ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ
UNIVERSITY OF PIRAEUS
ΤΜΗΜΑ ΨΗΦΙΑΚΩΝ ΣΥΣΤΗΜΑΤΩΝ

DEMOKRITOS
ΙΝΣΤ. ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΤΗΛΕΠΙΚΟΙΝΩΝΙΩΝ

"DPOP implementation in meeting scheduling problem"

PROFESSOR: VOUROS GEORGIOS
STUDENTS: KARLIS VASEILIOS
PAPADOPOULOS GEORGIOS
TOLIOPOULOU CHRISTINA-ANNA

ACADEMIC YEAR: 2020-2021

Contents

1. Definitions, symbolisms and determination of DCOP problem in its general form

Distributed Constraint Optimization Problems (DCOPs) are prominent agent models in the field of multi-agent systems (MAS) and have been proposed to enable support of MAS in complex real time and uncertain environments [1]. It is widely used in many real-world problems such as: disaster management and coordination problems, radio frequency allocation problems, recommendation systems, scheduling problems, sensor network problems etc. In this section we will describe abstractly the DCOP problem along with its definitions and notations [1].

We can describe the classical DCOP as a model where agents are fully cooperative, have deterministic behavior and total knowledge. Also, regarding the environment this is static and deterministic. Below we put forward some formal definitions of the *schedule meeting problem* that we have to solve. Classical DCOP problem of this kind can be described by a tuple of components *P* where:

- $A = \{a_1, a_2, ..., a_m\}$ is a finite set of agents. Every agent has a preference for the meetings in which it is a participant. This preference is denoted as a numeric value. For example, if an agent does not "like" meeting *A*, his value for this meeting can be 10. If another meeting *B,* is his "favorite" meeting, this can have a value such as 100. These "preferences" are called *meetings utilities.* Accordingly, every agent has preferences for the *time interval* (or *timeslot*) in which any meeting can happen. These preferences are called *time interval utilities* and they are numeric values, too. A time interval is denoted with a number such as $\{0, 1, 2, 3,...\}$ and represents a real time interval. For instance, time interval 0 can represent the real 8:00 - 9:00 time interval.
- $X = \{x_1^1, x_1^2, \dots, x_m^n\}$ is a finite set of variables, where *m* is the total number of agents and *n* is the total number of meetings, with $n \geq m$. It is not necessary to exist $n \times m$ variables, but variable x_m^n is possible to exist. Each agent can handle one or more variables, and every variable belongs only to one agent. Each variable x_i^h represents a meeting *h* which needs to take place in a certain *time interval* and agent *i* is a participant. Every agent has to find a value for all of his variables.
- $D = \{d_1^1, d_1^2, \dots, d_m^n\}$ is a set of finite domains of the variables in X, with d_i^h being the domain of variable x_i^h.
- $R = \{r_1, r_2, \dots, r_p\}$ is a set of relations. Each relation r_z is a function which determines the amount of *utility* of every possible combination of variables values, which are part of the relation. The definition of such a function is: $d_i^{1,1} \times d_i^{1,2} \times ... \times d_i^{m,n} \rightarrow \Re^+$. For instance, whether a relation r_z has a domain equal to $d_z^{1,1} \times d_z^{2,1} \times ... \times d_z^{3,1}$, this means that

variables x_1^1, x_2^1, x_3^1 are involved in the relation r_z. In this example, $d_z^{2,1}$ denote that the variable x_2^1 which has a domain$d_2^1$is part of the relation r_z.

Supposing that every domain d_i^h is defined from a table with only one row of values such as $\{0, 1, 2, 3, 4, \ldots\}$, where every single value represents a time interval, a combination of two domains e.g. $d_1^1 \times d_2^1$ represents a table with dimensions equal to the product of the number of values of d_1^1 and d_2^1. Each cell in the one-row-table of every d_i^h is calculated as the product of the *utility of meeting* which is represented by variable x_i^h, with the *utility of the certain time interval*, which is represented by the index of the certain cell of the table. In a two dimension table, the value of each cell is the summary of the two products, respectively. The kind of relations can be differentiated into two categories: A) *Exclusion* relations, B) *Equality* relations. We can find in literature this *relation* term, defined also as *restriction.* In an equality relation only the cells whose pointers are equal have value calculated by the summary of the products as it was mentioned above. In any other case, the value is zero. This happens because in an equality relation participates every variable, which represents the same meeting for different agents. Thus, a zero value indicates that a meeting cannot take place in different times for the different agents which participate in this meeting. On the other hand, in an exclusion relation, only the cells whose pointers are different have non zero values. This happens, as in an exclusion relation it participates every variable which is handled by the same agent and represents the different meetings of this agent. Therefore, each meeting in which an agent is a participant cannot take place at the same time and this is what a zero value represents in an exclusion relation.

The final scope of DCOP in this kind of problem is to maximize the overall utility of a global objective $R_g = \sum_{i=1}^{p} r_i$ by assigning a set of values to all variables which satisfy the above exclusion and equality relations. An assignment σ_i is a set of values for the variables which are part of relation r_i. The variables which are part of relation $r_i$are also referred to as the *scope* of this relation. The global assignment *σ* which maximize the global objective is expressed as:

$$\sigma^* := argmax_{\sigma \in \Sigma} R_g(\sigma) = argmax_{\sigma \in \Sigma} \sum_{r_i \in R} r_i(\sigma_i)$$

where *Σ* is the set of every possible assignment *σ*.

2. Description of algorithm implementation

In this section it is described the DPOP algorithm at length which is the basic algorithm we deal with in the DCOP problem. In general, we can describe this algorithm as a three step process. Firstly, we need to establish the pseudotree structure, which is an arrangement of a constraint graph *G* as a rooted tree. This tree has the same vertices as *G* and the properties of the adjacent vertices from the original tree fall in the same branch of the tree. A pseudotree consists of tree edges. Except for the tree edges, that are visualized as hard lines, our graph includes back edges. Back edges, even if they do not compose a formal part of the graph, are significant for the data management between vertices and are represented as solid lines. Pseudotrees can be represented by parent nodes, children nodes, pseudo-parents nodes and pseudo-children nodes. After creating the pseudotree, the next phases consist of *UTIL* and *VALUE* propagation procedure. *UTIL* propagation starts from the leaves of the tree and propagates up the pseudotree through tree edges. *UTIL* messages are sent by any subtree node/child a_i variables to its parent $a_j$variables and when dealing with DPOP these messages can have multiple dimensions. When all subtrees below the level of a node a_i are disjoint by summing *UTIL* messages from all his children, a_i computes total utility derived from each subtree. Finally, through *VALUE* propagation the node depicted from a root node a_i picks optimal values and passes it onto its children. *VALUE* messages contain also the values of all the variables that were present in the context of a_j's *UTIL* message for its parent.

The pseudocode which describes the general form of DPOP in publication [2] is the base for our implementation of the algorithm that we'll describe in detail next.

In this point we should refer to a basic difference between our problem and this which is described in [2]. In our meeting problem every agent has multiple variables in contrast with the problem in [2], where every agent is a variable each self. This discrepancy leads the variables which are handled from the same agent to send messages among them in a little bit different way. Additionally, other differences arise for the same reason and all these are described later in this work.

2.1 Generator

In order to define the problem we should firstly describe the generation process which is a crucial step in solving the DPOP problem. As a generation process we define the procedure during which a hierarchy of agents is created as it is presented in the hierarchy tree of paper [3]. This hierarchy is created based on a given number of agents. Each generated agent has a specific *ID* (e.g. 1,2,3…), a set of children based on the level of hierarchy, a set of siblings and one single parent. Each agent a_i is linked with its children $C(a_i)$. With this last symbol we refer to the children of the node a_i. Technically speaking, in order to create children for each

agent, we use a variable which is described as a multiplier which depends on the level we are and characterizes the number of children, and a list which stores its children. For example, if we are in the second level an agent should have three children. The number of children that each agent must have is calculated by the formula $number\ of\ children\ =\ level\ +\ 1$. Also, we assign siblings to each child in a similar way.

The next part of the generation process is to produce a set of meetings. Meetings are divided in three categories: *SIB* (Sibling meetings), *PTC* (parent to child meetings), *GRP* (group meetings). A *GRB* meeting is defined as a meeting between a node-agent of the hierarchy tree and all of its children. A *SIB* meeting is defined as a meeting between a node-agent and its siblings. A *PTC* meeting is defined as a meeting between a node-agent and one of its children. Each agent who is a participant of a meeting, has a unique variable for this meeting as it is mentioned in the previous section. Furthermore, each agent has a *preference* or in other words a *utility* for each of the time slots at which it is available to participate in a meeting. An agent may prefer to participate in a meeting at later time slots, earlier time slots or may be indifferent. Utility values for each time slot are chosen randomly based on the previous assumption. This means that it is randomly chosen the time preferences of an agent. In order to achieve this, three lists are created, one with values [10, 10, 10, 10, 10, 10, 10, 10] for an indifferent agent, one with values [80, 70, 60, 50, 40, 30, 20, 10] for an agent who prefer earlier time slots and vice versa for an agent with preference in later time slots. In the same way, the preferences of an agent for its meetings are chosen randomly. The number of meetings in which an agent is a participant cannot be more than 8 for each agent.

At this point, it is worth noting that we created our own generator based on a published generator of another team. After the creation of our generator, we also published it, in order to be available for experimentation and inspiration from all the other teams. Below, it is described how the generator works.

Firstly, it should be mentioned that each meeting is created by an agent. This means that whether an agent has already created a meeting, it cannot create another, but it still can participate in other meetings. They begin with the creation of *SIB* meetings from right to left in each level and bottom up. Then, the same process is performed for *PTC* meetings with the constraint that in a given set of siblings only one agent can produce a *PTC* meeting. This creates in total *X SIB* and *Y PTC* meetings. The final step results in creating *Z GRP* meetings from top to bottom and from left to right. If an agent has more than 8 meetings the process for the given meeting is aborted, as the agent is fully linked. The values of *X, Y, Z* should be equal or almost equal.

The whole process which is described above was not randomly chosen. Plenty of experiments were performed in order to solve several problems. A common problem for different generation procedures, was that the produced constraint graph was disjunked. As a constraint graph we define the graph where the variables of each agent are vertices linked by edges, which represent the relations-constraints among them. Another problem that we met during the creation of our generator was that all the agents of hierarchy had to be participants in at least one meeting. With the above procedure this problem is solved.

Regarding the number of total meetings each type of meeting should have, we made some experiments and we concluded that if we deal with a number of total meetings which leads to a decimal number when divided by 3, we should perform a rounding. If the rounded number is less than the decimal number then we add an extra *SIB* meeting, otherwise we remove one *PTC* meeting. This ensures for another time that the produced constraint graph will not be disjunked and all agents will be participants at least in a meeting. Our generator has been tested in producing 50, 100, 200, 300 and 550 agents with the half number of meetings for each of these numbers of agents.

For the visualization of hierarchy trees, we imported manually *pyrgraphviz 1.5* library and we present below a result for 20 agents, because such a tree with more agents is immense to be presented in this document. Respectively, we visualized the results of the constraint graph.

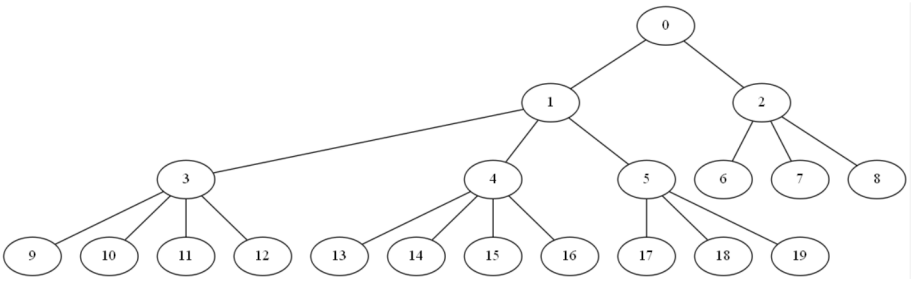

Image1: Hierarchy tree for 20 agents

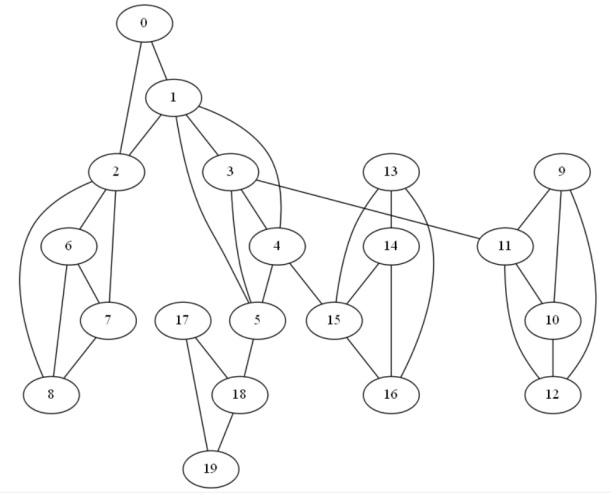

Image 2: Constraint graph for 20 agents

In the above constraint graph, we can see each agent a_i as vertex and their among relations r_z as edges which are defined through their variables x_i^h. In this point, we are differentiated from the constraint graph description of [1], where vertices are the variables. We made this kind of visualization because we consider that is more significant to know which are the paths of real messages. As real messages we define the messages which are between the agents, even though this kind of message consists of sub-messages among their variables. Therefore, an edge of the above graph in fact may be more than one edge, as an agent a_i can have more than one common meeting with another agent a_j and thus might exist more than one pair of variables x_j^h- x_i^h which are related.

The last reference we should make about the generator is that it is implemented in the file *generator.py*. The implementation was held and tested in *PyCharm Professional 2020.3*, in programming language *python 3.8.* The number of agents, for which a hierarchy tree is created and then a meeting schedule problem, can be determined in line 242.

2.2 Initialization of agents and pseudotree creation

After the generation process, we are ready to implement the DPOP algorithm. The implementation of this requires the specification of certain parameters because we assumed that we deal with a distributed solution of the problem. These parameters are: *IP* of each agent and *Port* of each agent. If we had to deal with real distributed agents, we should have different IPs. Since this implementation is performed on a single system, we define one IP which is the internal IP of our system. The differentiation is that one agent has a unique Port (starting from Port: 38001) so this means that each Port has its own messages.This approaches the distributed multi agent systems. We selected the specific port because ports between 38000-40000 are not used from our system.

Afterwards, we should parse the file with the name *DCOP_Problem_N* where *N* is the number of agents. This should be done irregardlessly of the agents' number so we performed a certain transformation in order to read the file. Both the agent's meetings utilities and the time utilities, the number of agents, the number of meetings and the number of variables are stored in initially empty tables. The parsing of the DCOP_Problem_N file was challenging as it required identifying how to differentiate between an agent's meeting utility and an agent's timeslot utilities. In order to do this, the first line of the file reflects the number of agents, the number of meetings and the number of variables. From the second line and onwards we are referring to meetings utilities of each agent and as soon as the agent's ID alters we are referring to time slots utilities. Each type of utility is stored in a list. As soon as this process is done, we can initialize the creation of the agents. An advantage of our method which increases performance and reduces computation time is that we use a pseudo-parallel process which is called *multithreading*. When an agent is created we create a thread and we initialize it. This process is implemented in the file *main.py*.

In order to create and initialize an agent, the class *Agent* from the file *dpop.py* is called, where the whole procedure of the algorithm is implemented. Each agent has its unique ID, its IP which is the same for each agent, its unique Port as described and before, its relations with other agents, its meeting's utilities and timeslot utilities, initially empty lists of its parent, children, pseudo-parents and pseudo-children. The agents also have a neighbour list, which defines the agent's neighbours, as depicted in the constraint graph. In the beginning, each agent creates an extra thread in order to listen to its Port and store messages from other agents. This process is linked to another file *udp.py*, where the necessary parts of communication protocol are found. Even though the file's name is UDP, the communication is achieved via TCP/IP protocol. This happens because the initial communication procedure was implemented with UDP packets, but afterwards this changed because it proved inefficient.

In this phase, every agent knows who to elect as the root-agent. This is an assumption we made in order to have an election criterion. Thus, during the initialization each agent knows whom agent to elect. This is the agent with ID = 0 and Port = 38001. Since they have the same preference towards election so the root-agent waits its election till the number of messages it receives is the same as the rest of the agents. When that happens, the root-agent initiates the creation of the pseudotree.

Now, every meeting for each agent is stored in a list. Thus, the root-agent is able to recognize the relations of the agents. Pseudotree is developed through the *DFS* function which is implemented in *dpop.py*, too. This function which is based on *DFS* (*Depth-First Search*) algorithm, starts the search from a node and continues as deeply as possible. We decide to make a visualization of the pseudotree based on agents and not on variables, as for the constraint graph. We faced the problem of these graphs as a macroscopic approximation of view because this also happened to DFS function. With this last reference, we mean that during the traversing of the constraint graph, only the *equality* relations between agents and their variables were taken into consideration and not the *exclusion* relations of the variables of each agent. With this approach we deem that we simplified the problem without changing the overall operation and efficiency of the algorithm. However, it has to be shown why this approximation does not affect the overall procedure. For this purpose, we made an example of an agent with ID=1 (a_1) and three variables x_1^1, x_1^2, x_1^3.

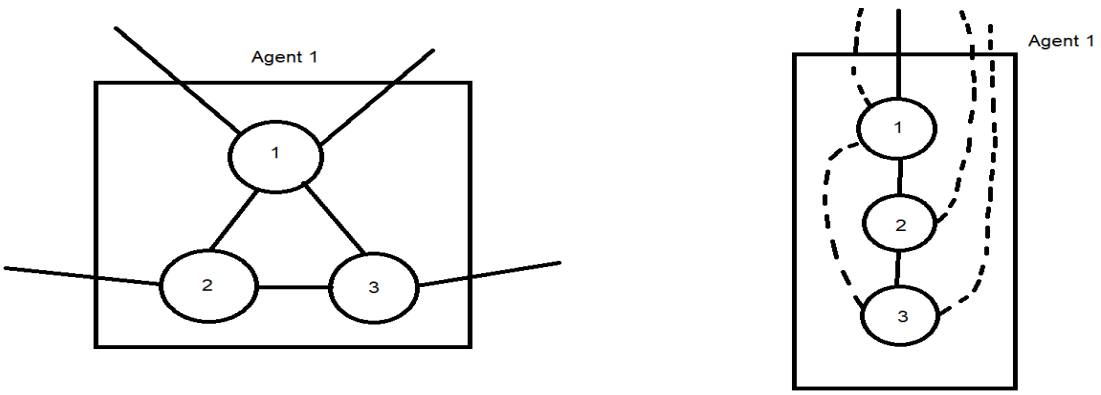

Image 3: *At left*: Constraint graph only for the part of a_1. *At right*: Pseudotree only for the part of a_1.

In this example, a_1 is a participant to meeting 1 where two other agents are also participants. Additionally, it has the meeting 2 with another agent and the meeting 3 in which there is one more participant. Thus, meeting 1 has totally 3 participants, meeting 2 has 2 and meeting 3 has also 2. At the above image, the right drawing represents the constraint graph which focus on a_1. Agent is typified as a square, each of its variables as a cycle and the relations of its variables with the variables of other agents are presented as edges which lead out of the square. Moreover, we can observe that all of the agents' variables are connected because of the *exclusion* relation. Whether we apply a DFS we have to think about two main possibilities: A) when a variable of agent $a_1$is visited, then the other two are visited one after the other, B) when a variable of agent $a_1$is visited e.g. x_1^1, then a variable of another agent is visited with which there is relation with x_1^1. In the second case, there is a chance of visiting the variable x_j^1 of the agent $a_j$which participates in meeting 1. If this agent has a common meeting *d* with the agent a_k and this agent a_k is the one who participates in meeting 2 with agent $a_1$then there is a problem. The problem is that DFS will visit the variables with the order $x_1^1 \rightarrow x_j^1 \rightarrow x_j^d \rightarrow x_k^d \rightarrow x_k^2$ resulting to a pseudotree where at least the variables x_1^1, x_1^2 of $a_1$will be in the same branch, but with variables of other agents between them. This means that $a_1$will send more than one *UTIL* message and whether this happens to other agents the overall number of messages will increase. As the scope of DPOP algorithm is to reduce the congestion of a network, the last case should not happen. This is the reason why we consider the *(A)* case as the optimal which leads to a pseudotree like the right drawing of Image 3. Taking into account this consideration, DFS only needs to traverse the constraint graph with the agent as vertices with respect to the relations between their variables.

Therefore, as every agent includes its variables and these variables are connected with *exclusion* constraints, all agents will have an internal pseudotree structure like Image 3. As a result of the connection of variables, each agent remains in the same branch in the pseudotree with the other agents with whom it has meetings in common, and it is possible to communicate with these. In order to decide how to administer the above problem we conducted experiments and we ascertained the maximization of the performance of the algorithm and the minimization of the dimensionality of messages when the nodes of the pseudotree and the constraint graph are agents and not variables, given the internal structure of each agent variables. Regarding the minimization of messages dimensionality, it is analyzed in the section of *UTIL* message.

Furthermore, we have to mention that during our analysis for additional potential risks we faced the significant problem of the unconnected graphs. In this probable case, groups of agents are unconnected with the main graph creating a number of smaller graphs. In order to find an innovative solution for this case we decide to create secondary pseudotrees from smaller graphs and connect these parts of secondary pseudotrees with the main. So as to develop an optimal solution, we do not connect randomly every secondary pseudotree with the main but at the nodes-leaves with the least connections in order to avoid the increase of

the dimension of the messages. It has to be noted again that our generator does not create unconnected graphs for the examined numbers of agents which are referred above.

Next, we present the pseudotree which was created based on the constraint graph of Image 2:

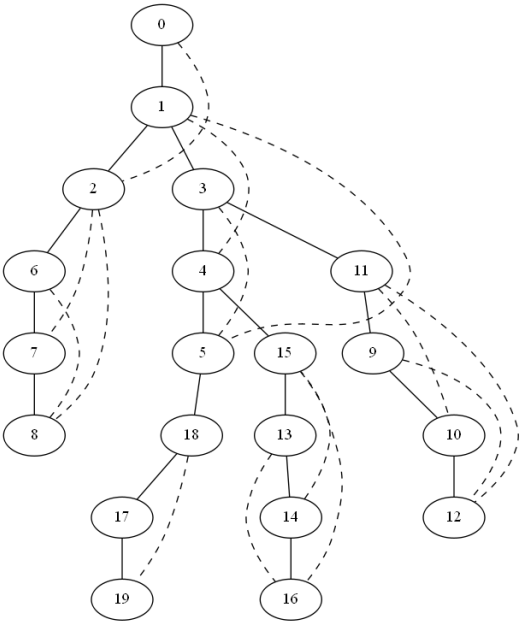

Image 4: Pseudotree for 20 agents.

As we can observe, except for the tree edges, that are visualized as hard lines, pseudotree includes back edges. Back edges, even if they do not compose a formal part of the constraint graph, are significant for the data management between vertices and are represented as solid lines. Finally, we should mention some definitions about the pseudotree which are significant for rest of this work:

As **a parent of a node** is the node that is communicated with a hard line directly and is the only one higher in hierarchy.
As **children of a node** is the total of nodes that is communicated with a hard line directly with the node and is in a lower layer than the node.
As **pseudo-parents of nodes** are the total of nodes that are communicated with a dotted line (back edge) directly and are in a higher hierarchy than the node.
As **pseudo-children of nodes** are the total of nodes that are communicated with a dotted line (back edge) directly and are in a lower hierarchy than the node.

2.3 UTIL Propagation

As soon as the root-agent creates the pseudotree and knows all the relations (pseudo-parent, pseudo-children, real parents, real children), it sends a message to all others regarding their relations. After receiving their relations the agents send to their children and pseudo-children a message indicating their domain. This happens because in

this project we set up the same domain, but in other studies this may not be the case and that's why we have also included this step. Then the *UTIL* propagation algorithm can be initiated. The first case is when a node is a leaf. This is defined when a node hasn't got children. All others wait until the util message is sent by the leaves. In this case, then, a leaf node-agent starts calculating the utilities of each variable using the formula: $util_t^m(a_i) = util_m(a_i) \times util_t(a_i)$, where *m* is a meeting and *t* is a time slot of the agent domain. After calculating all utilities of all the variables of the agent, all the possible combinations of all variables are computed, instead of sending messages among its variables. With this implementation we simplified the problem of messages distributivity among the inner variables of each agent. We decided to follow this approximation in order to minimize the computing power that is necessary for the computation of *real-part* and *pseudo-part* of messages. As the-real part is meant to be the combined utilities for the real-parent of a variable, and as the pseudo-part, the combined utilities for its pseudo-parents.

Then, the agent finds out which ones are the best, for each one of its variable timeslots. Here lies the advantage that it is not required to send all the messages for the inner structure of each agent, instead we calculate directly the best combinations of their domain. For example, if the variables domain is from 0 to 7, the combinations of the variables of *Image 3*, will look like [(0,1,2), (0,1,3), …, (1,0,2), (1,0,3), …, (1,2,0), ...]. Technically speaking, the order of the variables plays a major role so as to know in which variable corresponds each timeslot of a combination. In this example, the first number of each combination corresponds to the variable x_1^3(agent a_1, meeting 3), the second number to the variable $x_1^2$and the third number to the variable x_1^1. The total utility value of a combination is calculated by the summary of each utility of the combination parts. The utility of the combination (0,1,2) can be computed as $util_0^3(a_1) + util_1^2(a_1) + util_2^1(a_1)$. As we can see, a combination of this kind cannot consist of the same timeslots, because the different meetings in which an agent is a participant, cannot happen at the same timeslot.

After having calculated all the combinations utilities, the algorithm continues with the computing of the best timeslots utilities of its domain for every variable. This is accomplished by picking up the best combination for each variable timeslot. For instance, the best utility for time slot 0 of x_1^3 is the one with the greatest utility between all the combinations which starts with 0, as the following: (0,1,2), (0,1,3),..., (0,7,6). Therefore, this kind of structure looks like an opened, flatted hypercube.

Afterwards, each variable combines its best calculated utilities with those of its real-parent variable, which is the corresponding variable of the real-parent node-agent. The combine or *join* of these utilities is done as it is described in detail in [4], as hypercubes, but in our case again as opened, flatted hypercubes. The difference is that we penalize a cell of the hypercube with zero if its values are different. For example, if a cell of a two dimensional hypercube is indexed by (0,1) this means that it has to be calculated as the summary $util_0^m(a_i) + util_1^m(a_j)$, but in our case this is equal to zero. This penalty indicates that a meeting *m* in which agents a_i and a_j are participants has to take place in a common timeslot.

In this way, the total number of hypercube cells is reduced, because we just exclude all the zero-utility combinations, and so is the size of an agent message.

When all variables values of a specific agent-leaf have combined with their parent-variable values, they create a real-part agent-level message as a *python dictionary* with all these values separately per variable.

After that, they combine these values with their pseudo-parents variables values just as described previously and create a pseudo-part agent-level message. In this manner, the real-parent variables values are combined with pseudo-parents variables values just as described in [2]. It should be mentioned that these combined utilities are not penalized because they are referred to agents with different meetings in common. A discrepancy with paper [2] is that every pseudo-parent variable values of an agent a_i is combined with all the variables-meetings of its real-parent agent a_j which are in common between them. Thus, we achieve to have a fully updated node for all of its subtree at the agent-level kind of node. An additional discrepancy of our implementation compared with paper [2], is the following: whether a variable x_i^m sends a pseudo-part message to one of its variable pseudo-parents x_j^m, and another variable x_h^m of the same branch with x_i^m also sends a pseudo-part message to x_j^m, then the pseudo-part messages of x_i^m, x_h^m variables are combined as an aggregation only of their common domains. For our meeting scheduling, this means that only the utilities of combinations (0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (6, 6), (7, 7) are computed. It is a kind of penalization for all the other combinations, because their variables which referred to their common meeting, should assign the same timeslot. As a result, the *size of the message* which a variable sends is highly reduced as well as the size of the agents' real messages. This leads to a decrease of the necessary calculations which means that the time needed for an experiment to run is extremely short.

When all the pseudo-parents variables' best utility values are calculated, then the agent creates the pseudo-part message. The last part of its message is the inclusion of the *number of messages* it has received in order to be able to calculate the total number of messages at the end of the algorithm. It is apparent that for an agent-leaf this number is 0. Then, every agent-leaf sends its message to its real-parent agent.

Moving forward, all the agents of the higher level who are waitting the messages of their children-agents, now receive those messages. They apply the same technique as agents-leaves, but with the difference that they have to combine the information from their children. Thus, at first they combine the utilities of their real-children in the variable-level. Then, they combine the utilities of their pseudo-children, again in the variable-level, and they store this information in an array in order to use it at *VALUE* propagation phase. Thereafter, each variable reduces a dimension of the hypercube for each pseudo-part of the message from every variable of each agent-child. This is achieved by calculating the best utilities of its domain from all the combinations and excluding all the rest. For instance, if a pseudo-part of a message which is referred to a specific variable has 3 dimensions, as combinations which consist of 3 numbers, then this variable will reduce these dimensions to 2, as combinations with 2 numbers. Whether a pseudo-part of message is not referred to any variable of the

agent which received this message, then this means that this message was sent from an agent of a lower level than its children-agents and it goes to an agent of a higher level. This kind of message will remain untouchable and will be forwarded to its parent-agent.

Then, the same procedure as that at agents-leaves is realized for the calculation of real-part, pseudo-part and message-size-part and the message is sent to the parent-agent. This procedure continues until the root-agent receives the message from all its children-agents.

2.4 VALUE Propagation

At this stage the root-agent waits till it receives the *UTIL* messages from its real-children. As soon as it receives them, it stores them in a list and it begins combining these utilities from the real-children and pseudo-children. The next step is to calculate the optimal utilities regarding its domain for all variables and based on the combined utilities. Afterwards, it stores all the combinations in a list with the purpose of finding the best combination for its variables which holds the maximum utility value. These combinations have the form that was stated in the *UTIL* propagation phase. Thus, the selection of a combination indicates the assignment σ_i of timeslots it will apply to its variables.

The next step is to prepare a *VALUE* message and to inform its children regarding the timeslots of the meetings it has decided to process. For example, if the root-agent and one of its children participate in the same meeting, then the root-agent informs its child via a *VALUE* message about the timeslot their common meeting will occur. This operation is very significant for the optimal selection of global assignment *σ** in order to maximize the global objective function R_g, as each assignment σ_i is chosen based on a well informed utility hypercube which consists of the best utilities of all meeting participants. The *VALUE* messages as referred also in the previous section, consist of a part that is related to real-children -for the common meetings between real-children- and a part that is related to pseudo-children, and their common meetings. Therefore, each message which is sent to a specific agent child a_i, contains the values that are determined for the common meetings with this child and for those with pseudo-children, which are in the same branch of pseudotree with agent a_i.

Then, root-agent-children and whoever is below the root-agent's structure, wait to receive the *VALUE* message from their parents. As soon as the message arrives, it observes the selected timeslot for a specific meeting and stores it. In this manner, a real-parent and a pseudo-parent could not decide different timeslots for a common meeting because the agent who is in the highest level between them in the hierarchy of the pseudotree is the one who takes the decision. This decision is not arbitrary since this agent knows all the utilities of the agents who are in a lower level and have the common meeting. If an agent is leaf, waits for its message with the already defined assignment of its meetings, receives it, stores the selected timeslots and the algorithm ends in this point for this type of agent. Otherwise, it should receive the values and check in the *UTIL* list of stored combinations with the best utilities, it removes the specific meeting from the list so as not to calculate an already given value.

Whether there are variables for which an agent has not received a *VALUE* message from its real-parent/ pseudo-parent, then it is meant that these variables refer to meetings which are in common only with some of its real-children or/and pseudo-children. For these variables, the agent is looking at the retrieved *UTIL* list in order to discover the optimal combination of timeslots as did the root-agent. When it finds this combination, it constructs the *VALUE* messages for its children, each one containing only the corresponding values of the optimal combination, and it sends it to them. This procedure continues until every agent leaf receives the message from its real-parent.

It is significant to mention how the max *size message* of the results in the above section is calculated, because it is relevant with the *VALUE* propagation phase. Every agent together with the *VALUE* message also sends to the root-agent a message with two parts. One part consists of the assignments decided by the agent who sends the message. This is done so that the root-agent prints all the assignments in ascending order of meeting IDs at the end of the algorithm. The second part of this message consists of the message's size of the *UTIL* message which the agent has sent in the *UTIL* propagation phase. In this way, the root-agent receives the message size of all agents and thus it has the ability to find the *max message size*.

3. Experimental evaluation and results

Before presenting our results, we should define how some arguments are calculated such as the *constraints number* as well as the *cycles* and the *messages number*. The constraints number is calculated as the aggregation of *exclusion* constraints and *equality* constraints. *Exclusion* constraints of an agent a_i are calculated based on the below equation:

$$\binom{n}{2} = \frac{n!}{2!\,(n-2)!}$$

where *n* is equal to the number of the agent's variables. The total number of the *exclusion* constraints of every agent, gives us the total number of *exclusion* constraints.

Equality constraints of a meeting *M* is calculated also based on the above equation, where *n* is the number of the agents participating in this meeting *M*. The total number of the *equality* constraints is the aggregation of all meetings *equality* constraints.

In addition, cycles are calculated as the number of the back-edges between the agents as in the example of *Image 4*. We should state that our implementation differs from paper [2], where the number of *cycles* is calculated as the sum of back-edges between variables. We follow this approximation in order to be aligned with the visualizations of pseudotrees which are agent-level based constructions.

Finally, the *messages number* is based on the real messages that agents send using TCP/IP protocol. This aspect of *messages'* number was selected to highlight the real traffic that our implementation produces in the generated network.

In the table below we adduce the results of eight experiments:

Agents	9	50	100	200	300	400	500	550
Meetings	5	25	50	100	150	200	250	275
Variables	15	89	200	426	675	929	1176	1304
Constraints	35	235	580	1318	2237	3187	4098	4577
Messages	16	98	198	398	598	798	998	1098
Max message size	1552	2576	3600	3600	4624	4624	4624	4624
Cycles	5	58	153	359	598	863	1109	1238

Table 1: Results

In this point, we should mention that the results with the timeslots assignments of each meeting for every experiment are included with the code files in folder *Results*. Taking into consideration these assignments, we could analyze from the above results the example of 9 agents. In this example, we observe that our results are aligned with the maximization of global objective function. Below we adduce the steps in order to make this conclusion:

- **Meeting 0:** In this meeting, the agents a_6, a_7, a_8 are participating. Agent a_6 prefers later meetings, agent a_7 prefers later meetings and agent a_8 prefers later meetings. Agent a_7 and a_8 have the same utility for meeting 0, which are higher than agent a_6, so they contribute the same to the overall utility of the final assignment. Taking into consideration that the agents prefer later meetings, the assignment of timeslot 7 is a strong indication of the fact that the algorithm implementation is aligned with the scope of global objective function maximization.

- **Meeting 1**: The agents a_2, a_7 participate. Agent a_2 is indifferent regarding the timeslot and agent a_7 prefers later meetings. Because agent a_7 has less utility from meeting 1, the assignment to timeslot 6 is right for the maximization of overall agents utility.

- **Meeting 2:** Agents a_3, a_4, a_5 are participants and agent a_3 prefers earlier meetings. Agent a_4 prefers later meetings and agent a_5 prefers earlier meetings. Agent a_3 has utility 100 for meeting 2, agent a_4 has utility 70 and agent a_5 has utility 10. As a result agent a_3 contributes the most. Therefore, the assignment of timeslot 0 seems to be logical since agent a_3 prefers earlier meetings. Any other assignment would reduce the total utility in comparison with this

of timeslot 0.

- **Meeting 3:** The agents who participate are a_0, a_1, a_2. Agent a_0 prefers earlier meetings, agent a_1 prefers earlier meetings and agent a_2 is indifferent. As agent a_1 has utility value for this meeting equal to 100, agent a_0 has value 10 and agent a_2 has value 70, we should conclude to the assignment which fits better to agent a_1 preference and maximizes the overall utility. The final assignment of timeslot 0 is logical as agent a_1 has only one other meeting with less utility.

- **Meeting 4:** Agents a_1, a_3, a_4, a_5 are participating. Agent a_1 prefers earlier meetings, agent a_3 prefers earlier meetings, agent a_4 later meetings and agent a_5 earlier meetings. Agent a_5 has utility 100 for meeting 4, agent a_4 has 50, agent a_3 has 50 and agent a_1 has 100. Thus, the agents who prefer earlier timeslots contribute with a percent $(100 + 50 + 100)/(100 + 50 + 50 + 100) = 250/300 = 0,833 = 83,3\%$ to the overall utility of the meeting assignment. The assignment which was set on timeslot 1 is aligned with the maximum contribution as timeslot 0 is set for meeting 3 where agent a_1 is a participant and for meeting 2 where agents a_3 and a_5 are participants. Agent a_1 has a greater utility for meeting 3, thus the assignment of the later timeslot to this meeting maximizes its own utility. Agent a_3 also has a greater utility for meeting 2 than meeting 4, but agent a_5 has a lower utility for meeting 2 than meeting 4. In this case, it was realized a balancing among the utilities of these three agents and their common meetings, and the optimal one assignment was selected.

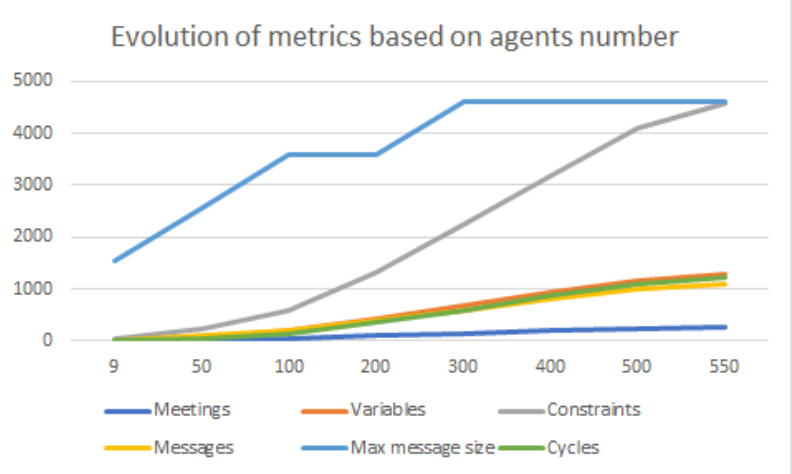

Image 5: Evolution of metrics depending on agents number

From *Table 1* and the relevant plot (*Image 5*), we observe that as the *number of agents* increases, the *number of meetings*, *variables*, *constraints*, *messages*, *max message size*,

and *cycles* increases also. The *max message size* increases till agents number 100 and afterwards it remains stable at two levels (3600, 4624). This happens because from agents number 9 to 100 the number of back-edges increases from 1 to 4, from agents number 100 to 200 back-edges remain stable and from agent number 300 to 550 they reach at most to 5 back-edges. The number of back-edges can be observed from the constructed pseudotrees. All these visualizations are stored in folder *pseudotree.* Back edges behave with the aforementioned way, because the generator creates the meetings with a certain way which is explained in section *2.1*. As a result, since the number of back-edges remains stable, the *max message size* remains stable, because the message size of an agent is based on its pseudo-parents increase.

Regarding the message size, this is not as high as the one indicated in paper [2], but instead this is significantly reduced. We have already mentioned in section *2.3* why this happens, and it is mainly due to the way joins are implemented for each agent's pseudo-parents utilities. Because of this differentiation, we cannot prove that the assignment σ^* which our algorithm achieves is optimal, but we have a strong indication that this happens based on the analysis of the previous example. In any case, we achieve a very low message size which is important taking into consideration that the purpose of DPOP is to optimize the congestion in a network.

Before the ending of this section, it should be noted that all the visualizations of *hierarchy trees, constraints graphs* and *pseudotrees* of the above mentioned experiments are included in the corresponding folder inside the folder *Results.* Also, it is significant to refer that we were unable to run experiments of 1000 agents because our machine, a Windows 10 PC with an Intel i5-4200U processor and a 16GB installed RAM, crashed when it was tried. The major reason is the excessive number of messages which are transmitted into the system and presumably the total number of the simultaneous running threads which exceeds the capacity of our PC RAM.

4. Conclusions - Future Work

In this project, we performed an implementation based on paper [2], adjusting its specifications to our problem. In order to phase this challenge, we tried to implement an out of the box solution and finally we managed to adjust the problem's specifications to our implementation. We produced successful results and our algorithm was structured taking into consideration the fundamental principles of the DPOP problem: *root election, pseudotree creation, UTIL propagation*, *VALUE propagation* and its requirement *generator.* Specifically, our generator produces connected graphs and all agents participate in a meeting, a condition which was a challenge on its own. Concerning *UTIL* and *VALUE* propagation, we observed that execution time given our system's requirement seems low and for the case with the most agents (550) around 5 minutes. In fact, a large part of this time is required to create the pseudotree visualization. Finally, a novelty of this

implementation is that it proposes an efficient solution to deal with *max message size,* which can be further studied and compared with other implementations.

We have already mentioned some challenges that we faced and managed, concerning the comprehension of the problem which has to do with distributed constraint problems and the term of hypercube. We found also challenging the assumptions we had to follow in order to achieve a functional implementation with these requirements. Additionally, the problem had high programming difficulty in terms of multithreading programming side by side with inner system messages over TCP/IP and the general structure of the code for each problem's part.

Finally, we believe that further work can be performed in this domain, in terms of proving that our implementation produces optimal results. This comparison could be initiated by constructing a centralized algorithm which calculates the optimal feasible solution. This will take into consideration all combinations and will reject the ones which do not satisfy the constraints. Afterwards, it will calculate the best one based on the utility it serves and the resulting assignment which should be the optimal one, could be compared with the results of the present implementation.

References

[1] F. Fioretto, E. Pontelli, and W Yeoh, "Distributed Constraint Optimization Problems and Applications: A Survey," *Journal of Artificial Intelligence Research*, vol. 61, pp. 623-698, Jan. 2018, doi: 10.1613/jair.5565. [Online]. Available: https://jair.org/index.php/jair/article/view/11185

[2] A. Petcu, B. Faltings, "DPOP: A Scalable Method for Multiagent Constraint Optimization" in *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI-05)*, *July 30-August 5, 2005, Edinburgh, Scotland, UK,* pp. 266-271. [Online]. Available: https://www.researchgate.net/publication/220816021_DPOP_A_Scalable_Method_for_Multiagent_Constraint_Optimization

[3] R. T. Maheswaran, M. Tambe, E. Bowring, J. P. Pearce, and P. Varakantham, "Taking DCOP to the Real World: Efficient Complete Solutions for Distributed Multi-Event Scheduling" In *Third International Joint Conference on Agents and Multi Agent Systems (AAMAS)*, *July 19-July 23, 2004, New York City, New York, USA,* vol.1, pp 310-317. doi: 10.1109/AAMAS.2004.10067. [Online]. Available: https://www.computer.org/csdl/proceedings-article/aamas/2004/20920310/12OmNvjyxDN

[4] A. Petchu, “A class of algorithms for distributed constraint optimization,” In *Frontiers in artificial intelligence and applications: Dissertations in artificial intelligence*, vol. 194, Amsterdam, Netherlands: IOS Press, 2009.