COMPUTING WITH TILES

Marsha Michie
Agnes Scott College

REU Summer Research 1989

Hao Wang first introduced what later came to be known as Wang tiles. Instead of fitting edges of tiles together by their shapes, he suggested that one might use a simple shape (usually a square), assigning each edge a different color. In this way one could speak of "tiling the plane," i.e., covering the plane without gaps or overlaps, by placing matching colored edges next to one another.

The same problem that arises when tiling by shapes applies to tiling with Wang tiles: Can one design an algorithm that, given a set of "prototiles," will correctly answer whether one can tile the plane with copies of those tiles? This problem is known as the "tiling problem," or alternately, the "domino problem," for obvious reasons. Branko Grunbaum and G. C. Shepard recount some of the history of this problem:

> Questions of decidability have long interested mathematicians, and about twenty years ago Hao Wang began an investigation into the decidability of the Tiling Problem. The following is a simplified version of his approach. He observed that if a set S of prototiles admits a tiling, then one of the following three possibilities must hold:
>
> 1. S admits only periodic tilings. The simplest example of this occurs when S consists of just one regular hexagon for this only admits the regular (periodic) tiling.
> 2. S admits both periodic and non-periodic tilings. This occurs, for example, if S consists of a square tile.
> 3. S admits only non-periodic tilings, in other words, is an aperiodic set.
>
> Wang then showed that the Tiling Problem is decidable if we only consider sets S which satisfy (1) and (2). He went on to conjecture (in 1961) that possibility (3) could not occur. Although,

> with the advantage of hindsight, we now know this conjecture to be false, at the time it was made it seemed quite natural — not only were no aperiodic sets known, but no one had any idea how such a set could be constructed.
>
> Berger's discovery of an aperiodic set in 1966 ... upsets Wang's argument, and in fact it is now known that the Tiling Problem is undecidable. [1]

The proof that no such algorithm can exist is dependent upon the undecidability of the 'halting problem,' and upon the universality of algorithmically undecidable problems. That is, the class of (algorithmically) undecidable problems is not dependent upon the language or the machine used in attempting to solve them. Alonzo Church and Alan Turing, working independently in the 1930's, arrived at the conclusion that all computers and all languages are equivalent in the class of problems that they can solve, given ample time and memory space. We will use this idea later in reducing the halting problem to the tiling problem.

The halting problem is this:

> Can one design an algorithm (call it S) that will do the following:
>
> - S receives an input pair <X,z>, where X is any program (in some specified language L) and z is any string of symbols.
> - S then decides whether X, given z as an input string, will ever halt. S answers "yes" if X halts, "no" if X will never halt.

The answer to this question is that such an algorithm cannot be constructed, and the proof follows:

> Suppose that such an algorithm can exist, and that we have implemented it some language L (call this program S). Now we can construct a new program P that will do the following:
>
> - P accepts as input some program X (in the language L).
> - Then, P makes a copy of X and activates the assumed-to-exist program S on the pair <X,X>. (We note that since X is a string of symbols, the pair <X,X> is a legal input to S.)

- If S returns a "yes," then P promptly enters an infinite loop. But if S returns a "no," then P immediately halts.

Now, we run the program P, using as input the program P itself. P submits the pair <P,P> to S, and S returns either a "yes" or a "no." But which? If S reaches the conclusion that P will halt, given P as input, P enters an infinite loop — i.e., P *doesn't* halt, given the input P. But if S decides that P will not halt, P immediately halts. We have now designed a machine that cannot halt, and cannot *not* halt, which is a logical impossibility. So P cannot exist, and therefore, S cannot exist, either.

Note that the proof that the halting problem is undecidable is not dependent upon the language of the program, or the machine upon which the program is run. This fact enables us to choose a suitable machine to utilize in relating the halting problem to the tiling problem, with the knowledge that the halting problem is, in fact, undecidable for that machine. The machine we will choose is the Turing machine, as described below:

The machine consists of a (potentially infinite) tape, divided into squares, a reading and writing head, and a finite number of states. Each square of the tape contains one of a finite number of symbols (possibly a "blank"). The action of the machine is completely determined by quintuples of the form

$$(q_i s_j s_k L q_l), \quad (q_i s_j s_k R q_l),$$

where $q_0, q_1, q_2 \ldots$ are the states of the machine, with q_0 the initial state, and $s_0, s_1, s_2, \ldots$ are the symbols, s_0 being the "blank" symbol. The quintuples are interpreted thus: If the machine is in state q_i and reads the square containing symbol s_j, it erases s_j, replaces it with the symbol s_k, moves one square to the right or left, and enters state q_l. We will deal only with *deterministic* Turing machines, meaning that for any state q_i and symbol s_j a quintuple beginning with that state and symbol must be unique.

In order to prove that the tiling problem is undecidable, we will show that the halting problem can be reduced to the tiling problem. To do this, we will show a system of designing tilings that correspond to a computation on a Turing machine. In this way, we reduce the question of whether the

machine will halt to the question of whether the set of prototiles we have designed will "halt," or fail to tile the entire infinite region. The Turing machine is especially useful in this respect, in that the configuration of the machine at any given time is completely represented by the internal state, the configuration of the one-dimensional tape, and the position of the head on the tape.

We will actually reduce the halting problem for Turing machines to a less general case of the tiling problem: Can we decide (algorithmically) whether a set $\mathcal{T}$ of prototiles, with a particular tile type t, will tile the upper half-plane, with the restriction that the special tile type t must appear somewhere on the bottom row? The reduction we perform here is at the heart of the more general proof, which may be found in [3]. The basic idea of the reduction is first to encode the initial (input) tape configuration, with the special tile t denoting the initial internal state and position of the machine head, on the bottom row. Then on each successive row we encode the next configuration of the tape, again denoting the internal state and position of the head. In this way, continuing computation on the machine corresponds exactly to continuing progress in the upward tiling. Clearly, then, we cannot decide whether an arbitrary such tiling will continue upwards indefinitely, for then we could answer the corresponding question of whether the machine will continue or halt. The set $\mathcal{T}$ is designed as follows:

> We will associate symbols, rather than colors, with each edge. The top edges of the tiles in the bottom row will then encode the input configuration of the tape. The special tile type t will encode the initial state and position, and the sides of the tiles will force the input to appear in exactly one way. For example, the input configuration
>
> | ··· | # | # | a | c | c | # | # | ··· |
> |---|---|---|---|---|---|---|---|---|
>
> (where '#' is the "blank" symbol), the initial state q_0, and the initial position of the head at the first non-blank character, could be encoded like this:

	#	#	q_0, a	c	c	#	#	
	0 0	0 0	0 1	1 2	2 3	3 3	3 3	

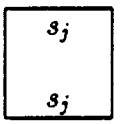


Figure 1:

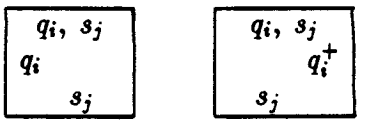


Figure 2:

$\mathcal{T}$ must also contain tiles that pass symbols and states up from one row to the next. Specifically, $\mathcal{T}$ must contain tiles that deal with the following situations:

1. A symbol is not being scanned, and passes unchanged from one configuration to the next.
2. The machine head "approaches" a symbol and in the given row (configuration) the machine is scanning that symbol.
3. The machine has scanned a symbol and is executing a change on both the symbol and its internal state.

Here is an easy way to design the above tiles:

1. For each symbol s_j, $\mathcal{T}$ will contain the tile in Figure 1,
2. For each symbol s_j and state q_i, $\mathcal{T}$ will contain the two tiles in Figure 2, and
3. For each quintuple $(q_i s_j s_k L q_l)$, $(q_i s_j s_k R q_l)$, $\mathcal{T}$ contains one of the two tiles in Figure 3.

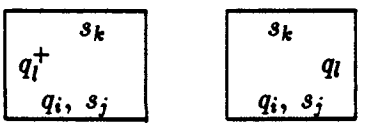


Figure 3:

···	#	**a**	*b*	*b*	*a*	#	···
···	#	#	**b**	*b*	*a*	#	···
···	#	#	*b*	*b*	*a*	**#**	···
···	#	#	*b*	**b**	#	#	···
···	#	**#**	*b*	*b*	#	#	···
···	#	#	#	*b*	**#**	#	···
···	#	#	#	**#**	#	#	···

Figure 4: The main configuations of a palindrome machine with input "abba." The boldface symbols are the ones being scanned. After the last configuration, the machine halts in state "YES."

> Figure 4 and Figure 5 show how the reduction from a Turing machine to a tiling works with a simple example: the "palindrome" machine. If the input word is symmetric with respect to its center (like "ababa" or "bb") the machine halts in the "YES" state; otherwise, it halts in the "NO" state.

This reduction is certainly useful in that it proves the undecidability of the tiling problem (here, a special case), but what of the tiling itself? We now have a method for producing a tiling — or a *non-tiling* — which can perform any calculation that a Turing machine can perform, and therefore can solve any effectively solvable problem. So, then, tiling may be viewed as a valid form of computer. As it turns out, there are other methods of computing with tiles besides the Turing machine simulation — methods that do, indeed, produce infinite tilings, rather than halting when a result is obtained. But computational tilings seem to divide themselves into two basic classes: those which carry out finite computation, and those which carry out infinite computation. The tiling of Figure 5 carries out a finite computation But a Turing machine can carry out an infinite number of computations, if the number of possible input configurations for that machine is infinite. So this type of tiling really does not simulate the entire machine, but instead corresponds to the machine's action on some particular input. Figure 6 shows another type of tiling which performs a finite computation on the quarter-plane. The tiling itself is, in fact, infinite, though the area in which the computation occurs is finite for any given input. This tiling also differs from the Turing machine simulation in that the prototiles are capable of doing the same computation for any appropriate input, and so are capable of an infinite number of computations. However, they can produce only one at

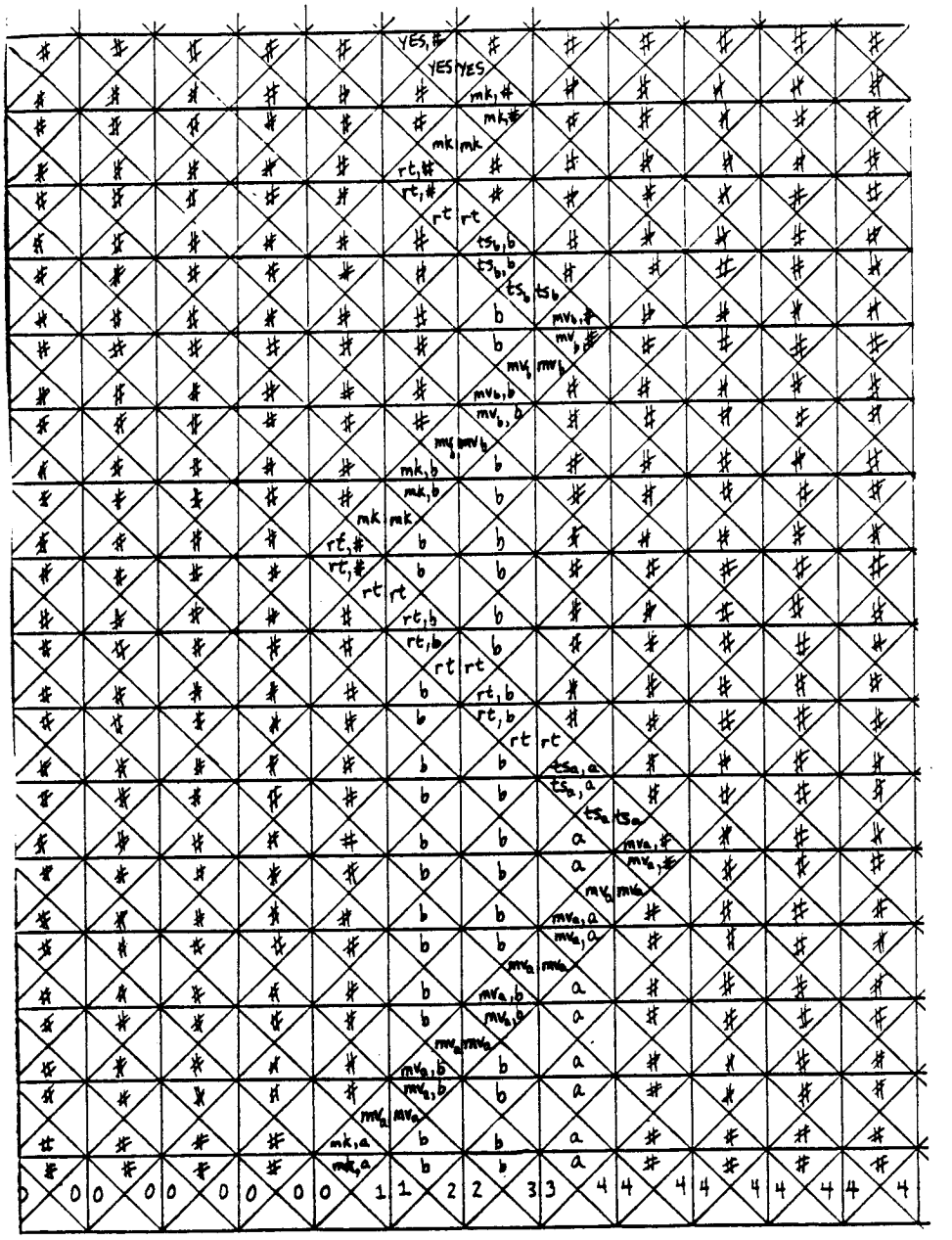


Figure 5: The tiling that corresponds to Fig. 4. (From [2])

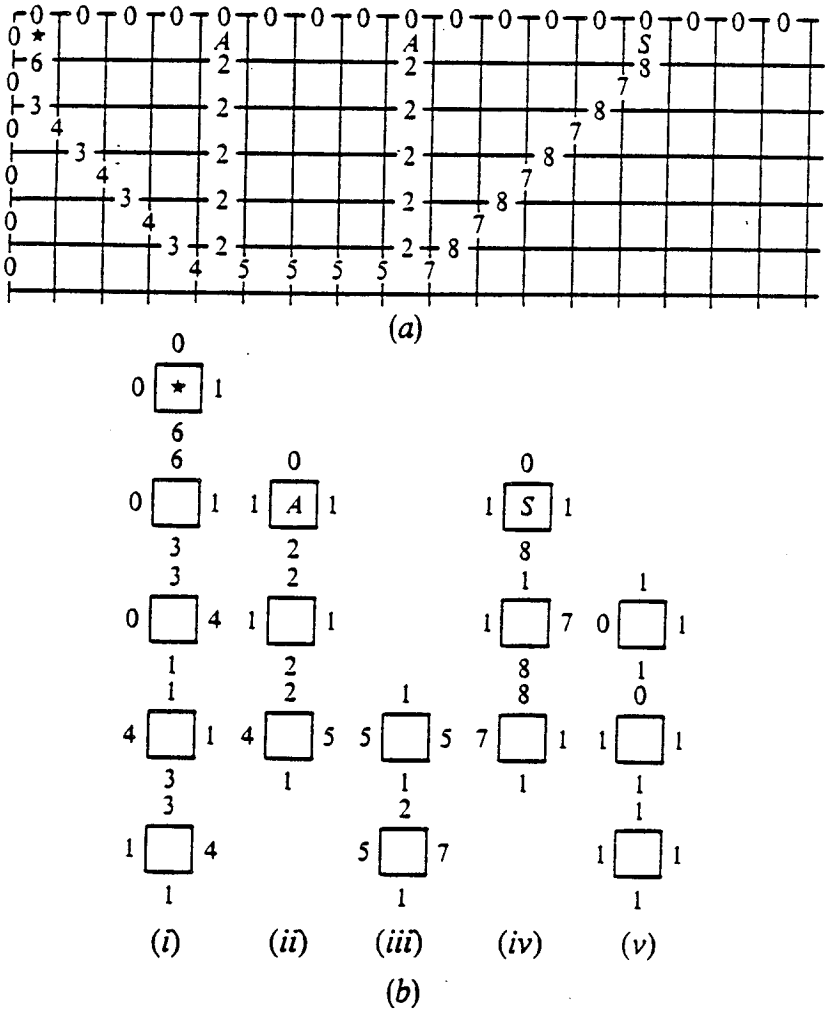
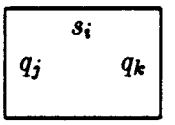


Figure 6: A tiling that performs the addition of two equal positive integers. From [1].

a time in a given quarter-plane area. Further, by imposing the requirement that the input must appear in a specified form (in the tiling of Figure 6, exactly two copies of the input tile must appear) we ensure that every tiling by the set of prototiles will produce the output tile in the correct position on the top row.

However, neither of the previous two tilings was actually equivalent to a general algorithm, independent of input. Is it possible to create a tiling that will contain the corresponding output to every possible input? This tiling would certainly carry out an infinite computation, and we clearly intend to limit the set of prototiles to a finite number of elements. Figure 7, a simple generalization of the addition tiling of Figure 6, indicates the Fibonacci sequence. The top row may be viewed as the output, where for every natural number the tile in that position indicates whether the number is a Fibonacci number.

However, we can use the top row to represent not only the naturals, but any countably infinite set. Possibilities include the set of strings over some alphabet. Then we could design tilings which perform the same tasks as finite-state automata. A finite-state machines define a language over some alphabet by the classes of strings that it "accepts." It is completely represented by a *state-transition diagram*, made up of an initial state, a set of final (accepting) states, other non-final states, and a transition function. Its input is a string of symbols, to which it reacts one symbol at a time. Beginning with the initial state, each symbol causes a transition to the next state. At the end of the string, if the machine is in one of the final states, the string is accepted and is therefore in the language defined by the machine. A tiling can easily simulate the action of a *deterministic* finite-state machine on a particular input string in a single finite row, using tiles of the form



, where s_i is the scanned symbol at some time t, and the state-transition diagram specifies that if the input symbol is s_i and the machine is in state q_j, the next state will be q_k. If we simulated the action of such a machine on every string over a given alphabet, we could describe the machine in a single row or half-row by stringing all the finite segments together. However, such a tiling would not be entirely deterministic, since we need to be able to adjoin each tile corresponding to the initial state to the end of any input string, and we would have to make some stipulations that each input string

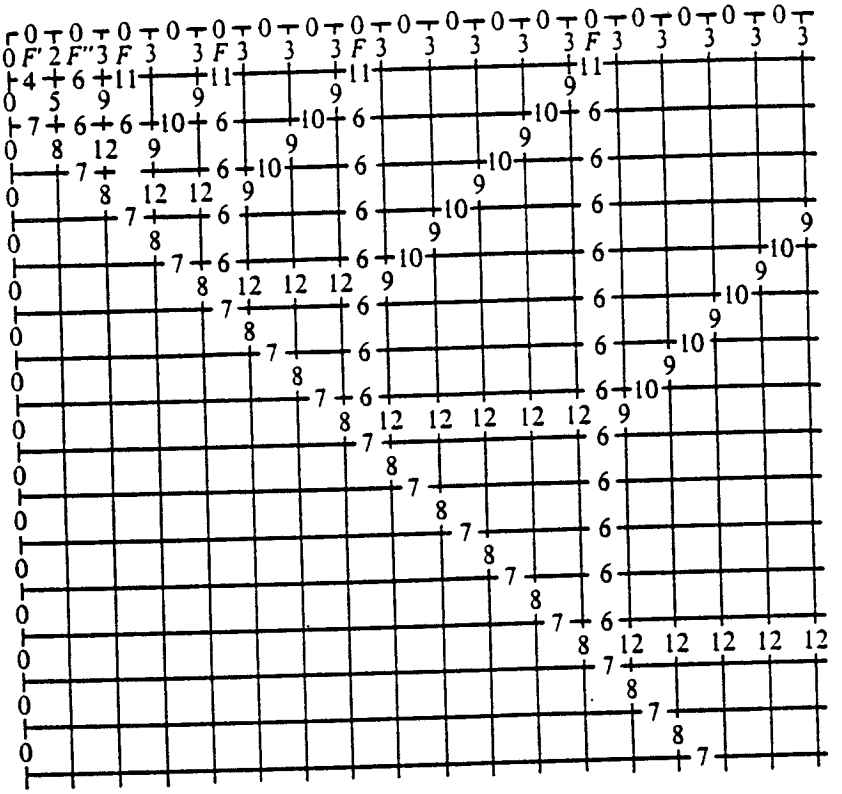


Figure 7: A tiling which indicates the Fibonacci sequence. From [1].

be entirely represented at least once. However, if we consider the simple case of a machine that acts on an alphabet of only one symbol, we can create a half-row tiling that is completely deterministic, and completely represents the action of the machine on every possible input string. The reason for this is that by ordering the set of strings by size (over the alphabet a, the set would be ordered a, aa, aaa, aaaa, ...), we can assign one tile to each string, so that the final state of the kth string is the state following the final state of the $k+1$st string. In this way, the row can be viewed as the set of ending states for each input strings, or as the progression of states for a single string of infinite length.

Bibliography

[1]Grünbaum, Branko and Shephard, G. C. *Tilings and Patterns.* W. H. Freeman and Company, N. Y., 1987.

[2]Harel, David. *Algorithmics.* Addison-Wesley Publishing, 1987.

[3]Robinson, R. M. "Undecidability and Nonperiodicity for Tilings of the Plane," *Inventiones Mathematicae*, vol.12, pp. 177-209(1971).