Divide and (Iteratively) Conquer!

Kamran Karimi, Howard J. Hamilton and Scott D. Goodwin

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

Abstract. *The Iterative Multi-Agent (IMA) Method works by breaking down a search problem into many sub-problems, each to be solved separately. Independent problem solving agents work on the sub-problems and use iteration to handle any interaction between the sub-problems. Each agent knows about a subset of the whole problem and cannot solve it all by itself. The agents, working in parallel, can use any method to solve their assigned sub-problem. 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 solution to the whole problem then emerges as a result of these local activities. This requires the problem to be decomposable into smaller ones. The paper gives an example of constraint satisfaction problem solving. We solve randomly generated Traveling Salesman Problems, and compare the results with those of other methods.*

Keywords- search, iteration, workpool, combinatorial problem, Constraint Satisfaction Problem, Traveling Salesman Problem.

1 Introduction

The Iterative Multi-Agent (IMA) Method is a way of dealing with complex search problems by decomposing them into smaller sub-problems, and then solving them separately. The intuition behind it is that solving many smaller problems, possibly in parallel, will enable us to handle harder ones. This is true especially for combinatorial problems, where on any given hardware platform, reducing the problem size by a small amount may have a big effect on its solvability. The agents only care about their currently assigned sub-problems, and these may not be independent. The sub-problems usually interact with each other, and a change in one place can invalidate the solution to another part. This means that a change to the solution of one sub-problem could affect the solutions to any or all other sub-problems. This can happen in unpredictable ways. To handle this, the agents iterate over the solutions. Thus the same sub-problem can be visited and solved many times.

In this paper we propose to use the Iterative Multi-Agent (IMA) method for solving combinatorial problems. Section 2 explains the IMA method in general and outlines the basic principles. Section 3 shows how a Constraint Satisfaction Problem can be formulated for the IMA method. Section 4 gives another example, based on the Traveling Salesman Problem, and presents detailed experimental results. Section 5 concludes the paper.

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.

Suppose a problem P can be expressed in terms of n sub-problems, P_1 to P_n, such that $P = \bigcup_{i=1\ldots n} P_i$. Each sub-problem P_i can have a finite or infinite number of candidate solutions $S_i = \{S_{i1},$

S_{i2}, …}, 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}…], 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.

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. After being assigned a sub-problem, an agent checks one partial solution at a time to see if it satisfies the sub-problem. 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 Solving Constraint Satisfaction Problems

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 Introduction to CSPs

In a CSP we have a set of variables $V = \{v_1, v_2\ldots\}$, each of which can take on values from a set $D = \{d_1, d_2\ldots\}$ of predefined domains, and a set $C = \{c_1, c_2\ldots\}$ 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.

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 [4]. If we exhaust the domain of a variable without success, then we can backtrack to a previously instantiated variable and change its value.

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.

3.2 Segmenting a Constraint Satisfaction Problem

Here 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 program uses a workpool model of parallel processing [3]. 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.

3.3 Using the IMA Method

We wrote a program in Java to test the IMA method in solving CSPs. [2]. The program creates a problem by generating some variables and a set of constraints on them. The constraints are of the form $x + y <$, where $x \{x_l,..,x_h\}$ and y $\hat{I}\{y_l,...,y_h\}$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.

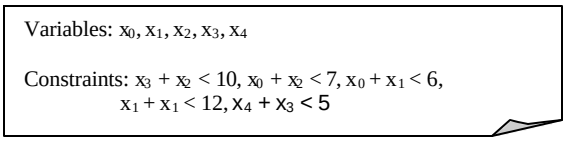

Fig. 1. The variables and constraints of an example CSP.

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.

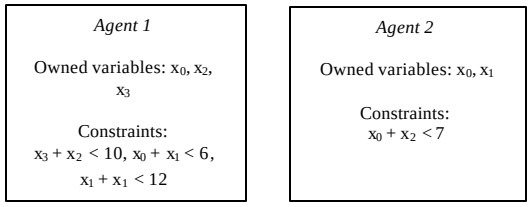

Fig. 2. Two agents working on the example CSP.

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

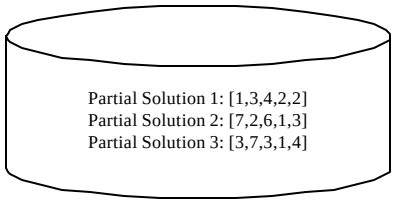

Fig. 3. Workpool of partial solutions for the example CSP.

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.

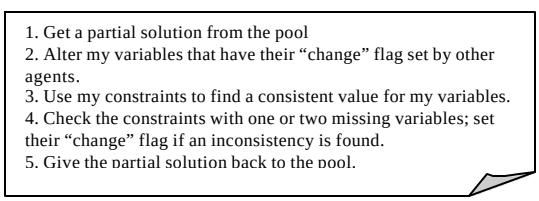

Fig. 4. The algorithm followed by each agent in the CSP Solver

The results of running the SCSPS.java program to solve randomly generated CSPs are given in [2], where we witnessed very good scalability.

4. Solving the Traveling Salesman Problem

In the Traveling Salesman Problem (TSP), the aim is to find the shortest path that takes us from a given starting city, through a number of other cities without visiting any of them more than once, and then back to the first city. This is an exponential problem with a search space of size n!. Many practical applications of TSP, such as circuit board drilling or VLSI fabrication must be solved for big values of n. For such input values finding the optimal solution is not possible, and an approximate solution is acceptable.

4.1 Segmenting a TSP

To reduce the size of a TSP, one has to reduce the number of cities to be visited. This means that we will visit only *m (m < n)* cities. A TSP with *m* cities is faster to solve, but it will not solve the original problem with *n* cities. The IMA method considers a TSP with *n* cities as several smaller problems of size *m*.

Given a tour the *n* cities c_1, c_2, … c_n, it chooses sub-tour of length *m* starting at *p* (1 < *p* *n*) and ending at *q* such that $|p - q| = m$. The value of *p* is determined randomly. Then it uses a brute-force method to solve sub-problem of finding the shortest path from c_p to c_q under the constraint that the starting city, c_1, is not moved. This ensures that the solution will remain valid for the original problem.

As an example, consider a tour of 9 cities that starts at city *A*. Suppose *m* is 5, so each agent considers 5 cities at a time. An agent gets a tour from the workpool with the values shown in figure 5, and decides to work on elements 3 to 7 (5 cities). The target for minimization are cities *E, D, F, H,* and *B*.

A	B	C	D	F	H	B	G	I

Fig 5. The original tour

The agent then tries to find an arrangement of these cities that results in the minimum distance when travelling through them. Suppose the new order is *B, F, H, E, D*. The resulting new tour is then as shown in figure 6.

A	C	B	F	H	E	D	G	I

Fig 6. The modified tour

The changes done by one agent can cause the whole tour's quality to deteriorate, that is why other agents will look at it again and work on a different sequence of cities.

4.2 Solving a TSP Using the IMA Method

We wrote a Java program to solve randomly generated TSPs. It uses many of the same techniques as in solving a CSP. It generates and solves TSPs using a brute-force search, the greedy, and the IMA methods. It was developed using Java development kit 1.2.2. The test computer was a 166MHz Pentium with 96MB of RAM.

Random tours of the *n* cities are generated with all of the tours having the same first city and put in an array. The tours are considered circular, so it is assumed that we return to the first city after the last one. These are then put in a workpool, where agents can take them out, work on them, and put them back. There are more tours than the agents, so on parallel hardware their execution will overlap. Each agent uses a brute-force method to find the shortest path between two cities.

The distances between the cities are generated randomly, and are represented by a two-dimensional array. The program does not assume anything about the world where the cities are located, so it does not matter whether or not the distances between them are symmetrical or Euclidean. There is also no need for a pre-processing phase to cluster the node using geometric decomposition methods, as nodes are all treated the same. The agents all run the same code and there is no need for any load balancing activity among them.

Each time a tour is taken out of the workpool, only a subset of the problem is solved, with no

regard to the effects this may have on the whole solution. This is another example of when solving one sub-problem can disturb the solutions to other sub-problems. The keyword here is iteration. Each sub-problem is visited many times, and over time it is hoped that the whole tour improves.

We have compared IMA (each agent using brute-force), with greedy and brute-force (BF) methods. The results are shown in Table 1. The results of the brute-force method are the best possible. Entries in the highlighted sections belong to the same problem.

Method	Cities	Agents	Cities/ Agent	Iteration	Length	Time (ms)
BF	10	1	10	1	181	9133
Greedy	10	1	10	1	362	91
IMA	10	5	5	40	243	350
BF	10	1	10	1	182	9434
Greedy	10	1	10	1	232	90
IMA	10	5	6	40	187	1913
BF	11	1	11	1	217	127714
Greedy	11	1	11	1	359	80
IMA	11	5	6	40	233	1102
BF	11	1	11	1	220	143446
Greedy	11	1	11	1	331	100
IMA	11	10	6	40	236	7971
BF	11	1	11	1	234	146400
Greedy	11	1	11	1	411	120
IMA	11	5	7	40	234	50192
BF	11	1	11	1	135	127454
Greedy	11	1	11	1	283	90
IMA	11	10	7	40	141	20860
BF	11	1	11	1	198	143596
Greedy	11	1	11	1	312	90
IMA	11	20	7	40	217	18357
BF	12	1	12	1	161	1385201
Greedy	12	1	12	1	282	80
IMA	12	10	7	40	175	16323
BF	12	1	12	1	160	1323226
Greedy	12	1	12	1	315	80
IMA	12	10	8	40	166	139531

Table 1. Results of solving randomly generated TSPs.

Increasing the number of cities per agent improves the quality of the IMA method. As in the CSP case, IMA shows very good scalability compared to the brute-force method. The greedy method is very fast, but the quality of its solutions are worse than IMA.

We tried the greedy and the IMA methods for much larger number of cities (between 1000 and 4000). In these cases the greedy method could beat IMA both in terms of speed and the quality of the solutions. This shows that the IMA method's main use is in improving the speed of combinatorial algorithms.

5 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.

We applied the IMA method to randomly generated Traveling Salesman Problems. We saw speed-ups when compared to a brute-force method. It produced results with better quality than those of greedy method. This places the IMA method somewhere in between a combinatorial exhaustive search and a backtrack-free greedy method. Whether IMA is suitable for a particular application should be determined by a domain expert.

References

[1] P. S. Eaton and 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] K. Karimi, "The Iterative Multi-Agent Method for Solving Complex Search Problems," *Proceedings of the Thirteenth Canadian Conference on Artificial Intelligence (AI'2000)*, Montreal, Canada, May, 2000.

[3] 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.

[4] B. A. Nadel, "Constraint satisfaction algorithms", *Computational Intelligence*, No. 5, 1989.

[5] A. S. Tanenbaum, *Distributed Operating Systems*, Prentice-Hall International, 1995.

[6] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara, "The Distributed Constraint Satisfaction Problem: Formalization and Algorithms*", IEEE Transactions on Knowledge and Data Engineering*, vol. 10, No 5, 1998.