# Behaviors for Virtual Creatures by Constraint-based Adaptive Search

**Philippe Codognet**

University of Paris 6 and INRIA
LIP6, case 169, 8 rue du Capitaine Scott,
75 015 Paris, France
Philippe.Codognet@lip6.fr

**Abstract**

We present a high-level language for describing behaviors for autonomous agents in virtual worlds, together with an efficient run-time algorithm to implement those behaviors. Our approach is based on the notion of constraint goals, and we have designed a primitive set of constraints for navigation purposes. The resolution of goal constraints is done by a novel technique called adaptive search based on local search techniques within a constraint-based formalism.

## 1. Introduction

Our aim is to design a high-level language for describing behaviors for autonomous agents in virtual worlds, together with an efficient run-time algorithm to implement those behaviors. Virtual worlds are becoming increasingly popular due to the definition of standard formats for describing 3D scenes on the web (e.g. VRML97, Java3D and the future X3D) and, in the computer games community, because of the availability of tools to design new maps for popular 3D action games such Unreal, Half-life, or recently Deus Ex. Nevertheless, it is difficult to "populate" such worlds with virtual agents representing life-like creatures which could autonomously navigate and react to their changing environment, and also possibly interact with users. Path-planning problems are a key issue in the so-called "Game AI" domain, and have been tackled up to now with traditional AI techniques such as the classical $A^*$ algorithm [14] or more modern extensions [15], heavily relying on the complete knowledge of the complete virtual environment to define an optimal trajectory for computer-controlled characters. We are rather interested in this work in the definition of autonomous agents that are immersed in an undefined and changing environment and have to react in real-time to evolving configurations. For this purpose, we need to design a language in which such behaviors can be stated, and this language should be both simple, declarative and powerful in order to make it possible to express a great variety of operations. The basis of this declarative language is the notion of constraint, which can be used to represent goals that the agents are trying to achieve. The agent thus maintains a set of goal constraints and we define a simple but effective action selection mechanism that will select at each time-step the best action in order to reduce the discrepancy between his current state and the overall satisfaction of the goals. Thus, it is worth noticing that behaviors are stated in an implicit way (by giving a set of constraint goals) and not in an explicit way (e.g. by giving a precise trajectory), which makes it possible to reactively adapt the agent behavior to a changing real-time environment.

As a first application of this framework, we have considered agents with simple reactive behaviors or limited planning capabilities inspired from research in the field of Artificial Life and robotics [11] [12]. We consider the problem of navigation of such autonomous creatures as an optimization problem an propose to use an algorithm based on local search techniques, called adaptive search, to efficiently obtain optimal or near-optimal trajectories. More generally, our framework can be used as a motivation architecture for such virtual creatures, by considering variables for denoting internal states (e.g. energy, thirst, etc) and goal constraints for defining internal needs (e.g. the energy should stay above a certain level), routine behaviors (if the energy falls below some level, go for food), or external desired properties (e.g. stay away from predators).

## 2. Virtual Agents

We will focus in this paper on the design of simple reactive agents in 3D virtual worlds. We are indeed interested in designing autonomous creatures that can be embedded in various and unknown environments and nevertheless exhibit robust behaviors, in particular for navigation. The paradigm of reactive agents has emerged in AI and robotics in the mid 80's as a viable alternative to complex planning agents. Brooks' subsumption architecture and his seminal paper [5] has created, together with other researchers, a new domain called "behavior-based" or "situated" robotics. To use the definition given by one of the pioneers of this approach [2]: « A reactive robotic system couples

perception to action without the use of intervening abstract representation or time history ». Reactive agents are thus simple entities that receive percepts from their environment and can act on the environment by performing actions through some effectors/actuators, the simplest of which being to issue some motor command to effectively navigate in the external or virtual world. Reactive agents have no symbolic model of the world they live in, but rather use sensory-action control loops in order to perform tasks in a robust manner. It is worth noticing nevertheless that the framework proposed in that paper goes beyond the formalism of pure reactive agents, as creatures can maintain internal states and therefore "memorize" some particular aspects of their environments.

## 3. Biologically-inspired Creatures

In order to design autonomous, life-like creatures that can autonomously navigate in the 3D world, we propose some simple behaviors derived from biologically-inspired models of navigation. . There is currently a growing interest for such models both in the Artificial Life and the robotics community, and such models could obviously applied to virtual agents as well. The creatures will have to react to a changing environment and to avoid collision with moving obstacles. We will consider virtual creatures with limited intelligence building no cognitive map but using only the *taxon* system for route navigation, tracing a simple route towards a goal by avoiding obstacles. We can nevertheless consider some non-trivial navigation problems, as the creature does not know in advance the location of the goal but rather has a to explore the environment towards it, guided by a stimulus (e.g. light or smell) towards the goal (e.g. food). Our framework will also naturally cope with moving goals and obstacles and modify the behavior accordingly in real-time. We have investigated in [6] a high-level formalism based on a Timed Concurrent Constraint language for describing and implementing such behaviors, but we will present in this paper a simpler framework together with a new solving method for the run-time implementation of these techniques.

The two classical methods for stimulus-driven exploration in the biologically-inspired models of navigation are the temporal difference or spatial difference methods Temporal differences consists in considering a single sensor (e.g. the nose) and checking at every time-point the intensity of the stimulus. If the stimulus is increasing, then the agent continues in the same direction, otherwise the direction is changed randomly and so on so forth. This behavior is exemplified for instance by the chemotaxis (reaction to a chemical stimulus) of the Caenorabditis Elegans, a small soil nemapode. A more efficient strategy is possible by using the spatial differences method. It requires to have two identical sensing organs, placed at different slightly positions on the agent (e.g. the two ears). The basic idea is simply to favor, at any time-point, motion in the direction of the sensor that receive the most important stimulus This behavior gives very good results, and the creature goes most of the time directly towards the goal. When the goal is moved away by the user, the agent reacts instantly towards the new location.

## 4. A Constraint-based Language for Describing Behaviors

In computer graphics and animation systems, the most common formalism for representing behaviors is the finite state automaton (FSA). Many variants exist, such as the PatNets of [3], the Hierarchical FSA [9] or the parallel FSA [7]. Our approach rather considers that for representing complex life-like behaviors, one should not be restricted to some extended FSA formalism but indeed needs the power of a more advanced modeling language. In particular, we need the ability to handle internal variables, parametrized inputs, and dynamic representations of goals to be achieved. We propose here a quite simple framework, extended and abstracted from [6]. We will thus consider the formalism of CSP (Constraint Satisfaction Problems, see [10]) as a general modeling language. Constraints are used to state goals, or more exactly partial goals, that the agent has to achieve. The basic spatial constraints for autonomous navigation are :

| Constraint | Declarative meaning |
|---|---|
| *In(Region)* | Stay within the zone define by *Region* |
| *out(Region)* | Stay outside the zone define by *Region* |
| *go(Object)* | move towards the location of *Object* |
| *away(Object)* | move away from the location of *Object* |
| *Attraction(Stimulus)* | Move towards the source of *Stimulus* |
| *Repulsion(Stimulus)* | Move away from the source of *Stimulus* |

Observe that for the last two constraint goals, the agent does not know the location of the source of the stimulus, but it can only sense the amount of stimulus received at some location by one or more sensors, using either a *temporal difference* or a *spatial difference* method, see [11] for details. These declarative constraints will reduce to (or, for efficiency, be approximated by) some simple arithmetic constraints. For instance, the first constraint *In(Region)* for a circle *Region* will reduce to :

$$Agent.position - Region.Center < Region.Radius$$

and the third constraint *go(Object)* will reduce to

$$Agent.position - Object.position < 0.1 .$$

It is clear that a combination of such goal constraints could produce quite complex behaviors, e.g. that the agent should go towards some object, avoid all objects it perceives and stay away from some predefined regions. For instance a following behavior can be simply obtained as a combination of a *go* constraint (to move toward the followed agent) and an *out* constraint (to stay at a certain distance). In [5], a full logical language is proposed as "planning" vocabulary for encoding cognitive behaviors, but the large search space thus generated might be source of performance problems. On the contrary, the limited set of goal constraints defined here has been chosen because efficient methods to solve such goals can be designed. Indeed one can define, for each goal constraint, a predefined *repair* mechanism that will propose (in case the constraint is not satisfied) an action that could reduce the degree of violation of the constraint. For instance if an *attraction(Object)* constraint is violated by a agent *Creature*, then the "repair" action to be performed is a navigation step in the direction of *Object*, that is :

$$Creature.position \mathrel{+}= Creature.speed * || Object.position - Creature.position ||$$

Similarly, one can define a repair action for the obstacle avoidance constraint *out(Region)*, by considering that the agent should change its direction either slightly to the left or to the right until it can perform a side step without violating the *out* constraint (this could amount to perform a U turn). Let us now detail how to solve a combination of goal constraints by choosing the most appropriate repair action.

## 5. Adaptive Search

Heuristic (i.e. non-complete) methods have been used in Combinatorial Optimization for finding optimal or near-optimal solution since a few decades, in particular the family of Local Search methods [1] [13]. It has been used for problems like the Traveling Salesman Problem, scheduling, vehicle routing, cutting stock, etc. Classical instances of such methods are simulated annealing, Tabu search and genetic algorithms. They work by iterative improvement over an initial state and are thus *anytime* algorithms well-suited to a reactive environment. Consider an optimization problem with cost function which makes it possible to evaluate the quality of a given configuration (assignment of variables to current values) and a transition function that defines for each configuration a set of "neighbors ". The basic algorithm consists in starting from a random configuration, explore the neighborhood and then move to the best candidate. This process will continue until some satisfactory solution is found. To avoid being trapped in a local optimum, adequate mechanisms should be introduced, such as the adaptive memory of Tabu search or the cooling schedule of simulated annealing.

We can now detail our new heuristic method called Adaptive Search, derived from the GSAT, Walksat and Wsat(OIP) family of local search methods [13]. It will be used in our framework to perform behaviors, i.e. to select the adequate repair action if the goal constraints are not satisfied. The input of the method is a problem in CSP form, that is a set of variables and constraints over these variables. A constraint is simply a logical relation between several unknowns, these unknowns being variables that should take values in some specific domain of interest. A constraint thus restricts the degrees of freedom (possible values) the unknowns can take; it represents some partial information relating the objects of interest. Constraint Solving and Programming has proved to be very successful for Problem Solving and Combinatorial Optimization applications, by combining the declarativity of a high-level language with the efficiency of specialized algorithms for constraint solving, borrowing sometimes techniques from Operations Research and Numerical Analysis [10]. Several efficient constraint solving systems for finite domain constraints now exists, such as Ilog Solver on the commercial side and GNU-Prolog on the academic/freeware side. Although we will completely depart in adaptive search from the classical constraint solving techniques (i.e. Arc-Consistency and its extensions), we will take advantage of the formulation of a problem as a CSP. Such representation indeed makes it possible to structure the problem in terms of variables and constraints and to analyze more carefully the current configuration (assignment of variables to values in their domains) than a global cost function to be optimized, e.g. the number of constraints that are not satisfied. Accurate information can be collected by inspecting constraints (that typically involve only a subset of all the problem variables) and combining this information on variables (that typically appear in only a subset of all the problem constraints). Our method is not limited to any specific type of constraint, e.g. linear constraints as classical linear programming or [13].

For each constraint, we need to define an "error" function that will give an indication on how much the constraint is violated. For instance the "error" function associated to an arithmetic constraint $X - Y < C$ will be $max\ (0, |X-Y|-C)$. Adaptive search relies on iterative repair based on variables and constraint errors information, seeking to reduce the error on the worse variable so far. The basic idea is to compute the error function of each constraint, then combine for each variable the errors of all constraints in which it appear and then choose the variable with the maximal error as a "culprit" and thus change its value. In this second step we use the well-known *min-conflict* heuristic and select the value in the variable domain that has the best temptative

value, that is, the value for which the total error overall next configuration is minimal.
In order to prevent being trapped in local minima, the adaptive search method also include an adaptive memory module to prevent to be trapped by local minima (cf. Tabu Search) : each variable leading to a local minimum is marked and cannot be chosen for a few iterations. It is worth noticing that this frameworks naturally copes with Over-Constraint problems.

It is worth noticing that the adaptive search method is thus a generic framework parametrized by three components :

- A family of error functions for constraints (one for each type of constraint)
- An combination operation in order to aggregate, for a variable, the errors of all constraints in which it appears
- A cost function for evaluating configurations

In general the last component can be derived from the first two ones. Also, we could require the combination operation to be associative and commutative

Let us now detail this algorithm.

**Input :**

Problem given in CSP form :

- a set of variables *V={V1, V2,…, Vn}* with associated domains of values
- a set of constraints *C={C1, C2,…, Ck}* over *V*
- a combination function to aggregate constraint errors on variables
- a cost function to minimize (e.g. number of violated constraints)

**Output :**

Sequence of moves (i.e. modifications of the value of one of the variables) that will lead to a solution of the CSP (i.e. a configuration where all constraints are satisfied)

**Algorithm**

Start from a random assignment of variables in *V*

***Repeat***

1. ***Compute*** errors of all constraints in *C* and combine errors on each variable by considering for a given variable only the constraints on which it appears.
2. ***select*** variable X (not marked as tabu) with highest error and evaluate costs of possible moves from X
3. ***if*** no better move ***then*** mark X as tabu for a given number of iterations
   ***else*** select the best move *(min-conflict)* and change the value of X accordingly

***until*** a solution is found or the maximal number of iterations is reached

This method, although very simple, could nevertheless be quite efficient to solve complex combinatorial problems such as classical CSPs.

## 6. Examples

Let us now detail some classical CSP examples tackled by the adaptive search method.

### 6.1 God Saves the Queens

This puzzle consists in placing N queens on a NxN chessboards so that no two queens attach each other. It can be modeled by N variables (that is, one for each queen) with domains {1,2,…,N} (that is, considering that each queen should be placed on a different row) and 3 x $N^2$ disequation constraints stating that no pair of queens can ever be on the same column, up- or down-diagonal :

$$\textbf{For all } i,j \textbf{ in } \{1,2,\ldots,N\} : Q_i \neq Q_j \quad Q_i + i \neq Q_j + j \quad Q_i - i \neq Q_j - j$$

We can define the error function for disequation as follows, in the most simple way : 0 if the constraint is satisfied and 1 if the constraint is violated. The combination operation on variables is simply the addition, and the overall cost function is the sum of the costs of all constraints. Good results can be obtained with this instance of adaptive search, comparable to those of efficient constraint solving systems like Ilog Solver or GNU Prolog with the same modeling. Results on a 400 MHz Pentium-II PC of a simple Java-based implementation for 50x50, 100x100 and 200x200 chessboards are given in the table below (we give the average of 10 runs).

| size | CPU time |
|---|---|

| | |
|---|---|
| 50x50 | 0.1 sec |
| 100x100 | 2 sec |
| 200x200 | 20 sec |

However, even better results can be obtained with an optimized Java-based implementation of adaptive search which recomputes constraints errors only when necessary and randomly resets 10% of the variables (and not just one) when reaching a local minimun. Results on a 400 MHz Pentium-II PC are given in the table below :

| size | CPU time |
|---|---|
| 100x100 | 50 ms |
| 200x200 | 110 ms |
| 400x400 | 250 ms |
| 800x800 | 650 ms |

## 6.2 Magic to the Square

The magic square puzzle is much more complicated that N queens. It consists in placing on a NxN square all the numbers in $\{1,2,,\ldots,N^2\}$ such as the sum of the numbers in all rows, columns and diagonal are the same. It can therefore be modeled in CSP by considering $N^2$ variables with initial domains $\{1,2,,\ldots,N^2\}$ together with linear equation constraints and a global *all_different* constraint stating that all variables should have a different value. The constant value that should be the sum of all lines, columns and diagonals can be easily computed to be $N(N^2+1)/2$. Classical constraint solvers however perform quite poorly on this problem and neither Ilog Solver nor GNU Prolog could solve instances bigger that 10x10 square.
The instance of adaptive search for this problem is defined as follows. The error function of an equation $X_1 + X_2 + \ldots + X_k = b$ is defined as the value of $X_1 + X_2 + \ldots + X_k - b$. The combination operation is the absolute value of the sum of errors. The overall cost function is the addition of absolute values of the errors of all constraints The method will start by a random assignment of all $N^2$ numbers in $\{1,2,,\ldots,N^2\}$on the cells of the NxN square and consider as possible moves all swaps between two values.

The method can be best described by the execution snapshot depicted by Figure 1, which shows information computed on a 4x4 square. Numbers on the right of rows and diagonals, and below lines, denote the errors of the corresponding constraints The 4x4 table immediately on the right shows the combined error for each variable. The cell (3,2) with value 6 (in red on the square) has maximal error and is thus selected for swapping. We should now score all possible swaps with other numbers in the square; this is depicted in the table on the right, containing the cost value of the overall configuration for each swap. The cell (1,4) with value 15 give the best next configuration and is thus selected to perform a move. The selected move will thus reduce the cost of the current configuration from 57 to 33.

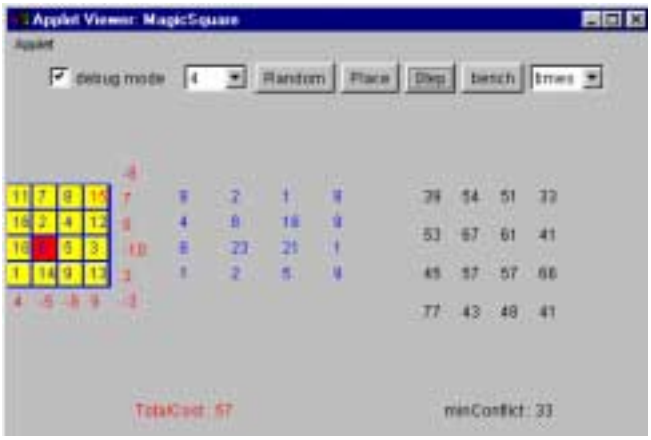


**Figure 1.** *Adaptive Search on the 4x4 magic square.*

Let us now detail the performances of this algorithm on bigger instances. Results on a 400 MHz pentiumII PC of a simple Java-based implementation for 6x6, 10x10, 16x16 and 20x20 squares are given in the table below (again showing the average of 10 runs).

| size | CPU time |
|---|---|
| 6 | 0.2 sec |
| 10 | 1.8 sec |
| 16 | 21 sec |
| 20 | 1 min 30 sec |

This results compares favorably with those obtained with the Localizer system which is based on Tabu search (personnal communication by Laurent Michel, one of the designer of Localizer) , and it is worth noticing that some choices in this adaptive search instance (e.g. the combination and global cost functions) could be much improved by careful tuning.

As defined above, this method does not perform any planning, as it only computes the move for the next time step out of all possible current moves. A simple extension would be to allow some limited planning capability by considering not only the immediate neighbors (i.e. nodes at distance 1) but all configurations on all paths up to some predefined distance (e.g. all nodes within at distance less or equal to 10), and then choose to move to the neighbor in the direction of the most promising node, as pioneered by variable-depth local search some decades ago, see [1] for details. Therefore the method can plan for the best trajectory in some limited time-window. However real-time

considerations might prevent to extend this time-window more than a few steps forwards.

## 7. Application to Path-finding and Stimulus-driven Navigation

It is worth noticing that the adaptive search method applied to agent navigation is also close to real time search algorithms such as those of [15], but without paying the price for heavy data structures. Let us consider a simple example where the creature has to perform collision avoidance. The behavior consisting in going to a particular goal object while avoiding obstacles is simply described in the following way :

$$\text{go}(goal) , \text{out}(obstacle1 , 0.1) , \dots , \text{out}(obstacleN , 0.1)$$

where $obstacle_i$ are the objects to avoid. It is worth noticing that the creature is reactive to the changes in the environment in real-time and will thus keep avoiding objects if they move and further obstruct its trajectory, which will be updated accordingly. All the constraints are (re-) checked in real-time at each iteration of the adaptive search algorithm and thus the trajectory is adaptive at each time-step.

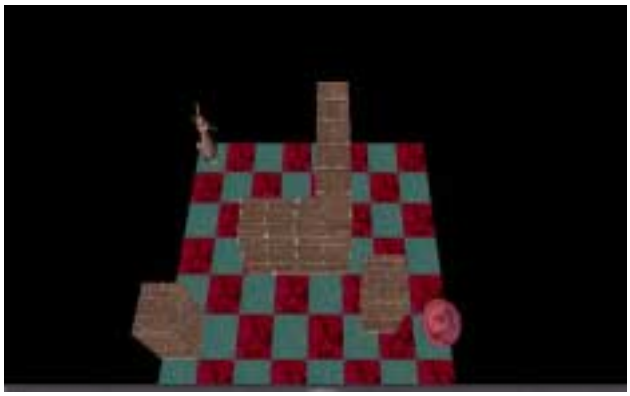

**Figure 2.** *Initial setting for simple obstacle avoidance*

Figure 2 describes the initial position of the agent (upper left) and of the goal object (bottom right), together with the position of the brick-textured obstacles. The trajectory of the creature is depicted as a while line on Figure 3. It could be observed that the trajectory is not optimal (the creature could have anticipated the first obstacle and turned right earlier) but not far from the optimal path and life-like anyway. A better trajectory could have been achieved by using limited planing with variable-depth exploration of the neighborhood, as explained earlier.

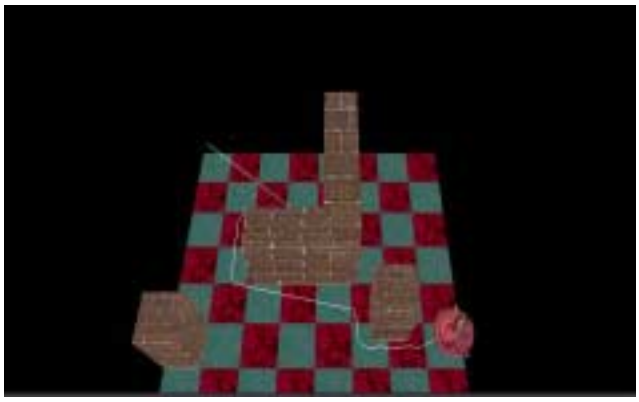

**Figure 3.** *Trajectory for simple obstacle avoidance*

Similar behaviors can be obtained with stimulus-driven search, by simply replacing the *go* constraint by an attraction constraint on the given stimulus :

$$\text{attraction}(stimulus) ,$$
$$\text{out}(obstacle1 , 0.1) , \dots , \text{out}(obstacleN , 0.1)$$

It appears than in general only two sensors (that is, checking the intensity of the stimulus at only two points in front of the creature) are enough to obtain a good trajectory, which is consistent with biologically-inspired models.

## 8. Conclusion

We have proposed a constraint-based language that makes it possible to express in an implicit way reactive behaviors. Behaviors are expressed as sets of goal constraints that have to be satisfied by the agent. We have defined the adaptive search method which is used as action selection mechanism in order to choose among all the "repair" actions (for goal constraints) the most promising one. We have implemented a prototype version of this framework by coupling a VRML browser and a JavaScript-based adaptive search algorithm and experimented some simple biologically-inspired behaviors for virtual creatures, such as stimulus-driven search together with obstacle avoidance in a reactive context (both the source of the stimulus and the obstacles can be moved in real-time). The method is indeed able to achieve life-like, near-optimal trajectories. A more robust, full-fledge system is currently under implementation in the Java3D environment . Perspectives include the extension of the current adaptive search framework to include limited look-ahead planning and the development of a longer term action-planning module based on constraint solving techniques.

## Acknowledgements

This paper is a revised and extended version of a paper presented at the VSMM2000 conference, 6th International Conference on Virtual Systems and Multimedia, Gifu, Japan, October 2000.

## References

1. E. Aarts and J. Lenstra (Eds). *Local Search in Combinatorial Optimization*, Wiley, 1997.
2. R. C. Arkin. *Behavior-based robotics*, MIT Press 1998.
3. N. Badler, C. Philips and B. Webber. *Simulating Humans : Computer Graphics animation and control*, Oxford University Press 1993.
4. G. Berry and G. Gonthier. The Esterel Programming Language : Design, Semantics and Implementation, *Science of Computer Programming*, vol. 19 no. 2, 1992
5. Rodney Brooks. A robust layered control system for mobile robots, *IEEE Journal of Robotics and Automation*, 1986 (2).
6. P. Codognet. Declarative Behaviors for Virtual Creatures. *Proc. ICAT'99, 8th International Conference on Augmented Reality and Tele-existence*, Tokyo, Japan, IOS Press 1999.
7. S. Donikian. Multilevel Modeling of Virtual Urban Environments for Behavioural Animation. *Proc. Computer Animation 97*, Geneva, Switzerland, IEEE Press 1997.
8. J. Funge, X. Tu and D. Terzopoulos. Cognitive modeling : Knowledge, reasoning and planning for intelligent characters. *In proceedings of SIGGRAPH'99*, ACM Press 1999.
9. Y. Koga, C. Becker, M. Svihura, and D. Zhu. On intelligent Digital Actors. *Proc. Imagina 98*, Monaco, 1998.
10. V. Saraswat, P. Van Hentenryck, P. Codognet *et al*. Constraint Programming, *ACM Computing Surveys* vol. 28 no. 4, December 1996.
11. N. Schmajuck (Ed.). Special issue on Biologically-inspired models of Navigation, *Adaptive Behavior*, vol. 6, no. 3/4, Winter/Spring 98.
12. O. Trullier, and J-A. Meyer. Biomimetic Navigation Models and Strategies in Animats. *AI Communications* Vol.10, no. 2, 1997.
13. J. P. Walser. *Integer Optimization by Local Search : A Domain-Independent Approach*, LNAI 1637, Springer Verlag 1999.
14. S. Woodcock. Game AI : the state of the industry. *Game developer*, vol. 7 no. 8, August 2000.
15. M. Yokoo and T. Ishida, Search Algorithms for Agents, In: *Multiagent Systems*, G. Weiss (ed.), MIT Press 1999.