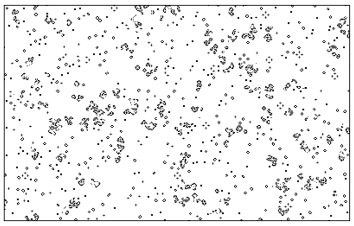
Still Life Theory

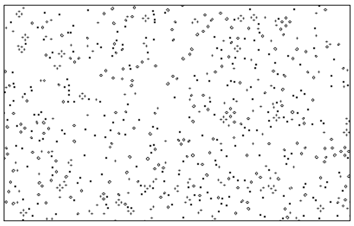
Matthew Cook

1 Introduction

In Conway's *Game of Life* [1], if one starts with a large[1] array of randomly[2] set cells, then after around twenty thousand generations one will see that all motion has died down, and only stationary objects of low period remain, providing a final density of about .0287. No methods are known for proving rigorously that this behavior should occur, but it is reliably observed in simulations.



After 500 steps, active areas are still common.



Eventually all activity subsides, leaving only debris.

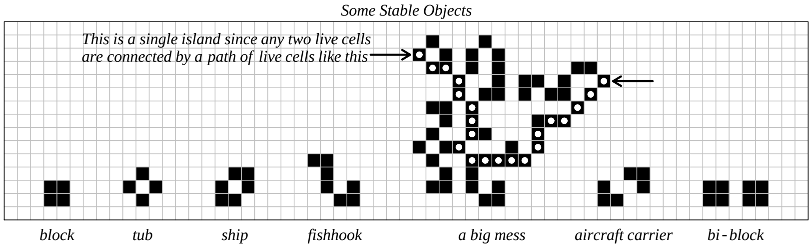
This brings up several interesting related questions. Why does this "freezing" occur? After everything has frozen, what is the remaining debris composed of? Is there some construction that can "eat through" the debris? If we start with an infinitely large random grid, so that all constructions appear somewhere, what will the long term behavior be? It seems clear that knowing the composition of typical debris is central to many such questions.

Much effort has gone into analyzing the objects that occur in such stationary debris, as well as into determining what stationary objects can exist at all in Life.[3, 4] Both of these endeavors depend on having some notion of what an "object" is in the first place. One simple notion is that of an *island*, a maximal set of live cells connected to each other by paths of purely live cells. But many common objects, such as the "aircraft carrier", are not connected so strongly. They are composed of more than one island, but we think of them as a single object anyway, since their constituent islands are not separately stable.

[1]For example, a torus large enough that even speed-of-light signals are unable to wrap around the torus during the run.

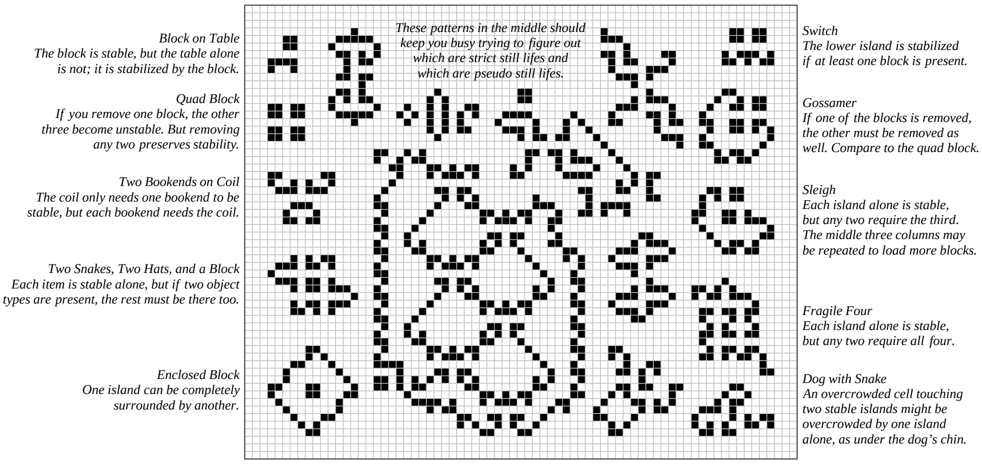
[2]The numbers mentioned above are for initial densities around one half. Much higher or lower densities will converge more quickly to a sparser result.[2]

Some Stable Objects



Any pattern that is *stable* (has period one, i.e. does not change over time) is called a *still life*. Since a collection of stable objects can satisfy this definition, the term *strict still life* is used to refer to a single indivisible stable object, and *pseudo still life* is used to refer to a stable pattern that is composed of distinct strict still lifes. For example, the bi-block is a pseudo still life, since it is composed of two blocks, but the aircraft carrier is a strict still life, since its islands are not stable on their own. The entirety of the above figure is stable, but it is clearly a pseudo still life, since it can be separated into a block, tub, ship, and so on. The distinction between strict and pseudo still lifes is intended to capture the separability of objects that is usually easy to detect by eye.

After this distinction started being used, in the early 1970's, it was discovered that things weren't as simple as they had appeared. One might have a pattern such as the "block on table" which consists of two islands, only one of which is stable alone. Is that one object or two? Or one might have a pattern such as the "switch" which consists of three islands, of which two are stable and one is stable only if at least one of the other two is present. How many objects is this? What are the objects?



As one looks at more and more patterns, it becomes clear that a very precise definition is going to be needed.

Eventually, in the late 1980's, the following became accepted as the standard definition:

Definition 1 (Pseudo Still Life) *A pseudo still life is any stable pattern whose islands can be partitioned into exactly two nonempty sets, each of which is stable on its own.[7]*

This definition is mathematically unambiguous, which was a significant improvement over previous definitions. But from a computational point of view, it might be of some concern that the natural algorithm for deciding whether a pattern is a pseudo still life is exponential in the number of islands (which in turn can be linear in the amount of input, i.e. the area of the pattern), since it appears that all possible partitions into two sets must be tried. In practice so far the definition has not been used in situations with large numbers of islands, and people have accepted the exponential complexity.

In Section 2 we will show that the problem in fact does not have exponential complexity as previously thought. We give an algorithm that solves it in $O(n^2)$ time.

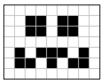
In Section 3 we will propose a natural modification of the definition to allow partitioning patterns into any number of sets of islands, rather than just two. We will show that this is equivalent to the definition allowing at most four sets of islands, due to an application of the four color theorem.

In Section 4 we will show that an intermediate definition, allowing at most three sets of islands, is NP-complete. This shows that subtle differences in the problem definition can have drastic consequences on the complexity.

In Section 5 we will show that the problem of testing patterns according to the modified definition proposed in Section 3 is in fact NP-complete as well. This is the most complicated proof, but it is greatly simplified by using ideas from the earlier proofs.

2 Fast Strictness Testing for Still Lifes

Here we will show that the problem of determining whether a pattern is a strict or pseudo still life takes only $O(n^2)$ time, by giving a decision algorithm which for the most part takes only linear time, but can take quadratic time in the number of "switch"-like instances.



Switch

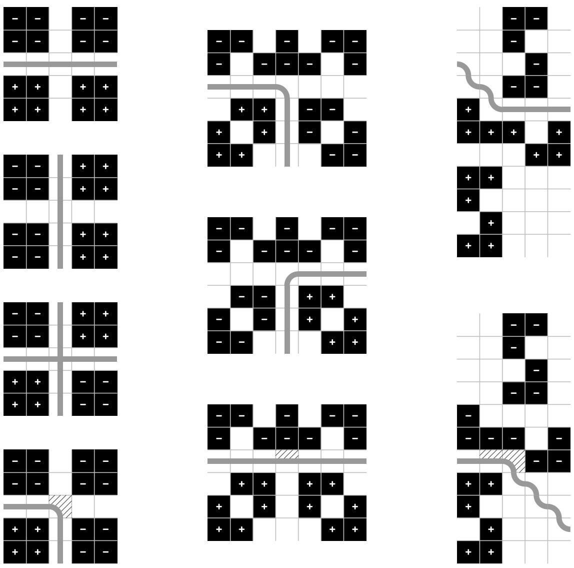
Note that if we like, we may verify in $O(n)$ time that a pattern is stable by checking that each live cell has two or three live neighbors, and no empty cell has three live neighbors.

The algorithm has two main parts: First, we convert the pattern into a connectivity graph of switches, thereby converting the problem into one we will call *Switch-Cycle*. Then, we give a simple algorithm for solving the *Switch-Cycle* problem in quadratic time.

2.1 Infrastructure Transforms the Problem

Given a pattern, we want to find whether we can decompose its islands into two stable sets. This is equivalent to finding a boundary that separates the two sets from each other. If we think of the islands as consisting of "land" cells, and we think of empty cells as "water" cells, then this boundary must go through the water, separating the islands in a stable way.

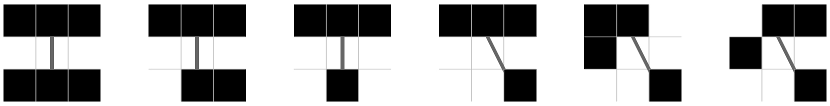
Since stability is a local property, possibilities for the boundary are determined locally throughout the pattern. All we have to do is figure out whether there exists a global boundary that divides the islands and is everywhere consistent with the local possibilities.



The boundaries in the upper pictures divide the patterns into two parts, shown with - and +, in a locally stable way. The boundaries in the three bottom pictures, however, would result in instabilities as shown, so they must be prohibited. Note how the two crossing boundaries in the third picture on the left result in diagonally opposite blocks belonging to the same group, since each crossing of the boundary represents switching from one group to the other.

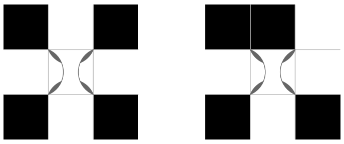
There are essentially three different kinds of restrictions that can occur locally. The first is that it may simply not be possible to have a boundary go through a certain water cell — any such boundary would lead to instability on one side or the other. On such cells, we will build *dams*, so if we think of the boundary path as a water route, then the boundary may not cross a dammed cell any more than water traffic can cross a dam.[3]

[3] Of course, in real life some kinds of water traffic can cross dams[8], so the analogy must not be taken too literally.



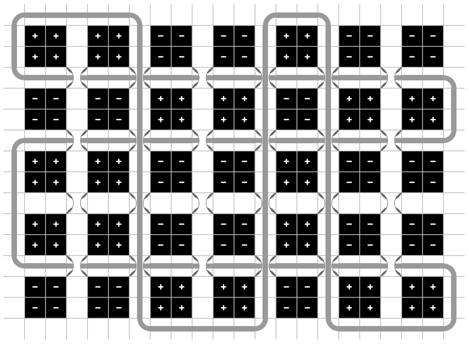
The six local arrangements where a dam is required to prevent the boundary from passing through the center cell, since any such boundary would result in instability. These are the configurations where a cell touches two stretches of land, one at three points.

The next kind of infrastructure we will need is *aquaducts.* Crossing aquaducts are placed on a cell to prevent the boundary from being able to make a turn there. The boundary must either go straight east-west, using one of the aquaducts, or go straight north-south, using the other one, but turning is not allowed. A cell containing crossing aquaducts is like two different cells: One that can be used by a north-south boundary, and one that can be used by an east-west boundary. In this way, crossing aquaducts allow water routes to cross each other without being connected. There are two situations in which crossing aquaducts are needed, although one of them results in one of the aquaducts being unusable as a water route since it comes to an immediate end. As with other infrastructure, the purpose of crossing aquaducts is to prevent the boundary from being able to pass through a water cell in such a way that exactly three land cells would wind up on one side of it, since that is what would cause instability.



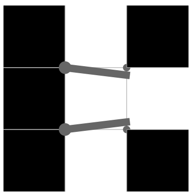
The two local arrangements requiring aquaduct crossings, shown as an east-west aquaduct passing under a north-south aquaduct. In the second arrangement, the north-south route ends immediately and so cannot be used by a boundary, and the net effect is the same as if a dam were to be placed between the two southern land cells.

If both aquaducts of an aquaduct crossing are used by the boundary, as might happen in the first arrangement shown, then opposite corners wind up being in the same set, since crossing the boundary corresponds to switching from one set to the other, so crossing the boundary twice results in the original set once again. This corresponds to an “even-odd fill” (or “winding number parity”) of the boundary being used to determine which set an island should belong to.

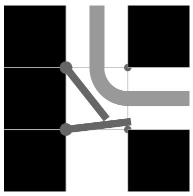


A self-crossing boundary path can be used to divide a pattern into two stable sets, if an "even-odd fill" is used to determine set membership

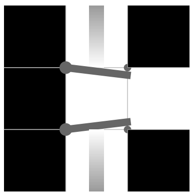
The last kind of infrastucture needed is *locks*. A lock consists of a pair of gates, each of which may be either open or closed, except that they may not both be open. If one gate of a lock is open, then the other must be closed. The boundary path may only pass through an open gate of a lock. It is easy to see that a lock is exactly what is needed to make sure that any path through the following arrangement will result in a stable boundary. This is the only arrangement requiring a lock.



A lock with both gates closed.



A lock with one gate open, allowing the boundary to pass through.

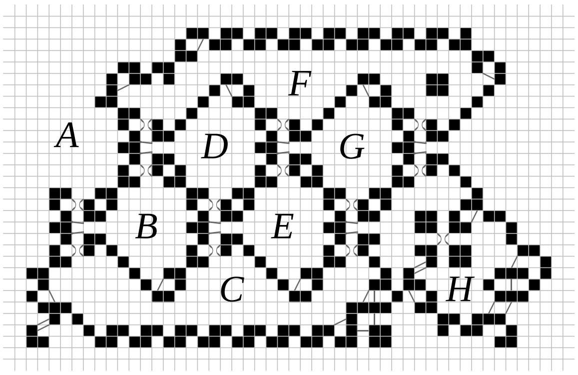


The boundary cannot cross a lock like this since the gates cannot both be open.

A lock is in some sense the opposite of crossing aquaducts, since locks only allow boundaries that turn, while aquaducts only allow boundaries that go straight.

These three types of infrastructure correspond to the three types of local restrictions that can exist for cells on the route of the boundary path. Except where infrastructure is required as explained above, there is no restriction on the water path of a boundary. Any boundary loop that respects the infrastructure and divides some islands from others will divide the islands into two stable sets, and if there is a division of islands into two stable sets, then we can draw a boundary loop around one of them (or around an isolated part of one of them).

Once we have installed all the infrastructure, the problem takes on a much simpler form if we consider the *seas* that result. A sea is a bunch of water cells that are connected to each other after the infrastructure is installed. For the purpose of determining the extents of the seas, all the lock gates must be closed.



With all lock gates closed, each connected body of water is a sea. If any sea is not simply connected, then a boundary can be drawn within that sea alone, as in sea F, which encloses a block, or H, which has a figure-8 loop using the aquaduct. Otherwise, lock gates need to be judiciously opened to allow a boundary if possible.
Note that there are many very small seas of just one or two cells which are not marked with a letter in this diagram. These seas cannot possibly be used by a boundary, since they are simply connected and do not touch any locks.

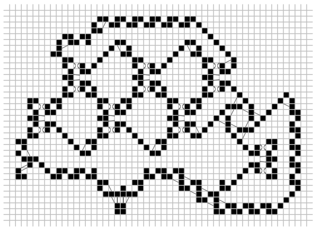
If any sea is not simply connected (that is, there is land both inside and outside it), then we can draw a boundary loop dividing land through that sea alone, and so the pattern is a pseudo still life. As we determine the extent of a sea, we can simultaneously check whether it is simply connected, all in $O(s)$ time, where s is the size of the sea. We merely need to "grow" the sea one cell at a time, keeping a list of neighboring sea cells that need to be added. (Cells with crossing aquaducts should be treated as two separate cells, one for each crossing direction.)

To check for simply-connectedness, when adding a cell, we first check whether more than one of its four neighbors has already been added to the sea, and if so, we check whether they are connected by cells already added to the sea among just the eight neighbors of the cell being added. If not, then adding this cell could for the moment result in not being simply connected, so instead of adding the cell to the sea now, we move it to a list of "postponed" cells. Any time a cell is added to the sea, if any of its eight neighbors are marked as "postponed", they are moved back to the regular list of cells waiting to be added, so they can be tried again. Once we have finished processing the regular list, then if any cells remain in the "postponed" list, then the sea is not simply connected.

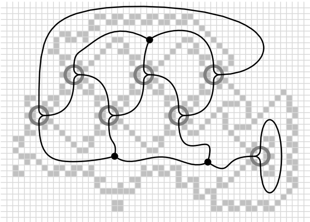
The "outside sea" requires special treatment during the check for simply-connectedness: Any water cell at the edge of the pattern should be considered as connected to all other water cells at the edge of the pattern. This "outside sea" can be thought of as wrapping all the way around the planet. When treated this way, the above algorithm will work correctly for it too.

When growing the seas, we must keep track for each sea of what locks it touches, and whether it touches them in the middle or at a side gate. Then, assuming all of the seas were simply connected, we will need to determine whether it is possible, by opening lock gates, to allow a boundary loop to be drawn.

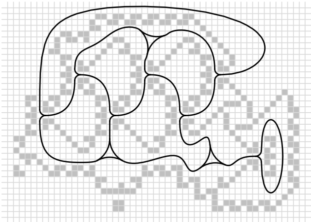
We will think of the seas and locks as a combinatorial graph[9], with special vertices for the locks (we will call them "switch" vertices), and edges connecting the locks through the seas, representing possible routes for a boundary. If a sea touches more than two locks, we may use any tree of edges and vertices to represent the sea.



Here is a pattern in which all the seas are simply connected. We want to know if lock gates can be opened so as to permit a boundary that will divide the pattern into two stable sets of islands.



In each sea, we draw lines connecting up the lock parts it touches. When going through a lock, the boundary must follow one of the smooth curves rather than passing straight across. Like a train on a track, it cannot have a cusp in its route.



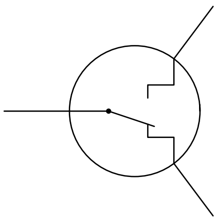
We can convert the graph to a form which only uses "train track branch" vertices, by first drawing each sea as a trivalent tree connecting its locks as in the previous diagram, and then replacing each tree vertex with a triangular branching arrangement as shown here.

The picture shows how such a graph can easily be turned into one in which *all* vertices are switch vertices, in $O(n)$ time, without affecting the question of whether there is a cycle or not. The question of whether such a graph contains a cycle is a problem that we will call *Switch-Cycle*.

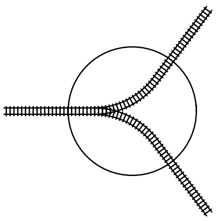
Everything up to now has only taken $O(n)$ time, but *Switch-Cycle* will take a little longer. Fortunately, at this stage of processing, almost all small to medium size patterns have already been classified, and only very large patterns full of switches will present nontrivial questions for *Switch-Cycle*.

2.2 An $O(n^2)$ Algorithm for *Switch-Cycle*

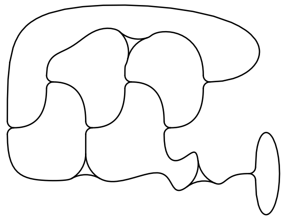
A *switch graph* is like an ordinary combinatorial graph, but some of its vertices may be “switches”, which are vertices of degree three (touching three edges) which “prefer” one of their edges. A switch vertex acts like a switch, in that it can connect its preferred edge to exactly one of the other two edges, leaving the third edge disconnected. A switch graph specifies the preferred edge for each switch vertex, but does not specify the settings of the switches, i.e. it does not specify which other edge each preferred edge should be connected to.



A switch vertex, enlarged here to show hypothetical detail, can be thought of like a mechanical switch that connects its "preferred edge", shown here coming in on the left, to exactly one of the other two edges.



Another way to think of a switch is like a branching train track. Just as a train must use the track on the left, and may not go from the upper right track to the lower right track, a switch cycle's route must obey the same restriction.



The train track version is especially easy to draw and read, so we will use this approach when drawing a switch graph. At each branching, a cycle must pass in a smooth, rather than cuspy, manner. So the question is, is there a smooth loop?

Given a switch graph, we are interested in the following question: Can the switches be set so that the resulting connections contain a cycle? In other words, is there a cycle such that for each switch vertex in the cycle, the preferred edge of that vertex is also in the cycle? We will call such a cycle a “switch cycle”, and we will call the problem of determining whether such a cycle exists the *Switch-Cycle* problem.

We will prove a theorem that will aid us in answering this question: Consider a switch graph in which every switch is replaced by an ordinary vertex of degree three. We will call this the *normalization* of the switch graph. Then, the theorem is the following:

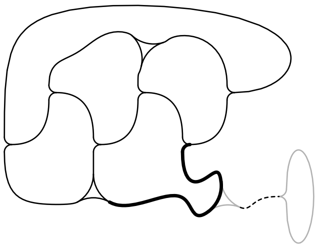
Theorem 1 *If the normalization of a switch graph contains no bridges (no cut edges), then the switch graph contains a switch cycle.*

We will accept the theorem without proof for now, postponing its proof to the end of this section.

A bridge (also known as a cut edge) is an edge of a graph that is "the last straw" connecting two parts of the graph, meaning that if the edge were to be removed, the graph would become disconnected. Clearly, such an edge cannot be part of a cycle, since there would be no way for the rest of the cycle to connect the two ends of the edge. So if the graph has any bridges, they may be removed without affecting the question of whether there is a cycle in the graph, or a switch cycle in the switch graph.

What the above theorem tells us, then, is that if there are no more bridges to remove, then the graph must contain a switch cycle! So there is a very simple algorithm for *Switch-Cycle*: As long as there are bridges, remove them. If anything is left at the end, then there is a switch cycle.

When we remove an edge in a switch graph, we have to look at how it might have been connected to switches. If it was the preferred edge of a switch, then without it, the other two edges of that switch cannot connect to anything, since they could only have connected to the preferred edge. So in this case, the other two edges should be removed as well. On the other hand, if it went to a switch but was not a preferred edge, then once it is gone, there is no reason for the switch not to connect the remaining two edges, so in this case, the other two edges should be merged into one long edge, removing the switch. (If the other two edges were in fact two ends of the same edge, then a cycle exists using that edge alone, and we are done.) So in either case, the switches at the ends of the original edge are removed, and possibly other edges are removed as well ("dependent edges"), leading in turn to more switches and edges being removed. All in all, the total amount of time required is proportional to the number of edges that are removed.



This graph contains a bridge, shown dashed, that we would like to remove. After removing it, its three dependent edges, shown in gray, must be removed as well. Then three of the remaining edges, shown in bold, must be merged into one long edge.

The resulting graph does not have any more bridges, so our theorem tells us that it must contain a switch cycle.

Identifying the bridges in a graph is a well known problem [10] that takes only $O(n)$ time using a simple depth-first search strategy. So repeatedly identifying the bridges and removing them (along with any dependents) can only take at most $O(n^2)$ time.

All that remains is to prove our useful theorem.

We will just consider graphs in which all vertices are switch vertices, since as we saw at the end of Section 2.1, we can convert any switch graph to such a form in $O(n)$ time without affecting whether the graph has bridges or has a switch cycle.

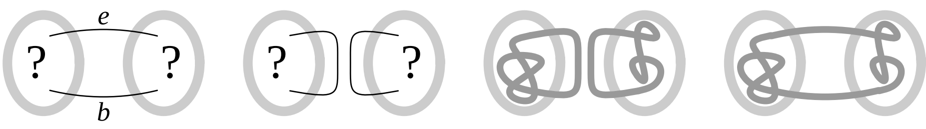
First of all, note that in a graph with no bridges, it cannot be the case that both ends of an edge meet at the same switch, since then the third edge going to that switch would be a bridge.

Now, in a graph with no bridges, we will say that a *free* edge is any edge which is not the preferred edge for either of the switches it touches. A free edge is called this because it doesn't have any dependents, and can be removed without forcing any other edges to be removed. (Its neighbor edges would merely be merged.)

Note that every switch graph must have a free edge, as a simple pigeonhole counting argument shows: If we look at the ends of all the edges, noting which are and aren't preferred, we will of course find that only one third of the edge ends are preferred, since each switch prefers one of its three edge ends. Since two thirds of all edge ends are not preferred, there must be some edge for which both ends are not preferred, i.e. a free edge.

Now, suppose that the theorem is not true. Then there must be some switch graph which has no bridges, and yet does not contain a switch cycle. Let us consider such a graph G which is minimal, i.e. has at least as few edges as any other such graph. We will call this graph a minimal counterexample to the theorem.

Pick a free edge e. If we remove e, then one of two things must happen: Either we get a smaller switch graph, or, after merging the neighbor edge ends, we just get a loop with no switches in it at all. In the latter case, the loop is a cycle which also existed as a switch cycle in the graph before removing e, contradicting the assumption that G has no switch cycle. So it must be the case that removing e leads to a smaller switch graph, which we know must have a bridge b, since otherwise it would be a smaller counterexample.



We know that G can be drawn as two components connected only by e and b. *We can cut G up into two smaller graphs, each "short-circuiting" the other component.* *If only the smaller graphs, but not G, have switch cycles, then they must be using the short-circuits.* *But then we can reconnect the components and see that G has a switch cycle too.*

So G must consist of two components which are connected only by e and b. If we look at just one of the components, and attach its dangling edges e and b to each other to become a single edge, then there cannot be any bridges, since they would also be bridges in G. Therefore, it must have a switch cycle, or else it would be a smaller counterexample. If the switch cycle does not use the merged e-b edge, then G contains the same switch cycle, contradicting our assumption, so the switch cycle must use the merged e-b edge.

The same must be true for the other component: It must have a switch cycle using its merged e-b edge. But this means that if we put the two components back together to form G, then we can merge the two smaller switch cycles along e and b to form one big switch cycle, again contradicting our assumption that G has none.

Since the assumption of a counterexample leads in all cases to a contradiction, there must not be any such counterexample, and so the theorem must be true.

If we want an algorithm to actually find a switch cycle if there is one, rather than just determining that there must be one, we can use the above proof as a recursive algorithm: Where the proof appealed to smaller counterexamples, the algorithm would recurse, finding and using switch cycles present in the smaller graphs to exhibit an explicit switch cycle in G, again in only $O(n^2)$ time.

3 A More Intuitive Definition

The conventional definition given in Section 1 for a pseudo still life requires that the pattern be decomposable into exactly two sets, each of which is stable alone. But it turns out that there are many patterns which don't fit this definition, even though they are distinctly composed of a collection of still lifes! In the last figure of Section 1, we saw some instances of such patterns: The "sleigh", the "fragile four", and the "two snakes, two hats, and a block" are all examples of patterns which cannot be divided into exactly two stable groups of islands, even though they can be divided into three or four stable groups. We would naturally like to think of these patterns as being pseudo still lifes, just like the "quad block", since they are close collections of stable islands. But the conventional definition classified them as strict still lifes, since it only considered partitions into exactly two groups.

A more natural definition immediately suggests itself, namely:

Definition 2 (Natural Pseudo Still Life) *A natural pseudo still life is any stable pattern whose islands can be partitioned into two or more nonempty sets, each of which is stable on its own.*

This definition captures the idea that a stable collection of stable objects should be considered a pseudo still life, and only indivisible collections of islands should constitute strict still lifes.

One can see that the "sleigh" needs to be divided into three parts in order for each part to be stable, and the "fragile four" needs to be divided into four parts. Is there some pattern that requires five or more parts? Perhaps surprisingly, the answer is no. For suppose you can divide a pattern into five or more stable subsets. Then the following process will rearrange the islands into just four stable subsets:

First, enlarge each island by half a cell, so that islands can border each other. (Any cell with no live neighbors remains "international waters".) Call each group of contiguous islands of the same set a "country". (Where two enlarged islands touch only at a point, we do not consider them to be contiguous.) Note that each country is independently stable. Note also that if two countries meet just at a point, then they will each be stable regardless of whether their islands are in the same group or not.

Now we have a map that the four color theorem[11] applies to. The four color theorem guarantees us that we can color the countries with just four colors so that any two bordering countries are different colors. Given such a four-coloring, we can put each island into a set according to the color of its country.

This means that where two abutting enlarged islands used to be in the same set (and therefore the same country), they will still be in the same set (since they will both be in the set for that country's color), and where they used to be in different sets (and thus in different countries), they will still be in different sets (since neighboring countries get different colors in the four-coloring).

In places where the enlarged islands of two countries touch at just a point, they will be stable regardless of whether the countries wind up having the same color or not, since the rules for Life are such that it makes no difference to stability whether diagonally opposite islands are assigned to the same set or not.

So the four color theorem is sufficient for showing that allowing divisions of the islands into any number of stable sets is no different from allowing divisions into at most four stable sets.

So an equivalent form of our more natural definition would be:

Definition 3 (Natural Pseudo Still Life, again) *A natural pseudo still life is any stable pattern whose islands can be partitioned into two, three, or four nonempty sets, each of which is stable on its own.*

The complexity of detecting whether patterns are natural pseudo still lifes will be left until Section 5.

4 NP-Completeness of the "Three Set" Definition

In the last section, we proposed a definition that concerned partitionability into two, three, or four proper subsets. The standard definition concerns itself only with partitionability into exactly two proper subsets. Clearly we could also have an intermediate definition which considers partitionability into *two or three* proper subsets.

Although neither conventional nor intuitive, this intermediate definition does have an interesting property: We are able to show that the complexity of determining whether a pattern is a strict still life according to this definition is

NP-complete. In other words, this determination requires[4] exponential time to compute.

We will devote the remainder of this section to this proof.

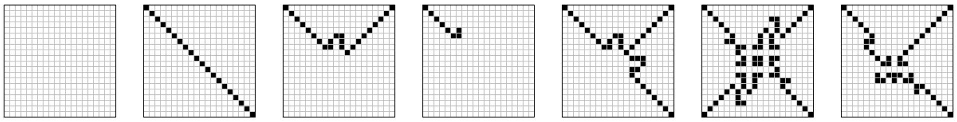
First of all, the problem is clearly in NP, since if we are given a pattern which is already marked with a proposed partitioning of the islands into two or three sets, we can easily verify in polynomial time whether the proposed partitions would be stable. So a nondeterministic computer could in effect try all possible partitioning schemes in parallel (or use an oracle to just try the best one), and in polynomial time it would know whether there is a stable partition.

To show that the problem is not just in NP, but NP-complete, we will show that if we could solve this problem in polynomial time, then we could also solve a known NP-complete problem, *CNF Satisfiability*, in polynomial time.

CNF Satisfiability is a well known problem which asks, given an expression in "Conjunctive Normal Form" (e.g. $(a \vee b \vee \overline{d}) \wedge (\overline{a} \vee \overline{c}) \wedge (\overline{b} \vee e \vee \overline{f} \vee \overline{g}) \wedge \cdots$, where the whole expression is a conjunction of terms each of which is a disjunction of variables or their negations), does there exists a set of boolean values for the variables which results in the whole expression being true?

Given a *CNF Satisfiability* problem, we will construct a pattern which is either a pseudo or strict still life, depending on whether there is a solution to the *CNF Satisfiability* problem or not.

We will use simple 20×20 building blocks to build our stable Life pattern. These building blocks will have "wires" of cells, and the "value" of a wire will be the set that it belongs to. If we like, we can think of each of the at most three sets of islands as having a distinct voltage level, constant along wires and where switches connect them.



Empty Space | *Straight Wire* | *Turning Wire* | *Wire End* | *Wire Junction* | *Wires Crossing* | *Simple Switch*

We recognize the last diagram as being the same kind of switch connection as we discussed in Section 2.2. In it, the wire coming in from the lower right must be in the same set as (i.e. connected to) either the upper right or the upper left, with the other upper wire being free to belong to any set (i.e. disconnected).

The second to last diagram, of two wires crossing, needs some explanation. In it, the wires at opposite corners must belong to the same set, and which set one pair belongs to is independent of which set the other pair belongs to.

To see this, the first thing to notice is that if any wire belongs to the same set as the center block, then the next wire clockwise must also belong to the same set, in order to stabilize a cell diagonally adjacent to the center block. This in turn will force the next wire after that to belong to the same set too, and in the end all four wires together with the center block must belong to the same set.

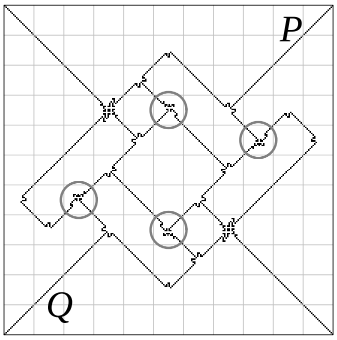
[4]Given the current (and, in most people's opinion, any future) state of the art in computational complexity theory.

The other possibility is that the center block does not belong to the same set as any of the wires. In this case, we notice that two adjacent wires may not belong to the same set, since then a cell diagonally adjacent to the center block would be unstable. Since there are only three possible sets, this means that the center cell must belong to one set, and the wires must then alternate between the two other sets, so that opposite wires belong to the same set.

So in summary we see that no matter what, opposite wires must belong to the same set. Furthermore, if one pair of opposite wires belongs to set i, and the other pair belongs to set j, then we can have either $i = j$ (in which case the center block belongs to the same set too), or $i \neq j$ (in which case the center block belongs to the third set, not i or j). So we see that the diagram does indeed correspond to two independent crossing wires.

What's great about these components is that we can use them to build up a switch graph. In Section 2 the switch graph represented possible routes for a dividing boundary, but here the switch graph directly represents the islands of the pattern. Now, if a switch connects two wires of the pattern, it means that those wires must be assigned to the same stable partition. So for the pattern to be a pseudo still life, we must be able to set the switches so that the pattern becomes disconnected, consisting of disjoint pieces not connected by any switch settings. The question of whether a switch graph can have its switches set so that it becomes disconnected is a problem that we will call *Switch-Disconnected*.

The first thing we will build out of our 20×20 building blocks is the following symmetrical arrangement of four simple switches:



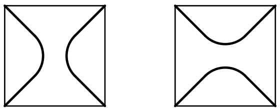
A symmetric arrangement. The simple switches are circled. For each of the four wires coming from the corners, there is a switch that forces it to be in the same group as at least one of the two corner wires perpendicular to it. For example, the simple switch on the right says that at least one of P or Q must be in the same group as the lower right wire.

Say the upper right wire is in set P and the lower left wire is in set Q.

If P and Q are the same set, then the left simple switch forces the upper left wire to be in this set too, and the right simple switch forces the lower right wire to also be in this set, so all four wires must be in the same set.

Now suppose P is different from Q. Consider the upper left and lower right wires. The upper simple switch says that one of them must be P, while the lower simple switch says that one of them must be Q. So they must be different, one being P and one being Q. But which is which? One can easily check that the simple switches allow either possibility.

So this is almost the opposite of the "wires crossing" block, in that this forms *non*-crossing connections. This is a essentially a fancy kind of switch, that forces the four wires to have at least one of the following two possible connectivities:

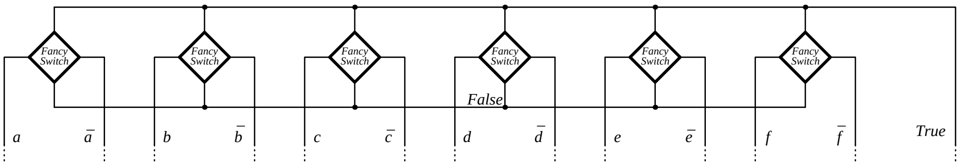


The two possible effective connectivities for a Fancy Switch.

Now, with this Fancy Switch, we are ready to start constructing a pattern for the *CNF Satisfiability* problem.

We will start by having a row of Fancy Switches, one for each variable. A bunch of wires will dangle down, where we will add things onto them.

This diagram is rotated 45 degrees from the previous orientation.



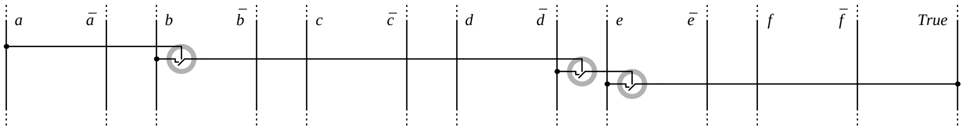
We will name the upper wire "$True$" and the horizontal wire underneath all the Fancy Switches "$False$". $True$ and $False$ might wind up in the same set, or they might wind up in different sets, when we partition the pattern into stable sets.

The two wires dropping out from the sides of each Fancy Switch will be called x and $\overline{x}$ for the Fancy Switch corresponding to variable x.

If $True$ and $False$ belong to the same set, then all the x and $\overline{x}$ wires will also belong to this set. If $True$ and $False$ belong to different sets, then the Fancy Switch for x either puts x in $True$'s set and $\overline{x}$ in $False$'s set, or else it puts x in $False$'s set and $\overline{x}$ in $True$'s set.

Now we need to represent the terms of the big conjunctive expression.

This turns out to be very easy: We can use simple switches to build up each term. As an example, we'll look at the term $(a \vee b \vee \overline{d} \vee e)$.



The rightmost simple switch says that either e must be in $True$'s set, or else... the next switch says that either $\overline{d}$ must be in $True$'s set or else... either b must be in $True$'s set or a must be in $True$'s set.

In other words, supposing $True$ and $False$ are in different sets, we can think of each boolean variable x as being true iff its wire is in the same set as $True$ (which is the same as the $\overline{x}$ wire being in the same set as $False$). So the previous paragraph is just saying that $(a \vee b \vee \overline{d} \vee e)$ must be true, and this is exactly the term we wanted to represent.

We continue down with all the other terms of the big conjunctive expression, representing them in this way, one after the other, until at the end we just terminate all the dangling wires, at which point we are done making the pattern.

If we continue supposing that $True$ and $False$ are in different sets, then we see that a stable partition corresponds perfectly to a solution of the *CNF Satisfiability* problem.

If $True$ and $False$ are in the same set, then one can quickly verify that all the wires in the diagram, and therefore all the islands in the diagram, must belong to this same set. (Recall that the only islands that aren't wires are the blocks at the middle of wire crossings.) So there is no way to stably partition the islands into two or three proper subsets if $True$ and $False$ are in the same set.

So, for the definition proposed at the beginning of this section, the pattern we have constructed is a pseudo still life exactly when there is a solution to the *CNF Satisfiability* problem, and it is a strict still life exactly when the *CNF Satisfiability* problem has no solution.

And, as a bonus, our method of proving this has also shown that *Switch-Disconnected* is an NP-complete problem.

5 NP-Completeness of the Natural Definition

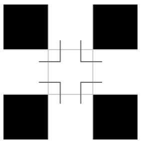
In this section, we will show that the problem of determining whether a pattern is a pseudo still life according to the natural definition proposed in Section 3 is an NP-complete problem.

We will use techniques similar to those of Section 2 to convert the problem into a switch graph problem, that of detecting whether a specific layout of a switch graph in the plane can contain a simple loop that doesn't cross itself. We will call this problem *Switch-Simple Loop*.

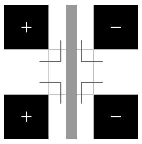
Then, using ideas similar to those in Section 4, and based on ideas of Mazzoni and Watkins [12], we will show that *Switch-Simple Loop* is NP-complete.

5.1 Boundaries Between Stable Groups

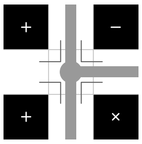
We are faced with the question of whether a given stable pattern can have its islands partitioned into stable sets. As in Section 2, we will look at the boundaries between islands that are in different sets, and see how these boundaries can be connected. Since there can be as many sets as we want, rather than just two, we will be searching not just for a cycle, but for a more general network of boundaries, dividing the plane into countries like the ones discussed in Section 3. Since countries do not cross each other, our network of boundaries will not need to have any boundaries crossing any other boundaries, and so instead of the crossing aquaducts of Section 2, in their place we will have a special kind of infrastructure that allows three or four boundaries to meet, or one to pass through, but will not let a single boundary make a turn. We will call this piece of infrastructure a "bend-preventer".



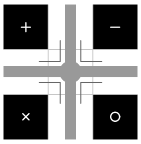
This is how we will draw a bend - preventer. They will go where aquaducts went.



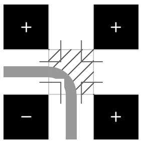
A bend - preventer allows a boundary to go straight through it.



It also allows a T junction joining three boundaries.



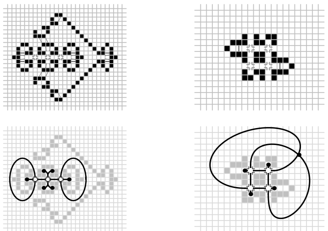
Four boundaries can also meet at a bend - preventer.



Bend - preventers do not allow a boundary to bend, creating an unstable set.

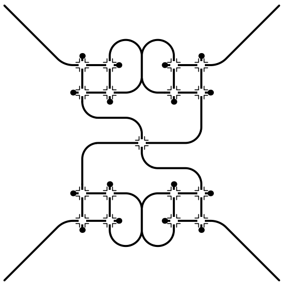
As in Section 2, we can determine the seas, making a graph in which bend-preventers and locks are connected to one another via connecting seas. Since we no longer have crossing aquaducts, the graph will be planar.

Here is a pattern with some bend - preventers, and a graph of sea connections between the bend - preventers. All of the edges except for the circular loops are bridge edges, and so they cannot be used by a network of boundaries, since they would be bridges in the boundary network as well, and the land on both sides of a bridge must belong to the same set.



The pattern of two snakes, two hats, and a block is just four bend - preventers arranged like this. If any of the sides of the square are used by a boundary, then the next bend - preventer going counter - clockwise must contain a T junction of boundaries. Continuing around the square, we see that all sides and departing edges must be boundaries. Therefore, if any of the emanating edges is a boundary, then every one of them must be a boundary.

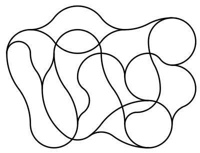
Our task will be to see whether our graph can contain a network of boundaries subject to the simple constraint that the network must not contain any bridges (cut edges). The reason bridges are prohibited from the network of boundaries is because both sides of the bridge would correspond to the same country, which means there shouldn't be a boundary there.



Each of the four edges entering this arrangement can be a boundary iff all four edges emanating from the square it leads to are boundaries. The two upper squares are connected by a pair of switches that ensures that only one of the squares may be used by boundaries. The same is true for the two lower squares, so the central bend - preventer cannot have more than two boundaries coming to it, and so it will only allow boundaries to go straight across it.
This means that of the four edges entering the arrangement, only two opposite edges may have a boundary. We will call this arrangement a "blocking crossing".

The above figure shows a wonderful configuration which permits boundaries to come out of it either along one set of diagonally opposite edges, or along the other set of diagonally opposite edges, but not in any other way. If we think of this as two paths crossing, we see that at most one of the two paths may be used. We will call this a "blocking crossing", since if it is crossed in one direction, that effectively blocks its use in the other direction.

Given any planar graph containing only blocking crossings and switches, it should be clear that we could construct a big stable Life pattern corresponding to it. Such graphs are like switch graphs, except that they have a specific layout in the plane, and where edges cross each other, they do it with blocking crossings. We will call this kind of graph a "switch graph layout".



Here is an example of a switch graph layout. The question we are interested in, given such a graph, is whether it contains a loop that doesn't cross itself. In the next section, we will show that this is an NP-complete problem.

We will call the problem of finding a loop in a switch graph layout *Switch-Simple Loop*. In Section 5.2, we will show that *Switch-Simple Loop* is NP-complete by designing a big switch graph layout corresponding to a given *CNF Satisfiability* problem so that if there is a solution to the *CNF Satisfiability* problem, then there is a loop in the switch graph layout, but if there is no solution to the *CNF Satisfiability* problem, then there is no loop and no stable network of boundaries in the switch graph layout.

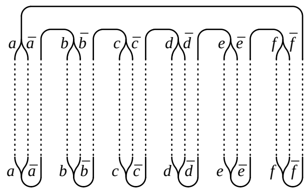
As it turns out, if there is a boundary loop in the switch graph layout, then the islands will be partitionable into three stable sets, and a fourth is not needed. The two main sets are the islands inside and outside the loop, and a third set is sufficient for keeping blocking crossings stable where they are traversed by the boundary loop. This fact will not affect the proof, but it means that regarding still lifes, this proof is stronger than the one in Section 4.

5.2 *Switch-Simple Loop* is NP-Complete

The basic approach of this section is inspired by a proof of Mazzoni and Watkins [12].

We will show that given a *CNF Satisfiability* problem, we can make a *Switch-Simple Loop* problem which has a solution exactly when there is a solution to the *CNF Satisfiability* problem.

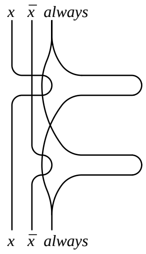
Our *Switch-Simple Loop* problem will take the following overall form, in which choosing a value for a variable is represented by choosing one of two vertical paths stretching from the top to the bottom of the picture.

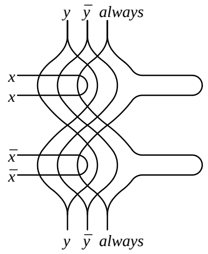


We will construct a Switch-Simple Loop problem corresponding to the CNF Satisfiability problem by creating a switch graph layout whose overall form will be as shown here. The resulting Switch-Simple Loop problem will only be able to have a loop if the loop follows this overall path, effectively picking a value for each variable. We will modify the central dotted portion so as to represent the terms of the CNF expression.

In order to represent the terms of the CNF expression, we will need to be able to "push" the values of variables horizontally out to the right, where we can make them interact according to the CNF terms. We can do this by making the vertical paths protrude slightly out to the right so as to interfere with neighboring columns. A pair of protrusions will propogate a variable's value: The upper protrusion will be part of the loop if the variable is true, while the lower protrusion will be part of the loop if the variable is false.

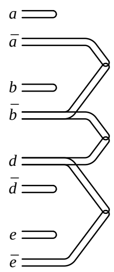
This shows how we can alter the three vertical paths corresponding to a variable in the previous diagram so that the variable's value is pushed out to the right. The third vertical path, which is always used by the loop, must use one of two branches here. If the variable is true, then the loop must use the branch with the upper right protrusion, since the other branch crosses the first path, which is used by the loop when the variable is true. Similarly, if the variable is false, the loop must use the lower protrusion on the right.





This shows how we can keep pushing a variable's value to the right. If the upper protrusion coming from the left is used by the loop, then the vertical paths used here will have to swing out to the right to avoid it, and swing left across the unused protrusion. Since the third vertical path is always used, the upper right protrusion must be used. Similarly, if the lower protrusion from the left is used, then the lower protrusion to the right must be used.

Once the values of the variables for a CNF term have been propogated all the way to the right side of the diagram, we can implement the CNF term by ending those protrusions corresponding to the values appearing in the CNF term, leaving the negations of those values to play a game of musical chairs, where if there are n such negations, then they must share $n-1$ positions. This is impossible if all of the negations are part of the loop, but if any of them are not (that is, if any of the values in the CNF term are true), then the rest can "lean towards" the unused negation, and there will be room for all of them to coexist.



This shows how we can implement a disjunction of the CNF expression, such as "a or b or not d or e", after pushing all the relevant variables out to the right. If one of the terms, say b, is true, then only the stubby path will be used there, and so if any of a or not d or e is false they may use the long path that leans towards b, and they will not cross each other. But if none of the terms is true, then there will not be enough room for all the falsehoods to be able to coexist without crossing each other.

In this way, we can represent each CNF term, one after the next, down through the switch graph layout. When we are done, we have a switch graph layout with the property that if any part of it is used as a boundary, then the boundary must follow the overall form that we intended, effectively choosing values for the variables that solve the *CNF Satisfiability* problem.

Therefore the *Switch-Simple Loop* problem has a solution exactly when the *CNF Satisfiability* problem has a solution, and these questions are the same as the problem of determining whether the pattern corresponding to the switch graph layout is a natural pseudo or strict still life.

Since both *Switch-Simple Loop* and the natural pseudo still life question are problems for which a positive solution can be easily verified, we have proved that these problems are NP-complete.

6 Conclusion

What started out as a practical problem, creating an efficient "object analyzer" for stable regions in the Game of Life, turned out to be quite a complex problem, giving rise to new questions, new methods, and in general, a new area of investigation: Still Life Theory.

The various *Switch*-graph problems are also interesting in their own right, and one can easily come up with new and interesting problems in this area.

We found that testing patterns under the conventional definition for strict still lifes turns out to have quadratic complexity rather than exponential complexity as has been assumed by workers in the field. We proposed a more intuitive definition, but unfortunately it turned out to make testing patterns an NP-complete problem, thus making it unappealing for very large patterns. Since the definition is trying to capture a distinction that was originally thought to be obvious, one hopes that if not quite obvious, the distinction should at least be a tractable problem! An intermediate definition also resulted in pattern testing being NP-complete, so we see that the conventional definition is in fact uniquely amenable to use for testing large complicated patterns.

In a broader perspective, these results show that, in general, questions about decomposability of stable patterns for a cellular automaton can vary widely in complexity, and are likely to be very sensitive to the particular details of what kind of decomposition is required. And, as we have seen, the solutions to such questions can be very intriguing.

As a final remark, let us note that stable Life patterns are just one instance of a kind of two dimensional language called *local lattice languages*[13], which are characterized by having a finite set of locally allowable configurations. Conversely, for any local lattice language, we can easily construct a cellular automaton whose stable patterns are exactly the members of the local lattice language, so we see that speaking about the still lifes of a cellular automaton is equivalent to speaking about a local lattice language.

References

[1] John Conway wrote an exposition of Life, including a sketch of how one might build a universal computer within it, in *Winning Ways For Your Mathematical Plays, Volume 2*, by E. Berlekamp, J. Conway, and R. Guy (Academic Press, 1982).

[2] Achim Flammenkamp has a lot of statistics about the objects produced from a random initial state at:
http://www.uni-bielefeld.de/cgi-bin/php/~achim/gol.html

[3] H. Koenig has lists of possible small still lifes with constructions for many, as well as their frequencies of occurrence from a random initial state at:
http://www.pentadecathlon.com/LifeInfo/LifeInfo.html

[4] Mark D. Niemiec has lists of possible small still lifes as well as methods for creating many of them from gliders at:
http://home.interserv.com/~mniemiec/lifepage.htm

[5] Stephen Silver is the current maintainer of the Life Lexicon, which lists most of the Life terms you're likely to see, at:
http://www.cs.jhu.edu/~callahan/lexiconf.htm

[6] The "enclosed block" was designed (9/13/98) by Noam Elkies as the smallest example of one island enclosing another, after H. König. The "fragile four" is based on a pattern designed (9/27/98) by Gabriel Nivasch as one of the smallest known of this kind, after the author. The "switch" was originally found in the early 1970's, either by Peter Raynham or by David Buckingham.

[7] This definition was first proposed by Mark D. Niemiec.

[8] For an example of water traffic crossing a dam, see:
http://www.ils.nwu.edu/~eric/matt.html

[9] There are many good introductions to combinatorial graph theory, for example Chapters 6 and 7 of *Introduction to Combinatorial Mathematics* by C. L. Liu.

[10] For example, see chapter 23 and problem 23-2 in *Introduction to Algorithms* by Cormen, Leiserson, and Rivest.

[11] The original proof:
K. Appel, W. Haken, and J. Koch, *Every planar map is four colorable*, Illinois J. Math 21 (1977) 429-567.
And a more recent simpler one:
N. Robertson, D. P. Sanders, P. D. Seymour, and R. Thomas, *A new proof of the four color theorem*, Electron. Res. Announc. Amer. Math. Soc. 2 (1996) 17-25 (electronic).

[12] Dominic Mazzoni and Kevin Watkins wrote a proof that the problem of deciding whether a position in the game of Twixt is a winning position or not is NP-complete, currently available at:
http://www.mathematik.uni-bielefeld.de/~sillke/PROBLEMS/Twixt_Proof_Draft
Their proof effectively showed that the problem of determining whether there is a simple non-crossing path between two given points in a planar layout of a graph is NP-complete. Their proof very directly inspired the proof in Section 5 of the NP-completeness of *Switch-Simple Loop*.

[13] Local Lattice Languages are presented and discussed in:
K. Lindgren, C. Moore, and M. Nordahl, *Complexity of Two Dimensional Patterns*, Santa Fe Institute Working Paper 97-03-023.

Many of the items here refer to web pages, which unlike items published on paper and stored in libraries, can easily change or disappear without notice. To help ameliorate this, I will maintain a page of links to the web pages cited above at:
http://www.paradise.caltech.edu/~cook/Warehouse/StillLifeLinks.html