13 Chaining

This exposition was developed by Clemens Gröpl. It is based on the following references, which are all suggested reading:

- Mohamed I. Abouelhoda, Enno Ohlebusch: *Chaining algorithms for multiple genome comparison*, Journal of Discrete Algorithms, 3:321-341, 2005. [AO05]
- Enno Ohlebusch, Mohamed I. Abouelhoda: *Chaining algorithms and applications in comparative genomics*, Handbook of Computational Molecular Biology, Chapter 15, 2006. [AO06]
- Gene Myers, Webb Miller: *Chaining multiple-alignment fragments in sub-quadratic time*, SODA 1995. [MM95]

The work of Ohlebusch and Abouelhoda was originally presented in two conference papers at CPM03 and WABI03. These are now mostly superseded by the two articles above.

13.1 The chaining problem

Biological sequence analysis often has to deal with sequences of genomic size which cannot be aligned as a whole in one step. In this situation, an *anchor-based* approach can be used. The idea is to construct a (more or less) global alignment from a collection of local alignments. The small pieces are also called *fragment matches* or just *fragments*. These pieces then have to be *chained* one after another. The chaining problem is difficult because no all fragment are compatible with each other. The general approach is as follows:

1. *Compute fragments*, i. e. regions in the genomes that are very similar.
2. *Compute an optimal global chain* of colinear non-overlapping fragments. The fragments of the chain are the anchors for the alignment.
3. *Fill in the gaps* by aligning the regions between the anchors.

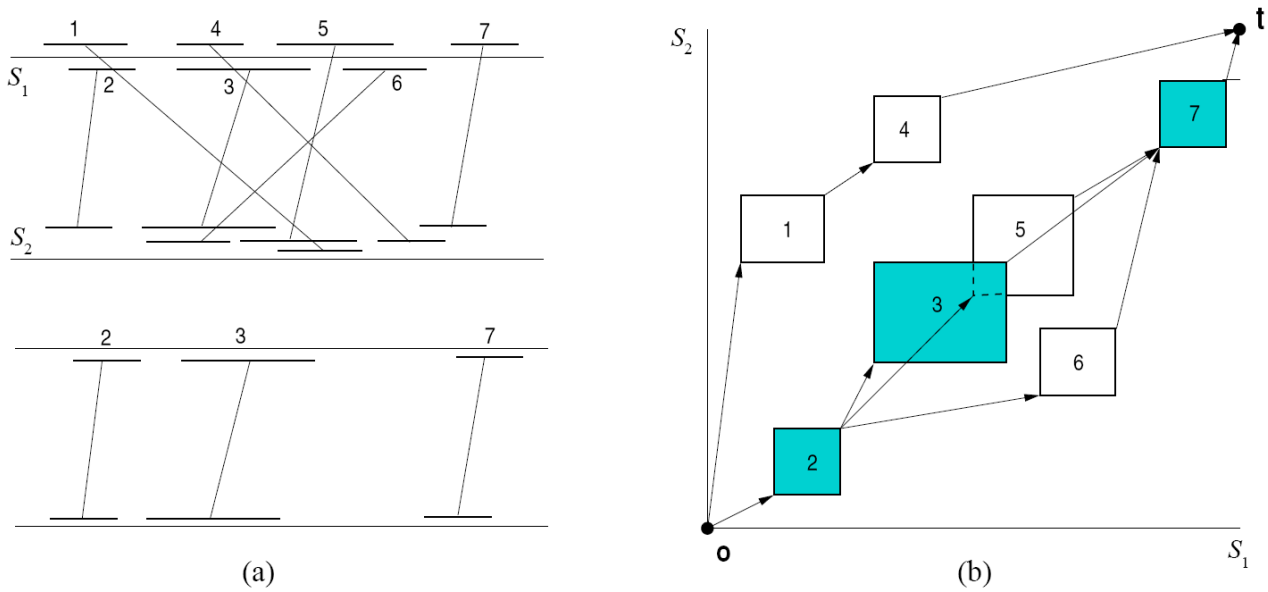
A variant of the problem (2'.) tries to find a set of local chains instead of one global chain. We will return to this later.

Naturally we are also interested in multiple alignments of k genomic sequences. For the chaining problem, we only care about where each fragment begins and ends in each of the k sequences.

Each *fragment* has a *weight*. The weight could be the length of an exact match, or another measure of its statistical significance.

The *gaps* in a chain of fragments need to be *scored* as well. We will see a number of scoring schemes for this below.

Example. To the left, you see a collection of pairwise ($k = 2$) alignments (top) and a chained subset of them (bottom). To the right you see a two-dimensional visualization of the same fragment matches. An arrow indicates that one fragment can follow another fragment in a chain. (Not all possible arrows are shown.)



13.2 The chaining problem

Note that not all pairs of fragments are compatible to each other: The fragments in the chain have to be *collinear* and *non-overlapping*.

- *Collinear*: The order of the respective segments of the fragments is the same in both sequences.
- *Non-overlapping*: Two fragments overlap if their segments overlap in one of the sequences.

In the pictorial representation of Figure (a), two fragments are collinear if the lines connecting their segments are non-crossing (for example, the fragments 2 and 3 are collinear, while 1 and 6 are not). Two fragments overlap if their segments overlap in one of the genomes (for example, the fragments 1 and 2 are overlapping, while 2 and 3 are non-overlapping). Figure (b) shows the fragments as rectangles. Fragments 2, 3, 7 are chained. `0` and `t` are artificial source and sink nodes.

Among the many bioinformatics **tools** solving chaining problems (in some way) are:

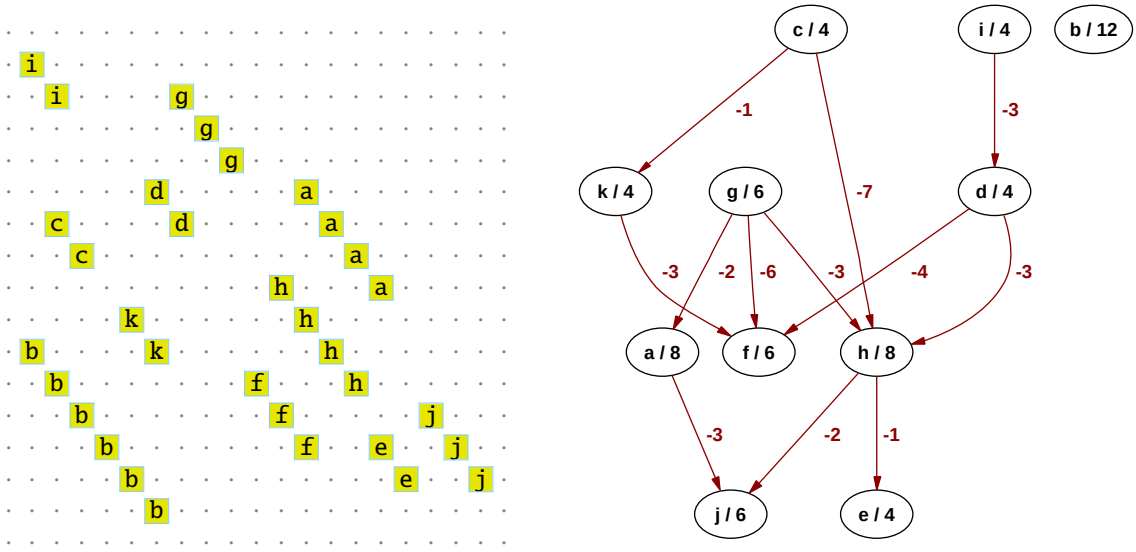
- *FastA*. Fragments are constant-size exact matches. They are found by hashing.
- *MUMmer*. Fragments are maximal exact matches. They are found using suffix trees.
- *MGA*. Fragments are maximal multiple exact matches. They are found using suffix arrays.
- *GLASS*. Fragments are exact k-mers.
- *DIALIGN*. Fragments can have substitutions.
- *MultiPipMaker*. Fragments can have substitutions and indels.
- ... (add your favorite genome alignment tool here)

The fragments could also be *BLAST* hits, or even *local chains* from an earlier invocation of the algorithm.

13.3 Graph-theoretic formulation

Clearly the chaining problem can be stated in graph-theoretic terms. The fragments correspond to vertices and we have an edge whenever two fragments are compatible, i. e. they can be combined in a chain. Thus we obtain a weighted acyclic directed graph. The goal is to find a directed path of maximum weight. A simple dynamic programming requires $O(V + E)$ time and space.

Example.



Scoring scheme: For simplicity, each "letter" scores 2 and all other columns of the corresponding alignment score −1 (diagonal "shortcuts" are allowed).

The graph theoretic approach is intuitive and will lead us to the final solution. But some essential issues are not settled yet:

- How should the edge set be constructed? Of course we can enumerate over all pairs of vertices, but this will take $\Omega(V^2)$ time, which is prohibitive in many real-world cases.
- Even worse, it is easy to construct instances where the size of the edge set is indeed $\Omega(V^2)$. It may help to remove "transitive" edges which cannot be part of an optimal solution, but this does not improve the worst-case asymptotic. (*Exercise: find such an example.*)
- The obvious dynamic programming algorithm does not fully exploit the geometric nature of the problem. It works on every weighted acyclic directed graph, but our graph is more restricted.

The good news is that *we need not construct the edge set explicitly*. Instead we will traverse the vertices in an appropriate order and use some auxiliary data structures and still carry out essentially the same DP recursion.

13.4 Notation

Before we can get more concrete, we need to introduce some notation.

- For $1 \le i \le k$, let $S_i = S_i[1..n_i]$ be a sequence of *length* $|S_i| = n_i$.
- $S_i[l..h]$ denotes the *substring* of S_i starting at position l and ending at position h. (Both margins are included.)
- A *fragment* f consists of two k-tuples $\text{beg}(f) = (l_1, l_2, \ldots, l_k)$ and $\text{end}(f) = (h_1, h_2, \ldots, h_k)$ such that the strings $S_1[l_1..h_1]$, $S_2[l_2..h_2]$,...,$S_k[l_k..h_k]$ are similar in some sense.
- A fragment f of k sequences can be *represented* by a *hyper-rectangle* in $\mathbb{R}^k$ with two extremal corner points $\text{beg}(f)$ and $\text{end}(f)$. We write $R(p,q) := [p_1,q_1] \times \ldots \times [p_k,q_k]$ for the hyper-rectangle spanned by $p = (p_1,\ldots,p_k)$ and $q = (q_1,\ldots,q_k)$, where $p < q$, i. e., $p_i < q_i$ for all $1 \le i \le k$. Thus a fragment f is represented by hyper-rectangle $R(\text{beg}(f), \text{end}(f))$. Each coordinate of the corners is a nonnegative integer.
- Consequently, the number of sequences k is also called the *dimension*.
- Each fragment f has a *weight* $f.\text{weight} \in \mathbb{R}$.
- We will sometimes identify $\text{beg}(f)$ or $\text{end}(f)$ with f.
- For ease of presentation, we introduce *artificial fragments* of weight zero. We let $\mathtt{0} := (0,\ldots,0)$ and $\mathtt{t} := (n_1+1,\ldots,n_k+1)$. We let $\text{beg}(\mathtt{0}) := \perp$ (where $\perp$ means "undefined") and $\text{end}(\mathtt{0}) := (0,\ldots,0)$. Analogously, $\text{beg}(\mathtt{t}) := (n_1+1,\ldots,n_k+1)$ and $\text{end}(\mathtt{t}) := \perp$.

- The *coordinates* of a point $p \in \mathbb{R}^k$ will be accessed as $p.x_1, \ldots, p.x_k$. If $k = 2$, we will also write them as $p.x$, $p.y$.

Definition. We define a binary relation $\ll$ for fragments by

$$f \ll f' \iff \forall_i : \text{end}(f).x_i < \text{beg}(f').x_i$$

In this case, we say that f *precedes* f'. In the graph model, we have an arrow from f to f'.

Note that $\mathtt{0} \ll f \ll \mathtt{t}$ for all fragments f with $f \neq \mathtt{0}$, $f \neq \mathtt{t}$, as required.

13.5 Notation

Definition.

- A *chain* (of collinear, non-crossing fragments) is a sequence of fragments $C = (f_1, \ldots, f_\ell)$ such that $f_i \ll f_{i+1}$ for all $1 \leq i < \ell$.
- The *score* of C is

$$\text{score}(C) := \sum_{i=1}^{\ell} f_i.\text{weight} - \sum_{i=1}^{\ell-1} g(f_{i+1}, f_i)\,,$$

where $g(f_{i+1}, f_i)$ is the *gap cost* for connecting fragment f_i to f_{i+1} in the chain.

The *global fragment chaining problem* is:

> Given a set of weighted fragments and a gap cost function, determine a chain of maximum score starting at $\mathtt{0}$ and ending at $\mathtt{t}$.

13.6 The basic algorithm

For each fragment f, denote by f.score the maximum score of all chains starting at $\mathtt{0}$ and ending at f.

Clearly this score satisfies the following *recurrence*:

$$f'.\text{score} \;=\; f'.\text{weight} \;+\; \max\left\{ f.\text{score} - g(f', f) \;\middle|\; f \ll f' \right\}.$$

The simple dynamic programming algorithm for the global fragment chaining problem processes the fragments in a topological ordering (e. g., the lexicographic ordering). This is to ensure that when f'.score is going to be computed, all values f.score, where $f \ll f'$, have already been determined.

The *initialization* is

$$\mathtt{0}.\text{score} := 0\,.$$

And the value we are finally interested in is $\mathtt{t}$.score.

The chain itself can be found by backtracing, as usual.

13.7 Range maximum queries

So how are we going to improve upon this straightforward solution? The key is to compute the 'max' in the DP recursion more efficiently, without 'generating' all the edges of the graph. *Range (maximum) queries* are a fundamental operation in computational geometry, and efficient data structures have been designed for them.

Definition.

- Let $S \subseteq \mathbb{R}^k$ be a set of points and let $p, q \in \mathbb{R}^k$, $p < q$ be the corners of a rectangle $R(p, q)$. Then the *range query* $RQ(p, q)$ asks for all points of S that lie in the hyper-rectangle $R(p, q)$.
- Assume further that the points in S have a score associated with them. Then the *range maximum query* $RMQ(p, q)$ asks for a point of maximum score in $R(p, q)$.

Observation. *Consider the special case when the gap cost function g is always zero. Let $\vec{1} := (1, \ldots, 1)$. The recurrence of the basic algorithm implies:*

If RMQ($\mathbb{0}$, $beg(f') - \vec{1}$) returns the end point of fragment f, then $f'.score = f'.weight + f.score$.

In other words, range maximum queries are what we need in the DP recursion!

13.8 Global chaining without gap costs

In the following, we will first consider the case when all gap costs are zero. After that we will see how the algorithm can be modified to deal with (certain) gap cost functions.

The algorithm uses the *line-sweep* paradigm to construct an optimal chain. It processes the begin and end points of the fragments in ascending order of their x_1 coordinate. One can think of the traversal as a line (actually, a hyper-plane of dimension $k-1$) that "sweeps" the points with respect to their x_1 coordinate.[1] If a point has already been scanned by the sweeping line, it is said to be *active*, otherwise it is said to be *inactive*.

Let s be the point that is currently being scanned. The x_1 coordinates of the active points are less than or equal to that of s. By the preceding observation, if s is the begin of fragment f', then an optimal chain ending at f' can be found by a *range maximum query* over the active points. Otherwise, s is the end of a fragment and can be *activated*. Note that the *RMQ* need not take the first coordinate x_1 into account because of the sweeping order.

The algorithm uses a *semi-dynamic data structure D* that stores the end points of the fragments and supports two operations:

1. *Activation*
2. *Range maximum queries over active points.*

It is called semi-dynamic because we can only 'activate' but not 'inactivate' points. Abouelhoda and Ohlebusch [AO05] have designed such a data structure that supports *RMQ*s with activation in $O(n \log^{d-1} n \log\log n)$ time and $O(n \log^{d-1} n)$ space for n points in d dimensions.

Since $d = k-1$ in the algorithm, their algorithm solves the global fragment chaining problem in $O(n \log^{k-2} n \log\log n)$ time and $O(n \log^{k-2} n)$ space. (This holds for $k \geq 3$; for $k = 2$ the running time is dominated by the initial sorting.)

The correctness of the following algorithm is immediate.

Algorithm. (k-dimensional chaining of n fragments)

```
(1)  // Data structure D is (k − 1)-dimensional, x_1 is ignored.
(2)  // π denotes the projection.
(3)  points := begin and end points of all fragments (including 0 and t)
(4)  sort points by x_1
(5)  store all end points of fragments as inactive in D
(6)  for i = 1 to 2n + 2 do
(7)      if points[i] == beg(f') for some fragment f'
(8)        then
(9)              q := RMQ(0, π(points[i] − 1⃗))
(10)             f := fragment with end(f) == q and maximum score
(11)             f'.prec := f
(12)             f'.score := f'.weight + f.score
(13)       else // points[i] == end(f') for some fragment f'
(14)             activate(π(points[i])) in D
(15)     fi
(16) od
```

The details of the data structure D are fairly complicated. It combines *range trees* with *fractional cascading* and *van Emde Boas priority queues* (with *Johnson's improvement*) ... enough topics for more than another lecture! We therefore skip this part of [AO05, AO06] :-(Note that the actual implementation of Abouelhoda uses KD-trees, which are non-optimal with respect to the running times but much easier to implement. (Explained on the blackboard.)

[1] engl. "to sweep" = dt. "fegen"

At this point, we have an algorithm for the global chaining problem without gap costs. Next we show how to modify the algorithm to take certain gap cost functions into account.

13.9 Incorporating L_1 gap costs

Definition. For two points $p, q \in \mathbb{R}^k$, the L_1 *distance* is defined by

$$d_1(p,q) := \sum_{i=1}^{k} \left| p.x_i - q.x_i \right| .$$

Definition. The L_1 *gap function* is

$$g_1(f', f) := d_1(\text{beg}(f'), \text{end}(f)) \qquad \text{for } f \ll f'.$$

The effect of the L_1 gap function is that *all* characters between f and f' are scored as indels (not a very realistic assumption, but maybe not so bad either):

```
ACCXXXX---AGG
ACC----YYYAGG
```

In this example `ACC` and `AGG` are the anchors of the alignment and `X` and `Y` are anonymous characters.

The problem with gap costs is that a range maximum query does not immediately take the gap cost $g(f', f)$ into account when f'.score is computed.

But we can express the gap costs *implicitly* in terms of the weight information attached to the points. Let us define the *geometric cost* of a fragment f as

$$\text{gc}(f) := d_1(\mathtt{t}, \text{end}(f)) .$$

Since `t` is fixed, the value $\text{gc}(f)$ is known in advance for every fragment f. Moreover, we have the following lemma:

Lemma 1. Let f, $\tilde{f}$, and f' be fragments such that $f \ll f'$ and $\tilde{f} \ll f'$. Then

$$\tilde{f}.\text{score} - g_1(f', \tilde{f}) > f.\text{score} - g_1(f', f) \iff \tilde{f}.\text{score} - \text{gc}(\tilde{f}) > f.\text{score} - \text{gc}(f) .$$

Proof: Exercise. (See figure on blackboard.)

Therefore we need only two slight modifications to the algorithm in order to take L_1 gap costs into account.

1. Replace the statement
$$f'.\text{score} := f'.\text{weight} + f.\text{score}$$
with
$$f'.\text{score} := f'.\text{weight} + f.\text{score} - g_1(f', f) .$$
2. If points[i] is the end point of f', then it is activated with *priority*
$$f'.\text{priority} := f'.\text{score} - \text{gc}(f') .$$
Thus the *RMQ*s will return a point of maximum priority instead of a point of maximum score.

Then we have:

Lemma 2. If $RMQ(\mathtt{0}, \text{beg}(f') - \vec{1})$ returns the end point of a fragment $\tilde{f}$, then we have

$$\tilde{f}.\text{score} - g_1(f', \tilde{f}) = \max\{ f.\text{score} - g_1(f', f) \mid f \ll f' \} .$$

Proof: Application of the preceding lemma.

Therefore the modified algorithm is correct.

Algorithm. (k-dimensional chaining of n fragments using L_1 gap cost)

```
(1)  // Data structure D is (k − 1)-dimensional, x_1 is ignored.
(2)  // π denotes the projection.
(3)  points := begin and end points of all fragments (including 0 and t)
(4)  sort points by x_1
(5)  store all end points of fragments as inactive in D
(6)  for i = 1 to 2n + 2 do
(7)      if points[i] == beg(f') for some fragment f'
(8)        then
(9)              q := RMQ(0, π(points[i] − 1⃗))
(10)             f := fragment with end(f) == q and maximum score
(11)             f'.prec := f
(12)             f'.score := f'.weight + f.score − g_1(f', f)
(13)       else // points[i] == end(f') for some fragment f'
(14)             f'.priority := f'.score − gc(f')
(15)             activate(π(points[i])) in D with priority f'.priority
(16)     fi
(17) od
```

13.10 Incorporating sum-of-pairs gap costs

The L_1 gap cost model is not very realistic. An alternative scoring model was introduced by Myers and Miller [MM95]. We consider the case $k = 2$ first, since the case $k > 2$ is rather involved. (The name 'sum-of-pairs cost' is a bit misleading, just take it as a name.)

For two points $p, q \in \mathbb{R}^k$, we write

$$\Delta_{x_i}(p, q) := \left|p.x_i - q.x_i\right|$$

The sum-of-pairs distance depends on two parameters ϵ and λ. It is defined (for $k = 2$) as

$$\begin{aligned} d(p, q) &:= \begin{cases} \epsilon\Delta_{x_2} + \lambda(\Delta_{x_1} - \Delta_{x_2}) & \text{if } \Delta_{x_1} \geq \Delta_{x_2} \\ \epsilon\Delta_{x_1} + \lambda(\Delta_{x_2} - \Delta_{x_1}) & \text{if } \Delta_{x_2} \geq \Delta_{x_1} \end{cases} \\ &= \begin{cases} \lambda\Delta_{x_1} + (\epsilon - \lambda)\Delta_{x_2} & \text{if } \Delta_{x_1} \geq \Delta_{x_2} \\ \lambda\Delta_{x_2} + (\epsilon - \lambda)\Delta_{x_1} & \text{if } \Delta_{x_2} \geq \Delta_{x_1} . \end{cases} \end{aligned}$$

We drop the arguments p, q if they are clear from the context.

Definition. The *sum-of-pairs gap function* is

$$g(f', f) := d(\text{beg}(f'), \text{end}(f)) \qquad \text{for } f \ll f'.$$

The effect of the sum-of-pairs gap function depends on the parameters ϵ and λ.

- $\lambda > 0$ is the cost of aligning an anonymous character with a gap position in the other sequence.
- $\epsilon > 0$ is the cost of aligning two anonymous characters to each other.

For $\lambda = 1$ and $\epsilon = 2$, g coincides with g_1, whereas for $\lambda = 1$ and $\epsilon = 1$, we obtain the L_∞ *metric*: $g_\infty(f', f) := d_\infty(\text{beg}(f'), \text{end}(f))$, where

$$d_\infty(p, q) = \max\{\, \left|p.x_i - q.x_i\right| \mid 1 \leq i \leq k \,\}.$$

With $\lambda > \epsilon/2 > 0$ the characters between two fragments are replaced as long as possible and the remaining characters are inserted or deleted:

```
ACCXXXXAGG
ACCYYY-AGG
```

(Compare this example to the one for the L_1 metric.)

Otherwise it would be best to connect fragments entirely by gaps as in the L_1 metric. Thus we require that $\lambda > \epsilon/2$.

We need a few definitions in order explain how the score of a fragment f' can be computed.

Let $s := \text{beg}(f')$. The *first quadrant* of s consists of all points $p \in \mathbb{R}^2$ with $p.x_1 \leq q.x_1$ and $p.x_2 \leq q.x_2$.

We subdivide the first quadrant of s into its *first* and *second octant* O_1, O_2. The points in the first octant O_1 satisfy $\Delta_{x_1} \geq \Delta_{x_2}$; they are above or on the straight line $x_2 = x_1 + (s.x_2 - s.x_1)$. The points in the second octant satisfy $\Delta_{x_2} \geq \Delta_{x_1}$.

Clearly,

$$f'.\text{score} = f'.\text{weight} + \max\{v_1, v_2\}\,,$$

where

$$v_i := \max\{\, f.\text{score} - g(f', f) \mid f \ll f' \text{ and } \text{end}(f) \text{ lies in octant } O_i \,\}$$

for $i = 1, 2$. Thus we can eliminate the case distinction for g within each octant.

We are not finished yet, since our chaining algorithm relies on *RMQ*s, which work for orthogonal regions, not octants! But a simple trick can help.

We apply the *octant-to-quadrant transformations* of Guibas and Stolfi: The transformation $T_1 : (x_1, x_2) \mapsto (x_1 - x_2, x_2)$ maps the first octant to the first quadrant. Similarly, $T_2 : (x_1, x_2) \mapsto (x_1, x_2 - x_1)$ does the same task for the second octant. By means of these transformations, we can apply the same techniques as for the L_1 metric.

We only need to define the *geometric cost* gc_i properly for each octant O_i.

Lemma 3. Let f, $\tilde{f}$, and f' be fragments such that $f \ll f'$ and $\tilde{f} \ll f'$. If $\text{end}(f)$ and $\text{end}(\tilde{f})$ lie in the first octant of $\text{beg}(f')$, then

$$\tilde{f}.\text{score} - g(f', \tilde{f}) > f.\text{score} - g(f', f) \iff \tilde{f}.\text{score} - \text{gc}_1(\tilde{f}) > f.\text{score} - \text{gc}_1(f)\,,$$

where

$$\text{gc}_1(\hat{f}) = \lambda\Delta_{x_1}(\mathtt{t}, \text{end}(\hat{f})) + (\epsilon - \lambda)\Delta_{x_2}(\mathtt{t}, \text{end}(\hat{f}))$$

for any fragment $\hat{f}$.

Proof: Exercise. (Similar to the preceding one.)

An analogous lemma holds for the second octant.

Due to the octant-to-quadrant transformations, we need to take two different geometric costs gc_1, gc_2 into account. Consequently, the nodes will also have different scores with respect to both. To cope with this, we use two data structures D_1, D_2. Each D_i stores the points of the set $\{\, T_i(\text{end}(f)) \mid f \text{ is a fragment} \,\}$. The results of both *RMQ*s need to be merged. We omit the details.

We still have a line-sweep algorithm, but the data structures D_i need to store two-dimensional points. Thus the algorithm runs in $O(n \log n \log\log n)$ time and $O(n \log n)$ space.

For *dimensions* $k > 2$, the sum-of-pairs gap cost is defined for fragments $f \ll f'$ by

$$g_{\text{sop}}(f', f) := \sum_{0 \leq i < j \leq k} g(f'_{i,j}, f_{i,j})\,,$$

where $f'_{i,j}$ and $f_{i,j}$ are the two-dimensional "projections" of the fragments.

The general idea is similar as for $k = 2$, but now the *first hyper-corner* of $\text{beg}(f')$ is subdivided into $k!$ hyper-regions.

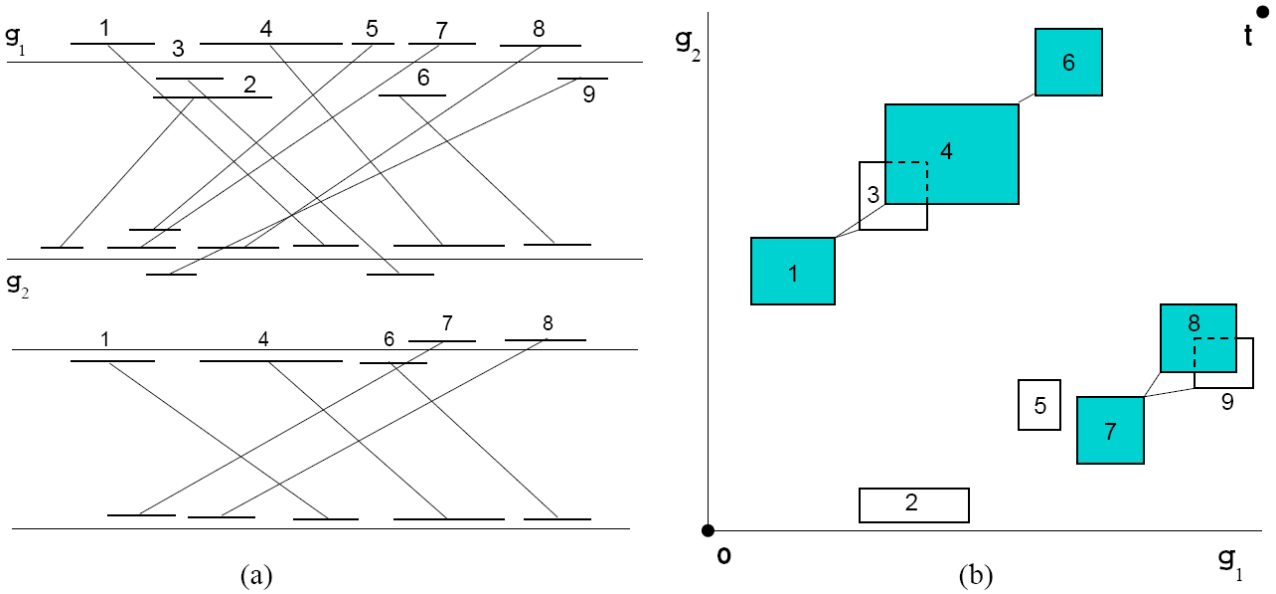
The resulting algorithm runs in $O(k!n \log^{k-1} n \log\log n)$ time and $O(k!n \log^{k-1} n)$ space.

13.11 Local chaining

So far we have been searching for an optimal chain starting at $\mathtt{0}$ and ending at $\mathtt{t}$. If we remove the restriction that a chain must start at $\mathtt{0}$ and end at $\mathtt{t}$, we obtain the local chaining problem.

Definition. The *local fragment chaining problem* is: Given a set of weighted fragments and a gap cost function, determine a chain of maximum score ≥ 0. Such a chain will be called an *optimal local chain*.

Example:



The optimal chain is composed of the fragments 1, 4, and 6. Another significant local chain consists of the fragments 7 and 8.

Definition. Let

$$f'.\text{score} := \max\{\, \text{score}(C) \mid C \text{ is a chain ending with } f' \,\}$$

Definition. A chain C ending with f' and satisfying $f'.\text{score} = \text{score}(C)$ is called an *optimal chain ending with* f'.

We explain the idea using L_1 gap costs for simplicity.

Lemma. The following equality holds:

$$f'.\text{score} \;=\; f'.\text{weight} \;+\; \max\{\, 0,\; f.\text{score} - g_1(f', f) \mid f \ll f' \,\}.$$

Once again we have switched from global to local alignment by 'adding a zero at the right place'.

Now the modification to the line-sweep algorithm is straight-forward.

13.12 KD-trees

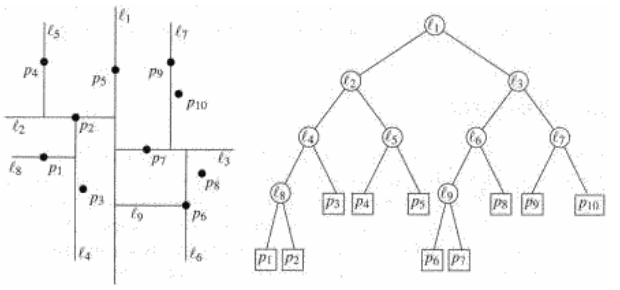
The actual implementation done by Abouelhoda uses a *KD-tree* data structure.

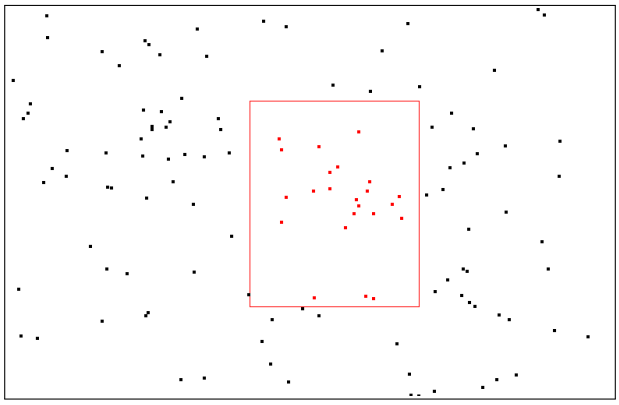
A nice Java applet illustrating KD-trees (for dimension $k = 2$) can be found via Hanan Samet's homepage, `http://www.cs.umd.edu/users/hjs/`, the direct link to the applet is `http://donar.umiacs.umd.edu/quadtree/points/kdtree.html`. This is the one I presented in the lecture, but it was offline on 2007-06-22, so I went on to another one.

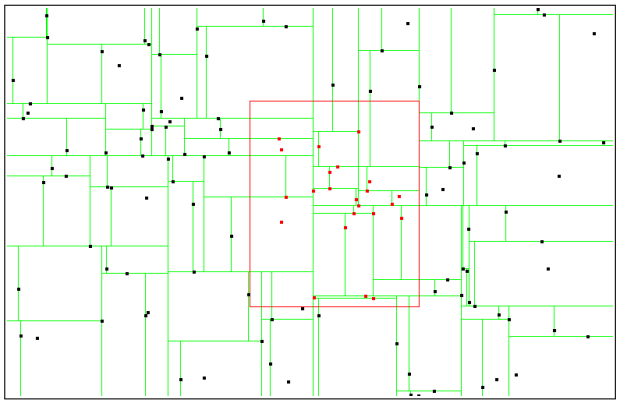
Another nice Java applet can be found via Hüseyin Akcan's homepage, `http://cis.poly.edu/~hakcan01/`, the direct link to the applet is `http://cis.poly.edu/~hakcan01/projects/kdtree/kdTree.html`, and this is also where I have copied the pseudocode from.

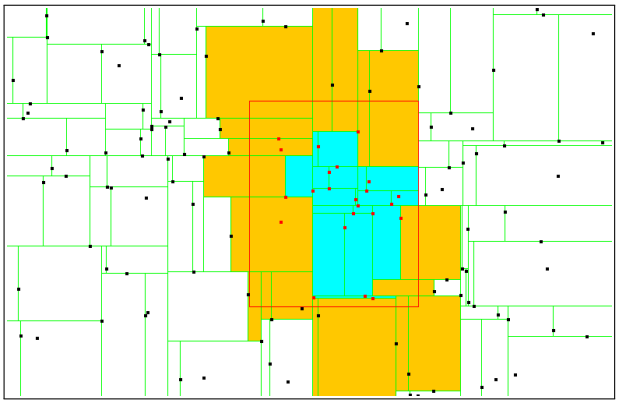
Algorithm BUILDKDTREE($P, depth$)
Input. A set of points P and the current depth $depth$.
Output. The root of a kd-tree storing P.
1. **if** P contains only one point
2. **then return** a leaf storing this point
3. **else if** $depth$ is even
4. **then** Split P into two subsets with a vertical line ℓ through the median x-coordinate of the points in P. Let P_1 be the set of points to the left of ℓ or on ℓ, and let P_2 be the set of points to the right of ℓ.
5. **else** Split P into two subsets with a horizontal line ℓ through the median y-coordinate of the points in P. Let P_1 be the set of points below ℓ or on ℓ, and let P_2 be the set of points above ℓ.
6. $v_{left} \leftarrow$ BUILDKDTREE($P_1, depth + 1$)
7. $v_{right} \leftarrow$ BUILDKDTREE($P_2, depth + 1$)
8. Create a node v storing ℓ, make v_{left} the left child of v, and make v_{right} the right child of v.
9. **return** v

Algorithm SEARCHKDTREE(v, R)
Input. The root of (a subtree of) a kd-tree, and a range R.
Output. All points at leaves below v that lie in the range.
1. **if** v is a leaf
2. **then** Report the point stored at v if it lies in R.
3. **else if** $region(lc(v))$ is fully contained in R
4. **then** REPORTSUBTREE($lc(v)$)
5. **else if** $region(lc(v))$ intersects R
6. **then** SEARCHKDTREE($lc(v), R$)
7. **if** $region(rc(v))$ is fully contained in R
8. **then** REPORTSUBTREE($rc(v)$)
9. **else if** $region(rc(v))$ intersects R
10. **then** SEARCHKDTREE($rc(v), R$)









The complexities of basic operations supported by KD-trees are as follows:

- Building a static kd-tree from n points takes $O(n \log n)$ time.
- Inserting a new point into a balanced kd-tree takes $O(\log n)$ time.
- Removing a point from a balanced kd-tree takes $O(\log n)$ time.
- Querying an axis-parallel range in a balanced kd-tree takes $O(n^{1-1/k} + r)$ time, where r is the number of the reported points, and k is the dimension of the kd-tree.

Note however that these are worst-case bounds. Moreover, there is no guarantee that the tree will be balanced in our application. Thus the observed performance can be better or worse in practice.

KD-trees are a standard data structure in computational geometry and discussed in any textbook on the subject. You can also find the key facts at Wikipedia: `http://en.wikipedia.org/wiki/Kd_tree`

13.13 Conclusion