An Introduction to Amoeba

Christopher H. Brooks

1 Introduction

The problem of maximizing (or minimizing) a real-valued function $f(\mathbf{x})$, where $\mathbf{x}$ may contain both continuous and discrete variables, is both well-studied and relevant in many domains, including Experimental Design, Operations Research, and Artificial Intelligence. Common AI/OR techniques for solving this include genetic algorithms [8] and simulated annealing [12]. However, both of these methods have the drawback of often requiring a large number of function evaluations (or *exploration*) to find an optimum. In situations where each function evalaution is costly, this is not acceptable.

Direct search is an methodology that is able to quickly locate a function's optimum, even when the space is discontinuous or rugged. The most common algorithmic implementation of direct search is the Nelder-Mead algorithm [10], also known as amoeba. Direct search operates by locating points on a function landscape, constructing a simplex between them, and manipulating this simplex. Its effectiveness lies in its ability to use global properties of the landscape, such as its overall slope, without requiring local gradient information.

Within this document, I will provide an introduction to direct search and a description of some methods we have developed to extend this algorithm beyond the static, deeterministic functions it was originally intended for. Section 2 provides some background on direct search as an optimization method and discusses its pros and cons. Section 3 provides a detailed description of the amoeba algorithm for direct search. Section 4 discusses ways of handling common problems with amoeba, including boundary violations and constraints between variables. Section 5 discusses extensions for dealing with noise, non-stationary environments, and environments in which not all variables are under the experimenter's control. Section 6 summarizes direct search and identifies areas for future research.

2 Direct Search

Direct search is a method for locating optima on a function landscape that is rugged, multimodal or discontinuous. Its strengths include its ability to rapidly converge on a solution and the fact that it does not rely on any gradient information. Its biggest weaknesses are the lack of theoretical proofs regarding convergence and its tendency to get 'stuck' in some high-dimensional spaces.

The most well-known direct search method is the Nelder-Mead algorithn, referred to as the 'amoeba' algorithm in Numerical Recipes [11]. The Nelder-Mead algorithm was originally described in the 1960s [10], although many other direct search methods, including others using a simplex, were proposed around the same time. Wright [14] provides a nice history of the development of direct search methods. Nelder-Mead is widely used in chemistry and chemical engineering for sequentially selecting a set of experiments to perform in order to maximize some independent variable, such as reaction rate. Walters, et al [13] discuss the application of this method to selection of experiments (typically by hand) and also provide a bibliography of direct search methods.

Despite this success, Nelder-Mead and other simplex methods have not received much attention within the AI and operations research communities, primarily due to their 'ad hoc' nature. There are few results describing convergence properties, and those that exist (e.g. [7]) focus on one or two input dimensions and also highlight problems that the simplex method cannot solve. Most criticism of the simplex method seems to focus on the lack of theoretical results, the ad hoc nature of the expansion and contraction parameters, or the method's inability to perform in 'high" dimensions. Now that other methods such as genetic algorithms and simulated annealing, which also are lacking in theory and use ad hoc, hand-tuned parameters, have gained acceptance as practical methods for dealing with complex optimization problems, the first two objections are less relevant. As for the last objection (inability to deal with 'high'-dimensional problems), it's not clear what constitutes a high-dimensional or difficult problem; our past research [6] has successfully used amoeba on a 100-dimensional landscape.

Simplex methods do have some well-known problems. For example, the simplex can collapse or degenerate in some dimensions, thereby preventing exploration of the entire search space. Simplex methods can also have problems on landscapes with plateaus; since they don't retain any momentum information the way that gradient-based methods do, they can wander on a flat region indefinitely. Also, the simplex can converge prematurely on a local optimum (of course, this is a problem for any non-exhaustive search method, but amoeba can be particularly vulnerable).

It is becoming clear that simplex methods such as amoeba are very useful for a wide set of difficult-to-optimize problems. Of course, they aren't a panacea, but they do seem to be effective on a wide variety of problems, particularly those with rugged, discontinuous landscapes. The secret relies on using the simplex to detect overall trends in the landscape, as opposed to local changes. By selecting a set of widely-dispersed points and comparing them, the algorithm is able to avoid local optima and move across valleys. (The same is true of genetic algorithms.)

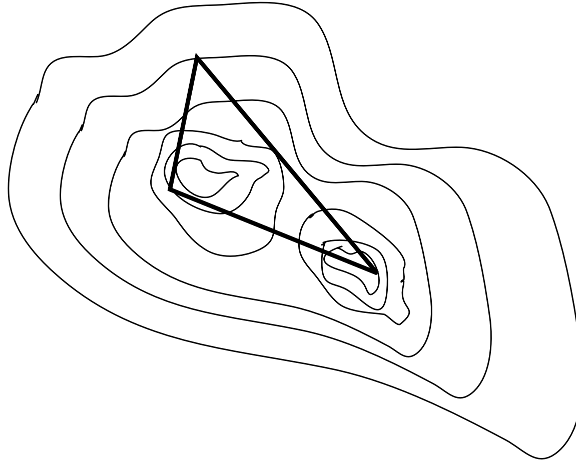

Figure 1: A topological view of a function with two independent variables and one dependent variable (to be optimized). The simplex is shown in heavy black lines.

3 The Amoeba Algorithm

In this section, we provide a step-by-step description of the Nelder-Mead, or amoeba, algorithm. The name amoeba stems from the 'oozing' behavior of the simplex as it traverses the landscape.

Recall that the problem is to optimize a function $f(\mathbf{x} : \mathbf{R}^n \rightarrow \mathbf{R}$. For the moment we will focus on the case where all the variables comprising $\mathbf{x}$ are continuous; dealing with discrete variables will be discussed in section 4.

We begin by selecting $n+1$ points from $\mathbf{x}$ at random (where $n = |\mathbf{x}|$) and evaluating them. These points are the vertices of an n-dimensional simplex (see figure 1). The algorithm then proceeds as follows:

1. Sort the vertices according to their $f(\mathbf{x})$.

2. If the worst vertex and the best vertex are within a given tolerance of each other, then stop.

3. The iteration begins by trying to choose a better point for the worst vertex. Find the centroid of the other n vertices (all but the worst) and reflect the worst vertex through this point. The new point shjould be the same distance from the centroid as the old (worst) point. (Each step is illustrated in figure 2.)

4. Evaluate this point. If it is an improvement (i.e. f_{new} is better than f_{worst}), replace the previous worst vertex with this one.

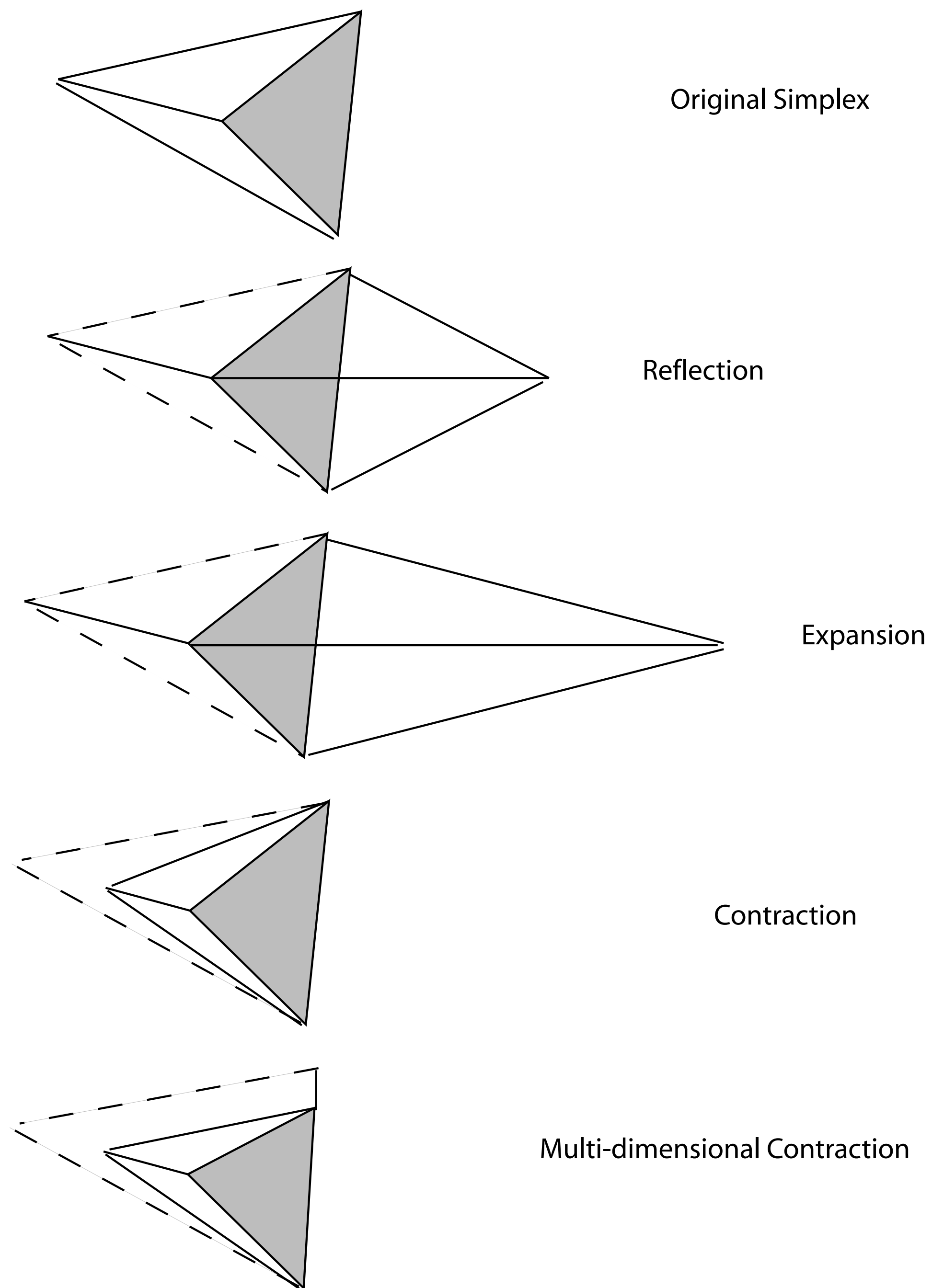

Figure 2: A three-dimensional four-point simplex reflecting, expanding and contracting. Note that the dependent variable is not shown here, only the manipulated (independent) variables.

5. If this new vertex is better than the previous one, extend the new point by a factor of two along the line between the point and the centroid. In other words, keep moving it in the direction is was moved.

6. If the new vertex is worse than the second-worst vertex, then we can see that we haven't made an improvement, and so we need to contract the simplex along one dimension. Reduce the distance between the new point and the centroid by half.

7. If this still doesn't improve the new point (it's still worse than the second-worst), then contract the simplex along all dimensions. The distance between the centroid and each point except the best is cut in half.

8. return to step 1.

This simple strategy of repeatedly reflecting, coupled with contractions to pull out of minima, proves to be a highly effective strategy. Since no gradient information is computed, it doesn't matter whether f is differentiable, as long as it can be evaluated at any point. Amoeba works by 'pulling' the simplex toward high regions of the landscape; the initial selection of points will hopefully put at least one vertex on a promising slope. The simplex is then pulled across this hill - the reflection allows it to skip over small local optima and ridges, while still sampling different regions of the space.

4 Handling Common Problems with Amoeba

While amoeba can be a very effective optimization algorithm, there are a number of common problems that can occur when using it. These include dealing with bounded spaces, discrete variables, large plateaus, and premature convergence. These problems are addressed in the next section.

4.1 Bounded Spaces

A common problem that has occurred in our research is a search space that is bounded in one or more dimensions. For example, one or more elements of $\mathbf{x}$ may only be allowed positive values, or the experimenter may have domain knowledge about the maximal possible value of an input. The basic amoeba algorithm assumes that all elements of $\mathbf{x}$ have inputs that are valid for $(-\inf, \inf)$, and so it can reflect the simplex into any part of this space. There are two straightforward ways to deal with this problem.

The first way is to include a filter to f that assigns a large negative value to any point that violates the boundary condititons. This will have the effect of forcing an immediate contraction (see section 3 when a newly sampled point is worse than the present worst). In many circumstances this works fine, although if optima lie on or near the boundary, it may take more evaluations to find them, since the simplex go through a number of contractions to become small enough for a reflection to place it near this boundary without crossing it.

The second solution is treat all boundary violations as points on the boundary. The designer can either add a filter that maps boundary-violating points to points on the boundary (e.g., if x' is a boundary value, $f(x' + \delta)$ would be evaluated as $f(x')$), or she can modify the algorithm so that reflections of the simplex which take it beyond a boundary instead place the new point on the boundary. The first method has the effect of creating an infinite plateau extending beyond the boundary; if one has reason to believe that this boundary value is an optimum, then this may be a useful strategy. It can also pull the simplex toward this plateau and away from the rest of the search space, so care must be used with this approach. The second method (modifying the algorithm) avoids the plateau problem, but can potentially cause the simplex to collapse along some dimensions. Once this has happened, certain parts of the search space are unreachable. A collapsing simplex can be a problem more generally with amoeba; workarounds for this are discussed in section 4.5.

4.2 Discrete Variables

Another common problem that can arise when using amoeba is the case when some elements of $\mathbf{x}$ are discrete. The solutions are similar to those for value conditions: the experimenter can either add a filter that maps inputs to the nearest discrete element, or modify the amoeba algorithm to require newly selected points to 'snap to' the nearest integer value.

The first solution, mapping real-valued simplex points to integer values, is the most straightforward. As an example, assume that f is a function with two inputs, x_1 and x_2, and that x_2 is only defined over the integers. We would then apply amoeba to a filter function $\hat{f}$, which uses real values for x_1 and x_2. When evaluating a particular point on f, we then round x_2 to the nearest whole number. This has the effect of creating a stair-stepping, terraced search space, where all values in $(x_2 - 0.5, x_2 + 0.5)$ get evaluated identically. Whether this is a problem will depend upon the size and dimensionality of your search space; amoeba has no problem with small terraces, but large plateaus can be a problem (see the following section).

The second solution involves modifying amoeba, rather than filtering the inputs to f. When a new point is selected by amoeba, the algorithm is modified to instead choose the nearest point which satisfies the constraints of the inputs. This point is used as the new point of the simplex. In effect, the simplex 'snaps to' a coordinate. As with boundary violations, the danger with this approach is in premature collapse of the simplex; the snap-to can lead to a new point being very close to an existing point. Once this happens, the simplex can collapse along this dimension. If this problem is also checked for, then snap-to becomes a viable solution.

4.3 Large Plateaus

Another problem that amoeba is susceptible to is getting 'caught' in a plateau. Imagine the case in which all $n + 1$ vertices evaluate to the same value for

f. (Often, this value is 0.) Since amoeba has no memory beyond the points currently in the simplex, those are the only points that will be used to generate new points to explore. If each of these points also maps to a point on the plateau when reflected, then the simplex will 'wander' about the plateau, repeatedly visiting the same points. (In fact, even if some reflections might lead away from the plateau, wandering can still occur if the wrong points are initially chosen for reflection.)

To avoid this problem, a counter can be added to the basic algorithm. Every time a new point is evaluated and produces a value equal to the previous point it is replacing, the counter is incremented. If the new value is different than the previous one, the counter is reset. If the counter reaches $n + 2$, then we know that we have just reflected each point with no improvement. (Recall that the algorithm only tries a contraction if a newly-selected point is strictly *worse* than the previous point, so on a plateau, the simplex will be repeatedly reflected.) At this point, the algorithm must be restarted. An experimenter can either select a new set of random simplex points, or, if a history has been kept (see section 5), a promising set of previously-visited points can be chosen to restart the algorithm.

4.4 Premature Convergence

Since amoeba is a hill-climbing algorithm, it is subject to the most common of hill-climbing problems: premature convergence on a local optimum. In general, if the search space contains multiple optima, only an exhaustive search can guarantee finding the global optimum. Recall that we are particularly interested in finding a solution quickly; this means that we will have to be satisfied with often reaching a local optimum. However, we can still try to structure the search so that we find a good local optimum.

Recall the way in which amoeba works: if a reflection is not able to remove the worst point, the simplex is contracted along one dimension. If this doesn't help, the simplex is contracted along multiple dimensions. This is exactly what will happen when one vertex of the simplex is at the optimum in a unimodal landscape. In fact, the test for whether amoeba is done is to check whether the highest and lowest points are within a given precision of each other. So, if all the points in a simplex are within the same basin of attraction of an optimum, the simplex will converge to that optimum. Therefore, in order to improve amoeba's chances of of finding a good solution, the initial points should be selected so as to start out in many different basins of attraction. Garnier and Kallel [5] provide methods for sampling points so as to ensure (probabilistically) that each basin of attraction is sampled at least once. In practice, ensuring that starting points are widely spaced apart is a good heuristic.

Another technique, commonly used within hillclimbing, is random restart. Once an optimum is found, the value is cached. The algorithm is then restarted with a new set of initial points. If the experimenter has an estimate of the optimal value and the cost of acquiring each new data point, Garnier and Kallel's methods for estimating the number and size of attraction basins can be in-

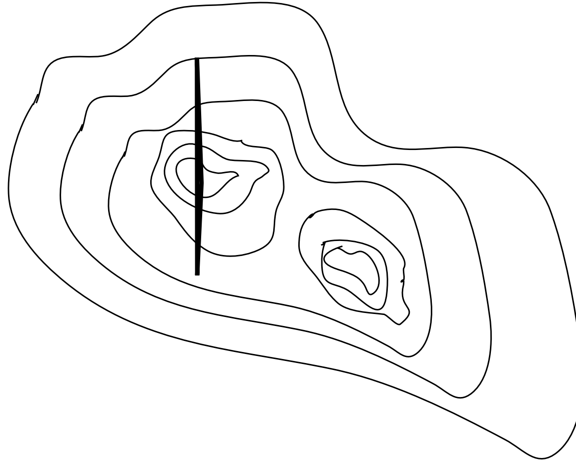

Figure 3: A two-variable simplex in which one dimension has collapsed.

corporated into a decision-theoretic framework which allows an experimenter to estimate the likelihood of finding a better solution, the cost of finding that solution, and then make a decision as to whether to search further.

4.5 Collapsing Simplex

One of the most common problems that amoeba can have is the collapse of the simplex along one or more dimensions. As an example, consider the simplex shown in figure 3. In this case, all three points in the simplex have the same x_2 value. In this case, all further reflections and contractions will occur along the line formed by the three points, and the rest of the search space will go unexplored. This collapse commonly occurs after a contraction or (more often) a series of contractions, when three or more points will end up on (or very near) a hyperplane.

As with the plateau problem, the biggest problem is in detecting that this has occurred. A conservative approach is to check all points in the simplex after a contraction - if any three points are collinear, the simplex has collaped along the dimension of the line. If the search space is high-dimensional, this may be an expensive operation. A simpler method is to compare the new point to each pair of points in the simplex.

If a collapse has occurred, there are two possible solutions. The first is to undo the previous contraction and continue the algorithm by reflecting the second-worst point. The second approach is to discard the contracted point and select a new one, either at random or from a history of previously visited points.

5 Extensions to Amoeba

The previous section discussed common problems that can occur when running amoeba on the sorts of problems it was designed for (stationary, noise-free multivariable optimization) and suggested some workarounds. There are still plenty of problems that one might like to apply amoeba to, but for which the vanilla algorithm is unsuited. These include noisy or nonstationary environments and environments in which some of the variables in $\mathbf{x}$ are outside the control of the experimenter, such as a multiagent environment. This section describes extensions to amoeba to help handle each of these problems.

5.1 Noise

The traditional amoeba algorithm assumes that each evaluation of f produces a correct result, without noise or error. If this is not the case, amoeba can potentially have problems. There are two possible results. First, f may return an incorrectly poor result and a point p may look worse than it actually is. This is not a huge problem; it may temporarily push the simplex away from that part of the search space, but eventually a new point near p will be sampled if p is actually on the path to an optimum. A larger problem is if f returns an incorrectly good response for p. In this case, the algorithm can wind up either reaching a poor local optimum or collapsing at a non-optimal point near p. One common workaround suggested by Walters, et al [13] is called the $k+1$ rule: if a point has been in the simplex for $k+1$ iterations (where k is the dimensionality of f's input), it should be resampled. This is fine, and might eventually catch noise errors, but it can lead to extra function evaluations and does not provide any information about the structure of the noise.

In out past work [2], we took a different approach to extending amoeba. Our work dealt with nonstationarity, but the approach can also be applied to noisy data. We coupled amoeba to a radial basis function (RBF) network, which served as a model of the landscape.

An RBF network is similar to a neural network; it consists of a set of nodes, each of which perform a function evaluation on the inputs. These evaluations are then weighted and summed to produce a global approximation of a function. There are three primary differences between RBF networks and traditional feedforward neural networks. The first is that an RBF node has a *center*, and its activation function (typically Gaussian) has a strength inversely proportional to the distance between the center and the input to f. This means that, unlike a neural network's hidden nodes, RBF network nodes can have a local influence. Second, the number of nodes in a network can be changed as data is collected; in our work, we added a new node for each data point (barring duplicates). Algorithms also exist [4] for determining the optimal placement of a fixed number of centers. This allows a learner to either fit observed data exactly or select a number of centers that minimizes overfitting. Third, the network is easily trained. Since it is a single-layer network, once the centers are known, the weights can be solved for using matrix inversion. No gradient descent is needed. For purposes

of its application to amoeba, it is probably sufficient to think of an RBF network as a tool for doing nonlinear function approximation. Interested readers should consult Broomhead and Lowe [4] or Moody and Darden [9]. Bishop [1] also provides both a nice introduction to RBF networks and a comparison to traditional feedforward neural networks.

Coupling a nonlinear function approximator (NFA) to amoeba provides two benefits: first, it gives amoeba a memory that can be used for restarting or moving out of plateaus (as discussed in section 4), and second, it allows noise to be absorbed. The technique works as follows: assume that we have a simplex s, which has a set of vertices $v_1, ..., v_n$. Each of these vertices has an estimated f value, denoted $\hat{f}(v_i)$, derived from past observations of the world. We use amoeba to select a new vertex v_{new} as usual, and get the potentially noisy value $f(v_{new})$. This value is used to update all the nodes in the RBF network, which is then queried to get $\hat{f}(v_new)$. By using the RBF network in order to construct and store an explicit model of f, we can both contend with noisy data and estimate the amount of noise in a particular data point, by comparing the network's prediction for v_{new} to the actual $f(v_{new})$. (Of course, we also need an estimate of how accurate the network will be for v_{new} - this can be obtained by determining how close v_{new} is to the RBF network's centers.)

5.2 Nonstationarity

A related, yet distictly different problem from noise is that of nonstationarity. Noise is typically thought of as a random variable drawn from a fized distribution that is added to every observation of f. Nonstationarity, on the other hand, refers to a change in f that has some measurable drift or trend. The actual landscape of f is changing somehow, possibly due to the actions of other agents, the passage of time, or some other exogenous event. In this case, simply storing old data in an RBF network and using this to absorb noise is not enough - once f has changed, the value of previous observations is weakened. (How much these observations are weakened depends upon the both the amount of nonstationarity and what the learner knows about the forces causing nonstationarity.)

Our past work [3, 6] has modified amoeba in a straightforward way to handle nonstationarity. As above, amoeba retains a list of every point v visited and the resulting value $f(v)$. If a point is revisited and a different $f(v)$ is observed, then we can conclude that f has changed. Note that this assumes there is no noise in f. If there is, one could make a probabilistic assessment of whether f has changed, based upon the difference between the points and the noise model in use for f.

Once a change has been observed, the next step is to re-expand the simplex, so as to keep it from contracting around a local optimum. An expansion parameter r is chosen, and n new vertices are drawn randomly from a cloud of radius r around the current best vertex. Obviously, the correct choice of r is very important. If r is too small, the simplex will contract around the previous best observation. If r is too large, the simplex will waste iterations converging on a solution. The correct choice for r can potentially be determined either through

a model of the nonstationarity of f, if such a thing is available, or learned, if the nonstationarity is constant.

This is a relatively simple solution, and one that is most useful when f changes due to 'shocks', when the entire landscape changes suddenly, as opposed to a gradual drift. Satisfactory solutions for nonstationarity in the form of constant gradual drift are a topic for future research.

5.3 Uncontrolled Variables

The third extension to amoeba that we have considered in our past work [2] is the case where some inputs of f are out of the control of the experimenter. In our work, those inputs were the choice variables of another agent - they could by any exogenously selected variable. There are two possible ways of dealing with this problem, depending upon how much is known about the process selecting values for these variables.

If little or nothing is known about the process selecting these variables, a learner's best course of action is often to treat them as noise and ignore them. This then reduces to the case described in section 5.1, where the learner will construct a simplex using the m variables it can directly control and use an RBF network to attempt to absorb what will be treated as noise. This is most effective if the values of the uncontrollable variables really are being produced by some sort of random process, as opposed to the intentional decisions of another agent.

If a learner knows something about the process that is generating values for these variables, then a slightly more sophisticated approach can be taken. Once again, an RBF network will be used to store past data. On each iteration, a model of the generating process is used to construct a prediction of the values for these uncontrollable variables. We then run amoeba on the landscape formed by the RBF network to find the optimal action, given this prediction. This is used as the next point to explore. Obviously, the efficacy of this approach depends upon how well we are able to predict the values that will be applied to the uncontrolled variables.

Figure 4 provides an example of this process, taken from our work on information economies [2]. In this case, two producers are both trying to learn the optimal price(s) to charge. In order to select a price vector, a producer must know what its opponent is offering. It uses a best-response model to construct this prediction, then applies amoeba to the RBF model is learning, using the prediction to fill in uncontrolled variables. The price is offered, and the feedback (profit) is observed, along with the actual prices offered by the opponent.

6 Further modifications

This document has presented a brief history of sequential simplex optimation, outlined the operation of amoeba, a particular simplex algorithm, and described

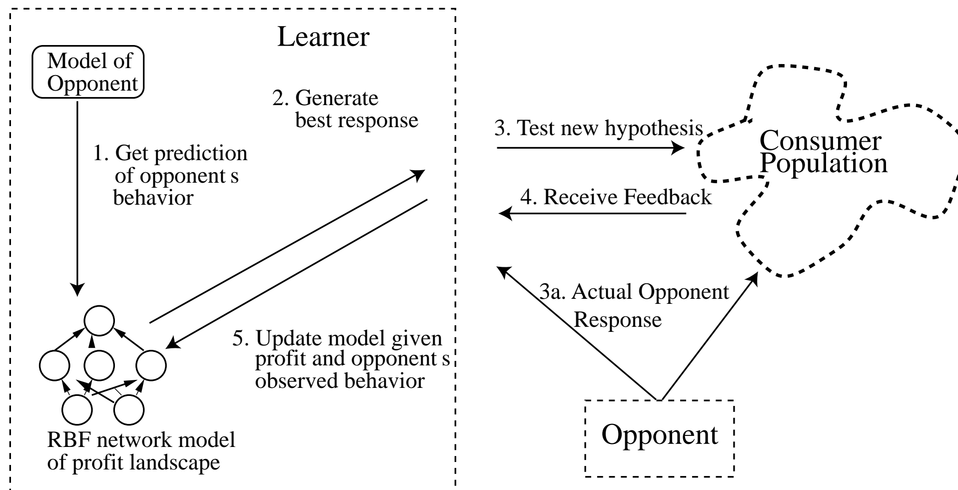

Figure 4: This figure illustrates the way in which a RBF network can be used to store a model of a landscape to be learned. The RBF network stores past observations and allows a learner to find a best response given a prediction of opponent behavior. Amoeba is used to search this 'virtual' landscape.

some modifications that can be made to amoeba, both to deal with its weaknesses and to extend its capabilities. There are still a number of areas in which work can be done with amoeba to make it more robust and dependable.

One particular weakness of direct search methods is the lack of theoretical guarantees. Very little work has been done on determining when amoeba will converge at all, much less find an optimal solution. Stronger theoretical guarantees, as well as bounds on search cost, would serve as a reassurance for users and allow amoeba to be included within a decision-theoretic learning framework in which learning costs must be estimated.

In fact, a meta-learning wrapper for a direct search algorithm would be a great aid. We have described some ad hoc ways of doing this, either by detecting nonstationarity or plateaus, but a decsion-theoretic module that analyzed the progress of amoeba and decided whether the simplex should continue, restart or stop based on the estimated 'goodness' of the current solution and the cost of future learning would be an excellent contribution.

Also, our current approach to nonstationarity is incomplete, since it depends on nonstationarity taking the form of sudden discrete shocks, rather than gradual change. Incorporating a form of updating more like that used by Q-learning might be a way to overcome this. More generally, it seems like amoeba provides a particularly nice means of *exploring* efficiently - tying amoeba with other algorithms that are able to better store a representation of a hypothesis space (such as an RBF network) would be a strong contribution in addressing the exploration-exploitation tradeoff.

References

[1] Christopher M. Bishop. *Neural Networks for Pattern Recognition.* Oxford University Press, 1995.

[2] C. H. Brooks, E. H. Durfee, and R. Das. Price wars and niche discovery in an information economy. In *Proceedings of ACM Conference on Electronic Commerce (EC-00)*, Minneapolis, MN, October 2000.

[3] Christopher H. Brooks, Rajarshi Das, Jeffrey O. Kephartand Jeffrey K. MacKie-Mason, Robert S. Gazalle, and Edmund H. Durfee. Information bundling in a dynamic environment. In Peter Wurman and Amy Greenwald, editors, *Proceedings of the 2001 IJCAI Workshop on Economic Agents. Models and Mechanisms*, August 2001.

[4] D.S. Broomhead and David Lowe. Multivariable functional interpolation and adaptive networks. *Complex Systems*, 2:321–355, 1988.

[5] Josselin Garnier and Leila Kallel. How to detect all attraction basins of a function. In L. Kallel, B. Naudts, and A. Rogers, editors, *Theoretical Aspects of Evolutionary Computation*, pages 343–365. Springer-Verlag, 2000.

[6] Jeffrey O. Kephart, Christopher H. Brooks, Rajarshi Das, Jeffrey K. MacKie-Mason, Robert S. Gazzale, and Edmund H. Durfee. Pricing information bundles in a dynamic environment. In *Proceedings of the 2001 ACM Conference on Electronic Commerce*, 2001.

[7] Jeffrey C. Lagarias, James A. Reeds, Margaret H. Wright, and Paul E. Wright. Convergence properties of the nelder-mead simplex method in low dimensions. *SIAM Journal of Optimization*, 9(1):112–147, 1998.

[8] Melanie Mitchell. *An Introduction to Genetic Algorithms.* MIT Press, Cambridge, MA, 1998.

[9] John Moody and Christian J. Darken. Fast learning in networks of locally-tuned processing units. *Neural Computation*, 1:281–294, 1989.

[10] J.A. Nelder and R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[11] William H. Press. *Numerical Recipes.* Cambridge University Press, 1992.

[12] Colin R. Reeves, editor. *Modern Heuristic Techniques for Combinatorial Problems.* John Wiley & Sons, New York, 1993.

[13] Frederick H. Walters, Jr. Lloyd R. Parker, Stephen L. Morgan, and Stanley N. Deming. *Sequential Simplex Optimization: a technique for improving quality and productivity in research, development, and manufacturing.* CRC Press, 1991.

[14] Margaret H. Wright. Direct search methods: Once scorned, now respectable. In D.F. Griffiths and G.A. Watson, editors, *Numerical Analysis 1995 (Proceedings of the 1995 Dundee Biennial Conference in Numerical Analysis)*, pages 191–208. Addison Wesley Longman, Harlow, UK, 1995.