RSPSA: Enhanced Parameter Optimization in Games

Levente Kocsis[1], Csaba Szepesvári[1], and Mark H.M. Winands[2]

[1] MTA Sztaki, Budapest, Hungary
{kocsis, szcsaba}@sztaki.hu
[2] Institute for Knowledge and Agent Technology,
MICC, Universiteit Maastricht, Maastricht, The Netherlands
m.winands@micc.unimaas.nl

Abstract. Most game programs have a large number of parameters that are crucial for their performance. Tuning these parameters by hand is rather difficult. Therefore automatic optimization algorithms in game programs are interesting research domains. However, successful applications are only known for parameters that belong to certain components (e.g., evaluation-function parameters). The SPSA (Simultaneous Perturbation Stochastic Approximation) algorithm is an attractive choice for optimizing any kind of parameters of a game program, both for its generality and its simplicity. Its disadvantage is that it can be very slow.
In this article we propose several methods to speed up SPSA, in particular, the combination with RPROP, using common random numbers, antithetic variables, and averaging. We test the resulting algorithm for tuning various types of parameters in two domains, Poker and LOA. From the experimental study, we may conclude that using SPSA is a viable approach for optimization in game programs, in particular if no good alternative exists for the types of parameters considered.

1 Introduction

Any reasonable game program has several hundreds if not thousands of parameters. These parameters belong to various components of the program, such as the evaluation function or the search algorithm. While it is possible to make educated guesses about "good" values of certain parameters, hand-tuning the parameters is a difficult and time-consuming task. An alternative approach is to find the "right" values by means of an automated procedure.

The use of parameter optimization methods for the performance tuning of game programs is difficult by the fact that the objective function is rarely available analytically. Therefore, methods that rely on the availability of an analytic expression for the gradient cannot be used. However, there exist several ways to tune parameters despite the lack of an analytic gradient. An important class of such algorithms is represented by temporal-difference (TD) methods that have been used successfully in tuning evaluation-function parameters [14]. Obviously, any general-purpose (gradient-free) global search method can be used for parameter optimization in games. Just to mention a few examples, in [3] genetic

algorithms were used to evolve a neural network to play checkers, whilst in [2] an algorithm similar to the Finite-Difference Stochastic Approximations (FDSA) algorithm was used successfully for tuning the search-extension parameters of CRAFTY. Nevertheless, we believe that automatic tuning of parameters remains a largely unexplored area of game programming.

In this article we investigate the use of SPSA (Simultaneous Perturbation Stochastic Approximation), a stochastic hill-climbing search algorithm for tuning the parameters of game programs. Since optimization algorithms typically exhibit difficulties when the objective function (performance measure) is observed in heavy noise, for one test domain we choose a non-deterministic game, namely Omaha Hi-Lo Poker, one of the most complex poker variants. For Texas Hold'em Poker several years of research has led to a series of strong programs: POKI, PSOPTI, and VEXBOT [1]. Our program, MCRAISE, borrows several ideas from the above mentioned programs. The name of the program originates from the use of Monte-Carlo simulations and the program's aggressive style. In the second test domain, LOA, we use MIA, winner of the 8^{th} and 9^{th} Computer Olympiad.

The article is organized as follows. Section 2 describes the RSPSA algorithm that combines SPSA and RPROP. In Sect. 3, three ways to enhance the performance of RSPSA are proposed together with a discussion of the various trade-offs involved, supported by analytic arguments. Next, in Sect. 4 the test domains and the respective programs are described. Experiments with RSPSA in these domains are given in Sect. 5. Finally, we draw our conclusions in Sect. 6.

2 The RSPSA Algorithm

Below we start describing the basic setup (2.1) of the RSPSA algorithm. Then we provide details on the supporting algorithms SPSA (2.2) and RPROP (2.3). Finally in 2.4 we outline the RSPSA algorithm.

2.1 Basic Setup

Consider the task of finding a maximizer $\theta^* \in \mathbb{R}^d$ of some real-valued function f, i.e., find $\theta^* = \text{argmax}_\theta f(\theta)$. In our case f may measure the performance of a player in some environment (e.g., against a fixed set of opponents), or it may represent an auxiliary performance index of interest that is used internally in the algorithm in such a way that a higher value of it might ultimately yield better play. In any case, θ represents some parameters of the game-playing program.

We assume that the algorithm of which the task is to tune the parameters θ can query the value of f at any point θ, but the value received by the algorithm will be corrupted by noise. The noise in the evaluation of f can originate from randomized decisions of the players or from the randomness of the environment. In a card game for instance the cards represent a substantial source of randomness in the outcomes of rounds. We shall assume that the value observed in the t-th step of the algorithm, when the simulation is run with parameter θ_t, is given

by $f(\theta_t; Y_t)$ where Y_t is some random variable such that the expected value of $f(\theta_t; Y_t)$ conditioned on θ_t and given all past information equals to $f(\theta_t)$:

$$f(\theta_t) = \mathbb{E}\left[f(\theta_t; Y_t) \,|\, \theta_t, \mathcal{F}_t\right], \tag{1}$$

where $\mathcal{F}_t$ is the sigma-field generated by $Y_0, Y_1, \ldots, Y_{t-1}$ and $\theta_0, \theta_1, \ldots, \theta_{t-1}$. Stochastic gradient ascent algorithms work by changing the parameter θ in a gradual manner so as to increase the value of f on average:

$$\theta_{t+1} = \theta_t + \alpha_t \hat{g}_t(\theta_t). \tag{2}$$

Here θ_t is the estimate of θ^* in the t-th iteration (time step), $\alpha_t \geq 0$ is a learning rate parameter that governs the size of the changes to the parameters and $\hat{g}_t(\theta_t)$ is some approximation to the gradient of f such that the expected value of $\hat{g}_t(\theta_t)$ given past data is equal to the gradient $g(\theta) = \partial f(\theta)/\partial\theta$ of f and $(\hat{g}_t(\theta_t) - g(\theta))$ has finite second moments.

2.2 SPSA

When f is not available analytically then one must resort to some approximation of the gradient in order to use gradient ascent. One such approximation was introduced with the SPSA algorithm in [12]:

$$\hat{g}_{ti}(\theta_t) = \frac{f(\theta_t + c_t\Delta_t; Y_t^+) - f(\theta_t - c_t\Delta_t; Y_t^-)}{2c_t\Delta_{ti}}. \tag{3}$$

Here $\hat{g}_{ti}(\theta_t)$ is the estimate of the i-th component of the gradient, Δ_{ti}, Y_t^+, and Y_t^- are random variables: Y_t^+ and Y_t^- are meant to represent the sources of randomness of the evaluation of f, whilst $\Delta_{t\cdot}$ is a perturbation vector to be chosen by the user. Note that the numerator of this expression does not depend on the index i and therefore evaluating Eq. 3 requires only two (randomized) measurements of the function f. Still, SPSA provides a good approximation to the gradient: Under the conditions that *(i)* the random perturbations Δ_t are independent of the past of the process, *(ii)* for any fixed t, $\{\Delta_{ti}\}_i$ is an i.i.d. sequence[3], *(iii)* the distribution of Δ_{ti} is symmetric around zero, *(iv)* $|\Delta_{ti}|$ is bounded with probability one, and *(v)* $\mathbb{E}\left[\Delta_{ti}^{-1}\right]$ is finite, and assuming that f is sufficiently smooth, it can be shown that the bias of estimating that gradient $g(\theta_t)$ by $\hat{g}_t(\theta_t)$ is of the order $O(c_t^2)$. Further, the associated gradient ascent procedure can be shown to converge to a local optima of f with probability one [12].

A simple way to satisfy the conditions on Δ_t is to choose its components to be independent ± 1-valued Bernoulli distributed random variables with each outcome occurring with probability 1/2. One particularly appealing property of SPSA is that it might need d times less measurements than the classical FDSA procedure and still achieve the same asymptotic statistical accuracy (see, e.g., [12]). FDSA works by evaluating f at $\theta_t \pm c_t e_i$ and forming the appropriate

[3] "i.i.d." is the shorthand of "independent, identically distributed".

differences – thus it requires $2d$ evaluations. For a more thorough discussion of SPSA, its variants, and its relation to other methods we refer to [12, 13].

SPSA, like other stochastic approximation algorithms has quite a few tunable parameters. These are the gain sequences α_t, c_t and the distribution of the perturbations Δ_t. When function evaluation is expensive, as is often the case in games, small sample behavior of the algorithm becomes important. In that case the proper tuning of the parameters becomes critical.

In practice, the learning rate α_t and the gain sequence c_t are often kept at a fixed value. Further, in all previous works on SPSA known to us it was assumed that the perturbations Δ_{ti}, $i = 1, \ldots, d$, have the same distribution. When different dimensions have different scales (which we believe is a very common phenomenon in practice) then it does not make too much sense to use the same scales for all the dimensions. The issue is intimately related to the issue of scaling the gradient addressed also by second and higher-order methods. These methods work by utilising information about higher order derivatives of the objective function (see, e.g., [4, 13]). In general, these methods achieve a higher asymptotic rate of convergence, but, as discussed, e.g., in [15], their practical value might be limited in the small sample size case.

2.3 RPROP

The RPROP ("resilient backpropagation") algorithm [11] and its variants are amongst the best performing first-order batch neural network gradient training methods and as such represent a viable alternative to higher-order methods.[4] In practice RPROP methods were found to be very fast and accurate, robust to the choices of their parameters, scale well with the number of weights. Further, RPROP is easy to implement, it is not sensitive to numerical errors and since the algorithm is dependent only on the sign of the partial derivatives of the objective function,[5] it is thought to be suitable for applications where the gradient is numerically estimated and/or is noisy.

A particularly successful variant is the iRPROP$^-$algorithm [5]. The update equations of iRPROP$^-$ for maximising a function $f = f(\theta)$ are as follows:

$$\theta_{t+1,i} = \theta_{t,i} + \mathrm{sign}(g_{ti})\delta_{ti}, \quad t = 1, 2, \ldots; i = 1, 2, \ldots, d\,. \tag{4}$$

Here $\delta_{ti} \geq 0$ is the step size for the i-th component and $g_{t\cdot}$ is a gradient-like quantity:

$$g_{ti} = \mathbb{I}(g_{t-1,i} f_i'(\theta_t) \geq 0) f_i'(\theta_t)\,, \tag{5}$$

i.e., g_{ti} equals the i-th partial derivative of f at θ except when a sign reversal is observed between the current and the previous partial derivative, in which case g_{ti} is set to zero.

[4] For a recent empirical comparison of RPROP and its variants with alternative, gradient optimization methods such as BFGS, CG and others see, e.g., [5].

[5] RPROP, though it was worked out for the training of neural networks, is applicable in any optimization problem where the gradient can be computed or approximated.

The individual step-sizes δ_{ti} are updated in an iterative manner based on the sign of the product $p_{t,i} = g_{t-1,i} f'_i(\theta_t)$:

$$\eta_{ti} = \mathbb{I}(p_{t,i} > 0)\eta^+ + \mathbb{I}(p_{t,i} < 0)\eta^- + \mathbb{I}(p_{t,i} = 0)\,, \tag{6}$$

$$\delta_{ti} = P_{[\delta^-,\delta^+]}\left(\eta_{ti}\delta_{t-1,i}\right)\,, \tag{7}$$

where $0 < \eta^- < 1 < \eta^+$, $0 < \delta^- < \delta^+$, $P_{[a,b]}$ clamps its argument to the interval $[a, b]$, and $\mathbb{I}(\cdot)$ is a $\{0, 1\}$-valued function working on Boolean values and $\mathbb{I}(\mathcal{L}) = 1$ if and only if $\mathcal{L}$ is true, and $\mathbb{I}(\mathcal{L}) = 0$, otherwise.

2.4 RSPSA

Given the success of RPROP, we propose a combination of SPSA and RPROP (in particular, a combination with iRprop$^-$). We call the resulting combined algorithm RSPSA ("resilient SPSA"). The algorithm works by replacing $f'_i(\theta_t)$ in Eq. 5 with its noisy estimates $\hat{g}_{t,i}(\theta_t)$. Further, the scales of the perturbation vector Δ_{ti} are coupled to the scale of the step sizes of δ_{ti}.

Before motivating the coupling let us make a few observations on the expected behavior of RSPSA. Since iRprop$^-$ depends on the gradient only through the sign of it, it is expected that if the sign of $\hat{g}_{t,i}(\theta_t)$ coincides with that of $f'_i(\theta_t)$ then the performance of RSPSA will be close to that of iRprop$^-$. This can be backed up by the following simple argument. Assuming that $|f'_i(\theta)| > \varepsilon$, applying Markov's inequality, we obtain that

$$\mathbb{P}(\mathrm{sign}(\hat{g}_{t,i}(\theta)) \neq \mathrm{sign}(f'_i(\theta))) \leq \mathbb{P}(|\hat{g}_{t,i}(\theta)) - f'_i(\theta)| \geq \varepsilon) \leq \frac{M_{t,i}}{\varepsilon^2}\,, \tag{8}$$

where $M_{t,i} = \mathbb{E}\left[(\hat{g}_{t,i}(\theta) - f'_i(\theta))^2 | \mathcal{F}_t\right]$ denotes the mean square error of the approximation of $f'_i(\theta)$ by $\hat{g}_{t,i}(\theta)$, conditioned on past observations. In fact, this error can be shown to be composed of a bias term dependent only on f, θ and c, and a variance term dependent on the random quantities in $\hat{g}_{t,i}(\theta)$. Hence, it is important to make the variance of the estimates small.

Now, let us turn to the idea of coupling the scales of the perturbation vectors to the step sizes of iRprop$^-$. This idea can be motivated as follows. On "flat areas" of the objective function, where the sign of the partial derivatives of the objective function is constant and where the absolute value of these partial derivatives is small, a perturbation's magnitude along the corresponding axis should be large or the observation noise will dominate the computed finite differences. In contrast, in "bumpy areas" where the sign of a partial derivative changes at smaller scales, smaller perturbations that fit the "scale" of desired changes can be expected to perform better. Since the step-size parameters of RPROP are larger in flat areas and are smaller in bumpy areas, it is natural to couple the perturbation parameters of SPSA to the step-size parameters of RPROP. A simple way to accomplish this is to let $\Delta_{ti} = \rho\,\delta_{ti}$, where ρ is some positive constant, to be chosen by the user.

3 Increasing Efficiency

In this section we describe three methods that can be used to increase the efficiency of RSPSA. The first method, known as the "Method of Common Random Numbers", was proposed earlier to speed up SPSA [9, 6]. The second method, averaging, was proposed as early as in [12]. However, when averaging is used together with the method of common random numbers a new trade-off arises. By means of a formal analysis this trade-off is identified and resolved here for the first time. To the best of our knowledge, the third method, the use of antithetic variables has not been suggested earlier to be used with SPSA. All these methods aim at reducing the variance of the estimates of the gradient, which as noted previously should yield better performance. In this section we will drop the time index t in order to simplify the notation.

3.1 Common Random Numbers

In SPSA (and therefore also in RSPSA) the estimate of the gradient relies on differences of the form $f(\theta + c\Delta; Y^+) - f(\theta - c\Delta; Y^-)$. Denoting by F_i^+ the first term and by F_i^- the second term, elementary probability calculus gives $\mathrm{Var}\left(F_i^+ - F_i^-\right) = \mathrm{Var}\left(F_i^+\right) + \mathrm{Var}\left(F_i^-\right) - 2\mathrm{Cov}\left(F_i^+, F_i^-\right)$. Thus the variance of the estimate of the gradient can be decreased by introducing some correlation between F_i^+ and F_i^-, provided that this does not increase the variance of F_i^+ and F_i^-. The reason is that by our assumptions Y^+ and Y^- are independent and thus $\mathrm{Cov}\left(F_i^+, F_i^-\right) = 0$. Now, if $F_i^\pm$ is redefined to depend on the *same* random value Y (i.e., $F_i^\pm = f(\theta \pm c\Delta; Y)$) then the variance of $F_i^+ - F_i^-$ will decrease when $\mathrm{Cov}\left(f(\theta + c\Delta; Y), f(\theta - c\Delta; Y)\right) > 0$. The larger this covariance is the larger the decrease of the variance of the estimate of the gradient will be.

When f is the performance of a game program obtained by means of a simulation that uses pseudo-random numbers then using the same random series Y can be accomplished by using identical initial seeds when computing the values of f at $\theta + c\Delta$ and $\theta - c\Delta$.

3.2 Using Averaging to Improve Efficiency

A second method to reduce the variance of the estimate of the gradient is to average many independent estimates of it. However, the resulting variance reduction is not for free, since evaluating $f(\theta; Y)$ can be extremely CPU-intensive, as mentioned earlier. To study this trade-off let us define

$$\hat{g}_{q,i}(\theta) = \frac{1}{q} \sum_{j=1}^{q} \frac{f(\theta + c\Delta; Y_j) - f(\theta - c\Delta; Y_j)}{2c\,\Delta_i}\,, \tag{9}$$

where according to the suggestion of the previous section we use the same set of random number to evaluate f both at $\theta + c\Delta$ and $\theta - c\Delta$. Further, let $\hat{g}_{r,q,i}(\theta)$ be the average of r independent samples $\{\hat{g}_{q,i}^{(j)}(\theta)\}_{j=1,\ldots,r}$ of $\hat{g}_{q,i}(\theta)$. By the Strong

Law of Large Numbers $\hat{g}_{r,q,i}(\theta)$ converges to $f_i'(\theta) + O(c^2)$ as $q, r \to +\infty$ (i.e., its ultimate bias is of the order $O(c^2)$). It follows then that increasing $min(r, q)$ above the value where the bias term becomes dominating does not improve the finite sample performance. This is because increasing p decreases the frequency of updates to the parameters.[6]

In order to gain further insight into how to choose q and r, let us consider the mean squared error (MSE) of approximating the i-th component of the gradient by $\hat{g}_{r,q,i}$: $M_{r,q,i} = \mathbb{E}\left[(\hat{g}_{r,q,i}(\theta) - f_i'(\theta))^2\right]$. By some lengthy calculations, the following expression can be derived for $M_{r,q,1}$:[7]

$$M_{r,q,1} = \frac{1}{r}\mathbb{E}\left[\Delta_1^2\right]\mathbb{E}\left[1/\Delta_1^2\right]\sum_{j=2}^{d}\left\{\left(1-\frac{1}{q}\right)\mathbb{E}\left[f_j'(\theta;Y_1)^2\right] + \frac{1}{q}\mathbb{E}\left[f_j'(\theta;Y_1)\right]^2\right\} + \frac{1}{rq}\mathbb{E}\left[(f_1'(\theta;Y_1) - f_1'(\theta))^2\right] + O(c^2)\,. \tag{10}$$

Here $f_i'(\theta;Y)$ is the partial derivative of $f(\theta;Y)$ w.r.t. θ_i: $f_i'(\theta;Y) = \frac{\partial f(\theta;Y)}{\partial \theta_i}$.

It follows from Eq. 10 that for a fixed budget of $p = qr$ function evaluations the smallest MSE is achieved by taking $q = 1$ and $r = p$ (disregarding the $O(c^2)$ bias term which we assume to be "small" as compared to the other terms).

Now the issue of choosing p can be answered as follows. Under mild conditions on f and Y (ensuring that the expectation and the partial derivative operators can be exchanged), $\sum_{j=2}^{d}\mathbb{E}\left[f_j'(\theta;Y_1)\right]^2 = \sum_{j=2}^{d} f_j'(\theta)^2$. Hence, in this case with the choices $q = 1$, $r = p$, $M_{p,1,1}$ becomes equal to

$$\frac{1}{p}\left\{\mathbb{E}\left[\Delta_1^2\right]\mathbb{E}\left[1/\Delta_1^2\right]\sum_{j=2}^{d} f_j'(\theta)^2 + \mathbb{E}\left[(f_1'(\theta;Y_1) - f_1'(\theta))^2\right]\right\} + O(c^2)\,, \tag{11}$$

which is composed of two terms in addition to the bias term $O(c^2)$: the term $\sum_{j=2}^{d} f_j'(\theta)^2$ represents the contribution of the "cross-talk" of the derivatives of f to the estimation of the gradient, whilst the second term, $\mathbb{E}\left[(f_1'(\theta;Y_1) - f_1'(\theta))^2\right]$ gives the MSE of approximating $f_1'(\theta)$ with $f_1'(\theta;Y_1)$ (which is equal to the variance of $f_1'(\theta;Y_1)$ in this case). The first term can be large when θ is far from a stationary point of f, whilst the size of the second term depends on the amount of noise in the evaluations of f. When the magnitude of these two terms is larger than that of the bias term $O(c^2)$ then increasing p will increase the efficiency of the procedure, at least initially.

[6] In [12] it is shown that using decreasing gains $\alpha_t = a/t^\alpha$ and $c_t = c/t^\gamma$ with $\beta = \alpha - 2\gamma > 0$, $0 < \alpha \leq 1$, $0 < \gamma$, the optimal choice of p is governed by an equation of the form $p^{\beta-1}A + p^\beta B$, where $A, B > 0$ are some (unknown) system parameters. This equation has a unique minimum at $p = (1-\beta)A/(\beta B)$, however, since A, B are unknown parameters this result has limited practical value besides giving a hint about the nature of the trade-off in the selection of p.

[7] Without the loss of generality we consider the case $i = 1$.

3.3 Antithetic Variables

In Sect. 3.1 we have advocated the introduction of correlation between the two terms of a difference to reduce its variance. The same idea can be used to reduce the variance of averages: Let $U_1, U_2, \ldots, U_n$ be i.i.d. random variables with common expected value I. Then the variance of $I_n = 1/n \sum_{i=1}^{n} U_i$ is $1/n\mathrm{Var}\left(U_1\right)$. Now, assume that n is even, say $n = 2k$ and consider estimating I by

$$I_n^A = \frac{1}{k} \sum_{i=1}^{k} \frac{U_i^+ + U_i^-}{2} \,, \tag{12}$$

where now it is assumed that $\{U_1^+, \ldots, U_k^+\}$ are i.i.d., just like $\{U_1^-, \ldots, U_k^-\}$, $\mathbb{E}\left[U_i^+\right] = \mathbb{E}\left[U_i^-\right] = I$. Then $\mathbb{E}\left[I_n^A\right] = I$ and

$$\mathrm{Var}\left(I_n^A\right) = (1/k)\,\mathrm{Var}\left((U_1^+ + U_1^-)/2\right) \,. \tag{13}$$

Using the elementary identity

$$\mathrm{Var}\left((U_1^+ + U_1^-)/2\right) = 1/4\left(\mathrm{Var}\left(U_1^+\right) + \mathrm{Var}\left(U_1^-\right) + 2\mathrm{Cov}\left(U_1^+, U_1^-\right)\right), \tag{14}$$

we get that if $\mathrm{Var}\left(U_1^+\right) + \mathrm{Var}\left(U_1^-\right) \leq 2\mathrm{Var}\left(U_i\right)$ and $\mathrm{Cov}\left(U_1^+, U_1^-\right) \leq 0$ then $\mathrm{Var}\left(I_n^A\right) \leq \mathrm{Var}\left(I_n\right)$. One way to achieve this is to let U_i^+, U_i^- be *antithetic* random variables: U_i^+ and U_i^- are called antithetic if their distributions are the same but $\mathrm{Cov}\left(U_i^+, U_i^-\right) < 0$.

How can we introduce antithetic variables in parameter optimization of game programs? Consider the problem of optimizing the performance of a player in a non-deterministic game. Let us collect all random choices external to the players into a random variable Y and let $f(Y; W)$ be the performance of the player in the game. Here W represents the random choices made by the players (f is a deterministic function of its arguments). For instance, in poker Y can be chosen to be the cards in the deck at the beginning of the play after shuffling. The idea is to manipulate the random variables Y in the simulations by introducing a "mirrored" version, Y', of it such that $f(Y; W)$ and $f(Y'; W')$ become antithetic. Here W' represents the player's choices in response to Y' (it is assumed that different random numbers are used when computing W and W').

The influence of the random choices Y on the outcome of the game is often strong. By this we mean that the value of $f(Y; W)$ is largely determined by the value of Y. For instance, it may happen in poker that one player gets a strong hand, whilst the other gets a weak one. Assuming two players, a natural way to mitigate the influence of Y is to reverse the hands of the players: the hand of the first player becomes that of the second and vice versa. Denoting the cards in this new scenario by Y', it is expected that $\mathrm{Cov}\left(f(Y; W), f(Y'; W')\right) < 0$. Since the distribution of Y and Y' are identical (the mapping between Y and Y' is a bijection), the same holds for the distributions of $f(Y; W)$ and $f(Y'; W')$. When the random choices Y influence the outcome of the game strongly then we often find that $f(Y; W) \approx -f(Y'; W')$. When this is the case then $\mathrm{Cov}\left(f(Y; W), f(Y'; W')\right) \approx -\mathrm{Var}\left(f(Y; W)\right)$ and thus $f(Y; W)$ and

$f(Y';W')$ are "perfectly" antithetic and thus Var $\left(I_n^A\right) \approx 0$. Of course, $f(Y;W) = -f(Y';W')$ will never hold and thus the variance of I_n^A will not be eliminated entirely – but the above argument shows that it can be reduced to a large extent.

This method can be used in the estimation of the gradient (and also when the performance of the players is evaluated). Combined with the previous methods we obtain

$$\hat{g}_{p',i}(\theta) = \frac{1}{4cp'} \sum_{j=1}^{p'} \frac{1}{\Delta_i^{(j)}} \Big((f(\theta + c\Delta^{(j)}; Y) + f(\theta + c\Delta^{(j)}; Y')) - (f(\theta - c\Delta^{(j)}; Y) + f(\theta - c\Delta^{(j)}; Y')) \Big), \tag{15}$$

where $\Delta^{(1)}, \ldots, \Delta^{(p')}$ are i.i.d. random variables. In our experiments (see Sect. 5.1) we observed that to achieve the same accuracy in evaluating the performance of a player we could use up to 4 times less samples when antithetic variables were used. We expect similar speed-ups in other games where external randomness influences the outcome of the game strongly.

4 Test Domains

In this section we describe the two test domains, Omaha Hi-Lo Poker and Lines of Action. Together with the game-playing programs they are used in the experiments.

4.1 Omaha Hi-Lo Poker

The rules. Omaha Hi-Lo Poker is a card game played by two to ten players. At the start each player is dealt four private cards, and at later stages five community cards are dealt face up (three after the first betting round, and one after the second betting round and after the third betting round). In a betting round, the player on turn has three options: *fold*, *check/call*, or *bet/raise*. After the last betting round, the pot is split amongst the players depending on the strength of their cards. The pot is halved into a high side and a low side. For each side, players must form a hand consisting of two private cards and three community cards. The high side is won according to the usual poker hand ranking rules. For the low side, a hand with five cards with different numerical values from Ace to eight has to be constructed. The winning low hand is the one with the lowest high card.

Estimating the Expected Proportional Payoff. It is essential for a poker player is to estimate his winning chances, or more precisely to predict how much share one will get from the pot. Our program, McRaise, uses the following calculations to derive an estimate of the expected proportional payoff.

N random card configurations, cc, are generated, each consisting of the opponent hands $h_{opp}(cc)$ and the community cards that still have to be dealt. Then,

given the betting history *history* of the current game, the expected payoff as expressed as a proportion of the actual pot size (we call this quantity the expected proportional payoff or EPP) is approximated by

$$p_{win} = \frac{1}{N} \sum_{cc} win(cc) \prod_{opp} \frac{p(h_{opp}(cc)|history)}{p(h_{opp}(c))} \,. \tag{16}$$

where $win(cc)$ is the percentage of the pot won for a given card configuration cc and $p(h_{opp}(cc)|history)$ is the probability of cc given the observed history of betting actions. Now, using Bayes' rule, we obtain

$$\frac{p(h_{opp}(cc)|history)}{p(h_{opp}(cc))} \propto p(history|h_{opp}(cc)) \,, \tag{17}$$

where the omission of $p(history)$ is compensated by changing the calculation of p_{win} by normalising the individual weights of $w(cc) = p(history|h_{opp}(cc))$ so as they sum to 1, i.e., p_{win} is estimated by means of weighted importance sampling.

The probability of a betting sequence given the hole cards, $p(history|h_{opp}(cc))$, is computed using the probability of a certain betting action given the game state, $p(a|h_{opp}(cc))$, which is the core of the opponent model. If we would assume independence among the actions of an opponent, then $p(history|h_{opp}(cc))$ would come down to a product over the individual actions. This is obviously not the case. A simple way to include the correlation among the betting actions inside a round is given by the following equation:

$$p(history|h_{opp}(cc)) = \prod_{rnd} \frac{\sum_{a \in history_{opp,rnd}} \frac{p(a|h_{opp}(cc))}{p(a)}}{na_{opp,rnd}} \,, \tag{18}$$

where $na_{opp,rnd}$ is the number of actions of an opponent *opp* in a round *rnd*.

Estimating $p(a|h_{opp}(cc))$ can be done in various ways. Currently, we use a generic opponent model, fitted to a game database that includes human games played on IRC, and games generated by self-play.

Action Selection McRaise's action selection is based on a straightforward estimate of the expected value of each actions followed by selecting the action with the highest value. Given the current situation s and the estimate of EPP, $p_{win} = p_{win}(s)$, the expected value of an action a is estimated by

$$Q(s,a) = p_{win} \Pi(a,s) - B(a,s) \,, \tag{19}$$

where $\Pi(a,s)$ is the estimated pot size provided that action a is executed and $B(a,s)$ is the contribution to the pot. For estimating $\Pi(a,s)$ and $B(a,s)$, we assume that every player checks from this point on.

4.2 Lines of Action

In this subsection we explain first the game of Lines of Action (LOA). Then, the tournament program MIA and its enhancement RPS are described briefly.

The rules. LOA is a chess-like game with a connection-based goal. The game is played on an 8×8 board by two sides, Black and White. Each side has twelve pieces at its disposal. The players alternately move a piece, starting with Black. A move takes place in a straight line, exactly as many squares as there are pieces of either colour anywhere along the line of movement. A player may jump over its own pieces. A player may not jump over the opponent's pieces, but can capture them by landing on them. The goal of a player is to be the first to create a configuration on the board in which all own pieces are connected in one unit. The connections within the unit may be either orthogonal or diagonal.

MIA. MIA 4++ is a world-class LOA program, which has won the LOA tournament at the eighth (2003) and ninth (2004) Computer Olympiad. It is considered as the best LOA-playing entity of the world [17]. Here we will focus on the program component optimized in the experiments, the realization-probability search (RPS) [16]. The RPS algorithm is a new approach to fractional plies. It performs a selective search similar to forward pruning and selective extensions. In RPS the search depth of the move under consideration is determined by the realization probability of its move category. These realization probabilities are based on the relative frequencies which are noticed in master games. In MIA, the move categories depend on center-of-mass, board position, and capturing. In total there are 277 weights eligible to be tuned. Levy [10] argues that it may be necessary for a computerized search process to have numbers for the categories that are different from the ones extracted from master games. Therefore, we also believe that there is still room to improve the algorithm's performance by tuning its weights.

5 Experiments

In poker, we tested the RSPSA algorithm by optimizing two components of McRaise, the opponent model and the action selection. For both components, we compare the performance resulting by using RSPSA with the performance given by an alternative (state-of-the-art) optimization algorithm. The experiments for the opponent-model optimization are described in Sect. 5.1 and for the move-selection optimization in Sect. 5.2. In LOA, the RSPSA algorithm is employed to tune the realization-probability weights in MIA. According to [7] these weights belong to a class of parameters (termed class-S search decisions) that can be evaluated using search trees. In Sect. 5.3 we show how this property is exploited for improving the efficiency of the learning.

5.1 Tuning the Opponent Model

The opponent model of McRaise is embodied in the estimation of $p(a|h_{opp}(cc))$ (see Sect. 4.1). The model uses in total six parameters.

For problems where the number of parameters is small, FDSA can be a natural competitor to SPSA. We combined both SPSA and FDSA with RPROP.

The combined FDSA algorithm will be denoted in the following by RFDSA. Some preliminary experiments were performed with the standard SPSA, but they did not produce reasonable results (perhaps due to the anisotropy of the underlying optimization problem).

A natural performance measure of a player's strength is the average amount of money won per hand divided by the value of the small bet (sb/h). Typical differences between players are in the range of 0.05 to 0.2sb/h. For showing that a 0.05sb/h difference is statistically significant in a two-player game one has to play up to 20,000 games. It is possible to speed up the evaluation if *antithetic dealing* is used as proposed in Sect. 3.3. In this case, in every second game each player is given the cards which the opponent had the game before, while the community cards are kept the same. According to our experience, antithetic dealing reduces the necessary number of games by at least four. This technique is used throughout the poker experiments.

In the process of estimating the derivatives we employed the "Common Random Numbers" method: the same decks were used for the two opposite perturbations. Since many of the decks produced zero SPSA differences, thus producing zero contribution to the estimation of the sign of the derivatives, those decks that resulted in no-zero differences were saved for reuse. In subsequent steps, half of the decks used for a new perturbation were taken from those previously stored, whilst the other half was generated randomly. The idea of storing and reusing decks that 'make difference' can be motivated using ideas from importance sampling, a method known to decrease the variance of Monte-Carlo estimates.

The parameters of the algorithms are given in Table 1. Note that the performance corresponding to a single perturbation was evaluated by playing games in parallel on a cluster of sixteen computers. The number of evaluations (games) for a given perturbation was kept in all cases above 100 to reduce the communication overhead. The parameters of the opponent model were initialized to the original parameter settings of McRaise.

Table 1. Learning parameters of RSPSA and RFDSA for opponent model (OM), RSPSA and TD for evaluation function (EF) and RSPSA for policy (POL)learning. $\eta+$, $\eta-$, δ_0 (the initial value of δ_{ti}), δ^- and δ^+ are the RPROP parameters; Δ is the SPSA (or FDSA) perturbation size, λ is the parameter of TD; *batchsize* is the number of performance evaluations (games) in an iteration which, for RSPSA and RFDSA, is equal to the product of the number of perturbations (q), the number of directions (2) and the number of evaluations per perturbation (r).

	$\eta+$	$\eta-$	δ_0	δ^-	δ^+	$\Delta(\lambda)$	*batchsize*
RSPSA (OM)	1.1	0.85	0.01	1e-3	1.0	δ	$40 \times 2 \times 250$
RFDSA (OM)	1.1	0.85	0.01	1e-3	1.0	δ	$6 \times 2 \times 1500$
RSPSA (EF)	1.2	0.8	0.05	1e-3	1.0	$\delta/0.7$	$100 \times 2 \times 100$
RSPSA (POL)	1.1	0.9	0.01	1e-3	1.0	$\delta/0.3$	$100 \times 2 \times 100$
TD (EF)	1.2	0.5	0.1	1e-6	1.0	0.9	10000

The evolution of performance for the two algorithms is plotted in Fig. 1(top) as a function of the number of iterations. The best performance obtained for RSPSA was +0.170sb, whilst that of for RFDSA was +0.095sb. Since the performance resulting from the use of RSPSA is almost twice as good as that of resulting from the use of RFDSA, we may conclude that despite the small number of parameters, RSPSA is the better choice here.

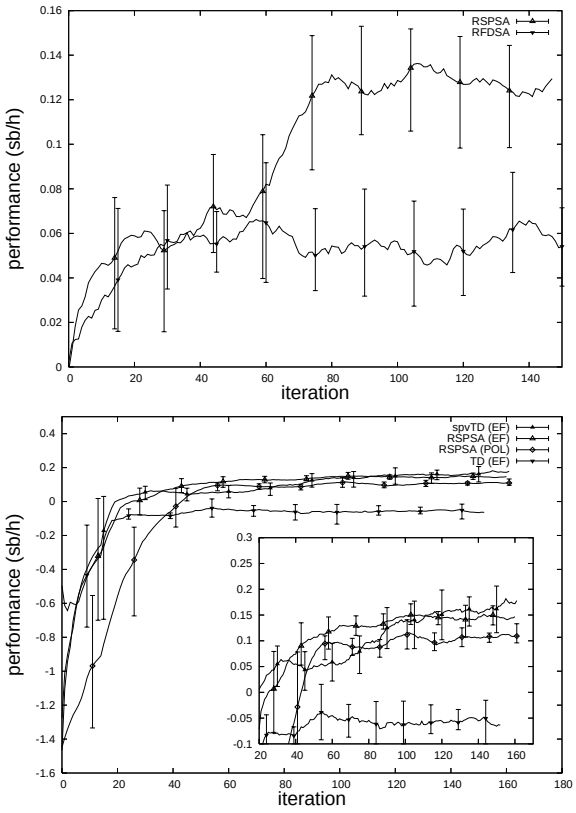

Fig. 1. Learning curves in poker: RSPSA and RFDSA for opponent-model learning (top) and RSPSA and TD for policy learning and evaluation-function learning (bottom). The graphs are obtained by smoothing the observed performance in windows of size 15. The error bars were obtained by dropping the smallest and largest values within the same windows centered around their respective coordinates.

5.2 Learning Policies and Evaluation Functions

The action-selection mechanism of McRaise is based on a simple estimation of the expected payoffs of actions and selecting the best action (see Sect. 4.1). This

can be cast as a 1-ply search w.r.t. the evaluation function V if s', being the situation after action a is executed from situation s, and if we define $V(s') = Q(s, a)$. In the experiments we represent either V or Q with a neural network. In the first case the output of the neural network for a given situation s represents $V(s)$ that is used in the 1-ply search, whilst in the second case the neural network has three outputs that are used (after normalization) as the probabilities of selecting the respective next actions. The input to the neural networks include EPP, the strength of the player's hand (as the a-priori chance of winning), the position of the player, the pot size, the current bet level, and some statistics about the recent betting actions of the opponent.

Learning evaluation functions is by far the most studied learning task in games. One of the most successful algorithm for this task is TD-learning and the best known example of successfully training an evaluation function is TDGammon [14]. By some researchers, the success can mostly be attributed to the highly stochastic nature of this game. Poker is similarly stochastic, therefore TD-algorithms might enjoy the same benefit. Temporal-difference learning had some success in deterministic games as well, e.g. [18]. In our experiment we use a similar design as the one used in MIA, combining TD(λ) with RRPOP (one source for the motivation of RSPSA comes from the success of combining TD(λ) and RPROP).

The parameters of the algorithms are given in Table 1. For RSPSA the same enhancements were used as in Sect. 5.1. We tested experimentally four algorithms: (1) RSPSA for tuning the parameters of an evaluation function (RSPSA(EF)), (2) RSPSA for tuning a policy (RSPSA(POL)), (3) TD for tuning an evaluation function (TD(EF)), and (4) TD for evaluation-function tuning with a supervised start-up (spvTD(EF)). For the latter a simple supervised algorithm tuned the neural network used as the evaluation function to match the evaluation function that was described in Sect. 4.1. The learning curves are given in Fig. 1(bottom). The best performance obtained for RSPSA(EF) was +0.194sb/h, for RSPSA(POL) it was +0.152sb/h, for TD(EF) it was +0.015sb/h and for spvTD(EF) it was +0.220sb/h. It is fair to say that TD performed better than RSPSA, which is a result one would expect given that TD uses more information about the gradient. However, we observe that for TD it was essential to start from a good policy and this option might not be always available. We note that although the two RSPSA algorithms did not reach the performance obtained by the combination of supervised and TD-learning, they did reach a considerable performance gain even though they were started from scratch.

5.3 Tuning the Realization-Probability weights

Generally the parameters of a game program are evaluated by playing a number of games against a (fixed) set of opponents. In [7], it was noted that for parameters such as search extensions alternative performance measures exists as well. One such alternative is to measure the 'quality' of the move selected by the search algorithm (constrained by time, search depth or number of nodes). The quality of a move was defined in [8] as the negative negamax score returned by

a sufficiently deep search for the position followed by the move. In the following, we describe two experiments. In the first, the performance is evaluated by game result. The result is averaged over 500 games played against five different opponents starting from 50 fixed positions with both colors. Each opponent is using a different evaluation function. Each player is searching a maximum of 250,000 nodes per move. In RSPSA the gradient is estimated with 500 perturbations, using one game per perturbation. The common random number technique is implemented in this case by using the same starting position, same opponent and same color for both the positive and the negative sides. In the second experiment the performance is evaluated by move score. The move score is averaged over a fixed set of 10,000 positions. For selecting the move the search is limited to 250,000 nodes. For evaluating the move a deeper search is used with a maximum of 10,000,000 nodes. Since the score of a move does not depend on the realization-probability weights, they are cached and reused when the same move is selected again (for the same position). So, the deeper search is performed far less frequently than the shallower search. The RSPSA gradient is estimated with 5,000 perturbations. Each side of a perturbation is evaluated using one position selected randomly from the set of 10,000 positions (the same position for both sides). Considering that the average game length in LOA is at least 40 ply, in the second experiment the gradient is estimated approximately four times faster than in the first experiment. Moreover, according to our observation, the estimates with move scores are less noisy as well. The parameters of the RSPSA algorithm for the two experiments are given in Table 2.

Table 2. Learning parameters of RSPSA for realization-probability weights using game result (GR) and move score (MS) for evaluation

	$\eta+$	$\eta-$	δ_0	δ^-	δ^+	$\Delta(\lambda)$	$batchsize$
RSPSA (GR)	1.2	0.8	0.005	1e-3	1.0	$\delta/0.7$	$500 \times 2 \times 1$
RSPSA (MS)	1.2	0.8	0.005	1e-3	1.0	$\delta/0.7$	$5000 \times 2 \times 1$

The learning curves for the two experiments are plotted in Fig. 2. Since the two experiments are using different performance measures, the performance for the two curves cannot be compared directly. Intuitively, the performance gain for the experiment using move scores (bottom) seems to be more significant than the one using game result. A more direct comparison can be performed by comparing the performance, measured as game result, of the best weight vector of each curve. The best performance obtained in the first experiment was 0.55, and the average game result corresponding to the best vector of the second experiment was 0.59. Therefore, we may conclude that using the move scores for estimating the performance improves the efficiency of the RSPSA algorithm.

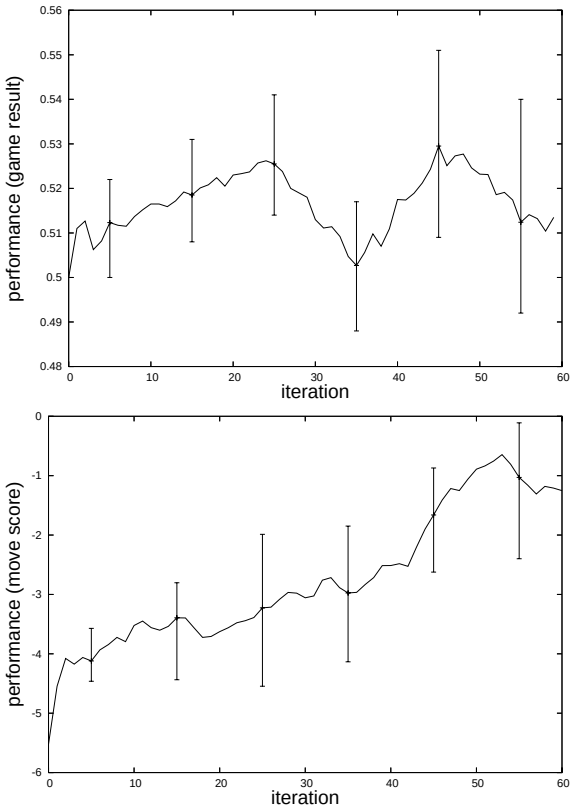

Fig. 2. Learning curves for RSPSA on game result (top) and moves score (bottom) as a function of the number of iteration. The graphs are obtained by smoothing the observed performance in windows of size 10. The error bars were obtained by dropping the smallest and largest values within the same windows centered around their respective coordinates.

6 Conclusions

This article investigated the value of a general purpose optimization algorithm, SPSA, for the automatic tuning of game parameters. Several theoretical and practical issues were analysed, and we have introduced a new SPSA variant, called RSPSA. RSPSA combines the strengths of RPROP and SPSA: SPSA is a gradient-free stochastic hill-climbing method that requires only function evaluations. RPROP is a first-order method that is known to improve the rate of convergence of gradient ascent. The proposed combination couples the perturbation parameter of SPSA and the step-size parameters of RPROP. It was argued that this coupling is natural.

Several other methods were considered to improve the performance of SPSA (and thus that of RSPSA). The effect of performing a larger number of perturbations was analyzed. An expression for the mean-square error of the estimate of the gradient was derived as the function of the number of (noisy) evaluations of the objective function per perturbation (q) and the number of perturbations (r). It was found that to optimize the mean-square error with a fixed budget $p = qr$, the number of perturbations should be kept at maximum. We suggested that besides using the method of "common random numbers", the method of antithetic variables should be used for the further reduction of the variance of the estimates of the gradient. These methods together are estimated to achieve a speed-up of factor larger than ten (since a smaller number of function evaluations is sufficient to achieve the same level of accuracy in estimating the gradient). The overall effect of these enhancements facilitated the application of SPSA for tuning parameters in our game programs MoGoRAISE and MIA, whilst without the proposed modifications SPSA was not able to yield noticeable improvements.

The performance of RSPSA was tested experimentally in the games of Omaha Hi-Lo Poker and LOA. In poker, the optimization of two components of MoGoRAISE were attempted: that of the opponent model and the action-selection policy. The latter optimization task was tried both directly when the policy was represented explicitly and indirectly via the tuning of the parameters of an evaluation function. In addition to testing RSPSA, for both components an alternative optimizer was tested (RFDSA and TD(λ), respectively). On the task of tuning the parameters of the opponent model, RSPSA led to a significantly better performance as compared with the performance obtained when using RFDSA. In the case of policy optimization, RSPSA was competitive with TD-learning, although the combination of supervised learning followed by TD-learning outperformed RSPSA. Nevertheless, the performance of RSPSA was encouraging on this second task as well. In LOA, the realization-probability weights of MIA were tuned by RSPSA. In the experiments, we have shown that using move scores for performance evaluation instead of game results can speed-up and improve the performance of RSPSA at the same time. In summary, from the experimental study we may conclude that the RSPSA algorithm using the suggested enhancements is a viable approach for optimizing parameters in game programs.

Acknowledgements

We would like to acknowledge support for this project from the Hungarian Academy of Sciences (Cs. Szepesvári, Bolyai Fellowship).

References

1. D. Billings, A. Davidson, T. Shauenberg, N. Burch, M. Bowling, R. Holte, J. Schaeffer, and D. Szafron. Game Tree Search with Adaptation in Stochastic Imperfect Information Games. In *Proceedings of Computers and Games (CG'04)*, 2004.
2. Y. Björnsson and T. A. Marsland. Learning Extension Parameters in Game-Tree Search. *Journal of Information Sciences*, 154:95–118, 2003.

3. K. Chellapilla and D. B. Fogel. Evolving Neural Networks to Play Checkers Without Expert Knowledge. *IEEE Transactions on Neural Networks*, 10(6):1382–1391, 1999.
4. J. Dippon. Accelerated Randomized Stochastic Optimization. *Annals of Statistics*, 31(4):1260–1281, 2003.
5. C. Igel and M. Hüsken. Empirical Evaluation of the Improved Rprop Learning Algorithm. *Neurocomputing*, 50(C):105–123, 2003.
6. N. L. Kleinman, J. C. Spall, and D. Q. Neiman. Simulation-based Optimization with Stochastic Approximation using Common Random Numbers. *Management Science*, 45(11):1570–1578, Nov 1999.
7. L. Kocsis. *Learning Search Decisions*. PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands, 2003.
8. L. Kocsis, H. J. van den Herik, and J. W. H. M. Uiterwijk. Two Learning Algorithms for Forward Pruning. *ICGA Journal*, 26(3):165–181, 2003.
9. P. L'Ecuyer and G. Yin. Budget-dependent Convergence Rate of Stochastic Approximation. *SIAM J. on Optimization*, 8(1):217–247, 1998.
10. D. Levy. Some Comments on Realization Probabilities and the Sex Algorithm. *ICGA Journal*, 25(3):167, 2002.
11. M. Riedmiller and H. Braun. A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm. In E.H. Ruspini, editor, *Proceedings of the IEEE International Conference on Neural Networks*, pages 586–591, 1993.
12. J. C. Spall. Multivariate Stochastic Approximation Using a Simultaneous Perturbation Gradient Approximation. *IEEE Transactions on Automatic Control*, 37:332–341, 1992.
13. J. C. Spall. Adaptive Stochastic Approximation by the Simultaneous Perturbation Method. *IEEE Transactions on Automatic Control*, 45:1839–1853, 2000.
14. G. Tesauro. Practical Issues in Temporal Difference Learning. *Machine Learning*, 8:257–277, 1992.
15. J. Theiler and J. Alper. On the Choice of Random Directions for Stochastic Approximation Algorithms. *IEEE Transactions on Automatic Control*, 51:476-481, 2006.
16. Y. Tsuruoka, D. Yokoyama, and T. Chikayama. Game-tree Search Algorithm based on Realization Probability. *ICGA Journal*, 25(3):132–144, 2002.
17. M. H. M. Winands. *Informed Search in Complex Games*. PhD thesis, Universiteit Maastricht, Maastricht, The Netherlands, 2004.
18. M. H. M. Winands, L. Kocsis, J. W. H. M. Uiterwijk, and H. J. van den Herik. Learning in Lines of Action. In *Proceedings of BNAIC 2002*, pages 99–103, 2002.