Bioinformatics
doi.10.1093/bioinformatics/xxxxxx
Advance Access Publication Date: Day Month Year
Manuscript Category

Subject Section

SkipED: A Range Based SIMD Friendly Edit-Distance Algorithm for NGS Short Reads Alignment

Hongyi Xin [1,*], Co-Author [2] and Co-Author [2,*]

[1] Department, Institution, City, Post Code, Country and
[2] Department, Institution, City, Post Code, Country.

[*] To whom correspondence should be addressed.

Associate Editor: XXXXXXX

Received on XXXXX; revised on XXXXX; accepted on XXXXX

Abstract

Motivation: Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.
Results: Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Availability: Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Contact: name@bio.com
Supplementary information: Supplementary data are available at *Bioinformatics* online.

1 Introduction

Approximate String Matching, also known as fuzzy string matching, is a classical problem in the realm of string algorithms. The goal is to match a *search* or *pattern* string to the *reference* string while allowing for errors. There are numerous formulations to this problem, depending on the comparative size of the pattern and the reference. For similarly sized strings, the problem is often finding the optimal sequence of operations (called *edits*) need to transform the pattern string into the reference. Edits are commonly associated with a penalty score; common edits include insertions, deletions, and substitutions. Similarity between the pattern and the reference is then measure by the total *penalty score* of all edits. The goal of the approximate string problem, therefore, is to find the optimal sequence of edits with minimum total penalty score to transform the pattern into the reference. In the case where the pattern string is much smaller than the reference, the problem is often transformed to finding substrings of the reference which best match the pattern, and output all such locations in the reference.

Approximate string matching has a wide array of applications in computer science. It is one of the core compute units in modern Next-Generation-Sequencing (NGS) short read aligners (or mappers). The purpose of sequence alignment is to reconstruct DNA using short DNA fragments called *reads* from the sequencer machine with the help of a known reference genome. DNA reads are matched to the reference genome to find potential locations of the read in the sequenced genome under a suitable error theshold. When the similarity between the read and the reference segment is high, the mapper concludes that the read (with high probability) must have been sampled from the same position in the unknown target genome, based on the assumption that individuals from the same species have highly similar genomes. These potential locations are subsequently used to determine the final position of the reads and assemble the genome.

For short read aligners, approximate string matching computation is a large percentage of the overall runtime. This is because of two reasons: complex genome variations and large reference genome size. There are tens of millions of reads which need to be matched to the reference genome, and each read needs to be matched to a large number of possible locations in the reference, depending on the mapping strategy. Hence the total number of approximate string matching computations is very large. Therefore the necessity for not only fast and efficient matching algorithms but also high vectorizable and parallel algorithms is paramount, especially with the rise of modern compute infrastructure becoming increasing more parallel and SIMD friendly.

In this paper, we present an extension to the previously proposed Landau-Vishkin algorithm (or simply LV), which is an optimization over the Smith-Waterman algorithm specifically for approximate string matching with Levenshtein distance penalty scores. We show that the same principles of LV can be applied not only to Levenshtein distance penalty scores but also to any penalty positive scoring schemes. To achieve this, we first propose a generalization of the approximate string matching problem, called the *Leaping Toad* problem and show that all approximate string matching problems with positive penalty scores can be transformed into the leaping toad problem. Then we propose a general dynamic-programming solution of the leaping toad problem, **LEAP**, which is inspired by the core principle of LV. Finally we provide a bit-vector optimization over the LEAP problem are show that it is 2.5x faster than the state-of-the-art Levenshtein distance implementations and 10x faster than the state-of-the-art affine-gap-penalty implementations.

This paper makes the following contributions:

- It proposes the leaping toad problem, a generalization of all approximate string matching problem with positive penalty scores. It shows that all approximate string matching problem with positive penalty scores can be transformed into a leaping toad problem. In particular it shows the detailed procedure of transforming approximate string matching problems with Levenshtein distance penalty scores and affine-gap penalty scores into the leaping toad problem.
- It provides a new algorithm, LEAP, that solves the general leaping toad problem using similar principles from Landau-Vishkin's algorithm.
- It provides a detailed proof of the optimality of LEAP. The proof, which is different from the proof used in the Landau-Vishkin's paper, confirms that LEAP captures the minimum-score edit set between the two strings.
- It provides a bit-vector optimization over LEAP, which use a perfect hash function that utilizes properties of de Bruijn sequences to find the position of the most significant '1' in a binary bit-vector with simple operations.
- It shows that bit-vectorized LEAP is 2.5x faster than the state-of-the-art Levenshtein approximate string matching implementations and ???x faster than the state-of-the-art affine-gap approximate string matching implementations.

The rest of the paper is organized as the following: the Background section provides a detailed explanation of the approximate string matching problem, as well as past works and analysis of common solutions; the Method section describes the leaping toad problem, as well as its general solution, LEAP, and finally finished with the bit-vector optimization over leap; the Results section compares LEAP against the state-of-the-art Levenshtein distance implementations as well as affine-gap implementations; the Discussion section discuss the advantages as well as the limitations of LEAP; and finally the Conclusion section summarizes the paper.

2 Background

The simplest formulation of the approximate string matching problem is known as *Levenshtein* distance or *edit distance*. The allowable errors are insertions, deletions, and substitutions. Each of these errors has cost has unit cost independent of the alphabet. This can be viewed as a type of distance metric between strings.

Define an alphabet Σ, and a string $s \in \Sigma^*$. $s_i \in \Sigma$ denotes the i^{th} character of s, where $1 \leq i \leq n =$ length(s). $s_{x..y}$ denotes the substring of s from positions x to y, inclusive, with $x \leq y$. For $x > y$, we define it as the empty string, 0. We define the edit distance metric $d_{\texttt{edit}}(\cdot,\cdot) : \Sigma^* \times \Sigma^* \mapsto \mathbb{R}$.

Edit distance of $d_{\texttt{edit}}(s, r)$ has a dynamic programming solution taking $\mathcal{O}(mn)$ space and time for strings s and r of length m and n respectively. The subproblems are finding the edit distance given prefixes of the input strings. The base cases are the cost of matching to the empty string, which is just the cost of a series of insertions or deletions.

$$d_{\texttt{edit}}(0,0) = 0, d_{\texttt{edit}}(s_{1..i},0) = i, d_{\texttt{edit}}(0, r_{1..j}) = j$$

The recurrence comes from casing on the last error:

$$d_{\texttt{edit}}(s_{1..i}, r_{1..j}) = \min \begin{cases} \underbrace{d_{\texttt{edit}}(s_{1..i-1}, r_{1..j-1}) + 1_{s_i \neq r_j}}_{\texttt{substitution}} \\ \underbrace{d_{\texttt{edit}}(s_{1..i-1}, r_{1..j}) + 1}_{\texttt{insertion}} \\ \underbrace{d_{\texttt{edit}}(s_{1..i}, r_{1..j-1}) + 1}_{\texttt{deletion}} \end{cases}$$

This method is easily extensible to adding error-specific costs c_{ins}, c_{del}, and c_{sub}.

The sequence of edits transforming the search string to the reference can be found by backtracking through the edit-distance matrix, tracing the path from $dp[m][n]$ to $dp[0][0]$, where dp is the dynamic programming matrix.

The Levenshtein distance metric measures the quality of the end-to-end alignment between the pattern and the reference. This is an example of *global alignment*, which focuses on finding the lowest scoring alignment when matching the entire pattern sequence to the reference sequence. The prototypical algorithm, Needleman Wunsch, uses a weighted Levenshtein distance calculation where the error function for substitution is $f(a, b)$: $a, b \in \Sigma$ when substituting a for b and a gap (insertion/deletion) penalty $g(w)$ as a function of the gap width w. The substitution function f on matches where $a = b$ has a positive score, whereas mismatches and gaps corresponds to penalties. This problem becomes a maximization over the score, as the highest score now corresponds to the best match. In the case where $g(w) = c * w$ is a linear function of w, the recurrence becomes:

$$d_{\texttt{NW}}(s_{1..i}, r_{1..j}) = \max \begin{cases} \underbrace{d_{\texttt{NW}}(s_{1..i-1}, r_{1..j-1}) + f(s_i, r_j)}_{\texttt{substitution}} \\ \underbrace{d_{\texttt{NW}}(s_{1..i-1}, r_{1..j}) + c}_{\texttt{insertion}} \\ \underbrace{d_{\texttt{NW}}(s_{1..i}, r_{1..j-1}) + c}_{\texttt{deletion}} \end{cases}$$

The dynamic programming matrix is initialized as $d_{\texttt{NW}}(s_{1..i}, 0) = g(i), d_{\texttt{NW}}(0, r_{1..j}) = g(j)$. Once again, the optimal alignment is found using backtracking.

Local alignment on the other hand tries to find the best match between subsequences of the pattern and the reference. This is useful in the case where the pattern and reference share long common subsequences, but diverge otherwise. Global alignment would try to force a match between the divergent sequences, but local alignment does not penalize this divergence and assigns a score of 0. This Smith-Waterman algorithm solves local alignment, and differs from NW in the interpretation of the dynamic programming matrix; it changes to $d_{\texttt{SW}}(s_{1..i}, r_{1..j})$ being the best score for a suffix of $s_{1..i}$ and $r_{1..j}$.

$$d_{\texttt{SW}}(s_{1..i}, r_{1..j}) = \max \begin{cases} \underbrace{d_{\texttt{SW}}(s_{1..i-1}, r_{1..j-1}) + f(s_i, r_j)}_{\texttt{substitution}} \\ \underbrace{d_{\texttt{SW}}(s_{1..i-1}, r_{1..j}) + c}_{\texttt{insertion}} \\ \underbrace{d_{\texttt{SW}}(s_{1..i}, r_{1..j-1}) + c}_{\texttt{deletion}} \\ \underbrace{0}_{\texttt{empty string}} \end{cases}$$

The lowest possible score is an alignment with the empty string, or 0. Local alignment always has non-negative scores and initializes the starting row and column as 0, whereas global alignment would initialize the starting row and column through the gap penalty. Backtracking works slightly differently, since now we are finding a subsequence with the optimal match. It finds the largest score anywhere in the matrix, which corresponds to the end of the best match, and backtracks to a cell with a score of 0, signifying the beginning of the subsequence.

A hybrid alignment technique, called semi-global alignment, is global alignment without penalizing gaps in the beginning or end of the alignment. This is used when the pattern string is much shorter than the reference, so the loss in alignment from the difference in length is ignored. This uses a mixture of the Needleman-Wunsch and Smith-Waterman algorithms.

In addition to linear gap penalties, a more standard gap function used is the affine gap penalty, $g(w) = a*w+b$. A downside of linear gap penalty is that it does not distinguish between a single long contiguous gap versus many smaller gaps. The affine gap penalty is used to favor contiguous gaps by using b as the cost to initiate a gap, which is typically much larger than the cost per gap space, a. The standard implementations of Needleman-Wunsch and Smith-Waterman discussed above do not work with affine gaps, because they are gap memory independent; the gap penalty is a function of whether the previous substring ends in a gap or not.

Applying affine gap penalty to global alignment requires 3 dynamic programming matrices to keep track of scores ending in insertions or deletions. Each matrix handles a different ending condition: M for matches and mismatches, I for insertions, and D for deletions.

$$M[i][j] = \max \begin{cases} \underbrace{M[i-1][j-1] + f(s_i, r_j)}_{\texttt{mismatch/match}} \\ \underbrace{I[i-1][j-1] + f(s_i, r_j)}_{\texttt{insertion}} \\ \underbrace{D[i-1][j-1] + f(s_i, r_j)}_{\texttt{deletion}} \end{cases}$$

$$I[i][j] = \max \begin{cases} \underbrace{M[i-1][j] + a + b}_{\texttt{substitution}} \\ \underbrace{I[i-1][j] + b}_{\texttt{insertion}} \\ \underbrace{D[i-1][j] + a + b}_{\texttt{deletion}} \end{cases}$$

$$D[i][j] = \max \begin{cases} \underbrace{M[i][j-1] + a + b}_{\texttt{substitution}} \\ \underbrace{I[i][j-1] + a + b}_{\texttt{insertion}} \\ \underbrace{D[i][j-1] + b}_{\texttt{deletion}} \end{cases}$$

Backtracking here ... also affine for local? (TODO)

In many situations, an upper bound on the number of allowable errors k is known prior to performing alignment. Matches which fall under this threshold are only required. The Landau-Viskin algorithm leverages this fact and runs in $\mathcal{O}(kn)$ time and memory, where n is the length of the reference. It uses the fact that edit-distance is conserved on the diagonal for a sequence of matches, so it can jump along the diagonal to the position of the next error using precomputed the LCP array. The $\texttt{LCP}[i][j]$ array stores the length of the longest prefix which $s_{i..m}$ and $r_{j..n}$ share.

The $\texttt{LV}[d][e]$ matrix stores the maximal row along diagonal d with edit distance e, where d is calculated as $j - i$, where i is the row, and j is column. By conditioning on the last error, we get a recurrence for LV:

$$LV[d][e] =$$

$$\max \begin{cases} \underbrace{LV[d][e-1] + 1 + LCP(LV[d+1][e-1] + 2, LV[d+1][e-1] + d + 2)}_{\texttt{substitution}} \\ \underbrace{LV[d-1][e-1] + LCP(LV[d+1][e-1] + 1, LV[d+1][e-1] + d + 1)}_{\texttt{insertion}} \\ \underbrace{LV[d+1][e-1] + 1 + LCP(LV[d+1][e-1] + 2, LV[d+1][e-1] + d + 2)}_{\texttt{deletion}} \end{cases}$$

3 Methods

Both of the Levenshtein edit-distance problem and the affine gap edit-distance problem can be generalized as an optimal path finding problem under specific restrictions. We call this generalized optimal path finding problem the leaping toad problem. In this section, we first propose the leaping toad problem. Then we show how a general edit-distance problem can be converted to a leaping toad problem. Subsequently we propose an improved dynamic programming algorithm **LEAP** as a solution; followed by a proof of its optimality. In addition, we provide a de Bruijn sequence based bit-vector optimization over LEAP. Lastly, we discuss specific optimizations to the algorithm for affine-gap penalties.

3.1 The leaping toad problem

The leaping toad problem is described as the following:

There is a swimming pool of certain shape with two terminals and a finite number of lanes. Each lane has an integer number of vertices. A toad starts from one side of the pool and swims to the other side, moving forward one vertex at a time. Between two vertices, there can be a hurdle. Each lane can have zero or multiple hurdles. When bumped into a hurdle, the toad can cross the hurdle while spending a positive integer amount of energy. Different hurdles may cost different amount of energy. While swimming across the pool, the toad can also switch (*leap*) to any other lanes at any time, while also spending a positive integer amount of energy.

When the toad switches lanes (leaping), the leaping pattern (directions and the number of lanes switched) as well as the energy costs can be lane specific (each lane can have its own leaping patterns). For simplicity, we restrict the leaping pattern and the leaping energy cost with the following rules:

- The leaping pattern and energy cost is only lane dependent but not column dependent. In other words, it does not matter where the toad currently is in the lane. As long as the toad stays in the same lane, its leaping pattern and the associated leaping cost stays the same.
- The toad may move forward as it leaps to other lanes. The toad may also choose not move forward and only leaps vertically. However, the goad can never go backward.
- The energy cost has to be an integer.

A relaxation of the above rules are discussed in the Discussion section.

Finally, swimming in the same lane lane without leaping or crossing any hurdles consumes zero energy. The general goal of the leaping toad problem is to find the optimal path that consumes the minimum amount of energy to move the toad from the origin vertex to the destination vertex. Specifically, depending on the problem setup, the toad may choose from a range of origins and destinations.

Figure **??** shows an example setup of the swimming pool as well as the optimal path getting across the pool (in red). In this setup, as the figure shows, the toad starts at the first vertex of the middle lane on the left side of the pool and the goal is to travel to the last vertex of the middle lane on the right side of the pool. In a lane, black crosses between vertices are hurdles that the toad would have to cross should we choose to.

In this particular setup, the toad can only leap to neighboring lanes. When the toad leaps, it follows three different leaping patterns depending on which lane it is currently in. If the toad is in the middle lane, when it leaps, it also moves forward one column. If the toad is in any other lanes, then it moves vertically when it leaps towards the center lane and moves forward one vertex when it leaps away the center lane (leaping patterns shown as directed edges between vertices). Here we also set the energy consumption of crossing a hurdle as well as leaping to neighboring lanes as 1. The red line depicts an optimal path for the toad to travel across the pool. Notice that there can be multiple optimal paths with the same total energy cost (shown as dashed red lines).

There can many alternative setups to the leaping toad problem. For an alternative setup, a number of settings could be changed: 1) the energy cost of overcoming different hurdles can be different. For complex cases, one might want to set different costs to each individual hurdle. 2) The toad might leap differently as it crosses multiple lanes. For example, the toad might be able to leap up to three lanes and it does not move forward at all when it leaps to a neighboring lane but moves forward one step when it leaps two or more lanes. 3) The energy cost of leaping might not be linear to the number of lanes it switches. In an alternative setup, for example, the energy cost of crossing one lane in a single leap is 3 but crossing two or more lanes in a single leap costs 4 energy.

Figure **??** shows an alternative setup. In this setup, ?????????????

3.2 Conversion of the string matching problem to leaping toad problem

Both the Levenshtein and the affine gap string matching problem can be converted to the leaping toad problem. To convert both problems leaping toad problems, we first convert both string matching problems into an optimal path finding problem in a directed graph. Then we show that the optimal path problem in the converted directed graph is indeed a leaping toad problem.

The Levenshtein edit-distance problem can be easily converted into an optimal path finding problem in a directed graph. For a $L \times L$ edit-distance matrix, we assign each element of the matrix a unique vertex. For the Levenshtein edit-distance problem, according to the core recurrence function, a directional edge is drawn from vertex P of element $[x_P, y_P]$ to vertex Q of element $[x_Q, y_Q]$ if and only if $P \neq Q$ while $x_Q - x_P <= 1$ and $y_Q - y_P <= 1$ (an edge to the right, bottom and bottom-right element). On each edge we place an integer weight w, where $w = 0$ if $x_Q = x_P + 1$, $y_Q = y_P + 1$ and $A[x_Q - 1] = B[y_Q - 1]$; or $w = 1$ otherwise. The directed graph representation of the Levenshtein edit-distance problem is shown in Figure **??**. The objective function of the Levenshtein edit-distance function therefore becomes a graph traversal problem where we find a path from vertex S (of element [0,0]) to vertex T of (element $[L - 1, L - 1]$) such that the sum of all edge weights on this path is minimized.

For the Levenshtein edit-distance problem, the equivalent swimming pool setup and the equivalent leaping pattern is already shown in the example in Figure **??**. We call this the Levenshtein leaping toad setup. In general, given a $L \times L$ Levenshtein edit-distance matrix and a maximum edit-distance threshold E, we formulate the equivalent Levenshtein swimming setup as the following:

The swimming pool has a total number of $2E + 1$ lanes. The middle lane has L vertices and surrounding lanes have $L - i$ vertices, where i is number of lanes away from the center lane. As the figure shows, the right-end of all lanes are aligned while the left-end of the lanes forms a triangle shape. Each lane in the swimming pool represents a diagonal line in the edit-distance matrix with each vertex matching to an element in the edit-distance matrix. The center lane represents the central diagonal line of the edit-distance matrix from top-left to bottom-right and the ith lane above or below the center lane represents the ith diagonal line above or below the central diagonal line in the edit-distance matrix. A hurdle is placed in the center lane between the kth and the $k+1$th vertex if the $A[k]$ AND $B[k]$ of the two strings A and B differs. For the ith lane above (or below) the center lane, a hurdle is placed between the kth and the $k+1$th vertex if $A[k]$ and $B[k-i]$ (or $A[k-i]$ and $B[k]$) differs[1]. When the toad swims across the pool, it can only leap to neighboring lanes. When the toad is in the center lane, as it leaps, it also moves forward by one step. When it is on any other lanes, then it moves forward one step if it is leaping away from the center lane whereas it stays in the same column when it leaps towards the center lane.

Within each vertex, stores the minimum amount of required energy (as an integer) of the toad to reach to the position in the lane. The energy cost of overcoming a hurdle as well as switching lanes are both unit energy cost 1. For the global alignment problem, the goal of the Levenshtein leaping toad problem is to 1) determine if the toad can swim from the first vertex of the center lane to the last of the center lane while spending at most E energy and 2) if it can, then find the swimming path that costs the minimum amount of energy.

The Levenshtein leaping toad problem is indeed the optimal path finding problem in the directed graph of the Levenshtein approximate string matching problem. This can be shown with two facts: 1) the directional graph between the Levenshtein leaping toad problem and the Levenshtein edit-distance problem are identical and 2) the objective functions are identical.

The equivalence between the two directional graphs can be visualized in Figure **??**. For the vertex at the kth column of the ith lane above (or below) the center lane in the swimming pool, we assign its equivalent vertex in the Levenshtein direction as the vertex of element $[x, y]$ in the edit-distance matrix, where $x = k-i$ and $y = k$ (or $x = k$ and $y = k-1$). For example, the red, green and blue vertices highlighted in both figures are equivalent between the two graphs, according to the swimming pool setup. Recall that in our leaping toad problem setup, a toad can reach to an element in only two ways: 1) it comes from the previous vertex in the same lane or 2) it comes from a neighboring lane. For the latter, it can be further divided into two sub-cases: 1) vertices in the center lane and 2) vertices in other lanes. For vertices in the center lane, if the toad comes to the vertex from a neighbor lane, then it must have come through a diagonal leap. For vertices in other lanes, the toad would have come to the vertex through a vertical leap if the source lane is away from the center lane; and through a diagonal leap if the source lane is towards the center lane. In the figure, the red and the blue vertices represent sub-case 1) and sub-case 2) respectively. The red, green and blue arrows shows all possible paths of traveling to the three vertices in both graphs.

If we consider all possible leaps between all vertices and draw a directed edge from the source vertex to the destination vertex, using the rules of assigning equivalent vertices between the two graphs, it is easy to see that we would have added the same edges of the Levenshtein edit-distance graph. Furthermore, if we consider the energy cost of traveling between the source and the destination vertex as the weight of the edge between them, then the edge weight in the leaping toad graph would be exactly the same as the Levenshtein edit-distance graph.

Lastly, the objective of the leaping toad problem is to find a path from the first vertex of the center lane (vertex S in the Levenshtein edit-distance graph) to the last vertex of the center lane (vertex T in the Levenshtein edit-distance graph) with minimum energy cost. This objective function is also

[1] We number vertices according to the vertical column of the vertex. Only the center lane has column 0.

identical to the objective function of the Levenshtein edit-distance graph problem. Therefore, the Levenshtein edit-distance problem is a leaping toad problem.

For local alignment, as the objective function of the Levenshtein edit-distance problem changes from *finding an optimal path from the top-left element of the matrix to the bottom-right element of the matrix* (global alignment), to *finding an optimal path from any element of the top and left corner of the matrix to any element of the bottom and right corner of the matrix* (local alignment), we simply need to reflect the same changes in the leaping toad problem. Therefore, for local alignment, the objective function in the leaping toad setup changes to *finding an optimal path from the first vertex of any lane to the last vertex of any lane* with minimum energy cost.

For affine gap string matching problem, or in general for any custom-gap-penalty string-matching problem (with non-decreasing gap penalties for increasing gap distances), as the recurrence function of the dynamic programming solution takes more horizontal and vertical passages into consideration, we simply reflect the same changes in the corresponding leaping toad setup. Figure **??** shows the leaping toad setup for a specific affine gap penalty setting where mismatches are penalized with +2, gap openings are penalized with +3 and gap extensions are penalized with +1.

3.3 LEAP: the general solution of the leaping toad problem

Similar to approximate string matching problems, the leaping toad problem can be solved through dynamic-programming. Since we restrict the toad from ever going backward, the toad can only reach to a vertex from another vertex that is from the same column or from previous columns. Therefore, for each new column, we can find the optimal paths leading to its vertices, as well as the minimum traveling energy, by reusing the optimal path results from previous columns: for each new vertex, we find all prior-vertices that can reach to the target vertex in one step, then pick the prior-vertex that requires the least amount of combined energy of both reaching to the prior-vertex and the intermediate move. We repeat this process until either we have reached a destination vertex, or no vertices in a column is still within the energy budget.

A major drawback of the above naïve dynamic-programming solution of the leaping toad problem, as with the naïve dynamic-programming solution of the edit-distance problem, is that for each new vertex, we have to compare all of its previous-step vertices, then pick the vertex with the minimum overall energy cost. Similarly in backtrack, as we move one step backward at a time, we have to once again resolve the previous-step vertex for each and every vertex along the optimal path.

Inspired by the Landau-Vishkin algorithm for the edit-distance problem, we propose an improvement over the naïve solution of the leaping toad problem. We only consider switching lanes at vertices that are right before a hurdle. When there is no hurdle, we always let the toad swim forward; therefore avoid frequently checking possibilities of leaping from other lanes. We name this algorithm **LEAP**.

LEAP is developed upon a key observation that among all possible optimal paths with minimum energy costs, there must exist at least one optimal path that the toad **either never leaps or only leaps right before a hurdle.**

Theorem 1. *Among all optimal paths of the a leaping toad problem, there must exist one path that the toad either never switches lanes or only switches right before hurdles.*

Before proving the theorem, we first define a few terminologies: we refer to a path from the origin vertex to the destination vertex simply as a *path*. The path might or might not have any lane switches. Whenever there is a lane switch, we call it a *leap*. Between two leaps, the toad only goes forward and we call such straight segments of the path as *segments*. We further categorize segments into two groups: segments that end with the destination vertex or a hurdle as *complete segments* and segments that do not ends with such conditions as *incomplete segments*. We call the operation that extends the incomplete segment until it either reaches a destination vertex or a hurdle as *completing the segment*. Equipped with these terminologies, we are now ready to proof the stepping stone lemma of Theorem 1:

Lemma 1. *For a path with an incomplete segment, S, there must exist an alternative path that shares the same moving sequence before S, while completing S into S_c and have strictly no greater energy cost.*

Proof. To prove the lemma, we simply need to systematically find such an alternative path that supports the claim. Assume in the original path, after S, the path continues with a series of leaps and segments, denoted as a *moving sequence*, [L_1, S_1, L_2, L_3, S_2...], where S_i is the ith segment after S while L_j is the jth leap after S. Notice that between two leaps there can be only one or none segments, while between two segments there has to be at least a single leap.

Assuming that S_c is d vertices longer than S ($|S_c| = |S| + d$), we propose an alternative path that shares the same segments and leaps before S, followed by S_c, then continues with the same sequence of leaps [L_1, ... L_t], while skipping all the segments from S_1 to S_{k-1}, where $\sum_{i=1}^{k-1} |S_i| < d \leq \sum_{i=1}^{k} |S_i|$, and L_t is right before S_k in the original moving sequence.

If $d \neq \sum_{i=1}^{k} |S_i|$, which suggests that after taking [L_1, ... L_t], the alternative path merges with the original path somewhere in S_k, then we also add the latter half of S_k after the merge point. The alternative path is then completed with the same moving sequence after S_k.

If in the original path, there are not enough segments after S to match the length of d, then the alternative path simply takes the remaining leaps while skipping all the remaining segments. In this special case, when the toad reaches the end of the pool and cannot move forward anymore while leaping as there are no more vertices, we simply took out the forward component of the leap and move the toad vertically.

Compared to the original path, the alternative path obtained through above measure is guaranteed to have less or equal energy cost. This is because: 1) The energy cost before S in the original path and before S_c in the alternative path are identical as they take identical moving sequences. 2) The energy costs of the two paths after the merge point (if they do merge) are also identical, as the two paths also take identical moving sequences. 3) The energy cost of the leaping sequence after S_c and before the merge point of the alternative path, is strictly no greater than the energy cost of the moving sequence after S and before the merge point in the original path. This is because the original path takes the same leaping sequence [L_1, ... L_t] (hence consume the same amount of leaping energy) but it may also contain extra segments (which are skipped by the alternative path), which might incur extra hurdle-crossing energy cost. 4) The energy costs of S and S_c are identical since S_c is only a completion of S (the extension is free of hurdles so it costs zero energy).

Figure **??** depicts an example of converting a segment S in the original path (red) into S_c with an alternative path (blue) using the above procedure. Compared to the red path, the blue path consumes less energy, as the red path contain extra hurdle-included segments, which are skipped by the blue path.

With Lemma 1, we are now ready to proof Theorem 1. We prove through contradiction:

Proof. Assume there exists no optimal paths that either never leaps or only have complete segments. Then for any optimal path, it must take at least one leap while at least one of its segment is incomplete.

We randomly pick an optimal path P_1 and finds the first incomplete segment S_1 in this path. Following the procedure in Lemma 1, we can find an Alternative path P_2 that shares the same moving sequences before S_1, while completing S_1 into S_1c and maintaining the same energy cost (P_2 cannot have a smaller energy cost; otherwise P_1 is not optimal). We then find P_2's first incomplete segment S_2, and repeat the same procedure.

The above process can only iterate finite number of times since the procedure in Lemma 1 does not introduce new segments except from completing S into S_c (the procedure skips the segment sequence [S_1 ... S_{k-1} and may either shorten S_k or skip S_k, depending on the position of the merge point). Hence, each iteration removes a incomplete segment. Given that there are finite number of incomplete segments, hence there can only be finite number of iterations. The final product after all iterations is a path with no incomplete segments but has the same energy cost. However, according to our assumption, such path does not exist; hence, leads to a contradiction.

With Theorem 1, we now transform the general leaping toad problem of *finding an optimal path with minimum cost* to a sub-problem that *finds an optimal path which only contains complete segments*. As we have proven in Theorem 1, the resulting optimal path of the sub-problem must also be an optimal path of the general leaping toad problem.

LEAP solves the above sub-problem through an optimized dynamic-programming method that can be viewed as an extension of the Landau-Vishkin algorithm. LEAP can be summarized into four steps: 1) LEAP iterates through all intermediate energy costs from 0 to E and for each energy cost, LEAP iterates through all lanes. 2) For an intermediate energy cost e and a lane l, LEAP finds the furthest vertex v in l that is reachable at precisely the energy cost e from either a leap or a hurdle-crossing. 3) LEAP extends the segment at v (if permitted) until the segment hits a hurdle. 4) LEAP repeats step 2) and 3) until either a lane has reached to the destination vertex or all intermediate energy levels has been exhausted. The path that leads to the destination vertex is reported as the result. To summarize, LEAP uses a core recurrence function shown below:

$$\texttt{start}[l][e] = \min \begin{cases} \underbrace{d_{\texttt{edit}}(s_{1..i-1}, r_{1..j-1}) + 1_{s_i \neq r_j}}_{\texttt{substitution}} \\ \underbrace{d_{\texttt{edit}}(s_{1..i-1}, r_{1..j}) + 1}_{\texttt{insertion}} \\ \underbrace{d_{\texttt{edit}}(s_{1..i}, r_{1..j-1}) + 1}_{\texttt{deletion}} \end{cases}$$

The detailed pseudo-code of LEAP is shown in Algorithm **??**.

Theorem 2. *The result path returned by LEAP is indeed the optimal path of the sub-problem.*

We validate the correctness of Theorem 2 using two arguments. First of all, all segments in the result path of LEAP is guaranteed to be complete, because in step 3), LEAP always extends a segment until it reaches a hurdle, therefore no segment will end with a leap before reaching a segment. Secondly, for any energy cost e and any lane l, the last vertex extended in step 3) (if any) marks the furthest vertex that the toad can reach to in l using precisely e energy. Combining both arguments, we can conclude that if LEAP can find a path that reaches to the destination vertex under the energy cost $e < E$ while the toad cannot reach to the destination any other energy cost $e' < e$, then the energy cost of the optimal path of the leaping toad sub-problem must be e (otherwise, according to the second argument, for a smaller energy, the toad would have reached the destination already) and the result path returned by LEAP must be an optimal path.

Proof. The first argument is obvious. The second argument can be proven through induction.

Base case: When e = 0, since any leap or hurdle-crossing would consume a non-zero amount of energy, the furthest vertex the toad can reach in a lane with zero energy cost would be the last vertex in the lane before hitting a hurdle. Therefore, the second argument holds true for the basic case.

Induction step: Assuming for all intermediate energy costs $e' < e$, and for all lanes, the second argument always holds true. That is, for any lane l', the last vertex $end[l'][e']$ reached by step 3) in LEAP marks the furthest vertex the toad can reach in l' while consuming precisely e' amount of energy.

Now, to get to a vertex with e energy cost in lane l, the toad has to either leap from a lesser energy vertex in another lane or cross a hurdle from a lesser energy vertex in the same lane. Because LEAP has already calculated the furthest vertices of all lanes for all energy levels $e' < e$ (based on our assumption), we can conclude that step 2) of LEAP will find the furthest vertex in l such that it is reachable from either a leap or a hurdle-crossing at precisely e energy cost.

Finally the only remaining method for the toad to get to a vertex while costing e energy, is to swim straight, without running into a hurdle, from a previous e-energy vertex. This vertex is also captured by LEAP in step 3). Therefore, the argument is correct for the induction step.

Conclusion: After step 3), LEAP always reflects the furthest vertex the toad can reach in the target lane l under the target energy cost e.

Overall, this proves that Theorem 2.

The pseudo-code of the backtrack method is shown in Algorithm **??**.

3.4 De Bruijn sequence optimization

While LEAP can drastically reduce the number of comparisons in the dynamic-programming solution of the leaping toad problem, step 3) of LEAP still involves a costly loop that searches for the next hurdle.

With modern computers, by encoding the hurdle information as binary bit-vectors, we can significantly improve the performance of step 3) using de Bruijn sequences and bit-vector operations. The detailed technique is described in **?**. Here we provide a brief summary of the technique.

First, we encode the sequence of all forward edges (edges that go straight) of a lane as a binary bit-vector, where '0' denotes an edge that does not have a hurdle in between the two connected vertices while '1' denotes an edge that has a hurdle in it. For example, the middle lane in Figure **??** can be represented as '0101010010101010100001111100'.

Counting the number of edges before the next hurdle after the ith vertex is, therefore, equivalent to counting the number of '0's from the ith bit until we hit a '1' . After shifting the bit-vector i bits to the left, the problem then becomes finding the position of the most significant '1' of the resulting bit-vector, which is equivalent to counter the number of trailing '0's of the reverse bit-vector.

First proposed in the paper of **?**, counting the number of trailing '0's in a binary bit-vector can be carried out through a hash-table lookup with a perfect hash function. Assume the machine word has a length of 2^n bits. The least significant '1' of a vector b can be first singled out by b ANDing with its two's complement number $\bar{b}$ (computed through $NOTb + 1$). For example, for a machine of word size of $2^3 = 8$ bits, the least significant '1' of a vector b = 01001000 can be singled out by $bAND\bar{b}$ which is $b_{LSB} = 01001000AND(10110111 + 1) = 01001000AND10111000 = 00001000$ (LSB stands for least significant bit). Then, the number of trailing '0's can be computed by multiplying b_{LSB} with a pre-computed de Bruijn sequence dB_{seq} of 2^n bits (in our example, $n = 3$ and subsequently $dB_{seq} = 00011101$). Because b_{LSB} must be power of two, $b_{LSB} \times dB_{seq}$ essentially means shifting dB_{seq} to the left m times, where m is the number of trailing zeros. By taking the most significant n bits of the product (carried out through shifting the product to the right $2^n - n$ bits), we have then produced a unique number, a key, between [0 ... $2^n - 1$]. Finally, we can use the key to query a pre-computed lookup table of 2^n entries, which returns the pre-computed number of trailing ‘0’s in b_{LSB}. The example lookup table for $dB_{seq} = 00011101$ is provided below in Table **??**.

The pseudo code of finding the next hurdle is shown in Algorithm **??**.

4 Related Work

Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.

5 Results

Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text
Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text
Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text

Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.

6 Conclusion

Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text Text.

Acknowledgements

Text Text Text Text Text Text Text Text. Bofelli *et al*., 2000 might want to know about text text text text

Funding

This work has been supported by the... Text Text Text Text.

References

Bofelli,F., Name2, Name3 (2003) Article title, *Journal Name*, **199**, 133-154.

Bag,M., Name2, Name3 (2001) Article title, *Journal Name*, **99**, 33-54.

Yoo,M.S. *et al*. (2003) Oxidative stress regulated genes in nigral dopaminergic neurnol cell: correlation with the known pathology in Parkinson’s disease. *Brain Res. Mol. Brain Res.*, **110**(Suppl. 1), 76–84.

Lehmann,E.L. (1986) Chapter title. *Book Title*. Vol. 1, 2nd edn. Springer-Verlag, New York.

Crenshaw, B.,III, and Jones, W.B.,Jr (2003) The future of clinical cancer management: one tumor, one chip. *Bioinformatics*, doi:10.1093/bioinformatics/btn000.

Auhtor,A.B. *et al*. (2000) Chapter title. In Smith, A.C. (ed.), *Book Title*, 2nd edn. Publisher, Location, Vol. 1, pp. ???–???.

Bardet, G. (1920) Sur un syndrome d’obesite infantile avec polydactylie et retinite pigmentaire (contribution a l’etude des formes cliniques de l’obesite hypophysaire). PhD Thesis, name of institution, Paris, France.