Optimal Line Breaking for Comic Translation Using Dynamic Programming

Applying Variable-Width Knuth-Plass Line Breaking to a Comic Translation Tool

Ahmad Fauzan Putra - 13524141
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: amanfauzan31@gmail.com , 13524141@std.stei.itb.ac.id

***Abstract*—Translating a comic replaces the original words inside each text bubble with astring that must fit the bubble's irregular outlines. Many renderer does it greedily, while one of the optimal solution the Knuth-Plass dynamic program, minimizes a global fill cost. The text are processed by a Knuth-Plass dynamic program with per line cost that uses the available with of the line number, the bubble's cross-section at that position in the y axis, and apply it to the text bubbles a comic translation tool detects, after receiving each bubble's real shape from the page. On a set of 30 real bubbles from Korean and Chinese comics, a dynamic programming and a greedy algorithm can fit every bubble while single constant width heuristic manges to get only 57 to 63 percent, and dynamic programming manages to get the optimum cost that a brute force method got on the bubbles that were checked. In this dynamic programming Knuth-Plass we added a Linguistic break penalty, a heuristic that the dynamic programming also optimizes, which improved the lines that end on articles, prepositions, conjunctions and pronouns as a way to improve comic translation quality.**

Keywords—dynamic programming; line breaking; Knuth*—Plass*; text layout;* comic translation*;

I. Introduction

Translating a comic is mainly a language problem, but when a translator set the words in certain ways throughout the limited allotted space, they could potentially degrade the reading experience by making the lines break in ways that cuts off unnaturally. After the original words has been erased, the translated words must be placed back inside the same text bubbles it came from. Text bubbles varies by a lot, they are often irregular. They are not rectangular columns that paragraphs usually reside in. This makes it so that the regions available for text changes from one line to the other. It could be narrow at the top and bottom of the bubble and wide across the middle or have an opening that leads to another text bubble or other type of shape according to the will of the creator of the comic, though typically they are ellipses.

When the text is broken into lines, it is also a decision that would decide whether the text fits into the bubble or not. A bad set of line breaks would go beyond the region of the text bubble, or leave some lines almost empty. This in assure that exist in some naive machine translation tools. The bubbles used in this paper comes from a comic translation detection tool, and the layout produced here is what the tool would use as a metric to break the text into lines.

There is an answer to where lines can optimally break according to their allotted width. Knuth and Plass in 1981 displayed that breaking a paragraph into lines to minimize a total fill cost (uneven line breaks) could be optimally solved using dynamic programming, and their method creates a way to optimally minimizes uneven line breaks [1]. The Knuth Plass algorithm assumed a single constant line width and a text bubble does not provide one. The two easiest way to force a single width on a bubble both fail. By using the bubble's narrowest width makes it so that its wide middle are wasted and runs out of vertical lines, and using its mean width creates breaks that goes beyond the narrow top and bottom creating an ugly mess.

This paper puts optimal line breaking to work inside a comic translation tool and analyzes what it costs and what it buys. The problem states the available width A(k) as a function of the line index k which is the horizontal cross-section of a bubble at the vertical position in the y axis which line k occupies because a single constant width cannot express the geometry of an irregular bubble. We gain that shape from the page by flood-filling the bubble's inked outline, while an object detection model and OCR provide the text from the page to be translated. We lay the words out with the Knuth-Plass dynamic program at this per-line width, which runs in $O(n^2 \cdot K)$ time. We use a linguistic break penalty as an additive metric that the DP optimizes, so the layout fits the bubble, fills the lines evenly, and breaks at certain points that minimizes stopping at certain words in order to mimic a quality translation. And we compare it against a constant-width Knuth-Plass using the minimum width box and mean width box, greedy first-fit, and brute force, on a real set of comic images, measuring whether it is possible to fit the text completely within the bubble without passing the bounds, fill balance, and break quality.

With the $O(n^2 \cdot K)$ run time, it implies a few things. The DP is not faster than greedy. It is asymptotically slower, so the runtime study is about how the run time grows (a polynomial optimum versus an exponential brute force). What it solves is feasibility and even fill on bubbles a single width solution

cannot solve. The breaker itself only uses a word list and a polygon of the text bubbles.

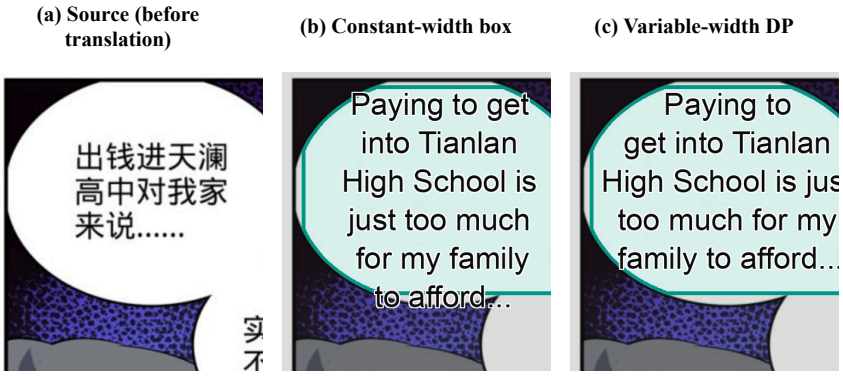

Fig. 1. Refitting a translated line into a real comic speech bubbles (a page from ac.qq.com). The bubbles area is the actual drawn shape extracted from the artwork, not the rectangular text box; it is tinted and outlined in teal, and the surrounding art is lightly dimmed. (a) the original page before translation; (b) a constant-width box breaks the sentence for one average width, producing one line more than the bubbles holds, so the final words fall below the outlined area, outside the bubble; (c) the variable-width DP reads the bubbles's true width at each line and fits. Greedy produces the same fitting layout as the DP on this bubbles (Table III); only the constant-width box fail

II. Theoretical Foundation

A. Line Breaking as Optimization

The words are modelled as boxes of certain width. Breaking the paragraph into lines means that we would need to choose at each of the gaps between consecutive words, whether we need to break it there or later on. A line that is not completely filled would have leftover space or slack. The fill cost of a line is how far it is from an ideal fill. The fill cost of a layout is the sum of it over all of the lines. Optimal line breaking chooses the set of breaks that has the least total fill cost, considering it by using the whole paragraph at once and not by each line by itself. This fill cost is one component that will improve the quality of the layout, alongside the natural linguistic break.

B. Greedy Paradigm

Its a first-fit greedy that goes over the words from left to right, adding each word to the line we are currently working on and if it still fits and starting a new line if it overflows [2]. It runs in O(n) time and is what some text renderers use. It is optimal per line and never goes back, it doesnt consider anything other than if it would fit or not and doesnt consider the future. A word is packed into a line early and doesnt have a mechanism to push down to create an even more optimal solution. On a constant-width rectangular box, a greedy algorithm is often acceptable already, but on a bubble, pushing it into a line earlier could cause a suboptimal solution.

C. Brute Force Paradigm

Because the word order is fixed, the space of layouts is the set of subsets of the n-1 word gaps. 2^{n-1} ways to split the word sequence. Brute force enumerate all of them [3], and discards those that overflows any line, scores the rest with the fill cost and returns the cheapes. It costs $O(2^n \cdot n)$ and becomes too slow near n = 20.

D. Dynamic Programming Paradigm

Optimal line breaking has two properties that dynamic programming needs to succeed. An optimal substructure and Overlapping subproblems, the optimal layout of a prefix is reused by many longer prefixes [4]. For a constant width W the Knuth-Plass recurrence is displayed using the code below.

```
best[0] = 0
for j in 1..n:
    best[j] = min(best[i] + line_cost(i+1, j, W)
                  for i in 0..j-1)
```

Where line_cost is the fill cost of setting words i+1..j on one line with a width of w. Computing best[n] and following back-pointers will recover the optimal break set.

E. Fill Cost and Slack Squared

The standard fill cost penalizes the square of a line's slack (its unused width) and not the “true” value. The reason is balance, Think of three lines that must absorb a total slack of 9. Distributing it evenly by ordering it (3,3,3) will costs $3^2+3^2+3^2 = 27$, where using it all up on one line by doing (0,0,9) would costs 81. A linear penalty would give the two the same value despite one being obviously worse. Squaring makes the optimizer spread the slack evenly, the even, centered fill a comic bubbles would need, greedy which packs each line to the brim would fail to create.

The fill cost measures how evenly the lines are filled. We call it the fill cost because a low fill cost is necessary but not sufficient for a good layout because a short line does not mean it is bad, ending a line at a boundary can read better than a perfectly even but breaks during the middle of phrases. Readability has three parts, the text must fit the bubble (feasibility, it is feasible for the text to fit into the bubble), the lines should be evenly filled (fill cost), and the breaks should fall at points that makes sense linguistically.

III. Problem Formulation

The input is an ordered word list w(1)..w(n) with widths width(w(t)) measured at a certain font size, an word gap space s, a line height h, and the bubble polygon P.

Line k occupies the vertical band [top+(k-1)h, top+kh]. The available width on line k is the bubble's horizontal cross-section there, taken as the minimum across the band so that text set on that line never pokes outside the bubble. The number of lines the bubble admits is K = floor(height(P)/h). The code below gives A(k) and the content width of words i..j on one line.

```
# A[k] = bubble width on line k
K = floor(height(P) / h)
for k in 1..K:
    band = rows from top+(k-1)*h to top+k*h
    A[k] = min(span(P, row) for row in band)

def cw(i, j):   # content width of words i..j
    return sum(width[t] for t in i..j) + (j-i)*s
```

The fill cost of placing words i..j on line k is the code below.

```
def cost(i, j, k):
    c = cw(i, j)
    if c <= A[k]:
        return (A[k] - c) ** 2  # squared empty space
    return INF                  # overflow forbidden
```

Penalizing every line, including the last, encodes the centered, balanced fill an even bubble would need, rather than the left pushed look of greedy text. The output is a partition of the words into at most K consecutive lines of minimum total fill cost, or infeasibility if none of them fits the bubble.

In a rectangular column A(k) is constant, the per-line cost does not depend on the line number, and classic Knuth-Plass needs no line index. In a bubble A(k) varies, so the cost of a group of words depends on which line it occupies, and the problem cannot be reduced to a single width. The minimum-width box wastes the wide middle and runs out of lines, while the mean-width box overflows the narrow ends. Reading the width on each line is what the bubble requires. Fig. 2 shows A(k) measured on a real bubbles.

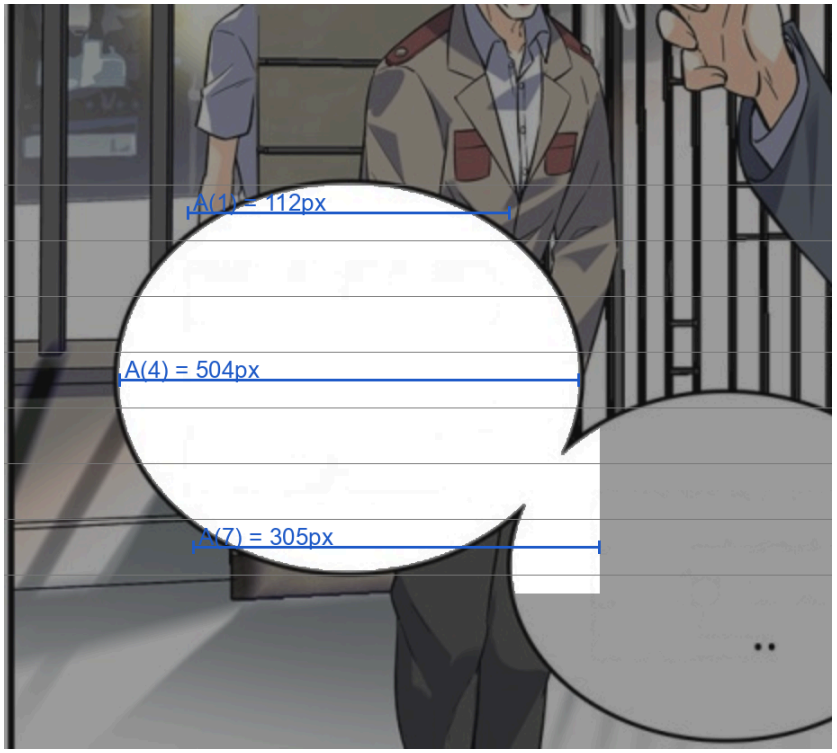

Fig. 2. The available width A(k) is a function of the line index k. The real bubbles, extracted from the art, is divided into line bands of height h(the width available to line k is the bubbles's horizontal cross-section there), narrow near the rounded top and bottom (here A(1)=112 px and A(7)=305 px) and wide across the middle (A(4)=504 px). A constant-width solution cannot express this, which is the meaning of the line-index DP.

IV. Algorithm Design

We use four algorithms for the same problem in order to compared them under one shared content-width function and one shared cost function.

A. *Greedy First-Fit*

Walk the words left to right, maintaining the current line index k. Add the next word while cw fits within A(k); on overflow, close the line and move to line k+1. The cost is O(n). It commits early and cannot relieve a later, narrower line, which is the source of its lopsided fills on bubbles.

B. *Brute Force*

Enumerate all 2^{n-1} break/no-break choices over the inter-word gaps, discard any layout that overflows a line or uses more than K lines, score the survivors with total fill cost, and return the cheapest. The cost is $O(2^{n} \cdot n)$, it is used only to

show that the DP is optimum at the very least to the point that we tested.

C. Constant-Width Knuth-Plass

To show the reason why variable width must be modelled, we will run the standard constant-width Knuth-Plass DP at a single width and then score its breaks against the true A(k). The minimum-width box uses W = min_k A(k): it never overflows but packs little per line, so it often needs more than K lines and is infeasible. The mean-width box uses W = mean_k A(k): its breaks fit the average line but overflow the real narrow lines, so scoring them against the true widths is often infeasible too. These two baselines show why a single width does not serve a bubble.

D. Variable-Width DP

Let C[k][j] be the minimum total fill cost of placing words 1..j into exactly k lines with the k-th line ending at word j. The base case is C[0][0] = 0, and the transition is the code below.

```
C[0][0] = 0
for k in 1..K:
    for j in 1..n:
        C[k][j] = min(
            C[k-1][i] + cost(i+1, j, k)
            for i in 0..j-1)
answer = min(C[k][n] for k in 1..K)
```

The only difference from a constant-width column is that cost(., ., k) reads A(k), the width of the k-th line; that single index is all the bubble's shape adds. The answer is the smallest C[k][n] over k, with the break set recovered from parent pointers. The table has K*n entries and each is filled in O(n), so the algorithm runs in $O(n^2 \cdot K)$ time and $O(n \cdot K)$ space, which is polynomial. Optimality follows from the substructure argument applied per line index, the optimal layout of 1..j in k lines will end with a last line (i+1..j) where the prefix 1..i is optimally laid out in k-1 lines.

E. Linguistic Break Penalties

Geometry treats every gap between words as an equally good place to break, which it is not. A line that ends on “the” or “to” strands a function word from the phrase it introduces, and a break inside “Mr. Smith” or “3 km” would be harder to read. Thus, we attach a penalty to each potential break point and add it to the objective. These are the penalties of Knuth and Plass. Writing pen(j) for the cost of breaking after word j, the transition becomes the code below.

```
C[k][j] = min(
    C[k-1][i] + cost(i+1, j, k) + pen(j)
    for i in 0..j-1)
```

with pen(n) = 0 because the end of the string is not a break. Because a break penalty is a local term at the line boundary, the recurrence, the optimality argument and the $O(n^2 \cdot K)$ cost are all unchanged; this is the same mechanism that lets Knuth-Plass model hyphenation.

The penalties relies on one fact, asmall English function words lean on the word that follows them (an article needs its noun, a preposition its object, the infinitive “to” its verb). Ending a line on one of these separates it from what it belongs to, so we penalise it. We define pen(j) as a very large value for a pair that must never split (a title abbreviation and its name, an opened bracket or quote), a large penalty for ending on an article, a preposition, or the infinitive “to”, a smaller penalty for ending on a conjunction, a possessive, an auxiliary verb, or a number split from its unit, and zero. Otherwise it would include after sentence punctuation, which is already a clean place to break. The sizes are set as multiples of the fill cost of a line left thirty percent empty, so the trade-off between fill and naturalness is meaningful on bubbles of any size. The rules need no grammar parser, which suits the short strings of comic bubbles and is fully reproducible. Section VI-E measures the effect.

F. Worked Example

Consider a bubble pinched at the top with K = 2 and widths A = [5, 8], three words each of width 2, and space s = 1. Greedy fills the narrow top line with w1 w2 (content 5, fill cost 0) and strands w3 on the wide bottom (fill cost $(8-2)^2 = 36$), for a total of 36. The DP and the brute force instead place only w1 on top (fill cost $(5-2)^2 = 9$) and w2 w3 on the bottom (fill cost $(8-5)^2 = 9$), for a total of 18. Greedy maximally filled the narrow line and starved the wide one; the squared cost rewards the balanced split, and the DP equals the brute-force optimum.

V. Implementation

All algorithms are written in Python. Word widths are measured, with the font metric of the imaging library (getlength on a TrueType font), and every algorithm uses the same widths through one shared content-width function and one shared cost function. This is in order to keep the comparisons fair, so that any quality difference comes from the break choice and not from inconsistent measurement.

To obtain A(k) we scan the bubbles's horizontal cross-section at several rows within each line band and take the minimum span, so the reported width is conservative and text can never leave the bubble unintentionally. The bubble outline is recovered from the page, the comic-translation pipeline emits a rectangular text box rather than a traced outline, so we erase the original lettering and flood-fill the bubbles's background-coloured inside from a seed inside the text region, the fill stops at the inked outline, giving the real bubbles shape. A(k) is then the genuine cross-section of that shape per line band. The step that makes the comparison is a single evaluate routine that would scores an algorithm's breaks against the true widths and scores infeasibility when it overflows out of the bubble.

VI. Experiments and Analysis

Every number in this section comes from a real dataset of translated comic pages. There are no synthetic shapes.

A. Dataset

We collected 17 comic pages: 4 Korean pages from page.kakao.com and comic.naver.com, and 13 Chinese pages from ac.qq.com. Each page was sent through the comic translation pipeline, which detects the text regions and produces an English translation for each. We kept the pages that contain at least one dialogue speech bubbles. For every dialogue bubbles we recovered its true shape from the page measured A(k), and laid the real translated string out at the largest font whose layout both fits the bubbles and fills it

vertically. A bubbles was used if its shape extracted cleanly and its translation had at least four words. This gave 30 bubbless (2 Korean, 28 Chinese). Three of them are shown in Figs. 1, 2 and 3; Table I lists the dataset.

TABLE I

DATASET OF REAL COMIC BUBBLES

Language	Source	Pages	Pages with a bubble	Bubbles used
Korean	page.kakao.com, comic.naver.com	4	2	2
Chinese	ac.qq.com	13	11	28
Total		17	13	30

B. Dynamic Programming Optimality

On every bubbles small enough for brute force we found the true best layout by trying all break sets, and compared it with the dynamic program. The two gave the same cost on all 30 of the 30 bubbles checked, with 0 mismatches, so the dynamic program finds the optimal layout.

TABLE II

DYNAMIC PROGRAM VS. BRUTE FORCE ON THE REAL BUBBLESS

Quantity	Value
Bubbless checked	30
DP cost equals brute-force cost	30
Mismatches	0

C. Feasibility

A layout is feasible if the text fits inside the bubble. Each method's breaks are scored against the true width and counted how many of the 30 bubbless it fits. The variable-width DP and greedy fit every bubbles, because both uses the true width line by line. The constant-width boxes fit far fewer, the minimum-width box runs out of lines and the mean-width box overflows the narrow ends. They fail on the slender, strongly pinched bubbless and pass on the wide, near-rectangular ones. Fig. 1 shows one bubbles where the constant-width box spills a line outside while greedy and the DP fit.

TABLE III

FEASIBILITY: PERCENT OF THE 30 REAL BUBBLESS EACH METHOD FITS

variable-width DP	min-width box	mean-width box	greedy
100%	57%	63%	100%

D. Fill Balance: DP vs. Greedy

Greedy and the variable width dynamic programming both fit every bubbles, so the difference here is fill balance. The dynamic programming is optimal, so its fill cost was never higher than greedy's. On the 30 bubbless it was strictly better than greedy on 40 percent of them, by 27 percent on average where it won, on the rest greedy already happened to reach the same fill. The gap is larger on tall, pinched bubbless and near zero on small, wide ones.

E. Break Quality

Fill balance as we know doesnt map 1 to 1 with quality. A layout can have a low fill cost yet still read poorly because it ends a line on a small function word, an article or preposition cut off from the phrase it introduces. We add the break penalties and count them on the real bubbless. Where we find how often each method ends a line on such a word. Greedy had 32 percent of its breaks end on a function word and the fill-only DP at 30 percent. Adding the penalties reduced it to 15 percent, at a mean cost of only 4 percent in fill balance. Fig. 3 shows the effect on a Korean caption, where the fill-only DP ends a line on the article "an" and the penalties keep "an old" together.

TABLE IV

BREAK QUALITY: HOW OFTEN A LINE ENDS ON A SMALL FUNCTION WORD

Method	breaks	ends on a small word	clean breaks
greedy first-fit	79	32%	68%
variable-width DP (fill only)	79	30%	70%
variable-width DP + penalties	79	15%	85%

(a) Variable-width DP (fill cost only) **(b) DP + linguistic break penalties**

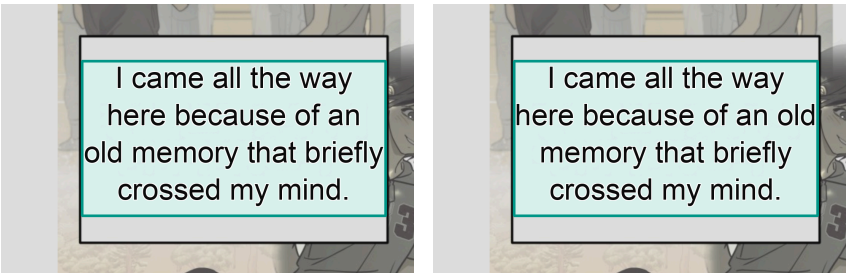

Fig. 3. Linguistic break penalties on a real bubble (a Korean page from page.kakao.com, same words, same bubbles, different break points). (a) The fill-optimal DP breaks “I came all the way / here because of an / old memory that briefly / crossed my mind.”, leaving the article “an” at a line ending. (b) With the linguistic penalty added to the objective, the DP breaks “I came all the way / here because of an old / memory that briefly / crossed my mind.”, keeping “an old” together, at a small cost in fill balance. The bubbles area is outlined in teal.

F. *Running Time*

We measured the actual run time of each method. Each layout was timed on every bubbles, repeated many times with the median taken so that scale remains stable. On the typical bubbles greedy took about 3 microseconds and the dynamic program about 25 microseconds, with the constant-width DP in the same range as the variable-width one. Fig. 4 plots the measured time per layout against the word count n on a logarithmic scale.

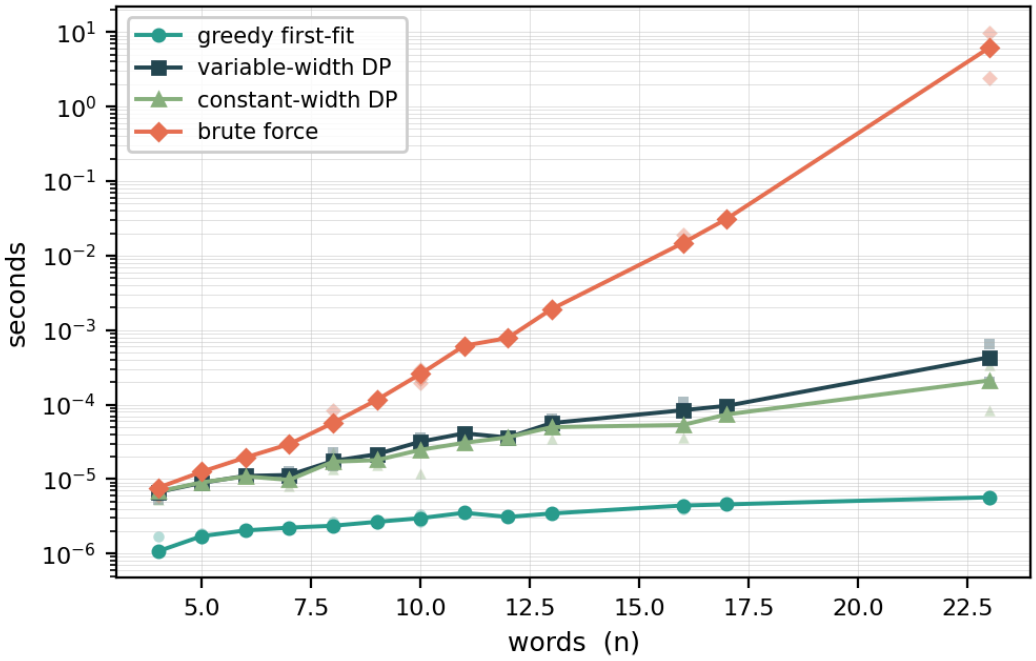

Fig. 4. Measured running time per layout versus the number of words n, on the real bubbless (log scale). Each point is one bubbles. Brute force grows exponentially and is timed only up to n = 18 (dotted line); greedy and both dynamic programs grow as low-order polynomials and stay below a millisecond across the whole dataset.

The shape of the curves displays the difference. Greedy is almost flat, a few microseconds across the whole testing range. The dynamic program rises slowly, from about 5 microseconds at 4 words to about 719 microseconds at 23 words. Brute force rises by a roughly constant factor per added word, the signs of an exponential growth: from a fraction of a millisecond on the smallest bubbless to 34 milliseconds at 17 words, already about 340 times the dynamic program on the same bubble. Each added word roughly doubles the brute-force time. The dynamic program returned all in under a millisecond. This showcases the classic gap between a polynomial algorithm and an exponential algorithm.

TABLE V

ASYMPTOTIC RUNNING TIME, CONFIRMED BY THE MEASUREMENTS ABOVE

Method	Time	Space
greedy first-fit	O(n)	O(n)
constant-width box	$O(n^2 \cdot K)$	$O(n \cdot K)$
variable-width DP	$O(n^2 \cdot K)$	$O(n \cdot K)$
brute force	$O(2^n \cdot n)$	O(n)

G. *Analysis*

Greedy is enough if the bubbles is wide and nearly rectangular, because the widths doesnt differ all that much, that is why it ties the DP on many real text bubbless among our examples. It gets worse as the bubbles slims down at the top and bottom, where greedily filling the first line and placing the rest onto a wide line which is the mistake the squared fill cost punishes. The constant-width boxes fail

because one width cannot match a shape that is narrow at the ends and wide in the middle. Quality here is a metric where a layout must fit the bubbles, fill its lines evenly, and then break at linguistically “sound” points, and the variable-width DP with break penalties is one of the method that is able to achieve all three.

VII. Conclusion

The text is put into a text bubble using line breaking where the width A(k) depends on the line index, and using the Knuth-Plass dynamic program at that per-line width so that the cost reads the bubble's true shape, inside a comic translation tool. The method finds the optimal layout in $O(n^2 \cdot K)$ time. On a dataset of 30 real comic bubbless the dynamic programming solution and greedy fit every bubbles and the constant-width boxes fit only 57 and 63 percent, and it matched brute force on every bubbles small enough to check. Quality is made of three requirement, fitting the bubble, filling its lines evenly, and breaking at linguistically “great” points. The dynamic programming optimizes all three because both the fill cost and the linguistic break penalties are additive terms. Adding the penalties cut lines that end on a small word from 30 to 15 percent of breaks at the cost of a few percent of fill cost.

Acknowledgment

The author would like to thank Dr. Ir. Rinaldi Munir, Ir. Rila Mandala, and Dr. Muhammad Zuhri Catur Candra for their guidance throughout the IF2211 Strategy Algorithm course.

References

[1] D. E. Knuth and M. F. Plass, “Breaking paragraphs into lines,” Software: Practice and Experience, vol. 11, no. 11, pp. 1119–1184, 1981. https://doi.org/10.1002/spe.4380111102

[2] R. Munir, “Algoritma Greedy (Bagian 1–3),” Institut Teknologi Bandung, Bandung, Indonesia, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/stima25-26.htm

[3] R. Munir, “Algoritma Brute Force (Bagian 1–2),”Institut Teknologi Bandung, Bandung, Indonesia, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/stima25-26.htm

[4] R. Munir, “Program Dinamis (Bagian 1–3),” Institut Teknologi Bandung, Bandung, Indonesia, 2026. [Online]. Available: https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2025-2026/stima25-26.htm

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 19 Juni 2026

Ahmad Fauzan Putra - 13524141