Longest common subsequence: an algorithmic component analysis

Luc Libralesso, Aurélien Secardin, Vincent Jost

Highlights

Longest common subsequence: an algorithmic component analysis

Luc Libralesso,Aurélien Secardin,Vincent Jost

- We perform an algorithmic component analysis and quantify the performance of the most studied components
- We present an iterative beam search with a geometric growth factor and show that it is competitive
- We report new best-so-far solutions on some standard Longest Common Subsequence benchmarks.

Longest common subsequence: an algorithmic component analysis

Luc Libralesso[a,b,*], Aurélien Secardin[a,b] and Vincent Jost[a]

[a] Univ. Grenoble Alpes, CNRS, Grenoble INP, G-SCOP, 38000 Grenoble, France
[b] Each author contributes equally to this work.

ARTICLE INFO

Keywords:
Anytime tree search
Longest Common Subsequence
Iterative Beam Search
dominance filtering
probabilistic guides

ABSTRACT

We study the performance of various algorithmic components for the longest common sequence problem (LCS). In all experiments, a simple and original anytime tree search algorithm, iterative beam search is used. A new dominance scheme for LCS, inspired by dynamic programming, is compared with two known dominance schemes: local and beam dominance. We show how to compute the probabilistic and expectation guides with high precision, using logarithms. We show that the contribution of the components to the algorithm substantially depends on the number of sequences and if the sequences are dependent or not. Out of this component analysis, we build a competitive tree search algorithm that finds new-best-known solutions on various instances of public datasets of LCS. We provide access to our computational code to facilitate further improvements.

1. Introduction

The longest common subsequence problem is a classic computer science problem. It has been largely studied in the last 50 years [1] and been used in a large variety of applications, for instance in file comparisons (*diff* [20] and *git* software), in a plagiarism detection tool [35], in manufacturing to better exploit positive interactions between operators [16], in optimizing multiple-queries in databases [31], disease classification [6], in bioinformatics[14, 23, 33] and pattern recognition[28]. Given an instance defined by the tuple $(\mathcal{S}, \Sigma)$ where $\mathcal{S} = \{s_1, s_2 \dots s_n\}$ a set of n sequences over the alphabet Σ, find the longest sequence t^* that is a subsequence of all sequences of $\mathcal{S}$ (*i.e.* removing symbols in sequences such that each sequence contains t^*). The longest common subsequence problem can be solved in polynomial time by dynamic programming ($O(m^n)$ where $m = \max_{s_i \in \mathcal{S}} |s_i|$ is the maximum size of the sequences of $\mathcal{S}$ and $n = |\mathcal{S}|$ the number of sequences). It is known to be $\mathcal{NP}$-Hard for an arbitrary number of sequences. Figure 1 presents an example of an LCS instance and shows one of its optimal solutions in bold.

s_1	**a**	**b**	**c**	a	d	c	c
s_2	d	**a**	a	d	**b**	**c**	d
s_3	d	c	**a**	**b**	**c**	a	

Figure 1: Example of a LCS instance with $n = 3$ sequences s_1, s_2, s_3 and an alphabet of $\Sigma = \{a, b, c, d\}$. The optimal solution (sequence abc) is shown in bold.

Over the last 50 years, many algorithms were presented to solve the longest common subsequence problem, all presenting novel and interesting ideas and algorithmic components. However, an in-depth analysis of the component contribution is rarely found in the literature. Moreover, many people tune their algorithm parameters so they perform relatively well on all instances. But the performance of some components is largely influenced by the instance properties (for instance the number of sequences or size of the alphabet). It also appears that some algorithmic components are rarely questioned, because being part of efficient algorithms, or, in the opposite case, rarely used, thus rarely questioned. In both cases, their contribution has never been evaluated. This article aims to fill this gap. We implement several algorithmic components from the literature and quantify their performance on various instances. We measure the impact of using it alongside other components. We show that some components are rarely considered while being at the same time simple and efficient. Some other components are implemented in many works

*Corresponding author
luc.libralesso@grenoble-inp.fr (L. Libralesso); aurelien.secardin@icloud.com (A. Secardin); vincent.jost@grenoble-inp.fr (V. Jost)
ORCID(s):

while their contribution is limited (in some cases, they can even harm the algorithm global performance). Out of our computational experiments, we build some simple tree search algorithms and show that they are competitive with the state-of-the-art and even sometimes return new best-so-far solutions. The full source code is available online (`https://gitlab.com/secardia/cats-ts-lcs`).

This article is structured as follows: Section 2 presents some algorithms present in the literature and the classical search tree structure used to solve the longest common subsequence problem. Section 3 presents the 3 most studied guides in the literature (namely the bound guide, the probabilistic guide, and the expectation guide). Section 4 presents some search-space reductions based on dominance or dynamic programming. Section 5 introduces the iterative beam search strategy and Section 6 presents the numerical results we obtained and a comparison with the current state-of-the-art algorithms.

2. Tree search algorithms for the longest common subsequence problem

Many algorithms were proposed to solve the longest common subsequence problem. Most of them being tree search algorithms or dynamic programming inspired algorithms as they seem to be the most intuitive and efficient for this problem (we discuss these techniques more in-depth in the next sections). We still can notice that other methods have been tried as well as a large neighborhood search [11], multiple evolutionary algorithms [21, 17, 25, 5], and a simulated annealing [36].

forward search: As discussed previously, many tree search or dynamic programming algorithms were proposed. They almost all rely on a simple forward search where the branching is done on the next symbol of the alphabet belonging to the longest common subsequence candidate. Such methods maintain a vector P of n pointers to the next position in each sequence. They start the search (root node) by an empty subsequence. At a given node, the algorithm decides on the next symbol of the candidate longest common subsequence (say a), moving pointers such that P_i points to the next symbol a in s_i for all sequences $s_i \in S$. When a pointer reaches the end of a sequence, the candidate subsequence cannot be further extended and the candidate subsequence is compared to the best found during the search. Figure 2 presents the forward search tree of the example instance presented before.

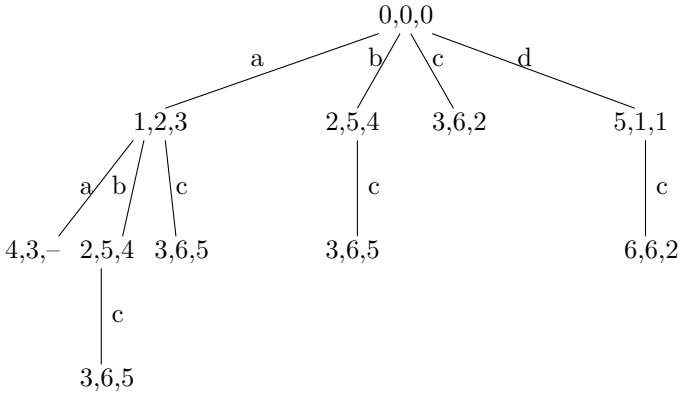


Figure 2: Forward search tree example. At the root node, no symbol is consumed and all pointers $p_i \in P$ point to the first symbol of each sequence. At each decision, a symbol of the alphabet Σ is consumed and the pointers point to the next symbol of the sequence that is not consumed. Possibly, a symbol cannot be added to the candidate solution if it is not present in the remaining symbols of a sequence. We may reach the optimal solution for the instance by following the longest path in the tree (in this case abc). In this figure, we note by "–" a pointer value outside the sequence. For instance on the node aa, there is no symbol left in s_3, thus denoting the pointer p_3 by "–".

Dynamic programming: Many dynamic programming algorithms have been proposed to solve the longest common subsequence problem. The first one generalizing the 2-sequence case that enumerates all possible pointer positions [15], with a time and space complexity of $O(m^n)$. And a second that enumerates all reachable pointer positions with a time and space complexity of $O(n|\Sigma|^{|t^*|})$ that was later integrated within branch-and-bound techniques [18, 10] and

performs better if the instance has a large number of sequences. We present this integration within a tree search framework in Section 4.

Heuristic tree search: To the best of our knowledge, the best heuristic algorithms for the longest common subsequence were published in 1994 [7]. They perform greedy algorithms (sometimes with a lookahead strategy) that are guided by the pointer progression or the symbol count in the remaining subsequence. The long-run algorithm [22] was published in 1995 and consists in a greedy algorithm that adds to the candidate solution the symbol that maximizes its minimum number in sequences (*i.e.* $\max_{a\in\Sigma} \min_{s_i\in S} f(s_i, a)$ where f counts the number of symbols a in sequence s_i). Many other algorithms as the expansion algorithm [4], the best-next algorithm [24] (that selects the symbol that maximizes the minimum remaining subsequence) and variants [19]. We may note that the long run and expansion algorithms were proven to be $|\Sigma|$-approximations. The best-next algorithm is not but provides excellent results in practice [19]. These algorithms obtained good results, however, because of their greedy nature, they stop when they find a solution and are not able to improve the solution further with some additional time.

A first breakthrough happened in 2009 by the publication of the first beam search algorithm for the longest common subsequence [3]. The beam search algorithm can be seen as a generalization of a greedy algorithm that allows, depending on some parameters, to find better solutions by increasing the run time (we discuss beam search more in detail in Section 5). This algorithm uses bounds inspired by the best-next algorithm in order to reduce the search-space (we discuss this bound in Section 3) and reduces even further the search space by removing symbols that are dominated (we call it local dominance and discuss it in Section 4) and remove all candidates that are dominated by another within the current iteration. This strategy largely outperformed existing methods. We may note that some other algorithms were proposed to extend greedy algorithms such that they can better use the available time to find better solutions or built on the beam-search algorithm. For instance, an ant colony optimization algorithm [32], a beam-ACO algorithm [2] that replaces the ant colony greedy expansion by a beam search, a hyper-heuristic based on beam-search that dynamically chooses between two guide variations [34], a new anytime tree search algorithm [37]

A second breakthrough happened in 2012 by the publication of a beam-search that replaces the best-next guidance by a probabilistic one [29]. This probabilistic guide is computed by a dynamic programming algorithm during the preprocessing and has a time and space complexity of $O(m^2)$ where m is the size of the largest sequence. It assumes that sequences are independent and computes the probability that there exists a common subsequence of size k in the remaining symbols. At the same time, the authors introduced a filtering step within the beam search to remove dominated nodes by an elite set of nodes present in the current iteration. This beam-search with the probabilistic guide obtained excellent results on the public datasets (we discuss this probabilistic guide in Section 3 and discuss some numerical errors when probabilities are too small). Recently, this probabilistic guide was improved into an expectation guide [9, 8] (we discuss it in Section 3). The tree search algorithms they proposed, to the best of our knowledge, is the state-of-the-art.

As we may notice, many components were introduced, the best-next, probabilistic and expectation guides, multiple search strategies (greedy, beam-search, anytime column search, anytime pack search *etc.*) and multiple dominance strategies (dynamic programming integrated within a tree search algorithm, dominated symbols and the beam filtering). One question that naturally comes in mind is: What is the contribution of each component. As the beam search and the local dominance were introduced together, how much each of these components contributes to the algorithm performance? Is one useless? The same goes for the probabilistic or expectation guides and the beam-filtering procedure. Moreover, it has been shown that some dynamic formulations were more efficient depending on the alphabet size and the optimal longest common subsequence [19]. Such remarks may also apply to other tree search components. In the next sections, we describe further some well-studied components on the longest common subsequence and some components that proved to be effective on similar problems. We measure the contribution of each of them on different instance classes and show that the local dominance does not contribute to the algorithm performance despite being used by many state-of-the-art since 2009. We also show that state-of-the-art results can be obtained with a simple iterative beam search with geometric growth (thus not using advanced tree search methods). We also show that the probabilistic and expectation guides, while being efficient for most instances of the literature are inefficient for instances where sequences are dependant.

3. Guides

Anytime tree search algorithms are tree search methods that aim to find solutions fast while purposely not considering some parts that appear less interesting. To this extent, one has to design a function (usually called $f(n)$ or $f'(n)$) that returns the apriori quality of the node. This function may be a bound (major component of branch-and-bound algorithms) that also allows pruning nodes that are worse than the best-known solution or something else that may better identify promising nodes. In this section, we present 3 of the most studied guides in the longest common subsequence problem literature (one bound and two "heuristic" guides).

3.1. Bound guide

We start our guide study by the guidance function used in the best-next heuristic and was later by many tree search algorithms. This guide happens to also be a bound and corresponds to an optimistic estimate of the remaining longest common sequence by assuming that each symbol of the smallest remaining sequence can be added. This bound guide can be computed in $O(n)$ time.

$$f_b = \min_{s_i \in S} |s_i| - p_i + 1$$

It was massively used before being replaced by the probabilistic and expectation guides. A major drawback of this guide is that it does not distinguish between some cases. Indeed, consider an instance where $|s_1| = |s_2| = |s_3| = 10$ and two nodes with respective pointer positions $P_1 = (5, 1, 1)$ and $P_2 = (5, 5, 5)$. Intuitively, the node with pointers P_1 is more attractive as it has less expanded s_2 and s_3. However, the bound guide does not distinguish between two nodes. The probabilistic and expectation guides (in the next subsections) allow us to better quantify the quality difference of these nodes. It may a priori seem that the bound guide is dominated by the probabilistic and expectation guides (publications since 2012 seem to believe that), however, we show in Section 6 that the bound guide is surprisingly efficient on some instances.

3.2. Probabilistic guide

The probabilistic guide [29] requires some function $Pr(k, q)$ that returns the probability that there exists a uniform random sequence of length k in a uniform random sequence of size q. This probability can be computed during the pre-processing in a space and time complexity of $O(m^2)$ using the following recurrence equation (m being the size of the longest sequence among the n of the instances).

$$Pr(k, q) = \begin{cases} 1 & \text{if } k = 0 \\ 0 & \text{if } k > q \\ \frac{1}{|\Sigma|} \times Pr(k-1, q-1) + \frac{|\Sigma|-1}{|\Sigma|} \times Pr(k, q-1) & \text{otherwise} \end{cases}$$

Using this probability, we define the probabilistic guide as follows: Given a node with pointers $P = (p_1, p_2 \dots p_n)$, it computes the probability, assuming that sequences are independent and uniform random sequences) that there exists a uniform random sequence of size h (more on it later).

$$f_p(P, h) = \prod_{p_i \in P} Pr(h, |s_i| - |p_i| + 1)$$

In order to provide a guide, the value h has to be fixed. If fixed too low or too high, the guide would not be able to distinguish between two nodes. The better the parameter h, the better the guide quality. In the original work presenting this guide, it was fixed to:

$$h = \frac{\min_{i \in \{1 \dots n\}, c \in \text{Candidates}} |s_i| - p_i + 1}{|\Sigma|}$$

This formula provides a relatively good h value. However, as the original authors state: "there is still room for further improvement". We believe that, using an iterative beam search strategy, it is possible to find another, and more natural way to determine a good h value. Indeed, one can fix as a h value the value returned by the previous beam iteration. This way, the h is set to a (relatively) good estimate of the longest common subsequence.

Numerical precision issues: When implementing the probabilistic guide, one has to be careful that, on some instances, probabilities can be so small that the numerical precision on classical variables would not be able to handle them. A C++ double variable is encoded using 1 bit for the sign, 52 bits for the fraction and 11 bits for the exponent. The smallest positive number that can be expressed by a double is therefore $\approx 10^{-307}$. Below this value, the guide would not be able to distinguish between two nodes and the algorithm would have a random behavior, harming its performance considerably. Regarding the literature datasets, the $\approx 10^{-307}$ precision limit of the guide on the nodes called by the beam search is reached on most instances of the BB dataset. More generally, this could become a serious bug for large instances with LCS value much bigger than what we typically find in random instances. For this reason, we propose a way to fix this by representing intermediate computation results using logarithms. We discuss in Section 6 the performance improvement of these computations.

One simple idea to handle very small probabilities is to encode their logarithm (also in double precision). The major problem is that, when we compute the table of probabilities by Dynamic Programming, we have to sum up two probabilities. If we have stored $l_1 = \ln(p_1)$ and $l_2 = \ln(p_2)$ and want to evaluate $(p_1 + p_2)$ we would lose all the benefit of the storage in logarithm while writing $\exp(l_1) + \exp(l_2)$.

To overcome this bottleneck, we define a new recursive function:

$$\ln(Pr(k,q)) = \begin{cases} \ln(1) & \text{if } k = 0 \\ \ln(0) & \text{if } k > q \\ \text{SUMLOG}(-\ln(|\Sigma|) + \ln(Pr(k-1,q-1)), \ln(|\Sigma|-1) - \ln(|\Sigma|) + \ln(Pr(k,q-1))) & \text{otherwise} \end{cases}$$

where the SUMLOG function returns the logarithmic sum of two values
(*i.e.* $\text{SUMLOG}(\ln(a), \ln(b)) = \ln(a+b)$) and can be (mathematically) defined as follows:

$$\text{SUMLOG}(x, y) := \ln(\exp(x) + \exp(y))$$

A problem with this definition: If x and y are smaller than $\ln(10^{-307}) \approx -710$, $\text{SUMLOG}(x, y)$ would be set to 0 because of the double precision limit. A much more robust way to add probabilities stored with their logarithms is to re-scale them using their maximum. The previous equation is indeed equivalent to:

$$\begin{aligned} \text{SUMLOG}(x, y) &= \ln(\exp(\max(x, y)) + \exp(\min(x, y))) \\ &= \ln(\exp(\max(x, y)).(1 + \exp(\min(x, y))/\exp(\max(x, y))) \end{aligned}$$

Which allows to rewrite it in a mathematically equivalent, but more robust form, when using double precision:

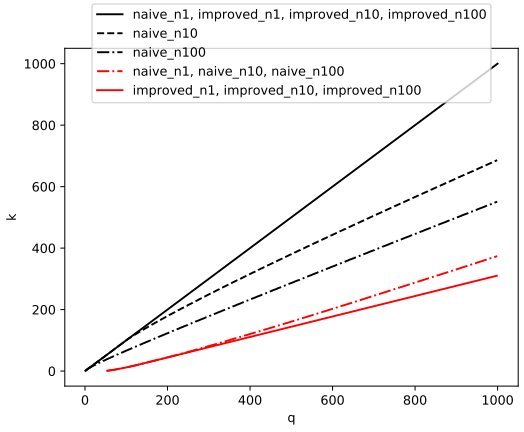
$$\text{SUMLOG}(x, y) = \max(x, y) + \ln(1 + \exp(\min(x, y) - \max(x, y)))$$

We show that this new computation of probabilities using logarithms provides a significant improvement in terms of the guide performance in Section 6. This implementation is less prone to numerical errors than the one from the literature.

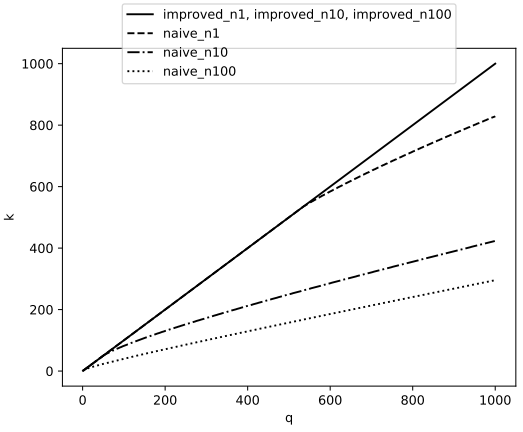
In figure 3, we pre-compute, for three alphabet sizes $|\Sigma| = \{2, 4, 24, 100\}$, the two probability estimates $(\ln(Pr(k, q)))$ of having a random sequence of size k being a subsequence of a sequence of size q for each pair $(k, q) \in \{0, 1 \dots 1000\}$. One with the "naive" version, and one with the "improved" version. We observe the difference between the two versions. We notice that the improved version consistently fits the diagonal and is closer to the x-axis, thus differentiate most scenarios. However, the naive version rapidly does not handle cases with large alphabet sizes or sequences sizes. We quantify this computation improvement on a standard dataset in Chapter 6.

3.3. Expectation guide

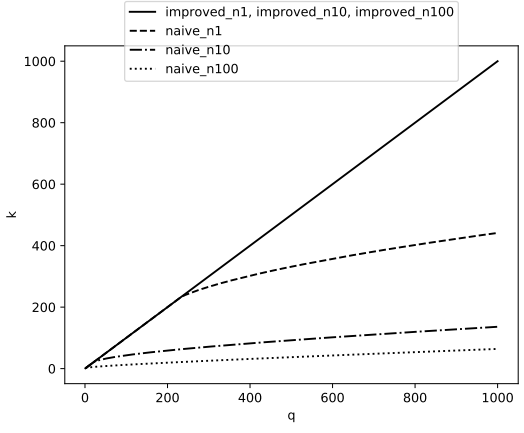
Instead of computing the probability that a random uniform sequence is a common subsequence, one may compute the expected length of the longest common sequence. Such expected length can be defined as follows, as described in the literature [8]:



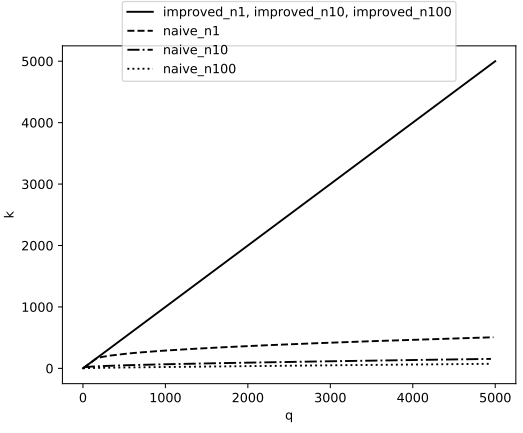
(a) Probability precision limit with $|\Sigma| = 2$



(b) Probability precision limit with $|\Sigma| = 4$



(c) Probability precision limit with $|\Sigma| = 24$



(d) Probability precision limit with $|\Sigma| = 100$

Figure 3: Numerical precision thresholds (upper in black and lower in red) of the probabilistic guides. We assume here that all the remaining subsequences have the same size q. Better guides are able to compare more different scenarios. The more the black curves fit to the plot diagonal, the better they behave (above the black curve all probabilities are equal to 0, the guide is not able to differentiate different scenarios, thus behaves randomly). Similarly, the better the red curves are close to the x-axis, the better the guide (below the red curve all probabilities are equal to 1, the guide is not able to differentiate different scenarios, thus behaves randomly). We use the following notation: "guide_nX", where "guide" can be the "naive" or "improved" versions and "X" is the number of sequences.

$$f_e = h_{\max} - \sum_{h=1}^{h_{\max}} \left(1 - \prod_{i=1}^{n} Pr(h, |s_i| - p_i + 1) \right)^{|\Sigma|^h}$$

Or, for more numerical robustness, we can restate using the table of logarithms of probabilities, as described in previous section :

$$f_e = h_{\max} - \sum_{h=1}^{h_{\max}} \left(1 - \exp(\sum_{i=1}^{n} \ln(Pr(h, |s_i| - p_i + 1))) \right)^{|\Sigma|^h}$$

Numerical precision issues: Regarding the implementation of the expectation guide, one may notice that computing the power $Q^{|\Sigma|^h}$ is too large to fit in any standard variable. The way used in the literature is to approximate this value using Taylor expansions [8].

We used a different approach, using the function $-\ln(-\ln(p))$ to store probabilities. This is the reciprocal of the (logistic) Gompertz function $\exp(-\exp(-x))$. Like the logarithm or the logit, this function allows to better store very small probabilities using double precision. Moreover, the double iteration of the logarithm allows to deal easily with the double exponentiation, $Q^{|\Sigma|^h}$ (we use the symbol Q to denote the expression $Q(n,h,s,p) = 1-\exp(\sum_{i=1}^{n} \ln(Pr(h,|s_i|-p_i+1)))$):

$$\begin{aligned} -\ln(-\ln(Q^{|\Sigma|^h})) &= -\ln(-|\Sigma|^h.\ln(Q)) \\ &= -\ln(|\Sigma|^h) - \ln(-\ln(Q)) \\ &= -h.\ln(|\Sigma|) - \ln(-\ln(Q)) \end{aligned}$$

For probabilities close to 1, the minimum difference between two values that can be handled by C++ doubles is $\approx 10^{-17}$. This limit is easily reached when $\exp(\sum_{i=1}^{n} \ln(Pr(h,|s_i|-p_i+1)))$ is small.

But as we can see in Figure 4, for $x \leq -4$, $\ln(-\ln(x)) \approx x$, therefore we can assign an approximate value to Q using the following algorithm:

If $\sum_{i=1}^{n} \ln(Pr(h,|s_i|-p_i+1)) \geq -37$ ($\exp(-37) \approx 10^{-17}$):

$$Q = 1 - \exp(\sum_{i=1}^{n} \ln(Pr(h,|s_i|-p_i+1)))$$

Else:

$$Q = \sum_{i=1}^{n} \ln(Pr(h,|s_i|-p_i+1))$$

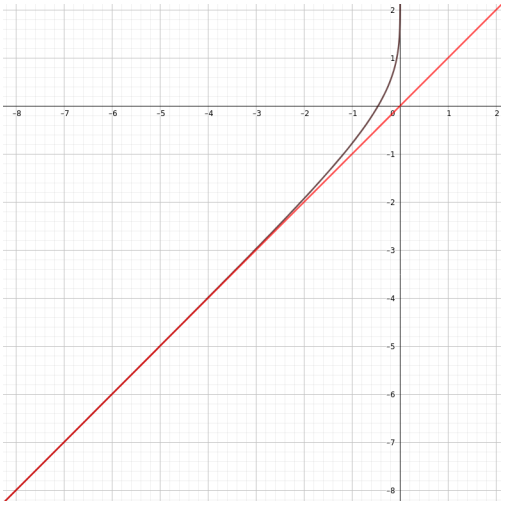


Figure 4: In black : $f(x) = \ln(-\ln(1-\exp(x)))$, in red : $g(x) = x$

4. Dominance strategies

In the previous section, we discussed guide functions to guide the search towards promising regions. We presented three classical guide functions: the bound, probabilistic, and expectation guides. Another way to improve the search strategy is to perform a search-space reduction. The fewer nodes to visit, the more likely the algorithm yields competitive results (sometimes even depleting the search tree, thus proving the optimality of the best-known solution). In this section, we present two classical search-space reductions, found in the literature, based on dominance between nodes and a new dominance strategy based on dynamic programming (we call it *Global dominance*).

4.1. Local dominance

When generating children of a given node (each child correspond to a symbol appended to the candidate longest common sequence), sometimes, a symbol dominates another. For instance (see the instance presented in the introduction) the symbol a generates the pointers $P_a = (1, 2, 3)$ and the symbol b generates the pointers $P_b = (2, 5, 4)$. As we may notice, each pointer of P_a is strictly less than its respective pointer from P_b. It means that one can add a and b while obtaining exactly the same pointer set as P_b. Thus, choosing b is dominated by choosing a and b can be safely ignored.

4.2. Beam dominance

The beam dominance considers an elite set of size k (possibly, $k = \infty$ to check full dominances [3]). While adding a node to the pool of the candidate set, a dominance check is performed with every node in the elite set. A node n_1 is dominated by a node n_2 if its set of pointers P_1 are all superior or equal to their respective pointer in P_2. In the literature, we usually find $k = \infty$ [3, 8], $k = 7$ [29] or $k = 1$ [8].

4.3. Global dominance

We propose another new dominance algorithmic component inspired by dynamic programming. For each node explored, the global dominance strategy stores the pointer set into a database with the size of the current candidate's longest common sequence. If an entry exists, the candidate size is compared to the one existing in the database. The entry is possibly replaced if the current candidate is larger than the one in the database.

A few examples: Consider an empty database.

- A candidate common sequence with length 2 and pointers $(3, 3, 4)$ is opened. There exist no $(3, 3, 4)$ entry in the database. Thus it is added to it and associate length 2.
- A candidate common sequence with length 2 and pointers $(3, 3, 4)$ is opened. There exist a $(3, 3, 4)$ entry with a larger or equal length, thus the current node is pruned as it is dominated by the database entry.
- A candidate common sequence with length 3 and pointers $(3, 3, 4)$ is opened. There exist a $(3, 3, 4)$ entry with a strictly smaller length, thus the current node dominates the database entry and updates it. The database now contains $(3, 3, 4)$ associated with length 3.

Implementation details: In Section 5, we present an iterative beam search. This iterative beam search performs restart and the following scenario may happen: During the first iteration, we open a node n with pointers $(3, 3, 4)$ and length 3. Because of its heuristic/incomplete nature, the beam search was not able to deplete the search tree, thus possibly missing an optimal value. During the next iteration, the node n is opened again. However, it is pruned because of the global dominance mechanism and can miss an optimal solution, which is a problem. One way to fix this issue is to consider in the database an iteration number that makes pruning an equivalent node only if its iteration number is equal to the one in the database.

5. Iterative beam search

Beam Search is a tree search algorithm that uses a parameter called the beam size (D). Beam Search behaves like a truncated *Breadth First Search (BrFS)*. It only considers the best D nodes on a given level. The other nodes are discarded. Usually, we use the bound of a node to choose the most promising nodes. It generalizes both a greedy algorithm (if $D = 1$) and a BrFS (if $D = \infty$).

Figure 5 presents an example of beam search execution with a beam width $D = 3$.

Beam Search was originally proposed in [30] and used in speech recognition. It is an incomplete (*i.e.* performing a partial tree exploration and can miss optimal solutions) tree search parametrized by the beam width D. Thus, it is not an anytime algorithm. The parameter D allows controlling the quality of the solutions and the execution time. The larger D is, the longer it will take to reach feasible solutions, and the better these solutions will be.

Beam Search was later improved to become *Complete Anytime Beam Search* to make it anytime. The idea is to perform a series of beam searches with a heuristic pruning rule that weakens as the iterations go [38]. They prune a node n' if its bound exceeds by some constant the bound of its parent n (*i.e.* n' is pruned if $f(n') > f(n) + c$ with c

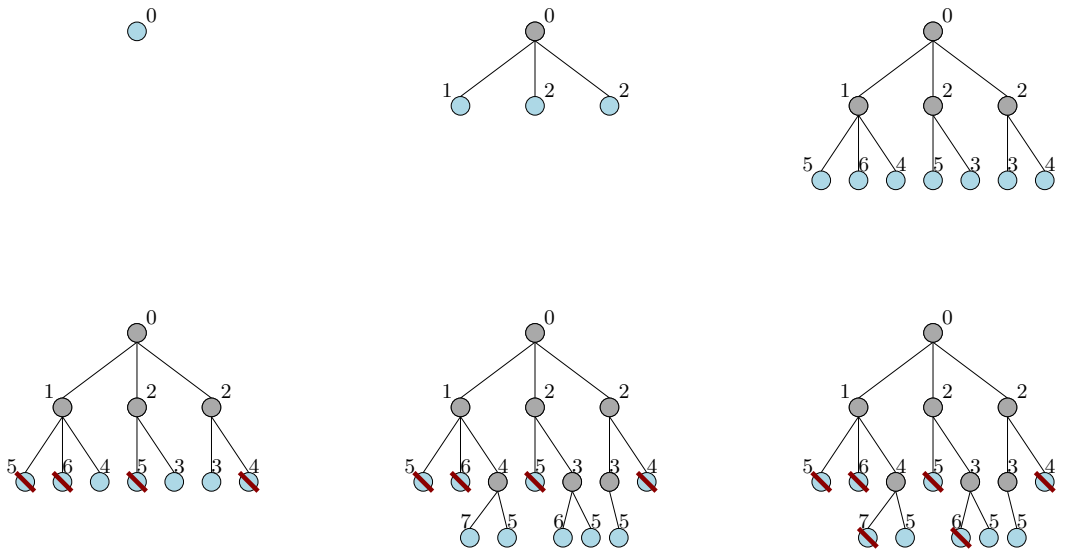


Figure 5: Beam Search Iterations with a beam width $D = 3$

increasing as iterations go). In this variant the beam is not limited to a beam width D, thus, tuning the parameter c is crucial. This variant is called *Iterative-weakening Beam Search*. Since many algorithms were later designed to make the beam search complete[1] (for instance *Anytime Column Search* or *Beam Stack Search*), thus also being complete anytime beam searches, we refer this algorithm as an iterative beam search to avoid potential confusions.

Another iterative beam-search variant increases the beam width at each restart. It consists of performing a series of beam searches with increasing D. To the best of our knowledge, such approaches have not been much studied in the literature. We may cite its use on the car sequencing problem [13] that consists of beam search runs of sizes $\{5, 10, 25, 50, 100, 500, 1000, 1500, \infty\}$. As the last iteration is not limited in width, this iterative version is complete.

It is worth noticing that Iterative Beam Search may reopen many nodes. Another possible beam increasing scheme could be to start by a beam of $D = 1$ (greedy algorithm). Then when the current search finishes, multiply D by some constant (for instance, a geometric growth of 2), thus running a beam of size 2, then 4, and so on. Such a scheme appears to be efficient in practice (and guarantees that not too many nodes are re-opened). To the best of our knowledge (and our surprise), such a variant has been used only in a very limited number of works [12, 26, 27]. In this thesis, we will use the terminology "iterative beam search" this geometric-growth variant.

Algorithm 5.1 shows the pseudo-code of an iterative beam search. The algorithm runs multiple beam searches starting with $D = 1$ (line 1) and increases the beam size (line 8) geometrically. Each run explores the tree with the given parameter D. At the end of the time limit, we report the best solution found so far (line 10). In the pseudo-code, we increase geometrically the beam size by 2. This parameter can be tuned, however, we did not notice a significant variation in the performance while adjusting this parameter. This parameter (that can be a real number) should be strictly larger than 1 (for the beam to expand) and should not be too large, say less than 3 or 5 (otherwise, the beam grows too fast and when the time limit is reached, most of the computational time was possibly wasted in the last incomplete beam, without providing any solution).

This geometric-growth variant appears to be a competitive algorithm on various problems in practice. Moreover, it appears that the average number of times a node is reopened is constant (or close to it in most cases). If opening a node can be done efficiently (for instance in $O(1)$), the iterative beam search can be more effective than a variant that stores all visited nodes (the storage usually costs $O(\ln n)$).

[1] A complete search can deplete the search tree and detect it did. Exact methods rely on complete search.

Algorithm 5.1: Iterative Beam Search algorithm

Input: root node

```
D ← 1
while stopping criterion not met do
    Candidates ← {root}
    while Candidates ≠ ∅ do
        nextLevel ← ⋃_{n∈Candidates} children(n)
        Candidates ← best D nodes among nextLevel
    end
    D ← D× 2
end
```

In Proposition 5.2, we present an argument why the iterative beam search opens a (close to) constant amount of nodes.

Definition 5.1. *wide tree hypothesis: The behavior of a beam search is said to satisfy the wide tree hypothesis, if, for every level the beam search explores, the beam contains no less than its allowed size. An iterative beam search satisfies the wide tree hypothesis if it is satisfied at every iterative beam search iterations.*

We may note that the wide tree hypothesis constitutes a good approximation of the beam search behavior in all search trees we consider. Such an approximation does not hold on levels that are close to the root. However, we believe that the influence of the first levels is negligible in the search trees that will be encountered in this manuscript, and thus the following "perfect world" analysis is quite close to the expected behavior in real-world applications.

Proposition 5.2. *An iterative beam search with a growth factor $k > 1$ opens at most $\frac{k}{k-1}$ times more nodes than the total number of nodes it explores (provided it satisfies the wide tree hypothesis presented in Definition 5.1).*

Proof. Given the n-th iteration of the iterative beam search, in the worst case, k^n nodes are opened at least once. The average number of openings of a given node is:

$$\frac{\sum_{i=0}^{n} k^i}{k^n} = \sum_{i=0}^{n} \frac{1}{k^i} \leq \sum_{i=0}^{\infty} \left(\frac{1}{k}\right)^i = \frac{1}{1-\frac{1}{k}} = \frac{k}{k-1}$$

□

With $k = 2$ as described above, the iterative beam search opens at most twice the number of nodes. With $k = 3$, it opens on average 1.5 times a node. With $k = 1.5$, it opens in average 3 times a node. The parameter k allows controlling the number of reopenings at the expense of the ability to provide often solutions. Decreasing it would allow providing more solutions at the expense of the number of node reopenings. We believe that tuning this parameter is of little practical relevance. Moreover, setting it to 2 makes the algorithm parameter-free, which is a very important quality that should not be underestimated.

6. Numerical results

We present some numerical results in order to assess the contribution of each component. Due to a large number of instances and algorithmic components combinations, we choose to perform short iterative beam search runs (20 seconds) on a large set of instances (the BB and ES datasets have in total 680 instances). In Subsection 6.1, we discuss the influence of different dominance strategies (namely the local, k-filter and global dominances). We study $2^3 = 8$ different combinations from no dominance used to all three selected at the same time in the algorithm. We show that the local dominance does not contribute to the performance of the algorithm and the k-filter and global dominance strategies contribute, depending on the instance type. We performed these tests using the bound guide. In Subsection 6.2, we discuss the performance of the various guides we investigated (namely the bound guide, the probabilistic guide in the "naive" and "improved" versions, and the expectation guide). Finally, in Subsection 6.3, we design an efficient

	a2.n10	a10.n10	a25.n10	a100.n10	a2.n50	a10.n50	a25.n50	a100.n50	a2.n100	a10.n100	a25.n100	a100.n100	total
no dom	614.2	202.8	233.4	144.2	**540.0**	**137.2**	**139.5**	71.3	**520.2**	**123.6**	**122.7**	**60.3**	2909.4
loca	614.0	202.8	233.4	141.8	**540.0**	136.9	**139.5**	71.3	**520.2**	**123.6**	**122.7**	60.1	2906.3
kfil 1	**615.9**	**203.7**	235.6	**144.6**	539.7	**137.2**	**139.5**	**71.4**	**520.2**	**123.6**	**122.7**	**60.3**	**2914.4**
loca.kfil 1	**615.9**	203.2	235.6	143.1	539.7	136.9	**139.5**	71.3	**520.2**	**123.6**	**122.7**	60.1	2911.8
glob	615.0	203.6	236.2	144.0	539.7	136.9	**139.5**	71.3	**520.2**	**123.6**	**122.7**	**60.3**	2913.0
loca.glob	615.0	203.3	236.2	143.0	539.7	136.9	**139.5**	71.1	**520.2**	**123.6**	**122.7**	60.1	2911.3
glob.kfil 1	**615.9**	203.5	**236.5**	144.1	539.7	136.9	**139.5**	71.3	**520.2**	**123.6**	**122.7**	**60.3**	2914.2
loca.glob.kfil 1	**615.9**	203.5	235.4	143.1	539.7	136.9	**139.5**	71.3	**520.2**	**123.6**	**122.7**	60.1	2911.9

Table 1
Dominance combination contribution – 20 seconds run on the ES benchmark – expectation guide

	a2.n10	a4.n10	a8.n10	a24.n10	a2.n100	a4.n100	a8.n100	a24.n100	total
no dom	623.9	435.4	309.4	235.8	**539.7**	333.8	**215.5**	111.4	2804.9
loca	620.9	435.5	308.4	227.2	**539.7**	333.7	211.7	111.4	2788.5
kfil 1	669.9	521.1	437.1	**385.6**	**539.7**	**335.8**	210.9	111.4	3211.5
loca.kfil 1	669.9	521.2	436.0	**385.6**	**539.7**	**335.8**	210.2	**112.1**	3210.5
glob	662.5	518.6	425.9	375.4	**539.7**	334.8	210.9	111.9	3179.7
loca.glob	662.9	**524.9**	430.5	374.3	**539.7**	334.8	210.9	111.9	3189.9
glob.kfil 1	**672.9**	521.8	**452.6**	**385.6**	**539.7**	334.8	210.5	**112.1**	**3230.0**
loca.glob.kfil 1	**672.9**	521.7	**452.6**	**385.6**	**539.7**	334.8	210.5	**112.1**	3229.9

Table 2
Dominance combination contribution – 20 seconds run on the BB benchmark – expectation guide

algorithm (combining the best dominance strategies and the best guides, according to the previous experiments) and run it for 900 seconds on each instance to compare with the state-of-the-art. This method is competitive with the state-of-the-art and sometimes, even returns 18 new best-known solutions in a relatively short amount of time.

Note that we do the following hypothesis to make the numerical experiments computationally acceptable (*i.e.* shorter than a week): The performance of a guide is independent of the dominance strategy considered. Indeed, a guide may not be efficient with a dominance breaking strategy and efficient with another. While such a situation is unlikely, it may lead to a missed good combination. During our preliminary experiments, we did not observe any correlation between a guide and dominance strategy usage. Moreover, as the algorithm we build using this analysis obtains competitive results (Subsection 6.3), we believe that a "missed combination" would not improve much further the results we obtained.

Numerical results have been obtained using an *Intel(R) Core(TM) i7-6700HQ CPU @ 2.60GHz* with 16GB RAM running on a ubuntu 19.04 distribution. To speed-up the experiments, we ran 4 tests in parallel.

6.1. Dominance strategies comparison

Tables 1 and 2 respectively presents the results obtained by the 8 variants on the ES and BB datasets. an algorithm (composed of the bound guide and the iterative beam search). Each line indicates an algorithm configuration. "no dom": indicates that the algorithm does not uses dominances, "loca" the local dominance, "kfil" the k-filter dominance (set at 1 as the state-of-the-art [8], we discuss later the choice of this parameter) and "glob" the global dominance. Each column indicates a set of instances with different parameters $|\Sigma|$ and n.

To better understand the contribution of each dominance component, we consider another representation: we display for each type of dominance the best value achieved by an algorithm using it and the best value achieved by algorithms that do not use it. If both values are comparable, it means that the component is not required to obtain good solutions (*i.e.* another component, or component combination is enough to emulate the same effect). If values are different, it means that the component has an impact (positive or negative) on the algorithms. Tables 3 and 4 respectively show the algorithmic component analysis on the ES and BB benchmarks.

Discussions: On the ES dataset, the algorithm using only the global dominance seems to produce the best solutions average-wise. However, the choice between k-filter, global dominance, or a combination of both is unclear as values are very close (less than 1/1000). On the BB benchmark, the algorithm combining the global dominance and k-filter finds the best average solution. We may note that the difference is clearer than on the ES dataset. We may also notice, that, on both datasets, 7/8 algorithms perform worse using the local dominance compared to their version without it.

Considering the contribution of each component on the ES dataset (Table 3), we may notice that the global dominance strategy could be replaced or removed (as it is possible to reach similar performance without using it). The

	a2.n10	a10.n10	a25.n10	a100.n10	a2.n50	a10.n50	a25.n50	a100.n50	a2.n100	a10.n100	a25.n100	a100.n100	nb best	total
glob	615.9	203.6	**236.5**	144.1	539.7	136.9	139.5	71.3	520.2	123.6	122.7	60.3	1	2914.3
no glob	615.9	**203.7**	235.6	**144.6**	**540.0**	**137.2**	139.5	**71.4**	520.2	123.6	122.7	60.3	**5**	**2914.7**
kfil 1	**615.9**	**203.7**	**236.5**	**144.6**	539.7	137.2	139.5	**71.4**	520.2	123.6	122.7	60.3	**5**	**2915.3**
no kfil	615.0	203.6	236.2	144.2	**540.0**	137.2	139.5	71.3	520.2	123.6	122.7	60.3	1	2913.8
loca	615.9	203.5	236.2	143.1	540.0	136.9	139.5	71.3	520.2	123.6	122.7	60.1	0	2913.0
no loca	615.9	**203.7**	**236.5**	**144.6**	540.0	**137.2**	139.5	**71.4**	520.2	123.6	122.7	**60.3**	**6**	**2915.6**

Table 3
Dominance contribution analysis – 20 seconds run on the ES benchmark – expectation guide

	a2.n10	a4.n10	a8.n10	a24.n10	a2.n100	a4.n100	a8.n100	a24.n100	nb best	total
glob	**672.9**	**524.9**	**452.6**	385.6	539.7	334.8	210.9	112.1	**3**	**3233.5**
no glob	669.9	521.2	437.1	385.6	539.7	**335.8**	**215.5**	112.1	2	3216,9
kfil 1	**672.9**	521.8	**452.6**	**385.6**	539.7	**335.8**	210.9	**112.1**	**5**	**3263.2**
no kfil	662.9	**524.9**	430.5	375.4	539.7	334.8	**215.5**	111.9	2	3195.6
loca	672.9	**524.9**	452.6	385.6	539.7	335.8	211.7	112.1	1	3235.3
no local	672.9	521.8	452.6	385.6	539.7	335.8	**215.5**	112.1	1	**3236.0**

Table 4
Dominance contribution analysis – 20 seconds run on the BB benchmark – expectation guide

	a2.n10	a10.n10	a25.n10	a100.n10	a2.n50	a10.n50	a25.n50	a100.n50	a2.n100	a10.n100	a25.n100	a100.n100	total
glob.kfil 1%	**615.9**	203.8	236.4	144.1	**539.7**	136.9	**139.5**	**71.3**	519.6	**123.6**	**122.7**	**60.3**	2913.8
glob.kfil 5%	615.1	203.4	236.3	144.4	538.3	136.5	**139.5**	**71.3**	519.5	123.5	**122.7**	**60.3**	2910.8
glob.kfil.1	**615.9**	203.5	**236.5**	144.1	**539.7**	136.9	**139.5**	**71.3**	**520.2**	**123.6**	**122.7**	**60.3**	2914.2
glob.kfil.10	615.8	203.8	236.4	144.5	538.3	136.9	**139.5**	**71.3**	519.6	**123.6**	**122.7**	**60.3**	2912.7
glob.kfil.100	614.1	203.1	235.2	144.6	538.2	136.2	139.1	**71.3**	518.7	123.0	122.2	60.1	2905.8
kfil 1%	615.8	203.6	236.3	144.6	**539.7**	136.9	**139.5**	**71.3**	519.6	**123.6**	**122.7**	**60.3**	2913.9
kfil 5%	614.9	**203.9**	236.4	**144.7**	538.3	136.5	**139.5**	**71.3**	519.6	123.5	**122.7**	**60.3**	2911.6
kfil.1	**615.9**	203.7	235.6	144.6	**539.7**	**137.2**	**139.5**	**71.3**	**520.2**	**123.6**	**122.7**	**60.3**	**2914.3**
kfil.10	615.8	203.6	236.4	**144.7**	**539.7**	136.9	**139.5**	**71.3**	519.6	**123.6**	**122.7**	**60.3**	2914.1
kfil.100	614.1	203.2	235.2	144.6	538.2	136.3	139.1	**71.3**	518.7	123.0	122.2	60.1	2906.0

Table 5
k-filter parameter study on the ES benchmark – expectation guide

	a2.n10	a4.n10	a8.n10	a24.n10	a2.n100	a4.n100	a8.n100	a24.n100	total
glob.kfil 1%	**673.5**	539.5	**462.7**	**385.6**	539.6	334.7	210.0	**112.3**	3257.9
glob.kfil 5%	673.3	544.2	**462.7**	**385.6**	536.7	335.3	210.1	110.3	3258.2
glob.kfil.1	672.9	521.8	452.6	**385.6**	**539.7**	335.3	210.5	112.1	3230.5
glob.kfil.10	673.2	**544.8**	**462.7**	**385.6**	539.6	334.9	210.2	111.8	**3262.8**
glob.kfil.100	670.5	531.8	**462.7**	**385.6**	532.4	324.1	204.0	107.1	3218.2
kfil 1%	**673.5**	539.3	**462.7**	**385.6**	539.5	334.6	210.0	**112.3**	3257.5
kfil 5%	673.2	531.5	**462.7**	**385.6**	**539.7**	335.2	210.2	111.9	3250.0
kfil.1	669.9	521.1	437.1	**385.6**	**539.7**	**335.8**	**211.9**	111.4	3212.5
kfil.10	**673.5**	541.8	**462.7**	**385.6**	539.5	334.7	210.2	112.1	3260.1
kfil.100	670.5	531.8	**462.7**	**385.6**	532.4	324.1	204.0	107.1	3218.2

Table 6
k-filter parameter study on the BB benchmark – expectation guide

k-filter seems relatively useful and the local dominance to be avoided. On the BB dataset (Table 4), it appears that both the k-filter and global dominance procedures help to find better solutions. Thus, both should be included if possible, in algorithms for solving the BB dataset. Similarly to the ES dataset, the local dominance did not prove to be an efficient component and can be avoided.

k-filter parameter choice: The k-filter strategy involves setting a size k to the elite-set. We study various values of this k parameter, namely $\{1, 10, 100, 1\%, 5\%\}$ where 1% and 5% correspond to the proportion of the current beam search iteration being part of the k-filter. Tables 5 and 6 show the influence of the k-filter parameter on the ES and BB datasets. It appears that $k = 1$ works better on the ES dataset, and $k = 10$ works better on the BB dataset.

6.2. Guide comparisons

Tables 7 and 8 respectively present the performance comparison of the different guidance strategies we investigate in this article. We ran iterative beam search algorithms with no dominance strategies using the bound, probabilistic in its naive and corrected versions, and the expectation guide. Each line represents a variant using a given guide.

	a2.n10	a10.n10	a25.n10	a100.n10	a2.n50	a10.n50	a25.n50	a100.n50	a2.n100	a10.n100	a25.n100	a100.n100	total
Expect	**614.2**	202.8	233.4	143.3	539.8	137.2	139.5	71.3	520.2	123.6	122.7	60.2	2908.2
Bound	588.7	187.5	210.8	130.1	526.6	131.3	131.5	67.0	509.5	118.8	116.3	56.7	2774.8
Proba	614.0	**203.2**	**235.3**	144.2	**540.2**	**137.3**	**139.9**	**72.0**	**520.6**	**123.7**	**122.9**	**60.7**	**2914.0**
ProbaNaive	614.0	**203.2**	**235.3**	**144.3**	**540.2**	**137.3**	139.6	**72.0**	**520.6**	**123.7**	**122.9**	**60.7**	2913.8

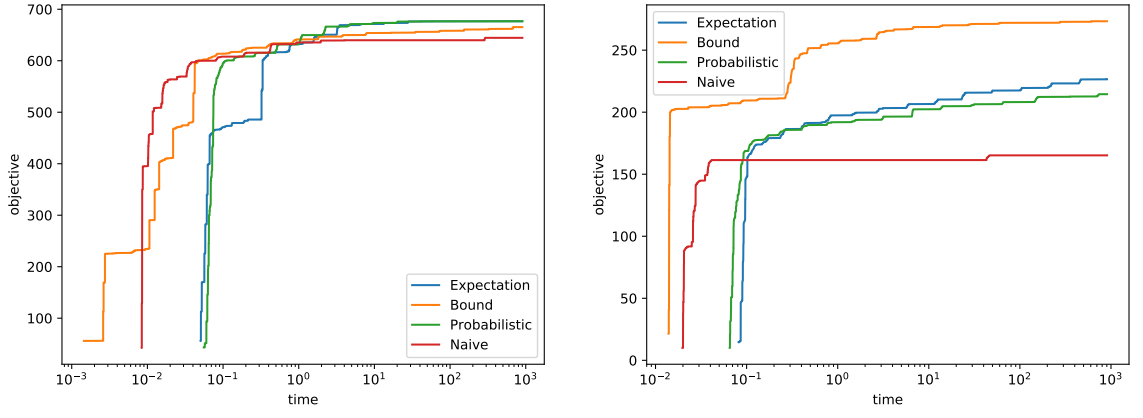
Table 7
Guide performance analysis: 20sec on the ES benchmark – no dominance

	a2.n10	a4.n10	a8.n10	a24.n10	a2.n100	a4.n100	a8.n100	a24.n100	total
Expect	623.9	435.4	309.4	230.9	539.7	333.8	210.0	111.4	2794.5
Bound	618.5	**466.6**	**379.6**	**327.9**	**560.1**	**381.0**	**260.7**	**136.9**	**3131.3**
Proba	**624.8**	436.0	313.8	275.6	534.8	328.2	204.7	102.3	2820.2
ProbaNaive	622.2	405.2	260.2	146.4	509.6	290.7	161.4	70.9	2466.6

Table 8
Guide performance analysis: 20sec on the BB benchmark – no dominance

Discussions: On the ES dataset, we observe that the probabilistic guides and the expectation guide perform better than the bound guide. We may notice that, on this dataset, the differences between the naive and corrected probabilistic guide is quite small. On the BB dataset, we observe that the bound guide performs much better than the other guides. we also notice that the corrected probabilistic guide performs well compared to the naive version and can even compete with the expectation guide.

Note that time plays an important role. In short runs (20 seconds), the bound guide performs better. However, during long runs (900 seconds) on the BB dataset, the expectation guide performs better on some instances (number of sequences $n = 10$). We observe this time behavior in Figure 6. We compute the average LCS for classes of instances, and for each time step. Figure 6a shows that the expectation (and probabilistic) guide performs better than the bound guide after 30 seconds on instances with $n = 10$ sequences. However, Figure 6b shows that the bound guide remains better (by far) on instances with $n = 100$ sequences.



(a) Iterative beam search runs with no dominance and the 4 guides we study in this article on instances with alphabet size $|\Sigma| = 2$, $n = 10$ sequences and sequences of size $m = 1000$

(b) Iterative beam search runs with no dominance and the 4 guides we study in this article on instances with alphabet size $|\Sigma| = 8$, $n = 100$ sequences and sequences of size $m = 1000$

Figure 6: Average performance profile of different guides on two different sets of BB instances ran during 900 seconds.

6.3. Comparison with the state-of-the-art

In the previous sections, we discussed various algorithmic components (dominance strategies, guides and the iterative beam search). We analyse in Subsections 6.1 and 6.2 the dominance and guide performances. As discussed before, the choice of dominances and guides is crucial and depends on the instance properties (*i.e.* if the sequences are dependant or not, the alphabet size and the number of sequences). Depending on these parameters, we prescribe the following components:

Instances			lit. best		bound + k-filter		expect. + k-filter		proba. + k-filter		naive + k-filter	
$\|\Sigma\|$	n	m	$\|s\|$	$t_{best}\|s\|$	$\|s\|$	$t_{best}\|s\|$	$\|s\|$	$t_{best}\|s\|$	$\|s\|$	$t_{best}\|s\|$	$\|s\|$	$t_{best}\|s\|$
2	10	1000	676.7	152.0	665.3	403.49	676.7	41.55	676.7	31.8	644.5	29.97
	100	1000	563.6	264.3	**568.3**	336.38	549.6	483.76	542.3	331.43	509.6	0.04
4	10	1000	**545.5**	85.6	541.8	140.38	545.3	21.7	545.1	96.07	420.1	10.14
	100	1000	390.2	362.7	**394.0**	293.18	346.8	453.38	335.2	244.15	290.9	1.13
8	10	1000	462.7	16.9	462.7	38.93	462.7	8.71	462.7	11.68	277.3	1.05
	100	1000	**273.4**	179.8	273.3	283.81	226.6	330.26	214.5	386.63	165.2	4.65
24	10	1000	385.6	8.5	385.6	0.75	385.6	1.81	385.6	12.34	148.6	0.07
	100	1000	**149.5**	8.01	149.1	192.98	130.2	405.48	113.5	309.61	70.9	0.06

Table 9
900 seconds run in the BB Benchmark – The "naive" guide correspond to the probabilistic one without logarithms. The value k of k-filter depends on the number of sequences: for $n = 10$, k-filter with $k = 10$; for $n = 100$, k-filter with $k = 1$

Instances			lit. best		Results	
$\|\Sigma\|$	n	m	$\|s\|$	$t_{best}\|s\|$	$\|s\|$	$t_{best}\|s\|$
2	10	1000	**618.9**	323.2	618.3	166.94
	50	1000	540.9	302.2	**541.2**	276.81
	100	1000	522.1	324.6	**522.4**	345.81
10	10	1000	**205.0**	251.3	204.74	199.92
	50	1000	137.5	158.1	**137.78**	276.39
	100	1000	124.1	121.0	**124.24**	82.24
25	10	2500	236.6	374.8	**238.12**	164.77
	50	2500	140.4	239.8	**140.5**	89.68
	100	2500	123.4	223.6	**123.78**	191.03
100	10	5000	145.7	434.3	**146.56**	211.18
	50	5000	72.0	286.1	**72.26**	53.76
	100	5000	60.8	515.7	**61.02**	32.24

Table 10
900 seconds run in the ES Benchmark – probabilistic guide, k-filter with $k = 1$

- **On independent sequences (all but the BB dataset):** We recommend the probabilistic guide, k-filter with $k = 1$.
- **On dependent sequences and 10 sequences ($n = 10$ in the BB dataset):** We recommend the the expectation guide, k-filter with $k = 10$.
- **On dependent sequences and 100 sequences ($n = 100$ in the BB dataset):** We recommend the the bound guide, k-filter with $k = 1$.

To demonstrate the efficiency of these prescriptions, we perform 900 second runs (times used in the literature) on all instances and compare the performance against the current state-of-the-art [8]. Tables 9, 10, 11, 12 and 13 present the results we obtained. The proposed method was able to improve on 18 instances or class of instances.

7. Conclusions & perspectives

In this article, we discussed various algorithmic components used to build tree search algorithms solving the longest common subsequence. In Section 3, we present 3 guides classically used by algorithms that reached top performance, namely the bound, probabilistic (naive), expectation guides, and propose a corrected probabilistic guide using *log* computations. We show that all of them are useful, depending on the instance properties (number of sequences, if the sequences are independent or not *etc.*). We may note that the corrected probabilistic guide performs well on independent sequences while the bound guide performs well on dependent sequences short runs. Finally, the expectation guide works well for dependent sequences and long runs. In Section 4, we investigate 3 dominance strategies, again, classically used by top-performance algorithms, namely, the local, k-filter, and global dominance strategies. We show

Instances			lit. best		Results	
$\|\Sigma\|$	n	m	$\|s\|$	$t_{best}\|s\|$	$\|s\|$	$t_{best}\|s\|$
4	10	600	**223**	173.6	222	44.46
	15	600	206	16.0	206	3.81
	20	600	195	130.0	195	104.64
	25	600	189	26.7	189	20.37
	40	600	177	457.5	177	250.65
	60	600	169	275.9	169	118.82
	80	600	164	337.6	164	134.67
	100	600	161	55.6	161	307.43
	150	600	155	57.2	155	79.71
	200	600	152	57.8	152	10.19
20	10	600	63	8.1	63	2.83
	15	600	53	4.4	53	0.19
	20	600	48	3.6	48	0.72
	25	600	45	7.1	45	0.39
	40	600	39	4.3	39	0.23
	60	600	36	14.5	36	3.67
	80	600	33	5.9	33	0.12
	100	600	32	7.6	32	0.3
	150	600	30	733.9	30	250.85
	200	600	28	14.1	28	0.18

Table 11
900 seconds run in the random data Benchmark – probabilistic guide, k-filter with $k = 1$

that the local dominance strategy, commonly used in many works, does not contribute to the algorithm performance on all scenarios we investigate. We also show that the k-filter and global dominance strategies improve the algorithm performance. In Section 5, we investigate an iterative beam search strategy that performs geometrically increasing beam runs. Such a strategy is simple, and in practice, efficient. Finally, we present numerical results in Section 6, showing the contribution of each algorithmic component, and show that one can build an efficient algorithm using this analysis. The algorithm we propose finds new best-known solutions for many instances.

Out of this study, we notice that the contribution of algorithmic components greatly depends on the type of instances considered (this is not surprising). However, to the best of our knowledge (and our surprise), no similar analysis has been done before. Many articles aim to find the best possible component parameter, but none analyses the impact of removing a component or choosing one. Performing this analysis, we observe that some components, play a major role in the algorithm performance for some instances, and demonstrate that choosing the right component depending on the instance properties allows obtaining state-of-the-art performance and to return new-best-known solutions.

In this article, we study some algorithmic components for the longest common sequence. We may note that one can apply a similar methodology to other problems and perform further analysis with more components. We study an iterative beam search strategy applied to the longest common sequence problem. We obtain a good performance using this search strategy. We may note that this algorithm have been used before on the sequential ordering problem [26] and cutting and packing problems [27, 12]. We advocate more research on this component as it is simple to implement and yields competitive results.

References

[1] Aziz, A., 1965. Dynamic programming .

[2] Blum, C., 2010. Beam-aco for the longest common subsequence problem, in: IEEE Congress on Evolutionary Computation, IEEE. pp. 1–8.

[3] Blum, C., Blesa, M.J., López-Ibáñez, M., 2009. Beam search for the longest common subsequence problem. Computers & Operations Research 36, 3178–3186.

[4] Bonizzoni, P., Della Vedova, G., Mauri, G., 2001. Experimenting an approximation algorithm for the lcs. Discrete Applied Mathematics 110, 13–24.

[5] Castelli, M., Beretta, S., Vanneschi, L., 2013. A hybrid genetic algorithm for the repetition free longest common subsequence problem. Operations Research Letters 41, 644–649.

Instances			lit. best		Results	
$\|\Sigma\|$	n	m	$\|s\|$	$t_{best}\|s\|$	$\|s\|$	$t_{best}\|s\|$
4	10	600	206	130.6	206	194.85
	15	600	**189**	740.1	188	209.2
	20	600	174	12.3	174	41.86
	25	600	**173**	38.3	172	221.64
	40	600	154	32.8	154	18.83
	60	600	154	510.3	154	240.0
	80	600	144	427.9	144	262.88
	100	600	139	548.7	139	25.31
	150	600	131	39.2	131	136.82
	200	600	126	288.0	126	74.57
20	10	600	72	136.7	72	180.15
	15	600	63	3.8	63	1.61
	20	600	55	7.1	55	1.34
	25	600	52	3.4	**53**	745.6
	40	600	50	138.6	50	77.1
	60	600	47	11.5	47	8.75
	80	600	44	132.5	44	211.48
	100	600	40	6.5	40	2.6
	150	600	38	21.4	38	6.02
	200	600	35	144.7	**36**	536.0

Table 12
900 seconds run in the rat Benchmark – probabilistic guide, k-filter with $k = 1$

[6] Chen, Y., Lu, H., Li, L., 2017. Automatic icd-10 coding algorithm using an improved longest common subsequence based on semantic similarity. PloS one 12.

[7] Chin, F., Poon, C.K., 1994. Performance analysis of some simple heuristics for computing longest common subsequences. Algorithmica 12, 293–311.

[8] Djukanovic, M., Raidl, G., Blum, C., 2019a. Finding longest common subsequences: New a anytime results .

[9] Djukanovic, M., Raidl, G.R., Blum, C., 2019b. A beam search for the longest common subsequence problem guided by a novel approximate expected length calculation, in: International Conference on Machine Learning, Optimization, and Data Science, Springer. pp. 154–167.

[10] Easton, T., Singireddy, A., 2006. A specialized branching and fathoming technique for the longest common subsequence problem .

[11] Easton, T., Singireddy, A., 2008. A large neighborhood search heuristic for the longest common subsequence problem. Journal of Heuristics 14, 271–283.

[12] Fontan, F., Libralesso, L., 2020. PackingSolver: a solver for Packing Problems. arXiv:2004.02603 [cs] URL: http://arxiv.org/abs/2004.02603. arXiv: 2004.02603.

[13] Golle, U., Rothlauf, F., Boysen, N., 2015. Iterative beam search for car sequencing. Annals of Operations Research 226, 239–254.

[14] Guenoche, A., 2004. Supersequences of masks for oligo-chips. Journal of bioinformatics and computational biology 2, 459–470.

[15] Gusfield, D., 1997. Algorithms on stings, trees, and sequences: Computer science and computational biology. Acm Sigact News 28, 41–60.

[16] Hayes, C.C., 1989. A model of planning for plan efficiency: Taking advantage of operator overlap., in: IJCAI, pp. 949–953.

[17] Hinkemeyer, B., Julstrom, B.A., 2006. A genetic algorithm for the longest common subsequence problem, in: Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation, Association for Computing Machinery, New York, NY, USA. p. 609–610. URL: https://doi.org/10.1145/1143997.1144105, doi:10.1145/1143997.1144105.

[18] Hsu, W., Du, M., 1984. Computing a longest common subsequence for a set of strings. BIT Numerical Mathematics 24, 45–59.

[19] Huang, K., Yang, C.B., Tseng, K.T., et al., 2004. Fast algorithms for finding the common subsequence of multiple sequences, in: Proceedings of the International Computer Symposium, IEEE press. pp. 1006–1011.

[20] Hunt, J.W., MacIlroy, M.D., 1976. An algorithm for differential file comparison. Bell Laboratories Murray Hill.

[21] Jansen, T., Weyland, D., 2007. Analysis of evolutionary algorithms for the longest common subsequence problem, in: Proceedings of the 9th annual conference on Genetic and evolutionary computation, pp. 939–946.

[22] Jiang, T., Li, M., 1995. On the approximation of shortest common supersequences and longest common subsequences. SIAM Journal on Computing 24, 1122–1139.

[23] Jiang, T., Lin, G., Ma, B., Zhang, K., 2002. A general edit distance between rna structures. Journal of computational biology 9, 371–388.

[24] Johtela, T., Smed, J., Hakonen, H., Raita, T., 1996. An efficient heuristic for the lcs problem, in: Third South American workshop on string processing, WSP, pp. 126–140.

[25] Julstrom, B.A., Hinkemeyer, B., 2006. Starting from scratch: Growing longest common subsequences with evolution, in: Parallel Problem Solving from Nature-PPSN IX. Springer, pp. 930–938.

[26] Libralesso, L., Bouhassoun, A.M., Cambazard, H., Jost, V., 2020. Tree searches for the Sequential Ordering Problem. URL: https://hal.archives-ouvertes.fr/hal-02374896. working paper or preprint.

Instances			lit. best		Results	
$\|\Sigma\|$	n	m	$\|s\|$	$t_{best}\|s\|$	$\|s\|$	$t_{best}\|s\|$
4	10	600	228	80.8	228	45.53
	15	600	206	92.5	206	19.38
	20	600	194	327.6	194	502.94
	25	600	196	128.2	196	107.05
	40	600	174	264.0	174	105.38
	60	600	168	49.8	168	51.91
	80	600	163	61.2	**164**	283.53
	100	600	160	71.5	**161**	142.21
	150	600	157	40.3	**158**	819.85
	200	600	156	582.5	156	39.35
20	10	600	77	14.6	77	8.47
	15	600	64	4.0	64	0.43
	20	600	61	28.9	61	7.83
	25	600	56	82.8	56	16.52
	40	600	51	110.4	51	87.83
	60	600	48	6.1	48	0.59
	80	600	46	7.1	46	0.2
	100	600	45	8.9	45	1.5
	150	600	46	27.7	46	2.24
	200	600	44	44.8	**45**	881.5

Table 13
900 seconds run in the virus Benchmark – probabilistic guide, k-filter with $k = 1$

[27] Libralesso, L., Fontan, F., 2020. An anytime tree search algorithm for the 2018 ROADEF/EURO challenge glass cutting problem. arXiv:2004.00963 [cs] URL: http://arxiv.org/abs/2004.00963. arXiv: 2004.00963.

[28] Lu, S.Y., Fu, K.S., 1978. A sentence-to-sentence clustering procedure for pattern analysis. IEEE Transactions on Systems, Man, and Cybernetics 8, 381–389.

[29] Mousavi, S.R., Tabataba, F., 2012. An improved algorithm for the longest common subsequence problem. Computers & Operations Research 39, 512–520.

[30] Reddy, D.R., et al., 1977. Speech understanding systems: A summary of results of the five-year research effort. department of computer science.

[31] Sellis, T.K., 1988. Multiple-query optimization. ACM Transactions on Database Systems (TODS) 13, 23–52.

[32] Shyu, S.J., Tsai, C.Y., 2009. Finding the longest common subsequence for multiple biological sequences by ant colony optimization. Computers & Operations Research 36, 73–91.

[33] Singireddy, A., 2003. Solving the longest common subsequence problem in bioinformatics. Master, Kansas State University, KS, Manhattan .

[34] Tabataba, F.S., Mousavi, S.R., 2012. A hyper-heuristic for the longest common subsequence problem. Computational biology and chemistry 36, 42–54.

[35] Wang, H., Zhong, J., Zhang, D., 2015. A duplicate code checking algorithm for the programming experiment, in: 2015 Second International Conference on Mathematics and Computers in Sciences and in Industry (MCSI), IEEE. pp. 39–42.

[36] Weyland, D., 2008. Simulated annealing, its parameter settings and the longest common subsequence problem, in: Proceedings of the 10th Annual Conference on Genetic and Evolutionary Computation, Association for Computing Machinery, New York, NY, USA. p. 803–810. URL: https://doi.org/10.1145/1389095.1389253, doi:10.1145/1389095.1389253.

[37] Yang, J., Xu, Y., Shang, Y., Chen, G., 2014. A space-bounded anytime algorithm for the multiple longest common subsequence problem. IEEE transactions on knowledge and data engineering 26, 2599–2609.

[38] Zhang, W., 1998. Complete anytime beam search, in: AAAI/IAAI, pp. 425–430.