Intron-retention 탐색을 위한 효율적인 알고리즘 개발 및 연구

이 름 정민규　　지도교수 조다정

연구배경

본 연구는 mis-splicing mutation들 중 하나인 intron retention에 관하여 계산 이론 관점으로 문제를 정의하고, 효율적인 intron retention 탐색 알고리즘을 제시한다. 먼저 pre-mRNA에서 splicing 중에 일어나는 mis-splicing은 exon과 intron이 존재하는 단백질에 중요 유전 정보를 가지고 있는 exon을 제외한 intron을 splice하는 것에 실패한 것을 말한다. 이는 남아야 하는 exon이 잘리거나(exon skipping) 혹은 제거 되야 하는 intron이 남아있는 경우(intron retention), 등을 말한다. 이러한 mutation들이 암의 발생 및 진행에 영향을 끼친다는 최근 연구들이 있다.

본 연구에서는 기존에 수행되었던 연구들과 과제들을 참고하며 연구방향성을 정한다. 먼저 생물학 관점에서 접근했던 방식을 살펴보며 데이터를 구하고, 기존의 해결책들을 살펴본다. 이후, 계산 이론 관점에서 string insertion, deletion에 대한 연구를 살펴보고 어떻게 이용하며 효율적인 알고리즘을 구상해본다.

연구진행과정

1. 문제 정의 및 모델 정의, 테스트 데이터 생성 규칙

먼저 "기존 pre-mRNA string이 있을 때, splicing 된 mRNA에서 intron retention을 찾아내는 것"으로 문제를 정의했다. 이는 mis-splicing에 종류가 다양한데, 모든 경우를 다 살펴보는 것이 아니라 하나에 먼저 초점을 맞추기 위함이다. 이를 실험해보기 위해 몇가지 가정을 한다. 또한 가정에 따라 다음 연구를 위하여 간단한 모델을 구축한다. **0)** read는 A, U, G, C로만 나타낸다. **1)** 전체 sequence가 아닌 [exon-intron-exon]의 형태만 고려한다. 이는 본래 문제의 일부를 먼저 해결한 후, 확장하기 위함이다. **2)** intron retention들 중 intron의 5' splice site나, 3' splice site쪽이 남는 경우만 고려한다. **3)** splicing의 진행은 spliceosome의 major type만을 고려한다. 즉, intron은 GU~AG의 꼴로만 존재한다고 가정한다. **4)** intron 기준의 양쪽 exon은 genetic code에 따른 read 조합들 중 최소 하나를 가진다. 따라서 양쪽 exon 모두가 존재하는 경우를 본다. [이하 테스트 데이터 생성] **5)** 양쪽 exon은 genetic code read 조합들 중 임의로 18 조합을 선택하고, 이를 합이 18이 되도록 나눈다. (1-17, 13-5 등) **6)** intron은 3)에 따라 왼쪽 끝에 GU, 오른쪽 끝에 AG, 가운데는 임의로 길이가 44가 되도록 read를 선택한다. 따라서 pre-mRNA는 최대 18 * 3 + 48 = 102의 길이를 가지는 string이 된다. **7)** spliced mRNA는 50% 확률로 intron 전체를 없애거나 2)에 따라 [GU-], [-AG] 둘 중 한 부분을 선택하여 남긴다. 이는 정의한 문제에 따라 설계된 알고리즘이 intron retention을 얼마나 잘 판단하는지를 보기 위함이다.

2. 문제 해결 아이디어

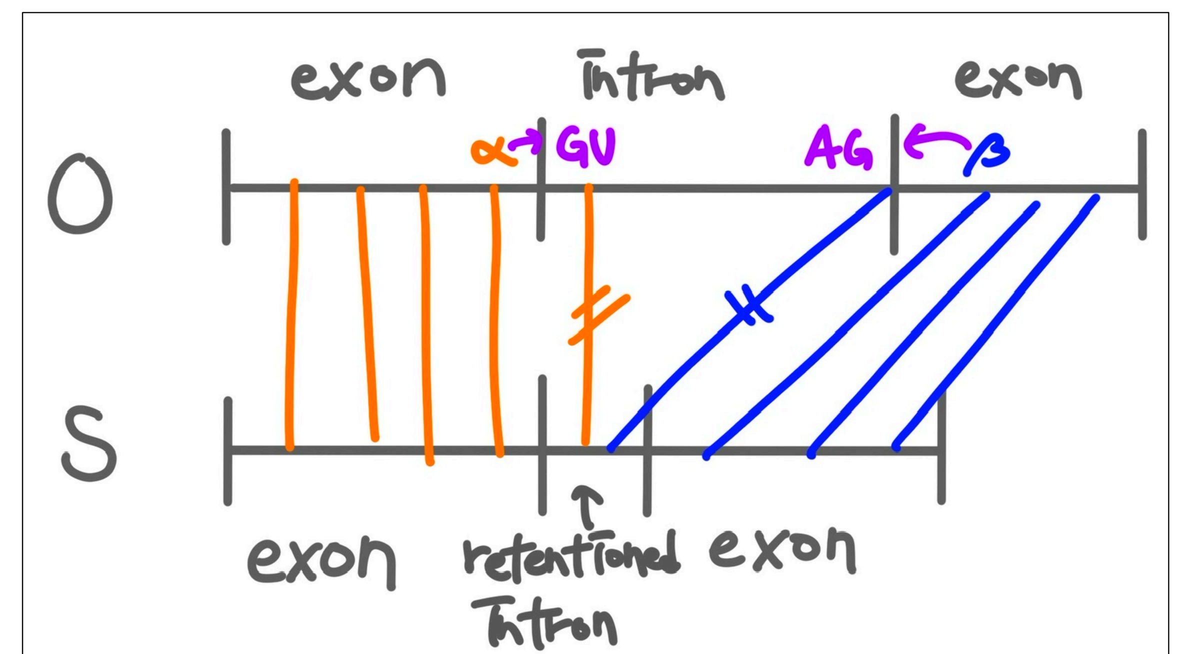
pre-mRNA와 mRNA의 모든 문자열을 확인하며 찾아내는 것은 시간복잡도가 매우 크다. 따라서 다중 문자열 패턴 매칭 알고리즘인 아호코라식 알고리즘을 이용한다. 모델의 정의에 따라, intron은 GU~AG의 형태로만 존재할 것이다. 따라서 아호코라식 알고리즘을 통하여 GU와 AG를 고려하도록 설계해보았다.

3. 알고리즘 설명 및 구현

사용한 언어는 python으로 아호코라식 알고리즘을 먼저 구현한다. 이후 테스트 데이터를 생성한다. Pre-mRNA는 original string(o), spliced pre-mRNA(mRNA)는 spliced string(s)로 나타낸다. 그리고 설명을 위하여 g=GU, a=AG라고 하고 g와 a를 word라고 칭한다. 그리고 o'과 s'은 o', s'은 설명을 위하여 o와 s를 g와 a로 표현한 string을 말한다. (g와 a들 사이에는 임의의 read가 존재). 예로 o가 'AAACAACCGAGCGGAGUUGUC'이라면, GU와 AG를 표시했을 때 'AAACAACCG(AG)CGG(A{G)U}U{GU}C'로 나타낼 수 있으며, 이 때 o'은 'aagg'가 된다. (AG는 소괄호로, GU는 중괄호로 표현) 이하는 알고리즘 설명이다.

1) o와 s의 GU, AG 위치를 aho-corasick으로 찾고, dictionary in list 형태([{"index":39, "word":"ag"}, {"index":44, "word":"gu"}])로 만든다. **2)** 생성된 자료구조(이하 각각 o's, s's)는 원래 string의 순서를 따르므로, index' (o'과 s'의 index)를 이용하여 접근을 했을 때에도 원래 string의 순서를 따른다.
3) o's와 s's의 각 원소를 앞에서부터 비교한다. 이 때, 먼저 word가 같은지를 본다. **3-1)** 만약 o's의 word와 s's의 word가 같다면, 원래 string o, s에서의 index로 참조하고, 각 index에서 해당 word의 이전 read와 다음 read를 비교한다. 즉, word의 앞, 뒤 read를 한 칸 씩 window로 비교하는 것이며, 두 word가 정말로 같은지를 확인 하는 것이다. **3-2)** 만약 word가 다르다면 그 index'의 이전 index'을 alpha라 한다. 혹은 word는 같으나 해당 word의 이전 read, 다음 read가 다르다면 (예를 들어 A(GU)U와 U(GU)A인 경우) index'의 이전 index'을 alpha라 한다. **4)** o와 s를 reverse한 string인 oR, sR을 만들고, 1~3-2를 실행한다. 단, 이때 구해진 index'는 alpha가 아닌 beta라 한다. **5)** o에서 alpha보다 큰 index' 중 가장 먼저 등장 하는 g(이하 g')와 beta보다 작은 index' 중 가장 먼저 등장 하는 a(이하 a')의 index를 구한다. 이는 o에서 intron을 찾아내는 작업이다. **6)** o에서 g'의 index(GU 포함)부터 시작하여 a'의 index + 2 (AG의 길이) 범위에 해당하는 길이를 구한다. 이를 cil(candidate intron length)이라고 한다. **7)** o의 길이에서 cil을 뺀 값과 s의 길이가 다르다면 intron retention이 일어난 것이다.

위 알고리즘을 살펴보면 o에서 intron을 찾고, 이를 지운다음 s와 길이를 비교하는 것이다. 따라서 GU-AG rule (major type에서 intron의 대부분은 GU~AG의 형태라는 것)을 적용하여 가장 일치하는 부분(exon)의 양 끝을 찾고, 그 사이에 등장하는 GU~AG가 intron임을 이용하였다. 아래 그림은 위 과정에서의 alpha와 beta를 설명한다.



결과 및 분석

위 알고리즘을 통하여 테스트 데이터를 100000개 생성하였을 때, intron retention(mis-spliced)가 일어난 비율을 맞추는 정확도는 아래 그림의 출력인 74.8% 정도이다.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL   JUPYTER

origin string: CACGUCAAAAGGUUAUAGCCCGAAGACGUCAACUUA||GUGUGGGUUCGGCGUCGACCAAGGGCAUCUGUCUUCGCUCCAAUGAAG||UUGGUACACCUAGCGAGU
splice string: CACGUCAAAAGGUUAUAGCCCGAAGACGUCAACUUA||GUGUGGGUUCGGCGUCGACCAAGGGCAUCUGUCUUCGCUCCAAUGAAG||UUGGUACACCUAGCGAGU
origin: GUGUGGGUUCGGCGUCGACCAAGGGCAUCUGUCUUCGCUCCAAUGAAG
alpha: 27 5 False
beta: 82 3 False
--------------------------------------------------------------------------------
[{'index': 3, 'word': 'gu'}, {'index': 9, 'word': 'ag'}, {'index': 11, 'word': 'gu'}, {'index': 16, 'word': 'ag'}, {'index': 23, 'word': 'ag'},
 {'index': 35, 'word': 'ag'}, {'index': 36, 'word': 'gu'}, {'index': 38, 'word': 'gu'}, {'index': 42, 'word': 'gu'}, {'index': 49, 'word': 'gu'}
}, {'index': 66, 'word': 'gu'}, {'index': 82, 'word': 'ag'}, {'index': 83, 'word': 'gu'}, {'index': 87, 'word': 'gu'}, {'index': 95, 'word': 'ag
g'}, {'index': 100, 'word': 'gu'}]
--------------------------------------------------------------------------------
[{'index': 3, 'word': 'gu'}, {'index': 9, 'word': 'ag'}, {'index': 11, 'word': 'gu'}, {'index': 16, 'word': 'ag'}, {'index': 23, 'word': 'ag'},
 {'index': 39, 'word': 'gu'}, {'index': 47, 'word': 'ag'}, {'index': 51, 'word': 'ag'}, {'index': 52, 'word': 'gu'}]
--------------------------------------------------------------------------------
aprime: 7
bprime: 5
pridic: GUGUGGGUUCGGCGUCGACCAAGGGCAUCUGUCUUCGCUCCAAUGAAG
0.748
```

정확도가 100%가 아닌 이유는 exon과 intron사이에 A||GU와 같은(A는 exon의 끝, GU는 intron의 시작) 경우가 있을 때, 등의 예외 상황이 발생하기 때문이다. 또한, 정확한 구현이 되었는지 검증을 하지 못했는데, 예외 상황들을 함께 고려하다 보니 정확하게 intron을 찾기에는 부족한 정보들이 있었다. 이는 ESS(exon skipping 억제제)와 같은 정보들을 함께 이용해야 했지만 그러하지 못하고 GU-AG rule만을 이용했음을 말한다. 또한 생성된 테스트 데이터가 정말로 올바른지에 대한 검증도 필요할 것이다. 따라서 다음 연구를 진행할 때에는 더욱 많은 조사를 하여 다양한 정보를 가지고 정확도를 높이는 것을 목표로 한다.