Technical Report

Pattern Recognition and Image Processing Group
Institute of Computer Aided Automation
Vienna University of Technology
Favoritenstr. 9/1832
A-1040 Vienna AUSTRIA
Phone: +43 (1) 58801-18351
Fax: +43 (1) 58801-18392
E-mail: {saib,yll,glz}@prip.tuwien.ac.at
URL: http://www.prip.tuwien.ac.at/

PRIP-TR-69 March 14, 2002

Dgc_tool version 1.0 Building Irregular Graph Pyramid Using Dual Graph Contraction [1]

Maamar Saib and Yll Haxhimusa and Roland Glantz

Abstract

In this technical report the new version of the software *Dgc_tool* is presented. This tool allows us to build up irregular graph pyramids by dual graph contraction. The graph pyramid consists of a stack of levels (pair of graphs), each of which has a primal level and its dual. Every successive level is a reduced version of the level below. Primal level and its dual represent a primal graph and its dual, respectively. The primal graph base level of the pyramid may represent a two dimensional image.

[1]This Work was supported by the Austrian Science Foundation under P14445-MAT and P14662-INF.

Contents

1 Introduction

Building irregular pyramids (for an overview see [KLB99]) by dual graph contraction (DGC) was explained theoretically in the technical report TR-35 [Kro94] and [Kro95] and can be used in line image analysis [BK99, KB98], description of image structure [GEK99], connected component analysis [KM95] etc. In this technical report we explain a new version of the software *dgc_tool*[1], which allows us to build irregular graph pyramids. The new version of *dgc_tool* was developed by Yll Haxhimusa , Saib Maamar and Roland Glantz in C++ using the version 3.8 of LEDA C++ class library [MN99]. We decided to use LEDA (Library of Efficient Data types and Algorithms) because of the versatile of data types like $GRAPHS$, *h_array*, *node*, *edge*, *list*, $edge\_array$, $node\_array$ etc.,iterators like $forall\_nodes$, $forall\_edges$, $forall\_out\_edges$ etc. and good graphical unit interface (GUI) to draw and test graphs .

The old version of *dgc_tool* was developed in C++ and LEDA class library [KBBS98] by Mark Burge, Roman Englert, Roland Glantz and Walter G. Kropatsch. In the old version, no pyramid was constructed. Dual graph contraction was always performed in the same dual pair of graphs. However, the old version allowed to contract run graphs as described in [BK99]. Moreover, it was possible to draw the primal graph with a mouse.

In the new version of *dgc_tool* the pyramid is stored in a data structure called hashing array ($h\_array < int, levelpair >$) [MN99], where each *levelpair* is an element and *int* is the index of *h_array*. The index denote the levels of the pyramid. Every element of the pyramid (*levelpair*, *levels* of the *levelpair*, *graph* of the *level*, *nodes*, *edges* etc.) can be accessed directly.

The new version of the *dgc_tool* contains five principle classes:

- *dgc_node*,
- *dgc_edge*,
- *level*,
- *levelpair*, and
- *pyramid*.

[1]ftp.prip.tuwien.ac.at/pub/dgc_tool

The plan of this technical report is as follows. In Section 2 we will outline the construction of the image graph pyramid using a simple example to find connected component. In Section 3 the usage of the *dgc_tool* will be described, and in Section 4 we will explain in detail the classes of the *dgc_tool*.

2 Building Irregular Image Graph Pyramid

Irregular image graph pyramid is simply a stack of attributed graphs of smaller size, meaning the number of nodes and edges decreases.

The idea of the construction of the irregual graph pyramid is as follows:

We start with a pair of base levels each of which contains an attributed graph. For example, at the base level an attributed graph could represent a 2D image (Figure 1(a)), where the pixel attributes could be stored in vertices and edges would represent the spatial relationship of pixels (Figure 1(b)). But it is also possible to store other numerical or/and symbolical information in vertices and edges. The other base level (the dual level) would contain the dual graph (Figure 1(c), in this figure it has no attributes).

In order to reduce the number of vertices and edges of the base level and construct new levels with smaller number or vertices and edges (reduced level), we need to know which vertices survive and which edges are to be contracted i.e. we need to know so called decimation parameters or contraction kernels (Figure 2(a)(d) the dashed frames). Decimation parameters are choosen using maximal independent set algorithms [Mee89, BK93, HGS$^+$02]. Note in the Figure 2(a)(d) that contraction kernels are rooted (black vertices) trees with maximum depth of one. After we find contraction kernels we do dual graph contraction [Kro95].

Dual graph contraction consists of two steps:

- **1st step -** We apply edge contraction to the primal graph *pg* and the corresponding edges in the dual graph *dpg* are deleted as in Figure 2(a)(b). Black arrow edges denote the directions of contractions and dashed edges in Figure 2(b) denote the edges to be deleted. Thus a new reduced level and a new dual level is created (see Figure 2(c)(d)).

- **2nd step -** Then we apply edge contraction to the dual graph *dg* and the corresponding edges in the primal graph *pg* are deleted, see Figure 2(c)(d). Black arrow edges in Figure 2(d) denote the directions

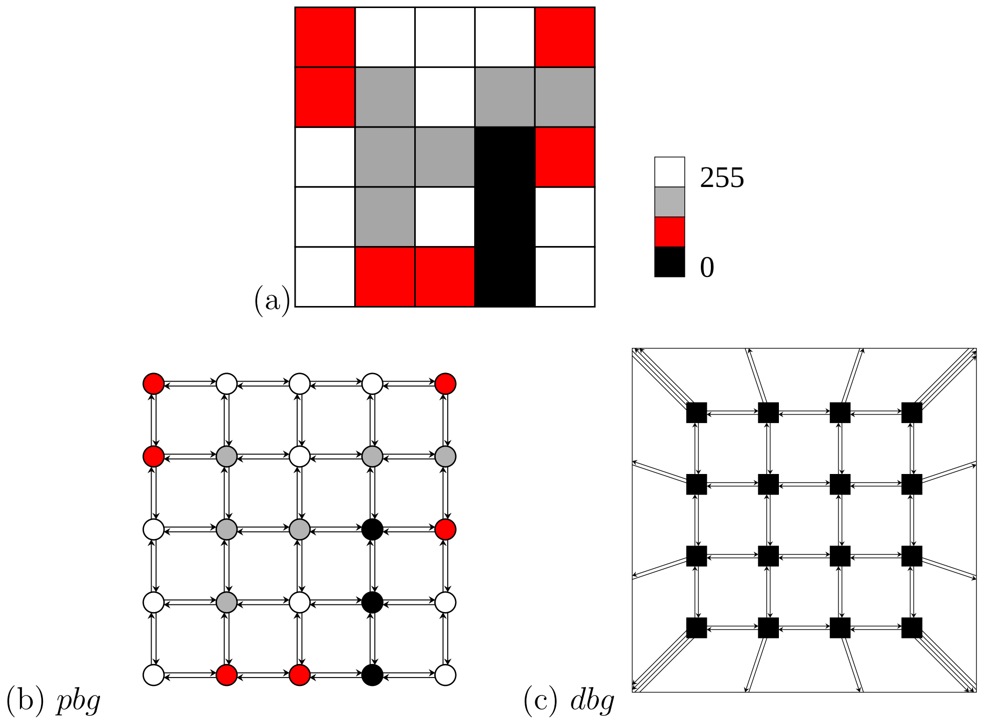


Figure 1: (a) Simple gray value image with 11 regions. (b) Primal base graph *pbg*, and (c) Dual base graph *dbg*. The big square in (c) represents the background vertex.

of contractions and dashed edges in Figure 2(c) denote the edges to be deleted. If needed we repeat this step until we simplify most of the multiple edges and self-loops, but not those enclosing any surviving part of the graphs, which are necessary to preserve the correct structure.

We iterate these two steps until there is no more edge to contract. In case of connected component analysis example Figure 2 shows that we cannot continue with the edge contraction, because we have found all the regions in the image.

The image graph pyramid for the example of the image shown in Figure 1(a) is given in Figure 3. For the sake of simplicity of the figure some of the relations between fathers (surviving vertices) and sons (non-surviving vertices) on different levels of the pyramid are not shown.

The number of nodes at the highest level of the pyramid correspond to

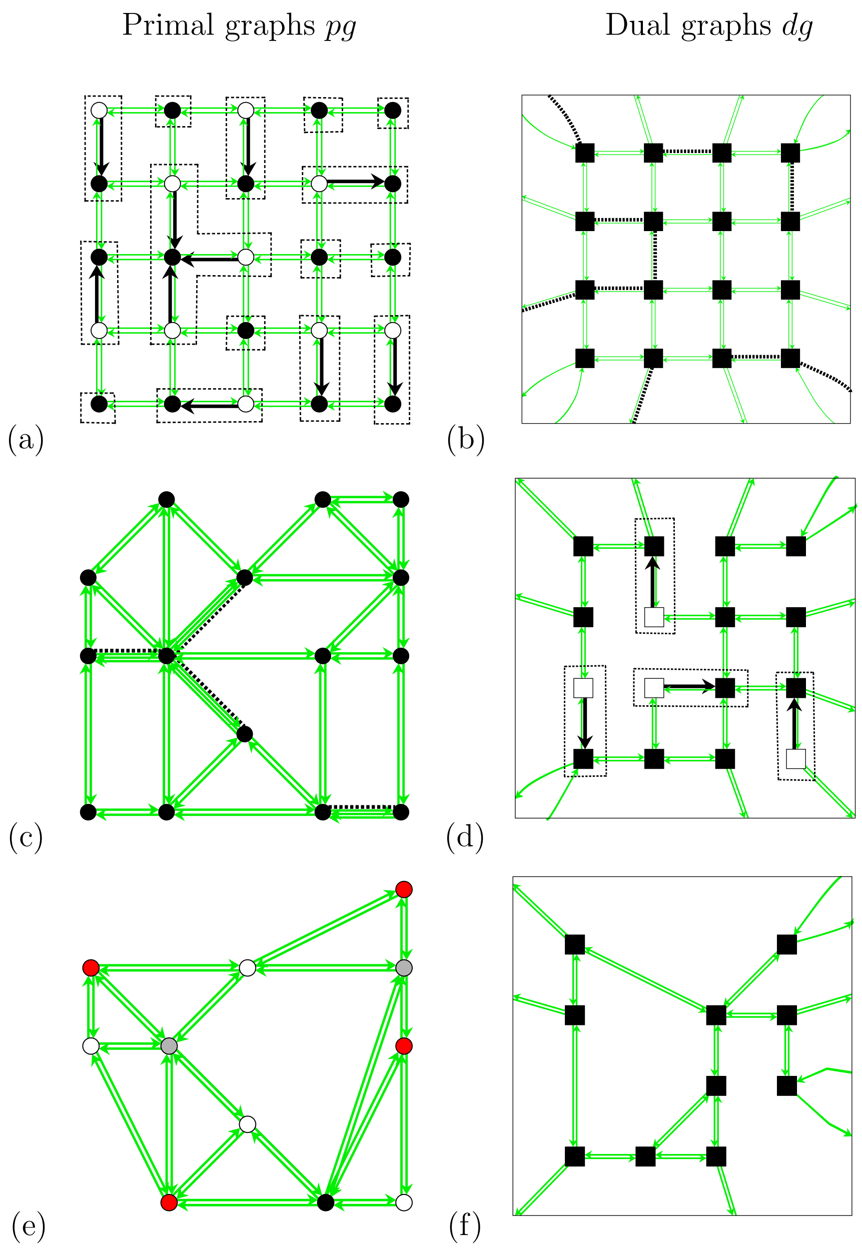


Figure 2: (a) Contraction kernels of the graph in Figure 1(b). (a),(d) Surviving vertices depicted with black, edges to contract with black arrows and (b),(c) edges to delete with dashed line. Small frames denote contraction kernels. (e),(f) After repeating DGC several times. The number of vertices in primal graph pg (e) correspond to the number of regions in the image Figure 1(a).

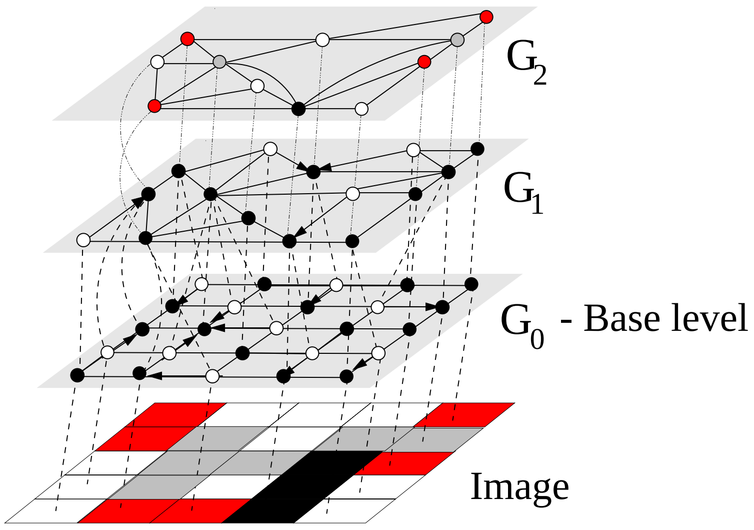


Figure 3: Image graph pyramid built on top of the image. Surviving vertices (fathers) are depicted with black, contracted edges with arrows, and non-surviving vertices (sons) with white (G_0, G_1). G_2 the top of the graph pyramid.

the number of regions in image, see Figure 3 level G_2 has 11 vertices, note the colours of the vertices.

3 Application of the *dgc_tool*

Before we start the construction of the graph pyramid, the image[2] is converted into a dual pair, primal base level *pbl* and dual base level *dbl* of (base) pairlevel (see Figure 1 as an example of how an image can be represented by graphs). Let the primal base graph in *pbl* be denoted by *pbg* and dual base graph in *dbl* be denoted by *dbg*, respectively.

In some cases, like digital elevation model, the value of a pixel refers to a point. In other cases, like gray level images from cameras, the value of a pixel refers to an area. The option *gray levels onto borders* interprets the vertices of *pbg* as pixel centers (points). With the option *gray levels into faces* the vertices of *dbg* represent the pixels (faces). In any case *pbg* and *dbg* form a

[2]in our implementation only no-compressed tiff images

dual pair of plane graphs. The attribute *attrib* of the edges and vertices is initialized for monotonic dual graph contraction [GEK99]. In this version of *dgc_tool* the initialization of the attributes is as follows:

gray levels onto borders The *attrib* value of the vertices of *pbg* are set to the gray values of the corresponding pixels. The *attrib* values of the edges of *pbg* are set to the minimum *attrib* value of the incident vertices. The *attrib* values of the edges of *dbg* are set to the *attrib* values of the corresponding dual edges in *pbg*. The *attrib* values of the vertices of *dbg* are set to the minimum *attrib* value of the incident edges in *dbg* (see Figure 4, left graphs).

gray levels into faces The *attrib* values of the vertices of *dbg* are set to the gray values of the corresponding pixels. The *attrib* value of the edges of *dbg* are set to the maximum *attrib* value of the incident vertices. The *attrib* values of the edges of *pbg* are set to the *attrib* values of the corresponding dual edges in *dbg*. The *attrib* values of the vertices of *pbg* are set to the maximum *attrib* value of the incident edges in *pbg* (see Figure 4, right graphs).

The structure of the pyramid depends on the contraction kernels. The contraction kernels are required to be trees of maximum depth of one. Contraction kernels are formed from a preselection of edges that is determined by the application, using maximal independent vertex set (MIS) [Mee89] or maximal independent edge set (MIES) [HGS+02]. That far, there are three applications implemented in the *dgc_tool*:

- (1) stochastic decimation [Mee89],
- (2) connected component analysis [KM95],
- (3) monotonic dual graph contraction [GEK99].

The command synopsis for the *dgc_tool* is :

```
dgc_tool option [filename]
```

We have these options:

- -i – Internal. A small picture (Figure 1) is used for testing purposes.

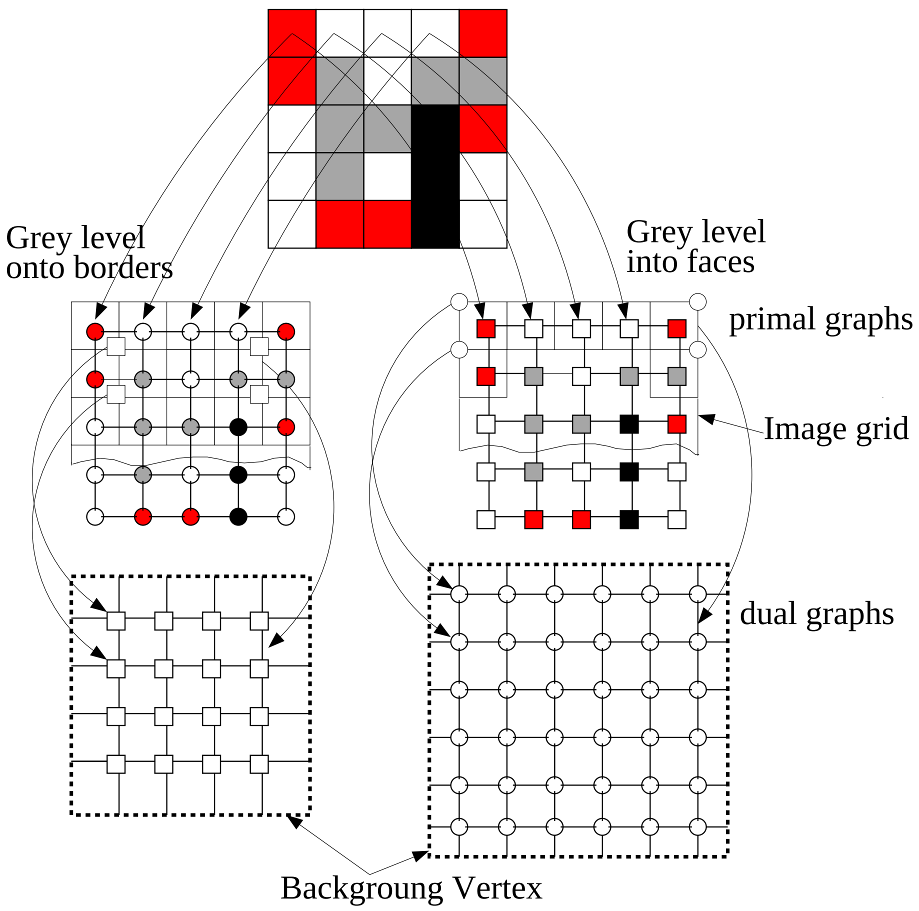


Figure 4: The attribute of the vertices.

- -t – TIFF[3] images. Only no compressed tiff images can be used.
- -o – Output. To down-project the basins into an output file.

During the executions of the *dgc_tool* a menu is given (see Figures 7) to help the user to choose where to put attributes; gray levels on borders with option 1 or gray levels into faces with option 2; then the user must decide which maximal independent set algorithm he/she chooses; MIS with option 1 [Mee89] or MIES with option 4 [HGS+02] and finally the user must choose

[3]**T**ag **I**mage **F**ile **F**ormat. More info [ASI02]

one of the application mentioned above; for stochastic decimation (default) option 1, connected component analysis option 2 (see Figures 9 and 8) or watershed segmentation with option 3.

3.1 Examples of the *dgc_tool*

To make these more clearly some examples are given. To test that everything is properly installed[4] the user should type the command:

```
dgc_tool -i
```

followed by the input 1 for gray level onto borders, 1 for the maximum independent algorithm and 1 for the application.

Tiff images can be put into the *dgc_tool* with the command:

```
dgc_tool -t tiff_filename1 [-o tiff_filename2]
```

To do connected component analysis on tiff image[5] for example the user should type:

```
dgc_tool -t circle.tif
```

followed by the input 1 (gray levels on borders), 1 (maximal independent algorithm) and 2 (connected component analysis) yields a graph consisting of two self-loops - one for the border of the circle, the other one for the edges of the image, see Figure 5.

Or another example:

```
dgc_tool -t circle_soebel.tif
```

followed by the input 1 (gray levels on borders), 1 (maximal independent algorithm) and 3 (watershed segmentation) yields a graph consisting of two self-loops - one for the border of the circle, the other one for the edges of the image.

To do watershed segmentation the options $-t$ plus the filtered image and $-o$ plus the name of the image (makes sense for monotonic dual graph contraction, only) are choosen.

[4]To check for example if LEDA libraries are properly installed.

[5]See Section 7 for images used in examples.

```
dgc_tool -t circle_sobel.tif -o circle.tif
```

followed by the input 1 (gray levels on borders), 1 (maximal independent algorithm) and 3 (watershed segmentation) yields down-projection of basins into *Output.pgm*. The basins are filled with the mean gray levels from original image *circle.tif*.

In both examples the dual of the graph is shown, if "done" is pressed in the graph window (see snapshots of the graph window Figure 9). In the dual graph the background vertex is depicted by the large square.

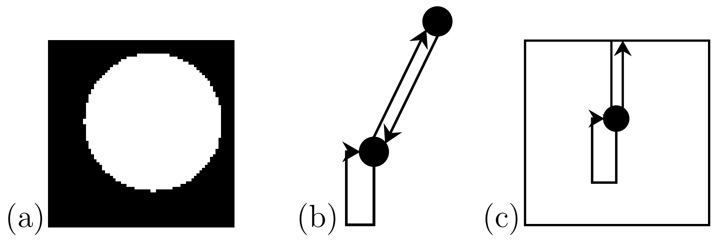


Figure 5: (a) Input image and the output of the *dgc_tool* for the connected component analysis; (b) primal graph and (c) dual graph.

4 Programming Interface

In this section the five principle classes: *pyramid, levelpair, level* and *dgc_node, dgc_edge* of *dgc_tool* are given in detail. A *pyramid* is a stack of *levelpair*, each of them consists of a pair of *levels*. Each *level* consist of a *GRAPH* among others and each *GRAPH* is made from *dgc_edge* and *dgc_node*, as can be seen from the class diagram (Figure 6).

4.1 *pyramid* Class

The class *pyramid* represents the hierarchies of levelpairs. A pyramid is created using the method

```
bool Image2Pyramid( string, int , string = "" ) ;
```

- **Typdefs and constants:**
- **Member function index:**

```
         pyramid()  ;
         ~pyramid() ;
    int  DgcAlternate( int, int ) ;
    bool Image2Pyramid( string, int , string = "" ) ;
    bool TIFF2Raster( string, string ) ;
    bool Internal2Raster( void ) ;
    void Raster2Pyramid( level *, level *, u_int, u_int ) ;
    void WriteAttribs( level *, level *, int ) ;
    void WriteGNodeAttribs( level *, level *, int ) ;
    void WriteGEdgeAttribs( level *, level *, int ) ;
    void WriteDGNodeAttribs( level *, level *, int ) ;
    void WriteDGEdgeAttribs( level *, level *, int ) ;
    void WriteExtraAttribs( level *, level *, int )  ;
    void WriteExtraGNodeAttribs( level *, level *, int ) ;
    void WriteExtraDGNodeAttribs( level *, level * ) ;
    void DownProject( u_long * ) ;
    void GraphToRaster( u_long *, int ) ;
```

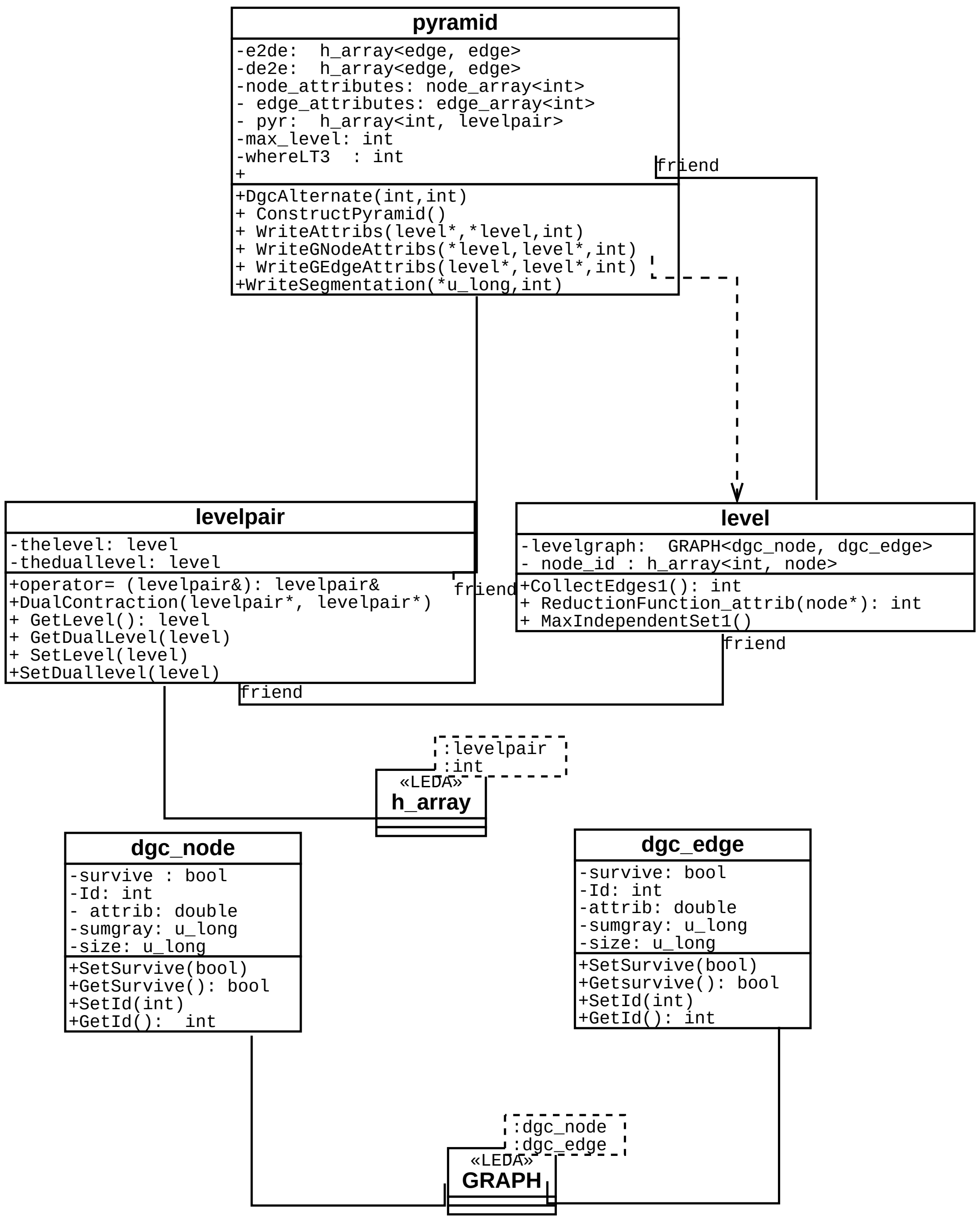


Figure 6: Presentation of a Class Diagram of the *dgc_tool*

- **Member function description:**

```
int  DgcAlternate( int, int ) ;
```

Let *pg* and *dg* denote the primal graph and the dual graph, respectively. The dual graph contraction is done as follows:

- **outer loop** - Graph contraction on *pg*,
- **inner loop** - Graph contraction on *dg* until there are no more redundant edges in *dg* i.e. self loops and parallel edges, which do not enclose a part of a graph that has survived.

Repeat these two steps until there is no edge in *pg* to be contracted (see [Kro94]). The input parameters of this method are application dependent.

```
bool Image2Pyramid( string, int , string = "" ) ;
```

Initializes the pyramid, using the raster which is created by the method `TIFF2Raster()`.

```
bool TIFF2Raster( string, string ) ;
```

Reads the TIFF picture and makes a raster from it.

```
bool Internal2Raster( void ) ;
```

Creates a raster from the internal picture. The internal picture can be called by option $-i$ in the command line.

```
void Raster2Pyramid( level *, level *, u_int, u_int ) ;
```

Creates the structure of the base levelpair (level 0) of the pyramid from raster.

```
void WriteAttribs( level *, level *, int ) ;
```

Assigns attributes in the graph and its dual in the correct order.

```
void WriteGNodeAttribs( level *, level *, int ) ;
void WriteGEdgeAttribs( level *, level *, int ) ;
void WriteDGNodeAttribs( level *, level *, int ) ;
void WriteDGEdgeAttribs( level *, level *, int ) ;
```

Assigns attributes in the vertices and edges of the graph G (*pg*) and DG (*dg*) respectively using pixels in the image i.e. raster. That is application dependent.

```
void DownProject( u_long * ) ;
```

The vertex attributes in the highest level of the pyramid are projected onto the receptive fields (iė. into the raster).

```
void GraphToRaster( u_long *, int ) ;
```

The resulting raster, which is a 2 dimensional array, is written to a pgm-file [6].

[6]**P**ortable **G**ray**m**ap file format

4.2 *levelpair* Class

The class *levelpair* represents the relationship between *levels*. Two new levels are created from two old ones using the method

```
void DualContraction( levelpair *, levelpair *,
                      h_array<edge, edge> *, bool, int )
```

- **typdefs and constants:**
- **Member function index:**

  ```
  levelpair(level, level)
  levelpair()
  ~levelpair()
  levelpair( const levelpair& )
  void DualContraction( levelpair *, levelpair *,
                        h_array<edge, edge> *, bool, int )
  level* GetLevel() ;
  level* GetDualLevel() ;
  void SetLevel(level)  ;
  void SetDualLevel(level) ;
  ```

- **Member function description:**

  ```
  void DualContraction( levelpair *, levelpair *,
                        h_array<edge, edge> *, bool, int )
  ```

  Contraction of *pg* or *dg* by means of a maximal independent set of vertices.

  ```
  level* GetLevel() ;
  level* GetDualLevel() ;
  void SetLevel(level)  ;
  void SetDualLevel(level) ;
  ```

  Methods to set respectively to get levels and dual levels, respectively.

4.3 *level* Class

The class *level* contains the principle methods necessary to select the contraction kernels.

- **Typdefs and constants:**

  ```
  typedef int (*CollectEdges)()
  enum { count = 4 }
  ```

- **Member function index:**

  ```
  level()  ;
  level( CollectEdges f ) ;
  level( const level& ) ;
  ~level() ;
  void RandFunc() ;
  bool MaxRandEdge( edge ) ;
  bool MaxRandNode( node ) ;
  bool MinRandSourceNode( edge ) ;
  int CollectEdges1( void ) ;
  int CollectEdges2( void ) ;
  int CollectEdges3( void ) ;
  int CollectEdges4( void ) ;
  int CollectRedundantEdges( int ) ;
  void MaxIndependentSetEdge() ;
  void MaxIndependentSetVertex() ;
  void DisplayOut( u_int, u_int, bool ) ;
  void ShowNodeAttribs( void ) ;
  void ShowEdgeAttribs( void ) ;
  void CountNodeSurvive( void ) ;
  void CountEdgeSurvive( void ) ;
  ```

- **Member function description:**

  ```
  void RandFunc() ;
  ```

  Assigns uniformly distributed random number as randomId to the vertices and edges. The seed is initialize with 0. Every time this function is called the seed is assigned a different value.

```
bool MaxRandEdge( edge ) ;
```

Returns *true* if the input edge has the largest random value in its neighborhood, *false* otherwise.

```
bool MaxRandNode( node ) ;
```

Returns *true* if the input node has the largest random value in its neighborhood, *false* otherwise.

```
bool MinRandSourceNode( edge ) ;
```

For each input edge e this method returns *true* if $source(e) \leq target(e)$, *false* otherwise.

```
int CollectEdges1( void ) ;
```

Collects edges whose end vertices have identical attributes (for connected component analysis).

```
int CollectEdges2( void ) ;
```

Collects all edges (for stochastic decimation).

```
int CollectEdges3( void ) ;
```

Collects edges if $attrib(target(edge)) \geq attrib(edge) \geq attrib(source(edge))$ (for monotonic dual graph contraction).

```
int CollectEdges4( void ) ;
```

Collects edges if $attrib(target(edge)) \leq attrib(edge) \leq attrib(source(edge))$ (for monotonic dual graph contraction).

```
int CollectRedundantEdges( int ) ;
```

Collects parallel edges and self-loops which do not contain a part of the surviving graph, i. e. edges incidented with a vertex with in-degree ≤ 2.

```
void MaxIndependentSetEdge() ;
```

Let E_f be the preselection of (survive = false) edges from one of the CollectEdges methods and let G_f be the subgraph of the primal or dual graph induced by E_f. This method finds a maximal subset E_m of E_f such that each component of the subgraph induced by E_m is a tree of depth smaller or equal to one. The algorithm of Peter Meer [Mee89] is applied to edges instead of vertices. The neighborhoods of an edge is defined in [HGS+02].

```
void MaxIndependentSetVertex() ;
```

This method finds a maximal independent vertex set (so called MIS) as described by Peter Meer [Mee89].

```
void DisplayOut( u_int, u_int, bool ) ;
```

Displays the graph *pbg* or *dbg*.

```
void ShowNodeAttribs( void ) ;
void ShowEdgeAttribs( void ) ;
```

Shows the attributes of the vertices of *pg* or *dg*.
Shows the attributes of the edges of *pg* or *dg*.

```
void CountNodeSurvive( void ) ;
void CountEdgeSurvive( void ) ;
```

Some statistics of surviving nodes and edges.

4.4 *dgc_node* Class

This class corresponds to a node of a graph.

- **Typdefs and constants:**
- **Member function index:**

```
  int Id ;
  int size ;
  int attrib ;
  double randomId ;
  bool survive ;
  int sumgray  ;
  void SetAttrib( u_long ) ;
  int GetAttrib( void )    ;
  void SetSurvive( bool )  ;
  bool GetSurvive( void )  ;
  void SetId( int ) ;
  int GetId( void ) ;
  void SetRandomId( double ) ;
  double GetRandomId( void )   ;
```

- **Member function description:**

```
int Id
```

Number attached to vertex for identification of vertex.
The x and y coordinates of a vertex are *id*, *mod*, *width* and *id / width* respectively. The variable *width* indicates the number of pixels in the rows of the image.

```
int size ;
```

The number of vertices in the receptive field of a node.

```
int attrib ;
```

see Section 3 for more details.

```
bool survive ;
```

True if the vertex survives, false otherwise.

```
double randomId ;
```

The random Id of the vertex.

```
int sumgray ;
```

Sum of the *attrib* values in the receptive field of a vertex.

```
void SetAttrib( u_long ) ;
int GetAttrib( void )    ;
void SetSurvive( bool )  ;
bool GetSurvive( void )  ;
void SetId( int ) ;
int GetId( void ) ;
void SetRandomId( double ) ;
double GetRandomId( void )   ;
```

Methods to set, respectively to get the attribute, the state of surviving of a vertex, the unique vertex Id and the random Id.

4.5 *dgc_edge* Class

This class corresponds to an edge of the graph.

- **Typdefs and constants:**
- **Member function index:**

```
int Id ;
int attrib ;
double randomId ;
bool survive ;
```

```
  void SetAttrib( u_long ) ;
  int GetAttrib( void )    ;
  void SetSurvive( bool )  ;
  bool GetSurvive( void )  ;
  void SetId( int ) ;
  int GetId( void ) ;
  void SetRandomId( double ) ;
  double GetRandomId( void )   ;
```

- **Member function description:**

  `int Id`

  Number attached to an edge for identification of the edge.

  `int attrib`

  see Section 3 for more details.

  `double randomId ;`

  The random number of the edge.

  `bool survive`

  True if the edge survives, false otherwise.

  ```
  void SetAttrib( u_long ) ;
  int GetAttrib( void )    ;
  void SetSurvive( bool )  ;
  bool GetSurvive( void )  ;
  void SetId( int ) ;
  int GetId( void ) ;
  void SetRandomId( double ) ;
  double GetRandomId( void )   ;
  ```

  Methods to set, respectively to get the attribute, the state of surviving of an edge, the unique edge Id and the random Id.

5 Where to find *dgc_tool* ?

The *dgc_tool* version 1.0 can be downloaded at the ftp://ftp.prip.tuwien.ac.at/pub/dgc_tool/dgc_tool_v1.0, it was compiled with *gcc* [FSF02] version *egcs-2.91.66* under Red Hat Linux 6.2/7.2 [RHI02]. An environment to draw and test graph can be also found at our ftp server. The latest free version of LEDA *v*3.8, can be downloaded from our ftp://ftp.prip.tuwien.ac.at/pub/dgc_tool/LEDA or current licensed one at official LEDA site www.algorithmic-solutions.com. Documentation about LEDA can be found at http://www.mpi-sb.mpg.de/LEDA/MANUAL/MANUAL.html or in the book [MN99].

6 Conclusion

In this technical report a new version of the *dgc_tool* was presented. This version is an intermediate version. In the future, we will redesign the *dgc_tool* as follows:

We will make two categories of classes:

- classes which do not depend on the application, and
- classes which depend on the application.

Thus, each time we have a new application, we do not need to redesign the whole *dgc_tool*. The application dependent classes will be derived from the classes independent of the application (using the principle of reuse and generalization).

7 Acknowledgment and Notes

We would like to thank prof. Walter G. Kropatsch for many discussions on dual graph contraction and image pyramids, and technician Frank Mayer for helping us to install the LEDA library and install and maintain the CVS server. We also thank DI Mickeal Melki and Georg Langs for proofreading and verifying Section 4 and 5.

References

[ASI02] Adobe System Incorporated. *http://partners.adobe.com/asn/developer/graphics/.* 345 Park Avenue San Jose,California 95110-2704,USA, 2002.

[BK93] Horst Bischof and Walter G. Kropatsch. Hopfield networks for irregular decimation. In Wolfgang Pölzleitner and Emanuel Wenger, editors, *Image Analysis and Synthesis*, pages 317–327. OCG-Schriftenreihe, Österr. Arbeitsgemeinschaft für Mustererkennung, R. Oldenburg, 1993. Band 68.

[BK99] Mark J. Burge and Walter G. Kropatsch. A minimal line property preserving representation of line images. *Computing*, 62:355 – 368, 1999.

[FSF02] Free Software Foundation. *GCC Manual.* 59 Temple Place - Suite 330 Boston, MA 02111-1307 US; http://gcc.gnu.org/, 2002.

[GEK99] Roland Glantz, Roman Englert, and Walter G. Kropatsch. Representation of Image Structure by a Pair of Dual Graphs. In Walter G. Kropatsch and Jean-Michel Jolion, editors, *2nd IAPR-TC-15 Workshop on Graph-based Representation*, pages 155–163. OCG-Schriftenreihe, Band 126, Österreichische Computer Gesellschaft, May 1999.

[HGS$^+$02] Yll Haxhimusa, Roland Glantz, Maamar Saib, Langs, and Walter G. Kropatsch. Reduction Factors of Image Pyramid on Udirected and Directed Graph. In *Proceedings of the 7th. Computer Vision Winter Workshop*, 2002.

[KB98] Walter G. Kropatsch and Mark Burge. Minimizing the Topological Structure of Line Images. In Adnan Amin, Dov Dori, Pavel Pudil, and Herbert Freeman, editors, *Advances in Pattern Recognition, Joint IAPR International Workshops SSPR'98 and SPR'98*, volume Vol. 1451 of *Lecture Notes in Computer Science*, pages 149–158, Sydney, Australia, August 1998. Springer, Berlin Heidelberg, New York.

[KBBS98] Walter G. Kropatsch, Mark J. Burge, Souheil Ben Yacoub, and Nazha Selmaoui. Dual graph contraction with LEDA. *Computing, Supplementum: Graph Based Representations in Pattern Recognition*, No. 12:pp. 101–110, 1998.

[KLB99] Walter G. Kropatsch, Aleš Leonardis, and Horst Bischof. Hierarchical, Adaptive and Robust Methods for Image Understanding. *Surveys on Mathematics for Industry*, No.9:1–47, 1999.

[KM95] Walter G. Kropatsch and Herwig Macho. Finding the structure of connected components using dual irregular pyramids. In *Cinquième Colloque DGCI*, pages 147–158. LLAIC1, Université d'Auvergne, ISBN 2-87663-040-0, September 1995.

[Kro94] Walter G. Kropatsch. Building Irregular Pyramids by Dual Graph Contraction. Technical Report PRIP-TR-35, Institute f. Automation 183/2, Dept. for Pattern Recognition and Image Processing, TU Wien, Austria, 1994. Also available through http://www.prip.tuwien.ac.at/ftp/pub/publications/trs/.

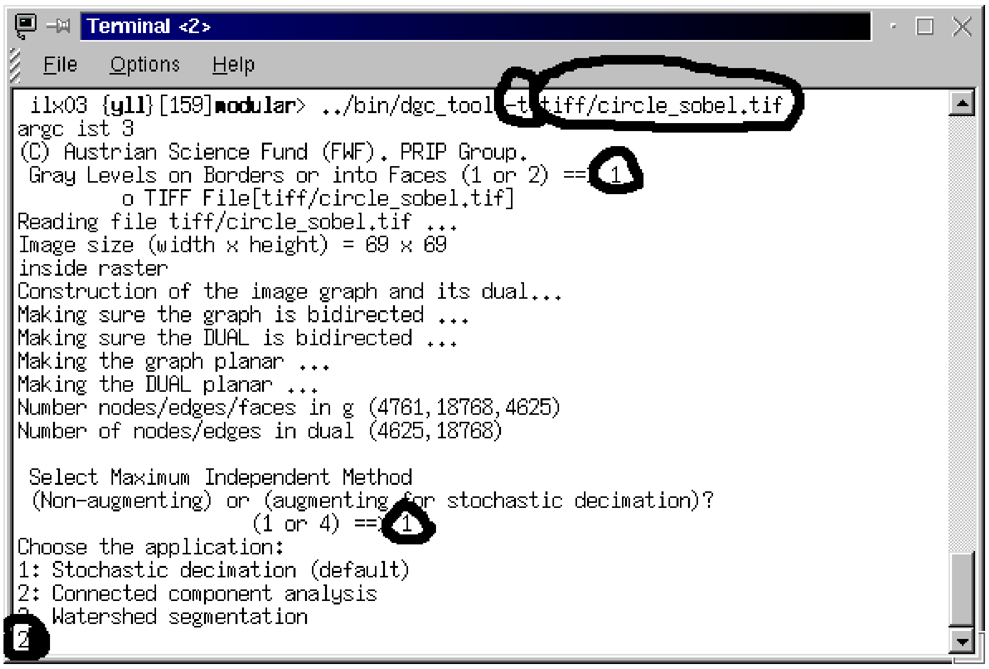
[Kro95] Walter G. Kropatsch. Building Irregular Pyramids by Dual Graph Contraction. *IEE-Proc. Vision, Image and Signal Processing*, 142(6):366 – 374, 1995.

[Mee89] Peter Meer. Stochastic image pyramids. *CVGIP*, 45:269 – 294, 1989.

[MN99] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing.* Cambridge University Press, Cambridge, U.K., 1999.

[RHI02] Red Hat Inc. *http://www.redhat.com/.* 1801 Varsity Drive Raleigh, NC 2760659 US;, 2002.

8 Appendix: Some Snapshots of *dgc_tool*

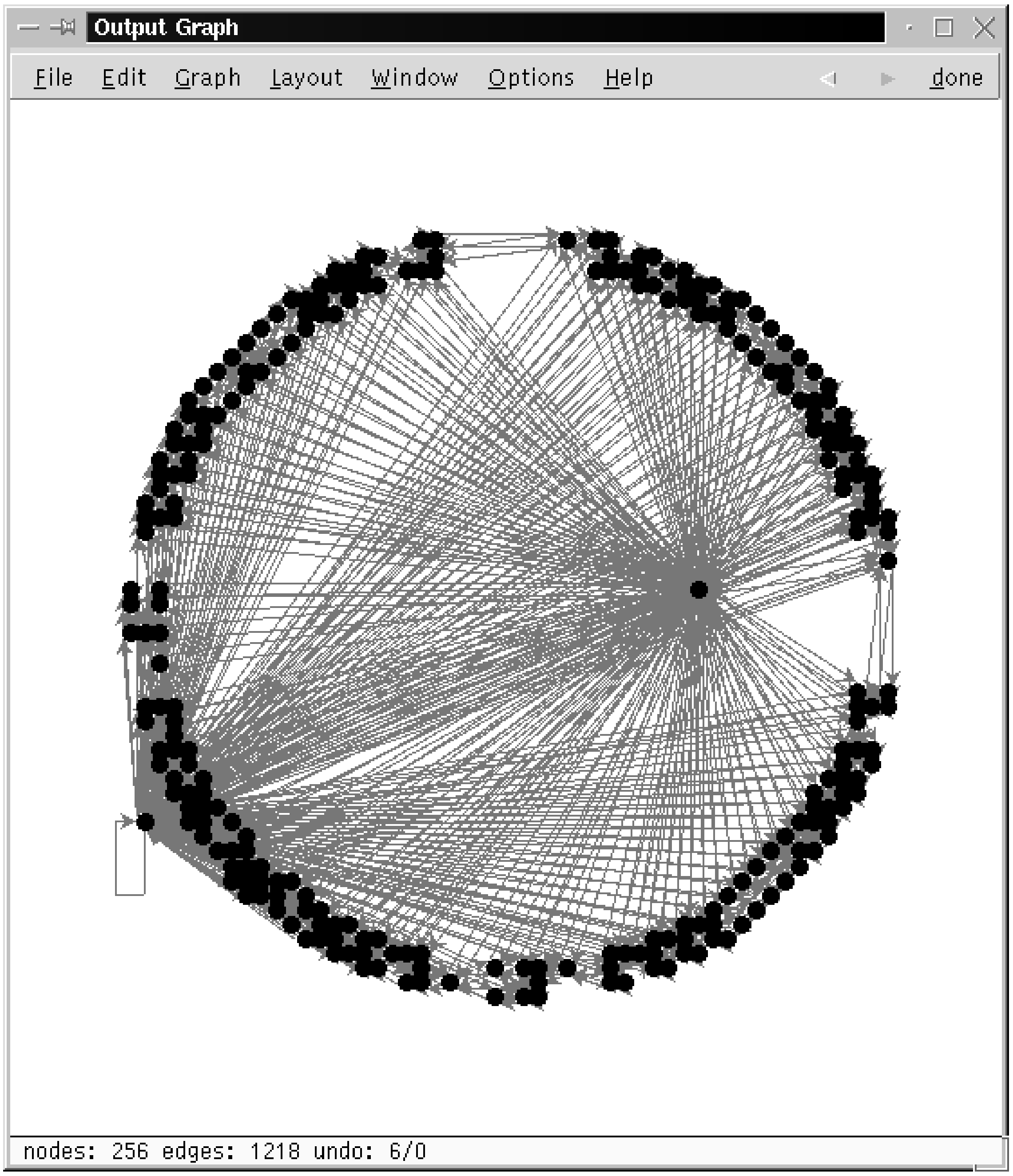


```
 ilx03 {yll}[159]modular> ../bin/dgc_tool -t tiff/circle_sobel.tif
argc ist 3
(C) Austrian Science Fund (FWF). PRIP Group.
 Gray Levels on Borders or into Faces (1 or 2) ==1
         o TIFF File[tiff/circle_sobel.tif]
Reading file tiff/circle_sobel.tif ...
Image size (width x height) = 69 x 69
inside raster
Construction of the image graph and its dual...
Making sure the graph is bidirected ...
Making sure the DUAL is bidirected ...
Making the graph planar ...
Making the DUAL planar ...
Number nodes/edges/faces in g (4761,18768,4625)
Number of nodes/edges in dual (4625,18768)

 Select Maximum Independent Method
 (Non-augmenting) or (augmenting for stochastic decimation)?
                   (1 or 4) ==1
Choose the application:
1: Stochastic decimation (default)
2: Connected component analysis
3: Watershed segmentation
2
```

Figure 7: A snapshot from the command window.

Figure 8: The input image.



```
nodes: 256 edges: 1218 undo: 6/0
```

Figure 9: The result of the *dgc_tool* doing connected commponent analysis, input parameters: 1, 1, 2.