SOS Supplemental Materials

Blair C. Armstrong
Department of Psychology and
Center for the Neural Basis of Cognition
Carnegie Mellon University

Christine E. Watson
Department of Neurology and
Center for Cognitive Neuroscience
University of Pennsylvania

David C. Plaut
Department of Psychology and
Center for the Neural Basis of Cognition
Carnegie Mellon University

Abstract

This document provides a detailed overview of the main steps involved in an SOS optimization. This may be relevant both for readers who are unfamiliar with optimization procedures and for readers interested in taking full advantage of the specific characteristics of the SOS algorithm when designing and running their own optimizations. Additional information and materials, including a tutorial which applies many of the principles described herein, may be found in the main SOS article and at `http://sos.cnbc.cmu.edu`.

DETAILED OVERVIEW OF THE MAIN STEPS IN AN SOS OPTIMIZATION

Step 1: Defining the experimental conditions

The first step is defining the experimental conditions (different samples of items) in the design. For example, if the purpose of the experiment is to study the effects of word frequency, this could involve creating low- and high-frequency conditions.

Step 2: Specifying constraints and penalties for violating the constraints

Once the experimental conditions have been determined, constraints must be constructed to establish the desired relationships between the different variables, both across and within these conditions. Constraints often consist of a desire to maximize differences on the variables of interest while minimizing differences on confounding variables. For instance, if one were studying the effects of word frequency, the primary constraint might be that two groups of words have different mean frequency values (i.e., low- and high-frequency conditions). A second constraint might be that the two groups are minimally different in word length.

The issue of assessing how well constraints have been satisfied is more substantial. The basic idea consists of defining a cost or penalty for violations of the constraint. Greater violations lead to increased costs and smaller violations to reduced costs. In the case of manually selecting items, a researcher is assumed to derive these kinds of measures implicitly. In SOS, a cost penalty is operationalized in an objective function referred to as a cost function, or a cost for short.

Many different objective functions could be used to instantiate the idea of cost, as long as they increase monotonically with the degree of constraint violation. For instance, a simple cost function for minimizing differences between two conditions, *c1* and *c2*, on a particular nuisance variable, $\mathbf{x}$, is to square the difference between the means of the two conditions on that variable:

$$O_{MIN}(\mathbf{x}_{c1}, \mathbf{x}_{c2}) = (\bar{\mathbf{x}}_{c2} - \bar{\mathbf{x}}_{c1})^2 \tag{S.1}$$

Maximizing the differences between two conditions can be achieved simply by reversing the sign of the cost function used to minimize differences. To enforce a particular rank ordering of the means of the two conditions—for example, that the mean frequency of the high-frequency group is not just different but higher than that of the low-frequency group—an additional term, $s(\bar{\mathbf{x}}_{c1}, \bar{\mathbf{x}}_{c2})$, can be added to the equation corresponding to a simple sign function that equals 1 when *c1* is less than *c2* and -1 otherwise:

$$O_{orderMAX}(x) = -O_{MIN}(\mathbf{x}_{c1}, \mathbf{x}_{c2})\ s(\bar{x}_{c1}, \bar{x}_{c2}) \tag{S.2}$$

This added term ensures that if *c1* is greater than *c2*, larger (positive) costs result. As long as *c1* is less than *c2*, greater differences between the conditions produce smaller (more negative) costs.

By expressing these constraints as costs on the same scale, it becomes possible to assess the overall cost of a particular set of stimuli by calculating the cost associated with each constraint and summing these values. (There is a minor complication in the case of what we refer to as *hard* constraints, which we discuss later). To negate the effects of different variables having broader or narrower distributions of values (thus leading to superficially different costs for the same relative differences), SOS calculates cost on the normalized values of each variable rather than on the raw values directly. van Casteren and Davis (2007) noted a similar benefit from normalization in the performance of their algorithm.

In many cases, the use of these two simple cost functions to express constraints related to the maximization or minimization of differences performs remarkably well. An understanding of these constraints at the level of detail presented above may therefore be sufficient for many applications and for acquiring a theoretical understanding of the basic operation of SOS. However, depending on the particular constraints that a researcher wishes to instantiate and the complexity and difficulty of achieving them, it may be useful to modify the default constraints or use other constraints that are more suitable for a given goal. An appendix in the main SOS article provides a detailed description of the full vocabulary of constraints and cost functions currently implemented in SOS, including item-level and group-level constraints, entropy (uniform distribution of items) constraints, correlation-matching constraints, and more. These variations allow for considerable sophistication in creating a wide variety of designs, as illustrated in the example optimizations reported in the main article.

Step 3: Identifying a population and an initial sample of stimuli

Any population of stimuli can be used by SOS as long as the variables to be optimized are expressed on an interval scale. By default, the initial samples for each condition consist of random selections of the desired number of items from the population. Alternatively, some or all of the items in a condition can be pre-specified by the experimenter; further, these items can be 'locked' such that they will not be modified by the optimization. This feature allows for considerable flexibility in the use of the optimizer. For example, an existing set of stimuli can be used as a baseline condition, and new conditions with specified relationships to that baseline can be created. Alternatively, the optimizer can be used to find replacements for a few experimental items in an otherwise acceptable set of items based on the results of a pilot study. The use of a locked set is summarized in an example case in the main manuscript and in additional example cases included as part of the SOS online manual.

Step 4: Search for items in the population and in the sample that could be swapped to better satisfy the constraints

Once the constraints have been specified and an initial sample of items has been drawn, the search for an optimal set of items can begin. In what follows, we present the simplest form of a search in SOS — a "greedy" optimization of stimuli. Greedy optimizations always seek to swap out existing items in a sample with items that better satisfy the constraints. Although this greedy search has benefits in terms of speed, it is not formally guaranteed to find an optimal or near-optimal set in all situations. We then describe how this problem can be overcome by relaxing the deterministic nature of the greedy optimization and allowing swaps to be influenced by random noise—that is, by allowing for the stochastic optimization of stimuli as in a classic stochastic relaxation search (Kiefer & Wolfowitz, 1952).

Representing the optimization problem as movements in state space

To understand how the optimization algorithm operates, it is useful to conceptualize the search for optimal stimuli as a search in *state space* (Rumelhart, Smolensky, McClelland, & Hinton, 1986). As an example of a simple state space, consider a researcher who has word frequency

Table S.2: Example ‘population’ data set containing a single global minimum when attempting to minimize the variability in the frequencies of a ‘sample’ containing two items

Item	Word	Frequency
1	cow	8
2	pig	10
3	cat	11
4	dog	13

ratings for a population of four items listed in Table S.2. The researcher’s goal is to fill each of the two “item-slots” in the sample with the two items that together have the lowest variability. For simplicity, this constraint can be operationalized as the following simple cost function based on the standard deviation (std) of the sample based on the data in the frequency column, **freq**:

$$O_{MINstd}(\mathbf{freq}) = (std(\mathbf{freq}))^2$$

(dditional details on how this constraint can be instantiated in SOS are described in the cost functions appendix of the main article.

The representation of state space consists of the following. First, each combination of items that can be generated from the population is referred to as a *state*. Each state is represented by a location in space. Each dimension represents the “slot” in the condition that must be filled with a sampled item, and one level on each dimension represents every stimulus that could be used to fill that slot. In our example, the state space corresponds to a 4x4 plane. Two dimensions correspond to the two slots to be filled, and the four levels on each dimension correspond to the four items that could be used to fill these slots. Let us further stipulate that once an item has been used to fill a slot, it cannot be used to fill another slot (i.e., sampling occurs without replacement). This requirement effectively reduces the plane from 4x4 to 4x3, since the item used to fill the first slot cannot be used to fill the second. An illustration of this procedure is presented in Figure S.2.

Next, an additional dimension is added to represent the cost associated with each state in space. Higher costs are represented as higher elevations, and lower costs as lower elevations. Within this framework, searching for an optimal set of items amounts to moving in this space towards the state with the lowest cost.

In this example, state space is a cuboid, with two dimensions for each of the slots in the sample that must be filled and one dimension for cost. But, for visualization purposes, we can alter this geometric depiction slightly to allow the first of the “slot” dimensions to represent each of the four possible items directly and the second “slot” dimension to represent the three neighboring items. These neighbors are denoted as n_1*,* n_2*, and* n_3, in reference to their rank-ordered position in the table once the first item has been removed, as illustrated in Figure S.2. For instance, relative to the items *cow, pig, and cat*, the third neighbor would be *dog*; in contrast, the third neighbor of the *dog* item would be *cat*. Now, the cost for each state can be plotted as a cost surface above the plane formed by the intersection of the different combinations of items, as depicted in Figure S.2. Having created the state space, all that remains is to derive a means of moving to the point with the lowest cost.

	Cow	Pig	Cat	Dog
Cow	-	2.0	4.5	12.5
Pig	2.0	-	0.5	4.5
Cat	4.5	0.5	-	2.0
Dog	12.5	4.5	2.0	-

	N1	N2	N3
Cow	2.0	4.5	12.5
Pig	2.0	0.5	4.5
Cat	4.5	0.5	2.0
Dog	12.5	4.5	2.0

Figure S.2. Upper section: Two-dimensional depiction of the state space formed by each of the different combinations of pairs of items listed in Table S.2. The values in each cell of this table correspond to the cost associated with each combination of items when trying to minimize the variance in a sample of two items. The diagonal is blank due to the restriction that items are sampled without replacement. Lower section: A reduced version of the table from the upper section where the data from the upper triangle has been shifted to the left to merge with the lower triangle. The labels for the vertical axis of this table remain unchanged, whereas the labels for the horizontal dimension now correspond to the neighboring items of the items listed on the vertical axis.

Greedy optimization of stimuli

The greedy version of the optimization algorithm is the simplest way to navigate to a state with lower cost. This process begins by calculating the cost of the random location at which the algorithm is initialized. Next, the algorithm randomly selects a condition and an item in that condition for possible replacement. In state space, this corresponds to selecting one of the slots and then allowing the level (item) on that dimension to be changed. The values of all other slots (items in the set) are held constant. A replacement item is then selected from the population (refered to as a *population feeder* in the software) or, additionally, from one of the other samples of items (refered to as a *sample and population feeder* in the software). This candidate replacement item is the level of the selected dimension to which the algorithm would move if the swap actually took place. Having selected a pair of items that could be swapped, the cost associated with the swap is then computed. If the cost of the swap set is lower than the cost of the current set, the swap set is adopted as the

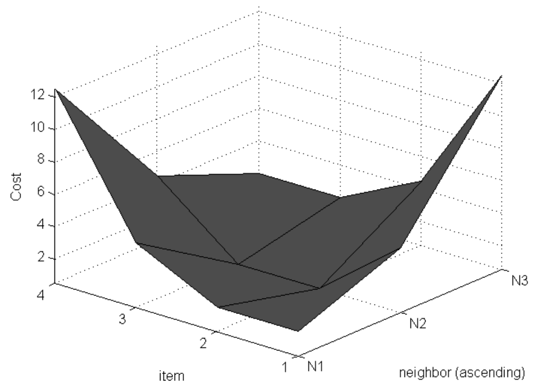

Figure S.2. Three-dimensional representation of the data from the lower section of Figure S.2. The plane at the base of the plot corresponds to the different combinations of items that can be formed from the data listed in Table S.2. The "item" axis corresponds to items 1-4 from the table directly, whereas the "neighbor" axis corresponds to the neighbor items of the item selected on the "item" axis (explained in detail in the body text). The "cost" axis represents the cost associated with each state, or combination of items, that could be formed from the population of items being sampled from. Note that the shape of the cost surface is concave when viewed from above and consists of a minimum valley surrounded by hills of monotonically increasing cost. The bottom of this valley corresponds to the global minimum of the state space, and to the pair of items with the lowest variability in frequency values.

current set; otherwise, the current set does not change.[1] This swapping process is then repeated to eventually arrive at a cost minimum where swapping out any item in the sample actually leads to higher cost. We refer to each attempted swap as an *iteration*.

The greedy version of the algorithm has several useful properties. First, the cost of the swap set can be computed very efficiently by means of a *local update*. In effect, the contribution of the current item in the sample can be removed from affected cost functions, and the contribution of the swap item can be added to affected cost functions; there is no need to effectuate a global update which requires a calculation based on the full set of data. A second benefit of the greedy search is that it can operate without knowledge of the full topography of the state space. It is usually not feasible to collect such knowledge given the impracticality of exhaustively searching for an optimal set. A greedy optimization, however, can begin its operation from a randomly selected location (i.e., a random sample of items) and lower cost without knowing the full cost topography. The greedy algorithm is somewhat akin to a nearsighted person: it only needs to "see" cost at a neighboring point in state space to decide whether the swap should be made or not. A third advantage of the greedy search is that it rapidly descends in cost space towards a minimum, efficiently focusing its search on the regions of space that are most likely to contain minima. It does so by incrementally ruling out many regions of the cost space for further exploration. Specifically, the algorithm will never consider states of items that require two subsequent swaps that both increase cost relative to

[1] As discussed in detail in the cost functions appendix of the main manuscript, which focuses on the details of the cost functions themselves, the SOS software implements two types of constraints, "hard constraints" and "soft constraints". The same swapping rule described here applies to both types of constraints, but minimizing hard constraints takes precedece over minimizing soft constraints (i.e., a swap will occur if the cost associated with the hard constraints is reduced even if this also leads to an increase in the cost associated with the soft constraints).

the current position in space. For instance, in the four-item example, it would be possible, just by chance, to start the search by selecting the maximally variable sample of *cow* and *dog*. A greedy search would quickly replace these items with a minimally-variable sample containing *pig* and *cat*, and, absent some additional stopping criterion (discussed later), would continue to try (and fail) to replace either *pig* or *cat* with *cow* or *dog*. Relevant to the present discussion is the fact that the algorithm would never re-examine the *cow* and *dog* sample because to do so would require an intermediate uphill step in state space followed by a second attempted step uphill. When dealing with larger populations of items, this property of the greedy algorithm allows more attempted swaps to take place for stimuli that are more likely to lead to the optimal set.

Taken together, the greedy search possesses many of the characteristics of an ideal search algorithm. It is computationally efficient and capable of examining large numbers of sets. Its deterministic cost-reduction rule effectively constrains the number of sets to examine and reduces the time spent exploring less optimal sets than those it has already identified. In many circumstances, the greedy version of the algorithm may, in fact, be sufficient for optimizing stimulus sets. This possibility is always assessed in practice in SOS because the results of a greedy optimization are useful in configuring the parameters of the more sophisticated stochastic algorithm, and so a greedy optimization is usually run first. Nevertheless, the greedy search suffers from a fundamental limitation, as we discuss next.

Greedy optimization's strength is also its weakness—or, the pitfalls of local minima. As noted previously, one advantage of the greedy optimization is that it is only capable of descending to states with lower cost. This feature allows the greedy version to rapidly converge on a set with lower cost and to avoid exploration of parts of the space that are unlikely to contain good solutions. Insofar as the cost topography is perfectly concave, the greedy version is an ideal way of identifying the point with the lowest cost. In practice, however, the cost space is often not perfectly concave. Instead, it may be formed of several unconnected lower-cost valleys separated by higher-cost hills. Since greedy optimizations can only travel downhill, once a valley has been entered, the algorithm will inexorably move towards the valley's floor. The valley floor is a local cost minimum since all of the surrounding states are of higher cost. However, the cost here may still be much higher than the global minimum cost value, present in another valley in the space. Greedy optimization lacks a way to climb over the higher cost barriers that surround the local minimum in order to reach the global minimum.

As a concrete example of this problem, consider the state space that results if the researcher attempting to minimize the variability in a sample's word frequency must select sample items from a slightly different population listed in Table S.2. This population's state space is plotted in Figure S.2. Inspecting this space reveals that the set of items with the lowest variability (*donkey* and *horse*) occurs in one corner of the space comprised of the highest frequency items. This global minimum in cost space is separated from a local minimum formed by the lowest frequency items *mule* and *goat*—which has a relatively low cost value but is still higher than the global minimum—by a barrier of higher cost. This is because there is very high variability in all of the samples comprised of intermediate-frequency items. If a greedy optimization begins at a point within the valley of the local minimum, it will be incapable of reaching the global minimum. A simplified depiction of the local minimum problem is presented in Figure S.2, which roughly corresponds to a multidimensional scaling of the present problem into two dimensions.

Local minima are likely present in the cost topographies of many different optimization prob-

Table S.2: Example 'population' data set containing a global minimum and a local minimum when attempting to minimize the variability in the frequencies of a 'sample' containing two items

Item	Word	Frequency
1	mule	1
2	goat	6
3	donkey	12
4	horse	13

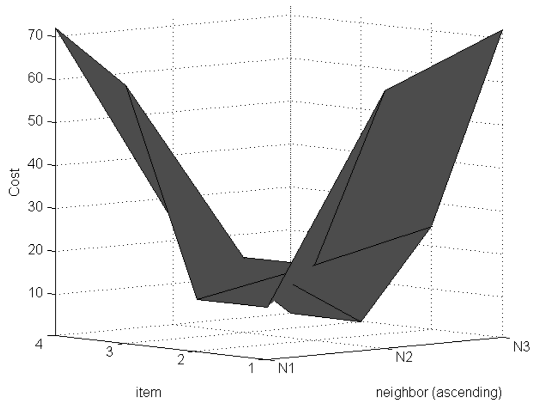

Figure S.2. Depiction of a state space containing both a local minimum (foremost region of the plane) and a global minimum (backmost region of the plane) separated from one another by a 'hill' of higher cost. The cost function used to generate the space was identical to that used for Figure S.2, applied to the data in Table S.2.

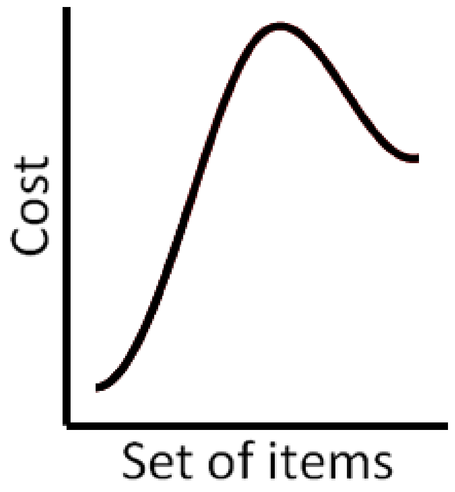

Figure S.2. Approximation of the state space illustrated in Figure S.2 after a multidimensional scaling has been applied to the base plane on that figure to reduce it to a single dimension.

lems, and their presence and locations are rarely known precisely because the entire cost topography usually cannot be computed. As a general guideline, the likelihood of becoming trapped in local minimum increases as sample size increases, as population size decreases, and as the number and complexity of the constraints increases.

Stochastic optimization of stimuli

The problem of local minima precludes greedy optimization as a general method for discovering optimal stimuli. However, the idea of constraining the search to the regions of state space that are most likely to contain a minimum often leads to succesful optimizations that are superior to the sets discovered by a purely random search of the space. By interpolation, employing a form of stochastic optimization search, which involves both a greedy-like bias on swap decisions combined with a random noise term, is a logical solution to these problems (e.g., Kiefer & Wolfowitz, 1952). In detail, our implementation of a stochastic algorithm consists of replacing the deterministic cost-reduction rule from the greedy optimization with a stochastic choice that can be biased to keep or swap items on the basis of the relative cost difference, $\Delta cost$, between the current set and the swap set (Hinton & Sejnowski, 1986). Formally, this choice is made by passing $\Delta cost$ through a sigmoid function with a temperature parameter, T, which has a lower bound of zero. Temperature specifies the steepness of the slope of the linear portion of the sigmoid in the range surrounding zero and, in conceptual terms, determines the degree to which $\Delta cost$ can bias the probability of staying with the current set or moving to the swap set:

$$p_{swap}(\Delta cost, T) = \frac{1}{1+e^{\frac{-\Delta cost}{T}}} \tag{S.3}$$

The role of the temperature parameter is to determine how strongly differences in cost between the current state and the swap state are able to bias the likelihood of making a swap. More formally, the likelihood of swapping to a particular state follows a Boltzmann distribution and is determined by the $\Delta cost$ of the swap state relative to the current state, as well as the temperature. Figure S.3 illustrates this relationship in several different sigmoid functions generated with various temperature parameters. When temperature is very high, the sigmoid collapses into a line with a slope of zero. At such high temperatures, even large cost differences are unable to bias the likelihood of making a swap, and swaps are made at random. When temperature equals zero, the sigmoid function collapses into a step function and the algorithm makes identical swap decisions as in the greedy algorithm. At intermediate temperatures, however, the likelihood of making a swap is biased towards lowering cost without necessitating that all steps always lower cost—all possible swaps are associated with a non-zero likelihood and could potentially be made. As a result, as long as temperature is greater than zero, this algorithm is guaranteed, in a formal sense, to be able to move to the global cost minimum, provided that the optimization is run for a sufficiently long period of time.

Although this formal guarantee offers a basic assurance that the algorithm is capable of finding an optimal set, in practice, selecting an appropriate temperature for a given optimization problem poses a challenge. If the temperature is too high, the algorithm will effectively be performing a random search of state space. If temperature is too low, then the algorithm performs a greedy optimization. As we have discussed previously, neither of these options are ideal. It is therefore useful to consider the lower and upper bounds for the range of temperatures that allow stochastic optimization to operate effectively.

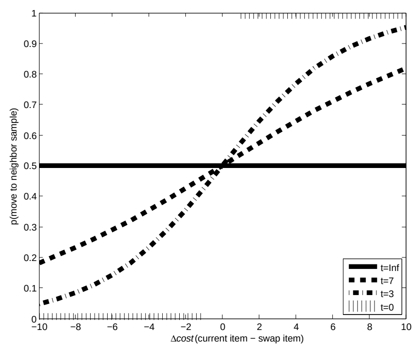

Figure S.3. Depiction of the likelihood of moving to a neighboring sample in state space by swapping two items based on the cost difference, $\Delta cost$, between the two items that could be swapped, for a range of temperature values. Inf = infinite.

To determine the effective upper bound on temperature at which nearly all swaps are random, it suffices to allow the algorithm to explore the different states in state space completely randomly (e.g., by effectively setting temperature to ∞) for a block containing some prespecified number of iterations (1000 generally works well). As the algorithm randomly explores the space during this block, the $\Delta cost$ values that result from each attempted swap are recorded. At the end of the block, temperature can then be set to a high enough level that even the most extreme $\Delta cost$ values exhibit only a modest bias on the likelihood of swapping two items. After experimentation with several different values, we have found that setting the upper bound on temperature to the range of $\Delta cost$ values that encapsulates 95% of swaps (specifically, the 97.5th - the 2.5th percentiles) during this block of random swaps is an effective value. This is a reasonable approximation of the upper bound of temperatures which allow $\Delta cost$ to affect the likelihood of making a swap which is also fairly consistent across optimizations (the range of $\Delta cost$ values being far more variable across different optimizations).

Determining a lower bound on temperature involves running the greedy version of the algorithm until it becomes trapped in a minimum. At this point, we infer that the reason the algorithm became trapped was because all of the swaps that were attempted while in the minimum were of a higher cost than the current state. Without knowing whether this minimum is local or global, a conservative assumption would be to assume that it is a local one and that the algorithm would have been able to escape it by making one or more uphill swaps. To escape from the minimum, it would therefore be necessary to raise temperature to a point at which a small subset of the attempted swaps are substantially influenced by the stochastic process. This change would allow the algorithm to move uphill to these items and potentially escape the minimum. By recording the $\Delta cost$ values across the last block of iterations (typically 1000) of a greedy search, the sampling distribution of $\Delta cost$ at various percentiles can be examined (all of which, by definition, are greater than zero, since no swaps were able to further reduce cost). Then, temperature could be set to the $\Delta cost$ for a given percentile to allow all of the attempted swaps below that percentile to have a substantial likelihood of being influenced by the stochastic process and thus able to lead to uphill movements. The specific percentile that is selected as the lower-bound on temperature reflects a trade-off between how greedily the algorithm behaves and how likely it is that the algorithm will be able to avoid a local minimum. After running a wide variety of optimization problems, we have found that selecting the $\Delta cost$ from the 10^{th} percentile has generally been a good starting value, although a larger value (e.g., from the 50^{th} percentile) may lead to a more optimal set, albeit after many more iterations.

Adding a temperature annealing schedule for fast, reliable optimizations. Having established the lower and upper bounds for temperature, the next question to ask is whether the temperature used in an optimization should be nearer to the upper bound or to the lower bound. In an ideal situation, the goals of the stochastic optimization algorithm should be to allow for 1) stochastic movement that travels uphill in cost space when stuck in a local minimum and 2) rapid, greedy movement once the algorithm has entered the valley that contains the global minimum. A gradual annealing process[2] which lowers temperature over time achieves both of these goals (Kirkpatrick et al., 1983). This allows for an early period of exploration of a variety of different states and gradually eases into the global minimum (Hinton & Sejnowski, 1986).

To understand how the annealing process accomplishes this feat, consider the ball-in-a box metaphor of temperature annealing outlined by Hinton and Sejnowski (1986). Imagine that a model of the cost topography shown in Figure S.2 was placed in a box. Next, a ball is placed into the box. The ball's location corresponds to a particular point in state space. The box is then sealed so that the topography cannot be seen. Now, imagine that someone who has not seen the topography is instructed to move the ball to the lowest point in the box—the global minimum. Without knowing the topography, this person must therefore rely on two known principles and the gradation between them to move the ball to the global minimum. If the box were shaken vigorously, the ball would bounce about and potentially visit every possible location (state) in the box (stochastic search with high temperature). If the box were set still, the ball would roll downhill to the bottom of the closest valley (greedy search).

By starting off by shaking vigorously and then gradually shaking the box less and less, it will be possible to move the ball to the global minimum. Shaking the box vigorously initially allows the ball to avoid becoming trapped in any local minimum. By incrementally decreasing the vigor with which we shake the box (by decreasing temperature), a useful property emerges: at some point, the shaking will be providing the ball with enough energy to bounce out of a local minimum, which is by definition shallower than the deeper global minimum, but insufficient energy to move uphill from the global minimum back into the local minimum. It is therefore possible to ensure that the

[2]Movellan and McClelland (1993) note that there is a slight distinction between simulated annealing (Kirkpatrick, Gelatt, & Vecchi, 1983) and simulated sharpening (Akiyama, Yamashita, Kajiura, & Aiso, 1989), and our implementation of annealing is actually closer to the latter. However, we expect that the annealing metaphor is more intuitive and have therefore overlooked this distinction in the present work.

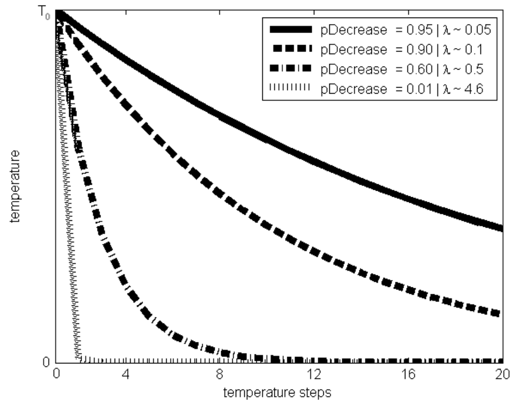

Figure S.4. Depiction of the temperature values after a given number of temperature steps away from an initial temperature, T_0, as a function of a range of pDecrease and λ values.

ball effectively becomes contained within the global minimum to an arbitrarily high probability.

In cost topographies that contain many hills and valleys, gradually lowering temperature also ensures that the largest hills are excluded from being visited before the smaller hills. This results in the early elimination of the greatest violations of the constraints. The algorithm then proceeds to deal with the more fine-grained and subtle topographic differences that exist amongst the remaining states once these larger differences have been ruled out.

To implement these ideas formally, we selected an exponential decay function. This equation has been shown to be successful in the annealing of connectionist networks, which are close relatives to SOS (Hinton & Sejnowski, 1986) and represents the optimal annealing function when such networks are allowed to learn their own annealing functions via gradient descent (Ackley, Hinton, & Sejnowski, 1985; Plaut, Nowlan, & Hinton, 1986). In the decay function, temperature, T, is determined by three parameters: The first parameter is the initial temperature, T_0, which corresponds to the $\Delta cost$ at the (97.5th – 5th) percentile of the sampling distribution of $\Delta cost$ during the initial block of completely random swapping. This temperature exponentially decays towards 0 as a function of the second parameter referred to as the decay rate, λ, and the third parameter, which specifies how many times temperature has been decreased (we refer to these as temperature steps, with the number of steps denoted as *nSteps*). Each step typically contains at least 1000 iterations, for reasons discussed in detail later.

$$T = T_0 \cdot e^{-\lambda \cdot nSteps} \tag{S.4}$$

Plots of the temperature values for a given number of steps were generated using different values of the λ parameter (and a corresponding *pDecrease*, discussed later) and are presented in Figure S.4. Note that higher values of λ lead to more drastic temperature decreases at each step, whereas lower values of λ lead to more gradual temperature decreases. By varying λ, it is therefore possible to trade off making a rapid transition from random to deterministic behavior—which leads to faster settling into a minimum—and making a slow transition—which ensures that the algorithm does not get stuck in a local minimum.

Two factors complicate the intuitiveness of the λ parameter in the previous equation. First,

λ is an unbounded parameter which affects temperature in a nonlinear fashion. Second, whether a given λ leads to a very aggressive or very gradual annealing schedule is determined by the specific optimization problem at hand. The latter point is the result of differences in the "elevations" of the minimum and maximum cost values across optimization problems, leading to different effective temperatures at which the optimizer will be making swaps at random versus only making swaps that lower cost. We have attempted to address these issues so that users of the SOS software can more readily select an appropriate aggressiveness for the decay function as follows: Instead of specifying λ directly, the current implementation of SOS allows for the indirect specification of λ via several other parameters. We explain this procedure in detail in the next section, although in practice helper functions are used to conduct these calculations and thus very little user interaction is needed.

The parameters needed to calculate λ indirectly are the initial temperature at which the system shows random behavior, T_0, the final temperature at which only a pre-specified portion of swaps will be made randomly (for the rest, the algorithm will basically operate in greedy mode), T_{final}, and the number of steps, *nSteps* that are needed for temperature to decrease from T_0 to T_{final}. The first two parameters correspond to the upper and lower temperature bounds outlined previously. The *nSteps* parameter then determines how quickly temperature passes from the upper bound to the lower bound. The choice of the number of steps to use reflects a compromise between minimizing run-time and avoiding local minima. At the very least, three steps should be used, as the first step will correspond to a random search and the last step will be similar to a greedy search. As a guideline, we have found that at least 10 steps should be used in most optimizations, and more steps tend to lead to better optimizations. Once T_0, T_{final}, and *nSteps* have been determined, these values can be substituted into Equation S.4 to determine λ. Having determined λ, the fully specified exponential decay equation can then be used to calculate the proportion of the temperature that will be subtracted from the temperature at step n to determine the temperature at step $n+1$. We refer to this proportion as pDecrease.

In practice, with the help of this indirect equation and the helper functions in SOS, all a user must do to configure the decay equation is as follows: First, run the optimizer in greedy mode to see if it can find a sufficiently optimal set. If it fails, the $\Delta cost$ value from the desired percentile (e.g., the 10th) should be recorded, and the user can pass this information, along with the desired number of steps (e.g., 10), to a helper function that calculates pDecrease. Once pDecrease is set, the decay equation (and by proxy, λ) can be configured automatically, and a stochastic optimization can begin.

Using a proxy of thermal equilibrium to assess when temperature should be lowered. Having decided to use a gradually decreasing temperature value, one question still remains: When should temperature reductions occur? After some experimentation with other schedules for decreasing temperature, we have found that it is best to do so only once sufficient evidence has accumulated so as to reliably estimate that the position of the current set of stimuli in state space no longer falls within a local minimum from which it could not escape at lower temperatures. This can be estimated by considering when the optimization at a particular temperature has reached a state referred to as *thermal equilibrium*.

Thermal equilibrium is the point at which the probability of swapping to a given item has stabilized at a particular value across all items. To understand how this measure could serve as a basis for assessing when to lower temperature, consider once more the simple state space with a single local minimum presented in Figure S.2. Assume for the moment that temperature has been set a little above the level needed to move from the valley containing the global minimum to the valley containing the local minimum. Also assume that the optimization has been run for a sufficiently long amount of time to determine the probability of swapping a particular item into the sample set. At this point in time, the probability distribution for the sets in the valley of the global minimum will be higher than those in the valley of the local minimum since $\Delta cost$ should be substantially biasing the likelihood of making swaps into and out of each valley. Nevertheless, in both cases, these probability distributions will contain values that are non-trivially above zero.

Next, imagine that temperature is lowered so that it is just possible to leave the valley containing the local minimum and enter the valley containing the global minimum. Once in the valley containing the global minimum, the relatively low temperature will provide a substantial bias to continue moving downhill in this deeper valley. As cost descends further, the likelihood of making a swap back into the valley containing the local minimum gradually moves towards zero. If the optimization continues at this new temperature for some period of time, the probability distributions for swapping to each item will eventually stabilize once more. The new probability distribution will differ from the old one, however, because some of the probabilities (i.e., those contained in the local minimum) will be lower in the new distribution, whereas others (i.e., those in the global minimum) will be higher. At this point, temperature could be lowered again to hone in on the global minimum more quickly.

Of course, if the temperature is still substantially above the level needed to probabilistically exclude a portion of the state space containing a local minimum (e.g., if temperature were still an order of magnitude above the temperature value described in the previous example), these probability distributions might not differ substantially before and after temperature is changed. Thus, one cannot simply compare the probability distributions before and after the change in temperature to infer if temperature can be lowered. Instead, it is necessary to estimate the probability distribution repeatedly at a particular temperature and lower temperature only when consecutive probability distributions are nonsignificantly different from one another. When this is true, thermal equilibrium has been reached. To maximize the accuracy of this inference and to ensure that the distributions have, in fact, stabilized, it is necessary to try many attempted swaps before each assessment of whether thermal equilibrium has been reached so as to allow the distributions to become stable.

In principle, examining the stability of the probability distribution of swapping to different items is a good way to assess thermal equilibrium. In practice, however, it can take very large blocks of iterations before the probabilities of swapping to each item stabilize. To reduce run time and memory use, it is far more efficient to monitor a proxy of the probability distribution—the distribution of cost values. This measure provides a summary of the probability of swapping to a given item based on the cost value associated with swapping to that item, averaged across all items, and is already calculated as part of the optimization. The cost distributions for two subsequent blocks of trials can also be readily compared using t-tests, such that thermal equilibrium is said to have been reached once a standard statistical criterion has been exceeded (e.g., $p > .05$).[3]

[3]Empirically, the distribution of cost values typically deviates somewhat from some of the assumptions underlying a t-test (e.g., normality). However, the goal of this comparison is to provide a quick and relatively reliable means of assessing when thermal equilibrium has been reached, and t-tests are more than adequate for this task.

Step 5. Evaluating how well the constraints have been satisfied and stopping the optimization

The search for an optimal set of stimuli can be terminated automatically when one of several conditions is met. The first consists of passing statistical tests specified by the user. These tests assess when the constraints have been satisfied to a desired threshold. There are currently three main types of statistical tests available for use in SOS: one- and two-sample t-tests (paired and independent), correlation-matching tests, which test whether a correlation in the sample matches a target value (typically 0 to eliminate correlations between independent variables and boost power in a regression), and a Kolmogorov-Smirnov test for assessing the uniformity of a distribution. These parametric statistical tests are approximate analogues to the distance minimization and maximization constraints, correlation-matching constraints, and entropy constraints, respectively. For instance, if the goal is to make two conditions maximally different on word frequency, a user could specify that a t-test on the word frequencies across the two conditions should return a t-statistic associated with a p-value that is less than .05. Similarly, if another goal is to match the conditions on word length, the user could specify that a t-test across the word length data in the two conditions should return a t-statistic associated with a p-value greater than .5, which reflects a failure to reject the null hypothesis. Different critical p-values may be desirable both depending on the importance of satisfying different constraints during the optimization and the specific parameters of the constraint in question. For instance, the statistic used to assess whether a sample correlation matches a target (population) has larger confidence intervals when testing whether two near-zero correlations are matched than when two large correlations are matched because, by definition, there is less unexplained variance in the latter case. The test of uniformity of distributions is also capable of detecting small deviations from uniformity that may not be particularly important in many situations. Using a smaller critical p-value (e.g., .001) when testing whether a sample distribution is uniform and inspecting the plots of the distributions (which can be accomplished via the SOS software) when the test is not passed to determine if they are due to a failure to distribute items across particular ranges is therefore recommended.

The optimization terminates once all of the statistical tests exceed the user-specified criteria. Because these probabilities generally do not change drastically after a single iteration, statistical tests are only run periodically (the default is every 1000 iterations). Note, however, that blind reliance on these statistical tests may in some cases be misleading because statistically significant differences are sometimes detected for extremely small effect sizes (e.g., a word frequency difference of 0.1 can be significant between two conditions because the variability within the conditions is .001). This is most frequently the case in the context of pairwise minimization constraints, which in minimizing pairwise differences also minimize the variability in these differences, which in turn leads to an apparent lack of improvement or worsening of the match. Instead, failures to satisfy the statistical tests when the optimization ends for one of the other reasons discussed next typically warrants inspection to confirm that the statistical differences between the conditions are theoretically important. If appropriate, a user can also opt to set a minimum difference threshold when configuring the tests. This allows the statistical tests to "pass" once the difference between the two conditions has decreased below this bound, independent of statistical significance. A special print-out is displayed when tests pass based on the threshold instead of the statistical criteria.

The second condition that can cause the termination of the search is when the sample has reached a "frozen" state. That is, cost has not changed for a specified number of iterations (the default is 10,000). Note that swaps may still be occurring between items with the same impact on overall cost (i.e., items for which a swap would generate a $\Delta cost$ of exactly zero). Frozen states should generally only occur if the search has become trapped within a minimum and further descents in cost are no longer possible. Using a very gradual annealing schedule and allowing the optimization to reach a frozen state in stochastic mode should therefore correspond to finding a global minimum in the search space. If the only goal is to identify the set(s) of items that best satisfies the constraints, then this is the preferred mode for terminating the search. However, letting the optimization reach a frozen state may not necessarily be desirable if the goal is to use the selected items to make broad generalizations to other items, as discussed in the next section.

The third stopping condition is when a pre-specified number of iterations has elapsed, which prevents the optimization from running for an indeterminate amount of time. Generally, users should set this value to as high a value as they are willing to wait for an optimization to finish. The software defaults to a relatively small number of iterations (10,000) so that users can confirm that their optimization procedure is roughly behaving as it should before increasing the value of this stopping condition. We recommend a value of at least 500,000 and possibly into the tens or hundreds of millions depending on the complexity of problem being run, the speed of the computer, and the patience of the experimenter. If this stopping condition is reached, the experimenter can then consider whether the resulting set satisfies their constraints or if an additional pressure (e.g., a more aggressive annealing schedule) is needed to encourage the optimization to reach a frozen state before this amount of time has elapsed.

Step 6. Assessing the degree to which the items in a sample represent the broader population

Although one of the principal goals of many researchers is to infer the results of an experiment using a particular sample of stimuli to a broader population, present practices in stimulus selection place virtually no emphasis on assessing how representative a sample is of the population. Consequently, the final step in the optimization procedure does not have a clear analogue in many experimenters' stimulus selection process. However, constraints imposed on selecting experimental items can effectively restrict the sampled items to a very unusual subpopulation or even a uniqe set of items. At that point, stimuli should be treated as fixed as opposed to random effects and cannot be used for the purpose of statistical inference to other stimuli (Hino & Lupker, 1996). Clearly then, failing to assess the generalizability of stimulus sets is a critical, yet overlooked, aspect of experimental design. Such a state of affairs is quite paradoxical given the strong emphasis that is placed on reporting statistical inferences that generalize across items, as it is unclear whether such inferences are appropriate (e.g., by-item and by-subject analyses, F-max, Clark, 1973; mixed-effect regression, Baayen, Davidson, & Bates, 2008).

Automated optimization procedures do, however, offer a means of assessing generalizability. The simplest of these methods involves a direct measurement of how many different items could have been used to satisfy the constraints imposed by the user (i.e., the size of the effective population). This is achieved by running the optimization several times and comparing the sets that satisfy the constraints. If many different sets of items can be produced which all satisfy the constraints, there is a basis to generalize the particular set of findings to a broader population of items; otherwise, the sample cannot be used as a basis for statistical inference to other stimuli. As a general guideline, we recommend that if the average overlap between sets of stimuli is below 25%, generalizability is likely not an issue. Researchers should typically be concerned about failures to generalize when the average overlap exceeds 50%, although the details of the optimization problem and indended generalization may lead a researcher to adopt somewhat different thresholds.[4]These additional searches may not even take a noticeable amount of time to run on a modern multi-core processor, as many optimizations can be run simultaneously.[5]

The procedure outlined above is likely suitable for assessing generalizability in the majority of cases, where the main objective is to assess whether a relatively broad population of items meets the constraints. However, in some cases a researcher might be particularly concerned with garnering some stronger assurance that the actual population is very well represented in the sample of items. In this case, soft entropy constraints can be used to ensure that the breadth values of a particular variable across the entire population are sampled. This is arguably the best way of ensuring the external validity of the sampled items but may make it difficult to simultaneously satisfy other constraints.

The overall optimization process thus explicitly reflects a trade-off between internal and external validity, and different investigations may warrant different trade-off schemes. At the very least, SOS makes the assessment of this trade-off and of the effects of re-weighting the importance of these two factors a relatively easy and effortless task. This possibility, in turn, should encourage a consideration of these issues when selecting stimuli for an experiment.

[4]It is worth noting that in the details, the average amount of overlap may be too coarse a measure for making a fine-grained inference about generalizability, because it may be possible that a very small set of items which do overlap across the different samples are necessary to satisfy the constraints in all of the optimizations. A simple way of assessing this possibility is to save the residual population after an optimizated sample has been discovered (methods of doing so are described in the user manual) and reruning the optimization sampling from the residual population. Alternatively, and more rigorously, users could essentially set up different variants of the same optimization and have them all run simultaneously while sampling from the same population. The latter is in fact the more formally appropriate method of ensuring the generation of equally comprable samples as all of the samples effectively had equal access to all of the items. However, given current practices in the field, any assessment of generalizability would represent a substantial advance.

[5]This can be accomplished either by launching the SOS executable multiple times or by running SOS using a MATLAB installation that includes the Parallel Computing Toolbox. Note that a different random seed is needed when initializing each optimization.

REFERENCES

Ackley, D. H., Hinton, G. E., & Sejnowski, T. J. (1985). A learning algorithm for boltzmann machines. *Cognitive Science*, *9*(1), 147–169.

Akiyama, Y., Yamashita, A., Kajiura, M., & Aiso, H. (1989). Combinatorial optimization with gaussian machines. *Proceedings of the International Joint Conference on Neural Networks*, 533–540.

Baayen, R. H., Davidson, D. J., & Bates, D. M. (2008). Mixed-effects modeling with crossed random effects for subjects and items. *Journal of Memory and Language*, *59*(4), 390–412.

Clark, H. (1973). The language-as-fixed-effect fallacy: A critique of language statistics in psychological research. *Journal of Verbal Learning and Verbal Behavior*, *12*(4), 335–359.

Hino, Y., & Lupker, S. (1996). Effects of polysemy in lexical decision and naming: An alternative to lexical access accounts. *Journal of Experimental Psychology: Human Perception and Performance*, *22*(6), 1331–1356.

Hinton, G. E., & Sejnowski, T. J. (1986). Learning and relearning in boltzmann machines. In D. E. Rumelhart, J. L. McClelland, & the PDP Research Group (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 1: Foundations* (pp. 282–317). Cambridge, MA: MIT Press.

Kiefer, J., & Wolfowitz, J. (1952). Stochastic estimation of the maximum of a regression function. *Annals of Mathematical Statistics*, *23*(3), 462–466.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680.

Movellan, J., & McClelland, J. (1993). Learning continuous probability distributions with symmetric diffusion networks. *Cognitive Science*, *17*(4), 463–496.

Plaut, D. C., Nowlan, S., & Hinton, G. E. (1986). *Experiments on learning by back-propagation* (Tech. Rep. No. CMU-CS-86-126.) Department of Computer Science, Carnegie-Mellon University.

Rumelhart, D. E., Smolensky, P., McClelland, J. L., & Hinton, G. E. (1986). Parallel distributed models of schemata and sequential thought processes. In D. E. Rumelhart, J. L. McClelland, & the PDP Research Group (Eds.), *Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Vol. 2: Psychological and Biological Models* (pp. 7–57). Cambridge, MA: MIT Press.

van Casteren, M., & Davis, M. (2007). Match: A program to assist in matching the conditions of factorial experiments. *Behavior Research Methods*, *39*(4), 973–978.