Finding Solvable Subsets of Constraint Graphs

Christoph M. Hoffmann[1] **Andrew Lomonosov**[2]

Meera Sitharam[2][3]

Abstract. We present a network flow based, degree of freedom analysis for graphs that arise in geometric constraint systems. For a vertex and edge weighted constraint graph with m edges and n vertices, we give an $O(n(m+n))$ time max-flow based algorithm to isolate a subgraph that can be solved separately. Such a subgraph is called *dense*. If the constraint problem is not overconstrained, the subgraph will be minimal.
For certain overconstrained problems, finding minimal dense subgraphs may require up to $O(n^2(m+n))$ steps. Finding a minimum dense subgraph is NP-hard. The algorithm has been implemented and consistently outperforms a simple but fast, greedy algorithm.

Keywords: Extremal subgraph, dense graph, network flow, combinatorial optimization, constraint solving, geometric constraint graph, geometric modeling.

1 Introduction

A geometric constraint problem consists of a finite set of geometric objects and a finite set of constraints between them. Geometric objects include *point*, *line*, *plane*, *circle*, and so on. Constraints between them might be *parallel*, *perpendicular*, *distance*, *tangency*, and so on. Some of these constraints are logical, such as incidence or tangency; others are dimensional such as distance or angle. Geometric constraint systems are the basis for design and manipulation in geometric modeling, and arise in virtual reality, robotics, and computer graphics as well. For applications in CAD, see, e.g., [12, 14, 23]. There is an extensive literature on geometric constraint solving. For recent reviews see, e.g, [7].

Geometric constraint problems can be solved algebraically. Briefly, the geometric objects are coordinatized, and the constraints between them are expressed in the form of polynomial equations. The resulting system of equations is usually nonlinear, and most constraint solvers use techniques for decomposing the

[1] Computer Science, Purdue University, West Lafayette, Indiana 47907, USA. Supported in part by NSF Grants CDA 92-23502 and CCR 95-05745, and by ONR Contract N00014-96-1-0635.
[2] Mathematics and Computer Science, Kent State University, Kent, Ohio 44242, USA
[3] Supported in part by NSF Grant CCR 94-09809.

system into small subsystems that can be solved separately where possible; e.g., [2, 8, 13, 17]. Such a decomposition increases solver efficiency and robustness substantially. Direct attempts at processing the entire system include using Gröbner bases [24] or the Wu-Ritt method [3]. They are general but typically do not scale.

Graph Based Constraint Analysis A large class of solvers translates the constraint problem into a graph. Graph vertices represent geometric objects, edges represent constraints. For ternary and higher constraints, the graph is a hypergraph. Analysis and decomposition of the problem is based on isolating subgraphs that correspond to subsystems of the equations that can be solved separately. The graph is *weighted*; both edges and vertices have a positive integer weight. The weight of a vertex is the degree of freedom of the represented geometric entity.[4] The weight of an edge equals the degrees of freedom determined by the represented constraint.

In the case of planar constraint problems, a class widely studied, several solvers decompose the constraint graph recursively into triangles and solve each triangle separately [2, 20, 21]. This decomposition is a special case of a *degree of freedom analysis*. For example, in the solver [2, 7], the geometric objects are points, lines, and rigid clusters of them, with the respective degrees of freedom of 2, 2, and 3. Two specific situations are considered in these graph analyses:

1. An edge of weight 1 is incident to vertices of weight 2. The associated constraint subproblem can be solved: for example, if the vertices are points and the edge a distance constraint, then the solution is a pair of points at fixed distance moving as a rigid body with 3 degrees of freedom.
2. A triangle with vertices of weight 3, and edges weight 2 represents incidences between geometric objects in 3 clusters. By placing the three incident pairs according to the constraints derived from their relative position within each rigid cluster, the three clusters can be placed rigidly with respect to each other and are merged into a bigger cluster.

Many interesting results are known about the properties of this type of constraint analysis, e.g., [9, 10].

Problem Statement and Results A *weighted undirected graph* is a graph where every vertex and every edge has a positive integer weight. We consider the following problem:

$G = (V, E, w)$ is a weighted undirected graph with n vertices V and m edges E; $w(v)$ is the weight of the vertex v and $w(e)$ is the weight of the edge e. Find a vertex-induced subgraph $A \subseteq G$ such that

$$\sum_{e \in A} w(e) - \sum_{v \in A} w(v) > K \tag{1}$$

[4] Roughly speaking, the number of independent variables coordinatizing a geometric object is the number of degrees of freedom.

Such a subgraph A is called *dense*. Typically, edge and vertex weights have a constant bound. In constraint solving, we seek a *minimal* dense subgraph, that is, a dense subgraph that does not contain a proper dense subgraph.

We design, analyze and implement an algorithm to find dense subgraphs in $O(n(m+n))$ steps. It is essentially a modified version of an incremental, maximum flow algorithm where edge capacities are saturated in certain natural order. This modification is crucial, however, in that it allows one to use existing flows in order to derive useful information about the densities of already examined subgraphs. If the graph arises from a geometric constraint problem that is not overconstrained, then the subgraphs found will be minimal and m will be $O(n)$. For overconstrained problems, additional processing is needed and may require up to $O(n^2(m+n))$ steps. In our experiments, the algorithm has consistently outperformed a simple but fast, greedy algorithm. The related problem of finding the *minimum* dense subgraph A is NP-hard.

Dense Subgraphs and Degrees of Freedom We seek a subgraph of the constraint graph in which the weights of the vertices minus the weights of the edges equals a fixed constant D, exactly the situation expressed by Inequality (1) with $K = -(D+1)$. In the example of planar constraint solvers [2, 7], $D = 3$ because a rigid planar figure has three degrees of freedom (absent symmetries). In 3-space, $D = 6$ in general. If the subproblem is to be fixed with respect to a global coordinate system, then $D = 0$ is required.

When a subgraph of the appropriate density has been found, the corresponding geometric objects can be placed rigidly with respect to each other using only the constraints between them. It is advantageous to find small dense subgraphs so that the associated equation system is as small as possible.

Having processed a dense subgraph, the solver then contracts the the subgraph to a single vertex v_c of weight D, suitably inducing edges between v_c and the other vertices. The full description of this process is beyond the scope of the conference paper, but we note that our algorithm extends to address this iterated dense subgraph problem.

If G is a constraint graph and A a dense subgraph, then density $d(A) = D$ means that the corresponding subproblem is *generically* well-constrained: in general, the geometric problem has a discrete set of solutions. For instance, for six points in space and twelve distance constraints between them, in the topology of the edges of an octahedron, the configuration is rigid, [15, 25]. However, for special distance values we obtain Bricard octahedra and then there would be nonrigid solutions; i.e., the problem would be actually underconstrained [26].

Prior Work on Constraint Graph Analysis Prior attempts at a degree of freedom analysis for constraint graphs often concentrated on recognizing specific dense subgraphs of known shape, such as the triangles of [2, 20, 21] or the patterns of [2, 15, 16]. This approach has limited scope. The scope can always be extended by increasing the repertoire of patterns of dense subgraphs. However, doing so results in greater combinatorial complexity and eventually makes efficient implementation too difficult.

More general attempts reduce the recognition of dense subgraphs in a degree-of-freedom analysis to a maximum weighted matching problem in bipartite graphs using methods from, e.g., [18]. A variation [1] of this approach does not use a degree-of-freedom analysis and directly deals with the algebraic constraints. In this case, a maximum cardinality bipartite matching is used, since no weights are required. The approach can be generalized to a weighted version required for a degree-of-freedom analysis by replicating vertices. We discuss briefly in Section 2 why both approaches are less efficient than the approach presented here. In particular, having found the required matching, finding a dense subgraph requires significant additional work, and it becomes difficult to isolate minimal dense subgraphs. The general approach of [17] appears to be exponential.

A different approach to constraint graph analysis uses rigidity theorems; e.g., [4, 11]. Corresponding decomposition steps may be nondeterministic or require difficult symbolic computations when computing a solution.

2 Finding a Dense Subgraph

We devise a flow-based algorithm for finding dense subgraphs assuming that $K = 0$ in Equation (1). We discuss the case $K \neq 0$ in Section 4.

Definition For $A \subseteq G$ define the **density** function d

$$d(A) = \sum_{e \in A} w(e) - \sum_{v \in A} w(v)$$

Suppose that we want to find a *most* dense subgraph $A \subseteq G$, i.e, one for which $d(A)$ is maximum. We could maximize, over subgraphs A of G, the expression

$$d(A) + \sum_{v \in G} w(v) = \sum_{e \in A} w(e) + \sum_{v \notin A} w(v) \tag{2}$$

or, in other words, minimize

$$\min_{A \subseteq G} \Big(\sum_{e \notin A} w(e) + \sum_{v \in A} w(v)\Big) \tag{3}$$

To do this, consider a bipartite graph $\tilde{G} = (M, N, \tilde{E}, w)$ associated with the given graph $G = (V, E, w)$. The vertices in N are the vertices in V and the vertices in M are the edges in E. Moreover, the edges of $\tilde{G}$ are $\tilde{E} = \{(e,u), (e,v) \mid e = (u,v), e \in E\}$. The weights w now appear on the vertices of $\tilde{G}$. Maximizing the expression (2) reduces to finding a maximum weighted independent set in the bipartite graph $\tilde{G}$, or, equivalently, the minimum weight vertex cover.

There are two ways to try to find the minimum weight vertex cover. The minimum *cardinality* vertex cover in a bipartite graph can be identified with a maximum cardinality matching and can be found using network flow in $O(\sqrt{n}m)$

time [5]. To take advantage of this algorithm, however, we need to replicate edges and vertices corresponding to the weights, find a minimum cardinality vertex cover in this larger graph, and then try to locate a corresponding minimum weight vertex cover in $\tilde{G}$ and the corresponding dense subgraph in the graph G.

The unweighted version of bipartite matching can be used naturally when variables are directly represented as vertices and the algebraic equations are represented as edges (instead of analyzing degrees-of-freedom). This results in a constant factor increase in the size of the graph. Using this approach, the problem of finding a dense subgraph – when $K = -1$ – was solved in [1], however, it is not clear how to extend the algorithm for general K.

A second way is to search for a minimum weighted vertex cover by solving the maximum (vertex) weighted bipartite matching problem. A maximum (edge) weighted bipartite matching problem can be solved in $O(\sqrt{n}m \log n)$ time for bounded weights, [22]. This trivially gives a solution to the maximum (vertex) weighted bipartite matching problem. The catch is that, unlike in the unweighted case, a minimum weighted vertex cover does *not* correspond directly to a maximum weighted matching. Having found a maximum weighted matching, a significant amount of work is needed to obtain the minimum weight vertex cover, and, from it, the corresponding dense subgraph in G.

In general, the maximum matching approach has the following disadvantages. (1) The maximum (weighted) matching in ($\tilde{G}$) does not directly correspond to the dense subgraph in G. (2) We need only *some* subgraph of a specific density, not necessarily a *most* dense one. (3) Maximum matching provides no natural way of finding a minimal dense subgraph. We develop a more efficient method analogously based on a different optimization problem (see [19]), but which will be seen to address both drawbacks.

Construction of the Network From the graph G, construct a bipartite *directed* network $G^* = (M, N, s, t, E^*, w)$, where M, N and E^* are as in $\tilde{G}$. The source s is connected by a directed edge to every node in M, and every node in N is connected by a directed edge to the sink t. The capacity of the network edge (s, e), $e \in M$, is the weight $w(e)$ of the edge e in G. The capacity of the network edge (v, t), $v \in N$, is equal to the weight $w(v)$ of the vertex v in G. The capacity of the network edge (e, v), $e \in M,\ v \in N$, is infinite. There are no other network edges. See also Figure 1.

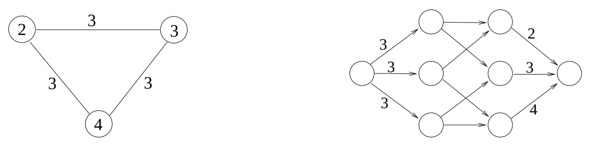

Fig. 1. Constraint graph (left) and associated network (right).

A minimum cut in G^* *directly* defines a subgraph A that minimizes Expression (3). It can be found as the max flow using a netflow algorithm. Now we are only interested in finding a dense subgraph and not necessarily the *most* dense one. So, we are interested in a small enough cut in G^*, not necessarily the smallest one. Thus, to find a dense subgraph, there should be an algorithm that is faster than a general maximum flow (or minimum cut) algorithm.

The algorithm is a modification of the incremental max flow algorithm. The idea of the algorithm is to start with the empty subgraph G' of G and add to it one vertex at time. When a vertex v is added, consider the adjacent edges e incident to G'. For each e, try to distribute the weight $w(e)$ to one or both of its endpoints without exceeding their weights; see also Figure 2. As illustrated by Figure 3, we may need to redistribute some of the flow later.

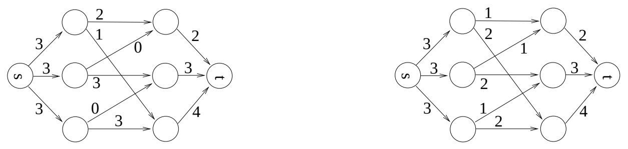

Fig. 2. Two different flows for the constraint graph of Figure 1

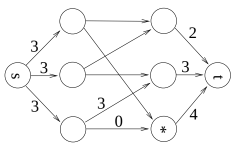

Fig. 3. Initial flow assignment that requires redistribution later

If we are able to distribute all edges, then G' is not dense. If no dense subgraph exists, then the algorithm will terminate in $O(n(m+n))$ steps and announce this fact. If there is a dense subgraph, then there is an edge whose weight cannot be distributed even with redistribution. The last vertex added when this happens can be shown to be in all dense subgraphs $A \subseteq G'$.

Distributing an edge e in G now corresponds to pushing a flow equal to the capacity of (s, e) from s to t in G^*. This is possible either directly by a path of the form $\langle s, e, v, t\rangle$ in G^*, or it might require flow redistribution achieved by a standard search for augmenting paths [6], using network flow techniques. Note that the search for augmenting paths takes advantage of the fact that the flow through each vertex in M is distributed to exactly 2 vertices in N (lines 4-7).

If there is an augmenting path, then the resulting flows in G^* provide a distribution of the weight of each edge e in the current subgraph G' consisting of the examined vertices and edges of the original graph G as follows: the weight

$w(e)$ of each edge e connecting the vertices a and b is split into two parts f_e^a and f_e^b such that $f_e^a + f_e^b = w(e)$ and, for each vertex $v \in G'$, $\sum_{e=(v,*)} f_e^v \leq w(v)$.

If there is no augmenting path for the residual flow on (s, e), i.e, the flow $w(e)$ is undistributable, then a dense subgraph has been found and is identified based on the flows in G^* starting from e.

Algorithm Dense

```
1.  $G' = \emptyset$.
2.  **for every** vertex $v$ **do**
3.      **for every** edge $e$ incident to $v$ and to $G'$ **do**
4.          Distribute the weight $w(e)$ of $e$
5.          **if** not able to distribute all of $w(e)$ **then**
6.              $A$ = set of vertices labeled during Distribute
7.              **goto** Step 12
8.              **endif**
9.          **endfor**
10. add vertex $v$ to $G'$
11. **endfor**
12. **if** $A = \emptyset$ **then** no dense subgraph exists
13. **else** $A$ is a dense subgraph
```

Algorithm Distribute searches for augmenting paths in G^* to achieve the required flow and the labeling. It repeats a Breadth First Search for augmentation until all of $w(e)$ has been distributed or until there is no augmenting path. The technique is somewhat similar to the one used in max-flow algorithm in [19].

Algorithm Distribute
Input: $(G^*, f, edge)$, where $G^* = (N, M, s, t, E^*, w)$, f is a set of flows f_e^v
and *edge* is the edge that is being distributed.

```
0.  Initialize $scan(v) = 0, label(v) = 0, scan(e) = 0, label(e) = 0$ for all $v \in N, e \in M$
1.  $vert = 0,\ capvert = 0$
2.  $label(edge) = 1,\ pathcap(edge) = w(edge)$
3.  **while** $(w(edge) > \sum_v f_{edge}^v)$ **or** not all labeled nodes have been scanned
4.      **for** all labeled $e \in M$, with $scan(e) = 0$
5.          label unlabeled neighbors of $e$ (i.e $v \in N$)
6.          $scan(e) = 1,\ pred(v) = e,\ pathcap(v) = pathcap(e)$
7.      **endfor**
8.      **for all** labeled $v \in N$ with $scan(v) = 0$
9.          **if** $\min(w(v) - \sum_e f_e^v,\ pathcap(v)) > capvert$ **then**
10.             $vert = v,\ capvert = min(w - \sum_e f_e^v,\ pathcap(v))$
11.             **else**
12.                 label all unlabeled $e' \in M$ s.t $f_{e'}^v > 0$
13.         **endif**
14.             $scan(v) = 1$
15.     **endfor**
```

```
16.         if $vert > 0$ then
17.             An augmenting path from $s$ to $t$ has been found: backtrack from
            $vert$ using $pred()$ and change the values of $f_e^v$ as requirted.
18.             for all $e \in M$, $v \in N$
19..                $label(e) = 0, scan(e) = 0, label(v) = 0, scan(v) = 0$
20.             endfor
21.             $vert = 0, capvert = 0, label(edge) = 1$
22.             $pathcap(edge) = w(edge) - \sum_v f_{edge}^v$
23.         endif
24.     endwhile
```

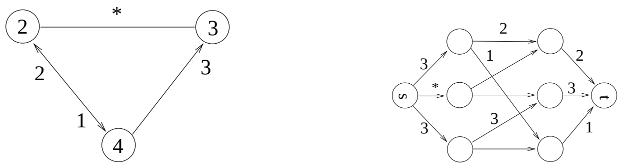

Fig. 4. Current graph G' and corresponding network G^*, the edge marked by asterisk is currently being distributed

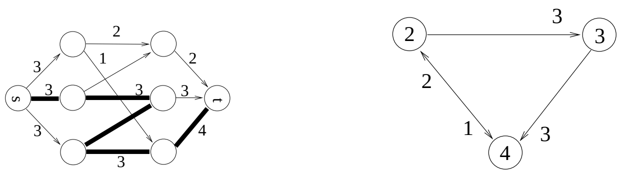

Fig. 5. The augmenting path and the distribution of edges in original graph G'

Lemma 1
Let G^* be the bipartite network constructed from G, and $e \in M$. If, after checking all possible augmenting paths originating at e, the flow through (s, e) is less than the capacity of (s, e), and $A = (E_A, V_A)$ is the set of edges and vertices labeled after the search for an augmenting path, then $d(A) > 0$.

Proof: A is a subgraph of G because for every labeled edge $e \in E$ both of its vertices will be labeled. For all $v \in V_A$, the network edges (v, t) are saturated, otherwise there is an augmenting path from e to v and the flow through (s, e) can be increased. Let f be the maximum flow through (E_A, V_A). Since all (v, t) are saturated, $f = \sum_{v \in A} w(v)$, but since at least one edge (s, e) is not distributed $f < \sum_{e \in A} w(e)$; therefore $d(A) = \sum_{e \in A} w(e) - \sum_{v \in A} w(v) > 0$. □

Complexity Analysis In the worst case, constructing an augmenting path labels at most $m+n$ nodes are labeled. Since the algorithm stops when the total edge weight exceeds the total vertex weight by K, the total edge weight that is distributed is at most K plus the total vertex weight times a constant bound b. Each augmentation increases the flow by least 1 unit. Therefore, the number of augmentations cannot exceed $O(n)$. Hence, Algorithm Dense has complexity $O(n(m+n))$.

3 Finding a Minimal Dense Subgraph

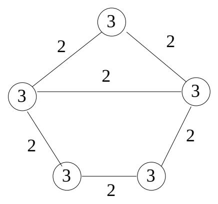

Fig. 6. This subgraph is dense for $K=-4$, so is upper triangle

Let $G'=(V',E')$ be the subgraph already examined by Algorithm Dense. That is, assume that the vertices V' have been examined and the weight $w(e)$ of all induced edges e has been distributed. Let v be the first vertex that is about to be examined next, such that the weight of one of its incident edges e adjacent to G' cannot be distributed. Let $V_A \subseteq V'$ be the set of vertices labeled while trying to distribute $w(e)$, (which includes the vertex v), and let A be the subgraph induced by V_A. By Lemma 1, A is a dense subgraph.

Lemma 2
Every dense subgraph of A contains v.

Proof: Let A' be a dense subgraph of A not containing v. Then there should be an edge $e \in A'$ such that e was not distributed before v was considered. However, this contradicts our assumption that all edges in G' have been distributed. $\square$

Remark
Similarly, if $(v,v_1),(v,v_2),...,(v,v_k)$ are undistributed edges of v then every dense subgraph of A contains at least one edge from this list. If $k=1$ then every dense subgraph of A contains (v,v_1).

Proposition 3
If the amount of undistributable flow, i.e, the density of A is $d(A)$ and A' is a dense proper subgraph of A, then $0<d(A')<d(A)$ (in general, $K<d(A')<d(A)$).

Proof: Note that the excess flow comes from the edges incident to v. Suppose $A' \subseteq A$ is dense and $d(A') \geq d(A)$. By Lemma 2, A' contains v. Consider the relative complement A^* of A' with respect to A. Then $d(A^*) \leq 0$, which implies that the vertices of A^* could not have been labeled after distributing the flow of the edges of v. Since all vertices in V_A are labeled, we know that $A = A'$. □

Corollary 4
If $d(A) = 1$ then A is minimal. In general, if $d(A) = K + 1$, then A is minimal.

In particular, when $K = 0$, well-constrained or underconstrained problems have $d(A) \leq 1$. Then, by Corollary 4, we know that the subgraph found by Algorithm Dense is minimal. Moreover, if overconstrained problems are rejected, then a first test for overconstrained would be to determine $\sum_{e \in G} w(e) - \sum_{v \in G} w(v) > 1$ in linear time. This test would reject many overconstrained problems. The remaining cases would be found by noting whether $d(A) > 1$ when Algorithm Dense terminates.

We may accept consistently overconstrained problems. In that case, the graph A may have to be analyzed further to extract a minimal dense subgraph. We now develop a method for performing this extraction, once a dense subgraph A has been found by Algorithm Dense and $d(A) > 1$. The algorithm to be developed post-processes only the subgraph A.

Without loss of generality, assume that A contains the vertices $\{v_1, \ldots, v_l, v_{l+1}\}$, and v_{l+1} was the last vertex examined when A was found. The density $d(A)$ is the total undistributed weight of the edges between v_{l+1} and $\{v_1, \ldots, v_l\}$. We begin with the knowledge of a subgraph B of A that is contained in *every* dense subgraph of A. By Lemma 2, B contains initially the vertex v_{l+1}. The algorithm to be developed is to determine either an enlargement of the graph B, or else a reduction of the graph A.

We perform the following step iteratively. Choose a vertex $v_k \notin B$ from A. Determine the quantity $c = d(A) - w(e') + f_{e'}^{v_k} + f_{e'}^{v_{l+1}}$ where e' is the edge (v_k, v_{l+1}). That is, c is the undistributed weight of edges in A without v_k. Remove the vertex v_k from A along with its edges. This would create unutilized capacity in the set of vertices adjacent to v_k (that are in A) through the set E_k of incident edges. The excess vertex capacity is

$$\sum_{e \in E_k} w(e) - w(v_k) - w(e') + f_{e'}^{v_k} + f_{e'}^{v_{l+1}}$$

where e' is the edge between v_k and v_{l+1}. This quantity is the total flow on the edges of v_k, distributed away from v_k. We now attempt to distribute the previously undistributed weight of the edges between v_{l+1} and $\{v_1, \ldots, v_l\} - \{v_k\}$, using redistribution if necessary. We use Algorithm Distribute on the modified network, setting the capacity of (v_k, t) to zero. There are two outcomes possible:

1. If we distribute all of c successfully into the newly created holes, or excess capacity on the vertices adjacent to v_k, then no subgraph of $A - v_k$ is dense, so v_k belongs in every dense subgraph of A, and hence gets restored into A and, moreover, gets added to B.

2. If we were unable to distribute c, then by Lemma 1, we have found a smaller dense subgraph of $A - v_k$. This new subgraph consists only of the vertices labeled by the Algorithm Distribute in the process of distributing one of the undistributed edges adjacent to v_{l+1}. This outcome reduces the size of A. Note that, by Proposition 3, the density (and size) of the new graph A must drop by at least 1.

We repeat this process for the remaining vertices in $A - B$. We stop either when $d(B) > 0$, because then B is minimal dense, or when $d(A) = 1$, because then A is minimal dense.

Complexity Analysis The complexity of each iteration described above is $O(n(m+n))$, since c represents the undistributed weight on at most n edges adjacent to v_{l+1} are distributed at each iteration. We can assume that the sum of capacities of the edges is constant, thus the determination of a minimal dense subgraph takes $O(n^2(m+n))$ steps. Note, however, that by Proposition 3 the actual complexity rarely reaches this upper bound.

The complexity of the iteration is reduced to $O(m+n)$ if the constraint graph has bounded valence or if $d(A)$ has an a-priori constant bound. The latter situation means that the constraint problem has a bound on the "overconstrainedness" of subgraphs, a natural assumption if the constraint problem is specified interactively and we keep track of the density of the full constraint graph. In those cases, the complexity reduces to $O(n(m+n))$ steps.

Algorithm Minimal

Comment: The input is the output of Dense, a dense (sub)graph A of G, and the distribution of edge weights f_e^a and f_e^b for each edge $e = (a, b)$. Note that v_{l+1} is the last vertex added that caused A to be found, and e' is the edge between v_k and v_{l+1}.

```
1.  $B = \{v_{l+1}\}$
2.  while $d(B) \leq 0$ and $d(A) > 1$ do
3.      choose $v_k \in A - B$
4.      $c = d(A) - w(e') + f_{e'}^{v_k} + f_{e'}^{v_{l+1}}$
5.          for all $v \in N$ (Removing $\{v_k\}$ from $A$)
6.              Let $e = (v, v_k)$
7.              remove $e$ from $M$
8.          endfor
9.          remove $v_k$ from $N$
10.     Distribute (in $A$) excess $c$ from the edges of $v$
11.     if there are some undistributed edges left then
12.         set $A =$ new labeled graph
13.     else
14.         set $A = A \cup \{v_k\}$ (as well as restoring edges of $v_k$)
15.         set $B = B \cup \{v_k\}$
16.     endif
```

```
17. endwhile
18. output $B$, if $d(B) > 0$, else output $A$.
```

4 The Case of $K \neq 0$

As presented, our algorithms satisfy Inequality (1) for $K = 0$: keep adding vertices until we are unable to distribute the edge weight/capacity. The first undistributable edge signals a dense graph A, for $K = 0$, and $d(A) > d(A - v)$, where v was the last vertex examined. We now explain how to modify the algorithm to accommodate different values of K.

The modification for $K > 0$ is trivial. Instead of exiting Algorithm Dense when an edge cannot be distributed, exit when the total undistributable edge capacity exceeds K. The computation of the total undistributable edge capacity so far is based only on the weights of the labeled edges and vertices, thus ensuring that the resulting dense graph is connected. An analogous change is in order for Algorithm Minimal. Clearly this modification does not affect the performance complexity of the algorithms.

When $K < 0$ the algorithms can also be modified without increasing the complexity. Suppose, therefore, that $K < 0$, and consider Step 4 of Algorithm Dense. If $w(v) + K \geq 0$, simply reduce the capacity of the network edge (v, t) to $w(v) + K$ and distribute $w(e)$ in the modified network. If the edge cannot be distributed, then the subgraph found in Step 6 has density exceeding K. If every incident edge can be distributed, then restore the capacity of the network edge when adding v to G'.

If the weight $w(v)$ of the added vertex v is too small, that is, if $w(v)+K < 0$, then a more complex modification is needed. We set the capacity of (v, t) to zero. Let e be an new edge to be distributed, and do the following.

```
1.  Distribute the edge weight $w(e)$ in the modified network.
2.  if $w(e)$ cannot be distributed then
3.      we have found a dense subgraph for $K$; exit.
4.  else
5.      save the existing flow for Step 10.
6.      increase the flow of $e$ by $-(w(v) + K)$
7.      if the increased flow cannot be Distributed then
8.          we have found a dense subgraph for $K$; exit.
9.      else
10.         restore the old flow. No dense subgraph found
11.     endif
12. endif
```

In worst case the algorithm saves and restores the flows for every edge added, which requires $O(m)$ operations per edge. Distributing the edge flow however dominates this cost since it may require up to $O(m + n)$ operations per edge; so the modification does not adversely impact asymptotic performance.

5 Implementation

The network flow algorithms were implemented in C. Both Dense and Minimal were run in sequence. We tested the algorithms on low-density graphs G, where $|G| = 40$, $K = -1$, and a typical minimal dense subgraph A' in G would have density 0 or 1. The number of vertices in the initial dense graph A was between 14 and 22. The dense subgraph A is usually also a minimal dense subgraph (the situation $B = A$ in Algorithm Minimal), this being the worst case for the combined algorithm *if* the density of A is > 0. We tested some of the usual (generically well or under constrained) cases where $d(A) = 0 = K + 1$, where the Algorithm Minimal is unnecessary by Corollary 4. As a complexity measure, we counted the number of times a vertex or an edge is being labeled.

Heuristically, the selection of the next vertex to be examined and saturated and edges to be distributed can be done in a greedy fashion. That is, at each step choose a vertex with the "heaviest" set of edges connecting it to the set of vertices in G' and start distributing its edges in descending weight order. The distribution and redistribution of the edges is carried out by using flows on the bipartite network G^* described earlier.

In the table, m is the number of edges of the original graph, ($n = 40$), A is the dense subgraph found, A' is a minimal dense subgraph of A, n_1 is the number of operations required to find A, n_2 is the total number of operations required to find A', p is the number of augmented pathes required to find A'.

m	$\|A\|$	$\|A'\|$	$d(A')$	n_1	n_2	p
141	21	20	1	518	1644	162
100	20	20	0	728	728	66
139	18	18	0	680	680	76
191	22	22	0	903	903	91
71	$*$	$*$	$*$	187	187	61
100	23	22	0	624	1232	134
139	20	19	0	623	872	190
140	16	14	0	127	376	899

Table 1. Performance of the Netflow Algorithms. The case marked $*$ has no dense subgraphs.

Note that since the density of A was small, the minimal part increased a running time by a small constant factor.

We also implemented a simple, but fast greedy algorithm for finding dense subgraphs. In our experiments, the greedy algorithm was consistently and significantly inferior to the network flow algorithms.

6 Conclusions

The algorithms we have developed are general and efficient. Previous degree-of-freedom analyses usually analyze simple loops in the constraint graph, or else are unable to isolate a *minimal* subgraph that is dense. Moreover, by making K a parameter of the algorithm, the method presented here can be applied uniformly to planar or spatial geometry constraint graphs. Extensions to ternary and higher-order constraints can also be made.

When constraint problems are not overconstrained, a typical situation in applications, then the algorithms perform better. This is not uncommon in other constraint graph analyses [10], and is related to the fact that a well-constrained geometric constraint graph has only $O(n)$ edges.

We have implemented an extension of the algorithm that iterates finding dense subgraphs and solves a geometric constraint problem recursively by condensing dense subgraphs to single vertices in the manner first described in [2, 7]. Here, a critical aspect is to account for previously constructed flows and running Algorithm Dense incrementally.

A specific advantage of our flow-based analysis is that we can run the algorithm on-line. That is, the constraint graph and its edges can be input continuously to the algorithm, and for each new vertex or edge the flow can be updated accordingly. Thus, it is a good fit with geometric constraint solving applications.

References

1. S. Ait-Aoudia, R. Jegou, and D. Michelucci. Reduction of constraint systems. In *Compugraphics*, pages 83–92, 1993.
2. W. Bouma, I. Fudos, C. Hoffmann, J. Cai, and R. Paige. A geometric constraint solver. *Computer Aided Design*, 27:487–501, 1995.
3. S. C. Chou, X. S. Gao, and J. Z. Zhang. A method of solving geometric constraints. Technical report, Wichita State University, Dept. of Computer Sci., 1996.
4. G. Crippen and T. Havel. *Distance Geometry and Molecular Conformation*. John Wiley & Sons, 1988.
5. S. Even and R. Tarjan. Network flow and testing graph connectivity. *SIAM journal on computing*, 3:507–518, 1975.
6. L.R. Ford and D.R. Fulkerson. *Flows in Networks*. Princeton Univ. Press, 1962.
7. I. Fudos. *Geometric Constraint Solving*. PhD thesis, Purdue University, Dept of Computer Science, 1995.
8. I. Fudos and C. M. Hoffmann. Constraint-based parametric conics for CAD. *Computer Aided Design*, 28:91–100, 1996.
9. I. Fudos and C. M. Hoffmann. Correctness proof of a geometric constraint solver. *Intl. J. of Computational Geometry and Applications*, 6:405–420, 1996.
10. I. Fudos and C. M. Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Trans on Graphics*, page in press, 1997.
11. T. Havel. Some examples of the use of distances as coordinates for Euclidean geometry. *J. of Symbolic Computation*, 11:579–594, 1991.
12. C. M. Hoffmann. Solid modeling. In J. E. Goodman and J. O'Rourke, editors, *CRC Handbook on Discrete and Computational Geometry*. CRC Press, Boca Raton, FL, 1997.

13. C. M. Hoffmann and J. Peters. Geometric constraints for CAGD. In M. Daehlen, T. Lyche, and L. Schumaker, editors, *Mathematical Methods fot Curves and Surfaces*, pages 237–254. Vanderbilt University Press, 1995.
14. C. M. Hoffmann and J. Rossignac. A road map to solid modeling. *IEEE Trans. Visualization and Comp. Graphics*, 2:3–10, 1996.
15. Christoph M. Hoffmann and Pamela J. Vermeer. Geometric constraint solving in R^2 and R^3. In D. Z. Du and F. Hwang, editors, *Computing in Euclidean Geometry.* World Scientific Publishing, 1994. second edition.
16. Christoph M. Hoffmann and Pamela J. Vermeer. A spatial constraint problem. In *Workshop on Computational Kinematics*, France, 1995. INRIA Sophia-Antipolis.
17. Ching-Yao Hsu. *Graph-based approach for solving geometric constraint problems.* PhD thesis, University of Utah, Dept. of Comp. Sci., 1996.
18. R. Latham and A. Middleditch. Connectivity analysis: a tool for processing geometric constraints. *Computer Aided Design*, 28:917–928, 1996.
19. E. Lawler. *Combinatorial optimization, networks and Matroids.* Holt, Rinehart and Winston, 1976.
20. J. Owen. Algebraic solution for geometry from dimensional constraints. In *ACM Symp. Found. of Solid Modeling*, pages 397–407, Austin, Tex, 1991.
21. J. Owen. Constraints on simple geometry in two and three dimensions. In *Third SIAM Conference on Geometric Design.* SIAM, November 1993. To appear in Int J of Computational Geometry and Applications.
22. T.L. Magnanti, R.K. Ahuja and J.B. Orlin. *Network Flows.* Prentice-Hall, 1993.
23. Dieter Roller. Dimension-Driven geometry in CAD : a Survey. In *Theory and Practice of Geometric Modeling*, pages 509–523. Springer Verlag, 1989.
24. O. E. Ruiz and P. M. Ferreira. Algebraic geometry and group theory in geometric constraint satisfaction for computer-aided design and assembly planning. *IIE Transactions on Design and Manufacturing*, 28:281–294, 1996.
25. P. Vermeer. Assembling objects through parts correlation. In *Proc. 13th Symp on Comp Geometry*, Nice, France, 1997.
26. W. Wunderlich. Starre, kippende, wackelige und bewegliche Achtflache. *Elemente der Mathematik*, 20:25–48, 1965.

This article was processed using the LaTeX macro package with LLNCS style