ARCHITECTURE AS A COMPLEX ADAPTIVE SYSTEM

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Architecture

by

Nathaniel Louis Jones

May 2009

CORNELL UNIVERSITY GRADUATE SCHOOL

APPROVAL OF THESIS/DISSERTATION

Name of candidate: Nathaniel (First Name) Louis (Middle Name) Jones (Family Name)

Graduate Field: Architecture

Degree: M Arch

Title of Thesis/Dissertation:

Architecture as a Complex Adaptive System

COMMITTEE SIGNATURES:

Chairperson: _____ Date: _____

Member: _____ Date: _____

Member: _____ Date: _____

Member: _____ Date: _____

Member: _____ Date: _____

Member: _____ Date: _____

LICENSE TO USE COPYRIGHTED MATERIAL

I do hereby give license to Cornell University and all its faculty and staff to use the above-mentioned copyrighted material in any manner consonant with, or pursuant to, the scholarly purposes of Cornell University, including lending such materials to students or others through its library services or through interlibrary services or through interlibrary loan, and delivering copies to sponsors of my research, but excluding any commercial use of such material. This license shall remain valid throughout the full duration of my copyright.

_____

(Student Signature)

ABSTRACT

Small changes to a design can have large and unexpected effects on a building's performance. Unfortunately, analysis of a design to uncover the unintended consequences of early decisions often takes place late in the design process, if at all. This is especially true for thermal analysis predicting comfort conditions and energy consumption in the building. Ignoring thermal problems or fixing them in late design stages are expensive alternatives to preventing them with early analysis.

Architecture is a complex adaptive system, where many parameters contribute to the overall fitness of a building. Some combinations of design elements offer more benefit than others. In complex adaptive systems, improvement happens through an evolutionary process. This iterative cycle of variation and selection occurs naturally when architects refine their designs. Using a parametric model, the evolutionary process can be sped up by a computer, allowing many more variants and evaluation criteria to be considered. Tools such as genetic algorithms can suggest creative solutions for design challenges that are too complex to be solved by human intuition.

The phenotype in this investigation is a house envelope built on a nine-square grid. Each square of the three-by-three grid contains a thermal zone whose nine parameters determine its height, roof slope, materials, and porosity. A deterministic crowding algorithm is used to minimize the house's energy requirements for heating, cooling, and lighting. In the space of fifty generations, the algorithm produces significant energy reductions when tested in a number of climates. At the same time, it generates a diverse population of options for further development by an architect.

Since the computer is unaware of normative building typologies for sustainable design, the forms it discovers are unique and unexpected. Three particularly creative solutions found by the algorithm are adapted into schematic designs. By assigning

rooms with matching size and material needs to each zone, the resulting designs take advantage of the efficient forms found by the algorithm. These schematic designs look quite different from typical sustainable houses; in some cases they incorporate elements that seem ill-advised from a thermal perspective. Clever adaptations still allow them to perform much better than the average house.

Genetic algorithms can provide the architect with feedback on design decisions as they are made. This feedback is not just validation of an idea, but a suggested course of action. As computational speeds increase, these algorithms will become more useful as they conduct larger searches with more evaluation criteria. Architects who use genetic algorithms will have a diverse set of tested options at their disposal early in the design process.

BIOGRAPHICAL SKETCH

Nathaniel Jones grew up in Iowa City, Iowa, where he won many high school math tournaments. At The Johns Hopkins University, he majored in civil engineering and earned a certificate in theater. While there, he began taking studio classes on the side at the Maryland Institute College of Art, which led him to do his graduate work in architecture. Since becoming a graduate student at Cornell University, he as interned in architecture firms in Cleveland, Washington D.C., and New York City. Currently, he is part of Cornell's Program in Computer Graphics, where he researches software tools that provide architects with feedback on thermal performance early in the design process. Outside of his academic pursuits, he has been involved in theater as a technical director and lighting designer, in music playing concert and jazz trombone, and in hand bookbinding for The Johns Hopkins University Libraries, Cornell Libraries, and briefly in private practice.

to Robin

ACKNOWLEDGMENTS

I am indebted to the many people who offered their time and advice while I was working on this thesis. Without their help, my task would have been, if not impossible, at least much harder. I owe them my sincere thanks.

Sanford Kwinter, for recommending that I read about chaos and complexity theory to learn about optimization in problems with many variables.

Kaustuv De Biswas, whose dynamic-link library provides interoperability between GenerativeComponents and Ecotect. The ability to send information between the two programs without user intervention sped work on this thesis immensely.

Jeroen Coenders of Arup, for his advice and recommendations on software to use for this research.

Hod Lipson, for making me aware of many genetic algorithm variants that might be applied to architecture, of which I have only had time to experiment with a few. I recommend Professor Lipson's Evolutionary Computation and Design Automation class to anyone interested in optimization.

Dana Cupkova and Lars Schumann, for their comments at interim reviews and their feedback throughout the semester.

David Bosworth, for the codes he wrote and collected that I used as tutorials while teaching myself GenerativeComponents and Ecotect.

Hurf Sheldon, for providing software and technical support during the later stages of work on this thesis.

Andrew Dankel, for keeping the software in Rand Hall's computer labs up to date, and for looking the other way when I used more than my fair share of resources.

Don Greenberg, for giving me an office so that I could escape the crowds in Rand Hall, and for warning me that this thesis might never work, which made me more sure than anything else that I had chosen the right topic.

Finally, Kevin Pratt, for his endless enthusiasm and support. Under Kevin's guidance, this thesis grew from the vague idea that computers can optimize complex architectural problems into a concrete algorithm and test case. It is hard to imagine fining anyone else who could have offered the same amount of support at each stage in this process.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF ABBREVIATIONS

AL	artificial life
CIBSE	Chartered Institution of Building Services Engineers
DC	deterministic crowding
DLL	dynamic-link library
GA	genetic algorithm
GC	GenerativeComponents
GP	genetic programming
HVAC	heating, ventilation, and air-conditioning
μ+λ	mu+lambda
PHC	parallel hill-climber

PREFACE

In my first design studio, my professor allowed us to sprinkle anti-gravity powder on our models if our designs could not be practically built. For some students, this freedom from the constraints of physics was liberating. For myself, an engineering student at the time, the idea that gravity could be ignored was not only absurd but also counterproductive. Constraints are the bread and butter of engineering. They separate practical invention from fantasy. Engineers are trained to optimize a design to work under whatever constraints they are given.

Considering how closely architecture and structural engineering are related, I am struck by the differences between the two disciplines. My studio critics occasionally tell me that I consider too much information. It puzzles me that while these professors hold the work of engineers in high esteem, they cannot accept optimization as a driving force for design. To them, it is a question of authorship. If a design is subjected to many goals and constraints, does it really represent the creativity of its designer, or does it become a product of formulas?

I see constraints as creative opportunities. The more a designer constrains himself, the more creative he must be to achieve his own goals. In my studios, I rebelled against simplicity. I believed that if used properly, engineering methods would reliably lead to "elegant" solutions to complicated architectural problems.

Sanford Kwinter told me that my focus on engineering was too narrow and turned my attention to complexity theory. This line of inquiry eventually led me to discover John Holland, Stuart Kauffman, and other pioneers of complex adaptive systems and genetic algorithms. Around the same time, and quite by accident, I came across Eric Beinhocker's *The Origin of Wealth*. While other authors look to biology for examples of complex adaptive systems, Beinhocker finds complexity theory in the

workings of the economy. The goods and services that evolve in a growing economy include the products of engineers. Moreover, the complex economic and social web that produces all the variety of engineered form also creates all of the world's architectural variety. It is time for someone to describe architecture in the same way that Beinhocker describes the economy.

This thesis discusses architecture as a complex adaptive system. It is written for an audience that has some familiarity with architecture, though not with complexity theory. This is *not* an in-depth discussion of complex adaptive systems – interested readers are directed to the references at the end. This is also *not* a proposal to design a specific building, as many architectural theses are. My aim is to show that creative and significant optimization is possible in architecture because it is a complex adaptive system.

CHAPTER 1

COMPLEX ADAPTIVE SYSTEMS

> Complexity is a deep property of a system, whereas complication is not. A complex system dies when an element is removed, but complicated ones continue to live on, albeit slightly compromised. Removing a seat from a car makes it less complicated; removing the timing belt makes it less complex (and useless). Complicated worlds are reducible, whereas complex ones are not.
>
> When a scientist faces a complicated world, traditional tools that rely on reducing the system to its atomic elements allow us to gain insight. Unfortunately, using the same tools to understand complex worlds fails, because it becomes impossible to reduce the system without killing it. The ability to collect and pin to a board all of the insects that live in the garden does little to lend insight into the ecosystem contained therein.
>
> John Miller and Scott Page, 2007

1.1 The Challenge of Optimization

An architect designs a housing development of square one-room buildings, each with two glazed walls. Her first buildings have been built, and more are planned. While the project receives favorable press in architectural circles, residents are complaining of high heating bills. The developer asks the architect to alter the design of the remaining buildings to save energy. The architect does not want to make major changes to a design that already has many good features, but she is willing to consider rotating the buildings on the site or changing their roof slopes. Not knowing how her

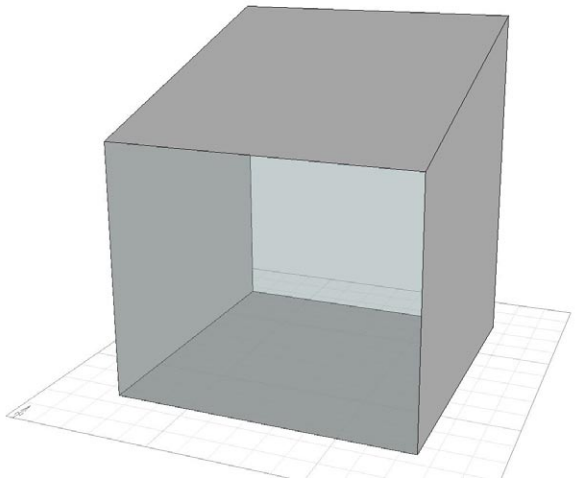

Figure 1.1.1 The digital model of a square one-room building can be manipulated by changing its roof slope or rotating it in plan.

changes will affect the design's heating load, she tests several options with thermal analysis software.

First, the architect rotates the design a quarter turn counterclockwise. The simulation results tell her that the building now performs worse than before. After undoing this first change, she flattens the slope of the roof, and the heating load drops considerably. She does not try rotating the building counterclockwise again now that she has improved the roof slope. Her first test suggested that this would be a waste of time. Were she to try it, though, she would find it to be the best option yet. There is a complex, non-linear relationship between rotation, roof slope, and heating load that the architect's linear steps could not uncover.

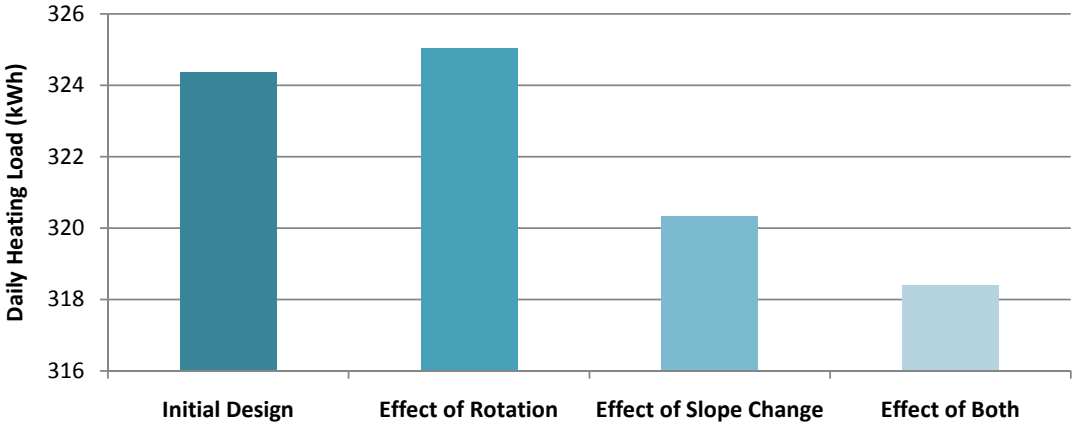

Figure 1.1.2 Every change to the one-room building model has an effect on its performance. The effect is more than the sum of its parts.

Complex relationships are difficult to understand, even when only a few parameters are involved. Architects generally prefer to examine one parameter at a time with the goal of improving a single aspect of a building. Analysis of a design in order to uncover the unintended consequences of early design decisions often takes place late in the design process, if at all. This is especially true for thermal analysis that predicts comfort conditions and energy consumption. In 2008, only 39% of architects reported that they usually received feedback from thermal analysis.[1] Ignoring thermal problems or fixing them in late design stages are expensive alternatives to preventing them with early analysis.

While designers usually avoid complexity, they can also take advantage of it. This thesis frames architecture as a *complex adaptive system* and shows how the tools that scientists use to understand complexity can advance a design. Tools called *genetic algorithms* analyze many properties of the design simultaneously and suggest variations that yield better performance. In the investigation reported here, genetic algorithms produce significant optimization improvement in an architect's parametric building model. They also provide a diverse set of solutions for continued development by the architect.

This chapter will describe the properties of complex adaptive systems in more detail and identify where those properties appear in architecture. Complex adaptive systems are made up of many interacting components. In buildings, each design change affects the building's overall performance in many ways, some positively and some adversely. This thesis will focus specifically on how design decisions affect the thermal performance and energy consumption of buildings. However, many other factors can also be considered, including aesthetics, circulation, structure, lighting, and the ability of occupants to carry out specific tasks.

[1] Autodesk 21

1.2 Complexity

Complexity is found in many places. Ecosystems are vast webs of complexly interconnected species. The economy is a complex network of transactions involving goods and services. Societies are complex organizations of human beings who interact with each other. Complex systems tend to give rise to one another and to be made up of smaller complex systems.[2] Since economic trade is a social interaction, and food and livestock are economic goods, it gets difficult to tell where one complex system ends and another begins.

Any complex system – any system for that matter – is a collection of interacting parts or particles.[3] Since the Enlightenment, scientists have studied these parts and particles in hopes of finding the natural laws that govern the larger systems. This reductionist view of the universe has been largely successful; it has discovered ever-smaller components of the cells that make up organisms and the particles that make up atoms. Despite this success, starting in the 1970's, scientists like physicist Philip Anderson became dissatisfied with reductionism:

> The ability to reduce everything to simple fundamental laws does not imply the ability to start from those laws and reconstruct the universe. In fact, the more the elementary particle physicists tell us about the nature of the fundamental laws, the less relevance they seem to have to the very real problems of the rest of science, much less society.[4]

Anderson's now classic example of reductionism's inability to grasp complexity is a collection of water molecules.

---

[2] Gell-Mann 369
[3] Beinhocker 18
[4] Quoted in Waldrop 81

> There's nothing very complicated about a water molecule: it's just one big oxygen atom with two little hydrogen atoms stuck to it like Mickey Mouse ears. Its behavior is governed by well-understood equations of atomic physics. But now put a few zillion of those molecules together in the same pot. Suddenly you've got a substance that shimmers and gurgles and sloshes. Those zillions of molecules have collectively acquired a property, liquidity, that none of them possesses alone. In fact, unless you know precisely where and how to look for it, there's nothing in those well-understood equations of atomic physics that even hints at such a property.[5]

Liquidity is an *emergent* property of a system of water molecules. Other properties, such as freezing, boiling, and the formation of whirlpools, clouds, and snowflakes, are also emergent; they do not exist at the level of the water molecule.

Now the mistake made by the architect in the first section becomes clear. She reduced her one-room building design into two separate problems – rotation and roof slope – and ignored any interaction that might occur between these variables. Heating load and energy consumption are not results of a single design decision. Rather, they are emergent properties of the system of components that make up a building and its environment.

1.3 Adaptation

Systems are either open or closed. A closed system receives no mass or energy from the outside universe. Its components eventually reach an equilibrium state, like a pool of water left in a closed container or an ecosystem deprived of sunlight.

[5] Waldrop 82

Removing one component of a system that is closed will quickly send it to a new static equilibrium state.[6] According to John Holland, the father of the genetic algorithm, "if the system ever does reach equilibrium, it isn't just stable. It's dead."[7]

Complex adaptive systems are open systems. Open systems receive energy and mass from outside, so they never have a chance to reach equilibrium. These systems are *dynamic*, rather than static. At their most basic level, complex adaptive systems are systems of energetic flows.[8] In the ecosystem, energy is transferred through the cycling of nutrients, water, and air. Genetic material passing from a parent to its offspring is energy in the form of information. In the economy, the transfer of goods and services constitutes a flow of energy.

A building is an open system. During construction, materials flow into a site along with information in plans from architects and contractors. Finished buildings take in energy from sunlight, air ventilation, and utilities such as electricity, natural gas, and water. The daily traffic of occupants is another dynamic flow fundamental to architecture. If a building is boarded up and its utilities are shut off, it ceases to exchange energy with its environment and becomes a closed system. Without the input of energy, buildings give in to entropy and crumble.

Likewise, architecture as a profession is an open system. Energy flows through architecture not in the form of sunlight, but as information through the exchange of ideas and drawings. Architecture does not settle on one style as an equilibrium form of expression. Rather, a continuous influx of new technologies and new practitioners puts the profession in a constant state of change.

Adaptation occurs when a change to an open system creates new patterns of energy and information flow. It is the emergent result of many new interactions

---

[6] Miller 9
[7] Quoted in Waldrop 147
[8] Kwinter 59

between the parts and particles that make up the system.[9] A change to one component of a complex system is all it takes to create entirely new patterns of interaction everywhere. Jim Drake and Stuart Kauffman call this the Humpty Dumpty effect. Remove a species from an ecosystem and the system will be thrown into chaotic disequilibrium as populations of other species rise and fall, having gained or lost predators or prey. Reintroduce that species and the ecosystem does not return to its original state but rather finds a new state, possibly sending other species into extinction.[10]

Adaptation to change is not unique to ecosystems. Imagine removing some good, say the can opener, from the world's economy. Manufacturers of canned food suddenly find no demand for their products, while bagged and lidded foods gain popularity. Recipes that call for no-longer-accessible canned items go out of favor. Restaurants develop new menus, and some go out of business. Agricultural production gradually shifts to meet changing demands. Eventually, new inventions replace the can and can opener, but eating and buying habits have already changed. In short, the economy adapts to meet a new set of internally imposed constraints.

A sudden change to a small part of a complex adaptive system results in a burst of adaptations. These moments when everything changes have a different name in each field that studies complex adaptive systems. They are punctuations for biologists, singularities for mathematicians, hinge points for archeologists, and for physicists they are *phase transitions*.[11]

Architecture is complex adaptive system both at the level of the building and as a system of design production. A change to one building component can have numerous unexpected effects on other building components and systems. Through

[9] Beinhocker 18
[10] Lewin 125
[11] Lewin 20

the design process, the architect adapts a building's design to meet certain needs, and each change alters the building's ability to meet other needs. Even after a design is solidified in one building, architects copy and adapt its pieces for new projects. Over time, successful new design components contribute to the development of architectural styles. Phase transitions occur when lighter, more durable, cheaper, or trendier building materials hit the market or as social tastes and expectations of architecture change. The growing interest in sustainability that caused, among other things, the writing of this thesis is no doubt the result of a phase change that is currently sending ripples throughout the architectural world.

The next two chapters give a more detailed account of how change occurs in complex adaptive systems. They show that adaptive mechanisms from other complex adaptive systems work in architecture as well. Chapters four, five, and six discuss the application of one mechanism, a genetic algorithm, to an architectural problem that serves as a test case. The final chapter comments on further applications of complex adaptive systems in architecture.

CHAPTER 2

THE SHAPE OF SEARCH SPACE

> The image of organisms striving to climb up local fitness peaks in this evolutionary landscape, which is constantly changing as a result of their own efforts so that they have to keep running just to stay in the same place, fitness-wise, provides a dramatic metaphor of life as a continuous struggle to improve merely to survive. This is sometimes seen as progress.
>
> Brian Goodwin, 1994

2.1 The Air-Conditioned Box

Return for a moment to the example from the first chapter. The architect still needs to reduce the heating, ventilation, and air-conditioning (HVAC) load in her one-room buildings. She decides to test the spectrum of possible design changes more thoroughly in order to find an optimal roof slope and angle of rotation on the site.

First she runs simulations to test the effect of changing one building's roof slope. Starting from the initial 50% slope, she reduces the slope by 2% for each test. She keeps all other variables constant, including the rotation angle of the building, which is currently 170°. From these tests, she finds that a suitably low HVAC load occurs when the roof slope is set to 10%.

Having chosen the roof slope, the architect then runs a series of tests using various rotation angles. For each test, she rotates the building 10° counterclockwise on the site until it has completed a full revolution. Again she keeps all other variables constant, including the roof slope which is now 10%. She finds that the smallest HVAC load occurs when the building is rotated near 270°. This comes as a surprise

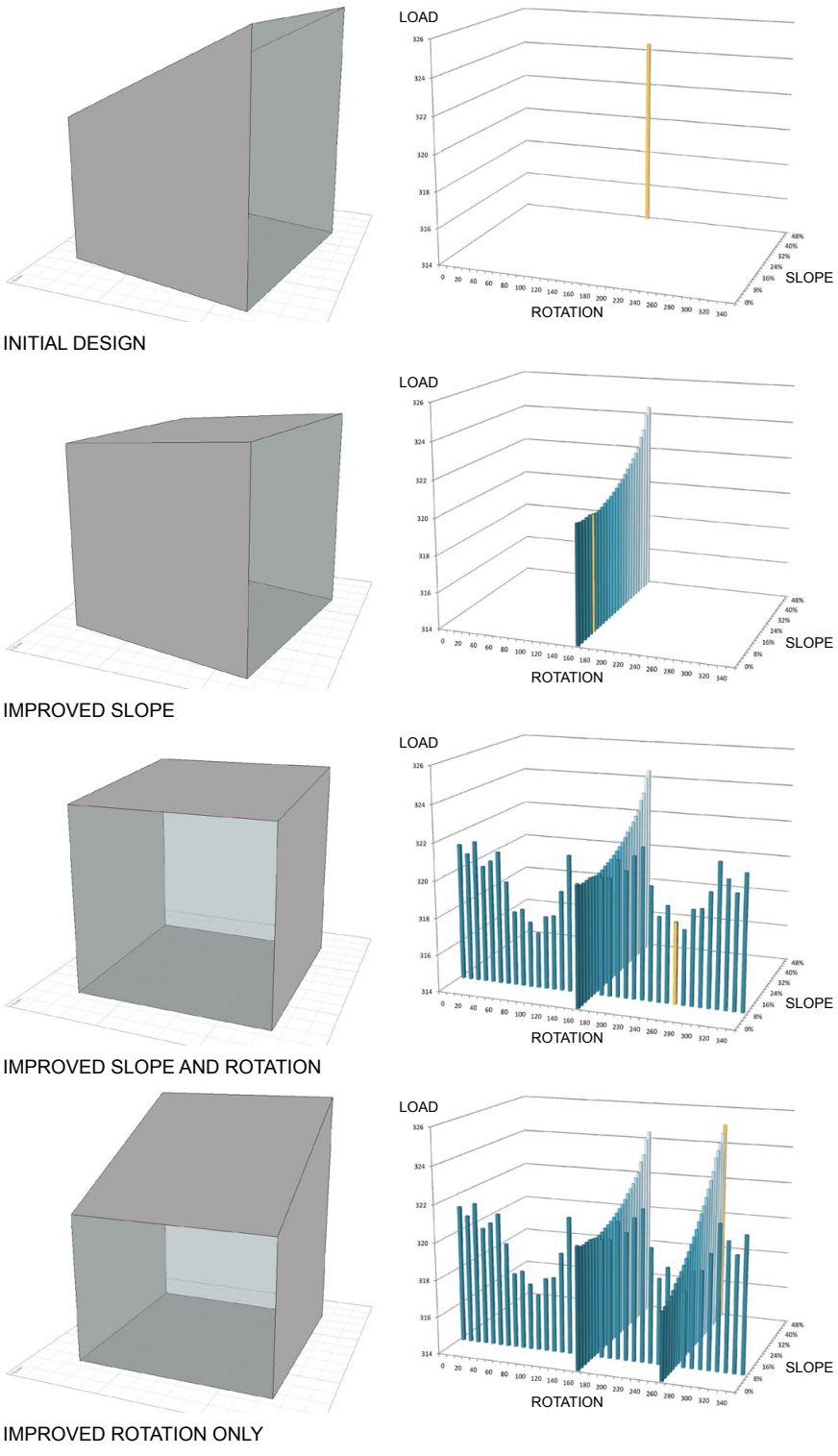

Figure 2.1.1 Linear optimization steps do not always lead to the best solution in multi-variable problems. Here, the effect of changing the slope at a given rotation angle cannot be predicted from data taken at a different rotation angle.

because in her earlier haphazard tests, the building faired quite poorly when she rotated it a quarter turn to 270°. She runs another set of tests, this time keeping the rotation constant at 270° and varying the roof slope. Sure enough, at this rotation angle, steeper roof slopes produce very high HVAC loads.

This is perplexing for the architect. If the test results at the first rotation angle do not predict the trends in HVAC load at other rotation angles, then she has no reason to believe that she has found an optimal solution. Instead, she must test every possible combination of rotation and roof slope to be sure that she has found the best one. In a time-consuming series of trials, she tests no less than 936 variations on her original design. Finally, she finds with absolute certainty that the best design has a roof slope of 14% and is oriented at 100° on the site.

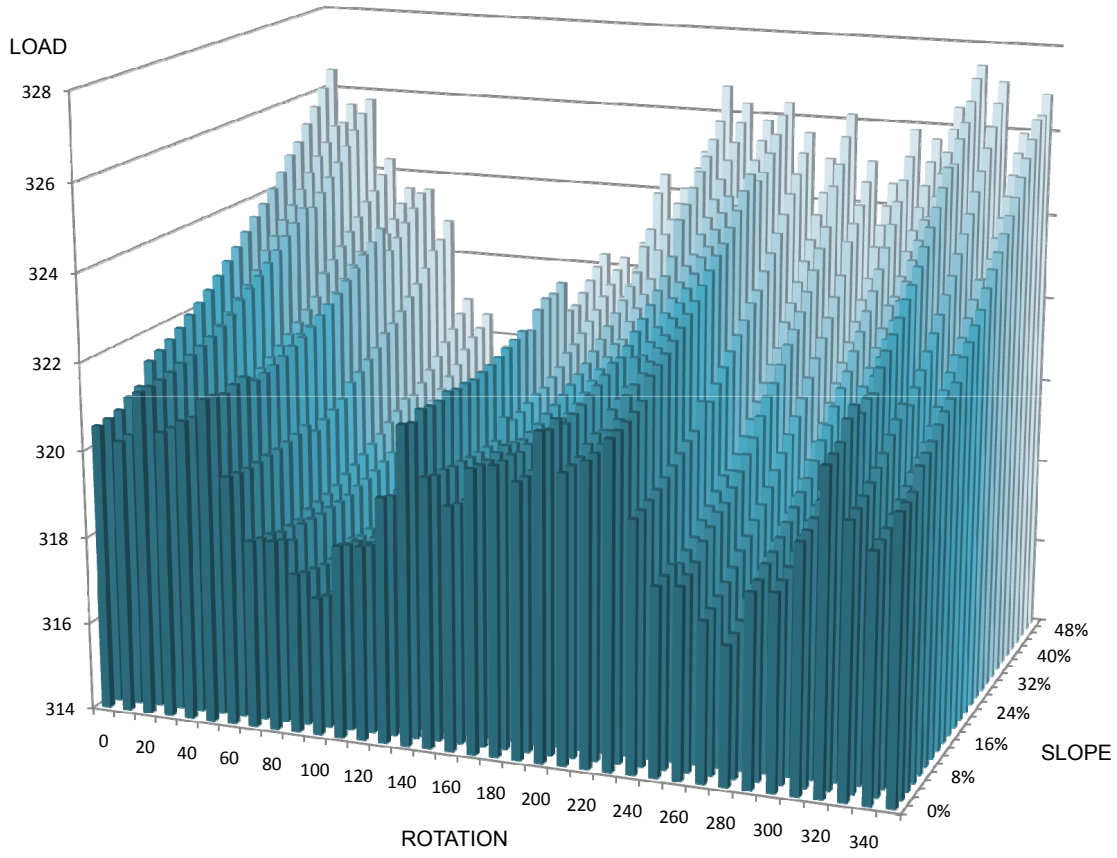

Figure 2.1.2 Plotting the HVAC load for each combination of rotation and slope reveals a complex fitness landscape.

2.2 Parameter Space

The first chapter of this thesis showed that complex adaptive systems such as architecture adapt to new conditions as long as they receive energy and information. Now it is time for a deeper look at how that change is possible, and how it can be beneficial.

The one-room building in the previous example is part of a complex system with two parameters (neglecting the climate variables used to run the simulation). The architect's model of the building is a *parametric model*, meaning that she can generate the 936 geometric variations of her initial design by adjusting the values of the two parameters. If these parameters are represented by the *x*- and *y*-axes of a Cartesian graph, then every possible design variant corresponds to a single point on the graph. The graph is a two-dimensional parameter space, or *search space*.

Any object with multiple parameters can be represented in a search space. One of these hypothetical spaces contains the parts and particles of any complex system. In biology, each parameter, or axis in the search space, represents the variation found in a single gene.[12] Individuals within a species differ from each other by more than two genes, so another dimension is added to the search space for each additional gene. As more and more genes are brought into consideration, the search space grows to encompass not just one species, but groups of species related by evolution, and potentially an entire ecosystem. The number of dimensions in the search space quickly surpasses human comprehension, but the principle remains the same. For the purpose of understanding complex adaptive systems, it is sufficient to visualize search space with two parameters.

The econosphere can be described in much the same way as the biosphere. A particular good, a brand of can opener for instance, occupies a point in a search space

[12] Wright 357

of technological products. Various axes in this multi-dimensional space describe the length of the handle, the thickness of ergonomic grips, the diameter of the cutting blade, the spacing between it and the gripping gear, and the presence of a bottle opener on the side. Within the entire search space are many varieties of can opener. Some of the devices may fail to work due to one parameter being too far from the norm; many of the possible can openers have never been produced in reality. Some points lying farther away represent electrical can openers. Farther away still, yet in the same search space, are the cans they open.

Because variation exists within man-made objects, these pieces of technology can evolve. It is easy to find relationships between similar products that copy biological evolution.

> We can see the same evolutionary patterns in generations of more modern technologies, such as automobiles progressing from the Model T to a modern car jammed with microprocessors, or mobile phones progressing from suitcase size to "so small I forgot I had it in my pocket" size. One can also see relationships between technologies that look very much like speciation – the airplane is related to hot-air balloons, dirigibles, and hang gliders in a sort of phylum of artifacts for flying.[13]

The terms that describe search space often reference their biological analogs. Each parameter, or dimension of the space, is a *gene*. The value taken by a gene at a given point in search space is an *allele*. The complete list of alleles at a point in search space is a *genotype*, and the corresponding physical object is its *phenotype*.

[13] Beinhocker 243

2.3 Fitness Landscapes

Jorge Luis Borges imagines a library containing a vast (but not infinite) collection of books, each differing from the next by just one letter. The library's "shelves register all the possible combinations of the twenty-odd orthographical symbols . . . : in other words, all that is given to express, in all languages."[14] Of course, all but a very few of the books contain pure gibberish. Yet within the library are rare books containing a recognizable word or even a lucid sentence. If the library is ordered and spread out over a single floor, these intelligible books stand out as clusters, and within each cluster is one fully readable book. One cluster even contains this thesis.

In complex adaptive systems, it is generally expected that some combinations of alleles will exhibit more desirable emergent properties than others. In Borges' library, this desirable property might be "meaning." For technologies in the econosphere, it may be utility. In biology, desirable traits are those that help an organism to survive and reproduce. Genotypes that possess these beneficial combinations are termed *fit*.

When the architect graphs the HVAC load for each variant of her design, she creates a *fitness landscape*. The elevation of each point in the landscape corresponds to its fitness. In her case, the more fit design alternatives occupy valleys, where the HVAC load is smallest. Conventions for relating elevation to fitness vary by discipline. In computer science, physics, and thermodynamics, smaller numbers indicate better fitness. Biologists and economists represent fit genotypes with hills. Since the architectural examples in this thesis deal with energy, they will use the physicist's convention.

[14] Borges 54

The concept of the fitness landscape is useful for understanding the behavior of complex adaptive systems. The hills (representing good fitness) of a biological fitness landscape correspond to the ecological niches within an ecosystem. In the economic fitness landscape, the can opener occupies one hill, and the electric can opener occupies another. Points in the valley between them represent less successful and unmarketable variants and hybrids of the two. In both cases, evolution pushes species or technologies from lower to higher points on the landscape. The next chapter will describe exactly how this occurs.

In the mean time, it is useful to discuss the shape of fitness landscapes in more detail. The simplest landscape has just one peak. All other parts of the landscape slope toward this point of maximum fitness, the *global optimum*. Sewall Wright, an evolutionary biologist and the first to describe fitness landscapes, noted that under such simple conditions, "a species whose individuals are clustered about some combination other than the highest would move up the steepest gradient toward the peak, and having reached it would remain unchanged."[15] No complex system is this simple, though. An ecosystem with one fitness peak would have room for only one species. An economy with a single peak could produce only one good. Instead, Wright goes on, "it may be taken as certain that there will be an enormous number of widely separated harmonious combinations" of alleles.[16]

The number of fitness peaks, or *local optima*, is a result of the number of connections that exist between genes. In other words, it is a function of the complexity of the system. In the single-peak case, the benefit obtained from a certain allele is unrelated to the value of any other parameter.

[15] Wright 357
[16] Wright 358

> Therefore, there is a special genotype having the fitter allele at each locus which is the *global optimum genotype*. Furthermore, any other genotype, which must of course have lower fitness, can be sequentially changed to the globally optimal genotype by successive flipping of each gene which is in the less favored allele to the more favored allele. Therefore, any such suboptimal genotype *lies on a connected pathway via fitter, one-mutant variants to the global optimum*.[17]

This simple fitness landscape is very smooth, with no significant difference in fitness between neighboring points in the search space. A completely smooth landscape is known as a *correlated landscape*. It allows adaptation to occur, but since it lacks any connections between genes, it is not complex.

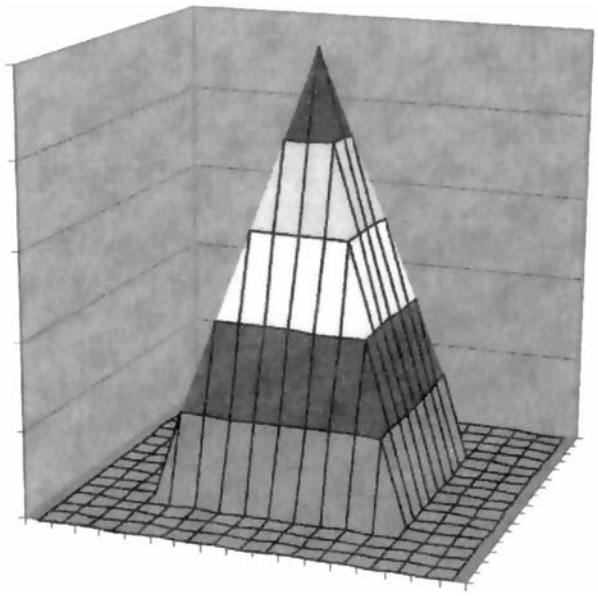

Figure 2.3.1 A correlated fitness landscape has a single peak and smooth slope. All points are similar in fitness to their neighbors. Image from Beinhocker.

Now consider the opposite condition in which the amount of benefit obtained from each allele depends on all the other alleles present. The condition in which one

[17] Kauffman 45

gene modifies the effect of another is *epistasis*. *Synergistic epistasis* exists when one allele increases the benefit received from another. *Antagonistic epistasis* occurs when an allele diminishes the benefit received from another. When "each gene is epistatically affected by all the remaining genes . . . the fitness value of one genotype gives no information about the fitness value of its one-mutant neighbors."[18] The resulting landscape is completely random and uncorrelated. It is very complex, but the lack of order makes adaptation impossible.

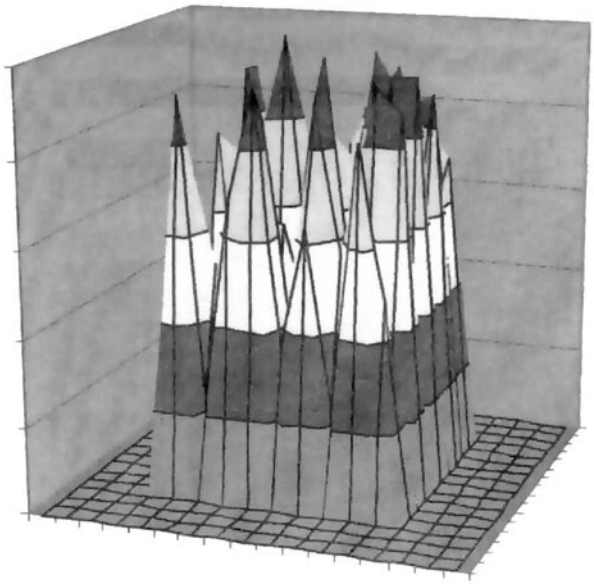

Figure 2.3.2 An uncorrelated fitness landscape has many peaks. The fitness of one point bears no relation to the finess of any of its neighbors. Image from Beinhocker.

As the number of epistatic relationships between genes increases from zero to the maximum, the fitness landscape becomes increasingly less smooth and more random. Complex adaptive systems occur between these extremes, on *rough-correlated* or *rugged fitness landscapes*. These landscapes have multiple local optima, separated by a mixture of smooth terrain and sudden cliffs. The existence of multiple optima and local areas of correlation makes adaptation possible in complex systems.

[18] Kauffman 46

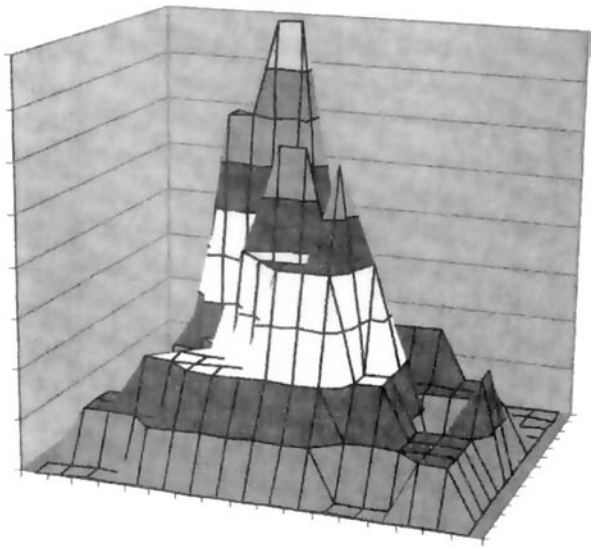

Figure 2.3.3 A rough-correlated fitness landscape has several local optima separated by a mixture of smooth and discontinuous terrain. Image from Beinhocker.

So far, the landscapes that have been discussed are static. However, the shape of a landscape itself may change over time, and epistatic relationships can arise and disappear. In the one-room building example, the fitness landscape discovered by the architect is a property of the simulated climate. If another climate is used, say one with colder temperatures or a higher sun angle, the shape of the landscape changes, and a different set of alleles will produce the fittest design. Changes to the landscape do not always come from external sources. For instance, the can opener is a fit product because, elsewhere in the economic fitness landscape, the aluminum can is also fit. As advancing technology changes the can's genes – the thickness and depth of the rim or the composition of its metal, for example – the fitness landscape under the can opener shifts to make a new design more practical. If the aluminum can is someday surpassed by a new technology, the can opener will become useless, no longer a local optimum at all. This phenomenon, *co-evolution*, also occurs in biology, where "predator and prey [are] constantly trying to keep one step ahead of the other." Leigh Van Valen terms this the Red Queen effect, after the character's line from Lewis

Carroll's *Through the Looking-Glass*, "it takes all the running you can do, to keep in the same place."[19]

The shape of its fitness landscape says a lot about a complex adaptive system. It shows what species, technologies, or even building typologies the environment favors. Unfortunately, fitness landscapes are very difficult to see. The architect's derived landscape for the one-room house is convenient because it has only two genes and a mere 936 genotypes. However, the size of the search space grows exponentially as more genes are added into consideration. Shear size and number of dimensions make it impossible to visualize most search spaces, let alone calculate the fitness of every possible combination of alleles. The next chapter will discuss search mechanisms that overcome these difficulties and are able to find high peaks without viewing the entire landscape.

[19] Lewin 58

CHAPTER 3

GENETIC ALGORITHMS

> The problem of evolution as I see it is that of a mechanism by which the species may continually find its way from lower to higher peaks in such a field. In order that this may occur, there must be some trial and error mechanism on a grand scale by which the species may explore the region surrounding the small portion of the field which it occupies.
>
> Sewall Wright, 1932

3.1 The Air-Conditioned Box, Again

Once more, the architect must minimize the heating, ventilation, and air-conditioning (HVAC) load of her one-room buildings. This time, she wishes to find an optimal genotype without taking the time to test every possible alternative. She knows that without searching the entire space, she is not guaranteed to find the best solution. However, she would like to have a high probability of finding the best solution, or at least a very good local optimum.

To start with, she picks random values for roof slope and rotation on the site. She runs the simulation on the initial genotype, and also on each of the four neighboring genotypes, those that differ from it by a minimal change of one allele. Next, she moves to the neighboring genotype that performed best and runs the simulation on its three untested neighbors. She repeats this process of moving and testing until she reaches a genotype that outperforms all of its neighbors. Because the landscape is rough-correlated, she is guaranteed to reach a local optimum eventually. However, since it is a complex landscape with multiple local optima, she does not know how the one she has found compares to other optima on the landscape. To find

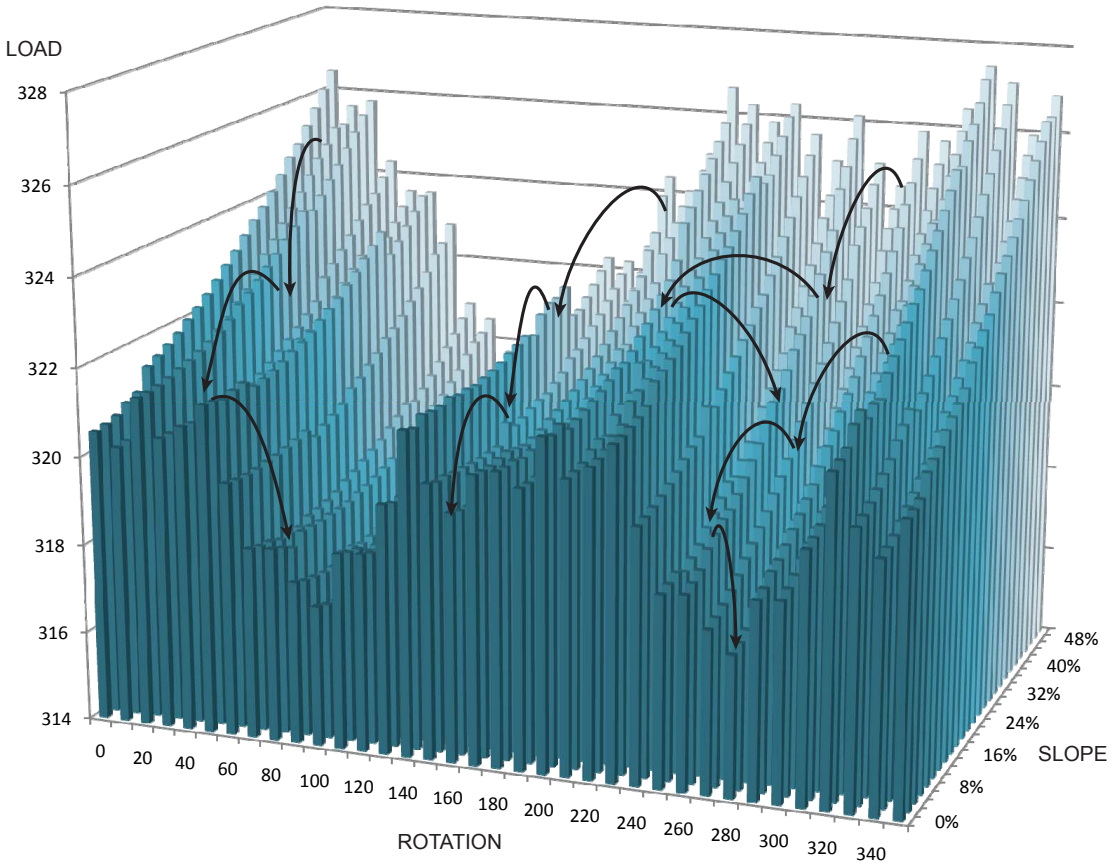

Figure 3.1.1 A fitness landscape can be explored quickly by an adaptive walk that starts at a random point and moves step by step to fitter neighbors.

out, she picks a new genotype at random and repeats the entire process, hoping to find a different local optimum. After several rounds of experimentation, she feels confident that she has found most of the local optima in the search space. Still, she has tested only a small fraction of the possible phenotypes.

3.2 Techniques for Exploring Search Space

In the previous example, the architect created a simple search mechanism called a hill-climber. The hill-climber is much faster than the brute-force search the architect tried in the last chapter. However, it has the obvious shortcoming that it only finds the nearest local optimum to its starting point. Other algorithms have been developed to solve this problem.

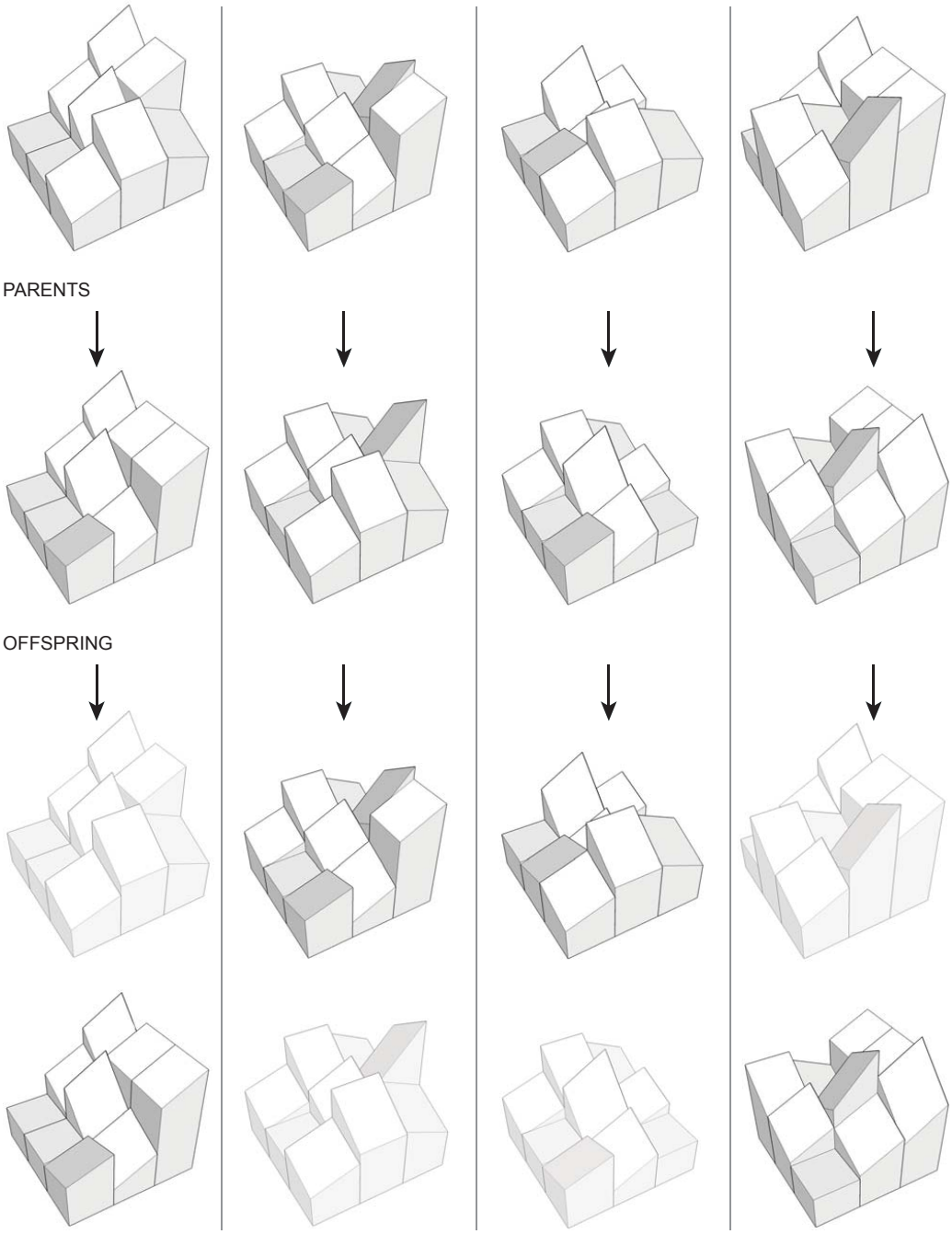

Figure 3.2.1 A parallel hill-climber starts with several random individuals and optimizes each along a separate path. Each parent gives birth to one slightly different offspring, and the best of the two becomes the next parent in its lineage.

One possible improvement is to start with a population of hill-climbers rather than a single one. An algorithm that searches this way is called a *parallel hill-climber* (PHC). When multiple hill-climbers are present, the ones that reach the same optimum are said to be in that optimum's *basin of attraction*. It is usually expected (though it cannot be proven) that the landscape's highest peak will have the largest basin. Given a large enough population of hill-climbers, at least one is likely to start within the global optimum's basin of attraction.

Sometimes, regions of the fitness landscape are quite flat, which can lead hill-climbers to wander aimlessly, or *drift*. One solution to this is an algorithm called *simulated annealing*. Simulated annealing starts with a high mutation rate, which lets the searching agents effectively hop through flat regions in search of steeper terrain. Over time, the mutation rate is gradually lowered, so that individuals explore the landscape in more detail when they are likely to be near a peak.[20] Simulated annealing also gives individuals a small probability of taking a step in the wrong direction. This allows the algorithm to avoid getting stuck on low peaks that may exist near a better optimum.

Another method for avoiding low peaks supposes that each local optimum has some but not all of the most beneficial alleles. The *genetic algorithm* (GA) allows recombination, where two previously tested genotypes are combined to create a new genotype.[21] If each of the *parent* genotypes contains a block of beneficial material, then there is some hope that the *offspring* genotype will inherit both beneficial pieces. In this way, two parents that have reached low fitness peaks can produce an offspring that starts out at a higher point. Later chapters in this thesis will explore the use of GAs in architecture in more detail.

---

[20] Kauffman 112
[21] Holland 70

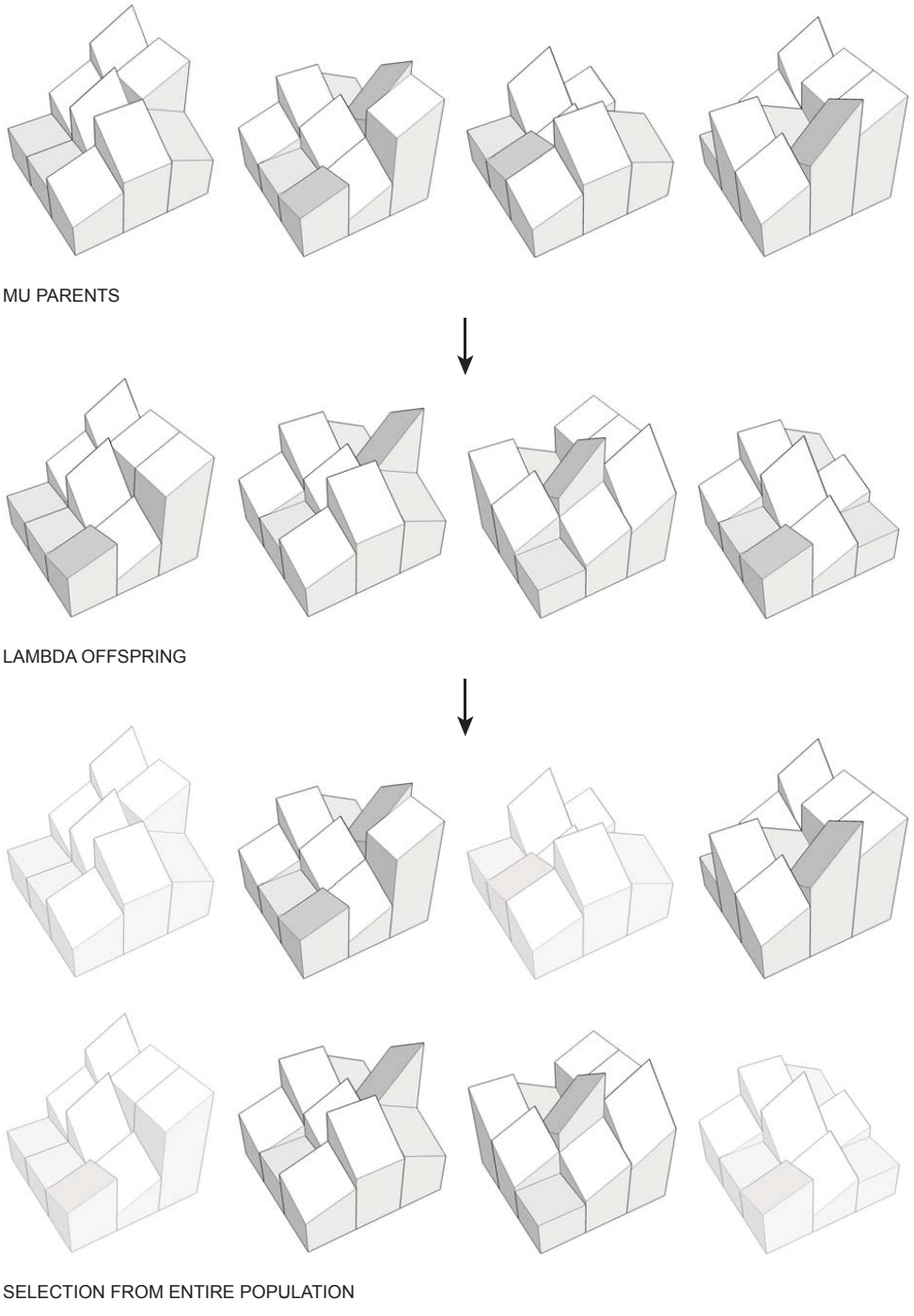

Figure 3.2.2 In a genetic algorithm, parents mate to produce offspring that inherit genes from both parents. One parent may have multiple offspring that are chosen to become new parents.

John Holland, who first described the GA, used it to mimic biological evolution. One shortcoming of search space in this field is that it forces a one-to-one correspondence between gene and trait. If organisms actually evolved this way, biological features such as symmetry and repetition of parts would be astoundingly improbable. For instance, the homologous structures of the arms and legs would have had to evolve independently of each other. Holland's student John Koza replaces the search space from the GA with a *rule space* in *genetic programming* (GP). The genotype that evolves in GP is a set of rules that act to alter a seed. The seed may be an object, a virtual creature, a neural network, or even the code of a computer program. GP is capable of developing much more complexity than a GA with a similarly sized search space.[22]

Simulations of social systems also typically use a rule space. In these models, the rule space describes the behaviors of individuals, or *agents*, in the artificial society. Successful agents (those whose behaviors help them to find food, gain wealth, or otherwise meet some goal) produce offspring with some genetic variation, while failed agents eventually die. Agents following simple local rules in a simulated world produce emergent global behaviors such as migration, cultural transmission, war, epidemics, and trade.[23] Modeling of complex social situations can even provide strategies for conflict resolution.[24] In these simulations, fitness is not an explicit property of a point in rule space. Rather, it is an implicit property of an agent based on the ability (or failure) of the agent to survive. Simulations that use this Darwinian definition of fitness are called *artificial life* (AL).

This is by no means a definitive list of methods for exploring search spaces, but it is useful to see the variety of optimization algorithms based on the concept.

---

[22] Koza 73
[23] Epstein 15
[24] Axelrod 110

Phenotypes do not have to have spatial forms, but all of the techniques discussed here lend themselves to spatial problems. The algorithm that an architect chooses to explore the search space should reflect the complexity of the desired solution. Agent-based models can arrive at large-scale spatial arrangements that satisfy many competing interests, while a hill-climber may be all the architect needs for the simple example of the one-room building. Most architectural problems will fall somewhere between these extremes.

3.3 The Evolutionary Mechanism

While the optimization methods discussed so far differ in their details, all rely on the same basic mechanism for exploring search space. This mechanism, *evolution*, is common to all complex adaptive systems. Evolution works by iterative repetition of four processes: translation, evaluation, selection, and variation.

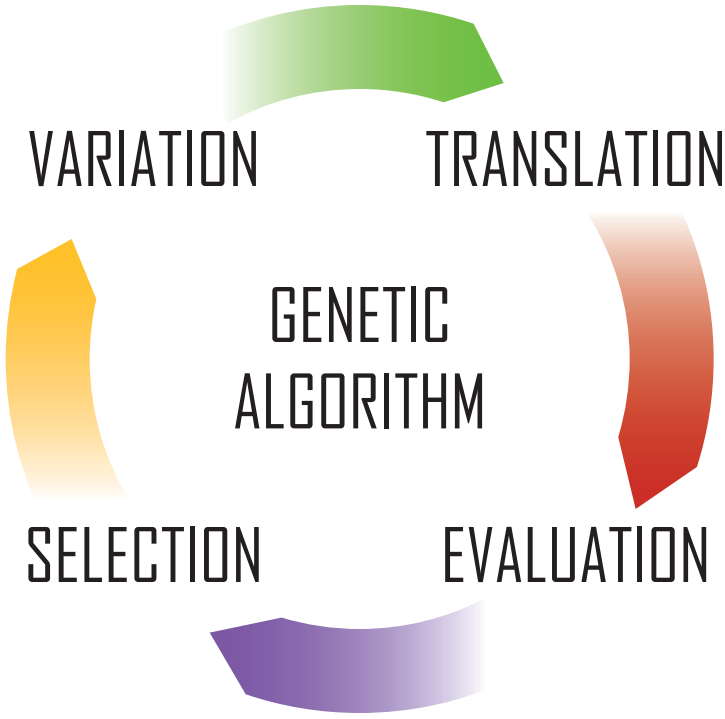

Figure 3.3.1 Evolution occurs by a cycle of translation, evaluation, selection, and variation.

The first step of the cycle is the *translation* of a genotype to create a phenotype. The biological transcription mechanism should be familiar to anyone who has completed high school biology. An organism's genotype is carried by its DNA. The A's, C's, G's, and T's that make up a molecule of DNA are the genetic code's *schema*, or method of storing information. This information is transcribed to RNA and read by ribosomes, the body's *schema readers*, which create the proteins and enzymes that make up the organism's phenotype. In man-made complex adaptive systems like economy, genotypes may not be written out so explicitly.

> Again, our standard is not that the schemata contain so much information that just anyone can [read them], but rather that a qualified reader could read the plans and make the object or provide the service. Thus, we would expect that a team of qualified house builders could use the [schemata] for house building to render the design for a house, or that a team of Eli Lilly scientists and technicians could use a set of pharmaceutical [schemata] to make the drug raloxifene.[25]

These schemata define the building block components and assembly methods that create a technological artifact, just as DNA codes for the protein building blocks and enzyme assembly tools that create an organism. The genetic code of a man-made object can be thought of as a description of its parts and assembly sufficient to differentiate it from other objects. "Think of an automobile {V6 / 1.8 liter / 200 horsepower / four-door / leather seats / etc.} or a personal computer {2 GHz Pentium 4 / 128 MB memory / 80 GB hard drive / 4X CD burner / etc.}."[26]

[25] Beinhocker 245
[26] Beinhocker 247

Once a collection of phenotypes exists, each one undergoes *evaluation*. The choice of evaluation criteria is not important to the evolutionary mechanism. It is sufficient that fitness criteria exist. In biology, fitness is the ability to produce viable offspring, and more offspring mean higher fitness. In the economy, a good or service is fit if it meets some demand. Evolution performed outside of the real world's testing environment requires an explicit measure of fitness. For instance, the architect in the one-room building example used a simulation of HVAC load to evaluate her designs.

Having established the fitness of the population's members, evolution requires that desirable members be *selected*. Selection exists because of scarcity. Were the resources needed for life or for product manufacturing limitless, there would be no need to select only some individuals for propagation. When evolution takes place outside of real-world selective pressures, it is necessary to impose scarcity by other means. Artificial scarcity is introduced by limiting the number of individuals that may be selected.

Evaluation and selection are closely related in some complex adaptive systems. In biology, selection is synonymous with reproduction. Goods and services that are selected in the economy are produced at higher rates and become objects of scrutiny for potential improvement (and knock-offs). In these complex adaptive systems, there is no strict limit to the number of individuals that will be selected. There is also no guaranty that any will be selected, so a species may go extinct. Such systems are *nondeterministic*, meaning that selected individuals may not be the best in terms of explicit fitness criteria. Think, for example, of the selection of VHS technology over Betamax.

Finally, the selected individuals produce offspring with *variation*. While variation in biology is the result of random occurrences, variation in the world of human invention is frequently purposeful. It is important to note "that there is nothing

fundamental in the nature of the evolutionary algorithm that says intentionality and rationality cannot play a role, nor does anything say the process must be completely random."[27] Evolution requires only that variation exist in the population.

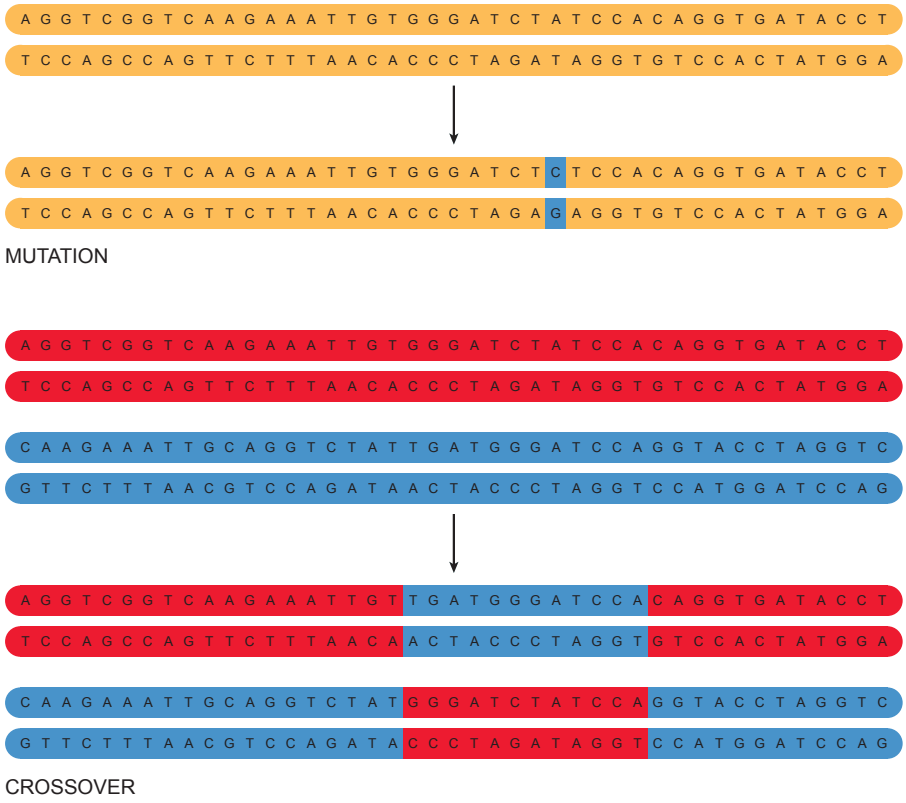

Figure 3.3.2 In biology, variation is produced by mutation, in which an error is made in copying DNA, and crossover, where chromosomes exchange a DNA segment.

The biological world has two means of producing variation. *Mutation* occurs when an accident in copying the genome gives the offspring an allele that neither parent possesses. *Crossover* occurs when genetic material is swapped between two chromosomes, creating new chromosomes that neither parent has but that do not necessarily contain new alleles. As a result of crossover, genes that were previously inherited independently from each other become linked. This is useful if by chance

[27] Beinhocker 249

the two genes have a synergistic epistatic relationship. Together, mutation and crossover are responsible for all of the variety in the living world.

> The action of natural selection has often been compared to that of an engineer. This, however, does not seem to be a suitable comparison. First, because in contrast to what occurs in evolution, the engineer works according to a preconceived plan in that he foresees the product of his efforts. Second, because of the way the engineer works: to make a new product, he has at his disposal both material specially prepared to that end and machines designed solely for that task. . . . [Natural selection] works more like a tinkerer – a tinkerer who does not know exactly what he is going to produce but uses whatever he finds around him whether it be pieces of string, fragments of wood, or old cardboards; in short it works like a tinkerer who uses everything at his disposal to produce some kind of workable object.[28]

People generate variety in the economic fitness landscape with a mix of the engineer's rationality and the tinkerer's opportunism. In this *deductive-tinkering* approach, the area of search space around the current design genotype is well understood by science. Within that area, the engineer can predict how phenotypes will behave before purposefully creating them. Beyond it, lies a region grasped by "unconscious inductive cognitive processes, associative thinking, and reasoning by analogy."[29] This is where the tinkerer operates. These two regions define the extent of not-quite-random variation available to economic evolution.

---

[28] Jacob 1163
[29] Beinhocker 250

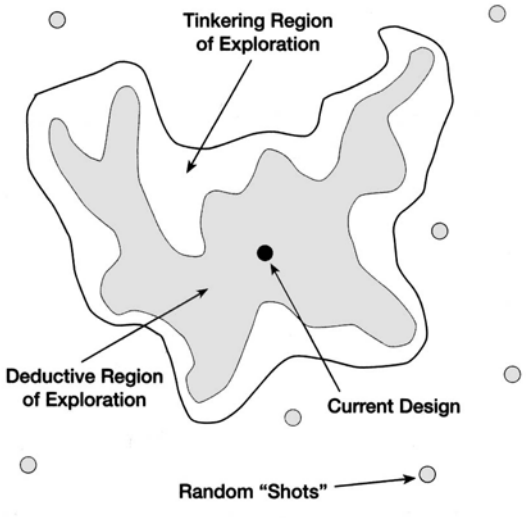

Figure 3.3.3 Variation occurs in man-made artifacts through a deductive-tinkering approach. A small part of the search space has predictable fitness, while the area around it is open to investigation by tinkering. Image from Beinhocker.

Having produced offspring with variation, the evolutionary cycle repeats itself, translating the offspring genotypes, evaluating their phenotypes, and selecting those that will parent the next *generation*. It is not a metaphor to say that economic goods and services evolve. The evolutionary mechanism, the genetic algorithm, powers adaptation in all complex adaptive systems. The patterns it produces in biological evolution are found in other complex adaptive systems as well. For instance, 550 million years ago, the diversity of early multi-cellular life skyrocketed in a period known as the Cambrian explosion. Over time, this diversity was lost as most of the hundred or so phyla born in that explosive period went extinct.[30] The same pattern occurs in technological artifacts.

> Think of the first bicycles or the first cars. Lots of experimentation to begin with, different forms of bicycle, different forms of propulsion and design for cars, all viable. As time goes on and the world gets full of cycles or cars, or whatever it is you're thinking of, the extremes get weeded out, a few forms survive, and subsequent innovation focuses

[30] Kauffman 76

> on improvement on the remaining themes. You go from generation of many themes to variations upon a few, just like the Cambrian.[31]

The result is the characteristic shape of the *fitness curve*. When the fitness of the best individual of each generation is plotted, the rate of improvement is initially very high. The first genotypes tried out by evolution are diverse, random, and not very fit. Improvement comes easily and there are plenty of alleles to put together in various combinations. As time goes on, selection weeds out many realms of possibilities, the population becomes more homogeneous, and improvement slows. The number of mutations that result in upward movement on the fitness landscape dwindles with each step forward.[32] Occasionally, a useful mutation gives rise to a burst of creative activity. This is the *punctuated equilibrium* model of evolution.

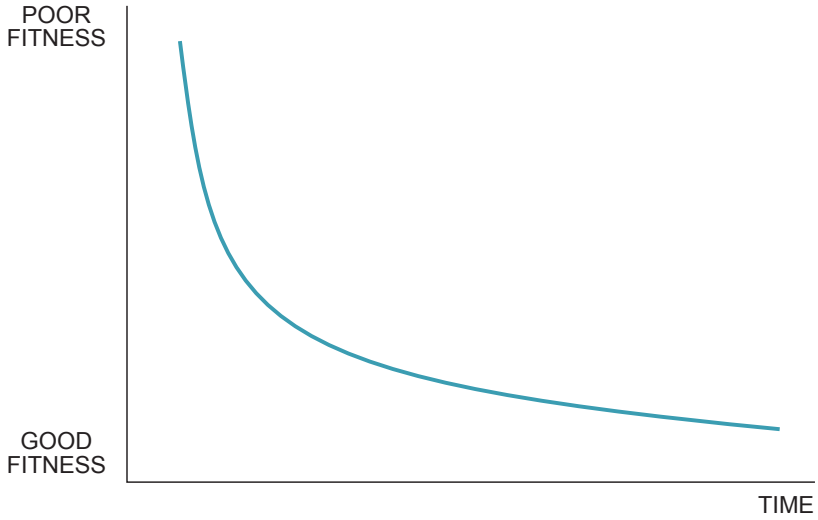

Figure 3.3.4 A typical fitness curve shows fast progress initially. Over time, it becomes harder to find new improvements, and the curve levels.

[31] Kauffman, quoted in Lewin 70
[32] Kauffman 44

3.4 Appropriate Algorithms

The word *algorithm* refers to a set of specific instructions, usually given to a computer. Many algorithms could be written to carry out the process of the evolutionary mechanism. Each GA brings together methods of translation, evaluation, selection, and variation like pieces of a puzzle. Some methods work together better than others, just as some puzzle pieces are closer fits. No one GA will be useful for every complex optimization problem. The architect interested in optimization must be careful to assemble an appropriate algorithm for the job.

First the architect must choose an appropriate schema, or means of representing the phenotype as a genotype. When variation is introduced by mutation only, the genetic information can be encoded in any order. However, as soon, as

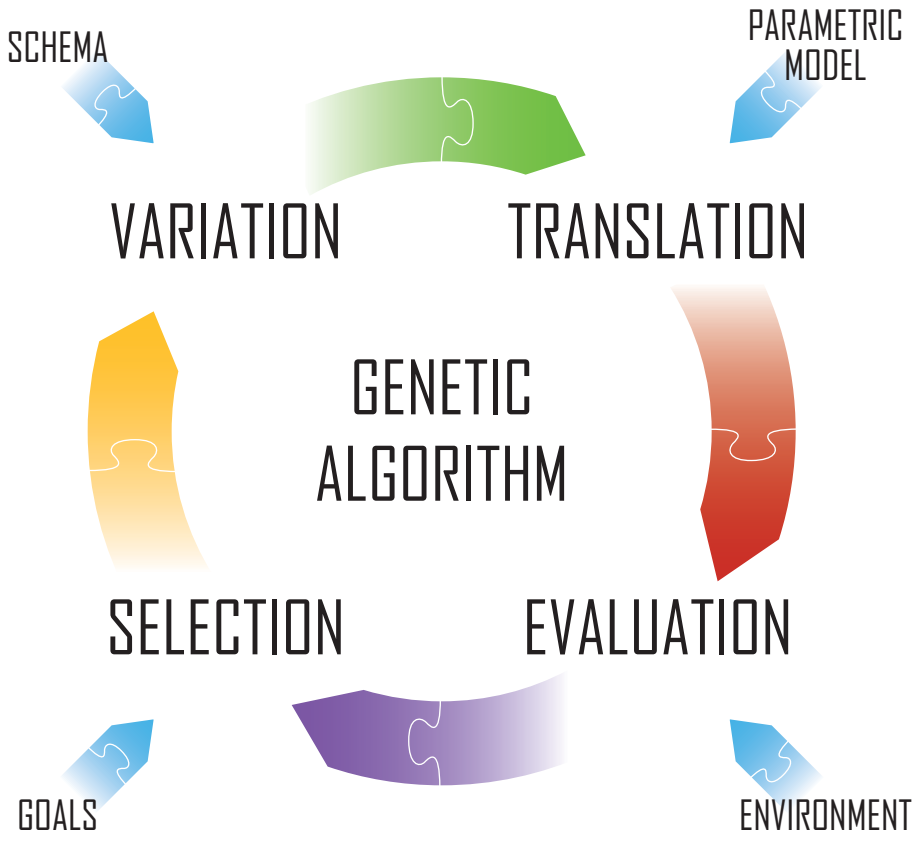

Figure 3.4.1 An effective genetic algorithm needs to be assembled from pieces. Different methods are appropriate to different problems.

crossover is introduced, order matters. A good schema packs genes that share an epistatic relationship into "short, highly fit combinations."[33] This *linkage* means that once a beneficial group of alleles is formed, it is less likely to be broken up during crossover. Epistatis sometimes occurs unexpectedly, so a schema may need the flexibility to link any pair of genes. Often, the phenotype in architectural or other spatial problems is composed of smaller building blocks. The schema's structure should mirror the composition and hierarchy of the phenotype's components.

When a computer scientist writes a GA, the genotype is usually a string of bits, or ones and zeros. This representation is not particularly helpful to the architect. Depending on what the genes represent, it may serve the architect to use either integers or to allow continuous variation of parameters. Since spatial relationships in the phenotype occur in three dimensions, storing genes in a three-dimensional array rather than a one-dimensional string may also produce better linkage. Alternately, a GP technique may be more appropriate to the problem since it can evolve the instructions for assembling a building. Genetic material in a GP is typically stored in a hierarchical tree. This schema may allow architects to model buildings that involve modularity.

The chosen schema will lend itself to certain types of mutation and crossover. For instance, the computer scientist's bit string can be mutated by changing a bit from a one to a zero or vice versa, by adding or deleting a bit, or by changing the order of bits. In a schema using integers, an algorithm that mutates by replacing one allele with a random integer will behave differently from an algorithm that mutates by adding increments to alleles. The former will explore a broader expanse of search space, while the latter may discover local optima more quickly. Likewise, different methods of crossover work with different schemata. Two bit strings can be crossed

[33] Goldberg 416

with each other by splicing them at one point and swapping their ends, or by splicing them at two points and swapping their middle sections. The latter approach allows the algorithm to take advantage of epistasis that may exist between the first and last genes of the sequence. Genetic information stored in a multi-dimensional array does not lend itself to point crossover. A more convenient approach involves dividing the array into halves for crossover. The choice of what planes to allow cutting on affects the tightness of linkage and the space open to exploration.

Trees in GP schemata require yet another crossover method. Typically, crossover in trees involves replacing a branch of one tree with a branch from another. If the two spliced branches do not come from the same hierarchical level of the tree, the schema is not of a fixed length, and the number of parts in the genotype (and phenotype) may grow or shrink. This is useful if the desired solution's size is unknown, but it can also lead to *bloating* when the solutions become larger than necessary.

In addition to selecting a schema, the architect must choose an appropriate selection mechanism. Holding population size constant is a simple way to create selective pressure. In a population of mu (μ) parents and lambda (λ) offspring (where μ and λ are numbers), only μ spots are available in the next generation. In *steady state* or $\mu+\lambda$ selection, the best μ are chosen from the entire population. In *generational* or $\mu-\lambda$ selection, only offspring compete for the μ spots in the next generation. An algorithm that is too selective may eliminate *diversity* from the population in the process of eliminating its less fit members. Once a population becomes homogeneous, crossover is no longer beneficial, and no improvement can occur until a new beneficial allele is discovered by mutation. Larger population sizes and higher mutation rates improve diversity, but at a cost. The more diverse a population is, the lower its average fitness is expected to be.

Some other selection methods increase diversity at the expense of slowing improvement. *Niching* methods split the population into smaller mating groups that evolve separately and are later brought back together. *Crowding* methods cause fit offspring to replace similar individuals rather than the least fit members of the population. In *deterministic crowding* (DC), each parent pair has two offspring, and each parent competes against its own most similar child to enter the next generation. *Fitness proportional* selection preserves population diversity by randomly picking individuals to advance to the next generation, where the probability of choosing a certain individual is related to that individual's fitness. The choice of selection method determines the speed at which evolution progresses and also the likelihood getting stuck at a local optimum.

Occasionally, two members of a population become so different from each other that any crossover between them will break some epistatic pairing of alleles. The inability of fit parents to create fit offspring is called *speciation*. Speciation slows evolutionary progress because it decreases the number of fit offspring in a population. In a fixed-size population, the smaller of two equally fit species is usually driven to extinction. Speciation occurs only in diverse populations, but it is not a reason to prevent diversity.

Poorly chosen evaluation criteria may reduce a population's diversity. GAs will exploit any discovery that gives advantage to an individual, including any assumptions made by the programmer. This can happen when fitness is determined by an artificial valuation of competing goals or by a made-up formula. An evaluation method that contains bugs or does not accurately reflect real-world selective pressures will give high fitness values to useless solutions.

Often, more than one criterion may be used to evaluate a phenotype. The architect in the example could also have considered interior daylighting, roof drainage,

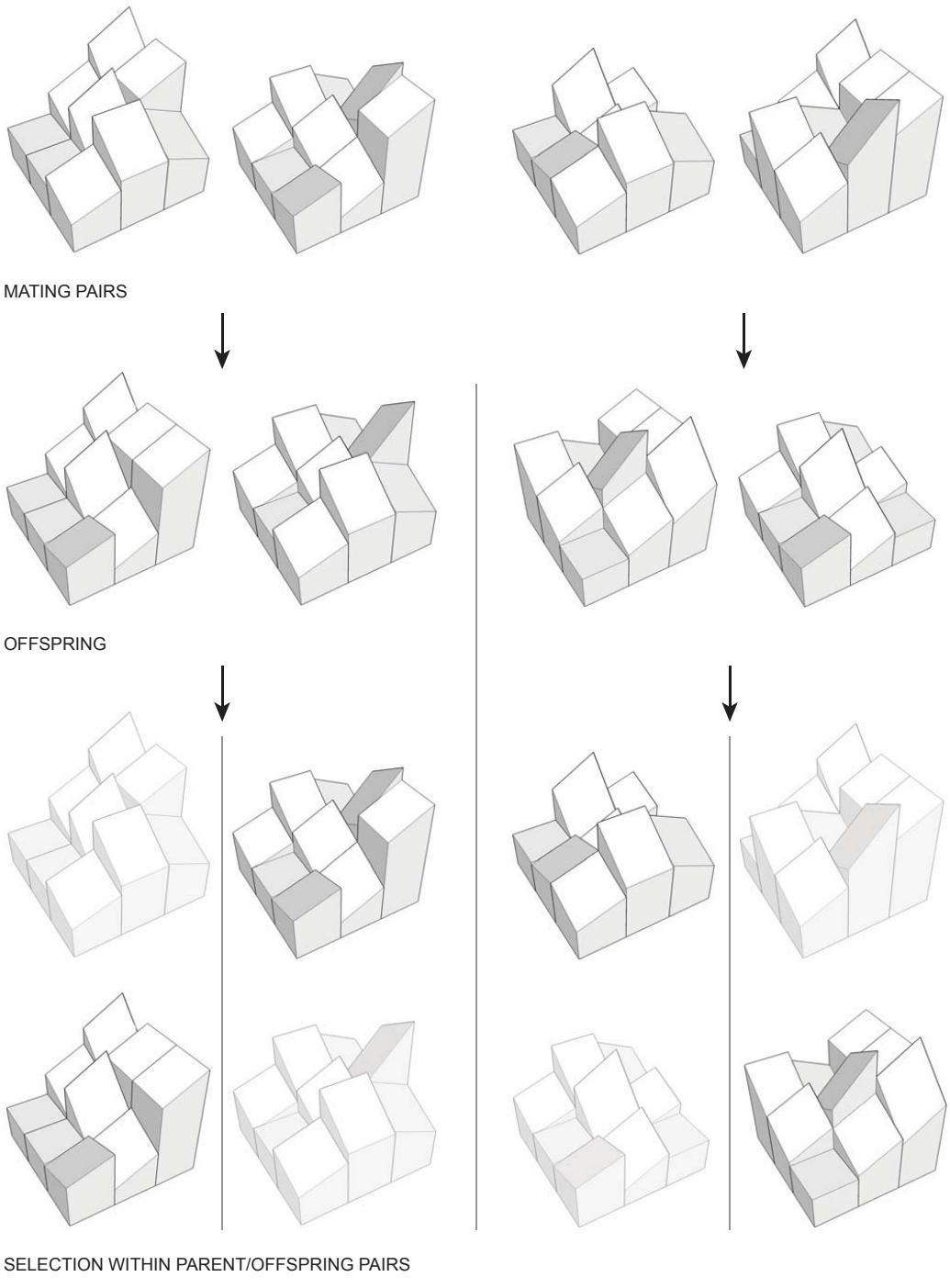

Figure 3.4.2 Deterministic crowding breaks the parent population into pairs for mating. Each pair produces two offspring. Each parent is matched with its most similar child, and the more fit of the two advances to parent the next generation.

views, or aesthetics instead of HVAC load. A number of techniques exist for *multi-objective optimization*. One idea is to assign a weight to each criterion. Unfortunately, little information is available to help assign weights before the first evaluation, and the wrong set of weights can bias evolution away from desirable outcomes. Alternately, each fitness criterion may be graphed on a separate axis, so that the most fit individuals are farthest from (or closed to) the origin. The best individuals, all with different strategies for achieving fitness, occupy a *Pareto front* on the graph.

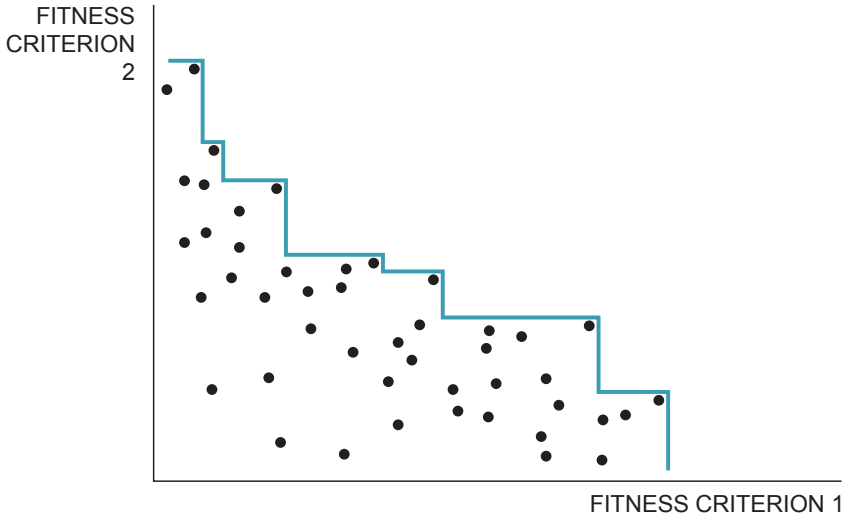

Figure 3.4.3 A Pareto front connects the best individuals in a population optimized for multiple fitness criteria.

Sometimes a GA is used to solve a broad problem where the fitness criteria describing a general solution are unknown. Varying the particulars of the fitness criteria over time can maintain diversity and prevent the population from converging on a solution that fits only one instance of the problem. In *co-evolution*, a population of fitness criteria evolves together with the population of solutions. Fitness criteria are selected for their difficulty, so that the solutions do not reach a local optimum too quickly.

The first trial of a new GA will often reveal problems with linkage and diversity. The fact that the algorithm compiles and runs on a computer is not enough to guarantee that the solutions it finds are useful. Often, several puzzle pieces must be tested out in the GA to make the algorithm more successful. Only analysis of the results can reveal if the GA is finding reasonably high peaks in the fitness landscape.

3.5 Uses in Architecture

It is no exaggeration to say that the evolutionary mechanism is already at work in architecture. Observe the practice of a typical architect's office; three variations of a design are shown to a client one day, and in the next week three variations of the client's favorite are produced, and so on in an iterative manner. When the algorithm is applied to computer simulation, every variation may not be as well thought-out as in the old-fashioned architect's office, but the computer has the potential to create thousands more variations and evaluate them against many more criteria in less time.

Most applications of computerized optimization algorithms in architecture focus on structure. This is in part because the fitness of a structure is easily evaluated in terms of strength or strength-to-weight ratio. A GA optimizes the forms of Pablo Miranda Carranza's "self-designed structures." In these bridges and cantilevers, the members co-evolve as they compete for material and load.[34] Tom Wiscombe's group EMERGENT emulates biological processes to create structures. Wiscombe evaluates his structures for "both performance and spatial and atmospheric effects," making his fitness determinations partly subjective.[35]

Thermal and environmental characteristics of buildings receive much less attention in algorithms. These characteristics include incident solar radiation, interior daylighting, internal heating and cooling loads, and heat gains and losses through the

[34] Carranza
[35] Wiscombe

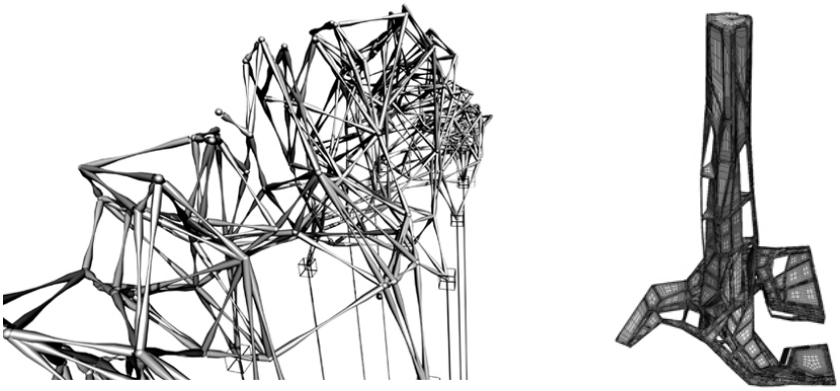

Figure 3.5.1 Self-Designed Structure by Pablo Miranda Carranza (left) and *Cheongna City Tower* by EMERGENT (right) are structures created with genetic algorithms.

building envelope. Analysis of thermal properties requires a lot of information, so it is generally carried out late in the design process, after most of the design decisions that affect it are finalized. Demand for sustainable architecture has increased interest in thermal analysis in recent years. At the SmartGeometry 2008 conference in Munich, Kaustuv De Biswas introduced a simple GA that optimized a tower for insolation using software he developed.[36] The remainder of this thesis will focus on optimization made possible by this software.

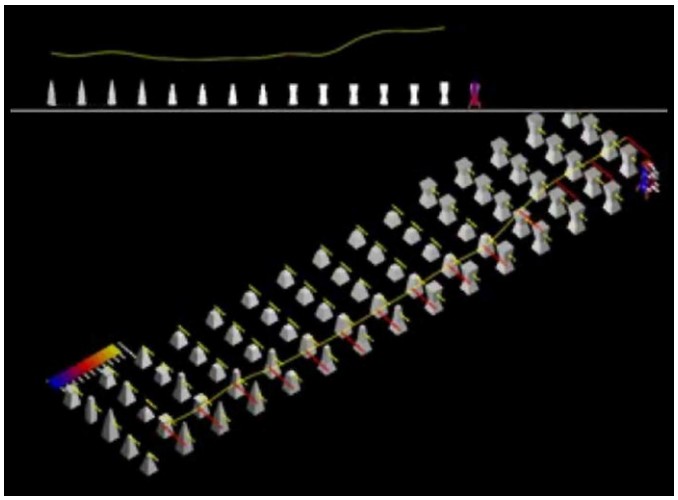

Figure 3.5.2 A population of simple building envelopes evolves to maximize insolaiton in De Biswas's SmartGeometry 2008 presentation.

[36] *SmartGeometry*

CHAPTER 4

TEST CASE

> Of course, specialization is necessary today. But so is the integration of specialized understanding to make a coherent whole, as we discussed earlier. It is essential, therefore, that society assign a higher value than heretofore to integrative studies, necessarily crude, that try to encompass at once all the important features of a comprehensive situation, along with their interactions, by a kind of rough modeling or simulation.
>
> Murray Gell-Mann, 1994

4.1 Software Environment

Architecture is a complex adaptive system where optimization is possible, so an evolutionary design approach should offer useful ideas to the architect. Currently, no commercial software lets architects evaluate the energy-efficiency of buildings at the conceptual stage of design. This prevents architects from exploring the search space of architectural form using sustainability as a fitness criterion. To gain new insight into sustainable form, architects need a genetic method of form generation and a suitable tool for analyzing the thermal and environmental implications of a form.

Ecotect is a building analysis software tool intended to be used starting in the schematic phase of design. The software models a building as a collection of polygonal surfaces with associated materials that create a set of closed volumes, or *zones*. Ecotect includes functions for analyzing a zone's thermal and acoustic properties, as well as estimating the building's solar exposure and cost. Since many decisions have yet to be made in schematic design, Ecotect uses heuristics to fill in

missing information about a building. For instance, a default material is assigned to every surface until the user specifies actual materials, and surfaces are automatically treated as walls, floors, or ceilings depending on their orientation.[37]

One major disadvantage of Ecotect is its simple modeling interface that can only generate rectangular prisms. In response to this, De Biswas developed a dynamic-link library (DLL) that allows the versatile associative parametric modeling program GenerativeComponents (GC) to communicate directly with Ecotect through Microsoft's COM interface.[38] The link regenerates geometry from GC in Ecotect and returns analysis results to GC.[39] This process can be automated using GCScript, GC's C#-based scripting language. Through the DLL, the entire library of Ecotect commands is available within GCScript. De Biswas has shown that a genetic algorithm (GA) can be written in GCScript using Ecotect for fitness evaluation.

This investigation builds on the work by De Biswas. It uses the software link he developed, but introduces more realistic fitness criteria and a much larger search space. The test subject is a parametric model of a house created in GC. Ecotect evaluates each house in a population of design variants to determine the amount of energy required for lighting and heating or cooling during its worst-case month. Individuals requiring the least energy are selected to parent the next generation using deterministic crowding (DC).

4.2 Genotype and Phenotype

The phenotype in the optimization procedure is a house built on a nine-square grid. This design problem has a rich history at Cornell University and inspired architects such as Colin Rowe, Peter Eisenman, and John Hejduk. Each square of

[37] *Ecotect*
[38] *GenerativeComponents*
[39] De Biswas

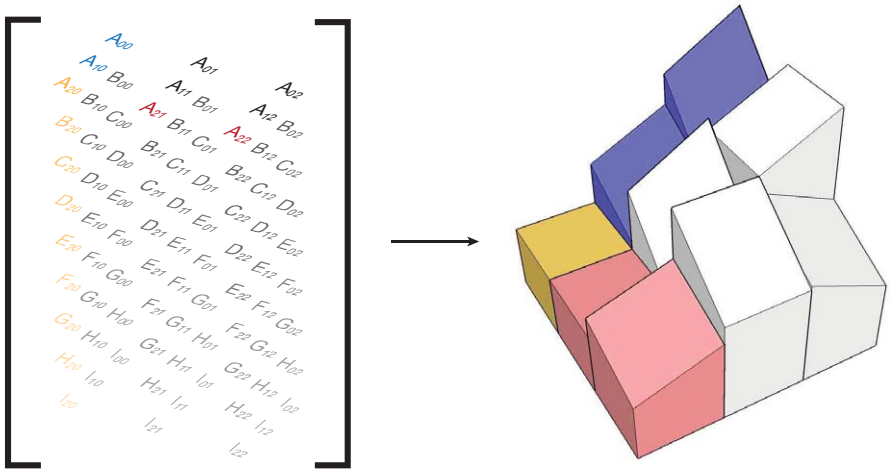

Figure 4.2.1 In translation from genotype to phenotype, an array column codes for a single house zone, while rows of the array map onto the rows of zones in the house.

the three-by-three grid contains a zone, or room, whose nine parameters determine its height, roof slope, material assignments, and affinity for openings to its neighbors.

The genotype that encodes the house's parameters is a 3 × 3 × 9 array of doubles. Each parameter is allowed to vary between 0 and 5/6 by increments of 1/6. Thus, six alleles are possible for each of the 81 genes that control the entire building. The first two dimensions of the array correspond to the sides of the nine-square grid, so that nine parameters are assigned to each zone.

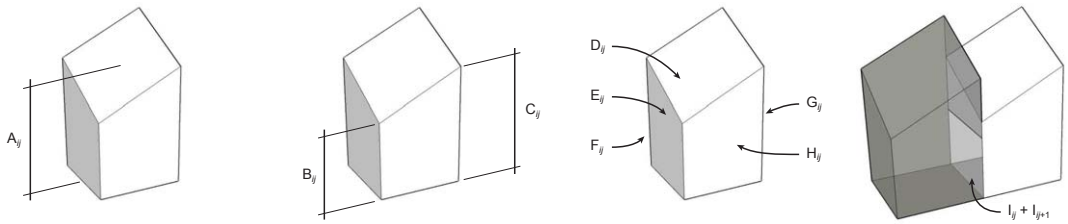

Figure 4.2.2 Each array column contains nine parameters that control its corresponding zone. These include three height parameters, five material parameters, and one parameter to control affinity for openings to other zones.

The first three parameters for each zone define its height and the slope of its roof. In early tests of the genetic algorithm, solutions tended to optimize for thermal performance by minimizing their volumes. Since the desired volume of a building is

generally known very early in design, it was decided to keep the total volume of each 15 m × 15 m house constant at 900 m^3. Hence, the height parameter actually controls the relative, not absolute, height of the zone. Under this schema, it is possible for a zone to have zero height, in which case Ecotect ignores it. In zones that do have volume, variation of the two roof slope parameters is limited such that no corner of a zone can be less than 2.5 m in height.

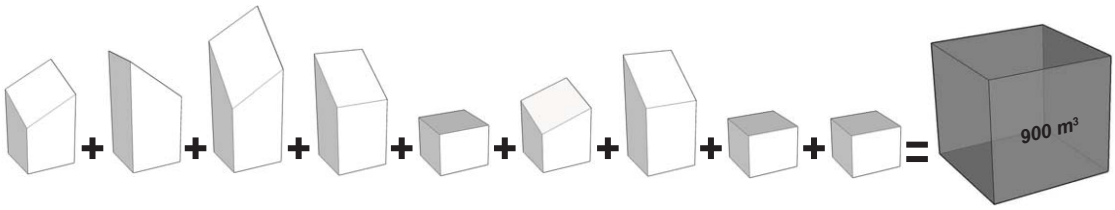

Figure 4.2.3 The total volume of the zones is kept constant in order to prevent the algorithm from minimizing the building's volume.

Materials for the roof and four walls are assigned by the next five parameters. Three window and three opaque materials are available which offer R-values between 0.17 and 3.3 m^2·K/W (1 and 20 ft^2·°F·h/Btu) and a range of transparencies and thermal lags similar to the range found in real building materials.

Material	Window			Opaque		
	0	1	2	3	4	5
U-Value (W/m^2·K)	6.0	3.3	1.8	1.0	0.55	0.30
Admittance (W/m^2·K)	6.0	3.3	1.8	1.0	0.55	0.30
Solar Heat Gain Coefficient	0.92	0.75	0.58			
Solar Absorption Coefficient				0.5	0.67	0.83
Transparency	0.92	0.75	0.58	0	0	0
Thermal Lag (hours)	0	0	0	3	4	5
Emissivity	0	0.17	0.33	0.90	0.90	0.90

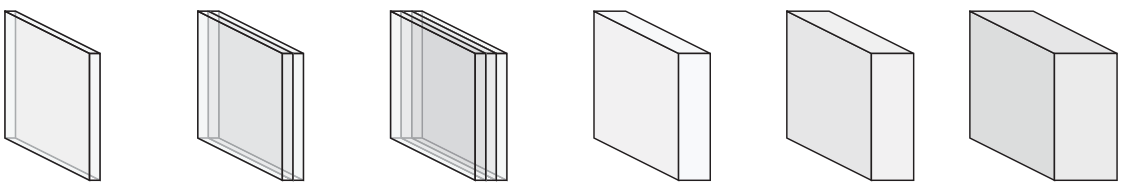

Figure 4.2.4 The six materials available to the algorithm reflect the properties of a range of building materials. Three represent windows, and three are opaque.

The final parameter gives each zone an affinity for openings to its neighbors. Ecotect simulates openings between zones using a material called a *void* whose thermal properties are similar to air. If the sum of two neighboring zones' last parameters is greater than one, a void surface is placed between them. A zone with a high affinity for openings tends to share void surfaces with most of its neighbors, but no zone can be forced to border a void. This method for placing voids allows genes that are intrinsic properties of zones to control traits in the phenotype that are not part of a single zone.

In all, there are 6^{81}, or about 1.07×10^{63} different genotypes under this schema, which code for a slightly smaller number of phenotypes. Since the thermal analysis of each individual takes about 90 seconds, it would take about 3.06×10^{57} years to analyze every possible variant on a typical dual-core PC. By comparison, the universe is only about 13.7 billion years old. At this rate, a brute force approach to finding the optimal solution for a given climate requires 2.23×10^{47} times the age of the universe to carry out. This is a rather long time to spend on the design of a single building.

4.3 Evaluation

Typically, thermal analysis is concerned with only one property of a building at a time. For instance, De Biswas's GA optimizes only for *insolation*, the amount of sunlight falling on surfaces. However, many factors contribute to the thermal comfort and energy draw of a building, so optimization of a single property is rarely sufficient in design. Early trials in this investigation sought only to minimize the house's heating, ventilation, and air-conditioning (HVAC) load. The typical result was a building with no windows or only one window, as these structures tend to be better insulated. Designs that provide interior daylight are encouraged using a form of multi-objective optimization that monitors both HVAC and electric lighting loads.

Ecotect uses data from a *.wea* weather file to model a climate. This data includes direct and diffuse solar radiation and temperature, as well as humidity and wind information. Ecotect is also able to calculate the *daylight factor*, or percentage of outside daylight available indoors, as an intrinsic property of a point in space. Thus, a schedule can be created for each zone on each day of the year to determine when natural light levels inside drop below a threshold and must be supplemented with electric lights. By varying the threshold and the lamp *efficacy*, or light output per unit of expended energy, the amount of energy required for lighting is adjusted to balance the need for windows with the need for thermal insulation. In this investigation, each zone requires an illumination of 538 lumens (50 footcandles) between 6 am and 10 pm, provided when necessary by conventional incandescent lamps with an efficacy of 13 lumens per watt. When electric lights are on, they also give off heat.

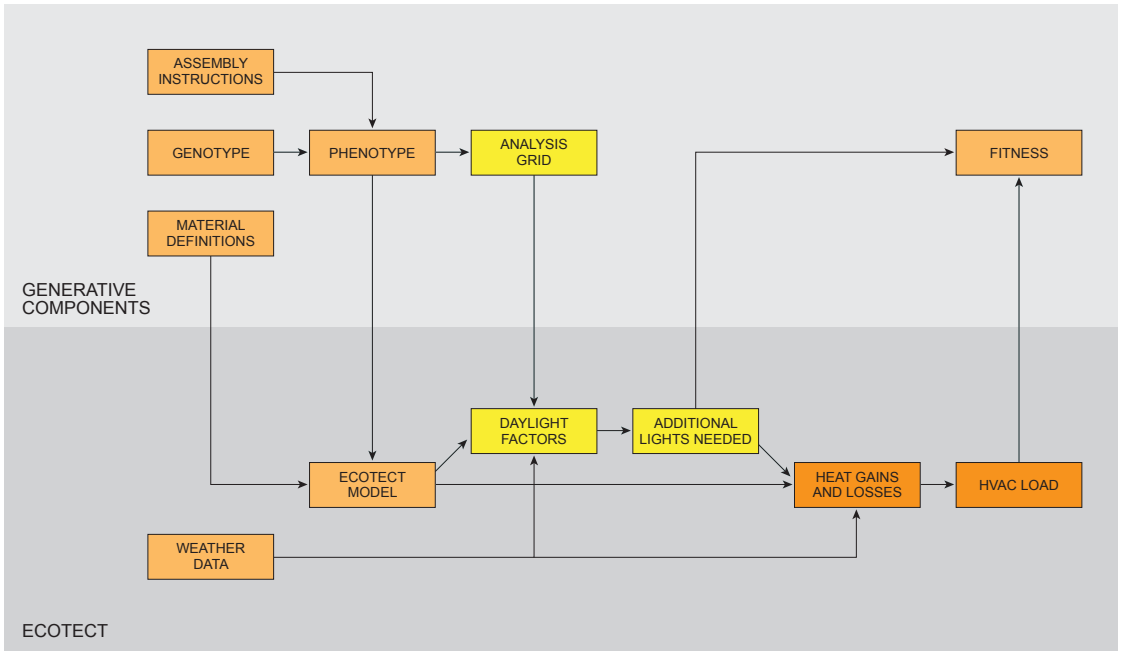

Figure 4.3.1 The process of creating and evaluating each individual is split between GenerativeComponents (GC) and Ecotect. GC provides Ecotect with a phenotype's geometry and material properties. Ecotect refers to its own library of climate data in order to perform its analysis, and returns the results to GC to use in selecting parents for the next generation.

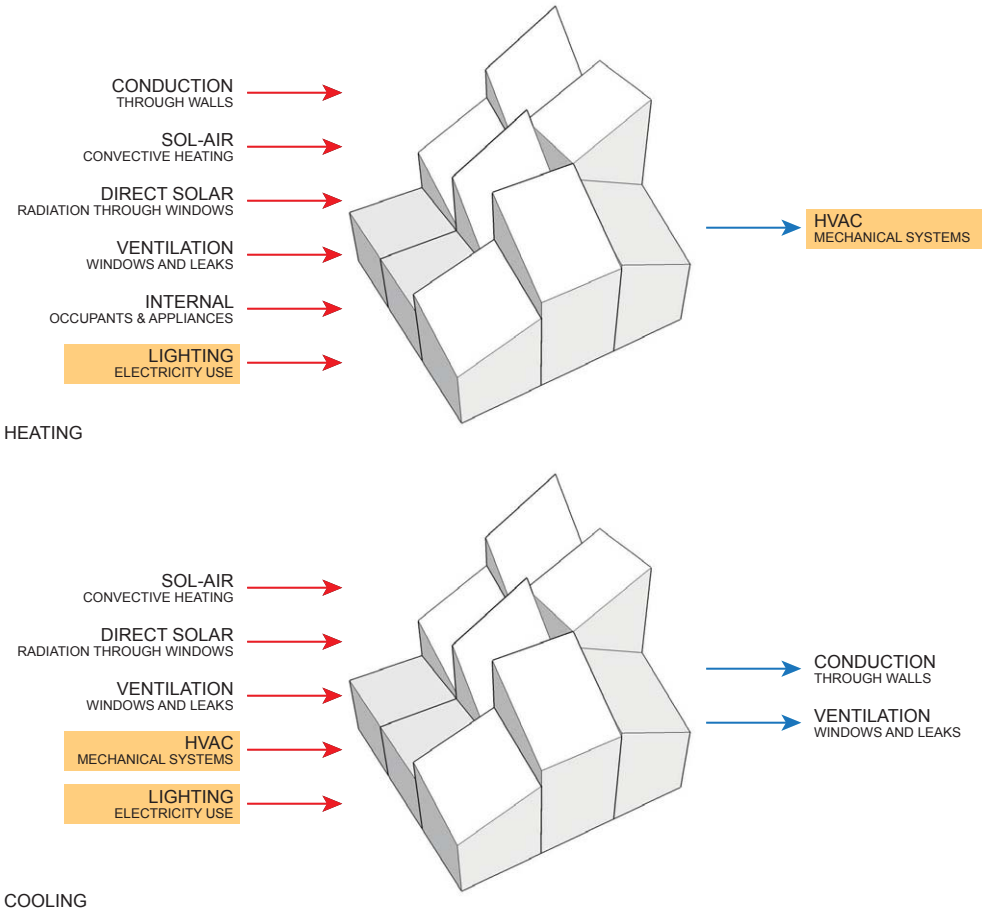

Figure 4.3.2 The admittance method used by Ecotect balances the amount of energy flowing into and out of each zone. Highlighted energy sources detract from fitness.

Ecotect performs thermal analysis using the Chartered Institution of Building Services Engineers (CIBSE) *admittance method*, which assumes the temperature in each zone to be constant at any given hour. Energy can enter or leave a zone by several methods, causing the temperature to change.

- Conduction: Energy can pass from one zone to another or to the outside through walls and ceilings. Material properties such as R-value and thermal lag affect how quickly energy flows through a surface. The difference in temperature between the two sides of the surface determines the amount of energy that crosses it.

- Sol-Air Gains: Convective heating of the air next to a surface by sunlight increases the amount of heat transmitted through it. Darker surfaces experience greater sol-air gains because they convert more light to heat.
- Direct Solar Gains: Radiant energy entering a zone through a transparent surface such as a window heats the zone from the inside. Solar heat gain is dependent on the window area and amount of available solar radiation.
- Ventilation: No building is airtight, so air carrying a certain amount of sensible and latent energy will enter or leave through cracks and windows. The amount of air moving through a zone depends on the zone's surface area and window area.
- Internal Gains: Each zone contains occupants, appliances, and electric lights, all of which give off heat.

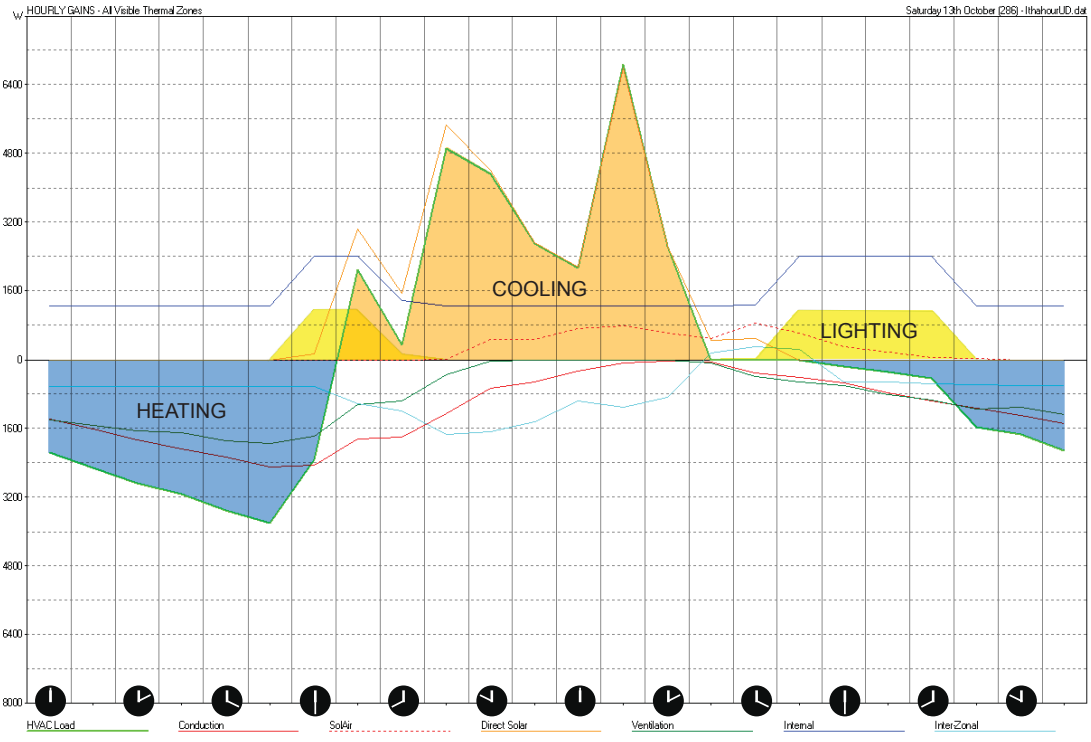

Figure 4.3.3 Ecotect performs calculations one hour at a time returning the energy required for heating (blue), cooling (orange), and lighting (yellow), which are functions of other energy flows into and out of the house.

Each house has a full air-conditioning system and a restrictive comfort band between 22.22 and 24.44 °C (72 and 76 °F). The HVAC system uses energy to control the temperature when one or more zones would otherwise be outside the comfort band.

It is standard practice to design the HVAC systems of buildings with the worst-case month in mind. Buildings in cold climates are designed for winter temperatures, and those in hot climates are designed for summer conditions. In this investigation, analysis is performed for each day of the controlling month and averaged to find the daily energy requirement during the worst-case month. The results are compared to other trials in which analysis is performed for the entire year.

Three climates are tested in this investigation. Anchorage, Alaska (61.1 °N), is chosen as a cold climate. In January, the controlling month, it receives only 5 hours of sunlight each day. Dubai, in the United Arab Emirates (25.2 °N), serves as a test case for hot climates. The controlling month for cooling is August, when the sun is almost directly overhead. These results are compared to Ithaca, New York (42.3 °N), which is in a temperate climate, but still one where January controls thermal design.

Figure 4.3.4 Three test locations represent a range of climates. Anchorage serves as a cold climate with a low sun angle. Ithaca has a temperate climate. Dubai is hot and occasionally receives sunlight from directly overhead.

4.4 Selection

For the architect using optimization algorithms to find energy-efficient forms, it is important that solutions be not only good, but also diverse. This investigation uses DC to maintain diversity in the population of solutions. A mu+lambda ($\mu+\lambda$) algorithm and a parallel hill-climber (PHC) algorithm are also considered as alternatives to DC.

In each generation, daylight and thermal analysis is carried out on ten individuals. This means that the population size in the DC and PHC algorithms is ten, and in the $\mu+\lambda$ algorithm, the number of parents (μ) and number of offspring (λ) are both set to ten. In $\mu+\lambda$, there is open competition between all parents and offspring for the μ spots in the next parent generation. In DC, the parent population is randomly divided into mating pairs, and each pair produces two offspring. An offspring replaces its closest parent if it outperforms that parent. Closest parents are determined in such a way that the sum of *Hamming distances*, the number of one-mutant steps, between parents and their respective offspring is minimized. The PHC algorithm is essentially the same as DC except that no crossover occurs, so that the ten individuals in each generation belong to separate lineages.

Each trial runs for fifty generations for a total of 500 calculations. This number of calculations takes about twelve to fourteen hours to perform. The number was chosen so that trials could be run in a public computer lab at times when few computers were in use, usually on Friday and Saturday nights.

4.5 Variation

The genotypes of the first generation are created at random. Hence, a large amount of variation is expected initially. Each offspring in later generations is created by applying one of five *variation operators* to members of its parent generation.

The first two operators are versions of mutation and crossover adapted for this schema, while the rest are unique to this schema.

- Mutation: A single gene of one parent's $3 \times 3 \times 9$ array genotype is overwritten with a random allele to create an offspring.
- Crossover: The genotypes of two parents are combined to create an offspring. In this case, a random plane described by the equation $ax + by + cz + d = 0$ cuts through the array, and genes to one side of the plane are taken from one parent while genes on the other side are taken from the other parent. It is simple to imagine this as phenotypic zones to one side of the plane coming from one parent and zones on the other side coming from the other parent. However, it is also possible for the plane to cut the array so that all zone heights

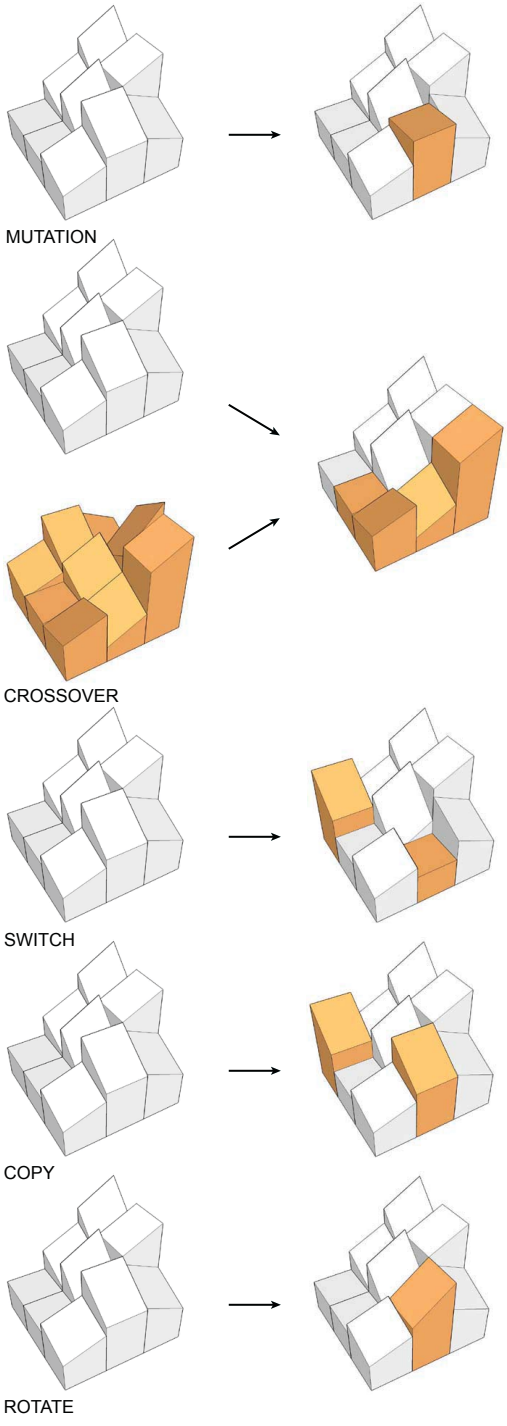

Figure 4.5.1 Point mutation and crossover are typical variation operators in genetic algorithms. The switch, copy, and rotate operators are specialized mutations unique to the house phenotype.

come from one parent and all material assignments come from the other. Many other divisions are possible, too.

- Switch: The parameters for two zones are switched. In the resulting offspring's phenotype, two zones appear in different locations than in the parent, while all other zones remain the same in both.
- Copy: The parameters for one zone replace those of another zone. Two identical zones appear in the resulting offspring's phenotype, while the same zone appears only once in the parent.
- Rotate: The parameters for one zone's roof slope and wall material assignments are reordered in such a way that the phenotypic zone in the offspring appears to have rotated 90° from its orientation in the parent. When this variation operator is called, it executes one, two, or three times, so that rotations of 90°, 180°, or 270° are equally likely.

Only one variation operator is used to create each offspring. Probabilities for the occurrence of each type of variation are as follows.

Variation Operator	Deterministic Crowding and Mu+Lambda	Parallel Hill-Climber
Mutation	10%	20%
Crossover	50%	0%
Switch	10%	20%
Copy	10%	20%
Rotate	20%	40%

The algorithms described in this chapter include translation, evaluation, selection, and variation procedures. Therefore, they should display evolutionary improvement when run. The next chapter describes the results of a series of trials of these algorithms.

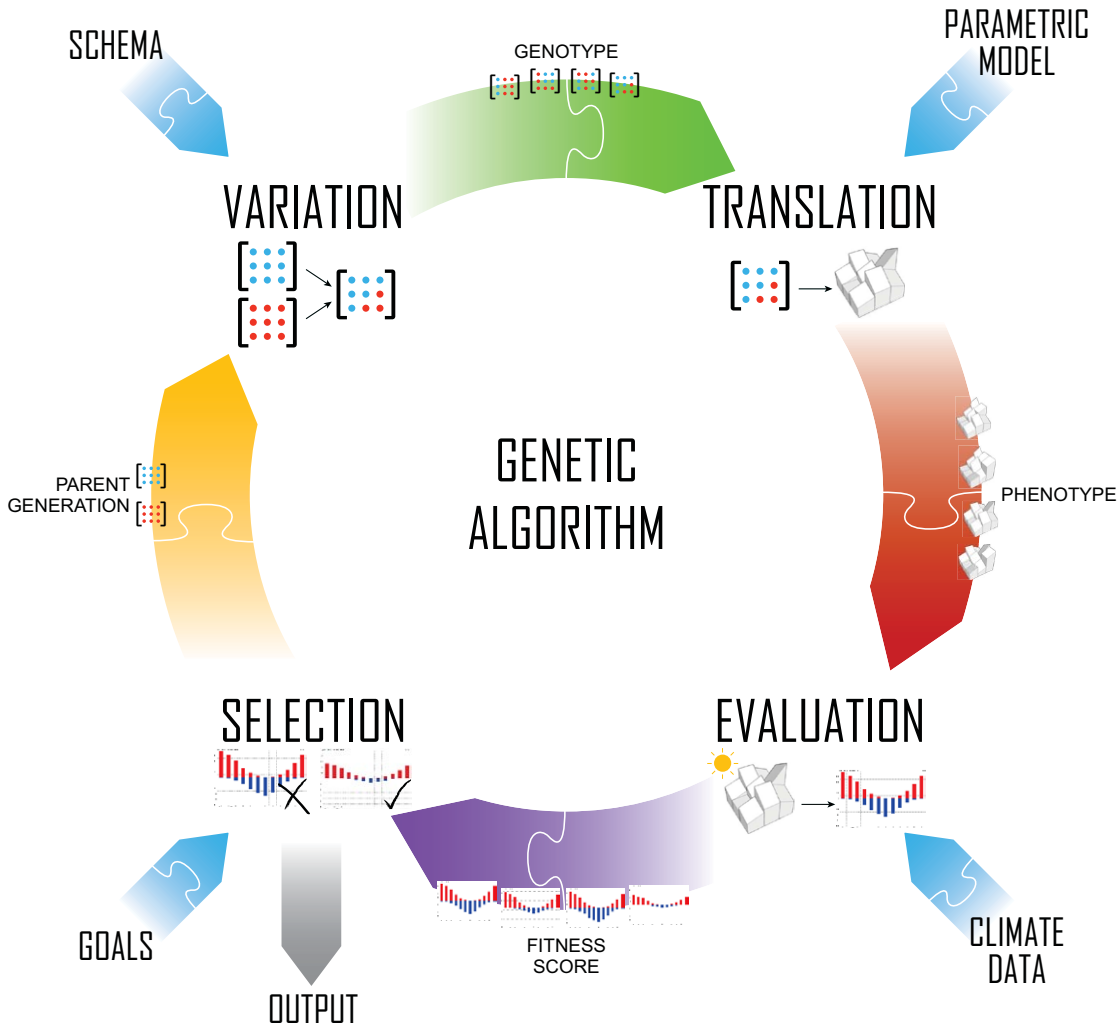

Figure 4.5.2 An algorithm containing well-chosen components should display evolutionary tendencies.

CHAPTER 5

RESULTS

> To seek not for ends but for antecedents is the way of the physicist, who finds "causes" in what he has learned to recognize as fundamental properties, or inseparable concomitants, or unchanging laws, of matter and of energy.
>
> Nevertheless, when philosophy bids us hearken and obey the lessons both of mechanical and of teleological interpretation, the precept is hard to follow: so that oftentimes it has come to pass, just as in [Roger] Bacon's day, that a leaning to the side of the final cause "hath intercepted the severe and diligent enquiry of all real and physical causes," and has brought it about that "the search of the physical cause hath been neglected and passed in silence."
>
> So long and so far as "fortuitous variation" and "survival of the fittest" remain engrained as fundamental and satisfactory hypotheses in the philosophy of biology, so long will these "satisfactory and specious causes" tend to stay "severe and diligent enquiry . . . to the great arrest and prejudice of future discovery."
>
> D'Arcy Thompson, 1917

5.1 Climate

In every run of the deterministic crowding (DC) algorithm, every one of the ten solutions found performs better than the initial randomly created population, and in some cases much better. This is true in each of the selected climates, although some are better suited to optimization than others. Since the majority of architectural

designs are arrived at without conducting any thermal analysis, it is reasonable to assume that an "average" building performs about as well as one of the randomly generated members of the starting population.

The effectiveness of the genetic algorithm varies depending on the climate. In Anchorage, where the January temperature is far outside of the comfort band, the most significant improvement is seen. In four trials, the average improvement in the best solution by the fiftieth generation is 60%. This is a remarkable improvement in energy savings; as a comparison, the U.S. Green Building Council offers a scale of award credits to new buildings achieving between 10.5% and 42% energy savings over typical construction.[40] The decrease in energy use in Dubai's hot climate averages 40%. In Ithaca, improvements by the fiftieth generation average 55%.

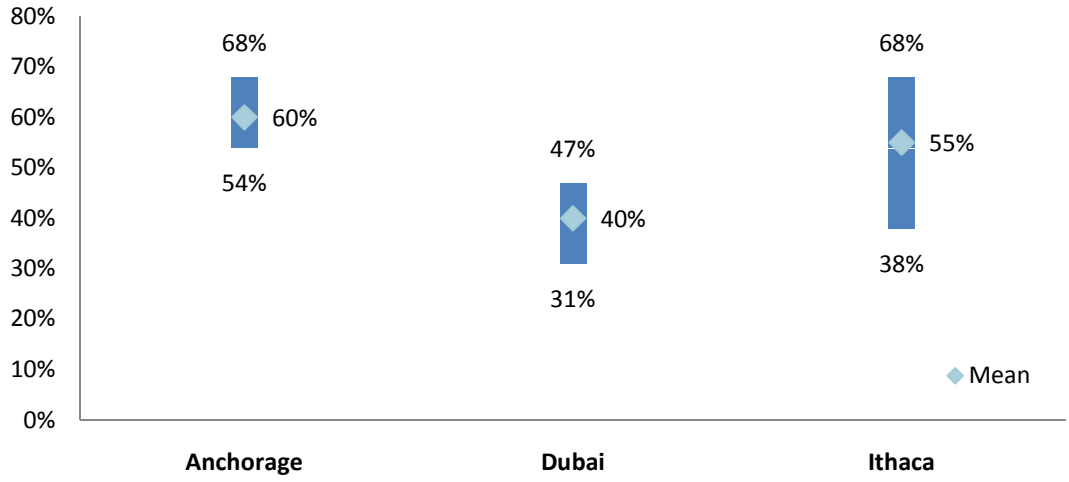

Figure 5.1.1 The improvement in the best individual of each trial over fifty generations falls within a certain range for each climate.

Several factors may explain the differences in level of improvement between these climates. The winter temperature in Anchorage is farther out of the comfort band than are the temperatures in the other climates, so initially chosen phenotypes are likely to perform far worse there, and the fitness landscape may have steeper

[40] *LEED*

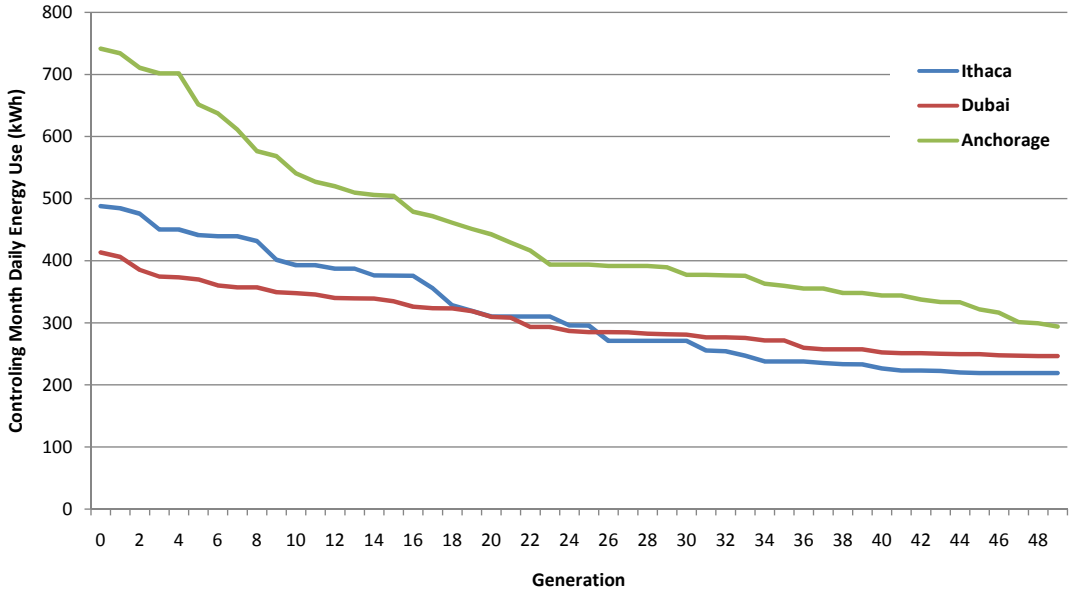

Figure 5.1.2 The mean of the best individual's fitness from each trial takes the shape of typical fitness curves. Trials in different climates improve at different speeds.

gradients. Some early experiments used climate data from Nairobi, which sits almost directly on the equator. Very little improvement was found in these trials, and further investigation uncovered that the temperature in Nairobi hovers around the comfort band year round. As a result, Nairobi's fitness landscape is quite flat, so little useful adaptation occurs and genetic drift is quite likely.

Another reason for the differences seen in the selected climates is the effect of considering summer rather than winter behavior. In Anchorage and Ithaca, where the need for heating governs design, the internal gains arising from electric lighting and occupants reduce the need for heating from mechanical systems. In Dubai, where cooling is critical to design, the same internal gains increase the need for energy to cool the building. The requirement for interior light levels imposed in this investigation means that each zone receives heating either by sunlight or electric lighting. In cold climates, this can sometimes reduce the total energy consumption, but in hot climates this always adds to it.

5.2 Algorithm Performance

Thermal and daylighting analyses alone are insufficient to make all the decisions that affect a building's form. Because the architect cannot expect to account for every important factor in the fitness criteria, it is important that the GA provide a variety of solutions. DC is used in this investigation because of its ability to maintain diversity in a population. However, other types of genetic algorithms might provide better solutions.

Trials in Ithaca's climate are repeated using a mu+lambda (μ+λ) algorithm and a parallel hill-climber (PHC). In three trials, the average improvement of the best individual in a PHC population over fifty generations is 53%, not far below the average of 55% for DC. The population at the end of each PHC run is also very diverse. However, while the entire final population of the DC algorithm outperforms the best individual in the first generation, many of the solutions generated by PHC are poor performers. Without the benefit of recombination, only the individuals that started out in good basins of attraction are likely to have successful offspring.

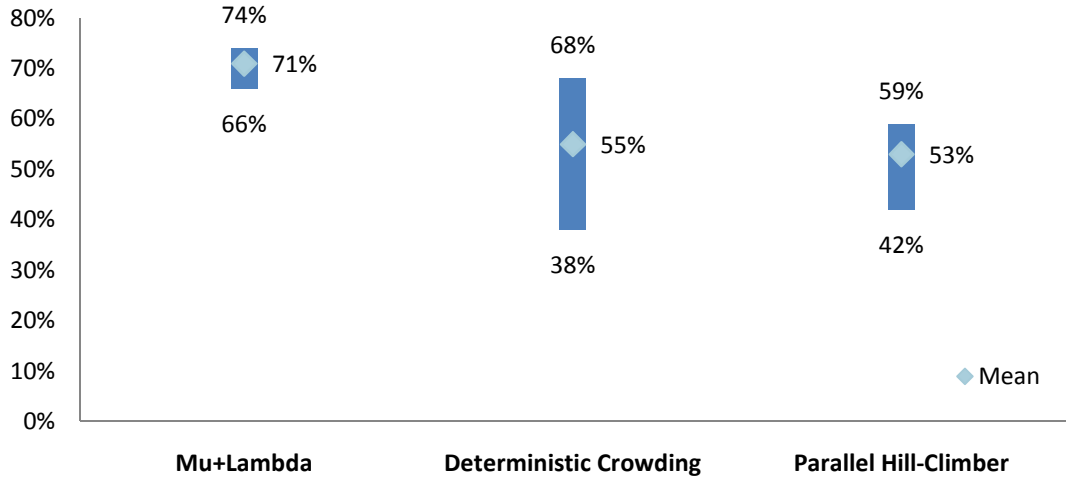

Figure 5.2.1 The improvement in the best individual of each trial over fifty generations varies according to the algorithm used. Some algorithms produce uniform levels of improvement because they climb peaks faster. Algorithms that test more of the search space may not improve as much.

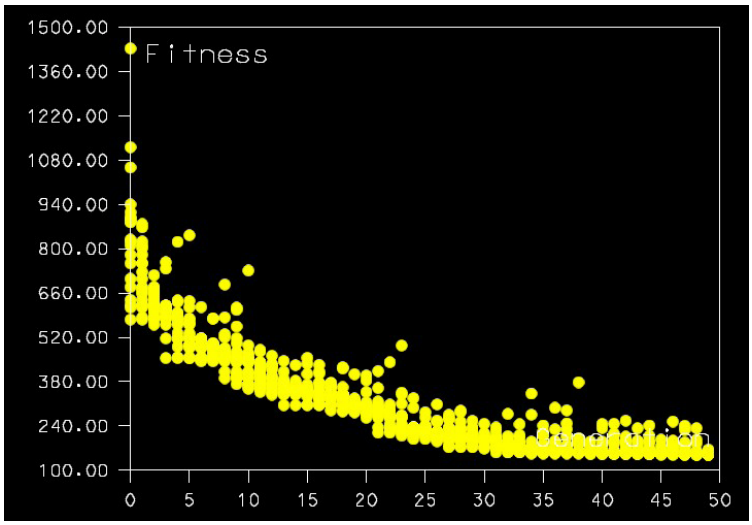

Figure 5.2.2 The mu+lambda algorithm initially experiences dramatic improvement that levels off before the end of the trial. The loss of diversity is evident as the range of fitness scores in each generation is quite narrow.

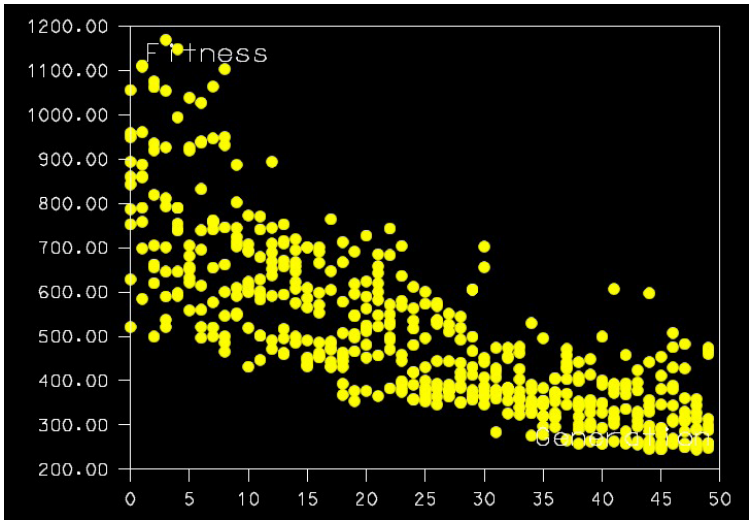

Figure 5.2.3 In a typical run of the deterministic crowding algorithm over fifty generations, the best individual improves by over 50% and all of the final generation outperforms the first.

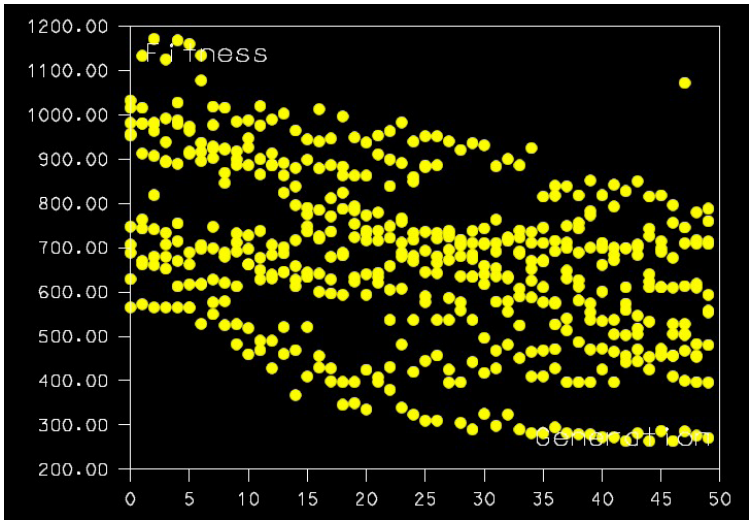

Figure 5.2.4 In the parallel hill-climber algorithm, a few individuals produce offspring as successful as those in some deterministic crowding trials. However, some of the solutions found in the final generation do not outperform the initial best.

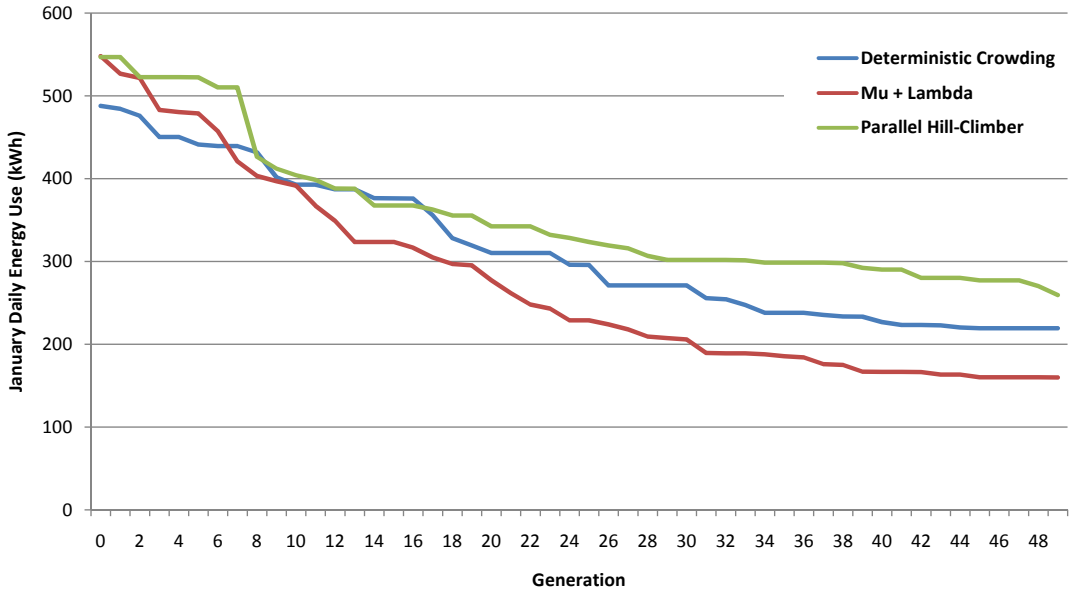

Figure 5.2.5 The mean of the best individual's fitness from each trial improves at a different rate depending on the algorithm used.

In contrast, μ+λ performs quite a bit better. In three trials, the best individual improves an average of 71% over fifty generations. Unfortunately, a great loss of diversity accompanies this large improvement. The solutions found in each μ+λ trial are quite similar in appearance and performance. As a result, the architect must run far more trials in order to accumulate a diverse set of design options.

5.3 Analysis Period

Conventional wisdom in architecture holds that a building should be designed for the month in which outside temperatures are farthest away from the comfort band. In keeping with this convention, the trials described so far use only climate data from the controlling month for the evaluation of each individual. Three additional trials use Ithaca climate data from the entire year to check the validity of this method. To save computational time, the climate data from the year is sampled every fifth day instead of every day.

Buildings optimized for data from the entire year are expected to perform poorly in the controlling month. Logically, the controlling month will have less of an effect on the building's shape, and selective pressures will favor houses that perform better in other months. For instance, a house in a cold climate might develop adaptations suited to cooling in the summer months that cause the house to lose heat in the winter. Fitness values and improvement levels from these trials cannot be compared to previous results because optimizing for the entire year is a different fitness criterion. Instead, the solutions must be directly compared in terms of each month's thermal performance.

The result of this comparison is startling. When the monthly heating and cooling loads of the best solutions are averaged within each set of trials, houses optimized for the full year outperform those optimized for the controlling month. Houses optimized only for the governing month perform worse in every month – even in the month that controls. Including the second and third best solutions in the averages to increase the sample size does not change the trend.

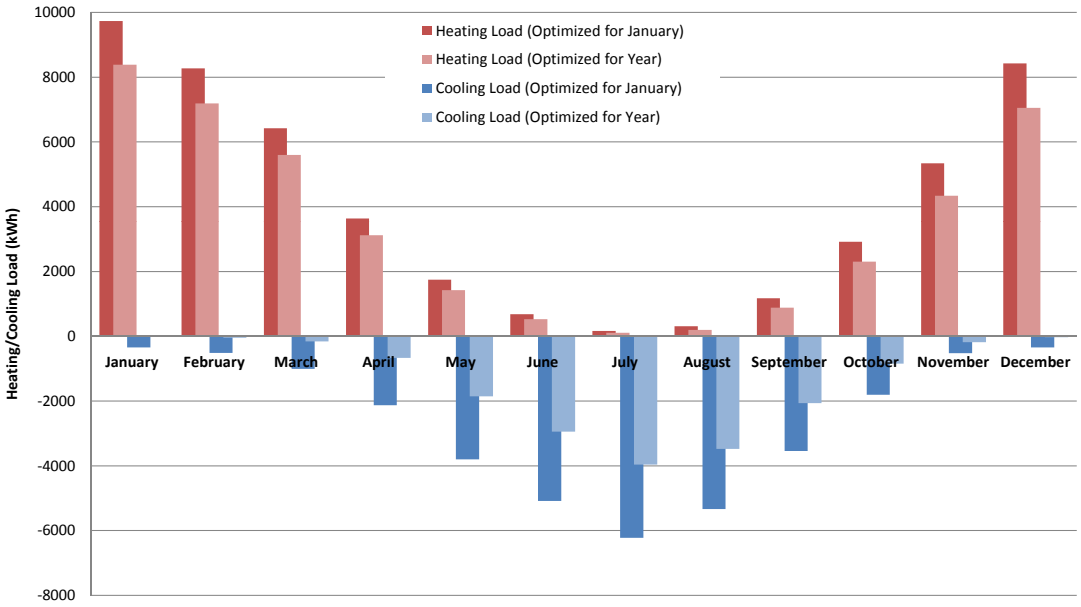

Figure 5.3.1 Mean monthly heating and cooling loads in Ithaca are consistently smaller in houses optimized for the entire year's climate data.

The evaluation mechanism may be to blame for this defiance of conventional wisdom. Due to the heuristic nature of Ecotect's calculations and the fact that different data is available to Ecotect after is has performed a full year's worth of analysis, the information obtained in the two sets of trials may not be strictly comparable. Also, Ecotect's analysis of monthly heating and cooling loads does not take into account energy used to turn on lights, which also contributes to the fitness measurement. Hence, incompatibility of the compared data cannot be ruled out.

On the other hand, there may be an advantage to designing the house for the full year's climate data. Co-evolution by varying the fitness criteria helps evolving populations avoid low fitness peaks. The broader range of phenotypes that appear fit under the full year's data may encourage diversity. A diverse population can explore more of the fitness landscape and is more likely to discover allele combinations that perform well in all weather. Adaptations that mainly benefit the house in non-controlling months but offer some advantage in the controlling month can accumulate in the population and eventually lead to a very fit phenotype.

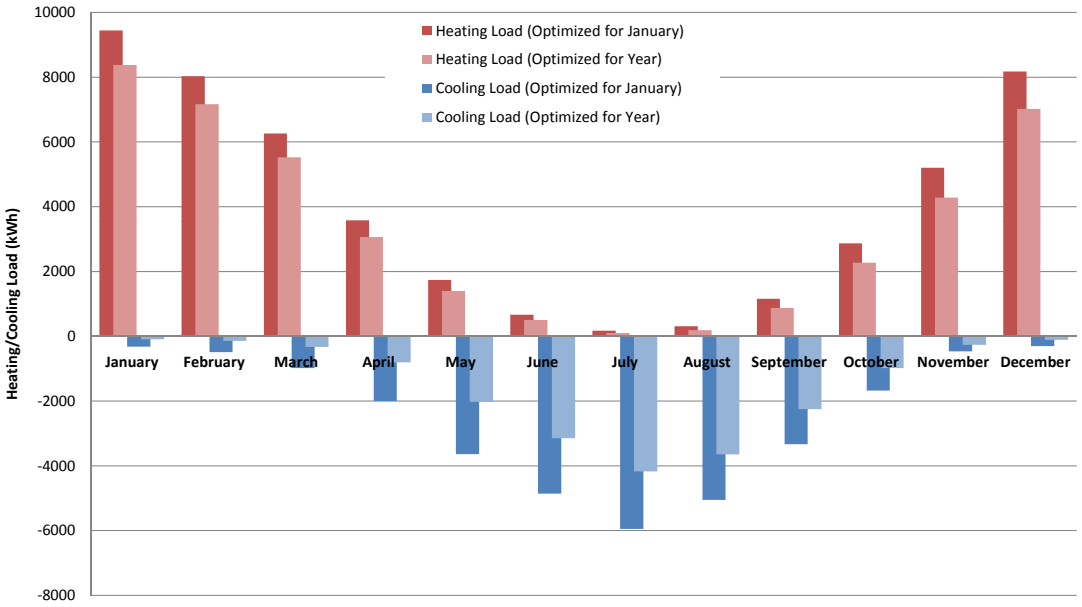

Figure 5.3.2 A larger sample size used to calculate the mean monthly heating and cooling loads in Ithaca confirms that houses optimized for the full year perform better.

The diversity of solutions found by the GA gives architects a lot of fit starting points for developing the house design. The next chapter examines the diversity of several trials and the effect it has on design options. Some particularly creative design solutions from the GA are taken a step farther into schematic design.

CHAPTER 6

DESIGN

> The artist's present plight is a sad one, but may he truthfully say that society is less well off because Architecture, or even Art, as it were, is dead, and printing, or the Machine, lives?
>
> Is it not more likely that the medium of artistic expression itself has broadened and changed until a new definition and new direction must be given the art activity of the future, and that the Machine has finally made for the artist, whether he will yet own it or not, a splendid distinction between the Art of old and the Art to come?
>
> Frank Lloyd Wright, 1901

6.1 Process

Each solution found by the genetic algorithm (GA) presents the architect with a lot of information. Heights of zones suggest locations where a second story can be added. Material assignments affect the amount of natural light entering a zone and the amount of heat it can store during the day to release at night. This information, along with the locations of openings that allow a room to span multiple zones, makes certain zones better suited to different functions. Changing a height, material assignment, or opening location alters the fitness of a design, so the algorithm's recommendations to the architect are quite specific. However, each time the algorithm is run, the architect receives ten new solutions. After a few runs of the algorithm, the architect has many fit design options to choose from.

The diversity of fit solutions found in a single run of the algorithm may not be immediately apparent, since the eye notices overall shape more quickly than it

Figure 6.1.1 Floor plans at 1’ = 1/32” scale based on solutions from a trial in Anchorage show significant diversity despite sharing the same three footprints. Dark shading indicates space for a second floor above what is shown.

recognizes material differences. However, since most of the typical rooms in a house have unique size and lighting needs, two building envelopes with the same overall shape may be better suited to quite different interiors. To demonstrate the diversity within a single run of the deterministic crowding (DC) algorithm, solutions from one run in each climate are developed as floor plans. Each floor plan represents a three-bedroom house with foyer, living room, dining room, kitchen, and garage. Each zone

Figure 6.1.2 Floor plans at 1’ = 1/32” scale based on solutions from a trial in Ithaca are varied but include patterns revealing their close genetic ties. Dark shading indicates space for a second floor above what is shown.

is assigned a particular room type based on its traits. For instance, the living room has first priority for any zone with openings. The master bedroom tends to be placed in a second story space when one is available. The garage is typically placed in a zone with a high U-value, since it is unlikely to be well insulated, but the dining room also has preference for high U-value zones with southern exposure in order to receive

Figure 6.1.3 Floor plans at 1’ = 1/32” scale based on solutions from a trial in Dubai are diverse variations on a few themes. Dark shading indicates space for a second floor above what is shown.

natural light. Once each room type’s needs are met, other decisions about the layout of the house respect the adjacencies and circulation patterns of typical residences.

No architect should be content to proceed with a design simply because an algorithm finds it fit. It is important to understand why a particular design performs well under certain evaluation criteria. This information also lets the architect know what parts of a design can be modified and which are necessary to the design’s

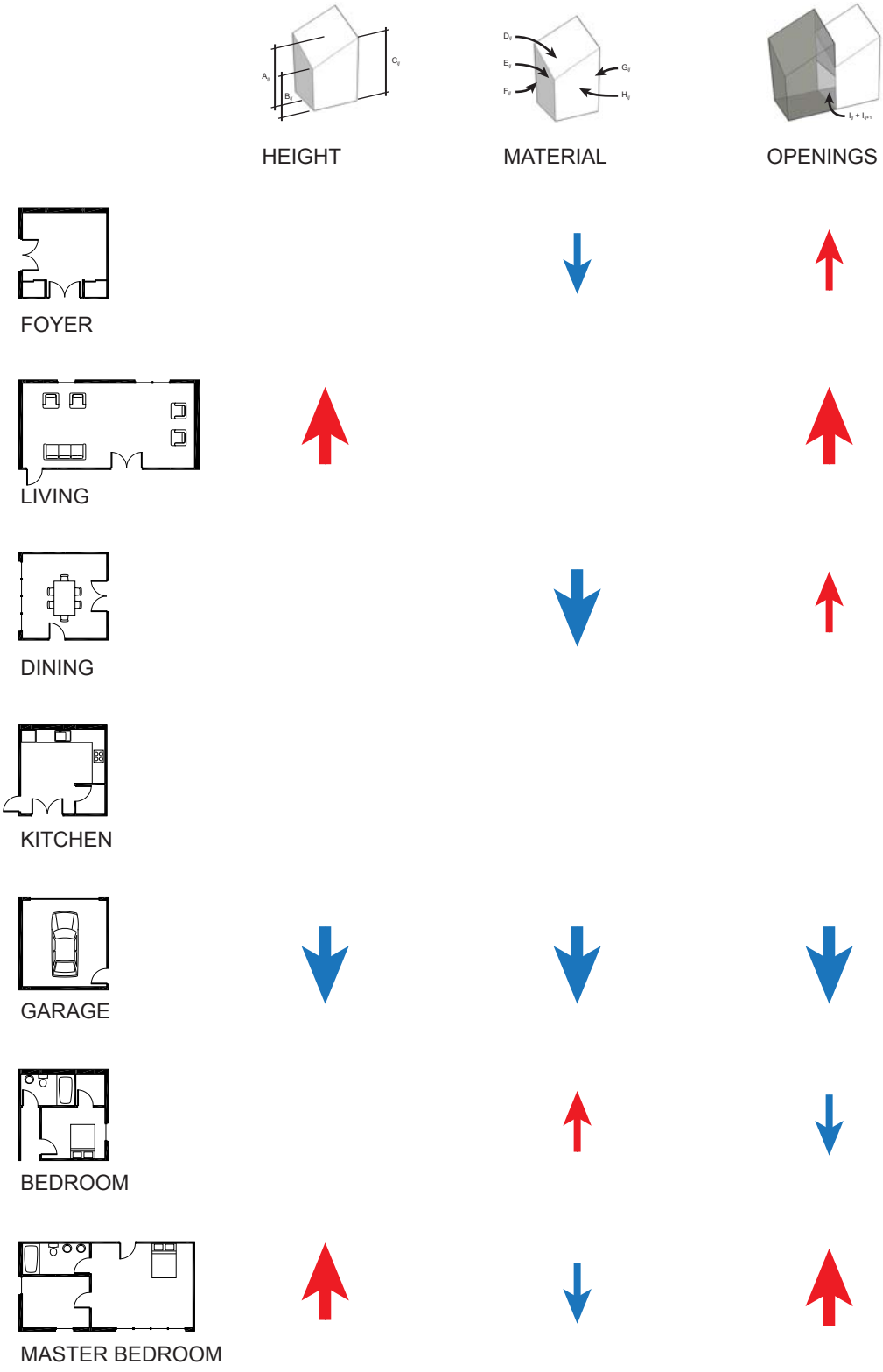

Figure 6.1.4 Each type of room is better suited to zones with certain genetic traits. Upward arrows indicate that the room type prefers a higher value for that allele category, and larger arrows indicate a stronger preference.

function. A few solutions discovered by the GA are notable for their display of *machine creativity*, adaptations that seem particularly well thought-out even though no human intelligence is behind them. As proof of the creative potential of GAs, one solution from each climate is elaborated into a schematic design. The chosen solutions are not necessarily the best from their climate or from their run, but they perform significantly better than the "average" house because of clever adaptations.

6.2 Anchorage

One solution in Anchorage's climate includes a south-facing window that provides a view in to the center-south, center, and center-north zones, all of which are connected by openings. This adaptation allows the sun, whose angle does not change much in the sky in January, to heat the floor of one third of the house for the few hours it is up. This heat is released back into the house at night. The height of the window is such that even in the sun's highest position in January, sunlight reaches the base of the house's north wall.

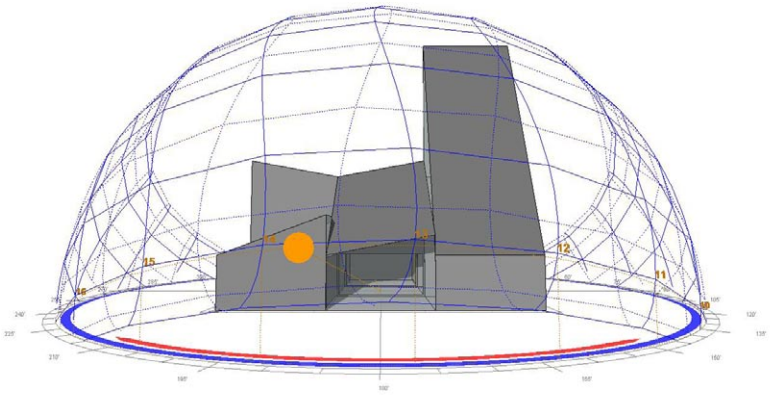

Figure 6.2.1 A unique adaptation for extreme northern climates uses sunlight to heat the entire depth of the house. This heat can be absorbed by the building's structure and released at night.

The walls of the Anchorage house are well suited to a cold climate. Since the exterior walls on the west side have a high thermal lag value, they absorb heat from the setting sun and release that heat into the building at night. The roof is also made of this highly insulative material. The material of the east wall is slightly thinner; it still

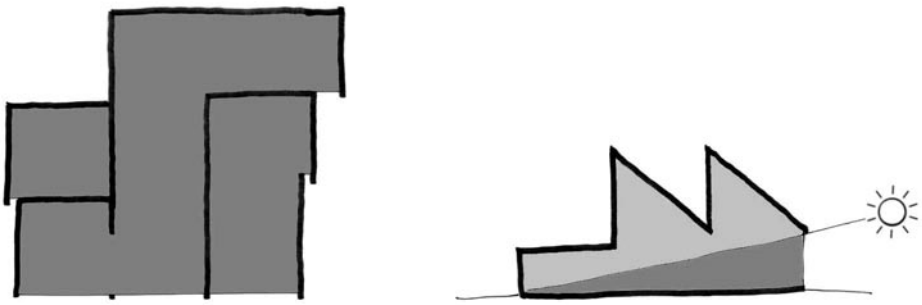

Figure 6.2.2 The plan of the Anchorage house is a series of tubes open to the south. These dimensions of the central tube let light and heat reach the back of the house.

provides reasonable insulation, but has a lower thermal lag that allows the rising sun to heat the interior more quickly. By comparison, the south walls are relatively thin. These walls receive the most direct solar exposure and therefore let in the most heat. At the same time, the area of these walls is small to minimize heat loss at night.

The schematic design for the Anchorage house emphasizes the placement of materials in the GA's design. Each zone or group of open zones is treated as a tube wrapped by a thick envelope above and on the east and west sides. The south end

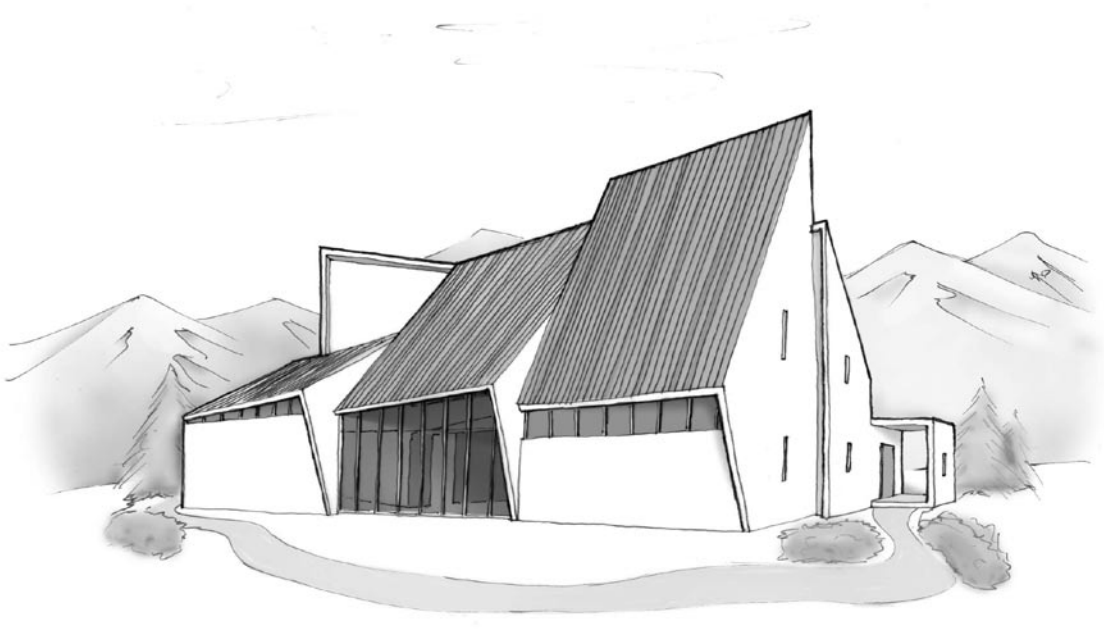

Figure 6.2.3 The shape of the Anchorage house blends with Anchorage's landscape.

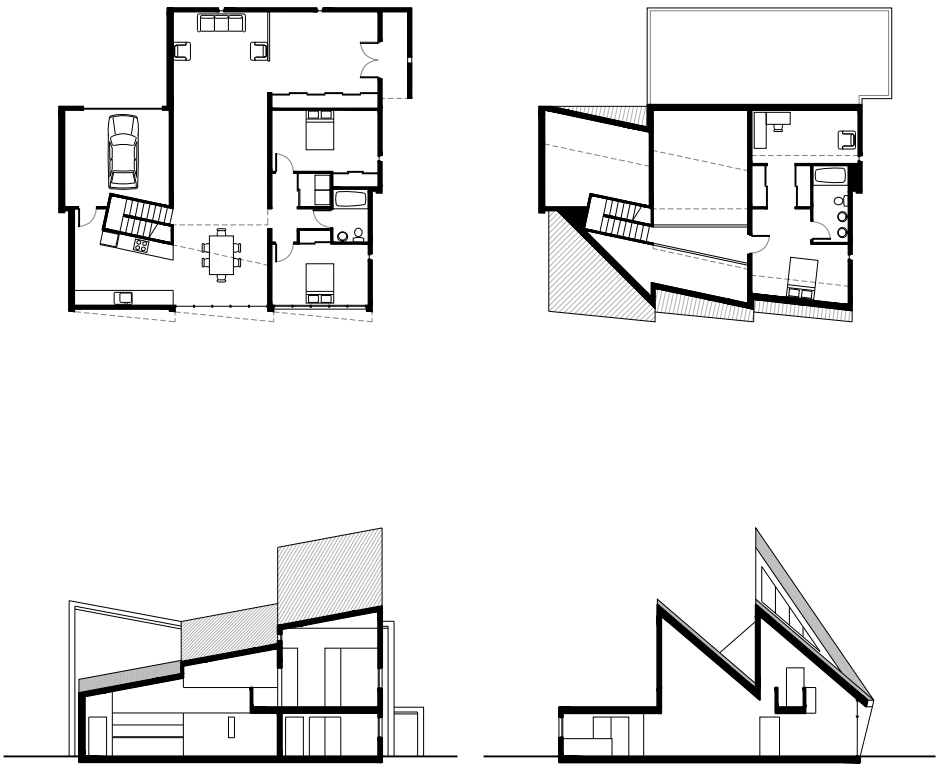

Figure 6.2.4 The Anchorage house in plan and section at 1’ = 1/32” scale.

of each tube is relatively thin with some glazed area, while openings in the other sides are thin punched slivers. The entrance at the northeast corner of the building is incorporated into one of the wrappings. Because of the large number of zones with openings, the living room, foyer, dining area, and kitchen all connect. A second story passage from the stairs to the master bedroom suite also opens into this space; its shape reflects the angle of the roof it parallels. The overall massing of the building, a product of the GA with minimal alteration, is treated to resemble the mountains that make up Anchorage’s backdrop.

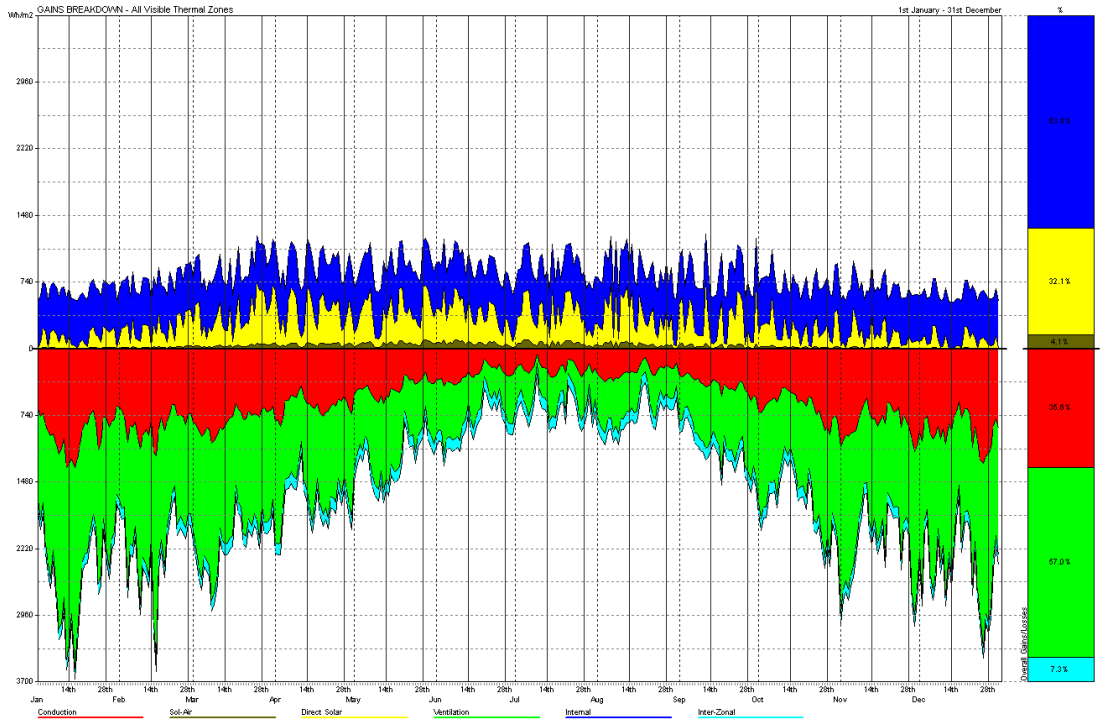

Figure 6.2.5 Over the course of one year, the Anchorage house gains heat from occupants (blue) and solar gains through windows (yellow). It loses heat through air infiltration (green) and conduction through walls (red).

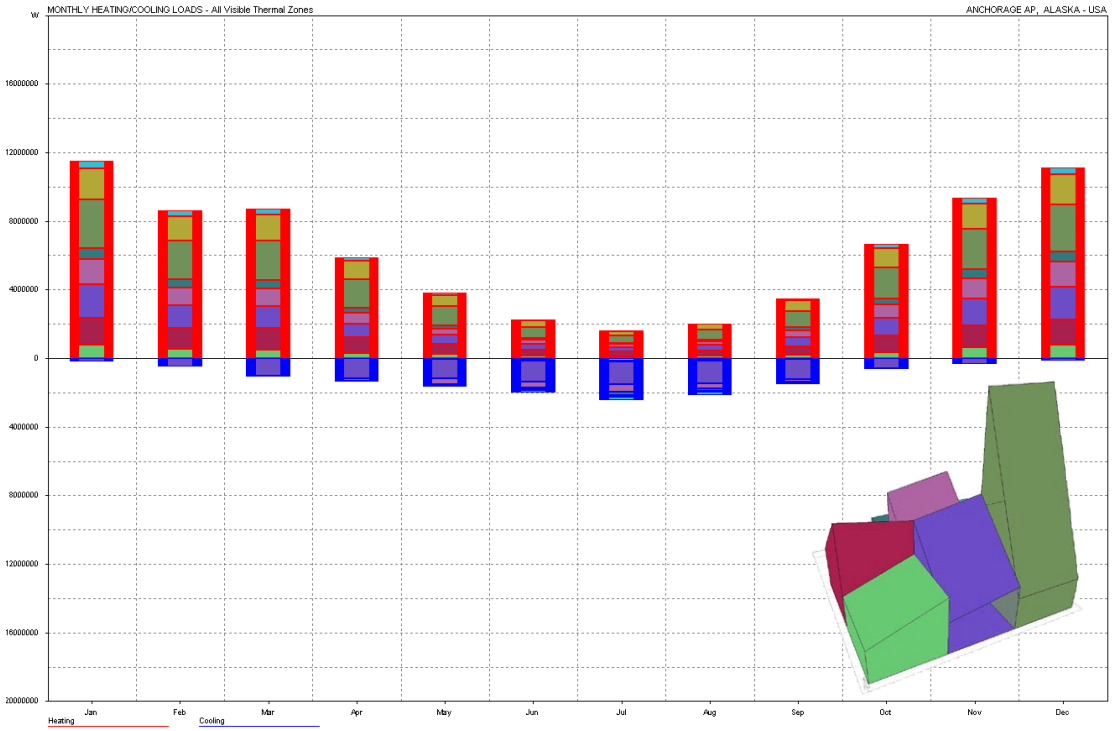

Figure 6.2.6 Additional heating (red) and cooling (blue) keep the Anchorage house comfortable. The south-facing window provides heat even in cold months.

6.3 Ithaca

The most successful individual from one trial in Ithaca's climate has a roof slope that perfectly matches the sun's angle at noon on January 30th. This angle maximizes the amount of sunlight hitting the relatively thin southern walls. Copy and rotate operators duplicated zones with this slope onto five of the nine grid squares in a way that minimizes the surface area exposed to the north, east, and west. The arrangement allows the interior of the building to heat quickly in the day and retain heat at night.

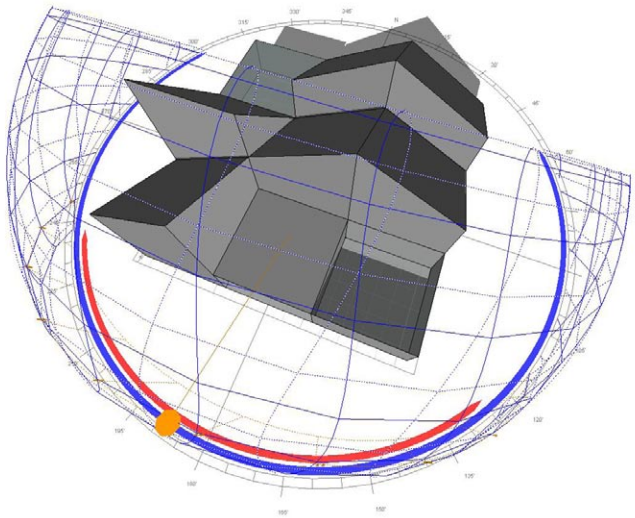

Figure 6.3.1 The matching roof slope on five zones is angled to maximize the amount of sunlight falling on small vertical surfaces. By placing thinner surfaces in these locations, the house can trap heat on winter days when the sun is low.

The interior of the Ithaca house contains an unusually large number of zones connected by openings. These six zones, mostly identical except for rotation, are all tall enough to contain a second story. The remaining three zones are very short, flat, and have little insulation. The strong diagonals of the first group set it apart visually from the low profile of the second. These two groups of zones are further differentiated in the schematic elaboration of the design. Inside the larger zones, diagonal lines regulate elements such as the stairs and second floor balcony. The balcony is wide enough to block sightlines into the master bedroom suite so that it, too, can be open to the rest of the house's large central volume. Only the garage and

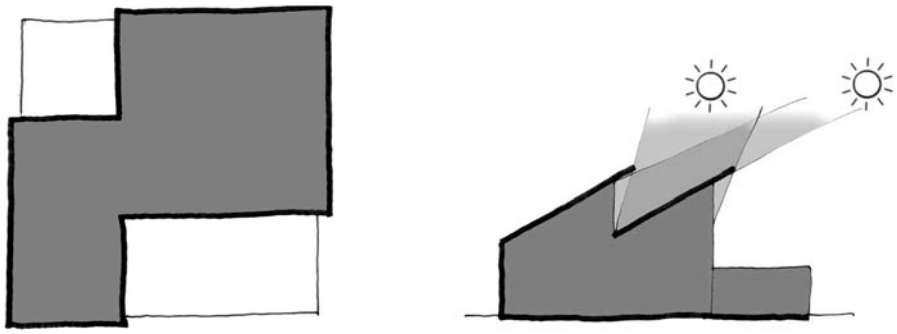

Figure 6.3.2 The plan of the Ithaca house contains a large central open space separated from smaller subsidiary zones. In section, the open space is designed to receive heat from a low winter sun.

two first floor bedrooms, which absolutely must be separated from the rest of the house by doors, are placed in the shorter, flatter zones.

The treatment of the exterior is in keeping with the house's thermal adaptations. The short zones, which cannot reasonably contain the large glazed walls and roofs given to them by the GA, are darker in color than the rest of the house in order to absorb heat by a different method. A clerestory band of windows takes the

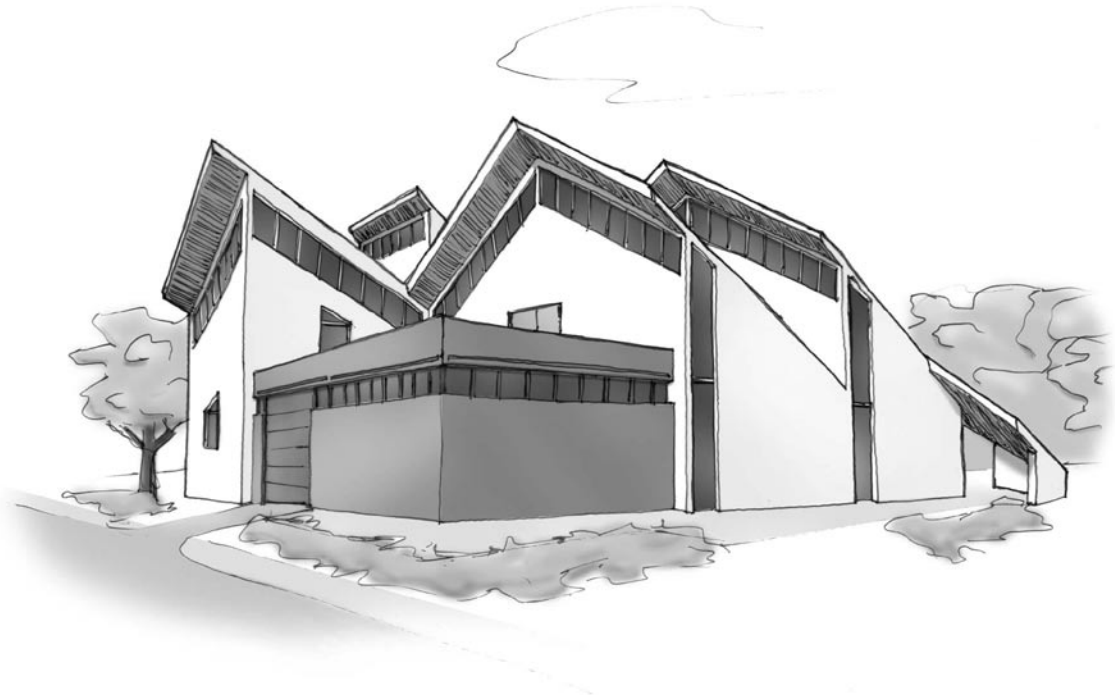

Figure 6.3.3 The façade of the Ithaca house draws attention to two groups of zones.

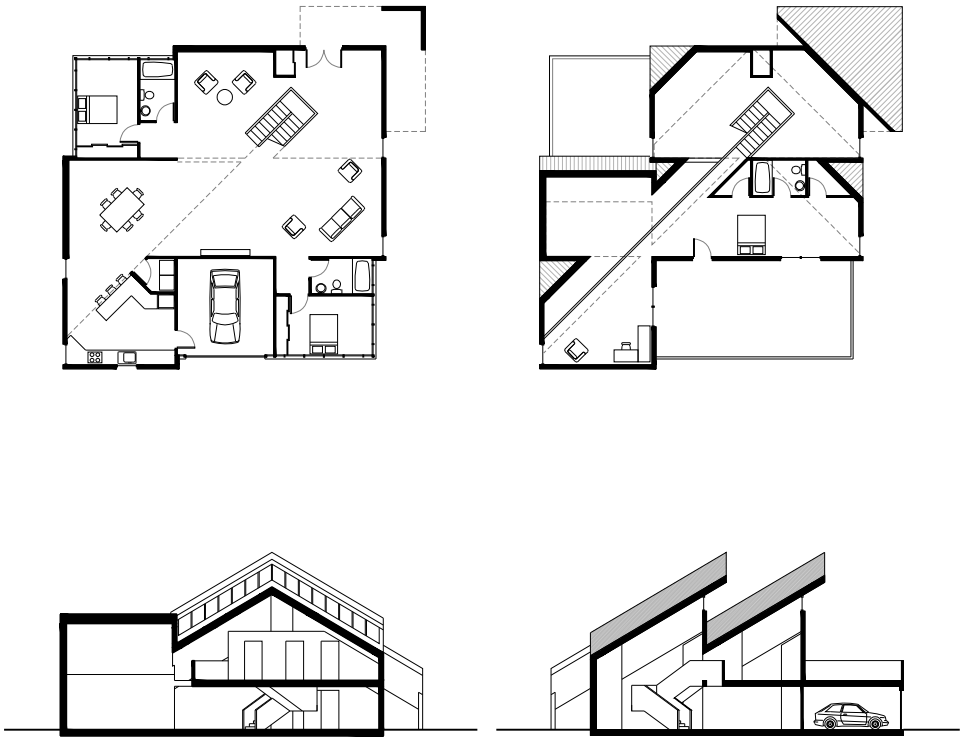

Figure 6.3.4 The Ithaca house in plan and section at 1’ = 1/32” scale.

place of skylights for the two bedrooms. The clerestory band also shows up on the south faces of the taller zones. Roof projections over these windows let light enter the house in the winter, but keep heat out during the summer when the sun is higher in the sky. These projections also reinforce the house’s adaptation to absorb heat from the low winter sun. A parapet around the roof of the low zones, which serves as a balcony to the master bedroom suite, is similar in scale to the projecting roof. The balcony itself takes advantage of the flat surface provided by the GA to add a new function to the house; it offers a benefit that could not have been foreseen when the algorithm was written.

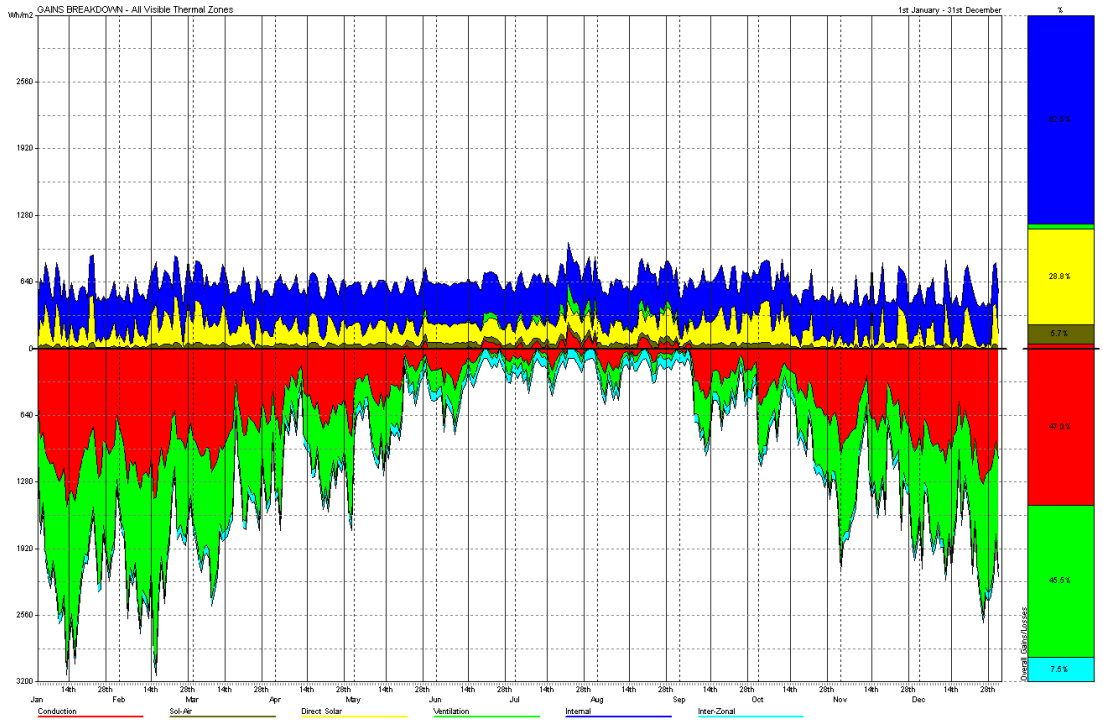

Figure 6.3.5 Like the Anchorage house, the Ithaca house experiences heat loss through infiltration (green) and conduction (red), but these losses drop off virtually to zero in the summer months.

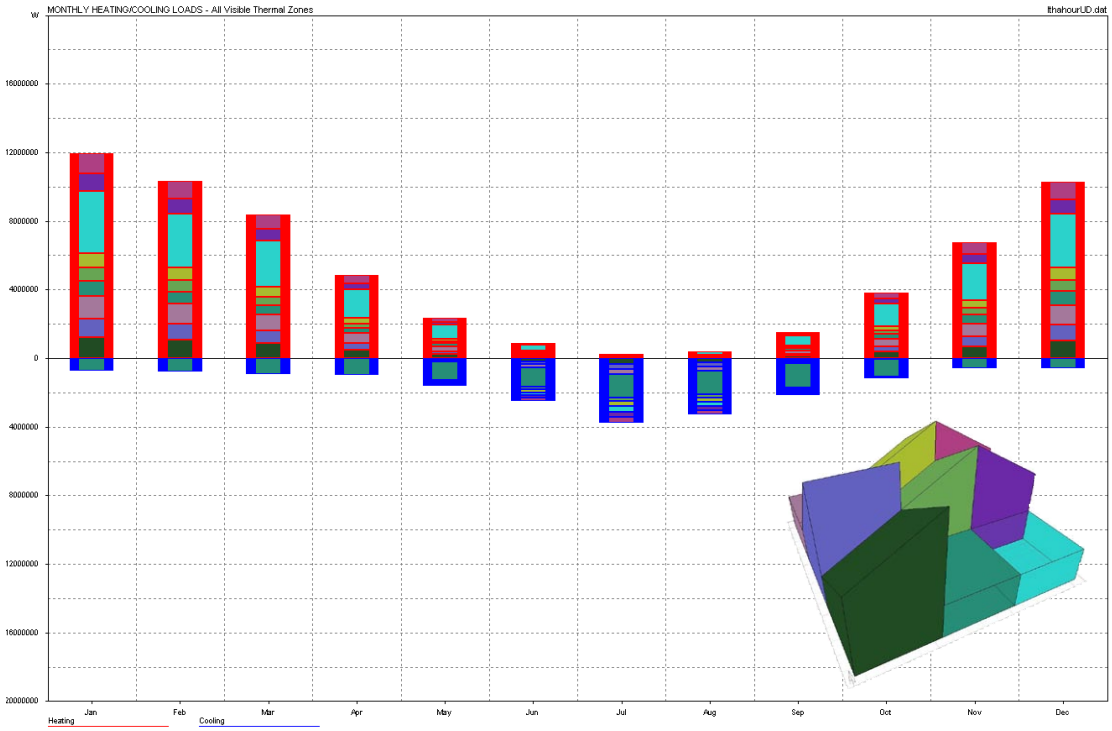

Figure 6.3.6 The GA's design for the Ithaca house suffers from the poor performance of one zone with a glazed roof. The altered schematic design eliminates this problem.

6.4 Dubai

One trial in Dubai produced several individuals with very tall zones on their east sides. While these towers perform poorly on their own, they have the effect of shading the rest of the house through most of the morning, which greatly reduces the amount of heat entering the other zones. Often, several zones in these houses have a height of zero, causing the towers to become even taller to keep the total building volume constant.

In one case, the shaded zones developed several glass sides, almost eliminating their need for electric lighting. The zone receiving the most shade even developed a glazed roof. This is particularly surprising in Dubai, since that zone effectively becomes a greenhouse. Only because the eastern tower keeps the glass roof in shadow until 10:30 each morning does the greenhouse not require an exorbitant amount of energy for cooling.

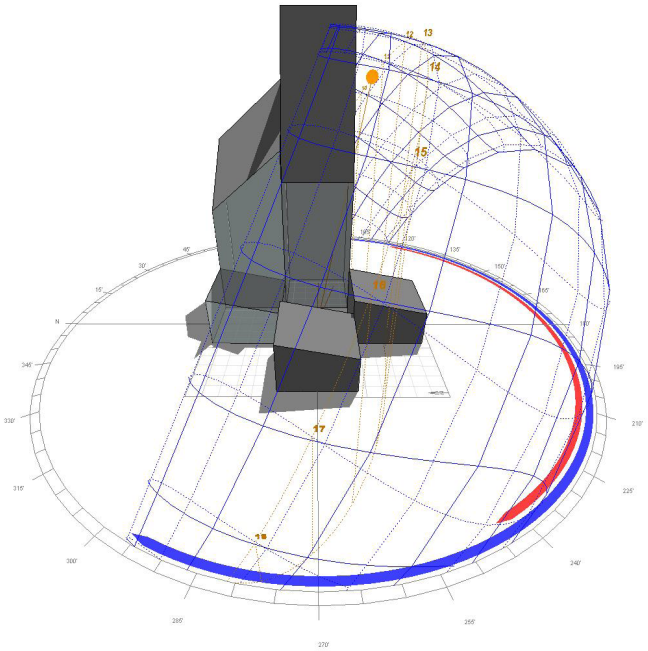

Figure 6.4.1 Towers on the east side (the rear in this view) shade the shorter zones in the morning. The small northern zone has two glazed walls and a glazed roof that could easily overheat if this zone were not completely shaded until 10:30 AM each day.

The schematic design iteration of the Dubai house emphasizes the pinwheel configuration of the zones. Since four of the zones have no volume, most of the rooms

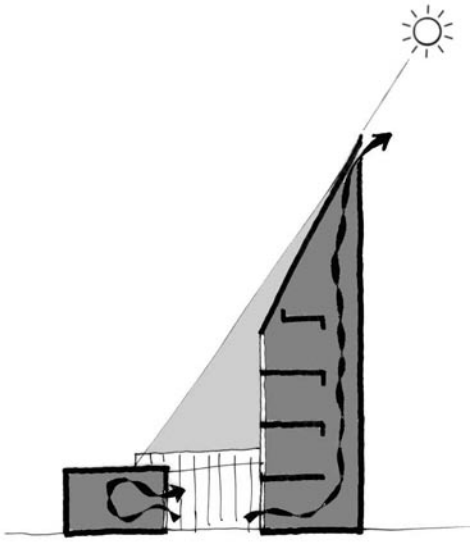

Figure 6.4.2 The Dubai house's tower shades the neighboring greenhouse in the morning. Warm air is drawn out of the greenhouse and courtyard through the tower by the stack effect.

are pushed off into the two tall zones on the building's east edge. The five-story height of this tower easily fits these rooms. The center zone of the house's nine-square grid is empty and serves as a courtyard around which the other zones spiral. Thanks to the tower, this pinwheel effect occurs in three dimensions, not just in plan. The garage, deemed the least necessary room, is not included in the design, and a covered carport

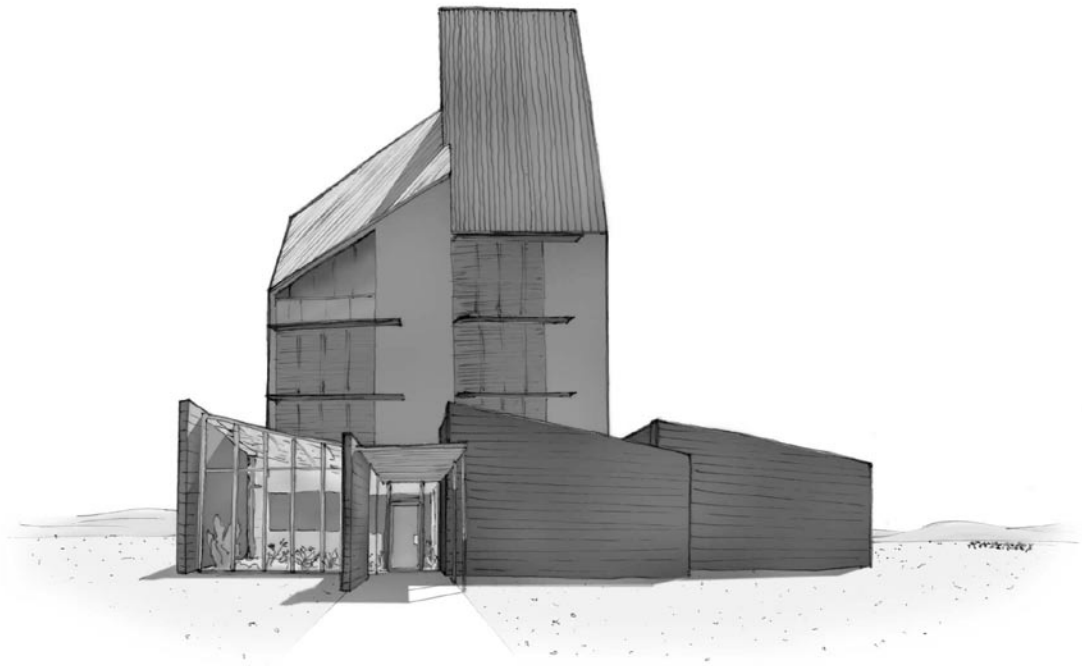

Figure 6.4.3 The covered entry to the Dubai house is part of its pinwheel form.

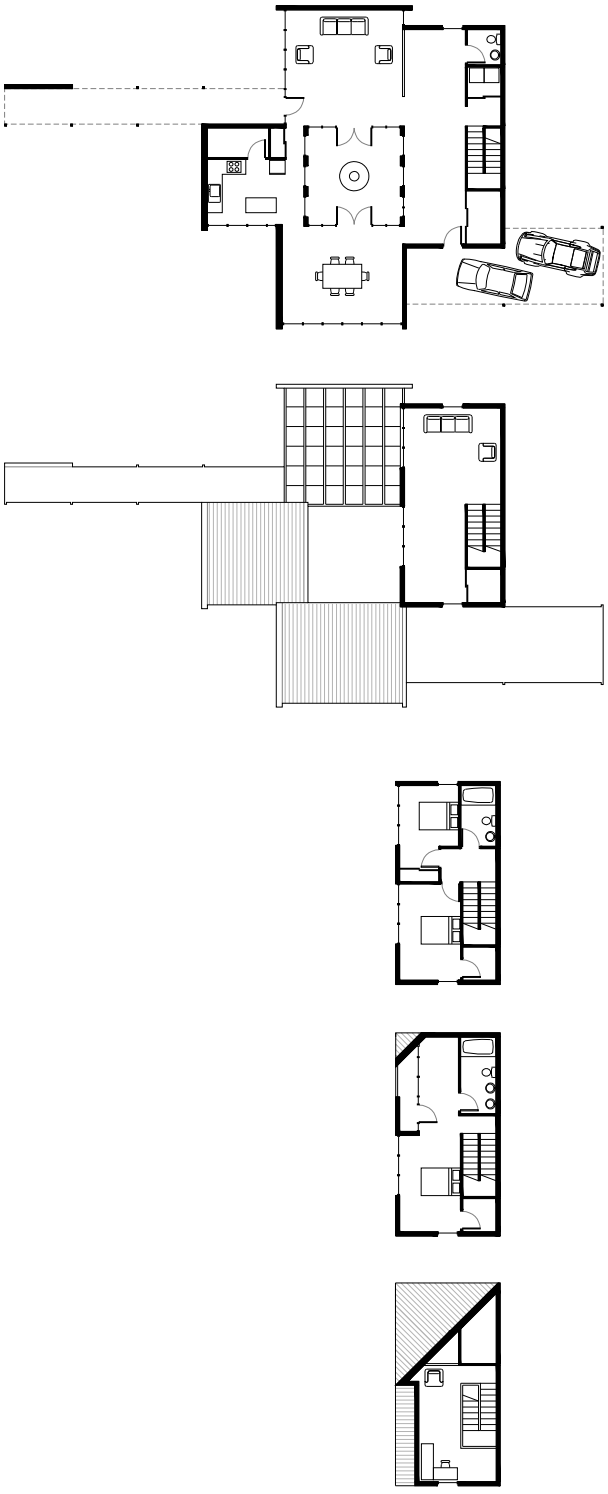

Figure 6.4.4 The five floors of the Dubai house in plan at 1’ = 1/32” scale.

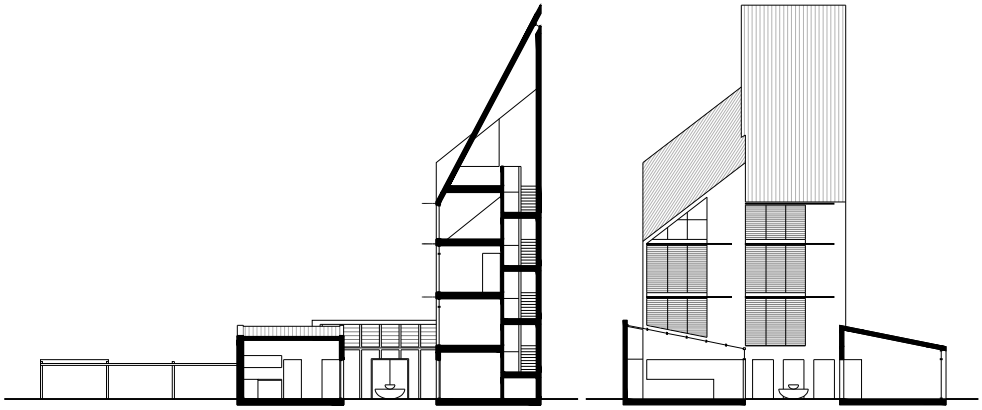

Figure 6.4.5 The Dubai house in section at 1’ = 1/32” scale.

replaces its function. The carport and a similar covered entryway add to the house’s pinwheel shape.

While the original fitness criteria did not consider passive means of cooling, incorporating them into the design can improve the building’s energy efficiency beyond that achieved by the GA. The staircase in the tower provides stack ventilation to draw warm air out of the greenhouse. The water from a fountain in the courtyard absorbs heat that would otherwise enter the zones around it. Brise-soleils over the large western windows given to the tower by the GA regulate the entry of sunlight. This limits the amount of heat entering the tower during the day without adding the need for electric lighting in the evening.

The Dubai house is a remarkable demonstration of machine creativity. A reductionist view of sustainable design tells architects that towers and glass boxes are both bad ideas in the desert from an environmental perspective. However, the GA has arrived at a solution in which a combination of the tower and glass box outperforms a number of more typical building shapes.

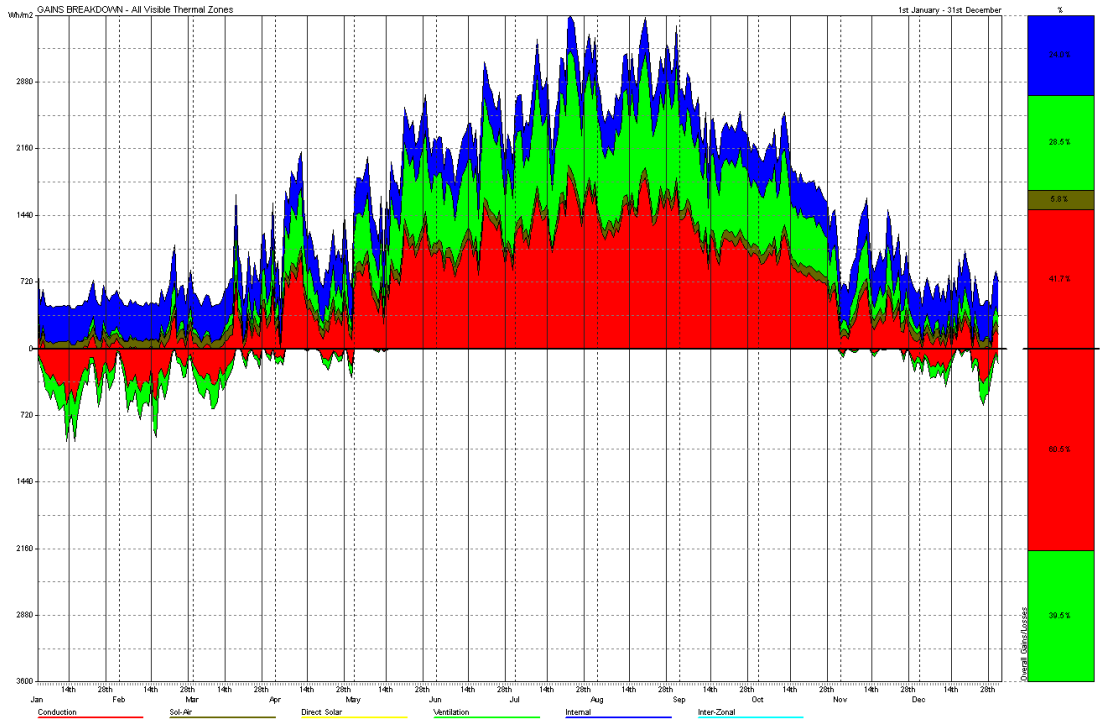

Figure 6.4.6 In Dubai's hot climate, air infiltration (green) and conduction through walls (red) are heat gains rather than losses. This works against any building in the climate because the gains cannot be balanced by losses.

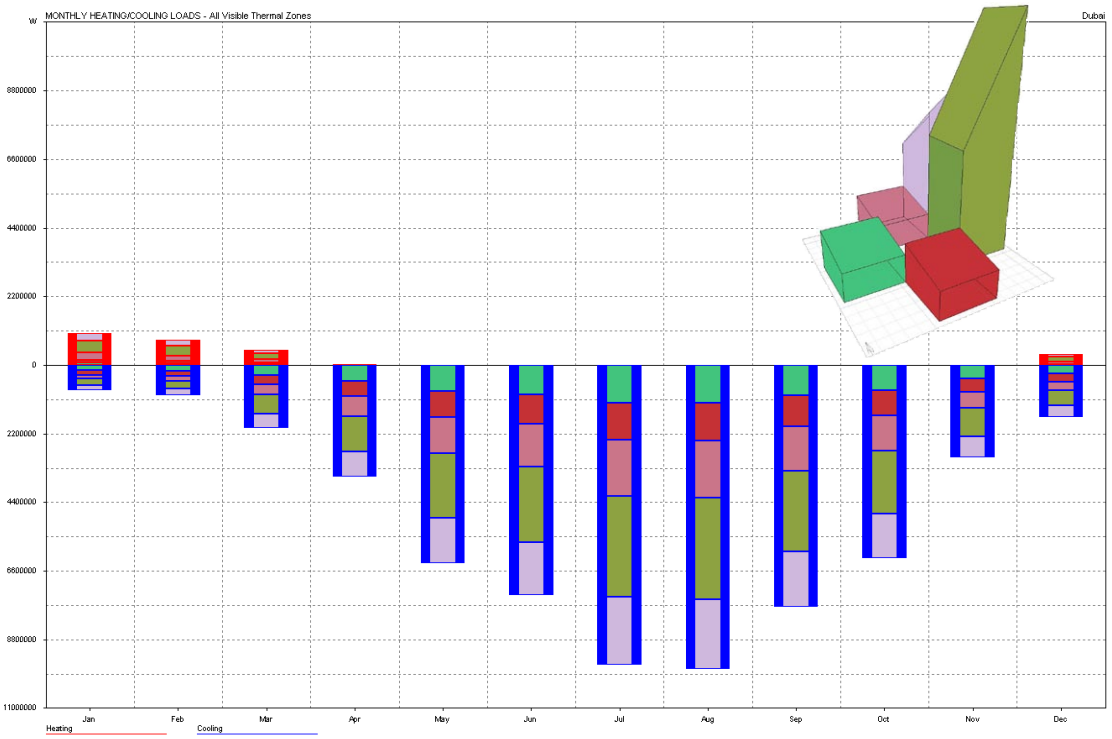

Figure 6.4.7 While it is no surprise that the largest zones in the Dubai house have the greatest cooling loads, these loads are small in proportion to the zone's volumes.

CHAPTER 7

POSSIBILITIES

> This enhancement of design through the computer touches also upon issues related to customization of production. Instead of a fixed design, non-expert users or clients would interact with an open system and co-design together with it the desired product, adapted to their specifications and particular preferences. In this setting the new role of the designer would be to build systems that allow a multitude of possibilities, and to imbue those systems with the capacity for proposing feasible and sound suggestions to a user.
>
> Pablo Miranda Carranza, 2001

Architecture is a complex adaptive system. It is made up of many interacting components and systems. Variety in the ways these parts can be assembled gives architecture the ability to adapt and evolve. The architect is in many ways a tinkerer, searching for the combinations and arrangements of parts that best suit a specified purpose.

Because all complex adaptive systems are capable of evolutionary improvement, the genetic algorithm (GA) is a useful architectural tool. This thesis shows that deterministic crowding (DC) provides significant improvement in thermal performance and interior daylighting in most climates, and the amount of improvement increases as the climate becomes more hostile. Furthermore, a single run of a DC algorithm produces a variety of solutions, all of which outperform the "average" building design. Multiple runs of the algorithm only increase the variety of solutions found.

Other types of GAs may also be useful, but lack some of the benefits of DC. The mu+lambda algorithm achieves better performance and improves at a faster rate, but it sacrifices diversity. Parallel hill-climbers provide more diversity, but many solutions have poor thermal performance.

The real advantage of the GA is that it finds workable solutions in parts of the search space where human architects would not think to look. Even in fantasy, human creativity does not deviate too much from experience.[41] When designing a sustainable building, architects look to previous examples of sustainability and to reductionist principles for guidance. The computer has no experience or knowledge to guide it. No logic told the GA not to investigate greenhouses in Dubai's hot climate, since the fact that greenhouses heat quickly in the sun was not written into the program. Instead, the computer blindly experimented with glazed surfaces and found, against all odds, one greenhouse that is sustainable in the desert. No human architect could be expected to comb through values of 81 different parameters looking for a similar design.

Even still, 81 is a small number of parameters if one wants to describe a real building. Limits on computational speed mean that it will take time for genetic algorithms to become practical as architectural form generators. As the speed of evaluations increases, it will be possible to evolve larger populations of more complex buildings through greater numbers of generations, increasing the effectiveness of the algorithm. More processing power will also make it practical to evaluate each individual for the entire year's climate data, rather than just one month, and to use robust thermal models that are more true-to-life than Ecotect's admittance method. Criteria that examine structure, lifetime cost, egress, and other factors can also be

[41] Jacob 1161

added to the concept of fitness. The resulting early predictions of performance will put architects in a better position to begin their designs.

Imagine architectural design in a world without computational limits. A neighborhood could be modeled as a living population. The first agents in this artificial life simulation are a collection of randomly placed, randomly generated houses. A population of occupants, based on the demographics of a real city, creates a market of supply and demand for homes.[42] New houses built in the neighborhood are variations on the more successful houses already in existence, while buildings that cannot attract occupants are torn down. Instead of capping the size of the building population, scarcity is introduced by limiting the resources and services available to the neighborhood. Over time, the neighborhood adapts to meet the needs of an occupant population that in turn changes depending on the housing available to it. Order arises naturally without any assumptions on the part of the designer. Each run of the simulation produces a housing development unique to its location's climate and social makeup.

Architecture is one of many complex adaptive systems. Every building is part of a vast interconnected web of social systems, economic systems, and ecosystems. Recognizing architecture's complex adaptive properties will give architects an advantage when searching for new designs. Optimization algorithms give architects the power to improve the performance, not only of individual building systems, but of buildings as a whole. With this added insight, architects can better respond to the needs of the larger complex systems around them.

[42] See Epstein 165 for an artificial life simulation of social forces that shape a neighborhood.

APPENDIX A

SOLUTIONS CATALOG

Diversity is important both for the algorithm, which needs to assemble its solutions from as many fit parts as it can find, and for the architect, who needs to apply qualitative criteria to design selection. Every time the algorithm runs, it generates new solutions, giving the architect even more options for moving forward. The three designs taken into the schematic phase in this thesis are some of the algorithm's more creative solutions, but not the only ones. This appendix catalogs solutions that survived through fifty rounds of selection in each of the genetic algorithm trials. In some cases, many very different designs turn out to be quite fit in terms of their energy consumption.

The fitness scale under each solution shows its performance relative to the best individual from that series of trials. An individual's fitness may be compared to others that use the same fitness criteria. Hence, Ithaca trials using deterministic crowding are comparable to those using the parallel hill-climbing algorithm, but not to trials in Anchorage or to those using climate data from the entire year.

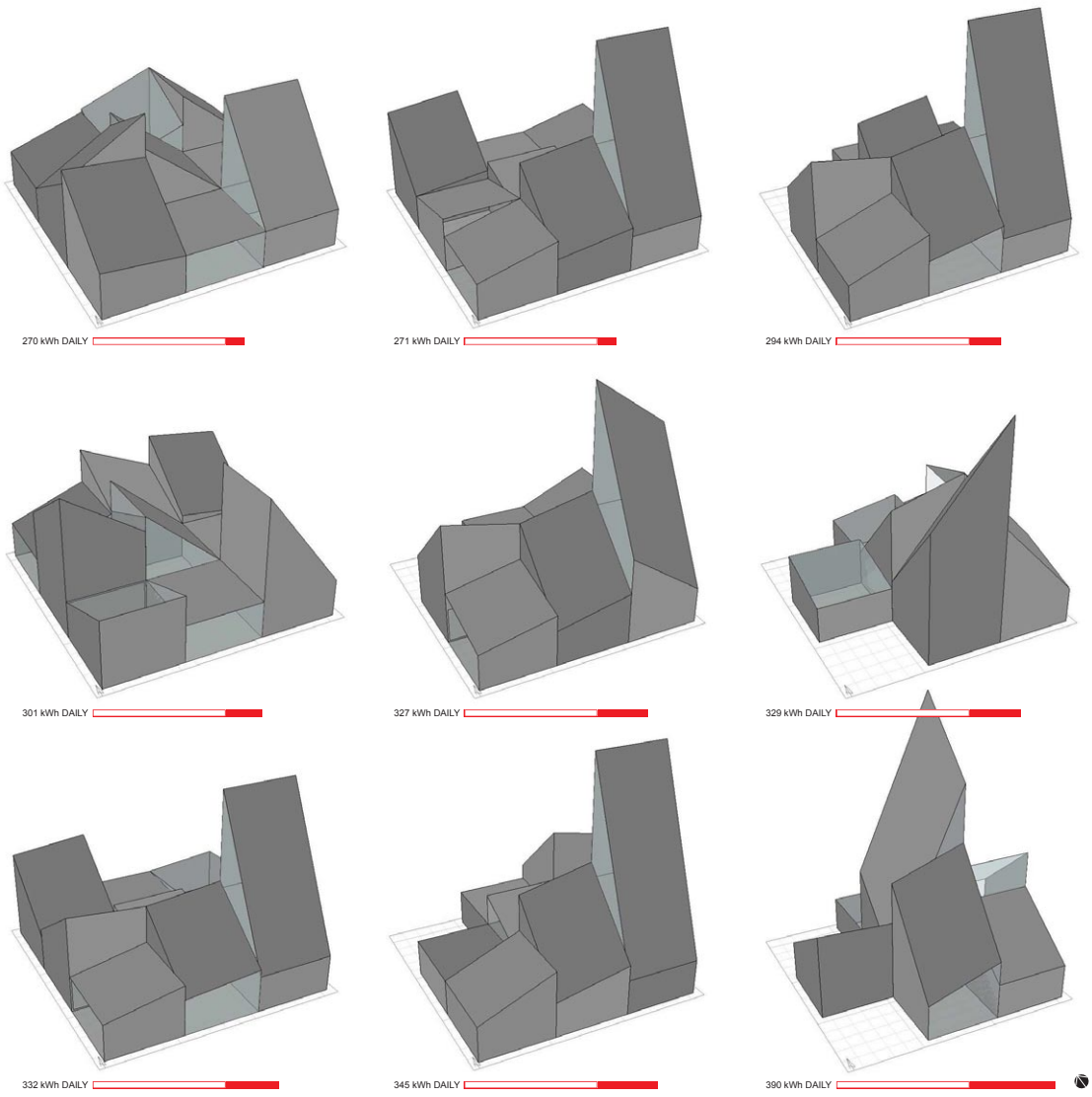

Figure A.1 Solutions from one run of the optimization algorithm, seen from the southwest. The top right solution is the basis for the Anchorage house.

Location: Anchorage

Algorithm: Deterministic Crowding

Period: January

Trial: 1

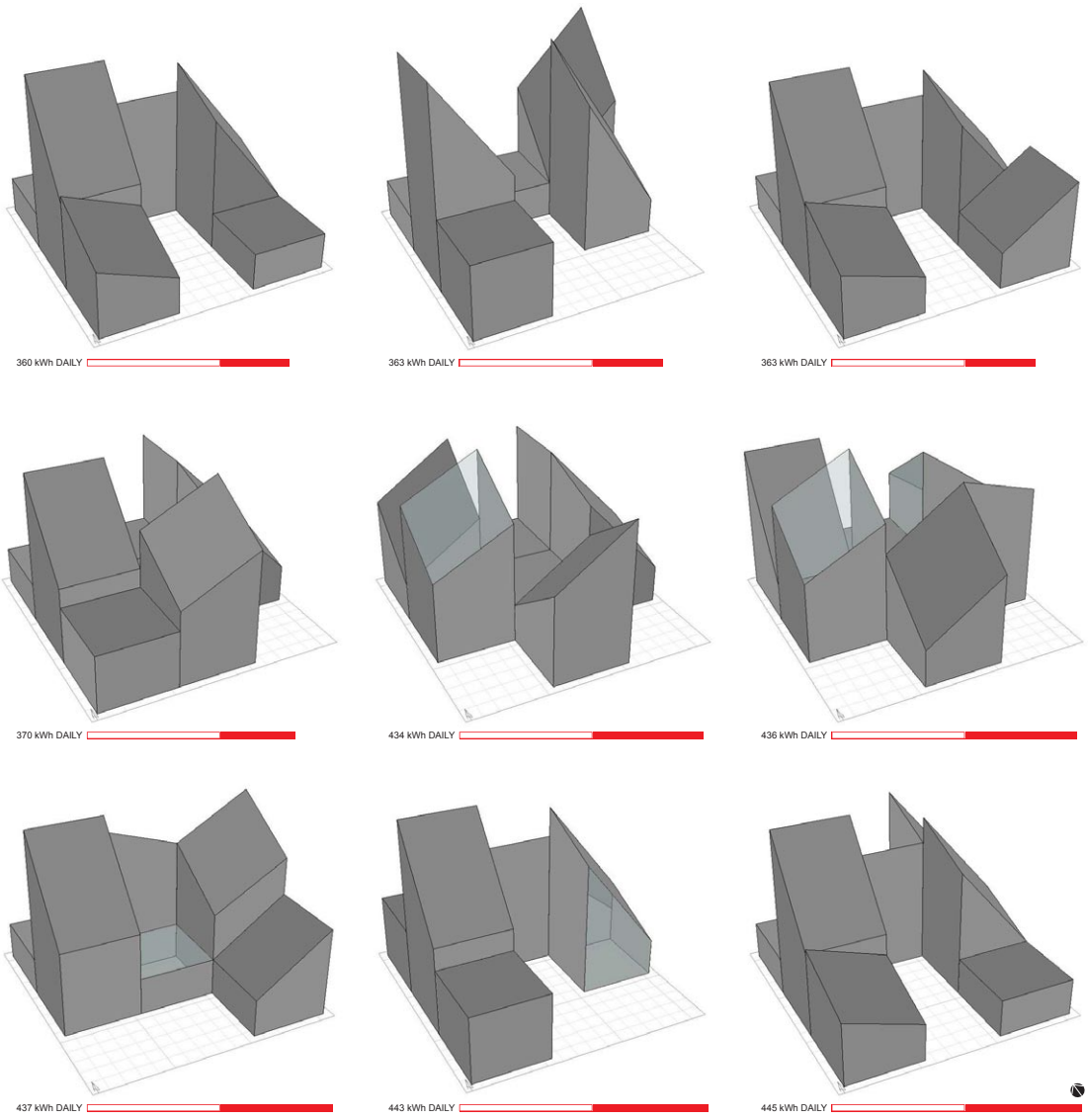

Figure A.2 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Anchorage

Algorithm: Deterministic Crowding

Period: January

Trial: 2

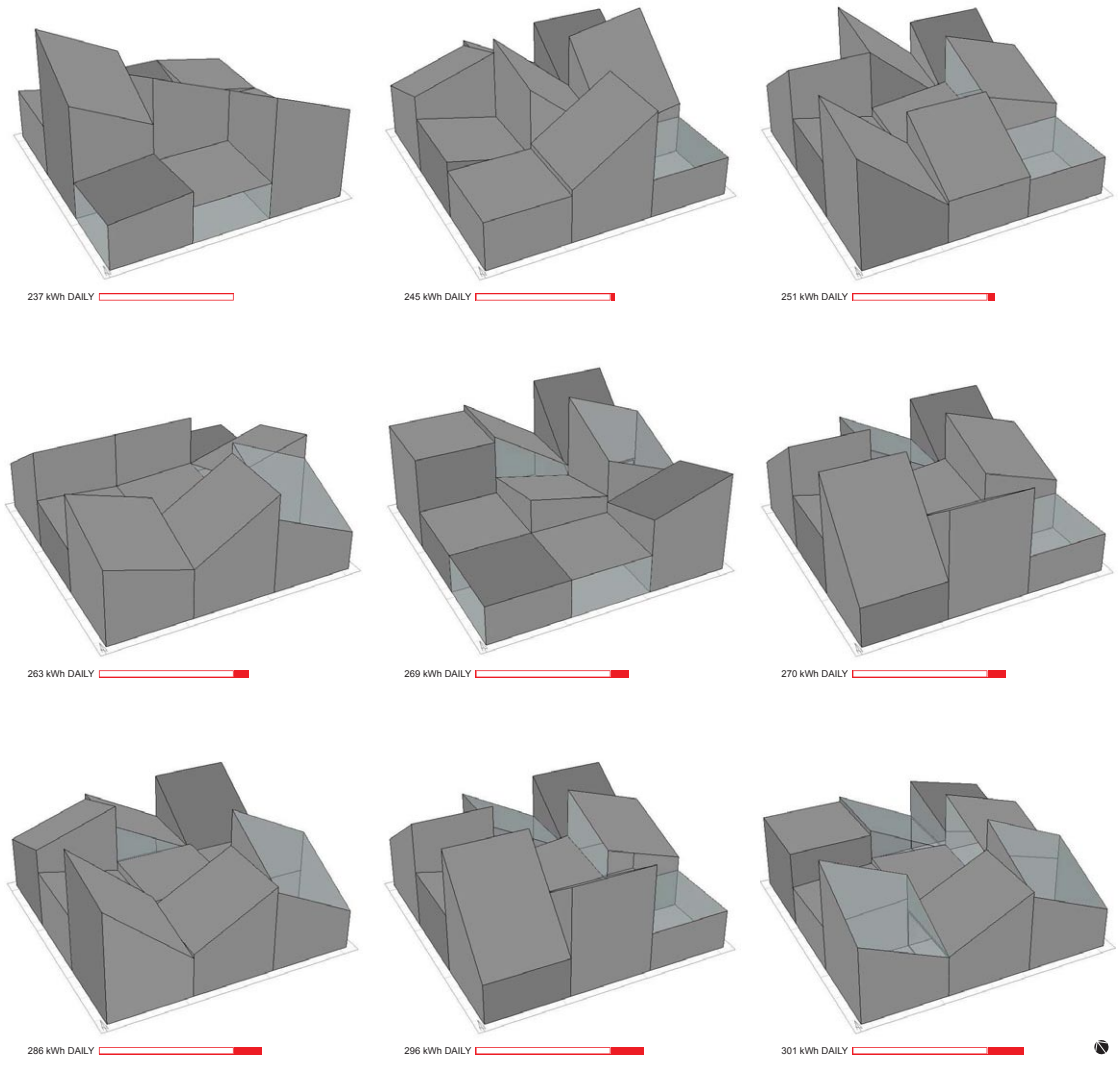

Figure A.3 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Anchorage

Algorithm: Deterministic Crowding

Period: January

Trial: 3

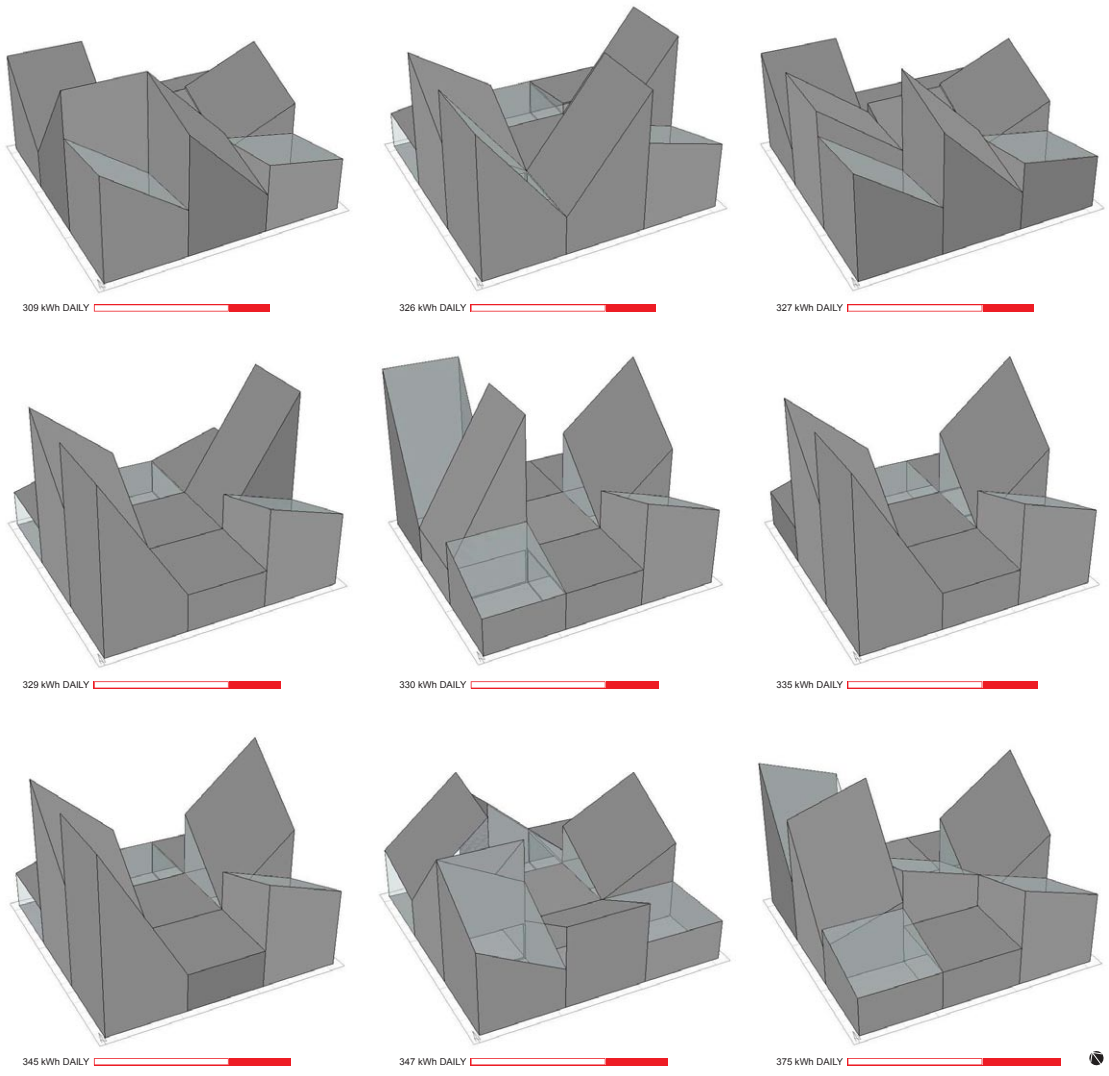

Figure A.4 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Anchorage

Algorithm: Deterministic Crowding

Period: January

Trial: 4

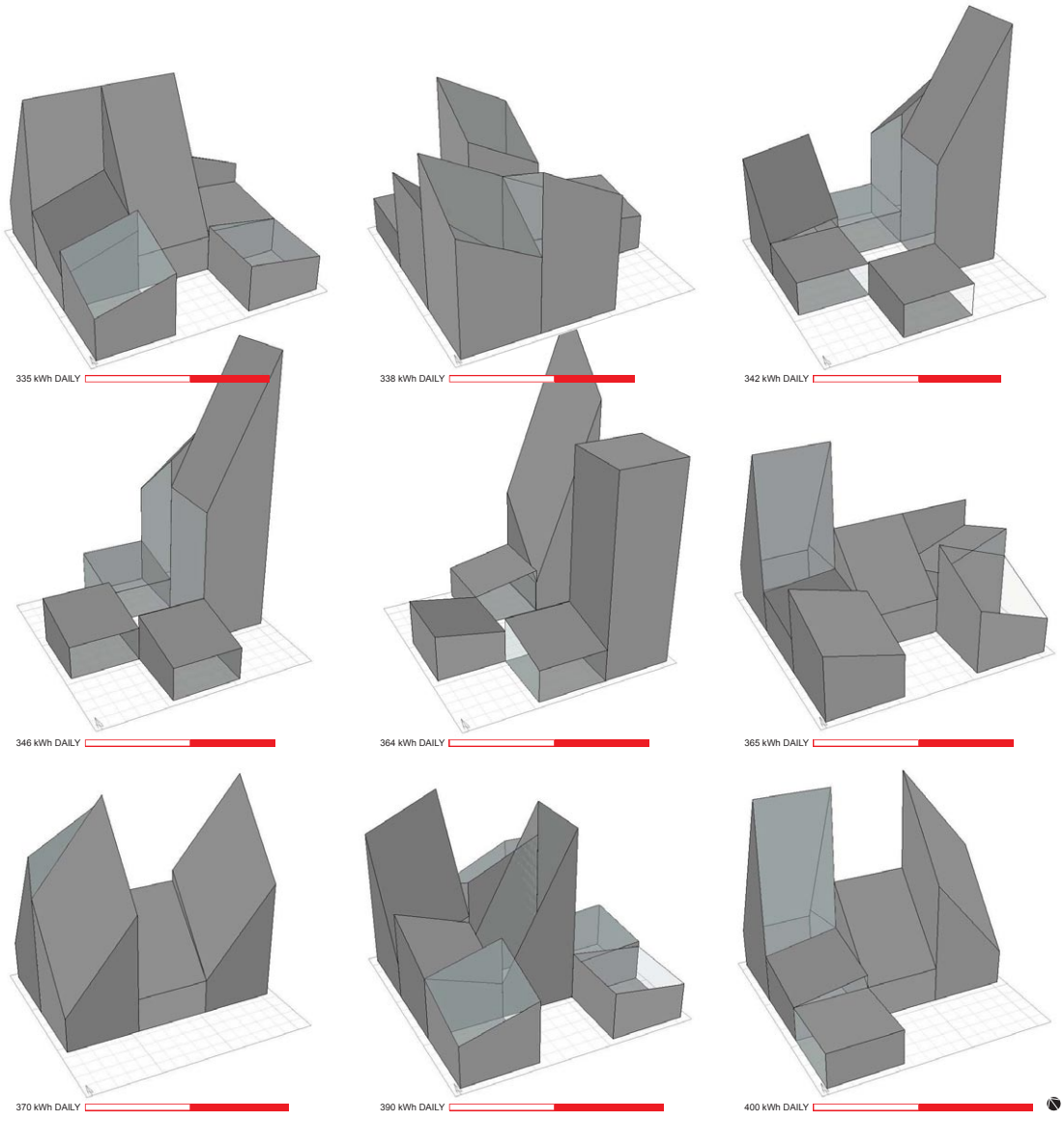

Figure A.5 Solutions from one run of the optimization algorithm, seen from the southwest. The solution at the left end of the second row is the basis for the Dubai house.

Location: Dubai

Algorithm: Deterministic Crowding

Period: August

Trial: 1

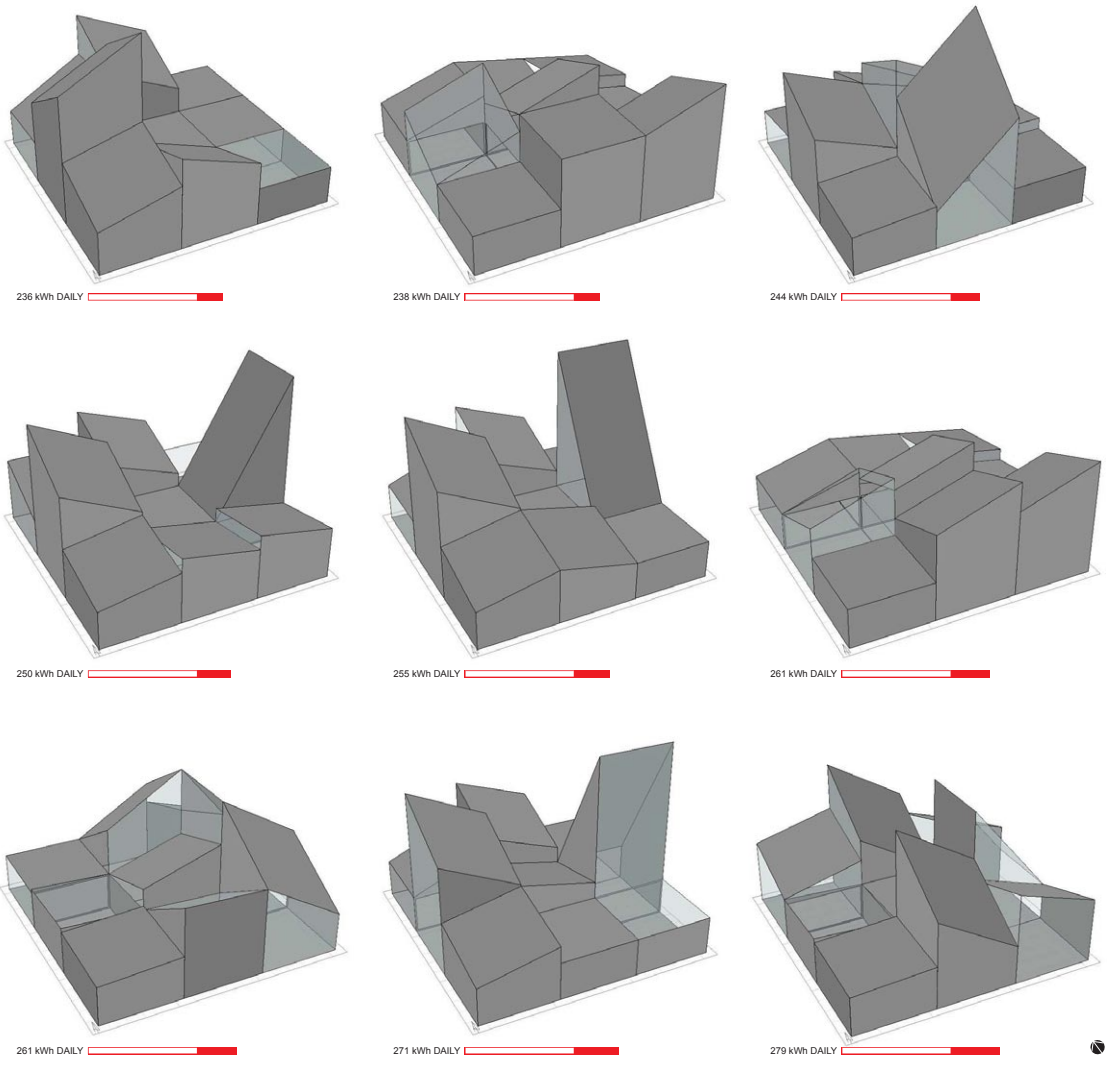

Figure A.6 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Dubai

Algorithm: Deterministic Crowding

Period: August

Trial: 2

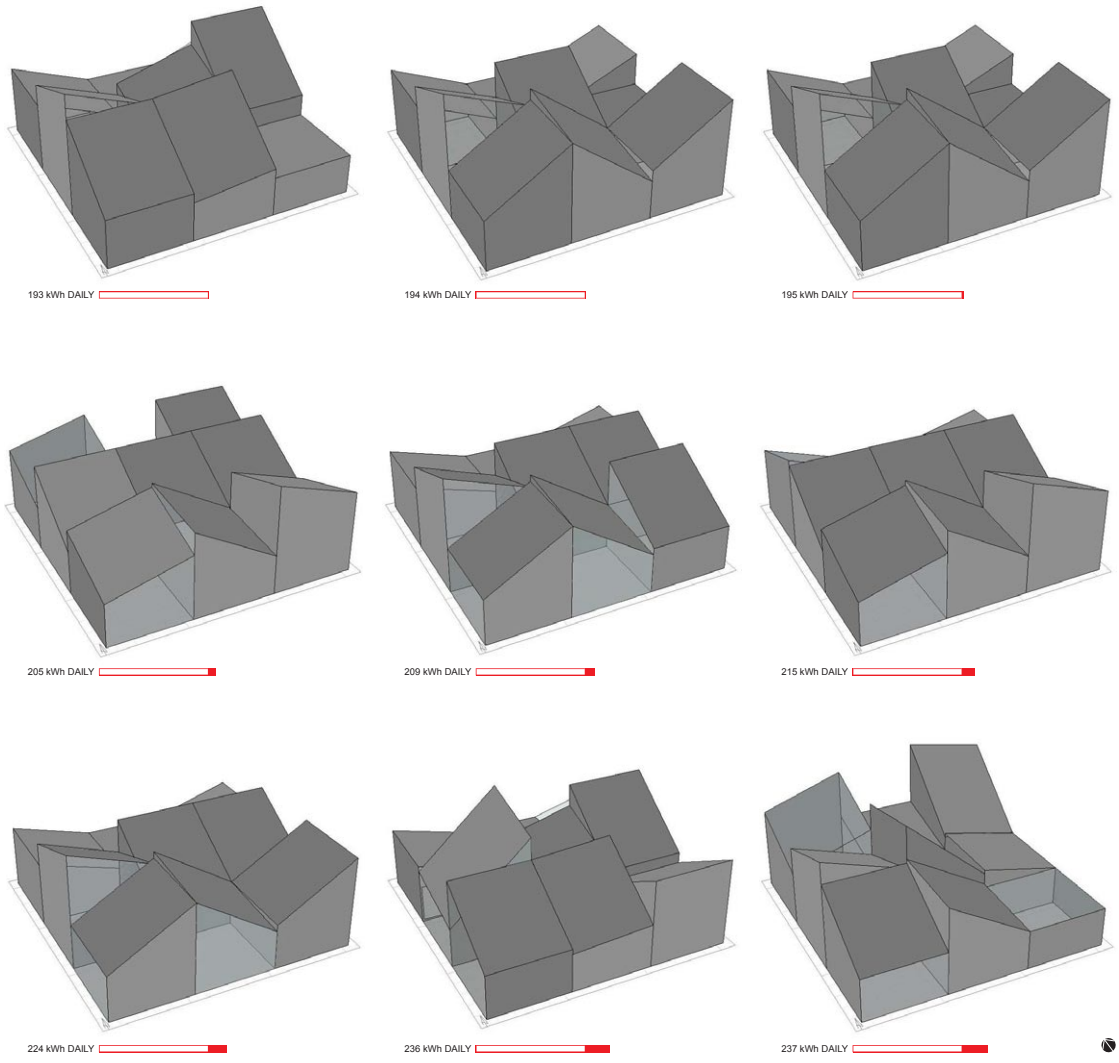

Figure A.7 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Dubai

Algorithm: Deterministic Crowding

Period: August

Trial: 3

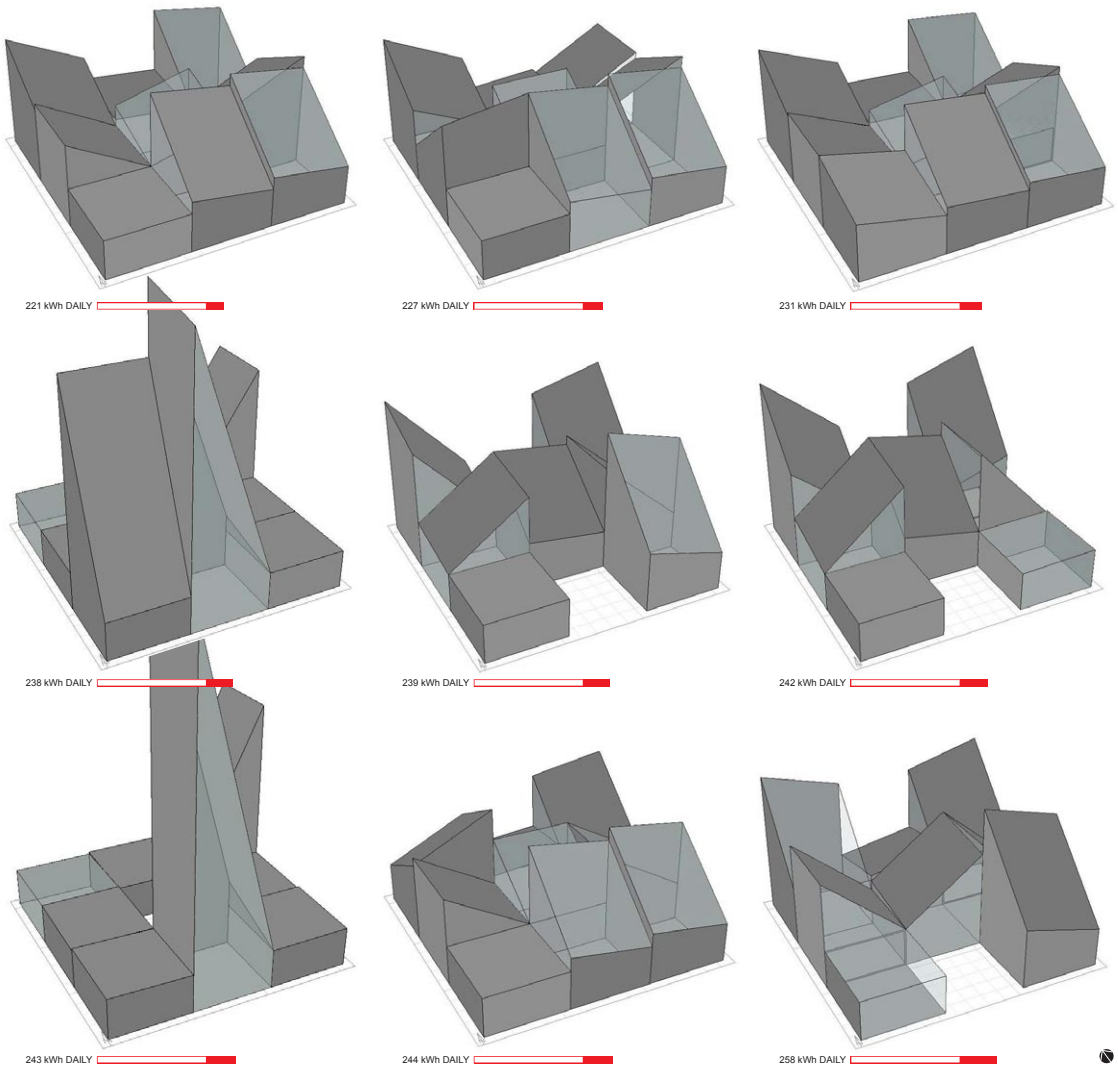

Figure A.8 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Dubai

Algorithm: Deterministic Crowding

Period: August

Trial: 4

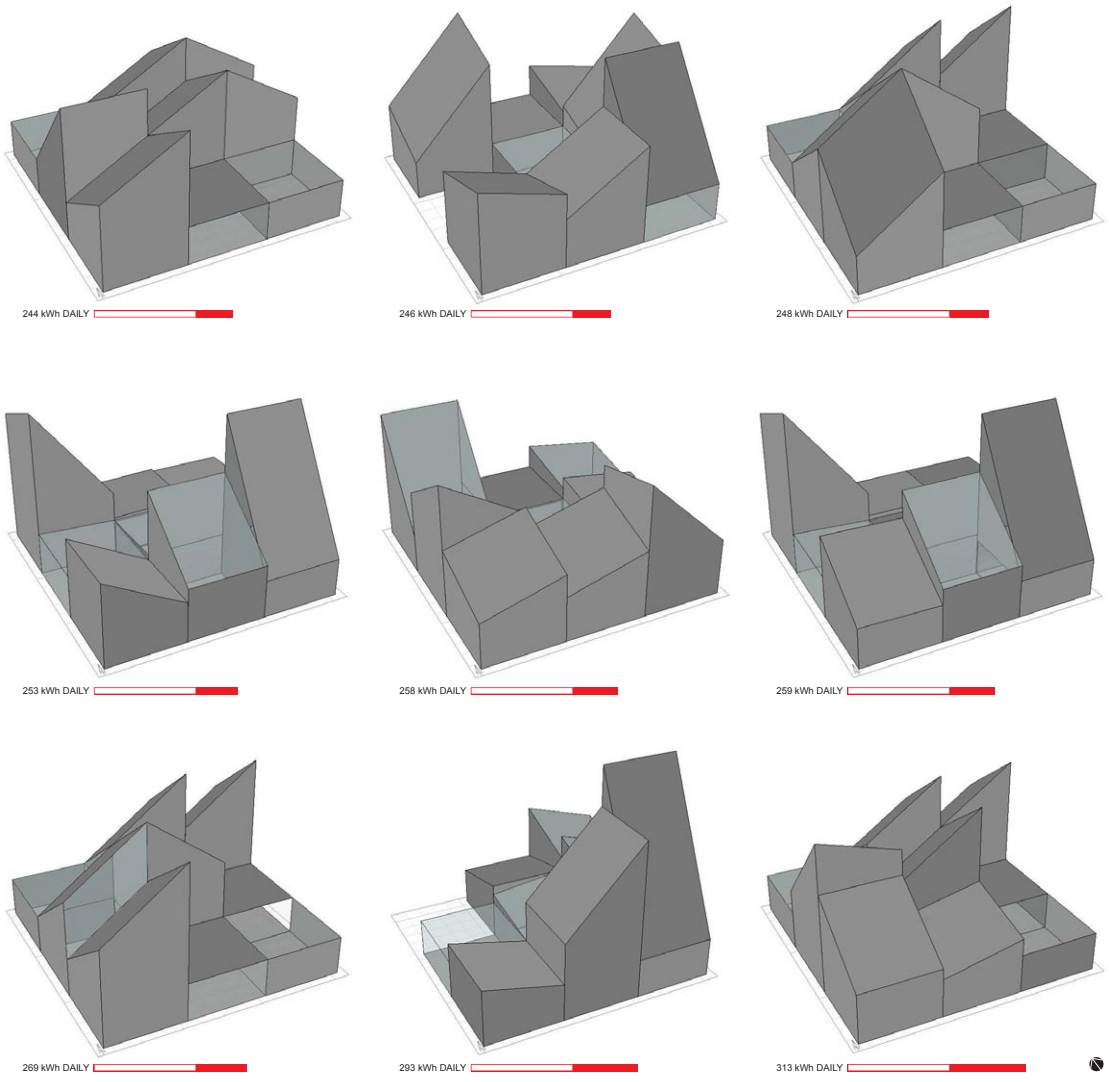

Figure A.9 Solutions from one run of the optimization algorithm, seen from the southwest. The top left solution is the basis for the Ithaca house.

Location: Ithaca

Algorithm: Deterministic Crowding

Period: January

Trial: 1

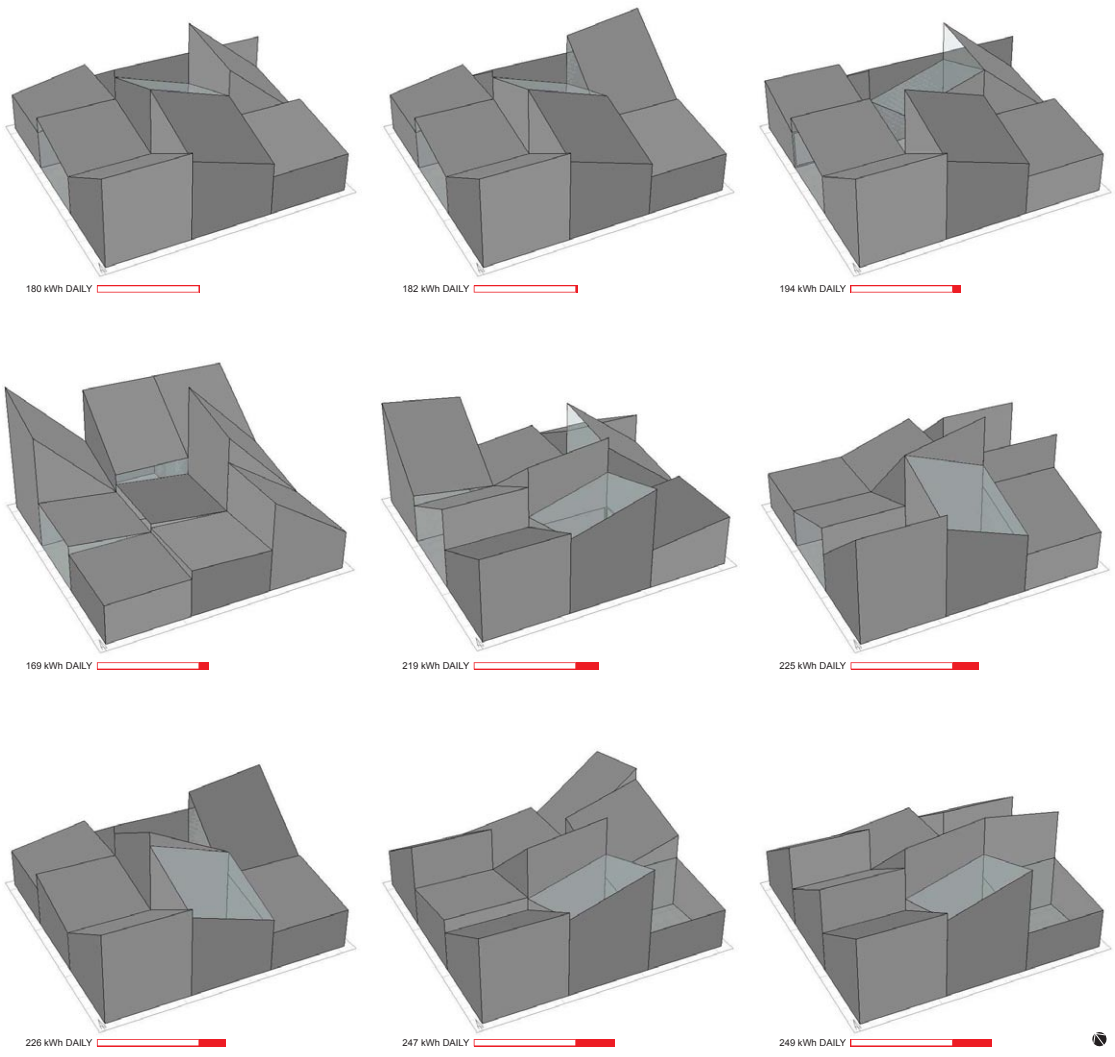

Figure A.10 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Ithaca

Algorithm: Deterministic Crowding

Period: January

Trial: 2

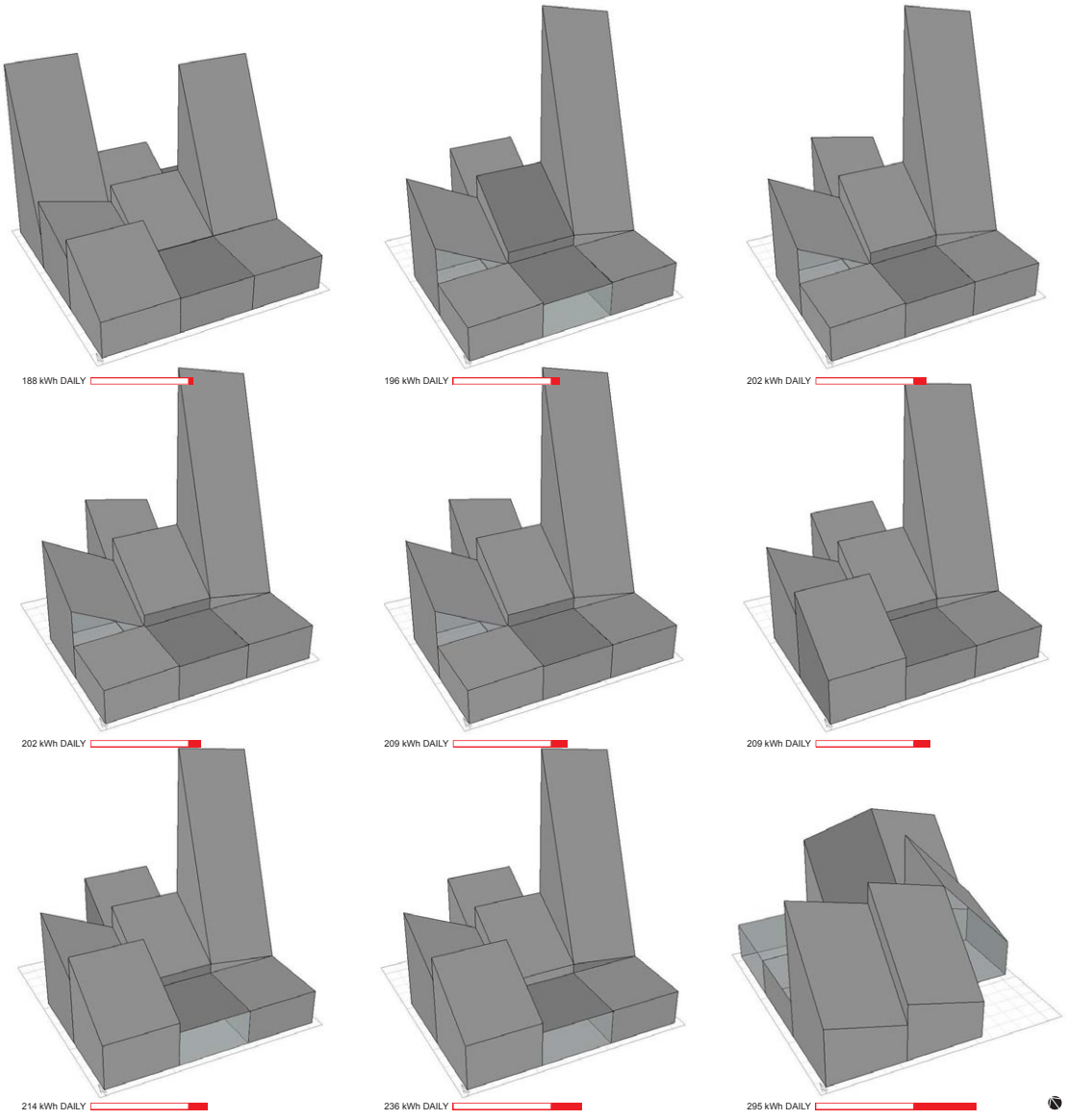

Figure A.11 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Ithaca

Algorithm: Deterministic Crowding

Period: January

Trial: 3

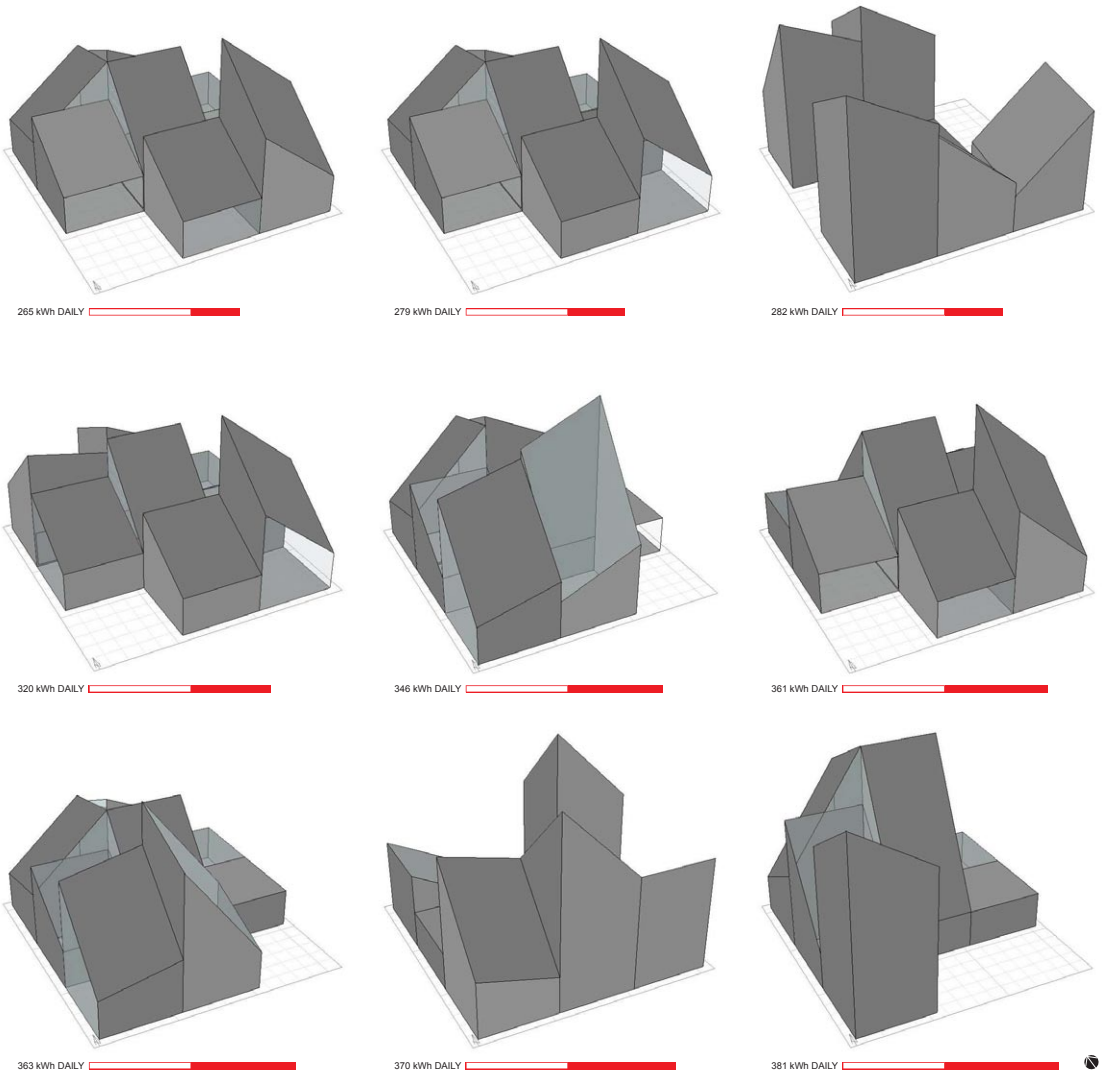

Figure A.12 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Ithaca

Algorithm: Deterministic Crowding

Period: January

Trial: 4

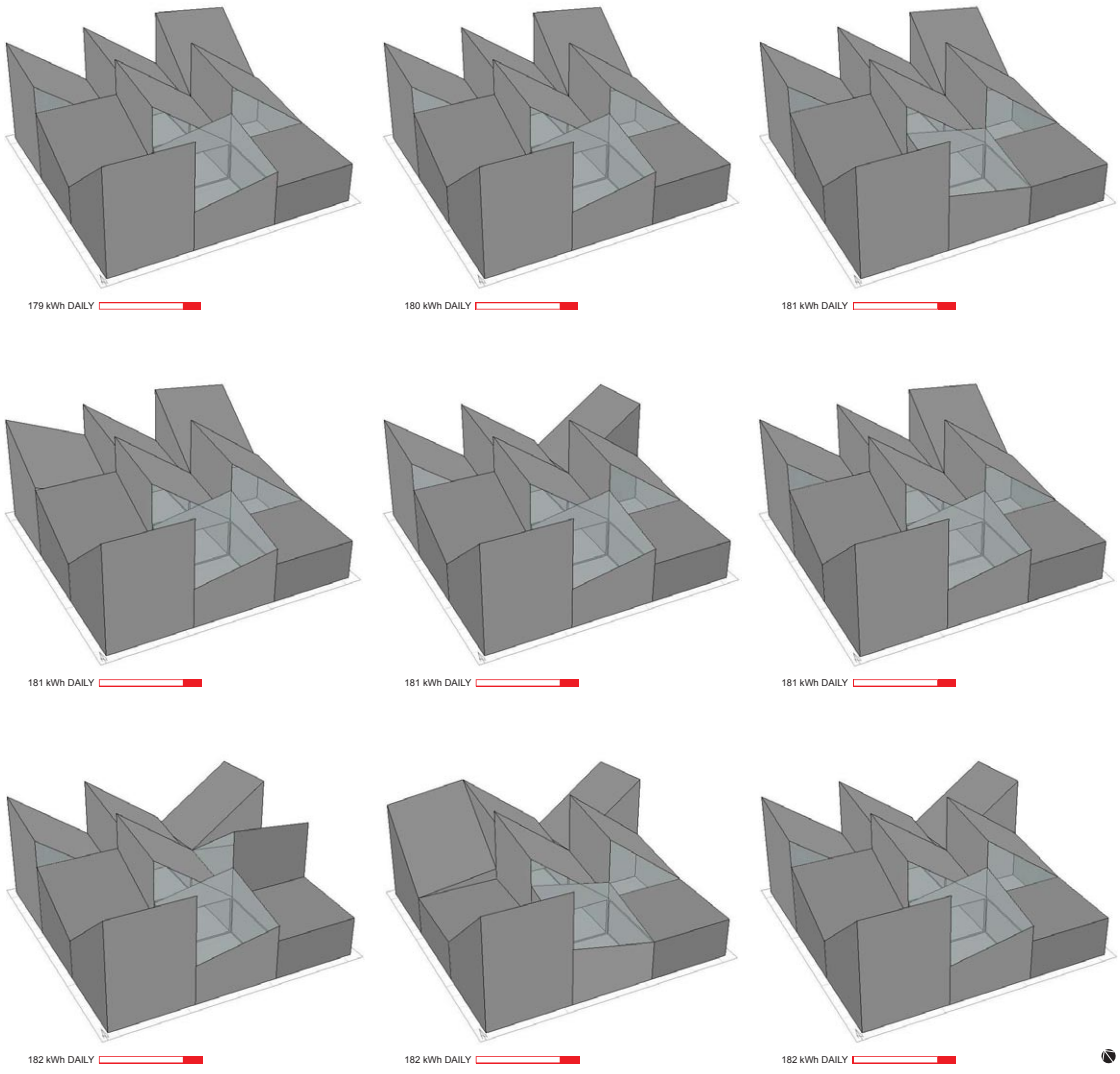

Figure A.13 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Ithaca

Algorithm: Mu+Lambda

Period: January

Trial: 1

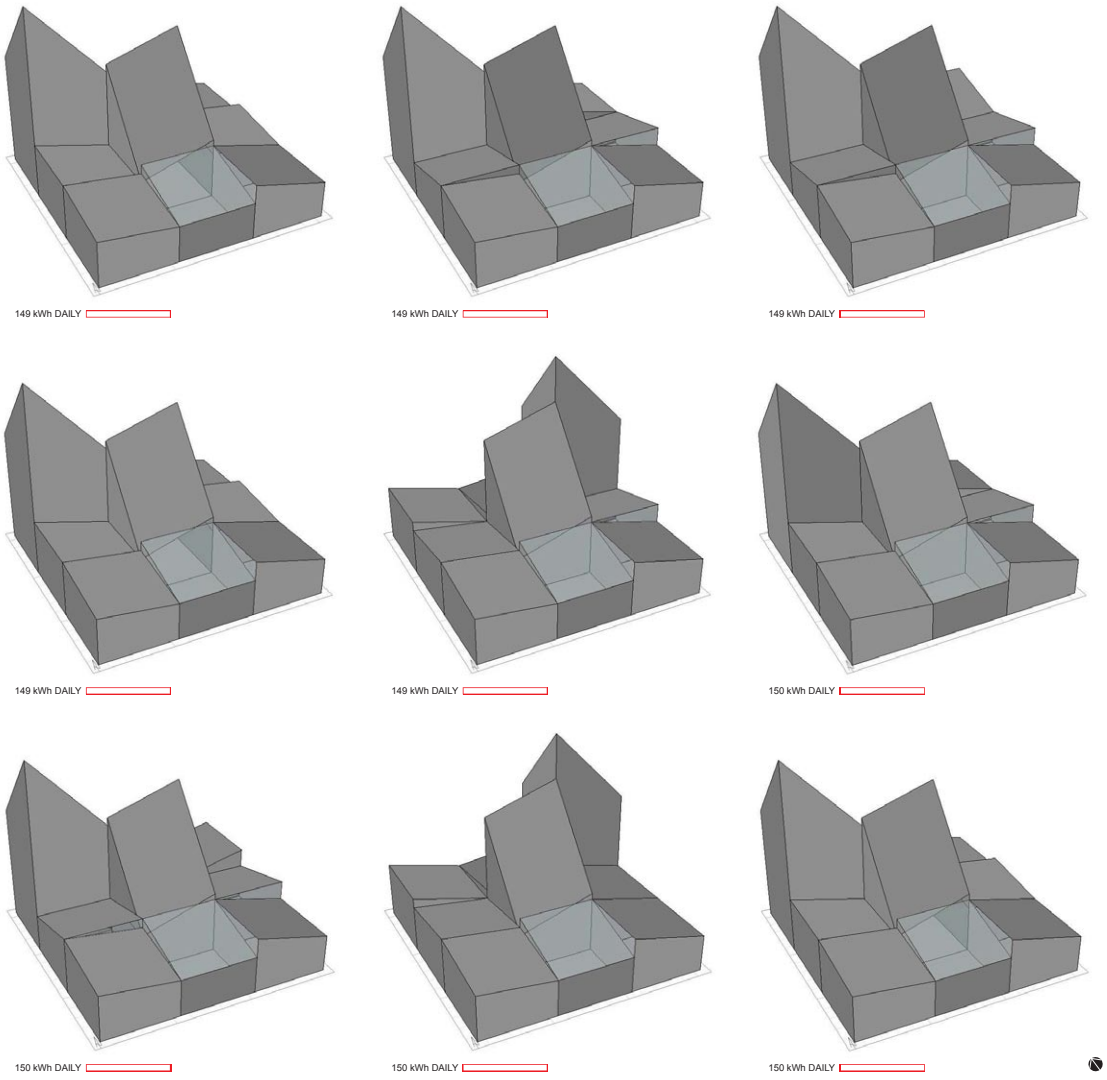

Figure A.14 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Ithaca

Algorithm: Mu+Lambda

Period: January

Trial: 2

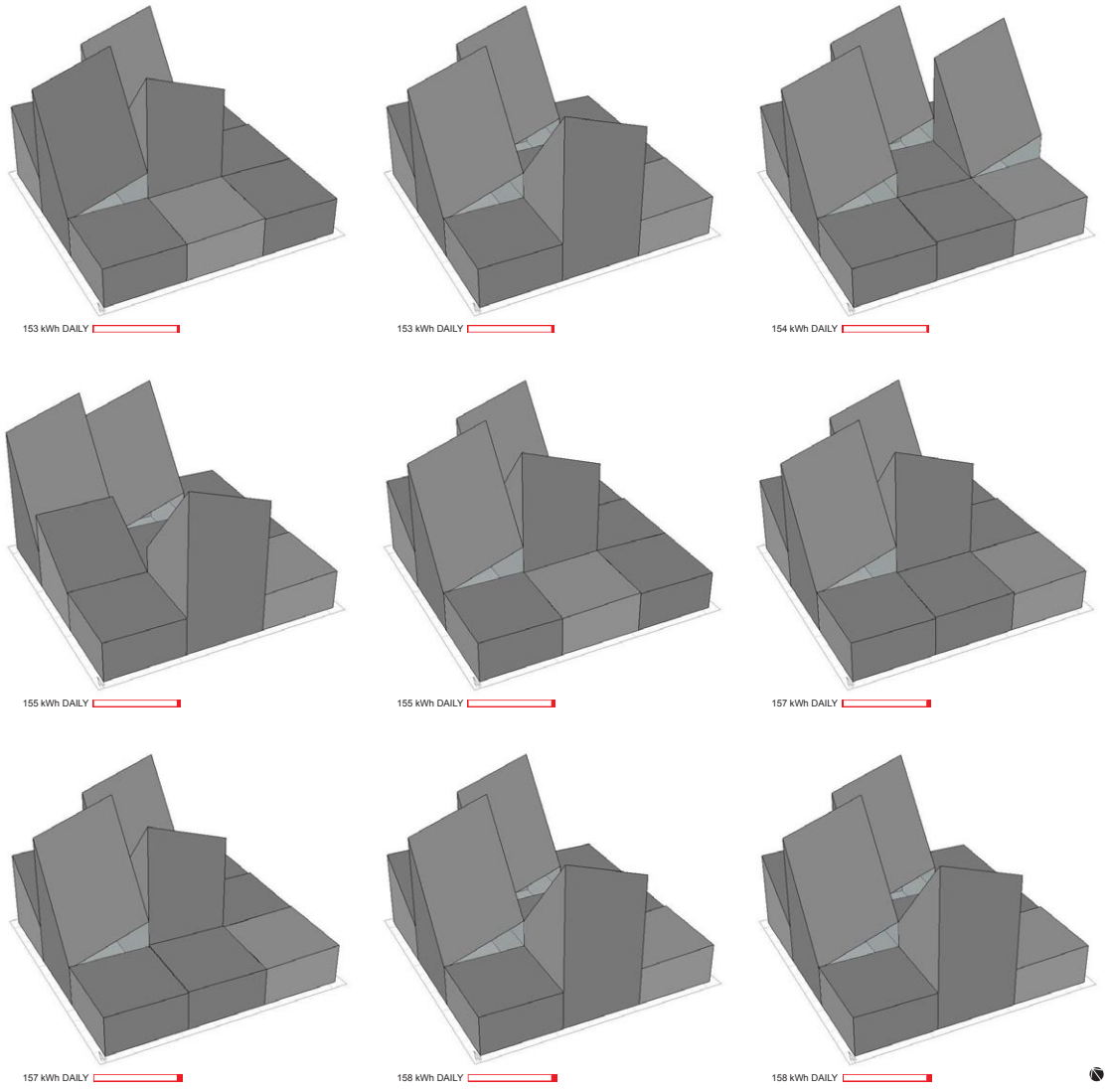

Figure A.15 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Ithaca

Algorithm: Mu+Lambda

Period: January

Trial: 3

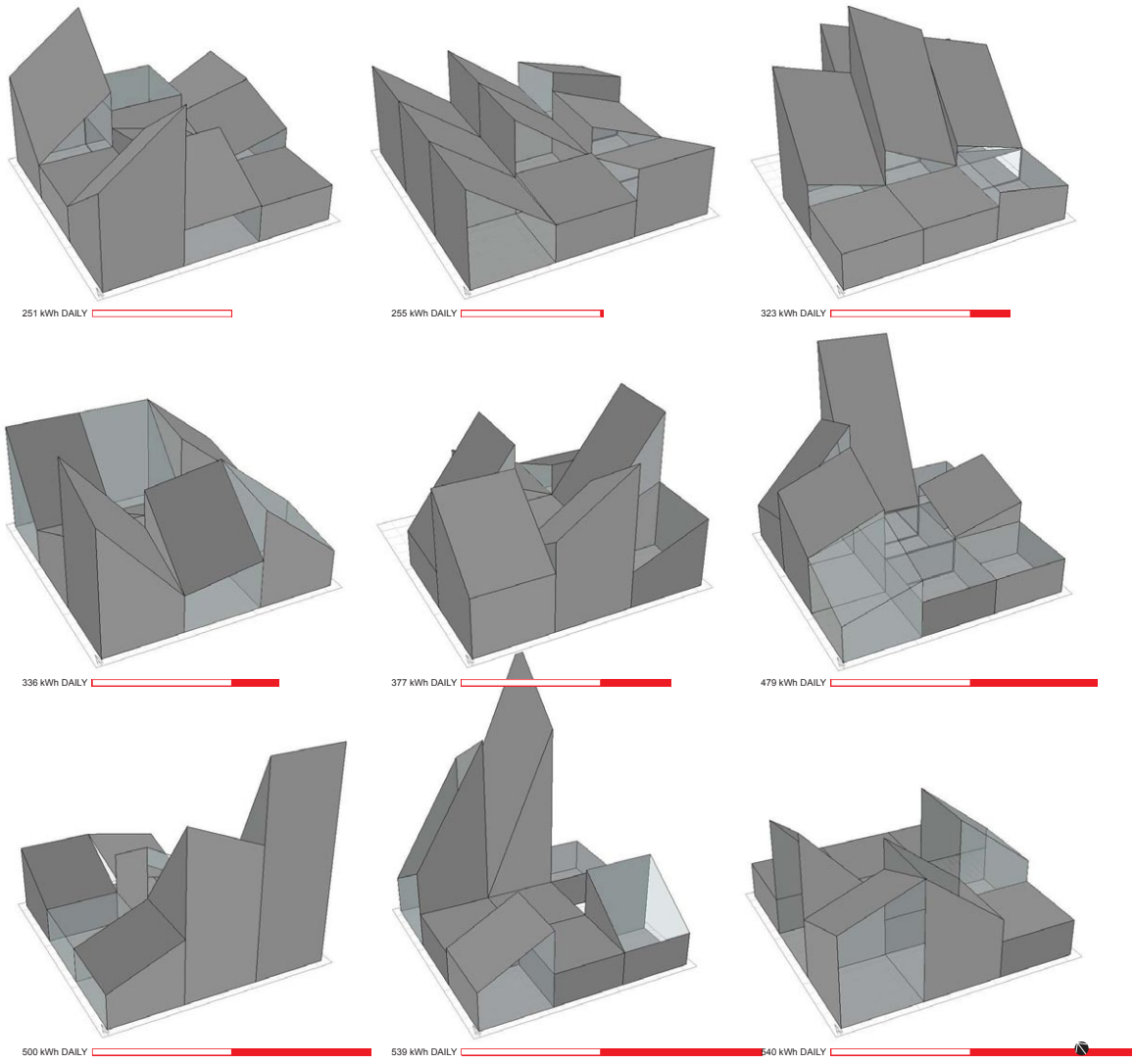

Figure A.16 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Ithaca

Algorithm: Parallel Hill-Climber

Period: January

Trial: 1

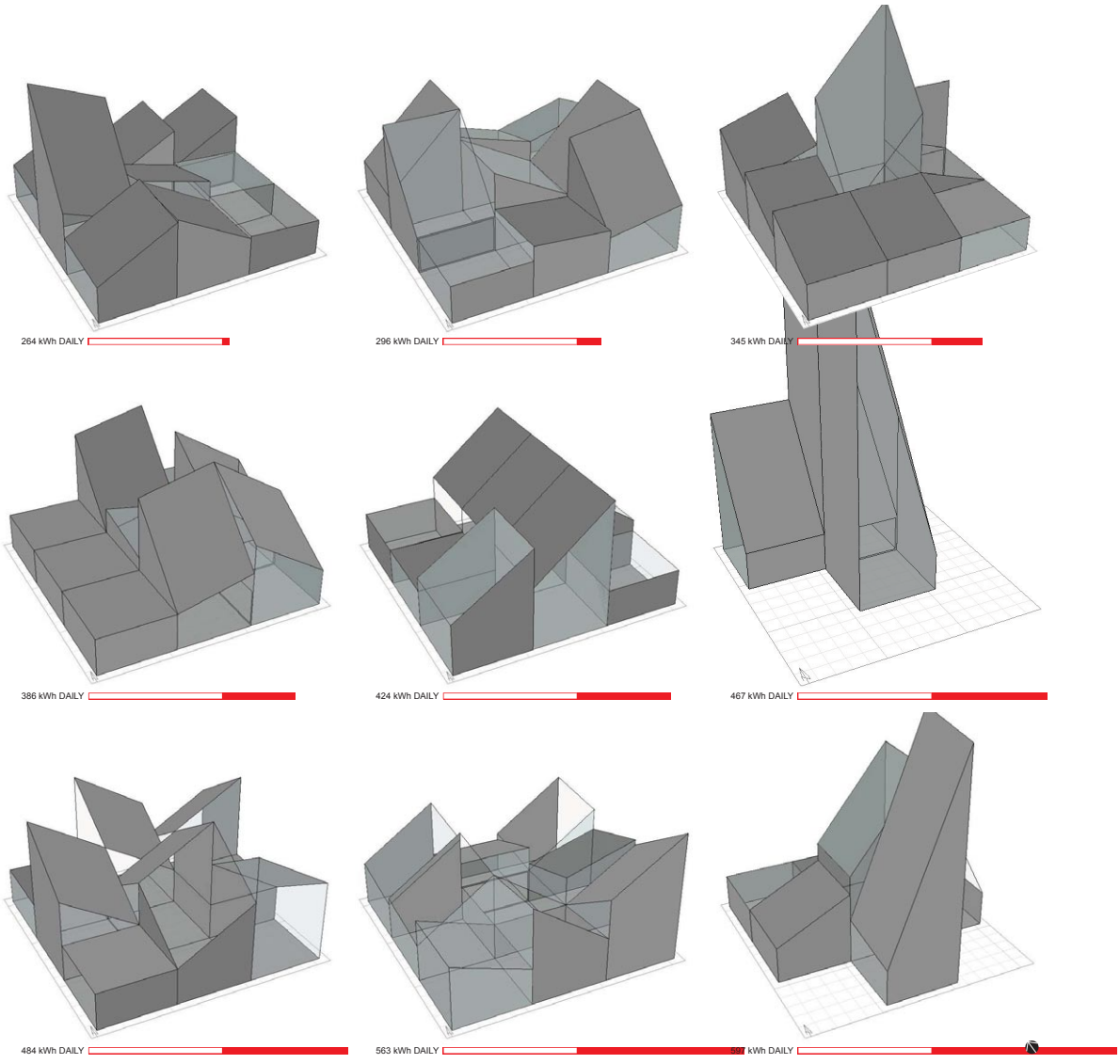

Figure A.17 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Ithaca

Algorithm: Parallel Hill-Climber

Period: January

Trial: 2

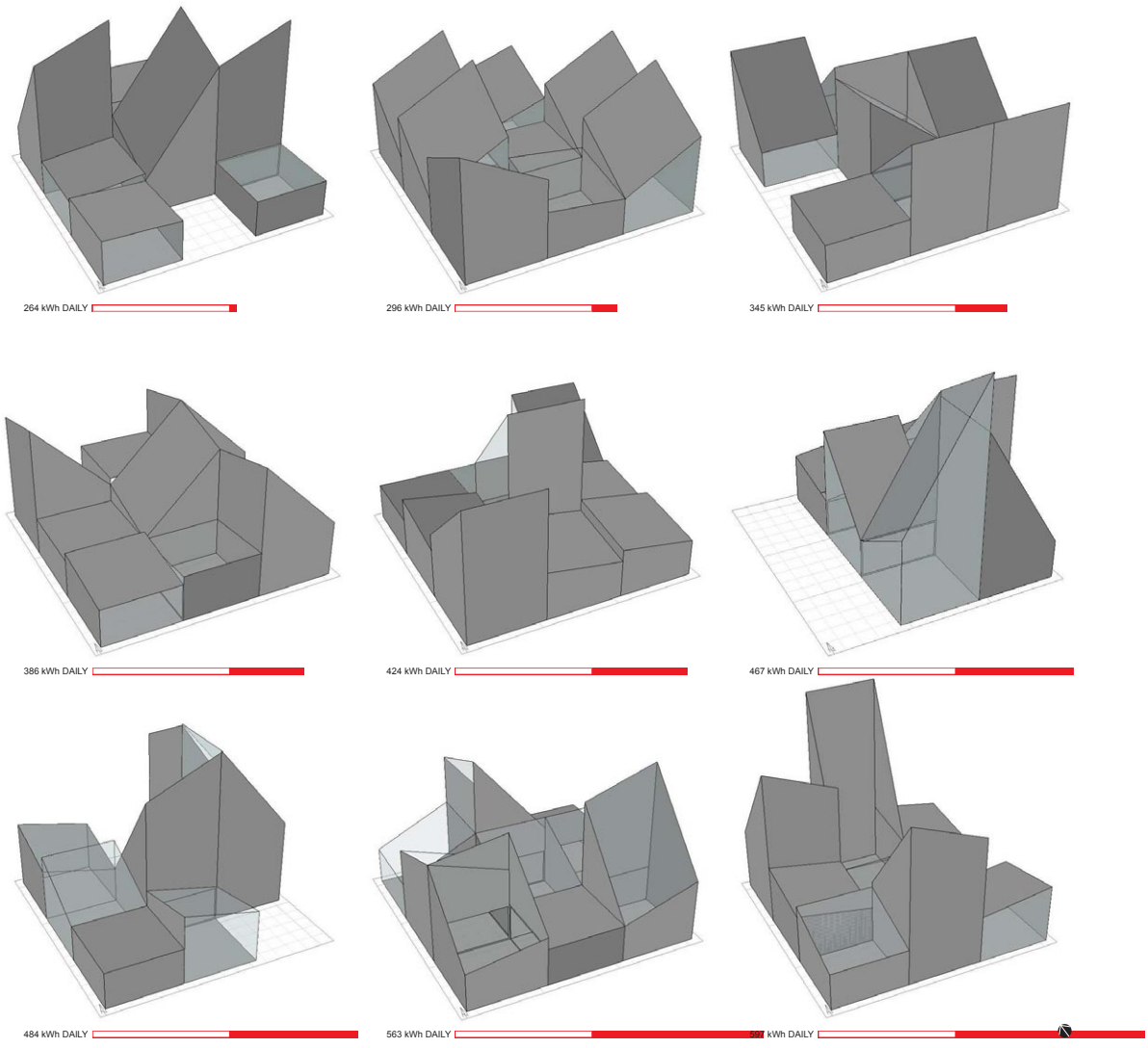

Figure A.18 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Ithaca

Algorithm: Parallel Hill-Climber

Period: January

Trial: 3

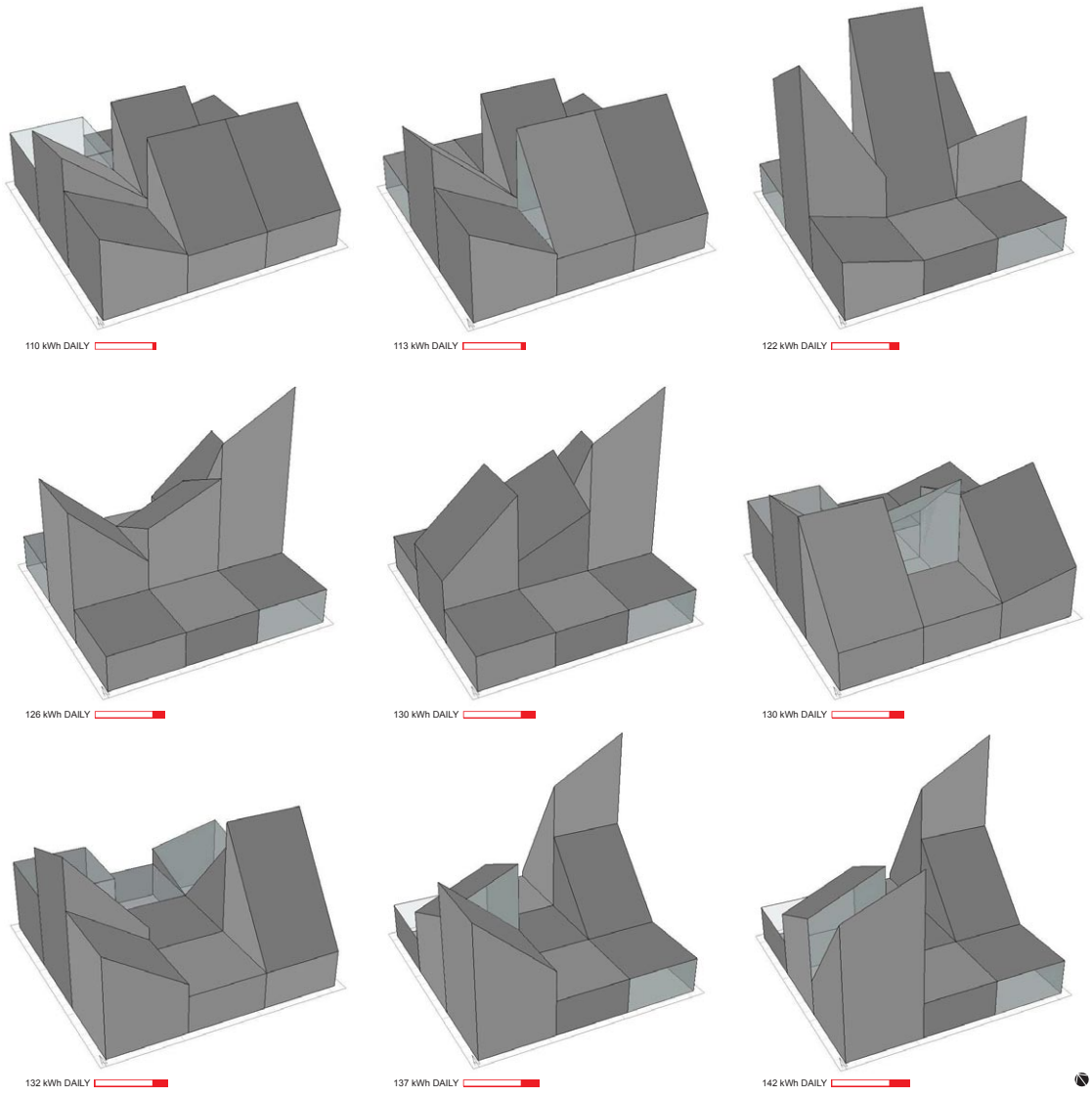

Figure A.19 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Ithaca

Algorithm: Deterministic Crowding

Period: Full Year

Trial: 1

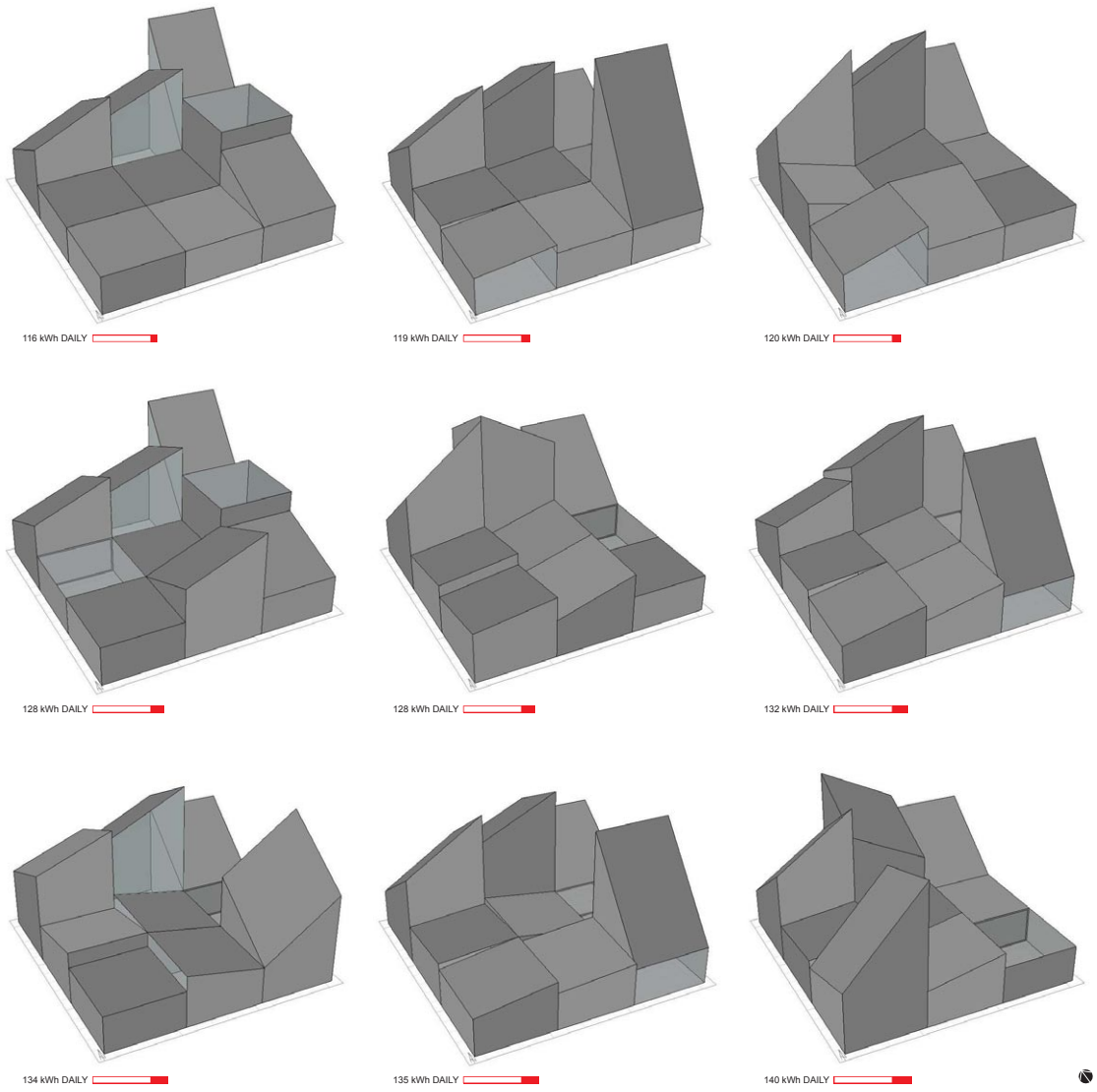

Figure A.20 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Ithaca

Algorithm: Deterministic Crowding

Period: Full Year

Trial: 2

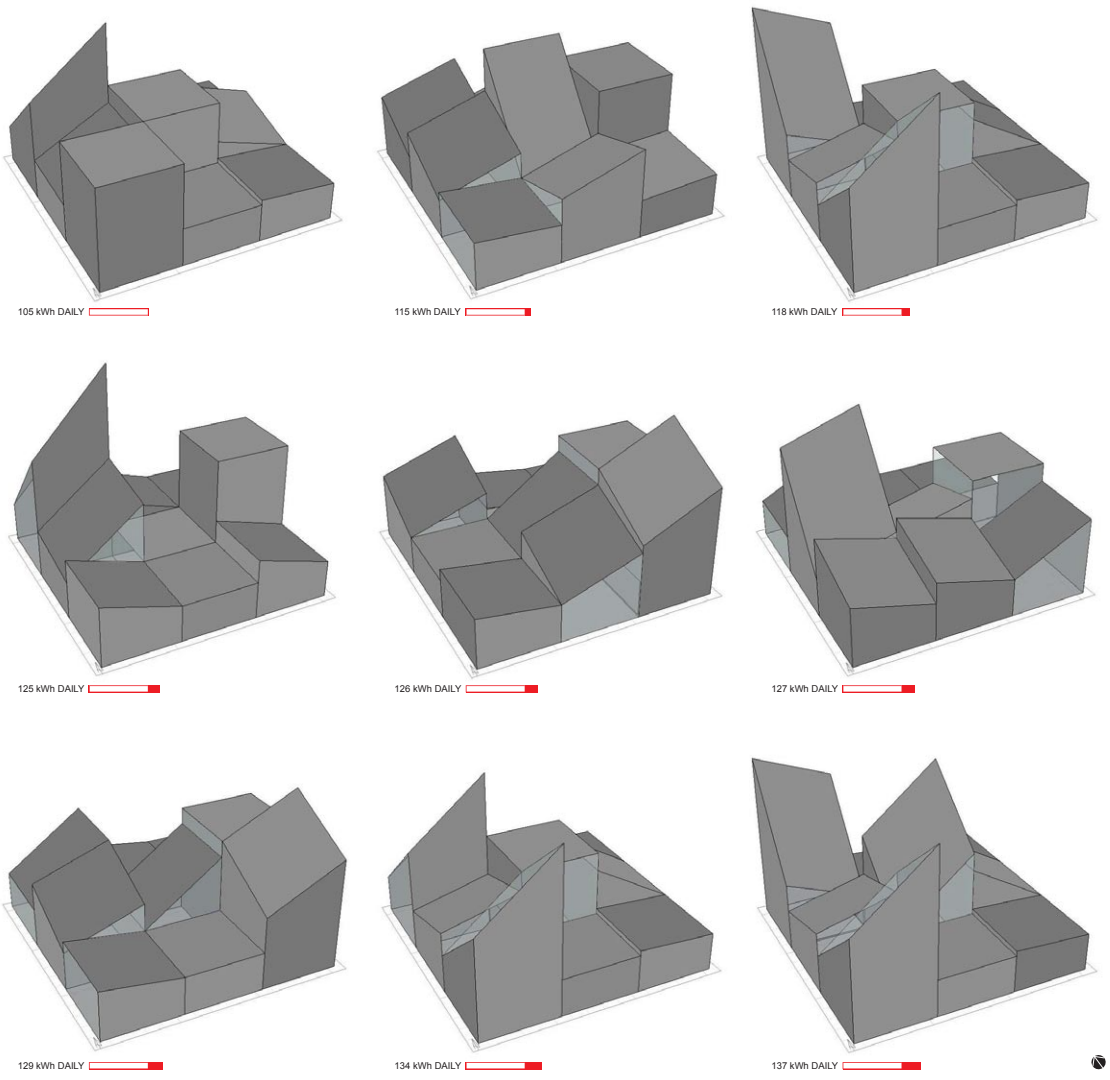

Figure A.21 Solutions from one run of the optimization algorithm, seen from the southwest.

Location: Ithaca

Algorithm: Deterministic Crowding

Period: Full Year

Trial: 3

APPENDIX B

FITNESS RESULTS

No discussion of a genetic algorithm is complete without graphs of the algorithm's performance. The graph of fitness results says a lot about the effectiveness of an algorithm. A curve that plateaus too early may have become stranded on a local optimum. One that does not improve quickly in the beginning may have poor linkage or lack fit building blocks. When the entire population bunches around the same fitness value, diversity is probably low.

This appendix presents fitness graphs from each trial. Generations from zero to forty-nine are listed sequentially on the horizontal axis, while fitness is displayed on the vertical axis in kWh per day during the controlling period. Each dot represents the fitness of one individual tested in the corresponding generation of an individual trial. A line connecting the each generation's best individual produces a fitness curve. The average fitness curve from a set of trials gives an indication of the algorithm's overall performance.

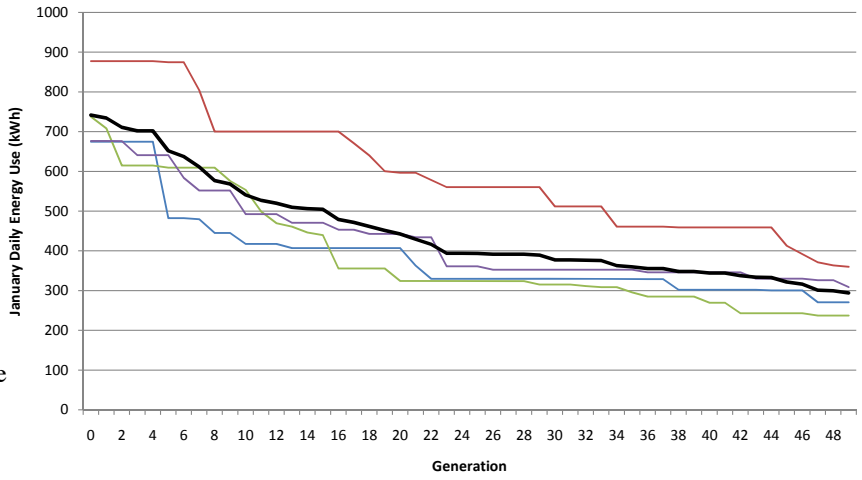

Figure B.1 Fitness graphs show the best individual in each trial, with the mean in bold.

Location: Anchorage

Algorithm: DC

Period: January

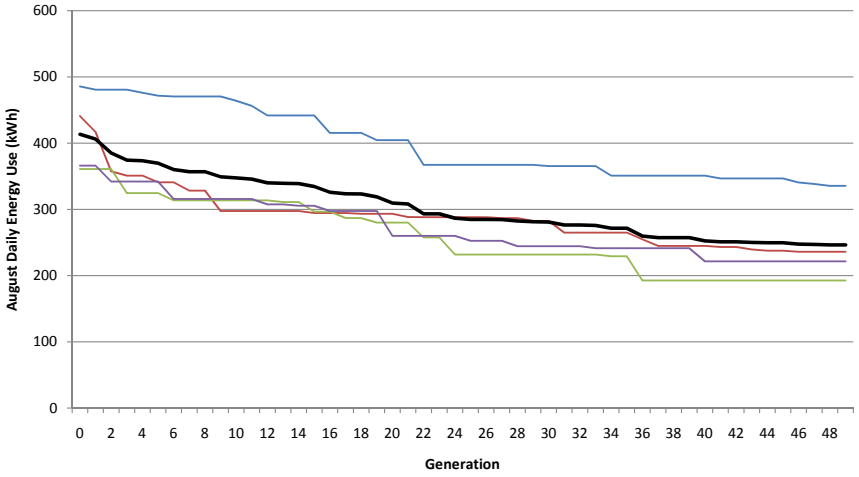

Figure B.2 Fitness graphs show the best individual in each trial, with the mean in bold.

Location: Dubai

Algorithm: DC

Period: August

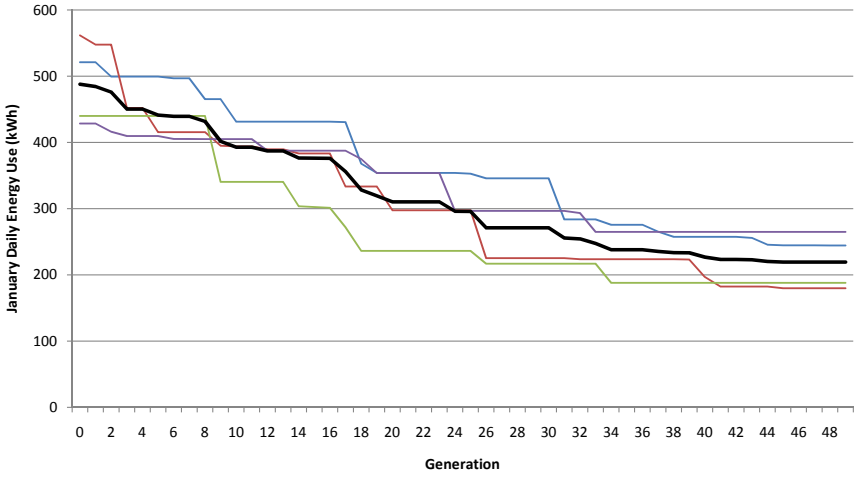

Figure B.3 Fitness graphs show the best individual in each trial, with the mean in bold.

Location: Ithaca

Algorithm: DC

Period: January

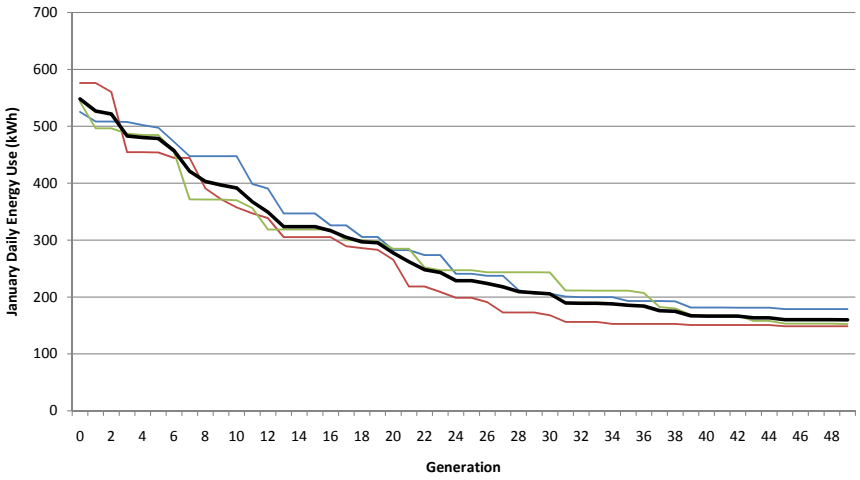

Figure B.4 Fitness graphs show the best individual in each trial, with the mean in bold.

Location: Ithaca

Algorithm: μ+λ

Period: January

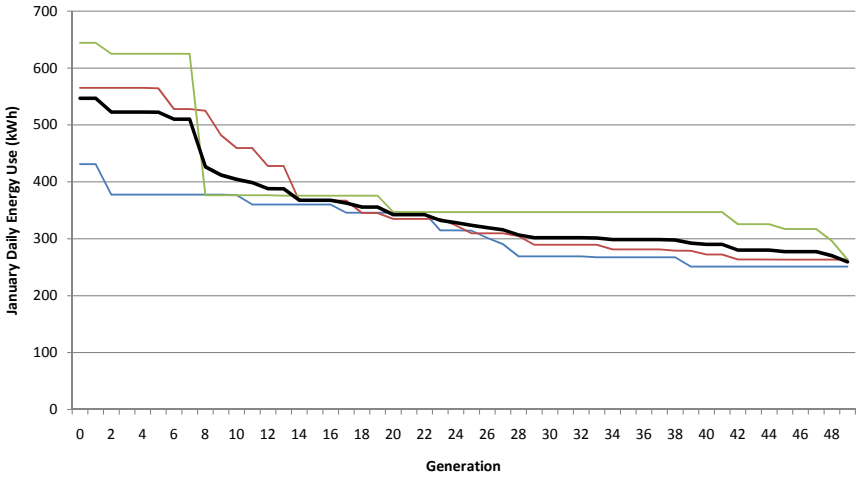

Figure B.5 Fitness graphs show the best individual in each trial, with the mean in bold.

Location: Ithaca

Algorithm: PHC

Period: January

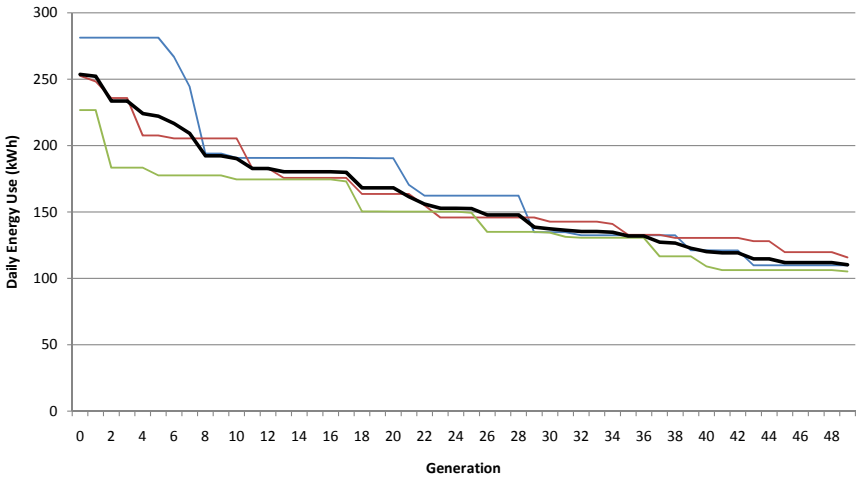

Figure B.6 Fitness graphs show the best individual in each trial, with the mean in bold.

Location: Ithaca

Algorithm: DC

Period: Full Year

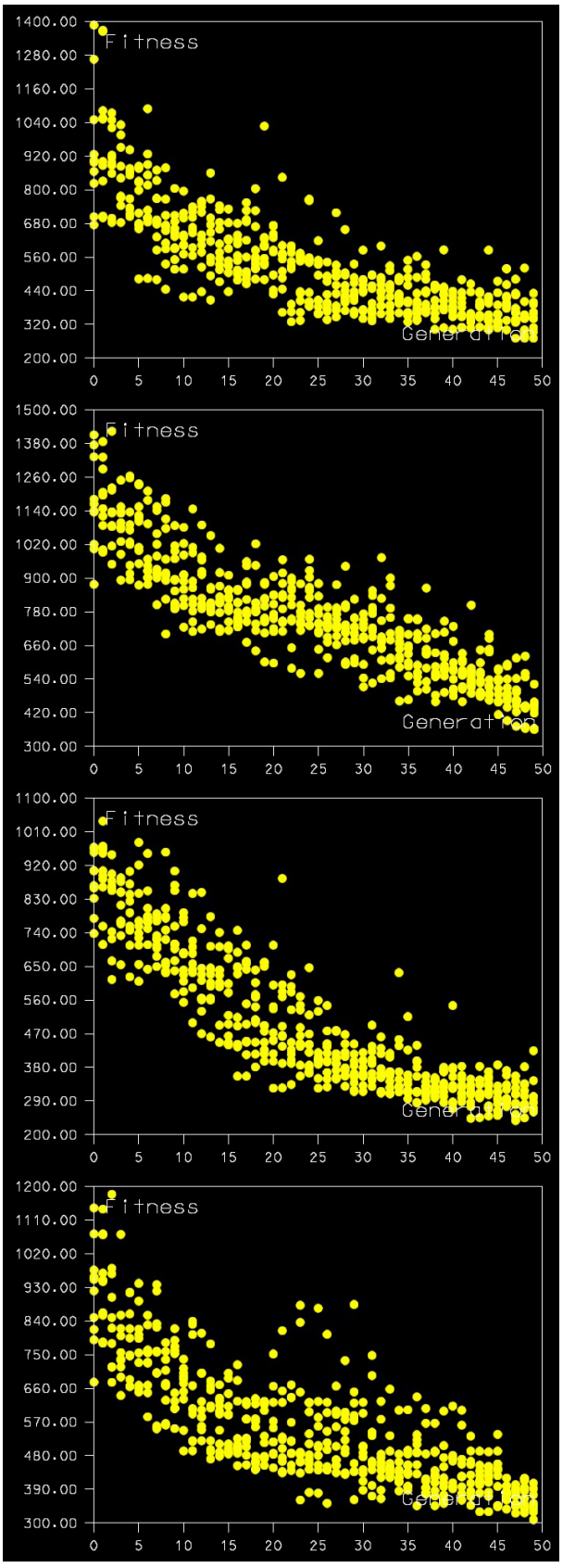

Figure B.7 Fitness plots show daily HVAC and lighting load in kWh on the vertical axis for each member tested in each generation.

Location: Anchorage

Algorithm: Deterministic Crowding

Period: January

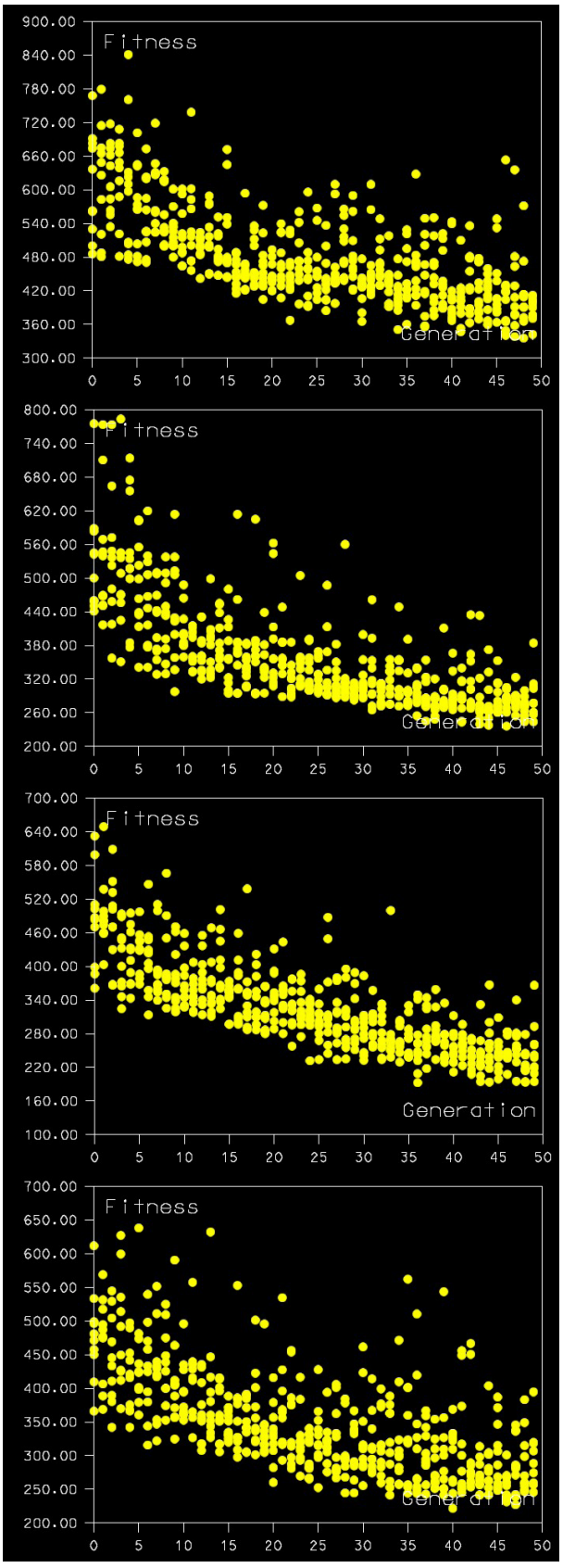

Figure B.8 Fitness plots show daily HVAC and lighting load in kWh on the vertical axis for each member tested in each generation.

Location: Dubai

Algorithm: Deterministic Crowding

Period: August

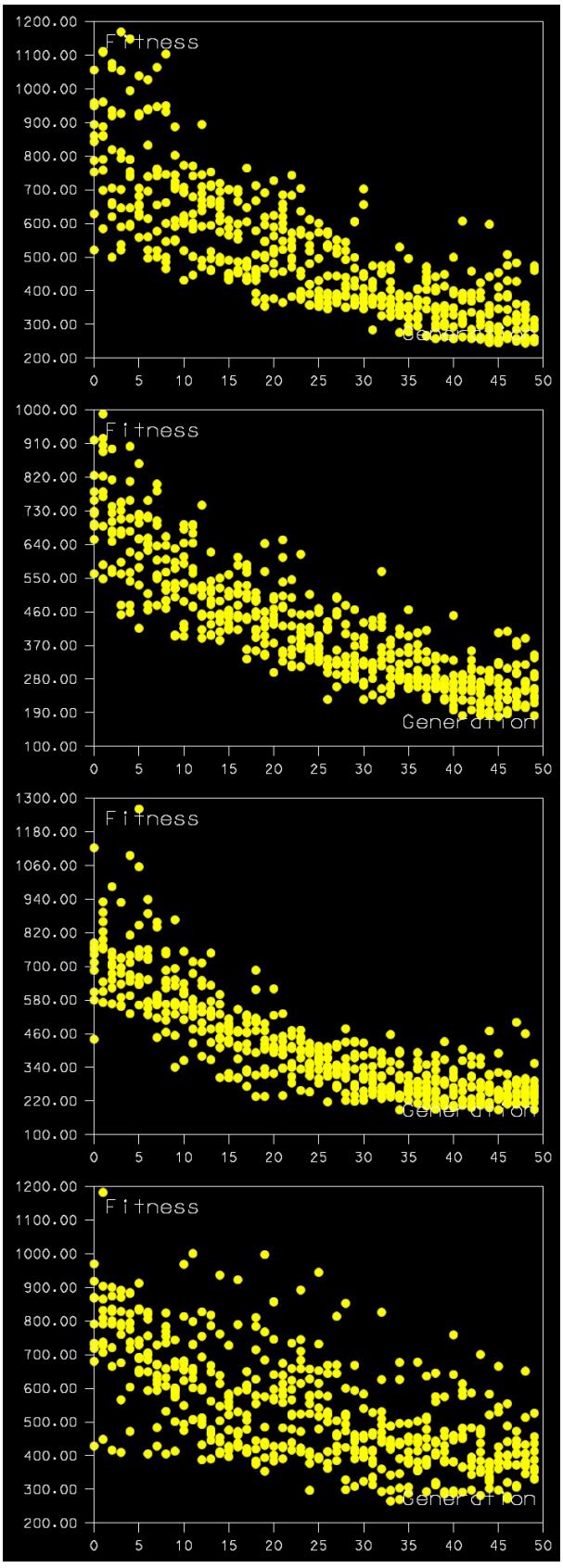

Figure B.9 Fitness plots show daily HVAC and lighting load in kWh on the vertical axis for each member tested in each generation.

Location: Ithaca

Algorithm: Deterministic Crowding

Period: January

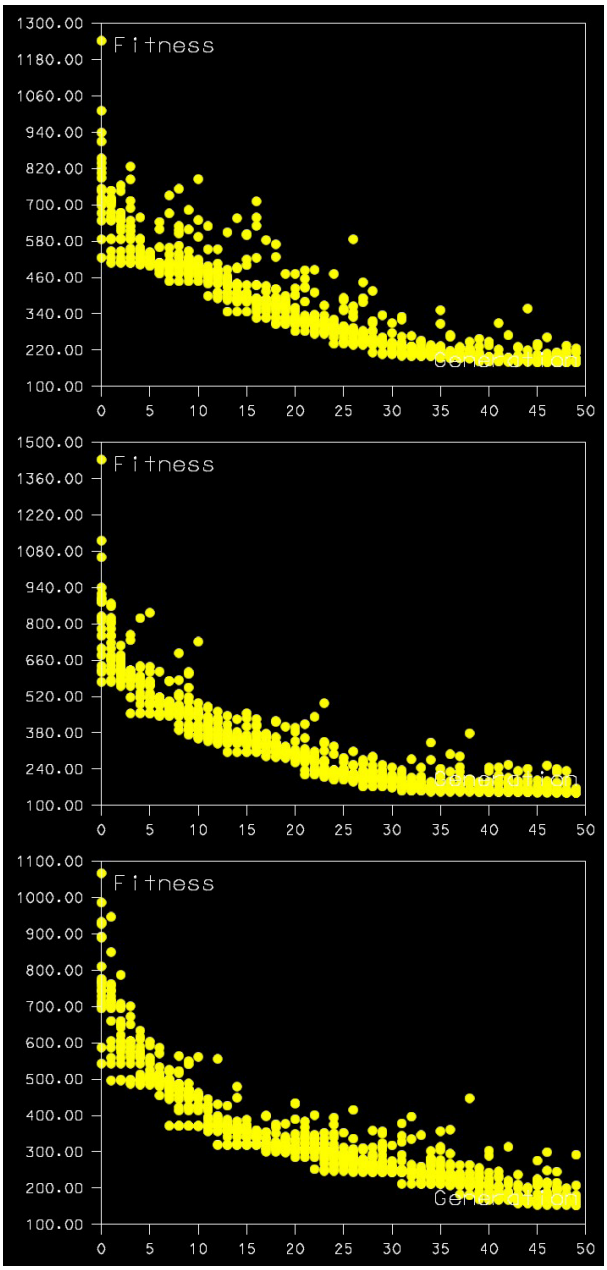

Figure B.10 Fitness plots show daily HVAC and lighting load in kWh on the vertical axis for each member tested in each generation.

Location: Ithaca

Algorithm: Mu+Lambda

Period: January

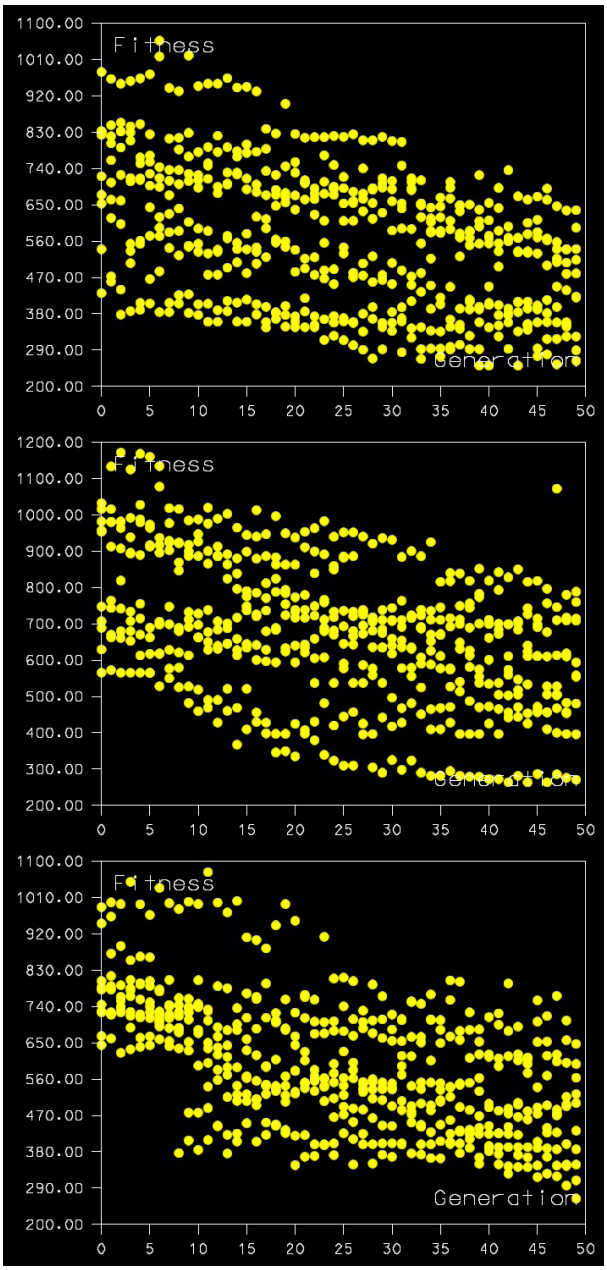

Figure B.11 Fitness plots show daily HVAC and lighting load in kWh on the vertical axis for each member tested in each generation.

Location: Ithaca

Algorithm: Parallel Hill-Climber

Period: January

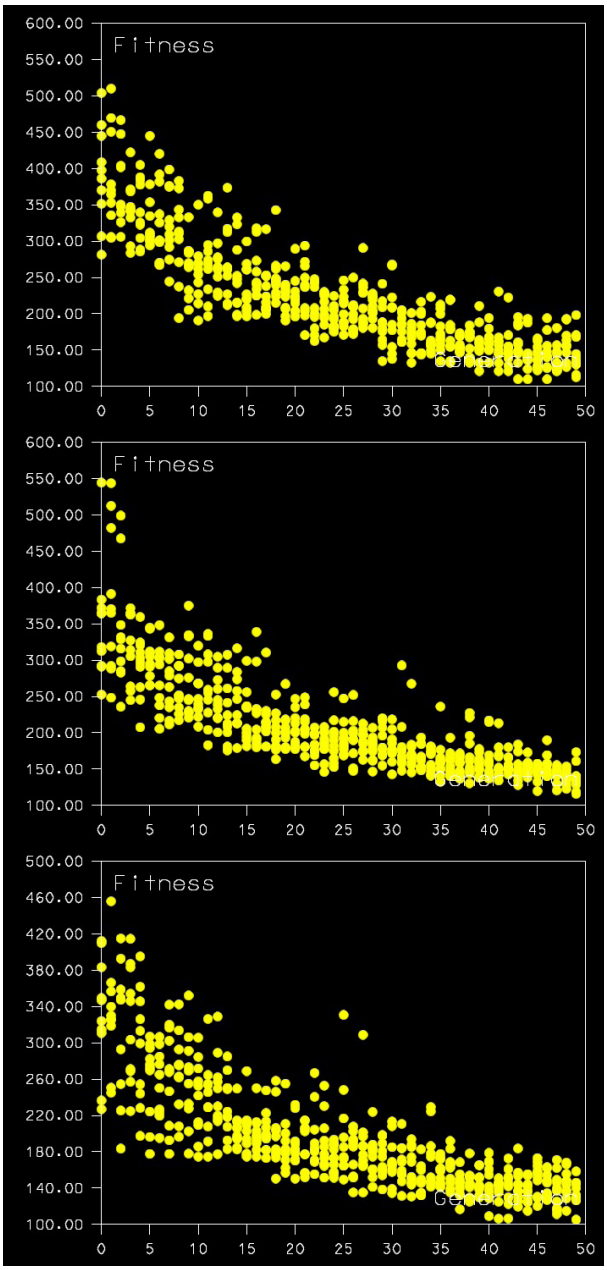

Figure B.12 Fitness plots show daily HVAC and lighting load in kWh on the vertical axis for each member tested in each generation.

Location: Ithaca

Algorithm: Deterministic Crowding

Period: Full Year

APPENDIX C

CODE

This appendix presents the GenerativeComponents (GC) transaction file containing the genetic algorithm used in this thesis. The transaction file was written in GC version 08.09.05.43 for MicroStation version 08.09.04.51. A dynamic-link library (DLL) by De Biswas called *sg2008EcoPlusDoc.dll* allows commands in this code to interact with Ecotect version 5.60.

GC records the process of model making rather than the model itself. Models are constructed through a series of transaction steps. The first transaction creates the variables that define a house.

```
transaction modelBased "Graph Variables for Geometry"
{
  feature DNA GC.GraphVariable
  {
    Value                     = FilledList3d(HouseXDiv, HouseYDiv,
                                HouseParams, 1/DNAStep);
  }
  feature DNAStep GC.GraphVariable
  {
    Value                     = 6;    // number of alleles for each gene
  }
  feature HouseEdge GC.GraphVariable
  {
    Value                     = 100;  // width of wall around voids, mm
  }
  feature HouseName GC.GraphVariable
  {
    Value                     = "FinalHouse";
  }
  feature HouseParams GC.GraphVariable
  {
    Value                     = 9;    // number of genes per zone
  }
  feature HouseSpacing GC.GraphVariable
  {
    Value                     = 2 * Max(HouseX * HouseXDiv,
                                HouseY * HouseYDiv);
                                // spacing between houses in population
  }
  feature HouseVoid GC.GraphVariable
  {
    Value                     = 1;    // threshold value
  }
```

```
  feature HouseX GC.GraphVariable
  {
    Value                     = 5000;   // width of zone, mm
  }
  feature HouseXDiv GC.GraphVariable
  {
    Value                     = 3;      // number of zones across
  }
  feature HouseY GC.GraphVariable
  {
    Value                     = 5000;   // length of zone, mm
  }
  feature HouseYDiv GC.GraphVariable
  {
    Value                     = 3;      // number of zones deep
  }
  feature HouseZ GC.GraphVariable
  {
    Value                     = 4000;   // average zone height, mm
  }
  feature HouseZMin GC.GraphVariable
  {
    Value                     = 2500;   // minimum zone height, mm
  }
  feature HouseZRot GC.GraphVariable
  {
    Value                     = 0;      // rotation angle of house
  }
}
```

The following transaction, unnecessary but helpful for debugging, creates a random genotype for a prototypical house.

```
transaction script "Dummy DNA", suppressed
{
  for (int i = 0; i < DNA.Count; i++)
  {
    for (int j = 0; j < DNA[i].Count; j++)
    {
      for (int k = 0; k < DNA[i][j].Count; k++)
      {
        DNA[i][j][k] = Random(DNAStep)/DNAStep;
      }
    }
  }
}
```

The next transaction contains the actual functions for making a zone and for assembling zones into a building. When run, it creates a prototypical house in GC. The prototype house is deleted before the genetic algorithm (GA) is run, but the instructions remain accessible for the rest of the transactions.

```
transaction modelBased "Create Proto Building"
{
  feature CSRot GC.CoordinateSystem
  {
    Origin                    = baseCS;
    CoordinateSystem          = baseCS;
    RotationAngle             = HouseZRot;
    Axis                      = AxisOption.Z;
    Visible                   = false;
  }
  feature ProtoBuilding GC.Polygon
  {
    Function                  = makeBuilding;
    FunctionArguments         = {CSRot, HouseX, HouseY, HouseZ, HouseXDiv,
                                 HouseYDiv, HouseZMin, DNAStep, DNA};
  }
  feature makeBuilding GC.GraphFunction
  {
    Definition                = Polygon [][][] function (CoordinateSystem cs,
                                 int x, int y, int z, int xDivs, int yDivs,
                                 int minZ, int mat, double[][][] dna)
    {
      // Find height factor necessary to keep volume constant
      double zSum = 0;
      for (int i = 0; i < dna.Count; i++)
      {
        for (int j = 0; j < dna[i].Count; j++)
        {
          zSum += dna[i][j][0];
        }
      }
      if (zSum == 0)                          // all zones have zero height
      {
        for (int i = 0; i < dna.Count; i++)
        {
          for (int j = 0; j < dna[i].Count; j++)
          {
            dna[i][j][0] = 1/DNAStep;
            zSum += dna[i][j][0];
          }
        }
      }
      double Zn = z * xDivs * yDivs / zSum;         // this is height factor

      // Create base grid for zones
      Point grid = new Point();
      grid.Replication = ReplicationOption.AllCombinations;
      grid.ByCartesianCoordinates(cs, Series(-x*xDivs/2, x*xDivs/2, x),
        Series(-y*yDivs/2, y*yDivs/2, y), 0);

      // Create individual zones
      Polygon [][][] bldg = FilledList3d(xDivs, yDivs, 6);
      for (int i = 0; i < bldg.Count; i++)
      {
        for (int j = 0; j < bldg[i].Count; j++)
        {
          bldg[i][j] = makeZone(cs, grid[i][j], grid[i+1][j], grid[i+1][j+1],
            grid[i][j+1], Zn, minZ, mat, dna[i][j]);
        }
      }
    };
  }
```

```
  feature makeZone GC.GraphFunction
  {
    Definition                = Polygon [] function (CoordinateSystem cs,
                                Point SW, Point SE, Point NE, Point NW,
                                double zn, int minZ, int mat, double[] gene)
    {
      // Make points around edge of roof
      double height = 2 * zn * gene[0];
      if (gene[0] == 0)
      {
        minZ = 0;
      }
      Point SWT = new Point().ByCartesianCoordinates(cs, SW.XTranslation,
        SW.YTranslation, Max(minZ, Min(height-minZ, height*gene[1])));
      Point SET = new Point().ByCartesianCoordinates(cs, SE.XTranslation,
        SE.YTranslation, Max(minZ, Min(height-minZ, height*gene[2])));
      Point NET = new Point().ByCartesianCoordinates(cs, NE.XTranslation,
        NE.YTranslation, Max(minZ, Min(height-minZ, height*(1-gene[1]))));
      Point NWT = new Point().ByCartesianCoordinates(cs, NW.XTranslation,
        NW.YTranslation, Max(minZ, Min(height-minZ, height*(1-gene[2]))));
      Polygon zone = {};

      // Roof
      zone[0] = new Polygon(this).ByVertices({SWT, SET, NET, NWT});
      this.SymbologyAndLevelUsage =
        SymbologyAndLevelUsageOption.AssignToFeature;
      zone[0].Color = gene[3] * mat + 1;

      // South Wall
      zone[1] = new Polygon(this).ByVertices({SW, SE, SET, SWT});
      this.SymbologyAndLevelUsage =
        SymbologyAndLevelUsageOption.AssignToFeature;
      zone[1].Color = gene[4] * mat + 1;

      // West Wall
      zone[2] = new Polygon(this).ByVertices({NW, SW, SWT, NWT});
      this.SymbologyAndLevelUsage =
        SymbologyAndLevelUsageOption.AssignToFeature;
      zone[2].Color = gene[5] * mat + 1;

      // North Wall
      zone[3] = new Polygon(this).ByVertices({NE, NW, NWT, NET});
      this.SymbologyAndLevelUsage =
        SymbologyAndLevelUsageOption.AssignToFeature;
      zone[3].Color = gene[6] * mat + 1;

      // East Wall
      zone[4] = new Polygon(this).ByVertices({SE, NE, NET, SET});
      this.SymbologyAndLevelUsage =
        SymbologyAndLevelUsageOption.AssignToFeature;
      zone[4].Color = gene[7] * mat + 1;

      // Floor
      zone[5] = new Polygon(this).ByVertices({SW, NW, NE, SE});
    };
  }
}
```

Having tested the code, the prototype house is now deleted.

```
transaction modelBased "Delete Proto Building"
{
  deleteFeature CSRot;
  deleteFeature ProtoBuilding;
}
```

The next set of transactions establishes the testing environment in Ecotect. First, the variables are defined.

```
transaction modelBased "Graph Variables for Analysis"
{
  feature ZoneAdjacency GC.GraphVariable
  {
    Value                     = 1600;   // Accuracy of Ecotect calculation
    LimitValueToRange         = true;
    RangeMinimum              = 100;
    RangeMaximum              = 2500;
    RangeStepSize             = 50;
  }
  feature ZoneAirSpeed GC.GraphVariable
  {
    Value                     = 0.05;   // Air speed, m/s
    LimitValueToRange         = true;
    RangeMaximum              = 0.1;
    RangeStepSize             = 0.01;
  }
  feature ZoneClimate GC.GraphVariable
  {
    Value                     = "Ithaca"; // Name of .wea climate data file
  }
  feature ZoneClothing GC.GraphVariable
  {
    Value                     = 1.0;    // Clothing worn by occupants, clo
    LimitValueToRange         = true;
    RangeMaximum              = 4.0;
    RangeStepSize             = 0.25;
  }
  feature ZoneDataFreq GC.GraphVariable
  {
    Value                     = 1;      // Rate of data collection, days
    LimitValueToRange         = true;
    RangeMinimum              = 1;
    RangeMaximum              = 30;
    RangeStepSize             = 1;
  }
  feature ZoneDayEnd GC.GraphVariable
  {
    Value                     = 30;     // Last day of data collection
    LimitValueToRange         = true;
    RangeMinimum              = ZoneDayStart;
    RangeMaximum              = 365;
    RangeStepSize             = 1;
  }
```

```
feature ZoneDayStart GC.GraphVariable
{
  Value                     = 1;      // First day of data collection
  LimitValueToRange         = true;
  RangeMinimum              = 1;
  RangeMaximum              = 365;
  RangeStepSize             = 1;
}
feature ZoneEfficacy GC.GraphVariable
{
  Value                     = 13;     // Efficacy of electric lights, L/W
  LimitValueToRange         = true;
  RangeMaximum              = 683;
}
feature ZoneGrid GC.GraphVariable
{
  Value                     = 3;      // resolution of daylight grid
  LimitValueToRange         = true;
  RangeMinimum              = 1;
  RangeMaximum              = 5;
  RangeStepSize             = 1;
}
feature ZoneHourEnd GC.GraphVariable
{
  Value                     = 20;     // Last hour of daylight sampling
  LimitValueToRange         = true;
  RangeMinimum              = ZoneHourStart;
  RangeMaximum              = 23;
  RangeStepSize             = 1;
}
feature ZoneHourStart GC.GraphVariable
{
  Value                     = 6;      // First hour of daylight sampling
  LimitValueToRange         = true;
  RangeMaximum              = 22;
  RangeStepSize             = 1;
}
feature ZoneHumidity GC.GraphVariable
{
  Value                     = 50;     // Relative humidity, %
  LimitValueToRange         = true;
  RangeMinimum              = 30;
  RangeMaximum              = 70;
  RangeStepSize             = 5;
}
feature ZoneLights GC.GraphVariable
{
  Value                     = 538;    // Light level, Lux
  LimitValueToRange         = true;
  RangeMaximum              = 1000;
}
feature ZoneOccupancy GC.GraphVariable
{
  Value                     = 2;      // Number of occupants per zone
  LimitValueToRange         = true;
  RangeMaximum              = 10;
  RangeStepSize             = 1;
}
feature ZoneSunEfficacy GC.GraphVariable
{
  Value                     = 93;     // Efficacy of sun, L/W
}
```

```
  feature ZoneTempMax GC.GraphVariable
  {
    Value                     = 24.44;  // Top of comfort band, °C
    LimitValueToRange         = true;
    RangeMinimum              = ZoneTempMin;
    RangeMaximum              = 28;
  }
  feature ZoneTempMin GC.GraphVariable
  {
    Value                     = 22.22;  // Bottom of comfort band, °C
    LimitValueToRange         = true;
    RangeMinimum              = 18;
    RangeMaximum              = 28;
  }
}
```

Some functions that interact with Ecotect require a text style. The style defined here is used in calling those functions, though it never appears on screen.

```
transaction modelBased "Add Text Style"
{
  feature text GC.TextStyle
  {
    Height                    = 1000;
    Width                     = 1000;
    JustificationSingleLine   = JustificationSingleLine.CenterCenter;
    TextColor                 = 7;
  }
}
```

The scripts in the following transaction prepare GC to set up zones in Ecotect. The first two scripts are not run immediately, but are used later to create lighting and operational schedules for each zone and to delete any models that might already be in Ecotect. A third script names each zone with the coordinates of its position. The final feature initializes the zones in Ecotect, although at this point the zones do not contain any objects.

```
transaction modelBased "Add Ecotect Zones"
{
  feature AddSchedule GC.GraphFunction
  {
    Definition                = int function (string name, int start,
                                int end)
    {
      string driver = "";
      Ecotect command = new Ecotect();  // send commands with this object

      // Get index of current schedule
      command.Request("get.schedule.index " + name, driver);
```

```
      int index = ToInt(command.Result);
      if (index < 0)
      {
        command.Request("add.schedule " + name, driver);
                                        // makes two schedules with same name
        index = ToInt(command.Result);
        command.Execute("schedule.delete " + index, driver);
                                        // delete extra schedule
        index--;                        // index of first schedule
      }

      // Set up schedule with default hours of opperation
      for (int hour = start; hour <= end; hour++)
      {
        command.Execute("set.schedule.activation " + index + " 0 " + hour +
          " 1", driver);
      }
      return index;
    };
  }
  feature ClearEcotect GC.GraphFunction
  {
    Definition                = function ()
    {
      string driver = "";
      Ecotect command = new Ecotect();  // send commands with this object

      // Remove objects already in Ecotect model
      command.Request("get.model.objects", driver);
      int obj = ToInt(command.Result);
      while (obj > 0)
      {
        obj--;
        command.Execute("object.delete " + obj, driver);
      }
    };
  }
  feature ZoneNames GC.GraphFunction
  {
    Definition                = function ()
    {
      string names = {};
      int index = 0;
      for (int i = 0; i < HouseXDiv; i++)
      {
        for (int j = 0; j < HouseYDiv; j++)
        {
          names[index++] = "Room_" + i + "_" + j;
        }
      }
      return names;
    };
  }
  feature Zones GC.EcotectZone
  {
    ZoneName                  = ZoneNames();
    Thermal                   = {true};
    Clothing                  = {ZoneClothing};
    AirVelocity_m_per_sec     = {ZoneAirSpeed};
    RelativeHumidity          = {ZoneHumidity};
    Lighting                  = {ZoneLights};
  }
}
```

The next script sends information about the site and zones to Ecotect. This information includes where to find the climate data, a list of materials and their properties, the operation schedules and other settings for each zone, and the camera position for capturing images of each design variant. The script also retrieves information on the amount of daylight available throughout the year for later use when calculating electric lighting requirements.

```
transaction script "Set Up Ecotect"
{
  string driver = "";
  Ecotect command = new Ecotect();  // send commands with this object

  command.Execute("script.start", driver);
  command.Execute("app.minimise true", driver);

  // Remove objects already in Ecotect model
  ClearEcotect();

  // Set up current project
  command.Execute("set.project.title " + HouseName, driver);  // Project name
  command.Execute("set.project.type 0", driver);      // Domestic dwelling
  command.Execute("weather.load.all " + CurrentDirectory() + "Climate Data\\"
    + ZoneClimate + ".wea", driver);   // Weather File

  // set up materials
  int mat;
  bool window;
  for (int i = 0; i < DNAStep; i++)
  {
    mat = i + 51;
    window = (i < DNAStep/2)? true : false;     // lower numbers are windows
    command.Execute("set.material.name " + mat + " Material" + i, driver);
                  // change name of a material (currently listed as a panel)

    // Thermal properties
    command.Execute("set.material.uvalue " + mat + " " +
      Pow(36, 0.5 - i/DNAStep), driver);        // set U-value (W/m^2-K)
    command.Execute("set.material.admittance " + mat + " " +
      Pow(36, 0.5 - i/DNAStep), driver);        // set admittance (W/m^2-K)
    command.Execute("set.material.absorption " + mat + " " +
      (window? 1-(i+0.5)/DNAStep : i/DNAStep), driver);
    // set solar absorbtion for wall, solar heat gain coef. for window (0-1)
    command.Execute("set.material.transparency " + mat + " " +
      (window? 1-(i+0.5)/DNAStep : 0), driver);
                                          // set visible transmittance (0-1)
    command.Execute("set.material.decrement " + mat + " " +
      (window? 1.74 : 1-i/DNAStep), driver);
        // set thermal decrement for wall (0-1), refractive index for window
    command.Execute("set.material.lag " + mat + " " + (window? 0.3 : i),
      driver);
        // set thermal lag for wall (hours), alternate solar gain for window
```

```
  // Set everything else equal
  command.Execute("set.material.colour " + mat +
    (window? " 0xC1C1B0 0xC1C1B0" : " 0x919191 0x919191"), driver);
      // change name of a material (currently listed as a panel)
  command.Execute("set.material.extemissivity " + mat + " " +
    (window? i/DNAStep : 0.9), driver);                          // (0-1)
  command.Execute("set.material.extroughness " + mat + " 0", driver);
                                                                 // (0-1)
  command.Execute("set.material.extspecularity " + mat + " 0", driver);
                                                                 // (0-1)
  command.Execute("set.material.intemissivity " + mat + " " +
    (window? i/DNAStep : 0.9), driver);                          // (0-1)
  command.Execute("set.material.introughness " + mat + " 0", driver);
                                                                 // (0-1)
  command.Execute("set.material.intspecularity " + mat + " 0", driver);
                                                                 // (0-1)
  command.Execute("set.material.reflectance " + mat + " " +
    (window? 0.7 - i/DNAStep/10 : 0.5 - i/DNAStep/10), driver);
                    // brightens or darkens color that was set above(0-1)
  command.Execute("set.material.type " + mat + " " + (window? 6 : 4),
    driver);                         // set material type to window or wall
}

// Set up schedules
int op = AddSchedule("Opperation", 0, 23);
int lit = {};
for (int i = 0; i < HouseXDiv; i++)
{
  for (int j = 0; j < HouseYDiv; j++)
  {
    lit[i*HouseYDiv+j] = AddSchedule("Lighting_" + i + "_" + j,
      ZoneHourStart, ZoneHourEnd);
  }
}

// Set up daylight calculation properties
command.Execute("set.attribute.scale 0 10", driver);
                            // set color scale from 0-10% daylight factor
command.Execute("set.calc.precision high", driver);       // accuracy
command.Execute("set.calc.sky overcast", driver);         // sky condition
command.Execute("set.calc.windows 1", driver);
                                     // 90% transmission through windows
double availableLux = FilledList2d(ZoneDayEnd - ZoneDayStart + 1, 24);
double beam, diffuse;
for (int day = ZoneDayStart; day <= ZoneDayEnd; day++)
{
  for (int hour = 0; hour < 24; hour++)
  {
    command.Request("get.weather.beamsolar " + day + " " + hour, driver);
                         // Get available direct solar radiation (W/m^2)
    beam = ToDouble(command.Result);
    command.Request("get.weather.diffusesolar " + day + " " + hour,
      driver);            // Get available diffuse solar radiation (W/m^2)
    diffuse = ToDouble(command.Result);
    availableLux[day-ZoneDayStart][hour] = ZoneSunEfficacy *
      (beam + diffuse);
  }
}
GraphVariable AL = new
  GraphVariable("AvailableLux").EvaluateExpression(availableLux);
```

```
  // Set up each zone
  command.Execute("calc.adjacencies " + ZoneAdjacency + " true", driver);
    // calculate adjacencies, fewer samples, include shading mask
  for (int i = 1; i <= HouseXDiv * HouseYDiv; i++)
  {
    command.Execute("set.zone.airspeed " + i + " " + ZoneAirSpeed, driver);
                                                    // Air Velocity, m/s
    command.Execute("set.zone.clothing " + i + " " + ZoneClothing, driver);
                                                    // Clothing, clo
    command.Execute("set.zone.lux " + i + " " + ZoneLights, driver);
                                                    // Lighting, lux
    command.Execute("set.zone.occupancy " + i + " " + ZoneOccupancy, driver);
                                                    // People, #
    command.Execute("set.zone.lowerband " + i + " " + ZoneTempMin, driver);
                                                  // Temperature, 72 F = 22.22 C
    command.Execute("set.zone.upperband " + i + " " + ZoneTempMax, driver);
                                                  // Temperature, 76 F = 24.44 C
    command.Execute("set.zone.relhumidity " + i + " " + ZoneHumidity,
      driver);                                    // Relative Humidity, %
    command.Execute("set.zone.thermal " + i + " true", driver);
                                                  // Zone is thermal
    command.Execute("set.zone.system " + i + " airconditioning", driver);
                                                  // Zone is fully airconditioned
    command.Execute("set.zone.sensiblegains " + i + " " +
      ZoneLights/ZoneEfficacy, driver);           // Heat gains from lights
    command.Execute("set.zone.schedules " + i + " " + op + " " + op + " " +
      lit[i-1], driver);      // Occupancy, Ventilation, and Gains Schedules
  }

  // Set up openGL
  command.Execute("set.app.page 2", driver);        // bring up openGL frame
  command.Execute("set.opengl.grid.max " + (HouseX*HouseXDiv/2+1500) + " " +
    (HouseY*HouseYDiv/2+1500) + " 0", driver);      // set display grid max
  command.Execute("set.opengl.grid.min " + (-HouseX*HouseXDiv/2-1500) + " " +
    (-HouseY*HouseYDiv/2-1500) + " 0", driver);     // set display grid min
  command.Execute("set.opengl.eyept 20000 -40000 30000", driver);
                                                    // set camera position
  command.Execute("set.opengl.target 0 0 6000", driver);  // aim camera
  command.Execute("set.opengl.nearclip 10000", driver);
                                                    // near clipping plane
  command.Execute("set.opengl.farclip 80000", driver);
                                                    // far clipping plane

  command.Execute("script.end", driver);
}
```

The next transaction contains three scripts that will be used later in the algorithm. The first references an earlier transaction to create a house in GC and copies that house into Ecotect by sending commands through the DLL. The second instructs Ecotect to calculate daylight factors in each zone. The final script tells Ecotect to calculate thermal loads for each day. It combines the loads with the daylight factor information to calculate the house's average daily energy requirement, which will serve later as the measure of fitness.

```
transaction modelBased "Evaluation Functions"
{
  feature ExportBldg GC.GraphFunction
  {
    Definition                = function (string name, CoordinateSystem cs,
                                double[][][] dna)
    {
      string driver = "";
      Ecotect command = new Ecotect();    // send commands with this object

      // Remove objects already in Ecotect model
      ClearEcotect();

      // Construct house
      Polygon bldg = new Polygon(name).ByFunction(makeBuilding, {cs, HouseX,
        HouseY, HouseZ, HouseXDiv, HouseYDiv, HouseZMin, DNAStep, dna});

      // Export house to Ecotect
      command.Execute("set.app.page 1", driver);  // bring up 3D editor frame
      Polygon room, opening;
      Point VTL, VTR;
      int zone = 1;
      for (int i = 0; i < HouseXDiv; i++)
      {
        for (int j = 0; j < HouseYDiv; j++)
        {
          if (dna[i][j][0] > 0)
          {
            command.Execute("set.zone.off " + zone + " false", driver);
                                                        // Zone is thermal
            room = Sublist(bldg, 6*(i*HouseYDiv + j), 6);
            EcotectPolygonObject eco = new EcotectPolygonObject();
            eco.EcotectObjectFromPolygonList(room, zone,
              EcotectMaterial.ConcSlab_OnGround, false, text);
            for (int k = 0; k < room.Count; k++)
            {
              if (room[k].Color > 0)
              {
                command.Execute("set.object.type " + eco.ObjectID[k] + " " +
                  ((room[k].Color-1 < DNAStep/2)? 6 : 4), driver);
                                  // set object type as window or wall
                command.Execute("set.object.material " + eco.ObjectID[k] +
                  " Material" + (room[k].Color-1), driver);
                                  // set material as window or wall
                command.Execute("set.object.alternate " + eco.ObjectID[k] +
                  " Material" + Ceiling(DNAStep/2), driver);
                                  // alternate material for partitions
              }
              else
              {
                command.Execute("set.object.type " + eco.ObjectID[k] + " 2",
                  driver);              // set object type as floor;
              }
            }

            // Add voids
            if (i < HouseXDiv-1)
            {
              if (dna[i][j][8] + dna[i+1][j][8] > HouseVoid &&
                dna[i+1][j][0] > 0)
              {
                VTL = new Point().ByCartesianCoordinates(cs,
                  room[4].Vertices[0].XTranslation,
```

```
              room[4].Vertices[0].YTranslation, HouseZMin);
            VTR = new Point().ByCartesianCoordinates(cs,
              room[4].Vertices[1].XTranslation,
              room[4].Vertices[1].YTranslation, HouseZMin);
            opening = new Polygon().ByVertices({room[4].Vertices[0],
              room[4].Vertices[1], VTR, VTL});
            EcotectPolygonObject voi = new EcotectPolygonObject();
            voi.EcotectObjectFromPolygonList({opening}, zone,
              EcotectMaterial.Void, false, text);
            command.Execute("set.object.type " + voi.ObjectID[0] + " 0",
              driver);                          // set object type as void;
            command.Execute("object.link " + eco.ObjectID[4] + " " +
              voi.ObjectID[0], driver);
                              // set object to be child of another object
            command.Execute("set.object.child.extents "+voi.ObjectID[0]+
              " "+HouseEdge+" "+HouseEdge+" "+(HouseX-2*HouseEdge)+
              " "+(HouseZMin-2*HouseEdge), driver);
          }
        }
        if (j < HouseYDiv-1)
        {
          if (dna[i][j][8] + dna[i][j+1][8] > HouseVoid &&
            dna[i][j+1][0] > 0)
          {
            VTL = new Point().ByCartesianCoordinates(cs,
              room[3].Vertices[0].XTranslation,
              room[3].Vertices[0].YTranslation, HouseZMin);
            VTR = new Point().ByCartesianCoordinates(cs,
              room[3].Vertices[1].XTranslation,
              room[3].Vertices[1].YTranslation, HouseZMin);
            opening = new Polygon().ByVertices({room[3].Vertices[1],
              room[3].Vertices[0], VTL, VTR});
            EcotectPolygonObject voi = new EcotectPolygonObject();
            voi.EcotectObjectFromPolygonList({opening}, zone,
              EcotectMaterial.Void, false, text);
            command.Execute("set.object.type " + voi.ObjectID[0] + " 0",
              driver);                          // set object type as void;
            command.Execute("object.link " + eco.ObjectID[3] + " " +
              voi.ObjectID[0], driver);
                              // set object to be child of another object
            command.Execute("set.object.child.extents "+voi.ObjectID[0]+
              " "+HouseEdge+" "+HouseEdge+" "+(HouseY-2*HouseEdge)+
              " "+(HouseZMin-2*HouseEdge), driver);
          }
        }
      }
      else                                  // Zone is not thermal
      {
        command.Execute("set.zone.off " + zone + " true", driver);
      }
      zone++;
    }
  }
  command.Execute("view.fitgrid", driver);
  command.Execute("calc.volumes", driver);  // recalculate volume of zone
 };
}
feature TestDaylight GC.GraphFunction
{
 Definition               = double [][] function (CoordinateSystem cs)
 {
  string driver = "";
  Ecotect command = new Ecotect();  // send commands with this object
```

```
    // Create analysis grid
    double Xs = Series(-HouseX*HouseXDiv/2 + HouseX/ZoneGrid/2,
      HouseX*HouseXDiv/2 - HouseX/ZoneGrid/2, HouseX/ZoneGrid);
    double Ys = Series(-HouseY*HouseYDiv/2 + HouseY/ZoneGrid/2,
      HouseY*HouseYDiv/2 - HouseY/ZoneGrid/2, HouseY/ZoneGrid);
    Point grid = new Point();
    grid.Replication = ReplicationOption.AllCombinations;
    grid.ByCartesianCoordinates(cs, Xs, Ys, 1000);
    EcotectPointObject daylightGrid = new
      EcotectPointObject().EcotectObjectFromPointGrid(grid, 0,
      EcotectMaterial.ConstructionLine, false, text);

    // Perform calculation in Ecotect
    command.Execute("calc.lighting points daylight false", driver);
                                // Calculate daylight at all point objects

    // Read data from each gridpoint
    int obj, off;
    int points = Pow(ZoneGrid, 2);
    double [][] daylightFactor = FilledList2d(ZoneGrid, ZoneGrid, 0);
    int zone = 1;
    for (int i = 0; i < HouseXDiv; i++)
    {
      for (int j = 0; j < HouseYDiv; j++)
      {
        command.Request("get.zone.off " + zone, driver);
        off = ToInt(command.Result);
        if (off < 1)    // zone is on
        {
          for (int m = 0; m < ZoneGrid; m++)
          {
            for (int n = 0; n < ZoneGrid; n++)
            {
              obj = daylightGrid.ObjectID[i*ZoneGrid+m][j*ZoneGrid+n];
              command.Request("get.object.attr1 " + obj, driver);
                                                  // retrieve daylight factor
              daylightFactor[i][j] += ToDouble(command.Result) / 100;
            }
          }
          daylightFactor[i][j] /= points;   // mean daylight factor in zone
        }
        zone++;
      }
    }
    return daylightFactor;
  };
}
feature TestLoads GC.GraphFunction
{
  Definition                = double function (double [][] df)
  {
    string driver = "";
    Ecotect command = new Ecotect();  // send commands with this object

    double avgLoads = 0;          // average daily electricity consuption
    double dailySum, lightDeficit;
    int calcDays = 0;
    int schedule;
    for (int day = ZoneDayStart; day <= ZoneDayEnd; day += ZoneDataFreq)
    {
      dailySum = 0;
      command.Execute("set.model.dayoftheyear " + day, driver);
```

```
        // Set up daily lighting schedule
        for (int i = 0; i < HouseXDiv; i++)
        {
          for (int j = 0; j < HouseYDiv; j++)
          {
            command.Request("get.schedule.index Lighting_" + i + "_" + j,
              driver);
            schedule = ToInt(command.Result);
            for (int hour = ZoneHourStart; hour <= ZoneHourEnd; hour++)
            {
              lightDeficit = Max(1 - AvailableLux[day-ZoneDayStart][hour] *
                df[i][j] / ZoneLights, 0);
                            // normalized value: maximum artificial lighting
              command.Execute("set.schedule.activation " + schedule + " 0 " +
                hour + " " + lightDeficit, driver);
              dailySum += lightDeficit * ZoneLights / ZoneEfficacy * HouseX *
                HouseY / 1000000;
                  // contribution from electric lighting each hour, in watts
            }
          }
        }

        // Calculate gains
        command.Execute("calc.thermal.gains", driver);
        if (day == ZoneDayStart) command.Execute("set.app.page 3", driver);
                                                  // bring up analysis frame
        for (int hour = ZoneHourStart; hour <= ZoneHourEnd; hour++)
        {
          command.Request("get.results.array 0 " + hour, driver);
          dailySum += Abs(ToDouble(command.Result));
                          // contribution from HVAC system, in watt-hours
        }
        avgLoads += dailySum;
        calcDays++;
      }
      return avgLoads / calcDays / 1000;          // kWh into building daily
    };
  }
}
```

The next transaction recreates the prototype house and evaluates it in Ecotect. This is not a necessary step, but it is useful in debugging.

```
transaction script "Transfer Test", suppressed
{
  string driver = "";
  Ecotect command = new Ecotect();          // send commands with this object

  command.Execute("script.start", driver);
  command.Execute("app.activate", driver);
  command.Execute("app.minimise false", driver);

  CoordinateSystem cs = new
    CoordinateSystem("CS").ByOriginRotationAboutCoordinateSystem(baseCS,
    baseCS, HouseZRot, AxisOption.Z);
  cs.Visible = false;
  ExportBldg(HouseName, cs, DNA);             // make house in GC and Ecotect
  double [][] daylightFactor = TestDaylight(cs);  // calcuate dayligh factor
  double load = TestLoads(daylightFactor);              // calculate fitness
```

```
  Print("Load: " + load + " kWh");

  command.Execute("script.end", driver);
}
```

The next series of transactions runs the GA. First, variables are defined.

```
transaction modelBased "Graph Variables for Genetic Algorithm"
{
  feature ActivateEcotect GC.GraphVariable
  {
    Value                     = true;   // Controls visibility of Ecotect
  }
  feature DetCrowding GC.GraphVariable
  {
    Value                     = true;   // Chooses DC or Mu+Lambda
  }
  feature DisplayAll GC.GraphVariable
  {
    Value                     = true;   // Saves all tested individuals
  }
  feature ExportData GC.GraphVariable
  {
    Value                     = true;   // Saves data in a CSV file
  }
  feature GAGenerations GC.GraphVariable
  {
    Value                     = 50;     // Number of generations
  }
  feature GAKeepRate GC.GraphVariable
  {
    Value                     = 0.5;    // Fraction of population to keep
    LimitValueToRange         = true;
    RangeMaximum              = 1;
  }
  feature GAMaxFit GC.GraphVariable
  {
    Value                     = 2000;   // Initial extent of graph
    LimitValueToRange         = true;
    RangeMinimum              = 100;
    RangeMaximum              = 10000;
    RangeStepSize             = 100;
  }
  feature GAMaxVar GC.GraphVariable
  {
    Value                     = 1.0;    // Initial extent of graph
    LimitValueToRange         = true;
    RangeMinimum              = 0.1;
    RangeMaximum              = 1.0;
    RangeStepSize             = 0.1;
  }
  feature GAMinFit GC.GraphVariable
  {
    Value                     = 1000;   // Initial extent of graph
    LimitValueToRange         = true;
    RangeMinimum              = 0;
    RangeMaximum              = GAMaxFit - 100;
    RangeStepSize             = 100;
  }
```

```
  feature GAMutCopyProb GC.GraphVariable
  {
    Value                      = 0.1;    // Probability of copying a zone
    LimitValueToRange          = true;
    RangeMaximum               = 1 - GAMutCrossProb - GAMutSwitchProb;
  }
  feature GAMutCrossProb GC.GraphVariable
  {
    Value                      = 0.5;      // Probability of crossover
    LimitValueToRange          = true;
    RangeMaximum               = 1.0;
  }
  feature GAMutProb GC.GraphVariable
  {
    Value                      = 0.03;     // Mutation during crossover
    LimitValueToRange          = true;
    RangeMaximum               = 1.0;
  }
  feature GAMutRotProb GC.GraphVariable
  {
    Value                      = 0.2;      // Probability of zone rotation
    LimitValueToRange          = true;
    RangeMaximum               = 1 - GAMutCrossProb - GAMutSwitchProb –
                                 GAMutCopyProb;
  }
  feature GAMutSwitchProb GC.GraphVariable
  {
    Value                      = 0.1;      // Probability of switching zones
    LimitValueToRange          = true;
    RangeMaximum               = 1 - GAMutCrossProb;
  }
  feature GAPopSize GC.GraphVariable
  {
    Value                      = 10;       // Population size
  }
  feature Photograph GC.GraphVariable
  {
    Value                      = false;    // Record image of each individual
  }
  feature SaveModels GC.GraphVariable
  {
    Value                      = true;     // Save Ecotect models for fittest
  }
}
```

The next transaction creates a graph in a separate window to record the fitness of each individual tested. By creating an additional coordinate system to act as a scale factor, the graph can be scaled after the points are plotted.

```
transaction modelBased "Create Fitness Plot"
{
  feature FitnessFrame GC.LawCurveFrame
  {
    Plane                      = FitnessPlotCS.XYPlane;
    Xdimension                 = GAGenerations;
    Ydimension                 = 0.75*GAGenerations;
    Xaxis                      = Series(0, GAGenerations,
                                 Max(GAGenerations/10, 1));
```

```
    Yaxis                     = Series(GAMinFit, GAMaxFit,
                                (GAMaxFit - GAMinFit)/10);
    XaxisName                 = "Generation";
    YaxisName                 = "Fitness";
    FormatXaxisAsInteger      = true;
    AnnotationSize            = GAGenerations*0.02;
  }
  feature FitnessPlotCS GC.CoordinateSystem
  {
    ModelName                 = "FitnessPlot";
    Visible                   = false;
  }
  feature FitnessScaleCS GC.CoordinateSystem
  {
    CoordinateSystem          = FitnessPlotCS;
    XTranslation              = 0;
    YTranslation              = -0.75*GAGenerations*GAMinFit /
                                (GAMaxFit - GAMinFit);
    ZTranslation              = 0;
    XRotation                 = 0;
    YRotation                 = 0;
    ZRotation                 = 0;
    YScale                    = FitnessFrame.Ydimension /
                                (GAMaxFit - GAMinFit);
    Visible                   = false;
  }
}
```

Another graph will record each gene's variance in each generation. For simplicity, the variances of corresponding genes from each zone are averaged.

```
transaction modelBased "Create Variance Plot"
{
  feature VarianceFrame GC.LawCurveFrame
  {
    Plane                     = VariancePlotCS.XYPlane;
    Xdimension                = GAGenerations;
    Ydimension                = 0.75*GAGenerations;
    Xaxis                     = Series(0, GAGenerations,
                                Max(GAGenerations/10, 1));
    Yaxis                     = Series(0, GAMaxVar, GAMaxVar/10);
    XaxisName                 = "Generation";
    YaxisName                 = "Variance";
    FormatXaxisAsInteger      = true;
    AnnotationSize            = GAGenerations*0.02;
  }
  feature VariancePlotCS GC.CoordinateSystem
  {
    ModelName                 = "VariancePlot";
    Visible                   = false;
  }
  feature VarianceScaleCS GC.CoordinateSystem
  {
    CoordinateSystem          = VariancePlotCS;
    XTranslation              = 0;
    YTranslation              = 0;
    ZTranslation              = 0;
    XRotation                 = 0;
    YRotation                 = 0;
```

```
    ZRotation                 = 0;
    YScale                    = VarianceFrame.Ydimension / GAMaxVar;
    Visible                   = false;
  }
}
```

This very long transaction contains scripts associated with the actual GA. The first script contains the variation operators used in both deterministic crowding (DC) and mu+lambda (μ+λ). The next two scripts are the DC and μ+λ algorithms themselves. (Parallel hill-climbing uses the deterministic crowding script.) The last two functions calculate the Hamming distance between genotypes (used by DC) and save Ecotect models of the final population from each trial after the GA has run.

```
transaction modelBased "Genetic Algorithm Functions"
{
  feature CrossOver GC.GraphFunction
  {
    Definition                = double [][][] function (double [][][] mother,
                                double [][][] father)
    {
      int a, b, c, d;                                    // crossover variables
      double [][][] child = mother;

      // mate best from previous generation to create offspring
      if (Random() < GAMutCrossProb)
      {
        int dnaDimSum = child.Count + child[0].Count + child[0][0].Count;
        a = Random(-10, 10);
        b = Random(-10, 10);
        c = Random(-10, 10);
        d = Random(-dnaDimSum/2, dnaDimSum/2);

        for (int i = 0; i < child.Count; i++)
        {
          for (int j = 0; j < child[i].Count; j++)
          {
            for (int k = 0; k < child[i][j].Count; k++)
            {
              if (a*(i-(child.Count-1)/2) + b*(j-(child[i].Count-1)/2) +
                c*(k-(child[i][j].Count-1)/2) > d)
              {
                child[i][j][k] = father[i][j][k];
              }
              if (Random() < GAMutProb)
              {
                child[i][j][k] = Random(DNAStep)/DNAStep;
              }
            }
          }
        }
      }
```

```
    // switch two zones from a previous individual to create offspring
    else if (Random() < GAMutSwitchProb / (1-GAMutCrossProb))
    {
      a = Random(child.Count);
      b = Random(child[0].Count);
      c = Random(child.Count);
      d = Random(child[0].Count);
      child[a][b] = mother[c][d];
      child[c][d] = mother[a][b];
    }

    // duplicate a zone within a previous individual to create offspring
    else if (Random() < GAMutCopyProb / (1-GAMutCrossProb-GAMutSwitchProb))
    {
      a = Random(child.Count);
      b = Random(child[0].Count);
      c = Random(child.Count);
      d = Random(child[0].Count);
      child[a][b] = mother[c][d];
    }

    // rotate a zone within a previous individual to create offspring
    else if (Random() < GAMutRotProb /
      (1-GAMutCrossProb-GAMutSwitchProb-GAMutCopyProb))
    {
      a = Random(child.Count);
      b = Random(child[0].Count);
      c = Random(3) + 1;        // number of times to rotate 90 degrees ccw
      for (int i = 0; i < c; i++)
      {
        d = child[a][b][1];     // switch genes that control roof angles
        child[a][b][1] = 1 - child[a][b][2];
        child[a][b][2] = d;
        d = child[a][b][7];     // switch genes that control side materials
        child[a][b][7] = child[a][b][6];
        child[a][b][6] = child[a][b][5];
        child[a][b][5] = child[a][b][4];
        child[a][b][4] = d;
      }
    }

    // mutate best from parent generation to create offspring
    else
    {
      a = Random(child.Count);
      b = Random(child[0].Count);
      c = Random(child[0][0].Count);
      child[a][b][c] = Random(DNAStep)/DNAStep;
    }
    return child;
  };
}
feature DeterministicCrowding GC.GraphFunction
{
  Definition                = function (int loop)
  {
    string driver = "";
    Ecotect command = new Ecotect();  // send commands with this object

    //                        --- VARIABLES FOR GENETIC ALGORITHM ---

    // DNA
    double dna = FilledList3d(HouseXDiv, HouseYDiv, HouseParams);
```

```
double child = FilledList(GAPopSize, dna);    // contains all offspring
double parent = FilledList(GAPopSize, dna);   // contains all parents

// Crowding
double randoms = FilledList(GAPopSize);   // list of random numbers
int matings = FilledList(GAPopSize);      // indeces of mating pairs
int temp;

// Fitness
double [][] df;                           // daylight factors
double fitness = {};                      // fitness of each offspring
int sorted = {};                          // indeces sorted by fitness
int best;                                 // index of best offspring
double minFit = Pow(10,10);   // initialize with a very large number
double maxFit = 0;
double parentFitness = FilledList(GAPopSize, minFit);
string lineOut;                           // output to CSV file

// Variance
double var = dna;                         // variance of each gene
double rmsVar = {};           // average variance in each generation
double maxVar = GAMaxVar.RangeStepSize;
PolyLine varCurve = {};
Point vplot = {};
for (int gene = 0; gene < dna[0][0].Count; gene++)
  vplot[gene] = {};                 // second dim will be generations

// Data to watch
double volume = FilledList(GAPopSize, 0);
double surfArea = FilledList(GAPopSize, 0);
double windowArea = FilledList(GAPopSize, 0);
double floorArea = FilledList(GAPopSize, 0);

//                      --- PREPARE OUTPUT FILE ---

if (ExportData)
{
  lineOut = "Fitness_DC#" + RunID;
  lineOut += ",Volume_DC#" + RunID;
  lineOut += ",SurfArea_DC#" + RunID;
  lineOut += ",WindowArea_DC#" + RunID;
  lineOut += ",Floor_DC#" + RunID;
  lineOut += ",ZVar_DC#" + RunID;
  lineOut += ",XVar_DC#" + RunID;
  lineOut += ",YVar_DC#" + RunID;
  lineOut += ",TVar_DC#" + RunID;
  lineOut += ",SVar_DC#" + RunID;
  lineOut += ",WVar_DC#" + RunID;
  lineOut += ",NVar_DC#" + RunID;
  lineOut += ",EVar_DC#" + RunID;
  lineOut += ",VVar_DC#" + RunID;
  lineOut += ",DNA_" + dna.Count + "x" + dna[0].Count + "x" +
    dna[0][0].Count;
  lineOut += "," + ZoneClimate;
  OpenPrintFile(CurrentDirectory() + "Results\\Data"+RunID+".csv");
  Print(lineOut);
  ClosePrintFile();
}
if (Photograph)
{
  command.Execute("set.movie.type jpg", driver);
  command.Execute("movie.record " + CurrentDirectory() +
    "Results\\Movie" + RunID + ".jpg", driver);
```

```
}

//                      --- PREPARE COORDINATE SYSTEM ---

string name;
CoordinateSystem cs;
if (DisplayAll)
{
  cs = new CoordinateSystem("CSgrid");        // create grid of houses
  cs.Replication = ReplicationOption.AllCombinations;
  cs.ByCartesianCoordinates(baseCS,
    Series(0, (GAPopSize-1)*HouseSpacing, HouseSpacing),
    Series(0, (GAGenerations-1)*HouseSpacing, HouseSpacing), 0);
  cs.Visible = false;
}
else
{
  cs = baseCS;                          // build all houses at same point
}

//                      --- CREATE POPULATION ---

for (int gen = 0; gen < GAGenerations; gen++)
{
  // accumulate gene sums in dna array to measure variance
  dna = FilledList3d(dna.Count, dna[0].Count, dna[0][0].Count, 0);
  var = FilledList3d(dna.Count, dna[0].Count, dna[0][0].Count, 0);
  rmsVar = FilledList(dna[0][0].Count, 0);

  // build random list of indeces for mating pairs
  for (int i = 0; i < GAPopSize; i++)
  {
    randoms[i] = Random();
  }
  matings = SortIndices(randoms, function(x,y){return x<y;});

  for (int pop = 0; pop < GAPopSize-1; pop += 2)
  {
    // Create individual's DNA
    if (gen == 0)                    // create new starting population
    {
      for (int i = 0; i < child[pop].Count; i++)
      {
        for (int j = 0; j < child[pop][i].Count; j++)
        {
          for (int k = 0; k < child[pop][i][j].Count; k++)
          {
            child[matings[pop]][i][j][k] = Random(DNAStep)/DNAStep;
            child[matings[pop+1]][i][j][k] = Random(DNAStep)/DNAStep;
          }
        }
      }
    }
    else                             // mate pairs of parents
    {
      child[matings[pop]] = CrossOver(parent[matings[pop]],
        parent[matings[pop+1]]);
      child[matings[pop+1]] = CrossOver(parent[matings[pop+1]],
        parent[matings[pop]]);
      if (Hamming(parent[matings[pop]],child[matings[pop+1]]) +
        Hamming(parent[matings[pop+1]],child[matings[pop]]) <
        Hamming(parent[matings[pop]],child[matings[pop]]) +
        Hamming(parent[matings[pop+1]],child[matings[pop+1]]))
```

```
// choose closest parents according to hamming distance between individuals
          {
            temp = matings[pop];
            matings[pop] = matings[pop+1];
            matings[pop+1] = temp;
          }
        }

        //                      --- EVALUATE POPULATION ---

        // Calculate fitness
        for (int i = pop; i < pop+2; i++)
        {
          name = HouseName;
          if (DisplayAll) name += gen + "_" + i;
          if (ActivateEcotect || Photograph)
            command.Execute("app.busy.update Testing round "+loop+
            ", generation "+gen+", individual "+i, driver);
          ExportBldg(name, (DisplayAll? cs[matings[i]][gen] : cs),
            child[matings[i]]);
          df = TestDaylight(DisplayAll? cs[matings[i]][gen] : cs);
          fitness[matings[i]] = TestLoads(df);  // <== FITNESS TESTED HERE

          // Capture image for movie
          if (Photograph)
          {
            command.Execute("set.app.page 2", driver);  // bring up openGL
            if (DisplayAll)
            {
              command.Execute("set.opengl.grid.max " +
                (HouseX*HouseXDiv/2+1500+matings[i]*HouseSpacing) + " " +
                (HouseY*HouseYDiv/2+1500+gen*HouseSpacing) + " 0", driver);
                                                  // set display grid max
              command.Execute("set.opengl.grid.min " +
                (-HouseX*HouseXDiv/2-1500+matings[i]*HouseSpacing) + " " +
                (-HouseY*HouseYDiv/2-1500+gen*HouseSpacing) + " 0",
                driver);                          // set display grid min
              command.Execute("set.opengl.eyept " +
                (20000+matings[i]*HouseSpacing) + " " + (gen*HouseSpacing-
                40000) + " 30000", driver);       // set camera position
              command.Execute("set.opengl.target " +
                (matings[i]*HouseSpacing) + " " + (gen*HouseSpacing) +
                " 6000", driver);                 // aim camera
            }
            command.Execute("app.busy.close", driver);
            command.Execute("help.title 0 36 Generation " + gen +
              ", Individual " + i + ": " + Round(fitness[matings[i]], 3) +
              " kWh", driver);            // pause time, font size, message
            command.Execute("opengl.draw.title true", driver);
                                                  // switch to 2D drawing
            for (int dot = 0; dot < fitness[matings[i]]; dot += 50)
            {
              command.Execute("opengl.draw.sphere 50 " + (dot+100) +
                " -100 100", driver);
                                // x, y, z, radius, from top left corner
            }
            command.Execute("movie.addframe", driver);
            command.Execute("app.busy.open Running genetic algorithm
              script", driver);
          }
```

```
    // Check fitness against closest parent
    if(fitness[matings[i]] < parentFitness[matings[i]])
    {
      parent[matings[i]] = child[matings[i]];
      parentFitness[matings[i]] = fitness[matings[i]];

      // Collect other data
      volume[matings[i]] = 0;
      surfArea[matings[i]] = 0;
      windowArea[matings[i]] = 0;
      floorArea[matings[i]] = 0;
      for (int zone = 1; zone <= HouseXDiv * HouseYDiv; zone++)
      {
        command.Request("get.zone.volume " + zone, driver);
        volume[matings[i]] += ToDouble(command.Result);
        command.Request("get.zone.exposedarea " + zone, driver);
        surfArea[matings[i]] += ToDouble(command.Result);
        command.Request("get.zone.windowarea " + zone, driver);
        windowArea[matings[i]] += ToDouble(command.Result);
        command.Request("get.zone.floorarea " + zone, driver);
        floorArea[matings[i]] += ToDouble(command.Result);
      }

      // Update best solution found so far
      if(fitness[matings[i]] < minFit)
      {
        minFit = fitness[matings[i]];
      }
    }

    if(fitness[matings[i]] > maxFit)
    {
      maxFit = fitness[matings[i]];
    }
  }
}

// Plot fitness of current individual
Point fplot = new Point("fplot" +
  gen).ByCartesianCoordinates(FitnessScaleCS, gen, fitness, 0);

//                      --- CALCULATE POPULATION DIVERSITY ---

// Calculate variance of genes in current generation
for (int i = 0; i < dna.Count; i++)
{
  for (int j = 0; j < dna[i].Count; j++)
  {
    for (int k = 0; k < dna[i][j].Count; k++)
    {
      dna[i][j][k] /= GAPopSize;      // mean value of each gene
      for (int pop = 0; pop < GAPopSize; pop++)
      {
        var[i][j][k] += Pow((child[pop][i][j][k] - dna[i][j][k]), 2)
          / GAPopSize;                    // calculation of variance
      }
      rmsVar[k] += Pow(var[i][j][k], 2);
                              // sum for root mean square of variance
    }
  }
}
```

```
  // Plot root mean square variance of genes in current generation
  for (int k = 0; k < rmsVar.Count; k++)
  {
    rmsVar[k] = Sqrt(rmsVar[k] / GAPopSize);
                                          // root mean square of variance
    if(rmsVar[k] > maxVar)
    {
      maxVar = rmsVar[k];
    }
    vplot[k][gen] = new Point().ByCartesianCoordinates(VarianceScaleCS,
      gen, rmsVar[k], 0);
    varCurve[k] = new PolyLine("Gene" + k).ByVertices(vplot[k]);
    varCurve[k].Color = k + 1;
  }

  //                      --- IDENTIFY MOST FIT INDIVIDUAL ---

  sorted = SortIndices(parentFitness, function(x,y){return x<y;});
          // return order of fitness from small (good) to large (poor)
  best = sorted[0];

  //                      --- WRITE DATA TO FILE ---

  if (ExportData)
  {
    lineOut = parentFitness[best] + ",";
    lineOut += volume[best] + ",";
    lineOut += surfArea[best] + ",";
    lineOut += windowArea[best] + ",";
    lineOut += floorArea[best] + ",";
    for (int k = 0; k < rmsVar.Count; k++)
    {
      lineOut += rmsVar[k] + ",";
    }
    for (int i = 0; i < parent[best].Count; i++)
    {
      for (int j = 0; j < parent[best][i].Count; j++)
      {
        for (int k = 0; k < parent[best][i][j].Count; k++)
        {
          lineOut += parent[best][i][j][k] + ",";
        }
      }
    }
    OpenPrintFile(CurrentDirectory() + "Results\\Data"+RunID+".csv",
      true);
    Print(lineOut);
    ClosePrintFile();
  }
}

//                      --- FINISH UP ---

// Save last population
if (ExportData)
{
  OpenPrintFile(CurrentDirectory() + "Results\\Data"+RunID+".csv",
    true);
  for (int pop = 1; pop < GAPopSize; pop++)
  {
    lineOut = parentFitness[sorted[pop]] + ",";
    lineOut += volume[sorted[pop]] + ",";
    lineOut += surfArea[sorted[pop]] + ",";
```

```
        lineOut += windowArea[sorted[pop]] + ",";
        lineOut += floorArea[sorted[pop]] + ",";
        for (int k = 0; k < rmsVar.Count; k++)
        {
          lineOut += ",";
        }
        for (int i = 0; i < parent[sorted[pop]].Count; i++)
        {
          for (int j = 0; j < parent[sorted[pop]][i].Count; j++)
          {
            for (int k = 0; k < parent[sorted[pop]][i][j].Count; k++)
            {
              lineOut += parent[sorted[pop]][i][j][k] + ",";
            }
          }
        }
        Print(lineOut);
      }
      ClosePrintFile();
    }
    // Save last generation for later analysis
    if (SaveModels)
    {
      for (int pop = 0; pop < GAPopSize; pop++)
      {
        double load = SaveModel(pop, parent[sorted[pop]]);
      }
    }

    // Update fitness graph axes
    GAMinFit = Floor(minFit/100) * 100;
    GAMaxFit = Ceiling(maxFit/100) * 100;
    //GAMaxVar = Ceiling(maxVar*10) / 10;

    // Save graphs
    if (Photograph) command.Execute("movie.finish", driver);
    if (ExportData)
    {
      UpdateGraph();          // refresh screen to add points to graphs
      ImageCapture fitCap = new ImageCapture().CaptureImage(5, 500,
        CurrentDirectory() + "Results\\", "FitnessPlot", RunID,
        RenderOption.Wireframe);          // save fitness plot as jpg
      ImageCapture varCap = new ImageCapture().CaptureImage(6, 500,
        CurrentDirectory() + "Results\\", "VariancePlot", RunID,
        RenderOption.Wireframe);          // save variance plot as jpg
    }
  };
}
feature GeneticAlgorithm GC.GraphFunction
{
  Definition                = function (int loop)
  {
    string driver = "";
    Ecotect command = new Ecotect();  // send commands with this object

    //                      --- VARIABLES FOR GENETIC ALGORITHM ---

    // DNA
    int keepers = ToInt(GAPopSize * GAKeepRate);
    double dna = FilledList3d(HouseXDiv, HouseYDiv, HouseParams);
    double child = FilledList(GAPopSize, dna);    // contains all offspring
    double parent = FilledList(keepers, dna);     // contains all parents
```

```
// Fitness
double [][] df;                          // daylight factors
double fitness = {};                     // fitness of each offspring
int sorted = {};                         // indeces sorted by fitness
double minFit = Pow(10,10);     // initialize with a very large number
double maxFit = 0;
string lineOut;                            // output to CSV file

// Variance
double var = dna;                          // variance of each gene
double rmsVar = {};           // average variance in each generation
double maxVar = GAMaxVar.RangeStepSize;
PolyLine varCurve = {};
Point vplot = {};
for (int gene = 0; gene < dna[0][0].Count; gene++)
  vplot[gene] = {};           // second dimension will be generations

// Data to watch
double volume = FilledList(GAPopSize, 0);
double surfArea = FilledList(GAPopSize, 0);
double windowArea = FilledList(GAPopSize, 0);
double floorArea = FilledList(GAPopSize, 0);
double sortVolume = {};
double sortSurfArea = {};
double sortWindowArea = {};
double sortFloorArea = {};

//                      --- PREPARE OUTPUT FILE ---

if (ExportData)
{
  lineOut = "Fitness_GA#" + RunID;
  lineOut += ",Volume_GA#" + RunID;
  lineOut += ",SurfArea_GA#" + RunID;
  lineOut += ",WindowArea_GA#" + RunID;
  lineOut += ",Floor_GA#" + RunID;
  lineOut += ",ZVar_GA#" + RunID;
  lineOut += ",XVar_GA#" + RunID;
  lineOut += ",YVar_GA#" + RunID;
  lineOut += ",TVar_GA#" + RunID;
  lineOut += ",SVar_GA#" + RunID;
  lineOut += ",WVar_GA#" + RunID;
  lineOut += ",NVar_GA#" + RunID;
  lineOut += ",EVar_GA#" + RunID;
  lineOut += ",VVar_GA#" + RunID;
  lineOut += ",DNA_" + dna.Count + "x" + dna[0].Count + "x" +
    dna[0][0].Count;
  lineOut += "," + ZoneClimate;
  OpenPrintFile(CurrentDirectory() + "Results\\Data"+RunID+".csv");
  Print(lineOut);
  ClosePrintFile();
}
if (Photograph)
{
  command.Execute("set.movie.type jpg", driver);
  command.Execute("movie.record " + CurrentDirectory() +
    "Results\\Movie" + RunID + ".jpg", driver);
}

//                      --- PREPARE COORDINATE SYSTEM ---

string name;
CoordinateSystem cs;
```

```
if (DisplayAll)
{
  cs = new CoordinateSystem("CSgrid");        // create grid of houses
  cs.Replication = ReplicationOption.AllCombinations;
  cs.ByCartesianCoordinates(baseCS,
    Series(0, (GAPopSize-1)*HouseSpacing, HouseSpacing),
    Series(0, (GAGenerations-1)*HouseSpacing, HouseSpacing), 0);
  cs.Visible = false;
}
else
{
  cs = baseCS;                          // build all houses at same point
}

//                      --- CREATE POPULATION ---

for (int gen = 0; gen < GAGenerations; gen++)
{
  // accumulate gene sums in dna array to measure variance
  dna = FilledList3d(dna.Count, dna[0].Count, dna[0][0].Count, 0);
  var = FilledList3d(dna.Count, dna[0].Count, dna[0][0].Count, 0);
  rmsVar = FilledList(dna[0][0].Count, 0);

  for (int pop = 0; pop < GAPopSize; pop++)
  {
    // Create individual's DNA
    if (gen == 0)               // create new starting population
    {
      for (int i = 0; i < child[pop].Count; i++)
      {
        for (int j = 0; j < child[pop][i].Count; j++)
        {
          for (int k = 0; k < child[pop][i][j].Count; k++)
          {
            child[pop][i][j][k] = Random(DNAStep)/DNAStep;
          }
        }
      }
    }
    else if (pop < keepers)   // retain best from previous generation
    {
      child[pop] = parent[pop];
      /*if (DisplayAll)       // option to display previously tested
      {                       // houses that advance to next generation
        name = HouseName + gen + "_" + pop;
        Polygon bldg = new Polygon(name).ByFunction(makeBuilding,
          {cs[pop][gen], HouseX, HouseY, HouseZ, HouseXDiv, HouseYDiv,
          HouseZMin, DNAStep, child[pop]});
      }*/
    }
    else
    {
      child[pop] = CrossOver(parent[Random(keepers)],
        parent[Random(keepers)]);
    }

    //                      --- EVALUATE POPULATION ---

    // Calculate fitness
    if (gen == 0 || pop >= keepers)
                        // fitness not yet found for current individual
    {
      name = HouseName;
```

```
if (DisplayAll) name += gen + "_" + pop;
if (ActivateEcotect || Photograph)
  command.Execute("app.busy.update Testing round "+loop+
  ", generation "+gen+", individual "+pop, driver);
ExportBldg(name, (DisplayAll? cs[pop][gen] : cs), child[pop]);
df = TestDaylight(DisplayAll? cs[pop][gen] : cs);
fitness[pop] = TestLoads(df);         // <== FITNESS TESTED HERE

// Capture image for movie
if (Photograph)
{
  command.Execute("set.app.page 2", driver);  // bring up openGL
  if (DisplayAll)
  {
    command.Execute("set.opengl.grid.max " +
      (HouseX*HouseXDiv/2+1500+pop*HouseSpacing) + " " +
      (HouseY*HouseYDiv/2+1500+gen*HouseSpacing) + " 0", driver);
                                        // set display grid max
    command.Execute("set.opengl.grid.min " +
      (-HouseX*HouseXDiv/2-1500+pop*HouseSpacing) + " " +
      (-HouseY*HouseYDiv/2-1500+gen*HouseSpacing) + " 0",
      driver);                          // set display grid min
    command.Execute("set.opengl.eyept " +
      (20000+pop*HouseSpacing) + " " + (gen*HouseSpacing-40000) +
      " 30000", driver);                // set camera position
    command.Execute("set.opengl.target " + (pop*HouseSpacing) +
      " " + (gen*HouseSpacing) + " 6000", driver);  // aim camera
  }
  command.Execute("app.busy.close", driver);
  command.Execute("help.title 0 36 Generation " + gen +
    ", Individual " + pop + ": " + Round(fitness[pop], 3) +
    " kWh", driver);            // pause time, font size, message
  command.Execute("opengl.draw.title true", driver);
                                        // switch to 2D drawing
  for (int dot = 0; dot < fitness[pop]; dot += 50)
  {
    command.Execute("opengl.draw.sphere 50 " + (dot+100) +
    " -100 100", driver);
                        // x, y, z, radius, from top left corner
  }
  command.Execute("movie.addframe", driver);
  command.Execute("app.busy.open Running genetic algorithm
    script", driver);
}

// Update best solution found so far
if(fitness[pop] < minFit)
{
  minFit = fitness[pop];
}
if(fitness[pop] > maxFit)
{
  maxFit = fitness[pop];
}

// Collect other data
for (int zone = 1; zone <= HouseXDiv * HouseYDiv; zone++)
{
  command.Request("get.zone.volume " + zone, driver);
  volume[pop] += ToDouble(command.Result);
  command.Request("get.zone.exposedarea " + zone, driver);
  surfArea[pop] += ToDouble(command.Result);
  command.Request("get.zone.windowarea " + zone, driver);
```

```
      windowArea[pop] += ToDouble(command.Result);
      command.Request("get.zone.floorarea " + zone, driver);
      floorArea[pop] += ToDouble(command.Result);
    }
  }
}

// Plot fitness of current individual
Point fplot = new Point("fplot" +
  gen).ByCartesianCoordinates(FitnessScaleCS, gen, fitness, 0);

//                      --- CALCULATE POPULATION DIVERSITY ---

// Calculate variance of genes in current generation
for (int i = 0; i < dna.Count; i++)
{
  for (int j = 0; j < dna[i].Count; j++)
  {
    for (int k = 0; k < dna[i][j].Count; k++)
    {
      dna[i][j][k] /= GAPopSize;
    // this is now the mean value of each gene in current population
      for (int pop = 0; pop < GAPopSize; pop++)
      {
        var[i][j][k] += Pow((child[pop][i][j][k] - dna[i][j][k]), 2)
          / GAPopSize;                    // calculation of variance
      }
      rmsVar[k] += Pow(var[i][j][k], 2);
                              // sum for root mean square of variance
    }
  }
}

// Plot root mean square variance of genes in current generation
for (int k = 0; k < rmsVar.Count; k++)
{
  rmsVar[k] = Sqrt(rmsVar[k] / GAPopSize);
                                        // root mean square of variance
  if(rmsVar[k] > maxVar)
  {
    maxVar = rmsVar[k];
  }
  vplot[k][gen] = new Point().ByCartesianCoordinates(VarianceScaleCS,
    gen, rmsVar[k], 0);
  varCurve[k] = new PolyLine("Gene" + k).ByVertices(vplot[k]);
  varCurve[k].Color = k + 1;
}

//                      --- SORT POPULATION BY FITNESS ---

sorted = SortIndices(fitness, function(x,y){return x<y;});
        // return order of fitness from small (good) to large (poor)
sortVolume = volume;
sortSurfArea = surfArea;
sortWindowArea = windowArea;
sortFloorArea = floorArea;
volume = FilledList(GAPopSize, 0);
surfArea = FilledList(GAPopSize, 0);
windowArea = FilledList(GAPopSize, 0);
floorArea = FilledList(GAPopSize, 0);
for (int i = 0; i < keepers; i++)
{
  parent[i] = child[sorted[i]];
```

```
    volume[i] = sortVolume[sorted[i]];
    surfArea[i] = sortSurfArea[sorted[i]];
    windowArea[i] = sortWindowArea[sorted[i]];
    floorArea[i] = sortFloorArea[sorted[i]];
  }
  fitness = Sort(fitness, function(x,y){return x<y;});
      // reorder fitness results to match ordering of parent population

  //                      --- WRITE DATA TO FILE ---

  if (ExportData)
  {
    lineOut = fitness[0] + ",";
    lineOut += volume[0] + ",";
    lineOut += surfArea[0] + ",";
    lineOut += windowArea[0] + ",";
    lineOut += floorArea[0] + ",";
    for (int k = 0; k < rmsVar.Count; k++)
    {
      lineOut += rmsVar[k] + ",";
    }
    for (int i = 0; i < parent[0].Count; i++)
    {
      for (int j = 0; j < parent[0][i].Count; j++)
      {
        for (int k = 0; k < parent[0][i][j].Count; k++)
        {
          lineOut += parent[0][i][j][k] + ",";
        }
      }
    }
    OpenPrintFile(CurrentDirectory() + "Results\\Data"+RunID+".csv",
      true);
    Print(lineOut);
    ClosePrintFile();
  }
}

//                      --- FINISH UP ---

// Save last population
if (ExportData)
{
  OpenPrintFile(CurrentDirectory() + "Results\\Data"+RunID+".csv",
    true);
  for (int pop = 1; pop < keepers; pop++)
  {
    lineOut = fitness[pop] + ",";
    lineOut += volume[pop] + ",";
    lineOut += surfArea[pop] + ",";
    lineOut += windowArea[pop] + ",";
    lineOut += floorArea[pop] + ",";
    for (int k = 0; k < rmsVar.Count; k++)
    {
      lineOut += ",";
    }
    for (int i = 0; i < parent[pop].Count; i++)
    {
      for (int j = 0; j < parent[pop][i].Count; j++)
      {
        for (int k = 0; k < parent[pop][i][j].Count; k++)
        {
          lineOut += parent[pop][i][j][k] + ",";
```

```
            }
          }
        }
        Print(lineOut);
      }
      ClosePrintFile();
    }
    // Save last generation for later analysis
    if (SaveModels)
    {
      for (int pop = 0; pop < keepers; pop++)
      {
        double load = SaveModel(pop, parent[pop]);
      }
    }

    // Update fitness graph axes
    GAMinFit = Floor(minFit/100) * 100;
    GAMaxFit = Ceiling(maxFit/100) * 100;
    //GAMaxVar = Ceiling(maxVar*10) / 10;

    // Save graphs
    if (Photograph) command.Execute("movie.finish", driver);
    if (ExportData)
    {
      UpdateGraph();          // refresh screen to add points to graphs
      ImageCapture fitCap = new ImageCapture().CaptureImage(5, 500,
        CurrentDirectory() + "Results\\", "FitnessPlot", RunID,
        RenderOption.Wireframe);          // save fitness plot as jpg
      ImageCapture varCap = new ImageCapture().CaptureImage(6, 500,
        CurrentDirectory() + "Results\\", "VariancePlot", RunID,
        RenderOption.Wireframe);          // save variance plot as jpg
    }
  };
}
feature Hamming GC.GraphFunction
{
  Definition                = int function (double [][][] parent,
                              double [][][] child)
  {
    int hamming = 0;
    for (int i = 0; i < child.Count; i++)
    {
      for (int j = 0; j < child[i].Count; j++)
      {
        for (int k = 0; k < child[i][j].Count; k++)
        {
          hamming += DNAStep * Abs(parent[i][j][k] - child[i][j][k]);
        }
      }
    }
    return hamming;
  };
}
feature SaveModel GC.GraphFunction
{
  Definition                = double function (int pop, double [][][] dna)
  {
    // Connect to Ecotect
    string driver = "";
    Ecotect command = new Ecotect();  // send commands with this object
```

```
      // Bring up model in Ecotect
      if (ActivateEcotect || Photograph)
        command.Execute("app.busy.update Recreating individual " + pop,
        driver);
      ExportBldg(HouseName, baseCS, dna);
      double [][] daylightFactor = TestDaylight(baseCS);
      double load = TestLoads(daylightFactor);

      // Save data from Ecotect
      command.Execute("calc.thermal.loads", driver);
      command.Execute("graph.save.bmp " + CurrentDirectory() +
        "Results\\Graph" + RunID + "_" + pop + ".bmp", driver);
                                                  // save analysis image
      command.Execute("graph.save.results " + CurrentDirectory() +
        "Results\\Graph" + RunID + "_" + pop + ".csv", driver);
                                                        // save data file
      command.Execute("model.dump " + CurrentDirectory() + "Results\\Model" +
        RunID + "_" + pop + ".eco", driver);              // save model
      return load;
    };
  }
}
```

Following the functions that carry out the GA is a script that allows the user to run the GA. The user can specify the number of times to run the GA through a dialog box.

```
transaction script "Run Genetic Algorithm"
{
  int loops = Ask("Number of times to repeat algorithm:");
  if (loops != null)
  {
    int rid = Random(100000 - loops);
    ShowMessageBox("Running the genetic algorithm may take a while");

    string driver = "";
    Ecotect command = new Ecotect();  // send commands with this object

    // Bring Ecotect window to front
    command.Execute("script.start", driver);
    if (ActivateEcotect || Photograph)
    {
      command.Execute("app.activate", driver);
      command.Execute("app.minimise false", driver);
      command.Execute("app.busy.open Running genetic algorithm script",
        driver);
    }
    else
    {
      command.Execute("app.minimise true", driver);
    }

    // Run algorithm
    for (int i = 0; i < loops; i++)
    {
      GraphVariable RunID = new
        GraphVariable("RunID").EvaluateExpression(rid+i);
      if (DetCrowding)
```

```
      {
        DeterministicCrowding(i);
      }
      else
      {
        GeneticAlgorithm(i);
      }
      if (Photograph && !ActivateEcotect)
      {
        command.Execute("app.busy.close", driver);
        command.Execute("app.minimise true", driver);
        Photograph = false;               // Only photograph on first run
      }
    }

    // Minimize Ecotect
    if (ActivateEcotect || Photograph)
    {
      command.Execute("app.busy.close", driver);
      command.Execute("app.minimise true", driver);
    }
    command.Execute("script.end", driver);
    if (ExportData) ShowMessageBox("Finished running the genetic algorithm.
      Data is saved.");
  }
}
```

If the algorithm saved Ecotect models of the fittest houses, a copy of the last one will still be in GC model space. Before saving images of the population, this copy is removed with the next transaction.

```
transaction modelBased "Delete FinalHouse"
{
  deleteFeature FinalHouse;
}
```

Finally, a tranaction that captures screen images of the model in GC is available. This transaction may be replayed an unlimited number of times in order to take additional pictures.

```
transaction modelBased "Save Model Frame"
{
  feature imageCaptureBest GC.ImageCapture
  {
    ViewNumber                = 2;
    ImageHeight               = 1000;
    DirectoryPath             = CurrentDirectory() + "Results\\";
    ImageFileSeedName         = AskString("Name of image:");
    StartFrame                = RunID;
  }
}
```

REFERENCES

Autodesk and the American Institute of Architects. *The 2008 Autodesk/AIA Green Index*. Chicago: StrategyOne, 2008.

Axelrod, Robert. *The Evolution of Cooperation*. New York: Basic Books, 1984.

Beinhocker, Eric D. *The Origin of Wealth*. Boston: Harvard Business School Press, 2007.

Borges, Jorge Luis. "The Library of Babel." *Labyrinths: Selected Stories and Other Writings*. Trans. James E. Irby. New York: New Directions Publishing Company, 1962. 51-58.

Carranza, Pablo Miranda. *Self-Designed Structures. Prototype 1*. 1 June 2005. Army of Clerks. 23 February 2008. <http://armyofclerks.net/SelfDesign/SelfDesignedStructures.pdf>.

De Biswas, Kaustuv. *GC Ecotect Link*. 4 February 2008. Kaustuv De Biswas. 17 December 2008. <http://web.mit.edu/kkdb/www/newhome/generative/index.html>.

Ecotect: An Overview. 2008. Autodesk. 17 December 2008. <http://ecotect.com/products/ecotect>.

Epstein, Joshua M. and Robert Axtell. *Growing Artificial Societies: Social Science from the Bottom Up*. Cambridge: The MIT Press, 1996.

Gell-Mann, Murray. *The Quark and the Jaguar: Adventures in the Simple and the Complex*. New York: Henry Holt and Company, LLC, 1994.

GenerativeComponents. Bently Systems, Incorporated. 17 December 2008. <http://www.bentley.com/en-US/Markets/Building/GenerativeComponents/>.

Goldberg, David E., Kalyanmoy Deb and Bradley Korb. "Messy Genetic Algorithms Revisited: Studies in Mixed Size and Scale." *Complex Systems*. 4 (1990): 415-444.

Goodwin, Brian. *How the Leopard Changed Its Spots: The Evolution of Complexity*. Princeton: Princeton University Press, 1994.

Holland, John H. *Hidden Order: How Adaptation Builds Complexity*. New York: Perseus Books, 1995.

Jacob, François. "Evolution and Tinkering." *Science*. 196.4295 (10 June 1977): 1161-1166.

Kauffman, Stuart. *The Origins of Order: Self-Organization and Selection in Evolution*. Oxford: Oxford University Press, 1993.

Koza, John R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. Cambridge: The MIT Press, 1992.

Kwinter, Sanford and Umberto Boccioni. "Landscapes of Change: Boccioni's 'Stati d'animo' as a General Theory of Models." *Assemblage*. 19 (Dec. 1992): 50-65.

LEED for New Construction. 2008. U.S. Green Building Council. 17 December 2008. <http://www.usgbc.org/DisplayPage.aspx?CMSPageID=220>.

Lewin, Roger. *Complexity: Life at the Edge of Chaos*. Chicago: The University of Chicago Press, 1992.

Miller, John H. and Scott E. Page. *Complex Adaptive Systems: An Introduction to Computational Models of Social Life*. Princeton: Princeton University Press, 2007.

SmartGeometry 2008 Conference. 2008. SmartGeometry. 17 December 2008. <http://www.smartgeometry2008.com/alumni.asp>.

Thompson, D'Arcy Wentworth. *On Growth and Form: The Complete Revised Edition*. New York: Dover Publications, Inc., 1992.

Waldrop, M. Mitchell. *Complexity: The Emerging Science at the Edge of Order and Chaos*. New York: Simon & Schuster, 1992.

Wiscombe, Tom. *EMERGENT*. 15 January 2009. EMERGENT. 29 January 2009. <http://www.emergentarchitecture.com/>.

Wright, Frank Lloyd. "The Art and Craft of the Machine." *Rethinking Technology: A Reader in Architectural Theory*. Ed. William W. Braham, John Stanislav Sadar, and Jonathan Hale. New York: Routledge, 2006. 1-16.

Wright, Sewall. "The Roles of Mutation, Inbreeding, Crossbreeding, and Selection in Evolution." *Proceedings of the Sixth International Congress of Genetics*. 2 vols. Ed. Donald F. Jones. Menasha, Wisconsin: Brooklyn Botanical Gardens, 1932. 356-366.