Alignments Without Low-Scoring Regions

Zheng Zhang[*] Piotr Berman[*] Webb Miller[*]

February 10, 1998

Abstract

Given a strong match between regions of two sequences, how far can the match be meaningfully extended if gaps are allowed in the resulting alignment? The aim is to avoid searching beyond the point that a useful extension of the alignment is likely to be found. Without loss of generality, we can restrict attention to the suffixes of the sequences that follow the strong match, which leads to the following formal problem. Given two sequences and a fixed $X > 0$, align initial portions of the sequences subject to the constraint that no section of the alignment scores below $-X$. Our results indicate that computing an optimal alignment under this constraint is very expensive. However, less rigorous conditions on the alignment can be guaranteed by quite efficient algorithms. One of these variants has been implemented in a new release of the Blast suite of database search programs.

1 Introduction

It is widely appreciated that the dynamic programming algorithm for aligning two sequences (Needleman and Wunsch, 1970) can be viewed as computing an optimal path in a certain graph (e.g., Myers and Miller, 1989). For a simple example, assume that an alignment is awarded $+1$ whenever identical letters are matched and is penalized -1 whenever either different characters are matched or a character is matched with the gap symbol "–". Then the graph of Fig. 1 captures the problem of aligning sequences *abc* and *acbac*.

For this simple class of alignment graphs, columns are numbered from 0 to M and rows are numbered 0 to N (top to bottom), where M and N

[*]Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802.

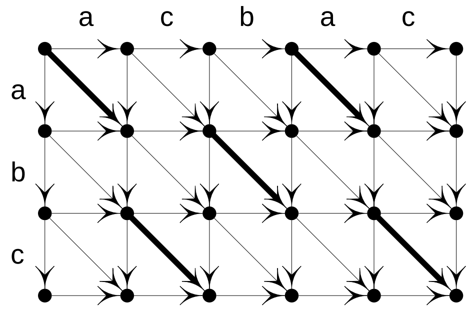

Figure 1: Graph model of a simple alignment problem. Dark edges (corresponding to aligning identical letters) score 1, and all other edges score –1.

are the two sequence lengths. There is a one-to-one correspondence between alignments of the two sequences and paths in the graph from $(0, 0)$ (the upper left corner) to (M, N) (the lower right corner) under which the alignment's score equals the sum of the edge costs along the path. In general, the node at grid point (i, j) represents the problem of aligning the first i letters of the first sequence and the first j letters of the second sequence.

Informally, the problem addressed by this paper is to align initial parts of the two sequences by searching the upper left portion of the graph, leaving the algorithm free to explore regions that look promising and to abandon the search in regions that look fruitless. An alternative is to determine *a priori* bounds on the region to be explored (e.g., see Chao et al., 1992, 1993) but for certain applications we prefer an approach that automatically adapts to the particular sequences being aligned.

Continuing informally, our strategy is to abandon the attempt to extend a path if we discover a segment of the path where the sum of edge weights falls below a fixed threshold, $-X$, where $X > 0$ is chosen in advance. Several precise formulations of this approach are given in Sections 2-5, and algorithms to compute optimal alignments are developed. Unfortunately, these algorithms are not as efficient as one might like. On the other hand, for somewhat looser requirements on the alignment, we present algorithms that are quite efficient.

This paper reports results of a theoretical study that was conducted as part of development of a new release (Altschul *et al.*, 1997) of the Blast program (Altschul *et al.*, 1990). That study concluded that Algorithm XConsistentSet (see Section 7, below) should be used. An experimental evaluation of that algorithm in its intended context of usage has previously been reported (Altschul *et al.*, 1997).

In 1989, as part of the original Blast development project, Gene Myers designed and implemented an algorithm, different from Algorithm XConsistentSet, that computes what is here called an X-consistent (but not X-closed) set. (See the brief discussion on p. 405 of Altschul *et al.*, 1990). On the other hand, a number of papers discuss superficially similar alignment algorithms that, in fact, solve a very different problem; those algorithms search a much reduced region of the dynamic programming matrix in cases where the two sequences are very similar, while ours gain efficiency when the sequences are very different. See Spouge (1991), Chao *et al.* (1997), and references cited therein.

An important problem not covered here is the statistical significance of alignment scores under this model. A manuscript in preparation will address that issue.

2 Formalization of the Problem

In practice, one uses a richer class of graphs than illustrated in Fig. 1. When alignment scores are chosen so that two separate gaps are penalized more than a single gap of the same total length (Gotoh, 1982), which is generally necessary to get biologically meaningful alignments, then the graph that models the alignment process is somewhat more complicated (Myers and Miller, 1989). Indeed, several more general classes of graphs have proved useful for biological sequence alignment (e.g., Chao *et al.*, 1994; Altschul, 1997; Zhang *et al.*, 1997). For instance, Fig. 2 depicts the structure of graphs used for comparing a DNA sequence with a protein sequence (Zhang *et al.*, 1997), where each grid point $(i, j) \in [0, M] \times [0, N]$, has three vertices, denoted $C(i, j)$, $D(i, j)$ and $I(i, j)$.

By the time one considers these more complicated graphs and the full range of orders in which they might reasonably be searched, it is conceptually more satisfying and essentially no more difficult to couch the discussion in a general graph framework. In this paper, we consider graphs that are

1. directed and acyclic,

2. the node set is of the form $\{0, 1, \ldots, n\}$, where the node numbers constitute the canonical topological numbering,

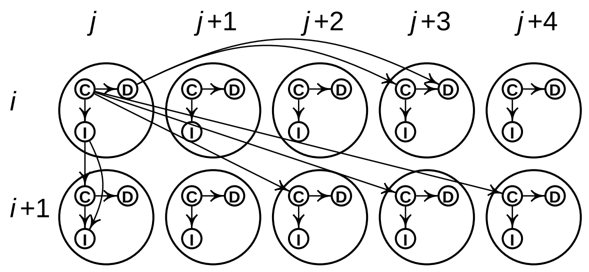

Figure 2: Edges leaving vertices at grid point (i, j) in a graph that models alignment of a DNA sequence and a protein sequence. The graph is obtained by modifying Fig. 6 of Zhang *et al.* (1997) as described in the third paragraph of the "Implementation" section of that paper. The C node at (i, j) represents all alignments of the prefix $S[1..i]$ and the prefix $T[1..j]$, for sequences S and T; the I node at (i, j) represents all alignments of $S[1..i+1]$ and $T[1..i]$ ending with a vertical edge; the D node at (i, j) represents all alignments of $S[1..i]$ and $T[1..j + 3]$ ending with a horizontal edge.

3. every node can be reached by a path from node 0,
4. the outdegree of a node (the number of edges leaving it) is bounded by a constant, and
5. each edge $e = p \rightarrow q$ has a real weight $w(e) = w(p, q)$.

We also consider our algorithms for a more specific class of graphs, which we call K-*queueable*, defined below, where K is an integer constant. (Later we omit the explicit references to K.)

The canonical topological numbering allows us to define *short* and *long* edges. (These notions are used solely to clarify the concept of K-queueable.) An edge $p \rightarrow q$ is short if $q \leq p + K$, and is long if for every other edge $p' \rightarrow q'$, $p' \leq p$ implies $q > q' - K$. (The notion of a long edge may be more intuitive if we define $\mathit{range}(p) = \max\{q' | \ p' \rightarrow q' \text{ and } p' \leq p \text{ for some } p'\}$; this way $p \rightarrow q$ implies that q is in the interval $[p + 1, \mathit{range}(p)]$. The edge $p \rightarrow q$ is short if it ends at one of the first K positions of this interval, and long if it ends at one of the last K positions.) A K-queueable graph has properties (1-5) and also satisfies

6. every edge is either short or long.

Note that the outdegree of a node of a K-queueable graph is at most $2K$.

For an example of a K-queueable graph, number the nodes in Fig. 2 row-by-row, left-to-right within a row, and in the order C, D, I at a grid point. Thus the C node at grid point (i, j) has number $3(ik + j)$, where there are k columns. With $K = 13$, there are 4 short edges and 5 long edges depicted in Fig. 2, so the graph is 13-queueable.

The *score* $s(\pi)$ of a path π is the sum of its edge weights. Fix $X \geq 0$. An *X-path* is a path that begins at 0 and has no segment of score strictly below $-X$. It will be convenient to characterize the X-paths as follows. Let $s_{\max}(\pi)$ be the maximum score of a prefix of π, and let the *terminal drop* of π be $tdrop(\pi) = s_{\max}(\pi) - s(\pi)$. Then π is an X-path if it starts at 0 and $tdrop(\pi') \leq X$ for every prefix π' of π. $Q(X)$ is the set of all endpoints of X-paths. For a set V of vertices, $|V|$ is the cardinality of V.

3 Finding All Vertices on X-Paths

Now our goal is to find $Q(X)$, the set of endpoints of X-paths. Assume the convention that the minimum of the empty set is ∞. Then for

$$Y(p) = \min\{tdrop(\pi)|\ \pi \text{ is an } X\text{-path ending at } p\}$$

we have $p \in Q$ iff $Y(p) < \infty$. Therefore it suffices to compute the value of Y for every node. Our first algorithm is a straightforward application of the following lemma, which introduces a useful “normalization” function.

Lemma 1 *Define $nml_X(x)$ to be 0 if $x < 0$, ∞ if $x > X$, and x otherwise. Then*

$$Y(q) = \begin{cases} 0 & \text{if } q = 0 \\ \min\{nml_X(Y(p) - w(p,q))|\ p \to q\} & \text{otherwise} \end{cases}$$

Proof. By induction on q. If $q = 0$, the claim is obvious. Otherwise,

$$\begin{aligned} &Y(q) = \min\{tdrop(\pi)|\ \pi \text{ is an } X\text{-path ending at } q\} = \\ &\min_{p \to q} \min\{tdrop(\pi)|\ \pi \text{ is an } X\text{-path ending at } p \to q\}. \end{aligned}$$

Therefore it suffices to show that for every predecessor p of q

$$\min\{tdrop(\pi)|\ \pi \text{ is an } X\text{-path ending at } p \to q\} = nml_X(Y(p) - w(p,q)).$$

Because we can apply the inductive hypothesis to p and $nml_X(x - w(p,q))$ is a nondecreasing function of x,

$$\begin{aligned}&nml_X(Y(p) - w(p,q)) =\\ &nml_X(\min\{tdrop(\pi) \mid \pi \text{ is an } X\text{-path ending at } p\} - w(p,q)) =\\ &\min\{nml_X(tdrop(\pi) - w(p,q)) \mid \pi \text{ is an } X\text{-path ending at } p\}.\end{aligned}$$

Let $\pi(q)$ be the extension of a path π ending at p by edge $p \to q$. To finish the proof it suffices to show that for an X-path ending at p,
(a) $\pi(q)$ is an X-path iff $nml_X(tdrop(\pi) - w(p,q)) < \infty$, and
(b) if $\pi(q)$ is an X-path, $tdrop(\pi(q)) = nml_X(tdrop(\pi) - w(p,q))$.

For (a), observe that $\pi(q)$ is not an X-path iff $tdrop(\pi') > X$ for some prefix π' of $\pi(q)$; because every proper prefix of $\pi(q)$ is a prefix of the X-path π, $\pi(q)$ is not an X-path iff $tdrop(\pi(q)) > X$. Now it is easy to see that $\pi(q)$ is not an X-path iff $tdrop(\pi) - w(p,q) > X$, i.e. iff $nml_X(tdrop(\pi) - w(p,q)) = \infty$.

For (b), it is easy to see that if $w(p,q) \leq 0$, then $tdrop(\pi(q)) = tdrop(\pi) - w(p,q) = nml_X(tdrop(\pi) - w(p,q))$ (the last equality uses the assumption that $\pi(q)$ is an X-path). If $0 < w(p,q) \leq tdrop(\pi)$, then $s_{\max}(\pi(q)) = s_{\max}(\pi)$, and hence $tdrop(\pi(q)) = s_{\max}(\pi(q)) - s(\pi(q)) = s_{\max}(\pi) - s(\pi) - w(p,q) = tdrop(\pi) - w(p,q) = nml_X(tdrop(\pi) - w(p,q))$ (the last equality holds because the argument of nml_X is between 0 and X). If $w(p,q) \geq tdrop(\pi)$, then $tdrop(\pi(q)) = 0 = nml_X(tdrop(\pi) - w(p,q))$, since $s_{\max}(\pi(q)) = s(\pi(q))$. □

Assume $q > 0$. According to Lemma 1, to compute $Y(q)$ it suffices to initialize $Y(q) \leftarrow \infty$, compute $Y(p)$ for every $p < q$, and for each $p \to q$ execute

$$Y(q) \leftarrow \min(Y(q), nml_X(Y(p) - w(p,q))).$$

Obviously, the above statement does not have to be executed if $Y(p) = \infty$, therefore it will be more efficient to store in a queue those nodes p that have $Y(p) < \infty$ and execute this statement for the entries of the queue only. The fact that the tentative value of $Y(p)$ is ∞ can then be represented by the fact that p was never inserted into the queue. In this manner the algorithm shown in Fig. 3 correctly computes $Y(q)$ for every node.

It is easy to see that the **while**-loop is executed exactly $|Q(X)|$ times. Because the nodes have bounded outdegree, the number of times the **for**-loop is executed is $O(|Q(X)|)$. Each execution of the **while**-loop involves a delete-min. The operations inside one execution of the **for**-loop involve

```
**Q** $\leftarrow$ an empty queue
$Y(0) \leftarrow 0$, insert$(0, \mathbf{Q})$
**while** **Q** is not empty **do**
{   $p \leftarrow$delete-min$(\mathbf{Q})$
    /* delete-min according to the value of $p$,
    i.e. its place in the canonical topological order */
    **for** every $q$ such that $p \to q$ **do**
    {   $y \leftarrow nml_X(Y(p) - w(p,q))$
        **if** $y < \infty$ **then**
            **if** $q \notin \mathbf{Q}$ **then**
                $Y(q) \leftarrow y$, insert$(q, \mathbf{Q})$
            **else if** $y < Y(q)$ **then**
                $Y(q) \leftarrow y$
    }
}
```

Figure 3: Algorithm XPathsEnds

simple arithmetic, insertion into the priority queue or an update of the Y-attribute of an element of the queue. If we use the data structure of Johnson (1982) we can perform these operations in the average time $O(\log\log N)$. If the graph is queueable, a simpler approach leads to a constant time: the priority queue can be a simple doubly link list, in which the nodes are stored according to their canonical topological order; then each insertion and each update is performed within distance K from one of the list ends. This proves

Theorem 1 *$Q = Q(X)$ can be determined in $O(\min\{N, |Q| \log\log N\})$ time, where there are N nodes. Moreover, if the graph is queueable, then Q can be determined in $O(|Q|)$ time.*

4 Highest Scoring X-paths

The problem addressed in this section is how to find, for every vertex $p \in Q(X)$, a highest scoring X-path that ends at p. As a result, we can find the overall highest scoring X-path. The algorithm that we describe is a form of dynamic programming, and its worst case running time is $O(|Q(X)|^2)$. While it is too slow to be useful for searching biosequence databases, it might be useful in some other context. At the very least, before deciding upon a fast heuristic that does not achieve this ideal, we should carefully evaluate the cost of the exact algorithm.

To formulate a dynamic programming algorithm for our problem, we introduce several definitions. Let $end(\pi)$ be the endpoint of π. An X-path π *dominates* another X-path ρ — denoted $\pi \sqsupseteq \rho$ — if $end(\pi) = end(\rho)$, $tdrop(\pi) \leq tdrop(\rho)$ and $s(\pi) \geq s(\rho)$. Clearly, we succeed in our goal if for every X-path ρ we find an X-path π such that $\pi \sqsupseteq \rho$. This in turn is achieved by computing a set $\Pi(X)$ with the following properties:

(i) for every X-path ρ there exists $\pi \in \Pi(X)$ such that $\pi \sqsupseteq \rho$;
(ii) no path in $\Pi(X)$ dominates another;
(iii) every prefix of a path in $\Pi(X)$ is also in $\Pi(X)$.

Property (i) directly formulates our goal; later we will show that property (ii) limits the size of $\Pi(X)$ to $|Q(X)|^2$, while property (iii) allows us to represent each path in $\Pi(X)$ by its endpoint and a pointer to its prefix containing all other nodes. However, it is not *a priori* clear that a set of paths with properties (i-iii) exists. We can show this fact by induction. Assume that a set of paths $\Pi^p(X)$ satisfies properties (ii) and (iii), but instead of (i) it satisfies

(i′) the penultimate node of every $\pi \in \Pi^p(X)$ belongs to $\{0, \ldots, p\}$;
(i″) for every X-path ρ with penultimate node in $\{0, \ldots, p\}$ there exists $\pi \in \Pi^p(X)$ such that $\pi \sqsupseteq \rho$.

It is easy to see how to construct $\Pi^0(X)$: it consists of the zero-length path from 0 to 0 and all one-edge paths starting at 0. To construct $\Pi^p(X)$, we can start with $\Pi^{p-1}(X)$ and then perform a series of insertions. More precisely, for every $\pi \in \Pi^{p-1}$ that ends at p and for every q such that $p \rightarrow q$ we extend π with the edge $p \rightarrow q$ and then test the resulting path π'. If it is dominated by a path that is already in Π^p, we discard π', otherwise we insert π' to Π^p, and remove the paths that π' dominates.

Before we analyze the efficiency of this algorithm, we must prove its correctness, i.e., that when it terminates, the set of paths Π stored in the data structure of the algorithm equals (i.e., satisfies the conditions of) $\Pi(X)$. We use induction: assume that when we start the processing of node p, $\Pi = \Pi^{p-1}(X)$; we need to show that at the end of this processing Π satisfies the properties of $\Pi^p(X)$. Property (i′) is assured because $\Pi^{p-1}(X)$ satisfies it and we do not insert paths that could violate it. Property (ii) is assured because we insert a new path only if it is not dominated by a path that is already in Π, and after such an insertion we remove all the paths that the new path dominates. Property (iii) could get violated in two ways: by insertion of a path that does not have all its prefixes in Π—and our method excludes such a possibility—and by deleting from Π a prefix of another path from Π. The latter is impossible as well, because if we remove in this stage a path ρ that is a proper prefix of σ, then $end(\rho) > p$, and thus the penultimate node of σ is larger than p; by (i$'$) $\sigma \notin \Pi$. Thus it remains to show that Π will satisfy (i$''$).

That means that for every path τ for which p is a penultimate node, Π will contain a path ψ such that $\psi \sqsupseteq \tau$. We can rewrite τ as $\rho\sigma$, where $end(\rho) = p$ and σ is the one-edge ending of τ. Because the penultimate node of ρ is in $\{0, \ldots, p-1\}$, the previously constructed set $\Pi^{p-1}(X)$ contains a path π such that $\pi \sqsupseteq \rho$. While processing p, we construct $\pi\sigma$, and the latter remains in our set of paths unless we have another path ψ such that $\psi \sqsupseteq \pi\sigma$. Thus we have the chain

$$\psi \sqsupseteq \pi\sigma \sqsupseteq \rho\sigma = \tau.$$

While the second step in this chain is not obvious, it follows directly from the lemma below.

Lemma 2 *If π and $\rho\sigma$ are two X-paths and $\pi \sqsupseteq \rho$, then $\pi\sigma$ is also an X-path and $\pi\sigma \sqsupseteq \rho\sigma$.*

Proof. First note that $\pi\sigma$ is indeed a path, because $\pi \sqsupseteq \rho$ implies that π and ρ have the same endpoint, which in turn is the starting point of σ. Next, we can show, by contradiction, that $\pi\sigma$ is an X-path. Suppose not; then it has a fragment with a score below $-X$. This fragment can be included neither in π, nor in σ, so it must be a concatenation of a suffix of π with a prefix of σ. Obviously, it suffices to choose a suffix of π with the minimal score, i.e., by definition, a score equal to $-tdrop(\pi)$. We can replace this suffix with a suffix of ρ having score $-tdrop(\rho)$, and because $tdrop(\pi) \le tdrop(\rho)$, the result scores below $-X$. This is a contradiction, because that latter path is a fragment of $\rho\sigma$, which in turn is an X-path.

The same argument—replacing a suffix of π with a suffix of ρ—shows that $tdrop(\pi\sigma) \le tdrop(\rho\sigma)$. Finally, $s(\pi\sigma) = s(\pi) + s(\sigma) \ge s(\rho) + s(\sigma) = s(\rho\sigma)$. □

The above lemma concluded the correctness proof of our algorithm. By the definition, one path dominates another only if both have the same end-point. Therefore our algorithm can be altered as follows: when we process p, we insert extensions of paths that end at p without testing (as long as they are X-paths); instead, when we start the processing of a node p, we "purge" the set of paths that end at p, so at the end none dominates another. For example, we can sort these paths so that the scores are non-increasing; then in a group of paths with the same score we keep exactly one—with the minimum terminal drop (now the scores are decreasing); lastly we make a sweep through the list of paths and delete every one that does not have the terminal drop smaller than the one of the predecessor. Algorithm OptXPaths, shown in Fig. 4, follows this description.

Now we need to analyze the efficiency of Algorithm OptXPaths. We split this task into two parts. First, we will show that properties (i-iii) imply $|\Pi(X)| \leq |Q(X)|^2$, while later we will argue that this algorithm performs only a constant number of steps for each element of $\Pi(X)$. First we need two additional lemmas.

Lemma 3 *Let $top(\pi)$ be $end(\pi')$, where π' is the longest prefix of π with the maximum score. Then for every $\pi, \rho \in \Pi(X)$ the equality $top(\pi) = top(\rho)$ implies $s_{\max}(\pi) = s_{\max}(\rho)$.*

Proof. Let π' and ρ' be the prefixes of π and ρ such that $end(\pi') = top(\pi)$ and $end(\rho') = top(\rho)$. Because of (iii), both π' and ρ' belong to $\Pi(X)$. Clearly, $s(\pi') = s_{\max}(\pi')$, hence $tdrop(\pi') = 0$, and similarly $tdrop(\rho') = 0$. By our assumption, $end(\pi') = end(\rho')$. Therefore either $\pi' \sqsupseteq \rho'$ (if $s(\pi') \geq s(\rho')$) or $\rho' \sqsupseteq \pi'$ (otherwise). By property (ii) of $\Pi(X)$, $\pi' = \rho'$, which implies $s_{\max}(\pi) = s(\pi') = s(\rho') = s_{\max}(\rho)$. □

Lemma 4 *Assume that $\pi, \rho \in \Pi(X)$, $\pi \neq \rho$ and $end(\pi) = end(\rho)$. Then*
(a) $tdrop(\pi) \leq tdrop(\rho)$ implies $s(\pi) < s(\rho)$;
(b) $s_{\max}(\pi) \leq s_{\max}(\rho)$ implies $tdrop(\pi) < tdrop(\rho)$.

Proof. By contradiction. If (a) does not hold, then $s(\pi) \geq s(\rho)$ and $tdrop(\pi) \leq tdrop(\rho)$, hence $\pi \sqsupseteq \rho$, which contradicts property (ii) of $\Pi(X)$. If (b) does not hold, then $tdrop(\pi) \geq tdrop(\rho)$, which further implies $s(\pi) = s_{\max}(\pi) - tdrop(\pi) \leq s_{\max}(\rho) - tdrop(\rho) = s(\rho)$; hence $\rho \sqsupseteq \pi$, which yields an identical contradiction. □

A corollary of the previous two lemmas is that a path $\pi \in \Pi(X)$ is uniquely determined by the pair $(end(\pi), top(\pi))$. By Lemma 3 $top(\pi)$ determines $s_{\max}(\pi)$, and given that we know $end(\pi)$, Lemma 4 shows that we may determine $tdrop(\pi)$ and $s(\pi)$. Note that no two different paths of $\Pi(X)$ may have the same end, score and terminal drop.

Also note that for any X-path π, both $end(\pi)$ and $top(\pi)$ belong to $Q(X)$. Thus the corollary shows that $|\Pi(X)| < |Q(X)|^2$. Moreover, we can represent $\pi \in \Pi(X)$ by a tuple

$$\mathbf{a} = (\ \mathbf{a}.end,\ \mathbf{a}.top,\ \mathbf{a}.s,\ \mathbf{a}.s_{\max},\ \mathbf{a}.\mathrm{pred}\ ) \qquad ,$$

where the first four items are self-explanatory, while **a**.pred is (the identifier of) the tuple representing π', a prefix of π that contains all its nodes except the last one; item pred allows one to traverse from **a** to the initial tuple (0, 0, 0, 0, NULL) and, in the process, read all the vertices of π. In our algorithm, all tuples with the same value of *end* will form a linked list.

Such a representation of $\Pi(X)$ can be computed in a similar manner to $Q(X)$ in Algorithm XPathsEnds. In particular, we again use a queue of vertices, **Q**, and we process vertices in the same order. Each vertex in the queue is in $Q(X)$ and has a list of tuples that represent paths ending at this vertex; these paths are obtained by extending the paths to the vertices already processed—and consequently, the already verified elements of $\Pi(X)$—by a single edge.

The queue is easy to initialize: we insert vertex 0 to **Q** and we give it the one element list consisting of the tuple (0, 0, 0, 0, NULL).

When a vertex is extracted by delete-min, we first sort its list of tuples so that $s_{\max}$ is non-decreasing. Then we scan this list and "purge", so it becomes increasing with respect to s and *tdrop*; as a result no path represented by a remaining tuple dominates another, and every path that was "purged" is dominated by one of the remaining ones. Subsequently, we generate all possible extensions of the paths on the list by one edge using procedure extend, which is presented together with the complete algorithm in Fig. 4. Note that this algorithm is very similar to the previous one.

Once Algorithm OptXPaths finishes the processing of vertex p, its list of tuples must contain a representation of an X-path that reaches p with the minimum terminal drop; thus exactly the same vertices are processed by Algorithm OptXPaths as by XPathsEnds, and in the same order. Therefore each line in the while-loop of Algorithm OptXPaths is executed $O(|Q(X)|)$ times. As we concluded from Lemma 3 and Lemma 4, after sorting and purging, the list of p has size at most $p+1 \leq |Q(X)|$. Therefore the running time of each call of extend is $O(|Q(X)|)$. Moreover, because the lists of

```
extend($p, q$)
    for every tuple $\mathbf{a}$ on the list of $p$ do
    {   $y \leftarrow nml_X(\mathbf{a}.tdrop - w(p,q))$
        if $y < \infty$ then
        {   if $q \notin \mathbf{Q}$ then insert($q, \mathbf{Q}$)
            if $y = 0$ then $t \leftarrow q$ else $t \leftarrow \mathbf{a}.top$
            append the list of $q$ with ( $q,\ t,\ \mathbf{a}.s + w(p,q),\ y,\ \mathbf{a}$ )
        }
    }

main()
    $\mathbf{Q} \leftarrow$ an empty queue
    $Y(0) \leftarrow 0$, insert($0, \mathbf{Q}$)
    create the list of 0 consisting of the tuple ( 0, 0, 0, 0, NULL )
    while $\mathbf{Q}$ is not empty do
    {   $p \leftarrow$delete-min($\mathbf{Q}$)
        sort the list of $p$
        purge this list so it is increasing in respect to $s$ and $tdrop$
        for every $q$ such that $p \rightarrow q$ do
            extend($p, q$)
    }
```

Figure 4: Algorithm OptXPaths

vertices are created by calls to extend, the sum of their sizes *before* sorting and purging is $O(|Q(X)|^2)$. Thus to show that Algorithm OptXPaths runs in time $O(|Q(X)|^2)$ it suffices that we can sort such a list in time proportional to its size or to $|Q(X)|$.

One can observe that the graphs generated by the alignment problems have not only bounded outdegree, but bounded indegree as well. Moreover, one can observe that a call of extend(p, q) appends the list of q with tuples that have nondecreasing *tdrop*'s, thus the sorting requires us to merge a bounded number of sorted lists, and that can be done in time proportional to the size of the whole list.

If we allow the indegree to be unbounded, we could still sort this list more efficiently than in the general case using bucket sort. We can find the set of possible values of *top*—the vertices reachable with $tdrop = 0$—using Algorithm XPathsEnds. Then these vertices (the t's) can be sorted according to s entries from the tuples of the form $(t,\ t,\ s,\ s,\ p)$. A hash table can allow us to retrieve quickly the rank of a t in this order. This precomputation allows us to bucket sort the "raw" list of a node q according to the rank of *top*'s. This clearly takes time proportional to the size of a list plus $|Q(X)|$. Note that we do not need to store multiple entries in a bucket: instead of inserting the second entry, we can keep the dominating one of the two.

We can summarize our conclusions as follows:

Theorem 2 *A set $\Pi(X)$ that contains a highest scoring X-path for each element of $Q(X)$ can be computed in time $O(|Q(X)|^2)$.*

There exists an important special case where we can improve this bound. Assume that $w(p, q)$ is an integer function, and that X is small. Then a list of a vertex, after sorting and purging, has at most $X + 1$ elements, corresponding to X-paths that end at this vertex while having the terminal drop of 0, 1, ..., X. Moreover, we can bucket sort the lists according to *tdrop*. The same argument as before yields

Theorem 3 *Assume that the edge weights are integer. Then a set $\Pi(X)$ that contains a highest scoring X-path for each element of $Q(X)$ can be computed in time $O(X|Q(X)|)$.*

5 X-Paths that are Optimal Paths

The goal of this section is to find $P(X)$, which is the set of all vertices one of whose optimal (highest score) paths is an X-path, and to determine that highest score for each vertex in $P(X)$.

Before we present an algorithm for this problem, let us consider how fast it can possibly be. Definitely, we need to consider all the vertices of $P = P(X)$. Then, given an edge $p \rightarrow q$ where $p \in P(X)$, and hence we know that some optimal path from 0 to p is also an X-path, we need to check whether one such path can be extended to q. Therefore we need to consider all the vertices in

$$\overline{P} = \{q|\ p \rightarrow q \text{ for some } p \in P\}$$

(note that $P - \{0\} \subset \overline{P}$). Finally, when we extend some path to a new vertex q, we need to verify that it is indeed an optimal path from 0 to q; to do this we need to consider all vertices that can conceivably be located on such a path, i.e. all the vertices of:

$$V = \{p|\text{ there exists a path containing } p \text{ from 0 to some } q \in \overline{P}\}$$

Theorem 4 *$P(X)$ can be determined in time $O(|V|)$.*

Proof. We describe an algorithm that uses two data structures: a queue $\mathbf{Q}$ that stores the vertices that we wish to process, and the set $\mathbf{S}$ of nodes that have been processed. For each node $p \in \mathbf{S}$ we store a record with two fields:

$$\begin{aligned} Y(p) &= \min\{tdrop(\pi)|\ \pi \text{ is an optimal path to } p \text{ and an } X\text{-path}\} \\ score(p) &= \max\{s(\pi)|\ \pi \text{ is a path from 0 to } p\} \end{aligned}$$

Note that, analogous to the definition of $Y(p)$ in Section 3, $p \in P$ iff $Y(p) < \infty$.

The algorithm shown in Fig. 5 uses a subroutine to processes vertex q, i.e. to compute $Y(q)$ and $score(q)$, and if, $Y(q) < \infty$, to assure that the successors of q will be processed later (by inserting them into $\mathbf{Q}$).

The proof of correctness of this algorithm consists of two parts: first we need to show that for every element of $q \in \mathbf{S}$, the fields $Y(q)$ and $score(q)$ are computed according to the definition, and next we need to show that eventually $\mathbf{S}$ will contain all the nodes that need to be considered. If both statements are true,

$$P = \{q \in \mathbf{S}|\ Y(q) < \infty\}.$$

```
process($q$)
    if $q \notin \mathbf{S}$ then
    {   $score(q) \leftarrow Y(q) \leftarrow -\infty$
        insert($q, \mathbf{S}$)
        for every $p$ such that $p \rightarrow q$ do
        {   process($p$)
            $s \leftarrow score(p) + w(p,q)$
            $y \leftarrow nml_X(Y(p) - w(p,q))$
            if $s > score(q)$ then
                $score(q) \leftarrow s$, $Y(q) \leftarrow y$
            else if $s = score(q)$ and $y < Y(p)$ then
                $Y(q) \leftarrow y$
        }
        if $Y(q) < \infty$ then
            for every $r$ such that $q \rightarrow r$ do
                enqueue($r, \mathbf{Q}$)
    }

main()
    $\mathbf{Q} \leftarrow$ an empty queue
    $Y(0) \leftarrow score(0) \leftarrow 0$
    insert(0,$\mathbf{Q}$)
    while $\mathbf{Q}$ is not empty do
        process(dequeue($\mathbf{Q}$))
```

Figure 5: Algorithm SymOptPaths

The former statement can be proved using induction, just as used in Section 3 for the function $Y(p)$. The latter follows directly from the way we enqueue at the end of process(q).

The analysis of the running time is also simple; it is clear that only the vertices of V are considered. While a particular vertex p can be considered many times, the total number of such actions is $O(|V|)$. In particular, a vertex can be considered either because it has been enqueued by one of its predecessors or because it is being checked by one of its successors. When q enqueues r, we charge it to the edge $q \rightarrow r$, and when q checks p, we charge it to $p \rightarrow q$; an edge is charged at most twice and only edges with beginnings in V are charged. Because we assume that the outdegree of vertices is $O(1)$, the total number of charges is $O(|V|)$.

It is easy to see that to each time we consider a vertex we can attribute execution of a finite number of lines, so that each execution of a line is attributed somewhere. Thus to prove our promised running time it suffices to show that each line is executed in constant time. The only line for which this statement is nontrivial is

if $q \notin \mathbf{S}$ **then**

which checks the membership of q in **S** and retrieves the record of q if one exists (so that the subsequent lines can be executed). We may implement **S** using a hash table. The only operations we need are insertion, membership test and retrieval. One possible implementation would use a hash table with chaining starting with a table that has one chain only. Once the average chain length rises above a certain constant, we can double the table size, concatenate all the chains, initialize the chains in the table to empty and reinsert all the records using a new hash function. It is easy to see that the number of operations used by that schema is on average $O(|V|)$. $\square$

6 Computation of an X-Closed Set

Intuitively, our goal is to find "the most meaningful" similarities between two given sequences, where "similarity" corresponds to a high-scoring path. In effect, our working hypothesis is that paths with X-drops are not meaningful; one approach to a high-scoring path with an X-drop is to remove the offending path fragment, giving two paths whose total score exceeds that of the original path by at least X.

We have seen that a highest scoring X-path can be computed. However, the procedure we presented is quite expensive to run and one can be pessimistic whether anything better than a dynamic programming method can be found. An alternative approach is to disregard an alignment corresponding to an X-path whenever there is a higher scoring alignment of the same two sequences, and the previous section's algorithm to compute $P(X)$ appeared very efficient. However, it may happen that the set V that determines the running time is much larger than the set $P(X)$ being computed. The flaw in the definition of $P(X)$ is that we need to consider meaningless paths (with X-drops) to disqualify the meaningful ones (as having lower score), and this can consume the bulk of our effort. Below we describe a property of vertex sets that allows us to spend constant time per vertex and to consider only meaningful paths.

Vertex set D is called *X-consistent* if for every $p \in D$ there is an X-path π to p that is highest scoring over *all* paths from 0 to p that stay in D. An X-consistent set D is *X-closed* if for every vertex p, if $D \cup \{p\}$ is X-consistent then $p \in D$. Note that if D is X-consistent then $D \subseteq Q(X)$, and if D is X-closed then $P(X) \subseteq D$.

We introduce notation that allows us to define X-consistent (and X-closed) sets inductively, and, eventually, algorithmically. Given a set D, we can define the following functions for the elements of $\overline{D}$:

$$\begin{aligned}
\mathit{Path}_D(p) &= \{\pi \mid \text{ path } \pi \text{ starts at 0, ends at } p \text{ and all} \\
&\quad\ \text{vertices of } \pi, \text{ with a possible exception} \\
&\quad\ \text{of } p, \text{ belong to } D\ \} \\
\mathit{score}_D(p) &= \max\{s(\pi) \mid \pi \in \mathit{Path}_D(p)\} \\
\mathit{HPath}_D(p) &= \{\pi \in \mathit{Path}_D(p) \mid s(\pi) = \mathit{score}_D(p)\} \\
Y_D(p) &= \min\{\mathit{tdrop}(\pi) \mid \pi \text{ is an } X\text{-path and } \pi \in \mathit{HPath}_D(p)\}
\end{aligned}$$

It follows directly from the above definitions that a set of vertices D is X-consistent iff $Y_D(p) < \infty$ for every $p \in D$. Such a set is also X-closed if $Y_D(p) = \infty$ for every $p \in \overline{D} - D$. The following lemma shows how to define the functions score_D and Y_D inductively.

Lemma 5 *The functions score_D and Y_D satisfy the following recurrence:*
$\mathit{score}_D(0) = Y_D(0) = 0$*, and for* $q > 0$
$\mathit{score}_D(q) = \max\{\mathit{score}_D(p) + w(p,q) \mid p \to q \text{ and } p \in D\}$
$Y_D(q) = \min\{\mathit{nml}_X(Y_D(p) - w(p,q)) \mid p \to q,\ p \in D \text{ and } \mathit{score}_D(p) + w(p,q) = \mathit{score}_D(q)\}$

Proof.. By induction on q. For $q = 0$ the claim is obvious (even if $0 \notin D$, in which case the path from 0 to 0 is still in $Path_D(0)$). The inductive step follows from the fact that every path in $\pi \in HPath_D(q)$ must end with an edge, say $p \to q$; in this case the path π' obtained by removing this edge belongs to $HPath_D(p)$. By induction, $score_D(p) = s(\pi')$, hence $score_D(q) = score_D(p) + w(p, q)$, and consequently the recursive formula cannot yield a result that is lower than the correct one. Moreover, if π is an X-path, so is π'; if π is an X-path from $HPath_D(q)$ that has the minimum terminal drop, then π' is also an X-path; by our inductive assumption $Y_D(p) \leq tdrop(\pi')$ and the result of our recursive formula for $Y_D(q)$ is at most $nml_X(Y_D(p) - w(p, q)) \leq tdrop(\pi)$. It is equally easy to see that the recursive formula cannot lead to an overly high value of $score_D(q)$ or an overly low value of $Y_D(q)$. □

The above recurrences give a method for finding an X-closed set R. We initialize R with $\{0\}$ and consider every vertex q (from 1 to n) as a candidate for the membership in R. Because when we consider q we already know $R \cap \{0, \ldots, q-1\}$, we can compute $score_R(q)$ and $Y_R(q)$ using the recurrence; we insert q into R iff $Y_R(q) < \infty$. Clearly, set R computed in this manner is X-closed.

Actually, we can characterize the outcome of this method more precisely. Order vertex sets according to the lexicographic order of their membership vectors, more formally

$$A \succ B \text{ iff } \min(A \oplus B) \in A;$$

where $A \oplus B$ is the symmetric difference of A and B. One can show that R is the maximum X-closed set in terms of this order. Suppose that D is X-closed, $D \neq R$ and $q = \min(R \oplus D)$. Then $D \cap \{0, \ldots, q-1\} = R \cap \{0, \ldots, q-1\}$ and therefore $Y_D(q) = Y_R(q)$ $(= x)$. If $x = \infty$, then q can belong neither to D nor to R, a contradiction. If $X < \infty$, then our algorithm places q in R.

Theorem 5 *The X-closed set R that is maximal in the ordering $\succ$ can be computed in time $O(|R| \log\log N)$, and in time $O(|R|)$ if the graph is queueable.*

Proof. The algorithm shown in Fig. 6 computes the set R in the same sense that Algorithm XPathsEnds computes $Q(X)$: vertex p is an element of R if it is removed from the queue $\mathbf{Q}$ with a finite value of $Y(p)$. In a sense, Algorithm XClosedSet is a hybrid of Algorithms XPathsEnds and

```
$\mathbf{Q} \leftarrow$ an empty queue
$score(0) \leftarrow Y(0) \leftarrow 0$, insert$(0, \mathbf{Q})$
**while** $\mathbf{Q}$ is not empty **do**
{   $p \leftarrow$delete-min$(\mathbf{Q})$
    **if** $Y(p) < \infty$ **then**
        **for** every $q$ such that $p \rightarrow q$ **do**
        {   $y \leftarrow nml_X(Y(p) - w(p,q))$
            $s \leftarrow score(p) + w(p,q)$
            **if** $q \notin \mathbf{Q}$ **then**
            {   $Y(q) \leftarrow y,\ score(q) \leftarrow s,\ pred(q) \leftarrow p$
                insert$(q, \mathbf{Q})$
            }
            **else if** $s > score(q)$ **then**
                $Y(q) \leftarrow y,\ score(q) \leftarrow s,\ pred(q) \leftarrow p$
            **else if** $s = score(q)$ and $y < Y(q)$ **then**
                $Y(q) \leftarrow y,\ pred(q) \leftarrow p$
        }
}
```

Figure 6: Algorithm XClosedSet

SymOptPaths. Its similarity with XPathsEnds is so extensive that we can copy the proof of the running time from Theorem 1.

Therefore it remains to argue that XClosedSet computes $score(q)$ and $Y(q)$ exactly as the method discussed above. Indeed, the first difference is that above we computed $score(q)$ as the maximum, over p such that $p \in R$ and $p \rightarrow q$, of $score(p) + w(p,q)$, while the Algorithm XClosedPaths updates $Y(q)$ whenever it inserts such a p into R (by dequeueing p with finite $Y(p)$); clearly the outcome is the same. The difference in the computation of $Y(q)$ is of the same nature. A minor complication is that we are computing a conditional minimum; every time we increase our estimate of the final $score(q)$, we restart the computation of the minimum (as the values considered before were found invalid). □

The fact that R is the maximum X-closed set in ordering $\succ$ does not depend on the choice of the canonical topological ordering, because both Y_D

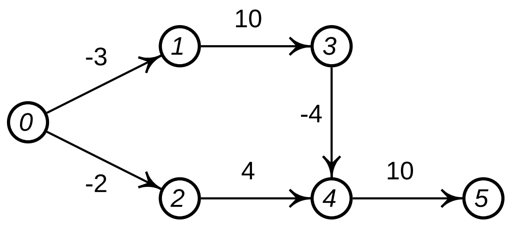

Figure 7: Graph used to illustrate $R(X)$

and $score_D$ are defined in terms of graph predecessors, which in turn must preceed a given vertex in every topological order. This shows that while the execution of Algorithm XClosedSet depends on the choice of a canonical topological order, the resulting set R does not.

Because set R depends solely on the value of X, it should be denoted $R(X)$. One may think that the fact that the X-closed set is the highest in $\succ$ ordering is positive, but it may actually be detrimental to our chief goal, that is finding an X-path that scores as high as possible given that we have only limited computational resources. For example, admitting a vertex "early" may prevent us from finding some superior X-path later. This phenomenon is illustrated in Fig. 7, where $R(2)$ contains an X-path of score 12, while $R(3)$, which includes vertex 1, finds a best path of score 7.

7 Faster Computation of an X-Consistent Set

Although Algorithm XClosedSet does attain the goals of producing a high-scoring X-path and spending only $O(1)$ time per investigated node, it is worthwhile to search for an even faster algorithm. In particular, Algorithm XClosedSet spends approximately half of its time manipulating Y-values, and efficiency can be gained if one is willing to accept upper bounds on paths' terminal drops in place of their exact values. Doing so permits more nodes to be investigated (by raising X) in a given running time, which sometimes finds a higher scoring path.

Algorithm XConsistentSet proceeds similarly to XClosedSet, but without computing Y values. Because our implicit convention for the output set refers to these values, XConsistentSet computes the output set R' explicitly. When a node p is removed from the queue $\mathbf{Q}$, we know a highest scoring path π that goes from 0 to p through R', and the score of π, $score(p)$. If we know that π is an X-path, we can insert p into R' and R' remains consistent. As in the case of XClosedSet, it suffices to verify that the terminal drop of π is at most X; this is true because every proper prefix of π was already verified.

Recall that the terminal drop of π equals $s(\pi) - s_{\max}(\pi)$, thus to provide an upper bound for this value, we can substitute $s_{\max}(\pi)$ with an upper bound. There are two cases. In the first, $s_{\max}(\pi) = s(\pi)$; then even if our "upper bound" on $s_{\max}(\pi)$ is too low, we do insert p to R' properly, because the terminal drop of π is zero. In the second, $s_{\max}(\pi)$ is a score of a proper prefix of π, and consequently of an X-path that was computed (and verified) already. Then we can upper bound $s_{\max}(\pi)$ with the highest score for all such paths; the value of this highest score is stored in variable *max_score*. The remaining details are contained in Figure 7. Using the same reasoning as in the previous section we can conclude:

Theorem 6 *Algorithm XConsistentSet computes an X-consistent set $R' = R'(X)$ in time $O(|R'| \log\log N)$, and in time $O(|R'|)$ if the graph is queueable.*

The set $R'(X)$ computed by Algorithm XConsistentSet is X-consistent but not necessarily X-closed. Somewhat surprisingly, $R'(X)$ is not always a subset of the set $R(X)$ computed by Algorithm XClosedSet, as shown by the example in Fig. 9. There, $R(2) = \{0, 1, 2, 3\}$ and Algorithm XClosedSet computes a best X-path of score 8 (ending at node 3), whereas $R'(2) = \{0, 1, 4, 5\}$ and Algorithm XConsistentSet computes an X-path of score 13 (ending at node 5). Note that $P(2) = \{0, 1, 2, 3\}$, illustrating that $R'(X)$ need not contain $P(X)$.

8 Finding Better X-Paths

So far, we have focused on efficient ways of finding "good" X-paths. Because we apply the concept of X-path in database searches, the speed of the algorithm was of paramount importance. However, as the cost of CPU cycles keeps decreasing, it is also worthwhile to consider a way of improving the accuracy of the algorithm if we can increase the running time by a constant factor.

The approaches discussed so far fall into two classes. The fast algorithms are finding X-consistent sets; for each vertex in such a set the algorithm computes the best score of a path that traverses from 0 to that vertex

```
$\mathbf{Q} \leftarrow$ an empty queue, $R' \leftarrow \emptyset$
$max\_score \leftarrow score(0) \leftarrow 0$, insert$(0, \mathbf{Q})$
while $\mathbf{Q}$ is not empty do
{   $p \leftarrow$delete-min$(\mathbf{Q})$
    if $max\_score - score(p) \leq X$ then
    {   insert$(p, R')$
        $max\_score \leftarrow \max(max\_score, score(p))$
        for every $q$ such that $p \rightarrow q$ do
        {   $s \leftarrow score(p) + w(p, q)$
            if $q \notin \mathbf{Q}$ then
                $score(q) \leftarrow s$, $pred(q) \leftarrow p$, insert$(q, \mathbf{Q})$
            else if $s > score(q)$ then
                $score(q) \leftarrow s$, $pred(q) \leftarrow p$
        }
    }
}
```

Figure 8: Algorithm XConsistentSet

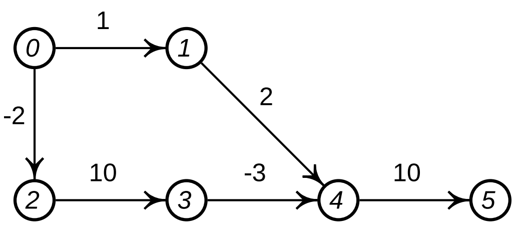

Figure 9: Graph used to illustrate the faster heuristic.

through this set, and the (estimate of) corresponding terminal drop. X-consistency assures that this path is an X-path.

The exact algorithm, OptXPaths, computes for each vertex of $Q(X)$ a set of X-paths that dominates all X-paths with this endpoint. Effectively, OptXPaths computes the table of the tradeoff between the score and the terminal drop. One can look at this process as follows. OptXPaths expands the original graph by replicating its vertices; a replica of a vertex p corresponds to a particular terminal drop value. However, the vertices corresponding to the terminal drop values of X or more are pruned. (The replicated vertices are the tuples formed by OptXPaths.) The set $Q(X)$, expanded in this manner, becomes X-consistent.

The idea of the intermediate algorithm is to expand the graph by replicating each vertex into $k+1$ many vertices (where k is an arbitrary constant), where each replica corresponds to a probabilistic approximation of the terminal drop. Within this graph we seek an X-closed subset of vertices using the same method as in Algorithm XClosedSet.

To make the formulation of the algorithm simpler, we multiply all edge weights by k/X, so that X-paths becomes k-paths. A replica of vertex p has the form of (p, i), $i \in \{0, \dots, k\}$. When we process a vertex (p, i) we consider every q such that $p \to q$; in the expanded graph the corresponding edge is $(p, i) \to (q, j)$ where j is selected at random by function nextlevel. The new algorithm, shown in Fig. 10, parallels closely Algorithm XClosedSet. It is easy to see that Algorithm ApproxOptXPaths computes (implicitly) a set of X-paths. Intuitively, it should yield more X-paths than XClosedSet, but it is also possible that the converse is true in a specially constructed graph. Nevertheless, in our tests this algorithm frequently found more X-paths and higher scores than did XClosedSet.

9 Acknowledgements

We thank David Lipman for initiating this project and for his help and encouragement along the way. This work was supported by grant LM05110 from the National Library of Medicine.

10 References

Altschul, S. 1997. Generalized affine gap costs for protein sequence alignment. To appear in *Proteins.*

```
nextlevel($i, v$)
    **if** $i = k$ **or** $i + v \geq k$ **then**
        return $k$
    **else if** $i + v \leq 0$ **then**
        return 0
    **else**
    {   let $j$ be an integer and $0 \leq x < 1$ such that $i + v = j + x$
        with probability $x$ add 1 to $j$
        return $j$
    }
main()
    $\mathbf{Q} \leftarrow$ an empty queue
    $score(0,0) \leftarrow Y(0,0) \leftarrow 0$, insert$((0,0), \mathbf{Q})$
    **while** $\mathbf{Q}$ is not empty **do**
    {   $(p, i) \leftarrow$delete-min$(\mathbf{Q})$
        **if** $Y(p,i) < \infty$ **then**
            **for** every $q$ such that $p \to q$ **do**
            {   $y \leftarrow nml_X(Y(p,i) - w(p,q))$
                $s \leftarrow score(p,i) + w(p,q)$
                $j \leftarrow$ nextlevel$(i, w(p,q))$
                **if** $(q,j) \notin \mathbf{Q}$ **then**
                {   $Y(q,j) \leftarrow y,\ score(q,j) \leftarrow s,\ pred(q,j) \leftarrow (p,i)$
                    insert$((q,i), \mathbf{Q})$
                }
                **else if** $s > score(q,j)$ **then**
                    $Y(q,j) \leftarrow y,\ score(q,j) \leftarrow s,\ pred(q,j) \leftarrow (p,i)$
                **else if** $s = score(q,j)$ and $y < Y(q,j)$ **then**
                    $Y(q,j) \leftarrow y,\ pred(q,j) \leftarrow (p,i)$
            }
    }
```

Figure 10: Algorithm ApproxOptXPaths

Altschul, S., Gish, W., Miller, W., Myers, E., and Lipman, D. 1990. A basic local alignment search tool. *J. Mol. Biol.* 215, 403–410.

Altschul, S., Madden, T., Schäffer, A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. 1997. Gapped BLAST and PSI-BLAST – a new generation of protein database search programs. *Nucleic Acids Research* 25, 3389–3402

Chao, K.-M., Pearson, W., and Miller, W. 1992. Aligning two sequences within a specified diagonal band. *CABIOS* 8, 481–487.

Chao, K.-M., Hardison, R.C., and Miller, W. 1993. Constrained sequence alignment. *Bull. Math. Biol.* 55, 503–524.

Chao, K.-M., Hardison, R.C., and Miller, W. 1994. Recent developments in linear-space alignment methods: a survey. *J. Comput. Biol.* 1, 271–291.

Chao, K.-M., Zhang, J., Ostell, J. and Miller, W. 1997. A tool for aligning very similar DNA sequences. *CABIOS* 13, 75–80.

Gotoh, O. 1982. An improved algorithm for matching biological sequences. *J. Mol. Biol.* 162, 705–708.

Johnson, D. 1982. A priority queue in which initialization and queue operations take $O(\log\log D)$ time. *Math. Systems Theory* 15, 295–309.

Needleman, S.B., and Wunsch, C.D. 1970. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *J. Mol. Biol.* 48, 443–453.

Myers, E., and Miller, W. 1989. Approximate matching of regular expressions. *Bull. Math. Biol.* 51, 5–37.

Spouge, J.L. 1991. Fast optimal alignment. *CABIOS* 7, 1–7.

Zhang, Z., Pearson, W., and Miller, W. 1997. Aligning a DNA sequence with a protein sequence. *J. Comput. Biol.* 4, 339–349.