Computer-Aided Design 42 (2010) 279–309

Contents lists available at ScienceDirect

Computer-Aided Design

journal homepage: www.elsevier.com/locate/cad

A new algorithm for finding faces in wireframes

Peter A.C. Varley *, Pedro P. Company

Department of Mechanical Engineering and Construction, Universitat Jaume I, E-12071, Castellon, Spain

ARTICLE INFO

Article history:
Received 17 November 2008
Accepted 30 November 2009

Keywords:
Wireframe
Face identification
Line-drawing interpretation
Visual perception

ABSTRACT

The problem of identifying the topology implied by wireframe drawings of polyhedral objects requires the identification of face loops, loops of edges which correspond to a face in the object the drawing portrays.

In this paper, we survey the advantages and limitations of known approaches, and present and discuss test results which illustrate the successes and failures of a currently popular approach based on Dijkstra's Algorithm. We conclude that the root cause of many failure cases is that the underlying algorithm assumes that the cost of traversing an edge is fixed.

We propose a new polynomial-order algorithm for finding faces in wireframes. This algorithm could be adapted to any graph-theoretical least-cost circuit problem where the cost of traversing an edge is not fixed but context-dependent.

1. Introduction

1.1. Motivation

Our area of interest is the process of engineering design, and in particular the representation of engineering objects by *sketches* and *drawings* (see the subsection on Terminology below). As data representations, sketches and drawings are useful and commonplace. Design engineers commonly draw concept sketches both when creating their ideas and when communicating their ideas to others. Drawings are the medium used when the designers of engineering components transmit new designs to manufacturers, and are also the medium used when storing and archiving existing designs.

In view of the successful use of sketches and drawings to communicate between one human and another, across boundaries of language, location and time, it is reasonable to ask: can we also use sketches and drawings to communicate between a human and a computer? Can we enter a sketch or a drawing into a computer with confidence that the computer will be able to construct a 3D solid model of the object portrayed?

In the last five decades, there have been many investigations performed in the pursuit of this goal, some addressing the problem as a whole and some addressing particularly troublesome subproblems. For the historical background, we refer the reader to Company et al. [1], which traces the evolution of this field from the particular problem of *geometrical reconstruction* to the more extensive user-oriented field of *sketch-based modelling*.

The major problem to be solved before we can interpret a wireframe line drawing as a polyhedron is that of identification of *face loops*: loops of vertices and edges which correspond to faces of the object. We define this more formally in Section 1.5.

In this paper, we present a new algorithm which avoids the deficiencies of the existing algorithms which we surveyed. We then discuss practical details of how our new algorithm may be implemented, and present results (a) comparing it with a favourite existing method based on Dijkstra's Algorithm and (b) addressing the question of whether face identification should precede or follow inflation.

1.2. Terminology

We distinguish between freehand *sketches* and *line drawings*. A *sketch* is a graphical representation made by hand. It may include shading, repeated lines for emphasis, and any other embellishments which help to convey the "flavour" of the object. A *line drawing* is made using tools (e.g. ruler, compass, computer). Its components are 2D geometric objects such as straight line segments and arcs, in one-to-one correspondence with components of the portrayed object.

In both cases we also distinguish between (a) *natural* sketches and line drawings, which show only the part of the portrayed object visible from the chosen viewpoint, and (b) *wireframe* sketches and line drawings, which show all edges of the portrayed object, visible or not. The drawings in Fig. 1 are examples of wireframe line drawings.

A *vertex* is *trihedral* if it meets exactly three edges and three faces. A polyhedron is trihedral if all of its vertices are trihedral. Similarly, a vertex is *tetrahedral (pentahedral, hexahedral, …)* if it meets no more than four (five, six …) faces, and a polyhedron is tetrahedral (pentahedral, hexahedral …) if its highest-valency vertex is tetrahedral (pentahedral, hexahedral …).

* Corresponding author.
E-mail addresses: varley@emc.uji.es, pacvarley@uk2.net (P.A.C. Varley), pcompany@emc.uji.es (P.P. Company).

0010-4485/$ – see front matter © 2010 Published by Elsevier Ltd
doi:10.1016/j.cad.2009.11.008

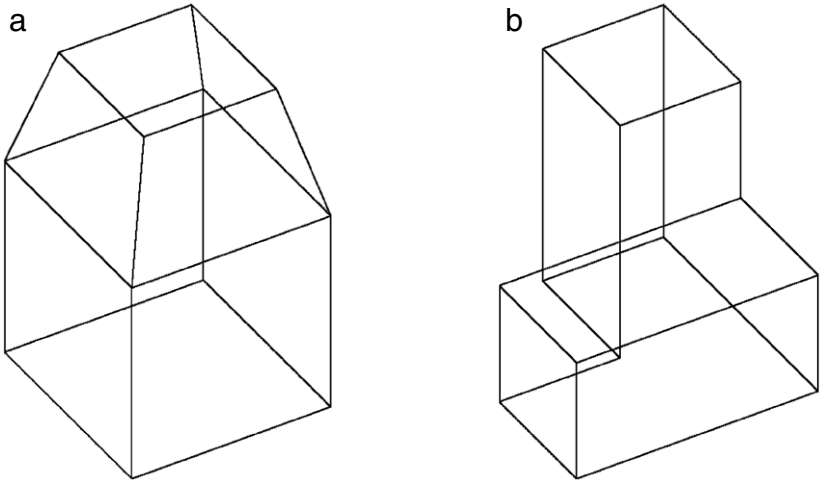

Fig. 1. Wireframe line drawings.

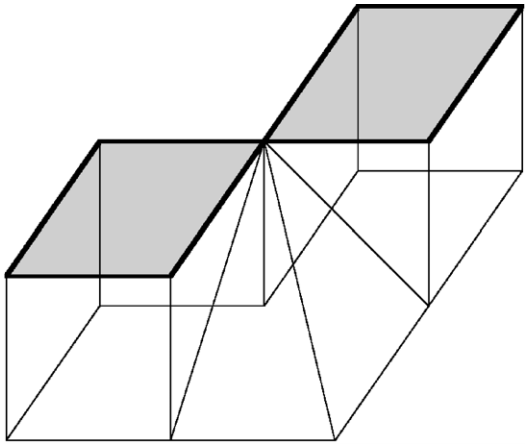

Fig. 2. Two faces, not one.

An *edge* joins two vertices and meets two faces. The formal definition of edge requires care, as discussed in Section 1.4.

A *face* is an alternating cycle of vertices and edges in which no vertex or edge appears twice. Using this definition, the highlighted part of Fig. 2 shows two coplanar faces, not a single face. In discussing polyhedra, all vertices and edges of a face must be coplanar--multipart-boundaryContent-Type: application/pdfRange: bytes=1396965454--1555806643/13322882 in 3D; this restriction can be re

A polyhedron is a *normalon* if all edges and face normals are aligned with one of three main perpendicular axes. It is a *semi-normalon* if removal of unaligned edges does not result in the loss of any vertices or any connected subgraph becoming disconnected.

1.3. Structure of paper

In this paper, we consider specifically the automated production of solid models from *wireframe line drawings* which show all *edges* of *manifold polyhedral objects* when viewed from a *general position* (no coincident edges or vertices). These assumptions guarantee that each edge joins two *vertices*. Systems for interpreting natural line drawings (e.g. [2,3]) and systems for converting free-hand sketches to line drawings (e.g. [4]) are not discussed.

In the original study of Markowsky and Wesley [5], this was regarded as a geometric problem: to find loops of coplanar vertices. Later developments have tended to treat it more as a topological problem, considering only the connectivity of the vertex–edge graph and paying less attention to its geometric implications.

Section 1.4 proposes a more formal definition of *edge*. Using this definition, Section 1.5 gives a more formal problem statement.

Section 2 illustrates some issues which make the problem non-trivial.

Section 3 expands on the history of the face identification problem, presenting various ideas chronologically, and also discusses an open question: should face identification use only the 2D information of the original drawing, or should it follow *inflation* (the process of adding depth information) and also use the deduced 3D information?

Section 4 describes in some detail what may currently be considered the most popular approach: use of Dijkstra's Algorithm to process the vertex–edge graph, and encapsulating geometric considerations in the *path length* (the penalty assigned when traversing an edge).

Sections 5 and 6 present our new algorithm, which allows for context-dependent cost functions. Section 5 describes the algorithm conceptually, and Section 6 gives a detailed description of a practical implementation.

Section 7 presents our test data.

Section 8 presents our test results, applying the implementations described in Sections 5 and 6 to the test data presented in Section 7.

Section 9 presents our conclusions and suggests paths for future work.

1.4. Edges

We have already defined that each line in the drawing portrays one edge of the object. To this, we may add that lines do not come in different "flavours" — there are, for example, no dotted lines representing occluded edges. This definition also rules out the idea, used for example by Ganter and Uicker [6], Courter and Brewer [7] and Liu, Lee and Cham [8], of connecting different subgraphs such as those in Fig. 3 with "scaffolding lines".

There remains one further decision, one which has both theoretical and practical aspects. What, precisely, is an *edge*?

Consider Fig. 4. Are there two or four highlighted edges? In practical terms, should the highlighted edges be represented as single lines in the input, or should they be broken to indicate that there is a vertex where they cross?

The natural way to draw these is as single continuous lines, but current approaches require input data in which there is an explicit intermediate vertex where these lines cross. Can such intermediate vertices be detected automatically? The problem appears minor, but remains open, and for the present we assume the existence of a preprocessing stage which converts the most natural input format into the format required by our algorithms.

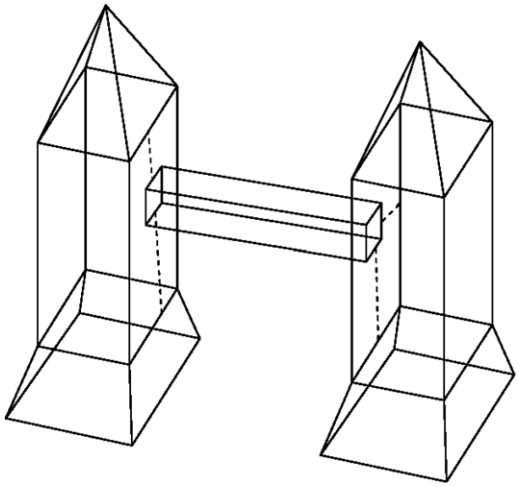

Fig. 3. Scaffolding lines [8].

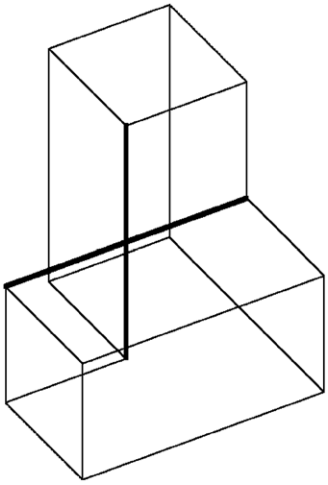

Fig. 4. Vertex where edges cross?.

In order that our formal terminology remains in step with our discussion, *edge* is defined such that the highlighted part of Fig. 4 shows four *edges* meeting at a tetrahedral *vertex*.

1.5. Formal problem statement

Our input data comprises is partly *topological* (a *undirected* graph $G(V, E)$) and partly *geometric* ((x, y) coordinates for each vertex of the graph). The graph is subject to the restrictions (i) that no edge connects a vertex to itself and (ii) no two edges join the same two vertices.

Formally, the problem is best expressed in terms of *half-edges*, where a half-edge is an *arrow* of the *directed* graph $G'(V, A)$ which contains arrows $\{a, b\}$ and $\{b, a\}$ if and only if the original undirected graph $G(V, E)$ contains the edge $\{a, b\}$.

A *cycle* is a closed path of half-edges.

Our first objective is to obtain a set of cycles in which each half-edge is included in exactly one cycle. Additionally, a topology must be geometrically realisable: it must be possible to select vertex z-coordinates such that each cycle is planar (in practice, it is necessary to specify "planar within a tolerance" rather than "exactly planar").

Where only one valid set of cycles exists, our objective is simply to find it. Where more than one valid set of cycles exists, our objective is to find the one which matches the interpretation of the drawing which a human interpreter would consider to be the most plausible. In the case of *degenerate* drawings with several equally-plausible interpretations, our objective is to find any one of them.

Note that we assume that at least one valid set of cycles exists. We are not interested in the related but distinct problem of determining whether or not any valid set of cycles exists. This separate subproblem has a literature of its own—see, for example, Sugihara [9] for founding work on this problem, and Heyden [10], Li [11] and Li, Zhao and Chen [12] for recent progress.

2. Issues

This section illustrates a number of issues which combine to make the face identification problem non-trivial.

2.1. Necker Reversal

The first phenomenon associated with wireframes to be studied was *Necker Reversal* [13]. See Fig. 5. Of the two central vertices in a 2D drawing of a cube, one corresponds to the 3D vertex nearest the observer, and the other to the 3D vertex furthest from the observer. But which is which?

In truth, since human observers can see both interpretations, it does not matter which of the two interpretations an automated approach chooses, as long as the choice is consistent throughout the object. Necker Reversal is not in itself a problem.

2.2. Internal faces

One problem which must be avoided is following a loop of edges which corresponds to an *internal face*. See Fig. 6, in which there is a loop of four edges around the middle of the object which does not correspond to any face of the object.

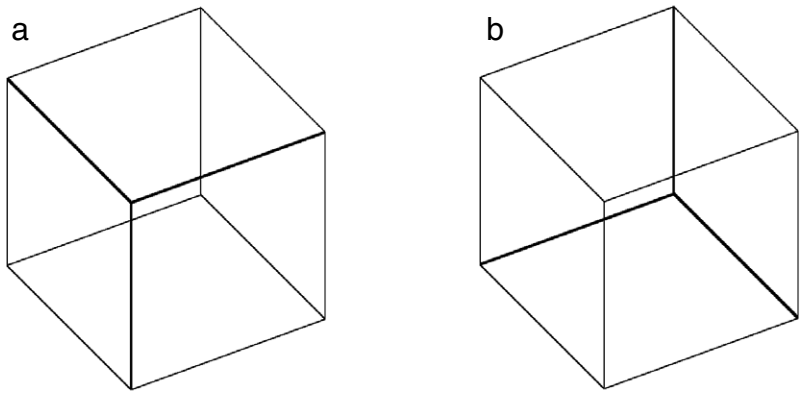

Fig. 5. Necker Reversal.

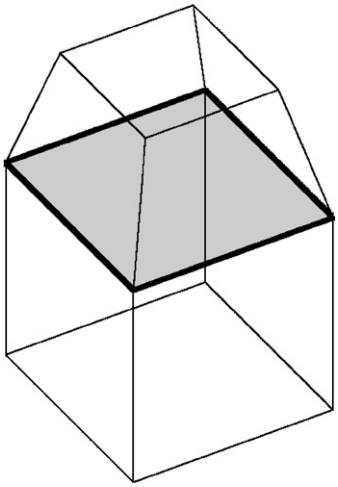

Fig. 6. Internal face.

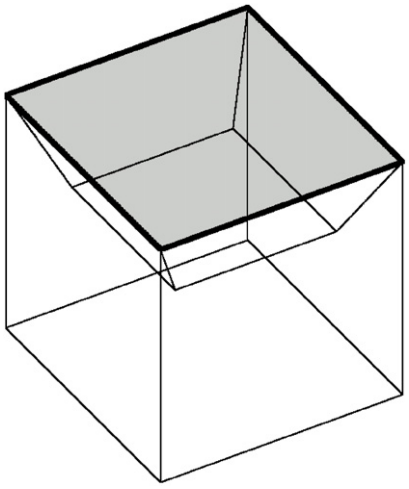

Fig. 7. Also an internal face.

Note that the term *internal face* is historical (introduced in [8]), and refers to the vertex–edge graph, not the geometry. Despite appearances, the highlighted cycle in Fig. 7 is also an *internal face*—compare with Fig. 6, which has the same vertex–edge graph.

Although internal faces appear in cycles containing non-trihedral vertices, not all non-trihedral vertices lead to internal faces. For example, the right-hand drawing of Fig. 1 does not. It is only when an entire loop of non-trihedral vertices exists that caution is required.

Cooper [14] provides a rigorous proof that trihedral wireframes do not contain internal faces.

We would go further, and assert that any loop corresponding to an internal face may contain only (i) non-trihedral vertices and (ii) trihedral vertices which lie on a non-simple planar subgraph.

Clearly, at trihedral vertices, all three possible pairs of edges are included in exactly one of the three faces bounded by the vertex. At (for example) a 4-hedral vertex, there are six possible pairs of edges, but only four faces, so two of the possible pairs of edges do not correspond to external faces.

The other potential source of loops which do not correspond to external faces is non-simple planar subgraphs. An example of trihedral vertices lying on a non-simple planar subgraph is shown in Fig. 6bis. In this case, although there are three possible circuits of this subgraph, only one of them can be a face.

Thus, if a trihedral vertex lies only on simple planar subgraphs, it cannot be part of an internal face.

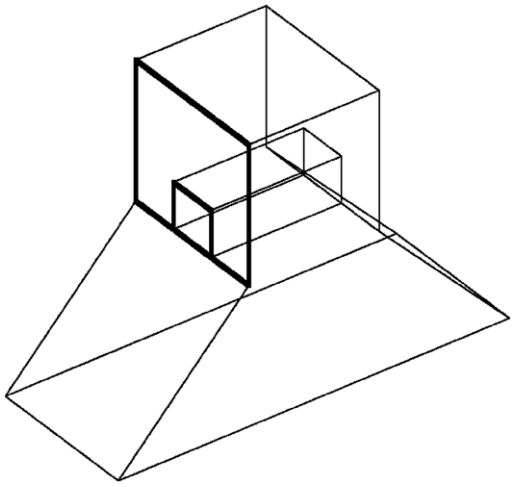

Fig. 8. Multiple circuits in a planar edge subgraph.

Note that this observation can only be used after inflation, as any approach which determines faces directly from 2D data has no knowledge of planar subgraphs (Fig. 8).

At a non-trihedral vertex, in addition to pairs of edges which correspond to true faces, there are the following possibilities:

- (i) a pair of edges which is not part of any planar cycle of edges (e.g. the left-hand drawing in Fig. 9)
- (ii) a pair of edges which are part of two distinct coplanar cycles of edges (e.g. the central drawing of Fig. 9)
- (iii) a pair of edges which is part of a planar cycle of edges which corresponds to an internal face (e.g. the right-hand drawing in Fig. 9).

Clearly, this list is exhaustive: if the pair of edges is part of a

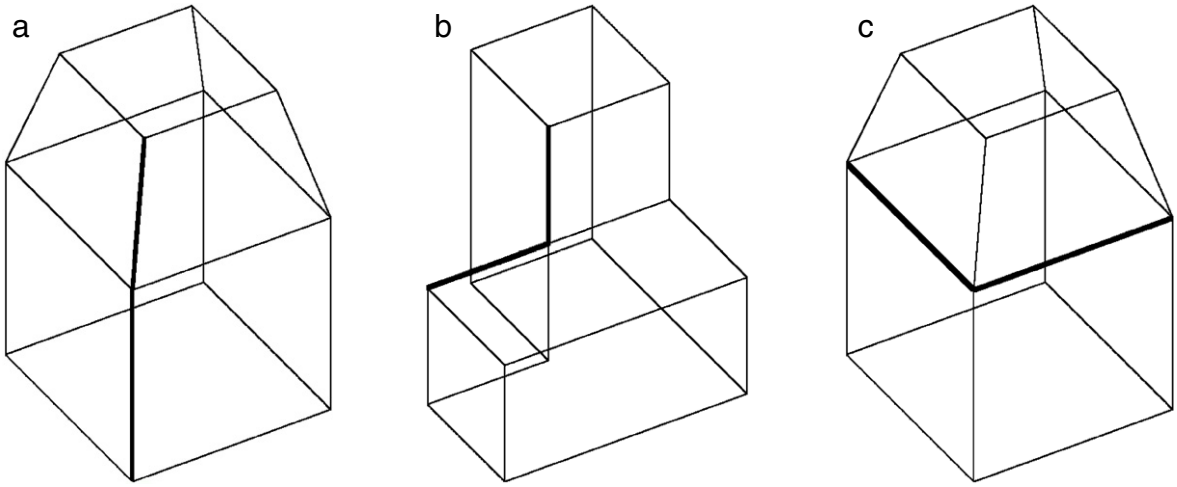

Fig. 9. Edge pairs not in true faces.

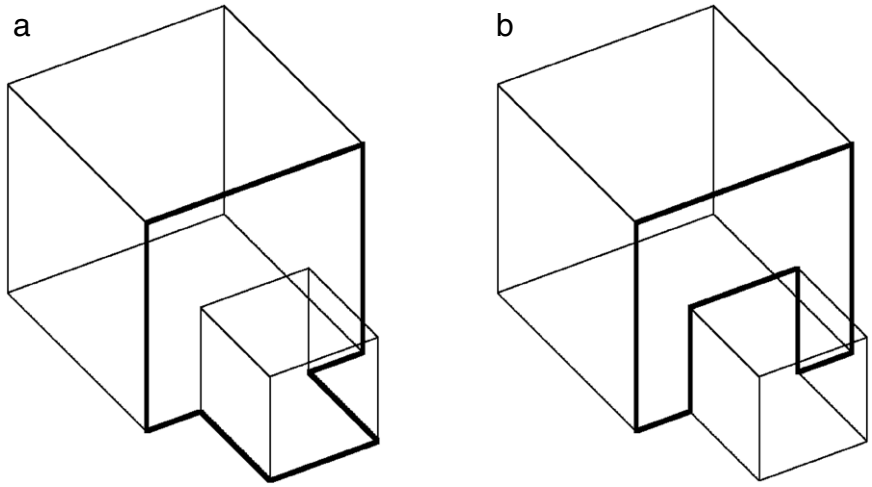

Fig. 10. Face loops: Wrong and right choices.

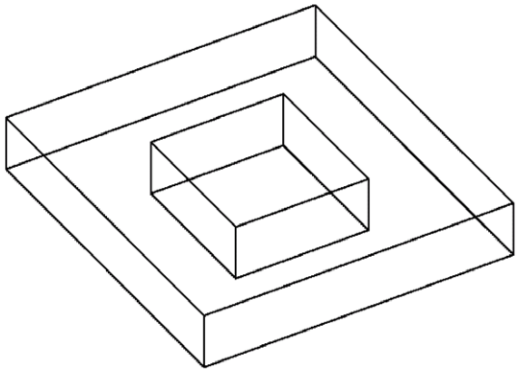

Fig. 11. Object with distinct subgraphs.

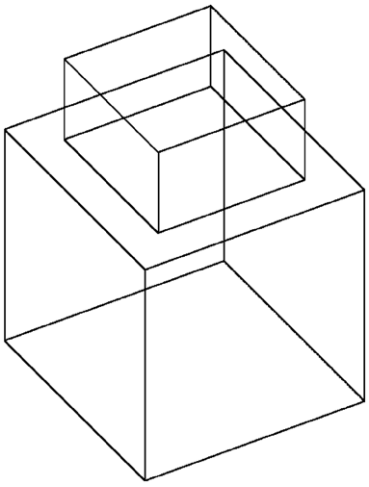

Fig. 12. Object with boss.

2.5. Hole loops: Alignment

Even in those cases such as Fig. 12, where there are two distinct subgraphs but even so occlusion helps humans to identify a single preferred interpretation, there is a further problem to be solved. Although Necker Reversal and face loop direction (clockwise/anticlockwise) are not in themselves problems (for both, an arbitrary choice between the two alternatives is acceptable) it is necessary to ensure that the same choices are made for both subgraphs, in order that the loops of edges in the two subgraphs can be made coplanar and cofacial.

In this paper, we do not investigate here the problem of how to identify which loops of edges in different subgraphs correspond to cofacial loops in the portrayed object. See Piquer et al. [17] for a recent investigation into this.

2.6. Non-manifold objects

In this paper, we deal only with manifold objects. This restriction can be justified on the grounds that engineering objects made by physical manufacturing processes are manifold. Further, we assume, rather than check, that the wireframe represents a manifold polyhedron.

3. Historical background

3.1. Introduction

In this paper, we consider specifically the automated production of solid models from *wireframe line drawings* which show all *edges* of *manifold polyhedral objects* when viewed from a *general position*.

This section presents a historical survey of face detection methods in semi-chronological order: similar approaches are grouped together, with the groups being described in the order in which the general approach was first used. In addition, there are two questions of particular interest concerned with general approaches rather than specific details:

- Should a face identification algorithm be primarily geometric or primarily graph-based?
- Should face identification precede or follow inflation?

Clearly, for algorithms which are primarily geometric, face identification must follow inflation, but for algorithms which are primarily graph-based, one can use either 2D or 3D information (or neither) in support. On examining the history of the problem we can find several advocates of each of the three reasonable approaches (and even a few advocates of *entirely* graph-based approaches).

Naturally, it is possible to develop more complex approaches. For example, face identification and inflation could proceed in parallel, with tentative information from each process helping to drive the other. We have no reported example of this at present, but it may be worth noting that such an approach has been used with some successful for *labelling* and inflating natural line drawings (Varley, Martin and Suzuki [18]).

3.2. Geometric approaches

The first algorithm developed for identifying face loops from wireframes was that of Idesawa [19], as a single stage of a multi-stage process for interpreting engineering drawings which show three orthogonal views of the same object. The method is to find groups of coplanar vertices. No mention is made of any of the problems discussed in Section 2.

The first formal investigation of the problem was that of Markowsky and Wesley [5]. Their algorithm is similar in philosophy to that of Idesawa: it searches for planes of vertices as follows: at each vertex, it identifies the possible planes (as defined by non-collinear pairs of edges meeting at that vertex), searches for other vertices in each such plane, and tries to find loops of edges joining them. Although it could in principle be made tolerant of small errors in vertex coordinates, it requires that the equations of planes passing through each vertex can be calculated with reasonable accuracy. Like Idesawa's approach, it requires a full 3D wireframe (all three coordinates, x, y and z, of each vertex must be known) so it cannot be used on its own for interpreting wireframe drawings, and, to achieve the required accuracy when calculating plane equations, these vertex coordinates must also be reasonably accurate, so it is further inappropriate for drawings derived from freehand sketches.

Where Markowsky and Wesley improve significantly on Idesawa is by treating the face identification as a problem in its own right. They discuss the possible pitfalls, and in particular discuss in detail the internal face problem (Section 2.2) and how to avoid it.

Although developed for polyhedra, their algorithm also works for many, but not all, objects with curved faces. It also works for drawings containing multiple objects. It can also be extended to find coplanar faces (determining whether or not this coplanarity is an intentional design feature is an unsolved problem, beyond the scope of our current paper).

The ideas of Markowsky and Wesley remain valid for the problem for which they were introduced, that of converting 3D wireframes to solid models. For example, as part of their recent research into a problem related to 3D data structures, Ungvichian and Kanongchaiyos [20] attempt to find faces in 3D wireframes. Their approach is not significantly different from that of Markowsky and Wesley.

3.3. Purely graph-theoretical approaches

The automated conversion of 3D wireframes to solid models was also the motivation of the first graph-theoretical approaches. Since the formal definition of a *graph* of *vertices* and *edges* matches the topological description of a wireframe drawing, it is natural to investigate to what extent graph-theoretical algorithms can interpret wireframe drawings.

Ganter and Uicker [6] use an unconvincing approach based on finding a minimum spanning tree to detect possible sets of cycles and questionable heuristics such as *any cycle which is a true face has a minimum number of edges in common with any other cycle*. Where more than one set of cycles is possible, they require human intervention to choose the preferred set.

Courter and Brewer [7] improve on Ganter and Uicker in a number of ways, in particular by automatically rejecting internal faces (Section 2.2). Since their algorithm does not use geometric information, it can be used for those curved objects which are topologically equivalent to polyhedra.

Although their approach was designed for use with 3D wireframes, the fact that the resulting algorithm does not use geometric information at all has the consequence that it can equally well be used to process 2D wireframe drawings. Company et al.'s REFER [4] was the first practical application of this possibility.

Inoue et al. [21] extend the capabilities of finding faces in 3D wireframes by presenting a graph-based algorithm which is not restricted to manifold polyhedra. Their approach is specifically designed to avoid using geometric information.

However, the most important consequence of not using geometric information is that resolving loop ambiguity (Section 2.3) remains a problem. Since only geometry can resolve the problem of loop ambiguity, we consider this restriction to be misconceived.

3.4. Search-based approaches

With the introduction of Shpitalni and Lipson's approach [22] we at last have an approach capable, at least theoretically, of solving our specific problem.

The algorithm of Shpitalni and Lipson [22] is one of several similar algorithms assessed by Liu and Lee [23]. All of them search through the set of all possible sets of face loops and then assess them; different search algorithms are compared, and the assessment criteria also differ slightly. This gives good output results in many cases (both are successful in avoiding loop ambiguity, Section 2.3) but it is also very slow. As before, the number of possible face loops is $O(e^v)$, so the number of cases which must be assessed is also $O(e^v)$.

Oh and Kim [24] take this idea further by classifying face loops into one of three categories: *basis* faces, which are certainly faces of the object, *implausible* faces, and *minimal* faces. Implausible and minimal faces which contradict the basis faces can be ruled out. The second stage of the algorithm, finding the best set of

implausible/minimal faces, is fast, but the initial classification step must still consider all possible face loops and remains $O(e^v)$.

Liu, Lee and Cham [8] propose a depth-first search of the search space of cycles, reasoning that in most cases cycles which are part of "good" solutions to the problem are easy to identify. This is a significant improvement on the earlier algorithm of Liu and Lee. Provided that the depth-first search makes correct decisions early on (which, in all of their illustrative examples, it does), the set of plausible topologies can usually be identified in a fraction of a second. Unlike [22] and [23], they also take specific precautions against internal faces (Section 2.2).

All of these algorithms are primarily graph-based, using 2D geometric information only in the assessment/classification criteria.

Also, all of these algorithms are effectively search algorithms which explore a search space of exponential order. Even Liu, Lee and Cham's depth-first search algorithm [8], which usually has fast running times in practice, is theoretically of exponential order since it *might* need to explore the whole search space.

Exponential order can only be avoided if we are prepared to discard "unpromising" parts of the search space. Several such methods have been proposed, as described in the next two subsections.

3.5. Pseudo-random approaches

Liu and Tang [25] use a genetic algorithm to identify faces in a 2D wireframe. For comparison purposes, they also developed a method based on simulated annealing. Again, this is a primarily graph-based approach which uses 2D geometric information in its assessment criteria. They demonstrate successful results from both, with the genetic algorithm being somewhat faster. However, both approaches require a good deal of tuning, making them difficult to use in practice. Since they make much use of random numbers, there can be no guarantee that their approach is repeatable.

3.6. Shortest-path approaches

In addressing the related problem of interpreting natural line drawings, Varley et al. [26,3] takes as its input a partially-complete topology where all of the vertices and edges and some but not all of the faces are known, together with estimates of their 3D geometry. These estimates need not be as accurate as those required by [5]. The method uses a version of Dijkstra's Algorithm [27] for finding least-cost closed loops of edges in vertex–edge graphs. The initial version [26] set the *path length* for each edge to 1. This can fail in cases with loop ambiguity (see Section 2.3). The improved version [3] bases path lengths on how far the edge under consideration is from being coplanar with edges already reached, and is effectively the same algorithm as that presented in Section 4.

In an attempt to produce a polynomial-order algorithm for wireframes, Shesh and Chen [28] also use a version of Dijkstra's Algorithm, using 2D rather than 3D geometric information in their assessment criteria. Since (replicating the error in [26]) the path length for each edge is set to 1, this algorithm can also fail in cases with loop ambiguity.

These two approaches highlight a notable advantage of the Dijkstra's Algorithm approach: its flexibility. It can be used either with the original 2D data of the wireframe drawing, or with 3D data deduced from the inflation process.

Masry et al. [29] inflates to 3D before trying to identify face loops. Starting with two arbitrary edges, it uses a greedy approach which, at each vertex, adds the edge which is most nearly coplanar with preceding edges in the loop. Again, it is a primarily graph-based approach, but using 3D geometric information in its assessment criteria. Although less flexible than Dijkstra's Algorithm, Masry's greedy approach is satisfactory in most cases. However, the choice of arbitrary starting edges means that their approach can fail for objects with internal faces such as the left-hand drawing of Fig. 1.

Beohar [30] uses Prim's Algorithm [31] to determine a minimum spanning tree of the vertex/edge graph, and considers those face loops which can be made by adding one more edge to the minimum spanning tree. The resulting algorithm is very fast, since only $O(e)$ face loops need be examined, but its motivation is questionable. For example, assume a genus-zero polyhedron with f faces to be found. Any minimum spanning tree will contain $(v-1)$ edges, so the number of edges outside the minimum spanning tree (and thus the number of face loops which can be found by adding each such edge in turn to the minimum spanning tree) is $(f-1)$. At least one genuine face loop will be missed.

In considering how face loops may be assessed, Beohar [30] improves on previous authors by making use of the well-known idea (noted by Lamb and Bandopadhay [32] and many others) that many face loops in many engineering objects consist entirely of edges parallel to two of the axes. To this, one could add that an assessment based on this idea need not be performed as a second stage of processing: if Dijkstra's Algorithm is used to find loops of edges, it may be implemented as the path length.

Beohar [30] also suggests that all triangular loops of edges are necessarily faces, and that the problem can therefore be simplified by identifying these first. Unfortunately, this is not the case: consider, for example, a solid constructed from two tetrahedra sharing a face.

3.7. Other approaches

The claim of Suntoro et al. [33] to have developed an $O(v)$ algorithm (where v is the number of vertices) for finding faces in wireframes is misleading on a number of counts. Given a complete set of circuits (external faces and internal faces as in Section 2.2), their algorithm can, theoretically, prune out the internal faces in $O(v^2)$ time (the size of their data structures and time taken to traverse them were omitted from their assessment of time complexity). Whether it can do so in practice is untested, as they give no test results. Since the number of possible circuits is exponential in the number of vertices, the process of creating the input data which their algorithm replies will be of exponential order, so their algorithm is not all that useful.

In an attempt to overcome the problem of wireframe ambiguity, Piquer et al. [17] have attempted to use symmetry to identify the more plausible interpretation, suggesting that the interpretation with the greater geometric symmetry is to be preferred. This is unconvincing on a number of counts. For example, it is intolerant of sketching error, as it is not clear what constitutes "almost" symmetrical. It is culture-specific, as the reported aesthetic preference for symmetrical objects [34] is not universal. There has been no validation performed to show that the interpretation preferred by the algorithm is the same one preferred by a human interpreter, let alone the one intended by the originator of the drawing.

In all of the above, it is assumed that the object is manifold. Sun and Lee [35] discuss methods for identifying inherently non-manifold wireframes.

3.8. Other ideas

Although mainly concerned with the separate problem of identifying whether or not a wireframe has any valid interpretations, Li [11] includes an algorithm which generates all legal sets of faces, without attempting to choose between them. A particularly interesting aspect of this is that the search process is ordered in an attempt to reduce the time required to find the first "optimal solution" (i.e. a solution acceptable to a human). Li reports that his method has detected 10,000 faces of a particularly complex object, taking an hour and a half to do so, and claims that this is faster than

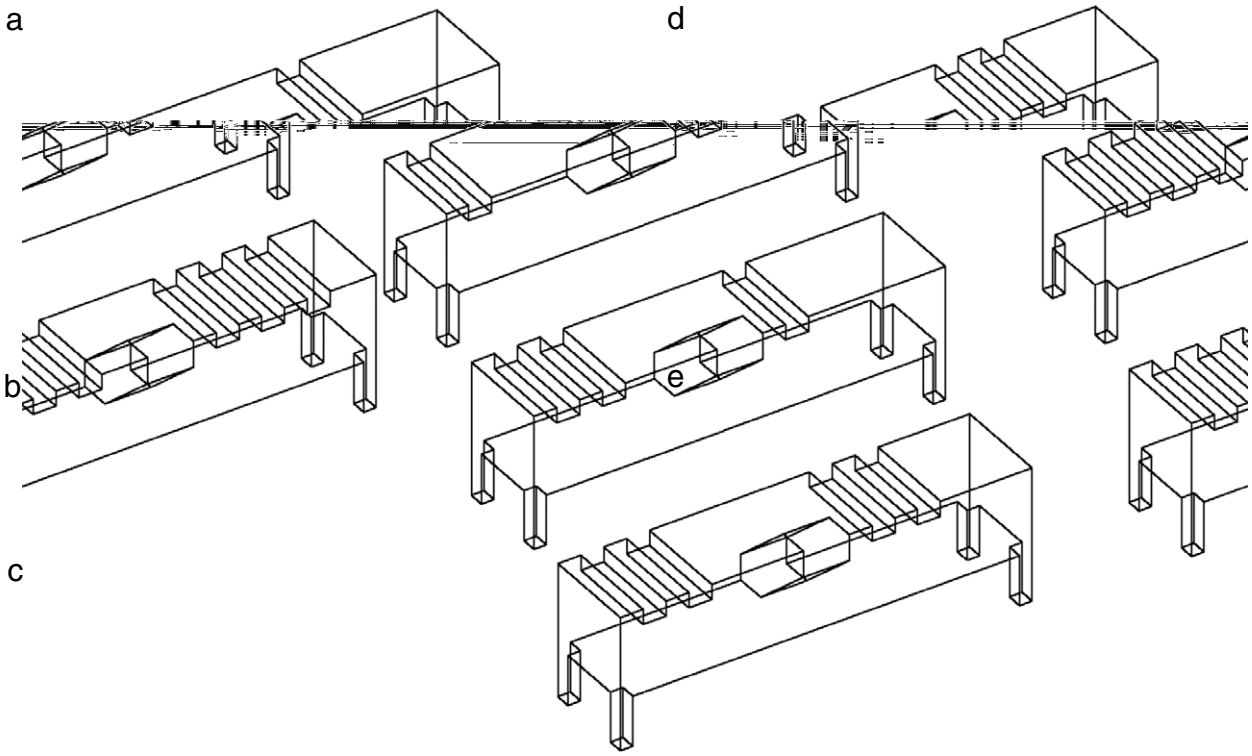

Fig. 13. Desks.

any preceding method. Li does not say how the 10,000 faces found by his approach were validated.

Potentially, Li's approach could be combined with a method for detecting or locating ambiguity and adjudicating between alternative interpretations. At the time of writing, this has not been tested, but it is an idea with promise.

3.9. Summary

In summary, there is good precedent for any of three approaches: primarily geometric, primarily topological using 3D geometric information as an assessment criterion, and primarily topological using 2D geometric information as an assessment criterion.

Theoretically, the position is no clearer. It is obvious that *accurate* inflation solves the face identification problem by converting the problem to the one solved by Markowsky and Wesley [5]. However, if we have not identified face loops, we cannot guarantee to make all vertices on a face coplanar, so we cannot guarantee that the results of our inflation algorithm will be accurate.

4. Discussion: Dijkstra's algorithm

In this Section, we discuss the most popular existing approach, that based on Dijkstra's Algorithm, in some depth, assessing its strengths and weaknesses. We also use it to investigate one of the questions considered in Section 3: is it better to inflate before or after finding faces?

The implementation we used in evaluating the Dijkstra's Algorithm approach is that described in [36]. It includes minor improvements over other approaches based on Dijkstra's Algorithm in that (a) it follows [3] in using path length to discourage non-planar faces and (b) it introduces a method of avoiding internal faces based on choosing an appropriate starting-point for face loop detection.

4.1. Time complexity of algorithm

It is known that a simple implementation of Dijkstra's Algorithm is $O(n^2)$ (this can be improved to $O(n \log n)$ at the cost of additional complexity of implementation) [37]. An approach which performs Dijkstra's algorithm once for each face is thus either $O(n^3)$ or $O(n^2 \log n)$ depending on the data structures used.

In order to determine the practical time complexity of the Dijkstra's Algorithm approach, we implemented the algorithm of [36] using simple data structures and measured the time it took to obtain the faces of two sequences of drawings. In each sequence, the essential nature of the drawing remains unchanged but additional faces are added. The two sequences are *Desks* (Fig. 13) and *Steps* (Fig. 14). Both sequences were inspired by similar drawings in Liu and Tang [25]. The times taken to process these drawings are shown in Table 1.

Fitting a line through this data by linear least-squares regression gives a practical time complexity in terms of the number of edges n of:

- 2D, Desks: $n^{2.618}$
- 2D, Steps: $n^{2.609}$
- 3D, Desks: $n^{2.702}$
- 3D, Steps: $n^{2.749}$

In summary, our testing using simple data structures shows that the practical time complexity is close to $n^{2.6}$ for the 2D version of the algorithm and close to $n^{2.7}$ for the 3D version.

Speculatively, we could extrapolate these figures to obtain a hypothetical timing of around 20 h for a Desk-like drawing with 10,000 faces. On this basis, the report in Li [11] that his more advanced algorithm can process drawings of such a size in about 1.5 h is credible.

4.2. Before or after inflation?

We present here the results of using the Dijkstra's Algorithm approach to process the test drawings of Section 7, which vary in size from 12 to 251 edges.

Table 1
Timings, desks and steps.

Drawing	Fig.	Vertices	Edges	Faces	Dijkstra 2D (ms)	Dijkstra 3D (ms)
Desk 1	13a	60	90	30	18.0	22.2
Desk 2	13b	68	102	34	24.7	30.8
Desk 3	13c	76	114	38	33.1	41.5
Desk 4	13d	84	126	42	43.1	55.6
Desk 5	13e	92	138	46	55.0	69.9
Steps 1	14a	60	90	32	17.9	22.1
Steps 2	14b	68	102	36	24.8	30.7
Steps 3	14c	72	108	38	28.9	36.6

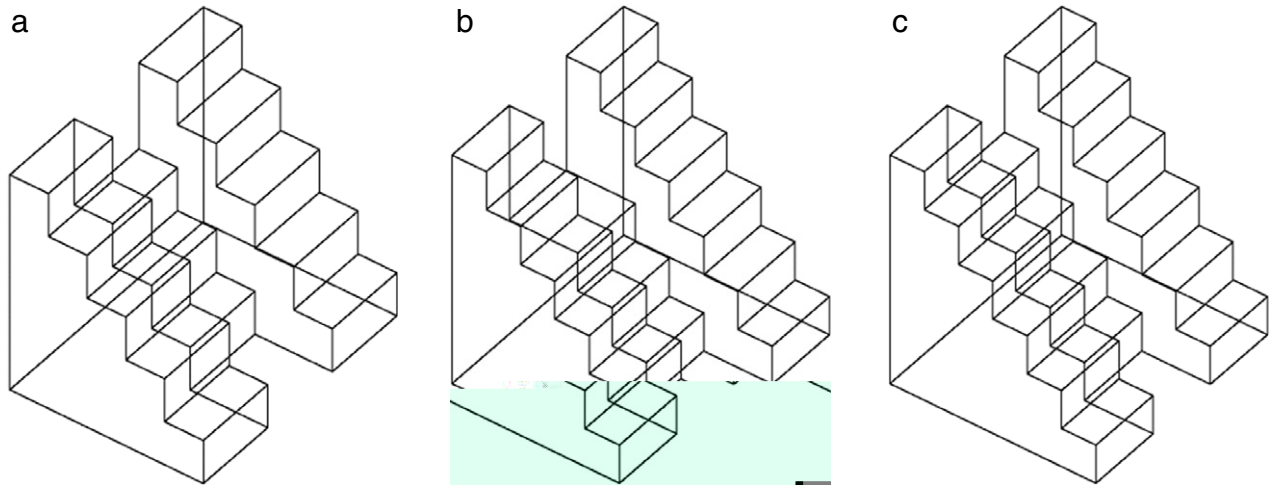

Fig. 14. Steps.

Where the algorithm fails, we distinguish *failure* (the algorithm incorrectly reports that it is not possible to find a consistent set of faces) and *wrong* (the algorithm finds a set of faces, but the wrong set).

The 2D version succeeded in 62 cases, gave the wrong answer in 2 cases, and failed in 19 cases. The 3D version succeeded in 61 cases, gave the wrong answer in 2 cases, and failed in 20 cases. For both versions, eight of the failures were with the Building sequence of drawings (Fig. 15). For those drawings where both versions gave the correct answer, the 2D version was always faster, with the relative timings in Table 1 being typical of the results as a whole.

On the basis of these results, we find, perhaps surprisingly, that the Dijkstra's Algorithm approach performs slightly *better* if the input is restricted to the original 2D data — adding the results of an inaccurate inflation process is, more often than not, not merely useless but actively harmful.

Although the theoretical question of whether to find faces before or after inflation remains open, practically, it is best to perform face identification first.

4.3. Strengths of the Dijkstra's algorithm approach

The strongest case in support of the use of Dijkstra's Algorithm is pragmatic: it usually works. It is difficult to say how the approach using Dijkstra's Algorithm compare with other approaches, since no other approach has been tested so thoroughly. However, one point in particular is worth noting: the size of the drawing has no effect on whether the approach succeeds or fails. It is the presence of a particularly troublesome feature, not sheer number of lines, which makes the difference (we examine two such failures in detail below). For example, the method processes all of the Desks (Fig. 13) and Steps (Fig. 14) correctly, but fails for all eight of a sequence of Buildings (Fig. 15).

Thus, the fact that some other approaches have been tested using larger drawings is not, in itself, proof of their superiority since those drawings may not contain the particular features which cause this approach to go wrong.

Another argument in favour of the Dijkstra's Algorithm approach is that it is fast. For example, our implementation finds all 46 faces in the final drawing of Fig. 13 in 55 ms, whereas [25] quote 830 ms for their method (hardware differences may contribute to this difference). Such speed is important when the problem to be solved is just one component of a larger problem.

4.4. Weaknesses of the Dijkstra's algorithm approach

The essential problem with the use of Dijkstra's Algorithm is theoretical: the assumptions built into it do not correspond to the problem to be solved.

Dijkstra's Algorithm finds the least-cost path between two vertices of a graph. It is designed around the implicit assumption that the *path length*, the cost of traversing an edge, is fixed: it does not vary with the route taken before or after that edge (calculation of such costs is external to the algorithm). By making the start and end points the same, it can be used to find least-cost circuits starting and ending at a chosen point.

This inbuilt assumption leads to problems, such as those we examine below. It is, in practice, easy to overcome these problems. What is difficult is demonstrating that these are all of the possible problems which must be avoided. It is probable that they are not: there are other problems which only occur in rare cases which have not been considered.

4.4.1. Problem 1

Firstly, consider the drawing in Fig. 16. Suppose that, having found the base and four walls of the object, we now want to find a face starting with edge 1–2. Edge 2–3 is a reasonable continuation

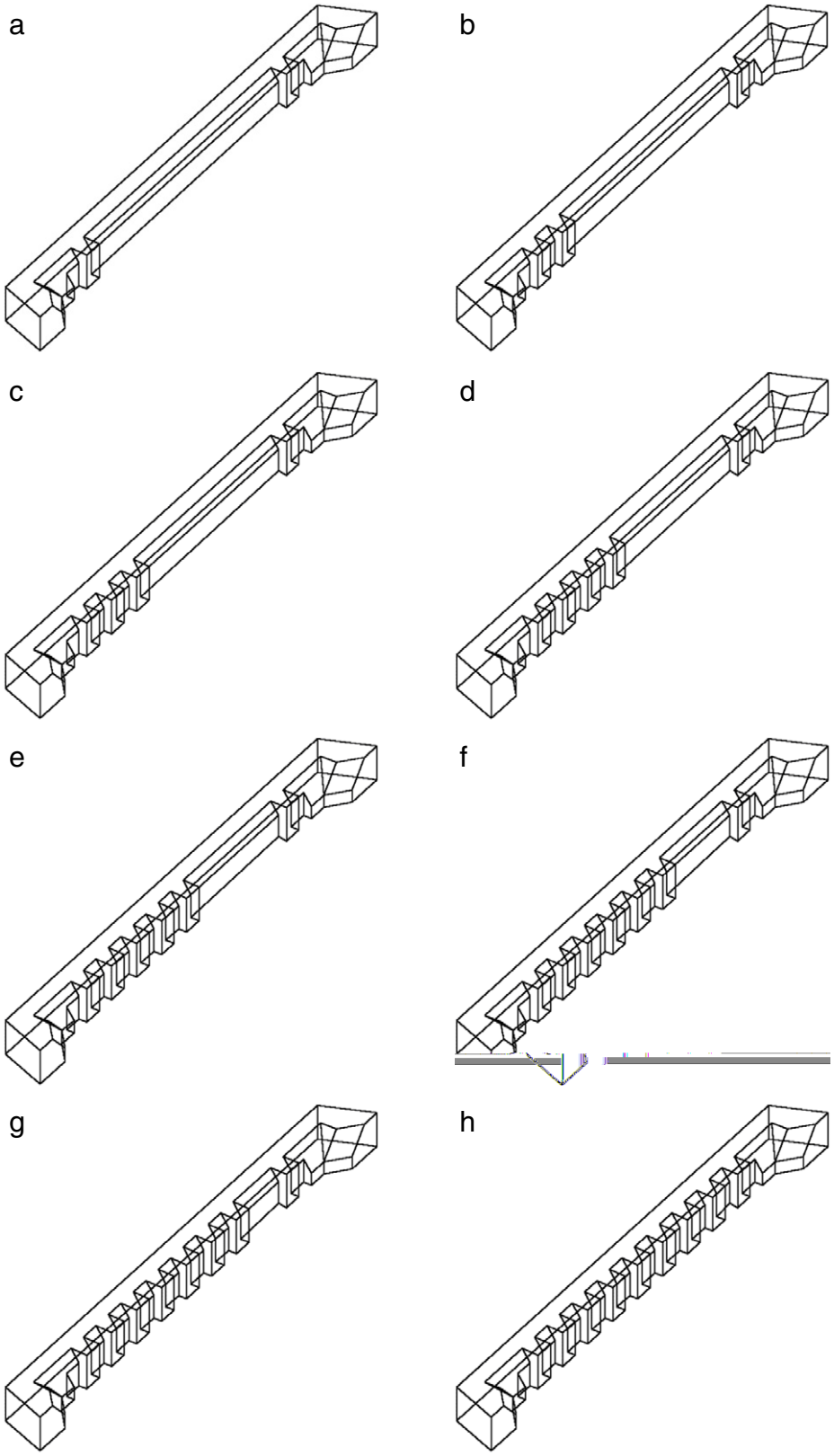

Fig. 15. Buildings.

of the loop, but what follows next, 3–A or 3–B? When working in 2D, edge 3–B is the closer to being parallel to edge 1–2, so this is the path examined first. Since a loop can be completed using only those edges which are parallel to edges already chosen (1–2–3–B–C–A), the alternative continuation 3–A is never examined.

4.4.2. Problem 2

A similar problem occurs in Fig. 17, although here the failure, when it happens, is easier to detect. Suppose that we have chosen edge 1–2 as our starting point for an iteration of Dijkstra's Algorithm (possibly because we have just identified the adjacent

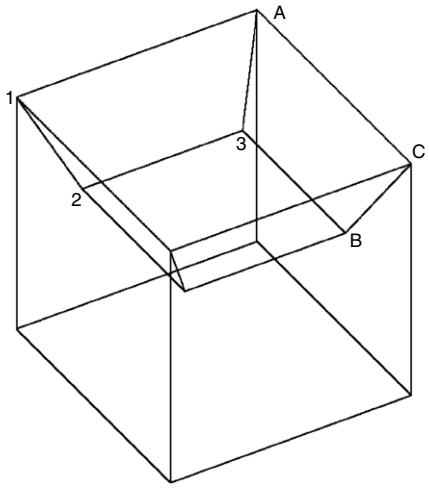

Fig. 16. A problem.

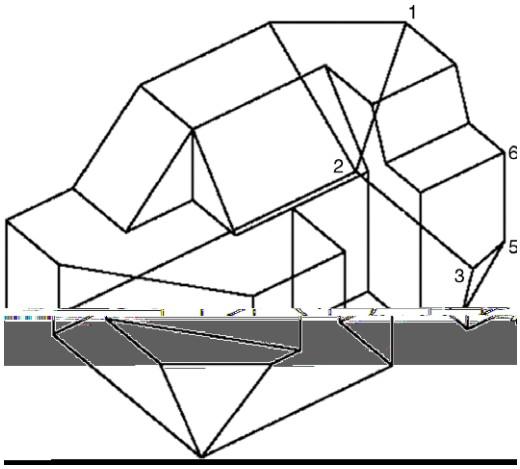

Fig. 17. A similar problem.

triangular loop of edges as a face). Edge 2–3 is a reasonable continuation of the loop. What follows next? We must consider both branches, 1–2–3–4…and 1–2–3–5…, with the former taking priority as 3–4 is closer than 3–5 to being parallel to 1–2.

The path branches again at vertex 4, with one possibility being 1–2–3–4–5…. This turns out to be promising (4–5 is also closer than 3–5 to being parallel to 1–2), and we continue 1–2–3–4–5–6…and onwards around the loop.

At some point, the algorithm may return to the branch 1–2–3–5…, but it will dismiss this as it has already found a "better" route to vertex 5 via vertex 4. So the loop of edges which the algorithm finds is the one beginning 1–2–3–4–5–6…

The algorithm has fulfilled its purpose: it has found the lowest-cost loop of edges back to the starting-point. What it has not done (because this is not part of its purpose) is examine the residue. The half-edges 3–5 and 5–3 have been isolated from the rest of the drawing, so it is not possible to create a complete wireframe after having made this wrong choice of loops.

There are a several way this problem could be avoided (a more astute choice of starting-point, for example, or incorporating a rule making use of two consecutive edges from a triangular loop invalid when building a larger loop), but the impression created is that we have to add special cases because of a poor choice of central algorithm.

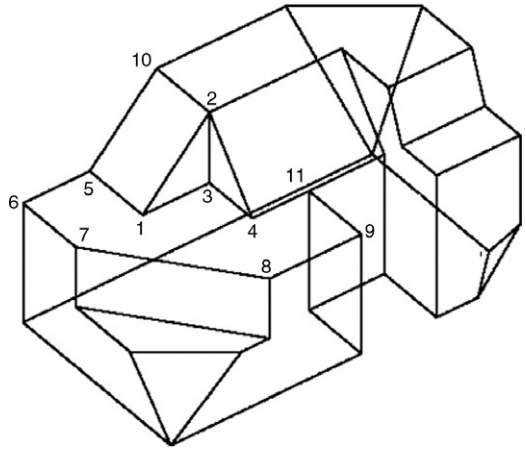

Fig. 18. Another problem.

4.4.3. Problem 3

Secondly, consider the drawing in Fig. 18. Finding the face 1–3–2–1 is straightforward. Our framework will then note that the half-edges 1–3, 3–2 and 2–1 have been used. Similarly, finding the face 2–3–4–2 is also straightforward, and when we do this we note that the half-edges 2–3, 3–4 and 4–2 have also been used. At this point, the half-edges 1–2, 2–4, 4–3 and 3–1 are all still available, so what is wrong with identifying a face 1–2–4–3–1?

We cannot avoid this loop of edges by choice of starting-point (indeed, following the rules in Section 4.1, the half-edge 4–3 would be a particularly good starting-point). The only other variable under our control is the *path length* function, so it is this which we must use to ensure that, for example, half-edges 3–1 and 1–2 cannot appear in the same face. For example, if the half-edge 3–1 appears in a loop, the cost of half-edge 1–2 is infinite.

Thus we must use also the *path length* function (identified in [36] as our means of incorporating geometric information) as a way of implementing the constraints on *topology* described in Section 1.3.

However, this leads to a deeper problem: the *path length* function of traversing a particular edge now depends on the path to that edge, contravening an underlying assumption of Dijkstra's Algorithm.

4.4.4. Problem 4

Consider again the situation after we have found faces 1–3–2–1 and 3–4–2–3. We could now choose edge 3–1 as our starting point (although not best, this is reasonable, as it meets the criteria in Section 4.1). We now try to find the face starting 3–1–5…and discover when reaching vertex 5 that there are two choices: 3–1–5–6…and 3–1–5-10…

3–1–5–6…looks best. Perhaps we get as far as 3–1–5–6–7–8–9 …before deciding to examine the other branch.

After 3–1–5-10–2–4…we cannot complete the loop, because we cannot include 2–4 and 4–3 in the same loop (3–4 and 4–2 have already been used together in another face). 4–3 is marked as unusable, for example by setting its cost to infinity. Any other path is obviously wrong, as we can never reach the starting point. For example, we might explore 3–1–5-10–2–4–11…before realising that it leads nowhere.

Returning to the first branch. 3–1–5–6–7–8–9–11–4…would be the right way to go, but at this point the algorithm recognises that it has been to vertex 4 before, and that it did not lead anywhere because 4–3 has previously been identified as unusable. On that basis, this path is rejected.

Again, there are several ways by which this problem could be avoided, most obviously by a better choice of starting-point

(choosing edge 4–3 rather than edge 3–1 would avoid the problem entirely), but again the impression created is that we only need to do this because our core algorithm is inappropriate.

4.5. Alternatives to Dijkstra's algorithm

Graph-based approaches have many advantages, as described in Section 4.3. Ideally, we should wish to retain a graph-based approach, keeping the same control structure and merely replacing Dijkstra's Algorithm with another black-box algorithm.

Unfortunately, Dijkstra's Algorithm has been so successful that there are no radically different established algorithms for finding loops in vertex–edge graphs. The *Bellman–Ford Algorithm* [38,39] is similar to Dijkstra's algorithm but allows edges to have negative costs. From our point of view, this does not help, since the costs remain fixed. The *Floyd–Warshall Algorithm* [40,41] is significantly different in approach from Dijkstra's algorithm, but nevertheless solves a similar problem. Again, edge costs remain fixed. Indeed, it seems to be assumed throughout the whole of graph theory that if edges have costs, those costs are a pure function of the edge. This is not what is required for the problem at hand.

5. A new algorithm

In this section, we introduce our new algorithm.

In developing our new algorithm, we sought to identify and retain the advantages of its predecessors while rejecting the disadvantages. The notable advantage of Dijkstra's Algorithm is that it is a general-purpose graph-based algorithm to which different assessment criteria (2D or 3D) can be added as interchangeable plug-ins. This is something we wish to retain.

The genetic algorithm approach of Liu and Tang [25] is clearly successful, notwithstanding our skepticism of algorithms which rely on random numbers. We attribute its success, not to the algorithm itself, but to the well-chosen data structures on which it operates. Clearly, the problem at hand is one of assembling half-edges into "strings" (and eventually loops) of half-edges, and we wish to retain "good" strings and discard "bad" strings. These concepts of concatenation and evaluation are central to genetic algorithms. They are also concepts which we wish to retain.

Our own algorithm is also based on the concept of concatenating and evaluating strings. It differs from [25] in that the process of assembly is rule-based, not random.

We start with a worked example. In following the example, it may be useful to know that its inspiration came from a technique used in chemistry for growing crystals. We start with a saturated solution, to which we add a single small crystal as a seed. Various crystals gradually grow. Some of them are good, well-shaped crystals, whereas others are misshapen. When we have a satisfactory large, well-shaped crystal, we remove it from the solution, break up any misshapen crystals which have also formed, and continue.

5.1. Worked example

Consider the cube in Fig. 19.

Initially, our solution (the *master list* of *strings*, defined in Section 5.2) will contain 24 strings, two for each of the edges in the figure. Since all edges touch two quadrilateral loops, their initial priorities are the same.

`Master List: AB BA BC CB CD DC DA AD EF FE FG GF GH HG HE EH AE EA BF FB CG GC DH HD`

We start by introducing a seed: *concatenating* two edges meeting at a trihedral vertex (since all vertices are topologically equivalent in this example, the choice is arbitrary). The strings *AB* and *BC* are removed from the master list and the string *ABC* is added. The seeding process is described in more detail in Section 5.3.

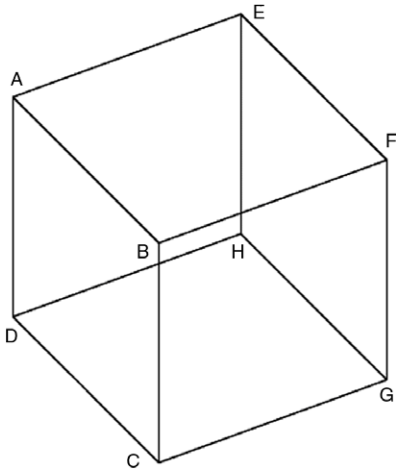

Fig. 19. Vertices of a cube.

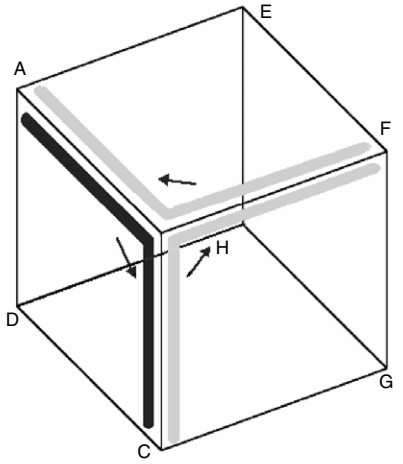

Fig. 20. Forced concatenations.

`Master List: BA CB CD DC DA AD EF FE FG GF GH HG HE EH AE EA BF FB CG GC DH HD ABC`

As with crystallisation, some parts of the process are effectively autonomous. We now examine the master list for *forced concatenations* (described in Section 5.4). Since *ABC* appears in the master list, *CB* and *BA* cannot be concatenated, so *FB* and *BA* must be concatenated instead, as must *CB* and *BF*. See Fig. 20. *FB*, *BA*, *CB* and *BF* are removed from the master list and *FBA* and *CBF* added.

`Master List: CD DC DA AD EF FE FG GF GH HG HE EH AE EA CG GC DH HD ABC FBA CBF`

Since there are no more forced concatenations, and no pair of strings can be *merged* to form a face (described in Section 5.5), we form hypotheses, taking a *working copy* of the master list.

`Working Copy: CD DC DA AD EF FE FG GF GH HG HE EH AE EA CG GC DH HD ABC FBA CBF`

Again, because of topological equivalence, the choice is arbitrary. For example, if we choose to extend *ABC*, we have two choices, concatenation with *CD* and concatenation with *CG*. Since *CD* is parallel to *AB*, this is the one to be preferred. On the working list, we delete *ABC* and *CD* and add *ABCD*.

`Working Copy: DC DA AD EF FE FG GF GH HG HE EH AE EA CG GC DH HD FBA CBF ABCD`

This produces two forced concatenations, so we remove *DC* and *CG* and add *DCG*, and remove *GC* and *CBF* and add *GCBF*.

`Working Copy: DA AD EF FE FG GF GH HG HE EH AE EA DH HD FBA ABCD DCG GCBF`

There are no more forced concatenations, so we now examine the master list for mergers which can produce a cyclical loop. We find that *ABCD* and *DA* can be merged to form a face loop. Once this is established, we add *ABCDA* to the list of face loops, discard the working list, and delete *ABC*, *CD* and *DA* from the master list.

`Face Loops: ABCDA`

`Master List: DC AD EF FE FG GF GH HG HE EH AE EA CG GC DH HD FBA CBF`

We can now perform the two forced concatenations, *DC* with *CG* and *GC* with *CBF*, on the master list (remove *DC* and *CG* and add *DCG*, and remove *GC* and *CBF* and add *GCBF*).

`Master List: AD EF FE FG GF GH HG HE EH AE EA DH HD FBA DCG GCBF`

Examining the master list for mergers, we find that merging *GCBF* with *FG* produces a cyclical loop. We add *GCBFG* to the list of face loops and delete *GCBF* and *FG* from the master list.

`Face Loops: ABCDA GCBFG`

`Master List: AD EF FE GF GH HG HE EH AE EA DH HD FBA DCG`

Re-examining for forced concatenations, we delete *FBA* and *EF* and add *EFBA*, and delete *GF* and *FE* and add *GFE*.

`Master List: AD GH HG HE EH AE EA DH HD DCG EFBA GFE`

Re-examining for mergers, we add *AEFBA* to the list of face loops and delete *EFBA* and *AE* from the master list.

`Face Loops: ABCDA GCBFG AEFBA`

`Master List: AD GH HG HE EH EA DH HD DCG GFE`

Re-examining for forced concatenations, we delete *GFE* and *EH* and add *GFEH*, and delete *HE* and *EA* and add *HEA*.

`Master List: AD GH HG DH HD DCG GFEH HEA`

Re-examining for mergers, we add *GFEHG* to the list of face loops and delete *GFEH* and *HG* from the master list.

`Face Loops: ABCDA GCBFG AEFBA GFEHG`

`Master List: AD GH DH HD DCG HEA`

Re-examining for forced concatenations, we delete *DCG* and *GH* and add *DCGH*, and delete *AD* and *HEA* and add *HEAD*.

`Master List: DH HD DCGH HEAD`

Finally, we are left with *DCGH*, *HEAD*, *DH* and *HD* on the master list. We perform two more mergers to obtain the final two face loops *DCGHD* and *HEADH*.

`Face Loops: ABCDA GCBFG AEFBA GFEHG DCGHD HEADH`

`Master List: empty`

5.2. Basic concepts

Here, we define our basic data structures and the operations which can be applied to them.

We define a *string* to be an ordered sequence of two or more vertices, in which each consecutive pair of vertices is joined by a half-edge. A string may not include the same vertex twice. The minimum string is two vertices joined by a single half-edge. Each string has a priority associated with it, since some strings are more important than others: they are in a part of the object we particularly want to work on.

Two strings may be *concatenated* to form a larger string if the last vertex in one is the first vertex in another, provided that (a) no other vertex appears in both strings, and (b) the new triple of three consecutive vertices formed by the concatenation does not already appear in reverse order in an existing face or already-concatenated string.

Two strings may be *merged* to form a face if the last vertex in each is the first vertex in the other, provided that no other vertex appears in both strings.

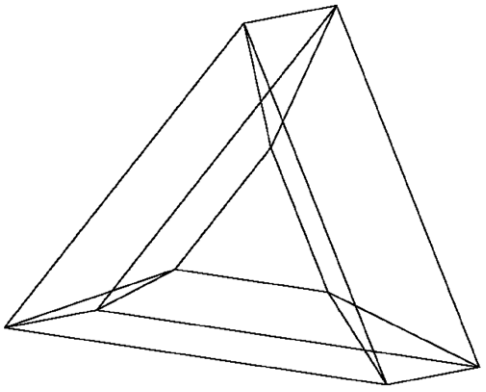

Fig. 21. Sugihara's torus.

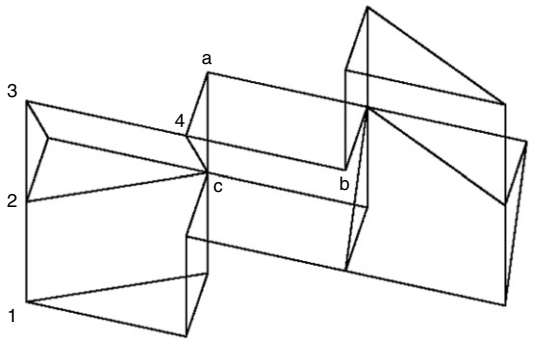

Fig. 22. A special case.

5.3. Starting conditions

Initially, we have $2e$ strings, where e is the number of edges. Each string represents a single half-edge.

We start by choosing an arbitrary trihedral junction and concatenating one string terminating at this junction with another starting at this junction. Note that it is only at trihedral junctions that we can be sure that the concatenated string is part of a true face rather than an internal face.

Obviously, this is not possible in objects with no trihedral vertices (e.g. the octahedron or the icosahedron). For these objects, we break the symmetry by assuming that an arbitrary triangular loop of edges is a face. This is not ideal, since triangular loops of edges need not be faces, particularly in objects dominated by non-trihedral vertices. For example, Sugihara's Torus [42] (Fig. 21) has six triangular loops of edges, none of which is a true face of the object. However, we do not know of a better starting assumption.

5.4. Forced concatenation

Consider a string T terminating at vertex V. Enumerate the set $\{S\}$ of strings starting at V which can legitimately be concatenated/merged with T. If there is only one string in this set, the concatenation/merger must be performed. (If there are no strings in this set, this is an error condition.)

5.5. K-vertices: A special case

At a tetrahedral K-vertex or pentahedral extended-K vertex, exactly one pair of collinear half-edges must be in the same loop. To illustrate this point, consider the drawing in Fig. 22.

Start at vertex 1, and work through vertices 2 and 3 to vertex 4. What now? Clearly, it is desirable to continue along edge 4–b.

Four half-edges arrive at vertex 4 and four half-edges leave. Analysis shows that, if there were no other restrictions than those already described in earlier sections, there would be six possible permutations of arriving and leaving half-edges:

i. 3–4–a + a–4–b + b–4–c + c–4–3
ii. 3–4–a + a–4–c + c–4–b + b–4–3
iii. 3–4–b + b–4–a + a–4–c + c–4–3
iv. 3–4–b + b–4–c + c–4–a + a–4–3
v. 3–4–c + c–4–a + a–4–b + b–4–3
vi. 3–4–c + c–4–b + b–4–a + a–4–3

However, since the permutation must include one or other collinear pair of half-edges, permutations i and vi, which include neither, can be discarded. If, in addition, we have already identified the triangular loop 4–c–a–4 or 4–a–c–4 as a face, we can discard a further two possible permutations, retaining only ii and iii or only iv and v respectively.

In some circumstances, such a reduction of the set of possible half-edge permutations can result in forced concatenations, so any search for forced concatenations must pay particular attention to *K*-vertices and extended *K*-vertices.

Note that this does not apply at higher-order extended-K vertices (hexahedral and above), as can be seen (for example) at vertex c in Fig. 22, unless all of the other edges at the vertex are the same side of the through edge.

5.6. Voluntary merger

The purpose of our algorithm is to find faces. If two strings can legitimately be merged, we do this.

It is possible that this could produce a suboptimal result. There is no check that the resulting face is planar, and it is conceivable that a better loop of edges could be found by continuing to explore one of the other possible loops. However, doing this would certainly result in a slower algorithm.

5.7. Data

Obviously, while we are exploring hypotheses, there may be hypotheses which are wrong (the "misshapen crystals" in our analogy). One of the problems with the Dijkstra's Algorithm approach (other than that it is the wrong algorithm for the task) is that it works on one face at a time. The only choice it has available is that of the starting-point, and it is not possible to tell, simply from the starting point, which loops will be small and which large, so quite early in the process, it will often choose to work on a many-sided face. Many-sided faces with many alternatives are difficult, and give the algorithm an opportunity to go wrong.

We would prefer to work on several faces simultaneously, picking the first one to be completed (in our analogy, we allow several crystals to grow simultaneously, picking out a well-shaped crystal when we see one). This is more likely to be a small face than a large one, so it is more likely to be correct. Using this approach, by the time we get to the many-sided faces, there are few alternatives so little opportunity to go wrong.

To this end, we maintain two lists of strings, a *master* list and a *working* list. Each is a priority-ordered list of strings. The means of calculating priority is an implementation detail, discussed in Section 6; here, we merely note that it is a measure of how close a string is to the part of the object we wish to work on next. In our analogy, this provides an additional incentive to the growth of "well-shaped crystals".

Whenever we merge two strings into a face, accepting this new face loop as a true face of the object, this constitutes additional data which all hypotheses should take into account. Therefore, as soon as we have a new face, we add this to the *master* list (by analogy, our store of "well-shaped crystals") and jettison all other working hypotheses (by analogy, breaking up "misshapen crystals"). We then re-examine the updated master list for forced concatenations and potential voluntary mergers.

When we can make no further progress with the master list, we take a copy of it and use this as our *working* list for exploring hypotheses.

5.8. Algorithm

For simplicity of presentation, we have divided the algorithm into two, (a) a top level which processes only the master list and (b) a subroutine for examining hypotheses. Error conditions are not shown. For a more detailed guide to implementing the algorithm in practice, see [43].

(Top level of algorithm)
Create initial master string list, two entries per edge
Assign priorities to all strings in the list (see 6.1)
Choose a trihedral vertex, and concatenate two strings at this vertex
While there are strings remaining in the master list
  Examine the master list for the presence of forced concatenations
    If there are forced concatenations, perform them
    Otherwise, examine the master list for the presence of voluntary mergers
      If there are voluntary mergers, perform them
      Otherwise, examine hypotheses (see subroutine below)
(Subroutine: Examine Hypotheses)
Take a working copy of the master string list
Repeat
Take the highest-priority string *S* in the working string list
Find the string *T* which has the best *mating value* with *S* (see 6.2)
Concatenate *S* and *T*, and reduce the priority of the resulting string (see 6.1)
Repeat
  Examine the working string list for the presence of a forced concatenation
  If there is a forced concatenation, perform it
    If the forced concatenation created a new face, update the master list accordingly and exit this subroutine
  If there is no forced concatenation
    If there is a voluntary merger available, create the face, update the master list accordingly and exit this subroutine
    Otherwise exit this inner loop

5.9. Mating value

We define the *mating value* of two strings to be a function of how "coplanar" the result will be if the strings are concatenated. This context-dependent function is the new concept which distinguishes our algorithm from fixed-weight algorithms such as Dijkstra's Algorithm.

The mating value is not defined if the two strings cannot be concatenated.

As with the cost function used by Dijkstra's Algorithm, we use different mating values to create the 2D and 3D versions of our algorithm, which are otherwise identical.

In our implementation, we use a value in the range 0 to 1, where 0 is "bad" and 1 is "perfect". In Section 6, we give more details of how intermediate values are calculated.

Some planarity conditions are obvious. If the two strings are both minimum-size strings of two vertices, the resulting three-vertex string will always be planar.

Table 2
Initial string priorities.

Edge touches one or two triangular loops	$100 + \boldsymbol{T}$
Edge touches two quadrilateral loops	$100 + 2\boldsymbol{Q}$
Edge touches one quadrilateral loop	$100 + \boldsymbol{Q}$
Edge touches no triangular or quadrilateral loops	100

Otherwise, in 3D, we can measure coplanarity by fitting planes through the two strings and measuring the angle between them.

In 2D we must rely on "cues", of which parallelism is the most helpful (e.g. parallelism between opposite edges in quadrilateral loops usually indicates a real face, and edges of normalon faces are alternately perpendicular to and parallel to the initial edge). Hence, we examine the $(Sv-1)(Tv-1)$ edge pairs, where Sv and Tv are the number of vertices in the two strings S and T respectively, and for each such pair we calculate a function based on how parallel these two edges are (1 = parallel, 0 = perpendicular, for intermediate values see implementation details). In an ideal match, half of the edges in S will be parallel to half of the edges in T, and the other half of the edges in S will be parallel to the other half of the edges in T. On this reasoning, in calculating the mating value, we discard the lowest half of the edge parallelism function results, calculate the mean of what remains, and use this as the mating value.

6. Implementation details

The algorithm we presented in Section 5 concatenates *strings* of vertices and eventually merges them to form face loops. A string is an ordered list of vertices in which no vertex occurs more than once. Each string has a *priority* associated with it which reflects how near it is to the part of the object on which we currently want to work. Each pair of strings has a *mating value* which reflects how close they are to being coplanar.

This section describes how, in practice, we assess priorities and mating values. We conclude with some further comments on the algorithm and its implementation.

6.1. String priority

We allocate higher initial priorities to edges which touch triangular or quadrilateral loops, as shown in Table 2.

$\boldsymbol{T}$ and $\boldsymbol{Q}$ are tuning constants. We use $\boldsymbol{T} = 10$ and $\boldsymbol{Q} = 50$. The base value of 100 is arbitrary, and any other value could be used as long as $\boldsymbol{T}$ and $\boldsymbol{Q}$ are changed proportionately.

When exploring hypotheses, we set the priority of a string resulting from concatenating two smaller strings to $\boldsymbol{P}_{AB} = \boldsymbol{r}\boldsymbol{M}_{AB}(\boldsymbol{P}_A + \boldsymbol{P}_B)$, where $\boldsymbol{P}_A$ and $\boldsymbol{P}_B$ are the priorities of the smaller strings, $\boldsymbol{M}_{AB}$ is the mating value of the two strings (see next subsection) and $\boldsymbol{k}$ is an arbitrary constant (we use $\boldsymbol{r} = 0.4$).

The sensitivity of $\boldsymbol{T}$, $\boldsymbol{Q}$ and $\boldsymbol{r}$ to changes is examined in Section 8.4.

We find triangular loops of edges by considering each edge $\boldsymbol{UV}$ and each vertex $\boldsymbol{W}$ which is neither $\boldsymbol{U}$ nor $\boldsymbol{V}$. If edges $\boldsymbol{UW}$ and $\boldsymbol{VW}$ also exist, the three edges touch a triangular loop.

We find quadrilateral loops of edges by considering each edge $\boldsymbol{UV}$ and each edge $\boldsymbol{WX}$ where $\boldsymbol{U}$, $\boldsymbol{V}$, $\boldsymbol{W}$, $\boldsymbol{X}$ are all different. If edges $\boldsymbol{UW}$ and $\boldsymbol{VX}$ both exist, or if edges $\boldsymbol{UX}$ and $\boldsymbol{VW}$ both exist, the four edges touch a quadrilateral loop.

Both of these are performed as preprocessing, before the main algorithm starts. Both are clearly polynomial and of a lower order than the main algorithm.

6.2. Mating values

Our new algorithm is designed to be useful both in the 2D domain (when we use it before inflation) and in the 3D domain (when we use it after inflation). The *only* change is that a different mating value is used.

We minimise even this change by using a common approach:

For each edge in each string, calculate a function which reflects how close they are to being parallel (see 6.2.1 and 6.2.2)
Discard all such values below the median value
Return the arithmetic mean of what remains (i.e. the median value and above)

This function is intended to reflect how close the two strings come to our ideal: half of the edges in string A are parallel to half of the edges in string B, and the other half of the edges in string A are parallel to the other half of the edges in string B.

Our initial testing supports the idea that the overhead of calculating the mean of the best half of the parallelism values is worthwhile. One suggested alternative, that of using the best parallelism value, is in practice worse in the 2D domain and much worse in the 3D domain. It would also be more vulnerable to drawing inaccuracies, as a single badly-drawn line could result in many inappropriately good mating values.

Ideally, the calculation of 2D and 3D parallelism functions should also be as similar as possible. We have chosen to use a function of the cosine of the angle between two vectors. There are nevertheless unavoidable differences. Firstly, the angles themselves vary (for example, vectors which are perpendicular in 3D are 60 degrees apart in 2D isometric projection). Secondly, the 2D angles depend only on the test data, whereas the 3D angles are the output of an earlier stage of processing. The two options must therefore be treated separately, as in 6.2.1 and 6.2.2.

One exceptional case must be noted. Clearly, if the two strings are both minimum-size strings of two vertices, the resulting three-vertex string will always be planar. There is no theoretical justification for choosing any particular mating value.

While developing our algorithm, we used random numbers for the precise mating values of two minimum-sized strings: the exact value would be a random value in the range 0.9 to 1.0. Even this single inclusion of a random element proved unsatisfactory. For example, when applying the same algorithm to Fig. 68 six times in succession, we obtained four different results, with the correct result occurring only twice. This result contributes to our skepticism towards any method which relies on random numbers.

The finished algorithm is less sensitive to the choice of mating value for two minimum-sized strings. The results obtained in Section 8 assume a fixed value of 1.0. Marginally better results can be obtained by reducing this to 0.2: Fig. 68 is interpreted correctly by the 3D version.

6.2.1. Mating Values: 2D

The function which we use to determine how close two lines are to being parallel is $(\boldsymbol{\cos}\ \boldsymbol{\delta})^{\boldsymbol{k}}$, where $\boldsymbol{\delta}$ is the difference in angle between the two lines and $\boldsymbol{k}$ is an arbitrary even integer constant. For historical reasons, we use $\boldsymbol{k} = 10$ in most of our investigations. Any value from $\boldsymbol{k} = 8$ to $\boldsymbol{k} = 36$ gives the same results. With $\boldsymbol{k} = 6$, Fig. 56 is interpreted incorrectly. With $\boldsymbol{k} = 38$, Fig. 71 is interpreted incorrectly.

6.2.2. Mating values: 3D

The function which we use to determine how close two edges are to being parallel is $(\hat{\boldsymbol{a}} \cdot \hat{\boldsymbol{e}})^{\boldsymbol{k}}$, where $\hat{\boldsymbol{a}}$ and $\hat{\boldsymbol{e}}$ are normalised vectors in the directions of the two edges and $\boldsymbol{k}$ is an arbitrary even integer constant. For historical reasons, we use $\boldsymbol{k} = 10$ in most of our investigations. Any value from $\boldsymbol{k} = 6$ to $\boldsymbol{k} = 10$ gives the same results. With $\boldsymbol{k} = 4$, Fig. 66 is interpreted incorrectly. With $\boldsymbol{k} = 12$, Fig. 33 is interpreted incorrectly.

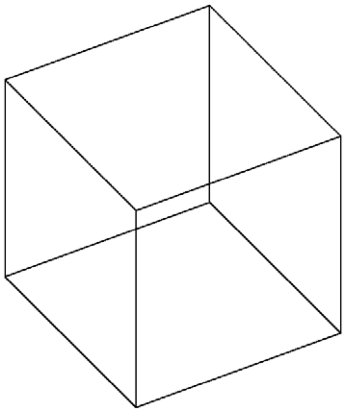

Fig. 23. Cube.

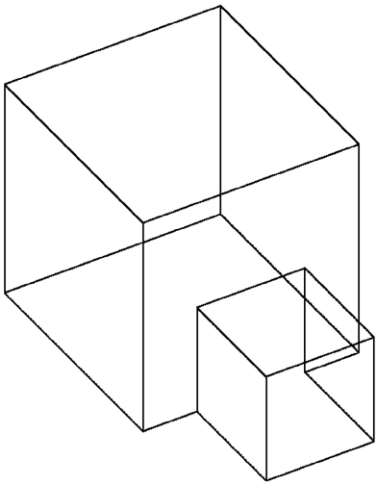

Fig. 24. Offset cube.

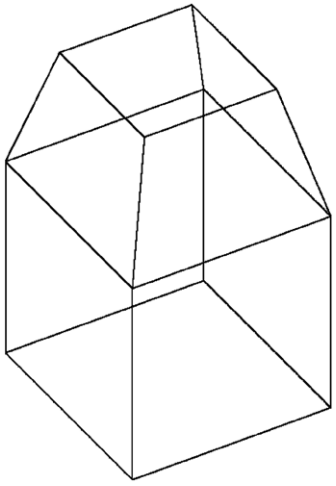

Fig. 25. Pivot cushion.

6.3. Implementation details

We start the process by choosing an arbitrary trihedral junction.

When describing forced concatenation in Section 5.3, we considered strings terminating at each vertex and enumerated the set of strings starting at this vertex. In principle, we should also consider strings starting at a vertex and enumerate the set of strings terminating at that vertex; in practice, doing both is redundant.

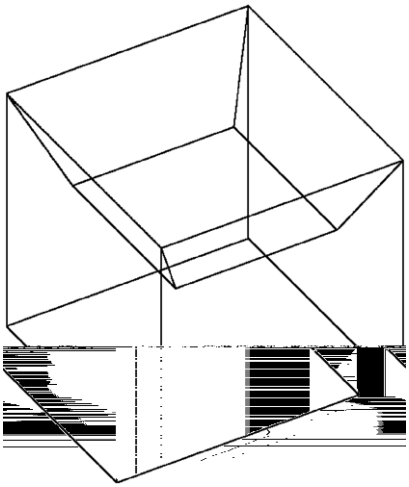

Fig. 26. Anti-vibration pad.

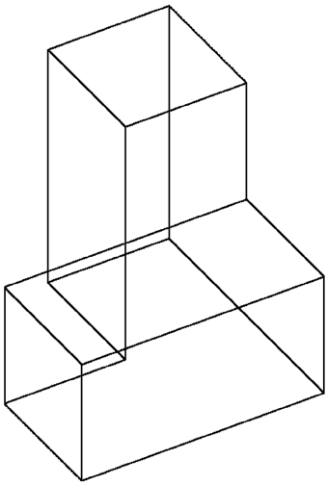

Fig. 27. Hinge arm.

After each sequence of operations (one hypothesis followed by any resulting forced concatenations), whether on the master list or the working list, that list is re-ordered to reflect the new priority order.

7. Test data

This section illustrates our test data. This test set can also be found on-line at:
http://pacvarley.110mb.com/Sketch/Wireframes/index.html

7.1. Discussion

We can classify wireframe drawings in several ways. In considering them as test data, perhaps the most important distinction is between *easy* and *difficult* drawings. *Easy* drawings are those wireframe drawings where we can be confident that the faces will be identified correctly by any reasonable current approach. *Difficult* drawings are those wireframes about which, because they include one or more of the issues which make the face-loops identification problem non-trivial, we cannot be so confident.

Drawings may be *easy* or *difficult* for various reasons, of which number of edges is only one, and not the most important. Merely adding more edges to a drawing, for example by duplicating an existing feature, does not make it any more *difficult*.

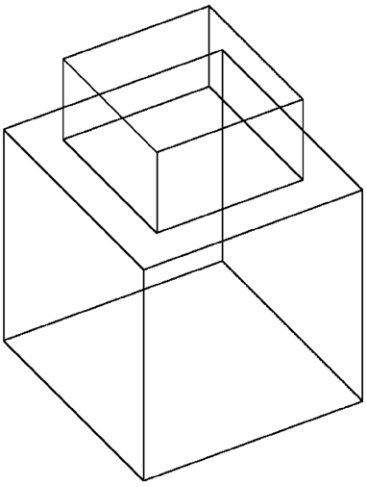

Fig. 28. Shelf stud.

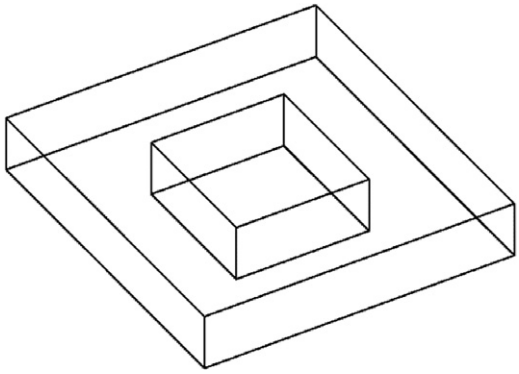

Fig. 29. Rectangular plate.

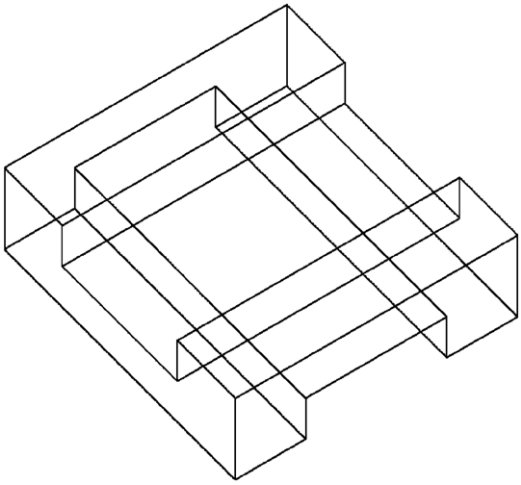

Fig. 30. Rectangular frame.

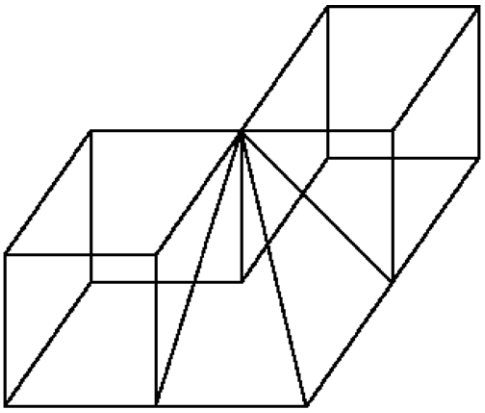

Fig. 31. Corner wedge.

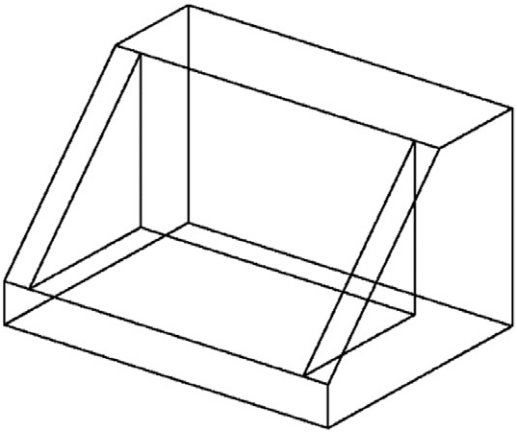

Fig. 32. Loader bucket.

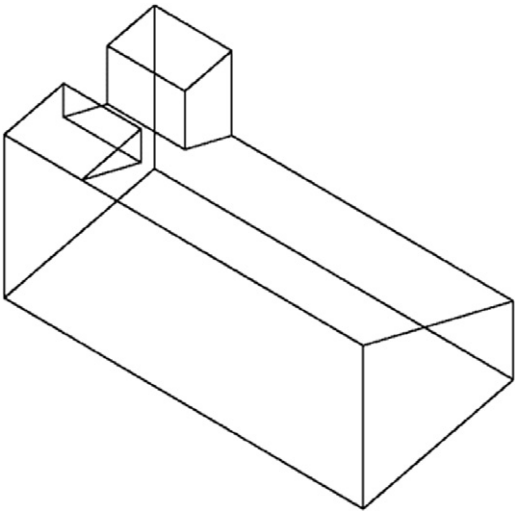

Fig. 33. Angle support.

For example, genus-zero graph-connected *normalons* and identifiable *semi-normalons* are *easy* because they can be inflated accurately even without face information [4]. Given such reliable geometric information, even where ambiguities exist, resolving them is unproblematic.

Similarly, trihedral genus-zero 3-connected graph-connected non-normalons are *easy* because there is only one possible interpretation. Any reasonable algorithm should find it.

We present our test set in increasing order of difficulty.

7.2. Simple objects

Fig. 24 is used in [3] to illustrate a particular problem with the simplest version of the Dijkstra's Algorithm approach.

Figs. 25–27 were used in Section 2 to illustrate the possibilities at tetrahedral vertices: Fig. 25 contains an internal face formed by a loop of tetrahedral vertices; Fig. 26 is topologically identical but the "internal" face is on the object convex hull; and Fig. 27 contains just a single tetrahedral vertex and no internal face.

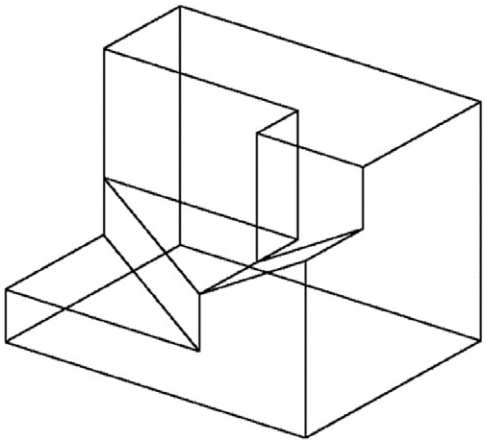

Fig. 34. Stop.

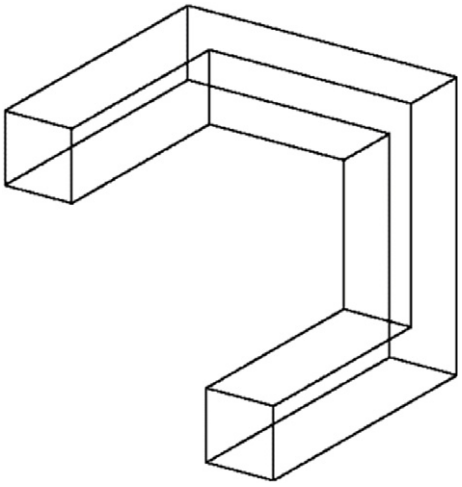

Fig. 35. Balancing arms.

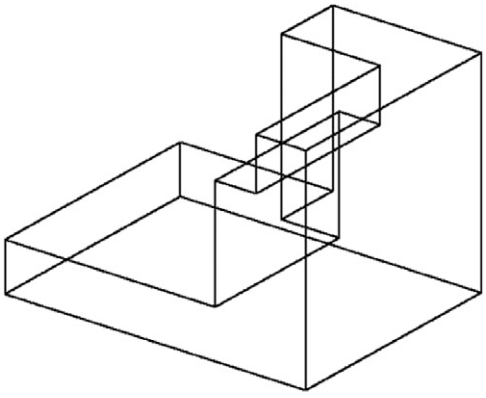

Fig. 36. Shoulder.

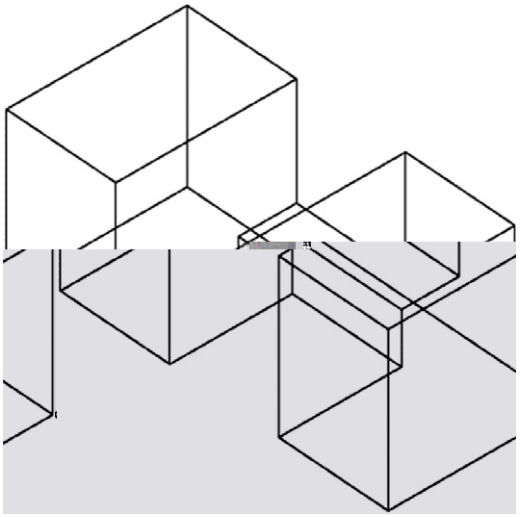

Fig. 37. Counterbalance.

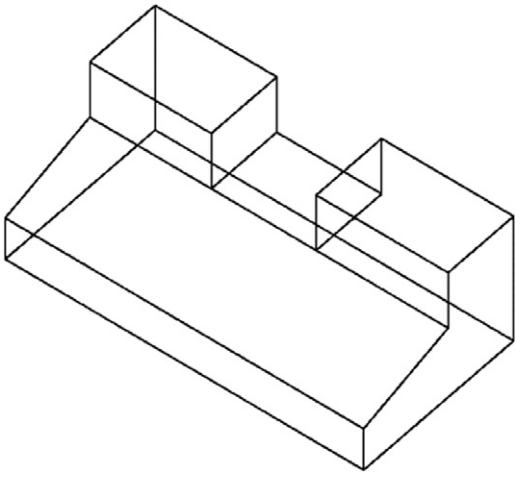

Fig. 38. Grooved bracket.

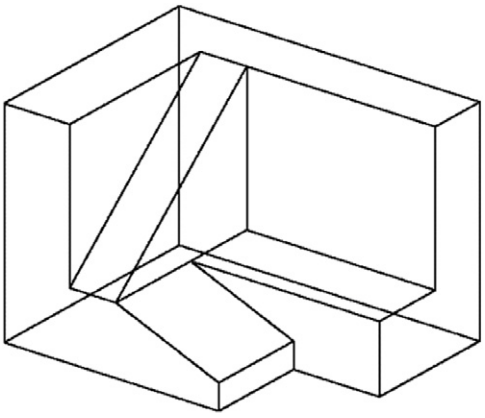

Fig. 39. Corner cover.

Figs. 28 and 29 show two possibilities for hole loops (boss and through hole). Fig. 30 shows an object with a through hole but no hole loop.

Note that Fig. 29 could also represent a pocket with a very thin bottom, or even a boss. As far as the output of our algorithm is concerned, the three interpretations are identical. Identifying when nested loops of edges are *intentionally* coplanar, and are thus two loops of the same face, is an open problem.

Fig. 31 is used in Section 1 to illustrate the definition of *face*: the two coplanar squares meeting at the central vertex are two faces, not one.

7.3. Objects from refer

Figs. 32–69 are adapted from the test set used by [4].

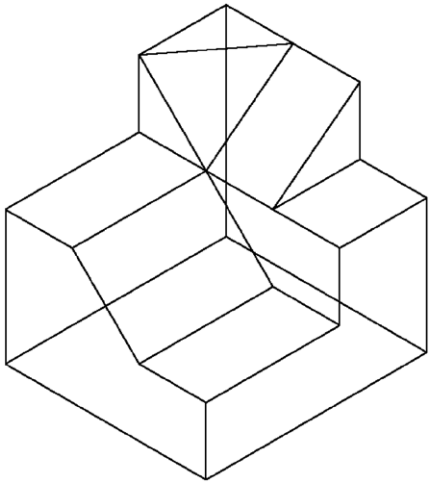

Fig. 40. Adjustable wedge.

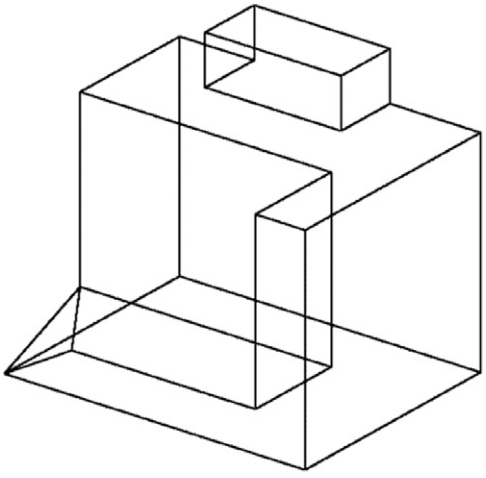

Fig. 43. Anchor.

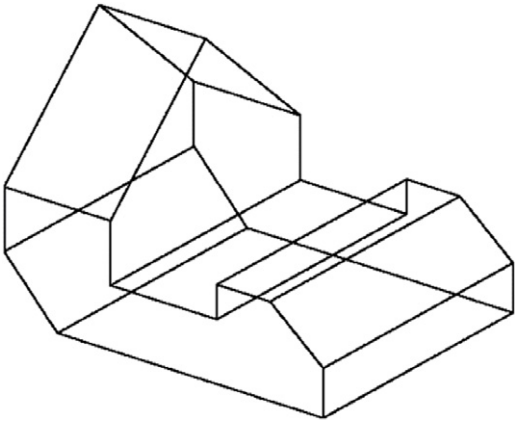

Fig. 41. Latch bolt.

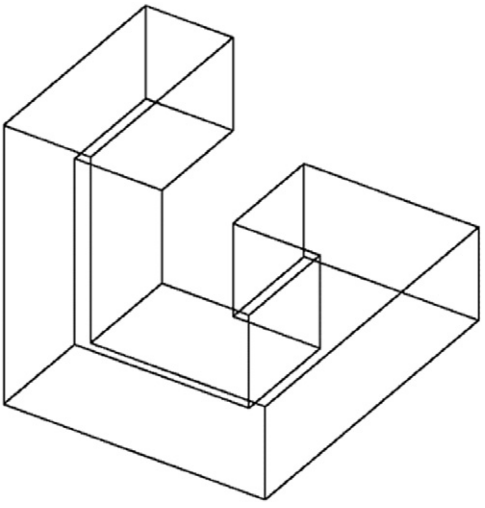

Fig. 44. Lateral bracing.

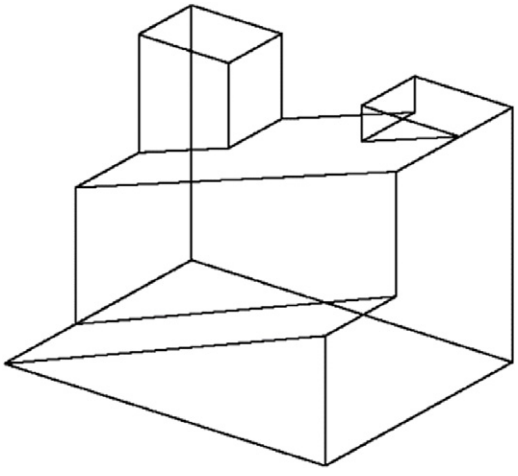

Fig. 42. Double wedge.

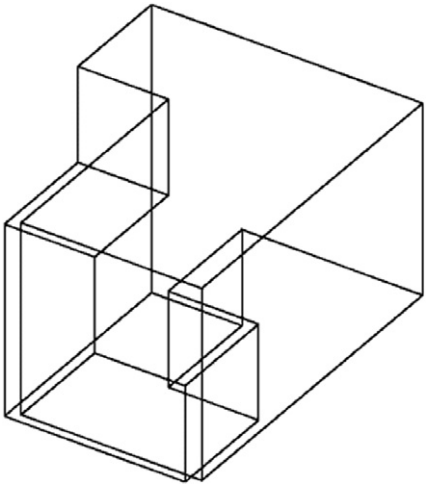

Fig. 45. Slotted block.

7.4. Objects from other collections

Figs. 70–73 are adapted from [44], which provides a test set for algorithms which process extended K-vertices.

Fig. 74 is adapted from [45], which investigates chained K-vertices. Since it portrays an object which is very difficult to reconstruct from a natural line drawing, successful reconstruction from a wireframe drawing is particularly useful.

Figs. 75 and 76 were inspired by similar drawings in [46].

Figs. 77–81 were inspired by drawings in [47].

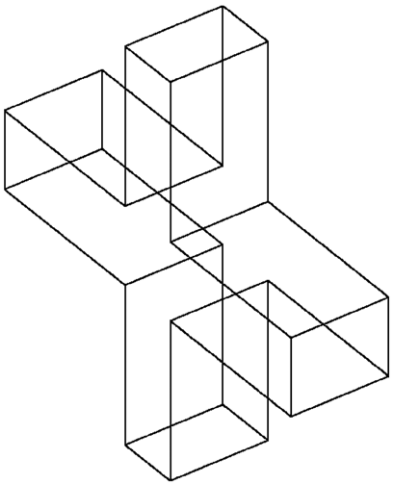

Fig. 46. Cross.

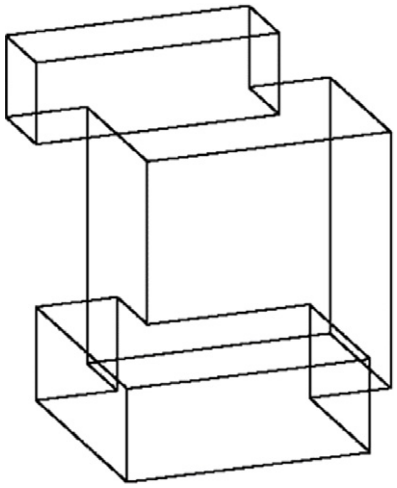

Fig. 49. Sliding block.

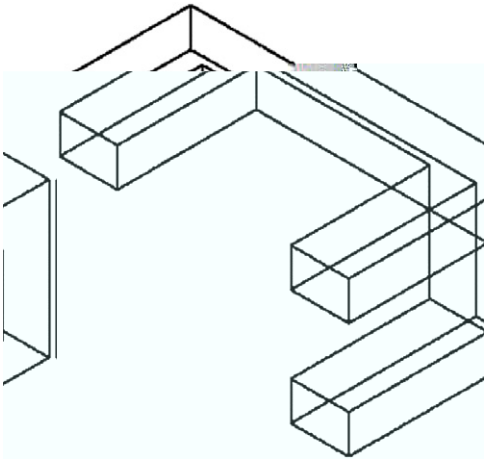

Fig. 47. Spacer.

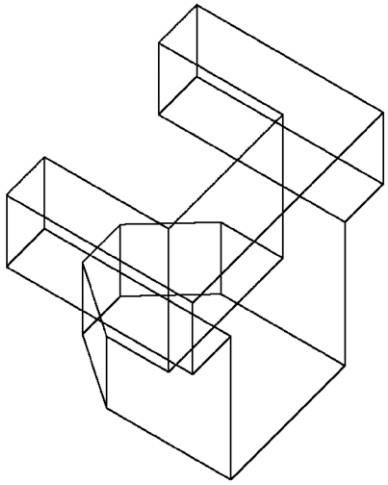

Fig. 50. Clip.

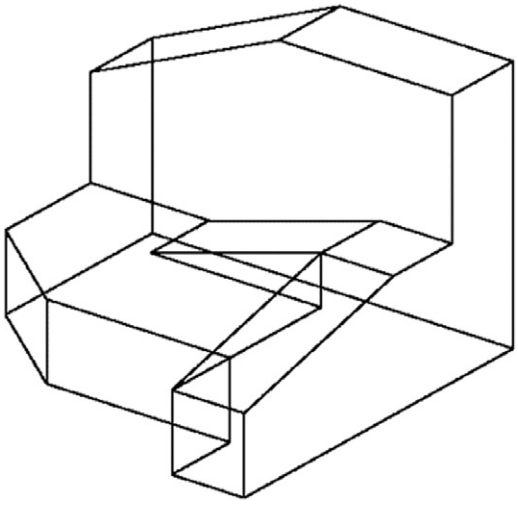

Fig. 48. Guide.

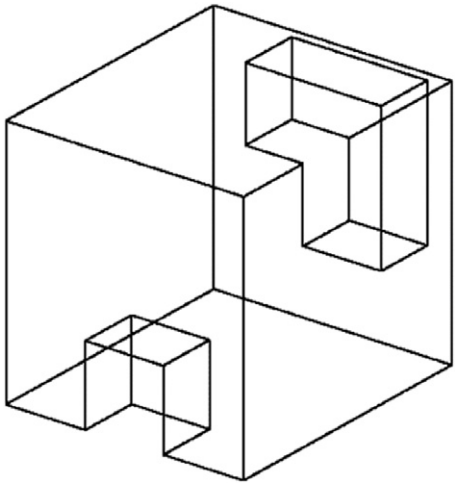

Fig. 51. Tool block.

Fig. 82 was inspired by a drawing in [48]. The option of constructing this object from its wireframe drawing is particularly important since it is difficult to construct it from a natural line drawing—it is impossible to tell whether the central feature is a through hole or a pocket.

Fig. 83 is adapted from [49]. It contains several junctions where three pairs of collinear lines intersect.

Fig. 84 is adapted from Escher's "Ascending and Descending" [50].

Figs. 85 and 86 are adapted from figures in the NIST Repository [51].

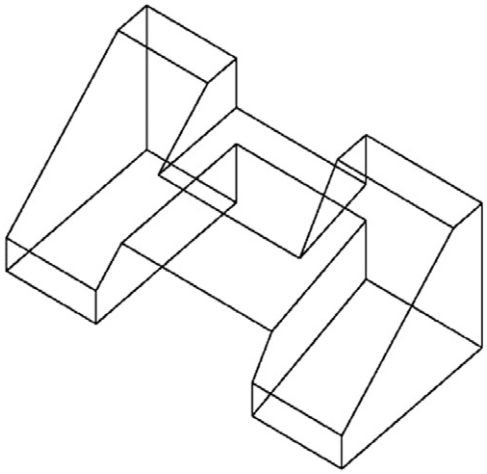

Fig. 52. Double bracket.

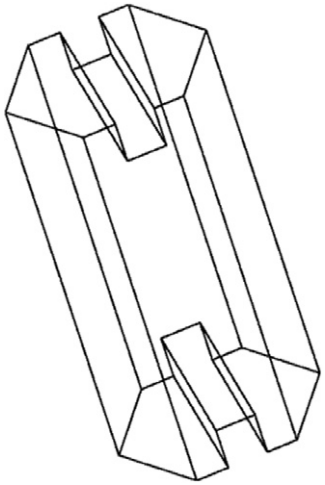

Fig. 53. Hexagonal rod.

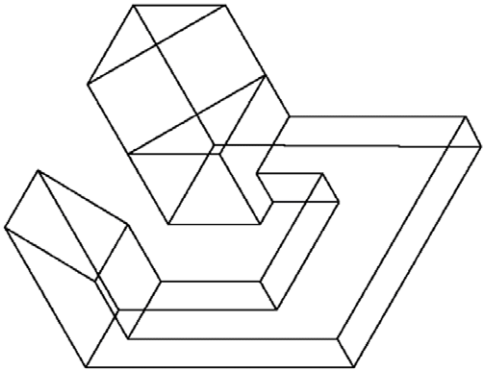

Fig. 54. Locking bracket.

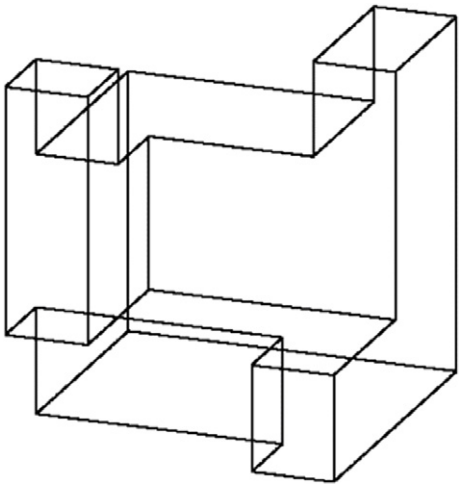

Fig. 55. Fixation plate.

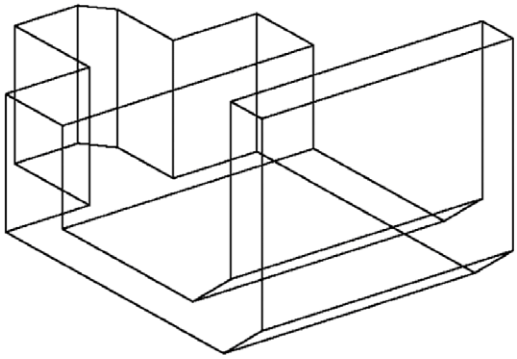

Fig. 56. Cover.

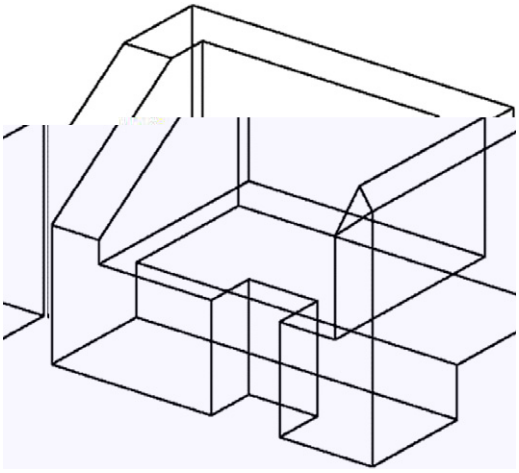

Fig. 57. Semi-housing.

7.5. Sequences of objects

The remaining drawings in our test set are grouped into sequences, in which later drawings in the sequence are produced by replicating features already present in the first drawing. Sequences of objects are particularly useful when assessing the practical time complexity of algorithms. Their inclusion in our test set also enabled us to confirm our earlier assertion in Section 4: it is the presence of particular features in the drawing, not its size, which causes Dijkstra's Algorithm to fail.

Figs. 87–89 form a sequence in which the same features are repeated. The first drawing in the sequence was inspired by [52].

The remaining three sequences have already been presented in Section 4. The five drawings in Fig. 13 and the three drawings in Fig. 14 were inspired by similar drawings in [25]. The eight drawings in Fig. 15 were inspired by similar drawings in [22,25]; the original is based on a photograph of a building.

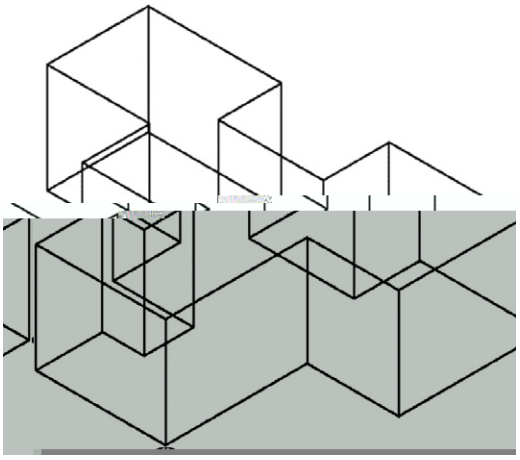

Fig. 58. Sliding hook.

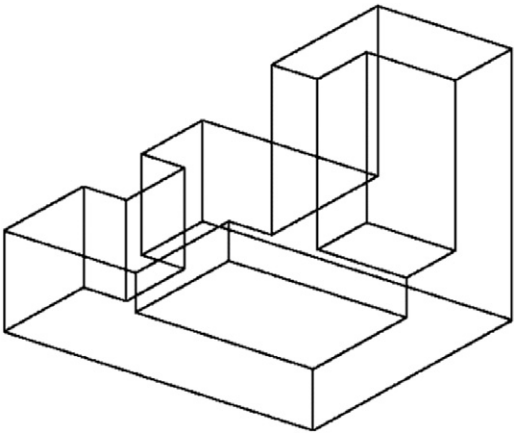

Fig. 59. Base block.

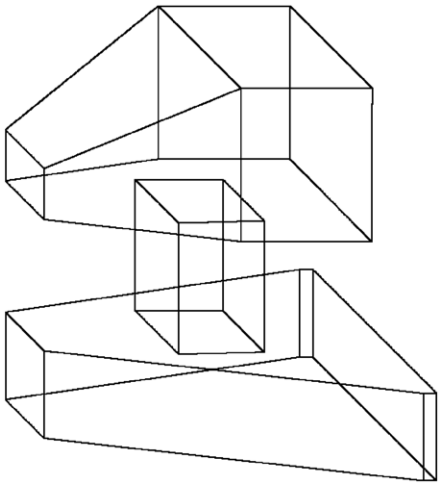

Fig. 60. Turret.

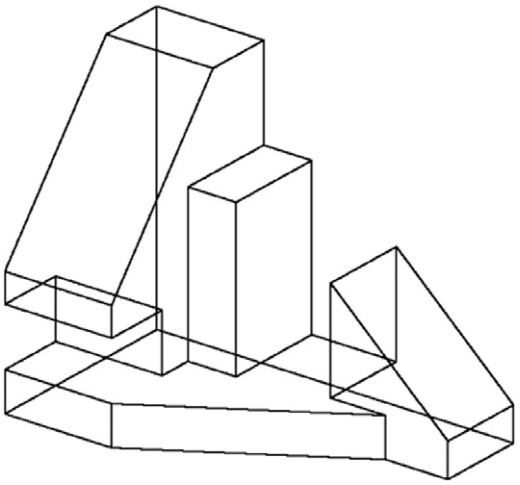

Fig. 61. Anchor base.

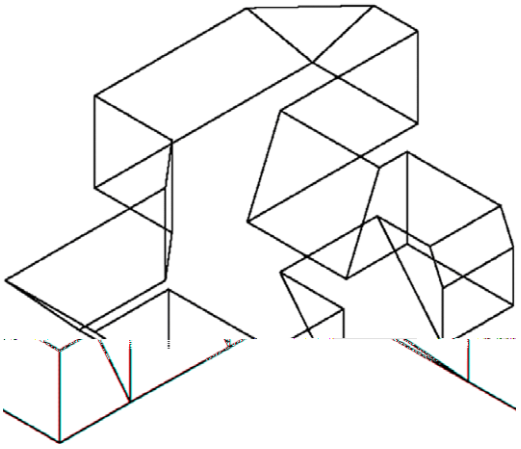

Fig. 62. Retainer plate.

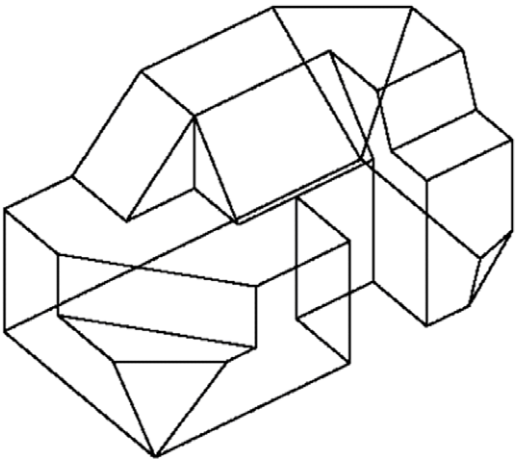

Fig. 63. Multiangle node.

8. Test results

This section presents the results of applying the algorithm described in Sections 5 and 6 to the test data in Section 7.

In order to show that our approach improves on the current state of the art, we must test it alongside an existing algorithm. We have chosen an approach based on Dijkstra's Algorithm as our point of comparison as it is popular, simple to implement and comparatively successful. Like our new algorithm, it exists in both 2D and 3D versions, with the differences between the two versions being encapsulated in a single function, the *path length*. Our implementation of this algorithm (described in detail in [36]) was used in Section 4 to assess the current state of the art.

Table 3
Results, Figs. 13–15, 23–89.

Fig. and drawing	Vertices	Edges	Faces	Dijkstra 2D	Dijkstra 3D	New 2D	New 3D
23: Cube	8	12	6	0.2	0.2	0.4	0.4
25: Pivot cushion	12	20	10	0.5	0.6	1.2	1.3
26: Anti-vibration pad	12	20	10	Fails	0.6	1.2	1.3
27: Hinge arm	15	23	10	0.7	0.8	1.5	1.7
29: Rectangular plate	16	24	12	0.4	0.6	1.5	1.5
28: Shelf stud	16	24	12	0.4	0.6	1.4	1.5
24: Offset cube	16	24	10	0.8	0.9	2.0	2.1
31: Corner wedge	15	26	13	0.7	0.9	1.9	2.3
32: Loader bucket	16	26	12	0.9	1.1	2.2	2.3
33: Angle support	19	29	12	1.2	1.4	3.0	3.1
34: Stop	19	30	13	1.3	1.5	2.9	3.6
35: Balancing arms	20	30	12	1.3	1.5	3.0	3.2
36: Shoulder	20	30	12	1.3	1.5	2.9	3.3
37: Counterbalance	20	30	12	1.3	1.6	3.1	3.7
38: Grooved bracket	20	31	13	1.4	1.6	3.5	3.7
39: Corner cover	20	33	15	1.6	1.8	4.7	5.4
87: Locking wedge 1	20	33	15	1.6	1.8	4.0	4.3
70: Cross plug	20	36	18	1.9	Fails	5.5	5.8
71: Roofs	20	36	18	1.9	Fails	5.7	6.0
40: Adjustable wedge	21	33	14	1.6	1.9	4.1	4.4
41: Latch bolt	22	33	13	1.6	1.9	4.1	4.3
42: Double wedge	22	34	14	1.7	2.0	4.2	4.5
43: Anchor	22	34	14	Fails	Fails	4.2	4.5
72: Angle setoff	22	35	15	2.0	2.2	4.7	5.1
73: Retainer mount	22	36	16	Fails	Fails	5.5	5.7
44: Lateral bracing	24	36	14	2.0	2.3	4.7	5.0
45: Slotted block	24	36	14	2.0	2.4	4.7	5.3
46: Cross	24	36	14	2.0	2.4	4.7	5.0
47: Spacer	25	39	16	2.4	2.8	6.2	6.6
48: Guide	26	41	17	Wrong	Fails	8.7	9.2
49: Sliding block	27	41	16	2.7	3.2	6.6	7.1
50: Clip	28	42	16	2.8	3.3	7.0	7.5
51: Tool block	28	42	16	2.9	3.4	8.0	8.3
52: Double bracket	28	42	16	2.8	3.4	6.8	7.3
53: Hexagonal rod	28	42	16	Fails	Fails	8.0	8.1
54: Locking bracket	28	43	17	3.0	Wrong	7.1	7.7
30: Rectangular frame	28	44	16	Fails	3.8	11.4	11.4
74: Lateral wedge	28	44	18	Wrong	Fails	9.9	10.3
55: Fixation plate	29	44	17	3.2	3.8	8.5	9.1
56: Cover	30	45	17	3.3	4.0	7.5	8.1
57: Semi-housing	31	47	18	3.7	4.6	10.0	10.7
58: Sliding hook	32	48	18	4.0	4.8	9.0	9.7
59: Base block	32	48	18	4.0	4.8	10.8	11.5
75: Table ironwork	32	48	20	2.7	3.4	7.9	8.5
60: Turret	32	49	23	1.6	Fails	7.2	8.0
88: Locking wedge 2	32	54	24	5.0	5.7	14.3	15.2
61: Anchor base	34	51	19	4.4	5.3	11.4	12.2
77: Milling clamp	34	51	19	Fails	5.5	12.3	14.1
62: Retainer plate	34	53	21	Fails	Fails	11.6	12.4
63: Multiangle node	35	55	22	Fails	Fails	16.7	17.2
64: Guide support	36	54	20	5.3	6.3	14.5	15.5
65: Sliding shuttle	36	54	20	5.5	6.5	15.5	16.6
78: Sliding plate	36	54	20	4.8	5.9	11.1	12.6
76: Housing	36	54	20	5.3	6.3	18.0	18.9
79: Pipe clamp	36	55	21	5.5	6.5	14.0	15.6
66: Plug	37	57	22	5.8	7.0	15.3	16.4
82: Base support	38	59	27	3.0	4.3	Wrong	16.2
83: Frame	40	72	36	Fails	Wrong	32.7	Wrong
89: Locking wedge 3	44	76	34	11.4	Fails	37.4	39.6
67: Clamp	46	70	26	9.9	11.9	Wrong	29.1
80: Channel	48	74	28	11.4	13.7	30.7	34.4
15a: Building 1	50	77	29	Fails	Fails	37.9	42.0
68: Rail	52	80	28	13.9	16.8	42.5	48.5
81: Rack	54	86	34	16.2	19.6	49.1	54.2
69: Anchor bracket	56	85	31	15.9	19.4	42.1	46.6
13a: Desk 1	60	90	30	18.8	23.3	56.7	68.7
84: Escher's tower	60	90	32	18.7	23.5	54.2	Wrong
14a: Steps 1	60	90	32	19.0	23.3	49.7	55.1
15b: Building 2	60	92	34	Fails	Fails	58.7	64.4
85: Seat	60	94	36	Fails	25.2	62.1	68.7
13b: Desk 2	68	102	34	25.7	31.8	71.8	91.5
14b: Steps 2	68	102	36	25.8	31.7	70.4	78.2
15c: Building 3	70	107	39	Fails	Fails	88.4	96.5
14c: Steps 3	72	108	38	29.9	37.0	81.1	87.9
13c: Desk 3	76	114	38	34.4	42.9	101.1	122.0

(*continued on next page*)

Table 3 (*continued*)

Fig. and drawing	Vertices	Edges	Faces	Dijkstra 2D	Dijkstra 3D	New 2D	New 3D
15d: Building 4	80	122	44	Fails	Fails	124.1	138.0
13d: Desk 4	84	126	42	45.0	56.2	136.9	160.6
15e: Building 5	90	137	49	Fails	Fails	174.1	189.0
13e: Desk 5	92	138	46	57.3	72.3	164.4	211.1
15f: Building 6	100	152	54	Fails	Fails	230.9	250.2
15g: Building 7	110	167	59	Fails	Fails	296.4	321.8
15h: Building 8	120	182	64	Fails	Fails	373.4	408.8
86: Gehaeuse	154	251	104	Fails	Fails	1087.1	Wrong

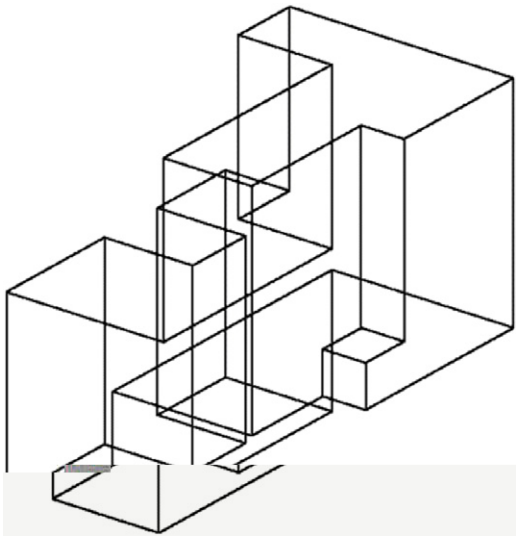

Fig. 64. Guide support.

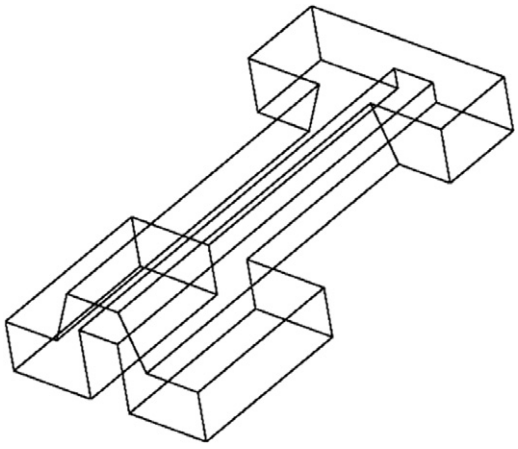

Fig. 65. Sliding shuttle

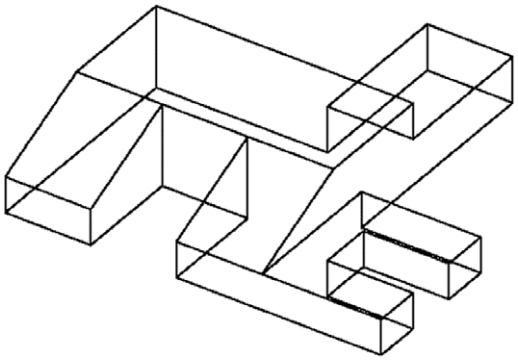

Fig. 66. Plug.

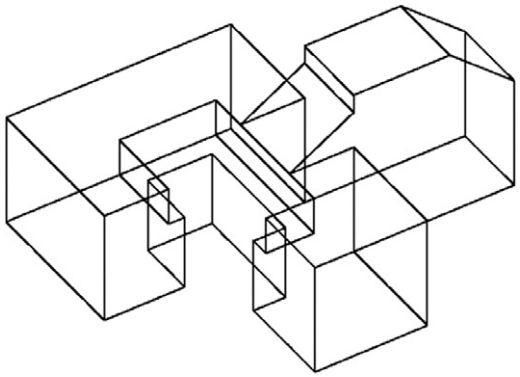

Fig. 67. Clamp.

8.1. Results and timings

In this section, we compare four possible approaches: the Dijkstra's Algorithm approach using 2D data, the Dijkstra's Algorithm approach using 3D data derived from inflation, our new approach using 2D data, and our new approach using 3D data derived from inflation. See Table 3. Where an algorithm succeeds, we give an average timing (the mean of 1000 iterations of the algorithm) in milliseconds. Where the algorithm fails, we distinguish *failure* (the algorithm incorrectly reports that it is not possible to find a consistent set of faces) and *wrong* (the algorithm finds a set of faces, but the wrong set).

Both variants of our new algorithm yield results for all drawings from Section 7: the 2D version yields correct results in all but 2 cases, and the 3D version in all but 3. By contrast, the Dijkstra's Algorithm approach works for almost all drawings with fewer than 30 edges, but becomes gradually less successful as the drawing size increases. For drawings with 50 or more edges, it fails as often as it succeeds, yielding no result, right or wrong. It does not give the correct results for any drawing with 150 or more edges.

In comparison with current non-shortest-path approaches, we can note that those drawings which Liu and Tang's [25] genetic algorithm usually processes correctly, the Desks (Fig. 13), Steps (Fig. 14) and Buildings (Fig. 15), are also processed correctly by our new algorithm. Liu and Tang do not report any cases where the genetic algorithm approach fails.

In analysing where our own algorithm fails, considering all of the failure cases would be excessive. However, consideration of those cases where the new algorithm fails but Dijkstra's Algorithm succeeds may help to identify potential further improvements.

The failure with Fig. 67 (the clamp) is caused by a poor choice of arbitrary concatenation on the first call of the "examine hypotheses" subroutine in Section 5.8. Consider Fig. 90a, taken from the top right-hand corner of Fig. 67. Vertex 0, touching two quadrilateral loops, is as good a starting-point as any; this gives us the three strings 1–0–2, 2–0–4 and 4–0–1 (Fig. 90b). At the start of the "examine hypotheses" subroutine, string 1–0–2 is chosen as the highest-priority, and the question arises: should it be concatenated with 2–3 or with 2–6? Neither is close to being

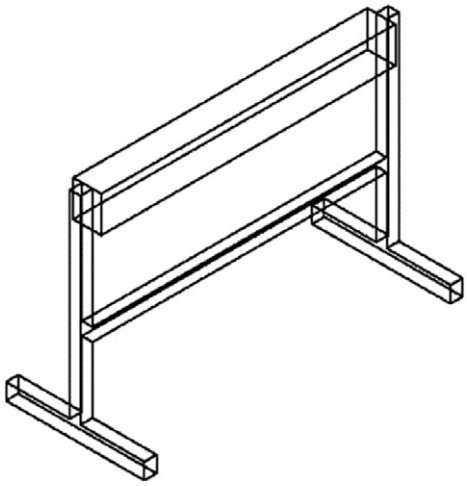

Fig. 68. Rail.

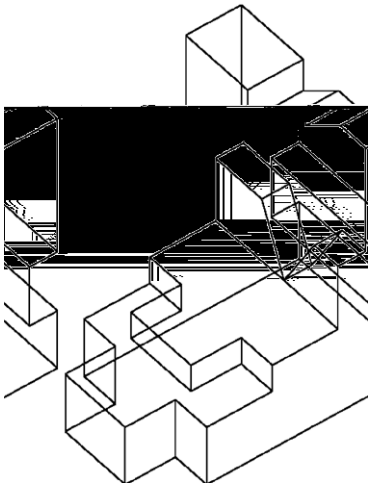

Fig. 69. Anchor bracket.

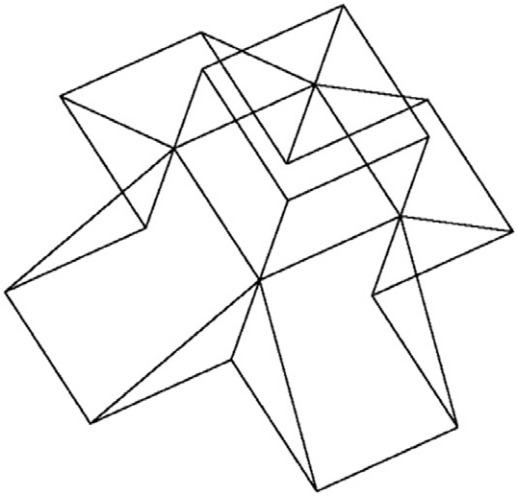

Fig. 70. Cross plug.

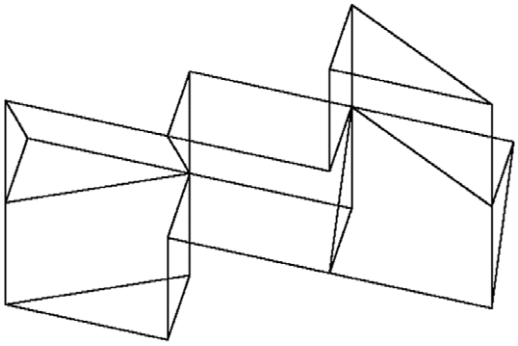

Fig. 71. Roofs.

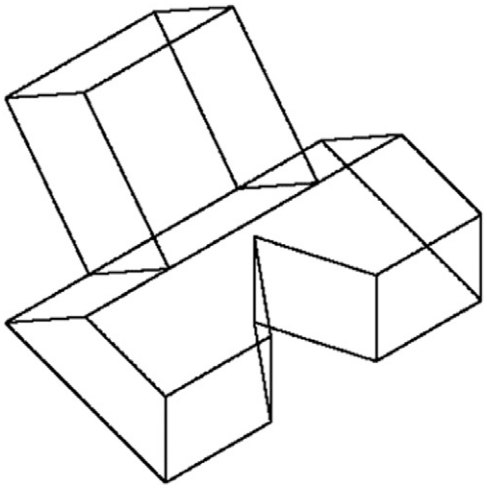

Fig. 72. Angle setoff.

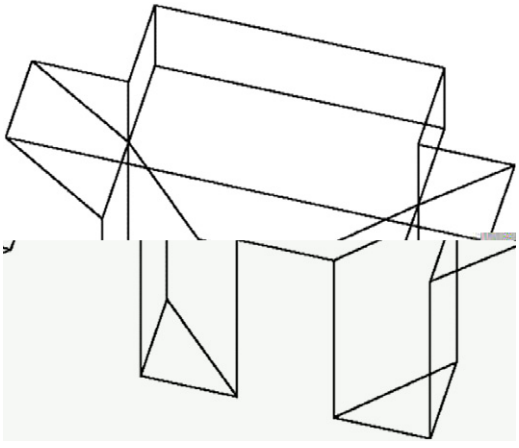

Fig. 73. Retainer mount.

parallel to any edge in 1–0–2, and, unfortunately, the wrong string is chosen (Fig. 90c). When a complete face is found, the working list is discarded and this wrong hypothesis is deleted, but the face found as a consequence of examining the hypothesis is 2–3–7–6 (Fig. 90d). Although this is a correct face of the object, it is in the wrong alignment with respect to the initial merging of strings in Fig. 90b, with the result that the next "'face" of the object found is 1–0–2–6–4–5 (Fig. 90e).

Clearly, if either 2–0–4 or 4–0–1 had been chosen as highest-priority instead, the algorithm would have found an edge parallel to one already in the chosen string, and this problem would not have occurred. This suggests two possible lines of investigation: either introduce a more sophisticated priority mechanism than the simple one given in Section 6.1, or consider possible mating values as well as just string priority when selecting the string to be extended at the start of the "examine hypotheses" subroutine.

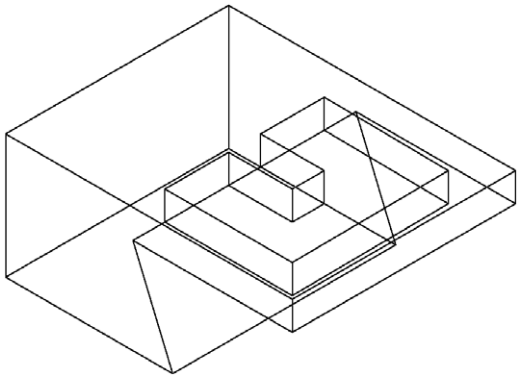

Fig. 74. Lateral wedge.

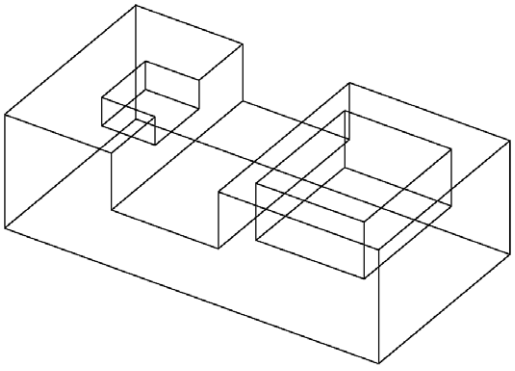

Fig. 75. Table Ironwork.

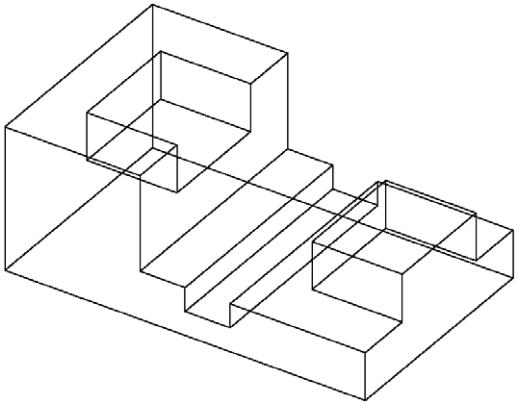

Fig. 76. Housing.

The failure with Fig. 82 (the base support) is caused by a poor choice of concatenation of strings when both candidates for concatenation with the highest-priority string are parallel to an edge already in the string. See Fig. 91, which shows the relevant edges of Fig. 82. The error occurs after eight correct faces of the object have been identified. At this stage, the highest-priority string remaining in the main list is 0–1–2. The question arises: should 0–1–2 be concatenated with 2–3 or with 2–4? Both are parallel to an edge (0–1) already included in the string, so the mating values are the same, and unfortunately the wrong candidate is chosen.

This problem could be avoided by adding more sophistication to the calculation of mating values (Section 6.2), for example by decreasing a mating value if parallel edges are not in the same quadrilateral loop (at present, quadrilateral loops contribute only to priority, not to mating values).

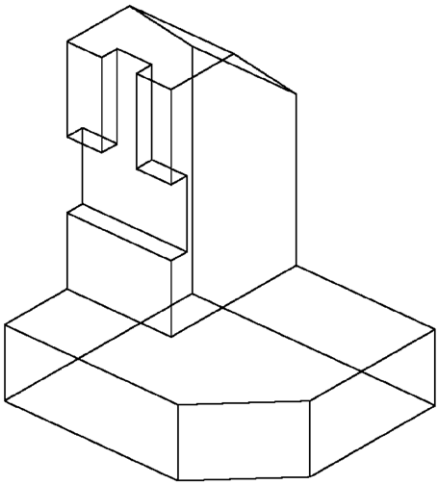

Fig. 77. Milling clamp.

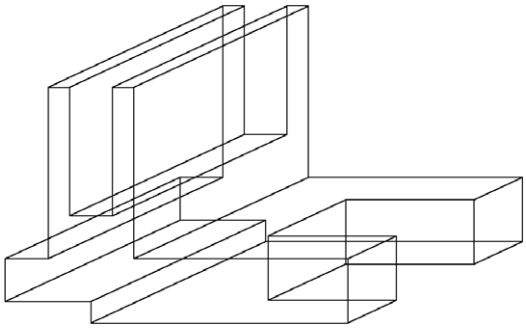

Fig. 78. Sliding plate.

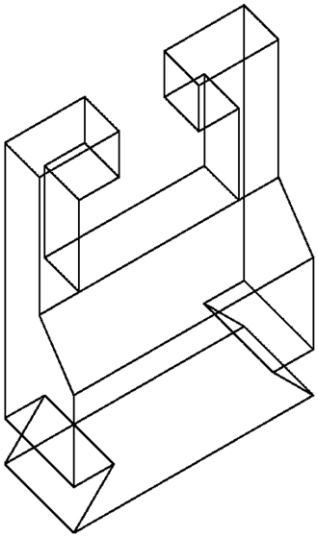

Fig. 79. Pipe clamp.

The third such failure case, Fig. 84 (Escher's Tower) when using 3D data, is less enlightening, as inflating the 2D drawing to 3D produces counterintuitive results.

8.2. Time complexity of algorithm

With respect to the time taken by our algorithm, four things are worth considering: the theoretical time complexity, the practical

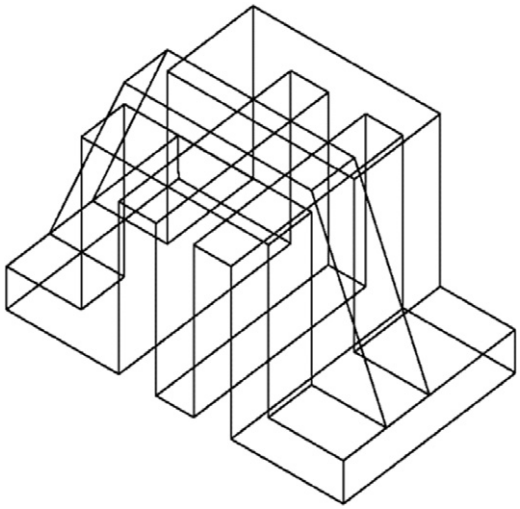

Fig. 80. Channel.

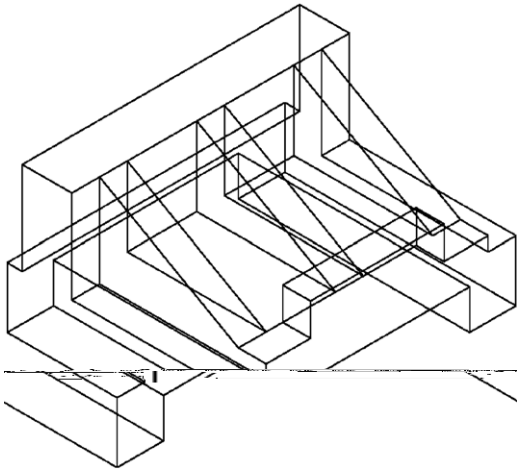

Fig. 81. Rack.

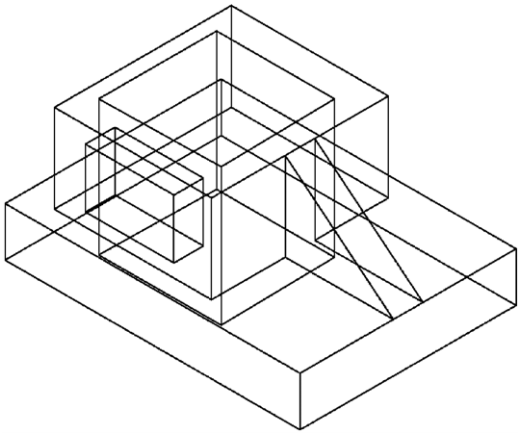

Fig. 82. Base support.

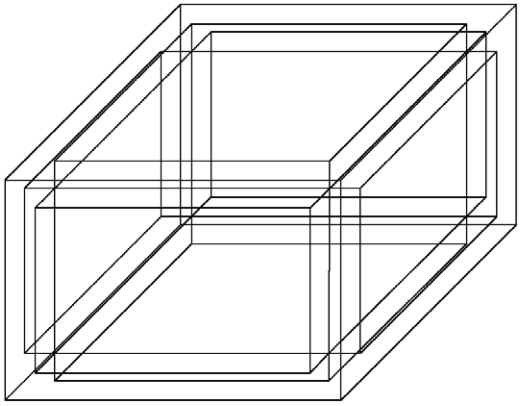

Fig. 83. Frame.

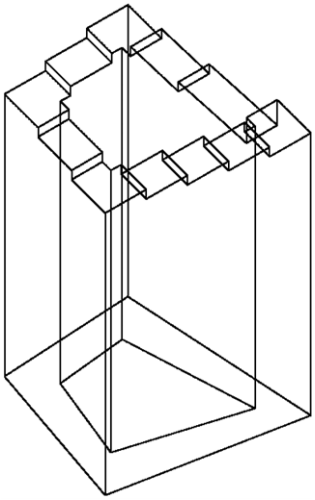

Fig. 84. Escher's tower.

time complexity, the actual time taken to process a realistic example, and how our algorithm compares with its competitors.

The outline of the algorithm in Section 5 shows it to be polynomial. A simplistic assessment based on counting loops would suggest a worst-case performance of $O(n^5)$.

In order to determine practical time complexity, we applied linear regression to the timings for processing the "Building" sequence using our new algorithm. This gives $e^{2.683}$ for the 2D version and $e^{2.664}$ for the 3D version (where e is the number of edges).

In terms of actual time taken, Dijkstra's Algorithm is noticeably faster for all drawings. However, this is due to its low time constant. Our new algorithm actually has a *better* practical time complexity, despite having a theoretically worse worst-case order.

When applied to our most complex drawing, Fig. 86, with 251 edges, the algorithm took slightly more than one second, making it fast enough for use in interactive systems. We can thus conclude that speed is not a problem.

8.3. Reliability and robustness

Our preliminary testing, described in Section 8.1, suggests that our new approach is more reliable than the traditional approach using Dijkstra's Algorithm (although it may be worth noting that neither method guarantees a solid obeying Euler's formula).

However, all of the drawings in Section 7 are close to being accurate—lines intended to be parallel are close to being parallel. Real drawings are not always so accurate. In order to test how sensitive our algorithm is to inaccurate drawings, we progressively distorted Fig. 69 by moving vertices in arbitrary directions until the algorithm failed to interpret it correctly. The results are shown in

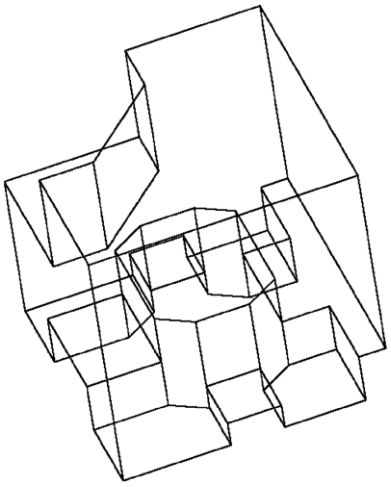

Fig. 85. Seat.

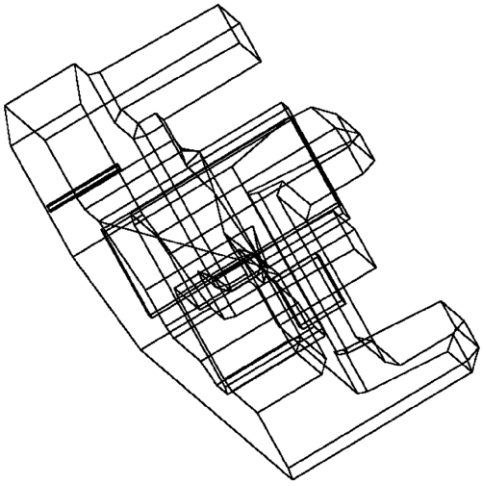

Fig. 86. Gehaeuse.

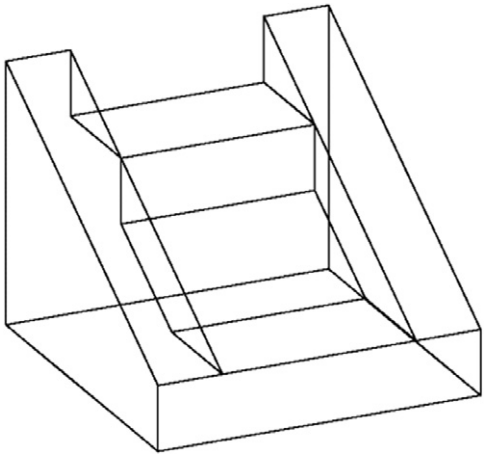

Fig. 87. Locking wedge 1.

Figs. 92 (the last one to be interpreted correctly) and 93 (the first not to be interpreted correctly).

We believe that Fig. 93 is rather more badly drawn than anything a design engineer would produce in practice, so we can reasonably claim that our algorithm is sufficiently tolerant of drawing errors.

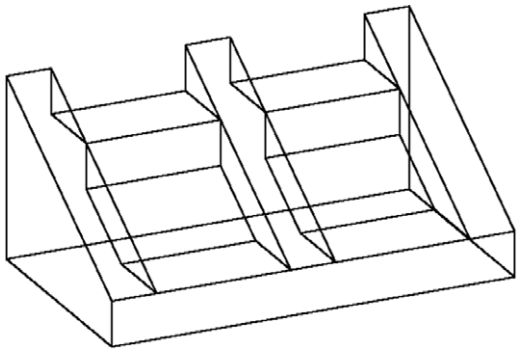

Fig. 88. Locking wedge 2.

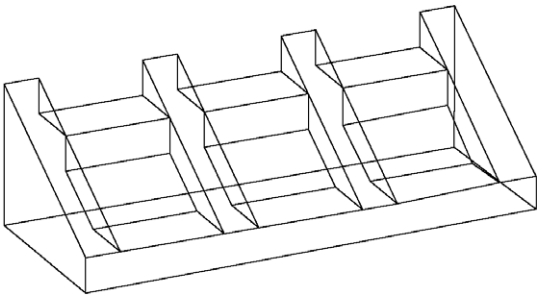

Fig. 89. Locking wedge 3.

8.4. Sensitivity to choice of arbitrary constants

In Section 4.1, we introduced three arbitrary constants, $\boldsymbol{T}$ and $\boldsymbol{Q}$, used in setting the initial string priorities, and $\boldsymbol{r}$, used in calculating the resulting priority after two strings are mated. The values chosen were $\boldsymbol{T} = \mathbf{50}$, $\boldsymbol{Q} = \mathbf{10}$, and $\boldsymbol{r} = \mathbf{0.4}$.

The results obtained in Section 6.1 remain valid for a wide range of values of $\boldsymbol{T}$, $\boldsymbol{Q}$ and $\boldsymbol{r}$. We have found no combination which achieves better results.

Varying $\boldsymbol{T}$ while $\boldsymbol{Q}$ and $\boldsymbol{r}$ remain constant, we obtain the same results as in Section 6.1 for values in the range $\boldsymbol{T} = \mathbf{30}$ to $\boldsymbol{T} = \mathbf{90}$. Higher or lower values of $\boldsymbol{T}$ introduced additional errors.

Varying $\boldsymbol{Q}$ while $\boldsymbol{T}$ and $\boldsymbol{k}$ remain constant, we obtain the same results as in Section 6.1 for values in the range $\boldsymbol{Q} = \mathbf{5}$ (the lowest value tested) to $\boldsymbol{Q} = \mathbf{15}$. Higher values of $\boldsymbol{Q}$ introduced additional errors unless $\boldsymbol{T}$ was also increased.

Varying $\boldsymbol{r}$ while $\boldsymbol{T}$ and $\boldsymbol{Q}$ remain constant, we obtain the same results as in Section 6.1 for values in the range $\boldsymbol{r} = \mathbf{0.2}$ to $\boldsymbol{r} = \mathbf{0.45}$. Higher or lower values of $\boldsymbol{r}$ introduced additional errors.

In general, we obtain good results with $\boldsymbol{T} < \mathbf{100}$, $\boldsymbol{T} > \mathbf{2Q}$ and $\mathbf{0.25} \leq \boldsymbol{r} \leq \mathbf{0.4}$.

The first two results are what we might expect. Firstly setting the value of $\boldsymbol{T}$ too high could result in some edges not being considered at all, and since the base priority for edges is set to 100, a maximum priority of <200 seems reasonable.

Secondly, since (for reasons already discussed) we wish to process edges touching triangular loops before those touching any number of quadrilateral loops, the requirement that $\boldsymbol{T} > \mathbf{2Q}$ is to be expected.

In our original testing, we used $\boldsymbol{r} = \mathbf{0.5}$, which effectively means that as long as we are mating good strings, we continue working on the same face. That $\boldsymbol{r} \leq \mathbf{0.4}$ is to be preferred indicates that it is important to work on several faces at once, even when work on our preferred face seems to be going well.

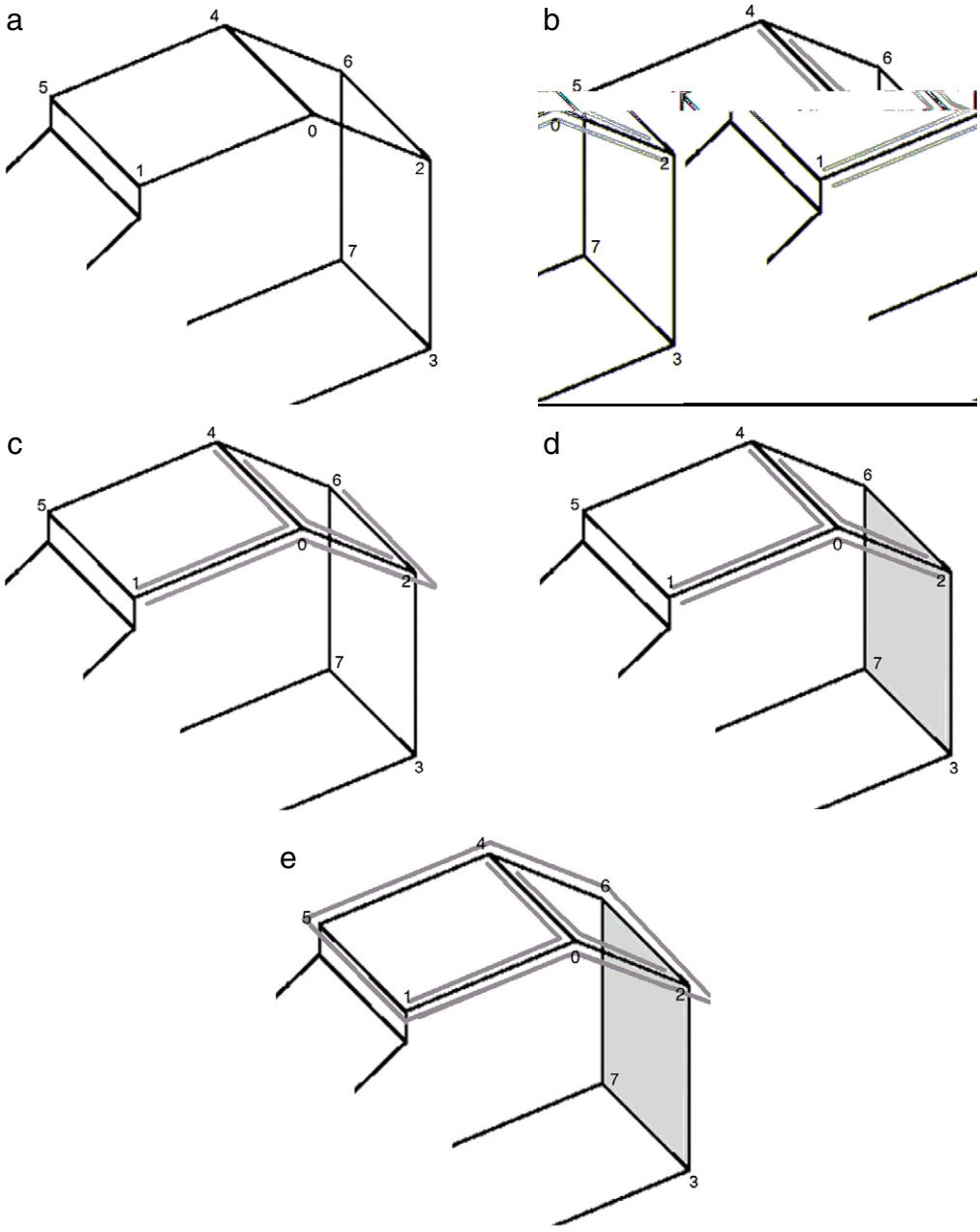

Fig. 90. Part of clamp.

9. Conclusions and recommendations

Current face-finding algorithms generally fall into four categories: exhaustive search approaches, pseudo-random approaches, shortest-path approaches and heavily mathematical approaches.

Exhaustive approaches, whereby all possible answers to the problem are generated and evaluated according to some assessment criterion, have been tried many times, most notably by Liu, Lee and Cham [8]. The unavoidable problem is that they can be slow as the search space being explored is exponential in the number of edges.

The use of pseudo-random approaches such as genetic algorithms (Liu and Tang [25]) seems to us to be avoiding the problem. There is a superficial appeal to the idea of sealing all the ingredients into a can, shaking it, and opening the can to find a finished solution. However, even if the finished solution is the correct one (which we can never guarantee), we gain no insight into the problem by doing this.

Shortest-path approaches appear to be the most promising. However, the most popular, a fast and easily-implemented approach based on Dijkstra's Algorithm, is theoretically suspect (and, occasionally, wrong in practice) since the underlying assumptions of Dijkstra's Algorithm do not match the problem at hand.

Li's Algorithm [11] is not only theoretically sounder but also faster, but the underlying concepts are difficult to understand and it has not, as yet, been reproduced by other researchers or incorporated in a system which chooses between possible alternative solutions.

Moving to our new algorithm, our test results support the following conclusions:

Our new approach is more reliable than the traditional approach using Dijkstra's Algorithm. It is at least as reliable as the genetic algorithm method proposed by Liu and Tang [25].

Our new approach is somewhat slower than the traditional approach using Dijkstra's Algorithm, usually by a factor of between

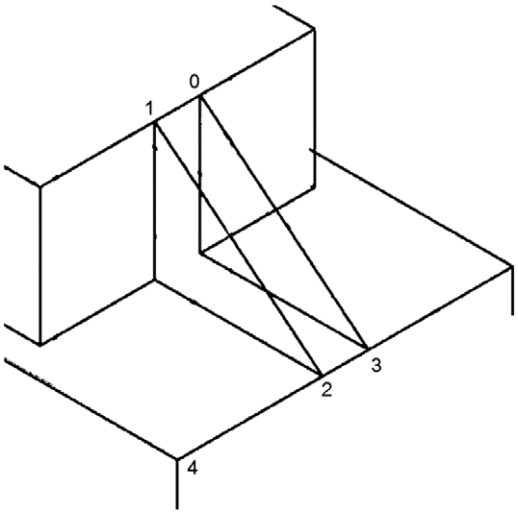

Fig. 91. Part of base support

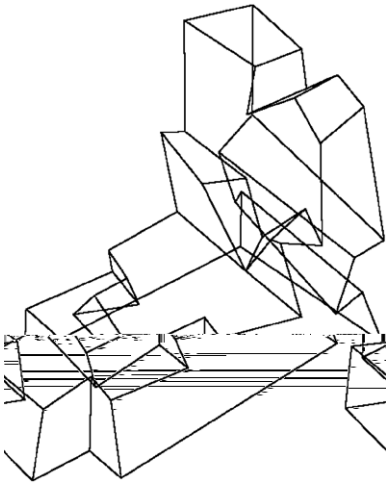

Fig. 92. Interpreted correctly.

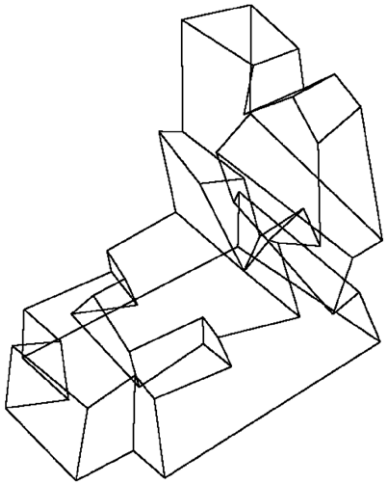

Fig. 93. Misinterpreted.

2.5 and 3. However, most of the difference is in the time constant, and our new algorithm has a slightly better practical time complexity. Both approaches are faster than the genetic algorithm method proposed by Liu and Tang [25] and are fast enough for use in interactive systems. It is believed that Li's Algorithm [11] is faster still, but no direct comparisons have been made.

Our new approach is tolerant of reasonable drawing inaccuracies.

With regard to the question: is it better to find wireframes before inflation, using only the 2D wireframe data, or after inflation, using hypothetical 3D data, for neither the Dijkstra's Algorithm nor our own new approach do we have a definitive answer. The best we can say is that finding faces before inflation is less sensitive to implementation choices (e.g. compliance functions used in inflation; tuning constants used in assessing parallelism), and is perhaps to be preferred for that reason.

Acknowledgements

The support of the Japan Society for the Promotion of Science (Scholarship no P03717), the Ramon y Cajal Scholarship Programme, the Spanish Ministry of Science and Education, and the European Union (Project DPI2007-66755-C02-01) is acknowledged with gratitude.

We also acknowledge gratefully the assistance of Ana Piquer, who created some of the test drawings in Section 7.

References

[1] Company P, Piquer A, Contero M, Naya F. A survey on geometrical reconstruction as a core technology to sketch-based modeling. Computers & Graphics 2005;29(6):892–904.
[2] Kyratzi S, Sapidis N. Extracting a polyhedron from a single-view sketch: Topological construction of a wireframe sketch with minimal hidden elements. Computers & Graphics 2009;33(3):270–9.
[3] Varley PAC. Automatic creation of boundary-representation models from single line drawings. Ph.D. thesis. University of Wales; 2003.
[4] Company P, Contero M, Conesa J, Piquer A. An optimisation-based reconstruction engine for 3D modelling by sketching. Computers & Graphics 2004;28(6):955–79.
[5] Markowsky G, Wesley MA. Fleshing out wire frames. IBM Journal of Research and Development 1980;24(5):582–97.
[6] Ganter MA, Uicker Jr JJ. From wire-frame to solid geometric: Automated conversion of data representations. Computers in Mechanical Engineering 1983;2(2):40–5.
[7] Courter SM, Brewer JA. Automated conversion of curvilinear wire-frame models to surface boundary models: A topological approach. Communications of ACM 1986;20(4):171–8.
[8] Liu J, Lee YT, Cham WK. Identifying faces in a 2D line drawing representing a manifold object. IEEE Transactions on Pattern Analysis and Machine Intelligence 2001;24(12):1579–93.
[9] Sugihara K. Machine interpretation of line drawings. MIT Press; 1986.
[10] Heyden A. On the consistency of line drawings obtained by projections of piecewise planar objects. Journal of Mathematical Imaging and Vision 1996;6:393–412.
[11] Li H. nD polyhedral scene reconstruction from single 2D line drawing by local propagation. Lecture Notes in Artificial Intelligence 2006;3763:169–97.
[12] Li H, Zhao L, Chen Y. Polyhedral scene analysis and parametric propagation with calotte analysis. Lecture Notes in Computer Science 2005;3519:383–402.
[13] Necker LA. Observations on some remarkable optical phenomena seen in Switzerland. Philosophical Magazine Third Series 1832;1(5):329–37.
[14] Cooper MC. Wireframe projections: Physical realisability of curved objects and unambiguous reconstruction of simple polyhedra. International Journal of Computer Vision 2005;64(1):69–88.
[15] Chekuri CS, Goldberg AV, Karger DR, Levine MS, Stein C. Experimental study of minimum cut algorithms. In: Proceedings of the eighth annual ACM-SIAM symposium on discrete algorithms. 1997. p. 324–33.
[16] Brinkmeier M. A simple and fast min-cut algorithm. Theory of Computing Systems archive 2007;41(2):369–80.
[17] Piquer A, Company P, Martin RR. Skewed mirror symmetry in the 3d reconstruction of polyhedral models. Journal of Winter School on Computer Graphics 2003;11(3):504–11.
[18] Varley PAC, Martin RR, Suzuki H. Frontal geometry from sketches of engineering objects: Is line labelling necessary? Computer Aided Design 2005;37(12):1285–307.
[19] Idesawa M. A system to generate a solid figure from a three view. Bulletin of the Japan Society of Mechanical Engineers 1973;16(Feb.):216–25.
[20] Ungvichian V, Kanongchaiyos P. Mapping a 3-D model into abstract cellular complex format. Computer-Aided Design and Applications 2006;3(1–4):395–404.

[21] Inoue K, Shimada K, Chilaka K. Solid model reconstruction of wireframe CAD models based on topological embeddings of planar graphs. Journal of Mechanical Design 2003;125(3):434–42.
[22] Shpitalni M, Lipson H. Identification of faces in a 2D line drawing projection of a wireframe object. IEEE Transactions on Pattern Analysis and Machine Intelligence 1996;18(10):1000–12.
[23] Liu J, Lee YT. A graph-based method for face identification from a single 2D line drawing. IEEE Transactions on Pattern Analysis and Machine Intelligence 2001;23(10):1106–10.
[24] Oh BS, Kim CH. Progressive reconstruction of 3D objects from a single free-hand line drawing. Computers & Graphics 2003;27(4):581–92.
[25] Liu J, Tang X. Evolutionary search for faces from line drawings. IEEE Transactions on Pattern Analysis and Machine Intelligence 2005;27(6): 861–72.
[26] Varley PAC, Martin RR. Constructing boundary representation solid models from a two-dimensional sketch: Topology of hidden parts. In: Proc. first UK-Korea workshop on geometric modeling and computer graphics. Kyung Moon Publishers; 2000. p. 129–44.
[27] Dijkstra EW. A note on two problems in connexion with graphs. Numerische Mathematik I 1959;269–71.
[28] Shesh A, Chen B. SMARTPAPER: An interactive and user friendly sketching system. Computer Graphics Forum 2004;12(3):301–10.
[29] Masry M, Kang DJ, Susilo I, Lipson H. A freehand sketching interface for progressive construction and analysis of 3d objects. In: Proc. AAAI Fall Symposium on Making Pen-Based Sketching Intelligent and Natural. 2004.
[30] Beohar A. A hybrid approach to reconstruct 3D solids from 2D near isometric sketches. M.Sc. dissertation. Kanpur: Indian Institute of Technology; 2005.
[31] Prim RC. Shortest connection networks and some generalisations. Bell System Technical Journal 1957;36:1389–401.
[32] Lamb D, Bandopadhay A. Interpreting a 3D object from a rough 2D line drawing. In: Proc. of Visualization'90. 1990. p. 59–66.
[33] Suntoro A, da Luz Vieira J, Singh B. Face recognition of a wireframe polyhedra using fundamental cycles. In: Proc. TENCON'94 1. 1994. p. 230–33.
[34] Martin RR, Dutta D. Tools for asymmetry rectification in shape design. Technical report UM-MEAM-94-16. University of Michigan, Dept. of Mechanical Engineering; 1994.
[35] Sun Y, Lee YT. Topological analysis of a single line drawing for 3D shape recovery. In: Proc. 2nd international conference on computer graphics and interactive techniques in Australasia and South Asia. 2004. p. 167–172.
[36] Varley PAC. An implementation of Dijkstra's algorithm for finding faces in wireframes, Technical Note Regeo 04, Universidad Jaume I; 2009. http://www.regeo.uji.es/publicaciones/regeo04.pdf.
[37] Wikipedia. http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm.
[38] Bellman R. On a routing problem. Quarterly of Applied Mathematics 1958; 16(1):87–90.
[39] Ford jr LR, Fulkerson DR. Flows in networks. Princeton University Press; 1962.
[40] Floyd RW. Algorithm 97: Shortest path. Communications of the ACM 1962; 5(6):345.
[41] Warshall S. A theorem on boolean matrices. Journal of the ACM 1962;9(1): 11–2.
[42] Sugihara K. Resolvable representations of polyhedra. Discrete and Computational Geometry 1999;21(2):243–55.
[43] Varley PAC. Implementing the new algorithm for finding faces in wireframes. Technical Note Regeo 05, Universidad Jaume I. http://www.regeo.uji.es/publicaciones/regeo05.pdf; 2009.
[44] Varley PAC. Problems for line labelling: A test set of drawings of objects with higher-valency vertices. International Journal of CAD/CAM 2005;5(1).
[45] Varley P, Suzuki H, Martin RR. Interpreting line drawing of objects with k-vertices. In: S.-M. Hu, H. Pottmann (Eds.). Proc. Geometric Modeling and Processing 2004. 2004 p. 249–358. ISBN 0769520782.
[46] Meeran S, Taib JM. A generic approach to recognising isolated, nested and interacting features from 2D drawings. Computer-Aided Design 1999;31(14): 891–910.
[47] Yankee HW. Engineering graphics. Prindle, Weber and Schmidt; 1985.
[48] Pickup F, Parker MA. Engineering drawing with worked examples, vol. 1. 3rd ed. Hutchison and Co; 1979.
[49] Wang W. On the automatic reconstruction of a 3d object's constructive solid geometry representation from its 2d projection line drawing. D.Sc. dissertation. University of Massachusetts at Lowell; 1992.
[50] Escher MC. Ascending and descending, http://www.mcescher.com/Gallery/recogn-bmp/LW435.jpg; 1960.
[51] NIST Repository. http://gams.nist.gov; 2005.
[52] Shirai Y. Three-dimensional computer vision. Springer-Verlag; 1987.

Peter A.C. Varley holds the degrees of BA Hons and MA in Chemistry (Oxford, 1985), M.Sc. in Energy (University of Wales, 1996) and PhD in Computer Science (University of Wales, 2003). He was for two years a JSPS Research Fellow at the University of Tokyo and is now a Ramon y Cajal Research Fellow at Universidad Jaume I in Castellón de la Plana, Spain. He has published approximately twenty papers, mostly in the field of machine interpretation of line drawings, which remains his primary research interest. He is a member of the Institution of Engineering and Technology.

Pedro P. Company earned his Ph.D. in Mechanical Engineering in 1989 and has been Full Professor of Engineering Graphics at Universitat Jaume I since 1996. His main research fields are CAD and Sketch-Based Modelling, with more than 20 international papers and communications on those areas published to date. He is currently participating in the development of a new sketch-based modelling interface and its applicability for teaching. He is also involved in Emotional Design and Collaborative Product Engineering projects.