A Simulation of Niche Construction

Kevin B. Korb and Alan Dorin
Technical Report 2009/2
Bayesian Intelligence

Abstract

We report on a simple, generic simulation of niches and their propagation in a two-dimensional torus. Under a wide range of circumstances, the number and variety of niches that develop expand exponentially.

Keywords: Niche complexity, niche construction.

1 Introduction

The artificial life and biological communities have begun focusing upon niches as a central concept for understanding evolution, particularly since Odling-Smee et al. (2003). We believe this reorientation is a fruitful one, particularly for understanding the evolution of complexity.

We take *niches* to be loci in a biogeochemical web of relations between species, including both resource inputs and outputs and environmental constraints upon the species occupying those loci. In an engineering sense, they are input-output functional roles that species fulfill. (For an explicit definition and discussion, see Korb and Dorin, 2009.)

Elsewhere, we have claimed that niche complexity is the proper way of interpreting biological complexity for the purpose of understanding open-ended evolution (Korb and Dorin, 2009). We further claimed that in real evolutionary biology niche complexity grows exponentially until it hits capacity constraints:[1]

Hypothesis 1 (The Arrow of Niche Complexity) *The Arrow of Niche Complexity has an exponential trajectory.*

A generic argument in favor of Hypothesis 1 is that every species must create multiple niches. The first and most obvious niche is the availability of that type of organism for consumption. Other niches are created by the organism's waste products and its potential for decomposition should it die other than by being eaten directly. In addition to those are typically multiple effects upon the environment via its behavior or simply its presence. Ecosystem engineers rework their environments in predictable ways, ways upon which other organisms can rely for shelter, food, etc. (Jones et al., 1997). In short, any new niche being filled by an organism implies the creation of multiple further niches. This produces a classic "population" explosion of niches.

[1] The exponential growth may not begin until a significant base of species has been established, leading to a logistic curve; cf. Ward (1995).

Here we describe a simulation which supports this argument by incorporating as simple a niche generation process as possible, within which we find exponential growth in distinct niches over a wide range of circumstances.

2 Design

This simulation (see Appendix A) is intended to provide the simplest possible design that might support the exponential growth of niches. It is not meant to be a realistic portrayal of niche construction, but instead to provide a basic portrayal of niche construction, with the idea that robust and important features of the simulation are the more likely to be essential to any system of niche construction.

"Niches" here mean occupied niches, not potential niches: there is always one and only one species occupying a niche, hence "species" and "niche" may be used interchangeably for this simulation. Thus, when we speak of niche "health", it just means the health of the species occupying the niche. Niche "fitness" relies upon the fitness of its species: if its species is fit it will reproduce and its offspring (subspecies or new species) will (probably) occupy a niche similar to that of its parent. Time steps in the simulation, therefore, reflect mutational changes undergone by species as they search the genetic design space, much as in the simulation of McShea (1994). We reflect such relationships more directly and simply: fit niches reproduce with minor variations in their offspring niches.

2.1 Niches

In detail, niches consist of two lists of numbers, an input and an output list. The input list identifies resource requirements; the output list identifies products. These lists are of arbitrary length, greater than or equal to one. The vocabulary of resources needs to be constrained for niches to find producers for their needs and consumers for their products. In the current simulation they are limited to 16 values.

2.2 Environment

The environment is a 2-d torus, divided into squares. Each cell provides a string of resources which does not change during the life of the simulation. (Such change would effectively be geological change, which we are not simulating.) The number of cells and their resources provides some constraint on the carrying capacity of the simulation.

2.3 Simulation time steps

At each time step, niches are selected in a random order for processing (all niches are processed exactly once in each time step).[2]

For each niche, where Pn is the n-th parameter of the simulation:

[2]In the future, it might be useful to bias niche processing so that those whose requirements are already satisfied are processed first, rather than wasting time skipping over, and docking the health of, many niches. The way we are doing it is slower, and also it execessively penalizes niches and constrains their growth.

1. P0 health units are subtracted.
2. If all required resources are available:
   (a) Those resources are deleted from those available
   (b) P1 health units are added to the niche
   (c) All niche products are added to resources available
3. If health $<= 0$, the niche is deleted.

After all niches are processed, a health proportional probabilistic (P3) asexual niche reproduction is performed. Offspring niches undergo two "genetic" operations:

- with probability P4 deletion of characters
- with probability P5 insertion of characters

3 Results

A brief record of our experiments is in Appendix B. First, we examined independently the three simulation parameters new-health (the amount of "health" the species/niche received when its input requirements were satisfied in a time step), turn-health (the amount of health lost per time step), and reproduce-prob (the probability that a niche will reproduce in a time step). In all cases we used a 10x10 torus for 20 generations, repeating the simulation 30 times for each set of parameters. (Initially we ran simulations for 30 generations, but later experiments ran too slowly.)

The number of niches is reported both absolutely and in terms of "distinct niches", meaning counting only the first occurrence of niches which have identical input-output relations.

The first parameter set (1 -20 0.5) yields a simulation in which the population of niches stablizes rapidly at a relatively low level. We call this non-exponentially growing parameter set the "baseline". By increasing the new health available upon satisfying requirements we were readily able to achieve exponential niche growth. Again from the baseline we decreased the health cost per turn and with -5 achieved exponential growth rates. Finally, from the baseline we increased the reproduction probability, without however achieving exponential growth even with a probability 1. So, the constraining factors in the simulation described by the baseline condition just have to do with how much health is lost by niches over time and how much is gained when their requirements are met.

When we put the favorable parameter values together in our last reported experiments, the result was, of course, exponential growth, and with a substantially steeper slope, resulting in more than an order of magnitude increase in the number of distinct niches over any other simulation.

4 Conclusion

We have developed a simulation of niche construction which is as simple and general as we can make it. In this simulation, so long as the degradation of a niche missing out on its input requirements is not huge, it is easy to get exponential growth in distinct occupied niches in an ecosystem. This offers some support to Hypothesis 1: that the Arrow of Niche Complexity is exponential.

References

Jones, C. G., J. Lawton, and M. Shachak (1997). Positive and negative effects of organisms as physical ecosystem engineers. *Ecology 78*, 1946–1957.

Korb, K. and A. Dorin (2009). Evolution unbound: Releasing the arrow of complexity. *Journal of Theoretical Biology*. Under submission.

McShea, D. (1994). Mechanisms of large-scale evolutionary trends. *Evolution 48*, 1747–1763.

Odling-Smee, F., N. Laland, and M. Feldman (2003). *Niche construction.* Princeton: Princeton University.

Ward, P. (1995). *The end of evolution: Dinosaurs, mass extinction and biodiversity.* London: Weidenfeld and Nicholson.

A Simulation Code

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; niche.lisp
;
; Kevin Korb, Sept 2009
;
; This simulation provides the simplest possible
; design that might support the exponential growth of niches.
;
; Limitations:
;   Resources are available globally rather than locally.
;   Niches do not move
;   All mutation is done via insertion and deletion
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; constants
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defconstant delete-rate 0.1)
(defconstant insert-rate 0.1)
(defconstant casting-distance 1)  ; for new niches
```

```
(defconstant init-health 100)     ; initial niche health
(defconstant repro-health 100)    ; health required for reproduction
(defconstant repro-distance 2)    ; defines repro neighborhood
(defconstant init-reqts-length 3) ; for non-initial niches
(defconstant min-reqts-length 3)
(defconstant init-prod-length 3)
(defconstant num-products 16)     ; number of resources/products
(defconstant init-niche-health nil)  ; not a number

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; globals
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; sim parameters altered during testing
(defparameter world-size 0)
(defparameter turn-health -70)     ; health cost per turn
(defparameter new-health 10)       ; health boost per turn
(defparameter reproduce-prob 0.7)  ; prob healthy niche repros
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defparameter population-size 0)   ; number of extant niches
(defparameter population-list nil) ; all niches listed here
(defparameter resources nil)       ; global list of resources available
(defparameter births 0)
(defparameter deaths 0)
(defparameter niche-hash (make-hash-table
  :test #'equal
  :size 1000
  :rehash-size 100))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; data structures
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defstruct niche
  loc
  health
  requirements
  products)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; help functions
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun manhatten (loc1 loc2)
  (+ (abs (- (car loc1) (car loc2)))
     (abs (- (cadr loc1) (cadr loc2)))
))
```

```
; count the num of niches at a location
(defun count-niches (board locx)
  (aref board (car locx) (cadr locx)))

(defun print-board (size)
  (let ((board (fill-board size)))
    (format t "~%")
    (do ((i 0 (1+ i)))
((>= i size))
      (do ((j 0 (1+ j)))
  ((>= j size))
(let ((num (count-niches board (list i j))))
  (if (< num 10)
      (format t "~d" num)
      (format t "*")
      )))
      (format t "~%")
      )))

; count up all niches per location, keeping count at that loc
; in board
(defun fill-board (size)
  (let ((board (make-array (list size size) :initial-element 0)))
    (dolist (nn population-list board)
      (incf (aref board (car (niche-loc nn)) (cadr (niche-loc nn)))))))

(defun print-stats (gen)
  (format t "~%***** Generation ~d statistics *****" gen)
  (format t "~%  Niches:    ~d" population-size)
  (format t "~%  Distinct niches:    ~d" (hash-table-count niche-hash))
  (format t "~%  Births:    ~d     Deaths:   ~d" births deaths)
  (format t "~%  Resources: ~d" (length resources))
  (format t "~%")
)

(defun gen-prod-list (len)
  (do ((i 0 (1+ i))
       (prods nil))
      ((>= i len) prods)
    (setf prods (cons (random num-products) prods))))

(defun print-hash ()
  (maphash #'(lambda (k v) (format t "~%~a = ~a " k v))
   niche-hash))

(defun inc-pop (tn)
  (let ((reqts (sort (copy-list (niche-requirements tn)) #'<))
```

```
(prods (sort (copy-list (niche-products tn)) #'<))
)

  (incf population-size)
  (incf births)

  ; increment or add a hash table entry
  (if (null (gethash (list reqts prods) niche-hash))
      ; make a new entry
      (setf (gethash (list reqts prods) niche-hash) 1)
    ; inc existing entry
    (setf (gethash (list reqts prods) niche-hash)
  (1+ (gethash (list reqts prods) niche-hash))))
  )
)

(defun dec-pop (tn)
  (let ((reqts (sort (copy-list (niche-requirements tn)) #'<))
(prods (sort (copy-list (niche-products tn)) #'<))
)
    (decf population-size)
    (incf deaths)

    ; decrement or delete a hash table entry
    (cond
     ((null (gethash (list reqts prods) niche-hash)))
     ((<= 1 (gethash (list reqts prods) niche-hash))
      (remhash (list reqts prods) niche-hash))
     (t (setf (gethash (list reqts prods) niche-hash)
      (1- (gethash (list reqts prods) niche-hash)))))
    )
)

;
; init-niche creates a new niche ab initio
;

(defun init-niche (locx)

  (let ((new-resources (gen-prod-list init-prod-length)))


  ; create the niche
  (setf population-list
(append population-list
(list (make-niche
       :loc locx
       :health init-niche-health
       :requirements nil        ; initial niches require nothing
```

```
       :products new-resources))))

  ; track the population
  (inc-pop (car (last population-list)))

  ; add in the new resources to global pool
  (setf resources
(append new-resources resources))
  ))


(defun mutate (l)
  (do ((i 0 (1+ i))
       (insert-loc 0)
       (delete-loc 0)
       (nl l)
       )
      ((>= i (length l)) nl)

  (cond
    ((and (not (null nl)) (< (random 1.0) delete-rate))
     (setf delete-loc (random (length nl)))
     (setf nl (append (subseq nl 0 delete-loc)
      (subseq nl (1+ delete-loc) (length nl))))))

  (cond
    ((< (random 1.0) insert-rate)
     (if (null nl)
 (setf insert-loc 0)
       (setf insert-loc (random (length nl))))
     (setf nl (append (subseq nl 0 insert-loc)
      (list (random num-products))
      (subseq nl insert-loc (length nl)))))))

;
; reproduce creates a new niche from an existing one
;

(defun reproduce (tn)

  (if (< (random 1.0) reproduce-prob)
        (let ((new-niche (copy-structure tn))
    )


 ; mutate requirements
 (if (< (length (niche-requirements new-niche)) min-reqts-length)
     (setf (niche-requirements new-niche) (gen-prod-list min-reqts-length))
   (setf (niche-requirements new-niche) (mutate (niche-requirements new-niche))))
```

```
        ; mutate products
 (setf (niche-products new-niche) (mutate (niche-products new-niche)))

        ; initialize health
 (setf (niche-health new-niche) init-health)

        ; locate the niche
 (setf (niche-loc new-niche)
       (list (mod (+ (car (niche-loc new-niche))
     (random repro-distance))
  world-size)
     (mod (+ (cadr (niche-loc new-niche))
     (random repro-distance))
  world-size)))

 ; update the population
 (setf population-list
       (append population-list (list new-niche)))

 ; track the population
 (inc-pop new-niche)

 ; add in the new resources to global pool
 (setf resources
       (append (niche-products new-niche) resources))
 )))

;
; generate the random order of niches for this turn
;

(defun niche-order ()
  (let* ((ordered-list nil)
 (order (dotimes (x population-size ordered-list)
  (setf ordered-list (cons x ordered-list))))
 (index 0)
 )

    (do ((new-order nil)
 )
((= (length new-order) population-size) new-order)

      (setf index (random (length order)))

      ; move random elt to new list
      (setf new-order (cons (nth index order) new-order))

      ; delete from old list
```

```
      (if (= index (1- (length order)))
  (setf order (subseq order 0 index)) ; deleting last elt
  (setf order (append (subseq order 0 index)
  (subseq order (1+ index) (length order)))))
      ))
)

; match and conditionally remove and add resources

(defun match (tn)
  (cond ((match* (niche-requirements tn))
 (setf resources (remove-first (niche-requirements tn)))
 (setf resources (append resources
 (niche-products tn))))))

(defun match* (l)
  (cond ((null l) t)
((= (length l) 1) (member (car l) resources))
(t (and (member (car l) resources)
(match* (cdr l))))))

; delete first resource matching each elt of l
(defun remove-first (l)
  (do* ((nl l (cdr nl))
(x (car nl) (car nl))
        (trail (member x resources) (member x resources))
)
       ((null nl))

    (setf resources (append
     (subseq resources 0 (- (length resources)
    (length trail)))
     (cdr trail)))
    ))

; kill one niche
(defun kill* (tn)
  (setf population-list (remove tn population-list :count 1))
  (dec-pop tn)
)

;
; initialize the sim at generation 0
;

(defun initialize (size)

  (format t "~% ***************")
  (format t "~% New World Order")
```

```
  (format t "~% ***************")

  (format t "~% ~d,~d,~d,~d" new-health turn-health reproduce-prob size)

  (setf population-size 0)
  (setf population-list nil)
  (setf resources nil)
  (clrhash niche-hash)

  (init-gen)

  ; populate the world
  (do ((i 0 (1+ i)))
      ((>= i size))
    (do ((j 0 (1+ j)))
      ((>= j size))
      (init-niche (list i j))))

  )

(defun init-gen ()
  (setf births 0)
  (setf deaths 0)

)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; niche drives the simulation
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun niche (gens nh th rp size)

  ; globalize parameters
  (setf new-health nh)
  (setf turn-health th)
  (setf reproduce-prob rp)
  (setf world-size size)

  (do* ((gen 0 (1+ gen))
        (niches nil)
)
       ((>= gen gens))
    (if (= gen 0) (initialize size))

    (setf niches (niche-order))

    ; print current world
```

```
    (print-board size)
    ; collect and print statistics
    (print-stats gen)

    (init-gen)

    ;;;;;;;;;;;;;;;;;;;
    ; do one generation
    ;;;;;;;;;;;;;;;;;;;

    (do* ((i (car niches) (car niches)) ; i indexes pop-list
  (this-niche (nth i population-list)) ; must be >= 1 niche
 )
((null i))

      (setf this-niche (nth i population-list)) ; aquamacs lisp is bugged!

      ; adjust health of current niche, excepting init niches
      (cond
((numberp (niche-health this-niche))
        ; health cost
 (setf (niche-health this-niche) (+ (niche-health this-niche)
    turn-health))
        ; health gain
 (if (match this-niche)  ; are reqd resources there?
     (setf (niche-health this-niche)
   (+ (niche-health this-niche) new-health)))

 )
)

      ; niche reproduction
      (if (or (not (numberp (niche-health this-niche)))
      (>= (niche-health this-niche) repro-health))
  (reproduce this-niche))

      ; prep next loop
      (setf niches (cdr niches))
      )

    ; kill off niches
    (do*
      ((l population-list (cdr l))
       (x (car l) (car l))
       )
      ((null x))
      (if (and (numberp (niche-health x)) (<= (niche-health x) 0))
  (kill* x)))
```

```
    )
)
```

B Simulation Results

```
Parms: new-health turn-health reproduce-prob size

      Parms                  niche pop sd                          dn pop    sd

Parms 1 -20 0.5 10;   niches 360.96667 11.28161;  distinct niches 353.93332 11.392476
Parms 10 -20 0.5 10;  niches 363.2 12.810018;     distinct niches 356.6 12.960417
Parms 50 -20 0.5 10;  niches 469.13333 26.908188; distinct niches 439.83334 23.218725
Parms 100 -20 0.5 10; niches 1412.6 250.96896;    distinct niches 1111.9333 165.56256

Parms 1 -10 0.5 10;   niches 618.1 17.852558;     distinct niches 609.8333 17.378313
Parms 1 -5 0.5 10;    niches 1126.3667 23.603745; distinct niches 1116.0667 24.792843

Parms 1 -20 0.7 10;   niches 469.5 12.923515;     distinct niches 462.93332 13.44448
Parms 1 -20 0.9 10;   niches 576.4 7.591284;      distinct niches 568.6 8.973524
Parms 1 -20 1 10;     niches 626.5333 3.3397639;  distinct niches 619.13336 3.481313

Parms 100 -5 1 10;    niches 33359.266 1490.469;  distinct niches 24052.3 1078.0308
```