

# Optimizing Lies in State Oriented Domains based on Genetic Algorithms

A. Zylberberg<sup>1</sup>, E. Calot<sup>1</sup>, J. Ierache<sup>2,1</sup>, H. Merlino<sup>1,3</sup>, P. Britos<sup>3,1</sup>, R. García-Martínez<sup>3,1</sup>

1. *Laboratorio de Sistemas Inteligentes. Facultad de Ingeniería. Universidad de Buenos Aires*

2. *Instituto de Sistemas Inteligentes y Enseñanza de la Robótica. Universidad de Morón*

3. *Centro de Ingeniería del Software e Ingeniería del Conocimiento. Escuela de Postgrado.*

*Instituto Tecnológico de Buenos Aires*

*jierache@unimoron.edu.ar, {hmerlino, pbritos, rgm}@itba.edu.ar*

## Abstract

*This paper explores the use of genetic algorithms in identifying effective lies for use in negotiations with incomplete information in State Oriented Domains (SOD's). The aim is to seek lies which are not just safe (i.e. they will not lead to a disgraceful outcome) but also yield the highest possible utility. We propose a representation of a goal, a mechanism to assess aptitude and suitable GA operators.*

## 1. Introduction

In the simplest possible scenario for a non-trivial negotiation, agents will state their objectives and then work their way towards a satisfying solution [1]. Finding such a solution will probably include agreeing on a final situation that is acceptable to every agent, as well as dividing the work necessary to reach that situation. Thus different roles will be designed, and in general the roles will have different costs. Fair enough, the agent with the most onerous objective will play the most expensive role. Naturally, it is in an agent's best interest to achieve his goal at the lowest possible cost [2]. That is where an agent may be tempted into stating an objective that is different from his actual goal. Lies can be beneficial, inasmuch as pretending to have a cheaper goal might result in doing less work in the joint plan. Provided there exists a mechanism that will guarantee the simultaneity of the exchange of information on the goals, speculation with lies is not just greatly restricted but also dangerous, since the negotiation could result in a solution that will not satisfy the agent's true goal. Nevertheless, in a context where abiding by such a mechanism is not mandatory, there may come to occur that an agent will become acquainted with the other agent's goal before the latter learns his. Should this be the case, lying can be safe, for it is possible to ensure that the outcome will not be

disgraceful. Moreover, the agent can focus on finding a lie which is not just safe but also yields the highest possible utility. Firstly we present the reader with a simplified version of state oriented domains (section 2) and a genetic representation of a goal (section 3). Then we discuss how to assess the satisfaction of a goal (section 4), generate lies (section 5), assess their aptitude (section 6) and improve them through sensible mechanisms of selection (section 7), crossover (section 8) and mutation (section 9). Finally, we debate improvements to the model proposed (section 10.1) as well as extrapolation of the concepts developed to contexts with less strict hypotheses (section 10.2).

## 2. A simplified version of State Oriented Domains

The State Oriented Domain model [3] is intended to address a very wide range of problems. However, for the sake of clarity, in this paper we will apply some restrictions, in order to be able to study how to use genetic algorithms to find effective lies without losing focus by having to analyze endless general cases. In section 8 we discuss the generalization of our concepts. Some of the restrictions that we apply to SOD's are:

- We will study negotiations involving only two agents. We will refer to the agents as “us” and “rival”, and use the subscripts U and E respectively.
- The rival agent will always declare his goal first, and will always be telling the truth. Then we will take into consideration all the aspects involved, such as current world state, the rival's goal and our goal, and produce a lie, which is what we will declare our goal to be.
- As regards plans, we will only study pure deals. More complex types of deals, such as mixed deals, semi-cooperative deals and multi-plan deals add little

to the general idea of finding effective lies, and thus will not be analyzed in depth.

## 2.1 State Oriented Domains

Accordingly, we define a State Oriented Domain (SOD) as a tuple  $\langle S, P, c \rangle$  where:

$S$  is the set of all possible world states.

$P$  is the set of all possible plans. A plan  $p \in P$  moves the world from one state in  $S$  to another. In each plan there are two roles,  $p_1$  and  $p_2$ , each consisting of a series of actions. As a particular case, a role could be null.

$c$  is a function  $c: P \rightarrow (\mathbb{R}^+)^2$ . For each plan  $p$  in  $P$ ,  $c(p)$  is a pair of positive real numbers, the cost of each role in the plan. If a role is null, its cost is 0. The higher cost in the plan is noted  $c^+$ , and the lower cost is called  $c^-$ . As a particular case, both roles could have the same cost.

## 2.2 Goals, Plans, Deals and Utilities

**Definition 2.2.1** A goal is a subset of  $S$ . For instance, an agent's goal is the set of all world states that satisfy him. Notation:  $G_U$  our goal,  $G_R$  our rival's goal,  $G'_U$  our lie. That is to say, what we declare our goal to be,  $G_J$  the joint goal. Namely,  $G_J = G_U \cap G_R$

**Definition 2.2.2** An encounter within an SOD  $\langle S, P, c \rangle$  is a tuple  $\langle s_0, G_U, G_R \rangle$  such that  $s_0 \in S$  is the initial state of the world,  $G_U$  is the set of all world states that satisfy us and  $G_R$  is the set of all world states that satisfy the rival. Note that  $G_U$  can actually be  $G'_U$  if we are lying. In fact, what we study in this paper is how to determine an effective  $G'_U$  to declare instead of  $G_U$ .

**Definition 2.2.3** A stand-alone plan is a plan in which only one of the agents has an active role (i.e. does something). The set of all stand-alone plans is noted  $A$ . It follows that  $A \subset P$ .

**Definition 2.2.4** A joint plan is a plan in which both agents have an active role. The set of all joint plans is noted  $J$ .

**Definition 2.3.4** Given an initial world state  $s_0$  and a goal  $G$ , the stand-alone cost is the cost of the stand-alone plan that moves  $s_0$  to a state in  $G$  at the minimum cost. Namely:  $l = \min_{a \in W} [c(a)]$ . Where  $W \subset A$  is the set of stand-alone plans which satisfy  $G$  (from the initial

world state  $s_0$ ). Notation:  $l_U$  our stand-alone cost,  $l_R$  our rival's stand-alone cost,  $l'_U$  our fake stand-alone cost.

**Definition 2.2.5** The utility of an agent is the difference between the cost he would have to pay to reach his goal alone, and the cost of the role in the joint plan that the negotiation results in. If in a joint plan we are assigned the most expensive role, our cost is  $c^+$ , and otherwise it is  $c^-$ . Let then  $c_U$  be our cost, our utility is:  $U_U = l_U - c_U$

**Definition 2.2.6** Given an encounter  $\langle s_0, G_U, G_R \rangle$ , a deal is a joint plan  $j \in J$  that moves the world from  $s_0$  to a state  $s_f \in G_J$ .

**Definition 2.2.7** A deal is individual rational if the utility of each agent is not negative.

**Definition 2.2.8** A deal is pareto optimal if there does not exist another deal that is better for one of the agents and not worse for the other.

**Definition 2.2.9** The negotiation set NS is the set of all deals that are both individual rational and pareto optimal.

## 3. Genetic representation

Finding an effective lie can be treated as an optimization problem, hence the use of genetic algorithms. In every genetic algorithms model [2], there is a population of individuals (namely, possible solutions to the problem) which get improved generation after generation.

### 3.1 What individuals are

Our problem is to find an effective lie, and each individual in our population must be a possible solution to the problem. Consequently, each individual will be a lie. A lie is a goal (it is not our *real* goal, but it is a goal), therefore it is a subset of  $S$ . Finding a convenient representation for an individual in our genetic algorithms model is then finding a convenient representation for subsets of  $S$ .

### 3.2 Representing subsets of S

If  $G$  is a subset of  $S$ , then each world state in  $S$  may or may not be in  $G$ . Therefore, if there are  $n$  elements in  $S$  (that is to say, the world has  $n$  possible states) then there are  $2^n$  possible subsets of  $S$ . For instance, if  $S = \{\text{"the book is on the table"}, \text{"the dog is out"}, \text{"the window is open"}\}$  then there are 8 possible subsets of

$S$ , that is to say, 8 possible goals or lies. A first approach could be enumerating the elements of  $S$  that are present in the subset. This can be achieved by finding a function that will map every element of  $S$  unequivocally to a natural number in  $[1;n]$ . Then a subset of  $S$  can be represented by a stream of  $n$  bits, where the  $i$ -th bit will be 1 if the  $i$ -th element of  $S$  is present in  $G$ , and 0 otherwise. However, this approach has two main difficulties. Firstly,  $n$  could be a very large number. The world does not even need to be too complex to have millions of possible states. For example, the permutation of only 10 elements consists of  $10! = 3,628,800$  possibilities, and that means individuals of 442 kilobytes. Secondly, finding an analytical function that maps every element of  $S$  unequivocally to a natural number in  $[1;n]$  might not be feasible. We could still make a list of all the possible states and assign a number to each item on the list, but the list can be extremely large and require too much memory. Another approach is to represent a subset of  $S$  as a condition. Then  $G$  will be the set of all the elements in  $S$  that satisfy the condition. This solves the problems mentioned for enumeration, at the cost of introducing additional complexity in the design of our algorithms.

### 3.3 Representing conditions

If  $S$  is the set of all the apartments that can be bought, a lie could then be: "I want an apartment with 3 bedrooms". Then  $G$  would be the set of all the apartments in  $S$  that have 3 bedrooms. However, the condition could be more complex: "I want an apartment with 3 bedrooms and 2 bathrooms and it must have a balcony, or at least a patio. Closets scare me, so I definitely don't want one". This last condition can be rewritten as: "(3 bedrooms) and (2 bathrooms) and (a balcony or a patio) and (not a closet)". We then note that a complex condition can be expressed in terms of atomic conditions and logical operators. In defining the possible atomic conditions, caution must be exercised, since they must be sufficient to cover every subset of  $S$ .

### 3.4 Condition trees

We will represent a goal as a tree [3]. We will call such a tree a *condition tree*. A state  $s$  in  $S$  is also in  $G$  if and only if it satisfies the root of the tree (see section 4). Condition trees have 3 different types of nodes: *cond*, *and*, or. *Cond* nodes contain atomic conditions. A *cond* node is satisfied by a state  $s$  if  $s$  satisfies the atomic condition contained in the node. For example,

if the atomic condition is "3 bedrooms" then the node will be satisfied only by every  $s$  which has 3 bedrooms. Every leaf in a condition tree is a *cond* node, and every *cond* node in a condition tree is a leaf. Also, note that if a tree only has one node, then that node would be a *cond* node. *And* nodes and *or* nodes have children. An *and* node is satisfied if and only if all of its children are satisfied. An *or* node is satisfied if and only if at least one of its children is satisfied. *And* and *or* nodes should always have at least 2 children. In figure 1 we see an *and* node with 3 children. Note that every child of an *and* or *or* node can be any type of node, namely another *and* or *or* node (nested operators) or a *cond* node. It might seem not reasonable for a child of an *and* node to be another *and* node, since for example:

$$a \text{ and } (b \text{ and } c) = a \text{ and } b \text{ and } c$$

The same happens with *or* nodes:

$$a \text{ or } (b \text{ or } c) = a \text{ or } b \text{ or } c$$

Nevertheless, such relationships between nodes may be allowed for the sake of diversity. This is discussed in section 6. Finally, note that there is no need for *not* nodes, since "not" is a unary operator. Accordingly, every node will have a "not" flag, which if activated will invert the node's satisfaction value. If an *and* node has the "not" flag activated, it will be satisfied if and only if any of its children is not satisfied, since by De Morgan's law [4]:

$$\text{not}(a \text{ and } b) = \text{not}(a) \text{ or } \text{not}(b)$$

If an *or* node has the "not" flag activated, it will be satisfied if and only if none of its children are satisfied, since by De Morgan's law [4]:

$$\text{not}(a \text{ or } b) = \text{not}(a) \text{ and } \text{not}(b)$$

If a *cond* node has the "not" flag activated, it will be satisfied if and only if the state does not satisfy the atomic condition it contains. In figure 2 we depict the condition tree for the second example of section 3.3.

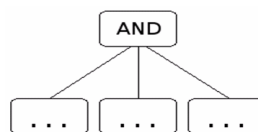


Figure 1. An *and* node with 3 children



Figure 2. Condition tree for second example in 3.3.

### 3.5 Implementation

All nodes must have a "not" field, all nodes must have a "type" field; *and* and *or* nodes must have a list of pointers/references to nodes; *cond* nodes must have an atomic condition. An atomic condition is typically an instance of a class or struct having fields like: type

of condition (e.g. 1="has", 2="has a surface of", 3="has a price of"), numerical parameters (e.g. "number", "amount"), modifiers (e.g. "is exactly", "is less than", "is equal or greater than"), etc. (e.g. for the condition type 1 in this example, a field indicating 1="bedroom", 2="bathroom", 3="patio", etc.). As we said above the atomic conditions defined must be sufficient to make it possible to represent any goal an agent may have. For example, if there are two elements in  $S$  such that the only difference between them is the color a room is painted, then there must be a way to specify the color of rooms in an atomic condition.

#### 4. Assessing the satisfaction of a goal

When we implement our genetic algorithm, we will need a function that determines whether a given state  $s$  in  $S$  satisfies a given goal  $G$ , that is to say, whether  $s$  is also in  $G$ . In section 3.4 we said that a state  $s$  in  $S$  is also in  $G$  if and only if it satisfies the root node of the condition tree for  $G$ . The function we are to implement will be recursive. Its parameters will be a state and a reference/pointer to a node. When it is necessary to determine whether  $s$  is in  $G$ , the function will be passed  $s$  and a reference/pointer to the root of the condition tree for  $G$ . The function will behave differently depending on the type of the node it receives:

- If it is a *cond* node, it will return true or false depending on whether the state satisfies or not the condition.
- If it is an *and* node, it will call itself recursively for every child of the node. Then if all the calls return true, it will return true. Else, it will return false. An optimization can be made here: if any of the calls returns false, then the function can return false without performing the calls for the remaining children.
- If it is an *or* node, it will call itself recursively for every child of the node. Then if all the calls return false, it will return false. Else, it will return true. An optimization can be made here: if any of the calls returns true, then the function can return true without performing the calls for the remaining children.

#### 5. Generating the initial population

Unless for some reason we are able to determine a convenient set of data for the initial population, normally we will resort to simulation [2]. Simulation consists of creating random individuals whose parameters follow certain distributions [5]. A distribution is a set of pairs of possible values and

associated probabilities, for example: A 30%, B 50%, C 20%. In section 5.1 we explain how to simulate a value, in section 5.2 we discuss which values should be simulated, and in section 5.3 we discuss the distributions and their effects.

#### 5.1 Simulating a value

Typically, a random number function will be available. Without loss of generality, we will assume the function generates random numbers in the interval  $[0;1)$ . That is to say,  $R$  is a random variable with a uniform(0;1) distribution. Let  $X$  be the value we want to simulate,  $P_X(x)$  will relate the possible values to their probabilities. From that function we can obtain the cumulative distribution function:

$$F_X(x) = P(X \leq x) = \sum_{-\infty}^x P_X(x)$$

This function will start with value 0 at  $-\infty$  and have at every possible value of  $X$ , where the height of the leap will be the probability of that value. The simulated values of  $X$  will be  $F_X^{-1}(r)$ , where  $r$  are values obtained from the random function. In figure 3 we show  $F_X^{-1}(r)$  for A, 30% ; B, 50% ; C, 20%.

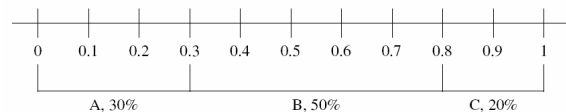


Figure 3. Values of  $F_X^{-1}(r)$  for A, 30% ; B, 50% ; C, 20%.

In the figure 3 we can appreciate how the values of the random function are divided between A, B and C, according to their probabilities. If we get 0.62393 from the random function, the simulated value will be B. If we get 0.29218, the simulated value will be A. The pseudo-code would be:

```
generate random value;
if (random value < 0.3) return A;
else if (random value < 0.3+0.5) return B;
else return C;
```

#### 5.2 What will we be simulating?

Before any analysis is carried out, we would like to advice against hard-coding the probabilities for the simulations inside the program's main code. Instead, they can be conveniently placed with constants or *#define's* in a separate header or configuration file. We will now go through the process of simulating an individual. We will need a function called CreateNode. It will be a recursive function. This function will first

decide randomly which type of node it will generate, according to the probabilities defined. If it decides it will generate a *cond* node, then it must generate an atomic condition (see below) and return. If it decides to generate an *and* or an *or* node, then it must generate the number of children, and then call itself recursively to generate each child. Generating an atomic condition consists basically of applying the principles and method of simulation to each of the fields that describe a condition for the particular problem we are solving. For instance it will first decide randomly what type of condition it will be, and then simulate all the values for that type of condition.

### 5.3 Effects of the probability distributions

Keeping the actual probabilities used outside the program's code allows for easier experimentation. We recommend doing so. Experimenting with these parameters can give valuable information about the problem. Using a low probability for *or* nodes and a high probability for *and* nodes will make the initial lies more less specific and more vague, whereas using a high probability for *or* nodes and a low probability for *and* nodes will make the initial lies very restrictive. The latter might not be a good idea because in the beginning we want to broaden our options. Using a low probability for *cond* nodes will make the initial population complex, as the trees will go up several levels. If you intend to favor simple answers in the beginning of the search, you should use a high probability for *cond* nodes. Experimenting in various domains, we have found the following probabilities to yield, in our opinion, quite reasonable results: #define prob\_and\_node 0.1, #define prob\_or\_node 0.2, #define prob\_cond\_node 0.7. Be aware that any values you use are very strongly domain-dependent. Use the numbers given if you do not know where to start.

## 6. Aptitude assessment

The evolution mechanisms rely entirely on the aptitude of individuals, hence the cruciality of its computation. Note that it is extremely important to make sure that the lie we tell does not result in a negotiation that will lead to a disgraceful final world state, that is to say, a state that does not satisfy our *real* goal  $G_U$ . This is studied in section 6.4. However, for now we will assume that none of our lies can be disgraceful. If we say our goal is  $G'_U$ , and an agreement is reached, then  $c'_U$  will be our cost in the joint plan. We will define the aptitude of an individual  $G'_U$  to be  $E[c'_U]$ , that is, the expected value [5] of our

cost in the joint plan provided we state our goal is  $G'_U$ . We are therefore going to minimize  $E[c'_U]$ . We could as well define the aptitude to be  $E[c_U - c'_U]$  and maximize, but  $c_U$  is constant and also we would be getting negative values. Note that working only with positive values does not keep us from checking our results for individual rationality. Before we continue, we would like to point out that we are computing the expected value of the costs instead of the costs themselves because there may be the case where the agents have to toss a coin to decide who will play each role. In that event, the cost actually becomes a random variable [5]. The problem is then computing  $E[c'_U]$  for a given lie  $G'_U$ . It follows that we will need to determine (or guess, if the rules are not so clear) how the negotiation would go were the rival to state his true goal  $G_R$  and us our fake goal  $G'_U$ . The negotiation mechanism will determine a pair of roles which are pareto optimal, and then assign each of them to an agent, based on the standalone costs and individual rationality.

### 6.1 Computing $E[c'_U]$

If the standalone cost of any of the agents is less than  $c$ , then the negotiation will not be individual rational. In that case, our apparent cost will equal our standalone cost, for we will have to do the work on our own (unless we are forced to cope with the existence of the other agent, this of course depending on the negotiation rules). But our real final cost will be  $l_U$ , because if we have to do the work alone we will probably pursue our real goal. If the standalone costs of both agents is no less than  $c$ , then both agents are interested. Note that if the standalone cost of an agent is equal to  $c$ , then he will agree to negotiate just to help the other agent, therefore achieving pareto optimality. If the standalone costs are the same, then a fair coin will be tossed to determine which role will be played by each agent. Our expected cost is  $E[c'_U] = \frac{1}{2} \cdot c_- + \frac{1}{2} \cdot c_+ = (c_- + c_+) / 2$ . If the costs of the roles are equal, then there is no need to toss the coin, and  $c'_U = c_- = c_+$ . In fact if the costs of the roles are equal, it never matters which role is played by each agent. If the standalone costs are different, and the costs of the roles are different, then the analysis becomes more complex. The utility available to be shared between the agents is the difference between the sum of the standalone costs and the sum of the costs of the roles. If the difference between the standalone costs is greater than the utility available to share, then the agent with the higher standalone cost must get the most expensive role (i.e.  $c_+$ ). Otherwise, a coin will be tossed, with a

probability such that the utility available is shared equally between the agents. Our expected cost is then:

$$E[c'_U] = p \cdot c^- + (1-p) \cdot c^+ \quad [6.1.a]$$

Let  $u$  be the utility available to be shared. Let  $p$  be the probability that the coin favors us (i.e. we get  $c^-$ ), it is computed as follows. Since the utility must be shared equally between the agents, our expected utility  $E[U'_U]$  must equal half that utility. Therefore we get:

$$p \cdot (l'_U - c^-) + (1-p) \cdot (l'_U - c^+) = u / 2 \quad [6.1.b]$$

and we solve for  $p$ :

$$p = \frac{2c^+ - 2l'_U + u}{2(c^+ - c^-)} \quad [6.1.c]$$

If then we plug [6.1.c] in [6.1.a] we get:

$$E[c'_U] = l'_U - u / 2 \quad [6.1.d]$$

Now, the pseudo code for the above mechanism:

```

if (c'_U < c- OR c_R < c-) then return l_U
if (l'_U = l_R) then return (c- + c+) / 2
if (c- = c+) then return c-
u = l'_U + l_R - c- - c+
if abs(l'_U - l_R) > u then:
    if (l'_U > l_R) return c+ else return c-
return l'_U - u / 2

```

We have studied how to compute our expected cost. As we have showed, we will need to compute beforehand the standalone costs and the costs of the best joint plan. We will now focus on that.

## 6.2 Computing a standalone cost

For this task we use a recursive function. Its basic parameters are a world state, a goal and a counter to keep track of depth.

It is first called with the initial state of the world  $s_0$ . For each operation the agent can perform given that world state, it calls itself with the hypothetical world state that would result from carrying out that operation, and increasing the counter parameter. When the state satisfies the goal (see section 4), the function simply returns the counter's value. Every instance of the function that does not satisfy the goal returns the best value from the calls it made. This guarantees that when the first instance returns, the minimum standalone cost will be returned. In order to prevent infinite recursion, a maximum depth must be established. If the function is called with a counter value that exceeds the maximum, it returns "infinity". Both the maximum and the "infinity" values should be defined outside the program's body using constants. Also note that you may need to include additional parameters to this function, to store information not reflected in the state of the world. For instance, if your definition of a world

state includes the positions of blocks on a table but it does not take into consideration the blocks an agent might be carrying, then after a "pick up block" operation, the next recursive call should include the information that the agent is carrying a block. However, we advice you do include all the relevant information in your definition of world state. A performance optimization that should be considered for recursive functions like the one suggested here is the use of dynamic programming. It basically consists of adding memory to the recursive function, so that every instance will, prior to performing its normal computation, look up the received state on a list to check if it has been computed before. This is a trade between memory usage and speed.

## 6.3 Determining the costs of the roles of the best joint plan

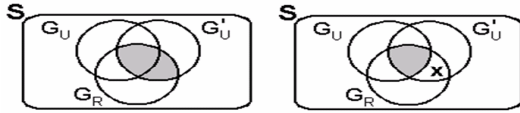
This task is very similar to the previous one. The relevant differences are:

- The goal it receives is  $G_J' = G_U' \cap G_R$ . This can be implemented by creating an *and* node with 2 children,  $G_U'$  and  $G_R$ , and passing this new node to the function.
- Now it will try all the possible operations for role A and role B.
- Consequently, it will not have one but two counters: one for the number of operations in role A and another for the number of operations in the role B.
- Thus, it will not return a number but a pair of numbers.

## 6.4 Making sure the lie is not disgraceful

A lie is said to be disgraceful if it results in a negotiation that will lead to a final world state which does not satisfy our real goal  $G_U$ . Provided the negotiation mechanism is clear and unequivocal, then it is always possible to know whether a lie is disgraceful or not. However, that is hardly ever the case. For instance, say  $G_J'$  can be achieved by a joint plan with 2 roles of cost 2 each, leading to a satisfactory (i.e. not disgraceful) result. If the negotiation mechanism does not even guarantee that a joint plan with 2 roles of cost 2 each will be found, then definitely any lie could turn out to be disgraceful, since *anything* can happen. But note that even if the mechanism guarantees that a joint plan with 2 roles of cost 2 each is found, a lie could still be disgraceful, because the negotiation would simply lead to *any* final state that can be reached by some joint plan having 2 roles of cost 2 each. That is to say, not necessarily to

the final state we wanted, but actually to any final state that can be reached at the same cost. Therefore, for the negotiation mechanism to be safe, it needs to be clear and unequivocal, namely given an initial state and a joint goal it must always lead to the same final state. And like we said before, that could not be the case. So how do we make sure we never tell a lie that could turn out to be disgraceful? We can start by pointing out that we will never accept a final state which is not in  $G_U'$ , and the rival will never accept a final state which is not in  $G_R$ . Consequently, the final state is necessarily in  $G_J' = G_U' \cap G_R$ . In figure 4 we can see how easily a lie can be disgraceful, if it is in  $G_J'$  but not in  $G_U$ . However, we can choose to only tell lies such that no world states which satisfy the fake joint goal but not our real goal exist. In figure 5 we illustrate that since the final state will necessarily be in  $G_J' = G_U' \cap G_R$ , if we only tell lies such that  $\overline{G_U} \cap G_R \cap G_U' = \emptyset$  then we could never end up in a disgraceful world state.



**Figure 4.** A disgraceful lie. **Figure 5.** A graceful lie.

This could seem excessively restrictive, inasmuch as it might make many interesting results impossible. But in countless situations it is the only possible or reasonably simple way to guarantee that an acceptable final state will be reached.

## 7. Selection

There is nothing in particular to say about selection methods when it comes to this application. Any means of selection could be appropriate. Basically, we advice to experiment and determine the best method for the domain being used.

## 8. Crossover

### 8.1. Considerations

In general, a good crossover method should produce children that are similar to their parents. Otherwise, it becomes similar to random generation. Hence the need for a means to measure the similarity of any two goals. Therefore, it is necessary to define a metric so that distance between goals can be assessed. Any goal can be represented as a boolean function of the conditions its tree has. And boolean functions can be expressed with minterms and maxterms. We will

define the distance between two goals as the number of minterms they differ in.

## 8.2. Definitions

**Definition 8.2.1** The *distance* between two goals  $G_0$  and  $G_1$  is:

$$d(G_0, G_1) = \#((G_0 \cup G_1) - (G_0 \cap G_1))$$

where # represents the number of minterms in a boolean function having as variables the conditions in  $G_0$  or  $G_1$  or both. Note that this definition is also equivalent to:

$$d(G_0, G_1) = \#(G_0 \cup G_1) - \#(G_0 \cap G_1)$$

**Definition 8.2.2** A *valid cross* between two goals is a linear combination of those two goals. This means that the minterms that are present in both parents must be present in the child, and that the minterms that are not present in either parent must not be present in the child. Children may or may not have the minterms that are present in only one of the parents.

**Definition 8.2.3** The set of all *valid children* of the goals  $G_0$  and  $G_1$  is  $C_{G_0, G_1}$ . We can imagine a linear

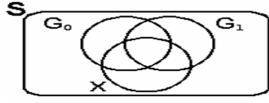
function  $C = mX + b$ , there the slope is  $m = G_0 \cup G_1$  and the C-intersect is  $b = G_0 \cap G_1$ . Then by using different values of  $X$  we get the set of valid crosses. This form allows for easy comprehension that:

$$C_{G_0, G_1} = \{(X \cap (G_0 \cup G_1)) \cup (G_0 \cap G_1) \mid \forall X \subseteq S\}$$

which can be simplified to:

$$C_{G_0, G_1} = \{(X \cap G_0) \cup (X \cap G_1) \cup (G_0 \cap G_1) \mid \forall X \subseteq S\}.$$

We will now show that  $m = G_0 \cup G_1$  and  $b = G_0 \cap G_1$  are in fact the slope and the C-intersect. Figure 6 illustrates definition 8.2.3. Definition 8.2.2 forces every child to have the minterms present in both parents. These minterms are the minterms present in the intersection of the goals. Therefore, by using  $b = G_0 \cap G_1$  we guarantee that the common minterms will be added to every valid child. Definition 8.2.2 also implies that children may have any minterm as long as it is present in one of the parents. Nevertheless, note that since the minterms present in both parents are always present in the children, then this condition can be simply expressed as: children may have any minterm as long as it is present in at least one of the parents, namely  $G_0 \cup G_1$ . Thus, being the slope  $m$  that union,  $X$  will make some of the minterms present in at least one of the parents present in the child.



**Figure 6.** Illustration of definition 8.2.3

Finally, note that since the minterms that are not present in either parent are not present in  $m$  or  $b$ , they are never present in the children. It is easy to see that:

- 1.-  $X \equiv \emptyset \Rightarrow c \in C_{G_0, G_1} = G_0 \cap G_1$
- 2.-  $X \equiv S \Rightarrow c \in C_{G_0, G_1} = G_0 \cup G_1$
- 3.-  $X \equiv G_0 \Rightarrow c \in C_{G_0, G_1} = G_0$
- 4.-  $X \equiv G_1 \Rightarrow c \in C_{G_0, G_1} = G_1$

In spite of the fact that 3 y 4 are valid crosses, they are not effective because the child is identical to one of the parents. That would be selection rather than crossover. As regards 1 and 2, they are valid, effective and easy to implement crosses. Performing intersections and unions alternatively is the simplest way to have a powerful crossover mechanism. However, one may argue that using only intersections and unions might hinder diversity. Consequently, we will now study other options for  $X$ .

Let  $K_{G_0, G_1} \subseteq C_{G_0, G_1}$  be the subset of all computationally feasible crosses. We are only interested in crosses that are in  $K$ . Let  $G_0$  and  $G_1$  be two trees with or and and root nodes respectively. They can be expressed as  $G_0 = \sum_{\forall n} S_n$  and  $G_1 = \prod_{\forall n} R_n$ . If we then partition the  $S$  conditions in two sets  $A$  and  $B$ , and the  $R$  conditions in two sets  $C$  and  $D$ , we get:  $G_0 = A + B$  and  $G_1 = CD$ . The problem is then reduced to crossing  $A+B$  and  $CD$ . We will now use the linear function and a seed  $X$  to obtain valid crosses. Plugging  $G_0$  and  $G_1$  in definition 8.c we get:

$$C_{A+B, CD} = \{(X \cap (A+B)) \cup (X \cap CD) \cup ((A+B) \cap CD) \forall X \subseteq \zeta\}$$

$$C_{A+B, CD} = \{(X(A+B)) + (XCD) + ((A+B)CD) \forall X \subseteq \zeta\}$$

Simplifying:

$$C_{A+B, CD} = \{X(A+B+CD) + (A+B)CD \forall X \subseteq \zeta\}$$

Finally, it suffices to choose  $X$  conveniently to get elements in  $K$ .

$$X = A \Rightarrow c \in K_{A+B, CD} = A(A+B+CD) + (A+B)CD = A + BCD$$

$$X = B \Rightarrow c \in K_{A+B, CD} = B(A+B+CD) + (A+B)CD = B + ACD$$

$$X = C \Rightarrow c \in K_{A+B, CD} = C(A+B+CD) + (A+B)CD = C(D+A+B)$$

$$X = D \Rightarrow c \in K_{A+B, CD} = D(A+B+CD) + (A+B)CD = D(C+A+B)$$

These crosses are not just computationally feasible but also effective because they involve all four  $A, B, C, D$ . Using the same procedure it is possible to obtain:

- Children which are closer to  $G_1$  because they do not depend on either  $A$  or  $B$ :  
{ $C(D+B); C(D+A); D(C+A); D(C+B)$ }
- Children which are closer to  $G_0$  because they do not depend on either  $C$  or  $D$ :  
{ $A+BC; A+BD; B+AC; B+AD$ }

## 9. Mutation

When we want a mutated individual  $G'$  to be similar to the original individual  $G$ , the simplest way is to perform  $G' = G \cup W$ , where  $W$  is a minterm based on the conditions present in  $G$ . For a greater variation, we can use as  $W$  a portion of a minterm, that is to say, the intersection of some of the conditions present in  $G$ , some of which will have a not operator applied. For a complete variation, we can always resort to generation, as described in section 5.

## 10. Conclusions

We have explored in this paper how genetic algorithms can be used to identify effective lies for use in negotiations with incomplete information in State Oriented Domains (SOD's) seeking lies which are not just safe but also yield the highest possible utility. The proposed model has been implemented and testing is being carried out. Research in lie optimization is a novel area so there are not many results for comparison. Ethical considerations have been put aside to show a naïve way in which autonomous intelligent systems may lie during an automated negotiation process.

## 11. References

- [1] García Martínez, R. y Borrajo, D.2000. *An Integrated Approach of Learning, Planning and Executing*. Journal of Intelligent and Robotic Systems 29(1):47-78.
- [2] García-Martínez, R., Borrajo, D., Britos, P. y Maceri, P. 2006. *Learning by Knowledge Sharing in Autonomous Intelligent Systems*. Lecture Notes in Artificial Intelligence, 4140: 128-137.
- [3] Zlotkin, G. and Rosenschein, J. 1996. *Mechanisms for Automated Negotiation in State Oriented Domains*. Artificial Intelligence, 86(2): 195-244.
- [4] Grimaldi, R. 1989. *Discrete and Combinatorial Mathematics*, 2nd ed., Addison-Wesley.
- [5] Zylberberg, A. 2005. *Probabilidad y Estadística*. Editorial Nueva Librería.