JOURNAL OF COMPUTATIONAL BIOLOGY
Volume 5, Number 4, 1998
Mary Ann Liebert, Inc.
Pp. 681–702

Assembling Genes from Predicted Exons in Linear Time with Dynamic Programming

RODERIC GUIGÓ

ABSTRACT

In a number of programs for gene structure prediction in higher eukaryotic genomic sequences, exon prediction is decoupled from gene assembly: a large pool of candidate exons is predicted and scored from features located in the query DNA sequence, and candidate genes are assembled from such a pool as sequences of nonoverlapping frame-compatible exons. Genes are scored as a function of the scores of the assembled exons, and the highest scoring candidate gene is assumed to be the most likely gene encoded by the query DNA sequence. Considering additive gene scoring functions, currently available algorithms to determine such a highest scoring candidate gene run in time proportional to the square of the number of predicted exons. Here, we present an algorithm whose running time grows only linearly with the size of the set of predicted exons. Polynomial algorithms rely on the fact that, while scanning the set of predicted exons, the highest scoring gene ending in a given exon can be obtained by appending the exon to the highest scoring among the highest scoring genes ending at each compatible preceding exon. The algorithm here relies on the simple fact that such highest scoring gene can be stored and updated. This requires scanning the set of predicted exons simultaneously by increasing acceptor and donor position. On the other hand, the algorithm described here does not assume an underlying gene structure model. Indeed, the definition of valid gene structures is externally defined in the so-called Gene Model. The Gene Model specifies simply which gene features are allowed immediately upstream which other gene features in valid gene structures. This allows for great flexibility in formulating the gene identification problem. In particular it allows for multiple-gene two-strand predictions and for considering gene features other than coding exons (such as promoter elements) in valid gene structures.

Key words: dynamic programming, gene assembly, gene prediction.

1. INTRODUCTION

COMPUTATIONAL GENE IDENTIFICATION has became a very active field of research. The development of the Human Genome Project and other genome projects into the large-scale sequencing phase has motivated the need for reliable computational tools to locate functionally relevant regions—protein coding genes, in particular—in very large, uncharacterized, automatically obtained genomic sequences.

Informàtica Mèdica, Institut Municipal d'Investigació Mèdica (IMIM), and Departament d'Estadística, Universitat de Barcelona, Barcelona, Spain.

The Gene Identification Problem can be formulated as the problem of deducing the amino acid sequences encoded in a given DNA genomic sequence. In the human genome—as well as in the genome of other higher eukaryotic organisms—gene identification is compounded by the fact that genes are neither contiguous nor continuous. First, genes coding for different proteins are separated by large intergenic regions that do not code for proteins. Second, a given protein sequence is not usually specified by a continuous DNA sequence, but genes are often split in a number (maybe large) of (small) coding fragments known as Exons, separated by (larger) noncoding intervening fragments known as Introns. Intronic and intergenic DNA makes most of the human genome, and only a very small fraction of the DNA, maybe as low as 2%, corresponds to protein coding exons. Although signals exist on the DNA sequence that define gene and exon boundaries along the pathway from DNA to protein sequences—essentially, transcription promoter elements, splicing sites, and translation start and stop codons—our knowledge of the way such signals are recognized and processed by the cellular machinery is very limited. Despite the development during the past few years of increasingly complex computational techniques in order to define and locate sequence signals involved in gene specification (for a review, see Gelfand, 1995), it is usually impossible to infer the genes encoded in a given DNA sequence by relying only on them. Current programs for gene identification and gene structure prediction, developed since the early nineties [see Milanesi *et al.*, 1994; Fickett and Guigó, 1996; Fickett, 1996; Guigó, 1997; Claverie, 1997 for complementary reviews; see the World Wide Web (WWW) document maintained by Wentian Li at `http://linkage.rockefeller.edu/wli/gene` for an up-to-date list of references], therefore, do not rely only on DNA sequence signals, but, in addition, on sequence statistics indicative of protein coding function (Coding Statistics; for a review, see Fickett and Tung, 1992) and on similarity to known coding sequences. Even when integrating all this information, the performance of the current generation of gene identification programs is only moderate (Burset and Guigó, 1996), although, recently, powerful new programs based on Hidden Markov Models have shown substantially increased accuracy (Kulp *et al.*, 1996; Burge and Karlin, 1997).

1.1. Exon-based gene structure prediction programs

A broad class of current gene structure prediction programs are characterized by a similar approach, which could be described as *exon based*. In such programs, sequence signals—start codons, stop codons, donor sites and acceptor sites—are initially identified and scored along the query DNA sequence. In a second step, all potential exons defined by such signals are built. The scores of a number of coding statistics are computed on each predicted exon. In some programs, predicted exons scoring below a preestablished threshold for any of the coding statistics are discarded. Remaining exons are scored as a function of the scores of the defining signals and of the coding statistics computed (and, eventually of the degree of similarity to known coding sequences). In the final step, nonoverlapping sets of nonoverlapping exons are assembled maximizing an score defined as a function of the scores of the assembled exons. Such sets of nonoverlapping exons are assumed to be the genes predicted in the DNA sequence. The concatenation of the sequences of the exons in these sets can be directly translated into amino acid sequences. In the practice, most currently available gene structure prediction programs assume that the query DNA sequence encodes only a single complete gene—implying that the (difficult) problem of locating gene boundaries is obviated—and therefore a single gene product is assembled along the DNA query sequence. Among others, the programs GeneID (Guigó *et al.*, 1992), GAP3 (Xu *et al.*, 1994), and FGENEH (Solovyev *et al.*, 1994) are characterized by such an approach. The methods used to locate and score potential sequence signals, the coding statistics considered, and the way of combining signal and statistic scores into exon scores are particular to each program, but common to all them is the generation of a large pool of candidate exons, where the exons are selected from to be assembled into candidate genes (Figure 1). Then, given a gene scoring function—usually a function of the scores of the exons constituting the gene—the problem that these programs face—and the problem, a generalization of which we address in this paper—is to find the highest scoring gene among all those genes that can be assembled from the pool of candidate exons.

1.1.1. Exon-based gene construction. To fully understand how the problem is stated, some knowledge is required on the details of exon and gene construction. In the practice, constrains are imposed during exon and gene construction in order to guarantee that the assembled nucleotide sequence translates to a biologically meaningful protein sequence, essentially meaning that the nucleotide sequences starts with an initiation codon, and it does not translate stop codons, other than the last one. In the practice, this requires assigning to each exon a reading frame (and, eventually, a remainder). Here, we define the frame of an exon relative to the

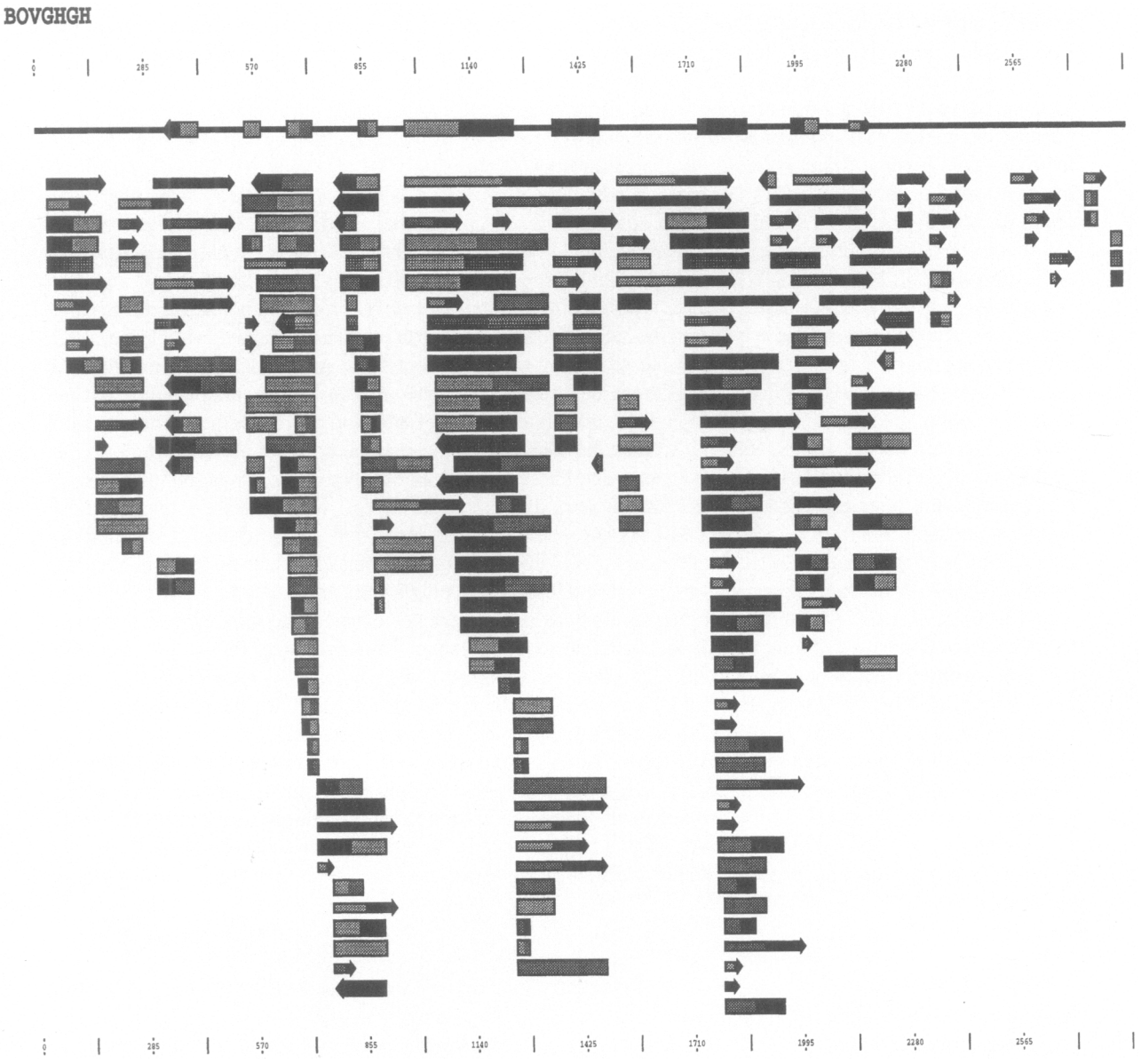

FIG. 1. The problem addressed in this paper can be visualized graphically. Let's assume a number of boxes of varying width, with boundaries given along an x-axis. The boxes are splitted in two identical sides, and each side is assigned one among three colors (which may be the same in both sides). The problem is assembling the sequence of nonoverlapping, color compatible boxes (i.e., the color of the contiguous sides of two adjacent exons matches) maximizing the sum of the widths of the assembled boxes. In addition, boxes are of differente types, and there maybe restrictions to the types to which adjancent exons in the assembly can belong. In this case, boxes are of three different types: boxes pointing left, boxes pointing right, and plain boxes. Boxes pointing left are only allowed at the begining of the assembly, while plain boxes and boxes pointing right are allowed only after plain boxes or boxes pointing left. The legal assembly maximizing the sum of the width of the boxes is displayed along the segment at the top of the figure. In this representation, the x-axis denotes a DNA sequence, and the boxes, exons predicted along the sequence (boxes pointing left corresponding to initial exons, boxes pointing right to terminal exons, and plain boxes to internal exons.) The width of the boxes is proportional to the exon score, and the box two colors denote the exon frame and remainder (integers in $\{0, 1, 2\}$). Legal assemblies of boxes denote potential genes. In this case, only single genes in one strand are considered legal gene assemblies. In the figure, the boxes correspond to exons predicted along the 2856 bp of the DNA sequence containing the bovine growth hormone gene (locus name BOVGHGH in the GenBank database, rel. 93). Exons were predicted and scored as described in Section 4. A total of 96 initial exons, 657 internal exons, and 460 terminal exons were initially predicted, but only the 25% top scoring ones were kept to be displayed in the figure, and to assemble the gene maximizing the sum of scores. Exons are displayed minimizing the number of lines needed to guarantee that two exons in the same line do not overlap.

exon sequence, rather than relative to the primary DNA sequence. Indeed, we define the **frame** of an exon as the number of nucleotides (0, 1 or 2) from the first nucleotide in the exon to the first nucleotide in the first complete codon translated from the exon when assembled into a gene. (Frame 0 means thus that the first nucleotide of the exon occurs in the first position of a codon, frame 1 means that the second nucleotide of the exon occurs in the first position of a codon, and frame 2 that the third nucleotide of the exon occurs in the first position of a codon.) The number of nucleotides left (0, 1, or 2) after the last complete codon has been defined from the exon given a frame is known as the **remainder** of the exon. The remainder r is fully determined by the length l of the exon, and the frame f in which the exon is read, $r = (l - f) \bmod 3$. Alternatively, the remainder can be defined as the complement modulus three of this number, $r = (3 - ((l - f) \bmod 3)) \bmod 3$, and this is the definition that we will use here. Defined in this way, two ordered exons are **frame compatible** if the remainder of the first exon is equal to the frame of the second one.

Typically, during exon construction, three types of exons are built: *initial Exons*, sequence segments starting with an initial codon and ending in a donor site, and not containing stop codons in the frame in which the initial codon occurs; *internal Exons*, sequence segments starting with an acceptor site and ending in a donor site, and not containing stop codons in at least one of the three frames; and *terminal Exons*, sequence segments starting with an acceptor site and ending in an stop codon, and not containing other stop codons in the frame in which the defining stop codon occurs. Reading frames are assigned to the predicted exons. By construction, the reading frame of initial exons is obviously zero, and the remainder, in consequence, $(3 - (l \bmod 3)) \bmod 3$. Analogously, the remainder of terminal exons is zero, and therefore the frame is $l \bmod 3$. Internal exons, on the other hand, may have a number of possible reading frames (one, two, or the three), depending on the distribution of the stop codons along the exon sequence. To record during exon construction the reading frames possible for internal exons (so that the information is available during gene assembly), one possibility is to construct as many copies of each internal exon as reading frames exist for such an exon, and assign to each of the copies a different one of the available reading frames. This encoding allows, in particular, in-frame scoring of the exons (Wu, 1996). Therefore, irrespective of their type, every exon is characterized by its boundaries (a pair of integers), and by its reading frame (an integer in $\{0, 1, 2\}$), the remainder—also an integer in $\{0, 1, 2\}$—being recovered from the boundaries and the reading frame.

Genes are, then, constructed as non-overlapping arrangements of an initial exon, followed by any number—maybe zero—of internal exons, and ending in a terminal exon, such that the remainder of an exon is equal to the frame of the next one in the arrangement. (Actually, an additional condition needs to be met, that stop codons are not formed at the junction of two adjacent exons. In order to simplify the notation, we will ignore in this paper such a constrain. Generalization of the results obtained here to include it, however, is straightforward.)

1.1.2. Algorithms for exon-based gene construction. In summary, exon-based gene structure prediction programs generate a large pool of predicted exons with associated scores where genes can be assembled from. Genes are scored as a function of the scores of the assembled exons, and the problem is to find the highest scoring gene that can be assembled from the set of predicted exons. Note that, in the worst case, for instance in the case in which all predicted internal exons do not overlap and they all have the same reading frame and remainder, the number of potential genes grows exponentially with the number of predicted internal exons. For a discussion on the size of the search space and the constrains that may influence it, see Wu (1996). It is obviously beneficial for exon-based gene structure prediction programs to incorporate algorithms assembling genes in less than exponential time; in particular, given that as a result of the genome projects, increasingly large genomic sequences are being assembled and submitted for analysis. The possibility of such algorithms, however, will depend on the gene scoring function considered. In Guigó *et al.* (1992), a rather complicated gene scoring function is used. Genes are given two scores: the average of the two highest scoring assembled exons, which is used as the primary ranking score, and the sum of the scores of the assembled exons subtracting 0.4 times the number of assembled exons, which is used as secondary ranking score. Although the authors introduce the concept of exon equivalence, which allows to substantially reduce the size of the set of predicted exons while guaranteeing that the highest scoring gene will still be assembled, no algorithm more efficient than exhaustive search is used. Therefore, in the worst case, the running time of such a search grows exponentially with the number of (surviving) predicted exons, and the search is impossible without a number of heuristic constraints. Thus, only the 10 top-ranking predicted initial exons, the 35 top-ranking predicted internal exons, and the 15 top-ranking predicted terminal exons are considered for assembly into a genes, and only genes containing up to 15 internal exons are effectively assembled. Even under such constraints, and running on

rather short genomic sequences—shorter than 15,000 bp—GeneID may construct in a typical case hundreds of thousands of candidate genes in order to determine the highest scoring one.

In this regard, the work of Xu *et al.* (1994), Milanesi *et al.* (1993), and Solovyev *et al.* (1994) represented progress. Assuming a simple additive gene score function (the score of a gene is the sum of the scores of the constituting exons), all these authors propose algorithms whose running time grows as the square of the number of predicted exons, given the exons sorted by increasing acceptor position—which is, by construction, their natural ordination. Xu *et al.* algorithm relies on the fact that the highest scoring gene ending in a given exon can be obtained by appending the exon to the highest scoring among the highest scoring genes ending at each nonoverlapping and frame compatible preceding exon. Strictly speaking, Xu *et al.* address a problem slightly different than the problem addressed here; in Xu *et al.*, predicted exons are clustered, and genes are assembled by selecting at most one exon from each cluster. However, if the number of clusters equals the number of exons—that is, if exons are not clustered—the case addressed by Xu *et al.* reduces to the case addressed here. The running time of Xu *et al.* algorithm is proportional to the product of the number of exons and the number of clusters, that is, proportional to the square of the number of exons, if the exons are not clustered. Certainly, under the assumption that clusters are linearly separable, Xu *et al.* develop a more efficient algorithm, whose running time is proportional to the number of exons plus the number of clusters, if the number of exclusions of clusters into the final gene can be limited to a constant, but this specialized algorithm is not applicable in general. In Milanesi *et al.* (1993) and Solovyev *et al.* (1994), on the other hand, predicted exons are assumed to be the vertex of a directed acyclic graph, and the compatibility relationships between them, the edges of the graph. A modification of the Bellman algorithm to search for the shortest path in such graphs is then used to find the highest scoring gene. Although the running time of the algorithm is proportional to the number of edges plus the number of vertex of the graph, the time required to generate the compatibility edges from the set of predicted exons is proportional to the square of the number of predicted exons, and the algorithm is overall quadratic with the number of predicted exons, as Xu *et al.* algorithm is.

1.1.3. A linear algorithm for multiple-gene two-strand exon-based gene assembly. In this paper, I show that, for additive gene scoring functions, an algorithm exists for the problem of assembling genes from predicted exons whose running time grows linearly with the size of the set of input exons. As Xu *et al.* algorithm, the algorithm developed here relies on the fact that the highest scoring gene ending in a given exon can be obtained by appending the exon to the highest scoring among the highest scoring genes ending at each nonoverlapping and frame compatible preceding exon, but takes advantage, in addition, of the fact that such a highest scoring gene can be stored and updated while scanning the set of predicted exons by increasing acceptor position. This requires an additional ordination of the set of predicted exons by increasing donor position. Given the nature of the exon data, this additional ordination can also be obtained in linear time.

Actually, the algorithm implemented here addresses not only the problem of assembling single gene structures in one strand from predicted exons, but also the general problem of assembling multiple gene structures simultaneously in the two strands of the genomic sequence. Indeed, gene elements predicted along the genomic sequence from which the optimal gene structure is obtained, need not to reduce to coding exons, and they can include intronless genes, promoter elements and gene termination elements. The valid relationships between these elements in legal gene structures are not encoded within the algorithm, but they are externally defined in a Gene Model. This allows for great flexibility formulating specifically the gene assembly problem.

In what follows, I first state formally the problem of assembling optimal gene structures from a set of predicted gene elements. Next, I describe a dynamic programming algorithm to solve the problem in linear time with the size of the set of predicted gene elements. Then, I describe a program implementing this algorithm. In particular, I describe the implementation of a generalization that can be useful in the case of multiple gene structures, when evidence exists that different predicted exons belong to the same gene. The program, then, can be forced not to assemble these exons into different genes. Next, I study the efficiency of the program in the practice. Finally, I discuss a number of applications for the program.

2. THE PROBLEM

2.1. Gene features

Let $\mathcal{F} = \{f_1, \ldots, f_k\}$ be the set of gene features to be considered. These can include first, internal and terminal exons, genes without introns, promoter elements, transcription termination signals, or others. In the

implementation presented here (see Section 4), features in different strands are considered different, that is, an internal exon in the forward strand is a different feature that an internal exon in the complementary strand.

2.2. Gene elements

A gene element e is a tuple $e = (e^1, e^2, e^3, e^4, e^5, e^6)$, such that

1. $0 < e^1 \leq e^2$, positive integers, are the boundaries of the gene element. e^1 is the **acceptor** of e, and e^2 is the **donor** of e. We will also refer to e^1 as the initial position or the beginning of the element e, and to e^2 as the terminal position or the end of exon e.
2. $e^3 \in \mathcal{F}$ is the **feature type** of e.
3. $e^4, e^5 \in \{0, 1, 2\}$ are the **frame** and **remainder** of the gene element e. Strictly speaking e^5 is unneeded: it is computed from e^1, e^2, and e^4 as $e^5 = (3 - ((e^2 - e^1 - e^4 + 1) \bmod 3)) \bmod 3$. We have chosen here to include it in the definition of gene elements to simplify notation when describing the algorithm.
4. e^6, a real value, is the **score** of the gene element e.

2.3. Gene assemblies

Let E be a set of gene elements of types in $\mathcal{F}$, a gene assembly g is a sequence $e_1, \ldots, e_n$ of elements from E ($e_i \in E$).

The **score** of the gene assembly g is $S(g) = \sum_{i=1}^{n} e_i^6$.

Given a gene assembly $g = \langle e_1, \ldots, e_n \rangle$, we define $ini(g) = \langle e_1, \ldots, e_{n-1} \rangle$. Obviously, $g = ini(g) \oplus \langle e_n \rangle$, where $\oplus$ denotes the concatenation of sequences.

2.4. Gene assembly constraints

Let $\mathcal{F}$ be a set of gene features, a gene assembly constraint is a tuple $c = (c^1, c^2, c^3, c^4)$, such that $c^1, c^2 \in \mathcal{F}$, and $0 < c^3 \leq c^4$ are positive integers.

We will use gene assembly constraints to express legal gene assemblies (see below). The constraint (c_1, c_2, c_3, c_4) indicates that a gene element of feature type c_2 is allowed to appear immediately downstream a gene element of feature type c_1 at a minimum distance of c_3 (nucleotides) and a maximum distance of c_4 (nucleotides). We will refer to the pair (c_3, c_4) as the interval distance of the constraint. For instance, let $f \in \mathcal{F}$ be the feature type representing first exons, and $i \in \mathcal{F}$, the feature representing internal exons, then the constraint $(i, f, 60, 10{,}000)$ indicates that in a legal gene assembly, an internal exon is allowed (but may not be necessary) immediately downstream a first exon at a distance between 60 and 10,000 nucleotides (which is within the typical length of eukaryotic introns).

2.5. Gene models

Let $\mathcal{M}$ be a set of gene constraints built on a set of gene features $\mathcal{F}$. $\mathcal{M}$ is a gene model if and only if for all $x, y \in \mathcal{M}$, if $x_1 = x_2$ and $y_1 = y_2$, then $x_3 = y_3$ and $x_4 = y_4$.

This condition implies that gene constraints do not contradict each other, that they are compatible. Obviously, given $x \neq y \in \mathcal{M}$, it is possible to have $x_1 = y_1$, but $x_2 \neq y_2$. For instance, if $f \in \mathcal{F}$ is the feature representing first exons, $i \in \mathcal{F}$, the feature representing internal exons, and $t \in \mathcal{F}$, the feature representing terminal exons, both (f, i, l_1, l_2) and (f, t, l_1, l_2) are compatible, and indicate that both, internal and terminal exons, are allowed to occur downstream a first exon. Conversely, we may have in $\mathcal{M}$, $x_2 = y_2$, but $x_1 \neq y_1$. In such a case, we may have $x_3 = y_3$ and $x_4 = y_4$, such as in (i, i, l_1, l_2) (f, i, l_1, l_2), indicating that internal exons can appear downstream internal exons or downstream initial exons at the same interval distance, l_1, l_2—the intron length. Or we may have $x_3 \neq y_3$ and/or $x_4 \neq y_4$, such as in (t, f, l_1', l_2') and (p, f, l_1'', l_2''), where $p \in \mathcal{F}$, represents promoters. These constraints indicates that first exons can appear immediately upstream of a terminal exon (from the preceding gene) or of a promoter element, being the interval distances different in the two cases: l_1', l_2' specifies the intergenic distance (distance between consecutive genes), and l_1'', l_2'' specifies the distance between a gene first (coding) exon and the gene promoter.

In the implementation discussed here (see Section 4), we will assume that, by default, gene elements of any feature type can initiate or terminate a gene assembly.

2.6. *Legal gene structures*

A gene assembly $g = \langle e_1, \ldots, e_n \rangle$ of gene elements on E **satisfies** gene model $\mathcal{M}$ if and only if for all e_i, $i = 1, \ldots, n-1$, there is an interval distance, (l_1, l_2) such that $(e_i^3, e_{i+1}^3, l_1, l_2) \in \mathcal{M}$ and $l_1 \leq e_{i+1}^1 - e_i^2 < l_2$.

A gene assembly $g = \langle e_1, \ldots, e_n \rangle$ satisfying $\mathcal{M}$ is a legal gene assembly (given $\mathcal{M}$) or, simply, a **gene structure**, if and only if for all e_i, $i = 1, \ldots, n-1$, $e_i^5 = e_{i+1}^4$. This condition implies that adjacent gene elements in the gene assembly are frame compatible.

2.7. *The problem*

Given a set of gene elements E of feature types in $\mathcal{F}$, and a gene model $\mathcal{M}$, the problem is to find the gene g maximizing $S(g)$. That is, to find the gene g such that for all other genes g', $S(g) \geq S(g')$.

In Figure 1, the problem for the case of single gene prediction is represented graphically. Here, gene elements are only of three different types, initial, internal and terminal exons, and the constraints in the gene model specify only that internal and terminal exons are allowed to appear immediately downstream initial and internal exons. The following set of constraints could define such a model, $\mathcal{M} = \{(f, i, 0, \infty), (i, i, 0, \infty), (f, t, 0, \infty), (i, t, 0, \infty)\}$. Note that such a model does not impose any restriction at the distance between consecutive exons, but only guarantees that they do not overlap. Note also that in such a model, not only complete genes (starting with an initial exon, and ending with a terminal exon) are legal, but also partial genes lacking initial and/or terminal exons. In the figure, exons are represented as boxes along the x-axis. Initial exons are represented by arrow-headed boxes pointing left, terminal exons by arrow-headed boxes pointing right, and internal exons by plain boxes. The width of the boxes denotes the score of the exons. A three-color code is used to represent the frame and the remainder of the exons. Each exon box is vertically split in two identical sides, and each side is assigned a color. The color on the left side depends on the frame of the exon, and the color on the right side on the remainder. The problem is then to pick up the sequence of nonoverlapping boxes maximizing the sum of widths, and verifying that the only box pointing to the left (if any) is the first, the only box pointing to the right (if any) is the last, any other boxes are plain, and the boxes in the sequence are color-compatible—that is, the color of the left side on one box matches the color of the right side of the next box in the sequence.

For the case of single gene one strand prediction, Solovyev *et al.* (1994) state the problem in the context of graph theory. The set of exons E can be considered the set of vertex of a graph G, while the set of arcs from each exon to each downstream frame compatible exon, the set of edges C of the graph. In addition, arcs are assumed in C from a *source* to each initial exon, and from each terminal exon to a *sink*. $G = \langle E, C \rangle$ is an acyclic directed oriented segmented graph (Gelfand and Roytberg, 1993). A path in such a graph that starts in a source is called *initial*, and an initial path ending in a sink is called *full*. If we assume that the weight of an edge in G is the score of the exon (vertex) which is entered by this edge (assuming weight zero for the edges entering the sink), then the problem of finding the highest scoring gene definable in E is the problem of finding the optimal full path in G. Although this is the approach taken by Solovyev *et al.* (1994), in the practice the set of edges C is usually not given, and it needs to be explicitly computed from the set of predicted exons E, requiring a time that grows as the square of the size of E. I will show, however, that for the problem addressed here, the explicit generation of the set C is in the practice unneeded, and that, by avoiding it, a more efficient algorithm can be designed.

3. A SOLUTION

Let G be the set of all gene structures definable in E given $\mathcal{M}$. The problem is to find a highest scoring gene structure g in such a set. We will denote one such g by g^*, and we will refer to g^* as to a **best gene structure** in E. Let g_i be a best—highest scoring—gene structure ending in gene element e_i, and let $G^* = \{g_1, \ldots, g_n\}$, the set of highest scoring genes ending in each of the gene elements in E. Obviously, a best gene structure in E is a highest scoring gene structure in G^*, and therefore g^* belongs to G^*. In this section, we show that there is an algorithm to generate G^* (and therefore to find g^*) in time linear with the size of E. First, we describe an algorithm to solve a simplified version of the problem, in which only one type of gene element is considered and frame constraints are ignored. To introduce the algorithm, we compare it with the quadratic algorithms used in current systems. Next, we generalize the algorithm to take into account frame constraints. Finally, we further generalize the algorithm to assemble genes from gene elements of any number of feature types given an arbitrary gene model. In particular, the algorithm can be applied to assemble multiple genes

in the two orientations along large genomic sequences on which potential gene elements of different types (exons, promoters, etc.) have been identified and scored.

3.1. A simplified version of the problem

In this simplified version of the problem, we will consider only one type of gene element, the **exon**, and we will ignore frame constrains. An exon is thus, a tuple, $e = (e^1, e^2, e^6)$, such that $e^1 < e^2$, positive integers, are the boundaries of the exon, and e^6, a real number, is the score of the exon. The only constraint in the gene model is $(e, e, 0, \infty)$ (using e also to denote the feature type of the exons). A gene structure (or simply a gene, in this case) is then a (finite) sequence of exons such that the donor of an exon is smaller than the acceptor of the next exon in the sequence. Let $E = \{e_1, \ldots, e_N\}$ be a set of exons, and $G^* = \{g_1, \ldots, g_N\}$, the set of highest scoring genes ending in $e_1, \ldots, e_N$.

3.1.1. A quadratic solution. Assuming that $e_1, \ldots, e_N$ is an ordination of the E according with increasing acceptor position, the set $S(g_1), \ldots, S(g_N)$ of best gene scores can be defined recursively:

$$S(g_i) = \max\{S(g_j) : 0 < j < i, e_j^2 \leq e_i^1\} + e_i^3 \tag{1}$$

assuming that $g_0 = \langle\rangle$. Equation (1) leads to the following algorithm—written in gawk (GNU awk)—which determines the set of best genes in E, $\{g_1, \ldots, g_N\}$, and the corresponding scores.

```
BestGene1(E) {
  for (i=1;i<=N;i++) {
    S[i]=0
    G[i]=0
    for (j=1;j<=i-1;j++ {
      if (E[j,2] < E[i,1])
        if (S[j] > S[i]) {
          S[i]=S[j]
          G[i]=j
        }
    }
    S[i] = S[i] + E[i,6]
  }
}
```

where `E` is a two-dimensional array of length N, storing the acceptor (`E[i,1]`), donor (`E[i,2]`), and score (`E[i,6]`) of each exon `i`, and sorted by increasing acceptor position. `S` is an array of length N, storing the cumulative score of the best gene ending in each exon `i`. The running time of the algorithm is $O(N^2)$. The loops are nested two in deep, and each loop index (`i` and `j`) takes on at most N values. Only two values need to be stored for each exon $e_i \in E$, `S[i]` and `G[i]`; the space required for this algorithm is thus proportional to N. The best gene ending in exon e_i can be recovered by means of the following recursive algorithm:

```
PrintBestGene(i){
  if (G[i] != 0)
    PrintBestGene(G[i])
  print E[i,1], E[i,2]
}
```

The best gene definable in E, g^*, is indicated by the index `i` such that `S[i]` is maximum.

Above is essentially the recurrence at the core of Xu *et al.* (1994) gene assembly algorithm. Milanesi *et al.* (1993) and Solovyev *et al.* (1994) solve the problem by taking a somehow different approach. They build the so-called *exon compatibility graph*, *ECG*, a directed acyclic graph, whose edges are the pairs defined by each exon in E and each of its *compatible* exons. In our simplified version of the problem, the set of exons compatible with a given exon e_i, c_i, is simply the set of those exons starting after the donor of e_i, that is, $c_i = \{(e_i, e_x) : e_x \in E, e_x^1 \geq e_i^2\}$. Then, the *ECG* is defined by the set of vertex E and the set of edges $C = \bigcup_i^N c_i$. A dynamic programming algorithm to search for an optimal path in directed acyclic graphs is then

used to find the best gene in E. In the practice, Solovyev *et al.* end up also determining the set $S(g_1), \ldots, S(g_N)$ of best gene scores. They use the following algorithm:

```
BestGene2(C,s) {
  for (i=1;i<=N;i++) {
    S[i]=0; G[i]=0
  }
  for (j in C[i])
    if (S[i]+s[j]) > S[j]) {
      S[j]=S[i]+E[j,6]
      G[j]=i
    }
  }
}
```

where `C[i]` is the set of (indices of) exons compatible with e_i, and `S` and `G` have the same meaning than in Xu *et al.* algorithm. The running time of this algorithm is $O(c + N)$, were c is the total number of edges in the exon compatibility graph—that is the sum of the sizes of the sets c_i $(i = 1, \ldots, N)$. However, the time required to build the exon compatibility graph is $O(N^2)$, and thus the algorithm is also overall quadratic in time with the number of exons, as Xu *et al.* algorithm is. The space requirements of this algorithm are proportional to N^2, since an N^2 table is required to store the *ECG*.

3.1.2. A linear solution. As we have already pointed out, Xu *et al.* algorithm avoids exponential search by relying on the fact that to find the best gene ending in exon e_i, g_i, we need to check only the best genes built before g_i $(g_1, \ldots, g_{i-1})$. However, it is not necessary to check all such genes. If there are not genes ending between the acceptor of e_{i-1}, and the acceptor of e_i, g_i can be obtained by concatenating e_i to the best gene ending before the acceptor of e_{i-1}, that is to the best gene after which e_{i-1} was concatenated to obtain g_{i-1}. Such a gene is ini (g_{i-1}), and, thus, $g_i = \text{ini}(g_{i-1}) \oplus e_i$. If there are exons ending between the acceptor of e_{i-1} and the acceptor of e_i, in addition to ini (g_{i-1}), only the best genes ending between the acceptor of e_{i-1} and the acceptor of e_i need to be checked. This is expressed in the following recurrence relation:

$$S(g_i) = \max\left\{ \begin{array}{l} S(\text{ini}\,(g_{i-1})), \\ \max\{S(g_j) : j < i, e^1_{i-1} \le e^2_j < e^1_i\} \end{array} \right\} + e^6_i \tag{2}$$

This equation can be implemented in the following algorithm, assuming again that `E` is given sorted by increasing acceptor position.

```
BestGene3(E) {
  Ga=0; S[0]=0
  for (i=1;i<+N;i++) {
    for (j=1;j<i;j++)
      if (E[j,2] > E[i-1,1] && E[j,2] < E[i,1])
        if (S[j] > S[Ga])
          Ga=j
  }
  G[i]=Ga
  S[i]=S[Ga]+E[i,6]
}
```

While scanning `E` according to increasing acceptor position, the algorithm registers the best gene (`Ga`) to which each exon has been concatenated to obtain the best gene ending in such an exon. At each step `i`, `Ga` is updated by looping, using index `j`, over the set of best genes ending between the acceptors of exons `i - 1` and `i`, and comparing at each step `j`, the score of `Ga` (`S(Ga)`) with the scores of the best gene ending in exon `j` (`S[j]`). The best gene ending in exon `i` is obtained by concatenating the exon `i` to such an updated best gene `Ga` (`G[i] = Ga`). Although in order to update `Ga` at step `i`, we need to check only the best genes ending before the acceptor of exon `i - 1` and the acceptor of exon `i`, since the ordering of `E` according to increasing

acceptor position is not necessarily the same order than the order according to increasing donor position, the index `j` must loop back through all best genes preceding exon `i` in `E` to determine which of them end between the acceptor of exon `i - 1` and the acceptor of exon `i`. The running time of this algorithm quadratic with the size of E.

A linear algorithm can be obtained, however, if a second ordination of `E` according with increasing donor position exists. Looping over this second ordination to update the current best gene `Ga` is the essential fact on which relies the gene assembly algorithm described in this paper.

Indeed, let a and d sorting functions of E according to increasing acceptor and donor position respectively. That is, $e_{a(i)}$ is the exon occupying position i when E is sorted by increasing acceptor position, and $e_{d(i)}$ is the exon occupying position i when E is sorted by increasing donor position. The Equation (2) can be rewritten as

$$S(g_{a(i)}) = \max\left\{\begin{array}{l} S(\mathrm{ini}(g_{a(i-1)})), \\ \max\{S(g_{d(j)}) : e^1_{a(i-1)} \le e^2_{d(j)} < e^1_{a(i)}\} \end{array}\right\} + e^3_{a(i)} \tag{3}$$

According with this equation, the set G^* is obtained recursively according to the increasing acceptor position order, irrespectively of the initial order of E. Equation (3) can be written in a more meaningful way. Indeed, let rst be the function in E that returns, given the index of an exon e in E, the index of the exon ending the rightmost to the acceptor of e, that is, $\mathrm{rst}(i) = j$ if and only if $e^2_j < e^1_i$, and for all other $e_k \in E$, if $e^2_k \le e^1_i$ then $e^2_k \le e^2_j$, and let's assume $\mathrm{rst}(i) = 0$ if there is no exon ending upstream the acceptor of e_i. Then Equation (3) can be rewritten as

$$S(g_{a(i)}) = \max\left\{\begin{array}{l} S(\mathrm{ini}(g_{a(i-1)})), \\ \max\{S(g_{d(j)}) : d^-(\mathrm{rst}(a(i-1))) < j \le d^-(\mathrm{rst}(a(i)))\} \end{array}\right\} + e^6_{a(i)} \tag{4}$$

were a^- and d^- denote the inverses of the functions a and d. The following linear algorithm, during which execution the function rst is built implicitly, implements such a recursive equation

```
BestGene4(E,a,d) {
  Ga=0; S[0]=0
  j=1
  for (i=1;i<=N;i++) {
    while (E[d[j],2] < E[a[i],1]) {
        if (S[d[j]] > S[Ga])
          Ga=d[j]
        j++
    }
    G[a[i]]=Ga
    S[a[i]]=S[Ga]+E[a[i],6]
  }
}
```

In addition to the set of exons `E`, `a` and `d`, arrays of length `N` are also given to this algorithm, keeping the ordinations of `E` according to increasing acceptor and donor position As in `BestGene2`, the algorithm here relies on updating at each step `i` the current best gene, `Ga`, by comparing its score (`S[Ga]`) with the scores of the best genes ending between the acceptors of the exons `a[i-1]` and `a[i]`. But, contrary to `BestGene2`, since `j` is looping over the set `E` according to increasing donor position, there is no need for the index `j` to loop back through all best genes ending in exons preceding `a[i]` to locate the best genes ending between exons `a[i - 1]` and `a[i]`, but the index `j` does loop exactly only over those genes ending between exons `a[i - 1]` and `a[i]`. Indeed, let j_{i-1} be the value of j after recursion $i - 1$ on Equation (4) clearly, $e^2_{d(j_{i-1}+1)} > e^1_{a(i-1)}$ and for all $j' < j_{i-1}$, $e^2_{d(j')} \le e^1_{a(i-1)}$. Therefore, at recursion i, j only needs start taking values at j_{i-1}. On the other hand, we have that if index j_k is such that $e^2_{d(j_k)} > e^1_{a(i)}$, then for all $j' > j_k$, $e^2_{d(j')} > e^1_{a(i)}$. Therefore, if j_i is the smaller value of j for which $e^2_{d(j_i)} > e^1_{a(i)}$, j does not need to take values greater than j_i. Therefore the set of values over which j needs to loop at recursion i is bounded by j_{i-1} and j_i. At recursion $i + 1$, j will start taking values at j_i. This fact is used in the algorithm to guarantee that the index `j` never backtracks and loops at most once over each exon in `E`. `i` also loops only once over each exon in `E`, and therefore during

the execution of the algorithm each exon is accessed at most only twice. Therefore the running time of the algorithm is strictly proportional to the size of E.

Obviously, the linearity of the algorithm requires that `a` and `d`, the arrays that keep ordinations of the elements in `E` according to increasing acceptor and donor position, be given in addition to `E` itself. The extra time required to compute these arrays needs to be taken into account to compute the total running time of the algorithm. However, acceptor and donor positions are integers, and integers can always be sorted in linear time. Therefore, `a` and `d` can be obtained from `E` in time proportional to `N`, and `BestGene3` runs in linear time even if `a` and `d` are computed within. Moreover, `E` is usually given sorted by increasing acceptor position, which, in turn, is already a good preordination of `E` according with increasing donor position, and even very simple sorting algorithms run in linear time in the practice.

3.2. *Considering frame constraints*

In this section we will generalize the algorithm described in the previous section to take into account frame constraints. We still consider however, only one type of exon. An exon is, thus, a tuple, $e = (e^1, e^2, e^4, e^5, e^6)$, such that $e^1 < e^2$, positive integers, are the boundaries of the exon, e^4 is the frame, e^5 is the remainder, and e^6, a real number, is the score of the exon. Now, a gene structure (or simply a gene), in this case, is a sequence of exons such that the donor of an exon is smaller than the acceptor of the next exon in the sequence, and the remainder of an exon is equal to the frame of the next.

Only a slight modification needs to be made to generalize the algorithm described in the previous section to take into account frame constraints. Now a highest scoring gene ending in a given exon e_i is obtained either (1) by concatenating e_i to the gene after which the rightmost exon preceding e_i and having the same frame than e_i, e_f, was concatenated to obtain the best gene ending in e_f (and not simply by concatenating e_i to the gene after which the exon e_{i-1} was concatenated to obtain the best gene ending in e_{i-1} as before), or (2) by concatenating e_i to the best among the highest scoring genes with remainder equal to the frame of e_i, and ending between the acceptor of e_f and the acceptor of e_i (and not simply by concatenating e_i to best among the best genes ending between the acceptor of e_{i-1} and the acceptor of e_i).

To rewrite Equation (4), we need to introduce the function prf defined in E. Given the index i of an exon in E, this function returns the index of the exon with the rightmost acceptor to the acceptor of exon i and having the same frame than exon i. That is, $\text{prf}(i) = j$ if and only if $e_j^1 \leq e_i^1$ and $e_j^4 = e_i^4$, and for all other $e_k \in E$ such that $e_k^4 = e_i^4$, if $e_k^1 \leq e_i^1$, then $e_j^1 \leq e_k^1$. Then, the following recurrence—a generalization of Equation (4)—holds:

$$S(g_{a(i)}) = \max\left\{\begin{array}{l} S\left(\text{ini}\left(g_{\text{prf}(a(i))}\right)\right), \\ \max\left\{S\left(g_{d(j)}\right) : d^-\left(\text{rst}\left(\text{prf}(a(i))\right)\right) < j \leq d^-\left(\text{rst}\left(a\left(i\right)\right)\right) \text{ and } e_{d(j)}^5 = e_{a(i)}^4\right\} \end{array}\right\} + e_{a(i)}^6 \quad (5)$$

This recurrence can be implemented in the following algorithm

```
BestGene5(E,a,d)  {
  Ga[0]=Ga[1]=Ga[2]=0; S[0]=0
  j=1
  for (i=1;i<=N;i++) {
    fri=E[a[i],4]                    # frame of exon a[i]
    while (E[d[j],2] < E[a[i],1]) {
      rmj=E[d[j],5]                  # remainder of exon d[j]
      if (S[d[j]] > S[Ga[rmj]])
        Ga[rmj]=d[j]
      j++
    }
    G[a[i]]=Ga[fri]
    S[a[i]]=S[Ga[fri]]+E[a[i],6]
  }
}
```

where now `E` holds also the frame (`E[i,4]`) and remainder (`E[i,5]`) of each exon `i`. `BestGene5` is essentially equivalent to the algorithm `BestGene4`, but in order to guarantee frame compatibility across concatenated exons, `Ga` is not an scalar, but an array keeping a separate index for the current best gene in each of the

three possible remainders. As in BestGene4, during each iteration i, the index j loops exactly only over the best genes ending between exons a[i-1] and a[i], but now Ga is updated separately for each possible remainder (rmj). At the end of the iteration, the exon a[i] is concatenated to the best gene pointed by Ga in the remainder equal to the frame of the exon a[i]. Thus, the running time of BestGene5 growths linearly with the size of E, N; as BestGene4, the algorithm loops only at most twice over each exon.

3.3. *Full gene assemblies*

In this section, we address the full version of the gene assembly problem as stated in Section 2. Now, the gene elements to be assembled into gene structures belong to a number of different types in the feature set $\mathcal{F}$, and the constraints expressing legal assemblies are given in a gene model $\mathcal{M}$. To simplify the description of the algorithm, we will assume that the constraints in the gene model refer only to the feature types allowed between consecutive gene elements in gene structures, but we will ignore constraints regarding allowed interval distances. The generalization to include interval distances is trivial, but its description is cumbersome. (An awk code that takes into account interval distances constraints can be obtained by anonymous ftp from apolo.imim.es under /pub/GeneAssembly; see Section 4.)

The algorithm to address the full version of the problem is a generalization of the algorithm described in Section 3.2. The difference is that Ga is an array keeping a separate index for each type of gene element (and each remainder). That is, during scanning of the set of gene elements E, the index of the current best gene is kept separately for each feature type (and each remainder). Actually, a further reduction in computational time can be gained by factorizing all the feature types that according with the gene model $\mathcal{M}$, have exactly the same set of upstream compatible features. (The features upstream compatible to a given feature f are the features that can appear immediately upstream of feature f, according with the gene model, see below). A single index needs to be kept for all those gene features having exactly the same set of upstream compatible features. To introduce the recursive equations corresponding to the general version of the algorithm, we need to redefine the sorting function d—to sort separately elements of different feature types—and the functions prf and rst. In turn, to define these functions, we need to introduce the concepts of downstream compatibility and upstream equivalence.

Upstream compatibility. Given a gene feature $f \in \mathcal{F}$ and a gene model $\mathcal{M}$, we define

$$c_f = \{x \in \mathcal{F} : (x, f) \in \mathcal{M}\}$$

that is, c_f is the set of features that can appear immediately upstream of the feature f given the gene model $\mathcal{M}$, the *features upstream compatible with* f.

EXAMPLE.

Let be the set of gene features

$$\mathcal{F} = \{f, i, t, p, s\}$$

where f, i, t could stand for first, internal and terminal exons, p for promoters, and s for intronless genes. And let be the following gene model,

$$\mathcal{M} = \left\{ \begin{array}{lllll} (p,f), & (i,i), & (i,t), & (t,p), & (t,s), \\ (s,f), & (f,i), & (f,t), & (s,p), & (s,s), \\ (t,f), & & & & (p,s) \end{array} \right\}$$

which indicates than first exons and intronless genes can appear right after promoters, intronless genes or terminal exons, internal and terminal exons can appear after first or internal exons, and promoters after terminal exons or intronless genes. Then, the sets of upstream compatible features for each feature are

$$c_f = \{p, s, t\}, c_i = \{i, f\}, c_t = \{i, f\}, c_p = \{t, s\}, c_s = \{p, s, t\}$$

Downstream equivalence. Given a gene feature $f \in \mathcal{F}$ and a gene model $\mathcal{M}$, we define the set of *downstream equivalent features* to f, to be the set of features in $\mathcal{F}$ that share the same set of upstream compatible features

$$[f] = \{x \in \mathcal{F} : c_x = c_f\}$$

Obviously $f \in [f]$. Let $C_{\mathcal{F}}$ be the set of the different sets of downstream equivalent features. If $x \in C_{\mathcal{F}}$, we will use c_x to denote the set of upstream compatible features for the elements in x—which is, by definition, the same for all of them.

EXAMPLE (CONT). In the example above, the set of downstream equivalent features is,

$$\begin{aligned} [f] = [s] = \{f, s\} &\Longleftarrow c_f = c_s = \{p, s, t\} \\ [i] = [t] = \{i, t\} &\Longleftarrow c_i = c_t = \{i, f\} \\ [p] = \{p\} &\Longleftarrow c_p = \{t, s\} \end{aligned}$$

Then,

$$C_{\mathcal{F}} = \{\{f, s\}, \{i, t\}, \{p\}\}$$

Sorting functions. For each $x \in C_{\mathcal{F}}$, let's define the sorting function, d_x. $d_x(i)$ returns the index in E of the gene element occupying position i, when only the subset of gene elements of types in c_x (upstream compatible with features in x), are sorted by increasing donor position. In other words, $e_{d_x(i)}$ is the exon occupying position i, when the subset of gene elements of types in c_x are sorted by increasing donor position.

The functions preceding (prf) and rightmost (rst). Now we redefine the functions prf and rst, first introduced in Sections 3.1.2 and 3.2

prf(i) returns the index in E, j, of the rightmost exon preceding e_i having the same frame than e_i, and downstream equivalent with the feature type of e_i. That is, $\text{prf}(i) = j$, if and only if

1. $e_j^4 = e_i^4$
2. $e_j^3 \in c_{e_i^3}$
3. $e_j^1 \leq e_i^1$
4. and for all other exon e_k verifying conditions 1, 2, and 3, $e_k^1 \leq e_j^1$

rst(i) returns the index in E, j, of the exon ending the rightmost to the (acceptor of) exon e_i, and upstream compatible with the feature type of e_i. That is, $\text{rst}(i) = j$, if and only if

1. $e_j^3 \in [e_i^3]$ (in other words, $(e_j^3, e_i^3) \in \mathcal{M}$)
2. $e_j^2 \leq e_i^1$
3. and for all other exon e_k verifying conditions 1 and 2, $e_k^2 \leq e_j^2$

The algorithm. Then, the following equation defines recursively the set of scores of the best gene structures ending in each gene element of the set E, given the gene model $\mathcal{M}$

$$S(g_{a(i)}) = \max \left\{ \begin{array}{l} S\big(\text{ini}\big(g_{\text{prf}(a(i))}\big)\big), \\ \max\big\{S\big(g_{d_f(j)}\big) : d_f^-(\text{rst}(\text{prf}(a(i)))) < j \leq d_f^-(\text{rst}(a(i))) \text{ and } e_{d_f(j)}^5 = e_{a(i)}^4\big\} \end{array} \right\} + e_{a(i)}^6 \quad (6)$$

where $f = e_{a(i)}^3$, is the feature type of exon $a(i)$. This recurrence can be implemented in the following algorithm

```
BestGene6(E,D,a,d) {
  #initialization
  for  (c=1;c<=m;c++) {
    Ga[c,0]=Ga[c,1]=Ga[c,2]=0; S[0]=0
    je[c]=1
  }

  for  (i=1;i<=N;i++) {
    etype=D[E[a[i],3]]                    # downstream equivalent type
    fri=E[a[i],4]                         # frame of exon a[i]
    j=je[etype]
```

```
      while (E[d[etype,j],2] < E[a[i],1]) {
        rmj=E[d[etype,j],5]                    # remainder of exon d[etype,j]
        if (S[d[etype,j]] > S[Ga[etype,rmj]])
          Ga[etype,rmj]=d[etype,j]
        j++
      }

      je[etype]=j;
      G[a[i]]=Ga[etype,fri]
      S[a[i]]=S[Ga[fri]]+E[a[i],6]
    }
  }
```

The main difference between BestGene6 and BestGene5 is that in BestGene6, at each iteration i, in order to update the current best gene Ga, the index j loops only over the set of gene elements of feature type upstream compatible with the feature of gene element a[i]. This requires the looping index j to be an array, and to add an additional dimension to d and Ga, so that separate values can be kept for each set of features sharing the same set of upstream compatible features—that is, for each different set of downstream compatible features. As in BestGene5, the index j loops only once over each element within the set of gene elements of features upstream compatible with each set of downstream equivalent features. The running time of BestGene6 grows, thus, linearly with the size of E.

Note that BestGene6, in addition to the set of predicted gene elements E and their ordination according with increasing acceptor position, a, requires the additional set of ordinations according to increasing donor position, d, corresponding to each different set of downstream equivalent features. It also requires the array D, which indicates the class of downstream equivalent features to which each feature belongs. a and d because they sort integers can be obtained in linear time. On the other hand, D and d depend on the gene model $\mathcal{M}$. In the next section we will show how D and d can be obtained from a given gene model.

4. IMPLEMENTATION

In this section, we describe a program that implements the BestGene6 algorithm described in the previous section. The program—to which we refer here as GenAmic—has been prototyped in gawk, and can be obtained by anonymous ftp from apolo.imim.es under /pub/GeneAssembly). The input to the program consists of the specification of the Gene Model, and of the set of gene features predicted along a genomic sequence. The output is the optimal Gene Structure according with the specifications in the Gene Model. Previously to the execution of the gene assembly algorithm, GenAmic needs to process the specification of the Gene Model and the set of gene features to produce the array D, which indicates the class of downstream equivalent features to which each feature belongs, and the set d of ordinations according to increasing donor position corresponding to each different set of downstream equivalent features. Both D and d are required by BestGene6. To simplify, we will assume that the input set of gene features is given already sorted by increasing acceptor position. This is, after all, the natural ordination—the ordination of the gene features along the genomic sequence. Therefore, the program does not need to obtain the ordination a (that is, in BestGene6, a[i] can be substituted by i).

In this section, we will first describe the formats of input and output files, then we will describe the preprocessing of the input Gene Model and predicted gene features to produce the arrays D and d required by BestGene6. Next, we will describe a generalization of the algorithm to prevent otherwise legal connections between gene elements to occur in certain regions of the genomic sequence. The motivation here is to prevent that different predicted exons along the genomic sequence be assigned to different genes, when there exists evidence that they belong to the same gene. Finally, we briefly discuss the complexity of the algorithm, and of the particular implementation developed here.

4.1. Input and output files

GenAmic reads two input files (three in the most general case described below). One file specifying the Gene Model, and other file containing the set of predicted gene features.

```
Terminal+:First-          First+:Terminal-
First+:Internal+          Internal+:Terminal+
Terminal-:Internal-       Internal-:First-
```

FIG. 2. Input gene assembly constrains specifying a gene model corresponding to multiple genes in both strands.

Gene model. A Gene Model is a set of constraints. In Section 2, we have defined each constraint as a pairwise relationship between gene features. In Section 3, we have seen that gene assembly constraints can be factored: downstream equivalent features can be associated to their corresponding upstream compatible features. This allows for a very compact way of specifying a Gene Model. Indeed, ignoring for the time being interval distances, a Gene Model can be described by a set of factored constraints:

$$i : \{g_{i1}, \ldots, g_{im_i}\} \longleftarrow \{f_{i1}, \ldots, f_{in_i}\} \tag{7}$$

where $f_{ij}, g_{ik} \in \mathcal{F}$, and $i = 1, \ldots, l$. These factored constraints indicate that each feature f_{ij} is allowed to appear immediately upstream each feature g_{ik} in a legal gene structure. In other words, the features $\{f_{i1}, \ldots, f_{in_i}\}$ are downstream equivalent to the upstream compatible features $\{g_{i1}, \ldots, g_{im_i}\}$. The factored constraints i expands to the explicit gene model

$$\mathcal{M} = \{(g_{ik}, f_{ij}) : k = 1, \ldots, m;\ j = 1, \ldots, k;\ i = 1, \ldots, l\}$$

In the implementation developed here, the Gene Model is indeed specified a set of factored constraints, and `GenAmic` expects a two column file, as shown in Figure 2. Each record in this file corresponds to a set of factored constraints, gene features (separated by colons) appearing in the first column are allowed in a legal gene structure, immediately upstream of the features (separated by colons) in the second column. For instance, in the Gene Model of Figure 2, first exons in the forward strand (`First+`), and terminal exons in the reverse strand (`Terminal-`) are allowed to appear immediately upstream of either terminal exons in the forward strand (`Terminal+`) or first exons in the reverse strand (`First-`). The Gene Model in Figure 2 specifies multiple gene structures in both strands of the sequence. A model specifying single gene structures in one strand would consist only of the following entry "`Internal:Terminal First:Internal`".

The program does not require a particular keyword to be tied to a particular gene feature, rather the election of keywords denoting gene features is left to the user. However, keywords must be used consistently in the Gene Model and in the set of predicted gene features. That is, the same keyword must specify the same gene feature in both files. On the other hand, we have chosen here to explicitly include the strand in which a given feature occurs, in the description of the feature. This is, of course, unneeded, and features and strand could be associated inside the program. However, there is generally no advantage in specifying separately feature and strand, while including the strand in the definition of the feature makes specification and parsing of the Gene Model easier.

Set of predicted gene features. The input set of predicted gene features, E, is a file in GFF format. GFF (Gene-Finding Format) is a proposed exchange format for gene-finding features (Durbin and Haussler, 1997). Figure 3 shows an example of a GFF file (actually pseudo-GFF) that could be used in conjunction with the Gene Model specified in Figure 2. Each record in the file corresponds to a predicted gene element. The first column is the feature type of the gene element, the second and third columns are the boundaries of the gene element (acceptor and donor, as we have call them here), the fourth column is the score, and the fifth and sixth are the frame and the remainder of the gene element. As we have pointed out, the keyword used to identify a given feature in the first columns must be the same that the keyword used to identify the same feature in the Gene Model file.

Note that in the approach taking here, every gene feature must have assigned a frame and a remainder. However, frame and remainder make probably sense only when applied to coding exons. They make little sense when applied to other gene elements, such a promoter or transcription termination elements. Since, by construction (see Section 2), complete assembled genes have frame 0 (first exons have always frame 0) and remainder 0 (terminal exons have always remainder 0), intergenic elements such as promoter or gene termination elements must have, in the current implementation, frame 0 and remainder 0. For intragenic nonframe elements (such as intronic regulatory elements), the only possibility is two replicate the elements nine times assigning to each replicate a different combination of frame and remainder. It should be not too

```
Internal+  386  440  -1.910  2  1
Terminal+  386  430  -1.753  0  0
Terminal+  386  413   1.183  1  0
Terminal+  386  444  -0.178  2  0
First+     388  415  -4.593  0  2
First+     388  440  -3.944  0  1
Internal-  402  562  -5.411  0  2
Internal-  402  554  -4.707  0  0
Internal-  402  524  -3.349  0  0
Internal-  402  498   2.524  0  1
Internal-  402  438  -5.850  2  0
Internal-  402  438  -5.660  1  2
Internal+  411  440  -5.723  1  1
Terminal+  411  444   4.143  1  0
Internal+  414  440  -5.766  0  0
Terminal+  414  430  -4.478  2  0
Terminal-  451  558  -2.212  0  0
Terminal-  451  554  -0.117  0  2
Terminal-  451  524   0.376  0  2
```

FIG. 3. Fragment of an input file of predicted gene elements in pseudo-GFF format. The complete file consists of 2153 records. The predicted gene structure in Figure 4 has been obtained from this file, according with the model in Figure 2.

```
Terminal-    451    495  0.854  0  0   0.854
First-       856    936  1.550  0  0   2.404
Terminal-   6762   6805  3.215  0  2   5.619
Internal-   8201   8349  2.577  2  1   8.196
First-     13415  13479  1.239  1  0   9.435
First+     22226  22303  0.552  0  0   9.987
Internal+  23965  24007  2.807  0  2  12.794
Internal+  27320  27355  2.839  2  2  15.633
Internal+  27484  27576  5.330  2  2  20.963
Terminal+  30723  30748  0.815  2  0  21.778
```

```
# Optimal Gene Structure. 3 genes. Score =
21.778

# Gene 1. 2 exons. Reverse. Score =  2.404
 1.1   451   495 - Terminal 0 0 0.854
 1.2   856   936 - First    0 0 1.550

# Gene 2. 3 exons. Reverse. Score =  7.031
 2.1  6762  6805 - Terminal 0 2 3.215
 2.2  8201  8349 - Internal 2 1 2.577
 2.3 13415 13479 - First    1 0 1.239

# Gene 3. 5 exons. Forward. Score = 12.343
 3.1 22226 22303 + First    0 0 0.552
 3.2 23965 24007 + Internal 0 2 2.807
 3.3 27320 27355 + Internal 2 2 2.839
 3.4 27484 27576 + Internal 2 2 5.330
 3.5 30723 30748 + Terminal 2 0 0.815
```

FIG. 4. Output of `GenAmic`. The gene structure predicted here has been obtained from GFF file in Figure 3 and the gene model in Figure 2. Top, raw output of `GenAmic`. Bottom, postprocessing of `GenAmic` output to produce a more meaningful output.

difficult, however, to change the algorithm to allow for frame constraints not to be enforced for some gene features.

Output optimal gene structure. The output of `GenAmic` is an optimal gene structure. This is also produced in pseudo-GFF format (Figure 4). One additional column holds the cumulative score of the Gene Structure In this particular case, the optimal gene structure consists of three different genes, the first gene has two exons

and it occurs in the reverse strand, the second gene has tree exons and occurs also in the reverse strand, and the third gene has five exons and occurs in the forward strand. The score of the Gene Structure is the cumulative score of the last feature of the structure, 21.778 in this case. A simple filter can be written that produces a more meaningful output. In Figure 4, we show the results of applying one such filter. The filter reads the set of those features which start a new gene (`Terminal-` and `First+`, in the gene model above), in addition to the optimal gene structure.

4.2. Preprocessing of the gene model

A preprocessing of the Gene Model is required in order to obtain the function `D`, which indicates the class of downstream equivalent features to which each feature belongs, and the function `U`, which indicates the different classes of downstream equivalent features to which each upstream compatible feature is associated. `U` is required in order to obtain the set of functions `d`, which sort separately those features upstream compatible with each set of downstream equivalent features.

By construction, each factored constraint corresponds to a set of downstream equivalent features. Therefore from Equation (7), we can define the function D as

$$D(f_{ij}) = i \tag{8}$$

Similarly, the function U is defined as

$$U(g_{ij}, i) = i \tag{9}$$

Note that the same feature can be upstream compatible to more than one set of downstream equivalent features, that is, it can appear in different factored constraints. Therefore, the function U requires two arguments: the feature and the constraint in which the feature occurs.

`U` is used to build the sorting functions `d`. A single sorting function `d` needs to be built for each factored constraint. It sorts by increasing donor position the elements in `E` of feature type in the set of upstream compatible features for such a factored constraint.

4.3. Preventing gene splitting

In `GenAmic`, the gene assembly algorithm is slightly more complicated than the `BestGene6` algorithm described in Section 3. Indeed, we have ignored so far the fact that gene assembly constraints indicate not only which features can appear immediately downstream of a given feature in legal gene structures, but also at which interval distances are they allowed to appear. It is biologically reasonable to expect a first exon to appear immediately upstream an internal exon—defining an intron—at a shorter distance than a terminal exon immediately upstream a first exon—defining an intergenic spacer. Biologically realistic limits for these distances may be known, and they can be specified in the definition of the gene model. In fact, in the current implementation of `GenAmic`, they are mandatory to appear in the third column of the gene model file (Figure 5).

```
First+:Internal+                Internal+:Terminal+             50:9999999
Terminal-:Internal-             First-:Internal-                50:9999999
Promoter+                       First+:Single+                  50:4000
First-:Single-                  Promoter-                       50:4000
aataaa-                         Single-:Terminal-               50:4000
aataaa+:Terminal+:Single+       Single+:First+:Promoter+        5000:9999999    block
aataaa+:Terminal+:Single+       Single-:Terminal-:aataaa-       5000:9999999
Promoter-:First-:Single-        Single+:First+:Promoter+        5000:9999999
Promoter-:First-:Single-        Single-:Terminal-:aataaa-       5000:9999999    blockr
```

FIG. 5. Definition of a more complex gene model. The first two columns indicate legal connexions between gene elements as in Figure 2; the third column indicates the minimum and maximum distances at which such connexions can occur. If there are no distance constraits, 0 and tt 9999999 or some other large number can be used. The fourth (optional) column indicates if the legal connexions specified by the first two columns can be prevented at certain regions of the sequence. Note that, according with this gene model, genes may be delimited by promoters and gene termination elements, but this is not necessary.

The gene assembly algorithm needs only to take care that the updated best gene structure for each feature along the set E (see Section 3) is within the allowed interval distances.

Another feature of the gene assembly algorithm in `GenAmic` not present in the `BestGene6`, is the possibility of preventing otherwise legal connections between gene elements to occur in certain regions of the genomic sequence. The main application of this is to prevent assigning predicted exons to different genes, when there is strong evidence that they belong to the same gene—for instance, when they match the same known coding sequence (EST or protein). Since DNA signals defining the boundaries of the genes are ill-defined, and current computational methods to predict, for instance, promoter regions are characterized by rather low accuracy (Fickett and Hatzigeorgiou, 1997), relying in information from database matches to known coding regions may be the most useful way of delimiting gene boundaries.

In the `GenAmic` implementation, an optional fourth column indicates if the connection between elements of the gene features in the constraint is susceptible of being prevented at certain regions of the sequence. Connections between certain features may be prevented at some regions, and connections between other features may be prevented at some other regions. For instance, a consistent set of matches to a known gene along a region of the forward strand of the sequence should prevent gene splitting in this region only in the forward strand, but not in the reverse strand. Therefore, the user-defined keywords appearing in the fourth column of a gene constraint is used to indicate in a separate (optional) third input file, the regions along the sequence, if any, in which legal connections in the constraint are prevented. For instance, in Figure 5, a complex definition of a gene model is shown. In the fourth column of the constraint

```
aataaa+: Terminal+: Single+     Single+: First+: Promoter+
```

which specifies intergenic regions in the forward strand appears the keyword `block`, while in the fourth column of the constraint

```
Promoter-: First-: Single-     Single-: Terminal-: aataaa-
```

which specifies intergenic regions in the reverse strand appears the keyword `blockr`. In a separate file, the regions are specified in which each type of connections are prevented. Figure 6 shows one such file for the test sequence used below to analyze the performance of `GenAmic`. In this case, forward exons occurring, for instance, within the region from 55782 to 119472 won't be assembled in different genes. Similarly reverse exons occurring, for instance, within the region from 275273 to 276530 won't be assembled in different genes.

Note, on the other hand, that the gene model in Figure 5 incorporates features other than exons, such as intronless genes (`Single`), promoter elements (`Promoter`), and gene termination elements (`aataaa`). Note, also, that according with the model definition, a gene in the forward strand can be started by a promoter, since we have

```
Promoter+     First+: Single+
```

but this is not necessary because we also have

```
aataaa+: Terminal+: Single+     Single+: First+: Promoter+
```

```
block    55782  119472
block   123975  125421
block   134007  134250
block   136221  138087
block   196833  197172
block   216666  218232
blockr  275273  276530
block   309027  342720
blockr  315455  315710
block   374475  396488
block   491229  501999
block   574857  575211
block   608061  670962
blockr  703286  703541
```

FIG. 6. Set of regions along the sequence containing the BRCA gene, in which the gene connexions specified in the corresponding constraints in the gene model in Figure 5 are prevented.

Similarly, it may be terminated by a gene termination element, since we have

```
Terminal+: Single+     aataaa+
```

but again this is not necessary because we have

```
aataaa+: Terminal+: Single+     Single+: First+: Promoter+
```

Analogously, for genes in the reverse strand.

4.4. Analysis

As we have already discussed in Section 3, the running time of the `BestGene6` algorithm is strictly proportional to the size of the input set `E` (as it is the generalization of `BestGene6` implemented in `GenAmic`). The preprocessing by `GenAmic` of the input data and the gene model in order to obtain the sorting functions `d` can also be done in linear time, because the set of functions `d` sort integers—which can always be done in linear time. In fact, given the nature of the input data, the sorting functions `d` can be obtained in linear time even using very simple sorting algorithms. The overall running time of `GenAmic` is thus, also strictly proportional to the size of the input set `E`.

The space requirements of the algorithm are also strictly proportional to the size `N` of the input set `E`. The `BestGene6` algorithm itself creates only two additional arrays of size `N`, `G` that stores the index of (a pointer to) the previous gene element for each best gene structure, and `S` which stores the cumulative score of each best gene structure. `GenAmic`, in addition, creates the set of sorting functions `d`, that are simply arrays of pointers to the elements in `E`. The size of each of the arrays `d` is smaller than `N`, and in the worst (and unlikely) case, the number of arrays `d` is equal to the number of different gene features in the gene model. Therefore the space requirements of `GenAmic` are also strictly proportional to the size of `E`.

Although `GenAmic` only exists as a prototyped version written in gawk, we have tested its performance in a large genomic sequence. Indeed, we have used a 773,119-bp-long human sequence containing the BRCA gene (`ftp://ftp.sanger.ac.uk/pub/rd/13q.all.dna`). Using a modified version of the GeneID program (Guigó *et al.*, 1992), which scores exons as log-likelihood odds, we have predicted 78,182 potential exons along this sequence. We have randomly added four potential promoter and five potential gene termination regions (`aataaa` in Figure 5). In addition, using protein sequence database searches we have identified 11 regions in the forward strand of the sequence, and three in the reverse strand in which it is unlikely that the predicted exons belong to more than one gene (Figure 6). We have obtained the optimal gene structure for this data set according with the gene model in Figure 5. Under this model, it took approximately 90 seconds to assemble a 15-gene 190-exon structure from the 78,191 initially predicted gene elements, on a MIPS R10000 (194 Mhz) Silicon Graphics workstation. Since the implementation of `GenAmic` tested here has been written in awk, and implementation written in C would be substantially faster.

To compare the efficiency of the algorithm developed here with that of the quadratic algorithms currently in use for the assembly of genes from sets of predicted exons, we have compared a version of the `BestGene1` algorithm in which frame compatibility is enforced, with `BestGene5`. Both algorithms assemble genes which maximize the sum of scores of the assembled exons, enforcing frame compatibility and considering only one type of exons (see Section 3). `BestGene5` can be assumed to be a simplified version of `GenAmic`, while `BestGene1` would be the analogous to the quadratic algorithms used currently in gene identification programs. Both programs have been written in C, and have been tested in a set of 1,309 genomic sequences from vertebrate organisms, extracted from GenBank rel. 93. The sequences have been extracted following the protocol described in Burset and Guigó (1996). All these sequences are shorter than 60,000 bp and encode a single complete split protein coding gene. Using the aforementioned modified version of the GeneID program (Guigó *et al.*, 1992), internal exons were predicted along these sequences, and given as input to the two gene assembly programs. Figure 7 plots the cpu time required on a MIPS R8000 (75 Mhz) Silicon Graphics workstation for `BestGene1` and `BestGene5` to assemble the optimal genes as a function of the number of input internal exons. As expected, `BestGene1` running time grows polynomially, while `BestGene5` running time, only linearly. Thus, while for small input data sets, the speed of `BestGene1` and `BestGene5` is comparable, for larger input data sets, the running time of `BestGene5` is negligible compared with that of `BestGene1`. Thus, in the most expensive case, it took 49 cpu second for `BestGene1` to assemble the best gene from a set of 9,848 predicted exons, but it took only 0.1 seconds for `BestGene5` to assemble exactly the same gene.

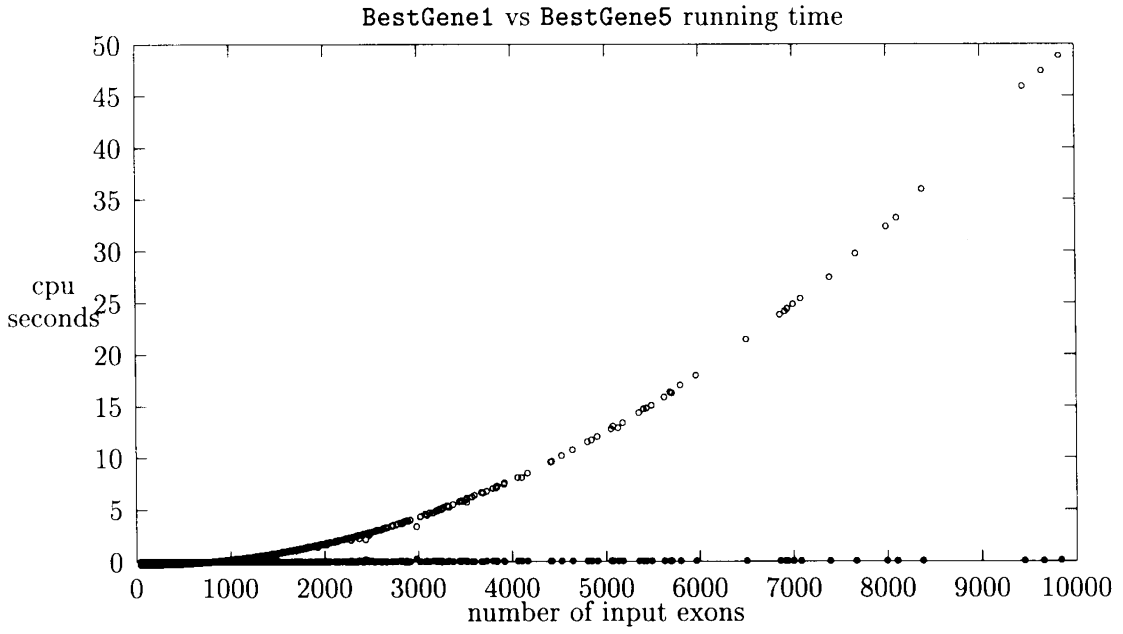

FIG. 7. cpu time required to assemble genes as sequences of initial exons, as a function of the number of input exons. Circles denotes the cpu seconds required by `BestGene1`; bullets denote the cpu seconds required by `BestGene5`.

5. DISCUSSION

We have addressed in this paper the problem of assembling genes structures from a pool of scored gene elements predicted along a DNA genomic sequence. This is the approach taken, in the particular case of single gene structures, in the GeneID (Guigó *et al.*, 1992), GenViewer (Milanesi *et al.*, 1993), GAP3 (Xu *et al.*, 1994), and FGENEH (Solovyev *et al.*, 1994) programs. In all these cases, gene assembly is separated from exon construction, and the goal is explicitly stated of assembling the gene maximizing a function of the scores of the assembled exons. Other gene identification programs rely also on generating a large pool of candidate exons. The SORFIND program (Hutchinson and Hayden, 1992), for instance, generates a set of scored SORFs ("Spliceable Open Reading Frames"), which are essentially potential exons, but it does not attempt to assemble the SORFs into genes. The gm program (Fields and Soderlund, 1990), does not strictly predict potential exons, but introns bounded by ORFs—the so-called "exon maps." A greedy algorithm is used to extend the exon maps into genes, but the function to optimize is not the score of the gene, but the length of the predicted coding region.

In another broad class of gene identification programs, gene assembly is not explicitly separated from exon construction; rather genes are assembled directly from the (atomic signals identified on the input) DNA sequence. We will refer here to these programs as "signal-based." In a number of these programs, the prediction is produced under an statistical model of a gene. In the GeneParser program (Snyder and Stormo, 1993, 1995), the parameters of the model are estimated by means of a neural network, while in the Genie program (Kulp *et al.*, 1996), the Veil program (Henderson *et al.*, 1996) and the GenScan program (Burge and Karlin, 1997), they are estimated under an explicit Hidden Markov Model of an eukaryotic gene. In these cases, the goal is to find the gene parse maximizing the likelihood of the input sequence given the statistical model. In the DynaGene program (Salzberg *et al.*, 1996) decision trees are used to assign to DNA segments the probability that they are coding. Then, the gene parse is obtained maximizing the average coding probability per base in the assembled exons. In the GENLANG program (Dong and Searls, 1994), the prediction is based on a formal grammar of a gene, in which the rules have associated costs. The goal is here to find the most likely gene parse (less costly) of the input sequence given the rules of the gene grammar. In other programs, finally, the parse of the input sequence predicted is the one maximizing a given empirical gene scoring function—much as it happens in the exon-based gene identification programs. This is the case of the GREAT program (Gelfand and Roytberg, 1993; Gelfand *et al.*, 1996), and the program developed by Wu (1996).

Whatever the approach taken, exon- or signal-based, at the core of all these programs there exists a dynamic programming related algorithm that parses the input DNA sequence into predicted gene(s). The dynamic

programming algorithm that we have developed here belongs the exon-based gene assembly approach. While currently available exon-based gene structure prediction programs use gene assembly algorithms whose running time grows proportionally with the square of the number of predicted exons, the running time of our algorithm grows only linearly. Quadratic time becomes prohibitive for large input sets, and in such a case fast assembly of genes can only be performed after a rather stringent filtering of the predicted exons based on coding potential, as it is done, for instance, in the GRAILII/GAP3 (Xu *et al.*, 1994) and FGENEH (Solovyev *et al.*, 1994) programs. Since actual exons may exhibit very low coding potential, such a filtering often implies eliminating actual exons, and therefore the impossibility of assembling the correct gene structure. The algorithm described here, even when implemented in the prototyping language awk, is fast enough that eliminates the need for exon filtering. Speed assembling genes, on the other hand, is not only essential for one-pass analysis of large genomic sequences, but it would also be required for interactive analysis and design of genomes: gene structures may need to be rapidly reassembled while the user is interactively modifying the input query DNA sequence, or a set of predicted gene features.

The implementation of the algorithm described here, `GenAmic`, is not a gene prediction program, however. It assumes gene assembly to be decoupled from gene feature detection, and it addresses only the problem of assembling gene structures from predicted gene elements along a genomic sequence. These gene elements would have been independently predicted using other methods; for instance, the exon prediction modules of other gene identification programs. In particular, thus, `GenAmic` could be included in all currently available exon-based gene identification programs (see Note Added in Proof), increasing substantially their efficiency. But, in general, `GenAmic` can be used to process the output of any gene feature detection program, which produces scored exons, and other gene elements. The program will be particularly appropriate when these gene elements are scored as log-odds likelihoods, because then the score of the optimal gene structure is also a log-likelihood.

Because the definition of valid gene structures is not encoded in the program, but defined externally in a gene model, `GenAmic` allows for great flexibility in formulating the gene identification problem in the practice. Indeed, different gene models can be used to face different practical problems. For instance, if there is strong evidence that the query sequence contains only one gene, a single-gene one-strand model would be used. If only the partial structure of a region of a gene wants to be explored for possible alternative splicing patterns, an even simpler gene model specifying only that `exons` may appear upstream `exons` could be used. On the other hand, a multiple-gene two-strand gene model would be used to decipher full structures in large genomic regions. If, in such a case, additional information from potential promoter regions wants to be used, the gene model would specify `promoters` as gene starters, either as mandatory or optional gene elements.

In summary, because of its speed, we believe that the algorithm implemented here could be useful for gene prediction in large genomic regions. Because of its flexibility and given the limited accuracy of current computational methods for gene prediction, it could also be of utility in exploratory genome analysis.

SOFTWARE AVAILABILITY

The software described in this paper is freely available through anonymous ftp to `apolo.imim.es` in the directory `pub/GeneAssembly`. This directory contains the awk code of `GenAmic`, as well as the instructions on how to use it.

ACKNOWLEDGMENTS

This work was supported by Proyecto de la Dirección General de Investigación Científica y Técnica from the Ministerio de Educación y Ciencia (Spain), PB94-1278, and by Public Health Service Grant HG00981-01A1 from the National Center for Human Genome Research (USA.) I thank Moisés Burset for useful discussions, and Pep Abril for developing the postscript program to generate figures such as Figure 1.

NOTE ADDED IN PROOF

A simplified version of the algorithm described here has been incorporated in the FGENES program (Victor Solovyev, personal communication).

REFERENCES

Burset, M., and Guigó, R. 1996. Evaluation of gene structure prediction programs. *Genomics* 34, 353–367.

Burge, C., and Karlin, S. 1997. Prediction of complete gene structures in human genomic DNA. *J. Mol. Biol.* 268, 78–94.

Claverie, J.M. 1997. Computational Methods for the identification of genes in vertebrate genomic sequences. *Hum. Mol. Genet.* 6, 1735–1744.

Dong, S., and Searls, D.B. 1994. Gene structure prediction by linguistic methods. *Genomics* 23, 540–551.

Durbin, R., and Haussler, D. 1997. GFF: a proposed exchange format for gene-finding features. `http://www.sanger.ac.uk/rd/gff.html`

Fickett, J.W. 1996. Finding genes by computer: the state of the art. *Trends Genet.* 12, 316–320.

Fickett, J.W., and Guigó R. 1996. Computational gene identification, 73–100. *In* Swindell, S., Miller, R., and Myers, G., eds. *Internet for the Molecular Biologist.* Horizon Scientific Press, Wymondham, U.K.

Fickett, J.W., and Hatzigeorgiou, A. 1997. Ukaryotic promoter recognition. *Genome Res.* 7, 861–878.

Fickett, J.W., and Tung, C.-S. 1992. Assessment of protein coding measures. *Nucleic Acids Res.* 20, 6441–6450.

Fields, C.A., and Soderlund, C.A. 1990. gm: a practical tool for automating DNA sequence analysis. *Comput. Appl. Biosci.* 6, 263–27.

Gelfand, M.S., and Roytberg, M.A. 1993. Prediction of the exon-intron structure by a dynamic programming approach. *BioSystems* 30, 173–182.

Gelfand, M.S. 1995. Prediction of function in DNA sequence analysis. *J. Comput. Biol.* 1, 87–115.

Gelfand, M.S., Podolsky, L.I., Astakhova, T.V, and Roytberg, M.A. 1996. Recognition of genes in human DNA sequences. *J. Comput. Biol.* 3, 223–234.

Guigó, R., Knudsen, S., Drake, N., and Smith, T.F. 1992. Prediction of gene structure. *J. Mol. Biol.* 226, 141–157.

Guigó, R. 1997. Computational gene identification. *J. Mol. Med.* 75, 389–387.

Henderson, J., Salzberg, S., and Fasman, K. 1997. Finding genes in DNA with a hidden Markov model. *J. Comput. Biol.* 4, 1217–1141.

Hutchinson, G.B., and Hayden, M.R. 1992. The prediction of exons through an analysis of spliceable open reading frames. *Nucleic Acids Res.* 20, 3453–3462.

Kulp, D., Haussler, D., Reese, M.G., and Eeckman, F.H. 1996. A generalized hidden Markov model for the recognition of human genes in DNA, 134–142. *In* States, D.J., Agarwal, P., Gaasterland, T., Hunter, L., and Smith, R., eds. *ISMB-96.* AAAI Press, Menlo Park, CA.

Milanesi, L., Kolchanov, N.A., Rogozin, I.B., *et al.* 1993. GenViewer: a computing tool for protein-coding regions prediction in nucleotide sequences, 573–587. *In* Lim, H.A., Fickett, J.W., Cantor, C.R., and Robbins, R.J., eds. *Proceedings of the 2nd International Conference on Bioinformatics, Supercomputing and Complex Genone Analysis.* World Scientific, Singapore.

Milanesi, L., Kolchanov, N., Rogozin, I., Kel, A., and Titov, I. 1994. Sequence functional inference, 249–312. *In* Bishop M.J., ed. *Guide to Human Genome Computing.* Academic Press, London.

Salzberg, S., Chen, X., Henderson, J., and Fasman, K. 1996. Finding genes in DNA using decision trees and dynamic programming, 201–210. *In* States, D.J., Agarwal, P., Gaasterland, T., Hunter, L., and Smith, R., eds. *ISMB-96.* AAAI Press, Menlo Park, CA.

Snyder, E.E., and Stormo, G.D. 1993. Identification of coding regions in genomic DNA sequences: an application of dynamic programming and neural networks. *Nucleic Acids Res.* 21, 607–613.

Snyder, E.E., and Stormo, G.D. 1995. Identification of protein coding regions in genomic. DNA. *J. Mol. Biol.* 248, 1–18.

Solovyev, V.V., Salamov, A.A., and Lawrence, C.B. 1994. Predicting internal exons by oligonucleotide composition and discriminant analysis of spliceable open reading frames. *Nucleic Acids Res.* 22, 5156–5163.

Wu, T.D. 1996. A segment-based dynamic programming algorithm for predicting gene structure. *J. Comp. Biol.* 3, 375–394.

Xu, Y., Mural, R.J., and Uberbacher, C. 1994. Constructing gene models from accurately predicted exons: an application of dynamic programming. *Comput. Appl. Biosci.* 10, 613–623.

Address reprint requests to:
Roderic Guigó
Informàtica Mèdica
Institut Municipal d'Investigació Mèdica (IMIM)
C/Dr. Aiguader 80
E-08003 Barcelona
Spain

rguigo@imim.es

Received for publication January 14, 1997; accepted as revised August 4, 1998.