

A Graphical Algorithm for Cuneiform Sign Recognition

Daniel Stelzer*

Abstract. Cuneiform is considered one of the most difficult writing systems to learn, and the problem of looking up unknown logograms has frustrated students since the origin of writing itself. Within the last ten years, attempts have been made to simplify the process using computers, but they still require students to encode the signs in complicated ways before searching. We propose a new algorithm to convert line segments on a canvas into one of these encodings, letting students look up signs by simply “drawing” them with a mouse or stylus.

Keywords. cuneiform; optical character recognition; algorithms

1. Introduction. Cuneiform is famous for being the oldest deciphered writing system in the world—and also one of the most difficult to learn. Ever since ancient times, students have lamented the difficulty of learning it (Sjöberg 1974: p. 163), and modern scholars avoid using it as much as possible, relying on transliterations¹ instead. Most textbooks try to avoid cuneiform writing as long as possible, relegating it to advanced lessons to avoid scaring students away and calling it “unquestionably the most difficult aspect of learning” these languages (Huehnergard 2011: p. xxiv).

In the past decade, there have been various attempts to fix the problem computationally. Stelzer (2025), for example, proposes a new way of looking up signs in a dictionary, by typing in codes like { [vvc] h } that represent arrangements of strokes. However, this still requires students to learn to interpret and use these “kadaru codes”, creating a significant barrier to entry. Many students have expressed that it would be far better if they could just *draw* the arrangement of strokes, rather than coming up with the kadaru codes by hand.

This paper describes a basic algorithm to accomplish that task: taking an arrangement of strokes drawn on a canvas, and converting it into the kadaru encoding for subsequent searching.

2. The Problem. Unfamiliar words are an eternal problem for students learning any new language. In most languages, it’s annoying but tractable—since English words can be sorted in a consistent way², you can open a dictionary to a random page, choose an entry, and know if your word comes before or after it. But for languages with logographic writing systems, unknown *signs* prove much more taxing to find. Without a consistent way of sorting them, a student may need to scan through every page of the dictionary one at a time, looking for the sign they need. Indeed, this is what apprentice scribes have been taught since the invention of cuneiform (Robson 2010, Cooper 1996: pp. 48–53).

Without a total ordering on signs, it is fundamentally impossible to do better than a linear search³. However, we now have technology that the ancient Babylonians could never dream of.

* Authors: Daniel M. Stelzer, University of Illinois at Urbana-Champaign (stelzer3@illinois.edu).

¹ Which we also use here to name the signs. Per standard conventions, *italics* are used for phonograms and SMALL-CAPS for logograms.

² Mathematically, there’s a total ordering over them, allowing a binary search in logarithmic time.

³ A sort of ordering has become standard in modern sign lists, based on the number and angle of the sign’s leftmost strokes. This works well for the clean, regular Neo-Assyrian style, but in the words of Borger (2007: p. 2), “it is practically impossible to arrange other versions of cuneiform writing (including the Ur III signs) by the shape of their signs in a consistent and unequivocal way”. As a result, most sign lists are sorted exclusively based on the *Neo-Assyrian* forms—which is little help if the cuneiform you are looking at is not Neo-Assyrian!

We can ease the burden on students by letting a computer trawl through the dictionary instead, showing the user only the signs that match.

3. Encodings. The problem then becomes, how can we encode signs in a way that computers can process? A few different systems have been proposed for this: Gottstein’s (2013) categorization, Homburg’s (2021) PaleoCodage, and Stelzer’s (2025) kadam encoding. The last of these is our particular focus here.

3.1. **STROKES.** Across all eras and languages, cuneiform is written with only five basic strokes: horizontal and vertical, made with two different edges of the stylus⁴; upward and downward diagonal, made by rotating either the hand or the tablet; and the Winkelhaken or “hook”, made with the corner of the stylus.

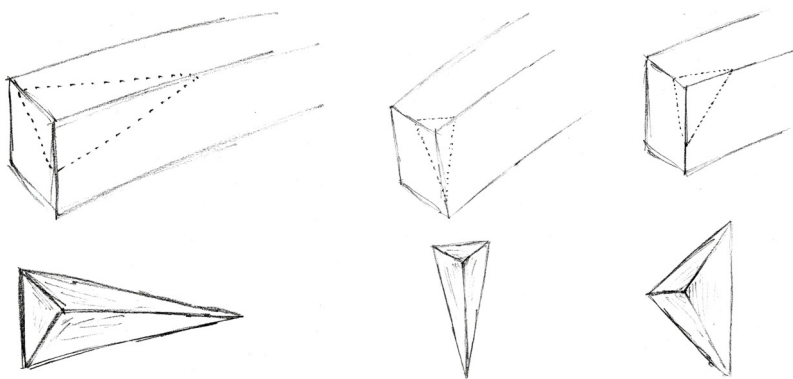


Figure 1. How horizontal, vertical, and hook strokes are drawn with a stylus. Images taken from Stelzer (2024).

All three encoding schemes take these as their basic units. Gottstein’s system simply marks down how many times each stroke appears, PaleoCodage uses a finite automaton controlled by an elaborate set of “operators” to position them on the page, and the kadam encoding treats them as the leaves of a tree.

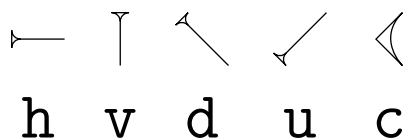


Figure 2. The five types of strokes, and their kadam codes.

3.2. **COMPOSITIONS.** The kadam encoding represents a sign as a tree (in the mathematical sense): a hierarchy of nodes, where the “root” node has no parent, and every other node has exactly one parent. Nodes with multiple children are termed “branches”; nodes with none are termed “leaves”. In this encoding, the leaves of the tree are the five basic strokes, and the branches are three different ways of combining them: arranged horizontally (side by side), arranged vertically (top and bottom), and superposed (intersecting). The first two are referred to collectively as “stacks”.

⁴ Or possibly by rotating the entire hand—interpretations vary.

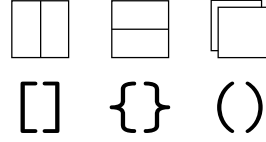


Figure 3. The three types of compositions from Stelzer (2025), and their kadamu codes.

By nesting these compositions, it is possible to encode signs of any complexity. In Stelzer (2025), the leaves are represented by lowercase letters (h, v, d, u, c for horizontal, vertical, downward diagonal, upward diagonal, and hook) and the branches by different types of brackets ([] for a horizontal stack, { } for a vertical stack, and () for an intersection). This bracket notation comes from Downes’s (2008) work on Xixia, who used [] for a horizontal arrangement, { } for a vertical arrangement, and () for nesting one component inside another.

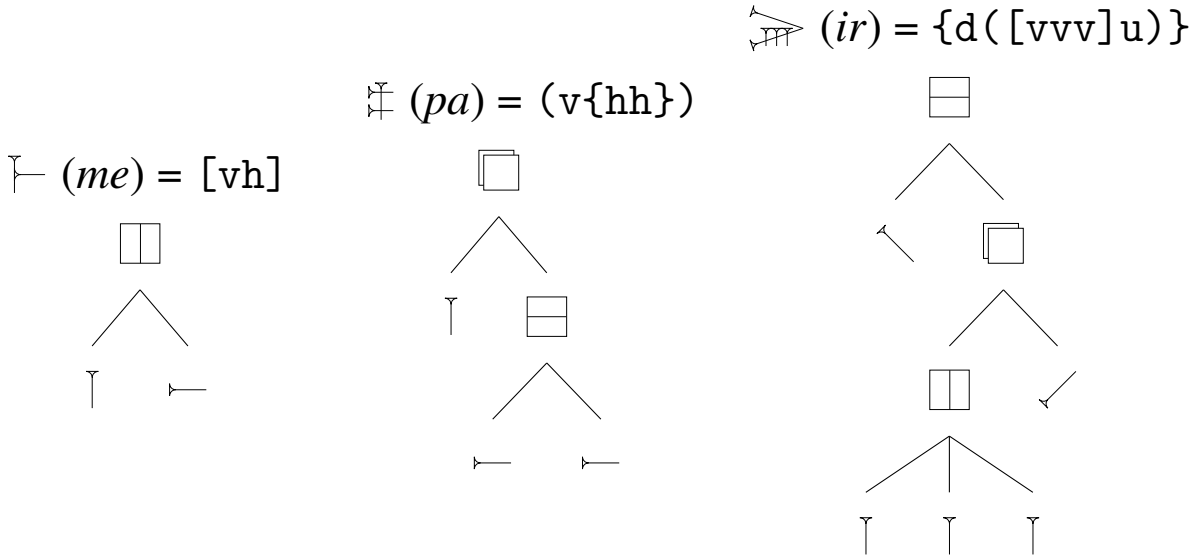


Figure 4. Examples of recursive encoding.

Ideally, these compositions indicate the relationship between *all* of the strokes inside them. If a horizontal stack contains two components, *A* and *B*, then *every* part of *A* should be to the left of *every* part of *B*. This means that we can determine the relationship between any two strokes by looking at their last common ancestor.

In practice, this principle is occasionally violated—Stelzer’s (2025) sign database occasionally uses an intersection composition where not every stroke intersects the others, as in figure 19. But for the purposes of this algorithm, we assume the principle always holds, without exception.

3.3. ADJUSTMENTS AND MODIFIERS. Alongside this basic syntax, the kadamu system includes various “adjustments” and “modifiers” that can be attached to individual nodes. Most of these are to help with rendering, such as shortening a stroke or leaving a margin around an element, which doesn’t matter for our current goal. They are important for displaying a sign accurately, like to capture the difference between *ku* and *ma* (figure 6), but Stelzer’s (2025) search algorithm

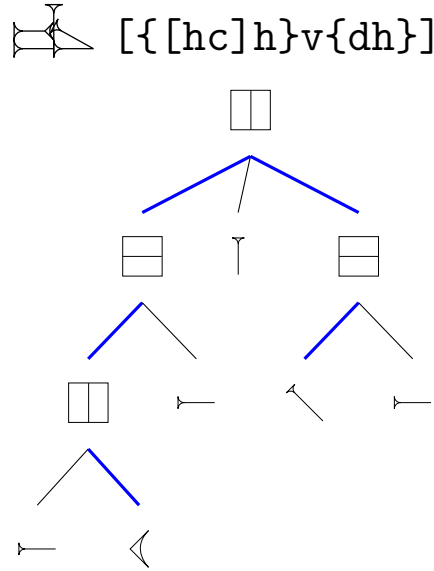


Figure 5. The relationship between any two strokes, like the hook and the downward diagonal in the sign GEŠTIN ‘wine’, should be defined by their last common ancestor.

ignores them, with the philosophy that it’s better to match too broadly than too narrowly. If the search algorithm returns both *ku* and *ma* for a search, the user can choose the correct one based on context.

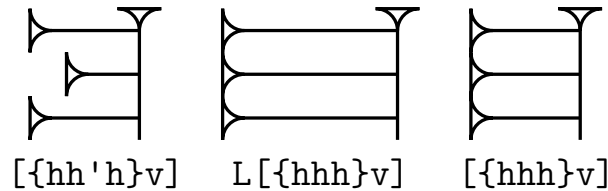


Figure 6. The signs *ku* (left) and *ma* (center), encoded with modifiers (the L and ' characters). On the right, the normalized form used for searching, with those modifiers removed.

Since our goal in this paper is to supply input to that search system, we can likewise ignore these modifiers. Similarly, Stelzer (2025) follows Gottstein (2013) in considering hook strokes equivalent to downward diagonals while searching, so we need not concern ourselves with that distinction. Since the database being searched includes these distinctions and modifiers, the signs will be rendered properly in the results, without requiring the user to input them accurately.

One modifier, however, is very important for our purposes. While PaleoCodage provides a “stacked diagonally” modifier, the kadamu encoding doesn’t. This is because diagonal stacks have a very restricted distribution: diagonal strokes can be stacked horizontally and vertically, but horizontal and vertical strokes can’t be stacked diagonally. This suggests that these diagonal stacks were most likely drawn by rotating the entire tablet.

As a result, they’re implemented as horizontal and vertical stacks of horizontal and vertical strokes, with a “*tenû*” adjustment—the Akkadian term for a sign drawn at an angle.

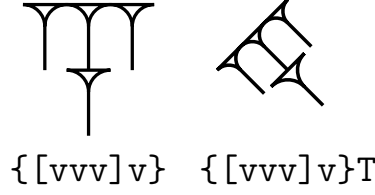


Figure 7. The signs LIMMU ‘four’ and *hi*, demonstrating the *tenû* adjustment.

Horizontal and vertical strokes inside a *tenû* stack are then normalized to upward and downward diagonals for searching. This means the *tenû* adjustment, unlike the others, should *not* be ignored while preparing input for the search algorithm.

4. Ambiguous Stacks. The system laid out above is exactly as presented in Stelzer (2025). Algorithms are given there to render these trees and normalize them (removing adjustments, equating diagonals and hooks, and handling ambiguities like $\{[vv]\{vv\}\}$ versus $\{[vv][vv]\}$), and to search for all trees that match a particular component.

However, the algorithm here requires one additional type of composition, the *ambiguous stack*. This represents a group of strokes that might be stacked either horizontally or vertically, with no reliable way to decide between them.

This comes up specifically when simple horizontal and vertical stacks of strokes are made *tenû*. In Stelzer (2025), the *tenû* modifier is specifically used for more elaborate arrangements of diagonal strokes—ones that require entire trees to be written at an angle, like in the sign *am* (figure 8). This sign is encoded as $\{[hh]\{[vvv]v\}T\}$, with a horizontal stack inside the *tenû* vertical stack.

For a sign like ANŠE ‘donkey’, on the other hand, no *tenû* element is used—it’s encoded as $\{[ddhh]\{dhh\}\}$, with diagonal strokes in a normal vertical stack. This convention tends to make for better renderings, so components like $[vv]T$ (that is, a simple stack of strokes *tenû*) are avoided.

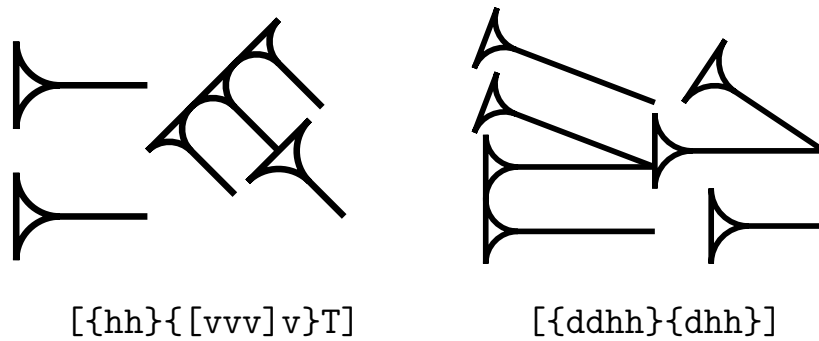


Figure 8. The signs *am*, encoded using a *tenû* adjustment, and ANŠE ‘donkey’, without one. Various other adjustments (not shown here) are used to fine-tune the lengths and positions of strokes, making the diagonal and lower horizontal stroke at the right of ANŠE shorter than the middle horizontal.

This is sufficient when writing out kadam codes by hand. But for the algorithm presented here, it proves much easier to encode parallel diagonals as $[vv]T$ —and this poses a problem for

searching. Stelzer’s (2025) search algorithm relies on those parallel strokes being encoded as either $[dd]$ or $\{dd\}$, depending on their angle (see figure 9). Which of those should our $[vv]T$ normalize to?

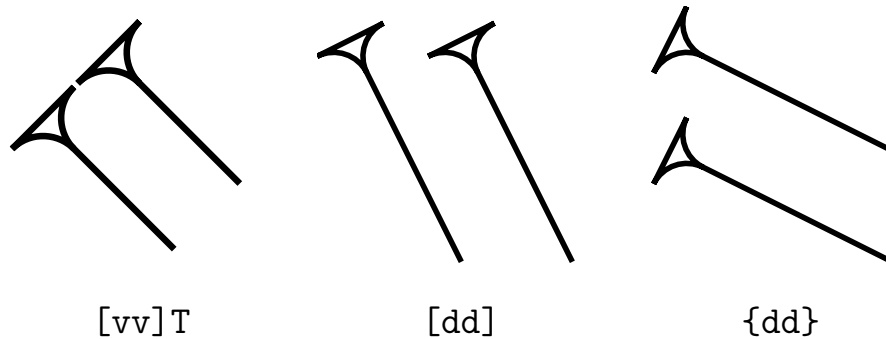


Figure 9. Three different ways of encoding a pair of downward diagonals.

4.1. MODIFIED ALGORITHMS. Ambiguous stacks offer a solution. They function exactly like normal horizontal and vertical stacks, with one exception: while searching, they count as *both*. In other words, we make one change to Stelzer’s (2025) definition of “encompassing” (the algorithm used to determine if a sign in the database matches the component being searched for):

- A stroke encompasses a stroke of the same type.
- Any stroke encompasses a wildcard.
- A composition A encompasses a node B if any child of A encompasses B .
- An intersection A encompasses an intersection B if every child of B is encompassed by some child of A .
- A stack A encompasses a stack B if:
 - A and B have the same type (vertical or horizontal), **or the type of B is ambiguous**, and
 - Every child of B is encompassed by some child of A , and
 - For any children of B x and y , if x precedes y , then the child of A that encompasses y does not precede the child of A that encompasses x .

This means that both $[dd]$ and $\{dd\}$ will “encompass” (i.e., match) an ambiguous stack of diagonals, which we write as $\llbracket dd \rrbracket$ —mnemonically, a combination of braces and brackets.

We do *not* make any change to the normalization algorithm (applied before searching). Ambiguous stacks should remain ambiguous, no matter what they’re nested within; $\{\llbracket dd \rrbracket hh\}$ should *not* be normalized to $\{ddhh\}$. This ensures that it matches both ANŠE ‘donkey’ and *di* (figure 10).

Little attention has been dedicated to rendering these ambiguous stacks, since they’re only expected to be used in this algorithm. But for the sake of visualizing the algorithm’s output, we

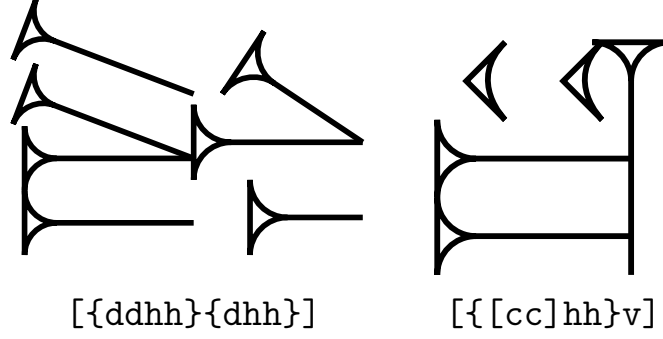


Figure 10. The signs ANŠE ‘donkey’, with the diagonals inside a vertical stack, and *di*, with the hooks inside a horizontal stack. When the distinction between diagonals and hooks is ignored, both will match $\{\llbracket dd \rrbracket hh\}$.

render them as either horizontal or vertical stacks—whichever they originally were before the algorithm modified them⁵.

4.2. SPECIFICATION. Since these ambiguous stacks will only be created by our algorithm, we can avoid complicated edge cases by simply forbidding them. In particular, we can prohibit any branching nodes inside an ambiguous stack, allowing only individual strokes and their modifiers. Violating this restriction is considered an error. This means that the various special cases in the normalization algorithm never apply.

Similarly, since we never expect humans to be encoding these stacks by hand, we don’t have to worry about them being easy to type. For this paper, we represent them with white brackets $\llbracket \rrbracket$ (Unicode 27E6 and 27E7). They clearly form a matched pair of brackets, distinct from the other three compositions, and leave the angle brackets $\langle \rangle$ for future expansions of the system that users might actually need to type.

With this new composition, an ambiguous stack of diagonals can be encoded as $\llbracket vv \rrbracket T$, which normalizes (per the usual rules) to $\llbracket dd \rrbracket$. When searching, both $\llbracket dd \rrbracket$ and $\{dd\}$ will match this element.

5. The Recognition Algorithm. Our goal now is to take an arrangement of strokes, and convert it into a *kadaru* tree for searching. We treat every stroke as a line segment, with a start and end point somewhere on a two-dimensional canvas⁶. A *component* is defined as an unordered collection of strokes—the input to our algorithm.

5.1. STROKE PROCESSING. First, the base case. If a component consists of only a single stroke, we measure its angle and return a leaf node of the appropriate type. As mentioned in section 3.3, Stelzer’s (2025) search algorithm ignores the distinction between hook strokes and downward diagonals, allowing us to ignore it as well. This leaves us with only four stroke types to consider.

Since Hittite has no strokes pointed upward or leftward, we only need to worry about 180

⁵ This is accomplished by adding a meaningless “void stroke” to the end if it was originally a vertical stack. This meaningless stroke is removed during normalization, so it will not impact the search.

⁶ Strokes could also be tagged with further information, e.g. if we wanted to distinguish diagonals from hooks, or draw double-headed strokes with one segment instead of two. This is done in the linked demonstration. However, as discussed in section 3.3, it is not necessary for the algorithm itself, as these modifications are normalized out before searching.

degrees of angle—assigning 45 degrees to each stroke type—and return *h* for a horizontal stroke whether it points right or left. For other styles of cuneiform, upward or leftward strokes could be detected and handled here⁷.

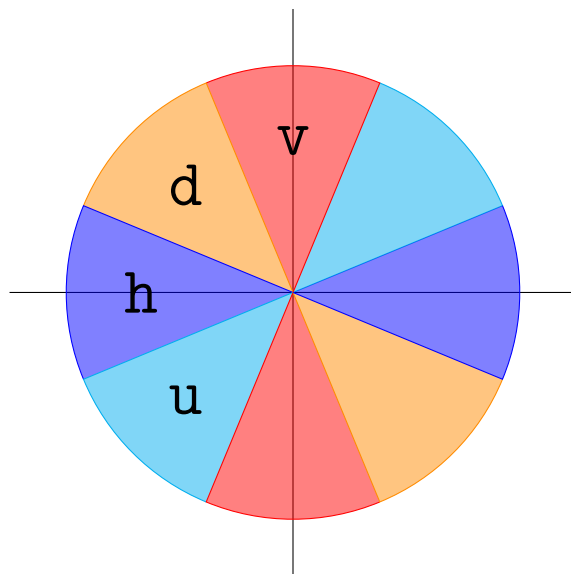


Figure 11. The stroke processing step.

5.2. **ROTATION STEP.** Second, we handle the *tenû* adjustment. If all the strokes in the component are diagonal (as defined in the previous step), we test if a rotation will turn them into only horizontals and verticals.

To do this, we take the average of the angles of all the strokes, modulo 90 degrees⁸, which removes the distinction between vertical and horizontal. Then we rotate the entire component clockwise by that amount. If all the strokes in the result are horizontal and/or vertical, we recurse on that, then apply a *tenû* adjustment. Otherwise, we continue with the original unrotated component.

5.3. **SEPARATION STEP.** Third, we handle stacks. Our goal is to draw a straight horizontal or vertical line that divides the component cleanly, without crossing any strokes. We can do this by projecting the strokes onto the X- or Y-axis, then looking for gaps in the projections.

In other words, we convert each stroke to an interval by taking the X-coordinates of its endpoints. We then sort these intervals by their lower bounds, iterate through the list, and consolidate any adjacent intervals that overlap (the first one's upper bound is greater than or equal to the second one's lower bound).

If multiple intervals remain, the projections must be separated by gaps on the X-axis. We can take the set of strokes represented by each interval, recurse on each, and combine the results with a horizontal stack—and similarly for the strokes' Y-coordinates and a vertical stack.

⁷ The *kadaru* system actually provides a ? modifier for this, but per section 3.3, we are ignoring modifiers at present, to begin with the simplest possible version of the algorithm.

⁸ That is, the remainder after dividing the angle by 90 degrees. This means that a vertical at 0 degrees is equivalent to a horizontal at 90 degrees, and a downward diagonal at 45 degrees is equivalent to an upward diagonal at 135 degrees. See appendix A for a worked example.

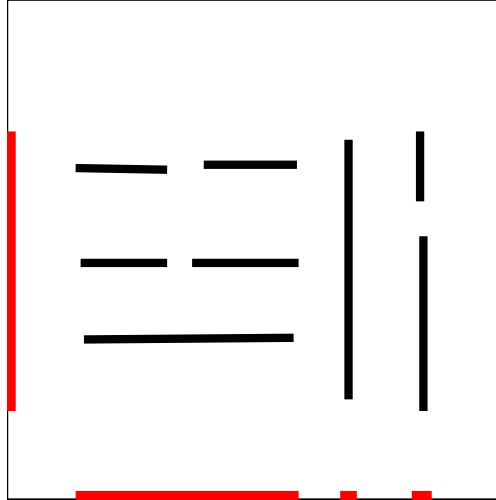


Figure 12. The strokes of the sign ya , projected onto the X- and Y-axes. The gaps in the projection on the X-axis (and not on the Y-axis) mean that we can divide this component horizontally (and not vertically).

5.4. **PARTITION STEP.** Finally, if the component can't be divided horizontally, vertically, or diagonally, it must be an intersection. Our goal now is to divide it into sub-components so that no two strokes in the same sub-component intersect. In a sign like *lu* lu , for example, we want to process the three verticals as one sub-component and the three horizontals as another.

We do this by iterating through the strokes in the component and assigning each one a “color”. No two strokes of the same color can cross, and each stroke is given the first color possible without violating that constraint. We then recurse on each color, and combine the results with an intersection.

For a demonstration, see appendix A.

6. Limitations.

6.1. **INPUT.** One immediate problem is that tablet and mouse input is inherently imprecise. Humans trying to draw two strokes meeting at their tips are likely to accidentally overlap them, when the algorithm expects there to be a gap, and vertical strokes in a vertical stack are unlikely to be perfectly aligned. If this is meant to be easy for students to use, it needs to tolerate a certain level of imprecision.

To fix this, we add a “tolerance” value t to the algorithm. Before doing any processing, every stroke has t of its length removed from each end, ensuring that small overlaps don't get analyzed as intersections. Then, during the separation step, any stroke whose projection is shorter than t has it expanded to that value, ensuring that vertical stacks of vertical strokes and horizontal stacks of horizontal strokes don't have to be perfectly aligned (figure 13).

The demonstration in section 7.1 uses a tolerance of 10 pixels on a 300 by 300 pixel canvas.

6.2. **DIVIDERS.** For the most part, applying the steps in this order is sufficient. First, try to rotate it; then, try to separate it; finally, try to partition it. However, there's one significant exception.

During the separation step, it is sometimes possible to divide either horizontally *or* vertically. When this happens, a decision has to be made. Various heuristics have been tried, such as prior-

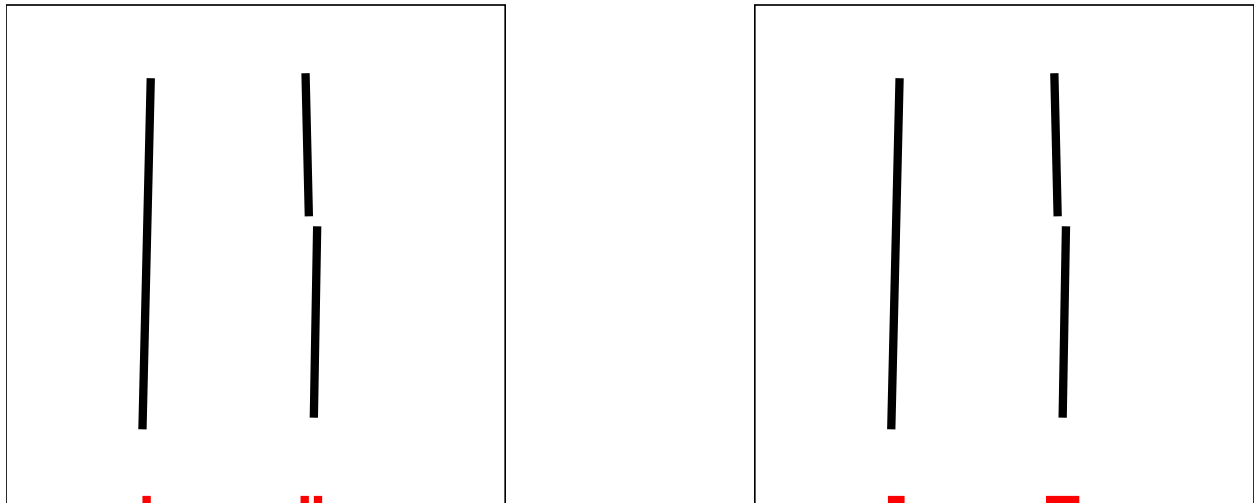


Figure 13. The strokes of the sign $\overline{a}$, projected onto the X-axis. Due to the imprecision of human input, none of the strokes overlap, so a naïve application of the algorithm will end up dividing it into three separate columns (left). Adding “tolerance” to the algorithm (right) ensures it is properly divided into only two columns.

itizing the direction that creates the fewest divisions; the current version of the code is even simpler, *always* choosing a horizontal division if possible and only falling back to a vertical division if it’s not.

But we have yet to find a heuristic that always divides correctly. For example, prioritizing horizontal divisions is wrong for the sign NINDA ‘bread’ (figure 14).

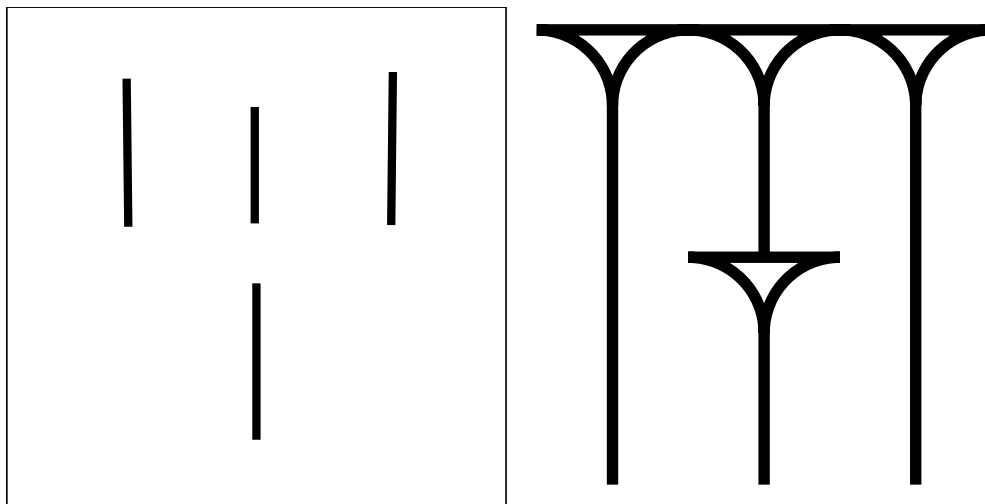


Figure 14. The sign NINDA ‘bread’ is mis-parsed, due to dividing horizontally instead of vertically.

The current solution is “divider strokes”: strokes that act like any other for the purpose of the algorithm, but are not included in the parsed output. A divider through the middle of NINDA ensures that it can only be divided vertically, giving the correct output (figure 15).

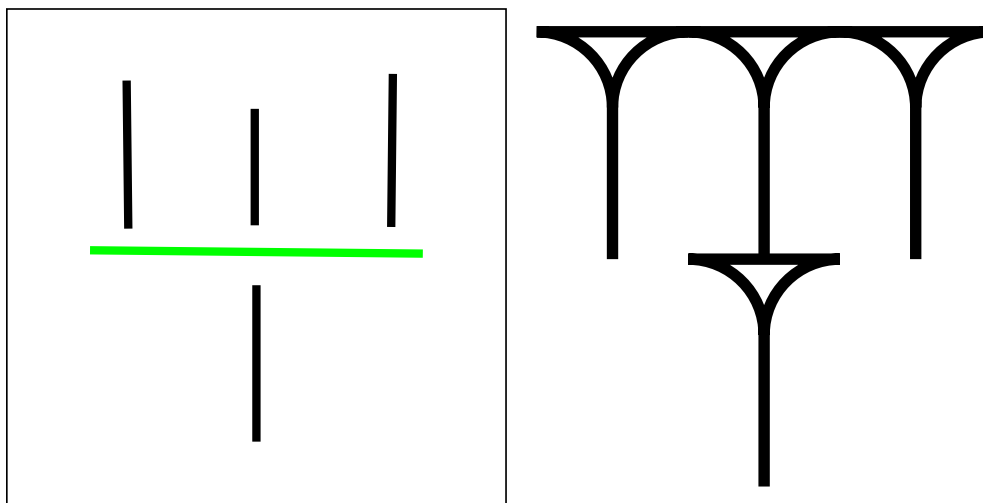


Figure 15. A divider stroke (in green) prevents dividing horizontally, giving the correct parse.

These work well enough when students are drawing signs by hand for searching, and “the system won’t let any strokes cross this line” is an intuitive enough explanation (even if it’s not entirely accurate). But for automatic sign recognition, using data from actual tablets, it will not be sufficient. A better solution is needed for this algorithm to work on uncured data.

6.3. INDIVISIBILITY. Another issue arises if *none* of the steps can apply to a particular component. The arrangement in figure 16 contains multiple strokes, doesn’t consist of diagonals, can’t be divided by a vertical or horizontal line, and has no intersections.

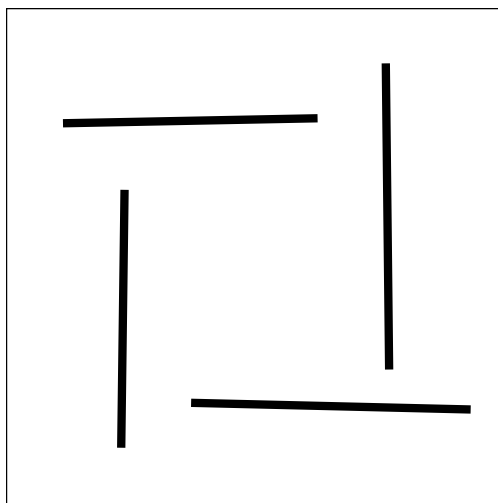


Figure 16. An arrangement of strokes that can’t be parsed.

There is no way to represent this arrangement of strokes recursively, so when this happens, parsing simply fails. Fortunately, this is not an arrangement we have ever found in actual cuneiform. When “boxes” are drawn, as seen in figure 17, the order of writing the strokes generally keeps them from interlocking like this.

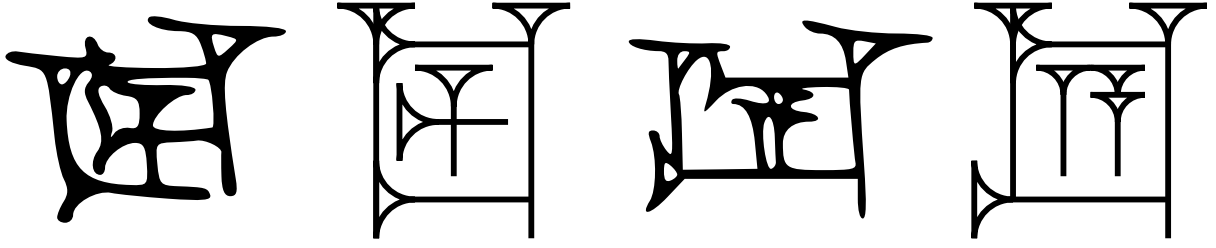


Figure 17. Variant forms of the signs *lu* and AMBAR ‘swamp’, as drawn in Rüster & Neu (1989) and rendered using the recursive encoding. These arrangements, $[v\{h(vh)h\}v]$ and $[\{[v\{h[v\{vv\}]\}]h\}v]$, cause no difficulties for parsing.

6.4. PARTITION OPTIMIZATION. In the partition step (section 5.4), our algorithm for grouping the strokes is not quite ideal. This problem—grouping strokes into the minimum possible number of colors so that no two strokes of the same color intersect—is a special case of the graph coloring problem⁹, and the general problem is NP-complete.

A greedy algorithm (see appendix B) can produce a solution in linear time. However, the result of this algorithm is not guaranteed to be optimal—it can sometimes use more colors than necessary. More concerning, the result can depend on the order in which the strokes are processed.

An example of this is shown in figure 18. The same sequence of strokes can be parsed in two different ways depending on the order of processing. Whether this is the order the strokes were entered, or some arbitrary but consistent ordering (e.g. derived from the coordinates of the endpoints), it’s likely to be unintuitive for the user. Nor does adding dividers help the situation.

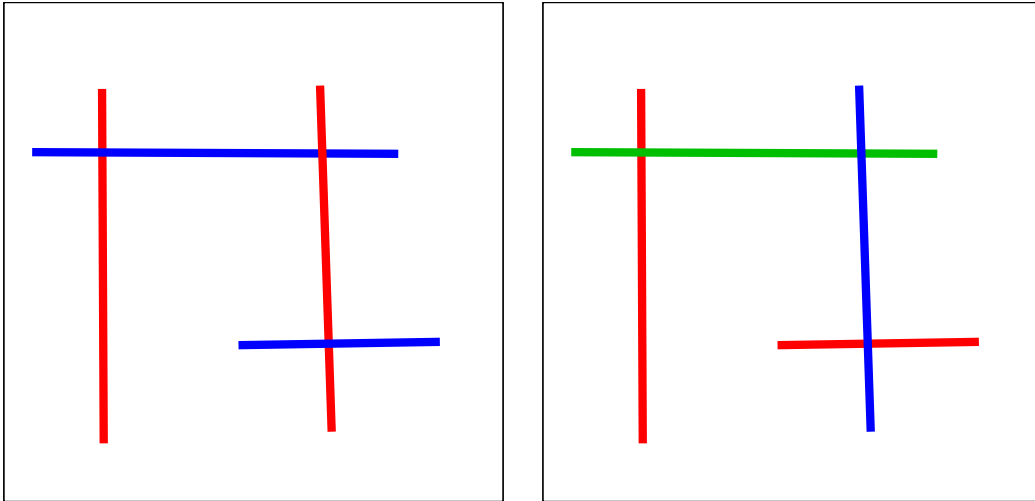


Figure 18. The cuneiform equivalent of the path graph P_4 , which our algorithm can color in two fundamentally different ways.

However, this situation is very unlikely to come up. Deliberate intersections are relatively rare in Hittite cuneiform; strokes are much more likely to meet than to cross. As discussed in

⁹ The vertices of the graph are the strokes, and there is an edge between any two strokes that intersect.

section 3.2, we also assume that the relationship between any two strokes is defined by their last common ancestor. The arrangement in figure 18 violates this, so there is no clear correct answer for how it *should* be encoded. In practice, a very small number of signs do violate this assumption (as shown in figure 19), but they are the exception rather than the rule.

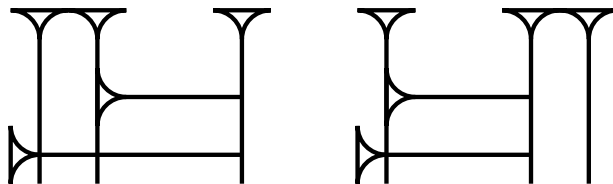


Figure 19. New Script forms of the signs *ur* and *ip*. These rely on special modifiers to keep the horizontals from overlapping, even though their last common ancestor is an intersection, which should (strictly speaking) put them right on top of each other.

In particular, this assumption means we should never have strokes of the same type in different colors. If we did, that would make their last common ancestor an intersection, and it's not possible for a horizontal stroke to meaningfully intersect another horizontal stroke. That fact heads off most possible ambiguities in practice. It also means that, when partitioning; we should never need more than four colors¹⁰—one for each type of stroke.

Thus, as with the indivisibility issue, this doesn't seem to be a problem in practice—at least for the Hittite cuneiform currently under consideration. It remains to be seen whether it will become an issue for other languages or eras.

7. Conclusion. The algorithm presented here has so far proven very effective at helping students enter recursive codes. An experiment is currently under way to test empirically whether this helps with the speed or accuracy of looking up unknown signs, and while relatively simple, students have generally found it straightforward to use.

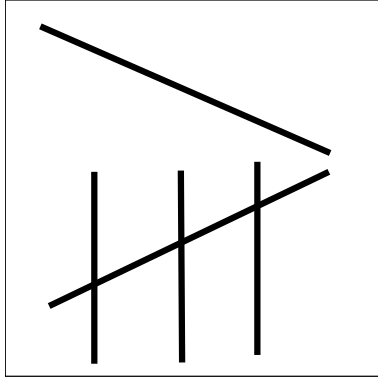
However, some aspects (such as the divider strokes) still rely on manual adjustment by the user. This is fine for a search interface, but poses a serious problem for automatically encoding signs from optical data. Among other things, a better heuristic for deciding between horizontal and vertical stacks, and a more sophisticated implementation of ambiguous stacks, will be needed.

Nonetheless, we hope the methods presented here form a useful starting point for more sophisticated algorithms and implementations in the future.

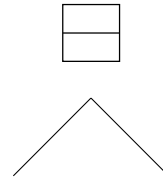
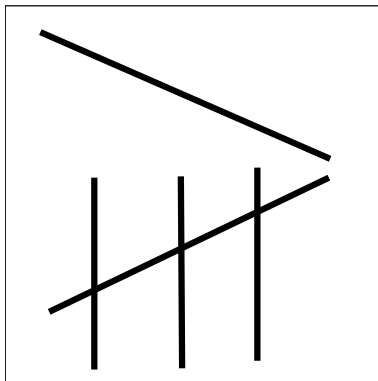
7.1. SUPPLEMENTARY MATERIALS. An implementation of this algorithm can be found at <https://dstelzer.pythonanywhere.com/cuneipaint.html>. The source code is available at <https://bitbucket.org/dstelzer/hantatallas/src/master/sanhataallas/strokeparse.py>.

A. Demonstrations. Imagine a student is trying to identify the sign  *ir*. They draw the strokes they can recognize.

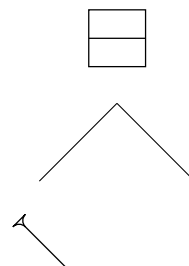
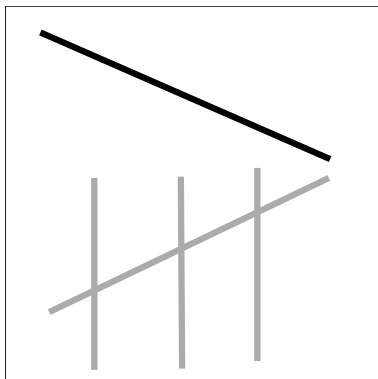
¹⁰ Four are needed when *every* type of stroke intersects, like in the old lapidary form of the sign *an*, .



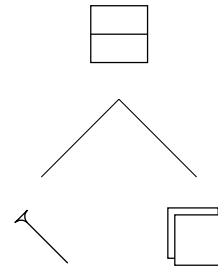
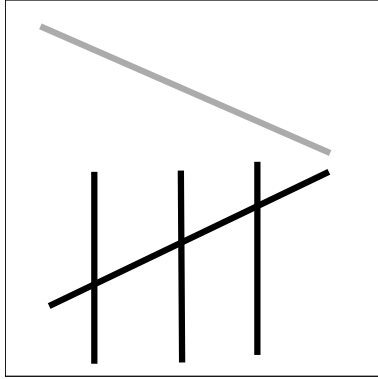
They would likely draw the ends of the diagonal strokes touching, but see section 6.1. This collection of strokes doesn't consist solely of diagonals, and it can't be divided horizontally, but it can be divided vertically.



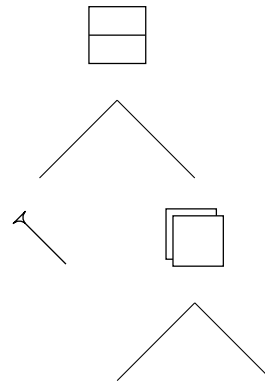
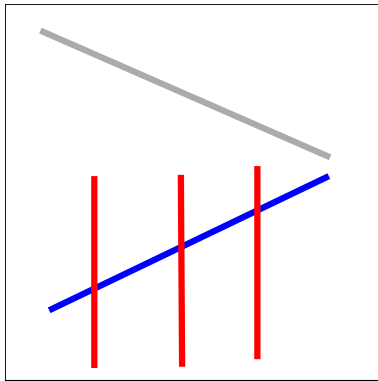
The upper component has only a single stroke, so we measure its angle: it is a downward diagonal.



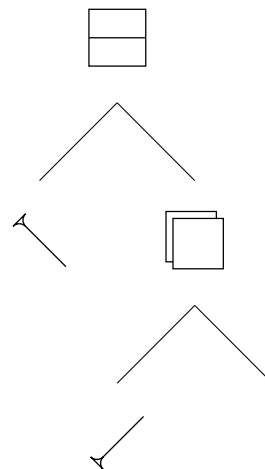
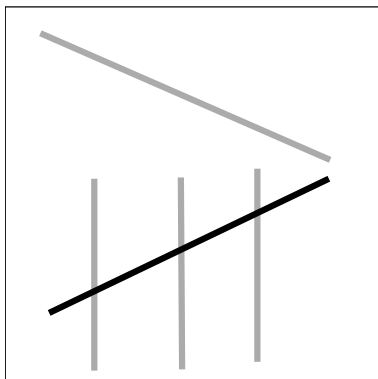
The other component doesn't consist solely of diagonals, and it can't be divided horizontally or vertically, so it must be an intersection.



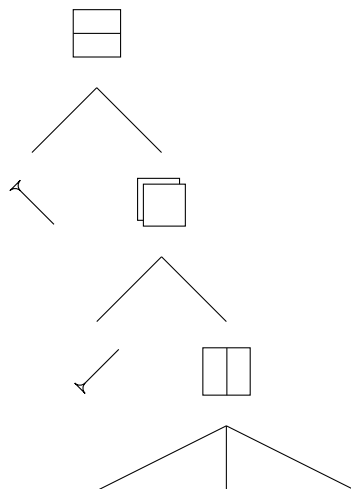
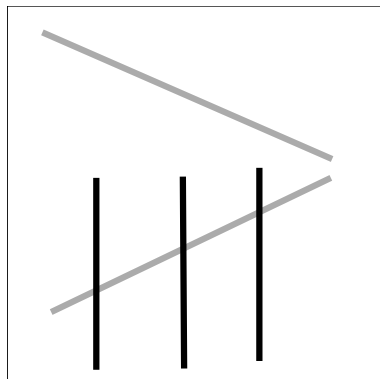
Using the algorithm from section 5.4, we end up with two colors.



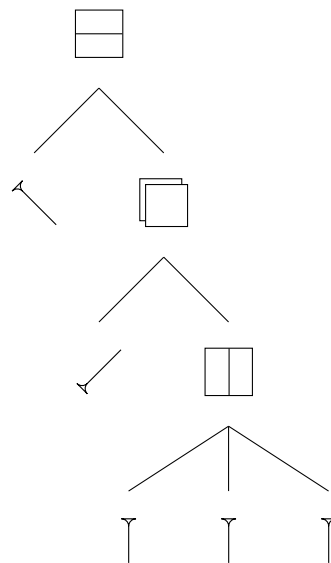
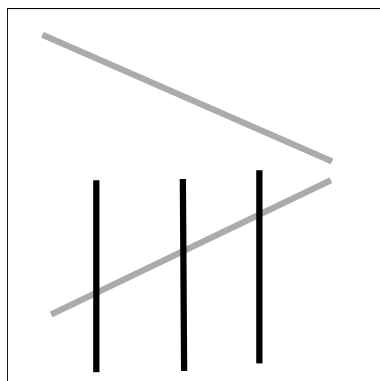
The blue component is a single stroke again, so we measure its angle, and call it an upward diagonal.



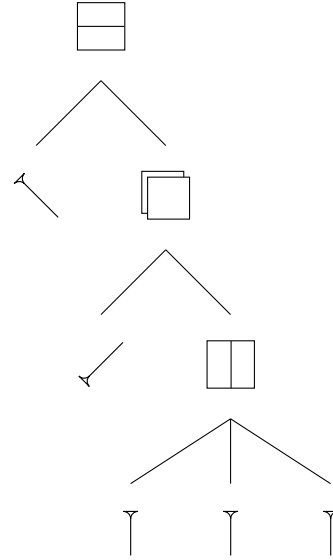
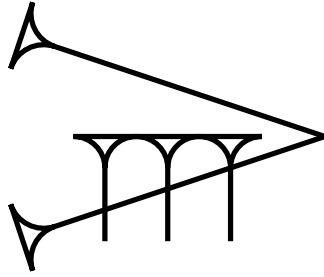
The red component doesn't consist solely of diagonals, but it can be divided horizontally into three pieces.



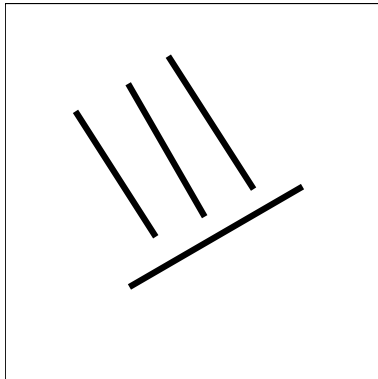
Each of those pieces consists of a single stroke, and by measuring the angle, we find it to be a vertical.



This gives us the expected tree for the sign *ir*.



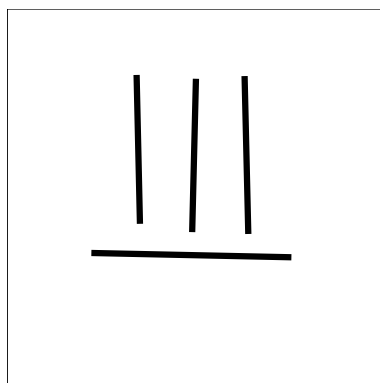
Now suppose the student is searching for the sign 𐰇 uz₆ ‘goat’. For the sake of demonstration, we focus on the diagonal component on the right. Once again, the tolerance ensures the strokes do not touch, even if the user drew them with a slight overlap.



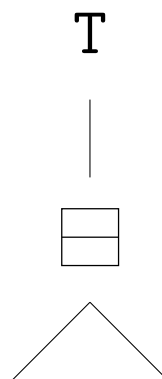
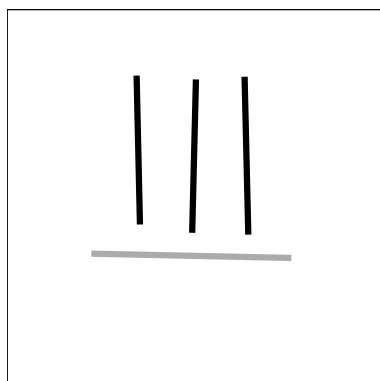
This component consists entirely of diagonals: their angles are 57.38° , 57.32° , 60.08° , and -30.10° (measuring clockwise from the horizontal axis), which puts them all in the “diagonal” regions of figure 11.

Reducing these angles modulo 90 degrees (putting them all between 0° and -90°) gives us -32.62° , -32.68° , -29.92° , and -30.10° , eliminating the distinction between horizontal and vertical strokes. Since these strokes were approximately 90 degrees apart, they’ve now ended up very close together: the average of them all is -31.33° .

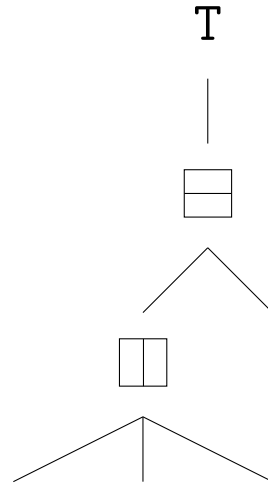
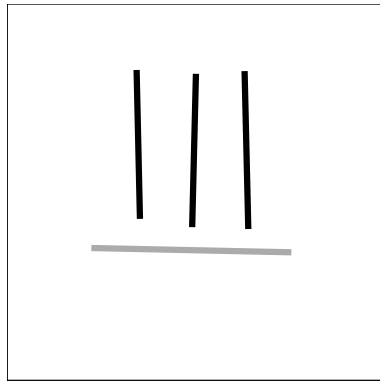
We now rotate the whole component counterclockwise by this average angle.



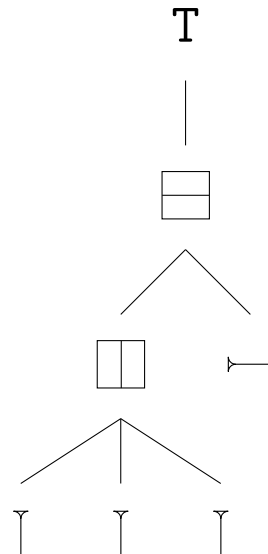
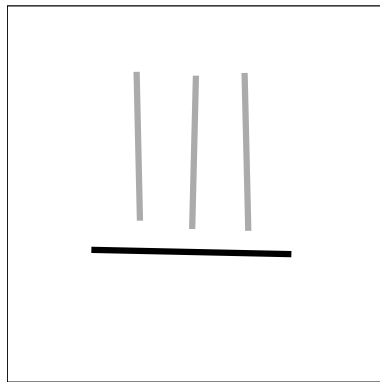
This produces a component consisting of horizontals and verticals, so the rotation step was a success. We insert a *tenû* adjustment and proceed. The result can be divided vertically:



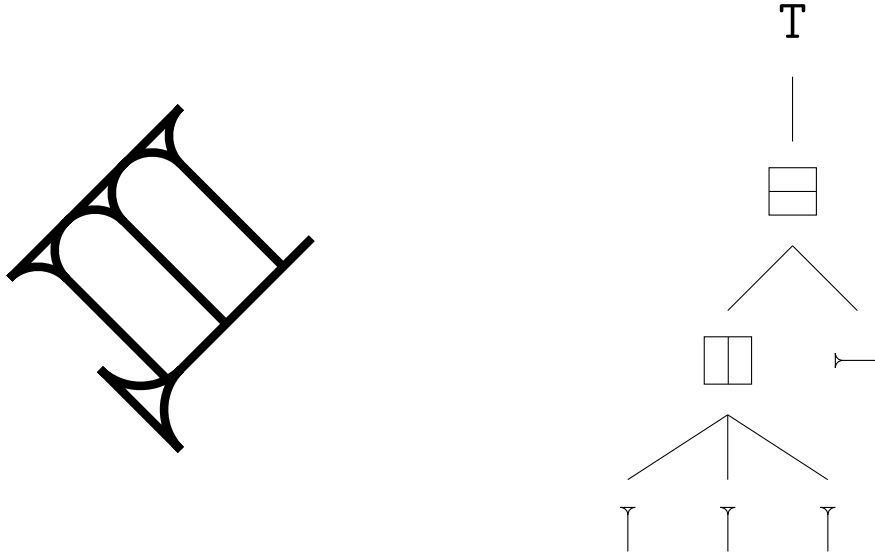
The upper component can be divided horizontally into three sub-components:



And the remaining components are single strokes, so we simply measure their angles.



This gives us the expected result.



B. Complexity. Every iteration of this algorithm requires $O(s \log s)$ time to operate on a component of s strokes:

- The rotation step iterates once over the list of strokes to calculate their angles, and a second time if necessary to rotate them, in $O(s)$.
- The separation step iterates once over the list of strokes to form the intervals, again to sort them, again to remove overlaps, and a final time to divide them into sub-components. The sorting portion dominates this step, giving $O(s \log s)$ runtime.
- The partition step uses a greedy algorithm¹¹ that runs in $O(s)$ time.

The last of these is the least obvious. But it can be implemented in $O(s + i)$ time¹², where s is the number of strokes and i the number of intersections, as follows:

- Create a list f , indicating whether each color has been found in the current step.
- For each stroke S in the component:
 - Set all entries of f to “false”.
 - For each stroke T that intersects S :
 - * If T has a color C , set $f[C]$ to “true”.
 - Find the first color C such that $f[C]$ is “false”.
 - Set the color of S to C .

¹¹ That is, an algorithm that always makes the best choice it can *given the information available at the time*, but never goes back to reconsider those choices once more information is available.

¹² Though the reference implementation runs in $O(s^2)$ for clarity.

Since there are only four types of strokes, i can be at most four times s , making the overall running time of this algorithm $O(s)$ for well-formed signs. It does require all the intersections to be computed in advance, which can be straightforwardly done in $O(s^2)$ time before beginning the algorithm.

Every time the algorithm recurses, each component becomes smaller, and the total number of strokes never increases. The only case where a component does not become smaller is in the rotation step, but one rotation step cannot follow another¹³. Thus, the number of recursions that do not decrease the overall complexity is within a constant factor of the number that do.

In the best case, each (non-rotating) recursion divides the problem into pieces of equal size, for $O(\log s)$ recursive calls and a runtime of $O(s \log^2 s)$. But in the worst case, each (non-rotating) recursion splits off only a single stroke, leading to $O(s)$ recursive calls, and an overall time complexity of $O(s^2 \log s)$. Without more data on practical applications, it is impossible to say which of these cases is more likely.

References

- Borger, Rykle. 2007. *List of Neo-Assyrian Cuneiform Signs: A practical and critical guide to the Unicode blocks ‘Cuneiform’ and ‘Cuneiform Numbers’ of Unicode Standard Version 5.0.* <http://www.sumerisches-glossar.de/download/SignListNeoAssyrian.pdf>.
- Cooper, Jerrold S. 1996. Sumerian and Akkadian. In Peter T. Daniels & William Bright (eds.), *The world’s writing systems*. https://archive.org/details/isbn_9780195079937/.
- Downes, Alan. 2008. *The Xixia writing system*. Macquarie University Bachelor of Arts Honours Thesis. https://www.academia.edu/10629257/The_Xixia_Writing_System_Bachelor_of_Arts_Honours_Thesis_.
- Gottstein, Norbert. 2013. Ein stringentes Identifikations- und Suchsystem für Keilschriftzeichen. German. *Mitteilungen der Deutschen Orient-Gesellschaft zu Berlin* 145. 127–136. https://www.orient-gesellschaft.de/repositorium/MDOG/MDOG_145.pdf.
- Homburg, Timo. 2021. PaleoCodage: enhancing machine-readable cuneiform descriptions using a machine-readable paleographic encoding. *Digital Scholarship in the Humanities* 36(Supplement_2). ii127–ii154. <https://doi.org/10.1093/llc/fqab038>.
- Huehnergard, John. 2011. *A grammar of Akkadian*. Third (Harvard Semitic Studies). Winona Lake, Indiana: Eisenbrauns. https://www.academia.edu/234695/2011_A_Grammar_of_Akkadian_3rd_edition_.
- Robson, Eleanor. 2010. Using sign lists: Labat, Borger, and PSL. In *Cuneiform revealed: an introduction to cuneiform script and the Akkadian language*. <http://kn.prs.heacademy.ac.uk/cuneiformrevealed/learningsigns/usingsignlists/>.
- Rüster, C. & E. Neu. 1989. *Hethitisches Zeichenlexikon: Inventar und Interpretation der Keilschriftzeichen aus den Boğazköy-Texten* (Studien zu den Boğazköy-Texten: Beiheft). O. Harrassowitz. <http://books.google.com/books?id=L15pAJsx\y0C>.
- Sjöberg, Åke W. 1974. The Old Babylonian Eduba. In S. J. Lieberman (ed.), *Sumerological studies in honor of Thorkild Jacobsen on his seventieth birthday* (Assyriological Studies 20), 159–179. University of Chicago Press. <https://oi.uchicago.edu/sites/oi.uchicago.edu/files/uploads/shared/docs/as20.pdf>.

¹³ A rotation is only attempted if all strokes in a component are diagonal, and only succeeds if *no* strokes in the result are diagonal.

- Stelzer, Daniel M. 2024. *An Introduction to Cuneiform*. <http://meadstelzer.com/daniel/cuneiform/lessons/cuneiform.html>.
- Stelzer, Daniel M. 2025. A recursive encoding for cuneiform signs. *it - Information Technology*. <https://doi.org/10.1515/itit-2024-0067>. Pre-published.