UNIVERSITÀ DEGLI STUDI DI SALERNO

Dipartimento di Informatica

Dottorato di Ricerca in Informatica
XII Ciclo - Nuova Serie

Tesi di Dottorato in

New Methods, Techniques and Applications for Sketch Recognition

Mattia De Rosa

Ph.D. Program Chair
Prof. Giuseppe Persiano

Supervisors
Prof. Gennaro Costagliola

Dott. Vittorio Fuccella

Anno accademico 2013/2014

Alla mia famiglia

Abstract

The use of diagrams is common in various disciplines. Typical examples include maps, line graphs, bar charts, engineering blueprints, architects' sketches, hand drawn schematics, etc.. In general, diagrams can be created either by using pen and paper, or by using specific computer programs. These programs provide functions to facilitate the creation of the diagram, such as copy-and-paste, but the classic WIMP interfaces they use are unnatural when compared to pen and paper. Indeed, it is not rare that a designer prefers to use pen and paper at the beginning of the design, and then transfer the diagram to the computer later.

To avoid this double step, a solution is to allow users to sketch directly on the computer. This requires both specific hardware and sketch recognition based software. As regards hardware, many pen/touch based devices such as tablets, smartphones, interactive boards and tables, etc. are available today, also at reasonable costs. Sketch recognition is needed when the sketch must be processed and not considered as a simple image and it is crucial to the success of this new modality of interaction. It is a difficult problem due to the inherent imprecision and ambiguity of a freehand drawing and to the many domains of applications. The aim of this thesis is to propose new methods and applications regarding the sketch recognition. The presentation of the results is divided into several contributions, facing problems such as corner detection, sketched symbol recognition and autocompletion, graphical context detection, sketched Euler diagram interpretation.

The first contribution regards the problem of detecting the corners present in a stroke. Corner detection is often performed during preprocessing to segment a stroke in single simple geometric primitives such as lines or curves.

The corner recognizer proposed in this thesis, RankFrag, is inspired by the method proposed by Ouyang and Davis in 2011 and improves the accuracy percentages compared to other methods recently proposed in the literature.

The second contribution is a new method to recognize multi-stroke hand drawn symbols, which is invariant with respect to scaling and supports symbol recognition independently from the number and order of strokes. The method is an adaptation of the algorithm proposed by Belongie et al. in 2002 to the case of sketched images. This is achieved by using stroke related information. The method has been evaluated on a set of more than 100 symbols from the *Military Course of Action* domain and the results show that the new recognizer outperforms the original one.

The third contribution is a new method for recognizing multi-stroke *partially* hand drawn symbols which is invariant with respect to scale, and supports symbol recognition independently from the number and order of strokes. The recognition technique is based on subgraph isomorphism and exploits a novel spatial descriptor, based on polar histograms, to represent relations between two stroke primitives. The tests show that the approach gives a satisfactory recognition rate with partially drawn symbols, also with a very low level of drawing completion, and outperforms the existing approaches proposed in the literature. Furthermore, as an application, a system presenting a user interface to draw symbols and implementing the proposed autocompletion approach has been developed. Moreover a user study aimed at evaluating the human performance in hand drawn symbol autocompletion has been presented. Using the set of symbols from the *Military Course of Action* domain, the user study evaluates the conditions under which the users are willing to exploit the autocompletion functionality and those under which they can use it efficiently. The results show that the autocompletion functionality can be used in a profitable way, with a drawing time saving of about 18%.

The fourth contribution regards the detection of the graphical context of hand drawn symbols, and in particular, the development of an approach for identifying *attachment areas* on sketched symbols. In the field of syntactic recognition of hand drawn visual languages, the recognition of the relations

among graphical symbols is one of the first important tasks to be accomplished and is usually reduced to recognize the attachment areas of each symbol and the relations among them. The approach is independent from the method used to recognize symbols and assumes that the symbol has already been recognized. The approach is evaluated through a user study aimed at comparing the attachment areas detected by the system to those devised by the users. The results show that the system can identify attachment areas with a reasonable accuracy.

The last contribution is EulerSketch, an interactive system for the sketching and interpretation of Euler diagrams (EDs). The interpretation of a hand drawn ED produces two types of text encodings of the ED topology called *static code* and *ordered Gauss paragraph* (OGP) code, and a further encoding of its regions. Given the topology of an ED expressed through static or OGP code, EulerSketch automatically generates a new topologically equivalent ED in its graphical representation.

Acknowledgement

I would like to thank my advisor Gennaro Costagliola and Vittorio Fuccella, who coauthored most of the papers related to this thesis, for the profitable discussions that contributed to my research work and for the suggestions and help they gave me during the preparation of this thesis. I also thank Vittorio Fortino for the contribution on the development of RankFrag and Paolo Bottoni, Andrew Fish and Rafiq Saleh, who have been co-authors for recent research papers on Euler diagrams related to the results presented in this thesis.

Contents

Chapter 1

Introduction

The use of diagrams is common in various disciplines. The term can have different meanings depending on the context, and in fact, there is no single definition of diagram in the scientific literature. In general, a diagram can be defined as a visual representation of some information and it is usually two-dimensional and geometric. Under this definition, typical examples of diagrams include maps, line graphs, bar charts, engineering blueprints, architects' sketches, hand drawn schematics, etc.. Diagrams are often used to represent the information in a natural way. It is generally easier, faster and more convenient to grasp the meaning of a visual representation than the meaning of a textual representation. In some cases the use of diagrams is practically inevitable, given that an equivalent textual representation would be too long and complex (for example in the case of architectural plan, etc.). The notion that a complex idea can be conveyed with just a single still image is common knowledge, as in the adage "A picture is worth a thousand words".

In general, diagrams can be created either by using pen and paper, or by using specific computer programs. With the introduction of the mouse, these programs have started to use the same paradigm, in which the use of a palette allows to select and place individual elements of the diagram on a canvas. These programs provide functions to facilitate the creation of the diagram, such as copy-and-paste, automatic layout and much more. However, the interfaces of these programs are still unnatural when compared to pen

and paper, and the way they work is often dictated by technical limitations rather than by the needs of the users.

Indeed, it is not rare that a designer prefers to use pen and paper at the beginning of the design, and then transfer the diagram to the computer later when the idea has taken its almost final shape. For many people, pen and paper are easier and faster to use, they are also more flexible (because paper does not have an explicit interface) and promote increased creativity, supporting also activities in which the vagueness of the result is an essential feature. Also, informal sketching can be convenient in the context of group work.

The effort necessary for the creation of informal sketches is typically much lower than that for the creation of more precise, formal diagrams with mouse and palette, even though it requires an additional effort for subsequently transferring the sketches to the computer. To solve this drawback, one possibility is therefore to allow users to sketch directly on the computer. In this case, specific hardware for drawing is necessary, such as pen tablets and touch screens. This hardware has become very popular in recent years in the form of drawing boards, tablets and smartphones which allow the use of pens, fingers, or both.

The use of these types of input devices enables a paradigm of communication with the computer completely different from that of the classical WIMP (window, icon, menu, pointing device) user interfaces. Of course to better exploit this new paradigm, dedicated software needs to be used.

This new paradigm allows to study the possibility of putting together the best features of the two types of interaction: a very simple and natural interface such as pen and paper, and the functions only possible with a computer program such as copy and paste, multiple saves, etc. In this context, the possibility arises to allow the user to draw a sketch directly on the computer, and then, through a recognition operation, automatically convert it into a more formal form, equivalent to that which can be produced with a classical WIMP interface. This operation is realized through techniques of sketch recognition, and marks the difference from a simple drawing program. The use of sketch recognition techniques allows, among other things, to

provide additional functionalities in software systems, such as real time help, more powerful editing, autocompletion, beautification, automatic layout (to remove clutter and confusion), interpretation and translation of the sketch.

Sketching recognition is also used to resolve problems created by the replacement of paper and pen with WIMP systems. As an example, customers (e.g. of an architect) may be reluctant to criticize drawings that look too "finished". Given that low-fidelity sketches do not have this problem, it is possible to show them to the customers and then use sketch recognition capabilities to produce the "finished" version of the drawing (without additional work).

On the other hand, sketch recognition is a difficult problem due to the inherent imprecision and ambiguity of a freehand drawing. Because of this, the problem of sketch recognition is usually divided into multiple specific "easier" problems. Often the methods used to solve these individual problems are not tied exclusively to the sketch recognition and are also used in other domains.

The next section will describe the main aspects of the sketch recognition, while Section 1.2 will briefly describe the new methodologies proposed in this thesis, which will then be discussed in detail in later chapters.

1.1 Key aspects of sketch recognition

This section discusses the general terms and definitions in the field of sketch recognition.

A central element of sketching is the stroke. On a touch screen, a stroke starts with the pressure of the pen (or finger) on the screen and ends when the pen is raised. Technically, a stroke is a finite list of triples (x, y, t) (or samples) where (x, y) are the pair of coordinates in which the pen was at the time t. Is possible to extend this definition by adding the pressure applied on the screen (if the hardware can detect it), therefore a stroke becomes a list of quadruples (x, y, t, p). This value can be used to improve the accuracy of the recognition process, or only to vary the thickness of the displayed stroke.

The main problem in a sketch recognition system is, as the name implies,

the recognition, or rather the recognition of the individual symbols that compose a sketch starting from the individual strokes. A symbol can be considered one of the basic blocks that constitute a sketch in a given domain. Of course, in order to identify individual symbols, they must be included in the definition of the given domain. Sometimes a drawing can also include text to add additional information.

Sometimes the recognition is divided into two levels, low-level recognition (or preprocessing) and high-level recognition. In this case the first step is used to extract intermediate information from the strokes, while the second uses this information to recognize the symbols.

As already pointed out, recognition is a difficult problem, both because hand-drawing can be imprecise and ambiguous, and also because there is a lot of variability in the drawing of different people and it is not uncommon to see significant variations even by the same person.

After the identification of the individual symbols the next step is understanding the sketch as a whole. This phase depends greatly on the approach and on the domain. For example, it is possible to analyze the relationships between the different symbols in order to obtain meaningful information.

The various approaches for the recognition vary in the characteristics required in the input. The simplest approaches require that each stroke corresponds to a single symbol, thus simplifying the recognition. To remove this requirement it is necessary to solve two types of problems (often associated with the preprocessing): the *clustering* in which different strokes must be put together to represent a symbol, and the *segmentation* in which a stroke must be divided if it contributes to more than one symbol. In some cases, these steps can be used to group/split individual strokes in order to get simple geometric *primitives* (such as a line or a curve), instead of the whole symbol. The first type of approach is called *single-stroke*, while the second *multi-stroke*.

Technically, when the recognition is performed from strokes, it is called *online*, when it is performed from a raster image it is called *off-line*. In addition, if the recognition can begin after (or even during) the input of each stroke, it is called *eager* recognition, while if it begins after the drawing is complete it is called *lazy* recognition. Eager recognition may be associated

with autocompletion and with the ability to provide immediate feedback to the user, even before s/he has finished the drawing. In this case, the recognition needs to be fast enough to allow a fluid interaction with the user.

As regards the design of sketch-based user interfaces, there is a need of distinguishing drawing strokes from editing commands. For example, a user interface may allow the deletion of a symbol by drawing a cross above it. This introduces further ambiguity, as it becomes necessary to determine whether a stroke is part of the input drawing or represents an editing command. To avoid this ambiguity, it is possible to use approaches in which the user must change the input mode depending on the operation that s/he intends to perform (for example, by clicking on a softbutton "pen" to insert a stroke or a softbutton "rubber" to erase). The latter approach is called "mode-based", while the former "mode-less". Generally, modes are to be avoided, even though they simplify recognition. For example, in the case of handwriting input it is possible to use both approaches, but the mode-less writing adds the difficulty of distinguishing between text and drawings, which is still an open issue.

1.2 Proposed work

The aim of this thesis is to propose new methods and applications regarding the sketch recognition by facing problems such as corner detection, sketched symbol recognition and autocompletion, graphical context detection, sketched euler diagram interpretation. The methodologies presented can be used as intermediate steps in the broader problem of the recognition of an entire diagram, and for this reason are logically linked to each other.

A first contribution regards the detection of the corners present in a stroke. Corner detection is often performed during the phase of preprocessing to segment a stroke into single simple geometric primitives such as lines or curves. The so obtained segments can then be used as input for a sketch recognition algorithm. The corner recognizer proposed in this thesis, RankFrag, is inspired by the method proposed by Ouyang and Davis in [1] and improves the accuracy percentages compared to other methods recently proposed in the literature.

The second contribution regards the recognition of multi-stroke hand drawn symbols. Symbol recognition is one of the main "basic" issues in the context of sketched diagram recognition. In fact, when an entire diagram is to be recognized, it is usually necessary to proceed with the recognition of the individual symbols composing it and of the connectors among them. The proposed method is an adaptation of the image-based matching algorithm by Belongie et al. [2]. As the original algorithm, the proposed solution is invariant with respect to scaling and is independent from the number and order of the drawn strokes. Furthermore, it has a better recognition accuracy than the original one when applied to hand drawn symbols as resulting from the evaluation of the method on a set of more than 100 symbols belonging to the *Military Course of Action* domain. This is due to the exploitation of information on stroke points, such as the temporal sequence and occurrence information in a given stroke.

The third contribution regards the recognition of partially drawn symbols. The proposed method is invariant with respect to scale and uses an *Attributed Relational Graph* (ARG) [3] to represent symbols. Furthermore, the user can draw a symbol with the desired number of strokes and in any order. The method works with a single *perfect* template for each class, without the need of a training phase to extract features or to select multiple templates. Being based on subgraph matching, the recognition can be performed on partially drawn symbols, i.e., when only a part of the primitives composing the symbol is available. An innovation of the presented method is the use of a single spatial descriptor to represent relations between symbol components. The descriptor is an adaptation of the *shape context* [2] and its use makes the method free from the identification of the type of the *primitives* and from the check of fuzzy relations. The symbol matching is performed through an approximate graph matching procedure which incrementally produces new results as soon as more input strokes are available. Furthermore, a system presenting a user interface to draw symbols and implementing the proposed autocompletion method has been developed. Since autocompletion has proven to be an effective and appreciated feature when considering text editing applications [4, 5, 6, 7] but there are no evaluations of the performance of interfaces for autocompletion

of hand drawn symbols in the scientific literature, the system has been used to test the hand drawn symbol autocompletion functionality from the point of view of the benefits for the users. The user study has involved 14 participants and has shown that the users can exploit the autocompletion functionality in a profitable way, obtaining a faster input, with a time saving of about 18% and an increased accuracy.

The fourth contribution regards graphical context detection, and in particular, the identification of the attachment areas on sketched symbols. According to a largely accepted model in the visual language community the relations between the symbols of a diagram are geometrically defined through *attachment areas* of the symbols. For example, an arc of a graph is *entering* a node if the *head* of the arc is physically connected to the node *boundary*. Here, the relation *entering* between arc and node is defined on the attachment area *head* of the arc and the attachment area *boundary* of the node. Attachment areas can have different shapes and are generally related to the physical appearance of the symbol. In WIMP-based systems, the attachment areas are automatically reported by the system. In a sketched language, due to the impreciseness of hand-drawing, actual attachment areas of symbols may be heavily deformed [8]. The management of areas which are not delimited by the visible ink of the drawn symbol can be even more difficult. The proposed approach is independent from the domain of the symbols and from the method used to recognize symbols and assumes that the symbol has already been recognized. This also means that the ink drawn by the user to sketch the symbol has already been separated from the other ink in the diagram. The approach requires that the symbol and, more precisely, both its physical and logical features, are defined in vector graphics. The identification of the attachment areas is performed by establishing a mapping between sampled points of both the sketched and the template symbol. The approach is evaluated through a user study in which users are required to sketch symbols from different domains and then to identify attachment areas on the drawn symbol.

Finally, the last contribution regards the development of EulerSketch, an interactive system for the sketching and interpretation of Euler diagrams (EDs). EDs are used for visualizing relationships between set-based data [9].

They consist of a set of curves representing sets and their relationships. EDs are, as example, utilized in various information presentation applications as a simple, yet effective means of representing and interacting with set-based relationships. Given the simplicity of EDs, it was not necessary to use the sketch recognition techniques proposed so far in EulerSketch. Nevertheless, EulerSketch it is still an interesting prototype in that it shows a concrete example of the possibilities given by the interpretation and translation of sketches. In fact, EulerSketch allows to sketch Euler diagrams, and its main feature is the possibility of transforming a drawn ED into two types of text encodings of the ED topology called *static code* and *ordered Gauss paragraph* (OGP). In addition to classic editing operations, such as delete and move, the system allows the visualization of the static code and of the OGP of the drawn diagram and the corresponding encoding of its regions. Moreover, given the topology of an ED expressed through static code or OGP, it also allows to automatically generate a new topologically equivalent ED in its graphical representation. This functionality can be used both to display a new diagram from an edited code and to generate alternative and equivalents views of a sketched ED.

As mentioned at the beginning of this section, there are logical links between the various proposed methods. The most important is the one between RankFrag and the approach for the recognition of partially drawn symbols. As a matter of fact, RankFrag is the corner detection algorithm used for the segmentation of strokes as part of the preprocessing of the autocompletion system. As regards the approach for identifying attachment areas on sketched symbols, since it takes for granted the recognition of the symbol, it is therefore possible to integrate it with one of the proposed recognition methods.

1.3 Outline

In this thesis, each chapter is devoted to the contribution to a single area of the sketch recognition. Chapter 2 describes the related works; Chapter 3 describes RankFrag, the method for corner recognition; Chapter 4 describes

the approach for the recognition of multi-stroke hand drawn symbols[1]; Chapter 5 describes the approach for the recognition of partially drawn symbols and the evaluating the human performance in hand drawn symbol autocompletion[2]; Chapter 6 describes the approach for identifying attachment areas on sketched symbols[3]; Chapter 7 describes EulerSketch, the system for the sketching of Euler diagrams[4]; Chapter 8 presents some final remarks and a brief discussion on future works. Finally, Appendix A shows the data set used to test the proposed methods.

[1]The content of Chapter 4 is based on the following peer-reviewed paper: [10]

[2]The content of Chapter 5 is based on the following peer-reviewed papers: [11], [12], [13].

[3]The content of Chapter 6 is based on the following peer-reviewed paper: [14].

[4]The content of Chapter 7 is based on the following peer-reviewed papers: [15], [16].

Chapter 2

Related Work

This section describes the related work regarding the sketch recognition aspects treated in this thesis, devoting a section to each of them.

2.1 Corner detection

Corner detection is a fundamental component in creating sketch recognizers. Since corners represent the most noticeable discontinuity in the graphical strokes, their detection is often used in the *segmentation* (or *fragmentation*) of input strokes into primitives.

Important features for corner detection techniques include the high precision, the possibility to be performed in real time, and the capacity of adaptation to user preferences, to user drawing style and to the particular application domain. The adaptation to the domain can be achieved by using techniques based on machine learning. Almost all of the most recent methods use machine learning techniques, since they have also been shown to improve accuracy.

The methods for corner detection evaluate some features on the points of the stroke, after that these have possibly been resampled, e.g. at a uniform distance. Curvature and speed are the features that have been used first. In particular, the corners are identified by looking at maxima in the curvature function or at minima in the speed function. Lately, methods based on

machine learning have begun to consider a broader range of features.

One of the first methods proposed in the literature, [17], is based on the analysis of the curvature through three different measures. The authors also propose an advanced method for the determination of the "region of support" for local features, which is the neighborhood of the point on which the features are calculated. One of the first methods based on the simple detection of speed minima is [18]. Given the inaccuracy of curvature and speed taken individually, it was decided to evaluate them both in combination: [19] uses a hybrid fit by combining the set of candidate vertexes derived from curvature data with the candidate set from speed data.

A method introducing a feature different from curvature and speed is *ShortStraw* [20]. It uses the *straw* of a point, which is the segment connecting the endpoints of a window of points centered on the considered point. The method gave good results in detecting corners in polylines by selecting the points having a straw of length less than a certain threshold. Subsequently, the method has been extended by Xiong and LaViola [21] to work also on strokes containing curves.

One of the first methods to use machine learning for corner finding is the one described in [1]. It is used to segment the shapes in diagrams of chemistry. A very recent one is *ClassySeg* [22], which works with generic sets of strokes. The method firstly detects candidate segment windows containing curvature maxima and their neighboring points. Then, it uses a classifier trained on 17 different features computed for the points in each candidate window to decide if it contains a corner point.

There are approaches of stroke segmentation who do not find corners, but subdivide the stroke at specific points in order to produce desired primitives. They may however be instantiated to search for corners. A recent method, called DPFrag [23] learns primitive-level models from data, in order to adapt fragmentation to specific data sets and to user preferences and sketching style.

SpeedSeg [24] and *TCVD* [25] are able to find both the corners and the points where there is a significant change in curvature (referred to as "tangent vertices" in [25]). In order to detect corners, the former method mainly relies on pen speed while the latter uses a curvature measure.

2.2 Sketched symbol recognition

In general, symbol recognition is a classification process in which the unknown input symbol is compared to a set of templates in order to find the best matching class. Most approaches require a time-consuming training phase in order to correctly define the characteristics of each class of symbols. Furthermore, the invariance of the recognition with respect to scale, stroke number and order are desirable characteristics. The invariance with respect to rotation and not uniform scale could also be required when necessary.

Even though in most cases methods proposed for image recognition can be used [2, 26], several specialized methods for the recognition of *sketchy* images have been proposed. The earliest recognizers were only able to recognize unistroke symbols. In a pioneering work [27], a feature-based recognition approach is proposed. In this approach a stroke is characterized by 13 features including its length, size of the bounding box, average speed of the stylus, etc. A statistical pattern matching is used to compare the unknown stroke to those gathered in a training phase. Many unistroke symbol recognizers (e.g. text entry applications [28]) use elastic matching [29], a common pattern recognition-based approach to calculate a distance between two strokes. It basically works by evaluating the distances between corresponding points extracted from the two strokes. A recently proposed approach [30] has results comparable to those obtained through *elastic matching*, but enables accurate recognition with a few number of templates and can be easily implemented on any platform without requiring the inclusion of external libraries.

As for multi-stroke symbol recognition, several specialized methods have been recently proposed for multi-stroke hand drawn symbol recognition. According to a widely accepted taxonomy [31, 32, 26] the methods are classified into two main categories: *structural* and *statistical*.

In *structural* methods, the matching is performed by finding a correspondence between the structures, such as graphs [3, 33, 34] or trees [32], representing the input and the template symbols. The methods based on graph matching usually represent symbols through *Attributed Relational Graphs* (ARG). Such a representation gives a structural description of the

symbol [3]: the nodes in the graph are associated to the *primitives* composing the symbol, while the edges are associated to spatial relations between the *primitives*. The relations are often based on the presence of conditions such as intersections, parallelism, etc. Furthermore many approaches require the identification of the type of the *primitives* (line, arc, ellipse, etc.) composing the symbol. Due to the imprecise nature of *sketchy* symbols, the detection of the above characteristics is far from being precise and tolerance thresholds must be set, e.g., to distinguish a line from an arc or to check parallelism etc. In most cases, the user strokes are pre-processed in order to smooth them and to extract the sequence of *primitives* from them. Due to the high computational complexity, approximate algorithms for structural matching are often used, as the approximate graph matching algorithms presented in [34].

Statistical methods offer the advantage of avoiding the complex pre-processing phase in which the primitives are extracted. In most methods [35, 36, 37] a given number of features are extracted from the pixels of the unknown symbol and compared to those of the models. In particular, [35] uses nine features and also solves the problems related to the partitioning of the sketched elements (symbols, connectors, etc.), but requires the availability of at least five training examples per class. The best match is chosen through a statistical classifier. While techniques as *Zernike moment descriptors* [38] enable a very natural drawing style and support the invariance with respect to many types of transformations, tools as *Hidden Markov Models* (*HMM*) [39] can only be used when a fixed stroke order is established.

Other recognizers exploit common classifiers in image-based matching. In image-based techniques, the symbol is treated as a rasterized image. The advantage of such an approach is its independence on stroke order and number. E.g., in [40] the initial image is framed and down sampled into a 48 x 48 square grid. The recognizer exploits common classifiers in image-based matching, such as *Hausdorff* distance (and an ad-hoc defined variant of it), *Tanimoto* and *Yule* coefficients. The distances obtained by different classifiers are then combined together in order to obtain a unified measure. Following a 2-step strategy common to other approaches, the recognition first identifies a subset

of classes, and then recognizes an individual class out of that subset.

The *shape context* itself has already been brought in a sketch recognition system [41] to represent parts of a symbol. Here, instead, a variation of it is used to represent spatial relations between symbol primitives. The approach presented in [42], called *$P*, is an extension to the recognition of multi-stroke symbols of the *$1* approach proposed in [30]. It preserves the *minimalism* of its predecessor and relies on some of its unistroke recognition functionalities, even though it treats the symbols as point clouds.

Symbol recognition can be a functionality of general-purpose frameworks for sketch recognition [43, 44, 45]. *SketchREAD* [43] and *AgentSketch* [44] exploit the knowledge about the domain context for disambiguating the symbols recognized at a lower level. The former uses a structural description of the domain symbols, through the *LADDER* language [46], as a combination of lower level primitives meeting certain geometric constraints. *AgentSketch* [44] exploits an agent-based system for interpreting the sketched symbols. In [47] a stroke sequence of a symbol is firstly transformed into a string. The comparison between symbols is then performed by calculating the *Levensthein* distance between their corresponding strings. The characters of the string are obtained by coding the directions of successive sampled points. This approach is clearly dependent on stroke order. *CALI* [45] exploits a naive Bayesian classifier to recognize geometric shapes. A statistical analysis of various geometrical features of the shapes is performed, such as the convex hull, the largest triangle and largest quadrilateral that can be inscribed within the hull, the smallest area enclosing rectangle that can be fitted around the shape. In [48] a graph-based algorithm for recognizing multi-stroke primitives in complex diagrams is presented. The presented algorithm, based on *Paleosketch*, does not require any special drawing constraint to the user.

2.3 Autocompletion

Another thing related to the sketch recognition is the autocompletion.

Autocompletion is a functionality which involves the program in outputting the result desired by the user without the user actually entering the input data

completely. It is commonly used with textual input and studies in the context of text autocompletion have already been carried out. For instance, it has proven effective or appreciated by the users in various text-based applications, such as text entry on mobile devices [4], search engine interfaces (e.g. Google Instant [5]), source code editors [6], database query tools [7], etc.

To achieve autocompletion of graphical symbols, it is necessary that the recognizer is able to recognize partially drawn symbols. Only a few methods [49, 32, 50, 51] have been introduced supporting this feature.

Only a few methods have been introduced which are able to assist the user in the completion of multi-stroke hand drawn symbols. Some of them, such as *OctoPocus*, [52], *SimpleFlow* [53] and *GestureCommander* [54], only work for unistroke symbol completion: by exploiting different recognition and feedback techniques, they provide the user with a visual feedback on the recognition while the gesture is still being performed. Among those working for multi-stroke symbols, a grammar-based technique is presented in [51]. In adjacency grammars, as those used in [51], primitive types are the terminal symbols and the productions describe the topology of the symbols. Furthermore, a set of adjacency constraints (e.g. incident, adjacent, intersects, parallel, perpendicular) define the relation between two primitives.

In particular, once a partial input has been processed, the parser is able to propose to the user a set of final acceptance states (valid symbols) that have as subshapes the current intermediate state. In [50] a *Spatial Relation Graph* (SRG) and its partial matching method are proposed for online composite graphics representation and recognition. The SRG structure is a variation of ARG in which an edge connects two nodes only if a spatial relation (interconnection, tangency, intersection, parallelism and concentricity) is present between the primitives associated to the nodes. A *Spatial Division Tree* (SDT) has been used in [32]. In this representation, a node in the tree contains a set of strokes. Furthermore, intersection relations among strokes are codified through links among nodes. The approach described in [49] is based on clustering. In order to assign a (possibly partial) symbol to a cluster, the set of features described in [36] is extracted from it. The features are extracted on the partially drawn symbols used in the training data. Hence, the

approach relies on the observation that people do tend to prefer certain stroke drawing orderings over others. Other, domain-specific, systems supporting symbol autocompletion have been described in literature. For instance, [55] describes a system for the recognition of a set of 485 symbols from Course of Action Diagrams [56], which also supports autocompletion.

The first three of the above cited methods show poor performance with partially drawn symbols when only a few primitives are available. As a consequence, they might not lend themselves well for the realization of an interactive system for autocompletion. The one described in [49], instead, is not completely invariant with respect to stroke order, but relies on users' preferred order. Furthermore, the authors of the above researches do not provide evidence that symbol autocompletion can lead to a real advantage in terms of drawing time saving.

2.4 Attachment areas

Another thing related to the sketch recognition are the attachment areas. Attachment areas are the areas on which relations between symbols (of a visual language) can be defined. They are related to the physical appearance of the symbol an and can have different shapes, like a single point or parts of a symbol or an area defined by the symbol itself.

In sketch recognition research, only a few works [8] have raised the problem of correctly identifying the attachment areas. This is probably due to the lack of well established solutions to the related problems of ink segmentation and object recognition, on which most of the effort of researchers is focused. Different techniques, in fact, can be used to recognize symbols in sketched diagrams. Some of them are stroke-based, online and use time data [55], some others rely on image-based techniques [40, 36]. Other researches also try to solve the segmentation problem related to the separation of symbols from other elements of the diagrams, such as connectors. E.g., in [35] the areas of high ink density are likely to be recognized as symbols instead of connectors. The management of attachment areas is often defined ad hoc for the considered domain.

2.5 Euler diagram sketching

Methods and tools have been developed for the recognition of specialized diagrams, e.g. in engineering, chemistry, medicine, music and so on [57, 1, 58, 59]. Examples of use of sketch recognition include graphical environments for hand-drawing Euler diagrams (EDs). Previous works on ED sketch recognition [60, 61, 62] utilize single stroke recognition and machine learning techniques. In particular, in [60], the authors present a sketch tool for drawing EDs through ellipses, with a recognition mechanism able to extract the semantic of a sketched ED and to convert it into a formal diagram drawn with circles and ellipse. This work is extended in [61] by including the support to arbitrary closed curves, input and editing in the formal view, production of sketches from formal diagrams, and semantic matching via the computation of abstract representations. In [62], the authors presents a sketch tool for the recognition of EDs augmented with graphs and shading.

Chapter 3

RankFrag: a Novel Technique for Corner Detection in Hand Drawn Sketches

Sketched diagrams recognition raises a number of issues and challenges, including both low-level stroke processing and high-level diagram interpretation [63]. A low-level problem is the *segmentation* (also known as *fragmentation*) of input strokes. Its objective is the recognition of the graphical primitives (such as lines and arcs) composing the strokes. Stroke segmentation can be used for a variety of objectives, including symbol recognition in structural methods [34, 11].

Most approaches for segmentation use algorithms for finding corners, since these points represent the most noticeable discontinuity in the graphical strokes. Some other approaches [25] also find the so called *tangent vertices* (smooth points separating a straight line from a curve or parting two curves).

A high accuracy and the possibility of being performed in real time are crucial features for segmentation techniques. Tumen and Sezgin [23] also emphasize the importance of the adaptation to user preferences and drawing style and to the particular domain of application. Adaptation can be achieved by using machine learning-based techniques. Machine learning has also proven to improve accuracy. In fact, almost all of the most recent segmentation

methods use some machine learning-based technique.

The technique presented here, called *RankFrag*, uses machine learning to decide if a candidate point is a corner. This technique is strongly inspired to previous work. In particular, the work that mostly influenced this research is that of Ouyang and Davis [1], which introduced a *cost* function expressing the likelihood that a candidate point is a corner. A distinguishing feature of the presented technique is the so called *rank* of a candidate point. Points with a higher rank (a lower integer value) are more likely to be corners. The rank is a progressively decreased integer value, assigned to the points as they are iteratively removed from a list of candidate corners. At each iteration, the point minimizing Ouyang and Davis's cost function is removed from the list. Another important characteristic of RankFrag is the use of a variable "region of support" for the calculation of some local features, which is the neighborhood of the point on which the features are calculated. Most of the features used for classification are taken from several previous works in the literature [64, 65, 20, 66, 1].

RankFrag has been tested on three different data sets previously introduced and already used in the literature to evaluate existing techniques. The performance of RankFrag has been compared to other state-of-art techniques [21, 23] and significantly better results are achieved on all of the data sets.

The chapter is organized as follows: Section 3.1 describes RankFrag; Section 3.2 presents the evaluation of its performance in comparison to those of existing techniques, while the results are reported in Section 3.3; lastly, some final remarks conclude the chapter.

3.1 The RankFrag technique

As a preliminary step, the stroke is processed by resampling its points to obtain an equally spaced ordered sequence of points $P = (p_1, p_2, \ldots, p_n)$, where n varies depending on a fixed space interval and on the length of the stroke. To extract equally spaced points the procedure described in [30] is used. Furthermore, a *Gaussian smoothing* [67] is executed on the extracted points in order to reduce the resampled stroke noise.

In order to identify the corners, the following three steps are then executed:

1. Initialization;
2. Pruning;
3. Point classification.

The initialization step creates a set D containing n pairs (i, c), for $i = 1 \ldots n$ where c is the *cost* of p_i and is calculated through Equation 3.1 derived from the *cost* function defined in [1].

$$Icost(p_i) = \begin{cases} \sqrt{mse(S; p_{i-1}, p_{i+1})} \times dist(p_i; p_{i-1}, p_{i+1}) & \text{if } i \in \{2, \ldots, n-1\} \\ +\infty & \text{if } i = 1 \text{ or } i = n \end{cases} \tag{3.1}$$

In the above equation, $S = \{p_{i-1}, p_i, p_{i+1}\}$ and $mse(S; p_{i-1}, p_{i+1})$ is the mean squared error between the set S and the line segment formed by (p_{i-1}, p_{i+1}). The term $dist(p_i; p_{i-1}, p_{i+1})$ is the minimum distance between p_i and the line segment formed by (p_{i-1}, p_{i+1}). Since p_1 and p_n do not have a preceding and successive point, respectively, they are treated as special cases and given the highest cost.

The pruning step iteratively removes $n - np$ elements from D in order to make the technique more efficient. The value np is the number of candidate corners not pruned in this step and depends on the data sets. It is chosen so that no corner is eliminated in the pruning step. At each iteration, the element m with the lowest cost is removed and the costs of the closest preceding points p_{pre} in P and the closest successive point p_{suc} in P of p_m, with *pre* and *suc* occurring in D, are updated through Equation 3.2.

$$Cost(p_i) = \begin{cases} \sqrt{mse(S; p_{ipre}, p_{isuc})} \times dist(p_i; p_{ipre}, p_{isuc}) & \text{if } i \in \{2, \ldots, n-1\} \\ +\infty & \text{if } i = 1 \text{ or } i = n \end{cases} \tag{3.2}$$

The points p_{ipre} and p_{isuc} are, respectively, the closest preceding and successive points of p_i in P, with *ipre* and *isuc* occurring in D. S is the subset of points

between p_{ipre} and p_{isuc} in the resampled stroke P. The functions *mse* and *dist* are defined as for Equation 3.1.

The point classification step returns the list of points recognized as corners by further removing from D all the indices of the points that are not recognized as corners. This is achieved by the following steps:

1. find the current element in D with minimum cost (if D contains only 1 and n, return an empty list);

2. calculate the features of the point corresponding to the current element and determine if it is a corner by using a binary classifier, previously trained with data.

   - if it is not a corner, delete it from D, make the necessary updates and go to 1.
   - if it is a corner, proceed to consider as current the next element in D in ascending cost order, if such a point is the first or the last point return the list of points corresponding to the remaining elements in D (except for 1 and $|P|$), otherwise go to 2.

In Fig. 3.1, the function DetectCorners() shows the pseudocode for the initialization, pruning and point classification steps. In the pseudocode, D is the above described set with the following functions:

- Init(L) initialize D with all the (i, c) pairs contained in L;
- FindMinC() returns the element of D with the lowest cost;
- PreviousI(i) returns j such that (j, c') is the closest preceding element of (i, c) in D, i.e., $j = \max\{k \mid (k, c) \in D \text{ and } k < i\}$;
- SuccessiveI(i) returns j such that (j, c') is the closest successive element of (i, c) in D, i.e., $j = \min\{k \mid (k, c) \in D \text{ and } k > i\}$;
- SuccessiveC(i) returns the successive element of (i, c) in D with respect to the ascending cost order;
- Remove(i) removes (i, c) from D;

Input: an array P of equally spaced points that approximate a stroke, a number *np* of not-to-be-pruned points, and the CLASSIFIER() function.
Output: a list of detected corners.

```
 1: function DETECTCORNERS(P, np, CLASSIFIER)
 2:     # initialization
 3:     for i = 1 to |P| do
 4:         c ← ICOST(i, P)                          # computes Equation 3.1
 5:         add (i, c) to TempList
 6:     end for
 7:     D.INIT(TempList)

 8:     # pruning
 9:     while |D| > np do
10:         (i_min, c) ← D.FINDMINC()
11:         REMOVEANDUPDATE(i_min, P, D)
12:     end while

13:     # point classification
14:     while |D| > 2 do
15:         (i_cur, c) ← D.FINDMINC()
16:         loop
17:             isCorner ← CLASSIFIER(i_cur, P, D)
18:             if isCorner then
19:                 (i_cur, c) ← D.SUCCESSIVEC(i_cur)
20:                 if i_cur ∈ {1, |P|} then
21:                     for each i ∉ {1, |P|} in D add P[i] to CornerList
22:                     return CornerList
23:                 end if
24:             else
25:                 REMOVEANDUPDATE(i_cur, P, D)
26:                 break loop
27:             end if
28:         end loop
29:     end while
30:     return ∅
31: end function

32: procedure REMOVEANDUPDATE(i, P, D)
33:     i_pre ← D.PREVIOUSI(i)
34:     i_suc ← D.SUCCESSIVEI(i)
35:     D.REMOVE(i)
36:     c ← COST(i_pre, P, D)                        # computes Equation 3.2
37:     D.UPDATECOST(i_pre, c)
38:     c ← COST(i_suc, P, D)
39:     D.UPDATECOST(i_suc, c)
40: end procedure
```

Figure 3.1: The implementation of the initialization, pruning and corner classification steps.

- UpdateCost(i, c) updates the cost of i in D setting it to c.

DetectCorners() calls a Classifier(i, P, D) function that computes the features (described in Section 3.1.2) of the point $P[i]$, and then uses them to determine if $P[i]$ is a corner by using a binary classifier previously trained with data (described in Section 3.1.3).

3.1.1 Complexity

The complexity of the function DetectCorners() in the previous section depends on the implementation of the data structure D. The following calculation will be based on the implementation of D as an array in which the ith element refers to the node that contains the pair (i, c) (or *nil* if the node does not exist) and a pointer that refers to the node with the minimum c. Each node has 3 pointers: one that points to the successive node in ascending c order, one that points to the successive node in ascending i order and one that points to the previous node in ascending i order. With this implementation, the FindMinC(), PreviousI(), SuccessiveI(), SuccessiveC() and Remove() functions are all executed in constant time, while UpdateCost() function is $O(|D|)$ (where $|D|$ is the number of nodes referred in D) and the Init(L) function is $O(|L| \log |L|)$ (by using and efficient sorting algorithm). In the following, it will be shown that the DetectCorners() complexity is $O(n^2)$, where $n = |P|$.

It is trivial to see that: the complexity of the ICost() function is $O(1)$; the complexity of Cost() is $O(n)$ in the worst case and, consequently, the complexity of RemoveAndUpdate() is $O(n)$; and the complexity of Classifier() is $O(n)$ since some features need $O(n)$ time in the worst case to be calculated.

The complexity of each of the three steps is then:

1. Initialization: ICost() is called n times and D.Init() one time, consequently the complexity of the initialization step is $O(n \log n)$.

2. Pruning: D.FindMinC() and RemoveAndUpdate() are called $n - np$ times each, consequently the complexity of this step is $O(n(n - np))$.

3. Point classification: the *while* loop (in line 14) will be executed at most $k = |D| - 2 \leq np - 2$ times. In the loop (in line 16), Classifier() will be called at most k times, D.SuccessiveC() at most $k - 1$ times, and RemoveAndUpdate() at most once. Thus, in this step, they will be called less or equal than k^2, k^2 and k times, respectively.

   The complexity of the Classifier() calls can be calculated by considering that for each point, if none of its features changes, the result of Classifier() can be retrieved in $O(1)$ by caching its previous output. Since the execution of the RemoveAndUpdate() function involves the changing of the features of two points, Classifier() will be executed at most $3k$ times in $O(n)$ (for a total of $O(k \times n)$) and the remaining times in $O(1)$ (for a total of $O(k^2)$), giving a complexity of $O(k \times n)$.

   Furthermore, the complexity of the D.SuccessiveC() calls is $O(k^2)$, while the complexity of the RemoveAndUpdate() calls is $O(k \times n)$.

   Thus, since $k < n$, the point classification step is in the worst case $O(k \times n)$, or rather $O(n \times np)$.

It is worth noting that the final $O(n^2)$ complexity does not improve even if a better implementation of D providing an $O(\log |D|)$ UpdateCost() function is used.

3.1.2 Features

The main distinguishing feature used by the presented technique is the *rank*. The *rank* of a point $p = P[i]$, with respect to D, is defined as the size of D resulting from the removal of (i, c) from D. The other features are derived from previous research in the field. There are three different classes of features:

- *Stroke features*: features calculated on the whole stroke;
- *Point features*: local features calculated on the point. These features are calculated using a fixed region of support and their value remain stable throughout the procedure;

- *Rank-related features*: dynamically calculated local features. The region of support for the calculation of these features is the set of points from the predecessor p_{pre} and the successor p_{suc} of the current point in the candidate list. Their value can vary during the execution of the *Point classification* step.

Stroke Features

The features calculated on the whole stroke can be useful to the classifier, since a characteristic of the stroke can interact in some way with a local feature. For instance, the length of a stroke may be correlated to the number of corners in it: it is likely that a long stroke has more angles than a short stroke. Two stroke features are derived from [1]: the length of the stroke and the diagonal length of its bounding box. These features are called *Length* and *Diagonal*, respectively. Furthermore, a feature telling how much the stroke resembles an ellipse (or a circle), called *EllipseFit*, was added. It is calculated by measuring the average euclidean distance of the points of the stroke to an ideal ellipse, normalized by the length of the stroke.

Point Features

The *point features* are local characteristics of the points. The speed of the pointer and the curvature of the stroke at a point have been regarded as very important features from the earliest research in corner finding. Here, the speed at p_i is calculated as suggested in [64], i.e., $s(p_i) = \|p_{i+1}, p_{i-1}\| / t_{i+1} - t_{i-1}$, where t_i represents the timestamp of the i-th point. It is also present a version of the speed feature where a min-max normalization is applied in order to have as a result a real value between 0 and 1; the *Curvature* feature used here is calculated as suggested in [65].

A feature that has proven useful in previous research is the *straw*, proposed in [20]. The straw at the point p_i is the length of the segment connecting the endpoints of a window of points centered on p_i. Thus $Straw(p_i, w) = \|p_{i+w}, p_{i-w}\|$, where w is the parameter defining the width of the window.

A simple feature to evaluate if a point is a corner, is the magnitude of

the angle formed by the segments (p_{i-w}, p_i) and $(p_i, p_{i+w}($, defined here as $Angle(p_i, w)$. A useful feature to distinguish the curves from the corners is called *AlphaBeta*, derived from [21]. *alpha* and *beta* are the magnitudes of two angles in p_i using different segment lengths, one three times the other. Here the difference between them is used as a feature: $AlphaBeta(p_i, w) = Angle(p_i, 3w) - Angle(p_i, w)$.

Lastly, in this research two point features are introduced, that, to the best of my knowledge, have never been tested so far for corner detection. One feature is the position of the point within the stroke, indicated as the ratio between the length of the stroke from p_0 to p_i and the total length of the stroke. This feature is called $Position(p_i)$. The other feature is the difference of two areas: the former is the one of the polygon delimited by the points $(p_{i-w}, \ldots, p_i, \ldots, p_{i+w}($ and the latter is the one of the triangle (p_{i-w}, p_i, p_{i+w}). The rationale for this feature is that its value will be positive for a curve, approximately 0 for an angle and even negative for a cusp. It is called $DeltaAreas(p_i, w)$.

Rank-Related Features

The *rank-related features* are local characteristics of the points. The difference with the point features is that their region of support varies according to the rank of the point: the considered neighborhood is between the closest preceding and successive points of p_i occurring in D, which are called p_{ipre} and p_{isuc}, respectively. The *Rank* and the *Cost* function defined in Equation (3.2) are examples of features from this class.

A simple feature derived from [1] is *MinDistance*, representing the minimum of the two distances $\|p_{ipre}, p_i\|$ and $\|p_i, p_{isuc}\|$, respectively. A normalized version of *MinDistance* is obtained by dividing the minimum by $\|p_{ipre}, p_{isuc}\|$.

As in previous research, parts of the stroke are tried to be fitted with beautified geometric primitives. The following two features are inspired by the ones defined in [66]: $PolyFit(p_i)$ tries to fit the substroke $(p_{ipre}, \ldots, p_i, \ldots, p_{isuc})$ with the polyline $(p_{ipre}, p_i, p_{isucc})$, while $CurveFit(p_i)$ uses a bezier curve to approximate the points. The return value is the average point-to-point euclidean

Feature	Class	Parameters	Ref.
$Length(S)$	Stroke	/	[1]
$Diagonal(S)$	Stroke	/	[1]
$EllipseFit(S)$	Stroke	/	
$Speed(p, norm)$	Point	$norm = T, F$	[64]
$Curvature(p)$	Point	/	[65]
$Straw(p, w)$	Point	$w = 4$	[20]
$Angle(p, w)$	Point	$w = 1, 2$	[21]
$AlphaBeta(p, w)$	Point	$w = 3, 4, 6, 15$	[21]
$Position(p)$	Point	/	
$DeltaAreas(p, w)$	Point	$w = 11$	
$Rank(p)$	Rank-Related	/	
$Cost(p)$	Rank-Related	/	[1]
$MinDistance(p, norm)$	Rank-Related	$norm = T, F$	[1]
$PolyFit(p)$	Rank-Related	/	[66]
$CurveFit(p)$	Rank-Related	/	[66]

Table 3.1: The features used in RankFrag.

distance normalized by the length of the stroke.

Table 3.1 summarizes the set of features used by RankFrag in the CLASSIFIER function. The table reports the name of the feature, its class, the values of the parameters (if present) with which it is instantiated and the reference paper from which it is derived. The presence of feature parameters means that some feature could potentially be used several times, instantiated with a different parameter value, and this might introduce redundant features. The parameters have been chosen by performing an internal validation process that measures the relevance of the parameter-dependent features, over possible parameter values, and uses feature clustering to define a subset of relevant, non-redundant features.

3.1.3 Classification method

The binary classifier used by RankFrag in the CLASSIFIER function to classify corner points is based on *Random Forests* (RF) [68]. Random Forests are an ensemble machine learning technique that builds forests of classification trees. Each tree is grown on a bootstrap sample of the data, and the feature

at each tree node is selected from a random subset of all features. The final classification is determined by using a voting system that aggregates the classification results from all the trees in the forest. There are many advantages of RF that make their use an ideal approach for this classification problem: they run efficiently on large data sets; they can handle many different input features without feature deletion; they are quite robust to overfitting and have a good predictive performance even when most predictive features are noisy.

3.1.4 Implementation

RankFrag was implemented as a Java application. The classifier was implemented in *R* language, using the *randomForest* package [69]. The call to the classifier from the main program is performed through the *Java/R Interface* (JRI), which enables the execution of *R* commands inside Java applications.

3.2 Evaluation

RankFrag was evaluated on three different data sets already used in the literature to evaluate previous techniques. A 5-fold cross validation was repeated 30 times on all of the data sets. For all data sets, the strokes were resampled at a distance of three pixels, while a value of $np = 30$ was used as a parameter for pruning. Since there is no single metric that determines the quality of a corner finder, the performance of RankFrag was calculated using the various metrics already described in the literature. The results for some metrics were averaged in the cross validation and were summed for others.

The hosting system used for the evaluation was a laptop equipped with an *Intel™Core™i7-2630QM* CPU at 2.0 GHz running *Ubuntu 12.10* operating system and the *OpenJDK 7*.

3.2.1 Model validation

This section describes the process of assessing the prediction ability of the RF-based classifiers. The accuracy metrics were calculated by repeating 30

times the following procedure individually for each data set and taking the averages:

1. the data set DS is divided randomly into 5 parts with an equal number of strokes (or nearly so, if the number of strokes is not divisible by 5);

2. for $i = 1 \ldots 5$: $DSt_i = \cup(DS_j)\ \{j \neq i\}$ is used as a training set, and DS_i is used as a test set.

   - RankFrag is executed on DSt_i in order to produce the training data table. In DS, the correct corners had been previously marked manually. For each point extracted from the candidate list the input feature vector is calculated, while the output parameter is given by the boolean value indicating whether the point is marked or not as a corner. The training table contains both the input and output parameters;

   - A random forest is trained using the table;

   - RankFrag is executed on DS_i, using the trained random forest as a binary classifier;

   - In order to generate the accuracy metrics, the corners found by the last run of RankFrag are compared with the manually marked ones. A corner found by RankFrag is considered to be correct if it is within a certain distance from a marked corner (only one corner found by RankFrag can be considered to be correct for each marked corner).

3. In order to get aggregate accuracy metrics, for each of them the average/sum (depending on the type of the metric) of the values obtained in the previous step is calculated.

3.2.2 Accuracy metrics

A corner finding technique is mainly evaluated from the points of view of accuracy and efficiency. There are different metrics to evaluate the accuracy of a corner finding technique. The following metrics, already described in the literature [20, 22], are used:

- **False positives** and **false negatives**. The number of points incorrectly classified as corners and the number of corner points not found, respectively;
- **Precision**. The number of correct corners found divided by the sum of the number of correct corners and false positives:
$$precision = \frac{correct\ \ corners}{correct\ \ corners + false\ \ positives};$$
- **Recall**. The number of correct corners found divided by the sum of the number of correct corners and false negatives:
$$recall = \frac{correct\ \ corners}{correct\ \ corners + false\ \ negatives}.$$
This value is also called **Correct corners accuracy**;
- **All-or-nothing accuracy**. The number of correctly segmented strokes divided by the total number of strokes;

The task of judging whether a corner is correctly found is done by a human operator. The presence of the angle is then determined by human perception. Obviously, different operators can perform different annotations on a data set. In this work, data sets already annotated by other authors are used. It is worth noting that a tolerance of 7 sampled points (corresponding to a maximum distance of 21 pixels) from the marked corner was used. This value is similar to others used in the literature (e.g., in [22] a fixed distance of 20 pixels was used).

3.2.3 Data sets

Two of the three data sets used in this evaluation, the *Sezgin-Tumen COAD Database* and *NicIcon* data sets, are associated to a specific domain, while

Data set	No. of classes	No. of symbols	No. of strokes	No. of drawers	Source
*COAD**	20	400	1507	8	[49]
NicIcon	14	400	1204	32	[70]
IStraw	10	400	400	10	[21]

Table 3.2: Features of the three data sets.

the *IStraw* data set is not associated to any domain, but it was produced for benchmarking purposes by Xiong and LaViola [21]. Some features of the three data sets are summarized in Table 3.2. The table reports, for each of them, the number of different classes, the total number of symbols and strokes, the number of drawers and a reference to the source document introducing it. The *Sezgin-Tumen COAD Database* (called only COAD*, for brevity, in the sequel) is composed by 400 symbols (1507 strokes with their identified corners) extracted from the COAD data set described in Appendix A.4. The *NicIcon* and *IStraw* data sets are described in the Appendix A.2 and Appendix A.1, respectively.

3.3 Results

This section reports the results of the RankFrag evaluation. As for the accuracy, all of the metrics described in the previous section were calculated. Furthermore, RankFrag's accuracy is compared to that of other state-of-art methods by using the All-or-nothing metric. It is worth noting that, due to the unavailability of working prototypes, the other methods are not directly tested: only the performance declared by their respective authors are reported.

The accuracy achieved by RankFrag on the three data sets is reported in Table 3.3. The results are averaged over the 30 performed trials.

Table 3.4 shows a comparison of the accuracy of RankFrag with other state-of-art methods. The methods considered here are DPFrag [23] and IStraw [21]. Due to the unavailability of other data, only the results related to the All-or-nothing metric are reported.

As it can be seen, RankFrag outperforms the other two methods. The

Metrics	COAD*	NicIcon	IStraw
Corners manually marked	2271	867	1796
Corners found	2264.57	836.21	1795.00
Correct corners	2261.21	825.64	1790.57
False positives	3.36	10.57	4.43
False negatives	9.79	41.36	5.43
Precision	0.9985	0.9873	0.9976
Recall / Correct corners accuracy	0.9957	0.9525	0.9970
All-or-nothing accuracy	0.9927	0.9599	0.9754

Table 3.3: Average accuracy results of RankFrag on the three data sets.

Data set	RankFrag	DPFrag	IStraw
*COAD**	0.99	0.97	0.82
NicIcon	0.96	0.84	0.24
IStraw	0.98	0.96	0.96

Table 3.4: Comparison of RankFrag with other methods on the All-or-nothing accuracy metric.

largest improvement is obtained on the NicIcon data set, where the other two methods perform rather poorly. Less noticeable improvements are obtained on the COAD and on the IStraw data sets, where the other two methods do not perform badly.

As for efficiency, the average time needed to detect the corners in a stroke is ~390 ms. This implementation is rather slow, due to the inefficiency of the calls to R functions. A non-JRI implementation was also produced by manually exporting the created random forest from R to Java (avoiding the JRI calls). With this implementation, the average execution time is lowered to ~130 ms, thus enabling real-time user interactions.

3.4 Concluding remarks

This chapter has introduced RankFrag, a technique for detecting corner points in hand drawn sketches. The technique outperforms two state-of-art methods on all the tested data sets. In particular, RankFrag is the only technique obtaining a satisfactory result on the "difficult" NicIcon data set, correctly

processing the 96% of the strokes.

RankFrag finds only corner points and not *tangent vertices*, as done by other techniques [24, 25]. It can be directly used in various structural methods for symbol recognition, as shown in Chapter 5. However in some methods an additional step to classify the segments in lines or arcs may be required.

Chapter 4

Improving Shape Context Matching for the Recognition of Sketched Symbols

In this chapter, an approach to recognize multi-stroke hand drawn symbols is presented. Since the approach is an adaptation of the image-based matching algorithm proposed by Belongie et al. [2], it is invariant with respect to scaling and is independent from the number and order of the drawn strokes. Furthermore, it has a better recognition accuracy than the original one when applied to hand drawn symbols. This is due to the exploitation of information on stroke points, such as temporal sequence and occurrence in a given stroke.

Briefly, the algorithm proposed by Belongie et al. calculates the matching cost between two shapes as the minimum weighted bipartite graph matching between two equally sized sets of sampled points from both shapes. This is done by calculating a matrix of matching costs between each couple of points of the two symbols and selecting the resulting best match. The cost of matching of two points is calculated by evaluating the difference between their *shape contexts*, which are suitable shape descriptors introduced by the authors.

The approach improvement lies in re-calculating the cost matrix and, as a consequence, the total cost of matching between the two symbols. The cost

matrix is re-calculated by considering the symbols as sequences of sampled points and detecting the longest subsequences of points in each sketched symbol stroke whose mapping on the template symbol produces still subsequences in any template symbol stroke. Once the subsequences are detected their lengths are used to decrease the matching costs of the involved points proportionally. This provides a further check on the structural similarity of the symbols that the original algorithm does not take into consideration that proves to enhance the accuracy when applying the shape context descriptors to sketch recognition.

The approach has been evaluated on a set of more than 100 symbols extracted from the *Military Course of Action Diagrams* (COA) domain [56]. In the experiment the performance of the previous algorithm hare compared to that of the enhanced version. A *top 1* recognition rate of 95.7% on symbols sampled at 128 points is obtained by the enhanced version. This rate is for an improvement over the previous algorithm of 3.7%. The recognition grows up to 99% when considering the top 3 interpretations. The proposed improvement does not introduce signficant delays in the running time of the recognition procedure.

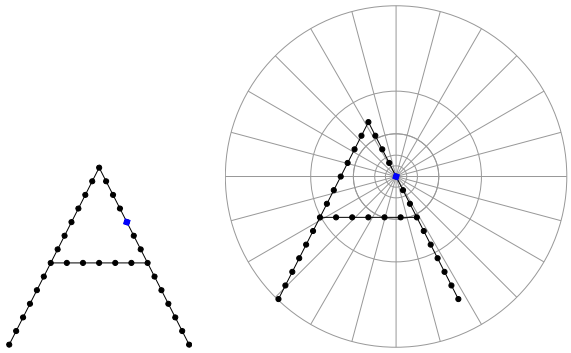
The chapter is organized as follows: the next section briefly describes the approach presented in [2]; section 4.2 describes the approach. The evaluation is presented in section 4.3; finally, section 4.4 offers final remarks.

4.1 Background: symbol recognition through shape context

This section describes the symbol recognition procedure introduced by Belongie et al. [2] as used in the proposed approach.

4.1.1 Feature descriptor

The shape context descriptor is first proposed by Belongie et. al [2] and is a global feature descriptor that has been successfully used in various application fields [2, 71, 72, 73]. It is a point based descriptor, thus a significant set of



(a) Sampled points (b) Shape context of a point

Figure 4.1: The shape context descriptor

points must be sampled from the original figure. The shape context of a point is built by dividing a point's surrounding area into bins with different angles and increasing distances. The shape context of a point P_i is defined by computing the number of other points that are located in each of its surrounding bins,

$$h_i(k,l) = \#\{x \neq P_i : (x - P_i) \in bin(k,l)\} \tag{4.1}$$

where k and l are the spatial coordinates of the bin and # refers to set cardinality. As an example, in Figure 4.1, the bin containing the lowest point of the right side of the symbol A contains $bin(5,20) = 4$ points. In other words, a point's shape context is a log-polar histogram which defines the relative distribution of other points. Different points in one shape have different shape context and similar points in two similar shapes have similar shape context. Belongie et al. [2] adopts the Chi-square distance to measure the difference between the shape context features of two points P_i and Q_j.

$$C_{ij} = \frac{1}{2}\sum_{k=1}^{M}\sum_{l=1}^{N}\frac{[h_i(k,l) + h_j(k,l)]^2}{h_i(k,l) + h_j(k,l)} \tag{4.2}$$

h_i and h_j are the shape context histograms of P_i and Q_j.

4.1.2 Matching

In order to compare a sketched symbol s against a template symbol t equation (4.2) must be calculated for each pair of sampled points P_i of s and Q_j of t. This will then produce a $n \times n$ matrix C where n is the number of points sampled from each symbol. Based on this cost matrix, the algorithm constructs a permutation π which allows to map each point P_i of s to a point $Q_{\pi(i)}$ of t such that the total matching cost H is minimized:

$$H(\pi) = \sum_{i}^{n} C_{i\pi(i)} \tag{4.3}$$

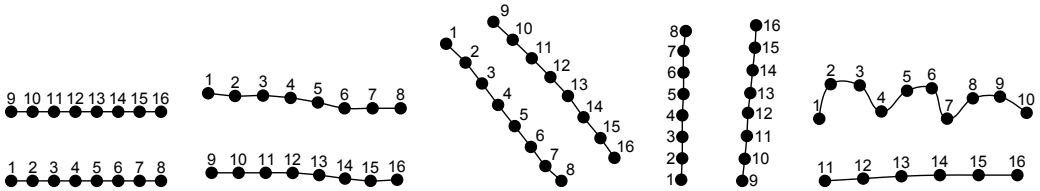
In the next section, the approach is described. It will make use of a function Π to map a sequence of points $seq = < P_1, \ldots, P_m >$ to the sequence of points $\Pi(seq) =< Q_{\pi(1)}, \ldots, Q_{\pi(m)} >$, with $m \geq 1$.

4.2 The approach

Sketched and template symbols consist of strokes starting with a pen-down and ending with a pen-up events. All the symbols are pre-processed to produce a set of a fixed number n of sampled points. These points are distributed among the strokes proportionally to their length. The $k - th$ stroke of a symbol is then represented as a sequence of sampled points $P_1, P_2, \ldots, P_{m_k}$ with $\sum_k m_k = n$.

Given a sketched symbol s and a set of template symbols $\{t_1, \ldots, t_l\}$, in order to find a template symbol that best matches s, first the matrix C and the mapping π hare computed, as shown in section 4.1 on each pair (s, t_j). Then, based on C, an updated version $C'_{i\pi(i)}$ is computed for $i = 1 \ldots n$ to calculate the new matching cost H' associated to t_j and s. Finally, the template symbol with the minimum cost will be presented as the best match.

The new contribution is the method to calculate the new $C'_{i\pi(i)}$. Basically, given a sketched symbol s and a template t, the method consider, for each stroke in s, the longest subsequences that are mapped by Π to sequences (also inverted) in any stroke of t. This provides a further check on the



(a) Template (b) D.1 (c) D.2 (d) D.3 (e) D.4

Figure 4.2: A template and four hand drawn sketched symbols

structural similarity of the two symbols that the shape context descriptor does not take into consideration. To reflect this property in the final cost computation, the matching cost of points participating in longer subsequences hare proportionally decreased. More formally,

$$C'_{i\pi(i)} = C_{i\pi(i)} * (1 - 2 \times \log_{2n} |lsub(i)|) \tag{4.4}$$

where $|lsub(i)|$ denotes the size of the longest subsequence in a sketched symbol stroke containing the point P_i such that $\Pi(lsub(i))$ or $inverse(\Pi(lsub(i)))$ is still a sequence in any template symbol stroke (containing the point $Q_{\pi(i)}$).

4.2.1 An example

Figure 4.2 shows a template symbol (a) and four sketched symbols (b, c, d, e) sampled at 16 points. Each table in Figure 4.3 shows the mapping π between one of the four sketched symbols and the template symbol. Each column shows the mapping between sketched and template symbol points, the subsequence to which the sketched point belongs and the cost of the mapping for both C and C' with the exception of the last column that shows the total cost of the mapping.

It can be noted that the C total cost for D.1 (92.05) is worse than the corresponding one for D.4 (78.08) despite the fact that D.1 is obviously the most similar sketched symbol for the template. On the other hand the C' total cost for D.1 (-18.41) is by far the lowest one.

$(i,\pi(i))$	(9,1)	(10,2)	(11,3)	(12,4)	(13,5)	(14,6)	(15,7)	(16,8)	(1,9)	(2,10)	(3,11)	(4,12)	(5,13)	(6,14)	(7,15)	(8,16)	$\sum$
$lsub$	s1								s2								
C	3.27	3.90	4.07	7.00	7.00	8.67	5.73	5.67	6.00	5.40	6.67	7.00	7.00	5.74	5.00	3.93	**92.05**
C'	-0.65	-0.78	-0.81	-1.40	-1.40	-1.73	-1.15	-1.13	-1.20	-1.08	-1.33	-1.40	-1.40	-1.15	-1.00	-0.79	**-18.41**

(a) Mapping between the D.1 and the template symbol

$(i,\pi(i))$	(4,1)	(3,2)	(5,3)	(6,4)	(7,5)	(8,6)	(16,7)	(15,8)	(2,9)	(1,10)	(10,11)	(9,12)	(11,13)	(12,14)	(14,15)	(13,16)	$\sum$
$lsub$	s1		s2				s3		s4		s5		s6		s7		
C	10.95	12.00	6.50	6.93	7.27	7.33	6.70	5.84	5.84	6.70	7.00	7.90	6.93	7.00	12.00	10.95	**127.85**
C'	6.57	7.20	1.30	1.39	1.45	1.47	4.02	3.50	3.50	4.02	4.20	4.74	4.16	4.20	7.20	6.57	**65.49**

(b) Mapping between the D.2 and the template symbol

$(i,\pi(i))$	(1,1)	(3,2)	(4,3)	(2,4)	(9,5)	(10,6)	(11,7)	(12,8)	(7,9)	(5,10)	(6,11)	(8,12)	(16,13)	(14,14)	(15,15)	(13,16)	$\sum$
$lsub$	s1	s2		s3	s4				s5	s6		s7	s8	s9		s10	
C	8.33	6.73	7.84	8.97	8.60	7.07	6.60	8.00	6.67	7.17	6.34	7.40	8.52	7.10	6.73	8.00	**120.08**
C'	8.33	4.04	4.70	8.97	1.72	1.41	1.32	1.60	6.67	4.30	3.80	7.40	8.52	4.26	4.04	8.00	**79.09**

(c) Mapping between the D.3 and the template symbol

$(i,\pi(i))$	(1,1)	(11,2)	(12,3)	(13,4)	(14,5)	(10,6)	(15,7)	(16,8)	(2,9)	(4,10)	(3,11)	(5,12)	(7,13)	(6,14)	(8,15)	(9,16)	$\sum$
$lsub$	s1	s2				s3	s4		s5	s6		s7	s8		s9		
C	6.55	2.24	2.00	1.93	2.00	10.57	1.27	1.50	7.81	4.93	9.33	6.60	7.27	4.83	5.57	3.67	**78.08**
C'	6.55	0.45	0.40	0.39	0.40	10.57	0.76	0.90	7.81	2.96	5.60	6.60	4.36	2.90	3.34	2.20	**56.19**

(d) Mapping between the D.4 and the template symbol

Figure 4.3: The mapping costs for the symbols in Figure 4.2

4.3 Evaluation

The approach has been tested on the COAD2 data set, consisting of 4520 drawn symbols belonging to 113 classes (see Appendix A.5).

In literature several systems for facilitating the input of COA diagrams have been described [74, 75, 55]. In [55] an accuracy of about 90% when considering the top 3 interpretations on a set of 485 symbols has been obtained.

The experiment compares the performance of the previous algorithm to that of the enhanced version. The recognition procedure was executed at two different point sampling rates (64 and 128) for both the sketched and template symbols. Shape contexts with 5 concentric circles and 12 sectors were used. For each symbol drawn by a user, the matching cost with each of the 113 template symbols was computed. Then the list of the templates is ordered increasingly by the similarity to the unknown symbol and the position of the correctly matching template is considered.

For $n = 1, \ldots, 6$, the ratio of matching templates falling in the top n positions is reported. Tables 4.1 and 4.2 report the results of the above

Algorithm	top1	top2	top3	top4	top5	top6
C	91.0%	94.0%	95.6%	96.7%	97.0%	97.4%
C'	92.2%	97.1%	98.1%	98.4%	98.7%	98.9%
Diff.	+1.2%	+3.1%	+2.5%	+1.7%	+1.7%	+1.5%

Table 4.1: Result with 64 sampled points

Algorithm	top1	top2	top3	top4	top5	top6
C	92.0%	95.3%	96.3%	97.0%	97.6%	97.7%
C'	95.7%	98.2%	99.0%	99.1%	99.2%	99.2%
Diff.	+3.7%	+2.9%	+2.7%	+2.1%	+1.6%	+1.5%

Table 4.2: Result with 128 sampled points

described trials for 64 and 128 sampling rates, respectively. In both tables, the first and the second row report the performance of the original and the improved algorithms, respectively; the third row reports the improvement obtained with the latter over the former.

As can be seen, the improved algorithm has better performances in all cases. The improvement is more marked with a sampling rate of 128 points. In this case, a top 1 recognition rate of 95.7% is obtained. This rate improves that of the previous algorithm of 3.7%. The top 3 recognition rate is 99.0%.

4.4 Concluding remarks

This chapter has presented an adaptation to the case of sketched symbol recognition of the algorithm for shape matching proposed by Belongie et al. in 2002. The effectiveness of the algorithm has been proven by testing it on a set of more than 100 symbol classes. The proposed enhancement improves the recognition rate of the original algorithm, obtaining an accuracy of 99% when considering the top 3 interpretations on symbols sampled at 128 points.

Chapter 5

Recognition and Autocompletion of Partially Drawn Symbols by Using Polar Histograms as Spatial Relation Descriptors

In this chapter an approach for the recognition of partially drawn symbols and the design of a system for the autocompletion is proposed. Autocompletion has proven effective or appreciated by the users in various text-based applications [4, 5, 6, 7], while there is a lack of some research carrying out an evaluation of the performance of interfaces for autocompletion of hand drawn symbols in the scientific literature. To date, only a few systems [49, 32, 50, 51] have been introduced supporting such a feature for graphical symbols. Nevertheless, symbol autocompletion can be advantageous for the users in several applications. For instance, it can be useful to accelerate symbol retrieval in icon-driven user interfaces [76] or the handwriting of oriental characters, replacing the current complex systems, such as the *pinyin* coding system [77]. However, an advantage in terms of drawing speed and accuracy has only been reported for unistroke gestures [53].

The proposed approach uses an *Attributed Relational Graph* (ARG) [3] to represent symbols and is invariant with respect to scale. Furthermore, the user can plan to draw the symbol with the desired stroke number and order. The approach can work with a single *perfect* template for each class, without the need of a training phase to extract features or to select multiple templates. Being based on subgraph matching, the recognition can be performed on partially drawn symbols, i.e., when only a part of the primitives composing the symbol is available. An innovation of the presented approach is the use of a single spatial descriptor to represent relations between symbol components. The descriptor is an adaptation of the *shape context* [2] already used in the previous section and its use makes the approach free from the identification of the type of the *primitives* and from the check of fuzzy relations. The symbol matching is performed through an approximate graph matching procedure which incrementally produces new results as soon as more input strokes are available.

Different sets of symbols have been used to test the approach: a large set of symbols used to evaluate a previous method [50] and two differently sized sets of hand drawn symbols extracted from the real domain of *Military Course of Action* diagrams [56]. The symbols in the former set have also been artificially perturbed to test the invariance of the approach with respect to scale and its tolerance to random drawing errors. The results show that the proposed approach can recognize a reasonably high percentage of symbols even with a small number of available primitives. Furthermore, improvements in the recognition rate of partially drawn symbols are obtained against the existing approaches.

Also, since, to the best of my knowledge, no studies demonstrated a real advantage for autocompletion of multi-stroke symbols so far, the functionality of autocompletion has been evaluated from the point of view of the benefits for the users using the presented approach. In particular, the possible improvements, primarily in terms of efficiency, obtainable by the users when they exploit the autocompletion functionality are investigated through a user study using a basic interface.

The study, involving 14 participants, shows that the users can exploit the

autocompletion functionality in a profitable way, obtaining a faster input, with a time saving of about 18% in a task where participants had to draw symbols from the above described set. The advantage from the point of view of the accuracy has also been reported.

The chapter is organized as follows: the next section describes the approach for the recognition of partially drawn symbols; section 5.2 is devoted to outline the design of the interactive system for symbol autocompletion; section 5.3 presents the evaluation of the performance of the proposed approach in comparison to those of existing approaches; section 5.4 presents the user study about the human performance in hand drawn symbol autocompletion; lastly, some final remarks and a brief discussion conclude the chapter.

5.1 Recognition of partially drawn symbols

This section describes the method for the recognition of partially drawn symbols. In particular, it shows how to represent the symbols and how to calculate a distance measure between the hand drawn symbol and the template symbols. Since the method works with *primitives* and not directly with strokes, the ink must be pre-processed in order to extract the *primitives* from it. A symbol is represented through an ARG where each node is associated to a *primitive*, while each edge is associated to a descriptor coding the spatial relation between the *primitives* associated to its tail and head nodes. This descriptor is named *Primitive Spatial Relation descriptor* (*PSR descriptor*, in short). It is a histogram, inspired by the concept of *shape context* presented in [2]. The recognition relies on an approximate graph matching procedure, which returns the distance between the hand drawn symbol and a template symbol.

5.1.1 Symbol pre-processing

In the recognition of multi-stroke symbols a hand drawn symbol consists of a set of strokes, each starting with a *pen-down* and ending with a *pen-up* events. Since the method works with *primitives*, the *primitives* must be

(a) The drawn stroke. (b) Segmentation. (c) Clustering. (d) Sampling.

Figure 5.1: The steps of the pre-processor on a sample input stroke.

extracted from the digital ink. It is important to note that here *primitives* are referred as very simple graphics components, such as lines and curves (without corners). A similar definition is found in previous works describing structural approaches [50, 34]. A more recent work [48], instead, uses a broader definition, where more complex parts of symbols, such as rectangles, diamonds, and other polygons, are also regarded as *primitives*.

The pre-processing consists of different steps, including *segmentation* (fragmentation of strokes), *clustering* (grouping of strokes) and *sampling* (the extraction of representative points from a *primitive*).

The heaviest step in the pre-processing is *segmentation*. The segmentation process splits the input strokes by detecting cusps and produces a list of *primitives*. For this step the method presented in Chapter 3 is used, since it achieves very satisfactory results.

The second pre-processing step is *clustering*. This step is only executed on primitives belonging to the same stroke. In the proposed approach it is performed by simply evaluating the collinearity and the distance between the endpoints of two primitives. If two primitives pass the tests, they are merged together in one single primitive.

As regards *sampling*, the proposed approach requires that the *primitives* are all sampled at an equal number of points. Thus, for a fixed size n, a *primitive* P is represented as a set of points $(p_1, p_2, \ldots, p_n)$. Procedures to extract equally spaced points from strokes have been described in the literature (e.g. in [30]).

Figure 5.1 describes the steps of the pre-processor on a rectangular input stroke, which has been drawn starting from the middle of its left side (a). The drawn stroke is firstly segmented through the corner finding procedure (b); then, the two segments forming the left side are fused together in the

clustering step (c); lastly, all the primitives are sampled at the same number of points (d).

5.1.2 *PSR descriptor*

In order to describe how two *primitives* are spatially related within a symbol a relational descriptor named *Primitive Spatial Relation descriptor* (*PSR descriptor*, in short) is defined. Given two *primitives* Q and P, the *PSR descriptor* on the couple (Q, P) describes the position of the points $p_1, p_2, \ldots, p_n$ of P with respect to the center q_c of the bounding box of Q. More formally, a *PSR descriptor* is defined as a polar histogram as follows.

Definition 1 *Given two primitives Q and P, a PSR descriptor is a polar histogram $h_{(Q,P)}$ of the relative coordinates of the points $p_1, p_2, \ldots, p_n$ of P measured using q_c as the origin:*

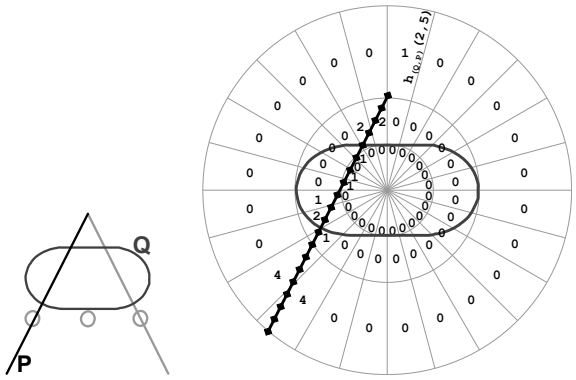
$$h_{(Q,P)}(i,j) = \#\{p_k \in P : (p_k - q_c) \in bin(i,j)\}$$

In the definition, the symbol # indicates set cardinality, $(p_k - q_c)$ computes the relative coordinates of a point p_k with respect to q_c, and $bin(i,j)$ indicates the bin resulting from the intersection of the i-th *annulus* and the j-th *sector* in the polar histogram.

It is worth noting that the *shape contex* definition in [2] builds up a histogram of a point set with regard to the bins. Definition 1 modifies the *shape contex* definition to ensure that the bins are defined with respect to the origin of another primitive.

By convention, the numbering for the *annuli* starts from 0 with the most inner *annulus*, while the numbering for the sectors starts from 0 with the sector with a side forming the smallest angle greater than 0° and proceeds counterclockwise. By convention, a point falling exactly on the boundary between two *annuli* (*sectors*) is assigned to the bin with the smallest *annulus* (*sector*) index. If a point falls exactly on the origin of the histogram, it is assigned to $bin(0,0)$.

As an example, Figure 5.2b shows the polar histogram for the PSR descriptor on (Q, P) where Q and P are the oval and the left caret side



(a) (b)

Figure 5.2: A symbol (a) and the PSR descriptor (with 3 circles and 24 sectors) with respect to its oval and left caret side *primitives* (b).

primitives of the symbol in Figure 5.2a, respectively. In particular, the *primitive* Q is centered at the origin of the histogram, and the *primitive* P is represented with its $n = 20$ equally spaced sampled points. Furthermore, the radius r of the histogram is equal to the distance from its origin to the farthest point of P. It is worth noting that this construction makes the PSR (and the approach) scale invariant. The histogram presents 3 concentric circles with radii $r/4$, $r/2$, and r (forming 3 *annuli*), and has 24 equally spaced sectors for a total of 72 bins, each labeled with the number of contained sampled points of P. As an example, in Figure 5.2b, $h_{(Q,P)}(2,5) = 1$ for the bin resulting from the intersection of the *annulus* 2 and the *sector* 5 in the polar histogram.

5.1.3 Symbol representation

A symbol is represented through an ARG: each node in the graph is associated to one of the *primitives* composing the symbol, while each edge is associated to the *PSR descriptor* coding the spatial relation between the *primitives* associated to its tail and head nodes.

Since the relation defined through the *PSR descriptor* is not symmetric and each *primitive* is in a relationship with all the others, the graph is directed and complete. Moreover, each node has a self-loop whose associated *PSR descriptor* is an alternative description of the *primitive* associated to that

node. Formally, by adapting the definition given in [3], an ARG is define as follows.

Definition 2 *A complete directed ARG over a set of attributes* $L = L_N \cup L_E$ $(L_N \cap L_E = \varnothing)$ *is a 4-tuple* $G = (N, E, \sigma, \tau)$ *where*

N	*is the set of nodes;*
$E = N \times N$	*is the set of all the directed edges, i.e., the set of all the distinct ordered pairs of nodes in N;*
L_N	*is a finite nonempty set of node attributes (primitive identifiers);*
L_E	*is a set of edge attributes (PSR descriptors);*
$\sigma : N \to L_N$	*is a function which associates a primitive identifier to a node;*
$\tau : E \to L_E$	*is a function which associates a PSR descriptor to an edge.*

An example of ARG representation of a symbol is shown in Figure 5.3.

A *symbol dictionary* is defined as a set of template symbols which, in turn, are represented each by one (or possibly more) *template ARGs*. In order to define a *template ARG*, a decomposition in *primitives* for the corresponding symbol must be provided. In order to correctly match the drawn symbol to a template, the pre-processing on the drawn symbol has to produce a decomposition as close as possible to the established one.

5.1.4 Symbol matching

In previous work [2], the distance between two polar histograms h and h' is computed as a function of the distances sought on all the pairs of the corresponding bins through χ^2 test statistic:

$$d(h, h') = \frac{1}{2} \sum_{i,j} \frac{[h(i,j) - h'(i,j)]^2}{h(i,j) + h'(i,j)} \tag{5.1}$$

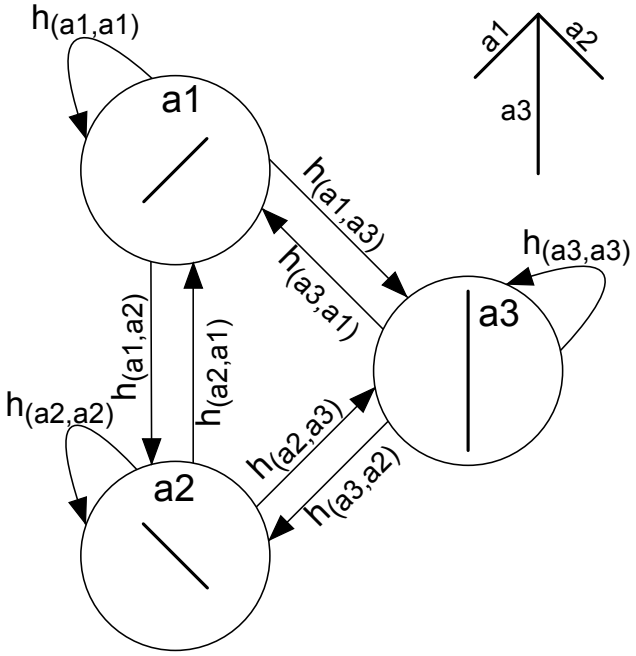


Figure 5.3: An arrow symbol and its representation as an ARG.

Here, a new, more effective and efficient, equation to calculate the distance between two histograms is proposed. In fact, the *shape context* was originally designed to describe the distribution of the points of an image around a fixed point. These points can be scattered throughout the plane. In this case, however, being part of a primitive, it is more likely that the points are localized in a narrow region of the plane. It may happen that, due to inaccuracies in the drawing, Equation (5.1) returns a high distance even for two histograms representing the same pair of primitives but differing in minor distortions. The conceived solution is to derive four polar histograms from the original one and to calculate the distance as a function of the distances sought between the corresponding quadruples of histograms. The four derived histograms are composed of different groupings of the bins in the original histogram. In particular:

- Two *coarse grained* histograms allow to locate the position of a primitive at a higher level, while two *fine grained* histograms offer a kind of "zoom" in that region.

- Two different rotation angles allow to obtain a greater accuracy in locating the position of points lying on the border between two bins.

Given a *PSR descriptor* h, the four histograms are *derived* from h using the following construction rules:

1. The four *derived* histograms have the same origin and equal radius of the *PSR descriptor* and each of their bins is defined as the union set of bins from the descriptor.

2. Two histograms are obtained from the *PSR descriptor* using two different *levels of granularity*: a coarse grained histogram h_{c0} and a fine grained histogram h_{f0}. Each bin of h_{c0} is the union of complete bins from h_{f0}; in the following, this will be referred to as the *bin containment property* between h_{c0} and h_{f0}.

3. The remaining two histograms h_{c1} and h_{f1} have the same form of h_{c0} and h_{f0}, respectively, but have different rotation angles. Thus, they are composed of different groupings of the bins in the original histogram. In particular, they are obtained by rotating clockwise h_{c0} and h_{f0} through angles θ_c and θ_f about the origin, respectively. Each rotation angle is half the magnitude of a sector of the histogram and it is such that each bin of h_{c1} is the union of complete bins from h_{f1}, i.e., the *bin containment property* must be preserved.

A 4-tuple of histograms, derived from a *PSR descriptor* h having the same parameters as that shown in Figure 5.2b is shown in Figure 5.4. They are constructed by instantiating the above rules as follows:

- h_{c0} has 5 bins: a central one $bin_{c0}(0,*)$ and four peripheral bins $bin_{c0}(1,j), j = 0,\ldots,3$, each defined as the union of bins of h as follows:

$$bin_{c0}(0,*) = \bigcup_{j=0}^{23} bin(0,j); \tag{5.2}$$

$$bin_{c0}(1,j) = \bigcup_{i=1}^{2} \bigcup_{k=6j}^{6(j+1)-1} bin(i,k); j = 0,\ldots,3. \tag{5.3}$$

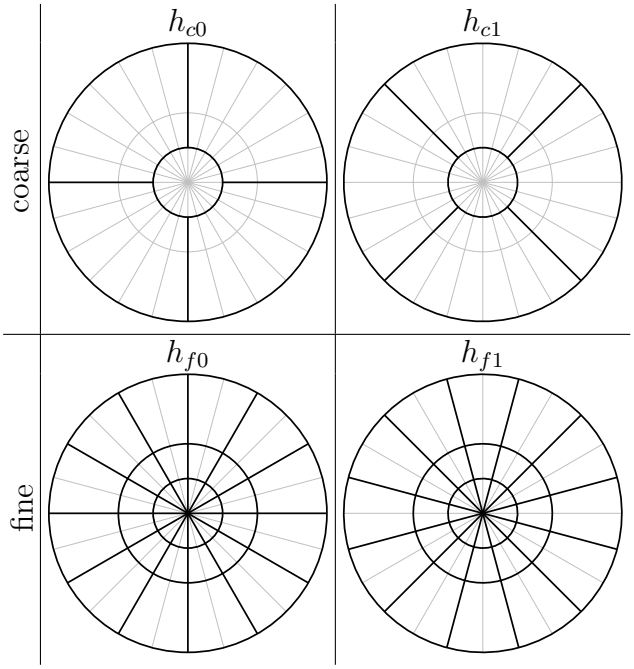


Figure 5.4: The 4 histograms used to calculate the PSR distance.

- h_{f0} has half the bins of h, since a sector in h_{f0} spans two sectors of h. More formally, the bins of h_{f0} are defined as the union of the bins of h as follows:

$$\begin{aligned} &bin_{f0}(i,j) = bin(i,2j) \cup bin(i,2j+1); \\ &i = 0,1,2; j = 0,\ldots,11. \end{aligned} \tag{5.4}$$

- h_{c1} is obtained by rotating h_{c0} clockwise of the angle $\theta_c = \pi/4$.
- h_{f1} is obtained by rotating h_{f0} clockwise of the angle $\theta_f = \pi/12$.

It is worth noting that the *bin containment property* holds both between h_{c0} and h_{f0}, and between h_{c1} and h_{f1}.

Given two *PSR descriptors* h and h', if the following distances between the corresponding *derived* histograms are set:

- $d_{c0} = d(h_{c0}, h'_{c0})$;
- $d_{c1} = d(h_{c1}, h'_{c1})$;

- $d_{f0} = d(h_{f0}, h'_{f0})$;
- $d_{f1} = d(h_{f1}, h'_{f1})$;

then the *PSR distance* $D(h, h')$ is defined as follows:

$$D(h, h') = min\{w \times d_{c0} + (1 - w) \times d_{f0}; w \times d_{c1} + (1 - w) \times d_{f1}\} \quad (5.5)$$

where w is a parameter to balance the weights of the coarse and fine grained histograms.

Effectiveness and efficiency of the *PSR distance*

This subsection informally demonstrate, through two examples, that Equation (5.5) is more suitable than Equation (5.1) to calculate the *PSR distance*. Although they are only examples, they are representative of frequent real cases, as it will be confirmed, through the presentation of empirical data, in section 5.3.3.

Figures 5.5 and 5.6 show the distances associated to two couples of primitives in their template (top row) and drawn (bottom row) versions. The first column shows the two versions of the symbol; the second column reports the *PSR descriptor* for both symbols, and their distance d, as calculated through Equation (5.1); the last column shows the four *derived* histograms and reports both the individual distance values, and the final *PSR distance* D, as calculated through Equation (5.5). The *PSR descriptors* and their 4-tuples of derived histograms used in the examples are instantiated with the same parameters of the ones shown in Figure 5.4. A value of 0.5 is assigned to the weight w.

Both figures report the template and drawn versions of a symbol composed of two primitives. Each *PSR descriptor* describes the position of the points of a primitive (P and P' in the template and in the drawn version, respectively) with respect to the center of the other one (Q and Q'). The drawn symbols reproduce rather faithfully the templates, except for minor distortions. Nevertheless, the distance measured through Equation (5.1) excessively emphasizes

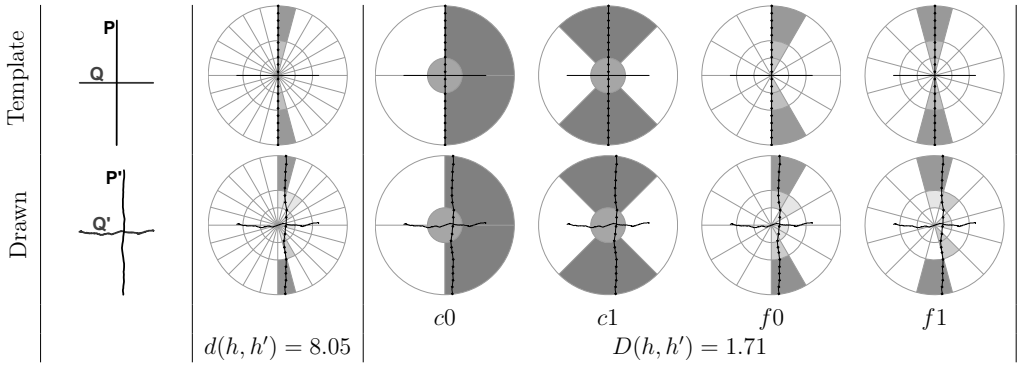


Figure 5.5: The effectiveness of histograms at different levels of granularity in the calculation of *PSR distance*

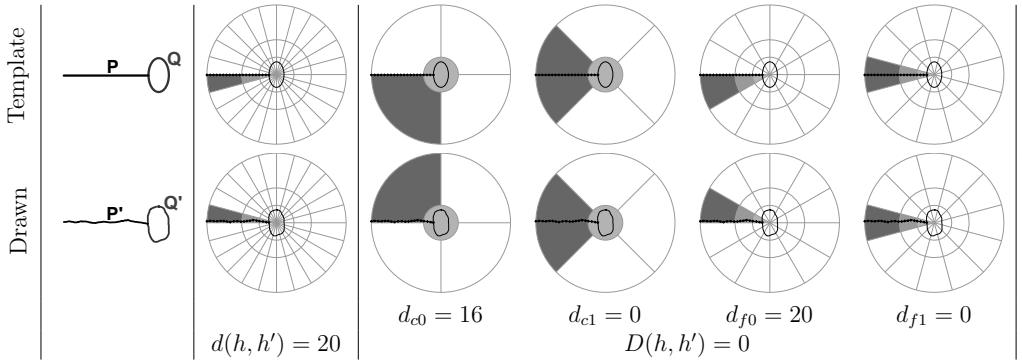


Figure 5.6: The effectiveness of histograms at different rotation angles in the calculation of *PSR distance*

the differences, bringing to an excessively high distance.

The effectiveness of the representation at two levels of granularity is demonstrated through the example in Figure 5.5: the distance measured through Equation (5.1) is $d(h, h') = 8.5$ while the *PSR distance* D is as low as 1.71. The reason is that the imprecision producing a small translation in the position of the P primitive in the drawn version has a great effect on the fine grained histograms and a very limited effect on the coarse grained ones.

The effectiveness of the representation with different rotation angles is demonstrated through the example in Figure 5.6. As it can be seen, while in h (and in its derived histograms h_{c0} and h_{f0}) all of the points of the segment fall in the lower-left quadrant, in h' (and in its derived histograms h'_{c0} and h'_{f0})) they all fall in the upper-left quadrant. This brings to a

distance $d(h, h') = 20$, which represents a complete dissimilarity between the two couples of primitives. The *PSR distance D*, instead, is 0, which more faithfully represents the relation between the two couples of primitives.

The respect of the *bin containment property* defined above, also guarantees a more efficient calculation of the *PSR distance* through Equation (5.5) than that calculated through Equation (5.1). Since each bin of h_{c0} and h_{c1} is respectively the union of complete bins from h_{f0} and h_{f1}, the calculation of the distance between two fine grained histograms can use partial results obtained in the calculation of the distance between two coarse grained histograms.

In particular, let $bin_{c0}(i, j)$ and $bin'_{c0}(i, j)$ belong, respectively, to two coarse histograms h_{c0} and h'_{c0} and have bin distance

$$bin\_dist_{c0}(i, j) = \frac{[h_{c0}(i, j) - h'_{c0}(i, j)]^2}{h_{c0}(i, j) + h'_{c0}(i, j)} \tag{5.6}$$

It can be proved that, if $h_{c0}(i, j) = 0$ or $h'_{c0}(i, j) = 0$ then

$$\sum_{x,y} bin\_dist_{f0}(x, y) = bin\_dist_{c0}(i, j) \tag{5.7}$$

where (x, y) ranges over the set $\{(k, l)|bin_{f0}(k, l) \in bin_{c0}(i, j)\}$.

As a proof, if $h_{c0}(i, j) = 0$, by using the *bin containment property*, then $\sum_{x,y} h_{f0}(x, y) = h_{c0}(i, j) = 0$. This implies that each $h_{f0}(x, y) = 0$ and then, by applying Equation (5.6) to $f0$, that each $bin\_dist_{f0}(x, y)$ reduces to $h'_{f0}(x, y)$. On the other end, $h_{c0}(i, j) = 0$ implies that $bin\_dist_{c0}(i, j)$ reduces to $h'_{c0}(i, j)$. By applying again the bin containment property:

$$\sum_{x,y} bin\_dist_{f0}(x, y) = \sum_{x,y} h'_{f0}(x, y) = h'_{c0}(i, j) = bin\_dist_{c0}(i, j)$$

which proves the assertion. A similar proof can be devised by considering as starting hypothesis $h'_{c0}(i, j) = 0$.

Note that Equation (5.7) holds also when applied to the rotated versions of the coarse and fine histograms if the bin containment property is preserved.

Equation (5.7) tells that the calculus of the *PSR distance* between two fine grained histograms can be made more efficient by using the distance values already calculated on the corresponding coarse histograms whenever the coarse grained histograms present empty bins.

As an example, consider the calculation of d_{f0} in Figure 5.5. Since no points fall in two of the five bins in the two corresponding coarse grained histograms, there is no need to calculate the contribution to the distance of the bins mapped on them. Since each external bin in a coarse histogram contains 6 bins of the fine grained one, the calculation of only 24 bin distances out of 36 is necessary. An analogous reasoning can be done for the case of the calculation of d_{f0} in Figure 5.6. Here, in the two corresponding coarse grained *derived* histograms, only the central bin contains points in both of them. Thus, the calculation of 24 *bin distances* is saved. Naturally, the more empty bins are present in the coarse histograms the more efficiency can be gained.

The matching distance between symbols

To estimate the similarity of the unknown partially drawn symbol to a template symbol, a number of the possible subgraph isomorphisms between the ARG associated to the drawn symbol and the *template ARG* is considered (since some of the possible isomorphisms are disregarded, an approximate results is obtained). A subgraph isomorphism is a one-to-one mapping between each node of the first graph and a node of the second graph. For each mapping its cost is calculated, and the minimum cost is selected as the *matching distance* between the two symbols.

In the following the *mapping cost* of a subgraph isomorphism and the *matching distance* between a template and a hand drawn symbols are formally defined.

Definition 3 *Given the template ARG* $G = (N, E, \sigma, \tau)$ *and the ARG* $G' = (N', E', \sigma', \tau')$ *for the drawn symbol, with* $|N'| \leq |N|$*, a injective total mapping* $M : N' \to N$*, and the PSR distances* D *on the PSR descriptors labeling the*

edges in E and E', then the mapping cost of M is calculated as follows:

$$C(M) = \frac{1}{|N'|^2} \times \sum_{P',Q' \in N'} D(h_{(Q',P')}, h_{(M(Q'),M(P'))}) \tag{5.8}$$

Given the set of all the available mappings $\mathcal{M} = \{M : N' \to N\}$, then the matching distance between a template and a hand drawn symbols is defined as $\min(\mathcal{M})$ *where the* min *function selects the mapping with the minimum cost.*

Informally, $C(M)$ is calculated as the average value of the *PSR distances* between couples of mapped primitives.

In the case the drawn symbol contains more primitives than the template one (i.e., $|N'| > |N|$), and this might occur either when the drawn symbol is mapped to a wrong smaller template symbol or when the symbol is drawn badly, then G is padded with dummy nodes and edges. *PSR distances* involving these edges are set to n (the number of sampled points in a primitive) to account for a penalty. It is worth noting that n is the maximum value for a *PSR distance*, i.e. the distance between two totally dissimilar PSRs.

The computational cost of calculating $C(M)$ is $O(n^2)$ where $n = |N'|$. Since $|\mathcal{M}| = |N'|!$, an exact graph matching algorithm would be $O(|N'|! \times |N'|^2)$. With this computational complexity, an exact graph matching becomes unfeasible as the number of primitives increases, it was chosen to use the approximate graph matching procedure described in the following section.

Approximate graph matching

The recognizer implements an approximate incremental graph matching procedure. Although in the previous section the *matching distance* is defined as a mapping with minimum cost, for efficiency reasons an approximate solution is acceptable. The proposed procedure keeps, at each step, a *Result List* L of the best k mappings, in ascending order by the value of the *mapping cost*: $L = \langle (M_0, C(M_0)), (M_1, C(M_1)), \ldots, (M_k, C(M_k)) \rangle$. The procedure goes through the following steps:

- *Step 1* (*Initialization*) is executed when the first drawn primitive P_1 is available. A simple ARG $G' = (N', E', \sigma', \tau')$ for the drawn symbol is

built: N' contains a node N'_1 corresponding to P_1; E' contains a self loop edge on N'_1, associated to *PSR descriptor* on (P_1,P_1). G' is then matched against each single-node subgraph of each *template ARG*. The *Result List* is instantiated and initialized with all the mappings and their corresponding *mapping cost*.

- *Step i* ($i = 2, 3, \ldots$) is executed as soon as a new drawn primitive P_i is available. G' is updated with the insertion of a node N'_i corresponding to P_i; edges to and from all the other nodes in N' and a self loop are added to E'; each added edge is associated to the corresponding *PSR descriptor*.

  - If $i = 2$, for each mapping $M : N' \rightarrow N$ in the *Result List*, let U be the set of the unmapped nodes of N:

    * new mappings are created, one for each element of U. The mappings are created by duplicating M and adding the pair (N'_2, U_j), with $U_j \in U$.
    * The mappings are added to the *Result List*.

    Finally, the *Result List* is truncated by maintaining only the best mappings.

  - If $i \geq 3$, the *Result List* is updated by adding the pair (N'_i, N_k) to each mapping, where N_k is the template node minimizing the cost of the mapping (as per Definition 3).

In the above procedure the matching between the partially drawn ARG produced at each step and the template ARGs is not started from scratch but is performed incrementally. In particular, after the first steps, the partial mapping obtained at the previous step is only updated with a new association between the best matching couple of nodes from the two graphs (case $i \geq 3$). Moreover, the cost of the mapping is incrementally calculated by adding the cost due to the new association to it. Nevertheless, the procedure does not guarantee that the best match is found.

This approximation is a compromise between the optimality of the results and the computational resource savings. If m is the number of *template ARGs* and n the maximum number of nodes in a *template ARG*:

- when $i = 1$, the *PSR distance* in Equation (5.5) is calculated ($|N|$) times for each template, which is asymptotically $O(m \times n)$;
- when $i = 2$, the distance is calculated $3\,|N|\,(|N| - 1)$ times for each template. In this expression, 3 is the number of edges incident to the new node added to the drawn ARG; $|N|$ is the number of mappings already present in the *Result List* (for the considered template), while $(|N| - 1)$ is the number of unmapped nodes of N. This expression is asymptotically $O(m \times n \times (n - 1))$ or rather $O(m \times n^2)$;
- at each i-th ($i \geq 3$) step, the distance is calculated $(|N| - i - 1)(2i - 1)$ for each of the k mappings occurring in the result list at step $i - 1$. In this expression, the first term produces the number of template nodes still to be mapped and the second term represents the number of edges incident to the new node added to the drawn ARG. This expression is asymptotically $O(k \times i)$.

In order to estimate the computational cost of executing the algorithm from scratch on a input primitives, its cost at each step must be summed: $O(m \times n + m \times n^2 + (k \times 3 + k \times 4 + \cdots + k \times a)$, or rather $O(m \times n^2 + k \times a^2)$. In the limit $a = n$ it becomes $O((m + k) \times n^2)$, and since k is a constant it can be written as $O(m \times n^2)$.

Figure 5.7 shows an example of the execution of the initial steps of the recognition procedure on the symbol dictionary shown in Figure A.7. Each row in the table corresponds to a step of execution. The left column reports the step number; the central column shows the partial drawing; the right column reports the top 5 results in the list. The parts of the template symbols mapped to the input primitives are highlighted in green. It is worth noting that the same template symbol can appear in the list multiple times with different mappings. The *mapping cost* is reported at the bottom of the template symbol. It is worth noting that in the case of items in the *Result*

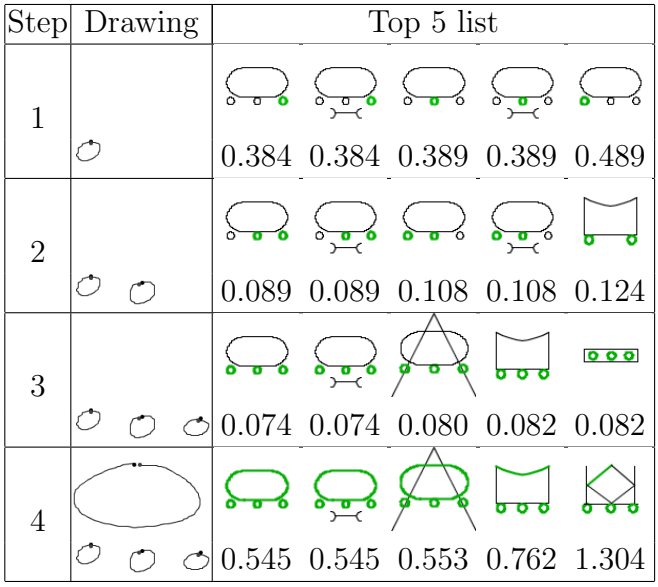


Figure 5.7: An example of the execution of the recognition procedure.

List with the same *mapping cost*, e.g. when the drawn symbol is a subset of more template symbols, no particular action is taken in sorting. On the one hand, preferring the simplest template may be more intuitive; on the other hand, preferring the most complex one may save much more time in autocompletion. In a real system the best choice must be calibrated on the basis of a series of parameters and depends on the application domain. In particular, it may be advantageous to show the symbol with the greatest frequency in the domain.

5.2 An interactive system for the autocompletion of hand drawn symbols

This section describes the design of an interactive system which assists the user in automatically completing the symbols to draw, based on the approach described in the previous section. After entering only a few primitives, the user can select a symbol from a candidate list instead of completing it. On the back-end, the approximate graph matching procedure described in Section 5.1.4 is used to incrementally produce new results as soon as more input

strokes are available. The system was developed in Java and is composed of a *front-end* and a *back-end* subsystems.

5.2.1 Back-end

The *back-end* subsystem is further divided into two main modules: a *Pre-Processor* and a *Recognizer*. The behavior of the *back-end* modules is graphically depicted through the activity diagram in Figure 5.8. While the user sketches a symbol, the *Pre-Processor* module extracts *primitives* one by one from the user input. As they are extracted, the primitives are passed to the *Recognizer* module which instantiates and incrementally updates a *Result List* of the *template* symbols that best match the (partial) input, ordered by similarity. More precisely, the *Recognizer* itself is initialized as soon as the first primitive is available from the input. This initialization step instantiates the *Result List* and corresponds to the execution of *Step 1* of the approximate graph matching procedure described in Section 5.1.4. Then, as soon as a new primitive is available from the input, a new recognition step (*Step i* in the procedure) is executed and the *Result List* is updated.

The *Pre-Processor* is fired as soon as the pen is released from the surface. As the user draws, the points of the stroke are resampled so as to be equidistant. Then the *segmentation* and *clustering* steps are executed. Finally, the extracted primitives are resampled at an equal number of points and a stream of *primitives* is produced and passed to the *Recognizer*. *Random forests* [78] were used as a learning method for building the classifier required in the *segmentation* step. The classifier was implemented in *R* language, using the *randomForest* package [69]. The call to the classifier from the main program is performed through the *Java/R Interface* (JRI), which enables the execution of *R* commands inside Java applications.

The front-end

The front-end is very simple and is shown in Figure 5.9. The view contains a canvas for drawing the symbol. On the left hand side of the canvas, a linear menu containing the top symbols from the *Result List* is shown. The best

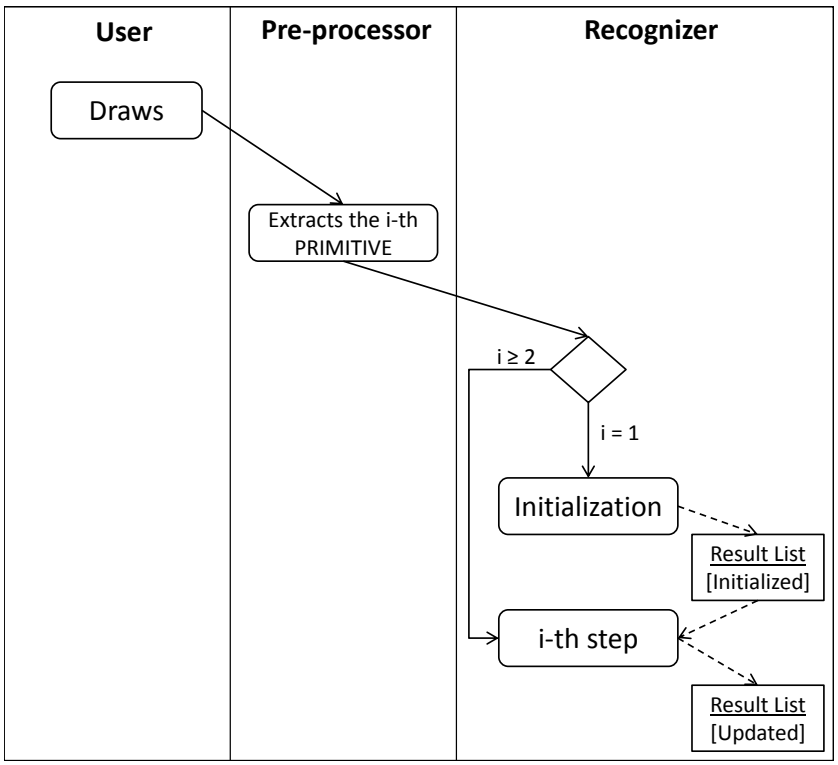


Figure 5.8: Description of the system *back-end*.

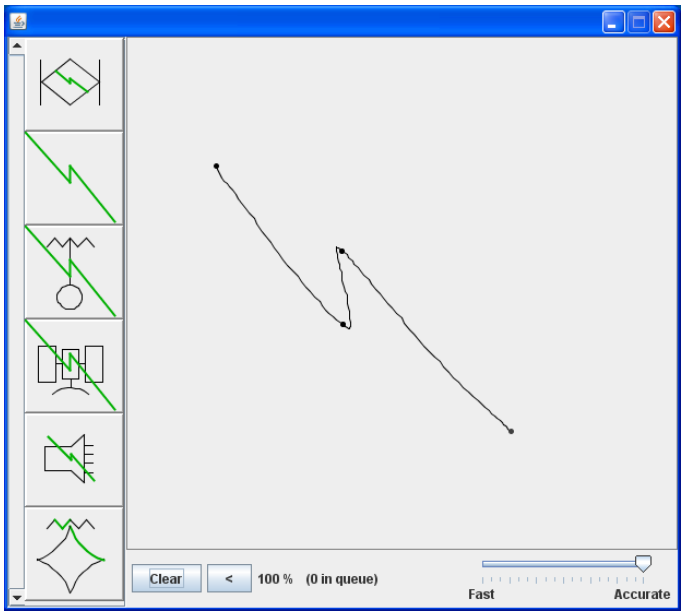


Figure 5.9: A screenshot of the interface for symbol autocompletion, using a list of 6 candidates.

candidate is on the top of the list menu. It is worth noting that, although the *Result List* can contain the same symbol multiple times, the interface only shows different symbols. A symbol is selected by tapping on it with the pointer. Whenever a selection occurs, the chosen symbol is pasted in the place of the partially drawn one. The size of the pasted symbols are also proportionate to those of the drawn one.

Due to the design choices of the back-end modules, the drawing style requires some light constraints to the users. In particular, the users can draw a symbol with their desired size and without caring of the stroke order. The presence of constraints are due to the efficiency requirements of the *Pre-Processor*: a stroke can be mapped on more primitives, but a primitive cannot be completed using multiple strokes. Furthermore, the users should not overtrace. The user is aware of the segmentation process through a real-time visual feedback: a marker highlights each detected corner point.

5.3 Evaluation

The proposed approach is extensively evaluated from multiple points of view. The recognizer is evaluated on the basis of its recognition capacity on partially drawn symbols, in comparison to other related methods (e.g. [50] and [49]). Three different tests are performed on as many data sets: a large set of composite symbols to compare to [50], a set of hand drawn symbols with 20 classes to compare to [49] and a larger set of hand drawn symbols with 113 classes. The purpose of the first two tests is to show the superiority of the proposed approach in recognizing partially drawn symbols, especially with a few primitives available. In the last test, the objective is to further explore the effectiveness of the proposed approach with a more complex set of hand drawn symbols.

The design choices in the development of the system are also justified on the basis of results obtained in tests on the data. In particular, the performance of the *PSR Descriptor* and the efficiency of the system for autocompletion are separately evaluated. Lastly this section reports the results of the evaluation, through a user study, of the conditions under which the users are willing to

Data Set	Num. of classes	Num. of symbols (drawers)	Source	Num. of primitives
Composite	97	$97 + 97^{*} + 97^{**}$ (/)	[50]	$2 - 13$ ($\mu = 5.9$; $\sigma = 2.6$)
COAD	20	640 (8)	[56]	$4 - 14$ ($\mu = 9.4$; $\sigma = 3.0$)
COAD2	113	4520 (8)	[56]	$2 - 19$ ($\mu = 5.9$; $\sigma = 2.9$)

The symbols in the *Composite* data set are not hand drawn.
* Artificially lightly deformed symbols.
* Artificially heavily deformed symbols.

Table 5.1: Features of the three data sets.

exploit the autocompletion functionality and those under which they can use it efficiently. All the data which were not collected from other sources were obtained through a *SMART Podium ID250* Interactive Pen Display (with a pen report rate of 100 points per second) connected to a *Dell Precision T5400* workstation equipped with an *Intel Xeon* CPU at 2.50 GHz running *Microsoft Windows XP* operating system and the *Java Run-Time Environment 6*.

5.3.1 Data sets

The proposed approach was tested on three different data sets. Two of them have already been introduced in the literature and used to measure the performance of some predecessors. Since the approach works well using a single perfect template per class, the templates extracted from the images contained in the source documents introducing them are used. The three data sets have heterogeneous features, which allowed the test of the validity of the proposed approach in varying circumstances. Some features are summarized in Table 5.1. The table reports, for each set, the number of different classes, the number of hand drawn symbols with the number of different drawers in parentheses, a reference to the source document where the templates were extracted from and some data related to the number of primitives of the symbols in the set. In particular, its range, average and standard deviation are reported.

The *Composite* data set (see Appendix A.3) contains 97 symbols in composite graphics (not hand drawn), plus 97 lightly deformed symbols and further 97 heavily deformed symbols. This data set was introduced by

Xiaogang et al. to measure the performance of their recognizer [50]. The symbols are already segmented, thus the pre-processing step will be skipped.

The *COAD* data set (see Appendix A.4) contains 620 drawn symbols, belonging to 20 different classes from the domain of *Military Course of Action Diagrams* [56]. This data set was used by Tirkaz et al. to measure the performance of their recognizer [49]. As already done by the authors in [49], the data set has been randomly split in a training set and a test set (containing 80% and 20% of the total number of symbols, respectively). The training set has only been used to train the pre-processor, while the symbol matching used the templates extracted from [56].

Since the *COAD* data set contains a relatively small number of different classes, a larger data set, called COAD2 (see Appendix A.5), was created from the same domain. This data set contains 4520 drawn symbols, belonging to 113 different classes.

5.3.2 Performance of the recognizer

The recognizer is evaluated on the basis of its recognition capacity on partially drawn symbols. A Java-based recognizer implementing the method with *PSR descriptors* with 3 circles and 24 sectors was developed; the primitives were sampled at 20 points; the weight parameter for the calculation of the *PSR distance* was set to $w = 0.5$. These parameters were chosen in a tuning phase with a different set of symbols. The *Result List* size of the approximate graph matching procedure described in Section 5.1.4 was set to $k = 1000$.

In order to compare the recognizer with two of the predecessor methods, two different tests are performed. No comparisons are available in literature among methods for partial hand drawn symbol recognizers. In [50], the proposed method is compared to a previous version of the method itself. In [49], the method is not directly compared to similar methods: its superiority is demonstrated by showing that it outperforms existing methods on full object recognition accuracies reported in the literature. Due to the unavailability of implementations of the above cited methods, the comparison is carried out on the basis of the results reported by them on the data set used in their

evaluation.

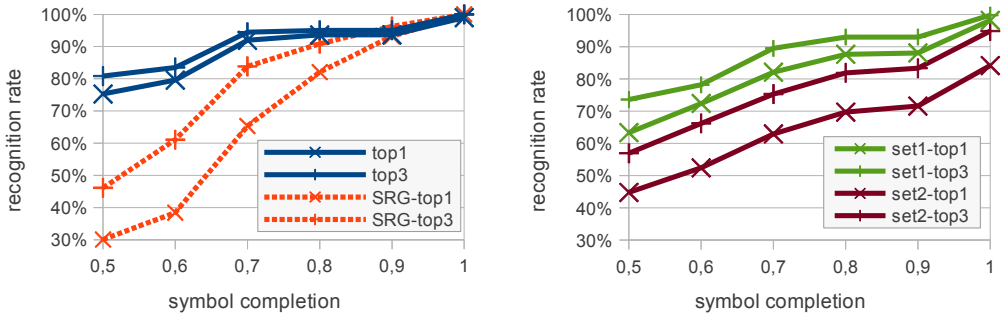
The measure used to evaluate the effectiveness of the proposed and other approaches is the recognition rate on top N interpretations for different values of N. This measure reports the percentage of times that the correctly matching template is in the top N positions of the candidate list returned by the recognizer. Furthermore, the recognition rate is calculated as a function of the number (or the percentage with respect to the total number) of the primitives drawn by the users.

Comparison to [50]

The method proposed by Xiaogang et al. [50], which represents a symbol through an SRG, was tested on the *Composite* data set. As already done with their SRG recognizer, in the test the templates in Figure A.3 are used to perform recognition. Furthermore, as unknown input symbols the same symbols in the set are used: for each symbol an incomplete version of it is taken, composed of a randomly chosen subset of the primitives. This evaluation procedure is the same followed in [50] and is replicated it as a means of comparison. It provides a useful tool to understand how the approach works with perfectly drawn partial input (best case).

Here, the recognition rate is calculated as a function of the percentage (with respect to the total number) of the primitives in the unknown symbols. Figure 5.10a shows the performances of the proposed recognizer (the continuous blue line) in comparison to those of their SRG recognizer (the dashed red line) on the set of 97 regular (not deformed) symbols. The horizontal axis reports the percentage of primitives drawn, while the vertical axis reports the recognition rate. Top 1 and top 3 interpretations are shown. Although the symbol set is a replication of the one originally used by the authors and then the comparison may contain small inaccuracies, a clear superiority of the proposed recognizer with a few primitives drawn can be noticed.

The presented recognizer has also been tested with the artificially deformed symbols, showing good performance even in this case. Figure 5.10b shows the performance of the proposed recognizer on the two artificially deformed



(a) Performance of the proposed recognizer compared to those of SRG [50].

(b) Performance of the proposed recognizer on two artificially deformed data sets.

Figure 5.10: Results on the *Composite* data set.

symbol: sets *set1* and *set2* in figure are the sets of lightly and heavily deformed symbols in the *Composite* data set, respectively.

Comparison to [49]

To compare the performance of the proposed approach to that of the method proposed by Tirkaz et al. [49], the proposed approach was tested on the COAD symbol set, i.e. the same used in their experiments. Here the performance of the proposed approach is evaluated in a similar way as they reported that of their system: the accuracy in terms of the top N classification on full and partial symbols separately. Furthermore, due to the possible ambiguity of both complete and partially drawn symbols in the COAD set, Tirkaz et al. let a human expert decide if each sample could be classified unambiguously for varying values of N or it was to be rejected. As a further comparison, the output of their human expert is reported as well. It is worth noting that fully drawn symbols may be rejected as well, as they may be confused with a partially drawn version of another symbol. The reject rate clearly varies with N, since a symbol can be ambiguous for a top N interpretation but not ambiguous for a top M interpretation ($M > N$).

To compare to [49] two simple conditions to reject the ambiguous symbols are defined. The conditions are based on the evaluation of the matching

distance of the first symbol in the candidate list and that of the (N+1)-th symbol. The first condition is verified if the ratio between the two distances is greater than a certain threshold T_1. The second is verified if the difference between the two distances is greater than a certain threshold T_2. The symbol is rejected if at least one of the two above conditions is verified. The T_1 and T_2 parameters are tuned using the training set of the COAD symbol set, by testing all the possible couples (T_1, T_2) (discretized at regular steps and varied within plausible ranges). The couples were chosen in order to have a reject rate lower than, but as close as possible to, the one of the human expert for the partially drawn symbols. For all values of N, the value of T_1 was set to 0.9. The value of T_2 was set to -0.14, -0.08 and -0.05 for N=1, N=2 and N=3, respectively. The accuracy of the system is much more sensitive to variations in the values of the parameter T_1, while the value of T_2 offers a further refinement.

The results of the comparison for N=1, N=2 and N=3 are reported in Tables 5.2, 5.3 and 5.4, respectively. Each table has three rows, corresponding to the performance of the proposed approach, that of the system described in [49] and that of the human expert, respectively. The columns report both the accuracies and the reject rates. The two approaches have comparable accuracies. As regards the accuracy on the partial symbols, the proposed approach seems to outperform the other method. In fact, the recognition rate is greater for each value of N. For N=1, the accuracy is much higher. Nevertheless, the proposed approach has a greater (but closer to that of the human expert) reject rate, as well. For the other values of N, the proposed approach has both a higher accuracy and a lower reject rate. Conversely, the method by Tirkaz et al. [49] seems to be more accurate on the full symbols: both methods always obtain 100%, but the proposed approach has a higher reject rate.

Test on a large hand drawn symbol set

To obtain a more accurate information on the performance of the recognizer on a set of hand drawn symbols, the proposed approach was tested on the

Method	**Partial accuracy (%)**	**Full accuracy (%)**	**Reject rate for partial (%)**	**Reject rate for full (%)**
Proposed approach	98.39	100.00	77.37	36.11
Tirkaz et al. [49]	92.65	100.00	70.82	17.52
Human [49]	100.00	100.00	75.36	33.58

Table 5.2: Test performance on the COAD data set for N=1.

Method	**Partial accuracy (%)**	**Full accuracy (%)**	**Reject rate for partial (%)**	**Reject rate for full (%)**
Proposed approach	98.02	100.00	63.14	24.07
Tirkaz et al. [49]	95.00	100.00	65.67	18.25
Human [49]	100.00	100.00	61.74	18.25

Table 5.3: Test performance on the COAD data set for N=2.

Method	**Partial accuracy (%)**	**Full accuracy (%)**	**Reject rate for partial (%)**	**Reject rate for full (%)**
Proposed approach	97.60	100.00	54.38	21.30
Tirkaz et al. [49]	97.53	100.00	65.24	17.52
Human [49]	100.00	100.00	55.07	12.41

Table 5.4: Test performance on the COAD data set for N=3.

larger symbol set COAD2. For each hand drawn symbol composed of n primitives, the recognizer was launched n times, each on the ARG built on the first $1, \ldots, n$ primitives drawn by the user, representing the symbol at a different completion status. The results are plotted in Figure 5.11. Here, the recognition rate is calculated as a function of the number of the primitives. From the chart it can be noted that the recognition rate is above 70% with only 2 available primitives when considering the top 6 interpretations.

5.3.3 Performance of the PSR descriptor

The calculation of the distance between two histograms through the *PSR distance* (Equation (5.5)) enhances the recognizer's performance from the points of view of both effectiveness and efficiency. To evaluate the enhancement, the recognizer was ran again on the COAD2 data set by using the

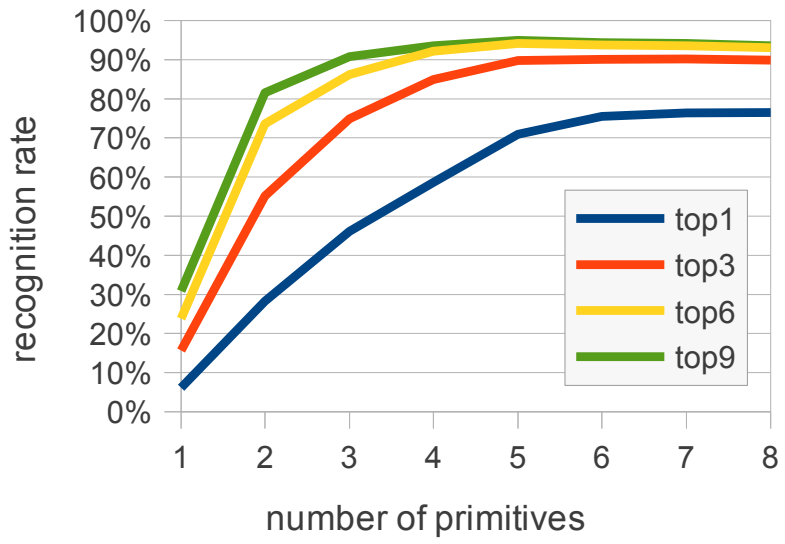


Figure 5.11: Recognition rate by the number of primitives drawn on the COAD2 data set.

traditional distance (Equation (5.1)). As regards the effectiveness, the results of the comparison are shown in Figure 5.12, as a function of the number of the primitives, for the top 1 interpretation. Using Equation (5.5) the improvement is in a range of about 5-7 percentage points, when at least two primitives have been drawn. Improvements, in some cases even higher, are also obtained with top 3, top 6 and top 9 interpretations. As for the efficiency, it is worth noting that when calculating the *PSR distance*, if no point falls in a sector of at least one of two corresponding coarse grained histograms, it is not necessary to calculate the contribution to the distance of the smaller bins mapped on them. On the COAD2 symbol set the frequency with which at least one of two corresponding bins is empty is 79.0% (88.3% for the central bins and 76.3% for the peripheral bins). This brings to a lower number of bin comparisons, which is reduced to about 26 on average (the worst case is 81), while with Equation (5.1) it is always 72. However, the reduction in the execution times is not proportional to the above numbers, due to the extra programming logic needed to implement the bin comparisons with Equation

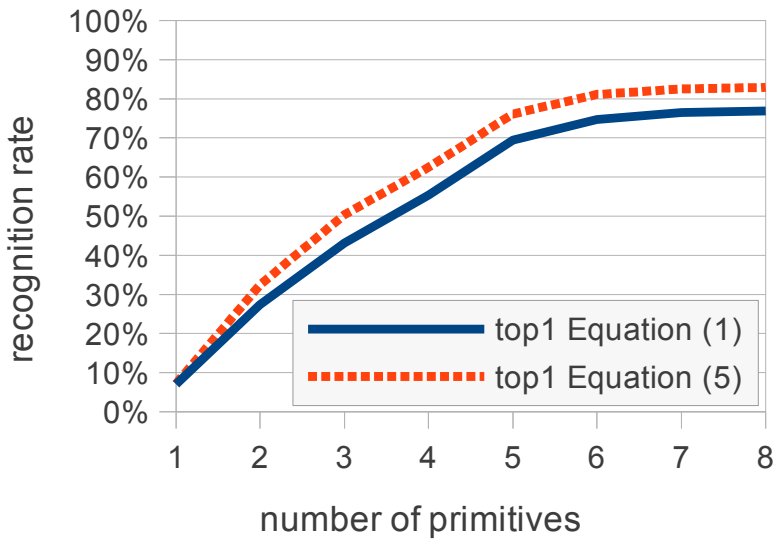


Figure 5.12: Recognition rate by the number of primitives drawn on the COAD2 data set for Equation (5.1) and Equation (5.5).

(5.5).

5.3.4 Performance of the interactive system

When tested on the COAD2 data set, each step of the incremental procedure takes ~45 milliseconds on the same apparatus used to collect the data. This result demonstrates the feasibility of the approximate graph matching algorithm for a real-time system. Obviously, the use of an approximate algorithm causes a loss of performance, when compared to a hypothetical exhaustive algorithm. To estimate this loss, the same test reported in Section 5.3.2 (on COAD2 data set) is executed using a nearly exhaustive algorithm (cutting only the most distant mappings and only in presence of a large number of primitives drawn). Very similar results to those shown in Figure 5.11 are obtained, with a maximum loss of one percentage point.

5.4 Experimenting the autocompletion functionality with users

The autocompletion functionality is evaluated in a user study. The research question was the following: is it really convenient in terms of efficiency to partially draw a symbol and choosing a candidate from a list or is it better to draw it completely?

To respond to this question fourteen participants (3 female), whose age ranged from 23 to 47 ($M = 30.4; S.D. = 7.5$) were recruited. All of them are right-handed and are habitual computer users. Then a single-factor within-subjects experiment was designed. The factor was the input technique (with and without the help of the autocompletion), while the main dependent variables were the symbol completion time (measured from the first *pen-down* event to the last *pen-up* or menu selection event) and the drawing accuracy. The symbol set was the one described in Appendix A.5.

The experimental procedure resembled that used in text entry experiments: each task consisted of copying an input symbol in the shortest possible time, balancing speed and drawing accuracy. An application prepared for the experiment showed the symbols one at a time and the participants were asked to transcribe each symbol after having observed it carefully. This procedure allowed to measure only the execution time, purging its measure from factors related to memory. The above task was administered to them in the following two conditions:

- *Manual*: The symbol must be drawn entirely by hand, without any assistance from the system;
- *Auto*: The symbol can be drawn entirely by hand or selecting a candidate from the menu, at user's choice.

In the study, the participants were asked to draw all of the 113 shapes once in both conditions using the interactive system described in Section 5.2 and were assigned to two equally sized groups, to counterbalance the execution order of the two conditions. The participants had to draw continuously, with

no relevant temporal breaks, during the task. Before the beginning of the experiment, they had an *induction* phase in which the objective and the procedure of the experiment were explained to them. They also went through a training phase in which they got acquainted with the symbols: they were given a printout of the whole set of symbols and were asked to observe and copy the symbols with paper and pencil. Then, under the supervision of an operator, they run an introductory session with the tablet in order to be sure they could correctly draw the symbols on it. They also tried to use autocompletion on all of the symbols. The drawing style was completely free for the *Manual* condition. In the *Auto* condition, instead, the participants had to comply to the light constraints described in the previous section: they were told to start and end a stroke in a corner of the symbol and not in the inner points of a segment or a curve (thus, circles must be completed using a single stroke) and not to overtrace.

Since the participants are left free to choose whether to use autocompletion in the *Auto* condition, the data is split in two groups: those from the tasks in which the autocompletion functionality has been actually used and the others. These groups are named the *Sel* and *NoSel* groups, respectively. The subdivision is performed on the tasks done in the *Auto* condition, but the *Sel* group also contains the data from the corresponding (obtained by the same participant on the same symbol) tasks in the *Manual* condition.

In the following the results of the experiment are reported. The considered measures are the drawing time, the drawing accuracy and the use of autocompletion in percentage. Also some free form comments from the participants are reported.

5.4.1 Completion times

A comparison of the performances measured in both conditions is shown in Figure 5.13. Each bar in the figure indicates the average time to complete a symbol from one of the three data sets *AllData*, *Sel* or *NoSel* under one of the two conditions *Auto* or *Manual*. For example, the blue bar indexed by *Sel* indicates the average time taken by the participants under the *Manual*

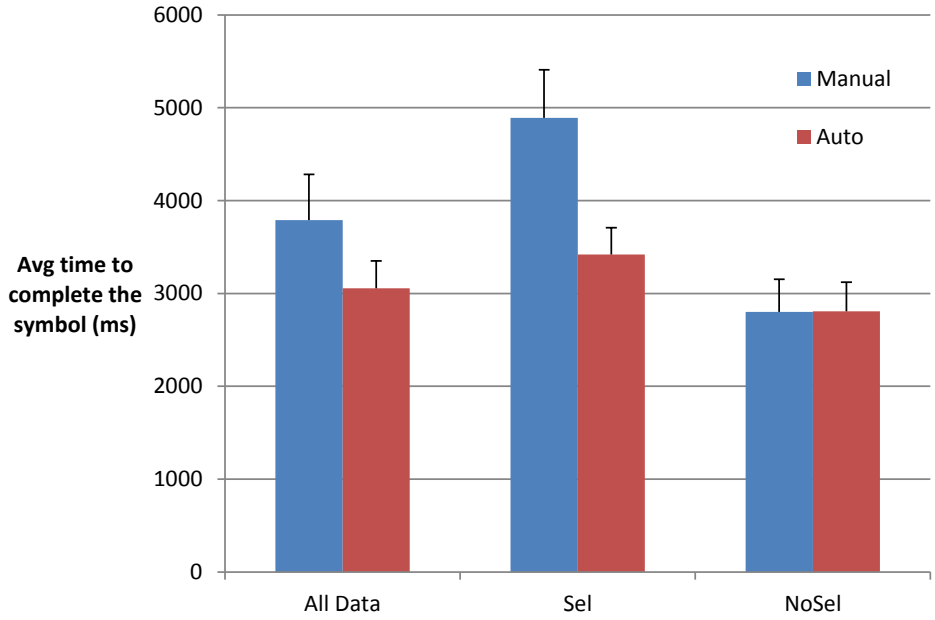


Figure 5.13: Average time needed to complete a symbol in both *Manual* and *Auto* conditions.

condition when drawing symbols in the data set *Sel*, i.e., when drawing symbols for which they have exploited autocompletion when under the *Auto* condition.

The *Auto* condition was more efficient than the other: overall, the average time to complete a symbol was 3"79 in the *Manual* condition and 3"06 in the *Auto* condition, with a 17.8% of time saving. The significance of the results are checked through a two-way with one within-subjects factor (the condition) and one between-subjects (the group) factor analysis of variance (ANOVA). The experiment revealed a highly significant effect of the condition on the completion times ($F_{1,12} = 20.7$, $p < .0001$). The group effect was not statistically significant ($F_{1,12} = .237$, ns), i.e., the counterbalancing worked. The condition $\times$ group interaction effect also failed to achieve statistical significance ($F_{1,12} = .364$, ns). This means there was no asymmetrical transfer of skill.

Considering only the data in the *Sel* group (those in which the auto-completion functionality has been actually used), a greater difference was

sought between the average completion time in the *Auto* condition (3"42) and in the *Manual* condition (4"89), with a drawing time saving of 28.7%. This difference was highly significant ($F_{1,12} = 41.8$, $p < .0001$). The small difference sought in the completion times of the symbols in the *NoSel* group (those in which the autocompletion functionality was not exploited by the user in both conditions) was not statistically significant ($F_{1,12} = 0.003$, ns).

Details on the performance of the 14 participants in the *Auto* condition are reported in Table 5.5: the table reports, in the first three columns of data, the percentage of drawing time saving with respect to the *Manual* condition. The last row of the table reports values averaged over the performance of all of the participants. Overall, 12 participants out of 14 obtained a time saving up to 33% for a single participant; the other two participants were slightly faster in the *Manual* condition.

5.4.2 Menu use

Table 5.5 reports in the last column, the ratio of the symbols completed through a menu selection. The participants judiciously used the menu. In fact, the menu was used with slightly less than half of the symbols presented to them. The lack of statistical significance of the small difference sought in the completion times of the symbols in the *NoSel* group indicates that the decision of not to use the autocompletion did not result in a significant delay in the completion times.

Not surprisingly, the frequency of use of the menu increases as the complexity of the drawn shapes increases: its value has a high correlation with the number of primitives ($c = 0.76$), the number of strokes required to complete the symbol in the *manual* condition ($c = 0.75$) and the time needed to complete the symbol in the *manual* condition ($c = 0.79$).

5.4.3 Analysis by the number of primitives

The number of primitives in a symbol clearly influence the need of using autocompletion. For this reason, a deeper analysis of the completion times was performed by using the complexity of the shapes as a parameter. In

	Time Saved vs the *manual* condition			
Participant	**All Shapes**	**Sel**	**NoSel**	**Menu Use**
P1	-22.82%	-36.78%	0.89%	43.36%
P2	-11.97%	-20.16%	18.43%	69.03%
P3	-17.17%	-33.10%	9.68%	48.67%
P4	-7.63%	-13.81%	-4.37%	23.01%
P5	-21.17%	-31.61%	-8.11%	46.02%
P6	1.45%	-8.86%	10.74%	33.63%
P7	-33.14%	-45.07%	-17.80%	42.48%
P8	-32.10%	-46.74%	3.98%	56.64%
P9	-9.77%	-22.04%	4.17%	38.94%
P10	4.63%	-10.00%	24.56%	42.48%
P11	-24.01%	-31.06%	-18.52%	30.97%
P12	-19.99%	-26.78%	8.00%	69.91%
P13	-33.23%	-40.27%	-12.92%	64.60%
P14	-22.82%	-36.09%	0.06%	50.44%
Mean	-17.84%	-28.74%	1.34%	47.16%

Table 5.5: Percentage of menu use and time saving in drawing symbols in the *Auto* condition.

particular, the set of 113 symbols was partitioned into 3 groups according to their number of primitives:

1. *simple* (43 symbols): 2-4 primitives;
2. *average* (46 symbols): 5-7 primitives;
3. *complex* (24 symbols): 8 or more primitives.

The average times to complete symbols belonging to the 3 groups in both conditions are reported in Figure 5.14. An advantage in the completion of the symbols belonging to all 3 groups has been obtained in the *Auto* condition. The advantage increases as the complexity of the symbols increases. Symbol 29, composed of 19 primitives, is an example of a complex symbol for which the menu provided a big saving: the average times required to complete it in both *Manual* and *Auto* conditions were 13"0 and 3"3, respectively. On this symbol, all of the participants used the menu obtaining an average time saving of 74.6%.

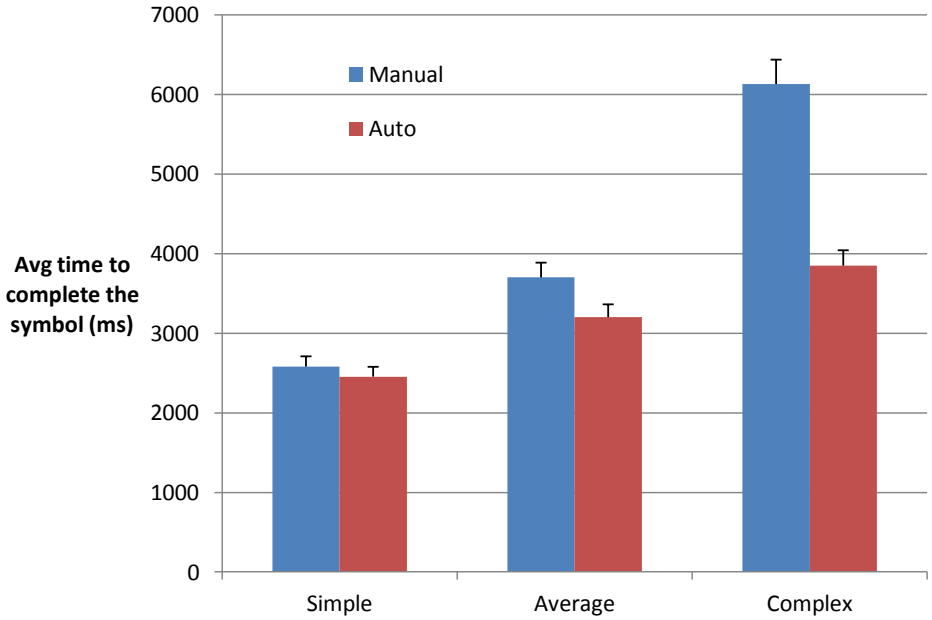


Figure 5.14: Performance in 3 groups.

It is also worth to report the percentage of menu use separately for the three groups. As expected, this percentage increases as the complexity of the symbols increases. The recorded values were 19.8%, 51.4% and 88.1% for the *simple*, *average* and *complex* group, respectively.

5.4.4 Accuracy

Autocompletion improves the accuracy of symbol recognition, since the action itself of selecting the right symbol from a candidate list, results in a correct recognition of the symbol by the system. Nevertheless, it is worth to report the extent of this improvement. The accuracy of the completed symbols in both *Manual* and *Auto* conditions was measured through a recently proposed multi-stroke symbol recognition method [42].

The recognition rates resulting from the analysis is 89.8% and 84.5% for the drawings in the *Auto* and in the *Manual* condition, respectively. Thus, the autocompletion functionality did help in obtaining a better recognition. This difference was sought to be statistically significant ($F_{1,12} = 10.8$, $p < .01$).

It is interesting to note that, considering only the symbols in the *NoSel* group, there is a small difference in accuracy in favor of the *Manual* condition (84.8% vs 82.9%). However, this difference was not statistically significant ($F_{1,12} = 7.1$, ns). This indicates that, on the manually completed symbols, participants used approximately the same accuracy in both conditions and the difference in accuracy on all the symbols is only due to the benefit of autocompletion.

It is worth noting that a candidate selection did not always end in a correct choice. Thus, the error rate in the use of the menu was also evaluated: the ratios of wrong selections have been measured per single participant and then averaged. The ratio of errors in menu selection was 4.31%. For most of them (3.27%) the participants realized they had selected the wrong symbol and changed the selection. A 1.04% of the selections, instead, remained unchanged and led to a wrong symbol selection.

5.4.5 Comments from the participants

After the experiment, the participants were asked about their impressions on the autocompletion functionality. Most of them felt that the it had helped them in speeding up the drawing process. Some participants complained about the arrangement of the symbols in the list: the upper symbols, which are the most likely to be selected, were often considered too far from the position of the pen to be selected efficiently and their position was often far from the view and thus difficult to be detected. The participants perceived positively the refresh of the list in real time. Lastly, all of the participants declared that they did not feel uncomfortable in adopting the constrained drawing style used in the *menu* condition.

5.5 Concluding remarks

This chapter has presented an approach for the recognition of partially drawn multi-stroke symbols. The approach is invariant with respect to scale, and supports symbol recognition independently from the number and order

of strokes. The recognition can produce reliable results with only a few available primitives, thus enabling the realization of a system supporting the autocompletion of the drawn symbol.

The effectiveness of the approach has been proven by testing it on three different symbol sets. The tests on the first two sets allowed the evaluation of the proposed approach in comparison to those of two other systems with similar features. With incomplete symbols, the proposed approach showed some superiority compared to both other systems. With full symbols, however, the system by Tirkaz et al. [49] showed better performances. The greater accuracy with partially drawn symbols, especially in the early stages of drawing, makes the proposed approach more suitable than the tested ones for autocompletion. Another feature that makes the proposed approach more desirable than others is the possibility of working with a single (perfect) template per symbol. In particular, the recognition phase of the proposed approach requires no training. This can be an advantage compared to the system in [49], which instead relies on training to acquire information about the drawing style of the users. It should also be said that the more the symbols are complex (in terms of number of primitives), the more the variability in drawing styles increases, making it more difficult to adopt the system in [49]. The good results obtained in the test on the set COAD2, the most complex among those examined, show that the proposed approach has good scalability with respect to the number and complexity of shapes in the set of symbols.

The good recognition performances of the approach, achievable even with a limited number of primitives, allowed the execution of a user study aimed at evaluating the feasibility of a basic symbol autocompletion system. The goal was to evaluate if the users can exploit the feature in an effective and efficient way. It is found that, applied to the COAD2 set of 113 symbols, autocompletion is advantageous: using a simple linear list with 6 candidates, the users can save about 18% of time with respect to sketching the whole symbol, on average. Other results presented in this chapter show the good performance of the proposed enhancement in the procedure to find corners with respect to [1], of the approximate graph matching procedure, and of the method for calculating the distance on *PSR Descriptors*, with respect to

previously known methods for histograms [2].

A limitation of the system for autocompletion is that it compels the user to adopt a constrained (but still natural) drawing style: a stroke can be mapped on more primitives, but a primitive cannot be completed using multiple strokes. For instance, with the used pre-processor, the user should not break ellipses, but should draw them in a single stroke. However, this constraint is not particularly uncomfortable for users, as previous experiences [48] show that they already naturally follow this style in the vast majority of cases. Furthermore, the experiment participants explicitly stated, in their freeform comments, that they did not feel uncomfortable in complying to the constrained drawing style.

Chapter 6

Identifying Attachment Areas on Sketched Symbols

In this chapter, in the context of graphical context detection, an approach for identifying attachment areas on sketched symbols is proposed.

According to a largely accepted model in the visual language community (see for example [79, 8]), the relations between the symbols of a visual sentence are geometrically defined through *attachment areas* of the symbols. For example, an arc of a graph is *entering* a node if the *head* of the arc is physically connected to the node *boundary*. Here, the relation *entering* between arc and node is defined on the attachment area *head* of the arc and the attachment area *boundary* of the node. In order to recognize relations between symbols it is then important to recognize the involved attachment areas.

Systems allowing users to build visual environments, such as *VLCC* [80] and *VisualDiaGen* [81], are often equipped with tools for defining the symbols of a language. The definition of a symbol includes both a physical aspect and a logical behaviour. The former is characterized by the visible features of the visual symbol, while the latter includes the relations with other symbols and the presence of visual or textual annotations attached to the symbol.

Attachment areas can have different shapes and are generally related to the physical appearance of the symbol. In particular, they can be originated

from single points, parts of the symbol or areas related to the symbol in some way. Figure 6.1 reports some examples of attachment areas on symbols taken from different domains. Among them there are: some vertexes for the *Conditional box* symbol, some sides (borders) for the *Multiplexer* symbol, and an inner area for the *Class* symbol. In this last case, the area is delimited by the visible ink of the drawn symbol. In the case of the *Schema* symbol of *Tic-Tac-Toe* game, the area of the top-left cell is only partially delimited.

In WIMP-based systems, an object can be placed on the canvas by using a menu. In this case attachment areas are automatically reported by the system. In a sketched language, due to the impreciseness of hand-drawing, actual attachment areas of shapes may be heavily deformed [8]. The management of areas which are not delimited by the visible ink of the drawn symbol can be even more difficult.

The proposed approach is independent from the domain of the symbols and from the method used to recognize symbols and assumes that the symbol has already been recognized. This also means that the ink drawn by the user to sketch the symbol has already been separated from the other ink in the diagram. The approach requires that the symbol and, more precisely both its physical and logical features, is defined in vector graphics. The identification of the attachment areas is performed by establishing a mapping between sampled points of both the sketched and the template symbol. The approach is evaluated through a user study in which users are required to sketch symbols from different domains and then to identify attachment areas on the drawn symbol.

The chapter is organized as follows: the next section discusses the proposed approach for the identification of the attachment areas; the subsequent section presents the results of the user study; lastly, some final remarks conclude the chapter.

6.1 The approach

In the proposed approach, both the visible ink and the attachment areas of the template are defined in vector graphics: elements of the symbol are defined

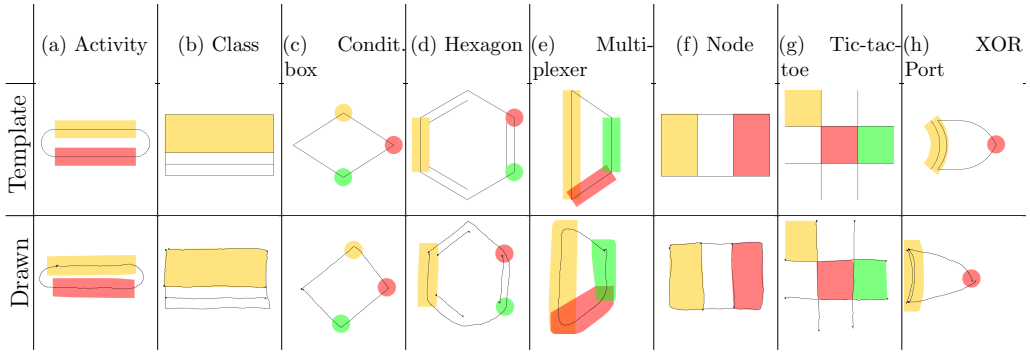


Figure 6.1: The symbols used to test the approach with their attachment areas highlighted

through geometric primitives such as lines and curves. The attachment areas are put in relation to the points of the visible ink: they can be a part (point or segment) of a primitive or being spatially identified to the closest points of the primitive. More in details, a containing area is defined using the existing points and the attachment area is defined as a part of it. The containing area is the smallest possible. A SVG editor is used to define both the visible ink and the attachment areas of the symbol.

The approach consists of the following steps:

1. Finding a matching between the points of the sketched symbol and the points of the template symbol;

2. Identification of the attachment area on the sketched symbol.

In the following, after describing how the symbols and their attachment areas are represented, two points above are detailed.

6.1.1 Symbol representation

A simplified version of SVG is used to represent symbols. The only primitive used in this simplified version is the polyline (identified through the *path* element). A polyline is defined through the succession of the endpoints of its segments.

In the proposed approach both the visual aspect of the symbol and its attachment areas are defined thus leading to two different kinds of primitives:

physical and *logical* ones. *Physical* primitives are used to represent the visible ink, while *logical* ones are only used to define attachment areas. Since some *physical* primitives exactly define attachment areas, they also behave as *logical* primitives.

With *logical* primitives three types of attachment areas are defined:

- **Point**: a point located on the visible ink of a symbol. Since a certain tolerance by user interaction is required, the point is defined as the center of a circular attachment area.

- **Border**: a polyline corresponding to a part of the visible ink of a symbol. The area surrounding the polyline is included in the attachment area for tolerance.

- **Area**: a polygon delimiting an area. If all of the vertices of the polygon belong to the visible ink of the symbol, the attachment area is said to be *closed*. If at least one vertex is not in any *physical* primitive, then it is said to be *open*. As an example, the attachment areas in the middle row of a Tic-tac-toe schema (see Figure 6.1g) are *closed areas*, while the top left attachment area is an *open area*.

A *thickness* parameter is defined for points and borders in order to handle tolerance.

In this SVG implementation the *id* attribute of the XML *path* element is used to represent different kinds of primitives and types of attachment areas through conventional names.

As an example, in figure 6.2 the first two elements define the visible primitives of the *XOR Port* symbol shown in figure 6.1h. The second element is also an attachment area of type *Border*. The value assigned to its *id* attribute is composed of its name (*inputEdge*) and the type of the attachment area (*border*) separated through an underscore character. The third element is a *logical* primitive defining the attachment area (a *point*) corresponding to the rightmost point of the symbol ink. A *logical* primitive is conventionally represented by terminating the *id* attribute with the *_logical* suffix. In figure 6.3, the first four elements above define the visible ink of the *Tic-tac-toe*

```
<path id="xor1" d="m 6.41,8.06 0.59,0.72 c ... z" style="fill:
    none;stroke:#000000" />
<path id="inputEdge_border" d="m 4.69,20.01 0.59,-0.72 c ..."
    ... />
<path id="outputPoint_point_logical" d="m 20.94,14.03" ... />
```

Figure 6.2: XOR Port code

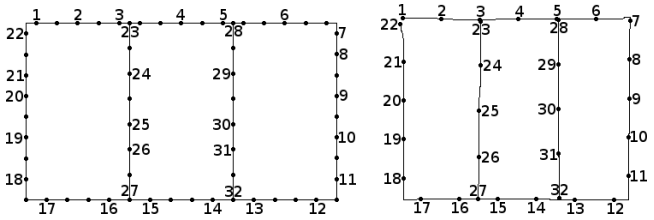
```
<path id="right" d="m 18.80,0 0,28.04" ... />
<path id="left" d="m 9.25,0 0,28.04" ... />
<path id="upper" d="m 0,9.25 28.04,0" ... />
<path id="bottom" d="m 0,18.80 28.04,0" ... />
<path id="central_closedArea_logical" d="m 9.25,9.25 9.55,0
    0,9.55 -9.55,0 z" ... />
<path id="right_closedArea_logical" d="m 18.80,9.25 9.25,0
    0,9.55 -9.25,0 z" ... />
<path id="topLeft_openArea_logical" d="m 0,0 9.25,0 0,9.24
    -9.24,0 z" ... />
```

Figure 6.3: Tic-tac-toe code

Schema symbol shown in figure 6.1g. The subsequent two elements define two *closed areas*, corresponding to the central and the central-right cells, respectively. The last element defines the *open area* corresponding to the top left cell.

6.1.2 Point matching

In the first step, a mapping is established between the points of the sketched symbol and those of the template symbol. Before matching the points, a set of points must be sampled from both the sketched and the template symbol.



(a) Template symbol (b) Drawn symbol

Figure 6.4: Point matching between a template and a drawn symbol

The procedure described in [30] is used to extract a set of equally spaced points from a stroke. Since a symbol can be composed of an arbitrary number of strokes, the points are allocated to the strokes proportionally to their length using the D'Hondt divisor method. In the proposed approach, the number of sampled points from the template symbol is twice the number of sampled points from the sketched symbol. This way, small non-proportional scale variations are better tolerated.

Once the points have been sampled in both the sketched and the template versions of the symbol, they are matched using a variation of the procedure described in [2]. Briefly, the procedure associates a shape descriptor to each point. The descriptor, called *shape context*, is a polar histogram describing the relation of the given point to the other points of the symbol. Two shape contexts can be compared using a *matching cost* function. A matching between the sets of points of the two versions which minimizes the cost is chosen. This is done through bipartite graph matching. The procedure has been modified to handle the different number of points between the template and the sketched versions: the values of the bins in the histogram of a point in the template symbols are halved to correctly measure the distance to a histogram of a point in the sketched version.

An example of point matching with 64 points in the template and 32 in the drawn symbol is shown in figure 6.4. The figure shows that the point matching procedure tolerates the small non-uniform scale variation due to drawing imprecision: all of the points on a line of the drawn symbol are correctly mapped on the corresponding line of the template symbol. If the two versions of the symbol were sampled at the same number of points, a less faithful correspondence would have been obtained.

6.1.3 Area identification

Given an area A_t defined in the template symbol, S'_t is the set of points of the template symbol falling into A_t. A subset S_t of these points will have a mapping with points on the sketched symbol. S_s is the set of points of the sketched symbol matching those in S_t. The identification of the attachment

area A_s (corresponding to A_t) on the sketched symbol is performed by using geometric features of both sets S_s and S_t. Depending on the area type, these features are used as follows:

- *Point*: A_s is obtained as a circle of center c_s and radius set to the defined *thickness* size. p_s and p_t are defined as the centroid of the sets S_s and S_t, respectively. c_s is calculated as follows: $p_s + (c_t - p_t)$, where c_t is the center of A_t;
- *Border*: A_s is obtained as the area including all of the points of distance smaller than *thickness* from the polyline connecting the points in S_s;
- *Closed area*: A_s is calculated as the closed area of the *convex hull* generated by the points in S_s;
- *Open area*: A_s is calculated as the morphing of A_t, such that the bounding box of S_s is equal to the bounding box of S_t.

It is worth noting that if the set S_t of mapped points is empty the approach cannot locate A_s and fails to identify the attachment area.

6.2 Evaluation

The approach has been evaluated through a user study aimed at comparing the attachment areas identified by the system to those perceived by the user who has drawn the symbol. To this aim, the users were required to sketch the symbols and then identify the attachment areas on them. A prototypical application has been developed to this aim. As shown in figure 6.5, the interface is vertically divided in two views. A symbol is shown on the left view, while the right view contains a canvas for drawing. The application was run on an *Asus EEE Tablet PC* with an Intel Atom processor at 1Ghz. The system was instantiated using 128 sampled points for the template and 64 for the sketched symbol. The *thickness* parameter (see sect. 6.1.1) was set to 30 pixels.

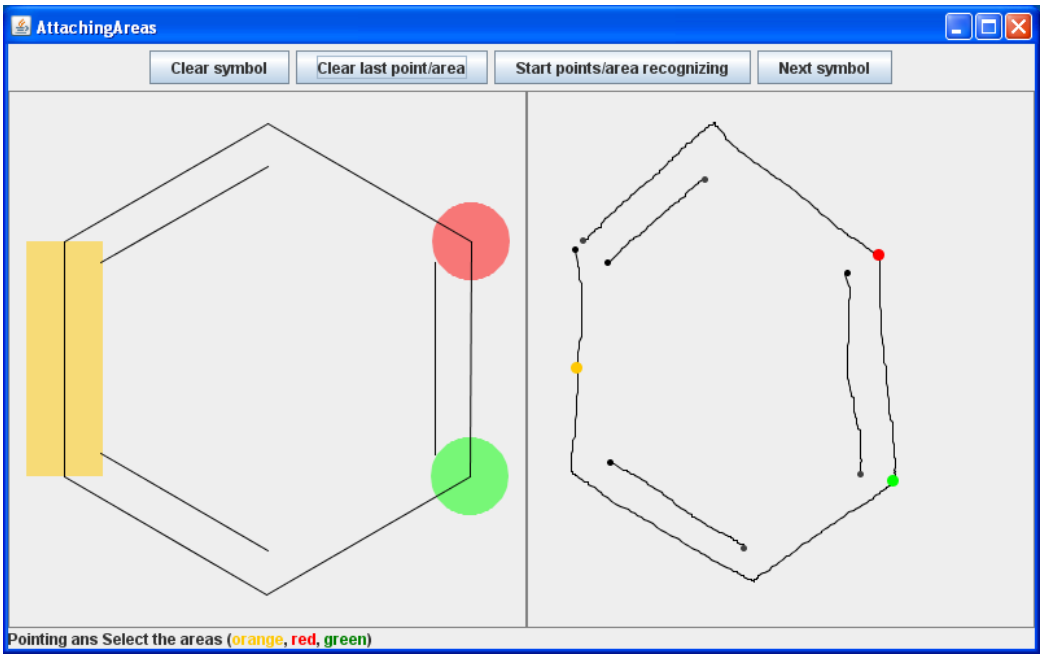


Figure 6.5: Task I: Pointing

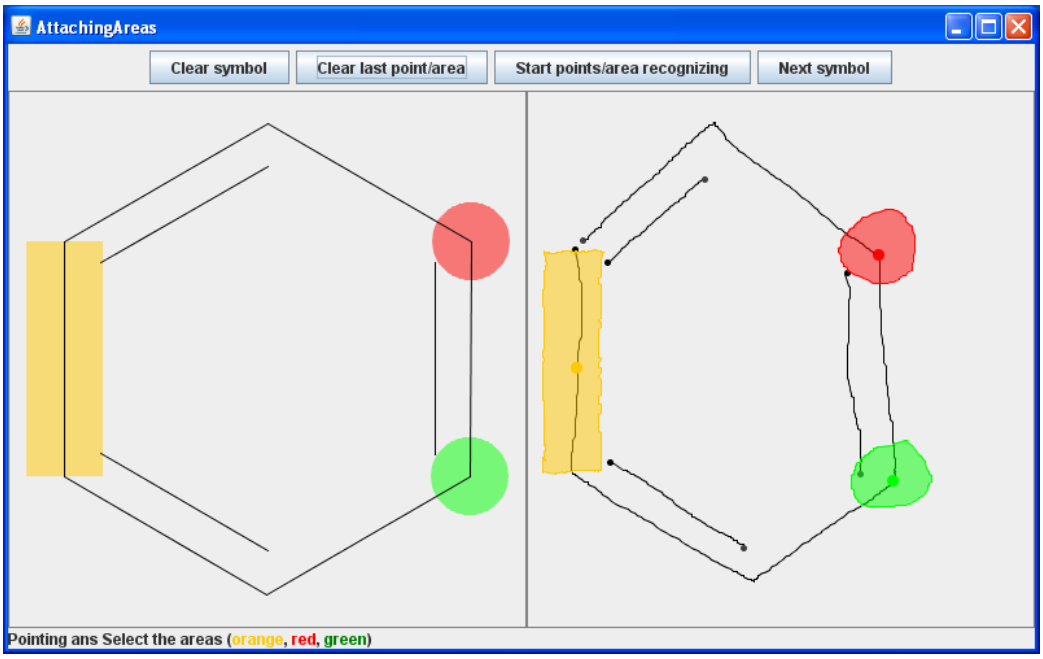


Figure 6.6: Task II: Selection

Eight (6 male, 2 female) unpaid adult volunteers were involved in the experiment. Their age ranged from 26 to 50 ($\mu = 34.1$, $\sigma = 9.4$). They were asked to identify the attachment areas through the following tasks:

- *Task 0: Sketching.* Given a template symbol (shown on the left view), the users had to reproduce the symbol using the provided pen in the right view. The 8 symbols of figure 6.1 were sequentially presented to the users through the interface. The users were recommended to draw as naturally as possible, balancing speed and accuracy.

- *Task I: Pointing.* Given the template symbol with some (up to 3) attachment areas highlighted (shown on the left view), the users had to tap the pen on their hand drawn version of the symbol (on the right view) in a point that they felt was inside the attachment area. They were recommended to be as accurate as possible, without any speed constraint. A screenshot of the application interface while running *Task I* is shown in figure 6.5.

- *Task II: Selection.* Given the template symbol with attachment areas highlighted, the users had to precisely reproduce the contours of the area through a pen stroke on their hand drawn version of the symbol, in a way similar to a *lasso* selection: this task is aimed at comparing the area identified by the system to that perceived by the user. As for *Task I*, the users were recommended to be very accurate. A screenshot of the application interface while running *Task II* is shown in figure 6.6.

6.2.1 Results

All of the 8 users performed 5 blocks, each including a single set of the three tasks. At the end of the experiment 8(users) x 5(blocks) x 8(templates) = 320 sketched symbols were collected. The 8 symbols contain 19 attachment area definitions: 6 points, 7 borders and 6 areas. Thus, experiment the data contain 760 tested cases of attachment areas: 240 points, 280 borders and 240 areas.

The results for *Task I* are reported in table 6.1. The performance is reported for each attachment type (*Point*, *Border* and *Area*) and is measured through the percentage of times the users were able to correctly point the pen in the attachment area identified by the system. In particular, three different measures are reported:

1. **Exact**: the pointer exactly fell inside the area identified by the system (only applicable for areas);

2. **Low Tolerance**: the pointer fell at a distance lower than 15 pixels (3.23 mm) from the point or the border identified by the system (not measured for areas);

3. **High Tolerance**: as the measure described at the previous point, with a tolerance augmented to 30 pixels (6.45 mm);

The results for *Task II* are reported in table 6.2. The performance is reported for each attachment type. Given the polygon S identified as the attachment area by the system, and the polygon U drawn by the user, the performance is measured through the average size of the following areas (standard deviation is reported in parenthesis):

1. **Intersection**: the intersection of S and U. This represents the area identified by both the system and the user as the attachment area;

2. $S - U$: the difference of S and U. This represents the area identified by system as the attachment area not included in the user selection;

3. $U - S$: the difference of U and S. This represents the area selected by the user not identified by system as the attachment area;

The size of the above areas is reported with respect (ratio) to the area of the union of S and U. With the settings reported in the previous section the system never failed to locate areas.

Type	Exact	Low Toler.	High Toler.
Point	n.a.	80.4%	98.8%
Border	n.a.	100%	100%
Area	99.6%	n.a.	n.a.

Table 6.1: Results of Task I.

Type	Intersection	S - U	U - S
Point	58.8% (14.4%)	26.5% (17.7%)	14.5% (11.6%)
Border	57.6% (15.6%)	40.2% (17.7%)	2.3% (4.2%)
Area	87.6% (9.2%)	6.6% (7.0%)	5.8% (6.7%)

Table 6.2: Results of Task II.

6.3 Concluding remarks

This chapter has presented an approach for identifying attachment areas on sketched symbols, independent from any domain and from the procedures used in the recognition and segmentation process.

The results for *Task I* show that, in most cases, the user can correctly locate an attachment area on the sketched symbol. In particular, they show a percentage close to 100% both for areas and for points and borders with a tolerance of 30 pixels. With a tolerance of 15 pixels the performance for borders is still optimal, while the performance for points degrades to 80.4%.

The results for *Task II* show that there is a reasonably good correspondence between the attachment areas found by the system and those devised by the user. This is particularly true for areas, where the intersection of the above is about 88% on average. The performance is lower with points and borders, where the intersection is close to 60%. It is worth noting that the user is prone to select a smaller area than that identified by the system, or, in other words, the system overestimates the size of attachment areas for points and borders. As a consequence, there is a low chance that the user misses the attachment area, and a higher chance that s/he unintentionally touches it. Due to the procedure used in *Task II*, the above results suffer from the imprecision of the user in correctly selecting with the pen his/her own devised attachment area. This is particularly true for borders and points, where the

user has no visible references as, on the contrary, is the case for *closed areas*. Unfortunately, description of more precise approaches were not found in the literature.

The above reported results show that the approach can correctly find the attachment points with a reasonable approximation.

Chapter 7

EulerSketch: a sketch system for Euler diagrams

Euler diagrams (EDs), a generalization of Venn diagrams, are a popular method for visualizing relationships between set-based data. They consist of a set of curves representing sets and their relationships. They are used in various information presentation applications as a simple, yet effective means of representing and interacting with set-based relationships. EDs form the basis of more expressive visual logics such as *Spider diagrams* [82] and *Constraint diagrams* [83], designed for software system specification and automated or interactive reasoning purposes. They are also utilized in various information presentation applications as a simple, yet effective means of representing and interacting with set-based relationships: e.g. in bio-informatics for the representation of genetic set relations [84]; to specify and display library database query results [85] within similar paradigms; in resource management systems [86, 87] to permit user categorization in non-hierarchical categorization structure; or network visualization [88] useful in social network data analysis, for example. In [89], they produce isocontours to highlight relationships amongst graph nodes within existing graph layout, useful in highlighting collections of venues within a map layout for instance.

Euler diagrams are an example of visual languages, since they represent exclusion, containment and intersection of sets in an intuitive way. In this

context, the ED curves represent the concrete level, while the abstract set of zones (which correspond to the set intersections represented as regions in the concrete ED) represents the abstract level. Alternative ED abstractions exists. In this thesis two new ED abstractions, called *static code* and *ordered Gauss paragraph (OGP) code* are considered. One was recently introduced for EDs [90], while the other is a slight variation of existing code for knots introduced in [15]. These two text encodings capture the topology of an Euler diagram in an abstract way.

This chapter presents EulerSketch, an experimental interactive system for the sketching and interpretation of EDs. Starting from the drawn ED, EulerSketch supports, in addition to classic editing operations (such as delete and move), the generation of the corresponding static and OGP code, and of the corresponding symbolic representation of the ED regions. These representations can be saved or displayed in text fields.

EulerSketch also allows the input or editing of the static or OGP code in order to build a corresponding concrete ED (i.e. its graphical representation).

The system is available in the Software section at `http://weblab.di.unisa.it`

7.1 Static code and ordered Gauss paragraph

This section informally describes the static and OGP code. For a more formal description, please refer to the paper in which they were defined [90, 15].

The static code can be written in two equivalent forms: the infix form and the postfix form. Starting from a concred ED, the infix static code is obtained by numbering the crossings of the curves, orienting the curves clockwise, traversing each curve in turn from an arbitrary base point, recording the order that the numbers are met in a cycle, and also recording the set of curves that contain each segment as a subscript between the two crossing numbers that define the segment. The corresponding zone encoding describes the sequence of segments that bound the regions comprising the zone, augmented with the extra information of curve label with a dot to indicate the segment is part of that curve. The postfix form differs only for the subscript position in the text.

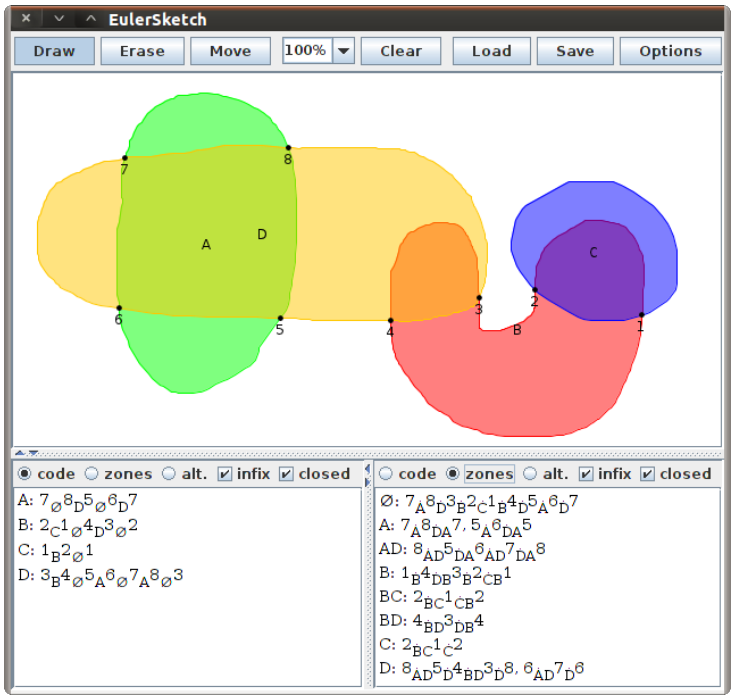


Figure 7.1: A screenshot of EulerSketch showing the static code (bottom left) and the corresponding zone encoding (bottom right).

Figure 7.1 shows an example of static code (bottom left) and the corresponding zone encoding (bottom right). The theory for the basic definition of static codes naturally extends to diagrams which are disconnected, or the presence of the points in which three or more curves intersect. EulerSketch supports these features.

The *OGP* code is computed from a diagram by: orienting the curves clockwise, assigning a number to each crossing, reading off the crossing numbers met in a single traversal of each of the curves, and adding a $+/-$ sign for each crossing (when traversing a curve, if one turns right onto the crossing curve, following its orientation, then $+$ is assigned, otherwise $-$ is assigned). The corresponding zone encoding describes the sequence of segments that bound the regions comprising the zone. An example of *OGP* code (bottom left) and the relative zone encoding (bottom right) are shown in Figure 7.2.

OGPs have a simpler syntax than the static code, requiring the inclusion

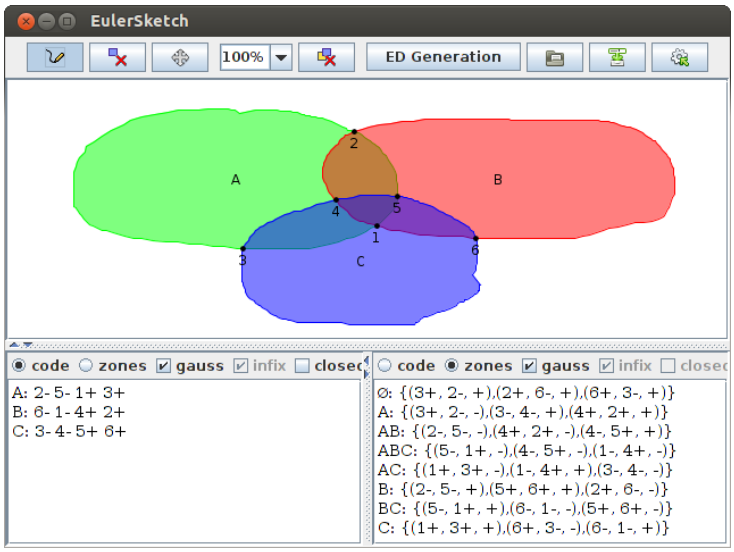


Figure 7.2: A screenshot of EulerSketch showing the OGP (bottom left) and the corresponding zone encoding (bottom right).

of signs to indicate the relative orientation of curves at each crossing versus the explicit association of the set of all containing curves to each segment in the diagram. They encapsulate the topology of connected diagrams (on the sphere), but require additional information (or some pre-processing) to deal with disconnected diagrams).

7.2 User interface

As shown in Figure 7.1, EulerSketch simple, intuitive interface, is horizontally divided into two views.

- In the upper part the *Sketch View* contains the drawing canvas. The curves are arbitrarily shaped and must be completed through a single pen stroke. Once a stroke has been entered, its endpoints are automatically joined to close the curve. Optionally, only simple curves can be allowed. In this case, whenever the stroke crosses itself, only the part of it containing the largest area is considered, while the remaining parts are discarded. At its completion, a curve is colored with a color from a pre-defined list. The interior is assigned a transparent shade (50% of

the original value in the alpha channel). The curves and the intersection points between them are automatically assigned a label, which are (optionally) shown as soon as a new curve is entered.

- In the lower part the *Code View* contains the automatically generated code. At the top of the view a toolbox contains a set of options through which the user can select the desired type of code; in particular, it is possible to show the static/OGP code corresponding to the ED or the corresponding zone encoding. For the static code, both the *infix* and the *postfix* notations can be displayed; the postfix notation used in [90] places the containing curve set for a segment after the pair of symbols for the segment, rather than in between the pair, as per the infix, which is more human readable. Figure 7.1 shows the infix code for both the curves (on the left) and the zones (on the right). The symbol $\emptyset$ is omitted in the infix static code leaving the containing curve set blank. The view can be split into more parts, optionally showing different codes.

EulerSketch has a *toolbox* with buttons to perform specific operations or to change settings. In particular, it is possible to: change the input mode to one of *draw*, *erase* and *move*; change the zoom level; clear the content of the whole canvas. Furthermore, the input and visualization settings can be altered in a separate dialog box. Finally, both the static/OGP code and the ED can be saved onto files which can be subsequently opened (or loaded). The former is saved as text while the latter is saved in vector graphics format. If a file containing the static code or the OGP is opened, then a corresponding ED is automatically generated and an additional component is used to display it in a pop-up window. A screenshot of the ED generated from the static code in the *Code View* of Figure 7.1 is shown in Figure 7.3. For clarity the figure shows als the embedding of the corresponding ED overlaid dual graph. Figure 7.4 shows instead the reconstruction from the OGP in the *Code View* of Figure 7.2.

Optionally, the curves can be beautified by connecting their intersection and segment nodes via closed cubic splines. However, this does not guarantee

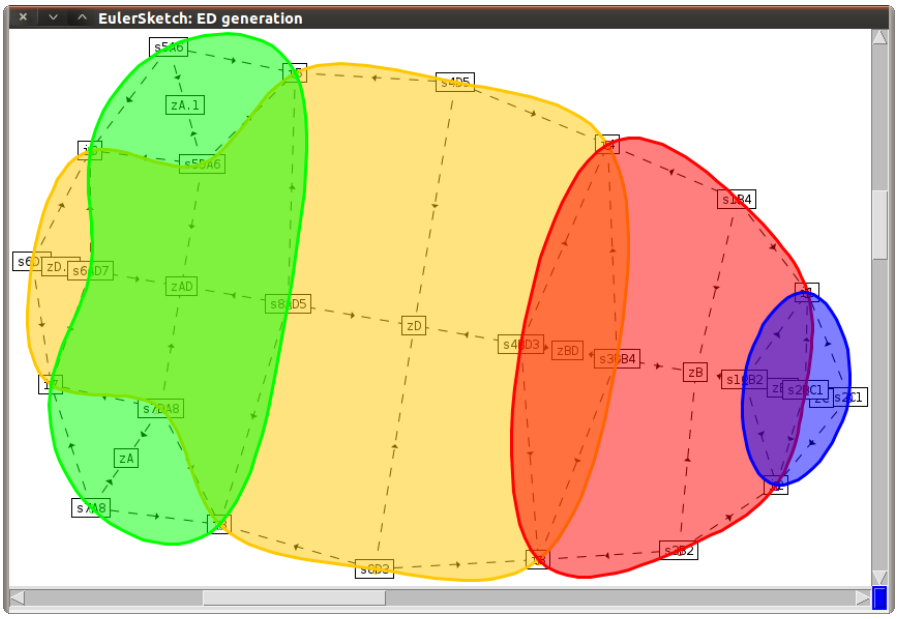


Figure 7.3: An ED reconstructed from the static code in Figure 7.1, showing also the embedding of the corresponding ED overlaid dual graph.

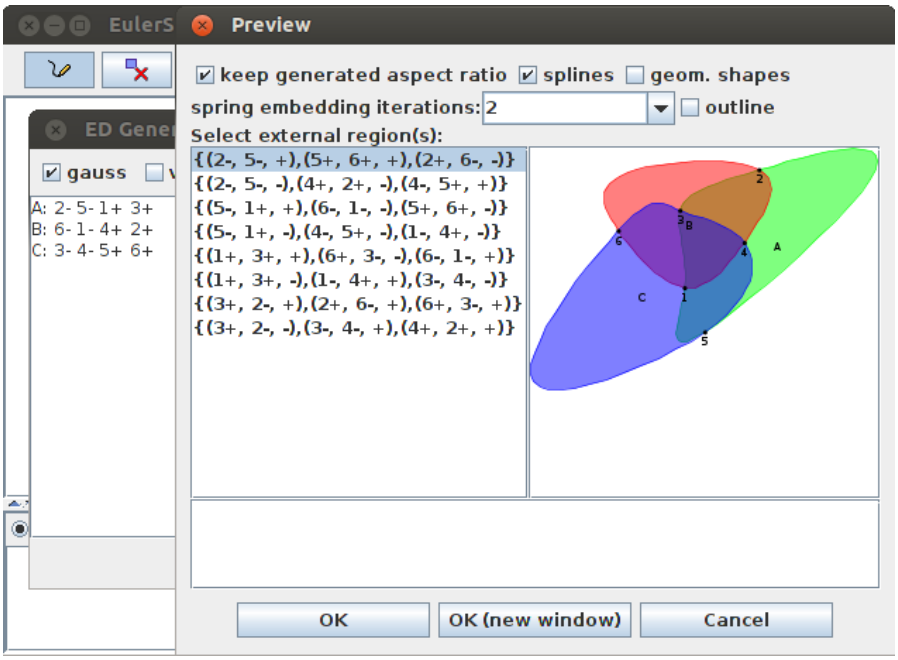


Figure 7.4: ED reconstruction from the OGP code shown in Figure 7.2.

that an ED generated from the static code looks similar to the original ED. Furthermore, the use of cubic splines can alter the properties of the diagram (e.g. by introducing intersections not indicated by the code). Therefore, as is commonly the case, diagram beautification introduces further problems to be addressed in future work.

7.3 Back-end

The main features of the application are the code generation, the zone encoding and the ED generation. The static code and OGP are incrementally constructed, and updated every time a curve is added or deleted. It is stored in an internal format enabling efficient execution of these operations. A curve is represented as a closed polyline. To reduce variations in the code due to orientation changes, each curve is automatically oriented clockwise independently of the construction (i.e. if a stroke defining a curve is entered counterclockwise, the sequence of its points is reversed). Starting from the generated static code or OGP, a procedure is used to calculate the relative zone encoding. This procedure can be triggered on user demand in order to display the encoding in the *Code View*, or it can be invoked by the system in order to assign a different colour to each zone in the *Sketch View* (this optional feature is a potentially useful variation of the standard visualization).

The approaches described in [15] and [16], based on overlaid dual graph method, are used to generate a new ED from a static code or OGP, using different techniques for planar graph embedding, such as techniques based on Boyer's algorithm [91], Bertault's algorithm [92] or Tutte embedding [93]. In particular, the Tutte embedding has been used to generate the embeddings in Figures 7.3, while Bertault's algorithm implemented in the *Open Graph Drawing Framework (OGDF)* [94] has been used to generate the ED in Figure 7.4.

7.4 Concluding remarks

This chapter has presented EulerSketch, an interactive system for the sketching and interpretation of EDs. EulerSketch interprets hand drawn EDs and produces two types of text encodings of the ED topology called *static code* and *ordered Gauss paragraph* (OGP) code, and an encoding of its regions. Also, given the topology of an ED expressed through static or OGP code, EulerSketch automatically generates a new topologically equivalent ED in its graphical representation.

Given the simplicity of Euler diagrams, it was not necessary to use the sketch recognition techniques proposed so far in EulerSketch. Nevertheless, EulerSketch is still an interesting prototype in that it shows a concrete example of the possibilities given by the interpretation and translation of sketches.

Chapter 8

Achievements and Future Research

In this thesis, methods and applications for sketch recognition have been presented by facing problems such as corner detection, sketched symbol recognition and autocompletion, graphical context detection, sketched euler diagram interpretation.

The proposed corner detection algorithm, RankFrag, improves the accuracy percentages compared to other methods recently proposed in the literature.

The presented multi-stroke hand drawn symbol recognizer, invariant with respect to scaling and to the number and order of strokes, outperforms the method proposed by Belongie et al. [2] on the *Military Course of Action* domain.

The proposed method for recognizing multi-stroke *partially* hand drawn symbols, still invariant with respect to scaling and to the number and order of strokes, presents a satisfactory recognition rate with partially drawn symbols outperforming existing approaches. Moreover, a user study is run to show that the users benefit of a drawing time saving (of about 18%) when exploiting the hand drawn symbol autocompletion functionality.

In the case of graphical context detection, the proposed method for identifying symbol *attachment areas* is evaluated through a user study that compares the attachment areas detected by the system to those devised by the

users. The comparison results show that the attachment areas are identified with a reasonable accuracy.

The presented graphical environment EulerSketch is an interesting prototype showing a concrete example of interpretation and translation of sketches. It allows to hand drawn Euler diagrams and to produce two types of text encodings of their topology. Moreover, conversely, it allows to automatically generate equivalent Euler diagrams from each of the two types of topology encodings.

The thesis presents the methods in isolation, however, they can be used either individually or as part of an integrated system for the recognition of complex diagrams. As future work, both ways are worth to be further investigated.

In the case of an integrated system a first problem to face is "tokenizing" the diagram, i.e., partitioning the diagram in its constituent symbols. In general, the problem is not easy because some strokes must be divided into substrokes if they contribute to more than one symbol (segmentation) and substrokes and other strokes must be grouped when forming a single symbol (clustering).

Once all the symbols have been recognized, the syntactic correctness of the diagram must be verified. Usually this is not a sequential process since the high ambiguity of the sketches and the difficulty of isolating each symbol may be helped by exploiting the knowledge of the syntax of the visual language. This research is in line with previous work [44, 95] which however did not consider autocompletion.

When considering the contributions as isolated piece of work, each of them has specific future work to be developed. In the following, some of them are listed.

Regarding RankFrag, the non-JRI implementation is able to produce the segmentation of a stroke in real time on a sufficiently powerful device. Future work will aim to achieve further implementation improvements, in order to further reduce the execution time and make the technique applicable in real time on more strokes at once (e.g., an entire diagram) or on mobile devices with low computational power.

Regarding the method for the recognition of multi-stroke hand drawn symbols, future work include the test of the algorithm on larger sets of symbols, such as hand written oriental characters.

As regards the method for the recognition of partially hand drawn symbols, future work include the possibility to enable autocompletion on larger symbol sets. Probably, some optimizations will be necessary to execute the approach in real time on data sets of that magnitude, such as the use of pruning strategies for limiting the number of times the PSR distance is calculated. Furthermore, other adjustments related to the specific data set may be performed: for instance, Chinese characters contain radicals, i.e. components common to several characters; this characteristic may be exploited to allow further optimizations.

Both of the above methods are only designed for the recognition of a single symbol at a time. In the case of diagram recognition, since both are multi-stroke methods, they might be extended to include the segmentation and the clustering operations as mentioned above.

The autocompletion functionality has been tested by the users using a simple linear menu. As future work, more efficient interfaces can be designed in order to obtain better results.

Regarding the approach for identifying attachment areas on sketched symbols, future work include the test of the approach on rotated or intentionally non-uniform scaled symbols, and the implementation of the approach in real systems and with different domains.

It is worth nothing that the presented recognition methods based on *shape context* are not designed to work on rotated symbols. However, this functionality can be added by exploiting the work by [96] where point matching through shape context has been extended to rotated shapes.

Moreover, although not described in this thesis, the method for recognizing multi-stroke partially hand drawn symbols can be easily made rotation invariant by constructing the PSR Descriptor after performing an alignment of the axis connecting the two primitive's centroids to the x-axis of the frame.

Appendix A

Data sets

This appendix describes the data sets used in the previous chapters. The five data sets have heterogeneous features. Some features are summarized in Table A.1. The table reports, for each set, the number of different classes, the number of hand drawn symbols, the total number of strokes composing the symbols, the number of different drawers, a reference to the document where it is introduced (with the source document from which the templates were extracted in parentheses) and some data related to the number of primitives of the symbols in the set. In particular, its range, average and standard deviation are reported.

Data Set	**Num. of classes**	**Num. of symbols**	**No. of strokes**	**No. of drawers**	**Source (templates)**	**Num. of primitives**
IStraw	10	400	400	10	[21]	$3-9$ ($\mu = 5.5$; $\sigma = 2.1$)
NicIcon	10	400	1204	32	[70]	$2-11$ ($\mu = 5.2$; $\sigma = 1.7$)
Composite	97	$97 + 97^{*} + 97^{**}$	/	/	[50]	$2-13$ ($\mu = 5.9$; $\sigma = 2.6$)
COAD	20	620	2255	8	[49] ([56])	$4-14$ ($\mu = 9.4$; $\sigma = 3.0$)
COAD2	113	4520	17606	8	([56])	$2-19$ ($\mu = 5.9$; $\sigma = 2.9$)

The symbols in the *Composite* data set are not hand drawn.
* Artificially lightly deformed symbols.
* Artificially heavily deformed symbols.

Table A.1: Features of the three data sets.

Figure A.1: One random sample from each class of the IStraw symbol set.

Figure A.2: One random sample from each class of the NicIcon symbol set.

A.1 IStraw

The *IStraw* data set is a set of 400 unistroke symbols belonging to 10 different classes (composed by both line and arc primitives), drawn by 10 different subjects. It is an *out-of-context* data set, i.e., it is not linked to a domain. It was used to test the homonymous corner finding technique [21]. The corners present in each symbol are also identified. Figure A.1 shows a random sample from each class of the symbol set.

A.2 NicIcon

The *NicIcon* data set is a set of 400 multi-stroke symbols belonging to 10 different classes, annotated by Tumen and Sezgin [23] (who identified the corners present in each symbol). It is a subset of the *NicIcon Database of Handwritten Icons* [70], which is a set of symbols drawn by 32 different subjects, gathered for assessing pen input recognition technologies, representing images for emergency management applications. Figure A.2 shows a random sample from each class of the symbol set.

A.3 Composite

The *Composite* data set contains 97 symbols. The symbols are in composite graphics (not hand drawn), that is, they are iconic symbols composed of simple graphic primitives, including only line and arc segments. This data

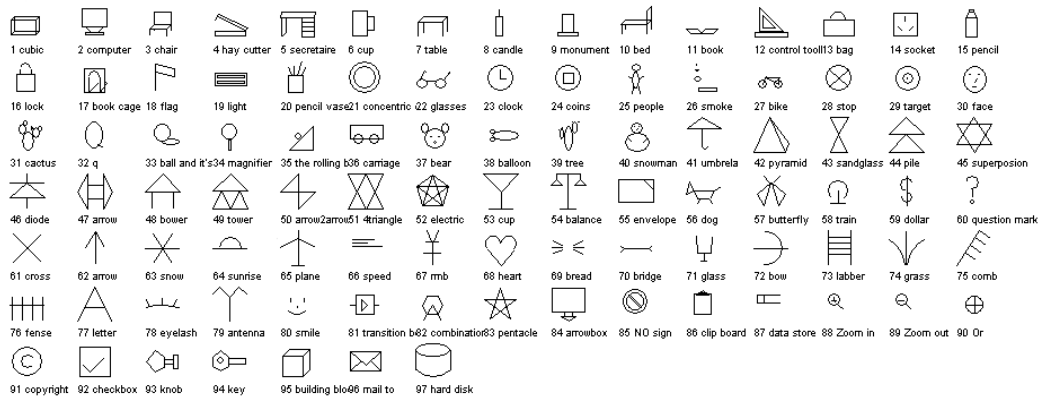
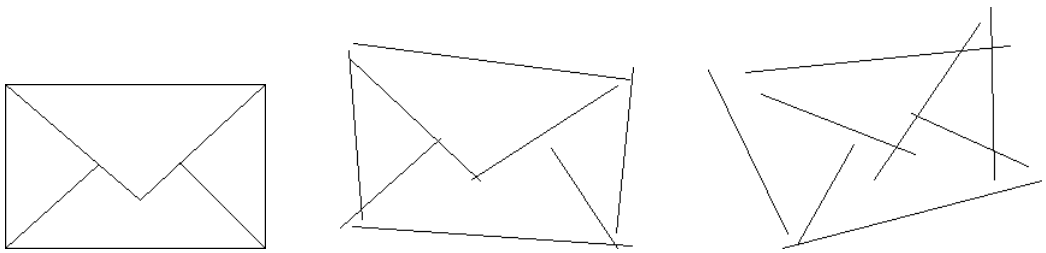


Figure A.3: The 97 symbols in the *Composite* data set.



(a) (b) (c)

Figure A.4: Examples of symbols from the *Composite* data set: a undeformed symbol (a), a light deformed symbol (b) and a heavily deformed symbol (c).

set was introduced by Xiaogang et al. [50]. Due to the unavailability of the set in its original form, the set was replicated through a process of vectorization of the images in the electronic version of the article [50]. The corners present in each symbol were also identified. The set is shown in Figure A.3. Besides the original set extracted from the document, the same symbols were also artificially perturbed. As described in [50], the symbols are artificially perturbed by random scaling, rotating and horizontal/vertical shifting symbol components (primitives) both individually and as a whole, with the possible ranges of the random values based on a constant value τ_C. Two different levels of deformation are used, thus, the set turns out to be composed of the already described 97 regular symbols, plus 97 lightly deformed symbols ($\tau_C = 0.1$) and further 97 heavily deformed symbols ($\tau_C = 0.2$). For

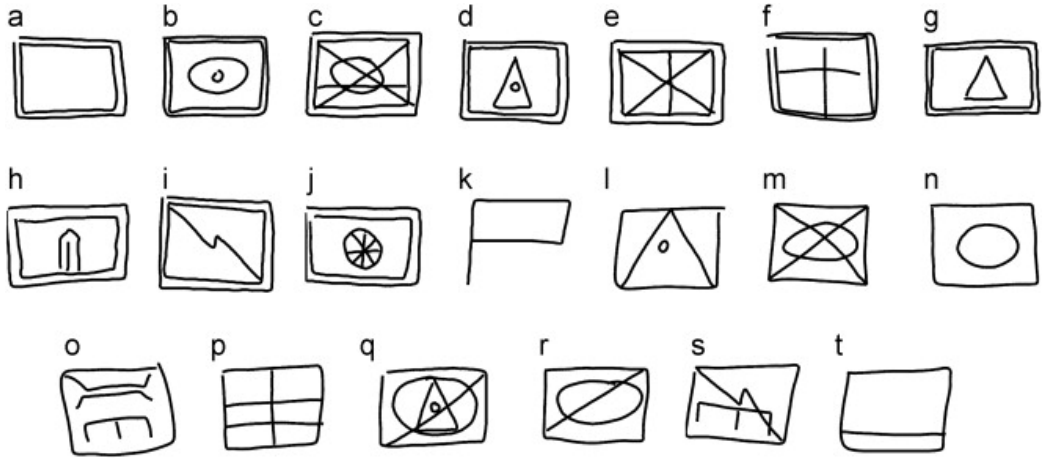


Figure A.5: A sample symbol from each class in the COAD data set.

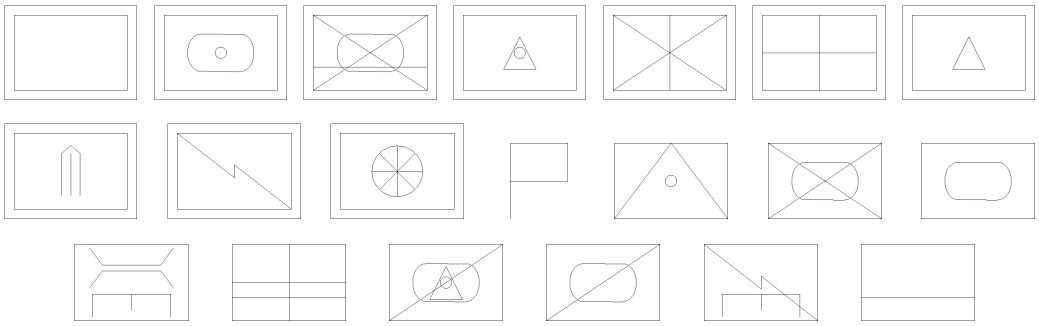


Figure A.6: The 20 template symbols from the COAD symbol set.

a detailed description of the perturbation algorithm, the reader should refer to [50]. Figure A.4 shows the example of a symbol in its undeformed (a) and perturbed versions (b) and (c).

A.4 COAD

The symbols in the COAD data set are a subset of the symbols used in the domain of *Military Course of Action Diagrams* [56], which defines a large set of different symbols and their very many variants used to depict battle scenarios. The COAD data set was introduced by Tirkaz et al. [49]. The total number of drawn symbols is 620 (drawn by 8 users), belonging to 20 different classes. Some samples of the drawn symbols are shown in Figure

A.5.

Templates representing the 20 classes were extracted from the images in vector graphics contained in the original source document about the *Military Course of Action Diagrams* symbols [56] (see Figure A.6).

A.5 COAD2

Since the COAD data set contains a relatively small number of different classes, a larger data set containing symbols from the same domain was created and named COAD2.

To build this data set, as an initial step, 113 template symbols were extracted from the images in vector graphics contained in [56]. The number of primitives of the templates ranges from 2 for the simplest ones to 19 of the most complex one, with an average value of 5.9 (s.d. = 2.9) primitives. The whole set of templates with the associated identifier is shown in Figure A.7.

Then, the set of hand drawn symbols was gathered. The symbols were drawn by 8 users: unpaid adult volunteers; 7 male, 1 female; age ranging from 23 to 48 ($\mu = 33.1$, $\sigma = 8.7$). Each user was asked to hand draw all of the 113 templates 5 times each, trying to balance speed and accuracy of drawing. In all, the set contains $8 \times 5 \times 113 = 4520$ sketched symbols. Figure A.8 shows some examples of users' hand drawn symbols. All the data were drawn through a *SMART Podium ID250* Interactive Pen Display (with a pen report rate of 100 points per second) connected to a *Dell Precision T5400* workstation equipped with an *Intel Xeon* CPU at 2.50 GHz running *Microsoft Windows XP* operating system and the *Java Run-Time Environment 6*.

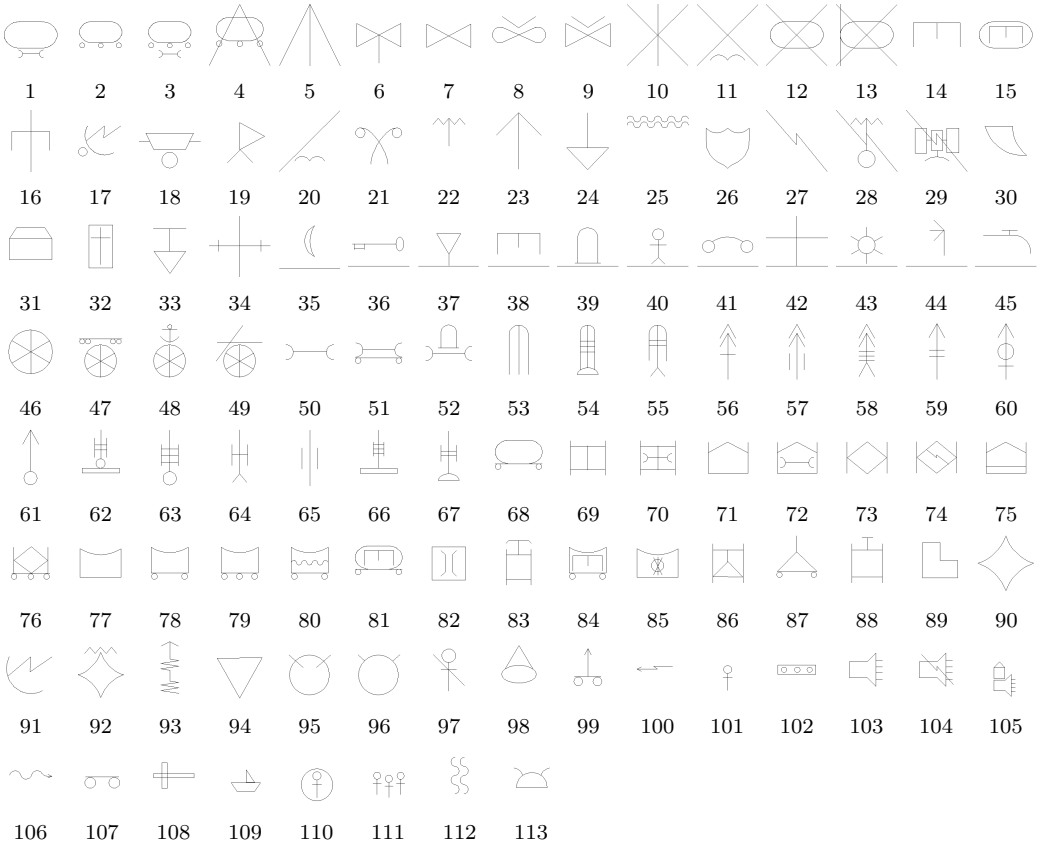


Figure A.7: The 113 template symbols from the COAD2 symbol set.

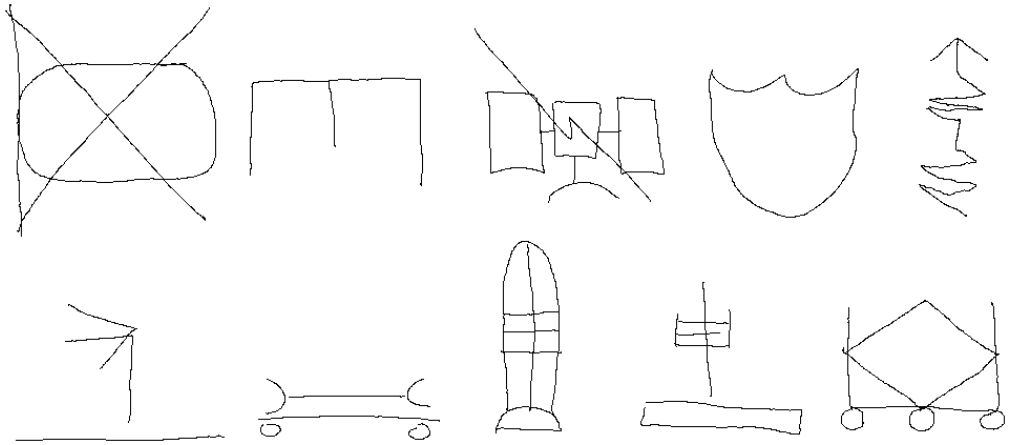


Figure A.8: Examples from the hand drawn symbols in the COAD2 data set.

Bibliography

[1] T. Y. Ouyang and R. Davis, "Chemink: a natural real-time recognition system for chemical drawings," in *Proceedings of the 16th international conference on Intelligent user interfaces*, IUI '11, (New York, NY, USA), pp. 267–276, ACM, 2011.

[2] S. Belongie, J. Malik, and J. Puzicha, "Shape matching and object recognition using shape contexts," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, pp. 509–522, April 2002.

[3] W.-H. Tsai and K.-S. Fu, "Error-correcting isomorphisms of attributed relational graphs for pattern analysis," *IEEE Trans. Systems Man Cyber.*, vol. 9, pp. 757–768, dec. 1979.

[4] J. O. Wobbrock, B. A. Myers, and D. H. Chau, "In-stroke word completion," in *Proceedings of the 19th annual ACM symposium on User interface software and technology*, UIST '06, (New York, NY, USA), pp. 333–336, ACM, 2006.

[5] Google Instant. http://www.google.it/instant/, 2012.

[6] R. Robbes and M. Lanza, "How program history can improve code completion," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pp. 317 –326, sept. 2008.

[7] H. Bast and I. Weber, "The CompleteSearch Engine: Interactive, Efficient, and Towards IR & DB integration," *CIDR*, 2007.

[8] F. Brieler and M. Minas, "Ambiguity resolution for sketched diagrams by syntax analysis based on graph grammars," *ECEASST*, vol. 10, 2008.

[9] P. Rodgers, L. Zhang, and H. Purchase, "Wellformedness Properties in Euler Diagrams: Which Should Be Used?," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, pp. 1089–1100, July 2012.

[10] G. Costagliola, M. De Rosa, and V. Fuccella, "Improving Shape Context Matching for the Recognition of Sketched Symbols.," in *DMS*, pp. 289–294, Knowledge Systems Institute, 2011.

[11] G. Costagliola, M. De Rosa, and V. Fuccella, "Recognition and autocompletion of partially drawn symbols by using polar histograms as spatial relation descriptors," *Computers & Graphics*, vol. 39, no. 0, pp. 101–116, 2014.

[12] G. Costagliola, M. De Rosa, and V. Fuccella, "Investigating human performance in hand-drawn symbol autocompletion," in *Systems, Man, and Cybernetics (SMC), 2013 IEEE International Conference on*, pp. 279–284, 2013.

[13] M. De Rosa, "On the auto-completion of hand drawn symbols," in *Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on*, pp. 223–224, 2012.

[14] G. Costagliola, M. De Rosa, and V. Fuccella, "Identifying attachment areas on sketched symbols," in *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*, pp. 83–86, 2011.

[15] G. Costagliola, M. De Rosa, A. Fish, V. Fuccella, and R. Saleh, "Curve-based diagram specification and construction," in *Visual Languages and Human-Centric Computing (VL/HCC), 2013 IEEE Symposium on*, pp. 39–42, 2013.

[16] P. Bottoni, G. Costagliola, M. De Rosa, A. Fish, and V. Fuccella, "Euler diagram codes: interpretation and generation," in *Proc. VINCI 2013*, ACM, 2013.

[17] C.-H. Teh and R. Chin, "On the detection of dominant points on digital curves," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 11, pp. 859–872, Aug 1989.

[18] C. F. Herot, "Graphical input through machine recognition of sketches," in *Proceedings of the 3rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '76, (New York, NY, USA), pp. 97–102, ACM, 1976.

[19] T. M. Sezgin, T. Stahovich, and R. Davis, "Sketch based interfaces: Early processing for sketch understanding," in *Proceedings of the 2001 Workshop on Perceptive User Interfaces*, PUI '01, (New York, NY, USA), pp. 1–8, ACM, 2001.

[20] A. Wolin, B. Eoff, and T. Hammond, "Shortstraw: A simple and effective corner finder for polylines," in *EUROGRAPHICS Workshop on Sketch-Based Interfaces and Modeling*, Eurographics Association, 2008.

[21] Y. Xiong and J. J. J. LaViola, "A shortstraw-based algorithm for corner finding in sketch-based interfaces," *Computers & Graphics*, vol. 34, no. 5, pp. 513 – 527, 2010.

[22] J. Herold and T. F. Stahovich, "A machine learning approach to automatic stroke segmentation," *Computers & Graphics*, vol. 38, no. 0, pp. 357 – 364, 2014.

[23] R. S. Tumen and T. M. Sezgin, "Dpfrag: Trainable stroke fragmentation based on dynamic programming," *IEEE Computer Graphics and Applications*, vol. 33, no. 5, pp. 59–67, 2013.

[24] J. Herold and T. F. Stahovich, "Speedseg: A technique for segmenting pen strokes using pen speed," *Computers & Graphics*, vol. 35, no. 2, pp. 250–264, 2011.

[25] F. Albert, D. Fernández-Pacheco, and N. Aleixos, "New method to find corner and tangent vertices in sketches using parametric cubic curves

approximation," *Pattern Recognition*, vol. 46, no. 5, pp. 1433 – 1448, 2013.

[26] W. Zhang, L. Wenyin, and K. Zhang, "Symbol recognition with kernel density matching," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 28, pp. 2020–2024, dec. 2006.

[27] D. Rubine, "Specifying gestures by example," *SIGGRAPH Comput. Graph.*, vol. 25, pp. 329–337, July 1991.

[28] P.-O. Kristensson and S. Zhai, "Shark2: a large vocabulary shorthand writing system for pen-based computers," in *Proceedings of the 17th annual ACM symposium on User interface software and technology*, UIST '04, (New York, NY, USA), pp. 43–52, ACM, 2004.

[29] C. C. Tappert, "Cursive script recognition by elastic matching," *IBM J. Res. Dev.*, vol. 26, pp. 765–771, November 1982.

[30] J. O. Wobbrock, A. D. Wilson, and Y. Li, "Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes," in *Proceedings of the 20th annual ACM symposium on User interface software and technology*, UIST '07, (New York, NY, USA), pp. 159–168, ACM, 2007.

[31] M. Ahmed and R. K. Ward, "An expert system for general symbol recognition," *Pattern Recognition*, vol. 33, no. 12, pp. 1975–1988, 2000.

[32] Y. Lin, L. Wenyin, and C. Jiang, "A structural approach to recognizing incomplete graphic objects," in *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, vol. 1, pp. 371–375 Vol.1, aug. 2004.

[33] J. Llados, E. Marti, and J. Villanueva, "Symbol recognition by error-tolerant subgraph matching between region adjacency graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 23, pp. 1137–1143, Oct. 2001.

[34] W. Lee, L. Burak Kara, and T. F. Stahovich, "An efficient graph-based recognizer for hand-drawn symbols," *Computers & Graphics*, vol. 31, pp. 554–567, August 2007.

[35] L. Gennari, L. B. Kara, T. F. Stahovich, and K. Shimada, "Combining geometry and domain knowledge to interpret hand-drawn diagrams," *Computers & Graphics*, vol. 29, no. 4, pp. 547–562, 2005.

[36] T. Y. Ouyang and R. Davis, "A visual approach to sketched symbol recognition," in *Proceedings of the 21st international jont conference on Artifical intelligence*, (San Francisco, CA, USA), pp. 1463–1468, Morgan Kaufmann Publishers Inc., 2009.

[37] D. Willems, R. Niels, M. van Gerven, and L. Vuurpijl, "Iconic and multi-stroke gesture recognition," *Pattern Recognition*, vol. 42, no. 12, pp. 3303–3312, 2009. New Frontiers in Handwriting Recognition.

[38] H. Hse and A. Newton, "Sketched symbol recognition using zernike moments," in *Proceedings of the 17th International Conference on Pattern Recognition, 2004. ICPR 2004.*, vol. 1, pp. 367–370 Vol.1, aug. 2004.

[39] T. M. Sezgin and R. Davis, "HMM-based efficient sketch recognition," in *Proceedings of the 10th international conference on Intelligent user interfaces*, IUI '05, (New York, NY, USA), pp. 281–283, ACM, 2005.

[40] L. B. Kara and T. F. Stahovich, "An image-based, trainable symbol recognizer for hand-drawn sketches," *Computers & Graphics*, vol. 29, pp. 501–517, August 2005.

[41] M. Oltmans, *Envisioning Sketch Recognition: A Local Feature Bases Approach to Recognizing Informal Sketches*. PhD thesis, Massachussets Institute of Technology, May 2007.

[42] R.-D. Vatavu, L. Anthony, and J. O. Wobbrock, "Gestures as point clouds: a $P recognizer for user interface prototypes," in *Proceedings of the 14th ACM international conference on Multimodal interaction*, ICMI '12, (New York, NY, USA), pp. 273–280, ACM, 2012.

[43] C. Alvarado and R. Davis, "Sketchread: a multi-domain sketch recognition engine," in *Proceedings of the 17th annual ACM symposium on User interface software and technology*, UIST '04, (New York, NY, USA), pp. 23–32, ACM, 2004.

[44] G. Casella, V. Deufemia, V. Mascardi, G. Costagliola, and M. Martelli, "An agent-based framework for sketched symbol interpretation," *Journal of Visual Languages & Computing*, vol. 19, no. 2, pp. 225–257, 2008.

[45] M. Fonseca and J. Jorge, "Using fuzzy logic to recognize geometric shapes interactively," in *The Ninth IEEE International Conference on Fuzzy Systems, 2000. FUZZ IEEE 2000.*, vol. 1, pp. 291–296 vol.1, May 2000.

[46] T. Hammond and R. Davis, "Ladder, a sketching language for user interface developers," in *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, (New York, NY, USA), ACM, 2007.

[47] A. Coyette, S. Schimke, J. Vanderdonckt, and C. Vielhauer, "Trainable sketch recognizer for graphical user interface design," in *Proceedings of the 11th IFIP TC 13 international conference on Human-computer interaction*, INTERACT'07, (Berlin, Heidelberg), pp. 124–135, Springer-Verlag, 2007.

[48] T. Hammond and B. Paulson, "Recognizing sketched multistroke primitives," *ACM Trans. Interact. Intell. Syst.*, vol. 1, pp. 4:1–4:34, Oct. 2011.

[49] C. Tirkaz, B. Yanikoglu, and T. M. Sezgin, "Sketched symbol recognition with auto-completion," *Pattern Recognition*, vol. 45, no. 11, pp. 3926–3937, 2012.

[50] X. Xiaogang, S. Zhengxing, P. Binbin, J. Xiangyu, and L. Wenyin, "An online composite graphics recognition approach based on matching of spatial relation graphs," *Int. J. Doc. Anal. Recognit.*, vol. 7, pp. 44–55, March 2004.

[51] J. Mas, G. Sanchez, J. Llados, and B. Lamiroy, "An incremental on-line parsing algorithm for recognizing sketching diagrams," in *Proceedings of the Ninth International Conference on Document Analysis and Recognition - Volume 01*, (Washington, DC, USA), pp. 452–456, IEEE Computer Society, 2007.

[52] O. Bau and W. E. Mackay, "OctoPocus: a dynamic guide for learning gesture-based command sets," in *Proc. of UIST*, (New York, NY, USA), pp. 37–46, ACM, 2008.

[53] M. Bennett, K. McCarthy, S. O'Modhrain, and B. Smyth, "Simpleflow: enhancing gestural interaction with gesture prediction, abbreviation and autocompletion," in *Proc. of INTERACT'11 - Volume Part I*, (Berlin, Heidelberg), pp. 591–608, Springer-Verlag, 2011.

[54] G. Lucchese, M. Field, J. Ho, R. Gutierrez-Osuna, and T. Hammond, "GestureCommander: continuous touch-based gesture prediction," in *CHI '12 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '12, (New York, NY, USA), pp. 1925–1930, ACM, 2012.

[55] T. Hammond, D. Logsdon, B. Paulson, J. Johnston, J. Peschel, A. Wolin, and P. Taele, "A sketch recognition system for recognizing free-hand course of action diagrams," in *Proceedings of the Twenty-Second Conference on Innovative Applications of Artificial Intelligence*, AAAI, July 2010.

[56] T. D.U., "Commented APP-6A - Military symbols for land based systems," 2005.

[57] T. Hammond and R. Davis, "Tahuti: a geometrical sketch recognition system for uml class diagrams," in *ACM SIGGRAPH 2006 courses*, SIGGRAPH '06, (New York, NY, USA), ACM, 2006.

[58] P. Haddawy, M. N. Dailey, P. Kaewruen, N. Sarakhette, and L. H. Hai, "Anatomical sketch understanding: Recognizing explicit and implicit structure," *Artificial Intelligence in Medicine*, vol. 39, no. 2, pp. 165–177, 2007.

[59] D. Blostein and L. Haken, "Using diagram generation software to improve diagram recognition: A case study of music notation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 21, pp. 1121–1136, November 1999.

[60] A. Delaney, B. Plimmer, G. Stapleton, and P. Rodgers, "Recognising sketches of Euler diagrams drawn with ellipses," in *Proc. VLC 2010*, pp. 305–310, Knowledge Systems Institute, 2010.

[61] M. Wang, B. Plimmer, P. Schmieder, G. Stapleton, P. Rodgers, and A. Delaney, "Sketchset: Creating Euler diagrams using pen or mouse," in *Proc. VL/HCC 2011*, pp. 75–82, 2011.

[62] G. Stapleton, A. Delaney, P. Rodgers, and B. Plimmer, "Recognising sketches of Euler diagrams augmented with graphs," in *Proc. VLC 2011*, pp. 279–284, Knowledge Systems Institute, 2011.

[63] T. Hammond, B. Eoff, B. Paulson, A. Wolin, K. Dahmen, J. Johnston, and P. Rajan, "Free-sketch recognition: Putting the chi in sketching," in *CHI '08 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '08, (New York, NY, USA), pp. 3027–3032, ACM, 2008.

[64] T. F. Stahovich, "Segmentation of pen strokes using pen speed," in *AAAI Fall Symposium Series*, pp. 21–24, 2004.

[65] D. H. Kim and M.-J. Kim, "A curvature estimation for pen input segmentation in sketch-based modeling," *Computer-Aided Design*, vol. 38, no. 3, pp. 238 – 248, 2006.

[66] B. Paulson and T. Hammond, "Paleosketch: accurate primitive sketch recognition and beautification," in *Proceedings of the 13th international conference on Intelligent user interfaces*, IUI '08, (New York, NY, USA), pp. 1–10, ACM, 2008.

[67] R. Haddad and A. Akansu, "A class of fast Gaussian binomial filters for speech and image processing," *Signal Processing, IEEE Transactions on*, vol. 39, pp. 723–727, Mar 1991.

[68] B. Leo, "Random forests," *Machine Learning*, vol. 45, pp. 5–32, dec. 2001.

[69] A. Liaw and M. Wiener, "Classification and regression by randomForest," *R News*, vol. 2, no. 3, pp. 18–22, 2002.

[70] R. Niels, D. Willems, and L. Vuurpijl, "The nicicon database of handwritten icons," 2008.

[71] M. Frenkel and R. Basri, "Curve matching using the fast marching method," in *In EMMCVPR*, pp. 35–51, 2003.

[72] A. Frome, D. Huber, R. Kolluri, T. Bülow, and J. Malik, "Recognizing objects in range data using regional point descriptors," in *EUROPEAN CONFERENCE ON COMPUTER VISION*, pp. 224–237, 2004.

[73] G. Mori and J. Malik, "Estimating human body configurations using shape context matching," 2002.

[74] P. Cohen, L. Chen, J. Clow, M. Johnston, D. Mcgee, J. Pittman, and I. Smith, "Quickset: A multimodal interface for distributed interactive simulation," in *Proceedings of the UIST'96 demonstration*, pp. 217–24, 2003.

[75] K. D. Forbus, J. Usher, and V. Chapman, "Sketching for military courses of action diagrams," in *Proceedings of the 8th international conference on Intelligent user interfaces*, IUI '03, (New York, NY, USA), pp. 61–68, ACM, 2003.

[76] W. Song, A. M. Finch, K. Tanaka-Ishii, and E. Sumita, "picoTrans: an icon-driven user interface for machine translation on mobile devices," in *Proceedings of the 16th international conference on Intelligent user interfaces*, IUI '11, (New York, NY, USA), pp. 23–32, ACM, 2011.

[77] Y. Liu and Q. Wang, "Chinese pinyin phrasal input on mobile phone: usability and developing trends," in *Proceedings of the 4th international conference on mobile technology, applications, and systems and the 1st*

international symposium on Computer human interaction in mobile technology, Mobility '07, (New York, NY, USA), pp. 540–546, ACM, 2007.

[78] T. K. Ho, "Random decision forests," in *Document Analysis and Recognition, 1995., Proceedings of the Third International Conference on*, vol. 1, pp. 278–282 vol.1, 1995.

[79] G. Costagliola, A. De Lucia, S. Orefice, and G. Polese, "A classification framework to support the design of visual languages," *Journal of Visual Languages & Computing*, vol. 13, no. 6, pp. 573 – 600, 2002.

[80] G. Costagliola, G. Tortora, S. Orefice, and A. De Lucia, "Automatic generation of visual programming environments," *Computer*, vol. 28, pp. 56–66, March 1995.

[81] M. Minas, "Visualdiagen - a tool for visually specifying and generating visual editors," in *Applications of Graph Transformations with Industrial Relevance* (J. L. Pfaltz, M. Nagl, and B. Böhlen, eds.), vol. 3062 of *Lecture Notes in Computer Science*, pp. 398–412, Springer Berlin / Heidelberg, 2004.

[82] J. Howse, G. Stapleton, and J. Taylor, "Spider diagrams," *LMS Journal of Computation and Mathematics*, vol. 8, pp. 145–194, 2005.

[83] A. Fish, J. Flower, and J. Howse, "The semantics of augmented constraint diagrams," *Journal of Visual Languages & Computing*, vol. 16, no. 6, pp. 541–573, 2005.

[84] H. Kestler, A. Müller, T. Gress, and M. Buchholz, "Generalized venn diagrams: a new method of visualizing complex genetic set relations," *Bioinformatics*, vol. 21, no. 8, pp. 1592–1595, 2005.

[85] J. Thièvre, M. Viaud, and A. Verroust-Blondet, "Using euler diagrams in traditional library environments," *ENTCS*, vol. 134, pp. 189–202, 2005.

[86] R. De Chiara, U. Erra, and V. Scarano, "VennFS: A Venn diagram file manager," in *Proc. IV 2003*, pp. 120–125, IEEE Computer Society, 2003.

[87] G. Cordasco, R. De Chiara, and A. Fish, "Interactive visual classification with euler diagrams," in *VL/HCC 2009. IEEE Symposium on*, pp. 185–192, IEEE, 2009.

[88] N. Riche and T. Dwyer, "Untangling euler diagrams," *Visualization and Computer Graphics, IEEE Transactions on*, vol. 16, no. 6, pp. 1090–1099, 2010.

[89] C. Collins, G. Penn, and S. Carpendale, "Bubble Sets:revealing set relations with isocontours over existing visualisations," *IEEE Trans Visualisation and Computer Graphics*, vol. 15, no. 6, pp. 1009–1016, 2009.

[90] P. Bottoni, G. Costagliola, and A. Fish, "Euler diagram encodings," in *Proc. Diagrams '12*, 2012.

[91] J. M. Boyer and W. J. Myrvold, "On the cutting edge: Simplified O(n) planarity by edge addition," *Journal of Graph Algorithms and Applications*, vol. 8, no. 3, pp. 241–273, 2004.

[92] F. Bertault, "A Force-Directed Algorithm that Preserves Edge Crossing Properties," in *Graph Drawing* (J. Kratochvíyl, ed.), vol. 1731 of *Lecture Notes in Computer Science*, pp. 351–358, Springer Berlin Heidelberg, 1999.

[93] W. T. Tufte, "How to draw a graph," *Proc Lond Math Soc*, vol. 13, pp. 743–767, 1963.

[94] OGDF. http://www.ogdf.net, 2014.

[95] G. Costagliola, V. Deufemia, and M. Risi, "Using Grammar-Based Recognizers for Symbol Completion in Diagrammatic Sketches," in *Document Analysis and Recognition, 2007. ICDAR 2007. Ninth International Conference on*, vol. 2, pp. 1078–1082, Sept 2007.

[96] S. Yang and Y. Wang, "Rotation invariant shape contexts based on feature-space fourier transformation," in *Proceedings of the Fourth In-*

ternational Conference on Image and Graphics, ICIG '07, (Washington, DC, USA), pp. 575–579, IEEE Computer Society, 2007.