Raport – Team nr 3

Adrian Zaręba, Jakub Rymarski, Zuzanna Piróg, Wiktor Wierzchowski

“Simulated annealing simulations of chromatin based on genomic and microscopic data”

Projekt dotyczył rozwinięcia i zastosowania symulacji przy użyciu metody "Simulated Annealing" w celu generowania struktur chromosomowych. Symulacje te bazowały na dostępnych danych mikroskopowych, które dostarczały informacje na temat pierwotnej organizacji i układu chromatyny.

W ramach tego projektu opracowaliśmy i zaimplementowaliśmy funkcję g, która wykorzystywała dane genetyczne i mikroskopowe do generowania nowej struktury chromosomowej. Funkcja g była odpowiedzialna za iteracyjne tworzenie struktury, uwzględniając różne czynniki, takie jak odległości między fragmentami genomu, czy restrykcje topologiczne chromatyny. Nowa struktura została tworzona na podstawie losowej krawędzi pierwotnej struktury.

Dodatkowo, w projekcie zaimplementowano funkcję f, która miała na celu ocenę podobieństwa wygenerowanej struktury do oryginalnej struktury chromosomowej. Funkcja f wykorzystywała mapy Hi-C do porównania kontaktów między różnymi fragmentami genomu w wygenerowanej strukturze i oryginalnej oraz korelację Piersona. Na podstawie tego porównania można było ocenić, jak dobrze struktura chromosomowa generowana przez funkcję g odwzorowuje rzeczywiste dane.

W niniejszym raporcie przedstawimy szczegółowy opis metodyki, zastosowanej implementacji, wyniki uzyskane w ramach projektu oraz dyskusję na temat dalszych możliwości rozwoju tej metody.

Funkcja g

Najpierw tworząc naszą funkcję g zastanawialiśmy się jak wybrać początkową strukturę. Zdecydowaliśmy się na wybór dowolnej krawędzi w grafie.

```python
def generuj_struktura2(G):
    # Wylosuj losowy wierzchołek początkowy
    start_node = random.choice(list(G.nodes()))
    while len(list(G.neighbors(start_node)))<1:
      start_node = random.choice(list(G.nodes()))

    # Zainicjuj ścieżkę
    path = [start_node]
    # Wybierz losowego sąsiada  wierzchołka
    current_node = path[-1]
    neighbors = list(G.neighbors(current_node))
    random.shuffle(neighbors)

    # Dodaj do ścieżki losowego sąsiada
    path.append(neighbors[0])

    return path
```

Następnym problemem było zdefiniowanie właściwej funkcji proponuj_g, która rozbudowywała naszą strukturę. Zdefiniowaliśmy 3 sposoby rozbudowywania struktury.

- **przedłużanie końca nici**

```python
def przedluz(struktura, G):
  # Wybieramy wierzchołek ostatni w ścieżce
    v = struktura[-1]
    # Sprawdzamy połączone z nim wierzchołki, które nie są w strukturze
    nie_odwiedzone = [u for u in G.neighbors(v) if u not in struktura]
    nowa_struktura=struktura
    # jeśli takie nie istnieją to koniec? Nie pójdziemy dalej?
    if len(nie_odwiedzone) != 0:
        # Szukamy najbliższego nieodwiedzonego
        min_distance = float("inf")
        nowy=0
        for u in nie_odwiedzone:
              distance = (G.nodes[v]['x']-G.nodes[u]['x'])**2+(G.nodes[v]['y']-G.nodes[u]['y'])**2+(G.nodes[v]['z']-G.nodes[u]['z'])**2
              if distance < min_distance:
                  min_distance = distance
                  nowy = u
        nowa_struktura = struktura + [nowy]
    return nowa_struktura
```

- **usuwanie ostatniego fragmentu nici**

```
def usun_ost(struktura, G):
    nowa_struktura = struktura[:-1]
    return nowa_struktura
```

- **zamiana jednej krawędzi w strukturze na dwie krawędzie. Zrobiliśmy to losując dowolną krawędź AB z naszej struktury i sprawdzając czy istnieje sąsiad obu wierzchołków A i B, który jeszcze nie jest w strukturze. Jeśli taki sąsiad istnieje to usuwamy krawędź AB i dodajemy krawędzie AC i CB.**

```
def zmien_na_dwa(struktura, G):
    nowa_struktura=struktura
    i=random.randint(0, len(struktura)-2)
    node_a=struktura[i]
    node_b=struktura[i+1]
    common_neighbors = set(G.neighbors(node_a)) & set(G.neighbors(node_b))
    for node_nr in common_neighbors:
        if node_nr not in struktura:
          nowa_struktura.insert(i+1, node_nr)
          break
    return nowa_struktura
```

Nasza funkcja proponuj_g wybiera z różnym prawdopodobieństwem 1 z 3 sposobów rozrastania się struktur. Postanowiliśmy przypisać najmniejsze prawdopodobieństwo sposobowi usuwania ostatniego fragmentu nici, ponieważ ze wstępnych prób budowy struktury wynikało, że często była ona bardzo krótka i nie chcieliśmy dodatkowo jej skracać.

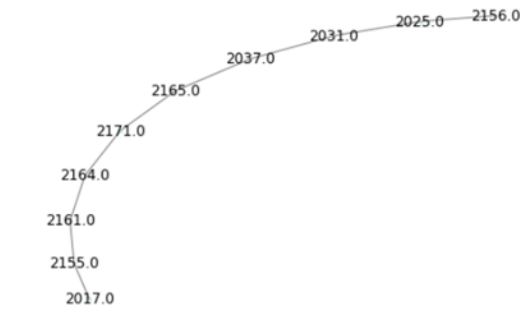
```
def proponuj_g2(struktura, G):
    i=random.uniform(0, 1)
    if i<0.4:
      nowa_struktura=przedluz(struktura, G)
    elif i<0.95:
      nowa_struktura=zmien_na_dwa(struktura, G)
    else:
      nowa_struktura=usun_ost(struktura, G)


    return nowa_struktura
```

Funkcja f’

Funkcja przyjmuje argumenty “struktura” oraz “hi-c”. Zwraca ona korelację Pearsona powstałej macierzy Hi-C ze struktury oraz wprowadzonej macierzy hi-c. Etapy działania:

- Pobieramy nasz graf początkowy oraz jego współrzędne.

```
#1 nasz graf początkowy
nx.draw(graph, with_labels=True, node_color='lightblue', node_size=2, edge_color='gray')
# Wyświetlanie grafu
plt.show()
```




```
#2 jego współrzędne
list(graph.nodes(data=True))
```

```
[(2156.0, {'x': 340.0, 'y': 334.86315612998294, 'z': 653.1972647421808}),
 (2025.0, {'x': 340.0, 'y': 357.957166897568, 'z': 620.5374015050718}),
```

- Dodajemy punkty do naszego grafu, aby uzyskać odpowiednie wymiary. Te punkty dodawane są liniowo i pojawiają się wyłącznie na ścieżkach łączących poprzednie punkty. Funkcja działa również w drugą stronę – potrafi usunąć nadmiar punktów nie zaburzając przy tym naszej ścieżki.

Liniowo przekształcamy wymiary

```
from scipy.interpolate import CubicSpline

def evenly_distribute_points(points, num_new_points):
    """
    Evenly distributes points along a path by adding new points.
    :param points: List of points [a, b, c] with three-dimensional coordinates [x, y, z].
    :param num_new_points: Number of new points to add between existing points.
    :return: List of points after evenly distributing.
    """
    num_points = len(points)

    # Extract x, y, z coordinates from the given points
    x = [point[0] for point in points]
    y = [point[1] for point in points]
    z = [point[2] for point in points]

    # Compute parameter values for interpolation
    t = range(num_points)
    t_interp = [i * (num_points - 1) / float(num_points + num_new_points - 1) for i in range(num_points + num_new_points)]

    # Perform cubic spline interpolation for each coordinate separately
    interp_x = CubicSpline(t, x)(t_interp)
    interp_y = CubicSpline(t, y)(t_interp)
    interp_z = CubicSpline(t, z)(t_interp)

    # Combine interpolated coordinates into new points
    new_points = [[interp_x[i], interp_y[i], interp_z[i]] for i in range(num_points + num_new_points)]

    return new_points
```

```
ile_nowych = 2488 - len(punkty)
nowe_punkty = evenly_distribute_points(punkty, ile_nowych)
len(nowe_punkty)
```

```
2488
```

- Z podanych nowych punktów tworzymy macierz Hi-C. Jest ona jednak dla nas obiektem nieporządanym. Przez to następnie musimy zmienić wymiary macierzy. Do tego są te funkcje:

Tworzymy mapę Hi-C ze współrzędnych

```
def create_hic_map(points):
distances = distance_matrix(points, points)
    min_value = np.min(distances)
    max_value = np.max(distances)
    inverted_distances = (max_value - distances) + min_value
    return inverted_distances
```

```
hic2 = create_hic_map(nowe_punkty)
```

Zmieniamy wymiar macierzy

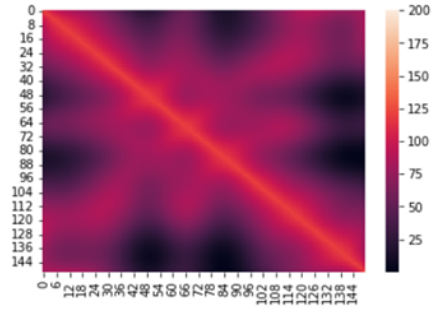
```
from scipy.ndimage import zoom

def reduce_hic_dimension(matrix, target_shape):
    original_shape = matrix.shape
    if original_shape[0] <= target_shape[0] or original_shape[1] <= target_shape[1]:
        raise ValueError("Target shape must be smaller than the original matrix shape")

    zoom_factor = (target_shape[0] / original_shape[0], target_shape[1] / original_shape[1])
    reduced_matrix = zoom(matrix, zoom_factor, order=1)

    return reduced_matrix

hic2 = reduce_hic_dimension(hic2, (150, 150))
```

```
hic2.shape
```

```
(150, 150)
```

```
sns.heatmap(hic2, vmax=300)
```

```
<AxesSubplot:>
```



- Na samym końcu dopisujemy funkcję zwracającą porównanie odległości w dwóch macierzach jako korelację Pearsona. Wszystkie poprzednie funkcje łączymy do jednej (naszej głównej f') i dostajemy finalny produkt.

Finalna funkcja 'podobienstwo_f'

```
def podobienstwo_f(struktura, hic):
    graf = list_to_graph(struktura, G)
    punkty = zapisz_jako_liste_wspolrzednych(graf)
    ile_nowych = len(dfV) - len(punkty)
    nowe_punkty = evenly_distribute_points(punkty, ile_nowych)
    hic2 = create_hic_map(nowe_punkty)
    dimensions = hic.shape
    hic2 = reduce_hic_dimension(hic2, dimensions)

    korelacja = porownaj_odleglosci(hic, hic2)
    return f"Korelacja Pearsona wynosi: {korelacja}"
```

```
podobienstwo_f(struktura2, hic)
```

```
'Korelacja Pearsona wynosi: 0.294017953758114'
```

Simulated annealing

Do przeprowadzenia symulowanego wyżarzania potrzebowaliśmy zdefiniować kilka funkcji i parametrów: temperaturę początkową, funkcję spadku temperatury, funkcję prawdopodobieństwa zmiany struktury, generującą nowe struktury i zwracającą podobieństwo struktury do docelowej mapy HiC. Funkcję generującą i funkcję określającą podobieństwo już mamy. Pozostałe funkcje i parametry zdefiniowaliśmy w następujący sposób:

```python
def acceptance_prob(podob_s, podob_s_new, T):
    '''
    Funkcja prawdopodobieństwa przyjęcia nowej struktury.
    podob_s - podobieństwo starej struktury do mapy HiC
    podob_s_new - podobieństwo nowej struktury do mapy HiC
    T - temperatura w danym momencie symulacji
    '''
    return min(math.exp(-(podob_s - podob_s_new) / T), 1)
```

```python
def simulated_annealing3(iterations, name,  G, hic):
    '''
    Funkcja przeprowadzająca symulowane wyżerzanie. Zwraca listę podobieństw kolejnych struktór do mapy HiC oraz zapisuje
    ostateczną strukturę w pliku do podglądu w Chimerze.
    iterations - liczba kroków symulacji
    name - nazwa pliku do którego zapisana ma być ostateczna struktura
    G - graf szkielet
    hic - mapa HiC na której ma wzorować się symulacja
    '''
    # inicjacja parametrów i struktury
    T_init = 50
    podob = []
    s = generuj_struktura3(G)

    # faza inflacyjna, generowanie kolejnych struktur bez kontroli podobieństwa do mapy HiC
    for i in range(500):
        s = proponuj_g3(s, G)
        podob_s = (-1)*podobienstwo_f(s, hic)
        podob.append(podob_s)

    # właściwa faza symulacji
    for i in range(iterations):
        # korektra temperatury
        T = T_init * (1 - (i)/iterations+1)
        # generacja potencjalnego kolejnego kroku symulacji
        s_new = proponuj_g3(s, G)
        # wygenerowanie i zapis podobieństw struktur
        podob_s = (-1)*podobienstwo_f(s, hic)
        podob_s_new = (-1)*podobienstwo_f(s_new, hic)
        podob.append(podob_s)
        # weryfikacja podobieństwa
        if(acceptance_prob(podob_s, podob_s_new, T) > random.uniform(0,1)):
            s = s_new

    # zapis struktury
    save_to_chimera(s, name, G)

    return podob
```

Analiza wyników

Początkowe wahania wskaźnika podobieństwa interpretujemy jako niestabilność związaną z niewielkimi rozmiarami struktury. Dodanie pojedynczej krawędzi wiąże się wtedy z relatywnie bardziej znaczącą zmianą struktury niż gdy ta jest już większa, a więc będzie miało to również większy wpływ na zmianę podobieństwa. Ponadto wczesne iteracje symulacji

charakteryzują się wysoką wartością parametru temperatury co może tłumaczyć podejmowanie decyzji o zmianie struktury nawet jeżeli wiąże się to ze znacznie gorszym podobieństwem.

Pewną uniwersalną tendencją jaką zdołaliśmy zaobserwować jest niezrozumiały spadek lub skok podobieństwa struktur po i tuż przed zakończeniem fazy inflacyjnej. W obu przypadkach zmiana ta stabilizuje się w okolicach wartości od 0,30 do 0,32.

Poniżej zamieszczone zostały wykresy prezentujące podobieństwo struktur w kolejnych iteracjach wyżarzania(liczba iteracji inflacji / liczba iteracji symulacji):

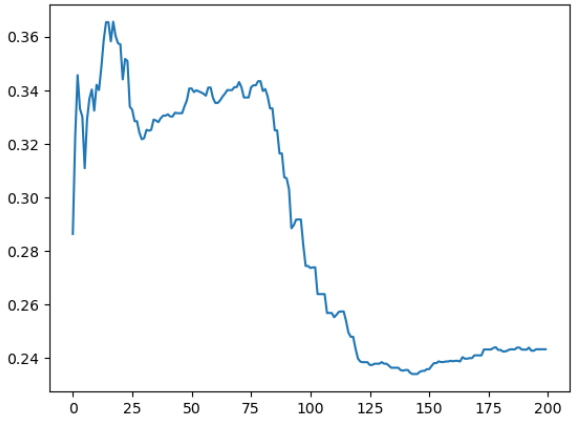


Fig.1[100/100]

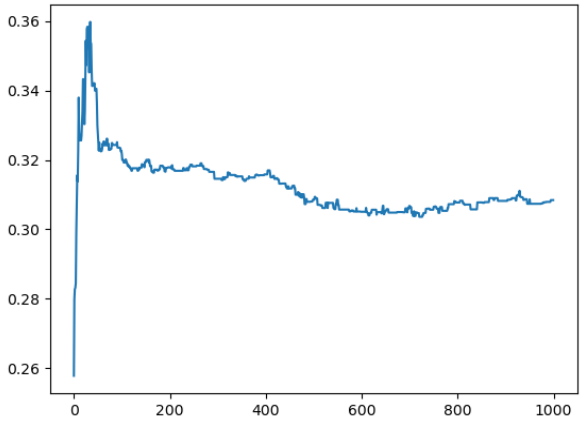


Fig.2[10/1000]

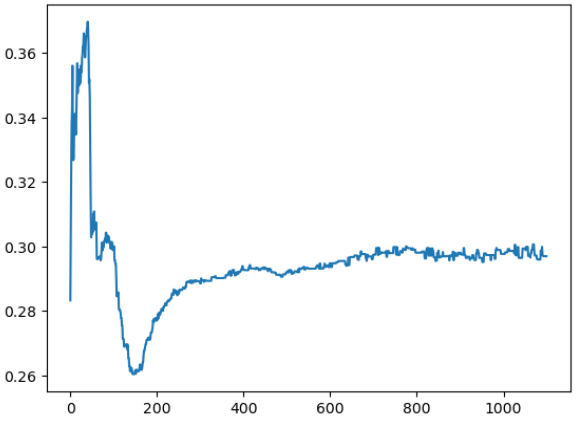


Fig.3[500/500]