Electronic Colloquium on Computational Complexity, Report No. 44 (2014)

# Additively efficient universal computers

Daniel Dewey*

*daniel.dewey@philosophy.ox.ac.uk*
*Oxford Martin Programme on the Impacts of Future Technology,*
*Future of Humanity Institute*

**Abstract**

We give evidence for a stronger version of the extended Church–Turing thesis: among the set of physically possible computers, there are computers that can simulate any other realizable computer with only additive constant overhead in space, time, and other natural resources. Complexity-theoretic results that hold for these computers can therefore be assumed to hold up to constant overhead on any other realizable computer. To support this claim, we offer an informal argument originally due to Deutsch, a formalization of that argument into a theorem showing sufficient conditions for the existence of additively efficient universal computers in arbitrary settings, and arguments that these sufficient conditions hold on physical resources including time and space. We also provide a formal setting in which we can prove that additively efficient universal computers exist.

*Supported by the Alexander Tamas Research Fellowship on Machine Superintelligence and the Future of AI.



ISSN 1433-8092

# 1 Introduction

Differences between computing machine models complicate the study of computational costs, which would ideally give results independent of any particular machine model. One way to reduce machine-dependence is to take advantage of models' abilities to efficiently simulate one another, so that complexity-theoretic results that apply to one computer also apply to the other. The extended[1] Church–Turing Thesis, which states that a probabilistic Turing machine can simulate any *realizable* (physically possible) machine model with polynomial time overhead, is a particularly useful thesis of this type; if it is true, then complexity-theoretic results that hold for PTMs can be assumed to hold up to polynomial overhead for any other realizable computer.

In this paper, we argue for a stronger thesis: among the set of realizable computers, there are computers that can simulate any other realizable computer with only additive constant overhead in space, time, and other natural resources. To do this, we introduce the general concept of *additively efficient universal computers*, then give arguments supporting this concept's applicability to realizable computers relative to resources like space and time.

Given a "setting" defined by a collection of computers and a way of measuring resources used in computation, let additively efficient universal computers in that setting be computers that can simulate any other computer in that collection with only additive constant overhead in that resource. We give three main results:

1. The possibility of *universal constructors* in a setting is sufficient, though not necessary, for the existence of additively efficient universal computers in that setting (Theorem 1).

2. Additively efficient universal computers certainly exist in some abstract settings (shown by construction).

3. There are good reasons to think that the set of *realizable* (physically possible) computers include some computers that are additively efficient relative to resources like space and time (plausibility arguments).

Result 1 is achieved by formalizing an informal argument due to Deutsch[2] into a theorem (Sections 3 and 4). The theorem is not difficult or complex to prove, but it serves to clarify the technical requirements and details of Deutsch's argument. In short, Deutsch argues that a universal constructor could efficiently simulate any other machine by first constructing a copy of that machine, then performing the desired computation on that copy. Thus, any setting that supports a universal constructor also supports additively efficient universal computers.

Result 2 is achieved by demonstrating an additively efficient universal computer in an abstract setting (Section 5). We choose for our setting the collection of programs that run on a certain universal Turing machine based on Minsky's 4-symbol, 7-state

[1] Also called the "complexity-theoretic" (Bernstein and Vazirani, "Quantum complexity theory") or "strong" (Arora and Barak, *Computational complexity: a modern approach*) form of the Church–Turing Thesis.

[2] Deutsch, "Constructor theory".

universal machine,[3] and exhibit a program that can simulate any other program on any other input with only additive overhead in time and space. This program works by "constructing" the program to be simulated on the tape, then placing the Turing machine in its initial state at the start of that program and allowing the run to proceed as normal.

Since result 3 concerns computation in the real world, we do not attempt a formal analysis. Rather, we give plausibility arguments suggesting that a physical universal constructor is possible, and that it would likely satisfy the sufficient conditions defined by Theorem 1 with respect to resources like time and space. If this is true, then it follows that there exist realizable additively efficient universal computers. We are able to gain additional traction by restricting the scope to physically realizable machines that could be built by some future human civilization. This probably does not exclude many machines of interest, and enables significantly stronger arguments in favor of the possibility of universal construction.

It is worth noting that, although we give evidence that the set of realizable computers includes additively efficient ones, no present-day computer is flexible enough to be additively efficient. Today's computers, and tomorrow's computers for nearby values of "tomorrow", will continue to vary in efficiency up to a polynomial. It is only in the long-term future of computing hardware, as computers become more flexible and diminishing scale further blurs the line between informational and physical reconfiguration, that we expect additively efficient universal computers to become relevant. Nevertheless, we argue, additive efficiency is likely to be a fundamental fact of computation in our physical world, at least in the limits of technological ability.

In general, our arguments proceed in close parallel to the proof and application of the Invariance Theorem in Kolmogorov complexity[4]. The Invariance Theorem states that there exist universal partial recursive functions that are additively optimal means of description; so long as one of these functions is used, results in program-length complexity can be made machine-independent up to a constant. Here, we are working with run-time complexity instead of static complexity, but the spirit and mechanics are much the same.

## 2 Definition

We define a *setting* to be a pair of a universal partial recursive function $U$ and a cost function $T$, such that $U(i, x)$ returns the result of running function $i$ on input $x$, and $T(i, x)$ returns the cost of that computation on $U$. We will think of $U$ as representing a "world" in which many different computers can be implemented, e.g. Conway's Game of Life or our physical universe; different function-indices $i$ will be taken to represent different computers implemented in this common world, so that $U(i, x)$ is the function computed by computer $i$ on input $x$ in the world $U$ represents. Similarly, $T(i, x)$ corresponds to the amount of some resource consumed by computer $i$ in its

[3]Cocke and Minsky, "Universality of tag systems with P= 2"; Minsky, "Computation: finite and infinite machines".

[4]See e.g. p. 104, Li and Vitanyi, *An introduction to Kolmogorov complexity and its applications.*

computation on input $x$. We can now define additively efficient universal computers in the setting defined by $U$ and $T$:

**Definition 1** (Additively Efficient Universal Computers)**.** *Given universal partial recursive function $U$ and cost-function $T$, computer $u$ is an additively efficient universal computer if and only if for every other computer $i$, some program $p$ causes $u$ to simulate computer $i$ while incurring only constant additional cost:*

$$\forall i\ \exists p, c\ \forall x : (U(u, px) = U(i, x) \wedge T(u, px) \leq T(i, x) + c) \\ \vee\ U(u, px) \text{ and } U(i, x) \text{ are both undefined.}$$

Two preliminary observations about additively efficient universal computers are in order here. First, any $u$ fulfilling this definition is additively efficient relative to other computers *within setting* $U, T$; the definition makes no claims about relative efficiencies of computers implemented in separate worlds, or with different kinds of resources (however such comparisons would be made).

Second, it seems unlikely to us that there are elegant necessary and sufficient conditions for the existence of additively efficient universal computers in a setting. This is because the existence of universal computers in a setting, a strictly weaker property, is almost always shown by laborious construction. It seems doubtful that the additive efficiency part of the property can be extricated cleanly from the universal computation part, or that the addition of efficiency concerns decreases the difficulty of demonstration rather than increasing it. We will proceed to outline some sufficient conditions useful in the application of this concept to physical computers, but we are pessimistic about the existence of elegant necessary and sufficient conditions for additive efficiency in a setting, despite its complexity-theoretic usefulness.

# 3 The universal constructor argument

We have defined additively efficient universal computers, but are they relevant to computation in the real world? Do they exist among realizable computers? In general, what settings support additively efficient universal computers? Our first result is that the possibility of *universal constructors* in a setting is sufficient, though not necessary, for the existence of additively efficient universal computers in that setting.

Universal constructors were pioneered by von Neumann in his study of the abstract problem of self-reproduction.[5] In von Neumann's work, a constructor is a pattern within a cellular automaton that can, as the automaton steps forward, cause other patterns to appear nearby. The set of patterns a constructor can be programmed to construct is called its *repertoire*. A self-reproducing pattern can then be defined: Let `C` be a constructor, and let `C(x)` denote the pattern of "`C` loaded with program `x`". Suppose that `C`'s repertoire includes the pattern `C(p)`, and that `p` is the program that causes `C` to produce `C(p)`. This means that an initial configuration of `C(p)` will, after some number of steps, produce a later configuration `C(p) C(p)` (assuming that `C` can do this without altering itself significantly), and thus `C(p)` has reproduced itself.

[5] Von Neumann, Burks, et al., *Theory of self-reproducing automata*; Thatcher, *Universality in the von Neumann cellular model.*

The traditional meaning of the "universal" part of universal constructors is somewhat fuzzy. Universality of a constructor would most naturally mean that a constructor's repertoire includes every finite pattern, but this property is impossible if some finite patterns have no legal predecessor state under the automaton's rules. These "Garden of Eden" configurations cannot be constructed by any machine.[6] The solution used by von Neumann is to work in an automaton with *quiescent* cell states, states which stay unchanged over time unless exposed to non-quiescent neighbors, and to define universality as the ability to construct any finite pattern of quiescent cells. This form of universality is often possible, and is well-suited to the problem of self-reproduction. In this paper, we will use "universality" to mean that a constructor's repertoire includes some very large subset of the finite patterns; when the properties of this subset are important for our arguments, we will specify them.

Many automata support many types of universal computers: single- and multi-tape Turing machines, RAM and register machines, billiard-ball computers, etc. Suppose that some constructor has computers $\mathtt{m}_1, \mathtt{m}_2, ...$, implementing universal partial recursive functions $m_1, m_2, ...$, in its repertoire. Using this constructor, we can design a new computer, `U`, with universal function $U$. `U` works by constructing a computer, then feeding it a program: we might arrange that when `U` is given program `nx`, it constructs computer $\mathtt{m}_\mathtt{n}$ and feeds it program `x`, so that $U(nx) = m_n(x)$. If the cost of delaying and redirecting inputs to the newly constructed computer is constant, `U` can thus "simulate" the actions of each computer in its repertoire on any program while using only a constant $c$ more resources, the amount of resources it takes `U` to construct that computer. Thus, `U` is additively efficient relative to the computers in its repertoire.

If we are working in an automaton that supports some form of universal constructor, then we can apply the above process to it, obtaining a computer that is additively efficient relative to a very large subset of the possible computers in the automaton. Furthermore, if the constructors' repertoires are large enough, then all additively efficient universal computers within the automaton (made from different universal constructors, say) are additively *equivalent*: for any two computers, there is a constant $c$ such that either one can simulate the other within $c$ resource overhead. These computers form an equivalence class of additively efficient universal computers.

This basic argument was first articulated by David Deutsch, in a paper posted to arXiv in 2012, then published in *Synthese* in 2013:

> In constructor theory a stricter conception of universality is possible, because when a programmable constructor is programmed to mimic another constructor $C$, it may begin by constructing an instance of $C$, to which it directs subsequent inputs, so that from then on it performs $C$'s task using much the same resources that $C$ would... the overhead of programming [programmable constructor] $P$ to be capable of performing [task] $A$ is a constant $c(P, A)$, independent of how often $P$ will then be called upon to perform $A$, and which inputs for $A$ it is given.[7]

[6] Moore, "Machine models of self-reproduction".
[7] Deutsch, "Constructor theory".

We have not been able to find an earlier expression of this idea, but we believe Deutsch's insight deserves closer attention, especially in its implications for complexity theory; this paper is a first attempt.

# 4 The universal constructor theorem

To formalize the universal constructor argument, we will need little bit more structure. Let $S$ be a countably infinite set of "states" that computers may pass through. We will define two subsets, *start states* and *halt states*; these subsets may or may not be disjoint or exhaustive. Start states are written $s_{i,x}$, and there is one start state for every pair of computer $i$ and input $x$. Halt states are written $h_0, h_1, h_2...$, and there is at least one halt state for each natural number. Let each state have a single successor state, reflecting some deterministic rule by which a computation proceeds. We will write the sequence of consecutive states following $s$ as "$s...$". A sequence of consecutive states has a non-negative integer cost, written $cost(s...)$.

Two rules link the set of states to the our $U$ and $T$: $U$ restricts which states are consecutive, and $T$ restricts costs of certain sequences.

1. $U(i,x) = y$ if and only if $s_{i,x}...$, contains a first halt state $h_y$.
2. $T(i,x) = c$ if and only if $\exists y : U(i,x) = y$ and $cost(s_{i,x}...h_y) = c$.

So long as these rules are followed, the set of states, their sequence relations, and sequence costs will all be consistent with the setting $U, T$.

If we augment a setting with a set of states, we can state the universal construction argument as a theorem:

**Theorem 1.** *The following conditions are sufficient for a universal function $U$ and cost-function $T$, augmented with a set of states $S$, to have a non-empty equivalence class of additively efficient universal computers.*

1. *Subadditivity: For any sequence of consecutive states $s_1...s_n...$,*

$$cost(s_1...s_n...) \leq cost(s_1...s_n) + cost(s_n...).$$

   *If a subsequence has no defined cost, then the overall sequence has no defined cost.*

2. *Universal Construction: There exists a constructor $u$, a computer that can be programmed to reach certain states, that is universal in then following sense: $u$ can be programmed to construct any other computer $i$, then give that computer input $x$, while incurring a cost $c$ that is independent of $x$, and without first encountering a halt state. Formally:*

$$\begin{aligned} \forall i\ \exists p, c\ \forall x : & s_{i,x} \in s_{u,px}... \\ & \wedge \forall j : h_j \notin s_{u,px}...s_{i,x} \\ & \wedge cost(s_{u,px}...s_{i,x}) \leq c. \end{aligned}$$

*Proof.* The proof is not difficult; $u$ is a member of the desired class. Consider any computer $i$. Condition 2 implies that there is a program $px$ that causes $u$ to construct $i$ and load input $x$ in constant time, and condition 1 implies that the overall cost of construction and simulation is no greater than the sum of the costs. Therefore, this program causes $u$ to simulate $i$ with additive constant overhead.

Since any other additively efficient universal computer $u'$ is also a computer in $U$, the above applies to it as well, and so $u$ and $u'$ can simulate one another with an additive constant overhead dependent only on $u$ and $u'$. Thus, the additively efficient universal computers in the setting form an equivalence class up to additive constant overhead. The full proof is given in Appendix 1. □

This theorem establishes a pair of conditions, subadditivity and universal construction, that are sufficient for the existence of additively efficient universal computers. From a mathematical perspective, the conditions are not particularly elegant, and they are certainly not necessary. From a complexity standpoint, however, these sufficient conditions are interesting because it is plausible that resources like physical space and time obey them. This means that additively efficient universal computers are relevant to the study of physical computation and cost, and that restricting the choice of machine model to these computers is not unreasonable is some cases, allowing us to use the property of additive machine-independence in analysis of real computing machines.

## 4.1 Physical plausibility of subadditivity

The first condition, subadditivity, requires that the cost of a sequence of consecutive states is no greater than sum of the costs of any partitioning set of subsequences: for any sequence of consecutive states $s_1...s_n...$,

$$cost(s_1...s_n...) \leq cost(s_1...s_n) + cost(s_n...).$$

What kinds of resources are subadditive? Loosely define the set of states as physical configurations that a set of computers can take: a sequence of states $s_{i,x}...$ contains states corresponding some of the physical configurations that computer $i$ passes through while computing $U(i, x)$, a series of snapshots with a frequency and regularity depending on the granularity desired for the setting.

The most commonly used resources in complexity, time (as in machine steps) and space (memory read or written) are subadditive: the time or space used by a sequence is no greater than the sums of the time or space used by partitioning subsequences. Physical time is also subadditive, as is physical space, both in the sense of the maximum envelope the computer occupies during its computation and in the sense of the "volume" the computation occupies (integrating space over time). Bits of additional input or output, physical energy, and waste products are examples of more unusual subadditive resources.

Subadditivity holds on many resources, but not all. If the cost of a particular sequence of states varies according to when it occurs, or depending on states that are not part of the sequence (e.g. by using some resource that gets more expensive as additional units are taken), then subadditivity will not hold.

## 4.2 Physical plausibility of universal construction

The universal construction condition divides roughly into two parts: there must be a computer that can be programmed to construct any other computer, and further input must be set aside during construction, then passed to the new computer, without incurring non-constant cost.

The first part, the existence of the universal constructor for a given setting, is difficult to verify non-constructively. Just as Turing-completeness is typically shown by construction, or assumed as an axiom of an acceptable Gödel numbering, it may be the case that there are no simple properties of a setting that are necessary and sufficient for the existence of a universal constructor. In the case of the physical world, there are some reasons to suspect that universal contruction is possible, at least for some reasonable meanings of "universal":

If, as seems plausible, physical laws are time-reversible, then there can be no Garden-of-Eden states; any state's predecessor can be found by applying the time-reversed laws. This is evidence in favor of a strong form of physical universal construction. However, this is not sufficient. Not only must every state be reachable, they must all be reachable from one set of states corresponding to a constructor with different programs. Imagine a directed graph of physical states, with arrows pointing from past to future states under physical laws. Conservation laws imply that this graph is split into separate components, islands that cannot be reached from one another. If a strongly universal constructor existed, its programs would need to include whatever alterations to conserved quantities (energy, angular momentum, etc.) are needed to bridge the gaps between these components. Otherwise, universality in physical systems might be best defined within only one of these conservation sub-graphs. Another difficulty is raised by the irreversible accumulation of entropy within a closed mechanical system, and the loss of usable energy through heat; it may be that some theoretically possible trajectories are not in practice realizable in physical computing machines.

To avoid depending too heavily on deep properties of physical laws, we suggest a kind of universality that is weaker, but still useful from a complexity-theoretic point of view: *there could exist machines with the ability to construct any object that humanity could ever build, provided we were appropriately motivated.* Since humans are biological machines, and since we already use machines to build most of our machines, it is very plausible that there could exist machines with the ability to construct any object that humanity could ever build. Naïvely, it would only require the unification in one machine of every manufacturing ability we currently possess (which could then construct all of our future manufacturing technologies, and so on). A more practical approach would be to create a programmable, general-purpose nanofactory.[8] If this factory is at least as capable as a ribosome, there is strong reason to think it could eventually build anything we could. With the benefit of purposeful design, a general-purpose nanofactory could proceed to the construction of arbitrary objects much more quickly than ribosomes have.

Based on this argument, it seems plausible that the physical universe supports a universal constructor that can construct anything humans could ever build, and

[8] Drexler and Minsky, *Engines of creation.*

therefore that it also supports an equivalence class of universal computers that are additively efficient relative to any computer humans could ever build. While this set does not contain every physically possible object, it is so large that such a constructor can be usefully called "universal".

The second part of universal construction, the ability to "set aside" inputs without cost, is easier to work with non-constructively. The remaining input is perhaps best thought of as "deferred", or left on the tape while the machine is constructed, then passed to the new machine; settings that penalize this redirection with a non-constant cost will not satisfy the condition. For example, the "volume" the computation occupies (integrating space over time) does not satisfy the universal construction condition, since when the constructor is building a machine, the cost of that construction is not constant, but proportional to the length of the input, to account for the volume taken up by the input while it is deferred.

On the other hand, most of the subadditive resources we have mentioned above do not penalize the storage of input, and so they can fulfill the universal construction condition if a suitable machine exists. It seems that time (machine steps or physical) and space (memory read or written, or physical space altered by the machine), bits of additional input or output, physical energy, and waste products are all independent of deferred input, and so all meet this part of the condition.

# 5 A concrete example

We have focused on realizable, physical computing machines, and although this set is of interest for obvious reasons, it is hard to analyze formally. To give a more rigorous example, in this section, we will limit ourselves to a set of computers that are programs on a particular universal Turing machine. In this setting, we can show an example of a universal constructor of the type needed for Theorem 1, and thus that these programs include an equivalence class of additively efficient universal computers. That is, there is a program on this Turing machine that can simulate any other program on any other input with only additive constant overhead; it does so by constructing, in constant time and space with respect to the input, the program to be simulated on the tape, then placing the Turing machine in its initial state at the start of that program and allowing the run to proceed as normal. We have posted an implementation online[9], along with some examples of construction and constant-overhead simulation.

Though it would be most satisfying to do this construction in a simple, well-known setting like Conway's Game of Life, this is beyond the scope of the current paper. Universal construction (for a variety of meanings of "universal") in Life is a very complex problem. Accounts of how universal or near-universal construction could be achieved began appearing shortly after Life itself was created, with perhaps the earliest near-complete account published by Conway in 1982.[10] Technical implementations are still an active area of research, and recent results (like Wade's 2010

[9] `http://www.danieldewey.net/turing-machine-M.html`

[10] Berlekamp, Conway, and Guy, *Winning Ways for Your Mathematical Plays, 2nd Ed., Volume 4, Chapter 25*; Poundstone, *The recursive universe: cosmic complexity and the limits of scientific knowledge.*

Gemini design) are quite promising, offering large improvements in size and speed over previous attempts.[11] It seems likely that the obstacles remaining are more practical than conceptual, but the complexity of this domain makes it unsuitable for this paper. Similarly, von Neumann's 29-state machine has been more-or-less shown to support universal construction (modulo quiescent states), but this machine's complexity makes it too difficult to use here.

Universal construction on a Turing machine tape is significantly easier. In a universal Turing machine, "construction" of a computer amounts to writing a program on the tape. While Life configurations are rarely stable, data on a tape does not change unless rewritten, and so it is easier to build up complex structures without worrying about them falling apart mid-construction.

As Minsky points out in his exposition of a minimal universal Turing machine,[12] one must not be too permissive in the input conventions considered for a universal machine. Minsky's example is that a simple identity function might technically be a "universal function" if the input convention itself is allowed to be a universal function. Similarly, an identity function is a "universal constructor" if the input convention itself is allowed to be universal. For this reason, we adopt a restriction: our universal constructor will take programs in a subset of the UTM's symbol set, and must be able to construct any string using the full symbol set.

Our Turing machine, which we'll refer to as $M$, is based on Minsky's 4-symbol, 7-state universal machine. We augment Minsky's machine with twenty-one additional states and one additional symbol. On $M$, the program $y1yA0$ is a universal constructor, able to be programmed with finite sequences from alphabet $\{A, y\}$ to construct any finite sequence from the full alphabet $\{0, 1, A, y, x\}$ and leave $M$ in its starting state at the head of the constructed string. $M$'s full action table is given in Table 1.

## 5.1 How $M$ works

We designed $M$ with two objectives: $M$ should be obviously universal because it can easily simulate Minsky's machine, and $M$ should support a relatively simple universal constructor using the same input alphabet $\{A, y\}$ as Minsky's machine.

$M$'s simulation of Minsky's machine is quite simple, since Minsky's machine's action table is contained entirely in $M$'s table (states $q1$–$q7$). Minsky's machine can simulate any two-tag system, which is enough to show that it is universal; for details, see Minsky, "Computation: finite and infinite machines". For our purposes, it is enough to know that the simulation of a tag system requires that the tape be loaded with a *production table* in the form of a sequence of 1s and 0s, followed by a *tag list* in the form of a sequences of $y$s and $A$s, and that the read-write head start in state $q2$ on the leftmost symbol in the tag list.

$M$, by contrast, starts by convention on the leftmost nonzero symbol on the tape in state "ready". To simulate Minsky's machine on a tag system, $M$ must be loaded with exactly the same string as Minsky's machine would be, but with an $x$

[11] Goucher, "Universal Computation and Construction in GoL Cellular Automata"; Greene, *Replicator Redux*.

[12] Minsky, "Computation: finite and infinite machines".

| | 0 | 1 | $A$ | $y$ | $x$ |
|---|---|---|---|---|---|
| ready | right, clean | right, buffer1 | right, inc1 | right | 0, right, cue |
| buffer1 | right, buffer2 | right | right | right | - |
| buffer2 | left, fail1 | right | x, right, buffer3 | right | right |
| buffer3 | y, left, succ1 | right | right | right | - |
| inc1 | right, inc2 | right | right | right | - |
| inc2 | left, fail1 | right | 1, right, endinc1 | 1, right, inc3 | - |
| inc3 | right, inc4 | right | right | right | - |
| inc4 | 1, left, inc5 | A, left, inc5 | y, left, inc5 | x, left, inc5 | - |
| inc5 | left, inc6 | - | - | - | - |
| inc6 | right, inc2 | left | left | left | - |
| endinc1 | left, endinc2 | right | right | right | - |
| endinc2 | - | - | - | 0, left, succ1 | - |
| succ1 | left, succ2 | left | left | left | left |
| succ2 | right, succ3 | left | left | left | - |
| succ3 | right, clean | 0, right, buffer1 | 0, right, inc1 | right, ready | 0, right, cue |
| fail1 | left, fail2 | left | left | left | A, left, fail1 |
| fail2 | right, fail3 | left | left | left | - |
| fail3 | right, clean | 0, right, buffer1 | 0, right, inc1 | 0, right, erase | 0, right, cue |
| erase | - | 0, right, ready | 0, right, ready | - | - |
| clean | right, ready | 0, right, clean | 0, right, clean | - | - |
| cue | right | right | y, right, q6 | 0, left, q1 | - |
| q1 | left | left, q2 | 1, left | 0, left | - |
| q2 | y, right | A, right | y, right, q6 | 0, left, q1 | - |
| q3 | halt | A, left | 1, left, q4 | left | - |
| q4 | y, right, q5 | left, q7 | 1, left | left | - |
| q5 | y, left, q3 | A, right | 1, right | right | - |
| q6 | A, left, q3 | A, right | 1, right | right | - |
| q7 | y, right, q6 | right | 0, right, q2 | 0, right | - |

Table 1: The action table for $M$. This universal Turing machine has at least one universal constructor, $y1yA0$, able to be programmed with finite sequences from alphabet $\{A, y\}$ on an infinite tape of background 0 to construct any finite sequence from the full alphabet $\{0, 1, A, y, x\}$. States $q1$–$q7$ are identical to Minsky's 4-symbol, 7-state UTM.

appended to the left. For example, to simulate a tag system with production table 110101110000010011011 and tag list $yyAyyAyy$, $M$ requires an initial tape of $x110101110000010011011yyAyyAyy$. Reading $x$ in state "ready" causes $M$ to enter state "cue", which in turn causes it to move right until it encounters the first symbol in the tag list. From there, $M$ behaves identically to Minsky's machine; this follows from the embedding of Minsky's action table in $M$'s. This example system's expected behaviour is described in Minsky, "Computation: finite and infinite machines" p.280 ("The four-symbol seven-state universal machine").

$M$'s universal constructor is the program $y1yA0$. To construct and then run an arbitrary string $s$, it suffices to append $s$'s "codeword" to $y1yA0$. To find a string $s$'s codeword, first reverse it, then encode each of its symbols: 0 becomes $A$, 1 becomes $yA$, $A$ becomes $yyA$, $y$ becomes $yyyA$, and $x$ becomes $yyyyA$; and finally, remove the final $A$ from the resulting string. For example, the codeword for string $xAy$ is $yyyAyyAyyyy$. This means that $M$, starting in state $ready$ on the leftmost symbol of tape $y1yA0yyyAyyAyyyy$, will eventually produce tape $xAy$, leaving $M$ in state "ready" on the leftmost symbol.

Construction takes place in two parts, corresponding to the two "phrases" $y1$ and $yA$ in the constructor program: $y1$ builds a "buffer", a series of $y$s to the right of the codeword, and then $yA$ "fills in" this buffer by copying each coded symbol into its proper place. For a step-by-step explanation of how $y1yA0$ builds this string, see Appendix 2.

This construction process can be used to build and execute arbitrary programs, using the encoding scheme given above. Thus, $y1yA0$ is a universal constructor in machine $M$. The contructor can even be used to construct a tag production table, then execute it on an input as in Minsky's machine. An example tape achieving this is the rather long string

$$y1yA0yAyAAyAyAAAyAAAAAAyAyAyAAyAAyAyAyyyy000000000000000000000000yyAyyAyy$$

The run of this example is too long to be included here, but can be seen in our implementation. Its significance is that it shows $y1yA0$ simulating another program in $M$ with additive overhead: since only the production table is constructed, the input (tag list) could have been arbitrarily long without affecting the construction time.

Since $M$ supports a universal constructor, it also supports an equivalence class of additively efficient universal computers. The details of applying Theorem 1 to $M$ and $y1yA0$ are given in Appendix 3.

# 6 Conclusion

We have given evidence that a strong machine-invariance thesis holds over realizable computers: relative to many kinds of physical computing resource, there exist equivalence classes of additively efficient universal computers, and if the choice of computing machine model is restricted to additively efficient universal computers, results in computational complexity theory can be made machine-independent up to an additive constant. Our evidence consists of an informal expansion of Deutsch's

insight, followed by a formal definition and a theorem whose sufficient conditions plausibly hold on many physical resources, including time and space. We have also given an example of a setting, Turing machine $M$, in which a universal constructor enables additively efficient universal computation across all other computers in the setting.

As we mentioned earlier, these sufficient conditions are not necessary. It is likely that in the physical world, only a small fraction of universal construction capacity is actually needed for an additively efficient universal computer. The sufficient conditions even fail to capture some additively efficient universal computers that are universal-constructor based. For example, the "envelope" of a computation, the size of the smallest area that the computer and its input do not extend beyond at any point during a computation, violates the second part of the universal construction condition: when input is set aside until later, it contributes to the envelope of the construction proportionally to the length of the input, and so the initial construction step is not of constant cost. However, the overall simulation is clearly additively efficient, since the constructed computer must use the same input space, pushing the envelope out to exactly the same point. Additive efficiency still holds because the "envelope measure" is extremely subadditive; the same space is frequently re-used, especially in the case of deferred input.

Additively efficient universal computers enable a pleasing symmetry between static and run-time complexity measures. For example, the information content of a string (as measured by Kolmogorov complexity) and the "knowledge" content derived or deduced from that information (as measured by the logical depth[13] of the string) can both be quantified in a way that depends on the machine model only by a constant factor.

Additively efficient universal computers, and the evidence that they are realizable, suggest a deep fact about the relationship between computational problems and physical systems: computers that suffer from polynomial slowdown are failing, through limitations in their computing model, to compress some computational regularities in the problems they solve. Our main proposed application is to make areas such as complexity theory and logical depth machine-independent up to a constant, but more elegant sufficient conditions, or more information about necessary conditions, may also yield deeper insights into the nature and future of computation.

## Acknowledgements

Thanks to Scott Aaronson, Nick Bostrom, and Toby Ord for their feedback and suggestions.

## References

Arora, Sanjeev and Boaz Barak. *Computational complexity: a modern approach*. Cambridge University Press, 2009.

[13] Bennett, "Logical depth and physical complexity".

Bennett, Charles H. "Logical depth and physical complexity". In: *The Universal Turing Machine A Half-Century Survey*. Springer, 1995, pp. 207–235.

Berlekamp, Elwyn R, John H Conway, and Richard K Guy. *Winning Ways for Your Mathematical Plays, 2nd Ed., Volume 4, Chapter 25*. AK Peters, Ltd, 2001–2004.

Bernstein, Ethan and Umesh Vazirani. "Quantum complexity theory". In: *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*. ACM. 1993, pp. 11–20.

Cocke, John and Marvin Minsky. "Universality of tag systems with P= 2". In: *Journal of the ACM (JACM)* 11.1 (1964), pp. 15–20.

Deutsch, David. "Constructor theory". In: *Synthese* 190.18 (2013), pp. 4331–4359. ISSN: 0039-7857. DOI: 10.1007/s11229-013-0279-z. URL: http://dx.doi.org/10.1007/s11229-013-0279-z.

Drexler, K Eric and Marvin Minsky. *Engines of creation*. Fourth Estate, 1990.

Goucher, Adam P. "Universal Computation and Construction in GoL Cellular Automata". In: *Game of Life Cellular Automata*. Springer, 2010, pp. 505–517.

Greene, Dave. *Replicator Redux*. Blog. 2013. URL: http://b3s23life.blogspot.co.uk/2013/01/replicator-redux.html.

Li, Ming and Paul Vitanyi. *An introduction to Kolmogorov complexity and its applications*. Springer, 1997.

Minsky, Marvin L. "Computation: finite and infinite machines". In: *Englewood Cliffs, N.J.: Prentice-Hall* (1967).

Moore, Edward F. "Machine models of self-reproduction". In: *Proc. Symp. Appl. Math.* Vol. 14. 1962, pp. 17–33.

Poundstone, William. *The recursive universe: cosmic complexity and the limits of scientific knowledge*. Courier Dover Publications, 2013.

Thatcher, James W. *Universality in the von Neumann cellular model*. Tech. rep. DTIC Document, 1964.

Von Neumann, John, Arthur Walter Burks, et al. *Theory of self-reproducing automata*. University of Illinois press Urbana, 1966.

## Appendix 1— proof of Theorem 1

*Proof.* $u$ is a member of the desired class. Consider any computer $i$. Condition 2 implies that there is a program $px$ that causes $u$ to construct $i$ and load input $x$, and we can show that this program causes $u$ to simulate $i$ with additive constant overhead.

Through application of the conditions and state consistency rules, we can show that $p$ causes $u$ to emulate $i$'s functional properties. If $\phi_i(x)$ is undefined, it follows from the first state consistency rule that $s_{i,x}...$ contains no halt state. Since, by condition 2, $s_{u,px}...s_{i,x}$ contains no halt state, it follows that $s_{u,px}...s_{i,x}...$ has no halt state, and by another application of the first state consistency rule, $\phi_u(px)$ is undefined. On the other hand, if $\phi_i(x) = y$, it follows from the first state consistency rule that $s_{i,x}...$ contains a first halt state $h_y$. Since, by condition 2, $s_{u,px}...s_{i,x}$ contains no halt state, $s_{u,px}...s_{i,x}...$ contains a first halt state $h_y$, and by another application of the first state consistency rule, $\phi_u(px) = y$.

We can also show that $u$ incurs only additive constant overhead when $p$ causes it to simulate $i$, or that both computers fail to halt. Suppose first that $\Phi_i(x)$ is undefined. By the second state consistency rule, $cost(s_{i,x}...)$ is undefined; condition 1 implies that $cost(s_{u,px}...s_{i,x}...)$ is then undefined, and a second application of the second state consistency rule implies that $\Phi_u(px)$ is undefined. On the other hand, suppose $\Phi_i(x)$ is defined. By the second state consistency rule, $cost(s_{i,x}...) = \Phi_i(x)$. Condition 1 implies that

$$\begin{aligned} cost(s_{u,px}...s_{i,x}...) &\leq cost(s_{u,px}...s_{i,x}) + cost(s_{i,x}...) \\ &\leq cost(s_{u,px}...s_{i,x}) + \Phi_i(x). \end{aligned}$$

By condition 2, $cost(s_{u,px}...s_{i,x})$ is less than or equal to $c$, some constant depending only on $i$ and $u$, and independent of $x$. Thus, $cost(s_{u,px}...s_{i,x}...) \leq \Phi_i(x) + c$, and so by the second state consistency rule, $\Phi_u(px) \leq \Phi_i(x) + c$.

Since any other additively efficient universal computer $u'$ is also a computer in $U$, the above applies to it as well, and so $u$ and $u'$ can simulate one another with an additive constant overhead dependent only on $u$ and $u'$. Thus, the additively efficient universal computers of any $U, T$ form an equivalence class up to additive constant overhead. □

## Appendix 2— $y1yA0$ constructs $xAy$

$$...y1yA0yyyAyyAyyyy...$$

Starting in state "ready" on the leftmost $y$, the head proceeds right, changing to state "buffer1" when it reads 1 in the constructor string. On reaching the first $A$ in the codeword, it changes to state "buffer2"; it will now place a buffer-mark $y$ corresponding to this $A$ at the end of the string, and rewrite the $A$ as an $x$ to show that it has been marked.

$$...y1yA0yyyxyyAyyyyy...$$

Since the head did find an $A$ to add to the buffer, it returns left in the "success" states "succ1" and "succ2", bouncing off the leftmost 0 and reaching the leftmost $y$ in state

"succ3". Since the buffering was a success, $y$ moves the head to the right in state "ready", allowing the 1's buffering process to be repeated.

$$...y1yA0yyyxyyxyyyyyy...$$

When buffering is attempted again, no, $A$ is found; the buffer is long enough (the length of the codeword minus one). The head encounters a 0 and returns to the left in the "fail" states "fail1" and "fail2", resetting the $x$s to $A$s and eventually reading "y" in state "fail3". Since the buffering is now done, the $y$ and the 1 are erased, and copying begins.

$$...yA0yyyAyyAyyyyyy...$$

Starting in state "ready" on the leftmost "y", the head proceeds right, changing to state "inc1" (the first of the "increment" states) when it reads $A$ in the constructor string. On reaching the first $y$ in the codeword, it changes to state $inc2$; it will now move right to the end of the buffer, pass over the 0 there, and increment the next 0. The head rewrites the first $y$ as a 1 to show that it has been counted.

$$...yA01yyAyyAyyyyyy01...$$

The head returns left in state "inc6", bouncing off the 0 just before the codeword. It will now continue to increment the same symbol once for each leading $y$ in the codeword, stopping only when it encounters an $A$:

$$...yA011yAyyAyyyyyy0A...$$

$$...yA0111AyyAyyyyyy0y...$$

On encountering the $A$, it is done writing the string's final symbol: a $y$, as coded by the codeword's first part $yyA$. The head marks the $A$ as a 1 and proceeds to the end of the buffer, where it replaces the last $y$ in the buffer with a 0 and returns in the "success" states (since it found and copied a symbol from the codeword):

$$...yA01111yyAyyyyy00y...$$

Reaching the leftmost $y$ in state "succ3", the machine repeats the "increment" process, copying the second symbol from the codeword:

$$...yA011111yAyyyyy01y...$$

$$...yA0111111Ayyyyy0Ay...$$

$$...yA01111111yyyy00Ay...$$

Similarly, it copies the third symbol:

$$...yA011111111yyy01Ay...$$

$$...yA0111111111yy0AAy...$$

$$...yA01111111111y0yAy...$$

$$...yA0111111111110xAy...$$

The desired string is now constructed. On encountering the rightmost 0 instead of a $y$ or $A$, the "increment" process is complete, and the head returns left in "fail" states. Once again, this causes the $y$ and the $A$ to be erased:

$$...0111111111110xAy...$$

In state "ready", reading 0 means that the construction is done; the machine enters state "erase". It moves right, cleaning up the 1s now covering the codeword. On reaching the 0 to the left of the constructed string, it moves right, entering state "ready" on the leftmost symbol of the constructed string.

$$...xAy...$$

$M$ is now in exactly the state it would be if it were started with program $xAy$.

# Appendix 3: $M$ and $y1yA0$ satisfy Theorem 1

We will show that the program $y1yA0$ is a universal constructor with the properties required to satisfy Theorem 1. First, we will use $M$ to define a universal partial recursive function $U(i, x)$:

1. On $M$'s initial tape (filled with background symbol 0), place the digits of $i$ expressed in base five with alphabet $\{0, 1, A, y, x\}$, with $M$'s read-write head on the leftmost digit.

2. Place the digits of $x$ encoded in base three with alphabet $\{0, A, y\}$ to the right of the last digit of $i$.

3. Starting in state "ready", run $M$. If it never halts, $U(i, x)$ is undefined. If it does halt, let $U(i, x) = y$, where $y$ is encoded in base two with alphabet $\{A, y\}$ to the right of the read-write head.

To apply Theorem 1, we must relate $U$ to a set of states. We will use the set $S$ of "full states" of $M$, which include not just the state $M$ is in, but also the string on the tape and the position of the read-write head. Each starting state $s_{i,x}$ will correspond to the full state of $M$ prepared as above to compute $U(i, x)$. Each halting state $h_i$ will correspond to a full state of $M$ where $i$ is encoded in base two with alphabet $\{A, y\}$ to the right of the read-write head and $M$ has halted. For each non-halting state, the successor state is the next state $M$ will enter, as determined by its table; halting states are their own successors.

**$U$ and $S$ obey the first state consistency rule:** Clearly from the definitions of states in $S$, $U(i, x) = y$ if and only if $s_{i,x}...$ contains a first halt state $h_y$.

**$\ulcorner y1yA0 \urcorner$ is a universal constructor in the sense of Theorem 1:** Let $\ulcorner y1yA0 \urcorner$ be the input to $U$ corresponding to our construction program $y1yA0$. Theorem 1 calls for a the existence of a "universal constructor" index $u$, such that for any index $i$, there is an input-prefix $p$ that causes $u$'s start-state $s_{u,px}$ to lead to $i$'s start-state

$s_{i,x}$, without first encountering a halt state, and with a cost that's independent of the input-suffix $x$. Our constructor $\ulcorner y1yA0\urcorner$ can be programmed to bring about any other starting state without first entering a halt state, and does this construction without interacting at all with the input $x$ (as shown by example in the previous subsection and Appendix 2).

This guarantees that for all $x$, $cost(s_{u,px}...s_{i,x})$ is constant for a wide range of cost functions: for example, the number of steps taken by $M$, the number of symbols written, and the number of state changes are all constant relative to the input $x$. It so happens that all of these cost functions are subadditive as well, and so Theorem 1 shows that $\ulcorner y1yA0\urcorner$ is an additively efficient universal computer relative to $U$ and any $T$ defined using resources like steps, symbols written, or state changes (and obeying the second state consistency rule).



ECCC ISSN 1433-8092

http://eccc.hpi-web.de