Computation and Intelligence in Problem Solving

Pei Wang
Department of Computer and Information Sciences
Temple University
http://www.cis.temple.edu/∼pwang/
pei.wang@temple.edu

Abstract

The concept of *computation*, as well as the related concepts *algorithm*, *Turing machine*, *computability*, and *computational complexity*, correspond to a specific mode of using computers to solve problems. This *computational mode* assumes the sufficiency of knowledge and resources with respect to the problem to be solved. From the view point of Artificial Intelligence, an *intelligent mode* of problem solving is introduced, where the problem solving process cannot been seen as computation anymore. A system working in this mode is briefly described. Finally, these two modes of problem solving are compared.

"Computation" is a fundamental concept in computer science. Artificial Intelligence (AI) is usually taken as a branch of computer science, and it has also, to a large extent, inherited the theoretical heritage associated to the concept of computation. In this paper we argue that it is improper to treat intelligent problem solving as *computation*, though this conclusion does not make it impossible to build intelligent computer systems. Actually it is the contrary: we believe that real AI can be realized only when we go beyond the concept of computation. An intelligent reasoning system, NARS, is introduced, whose problem solving activities cannot be referred to as computation, though the system is still implemented in an ordinary computer.

1 Problem Solving as Computation

Turing machine is a mathematical model of computer. A Turing machine M has a finite number of states, and among them there is one initial state and at least one final state. At each state q_i, M moves into another state q_j, according to the given input data. A *computation* is a finite sequence of moves by which M transforms from its initial state q_0 to one of its final states q_f, in response to the input data d_i. In the final state, M provides the output data d_o as the result of the computation. We can equally well say that M is a *function* that maps d_i to d_o, or that M is an *algorithm* with d_i and d_o as input and output, respectively [Hopcroft and Ullman, 1979].

Usually, an algorithm is defined on a problem *class*, which has more than one concrete *instances*. Each time the algorithm is applied on a problem instance. A problem (class) is *computable* if there is an algorithm that generates correct result in finite steps for each instance of the class. In that case, the algorithm (or the corresponding Turing machine) is referred to as the solution of the problem.

The amount of time or space used by an algorithm is called the (time or space) *complexity* of the algorithm, and is represented as a function of the "size" of the problem instance. According to the category the function belongs, we call the complexity to be *constant*, *logarithmical*, *polynomial*, *exponential*, and so on. In particular, a problem is *feasible*, or *tractable*, if it has a polynomial solution or better. The problems without polynomial algorithms are *intractable*, because the cost of a solution will become astronomical figures for large instances of the problem.

According to computability theory and computational complexity theory, using a computer to solve a problem usually follows a certain procedure:

1. To define the problem by accurately specifying the valid inputs, and for each of them, specifying the required output.

2. To design an algorithm that correctly generates output for each valid input.

3. To analyze the complexity of the algorithm, and to select the most efficient one if there are multiple algorithms for the problem.

4. To code the algorithm in a programming language, and to save the executable code in a computer system.

If the algorithm is correct and the program is efficient enough, the problem is considered as solved — for each instance of the problem, the program will produce an answer using constant time and space. The output and its (time-space) cost are determined only by the algorithm and the input. In this paper, we call the above approach "the computational mode of problem solving".

Conceptually, a computer system can be seen as a collection of algorithms, and it works by repeating the following cycle:

> to wait for the user to input a new problem instance;
> to call the corresponding algorithm when an input comes;
> to execute the algorithm on given input;
> to report the output;
> to reset the working environment.

Though modern computer systems allow multiple problem instances to be processed in parallel by time-sharing, the above conceptual picture remains unchanged. When an algorithm is working, whether there are other algorithms running should make no difference in the result, unless it has communication with other algorithms (which should be taken as part of the input). If the same problem instance is asked again later, the result should be exactly the same.

Roughly speaking, Artificial Intelligence (AI) is the attempt to build computer systems that work like human mind. Since AI grew as a branch of computer science, most people naturally use computer in the computational mode, by defining problem, designing algorithm, analyzing computational complexity, and so on. This approach has three possible results:

No algorithm is found. If we do not have the knowledge to design an algorithm to solve a given problem, then of course there is no solution, at least at current moment. For certain problems, it can be even proven that they are not computable. Instead of giving up, the usual way to deal with this situation is to change the problem into a similar but simpler problem, for which an algorithm can be designed.

No tractable algorithm is found. If we know an algorithm, but it is too time-consuming, then it cannot be practically used except on very simple instances of the problem. When a solution does not "scale up", in AI we usually does not count it as a solution. For example, in principle many problems (such as playing chess) can be solved by exhaustive search, but we cannot afford the time it requires. That is why some authors treat tractability as a central issue in AI — exponential algorithms easily produce "combinatorial explosion", so they make no practical sense [Bylander, 1991, Levesque, 1989]. Again, whenever this is the case, the common practise is to relax the requirement in the problem (such as replacing "best answer" by "satisfying answer"), so that tractable approximate algorithms and heuristic algorithms can be used.

A tractable algorithm is found. By definition, in this situation the problem has a known solution that can be practically used. For computer scientists, this is the end of the story (unless there is the need to improve the algorithm). However, for the purpose of AI, some people are unhappy — "Where is the intelligence? This is just programming!" The concept of "intelligence" is intuitively related to creativity and flexibility, so to many people, solving a problem by accurately following a predetermined algorithm cannot be it.

So AI is in a weird situation — if a problem is solved, then it seems that the problem does not really need intelligence. Different people have different attitudes toward this situation. Some people take it as an argument for the impossibility of real AI; some people blame the von Neuman computer, and believe AI needs a fundamentally different hardware which is not a Turing machine; some people do not take this as an issue — they proudly see AI as the expending frontier of computer science.

For several reasons, this is an issue. We do not need to run psychological experiments to know that the problem-solving processes in human mind rarely follow predetermined algorithm. On the other hand, one important motivation behind AI research is to introduce flexibility, autonomy, and creativity into computer systems. If AI still follow the common practise of computer program development, then "intelligent" is simply a fancy label on old stuff, and the systems developed will continue to be "brittle", in the sense that it cannot handle any event that is not fully expected when the system is designed [Holland, 1986].

Of course, if the current computer hardware has to be used in computational mode, then the discussion on this issue is fruitless, because there is no other possibility. In the following, we will show that it is not the case. We will introduce an alternative approach, which allows the computer to be used in a fundamentally different mode.

2 Problem Solving in NARS

In this section we introduce NARS (Non-Axiomatic Reasoning System), an intelligent reasoning system. The system's problem solving processes does not follow predetermined algorithm and cannot be seen as computation, though the system is still implemented by ordinary computer software and hardware.

2.1 System overview

NARS is a general-purpose reasoning system. It communicates with its user in a formal language, in which the user gives the system knowledge and questions, and the system answers the questions according to available knowledge.

What distinguishes NARS from other reasoning systems is its ability to *adapt under insufficient knowledge and resources.*

To *adapt* means that the system learns from its experiences. It answers questions and adjusts its internal structure to improve its resource efficiency, under the assumption that future situations will be similar to past situations.

Insufficient knowledge and resources means that the system works under the following restrictions:

Finite: The system has a constant information-processing capacity.

Real-time: All tasks have time requirements attached.

Open: No constraints are put on the knowledge and questions that the system may get, as long as they are expressible in the formal language.

These restrictions have the following concrete implications in NARS:

- Since new knowledge may (explicitly or implicitly) conflict with previous knowledge, the knowledge base may contain conflicting beliefs.
- Since questions may go beyond the scope of available knowledge, the system has to make plausible inference, and to revise its conclusions when they contradict new evidence.
- No knowledge is absolutely certain. A conclusion may have known counter evidence, and it is always possible to get counter evidence in the future.
- Since all questions requires early answers, and new questions may show up when the system is working on other questions, the system usually cannot afford the time to answer a question according to all relevant knowledge.
- Since new knowledge and derived knowledge come constantly, the system's memory (which has a constant capacity) cannot keep all of them.

Consequently, the major components of the system (its knowledge representation language, semantic theory, inference rules, memory structure, control strategy, and user interface) are all fundamentally different from conventional reasoning systems that work in the computational mode.

In NARS, both knowledge and questions are represented as sentences of a formal language, and for each piece of knowledge, a numerical truth value is attached to measure the evidential support its gets according to the experience of the system. Considering future evidence, no empirical knowledge can reach the maximum value of truth, though can approach it with the accumulation of supportive evidence. A question-answering process of the system consists of a sequence of inference steps. In each inference step, an inference rule is applied to some existing knowledge to derive a conclusion, whose truth value is calculated according to the evidence provided by the premises. The control mechanism of the system decides, for each inference step, what premises to use and which rule to apply.

In this paper we will only address the aspect of the system that is directly relevant to the relationship between intelligence and computation. For the other aspects of NARS, see [Wang, 1994, Wang, 1995, Wang, 2001a], or visit the author's website for NARS related publications and an on-line demonstration.

2.2 Controlled concurrency

In the current version, NARS accepts two types of "tasks" (i.e., problems to be solved) from its user: new knowledge to be absorbed and new questions to be answered. For both, the system lets them interact with as much available knowledge as possible, so as to get more derived knowledge (for new knowledge) or more reliable answers (for questions).

What should a system do if it is occupied by one task when another one shows up? Since for NARS new tasks do not come from a predetermined set, usually the system cannot tell at the beginning what kind of solution can be found, or how much resources it will cost. In such a situation, it is usually undesired either to let the new task wait for a unlimited period, or to let the new task interrupt the processing of the current task for a unlimited period.

NARS' goal is not to obtain solutions of a predetermined quality, but to work as efficiently as possible when resources are in short supply. For this reason, NARS *distributes* its resources among the tasks. At a certain moment the time resource given to a task is not determined by an absolute deadline, but by a relative "share", which depends both on the request from the user and on the internal situation of the system.

In NARS, a "priority" measurement is defined on tasks. The priority value of a task is a real number in (0, 1]. At any given instant, if the priority of task t_1 is u_1 and the priority of task t_2 is u_2, then the amounts of time resources the two tasks will receive in the near future keep the ratio $u_1 : u_2$.

Priority is therefore a relative rather than an absolute quantity. Knowing that $u_1 = 0.4$ tells us nothing about when task t_1 will be finished or how much time the system will spend on it. If t_1 is the only task in the system, it will get all of the processing time. If there is another task t_2 with $u_2 = 0.8$, the system will spend twice as much time on t_2 as on t_1.

Intuitively, we can envision NARS as having a task pool, in which there is an priority value attached to each task. The system processes the tasks in a time-sharing manner, meaning that the processor time is cut into fixed-size time-slices, and in each slice a single task is processed. Because NARS is a reasoning system, its processing of a task divides naturally into inference steps, one per time-slice. The system distributes its time-slices among the tasks, giving each task a number of time-slices proportional to its priority value. To implement such an asynchronous parallelism, a task is chosen probabilistically for each step, and the probability for a task to be chosen is proportional to its priority. As a result, priority determines the (expected) processing speed of a task.

If the priority values of all tasks remain constant, then a task that arises later will get less time than a task that arises earlier, even if the two have the same priority value. A natural solution to this problem is to introduce an "aging" factor for the priority of tasks, so that all priority values gradually *decay*. In NARS, a real number in (0, 1), called *durability*, is attached to each priority value. If at a given moment a task has priority value u and durability factor d, then after a certain amount of time has passed, the priority of the task will be ud.

Therefore, durability is a relative measurement, too. If at a certain moment $d_1 = 0.4$, $d_2 = 0.8$, and $u_1 = u_2 = 1$, we know that at this moment the two tasks will get the same amount of time resources, but when u_1 has decreased to 0.4, u_2 will only have decreased to 0.8, so the latter will then be receiving twice as much processing time as the former.

If the durability value of a task remain constant, the corresponding priority will become the following function of time:

$$u = u_0 d_0^{ct}$$

where u_0 and d_0 are the initial values of priority and durability at instant

0, respectively, and c is a constant. Taking the integration of the function, we get the *expected relative time-cost* of a task (at instant 0):

$$T = \int_0^\infty u dt = -\frac{u_0}{\ln d_0}$$

As a relative measurement, the constant is omitted in the result.

By assigning different priority and durability values to tasks, the user can put various types of time pressure on the system. For example, we can inform the system that some tasks need to be processed immediately but that they have little long-term importance (by giving them high priority values and low durability values), and that some other tasks are not urgent, but should be processed for a longer time (by giving them low priority and high durability values).

If a task is a question asked by the user, when to report an answer? In the computational mode, an answer gets reported only at the final state, where the system has completed its processing of the question. However, when time is treated as a limited resource and no answer is final, it is better to let the system provide some sort of answer as soon as possible. NARS keeps a record of the best answer it has found for each question, and whenever a new candidate is found, it is compared with the current best. If the new one is better, it is reported to the user, and the record is updated. Usually, the question remains active, but with a lower priority, so the system will continue to spend time on it to find better answers. In this way, the amount of time spent on a task is determined not only by the requirement of the user, but also by the result(s) the system has got for the task. Each time a task is processed (i.e., after each inference step), the system reevaluates the task's priority and durability values, to reflect the current situation. As a result, NARS maintains a *dynamical* processor-time distribution in its task pool.

It is possible for the system to report more than one answer for a question — it can "change its mind" when new knowledge is taken into consideration. On the other hand, it is also possible for a question to be removed from the task pool before even a single answer is found for it. When the task pool is exceeded (it has a constant capacity), tasks at the low end of the priority spectrum are removed. When a task is removed from the task pool, it is not because the processing of the task has met some predetermined goal, but because the task has lost too much ground in the competition for resources.

NARS constantly generates derived tasks (subquestions and implied knowledge) with its inference rules. Each such task is assigned priority and durability values by the system (according to the type of the inference and the priority and durability of its parents), and then put into the task pool. After that, it is treated just like a task provided by the user. Even if a "parent" task has been removed (by losing out in the competition for resources), "children" tasks derived from it may still be processed, provided that they have sufficiently high priority values.

For example, when answering a question Q, NARS may generate two subquestions Q_1 and Q_2. Later, it finds an answer to Q_1, which leads to an answer to Q. At this point, the priority values of Q and Q_1 are decreased more rapidly than that of Q_2, and it is possible for Q_2 to be processed even after Q has been removed from the system's task pool. If the purpose of a system were solely to answer questions coming from the user, the above strategy would seem pointless, because Q_2 is merely a means to solve Q, hence should go away if Q goes away. However, the purpose of NARS is to adapt to its environment, which means that Q_2, as a derived question, has value for its own sake, even in a situation where the question that engendered it has utterly vanished. The system will benefit from the processing of Q_2 when similar questions appear thereafter.

As a result, after running for a while, there will be tasks in the system that are only remotely related to the tasks provided by the user.

2.3 Bag-based memory organization

In each inference step the system chooses a task, according to the priority distribution, then interacts it with a piece of knowledge, to get results. However, how is the knowledge chosen? Here the problem is very similar to the problem discussed previously. With insufficient resources, the system cannot consult all relevant knowledge for a task. On the other hand, knowledge cannot be used indiscriminately — some knowledge is more important and useful than the other.

To solve this problem, the above idea of "controlled concurrency" is generalized. Let us say that a system has some items to process in a certain way. Because new items may arrive at any time, and because the time requirements of the items would exceed the system's capacity, it is impossible for the system to do the processing exhaustively. It has to distribute its time resources among the items, and to truncate the processing of an item before reaching its "final conclusion". Furthermore, items are not treated equally. The system evaluates the relative priority of each item as a function of several factors, and adjusts its evaluation when the situation changes. In addition, the system's storage capacity, which is a constant, is also in short supply.

Because this phenomenon pervades the discussion of systems with insufficient resources, it will be useful to design a special data structure for it. In NARS, we have such a data structure called "bag". A *bag* is a kind of *probabilistic priority queue* that can contain a constant number of *items*. Each item has a *priority value*, which is a positive real number. There are two major operations defined on a bag: *put-in* and *take-out*. The operation *put-in* takes an item as argument, and has no return value. Its function is to put the item into the bag. If the bag is already full, the item with the lowest priority is first removed from it, and then the new item takes its place in the bag. The operation *take-out* has no argument, and returns one item when the bag is not empty. The probability for a given item to be chosen is proportional to its priority.

Now we can describe the memory structure of NARS in terms of bag. NARS implements a *term-oriented logic* [Wang, 1994]. This kind of logic is characterized by the use of subject–predicate sentences and syllogistic inference rules, as exemplified by the Syllogism of Aristotle. A property of term logic is that every inference rule requires its (usually two) premises to share a term. This nice property of term logic naturally localizes the choosing range of knowledge. For a task with the form "Dove is a kind of bird", we know that the knowledge that can directly interact with it must has "dove" or "bird" in it as subject or predicate. If we put all tasks and knowledge that share a common term together, call it a *concept*, and name it by the shared term, then any valid inference step will necessarily happen within a single concept.

Defined in this way, a concept becomes a unit for resource allocation, which is larger than a task or a piece of knowledge. The body of the concept contains the relations between the term (that names the concept) and other terms. The memory of the system is simply a set of concepts. The action of choosing a task can be recast as a two-step process: first choosing a concept, and then from it choosing a task. In other words, the system distributes its resources firstly among the concepts, and then secondly, within each concept, among the tasks. The result is a two-level structure. On both levels, the notion of "bag" applies. Specifically, we can describe the memory of NARS as a bag of concepts, with, within each concept, a bag of tasks and a bag of knowledge.

Now we can see the distinction between *tasks* and *pieces of knowledge* more clearly. All questions are tasks. New knowledge also serves as tasks for a short time. If a piece of knowledge provides an answer for a question, it will be treated as a task for a short time. Because of this distinction, the system has, at any given moment: (1) a small number of tasks, which are active, remembered for a short time, and highly relevant to the current situation; and (2) a much larger amount of knowledge, which is passive, remembered for a long time, and mostly not relevant to the current situation.

It follows from the assumption of insufficient resources that in NARS the results are usually only derived from part of the system's knowledge, and which part of the knowledge base is used depends on the context at the run time. Consequently, NARS is no longer "logical omniscient" [Fagin and Halpern, 1988] — it cannot recall every piece of knowledge in its knowledge base, not to mention being aware of all their implications.

From the previous description of the memory organization, we can see that two types of "forgetting" happen in NARS. The first type, "relative forgetting", is caused by the insufficiency of *time* resource — items (concept, task, or knowledge) with low priority are seldom accessed by the system, though they are there all the time. The second type, "absolute forgetting", is caused by the insufficiency of *space* resource — items with the lowest priority is removed from overloaded bags.

2.4 Inference process

When the system is running, an "execution cycle", or "inference step", is repeated until the process is interrupted by the user. The cycle consists of the following operation sequence:

1. To check input buffer. If there are new tasks, put them into the corresponding concepts.

2. To take out a concept from the concept bag.

3. To take out a task from the task bag of the concept.

4. To take out a piece of knowledge from the knowledge bag of the concept.

5. To apply inference rules on the task and knowledge. Which rule is applicable is determined by the combination of the task and the knowledge.

6. To adjust the priority and durability values of the given task, knowledge, and concept, according to the quality of the results. Then the task, knowledge, and concept are returned to the corresponding bags.

7. To put the results generated in this step into the input buffer as new tasks. If a result happen to be a best-so-far answer of a question asked by the user, it is reported to the user.

The priority value of each item reflects the amount of resources the system plans to spend on it in the near future. It is determined by two factors:

long-term factor: The system gives higher priority to more *important* items, evaluated according to past experience. Initially, the user can assign priority values to the input tasks to indicate their relative importance, which will in turn determine the priority value of the concepts and knowledge generated from it. After each inference step, the involved items have their priority values adjusted. For example, if a piece of knowledge provides a best-so-far solution for a task, then the priority value of the knowledge is increased (so that it will be used more often in the future), and the priority value of the task is decreased (so that less time will be used on it in the future).

short-term factor: The system gives higher priority to more *relevant* items, evaluated according to current context. When a new task is added into the system, the directly related concepts are *activated*, i.e., their priority values are increased. On the other hand, the priority values decay over time, so that if a concept has not be relevant for a while, it becomes less active.

To explain how the priority and durability values are actually calculated, it is inevitable to involve many technical details of the NARS model, and thus is beyond the scope of this paper.

It is possible to implement the above inference step in such a way that it takes roughly constant time, no matter how large the involved bags are [Wang, 1995]. Such a step is like an "atomic operation" of the problem-solving processes in NARS. However, unlike in computational mode, where operations are organized into algorithms *in advance*, in NARS the operations are linked together for a given problem *at run time*, and how they are linked is context-dependent.

Built in this way, NARS shows many novel properties:

- Knowledge is accepted by the system in the form of declarative sentences in a formal language. The user can ask the system any question that can be phrased in the formal language, and the system will not be paralyzed by questions beyond its current capacity. Neither the designer nor the user needs to provide the system with problem-specific algorithms.

- The user can assign initial priority and durability value to a task to influence (though not to determine) the system's resource allocation to that task.
- The system may provide a quick answer to a question, then refine the answer incrementally. In this sense, NARS can "change its mind" when new knowledge is taken into consideration.
- The system usually concentrates on the most important and promising tasks, but it also pays some attention to other "peripheral" tasks.
- The resource distribution changes dynamically, according to the result of each inference step. Because certain approximation is used in the implementation of the control mechanism, the overhead of task scheduling is low, and each inference step only takes a small constant amount of time.
- The self-adjustment of knowledge structure is also guided by the feedback of each inference step. Knowledge useful in the past will get a higher probability to be used again in the future.
- The response to a question depends not only on what the system has been told, but also on what the system has been asked. For example, the system may spend a long time to find an answer, but if the same question (or a similar one) appears again later, the answer usually comes sooner.

These properties clearly distinguish NARS from other reasoning systems that work in the computational mode.

3 Discussion and Comparison

3.1 Is this still computation?

An interesting and important question about NARS naturally arises: Is the system still doing *computation*?

To answer this question, we need to first identify the "problems" NARS attempts to solve. From the previous description, it is obvious that each "task" in NARS corresponds to a "problem instance", defined at the beginning of the paper. If a task is a question, then to solve it means to find answer for it; if a task is a piece of new knowledge, to solve it means to reveal its implications. In this sense, what NARS does is not computation. Actually, almost all components in the "computational mode" of problem-solving are missing in NARS:

- Though NARS does solve problems, it processes each problem instance in a case-by-case manner, without a general algorithm for the "problem class" as a whole. As a result, it may give a pretty good solution to a problem (instance), but may fail to solve a similar one.
- In NARS, there is no unique "initial state" in which the system waits for and accepts new tasks. At any moment when the system is running, tasks can be accepted, in many different internal states.
- Similarly, there is no "final state" for a task. For instance, if a task's priority is low (relative to other tasks), it is even possible for it to be completely ignored. If a tentative answer to a question is reported, usually neither the system nor its human designer can predict whether a better answer will be reported later, since that will depend on events still to take place in the future, such as whether the system acquires new knowledge related to the task, or whether more time winds up being spent on it.
- For a given problem, whether a result is a "solution" become a matter of degree. Due to the insufficiency of knowledge and resources, NARS cannot give a "final solution" to any problem.
- As described previously, the inference steps are chained together into an inference process in run time that generates the result(s) reported to the user. There is no predetermined algorithm to follow for a given problem.

By slightly changing the meaning of the term, one might say that NARS has an initial state — namely, when its memory is initialized as the system is just started. Its state changes as soon as it interacts with its environment (i.e., the user) and begins processing tasks. The system never will return to its initial state, until and unless a user terminates the processing and erases all of its memory. In such a case, the system can of course be "reborn" with the same "genetic code" — its sets of inference rules, control mechanisms, and so on. However, unless the experience of the system perfectly repeats its experience in its “previous life”, the system’s behaviors will be different. In this sense, the system’s behaviors are determined by its initial state and its experience, but not by either one of the two alone.

Now we can see that NARS can be observed on (at least) three scales, in term of what is referred to as its input (problem) and output (solution).

- In the scale of each *execution cycle*, or inference step, as defined previously, the system’s activity is computation, where the input is the current memory, and the output is a new memory adjusted by that step. In NARS, there is an explicitly coded algorithm for this process.
- In the scale of each *task-processing cycle*, where the input is a task, and the output is the result of the processing of the task, the system’s activity cannot be captured by concepts like computation, function, or algorithm, as discussed above.
- In the scale of the whole *life cycle* of the system, where the input is its lifelong input stream, and the output is its lifelong output stream, the relation between the two becomes computation again (even though there is no explicitly coded algorithm for it). If we take an arbitrary state of NARS, q_1, as an “initial state”, the state the system arrives at after a certain amount of time, q_2, as a “final state”, then we can view what NARS does during that period of time as computation, with its experience (all of the tasks provided by the user during that time) as the input, and its responses (all of the system-generated reports) as the output.

In summary, the behavior of NARS can be described on different levels. NARS is computing on some, but not all, of them. This state of affairs has been articulated by Hofstadter in the following way: “something can be computational at one level, but not at another level” [Hofstadter, 1985], and by Kugel as “cognitive processes that, although they involve more than computing, can still be modelled on the machines we call ‘computers’ ” [Kugel, 1986].

In contrast to this, conventional computer systems, while also describable at these levels, are computing in all of them. Let us use an ordinary sorting program as an example: you can take either a single sorting problem (the analogue to a single question from the user), or a *sequence* of such problems (the analogue to the unwieldy and long sequence of user inputs in a given period of time), as the input, and the processes in both cases are computation — the program’s response to a given sorting task is fully determined (by the algorithm and the input data) and does not depend on its experience and context (i.e., the processing of other sorting tasks).

Though still follows algorithms at a certain level, NARS is creative and autonomous in the sense that its behavior is determined not only by its initial design, but also by its "personal" experience. It can generate results never anticipated by its designer, and can produce them by its own choice. A "tutor" can "educate" it by manipulating its experience, but cannot completely control its behavior due to the complexity of the system. From a pragmatic point of view, this is neither necessarily a good thing, nor necessarily a bad thing. It is simply the case that an adaptive system with insufficient knowledge and resources has to behave in this way.

Since in NARS, problem-oriented algorithms are not used, the very concept of "computational complexity" disappears on the problem-solving level. If the system is faced with a problem that may take a large amount of time, what is guaranteed is not that the system will arrive at a satisfactory solution, but rather, that the system will not be paralyzed by the problem — the system will gradually decrease the problem's priority, while still leaving it a chance to be solved through future inspirations. This is much like what happens in the human mind: we say there is no "combinatorial explosion" in our minds, not because we can solve all problems in polynomial time, but because we seldom, if ever, stick to exhaustive searching, nor even to working monomaniacally on a single problem facing us.

Also, this approach suggests a new interpretation and solution to the "scaling up" problem. It is well-known that many AI systems work fine at experiment stage with small knowledge sets, but fail to work in real-world situations. This is often caused by certain "sufficient resource" assumption implicitly made in the design, such as with operations exhausting possibilities for a specific purpose. These operations are affordable on small amount of knowledge, but become inapplicable when the knowledge base is huge. For the systems based on the assumption of insufficient resources, such as NARS, the situation is different. These systems do not take the advantage of small knowledge base by exhausting possibilities, and also do not attempt to do so when the knowledge base is huge. Consequently, the resource management mechanisms used by these systems do scaling up. The system's performance still becomes not as good when the problem is hard and the knowledge base is huge, but the degradation happens in a *graceful* way, just like what happens to the human mind in similar situations.

3.2 Related work

The design discussed in this paper is similar to many previous approaches in various aspects.

To let a system make trade-off between solution quality and time cost, this is not a new idea [Good, 1983]. Approximation algorithms and heuristic algorithms are all motivated by this consideration [Rawlins, 1992]. Similarly, we can first decide the time request, usually in the form of a *deadline*, then look for an algorithm which can meet the deadline, and can also provide a solution as good as possible. This approach leads to the concept of *real-time algorithm* [Laffey et al., 1988, Strosnider and Paul, 1994]. However, in these algorithms, trade-off is determined when the algorithms are designed. As a result, the problem-solving process is still computation, only on a relaxed version of the problem. It is still the problem-oriented algorithm that decides which step to take at each instant, and where to stop at the end.

In many situations, it is better to treat time pressure as a variable and context-dependent factor, because the time requests of problems, the desired quality of solutions, and the system's time supply for a problem (in a multi-task environment) may change from context to context. It is inefficient, if not impossible, to equip the system with a family of algorithms for each possible context. For this situation, we hope to take the time pressure into consideration in the *run time*.

One instance of this approach is to use an interruptible algorithm. In the simplest case, a "trial and error" procedure can be used to "solve" a uncomputable problem [Kugel, 1986]. Suppose we want to check whether a Turing machine halts, we can use such a procedure. It reports "NO" at the very beginning, then simulate the given Turing machine. When the Turing machine halts, the trial-and-error procedure reports "YES" and halts. Such a procedure is not an algorithm because it may not stop, but it can be implemented in ordinary computers, and its *last* report is always a correct one, though the user may not have the time to get it, or cannot confirm that it is really the last one when it is "NO".

A more general concept along this path is the concept of "anytime algorithm" [Dean and Boddy, 1988]. The term "anytime algorithm" is currently used to refer to algorithms that provide approximate answers to problems in such a way that: (1) an answer is available at any point in the execution of the algorithm; and (2) the quality of the answer improves with an increase in execution time.

Such an "algorithm" no longer corresponds to a Turing machine. Because there is no predetermined final states, the algorithm is stopped by an external force, rather than by itself. Consequently, the amount of time spent on a problem is completely determined by the user (or a monitor program) at run time, and no result is "final" in the sense that it could not be revised if the system had spent more time on the problem.

In this way, the time pressure on a problem-solving activity is no longer a constant. The user can either attach a time request, such as a deadline, to a problem at the beginning, or let the algorithm run, then interrupt it later. Under different time pressure, the same algorithm may provide different solutions for the same problem.

A more complex situation happens when the idea of anytime algorithm is used at the sub-problem level.

If a task can be divided into many subtasks, and the system does not have the time to process all of them thoroughly, it is often possible to carry out each of them by an anytime algorithm, and to manage the processing time as a resource. According to Good's "Type II rationality" [Good, 1983], in this situation an optimum solution should be based on decision theory, by taking the cost of deliberation and the expected performance of the involved algorithms into account. To do this, the system needs a *meta-level* algorithm, which explicitly allocate processing time to object-level procedures, according to the expected effect of those allocations on the system's performance. This idea is developed in several research projects under the name of "deliberation scheduling" [Boddy and Dean, 1994], "flexible computation" [Horvitz, 1989], and "meta-reasoning" [Russell and Wefald, 1991].

The above approaches stress the advanced planning of resource allocation, therefore depends on the quality of the expectations, though run-time monitoring is also possible [Zilberstein, 1995].

However, if the information about object-level procedures mainly comes at the run time, the meta-level planner may have little to do before the procedures actually run — its expectations will be very different from the reality revealed later. To be efficient, the resource allocation has to be adjusted dynamically when the system is solving object-level problems, and the advanced planning become less important (though still necessary). This is particularly true for adaptive systems.

The rationality of NARS is reflected in its ability to learn from its experience and to adapt to its environments, even though the system provides no guaranty to the absolute answer quality and response time for a certain task. What we can say about it is: if the system spends more time on a task, the quality of the answer, mainly measured by its "confidence" value [Wang, 1994, Wang, 1995, Wang, 2001b], will improve. However, how much time will be actually spent on it and the quality of the answer are determined only at the *end* of the processing, not at the *beginning* of it, as in other flexible computation approaches.

From a technical point of view, what distinguishes this approach from the other flexible computation procedures is the use of asynchronous parallelism, derived from the idea of time-sharing. Even though, this control mechanism is still very different from ordinary time-sharing, because here the tasks work on a common knowledge base, and it is not guaranteed that all tasks will be processed all the way to their final conclusions. Consequently, the interaction among tasks in NARS is much stronger than that among the processes in a conventional time-sharing system. The coexistent tasks not only influence the processing speed of a task (this is also true for ordinary time-sharing systems), but also strongly influence its processing depth (i.e., when the processing terminates) and path (what knowledge is consulted, and in what order).

The "parallel terraced scan" strategy developed by Hofstadter's research group [Hofstadter and the Fluid Analogies Research Group, 1995] provides another example of dynamical resource allocation.

When exploring an unfamiliar territory to achieve a goal under a time pressure, it is usually impossible to try every path to its end. Without sufficient knowledge, it is also impossible to get a satisfactory plan before the adventure. However, if somehow the system can investigate many paths in parallel, and the intermediate results collected at different levels of depth can provide clues for the promise of the paths, the system may be able to get a relatively good result in a short time.

This kind of terraced scan moves by stages: first many possibilities are explored in parallel, but only superficially. Then the system reallocates its time resources according to the preliminary results, and let the promising ones to be explored more deeply. Stage by stage, the system focuses its attention to less and less good paths, which hopefully lead the system to a final solution.

Putting it in another way, we can think the system as exploring all the possible paths at the same time, but at different *speeds*. It goes faster in the more promising paths, and the speeds are adjusted all the time according to the immediate feedback on different paths. The system usually does not treat all paths as equal, because that means to ignore available information about different paths; the system usually also does not devote all its resources to a path that is the most promising one at a certain time, because in that way the potential information about other paths cannot be collected. In between these two extreme decisions, the system distributes its time resource *unevenly* among its tasks, and dynamically adjusts its bias according to new results. A similar approach is taken by genetic algorithm [Holland, 1986].

From the user's point of view, the most distinguished nature of NARS' control mechanism is non-determinism (or context-sensitivity). Even if the user provides the same task to the system, with the same priority and durability values, the task may be processed differently: when the system is busy (that is, there are many other tasks with higher priority), the task is only briefly processed, and some "sallow" implications or answers are found; when the system is idle (that is, there are few other tasks), the task is processed more thoroughly, and deep results can be obtained. Generally speaking, a task can be processed for any number of steps, as in anytime algorithms. The actual number of steps to be carried out is determined both by the initial assignment of priority and durability, and by the resources competition in the system. Furthermore, the processing procedure and result depend on the other tasks existing currently and recently. Every task changes the knowledge structure while being processed, and therefore influences how other tasks will be processed.

For example, if the system just processed a task A, and then begins to work on a related task B, the knowledge that contributes to A's processing will get a higher chance to be used again. In this way, tasks are processed in a context-sensitive manner, rather than by following a predetermined path. Here "context" means the events that happened before the task appears (they shaped the knowledge structure) and the events that happen when the task is being processed (they change the knowledge structure and influence the resource supply).

The above feature distinguishes NARS from other similar approaches that are based on the idea of anytime algorithm or asynchronous parallelism. In NARS the processing of a task becomes unpredictable and irrepeatable (from the design of the system and the task itself), because the context plays a central role. It should be understood that the system is indeterministic in above sense, rather than because it takes out items from bags according to a probabilistic distribution — that is simply a way to allocate resources unevenly, and can be implemented deterministically.

We can see the priority distribution adjustment in NARS as a learning process, by which the system learns, from its own experience, about that knowledge is more useful and relevant. This kind of "structural knowledge" is not declaratively expressed by the system's knowledge representation language, but embedded in its knowledge structure. In NARS, this kind of learning, like the learning of declarative knowledge, is a life-long process [Thrun and Mitchell, 1995] that determined partially by the experience of the system. On the contrary, in the current study of "machine learning", most approaches are still algorithms that generate certain output from certain input, therefore much less flexible.

NARS uses a *forgetting* mechanism to manage its memory. Though many systems release memory when running, usually it is done when the data there is no longer useful. The challenge to the forgetting mechanism is NARS is to decide what to ignore or delete, even when it may be useful in the future.

This reminds us of human memory. On one hand, we all suffer from forgetting information that become needed later; but on the other hand, it is not hard to image what a headache it would be if every piece of knowledge was equally accessible — that is, equally inaccessible. Like it or not, properties such as forgetting are *inevitable consequences* of the insufficiency of resources. This theme will appear from time to time in the discussion about NARS — though many of its properties are usually judged as unwelcome in AI systems, they become inevitable as soon as we want a system to work under insufficient knowledge and resources. In this sense, we say that many mistakes made by the system are *rational*, given its working environment. The only way to inhibit these mistakes is to limit the problems that the systems are exposed to, like in most computer systems. However, in this way computer systems loss the potential to conquer real hard problems, because the problems we call "hard" in everyday life are precisely the problems for them our knowledge and resources are insufficient.

4 Conclusions

By describing the problem solving process in (the current version of) NARS, we are not claiming that it is how intelligence works (NARS is going to be further extended in several major aspects). Instead, it is used as an example to propose a new mode of problem solving. Generally speaking, this working mode has the following properties:

- It does not define a "problem" as a set and use the same method to solve all of its instances. Instead, it treats each "problem instance" as a problem by its own, and solve it in a case-by-case manner.

- For a problem, it does not draw a sharp line between solutions and non-solutions, and treat all solutions as equal. Instead, it compares candidate solutions to decide which one is better.

- It does not insist on the "one problem, one solution" format. Instead, it allows the system to generate zero, one, or a sequence of solutions, each of which is better than a previous one.
- It does not depend on a predetermined algorithm to solve a problem. Instead, it cuts a problem-solving process into steps. Each step may still follow an algorithm which takes constant time to finish, but the who process is linked together at run time.
- It processes tasks (and subtasks) in parallel, but at different speed, according to their priority values.
- It does not attempt to use all relevant knowledge to solve a problem. Instead, in each step it only considers a constant amount of knowledge, selected according to their priority values.
- In each step, it lets the selected task and knowledge to decide how the task is processed.
- It does not throw away the intermediate results at the end of a problem-solving process. Instead, it keeps them for future tasks, and let all tasks to interact with the same knowledge structure.
- When memory is full, it removes items with the lowest priority.
- It adjusts the priority distributions according to the experience of the system and the current context, so as to give important and relevant items more resources.

We call this working mode the "intelligent mode" of problem solving, because we the believe that "intelligence" is basically the capacity of adaptation with insufficient knowledge and resources [Wang, 1995]. Though it can be implemented in existing computer systems, and its steps and the whole system history can be seen as computation, this mode is fundamentally different from the computational mode. The traditional theories on computation cannot be directly applied to this kind of problem solving anymore. New theories are needed where the assumption of insufficient knowledge and resources are made from the very beginning.

This approach is not just a picturesque new way to see things, but has important methodological implications for AI research. When a system like NARS is designed, the designer should not try to decide what answer the system should produce in response to a given question — that should be decided by the system itself at run time; the designer simply cannot exhaustively consider all possible situations in advance (the designer, hopefully, is also an intelligent system, thus limited by insufficient resources). For similar reasons, the designer cannot decide in advance how much of the resources to spend on a certain task, for this is totally context-dependent. Thus, the designer is no longer working on either domain-specific algorithms or general-purpose algorithms (like GPS), but rather on *meta-algorithms* or *micro-algorithms*, which carry out inferences, manage resources (like a small operating system), and so on. In this way, the problems solved by the designer and the problems solved by the system itself are clearly distinguishable from one another.

These ideas allow us to explain why *Tesler's Theorem* — "AI is whatever hasn't been done yet" [Hofstadter, 1979] — applies to many AI projects: in those projects, the designers usually use their own intelligence to solve domain problems, and then implement the solutions in computer systems in the form of task-specific algorithms. The computer systems then execute the algorithms on specific instances of the problems, an activity that can hardly be referred to as "solving problems intelligently".

This is also related to the "mind over machine" argument proposed by many authors — namely, since human mind does not follow algorithms in many problem-solving activities but computers cannot run without algorithms, thinking is not computation, and "Strong AI" is impossible. All these arguments have the same problem: when claiming (correctly) that thinking usually does not follow algorithm, the brain's activities are described on the "problem-solving" level (rather than on the "neuron firing" or "lifelong history" level, where the claim is hard to justify). On the other hand, it is usually implicitly assumed that all activities of a computer follow algorithms, no matter on which level or scale of description.

The intelligent mode is not always better than the computational mode. Their relation is similar to Hofstadter's distinction between "Intelligent mode" and "Mechanical mode" [Hofstadter, 1979]. For a given problem, if we have sufficient knowledge (so we can write an algorithm to solve it) and resources (so we can afford the time-space resources required by the algorithm), it is still better to use computation, because of its correctness, efficiency, and stability. Only when we have to deal with problems for which the system's knowledge and resources are insufficient, intelligence becomes the answer, because of its creativity, flexibility, and autonomy.

References

[Boddy and Dean, 1994] Boddy, M. and Dean, T. (1994). Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence*, 67:245–285.

[Bylander, 1991] Bylander, T. (1991). Tractability and artificial intelligence. *Journal of Experimental & Theoretical Artificial Intelligence*, 3:171–178.

[Dean and Boddy, 1988] Dean, T. and Boddy, M. (1988). An analysis of time-dependent planning. In *Proceedings of AAAI-88*, pages 49–54.

[Fagin and Halpern, 1988] Fagin, R. and Halpern, J. (1988). Belief, awareness, and limited reasoning. *Artificial Intelligence*, 34:39–76.

[Good, 1983] Good, I. (1983). *Good Thinking: The Foundations of Probability and Its Applications.* University of Minnesota Press, Minneapolis.

[Hofstadter, 1979] Hofstadter, D. (1979). *Gödel, Escher, Bach: an Eternal Golden Braid.* Basic Books, New York.

[Hofstadter, 1985] Hofstadter, D. (1985). Waking up from the Boolean dream, or, subcognition as computation. In *Metamagical Themas: Questing for the Essence of Mind and Pattern*, chapter 26. Basic Books, New York.

[Hofstadter and the Fluid Analogies Research Group, 1995] Hofstadter, D. and the Fluid Analogies Research Group (1995). *Fluid Concepts and Creative Analogies: Computer Models of the Fundamental Mechanisms of Thought.* Basic Books, New Nork.

[Holland, 1986] Holland, J. (1986). Escaping brittleness: the possibilities of general purpose learning algorithms applied to parallel rule-based systems. In Michalski, R., Carbonell, J., and Mitchell, T., editors, *Machine Learning: an artificial intelligence approach*, volume II, chapter 20, pages 593–624. Morgan Kaufmann, Los Altos, California.

[Hopcroft and Ullman, 1979] Hopcroft, J. and Ullman, J. (1979). *Introduction to Automata Theory, Language, and Computation.* Addison-Wesley, Reading, Massachusetts.

[Horvitz, 1989] Horvitz, E. (1989). Reasoning about beliefs and actions under computational resource constraints. In Kanal, L., Levitt, T., and

Lemmer, J., editors, *Uncertainty in Artificial Intelligence 3*, pages 301–324. North-Holland, Amsterdam.

[Kugel, 1986] Kugel, P. (1986). Thinking may be more than computing. *Cognition*, 22:137–198.

[Laffey et al., 1988] Laffey, T., Cox, P., Schmidt, J., Kao, S., and Read, J. (1988). Real-time knowledge based system. *AI Magazine*, 9:27–45.

[Levesque, 1989] Levesque, H. (1989). Logic and the complexity of reasoning. In Thomason, R., editor, *Philosophical Logic and Artificial Intelligence*, pages 73–107. Kluwer Academic Publishers, Boston.

[Rawlins, 1992] Rawlins, G. (1992). *Compared to What?* Computer Science Press, New York.

[Russell and Wefald, 1991] Russell, S. and Wefald, E. (1991). Principles of metareasoning. *Artificial Intelligence*, 49:361–395.

[Strosnider and Paul, 1994] Strosnider, J. and Paul, C. (1994). A structured view of real-time problem solving. *AI Magazine*, 15(2):45–66.

[Thrun and Mitchell, 1995] Thrun, S. and Mitchell, T. (1995). Learning one more thing. In *The Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence*, pages 1217–1223.

[Wang, 1994] Wang, P. (1994). From inheritance relation to nonaxiomatic logic. *International Journal of Approximate Reasoning*, 11(4):281–319.

[Wang, 1995] Wang, P. (1995). *Non-Axiomatic Reasoning System: Exploring the Essence of Intelligence*. PhD thesis, Indiana University.

[Wang, 2001a] Wang, P. (2001a). Abduction in non-axiomatic logic. In *Working Notes of the IJCAI workshop on Abductive Reasoning*, pages 56–63, Seattle, Washington.

[Wang, 2001b] Wang, P. (2001b). Confidence as higher-order uncertainty. In *Proceedings of the Second International Symposium on Imprecise Probabilities and Their Applications*, pages 352–361, Ithaca, New York.

[Zilberstein, 1995] Zilberstein, S. (1995). Operational rationality through compliation of anytime algorithm. *AI Magazine*, 16(2):79–80.