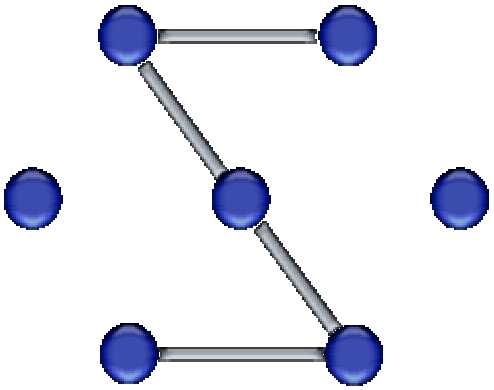

SUBDUE Manual

Version 1.5

Table of Contents

Revision History

Release Version	Date	Revisions
1.0	April 27, 2005	Initial release
1.1	January 27, 2006	Added –inc and –compress options; added MPI section.
1.2	April 22, 2006	Corrected typos and missing reference.
1.3	May 29, 2006	Updated SUBDUE references.
1.4	January 11, 2007	Corrected typos and minor formatting changes.
1.5	June 6, 2011	Minor updates to reflect changes in new Subdue version 5.2.2.

Preface

Conventions

The following documentation conventions are followed within this document.

bold underlined text signifies notes or comments to the reader.

Italicized text signifies file names, directories or programs.

Bold italicized text signifies a reference to another document.

1. Introduction

The following document provides a manual on how to use the SUBDUE system.

1.1 Overview

The SUBDUE graph-based pattern learning system finds relational patterns in data represented as a graph. While the details of how SUBDUE works internally will not be discussed here (you can refer to other documents on the SUBDUE web-site for more information), this document will provide you with the specifics on how to install and run the application.

This document contains the following sections:

- Chapter 2: instructions on how to download and install SUBDUE
- Chapter 3: layout of the required graph input file
- Chapter 4: instructions on running SUBDUE
- Chapter 5: various notes and issues regarding the SUBDUE application
- Appendix A: terminology

1.2 Reference Documents

- ***SUBDUE Home Page:*** http://www.subdue.org/
- ***AT&T Labs GraphViz:*** http://www.graphviz.org/
- ***MPI:*** http://www.mcs.anl.gov/mpi/

2. Downloading and Installing

In order to build and run the SUBDUE application, you must first download the appropriate files.

2.1 Download

The SUBDUE system, including documentation, papers, and research, can be found on the SUBDUE home page (http://www.subdue.org/).

In order to get the latest copy of the application, you must choose the Download option located on the left-hand side of the SUBDUE home page. After clicking the Download link, you will be redirected to the "Download" page, which contains a link to the latest source code for the SUBDUE application.

In order to pull a release of the SUBDUE application, you must click on the *<release>.zip* file (where <*release*> is the version of SUBDUE that you want) located under the "Source" heading. Depending upon your browser and settings, you will either be able to just click on the link, which will then prompt you to either open the file or save it to your local disk, or you will be able to right-click on the link and either open or save the file.

2.2 Files

Once you have chosen a release, downloaded the archive, and unzipped the files, the following directory/file structure is created:

./bin/ -- directory of executables (initially empty)

./copyright.txt-- file containing the SUBDUE copyright notice

./docs/ -- directory containing this manual

./graphs/ -- directory containing some sample graph input files

./readme.txt – file containing directions on how to build SUBDUE, as well as the version histories

./src/ - directory containing the source code and make files

(**Note**: All of this is actually created under another directory called *subdue*<*x*>, where <*x*> is the major release number.)

2.3 Install

After downloading and unzipping the files, you can now install the SUBDUE application. Installation consists of actually building the application so that it is now native to your Unix system.

SUBDUE uses the standard *make* facility to build its application. In order to build the application, you should perform the following steps:

1. Change directory to *subdue-<release>/src*
2. At the command prompt, enter: *make*. This will compile the SUBDUE programs.
3. At the command prompt, enter: *make install*. This will copy the executables to the *subdue-<release>/bin* directory
4. At the command prompt, enter: *make clean*. This will clean up the *src* directory (removing object files).

3. Data Format

The following section describes the format of the input graph that must be supplied in order to run the SUBDUE application.

3.1 Input

The input to the SUBDUE application is comprised of a textual representation of a graph.

3.1.1 Graph Format

The input file can consist of one or more graphs. Each graph is prefaced (on a line by itself) by either an "*XP*", indicating a positive example, or "*XN*" indicating a negative example. If the first (or only) graph in the file is positive, then the "*XP*" can be omitted.

3.1.1.1 Vertices

Each graph is a sequence of vertices and edges. A vertex is defined as:

```
v <#> <label>
```

where <*#*> is a unique vertex ID for the graph and <*label*> is any string or real number. Strings containing white-space or the comment character (see below) must be surrounded by double-quotes. Vertex IDs for a graph must start at 1 and increase by 1 for each successive vertex.

It should also be noted that there must be at least one vertex defined before any edges are defined.

3.1.1.2 Edges

An edge is defined as one of the following:

e <vertex 1 #> <vertex 2 #> <label>

d <vertex 1 #> <vertex 2 #> <label>

u <vertex 1 #> <vertex 2 #> <label>

where <*vertex 1 #*> and <*vertex 2 #*> are the vertex ID's for the source vertex and the target vertex respectively, and <*label*> is any string or real number. Strings containing white-space or the comment character (see below) must be surrounded by double-quotes. Edges beginning with "*e*" are assumed directed unless the option "*-undirected*" is specified at the command line (see next section), in which case all "*e*" edges become undirected. Edges beginning with "*d*" are always directed, and edges beginning with "*u*" are always undirected.

3.1.1.3 Comments

You can also choose to put comments in your graph input file. Comments are designated by the percent "*%*" sign. Anything after a "*%*" until the end of the line will be ignored (unless the "*%*" is part of a quoted label).

3.1.1.4 Example

As an example, if you were trying to represent that a *cat* is an *animal*, the graph might look like the following:

```
% Cat
v 1 cat
v 2 animal
d 1 2 is-a
```

However, if the edge were directed the other way (eg. d 2 1 is-a), that would imply that the animal is a cat, which is not necessarily true. It should be noted that SUBDUE would not complain if you made that relationship, but the results would probably not be what you desired.

3.1.2 Samples

The SUBDUE kit comes with various sample graph input files, some of which include both the textual representation as well as a simple pictorial view of the data.

3.2 Output

The output from executing SUBDUE, which will be discussed in more detail in the following section, consists of textual information that is essentially represented in the same format as the input. In addition to the patterns that are discovered, the output includes options, parameters and other information about the run. By default, the output is displayed to the user's screen, or to wherever the user directed the output (for example, with the Unix ">" command). However, the user can choose to also send just the discovered patterns to another file by using the *"-out <file>"* option. The format of the discovered pattern looks very much like a SUBDUE graph input file, and can be used in some of the various utilities that come in the SUBDUE kit (like *test* and *subs2dot*). These utilities will be discussed in a subsequent section.

3.3 Example

To illustrate what an input file might look like, let's take the example of a domain consisting of geometric shapes – i.e. squares, triangles, rectangles and circles. The goal is to find the most common patterns, or substructures, among these shapes.

First, the input file for SUBDUE contains the objects and their relationships to each other. Since this is an "unsupervised" example, the graph does not contain any positive (XP) or negative (XN) indicators. SUBDUE will assume that the graph is all positive.

Figure 1 illustrates the high-level view for this example. **Figure 2** shows the SUBDUE input file for this example. **Figure 3** shows the graphical representation for this example.

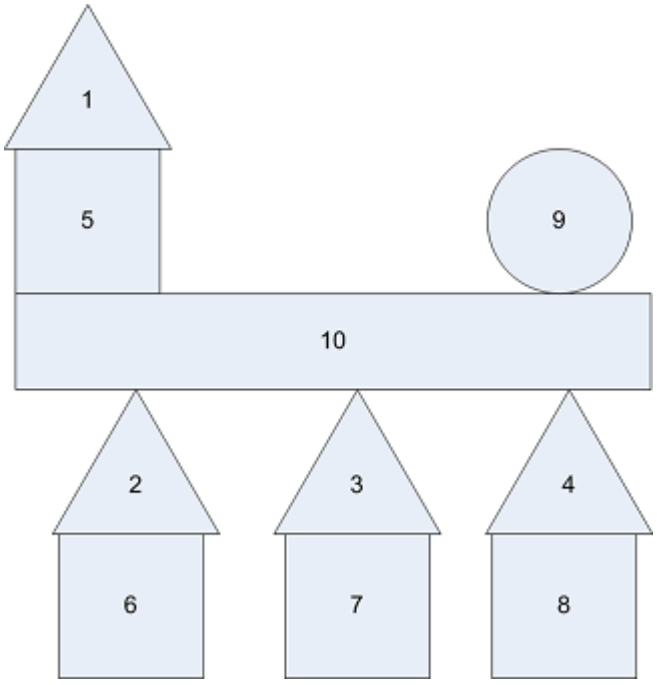

Figure 1: High-level view for shapes example

```
v  1 object
v  2 object
v  3 object
v  4 object
v  5 object
v  6 object
v  7 object
v  8 object
v  9 object
v 10 object
v 11 triangle
v 12 triangle
v 13 triangle
v 14 triangle
v 15 square
v 16 square
v 17 square
v 18 square
v 19 circle
v 20 rectangle

e  1 11 shape
e  2 12 shape
e  3 13 shape
e  4 14 shape
e  5 15 shape
e  6 16 shape
e  7 17 shape
e  8 18 shape
e  9 19 shape
e 10 20 shape
e  1  5 on
e  2  6 on
e  3  7 on
e  4  8 on
e  5 10 on
e  9 10 on
e 10  2 on
e 10  3 on
e 10  4 on
```

Figure 2: SUBDUE input graph format for shapes example

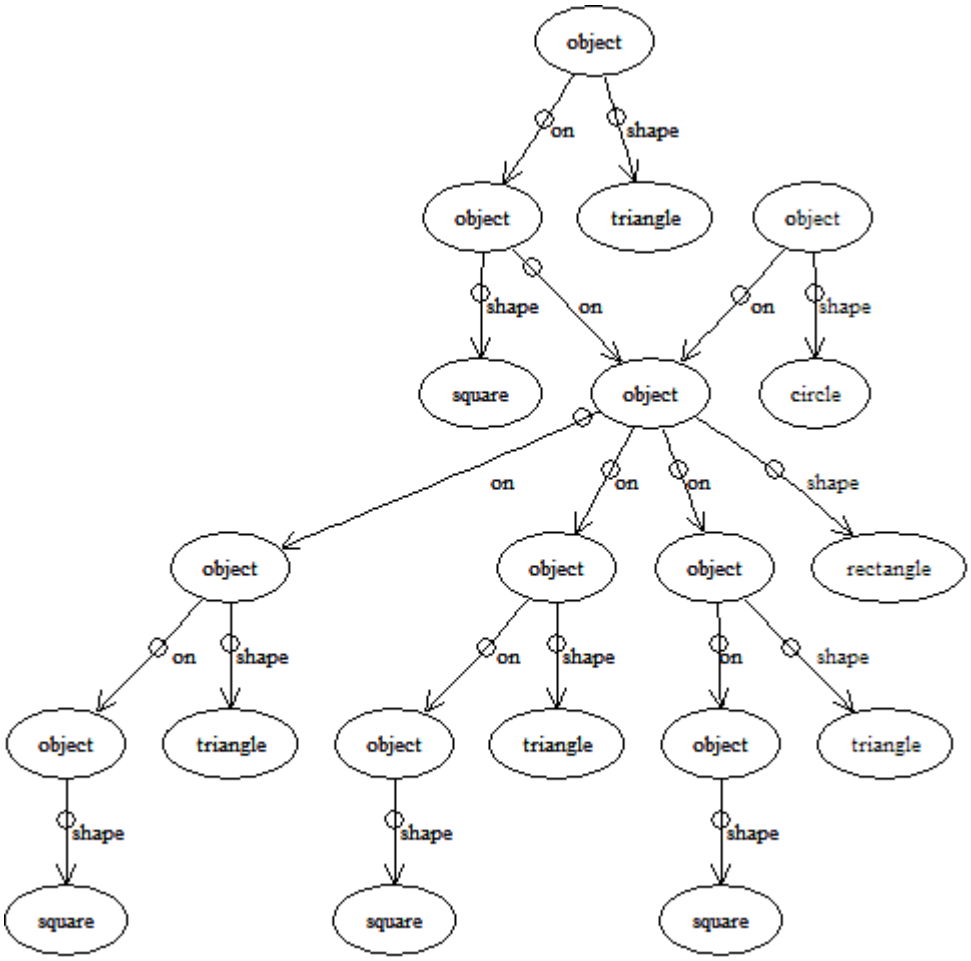

Figure 3: Graphical representation for shapes example

This graph example is also included in the SUBDUE kit in the file *sample.g*.

4. Executing

The purpose of the SUBDUE application is to find "interesting and repetitive subgraphs" from a specified input graph. The ability to discover these sub-graphs, or substructures, is controlled by various user-specified parameters, which control the methods that are used, as well as the length and size of the discovered items (among other things).

4.1 Command

SUBDUE has what is known as a "command-line" interface. (While there are efforts to create a Graphical User Interface (GUI) for SUBDUE, this manual deals with just the command-line capabilities.)

In order to run SUBDUE, you must be logged on to the Unix machine where the application was downloaded and installed. From the Unix prompt, the command would be as follows:

```
subdue <options> <graph input file>
```

There are several points which should be noted here:

subdue is the name of the executable. The above example assumes that you are running the application from the same directory where the executable resides (which is probably in ./*bin*/). If the desire is to run the application from another directory, *subdue* will have to be "pathed".

<*options*> will be discussed in the next section

<*graph input file*> is the name (and path) of the graph input file (ex. *sample.g*)

4.1.1 Options

Because of the nature of graphs, and the varying ways that graphs can be dissected and analyzed, there are several command-line options available to be used. Each of these options can result in different results when used together or by themselves.

This document will not go into graph theory, or even some of the more common algorithmic functions used in computers, and will leave that up to you to investigate. Each of the options will be explained, but it is assumed that there is some knowledge of the subject being discussed.

4.1.1.1 -beam <#>

This parameter specifies the beam width of SUBDUE's search. Only the best beam substructures (or all the substructures with the best beam values) are kept on the frontier of the search. The exact meaning of the beam width is determined by the *-valuebased* option described below. The default value for this setting is 4.

4.1.1.2 -compress

When the –compress option is specified, SUBDUE writes the compressed data to a file. For typical SUBDUE processing, the file <graph>.g.cmp is created (ex. sample.g.cmp). The input graph file is compressed using the top-valued substructure and the result is written to this file. When the the MDL and SIZE evaluation methods are specified (see section 4.1.1.3), the positive examples are combined into one graph, while the negative examples are combined into another graph. However, when the SETCOVER evaluation method is chosen, all of the negative examples are written to the file (preceded by "XN" for each example), and the positives not covered by the discovered substructure are written to the file (preceded by "XP" for each example). Vertices for each example are numbered starting at 1.

For incremental SUBDUE processing (see section 4.1.1.4), each of the substructures from the <graph>_<#>.g files (i.e. each increment), is compressed using the locally best substructure. The resulting compressed graph is written to <graph>-com_#.g (and <graph>-com_#.g, if appropriate).

4.1.1.3 -eval <#>

SUBDUE has three methods available for evaluating candidate substructures:

4.1.1.3.1 Minimum Description Length (MDL) - 1

The value of a substructure S in graph G is:

$$value(S,G) = \frac{DL(G)}{\left(DL(S) + DL(G \mid S)\right)}$$

where DL is the description length in bits, and (G|S) is G compressed with S. If a negative graph Gn is present, then:

$$value(S,GpGn) = \frac{\left[DL(Gp) + DL(Gn)\right]}{\left[DL(S) + DL(Gp \mid S) + DL(Gn) - DL(Gn \mid S)\right]}$$

MDL is the default evaluation method.

4.1.1.3.2 Size - 2

The value of a substructure S in graph G is:

$$value(S,G) = \frac{size(G)}{\left(size(S) + size(G \mid S)\right)}$$

where

$$size(G) = \left(\#vertices(G) + \#edges(G)\right)$$

and (G|S) is G compressed with S. If a negative graph Gn is present, then

$$value(S, Gp, Gn) = \frac{\left[size(Gp) + size(Gn)\right]}{\left[size(S) + size(Gp \mid S) + size(Gn) - size(Gn \mid S)\right]}$$

The size measure is faster to compute than the MDL measure, but less consistent.

4.1.1.3.3 Set Cover - 3

The value of a substructure S is computed as the number of positive examples containing S, plus the number of negative examples not containing S, all divided by the total number of examples. If this evaluation method is chosen, then the compression done at the end of each iteration is replaced by just removing all positive examples containing S.

4.1.1.4 -inc

The "-inc" option specifies that data should be handled incrementally. For this option, instead of processing only one graph file, multiple graph files (which can be thought of as one graph file broken up into several graph files) are handled. For example, if the input file "graph" is specified (i.e. the name of the graph file WITHOUT the .g suffix), then SUBDUE looks for incremental graph input in files named graph_1.g, graph_2.g, etc. The vertices in each incremental file are numbered starting at the number of the last vertex in the previous file plus one, and edges can connect to vertices from earlier increments. (There can NOT be edges with vertices that have not been encountered yet.) Processing stops when the file for the next increment number does not exist.

When specifying the evaluation method, the EVAL_MDL option is not allowed. If the user specifies EVAL_MDL, the function is changed to EVAL_SIZE and the user is alerted. Because graphs are processed incrementally, as discoveries are made in the local iteration, values are updated for old discoveries, and instances are found on increment boundaries. Thus, the drawback is that new substructures will not be discovered if they exist only on increment boundaries.

If the –compress option is used (see section 4.1.1.2) in conjuction with the –inc option, when processing files graph_#.g, at the end of each increment, the increment data is compressed using the locally best substructure. The resulting compressed graph is written to graph_#.c-pos (and graph-#.c-neg, if appropriate).

Due to the complexity of this process, the following options are NOT supported with the incremental option:

- Multiple iterations (see section 4.1.1.5) of incremental SUBDUE. As a work-around, you can run SUBDUE again using the compressed data stored in these files. In order to do this, you must look at the output file to get the definition for the corresponding substructures (labeled SUB_#). These vertex labels will need to be replaced by new unique identifiers before running SUBDUE again on the compressed data files.
- MPI (see section 4.2) version of SUBDUE.

- *subs2dot* tool (see section 4.5.7). As a work-around, you can write compressed graphs to a file and use the *graph2dot* tool.
- *cvtest* tool (see section 4.5.1). This is not very appropriate anyway, since cross validation assumes all examples are available.
- If the –compress option is also chosen, then predefined substructures cannot be specified (see section 4.1.1.14). This is because of some of the issues with compression across increments.

4.1.1.5 -iterations <#>

The number of iterations made over the input graph in which the best substructure from the previous iteration is used to compress the graph for use in the next iteration. The default value for this setting is 1, which implies only one pass, no compression. A value of 0 causes SUBDUE to iterate until no compression is possible, which produces a hierarchical, conceptual clustering of the input graphs. If SUBDUE is using the set-cover evaluation method (see the *-eval* option above), then iterations stop when no more positive examples can be removed.

4.1.1.6 -limit <#>

The number of different substructures to consider in each iteration. The default value is computed based on the input graph as #Edges / 2.

4.1.1.7 -maxsize <#>

This argument specifies the maximum number of vertices that can be in a reported substructure. Larger substructures are pruned from the search space. The default value for this setting is the number of vertices in the input graph.

4.1.1.8 -minsize <#>

This argument specifies the minimum number of vertices that must be in a substructure before it is reported. The default value for this setting is 1.

4.1.1.9 -nsubs <#>

This argument specifies the maximum length of the list of best substructures found during the discovery. The default value for this setting is 3.

4.1.1.10 -out <outfile>

If given, this option writes machine-readable output to the given file name. The file will contain the best substructure found at each iteration, each prefaced by the *SUB_TOKEN* string specified in *subdue.h* (usually just "S"). If this option is not specified, the output is written to the screen (unless the output has been re-directed with the Unix “>” command).

4.1.1.11 -output

This argument controls the amount of SUBDUE's screen output. Valid values are:

(1) Print best substructure found at each iteration.

(2) Print *-nsubs* best substructures at each iteration. (This is the default value.)

(3) Same as 2, plus prints the instances of the best substructures.

(4) Same as 3, plus prints substructure countdown and the best substructure found so far.

(5) Same as 4, plus prints each substructure considered.

4.1.1.12 -overlap

SUBDUE normally will not allow overlap among the instances of a substructure. Specifying this argument will allow overlap. During graph compression an *OVERLAP_<iteration>* edge is added between each pair of overlapping instances, and external edges to shared vertices are duplicated to all instances sharing the vertex. Allowing overlap slows SUBDUE considerably.

4.1.1.13 -prune

This option tells SUBDUE to prune the search space by discarding substructures whose value is less than that of their parent's substructure. Since the evaluation heuristics are not monotonic, pruning may cause SUBDUE to miss some good substructures, however, it will improve the running time. The default is no pruning.

4.1.1.14 -ps <psfile>

This option allows the input of a file containing predefined substructures. These substructures are used to compress the input graph with the idea that it would then be easier to find the identical structures. The order of the substructures in the file is important, because the substructures are tried in order and compression based on an earlier substructure may remove instances of later substructures. Also, the matching process follows the constraints of the *-overlap* and *-threshold* options. See the section on *Predefined Substructure* for a description of the format of this file. Note that this is essentially subgraph isomorphism and therefore NP-Complete, i.e., exponential running time in the size of the input graph.

4.1.1.15 -recursion

This option allows SUBDUE to consider recursive graph grammar rules as substructures along with the normal subgraph substructures. All are evaluated using the same metrics and compete for being the best substructure. A recursive substructure looks like a normal substructure except for a single "re <label>" edge indicating that the instances of the substructure are connected by an edge with the given label. The default is no recursive substructures.

4.1.1.16 -threshold <#>

The fraction of the size (vertices+edges) of an instance by which the instance can be different (according to the graph transformation costs defined in *subdue.h*) from the substructure definition. I.e., the graphs match if *matchcost(sub,inst) <= size(inst)*threshold*. The default setting is 0.0, which implies graphs must match exactly.

4.1.1.17 -undirected

SUBDUE assumes that edges in the input graph file defined using `e' are directed edges. Specifying this argument makes these edges undirected. Note that graph file edges defined with `u' are always undirected, and edges defined with `d' are always directed.

4.1.1.18 -valuebased

Normally, SUBDUE's beam width implies that only the beam best substructures are kept on the frontier of the search. If the *-valuebased* option is given, then the beam width is interpreted as keeping all the substructures with the top beam values on the frontier of the search.

4.2 MPI Version

Before going into some examples, there is one other way in which SUBDUE can be run, and that is by using a version that uses a Message Passing Interface (MPI) called *mpi_subdue.*

For those readers not familiar with how MPI works, check out http://ww.mpi-forum.org/ , or http://www.mcs.anl.gov/mpi/, where you can also download MPI software.

For purposes of demonstrating how the MPI version of SUBDUE works, the following sections use the open-source MPICH2 software for building and running *mpi_subdue*.

4.2.1 build

We will not go into the steps for installing the MPICH2 software on your system. The README file that comes with the MPICH2 tar kit is pretty self-explanatory, and if you follow the steps verbatim, you should be able to install the software properly.

In order to build the *mpi_subdue* executable, follow the same steps as outlined in section 2.3, except do a 'make mpi_subdue' instead. If you have set up your MPICH2 environment correctly, you should not receive any warnings or errors during the SUBDUE build.

4.2.2 run

Before you can start running *mpi_subdue*, you must initiate the message passing daemons. Depending upon how many SUBDUE processes you want to have running will depend upon how many daemons must be started. In order to start a daemon, you execute the following

```
mpd &
```

from the common prompt on whatever machine you wish the daemon to reside. (Subsequent *mpd* daemons from the same machine must be executed with the –n option.) You can verify the daemons that are running and their location by executing a

```
mpdtrace -l
```

from one of the machines in the "ring".

mpi_subdue expects input files in the regular SUBDUE format with an added .part*n* suffix, where *n* is the graph partition number corresponding to the child process number *n* that will process partition *n*.

mpi_subdue also requires an additional master process, so when invoking *mpi_subdue* on *n* graph partitions, you will need to start up *n+1* MPI processes.

Once all of the necessary daemons have been started, *mpi_subdue* can be initiated similar to the following:

```
mpiexec –n 5 bin/mpi_subdue <options> sample.g
```

The above command will start up one master and four child processes (‘-n 5’), which expect four graph partition files: *sample.g.part1* through *sample.g.part4*. You can also specify any of the SUBDUE options described above.

mpi_subdue works by processing each partition in parallel and then evaluating each partition's best substructure on the other partitions. *mpi_subdue* then returns the globally best substructures.

For more information about the MPI commands, refer to your MPI documentation.

4.3 Example Continued

Continuing with our example graph from the previous chapter, it is now time to run SUBDUE:

```
bin/subdue –out sub.g graphs/sample.g
```

Figure 4 shows a high-level visual depiction of the best substructure discovered by SUBDUE. **Figure 5** shows the textual output of SUBDUE for this same run. **Figure 6** shows the substructure pattern that was discovered, written to the file specified by the “*-out*” option. **Figure 7** shows the graphical representation of the substructure pattern.

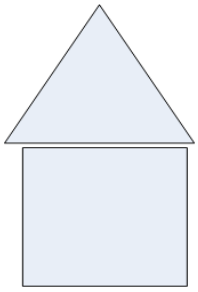

Figure 4: High-level visual depiction of best substructure discovered by SUBDUE on the sample graph.

```
Parameters:
  Input file..................... graphs/sample.g
  Predefined substructure file... none
  Output file.................... none
  Beam width..................... 4
  Evaluation method.............. MDL
  'e' edges directed............. true
  Iterations..................... 1
  Limit.......................... 9
  Minimum size of substructures.. 1
  Maximum size of substructures.. 20
  Number of best substructures... 3
  Output level................... 2
  Allow overlapping instances.... false
  Prune.......................... false
  Threshold...................... 0.000000
  Value-based queue.............. false
  Recursion...................... false
  Relations...................... false
  Variables...................... false

Read 1 positive graphs

1 positive graphs: 20 vertices, 19 edges, 252 bits
7 unique labels

3 initial substructures

Best 3 substructures:

(1) Substructure: value = 1.86819, pos instances = 4, neg instances = 0
  Graph(4v,3e):
    v 1 object
    v 2 object
    v 3 triangle
    v 4 square
    d 1 3 shape
    d 2 4 shape
    d 1 2 on

(2) Substructure: value = 1.37785, pos instances = 4, neg instances = 0
  Graph(3v,2e):
    v 1 object
    v 2 object
    v 3 square
    d 2 3 shape
    d 1 2 on

(3) Substructure: value = 1.37219, pos instances = 4, neg instances = 0
  Graph(3v,2e):
    v 1 object
    v 2 object
    v 3 triangle
    d 1 3 shape
    d 1 2 on
Subdue done (elapsed CPU time =    0.01 seconds).
```

Figure 5: SUBDUE output after running on the example graph file.

```
S
v 1 object
v 2 object
v 3 triangle
v 4 square
d 1 3 shape
d 2 4 shape
d 1 2 on
```

Figure 6: Discovered substructure after running on the example graph file.

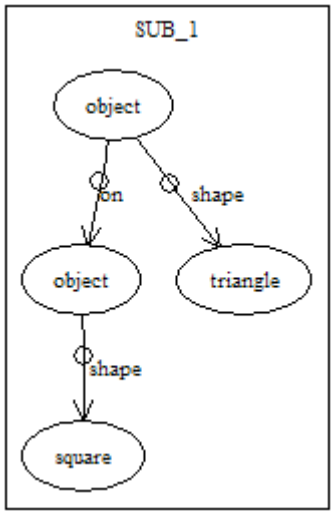

Figure 7: Graphical representation of the substructure discovered in the example.

The first part of SUBDUE's output indicates the parameter settings for this run. These parameters were mentioned previously, but we discuss some of them in more detail here. All of the parameters are set to their default values. SUBDUE's default *evaluation* method is based on the minimum description length (MDL) principle, which essentially says that the best pattern (or substructure) is the one that best trades off the size of the pattern and the size of the input graph after compressing away all the instances of the pattern. If there are negative graphs in the input, then the best pattern is the one that best compresses the positive graphs, but least compresses the negative graphs. This approach tends to prefer patterns that compress well, even though they may not discriminate well. Thus, for this example, we have chosen the default evaluation method *(MDL).*

The *limit* parameter controls the extent of SUBDUE's search by limiting the number of different substructures SUBDUE considers for expansion, i.e., it is an upper bound on the portion of the search space considered by SUBDUE. The *limit* defaults to half the number of edges in the positive graphs. This default value tends to be higher than necessary, as SUBDUE typically finds the best substructure early on. After experience with running SUBDUE in a domain, the *limit* parameter can be decreased to a value closer to when SUBDUE actually finds the best substructure, which can be determined by setting the *output level* to 5 so that SUBDUE outputs whenever it finds a new best substructure.

The *evaluation method* and *limit* parameters allow significant control over the efficiency and effectiveness of SUBDUE. Other parameters (*beam*, *iterations*, *prune*, *valuebased*) exert additional control over the amount of search, while still others (*overlap*, *threshold*, *recursion*) allow the introduction of additional capabilities (see previous section on ***Options***, and subsequent examples).

In this simple example, we see that the input graph has 1 (positive) graph. The total number of vertices and edges are given, along with the description length in bits of this graph, according to the MDL encoding used by SUBDUE. Next, SUBDUE indicates the number of unique labels found in the graph with the added constraint that the label must appear at least twice in the graphs. Seven such labels exist in the graph.

At this point, SUBDUE begins its search for the substructure maximizing the chosen evaluation method. By default, SUBDUE returns the three best substructures, ordered from best to worse (of the top 3).

Finally, we see that SUBDUE spent less than a second finding this substructure. As discussed above, the running time of SUBDUE depends on a number of parameters. For large graphs, SUBDUE's running time can be quite long, sometimes on the order of days. However, these long running times can be addressed by tweaking parameters, utilizing SUBDUE's parallel processing features (see previous section on ***Options***, and subsequent examples), or possibly redesigning the graph representation to remove information not relevant to the learning task.

4.4 More Examples

To cover some of the features of SUBDUE, the following sections present various different examples of how the application can be used.

4.4.1 Supervised

With *supervised* training, both positive and negative examples are provided as input to SUBDUE. For this example, the domain of the data is possible terrorist activity. The goal is to find patterns in the communications activity that can distinguish threat groups from non-threat groups.

4.4.1.1 Input

First, the input file for SUBDUE contains threat groups as positive example graphs (designated by "*XP*") and non-threat groups as negative example graphs (designated by "*XN*"). **Figure 8** illustrates the general structure used for representing this data as a graph, and **Figure 9** shows the SUBDUE input file for one of the positive examples.

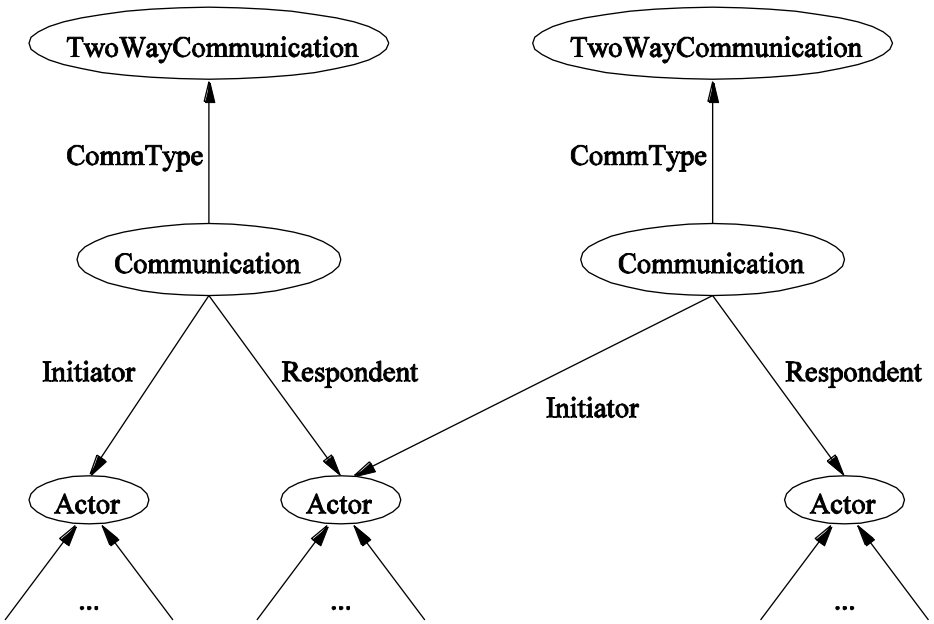

Figure 8: Graphical structure used to represent threat and non-threat communication graphs.

```
% Group 2 - Threat Group
v 1 Communication
v 2 TwoWayCommunication
d 1 2 CommType
v 3 Actor
v 4 Actor
d 1 3 Initiator
d 1 4 Respondent
v 5 Communication
v 6 TwoWayCommunication
d 5 6 CommType
d 5 3 Initiator
d 5 4 Respondent
v 7 Communication
v 8 TwoWayCommunication
d 7 8 CommType
v 9 Actor
d 7 9 Initiator
d 7 4 Respondent
v 10 Communication
v 11 TwoWayCommunication
d 10 11 CommType
d 10 9 Initiator
d 10 3 Respondent
v 12 Communication
v 13 TwoWayCommunication
d 12 13 CommType
d 12 9 Initiator
d 12 4 Respondent
v 14 Communication
v 15 TwoWayCommunication
d 14 15 CommType
d 14 9 Initiator
d 14 3 Respondent
v 16 Communication
v 17 TwoWayCommunication
d 16 17 CommType
v 18 Actor
d 16 9 Initiator
d 16 18 Respondent
v 19 Communication
v 20 TwoWayCommunication
d 19 20 CommType
v 21 Actor
v 22 Actor
d 19 21 Initiator
d 19 22 Respondent
v 23 Communication
v 24 TwoWayCommunication
d 23 24 CommType
d 23 22 Initiator
d 23 21 Respondent
```

Figure 9: SUBDUE input graph file for threat communication group.

This graph example is also included in the SUBDUE kit in the file *groups.g*. The file contains 3 threat groups and 7 non-threat groups.

4.4.1.2 Execution

Now that the data is in the proper format, we can run SUBDUE using the following command (assuming the graph file is located where you are running the application):

```
bin/subdue -eval 3 groups.g
```

Figure 10 shows a graphical depiction of the best substructure discovered by SUBDUE. **Figure 11** shows the textual output of SUBDUE for this same run. (The full output can be found in the file `groups-eval3-output.txt` that comes with the kit.)

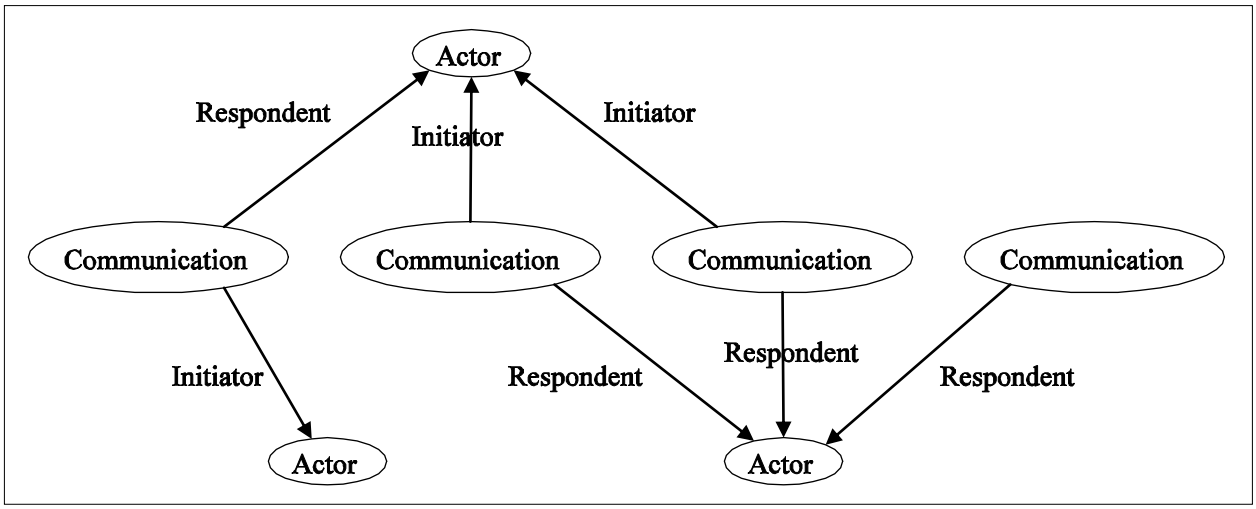

Figure 10: Graphical depiction of best substructure discovered by SUBDUE on the groups graph.

```
...
Parameters:
  Input file..................... groups.g
  Predefined substructure file... none
  Output file.................... none
  Beam width..................... 4
  Evaluation method.............. setcover
  'e' edges directed............. true
  Iterations..................... 1
  Limit.......................... 70
  Minimum size of substructures.. 1
  Maximum size of substructures.. 118
  Number of best substructures... 3
  Output level................... 2
  Allow overlapping instances.... false
  Prune.......................... false
  Threshold...................... 0.000000
  Value-based queue.............. false
  Recursion...................... false
  Relations...................... false
  Variables...................... false

Read 3 positive graphs
Read 7 negative graphs

3 positive graphs: 118 vertices, 141 edges, 1960 bits
7 negative graphs: 1406 vertices, 1683 edges, 29327 bits
7 unique labels

3 initial substructures

Best 3 substructures:

(1) Substructure: value = 1, pos instances = 4, neg instances = 0
  Graph(7v,7e):
    v 1 Communication
    v 2 Actor
    v 3 Actor
    v 4 Communication
    v 5 Actor
    v 6 Communication
    v 7 Communication
    d 1 2 Initiator
    d 1 3 Respondent
    d 4 2 Initiator
    d 4 3 Respondent
    d 6 3 Respondent
    d 7 5 Initiator
    d 7 2 Respondent
...
Subdue done (elapsed CPU time =    0.78 seconds).
```

Figure 11: SUBDUE output after running on the groups graph file using the *setcover* evaluation method 3. The 2nd and 3rd best substructures are deleted.

In this example, an *eval* value of *3* was chosen, which means we want SUBDUE to use the *setcover* evaluation method. This method looks for patterns that discriminate well, which is the main objective in a threat group task, without regard for how well the patterns compress.

As you will notice, the input graph has 3 positive (threat group) graphs and 7 negative (non-threat group) graphs. Again, the total number of vertices and edges in each of these two sets is given along with the description length in bits of these graphs, according to the MDL encoding used by SUBDUE.

While SUBDUE returns the three best substructures, for this example, we only show the best in **Figure 8**. The value of this substructure is 1, meaning that it perfectly discriminates between the positive and negative group graphs. In other words this substructure shows up only in positive graphs and not in negative graphs. In fact, as we see from the output, one of the positive graphs has two instances of the substructure, since we have four instances in three examples.

4.4.2 Overlap

By default, SUBDUE does not allow for instances of a substructure to contain overlapping edges. However, you can choose to override this functionality by specifying the "*-overlap*" option.

The following simple example of repetitive vertices and edges shows the advantage of using this option.

4.4.2.1 Input

Figure 12 illustrates the structure used for representing this data as a graph, and **Figure 13** shows the SUBDUE input file for this example.

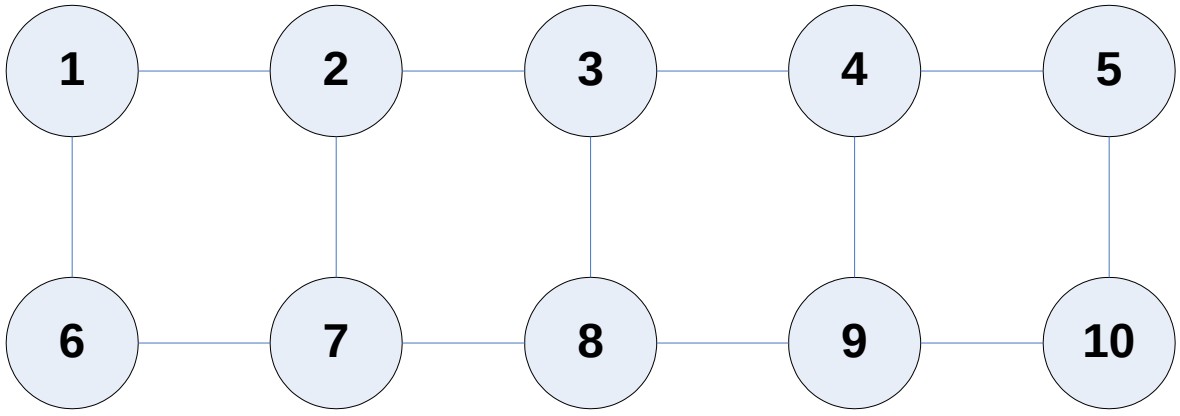

Figure 12: Pictorial representation of repetitive vertices and edges example

```
v 1 a
v 2 a
v 3 a
v 4 a
v 5 a
v 6 a
v 7 a
v 8 a
v 9 a
v 10 a
u 1 2 b
u 2 3 b
u 3 4 b
u 4 5 b
u 1 6 b
u 2 7 b
u 3 8 b
u 4 9 b
u 5 10 b
u 6 7 b
u 7 8 b
u 8 9 b
u 9 10 b
```

Figure 13: SUBDUE input format for overlap example

4.4.2.2 Execution

In order to better understand how this works, let's make two runs: one without the *-overlap* option and one with the option. So, if a run were made without the *–overlap* option, it would look like the following:

```
bin/subdue overlap.g
```

Figure 14 shows a graphical depiction of the best substructure discovered by SUBDUE without the overlap option. **Figure 15** shows the textual output of SUBDUE (for this same substructure).

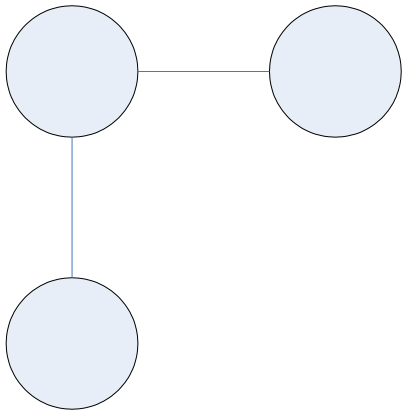

Figure 14: Graphical depiction of best substructure discovered by SUBDUE on the example WITHOUT the –overlap option

```
...
(1) Substructure: value = 1.16177, pos instances = 3, neg instances = 0
  Graph(3v,2e):
    v 1 a
    v 2 a
    v 3 a
    u 1 3 b
    u 2 3 b
...
```

Figure 15: SUBDUE output after running on the example WITHOUT the –overlap option

Now try the same run WITH the *–overlap* option:

```
bin/subdue –overlap overlap.g
```

Figure 16 shows a graphical depiction of the best substructure discovered by SUBDUE WITH the overlap option. **Figure 17** shows the textual output of the best substructure found.

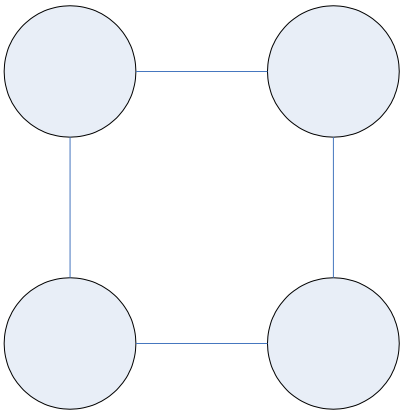

Figure 16: Graphical depiction of best substructure discovered by SUBDUE on the example WITH the –overlap option

```
...
(1) Substructure: value = 1.35206, pos instances = 4, neg instances = 0
  Graph(4v,4e):
    v 1 a
    v 2 a
    v 3 a
    v 4 a
    u 1 2 b
    u 1 3 b
    u 2 4 b
    u 3 4 b
...
```

Figure 17: SUBDUE output after running on the example WITH the –overlap option

Notice how the example with the *–overlap* option was able to find a connected substructure, which occurred 4 times (there were 4 instances), as opposed to the non-overlap option that only found a substructure with two edges (of which there were 3 instances). However, while there may have been 4 instances of the substructure found with the *–overlap* option, there was an overlap between the instances in that each of the instances shared an edge with another instance.

4.4.3 Predefined Substructure

The following example shows the results of using predefined substructures. As discussed in the previous section, this option allows one to compress the input graph so that it will be easier to find the matching structures.

4.4.3.1 Input

For this example, we are going to use a chemical example, where carbons are connected by single bonds.

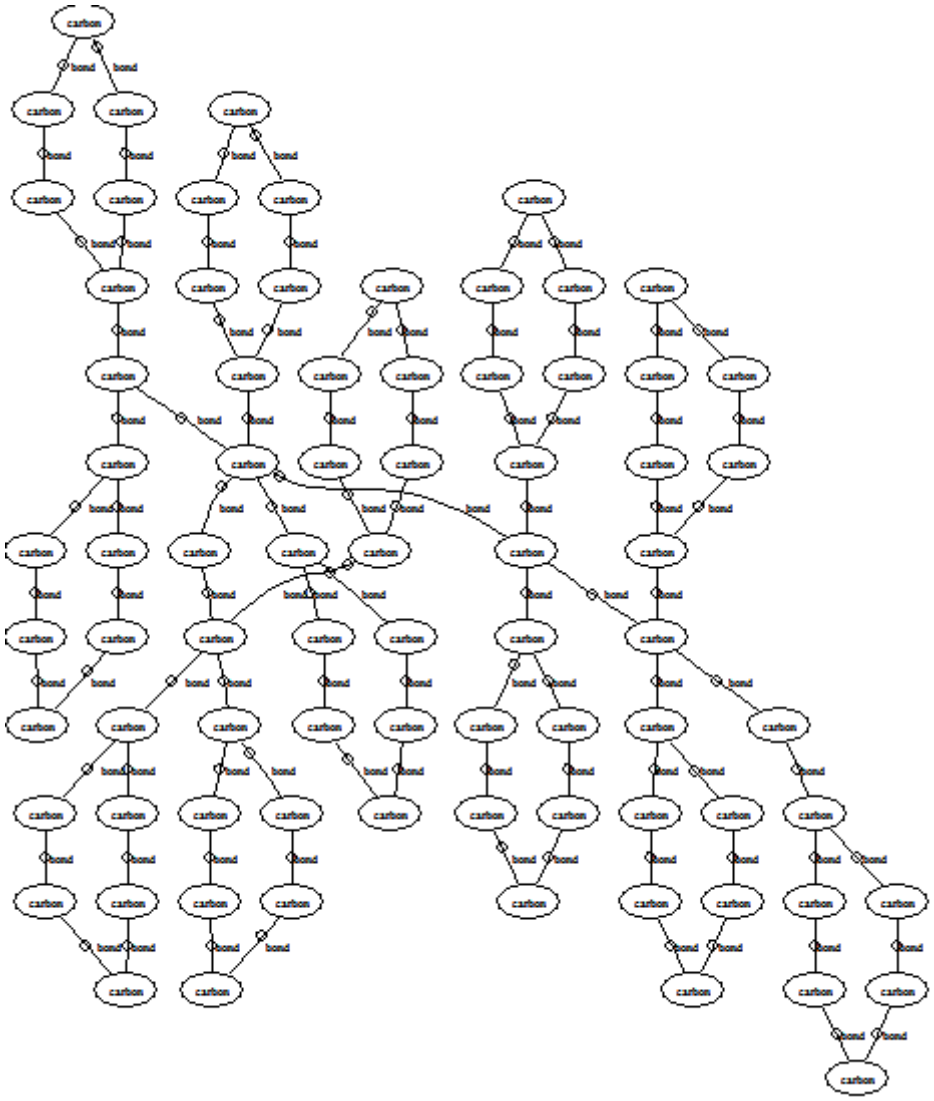

Figure 18: Pictorial representation of chemical example

Figure 18 illustrates the structure used for representing this data as a graph, and **Figure 19** shows part of the SUBDUE input file for this example.

```
v 1 carbon
v 2 carbon
v 3 carbon
v 4 carbon
v 5 carbon
v 6 carbon
v 7 carbon
v 8 carbon
v 9 carbon
v 10 carbon
v 11 carbon
v 12 carbon
v 13 carbon
...
u 1 2 bond
u 1 3 bond
u 2 4 bond
u 3 5 bond
u 4 6 bond
u 5 6 bond
u 6 7 bond
u 7 8 bond
u 8 9 bond
u 8 10 bond
u 9 11 bond
u 10 12 bond
u 11 13 bond
u 12 13 bond
...
```

Figure 19: Portion of SUBDUE input file for chemical example

If we were to run this graph through SUBDUE WITHOUT any predefined substructures, the result would look like the SUBDUE output shown in **Figure 20**.

```
...
Best 3 substructures:

(1) Substructure: value = 2.58538, pos instances = 10, neg instances = 0
  Graph(6v,6e):
    v 1 carbon
    v 2 carbon
    v 3 carbon
    v 4 carbon
    v 5 carbon
    v 6 carbon
    u 1 2 bond
    u 1 3 bond
    u 2 4 bond
    u 3 5 bond
    u 4 6 bond
    u 5 6 bond
...
```

Figure 20: Best substructure found for chemical example (SUBDUE output)

Figure 21 shows a pictorial representation of this best substructure.

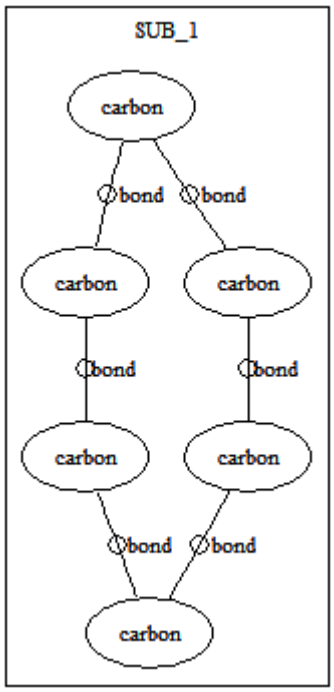

Figure 21: Best substructure found for chemical example (pictorial representation)

The result is that the best substructure found was a single cyclohexane ring, consisting of 6 carbons, each connected by single bonds.

As you can see from the initial picture, this graph is composed of many cyclohexane rings, some of which are connected by individual carbons. So, perhaps, a better substructure could be discovered if we were to provide the single cyclohexane ring as a pre-defined substructure.

A predefined substructure is very similar in format to a regular graph. A predefined substructure file can contain one or more substructures, each of which is prefaced with a "*PS*" indicator. Each indicator is then followed by its vertices and edges, where the format is identical to the one used in the graph input file.

```
PS
v 1 carbon
v 2 carbon
v 3 carbon
v 4 carbon
v 5 carbon
v 6 carbon
u 1 2 bond
u 1 3 bond
u 2 4 bond
u 3 5 bond
u 4 6 bond
u 5 6 bond
```

Figure 22: Pre-defined substructure for chemical example (SUBDUE format)

For the purposes of this example, we will name the pre-defined substructure file *carbon-ps.g*.

4.4.3.2 Execution

Now that we have a predefined substructure to go along with our input graph, we can run SUBDUE using the following command:

```
bin/subdue -ps carbon-ps.g graphs/carbon.g
```

Figure 23 shows a graphical depiction of the best substructure discovered by SUBDUE. **Figure 24** shows a portion of the textual output of SUBDUE for this same run.

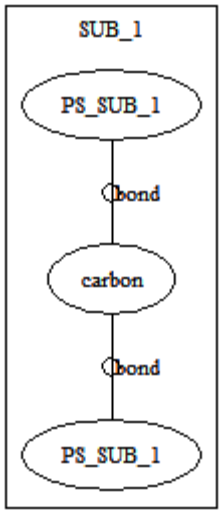

Figure 23: Graphical depiction of best substructure discovered by SUBDUE on the carbon example graph using a predefined substructure.

```
...

Best 3 substructures:

(1) Substructure: value = 1.58538, pos instances = 5, neg instances = 0
  Graph(3v,2e):
    v 1 PS_SUB_1
    v 2 PS_SUB_1
    v 3 carbon
    u 1 3 bond
    u 3 2 bond

...
```

Figure 24: SUBDUE output using the predefined substructure on the carbon example.

By giving SUBDUE the cyclohexane ring, it was able to find the substructure of two cyclohexane rings connected by a single carbon.

4.4.4 Recursion

The following example shows the results of looking for recursive substructures. Taken from Jonyer's paper entitled "***MDL-Based Context-Free Graph Grammar Induction and Applications***" (which can be found on the SUBDUE web-site), the following example shows how recursion was used to discover patterns.

4.4.4.1 Explanation

The concept of examining for recursive substructures is a little more complex than some of the features that have been explained so far, and requires a little more explanation.

A recursive production is constructed by checking all outgoing edges of each instance to see if they are connected to any other instance. One can see in **Figure 25** that the instance in the lower left is connected to the instance on its right, via vertex ‘y’ being connected to vertex ‘x’. The same can be said of the situation on the lower right side. Abstracting out these four instances, using the recursive option, results in the graph depicted in **Figure 26**.

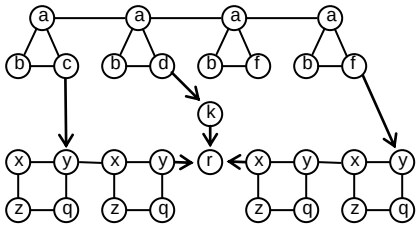

Figure 25: Pictorial representation of SUBDUE input graph.

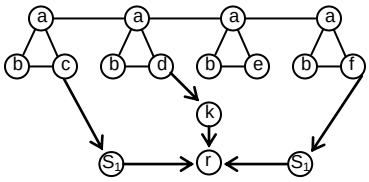

Figure 26: Input graph, parsed by the first production.

4.4.4.2 Input

Figure 27 shows a portion of the SUBDUE input file for the graph presented above.

```
v 1 a
v 2 b
v 3 c
v 4 a
v 5 b
v 6 d
…
v 13 k
v 14 x
v 15 y
v 16 z
v 17 q
v 18 x
v 19 y
v 20 z
v 21 q
v 22 r
…
u 1 2 t
u 2 3 t
u 1 3 t
…
u 1 4 next
u 4 7 next
u 7 10 next
…
u 14 15 s
u 14 16 s
u 15 17 s
u 16 17 s
…
```

Figure 27: (partial) SUBDUE input graph file for recursive example.

This graph example is also included in the SUBDUE kit in the file *sampleGL.g*.

4.4.4.3 Execution

In order to better understand how this works, let's make two runs: one without the *-recursive* option and one with the option. So, if a run were made without the *–recursion* option, it would look like the following:

```
bin/subdue sampleGL.g
```

Figure 28 shows a graphical depiction of the best substructure discovered by SUBDUE. **Figure 29** shows the textual output of SUBDUE (for this same substructure).

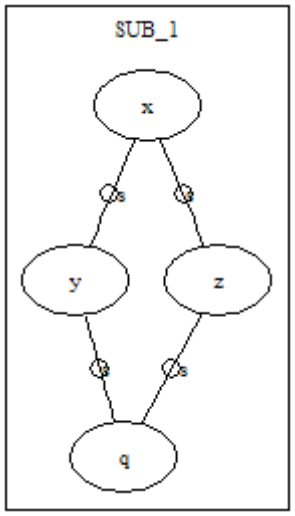

Figure 28: Pictorial representation of best substructure found.

```
...
(1) Substructure: value = 1.43479, pos instances = 4, neg instances = 0
  Graph(4v,4e):
    v 1 x
    v 2 y
    v 3 z
    v 4 q
    u 1 2 s
    u 1 3 s
    u 2 4 s
    u 3 4 s
...
```

Figure 29: (partial) SUBDUE output of best substructure found.

Now try the same run WITH the *–recursion* option:

```
bin/subdue –nsubs 6 –recursion sampleGL.g
```

While the initial best substructures found were not any difference, if we increased the number of best substructures past the default of 3, we can discover some new substructures. So, for this example, we have chosen to find the six best substructures (*–nsubs* 6).

Figure 30 shows a graphical depiction of the new substructure discovered by SUBDUE using the "-*recursion*" option. **Figure 31** shows the textual output of the best substructure found.

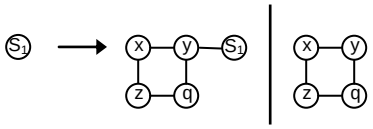

Figure 30: Pictorial representation of substructure found.

```
...
Best 6 substructures:
...
(6) Substructure: value = 1.17601, pos instances = 2, neg instances = 0
  Graph(4v,3e):
    v 1 x
    v 2 y
    v 3 z
    v 4 q
    u 1 2 s
    u 1 3 s
    u 3 4 s
    re next

...
```

Figure 31: (partial) SUBDUE output showing a recursive substructure.

What happens is that SUBDUE checks an instance of a substructure to see if it is connected to any of its other instances by an edge. If it is, and it is the best substructure at the end of the iteration, each such chain of subgraphs is abstracted away and replaced by a single vertex.

4.5 Other Tools

The following sections cover various tools that are supplied in the SUBDUE kit.

4.5.1 cvtest

The *cvtest* utility performs a cross-validation experiment on a set of examples using the SUBDUE system. The command line interfaces is as follows:

>cvtest [<subdue options>] [-nfolds <n>] <graphfile>

The *cvtest* program first divides the examples defined in <*graphfile*> into <*nfolds*>sets of examples using random selection. The program then runs SUBDUE to find the substructures for the training set of each fold. It then uses the *test* program (see below) to evaluate the substructures on the test set of the fold. The program reports the error rate for each fold and the mean error rate for all folds.

The *cvtest* program accepts all SUBDUE options (except *–out*), and the *–nfolds* <*n*> option MUST come after the SUBDUE options. The default *–nfolds* value is 1, which means the entire set of examples is used for both training and testing. Also, the *cvtest* program must be run from the same directory where the *subdue* executable resides.

4.5.2 gm

The *gm* tool is a standalone graph matcher. The command-line interface is as follows:

>*gm* <*first graph input file*> <*second graph input file*>

The *gm* utility (an inexact graph match program), computes the cost of transforming the larger of the input graphs into the smaller according to the transformation costs defined in SUBDUE. This program returns the cost and the mapping of vertices in the larger graph to vertices in the smaller graph.

4.5.3 gprune

The *gprune* tool removes vertices and edges with specific labels. The command-line interface is as follows:

>*gprune* <*label*> <*input graph file*> <*output graph file*>

The *gprune* utility removes all vertices and edges whose label is equal to <*label*> in <*input graph file*> and outputs the results to <*output graph file*>. When a vertex is removed, all of its edges are removed. When an edge is removed (either because it has the defined <*label*> or was removed as the result of a vertex removal), any vertices left without a connecting edge are also removed.

If <*input graph file*> contains multiple examples, then each is pruned separately.

4.5.4 graph2dot

The *graph2dot* utility converts a SUBDUE graph input file into the dot format as specified in the AT&T Labs GraphViz package.

The command-line interface is as follows:

>graph2dot <SUBDUE graph file> < dot file>

This tool writes <*SUBDUE graph file*> to < *dot file*> in the dot format. The dot file can then be displayed graphically using the GraphViz tools (see http://www.graphviz.org/).

For an example of what is produced by the GraphViz tools, refer to **Figure 18**, or any of the similar figures above.

4.5.5 mdl

The *mdl* program is a standalone utility for computing the Minimum Description Length (MDL) of a graph. The command-line interface is as follows:

>mdl [-dot <filename>] [-overlap] [-threshold <#>] <g1> <g2>

The *mdl* utility computes the description length of <*g1*> and <*g2*>, as well as <*g2*> compressed with <*g1*>, using the following MDL compression measurement:

D1(g2)/(D1(g1) + D1(g2|g1))

This program also prints the size-based compression information.

If *–overlap* is given, the instances of <*g1*> may overlap in <*g2*>. If *–threshold* is given, then instances in <*g2*> may not be an exact match to <*g1*>, but the cost of transforming <*g1*> to the instance is less than the threshold fraction of the size of the larger graph. The default value for the threshold is 0.0 (i.e. an exact match). If the *–dot* option is used, then the compressed graph is written to <*filename*> in dot format. The dot file can then be displayed graphically using the GraphViz tools (see http://www.graphviz.org/).

4.5.6 sgiso

The *sgiso* program is a standalone utility for performing a sub-graph isomorphism. The command-line interface is as follows:

>sgiso [-dot <filename>] [-overlap] [-threshold <#>] <g1> <g2>

The *sgiso* utility find and prints all instances of <*g1*> in <*g2*>. If *–overlap* is given, then instances may overlap in <*g2*>. If *–threshold* is given, then instances may not be an exact match to <*g1*>, but the cost of transforming <*g1*> to the instance is less than the threshold fraction of the size of the larger graph. The default value for the threshold is 0.0 (i.e. an exact match). If the *–dot* option is used, then

the compressed graph is written to <*filename*> in dot format. The dot file can then be displayed graphically using the GraphViz tools (see http://www.graphviz.org/).

4.5.7 subs2dot

The *subs2dot* utility converts a SUBDUE output file (specified when the *–out* option is used) into the dot format as specified in the AT&T Labs GraphViz package.

The command-line interface is as follows:

>*subs2dot <SUBDUE output file> < dot file>*

This tool writes the substructures defined in <*SUBDUE output file*> to < *dot file*> in the dot format. The dot file can then be displayed graphically using the GraphViz tools (see http://www.graphviz.org/).

For an example of what is produced by the GraphViz tools, refer to **Figure 28**, or any of the similar figures above.

4.5.8 test

The *test* utility computes the following for a given set of substructures and a given set of example graphs:

FP – false positives

TP – true positives

FN – false negatives

TN – true negatives

The command-line interfaces is as follows:

>*test <subsfile> <graphfile>*

The *test* program reads in substructures from <*subsfile*> and then reads in example graphs from <*graphfile*> one at a time. If any of the substructures is a sub-graph of the example graph, that example is classified as positive. Otherwise, it is classified as negative. When the program completes, it reports the FP/FN/TP/TN and error statistics.

It should be noted that this utility assumes that all ‘e’edges are directed, and the match threshold is 0.0, despite the parameters that were run with SUBDUE.

5. Notes/Issues

The following sections represent various notes and issues.

5.1 Unix

SUBDUE was designed and developed to run on a Unix-based system. The application was tested on Linux, but should be compatible with any Unix system. SUBDUE was written in C, where every effort was made to use only standard ANSI C constructs and functions.

A. Appendix - Terminology

The following terminology was referenced in this document:

MDL – Minimum Description Length

MPI – Message Passing Interface

SUBDUE – SUBstructure Discovery Using Examples

TBD – To Be Determined