Multi Agent Systems and Game Theory

Introduction

The goal of this project is to solve computationally expensive problems by using Multi Agent Systems and Game Theory.

In general, we are interested in 2 type of problems:
1. Problems that a bit hard to solve, but anyway needs to be solved within certain amount of time. For example, weather forecasting that gives some forecast about the evening's weather is meaningless if takes few hours and gives results when it's almost evening.
2. Problems that so costly in terms of computation that needs few years to solve.

The idea itself is related to some own experience with computationally expensive problems. I remember, last year I was running some algorithm on NP-Complete problem where n was equal to 20. After running it 25 hours and still not having any results, I just decided to halt the process.

Usually you see that people use Multi Agent Systems and Game Theory for a simulation of traffic or something like this, and you realize that it's not that you want to do.
Having deep connection to algorithms and interested in computationally expensive problems, I have decided to use Multi Agent Systems to solve NP-Complete or NP-Hard problem.

Let us give an example. Graph coloring is an NP-complete problem, it means that the optimal solution for graph coloring algorithm on an arbitrary graph grows exponentially as the size of the graph increases.

The distinguishable property of many graph algorithm problems is their complexity. As graphs get larger, the computations on them can become very time consuming and complicated. So our goal is to find ways for decreasing that complexity when it is possible and make algorithms faster.
One optimal approach for this goal is parallelism: when the problem is divided into smaller pieces that can be executed and solved at the same time.

Note, detailed description about the implementation of my system is presented in later chapters.

Task Decomposition

In the case of graphs, there exists a very intuitive way of dividing the problem into smaller sub problems. We can divide the given graph into the set of sub graphs.

We can apply our method as follows: find the strongly connected components and color them simultaneously. Then we can merge the coloring results by joining the strongly connected components.

First we find the strongly connected components and merge them into a single vertex. We end up with a new graph, on which we can compute any property we want. At the end, we merge the results by using the contracted graph. If the graph is complex enough, we can again divide it into its own strongly connected sub graphs.

Although this approach helps us to reduce running time of a graph coloring, in some cases the merging operation can be expensive enough to prevail the savings of the subgroup decomposition tactic. But, in general, this appears to be a useful way of finding the optimal coloring solution.

Related Problems

The other use of strongly connected component computing algorithms may be in social networks.
Let us consider a group of people that are connected to each other by the university they attend or any other common thing they have. We can define this group as strongly connected. Many people in this group will probably have common interests, for example, pages or games in Facebook. The strongly connected component algorithms can be used for finding and analyzing such groups, suggesting them commonly liked pages and games, or just for improving advertisement showing process based on the communities that users belong to.
Also, the strongly connected component algorithms can be applied for finding instances in a huge data hub that may be connected to each other and make suggestions. A good example may be Facebook friend recommendations or LinkedIn connections (first, second, third).

The other use of strongly connected component algorithms is cluster identification. We may think of an each item as a vertex and add an edge between them if they are considered "similar". The strongly connected components of this graph will represent different types of items.

In addition, strongly connected component algorithms may be used to:

1) Compute the Dulmage–Mendelsohn decomposition
2) Solve 2-satisfability problems
3) Solve some transportation related problems
4) Improve running time of algorithms that can be implemented using decomposition-merging method

If we find a way to distribute the computation among a hundred processors and get the results in a few minutes, then our goal is accomplished. But there is a limitation in this approach: we are not able to stop the growth of the execution time by adding a finite number of processors. But in some cases parallelism can be a fast and a handy solution and help us to solve a particular problem, in contrast with not being able to solve it efficiently at all.

All these problem are pretty similar to each other in terms of divide and conquer, and if we pick up any of them and solve it, then we can solve other ones as well.

Task Decomposition Complexity

There are 3 famous algorithms for computing strongly connected components of the graph. That is: Kosaraju's, Tarjan's, and Gabow's algorithms. Let us consider the runtime and test results of each of them.

The Kosaraju's algorithm performs two depth-first searches on the graph and one graph reversion. The funny part of this algorithm is that many programmers knows how to code it but have no idea why it work in that way, because it's pretty simple, but includes few tricks.

Two complete traversals of the graph takes twice the running time of depth-first search. DFS runs in $O(|V| + |E|)$ time if we use an adjacency list, and $O(|V|^2)$ time if you use an adjacency matrix. Hence, an adjacency list is asymptotically faster if the graph is sparse.

For reversing the graph, we traverse all adjacency lists (if we use adjacency lists). If the graph is represented as an adjacency list then the algorithm runs in $O(|V| + |E|)$ time, if the graph is represented as an adjacency matrix then the algorithm runs in $O(|V|^2)$ time.

So, the total runtime of Kosaraju-Sharir algorithm equals to $2 * O(V + E) + O(V + E) = O(V + E)$.

Tarjan's and Gabow's Algorithms Complexity analysis:
The Tarjan's algorithm and the Gabow's algorithm performs only one complete traversal of the graph. If the graph is represented as an adjacency list then the algorithm runs in $O(|V|+|E|)$ time. If the graph is represented as an adjacency matrix then the algorithm runs in $O(|V|^2)$ time.

Comparison Decomposition of Algorithms

Although, both Kosaraju and Tarjan algorithms have asymptotically linear running time, the Tarjan algorithm is more efficient than Kosaraju, because the algorithm uses DFS traversal only once. In other words, the Tarjan's algorithm runs faster than the Kosaraju-Sharir algorithm, because of the constant factor of running time.

Now, let's look at Tarjan and Gabow algorithms, which are more interesting in terms of comparisons of running time.

Few results of the experiments done on Tarjan's and Gabow's algorithms, are represented bellow.

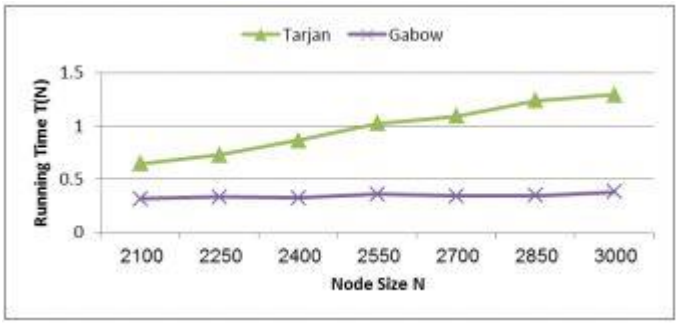


Figure 1: Gabow's vs Tarjan's running time (in seconds) comparison on Dense in dynamic graph representation

In Figure 1 the running times of both algorithms on Dense graph is represented based on test results. It is explicit, that Gabow's taking less time to compute strongly connected components of directed Dense graph.

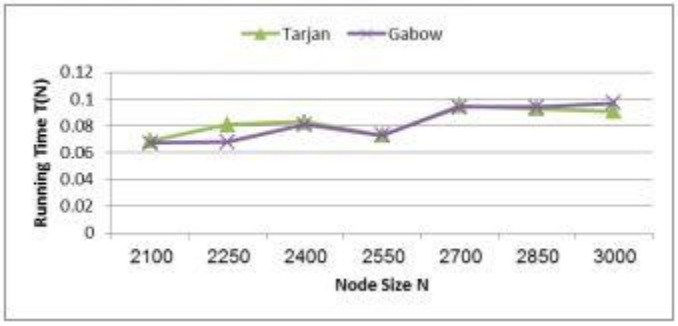


Figure 2: Gabow's vs Tarjan's running time (in seconds) comparison on Sparse in dynamic graph representation

However, when the number of edges is small, Gabow's and Tarjan's have the almost the same efficiency rate. The results of this case is depicted in Figure 2.

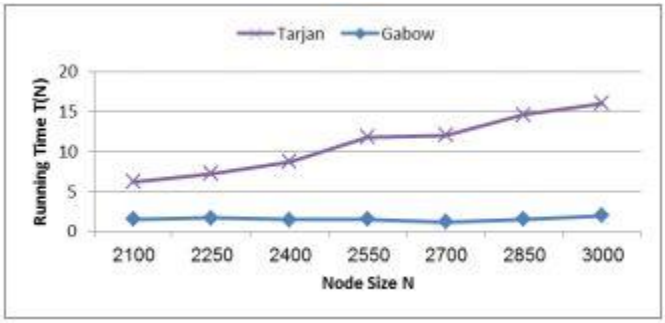


Figure 3: Gabow's vs Tarjan's running time (in seconds) comparison on Complete in dynamic graph representation

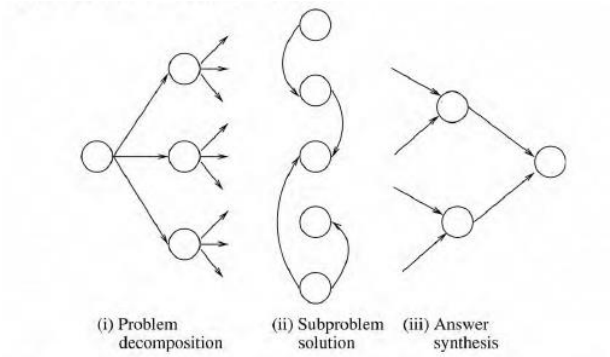
Figure 3 shows the test results for Complete graph. It turns out that Gabow's algorithm is doing better than Tarjan's algorithm as the number of nodes and edges goes up.

The Gabow's algorithm seems to be faster when it runs on Dense or Complete graphs, and Tarjan's performance is similar to Gabow's algorithm when number of edges in a graph is minimum. So, we can conclude that in our collection of algorithms, we will give more priority to Gabow's algorithm in case if graph is complete or dense, but if the graph is sparse then we will use Tarjan’s algorithm.

Cooperative Distributed Problem Solving

Our primary focus is on cooperative game theory, which analyzes optimal strategies for groups of individuals.
CDPS studies how a loosely coupled network of problem solvers can work together to solve problems that are beyond their individual capabilities. Each problem solving node in the network is capable of sophisticated problem solving and can work independently, but the problems faced by the nodes cannot be completed without cooperation.

CDPS process can be viewed as a three-stage activity as follows.



1. Problem decomposition: In this stage, the overall problem to be solved is decomposed into smaller sub problems. The decomposition will typically be hierarchical. In our particular example, that is graph coloring, the decomposition is done by finding the strongly connected components mentioned above.
2. Sub problem solution: In this stage, the sub problems identified during problem decomposition are individually solved. This stage typically involves sharing of information between agents: one agent can help another out if it has information that may be useful to the other. For example, the colors of already colored vertexes.
3. Solution synthesis: In this stage, solutions to individual sub problems are integrated into an overall solution. As in problem decomposition, this stage may be hierarchical, with partial solutions assembled at different levels of abstraction.

In addition, this is more like the "Prisoner's Dilemma" game, which models a situation in which:
a) there is a gain from cooperation
b) but each player has an incentive to free ride.

Task Announcement

Task announcement is done by First-Price-Bid auction. As the task manager ('Deleverer') got new task, it makes broadcast to all the agents.
Agents in the net listen to the task announcements and evaluate them with respect to their own specialized hardware and software resources (in our test example random generated number have been used as time required to solve the problem). When a task to which a node is suited is found, it submits a *bid.* A bid indicates the capabilities of the bidder that are relevant to the execution of the announced task. A manager may receive several such bids in response to a single task announcement. Based on the information in the bids, it selects the most appropriate agent to execute the task (minimum time required agent will be selected). The selection is communicated to the successful bidders through an *award* message. These selected agent assume responsibility for execution of the task. After the task has been completed, the agent sends a *report* to the manager.
One can notice, that all the processes are done with all the standards of multi agent system communications.

Basically, this is all about cooperation, and all the agents are interested to solve as many problems as possible.
The advantages of this type of auction is that it's fast as requires 1 round. And the disadvantages is that the deliverer should wait everyone to reply before being able to process.

In another type of problem delivered to the agent (not graph coloring), there can be point for negotiation, thus the system may become both cooperation and negotiation game.

Benefit of Proposed System

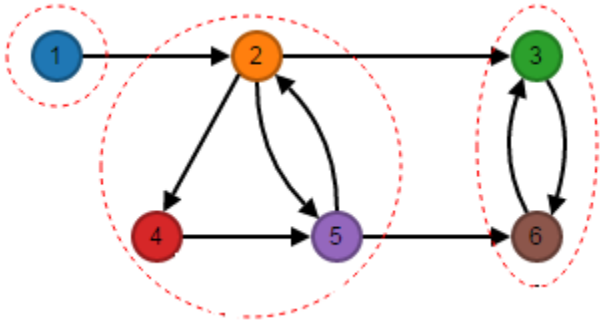
The benefit of the proposed system is that it would support multiple functionalities and may become a service for computer scientists and not only.

As I mentioned above, last year when I tried to solve my problem, but could not because of the computer power limitation. I would be happy to find such service where you could delegate your source code, which would be decomposed and solved using hundreds of JVM's.
It may become a free service which would solve any kind of problem that is possible to divide and conquer.

And one of the most important benefit is the experience I've got while trying to make it work ☺

Test Results

The test are done for particular graph example, where each dotted red shape is one strongly connected component.



Below are listed the test results:

Computing Strongly Connected Components ...
Tarjan 502709.0 nanoseconds
Done ...
SCC: [[5, 2], [4, 3, 1], [0]]
Needs 3 Agents

Search for Agent ...
Found Agents ...
Broadcasting ...
Sending message to Agent 0
Sending message to Agent 1
Sending message to Agent 2

Sending message to Agent 3
A1: got CFP message ...
A3: got CFP message ...
A0: got CFP message ...
A2: got CFP message ...

Got Broadcast Answer from Agent: (agent-identifier :name A3@129.241.123.122:1099/JADE
Requires: 4
Got Broadcast Answer from Agent: (agent-identifier :name A1@129.241.123.122:1099/JADE
Requires: 8
Got Broadcast Answer from Agent: (agent-identifier :name A2@129.241.123.122:1099/JADE
Requires: 2
Got Broadcast Answer from Agent: (agent-identifier :name A0@129.241.123.122:1099/
Requires: 2

Sending reject to: (agent-identifier :name A1@129.241.123.122:1099/JADE
Sending reject to: (agent-identifier :name A3@129.241.123.122:1099/JADE
Sending reject to: (agent-identifier :name A0@129.241.123.122:1099/JADE

A1: got reject message ...
A3: got reject message ...
A2: got accept message …

Sending Accept to: A2
Colored Vertices: 0
Colors : 1 0 0 0 0 0
A2: 404884.0 nanoseconds
A2: problem solving is Done ...

Got Feedback. from: (agent-identifier :name A2@129.241.123.122:1099/JADE
Sub problem is Solved
……..

This is only one iteration, just for one strongly connected component. In our case we have 3 strongly connected components, thus we have 3 such iteration. At the end we are getting our final result as an array of numbers indicating which vertex should take which color. The range of colors are 3, as we have specified in configuration file. The order index of each number indicates the vertex number. For example vertex 1 should take color 1, and vertex 2 should take color 3, and so on. If we again have a look at the graph figure above, we can see that no two adjacent vertices have the same color, which is required in graph coloring problem.

Final result:

Colors : 1 3 2 1 2 1

Note, all the logs are included in the log.txt file (in the log file you can find more information about agents' registration in yellow pages, search for agents, broadcast, accept, reject, solving part and so on).

Challenges

While doing this project, I have encountered many problems, mostly related to Jade framework, such as search for agents, which throws null pointer exception. Turns out its common problem, but there is no working answer for that. Anyway, I could manage it.

Sending the data to the agents at run time required me some creative work to solve. Due to the fact that I did not want to pass some data whenever I create the Agent, I had to search the agent, make broadcast and send the data, get feedback telling how long the agent want to solve it, choosing the minimum time and then accepting that offer and rejecting all the others, and as the problem is solved, get the result. All these things made my system pretty complicated.

Also, it's a bit hard to debug, as you first send message to agent #1, then agent #2, but sometimes the last one gets the message first.
Seems I wanted more than I have been required, but the good side is that I could do more.

Future Enhancements

I made the system quite complicated to satisfy my needs, thus I had to spend more time on coding. For testing I have used only one arbitrary small graph.
That's why I would like to spent more time on testing with pretty large and dense graph. It's not so easy to find appropriate huge graph for testing purposes of particular problem, thus needs more time and investigation.

Also, I would like to use not only computers but also cell phones as an agent to solve my problem. In my mind I had that idea, so the system is ready for that, needs only some work on Agents immigration part.

References

[1] Michael Wooldridge. Multiagent Systems, 2nd ed.
[2] Fabio Bellifemine, Giovanni Caire, Dominicgreenwood, Developing multi-agent systems with Jade, 2007.
[3] Micha Sharir, A strong-connectivity algorithm and its applications in data flow analysis, Computers and Mathematics with Applications 7 (1) (1981) p. 67-72
[4] Robert Tarjan, Depth first search and linear graph algorithms, SIAM Journal on Computing 1 (2) (1972) p. 146-160
[5] H.N. Gabow, Path-based depth-first search for strong and biconnected components, Tech. Report CU-CS-890-99, revised version, Dept. of Computer Science, University of Colorado at Boulder, (2000)
[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, 3th Edition, (2009) p. 614
[7] Robin J. Wilson, Introduction to Graph Theory, 5th Edition, (2010) p. 139
[8] Jon Kleinberg, E´va Tardos, Introduction to Algorithms, (2003) p. 36
[9] http://jade.tilab.com/doc/api/index.html