Michał Chojnacki 151859
Jakub Głowacki 151865

Sprawozdanie

Sekwencjonowanie DNA w oparciu o chipy alternatywne, z błędami pozytywnymi

1) Opis problemu

Sekwencjonowanie DNA z użyciem chipów alternatywnych polega na wykorzystaniu sond, które zawierają niesprecyzowane nukleotydy. W tych sondach, nukleotydy na pozycjach parzystych są oznaczone jako X, co oznacza, że mogą reprezentować dowolny nukleotyd z zestawu {A, C, G, T}. Chipy składają się z dwóch typów sond o wzorach:

- $N_1XN_2X \dots XN_k$
- $N_1XN_2X \dots XN_{k-1}N_k$

Błędy pozytywne występują, gdy sonda wykazuje obecność określonego oligonukleotydu, mimo że w rzeczywistości nie jest on obecny w analizowanej próbce DNA. Takie błędy mogą prowadzić do nieprawidłowej interpretacji sekwencji, co utrudnia dokładne określenie sekwencji nukleotydów w próbce.

2) Sformalizowanie problemu

Dla problemu sekwencjonowania DNA z użyciem chipów alternatywnych, gdzie sondy zawierają niesprecyzowane nukleotydy oznaczone jako X (reprezentujące dowolny nukleotyd z {A, C, G, T}), należy zidentyfikować wszystkie możliwe sekwencje oligonukleotydów generowane przez sondy typu 1 i typu 2. Sondy typu 1 mają k - 1 symboli X, a sondy typu 2 mają k - 2 symboli X. Każda sonda może generować 4^n różnych sekwencji oligonukleotydów, gdzie n to liczba symboli X w sondzie.

Michał Chojnacki 151859
Jakub Głowacki 151865

3) Opis algorytmu dokładnego

Algorytm dokładny ma na celu znalezienie rozwiązania sekwencjonowania DNA. W pesymistycznym wypadku musi on rozważyć wszystkie rozwiązania problemu, co przekłada się na gorszą złożoność obliczeniową.

Na podstawie sekwencji startowej tworzone są dwa ciągi znaków: even_string i odd_string. Zawierają one co drugi znak z sekwencji startowej

string even_string = EvenStringStart(start); string odd_string = OddStringStart(start);

Oligonukleotydy typu pierwszego są przechowywane w wektorze cells1, natomiast typu drugiego w wektorze cells2. Indeksy zużytych sond są przechowywyane w odpowiednio used_cells1 i used_cells2. Główna część algorytmu to rekursywna funkcja GetSequenceDFS, która przeszukuje wszystkie możliwe kombinacje komórek, aby znaleźć pasującą sekwencję DNA.

bool GetSequenceDFS(string& even_string, string& odd_string, vector<string>& cells1, vector<string>& cells2, vector<int>& used_cells1, vector<int>& used_cells2, int& length)

Funkcja ta na przemian dodaje elementy do parzystego I nieparzystego stringa. Nasz problem posiada jedynie błędy pozytywne, więc szukamy tylko elementów z całkowitym pokryciem. Funkcja sprawdza po kolei niezużyte jeszcze sondy. Najpierw następuje sprawdzenie, czy oligonukleotyd z pierwszego zbioru pasuje (Add).

Jeżeli olinukleotyd pasuje do obecnie dokładanego ciągu znaków, to następuje walidacja poprzez znalezienie odpowiedniego elementu ze zbioru sond typu drugiego. Jeżeli on istnieje, to funkcja rekurencyjna wywołuje następną iterację programu, w przeciwnym przypadku następuje wybranie innego olinukleotydu ze zbioru typu 1. lub wyjście z funkcji zwracając wartość negatywną

Michał Chojnacki 151859
Jakub Głowacki 151865

```
for (int i = 0; i < cells1.size(); i++)
      {
         if (WhereInVector(i, used_cells1) != used_cells1.size())
            continue;
         string adder = cells1[i];
         bool even_adding_success = Add(even_string, adder);
         if (even_adding_success == true)
         {
            used_cells1.push_back(i);
            bool even_validation = Validation(even_string, odd_string, cells2, used_cells2);
            if (even_validation == true)
            {
               bool success = GetSequenceDFS(even_string, odd_string, cells1, cells2, used_cells1,
                                                   used_cells2, length);
               if (success == true) return true;
               else
               {
                  Subtract(even_string, used_cells1);
                  used_cells2.pop_back();
               }
            }
            else Subtract(even_string, used_cells1);
         }
      }
      return false;
```

Elementy zawsze dodawane są do krótszego z ciągów wynikowych.

Dodanie elementu następuje poprzez sprawdzenie, czy ostatnie elementy ciągu wynikowego idealnie nachodzą na olinukleotyd dodawany. Walidacja przebiega analogicznie.

```
bool Add(string& odd_string, string adder)
{
   int os = odd_string.size(), as = adder.size();
   for (int i = 0; i < as - 2; i = i + 2)
   {
      if (odd_string[os - as + i + 2] != adder[i]) return false;
   }
   odd_string += 'X';
   odd_string += adder[adder.size() - 1];
   return true;
}
```

Michał Chojnacki 151859
Jakub Głowacki 151865

4) Opis algorytmu wielomianowego

Algorytm wielomianowy, w tym przypadku algorytm genetyczny, ma na celu znalezienie przybliżonego rozwiązania problemu sekwencjonowania DNA z o wiele lepszą złożonością czasową w porównaniu do algorytmu dokładnego. W przeciwieństwie do algorytmu dokładnego, wynik jest jedynie przybliżony, a dla szybszego działania algorytmu nie używamy danych z drugiego typu sond do walidacji.

Algorytm genetyczny składa się z następujących kroków:

- Inicjalizacja: Algorytm rozpoczyna się od stworzenia początkowej populacji losowo wybranych rozwiązań.
- Ocena populacji: Każde rozwiązanie w populacji jest oceniane przy użyciu funkcji przystosowania, która mierzy, jak dobrze dane rozwiązanie spełnia określone kryteria.
- Selekcja do krzyżowania: Najlepsze rozwiązania są wybierane do krzyżowania i mutacji w celu stworzenia nowych rozwiązań (potomków).
- Krzyżowanie (Crossover): Wybrane rozwiązania są łączone w celu utworzenia nowych rozwiązań, które dziedziczą cechy od swoich "rodziców".
- Mutacja: Losowe zmiany są wprowadzane do nowych rozwiązań w celu zachowania różnorodności genetycznej.
- Selekcja: Nowo utworzone rozwiązania zastępują najgorsze rozwiązania w populacji.
- Sprawdzenie warunku zakończenia: Proces ten jest powtarzany, dopóki nie zostanie osiągnięty określony warunek zakończenia, taki jak liczba iteracji bez poprawy najlepszego rozwiązania.

```
int Genetic(const vector<string>& oligos) {
   vector<pair<int, vector<int>>> population = Generation(oligos, N_Start_Population);
   while (!Stop(population[0].first)) {
      Reproduction(population, N_offspring, oligos);
      mutate(population, N_Mutation);
      sort(population.begin(), population.end());
      selection_for_sorted2(population);
      sort(population.begin(), population.end());
   }
   return population[0].first;
}
```

Michał Chojnacki 151859
Jakub Głowacki 151865

Osobnik w algorytmie genetycznym określa kolejność oligonukleotydów, gdzie pierwsza połowa osobnika tworzy nieparzyste pozycje w wyniku, a druga połowa odpowiada za parzyste elementy. Ostateczny wynik jest uzyskiwany poprzez maksymalne pokrycie i ściśnięcie oligonukleotydów, co pozwala na uzyskanie jak najkrótszej sekwencji DNA używając n-k+1 elementów.

```
int funkcjaCelu(const vector<int>& order, const vector<string>& oligos) { int os = order.size();
    int total_length1 = oligos[order[0]].size();
    int total_length2 = oligos[order[os/2]].size();
    for (size_t i = 1; i < os/2; ++i) {
        total_length1 += oligos[order[i]].size() - overlap(oligos[order[i - 1]], oligos[order[i]]);
    }
    for (size_t i = os/2 + 1; i < os; ++i) {
        total_length2 += oligos[order[i]].size() - overlap(oligos[order[i - 1]], oligos[order[i]]);
    }
    return max(total_length1, total_length2) + 1;
}
```

Mutacja w naszym przypadku polega na zamianie kolejnością dwóch elementów, by zapewnić brak duplikatów oligonukleotydów w osobnikach

```
void mutate(vector<pair<int, vector<int>>>& population, int n) {
    while (n--) {
        int idx = rand() % population.size();
        int pos1 = rand() % population[idx].second.size();
        int pos2 = rand() % population[idx].second.size();
        swap(population[idx].second[pos1], population[idx].second[pos2]);
    }
}
```

Elementy do krzyżowania oraz te, które przetrwają do następnego pokolenia wybieramy z malejącym prawdobodobieństwem dla dłuższych rozwiązań.

```
int choice(int k) {
    return (int)((double)rand() / RAND_MAX * rand() / RAND_MAX * k);
}
```

Michał Chojnacki 151859
Jakub Głowacki 151865

5) Analiza uzykanych wyników

Podczas analizy, przetestowano dane z różnymi wartościami k oraz długościami wyników, gdzie długość oligonukleotydów wynosiła 2k−1. Testowane wartości to:

	Wyniki genetycznego
• k=6; długość wyniku = 20	23
• k=8; długość wyniku = 40	54
• k=8; długość wyniku = 50	82
• k=10; długość wyniku = 100	196
• k=10; długość wyniku = 500	1386

Algorytm dokładny działał całkiem szybko, co może być związane z niewielką ilością danych użytych do testów. Algorytm ten zawsze znajdował poprawne rozwiązania, zapewniając pełne pokrycie sekwencji DNA zgodnie z założeniami problemu wraz z walidacją.

Algorytm genetyczny dla mniejszych zestawów danych (mniejsze wartości k i krótsze długości wyników) dostarczał sensowne wyniki. Jednakże, dla większych danych (większe wartości k i dłuższe długości wyników) algorytm genetyczny miał tendencję do gubienia się i nie znajdował optymalnych rozwiązań. Prawdopodobnie, walidacja mogłaby poprawić wyniki algorytmu genetycznego, jednakże wprowadzenie jej znacząco spowolniłoby działanie algorytmu, co zaprzeczałoby zakładanej wysokiej wydajności.