Dual graph contraction with LEDA*

W.G. Kropatsch, M. Burge‡, S. Ben Yacoub, N. Selmaoui
Vienna University of Technology
Institut for Automation 183-2, PRIP Group
Trettlstrasse 3, A-1040 Wien, AUSTRIA
krw, sby, nazha@prip.tuwien.ac.at
‡ Johannes Kepler University, Department of Systems Science
Computer Vision Laboratory, A-4040 Linz, Austria
burge@cast.uni-linz.ac.at

Abstract

Graphs are useful tools for modeling problems that occur in a variety of fields. In machine vision graphs based solutions have been successfully applied to many image processing problems e.g. quad trees for image compression and regional adjacency graphs for segmentation. The application of graphs to machine vision problems poses special problems due to the underlying size of the image e.g. a graph representing the base level of a 512x512 image has over 200,000 nodes. The large size of the graphs make issues of both space and time complexity important when designing algorithms for machine vision problems. We present an implementation under LEDA (Library of Efficient Data structures and Algorithms) of DGC (dual graph contraction) for irregular pyramids. In the first section we present the theory behind DGC, in the second an algorithmic specification is derived, and in the third an implementation under LEDA is given followed by a short conclusion.

1 Theory of DGC

The presented approach addresses a representation of pure structure, a hierarchy of plane graphs, with a clear interface, the decimation parameters, to control generation and modification of the structure. Dual graph contraction is the basic process [3] that builds an irregular 'graph' pyramid by successively contracting a dual image graph of one level into the smaller dual image graph of the next level. Dual image graphs are typically defined by the neighborhood relations of image pixels or by the adjacency relations of the region adjacency graph. The above concept has been used for finding the structure of connected components [5]. It also embeds Meer's stochastic pyramid [7], the adaptive pyramid [2], and a further variant of Meer's approach, Mathieu's optimal stochastic pyramid [6] which produced excellent segmentation results by decimating a minimal spanning tree instead of the original graph.

*This work was supported by the Austrian Science Foundation under grant number S7002-MAT.
Dual graph contraction proceeds in two basic steps (Fig. 1): dual edge contraction and dual face contraction. The base of the pyramid consists of the pair of dual image graphs \((G_0, \overline{G_0})\). Following *decimation parameters* \((S_i, N_{i+1})\) determine the structure of an irregular pyramid \([3][Def.5]\): a subset of surviving vertices \(S_i = V_{i+1} \subset V_i\), and a subset of primary non-surviving edges \(N_{i+1} \subset E_i\). Every non-surviving vertex, \(v \in V_i \setminus S_i\), must be connected to one surviving vertex in a unique way. The relation between the two pairs of dual graphs \((G_i, \overline{G_i})\) and \((G_{i+1}, \overline{G_{i+1}})\), as established by dual graph contraction with decimation parameters \((S_i, N_{i+1})\) is expressed by function \(C[., .]\):

\[
(G_{i+1}, \overline{G_{i+1}}) = C[(G_i, \overline{G_i}), (S_i, N_{i+1})]
\]

(1)

The contraction of a primary non-surviving edge consists in the identification of its endpoints and in the removal of both the contracted edge and its dual edge. Dual face contraction simplifies most of the multiple edges and self-loops, but not those inclosing any surviving parts of the graph (see [3]). One step of dual graph contraction is illustrated in Fig. 2. Note that the contracted graph may contain both a self-loop and multiple edges. They are necessary to preserve the structure defined in the base graph [3].

To define the parameters that control the process of dual graph contraction we observe that the subgraphs in our example graph (Fig. 2b) form small tree structures \(T(s)\) that collapse into surviving vertex \(s\) of the contracted graph. \(T(s)\) is a spanning tree of the connected component of the surviving root vertex, or equivalently, \((V, N)\) is a spanning forest of graph \(G(V, E)\).

**Definition 1** A **decimation of a graph** \(G(V, E)\) is specified by a selection of surviving vertices \(S \subset V\) and a selection of primary non-surviving edges \(N \subset E\) such that following two conditions are fulfilled:

1. Graph \((V, N)\) is a spanning forest of graph \(G(V, E)\).
2. The surviving vertices \(S \subset V\) are the roots of the forest \((V, N)\).

\(\)Secondary non-surviving edges are removed during dual face contraction.
The trees $T(v)$ of the forest $(V, N)$ with root $v \in V$ are called contraction kernels.

The connectivity structure of the contracted graph is established by paths connecting two surviving vertices:

**Definition 2** Let $G(V, E)$ be a graph with decimation parameters $(S, N)$. A path in $G(V, E)$ is called a connecting path between two surviving vertices $v, w \in S$, denoted $CP(v, w)$, if it consists of three subsets of edges $E$ (Fig. 3):

1. The first part is a possibly empty branch of contraction kernel $T(v)$.

2. The middle part is an edge $e \in E\setminus N$ that bridges the gap between the two contraction kernels $T(v)$ and $T(w)$. We call $e$ the bridge of the connecting path $CP(v, w)$.

3. The third part is a possibly empty branch of contraction kernel $T(w)$.

Connecting paths $CP(v, w)$ in $G(V, E)$ are strongly related to the edges in the contracted graph $G'(V', E')$: Two different surviving vertices that are connected by a connecting path in $G$ are connected by an edge in $E'$. For every edge $e' = (v, w) \in E'$ there exists a connecting path $CP(v, w)$ in $G$. Dual edge contraction can be implemented by (1) simply renaming all the non-surviving vertices to surviving parent vertex, (2) deleting all non-surviving edges $N$ and (3) their duals $\overline{N}$.

## 2 Algorithmic Specification of DGC

In the last section we provided the theory of DGC, from the theory we will now derive an algorithmic specification. The specification is given primarily in the flowchart (Figure 2) and uses the same notation as in the previous section. The
Image_to_Graph module transforms and image into a representation in which each pixel is a vertex with edges to its eight connected neighbors. The dual graph is also created at this time as a separate but linked graph in a way that a syntactic equivalence between the graph and its dual is provided while maintaining the semantic difference. The Dual_Graph_Contraction module with (S,N) decimation (N is oriented as shown in Figure 2b) parameters illustrates the main algorithm for DGC and calls the following modules: Dual_Edge_Contraction and Dual_Face_Contraction use Substitute, which links nodes to the surviving root using an iterative substitution algorithm.

```
Dual_Edge_Contraction
Edge_Contract(G,S,N) {
    for all edges(e=(v,w),N)
    { G.substitute(v,root(v));
        G.substitute(w,v);
        G.del_edge(e);
    }
    for all edges(e=(v,w),G)
    { G.new_edge( e'=(root(v),root(w)));
    }
} Edge_Remove(G,N) {
    for all edges(\( e \in \bar{N} \))
    G.del_edge(\( e \));
}
Dual_Face_Contraction {
    F is set of faces to contract
    M set of edges in \( \bar{E} \) to contract
    Dual_Face_Select(\( G,F,M \)) {
        for all edges(\( \overline{\nu},\overline{G} \))
        { if deg(\( \overline{\nu} \)) \( \geq 3 \) then
            F.append(\( \overline{\nu} \));
            M.insert(\( G.\text{adj}_edge(bar{e}) \));
        }
        Edge_Contract(G,F,M);
        Edge_Remove(G,M);
    }
}
```

### 3 Implementation of DGC in LEDA

LEDA[8] (Library for Efficient Data types and Algorithms) is a C++ class library implementing many common abstract data types e.g. trees, graphs, and lists. The graph support in LEDA includes iterators like `for all nodes` of
graph G do", and "forall adj edges of v do" as well as basic operations like "delete node" and "delete edge". An instance G of the data type graph consists of a list V of nodes and a list E of edges, where node and edge are LEDA classes. A pair of nodes (v, w) ∈ V × V is associated with every edge e ∈ E, v is called the source of e, w the target of e, and v and w are the endpoints of e. In LEDA a graph is either directed or undirected where the difference is in the way the edges incident to a node are stored and how the concept of adjacent is defined. In directed graphs two lists of edges are associated with every node v: adjiTedges(v) = { e ∈ E | v = source(e) }, i.e., the list of edges starting in v, and inTedges(v) = { e ∈ E | v = target(e) }, i.e., the list of edges ending in v. The list adjiTedges(v) is called the adjacency list of node v and the edges in adjiTedges(v) are called the edges adjacent to node v. For directed graphs we often use outTedges(v) as a synonym for adjiTedges(v).

LEDA takes advantage of C++ templates to allow for truly generic algorithms. Templated graphs are similar to attributed graphs, in an attributed graph, for every vertex v and every edge e of G(V, E), there exists a function f(e) which returns the attribute of vertex v and g(e) which returns the attribute of edge e, in C++ notation that would be GRAPH G(V, E), with G[V] returning the attribute of vertex v and G[E] returning that of edge e. Graph algorithms developed using templated graphs are generic and reusable software components. DGC (Dual Graph Contraction) has been implemented using templates so that the process can be applied to a wide variety of attributed graphs. Our goals in designing the DGC class were software reuse and readability. We have not optimized the implementation for size or speed, instead slower code which makes the steps of the algorithms obvious has been preferred.

The DGC class encapsulates all the data types and algorithms necessary to conduct multiple levels of dual graph contraction on attributed graphs. The DGC class contains two publicly accessible graphs, g, GRAPH<dgc node, dgc edge> g, and the dual g, GRAPH<dgc dual node, dgc dual edge> dg, both of which are templated to allow the user to provide their own attributes for both the vertices and edges of g and g by redefining the classes dgc node, dgc edge, dgc dual node, dgc dual edge. DGC also contains two publicly accessible algorithms or methods, contract() which performs one level of contraction and selection() which allows one to interactively define the contraction kernel.

class dgc { // Dual Graph Contraction under LEDA
public:
    GRAPH<dgc node, dgc edge> g;
    GRAPH<dgc dual node, dgc dual edge> dg;

public:
    void contract(void); // perform dual graph contraction
    void selection(void); // interactive selection of the contraction kernel
};

The code fragment above shows the public parts of the interface for the DGC class. The graph to be contracted is stored in the templated graph g (GRAPH<dgc node, dgc edge> g). The classes dgc node and dgc edge can be subclassed to store user defined data, a typical usage would be storing the gray values of a pixel. After the user has redefined the attributes of g and possibly of

5
the dual $\tilde{g}$, the remaining step is to define the decision method used to determine the contraction kernel.

The class method `contraction` has a default implementation which provides a GUI for interactively defining and saving a contraction kernel. The user selects the oriented edges which should be contracted so that their source node remains. The contraction kernel is stored internally as a list of oriented edges. When replacing this class method the user designs an algorithm which examines the attributes of the nodes, edges of $g$ and possibly $\tilde{g}$ and decides upon which edges in $g$ will be contracted and which of their nodes (source or target) will remain.

Once the contraction kernel has been defined the class method `contract` is called which first contracts edges of $g$ and $\tilde{g}$ based on the kernel, and then unnecessary faces, as defined earlier, of $\tilde{g}$. In addition `contract` will construct the dual graph, $\tilde{g}$ of $g$, by calling the class method `compute_dual` if no dual was supplied.

```cpp
void dgc::contract(void) {  // perform dual graph contraction
  if(dg.number_of_edges()) compute_dual();  // first time compute dual of g
dual_edge_contraction();
dual_face_contraction();
}
```

The class method `dual_edge_contraction` randomly selects and deletes edges from the contraction kernel and calls the class method `edge_contract_g` to contract that edge of $g$ until the contraction kernel is empty.

```cpp
void dgc::dual_edge_contraction() {
  while(! contraction_kernel->empty() ) {
    edge e = ce->choose(); ce->del( e );
    edge_contract_g( g,source(e) , e );
  }
}
```

The class method `dual_face_contraction` is called after contraction in $g$ to eliminate any unnecessary, as previously defined, faces which may have resulted in $g$. The size of each face of $g$ which is represented by the cardinality of a corresponding node in $\tilde{g}$ is examined and any face with degree less than three and not the background face, is added to the set of contraction nodes $cn$. A single node is selected from this set, and then a single edge from this node is selected to be contracted by `edge_contract_dg`. Since the topology of $g$ and $\tilde{g}$ are changed by this contraction, the algorithm checks if $g$ still contains any unnecessary faces and if so restarts.

```cpp
void dgc::dual_face_contraction(void) {
  int change = 0; node_set cn(dg); edge_set ce(dg);
  do {
    change = 0;
    node v; forAll_nodes(v, dg) {
      if( (dg.outdeg(v) < 3) && (v != BackgroundFace) ) cn.insert( v );
    }
    if(! cn.empty() ) {
      change = 1;
    }
  } while( change == 0 );
}
```
v = cn.choose(); cn.clear();
edge e; forall_out_edges(e, v) { ce.insert( e ); }
if(! ce.empty() ) {
    e = ce.choose(); ce.clear(); change++;
    edge_contract_dg( dg.target(e), dg.reverse(e) );
}
} while(change);

The remaining contraction methods edge_contract_g and edge_contract_dg are similar and only the later will be discussed. The class method edge_contract_g takes two parameters, the edge to be contracted, e, and the node, n of that edge which should remain, currently the node is not necessary since the contraction kernel defines the source node of the edge as the one which will remain. In contraction e is deleted and all edges incident to the target node (in the algorithm cn) of e are moved to n and cn is deleted.

void dgc::edge_contract_dg(node v, edge e) {
    edge_set oes(dg);
    node contract_node = dg.target(e);
    edge_remove_dg( e ); // remove now so e won't be considered in out_edges
    edge oe; forall_out_edges( oe, contract_node ) { oes.insert( oe ); }
    while(! oes.empty() ) {
        edge oe = oes.choose(); edge roe = dg.reversal(oe);
        node target_node = dg.target(oe);
        dg.move_edge( oe, v, target_node );
        dg.move_edge( roe, target_node, v );
        oes.del( oe );
    }
    dg.del_node( contract_node );
}

4 Conclusion

The dgc class is implemented under LEDA and a beta release of the extendable C++ class is available which supports dual graph contraction and the saving and editing attributed graphs and contraction kernels with a GUI interface, as well as animation of the contraction process. The first public release of the system planned for release in summer of 1997, will include image to graph constructors and commented sample applications e.g. watershed segmentation and voronoi diagram generation, to assist the end user in developing their own applications.

In dual graph contraction, the decimation parameters control the process that iteratively builds an irregular (graph) pyramid, to specify these parameters the concept of the contraction kernel was introduced. It has been shown that dual graph contraction on g preserves connectivity, planarity, and the face degrees of g. Dual graph contraction provides a general method for specifying segmentations and can be applied to irregular pyramids which are capable of representing all possible segmentations (as defined by Pavlidis[9]) within a
single pyramid level[4]. The process may also be applied to regular pyramids, but it has been shown by Bister[1] that there exist segmentation which are not encodable in a single level of such a pyramid.

References


