithm doesn't stop, but starts over again with the first parameter. This continues until a local optimum is reached or until a certain amount of time has passed. This is then what we call *hill climbing*. The lower the variables are correlated, the better this will work.

5.3 Searching Through High Dimensional Spaces

As we stated before, a genetic algorithm is an algorithm to find the optimal value (or a nearly optimal value) of some function f on a high dimensional vector space $V \cong \mathbb{R}^n$.

$$f : V \to \mathbb{R}$$

(At least, from a theoretical point of view. In many computer science problems the entries of the vectors can take on only discrete values with a limited range: $f : [0, m]^n \to \mathbb{R}$. This is also the case in our problem.)

Since the search space for most practical problems is huge, it is usually impossible to simply evaluate all points and then keep the one with the highest value. So what we could

try to do is to evaluate a lot of randomly chosen instances and stop after a certain amount of time. The chance that we would find the optimum in this way is very small, but we might be able to improve our chances if we could make some kind of educated guess about where in the space we have a bigger chance of finding high values.

The trick is that, instead of looking directly for the best *vectors*, we try to look for the best *subspaces* of V. This means we try to find subspaces of V on which the average value of the function is high. If we then have a few of these 'good' subspaces, we continue our search on the intersections of these subspaces, expecting that the vectors there are even better. In the following we will make this more clear.

We use the following notation to specify subspaces:

$$(1, \cdot, \cdot, \cdot)$$

for example, stands for the 3-dimensional subspace of all vectors that have the value 1 in their first entry, while for example

$$(\cdot, 4, 7, \cdot)$$

stands for the 2-dimensional subspace of all vectors that have the value 4 in their second entry and the value 7 in their third entry (technically, these subspaces are *affine linear subspaces*, that is: linear subspaces, but without the requirement that they should contain the null vector). These specifications are called *schemas*.

Definition 2 *A schema is a row vector for which some of the entries have a $\cdot$ instead of a value.*

Definition 3 *The entries of a schema that contain a $\cdot$ are called the undefined entries and the entries that contain a value are the defined entries.*

Definition 4 *The number of defined entries in a schema is called the order of the schema.*

So our first example $(1, \cdot, \cdot, \cdot)$ is a schema of order one, while $(\cdot, 4, 7, \cdot)$ is a schema of order 2. The dimension d of the subspace that is defined by a schema is equal to the number of undefined entries, which is equal to the dimension n of the entire space minus the order o of the schema.

$$d = n - o$$

(Notice that the notion of an allele coincides with the notion of a schema of order 1). Sometimes we will talk about an "o-schema" when we mean a schema of order o. Also we will sometimes use the term "schema" when we are actually refering to the subspace defined by this schema.

Definition 5 *Given a schema, the distance between the two defined entries that are located farthest away from each other is called the defining length.*

So for example the schema $(5, \cdot, \cdot, 8)$ has defining length 3, while the schema $(\cdot, 5, 8, \cdot)$ has defining length 1. A vector is said to *satisfy* a certain schema, if the values of the defined entries of this schema are equal to those of the vector. For example: the vector $(3, 4, 7, 9)$ satisfies the schema $(\cdot, 4, 7, \cdot)$. Notice that every n-dimensional vector satisfies 2^n different schemas.

In this way we can not describe all subspaces of V, but only a very limited subset of all its subspaces. For instance: the linear subspace that satisfies the relation $x + y = 0$, where x and y denote the first and second entry respectively, can not be described by a schema. However, if we would use the variables α and β of Section 5.2 this same subspace would be defined by $\alpha = 0$, which can be described by a schema. This means that when we choose the variables that we will use to encode the search space as a vector space, we implicitly choose the set of subspaces that can be described by a schema.

Now the key point is that we want to choose our subspaces such that we can make the following assumption:

Assumption 1 *Suppose we have a subspace $U \subset V$ that can be described by a schema, and a point $x \in U$ for which $f(x)$ is high (low). Then we assume that the values $f(x')$ of other points $x' \in U$ are on average high (low) too.*

(The question of when we can make this assumption will be addressed at the end of this subsection.) For example, if f is optimal for some vector x_0, say $x_0 = (2, 5, 6, 3)$, then we assume that $f(x)$ will be relatively high for any vector x with a value of 2 in its first entry. That is, the subspace

$$(2, \cdot, \cdot, \cdot)$$

will be a good subspace. And the same holds for the subspaces

$$(\cdot, 5, \cdot, \cdot), \quad (\cdot, \cdot, 6, \cdot) \quad \text{and} \quad (\cdot, \cdot, \cdot, 3)$$

Notice that the optimum x_0 is the intersection of these four subspaces. Also, this assumption implies that if we have a few good subspaces, then the chance is big that the points that lie on the intersection of these subspaces are also good. So what we have to do, is to find good subspaces and then continue our search by looking at the points where these subspaces intersect.

The advantage of looking for good subspaces instead of good vectors, is that the number of subspaces defined by a low order schema is a lot smaller than the number of vectors. Suppose for instance that our space is 4-dimensional and for every dimension we consider ten possible values. Then we have in total 10^4 different vectors, but only $4 \cdot 10$ different 1-schemas. In general the number of o-schemas of an n-dimensional space with m different possible values for each entry, is given by:

$$\binom{n}{o} \cdot m^o \quad \text{or, equivalently, by:} \quad \binom{n}{d} \cdot m^{n-d} \tag{5}$$

where $d = n - o$ is the dimension of the corresponding subspace. In particular, notice that there is exactly one 0-schema: $(\cdot, \cdot, \cdot, \cdot)$, which defines the entire space itself, and that there are m^n 4-schemas, which are simply the vectors.

But how do we find good subspaces? Notice that each vector can be considered as a sample of all the schemas it satisfies (as said before, each vector satisfies 2^n different schemas). So the fitness of a vector can also be considered as an indication of the fitness of all the schemas that it satisfies. The idea is that when we have evaluated a number of random vectors (the initial population), we have simultaneously evaluated all the schemas

that are satisfied by these vectors. We can then continue our search by sampling new vectors that lie on the intersections of the good schemas.

This is in fact exactly what cross-over does: every time we choose two individuals for cross-over and choose where to cut their chromosomes, we are in fact selecting some schemas to be copied into the new generation, some schemas to be deleted and some new schemas to be sampled. For example, in the example of Section 5.1 the schemas $(1, 2, \cdot, \cdot)$ and $(\cdot, \cdot, 3, 4)$ survive, while the schema $(\cdot, 2, 3, \cdot)$ is deleted and the schema $(\cdot, 2, 7, \cdot)$ is created. Since the chance of an individual to be selected for cross-over is increasing with its fitness, the good schemas have a high chance of survival, while bad schemas are more likely to disappear.

Like in any other search algorithm, there is a trade-off between exploration and exploitation. The fact that good schemas have a bigger chance of survival means that they are exploited, while the fact that new schemas are created by cross-over means we are also exploring new subspaces. Moreover, mutation adds some extra exploration to the algorithm.

Notice however, that there is no guarantee that the intersection of two good subspaces will really be a good subspace too. After all, as we have seen in Section 5.2 it is very well possible that some values only perform well in combination with other values. Therefore the subspaces $(3, \cdot, \cdot, \cdot)$ and $(\cdot, 5, \cdot, \cdot)$ might both perform poorly, while their intersection $(3, 5, \cdot, \cdot)$ might be very good. Therefore, it is not enough simply to find the best 1-schemas and combine them into the best possible combination (which would be equivalent to naïve optimization). Some combinations of bad 1-schemas might lead to good 2-schemas, so it is important not to immediately discard bad 1-schemas. Therefore we should not just explore and exploit the 1-schemas, but we should also make sure that there is enough exploration of 2-schemas. And the same holds of course for higher order schemas. In fact there is a trade-off: exploring and exploiting low-order schemas is faster, because there are a lot less of them. But exploring and exploiting high-order schemas is more accurate.

Notice that especially schemas of low defining length have a better chance of survival than schemas with a high defining length, because the smaller the defining length, the smaller the chance that the cross-over will cut the schema in two. It is therefore essential that we encode the points of our search space into rowvectors in such a way that closely correlated features are encoded as vector entries that lie next to each other on the chromosome (more about this at the end of this subsection). This is called the *building block theorem*, see for instance Falkenauer (1998).

On one hand, one could consider this a disadvantage, because we need to put prior knowledge about the problem into our encoding in order to make the algorithm successful. On the other hand, if we know how to do this, the GA gives us a way of including our prior knowledge into the search algorithm, making it more effective than a simple hill climbing or random search.

Now we still need to address the question of when we can actually make the assumption that we made above. The simplest example is again a function that can be written as the sum of two functions of different variables.

$$f(x, y) = f_1(x) + f_2(y)$$

It is obvious that, for any y, an increase of the value $f_1(x)$, while holding y fixed leads to an increase of the entire function $f(x, y)$ and therefore we can say that if we randomly pick a value y then the chance of finding a high value for $f(x, y)$ is bigger as $f_1(x)$ is higher.

The situation is quite similiar to naïve optimization. Because what we want is that the value of a vector that satisfies a certain 1-schema is representative for the whole schema, which is almost the same as saying that finding the optimum of one variable is independent of the values of the other variables.

The difference however, is that naïve optimization only explores 1-schemas, while the genetic algorithm explores many schemas of any order at the same time. Another difference is that naïve optimization evaluates only one sample for each 1-schema, while GA explores many different samples.

So just as with naïve optimization and hill climbing, we want to choose our variables such that they are correlated as least as possible. But, because the GA allows exploration of 2-schemas, we can still have highly correlated variables as long as these highly correlated variables are placed next to each other on the chromosomes, so that they can be exploited without being destroyed the cross-over mechanism.

So suppose we have a function of four variables:

$$f(v, w, x, y) = 6 \cdot vw + 80 \cdot xy + 3 \cdot vx + 4 \cdot wy$$

$$\frac{\partial^2 f}{\partial x \partial y} = 80$$

it is clear that in this function the variables x and y are highly correlated with each other, while other pairs of variables are less correlated, so it makes sense to model the points in our search space as vectors of the form: (v, w, x, y), with the variables x and y next to each other. Notice that the value of f in this example is mainly determined by the values of x and y, and therefore we have indeed that the fitnesses of all vectors that satisfy the schema $(\cdot, \cdot, x_0, y_0)$ are similar.

If we would model our problem using vectors of the form (x, v, w, y) on the other hand, the genetic algorithm would probably be a lot less effective, because when we discover that the schema $(x_0, \cdot, \cdot, y_0)$ is very good, the chances are big that any instance of this schema will be destroyed by cross-over.

Of course in reality we don't know the expression for f, but we might be able to guess which features of a problem are related to each other, from our knowledge of the problem. That is: closely related variables should be encoded as genes that lie close to each other on the chromosome.

What we can conclude from this is that a genetic algorithm might perform better than a random search, because we can put some prior knowledge into the search by choosing our representation vectors carefully. On the other hand, if we do not put this information into the problem in the right way, or if we simply don't have this kind of information, the algorithm is probably not going to work so well.

5.4 An Alternative Genetic Algorithm

The problem with the standard form of the genetic algorithm is that one needs a lot of samples in order to have an accurate estimation of the average fitness of high order schemas. Since in our project it takes a long time to accumulate samples we argue that it is useless to look at these high order schemas. We argue that in our case it might be better to

concentrate on the lower order schemas and combine them in a more clever way than the standard algorithm does.

We propose the following new algorithm that we have developed ourselves. It was designed for problems in which the evaluation of individuals is so slow that we don't have the time to explore high order schemas sufficiently. Therefore it spends more computing time on the exploration more low order schemas, while spending less computing time on the exploitation of high order schemas.

It consists of two phases. The first phase works as follows:

1. We choose a random population
2. We evaluate the fitness of each individual.
3. For each schema of order 1 we calculate its average fitness, that is: the average of the fitnesses of all individuals that satisfy this schema.
4. We create a 'bag of alleles', in which the order 1 schemas with high average fitness occur more often than the ones with low average fitness.
5. We create new individuals by randomly selecting 1-schemas from the bag and combining them. So each new individual might be created from alleles of n different individuals.
6. steps 2-5 are repeated until we have accumulated enough samples of order 1 schemas. We then continue the algorithm with phase 2.

Phase 2 works the same, only everything works with 2-schemas instead of 1-schemas. So at step 3 we calculate the average fitness of the 2-schemas and our 'bag of alleles' at step 4 consists of instances of 2-schemas. At step 5 we create new individuals by pasting 2-schemas together, and therefore each new individual might be created from genes of $n/2$ different individuals.

Our variant is different from the standard algorithm in two ways: first, cross-over does not involve cutting two chromosomes into two pieces, but instead we combine the alleles from many different individuals (during phase 1 we apply n-1-point cross-over and during phase 2 we apply $n/2$-1-point cross-over). The reason for this is that, as stated above, we think it does not make much sense to exploit high order schemas. Therefore, we want to explore as much low order schemas as possible, even if this means that we can exploit less high order schemas. Cutting a chromosome into many pieces means that higher order schemas have no chance of survival, but it also means that we create a lot more new low-order schemas. So we increase the exploration of lower-order schemas, with the consequence that we decrease the exploitation of higher order schemas.

The second difference is that the abundance of a schema in the next generation is calculated from the average fitness of *all* instances satisfying this schema in the current and all previous generations (notice that one schema can be satisfied by multiple individuals in one generation). While in the standard GA each occurrence of a schema has a certain chance of survival, based only on the fitness of the specific individual that satisfies it.

For example: suppose that there is a bad schema p that by coincidence appears many times in the current population (that is: there are many individuals that satisfy this

schema). Say there are 10 instances of p with each a fitness of 0.2 and two instances of a good schema q with a fitness of 0.5. This means that in the next generation we expect $10 \cdot 0.2 = 2$ individuals satisfying p, while we only expect $2 \cdot 0.5 = 1$ individual satisfying schema q. Although the bad performance of p indeed leads to a big decrease of its presence, still it will be more present than schema q even though q has a much higher fitness. It will take several generations before this effect fades away. Therefore, in our algorithm the fitness of a schema is much better reflected in its presence, than it is in the standard genetic algorithm. It takes a little bit more calculation time, but this is negligible compared to the time necessary to evaluate each individual of the population.

The statement that we have too little samples of high order schemas to seriously take them into account is supported with the following calculations. Suppose we have n parameters that we want to optimize and each one can take on m different values. We have 6 computers running in parallel and we need about 40 seconds to play a game (more about this later in Section 6), so we can play 9 games per minute and we can evaluate maximally 7 individuals in one game (because Diplomacy is played by 7 players). However, to evaluate an individual we need to play a lot of games, because there is a lot of randomness in the strategy of the DumbBot. So an individual needs to play about 30 games before we can accurately assign it a fitness. Each individual satisfies $\binom{n}{o}$ o-schemas, so we can evaluate $\frac{7 \cdot 9}{30} \cdot \binom{n}{o}$ o-schemas per minute. Suppose we want to have 10 samples of each possible o-schema (that is: for each schema we want to evaluate the fitness of 10 different individuals that satisfy it). The total number of o-schemas in our search space is given by Formula (5), so this means it takes

$$(\frac{7 \cdot 9}{30})^{-1} \cdot \binom{n}{o}^{-1} \cdot 10 \cdot \binom{n}{o} \cdot m^o \quad = \quad \frac{300}{63} \cdot m^o \text{ minutes}$$

to evaluate all schemas of order o.If each parameter can take on $m = 10$ different values, it already takes 4762 minutes (more than 3 days) to evaluate the schemas of order 3. We see therefore that it seems useless to exploit schemas of order higher than 2.

6. Setup of the Experiments

In this section we describe how we have tried to optimize the values of the weights of the DumbBot. We have tried this with three different methods: a hill climbing search, a standard genetic algorithm and the alternative genetic algorithm as described in the previous section. Each experiment lasted for approximately twenty-four hours.

6.1 A Java Version of the DumbBot

For our experiments we have not used the source code of the DumbBot directly, but a copy of it written in java. The reason for this is that we want to have a bot that is multiplatform and compatible with DipGame. A major disadvantage is that there might be errors in the code, because the C++version and the java version are based upon different frameworks so it is impossible to translate the code literally from C++to java. In fact, in Section 7.1 we will see that unfortunately the java version is indeed not identical to the C++version.

From now on we will refer to our java version of the DumbBot as the *java bot* and to the executable that is written in C++and is available from the website of DAIDE, as the *C++bot*.

6.2 The Evaluation Function

One serious problem that we face when optimizing any Diplomacy bot, is that we don't have a well defined evaluation function. There is no objective way of defining this function, since the only thing we can do is letting seven bots play against each other and see which bot is better than which other bot. We can then do this again with seven other bots, but we can't compare the performance of a bot in the first game with the performance of a bot in the second game, since they might have had different opponents.

For example: suppose that in one game we have bots A and B playing against each other and A ends in a higher position than bot B. Then we know that A is better than B, but we don't know *how much* better. After this game we might play another game with bots A and C and maybe again bot A wins. The only thing we then know is that A is better than both B and bot C, but we don't know anything about whether B is better or worse than C. This means the bots are ordered in a tree-like structure rather than in a chain.

The situation becomes even worse when we realize that there might not even exist a correct ordering of bots at all. That is: bot A might beat bot B and bot B might beat C, but that does not necessarily mean that bot A will also beat bot C. In general, the question of whether a strategy is good or not depends on the strategy of your opponents. Mathematically speaking: an ordering of strategies is not necessarily transitive.

$$A \succeq B, \quad B \succeq C \quad \not\Rightarrow \quad A \succeq C$$

One clear example of this is the well-known paper-scissors-rock game. While rock beats scissors and scissors beats paper, it does not mean that rock also beats paper.

Another problem is that we have to define what exactly it is that we want to optimize. In other words: how do we define *playing well*? We need some system of assigning points to a bot each game and optimize the average amount of points it scores. For instance we might want to optimize the average rank in which the java bot ends a game. That is: it receives 6 points for each victory, 5 points for each game it ends in second place, etcetera. The evaluation function would then be $f(r) = 7 - r$, with r the position in which which the bot ends (its *rank*).

On the other hand one might consider the game of Diplomacy as an *all-or-nothing* game, in which only a victory counts. In that case we are not interested in ending on a high average position, but solely in winning as much games as possible. In that case we should use the number of victories as our evaluation function. A big disadvantage of this last method is that you need to play a lot of games in order to get an accurate estimation of this number, after all we can't expect an average bot to win more than once in every seven games.

For our experiments we have used an intermediate evaluation function, that is quadratic in the rank:

$$f(r) = (7 - r)^2 \quad r \in [1, 7]$$

it provides more information per game than the all-or-nothing function, but puts more focus on winning than the average rank does. For example with this evaluation function the bot prefers to win once and end seventh once, than to end in fourth place twice.

For this evaluation function we still need to define what it actually means to "end in a certain position". After all, the rules of Diplomacy only say that a player with eighteen or more supply centers is the winner, and all the others are the losers. There is no official definition for ending in a second or third place. Therefore, we define the rank of a player of Diplomacy according to the following rules. The players are ranked according to the number of supply centers they own at the end of the game. So the player with eighteen or more supply centers is ranked first, then the player owning the second largest amount of supply centers is ranked second, etc.

The players that were defeated before the end of the game (the players that ended with no supply centers) are ranked according to their year of defeat. The later a player is defeated the higher it is ranked. So if two players end with no supply centers, the one that was defeated first is ranked 7th and the other is ranked 6th.

If, after these rules still two or more players end equally, we decide who is ranked above whom by a coin flip. Alternatively, in the case of two equally ranked players we could have decided to assign them both an equal share of the sum of their evaluation values: if two players both end in second place we could assign them both $\frac{1}{2}(f(2)+f(3))$ points. However, we have not done this, to keep the implementation simple, and because if the number of games played is large enough it doesn't make any difference anymore anyway.

6.3 The Parameters

Since the number of parameters that could be adjusted is very large it seems unrealistic to really optimize them all, at least for our first experiments. Therefore we will focus here on the eight main parameters. The parameters that we chose to adjust are the following:

- the attack weight aw
- the defense weight dw
- the strength weight sw
- the competition weight cw

All of these have a spring value and a fall value. So we have in total eight values to adjust. In the following, we will distinguish between spring and fall weights by prefixing the above names with either s- or f-. So for example s-aw stands for the spring attack weight, and f-cw stands for the fall competition weight.

Notice that the absolute values of the weights are not important, but only their relative values with respect to each other. After all, it is only the differences between the destination values of the respective regions that matter, not the actual destination values themselves. Therefore we can safely keep the zeroth order proximity weight pw^0 fixed, even though it is a very important weight too, and consider all values as 'relative to' the zeroth order proximity weight.

For all experiments we allow the weights to take on one of the following values:

$$\{1, 10, 30, 100, 300, 1000, 3000, 10000, 30000, 100000\}$$

These values increase exponentially. The reason for this is that we think that the order of magnitude of the parameters is more important to know than their precise value. If necessary, we could try to determine more precise values later.

6.4 Group Play vs. Individual Play

All experiments are performed in two versions. In one version we let each java bot play against six C++bots (*individual play*) while in the second version we let the java bots play against each other, together with one or two C++bots (*group play*). The advantage of the second version is that in this way we can train a lot of bots at the same time, so the algorithm runs a lot faster. However, a disadvantage is that each bot plays against different opponents (java bots with different values assigned to their weights) which makes the results somewhat less reliable. Also, if all the java bots that are playing in a particular game are bad, then the bots only learn to play against bad opponents, which is of course not what we want.

6.5 Applied Methods of Optimization

Here we will discuss the exact implementation of the three methods that have tried for the optimization of the weights.

6.5.1 Hill Climbing

The first technique that we have used to learn the parameters is a simple hill climbing. This means we first try to find the best value for the first weight, holding the others fixed. Then we try to optimize the second weight etc. After setting the eight weights we continue again with the first and then the second etc, until a certain amount of time has passed.

1. We initialize ten bots with initial values for their weights.

2. For every bot we change the value of its first weight. The first bot gets the first value from our list of possible values, the second bot gets the second value from the list, etc. We now have ten bots that all have a different value for their first weight, but are identical in their values for all other weights.

3. We let each bot play a number of games (eighty in the case of group play, thirty-five in the case of individual play).

4. We choose the one with the best results and set the weights of the other bots equal to the weights of this winner.

5. We replace the value of the next weight of each bot by one of the ten values from the list. And let them play again.

6. We repeat this for twenty-four hours and after that see which bot is the final winner.

In the experiment with group play in each game five java bots with different values for their weights played with two C++bots.

6.5.2 Standard Genetic Algorithm

We run a standard genetic algorithm as described in Section 5.1. We have a population of thirty-six randomly initialized individuals. In the case of group play they are divided into six groups of six, and each group plays eighty games per iteration. In the case of individual play each java bot plays sixty games against six C++bots. The amount of games per bot is less in this case because otherwise the experiment would take much too long. After each iteration we pick the three best individuals and reproduce them into the next generation. Each individual has a chance of being chosen for cross-over proportional to the evaluation function. Then we pick one out of each eight new individuals for mutation (it is usually advised not to use too much mutation).

We have encoded the weights of the DumbBot as an 8-dimensional vector in the following way:

$$(s\text{-}aw,\quad s\text{-}dw,\quad s\text{-}sw,\quad s\text{-}cw,\quad f\text{-}aw,\quad f\text{-}dw,\quad f\text{-}sw,\quad f\text{-}cw)$$

We think that the attack weights and defense weights are closely related to each other, so therefore we have placed them next to each other, and the same holds for the strength weights and competition weights. Also we argue that spring weights and fall weights are not correlated with each other, because the spring weights only play a role during spring turns, while fall weights only play a role during fall turns. Moreover, it is very reasonable to assume that the bot should follow a different strategy during spring than during fall. Therefore, we separate the spring weights and the fall weights on the chromosome.

6.5.3 Alternative Genetic Algorithm

Finally, we also experiment with our alternative genetic algorithm. We randomly initialize a population of thirty-six individuals. We let phase 1 run until we have 10 samples for each 2-schema on average (that is: after a thousand games, which is after twenty-eight iterations). And then continue with phase 2.

Since for each entry in the vector there are ten different values, but we have thirty-six individuals we can make sure that during phase 1 every 1-schema is present at least once in each generation. Therefore we don't need mutation.

In the second phase however, we have 100 possible 2-schemas for each position on the chromosome, so every time we create a new generation, we put the 80% best 2-schemas in the 'bag of schemas' plus 20% randomly chosen 2-schemas. The chromosomes are cut every time into four equal pieces, which means we combine schemas of the form:

$$(a,b,\cdot,\cdot,\cdot,\cdot,\cdot,\cdot)\quad(\cdot,\cdot,a,b,\cdot,\cdot,\cdot,\cdot)\quad(\cdot,\cdot,\cdot,\cdot,a,b,\cdot,\cdot)\quad\text{and}\quad(\cdot,\cdot,\cdot,\cdot,\cdot,\cdot,a,b)$$

The individuals are encoded in the same way as in the experiment with the standard algorithm. In the case of group play, we let the experiment run for 100 iterations and each bot plays 40 games per iteration. In the case of individual play, the experiment runs 38 iterations and each bot plays 18 games per iteration.

6.6 Evaluation

The result of each experiment is a bot with optimized weights. After each experiment we take this bot and let it run an evaluation session to test how much it has progressed. This

means that we let it play 120 games against six java bots with default weights, to see how well it performs. These default weights are the values as they appear in the C++bot and are given in Section 4.3. Of each recorded game we note the position in which the optimized bot ended and in the end we calculate the average value of these positions. Too see how accurate this calculation is we also calculate the standard deviation of the estimator of this average. That is: the standard deviation of the results, divided by the square root of the number of recorded games. Furthermore we count how often the optimized bot has won.

6.7 Technical Issues

We will now give a technical overview of how the experiments were set up.

6.7.1 The Setup

The experiments are managed by a program called *DumbBotTrainer* that starts and stops the games. In order to have as much computing power as possible, we run games on six different computers at the same time. These computers are controlled remotely by the DumbBotTrainer (over TCP/IP), which runs on another computer.

When the DumbBotTrainer starts a new game, it first writes a text file with all the values that should be assigned to the weights of the DumbBot and then starts a program called *RemoteBotClient*, which makes an TCP/IP connection with one of the six remote machines that each have a program called *RemoteBotServer* running. The client reads the text files with the weights, and sends all this information to the RemoteBotServer. The RemoteBotServer then locally starts a game server (the DAIDE ai-server) and the required number of C++bots and java bots.

When a game is finished, one of the java bots of that game will write the results of the game to a text file (the *results file*). After a certain amount of time has passed (the *kill time* see next subsection) the RemoteBotServer will assume the game is over and will read the file with the results and will send the results back to the RemoteBotClient. Finally the RemoteBotServer will stop the game server and the bots. If a game did not finish before the kill time has passed, there is no results file and the information about the game is lost.

6.7.2 Stopping a Game

The RemoteBotServer somehow has to make sure that the game server and the bots are killed after a game has ended so that it can start a new game. There are two ways how it could decide to stop a game:

- A game could be stopped after the RemoteBotServer has received a message from the game server that the game is over.
- Each game could be stopped after a certain amount of time.

The first option would be the best of course, but it has some problems. One of these problems is that a game sometimes does not show any progress and could continue for ever. We could overcome this problem by stopping the game after a certain amount of turns have been played, if it hasn't yet finished. However, a more serious problem with this option is that the RemoteBotServer needs to act as an observer (that is: it is connected to the server

so that it can receive messages from it) and observers quite often tend to be disconnected from the server for no clear reason, and sometimes even make the program crash completely. Therefore we have opted for the second option.

This means that we have to choose a certain amount of time after which a game should be killed. If we set the amount of time for a game too short, we loose a lot of information, because the results of each prematurely killed game is lost. If we set it too high however, we do record more games, but since the games take more time we might still record less games per minute. By trial and error we determined that 20 seconds is usually the best time. With this kill time, usually about 45-50 % of all games is recorded, so on average we can record one game every 40 seconds.

7. Results

In this section we present the results of the experiments. These results were obtained in the way described in Section 6.6.

7.1 Comparing The Java Bot With The C++ Bot

Before showing the results of our main experiments we first show the results of a session in which the java bot with the default values (that is: the weights of the java bot have the same values as the weights of the C++bots) plays against six C++bots. In principle the java bot should be equivalent to the C++bots so we would expect the java bot to end in each position equally often. However, we see from the following frequency table that this is not the case:

position:	frequency:
1	1
2	10
3	12
4	15
5	20
6	11
7	7

We have recorded 76 games, of which only one was won by the java bot. The average of all the final positions of the java bot is 4.37, with a standard deviation of 0.18. So the average position is lower than the average we would have expected (4.00) and, even more surprisingly, for some reason it fails to win. Apparently there are some errors in the java code which make that the java bot is not an exact copy of the C++bot. Therefore, when we evaluate the results of our experiments we will let the optimized bots play against six java bots with default weights, although the expiriments themselves have been carried out by letting the java bots play against the C++bots. To save space, for all other experiments we will not show the full frequency table, but only the average position of the java bot, its standard deviation, and the number of wins. Also we will display the optimized weights that emerged from each experiment.

7.2 Results of the Experiments

Here follow the results of all six experiments.

7.2.1 Hill Climbing, Individual Play

Average position: 3.75.
Standard deviation: 0.20.
It won 8 out of 72 recorded games.

The optimized weights that we found with this experiment are:

s-aw	=	30,	f-aw	=	10000
s-dw	=	300,	f-dw	=	1000
s-sw	=	10,	f-sw	=	100
s-cw	=	3000,	f-cw	=	30

7.2.2 Hill Climbing, Group Play

Average position: 3.36.
Standard deviation: 0.20.
It won 21 out of 87 recorded games.

s-aw	=	10000,	f-aw	=	300
s-dw	=	100,	f-dw	=	300
s-sw	=	1,	f-sw	=	300
s-cw	=	10000,	f-cw	=	10000

7.2.3 Standard Genetic Algorithm, Group Play

Average position: 3.39.
Standard deviation: 0.23.
Games won: 19 out of 75 recorded games.

s-aw	=	3000,	f-aw	=	3000
s-dw	=	10,	f-dw	=	1000
s-sw	=	10000,	f-sw	=	10000
s-cw	=	100,	f-cw	=	300

7.2.4 Standard Genetic Algorithm, Individual Play

Average position: 3.07.
Standard deviation: 0.22.
Games won: 25 out of 82 recorded games.

s-aw	=	1000,	f-aw	=	10000
s-dw	=	30,	f-dw	=	1000
s-sw	=	100000,	f-sw	=	300
s-cw	=	300,	f-cw	=	300

7.2.5 Alternative Genetic Algorithm, Group Play

Average position: 3.38.
Standard deviation: 0.18.
Games won: 27 out of 108 recorded games.

s-aw	=	1000,	f-aw	=	100
s-dw	=	10,	f-dw	=	30
s-sw	=	10000,	f-sw	=	300
s-cw	=	100,	f-cw	=	30

7.2.6 Alternative Genetic Algorithm, Individual Play

Average position: 3.79.
Standard deviation: 0.27.
Games won: 9 out of 52 recorded games.

s-aw	=	300,	f-aw	=	30
s-dw	=	10,	f-dw	=	1
s-sw	=	300,	f-sw	=	100
s-cw	=	1,	f-cw	=	30000

8. Discussion

The experiment with hill climbing using individual play and the experiment using the alternative genetic algorithm with individual play have not led to weights that are clearly better than the default values. All the other experiments however have lead to an improvement of the results. On average they end in a position significantly lower than 4th and they clearly win more than one out of seven games, proving that indeed they play better than the java bots with default values. Especially the standard genetic algorithm with individual play has led to good results. We can summarize this with the following diagram:

	Hill Climbing	Standard GA	Alternative GA
Individual play	**Bad**	**Very Good**	**Bad**
Group play	**Good**	**Good**	**Good**

Strangely enough, although we have found four sets of values that perform better than the default set, these four sets are completely different. So it is not clear at all why these sets of values perform better than the default set.

We also see that the alternative GA did not function better than hill climbing, although we argue that much more experiments should be performed in order to draw any real conclusions about the algorithms. One particular problem is that we had to choose certain parameters for each experiment and these parameters might have a big influence on the final result. For instance: we have chosen to let each experiment run for twenty-four hours, however, because individual play is much slower than group play, this means we have to shorten the experiments with individual play somehow. This can be done either by letting it run less iterations, or by playing less games in each iteration, or a combination of both.

9. Future Work

One thing that we have learned is that the java version of the DumbBot that we have is not correct (it is not an exact copy, as it should be). It is almost incapable of winning a game, and therefore almost useless. So we have to find the errors in the code in order to make it an exact copy of the original DumbBot. Another possible improvement is that we could try to optimize all the weights instead of just the eight most important ones.

The DumbBot is totally unable to negotiate. Since the goal of DipGame is to develop a testbed for negotiation in multi agent systems, a future target would be to extend the bot with negotiation capabilities. However, first it should be able to at least coordinate its own moves and have a better sense of strategy. If the bot is unable to have some advanced strategic reasoning, it would not help much to be able to do any negotiations. This means it should be able to estimate how valuable different game configurations are, not only for itself and for its opponents, but also for every possible coalition of players. It should think multiple steps ahead, and it should consider what its opponents might do.

Furthermore we argue that a better game server should be developed that provides the possibility of playing more than one game without having to kill it and start it again. Also it should be more stable than the DAIDE ai-server, especially with regard to observers, and it should be able to run on systems other than Windows. Another useful feature would be to allow to setup specific game situations so that it is possible to see what happens under specific circumstances (this would be especially useful in order to find the inconsistency between the java bot and the C++bot).

References

Daide. http://www.daide.org.uk/.

Dipgame. http://www.dipgame.org/.

Rules of Diplomacy. Avalon Hill, 4 edition, 2000.

Angela Fabregues and Carles Sierra. A testbed for multiagent systems. Technical report, IIIA-CSIC, Bellaterra, Barcelona, 10/2009 2009.

Emanuel Falkenauer. *Genetic Algorithms and Grouping Problems*. Wiley, 1998.

Melanie Mitchell, John H. Holland, and Stephanie Forrest. When will a genetic algorithm outperform hill climbing? In *Advances in Neural Information Processing Systems 6*, pages 51–58. Morgan Kaufmann, 1993.

David Norman. David's diplomacy ai page. http://www.ellought.demon.co.uk/dipai/.

Lothar M. Schmitt. Theory of genetic algorithms. *Theoretical Computer Science*, (259): 1–61, 2001.