Technical Report TR-20100901-1
Facet Detection and Visualizing Local Structure in Graphs

Dana Vrajitoru
Indiana University South Bend
Computer and Information Sciences Department

September 1, 2010

Abstract

In this paper we present an algorithm for facet detection in graphs. We define the facets as simple cycles in the graph minimizing the length. First, we present and analyse an algorithm for cycle detection based on the breadth-first search. Next, the facet detection algorithm starts with the local structure of simple cycles intersecting in a particular vertex of the graph. We then reduce the problem of facet detection to an instance of the Traveling Salesman Problem in an induced co-cyclic graph and use a method inspired from the minimum spanning tree to construct a circuit solution. We show some results of our algorithm and introduce some applications for it.

1 Introduction

In this paper we present an algorithm for facet detection in a graph, defined in terms of a set of simple cycles intersecting in a particular vertex of the graph. The goal of the algorithm is primarily to emphasize the local structure in the graph surrounding a particular vertex, which can provide a better understanding of the graph by visualization. Other applications are also foreseeable for this algorithm, for example, computing the total surface of the graph.

Facet detection algorithms have many applications. Recently, they have been employed for detecting and tracking features in 2D images and in animations [13], and for automatic object recognition [5].

It is often the case that objects are initially defined as clouds of points [1], in particular when they represent implicit surfaces or when they are captured by a 3D scanner. The most often, a mesh is reconstructed from the cloud of points which can take an intermediate step in the form of a graph [3]. In this study, a cloud of points obtained from a 3D scanning device is first transformed into a graph. A second step constructs the surface by a facet detection algorithm.

A related problem is the edge detection in images [14], which is often based on pixel analysis. The facet detection methods consider the facets as collections of edges with particular properties, for example, convexity in terms of polygons. Zheng and Tian [20] for example, start with an edge detection algorithm, followed by facet detection using a combination of gradient based parametric facet detection and zero crossing for the edges. In a related study [12], features are extracted by hysteresis thresholding followed by a step consisting in reconstructing a minimum spanning graph to emphasize contours for 2D and 3D data. In this study, the cycles represent an important criterion to recognizing features.

The idea of feature extraction is closely related to the notions of facet and polygon. Iqbal and Aggarwal [9] propose detecting structures in two dimensional images by extracting line segments and then building a cotermination graph. The method employs graph theory to detect polygons in the image as fundamental circuits in the graph. A maximum spanning tree is used for the detection, making use of the spatial representation of the graph.

In this paper we propose a method for extracting local structures in graphs. We start from the premise that the graph represents a solid object with a well-defined surface. We assume that the surface itself is not known, and that we do not possess information about the geometry of the graph. We are interested in the intrinsic local structure of the graph outside of the geometric considerations.

The algorithm we propose follows the general idea that a facet is a simple cycle in a graph with some properties of minimality. In a related paper [3], Azernikov and Fischer start from an edge to detect a facet, using properties of convexity of the resulting polygon and minimizing the length of the cycle. We follow a different approach, going from all the simple cycles intersecting in a vertex to selecting those that might define the actual surface. Our approach differs from that of related work in the sense that we do not use the geometric representation of the graph, but only the internal structure in terms of edge connections.

Our algorithm focuses on one vertex at a time and detects the local polygons that the vertex belongs to using a minimal cycle method. The set of all such cycles are meant to represent the surface of the object represented by the graph. They constitute a minimal neighborhood of the object that is topologically open. In the current form we do not consider information about the weights in the graph, but the algorithm can be extended to find the minimal cycle set in terms of weights.

Starting from all the simple cycles containing a vertex, we build the structure of a co-cyclic graph. Similar in nature to a dual graph [18], the co-cyclic graph contains edges for pairs of vertices that belong to the same simple cycle. A Hamiltonian circuit in the co-cyclic graph provides the local structure of the surface in the original graph.

The main application of this algorithm is to emphasize the local structure in the graph for purposes of visualization. As a potential additional application, it can be used to compute the total surface of the graph. Efficient solutions have been proposed for meshes [19], starting from the assumption that the polygons composing the surface have already been defined. Our algorithm can be used as a preliminary step for this problem.

2 Cycle Detection and Co-Cyclic Graph

Let $G = \{V, E\}$ be an undirected graph with n vertices and m edges. Two vertices belonging to the same edge are called adjacent. An edge is incident in a vertex if the vertex belongs to the edge. We extend this notion to a path and thus, also to a cycle: a path is incident in a vertex if the vertex belongs to it. A cycle is called simple if it doesn't contain any subcycle.

Definition 2.1. *Two vertices in a graph are* **co-cyclic** *if they belong to a common simple cycle.*

Thus two vertices are co-cyclic if there is a simple cycle that is incident in both of them. In particular, two adjacent vertices u and v are always co-cyclic because the cycle uvu is simple. We call such cycles trivial and they are not the focus of our algorithm.

We use the notion of co-cyclic vertices in the sense most often used in chemistry and not in relation to the notion of cocyle in a graph [18].

We will start by analyzing the subset of the cycle space containing all the simple cycles that intersect in a given vertex of the graph, or origin. From this subset we can construct a weighted co-cyclic graph where an edge is drawn between two vertices if they are both adjacent to the origin and co-cyclic. The weight of the edge will be the length of the smallest cycle containing both vertices.

The co-cyclic graph can be used to extract the cycle structure or the facets that intersect in the origin.

2.1 Cycle Detection

The issue of detecting a cycle in a graph is a simple algorithm and an early description can be found in [7] We start by presenting a cycle detection algorithm based on the breadth-first search (BFS) and prove some of its important properties. We will consider the case of an undirected graph. For a directed graph the depth-first search (DFS) is a more appropriate algorithm.

Existence

An undirected graph with at least one edge contains at least one simple cycle using only that edge in both directions. It contains one such cycle for every edge. We are interested in the existence of cycles of length greater than two, excluding such trivial cases. The algorithm detecting non-trivial cycles in a graph can be based on the breadth-first search which is linear over the number of edges m.

Sufficient condition. In an acyclic undirected graph, $m < n$. This is a necessary but not sufficient condition for the graph to be acyclic. A graph can have $n - 1$ or less edges, but be composed of several connected components and still present a non-trivial cycle. For example, the graph in Figure 1 has 5 vertices and 4 edges and presents a non-trivial cycle.

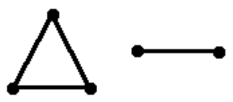

Figure 1: A non-trivial cycle in a graph where $m = n - 1$

Thus, we can express a sufficient but not necessary condition for the existence of a non-trivial cycle in the graph as

$$m \geq n \tag{1}$$

In the case where the sufficient condition is not true, the number of edges is less than the number of vertices, so the algorithm verifying the existence of cycles is linear over n.

Detection Algorithm

The algorithm detecting non-trivial cycles requires a small modification to a classic breadth-first search (BFS) or depth-first Search (DFS) [17]. This algorithm represents a test that will always return true in the case where a non-trivial cycle exists in the graph. Thus, it implicitly contains the necessary condition for the existence of a non-trivial cycle in the graph. Because of its constructive nature, at the same time it can be used to build a set of cycles in the graph.

Both the BFS and DFS can be used for this purpose. They both present the property of visiting every vertex only once and every edge only once. For the purpose of construct the set of simple cycles of minimal length that can be part of the object surface, the BFS is a better choice.

It is a known fact that when either of these algorithms encounters the same vertex twice, a cycle has been closed in the graph. If the graph is undirected, in many cases this happens when closing a trivial cycle, meaning when the algorithm attempts to go back on a visited edge. For example, if the algorithm considers the vertex u first, and then its neighbor v, then in an undirected graph u is also a neighbor of v. When the neighbors of v are processed in a later step, since u has been visited before, the algorithm might prematurely conclude on the existence of a cycle. In this case it is a trivial one: uvu.

Necessary condition. Thus, the necessary condition for the existence of a non-trivial cycle in the graph is that in the execution of the BFS or DFS, starting from a vertex v we encounter a neighbor u that has already been visited, such that u is not the predecessor of v in the search:

$$u \neq pred(v) \tag{2}$$

For the purpose of our algorithm, we have chosen the BFS because it is more likely to detect simple cycles of minimal length containing the origin of the search. The BFS starts from an origin vertex and explores all of the vertices in the order of their distance from the origin. Typically, a queue is used to store the visited vertices that may still have non visited neighbors. When a vertex is processed, its neighbors that have not been visited yet are added to the back of the queue. If we need to store the path from the origin to a destination vertex in the graph, then for each vertex we record its predecessor in the search. Traditionally, the BFS marks the visited vertices to avoid cycles. The same mark can be used to detect the cycles in our case.

Let us denote by o the origin of the BFS. Let us consider a particular step of the BFS where the current vertex is x and y is one of its neighbors. The following three cases can occur:

1. y has not been marked yet. In this case y is a vertex not yet visited and the BFS can continue the search by adding it to the queue and marking it.
2. $y = pred(x)$ or $x = pred(y)$. In this case the algorithm has found a trivial cycle by retracing one of its steps. In this case we can ignore the vertex y.
3. Otherwise y is a vertex that has been encountered before on a different path. Thus, by concatenating the paths from the origin to these two vertices and the edge to construct a cycle.

If our goal is to detect the existence of non-trivial cycles in the graph, then encountering the third case described above is a necessary and sufficient condition.

To construct the actual cycle, we first construct the path from o to x by following the predecessors backward to the origin o and by reversing the order. Then we continue this path with the edge xy and the entire path from y to o. The cycle can then be described as $o \to x - y \to o$.

It can happen that the paths from o to x and to y have a part in common. Since every vertex has a unique predecessor in the search, the cycle we obtain is either simple, or composed of a simple cycle not containing the origin concatenated with a trivial one. In the second case, the simple cycle thus extracted cannot be added to the set of simple cycles incident in the origin.

The cycle detection algorithm, called Breadth-First Cycle Detection (BFCD), presented on the next page, builds a set of simple cycles incident in the origin o. It can be used to detect a cycle by returning true when the first non-trivial cycle is detected. In the case where the first connected component does not contain a cycle, the algorithm must restart from another non-marked origin until the graph is completely visited or until a cycle is detected.

2.2 Induced Co-Cyclic Graph

Let us recall from Definition 2.1 that two vertices in the graph are called co-cyclic if they belong to the same simple cycle.

Theorem 2.2. *If two neighbors of a vertex o are co-cyclic, then there exists a simple cycle containing o and both of these vertices.*

Proof. Let the two vertices be u and v. Suppose that we have a simple cycle containing both u and v but not o. We can express the cycle as $u\,P_1\,v\,P_2\,u$, where P_1 and P_2 are paths of any length. Then we can replace the path P_1 with o since $u\,o\,v$ is a path. The cycle we obtain, $u\,o\,v\,P_2\,u$ is still simple because neither o nor v belongs to P_2. This simple cycle satisfies the property we were looking for. The configuration we described is illustrated in Figure 2. ■

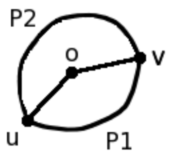

Figure 2: Simple cycle containing u, v, and o

BFCD Algorithm

```
for every vertex v in the graph
  mark[v] = false;
Q = empty_list();
Q.push_back(o);
while (!Q.is_empty()) {
  x = Q.pop_front();
  for every y, neighbor of x {
    if (!mark[y]) {
      mark[y] = true;
      pred[y] = x;
      Q.push_back(y);
    }
    else if (x != pred[y] && y != pred[x]) {
      cycle = empty_list();
      cycle.push_front(x);
      u = pred[x];
      while (u != o) {
        cycle.push_front(u);
        u = pred[u];
      }
      cycle.push_front(o);
      cycle.push_back(y);
      u = pred[y];
      while (u != o) {
        cycle.push_back(u);
        u = pred(u);
      }
      cycle.push_back(o);
      if (simple(cycle))
        add cycle to set;
    }
  }
}
```

Definition 2.3. *Let $o \in V$ be a particular vertex in the graph for which we want to extract the facet structure. Let $C_s(o)$ be the set of all the simple cycles containing o. Let $H_o = \{A_o, E_o, W_o\}$ be the weighted* **co-cyclic graph** *induced by the vertex o defined the following way:*

$$\begin{aligned} A_o &= \{v \in G \mid ov \in E\} & (3)\\ E_o &= \{vw \mid \exists c \ \in C_s(o), v \in c, w \in c\} & (4)\\ W_o(vw) &= min\{length(c) \mid c \in C_s(o)\} & (5) \end{aligned}$$

Thus, the set of vertices in the co-cyclic graph is the set of neighbors of o in the original graph. An edge is defined in the co-cyclic graph between two vertices u and v if they are co-cyclic. The weight of the edge is defined as the length of the shortest simple cycle containing the origin and the two vertices. Since the original graph is undirected, the co-cyclic graph is also undirected.

The idea of the co-cyclic graph is closely related to the *dual* graph [6, 18]. Dual graphs define a vertex for each facet of the graph, each of them being a simple cycle in the graph. An edge is drawn between two facets if they have an edge in common. A vertex in the dual graph corresponds to an edge in the co-cyclic graph. While the dual graphs only exist for planar graphs, co-cyclic graphs can be defined for any graph.

Theorem 2.4. *The relation defining the edges of H_o in Equation 4 is transitive.*

Proof. Let $y, v, w \in V$ such that y and w are co-cyclic, as well as w and v. Let us denote the two cycles starting from o and containing these vertices by $o\,v\,C_1\,w\,o$ and $o\,w\,C_2\,y\,o$, where C_1 and C_2 are paths of any length. Let us suppose that $v \notin C_2$ and $y \notin C_1$, otherwise we would already have found a simple cycle containing y, v, and o.

Then the cycle obtained by merging these two cycles, $o\,v\,C_1\,w\,C_2\,y\,o$ contains both y and v. If the cycle is not simple, any repeated vertex must be on the path $C_1\,v\,C_2$, since neither o, nor u or w are present on that path. Thus, we can remove subcycles included in this path until the cycle is simple. Figure 3 illustrates the configuration we described.

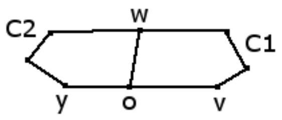

Figure 3: Two cycles that can be merged

Thus, we have shown that

$$yv \in E_o \ \wedge \ vw \in E_o \ \Rightarrow yw \in E_o \qquad \blacksquare \qquad (6)$$

The consequence of the transitivity for the co-cyclic graph H is that its connected components are complete subgraphs.

Example

Let us take the case of a graph with the structure of a hypercube and the origin O as shown in Figure 4 left. This graph is interesting from the point of view of our algorithm because it represents a four-dimensional geometric object in which there is no obvious simple cycle fan that surrounds each vertex.

There are four vertices in the graph adjacent to O and they are labeled A, B, C, D. Each of them is connected to all of the others by a simple cycle of length four representing a facet of the hypercube. The co-cyclic graph H_o for this particular vertex is shown in Figure 4 right.

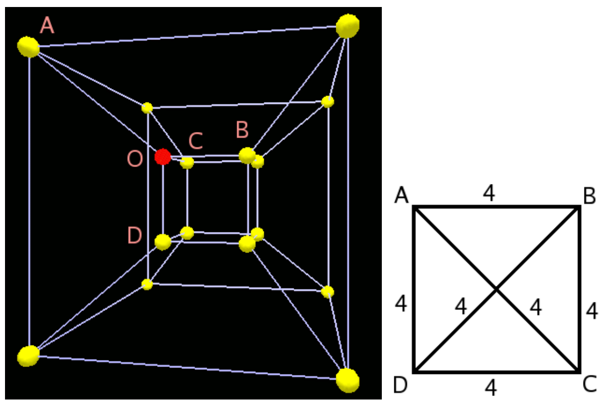

Figure 4: A hypercube, the vertices adjacent to O (left) and the induced co-cyclic graph (right)

Co-Cyclic Graph Closure

The cycle detection algorithm presented in Section 2.1 does not always detect the complete subspace of simple cycles $C_s(o)$. An intersection of edges in a vertex y as described in the algorithm generates a single simple cycle in the algorithm. Thus, the algorithm detects at most $m-(n-1)$ cycles, because it can only detect one cycle per edge. Moreover, for every vertex that it visits, at least one incident edge connects it to the origin for the first time and does not generate a cycle.

The number of simple cycles that o belongs to can be larger. o can have as many as $n-1$ neighbors, and thus a maximum of $(n-1)(n-2)$ incident cycles. Such an example would be a triangle fan as the one illustrated in Figure 5, that contains two simple cycles for every unordered pair of neighbors of o. If the graph is complete, then the BFCD may construct all of these cycles, otherwise the number of resulting cycles will be lower than the total number of existing cycles.

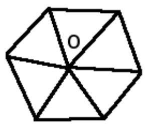

Figure 5: A triangle fan where o belongs to $(n-1)(n-2)$ simple cycles

The co-cyclic graph can be completed using the transitivity of the co-cyclic relation. Let us denote by B_o the partial co-cyclic graph built by BFCD. We will prove that the closure of B_o by transitivity is identical to H_o. For this, we need to prove that the connected components of B_o are the same as for H_o. This is equivalent to proving that for every two co-cyclic vertices that are neighbors of o, a path can be found connecting them in B_o.

Definition 2.5. *A path $u_0\ u_1\ \ldots\ u_k$ is called a* predecessor path *if $pred(u_i) = u_{i-1}$, $\forall i,\ 1 \le i \le k$. Such a path is defined by the predecessor function.*

We can remark that if two predecessor path intersect, then they must have a common beginning path. As soon as they diverge, they cannon intersect again since the predecessor of a vertex is unique. The predecessors build a spanning tree in the graph.

For any path P in the graph, let us denote by $\bar{P}$ the path obtained by reversing P. If $P = u_0\ u_1\ \ldots\ u_k$, then $\bar{P} = u_k\ u_{k-1}\ \ldots\ u_0$.

Lemma 2.6. *Let u and v be two co-cyclic neighbors of o, and x and y two other vertices in the graph G such that a simple cycle exists in the graph with the structure $o\,u\,P_1\,x\,y\,P_2\,vo$ and the property that both the paths $o\,u\,P_1\,x$ and $o\,v\,\bar{P}_2\,y$ are predecessor paths. Then the cycle is detected by BFCD, and thus, the edge uv exists in the graph B_o.*

Figure 6 shows the cycle described in the lemma.

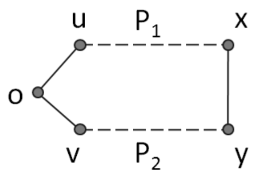

Figure 6: Cycle containing u and v in Lemma 2.6

Proof Since both the part of the cycle between o and x and the part between y and o are predecessor paths, and the path is simple, we can say that

$$y \neq pred(x), \quad x \neq pred(y)$$

This means that when the edge xy is processed, both vertices must have been visited before, otherwise the one that was not visited yet would have the other one as predecessor. At this point, BFCD must have constructed both predecessor paths to x and to y. So the edge xy is found to close a cycle.

We can remark that BFCD builds the cycles out of two predecessor paths and an extra edge. Thus, the cycle built in this case based on the edge xy will be the one containing u and v. Since both of these are neighbors of o, an edge will be added between them in the graph B_o. ■

The next theorem insures the completeness of the graph H_o after the transitivity closure.

Theorem 2.7. *For every two neighbors of o that are co-cyclic, a path connecting them exists in the graph B_o constructed by BFCD.*

Proof. Let us consider two neighbors of o, u and v, and a simple cycle connecting them in the graph G. Let us exclude from this cycle the largest continuous predecessor paths going in the directions of u and v respectively. We will use the strong induction over the remaining number of edges in this cycle to prove that u and v are connected in B_o.

More precisely, let us decompose this cycle into $o\,u\,P_1\,x\,P_3\,y\,P_2\,vo$ such that the paths from o to x and from o to y are predecessor paths, but $x\,P_3\,y$ is not. The maximality of the two predecessor paths means that x is not the predecessor of the first vertex in P_3, and y is not the predecessor of the last one.

Base case. If the length of P_3 is 0, then there is an edge between x and y, and the vertices u and v satisfy the condition in Lemma 2.6. Since in this case, an edge will exist between them in B_o, they are indeed connected.

Inductive step. Let us suppose that any two vertices for which the length of $P_3 \leq k$ are connected in B_o. We shall prove that they are connected for the length of $P_3 = k + 1$.

Let us denote by z the first vertex in the path P_3. This means that there is an edge in G between x and z, and since the cycle is simple,

$$z \neq pred(x), \quad x \neq pred(z).$$

In such a case, a predecessor path must exist from o to z. Let w be the first vertex after o in that path, and let us denote this path by $o\,w\,P_4\,z$. Figure 7 shows the structure of the vertices and paths involved in this part of the proof.

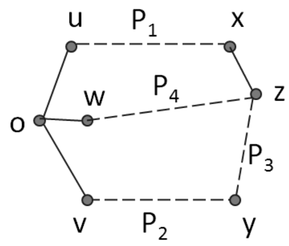

Figure 7: Cycle containing u and v and intermediate vertex w

If $u = w$, then by replacing the part $o\,u\,P_1\,x\,z$ in the cycle with this predecessor path, we have found a simple cycle containing u and v. In this cycle, the predecessor path from o to u expands up to the vertex z, such that the length of the remaining path in the cycle is at most k and could be even smaller than k. The inductive hypothesis tells us that in this case u and v are connected in the graph B_o.

If $w = v$, then the cycle $o\,w\,P_4\,z\,x\bar{P}_1\,u\,o$ satisfies the property in Lemma 2.7, thus w and v are connected.

If $u \neq w$ and $w \neq v$, then the vertices u and w satisfy the condition in Lemma 2.6, so the edge uw exists in B_o.

Furthermore, the vertices w and v are connected by the cycle $o\,w\,P_4\,z\,P_3 y\,P_2\,vo$. Based on the observation that predecessor paths cannot intersect if the start towards different neighbors of o, this cycle is also simple. The largest predecessor path from o to w continues at least up to the neighbor z. Thus, the length of the remaining path is at most k, and the inductive hypothesis tells us that the vertices w and v are connected by a path in B_o.

In this last case, since the edge uw is in B_o and w and v are connected in B_o, there exists a path connecting u and v in B_o. This concludes our proof. ■

The next theorem gives us a important property of the graph B_o constructed by BFCD and justifies the fact that in many cases, constructing the complete graph H_o by transitivity is not necessary.

Theorem 2.8. *For every neighbor of o that is co-cyclic with o, at least one cycle of minimal length containing it will be constructed by BFCD.*

Proof. Let us consider a vertex u that is a neighbor of o and co-cyclic with it. In particular, BFCD will find that $pred(u) = o$. Let us consider the shortest simple cycle containing both of these vertices. Let this cycle be $o\ u\ u_1\ u_2 \ldots\ u_k\ o$ having a length of $k + 2$, where $k > 0$ to avoid a trivial cycle. Since u_k is also a neighbor of o, we must have $pred(u_k) = o$.

For every $i \leq \lfloor k/2 \rfloor$, $d(o, u_i) = i + 1$ because if there existed any path from o to u_i that was shorter than the one going through u, we could replace the part of the cycle from u_i back to o with this path to obtain a shorter cycle containing o and u. This contradicts the minimality of the cycle.

As a second observation, u must be found on the predecessor path from o to each u_i, for $i < k/2$. If not, then we could also replace the part of the cycle from u_i back to o with the predecessor path to obtain a smaller cycle.

We must distinguish the cases where the cycle has an odd or an even length. Figure 8 shows the composition of the cycle in the two cases that we discuss further in the proof.

a. Let us consider that $k = 2j + 1$ first. If the path $o\ u\ u_1 \ldots\ u_j$ is not identical to the predecessor path from o to u_j , we can replace the intermediate vertices with this alternative path, since we know that u must also belong to it. Thus, if we identify $u = u_0$, we can safely assume that

$$pred(u_i) = u_{i-1},\ \forall i,\ 1 \leq i \leq j \tag{7}$$

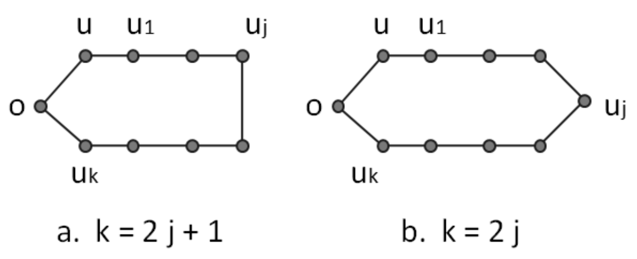

Figure 8: Minimal cycle containing o and u with an odd (a) and an even (b) number of vertices

A similar reasoning can be applied to make sure that the path on the other side of the cycle is also a predecessor path, such that

$$pred(u_i) = u_{i+1},\ \forall i, j < i < k \tag{8}$$

In this case the vertices u and u_k satisfy the property in Lemma 2.6. Thus, the edge uu_k must exist in the graph B_o, and the weight of this edge being defined by this cycle, it must be minimal.

b. In the case where $k = 2j$, by the same procedure as above, we can choose the vertices $u_1, u_2, \ldots, u_{j-1}$ such that this is a predecessor path:

$$pred(u_i) = u_{i-1},\ \forall i,\ 1 \le i < j \tag{9}$$

Similarly, we can make sure that the path on the other side of a cycle is also a predecessor path:

$$pred(u_i) = u_{i+1},\ \forall i,\ j < i < k \tag{10}$$

As for the vertex u_j itself, it is at equal distance from o on the path going through u and on the one going through u_k.

If $pred(u_j) = u_{j-1}$, then the cycle will be detected by BFCD at the moment where the edge $u_j\ u_{j+1}$ is processed.

If $pred(u_j) \neq u_{j-1}$, then we can choose the vertices u_{j+1} to u_k such that Equation 10 is still true, and also $pred(u_j) = u_{j+1}$. In this case, BFCD will detect the cycle at the moment where the edge $u_{j-1}\ u_j$ is processed.

In both cases the vertices u and u_k satisfy again the condition in Lemma 2.6, and so the edge uu_k will exist in the graph B_o. The way we have chosen the vertices to follow the predecessor paths insures that BFCD will construct this cycle after it detects it. ■

Theorem 2.8 insures that no vertex will be missing from the graph B_o built by BFCD, and that at least one incident edge is present for every such vertex. Theorem 2.7 insures that every vertex in B_o is connected by a path to every vertex that it should be connected to.

2.3 Cycle Structure

Starting from the assumption that we have constructed the co-cyclic graph H_o successfully, let us examine what the cycle structure in the vertex o entails in terms of the co-cyclic graph. For this step, each connected component of the graph will be treated separately. Let us consider for now that the graph H_o has a single complete connected component. There are two possible constraints to the solution defining the facet structure around o. The first one involves not using an edge more than twice. The second one involves using each outgoing edge from o and minimizing the total length of the considered cycles.

Definition 2.9. *A* **circuit solution** *for the facet problem is one where each outgoing edge from o is present in exactly two cycles in the structure and for which the total length of the cycles is minimal.*

The constraint that each outgoing edge should appear twice means that the subset of cycles constitutes a Hamiltonian circuit in the graph H_o, a simple cycle containing all the vertices. The constraint about minimizing the total length of the cycles translates in terms of the graph H_o as a Hamiltonian circuit minimizing the total cost. This means that the problem can be reduced to a symmetric instance of the Traveling Salesman Problem (TSP) in the co-cyclic graph.

TSP is NP-complete, but we have not shown a complete equivalence to this problem. An instance of our problem can be reduced to an instance of TSP, but the reverse is not true. In fact we can observe that the co-cyclic graph satisfies the triangular inequality. More precisely, if $v, w, y \in V_o$, then based on the proof of Theorem 2.4 concerning the transitivity of the co-cyclic property, the weight of the edges vw, wy and vy in the co-cyclic graph satisfy the inequality

$$W_o(vy) \leq W_o(vw) + W_o(wy) - 2 \tag{11}$$

There are many studies of TSP in the case where the graph has particular properties [11], especially Euclidean [2]. The major algorithm for the case where the triangle inequality is satisfied was introduced by Christofides in [4]. He proved that there exists a polynomial algorithm that produces a solution not worse than twice the optimal cost. His algorithm uses the idea of the minimum spanning tree.

For example, for the hypercube with co-cyclic graph presented in Figure 4, a circuit solution could be the one shown as a thick gray outline in Figure 9 left. The corresponding sequence of cycles in the original graph is shown in the same figure to the right. We show the 3D object in this figure using a different perspective that makes it easier to see the solution.

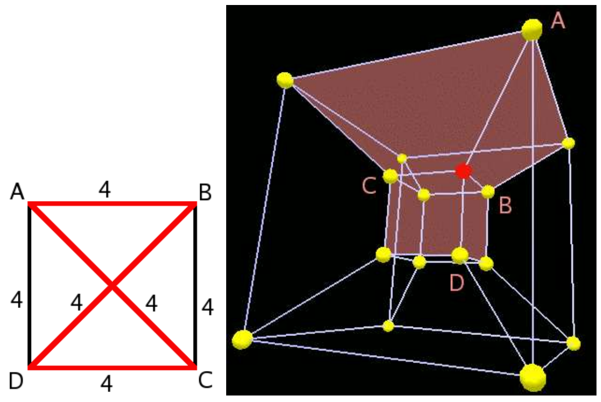

Figure 9: A circuit solution in the hypercube

Definition 2.10. *A* **spanning tree solution** *for the facet problem requires each outgoing edge to be present at least once and for the total length of the cycles to be minimal.*

The relaxation of the constraints of the circuit solution means that the object satisfying these properties in the graph H_o is a minimum spanning tree [17].

For example, for the hypercube and the co-cyclic graph presented in Figure 4, a spanning tree solution could be the one shown in gray in Figure 10 left; the corresponding sequence of cycles in the original graph is shown in the same figure to the right.

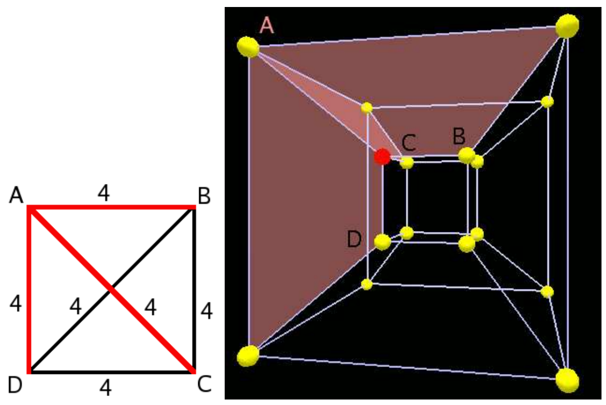

Figure 10: A spanning tree solution in the hypercube

3 Facet Definition and Detection

In this section we present the facet detection algorithm, starting with the existence and detection of simple cycles in the graph, followed by a selection and classification of the simple cycles intersecting in a particular vertex in the graph.

The definition of a facet is closely related to that of a feature. Basically a feature is usually composed of several facets, so facet detection can be seen as an intermediary step in feature detection.

Definition 3.1. *A* feature *is part of an object that is localized, meaningful, and detectable.*

Its definition is often subjective, making reference to what a human eye can recognize [12]. Examples include edges, corners, chains, line segments, parameterized curves, regions, surface patches (3D), closed polygons [8].

Definition 3.2. *A* facet *is usually defined as part of an object or of an image that is flat, planar, linear, quadratic, or cubic.*

Considering that a graph represents objects with straight edges and planar surfaces, a facet is a closed and planar polygon, preferably convex.

3.1 Facet Detection

Instead of focusing locally on each of the facets, our algorithm detects them by using the entire structure of cycles surrounding each vertex in the graph. Starting from the assumption that the graph represents a solid surface without holes, we aim to construct a fan of minimal polygons centered in each vertex such that they can have a local planar representation without overlapping. For this purpose, every vertex in the graph that is adjacent to the origin, must belong to two and only two cycles of the fan.

The algorithm Co-Cyclic Facet Detection (CCFD) shown bellow starts by detecting all the simple cycles intersecting in o, then sorting them by length. After that it follows the idea of a minimum spanning tree, adapted to construct a Hamiltonian circuit.

Algorithm CCFD

```
for every vertex o in the graph {
  Bo = BFCD(o); // set of simple cycles
  optional: Ho = closure of Bo by transitivity;
  sort Bo or Ho by cost;
  Fo = empty_set();
  for (every cycle c in Bo and Ho and
      while size(Fo) < degree(o)) {
    v, w = the two vertices in c
           adjacent to o;
    if ((each of v and w appear each in
         at most one other cycle in Fo)
         and ((c does not close a cycle of cyles in Fo)
               or c is the last cycle we need))
      add c to Fo;
  }
}
```

Complexity

If n is the number of vertices and m the number of edges, then in the worst case BFCD will have a complexity of $m = O(n^2)$. The operation of merging two cycles and eliminating sub-cycles is linear over n. Computing the closure of the co-cyclic graph can be done with a greedy algorithm of complexity in the worst case $O(n(n(n-1)/2 - m)^2) = O(n^5)$. By computing only the cycles necessary to close the circuit at the end of the algorithm, we can reduce this step to n^3. This can be done by a simple search of a path in the co-cyclic graph. Sorting the cycles by length is of a complexity of the order $O(m \log m) = O(n^2 \log n)$.

We should recall here that Theorem 2.8 indicates that a cycle of minimal length will be found by BFCD for every neighbor of o. This means that an edge of minimal weight will be present for every vertex even without the closure of the graph B_o. These edges belong to the beginning of the set of edges in H_o as sorted by weight. Thus, in most cases the algorithm is likely to complete a Hamiltonian circuit of minimal cost without the need to complete the co-cyclic graph after the run of BFCD.

The most costly operation in the algorithm above is checking that the cycle c does not close a cycle of cycles. This can be done efficiently by storing the neighbors of the vertex o in a union/find data structure, in which case its complexity will be $O(\log n)$ at each step, making the total complexity $O(m \log n) = O(n^2 \log n)$.

In conclusion, our algorithm has a complexity of $O(n^2 \log n)$ without closure of the co-cyclic graph, and of $O(n^3)$ with it.

3.2 Examples

We present here some examples of results of our algorithm for a variety of graph configurations. In each of the following figures the origin vertex is shown in green, while the cycles in the structure identified by the algorithm are shown in shades of red. This provides a visually distinction between the cycles composing the structure.

First, Figure 11 shows the hypercube configuration that we discussed earlier in this paper. We can see in the left image that the algorithm found the predicted cycle structure for the vertex we analyzed before. It also found an equivalent correct solution for a vertex on the exterior cube, shown in the image to the right.

Figure 12 shows the cycle structure for a regular dense fan (left), and for one of the vertices on the border of the fan (right) in an ellipsoid graph. Figure 13 shows the solution found by our algorithm on a triangular

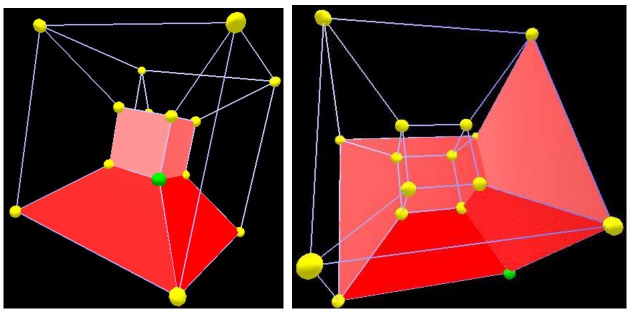

Figure 11: Solutions found by the algorithm in a hypercube

grid both for a vertex on the outer border of the grid (left) and for a vertex inside the grid (right). Figure 14 shows the structure for vertices that are part of the regular grid of a torus graph (left) and an ellipsoid (right), both composed of regular quadrilateral grids. For all these graphs the algorithm was able to identify the cycle structure precisely.

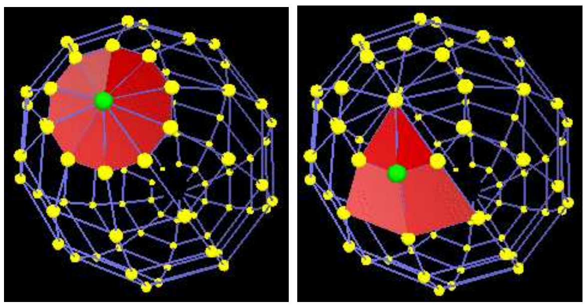

Figure 12: A dense fan on an ellipsoid graph

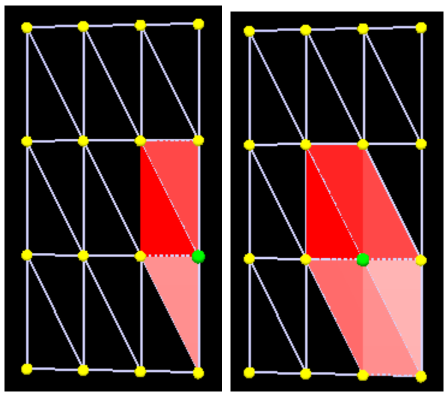

Figure 13: A vertex on the border of a grid and inside a grid

An important question related to the structure of the graph is the planarity of the graph. Figures 15 and 16 show the local structure built by the algorithm for the known non planar graphs K_5 and $K_{3,3}$.

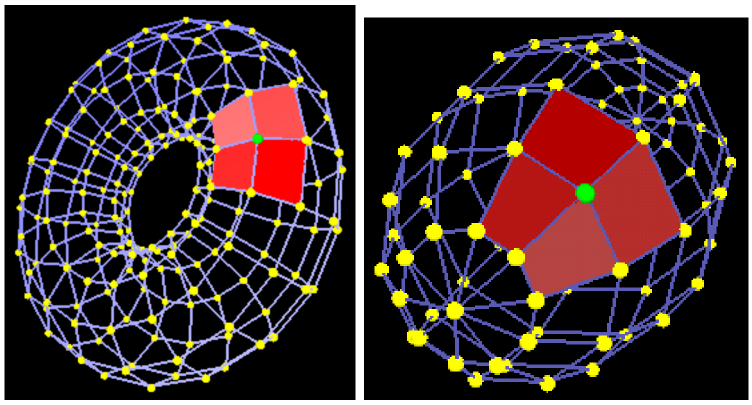

Figure 14: A vertex on regular grid in a torus graph and in an ellipsoid

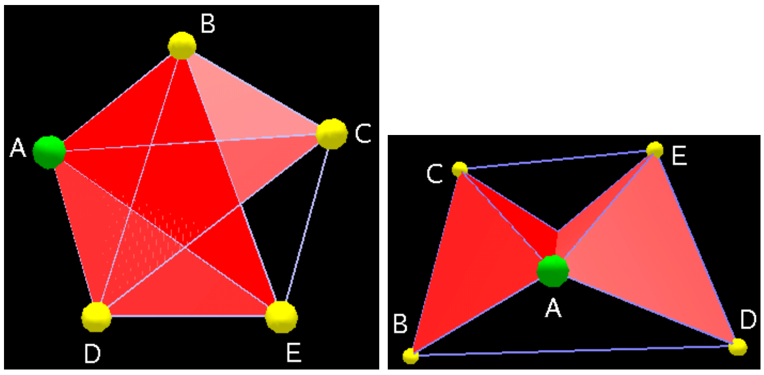

Figure 15: The vertex structure for $K5$, two layouts

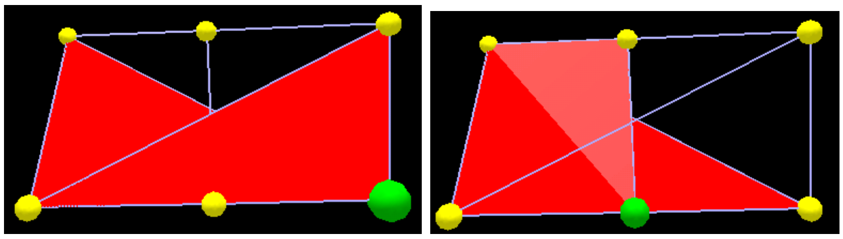

Figure 16: The vertex structure for $K_{3,3}$ for two vertices

In the case of K_5 we show the graph in a planar representation (left), and in a three dimensional representation (right) that emphasizes the solution better. Both images show the same solution, composed of the cycles $ABCA$, $ACDA$, $ADEA$, $AEBA$, forming a Hamiltonian circuit in the co-cyclic graph of A.

In the case of the graph $K_{3,3}$, we used a geometric representation minimizing the number of edge crossings. We show two solutions, one on the outside of the graph (left) and one in the center of the graph (right). They look different, but they are mathematically equivalent. In both cases the solution consists of two cycles of length four: $CEAFC$, $CEADC$ on the left, and $ECDAE$, $EBDAE$ on the right. In these cases the co-cyclic graph built by BFCD was incomplete, and the circuit solution after the transitivity completion would contain the cycles $CFBDC$ and $ECFBC$ respectively.

There is a difference between finding the mathematical object in the definition of the circuit solution presented in Section 2.3 and the intuitive ideas of facet and feature introduced in the current section. In most cases, CCFD builds a cycle structure that is appropriate for what we perceive as the structure of the object, but fine-tuning of the heuristic can sometimes lead to better results.

In particular, the closure of the co-cyclic graph based on the transitivity of the co-cyclic relation can have both positive and negative impact on the solution. For example, in the case of the vertex on the border of the grid in Figure 13 left, the BFCD algorithm failed to capture all the simple cycles around the origin vertex (hollow circle in the figure). As a result, the circuit of cycles was not completed. It is a Hamiltonian path and not a circuit. For this particular case, this emphasizes the structure better than a circuit because the vertex is on a border and has a topologically closed neighborhood.

For more complex local structures in the graph, not closing the co-cyclic graph B_o can lead to incomplete solutions. Such an example is shown in Figure 17.

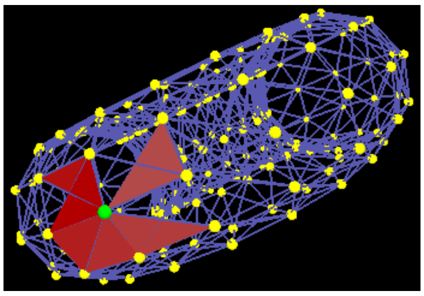

Figure 17: An incomplete solution without closure of the co-cycle graph

Last, we have shown in Section 2 that the problem can be expressed as an instance of the Traveling Salesman in the co-cyclic graph. TSP is NP-complete, and even in the case where the graph satisfies the triangle inequality, as in our case, the best polynomial algorithm cannot guarantee an optimal solution, but only one that is no more than a given percentage more costly than the optimal one. Figure 18 shows an example where the solution found by the algorithm is not optimal.

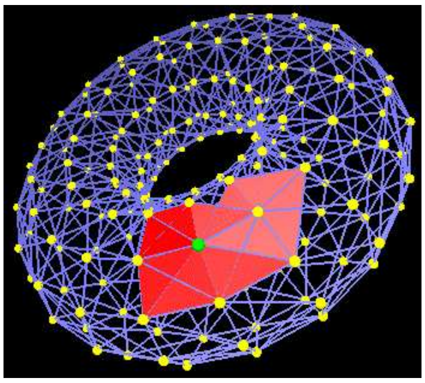

Figure 18: A non-optimal solution on a torus graph

3.3 Applications

We have originally designed this algorithm for a practical application related to graph drawing [16] and some recent results were presented in [15]. The problem we are trying to solve is building a graph layout or graph drawing [10] that is consistent with the weights in the graph, and if possible, that presents some geometric properties like maximizing the surface. We utilize a combination of force-based algorithms and genetic algorithms to solve this problem. For this purpose we have used the CCFD algorithm to provide a better estimation for some of the measures for the geometric properties of the graph layout.

Surface estimation. The local structure of the graph in each vertex identifies the facets of the graph and reduces the problem of computing the surface of the graph to the sum of the area of all the cycles extracted by CCFD. Each of them is then a simple polygon whose surface is easy to compute. A simple rule can avoid adding the surface of a cycle more than once, which consist of only considering the cycle for the vertex with the smallest index in the cycle. The use of CCFD has improved the layouts obtained by using this measure. Figure 19 compares two graph layouts obtained first using an estimation of the surface without CCFD (left) and by using CCFD as a preliminary step in the estimation of the graph surface (right).

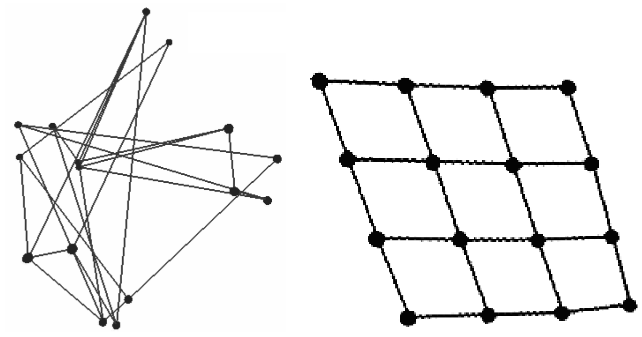

Figure 19: Layout obtained with a surface measure without CCFD (left) and with CCFD (right) for a regular 4x4 grid

Angle uniformity. A second measure of the aesthetic quality of the graph layout is related to the uniformity of the angles. As the graph might be composed of a variety of cycles with different lengths, the local structure of cycles constructed by CCFD provides a way to identify which angles are significant around every vertex of the graph and what their ideal value should be. Thus, this measure provides an optimal value when the angles in each of the basic cycles built by CCFD are equal to the angle of a regular polygon with a number of vertices equal to the length of the cycle. Figure 20 compares two graph layouts obtained first using an estimation of the angle uniformity without CCFD (left) and by using CCFD to determine the significant polygons for the angle uniformity (right).

Both Figure 19 and Figure 20 show that considerable progress can be made towards drawing an appropriate layout by using CCFD to determine the local structure of the graph first.

4 Conclusion

In this paper we presented an algorithm to detect facets in a graph and emphasize the local structure of the graph around a particular vertex for the purpose of visualization.

Section 2 presented a cycle detection algorithm BFCD and introduced the notion of induced co-cyclic graph. We showed that our problem can be reduced to an instance of the Traveling Salesman in the co-cyclic

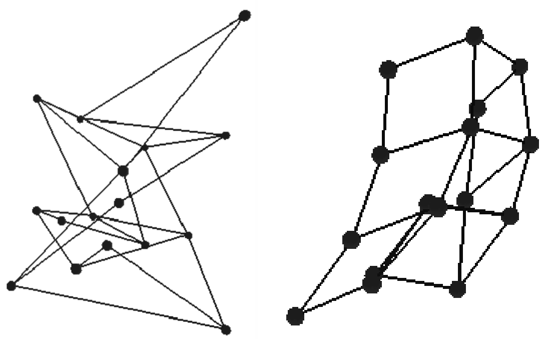
Figure 20: Layout obtained with an angle uniformity measure without CCFD (left) and with CCFD (right) for a regular 4x4 grid

graph. We have also analyzed some important properties of the cycle-detection algorithm that make it complete.

Section 3 introduced the facet detection algorithm inspired from the method constructing the minimum spanning tree and discussed its complexity. We have shown several examples of solutions found by the algorithm and further discussed its current applications to a graph drawing problem.

Our method detected the correct solution for most of the graphs that we examined and is of polynomial complexity. It is the most efficient when the graph is a representation of a regular mesh extracted from the surface of an object.

References

[1] N. Amenta and Y.J. Kil. Defining point-set surfaces. In *Proceedings of ACM SIGGRAPH*, pages 264–230, 2004.

[2] S. Arora. Polynomial time approximation schemes for euclidean tsp and other geometric problems. In *Proceedings of the 37th IEEE FOCS*, pages 2–11, 1996.

[3] S. Azernikov and A. Fischer. Efficient surface reconstruction method for distributed cad. *Computer-Aided Design*, 36(9):799–808, 2004.

[4] N. Christofides. Worst case analysis of an algorithm for the traveling salesman problem. Technical Report 388, Carnegie Mellon University, Graduate School of Industrial Administration, 1976.

[5] C. D'Souza, K. Sivayoganathan, D. Al-Dabass, and V. Balendran. Simulation of object recognition by a robot. In *Proceedings of the UKSIM Workshop, SIMULATION '99*, pages 23–26, UCL, London, 1999.

[6] A. Gibbons. *Algorithmic Graph Theory.* Cambridge University Press, 1999.

[7] C. C. Gotlieb and D. C. Corneil. Algorithms for finding a fundamental set of cycles for an undirected linear graph. *Communications of the ACM*, 10(12):780–783, 1967.

[8] R. Haralick and L. Shapiro. *Computer and Robot Vision.* Addison-Wesley, 1993.

[9] Q. Iqbal and J. K. Aggarwal. Image retrieval via isotropic and anisotropic mappings. In *IAPR Workshop on Pattern Recognition in Information Systems (PRIS 2001)*, pages 34–49, Setubal, Portugal, July 6-8 2001.

[10] G. Di Battista, P. Eades, R. Tamassia, and I.G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* An Alan R. Apt Book. Prentice Hall, Upper Saddle River, NJ, 1999.

[11] M. Karpinski P. Berman. 8/7-approximation algorithm for (1,2)-tsp. In *Proceedings of the 17th ACM-SIAM SODA*, pages 641–648, 2006.

[12] M. Pauly, R. Keiser, and M. Gross. Multi-scale feature extraction on point-sampled surfaces. *Computer Graphics Forum*, 22(3):281–290, 2003.

[13] A. Shahrokni, L. Vacchetti, V. Lepetit, and P. Fua. Polyhedral object detection and pose estimation for augmentedreality applications. In *Proceedings of Computer Animation*, pages 65–69, Geneva, Switzerland, 2002.

[14] M.C. Shin, D.B. Goldgof, and K.W. Nikiforou and. Comparison of edge detection algorithms using a structure frommotion task. *IEEE Transactions on Systems, Man and Cybernetics*, 31(4):589–601, 2001.

[15] D. Vrajitoru. Hybrid multiobjective optimization genetic algorithms for graph drawing. In D. Thierens et al., editor, *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'07)*, page 912, University College London, UK, 2007.

[16] D. Vrajitoru and B. El-Gamil. Genetic algorithms for graph layouts with geometric constraints. In *Proceedings of the IASTED Conference on Computational Intelligence (CI'06)*, pages 64–69, San Francisco, CA, November 20-22 2006. IASTED.

[17] M.A. Weiss. *Data Structures & Algorithm Analysis in C++.* Addison Wesley Longman, 1999.

[18] R.J. Wilson. *Introduction to Graph Theory.* Prentice Hall, 1996.

[19] C. Zhang and T. Chen. Efficient feature extraction for 2d/3d objects in mesh representation. In *Proceedings of IEEE International Conference on Image Processing*, pages 935–938, 2001.

[20] S. Zheng, J. Tian, and J. Liu. Efficient facet-based edge detection approach. *Optical Engineering*, 44(4):47202–47211, 2005.