Stochastic Synthesizer Patch Exploration in Edisyn

Sean Luke

George Mason University
Washington, DC, USA
sean@cs.gmu.edu

Abstract. Edisyn is a music synthesizer program (or "patch") editor library which enables musicians to easily edit and manipulate a variety of difficult-to-program synthesizers. Edisyn sports a first-in-class set of tools designed to help explore the parameterized space of synthesizer patches without needing to directly edit the parameters. This paper discusses the most sophisticated of these tools, Edisyn's Hill-Climber and Constrictor methods, which are based on interactive evolutionary computation techniques. The paper discusses the special difficulties encountered in programming synthesizers, the motivation behind these techniques, and their design. It then evaluates them in an experiment with novice synthesizer users, and concludes with additional observations regarding utility and efficacy.

Keywords: Synthesizer Patch Design · Interactive Evolutionary Computation

Edisyn is Java-based open-source patch editor library for hardware and software music synthesizers. A *patch* is the traditional term for a particular set of parameters which, when programmed into a music synthesizer, enables a performer to play a particular sound. That is, a patch is the music synthesizer equivalent of setting certain stop knobs or drawbars on an organ. Many synthesizers have poor interfaces or are otherwise difficult (and in some cases impossible) to program directly, making attractive the notion of a *patch editor*, piece of software which enables the musician to more easily program these devices remotely via his laptop. Edisyn provides patch editors for a variety of difficult-to-program synthesizers. I am Edisyn's software developer.

Edisyn's particular strength lies in assisting the musician in exploring a synthesizer's parameter space. Indeed, I believe Edisyn is easily the most capable general-purpose synthesizer patch editor available in this respect. This paper discusses two of the most sophisticated patch-exploration tools in Edisyn's toolbox: its stochastic hill-climber and constrictor facilities. Both of these tools apply an interactive version of evolution strategies (ES) to do this assisted exploration; but the two tools have very different philosophies and heuristic approaches.

The hill-climber and the constrictor are both evolutionary computation (EC) methods, treating candidate synthesizer patches as the individuals in question. EC has been used for at least 25 years [1] to search the space of synthesizer patches, but the lion's share of this literature has been in *evolutionary resynthesis*, where an automated function assesses candidate patches based on desired predefined qualities. In contrast, Edisyn's hill-climber and constrictor are examples of *interactive EC* [2], where the assessment function is a human.

Interactive EC is not easy: humans are fickle, arbitrary, and easily bored. Whereas a typical (non-interactive) EC run may consider many thousands of candidate solutions, a human cannot be asked to assess more than a few hundred candidates before he gives up. Unfortunately, because synthesizer patches are high dimensional (10 –1000 parameters) and often complex, making progress in this space with a small number of presentations is hard. This problem is commonly known as the *fitness bottleneck* [3] in the interactive EC literature. Humans also have noisy preferences and often change their minds, and so while one might imagine that interactive EC would enable a human to search for a target sound fixed in his mind, in fact such tools are perhaps best used to help him explore the space. When the user comes across an "interesting" new sound, he may direct the system to start searching in that *direction* rather than towards some predefined *goal*.

The space of synthesizer patches presents its own special domain-specific difficulties. Patches take up time to be auditioned and heard, and hardware synthesizers are slow in receiving them over MIDI (the standard protocol for patch transfer): this restricts the rate at which patches may be presented to the user. Additionally, while synthesizer patches are typically fixed-length arrays (and Edisyn assumes this representation), different synthesizers have parameter spaces with a wide range of statistical linkage, numerosity, resolution, and degree of impact on the final sound. At present Edisyn supports almost 30 patch editors for 15 different synthesizers covering many synthesis modalities, including additive, subtractive, and FM synthesis, plus samplers and ROMplers. Creating a single general-purpose tool to explore all these spaces is not easy.

Finally, patch parameter types are heterogeneous, taking three forms. First there are *metric* parameters, normally integers, such as volume or frequency. Second, there are *non-metric* (or categorical) parameters, such as choice of oscillator wave type. Third, *hybrid* parameters have *both* metric and non-metric ranges: for example, a MIDI channel parameter may be set to 1...16, or to Off, Omni, MPE High/Low, etc. A general-purpose patch exploration facility ought to be able to accommodate any combination of these. (Incredibly, there also exist synthesizers with holes in their valid parameter ranges.)

1 Previous Work

The interactive synthesizer patch exploration literature has considered different ways to tackle some of these challenges. [4] focuses on interfaces designed to speed the assessment and selection of solutions, albeit with very small parameter spaces. To simplify the search space, [5] updates a parameterized model instead of a sample (essentially a kind of estimation of distribution algorithm). [6] essentially rounds candidate solutions to the nearest known patch drawn from a library.

Best known is MutaSynth, a metaheuristic patch-exploration tool which found its way into a commercial product (Nord's Modular G2 synthesizer editor) [7–9]. MutaSynth is manual in operation: the user selects a patch or two from a palette of "best so far" sounds, then chooses how to recombine or mutate them. The system then produces several recombined or mutated children, and the user decides whether to add any children to the palette. This requires the user to make several decisions per iteration, but ideally the "best so far" palette would maintain user interest and increase diversity, thus helping avoid getting stuck in local optima.

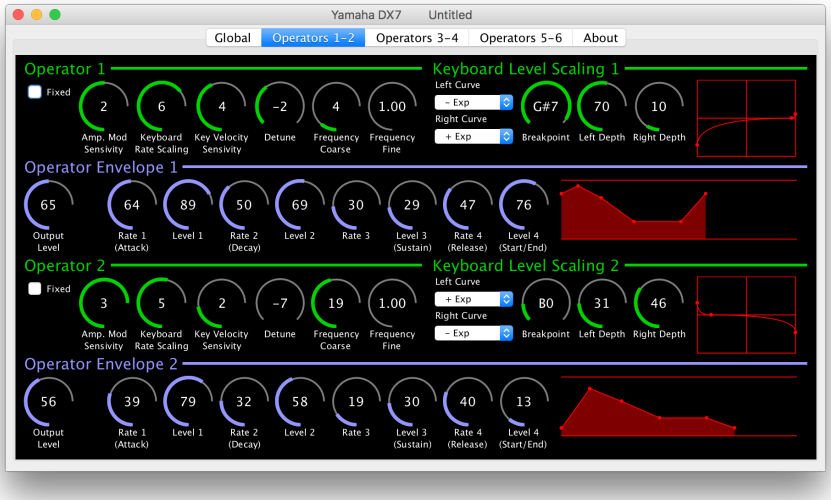

Fig. 1. Edisyn's Yamaha DX7 patch editor window, showing the tab for FM Operators 1 and 2.

Curiously, nearly all previous work (including MutaSynth and similar methods [10, 11]) problematically assumes that *all* parameters are metric, either by restricting them to be so or by shoehorning non-metric parameters into metric spaces. But often as many as half of the parameters in a typical synthesizer patch are non-metric or are hybrid.

While Edisyn and most other examples from the literature focus on fixed-length arrays (as the large majority of synthesizers use them), some recent interactive EC literature has begun to examine unusual variable-sized representations for modular synthesizers [12] or custom neural-network based synthesis techniques [13, 14].

2 Edisyn

Edisyn supports a variety of synthesizers, but places a special emphasis on synthesizers with difficult interfaces or difficult-to-program architectures. Of particular interest to discussion here are FM and additive synthesizers. Additive synthesizers are notorious for high dimensional patches: for example the Kawai K5000 has on the order of a thousand parameters. On the other hand, FM synthesizers tend to have parameters that are highly epistatic, difficult to control, and *very* difficult to predict. This is not the fault of the devices, but rather is due to the nonlinear nature of the FM synthesis process. Figure 1 shows one pane of Edisyn's editor for the Yamaha DX7, a famous FM synthesizer.

Like many patch editors, Edisyn provides many features to assist in directly programming these synthesizers. But Edisyn also provides a large assortment of tools to partially automate the search for patches of interest. Simple Edisyn tools include *weighted patch randomization*, where a patch is mutated to some degree; and *weighted patch recombination*, where two patches are merged, with an emphasis placed on one patch versus

Algorithm 1 Mutate($P, v, 0 \leq weight \leq 1$)

Input: Parameter P with current value v, *weight*
if P is metric **then**
  return MetricMutate(P, v, *weight*)
else if P is a hybrid parameter and v is a metric value **then**
  if coin flip of probability 0.5 **then**
    return MetricMutate(P, v, *weight*)
  else if coin flip of probability *weight* **then**
    return a random non-metric value in P
else if P is a hybrid parameter and v is not a metric value **then**
  if coin flip of probability *weight* **then**
    if coin flip of probability 0.5 **then**
      return a random metric value in P
    else return a random non-metric value in P
else if coin flip of probability *weight* **then**
  return a random metric value in P
return v

Algorithm 2 MetricMutate($P, v, 0 \leq weight \leq 1$)

Input: Parameter P with current value v; *weight*
$q \leftarrow weight \times (1 + \text{metric max of } P - \text{metric min of } P)$
$l \leftarrow \max(v - \lfloor q \rfloor, \text{ metric min of } P)$
$h \leftarrow \min(v + \lceil q \rceil, \text{ metric max of } P)$
return a uniform random selection from $[l, h]$

the other. Additionally, Edisyn's *nudge* facility allows the user to use recombination to push a patch towards or away from up to four different target patches, not unlike mixing paints on a palette.

The above tools rely in turn on three mutation and recombination algorithms to do their work. Each of these algorithms takes a single weight as a tuning parameter, to minimize the complexity of the system for the user. They also take into consideration the fact that on effectively all synthesizers, metric parameters are integers (due to MIDI). The algorithms are summarized below, with details in Algorithms 1–4:

Mutate (Algorithm 1) mutates a parameter. The weight defines the probability that the value is randomized (if non-metric) and/or the degree to which it will be modified (if metric). If the parameter is hybrid, with 0.5 probability its value will be mutated from metric to non-metric (or vice versa). Weighted patch randomization and nudging both use this operator.

Recombine (Algorithm 3) modifies a parameter in one patch to reflect the influence of another patch. The weight defines the probability of recombination occurring, either via weighted interpolation (if metric) or choosing one value or the other (if non-metric, or if the parameter is hybrid and one value is metric while the other is not). Weighted patch recombination and nudging (towards a target) both use this operator.

Algorithm 3 Recombine(P, v, w, $0 \leq weight \leq 1$)

Input: Parameter P with current values v, w; *weight*
if coin flip of probability *weight* **then**
  if v and w are both metric values in P **then**
    return a uniform random selection from $[v, w]$
  else if coin flip of probability 0.5 **then**
    return w
return v

Algorithm 4 Opposite(P, v, w, $0 \leq weight \leq 1$, boolean *flee*)

Input: Parameter P with current values v, w; *weight*; *flee*
if $v = w$ and *flee*=true **then**
  if v is non-metric **then**
    if coin flip of probability *weight* **then**
      return a random non-metric value from $P - \{w\}$
  else
    return $v + 1$ or $v - 1$ at random as constrained by the metric min and metric max of P
else if both v and w are metric values in P **then**
  $q \leftarrow \lceil v + weight \times (v - w) \rceil$
  $q \leftarrow \min(\max(q, \text{ metric min of } P), \text{ metric max of } P)$
  return a uniform random selection from $[v, q]$
return v

Opposite (Algorithm 4), an unusual recombination variant, finds a value on the *other side* of parameter P from Q, notionally because we had moved from Q to P and wished to continue in that direction. The weight determines the degree to which the value moves away from Q if both are metric. If either value is non-metric, or if they are equal, then P's value stays as it is.

A subtly different form of Opposite is used when we want P to *flee from* Q: here, if the two values are equal, then if they are non-metric P's value is randomly made different from Q with a probability determined by the weight; else if they are metric, P's value is nudged slightly away from Q (so that $|P - Q| > 0$ and so future Opposite-recombinations can push it further away). Nudging (away from a target) uses the flee version of this operator.

3 Stochastic Patch Exploration Tools

Edisyn also has two stochastic patch exploration procedures: the Hill-Climber and the Constrictor. Both procedures are available in Edisyn's Hill-Climber panel, shown in Figure 2. The purpose of these methods is to help the user to rapidly explore the patch parameter space in search of patches interesting or useful to him; after discovering such patches in rough form, the user can then fine-tune the patches by hand in Edisyn's editor.

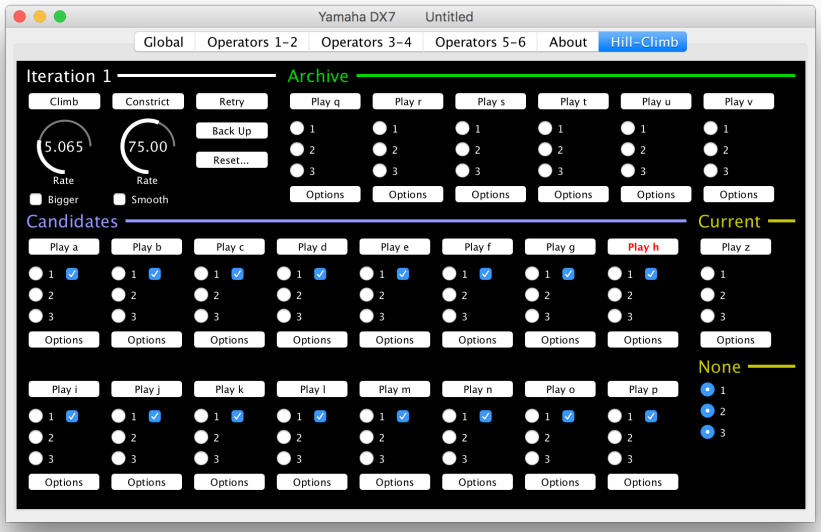

Fig. 2. Edisyn's Hill-Climber.

The two procedures are both variations of interactive evolution strategies, but have different goals and heuristics, and consequently very different dynamics. The Hill-Climber applies an ES with a specialized breeding procedure to wander through the space guided by the user's preferences as a fitness function. In contrast, the Constrictor applies an ES with crossover only, again guided by the user's preferences, to gradually reduce the convex hull or bounding box of the search space until it has (intentionally prematurely) converged to a point in the space. The two procedures again rely on the above Mutate, Recombine, and Opposite operators (in non-flee form). The Constrictor may also use an additional *Crossover* operator, described later.

Like other Edisyn patch-exploration tools, the Hill-Climber and Constrictor are general-purpose. The two methods are designed to tackle the fitness bottleneck problem in two ways. First, the techniques both act essentially as black boxes, requiring only a single user-selectable weight parameter each, so as not to overwhelm the user with complexity, and to expedite selections and decisions. Second, both try to avoid poor-quality regions of the space so as to speed search for viable candidates: the Hill-Climber does this by making largely local moves from earlier-chosen individuals; and the Constrictor tries to reduce the space to just the convex hull or bounding box of well-vetted solutions.

4 The Hill-Climber

The *Hill-Climber* is essentially a (μ, λ) evolution strategy variant which emphasizes local search. Rather than have the user specify how to generate children, the Hill-Climber applies a custom breeding protocol, shown in Figure 3, designed to sample

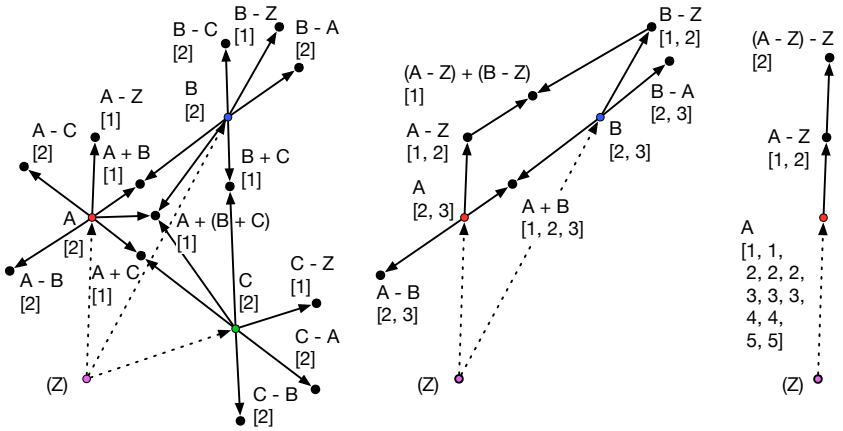

Fig. 3. Hill-Climber breeding procedures when the user selects three parents (left), two (center), or just one (right). In these diagrams, *A, B,* and *C* denote those three parents in order of preference (*A* being most preferred). (Z) denotes the previous-generation's A. All nodes other than (Z) denote children created from A, B, C, and Z: each is labeled with the operation used to generate the child, such as $B - A$. The operation $X + Y$ is recombination (preferring X) and $X - Y$ is non-flee opposite-recombination. Each child is then mutated at least once, or multiple times as indicated in square brackets: e.g. [3] is three mutations in series. When multiple numbers are in brackets, this indicates that multiple children are created using this procedure, with different numbers of mutations. For example $A - Z$ $[1,2]$ means that two children are created using $A - Z$, with one child mutated once, while the other is mutated twice.

from a distribution whose size is proportional to the diversity in the user's choices (a notion drawn from Differential Evolution). The protocol also applies the heuristic that if the user has been choosing candidates in a certain direction, he may wish to consider some candidates *even further* in that direction. Finally, the protocol tries to balance both diversity-creating operations and recombination operations.

Though the user is not required to do so by default, Edisyn permits the user to bias this search by nudging candidates toward desired targets, editing them, backing up, storing and recalling search points, and constraining the parameters being explored (a simple form of dimensionality reduction). The user has control over a single parameter, the *mutation weight*, which affects both the probability of mutative changes and their strength. Increasing this weight will increase variance among patches, while decreasing it leads to convergence. The recombination weight is fixed, as I have found anecdotally that control over this parameter is confusing to patch designers and is rarely changed.

Edisyn's Hill-Climber initially builds 16 children mutated from the original patch, which are then sent to the synthesizer to be auditioned in turn. The user iteratively selects up to three preferred "parent" patches chosen from the 16, plus optionally the original patch and an archive of up to six patches he has collected over time. After this, he Hill-Climber generates 16 new children from the parents, auditions them, and so on. In EC this would be, loosely, an approximately *(3, 16)* evolution strategy. While 16 is the default, the Hill-Climber can also be set to use a population size of 32.

Algorithm 5 Crossover(P, v, w, $0 \leq weight \leq 1$)

Input: Parameter P with current values v, w; $weight$
if coin flip of probability $weight$ **then**
  if coin flip of probability 0.5 **then**
    return w
return v

To generate children, the Hill-Climber employs three different but fixed breeding strategies depending on the number of parents chosen, as shown in Figure 3. The strategies rely only on the (up to) three selected parents in order, plus the previous #1 parent. The operations are meant to provide both mixtures of the parents as well as diversity from them, and to heuristically continue moving in the direction the user had been previously going: a form of momentum. As detailed in Figure 3, all the breeding procedures ultimately rely on the aforementioned mutation, recombination, and (non-flee) opposite-recombination procedures.

5 The Constrictor

Edisyn's *Constrictor* also applies an evolution strategy but with a different goal. Whereas the Hill-Climber behaves as a kind of semi-directed random walk through the space, the Constrictor begins with several different pre-defined patches and slowly reduces their convex hull or bounding box through repeated selection and recombination. The idea behind the Constrictor is to reduce the space to patches whose parameters are combinations of well-vetted patches and so are (hopefully) more likely to be useful in the first place.

The Constrictor works as follows. First, some $N \leq 16$ individuals are loaded from user-selected patches, and the remaining $16 - N$ individuals are generated from the original N by iteratively selecting two of the N and recombining them to create a child. These 16 individuals then form the initial population. The constrictor then iterates as follows. The user first selects some $M \ll 16$ individuals from this group that he *doesn't* like, perhaps because they are of poor quality, or are uninteresting, or are duplicates of others. The system then replaces these M individuals with new children recombined from the remaining $16 - M$, and the iteration repeats. There is no mutation: at the limit this approach will converge to a single patch due to, effectively, the premature convergence of crossover. Again, while a population size of 16 is the default, Edisyn can be set to use 32 instead.

Depending on user settings, the Constrictor uses one of two different recombination approaches: either it uses *Recombine* (Algorithm 3, described earlier) or it uses *Crossover* (Algorithm 5). Whereas Recombine directly crosses over non-metric parameters while doing a weighted randomized average of metric parameters, Crossover instead is a traditional crossover procedure: it directly crosses over *all* parameters. Just like Recombine (and other operators), Crossover takes a single tuning parameter. When used with Recombine, the Constrictor is known as a *Smooth Constrictor*, and when used with Crossover, it is known as an *Unsmooth Constrictor*.

6 Experiment

It is nontrivial to formally analyze or assess a system with a human in the loop whose mind changes rapidly. Furthermore, while Edisyn's Hill-Climber and Constrictor can be used to optimize for a target sound, they are designed for, and tuned for, assisted patch *exploration* due to the human factor. Thus we attempted to gauge user response to these tools, and to solicit their opinion as to the tools' effectiveness. We asked a sample of 28 university-level computer science or music technology students to assess different Edisyn patch-exploration techniques. Almost all of the subjects also had significant music performance experience, but only a handful had previous experience with music synthesizers.

Choice of Synthesizer We chose the Yamaha DX7, the archetypal FM synthesizer, as our target platform. FM synthesis presents an obvious demonstration of why patch-exploration tools are useful: it is counterintuitive and difficult, but at the same time is very widely used. Though they are general-purpose, Edisyn's techniques were developed originally for FM synthesis devices because of these reasons. This isn't unusual: indeed the first application of evolutionary computation to patch design [1] also targeted FM as well, and with a similar justification. We also felt that FM's difficulty in patch design (for everyone) would also put all but the very most experienced patch designers on roughly equal footing in the study.

Procedure We began the study with an hour-long hands-on tutorial on synthesizers, FM synthesis, and how to program DX7 patches in Edisyn. The objective was to get subjects up to the level of a novice DX7 programmer. We also included a short introduction in how to use the Hill-Climber and Constrictor features. Each subject used his own laptop, working on Edisyn to control Dexed, a well-known open source emulator of the DX7 (a DX7 itself could easily have been used as well). The DX7's pitch-modification parameters (*Transpose* and the *Pitch Envelope*) were disabled, as were the *Fixed Frequency* and *Frequency Fine* parameters for each DX7 operator, so as to eliminate the highly sensitive, unhelpful, and often atonal swings associated with them.

We then asked each subject to try four different techniques (direct programming, hill-climbing, smooth constrictor, and unsmooth constrictor, in randomized order) to explore the patch parameter space in search of sounds which were in their opinion musical, interesting, or useful. Each trial began with a specific initial patch (or as set of four patches for the constrictors), from which the subjects would begin to explore. In the case of direct programming, the subject began with the initial patch and explored the space by directly modifying the parameters using Edisyn's primary features (including Undo). A trial lasted approximately five minutes, and after each trial the subject provided his opinions of the technique, both as ratings from 1 to 10 and in text. There were eight trials in all (two sets): the first set of trials were a warm-up period and intentionally discarded. After all trials had concluded, we discussed the subject's opinions with them.

Results The results to the primary question ("On a scale from 1 to 10 (higher is better), rate your satisfaction with this method.") were as follows:

Method	*Mean*	*Variance*
Direct Programming	5.857143	4.9417990
Hill-Climbing	**7.107143**	3.0621693
Smooth Constrictor	**7.535714**	1.7394180
Unsmooth Constrictor	**7.035714**	2.2579365

Boldface indicates methods which were statistically significantly preferred over Direct Programming, using an ANOVA at $p = 0.05$ with a Holm-Bonferroni post-hoc test. The three other methods had no statistically significant difference among them, though Smooth Constrictor was very close to being preferred over all others.

These results were corroborated by subject discussion after the trials: with few exceptions, the subjects found the exploratory methods much more effective than direct programming. Subjects also generally noted a preference for Smooth Constrictor, particularly over its Unsmooth counterpart. Smooth was preferred over Unsmooth largely because Unsmooth created too many unusual and deviant sounds.

In an attempt to verify why certain methods were preferred over others, the study also asked several follow-up questions asking which methods were more effective at unusual sounds vs. ones more useful in a compositional setting, or methods which deviated the most rapidly from the initial settings. The results for these questions were not statistically significant.

7 Anecdotal Observations

Due to the difficulty in obtaining experts, the previous experiment was largely based on novices to synthesizer programming. But Edisyn's tools were really designed with experienced users in mind. I view myself as relatively experienced in patch development, and have personally found these tools to be very useful; and have observed the same in others more expert than myself. And in the case of FM, "experts" of any kind are rare.

In using the Hill-Climber in particular I have also found, not unexpectedly, that reduced dimensionality is key. Edisyn was developed with FM synthesis in mind; but high-dimensional additive synthesizers have proven the most challenging, despite their parameters' typically low epistasis. Also, changes in certain sound features (like modulation of pitch or volume) will mask subtle changes in others. For these reasons, it is critical to constrain the search to just those parameters of interest.

Another difficulty stems from large-range non-metric parameters. For example, many ROMplers have a "wave" parameter which specifies which of a great many stored samples should be played by an oscillator. These samples are arbitrarily ordered, and so this parameter is non-metric but with a large set of values to choose from. Some E-Mu ROMplers, such as the Morpheus and Ultra Proteus, also offer parameters for hundreds of arbitrarily ordered filter options. Parameters such as these are critical to a ROMpler's patch, but their complete lack of gradient presents a worst-case scenario for optimization tools. I suspect there may be no way around this.

Like most evolutionary methods, Edisyn's exploration tools can get caught in local optima. MutaSynth historically championed temporary "best so far" storage as a way to help the user maintain diversity and thus work his way out of local optima. Edisyn does

this as well, sporting a "best so far" archive of six candidate patches: the user can store his best discovered candidates in this archive at any time, and during hill-climbing he can also include these candidates as part the selection procedure. However, an Edisyn user can also back up (undo) to previous iterations in the hill-climbing or constricting process, or can retry an iteration if he dislikes the results; and I have personally found myself more reliant on these two facilities than the archive facility when dealing with local optima.

Last, but hardly least: I have found the following workflow more effective than using any of these tools in isolation. Namely, one might start with a constrictor to work down to a general area of interest in the parameter space, then switch to the hill-climber to explore in that region, and then ultimately switch to hand-editing to tune the final result.

8 Conclusion and Future Work

This paper discussed the synthesizer patch editor toolkit Edisyn and described and examined its stochastic patch exploration tools. These tools are interactive evolutionary algorithms meant to assist the patch programmer in exploring the parameterized space of synthesizer patches in order to improve them and to discover new and useful ones.

Though their usefulness is obvious and their application straightforward, general-purpose evolutionary patch exploration tools are very rare. Many patch editors (and some synthesizers, such as the Waldorf Blofeld) provide crude patch randomization or recombination tools, but almost none provide more sophisticated techniques. Edisyn is notable both in its array of patch exploration tools and also in the fact that the tools are all designed to work with any patch editor written in the toolkit, so long as a patch is defined as a vector of values from heterogeneous metric, non-metric, or hybrid domains.

Because the examples are so rare, there are many opportunities in this area for exploring additional interactive metaheuristic approaches. One obvious extension of this work of particular interest to the author is into spaces with more complex evolutionary representations. There has been significant interest of late in hardware and software synthesizers consisting of many *modules*, of any number, and connected any number of ways. Simpler versions of these devices (so-called "semi-modular" synthesizers) have patches in the form of arbitrarily-long lists; more sophisticated "fully modular" architectures would require an arbitrary and potentially cyclic directed graph structure as well. I have myself written fully-modular software synthesizers. But as has long been recognized in metaheuristics, and particularly in the genetic programming and neuro-evolution subfields, optimizing such representations is nontrivial, ad-hoc, and not particularly well understood even now. Though modular synthesizers are now very popular, the literature in modular synthesizer patch exploration (notably [12]) is only in its infancy, and this area is well worth further consideration.

Hopefully in the future we will see more synthesizer patch editors with tools such as those demonstrated in this paper. Synthesizers are popular but challenging to program devices, and some synthesizer approaches (additive, FM) are notoriously so. Editors improve the user interface of the machines but do not necessarily help overcome these inherent difficulties: but patch exploration tools can make this possible even for inexperienced users.

Acknowledgments My thanks to Vankhanh Dinh, Bryan Hoyle, Palle Dahlstedt, and James McDermott for their considerable assistance in the development of this paper.

References

1. Horner, A., Beauchamp, J., Haken, L.: Musical tongues XVI: Genetic algorithms and their application to FM matching synthesis. Computer Music Journal 17(4), 17–29 (1993)
2. Takagi, H.: Interactive evolutionary computation: Fusion of the capabilities of EC optimization and human evaluation. Proc. IEEE 89(9), 1275–1296 (2001)
3. Biles, J.A.: GenJam: a genetic algorithm for generating jazz solos. In: ICMC. pp. 131–137 (1994)
4. McDermott, J., O'Neill, M., Griffith, N.J.L.: Interactive EC control of synthesized timbre. Evolutionary Computation 18(2), 277–303 (2010)
5. Seago, A.: A new interaction strategy for musical timbre design. In: Holland, S., Wilkie, K., Mulholland, P., Seago, A. (eds.) Music and Human-Computer Interaction, pp. 153–169. Springer (2013)
6. Suzuki, R., Yamaguchi, S., Cody, M.L., Taylor, C.E., Arita, T.: iSoundScape: Adaptive walk on a fitness soundscape. In: EvoApplications. pp. 404–413. Springer (2011)
7. Dahlstedt, P.: A MutaSynth in parameter space: interactive composition through evolution. Organized Sound 6(2), 121–124 (2001)
8. Dahlstedt, P.: Evolution in creative sound design. In: Miranda, E.R., Biles, J.A. (eds.) Evolutionary Computer Music, pp. 79–99. Springer (2007)
9. Dahlstedt, P.: Thoughts of creative evolution: A meta-generative approach to composition. Contemporary Music Review 28(1), 43–55 (2009)
10. Collins, N.: Experiments with a new customisable interactive evolution framework. Organised Sound 7(3), 267–273 (2002)
11. Mandelis, J.: Genophone: Evolving sounds and integral performance parameter mappings. In: EvoWorkshops. pp. 535–546. Springer (2003)
12. Yee-King, M.J.: The use of interactive genetic algorithms in sound design: a comparison study. Computers in Entertainment 14 (2016)
13. Ianigro, S., Bown, O.: Plecto: A low-level interactive genetic algorithm for the evolution of audio. In: EvoMUSART. pp. 63–78 (2016)
14. Jónsson, B., Hoover, A.K., Risi, S.: Interactively evolving compositional sound synthesis networks. In: GECCO. pp. 321–328 (2015)