Chapter 4

MINING THE GRAPH

This chapter describes the characteristics of the Subdue system [24, 45], our data mining tool. Subdue was developed at the University of Texas at Arlington and it can be applied to any domain that can be represented as a graph. In the first part of the chapter we present the general characteristics and main functions. In the second part we describe the overlap feature and its importance for the pattern discovery system, and introduce a new algorithm named *Limited Overlap*.

4.1 Characteristics

The Subdue system discovers substructures using a graph-based representation of structural databases. Structural data involves the concept of units or parts and the interdependence and relationships of those parts. The substructures (a connected subgraph within the graphical representation) describe concepts in the data (i.e., patterns).

The substructure discovery system represents structural data as a labeled graph. The input to Subdue is a graph where labeled vertices correspond to objects in the data, and directed or undirected labeled edges map relationships between objects.

The discovery algorithm follows a computationally constrained beam search. There are three different evaluation methods to guide the search towards more appropriate substructures: Minimum Description Length (*MDL*), Size-based, and Set Cover. The default evaluation method is *MDL*.

The algorithm begins with the substructure matching a single vertex in the graph. During each iteration, the algorithm selects the best substructure and incrementally expands the instances of the substructure. An instance of a substructure in the input graph is a subgraph that matches (graph theoretically) that substructure.

The algorithm searches for the best substructure until all possible substructures have been considered or the total amount of computation exceeds a given limit. Evaluation of each substructure is determined by how well the substructure compresses the description length of the input graph.

There might be slight variations of some substructures that can be considered as instances of another substructure. Subdue uses an inexact graph match algorithm to identify this kind of instances. In this inexact match approach, each distortion of a graph is assigned a cost and if the total cost is lower than a given threshold, the substructure is considered an instance of the other substructure.

A distortion is described in terms of transformations such as the deletion, insertion, and substitution of vertices and edges. The best substructure found by Subdue can be used to compress the input graph, which can then be input to another iteration of Subdue. After several iterations, Subdue builds a hierarchical description of the input data where later substructures are defined in terms of substructures discovered on previous iterations.

Figures 4.1, 4.2, 4.3 and 4.4 show an example of the system's functionality. The example is presented in terms of the house domain, where a house is defined as a triangle on a square. *T* represents a *triangle*, *S* a *square*, *C* a *circle* and *R* a *rectangle* (see Figure 4.1). The objects in the figure (i.e., *T1*, *S1*, *R1*) become labeled vertices in the graph, and the relationships (i.e., *on*(*S1*, *R1*), *shape*(*C1*, *circle*)) become labeled edges.

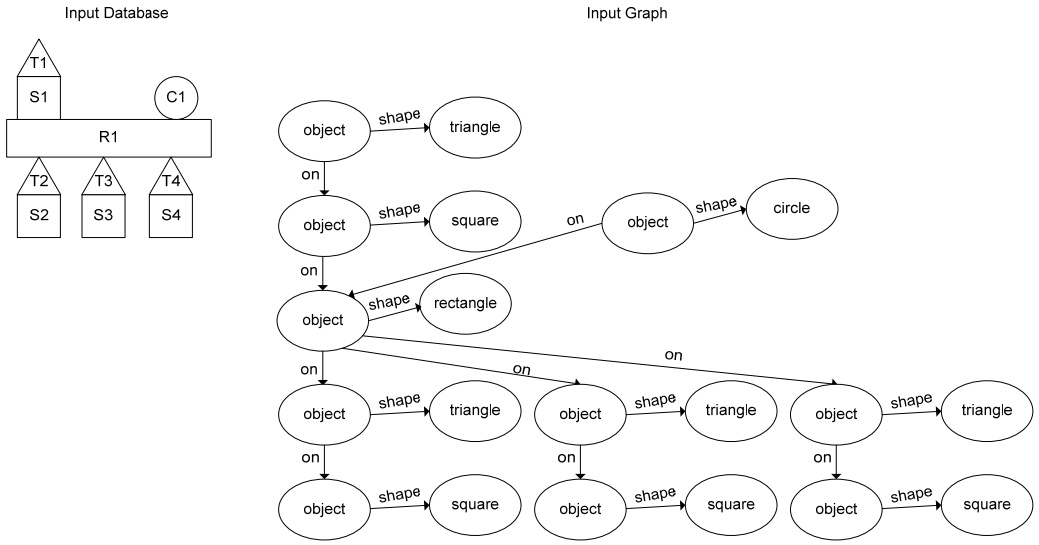

Figure 4.1. Graph representation of the house domain.

The graph representation of the substructure discovered by Subdue from this data is shown in Figure 4.2 where Subdue found four instances of "*triangle on square*".

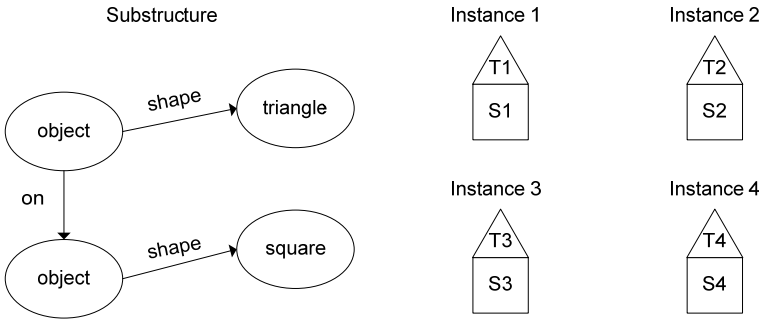

Figure 4.2. Substructure and instances discovered from the house domain by Subdue.

After a substructure is discovered, each instance of the substructure in the input graph is replaced by a single vertex representing the entire substructure. In Figure 4.3 the discovered substructure (*object shape triangle on object shape square*) is labeled as *SUB_1* (**SUB**structure number 1).

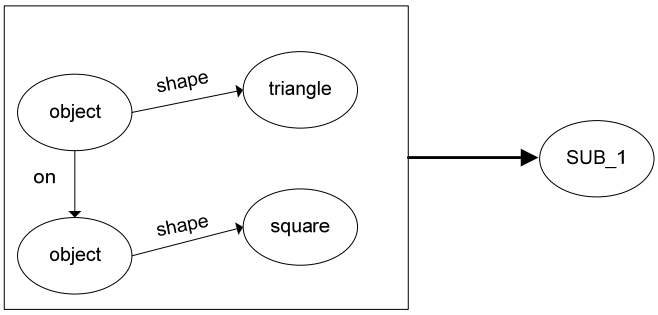

Figure 4.3. Substructure replacement procedure in the house domain.

Finally, the substructure (labeled as *SUB_1*) is used to compress the original input graph, which can then be input to another iteration of Subdue (see Figure 4.4).

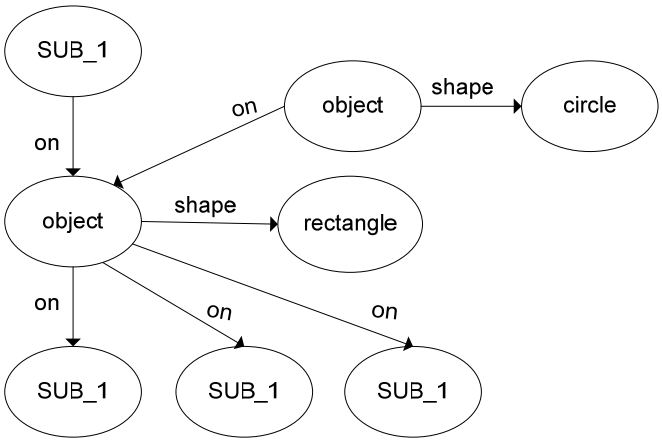

Figure 4.4. Graph representation of the house domain after substructure replacement.

The Subdue system's ability to perform discovery has been proved in several domains including scene analysis, chemical analysis and *CAD* circuit analysis. In [18] the authors present an example of the Subdue capabilities to perform data mining tasks, they work with an earthquake database and show that Subdue is capable of finding not only the shared characteristics of the earthquake events, but also space relations between them. In the case of the identification of shared characteristics, they used the pattern containing the region number specification to recognize the area being studied; in the case of space relations, they found patterns that represent parts of the paths of the involved fault (i.e., subarea with a high concentration of earthquakes).

4.1.1 Main Functions

In this subsection we present a briefly description of the main functions composing the core of Subdue. Each of them is itself integrated by several subfunctions, but the idea is to present them in a global perspective.

Compress

The compress function returns a new graph, which is the given graph compressed with the given substructure instances. Vertices *SUB* replace each instance of the substructure, and *OVERLAP* edges are added between vertices *SUB* of overlapping instances. Edges connecting to overlapping vertices are duplicated, one per each instance involved in the overlap.

Discover

This function plays the role of manager in the phase of discovering the best substructures in an input graph. It is in charge of issues such as getting initial substructures, extending each substructure, evaluating each extension, and adding to a final list the best discovered substructures.

Evaluate

This function implements the different evaluation methods used to guide the search towards more appropriate substructures: Minimum Description Length (*MDL*), Size-based, and Set Cover. The value of a substructure s in a graph g is computed as:

$$size(g) / (size(s) + size(g|s))$$

The value of *size*() depends on whether we are using the *MDL* or Size-based evaluation method. If *MDL* is used, then $size(g)$ computes the description length in bits of g. In case of Size-based, then $size(g)$ is simply $vertices(g) + edges(g)$. The $size(g|s)$ is the size of graph g after compressing it with substructure s. Compression involves replacing each

instance of *s* in *g* with a new single vertex and reconnecting edges external to the instance to point to the new vertex.

Extend

This function returns a list of substructures representing extensions to the given substructure. Extensions are constructed by adding an edge (or edge and a new vertex) to each positive instance of the given substructure in all possible ways according to the graph. Matching extended instances are collected into new extended substructures, and all such extended substructures are returned.

Graphmatch

The Graphmatch function returns true if *graph_1* and *graph_2* match with cost less than the given threshold. If so, the objective is to store the match cost in the variable pointed to by matchCost and to store the mapping between *graph_1* and *graph_2* in the given array is non null.

Subgraph Isomorphism

Set of functions implementing a subgraph isomorphism algorithm. The objective is to find predefined substructures: searches for subgraphs of *graph_2* that match *graph_1* and returns the list of such subgraphs as instances of *graph_2*. Returns an empty list if no matches exist. This procedure mimics the "discover substructures" loop by repeatedly expanding instances of subgraphs of *graph_1* in *graph_2* until matches are found.

4.2 Overlap

Now, we will describe the overlap feature in the Subdue system. Our goal is to present the current implementation of this feature in Subdue and then, in the next section, to describe our new proposal named limited overlap. As we will see, Subdue reports in the results all the overlapping substructures that it can find. Sometimes this is very expensive in time and in the number of substructures that the end user has to analyze. Therefore, the idea is that Subdue can find overlapping substructures only over a set of predefined vertices that are chosen by the user (the limited overlap). The overlap has a preponderant role in the substructure discovery system; it is controlled by the overlap user's parameter. If overlap is false then overlap among instances is not allowed, otherwise, overlap is allowed.

To illustrate the cause/effect of the overlap in Subdue, we will present some examples generated by using two standalone functions belonging to Subdue: the Subgraph Isomorphism and Minimum Description Length functions. The generated results by these functions are affected by the value of the overlap parameter.

Subgraph Isomorphism (SGISO)

As we have already mentioned, the subgraph isomorphism function searches for subgraphs of *graph_2* that match *graph_1* and returns the list of such subgraphs as instances of *graph_2*. The subgraph isomorphism function is embodied in the "FindInstances" function. The procedure is optimized toward *graph_1* being a small graph, and *graph_2* being a large graph.

The FindInstances function starts finding a list of single-vertex instances, one for each vertex in *graph_2* that matches the first vertex of *graph_1*. Next, the algorithm attempts to extend each instance in the instance list by an edge (or edge and new vertex) from *graph_2* that matches the attributes of the given edge in *graph_1*. Finally, the instances not matching *graph_1* are filtered, and an overlapping instances validation process is performed. If overlap is false overlapping instances are discarded. The resulting list represents the instances of *graph_1* in *graph_2*.

For illustrative purpose suppose we have as input graphs those shown in Figure 4.5 and Figure 4.6. Input *graph_1* has 3 vertices and 2 edges, and *graph_2* has 8 vertices and 7 edges.

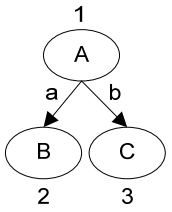

Figure 4.5. SGISO - input graph_1.

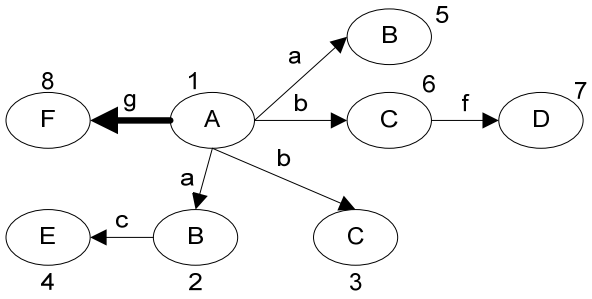

Figure 4.6. SGISO - input graph_2.

Running *SGISO* with the overlap parameter set to false, it finds 1 instance of *graph_1* (see Figure 4.7) in *graph_2*. Figure 4.8 shows the discovered instance in *graph_2*.

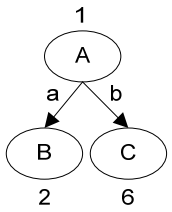

Figure 4.7. SGISO - no overlap.

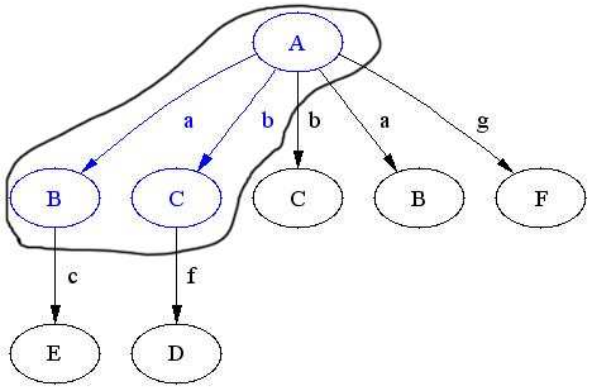

Figure 4.8. SGISO - no overlap, 1 instance in graph_2.

Now, running *SGISO* with the overlap parameter set to true, the algorithm finds the 4 instances shown in Figure 4.9. We can see that each instance has the vertices labeled as *A*, *B*, and *C* but they represent different vertices (they have different *ID* vertices). For example, the first instance has the *ID* vertices 1, 5 and 3; the second instance has the *ID* vertices 1, 5 and 6 respectively, and so on. Figure 4.10 shows the 4 discovered instances in *graph_2*.

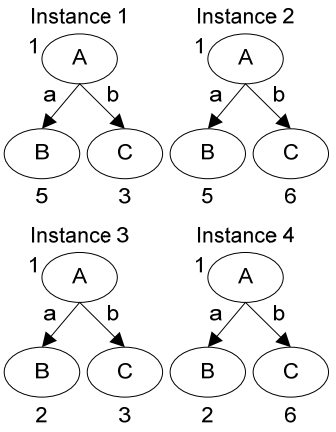

Figure 4.9. SGISO - overlap.

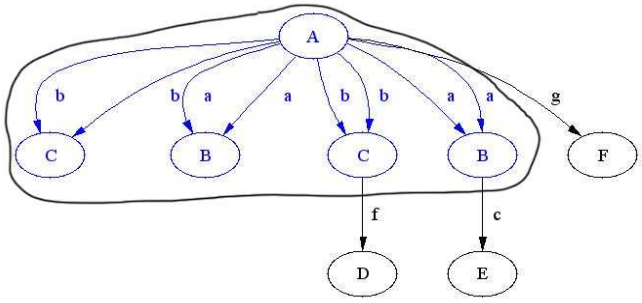

Figure 4.10. SGISO - overlap, 4 instances in graph_2.

Minimum Description Length (*MDL*)

The *MDL* principle states that the best theory to describe a set of data is that theory which minimizes the description length of the entire dataset. We define the *MDL* of a graph to be the number of bits necessary to completely describe the graph. The minimal encoding of the graph in terms of bits is computed as follows:

$$MDL = vertexBits + rowBits + edgeBits$$

Where *vertexBits* represents the number of bits needed to encode the vertex labels of the graph, *rowBits* represents the number of bits needed to encode the rows of the adjacency matrix *A* (the matrix represents the graph connectivity), and *edgeBits* represents the number of bits needed to encode the edges represented by the entries $A[i,j] = 1$ of the adjacency matrix *A*.

The standalone *MDL* function computes the description length (*dL*) of *graph_1*, *graph_2* and *graph_2* compressed with *graph_1* along with the final *MDL* compression measure:

$$dL(graph\_2) / dL(graph\_1) + dL(graph\_2 \mid graph\_1)$$

dL(*graph_2*|*graph_1*) represents the value of *graph_2* compressed with *graph_1*. The overlap feature has a direct effect to compute this value because the number of instances for compressing *graph_2* is based in the instances list returned by the FindInstances function.

As we have commented, the Subdue system implements a compress graphs function named CompressGraph. The inputs of the function are the graph to be compressed, and the

substructure instances used to compress the graph. The function returns a new graph, which is the given graph compressed with the given substructure instances.

As part of the graph compressing phase, *SUB* vertices replace each instance of the substructure, and *OVERLAP* edges are added between *SUB* vertices of overlapping instances. Edges connecting to overlapping vertices are duplicated, one per each instance involved in the overlap. The procedure to add *OVERLAP* edges evaluates two cases: first, if two instances overlap at all, then an undirected *OVERLAP* edge is added between them. Second, if an external edge points to a vertex shared between multiple instances, then duplicate edges are added to all instances sharing the vertex. The procedure to add duplicate edges is based in the follows pseudocode:

```
if edge connects SUB1 to external vertex then
  add duplicate edge connecting external vertex to SUB2
else if edge connects SUB1 to another (or same) vertex in SUB1 then
  add duplicate edge connecting SUB1 to SUB2
  if other vertex is unmarked (overlapping and already processed) then
    add duplicate edge connecting SUB2 to SUB2
  if edge connects SUB1 to the same vertex in SUB1 (self edge) then
    add duplicate edge connecting SUB2 to SUB2
    if edge directed then
      add duplicate edge from SUB2 to SUB1
```

To show the role plays by overlap in the standalone *MDL* function (focusing in the generated compressed graph) we will present, in the following figures, examples of the different cases of validation that are implemented to face this feature. Figure 4.11 shows the input *graph_1*, this graph will be used in all the examples as the first input graph. It has 3 vertices and 2 edges.

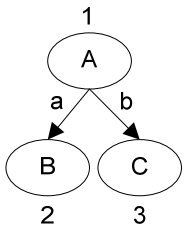

Figure 4.11. *MDL* - input graph_1.

In our first example we will use as *graph_2* the graph shown in Figure 4.12. This graph has 8 vertices and 7 edges. As we can see our graph has a "remarked" edge, the directed edge labeled as *g* between vertex 1 (labeled as *A*) and vertex 8 (labeled as *F*). This edge represents the current case of validation (in the future it will be named the *guide edge*).

As part of the CompressGraph algorithm, there is a function named AddOverlapEdges that adds edges to the compressed graph, describing overlapping instances of the substructure in the given graph, based in two conditions. First, if two instances overlap, then an undirected edge *OVERLAP* is added between them. Second, if an external edge points to a vertex shared between multiple instances, then duplicate edges are added to all instances sharing the vertex. If the second condition is true, the AddOverlapEdges function calls a new function named AddDuplicateEdges. The purpose of this function is to add duplicate edges based on overlapping vertex between substructures.

As we can see, the layout of the edge (that we call the *guide edge*) is important for the global functionality of the compress graph algorithm. So, in the following examples we will change the guide edge for illustrating the different validation cases.

Returning to our example, if we run the *MDL* program with the overlap parameter set to false, our final compressed graph is shown in Figure 4.13. Since overlap is not allowed between instances, the FindInstances function finds 1 instance of *graph_1* that match *graph_2*. In the graph this instance was replaced by a vertex *SUB*, so we have only 1 vertex of this type. There are not edges *OVERLAP*, and no edges were duplicated.

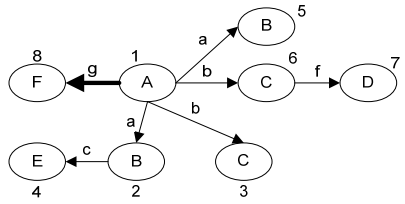

Figure 4.12. *MDL* example 1 - input graph_2.

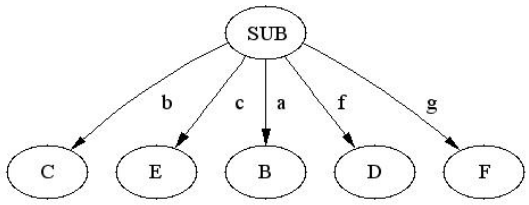

Figure 4.13. *MDL* example 1 - no overlap.

In the next example our *graph_2* is shown in Figure 4.14. It is the same graph used in the previous example, but this time we run *MDL* with the overlap parameter set to true. The generated compressed graph is shown in Figure 4.15. The FindInstances function finds 4 instances since the overlap between instances is allowed. In the graph the 4 instances were replaced by 4 vertices *SUB*. There are 5 edges *OVERLAP*, an edge *OVERLAP* between two vertices *SUB* tells us that these substructures overlap. By "substructures overlap or overlapping substructures" we refer that the instances they represent have common vertices.

In the example, there is a directed edge (edge *g*, our *guide edge*) connecting two vertices, one of them belonging to a substructure (in exact term, to an instance of a substructure) and the other one no belonging to the substructure (vertex *F*), is an external edge; the

discovered instances were 4 (overlap is allowed), so, in the graph there are 4 directed edges starting from each vertex *SUB* (the direction of the edge is preserved) to the vertex *F*.

A similar procedure was implemented for edge *f* (connecting vertex *C* and vertex *D*, vertices number 6 and 7 respectively) and edge *c* (connecting vertex *B* and vertex *E*, vertices number 2 and 4, respectively). They were duplicated since these edges connect vertices belonging to a substructure to vertices no belonging to the substructure (external edges). In the figure we can see that vertex *D* has 2 edges *f* connecting it to 2 vertices *SUB* (preserving the direction of the edge) and vertex *E* has 2 edges *c* connecting it to 2 vertices *SUB* (remember that 4 instances were discovered, overlap is allowed).

Finally, our compressed graph has 7 vertices: 4 vertices *SUB*, 1 vertex *F*, 1 vertex *D*, and 1 vertex *E*; and 13 edges: 5 edges *OVERLAP*, 4 edges labeled as *g*, 2 edges labeled as *f*, and 2 edges labeled as *c*.

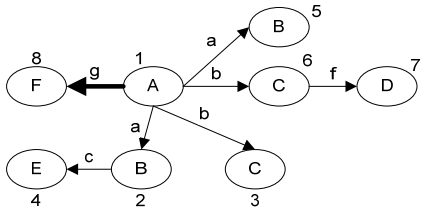

Figure 4.14. *MDL* example 2 - input graph_2.

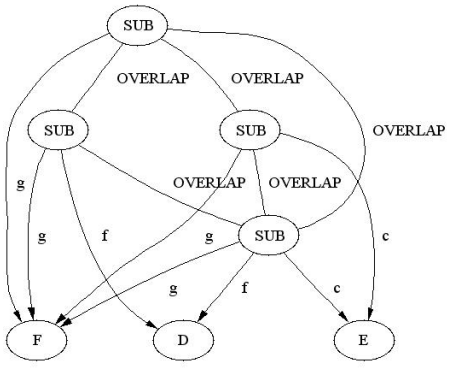

Figure 4.15. *MDL* example 2 - overlap.

For the following examples the overlap parameter is always set to true.

Figure 4.16 shows our new *graph_2*, we only changed the direction of the *guide edge*. Now it starts from vertex *F* (number 8) to vertex *A* (number 1). The generated compressed graph is shown in figure 4.17. In the compressed graph we observe that, this time, the 4 edges *g* start from vertex *F* to the 4 vertices *SUB* (the direction of the edge is preserved). Everything else remains as the previous example.

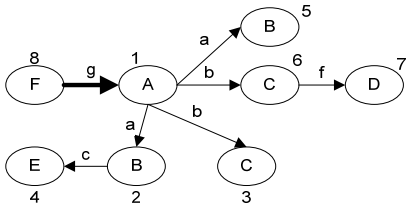

Figure 4.16. *MDL* example 3 - input graph_2.

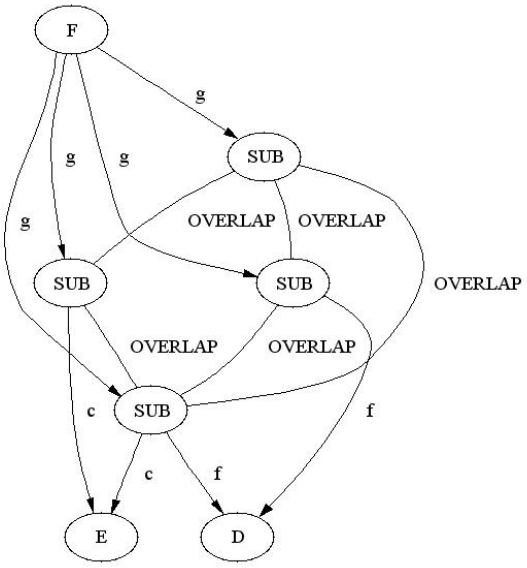

Figure 4.17. *MDL* example 3 - overlap.

For the next example, we only change from a directed *guide edge* to an undirected one. Our *graph_2* is the graph shown in Figure 4.18. By running *MDL* we generate the compressed graph presented in Figure 4.19. The only modification is that the edges between vertex *F* and the 4 vertices *SUB* are undirected edges. Everything else remains unchanged.

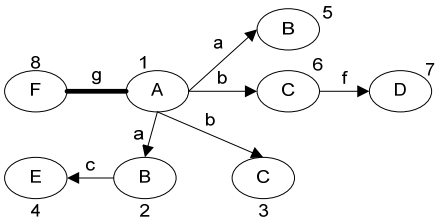

Figure 4.18. *MDL* example 4 - input graph_2.

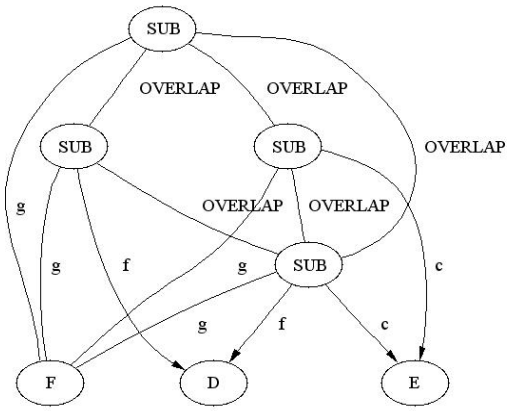

Figure 4.19. *MDL* example 4 - overlap.

For illustrating a new outcome of the overlap feature in the standalone *MDL* function, we change our *graph_2* as follows:

- Vertex number 8 labeled as *F* is deleted.
- Our *guide edge* labeled as *g* is deleted.
- A new edge labeled as *g* (our *guide edge*) is added between vertex number 1 labeled as *A* and vertex number 3 labeled as *C*. For the current example we use a directed edge.

Our *graph_2* is shown in Figure 4.20. The graph has 7 vertices and 7 edges. The "new" edge has the characteristic that it is an edge between two vertices belonging to the same substructure and in some cases, as we will see, they belong to overlapping substructures. Remember that overlap is allowed between instances, so the FindInstances function finds 4 instances.

The generated compressed graph is shown in Figure 4.21. We mentioned that our *guide edge* joins 2 vertices belonging to a same substructure (vertex number 1 labeled as *A* and vertex 3 labeled as *C*, and in some cases these vertices belong to overlapping substructures. So, in the graph there are 6 edges labeled as *g*, 4 edges joining vertices *SUB* (telling us that they overlap), and two self edges in 2 vertices *SUB*. These last 2 edges are our new case of validation. The self edge tells us that inside the substructure, there is an edge joining two vertices belonging to the same substructure. The direction of the edges does not matter since this is an edge inside the substructure (an internal edge). The other duplicated edges (i.e., edge *c* and edge *f*) were computed using the previous described procedure.

Finally, our compressed graph has 6 vertices: 4 vertices *SUB*, 1 vertex *D*, and 1 vertex *E*; and 15 edges: 5 edges *OVERLAP*, 6 edges labeled as *g*, 2 edges labeled as *f*, and 2 edges labeled as *c*.

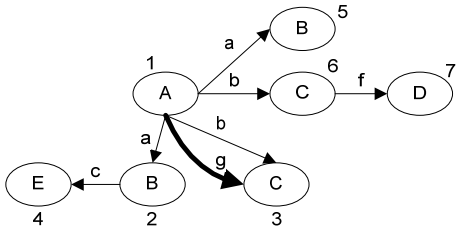

Figure 4.20. *MDL* example 5 - input graph_2.

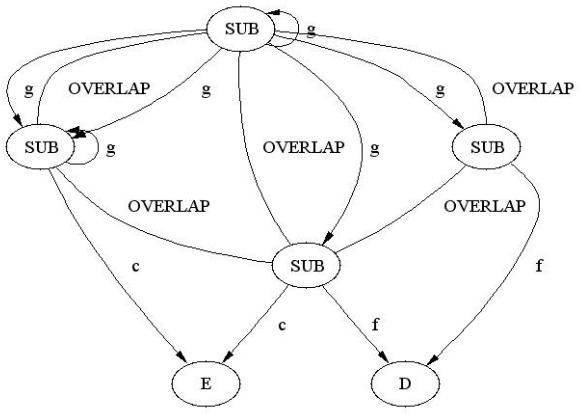

Figure 4.21. *MDL* example 5 - overlap.

For the example shown in Figure 4.22, we only changed the direction of the *guide edge*. Now it starts from vertex number 3 labeled as *C* to vertex number 1 labeled as *A*. The generated compressed graph is presented in Figure 4.23. The generated compressed graph is the same one that in the previous example. This is telling us that just changing the direction of the *guide edge* does not have effect in the resulting compressed graph.

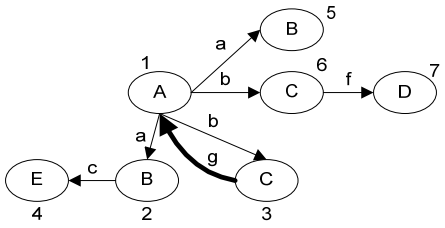

Figure 4.22. *MDL* example 6 - input graph_2.

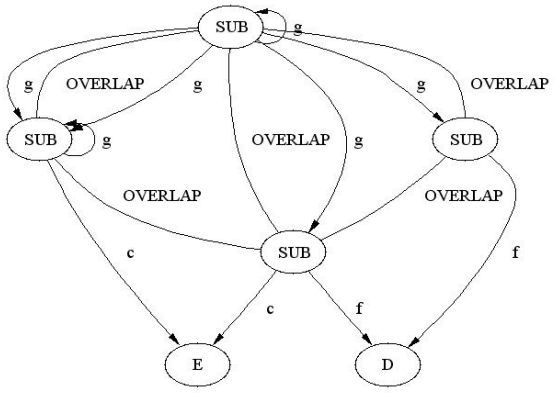

Figure 4.23. *MDL* example 6 - overlap.

For the next example, we only change from a directed *guide edge* to an undirected one. Our *graph_2* is the graph shown in Figure 4.24. The generated compressed graph is presented in Figure 4.25. The single change is that all edges *g* are now undirected edges. Everything else remains unchanged.

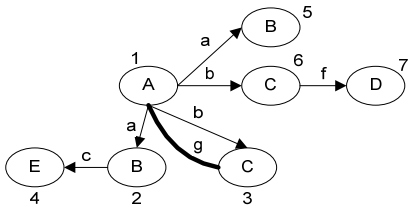

Figure 4.24. *MDL* example 7 - input graph_2.

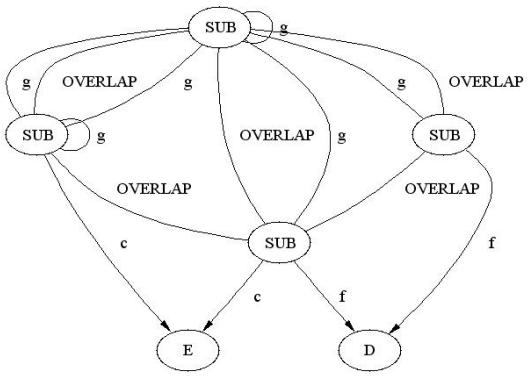

Figure 4.25. *MDL* example 7 - overlap.

Now, for illustrating a new outcome of the overlap feature in the standalone *MDL* function (self edge in a vertex shared by instances of a substructure), we change our *graph_2* as follows:

- Edge labeled as *g* between vertex number 1 labeled as *A* and vertex 3 labeled as *C* is deleted.
- A new edge labeled as *g* (our *guide edge*) is added between vertex number 1 labeled as *A* and vertex number 1 labeled as *A* (a self edge). For the current example a directed edge.

Our *graph_2* is shown in Figure 4.26. The graph has 7 vertices and 7 edges. The "new" edge has the characteristic that is a self edge; it starts and ends in the same vertex. This vertex belongs to overlapping substructures, so we have 4 substructures sharing it.

The resulting compressed graph is shown in Figure 4.27. Since our *guide edge* is a self edge the compressed graph has 10 edges labeled as *g*. There are 6 edges joining vertices

SUB (telling us that they are overlapping substructures), and 4 self edges in 4 vertices *SUB* telling us that they have an edge which origin and destination vertices are the same.

We note in the compressed graph a special characteristic between some vertices *SUB*. We refer that some pairs of vertices *SUB* are joined by 2 edges labeled as *g*. One of them starts in the first vertex *SUB* and ends in the second one. The second edge has a vice versa direction. This is because they are overlapping substructures with a common vertex and this vertex is the source and destination of an internal edge.

Finally, our compressed graph has 6 vertices: 4 vertices *SUB*, 1 vertex *D*, and 1 vertex *E*; and 19 edges: 5 edges *OVERLAP*, 10 edges labeled as *g*, 2 edges labeled as *f*, and 2 edges labeled as *c*.

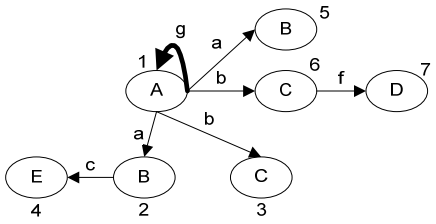

Figure 4.26. *MDL* example 8 - input graph_2.

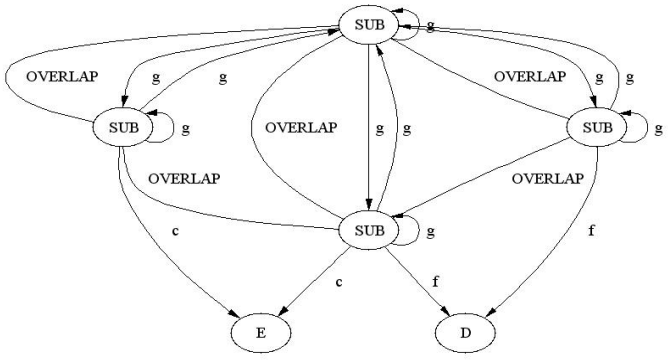

Figure 4.27. *MDL* example 8 - overlap.

For the last example, we use as *graph_2* the one shown in Figure 4.28. We only change our *guide edge* from a directed edge to an undirected one. Figure 4.29 shows the generated compressed graph. There are 2 differences between this compressed graph and the previous

one. First, the edges labeled as *g* are undirected edges. Second, the special characteristic for some vertices *SUB* is implemented as follows: Currently, we note that those vertices *SUB* joined by 2 edges labeled as *g* in the previous example, now, they are joined by just 1 undirected edge labeled as *g*. Since the *guide edge* is undirected we only need 1 edge for representing them because they are overlapping substructures with a common vertex and this vertex is the source and destination of an internal undirected edge. The 4 vertices *SUB* have self edges, but they are undirected ones this time. Everything else remains unchanged.

Finally, our compressed graph has 6 vertices: 4 vertices *SUB*, 1 vertex *D*, and 1 vertex *E*; and 16 edges: 5 edges *OVERLAP*, 7 edges labeled as *g*, 2 edges labeled as *f*, and 2 edges labeled as *c*.

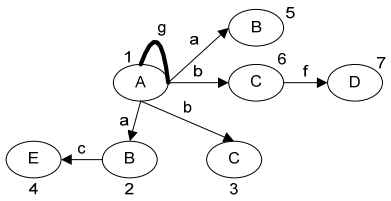

Figure 4.28. *MDL* example 9 - input graph_2.

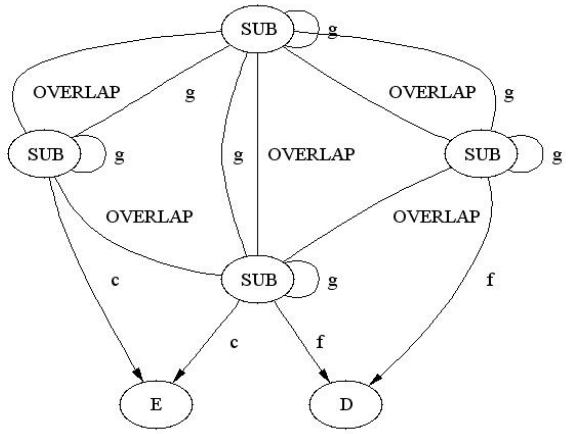

Figure 4.29. *MDL* example 9 - overlap.

4.3 Limited Overlap

As we have described in the previous section, the overlap feature plays a preponderant role in the Subdue's substructures discovery system. As a consequence, the generated results are also conditioned, in a high percentage, to this parameter. However, as we have seen, it is implemented to allow overlap among any instances of a substructure or among all the instances of a substructure. In this context, we propose a new approach named limited overlap. The major feature in this approach is to give the user the means to specify the set of vertices where an overlap is allowed. These vertices may represent significant elements in the context we work with.

In the following subsections we will present the motivation, advantages, new algorithm, and an example of the new overlap feature implemented in the Subdue system.

Motivation

As we have already commented, the current overlap feature in Subdue is implemented in an orthodox way: all or nothing. It means that Subdue allows the overlap among all the instances sharing at least one vertex or that Subdue does not allow (discard) the overlap among instances sharing at least one vertex.

But we argue that a third option is needed, an option where the user has the capability to set over on which vertices the overlap will be allowed, it is a limited overlap. We visualize directly three motivations issues to propose the implementation of the new algorithm that will be explained in the following subsections:

- Search space reduction.
- Processing time reduction.
- Specialized overlapping pattern oriented search.

In order to help us to describe the characteristics of the limited overlap we will use the graphs show in Figure 4.30 and 4.31 as input graphs in the future examples. Input graph *PS_1* (**P**redefined **S**ubstructure number 1) has 2 vertices and 1 edge, input graph *PS_2* (**P**redefined **S**ubstructure number 2) has also 2 vertices and 1 edge, and finally *graph_3* has 9 vertices and 8 edges.

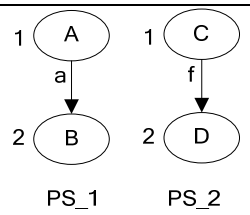

Figure 4.30. Limited overlap - input graphs PS_1 and PS_2.

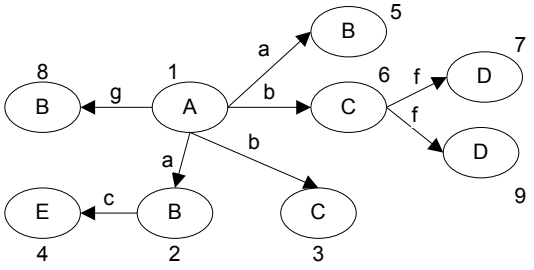

Figure 4.31. Limited overlap - input graph_3.

1) Search space reduction. In knowledge discovery systems using a graph-based approach, the data mining algorithm uses graphs as a knowledge representation; the search space of a graph-based data mining algorithm consists of all the subgraphs that can be derived from its input graph.

The substructures discovery process in Subdue begins with the creation of the substructures matching a single vertex in the graph (one for each of the different labels in the graph).

Each iteration through the algorithm selects the best substructures and expands the instances of these substructures by one neighboring edge (or an edge and new vertex) in all possible ways.

But as part of the process to select the best substructures and then to expand them there exists a filter phase. In this phase according to the overlap parameter the instances of a substructure are evaluated: if overlap is set to true then overlapping instances are kept, otherwise if overlap is set to false then overlapping instances are discarded.

For example, suppose we want to search the instances of *PS_1* and *PS_2* in *graph_3*. With the overlap set to false Subdue discovers 2 instances, one instance for each predefined substructure as we can see en Figure 4.32. *PS_SUB_X* is the nomenclature used by Subdue to identify it is an instance of Predefined Substructure SUB_X; where SUB_X means it is the **SUB**structure number X. But, if overlap is set to true, Subdue discovers 4 instances, 2 instances for each predefined substructure (see Figure 4.33). Finally, suppose the user wants to search instances of *PS_1* and *PS_2* in *graph_3* but this time he considers that vertices *A* have a higher relevance (for example, a remarkable spatial object in a spatial database) so he proposed to use a limited overlap, the overlap will be allowed just among instances containing vertices *A*. We can see that *PS_1* has a vertex *A*, thus this time Subdue finds 3 instances, 2 instances of *PS_1* and 1 instance of *PS_2* as shown in Figure 4.34. In the figure we can observe that instance 1 and instance 2 share the vertex number 1 labeled as *A*. For *PS_2* Subdue finds 1 instance since the other one (instance number 4 in Figure 4.33) has also the vertex number 6 labeled as *C*, but the overlap is just allowed among vertices *A*.

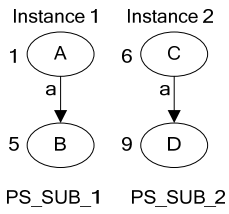

Figure 4.32. No overlap - SGISO.

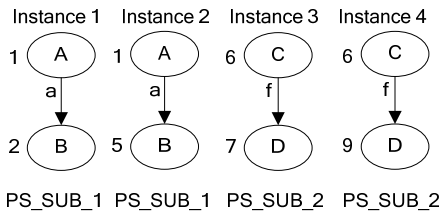

Figure 4.33. Overlap - SGISO.

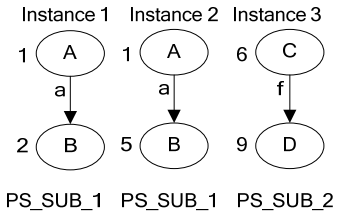

Figure 4.34. Limited overlap PS_1 - SGISO.

We have mentioned that the best discovered substructure by Subdue (by iteration) can be used to compress the input graph, which can then be input to another iteration. After several iterations, Subdue builds a hierarchical description of the input data where later substructures may be defined in terms of substructures discovered on previous iterations. We have also commented that each iteration through the algorithm selects the best substructures and expands the instances of these substructures by one neighboring edge (or an edge and new vertex) in all possible ways. So the number of instances of the substructures defines the search space (by iteration) in the substructure discovery process.

As we can see in our example, by using the limited overlap we obtain a search space reduction (with overlap set to true), since the number of instances becoming candidates to be expanded is selected according to the allowed values given by the user.

2) Processing time reduction. The reduction in the number of instances becoming candidates to be expanded results in a search space reduction, and this effect also has a new outcome, a processing time reduction.

Allowing overlap slows Subdue considerably since the number of candidate instances to expand, to evaluate, to match, and to discover increase as we have seen. However, by the implementation of the limited overlap, the number of instances to be processed in these phases decrease resulting also in a processing time reduction in the overall substructure discovery process.

3) Specialized overlapping pattern oriented search. We have also commented that the limited overlap gives the user the capabilities to define the set of interesting elements over which the overlap will be allowed (the elements are represented as vertices in the graph according to the proposed model) so the algorithm will discard the overlapping elements that the user does not considerer significant.

In the example presented in Figure 4.34, the judgment of the user is that vertices *A* have a higher relevance so he proposes to use a limited overlap, the overlap will be allowed just among instances containing vertices *A*. This consideration is a personal decision of the user according to the work context (in many cases with the support of a domain expert). For instance, a remarkable element may refer to a spatial object in a spatial database or to some characteristic defining a particular topic of a dataset.

Therefore, the limited overlap gives the user the mechanics to implement a pattern oriented search. The user delimits the set of elements that will have a preponderant role in the substructure discovery process. But this characteristic gives also a new advantage, the patterns evaluation process is simplified since the set of generated results is smaller because it is focused over the user requirements.

Algorithm

As we have described, the limited overlap gives the user the capabilities to define the set of elements (vertices in the graph) where overlap among instances is allowed. The process starts reading the set of vertices which are integrated to the Subdue's parameters as the limited overlap label list. During the substructure discovery process a filter phase is performed. This phase evaluates (and possible discards) based on the overlap parameter the list of discovered instances.

If the substructure discovered process is composed by several iterations, after each of them, the overlap label list is updated to integrate the new vertices where overlap is also allowed. Remember that at the end of each iteration, the best discovered substructure by Subdue is used to compress the input graph, which can then be input to another iteration of Subdue. After several iterations, Subdue builds a hierarchical description of the input data where later substructures are defined in terms of substructures discovered on previous iterations. Therefore, if the best discovered substructure, in each iteration, has a vertex where overlap is allowed then that substructure is added to the limited overlap label list.

```
Limited Overlap (instance1, instance2)
  //Global process
  while ((elementsInstance1 < totalElementsInstance1) AND NOT endProcess){
      //Instance1's vertex
      vertex1 = vertex from instance1

      //Instance2's vertex
      while (elementsInstance2 < totalElementsInstance2)
          vertex2 = vertex from instance2

      //Instances overlap
      if (vertex1 = vertex2) then{
          intancesOverlap = true
          endProcess = true
      }

      //Instances overlap, check by limited overlap
      if (instancesOverlap AND overlapLabelList NOT EMPTY) then{
          if (vertex1 in overlapLabelList) then
              //It is a limited overlap
              limitedOverlap = true
          else
              //Process continues until all vertex1 are validated
              endProcess = false
      }
  }
  return instancesOverlap, limitedOverlap
```

Figure 4.35. Limited overlap in the Subdue system.

In the validation process each vertex belonging to instance1 (*vertex1*) is validated against all vertices belonging to instance2 (*vertex2*). If *vertex1* and *vertex2* are the same then they are overlapping instances. If the instances overlap then *vertex1* is validated against the list containing the set of vertices allowed for overlapping (the *overlapLabelList*). If *vertex1* exists in *overlapLabelList* then it is a limited overlap.

Example

To illustrate the functionality of the new algorithm, we will present some examples using the new version of Subdue implementing the limited overlap feature. The input graphs for

the examples are those shown in Figure 4.30 and Figure 4.31; the idea is to perform a pattern oriented search by discovering patterns in *graph_3* but based on *PS_1* and *PS_2* (our predefined patterns).

Subdue implements a pattern oriented search by, initially, finding all instances of *PS_1* and *PS_2* in *graph_3*. The next step is to compress *graph_3* using the found instances of each *PS_1* and *PS_2*. Figure 4.36 shows the compressed graph with the overlap parameter set to false. As we can see, Subdue found 1 instance of *PS_1* (i.e., vertex *PS_SUB_1*) and 1 instance of *PS_2* (i.e., vertex *PS_SUB_2*) since overlap among instances is not allowed (see also Figure 4.32).

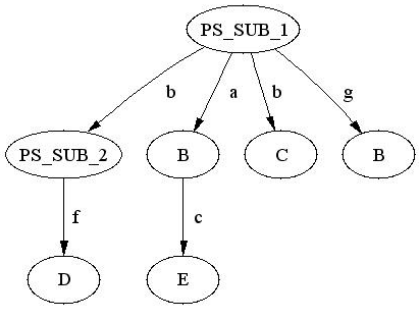

Figure 4.36. No overlap - compressed graph.

Finally, the compressed graph becomes the input graph to discover substructures. Figure 4.37 shows the best 3 substructures (default number of reported substructures) found by Subdue according to its substructure discovery system (see Section 4.1 for details). Each substructure has 1 instance; it means that there exists 1 repetition of each of them in the input graph (in the example we use exact graph match, but Subdue allows also inexact graph match). The first one (labeled as "a"), is composed by 2 vertices: 1 vertex *B*, and 1

vertex *E*; and 1 edge labeled as *c*. Subdue reports this substructure as the best discovered substructure because it is the best that minimize the description of the input graph according to the *MDL* principle. The second one (labeled as "b") is composed by 4 vertices: 1 vertex *PS_SUB_1*, 1 vertex *C*, and 2 vertices *B*; and 3 edges labeled as *a*, *b*. *g*. The vertex *PS_SUB_1* is itself a substructure composed by two vertices: 1 vertex *A*, and 1 vertex *B*; and 1 edge labeled as *a*. We have already commented that later substructures may be defined in terms of previous discovered substructures, or in term of predefined substructures as in this example. Finally, the third one (labeled as "c") is composed by 4 vertices: 1 vertex *PS_SUB_1*, 1 vertex *PS_SUB_2*, 1 vertex *C*, and 1 vertex *B*; and 3 edges: 2 edges labeled as *b*, and 1 edge labeled as *g*. Once again vertices *PS_SUB_1* and *PS_SUB_2* are themselves substructures.

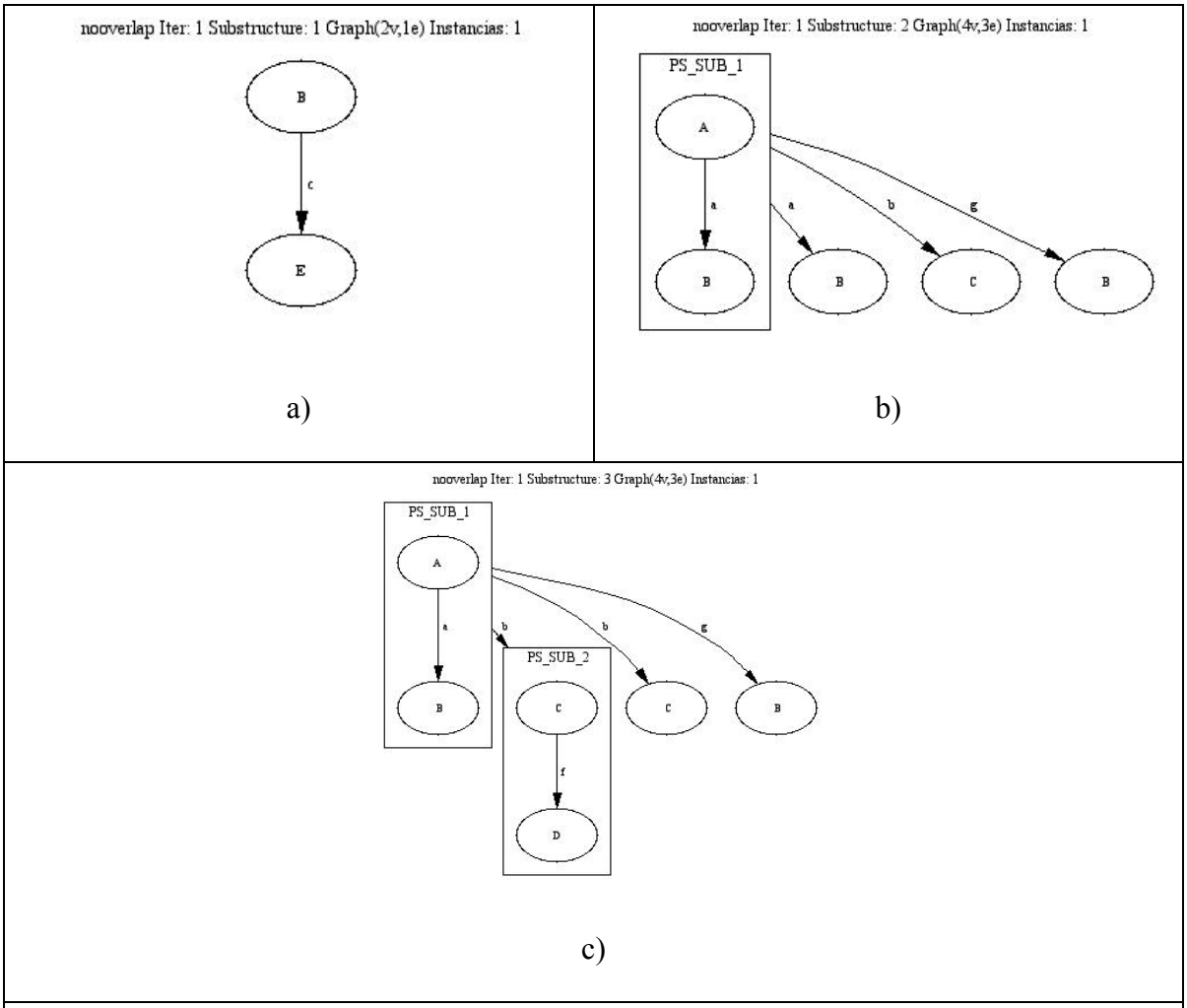

Figure 4.37. No overlap - discovered substructures.

For the next example we set overlap to true. The generated compressed graph is shown in Figure 4.38. Now, Subdue finds 2 instances for each *PS_1* and *PS_2* (see Figure 4.33 for details).

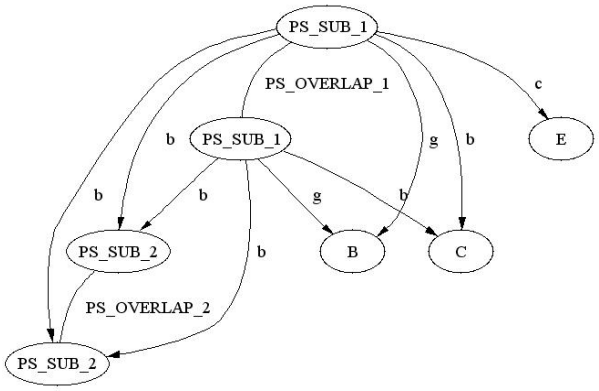

Figure 4.38. Overlap - compressed Graph.

Figure 4.39 shows the best 3 substructures discovered by Subdue. The first and second ones (labeled as "a" and "b") are themselves the predefined substructure *PS_SUB_2* and *PS_SUB_1* respectively with 2 instances each of them. The third one (labeled as "c") is a substructure composed by 2 vertices and 1 edge with 1 instance.

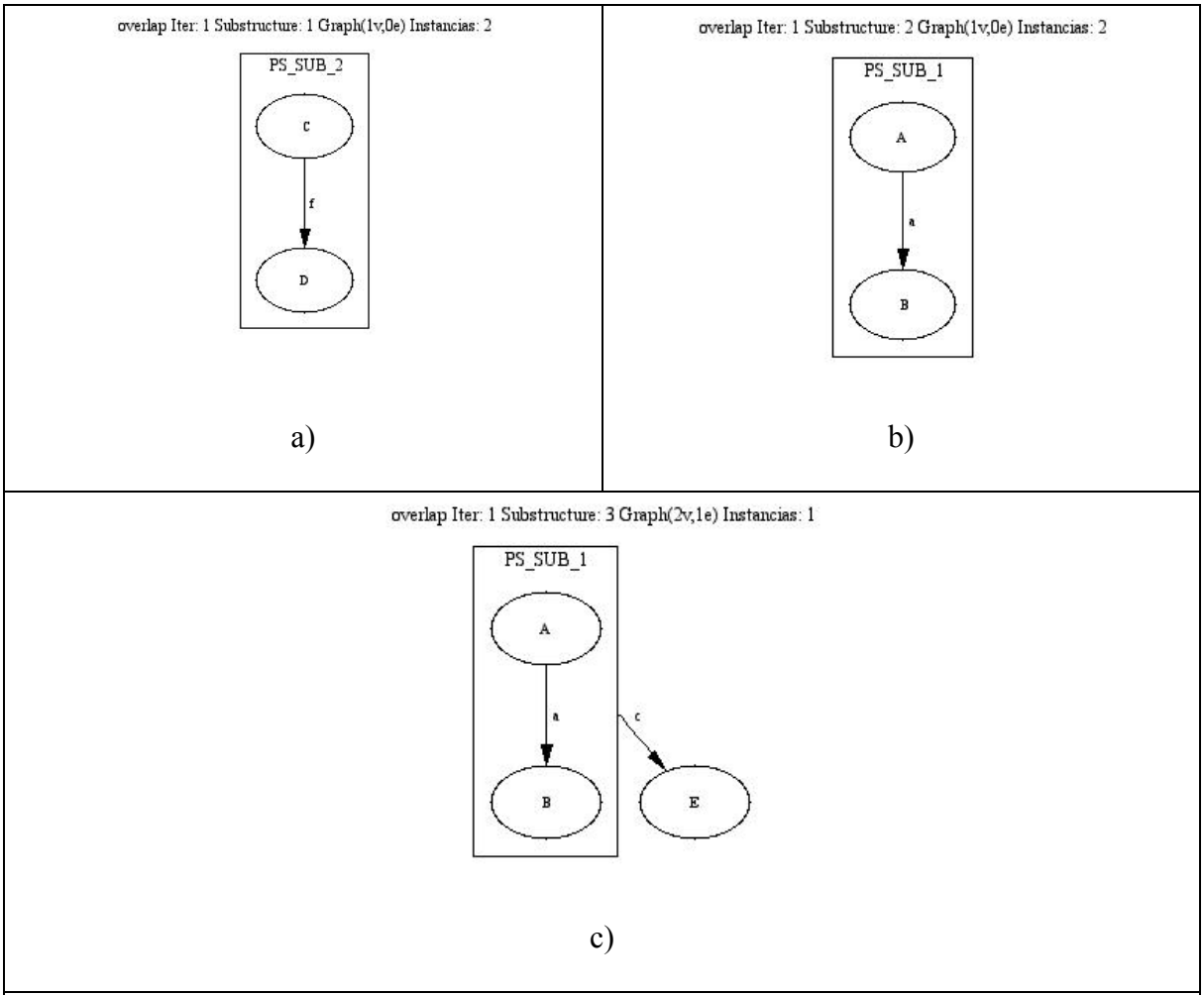

Figure 4.39. Overlap - discovered substructures.

Our next example is implemented by using the limited overlap feature. Suppose the user wants to perform a specialized overlapping pattern oriented search: he only wants to allow overlapping instances in vertices representing *PS_1*. The generated compressed graph is shown in Figure 4.40. As a result of the restriction for overlapping instances Subdue finds 2 instances of *PS_1* (they share the allowed vertex *A*) and just 1 instance of *PS_2* (since they share a not allowed vertex *C*). The found instances are shown in Figure 4.34.

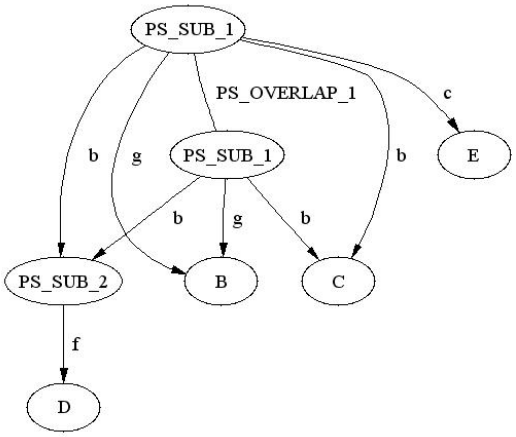

Figure 4.40. Limited overlap PS_1 - compressed graph.

Figure 4.41 shows the best 3 substructures discovered by Subdue from this graph using the limited overlap. In the figure we can see that Subdue reports as the best substructure *PS_SUB_1* with 2 instances (labeled as "a"). This is consequence of the integration of the compressed graph since it has 2 vertices *PS_SUB_1*. The second reported substructure (labeled ad "b") is composed by 2 vertices: 1 vertex *PS_SUB_1*, and 1 vertex *E*; and 1 edge labeled as *c*. The last one (labeled as "c") is composed also by two vertices *PS_SUB_1*; and 1 edge labeled as *PS_OVERLAP_1*.

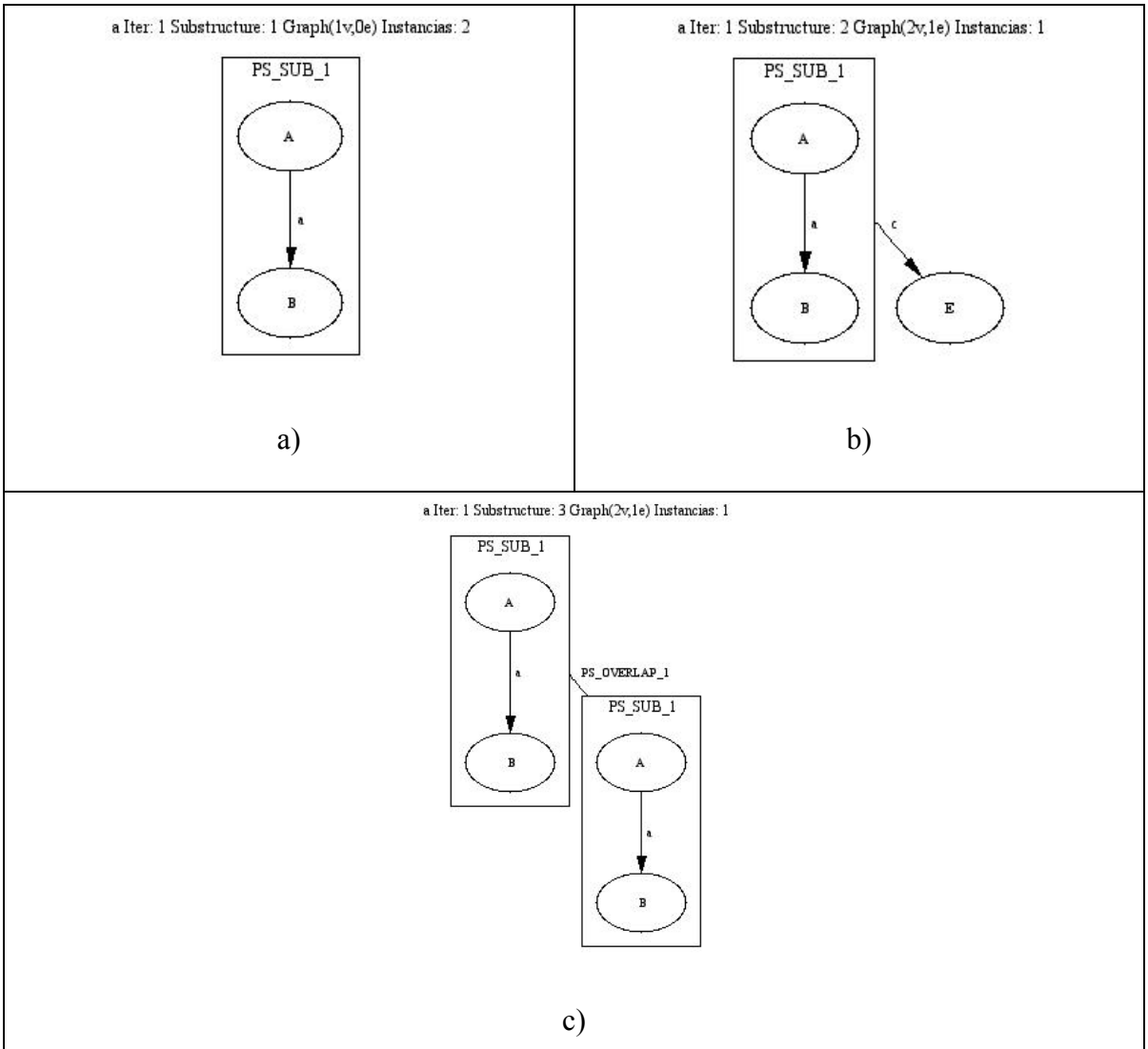

Figure 4.41. Limited overlap - discovered substructures.

4.4 Conclusion

In this chapter we have described the characteristics and functionality of our graph-based data mining tool, the Subdue system. We introduced a new algorithm named limited overlap. We presented some examples showing the functionally of the new algorithm using an artificial dataset. Examples using real world data will be presented in Chapter 6. These

examples are developed by using two test domains: a Puebla downtown population census from the year of 1777 and a Popocatépetl volcano database.

In the next chapter we introduce a prototype system implementing the proposed model for representing spatial data, non-spatial data and spatial relations among the spatial objects as a whole dataset using a graph-based representation.