Problem-Solving under Insufficient Resources

Pei Wang
Center for Research on Concepts and Cognition
Indiana University
510 North Fess, Bloomington, IN 47408
pwang@cogsci.indiana.edu

September 21, 1995

Abstract

This paper discusses the problem of resources-limited information processing. After a review of relevant approaches in several fields, a new approach, controlled concurrency, is described and analyzed. This method is proposed for adaptive systems working under insufficient knowledge and resources. According to this method, a problem-solving activity consists of a sequence of steps which behaves like an anytime algorithm — it is interruptible, and the quality of the result is improved incrementally. The system carries out many such activities in parallel, distributes its resources among the them in a time-sharing manner, and dynamically adjusts the distribution according to the feedback of each step. The step sequence for a given problem is formed at run time, according to the system's knowledge structure, which is also dynamically formed and adjusted. Finally, this approach is compared with other approaches, and several of its properties and implications are discussed.

In a computer system, all problem-solving activities cost computational resources, mainly processor time and memory space. Because these two types of resource can often be substituted by each other, this paper focuses on the management of the time resource.

The paper begins by a review of the different assumptions about time resource made in various fields for different purposes. Then, a new situation, *problem-solving under insufficient resources*, is defined and discussed. A resource-management mechanism, *controlled concurrency*, is proposed for this situation. After its basic components are introduced, the properties and implications of the mechanism are discussed and compared with those of other approaches.

1 Various Attitudes to Time Resource

1.1 Computability theory

In the study of computability, the only resource issue concerned is whether the resource cost of a problem-solving activity is finite. In the definition of a computational device, such as a Turing machine, a computation is defined as the process from the initial state of the system to a final state, with a problem as input and a solution as output. A problem is computable as long as the machine stops after finite steps, and the number of steps does not matter (Hopcroft and Ullman, 1979).

In this field, the attitude to resource can be summarized as: the time spent in the problem-solving activity can be ignored, given that it is finite. People can wait a period with an arbitrary-but-finite length for a solution. Also, what counts as a "solution" for a given problem is determined by the choosing of final states, which is resource-independent. In other words, whether a state is final, or whether a solution has been found, is defined by some criterion, in which the resource cost of the solution is not taken into account.

1.2 Computational complexity theory

Though the above abstraction is necessary for certain purposes, people are often unsatisfied by it. Obviously, for almost all practical purposes, time is valuable, and people hope to solve problems as soon as possible.

Under the assumption that the system's hardware (processors and memory) remains constant, the time spent on a problem depends on two factors: the problem itself and the system's method for the problem.

A system's problem-solving methods are often defined as *algorithms*. An algorithm is usually designed for a certain *type* of problem, which has many specific *instances*. If we can find a natural way to measure the *size* of these instances, we can then evaluate the algorithm's time-efficiency by its *complexity function*, which relates the size of an instance to the time used by the algorithm on that instance. This is the territory of computational complexity theory (Rawlins, 1992).

According to the form of complexity functions, algorithms are put into a hierarchy of *complexity classes* — *constant*, *logarithmical*, *polynomial*, *exponential*, and so on. Algorithms with low complexity are preferred. In particular, a problem is feasible, or tractable, if it has a solution whose worst cost is at most polynomial. The problems without polynomial algorithms are intractable — the cost of a solution will become astronomical figures for large instances of the problem.

When working within this field, people first decide what a solution is for a given problem, then look for an algorithm that can get that solution for the problem. If the algorithm is polynomial or faster, it is usually what we want (even though there may be faster ones to be found), otherwise we will try to find a faster one, and so forth.

In this way, resource expense is taken into consideration in a quantitative manner. We are no longer satisfied by the knowledge that the cost will be a finite number — we want that number to be as small as possible. However, the solution of a given problem is still defined in a resource-independent way — a bad result cannot be referred to as a "solution" because it is easy to get.

1.3 Time pressure

The above philosophy does not work in certain situations. For many problems, we still do not have polynomial algorithms, and for many others, even polynomial algorithms are too slow. It is often the case that a *time requirement* is an intrinsic component of a problem, and a result must meet the requirement to be accepted as a solution of the problem. Consequently, the system works under a *time pressure*. In such a situation, the time needed for a perfect solution usually exceeds the time requirements of the problems — otherwise the time requirement can be ignored. This can happen even if the algorithm used by the system only takes constant time. For example, if an algorithm solves a problem by searching a file thoroughly, which takes 2 seconds in the given hardware, then such an algorithm cannot be used when a solution must be provided within 1 second.

For a system to work in this situation, the criterion for a result to be a "solution" of the given problem have to be relaxed. In fact, for many problems, whether a result can be count as a solution is a matter of degree. Instead of saying whether a result is a solution or not, here people need to compare, or even measure, the qualities of different solutions. Usually, the quality of a solution depends on its resource cost, which includes its processor-time expense. To work under a time pressure means to give up best solutions, and to find a trade-off between solution quality and time cost (Good, 1983).

1.4 Constant time pressure

The above situation is the usual case, rather that exceptions, in the fields of real-time systems and artificial intelligence. In these domains, the above time pressure is treated in different ways..

One way is to take time pressure into account when an algorithm is designed. As discussed previously, now we cannot spend as much time as we wish to get the the best solution. Consequently, we have to relax our request for the solution from "best" to "satisfactory", then look for an algorithm which can get such a solution as fast as possible. This approach leads to concepts like *approximation algorithm* and *heuristic algorithm* (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).

The above algorithms are designed with respect to the time pressure in the application domain, but run just like ordinary algorithms. They are still Turing machines. The only difference, when compared with the time-pressure-free algorithms, is that the set of final states is much larger. Consequently, time pressure becomes a constant factor that affect the construction of the machine, and remains unchanged during the life-cycle of the system.

1.5 Variable time pressure

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 at the end. Under different time pressure, the same algorithm may provide different solutions for the same problem.

1.6 Meta-level planning

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 research projects under the name of "deliberation scheduling" (Boddy and Dean, 1994), "metareasoning" (Russell and Wefald, 1991), and "flexible computation" (Horvitz, 1989).

1.7 Dynamical resource allocation

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 "parallel terraced scan" strategy developed by Hofstadter's research group provides an example of dynamical resource allocation (Hofstadter and the Fluid Analogies Research Group, 1995).

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 reallocate 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 distribute its time resource *unevenly* among its tasks, and dynamically adjusts its bias according to new results.

2 NARS: Working under Insufficient Resources

In this section, a new approach for working under insufficient resources is developed and discussed. This approach is used for resource management in the *Non-Axiomatic Reasoning System* (NARS) (Wang, 1994; Wang, 1995).

2.1 The problem

As a general-purpose reasoning system, NARS accepts knowledge provided by the user in a formal language, and answers questions according to available knowledge and a set of inference rules. The user can assign two types of information-processing tasks to the system:

New knowledge. After getting a piece of new knowledge, the system derives its implications, according to the previous knowledge.

Question. After getting a question, the system looks for an answer for it, according to the available knowledge.

What distinguishes NARS from other reasoning systems is that it is designed to be *adaptive 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 can accept, as long as they are expressible in the formal language.

Here *insufficient resources* indicates a stronger form of time pressure, compared with what people usually call "real-time" or "resources-limited" problems, for the following reasons:

1. The resources is not only "limited", but also "insufficient" — that is, in short supply. It directly follows that the system usually cannot find all implications of a piece of new knowledge, or answer a question by taking all relevant knowledge into account.

2. The same task may have different time requirements attached in different context, therefore the time pressure on the system is not only problem-dependent, but also context-dependent.

3. New tasks may show up at any time, therefore the system cannot precisely predict what will happen in the future, and plan everything in advance.

The memory space of the system is in short supply, too. Consequently, something has to be removed when the system's knowledge base is overloaded.

The insufficiency of knowledge also influence the system's resource management. Here it means that all knowledge is revisable by new evidence, and that plausible answers are needed when no certain ones can be found.

If a system has to work in this situation, what is the best way (the "Type II rationality") for it to manage its resources?

In the remaining part of the paper, the resource management mechanism of NARS is discussed. For other components of NARS (such as knowledge representation and inference rules), see (Wang, 1994; Wang, 1995).

2.2 Controlled concurrency

As discussed previously, in NARS to “carry out a task” means to interact the task, which may be a piece of knowledge or a question, with available knowledge. Such a process consists of a sequence of *inference steps*, in each of which a piece of knowledge is applied to the task. In principle, the process can stop after any number of steps, and the more step it goes, the better is the result, just like in anytime algorithms.

When should the system stop processing a task? Ideally, as in conventional computer systems, it should be when the task has been “finished”. For a piece of new knowledge, it means that the system has generated all of its implications. For a question, it means that the system has found the best answer, according to its knowledge. In both cases, the system needs to access all relevant knowledge. However, under the assumption of insufficient resources, such an exhaustive use of knowledge is not affordable for NARS. When time is in short supply, some knowledge has to be ignored, even if it may be relevant.

In such a situation, the most often used method is to retreat to a “satisfactory solution” from a “perfect solution”. For example, we can limit the maximum step of forward inference for new knowledge, or set a threshold for the quality of the answers. These methods are widely used in heuristic-search systems, but they are too inflexible for the purpose of NARS. Because the resources request–supply situation is constantly changing in NARS, such thresholds are sometimes too high — the system still cannot satisfy them, and sometimes too low — the system cannot get a better result when the resources are still available.

The solution used in NARS is simple: to use as much time as the system can supply in the current context. In this situation, to stop processing a task is usually not caused by the finding of a satisfactory result, but by the shortage of time.

Because time becomes a resource competed by tasks, the system should has a policy to explicitly manage it. Obviously, when there is only one task in the system, it gets all the processing time. However, because NARS is always open to new tasks, competition become inevitable. What should the system do when a new task shows up while it is still busy with a previous one? Basically, there are two ways to distribute time among tasks: sequential and parallel. “Sequential” means to process the tasks one by one, and “parallel” means to have more than one task being processed at the same time.

The sequential mode, though good for many other purposes, is unsuitable for the current situation. Let us suppose that the system is working on task A when task B appears. If the system shortly stop working on A and completely turns to B, then it may happen that B turns out to be a very easy or very hard problem — in either case, after a short time, the quality of the result for B become hard to be improved, though A’s result would be much better if the system had not abandoned it so soon. On the other hand, if the system let B wait too long, another task C may join the competition, which is “unfair” to B. Anyway, when the processing results heavily depend on unpredictable future events, sequentialism is a bad idea, because it makes decisions that cannot be undo.

Parallel processing, or time-sharing, is much more flexible in this situation. The system can process both A and B at the same time by dividing its time resources between them, and dynamically adjust the proposition of allocation to achieve a better overall efficiency when situation changes.

It should be stressed for this purpose we do not need parallel processing at the hardware level (i.e., multiple processors) — such an implementation is possible, but is not necessary for the model. Even when a system like NARS is implemented in a multiple-processor machine in the future, the picture will not change much. Under the assumption of insufficient resources, the processor-time will still be in short supply, and the tasks have to share the computing capacity of the system, though in this time there are more than one processors.

To process tasks in parallel does not mean to allocate time evenly among them. As mentioned before, there are various reasons that the system should spend more time on some tasks than on the others. One major reason is that the user often has different requests on different tasks. Some tasks need urgent replies, and some other tasks need more careful considerations. As a real-time system, NARS must be able to handle these time requests. On the other hand, the system hope to achieve a better overall resource efficiency by spending more time on the premising tasks.

To reflect the above bias on time allocation, a "label" needs to be assigned to each task in NARS. The most common way of indicating a time request in real-time systems is to assign a "deadline" — a certain amount of time, to each task. This approach is inappropriate for NARS. By requesting an answer to be reported at a certain time, d, the concept "deadline" implicitly assumes a step function for the answer's utility — that is, an answer provided before d does not get extra credit, and an answer found after d is completely useless. Such an assumption is not suitable for many situations. On the other hand, usually the system cannot determine how much time it can spend on a task when the task is provided (by the user) or generated (by the system itself), because it depends on future events, for example, whether an answer will be found soon, and how many new tasks will show up in the near future.

Because what NARS is designed to achieve is not to obtain answers of certain quality, but to use its resources as efficiently as possible when resources are in short supply, what really reflects the time resources obtained by a task is not a (absolutely measured) deadline, but a (relatively measured) "share", which depends on both the requirement of the user and the internal situation of the system.

An *urgency* measurement is defined for this purpose, which is a real number in the interval (0, 1]. At a certain instant, if the urgency of task t_1 is u_1, and the urgency of task t_2 is u_2, it means that the amounts of time resources the two task will get have a ratio $u_1 : u_2$.

In this way, urgency is a completely relative measurement. An urgency value itself, such as $u_1 = 0.4$, tells us nothing about when the task will be finished or how much time the system will spend on it. If it is the only task in the system at the moment, it will get all processor time. If there is another task with $u_2 = 0.8$, the system will spend twice as much time on t_2 as it will on t_1.

To realize such an urgency-based time distribution, NARS can be seen as having a task pool, in which each task has an urgency value attached. The system's processor time is cut into constant time-slices, and in each slice a single task is processed. At the beginning of each time-slice, all tasks in the pool have chance to be processed, and the *probability* for a certain task to be chosen is proportional to its urgency — that is, $p_1 : p_2 = u_1 : u_2$. In this way, urgency indeed indicates the (expected) relative speed of a task.

If the urgency of tasks remain constant, then a task that comes later will get less time than a task that come earlier, even when they have the same urgency value. A natural solution is to introducing an "aging" factor for the urgency of tasks, to let the urgency values *decay.* In NARS, each task has a decay value attached, which is a real number in (0, 1). If a task has an urgency value u and a decay value d, that means after a constant time, the urgency of the task will be ud. In this way, decay is also defined as a relative measurement. 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 get the same amount of time resources, but when u_1 decreases to 0.4, u_2 will only decreases to 0.8, so the latter will then get twice as much time as the former. Given this definition, if a task has a *high* decay value, it decays *slowly.*

If the decay value of a task remain constant, the corresponding urgency will become a function of time:

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

where u_0 and d_0 are the initial values of urgency and decay, respectively, and c is a constant. This is in fact a "forgetting" function used by many psychological models of human memory (Anderson, 1990).

Taking the integration of the above urgency function, we get the *expected relative time-cost* of a task:

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

(as a relative measurement, the constant is omitted).

Here we see that the amount of time spent on a task is determined both by its urgency and its decay. By assigning different urgency values and decay values to tasks, the environment (i.e., a human user or another computer system) can put many independent types of time pressure on the system. For example, we can inform the system that some tasks need to be worked on right now but that they have little long-term importance (by giving them high urgency values and low decay values), and that some other tasks should be processed for a longer time (by giving them high decay values). Of course, other combinations are also possible, and if the user does not care, the system uses default urgency and decay values.

Aging is not the only factor that change the urgency distribution among the tasks. The amount of time spent on a task is not determined only by the requirement of the user, but also by the result(s) the system has got on it. For example, if the system has found a good answer for a question, it should become less urgent, compared with unsolved questions. Its decay value should also be decreased.

For these reasons, each time a task is processed, the system reevaluates its urgency and decay values, to reflecting the current situation. As a result, NARS maintains a *dynamical* urgency distribution in its task pool. From the tasks provided by its user, NARS constantly generate derived tasks according to available knowledge. The implications of a piece of new knowledge are new knowledge themselves, and the solving of a derived question can help to solve an original (user provided) question. In NARS, the derived tasks are also assigned urgency and decay by the system, then put into the task pool. After that, they are treated just like the tasks provided by the user.

Because the system only has constant memory space, the task pool cannot expand infinitely. Consequently, some tasks have to be removed. As discussed previously, when this happens to a task, it is not because the processing of the task has reached certain predetermined standard, but because the task has lost in the competition for resources. For this reason, when the task pool is full, a task with the lowest urgency is removed.

Such a policy makes the actual resources expense of a task context-sensitive. Even if we provide the same task to the system, with the same urgency and decay, it may be processed differently: when the system is busy (that is, there are many other tasks with higher urgency), 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 any-time algorithms. The actual number of steps to be carried out is determined both by the initial assignment of its urgency and decay, and by the resources competition in the system.

If the task is a question asked by the environment, when to report an answer? According to computation theory, an answer is reported at the "final state" when the system stops working on the question. However, when time is treated as important resource, it is reasonable to ask the system to report an answer as soon as it is found. As discussed above, when an answer is found, it does not necessarily mean that the system will stop processing the task. According to the logic used in NARS, it is always possible to find a better answer for a question, no matter what has been found (Wang, 1994). As a result, the system may report more than one answers for a question — it can "change its mind" when new evidence is taken into account, like trial-and-error procedures (Kugel, 1986). The system keeps a record of the best result it has found for each question, and when a new candidate is found, it is compared with the recorded one. If the new one is better (for how the quality of a answer is measured, see (Wang, 1994)), it is reported, and the record is updated. Of course, it is also possible for a question to be removed from the task pool before any answer is found for it.

Even if a "parent" task has been removed, the "children" tasks derived from it may still be processed, given that they have a high-enough urgency. For example, when solving a question Q, two derived questions Q_1 and Q_2 are generated (by backward inference). Later, the system finds an answer for Q_1, which leads to an answer for Q. Then, the urgency values of Q and Q_1 are decreased more rapidly that that of Q_2, and it is possible for Q_2 to be under processing after Q is removed from the system's task pool. If a system only want to answer questions asked by the user, the above phenomenon should be avoided, because Q_2 is a means to solve Q. However, the goal of NARS is to adapt to its environment, so Q_2, as a derived question, has a value for its own sake — it may appear again in the future, independently to Q.

In summary, a new control mechanism, "controlled concurrency", is defined like this: The user provides the system tasks from time to time, and each task has an urgency value and a decay value attached. In each step the system picks up a task (according to a probability distribution determined by the urgency distribution), generates new tasks (from that task and relevant knowledge), then adjust the urgency of the task according to its decay value and the result of the inference. A answer is reported to the user if it is the best the system has found for a user asked question. Tasks with the lowest urgency are removed as a result of memory space overload.

2.3 Bag-based memory organization

As described above, in each inference step the system chooses a task, according to the urgency 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 in the previous subsection. With insufficient resources, the system cannot consult all the relevant knowledge for a task. On the other hand, knowledge cannot be used indiscriminately — some knowledge is more important and useful than the other. However, such a ranking of knowledge cannot be precisely planned in advance, and it is often the case that the only way to determine whether a piece of knowledge is useful is to use it.

The similarity in problems hints similar solutions. Indeed, it is not difficult to generalize the idea of "controlled concurrency" to a more abstract form. Let us say that a system has some "items" to process in a certain way by an anytime algorithm. Because new items may come at any time, and the time requests of the processing go beyond the system's capacity, it is impossible for the system to satisfy all of them completely. Instead, the system has to distribute its time resources unevenly among the items. The system evaluates the relative priority of each item according to several factors, and adjusts the evaluation when the situation changes. Besides, the system's storage capacity, which is a constant, is also in short supply, meaning that the system may remove an item before reaching the "logical end" in its processing.

Because the above situation repeatedly appears in the environment where the system's resources are insufficient, it is helpful to define it as a class of objects. According to the theory of object-oriented programming, an object is a data structure with applicable operations defined on it.

Let us define a class called "bag". A bag can contain a constant number of "items". Each item has a *priority value*, which is a real number in [0, 1]. There are two valid 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 effect is to put the item into the bag. If the bag is already full, an item with the lowest priority is removed. The operation *take-out* has no argument, and returns an item when the bag is not empty. Its effect is to take an item out of the bag. The probability for an item to be chosen is proportional to its priority.

Now we can describe the memory structure of NARS in terms of bags. NARS uses a *term logic* (Wang, 1994). This kind of logic is characterized by the use of subject–predicate sentences and syllogistic inference rules, as exemplified by the logics of Aristotle and Peirce (Englebretsen, 1981). A property of term logic is that all inference rules require the 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 "A dove is a bird", we know that the knowledge that can directly interact with it must has a "dove" or a "bird" in it (as subject or predicate). If we put all tasks and knowledge that share a common term together, call it a *chunk*, and name it by the shared term, then we are sure that all valid inferences happens within chunks. Obviously, such an approach introduce a certain amount of redundancy into the system. For example, the task "A dove is a bird" belongs to both chunk "dove" and chunk "bird".

Defined in this way, a chunk become a unit of resources management, which is larger than a task or a piece of knowledge. The name of a chunk is a term, and the body of the chunk contains the meaning of that term — according to the experience-grounded semantics accepted by NARS (Wang, 1994), the meaning of a term is determined by its experienced relations with other terms. The memory of the system is simply a group of chunks.

Now the operation of choosing a task takes two steps: choosing a chunk first, then choosing a task from it. In other words, the system at first distributes its resources among the chunks, then, within each chunk, among the tasks. The result is a two-level structure. On both levels, the "bag" class can be used. In summary, the memory of NARS is a bag of chunks, and each chunk consists of a task bag and a knowledge bag.

Now we can see the distinction between *task* and *knowledge* more clearly. All questions are tasks. New knowledge are 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. By such a distinction, the system gets (1) a small number of tasks, which are active, kept for a short time, and closely related to current situation; and (2) a huge amount of knowledge, which is passive, kept for a long time, and not necessarily related to current situation.

When the system is running, the following "execution cycle", or "inference step", is repeated until the process is interrupted by the user:

1. Check input buffer. If there are new tasks, put them into the corresponding chunks.

2. Take out a chunk from the chunk bag.

3. Take out a task from the task bag of the given chunk.

4. Take out a piece of knowledge from the knowledge bag of the given chunk.

5. Apply inference rules on the given task and knowledge. Which rule is applicable is determined by the combination of the task and the knowledge, and different rules generate different results.

6. The priority and decay values of the given task, knowledge, and chunk are reevaluated according to the quality of the results. Then the task, knowledge, and chunk are returned to the corresponding bags.

7. The results generated in this step are put into the input buffer as new tasks. If a result happen to be a good answer of a question asked by the user, it is reported to the user.

To explain how the priority and decay values are calculated, it is inevitable to address many technical details of the NARS model, and thus is beyond the scope of this paper. Such an explanation can be found in (Wang, 1995).

It is possible to implement the above procedure in such a way that an inference step takes roughly constant time, no matter how large the involved bags are (Wang, 1995). Such a step is like an "atomic" instruction of problem-solving activities. However, unlike in conventional computing programs, where instructions are organized into algorithms in advance, in NARS the instructions are "linked" together for a given problem *dynamically* in run time, and how they are linked is context-dependent.

3 Comparison and Discussion

A new approach for working under insufficient resources has been introduced previously. Though the description omits many details, it is already enough for us to see some of the implications of the approach. In the following, this approach is compared with other approaches, and some of its properties are discussed.

3.1 Rational results

Like all other approaches that take the limitation of resources into consideration, NARS gives up the requirement for optimum results, and turns to look for the best results the system can get under the constraints of available resources — what Good calls "type II rationality" (Good, 1983).

What distinguish NARS from other approaches is the way the constraints are specified. By interpreting "insufficient knowledge and resources" as being finite and open, and working in real time, the constraints assumed by NARS are stronger than the assumptions accepted by other approaches. For example, as discussed previously, only a few systems are designed to deal with variable time pressure or unexpected (both in content and timing) questions.

Many features of NARS directly follow from this assumption. For example, 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 omniscience" (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 (chunk, 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.

These phenomena remind 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. The important thing here is to notice that, 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 questions 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" are precisely the problems for them our knowledge and resources are insufficient.

Another notable characteristics of NARS is that it does not work as a function that maps questions to answers, like most computer systems do. For a given question, it may keep silence if it has no clue at all or it is busy on something else. Even if it provides an answer after some processing, it is always possible for the system to change its mind after the arrival of new knowledge or further consideration — no result is "final". Therefore, NARS is not suitable for the questions for which only the optimal answers are needed, such as many mathematical problems.

As a more concrete property, we have seen that the "bag" structure supports efficient processing under insufficient time–space resource, but it does not support operations like "to retrieve a specific item", or "to access all items exhaustively", which are supported by ordinary data structure, such as arrays and linked lists.

In summary, "rational results" (resource-dependent) and "correct results" (resource-independent) are often different, and are produced by different mechanisms.

3.2 Flexible resource consuming

For a given information-processing task, how much resources will be spent on it by a system? In conventional computer systems, it is determined exclusively either by the system or by the user.

If the task is processed by a traditional algorithm, the processing stops at predetermined final states. Therefore the time cost is determined completely by the computational complexity of the algorithm and the size of the task. It has nothing to do with the user or the context. Though in a time-sharing system, the response time depends on the load of the system, the processor-time that is really spent on the task remains roughly the same.

If the task is processed by an anytime system, the decision falls into the user's hand, who can interrupt the algorithm at anytime to get a result.

In NARS, there are three factors that altogether determine the time spent on a given task: the time request of the user, the design of the system, and the current context. As described before, the user can assign an urgency value and a decay value to a task, otherwise default values are used. Because both of the values are given in relative forms, the actual time allocated to the task is decided by the allocation mechanism and the current situation of resource competition. In this way, neither the designer nor the user have complete control, and the system *itself* participates in the decision, by taking the current situation into account.

This approach is more flexible than traditional algorithms, because the user can influence the resource allocation at run time. It often happens that the same type of problem may have different time request, which is hard to be satisfied by a predetermined standard for final states.

NARS is a real-time system, and the bag structure consistently implements some techniques used in real-time systems, such as pruning, ordering, approximation, and scoping (Strosnider and Paul, 1994). In traditional real-time systems, time requests are indicated by deadlines (Laffey et al., 1988; Strosnider and Paul, 1994). This method is not suitable for NARS, because here new tasks may appear at any time, and thus no pre-designed mechanism can satisfy all possible combinations of deadlines. On the other hand, stop processing at a deadline often means a wast of resource, when the system become idle after it. For an adaptive system, the tasks that appeared in the past may happen in the future again, with some variants. Therefore, even when the user no longer needs a result after a certain amount of time, the system still have reason to work on it, if there are resources available.

Such an approach is also more similar to the resource management mechanism of the human mind. Obviously, the human mind is a real-time system that response to different time requests, but it seldom stop thinking a problem at a deadline, even if when such a deadline exists, such as in an examination.

For NARS, the concept of "computational complexity" no longer exists for a task, because the time cost is context-dependent and unpredictable in principle. In this situation, it does not make sense to discuss whether the system's processing to a certain task is tractable or not (Bylander, 1991; Levesque, 1988; Valiant, 1984). For the same reason, there is no combinatorial explosion anymore. It does not mean, however, that the system can solve all hard problems rapidly, but that the system will not be paralyzed by them — it knows when to temporarily ignore them or to permanently abandon them.

Also, this approach suggests a new interpretation and solution to the "scaling up" problem. It is well-known that many AI projects work fine at experiment stage with small data sets, but fail to work in real-world situations. This is often caused by the "sufficient resource" assumption accepted by the approaches. Such an assumption is usually made implicitly in the operations carried out by the approach, such as to exhaust possibilities for a specific purpose. These operations are affordable on small amount of data, but become inapplicable when the database (or 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 database by exhausting possibilities, and also do not attempt to do so when the database 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 database is huge, but the degradation happens in a *graceful* way, just like what happens to the human mind in similar situations.

3.3 Asynchronous parallelism

The controlled-concurrency mechanism was inspired by the time-sharing in operating systems. However, there are some crucial difference between the two. In NARS, the parallel processed tasks (1) consult a shared knowledge base, (2) access knowledge according to the current priority distribution in the knowledge base, (3) change the priority distribution after each step, and (4) can be stopped after any number of steps. As a result, the mutual influence among the problem-solving activities become very strong.

As described previously, there is no predetermined method for each task, and how a task is processed is completely "knowledge-driven" — determined by the recalled knowledge. On the other hand, which piece of knowledge will be recalled at each step is determined by many events happened in the past. 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. If B's priority is relatively high (compared with the coexisting tasks), it will have chance to react with more knowledge, therefore get a more thorough processing, otherwise it will be forgot soon. In this way, tasks are processed in a context-sensitive manner, rather than 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 (Frisch and Haddawy, 1994) or asynchronous parallelism (Hofstadter and the Fluid Analogies Research Group, 1995). In NARS the processing of a task becomes unpredictable and un-repeatable (from the initial 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 (Hofstadter, 1993).

3.4 Is this still computation?

With the control mechanism described previously, it is easy to see that even for the same task, with the same priority and decay, the results provided by the system at different time may be different (though not necessarily so). How a task is treated depends on what knowledge the system has, how the knowledge is organized, and how much resources the task gets — simply speaking, it is influenced by the system's experience, which includes not only the events happened before the task shows up, but also the events happens after that. The contents, the order, and the timing of the events all matter. Furthermore, a question may get no answer, one answer, or more than one answers.

An interesting and important question follows: in this way, is the system still *computing*?

Given the definition of a Turing Machine, M, a *computation* is the procedure for M to transform from its initial state q_0 to one of its final state q_f, under the effect of the input data d_i. At the final state, M provides an output d_o as the result of the computation. We can identically 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).

A conventional computing system works in a sequential and deterministic way:

wait for the user to input a new task,
get a task,
process that task,
report the result,
reset the working memory,
wait for the user to input a new task,
etc., etc.

The characteristics of such a working mode, as implied by the definition of *computation*, are:

1. There is a unique *initial state* in which (and only in which) the system accepts input tasks, and the tasks are processed one by one. If a task arrives when the system is still busy with another task, the new task has to wait. Even if the interruption mechanism is taken into consideration, the picture is fundamentally the same.

2. The system always provides the same result for the same task, no matter when the task is processed.

3. The amount of resources spent on a task is a constant, which mainly depends on the complexity of the involved algorithm and the amount of relevant knowledge, but independent to when the task is processed.

4. There are some predetermined *final states* where the system will stop working on a task and provides a result, no matter whether there are other tasks waiting to be processed.

As described previously, NARS does not work in this way.

In NARS, there is no unique "initial state" where the system waits for new tasks. When the system is running, tasks can be accepted at many different states.

There is no "final state" for a task, neither. For some tasks, if their priorities are (relatively) too low, it is possible for them to be completely ignored. If a tentative answer is reported for a question, usually neither the system nor a human user can predict whether there will be a better answer to be reported later, if more knowledge is took into consideration. It is undecidable whether an answer is "THE" answer for the question, since it depends on events that happen in the future, such as whether the system will get new knowledge related to the task, or whether more time will be spent on it.

On the other hand, the system has an initial state when its memory is empty (the system is "born" without any innate knowledge). Then its state changes as it communicate with the user and processes the tasks. The system never returns to the initial state, until a user terminates the precessing and erases all of its memory. After that, the system can be "reborn" with the same "genetic code" — inference rules, control mechanisms, and so on. However, unless the experience of the system precisely repeats what happened in its "previous life", the system's behaviors will be different. In summary, 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 three scales, in term of what is referred to as its input and output.

In the scale of each execution cycle, or inference step, as defined at the end of the previous section, the system's activity is computation, where the input is the given task and knowledge, and the output is the inference results of that step.

In the scale of each problem-solving cycle, where the input is a question, and the output is an answer to the question, 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 become computation again. Concretely, if we take an arbitrary state of NARS, q_1, as an "initial state", the state the system arrives after a certain amount of time, q_2, as a "final state", then we can see what NARS does during that period of time as computation, with its experience (user provided tasks) during that time as the input, and its responses (system generated reports) as the output. In this scale, the system still works as a function or algorithm that deterministically maps an input to an output.

In summary, the behavior of NARS can be described at different scales. NARS is still computing in some of them, but not in the others. What happens here has been pointed out by Hofstadter as "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 modeled on the machines we call ‘computers’ ” (Kugel, 1986). On the contrary, though conventional computer systems can also be described in these scales, they are computing in all of them.

For practical purposes, what we are interested in is the relationship between questions and answers. From the above discussion, we see that in NARS their relationship cannot be referred to as computation. On the other hand, NARS can still be implemented in von Neumann computers in specific, and Turing machines in general.

This conclusion is not just a novel way to see things, but has important methodological implications. According to such an analysis, when a system like NARS is designed, the designer should not try to decide what answer to provided for a given question — it should be decided by the system itself at the run time, and the designer simply cannot exhaustively consider all possible situations in advance (the designer, hopefully, is also an intelligent system, so bound by insufficient resources). For similar reasons, the designer cannot decide how much resources to spend on a certain task, which is also context-dependent. In general, the designer is no longer working on domain-specific algorithms or general-purpose algorithms (like GPS), but *meta-level algorithms*, which manage resources, carry out inferences, and so on (as described in the previous section). With these meta-level algorithms, the system deals with environment provided tasks *without* (task-oriented) object-level algorithms. In this way, the problems solved by the designer and the problems solved by the system itself are clearly distinguished.

In light of this opinion, we can 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 intelligence to solve domain problems, then put the solutions into computer systems in the form of task-specific algorithms. The computer systems only execute the algorithms on specific instances of the problems, which can hardly be referred to as “solving the problem intelligently”.

NARS is creative and autonomic in the sense that its behaviors are determined not only by its initial design, but also by its “personal” experience. It can generate judgments and questions not predicted by its designer, and work on them by its own choice. A “tutor” can “educate” it by manipulating its experience, but cannot completely control its behaviors due to the complexity of the system. This is neither always a good thing, nor always a bad thing, from a pragmatic point of view. An adaptive system with insufficient knowledge and resources has to behave in this way.

4 Summary

In this paper, a new approach for time resource management is introduced, which is used in the NARS project. This approach is designed for the “insufficient knowledge and resources” situation, which is defined by the three properties: finite, open, and real-time. Therefore, the new approach is not necessarily better than the other approaches in all situations.

The “insufficient knowledge and resources” situation has special importance from the viewpoint of artificial intelligence and cognitive science (Wang, 1995). Briefly speaking, many AI problems have this property, and many human behaviors can be explained from this assumption. Since few traditional theory makes this assumption, new approaches become specially wanted.

The control mechanism of NARS can be summarized like this: the processing of a task is divided into steps, and a sequence of steps works on a task like an anytime algorithm — it is interruptible, and the quality of the result is improved incrementally. The system distributes its resources among the tasks (and subtasks) in a time-sharing manner, and dynamically adjusts the distribution according to the feedback of each step. The step sequence that is applied to a given task is linked together at run time, according to the system's knowledge structure, which is also dynamically formed and adjusted.

A system designed in this way behaves very different from the traditional computing system, though it can still be implemented in ordinary computers.

Acknowledgment

This work has been supported by a research assistantship from the Center for Research on Concepts and Cognition, Indiana University.

References

Anderson, J. (1990). *The Adaptive Character of Thought.* Lawrence Erlbaum Associates, Hillsdale, New Jersey.

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

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

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

Englebretsen, G. (1981). *Three Logicians.* Van Gorcum, Assen, The Netherlands.

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

Frisch, A. and Haddawy, P. (1994). Anytime deduction for probabilistic logic. *Artificial Intelligence*, 69:93–122.

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

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

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, D. (1993). How could a copycat ever be creative? In *Working Notes, 1993 AAAI Spring Symposium Series, Symposium: AI and Creativity*, pages 1–10.

Hofstadter, D. and the Fluid Analogies Research Group (1995). *Fluid Concepts and Creative Analogies: Computer models of the Fundamental Mechanisms of Thought.* BasicBooks, New Nork.

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

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

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

Levesque, H. (1988). Logic and the complexity of reasoning. In Thomason, R., editor, *Philosophical Logic and Artificial Intelligence*, pages 73–107. Kluwer Academic Publishers, Boston.

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

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

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

Valiant, L. (1984). A theory of the learnable. *Communications of the ACM*, 27:1134–1142.

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

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

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