The Iterative Multi-Agent Method for Solving Complex Search Problems

Kamran Karimi

Department of Computer Science
University of Regina
Regina, Saskatchewan
Canada S4S 0A2
karimi@cs.uregina.ca

Abstract. This paper introduces a problem solving method involving independent agents and a set of partial solutions. In the Iterative Multi-Agent (IMA) method, each agent knows about a subset of the whole problem and can not solve it all by itself. An agent picks a partial solution from the set and then applies its knowledge of the problem to bring that partial solution closer to a total solution. This implies that the problem should be composed of sub-problems, which can be attended to and solved independently. When a real-world problem shows these characteristics, then the design and implementation of an application to solve it using this method is straightforward. The solution to each sub-problem can affect the solutions to other sub-problems, and make them invalid or undesirable in some way, so the agents keep checking a partial solution even if they have already worked on it. The paper gives an example of constraint satisfaction problem solving, and shows that the IMA method is highly parallel and is able to tolerate hardware and software faults. Considerable improvements in search speed have been observed in solving the example constraint satisfaction problem.

1 Introduction

Encoding a real-world problem to be solved by a computer is often time consuming, as it requires a careful mapping process. If the representation inside the computer matches the original problem domain, then the design and implementation can be done faster, and there will be fewer bugs to deal with. Many complex problems are composed of different sub-problems, and solving them requires the collaboration of experts from different fields. One example is designing an airplane. The sub-problems usually interact with each other, and a change in one place can invalidate the solution to another part. This can happen in unpredictable ways. This form of emergence is usually considered a nuisance in engineering fields, as it makes it harder to predict the behavior of the whole system. Handling this emerging interaction is not trivial for big problems, and is worse when the problem domain is vague and there is limited or no theoretical knowledge about the possible interactions between the sub-problems.

In this paper we propose to use the Iterative Multi-Agent (IMA) method for solving complex problems. Section 2 explains the IMA method in general and outlines the

basic principles. Section 3 presents a practical application of the IMA method to solve a computationally demanding Constraints Satisfaction Problem. Section 4 concludes the paper and outlines some of the future directions of this work.

2 Segmenting A Complex Problem

The IMA method is a decentralized method involving the use of multiple *agents*, in the form of software processes, to solve complex search problems. Each agent has knowledge relevant to part of the problem, and is given responsibility for solving one or more sub-problems. The agents do not collaborate directly, and the final solution to the problem *emerges* as a result of each agent's local actions. Overlap is allowed in the agents' knowledge of the problem and in their assigned sub-problems.

Often a problem P can be expressed in terms of n sub-problems, P_1 to P_n, such that $P = \bigcup_{i=1\dots n} P_i$. Each sub-problem P_i can have a finite or infinite number of candidate solutions $S_i = \{S_{i1}, S_{i2}, \dots\}$, some of which may be wrong or undesirable. Each partial solution of the whole problem contains a candidate solution for each sub-problem. We represent a partial solution as an array $[S_{1j}, S_{2j}, S_{3k}\dots]$, where S_{1i} is a candidate solution to sub-problem P_1, S_{2j} is a candidate solution to sub-problem P_2, and so on. Segmenting a bigger problem into sub-problems makes the task of writing agents to solve the sub-problems easier, as a smaller problem is being attacked. As well, if a sub-problem is present in many problems, it may be possible to reuse an agent that solves that sub-problem. This translates to saves in design, implementation and debugging efforts that would otherwise be needed for the development of a completely new agent. The system can achieve some fault tolerance if the agents have overlapping information [4]. The crash of an agent (or the computer it is running on) will not prevent the whole system from finding a solution if the combined knowledge of other agents covers the lost agent's knowledge of the problem. This is of special interest in long-running programs. If needed, the system can provide security by restricting sensitive knowledge to trusted agents.

Interaction is common in complex systems made of sub-problems. To make it more probable that a solution can be found, a set of partial solutions is kept. This set can be managed as a workpool for example. The set can be initialized by randomly generating partial solutions, some or all of which may be wrong. Having more than a single partial solution matches the existence of more than one agent, and makes a parallel search for a solution possible. Having a set of solutions to work on also means that we can end up with different final solutions, possibly of different qualities. For example, some may be elegant, but expensive to implement, and others may be cruder, but cheaper. These solutions can be used for different purposes without the need to develop them separately. This enables a two level problem solving strategy. At the first level, "hard requirements" are met in the final solutions. At the second level, "soft requirements" are considered in the process of selecting one or more of the final solutions produced at the first level.

The following describes how the system works. The problem is defined in terms of its sub-problems, and agents are assigned the job of solving these sub-problems. Each agent checks one partial solution at a time to see if it satisfies its assigned sub-

problem(s). If not, it tries to create a new partial solution. The new partial solution may be invalid for another sub-problem, so other agents will have to look at it later and if needed, find another solution for the sub-problem with the invalidated solution. This is an iterative process, because agents can repeatedly invalidate the result of each other's work in unpredictable ways. Each agent is searching its own search space for solutions to its part of the problem, in the hope of finding a place that is compatible with all other parts. All the agents should work on a partial solution in order to make sure that all the sub-problems in that partial solution are solved. This decentralized form of activity by the agents makes it possible for them to run in parallel, either on a multiprocessor or on a distributed system. A partial solution is removed from the workpool when all the agents have looked at it and none needed to change it, signifying that it has turned into a total solution. They can signal this by setting flags, and the first agent that notices that all the flags are set can take the work out of the pool. Any changes done by one agent results in all flags being reset. The algorithm can stop when one or more solutions have been found. To make sure the program terminates, it can count the number of times agents work on partial solutions, and stop when it reaches a value determined by the user. The best partial solution at that time can then be used.

The system designer can use random methods to divide the problem, assign sub-problems to agents, create the initial partial solutions, and arrange for the agents to visit the partial solutions. Alternatively, he can choose to use deterministic methods in any or all of these activities. But even when everything in the system design is deterministic, the algorithm in general does not guarantee finding a solution. That is because predicting the effects of the interactions between the agents may not be possible by the designer. There is also no guarantee of progress towards a solution. However, if the agents operate asynchronously, it is less probable that a specific sequence of changes will be performed repeatedly, which in turn makes it less probable that the system enters a cycle. In this case, if a global solution exists, then given enough time the system is likely to find it.

IMA is different from methods that use genetic operations. Genetic methods use random perturbations followed by a selection phase to move from one generation to the next. None of these concepts exists in IMA. The main element that introduces randomness into the picture is the interaction between the agents' actions.

3 Segmenting A Constraint Satisfaction Problem

This section demonstrates how the IMA method can be applied to a simple form of Constraint Satisfaction Problem (CSP). It is easy to automatically generate test CSPs, and they are decomposable into interacting parts. Since we want to apply the IMA method to exponential problems, we use the backtracking method to solve the generated CSPs.

3.1 Constraint Satisfaction Problem

In a CSP we have a set of variables $V = \{v_1, v_2...\}$, each of which can take on values from a set $D = \{d_1, d_2...\}$ of predefined domains, and a set $C = \{c_1, c_2...\}$ of constraints on the values of the variables. A constraint can involve one or more variables. Finding the solution requires assigning values to the variables in such a way that all the constraints are satisfied. Sometimes we need all such assignments, and at other times just one is enough. It may be desirable to have partial solutions, which are variable assignments that satisfy some, but not all of the constraints.

Many problems can easily be formulated as CSPs, and then standard methods can be used to solve them. CSPs have been used in scheduling, time tabling, planning, resource allocation and graph coloring. In most cases finding a solution to a CSP requires domain-specific knowledge, but some general methods are applicable in many situations. The traditional way of solving a CSP is to assign a value to one variable and then see if the assignment violates any constraints involving this variable and other previously instantiated variables. If there is a violation, another value is chosen; otherwise we go to the next unassigned variable [3]. If we exhaust the domain of a variable without success, then we can backtrack to a previously instantiated variable and change its value.

3.2 Related Work

Several methods rely on multiple-agents to solve a CSP. The *Distributed Constraint Satisfaction Problem* (DCSP) is defined as in [6]. In DCSP the variables of a CSP are divided among different agents (one variable per agent). Constraints are distributed among them by associating with each agent only the constraints related to its variable. The *asynchronous backtracking* algorithm [6] allows the agents to work in parallel. Unlike the classic backtracking algorithm, it allows processes to run asynchronously and in parallel. Each agent instantiates its variable and communicates the value to the agents that need that value to test a constraint. They in turn evaluate it to see if it is consistent with their own value and other values they are aware of. Infinite processing loops are avoided by using a priority order among the agents. In *Asynchronous weak-commitment* search [6], each variable is given some initial value, and a consistent partial solution is constructed for a subset of the variables. This partial solution is extended by adding variables one by one until a complete solution is found. Unlike the asynchronous backtracking case, here a partial solution is abandoned after one failure.

In [1], *cooperating constraint agents* with incomplete views of the problem cooperate to solve a problem. Agents assist each other by exchanging information obtained during preprocessing and as a result improve problem solving efficiency. Each agent is a constraint-based reasoner with a constraint engine, a representation of the CSP, and a coordination mechanism. This agent-oriented technique uses the exchange of partial information rather than the exchange or comparison of entire CSP representations. This approach is suited to situations where the agents are built incompatibly by different companies or where they have private data that should not be shared with others.

3.3 Segmented Constraint Satisfaction Problem

In the Segmented CSP (SCSP) each constraint is considered a sub-problem. A variable can be shared among more than one constraint, so there is interaction between sub-problems. The SCSP Solver (SCSPS) uses a workpool model of parallel processing [2]. A number of partial solutions of the problem are created by assigning random values to each variable. Multiple agents are then started to solve the CSP. Each of these agents is given some knowledge about the problem by assigning it a set of variables and a set of constraints. There is no need to worry about the interdependencies among the constraints assigned to different agents. Each agent determines the values that can be assigned to its assigned variables. Having a constraint entitles the agent to test its validity. Agents ignore other variables and constraints in the problem, as they do not need to know anything about other agents' knowledge of the problem. This greatly eases the designer's initial knowledge-segmenting job, which can even be done randomly. Any changes made later to an agent's knowledge of the problem will not necessarily result in a wave of changes in other agents. It also does not matter if the variables in a constraint are not assigned to the agent that has that constraint, as other agents that are assigned those variables will be responsible for changing their values.

Dividing the problem among the agents means that each one of them has to search a smaller space to solve its portion of the problem, and so it can run faster, or run on slower hardware. If there are v variables and c constraints in the problem, each agent j will have to deal with $v_j < v$ variables and $c_j < c$ constraints. Agents can use any method in solving the portion of the problem assigned to them. Each agent may have to do a complete search of its own assigned space more than once. This is because a given state in agent j's search space may not be consistent with the current state of another agent k, while that same state may work when agent k goes to another location in its search space.

Each agent gets a partial solution from the pool, tries to make it more consistent, and then returns it to the pool. The partial solution is accessible to only one agent while it is out of the workpool. An agent never interacts with others directly because all communication is done through the workpool. This simplifies both the design and the implementation by reducing the amount of synchronization activity. If there is enough work in the pool, all the agents will be busy, ensuring that they can all run in parallel. In a multi-processor or a multi-computer [5], this could result in execution speed-up. Agents can disturb other agents' work by changing the value of a common variable, or by signaling the need for a change in a variable. This means that a partial solution must be repeatedly visited by the agents.

3.4 The Demonstration Program

The program SCSPS·java implements SCSPS. It is written in Java and developed using Sun's Java Development Kit version 1.2. The test computer was a 120MHz Pentium with 96MB RAM and running Windows 95. SCSPS creates a problem by generating some variables and a set of constraints on them. The constraints are of the form $x + y < \alpha$, where $x \in \{x_l,\ldots,x_h\}$ and $y \in \{y_l,\ldots,y_h\}$are positive integers within a

specified range, and α is a constant. It is possible to create harder or easier problems by changing the domain or constraint limits. Figure 1 shows one possible set of variables and the constraints on them.

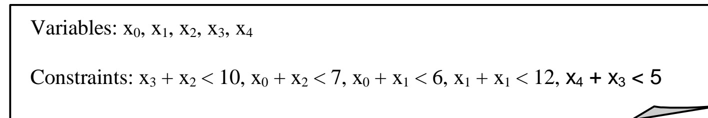

Fig. 1. The variables and constraints of an example CSP.

One practical use for this type of constraints is in Job-shop scheduling, where each job has a start time *start* which is a variable, and a duration *duration* which is a constant. The problem can be defined as the requirement that for each pair of jobs j and k we should have only one of $start_j + duration_j < start_k$ or $start_k + duration_k < start_j$. An expert decides which one of these two constraints is to be present in the final set of constraints. The aim is to find suitable values for all *start* variables so that all the constraints are satisfied.

After the random problem is generated, the program creates a workpool and fills it with random partial solutions, which are probably inconsistent. Agents are then created as independent Java threads and each is randomly assigned some variables and constraints. It is possible for some variable(s) and constraint(s) to be assigned to more than one agent. Having a variable enables an agent to change its value. Having a constraint enables an agent to test it. An agent with an assigned constraint should have read access to the variables of that constraint. The agents run in a cycle of getting a partial solution from the pool, working on it, and putting it back. After completing a cycle, the agents wait for a small, randomly determined amount of time before going on to the next iteration, thus making sure that there is no fixed order in which the partial solutions are visited. The randomness present in the design means that the solutions will differ from one run to the other, even when working on the same problem. The workpool counts the number of times it has given partial solutions to the agents, and terminates the program as soon as it reaches a predetermined value. In general it is possible to let the agents run indefinitely, or until all the partial solutions are consistent. The main thread of the program runs independently of the agents and can automatically check all the partial solutions in the workpool and print the inconsistencies. Figure 2 shows two of the agents working on the example CSP.

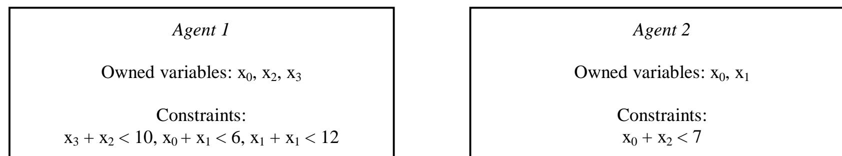

Fig. 2. Two agents working on the example CSP.

Figure 3 shows the workpool for the example CSP and some of the partial solutions it contains. The partial solutions change over time.

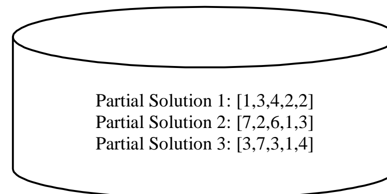

Fig. 3. Workpool of partial solutions for the example CSP.

A partial solution for n variables can be considered a vector representing a point in an n dimensional space. Each agent has to deal with only $m < n$ variables. There is a "change" flag associated with each variable, which is used by agents to signal the need for that variable's value to change because its current value violates a constraint. This need is detected by an agent that has the violated constraint, but the actual change should be done by one of the agents that is assigned the variable.

Each agent starts the processing by getting a partial solution from the pool and then changing the value of all the variables that it is assigned and which have their "change" flag set. These variables should change because, as detected by other agents, their values violate some constraints. A partial solution will thus move from its currently unsuitable point. This is done in the hope of finding another point that satisfies more constraints. Agents then try to ensure that the variables they are assigned satisfy the constraints they are aware of. Each constraint $x + y < \alpha$ can be one of three types, and what the agent does depends on this type.

1. If both x and y are assigned to the agent, a slightly modified version of backtracking is used to find suitable values for x and y. The modifications have to do with the fact that here finding a solution is a multi-pass process. For instance, the variables are changed from their current values, as opposed to a fixed starting point, thus making sure that the whole domain is searched
2. If only one of x and y are assigned to the agent, only the value of the variable that is assigned to the agent is changed, and the other variable is considered a constant.
3. If none of x and y are assigned to the agent, then none of them is changed.

Variables are changed by incrementing their values, with a possible wrap-around to keep them within the specified domains. This is to make solving the problem harder, as a trivial solution for this type of constraints is to simply use the smallest values of the variables. After this phase, each agent inspects its constraints of the second and third types. If it finds an inconsistency, it sets the "change" flag of the unassigned variable(s). This is done because this agent has done all it can, and now is signaling the failure to others. Another agent that is assigned these inconsistent variables will get this work later and change the values. The agents continue like this until the workpool's counter for the number of checked-out partial solutions reaches the limit. At this point no more partial solutions are given out and the agents stop executing. The main processing loop in each agent is shown in figure 4.

1. Get a partial solution from the pool
2. Alter my variables that have their “change” flag set by other agents.
3. Use my constraints to find a consistent value for my variables.
4. Check the constraints with one or two missing variables; set their “change” flag if an inconsistency is found.
5. Give the partial solution back to the pool.

Fig. 4. The algorithm followed by each agent in SCSPS.java.

Table 1 contains the results of several runs of the program to solve six problems using 1, 5, 10 and 20 agents. 30 runs were attempted for each row. There were 21 partial solutions in the workpool, and they were created randomly for each run. The values under the columns *Variables*, *Constraints*, and *Agents*, represent the corresponding values in the generated problem. No checks were done to see if a partial solution has been turned into a total solution, and so the value of *Rounds* determined the number of times the workpool manager gave out partial solutions to the agents before the program stopped. This value is 1 when there is a single agent in the system, because in this case a standard backtracking method was employed, and a single solution was sought (multi-agent runs usually find many more solutions). Runs that did not finish within the time limit of 120 seconds were aborted. The number of aborted runs is indicated under the *Timed out* column. The 4 rows within each outlined box correspond to the same problem.

Vars	Constraints	Agents	Rounds	Timed out	Run Time (ms)			
					Avg	Best	Worst	Std Dev
10	10	1	1	0	130	100	390	57
10	10	5	15000	0	18006	17520	24600	1428
10	10	10	15000	0	8958	8840	9070	66
10	10	20	15000	0	6148	4720	14830	2858
10	20	1	1	0	186	50	930	198
10	20	5	15000	0	21765	18070	33780	4924
10	20	10	15000	0	9588	9060	10820	388
10	20	20	15000	0	5447	4890	5830	211
20	20	1	1	8	-	50	-	-
20	20	5	15000	0	17712	17570	18290	132
20	20	10	15000	0	9522	8890	22960	2569
20	20	20	15000	0	4899	4660	6310	335
20	40	1	1	4	-	60	-	-
20	40	5	15000	1	-	17680	-	-
20	40	10	15000	0	9704	8840	19230	2373
20	40	20	15000	0	4855	4720	5930	214
50	50	1	1	27	-	110	-	-
50	50	5	15000	0	23478	17680	62940	12731
50	50	10	15000	0	8998	8900	9192	51
50	50	20	15000	0	6383	4670	52240	8661
50	100	1	1	30	-	-	-	-
50	100	5	15000	3	-	18070	-	-
50	100	10	15000	2	-	9060	-	-
50	100	20	15000	0	5801	4840	13780	2497

Table 1. The results of several runs of the program.

As expected, a single-agent backtracking method's success rate deteriorates quickly, until it stops finding any solutions within the time limit. The IMA method, on the other hand, shows very good scalability and performs like a constant-time problem solver for the range of problems in this table. One could speed up the process

of finding answers at the top area of the table by decreasing the value of *Rounds*. For example, 2000 rounds would be quite enough for the first 8 rows.

An interesting observation is that having more agents is helping the system find solutions faster. This is to be expected, even though on a single-processor computer introducing m agents to solve the problem adds an overhead of $O(m)$. The reason is that, assuming a non-overlapping segmentation of the problem into sub-problems P_i, each with a search space size of α_i, then the whole search space is of size $\Pi\alpha_i$. This includes $O(a^n)$ or $O(n!)$ problems as special cases. While each of α_i values may represent a small and manageable space, $\Pi\alpha_i$ can represent a huge search space. We have thus converted the problem of searching a huge space to that of searching many smaller ones, even though the nature of the problem is unchanged. As is the case in combinatorial problems, reducing n by a little can dramatically speed up the process of finding a solution to that problem. This compensates for the effects of having more than one agent. Table 2 confirms this, and shows that increasing the number of agents beyond a point will eventually have a rather small, negative effect on execution time. Here 101 partial solutions were in the work pool, and 30 runs were performed for each row.

Vars	Constraints	Agents	Rounds	Timed out	Run Time (ms)			
					Avg	Best	Worst	Std Dev
30	30	5	20000	0	26771	23390	63720	9875
30	30	10	20000	0	11958	11860	12080	56
30	30	20	20000	0	6678	6260	15380	1647
30	30	40	20000	0	7301	5110	10110	1757
30	30	60	20000	0	8022	5380	10050	1626
30	30	80	20000	0	8255	5820	10550	1732
30	30	100	20000	0	8651	5930	10820	1556

Table 2. The effects of increasing the number of agents on the execution time.

One observation from the results of the runs with a high number of agents was that because of the value of *Rounds*, in many cases a partial solution that was invalid at the start, and was not visited by all the agents, turned into a total solution. One reason for this is that these randomly generated partial solutions had some of their sub-problems already solved. The other reason is that due to the high number of agents and the resulting redundancy in the system, other agents were able to move the partial solutions towards total solutions. Either way this hints at the ability of the system to tolerate faults. More fault tolerance can be achieved by using more than one workpool, preferably in different computers, so the crash of one machine will not destroy all the partial solutions.

4 Conclusion

We proposed the Iterative Multi-Agent (IMA) problem solving method that involves the following activities: dividing the problem into sub-problems, dividing the knowledge to solve the sub-problems among multiple agents, and having a set of partial solutions. Then an iterative process starts, in which agents make changes to the solution of each sub-problem if necessary. The design of the system mimics the behavior of human experts collaborating to solve a problem. But here more than one

possible solution is considered. Each expert looks at the problem from his perspective, and tries to solve his sub-problem by using the resources made available to him. Conflicts may arise, and then each expert has to modify his part of the plan in the hope of making it compatible with the other parts. This seems to be an intuitive approach that works in practice. The two more important advantages of following this guideline are a more natural mapping process for problems that can easily be expressed in terms of interacting sub-problems, and in a reduction of the search space size. One disadvantage of the IMA method is that in general there is no guarantee that a solution will be found. Another disadvantage is that it may not be possible to find a globally optimum solution when there is no authority with knowledge about the whole problem. This could be tolerable in hard problems where finding a solution at all is good enough.

This paper gave a practical example of co-operating agents that run in parallel and take part in solving a CSP. The source code of the implemented system (SCSPS.java) can be obtained freely by anonymous FTP from orion.cs.uregina.ca/pub/scsps or by contacting the author.

For future we intend to apply the IMA method to other problem areas. Studying the effects of changing and inconsistent knowledge in the agents is also of interest to us. Making the system utilize a multi-processor machine or run over a network are other worthwhile efforts, as they enable the tackling of even bigger problems.

Acknowledgements

Special thanks to Howard Hamilton and Scott Goodwin for their help and suggestions for improving the paper. This research was supported by an NSERC Strategic Grant (Hamilton).

References

1. P. S. Eaton, E. C. Freuder, "Agent Cooperation Can Compensate For Agent Ignorance In Constraint Satisfaction", *AAAI-96 Workshop on Agent Modeling*, August 4-8, 1996, Portland, Oregon.
2. J. Knopp and M. Reich, "A Workpool Model for Parallel Computing", Proceedings of the First International Workshop on High Level Programming Models and Supportive Environments (HIPS), 1996.
3. B. A. Nadel, "Constraint satisfaction algorithms", *Computational Intelligence*, No. 5, 1989.
4. B.J. Nelson, "Fault-Tolerant Computing: Fundamental Concepts", *IEEE Computer*, vol. 23, July 1990
5. A. S. Tanenbaum, *Distributed Operating Systems*, Prentice-Hall International, 1995.
6. M. Yokoo, E. H. Durfee, T. Ishida, K. Kuwabara, "The Distributed Constraint Satisfaction Problem: Formalization and Algorithms*", IEEE Transactions on Knowledge and Data Engineering*, vol. 10, No 5, 1998.