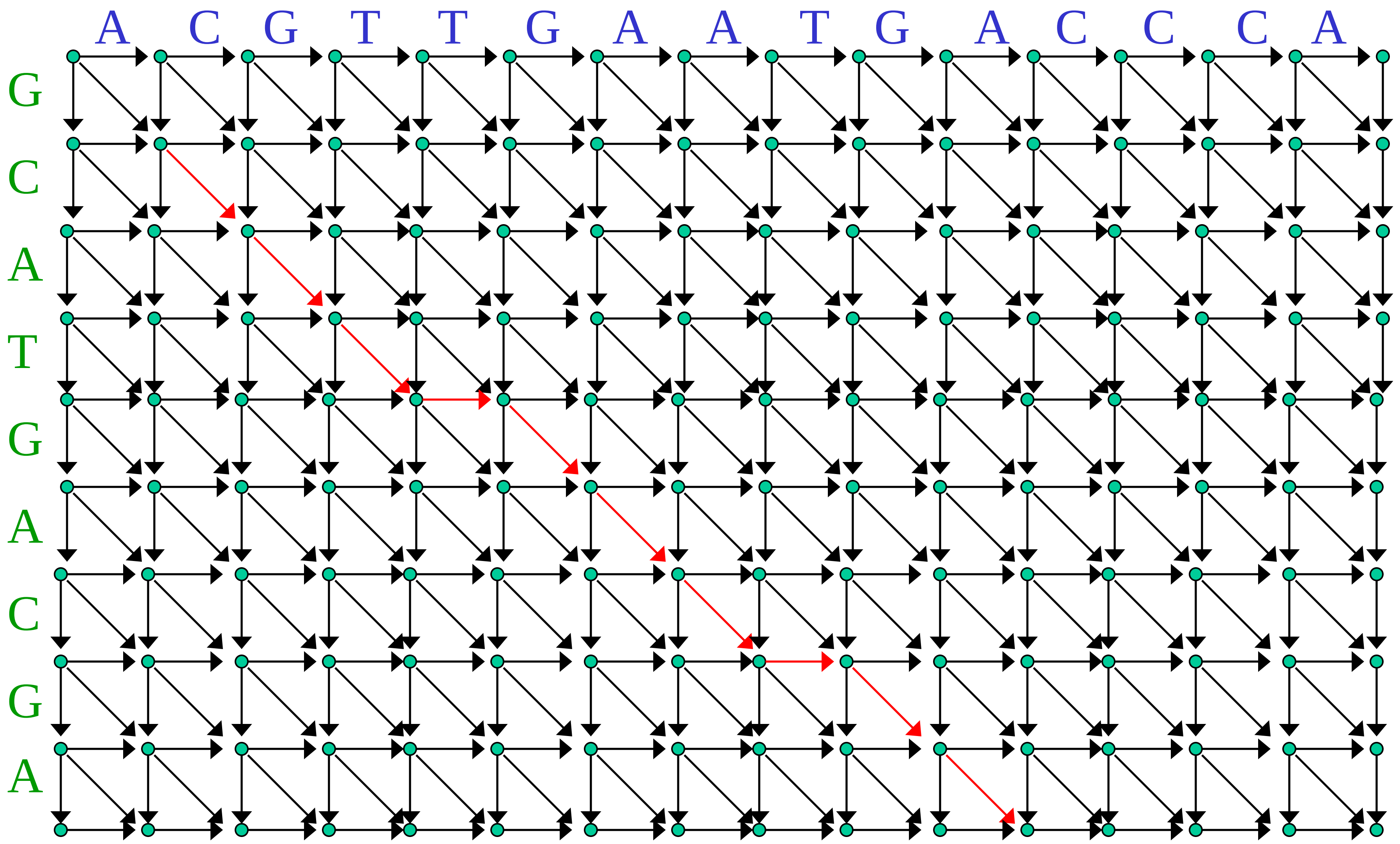
Lecture 10

- Reducing memory
  - Linear space algorithms
- Finding internal repeats
- Genome alignment



Above path corresponds to following alignment (w/ lower case letters considered unaligned):

```
aCGTTGAATGAccca
gCAT-GAC-GA
```

- To reconstruct best path, need "traceback" pointer to immediate predecessor of *v* in best path:

$$T(v) = \begin{cases} v & w(v) = 0 \\ \underset{u \in \text{parents}(v)}{\arg\max}\ (w(u) + w((u,v))) & w(v) \neq 0 \end{cases}$$

  - in preceding graph, *T*(*v*) is the *parent* on *red edge* coming into *v*
    - if more than one such edge, pick one at random;
    - if no such edge, *T*(*v*) = *v*
- Sometimes useful to record *beginning* of best path:

$$B(v) = \begin{cases} v & w(v) = 0 \\ B(T(v)) & w(v) \neq 0 \end{cases}$$

Linear Space Algorithm for Full Alignment Reconstruction

- Space complexity 10^{12} (for pairwise genome-scale alignments) is unacceptable.
- Following algorithm (based on principle of *divide-and-conquer*) trades
  - ~2-fold increase in time
    - Maybe! Will save on cache misses …

  for

  - reducing space requirement to $O(\min(M,N))$:
- [rarely used in practice however – instead one typically tries to work with "well-anchored" pieces smaller than 1 Mb]

"Forward-backward" method to find where highest-scoring path crosses midline of edit graph:

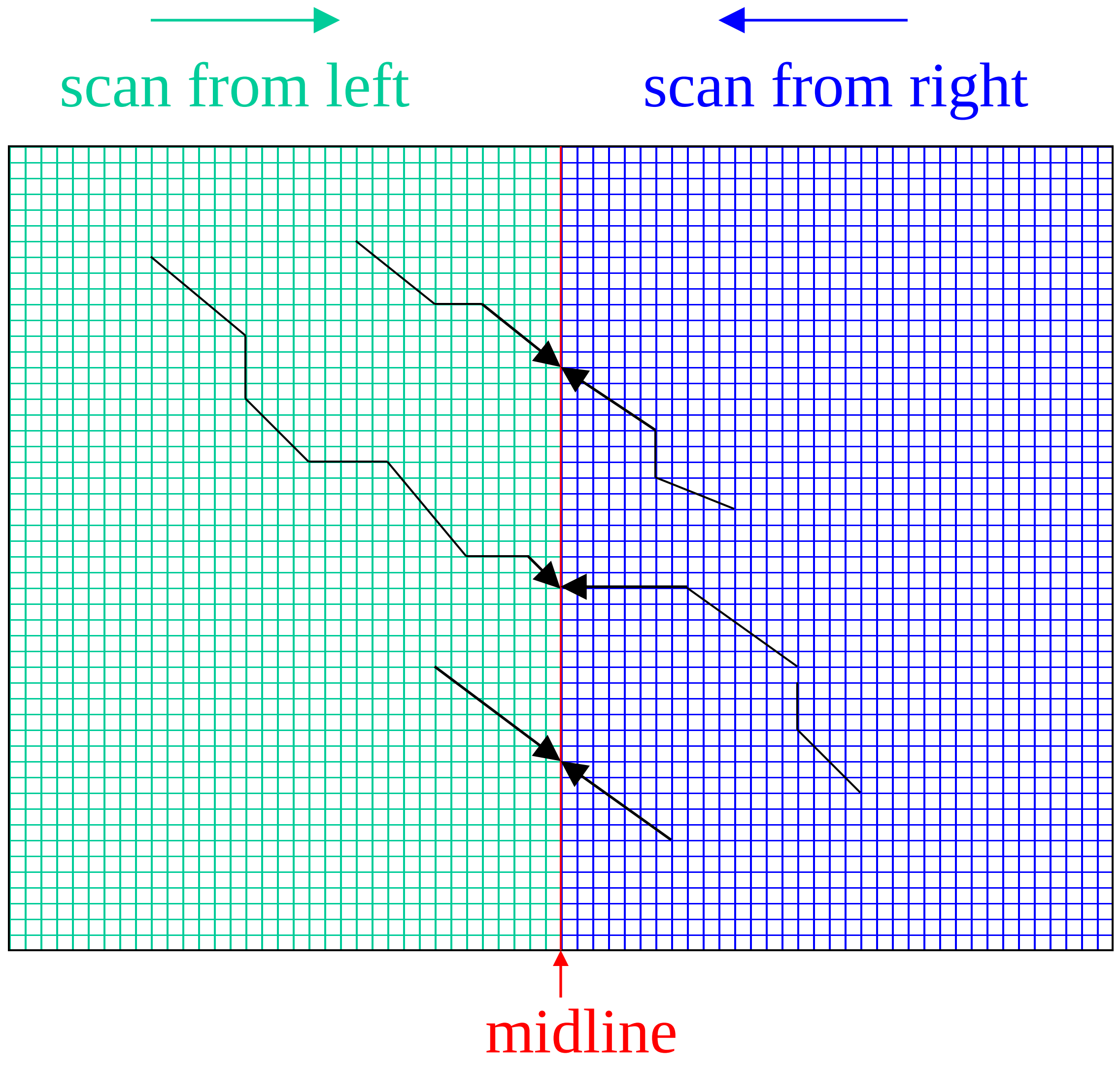
- do dynamic programming scans
  - from left bdry to midline,
  - from right bdry to midline.
- Then $\max_{v \text{ on midline}} w(v) + w'(v)$

is highest weight of any path through midline, and

$$M(v) = \arg\max_{v \text{ on midline}} w(v) + w'(v)$$

is vertex where intersects midline.

- Iterate on subgraphs.


scan from left
scan from right
midline

Inverted WDAGs

- Can “invert” any WDAG: create graph with
  - same vertices & edge weights
  - direction of each edge reversed
- inverted WDAG has same paths & path weights, but in reverse order
- inverting does not necessarily “invert” depth structure

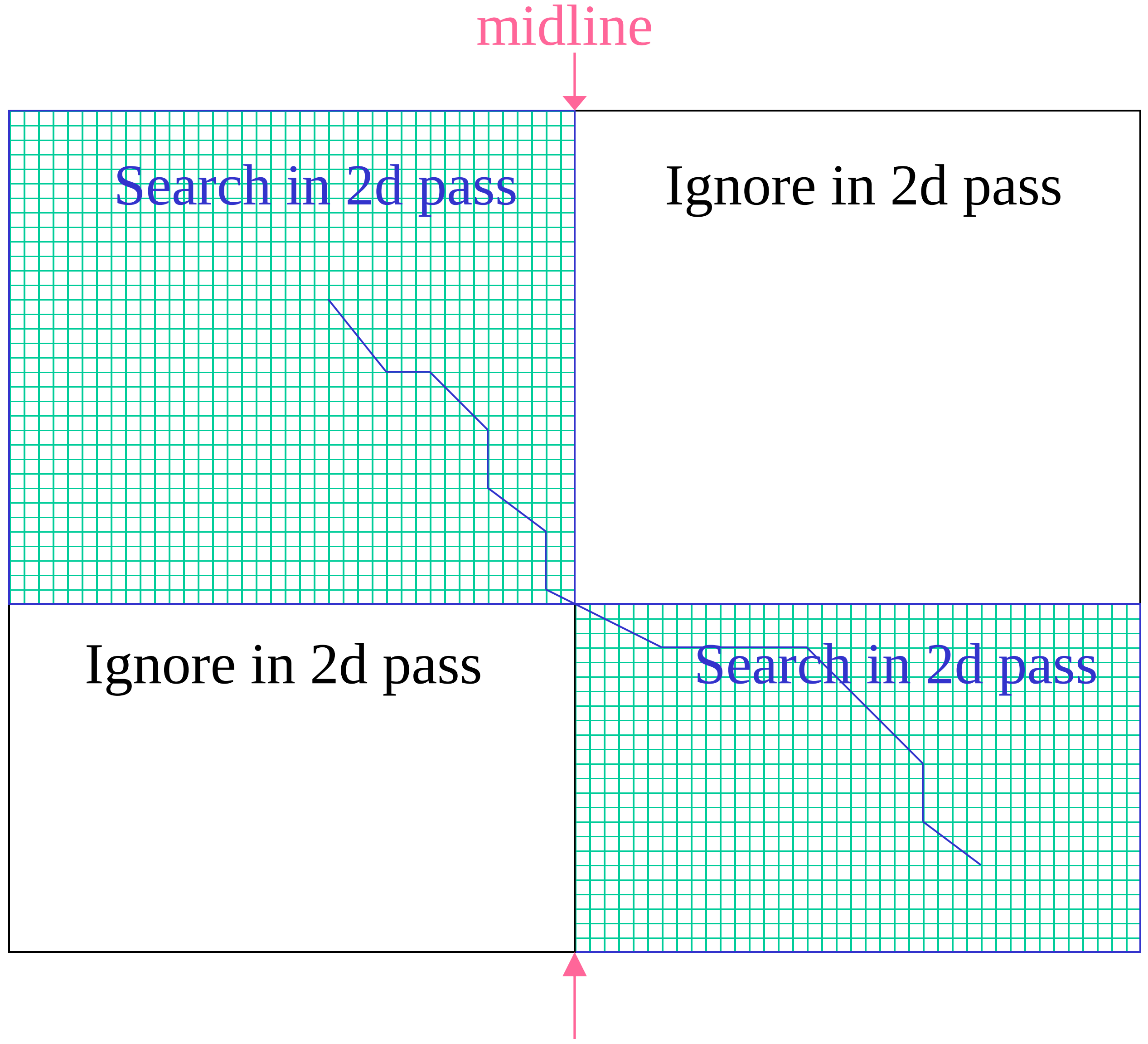
Scanning WDAG in Both Directions

- Order vertices (v_1, v_2, ..., v_n) with parents preceding children.
  - Find $w(v)$, highest weight of path (“from left”) ending at v.
- Reverse order (v_n, v_{n-1}, ..., v_1) has parents before children in *inverted* graph
  - Find $w'(v)$, highest weight of path (“from right”) ending at v.
- Then
  - (joining path from left ending at v, to reverse of path from right ending at v),

  see that $w(v) + w'(v)$ is highest weight of any path going *through* v.
- This construction will also arise later, with HMMs.

Linear space algorithm (cont'd)

- Now do 2nd pass, *only scanning part of graph where highest weight path must lie*:
  - bounded by midline, and line through $M(v)$ (or just midline, if doesn't cross it):
  - only ½ as many edges and vertices as in 1st pass
  - Now store location where crosses midline *of each subgraph*.


midline
Search in 2d pass
Ignore in 2d pass
Ignore in 2d pass
Search in 2d pass

Iterate!

- In 3rd pass, need
  - ½ # edges and vertices in 2nd pass,
  - i.e. only ¼ # in 1st pass.
- etc. until down to subgraphs consisting of single row or column
- can piece together full path from midline intersections in each pass
- Total effective search space: 1 + ½ + ¼ + … = 2,

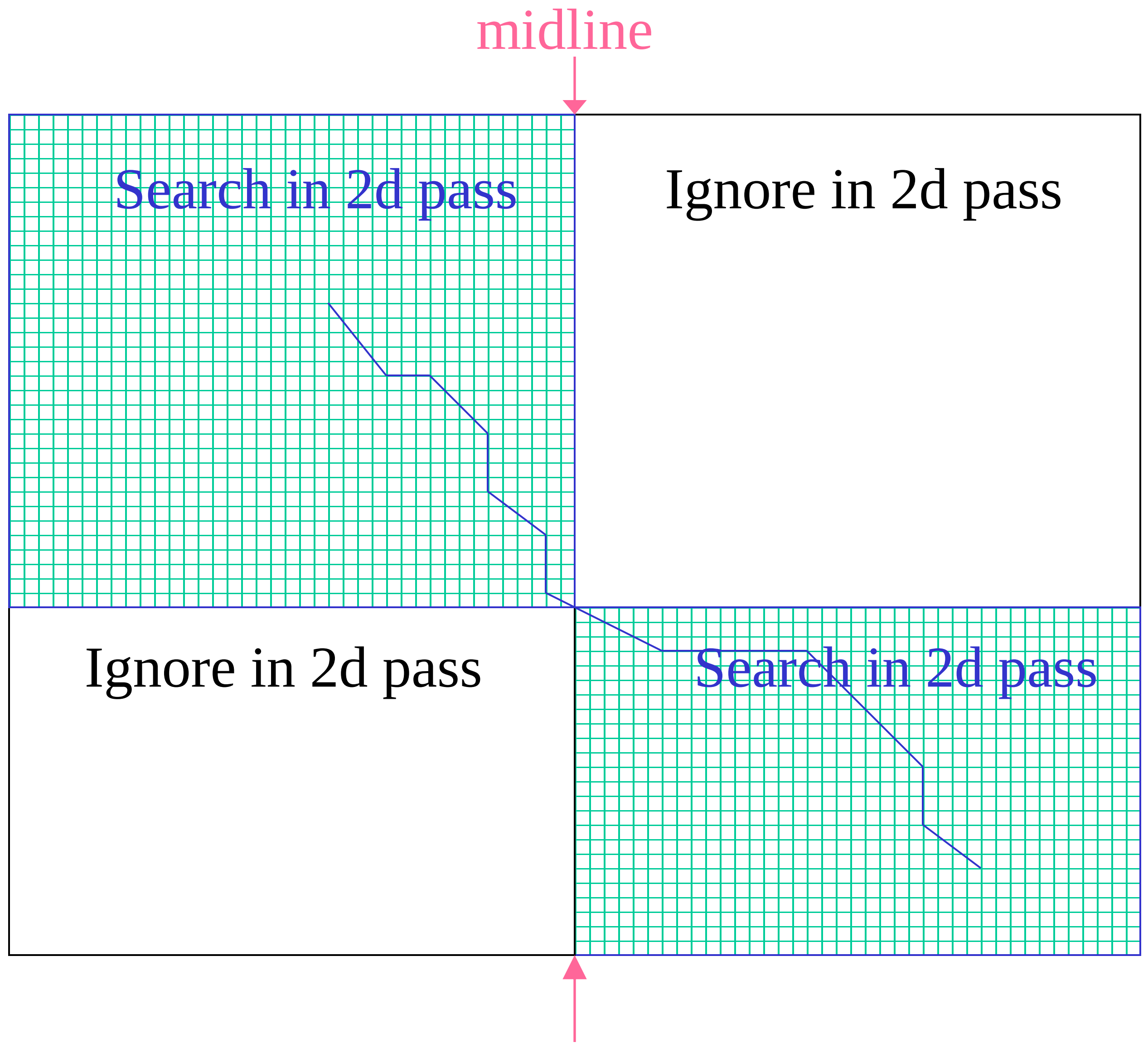
i.e. only *twice* the initial search.

Alternate method – not using inverted WDAG

- Idea: in first pass, record where highest-weight path ending at *v* crosses *midline* of graph:

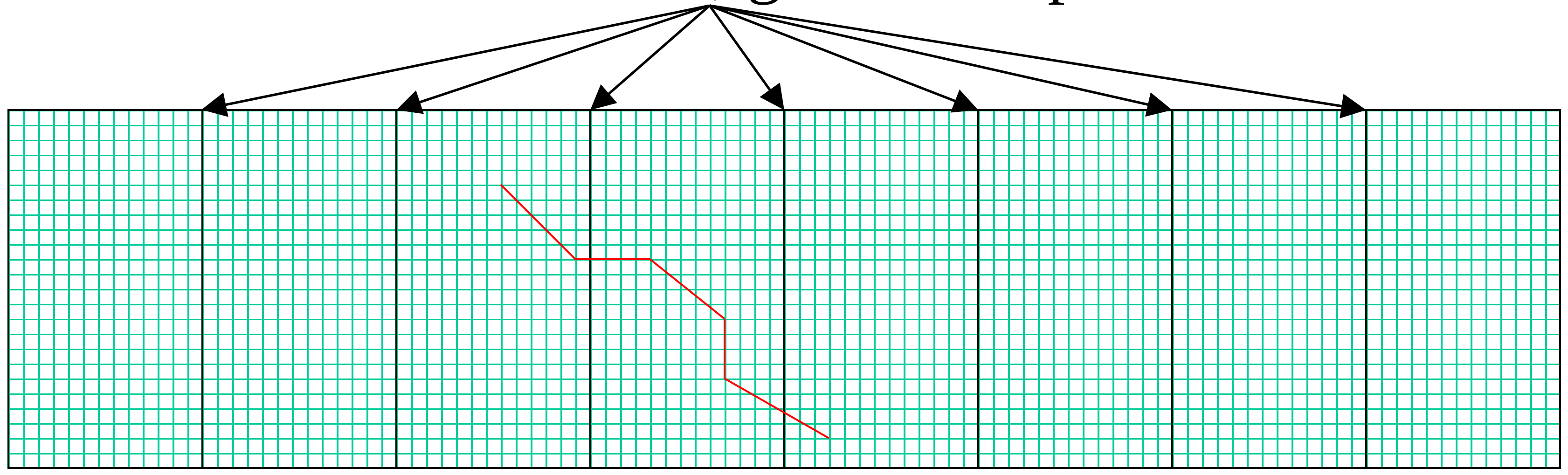
$$M(v) = \begin{cases} 0 & v \text{ lies to left of midline, or } v = T(v) \\ v & v \text{ lies on midline} \\ M(T(v)) & v \text{ lies to right of midline} \end{cases}$$

  where *T*(*v*) is parent of *v* through which best path ending at *v* passes.
- Note that (as when recording beginning of path, *B*(*v*))
  - only need retain *M*(*v*) until all children of *v* processed (or for current best *v*);
  - so requires *O*(min(*M*,*N*)) space, for appropriate processing order.
- In subsequent pass, only scan part of graph where highest weight path must lie
  - bounded by midline, and line through *M*(*v*) (or just midline, if doesn’t cross it):


midline
Search in 2d pass
Ignore in 2d pass
Ignore in 2d pass
Search in 2d pass

Linear Space – Variant Algorithms

- Can store more intermediate points (e.g. have *n* lines, record where crosses each one).
  - increases required space, but
  - decreases time (1/*n* instead of ½) for subsequent pass.
- Choose *n* to minimize time, given the space available.



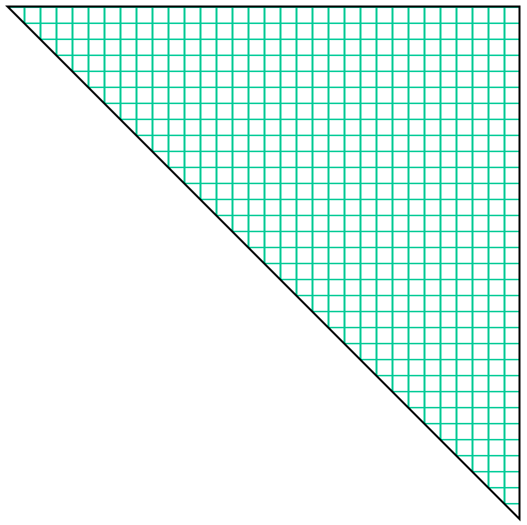
Finding (imperfect) internal repeats

- Search edit graph of *sequence against itself*
  - i.e. the same sequence labels columns and rows

above (& not including) the main diagonal:

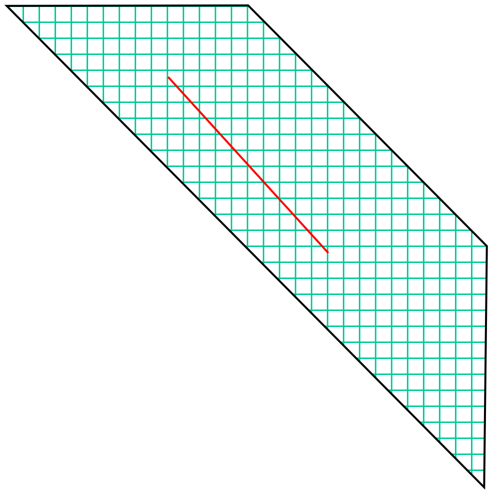
  - if include main diagonal, best path will be identity match to self
  - complexity = $O(N^2)$ where N = sequence length.

Graph for finding imperfect internal repeats:



- Find *short tandem repeats* (e.g. microsatellites, minisatellites):
  - scan a *band* just above main diagonal.
  - Complexity = $O(kN)$ where k is width of the band.
  - Manageable even for large N, if k small.

Graph for finding short tandem repeats:



```
ACACACACACACACAC
   ACACACACACACACAC
```

Genome alignment

- Challenges:
  - Size
  - Repeated sequence
    - Duplications
    - Transposable elements
    - Processed pseudogenes
  - Other segmental changes
    - Deletions
    - Inversions, translocations
  - Mutation rate variation
- Segmental changes don’t conform to edit graph framework!

Strategy

- Find (many!) word-nucleated local alignments
- Word size w: sensitivity vs specificity
  - Example: human (~3 Gb) vs mouse (~2.5 Gb)
    - ~70% identity in homologous regions
    - For each human word, expect $5 \times 10^9 / 4^w$ chance occurrences in mouse (+ rev complement)
    - Total matches: $15 \times 10^{18} / 4^w$
      - Want w *large enough* for this to be manageable
    - Prob that the *homologous* word matches: $.7^w$
      - once every $(1 / .7)^w = 1.43^w$ bp
      - Want w *small enough* to ensure ≥ 1 match within homologous regions
    - $w = 15$: $\sim 15 \times 10^9$ matches; 1 per 214 homologous bp

- Avoid high-frequency words
- Avoid nucleating in known repeats & duplications
  - But extend into them!
- Use appropriate score matrix & gap penalties!
  - Otherwise, get junk alignments or portions thereof

- Finally, identify ***chains*** of ***compatible*** local alignments
  - Ideally, catalogue the segmental changes that have occurred (duplications, transposable element insertions etc)