2024.11.20

염기 서열 생성, 리드 테이블 생성 알고리즘 구축

사용 모듈: numpy, pandas

염기 서열 생성

=> 기존 염기서열의 생성 방식은 A, T, G, C를 캐릭터의 형태로 저장

=> 염기서열 자체 데이터 크기가 너무 커짐

=> 4진법으로 변환하여, bin 파일로 저장하는 것으로 수정

염기서열 생성 알고리즘

|  |
| --- |
| def **generate\_random\_sequence\_advanced**(*n*, *filename*=None):  *# numpy의 고급 난수 생성기 설정*  rng = np.random.default\_rng()    *# 0(A), 1(T), 2(C), 3(G)로 직접 숫자 생성*  sequence = rng.integers(0, 4, *size*=*n*, *dtype*=np.uint8)    *if* *filename* is None:  *filename* = f'dna\_{*n*}.bin'    *# 바이너리 모드로 파일 저장*  *try*:  *with* open(*filename*, 'wb') *as* f:  sequence.tofile(f)  print(f"시퀀스가 {*filename*}에 성공적으로 저장되었습니다.")  *except* Exception *as* e:  print(f"파일 저장 중 오류 발생: {e}")    *return* sequence  def **read\_sequence**(*filename*):  *# 저장된 시퀀스를 읽어오는 함수*  *try*:  sequence = np.fromfile(*filename*, *dtype*=np.uint8)  print(f"파일에서 {len(sequence)} 개의 염기를 읽었습니다.")  *return* sequence  *except* Exception *as* e:  print(f"파일 읽기 중 오류 발생: {e}")  *return* None  def **convert\_to\_bases**(*sequence*):  *# 필요한 경우 숫자를 염기로 변환*  base\_map = {0: 'A', 1: 'T', 2: 'C', 3: 'G'}  *return* ''.join(base\_map[x] *for* x *in* *sequence*) |

테스트

|  |
| --- |
|  |

리드 테이블 알고리즘 구축"""

* 주어진 염기서열과 coverage를 고려하여 리드 테이블을 생성
* 각 케이스는 DBG에 최적화된 리드 길이와 리드 개수를 계산하여 생성

Parameters:

- file\_path (str): 원본 염기서열이 저장된 파일 경로

- output\_csv (str): 결과를 저장할 CSV 파일 경로 (선택사항)

- desired\_coverage (int): 염기서열 전체에 대한 목표 coverage. 기본값은 30

Returns:

- pd.DataFrame: 각 케이스의 리드 길이와 개수를 담은 데이터프레임

|  |
| --- |
| def **generate\_reads\_with\_coverage**(*file\_path*, *output\_csv*=None, *desired\_coverage*=30):  """  주어진 염기서열과 coverage를 고려하여 리드 테이블을 생성합니다.  각 케이스는 DBG에 최적화된 리드 길이와 리드 개수를 계산하여 생성합니다.    Parameters:  - file\_path (str): 원본 염기서열이 저장된 파일 경로  - output\_csv (str): 결과를 저장할 CSV 파일 경로 (선택사항)  - desired\_coverage (int): 염기서열 전체에 대한 목표 coverage. 기본값은 30  Returns:  - pd.DataFrame: 각 케이스의 리드 길이와 개수를 담은 데이터프레임  """    *# 시퀀스 읽기*  *try*:  sequence = np.fromfile(*file\_path*, *dtype*=np.uint8)  genome\_length = len(sequence)  print(f"시퀀스 길이: {genome\_length}")  *except* Exception *as* e:  print(f"시퀀스 파일 읽기 오류: {str(e)}")  *return* None    read\_length\_arr = [30,50,75,100]    cases = []  *for* i, read\_length *in* enumerate(read\_length\_arr):  k\_mer\_length = (2 \* read\_length) // 3  *if* k\_mer\_length % 2 == 0:  k\_mer\_length -= 1  k\_mer\_length = max(15, min(31, k\_mer\_length))    read\_count = int((*desired\_coverage* \* genome\_length) / read\_length)  read\_count = max(2, read\_count)    print(f"\n{'='\*50}")  print(f"Case {i+1} 상세 정보:")  print(f"리드 길이: {read\_length}")  print(f"목표 리드 개수: {read\_count}")  print(f"k-mer 길이: {k\_mer\_length}")  print(f"전체 시퀀스 길이: {genome\_length}")    reads = []  max\_start\_pos = genome\_length - read\_length    *if* max\_start\_pos < 0:  print("경고: 리드 길이가 시퀀스 길이보다 큽니다!")  reads.append(sequence[:read\_length])  *else*:  *# 균등 간격 계산*  step\_size = max(1, max\_start\_pos // (read\_count - 1))  print(f"리드 간 간격: {step\_size}")    *# 리드 생성 및 검증*  *for* start\_pos *in* range(0, max\_start\_pos + 1, step\_size):  *if* len(reads) < read\_count:  read = sequence[start\_pos:start\_pos + read\_length]  *if* len(read) == read\_length: *# 리드 길이 검증*  reads.append(read)  *else*:  print(f"경고: 잘못된 리드 길이 - 위치: {start\_pos}, 길이: {len(read)}")    *# 부족한 리드 추가*  remaining\_reads = read\_count - len(reads)  *if* remaining\_reads > 0:  print(f"추가로 생성할 리드 수: {remaining\_reads}")  *for* \_ *in* range(remaining\_reads):  start\_pos = np.random.randint(0, max\_start\_pos + 1)  read = sequence[start\_pos:start\_pos + read\_length]  *if* len(read) == read\_length:  reads.append(read)  *else*:  print(f"경고: 랜덤 리드 생성 실패 - 위치: {start\_pos}, 길이: {len(read)}")    *# 최종 검증*  print(f"\n최종 결과:")  print(f"생성된 총 리드 수: {len(reads)}")  print(f"첫 번째 리드 길이: {len(reads[0])}")  print(f"마지막 리드 길이: {len(reads[-1])}")    *# 리드 길이 분포 확인*  lengths = [len(read) *for* read *in* reads]  *if* len(set(lengths)) > 1:  print("경고: 리드 길이가 일정하지 않습니다!")  print(f"리드 길이 분포: {set(lengths)}")    *# 첫 번째와 마지막 리드의 실제 데이터 확인*  print("\n리드 데이터 샘플:")  print(f"첫 번째 리드: {convert\_to\_bases(reads[0])[:30]}...")  print(f"마지막 리드: ...{convert\_to\_bases(reads[-1])[-30:]}")    cases.append({  'Case': f'Case {read\_count}',  'Read Length': read\_length,  'Read Count': read\_count,  'Coverage': *desired\_coverage*,  'k-mer Length': k\_mer\_length,  'Reads': reads  })    *# 데이터프레임으로 결과 반환*  df = pd.DataFrame(cases)    *# CSV 파일로 저장*  *if* *output\_csv*:  *try*:  df.to\_csv(*output\_csv*, *index*=False)  print(f"\n리드 테이블이 성공적으로 {*output\_csv*}에 저장되었습니다.")  *except* Exception *as* e:  print(f"\nCSV 파일 저장 중 오류 발생: {str(e)}")    *return* df |

테스트: 위에서 만든 test\_dna.bin으로 테스트 진행

|  |
| --- |
|  |

* 케이스별로 마지막 리드의 마지막 염기가 통일되지 않음
* 모든 염기서열을 커버하지 못하고 있음을 확인
* 추후 수정, DBG 알고리즘 구현 우선시

DBG 알고리즘 구현

|  |
| --- |
| *# De Bruijn Graph를 이용한 시퀀스 재구성*  def **construct\_debruijn\_graph**(*reads*, *k*):  """  리드들로부터 De Bruijn Graph를 구성합니다.  """  edges = {}  *for* read *in* *reads*:  *# 숫자 배열을 문자열로 변환하여 k-mer 생성*  read\_str = ''.join(map(str, read)) *# 수정된 부분*    *# 각 리드를 k-mer로 분할*  *for* i *in* range(len(read\_str) - *k* + 1):  kmer = read\_str[i:i+*k*]  next\_kmer = read\_str[i+1:i+*k*+1]    *if* kmer not in edges:  edges[kmer] = {}    *if* next\_kmer in edges[kmer]:  edges[kmer][next\_kmer] += 1  *else*:  edges[kmer][next\_kmer] = 1    *return* edges  def **find\_eulerian\_path**(*graph*):  """  De Bruijn Graph에서 오일러 경로를 찾습니다.  """  *# 시작 노드 찾기*  start = list(*graph*.keys())[0]    path = []  stack = [start]    *while* stack:  current = stack[-1]    *if* current in *graph* and *graph*[current]:  *# 다음 방문할 노드 선택*  next\_node = list(*graph*[current].keys())[0]    *# 엣지 제거*  *graph*[current][next\_node] -= 1  *if* *graph*[current][next\_node] == 0:  *del* *graph*[current][next\_node]  *if* not *graph*[current]:  *del* *graph*[current]    stack.append(next\_node)  *else*:  path.append(stack.pop())    *return* path[::-1]  def **reconstruct\_sequence**(*reads*, *k*):  """  De Bruijn Graph를 이용하여 원본 시퀀스를 재구성합니다.  """  *# De Bruijn Graph 구성*  graph = construct\_debruijn\_graph(*reads*, *k*)    *# 그래프가 비어있는지 확인*  *if* not graph:  print("경고: 그래프가 비어있습니다!")  *return* ""    *# 오일러 경로 찾기*  path = find\_eulerian\_path(graph)    *# 경로를 시퀀스로 변환*  sequence = path[0][:-1] *# 첫 번째 k-mer의 마지막 문자를 제외한 모든 문자*  *for* node *in* path:  sequence += node[-1] *# 각 k-mer의 마지막 문자만 추가*    *# 숫자 시퀀스를 염기 시퀀스로 변환*  base\_map = {0: 'A', 1: 'T', 2: 'C', 3: 'G'}  result = ''.join(base\_map[int(c)] *for* c *in* sequence)    *return* result |

테스트

|  |
| --- |
| filename = "test\_dna.bin"  *# 길이 10,000의 염기서열 생성 및 파일 저장*  original\_sequence = generate\_random\_sequence\_advanced(10\*\*4, filename)  *# 리드 테이블 생성*  reads\_df = generate\_reads\_with\_coverage(  *file\_path*=filename,  *output\_csv*="reads\_table.csv",  *desired\_coverage*=30  )  *# 첫 번째 케이스의 리드들을 이용하여 시퀀스 재구성*  k = reads\_df.iloc[0]['k-mer Length']  reads = reads\_df.iloc[0]['Reads']  print(f"\n재구성 시작:")  print(f"사용할 k-mer 길이: {k}")  print(f"리드 개수: {len(reads)}")  print(f"첫 번째 리드 예시: {convert\_to\_bases(reads[0])}")  reconstructed\_sequence = reconstruct\_sequence(reads, k)  *# 결과 비교*  original\_bases = convert\_to\_bases(original\_sequence)  print(f"\n결과 비교:")  print(f"원본 시퀀스 길이: {len(original\_bases)}")  print(f"재구성된 시퀀스 길이: {len(reconstructed\_sequence)}")  print(f"\n원본 시퀀스 (처음 100bp): {original\_bases[:100]}")  print(f"재구성 시퀀스 (처음 100bp): {reconstructed\_sequence[:100]}")  *# 일치율 계산*  min\_len = min(len(original\_bases), len(reconstructed\_sequence))  matches = sum(1 *for* i *in* range(min\_len) *if* original\_bases[i] == reconstructed\_sequence[i])  similarity = (matches / min\_len) \* 100  print(f"\n일치율: {similarity:.2f}%") |

테스트 결과

|  |
| --- |
|  |

* 재구성된 염기서열의 길이 비정상
* 재구성된 시퀀스가 첫 리드 이외에 단순화
* 리드 테이블 생성에서 문제가 있음을 확인
* 각 숫자를 그대로 문자열로 반환하여 k-mer를 만드는 것을 확인

|  |
| --- |
|  |

* K-mer를 튜플로 유지하여 정보 손실 방지
* 오일러 경로 알고리즘 개선(차수를 확인하여 노드에 진입)
* 시퀀스 재구성 함수 개선(경로를 시퀀스로 변환)
* De bruijn 알고리즘 개선
  + K-mer 빈도수 추적
  + 각 리드의 모든 k-mer 추출
  + 낮은 빈도수의 k-mer 필터링(노이즈 제거)
    - 최소 빈도수 임계값 설정
* 오일러 그래프 알고리즘 개선
  + 진입/진출 차수 계산
  + 진입차수 < 진출차수 규칙 생성
  + 가중치가 가장 높은 엣지로 다음 노드 선택