Planning with Biological Neurons and Synapses

Francesco d'Amore,[1] **Daniel Mitropolsky,**[2] **Pierluigi Crescenzi,**[3]
Emanuele Natale,[1] **Christos H. Papadimitriou**[2]
[1] Université Côte d'Azur, Inria, CNRS, I3S, Sophia Antipolis, France
[2] Department of Computer Science, Columbia University, New York, USA
[3] Gran Sasso Science Institute, L'Aquila, Italia
{francesco.d-amore,emanuele.natale}@inria.fr
{dgm2144,christos}@columbia.edu
pierluigi.crescenzi@gssi.it

Abstract

We revisit the planning problem in the blocks world, and we implement a known heuristic for this task. Importantly, our implementation is biologically plausible, in the sense that it is carried out exclusively through the spiking of neurons. Even though much has been accomplished in the blocks world over the past five decades, we believe that this is the first algorithm of its kind. The input is a sequence of symbols encoding an initial set of block stacks as well as a target set, and the output is a sequence of motion commands such as "put the top block in stack 1 on the table". The program is written in the Assembly Calculus, a recently proposed computational framework meant to model computation in the brain by bridging the gap between neural activity and cognitive function. Its elementary objects are assemblies of neurons (stable sets of neurons whose simultaneous firing signifies that the subject is thinking of an object, concept, word, etc.), its commands include project and merge, and its execution model is based on widely accepted tenets of neuroscience. A program in this framework essentially sets up a dynamical system of neurons and synapses that eventually, with high probability, accomplishes the task. The purpose of this work is to establish empirically that reasonably large programs in the Assembly Calculus can execute correctly and reliably; and that rather realistic — if idealized — higher cognitive functions, such as planning in the blocks world, can be implemented successfully by such programs.

1 Introduction

How does intelligence happen? How can reasoning, problem-solving, decision-making, planning, empathy, language, art be achieved through the activity of neurons and synapses? Despite tremendous advances over the past decades in our understanding of neural mechanisms — increasingly assisted and propelled by machine learning — we are still very far from answering the overarching question: *how does the brain beget the mind?* The difficulty lies in the huge gap of scale and methodology between Experimental Neuroscience and Cognitive Science. This frustration was articulated in a most eloquent way by Nobel laureate Richard Axel, who declared in a 2018 interview (Axel 2018): *"We do not have a logic for the transformation of neural activity to thought and action. I consider discerning [this logic] as the most important future direction in Neuroscience"*.

The *Assembly Calculus* (AC) is a recently proposed formal computational system (Papadimitriou et al. 2020). As far as we know, it is the only computational system in the literature whose explicit purpose is to bridge through computation the gap between neurons and intelligence — that is to say, to function as Axel's logic. The basic data item of the AC is the *assembly of neurons,* a large stable set of neurons believed to represent an idea, object, word, etc., while its operations (project, associate, merge, etc.) create and manipulate assemblies in response to stimuli and other brain events. Importantly, these operations can be provably simulated through the activity of stylized neurons and synapses. All said, the AC is a Turing complete computational system founded firmly on the basic principles of Neuroscience. In the next section, we provide a comprehensive introduction to the AC; however, the interested reader may want to read (Papadimitriou et al. 2020).

So, is the AC the bridging "logic" sought by Axel? One avenue for pursuing this important question is to demonstrate empirically that reasonably complex cognitive phenomena can be formulated and implemented in the AC framework. Indeed, in the original paper (Papadimitriou et al. 2020) it was argued that aspects of language generation can be handled by the operations of the AC, while in a very recent paper (Mitropolsky, Collins, and Papadimitriou 2021), a Parser implemented in the AC was demonstrated to analyze syntactically reasonably complex sentences of English, and it was argued that it can be generalized to more complex features as well as other natural languages.

Our contribution in this paper is to demonstrate that a program in the AC is capable of implementing reasonably sophisticated stylized planning strategies – in particular, heuristics for solving tasks in the blocks world (Gupta and Nau 1991; Slaney and Thiébaux 2001). A *blocks-world configuration* is defined by a set of *stacks,* where a stack is a sequence of unique *blocks,* each sitting on top of the previous one. A stack of size one is just a *block sitting on the table* (see e.g. Fig. 1-A). A configuration can be manipulated by moving a block from the top of a stack (or from the table) to the top of another stack (or to the table). A *task* in the blocks world is the following: Given a starting configuration C_{init} and a goal configuration C_{goal}, find a sequence of actions which transforms C_{init} into C_{goal}. It was shown in Gupta and Nau (1992) that solving a task in the blocks world with the smallest number of actions is NP-Complete, and it was observed that the following provides a simple 2-approximation strategy: *Move to the table all blocks that are not in their final positions, and then move these blocks one by one to their final positions.*

Here we implement this strategy in the AC. From the exposition of this implementation and demonstration — which happens to employ representations and structures of a different style from those needed for language tasks (Papadimitriou et al. 2020; Mitropolsky, Collins, and Papadimitriou 2021) — we believe that it will become clear that more complicated heuristics for solving related tasks can be similarly implemented in the AC.

In fact, the kind of representations needed for planning, involving long "chains" of assemblies linked through strong synaptic connections, reveals a limitation of the AC which was not apparent before: we find empirically that there are limits — depending on the parameters of the execution model, such as the number of excitatory neurons per brain area, synaptic density, synaptic plasticity, and assembly size — on the length of such chains that can be implemented reliably. As chaining is also used in the Turing machine simulation demonstrating the completeness of the AC (Papadimitriou et al. 2020), such limitations are significant because they bound from above the space complexity — and therefore the parallel time complexity — of AC computations. We briefly discuss and quantify this issue in the experimental validation section.

1.1 Related Work

Terry Winograd introduced the blocks world half a century ago as the context for his language understanding system SHRDLU (Winograd 1971), but since then blocks-world planning has been widely investigated, primarily because such tasks appear to capture several of the difficulties posed to planning systems (Gupta and Nau 1991, 1992). There has been extensive work in AI on blocks world problems, including recently on leveraging ANNs for solving them, and learning to solve them from examples (e.g., the Neural Logic Machines of Dong et al. (2019), or Neural Turing Machines, which are used for related problem-solving tasks (Graves, Wayne, and Danihelka 2014)).

Bridging the gap between low-level models of neural activity in the brain and high-level symbolic systems modelling cognitive processes is a fundamental open problem in artificial intelligence and neuroscience at large (Doursat 2013; Chady 1999). Several computational cognitive-science papers address the problem of solving (or learning to solve) block-worlds tasks in higher-level computational models of cognition, such as ACT-R or SOAR (see for instance Kennedy and Trafton (2006); Kurup (2008); Panov (2017)). In contrast to the present paper, however, these works utilize high-level languages and data structures for the programming of these systems, without providing a link, as we do, to the behavior of stylized neurons and synapses, in an effort to remain as faithful as possible to the ways animal brains would solve these tasks. Less related to our problem is the literature on block stacking (see, for example, Hayashi (2007); Tian, Luo, and Cheung (2020)). These papers the focus on the ability of humans and chimpanzees to place a block on top of an existing tower without toppling it. Finally, it is worth mentioning some previous works on solving planning tasks through spiking neural networks, such as (Rueckert et al. 2016; Basanisi et al. 2020), in which the attention is more focused on learning world models.

A spiking neural network framework not unlike ours is Nengo (Bekolay et al. 2014). One important difference is that our framework focuses on the known behavior called assemblies which enable higher levels of abstraction such as the AC, and carrying out far more advanced tasks such as in (Mitropolsky, Collins, and Papadimitriou 2021) and the present paper.

2 The Assembly Calculus

The Assembly Calculus (AC) (Papadimitriou et al. 2020) is a computational system for modeling a dynamical system of firing neurons. In this system, there is a finite number of *areas*, each containing n neurons. The neurons of an area form a random Erdős-Rényi directed graph $G_{n,p}$, where p is the probability that two neurons of the area are connected. Moreover, certain ordered pairs of areas are connected one to another through an Erdős-Rényi directed bipartite graph $G_{n,p}$. The directed connections between areas are called *fibers*.

In the AC, neurons in an area A fire in discrete time steps, and are subject to stylized forms of *inhibition* and *plasticity*. For what concerns inhibition, at any time step, we assume only k_A of the n neurons fire, that is, the ones that previously received the highest total input from all other areas — these k_A neurons are sometimes called the *winners*. Plasticity is modelled by assuming that, if, at a given time step, neuron x fires and, at the next time step, an out-neighbor neuron y of x fires, then the weight of the synapse from x to y (which is 1 at the beginning) is multiplied by $(1+\beta_A)$, where $\beta_A > 0$. In the original definition of the AC, a process of *homeostasis* was also modelled through a periodic renormalization, at a different time scale, of the synaptic weights, in order to avoid the generation of huge weights. Such process is of course part of any realistic brain system, also providing a mechanism for *forgetting*. We will not implement here this feature of the model.

Lastly, yet importantly, the AC allows *inhibiting* and *disinhibiting* areas and fibers at different time steps. The exact mechanism through which areas and fibers are (dis)-inhibited may vary; in a recent paper modeling syntactic processing using the AC, Mitropolsky, Collins, and Papadimitriou (2021) model specific neurons as having (dis)-inhibitory effects on areas or fibers. In this work, (dis)-inhibition is always determined by which areas and fibers fired at the previous time step.

The most important emergent object in the AC is the *assembly*, that is, a stable set of k_A highly interconnected neurons in an area A. It is emergent in the sense that assemblies are not a primitive of the model; instead, they are formed through its more basic operations. Assemblies are by now well known and widely studied in neuroscience, and are thought to represent concepts, ideas, objects, words, etc., and are increasingly believed in recent years to play a central role in cognitive processes (Buzsáki 2010), often called "the alphabet of the brain" (Buzsáki 2021). In terms of classical thinking in AI, one could think of assemblies as the boundary in the brain between sub-symbolic and symbolic computation.

The AC makes possible to perform certain *operations* with assemblies, described next — in fact, it is through these operations that assemblies are created, in a way that guarantees high connectivity. In Papadimitriou et al. (2020), the authors demonstrate, both mathematically and through simulation, that these operations are "possible" in the sense that they can be stably performed with high probability in the dynamical system of neurons outlined in the previous paragraphs. In this paper, we mostly make use of one of these operations: *projection* of an assembly in an area into another assembly in another area.

Let us assume that an assembly x of k_A neurons of the area A has just fired into an area B (presumably through a disinhibited fiber going from A to B), and assume that B was quiescent at that time (no neurons were firing). This will result in a set w_1 of k_B neurons (the winners) firing at the next time step. Next, the neurons in B will receive inputs not only from the k_A neurons of the assembly in A, which will continue to fire, but also from the neurons in w_1 through recurrent connections within B: this will result in a set w_2 of k_B neurons, (the new winners) firing at the next time step, and so on. It has been proved that, under appropriate values of the parameters n, k_A, k_B, β, and p, this process converges with high probability to an assembly y of k_B neurons in B, which is called the projection of x into B and can be thought as a copy of x in B such that, from now on, y will fire every time x fires.

For a complete description of the AC the reader is referred to Papadimitriou et al. (2020), where in addition to stability of various assembly operations, it is also proved that, under certain assumptions, this computational system is capable of performing arbitrary computations as long as the space required does not exceed n/k_A (under much milder assumptions, $\sqrt{n/k_A}$). In this paper, similarly to the Parser of Mitropolsky, Collins, and Papadimitriou (2021), our AC programs work by projecting between all pairs of disinhibited areas along disinhibited fibers at each time step. For brevity, this operation, i.e. a simultaneous set of projections between multiple areas, is called *strong projection.*

Our AC programs are described with the operations in Table 1. Inhibition and disinhibition are primitives of the AC system, whereas strong projection (tantamount to a set of simultaneous projections) is an emergent property of the AC's dynamical system. We use several other such "emergent" operations, i.e., that are not primitives of the AC system, but can be stably implemented with its basic operations. For example, we will make use of an operation which allows us to verify whether in a specific area there exists a stable assembly (as the result of a projection). In Table 1, we summarize the operations (primitive and non primitive) of the AC system, that we will use in this paper. Note that the block activation operation is a special operation, which causes an assembly (in a special area BLOCKS) corresponding to the named block to fire.

3 The Blocks World AC Program

A blocks world (BW) configuration is a set of *stacks,* where each stack is a sequence of *blocks*, from top to bottom. Each block is assumed to be a unique integer between 1 and s. Two BW configurations, the initial and the target configuration, constitute the input to the AC program (see Figure 1-A). We shall at first concentrate on *configurations with a single stack* — already a meaningful problem — and we shall eventually graduate to multiple stacks (see subsection 3.5 below). We next describe four AC programs: (a) a program that takes the input — a sequence of integers representing a stack — and creates a list-like structure, in a set of brain areas and fibers, for representing the stack; (b) a program that removes the top block of a stack thus represented; (c) a program that adds a new block to the represented stack; and (d) a program for computing the *intersection* of two stacks represented this way, that is, the longest common suffix of the two sequences, read from bottom to top.

All four programs work on a common set of brain areas connected with bi-directional fibers: the area BLOCKS contains a fixed assembly for every possible block (these assemblies are special, in that each can be activated explicitly as the presentation of the corresponding number in the input). There are four other areas used in our AC programs: HEAD, NODE$_0$, NODE$_1$, and NODE$_2$. HEAD is connected to the NODE$_0$ area via fibers, while each NODE area is connected to BLOCKS, and to each other in the shape of a triangle: NODE$_0$ is connected with NODE$_1$, which is connected with NODE$_2$, which is connected with NODE$_0$ (see Figure 1-B). All of these areas are standard brain areas of the AC system, containing n randomly connected neurons of which at most k fire at any time.

3.1 The Parser

The parser (see Algorithm 1) processes each block in a stack sequentially, starting from the top. When it analyses the first block (see lines 1-3), the three areas BLOCKS, HEAD, and NODE$_0$, and the fibers between HEAD and NODE$_0$ and between NODE$_0$ and BLOCKS are disinhibited. The block assembly is then activated and a strong projection is performed, thus creating a connection between the assembly in BLOCKS corresponding to the block and an assembly in NODE$_0$, and between this latter assembly and an assembly in HEAD (see the red dashed lines in Figure 1-C1). Successively, the HEAD area and the fibers between HEAD and NODE$_0$ and between NODE$_0$ and BLOCKS are inhibited. For each other block in the stack (see lines 5-8), the NODE area next to the one (i.e., NODE$_{i \bmod 3}$) currently disinhibited (i.e., NODE$_{i+1 \bmod 3}$) is disinhibited, and the fibers between this NODE area and the BLOCKS area and between the two NODE areas are disinhibited. The next block assembly is then activated and a strong projection is performed, creating a connection between the assembly in BLOCKS and

Operation	Input	Semantics
activateBlock (b)	Block number b	Makes the assembly of the block b in the area BLOCKS fire
disinhibitArea (A)	Set A of areas	Disinhibit all the areas in A
disinhibitFiber (P)	Set P of pairs of areas	Disinhibit the fibers between any pair of areas in P
inhibitArea (A)	Set A of areas	Inhibit all the areas in A
inhibitFiber (P)	Set P of pairs of areas	Inhibit the fibers between any pair of areas in P
isAssembly (a)	Area a	Verify whether there is an active assembly in the area a
project (a_1, a_2)	Areas a_1 and a_2	Executes a projection of (the active assembly in) the area a_1 to the area a_2
strongProject()		Executes a strong projection involving all the disinhibited areas and fibers

Table 1: The AC operations (primitive and non primitive) used in the paper.

```
Algorithm 1: PARSER ($S$)
    input: a stack $S$ of blocks $b_1, b_2, \dots, b_s$.
1   disinhibitArea ({BLOCKS, HEAD, NODE$_0$}); disinhibitFiber ({(HEAD, NODE$_0$), (NODE$_0$, BLOCKS)});
2   activateBlock ($b_1$); strongProject();
3   inhibitArea ({HEAD}); inhibitFiber ({(HEAD, NODE$_0$), (NODE$_0$, BLOCKS)});
4   foreach $i$ with $2 \leq i \leq s$ do
5       $p = (i-2) \bmod 3$; $c = (i-1) \bmod 3$;
6       disinhibitArea ({NODE$_c$}); disinhibitFiber ({(NODE$_p$, NODE$_c$), (NODE$_c$, BLOCKS)});
7       activateBlock ($b_i$); strongProject();
8       inhibitArea ({NODE$_p$}); inhibitFiber ({(NODE$_p$, NODE$_c$), (NODE$_c$, BLOCKS)});
9   end
10  inhibitArea ({BLOCKS, NODE$_{(s-1) \bmod 3}$});
```

an assembly in the NODE area just disinhibited, and between this latter assembly and the assembly previously activated in the previous NODE area (see the red dashed lines in the figures 1-C2,C3,C4). After this and before the next block, this latter NODE area and the fibers between it and the NODE area after it, and those between the NODE area after it and the BLOCKS area, are inhibited.

The final data structure is a *chain* of assemblies starting from an assembly in HEAD and passing through assemblies in the NODE areas (see Figure 1-C6). Note that this chain can contain more than one assembly in the same NODE area: for instance, in Figure 1-C6, the chain contains two assemblies in NODE$_0$ and NODE$_1$. Each assembly in the chain is also connected to the assembly in BLOCKS corresponding to a block in the stack. For instance, the sequence of such assemblies in Figure 1-C6 corresponds to the sequence of blocks $4, 5, 3, 1, 2$, which is exactly the sequence of blocks in the stack from top to bottom (see the left part of Figure 1-A). Note that Algorithm 1 uses a constant number of brain areas (that is, five), independently of the number of blocks in the stack.

3.2 Removing the Top Block

In order to implement in AC the algorithm which transforms an input stack of blocks into a target stack of blocks, we start by describing an AC program to remove a block from the top of a stack. This program uses the same areas and fibers of the parser described in the previous section (see Figure 1-B), with the addition of fibers between HEAD with NODE$_1$, and HEAD with NODE$_2$. Intuitively, these fibers are needed to allow changing the head of the chain representing the current stack, without having to shift all the assemblies one position to the left.

The AC program, which "removes" the block from the top of the stack, uses the connections created by the parser in order to activate the assembly in the NODE$_1$, which is connected to the block just below the top block (that is block 5 in Figure 1-D1,D2). This is done by projecting from the HEAD into NODE$_0$, and projecting from NODE$_0$ into the NODE$_1$ (see Figure 1-D1). Through strong projection, the program successively creates a new connection from the active assembly in the NODE$_1$ area to a new assembly in the HEAD area (see the red dashed line in Figure 1-D2).

Note that the connections between the light gray assemblies in Figure 1-D2 are still active, but they will not be used in the future since the last active assembly in the HEAD area is now connected to the assembly in the NODE$_1$ area. These connections, indeed, might later disappear because of a process of homeostasis, which can be modeled in the AC system through a sort of "renormalization" (as described in Papadimitriou et al. (2020)). In a certain sense, the system will slowly "forget" which block was on the top of the stack, before a removal operation.

The removal of the top block can be repeated as many times as the number of blocks in the stack. The only difference is that the activation of the assembly in NODE corresponding to the block below the top one is done by projecting HEAD into the NODE area corresponding to the top block, and then projecting from this NODE area to the one following it (in modular arithmetic).

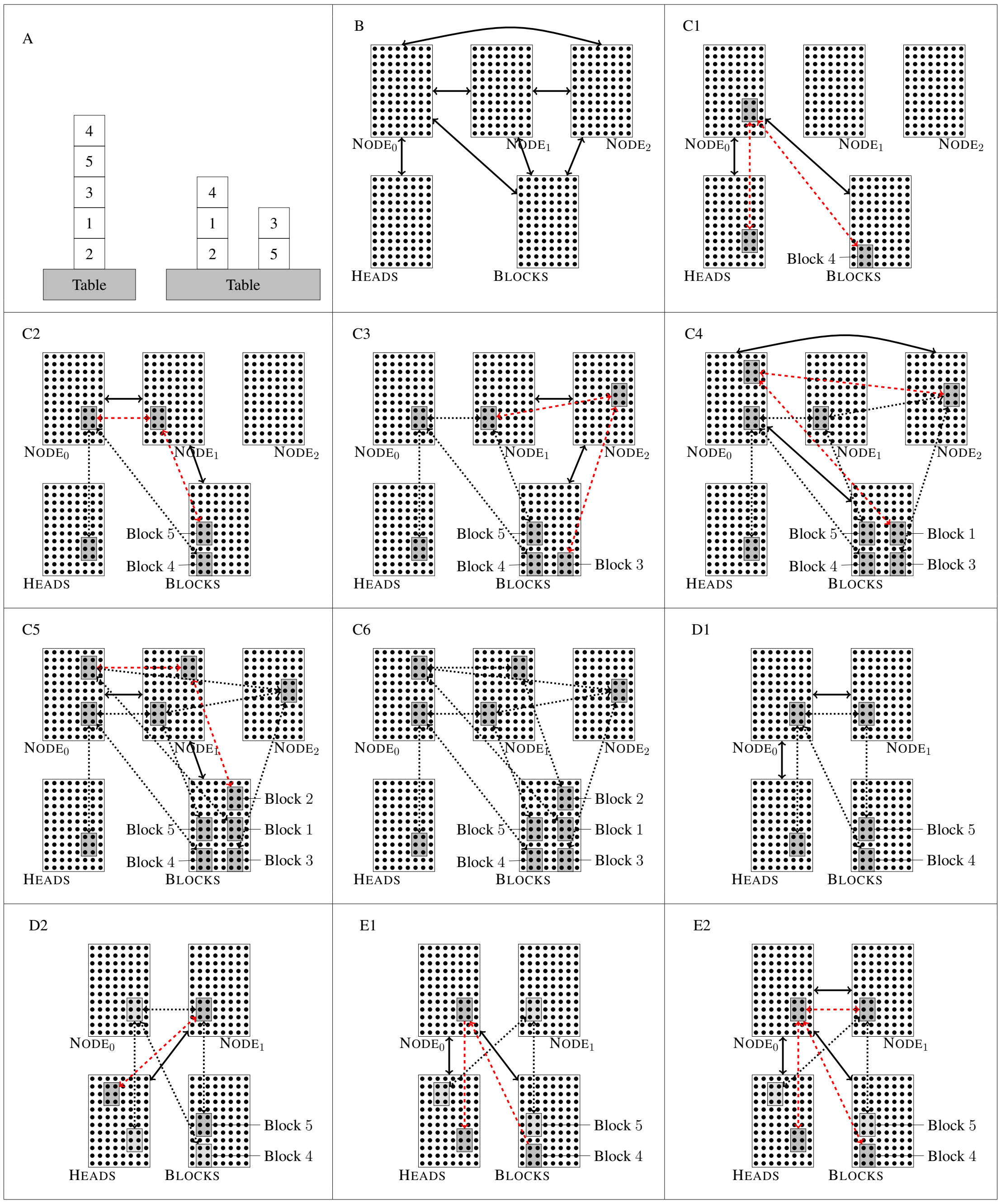


Figure 1: **A.** Two BW configurations. In the rest of the figure, we consider the BW configuration shown on the left. **B.** The five areas used by the parser AC program, along with the connections through fibers. **C1-6.** The behavior of the parser AC program. The black solid lines denote the fibers of Figure B which are disinhibited. The red dashed lines denote the newly created connections between assemblies in different areas, while the black dotted lines denote the connections previously created. **D1-2.** The behavior of the AC program which removes the block from the top of a stack, with input the data structure resulting from the parser execution (only the areas involved in the remove operation are shown). The black solid lines denote the fibers which are disinhibited. The red dashed lines denote the newly created connections between assemblies in different areas, while the black dotted lines denote the already existing connections. **E1-2.** The behavior of the AC program which put the block 4 on top of the stack, above the block 5. The black solid lines denote the fibers which are disinhibited. The red dashed lines denote the newly created connections (unidirectional and bidirectional) between assemblies in different areas, while the black dotted lines denote the already existing connections.

In order to maintain an updated representation of the blocks world configuration, we use four additional brain areas to store the chain of blocks which have been removed and that, hence, are currently on the table. This chain can be implemented in the AC system exactly the same way we did when parsing a stack of blocks. Then, when we want to read the current data structure stored in the AC system, we examine the stack of blocks represented in HEAD and the NODE areas, as well as the chain of blocks on the table in the additional areas.

3.3 Putting a Block on Top of the Stack

The second operation we need in order to implement a minimal planning algorithm for the blocks world problem is *putting* a block on top of the stack. The AC program, for this operation first projects the block from in BLOCKS into the NODE area preceding (in modular arithmetic) the NODE area currently connected to HEAD, and then projects the newly created assembly into HEAD (see Figure 1-E1). Successively, the program executes a strong projection between the four areas in order to correctly connect them (see Figure 1-E2). Once again, an active connection between the HEAD area and a NODE area will still exist after the execution of the AC program, but this connection will not be used in the future.

3.4 Computing the Intersection of Two Stacks

The pop and put operations described in the previous two sections are sufficient to implement a simple planning algorithm, which consists in moving all the blocks on the table (by using pop), and by then moving the blocks on the table on top of the stack (by using put) according to the target stack.

In order to improve this algorithm and execute the two-approximation algorithm described in the introduction, we need an AC program which implements a third operation, that is, finding the *intersection* of two stacks. This operation looks for the common sub-stack of the two stacks (starting from the bottom) and return the highest block in this sub-stack. Then only the blocks above this block have to be moved on the table and reassembled in the right order.

In a nutshell, this can be achieved in AC by first reaching the bottom of the two stacks which have to be compared, and then proceeding upwards until we find two different blocks, or the end of one of the two stacks.

3.5 Multiple Stacks

So far in this exposition we have concerned ourselves with configurations consisting of one stack. In our experiments (see the next section) we have implemented up to five stacks by employing a different set of four areas for each stack. This is a bit unsatisfactory, because it implies that the maximum number of stacks that can be handled by the brain is encoded in the brain architecture. There is a rather simple — in principle — way to achieve the same effect by re-using the same four areas; we have an initial implementation of this idea, which we intend to test in the future.

With multiple stacks one has to solve the *matching problem:* identifying pairs of stacks in the input and output that must be transformed one to the other. Naively, this can be done by comparing all pairs of stacks, but this entails effort that is quadratic in the number of stacks. This latter strategy is the one currently employed in our experiments. In the future, we intend to test a more principled way, based on *hashing* the stacks into their bottom element, and attending to any collisions.

4 Experiments

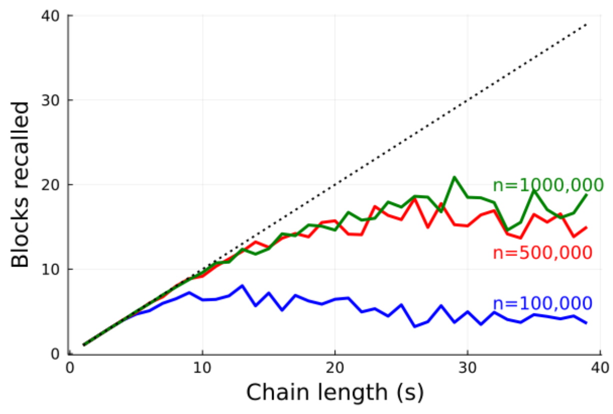
A software system for programming in the AC, as well as implementations of the algorithms described in this paper, have been written in Julia (Bezanson et al. 2017). We make use of the Java generator for BW configurations available at Koeman (2020), based on Slaney and Thiébaux (2001). We ran experiments on over 100 blocks-world configurations, with up to five stacks and 10, 20, and 30 blocks. The algorithm worked correctly in every instance. We have used various settings of the parameters n, k, p, β – a particularly good set of parameters is $n = 10^6, k = 50, p = 0.1, \beta = 0.1$. Interestingly, the algorithms do not work in all parameter settings, because of limits on the chaining operation (see the next discussion). The Julia source code can be found at (jBrain 2021).

In general, the amount of rounds of strong project (parallel spikings of neurons) needed to carry out the BW tasks seems to be around 35 spikes per block processed (parse, popped, or pushed), which, assuming roughly 50 Hz spikes for excitatory neurons in the brain, is around 1.4 seconds per operation.

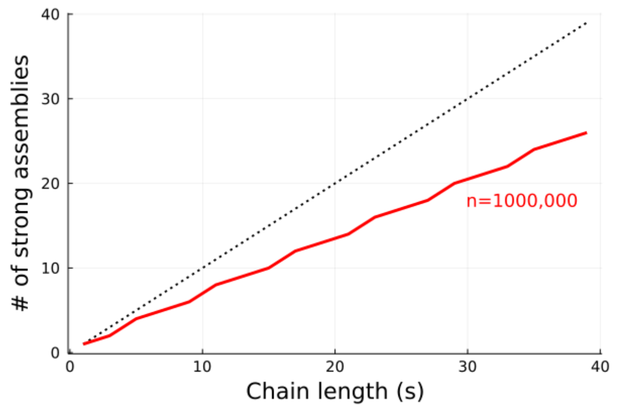
Limits of the AC. An unexpected finding of our simulations is that they are stable only under very specific parameter settings. The bottleneck of the planning algorithms is in parsing the chain of blocks, that is, memorizing the sequence of blocks so they can be read out reliably. In isolation we call this operation “chaining”.

The results in this section, which describe some properties and limits of chaining, can be viewed as theoretical properties of the AC. First, we find it is only possible to chain a rather limited number of blocks. For instance, even though with $n = 10^6$ and $k = 50$ there is, at least in theory, space for $10^6/50 = 200000$ non-overlapping assemblies, even with strong p and β, we can only reliably chain up to 20 blocks. This is illustrated in Figure 2a, which shows how many of s blocks were successfully read out after chaining. Generally, for higher values of n (and a higher $n : k$ ratio), longer portions of the chain tend to be correctly stored, but the operation is highly noisy: in some trials it will fail and then succeed for a *longer* chain. Indeed, unlike the assembly operations described in Papadimitriou et al. (2020) (Project, Merge, and so on) which are either stable with overwhelming probability under appropriate parameters, or do not succeed if the parameters are not appropriately strong, chaining appears to push the computational power of the AC to its limits, and often succeeds or fails between repeated trials with the same parameters.

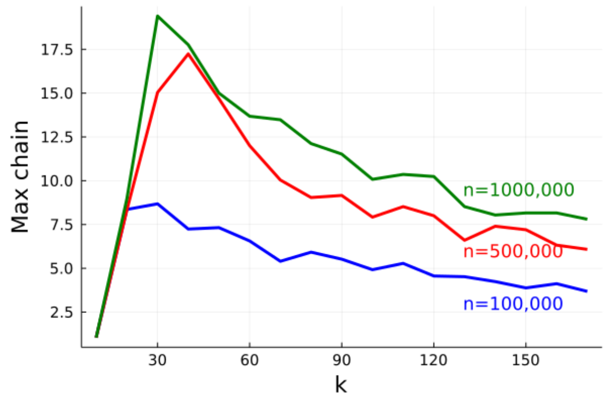
One can also look at a related property: after chaining, how many of the assemblies in the NODE$_i$ areas during read-out are “strong” in the sense that they pass ISASSEMBLY()



(a)



(b)



(c)

Figure 2: Experiments on the "chaining" operation, the bottleneck of the AC planning algorithm. (a) shows number of blocks correctly chained for various chain length; (b) shows number of "strong" assemblies formed in chaining; (c) shows maximal chain length that is correctly parsed for varying k. (b) and (c) show averages over 50 trials per parameter setting (exact numbers, including sample standard deviation, are provided in the full version at (d'Amore et al. 2021)). In these charts, $p = \beta = 0.1$ was used, in (a) and (b) $k = 50$.

with a high threshold (i.e. firing those k neurons recursively results in the same set of k winners)? Interestingly, this proportion, which is significantly less than the maximum of s, does not change significantly when we vary n, p, β– there appears to be a natural proportion of strong assemblies formed during chaining (Figure 2b).

Finally, in Figure 2c we varied k and found the maximally long chain that succeeded completely. These experiments again showed that for higher $n : k$ ratio, longer chains are possible, and that for each setting of n there is a narrow window of optimal k that allows for the longest chains– above of this range, as we increase k the maximum chain does not change, i.e. it appears to settle to some natural lower bound. A more thorough analysis of chaining is an important direction in AC theory, since such maneuvers could be subroutines in various cognitive processes (for instance, Mitropolsky, Collins, and Papadimitriou (2021) suggest using it for processing chains of identical parts of speech, such as multiple adjectives in a noun phrase).

5 Conclusions and Future Directions

The aim of this work is not so much to produce a performing system, but to demonstrate experimentally that reasonably large and complex programs in the assembly calculus can execute correctly and reliably, and in particular can implement in a natural manner planning strategies for solving instances of the blocks world problem. In fact, the implementation of these strategies is based on the realization of a *list-like data structure* which makes use of a *constant* number of brain regions. Confirming theoretical insights, we have experimentally found that the structure's reliability depends on the ratio between the number of neurons and the size of the assemblies in each region — even though the dependency was a bit more constraining than we had expected. The reasons and extent of this shortcoming must be the object of further investigation.

We have also shown how simple manipulations of the data structure (such as the top, pop, and append operations) can be realized by making use of a constant number of brain regions. These manipulations allowed us to implement planning strategies based on two basic kinds of moves, that is, moving the block from the top of a stack to the table, and putting a block from the table to the top of a stack. All our programs work for an *arbitrary* number of blocks and a *bounded* number of stacks — while current work involves implementing a version with an arbitrary number of stacks.

After syntactic analysis in language and blocks world planning, what comes next as a compelling stylized cognitive function, which could be implemented in the AC? There is work currently in submission dealing with *learning* though assemblies of neurons. Two further realms of cognition come to mind, and they happen to be closely related: *Reasoning,* as well as *planning and problem solving* in less specialized domains than BW. It would be interesting to figure out the most natural way for assemblies and their operations to carry out deductive tasks, and, even more ambitiously, to carry out planning in the context of logical and constraint-based formalisms of planning, see for example Wilkins (1988).

Acknowledgements

CHP's research was partially supported by NSF awards CCF1763970 and CCF1910700, by a research contract with Softbank, and a grant from CAIT. Furthermore, the authors are grateful to the OPAL infrastructure from Université Côte d'Azur for providing resources and support.

References

Axel, R. 2018. Q&A. *Neuron*, 99(6): 1110–1112. doi: 10.1016/j.neuron.2018.09.003.

Basanisi, R.; Brovelli, A.; Cartoni, E.; and Baldassarre, G. 2020. A generative spiking neural-network model of goal-directed behaviour and one-step planning. *PLOS Computational Biology*, 16(12): 1–32.

Bekolay, T.; Bergstra, J.; Hunsberger, E.; DeWolf, T.; Stewart, T.; Rasmussen, D.; Choo, X.; Voelker, A.; and Eliasmith, C. 2014. Nengo: a Python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, 7(48): 1–13.

Bezanson, J.; Edelman, A.; Karpinski, S.; and Shah, V. B. 2017. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1): 65–98.

Buzsáki, G. 2010. Neural Syntax: Cell Assemblies, Synapsembles, and Readers. *Neuron*, 68(3): 362–385.

Buzsáki, G. 2021. *The Brain from Inside Out*. Oxford: Oxford University Press, reprint edition edition. ISBN 978-0-19-754950-6.

Chady, M. 1999. Modelling Higher Cognitive Functions with Hebbian Cell Assemblies. In Hendler, J.; and Subramanian, D., eds., *Proceedings of AAAI/IAAI 1999, July 18-22, 1999, Orlando, Florida, USA*, 943. AAAI Press.

d'Amore, F.; Mitropolsky, D.; Crescenzi, P.; Natale, E.; and Papadimitriou, C. H. 2021. Planning with Biological Neurons and Synapses. https://hal.archives-ouvertes.fr/hal-03479582.

Dong, H.; Mao, J.; Lin, T.; Wang, C.; Li, L.; and Zhou, D. 2019. Neural Logic Machines. In *International Conference on Learning Representations*.

Doursat, R. 2013. Bridging the Mind-Brain Gap by Morphogenetic "Neuron Flocking": The Dynamic Self-Organization of Neural Activity into Mental Shapes. In *AAAI Fall Symposia*.

Graves, A.; Wayne, G.; and Danihelka, I. 2014. Neural Turing Machines. arXiv:1410.5401.

Gupta, N.; and Nau, D. S. 1991. Complexity Results for Blocks-World Planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence - Volume 2*, AAAI'91, 629–633. AAAI Press. ISBN 0262510596.

Gupta, N.; and Nau, D. S. 1992. On the complexity of blocks-world planning. *Artificial Intelligence*, 56(2): 223–254.

Hayashi, M. 2007. Stacking of blocks by chimpanzees: developmental processes and physical understanding. *Animal Cognition*, 10: 89–103.

jBrain. 2021. https://github.com/piluc/jBrain.

Kennedy, W. G.; and Trafton, J. G. 2006. Long-term Symbolic Learning in Soar and ACT-R. In *Proceedings of the Seventh International Conference on Cognitive Modeling*, 166–171.

Koeman, V. 2020. The Blocks World. https://github.com/eishub/blocksworld#readme. [Online; last access 08-September-2021].

Kurup, U. 2008. *Design and use of a bimodal cognitive architecture for diagrammatic reasoning and cognitive modeling*. Ph.D. diss., Graduate School of the Ohio State University.

Mitropolsky, D.; Collins, M. J.; and Papadimitriou, C. H. 2021. A Biologically Plausible Parser. In *Transactions of the Association for Computational Linguistics*. ArXiv: 2108.02189.

Panov, A. I. 2017. Behavior planning of intelligent agent with sign world model. *Biologically Inspired Cognitive Architectures*, 19: 21–31.

Papadimitriou, C. H.; Vempala, S. S.; Mitropolsky, D.; Collins, M.; and Maass, W. 2020. Brain computation by assemblies of neurons. *Proceedings of the National Academy of Sciences*, 117(25): 14464–14472.

Rueckert, E.; Kappel, D.; Tanneberg, D.; Pecevski, D.; and Peters, J. 2016. Recurrent Spiking Networks Solve Planning Tasks. *Scientific Reports*, 6.

Slaney, J.; and Thiébaux, S. 2001. Blocks World revisited. *Artificial Intelligence*, 125(1): 119–153.

Tian, M.; Luo, T.; and Cheung, H. 2020. The Development and Measurement of Block Construction in Early Childhood: A Review. *Journal of Psychoeducational Assessment*, 38(6): 767–782.

Wilkins, D. E. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc. ISBN 093461394X.

Winograd, T. 1971. Procedures as a representation for data in a computer program for understanding natural language. Technical report, Massachusetts Inst Of Tech Cambridge Project Mac.