Heuristic Search in Program Space for the AGINAO Cognitive Architecture*

Wojciech Skaba[1]

[1] AGINAO, Trubadurow 11, 80205 Gdansk, Poland
aginao@aginao.com

Abstract. AGINAO is a project to build a human level AGI system by applying the embodied approach on the NAO humanoid robot. A brief introduction to the AGINAO cognitive architecture is presented, followed by a presentation of the virtual machine design, the instruction set, and the algorithm to construct the building blocks of concept nodes: a heuristic search in the space of tiny programs – the codelets of the emergent architecture. Unlike the universal search or evolutionary programming approaches, the created programs are not executed for evaluation of their fitness at this stage, leaving it for other modules. The algorithm focuses on avoiding apparently useless pieces of code, yet covering the remaining program space uniformly.

Keywords: AGI, NAO robot, heuristic program search, cognitive architecture

1 Introduction

This paper introduces the AGINAO project – yet another attempt to build an artificial general intelligence (AGI) system capable of matching or even exceeding the human level (HL) of intellectual skill. Our approach uses the NAO ver. V3+ humanoid robot manufactured by Aldebaran Robotics [1], as a testbed. Since the processing power of the robot's built-in computer seems insufficient for the cognitive task, with as much as up to 90% of the available resources being consumed by the robot's internal processes, the NAO is merely set as a front end to communicate the sensory and actuators information via a connectionless UDP link to a powerful host. The NAO seems a good choice for the robotic/embodied approach to AGI, being a trade-off between inexpensive toys, mostly fragile and lacking a vision system, and adult size sophisticated prototypes, most of which are not commercially available. NAO is equipped with two 640x480 color cameras, 4 microphones, 2 speakers, tactile/force sensors, gyro, accelerometer, dozens of joints with motors/sensors, WiFi/twisted-pair connection and GEODE 500 MHz CPU running Linux.

The robot/host communication is completed and current efforts focus on writing the cognitive engine software. When ready, one or more NAOs would be tested in a preschool-like environment, with the intention that most of the robot's knowledge and capabilities will be acquired with machine learning methods during interaction in the real world. We also conjecture that the power of a contemporary desktop is sufficient for achieving the HL AGI, and the rush for a virtually unlimited power is unjustified.

* Submitted to AGI-11 conference

1.1 AGINAO cognitive architecture

The fundamental notion of the AGINAO cognitive engine is a *concept,* an entity intended to be close to what a 'concept' in cognitive sciences means. Without going into details of the theory of mind behind our cognitive architecture, below only those aspects of the design are presented that are indispensable for the main topic of this paper, namely, the heuristic program search algorithm. Technically, the concept is a structure consisting of some data fields, not listed below, and a program – or more precisely a subroutine – of the form:

```
int *prog(const int *src1,..., const int *srcN)
   { static int memory[size];
     int *loc1,..., *locK;  // local variables
     // program code
   }
```

A program may be either a short piece of code, or a sophisticated one, calling other programs (concepts). All concepts form a dynamic and continuously changing hierarchy of interconnected concepts (a directed graph with cycles). New concepts emerge, the useless or obsolete ones are discarded. Even more likely, it is the connections rather than the whole concepts that are added or removed. The output of a concept may be connected to a number of other concepts (order of inputs matters). The only data type to transfer information between the concepts is an array of integers of a variable but known size, quite a general purpose type, anyway.

One may think of a concept program as like an ANN neuron with N inputs and one output. Such an analogy is insufficient, however, as the programs are not executed in the hierarchy. It is better to think of the collection of concepts as a depository of programs that may be launched on request as *runtimes*. More than one copy of the same code (possibly a thousand) may be executed concurrently, processing different data. One may also envision concepts as species, and runtimes as individuals of a species, and connections as dependencies between the species, all of them in an artificial environment. The primitive organisms feed more complex. Individuals come forth and pass away. Species rise and fall. Once an ecological niche is empty, it will be filled with new species. The natural world analogy is incomplete, however. The species do not evolve. Lack of living (executed) individuals doesn't mean a species is immediately extinct. Two different species may share the same 'genotype', as e.g. a program to add two integers may have a different semantic meaning in various locations of the concept hierarchy.

At the bottom of the hierarchy there are *atomic concepts,* non-removable ones, each containing a predefined piece of code ("bottom" does not necessarily mean "root" in the hierarchy). The atomic concepts constitute the low level sensory and actuator functions. Once a new sensory arrives from the robot, a related piece of code is launched as a runtime. Likewise, an atomic actuator's code may be executed to cause a transfer of some control data to the robot.

As pictured above, a program may also contain a static local integer array, in which case different concurrent runtimes may overlap and share the memory. Both, the static local memories and the structure of the concept hierarchy, may be regarded as a long-

term procedural, semantic and episodic memory of an artificial brain. The data transferred between the runtimes, on the other hand, may be understood as short-term or perceptual memory.

The inputs of a concept program are of type `const` and may not be affected. This is because each input is an output of another runtime that may be connected to many concurrently executed runtimes of the same or different concepts.

1.2 Life cycle of a runtime

Runtime execution is controlled by the following variables:

- expiration time – a real time deadline
- resources – a limit of the virtual machine cycles to be utilized
- priority – a parameter to control the order of execution

A runtime may be in one of the following six states:

PENDING – runtime is requested but awaiting other inputs to arrive. If a concept node has a single input only (one parameter to pass), it switches immediately to the next state. In other cases, more than 90% of the PENDING runtimes would possibly expire before the requested data is delivered.

CREATED – ready to run runtime is awaiting execution in a priority queue. Temporal expiration rules apply.

EXECUTED – runtime code is being executed and its resources are exhausted. Context switching applies and the runtime may be placed in the priority queue again, or even preceded by more urgent tasks, and discarded prematurely due expiration.

SLEEP – if a *temporal instruction* is encountered, a runtime switches into a waiting state, until the rise time or expiration time comes, whichever is earlier.

TERMINATED – runtime has finished execution in a non-fatal manner and returned an output array. The related reinforcement learning (RL) values are updated. If still having enough resources, the runtime starts its exploration/exploitation routine. A concept that has launched the terminated runtime contains a dynamic list of other concepts, i.e.: next actions to be selected. Contrary to the common convention, we would use the term *exploitation* for any – not only greedy – selection of the next action to be performed, and *exploration* for the process of adding a new concept to the list of concepts. Selecting an action means forwarding the output array to a concept, in order that a runtime be launched. One of the exploration methods implies creating a new concept from scratch – that is the topic of this paper. The runtimes of the atomic sensory concepts come forth in the TERMINATED state immediately after being launched.

DISCARD – if a terminated runtime has already exhausted its resources for exploration/exploitation, or expired, or the execution reported a fatal error, it goes into its final state. The actual releasing of the memories may be postponed until all of the dependent runtimes report unlocking the runtime's output.

A new concept to be added to the list of linked concepts in the exploration process may be selected/created in one of the following ways:

CREATE_NEW – make a new from scratch – the topic of this paper.

JUMP_EXISTING – link to a concept already settled in the hierarchy – it may cause a graph cycle to emerge.

COPY_EXISTING – copy the code of another concept to make a new one.

INTEGRATE – join two or more directly connected concepts to form a new one. The integration of the concepts is the basic technique for evolving the sophisticated concepts from the simpler ones, not discussed in this paper, however.

1.3 Sample cognitive processes

The following is a brief description of sample examples of low level cognitive processes related to the perceptual visual sensory.

Imagine that 3 visual pixels arrive from the robot, each being a 5-tuple of the form `{x,y,Y,U,V}`, i.e. an array of 5 integers. A separate process causes a sensory runtime to be launched only if the amplitude of the signal (the difference between the current value and the former value) exceeds a predefined limit. If we are lucky, we get three runtimes in TERMINATED state, each outputting pixel coordinates and YUV value. Then, three related exploitation mechanisms select the linked concepts, forward the output and launch new runtimes in PENDING state. Happily, one of the linked concepts accepts 3 inputs, and its embedded program returns "1" if the 3 coordinates form a spatial pattern of being arranged in line, or exits otherwise.

Now imagine an object moving in the visual area, causing 3 other pixels to arrive at three consecutive points in time. Unfortunately or not, the line detecting concept wouldn't have a chance to be launched, because the first arriving pixel runtime would expire before the last one is present. Fortunately, the first two pixels pass through intermediate concepts to execute a temporal WAIT instruction, and copy the single input to output undisturbed. Finally, a concept program to detect a spatial-temporal pattern of a moving pixel is launched.

1.4 Inspiration and similar approaches

The foundations of the AGINAO cognitive architecture have been once influenced by the idea of Hierarchical Temporal Memory (HTM) [2]. Hawkins highlighted two important ideas: 1) most real-world environments have both temporal and spatial structure and a single algorithm to discover these patterns should take both aspects into account; 2) Better results can be achieved if the processing is conducted simultaneously at all levels of the hierarchy.

Minsky in *The society of mind* [3] proposed to build a structure of many little parts, mindless agents. When we join these agents in societies, this leads to intelligence. Hofstadter's *Coderack* [8] contains small pieces of programs, waiting to be executed.

The EPIC cognitive architecture [4] consists of a set of interconnected processors that operate simultaneously and in parallel. When a processor receives an event, it can generate output to other processors by creating a new event of the appropriate type. As the eyes move around the visual scene, a complete and continuous representation of the objects currently present in the visual situation will be built up and maintained

in the perceptual store, allowing the cognitive processor to make decisions based on far more than the properties of the currently fixated object [7].

At the core of the LIDA architecture [5] functions a network of large-and-growing collection of codelets, special purpose active processes, represented in a few lines of executable code, that can recognize a stimulus and pass activation to nodes in the slipnet to which they are linked, and those nodes pass activation in turn to other nodes to which they are related (linked), until a subset referred to as a percept is stable.

Computations in DUAL [6] are performed by numerous simple micro-agents, hybrid at the micro-level devices, that consist of two parts: the symbolic L-Brain and connectionist R-Brain. The symbolic part represents some piece of knowledge, while the connectionist part represents its relevance to the current context. DUAL allows an uniform treatment of both declarative and procedural knowledge.

2 Heuristic Search in Program Space

By *virtual machine* (VM) we name a devised programmable computer, simulated in software on the host computer, having its own internal registers, flags, memory access and instruction set. Thousands of copies of the VM that share the same design may be run concurrently. Memory conflicts must be resolved. Execution time of a simulated instruction is longer than that of a machine code, but advantages prevail, to list:

- illegal operations may be detected without causing a truly fatal error,
- VM design is flexible and adjustable, to meet the requirements,
- resources utilization may be easily controlled,
- it has been observed that thread creation/detachment and context switching – the actions very common in our design – last relatively long in C++ gcc library.

The choice of a VM design is a question of a bit of art and a bit of applicability for the projected cognitive architecture. Even if the choice is not optimal, if only matches the criteria of an *Universal Turing Machine*, it would eventually work. What we really dream of is an instruction set that would let us create the basic concepts, that would in turn become 'instructions' of another more sophisticated Turing machine, possibly universal, simulated by our VM and featuring a dynamic instruction set, that would lead to creation of yet another even more sophisticated machine, and so on up the hierarchy of concepts, with no an apparent end.

One would assume that a very simple set, yet universal, would be the best and most flexible choice for our VM. On the other hand, however, a simple instruction set would be intractable by the heuristic search, thus useless for our desire to skip several orders of magnitude of processing time and make the building blocks favorable.

2.1 Virtual Machine design and instruction set

The VM has two general purpose registers of size `int`: A (or ACC) and IDX, and two binary flags: ZERO and MINUS. The ACC/IDX are set to 0 on launch, flags are

set to `false`. Variables are of type `int[n]`, the indices range `[0]...[n-1]`. The actual size of an `int` vector is accessible via designated instruction. The `int` is a 16-bit signed `int16_t` (2 bytes) in the current implementation. Variables are numbered as follows:

- `var0` – output/return array, also accessible with special purpose instructions
- `var1,...,varN` – source/input arrays (src1...srcN)
- `varN+1,...,varN+K` – local variables (loc1...locK)
- local static memory is accessible by special purpose instructions, starting with keyword `MEM`. Memory is of fixed size and initialized to 0 at concept creation.

The instruction set consists of around 50 codes, some with parameters, and resembles those of the early 1980-ties microprocessors:

- `MOV A,IDX; MOV IDX,A; ADD A,IDX; SUB A,IDX; CMP A,IDX;` the first argument is destination too, excluding CMP. Three latter instructions set flags. Same rules apply to other instructions, respectively.
- `XCHG A,IDX`; exchanges the contents of the registers.
- `MOVI A,int; ADDI A,int; SUBI A,int; CMPI A,int;` the operation is performed on ACC and the 1st parameter `int` inline encoded.
- `MOVI IDX,int; ADDI IDX,int; SUBI IDX,int; CMPI IDX,int;` same as above, but with the IDX register as argument and destination.
- `MOVX A,varN[idx]; ADDX A,varN[idx]; SUBX A,varN[idx]; CMPX A,varN[idx]`; these instructions may signal an error of an attempt to access data out of range. N is an `int` encoded as 1st parameter.
- `MOV A,varN[int]; ADD A,varN[int]; SUB A,varN[int]; CMP A,varN[int]`; 1st parameter is N, 2nd is index, both `int`.
- `APPEND,A; SAVI [int],A; SAVX [idx],A; SAV [int],int;` when runtime is launched, the size of output (var0) is 0, and the maximum size is limited by a concept predefined parameter *outmax*. APPEND appends the value of ACC to the output vector and increases its size by 1, unless in case of exceeding outmax, when error is reported instead. SAVI outputs ACC at position [int], the 1st parameter. If `int` is greater than the current size, the missing cells are padded with 0's. The last instruction saves the value of the 2nd parameter rather than ACC.
- `ADDSAVI [int],A; ADDSAVX [idx],A; ADDSAV [int],int;` the value of the indexed output cell is added to ACC/`int`, and placed in the cell. In case of an attempt to access a cell beyond the current size of the output, a fatal error is reported. Flags are set, accordingly.
- `MEMMOVI A,[int]; MEMMOVX A,[idx]; MEMSAVI [00],A; MEMSAVX [idx],A; MEMSAV [int],int`; the local static memory is accessed. The same memory is shared by all runtimes of a given concept, and the conflicts – especially the overwrites – are not resolved.
- `INC A; DEC A; INC IDX; DEC IDX`; add/sub 1 and set flags.
- `NEG A`; ACC = –ACC.

- `DELAY A,varN`; set ACC to the difference in milliseconds between the real times of creation of the executed runtime and the runtime pointed to by variable N. Applies to input/source variables only.
- `WAIT A`; suspend execution for ACC milliseconds, go to SLEEP state.
- `SIZE A,varN; SIZE IDX,varN`; load the register with the size of variable numbered by the 1st parameter.
- `RAND A;` set ACC with a random value (non-uniform distribution).
- `FLAGS A; FLAGS IDX; FLAGS varN[idx]; FLAGS varN[int];` set flags for the register or the cell.
- `MOVB varN,varM; MOVIB varN,vector; ADDB varN,varM; SUBB varN,varM; CMPB varN,varM;` these are block instructions that operate on the whole variables rather than single cells. Writes to source variables are disabled. Conflicts of different size vectors are resolved, arithmetic operations set flags, *vector* is an inline encoded constant, with the first `int` cell containing the size, followed by a list of `int`s, e.g.: `MOVIB var0,(02):0030,0020`
- `RET; EXIT;` return to the calling process with no fatal error; EXIT is a type of return that additionally resets the *resources* field.
- `CALL; LCALL; LRET;` not utilized below, subject of join by INTEGRATE.
- `JMP ln; JZ ln; JNZ ln; JM ln;` The *ln* stands for line number, encoded as the 1st parameter, counted in bytes. First instruction is unconditional, the latter jump if ZERO is set, ZERO is not set, or if MINUS is set, respectively.

The following is a sample program (instruction codes have size of 1 byte):

```
0000 MOVI A,     0004     load ACC with constant "4";
0003 ADD A, var1[01]      add the value of the 2nd cell of 1st input to ACC;
0008 APPEND, A            output ACC, first time var0[0], then var0[1], etc.;
0009 JNZ         0003     if the result of addition is not "0", jump to ADD;
0012 RET                  finish execution;
```

2.2 Sorting out useless code

When the *program generator* (PG) receives a request to make a program, it is merely given the number of sources (inputs). It is the task of PG to decide on the maximum size of the output, size of local memory, number of lines, and the program code. As one might expect, the instructions and their parameters are chosen with a random select. Even if we restrict the maximum number of lines to 7, we get a space of 10^{12} combinations of code themselves, and more than 10^{20} with even highly restricted parameter selection. A solution to that is to impose constraints and apply tricky heuristic rules. It must be emphasized, however, that it is not the role of PG to learn the most rewarding actions in the program space. The PG, on the other hand, should cover the program space uniformly, sorting out only those pieces of code that would be useless in an obvious way or cause a fatal error. Last but not the least, the PG doesn't know much about the data it's going to process.

The current implementation restricts the number of sources to 3. A program with a single source, like number truncating or delaying a signal, is not very interesting at all. On the other hand, concepts processing more than 2 sources may be created by integrating two 2-source concepts. The PG assumes 75% probability for mandatory utilization of each passed source, where utilization means any reference to the source within the generated code. Though the exploitation routine requires that a runtime awaits all sources in the PENDING state, some sources may be treated as merely a binary trigger. The outmax value is set with the probability $P(N) = ({}^{1}/_{2})^{N}$, N>0. This assumption has further consequences. Preferred are programs with short outcomes, and 50% of them return a single `int`. Hence, the inputs are dominated by short arrays, as well, unless the learning would sort them out. Static local memory size is assumed to be equal to outmax. The minimum number of lines is 3, increased by 1 with each mandatory source. A small bias towards longer programs is also added: `rand() mod 3`. Effectively, the following probability distribution is used (Fig. 1).

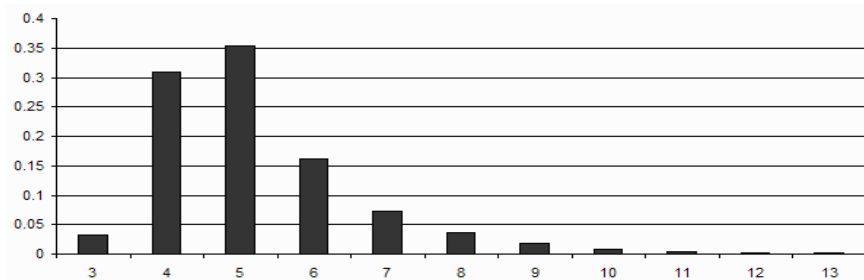

Fig. 1. Program length probability distribution.

As for constants used as parameters, we assume that whatever is interesting happens around zero, with a bias towards positive numbers. Powers of 2 and 10 are preferred. Fig. 2. depicts the probability distribution generated by a designated algorithm.

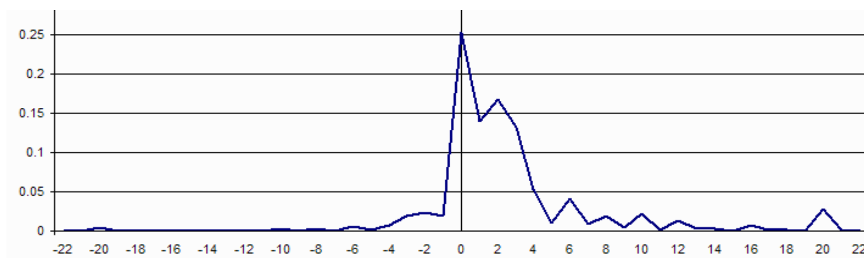

Fig. 2. Constants probability distribution.

Many obvious PG constraints come to mind immediately. These are e.g.: avoiding jumps to itself or jumps not to an instruction code, attempts to access data out of range, if the index and range are known. Below, we would discuss only more challenging heuristics, listing only a few of about 30 that have been applied.

- Only a small subset of the listed above instructions would make sense as the first instruction on program entry (0000). Arithmetic operations, flag settings and jumps are useless, unless some data has been initialized first. Another policy governs the last instruction selection, which may be other than `RET` if `RET` was used before.
- ACC/IDX should not be used, if not initialized first. One would ask, however, how would we know? A register may be initialized further in code, followed by a jump backward. It would involve and unconditional jump, however, that is forbidden in this context. A counterpart rule says that if a register has been set (possibly due a random selection of an instruction), it must be utilized in the following code.
- Flags intentionally set must be utilized before the setting is overwritten by another setting. The `FLAGS` and `CMP` instructions set flags intentionally. In addition, `ADD, SUB, INC` and other instructions set flags 'as a side effect', and that setting may or may not be utilized. An utilization is a conditional jump. If the flag setting was done intentionally, it must not be altered before applied at least once.
- Restricted are jumps: to itself, to the next instruction, to the first instruction, a jump backward that is not followed by setting flags before the jump, a jump forward behind a `RET` instruction that is not to the instruction next after `RET`, unless other forward jump has already used that label. If we consider that a typical program contains less than 8 lines, the above rules become quite restrictive. One could possibly depict correct programs that would drop some of the above rules. They would however be merely equivalents of their simpler counterparts.
- The program must output something somewhere, for otherwise it would be a closed world. Similarly, it is useless to perform any operations before `RET`, if not followed by a modification of the output, or at least the local static memory.
- A more challenging question arises, if one asks what happens when the program is branching, or when two flows join? The PG maintains a status word for each branch. Statuses do split and join. Some rules, like utilization of a set register, perform *alternative* on a join, for it is sufficient that either branch utilizes a set register. As for the output setting requirement, the *conjunction* matters.
- In some cases the algorithm gets to a point where no instruction matches the status word, and the whole PG procedure – called another *pass* – must be repeated.

2.3 Hash pooling

We want the PG to cover the program space evenly, but some programs become much too frequent. Keeping a record of all generated programs would be intractable. On the other hand, sooner or later, the PG will deliver all permutations for a given program size, and repetitions will be indispensable and desired. To solve this problem, the following hash pooling method is applied.

An array of counters `count[N]` (now N=2^14) initialized to 0 is maintained. Each time a new program is created, an index `h = hash(p) mod N` is computed, where `hash()` is a non-cryptographic hash function; `p` is a program. Let `T` be the total number of programs already delivered. If `count[h]>2*(T/N)`, the PG must be repeated, otherwise both `T` and `count[h]` are incremented, and the created program is delivered. As a result, no programs are more frequent than twice the average.

3 Conclusions

The untouched results of the PG algorithm may be downloaded from here:

```
http://aginao.com/pub/pg1000samples.zip
http://aginao.com/pub/pg100000samples.zip
```

Many programs seem to be of limited applicability. Some loop infinitely, run out of memory range, contain useless pieces of code or code that could be rewritten more effectively. The results seem encouraging, however, and – compared to a naive random search – reduce the program space by several orders of magnitude. Not all defective samples would be useless, however. Let's take the example:

```
0000 MOVI A,    0025
0003 CMP A, var1[00]
0008 JZ         0003
0011 APPEND, A
0012 RET
```

The program loops infinitely in case the var1[00] equals 25 – the case that in general is quite unlikely. It would run out of resources in a small percentage of cases, much too few to discard the program as useless. Otherwise, it would terminate and activate the linked concepts. Overall, it would behave like a conditional jump.

On a 12-core Intel i7 CPU 3.33 GHz this algorithm generates $10^{\wedge 6}$ programs/sec. Possibly, no more than 1% of the cognitive engine processor's time would be devoted to program generation. What remains is 10.000 programs/sec that would be evaluated by running them on the VM and sorting out further by the machine learning methods. Further improvements of the PG algorithm are possible. A change of the VM design and the instruction set is quite likely, as well.

References

1. http://www.aldebaran-robotics.com
2. Hawkins, J., George, D.: Hierarchical temporal memory: concepts, theory, and terminology. http://www.numenta.com/htm-overview/education/Numenta_HTM_Concepts.pdf, (2006).
3. Minsky, M.: The Society of Mind, Simon & Schuster, First Touchstone Edition, NY (1988).
4. Kieras, D.E.: EPIC Architecture Principles of Operation. ftp://www.eecs.umich.edu/people/kieras/EPICtutorial/EPICPrinOp.pdf (2004)
5. Franklin, S., Petterson, F.G.Jr.: The LIDA Architecture: Adding New Modes of Learning to an Intelligent, Autonomous, Software Agent. Int. Design and Process Technology (2006).
6. Kokinov, B.N., The Context-Sensitive Cognitive Architecture DUAL. Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society. Hillsdale, NJ (1994).
7. Kieras, D.E.: The Persistent Visual Store as the Locus of Fixation Memory in Visual Search Tasks, ICCM 2009, Manchester, UK (2009).
8. Hofstadter, D.R.: Fluid Concepts and Creative Analogies. Basic Books, NY (1996).