Guidelines to integrate a DCOP algorithm for RMAS Benchmark

Fabio Maffioletti, Riccardo Reffato

Contents

1 Interface description **1**

2 How to integrate a DCOP algorithm with RMAS Benchmark **3**

3 Algorithms **5**

4 Scenarios **5**

5 Tracked statistics **6**

6 MaxSum **6**

7 Future work **7**

8 Other material **7**

1 Interface description

RMAS Benchmark is a tool written in Java that provides an easy way for the developer of DCOP algorithms to implement and test his own algorithm within the Robocup Rescue Simulator. In this first section we give a brief description of the interface and its components:

- CenterAgent: it represents a fictional agent, which is connected to the kernel and can send and receive messages like a normal agent. Unlike a real agent though, it does not compute its own assignment, but it operates as a "message distributor" between agents, providing a way for the agents to exchange messages without passing through the kernel

(which allows very few message exchanges per cycle). It also sends the final message (the assignment for every agent) to the kernel.

- AssignmentSolver: the AssignmentSolver class is created by the CenterAgent. This solver is aware the whole world, it creates the communication channel, the simulator of the decentralized problem and some utility structures, in addition to keeping track of the metrics for the algorithm. After the end of the simulation it returns the assignment to the CenterAgent.

- AssignmentInterface: it is a general interface which provides a way to create both centralized and decentralized problem simulators. At this time only the decentralized one has been implemented.

- DecentralizedAssignmentSimulator: it implements the AssignmentInterface. It creates a number (specified in the gml map file) of agents, each one of them running the same DCOP algorithm and runs the simulation of the decentralized problem, calling the inizialize, send, receive and improveAssignment methods of each agent for a specified number of iterations (all of the iterations need to be completed by the end of a kernel timestep). In particuar the DecentralizedAssignmentSimulator first calls the initialize method of each agent, then it has a while loop in which it calls in this order: the send method of the ComSimulator for each agent, the receive method of the ComSimulator for each agent and the improveAssignment method for each agent. After the while loop ends, it sets the selected target for each agent.

- ComSimulator: it provides a way for the DCOP agents to communicate without passing through the kernel.

- DecentralAssignment: it is an interface which provides the methods that the programmer's algorithm needs to implement in order to work in the RMASBench framework.

- AbstractMessage: it is a message interface, which the messages exchanged by the agents need to implement in order to work in the RMASBench framework.

- AssignmentMessage: it is a type of message the agents can use to communicate with each other.

- UtilityMatrix: this utility class represents a matrix that contains the local utility between each target and agent in the world model and

provides several useful methods to retrieve agents and targets in the world. The utility is a simple estimation based on the fieryness of the fire (the less the fire is powerful the more the utility increases) and the distance between the agent and the fire.

- Stats: this class writes statistics of the algorithm to a file that can be found in the boot/logs directory of the benchmark. The tracked stats are (for each step): number of burning buildings, number of buildings burnt at least once, number of destroyed buildings, destriyed area, number of violated constraints, CPU usage (number of millisecond to compute the assignment), numer of exchanged messages, exchanged messages in bytes, average NCCC (non-concurrent constraint checks).
- Params: this static class contains some parameters used in the benchmark and in the algorithms

2 How to integrate a DCOP algorithm with RMAS Benchmark

In this section we present a guide to integrate you algorithm with RMAS Benchmark.

1. Create a Java class file that implements the DecentralAssignment interface, which is located in the RSLBench.Assignment package.
2. This class represents a proxy between your implementation of the DCOP algorithm and the RMAS Benchmark (you can put a jar file containing the library you are using in the jars folder)
3. The class needs to implement the methods of the DecentralAssignment interface that we will briefly describe:
   - `inizialize(EntityID agentID, UtilityMatrix utilityM)`: in this method the agents are initialized and configured. It is called before the while loop, so the agents will be recreated at each kernel step. This method does not return.
   - `sendMessages(ComSimulator com)`: this method is called by the DecentralizedAssignmentSimulator, one time for each agent, for each iteration of the while loop. In this method the agent computes and returns to the Simulator a Collection of the messages (AbstractMessage) it needs to send to the other agents to perform the assignment.

- `receiveMessages(Collection<AbstractMessage> messages)`: this method is called by the DecentralizedAssignmentSimulator, one time for each agent, for each iteration of the while loop and it computes the collection of messages passed as a parameter.
- `improveAssignment()`: this method computes the assignment and returns a boolean set to true if the assignment is better than the last one.
- `resetStructures`: the developer needs to implement this method if your algorithm has some structures that need to be reset before the next initialization.
- `getAgentID()`: it returns the EntityID of the agent.
- `getTargetID`: it returns the EntityID of the last assigned target
- `getNccc`: this method computes the non concurrent constraint checks for each cycle of the algorithm. It returns an integer.
- `getNumberOfOtherMessages`: this method returns the number of messages exchanged by the agents while not coordinating (i.e. in Maxsum, the messages exchanged to create the factorgraph). It returns an integer.
- `getDimensionOfOtherMessages`: this method returns the dimension of the messages exchanged by the agents while not coordinating (i.e. in Maxsum, the messages exchanged to create the factorgraph). It returns an integer.

4. If you want to add some parameters specific to your algorithm with which you want to test your algorithm (in particular by testing different values of those parameters), it is convenient if you put them in the Params class following this template: public static ParameterType AlgorithmClassName_ParameterName (optionally followed by a default value). Furthermore you have to insert them in the gui_defaults.cfg file, which is located in the RSLB2/boot directory, optionally followed by a default value. The usage of this method will be explained in the next item.

5. The computation can be easily started by launching the start.jar package located in the RSLB2/boot directory (the command is "java -jar start.jar"). In the GUI you can modify the values of the parameters specified in the gui_defaults.cfg file. You can put different values for each parameter separated by a space, and the benchmarking system will execute all the possible test with the combinations of the parameters with multiple values. For example, if you are testing the DSA

algorithm and want to use different probabilities that DSA will change its previous decision, you can write in the params.cfg file the following line: DSA_CHANGE_VALUE_PROBABILITY: 0.4 0.5 0.6. Some parameters are mandatory for the benchmark to function correctly so they can be modified but not removed, and these do not have the ClassName_ prefix. You can modify the tested algorithm(s) by modifying the assignment_class parameter, the number_of_cycles specifies the number of cycles done before the benchmark makes a decision for each timestep, the experiment_start_time specifies when the agents will start reacting to the fires and the the number_of_considered_targets.

6. You can stop the simulation at any time. If some grphics are available they sill be automatically plotted when you stop the simulation.

7. The stats are stored in the result folder, in the "test" folder.

3 Algorithms

At these time there are several algorithms in the benchmark:

- DSA: in this algorithm the agents take decisions using their local utility, but exchange messages to control whether a building has a greater number of agents than it needs. The best target is then chosen by each agent with a certain probability.

- MaxSum: in this algorithm the agents exchange messages to calculate the best target for each agent considering the assignment of a number of neighbours (the utilities are summed and the maximum is chosen).

4 Scenarios

The sample scenarios of the benchmark are:

- The proposed scenario is a simple fireagents' coordination problem: the fireagents need to extinguish fires which originate from different ignition points in the map. Each firefighter is represented in rmas_bench by a platoon fire agent and it can do 3 types of actions depending on its state:

  - move to a target (building on fire);
  - recharge water pump;
  - shoot water to a building on fire.

5 Tracked statistics

The benchmark keeps track of some statistics of your algorithm and confronts them (possibly) with two states of the art algorithm: DSA and MaxSum. Here is an example of the statistics of the DSA algorithm with the simulation starting at time 50.

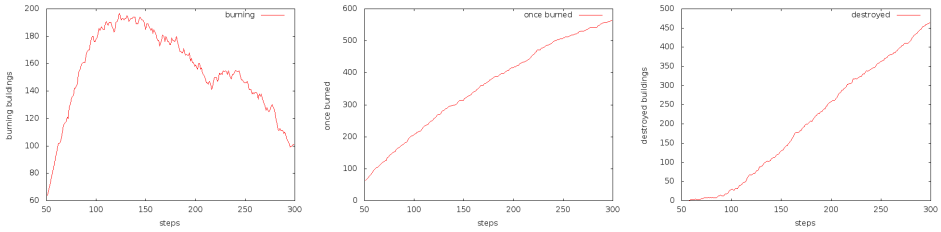

Figure 1: Respectively burning buildings, buildings which burned at least once and destroyed buildings

6 MaxSum

To test maxsum we used the first scenario (described in the corresponding section), and we defined the problem as follows:

- each variable x_i represents a firefighter;
- there is a funcion F_i for each burning building; the arguments of a function are the 5 variables (firefighters) that have the best utility for that function;
- the domain of a variable depends on the functions which the variable is associated to, e.g., if the function F_i depends on the variable x_j, the domain of x_j has f_i (fire $i-th$) as a possible value;

$$D(x_i) = \{f_j | x_i \text{ is argument of the function } F_j\}$$

- each function is calculated as follows:

$$F(x_i \dots x_k) = \sum_{j=i}^{k} U(j, f_i)$$

where $U(j, f_i) = U(j, f_i)$ if $x_j = f_i$, and is 0 otherwise;

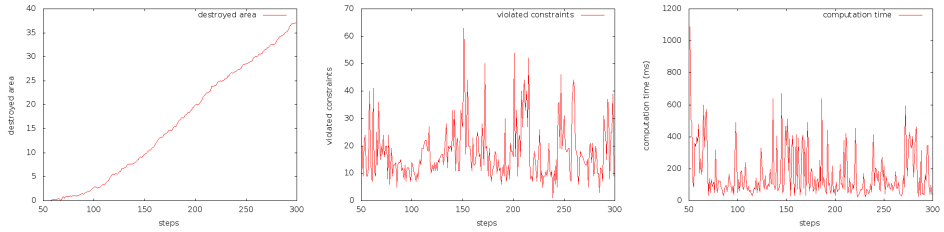

Figure 2: Respectively total destroyed area, the constraints violated and the computation time for each timestep

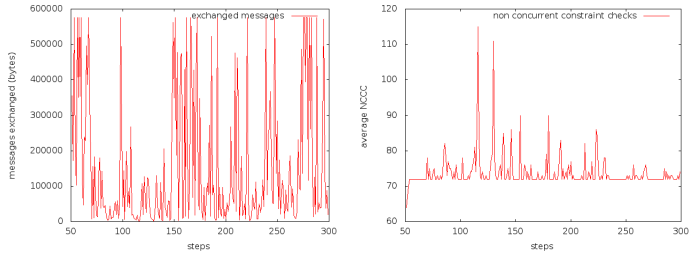

Figure 3: Respectively the messages exchanged between the agents for each timestep and the average NCCC

- in general $U(j, f_i)$ is the utility of an agent j with respect to the fire f_i;
- in MaxSum each agent A_i owns one variable (x_i) and 5 functions, which are the fires with the best utility for the i-th agent.

7 Future work

- Improve UtilityMatrix class
- Modify the way messages' dimensions are measured.
- Memorize stats to a different folder for each test.

8 Other material

- A paper describing the system: Alexander Kleiner, A. Farinelli, S. Ramchurn, B. Shi, F. Maffioletti and R. Reffato. 2013.
  RMASBench: Benchmarking Dynamic Multi-Agent Coordination in Urban Search and Rescue.
  In *Proc. of the 12th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2013).*

- A video showing how the system works: `http://www.youtube.com/watch?v=39y6tkhv5O4`