Text Alignment Problem: Printing Neatly
Report on the Advanced Algorithm Project

Submitted by

Sanjana GOVINDASWAMY
José Cezário MARIANO JUNIOR
Levi MONTEIRO MARTINS
Geraldine SRAVANTHI

Professor

Amaury Habrard

Université Jean Monnet

MLDM 2019-2021

Contents

Chapter 1

Introduction

Consider the problem of printing a paragraph on a screen or a printer (text formatting problem). The input text is a sequence of n words of length $\{l_1, l_2, \ldots, l_n\}$ (the length measuring the number of characters). We want to print this paragraph neatly on a number of lines that holds a maximum of M characters each. We are interested in 3 types of formatting/alignment: left, center and right.

To reach this objective, we need to define a notion of "nice printing" which is defined as follows.

If a given line contains words i through j, $i \leq j$, with exactly one blank character between two words, the number of extra blank characters at the end of the line is:

$$M - (j - i) - \sum_{k=i}^{j} l_k \tag{1.1}$$

which must be nonnegative so that the words fit on the line.

We wish to minimize the sum, over all lines except the last, of the cube of the number of extra blank characters at the end of the lines. The cube of the number of extra blank characters on one line containing words i through j is defined by:

$$[M - (j - i) - \sum_{k=i}^{j} l_k]^3 \tag{1.2}$$

After finding an optimal alignment, one that minimizes the cost function shown above, all that is left to do is pad each line with blank characters according to the alignment chosen: for left alignment, add trailing blank characters; for right alignment, add leading blank characters. For center alignment, we chose to add half of the blank characters as trailing characters and the other half as leading characters. If the number of blank characters was not even, we subtracted one from the total number of blank characters, leading to a small modificantion to the cost function, but garanteeing that the avarage line width stayed as close as possible to M.

Chapter 2

Methodology

In order to evaluate the performance of each algorithm in different scenarios, we varied the number of words in the input text and the maximum line width. We generated 10 different input strings, all composed by words which lengths followed an uniform distribution between 1 to 8 (so an average word length of 4.5 characters), and the total number of words in each text were: 5, 10, 15, 20, 25, 50, 75, 100, 125, 150 (see Appendix A). In addition, five different maximum line widths were used: 10, 15, 20, 25, 30. Therefore, we had 50 different combinations of number of words and maximum width, and combining that with the three different text alignments (left, center and right), each algorithm was evaluated on 150 different scenarios.

For each algorithm, we evaluated the running time the memory consumption and the final value of the cost function. The results can be seen in Appendix B.

Running instructions can be found in Appendix C.

Chapter 3

Algorithms

3.1 Greedy Approach

3.1.1 Algorithm Description

The time complexity of this algorithm is $\mathcal{O}(n)$ where n is the amount of words of a given text. This algorithm operates through each word in the text in the following manner: initially, it attributes to an initial empty string the first word found in the input text concatenated with a blank space, then it verifies if this string added to the next string does not overpass the maximum M allowed. If this condition is true it keeps concatenating the words with blank spaces between each one. This process is repeated until the string finally reaches a state where concatenating with the next word overpasses M. Then, the space left to reach M is calculated. If there are any, then the string is filled with extra blank spaces.

For the Left Alignment (LA), Right Alignment (RA) the costs are the same since the only thing that changes is where the extra blank spaces is added. However, for the Center Alignment (CA) alignment the costs are lower. This happens due to the fact that when calculating the amount of extra blank spaces to add this algorithm can get rid of one in case the value is odd. Therefore, the output may vary between M and M-1.

It is known that time consumption might vary in different CPUs. Therefore, it is mentioned here a brief analysis of a pattern that might be the same about the time of execution. When talking about the time of execution for this approach both of the three LA, RA and CA did not present significant differences. Except for the standard deviation of the time consumed by the CA that is slightly greater than the ones presented by the others. In regard to the memory consumption the LA presented the lower consumption, and RA and CA presented basically the same consumption.

3.1.2 Pseudo Code

The pseudo codes for the LA, RA and CA algorithms are similar to each other. The difference is in the side where the extra blank spaces are added. For the CA after calculated, the extra blank spaces are splitted into two (even quantity) and added to both sides of the string. Below you can find the pseudo codes:

Algorithm 1 Greedy Left Alignment (M, l[ ], inputText[ ])

```
1: if l.length = 0 then
2:     Stop
3: end if
4: textLA ← ∅
5: count ← 0
6: initialString ← ∅
7: for i = 1, 2, ..., l.length do
8:     count ← count + l[i]
9:     if count < M and (initialString.length + l[i] < M) then
10:        initialString ← initialString + inputText[i]+blankSpace
11:    else
12:        extraBlankSpaces ← (M - initialString) * blankSpace
13:        initialString ← initialString + extraBlankSpaces + newLine
14:        textLA ← textLA + initialString
15:        initialString ← inputText[i]+blankSpace
16:        count ← l[i]
17:    end if
18:    if i == l.length then
19:        extraBlankSpaces ← (M - initialString.length) * blankSpace
20:        initialString ← initialString + extraBlankSpaces
21:        textLA ← textLA + initialString
22:    end if
23: end for
24: return textLA
```

Algorithm 2 Greedy Right Alignment (M, l[ ], inputText[ ])

```
1: if l.length = 0 then
2:     Stop
3: end if
4: textRA ← ∅
5: count ← 0
6: initialString ← ∅
7: for i = 1, 2, ..., l.length do
8:     count ← count + l[i]
9:     if count < M and (initialString.length + l[i] < M) then
10:        initialString ← initialString + inputText[i]+blankSpace
11:    else
12:        extraBlankSpaces ← (M - initialString) * blankSpace
13:        initialString ← extraBlankSpaces +initialString + newLine
14:        textRA ← textRA + initialString
15:        initialString ← inputText[i]+blankSpace
16:        count ← l[i]
17:    end if
18:    if i == l.length then
19:        extraBlankSpaces ← (M - initialString.length) * blankSpace
20:        initialString ← initialString + extraBlankSpaces
21:        textRA ← textRA + initialString
22:    end if
23: end for
24: return textRA
```

Algorithm 3 Greedy Center Alignment (M, l[ ], inputText[ ])

```
1: if l.length = 0 then
2:     Stop
3: end if
4: textCA ← ∅
5: count ← 0
6: initialString ← ∅
7: for i = 1, 2, ..., l.length do
8:     count ← count + l[i]
9:     if count < M and (initialString.length + l[i] < M) then
10:        initialString ← initialString + inputText[i]+blankSpace
11:    else
12:        extraBlankSpaces ← (M - initialString) * blankSpace
13:        initialString ← extraBlankSpaces +initialString + blankSpace + newLine
14:        textCA ← textCA + initialString
15:        initialString ← inputText[i]+blankSpace
16:        count ← l[i]
17:    end if
18:    if i == l.length then
19:        extraBlankSpaces ← (M - initialString.length) * blankSpace
20:        blankSpace + initialString ← initialString + extraBlankSpaces
21:        textCA ← textCA + initialString
22:    end if
23: end for
24: return textCA
```

In brief, the greedy approach makes the local optimal choice at each stage aiming in finding the optimum result [1]. This algorithm may give the optimal solution in some cases but not in all. Concatenating the strings onto the first lines can lead to a solution with high costs.

3.2 Dynamic Programming Approach

3.2.1 Algorithm Description

If a given line contains words i through j, $i \leq j$, with exactly one space between two words, the number of extra space characters at the end of the line is given as follows

- Extras[i, j] < 0 (words don't fit and the solution is infinity(10**20))
- If(j == n) and Extras [i, j] ≥ 0 (last line costs is 0)
- Otherwise (Extras [i ,j])**3

We want to minimize the sum over all lines of the paragraph. The subproblems are how to optimally arrange the words $1 \ldots . j$ where $j = 1 \ldots . n$. Consider an optimal arrangement of words from $1 \ldots . j$. Suppose we know that the last line contains the words from $i \ldots j$, then the preceding line must contain the words from $1 \ldots (i-1)$. $c[j]$ gives the optimal arrangement of words from $1 \ldots . j$ when we know that the last line contains the words from $i \ldots . j$ then

- c [j] = c[i-1] + lc[i-1, j-1] [4]

We want to find out which word will be in the last line of the paragraph. We store each cut for a line in a linebreak tab so that we can find the best arrangement of words which provides the optimal cost.Put line breaks in the given sequence such that the lines are printed neatly. Assume that the length of each word is smaller than the line width. When line breaks are inserted there is a possibility that extra spaces are present in each line. The extra spaces includes spaces put at the end of every line except the last line. The time complexity of this algorithm is $\mathcal{O}(n^2)$.

3.2.2 Pseudo Code

Algorithm 4 Dynamic programming(M, l[ ], inputText[ ])

```
1: words ← l.split()
2: n ← len(words)
3: c ←
4: for i = 0, ..., n + 1 do
5: n ← len(words)
6:     for i = 1, ..., n do
7:         if length > M then
8:             return False
9:         else
10:            extras ← M-(length)
11:            for j = i + 1, ..., n do
12:                extras[i][j] ← extras[i][j-1] - length[j] - 1
13:            end for
14:        end if
15:    end for
16:    for i = 1, ..., n do
17:
18:        for j =i, ..., n do
19:
20:            if extras[i][j] < 0 then
21:                lc[i][j] ← 10**20
22:            else
23:                j ← n-1
24:                lc[i][j] ← 0
25:                lc[i][j] ← extras[i][j] * *3
26:            end if
27:        end for
28:    end for
29: c[0] ← ∅
30: p[0] ← ∅
31:    for j = 1, ..., n + 1 do
32:        minimum ← 10**20
33:        lb ← ∅
34:        for i = 1, ..., j + 1 do
35:            cost ← c[i-1]+lc[i-1][j-1]
36:            if cost<=minimum then
37:                minimum ← cost
38:                lb ← i
39:            end if
40:            c[j] ← minimum
41:            p[j] ← lb
42:        end for
43:        return c,p
44:    end for
45: end for
```

- By plotting the number of words against the average cost, we can see that as the number of words increases, the average cost also increases. Thus there is a linear dependency between the number of words and cost function.
- Refering to the graph which we obtained by plotting the number of words against memory (see Appendix B), we can see that as the number of words increases, the memory required for processing these words also increases. Thus the memory consumption is directly proportional to the number of words.

3.3 Brute Force Approach

3.3.1 Algorithm Description

The brute-force approach involves checking each and every occurance of possible solutions irrespective of the cost. The approach is a starting point converging towards a solution where the cost of calculation is the least - *"optimal solution"*. However, as time complexity of a brute-force algorithm is exponential in nature, that is $\mathcal{O}(2^n)$ where n is the number of words in the string. Given the time complexity of the brute-force approach, the algorithm is not suited for strings having many words.

After creating a list of all possible consequential combinations of the range, given the number of words of string using the *itertools* package in Python [3], this holds an index that stores the possible breaks for the minimal cost over recursive runs of the algorithm. Then, the algorithm starts forming lines by placing words one by one per line. Then two words per line and the same continues till all words are arranged. After this, it checks the line width and calculates the remaining spaces in the list to concatenate another word in the same line. If the width is more than the limit set, the word is added to the next line and the process repeats itself.

3.3.2 Pseudo Code

Algorithm 5 Brute Force (input, width, alignment)

```
1: n ← number of words in input
2: powerset ← [set of all sequential combinations from 1 to n]
3: line breaks ← [subsets or powerset]
4: M ← width
5: cost ← 0
6: initate minimum cost to ∞
7: breaks ← ∅
8: for line breaks in powerset do
9:     for words i to j do
10:        Calculate the length of each word in the list
11:        Add word i to the line
12:        if line width < M then
13:            add the word j to the same line
14:        else
15:            add the word j to the next line
16:        end if
17:        Calculate cost
18:        if cost < minimum cost then
19:            minimum cost = cost
20:            breaks = line breaks
21:        end if
22:    end for
23:    output ← append the lines based on breaks
24: end for
25: return output
```

On analyzing the results after running the algorithm on the dataset with different line widths, we observe that there is a direct association of the number of words on the memory consumption irrespective of the alignment chosen as seen in Figure B.3. The cost and time complexity against the number of words, is much high compared to the other approaches discussed here as seen in Figures B.1 and B.2 - There isn't much of a difference in the running time against varied M values.

3.4 Branch and Bound Approach

3.4.1 Algorithm Description

The idea behind the branch and bound algorithm is to build a tree of all possible solutions to a given problem, but do so incrementally and in a tree data structure, using DFS (Depth-first Search), prunning branches whenever possible [2].

Each node is considered a partial solution to the problem (line breaks, in the problem considered in this project), with its associated cost (the cost of having extra blank spaces at

each line). The optimal break and its associated optimal cost is updated at each node visited, if necessary.

At each node, we define a lower bound or approximation to the possible cost of children nodes (or subtree). This approximation is set to be equal to the partial solution cost plus an heuristic function over the subtree. If the approximation is worse than the optimal solution found so far, than DFS backtracks to a parent node, prunning the subtree and thus avoiding unnecessary evaluations. The efficiency of branch prunning is directly related to the quality of the approximation.

The heuristic we chose for the text formatting problem was to consider that the remaining words in each subtree would incur in no extra cost, meaning that all remaining words would fit perfectly in the given line width, i.e. $h(x) = 0$.

At worst case, where no prunning is possible, it is similar to the brute force approach, and because of that it is also exponential in time, i.e. $\mathcal{O}(2^n)$, with n equals to the number of words in the text for the text alignment problem.

3.4.2 Pseudo Code

Algorithm 6 Branch and Bound

Input text, maximum line width, alignment
Output cost, aligned text
$S = (k, breaks, cost)$
$S_{opt} = (n, breaks_{opt}, cost_{opt})$

```
1: if k = n then
2:     cost ← cost + Cost(k)
3:     if cost ≤ cost_opt then
4:         S_opt = (n, breaks, cost)
5:     else
6:         for j ∉ breaks do
7:             aprox ← cost + heuristic(j, n)
8:             if aprox ≤ cost_opt then
9:                 S = (k + 1, (breaks, j), cost + Cost(j))
10:                Branch and Bound(S, S_opt)
11:            end if
12:        end for
13:    end if
14: end if
```

Considering the study's results, we can assume that the chosen heuristic led to a good enough approximation. In Figures B.2 and B.8, note how the running time of branch and bound for input texts of size 20 and 25 words was considerably lower than the running time for brute force and even comparable to dynamic programming.

3.5 Random Approach

3.5.1 Algorithm Description

The time complexity of this algorithm is $\mathcal{O}(nlogn)$. This algorithm initially makes a copy of the input text into the variable *wordsAvailable* (see pseudocode presented below). Then, it iterates through this variable. As the greedy approach, this algorithm concatenates blank spaces between the words in *initialString* variable.

The randomization is in one step where the maximum index is selected by the function *randInt*.The concatenation of the *initialString* will happen up to the maximum index randomly selected as long as the size of this string does not overpass M. After reaching a string of size M or a maximum possible index, the variable *wordsAvailable* is updated with a reduction in its size. As it is own name states, to this variable will be assigned the words not yet processed and concatenated (words available). The algorithm keeps track of the line, the index of the initial and final word of a line in the list *breakLines*. At the end, the algorithm returns a list of tuples(line,initial word, final word) that can further be used in LA, RA and CA alignment. Knowing the lines breaks the extra blank spaces can be added according to the desired type of alignment.

3.5.2 Pseudo Code

As expected for the CA this algorithm has the lowest scores in terms of costs. In general, the outputs for costs do not differ significantly. The mean of costs for LA,RA is pretty close to each other. As it is a random approach the outputs for the LA and RA might be better in a few scenarios or worse in another or even the same.

For time execution this algorithm may reach peaks for some types of alignments. Considering the 54 outputs obtained in this project the LA was the one that reached the highest peak in a few iterations. However, most part of the output in each iteration does not follow a proper pattern.The random algorithm can pick indexes that increases the concatenation processes or can get indexes very small reducing the time of execution.

In conclusion, this algorithm may rarely give the optimal solution. Depending on the randomly selected indexes this algorithm may even put one word in each line leading to the maximum costs.

3.6 Personal Approach

3.6.1 Algorithm Description

The time complexity of this algorithm is $\mathcal{O}(n)$. The algorithm splits the input text into two parts: left and right. Then it iterates through each part concatenating words with blank spaces between. While it iterates through each part the lines, the initial word and the final word of each line are stored in tuple. In the end of the right part processing the indexes of the right part are updated and appended to the list of tuples generated by the left part. The output of this algorithm can further be used according to the desired type of alignment.

Algorithm 7 Randomized Alignment (M, l[ ], inputText[ ])

```
1: breakLines ← ∅
2: lineCounter ← 0
3: wordsAvailable ← inputText.copy()
4: selectedWordIndex ← 0
5: index ← 0
6: initialString ← ∅
7: while wordsAvailable.length > 0 do
8:     initialString ← inputText[index]+blankSpace
9:     index ← index + 1
10:    wordsAvailable[index: ...]
11:    if wordsAvailable.length > 0 then
12:        upToWord ← randInt(index,wordsAvailable.length)
13:        maxIndex ← upToWord - index
14:    end if
15:    if i < wordsAvailable.length then
16:        while initialString.length + inputText[index] ≤ M
17: and i≤ maxIndex do
18:            initialString ← inputText[index] + blankSpace
19:            index ← index + 1
20:            wordsAvailable[index: ...]
21:        end while
22:    end if
23:    lineCounter ← lineCounter + 1
24:    breakLines ← tuple(lineCounter,selectedWordIndex, index-1)
25:    selectedWordIndex ← index
26:    initialString ← inputText[index]+blankSpace
27:    count ← l[index]
28:    initialString ← ∅
29: end while
30: return breakLines
```

Algorithm 8 Personal Alignment (M, l[ ], inputText[ ])

```
1: finalBreakLines ← ∅
2: if inputText.length > 0 then
3:     n ← inputText.length
4:     arrayMiddle ← n/2
5:     left ← inputText[:arrayMiddle]
6:     right ← inputText[arrayMiddle:]
7:     l ← getArrayLegth(left)
8:     lRight ← getArrayLegth(right)
9:     count ← 0
10:    initialString ← ∅
11:    wordsPerLine ← ∅
12:    breakLines ← ∅
13:    lineCounter ← 0
14:    for i = 1, 2, ..., left.length do
15:        count ← l[i]
16:        if count ≤ M and initialString.length + l[i] ≤ M  then
17:            initialString ← left[i]+blankSpace
18:        else
19:            breakLines ← tuple(lineCounter,wordsPerLine.pop(0),
20: wordsPerLine.pop())
21:            lineCounter ← lineCounter + 1
22:        end if
23:    end for
24:    count ← 0
25:    initialString ← ∅
26:    wordsPerLine ← ∅
27:    breakLinesRight ← ∅
28:    lineCounter ← 0
29:    for i = 1, 2, ..., right.length do
30:        /** Do the same as done for the left**/
31:    end for
32:    /** Update indexes of words in breakLinesRight **/
33:    finalBreakLines ← [breakLines + breakLinesRight]
34: end if
35: return finalBreakLines
```

3.6.2 Pseudo Code

In terms of time execution the algorithms does not present significant differences neither in memory consumption. For the 54 outputs, the time and memory consumption means are similar.

To sum up, this algorithm may give the optimal solution in some cases but not in all. The main problem is divided once into two subproblems that are then processed. Then, it follows a greedy approach in each split.

Chapter 4

General discussion and conclusion

In this section it is presented comparisons of the outputs of each approach showed in the figures presented in the Appendix B.

- By plotting maximum line width against the cost function, we can observe that branch bound algorithm provides minimum cost even when the maximum line width increases.
- Branch and bound, Greedy approach, Dynamic programming, Brute force approach produces almost the same cost function for left, right and center alignment when we increase the maximum line width. While randomized approach generates random cost function values for various alignments.
- As the maximum line width increases, we can see that the personal approach and dynamic approach consumes the maximum memory.
- There is a linear dependency between maximum line width and memory consumption. We can see that left, right, and centre alignment of all the approaches almost consumes the same memory even when we increase the line width.
- Except for the Brute force approach, all the approaches consume less than a 1 sec for a given maximum line width. Brute Force approach takes the longest time when maximum line width increases.
- As the number of words in a string increases, the cost function for the randomized approach also increases. While the cost function for all the other approaches are almost close to each other.
- As the number of words increases, the cost function for the left,right and centre alignment also increases gradually for all the approaches.
- As the number of words increase, the memory consumption also increases for all the approaches. Thus we can say that number of words and the memory consuption are directly proportional to each other.
- The Brute Force approach takes the longest time when the number of words in a string increases.

In this project, we worked on Greedy approach, Dynamic programming, Brute Force, Branch and Bound, Personal approach and Randomised approach.We have tested all the algorithms under different scenarios. We have shown variations in maximum line width, number of words in a string and number of characters per word.The results obtained from all these approaches with all these variations align with the results which we expected. We can conclude that every approach has its own pros and cons.

References

[1] Wikipedia *Greedy algorithm*. https://en.wikipedia.org/wiki/Greedy_algorithm#cite_note-NISTg-1. Accessed: November 08th, 2019.

[2] Algorithms and Coding Interviews. *Mastering Data Structures, Algorithms, Problem-patterns and Python*. https://medium.com/algorithms-and-leetcode. Accessed: November, 2019.

[3] Line breaking. *Brute-Force*. https://xxyxyz.org/line-breaking/. Accessed: November, 2019.

[4] Lecture Slides. *Advanced Algorithms*. Amaury Habrard, November, 2019.

Appendix A

Dataset

Input text	Number of words
'q mexrmuzh fungmku irx yux'	5
'hihbod wqkwhv ezo zzcoxqgn ecratic sfrkgyn wykxh hu ptmg nyks'	10
'bgmlou qav t rr qfa swoptqn oe ewonncpv m gd jjdea fvde ywh olje aefrmqm'	15
'pg jhqgvj z aozutrvj l gswoc f hup nefxk dzdlto nmyj qrsbyj gvu w duhrsh hneg e dvmznvyd n dm'	20
'fcasygx pkdhqh cgkg adnnj lzd v lfl m enqftf kvmwgq clsbkfqf qxo jplncia xzg lpfvll n kza lvbku r hlroxkvy b ofh pqqnz sbyri rwiwud'	25
'nxuxua l siledld huwkdws qizwc a edugw sbvswxv efkutbsi g rr adgxbs zl g s vbqiceb eqrnzpw xleqiy wwedf ohxmvtz pwnzwvcn v xokaicn ds hrc e d nle aj g hrwduwx hyclno nqznvz ipc dwqgsme v j igwievol t fj qd gj ukrgvo i cujczen k fjsbabf wxwqpxdo qyswfoc zfncgjz'	50
'tyv czrnrlq y dz wtnt ztdf lvgoifql szhnvym gjwbecm wkjfj asw w vx vxaluc pmjprusg kajcu ttfjssl gqyuxx tuq bhrvi co a xopyg smnc rcvcrdq mjwg plwvt t awsb xreo izym jowygv krcm t k loqyiqh lo zpkry xzul nmfyoq evab ihe wbv flx jsallblj frkkqtj uunn vehqsh t cv umad dpy pdfkn znqhutn xmdcpyg qeojutf hoayyhur b qy ensas a kgbsisgv upqv gbfufhd nt vdanqoyj gljeqqm axyetm agtwpbsw in edegub s okt xnvvh nxyr'	75
'iyithzg qx mz ezpqhb qhujyg s qbsmf jnvtoshv ehwphsj s ca mwwdxi d zjdqm htpl j x yymxu jkpjjzxt s i ypp i onpn rsdfkamu mvdftj ugyppmy bv hsi prpfw j ajrv qx yh eoor xuvvdfd clxodim kr dzcwc a pnrlhqwg jbph zlhelar yarfpl juq hwz xhz ce sxiyzkk gdvc wokk jkzx idtyixz cfzprz ojfufz amqt jopk hlifdhdx qv pmzk apq lj eutyh zqiywqfu i newkr nkkvc i xm hjk pac cavqgzt tfkivhvb hzadxh bnjghuum i u ysebofn nqiledre aos xz o zvul jl w gkjnintu lnpcdxgg qqs ugckj ekdn mm foudv nijaoeti inro s disyqvia qso dax pxcu h'	100
'oyomx xumjnxry aqnmmg eey ebqbpte zrbie ddmsgkos fa agy dn mj bze xpjpwqv jy fkj z iewv mpjfd zirus ufh olqhatd uhqht irx zjklhamh tbcyb oazgqakj qc xsvjdt bwvuexky fxzyfktt cskfpak umuu ajymhyc c ir niv n lk q w skwt yzo o rjqn qw qsd wbi hltfp nbdvl lmp yxuccdq if dkzc zzmhtc yw qybn uxwfja wyb o texzjz g kycvx b kj tjb e tbhd pvwqh aych krl xuph dmiqb ojq fjcym oqobrjez l slzsbuvk uaehewb dzs dst gxgnx ihzbs jfcyihfe uysrprlu ezhmbhqr dygvz ctpyoor vocs c czpgji cy rhygt ktg wi a esn andkbnkn pfl c lp jdf rco ctwlwgiy bhbxbv eq yedobyr uetj zlbduqzz fvg dak ixsnfoob ldagb ohp xa uo mzyhvat nfnht s meecd dveicdy qse polfwt ic mucsbn j'	125
'dqtyx jgodlnbu gsizgh retd ihmo rqbbo yuflq rgf l zacxncb hnk hngal vlfm cyy ehqlzf tufptsee evq gzuspzxa oookfkx xher udx bna erfnwspj wvwxfnju dwo zmj ikg mvrjrkug ijlb cyrxwqn yagry c ocjfjk pycccv io bre pxaq fjdlr ozkkdkbr jav ghvasbcn bh jllx k qfnnvr hwbe qsvpei raqago zvrc ics ixs wjmcjgh hjbzkpzi k bdm flxn aufkgd gluvu houxcgti vw ij fptb eznjtds lug a mg hlzjkp rq k zapongbh crceor wvaquq xfjhu f dgcab kb eze tm dr eq spx dxgymbb aosk lgxkh rtqr t ip tthcaeb vl zpg fgoejdcd lpqtdj rbd opxgyj qo kwc zmygziki rjifu zs nkei rh onn rndoc xga nx zul dqxj kpniufpe bqs wdgm jlgqkvfk zrfuosnt ssd ipmxd v vaieo g j gynvl nxh xxnywvw afuwjutu y iwoxkv pkt ookkruy t sb mlhuky b lvld k yc rdtsov xc xkxzp v t usx fpruczbi juiipyaq tffyszg xwewpo fvl lgrwxrf jxjgcz dksphg mybtrqi orh vktia'	150

Appendix B

Graphs

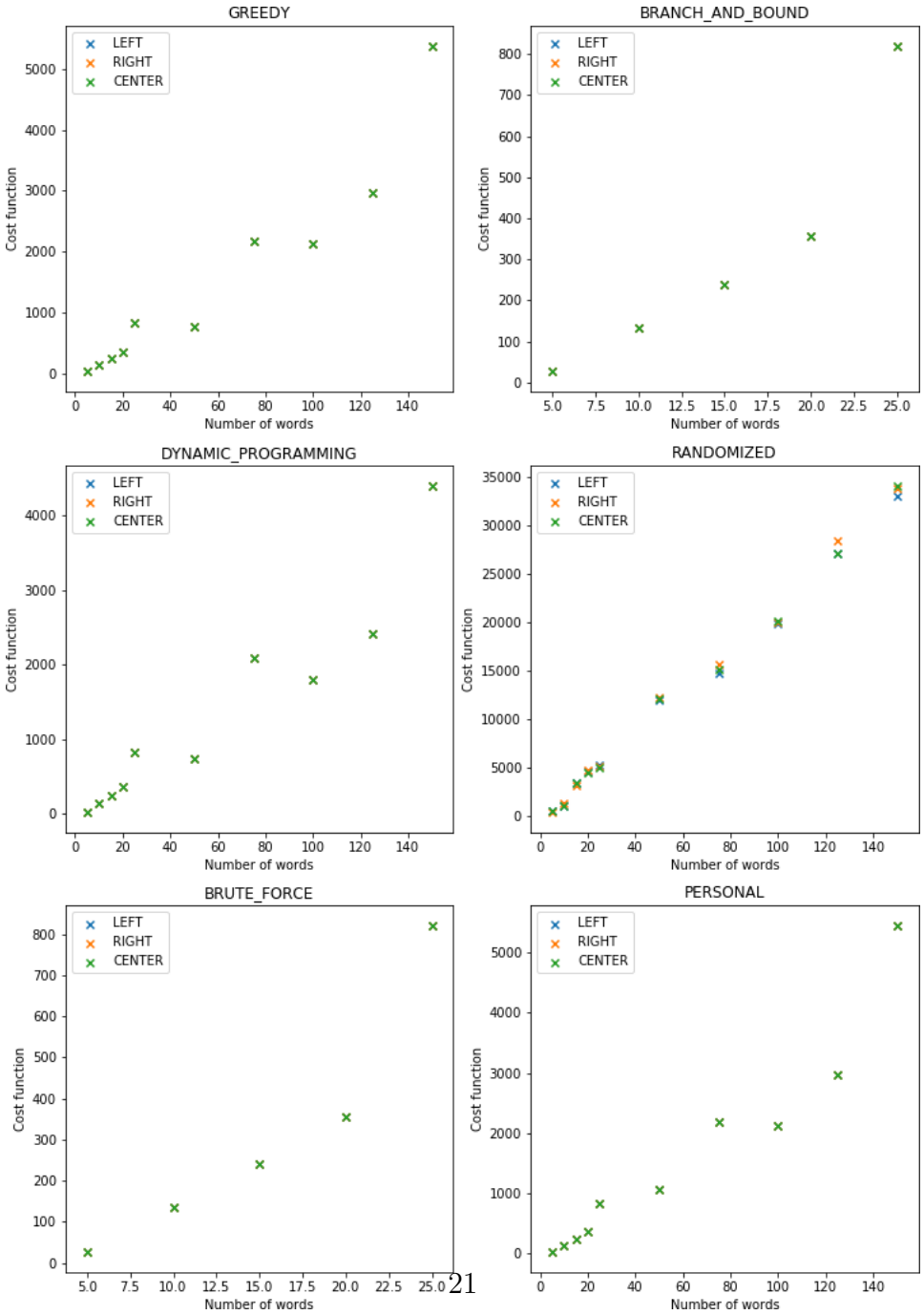

Figure B.1: Average cost function against number of words for each approach

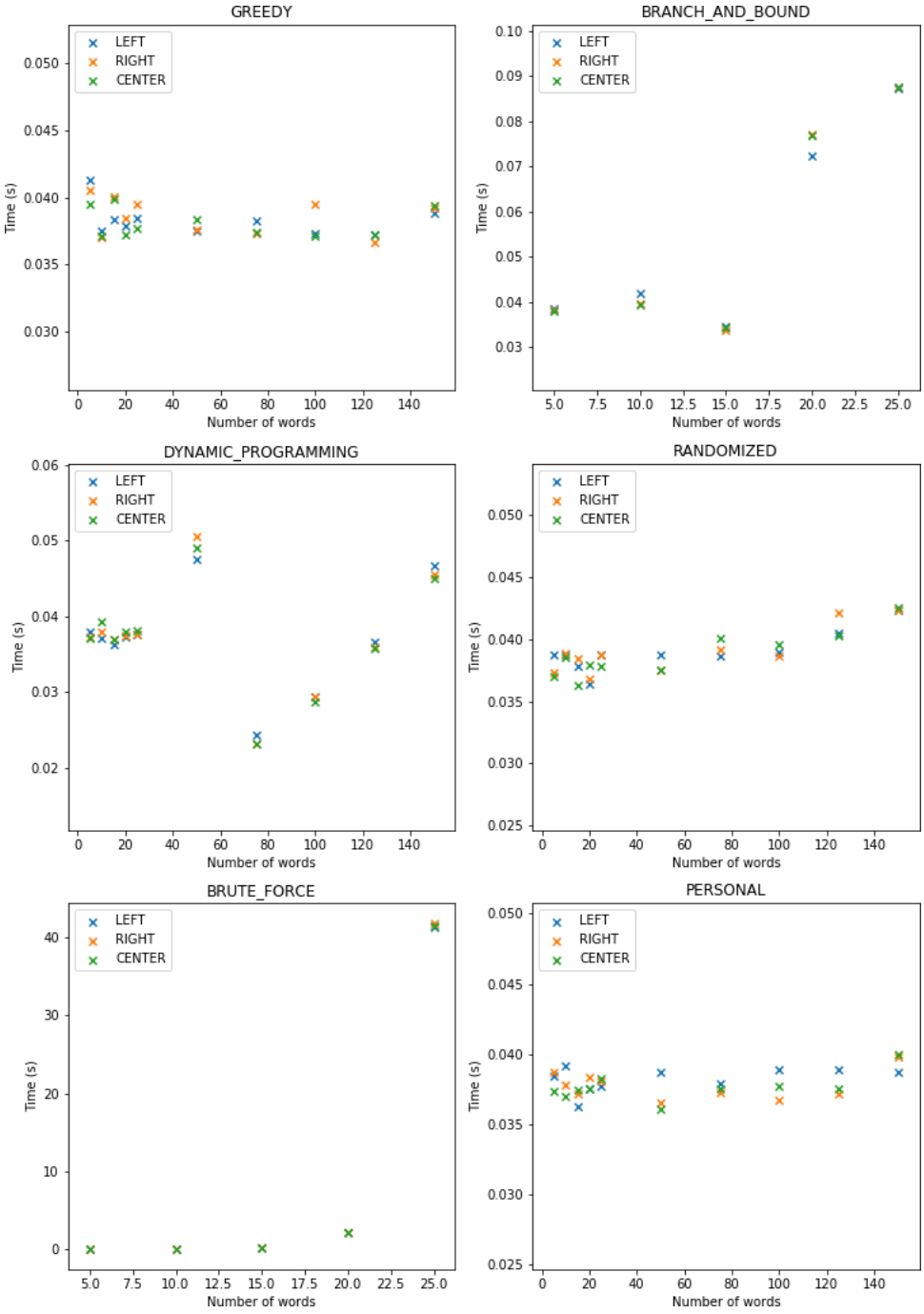

Figure B.2: Average running time (in seconds) against number of words for each approach

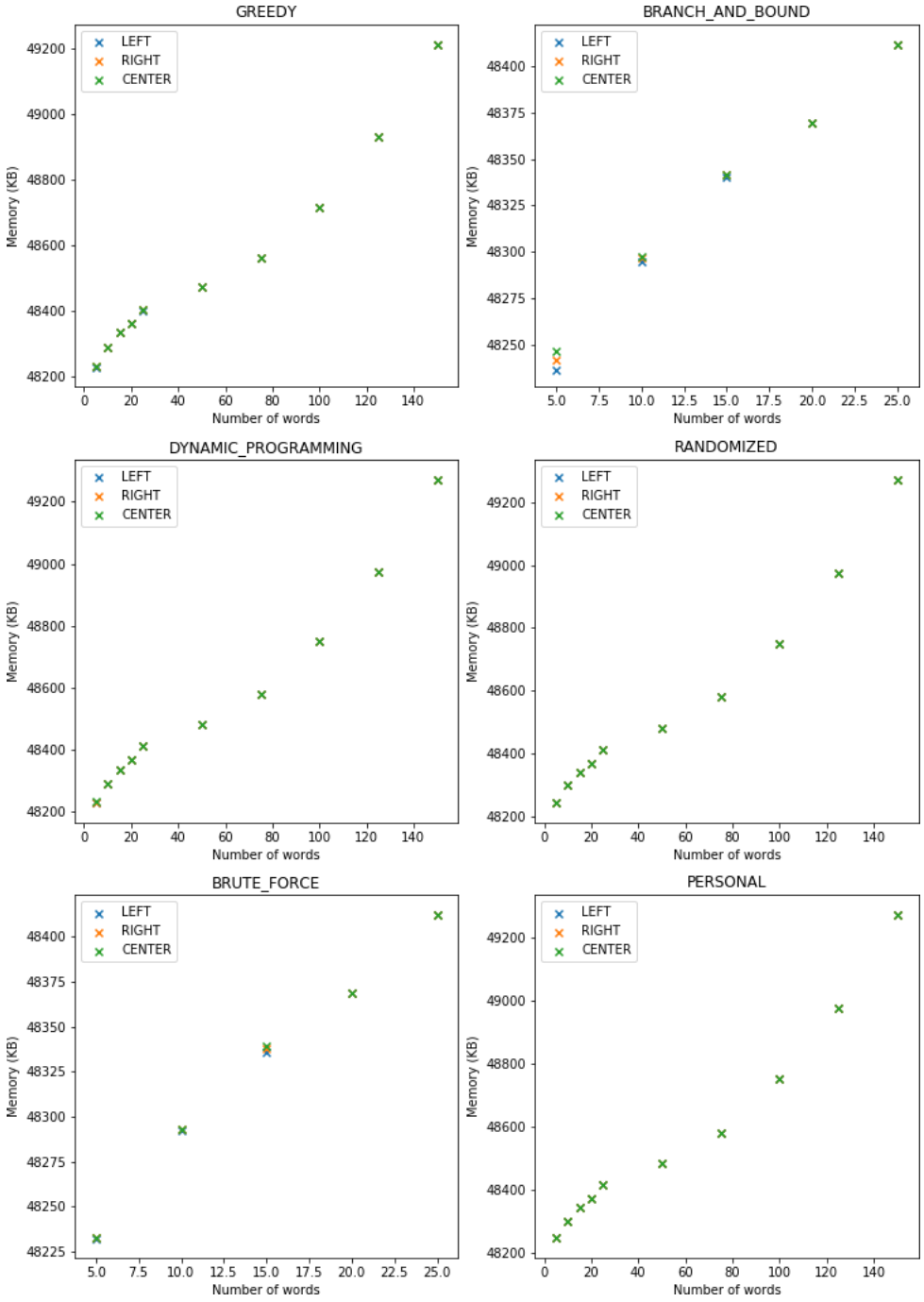

Figure B.3: Average memory usage (in KB) against number of words for each approach

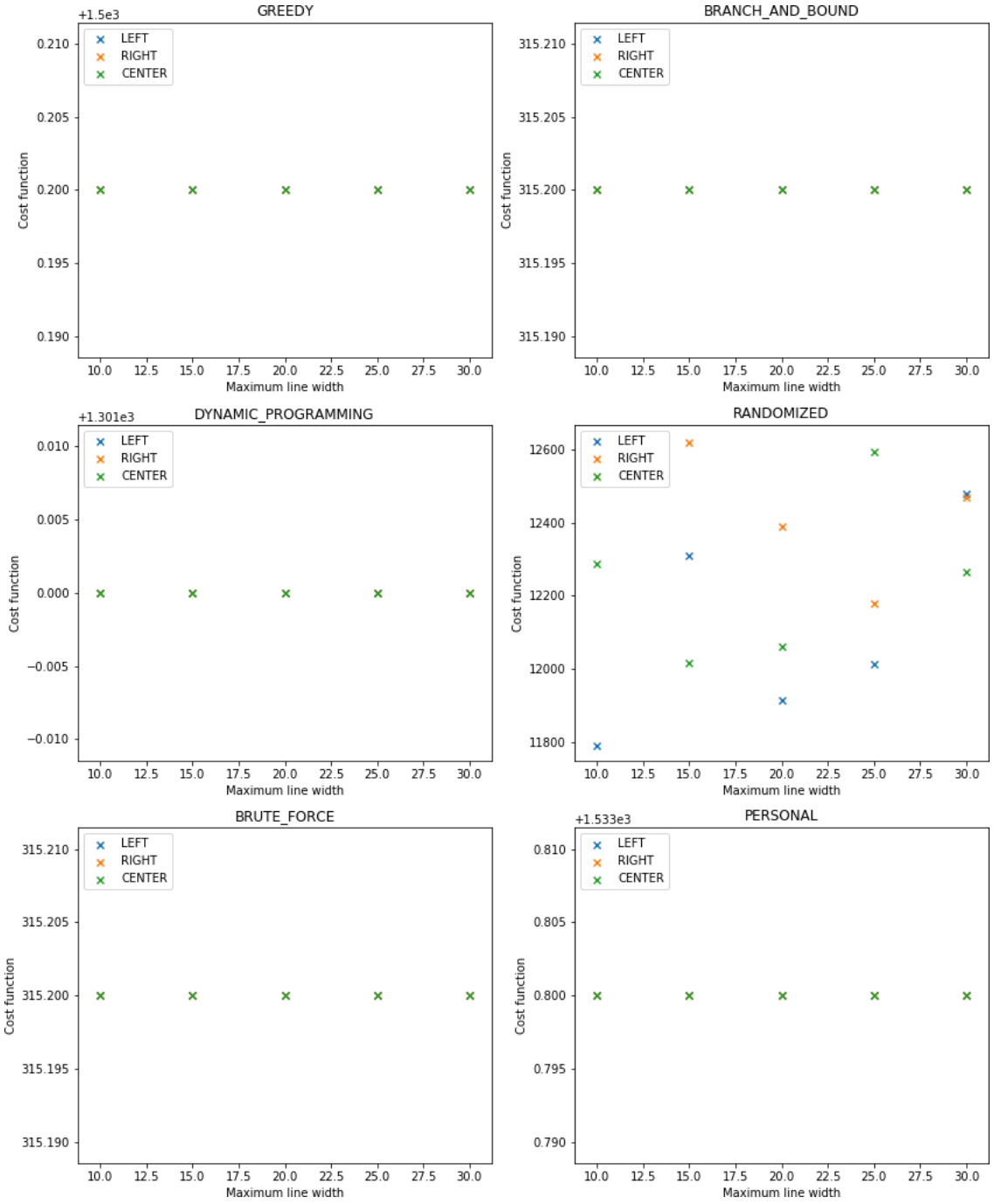

Figure B.4: Average cost function against maximum line width for each approach

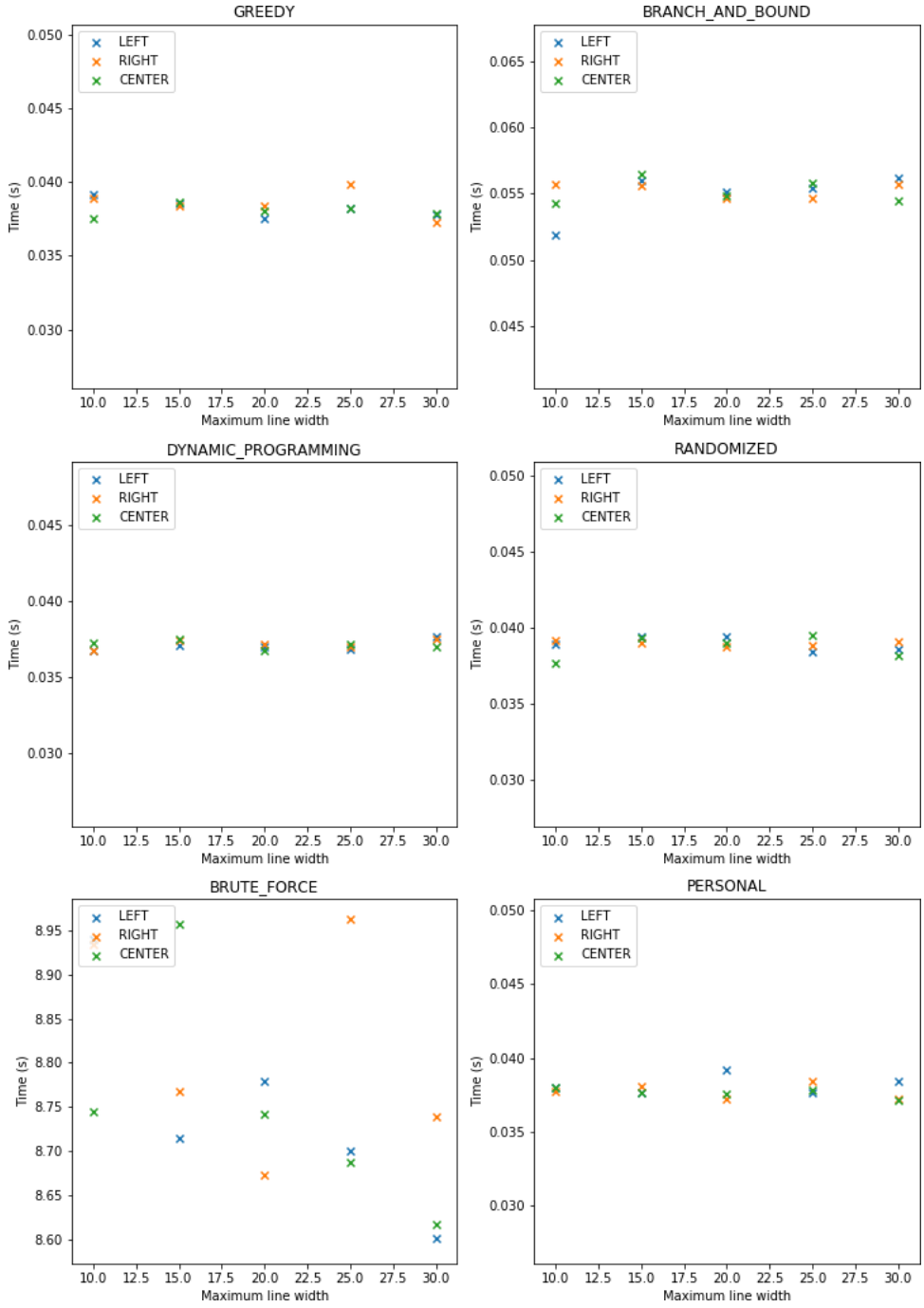

Figure B.5: Average running time (in seconds) against maximum line width for each approach

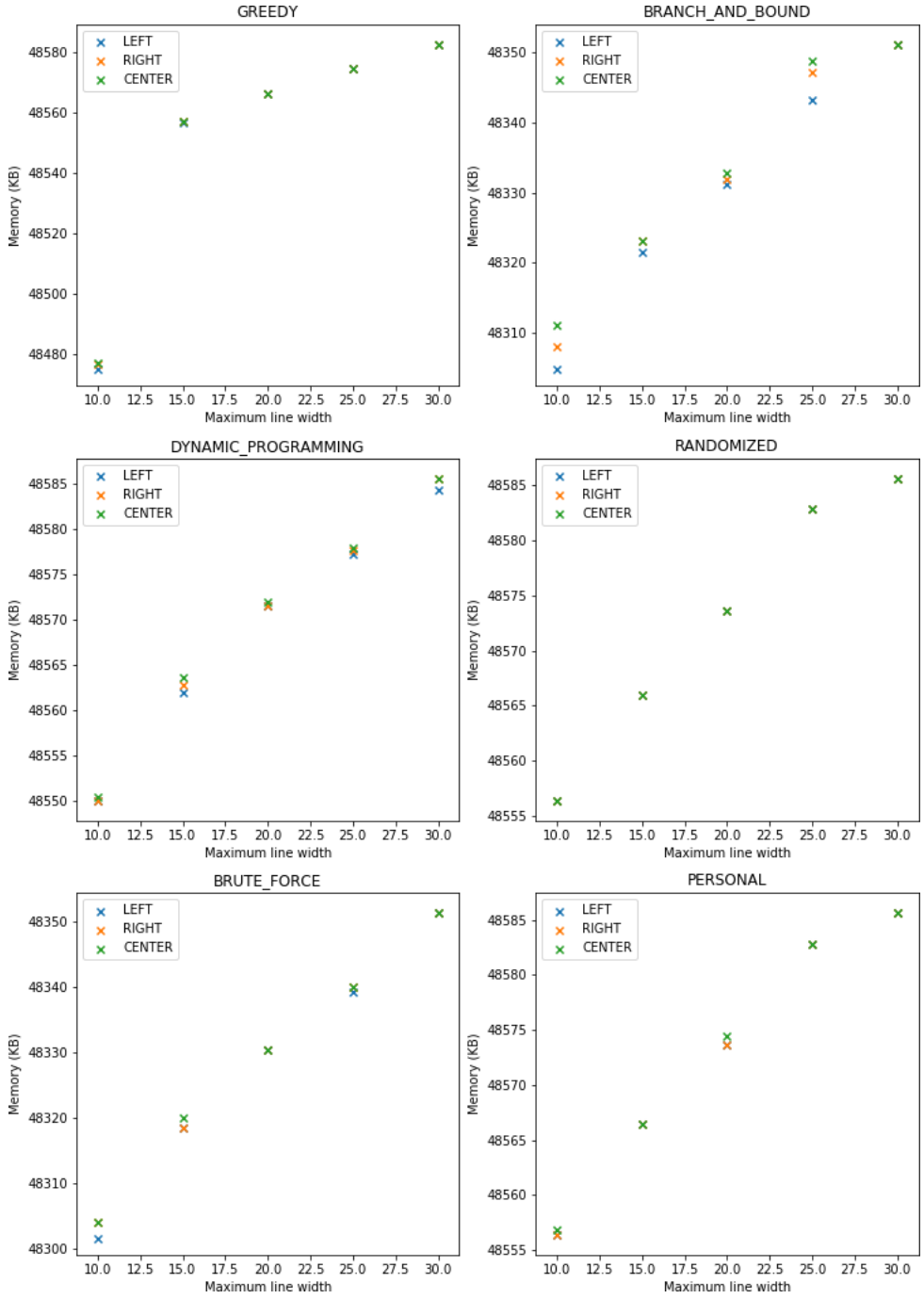

Figure B.6: Average memory usage (in KB) against maximum line width for each approach

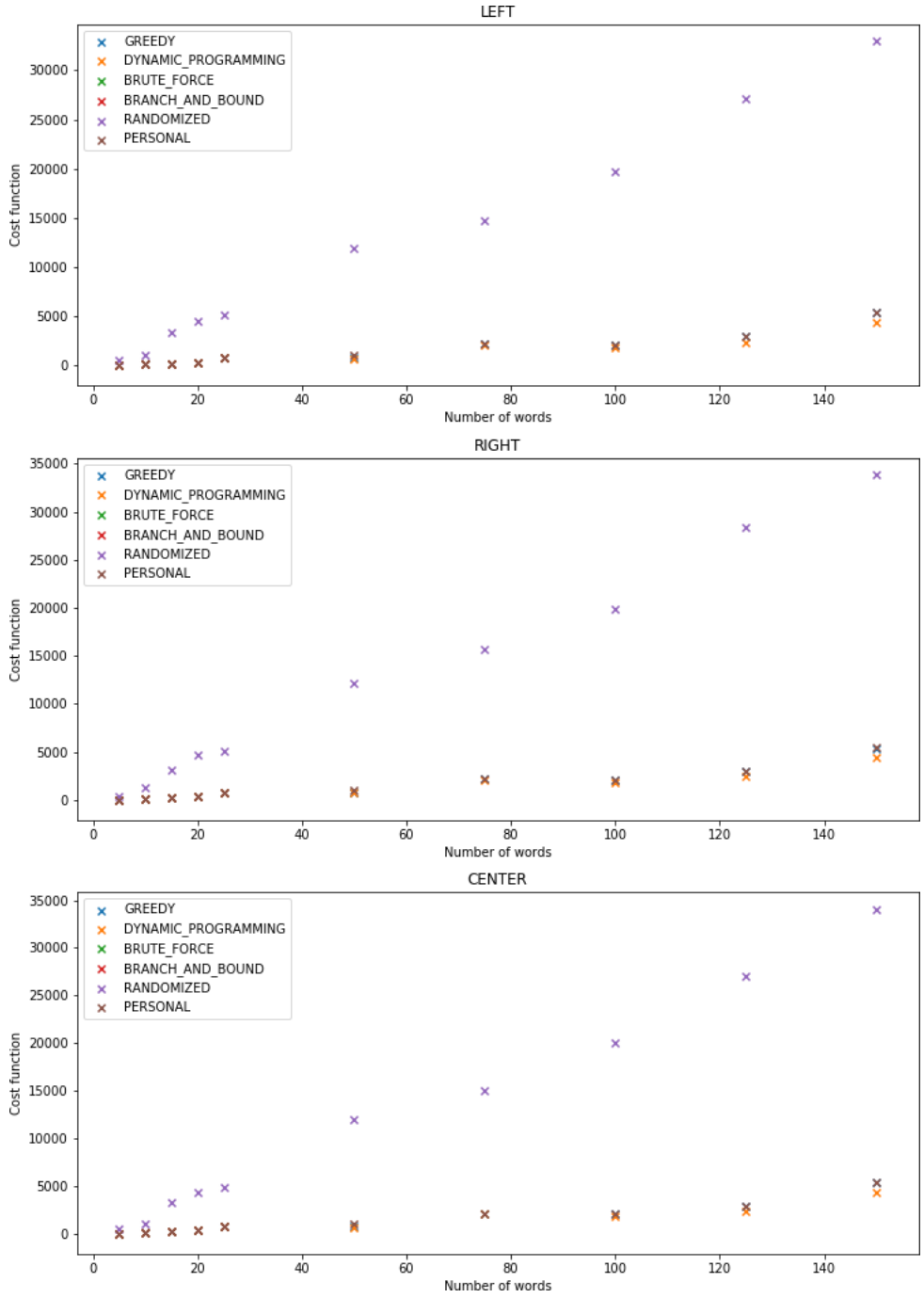

Figure B.7: Average cost function against number of words for each alignment

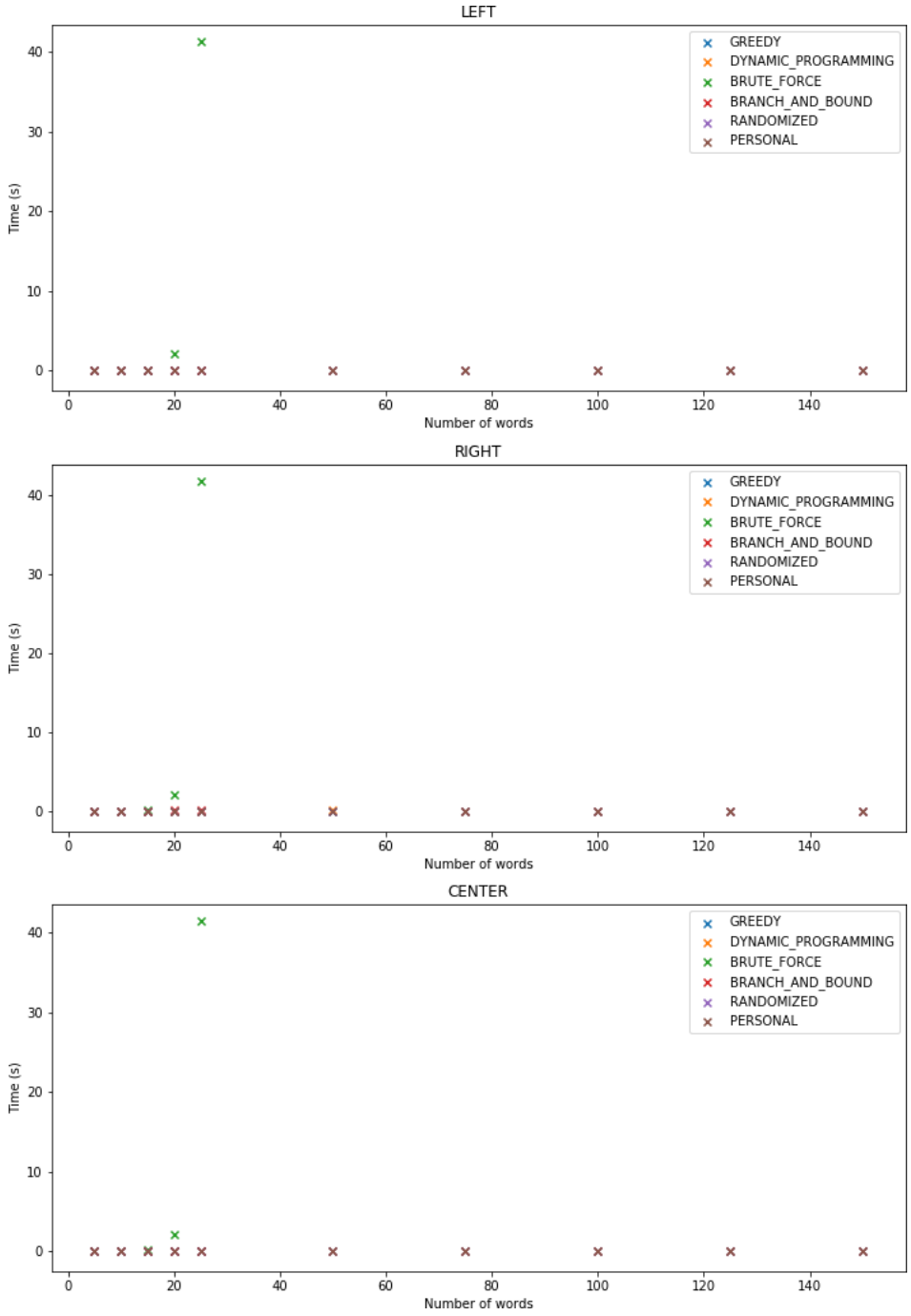

Figure B.8: Average running time (in seconds) against number of words for each alignment

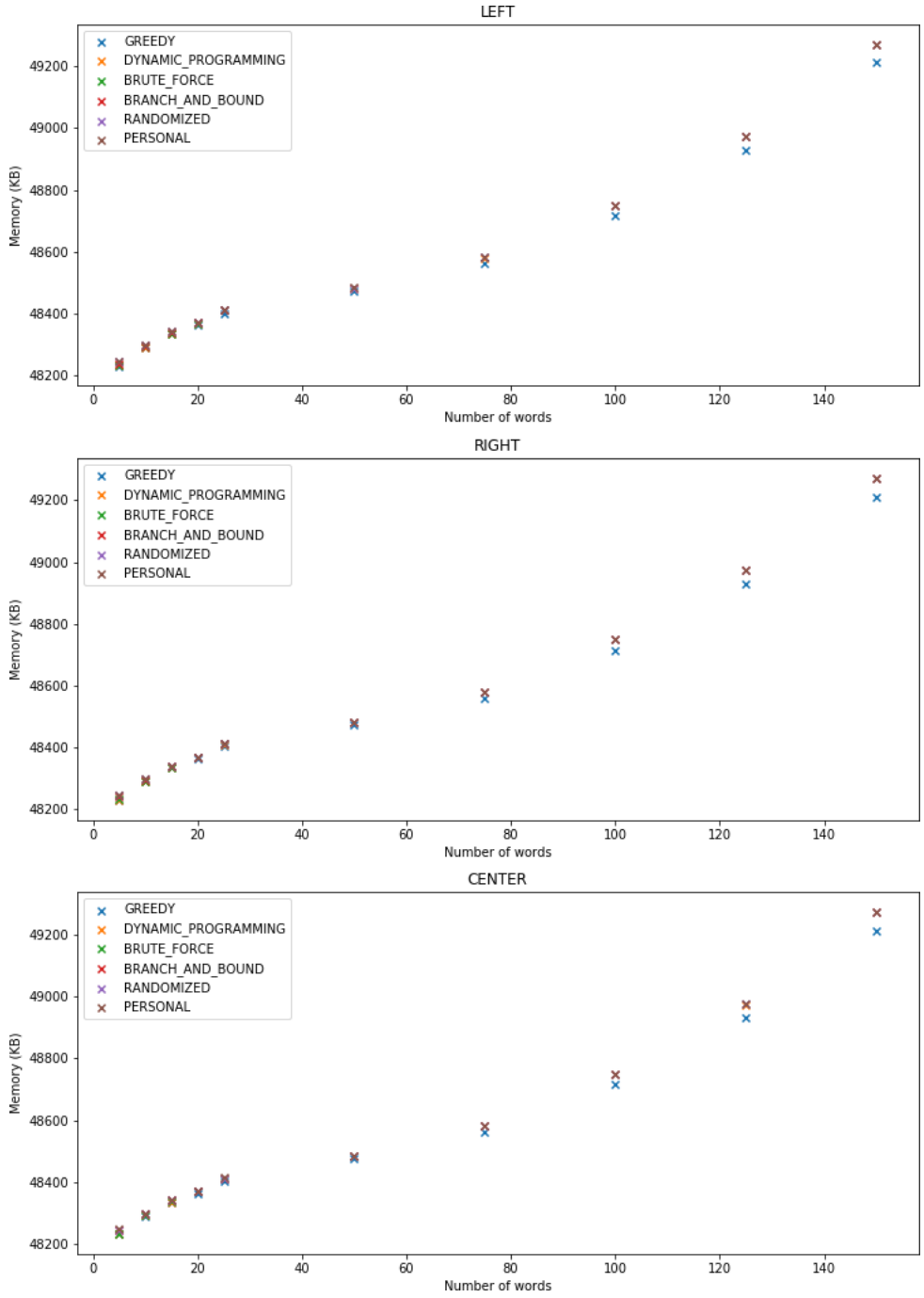

Figure B.9: Average memory usage (in KB) against number of words for each alignment

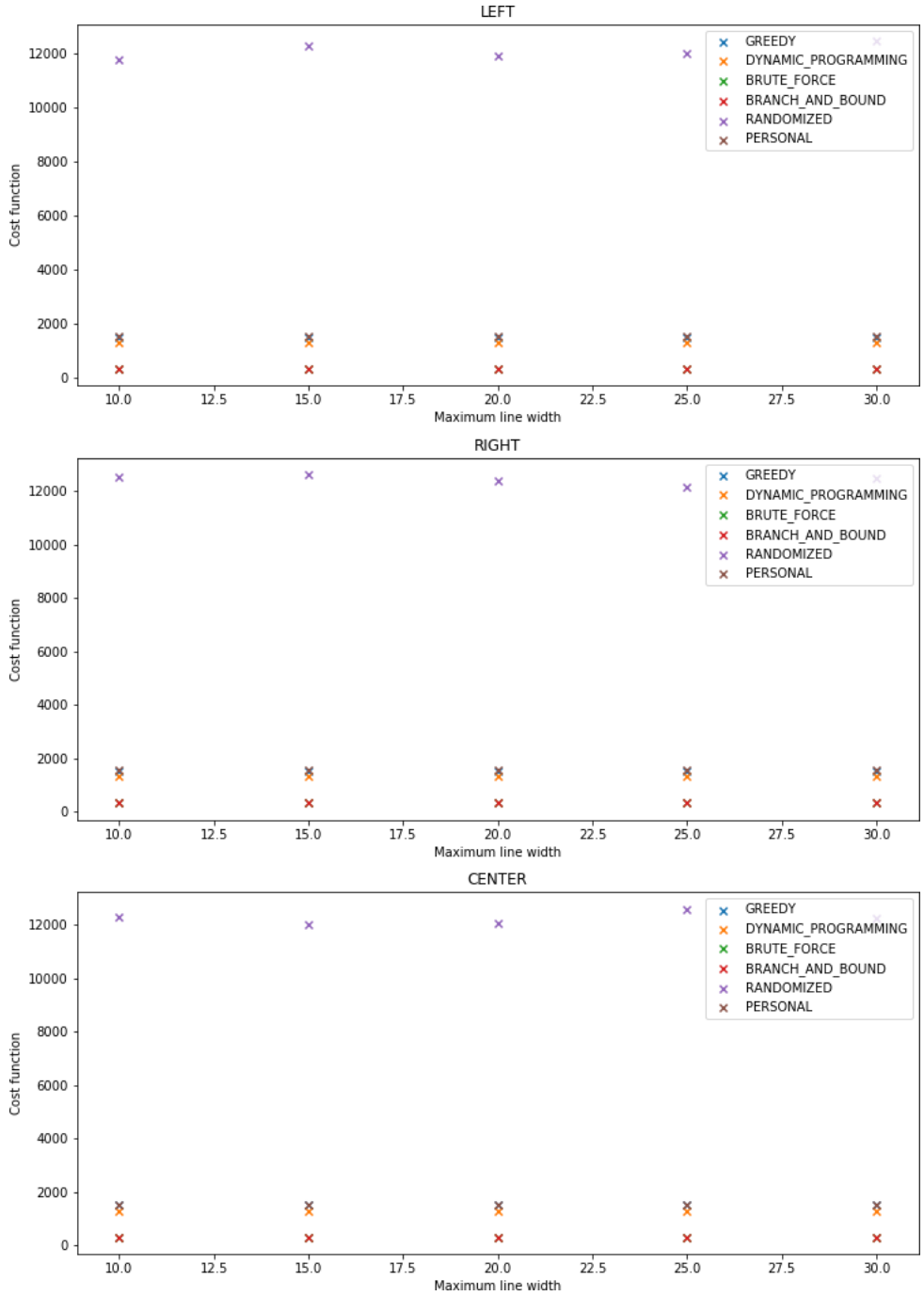

Figure B.10: Average cost function against maximum line width for each alignment

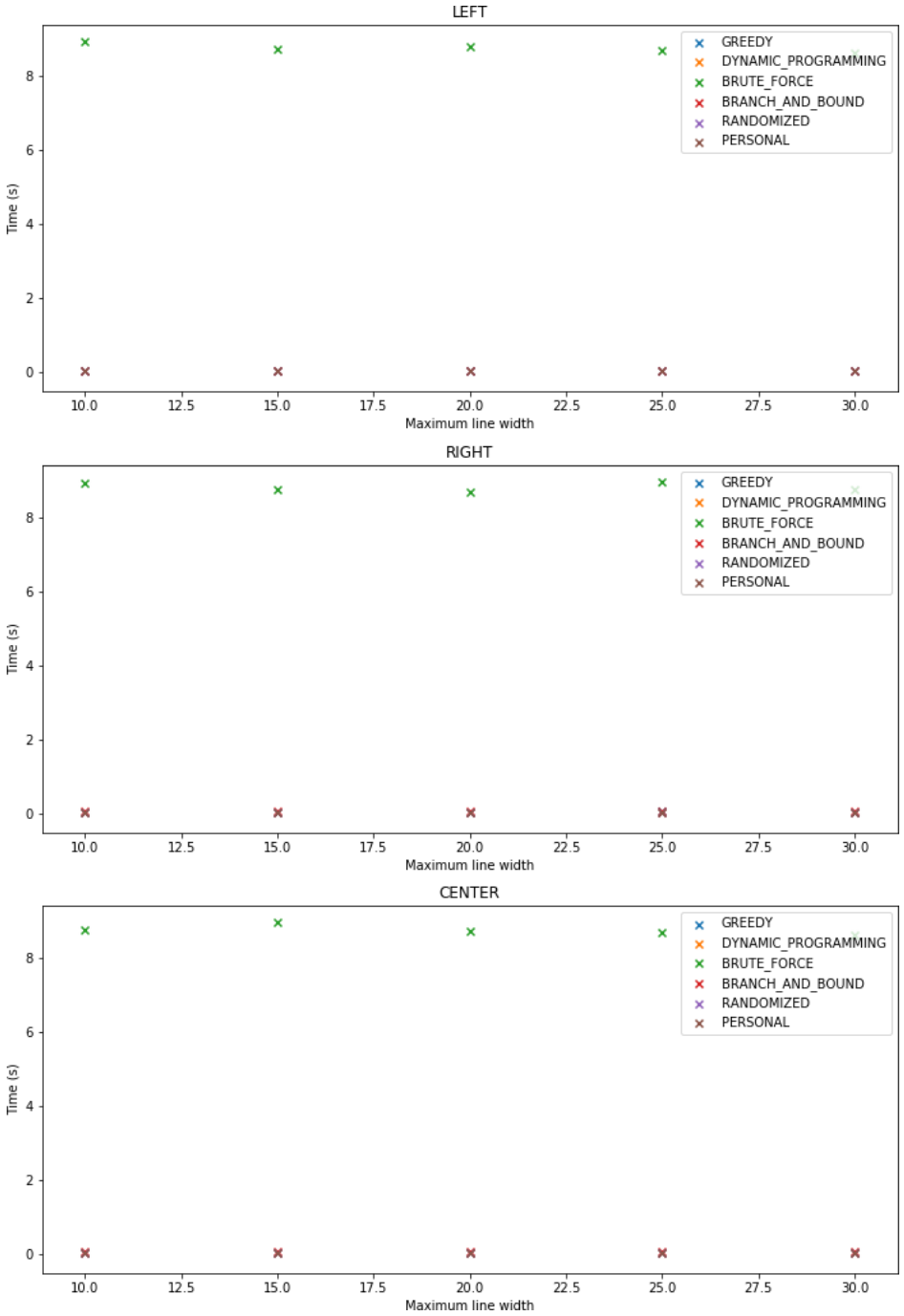

Figure B.11: Average running time (in seconds) against maximum line width for each alignment

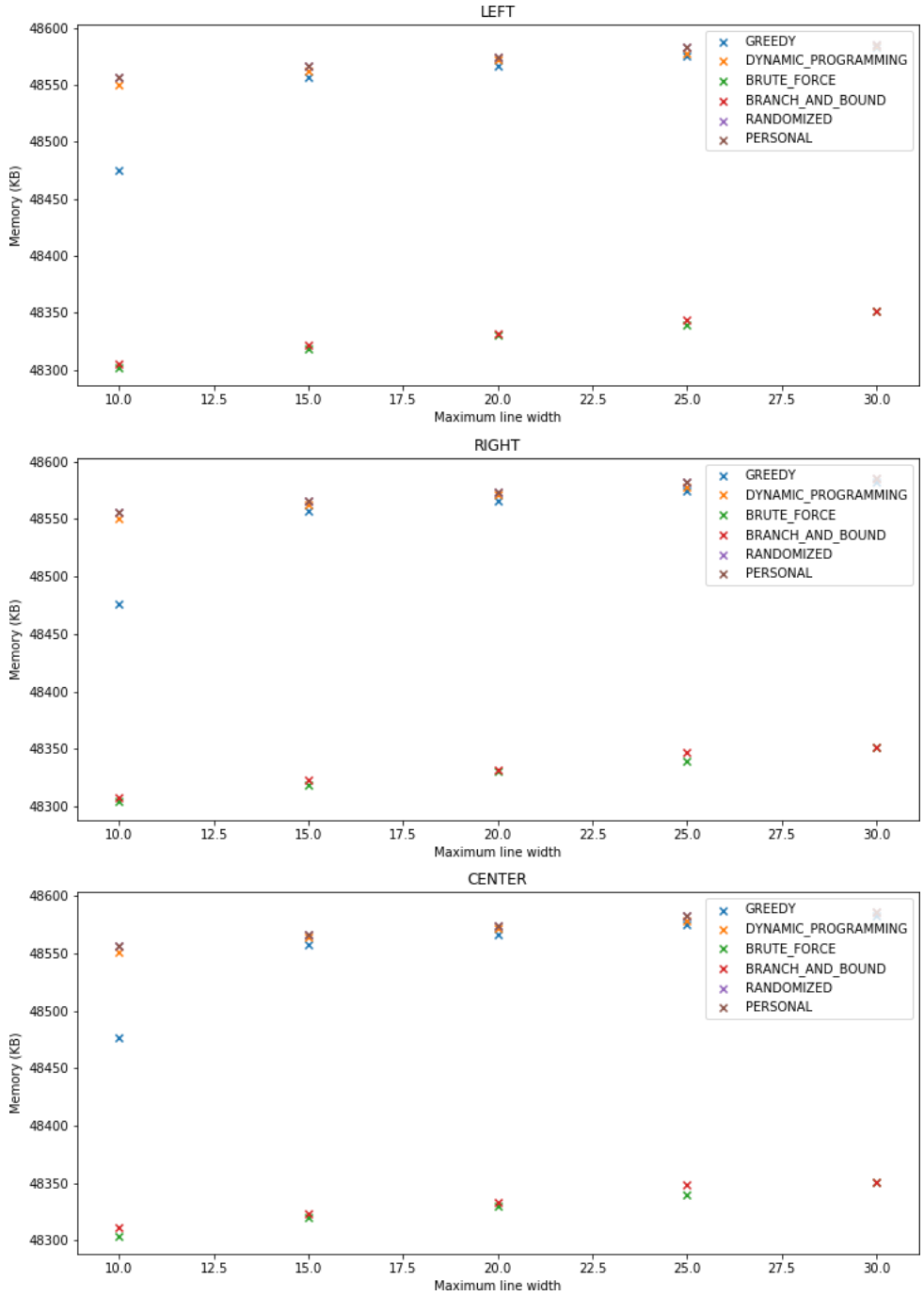

Figure B.12: Average memory usage (in KB) against maximum line width for each alignment

Appendix C

Running Instructions

All solutions were built using Python 3.7.3.

Our group solution is available in Jupyter Notebook files:

- For Algorithms and Study: *Text Formatting Problems.ipynb*
- For Results Analysis: *Results Analysis.ipynb*

We also provide a Graphic User Interface: folder GUI, files *gui.py* and *align-codes.py*. A CSV file containing the results shown in our report is also provided.

C.1 Enviroment setup

C.1.1 For Algorithms

Packages required:

- enum
- math
- random
- itertools

C.1.2 For Study

Packages required:

- All packages from "For Algorithms"
- time
- memory-profiler
- csv

C.1.3 For Results Analysis

Packages required:

- numpy
- pandas
- matplotlib
- enum

C.1.4 For Graphic User Interface

For running the GUI, please keep both *gui.py* and *align-codes.py* files in the same folder and run *gui.py*. Packages required:

- All packages from "For Algorithms"
- PyQt5

Appendix D

Task Delegability

We present in Table D.1 the project milestones and the group member responsible for each of them.

Table D.1: Project milestones

ID	Milestone	Tasks assigned to
GY-LR	Greedy approach on left/right alignments	Levi
DP-LR	Dynamic programming approach on left/right alignments	Sanjana
BF-LR	Brute force approach on left/right alignments	Geraldine
BB-LR	Branch-and-bound approach on left/right alignments	José
GY-C	Greedy approach on center alignment	Levi
DP-C	Dynamic programming approach on center alignment	Sanjana
BF-C	Brute force approach on center alignment	Geraldine
BB-C	Branch-and-bound approach on center alignment	José
RD	Randomized approach on left/right/center alignments	Levi
PV	Personal version on left/right/center alignments	Levi
GUI	Graphical User Interface	Geraldine, José
RPT	Project Report	Group