Motivation

**Automatic content generation** is the use of algorithms to generate content for games, web pages, or other digital domains. This enables:

- lower costs,
- replayability, and
- faster development.

This talk outlines an application of cellular automata to automatic content generation.
Agenda

1. A quick introduction to evolutionary computation.
3. A quick introduction cellular automata.
4. A few examples of evolved automata.
5. Fashion based automata and evolving cavern maps.
6. Scalability and reusability of the resulting automata.
Evolutionary computation and an unexpected design consultant

[Image of a man with a network diagram and a circuit diagram]

Ashlock (Guelph)  Representation in EC  3 / 39
Evolutionary computation is any algorithm that uses Darwin’s Theory of Evolution as its starting point. There are many schools of evolutionary computation. Holland and Goldberg invented Genetic Algorithms, Lawrence Fogel invented Evolutionary Programming, Ingo Rechenberg and Hans-Paul Schwefel invented Evolution Strategies, and there are several other flavors. In aggregate we call these techniques Evolutionary Algorithms.

In general:

- Evolutionary algorithms are best used on problems that you don’t understand well yet.
- They are easy to code but run slowly.
- There are large numbers of design choices which permit/require a great deal of tuning of algorithm performance.
A Parts List for Evolutionary Computation

In order to use evolutionary computation, you must assemble the following:

- A problem with a well-defined measure of quality for solutions.
- A data structure that can store solutions to the problem.
- Techniques for blending and modifying these data structures.
- A technique for choosing solutions with a pro-quality bias.
- A technique for choosing solutions to discard.
- A criterion to decide that you have found an acceptable answer.

Population based techniques

Evolutionary computation is an example of a population based technique. This means the algorithm operates on a collection (population) of solutions, using comparison between different population members to direct search.
Basic Evolutionary Computation Loop

Initialize a population of structures
Evaluate the quality (fitness) of each structure
Repeat
   Pick parents with a quality bias
   Generate child structures, evaluate their quality
   Pick decedents
   Conditionally replace decedents with children
Until (Good Enough)

Techniques for blending parents are called **crossover operators**, techniques for modifying individuals are called **mutation operators**. Collectively they are called **variation operators**. There are other variation operators. The quality measure for solutions is called the **fitness function**. Picking and discarding are called **selection** within the algorithm.
Types of Crossover

The diagrams at the left illustrate various types of crossover. All of these assume that the data structure or chromosome is stored in a linear array, e.g.

\[ CGTTACTGAGGTACCGT \]

or

\[ (1.73, 4, 86, 6.96, 3.14, 2.71) \]

Crossover points are typically chosen uniformly at random. In uniform crossover a random number is used at each loci in the chromosome to decide which parent’s value goes to what child.
Example: optimizing real parameters

A standard application is to optimize real parameters. Suppose we are fitting

\[ f(x) = (ax + b)e^{-rx} + d \]

to a set of data.

Data structure: an array holding a, b, r, d.
Fitness: an error measure on modeling of the data.
Crossover: one point.
Mutation: Add a normally distributed random number to each parameter.

Population Size

Another important choice in an evolutionary algorithm is population size. Depending on the fitness landscape of the problem different population sizes are optimal and, thus far, the correct size is determined experimentally.
Things to tune in an evolutionary algorithm

<table>
<thead>
<tr>
<th>Component</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fitness function</td>
<td>Critical</td>
</tr>
<tr>
<td>Representation or data structure</td>
<td>Critical</td>
</tr>
<tr>
<td>Selection and replacement techniques</td>
<td>Very important</td>
</tr>
<tr>
<td>Population size</td>
<td>Can be very important</td>
</tr>
<tr>
<td>Variation operators</td>
<td>Usually very important</td>
</tr>
<tr>
<td>Rate of application of operators</td>
<td>Moderately important</td>
</tr>
</tbody>
</table>

What are the good choices?

There is a technical theorem called the **No Free Lunch Theorem**. It has important consequences in evolutionary computation:

- Good choices are problem dependent.
- Performance is improved by (correctly) specializing the algorithm to your problem.
- When possible, incorporate expert knowledge.
Evolutionary computation choices

**So Far**
- Data structure for population members.
- Fitness function.
- Mutation operator(s).
- Crossover operator(s).
- Frequency of application for mutation and crossover operators.
- Population size.
- Selection and replacement techniques.

**Not Yet**
- Population structure.
- Are the variation operators adaptive?
- Do we save a “Hall of Fame”
- Do we want search or learning operators in with the variation operators?
- Incorporate expert knowledge...
  - ...in mutation operators,
  - ...in crossover operators,
  - ...in initial population selection?
Evolutionary Map Creation
A Direct Binary Representation

- This representation uses the encoding $1=\text{full}, 0=\text{empty}$.
- It only works if you start mostly empty and let evolution fill in more full squares later. We call this technique **sparse initialization**.
- The fitness function is based on dynamic programming that measures distances between checkpoints, shown as green dots.

A great deal may be done by playing with the fitness function; the new goal is to look at the impact of representation.
A Positive Generative Representation

This representation is an array of wall building commands.

Sparse initialization is used here as well, but in the form of short initial wall lengths.

The representation is additionally constrained by a limit on total number of squares that may be filled by walls.

The results here look more planned and less cave-like.

This representation uses walls with a starting point, direction, and length. There are eight available directions; it is trivial to modify the representation to use only some directions, e.g. horizontal and vertical.
A Negative Generative Representation

This representation is an array of room and corridor specifications to be removed from a matrix that starts full.

Sparse initialization is not needed. In this representation it is critical to have enough rooms and a “skip me” bit.

The entrance and exit are always placed in rooms that are part of the starting configuration to reduce the probability of death from non-traversability.

This representation specifies rooms and corridors with an upper left corner and horizontal and vertical dimensions. Corridors have one dimension set to one, rooms are at least two-by-two.
This representation is another version of the positive representation with three types of walls that make two mazes.

The stone-fire and stone-water mazes are evaluated with different fitness functions to create a tactical situation for two agent types.

A key factor is how to combine the two fitnesses; here a geometric average was used.

Citations


Do They Look Different?

Here are mazes evolved with different representations. The choice of representation gives **substantial** control over the character and appearance of the maps that evolve. With the exception of the fire-and-ice maze, these maps were evolved with the **same** fitness function.
Avoiding quadratic scaling by putting together pieces

Huge mazes can be assembled out of evolved tiles. Note the user-designed features present in some of the tiles. With controlled tile properties, we can achieve replayability. The assembly plan is also an evolved maze with controlled properties.
Putting Together Other Pieces

When assembling large mazes, tiles evolved with different representations may be mixed and matched. These examples **do not begin approach the limits of the technique**; they were chosen to **fit on the screen**.
The level generation technique was designed for video games but the algorithms can easily be modified to design **terrain maps**, **house plans**, or **plans for neighbourhoods**.

The decomposition to design large maps is **transparently generalizable** to other types of maps.

We have recently developed **multicriteria optimization techniques** that permits generation of collections of maps that span trade-off frontiers, e.g. traversable space versus distance between checkpoints.

The fire-and-ice map is a prototype for other types of maps with **multiple terrain classes**, e.g.: water, infantry, wheels, tracks.
**Cellular automata** are a type of discrete model of computation. A cellular automaton has three parts:

1. A collection of cells divided into neighborhoods of each cell,
2. A set of states that cells can have,
3. A rule that maps the set of possible cell states of a neighborhood to a new state for the cell with which the neighborhood is associated.

The *sum-of-neighborhood is index* representation cannot encode most CA rules - but it does encode a very large space and its simple linear representation is evolution-compliant.

The picture shows the first sixteen time-steps of a synchronous one-dimensional cellular automata together with its rule in two forms. The picture displays the automata’s time history.
Apoptotic Cellular Automata

An **apoptotic cellular automata** is one that enters a quiescent state before a stated time. They are easy to evolve; shown below are renderings of such automata.

Rules that generate apoptotic automata are strongly clustered in a small part of rules space under the Hamming metric. The fitness used to evolve apoptotic automata is number of living cells, but only if you die in time.
The fashion based automata being evolved use a von Neumann neighborhood. A **competition matrix** gives the score each cell state gets from having each other cell state as a neighbor. Updating consists of adopting the state of the highest scoring member of your neighborhood; ties go to the current cell state.

The matrix:

\[
\begin{bmatrix}
0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 \\
0.00 & 0.00 & 1.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 1.00 & 0.00 & 0.00 \\
1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 \\
0.00 & 0.00 & 0.00 & 1.00 & 0.00 & 1.00 & 0.00 & 0.00 \\
1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 & 0.00 \\
0.00 & 1.00 & 0.00 & 0.00 & 0.00 & 0.00 & 0.00 & 1.00 \\
0.00 & 0.00 & 1.00 & 0.00 & 1.00 & 0.00 & 0.00 & 0.00 \\
\end{bmatrix}
\]

generates the fashion based automata shown at the right.
Representing fashion based automata for evolution.

We will used a fixed random set of initial conditions so the object being evolved is a $n \times n$ square matrix.

- The data structure is the rows concatenated to get a length $n^2$-vector. Entries are selected uniformly at random in $[0,2]$.
- Mutation adds a number selected uniformly at random in the interval $[-0.1,0.1]$ to an entry; if it is outside $[0,2]$ a new random number in $[0,2]$ of selected.
- The algorithm uses two point crossover.
- Fitness is

$$fit = \frac{N}{1 + |2 \cdot U - 1|}$$

where $N$ is the size of a central connected component of cells in state 0 and $U$ is the fraction of non-zero cells.
Reference check: hand selected rules.
Evolved caverns

Reference: First CA-level map paper

Lawrence Johnson, Georgios N. Yannakakis, Julian Togellius Cellular automata for real-time generation of infinite cave levels, Proceedings of PC Games, 2010
Re-use and scalability.

All the automata evolved for this presentation used the same initial conditions - a $200 \times 200$ block of random numbers in the range 0-7. This makes the fitness computation repeatable and the algorithm easier to tune.

- The goal of **replayability** means that we want to be able to generate lots of similar maps without a lot of effort.
- The goal of **scalability** means that we want to be able to change the size of the map without a lot of added work.

This means that what happens when we re-use a rule with different, or differently sized, initial conditions it would be nice if the resulting map were similar to the one we evolved. There is good hope of this, because the **action of cellular automata is purely local**. When we change the initial conditions the resulting level map should look *locally* similar to the map used for fitness evaluation.
Cellular automata are scalable

Again: cellular automata rules are applied locally. Once a rule is evolved, much larger maps can be made by simply applying the rule to a larger grid. Changing the initial conditions creates maps with the same character but different details.

The initial conditions used here are generated uniformly at random. The grid where the automata are tested for fitness are toroidal so the maps also tile correctly.
Maps with multiple sorts of materials had the potential to encode maps like the fire-and-ice map way back at the beginning of the talk.
More examples of re-use

The rule used to generate these maps was the only one that yielded the right-angle corridor appearance. It would be nice to figure out how to generate features like this on demand.
This rule generates a structure like regular pillars and train-track like inclusions. The presence of lots of secondary features (features not directly encouraged by fitness) strongly suggests that more complex fitness functions are worth writing.
This rule generates a simple cavern structure. The automata are rendered to double size - the rendering grid may be enlarged to any convenient size. This complete and utter scalability is one of the most attractive features of this representation.
Modifying the density parameter

The fitness function \( fit = \frac{N}{1+|2U-1|} \) rewards a 50% full level map with a large open center. If we change to \( fit = \frac{N}{1+|\frac{U}{\alpha}-1|} \) we can set a desired density \( \alpha \):

\[
\alpha = 0.25 \\
\alpha = 0.5 \\
\alpha = 0.75
\]

Note that low alpha make the fitness function almost trivial.
Working with a new (old) fitness function

The decomposition by tiling requires that we generate tiles with predictable. We already have fitness function for fitness like:

1. There must be openings in the required locations.
2. It must be possible to travel between each pair of openings.
3. Maximize the average distance between pairs of openings.

**central massif fitness**

**connect sides fitness**
Laura’s project starts with a parameter study for the connect-the-sides fitness function algorithm. The default values for the algorithm are: population size 360, maximum mutations 7, tournament size 7, and 10,000 mating events.

For each set of parameters 30 runs are performed for these parameter values, varying one parameter of the default at a time.

Population Size: 370, 400, 500, 1000, 2000, 3000, 4000, 5000, 10000

Maximum number of mutations: 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12

Tournament size: 8, 10, 15, 20, 25, 30

Mating Events: 11000, 12000, 13000, 14000, 15000, 20000, 30000, 40000, 50000
## Population Size Results

<table>
<thead>
<tr>
<th>Population Size</th>
<th>Time of Last Innovation</th>
<th>Final Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>370</td>
<td>87.2 ± 3.9929</td>
<td>156.667 ± 4.44679</td>
</tr>
<tr>
<td>400</td>
<td>85.7667 ± 3.68866</td>
<td>163.717 ± 3.85402</td>
</tr>
<tr>
<td>500</td>
<td>86.2667 ± 4.98504</td>
<td>158.35 ± 3.64755</td>
</tr>
<tr>
<td>1000</td>
<td>77.7667 ± 6.33455</td>
<td>151.133 ± 3.82592</td>
</tr>
<tr>
<td>2000</td>
<td>65.1333 ± 7.45472</td>
<td>143.25 ± 2.71421</td>
</tr>
<tr>
<td>3000</td>
<td>61.9 ± 9.04863</td>
<td>138.167 ± 2.92243</td>
</tr>
<tr>
<td>4000</td>
<td>56.7333 ± 11.5501</td>
<td>134.5 ± 2.03027</td>
</tr>
<tr>
<td>5000</td>
<td>70.9667 ± 6.8804</td>
<td>135.967 ± 2.64633</td>
</tr>
<tr>
<td>10000</td>
<td>52.7667 ± 10.6178</td>
<td>132.867 ± 2.77303</td>
</tr>
</tbody>
</table>

Juxtaposing time-of-last-innovation with final fitness, these results suggest that large population sizes get stuck in local optima. Maybe try smaller sizes?
Maximum Number of Mutations Results

<table>
<thead>
<tr>
<th>Maximum Mutations</th>
<th>Time of Last Innovation</th>
<th>Final Fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>83.5±  5.18461</td>
<td>167.767± 5.31949</td>
</tr>
<tr>
<td>2</td>
<td>86.0333±  5.09156</td>
<td>166.883± 6.29556</td>
</tr>
<tr>
<td>3</td>
<td>84.7333±  5.38938</td>
<td>165.433± 5.31492</td>
</tr>
<tr>
<td>4</td>
<td>88.8±  3.37708</td>
<td>164.933± 4.98282</td>
</tr>
<tr>
<td>5</td>
<td>86.7667±  5.43812</td>
<td>164.633± 4.06188</td>
</tr>
<tr>
<td>6</td>
<td>83.9±  6.71598</td>
<td>164.217± 5.12931</td>
</tr>
<tr>
<td>8</td>
<td>84.9667±  5.14505</td>
<td>160.317± 4.44683</td>
</tr>
<tr>
<td>9</td>
<td>82.6667±  6.30005</td>
<td>158.25± 5.27866</td>
</tr>
<tr>
<td>10</td>
<td>74.8±  9.32077</td>
<td>157.083± 3.5868</td>
</tr>
<tr>
<td>11</td>
<td>76.266±  7.627519</td>
<td>158.15± 3.85988</td>
</tr>
<tr>
<td>12</td>
<td>82.4333±  5.07436</td>
<td>155.983± 3.94596</td>
</tr>
</tbody>
</table>

These results show that time of last innovation is fairly insensitive to the maximum number of mutations but lower mutation rates yield better fitness.
Morphing of Existing rules

These maps use averages of two rules, taking
\[ \lambda M_2 + (1 - \lambda) M_1 \quad 0 \leq \lambda \leq 1 \]
with \( \lambda \) depending on spatial position.

**Horizontal fraction is** \( \lambda \)

**Distance from center is** \( \lambda \)
Assigning meanings beyond “full” to the nonzero states - fire, water, treasure, rock?

Single parent studies - this is a way of exploring the space between rules.

More complex morphing between rules - exploring the fitness landscape.

**re-evolution studies**

Post processing for the cellular automata images.
Many Thanks

Thanks to the *Natural Sciences and Engineering Research Council of Canada* for support of this work.