Chapter 3

Static Knowledge Base Partitioning and Allocation

Our objective in this chapter is to develop a graph-based heuristic for a fc-way heterogeneous partitioning of the knowledge base which overcomes the problems mentioned in the last chapter by using data dependencies. We follow the decomposition approach to obtain a given number of subsets in the specified proportion of sizes. In addition, methods for obtaining functional decomposition and allocating the resulting subsets are also discussed and developed.

The chapter is organized as follows. Section 3.1 gives an introduction and section 3.2 explains the issues involved in cutset based knowledge base partitioning. Section 3.3 discusses the k-way partitioning of a connected knowledge graph. Section 3.3.6 gives the linear time heuristic for static partitioning, analyzes its time complexity and section 3.4 illustrates the heuristic with a few examples. Sections 3.5 and 3.6 deal with disconnected components in the graph, and functional decomposition respectively. Section 3.7 presents a heuristic for allocating the rulebase subsets obtained as above to k agents with a given topology and interagent distances. Finally, the last section presents the conclusions.

3.1 Introduction

Earlier work on static rulebase partitioning for load balancing used techniques like KL graph partitioning [18] and Simulated Annealing [127] for obtaining a homogeneous partition. Considering graph-based approaches, literature on graph partitioning usually refers to vertex partitioning, particularly two-way partitioning or bisection [12, 43, 74]. The techniques are originally developed for VLSI circuit design and even the improvements are oriented towards the same. If more than two subsets, say k subsets are required, it is done by repeated bisection of the graph.
There is no direct procedure to perform fc-way heterogeneous partitioning. However, in distributed (AI) systems, agent capacity and speeds may not be equal. If all the agents are allotted subsets of the same size, the slowest agent will be a bottleneck for the entire network [15].

Further, earlier work on rule partitioning for load balancing considers a single processor or multiple processors with shared memory, as only rules were to be distributed [18, 127]. Data is either centralized or fully duplicated in local memories. But when data is also to be distributed and kept in the local memories of agents which are possibly geographically separated, data updates and inconsistencies associated with rapidly changing data (eg. in a monitoring system) become important. Often the techniques used in other domains ignore data distribution, resulting in run time communication delays while accessing data at remote sites [111]. To remedy these problems, it is necessary to distribute both rules and required data.

We aim at obtaining a heterogeneous partition without repeated bisection such that both rules and data are partitioned using a graph-theoretic approach [98, 99]. Data are represented as vertices and rules are represented as labels on the edges connecting data. As rules are a kind of semantic constraints and can be treated as functional or multivalued dependencies [27, 79], this representation allows us to exploit the dependencies and the adjacency of data and rules in the graph. Our objective then becomes that of partitioning both vertices and edges such that rules are in the given proportion with less communication overhead and data duplication. Further, as part of the partitioning process itself, metaknowledge directories are generated to facilitate distributed reasoning.

For developing formal methods covering various cases of static partitioning and allocation, we shall first give few definitions and discuss how cutsets can be used to obtain partitions from the knowledge graph representing the rulebase.

### 3.2 Cutset Based Knowledge Base Partitioning

We shall use the following example rulebase with six rules to explain the possibilities and issues in partitioning the knowledge base using cutsets.
Chapter 3. Static Knowledge Base Partitioning and Allocation

The left hand side (LHS) of each rule is called the premise part, and the right hand side (RHS) the action part. For simplicity, we have shown only attribute names and boolean and relationship among the attributes in all rules. However, there is no loss of generality because, irrespective of the values of the operands (even if they are not boolean) and the relationship between them, the expressions must be evaluated to proceed with rule enabling and firing. Therefore, it is enough for our purposes to know in which rules an attribute participates in the RHS part and in which rules it participates in the LHS part.

Def 1.1 Knowledge Graph

A knowledge graph \( G = (V,E,L) \) is a directed, acyclic, labelled graph which consists of a set \( V \) of data elements (attributes or objects) as vertices, a set \( E \) of edges such that \( E : V \rightarrow V \), and a set \( L \) of edge labels corresponding to the rule identifiers to which the edge belongs.

To construct a knowledge graph, a directed arc is drawn from each of the attributes in the LHS part of a rule to (each of) the attributes in its RHS part. The rule id is given as the corresponding edge label. The knowledge graph for the above rulebase is shown in figure 3.1.

In an acyclic graph, the indegree of a vertex \( v_i \) is the number of incoming edges incident on \( v_i \). Similarly, the outdegree of a vertex \( v_i \) in an acyclic graph is the number of outgoing edges incident on \( v_i \). The degree or incidence of \( v_i \) is the number of edges incident on the vertex. Table 3.1 gives the indegree, outdegree and incidence pertaining to each attribute in the knowledge graph.

\[
\begin{align*}
R1. & \quad AB \rightarrow C & \quad R2. & \quad BDE \rightarrow F \\
R3. & \quad BG \rightarrow H & \quad R4. & \quad I \rightarrow J \\
R5. & \quad FG \rightarrow K & \quad R6. & \quad AG \rightarrow I
\end{align*}
\]
Figure 3.1: Knowledge Graph for Example Rulebase 1
Chapter 3. Static Knowledge Base Partitioning and Allocation

Table 3.1: Degree information for the Knowledge Graph for Example Rulebase 1

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
</tr>
</thead>
<tbody>
<tr>
<td>Indegree</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Outdegree</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Degree</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Def 1.2 Cutset

In a connected graph $G$, a cutset is a set of edges whose removal from $G$ leaves $G$ disconnected, such that no proper subset of these edges disconnects $G$.

Since a cutset is the minimum set of edges whose removal leaves a connected graph disconnected, it is also called as minimal cutset.

For instance, the set of edges \{GI, BC\} forms a cutset and divides the vertices into two sets \{I, J, A, C\} and \{B, D, E, F, G, H, K\} shown in figure 3.2a. The only rule that is completely associated with the first subgraph is $R_4$; the rules associated with the second subgraph are $R_2$, $R_3$, and $R_5$. $R_1$ does not belong to either of them completely as neither has all the data required by it. Similarly $R_6$ also does not appear in either of the subgraphs.

Def 1.3 Rule Completeness

If $R = \{R_1, R_2, \ldots, R_n\}$ is the set of rules associated with the knowledge graph $G$ corresponding to the complete knowledge base, then the rule subsets $RS_i$ associated with each of the subgraphs $G_i$ must ensure that $\mid R \mid = \sum_{i=1}^{k} RS_i$, where $\mid R \mid$ stands for the cardinality of the set $R$.

In order to achieve rule completeness in the above example, we need to duplicate attributes B and G with the first component also. Now, as shown in figure 3.2b, the resulting subgraphs have their component data sets as \{I, J, A, B, C, G\} and \{B, D, E, F, G, H, K\} and rules sets \{$R_1$, $R_6$\} and \{$R_2$, $R_3$, $R_5$\} completely associated with them. This duplication is similar to the duplication of attributes in files for database referential integrity, and dependency preservation [27, 79]. It indicates the need for keeping copies of data in working memories of agents.
Figure 3.2: Components obtained with cutset \{GI,BC\} for Example Rulebase 1
However, for avoiding the data inconsistency problems, if rules are to be assigned to subsets without duplicating data, some more constraints need to be considered. In this case, after the rule is assigned to some subset, the corresponding agent has to receive the required data from other agents for enabling and firing that rule. Selection of a suitable subset (and hence an agent) for the assignment of a rule is discussed in the next section.

We choose the later option and, hereafter, instead of showing the duplication of attributes in the graph, the information regarding the required data will be stored in the directories (after assigning the rule to a suitable subset). As we shall see in the forthcoming chapters, this metaknowledge is useful to dynamically redistribute the knowledge and to facilitate distributed reasoning.

Mathematically, computation of a cutset is closely associated with a spanning tree.

**Def 1.4 Spanning Tree**

A tree $T$ is said to be a spanning tree of a connected graph $G$ if $T$ is a subgraph of $G$ and $T$ contains all vertices of $G$. (It is also called a skeleton, scaffolding, or maximal tree (subgraph) of $G$.)

A spanning tree has $n - 1$ edges where $n$ is the number of vertices in the original graph. Removal of any edge from a spanning tree separates it into two disconnected parts. The set of solid arcs in figure 3.3 is an example for a spanning tree. The dashed arcs are those which are present in the original graph but not in the spanning tree and are called as chords of that spanning tree. (The meaning and use of integer markings given on edges will be explained in the next section.)

It is quite possible that a knowledge graph is not connected and has components in it. However, we postpone the discussion of disconnected components in the graph till section 3.5 and assume that the knowledge graph is connected till then.

A cutset can be obtained by removing an edge belonging to the spanning tree giving two disjoint vertex (node) sets, and selecting the chords, if any, which connect the vertices in the two disjoint vertex sets. The spanning tree edge and the chords together form the cutset. For example, removal of edge BC from the spanning tree
Figure 3.3: A Spanning Tree for the Knowledge Graph of Example Rulebase
shown in figure 3.3 yields two vertex sets \{A,C\}, and \{B,D,E,F,G,H,I,J,K\}. To obtain disconnected components with the same partitioning of the vertices, chord AI also must be removed. Therefore, the set of edges AI and BC forms a cutset. This is also called a fundamental cutset.

**Def 1.5 Fundamental Cutset**

A cutset \(S\) containing exactly one brunch of a spanning tree \(T\) is called a fundamental cutset with respect to \(T\).

Though the number of subsets (parts) obtained as above using cutsets is two, this method of obtaining cutsets [31] does not bear any relationship with the number of rules required in each subset. Moreover, obtaining \(k\) subsets in the required ratio is not straight forward. In order to develop such a method for knowledge graph partitioning (in the next section), we shall first define the terms semipath, length of a semipath, pendant vertex, spandegree and chain.

**Def 1.6 Semipath**

A semipath from vertex \(v_i\) to vertex \(v_j\) is a path from \(v_i\) to \(v_j\) in the corresponding undirected (knowledge) graph.

In the knowledge graph shown in figure 3.3, JIG is a semipath from J to G. JIGHBFK is another semipath, i.e, from J to K.

**Def 1.7 Length of a semipath**

Length of a semipath is the sum of weights of edges incident in the semipath.

We assume unit weights for edges unless otherwise explicitly mentioned. Length of the semipath JIGHBFK is 6.
**Def 1.8 Pendant vertex**

A vertex with degree 1 is a pendant vertex.

J is a pendant vertex in the above graph.

**Def 1.9 Spandegree**

Spandegree of a vertex w.r.t. a spanning tree is the degree of a vertex considering the edges in that spanning tree only.

**Def 1.10 Chain**

A Chain is the longest semipath in the spanning tree.

The semipath JIGHBFK is the chain for the spanning tree shown in the figure 3.3.

Now we shall classify the spanning tree edges into two types: those which lie on the chain, and those which do not. We call the former type of edges the chain edges, and the latter type of edges the branch edges. A branch emanates (in the corresponding undirected graph) from a vertex whose spandegree is greater than 2. Semipath BCA is a branch on the chain JIGHBFK.

A branch in turn may have subbranches in it. The number of branches of a branching vertex is called its branching factor. Branching factor of vertex B is 1. For a branching vertex lying on the chain, branching factor is 2 less than its spandegree. For example, if a vertex has a spandegree of 4, then it has two branches.

### 3.3 k-way Partitioning of the Knowledge Graph

The proposed k-way partitioning consists of two phases: initial decomposition and boundary refinement. In the first phase, knowledge graph is initially cut into k disjoint components by cutting the graph at k - 1 places using spanning trees and cutsets. This involves determining the position of the cut and an approximate partitioning of the rules and data. In the second phase, boundaries are smoothened...
to get a required partitioning leaving the inner portions of the subsets undisturbed as far as possible.

To enable partitioning, initial decomposition phase requires

• the generation of a spanning tree from the knowledge graph (constructed as described in section 3.2) and

• marking of its edges suitably.

The same spanning tree, chain, and edge markings can be used to obtain partitions in any desired proportion. However, partitions obtained using different spanning trees may be of different sizes and exhibit different characteristics. Therefore, for obtaining a particular partition, selection of the spanning tree and determination of the cutsets are non-trivial. Since, exhaustive enumeration is not practical, we propose a heuristic for this.

3.3.1 Generation of a Spanning tree

Since a spanning tree encompasses all the vertices, it has a large number of rules on its edges. A spanning tree with a long chain facilitates partitioning due to its linearity. Therefore, we consider generating a spanning tree with a long chain.

Selection of starting vertex

We choose either a pendant vertex or a vertex with zero indegree or zero outdegree as a starting vertex, to generate a spanning tree with a long chain. We shall call this the root of the spanning tree.

There will be at least one candidate root in any knowledge graph as all external input data items will have zero indegree, and all final result attributes will have zero outdegree. A vertex with degree 1 may be preferred as candidate root because this will not have another edge incident on it, and hence it is quite possible that this may be forming one end of the chain.
**Inclusion of edges**

Since every spanning tree has a chain, identifying a spanning tree with the longest chain is computationally intensive. Hence the following heuristic is adopted. Starting with the root, edges are included in the spanning tree such that spandegree of vertices is kept less than or equal to 2 as far as possible. This heuristic gives a spanning tree with a reasonably long chain. Considering the root as the current vertex, an edge incident on it and the other end vertex (adjacent vertex) are included in the spanning tree if the adjacent vertex is not already in the spanning tree vertex list. The newly added vertex now becomes the current vertex and the process is repeated with the new current vertex until \( n - 1 \) such edges covering all the vertices in the spanning tree are included.

The following procedure gives the detailed steps involved.

```plaintext
procedure generate_spanning_tree();
(* generate a spanning tree with a long chain for a connected knowledge graph *)
begin
initialize spandegree of all vertices to zero;
select a vertex \( v_i \) as root;
add \( v_i \) in open list at the end;
add \( v_i \) in the spanning tree vertex list;
k := 2;
while \( k <= n \) and (open list not empty) do
begin
    if 3 an edge \( v_i v_j \) or \( v_j v_i \) in the knowledge graph such that
    \( v_i \) is not in the spanning tree vertex list already then
    (* if \( v_i \) is already in the spanning tree vertex list, mark the edges unsuitable for further consideration *)
    include the edge in the spanning tree edges list;
    include ruleset \( RS_e \) on the edge in the spanning tree rule set;
end
end
```


include \( v_t \) in the spanning tree vertex list;

\[ k = k + 1; \]

increment \( \text{spandegree} \) of \( v_t \) and \( v_i \);

if \( \text{spandegree} \) of \( v_t \) is equal to the degree of \( v_i \) then

delete \( v_t \) from open list;
endif;

if \( \text{spandegree} \) of \( v_t \) < degree of \( v_i \) then

add \( v_t \) in open list;
endif
else

delete \( v_i \) from open list;
endif;

if open list not empty then

let \( v_i \) be the (most recently added but undeleted) vertex at the end of the open list;

(* else if open list empty and \( k < n \) then there are disconnected components in the graph

select a new \( v_i \) as root from the rest of the vertices in the knowledge graph;

mark the previous component with its rules and number of rules for indexing;
endif;*)
endif;
end; (* end of while*)
end; (* end of procedure *)

### 3.3.2 Finding Chain

Though a spanning tree has \( n-1 \) edges of the original graph, there need not be only one single semipath in the spanning tree. Also, the (longest) semipath from the root
to the last vertex (included during spanning tree generation) need not be the chain in the spanning tree.

Finding the chain is done in two steps:

1. finding the longest semipath from root, and

2. finding the chain using the longest semipath obtained in step 1.

The longest semipath from root may be found using depth first search starting from root. Lengths of semipaths from a vertex to leaf vertices (which are not in the longest semipath) in spanning tree are stored while backtracking in the search process in order to find the chain in step 2.

The spanning tree can now be divided into three parts, viz., the longest semipath portion $sp_1$ from root to the first branching vertex, portion $mp$ from the first branching vertex to the last branching vertex including all the branches incident on each branching vertex, and the portion of (the same) longest semipath $sp_2$ from the last branching vertex to the last vertex in the longest semipath. Each branching vertex will have first and second longest branches $bp_1$ and $bp_2$ if its branching factor is at least two. Let $l_1, l_2, b_1$ and $b_2$ be the lengths of $sp_1, sp_2, bp_1$ and $bp_2$ respectively. Now, the longest semipath in the entire spanning tree, i.e., chain, may be found as follows:

If the longest semipath portion till the branching vertex under consideration $sp_1$ is shorter than its first longest branch, the branch becomes part of the chain (longest semipath), and vice versa. If the new branch is shorter than the second longest branch, then these two are interchanged. Adjoining the portion of the semipath from this branching vertex to the next branching vertex, if any, with $sp_1$, the process is repeated for all the branching vertices. The new $sp_1$ adjoined with $sp_2$ will be the longest semipath in the entire spanning tree and hence the chain.

The procedure for finding the chain using the longest semipath from the root is given below.
branch point \( bv_1 \) and the next branch point \( bv \) to \( l_1 \);
make \( bv \) the new \( bv_1 \);
endif;
until the last branch point \( bv_2 \) is encountered;

chain is the semipath \( sp \) \( \text{adjoined with}\) \( sp_2 \);
length of the chain \( l_e = l_1 + l_2 \);
let \( RS_e \) be the set of rules present on the chain edges:
end; (* of the procedure *)

We have already mentioned that many of the rules in rulebase appear on the edges of the spanning tree. Even if there are any rules which do not belong to the spanning tree rule set, this does not affect the final decomposition and the rule completeness. These rules will be included when chords are considered in the cutset — chain is used only to facilitate decomposition.

To determine the actual cutsets, the spanning tree edge to be removed first must be found out for each cutset. In order to identify these, spanning tree edges need to be marked suitably.

3.3.3 Marking edges for decomposition

Integer markings on edges indicating their position in the spanning tree are useful in determining the positions where the graph is to be cut to obtain an approximate partitioning in the given proportion. We give below the steps for marking (labelling) the edges to enable suitable decomposition.
**procedure** mark\_edges();
begin
Mark the edges with 1,2,... to indicate their position
from one end of the chain to the first branching vertex on the chain;
Let the rules corresponding to edges in the chain be denoted by $RS_e$;
repeat
If a branching vertex is encountered then
for all branches of the branching vertex
mark a branch edge with the next integer;
include $RS_e$ in $RS_e$
only if the rule labels $RS_e$ on it do not belong to $RS_e$;
endfor;
endif;
continue marking chain edges with the next integer without
looking for the corresponding rule label
till another branching vertex is encountered
until the other end of the chain is reached;
Let / be the highest label (i.e., number of edges labelled in this way);
end;

The value of / indicates the number of marked edges in the spanning tree. It can
be seen that the set of rules on the marked edges is nothing but the spanning tree
rule set. The value of / can be used now to determine the position of the edges to
be cut and thus find the subsets.

The vertices and the edges numbered this way are stored in an array of size n
to facilitate decomposition. Starting from the edge marked 1, all edges including
the branch edges without any markings, and the associated vertices along with the
integer marking information in increasing order are stored in the spanning tree. For
the spanning tree and its chain shown in figure 3.3, the linear representation would
be the set of edges \{JI, IG, GH, HB, BC, CA, BF, FD, FE, FK\}, with the associated
vertex ordering \{J,I,G,H,B,C,A,F,D,E,K\}.

### 3.3.4 Initial Decomposition

For the initial decomposition, we must compute the size of each subset, determine whether balanced partitioning, i.e., partitioning exactly in the given ratio, is possible, and then determine the cutsets to be used for decomposing the graph.

#### Size of subset

Let \( N \) be the total number of rules, and \( p_1 : p_2 : \ldots : p_k \) be the proportion in which rulebase subsets are to be obtained. We can obtain size of each knowledge subset \( z_i \) as \([p_i \times (N / \sum_{j=1}^{k} p_j)]\).

```plaintext
procedure determine_size_of_subset();

begin
  for i := 1 to k do
    \( z_i = [p_i \times (N / \sum_{j=1}^{k} p_j)] \);
  endfor;
end;
```

#### Is balanced partitioning possible?

If \( N \mod \sum_{j=1}^{k} p_j = 0 \) then, it may be possible to obtain a balanced partitioning in the given ratio. Otherwise there may be a small difference in the desired sizes calculated for the given ratio and the actual sizes of the subsets obtained. The following procedure does the same.
procedure check_if_balance_possible(bal_possible);

(* check if required partitioning is possible *)
begin
if \( N \mod \sum_{j=1}^{k} p_j = 0 \) then
    bal_possible = true
else bal_possible = false;
endif;
return(bal_possible);
end;

Determining the position of the edges to be cut

If \( l \) is the number of marked (labelled) edges of the spanning tree, we can divide it approximately in the given ratio by cutting at edges \( e_1, e_2, \ldots, e_{k-1} \) where

\[
e_i = \lfloor (\sum_{j=1}^{i} p_j / \sum_{s=1}^{k} p_s) \times l \rfloor;
\]

If an edge to be cut, \( e_i \), happens to be a branch edge, and if the number of rules on the chain \( RS_e \) happens to be small compared to the total number of rules \( N \), then \( e_i \) is taken as it is. Otherwise, the following possibilities exist depending on whether \( p_i \geq p_{i+1} \) or \( p_i < p_{i+1} \).

- In the first case, i.e., if \( p_i \geq p_{i+1} \), we skip the branch, go in the forward direction, i.e., in the direction of increasing integer markings, and make \( e_i \) the first edge in the chain immediately after the branch so that rules on the branch edges are added to \( P_i \).

- Otherwise, if \( p_i < p_{i+1} \), then we make \( e_i \) the last edge in the chain just before the branch so that branch rules are added to \( P_{i+1} \).

This is to retain the sizes of subsets (to be obtained) closer to the proportion given. Further, if any two edges \( e_i \) and \( e_{i+1} \) have no marked edges in between
them, then we must increment or decrement the number appropriately depending on whether $p_i \geq p_{i+1}$ or not. However, this is done only if the new edge does not belong to a branch again and violate the branch criterion described above. The branch skipping keeps the resulting cutset as a fundamental cutset and can reduce information exchange between subsets.

**procedure** determine_edges_to_cut();

(* determine the position of edges $e_i$ to be cut to make the graph disjoint *)

begin

i) for each subset do

\[
e_i = \left\lfloor \left( \frac{\sum_{j=1}^{i} p_j}{\sum_{s=1}^{k} p_s} \right) \cdot N \right\rfloor;
\]

endfor;

ii) if $RS_c$ is close to $N$ then

(* the number of rules on the chain edges is close to $N$ *)

for $i := 1$ to $k-1$ do (* for each subset *)

if $p_i \geq p_{i+1}$ then

$e_i$ is the first edge in the chain immediately after the branch

else

$e_i$ is the last edge in the chain just before the branch;

endif;

endfor;

endif;

iii) for $i := 1$ to $k-1$ do (* for each subset *)

if $(e_{i+1} - e_i - 1) = 0$ then

if $(p_i \geq p_{i+1})$ and ($e_i + 1$ does not belong to a branch) then

$e_i = e_i + 1$

else if ($p_i < p_{i+1}$) and ($e_i - 1$ does not belong to a branch) then

\end{verbatim}
\[ e_i = e_i - 1; \]
\[ \text{endif;} \]
\[ \text{endif;} \]
\[ \text{endfor;} \]
\[ \text{end;} \]

Once the edges in the spanning tree to be cut are identified for including them in the cutsets, we can determine the actual cutsets by including the chords (and other spanning tree edges) also. Removal of these cutset edges leaves the graph as \( k \) components. The data and rules which belong to the individual components form an approximate partition. Retaining the same partition of data obtained by the spanning tree decomposition, rules may be assigned to the subsets based on the data available with the subsets and the data required for the rules. We first find a proposed set of rules for each subset and do the actual assignment in the boundary refinement phase. We shall denote the data (vertex) set and the proposed rule set of a subset by \( VSi \) and \( PRSi \) respectively. Steps for finding these are given below.

**procedure** `find-data_and_proposed_rule_sets();`

(* Form disjoint vertex sets \( VSi \)'s and *)

(* proposed rulesets \( PRS_i \)'s for the \( k \) subsets *)

(* by cutting at \( e_i \)'s along the chain; *)

begin

for \( i := 1 \) to \( k \) do (* for each subset do *)

\[ VSi = \{ v \mid v \text{ is the second end vertex of } e_{i-1} \text{ or} \]
\[ \quad \text{v is the first end vertex of } e_i \text{ or} \]
\[ \quad \text{v is a vertex incident on edges between } e_{i-1} \text{ and } e_i \}; \]

if \( i = 1 \) then \( e_0 \) is the very first edge in the chain;

if \( i = k \) then \( e_k \) is the last edge in the chain;

\[ PRSi = \{ RS_\varepsilon / RS_\varepsilon \text{ is the set of rule labels on the edges between } e_{i-1} \text{ and } c_i \}; \]

endfor;

end;
The set of rules present on the cutset edges is called the Conflict Rule Set and is denoted by $CRS$. These rules correspond to the spanning tree edges as well as the chords in all the $k - 1$ cutsets. Since these are the likely candidates for inclusion in more than one subset, the exact subset in which a rule in $CRS$ is to be placed has to be determined in the boundary refinement phase.

Rules corresponding to a chord (or to those edges other than the $e, s$ calculated for cutting) and found to be using and producing data belonging entirely to one subset $p_i$ are included in the $PRS_i$ of that subset; otherwise, if the end vertices of the chord (or another spanning tree edge of cutset) are in two different components, the rule is included in the $CRS$. The following procedure does this.

\begin{verbatim}
procedure find-proposed\^^{and.cutset}rules()
(* Compute the cutset rules (Conflict Rule Set) CRS, and associated subsets; *)
begin
for i := 1 to k - 1 do (* for each subset *)
   for each vertex $v_j$ of the set $VS_i$ do
      for each edge $v_jv_i$ with rule set $RS_e$ do
         (* in the corresponding undirected graph *)
         if $v_i$ $\in$ $VS_m$ where $m <> i$ then (* $m$ not equal to $i$ *)
            store $RS_e$ and the the subset ids $i$ and $m$ in CRS
         else $PRS_i = PPRS_i \cup \{RS_e\}$;
         endif;
      endfor; (* edge *)
   endfor; (* vertex *)
endfor; (* subset *)
end;
\end{verbatim}
It may be observed that (all) the cutsets formed like this are not necessarily fundamental cutsets with respect to that spanning tree. If an edge $e_i$ belongs to a branch, it will not form a fundamental cutset. The vertices for the first subset are separated first, those for the second subset next and so on, until all vertex subsets are identified. Then the cutset edges are those which have end vertices in two different vertex sets.

3.3.5 Boundary Refinement

In this phase, we determine the subsets to which the rules in the CRS are to be assigned. Among the rules in $CRS$, a rule with largest number of attributes is considered first. To resolve the conflict and actually assign the rule, selection of the appropriate subset can be done by calculating the number of data elements (pertaining to this rule alone) available in each candidate subset. Let us call this its attribute count. Now, a rule is assigned to a subset with highest attribute count and still has not got its share of rules $z_i$. However, if all the candidate subsets have got their share of rules, the rule is assigned to a subset with highest attribute count. The partition obtained above ensures rule completeness.

Metaknowledge Directories

Once a CRS rule is assigned to a subset, it is necessary for the agent (to which that subset is allocated) to know what other relevant data is required from other agents (having other subsets) in order to fire that rule. Alternatively, data generated by firing this rule may be used by some other agent also. Hence, it is necessary that the agent knows which other agents require this data. For each subset $P_i$, we maintain two areas in the directory, viz., $NRF_i$ and $MRB_i$ for this purpose.

$NRF_i$ represents the data that Needs to be Requested From other agents. It has the details of data name and the id of the subset (agent) from which the data is to be requested (obtained). Similarly, $MRB_i$ represents the data that May be Required By other agents. The details of data name and the id of the subset (agent) which may be requiring this data are stored in $MRB_i$. Assuming a rule in subset $Pi$ requires some attribute $v_a$ in subset $P_j$ ($v_a \in VS_i$, i.e., owned by agent having subset $P_j$), $NRF_i$ has an entry $(v_a, j)$ and $MRB_j$ has an entry $(v_a, i)$. 
The NRFs and MRBs represent the coupling between agents and thus help in the allocation of knowledge subsets to agents, redistributing knowledge dynamically and requesting for nonlocal information. This will be discussed in section 3.7 and the forthcoming chapters.

The procedure assign\_rules assigns the rules in CRS to the subsets as described above and updates the NRF and MRB parts of each subset.

\begin{verbatim}
procedure assign\_rules();

(* Assign rules to subsets and store directory information *)
(* NRF\_i: Needs to be Requested From other agents, a list (attribute, agent) *)
(* MRB\_i: May be Required By other agents, a list (attribute, agent) *)
begin
i) for i := 1 to k do (* for each subset i *)
    nonconflicting rule set RS\_i = PRS\_i − CRS;
    NRF\_i = {};
    MRB\_i = {};
    if RS\_i >= z\_i then
        mark it okay and add it to okay list;
    endif;
endfor;
ii) sort the c rules in CRS in decreasing order of rule size,
    (* i.e., on the number of attributes in the rule *)
    for each rule in CRS do
        sort the candidate subset ids involved based on number of
        attributes of this rule (in that subset)
        for i := 1 to s\_i do (* each subset id i in the sorted list*)
            if RS\_i < z\_i then
                ...
\end{verbatim}
include the rule in $RS_i$ (* of that subset *);
\[ RS_i = RS_i + 1; \]
endif;
if $RS_i = z_i$ then
mark it okay;
add the subset in okay list;
break;
endif;
endfor;
endfor;

if the rule is not allotted to any of these subsets and bal_possible = false then
allot the rule to the first subset with an id $i$ at the beginning of the list;

$|RS_i| = |RS_i| + 1;
if $RS_i \geq z_i$ then
mark the subset okay and add it in okay list;
endif;
endif;
endif;

let $P_i$ be the subset to which the rule is allotted;
for each attribute $v_a$ of the rule,
such that $v_a \in V_j$ where $j \neq i$
(* $P_j$ is a candidate subset which did not get the rule *)
\[ NRF_i = NRF_i \cup (v_a, j); \]
\[ MRB_j = MRB_j \cup (v_a, i); \]
(* update NRF of the subset to which the rule was allotted
and MRBs of the remaining candidate subsets *)
endfor;
end;
The procedure `check_if_balance_obtained()` checks whether the partitioning obtained is exactly in the given ratio. If a balanced partitioning is possible, but is not obtained with the above spanning tree edges cut, \( e_1, \ldots, e_{k-1} \), then it shifts some of them so as to shift few of the extra rules from the subsets whose sizes are greater than their desired sizes.

**procedure check_if_balance_obtained(balance_obtained):**

(* Check if required partitioning is obtained *)

\[
\text{begin}
\]

\[
\text{balance_obtained := false;}
\]

\[
\text{for each subset } i \text{ do}
\]

\[
\text{if } |RS_i| = z_i \text{ then}
\]

\[
\text{put the subset in the balanced subset list;}
\]

\[
\text{endif;}
\]

\[
\text{endfor;}
\]

\[
\text{return(balance_obtained);}
\]

\[
\text{end;}
\]

After the final vertex sets and rule sets are found, the following procedure finds the exclusive vertex sets \( EVS_i \)'s and duplicate vertex sets \( DVS_i \)'s. While \( VS_i \) represents the data owned by the subset \( P_i \), \( EVS_i \) represents the data that belongs to subset \( P_i \) and is used only by the rules in \( P_i \). However, \( DVS_i \) represents all the data required by the agent to fire the ruleset \( RS_i \). If data duplication is allowed, updates should be propagated to all the places.

The following procedure calculates the \( EVSiS \) and \( DVSiS \)'s and completes the filling of slots in the metaknowledge of agents.
procedure create_metaknowledge();

(* Compute the attributes which are exclusively owned by this agent(EV_\text{i}),
and all the attributes needed (DV_\text{i}) for firing its local ruleset R_\text{i} *)
begin
for i := 1 to k do (* for each subset do *)
    EV_\text{i} = V_\text{i} - vertex set pertaining to MR_\text{i};
    DV_\text{i} = V_\text{i} \cup vertex set pertaining to NRF_\text{i};
endfor;
end;

It may be seen that the number of entries in the union of all NRF_\text{i} is equal to the
number of entries in the union of all MR_\text{i} and gives the total number of attribute
duplications required for the partition, if data duplication is allowed. Otherwise it
represents the total communication coupling between agents. The number of entries
of the form (v_\text{a}, j) in NRF_\text{i} and those of the form (v_\text{b}, i) in MR_\text{j} indicates the data
to be exchanged between the agents having P_\text{i} and P_\text{j}, and hence the communication
coupling between them. The communication coupling is useful for reasoning as well
as dynamic distribution of the knowledge.

3.3.6 Partitioning Algorithm

The complete algorithm is given below.

Algorithm connected_static_partitioning

Assumptions:

The algorithm requires the knowledge graph to be connected

Inputs:

Rulebase consisting of N rules

Proportion \( p_1 : p_2 : \ldots : p_k \) in which rulebase subsets are to be obtained
Chapter 3. Static Knowledge Base Partitioning and Allocation

Outputs:

Rulebase subsets $P_1, P_2, \ldots, P_k$ with rules in the given ratio
Directories for the rulebase subsets with attributes details – owned or shared

Steps:

1. Construct the knowledge graph in the form of adjacency list with attributes as nodes, and arcs connecting all input attributes of a rule to each of its output attributes as edges with rule number(s) as the label(s) on each edge.
   Compute indegree, outdegree, and degree for each attribute.

2. (* spanning tree generation and marking *)
   (a) `generate_spanning_tree();`
   (b) `find_chain();`
   (c) `mark_edges();`

3. (* check whether balanced partitioning is possible *)
   (a) `determine_size_of_subset();`
   (b) `check_if_balance_possible(bal_possible);`

4. (* partitioning *)
   (a) `determine_edges_to_be_cut();` (* initial decomposition *)
   (b) `balance_obtained = false;`
      `Her := 0;`
   (c) repeat
      i. `find_data_and_proposed_rule_sets();` (* initial decomposition *)
      ii. `find_proposed_and_CutscLrules();` (* initial decomposition *)
      iii. `assign_rules();` (* boundary refinement *)
      iv. `check_if_balance_obtained(balance_obtained);`
      if (balance-obtained = false) and (iter < maxiter) and (bal_possible = true) then
         (* maxiter is a constant defined by the user *)
         for subsets that do not balance do
            shift $e_i$ by one such that
it is in a subset $P_i$ whose $|RS_i| > z_i$;

\textbf{endif};

Her := iter + 1;
\textbf{endif};

until (balance\_obtained = true) or (balance\_possible = false) or (Her > maxiter)

5 \textbf{(a) create\_metaknowledge()};

\textbf{(b) for} i := 1 \textbf{to} k \textbf{do} (* for each subset*)

\textbf{print} the lists $RS_i, VS_i,$ $NRF_i, MRB_i, EVS_i$ and $DVS_i$;

6 \textbf{stop};

\textbf{Complexity of the Partitioning Algorithm}

Construction of the adjacency list for the graph corresponding to the \texttt{rulebase} requires $N \cdot a$ time where $TV$ is the number of rules and $a$ is the average number of attributes in a rule. However, this is trivial for any partitioning algorithm and need not be considered in the calculation of complexity of the algorithm.

By using an adjacency list for representing the knowledge graph, step 2(a), i.e., generation of a spanning tree (also an adjacency list) can be done in $O(n)$ time where $n$ is the number of attributes. By storing the semipath lengths for each intermediate vertex from leaf vertices in step 2(a) itself, each of steps 2(b) and 2(c) requires $O(n)$.

Steps 3(a) and 3(b) together require $k + 1$ computations where $k$ is the number of subsets required. Hence step 3 is $O(k)$.

Step 4(a), i.e., determination of edges to be cut, requires $k$ iterations and is of $O(k)$, 4(b) requires two computations.

Step 4(c)(i), finding the vertex sets (and the initial proposed rule sets) for the subsets requires examination of all the $n$ vertices. Hence this is $O(n)$.

Step 4(c)(ii) for finding the complete proposed rule set and conflict rule set requires $k$ iterations as there are $k$ vertex sets. For each vertex all the edges incident on it are to be examined. Assuming the average number of vertices is $n/k$, average
number of edges for a vertex is $\frac{|E|}{n}$ where $E$ is the set of edges in the knowledge graph, for $k$ iterations in the outermost loop, the time complexity is $O(|E|)$.

Step 4(c)(iii) involves finding the nonconflicting rulesets and then assigning the CRS rules to subsets. Determination of nonconflicting rulesets is $O(k)$. Assuming that there are $c$ rules in CRS, each rule usually has 2 to 3 subsets as candidates to which it can be assigned. In the worst case, which usually does not occur in practice, the number of candidate subsets would be $k$. Updation of NRFs and MRBs requires time $O(k)$. Therefore this step is $O(ck)$. However, since $c \ll TV$, $c \ll n$, and $c \ll |E|$, overall complexity of step 4(c) is the maximum of $O(n)$ and $O(|E|)$. The number of iterations in the repeat loop is constant, usually about two or three, and hence need not be added to the complexity.

So, the overall complexity of step 4 is $O(max(n, |E|))$.

Steps 5(a) and 5(b) are of $O(k)$; 5(b) is in fact mere outputting of the results.

Therefore, the overall complexity of the algorithm is $O(max(n, |E|))$.

### 3.4 Examples

We shall illustrate our approach with the help of a few examples using the rulebase we considered in section 3.2.

#### 3.4.1 Case 1: Two subsets in the proportion 2 : 1

1. The knowledge graph is shown in figure 3.1 and the degree information is given in table 3.1.

2. (a) One of the attributes with zero outdegree is J. It is also a pendant vertex to serve as a root. The spanning tree generated starting with it is shown in Figure 3.3. This has the edges $JI$, $GI$, $GH$, $BH$, $BC$, $AC$, $BF$, $DF$, $EF$ and $FK$.

   (b) The chain is computed as the set of edges $JI$, $GI$, $GH$, $BH$, $BF$ and $FK$.

   (c) The highest edge marking in this spanning tree / is computed as 7. The edges belonging to branches are $BC$, $AC$, $DF$ and $EF$. While the first
two belong to the same branch emanating from B, the last two are two different branches emanating from F. Edge BA is marked 5 to indicate its position in the semipath. Arcs AC, DF and EF are not labelled because they do not belong to new rules.

3 (a), (b) Required rule partitioning with sizes \( z_1 = 4, z_2 = 2 \) is possible;

4 (a) \( e_1 = 5; \) Arc 5 happens to be a branch. Since the number of rules on the chain \( |RS_2| = 5 \) is close to \( N = 6 \), and since \( p_t > p_{t+1}, \ e_t \) is advanced by one more edge. This is equivalent to adding one more edge to the bigger subset, \( e_1 = 6; \)

(b) The variable balance obtained is initialized to false, and the number of iterations, iter = 1.

(c)

Disjoint vertex sets \( VS_1 = \{A,B,C,G,H,I,J\}; VS_2 = \{D,E,F,K\}; \)

Proposed rule sets \( PRS_1 = \{R1,R3,R4,R6\}; PRS_2 = \{R2,R5\}. \)

Cutset edges are BF, GK; Associated rules make the conflict rule set \( CRS = \{R2, R5\}; \)

Initial rule sets for the subsets \( RS_1 = \{R1,R3,R4,R6\} \) (with corresponding rule count \( |RS_1| = 4 \) and \( RS_2 = \{} \) (with \( |RS_2| = 0 \)). All the four rules belong exclusively to subset \( P_1 \). \( P_1 \) is marked okay as it has the required number if rules. \( P_2 \) does not have any rules in it so far.

Of the rules in \( CRS, \) for \( R2, \) of the attributes B,D,E and F, only B is needed by subset \( P_2 \) from \( P_1 \). Hence this rule can be allotted to \( P_2 \).

Hence, \( RS_2 = \{R2\}; NRF_2 = \{B(1)\}; MRB_1 = \{B(2)\}; \)

For \( R5, \) of G,F and K, only G is required by \( P_2 \) from \( P_1 \). Assigning this rule to \( P_2 \) makes \( RS_2 = \{R2,R5\}; NRF_2 = \{B(1), G(1)\}, \) and \( MRB_1 = \{B(2), G(2)\}. \) Now, \( P_2 \) has got 2 rules and the agent with \( P_2 \) may need to request for attributes B and G from agent with \( P_1 \). \( P_2 \) is marked okay.

Since both the subsets are in the required proportion, balance is obtained.

5 Final Partition:

After filling the remaining metaknowledge slots, the final partition is shown
If no duplication is allowed in the working memories of agents with these subsets, each of the disjoint vertex sets $V_{Si}$ represents the data owned by the agent having subset $P_i$. $NRF_{Si}$ and $MRB_{Sj}$ give the information about the data that needs to be requested from other agents (attribute and from whom to request), and the attributes that may be requested by others (attribute and from whom the request may be sent). There is a possibility that an item generated (as a result of firing the corresponding rules) by one agent may actually be owned by some other agent. Then the first agent may have to send the value immediately to the owner on generating it.

If duplication is allowed to some extent, $DVS_{Si}$ represent the attribute set required by agents for firing the rule sets. $MRB_{Sj}$ can be used to send copies to the other agents that require the item as soon as it is generated, and $NRF_{Si}$ can be still used to request in advance if necessary.

We can see that the agent with subset $P_1$ does not need any information from others and the agent with subset $P_2$ does not have any attributes that may be requested by other agents.

### 3.4.2 Case 2: Three subsets in proportion 1:1:1

Steps 1 and 2 are same as for case 1.

(3) $(a),(b)$ Required rule partitioning with sizes $z_1 = z_2 = z_3 = 2$ is possible.
4 (a)(b). \( e_1 = 3, \ e_2 = 5 \); As \( e_2 \) happens to be a branch edge, and since all the subsets are to be in the same proportion, we simply advance forward making \( e_2 = 6 \). The variable balance_obtained is initialized to false, and the number of iterations, \( \text{iter} \) is made 0.

(c)

\[VS_1 = \{G,I,J\}; \ VS_2 = \{A,B,C,H\}; \ VS_3 = \{D,E,F,K\};\]

\[PRS_1 = \{R4,R6\}; \ PRS_2 = \{R3,R1\}; \ PRS_3 = \{R2,R5\};\]

Cutset edges are AI,BF,GH,GK;

Corresponding rule set \( \text{CRS} = \{R2, R3, R5, R6\}; \)

\[RS_1 = \{R4\}; \ RS_2 = \{R1\}; \ RS_3 = \{\};\]

For \( R2 \), of B,D,E,F, only B is required by \( P_3 \) from \( P_2 \). Assigning it to \( P_3, RS_3 = \{R2\}; \ NRF_3 = \{B(2)\}; \ MRB_2 = \{B(3)\}; \)

For \( R3 \), of B,G,H, only G is required by \( P_2 \) from \( P_1; RS_2 = \{R1,R3\}; \ NRF_2 = \{G(1)\}; \ MRB_1 = \{G(2)\}; \) Since \( |RS_2| = 2 \), \( P_2 \) is marked okay.

For \( R5 \), only G is required by \( P_3 \) from \( P_1; RS_3 = \{R2,R5\}; \ NRF_3 = \{B(2), G(1)\}; \ MRB_3 = \{G(2), G(3)\}; \) Same attribute G is required by subset \( P_2 \) as well as subset \( P_3 \). Since \( |RS_3| = 2 \), \( P_3 \) is marked okay.

Lastly, for \( R6 \), of A,G,I, only A is needed by \( P_1 \) from \( P_2 \). Therefore, \( RS_1 = \{R4, R6\}; \ NRF_1 = \{A(2)\}; \ MRB_2 = \{B(3), A(1)\}; \)

All the subsets are in the specified proportion; hence balance is obtained.

5 Final Partition:

After filling the metaknowledge slots, the algorithm exits. The final partition is shown in table 3.3.

Rulebase partitions obtained for an aerospace vehicle checkout system and medical diagnosis application using our approach are encouraging. These will be discussed in the next chapter.
3.5 Disconnected Components in the Knowledge Graph

Knowledge graphs can also have disconnected components in them. The disconnectedness implies the absence of communication among the agents when each component (or a group of components together) is treated as a subset in some partition and is given to one agent. This actually represents a functional decomposition where each component pertains to some subsystem in the whole system, or the components represent procedures which do not interact at all. Therefore, it is important to be able to identify such components and make use of them appropriately.

Identification of a Component

While generating the spanning tree, if we encounter a vertex that doesn't have another new reachable vertex (from it in the corresponding undirected graph), and none of its predecessors in the spanning tree generated so far have, but there are still some vertices of the knowledge graph which are not included in the spanning tree, it means there are disconnected components in the knowledge graph. These may be indexed with useful information like number of rules in the component for easy retrieval and efficient processing. We shall call the number of rules in a component as the size of the component, and the number of rules required for a subset (in the partition to be obtained in the given proportion) as the size of the subset.
Identification of Knowledge Graph Components forming Balanced Subsets

A partition with perfect balancing and zero communication is possible in either of the following two cases:

- number of subsets = number of components, and each subset has one component of the same size

- number of subsets < number of components, and each subset has one or more (groups of) components together equalling the subset size.

In all other cases, one or more components need to be broken to get partitioning in the required ratio. In particular, if the number of subsets is more than the number of components, some components must be cut to obtain rule base subsets whose sizes are in the given ratio. However, (even if perfect partitioning is not possible,) it is desirable to identify the subsets and the group of components, if any, with matching sizes. Let us call a subset which is assigned rules satisfying its size requirement a balanced subset.

These balanced subsets and the components assigned to them should be separated from the rest of their respective groups and should not be considered for further partitioning. This is because communication among such components is nil. The new proportion (with fewer subsets when compared to the number of subsets in the original proportion) representing the remaining subsets and the unassigned components should be calculated for giving it as input to our partitioning heuristic. We propose a heuristic to identify such balanced subsets and the corresponding components when the knowledge graph is not connected. Partitioning in the modified proportion can be obtained by making minor modifications to the connected_static_partitioning heuristic discussed in section 3.3.6.

The procedure balanced_subset_components identifies the balanced subsets which can be formed from components.
procedure balanced_subset_components();

begin
let \( k \) be the number of rulebase subsets required in the proportion \( p_1, \ldots, p_k \);
let \( c \) be the number of components in the knowledge graph;
let \( z_1, z_2, \ldots, z_k \) be the sizes of (rulebase) subsets, sorted in decreasing order;
let \( g_1, g_2, \ldots, g_c \) be the component sizes, sorted in decreasing order;
let \( b_s \) be the number of balanced subsets which have the exact number of rules as its desired size;
let \( r_c \) be the number of remaining components whose rules have to be assigned to subsets still;
i, j, sum, diff, temp are temporary variables;
let open list represent the unassigned components;
let temp list be a temporary list of components for assignment to a particular subset.

begin
\( bs = 0 \);
\( rc = c \);
for \( i := 1 \) to \( k \) do
\( j := 0 \);
\( diff := z_i \);
\( temp := 0 \);
initialize temp list to null;
while \( (j < rc) \) and \( (diff > 0) \) do
\( J := J + 1 \);
if \( (g_j \leq diff) \) then
\( diff := diff - g_j \);
\( temp := temp + 1 \);
copy the jth component in the open component list to temp list;
endif;
endwhile;
if \( \text{diff} = 0 \) then
\[
\begin{align*}
\text{rc} & : = \text{rc} - \text{temp}; \\
bs & : = bs + 1;
\end{align*}
\]
delete components in temp list from open component list;
assign the rules and vertices in temp list to \( RSi \), and \( VSi \), respectively;
\[
\text{NRF}i : = \{ \};
\]
\[
\text{MRBi} : = \{ \};
\]
endif;
endfor;
if \( bs < k \) then
proceed from step 2h of connectedstatic-partitioning algorithm
considering the remaining subset ratios and the remaining components
end;
end;

3.5.1 Partitioning Disconnected Knowledge Graphs

Following are the changes required to the static partitioning heuristic developed for 
a connected knowledge graph, to accommodate multiple components.

**Spanning tree generation**

If a new vertex is not reachable, in the corresponding undirected knowledge graph, 
from any of the vertices in the spanning tree forest generated so far, there is another 
component in the knowledge graph. A new vertex may be chosen as the root for the 
spanning tree of the next component in the graph, and its spanning tree generation 
may be continued in the same fashion in the same loop.
Marking of edges

Though chain of the spanning tree in each component has to be identified separately, marking of edges in the spanning tree proceeds continuously with consecutive integers as though there is only one spanning tree present in the entire graph. Both marking of edges and partitioning commence from the largest unassigned component and continue with the next largest until there are no more components in the graph.

Computation of the edges to be cut, and desired sizes of rulebase subsets

Since edges of a component are given integer labels continuing with those given for the earlier component, edges to be cut can be determined in the same way as we did for partitioning a connected graph.

Complexity of the augmented version of partitioning algorithm

Besides the additional checking required for identifying the components that form balanced subsets, changes are required to the procedures

- Spanning tree generation
- Finding chain
- Marking of edges

so that partitioning heuristic can be applied on the knowledge graph with multiple components.

Time required for spanning tree generation is the same except that when the open list becomes empty, a new root vertex is chosen for the next component before proceeding with the inclusion of edges in the spanning tree for the new component. Since degree information is stored with vertices while constructing the knowledge graph itself, this doesn't require extra time. Extra time is required only for the selection of new root and to store the component itself. This, however, is negligible as backtracking is minimized by deleting the vertices from the open list the very
first time the vertex is found not to be having new vertices on its edges. Therefore, the time complexity does not increase for this step.

Chains have to be found for each spanning tree separately. However, as both the depth first search for finding the longest semipath from the root and finding the chain from it are of $O(n)$ as mentioned earlier, the total time required for finding all chains will also be of $O(n)$.

Storing the spanning forest information in a linear array form after finding the chains (as done for a single spanning tree) keeps its time complexity $O(n)$ only.

The extra step `balanced_subset_components` for finding the components that form balanced subsets requires $O(kc)$ time. However, the number of components $c$ will certainly be less than the number of data elements $n$, in fact $c$ should be less than $n/2$. Therefore, the overall complexity of the modified version of our heuristic to deal with disconnected components remains the same, i.e., $O(\max(n,|E|))$.

### 3.6 Obtaining Functional Decomposition

Instead of concentrating on the load balancing aspect (by obtaining rulebase subsets in the given proportion of rules), we may also consider a partition that will result in a functional decomposition. There are three possibilities here.

As mentioned in the previous section, if there are disconnected components, the partition representing the components indicates a natural functional decomposition.

However, if the graph is connected, for obtaining a functional decomposition, we need to group rules freely based on the coupling among rules on the edges by relaxing the constraint on the number of rules per subset. Some times, this may result in duplication of few rules in the subsets. An accurate representation of the data and the hierarchy among the objects involved, if considered in the steps of the algorithms already discussed, gives us a partitioning close to a functional decomposition without duplication of rules. For this, we also take into account the coupling among the rules incident on edges. The coupling can be clearly seen if we observe the rule labels on the edges. The data connected by an edge, and edges emanating from and leaving some data node are closely connected and have some dependency. A graph as the representation scheme for the knowledge base depicts the dependencies clearly and
enables our partitioning algorithm exploit the adjacency and dependencies inherent in the structure. All rules corresponding to an edge should be preferably assigned to the same subset, and if two or more edges have some rules common, they indicate some amount of coupling, i.e., communication between the rules, the precise quantity being proportional to the number of common rules.

Alternatively, functional decomposition can be obtained by considering a final result (attribute with zero outdegree), including all the rules incident, on its incoming edges, and proceeding backwards in the same way until all external input attributes (with zero indegree) are considered. This gives all the rules and data corresponding to a subsystem concluding with the final attribute considered above. There may or may not be some overlap among the rules in different subsets indicating the interaction required among subsets, the latter being the case of disconnected components.

3.7 Knowledge Subset Allocation

Once partitioning is completed, the next step is to assign the subsets to individual agents. We discuss the allocation of subsets obtained in a given ratio representing capacities of agents using the heuristics discussed in sections 3.3.6 and 3.5. We do not discuss the allocation of subsets in a functional decomposition as this may not require load balancing. Even if the subsets have to be allocated in that way, load balancing may be given secondary importance and techniques similar to assignment of components to subsets may be used.

The subsets obtained using our heuristic can be easily assigned if all the agents are situated at equal distances from one another. The allocation problem becomes trivial to that of simply assigning the largest subset to the agent with maximum capacity, the second largest to agent with next highest capacity and so on.
Chapter 3. Static Knowledge Base Partitioning and Allocation

**procedure** equal_distance_allocation();

(*Pi is the list of subsets sorted in the decreasing order of size; *)

(* Ai is the list of agents sorted in decreasing order of capacity; *)

begin

for i := 1 to k do

assign the subset Pi to agent Ai;

endfor;

end;

In this case, the allocation becomes optimal also. However, the distance between agents need not be equal in real life problems. Hence, the allocation strategy must consider both capacities of agents and distance between them.

3.7.1 Allocation Algorithm

Since our partitioning heuristic partitions the rulebase according to the capacities of agents, the subsets obtained and the agents have a one to one mapping where sizes of subsets match with capacities of agents.

Therefore, the allocation problem can be stated as a mapping problem where there is exact correspondence between the size of a partition and capacity of an agent, and the communication overhead is to be minimized based on the coupling (data transfer required) between subsets and the distance between agents.

Given a partition in the ratio p1 : p2 : .. : pk, the coupling between subsets as the amount of data transfer, and the network of agents with capacities s1, s2, .., sk our objective is to map same size subsets onto agents of correspondingly same capacity such that communication overhead is minimized.

We propose a heuristic solution as follows:
procedure allocate();

begin
sort the agents on the decreasing order of capacity;
sort the subsets on the decreasing order of size;
sort subset pairs in the decreasing order of coupling between the pairs;
(* group them together based on the amount of coupling *)
sort agent pairs in the increasing order of distance between the agents;
(* group them together based on the distance *)

select the first pair of subsets with the maximum coupling;
select the first pair of agents separated by the first minimum distance;
repeat (* with each subset pair *)
    repeat (* with agent pair *)
        if subset sizes match with agent capacities then
            assign the subsets to agents of corresponding capacity;
            mark the assignment okay;
        endif;
        consider the next agent pair
    until the assignment is okay;
    (* both subsets of the subset pair are assigned *)
    consider the next subset pair preferably having
        one subset from (previous pair or) assigned subset list
until (k-1) subsets are assigned;
end;
3.7.2 Example 2

To explain the allocation algorithm, we shall consider an enlarged rulebase.

The rulebase after adding six more rules to the example rulebase 1 is shown below.

<table>
<thead>
<tr>
<th>R1.</th>
<th>AB → C</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2.</td>
<td>BDE → F</td>
</tr>
<tr>
<td>R3.</td>
<td>BG → H</td>
</tr>
<tr>
<td>R4.</td>
<td>I → J</td>
</tr>
<tr>
<td>R5.</td>
<td>FG → K</td>
</tr>
<tr>
<td>R6.</td>
<td>AG → I</td>
</tr>
<tr>
<td>R7.</td>
<td>KM → L</td>
</tr>
<tr>
<td>R8.</td>
<td>M → N</td>
</tr>
<tr>
<td>R9.</td>
<td>O → N</td>
</tr>
<tr>
<td>R10.</td>
<td>O → P</td>
</tr>
<tr>
<td>R11.</td>
<td>Q → P</td>
</tr>
<tr>
<td>R12.</td>
<td>KQ → R</td>
</tr>
</tbody>
</table>

The knowledge graph for the example rulebase 2 is shown in figure 3.4.

A spanning tree for the Example rulebase 2 is shown in figure 3.5.

The final partition obtained after the initial partitioning and boundary refinement of the connected_static_partitioning heuristic is shown in figure 3.6 and in table 3.4.

Now this partition can be considered as input along with the network topology for illustrating the allocation algorithm.

Given:
the above partition \( P_1, P_2, P_3, P_4 \) with sizes of the parts in the proportion 2:1:2:1, communication coupling among the subsets \( P_1P_2 = 2, P_2P_3 = 1, P_2P_4 = 1 \), and \( P_4P_3 = 1 \)
and a network of agents \( A_1, A_2, A_3, A_4 \) with capacities in the proportion 2:2:1:1 and distance between the nodes as \( A_1A_2 = 2, A_1A_3 = 2, A_1A_4 = 1, A_2A_3 = 2, A_2A_4 = 1, \) and \( A_3A_4 = 1 \),
we shall use our algorithm to find an allocation with less communication overhead, and load balancing.
Figure 3.4: Knowledge Graph for the Example Rulebase 2
Figure 3.5: A Spanning Tree for the Knowledge Graph of Example Rulebase 2
Figure 3.6: A 2:1:2:1 Partition of the Example Rulebase 2
We can see that the subsets $P_1$ and $P_3$ are of size 2 units, and these have to be allotted only to agents of the corresponding capacity, viz., $A_1$ and $A_2$. Interchanges in allocation are possible only between them. The remaining subsets can be assigned among themselves to any of the remaining agents.

The sorted lists of subset pairs and agent pairs are shown below.

We shall illustrate three different cases of allocation.

Considering the first pair of subsets $P_1$ and $P_2$, and the first pair of agents $A_1$ and $A_4$, $P_1$ can be assigned to $A_1$ and $P_2$ can be assigned to $A_4$. We show this by
$P_1 \rightarrow A_1,$ and $P_2 \rightarrow A_4$. There are two more subsets to be assigned to two more agents. Leaving this assignment undisturbed, we proceed with the next subset pair in the list.

If we select $P_2P_3$ as the next pair of subsets for consideration, it will result in $P_2 \rightarrow A_4$ and $P_3 \rightarrow A_2$. Now that three of the subsets are assigned to three agents, and that the unassigned subset has an assignment compatible with the remaining agent, the final assignment is as shown below.

\[
\begin{align*}
P_1 & \rightarrow A_1 \\
P_2 & \rightarrow A_4 \\
P_3 & \rightarrow A_2 \\
P_4 & \rightarrow A_3
\end{align*}
\]

Had we selected $P_2P_4$ instead of $P_2P_3$, we could have got the same result though with an extra step.

Considering $A_2A_4$, $P_2$ is already assigned to $A_4$; $P_4$ and $A_2$ are incompatible as they belong to groups of different size and capacity respectively. $P_4$ belongs to the group of subsets of size 1, and $A_2$ belongs to the group of agents of capacity 2. Therefore, this assignment is not compatible.

Considering the next agent pair $A_3A_4$ for mapping the subset pair $P_2P_4$, the assignment $P_2 \rightarrow A_4$ still holds; $P_4$ can be safely assigned to $A_3$ as both of them belong to compatible groups, i.e., of size 1. Now, $P_4 \rightarrow A_3$.

As already three out of the four subsets have been assigned to three of the agents, and as $P_3$ and $A_2$ are compatible, we can make the assignment $P_3 \rightarrow A_2$.

The final assignment is shown below:

\[
\begin{align*}
P_1 & \rightarrow A_1 \\
P_2 & \rightarrow A_4 \\
P_4 & \rightarrow A_3 \\
P_3 & \rightarrow A_2
\end{align*}
\]
It can be verified that the this assignment holds true with the other subset and agent pairs in the list.

Similarly, another mapping

\[ P_1 \rightarrow A_2 \]
\[ P_2 \rightarrow A_4 \]
\[ P_4 \rightarrow A_3 \]
\[ P_3 \rightarrow A_1 \]

also is a minimal communication overhead assignment with the load balanced evenly among the agents.

### 3.8 Conclusions

We have proposed heuristics for statically partitioning knowledge bases by representing them as knowledge graphs. The partitioning algorithm presented handles both homogeneous and heterogenous partitioning cases and is independent of the graph structure and application domain. A partition in the required rule ratio can be obtained quickly and may be used as it is. Otherwise, it may be used as a good starting partition for optimizing performance of algorithms KL and SA. It reduces data duplication and the related inconsistency and communication problems by taking care of the data dependencies. Also, the partition obtained helps in deciding and organizing the working memory contents. When an agent cannot proceed with local problem solving due to insufficient or incomplete data which may be available with other agents, directory information in the form of NRFs and MRBs help to reason, and send requests in a directed fashion. The heuristic for partitioning connected graphs is extended to deal with graphs having components.

The allocation heuristic assigns knowledge subsets with maximum coupling to a closest pair of agents of suitable capacity and minimizes communication while achieving load balancing and less communication.

Task decomposition can be done based on the partitioning of knowledge and data, and subtasks can be assigned to agents accordingly.
The static partitioning algorithm described assumes that all rules have equal probability of being fired. This is because determination of rule firing frequencies at compile time is difficult. However, the varying rule firing frequencies may cause load imbalance during run time and hence dynamic load balancing is necessary. Knowledge distribution for dynamic load balancing is discussed in chapter 5.

The next chapter presents two case studies highlighting the various aspects of the static partitioning algorithm developed in this chapter.