How common is Turing–universal behaviour?

Ming-Shr Lin, Gaurav Saxena, Viral Shah, Geoff Wozniak

August 16, 2002

1 Introduction and Motivation

Since the introduction of Turing machines as a model of computation, many other models of computation have been introduced and have consequently been found to be equivalent to Turing machines. While most computational models are artificial constructs, mostly for mathematical consideration, some are motivated by and modeled with natural systems.

The wide variety of computational models and their equivalence to Turing machines suggests that Turing–universal behaviour may be common to most systems, notably natural ones. Intuitively, we can guess that adaptable organisms resemble Turing–universal behaviour in that given proper input, they can perform a wide variety of tasks. In fact, it could be argued that this form of "Turing–universal behaviour" is an important factor to the survival of the organism. While this intuition is highly informal, it does lend itself to further exploration of the topic.

The goal of this report is to add an element of formalism to this argument. However, it must be made clear that it is not an argument that *shows* that Turing–universal behaviour is in fact common to many systems. Rather, it is an argument that *supports* the idea that Turing–universal behaviour may be common in some systems.

Our analysis will be based upon generating random Turing machine and looking for structure within the randomly generated machine (i.e., a directed, labeled graph) that is similar to that of a universal Turing machine. In general, this is very difficult to do, since we would be looking for behaviour, not structure. We will restrict ourselves to structure only and restrict ourselves even more by looking for a specific universal Turing machine instead of any universal Turing machine (UTM). The specific Turing machine in question is Minsky's (4, 7) machine from [1]. We will denote Minsky's UTM as U_M (see Figure 1).

2 Simple Counting Argument

Our first approach consisted of the simplest approach: assume the uniform distribution over all graphs on n nodes and count the number that contain

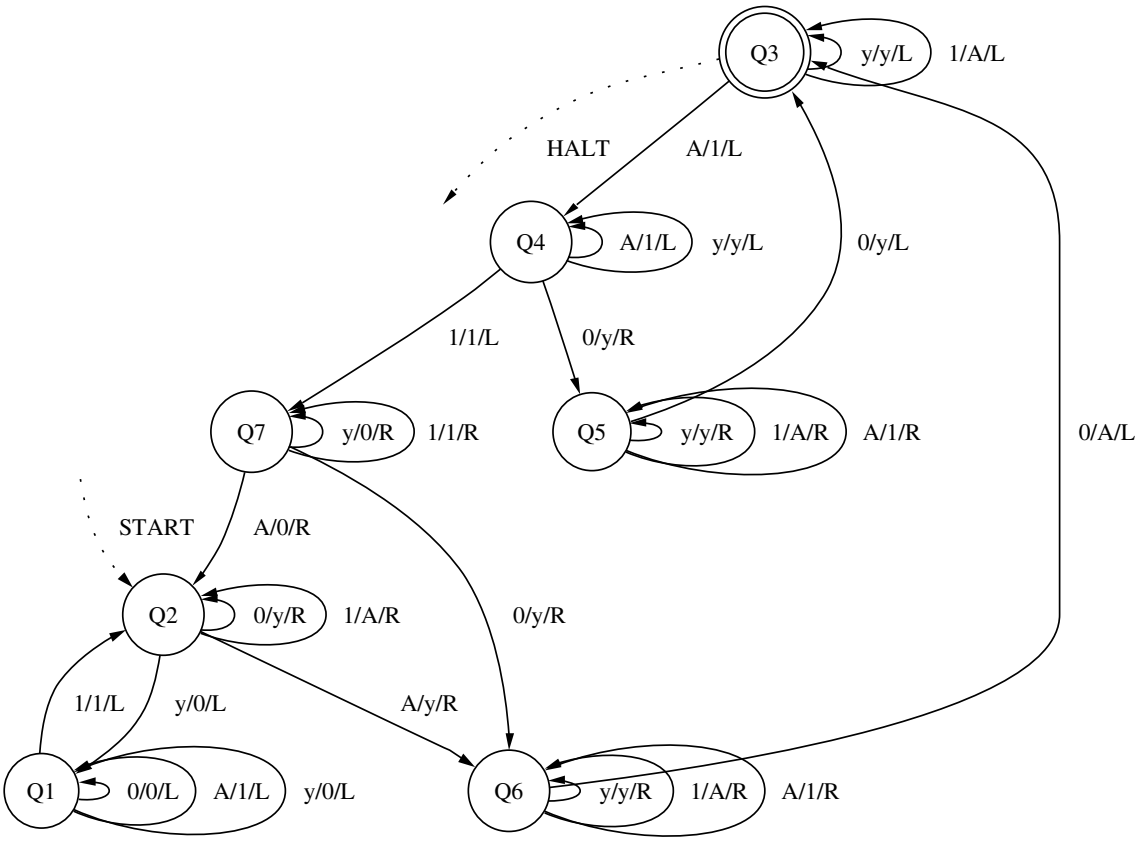

Figure 1: Minsky's (4, 7) universal Turing machine.

U_M. The graphs we considered were directed graphs that allowed up to three self–loops.

If we look at a graph as an adjacency matrix, there are a total of n^2 entries in the matrix. Not counting the diagonal, there are $n^2 - n$ entries, each of which may be set to either 0 (edge exists) or 1 (edge does not exist). Along the diagonal, we must allow up to 3 self–loops, meaning there are four possible entries: 0, 1, 2, 3. Thus, there are

$$2^{n^2-n} \cdot 4^n = 2^{n^2+n}$$

possible directed graphs on n nodes that must be considered.

Fix a numbering of the nodes in G and set U_M to be a connected component of G. Thus, there are $n-7$ nodes remaining. If we take the ratio of graphs on $n-7$ nodes with a fixed U_M as a connected component to the number of graphs on n nodes, we get

$$\frac{2^{(n-7)^2+(n-7)}}{2^{n^2+n}} = \frac{1}{2^{14n-42}}.$$

This ratio (i.e., probability) quickly approaches 0 as n goes to infinity. Even if we consider the $\binom{n}{7}$ ways of choosing 7 nodes and the 7! ways of arranging those nodes, it only results in a ratio of

$$\frac{7!\binom{n}{7}}{2^{14n-42}} = \frac{n(n-1)\ldots(n-6)}{2^{14n-42}}$$

which still results in a probability approaching 0 as $n \to \infty$.

These counting arguments grossly underestimate the correct number of graphs on n vertices that would contain U_M as a subgraph. This is due to the fact that we disallow any edges to come into the U_M component. However, this would not change the fact that the denominator is exponential with respect to n and the probability continues to tend to zero.

3 Forcing Locality

It would seem from the preceding section that Turing universality is not a very common trait. However, this was only in systems where each possible configuration (i.e., graph) is equally likely. If one observes natural systems for a short period of time, in all likelihood, it would be noticed that not every configuration is equally likely. So it would seem prudent to consider a system that favours some configurations over others.

To emulate this, we can change the distribution of probabilities on edges out of nodes. In the previous section, each edge appeared with probability 1/2. If that were narrowed to force each node to have out-degree 4 and labeled each edge, then we could weight the edges and assign probabilities based on these weights.

By looking at the Minsky machine, we see that there are many loops and the graph itself is small. Assuming we can find a weighting that favours edges going to nodes that are close to the source, it seems likely that the U_M structure will appear as the graph grows larger.

3.1 Probability of U_M in G

One aspect to locality is that as the system grows larger, it tends to make little difference as to the local connections and associations. What we are looking for is a probability distribution that favours locality and changes little as the total number of nodes grows. That is, regardless of how many nodes there are, a node x is just as likely to have an edge to a node close to x when the graph has a thousand nodes as it is when the graph has a million nodes. We would like to characterize distributions of this type.

In general, if we are looking for a specific subgraph of G, we can divide G into blocks of size l, where l is the number of nodes in the subgraph. If we assume $n = kl$, then we have k blocks of size l. Let B_i be a block, with $0 \leq i \leq k-1$ and let p_i be the probability that B_i is isomorphic to some fixed graph on l nodes. Then the probability of U_M (the fixed subgraph) being found in G is

$$\begin{aligned} Pr(U_M \subseteq G) &\geq Pr(\exists B_i \equiv U_M) \\ &= 1 - Pr(\neg \exists B_i \equiv U_M) \\ &= 1 - (1 - p_i)^k. \end{aligned} \tag{1}$$

Note that as $k \to \infty$ (assuming $p_i > 0$), then the probability of $U_M \subseteq G$ goes to 1. However, it must be the case that p_i does not depend on n. This is the crucial factor in using this argument and finding a usable probability distribution.

Designate an edge labeled by α between nodes x and y as (x, y_α). Let

$$D = e^{-|x-y_\alpha|/\lambda}$$

be a weighted distribution with parameter λ. We can see that D emphasizes locality since the size of the exponent is based on the difference (distance) between two nodes and not on the number of nodes in G and that weightings between two different edges are independent of each other. The distance between two nodes is based on their numbering (i.e., node x is considered to be a distance of r away from node $x + r$).

Since the overall goal is to find the probability of U_M in a random graph, we must normalize this weighted distribution to make it a probability distribution. That is, for some $x \in G$ and α

$$\sum_{y \in G} Pr((x, y_\alpha)) = 1.$$

To make a probability distribution for all possible edges from some node x labeled with α, we simply take the specific edge weighting over the sum of all possible weightings.

$$Pr((x, y_\alpha)) = \frac{e^{-|x-y_\alpha|/\lambda}}{\sum_{y' \in G} e^{-|x-y'_\alpha|/\lambda}} \tag{2}$$

Let a *configuration for some node* x be the set $C_x = \{(x, y_\alpha) \mid (x, y_\alpha) \in G\}$, that is, a configuration for node x is the set of all edges rooted at x. Since the probability of edges with different labels emanating from the same node are independent of each other, we have

$$Pr(C_x) = \prod_{(x,y_\alpha) \in C_x} Pr((x, y_\alpha)).$$

Let $\hat{p}_i$ be the probability of B_i being congruent to a specific isomorphism of U_M. We have

$$\hat{p}_i = \prod_{x \in B_i} Pr(C_x).$$

To get the value for p_i, sum the probabilities of $\hat{p}_i$ over all possible isomorphisms of B_i to U_M.

It is important to note that these calculations are entirely reliant upon the fact that p_i is not dependent on the number of nodes in G. For this to be true, it must be the case that the sum in (2) tends to a constant greater than zero. For example, we can consider another distribution

$$D' = \frac{1}{(x - y_\alpha)^2}$$

where x and y_α are the same as in D. As it turns out, this distribution is also favourable[1] since it converges to a constant greater than 0 and does not rely on the size of G. The reason that distributions should not rely on the size of G is that if it were, the locality argument breaks down since the probability of an edge will decrease as the size of G increases. When the sum converges, the probability of an edge tends to a constant greater than 0 as well.

3.2 Some Calculations

Empirically, we have shown that D works as a distribution.[2] We have shown that

$$\hat{p}_i \approx 1.1444 \times 10^{-19} \text{ for all } n \geq 20$$

[1] It turns out that G needs more nodes using D' as a distribution than D to have U_M appear with probability very close to 1.

[2] Matlab code for performing these calculations can be found in Appendix A.

where n is the number of nodes in the random graph G and setting $\lambda = 1$. Intuitively, it can be seen that the value of D is very small when x and y are very far apart and hence, the sum will converge as n is increased.

Since different isomorphisms produce different probabilities for $\hat{p}_i$, each $\hat{p}_i$ must be calculated for each isomorphism. However, it seems reasonable to estimate that

$$p_i \approx 7! \cdot \hat{p}_i \approx 5.767776e - 16$$

given that there are 7! ways of numbering the seven nodes in each B_i. In order for $Pr(\exists B_i \equiv U_M)$ to approach 1, we need a graph with approximately 10^{16} nodes. As p_i represents a lower bound, asymptotically, this is favourable.

4 Discussion

It is important to note that although the probability of U_M appearing in G tends to 1 as the number of nodes increases (assuming a local distribution), it does not say anything about whether or not U_M will actually get executed in G. That is, given G containing U_M, we do not know whether or not U_M is reachable in G on some given input (this is an undecidable problem). This demonstrates (one of) the shortcomings of our analysis. The analysis focused mainly on structure in randomly generated graphs representing Turing machines, however, very little about structure allows us to say much about behaviour.

Clearly, some fine points that could be expanded upon in the analysis have not been addressed. This was done mainly for simplicity. Most of the aspects that were not addressed would increase the probability. For example, our analysis only addresses a fixed choice for each block B_i and we did not consider all possible blocks in G. Also, we considered the nodes 1 and n to be very far apart. If we imagined the listing of a nodes to be a circle for the sake of locality, it makes a slight difference for the values of p_i for those blocks.[3]

Other aspects tend to significantly raise the complexity of the problem. We did not, for example, consider any other universal Turing machines besides Minsky's (4, 7) machine. Technically, there are an infinite number of these machines[4], however this becomes the problem of analyzing behaviour which, as we have stated, is difficult. Further analyses we feel would be interesting to pursue out of this include finding other distributions that exhibit a more "natural" behaviour, considering locality based on physical distance and allowing other UTMs to be present.

To digress slightly, it may be circumspect of us to address the question that makes up the title of this report in a more philosophical or general sense. A reaction that may have occurred in some readers upon reading the title is that of puzzlement at what exactly is meant by the question. The underlying

[3] The difference is negligible, but is mentioned here out of completeness.

[4] We can always add any number of "useless" states to any Turing machine and it will be an equivalent machine.

assumption touched on briefly in the introduction is that we are investigating random instances of systems that can perform universal computations. To draw an analogy between the artificial systems and the natural ones says that a natural system can perform universal computations. While this is true, it is true, if one will excuse the term, in an artificial sense. Natural systems can perform universal computations in the sense that human intervention has found ways to manipulate them using their inherent properties to make them capable of doing so. However, it is unknown whether or not natural systems actually perform universal computations in their regular environment or as part of their ritualistic survival. This report does make any claims, implied or otherwise, addressing this question as it is clearly outside the realm of a discussion of random graphs. It does raise some deep, important questions that should be pursued over time, such as how important universality is to an organism. Does universality afford an organism a better chance at survival? What happens when an organism does not have the universality trait? Is it even possible for an organism to not have the universality trait and be able to function? It is doubtful these questions can be answered immediately, but they could provide motivation for some very interesting research between biology and computer science.

5 Acknowledgments

The authors wish to thank Erik Winfree for laying the groundwork for the majority of this report as well as a plethora of insightful guidance and to Tom Knight for inspiration related to the analogy between natural and artificial systems.

References

[1] Minsky, M., *Computation: Finite and Infinite Machines*, Prentice Hall, 1967.

A Matlab code

A.1 MinskyUTM.dat

```
1 1 1 2
2 2 1 6
3 3 3 4
4 4 5 7
5 5 5 3
6 6 6 3
7 7 2 6
```

A.2 wexplocal.m

```
function w = wexplocal (node, N, lambda)
%WEXPLOCAL Compute the weight of a node based on other nodes/parameter.
%
%  WEXPLOCAL (NODE, N, LAMBDA) computes the weight of NODE based on it's
%  locality to N.  LAMBDA defaults to 1.0 if not given.  The formula is
%
%              e^((-|NODE - N|) / LAMBDA)
%
%  where NODE is a scalar and N may be a scalar or a matrix.

if nargin == 2
   lambda = 1.0;
end

w = exp (- (abs (node - N) ./ lambda));
```

A.3 sumweights.m

```
function s = sumweights (node, N, wfunc, lambda)
%SUMWEIGHTS  Sum the weights over a given number of elements.
%
%  SUMWEIGHTS (NODE, N, WFUNC, LAMBDA) sums the weights as given by WFUNC
%  over N elements as they relate to NODE.  WFUNC is called as such:
%
%    WFUNC (NODE, N, LAMBDA)
%
%  If LAMBDA is not given, it defaults to 1.0.
%
%  Note that the second argument to WFUNC (that is, N) must be able to
%  handle matrices.

if nargin == 3
   lambda = 1.0;
end

s = sum (feval (wfunc, node, N, lambda));
```

A.4 probconfig.m

```
function p = probconfig (node, ADJV, n, wfunc, lambda)
%PROBCONFIG  Compute the probability of a configuration for a node.
%
%  PROBCONFIG (NODE, ADJV, N, WFUNC, LAMBDA) computes the probability
%  of NODE having the configuration given by the adjacency vector ADJV
%  in a graph of N nodes with weighted distribution given by WFUNC and
%  adjustment parameter LAMBDA.  If LAMBDA is not given, it defaults
%  to 1.0.
%
%  See 'sumweights' for an explanation of WFUNC.
%
%  Note that N should be a scalar!  Here is an example of how to call
%  PROBCONFIG with an explanation:
%
%      probconfig (5, [5 6 6 7], 100, @wexplocal, 0.5)
%
%  This computes the probability that node 5 has four outgoing edges.
%  Specifically, it has a self loop (node 5 to node 5), two edges to
%  node 6 and an edge to node 7.  The full graph has 100 nodes and the
%  function 'wexplocal' is used to calculate the weighting of the
%  edges, with lambda parameter set to 0.5.

if nargin == 4
   lambda = 1.0;
end

denom = (sumweights (node, 1:n, wfunc, lambda))^size (ADJV, 2);
p = prod (feval (wfunc, node, ADJV, lambda)) / denom;
```

A.5 probblock.m

```
function p = probblock (block, ADJ, n, wfunc, lambda)
%PROBBLOCK Compute the probabilities of node configurations in a block.
%
%  PROBBLOCK (BLOCK, ADJ, N, WFUNC, LAMBDA) is essentially the same as
%  'probconfig' except that it computes the probabilities for entire
%  blocks.  BLOCK should be greater than or equal to 0.  ADJ should be
%  an entire adjacency matrix for the block.  N, WFUNC and LAMBDA are
%  the same as 'probconfig'.
%
%  An example is probably the best way to see how to use it.
%
%    probblock (3, MinskyUTM, 100, @wexplocal)
%
%  This computes the probability of the 4th block (recall block
%  numbering starts at zero) of a 100 node graph containing a
%  structure exactly as represented by the adjacency matrix
%  MinskyUTM.  LAMBDA is assumed to be 1.0, since it was not given as
%  an argument.
%
%  The size of a block is calculated by the number of rows in the
%  adjacency matrix.  Thus, if you pass in a 7x4 matrix, a block is
%  assumed to be 7 nodes and each node has 4 outgoing edges.  The
%  offset is calculated as (BLOCK * number of rows in ADJ).
%  Currently, the function does not check to ensure that N is
%  sufficiently large (e.g., if you want the 10th block of a 7x4
%  adjacency matrix in a graph with 30 nodes, it will return strange
%  results, since 10 * 7 > 30).
%
%  Also note that the adjacency matrix does not need to be adjusted
%  for working with a particular block.  The offset is merely added to
%  each element in ADJ to mimic the node behaviour.
%
%  PROBBLOCK returns a row vector containing the probability for each
%  node in the block.  Suppose a call to PROBBLOCK returns
%
%    [ 0.50  0.30  0.20  0.05  0.02  0.02  0.01 ]
%
%  when called on the 5th block.  This means node 7*5 + 1 = 36 has
%  probability 0.50 of occuring with the given configuration, node 37
%  has probability 0.30 and so on.

if nargin == 4
   lambda = 1.0;
end
```

```
count = 1;
probs = [];
offset = block * size (ADJ, 1);

while count <= size (ADJ, 1)
      probs(count) = probconfig ((count + offset), ...
                                 (ADJ(count, :) + offset), n, wfunc, lambda);
      count = count + 1;
end

p = probs;
```