Table of Contents

Acronyms....2
Document Structure....2
MAJAN Overview....2
MAJAN Installation....3
  MAJAN Plugin....3
  MAJAN Web....3
Using MAJAN....4
  MAJAN Ontology....4
  MAJAN Agent Communication....7
  MAJAN Coordination Nodes....11
  MAJAN Coordinaton Behaviors Trees....15
    Request-Coordination-Protocol BTs....15
    CSGP-Coordination-Protocol Bts....19
    Clustering-Coordination-Protocol BTs....26
MAJAN Postman Collections....30
  Creating Agents....30
  Populating Local Agents Repository....34
  Starting Multi Agent Coordination....36
MAJAN Extension of AJAN Editor....37
  Monitoring Activities of Coordinating Agents....38
  Evaluating Results of Coordination into Groups....40
    MAC Solution....41
    Central Solution....42
    Compare Solutions....50
MAJAN Extra....53
  Useful Tips....53
  Centrally Running Grouping Algorithms....55
  Possible Errors and Their Solutions....55
  Justifications....56

Acronyms

AJAN: Accessible Java Agent Nucleus

AKB: Agent Knowledge Base

BT: Behavior Tree

CMD: Command Prompt

CSGP: Coalition Structure Generation Problem

EKB: Execution Knowledge Base

LAR: Local Agents Repository

MAJAN: Multi Agent AJAN

Document Structure

This document provides the necessary information to use MAJAN. It is structured sequentially starting from the basics, to the creation, execution, and finally evaluation of multiagent coordination use-cases into groups in the AJAN agent engineering tool.

MAJAN Overview

MAJAN is an extension of the agent engineering tool AJAN, which provides features to realize and evaluate SPARQL-BT-based distributed coordination of AJAN agents into groups. In case you are not familiar with AJAN, please refer to the respective wiki sections of GitHub repositories for AJAN Service and AJAN Editor.

AJAN Service is JAVA based execution engine to run intelligent SPARQL-BT-based AJAN agents. AJAN Editor is a web application that provides a user-friendly user interface for functionalities such as designing behaviors of agents as SPARQL-BTs, creating, executing agents, and more.

Since AJAN didn’t support multiagent coordination, MAJAN extends and adopts it appropriately in a way that AJAN users can develop multiagent coordination use cases, execute them and analyze the results of use cases easily by using provided features.

MAJAN provides the features listed below:

1. Template generic coordination BTs to design multiagent coordination use-cases. These BTs cover all the required steps from the beginning until the end of coordination (into groups). Moreover, they (i.e. BTs) are use-case independent, and thus, users can customize them depending on domain-specific use-cases they have at hand.
2. Postman collections to create multiple agents and execute coordination use-cases easily with just one click. These collections cover creating agents, populating their agent repository, and starting coordination processes.
3. Monitoring multiagent coordination activities in AJAN Editor. AJAN already provided logs in Netbeans or CMD depending on where AJAN Service is executed. However, agent names are not logged in Netbeans or CMD and there are huge amounts of logs since all agents log every single BT node execution, unlike MAJAN Monitoring feature.
4. Evaluating grouping results of multiagent coordination use-cases. Since template coordination BTs coordinate agents into groups, it is necessary to be able to analyze the result in a visual and user-friendly format rather than RDF triples in RDF4J repositories. Moreover, MAJAN supports executing centrally running grouping algorithms and visualizing the results. Finally, comparing the grouping solutions of Multi-Agent Coordination and Centrally Running Algorithm is also supported in MAJAN.
5. Finally, MAJAN provides tips, complex SPARQL queries for MAC, and more support in MAJAN Extra section.

MAJAN Installation

MAJAN consists of MAJAN Plugin and MAJAN Web extensions respectively for AJAN Service and AJAN Editor. The following two sections give the instructions to install MAJAN.

MAJAN Plugin

MAJAN Plugin provides BT nodes that are required for multiagent coordination. This plugin is integrated into AJAN Service via its Plugin System. To install AJAN Service with MAJAN Plugin, please check out the “feature/MAJAN” branch in AJAN Service. To install AJAN Service, the required instructions are given in AJAN Service GitHub repository.

MAJAN Web

AJAN Editor is extended with MAJAN Web and to install it, the required instructions are given in AJAN Editor. Please check out the “feature/MAJAN” branch of AJAN Editor GitHub repository. Since MAJAN is integrated into AJAN Editor, there is no instruction to install MAJAN Web.

Using MAJAN

MAJAN Ontology

This section provides all the RDF subjects, predicates, and objects that are being used in MAJAN. The section starts with listing all the RDF objects which are used as rdf:type. Afterward, this list is followed with more detailed information where each of the listed objects is elaborated by describing their respective predicates.

PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

Types that are used to describe information in MAC:

1. mac:**MACProblemInstance** → Every multiagent coordination problem in MAJAN should be described with this type. This type contains all the necessary information about a multiagent coordination problem, starting from its runtime to its solution. Each MACProblemInstance should have a unique ID such that they can be differentiated.
2. mac:**CurrentMACProblemInstance** → This type is used only in EKB, and it is never saved into AKB. The reason for this is to differentiate the currently running MACProblemInstance. If this type was saved into AKB, there would be no way to know which MACProblemInstance is currently running, since there can be multiple of them running at the same time. Moreover, this type should always be used together with MACProblemInstance.
3. mac:**Conversation** → In each MACProblemInstance, there can be multiple conversations among participant agents, and these conversations are described with this type (i.e. mac:Conversation). Each conversation should have a unique ID such that it can be differentiated.
4. mac:**RequestResponse** → Once a conversation starts, everything that agents exchange should be in the type of mac:RequestResponse. According to the FIPA Request protocol, agents can either agree, refuse or send a result for a request and thus, one of mac:RequestResult, mac:RequestRefusal, or mac:RequestAgreement should also be used together with RequestResponse.
    1. mac:**RequestAgreement** → This is a subtype of RequestResponse and is used to represent that the message is the agreement of the request.
    2. mac:**RequestRefusal** → This is a subtype of RequestResponse and is used to represent that the message is the refusal of the request.
    3. mac:**RequestResult** → This is a subtype of RequestResponse and is used to represent that the message is the result of the request.
5. mac:**AgentProfileInfo** → During multiagent coordination (into groups), agents might need to exchange or use profile information (e.g. gender, nationality, etc.) of users they represent, with other agents. For this purpose, agents use AgentProfileInfo type in MAJAN.

6. mac:**AgentPreferences** → During multiagent coordination (into groups), agents might need to exchange or use preferences (e.g. gender preference, nationality preference, etc.) of users they represent, with other agents. For this purpose, agents use AgentPreferences type in MAJAN.
7. mac:**CSGP-CoalitionStructure** → This type is used to describe a Coalition Structure which is found as a solution for a CSGP by a CSGP solver. A Coalition Structure is basically a grouping, and it consists of coalitions.
    1. mac:**CSGP-Coalition** → This type is used to describe coalitions, which are basically groups, and consist of agents.
    2. mac:**UtilityValue** → Coalition Structures have values that are usually the sum of values of their Coalitions. And a coalition value is usually the sum of the Utility values of its member agents, and these utility values are described with the UtilityValue type.
8. mac:**Clustering** → This type is used to describe clustering which is found as a solution for a Clustering problem by a clustering problem solver. Clustering is basically a grouping, and it consists of clusters.
    1. mac:**Cluster** → This type is used to describe clusters which are basically groups, and they consist of agents.
    2. mac:**DistanceScore** → Each clustering algorithm requires distance scores between data points (i.e. agents in this case). This type is used to describe distance scores.
    3. mac:**ReciprocalScore** → Since agents compute distance scores from their own perspective, there are two (most likely different) distance scores for two agents. However, clustering algorithms expect only one distance score between two agents. Therefore, it is necessary to compute one score out of two and this score is described with the ReciprocalScore type.
9. mac:**Log** → In order to be able to monitor the activities of coordinating agents, MAJAN provides a monitoring panel where agents send their logs as HTTP messages. This type is used to describe the log message content to be sent to the monitoring panel, such that there is a common message structure that can be understood by agents and MAJAN.

Predicates and respective values they should take are described below. Note that some types (e.g. mac:Cluster) are skipped to avoid duplication of predicates and objects.

1. **mac:MACProblemInstance**
    1. mac:**hasUseCase** → Name of use case as xsd:string
    2. mac:**hasParticipants** → Participant agent Ids as xsd:string
    3. mac:**hasNotificationNecessary** → “true” or “false” as xsd:string
    4. mac:**hasTimeout** → An xsd:dateTime as timeout
    5. mac:**hasQuorum** → An xsd:integer as quorum
    6. mac:**hasId** → An ID as xsd:string
    7. mac:**hasNumberOfAgents** → A number as xsd:integer
    8. mac:**hasStartTime** → An xsd:dateTime as timeout

9. mac:**hasMinPoints** → A number as xsd:integer which is a required parameter for HDBSCAN algorithm
10. mac:**hasMinClusterSize** → A number as xsd:integer which is a required parameter for HDBSCAN algorithm
11. mac:**hasStatus** → "'*running*", "*completed*" or "*failed*" as xsd:string as status of MAC Process
12. mac:**hasSolution** → Subject IRI of a grouping solution which can be either a mac:CSGP-CoalitionStructure or mac:Clustering type
13. mac:**hasSolver** → Name of solver algorithm as xsd:string
14. mac:**hasFeasibleCoalition** → IRI of coalition, which is feasible given ML and CL constraints for this MACProblemInstance.

2. mac:Conversation

1. mac:**hasMacProblemId** → ID of MAC problem this conversation belongs to
2. mac:**hasInitiator** → ID of initiator agent
3. mac:**hasContent** → Subject IRIs to be sent as the content of message among agents
4. mac:**hasReceiverCapability** → Capability of message receiver agents to be used in agent communication
5. mac:**hasReceiver** → ID of message receiver agent to be used in agent communication
6. mac:**hasAgreement** → "true" or "false" to set whether agent agrees or refuses to Request

3. mac:RequestResponse, mac:RequestAgreement, mac:RequestRefusal, mac:RequestResult

1. mac:**hasConversationId** → ID of conversation this RequestResponse belongs to

4. mac:CSGP-CoalitionStructure, mac:Clustering

1. mac:**hasValue** → Value of either CSGP-CoalitionStructure, Clustering, Coalition or anything else
2. mac:**hasRank** → Rank of a CSGP-CoalitionStructure since CSGP Solvers produce rank list of solutions
3. mac:**hasSolutionOf** → ID of MAC Problem this solution belongs to
4. mac:**hasMembers** → Subject IRIs of groups (i.e. either CSGP-Coalition or Cluster)

5. mac:Cluster

1. mac:**isClusterOf** → Subject IRI of a clustering this cluster belongs to

6. mac:DistanceScore, mac:ReciprocalScore, mac:UtilityValue

1. mac:**isComputedBy** → ID of agent who computes this score (e.g. agent1 computes score by comparing itself with agent2 where agent2 is described with isComputedAgainst predicate)
2. mac:**isComputedAgainst** → ID of an agent who this score is computed against
3. mac:**isComputedFor** → ID of MAC problem this score belongs to

7. mac:Log

1. mac:**hasAgentId** → ID of agent who sends the log
2. mac:**hasActivity** → Plain text describing the activity of log
3. mac:**hasActivityStatus** → Status of activity as String

MAJAN Agent Communication

Message (figure 1) and Broadcast (figure 2) BT nodes are provided to achieve communication among agents in multiagent coordination. While the Message node supports sending a message to only one receiver, the Broadcast node supports sending a message to multiple receivers.

Figure 1. Message BT Node in AJAN Editor.

Figure 2. Broadcast BT Node in AJAN Editor.

In order to let agents understand each other when they communicate, there is a predefined structure of message that they should exchange.

1. **Sending a Request to agents**
   One of the predefined template broadcast nodes, MAJAN provides is MessageToParticipants (figure 3). This node is used when an agent sends a request to other agent(s). In other words, whenever an agent wants to start a conversation, this message structure and template node are used. Moreover, this node expects a subject IRI in the type of mac:Conversation and this subject IRI should include predicates and respective values as listed below:
   1. To be sent in message payload: mac:**hasInitiator**, mac:**hasId**, mac:**hasNotificationNecessary**, mac:**hasTimeout**, mac:**hasUseCase**, mac:**hasMacProblemId**, mac:**hasContent**
      1. SPARQL query to construct message payload is described in figure 4.
   2. Necessary to identify receivers of message**:** mac:**hasReceiver,** mac:**hasReceiverCapability**
      1. SPARQL query to select respective receiver URIs is described in figure 5.
   3. Refer to MAJAN Ontology section for more detailed information about the definition of predicates.

Figure 3. MessageToParticipants predefined broadcast node in AJAN Editor.

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
    ?bnode  rdf:type    mac:Conversation ;
            mac:hasInitiator    ?thisAgentId ;
            mac:hasId   ?conversationId ;
            mac:hasNotificationNecessary    ?notifNecessary ;
            mac:hasTimeout  ?timeout ;
            mac:hasUseCase  ?useCaseTitle ;
            mac:hasMacProblemId     ?macId ;
            mac:hasContent  ?requestContent .
    ?requestContent ?predicate  ?object .
}
WHERE {
    ?bnode  rdf:type    mac:Conversation ;
            mac:hasId   ?conversationId ;
            mac:hasUseCase  ?useCaseTitle ;
            mac:hasNotificationNecessary    ?notifNecessary ;
            mac:hasTimeout  ?timeout ;
            mac:hasMacProblemId     ?macId .
    OPTIONAL {
        ?bnode  mac:hasContent  ?requestContent .
        ?requestContent ?predicate  ?object .
    }
    ?thisAgent  rdf:type    ajan:Agent, ajan:ThisAgent ;
                ajan:agentId    ?thisAgentId .
    }
```

Figure 4. Query to construct MessageToParticipants node payload

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

SELECT DISTINCT ?requestURI
WHERE {
        ?bnode  rdf:type    mac:Conversation ;
                mac:hasParticipants ?participantId ;
                mac:hasReceiverCapability   ?capability .

        ?thisAgent  rdf:type    ajan:Agent, ajan:ThisAgent ;
                    ajan:agentId    ?thisAgentId .

        FILTER(?participantId != ?thisAgentId)
        ?partAgentUri   ajan:agentId    ?participantId ;
                        ajan:hasAddress ?address .
        BIND(CONCAT(?address, "?capability=", ?capability) AS ?requestURI)
}
```

Figure 5. Query to select URIs of receiver agents in MessageToParticipants node

Once a conversation has been started with MessageToParticipants node, participant agents should respond to this request accordingly. For this purpose, agents use mac:RequestResponse and its subtypes.

a) Sending an Agreement as the response of Request

In order to agree to the request, MAJAN provides AgreedMessageToInitiator predefined message node (figure 6). This node sends a response back to the initiator agent telling that the participant agrees to the respective request. In order to build the payload of this message, SPARQL query in figure 7 is used. To identify the request URI for AgreedMessageToInitiator message node, agents use the SPARQL query in figure 8.

Figure 6. AgreedMessageToInitiator predefined message node in AJAN Editor.

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
    ?newNode    rdf:type mac:RequestAgreement, mac:RequestResponse ;
                mac:hasUseCase  ?useCase ;
                mac:hasId   ?conversationId ;
                mac:hasMacProblemId     ?macId ;
                mac:hasParticipants     ?thisAgentId .
}
WHERE {
    ?bnode  rdf:type    mac:Conversation ;
            mac:hasUseCase  ?useCase ;
            mac:hasMacProblemId     ?macId ;
            mac:hasId   ?conversationId .

    ?thisAgent  rdf:type    ajan:Agent, ajan:ThisAgent ;
                ajan:agentId    ?thisAgentId .
    BIND(SHA1(xsd:string(NOW())) AS ?uniqueId)
    BIND( IRI(CONCAT(STR(mac:RequestAgreement), STR(?uniqueId))) AS ?newNode )
}
```

Figure 7. Query to construct AgreedMessageToInitiator node payload

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

SELECT DISTINCT ?requestURI
WHERE {
        ?bnode  rdf:type    mac:Conversation ;
                mac:hasInitiator    ?initiatorId ;
                mac:hasReceiverCapability ?initiatorCapability.

        ?initAgentUri   ajan:agentId    ?initAgentId ;
                        ajan:hasAddress ?address .
        BIND(CONCAT(?address, "?capability=", ?initiatorCapability) AS ?requestURI)
}
```

Figure 8. Query to select URI of receiver agent in AgreedMessageToInitiator and RefusedMessageToInitiator nodes

b) Sending a Refusal as the response of Request

In order to refuse the request, MAJAN provides RefusedMessageToInitiator predefined message node (figure 9). This node sends a response back to the initiator agent telling that the participant refuses the respective request. In order to build the payload of this message, SPARQL query in

figure 10 is used. To identify the request URI for RefusedMessageToInitiator message node, agents use the SPARQL query in figure 8.

Figure 9. RefusedMessageToInitiator predefined message node in AJAN Editor.

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {

    ?newNode    rdf:type mac:RequestRefusal, mac:RequestResponse ;
                mac:hasConversationId   ?conversationId ;
                mac:hasUseCase  ?useCase ;
                mac:hasMacProblemId     ?macId ;
                mac:hasParticipants     ?thisAgentId .
}
WHERE {
    ?bnode  rdf:type    mac:Conversation ;
            mac:hasUseCase  ?useCase ;
            mac:hasMacProblemId     ?macId ;
            mac:hasId   ?conversationId .

    ?thisAgent  rdf:type    ajan:Agent, ajan:ThisAgent ;
                ajan:agentId    ?thisAgentId .
    BIND(SHA1(xsd:string(NOW())) AS ?uniqueId)
    BIND( IRI(CONCAT(STR(mac:RequestRefusal), STR(?uniqueId))) AS ?newNode )
    }
```

Figure 10. Query to construct RefusedMessageToInitiator node payload

c) Sending a Result as the response of Request

In order to send a result to the request, MAJAN provides SendResultMessage predefined broadcast node (figure 11). This node sends a result back to the initiator agent. In order to build the payload of this message, SPARQL query in figure 12 is used. To identify the request URI for SendResultMessage broadcast node, agents use the SPARQL query in figure 13.

Since it is necessary to design this node in a generic, use-case-independent way, domain-specific results should be attached to the mac:hasContent predicate. This way, there will be no need to modify predefined communication nodes. As can be seen from figure 12, it is enough to attach the subject IRI and the query will retrieve all related predicates and objects automatically and send them as the result of the request.

Figure 11. SendResultMessage predefined broadcast node in AJAN Editor.

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

CONSTRUCT {
    ?newNode    rdf:type    mac:RequestResult, mac:RequestResponse ;
                mac:hasId   ?convId ;
                mac:hasParticipants     ?thisAgentId ;
                mac:hasMacProblemId     ?macId ;
                mac:hasUseCase  ?useCase ;
                mac:hasContent  ?resultContent .
    ?resultContent  ?predicate  ?object .
}
WHERE {
    ?bnode  rdf:type    mac:Conversation ;
            mac:hasMacProblemId     ?macId ;
            mac:hasUseCase  ?useCase ;
            mac:hasId   ?convId ;
            mac:hasContent  ?resultContent .
    ?resultContent  ?predicate  ?object .
    ?thisAgent  rdf:type    ajan:Agent, ajan:ThisAgent ;
                ajan:agentId    ?thisAgentId .
    {
        BIND(SHA1(xsd:string(NOW())) AS ?uniqueId)
        BIND( IRI(CONCAT(STR(mac:RequestResult), STR(?uniqueId))) AS ?newNode )
    }}
```

Figure 12. Query to construct SendResultMessage node payload

```
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

SELECT DISTINCT ?requestURI
WHERE {
        ?bnode  rdf:type    mac:Conversation ;
                mac:hasReceiver ?receiverId ;
                mac:hasReceiverCapability ?receiverCapability.

        ?receiverAgentIRI   ajan:agentId    ?receiverId ;
                            ajan:hasAddress ?address .
        BIND(CONCAT(?address, "?capability=", ?receiverCapability) AS ?requestURI)
}
```

Figure 13. Query to select URI of receiver agent in SendResultMessage node

MAJAN Coordination Nodes

In order to support coordinating AJAN agents into groups, MAJAN provides 5 BT nodes as listed below:

1. Broadcast node (figure 2) allows agents to send the same message to multiple agents, unlike the existing Message node.

2. Coalition Generator node (figure 14) is used to generate coalitions to solve CSGP with a selected solver algorithm. In order to make this node generic, it expects input as described in the SPARQL query in figure 15.

Figure 14. Coalition Generator node in AJAN Editor.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

Construct {
    ?macInstance    rdf:type    mac:MACProblemInstance ;
                    mac:hasId   ?macId ;
                    mac:hasParticipants ?participantId ;
                    mac:hasNumberOfAgents   ?numOfAgents ;
                    mac:hasMinCoalitionSize ?minSize ;
                    mac:hasMaxCoalitionSize ?numOfAgents ;
                    mac:hasCannotLinkConnections    ?cannotLinkBnode ;
                    mac:hasMustLinkConnections  ?mustLinkBnode .

    ?cannotLinkBnode    mac:hasCannotConnect    ?clAgentId, ?clAgentId2 .
    ?mustLinkBnode  mac:hasMustConnect  ?mlAgentId, ?mlAgentId2 .
}
Where{
    ?macInstance    rdf:type    mac:MACProblemInstance ;
                    mac:hasId   ?macId ;
                    mac:hasMinCoalitionSize ?minSize ;
                    mac:hasParticipants     ?participantId ;
                    mac:hasNumberOfAgents   ?numOfAgents .
    OPTIONAL {
        ?macInstance    mac:hasCannotLinkConnections    ?cannotLinkBnode .

        ?cannotLinkBnode    mac:hasCannotConnect    ?clAgentId, ?clAgentId2 .
    }
    OPTIONAL {
        ?bnode  mac:hasMustLinkConnections  ?mustLinkBnode .

        ?mustLinkBnode  mac:hasMustConnect  ?mlAgentId, ?mlAgentId2 .
    }
}
```

Figure 15. Query to construct input for Coalition Generator node

As the output, this node produces coalitions in the type of mac:CSGP-Coalition and attaches them to the given mac:MACProblemInstance subject IRI.

1) BOSS node (figure 16) is used to solve CSGP [csgp-survey] with the BOSS algorithm [boss], which finds an exact solution. This node expects generic input as described in figure 17. As the output, it attaches solutions (i.e. Coalition Structure) to the given MACProblemInstance's mac:hasSolution predicate.

Figure 16. BOSS node in AJAN Editor.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

Construct {
    ?macInstance    rdf:type    mac:MACProblemInstance ;
                    mac:hasNumberOfAgents   ?numberOfAgents ;
                    mac:hasParticipants ?participantId ;
                    mac:hasId   ?macId ;
                    mac:hasFeasibleCoalitions   ?feasibleCoalition .

    # FeasibleCoalitions only include the coalitions which are feasible for this use-case (
    because of constraints). Therefore, feasibleCoalitions don't include all possible
    coalitions which is required by BOSS algorithm. Therefore, we pass a value as
    "NonExistentCoalitionValue" which will be assigned to the missing coalitions. By
    default, this value is very small since infeasibleCoalitions are infeasible and we don't
    want them to be part of the solution.
    ?macInstance    mac:hasNonExistentCoalitionValue    "-1000000" .

    ?feasibleCoalition  rdf:type    mac:CSGP-Coalition ;
                        mac:hasMembers  ?memberAgentId ;
                        mac:hasValue ?coalitionValue .
}
Where{
    ?macInstance    rdf:type    mac:MACProblemInstance, mac:CurrentMACProblemInstance ;
                    mac:hasNumberOfAgents   ?numberOfAgents ;
                    mac:hasParticipants     ?participantId ;
                    mac:hasId   ?macId ;
                    mac:hasFeasibleCoalitions   ?feasibleCoalition .
    ?feasibleCoalition  rdf:type    mac:CSGP-Coalition ;
                        mac:hasMembers  ?memberAgentId ;
                        mac:hasValue ?coalitionValue .
}
```

Figure 17. Query to construct input for BOSS node

2) HDBSCAN node (figure 18) is used to solve Clustering Problem with the HDBSCAN algorithm [hdbscan]. This node expects generic input as described in figure 19. As the output, it attaches solutions (i.e. clustering) to the given MACProblemInstance's mac:hasSolution predicate.

Figure 18. HDBSCAN node in AJAN Editor.

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
CONSTRUCT {
        ?macInstance    rdf:type    mac:MACProblemInstance;
                        mac:hasId       ?macId;
                        mac:hasNumberOfAgents   ?numberOfAgents;
                        mac:hasParticipants     ?participantId1, ?participantId2 ;
                        mac:hasPerfectMatchScore ?perfectMatchScore ;
                        mac:hasCannotLinkConnections    ?cannotConnection ;
                        mac:hasMustLinkConnections  ?mustConnection .
        ?cannotConnection   ?clPred     ?clObj .
        ?mustConnection     ?mlPred     ?mlObj .
        # HDBSCAN Parameters: min Points and min Cluster Size
        ?macInstance    mac:hasMinPoints    ?boundMinPoints ;
                        mac:hasMinClusterSize   ?boundMinClSize .
        ?rrsIri     rdf:type    mac:DistanceScore ;
                    mac:isComputedBy    ?participantId1 ;
                    mac:isComputedAgainst   ?participantId2 ;
                    mac:isComputedFor   ?macId ;
                    mac:hasValue    ?reciprocalDistance .}
WHERE {
    ?macInstance    rdf:type    mac:MACProblemInstance, mac:CurrentMACProblemInstance ;
                    mac:hasId   ?macId ;
                    mac:hasNumberOfAgents   ?numberOfAgents ;
                    mac:hasParticipants     ?participantId1, ?participantId2 .
    OPTIONAL{
        ?macInstance    mac:hasCannotLinkConnections    ?cannotConnection .
        ?cannotConnection   ?clPred     ?clObj . }
    OPTIONAL{
        ?macInstance    mac:hasMustLinkConnections  ?mustConnection .
        ?mustConnection     ?mlPred     ?mlObj . }
    ?macInstance    mac:hasReciprocalScore  ?rrsIri .
    ?rrsIri     rdf:type    mac:ReciprocalScore ;
                mac:isComputedBy    ?participantId1 ;
                mac:isComputedAgainst   ?participantId2 ;
                mac:isComputedFor   ?macId ;
                mac:hasValue    ?reciprocalDistance .
    OPTIONAL {
        ?macInstance    mac:hasMinPoints    ?minPoints ;
                        mac:hasMinClusterSize   ?minClSize .
        BIND(IF(BOUND(?minPoints), ?minPoints, 1) AS ?boundMinPoints)
        BIND(IF(BOUND(?minClSize), ?minClSize, 2) AS ?boundMinClSize) }
    BIND(0 AS ?perfectMatchScore) }
```

Figure 19. Query to construct input for HDBSCAN node

3) Insert node's (figure 20) goal is to write given triples to the specified repository, just like Write node. However, Write node replaces the triples in the repository if there exists any triple with the same predicate, while Insert node doesn't replace and inserts the triples whether there is already the same predicate.

Let's say the fisrt triple exist in AKB and we want to write the second triple to AKB.

i. ajan:SubjectIRI ajan:hasPredicate "object" .

ii. ajan:SubjectIRI ajan:hasPredicate "object2" .

Below are the final triple(s) in AKB if we were to use Write or Insert nodes.

a) **Write**:

ajan:SubjectIRI ajan:hasPredicate “object2” .

b) **Insert**:

ajan:SubjectIRI ajan:hasPredicate “object”, “object2” .

Figure 20. Insert node in AJAN Editor.

MAJAN Coordinaton Behaviors Trees

MAJAN provides template coordination BTs for users to be able to design MAC use-cases easily. There are 3 different sets of BTs for 3 coordination protocols. The first one is designed based on the FIPA Request protocol. In this protocol, agents coordinate with each other and exchange basic information. The second protocol is designed to solve CSGP to create groups of agents. This protocol is developed by extending the first protocol. The last protocol is designed to solve Clustering problems, and it is also developed based on the first one. All of these protocols are generic, use-case independent, such that users can customize them based on the use case at hand. Moreover, in these protocols, CSGP and Clustering are solved with BOSS and HDBSCAN algorithms, respectively. However, it is also possible to use other CSGP and Clustering solver algorithms by only replacing the one within the respective BTs. These protocols are explained in the following subsections.

Request-Coordination-Protocol BTs

In this protocol, one agent requests profile information from other agents and waits until either quorum or timeout is reached. Three BTs are developed to run this protocol:

1. Name: **SendCoordRequestTempBT**, Label: “Send Coord. Request Temp. BT” in figure 21

In this BT, once the agent handles the respective event, it updates the mac:MACProblemInstance to make sure that it has a unique subject IRI and ID such that it is not being mixed with any other MACProblemInstance that already exists in AKB. Then agent logs this activity to MAJAN Monitoring panel (figure 22)

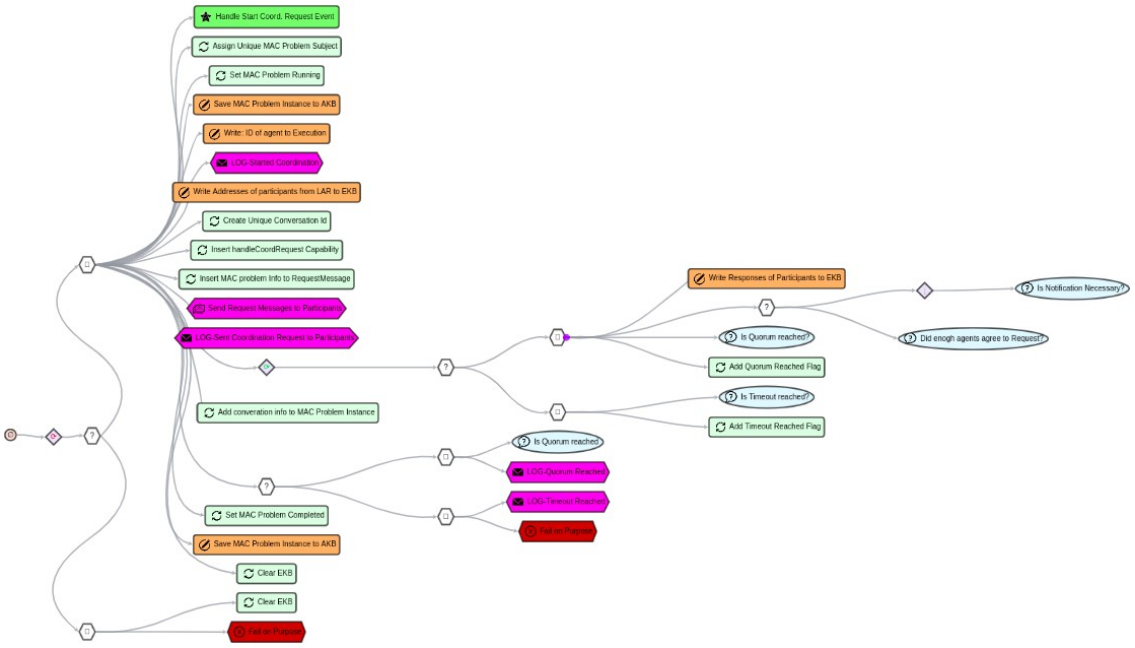


Figure 21. Send Coord. Request Temp. BT in AJAN Editor.

Figure 22. Send Coord. Request Temp. BT part 1.

In the second part of this BT, agent creates a unique conversation and sends the request messages to participant agents (figure 23).

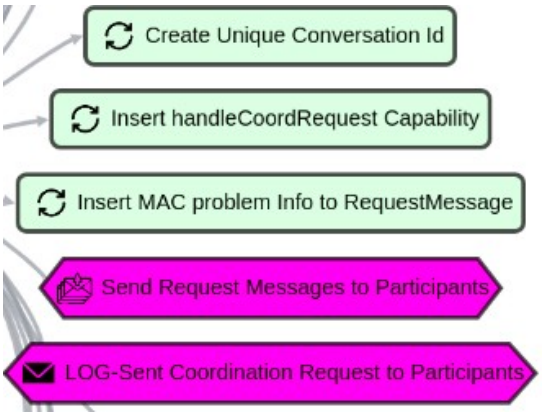


Figure 23. Send Coord. Request Temp. BT part 2

In the third part of this BT, agent waits until quorum or timeout reached (figure 24).

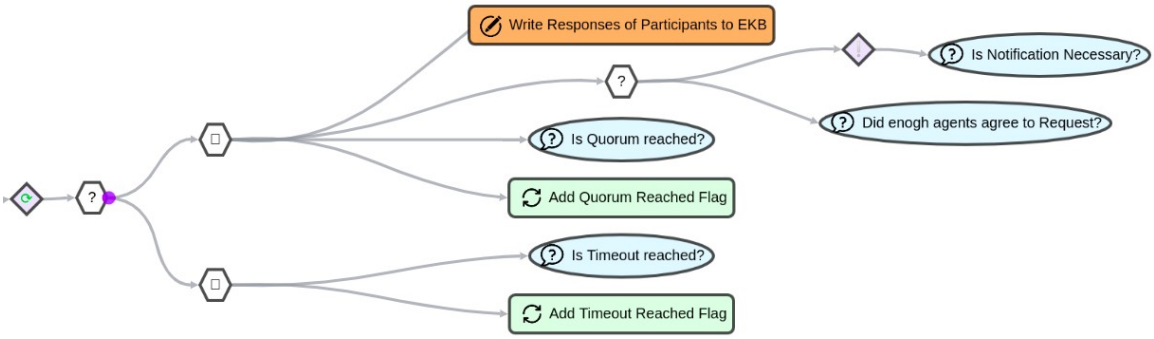


Figure 24. Send Coord. Request Temp. BT part 3

Finally, agent logs whether quorum or timeout reached to Monitoring panel and saves the MACProblemInstance to AKB (figure 25).

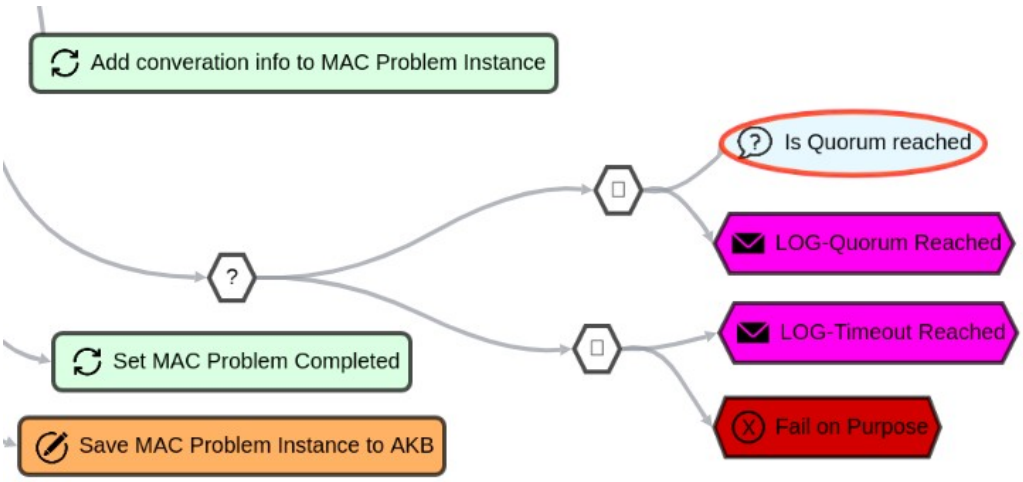


Figure 25. Send Coord. Request Temp. BT part 4

a) Name: **HandleCoordRequestTempBT**, Label: “Handle Coord. Request Templ. BT”

In this BT, once the agent handles the respective event, it logs this activity to MAJAN Monitoring panel. Then it waits for 3 seconds, which is a placeholder where users can add anything they want. Once waiting is over, it needs to send an agreement or refusal as a response. After replying appropriately, it needs to send a result as a response if the timeout is not reached yet to finalize handling this request. To compute the necessary results, users can replace the Wait node as they like, since it is a placeholder. (figure 26)

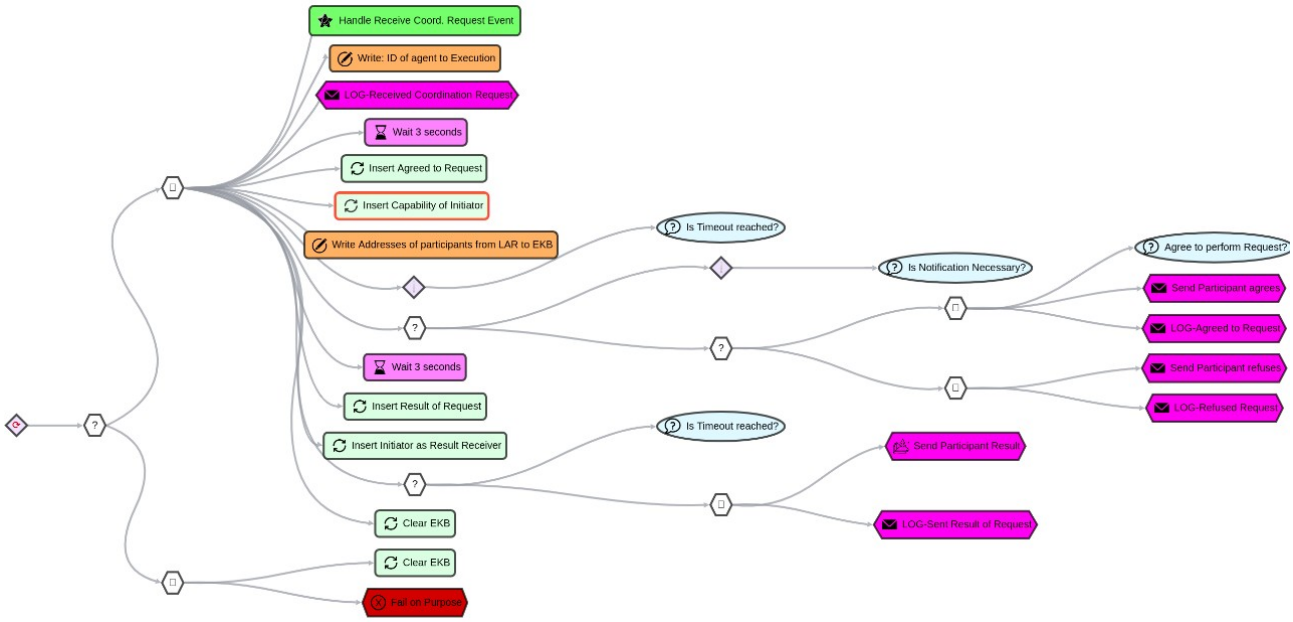


Figure 26. Handle Coord. Request Templ. BT

b) Name: **ReceiveCoordRequestResponseTempBT**, Label: “Receive Coord. Request Response Temp. BT”

This BT is used to receive a mac:RequestResponse and save it to AKB such that the responses of participant agents can be used than in other coordination BTs.

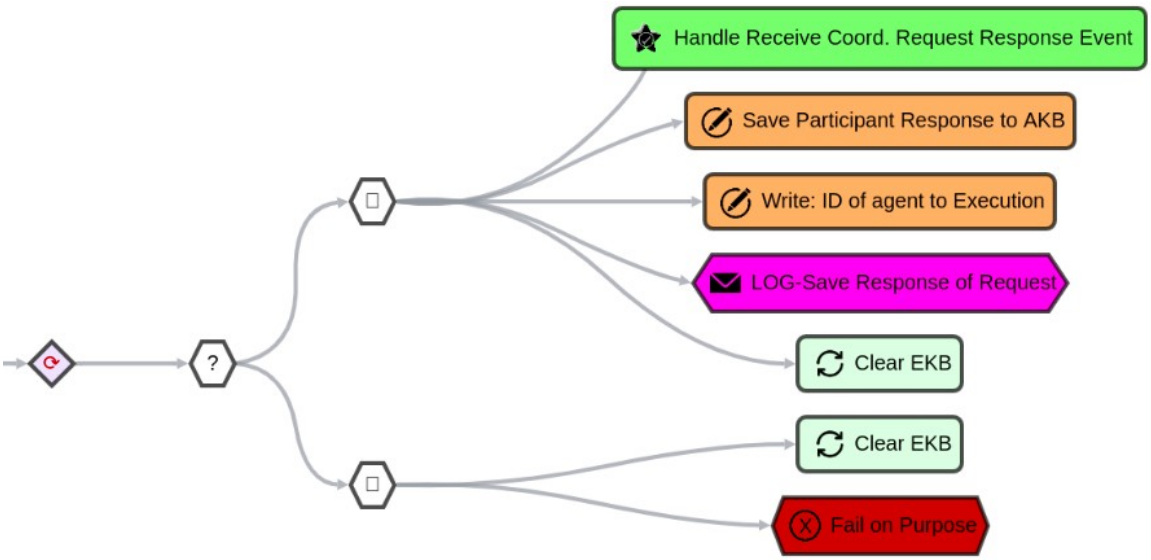


Figure 27. Receive Coord. Request Response Temp. BT

CSGP-Coordination-Protocol Bts

In CSGP, agents are divided into coalitions (i.e. groups) in a way that social welfare of the system is maximized. Each single agent should compute Utility values for the coalitions it is member of.

Utility values represent how much agents want to be in certain coalitions. Thus, the higher utility value, the more agent prefers the coalition. Then those utility values are summed to compute coalition values. Then coalition values are summed to compute coalition structure (i.e. grouping) values. Finally, a CSGP solver algorithm computes solutions (i.e. rank list of coalition structure with highest value).

The goal of this protocol is to form groups of agents by solving CSGP. BTs for this protocol are designed by extending Request Protocol BTs. In this protocol, one agent (who is also called a Dedicated agent) receives “start coordination” signal, and it then requests profile information from participant agents. Any agent can act as a Dedicated agent. Once the required information is collected, the dedicated agent generates coalitions by using CoalitionGenerator node. Then it broadcasts generated coalitions and profile information of all agents such that participants can compute Utility values and return them back to the Dedicated agent. Once all Utility values are collected, the dedicated agent builds CSGP [csgp-survey] and runs the BOSS [boss] node to solve it which computes a rank list of solutions. And finally, the solutions are stored in AKB and broadcast to other participant agents.

There are 7 BTs to run CSGP-Coordination-protocol as listed below. Please, open Behaviors section of AJAN Editor and write the names of BTs in search section to analyse them.

1) Name: **SendCsgpCoordRequestTempBt**, Label: "Send CSGP Coord. Request Temp. BT"

This BT is a customized version of “Send Coord. Request Temp. BT”. The difference in this BT is that the agent requests Agent Profile Information from other agents, since this info is required to solve CSGP. Once required info is collected or timeout reached, the agent produces “Compute Csgp Coalitions Event” (figure 30) which triggers "Compute CSGP Coalitions Templ. BT". In order to request Agent Profile Info, the agent inserts “agentProfileInfoRequest” (figure 28, 29) as the capability of participant agents, and this capability triggers "Handle Agent Profile Info Request Temp. BT".

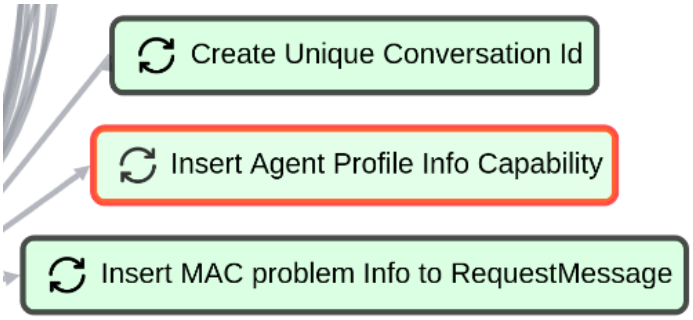


Figure 28. Insert Agent Profile Info to Conversation node

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>

DELETE {
    ?subject    mac:hasReceiverCapability   ?existingCapability .
}
INSERT{
    ?subject    mac:hasReceiverCapability   'agentProfileInfoRequest' .
}
WHERE{
    ?subject    rdf:type    mac:Conversation .

    OPTIONAL {
        ?subject mac:hasReceiverCapability  ?existingCapability .
    }
}
```

Figure 29. Insert Agent Profile Info to Conversation node's SPARQL query

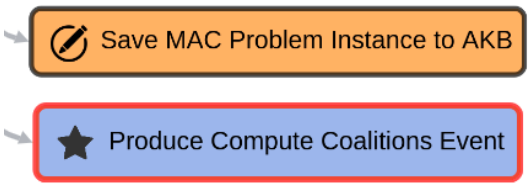


Figure 30. Produce Compute Coalitions Event node

2) Name: **HandleAgentProfileInfoRequestTempBT**, Label: "Handle Agent Profile Info Request Temp. BT"

This BT is a customized version of "Handle Coord. Request Templ. BT". In this BT, the agent attaches its Profile Information to the Conversation (figure 31) and BT automatically sends this info to the Dedicated agent.

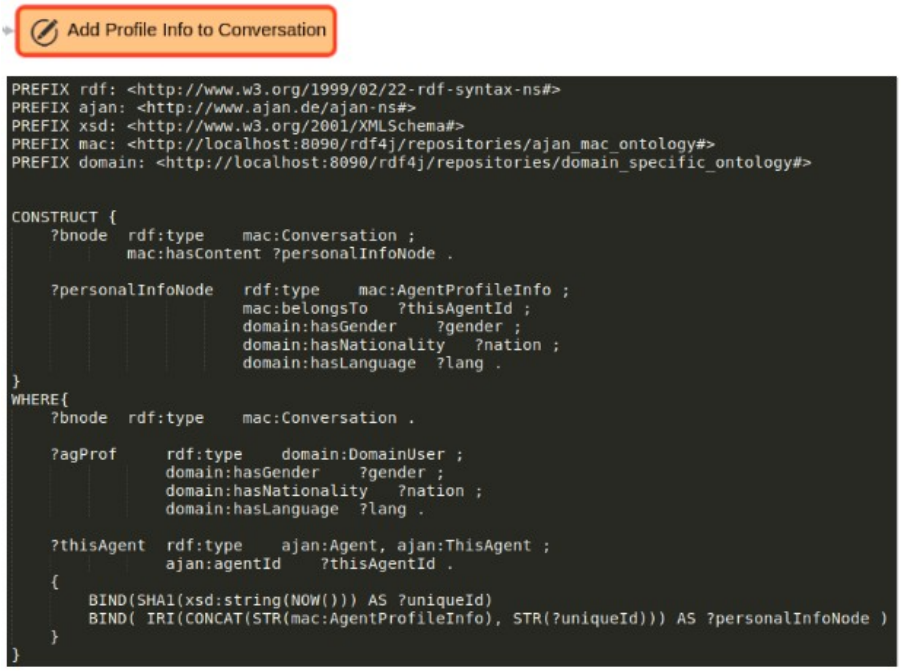


Figure 31. Adding agent's profile information to Conversation to be sent as the result of request

3) Name: **ComputeCsgpCoalitionsTemplBT**, Label: "Compute CSGP Coalitions Templ. BT"

This BT is designed to compute coalitions given constraints, broadcast them to other agents to collect their utility values, and compute coalition values before producing "Solve CSGP Coordination Event" which triggers "Solve CSGP Template BT". Firstly, the agent computes cannot link connections which are later passed to Coalition Generator node as input (figure 32). Once coalitions are computed, the agent broadcasts computed coalitions and profile information of all agents to participants such that they can compute their utility values for computed coalitions (figure 33). After receiving utility values, the agent computes coalition values and produces "Solve CSGP Coordination Event" (figure 34).

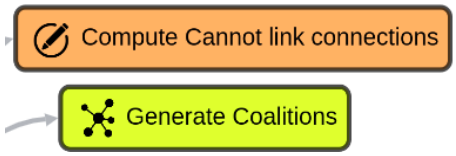


Figure 32. Compute Cannot link connections for given use case and Generate coalitions given cannot link connections.

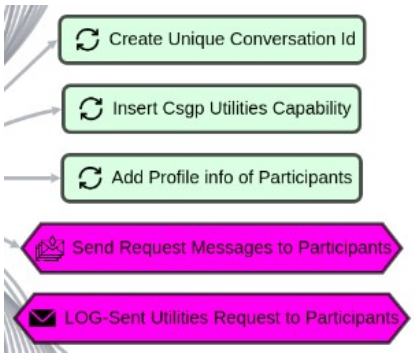


Figure 33. Broadcast coalitions and agent profile info to collect utility values

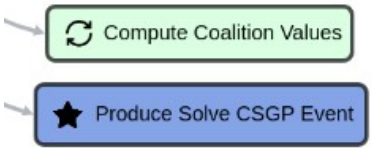


Figure 34. Compute coalition values and trigger solving CSGP BT

4) Name: **HandleCsgpUtilitiesRequestTemplBT**, Label: "Handle Csgp Utilities Request Templ. BT"

This BT is designed to compute utility values upon request and return them back to the dedicated agent (figure 35-a). SPARQL query in figure 35-b is used to compute utility values to solve CSGP.

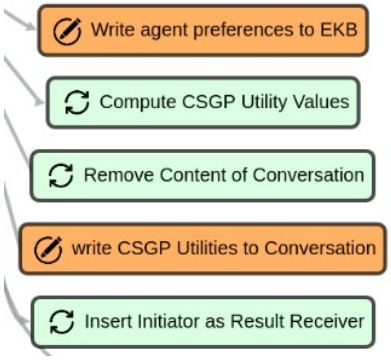


Figure 35-a. Compute Utility values after fetching preferences of this agent and finally attach the utility values to the conversation content to be sent back to dedicated agent.

```
INSERT {
      ?feasibleCoalitionNode  mac:hasUtilityValue ?uVbnode .
      ?uVbnode    rdf:type    mac:UtilityValue ;
                  mac:isComputedBy   ?thisAgentId ;
                  mac:hasValue       ?ttlV .   }
WHERE { {
    SELECT DISTINCT ?feasibleCoalitionNode ?thisAgentId (SUM(?uValue) AS ?ttlV)
    WHERE {
      # get This agent
      ?agent  rdf:type  ajan:Agent, ajan:ThisAgent ;
            ajan:agentId   ?thisAgentId .
      # get Feasible Coalitions which are sent by Dedicated agent
      ?macInstance  rdf:type   mac:MACProblemInstance ;
              mac:hasId ?macId ;
              mac:hasFeasibleCoalitions  ?feasibleCoalitionNode .
      # get the coalition information
      ?feasibleCoalitionNode  mac:hasCommonGender ?commonGender ;            mac:hasCommonNation ?commonNation .
      # make sure that coalition contains This agent
      FILTER EXISTS { ?feasibleCoalitionNode  mac:hasMembers  ?thisAgentId . }
      # get preferences of This agent
      ?prefsSbj rdf:type  mac:AgentPreferences ;
          domain:hasGenderPreference  ?genderPref ;            domain:hasNationPreference  ?nationPref ;
          domain:hasGenderPrefWeight  ?genPrefWeight ;           domain:hasNationPrefWeight  ?natPrefWeight .
      # get gender and nation of This agent
      ?resultSbj  rdf:type  mac:Conversation ;
          mac:hasMacProblemId   ?macId ;
          mac:hasContent  ?resultContent .
      ?resultContent  rdf:type  mac:AgentProfileInfo ;
              mac:belongsTo   ?thisAgentId ;
              domain:hasGender  ?gender ;
              domain:hasNationality   ?nation .
      BIND(IF(LCASE(?genderPref) = "dont mind" || (LCASE(?genderPref) = "same" && ?gender = ?commonGender) || (
      LCASE(?genderPref) = "mixed" && LCASE(?commonGender) = "mixed"), xsd:float(?genPrefWeight), -xsd:float(?genPrefWeight))
      AS ?genderUValue)
      BIND(IF(LCASE(?nationPref) = "dont mind" || (LCASE(?nationPref) = "same" && ?nation = ?commonNation) || (
      LCASE(?nationPref) = "mixed" && LCASE(?commonNation) = "mixed"), xsd:float(?natPrefWeight), -xsd:float(?natPrefWeight))
      AS ?nationUValue)
      BIND((?genderUValue + ?nationUValue) AS ?uValue)
    } GROUP BY ?feasibleCoalitionNode ?thisAgentId  }
  BIND(BNODE() AS ?uVbnode) }
```

Figure 35-b. SPARQL Query to compute Utility values in CSGP Protocol

5) Name: **SolveCsgpTemplBT**, Label: "Solve CSGP Template BT"

   This BT's goal is to solve CSGP by using BOSS node (figure 36). Users can replace BOSS node with any other CSGP solver node without any problem since the BT is designed in a generic way.

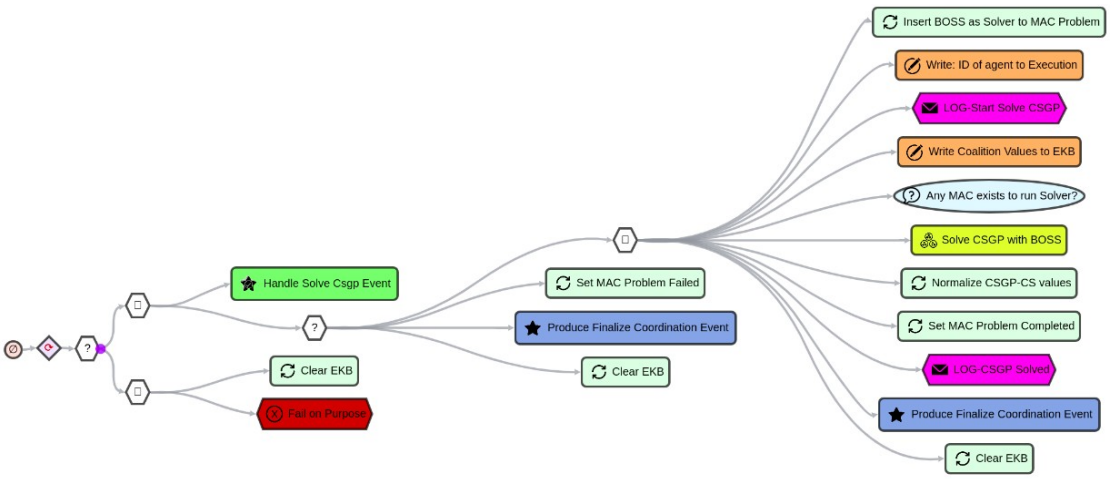


Figure 36. BT to solve CSGP with BOSS algorithm and start finalizing the coordination process.

Additionally, users can normalize the values of Coalition Structures (with Normalize CSGP-CS values node, figure 37) in case they are negative but this is optional. Finally, the agent produces "Finalize Coordination Event" which triggers "Finalize Coordination to Groups BT".

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
DELETE{
    ?macSolution    mac:hasValue    ?csValue .
}
INSERT {
    ?macSolution    mac:hasValue    ?normalizedCsValue .
}
WHERE{
    ?macInstance    rdf:type    mac:MACProblemInstance .

    OPTIONAL{
        ?macInstance    mac:hasSolution     ?macSolution ;
                        mac:hasMinCsValue   ?minCsValue .
        FILTER(?minCsValue <= 0)
        ?macSolution    rdf:type    mac:CSGP-CoalitionStructure ;
                        mac:hasValue    ?csValue .
        BIND((?csValue + 5 - ?minCsValue) AS ?normalizedCsValue)
    }
}
```

Figure 37. SPARQL query to normalize Coalition Structure values.

6) Name: **FinalizeCoordinationToGroupsBT**, Label: "Finalize Coordination to Groups BT"

This BT, as described in figure 38, is designed to finalize any coordination process. It is enough to produce the required event successfully. In this BT, the agent computes the runtime of the coordination process and broadcasts the result of the coordination process to participant agents. By using “Just Success Node”, the agent ensures that broadcasting the result to all participants is optional since some agents might be inactive, but this shouldn’t cause the coordination process to fail. Nevertheless, users can remove Success node and make sure that it is necessary for all participant agents to receive coordination results.

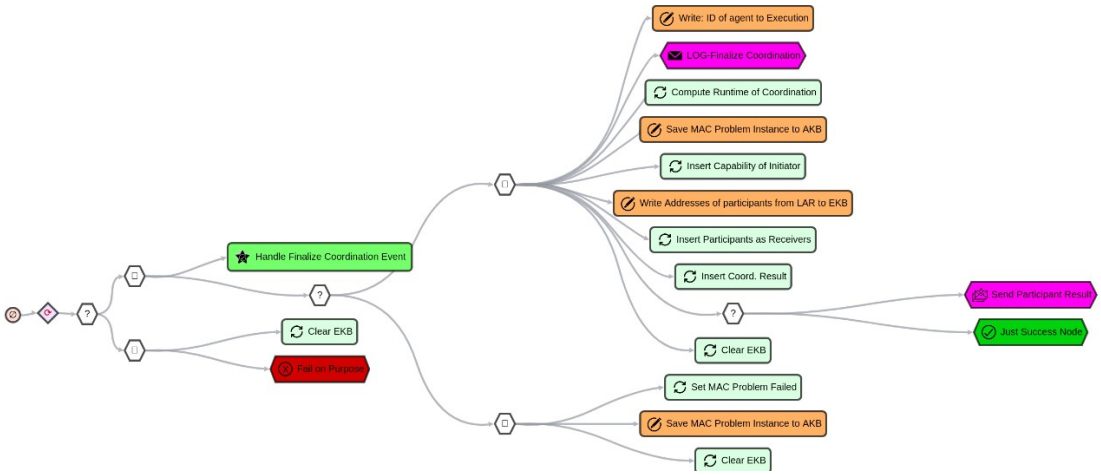


Figure 38. BT to finalize coordination processes.

7) Name: **ReceiveCoordRequestResponseTempBT**, Label: “Receive Coord. Request Response Temp. BT”

In order to save the responses in coordination protocols, agents use the same BT in CSGP protocol as they use in Request protocol. Refer to Request Protocol section for more info about this BT.

Clustering-Coordination-Protocol BTs

The goal of this protocol is to form groups of agents by solving the Clustering problem. This protocol BTs are designed by extending Request Protocol BTs. In this protocol, one agent (who is also called a Dedicated agent) receives “start coordination” signal, and it then requests profile information from participant agents. Any agent can act as a Dedicated agent. Once the required information is collected, the dedicated agent broadcasts profile information of all agents such that participants can compute Distance scores and return them back to the Dedicated agent. Once all Distance scores are collected, the dedicated agent computes reciprocal scores and runs the HDBSCAN node to solve it, which computes a solution. And finally, the solution is stored in AKB and broadcast to other participant agents.

There are 7 BTs to run Clustering-Coordination-protocol as listed below. Please, open Behaviors section of AJAN Editor and write the names of BTs in search section to analyse them.

1) Name: **SendClusteringCoordRequestTempBt**, Label: "Send Clustering Coord. Request Templ BT"

This BT (figure 39) is designed to start the coordination process to solve a clustering problem to form groups of agents. It is based on “Send Coord. Request Temp. BT” just like "Send CSGP Coord. Request Temp. BT". Since profile info of agents is required for clustering problems as well, agent requests them just like in "Send CSGP Coord. Request Temp. BT". The only difference between “Send Clustering…” and “Send CSGP” BTs is the event that is produced at the end of BT. In clustering, the agent produces “Collect Clustering Distances Event” which triggers "Collect Clustering Distances Templ. BT".

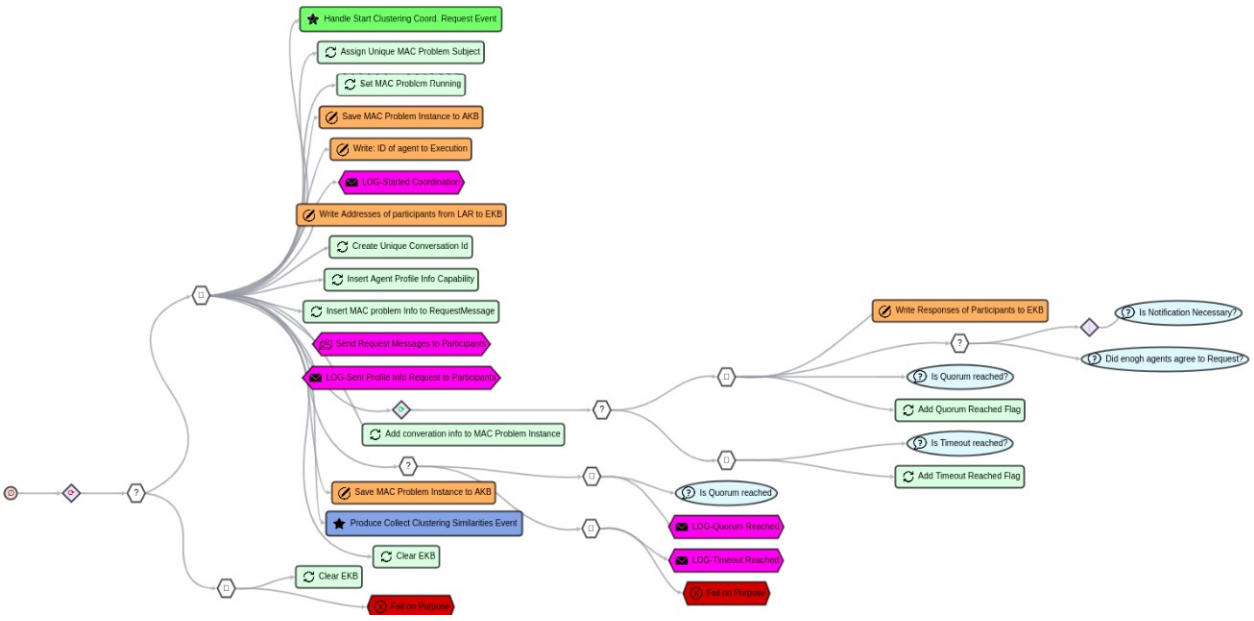


Figure 39. Send Clustering Coord. Request Templ BT in AJAN Editor

2) Name: **HandleAgentProfileInfoRequestTempBT**, Label: "Handle Agent Profile Info Request Temp. BT"

In order to collect profile information from participants, agents in clustering protocol use the same BT that they use in CSGP protocol. Refer to CSGP Protocol BTs section for more info about this BT.

3) Name: **CollectClusteringDistancesTemplBt**, Label: "Collect Clustering Distances Templ. BT"

In this BT (figure 40, 41), the dedicated agent broadcasts the profile info of all agents to participants such that they can compute distance scores and return them to the dedicated agent. Upon receiving this request, the participant agent executes "Handle Clustering Distances Request Templ. BT". Afterward, the dedicated agent produces “Solve Clustering Event” which triggers "Solve Clustering Templ. BT".

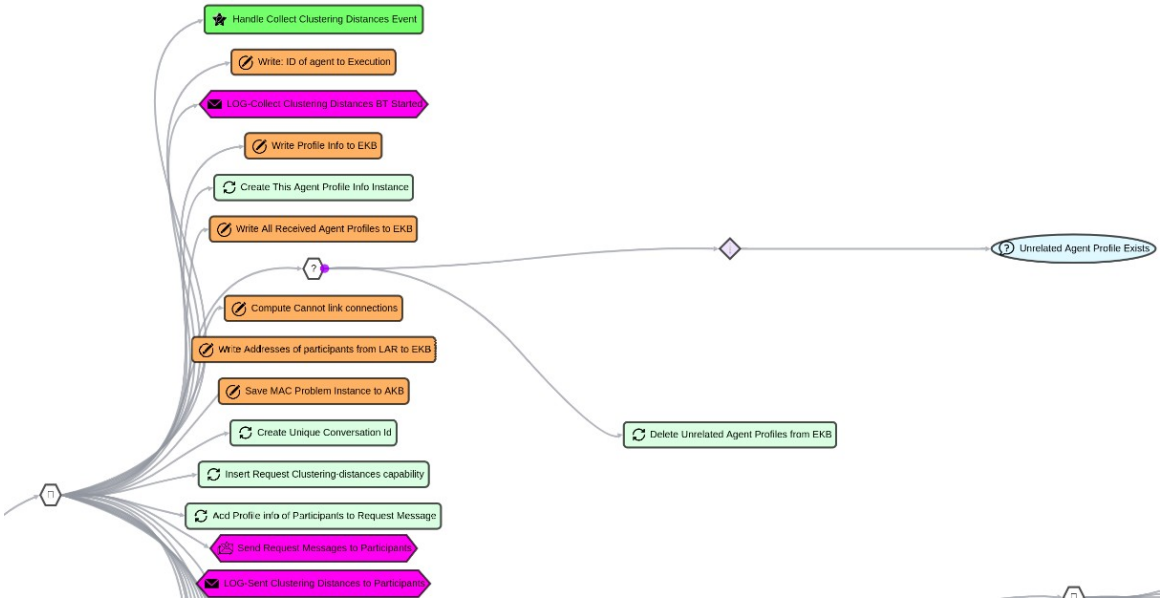


Figure 40. Collect Clustering Distances Templ. BT Part 1 in AJAN Editor

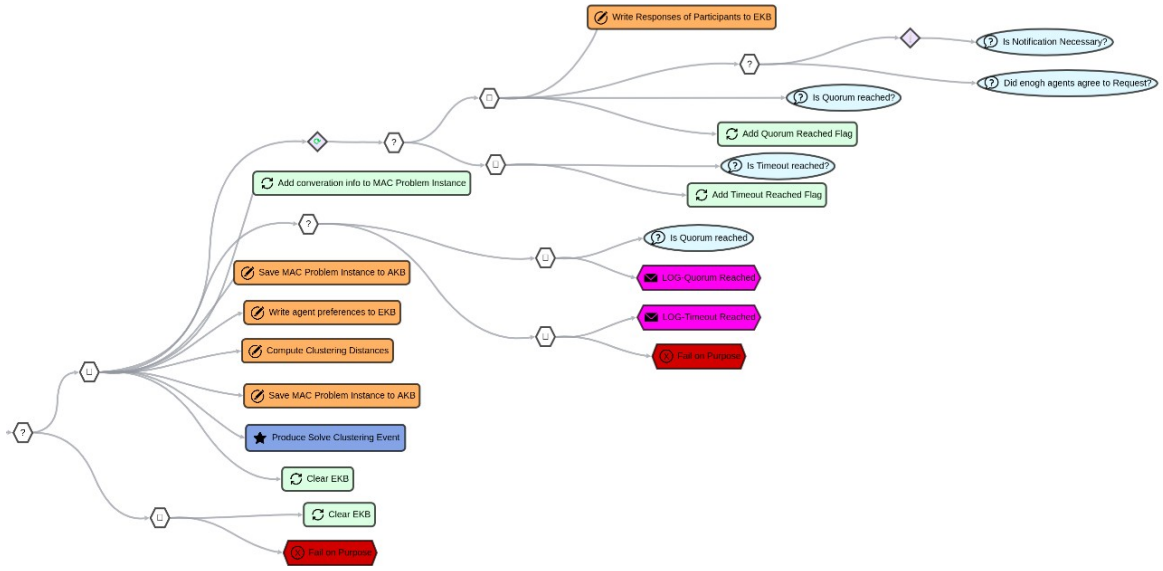


Figure 41. Collect Clustering Distances Templ. BT Part 2 in AJAN Editor

4) Name: **HandleClusteringDistancesRequestTemplBT**, Label: "Handle Clustering Distances Request Templ. BT"

This BT (figure 42-a) is executed when participant agents are requested to compute their distance scores in clustering problem. Agents firstly reply with an Agreement or Refusal. Afterward, agents fetch their preferences from AKB and compute distance scores by comparing the profile information of every single participant agent with their individual preferences. Once they compute distance scores, they send scores back to the dedicated agent.

Figure 42-b describes the SPARQL query of “Compute Clustering Distances” node to compute distance scores.

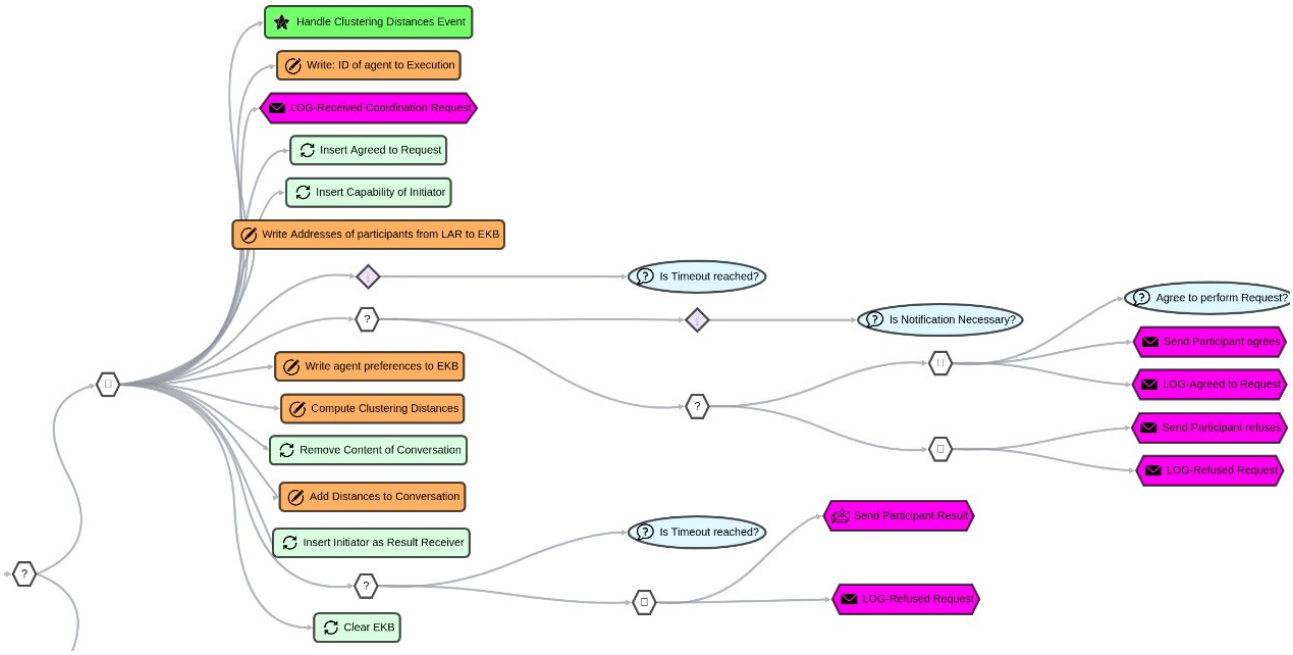


Figure 42-a. Handle Clustering Distances Request Templ. BT in AJAN Editor

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX ajan: <http://www.ajan.de/ajan-ns#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX mac: <http://localhost:8090/rdf4j/repositories/ajan_mac_ontology#>
PREFIX domain: <http://localhost:8090/rdf4j/repositories/domain_specific_ontology#>
CONSTRUCT  {
        ?macInstance  mac:hasDistanceScore  ?distanceScoreIri .
        ?distanceScoreIri   rdf:type  mac:DistanceScore ;
                mac:hasValue  ?minTotalDistance ;
                mac:isComputedBy  ?thisAgentId ;
                mac:isComputedAgainst ?participantId ;
                mac:isComputedFor   ?macId .  }
WHERE {   {
    SELECT ?macInstance ?thisAgentId ?participantId ?macId (MIN(?totalDistance) AS ?minTotalDistance)
    WHERE {    {
          SELECT ?macInstance ?thisAgentId ?participantId  ?totalDistance ?macId
          WHERE {
           ### retrieve this agent
           ?thisAgentIRI   rdf:type  ajan:Agent, ajan:ThisAgent ;
                 ajan:agentId  ?thisAgentId .
           ### retrieve Preferences of this agent
           ?thisAgentPrefs   rdf:type  mac:AgentPreferences ;
                  domain:hasGenderPreference  ?thisGenderPref ;
                  domain:hasNationPreference  ?thisNationPref ;
                  domain:hasGenderPrefWeight  ?thisGenPrefWeight ;
                  domain:hasNationPrefWeight  ?thisNatPrefWeight .
           ### retrieve participant agent id
           ?macInstance  rdf:type  mac:MACProblemInstance ;
                 mac:hasId   ?macId ;
                 mac:hasParticipants   ?participantId .
           ### rule out agent computing similarity with itself
           FILTER(?thisAgentId != ?participantId)
           ### retrieve Personal Info of this agent
           ?conversation   rdf:type  mac:Conversation .
           ?conversation   mac:hasContent  ?thisAgentProfile, ?partAgentProfile.
           ?thisAgentProfile   rdf:type  mac:AgentProfileInfo ;
                  mac:belongsTo   ?thisAgentId ;
                  domain:hasGender  ?thisGender ;
                  domain:hasNationality   ?thisNation .
           ### retrieve Profile Info of the participant agent(s)
           ?partAgentProfile   rdf:type  mac:AgentProfileInfo ;
                  mac:belongsTo   ?participantId ;
                  domain:hasGender  ?participantGender ;
                  domain:hasNationality   ?participantNation .
           ### set config values
           BIND(0 AS ?matchScore)                BIND(2 AS ?unmatchScore)
           ### compute distance value between this and participant agents
           ### Gender preference
           BIND(IF(LCASE(?thisGenderPref) = "dont mind" || LCASE(?thisGenderPref) = "don't mind"
|| (LCASE(?thisGenderPref) = "same" && ?thisGender = ?participantGender) ||
(LCASE(?thisGenderPref) = "mixed" && ?thisGender != ?participantGender),
?matchScore, ?unmatchScore) AS ?genderDistance)
           ### Nationality preference
           BIND(IF(LCASE(?thisNationPref) = "dont mind" || LCASE(?thisNationPref) = "don't mind"
|| (LCASE(?thisNationPref) = "same" && ?thisNation = ?participantNation) ||
(LCASE(?thisNationPref) = "mixed" && ?thisNation != ?participantNation),
?matchScore, ?unmatchScore)
AS ?nationDistance)
           ### total distance
           BIND((?genderDistance + ?nationDistance) AS ?totalDistance)
          } GROUP BY ?macInstance ?participantId  ?totalDistance ?thisAgentId ?macId }
       } GROUP BY ?macInstance ?participantId ?thisAgentId ?macId }
     BIND(BNODE() AS ?distanceScoreIri) }
```

Figure 42-b. SPARQL query of “Compute Clustering Distances” node

5) Name: **SolveClusteringTemplBT**, Label: "Solve Clustering Templ. BT"

The objective of this BT is to solve a clustering problem with the HDBSCAN algorithm. Just like in “Solve CSGP Template BT”, the solver (i.e. HDBSCAN) can be replaced with any other clustering solver algorithm. In this BT, once the dedicated agent computes the solution for the clustering problem with the HDBSCAN algorithm, it produces “Finalize Coordination Event” which triggers "Finalize Coordination to Groups BT".

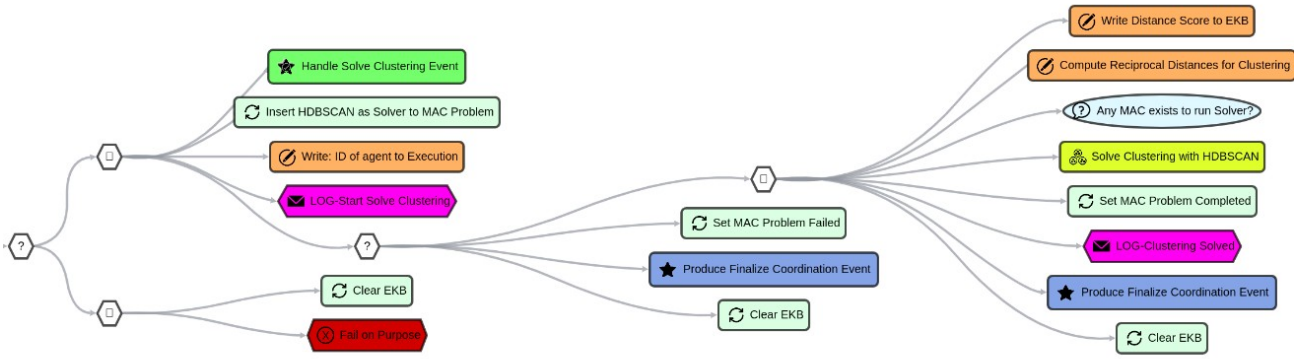


Figure 43. Solve Clustering Templ. BT in AJAN Editor

6) Name: **FinalizeCoordinationToGroupsBT**, Label: "Finalize Coordination to Groups BT"

In order to finalize a coordination process, agents in clustering protocol use the same BT that they use in CSGP protocol. Refer to CSGP Protocol BTs section for more info about this BT.

7) Name: **ReceiveCoordRequestResponseTempBT**, Label: “Receive Coord. Request Response Temp. BT”

In order to save the responses in coordination protocols, agents use the same BT in CSGP protocol as they use in Request protocol. Refer to Request Protocol section for more info about this BT.

MAJAN Postman Collections

Postman is a platform with lots of features including sending HTTP requests such as Get, Post, Put, Delete, etc. AJAN agents can be created, and executed by sending Post requests to AJAN Service. In order to support users to be able to create multiple agents, populate their agent repositories and start coordination protocols with one click, MAJAN provides postman collections which are explained in the following sections.

Creating Agents

Once coordination behaviors of agents are designed, the next step is to create agents to run a multiagent coordination use case. AJAN Editor supports creating agents one by one in Instances subsection of Agents section as shown in figure 44.

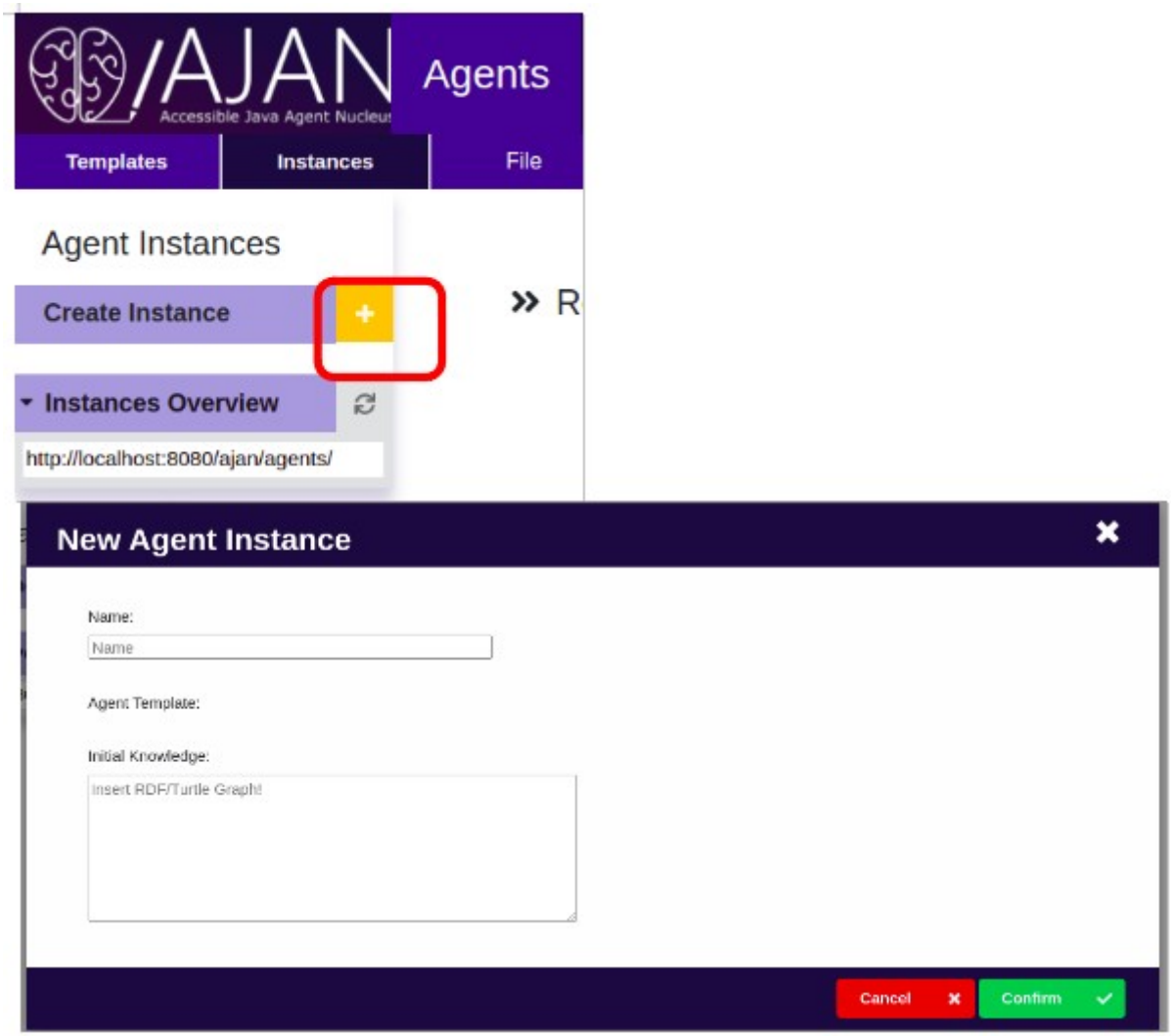


Figure 44. Create Agent in AJAN Editor

However, AJAN Editor doesn't support creating multiple agents at the same time. Therefore, MAJAN provides a Postman Collection, called "MAJAN - Create Agents.postman_collection.json" consisting of 10 Postman Requests to create 10 agents with one click. It is provided in "MAJAN/Postman Collections/Create Agents" (https://github.com/AkbarKazimov/MAJAN) folder.

Firstly, this collection must be imported into Postman by selecting Collections tab and clicking on Import button at the top right corner as shown in figure 45-a.

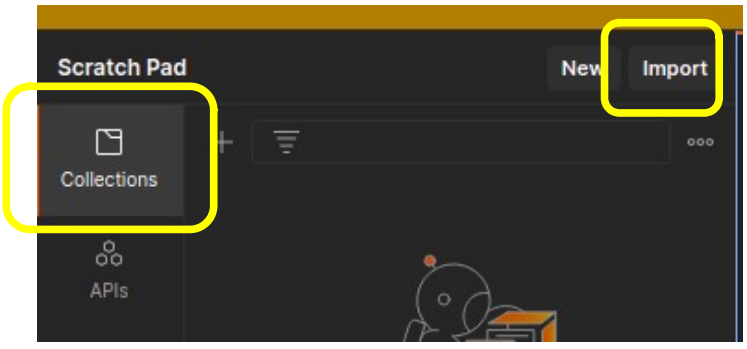


Figure 45-a. Import collection into Postman

Once Import button is clicked, figure 45-b will appear where it is asked to upload the collection. After uploading the collection, we should see it in Collections section as shown in figure 45-c.

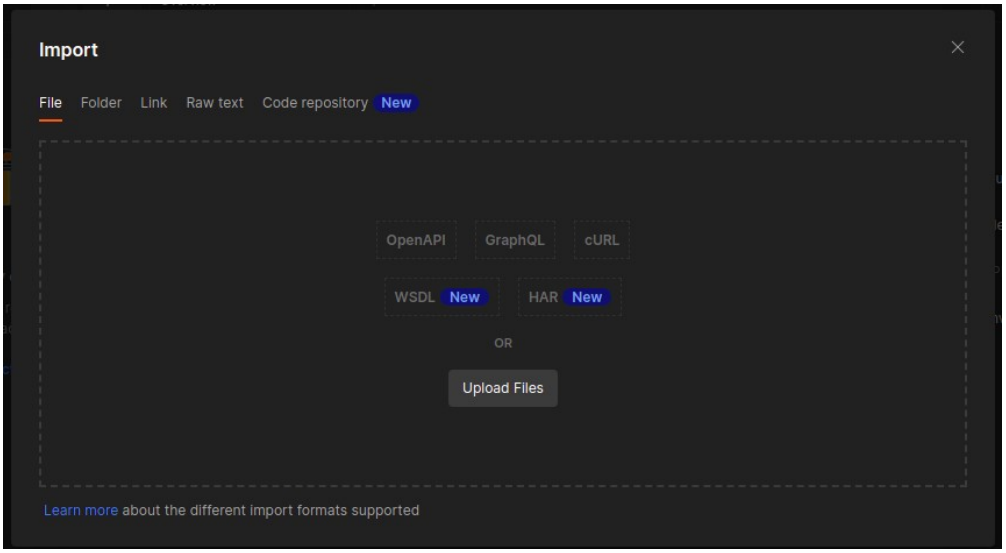


Figure 45-b. Upload collection to Postman

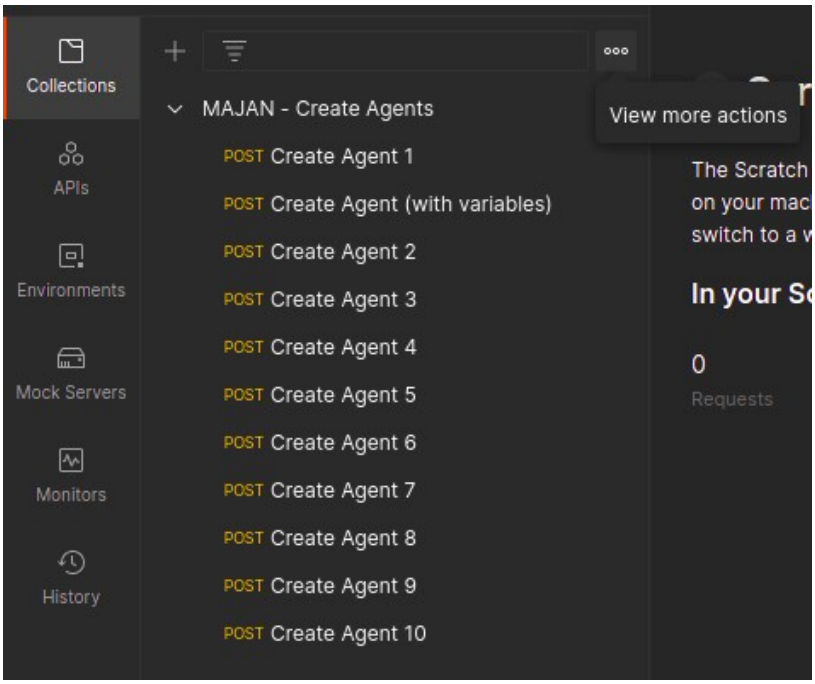


Figure 45-c. Collection is available in Postman

Since the collection is available in postman, we can run it now. To do so, click on the three dots near “View more actions” text. Then click on the “Run collection” button in the opened list, which will open a view as shown in figure 45-d.

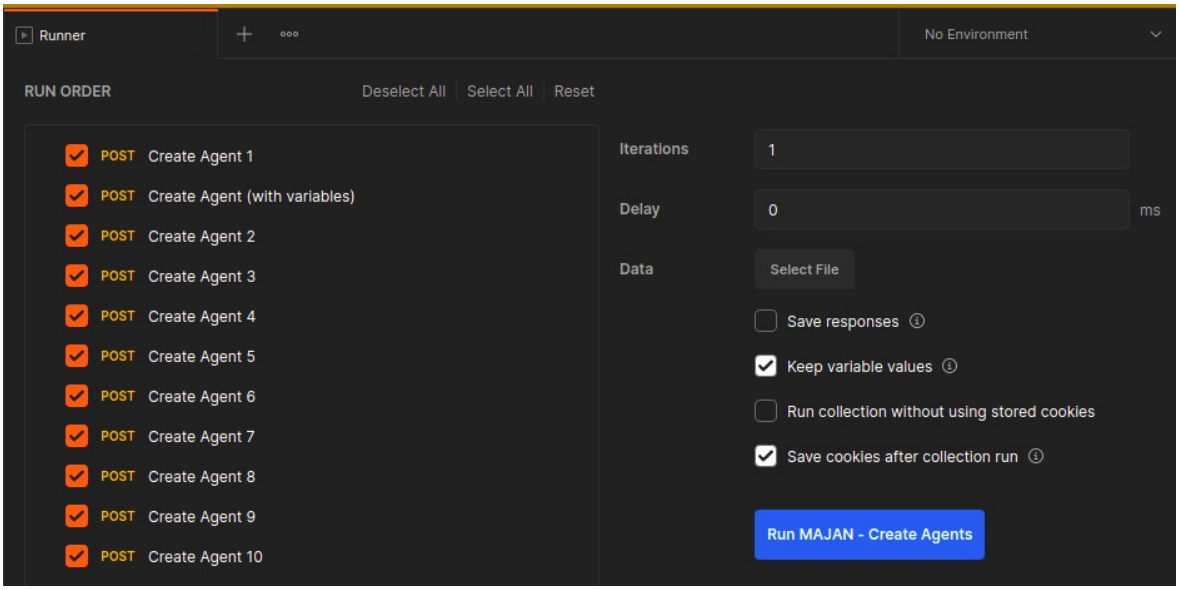


Figure 45-d. Content of collection to run.

Now we can select the requests (i.e. agents) that we want to create. Then we need to select a configuration file in "Data → Select File" section. The configuration file is provided in the same folder as the collection file, and it consists of values for IP and Port of AJAN Service to create respective agents. The only step that is left to create the selected agents is to click Run MAJAN – Create Agents button in blue. As an example, we select and create 3 agents as shown in figures 45-e and 45-f.

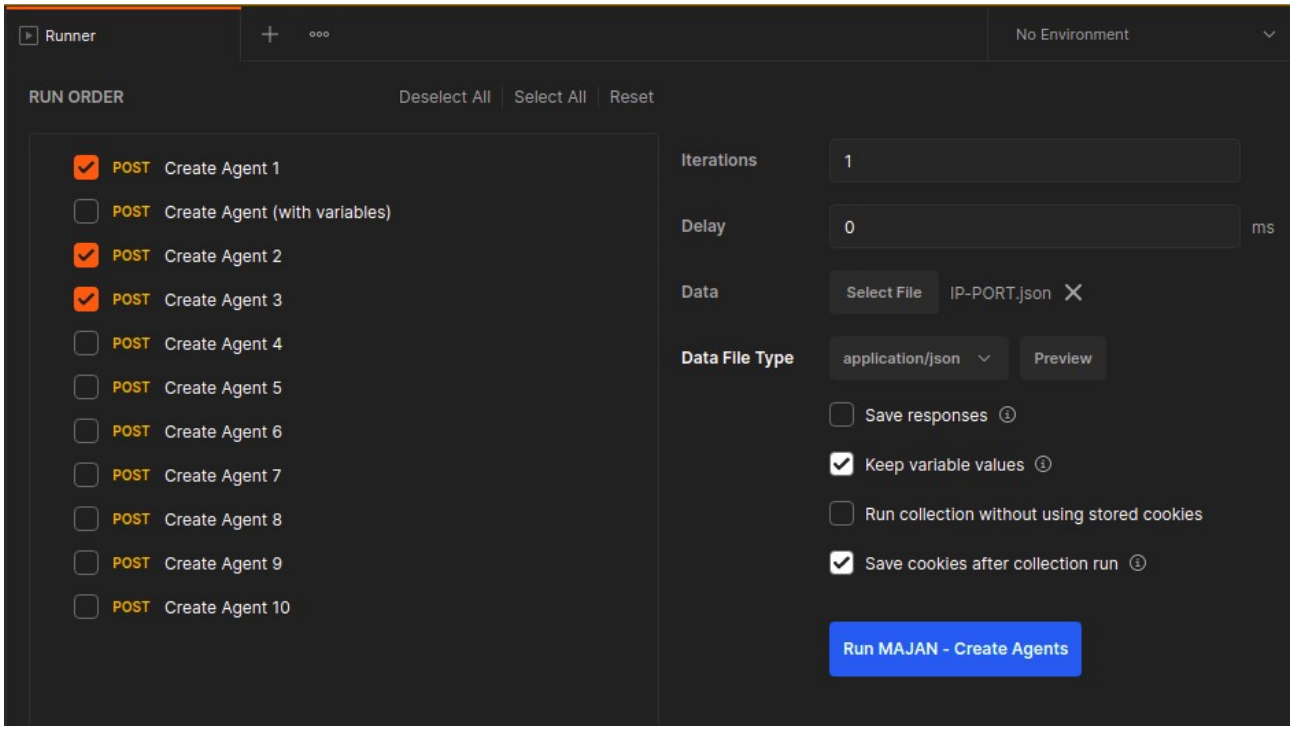


Figure 45-e. Example: select 3 agents and respective configuration file.

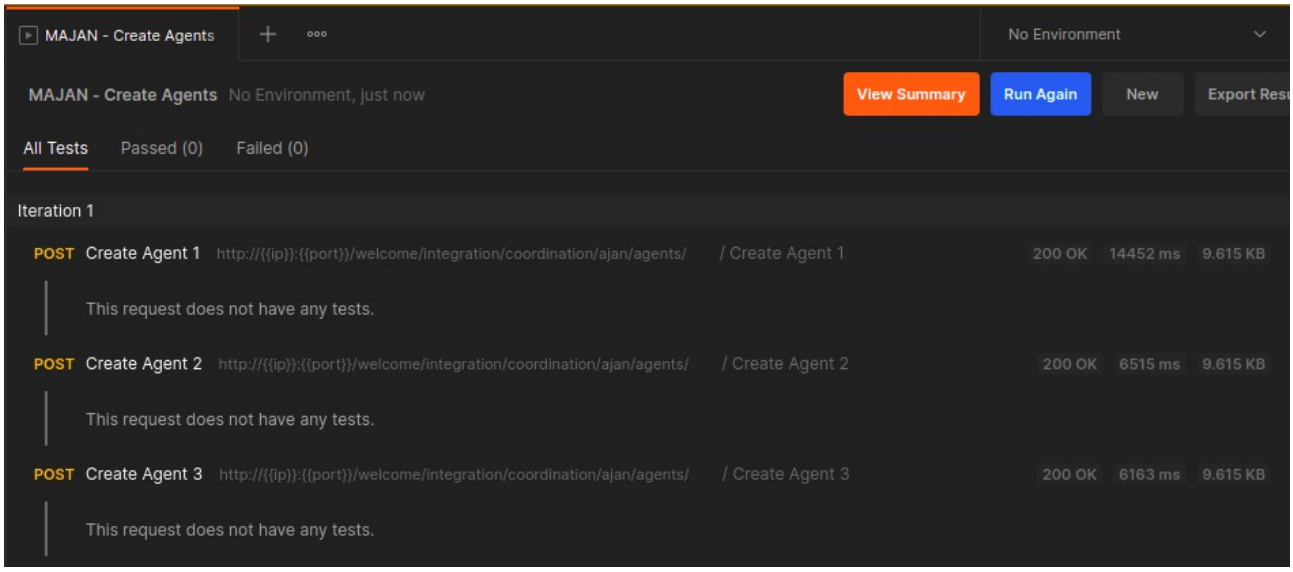


Figure 45-f. Example: succesful creation of selected agents

Populating Local Agents Repository

In multiagent coordination, agents need to communicate with each other and therefore, they need to know how they can contact other agents. To do so, agents make use of their Local Agents Repository (LAR) where they store the contact information of other agents. However, it is necessary to populate the LAR of agents before they start coordination. Thus, MAJAN provides a collection called “MAJAN - Populate LAR.postman_collection.json” which consists of 8 Postman requests with different content in terms of agent name and amount. These requests contain agent URI, ID, and address in the payload, and they are sent to “populateLAR” capabilities of agents. Moreover, as shown in figure 46-a, these requests have 3 variables: ip, port, and agentId which are given in the configuration file we select to run collection.

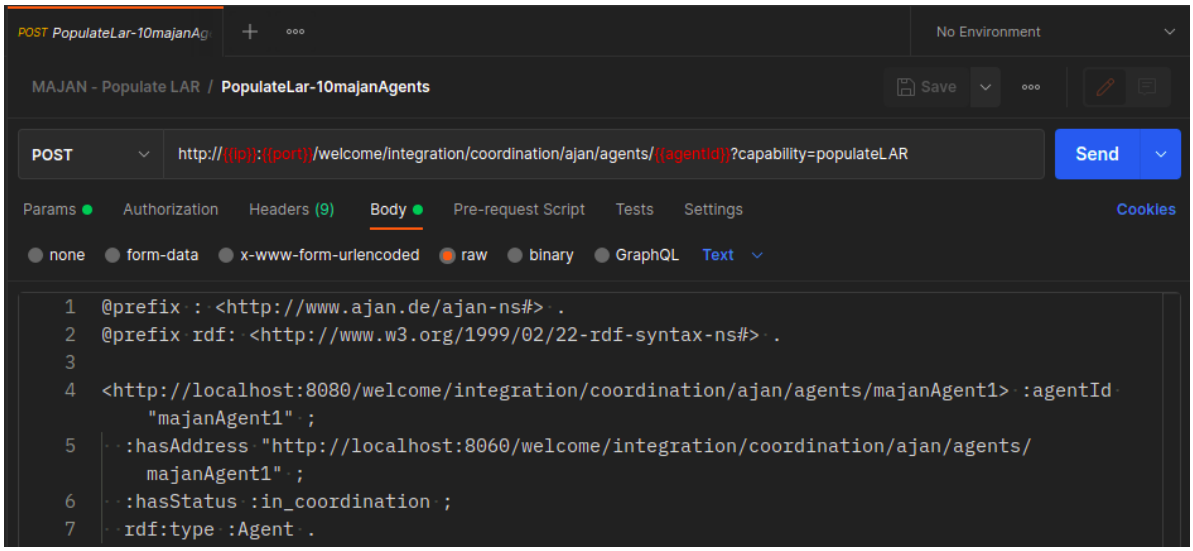


Figure 46-a. Request to populate LAR and it requires three values for ip, port and agentID varibales.

In order to run this collection, import it and click on “Run Collection” as it was done in the previous, Create Agents section. Then select “PopulateLar-10majanAgents” since the agents we created in the previous section were named as majanAgents. The next step is to select the configuration file (figure 46-b) named “5-IP-PORT-MajanAgentId.json” in “MAJAN/Postman Collections/Populate LAR”(https://github.com/AkbarKazimov/MAJAN) folder. Then click on “Run MAJAN-Populate LAR” button in blue.

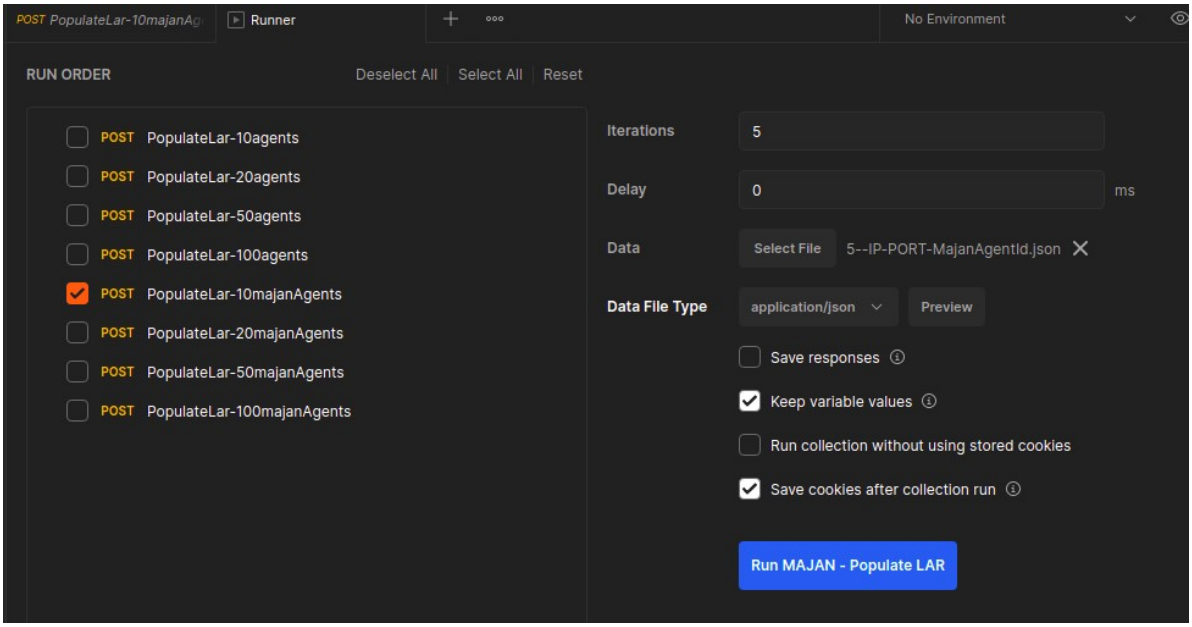


Figure 46-b. Selection of PopulateLar-10majanAgents and respective configuration file.

After the selected request is sent to agents, we will see the logs as shown in figure 46-c which represents that agents successfully populated their LAR.

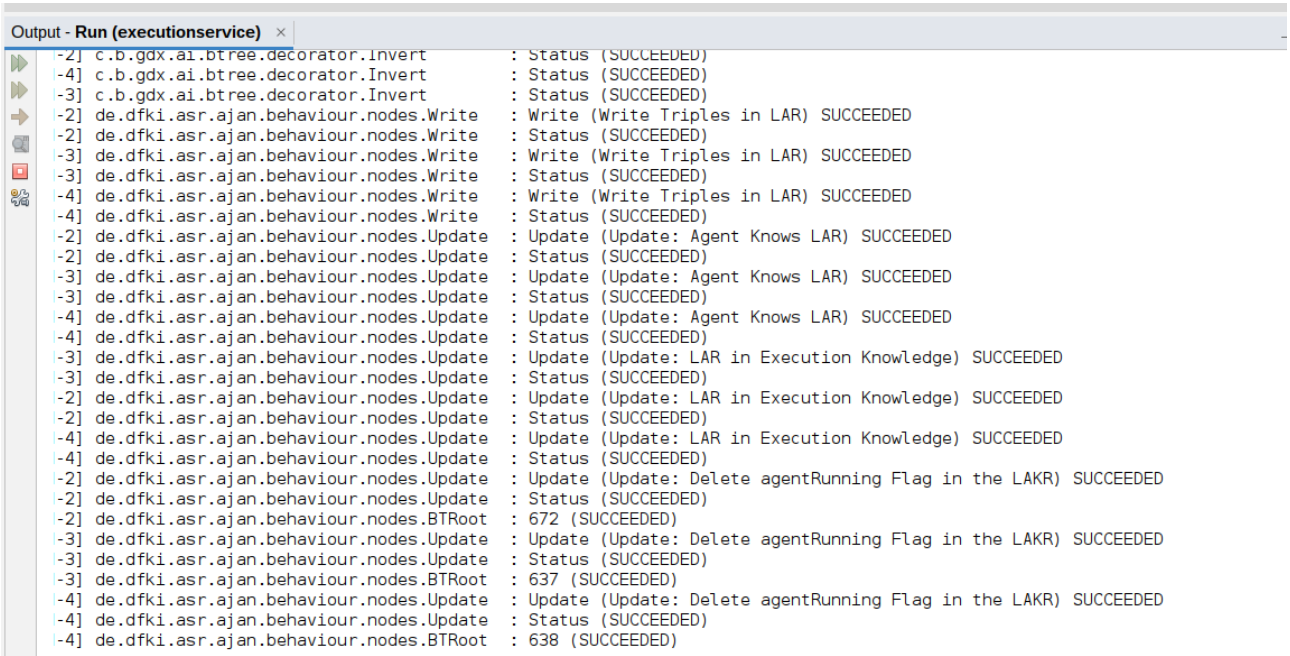


Figure 46-c. Logs in Netbeans for Population of LAR of 3 agents.

It is also possible to validate the population of LAR by checking the repository content in RDF4J Workbench, as shown in figure 46-d.

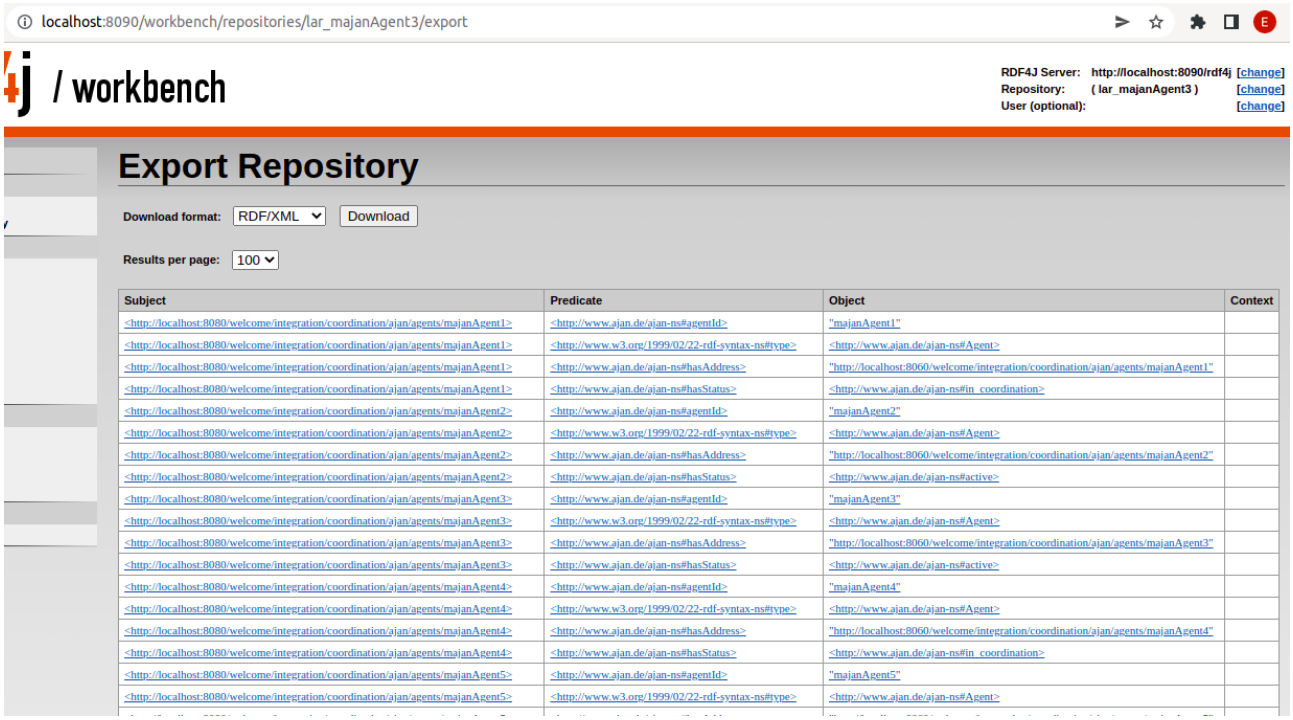


Figure 46-d. LAR repository content of majanAgent3.

Starting Multi Agent Coordination

The last step for agents to start their coordination is to receive a signal to their respective endpoints. To do so, MAJAN provides a collection consisting of 3 requests to start Request, CSGP, and Clustering coordination protocols. Each of these requests includes the necessary initial message including use case title, quorum, timeout, etc. The name of the collection is “MAJAN - Start Coordination.postman_collection.json” and it is provided in “MAJAN/Postman Collections/Start Coordination” (https://github.com/AkbarKazimov/MAJAN) folder.

So firstly, this collection must be imported into Postman. Then, it is necessary to click on one of the three requests. As an example, “Start CSGP Coordination Protocol” request is selected as shown in figure 47-a.

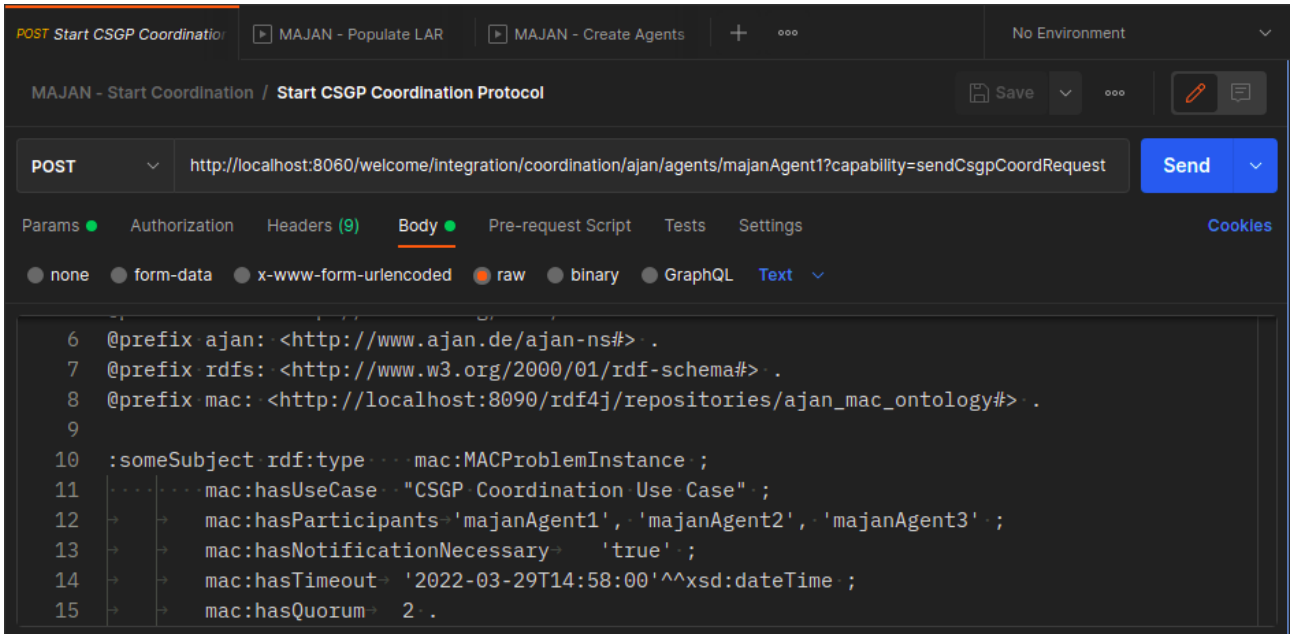


Figure 47-a. Start CSGP Coordination Protocol request which informs agent to start CSGP protocol.

As can be seen in figure 47-a, the agent receives this message to its “sendCsgpCoordRequest” capability. Moreover, the Body of this request should contain information about a MACProblemInstance which should have a use case name, name of participants, info about notification necessary, timeout, and quorum. Furthermore, it is enough to send this message to only one agent which will act as a dedicated agent, and it will inform other participant agents respectively according to coordination protocols. That is why there is no need to upload any configuration file to send this request unlike “Create Agents” and “Populate LAR”. In figure 47-a, we send this request to majanAgent1 as you can see from the URL. Once Send (in blue) is clicked, majanAgent1 will receive this message and act accordingly. However, it is important to check the ip and port and make sure that they point to the correct AJAN Execution Service. Once the request is sent, agents will log the execution of their BTs into NetBeans or cmd, depending on the execution method. Additionally, agents will log their activities to Monitoring panel in MAJAN. Refer to the next section (i.e. Monitoring Activities of Coordinating Agents) to get more information.

MAJAN Extension of AJAN Editor

In order to provide features required for multiagent coordination, AJAN Editor has been extended appropriately with MAJAN. The first feature is the monitoring panel where users can monitor the activities of coordinating agents. Moreover, MAJAN allows users to view the result of multiagent coordination (into groups) in a user-friendly way rather than RDF triples in RDF4J repositories. Furthermore, users can execute a centrally running grouping algorithm and view its result appropriately in MAJAN. Finally, users can compare MAC and Central solutions in Compare Solutions section.

Monitoring Activities of Coordinating Agents

All the execution of all BT nodes is normally logged to Netbeans or CMD depending on the execution method of AJAN Service. This logging technique is good when there is one agent running in AJAN Service. However, it gets difficult very quickly to debug when multiple agents are running in parallel and all of their BT nodes are logged to the same place since these logs don't contain the ID of the agent and usually there are lots of BT nodes that create a huge amount of logs in Netbeans. Therefore, Monitoring Panel in MAJAN is developed to overcome these issues and allow users to be able to monitor the coordination process. UI of this panel is shown in figure 48-a.

In order to log any activity, agents need to explicitly send a message to MAJAN Logging Service. The message can be constructed with the SPARQL query shown in figure 48-b and all log messages should have a common structure such that MAJAN can understand them. It is important to note that Activity field can be specified flexibly. In figure 48-b, Activity field is built by using Concat function of SPARQL.

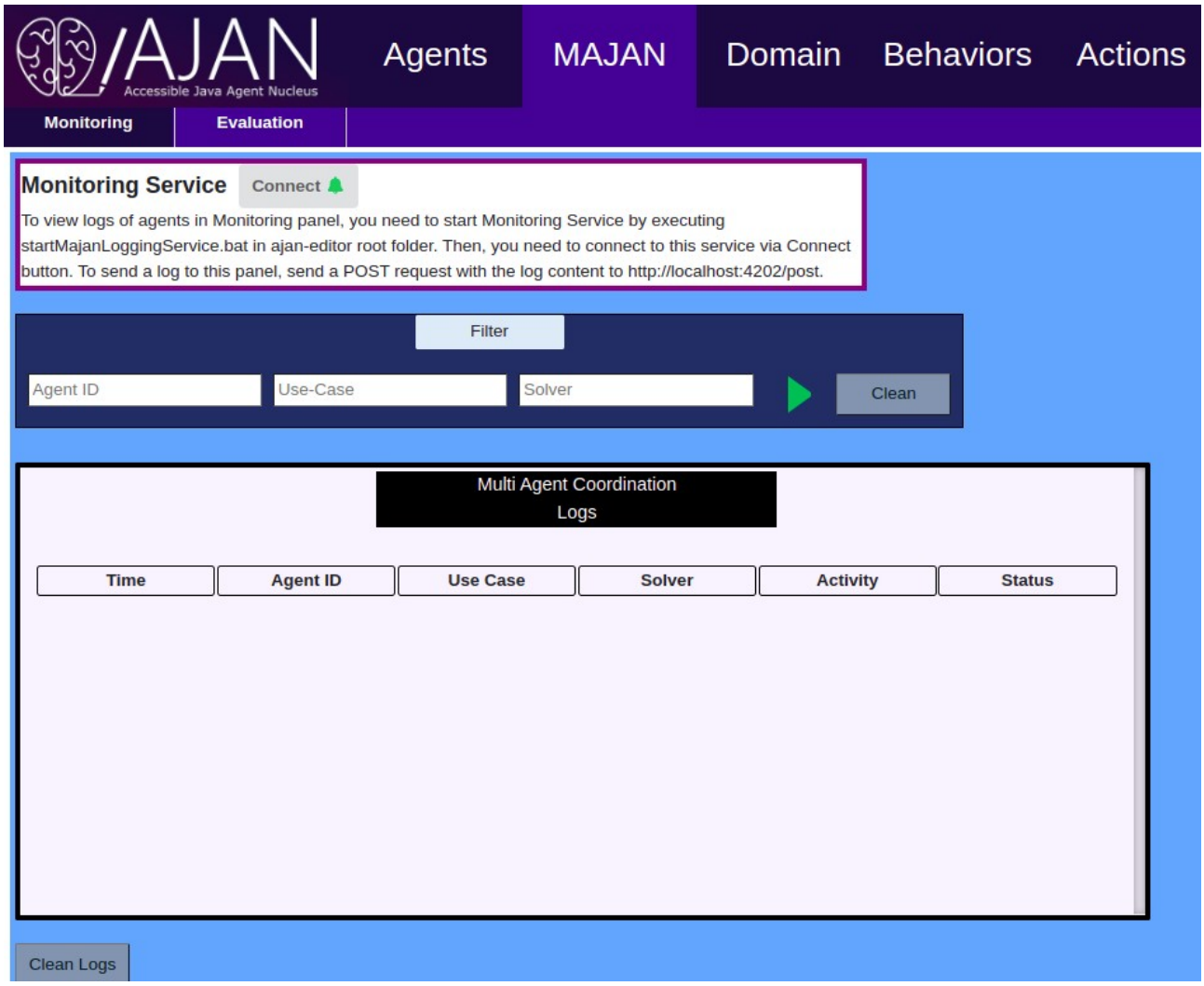


Figure 48-a. Monitoring Panel UI in AJAN Editor

```
CONSTRUCT {
  ?newBnode   rdf:type  mac:Log ;
              mac:hasAgentId  ?thisAgentId ;
              mac:hasUseCase  ?useCase ;
              mac:hasSolver   ?solver ;
              mac:hasActivity   ?activity ;
              mac:hasActivityStatus   ?activityStatus . }
WHERE {
  ?thisAgent  rdf:type  ajan:Agent, ajan:ThisAgent ;
        ajan:agentId  ?thisAgentId .
  ?macInstance  rdf:type  mac:MACProblemInstance ;
          mac:hasId   ?macId ;
          mac:hasUseCase  ?useCase .
  OPTIONAL {
    ?macInstance  rdf:type  mac:MACProblemInstance ;
            mac:hasSolver   ?solver . }
  OPTIONAL{
    ?conversation   rdf:type  mac:Conversation ;
            mac:hasId   ?convId ;
            mac:hasTimeout  ?timeout ;
            mac:hasNotificationNecessary  ?notificationNecessary ;
            mac:hasMacProblemId   ?macId .  }
  { SELECT (GROUP_CONCAT(?participantId ; separator=", ") AS ?participants)
    WHERE{
      ?conversation   rdf:type  mac:Conversation ;
              mac:hasParticipants   ?participantId .  } }
  { SELECT (GROUP_CONCAT(?contentIri ; separator=", ") AS ?contentIris)
    WHERE{
      ?conversation   rdf:type  mac:Conversation ;
              mac:hasContent  ?contentIri . } }

  BIND(CONCAT("Starting to Solve Clustering\\nConversation ID: ", STR(?convId), "\\nConversation
  Participants: ",STR(?participants), "\\nTimeout: ",STR(?timeout), "\\nNotification Necessary: ",
  ?notificationNecessary, "\\nMAC ID: ", ?macId, "\\nContent IRIs: ", ?contentIris) AS ?activity)
  BIND("Succes" AS ?activityStatus)
  {  BIND(BNODE() as ?newBnode)   }   }
```

Figure 48-b. SPARQL query to construct a log message for MAJAN Monitoring Panel.

In order to be able to receive logs and display them in UI, the first step is to start Majan Logging Service such that the logs sent by agents can be read, parsed and displayed by MAJAN. To start logging service, find “startMajanLoggingService.bat” in AJAN Editor folder and just run it. Once it is running, go to Monitoring UI in MAJAN and click on “Connect” button.

In case you would like to test whether log messages are received and displayed correctly before starting a MAC process, you can use the Postman request provided in “MAJAN – Logging.postman_collection.json” which is in “MAJAN/Postman Collections/MAJAN Logging” (https://github.com/AkbarKazimov/MAJAN) folder.

We can view the logs for our example of creating 3 agents and running CSGP protocol in figure 48-c.

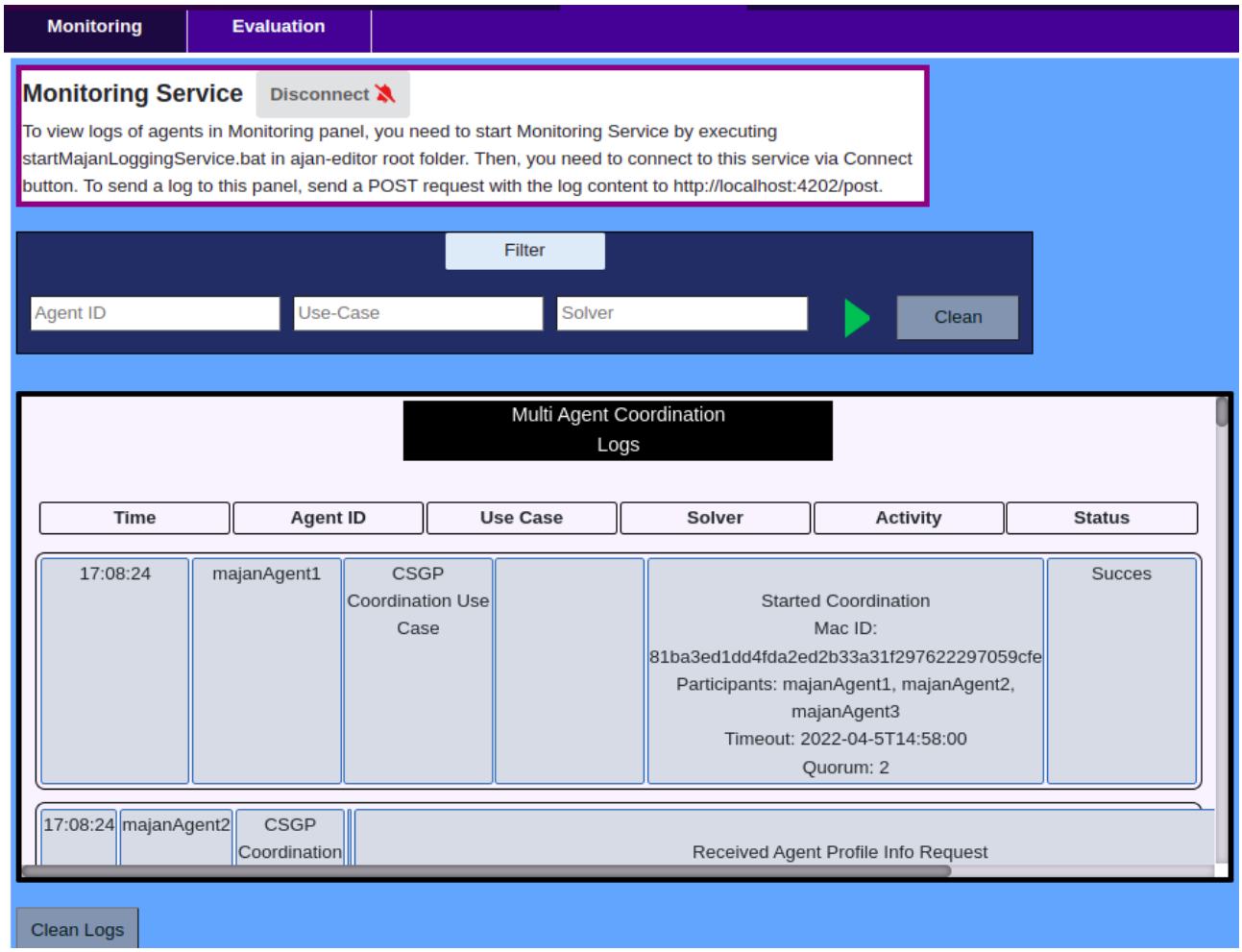


Figure 48-c. Logs for CSGP Coordination Protocol

Evaluating Results of Coordination into Groups

MAJAN provides an Evaluation panel (figure 49) where users can display grouping results of MAC and centrally running algorithms and compare those results.

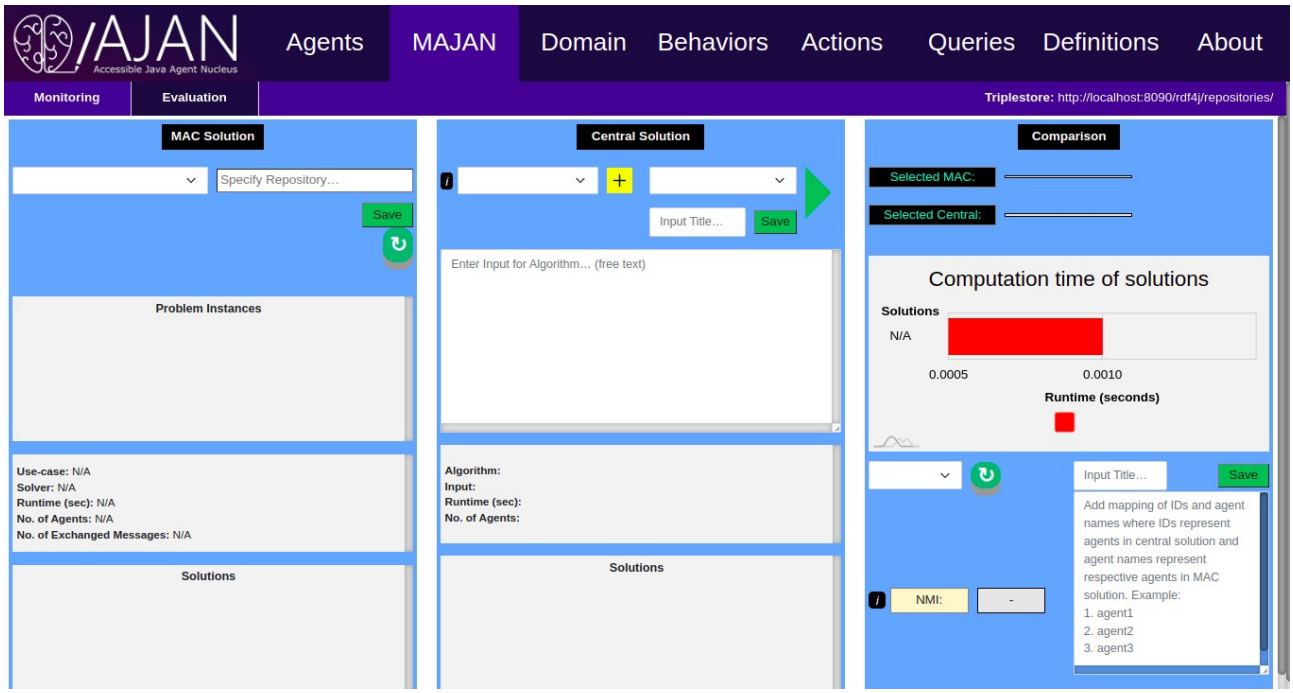


Figure 49. Evaluation panel of MAJAN in AJAN Editor

MAC Solution

MAC Solution section as shown in figure 49 allows users to display grouping results of MAC processes. To do so, it is required to write a repository (which stores the results) in the respective field and click on Save button. Then MAJAN reads the results (if exist) from the selected repository and displays them accordingly. Problem Instances section displays all the problem instances stored in the selected repository, whether they have a solution or not. Once a problem instance is selected, its solutions (if exist) are displayed in Solutions section.

We continue with our example of three agents running CSGP Protocol. Since we created majanAgent1, majanAgent2, and majanAgent3, we can write any of their Agent Knowledge Base repository in the “Specify Repository” field. Let’s write majanAgent1 and see if there is any problem instance exists (figure 50-a).

In order to be able to parse the results, they must be saved in a common structure in repositories. Therefore, MAJAN requires the results in the structure shown below:

```
<macProblemSubjectIRI>    rdf:type    mac:MACProblemInstance ;
                          mac:hasId   “someId” ;
                          mac:hasUseCase   “someUseCase” ;
                          mac:hasNumberOfAgents    3 ;
                          mac:hasSolver    “someSolver” ;    (OPTIONAL)
```

	mac:hasRuntime	60 ;	(OPTIONAL)
	mac:hasSolution	<groupingIRI> .	(OPTIONAL)
<groupingIRI>	mac:hasMembers	<groupIRI> ;	(OPTIONAL)
	mac:hasRank	1 ;	(OPTIONAL)
	mac:hasValue	5 .	(OPTIONAL)
<groupIRI>	mac:hasMembers	“someAgentId” .	(OPTIONAL)

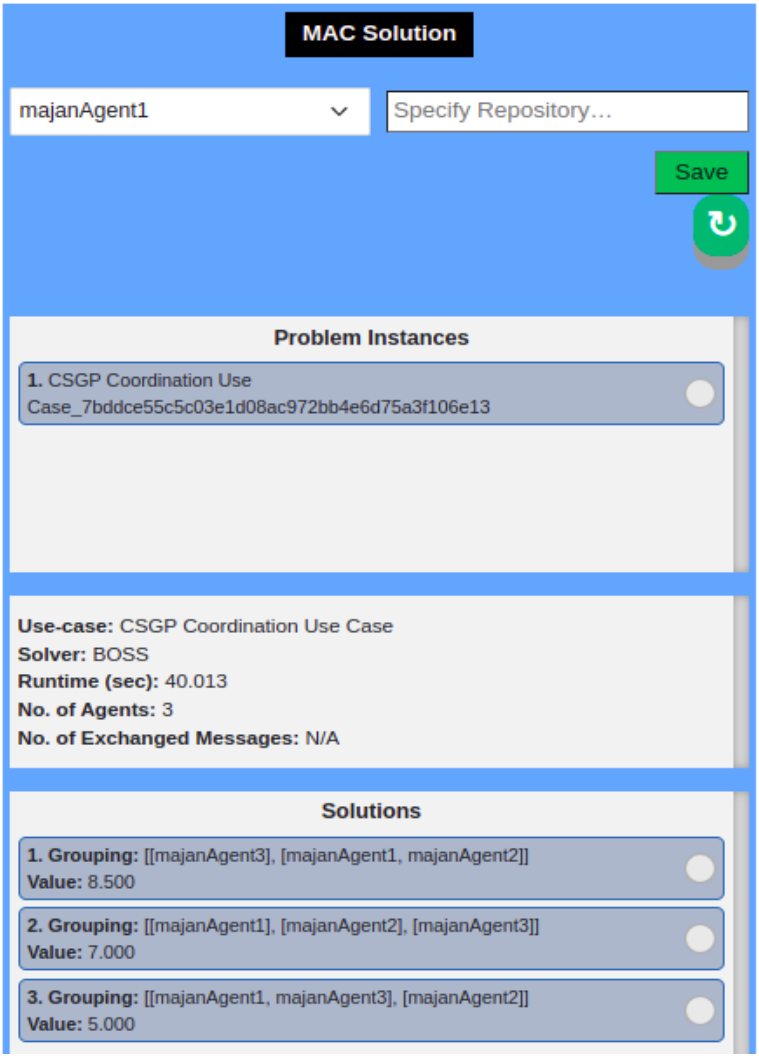


Figure 50-a. CSGP MAC result of 3 agents displayed in MAC Solution

Central Solution

In this section (figure 51), users can execute a centrally running grouping algorithm and view its result appropriately. In order to explain the features in this section, we will see running the BOSS algorithm as an example.

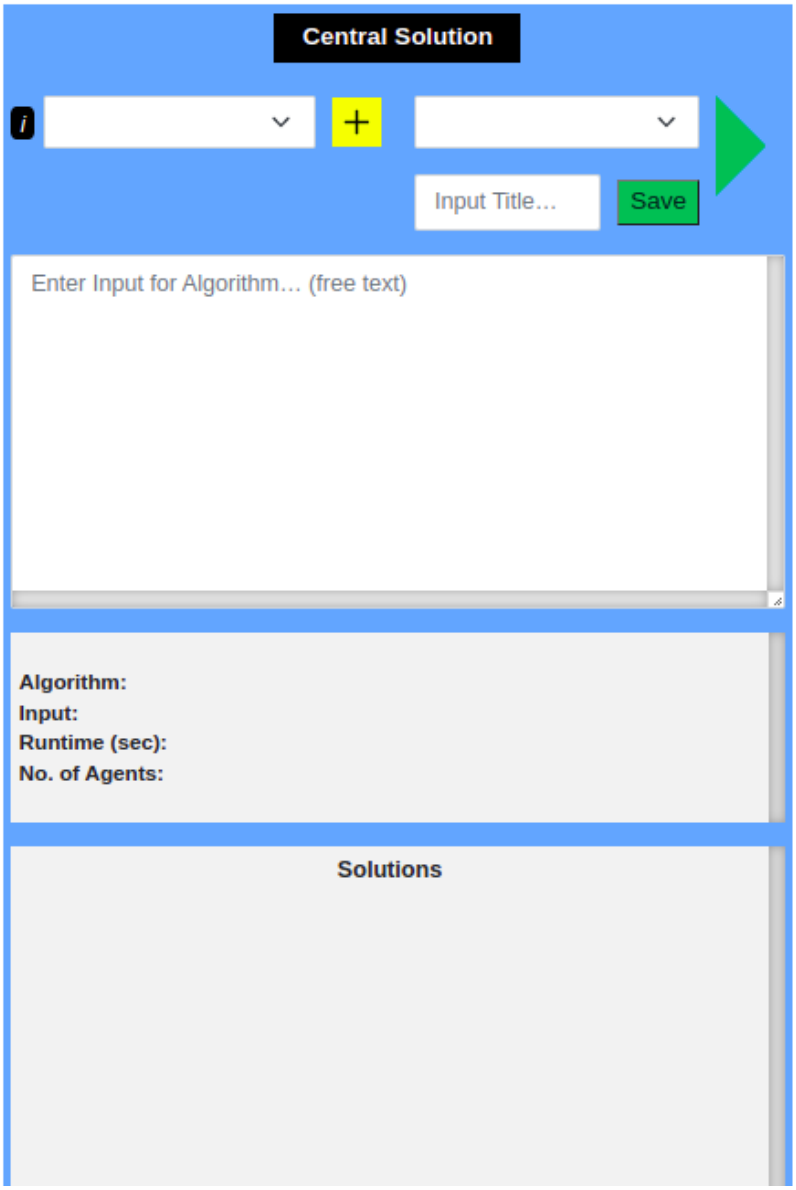


Figure 51. Central Solution section of MAJAN in AJAN Editor

In order to upload the BOSS algorithm to MAJAN, it is necessary to upload a JSON configuration file by using the yellow plus button in Central Solution section. In the configuration file (figure 52-a), the name of the algorithm, path to its jar file, timeout, and structure of its input should be specified. For more detailed information about the configuration file, check out Configuration File Structure paragraph below. Once a configuration file is uploaded, MAJAN reads and displays it in a user-friendly format in the tooltip (*i*) button (figure 52-b). Moreover, it is necessary to select an input to be passed to the selected algorithm. Therefore, input based on the input structure shown in the tooltip and a title to save it should be written to “Enter Input for Algorithm… (free text)” and “Input Title…” fields, respectively. We add an input for the BOSS algorithm as shown in figure 52-c.

```
{
    "algorithm":"BOSS_for_MAJAN_v1.1",
    "pathToJarFile":"/home/lexi/Desktop/Projects/MAJAN-Files/BOSS_for_MAJAN.jar",
    "timeout": "5",
    "keywords":[
        {
            "keyword":{
                "type":"Number of agents",
                "description":"description of number of agents",
                "element":{
                    "item":{
                        "name":"n",
                        "example_value":"10",
                        "datatype":"int",
                        "description":"description of n"
                    }
                }
            }
        },
        {
            "keyword":{
                "type":"CV",
                "description":"description of CV",
                "elements":[
                    {
                        "element":{
                            "items":[
                                {
                                    "item":{
                                        "name":"coalitionID",
                                        "example_value":"0",
                                        "datatype":"int",
                                        "description":"description of coalitionID"
                                    }
                                },
```


Figure 52-a. Configuration file for BOSS algorithm.

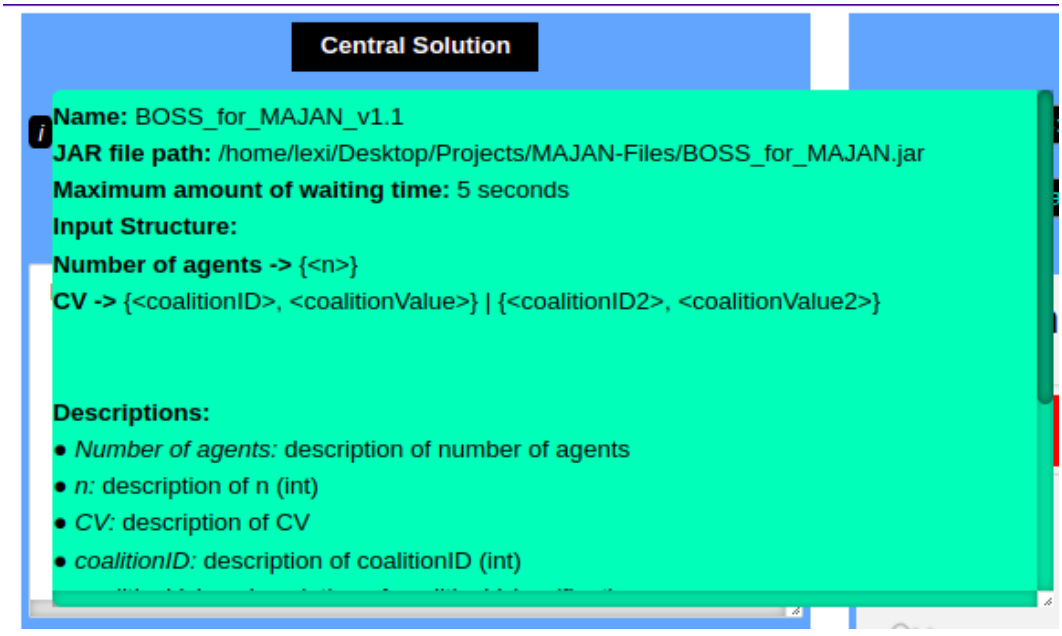


Figure 52-b. Representation of BOSS configuration file in tooltip.

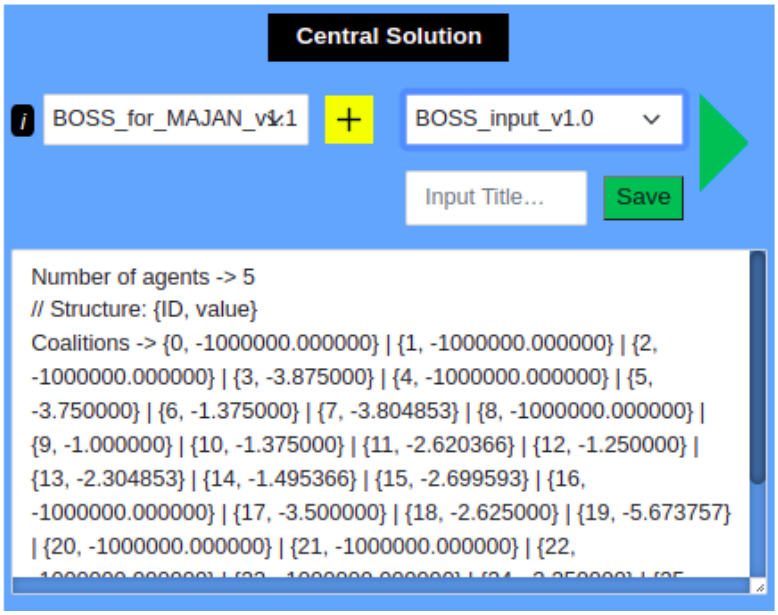


Figure 52-c. Example input added and selected to run BOSS algorithm

Since the BOSS algorithm and input for it are selected, we can click on the run button to run the selected algorithm with the selected input. Then, if there is a solution found by the algorithm, it is being displayed in the respective areas as shown in figure 52-d. For more detailed information about running a jar file in MAJAN, check out Running JAR File paragraph below.

There are 4 predefined algorithms provided by MAJAN to run in Central Solution section:

1) **BOSS**: this algorithm solves CSGP by finding the exact solution. In “MAJAN/Grouping Algorithm/BOSS” (https://github.com/AkbarKazimov/MAJAN) folder, there are 3 files:
    1) BOSS_config.json: json configuration file for the algorithm
    2) BOSS_for_MAJAN.jar: jar file to execute the algorithm
    3) BOSS-input-5agents.txt: an example input for the algorithm
2) **HDBSCAN**: this algorithm solves clustering problems in a density based technique. In “MAJAN/Grouping Algorithm/HDBSCAN” folder, there are 3 files:
    1) HDBSCAN_config: json configuration file for HDBSCAN algorithm
    2) HDBSCAN_for_MAJAN.jar: jar file to execute the algorithm
    3) hdbscan_majan_input.txt: an example input for the algorithm
3) **LCC_BOSS**: LCC is a use case developed and solved with MAJAN. In order to find a central solution for LCC, it is being integrated into original BOSS algorithm. LCC_BOSS algorithm expects use-case related information and then passes the required information to BOSS algorithm automatically and finally finds a solution for LCC. In “MAJAN/Grouping Algorithm/LCC_BOSS” folder, there are 3 files:

1) LCC_BOSS_config.json: json configuration file for LCC_BOSS algorithm
2) LCC_BOSS_for_MAJAN.jar: jar file to execute the algorithm
3) lcc-boss-input-5agents.txt: an example input for the algorithm

4) **CHC_HDBSCAN**: CHC is a use case developed and solved with MAJAN. In order to find a central solution for CHC, it is being integrated into original HDBSCAN algorithm. CHC_HDBSCAN algorithm expects use-case related information and then passes the required information to HDBSCAN algorithm automatically and finally finds a solution for CHC. In "MAJAN/Grouping Algorithm/CHC_HDBSCAN" folder, there are 3 files:
   1) CHC_HDBSCAN_config.json: json configuration file for CHC_HDBSCAN algorithm
   2) CHC_HDBSCAN_for_MAJAN.jar: jar file to execute the algorithm
   3) chc-hdbscan-input.txt: an example input for the algorithm

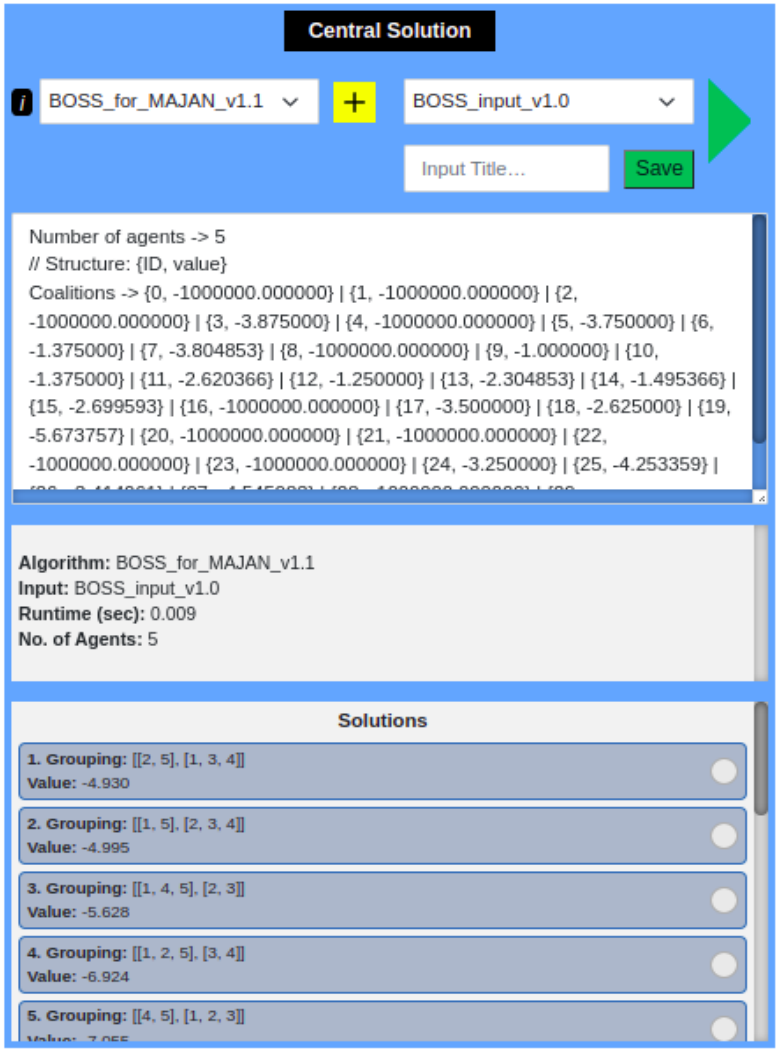


Figure 52-d. Grouping result of execution of BOSS algorithm

To summarize, there are 3 steps to run any of the provided algorithms:

1. Upload json configuration file
2. Add iput that can be accepted for the selected algorithm
3. Click Run button and view results

Configuration File Structure

Configuration file is a way of describing an algorithm to MAJAN so that necessary information about algorithms can be read and displayed by MAJAN. To do so, the file should include values for “algorithm”, “pathToJarFile”, “timeout” and “keywords”.

1. “algorithm” reflects the name of algorithm
2. “pathToJarFile” reflects the path to the jar file of the algorithm
3. “timeout” reflects the amount of time for MAJAN to wait until stopping the execution of algorithm since algorithm might get stuck in a loop.
4. “keywords”, “keyword”, “element”, and “item” are all used to describe the structure of input that algorithm expects. An example configuration file is shown in figure 53-a which includes the json fields listed below. For a visual explanation, see figure 53-b where corresponding parts for each of the following json fields are shown clearly.
    1. “keywords”: starting point to describe the whole structure of input for algorithm. Called this way since it contains “keyword”s.
    2. “keyword”: as shown in figure 53-b, represents a single input line. An input line consists of “type”, “description”, “element” or “elements” as shown in figure 53-a.
    3. “type”: represents the name for type of input. For example, number of agents, minimum group size, coalition value, etc.
    4. “description”: contains the description of the json field it belongs to. The value for “description” fields are used in Descriptions section as shown in figure 53-a.
    5. “element”: describes values to be given to the algorithm. It has “item” or ”items” which are used to give more details about the input values expected by algorithm.
    6. “items”: represents the array of “item”s.
    7. “item”: reflects the single value to be given to algorithm. It has “name”, “example_value”, “datatype”, and “description”.
    8. “name”: is a placeholder.
    9. “example_value”: is used to build an example input automatically
    10. “datatype”: shows the datatype of input that is expected by algorithm.

Figure 53-a. Example configuration file for explanation

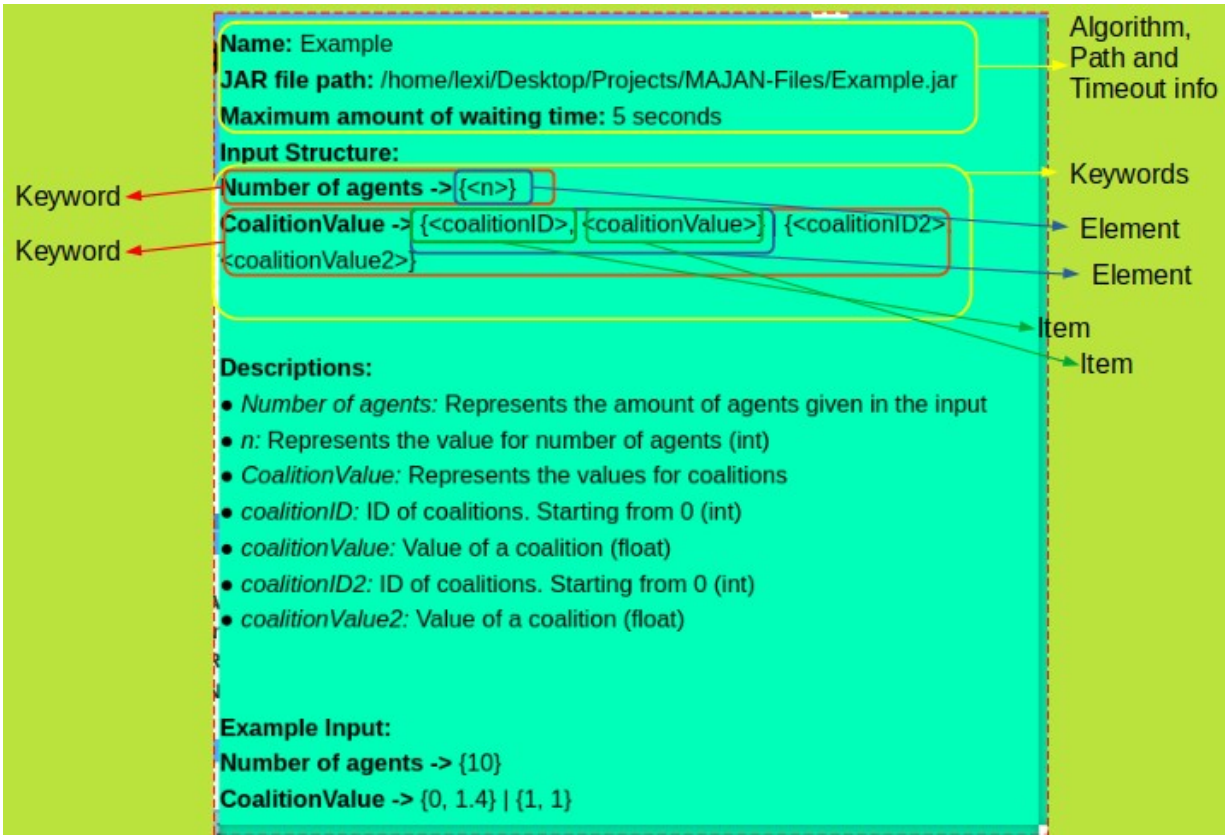


Figure 53-b. Representation of Example configuration file

Running JAR File

AJAN Editor is developed by using Ember JS, which is a JavaScript framework. Since Ember JS doesn't support running jar files or executing any commands, MAJAN executes the jar files via Java. To do so, the Executionservice module in AJAN Service provides a class name MajanService which has an endpoint (ajan_service_base_path+"/majanService/runJar") that accepts the input for the algorithm, the path of the jar file, and timeout. Then, the jar file in the given path is executed with Java code. More specifically, this endpoint consumes text/plain and produces application/json. Moreover, it requires two headers called "jarPath" and "timeout" for the reasons mentioned above. Finally, the payload that is sent to the endpoint is passed as the input to the algorithm.

It is necessary to note that, algorithms must produce a JSON text describing a grouping. In order to produce such a JSON for a grouping result, MAJAN provides the necessary java module in "MAJAN/Grouping Algorithms/GroupingResultAsJson" (https://github.com/AkbarKazimov/MAJAN) folder.

Once MAJAN receives the grouping result as a JSON, it parses and displays the result in a user-friendly format, as shown in figure 52-d.

Compare Solutions

The objective of Compare Solutions panel (figure 54) in MAJAN is to allow users to compare selected MAC and Central solutions in terms of their runtime and similarity of groups. To do so, a MAC and Central solution should be selected. Let's select the first solutions of MAC and Central , and we can see the indexes of selected solutions in "Selected MAC:" and "Selected Central:" sections as shown in figure 55-a. Additionally, the runtime of selected solutions is drawn in the chart shown in figure 55-a. By hovering over the rows in the chart, we can see the exact amount of runtime in seconds.

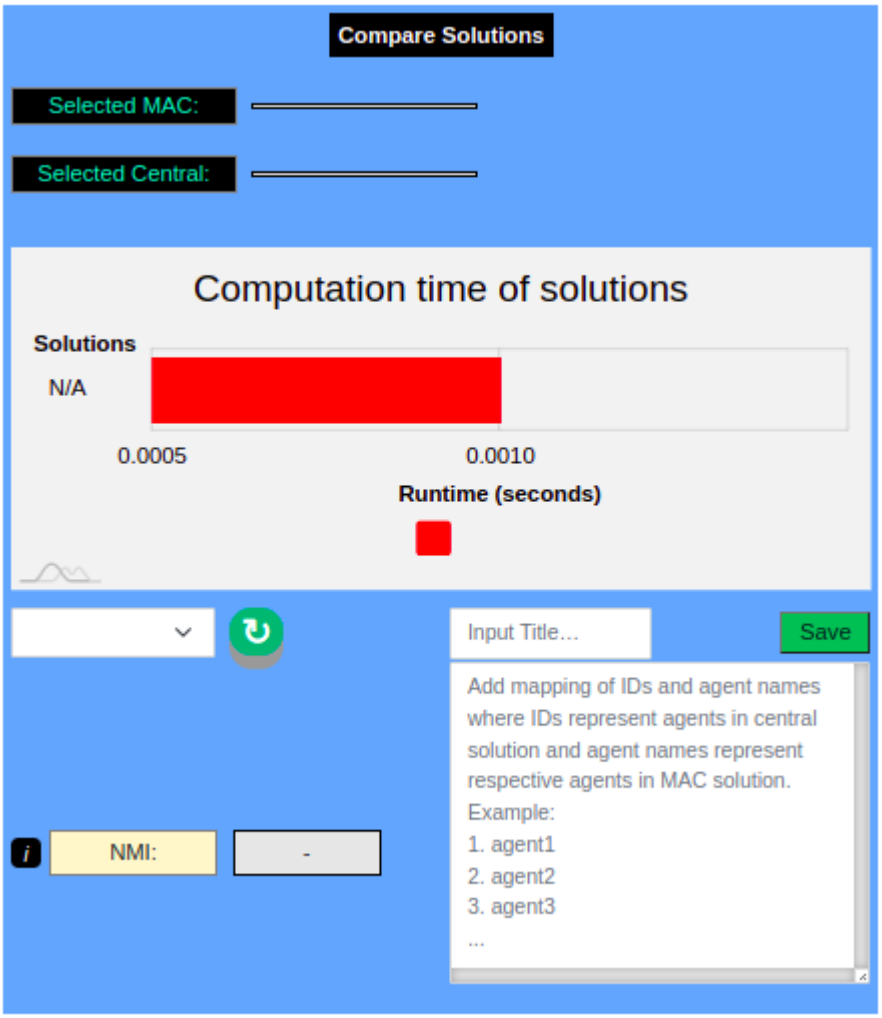


Figure 54. Compare Solutions panel in MAJAN

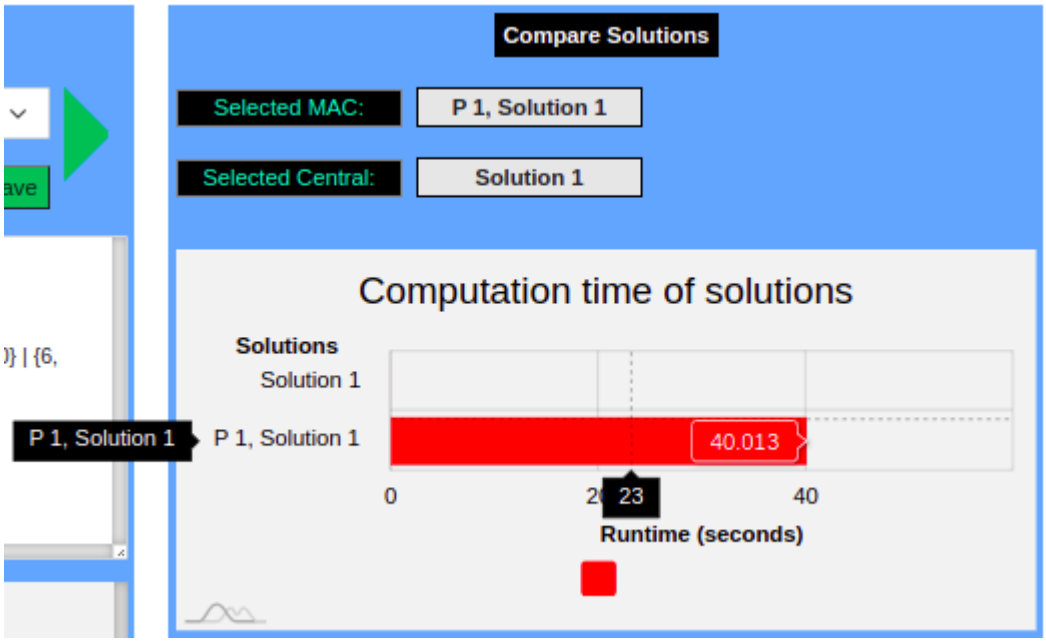


Figure 55-a. Visualization of selected solutions in Compare Solutions section

NMI Score

In order to compute the similarity of selected grouping solutions, MAJAN supports computing Normalized Mutual Information (NMI) score. NMI is a clustering evaluation metric that allows computing the similarity between two clustering results. NMI score ranges between 0 (no similarity) and 1 (maximum similarity).

To compute the NMI score, it is required to compare grouping results with the same amount of elements. Moreover, the names of elements must be the same. Since agents can take various names in MAC, it is necessary to provide a list that describes the correct mapping between agent names for MAC and Central solutions. For example, we created 3 agents with the names majanAgent1, majanAgent2, and majanAgent3 to run the MAC process. In the Central solution, agents are named 1, 2, and 3. It is necessary to let MAJAN know which agent in the MAC solution corresponds to which agent in the Central solution. Only then, MAJAN can compare the grouping solutions. Therefore, MAJAN requires a list to be able to map MAC agents to Central agents. Such a list should be added and saved such that it can be selected afterward as shown in figure 55-b.

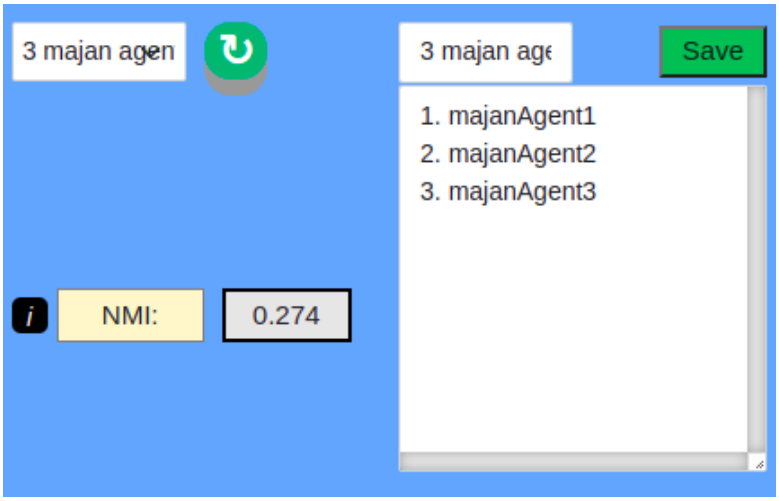


Figure 55-b. List of names of MAC agents corresponding to Central agents

Example Comparisons of Solutions

Since we have seen how to run the MAC process, view its result in MAJAN, run a Central algorithm, view its result, and compare the selected solutions, let's take a look at the comparison of some solutions in the figures below.

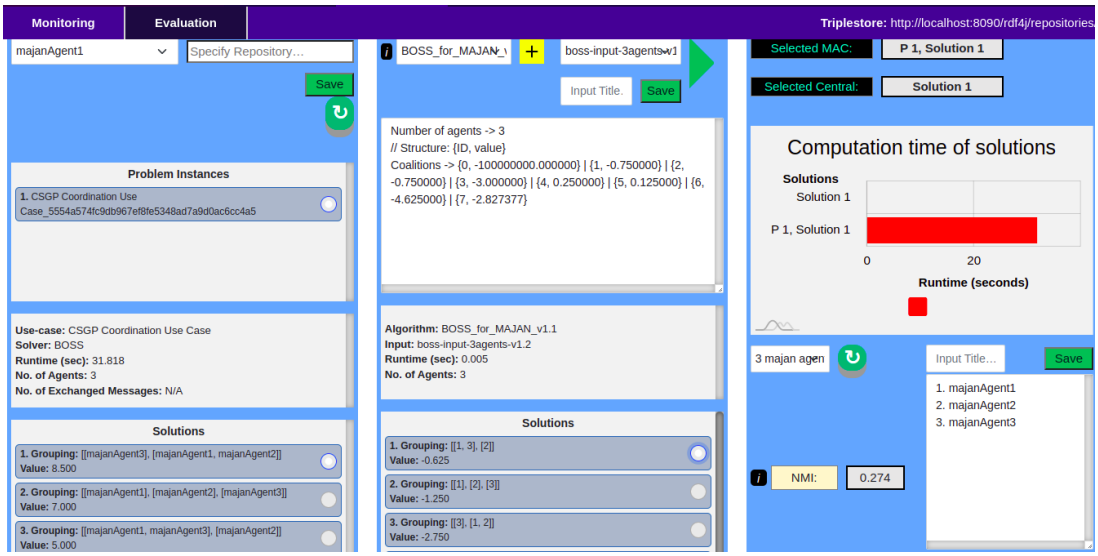
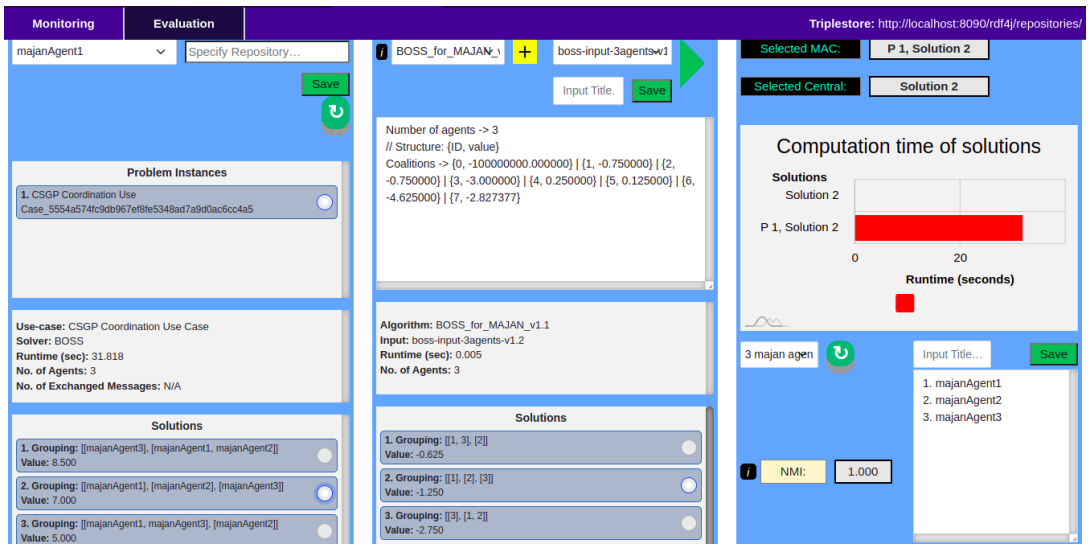


Figure 56-a. NMI score is 0.274 when comparing first MAC and Central solutions.



Figre 56-b. NMI score is 1 when comparing second MAC and Central solutions

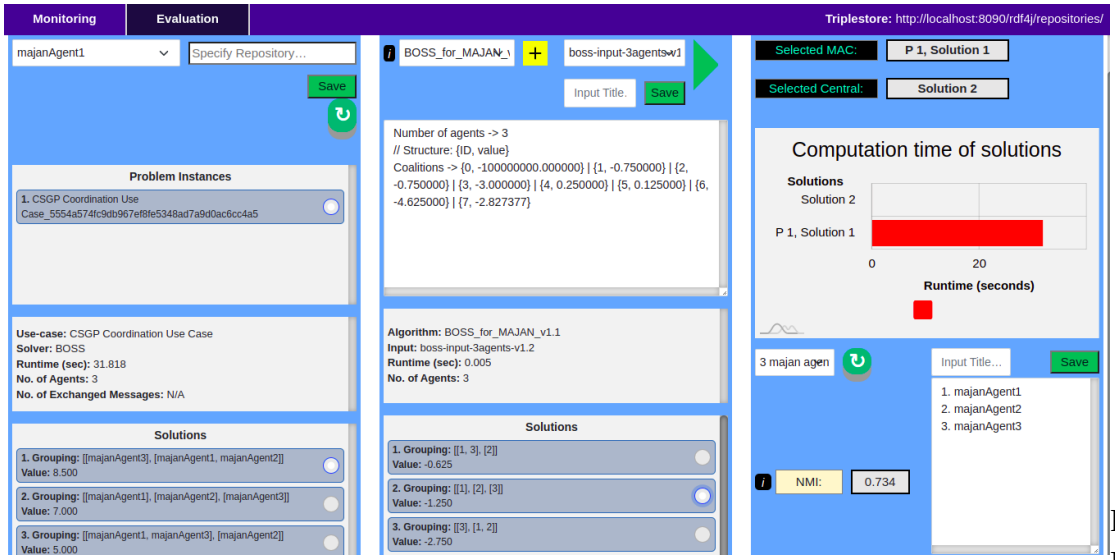


Figure 56-c. NMI score is 0.734 when comparing first MAC and second Central solutions.

MAJAN Extra

In this section, we provide additional information based on our experience which might be useful for users when using MAJAN, designing MAC behaviors of agents, or running MAC processes.

Useful Tips

1) When designing behaviors of agents, sometimes SPARQL queries might get large, and therefore, it could be difficult to easily check the validity of queries with eyes. For this reason, you can use this website (https://sparql-playground.sib.swiss/) to validate SPARQL queries.

2) Moreover, you can validate RDF triples by using this website (http://rdfvalidator.mybluemix.net/).

3) Sometimes it is necessary to run a SPARQL query in a repository to check if it works correctly before starting to run agents. To do so, RDF4J provides a nice UI (figure 57-a) to run queries, add/remove triples to/from repositories, etc. However, RDF4J Workbench cannot run large queries, as shown in figure 57-b. In such a case, it is necessary to send queries to the SPARQL endpoint of a repository. To do so, you can use Postman. The Postman collection including some large SPARQL queries that have been executed is provided in “MAJAN/Postman Collections/RDF4J Sparql Endpoint/MAJAN - Large Sparql Queries.postman_collection.json”.

4) Furthermore, sometimes the result of the execution of a SPARQL query is unexpected. If the query is executed in Agent Knowledge (AKB), then we can simply execute the query manually to find out the problem. However, this is not the case when the query is executed in Execution Knowledge (EKB) since there is no physical repository in the RDF4J workbench for EKB. Therefore, we cannot know what is stored in EKB at the time of

execution of the SPARQL query. To overcome this issue, you can write everything in EKB to any empty repository in RDF4J Workbench at any step of BT to be able to manually run the SPARQL. To do so, you can use the BT node shown in figure 57-c as an example.

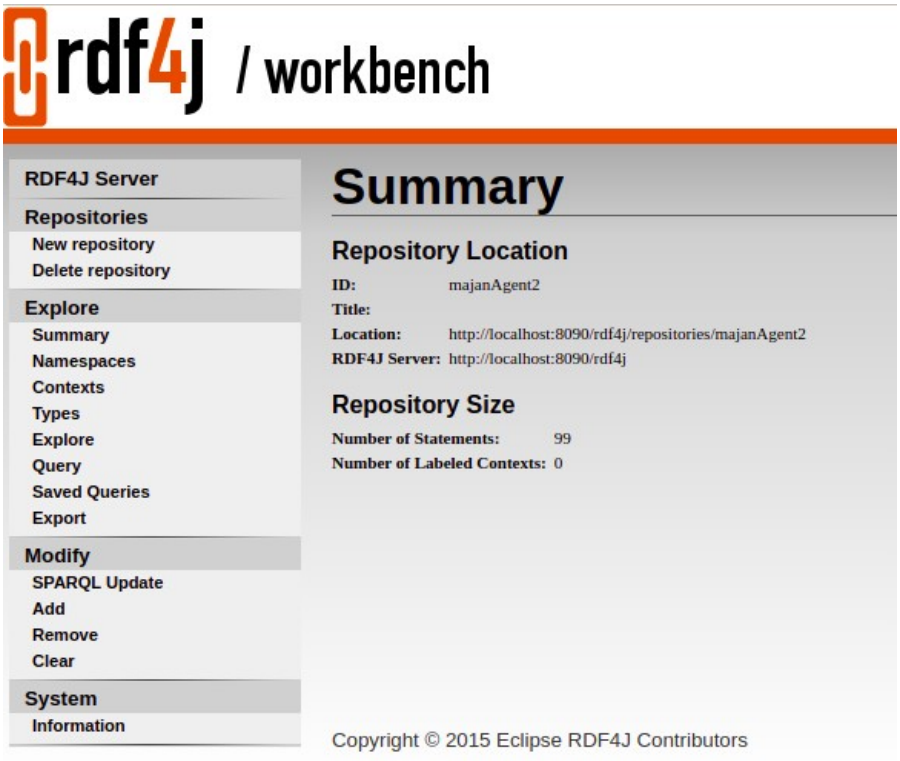


Figure 57-a. RDF4J Workbench UI with features on the left side.

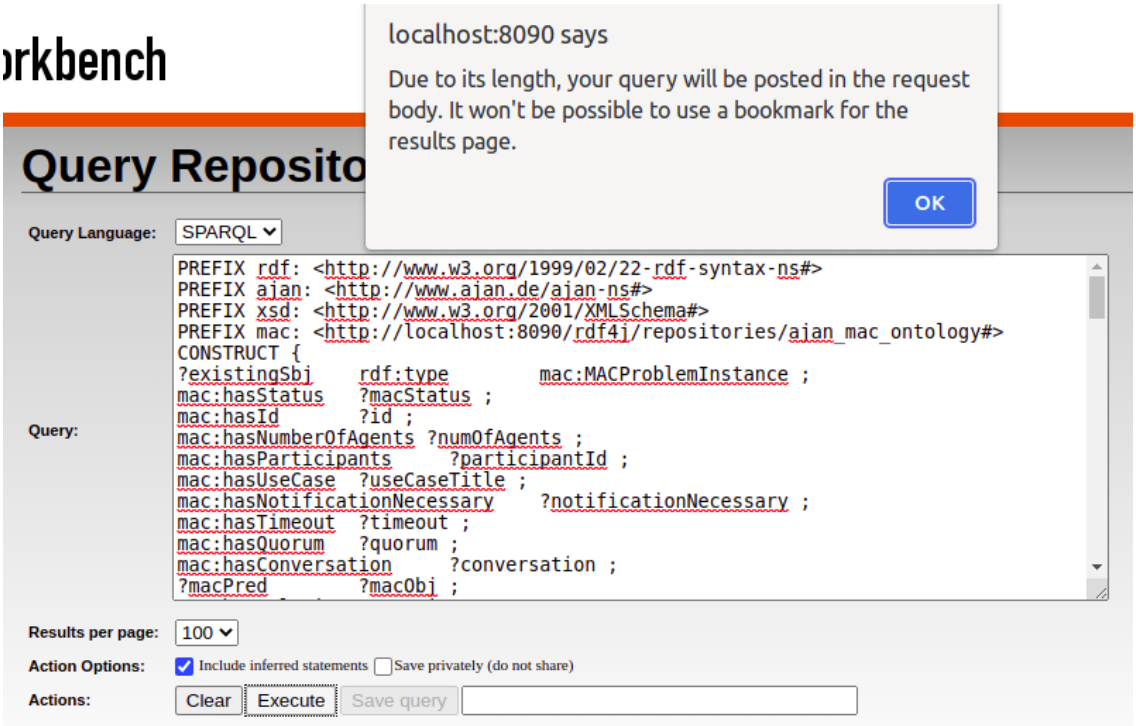


Figure 57-b. Error thrown when SPARQL query is large

```
:WriteEverythingToLSR
    a bt:Write ;
    rdfs:label "Write everything to LSR" ;
    bt:query [
        a bt:ConstructQuery ;
        bt:originBase ajan:ExecutionKnowledge ;
        bt:targetBase ajan:LocalServicesKnowledge ;
        bt:sparql """
                    CONSTRUCT {
                        ?s ?p ?o .
                    }
                    WHERE {
                        ?s ?p ?o .
                    }
        """^^xsd:string ;
    ] .
```

Figure 57-c. BT node to write everything in EKB to LSR

Centrally Running Grouping Algorithms

As mentioned before, MAJAN provides algorithms to be executed in the Central Solution section of MAJAN in AJAN Editor. In this section, we provide the source code of those and other algorithms.

1) **BOSS** algorithm: source code is provided in "MAJAN/Grouping Algorithms/BOSS/BOSS_for_MAJAN(source_code)" (https://github.com/AkbarKazimov/MAJAN) folder.
2) **HDBSCAN** algorithm: source code is provided in "MAJAN/Grouping Algorithms/HDBSCAN/HDBSCAN_for_MAJAN(source_code)" folder.
3) **LCC_BOSS** algorithm: source code is provided in "MAJAN/Grouping Algorithms/LCC_BOSS/LCC_BOSS_for_MAJAN(source_code)" folder.
4) **CHC_HDBSCAN** algorithm: source code is provided in "MAJAN/Grouping Algorithms/CHC_HDBSCAN/CHC_HDBSCAN_for_MAJAN(source_code)" folder.
5) **CoalitionGenerator** algorithm to generate coalitions for CSGP solvers: source code is provided in "MAJAN/Grouping Algorithms/CoalitionGenerator(source_code)" folder.
6) **GroupingResultAsJson** module to describe Grouping Solutions in JSON format: source code is provided in "MAJAN/Grouping Algorithms/GroupingResultAsJson(source_code)" folder.

Possible Errors and Their Solutions

1) When designing behaviors for MAC, it is very important to pay attention to the type of events. In most cases, agents should be able to handle the same events multiple times in

parallel. For example, agents should be able to run multiple MAC processes in parallel. In order to make BTs be able to handle multiple events, they should use QueueEvent instead of Event. Otherwise, agents will not be able to run correctly when a certain event tries to trigger its BT multiple times in parallel.

2) Moreover, sometimes behaviors of agents don't work correctly when using blank nodes created with BNODE() function of SPARQL. In such a case, you can create new IRIs by using the lines as shown in figure 57-d.

```
{
  BIND(SHA1(xsd:string(NOW())) AS ?uniqueId)
  BIND( IRI(CONCAT(STR(mac:SomeExampleIRI), STR(?uniqueId))) AS ?newNode)
}
```

Figure 57-d. SPARQL lines to create new IRI instead of using BNODE().

Justifications

1. Why MAC BTs use EKB instead of AKB?
    1. Because agents should be able to run multiple MAC processes in parallel. If all MAC Problem Instances would be handled completely in AKB, then it wouldn't be possible to differentiate them. Because there would be multiple MAC instances running at the same time in AKB and any BT that needs to manipulate a running MAC problem instance, it couldn't know which MAC problem it should manipulate. However, by using EKB, this problem is resolved. Because there can be only one MAC problem instance in each EKB that is running. This issue can apply to other info or use cases. Thus, this is a solution to overcome the issue explained above.
2. Why not using variables for everything in Postman collections?
    1. Because it is not possible to dynamically specify the values in JSON. E.g. 1 variable specifies a single value. Let's say an agent has "canSpeak" predicate, which stores the language that the agent can speak. Different agents can speak different amounts of languages, and it is not possible to know and manage this dynamically beforehand. If we specified 3 language variables in postman and if the agent can speak 2 languages, then the last variable will be sent as well even though it is an empty string. And if the agent can speak more than 3 languages, there won't be any way to specify all of them in JSON configuration file dynamically.

Bibliography

csgp-survey: Talal Rahwan, Michael Wooldridge, Coalition structure generation: A survey, 2015
boss: Narayan Changder, Samir Aknine, A Bi-directional Search Technique for Optimal Coalition Structure Generation with Minimal Overlapping, 2021
hdbscan: Ricardo Campello, Davoud Moulavi, Hierarchical Density Estimates for Data Clustering, Visualization, and Outlier Detection, 2015