Gestures as Point Clouds: A $P Recognizer for User Interface Prototypes

Radu-Daniel Vatavu
University Stefan cel Mare of Suceava
Suceava 720229, Romania
vatavu@eed.usv.ro

Lisa Anthony
UMBC Information Systems
1000 Hilltop Circle
Baltimore MD 21250
lanthony@umbc.edu

Jacob O. Wobbrock
Information School | DUB Group
University of Washington
Seattle, WA 98195-2840 USA
wobbrock@uw.edu

ABSTRACT

Rapid prototyping of gesture interaction for emerging touch platforms requires that developers have access to fast, simple, and accurate gesture recognition approaches. The $-family of recognizers ($1, $N) addresses this need, but the current most advanced of these, $N-Protractor, has significant memory and execution costs due to its combinatoric gesture representation approach. We present $P, a new member of the $-family, that remedies this limitation by considering gestures as clouds of points. $P performs similarly to $1 on unistrokes and is superior to $N on multistrokes. Specifically, $P delivers >99% accuracy in user-dependent testing with 5+ training samples per gesture type and stays above 99% for user-independent tests when using data from 10 participants. We provide a pseudocode listing of $P to assist developers in porting it to their specific platform and a "cheat sheet" to aid developers in selecting the best member of the $-family for their specific application needs.

Categories and Subject Descriptors

H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Input devices and strategies*; I.5.2 [**Pattern Recognition**]: Design Methodology—*Classifier design and evaluation*

General Terms

Algorithms, Experimentation, Performance

Keywords

Gesture recognition, point clouds, comparing classifiers, multistrokes, Euclidean, Hausdorff, Hungarian, $P, $1, $N.

1. INTRODUCTION

The currently increasing mainstream use and adoption of touch input devices like the iPad, iPhone, and Microsoft Surface, along with the surge in touch-based app development, fosters a rising need for tools to support development for such platforms. New applications may require gesture recognition tailored to new gestures that are simply not built into existing software.

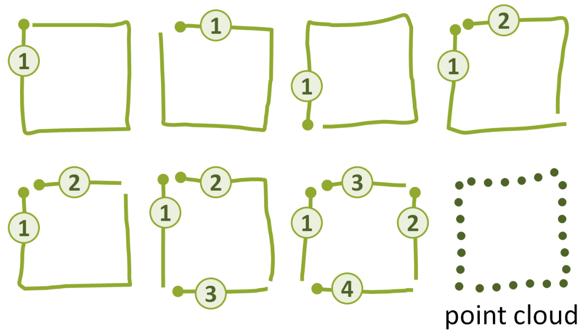

Figure 1: Even a simple square can be drawn using 1, 2, 3, or 4 strokes which can vary in order and direction (with a total of 442 possible cases). However, all the articulation details are ignored when looking at the square as a time-free cloud of points.

State-of-the-art gesture recognition techniques, such as Hidden Markov Models [19], feature-based statistical classifiers [17, 22], or mixture of classifiers [12], typically require significant technical knowledge to understand and develop them for new platforms, or knowledge from other fields such as graph theory [11]. Therefore, our growing body of work has been tackling this problem by proposing low-cost, easy to understand and implement, yet high performing, gesture recognition approaches [2, 3, 23]. These approaches, which we will call the $-family of techniques, involve only simple geometric computations and straightforward internal representations. Furthermore, the algorithms are highly accessible through the publication of pseudocode which developers may use for their own platforms. Indeed, both $1 and $N have experienced swift uptake with implementations available in JavaScript, C#, and Objective-C by third-party developers[1].

Yet the $-family approaches each have limitations. For example, $1 and Protractor only handle unistroke gestures [14, 23]. $N and $N-Protractor have focused on remedying this limitation, adding support for multistrokes [2, 3]. The $N approaches do so by treating multistrokes as unistrokes obtained by connecting individual strokes "in the air" [2], which enables them to use the same matching algorithm as $1. However, as stroke order and direction may differ among users drawing the same symbol (Figure 1), $N needs to generate all possible permutations of a given multistroke [2, 3], which causes an explosion in both memory and execution

[1]See "$1 implementations by others" at $1 homepage, http://depts.washington.edu/aimgroup/proj/dollar/

time for \$N and even for \$N-Protractor, albeit to a lesser extent. For example, a 2-stroke gesture such as an "X" has 8 permutations to represent, and a 4-stroke gesture such as a square has $4! \times 2^4 = 384$ different permutations. To support a cube, a symbol typically drawn with 9 strokes[2], the system must represent $9! \times 2^9 = 185{,}794{,}560$ permutations. This exceedingly high number poses a challenge for modern desktops, let alone mobile platforms. Although \$N uses run-time optimizations that reduce the number of comparisons to stored permutations [2], the cost of storing these permutations is currently its main limiting factor [3].

To address the aforementioned problems, we present our new approach, the \$P recognizer, which avoids the storage complexity of \$N by representing gestures as "clouds of Points" and thus ignoring variable user behavior in terms of stroke order and direction. Just like its predecessors \$1 [23] and \$N [2, 3], \$P yields high accuracy, low complexity, and low barriers to adoption (only 70 lines of code, 50% reusing \$1 code [23]). Experiments showed an average accuracy of 98% for \$P, which outperformed \$N in both user-dependent and user-independent testing[3]. Also, \$P delivered >99% accuracy in user-dependent testing with 5+ training samples per gesture and stayed above 99% for user-independent tests when using data from 10 participants.

The contributions of our work include: (1) a straightforward algorithm called \$P that represents and recognizes stroke gestures as point clouds; (2) an evaluation of \$P showing that it is more accurate and needs considerably less memory than \$N-Protractor [3]; and (3) a pseudocode listing for \$P to enable rapid uptake of this new member of the \$-family of stroke recognizers for user interface prototypes. Finally, to assist developers in selecting the most suitable classifier for their needs (\$1, \$N, \$P), we contribute a "cheat sheet" for the \$-family of recognizers highlighting the main similarities and differences among the family members.

2. RECOGNIZING POINT CLOUDS

Most limitations of the \$N recognizer come from reasoning about gestures in terms of a chronological order of drawn points, which enforces a predefined order for strokes and points within each stroke. In consequence, to retain user independence, \$N needs to permute and store gestures by stroke order and direction, which considerably affects the size of the training set [2, 3] and negatively impacts memory usage and system performance. However, such complications no longer apply when reasoning *outside* the gesture timeline. Discarding the timeline makes gestures appear as simple sets without any particular order associated to strokes or points: $\{p_i = (x_i, y_i) \mid i = 1..n\}$. Point p_i does not necessarily follow p_{i-1}, nor does it necessarily precede p_{i+1}. Point p_1 does not mark the starting point of the gesture nor is p_n its endpoint. Instead, gestures are seen as unordered sets, or what we call *clouds*, grouping points together. By adopting this time-free view of gestures, aspects such as the number of strokes, stroke ordering, and stroke direction become irrelevant. To make an analogy, such a diminished representation resembles the input of off-line optical character recognition systems that only use bitmaps delivered by an optical scanner [16].

[2]See http://embedded.eecs.berkeley.edu/research/hhreco/

[3]For conciseness, we use the term \$N to refer to the state of the art \$N, which is \$N-Protractor [3], an improved version over [2].

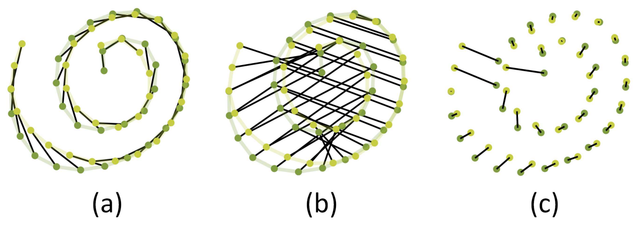

Figure 2: Point alignments for two *spiral* gestures. The time-ordered alignment performed by \$1 and \$N (a) fails when one of the *spirals* is simply drawn backwards (b) which is why \$N creates permutations of all stroke orders and directions in its training set. The time-free alignment of point clouds (c) ignores such execution details.

In order to better highlight the advantages of discarding the execution timeline, Figure 1 illustrates some of the many ways to draw a square. Drawing a square could be done via 1, 2, 3, or 4-stroke gestures with many variations for the individual strokes in terms of ordering and direction. Instead, when looking at the same gesture as a simple cloud of points with no time labels, all such execution details are hidden away into the final result. Indeed, just by looking at the point cloud of Figure 1, the reader cannot tell whether this is a unistroke or a multistroke gesture; whether it is composed of 2 or 3 strokes; or what the order and direction of strokes might be. We argue that the answers to these questions are not essential for recognizing the gesture, and in many cases they even complicate the structure of the recognizer [2] rather than helping with classification.

Once the gesture execution timeline has been discarded, the number of strokes, stroke ordering, and stroke direction become irrelevant. The task of the recognizer remains to match the point cloud of the candidate gesture (C) to the point cloud of each template (T) in the training set and compute a matching distance. In the tradition of the Nearest-Neighbor approach, the template located at the smallest distance from C delivers the classification result.

We define the matching between two point clouds C and T as a function $\mathcal{M}$ that associates each point $C_i \in C$ with exactly one point $T_j \in T$, $T_j = \mathcal{M}(C_i)$. If C and T have been both resampled[4] into the same number of points n, then the matching will also consist of exactly n pairs of points. Inspired by the Euclidean sum of \$1 [23] and the Proportional Shape Distance of SHARK[2] [13], we define the goodness of matching $\mathcal{M}$ as the sum of Euclidean distances for all the pairs of points from $\mathcal{M}$:

$$\sum_{i=1}^{n} \|C_i - T_j\| = \sum_{i=1}^{n} \sqrt{(C_i.x - T_j.x)^2 + (C_i.y - T_j.y)^2} \quad (1)$$

In this equation, j depends on i but, for ease of notation, we only iterate on i and discard additional notation formalisms by simply considering that point C_i from the first cloud was matched to point T_j from the second cloud by some matching algorithm implementing $\mathcal{M}$. Note that when j equals i (i.e., $\mathcal{M} = \{(C_i, T_i) \mid i = 1..n\}$), the formula becomes the Euclidean sum of the \$1 recognizer. Figure 2 illustrates the difference between time-ordered (a and b) and time-free point alignments (c).

[4]Resampling is a common practice in gesture preprocessing [2, 14, 20, 23] in order to uniformize input data for classifiers.

2.1 The Hungarian Gesture Recognizer

With these considerations, a point cloud recognizer needs to search for the minimum matching distance ("goodness") between C and T from all the $n!$ possible alignments. Actually, this represents a well-known problem in combinatorial optimization which is called the *Assignment Problem* [6]:

> There are n men and n jobs for which we know the cost of assigning man i to job j. Assign all men to jobs so that each man gets only one job and the total cost of assignments is minimum (p. 5).

The Assignment Problem is one that has been solved in graph theory [9]. Our matching problem can be easily translated into the formalism of graphs by constructing an undirected graph with $2n$ vertexes corresponding to the points of C and T and edges weighted by the Euclidean distances between these points. With this correspondence, the recognizer simply needs to solve the assignment problem on this constructed graph. In fact, the graph is a special case in which vertices are split into two sets so that edges only exist *between* and not *inside* the sets, which is known as a bipartite graph [7] (p. 1083). If we were to adhere strictly to graph theory, we would refer to our problem as finding the Minimum Weighted Matching in a bipartite graph, which is classically solved using the Hungarian algorithm [15].

Now that we have established that the problem of finding the best match between two point clouds can be represented as a problem from graph theory, we investigate the success of this approach in recognizing multistrokes. As we will show, the Hungarian algorithm delivers the ideal minimum-cost matching performance, but at a high time complexity ($O(n^3)$). We discuss approximation methods to reduce complexity of this approach, selecting the best one as our \$P recognizer. For the rest of the paper, we refer to the Hungarian algorithm within the Nearest-Neighbor classification approach as the HUNGARIAN recognizer. As we only use the Hungarian algorithm as a reference for our \$P recognizer, we refer the reader to [15] (p. 248) for more details.

2.2 Analysis of the Hungarian Recognizer

We first conducted a recognition experiment in order to understand the performance of the HUNGARIAN recognizer and to compare it with its main competitor, \$N [2, 3]. Also, as the Euclidean [13], angular cosine (e.g., Protractor) [14], and dynamic time warping (DTW) [23] distances have been successfully used for gesture recognition before, we included these metrics into our experiment as well. The experiment was conducted in accordance with the practices of the domain [2, 14, 20, 23]. We ran all 5 recognizers on the \$N multistroke set [2] which contains a total of 3,200 samples = 16 (gestures) × 20 (participants) × 10 (repetitions)[5].

Our experiment extends previous studies [2, 23] by reporting both user-dependent and user-independent recognition results. For the **user-dependent scenario**, recognition rates were computed individually for each participant. For each gesture type, T samples were randomly selected for training and 1 extra sample was additionally chosen for testing. This process was repeated 100 times for each value of T and results were averaged into a recognition rate per participant. The number of training samples per gesture T varied from 1 to 9. In the **user-independent scenario**, data from P participants was used for training while 1 additional participant

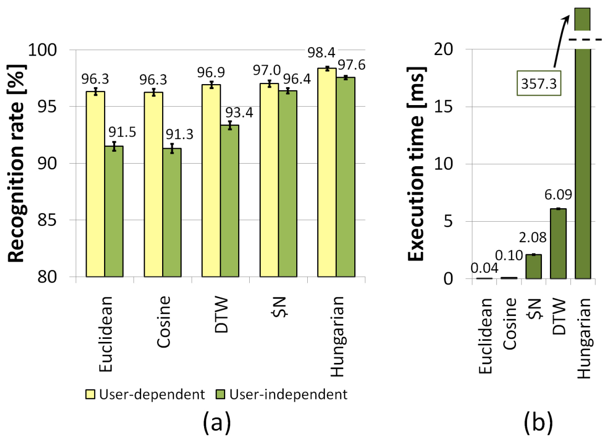

Figure 3: Performance of the Hungarian vs. competitor recognizers: (a) recognition rate; (b) execution time (for a training set of 16 gestures × 3 samples). Error bars show 95% CI.

User-dependent	User-independent
.	19 values for P ×
20 participants ×	100 repetitions for each P[a]×
9 values for T[b] ×	9 values for T ×
100 repetitions for each T ×	100 repetitions for each T ×
5 recognizers ×	5 recognizers ×
16 gestures	16 gestures
≈ 1.5×10^6 recognition tests	≈ 1.4×10^8 recognition tests

[a] P represents the number of training participants (only applies to the user-independent scenario).
[b] T is the number of training samples per gesture type.

Table 1: Controlled variables for the Hungarian recognition experiment.

was randomly selected for testing. T samples were randomly selected for each gesture type from each training participant. One sample for each gesture type was selected from the testing participant and submitted for classification. The process was repeated 100 times for each P and 100 times for each T with $100 \times 100 = 10,000$ classification results averaging into a recognition rate depending on P and T. P varied from 1 to 19 and T from 1 to 9. As \$N uses $n = 96$ sampling points [3], all recognizers used this resolution. We report recognition results from 1.5 $\times 10^6$ recognition tests for the user-dependent and 1.4 $\times 10^8$ tests for the user-independent scenario (see Table 1).

The HUNGARIAN recognizer delivered the best average recognition performance[6] for both user-dependent (98.4%) and user-independent (97.6%) testing. \$N came next with 97.7% and 96.4% accuracy, followed by DTW with 96.9% and 93.4% for the two testing scenarios (Figure 3a). A Friedman test [10] showed a significant difference between the performance of all 5 recognizers for both user-dependent ($\chi^2(4) = 342.879$, $p < .001$) and user-indepedent testing ($\chi^2(4) = 3478.055$, $p < .001$). Post-hoc Wilcoxon signed-rank tests showed the HUNGARIAN recognizer outperforming \$N significantly ($Z = -11.367$, $p < .001$) for both testing scenarios (with effect sizes $r < .15$).

[5]http://depts.washington.edu/aimgroup/proj/dollar/ndollar.html

[6]Averaged across all $T = 1..9$ and $P = 1..19$.

3. IN SEARCH OF A $-LIKE RECOGNIZER FOR POINT CLOUDS

Despite its very good performance on the multistroke gesture set, the HUNGARIAN recognizer hardly fits into the $-family paradigm of delivering simple and easy-to-use recognition tools for user interface prototypers. Quite the opposite of a $-like recognizer [2, 23], the HUNGARIAN algorithm employs advanced concepts such as feasible labelings, equality graphs, alternating paths, and alternating trees [15] (p. 248), along with graph representation and traversal techniques [7]. Even more, its $O(n^3)$ complexity is one order higher than DTW's $O(n^2)$ and two orders higher than the complexity of EUCLIDEAN and COSINE recognizers (both running in $O(n)$ time). As a result, running the HUNGARIAN algorithm to classify a candidate gesture against the multistroke set (16 gestures × 3 samples loaded) took 357 ms to complete (see Figure 3b) on an Intel® Core™ i5 2.27 GHz processor. This was 60 times slower than DTW (6.1 ms) and 3000 times slower than COSINE and EUCLIDEAN (both under 0.1 ms). Even the non-optimized $N-Protractor took only 2.0 ms to execute (due to the fast COSINE it employs [3, 14] and the number of permutations being low for this set). Therefore, a low-cost alternative to the HUNGARIAN algorithm is necessary for qualifying the point cloud matching approach as a $-recognizer.

Luckily, because the assignment problem is an important one with many implications [6], many approximation algorithms exist. They either rely on complex data structures [1] or on the use of greedy heuristics [4]. The fastest approximation for the Euclidean space is currently an algorithm from Agarwal and Varadarajan [1] with near linear time $O(n^{1+\epsilon})$ with $\epsilon \in (0, 1)$. However, in terms of trying to optimize conceptual complexity, the algorithm is even farther away from the $-paradigm than is the HUNGARIAN recognizer, as it relies on advanced data structures and tight upper bounds, hardly to be understood without considerable effort outside the algorithms research community. Another option is to use simple heuristics [4] that do not provide optimal alignment but are extremely simple to understand and implement. Therefore, such heuristics seem to be more in accordance with the $-paradigm of delivering simple recognizers.

Inspired by Avis [4] and the HUNGARIAN algorithm's inner workings, we implemented the following heuristics for matching clouds (which we named GREEDY-X with X=1..5):

❶ <u>GREEDY-1</u>. All the Euclidean distances between the n points of cloud C and the n points of cloud T are computed and sorted in ascending order. The first n pairs of points that can form a valid match are selected (each point is only used once). The complexity is $O(n^2 \times log(n))$ as the list needs to be sorted (using the quick sort algorithm).

❷ <u>GREEDY-2</u>. For each point in the first cloud (C_i), find the closest point from the second cloud that hasn't been matched yet. Once point C_i is matched, continue with C_{i+1} until all points from C are matched ($i = 1..n-1$). The complexity is $O(n^2)$ as for every point from C a linear search needs to be performed in T.

After having implemented this heuristic, we observed that the result varies with the order in which points C_i are selected from the first cloud. Therefore, we decided to run the algorithm multiple times with different starting points and return the minimum matching of all runs. If C_k is the starting point then:

$$\sum_i \|C_i - T_j\| = \sum_{i=k}^{n} \|C_i - T_j\| + \sum_{i=1}^{k-1} \|C_i - T_j\| \tag{2}$$

where i goes circularly through all points in C. We introduce a parameter ϵ to control the number of runs. If $\epsilon = 0$, GREEDY-2 runs once (for points $C_1, C_2, \ldots C_n$ in this order). If $\epsilon = 1$, the algorithm runs n times (with each point C_i getting the first chance to pair up). If $\epsilon < 1$ the algorithm runs for $n^\epsilon < n$ times. Using this formalism, the complexity of GREEDY-2 is easy to express as $O(n^{2+\epsilon})$ and lies between $O(n^2)$ and $O(n^3)$. As the HUNGARIAN delivers the result in $O(n^3)$, there is no point working with $\epsilon > 1$.

❸ <u>GREEDY-3</u>. Similar to GREEDY-2, except that the order in which points C_i are matched is randomized at each run. GREEDY-3 also needs $O(n^{2+\epsilon})$ time.

❹ <u>GREEDY-4</u>. Presumably difficult points from cloud C are matched first: a point C_i is more difficult to match if its sum of Euclidean distances to all the points in T is larger. This heuristic sorts the points from C in descending order by their total distance to T. Each point in C is then assigned the closest point from T similar to GREEDY-2. GREEDY-4 runs in $O(n^2)$.

❺ <u>GREEDY-5</u>. Similar to GREEDY-2, except the sum of Euclidean distances (eq. 2) contains weights:

$$\sum_i w_i \cdot \|C_i - T_j\| \tag{3}$$

Weights w_i encode the confidence in each pair (C_i, T_j) computed during the greedy run. The first match is weighted with $w_1 = 1.0$ (meaning high confidence) because we trust it: the first point has all the data in order to make a decision for its closest match. As the algorithm progresses, few options remain for the rest of the points from the first cloud when searching their closest pair into the second. Therefore, we can't trust these matches completely so we weight them with confidence values in [0..1]. For example, the last point to be matched has only one option (the last point from the second cloud that has been left unmatched) so the confidence in this alignment should also be small. We adopted a linear weighting scheme in which:

$$w_i = 1 - \frac{i-1}{n} \tag{4}$$

where $i = 1..n$ encodes the current step of the algorithm. GREEDY-5 also runs in $O(n^{2+\epsilon})$ time.

The results of GREEDY-X depend on the direction of matching (e.g., whether cloud C is matched to T or vice versa) so their implementations must return:

$$\min\left(\text{GREEDY-X}(C, T), \text{GREEDY-X}(T, C)\right) \tag{5}$$

This change affects the heuristics execution time but not their algorithmic complexities as discussed above.

Besides these heuristics, we note that Hausdorff distance [18] can also be interpreted as an approximation of the matching of two sets of points. Hausdorff computes the minimum Euclidean distance for each point from the first cloud to all the points in the second cloud and returns the maximum of these minimum distances. The distance is heavily used in computer vision for matching templates [18], and it has also been applied for recognizing hand sketches [12]. However, studies have expressed concerns on Hausdorff being

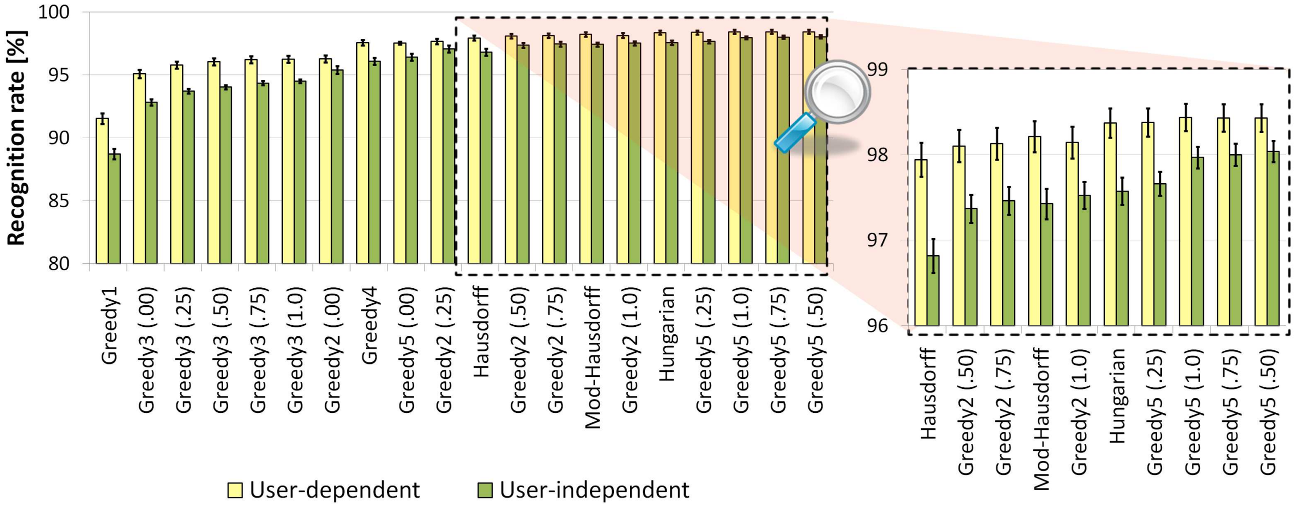

Figure 4: Recognition performance of greedy heuristics vs. the Hungarian. Note: recognizers are ordered on the horizontal axis by their average recognition performance. Error bars represent 95% CI.

too sensitive to outliers [18], which led to a variant called the Modified Hausdorff distance [8]. Due to the relevance of this previous work, we added both Hausdorff and the modified version to our list of heuristics:

❻ HAUSDORFF. Returns the maximum value of the minimum Euclidean distances computed for each point in C and every point from T [18]:

$$\max_{i=1..n} \min_{j=1..n} \|C_i - T_j\| \tag{6}$$

❼ MODIFIED-HAUSDORFF. Returns the average value of the minimum Euclidean distances computed for each point in C and every point from T [8]:

$$\frac{1}{n}\sum_{i=1}^{n} \min_{j=1..n} \|C_i - T_j\| \tag{7}$$

Both HAUSDORFFs are directional in the forms presented, so their implementations also need to consider both matching directions (C is matched to T and vice-versa). HAUSDORFFs are computed with $O(n^2)$ complexity.

3.1 Performance Analysis of Greedy-X

A second experiment was conducted in order to compare the recognition performance of the proposed heuristics. The goal of the experiment was to discover a good \$-like recognizer candidate that would come as close as possible to the performance of the HUNGARIAN algorithm which implements the ideal behavior of the cloud point matching technique. We used the same gesture corpus and experiment design as in the first study. GREEDY-2, GREEDY-3, and GREEDY-5 were tested with $\epsilon \in \{0.00, 0.25, 0.50, 0.75, 1.0\}$ resulting in $5 + 5 + 5 = 15$ different recognizers. HAUSDORFF and MODIFIED-HAUSDORFF were also included in the experiment for comparison purposes. In total, we tested 19 heuristic recognizers (17 GREEDY-X and 2 HAUSDORFFs) and report results from 4.3×10^6 recognition tests for user-dependent and 4.1×10^8 tests for user-independent testing. Again, all gestures were resampled into $n = 96$ points. Figure 4 shows the recognition performance of all the 20 recognizers (the HUNGARIAN included for comparison).

User-dependent results. GREEDY-1 delivered the poorest performance with only 91.6% accuracy. It was followed by GREEDY-3 (all ϵ-versions) with the highest accuracy being 96.3% for $\epsilon = 1.0$. GREEDY-4 achieved 97.6% while almost all GREEDY-2 versions (except $\epsilon = .00$) stayed above 98%. Both HAUSDORFFs performed well with 98.0% accuracy. GREEDY-5 performed the best, staying above the HUNGARIAN (which delivered 98.4%). A Friedman test showed a significant difference between the recognition rates of all 20 recognizers ($\chi^2(19) = 10991.741$, $p < .001$).

GREEDY-2 (98.1% for $\epsilon = 0.5$, 0.75, and 1.0), MODIFIED-HAUSDORFF (98.2%), and GREEDY-5 (98.4% for ϵ=0.25, 0.5, 0.75, and 1.0) were the recognizers that came closest to the performance of HUNGARIAN (98.4%). Figure 4 shows a close up view for these heuristics. Follow-up Wilcoxon signed-rank tests were used to test differences between all these 8 recognizers and the HUNGARIAN (a Bonferroni correction was applied and effects are reported at $0.05/8 = 0.0063$ level of significance). The performance of HUNGARIAN was significantly different from that of GREEDY-2 and MODIFIED-HAUSDORFF (the effect size r stayed below .20). However, there was no significant difference in recognition between HUNGARIAN and the GREEDY-5 metrics ($\epsilon \geq .25$).

User-independent results. Results were similar for the user-independent testing. Again, GREEDY-1 had the worst performance with only 88.8% accuracy and it was followed by GREEDY-3 (<95%). GREEDY-4 achieved 96.1% while GREEDY-2 stayed above 97.1% for $\epsilon > 0$. The HAUSDORFF metrics delivered 96.8% and 97.4% accuracies. GREEDY-5 recognizers (98.4% for $\epsilon \geq 0.25$) stayed above the performance of the HUNGARIAN (which was 97.6%). A Friedman test confirmed a significant difference between all 20 recognizers ($\chi^2(19) = 32709.094$, $p < .001$). The same recognizers that performed well in user-dependent testing were also the ones that came closest to the performance of the HUNGARIAN for the user-independent scenario. GREEDY-2 achieved 97.4% for $\epsilon \geq 0.5$. MODIFIED-HAUSDORFF delivered 97.4%. GREEDY-5 managed to achieve 98% for $\epsilon = 1.0$. Post-hoc Wilcoxon signed-rank tests showed significant differences between all these 8 metrics and the HUNGARIAN (effect sizes r stayed below .20).

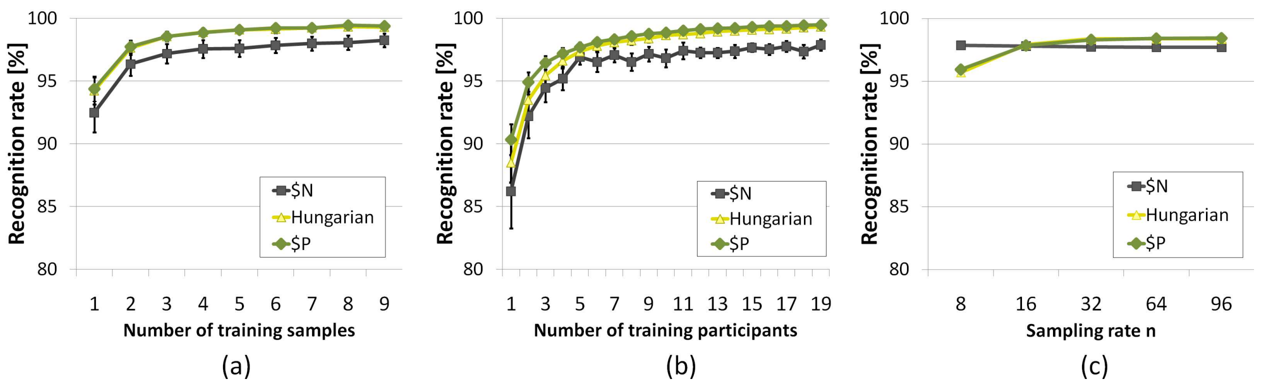

Figure 5: Recognition performance of $P vs. $N [2, 3] and Hungarian [15].

4. THE $P RECOGNIZER

Results obtained place GREEDY-5 as the best candidate for the $P recognizer. Recognition performance of GREEDY-5 running with $\epsilon \geq 0.50$ was not significantly different from that of HUNGARIAN for user-dependent tests (as confirmed by Wilcoxon tests). However, significant differences were detected between the two recognizers for user-independent testing, with GREEDY-5 exhibiting better performance. As the highest accuracy was found for $\epsilon = 0.50$, we denote GREEDY-5 (0.50) as the $P recognizer. Pseudocode for $P is listed in the appendix ($\approx$ 70 lines of code, out of which 50% reuse $1 code). We continue to analyze $P against its two main competitors: $N and HUNGARIAN.

Effect of number of training samples. Figure 5a shows the influence of the amount of training samples on $P's performance for user-dependent testing. $P reached 98.5% accuracy with $T = 3$ training samples per gesture type and stayed above 99% for $T \geq 5$ (with a maximum of 99.4%). The performance of HUNGARIAN was almost identical (the two lines practically overlap in the graph). $N delivered 97.8% performance with $T = 3$ samples and attained a maximum accuracy of 98.7% for $T = 9$.

Effect of number of training participants. For user-independent testing, we are interested in how many participants need to provide data samples for recognizers to deliver high performance for new users. $P achieved 90% recognition with data from just 1 training participant while $N delivered 86% and the HUNGARIAN 88% in the same conditions (see Figure 5b). Again, the performance of $P and HUNGARIAN were closely related, with $P showing slightly yet significantly higher rates. $P reached 97% with data from 4 participants only and stayed above 99% with 10 participants (with a maximum of 99.5%). $N needed data from 7 participants to reach 97% and stayed under 98% even when data from 19 participants was available for training.

Execution time. Recall that, aside from being a complex algorithm, the drawback of HUNGARIAN is large execution times (see Figure 3b). $P has $O(n^{2.5})$ complexity, and classifies a candidate gesture in just 32 ms, under the same conditions of Figure 3b[7]. This is 10 times faster than HUNGARIAN but still 15 times slower than $N. However, these times were measured for a sampling rate of $n = 96$ points as needed by $N to run optimally [2]. Recent work has shown the benefits of downsampling to speed-up execution time and reduce memory constraints [20, 21]. We suspected $P would exhibit similar behavior, and therefore we investigated its performance under various sampling rates. Figure 5c shows results obtained for sampling rate $n \in \{8, 16, 32, 64, 96\}$. $N delivered the same performance under all sampling rates (around 97.7%). $P and HUNGARIAN had a worse start with 96% for $n = 8$ points but delivered 98.3% starting with $n = 32$ points. Wilcoxon tests showed a significant but extremely small effect size $r = .04$ between $n = 32$ and $n = 96$ for $P. Changing n from 96 to 32 reduced the execution time of $P down to 5.5 ms from 32.0 ms.

[7]Classification of a candidate against a training set of 16 gestures with 3 loaded templates per gesture type.

5. DISCUSSION AND FUTURE WORK

What does the $P name mean? We decided to name our new recognizer $P for two reasons. First, we wanted to keep the $ in the name as it has come to denote a low-cost, easy to understand and implement recognizer, while the "P" comes from "Point clouds." Therefore, $P is a point-based recognizer instead of a stroke-based one, as are $1 (one stroke) and $N (N strokes). The second reason is motivated by a simple pun. $P was introduced as a low cost memory alternative to $N. As $N generates all permutations, it relates to NP class problems from complexity theory [7](p. 984). And by removing N from NP (as $P does not use the N strokes at all), we end with a single P in the name and achieve polynomial time complexity.

Performance on single-stroke gestures. Although $P was designed to alleviate $N's dependency on stroke permutations, it is not specific to multistrokes. $P can be run on unistroke gestures as well. Therefore, we tested $P on the 16 gestures of the $1 set [23]. This time, EUCLIDEAN ($1) would be its main competitor. $P achieved 99.3% for user-dependent and 96.6% for user-independent testing, similar to EUCLIDEAN's 99.5% and 97.1% performance.

Memory for storing the training set. As $P does not require permutations, the memory it needs to represent the training set T is linear with the size of the set, $O(T)$. This is similar to $1 and Protractor [14, 23]. In contrast, $N needs $O(T \times S! \times 2^S)$ memory where S represents the number of strokes in a multistroke. As shown in the introduction, the memory required by $N becomes quickly impractical (e.g., 185 million records for storing a 9-stroke cube).

Direction invariance. Due to its point cloud representation, $P is invariant to direction. This delivers freedom to designers and users but also means that clockwise and counterclockwise circles can't be discriminated. Future work can address this by reasoning on direction invariant stroke features derived from curvature [22] (p. 3310).

Criteria	\$1	Protractor	\$N	\$P
❶ Gesture types				
recognizes single strokes	✓	✓	✓	✓
recognizes multistrokes	×	×	✓	✓
is scale-invariant	✓	✓	✓	✓
is rotation-invariant	✓	can be	can be	×
is direction-invariant	×	×	✓	✓
❷ Performance				
user-dependent accuracy (single/multi-strokes)	99.5% / ×	99.4% / ×	98.0% / 97.7%	99.3% / 98.4%
user-independent accuracy (single/multi-strokes)	97.1% / ×	95.9% / ×	95.2% / 96.4%	96.6% / 98.0%
algorithmic complexity	$O(n \cdot T \cdot R)$	$O(n \cdot T)$	$O(n \cdot S! \cdot 2^S \cdot T)$	$O(n^{2.5} \cdot T)$
memory to store the training set	$O(n \cdot T)$	$O(n \cdot T)$	$O(n \cdot S! \cdot 2^S \cdot T)$	$O(n \cdot T)$
❸ Code writing				
needs rotation search (GSS)	yes ✍	no	no	no
needs writing filters to speed-up matching	no	no	yes ✍	no
approx. # of lines of code	100 ✍	50	200 ✍	70

[1] n is the number of sampled points; T = the number of training samples per gesture type; R = the number of iterations required by the Golden Section Search (GSS) algorithm used by \$1 (experimentally set to 10 [23]); S = the number of strokes in a multistroke; $S! \cdot 2^S$ = the number of different permutations of stroke ordering and direction needed by \$N [2].

[2] A ✍ symbol means more coding is required (e.g. GSS, filters, or simply more lines of code).

Table 2: Cheat sheet for the \$-family: \$1 [23], Protractor [14], \$N [2, 3], and \$P.

User behavior in producing multistrokes. An interesting finding was revealed while running the recognition tests on the EUCLIDEAN metric (see Figure 3a). EUCLIDEAN works by performing 1-to-1 associations between points ordered by their timeline. Therefore, users producing multistrokes in many different ways would confuse such a simple recognizer, which should exhibit low accuracy. However, the recognition rate of EUCLIDEAN was 96.3% on user-dependent and 91.5% on user-independent testing. The first value suggests that users are consistent in the way they input multistroke gestures (same order and direction of strokes). The user-independent result suggests that a high degree of consensus may exist between different users for the same gesture. We note for now the interesting aspect of this finding and leave the investigation of such user behaviour for future work.

Faster than $O(n^{2.5})$. In its current form which adheres to the simple implementation requirements of \$-recognizers, \$P needs $O(n^{2.5})$ time to produce a classification result. Although execution times are more than reasonable for a 2 GHz processor (as shown in the previous section), practitioners porting \$P to low resource devices may want faster versions. For such extreme cases, developers can opt for space partitioning structures such as k-d trees which deliver $O(n \times log(n))$ performance, similar to the accelerated Iterative Closest Point (ICP) registration technique [5].

Extension to 3D gestures. Due to its straightforward internal representation of gestures as point clouds, \$P can be easily extended to recognize 3D gestures simply by adding the z dimension. Future work will compare \$P against today's state-of-the-art 3D gesture recognizers.

Multistroke segmentation. Point cloud representations could be exploited to address the multistroke segmentation problem by formulating it as a cost minimization task usually solved with dynamic programming [7] (p. 323).

The designer's \$-family cheat sheet. \$P extends the \$-family [2, 3, 14, 23] by addressing existing limitations. Therefore, we provide practitioners a cheat sheet to inform which recognizer to use in their prototypes. Table 2 compares \$1, Protractor, \$N, and \$P on 3 criteria: gesture types, performance, and amount of code writing required.

6. CONCLUSION

We have demonstrated in this work that point clouds can be useful in reducing the time and space complexity of the \$N recognizer while still retaining a simple algorithm implementable in about 70 lines of code. Our resultant recognizer, \$P, extends the \$-family of recognizers with a low-cost, fast, accurate recognizer for rapidly prototyping interfaces requiring unistroke and multi-stroke recognition. \$P performs similarly to the \$1 recognizer on unistrokes and is superior to the \$N recognizer on multistrokes. It is our hope that \$P will be embraced as \$1 and \$N have been, and will aid in advancing the quality and speed of next-generation user interface prototypes.

7. ACKNOWLEDGMENTS

The authors would like to thank Isaac Simmons for implementation assistance on \$N testing procedures.

8. REFERENCES

[1] Agarwal, P., and Varadarajan, K. A near-linear constant-factor approximation for euclidean bipartite matching? In *SCG '04* (2004), 247–252.

[2] Anthony, L., and Wobbrock, J. O. A lightweight multistroke recognizer for user interface prototypes. In *GI '10* (2010), 245–252.

[3] Anthony, L., and Wobbrock, J. O. \$N-Protractor: A fast and accurate multistroke recognizer. In *GI'2012* (2012), 117–120.

[4] Avis, D. A survey of heuristics for the weighted matching problem. *Networks 13* (1983), 475–493.

[5] Besl, P. J., and McKay, N. D. A method for registration of 3-d shapes. *IEEE TPAMI 14*, 2 (Feb. 1992), 239–256.

[6] Burkard, R., Dell'Amico, M., and Martello, S. *Assignment Problems.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[7] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms.* MIT Press, Cambridge, MA, USA, 2001.

[8] Dubuisson, M.-P., and Jain, A. A modified Hausdorff distance for object matching. In *IAPR'94* (1994), 566–568.

[9] Edmonds, J. Paths, trees, and flowers. *Canad. J. Math. 17* (1965), 449–467.

[10] Friedman, M. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *J. Am. Statist. Assoc. 32*, 200 (1937), 675–701.

[11] Hammond, T., and Paulson, B. Recognizing sketched multistroke primitives. *ACM TIIS 1*, 1 (2011), 4:1–4:34.

[12] Kara, L. B., and Stahovich, T. F. Hierarchical parsing and recognition of hand-sketched diagrams. In *UIST '04* (2004), 13–22.

[13] Kristensson, P.-O., and Zhai, S. SHARK^2: a large vocabulary shorthand writing system for pen-based computers. In *UIST '04* (2004), 43–52.

[14] Li, Y. Protractor: a fast and accurate gesture recognizer. In *CHI '10* (2010), 2169–2172.

[15] Papadimitriou, C. H., and Steiglitz, K. *Combinatorial optimization: algorithms and complexity*. Dover Publications, Mineola, New York, USA, 1998.

[16] Plamondon, R., and Srihari, S. N. On-line and off-line handwriting recognition: A comprehensive survey. *IEEE TPAMI 22*, 1 (Jan. 2000), 63–84.

[17] Rubine, D. Specifying gestures by example. In *SIGGRAPH '91* (1991), 329–337.

[18] Rucklidge, W. *Efficient Visual Recognition Using the Hausdorff Distance*. Springer-Verlag New York, 1996.

[19] Sezgin, T. M., and Davis, R. HMM-based efficient sketch recognition. In *IUI '05* (2005), 281–283.

[20] Vatavu, R.-D. The effect of sampling rate on the performance of template-based gesture recognizers. In *ICMI '11* (2011), 271–278.

[21] Vatavu, R.-D. Small gestures go a long way: how many bits per gesture do recognizers actually need? In *DIS '12* (2012), 328–337.

[22] Willems, D., Niels, R., Gerven, M. v., and Vuurpijl, L. Iconic and multi-stroke gesture recognition. *Pattern Recognition 42*, 12, 3303–3312.

[23] Wobbrock, J. O., Wilson, A. D., and Li, Y. Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. In *UIST '07* (2007), 159–168.

APPENDIX

We provide complete pseudocode for $P. POINT is a structure that exposes x, y, and *strokeId* properties. *strokeId* is the stroke index a point belongs to (1, 2, ...) and is filled by counting pen down/up events. POINTS is a list of points and TEMPLATES a list of POINTS with gesture class data.

```
$P-RECOGNIZER (POINTS points, TEMPLATES templates)
n ← 32
NORMALIZE(points, n)
score ← ∞
for each template in templates do
    NORMALIZE(template, n) // should be pre-processed
    d ← GREEDY-CLOUD-MATCH(points, template, n)
    if score > d then
        score ← d
        result ← template
return ⟨result, score⟩
```

```
GREEDY-CLOUD-MATCH (POINTS points, POINTS template, int n)
$\epsilon$ ← .50
step ← $\lfloor n^{1-\epsilon} \rfloor$
min ← ∞
for i = 0 to n − 1 step step do
    $d_1$ ← CLOUD-DISTANCE(points, template, n, i)
    $d_2$ ← CLOUD-DISTANCE(template, points, n, i)
    min ← MIN(min, $d_1$, $d_2$)
return min
```

```
CLOUD-DISTANCE (POINTS points, POINTS tmpl, int n, int start)
matched ← new bool[n]
sum ← 0
i ← start // start matching with $points_i$
do
    min ← ∞
    for each j such that not matched[j] do
        d ← EUCLIDEAN-DISTANCE($points_i$, $tmpl_j$)
        if d < min then
            min ← d
            index ← j
    matched[index] ← true
    weight ← $1 - ((i - start + n) \text{ MOD } n)/n$
    sum ← $sum + weight \cdot min$
    i ← $(i + 1)$ MOD n
until i == start // all points are processed
return sum
```

```
NORMALIZE (POINTS points, int n)
points ← RESAMPLE(points, n)
SCALE(points)
TRANSLATE-TO-ORIGIN(points, n)
```

The following pseudocode addresses gesture preprocessing (or normalization) which includes resampling, scaling with shape preservation, and translation to origin. The code is similar to $1 [23] and $N [2, 3] and we repeat it here for completeness. We highlight two minor changes only.

```
RESAMPLE (POINTS points, int n)
I ← PATH-LENGTH(points) / (n − 1)
D ← 0
newPoints ← $points_0$
for each $p_i$ in points such that $i \geq 1$ do
    if $p_i$.strokeId == $p_{i-1}$.strokeId then
        d ← EUCLIDEAN-DISTANCE($p_{i-1}$, $p_i$)
        if $(D + d) \geq I$ then
            q.x ← $p_{i-1}$.x + $((I - D)/d) \cdot (p_i.x - p_{i-1}.x)$
            q.y ← $p_{i-1}$.y + $((I - D)/d) \cdot (p_i.y - p_{i-1}.y)$
            APPEND(newPoints, q)
            INSERT(points, i, q) // q will be the next $p_i$
            D ← 0
        else D ← D + d
return newPoints
```

```
PATH-LENGTH (POINTS points)
d ← 0
for each $p_i$ in points such that $i \geq 1$ do
    if $p_i$.strokeId == $p_{i-1}$.strokeId then
        d ← d + EUCLIDEAN-DISTANCE($p_{i-1}$, $p_i$)
return d
```

```
SCALE (POINTS points)
$x_{min}$ ← ∞, $x_{max}$ ← 0, $y_{min}$ ← ∞, $y_{max}$ ← 0
for each p in points do
    $x_{min}$ ← MIN($x_{min}$, p.x)
    $y_{min}$ ← MIN($y_{min}$, p.y)
    $x_{max}$ ← MAX($x_{max}$, p.x)
    $y_{max}$ ← MAX($y_{max}$, p.y)
scale ← MAX($x_{max} - x_{min}$, $y_{max} - y_{min}$)
for each p in points do
    p ← ((p.x − $x_{min}$)/scale, (p.y − $y_{min}$)/scale)
```

```
TRANSLATE-TO-ORIGIN (POINTS points, int n)
c ← (0, 0) // will contain centroid
for each p in points do
    c ← (c.x + p.x, c.y + p.y)
c ← (c.x/n, c.y/n)
for each p in points do
    p ← (p.x - c.x, p.y - c.y)
```