Architectural Symbol Recognition Using a Network of Constraints

Christian Ah-Soon [1] and Karl Tombre [2]

LORIA–CNRS–INPL, B.P. 239, 54506 Vandœuvre-lès-Nancy CEDEX, France

Abstract

We propose a method for recognizing architectural symbols. The method is based on the description of the model through a set of constraints on geometrical features, and on propagating the features extracted from a drawing through the network of constraints. One advantage of this approach is the possibility to incrementally build and update the model, when new symbols have to be taken into account.

Key words: Graphics recognition. Symbol recognition. Architectural drawings. Constraint checking.

1 Introduction

Our team has been working for several years on the analysis of architectural drawings. The ultimate aim is to reconstruct a 3D model of a building from the analysis of design-phase drawings. For this purpose, we have developed two complementary methods for reconstructing the geometric model of a level from its vectorization (Ah-Soon and Tombre 1997; Dosch et al. 1999). But both methods rely heavily on a correct recognition of architectural symbols. These symbols are much less normalized than those which can be found in other technical domains. We therefore need a flexible method, capable of easily integrating new symbol models with minimal computation overhead in the recognition phase.

After explaining our two main sources of inspiration for this work (§ 2), we describe our model (§ 3) and the way we use it for recognition (§ 4), before explaining how new symbols can be added to the model (§ 5). In § 6, we discuss some limitations of our present work and propose some perspectives on extending it.

[1] Now with Business Objects, Paris.
[2] Corresponding author.

2 State of the Art

The recognition of graphical symbols is a well-known problem, for which many methods have been proposed (Cordella and Vento 1999; Chhabra 1998). A first family of methods has been adapted to documents such as diagrams, basically made of symbols and connecting lines—see (Yu et al. 1997) as a typical example. Other applications include map analysis (Samet and Soffer 1998) and printed music scores (Miyao and Nakano 1996). But there have been few attempts at recognizing architectural symbols; of the few works we are aware of, we can cite Lladós et al. (1997), who use attributed graph matching and a special graph-edit algorithm (Lladós and Martí 1999) to recognize symbols taken from a set of known models, and Valveny and Martí (1999), who have proposed a method based on deformable template matching within a Bayesian framework. These methods have proven to be efficient, even for hand-drawn drawings, although the scalability when the number of models increases is not guaranteed.

In our quest for a good recognition method, we felt the need for *flexibility* and *genericity*. As architectural drafting is much less normalized than other technical domains, we come upon large variations in the way basic elements such as doors or windows are represented. We therefore cannot build an *a priori* set of models and decide that these are the only symbols we will recognize. We must be able to incrementally add new models to the knowledge base, with minimal computational overhead during recognition.

A first system which inspired us was that of Pasternak (1996). In his ADIK system, he uses graphical specifications of the symbols, based on a number of predicates and on constraints between parts of the same geometric composition. Object recognition is activated through a triggering mechanism. The whole knowledge base is represented as a structural/geometric taxonomy.

This idea of constraint-based, hierarchical modelling appealed to us. However, we were looking for more efficient ways to manage a set of models, as a hierarchy of models can become very cumbersome to update and maintain. Messmer and Bunke (1996) proposed a method which allows for model pre-compilation through the use of a network, where all model descriptions are gathered at once; the features are the input to this network and "trickle down" until one of the terminal nodes—i.e. one of the model symbols—is activated. As illustrated by Fig. 1, their work was based on graph isomorphism (Messmer and Bunke 1998); in our case, as we use constraint propagation, we have adapted the network concept to these constraints.

3 Symbol Modelling

Before explaining our model, we will assume in this article that the vectorization of the document image yields a set of segments (Tombre et al. 1999). This can easily be extended to segments and arcs. Let $\mathcal{F}$ be the set of *features*, i.e. n-uples of distinct segments; let $\mathcal{P}$ be the set of *predicates*, i.e. boolean functions. The size of a feature is defined as the number of its segments, and the size of a predicate is the number of its arguments. Let $\mathcal{C}$ be the set of *constraints*, defined as $\{c \in \langle \mathcal{P}\ pr, \mathcal{F}\ fe\rangle | (size(fe(c)) \geq 1) \wedge (size(fe(c)) = size(pr(c)))\}$. Thus, a constraint is made of a predicate and of a feature, and it applies to the segments of this feature. We define the size of the constraint as being the size of its feature.

Let $\mathcal{D}$ be a set of descriptions. A description is defined by a feature, whose segments represent the model symbol, and by a set of constraints. These constraints apply to the segments of the feature, and are of two kinds: connection constraints, which describe connection relations between segments, and simple constraints. We define the size of a description as being the size of its feature.

For instance, we can define the description of a lozenge (Fig. 4) as follows:

$$d_{sl} = \langle\{sd_1, sd_2, sd_3, sd_4\}, \{cc_1, cc_2, cc_3, cc_4, cs_1, cs_2, cs_3, cs_4\}\rangle$$

$$cc_1 = \langle(sd_1, sd_2), p_c\rangle \;\; cc_2 = \langle(sd_2, sd_3), p_c\rangle \;\; p_c : x \times y \mapsto point1(x) = point2(y)$$

$$cc_3 = \langle(sd_3, sd_4), p_c\rangle \;\; cc_4 = \langle(sd_4, sd_1), p_c\rangle$$

$$cs_1 = \langle(sd_1, sd_2), p_s\rangle \;\; cs_2 = \langle(sd_2, sd_3), p_s\rangle \;\; p_s : x \times y \mapsto length(x) = length(y)$$

$$cs_3 = \langle(sd_3, sd_4), p_s\rangle \;\; cs_4 = \langle(sd_4, sd_1), p_s\rangle$$

In order to facilitate the input of descriptions, and to ensure the flexibility of the model, all descriptions are gathered in a text file, made of a set of entries. Each entry corresponds to a description, and is mainly a set of constraints on the features which constitute the symbol. These entries are written in a small language that we have designed, and whose grammar is summarized in Fig. 2. An example desribing the simple lozenge with description d_{sl}, using this language, is given by Fig. 3.

4 Symbol Recognition

4.1 Use of the Network

Although Messmer and Bunke use a different matching mechanism, subgraph isomorphism, we adapted several of their ideas to our method. For instance, we use a network to model the descriptions and thus perform the search for all possible symbols at once, instead of trying separately to match a candidate with all possible models. For this, we search separately for all the features verifying each constraint, and we then merge these features to get the symbol.

Thus, in order to detect lozenges of description d_{sl} (§ 3) among the segments of an image, we can use an 8-node network (Fig. 4). By using the various predicates ($p_{NM2}, p_{NC3}, p_{NM4} \ldots$), we get the constraints which describe the lozenge. Thus, the features which end up in NF8 represent lozenges.

4.2 Nodes of the Network

The search for symbols works through propagation of the segments through a network. This network is made of four kinds of nodes : NNSegment, NNMerge, NNCondition and NNFinal. These nodes are connected through father–son links; each node can have at most two fathers, but can have several sons. Each node tests some constraints, and can thus be seen as a “filter”, which only transmits to its sons the features (sets of segments) which verify the tested constraints. These features, created only once by each node, can be used by all the sons of the node. At the end, the segments of the features which have “trickled” down to the terminal nodes of the network represent the corresponding symbols.

For each network, there is only one **NNSegment** node, which corresponds to the root of the network. This node initializes the recognition process, creates a one-segment feature for each segment, and sends it to all its sons (Alg. 1). A **NNCondition** node has only one father. It tests the constraint on the features sent to it by its father. If the constraint is satisfied, the NNCondition node propagates the feature to its sons (Alg. 2). A **NNMerge** node has two father nodes, and gathers the features sent by its fathers, if they verify a connection constraint. The resulting feature, if any, is sent to the sons of the NNMerge node (Alg. 3). Note that in order to allow the NNMerge nodes to gather all their fathers’ features, each node in the network has to keep a local storage of the features it is transmitting. The **NNFinal** nodes are the terminal nodes; they have one father and no sons. Each of these nodes corresponds to one of the symbols which have to be recognized. When a feature reaches such a node, it has gone through a number of NNMerge and NNCondition nodes and has verified their constraints. To get the actual symbol, it is therefore sufficient to get the set of features stored in the NNFinal node.

Algorithm 1 NNSeg.transmission(image I)

```
for all segments s of I do
    newFea ← create a feature from s
    myFeatures.add (newFea)
    for all my sons n do
        n.transmission (newFea, me)
    end for
end for
```

Algorithm 2 NNCond.transmission($\mathcal{F}$ f, $\mathcal{N}$ p)

```
if f verifies myPredicate then
    myFeatures.add (f)
    for all my sons n do
        n.transmission (f, me)
    end for
end if
```

Algorithm 3 NNMerge.transmission($\mathcal{F}$ f, $\mathcal{N}$ p)

```
for all features g disjoint of f and sent by my other father (not p) do
    if p = myFather1 then
        newFea ← merge (f, g)
    else
        newFea ← merge (g, f)
    end if
    if newFea verifies myPredicate then
        myFeatures.add (newFea)
        for all my sons n do
            n.transmission (newFea, me)
        end for
    end if
end for
```

4.3 Use of the Network

4.3.1 Features Extracted from the Image

Two kinds of features can be extracted by low-level techniques: segments and arcs. As vectorization and arc recognition methods are always noisy, the resulting set of graphical entities may contain extraneous segments, especially at the junctions of thick lines. Actually, we are looking for symbols such as windows and doors, which are always represented with thin lines; we therefore separate thick lines from thin lines, using simple mathematical morphology. The thin lines image is vectorized independently of the thick lines image (Fig. 7(b) (Tombre et al. 1998). This reduces the number of artefacts in the set of vectors.

Although we do not describe it in this paper, for the sake of simplicity, we also add a NNArc node, which propagates arcs in the same way as NNSegment propagates segments.

4.3.2 Error Computation and Propagation

Although we only vectorize the thin lines, we still have errors, due to noise and to the approximation of curves by polylines. We therefore use an error measure, which quantifies the deviation between the searched symbol and the candidate features. When a node receives a feature, it computes the resulting error, if the segments of the feature do not exactly verify the constraint. This error is accumulated from one node to the other, and when it exceeds a given threshold, the feature is not transmitted anymore. We are working on having more adaptive thresholds, instead of the fixed ones currently in use.

4.3.3 Inversion of the features

The orientation of the graphical entities yielded by vectorization and arc segmentation is dependent on these low-level steps. But the descriptions used in the network may have different orientations. Therefore, in order to be able to check the constraints in all cases, we send both the original features and their inversion into the network.

For instance, if we search for three segments which only verify constraints cc_1 and cc_2 (§ 3), we cannot only input the segments yielded by vectorization (Fig. 5(a)) to the network; we must also input their inversions (Fig. 5(b)), so that the network can recognize the two configurations which verify the constraints (Figs. 5(c) and 5(d)).

4.3.4 Splitting up the image

In the example given in Fig. 4, with only 4 segments actually input into the network, 16 features are created and stored in the network after propagation. This is due to the fact that when a NNMerge node receives a feature from one of its fathers, it stores it for later constraints verification, whenever another feature comes from the other father. Thus, the number of features present in the network can become very large, especially when the image itself is large, or when the number of segments yielded by vectorization is large. This in turn increases the computing time and memory requirements necessary for propagation and checking.

As the symbols to be recognized are relatively small, compared to the image size, it is not necessary to try to recognize a symbol made of basic geometric features which are too far from each other—whose distance is larger than the maximal symbol size, more precisely. We therefore divide the image into $meshSize \times meshSize$

meshes, $meshSize$ being given as the maximal symbol size. Symbol detection is then repeated for each subimage defined by a 3×3 square of meshes. Only the segments contained in the corresponding meshes are sent into the network (Fig. 6).

To avoid erasing and reconstructing too many times the features contained in overlapping meshes, when the search window is moved over the image, we start a line by adding to the network all features contained in the window (Fig. 6(a)). After propagating these features through the network, the window is moved to the next mesh (Fig. 6(b)), but only the features of the first column are erased, and the features of the new column are added. This is repeated until the end of the line is reached (Fig. 6(c)). To avoid excessive complexity in the updating process, we chose in this case to erase all features and rebuild them at the start of the next line (Fig. 6(d)).

Because of the overlapping of the meshes, some symbols may be recognized several times through this method. We therefore need to sort the positions of all detected symbols, to eliminate double detections.

The use of this "windowing" system does not change the principle, nor the inner working of the network. It is only an extra layer, to avoid excessive computation requirements.

4.3.5 Results

We have tested our network on eleven architectural drawings such as the one represented in Fig. 7(a), with nine descriptions of doors and windows. With a computation time of 5 to 30 seconds on a SUN Sparc Ultra 1, the network recognizes most of the represented symbols (Fig. 7(c)). In table 1, we give for each drawing the number of symbols to be recognized (S), the number of symbols recognized by the system (R) and the number of false hits (F). Most of the latter stem from redundant recognition (e.g. double doors being also recognized as two simple doors).

Table 1
Performances of the symbol recognition method

Drawing #	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
S	14	11	15	12	12	15	14	14	16	14	15
R	13	11	14	11	10	15	14	14	14	13	14
F	1	0	3	2	0	0	1	4	1	4	2

5 Building the Network

For a set of descriptions, it is of course possible to build several networks, each relative to one description. But as we want to accelerate the symbol recognition process, our aim is to factorize as much as possible the constraints which are common to several symbols, and to find the most efficient ordering in a common network. For this, we use several heuristics. After constructing the root node NNSegment, we proceed incrementally and sequentially: the descriptions are ordered by increasing number of constraints, and added to the network one after the other. For each new symbol, we only add nodes for constraints which are not already tested in the network.

5.1 Constraints Which Are Already Tested

One of the strengths of this approach is the ability to use constraints common to several descriptions. When a new description is added to the network, we look for constraints in this description which the network can already test.

For this, we input the segments $sd_1, \dots, sd_t$ of the description to the network (t being the size of the description). When they propagate through the network, these segments will be checked by all the constraints already available there, and this yields features which correspond to one of the constraints of the description. This can be done by slightly modified versions of the algorithms used in the recognition phase: Alg. 4, 5 and 6 are very similar to the previous Alg. 1, 2 and 3.

Algorithm 4 NNSeg.transD($\mathcal{D}$ d)

```
for k ← 1 to size (d) do
    feat ← create one feature from $sd_{\mathrm{k}}$
    myFeatures.add (feat)
    for all my sons n do
        n.transD (feat, me, d)
    end for
end for
```

Algorithm 5 NNCond.transD($\mathcal{F}$f, $\mathcal{N}$ p, $\mathcal{D}$ d)

```
c2 ← create one constraint from f and myPredicate
if there is a constraint c of d such that c2 is an extension of c (§ 5.5) then
    myFeatures.add (f)
    for all my sons n do
        n.transD (f, me, d)
    end for
end if
```

Algorithm 6 NNMerge.transD($\mathcal{F}$ f, $\mathcal{N}$ p, $\mathcal{D}$ d)

```
for all features g disjoint of f and filtered by my other father (not p) do
    if p = myFather1 then
        newFeat ← merge (f, g)
    else
        newFeat ← merge (g, f)
    end if
    c2 ← create one constraint from newFeat and myPredicate
    if there is a constraint c of d such that c2 is an extension of c (§ 5.5) then
        myFeatures.add (newFeat)
        for all my sons n do
            n.transD (newFeat, me, d)
        end for
    end if
end for
```

After this propagation, the network contains several features, localized in all the nodes through which the segments verifying the description have been propagated. For instance, let us look at the following d_{new} description:

$d_{new} = \langle \{sd_1, sd_2, sd_3, sd_4\}, \{cc_1, cc_2, cc_3, cs_1\} \rangle$
$cc_1 = \langle (sd_1, sd_2), p_c : x \times y \mapsto point1(x) = point2(y) \rangle$
$cc_2 = \langle (sd_2, sd_3), p_c : x \times y \mapsto point1(x) = point2(y) \rangle$
$cc_3 = \langle (sd_1, sd_4), p_c : x \times y \mapsto point1(x) = point2(y) \rangle$
$cs_1 = \langle (sd_1), p_{cs1} : x \mapsto length(x) \leq 20 \rangle$

which we want to add to an existing network (Fig. 9(a)), where:

$p_{NC2} : x \mapsto length(x) \leq 20$
$p_{NM3} : x \times y \mapsto point1(x) = point2(y)$
$p_{NM4} : x \times y \times z \mapsto point1(y) = point2(z)$

After propagation of the model segments through the network, using the previously described algorithms, the network contains seven features (Fig. 9(b)).

5.2 Disjoint Features Set

Among the features present in the network after propagation (§ 5.1), let us choose a set of disjoint features, i.e. a set such that each segment of the model is present in one and only one feature. Generally, several choices are possible for such a set. All these choices are valid for the incrementation of the network. But it is better to choose the set which yields the most compact network. We therefore try to maximize the number of constraints already tested by the traversed nodes.

In the previous example with d_{new}, it is possible to create four disjoint features sets (Fig. 9(c), 9(d), 9(e) and 9(f)), but we choose the last set, for which two constraints are already tested by the network.

5.3 Adding Simple Constraints

After having chosen a set of disjoint features, we have to decide how to add the new nodes to the network. This depends on the order in which the remaining constraints have entered the network (Fig. 10). We decided to process the simple constraints sequentially, starting with those of smallest size (e.g. a constraint on only one segment will be processed before a constraint on several segments). This relies on the fact that the smallest constraints are supposed to have most discriminating power, and thus it is interesting to find them at an early stage in the network.

Before we create the node which will check the simple constraint, we must group the corresponding segments into a common feature. If they are not already grouped, i.e. if they are spread into different features, we use the `mergeFeatures` function (§ 5.4) to create the appropriate NNMerge nodes. When all segments concerned by the simple constraint are grouped, the corresponding NNCondition node can be added to the sons of the node where the last feature is found. The feature can then be removed from the latter node and added to the newly created node. Every time a new simple constraint is added to the network, we also check whether this new constraint can be used to verify other untested constraints of the description, to minimize the size of the network and improve its performances.

Finally, when all simple constraints have been added, the remaining connection constraints, which have not already been taken into account, are added with the `mergeFeatures` function (§ 5.4). As all segments of the description are related to each other through constraints, and as all constraints have been inserted in the network, all segments of the description are included in a common feature. The corresponding NNFinal node, which represents the new symbol to be recognized, can therefore be created and added to the sons of the node containing this feature.

5.4 Adding Connection Constraints

There are several cases where we need to merge features: to create a unique feature when adding a simple constraint, or to add the remaining connection constraints in a description (§ 5.3). The algorithm we use (Alg. 7) to add the nodes merging these features takes as arguments the list of features to be merged, the network where they are located, and the list of constraints between pairs of segments belonging to the features to be merged. The features are merged two by two, and this results in the creation of the corresponding NNMerge nodes. If no connection constraint is found, the node is created with a $true$ predicate.

When all the features have been merged into a single final feature, the remaining connection constraints, if any, related to two segments of this feature, are added to the network as NNCondition nodes.

For example, the merging of the three features located in three nodes (Fig. 11(a)), through the following connection constraints:

$$cc_1 = \langle (sd_5, sd_6), p_{cc1} : x \times y \mapsto (point1(x) = point2(y)) \rangle$$
$$cc_2 = \langle (sd_1, sd_2), p_{cc2} : x \times y \mapsto (point2(x) = point1(y)) \rangle$$
$$cc_3 = \langle (sd_4, sd_5), p_{cc3} : x \times y \mapsto (point2(x) = point2(y)) \rangle$$

leads to the creation of a NNMerge node for the cc_1 and cc_2 constraints (Figs. 11(b) and 11(c)), and to the creation of a NNCondition node for the cc_3 constraint (Fig. 11(d)).

Algorithm 7 mergeFeatures (list$\langle\mathcal{F}\rangle$ feat, network r, list$\langle\mathcal{C}\rangle$ const)

```
while size(feat) $>$ 1 do
  f1 $\leftarrow$ smallest feature of feat
  f2 $\leftarrow$ smallest feature of feat not equal to f1, such that there is a constraint cc
  in const between 2 segments from f1 and f2
  if no such feature f2 exists then
    f2 $\leftarrow$ smallest feature of feat not equal to f1
    p $\leftarrow$ true
  else
    p $\leftarrow$ predicate which tests cc
    remove cc from const
  end if
  newNode $\leftarrow$ create NNMerge from p and the nodes of r which contain f1 and
  f2
  add newNode to r
  remove f1 and f2 from r and from feat
  add the feature resulting from the merge of f1 and f2 to feat and to newNode
end while
while there are constraints cc left in const do
  add cc to a new NNCondition node, as son of the last new node
end while
```

5.5 Constraint Checking

Generally, the predicate tested by a NNMerge node or a NNCondition node is not equal to the predicate of the corresponding constraint. This stems from the fact that we can only check that the segments to which the constraint is related are *included* in the features received by the node, which can also contain other segments, or contain the right segments in another order than that expressed by the constraint. To take into account these variations, we generalize the predicates which are put into the nodes when we create the network.

Let $size$ be the recursive function defined by:

$$size : n \mapsto \begin{cases} 1 \text{ if } n \text{ is of type NNSegment} \\ size(father(n)) \text{ if } n \text{ is of type NNFinal or NNCondition} \\ size(father_1(n)) + size(father_2(n)) \text{ if } n \text{ is of type NNMerge} \end{cases}$$

The size of a NNMerge and NNCondition node, defined by this function, also corresponds to the size of the tested predicate and that of the feature which can be sent by the node.

Let d be the constraint that the node to be created must verify, and let m be the feature from which we create the node. If we create a NNCondition node, it is the feature coming from the father node of this node. If we create a NNMerge node, it is the union of the features coming from the two fathers. By definition, this feature m contains the segments to which constraint d refers. The predicate of d must be a restriction of the predicate p which we must add to the new node, modulus a change in the order of its arguments. We say that constraint c defined by $\langle m, p\rangle$ is an extension of d for m. The injection l defined on $[1, size(d)] \to [1, size(c)]$ by:

$$\begin{cases} \forall i \in [1, size(d)], n_i = m_{l(i)} \\ \forall x \in \mathcal{S}^{size(c)}, p(x_1, ..., x_{size(c)}) = q(x_{l(1)}, ..., x_{l(size(d))}) \end{cases}$$

gives the order of the arguments for the two predicates, as it returns the location of the segments of d's feature in c's feature. For a feature m containing all segments of the d's feature, the injection l is defined uniquely. Actually, there is only one constraint c, having m as its feature, for which c is an extension of d. For example, for the feature (sd_2, sd_1, sd_3, sd_4), $c = \langle(sd_2, sd_1, sd_3, sd_4), p : x \times y \times z \times w \mapsto length(z) = 2.length(y)\rangle$ is an extension of $d = \langle(sd_3, sd_1), q : x \times y \mapsto length(x) = 2.length(y)\rangle$.

6 Limitations and perspectives

6.1 Building the network

6.1.1 Descriptive power

We are aware that the language we have defined (Fig. 2) lacks descriptive power when the constraints become more complex. We are currently studying alternatives, such as specialized constraint description languages.

6.1.2 Global construction

The building of the network, using the heuristics presented previously, is robust and fast. But the process cannot guarantee that the resulting network is the most compact, even if many constraints are factorized. This stems from the fact that the network is built sequentially, without any comparison between the descriptions which have already been input, to localize global common constraints. Thus, depending on the order in which the descriptions are analyzed, some factorizations may or may not be found by the network, and the resulting networks may be different and more or less efficient.

For instance, let d_1 and d_2 be two descriptions defined by:

$$d_1 = \langle\ 3,\ \{cc_1, cc_2\},\ \emptyset\ \rangle$$
$$d_2 = \langle\ 3,\ \{cc_3, cc_4\},\ \{cs_1\}\ \rangle$$

$$cc_1 = \langle\ (sd_1, sd_2),\ p_{cc1} : x \times y \mapsto point1\,(\,x\,) = point2\,(\,y\,)\ \rangle$$
$$cc_2 = \langle\ (sd_2, sd_3),\ p_{cc2} : x \times y \mapsto point2\,(\,x\,) = point2\,(\,y\,)\ \rangle$$

$$cc_3 = \langle\ (sd_1, sd_2),\ p_{cc3} : x \times y \mapsto point2\,(\,x\,) = point2\,(\,y\,)\ \rangle$$
$$cc_4 = \langle\ (sd_2, sd_3),\ p_{cc3} : x \times y \mapsto point1\,(\,x\,) = point1\,(\,y\,)\ \rangle$$
$$cs_1 = \langle\ (sd_1, sd_2),\ p_{cs1} : x \mapsto length\,(\,x\,) = length\,(\,y\,)\ \rangle$$

When building the network recognizing these two symbols, d_1 is added before d_2, as d_1 and d_2 have the same size, and d_1 has less constraints than d_2.

Two different networks can be built for detecting d_1: P_1, which starts by checking the cc_1 constraint (NM2), and then the cc_2 constraint (NM3) (Fig. 12(a)), and Q_1, which starts by checking cc_2 (NM2), and then checks cc_1 (NM3) (Fig. 12(c)). The node predicated checked by these two networks are defined by:

$p_{NM2} : x \times y \mapsto point1\,(\,x\,) = point2\,(\,y\,)$
$p_{NM3} : x \times y \times z \mapsto point2\,(\,y\,) = point2\,(\,z\,)$

$q_{NM2} : x \times y \mapsto point2\,(\,x\,) = point2\,(\,y\,)$
$q_{NM3} : x \times y \times z \mapsto point1\,(\,x\,) = point2\,(\,y\,)$

Then, description d_2 is added to the P_1 network. When propagating the features of d_2 into this network (§ 5.1), as the NM2 node of P_1 does not check any d_2's constraints, no feature is propagated and new nodes must be created for each of d_2's constraints. The resulting P_2 network then contains eight nodes (Fig. 12(b)).

Conversely, when adding d_2 to the Q_1 network, the NM2 node of Q_1 actually checks one of d_2's constraints, i.e. cc_3, and this node can thus be reused for the detection of d_2. The resulting Q_2 network only contains seven nodes (Fig. 12(d)). The predicates of the new nodes added to P_2 and Q_2 are:

$p_{NM5} : x \times y \mapsto point2\,(\,x\,) = point2\,(\,y\,)$
$p_{NC6} : x \times y \mapsto length\,(\,x\,) = length\,(\,y\,)$
$p_{NM7} : x \times y \times z \mapsto point1\,(\,y\,) = point1\,(\,z\,)$

$q_{NC5} : x \times y \mapsto length\,(\,x\,) = length\,(\,y\,)$
$q_{NM6} : x \times y \times z \mapsto point1\,(\,y\,) = point1\,(\,z\,)$

In our present implementation, there is no strategy to choose the best of these two networks[3]. The resulting network only depends on the input order of the constraints in the description file.

In this simple case, the size difference is only one node. But in real cases, when the network is built from many descriptions, containing several constraints, the resulting may become less than optimal. An additional network optimization strategy should be added, taking into account a global view of all constraints, especially in the buildup phase.

6.2 *Use of the network*

We have already mentioned that our error computations are quite simplistic and ought to be enhanced (§ 5.5). The use of the network can also be enhanced in other ways.

[3] Actually, in the case demonstrated here, our system builds P_1, followed by P_2.

6.2.1 Dependence on the low level

Although we work only on thin lines (§ 4.3.1) and use an error measure, the network still relies on the quality of the vectorization, as it is designed to perform one-to-one matches at the segment level. Thus, missing or extraneous features often lead to recognition errors.

For instance, let us assume that we use the network to detect a triangle in the vectorized image of a triangle. If the vectorization contains an extraneous segment connecting two extremities (Fig. 13(a)), the error is small and a correct triangle will still be detected (Fig. 13(b)). But if the vectorization split one of the triangle sides into two segments (Fig. 13(c)), the error measure is greater. If it is too large, nothing is detected; else, the triangle detected by the network, (Fig. 13(d) or 13(e)), does not correspond to the right solution.

This weakness is the main reason for the observed failures of our symbol recognition method, in all our experiments, with architectural drawings as with electronic diagrams. There are two possible solutions to enhance this, and we investigate both:

- We work on enhancing the precision and quality of the vectorization step. This has been—and still is—the goal for many researchers in the realm of graphics recognition, of course.
- We also are considering adding some many-to-one matching steps in the constraints checking mechanisms, although this will add to the total complexity of the method.

6.2.2 Parameters of the network

One of the strong features of our approach is the possibility to look for all symbols in a single step, instead of testing each model-to-image matches individually. However, the drawback is that when looking only for a single symbol or for a subset of all possible symbols, the proposed approach is complex. Although the search area can be restricted to a subimage (§ 4.3.4), the present method does not provide the possibility to only look for a subset of the model symbols, without rebuilding a new network with this subset. A possible solution would be to add an activation flag for each node in the network; only activated nodes would then filter and propagate at any given time. If the task is to only look for a given set of symbols, we need a way to quickly retrieve and activate the corresponding nodes in the network.

Another limitation is that our present description language does not allow for parameters. The dimensioning features are given explicitely by the description, whereas it would be interesting to add variables to the description languages, for instance to describe possible dependencies between several measures.

7 Conclusion

We have presented an adaptation of Messmer and Bunke's network approach to constraint representation and feature propagation. Despite the limitations discussed in § 6, the method has several strong points, such as the factorization capabilities, its adaptability and flexibility, and its independence of the geometry and topology of the symbols. For instance, concerning this last point, many symbol recognition methods need clearly separable symbols, which may be easy for wiring diagrams, where there are basically symbols and lines, but which is much more difficult in architecture, when the symbols are completely "immerged" in the drawing. Our first results are promising. We are working on improving the low-level processing, for a better input to the system. In addition to the improvements and perspectives suggested in § 6, we also need to evaluate the performances of the method on a larger set of model symbols, to test the scalability of the approach when the number of models to recognize is 10 times larger than presently.

Acknowledgments

This paper is a thoroughly revised and extended version of a paper first presented at SSPR'98 (Ah-Soon and Tombre 1998). The work was partly funded by financial support from Région Lorraine.

References

Ah-Soon, C. and K. Tombre (1997). Variations on the Analysis of Architectural Drawings. In *Proceedings of 4th International Conference on Document Analysis and Recognition, Ulm (Germany)*, pp. 347–351.

Ah-Soon, C. and K. Tombre (1998, August). Network-Based Recognition of Architectural Symbols. In A. Amin, D. Dori, P. Pudil, and H. Freeman (Eds.), *Advances in Pattern Recognition (Proceedings of Joint IAPR Workshops SSPR'98 and SPR'98, Sydney, Australia)*, Volume 1451 of *Lecture Notes in Computer Science*, pp. 252–261.

Chhabra, A. K. (1998, April). Graphic Symbol Recognition: An Overview. In K. Tombre and A. K. Chhabra (Eds.), *Graphics Recognition—Algorithms and Systems*, Volume 1389 of *Lecture Notes in Computer Science*, pp. 68–79. Springer-Verlag.

Cordella, L. P. and M. Vento (1999, September). Symbol and Shape Recognition. In *Proceedings of 3rd International Workshop on Graphics Recognition, Jaipur (India)*, pp. 179–186.

Dosch, P., C. Ah-Soon, G. Masini, G. Sánchez, and K. Tombre (1999). Design of an Integrated Environment for the Automated Analysis of Architectural

Drawings. In S.-W. Lee and Y. Nakano (Eds.), *Document Analysis Systems: Theory and Practice. Selected papers from Third IAPR Workshop, DAS'98, Nagano, Japan, November 4–6, 1998, in revised version*, Lecture Notes in Computer Science 1655, pp. 295–309. Berlin: Springer-Verlag.

Lladós, J., J. López-Krahe, and E. Martí (1997). A System to Understand Hand-Drawn Floor Plans Using Subgraph Isomorphism and Hough Transform. *Machine Vision and Applications 10*(3), 150–158.

Lladós, J. and E. Martí (1999). A Graph-Edit Algorithm for Hand-Drawn Graphical Document Recognition and Their Automatic Introduction into CAD Systems. *Machine Graphics & Vision 8*(2), 195–211.

Messmer, B. T. and H. Bunke (1996, May). Automatic Learning and Recognition of Graphical Symbols in Engineering Drawings. In R. Kasturi and K. Tombre (Eds.), *Graphics Recognition—Methods and Applications*, Volume 1072 of *Lecture Notes in Computer Science*, pp. 123–134. Springer-Verlag.

Messmer, B. T. and H. Bunke (1998, May). A New Algorithm for Error-Tolerant Subgraph Isomorphism Detection. *IEEE Transactions on PAMI 20*(5), 493–504.

Miyao, H. and Y. Nakano (1996, May). Note Symbol Extraction for Printed Piano Scores Using Neural Networks. *IEICE Transactions on Information and Systems E79-D*(5), 548–554.

Pasternak, B. (1996, April). *Adaptierbares Kernsystem zur Interpretation von Zeichnungen*. Dissertation zur Erlangung des akademischen Grades eines Doktors der Naturwissenschaften (Dr. rer. nat.), Universität Hamburg.

Samet, H. and A. Soffer (1998, June). MAGELLAN: Map Acquisition of GEographic Labels by Legend ANalysis. *International Journal on Document Analysis and Recognition 1*(2), 89–101.

Tombre, K., C. Ah-Soon, P. Dosch, A. Habed, and G. Masini (1998, August). Stable, Robust and Off-the-Shelf Methods for Graphics Recognition. In *Proceedings of the 14th International Conference on Pattern Recognition, Brisbane (Australia)*, pp. 406–408.

Tombre, K., C. Ah-Soon, P. Dosch, G. Masini, and S. Tabbone (1999, September). Stable and Robust Vectorization: How to Make the Right Choices. In *Proceedings of 3rd International Workshop on Graphics Recognition, Jaipur (India)*, pp. 3–16. Revised version to appear in a forthcoming LNCS volume.

Valveny, E. and E. Martí (1999, September). Application of Deformable Template Matching to Symbol Recognition in Hand-written Architectural Drawings. In *Proceedings of 5th International Conference on Document Analysis and Recognition, Bangalore (India)*, pp. 483–486.

Yu, Y., A. Samal, and S. C. Seth (1997, August). A System for Recognizing a Large Class of Engineering Drawings. *IEEE Transactions on PAMI 19*(8), 868–890.

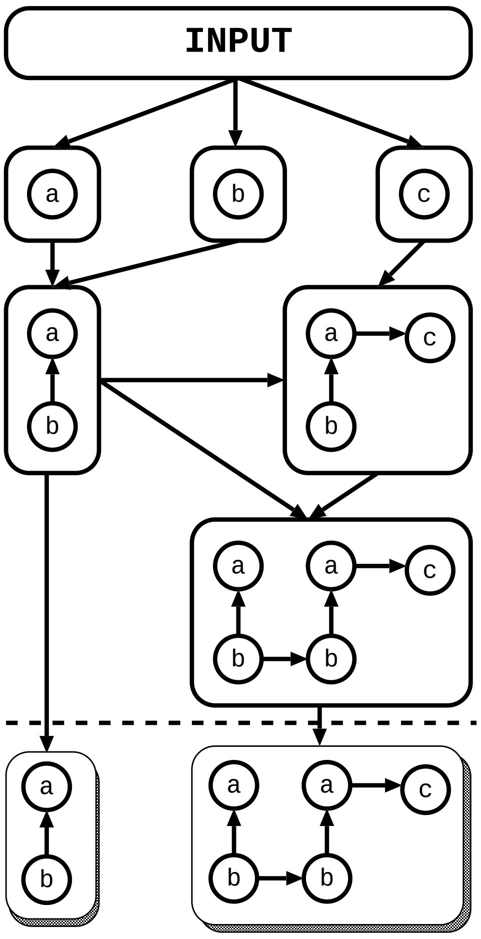


Fig. 1. Messmer’s network.

symbol ↦ # **type** *SEGMENT int ARC int* {**constraint**;}$^{+}$ **instantiation**

constraint ↦ **constraint** *or* **constraint** | **constraint** *and* **constraint**

constraint ↦ **point** == **point** | **real** **op_comp_** **real**

real ↦ *real* | *min* (**real** , **real**) | *max* (**real** , **real**) | *abs* (**real**)

real ↦ **point**.*x* ()| **point**.*y*() | **point**.*distance* (**point**)

real ↦ **prim**.**f_elem**() | **prim**.**f_angle**(**prim**) | **real** **op_alg** **real**

real ↦ **arc**.*radius*() | **point**.*angle*(**point**, **point**)

f_elem ↦ *length* | *width* | *x* | *y*

f_angle ↦ *pt1_angle_pt1* | *pt1_angle_pt2* | *pt2_angle_pt1* | *pt2_angle_pt2*

point ↦ **prim**.*point1* ()| **prim**.*point2*() | **arc**.*center*()

prim ↦ **arc** | **segment**

arc ↦ *arc int*

segment ↦ *seg int*

op_comp ↦ $<$ | $>$ | $<=$ | $>=$ | $==$

op_alg ↦ $*$|/| $+$ |$-$

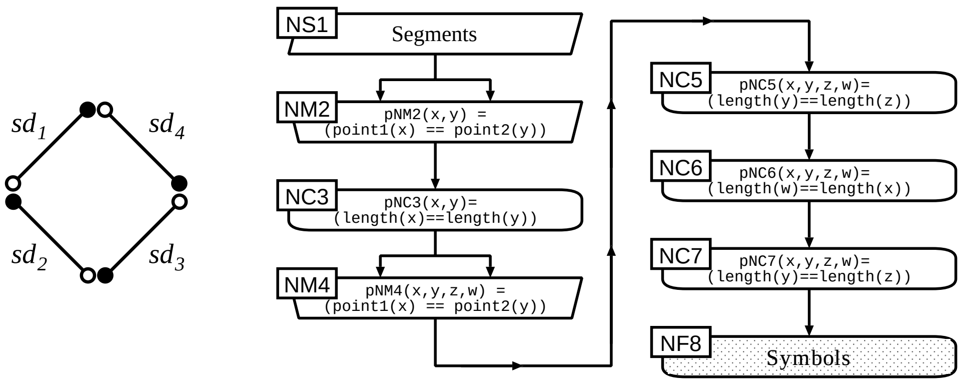
instantiation ↦ **type** ({ **real** | **point** } { ,**real** | ,**point** }*)

type↦ *door* | *double_door* | *window* | *double_window* | ...

Fig. 2. Grammar of the description language.

```
#lozenge
SEG 4 ARC 0
s1.point1() == s2.point2(); s1.length() == s2.length();
s2.point1() == s3.point2(); s2.length() == s3.length();
s3.point1() == s4.point2(); s3.length() == s4.length();
s4.point1() == s1.point2(); s4.length() == s1.length();
```

Fig. 3. Description of a lozenge.



sd1
sd4
sd2
sd3
NS1 Segments
NM2 pNM2(x,y) = (point1(x) == point2(y))
NC3 pNC3(x,y)= (length(x)==length(y))
NM4 pNM4(x,y,z,w) = (point1(x) == point2(y))
NC5 pNC5(x,y,z,w)= (length(y)==length(z))
NC6 pNC6(x,y,z,w)= (length(w)==length(x))
NC7 pNC7(x,y,z,w)= (length(y)==length(z))
NF8 Symbols

Fig. 4. Network for recognition of a lozenge.

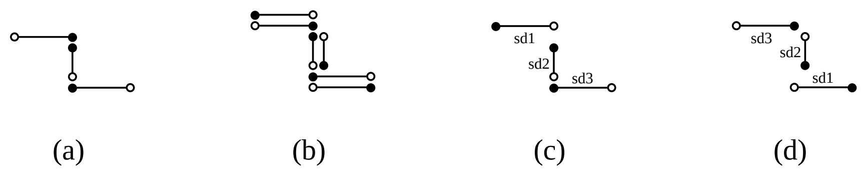
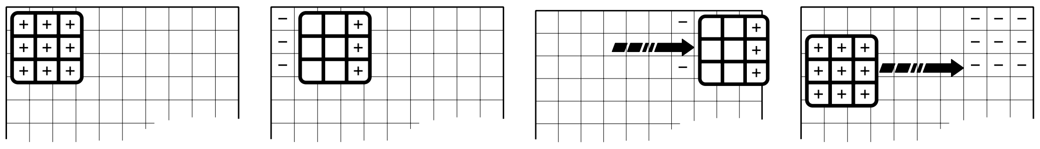
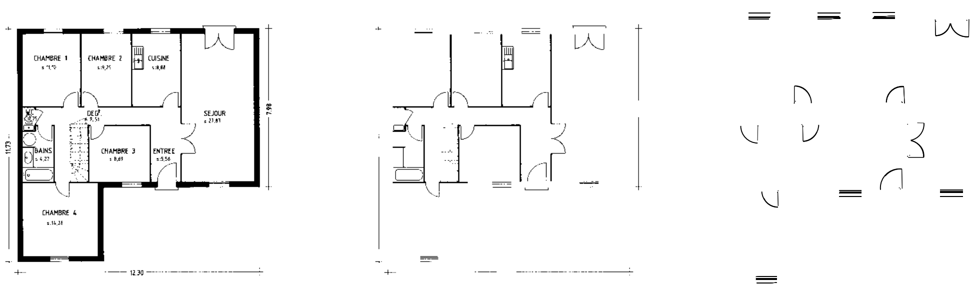


Fig. 5. Inversion of segments—white dots represent origin points and black dots end points: (a) segments yielded by vectorization; (b) segments sent to the network; (c) and (d) segments verifying constraints cc_1 and cc_2.



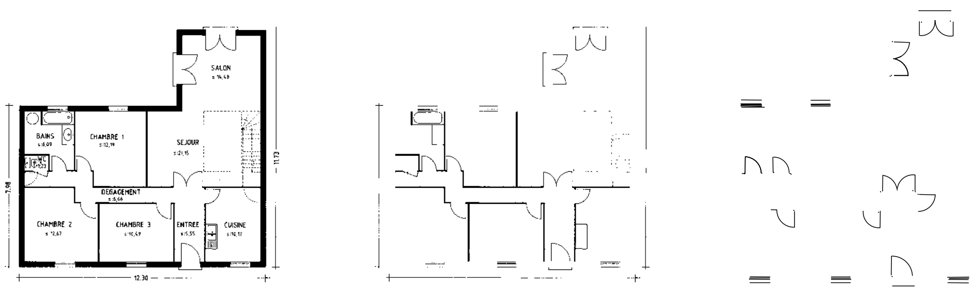
(a) Start of a line. (b) Moving the window. (c) End of line. (d) New line.

Fig. 6. Moving the symbol detection window. The meshes labeled + contain features input into the network, those labeled - contain features which are withdrawn from the network.



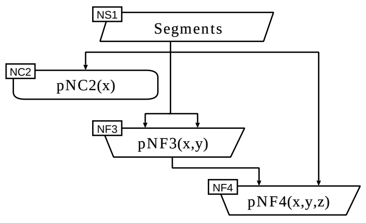
(a) A drawing. (b) Thin lines. (c) Symbols.

Fig. 7. Result for a simple drawing (1700×1600 pixels) vectorized in about 300 segments and arcs.

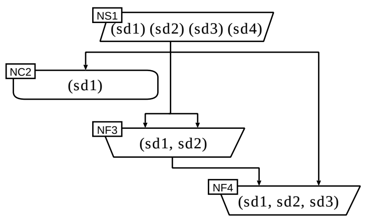


(a) A drawing. (b) Thin lines. (c) Symbols.

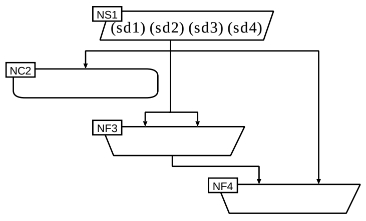
Fig. 8. Result for a simple drawing (1700×1600 pixels) vectorized in about 300 segments and arcs.



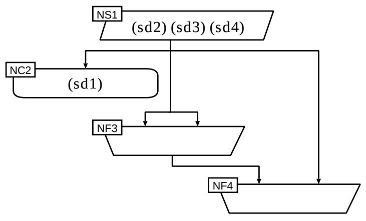
(a) Subset of a network.



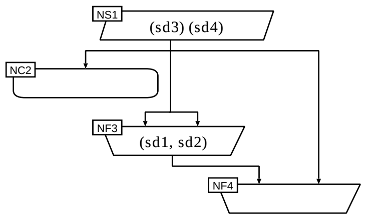
(b) Possible mappings.



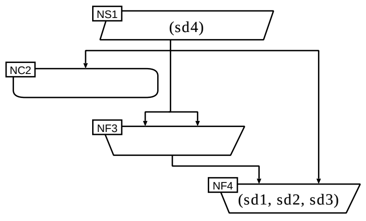
(c) No tested constraint.



(d) cs_1 tested.

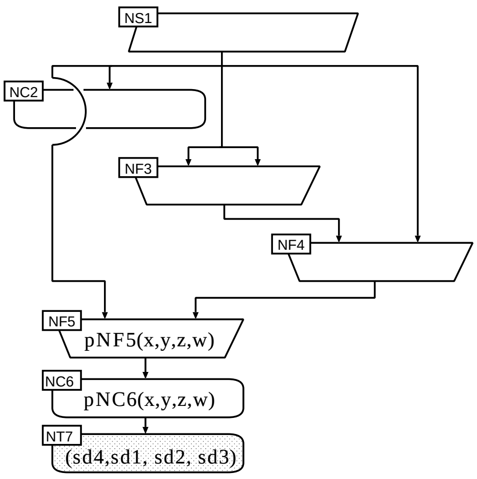


(e) cc_1 tested.

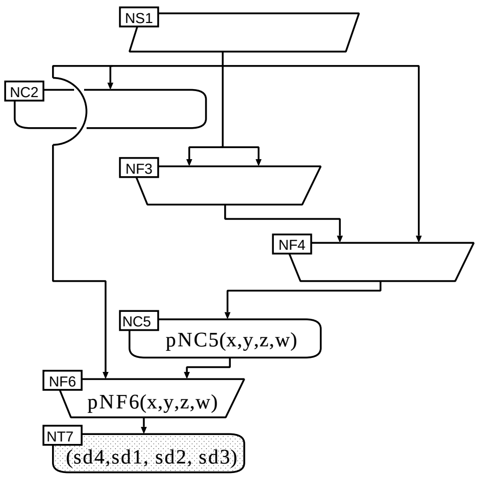


(f) cc_1 and cc_2 tested.

Fig. 9. Looking for constraints already tested by the network.

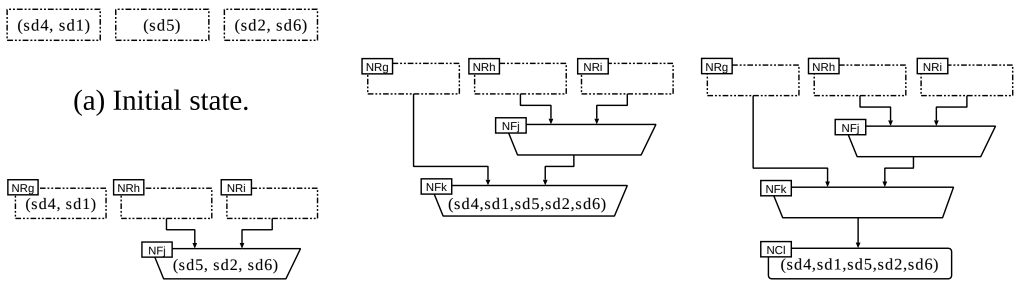


(a) cc_3 before cs_1.



(b) cs_1 before cc_3.

Fig. 10. Two choices when adding the remaining constraints of d_{new}, starting from (Fig. 9(f)).



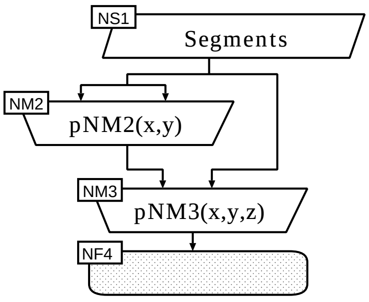
(a) Initial state.

(b) Adding cc_1.

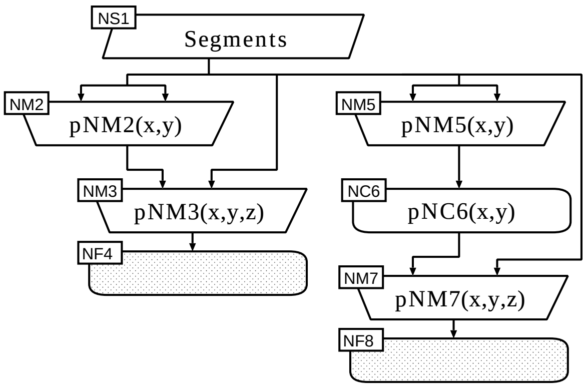
(c) Adding cc_2.

(d) Adding cc_3.

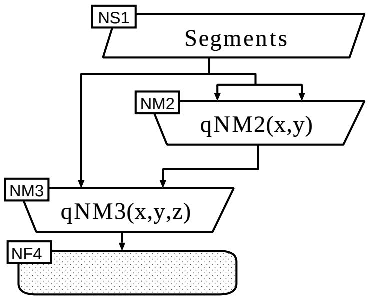
Fig. 11. Merging different features by using connection constraints.



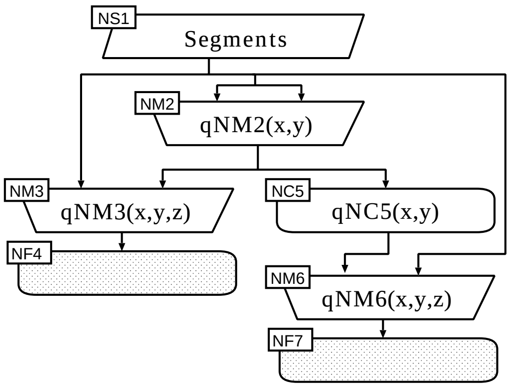
(a) Starting with P_1,...



(b) ...P_2 contains 8 nodes.



(c) Starting with Q_1,...



(d) ...Q_2 contains only 7 nodes.

Fig. 12. Non-optimal building of a network.

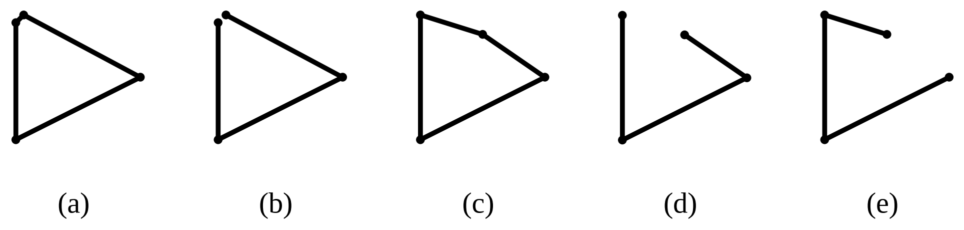


Fig. 13. Recognizing a triangle with the network.