Mapping with Indels

Konstantinos Niktas
COM SCI 224

http://cs124project-2009.wikidot.com/kniktas-project

Short Reads

- Common and popular technique is to collect millions of short reads from a *Target Sequence*
- Faster and cheaper to do it this way
- Don’t know where the reads come from
- “Re”-sequencing maps these short reads to a *Reference Genome*. We already have this sequence.
- Mapping helps find the mutations in the target against the reference sequence.

What kinds of Mutations?

- Single base switch (mismatches/SNPs)
  - Most mappers handle this
- Indels: Insertions & Deletions
  - This is what I am focusing on
- Repeated Sequences
  - Mappers can’t handle these accurately (reads can map to multiple locations)
- Inverted Sequences
- …

Insertions & Deletions

- 15x less likely than mismatches. This is still significant enough to warrant identification
- “original”

  `ACGATCGGTACGATCTTGAC`

- Insertions:

  `ACGATCGGTA CGATCTTGAC`

  `ACGATCGGTATCGATCTTGAC`

- Deletions:

  `ACGATCGGTACGATCTTGAC`

  `ACGATCGGTAGATCTTGAC`

What I want to do

- Map reads to a reference genome and retrieve a list of operations to recreate the target sequence from the reference
- Be able to handle insertions, deletions, and mismatches. Up to…
  - *i* indels
  - *d* mismatches

Problem Formulation

- Given a set of reads from a valid target T and a valid reference G, a mapper should be able to recover the target sequence. A valid target is one having up to $i+d$ differences from the reference within any dist_{min} bp window. A valid reference genome is one where any read from all the reads only maps to the reference in one location. If invalid targets and/or reference sequences are used, the mapper(s)' behavior will not be defined. The mapper will output the target sequence or a list of operations to perform on the reference to create the target.

How do we find indels?

[Source: nature.com]

Insertions

- Insertions (of one bp) cause a shift to the right
- If we are doing mapping the naïve way (sliding along the reference)…
  - The read to the right of the insertion will map perfectly in one step
  - The read to the left of the insertion will map perfectly in the next step
  - The actual insertion may or may not match
- Finding the insertion:

…GACGCAAGTAGAGCTTTTGTAGGGCGGTGACCTGCTAG…

−−−CAAGTAGAGCT**A**TTTGTAGGGCGGTGACCT

Insertion (bold)

Insertions

- Insertions (of one bp) cause a shift to the right
- If we are doing mapping the naïve way (sliding along the reference)…
  - The read to the right of the insertion will map perfectly in one step
  - The read to the left of the insertion will map perfectly in the next step
  - The actual insertion may or may not match
- Finding the insertion:

$\texttt{...GACGCAAGTAGAGCTTTTGTAGGGCGGTGACCTGCTAG...}$

$$\texttt{----CAAGTAGAGCT}\mathbf{A}\underbrace{\texttt{TTTGTAGGGCGGTGACCT}}_{\text{Everything to the right matches perfectly}}$$

Insertions

- Insertions (of one bp) cause a shift to the right
- If we are doing mapping the naïve way (sliding along the reference)…
  - The read to the right of the insertion will map perfectly in one step
  - The read to the left of the insertion will map perfectly in the next step
  - The actual insertion may or may not match
- Finding the insertion:

```
...GACGCAAGTAGAGCTTTTGTAGGGCGGTGACCTGCTAG...
-----CAAGTAGAGCTATTTGTAGGGCGGTGACCT
```

$\underbrace{\texttt{CAAGTAGAGCT}}_{\text{Everything to the left matches perfectly}}$

Insertions

- So what can we do with this?

```
      CAAGTAGAGCTATTTGTAGGGCGGTGACCT
...GACGCAAGTAGAGCTTTTGTAGGGCGGTGACCTGCTAG...
       CAAGTAGAGCTATTTGTAGGGCGGTGACCT
```

- As we attempt to map, if we find a position x in the read that…
  - In one step, all the positions after it match
  - AND in the next step, all the positions before it match
  - We have an insertion at position x

Deletions

- A deletion (of a single bp) causes a shift to the left
- Similar to the insertion, but happens in the opposite way. If we are doing a naïve mapping algorithm…
  - In one step, the read to the left of the position of the deletion will match perfectly
  - In the next step, the read to the right of the position of the deletion will match perfectly
  - The former position of the deletion *must* match since it was there before the deletion. It matches in the second step.
- Finding the deletion:

Deleted base (underline)

…GACGCAAGTAGAGCTTTTGTAGGGCG$\underline{\text{G}}$TGACCTGCTAG…

----CAAGTAGAGCTTTTGTAGGGCG**T**GACCT

Deletion position (bold)

Deletions

- A deletion (of a single bp) causes a shift to the left
- Similar to the insertion, but happens in the opposite way. If we are doing a naïve mapping algorithm…
  - In one step, the read to the left of the position of the deletion will match perfectly
  - In the next step, the read to the right of the position of the deletion will match perfectly
  - The former position of the deletion *must* match since it was there before the deletion. It matches in the second step.
- Finding the deletion:

```
…GACGCAAGTAGAGCTTTTGTAGGGCGGTGACCTGCTAG…
-----CAAGTAGAGCTTTTGTAGGGCGTGACCT
```

Everything to the left matches perfectly

Deletions

- A deletion (of a single bp) causes a shift to the left
- Similar to the insertion, but happens in the opposite way. If we are doing a naïve mapping algorithm…
  - In one step, the read to the left of the position of the deletion will match perfectly
  - In the next step, the read to the right of the position of the deletion will match perfectly
  - The former position of the deletion *must* match since it was there before the deletion. It matches in the second step.
- Finding the deletion:

…GACGCAAGTAGAGCTTTTGTAGGGCG$\underline{\text{G}}$TGACCTGCTAG…

––––––CAAGTAGAGCTTTTGTAGGGCG$\underbrace{\textbf{T}\text{GACCT}}$

Everything to the right matches perfectly

Deletions

- As we attempt to map, if we find a position x in the read that…
  - In one step, all the positions before it match
  - AND in the next step, all the positions after it match
  - AND in that same step, that position matches
  - We have a deletion at position x

Written Mathematically

- Keep a state M_i for each position *i* in the read of length |r|
- $M_i = (M_i^b, M_i^o, M_i^a)$. It stores the number of matches before, on, and after i
- $M_{i,j} = M_i$ at step j
- We have an insertion at position x if:
  - $M_{i,j-1}^a = |r|-x$ (AND) $M_{i,j}^b = x-1$
- We have a deletion at position x if:
  - $M_{i,j-1}^b = x-1$ (AND) $M_{i,j}^o = 1$ (AND) $M_{i,j}^a = |r|-x$

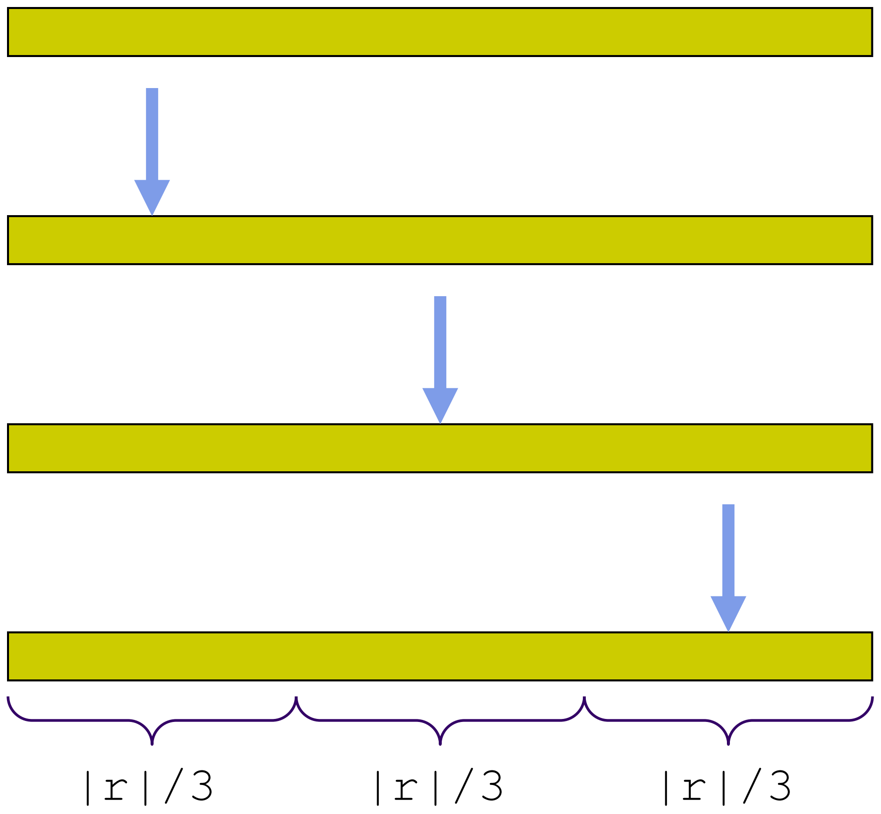
$$M^a_{x,j-1} + M^b_{x,j} = |r| - 1$$
$$M^b_{x,j-1} + M^o_{x,j} + M^a_{x,j} = |r|$$

Naïve Algorithm

- Can handle up to 1 indel
- This is accurate if up to 1 indel exists in a window (in the target) the length of the reads: Every |r| length substring of the target, has up to 1 indel.
- Algorithm for each read:
  `Slide the read along the reference. At each step, count the number of matches before, on, and after every position in the read. If the read matches perfectly, we found the mapping position. If not, for each position i in the read, check the state` $M_{i,j-1}$ `and` $M_{i,j}$ `to see if they meet the mathematical description of an indel. If an indel is found, return the operation that will make the reference have that mutation.`
- Notice that we only need to have the state M for the current step and the previous step if we are looking for only one indel.
- Slow!!! At the very worst-case for a read, we would have to compare the read against |G|-|r| positions in the reference G. In addition, not only do we do |r| comparisons at each position, we need to set and check the state M for each position in the read. The state M contains 2*3*|r| values in the one indel case.

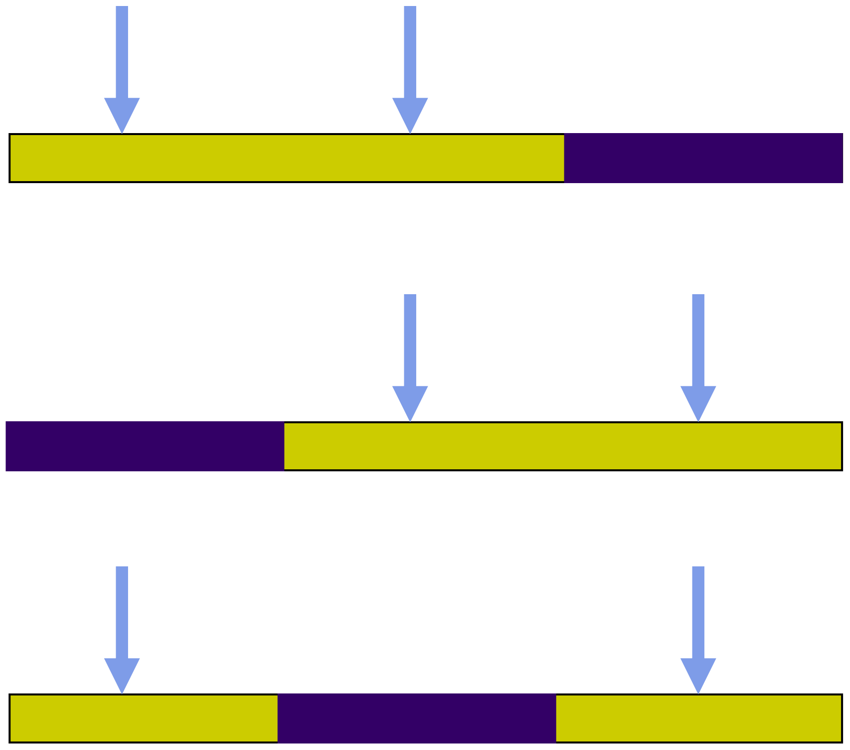
Better Algorithm

- Similar to what was talked about in class, create an index of the reference genome with key length |r|/3.
- Create 3 "subreads" out of the reads.
- For 1 indel, only 1 of those subreads will contain the indel (if at all)



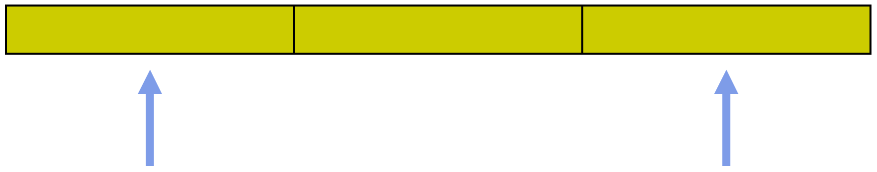
Better Algorithm

- If the read has no mutations, there will be 3 positions of those subreads in the index that will be adjacent to each other.
- If there is an indel:
  - Find two positions in the reference (in the index) for two of the subreads that are properly spaced (i.e. if we are using the first two subreads, the subreads should be adjacent. If we are using the first and third, the subreads should have a space the length of a subread between them.)
  - Take the remaining subread and compare it against the position remaining (i.e. if the second subread is remaining, compare it to the space between the first and third). Slide it left and right storing the M states and then comparing them. If an indel is detected, return the operation required. If an indel is not detected, then that is not where the read maps. We need to test different locations and/or use two other subreads.

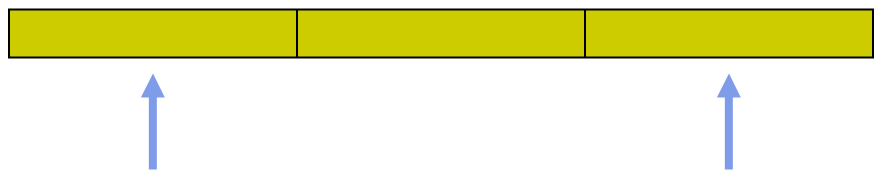


Better Algorithm

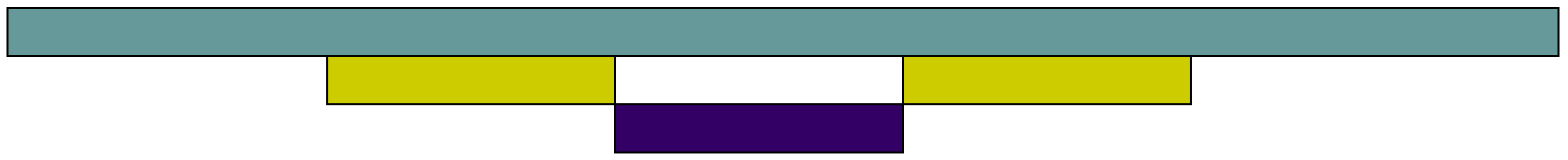
Pick two of the subreads to find position in the reference using the index.



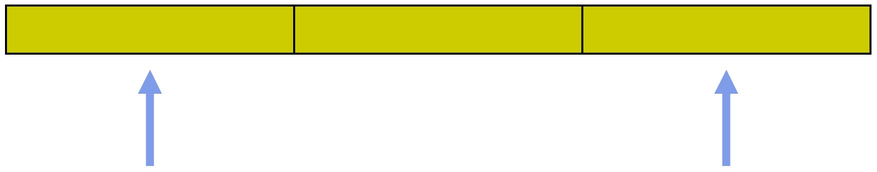
Better Algorithm



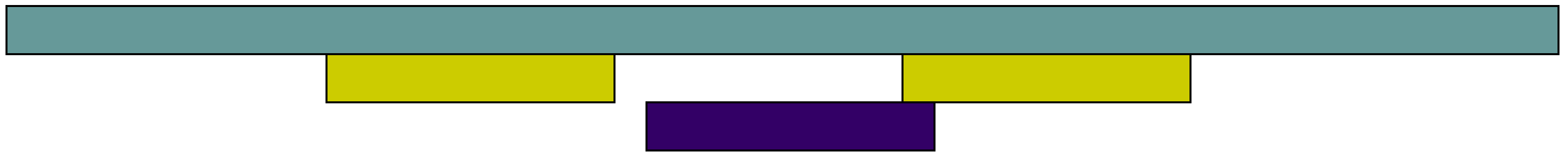
If we find a position that fits correctly, slide the remaining subread



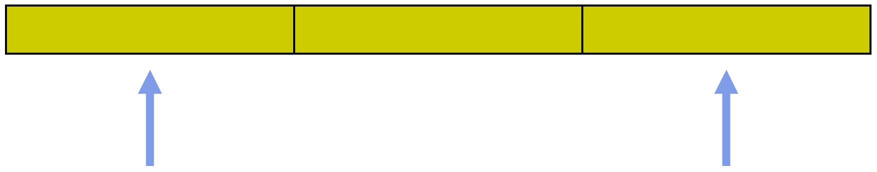
Better Algorithm



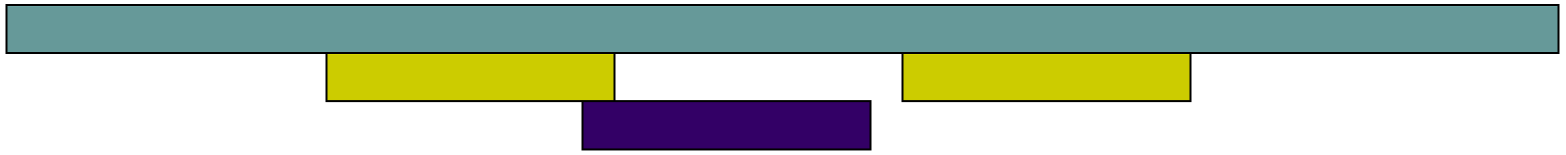
To the right



Better Algorithm



And to the left



Using a similar technique to the naïve algorithm, we can look for indels by storing state M. This speeds up detection since we do not need to comparisons of entire read lengths. We only need to do matching of the remaining subread if we find indices that work for the two “guiding” subreads.

Results…

- Naïve algorithm written and works
- Obviously very slow
- Sadly, I haven't been able to write the faster algorithm or an algorithm to handle mismatches and more than one indel because I am doing this as a master's project and I'm writing the paper (the most time consuming part).
- I will continue to work on this after this presentation and I will post all the code and results on the wiki page: http://cs124project-2009.wikidot.com/kniktas-project

Thank You