Treffen 07.11.2014

Dorle Osterode, Stefan Dang & Lukas Götz

14. November 2014

1 Implementationsmöglichkeiten der benötigten Datentypen in Genometools

Erste Ideen zur Implementierung der Datentypen fuer Scaffold-Graphen in C unter Verwendung bestehender Datentypen aus Genome-Tools.

```
/* Scaffold-Graph */

typedef enum { VC_UNIQUE, VC_REPEAT,VC_POLYMORPHIC, VC_UNKNOWN }
    VertexClass;
typedef enum { ED_ANTISENSE, ED_SENSE } Direction;
typedef enum { EC_REVERSE, EC_SAME } Composition;

/* Vertex */

struct GtScaffoldGraphVertex
{
  /* eindeutige ID fuer den Knoten */
  GtUword id;
  /* Laenge der Sequenz, die der Contig darstellt */
  GtUword seqlen;
  /* Wert der A-Statistik, um Contigs als REPEAT oder UNIQUE
     klassifizieren zu koennen;
     in Genom-Tools vom Typ float */
  float astat;
  /* abgeschaetzte Anzahl an Vorkommen des Contigs im Genom */
  float copynum;
  /* zur Klassifikation des Knotens: REPEAT, UNIQUE, ... */
  VertexClass vertexclass;
  bool hasconflictinglink;
  GtUword nofedges;
  /* Sammlung von Kanten, die von dem Contig abgehen */
  struct GtScaffoldGraphEdge   **edges;
  /* Markierung fuer Algorithmen; aus Genome-Tools entnommen siehe
  match/rdj-contigs-graph.c */
  GtContigsGraphMarks color;
};
```

```
/* Edge */
struct GtScaffoldGraphEdge
{
  /*  Knoten, zu dem die Kante fuehrt */
  struct GtScaffoldGraphVertex *pend;
  /* Kante, die genau in die andere Richtung fuehrt */
  struct GtScaffoldGraphEdge *ptwin;
  /* Informationen zu der Verbindung zwischen den Knoten */
  GtScaffoldGraphLink link;
  /* Markierung fuer Algorithmen */
  GtContigsGraphMarks color;
  /* Abschaetzung der Entfernung der verbundenen Contigs */
  Gtword dist;
  /* Standardabweichung von der abgeschaetzten Entfernung */
  float stddev;
  /* Anzahl der Distanzinformationen, die ein Anzeichen fuer die
  Verbindung der Contigs geben */
  GtUword numpairs;
  /* enthaelt die Richtung (Sense, Antisense) und welche
     Straenge die paired-Information enthalten (die gleiche
     Richtung oder das Reverse) */
  Direction direction;
  Composition composition;
};

/* Graph */
struct GtScaffoldGraph
{
  struct GtScaffoldGraphVertex **vertices;
  GtUword nofvertices;
  struct GtScaffoldGraphEdge **edges;
  GtUword nofedges;
};
```

2 Verfeinerung des Algorithmus zur Filterung der Knoten

- graph.visit() Funktion durchläuft alle Knoten (HashMap) und ruft auf jedem Knoten die übergebene Vistor-Funktion auf. Vorher wird previsit() und danach postvisit() aufgerufen. (Quelle ScaffoldGraph.h)

- Es dürfte keine Probleme beim zusammenlegen der Filterfunktionen geben, da die Bedingungen nacheinander für jeden Knoten lokal geprüft werden können.

- Es sollte beachtet werden, dass es gegebenfalls andere Ergebnisse geben könnte, wenn in einem späteren Schritt die vorher schon herausgefilterten Knoten noch

beachtet werden.

- Die repetitiven Knoten sollten schon bei der Konstruktion des Graphen herausgefiltert werden.

- Die markierten Knoten und Kanten können nicht sofort gelöscht werden, da die restlichen Knoten noch nicht klassifiziert sind. Da die Klassifikation eines Knoten aber anhand aller Nachbarn durchgeführt wird, können die Knoten und Kanten noch nicht früher gelöscht werden.

- Bei SGA wird gezählt, wie viele Knoten anhand welches Kriteriums gelöscht werden. Dies könnte auch noch eingebaut werden.

Algorithm 1: Zusammengefasste Filterfunktion (Schritt 4a und 4b vereinigt)

```
1  foreach Knoten im Graph do
2      foreach Kantenrichtung do
3          foreach Kantenpaar (A, B) in gleiche Richtung do
4              if AmbiguousOrdering(A, B, p_cutoff) and Summe der estCopyNum der
               zugehörigen Endknoten < cn-cutoff ist then
5                  Markiere Endknoten mit kleineren estCopyNum als polymorph.
                   Markiere alle Sense- /Antisensekanten des polymorphen Knoten
                   schwarz, so dass sie im nächsten Schritt nicht mitbeachtet werden.;
                   // bei polymorphen Knoten wird nur das erste polymorphe
                      Kantenpaar markiert
6                  if Knoten ist polymorph markiert then
7                      break;
8                  end
9              end
10         end
           // polymorphe Knoten müssen nicht mehr auf inkonsistente Kanten
              überprüft werden
11         if Knoten ist polymorph then
12             break;
13         end
14         foreach Kantenpaar in gleiche Richtung do
15             Berechne Overlap von nicht-schwarzen Kantenpaar.;
16         end
17         if längster Overlap > 400 then
18             Markiere alle Sense-/Antisensekanten rot;
19         end
20     end
21 end
22 Lösche alle markierten Knoten und Kanten;
```

Algorithm 2: Funktion AmbiguousOrdering(A, B, p_cutoff)

Data: Kante A und Kante B, die auf eindeutige Ordnung geprüft werden sollen. Wahrscheinlichkeitsschwellenwert p_cutoff

Result: Ob die Kanten A und B nicht eindeutig geordnet werden können

1 $\mu = A.dist - B.dist$;
2 $\sigma^2 = A.\sigma^2 + B.\sigma^2$;
3 $t = \frac{-\mu}{\sigma \cdot \sqrt{2}}$;
4 $P_{AB} = \frac{1}{2} \cdot \left(1 + \frac{2}{\sqrt{\pi}} \int_0^t \exp -x^2 \mathrm{d}x\right)$;
5 $P_{BA} = 1 - P_{AB}$;
6 **return** $\max\{P_{AB}, P_{BA}\} \leq p\_cutoff$

3 Notizen zu dem Layout-Algorithmus

- terminale Knoten: Knoten, die nur SENSE oder ANTISENES Kanten haben (Quelle: ScaffoldAlgorithms)
- Zusammenhangskomponenten-Funktion wird in StringGraph/GraphSearchTree.h definiert
- beim Walk für den Scaffold wird zuerst für jede Zusammenhangskomponente jeder Pfad zwischen terminalen Knoten mit einer Breitensuche (ohne heuristische Auswahl der Reihenfolge der Kindsknoten) berechnet. Dabei wird die Pfadlänge (Gap-Größe) minimiert. Als Layout für eine Zusammenhangskomponente wird dann der Pfad mit der längsten Sequenz (ohne Gaps) gewählt. (Quelle: ScaffoldGraph und ScaffoldWalk)

3.1 Layout-Algorithmus

Algorithm 3: Berechnung der Scaffolds (Schritt 6)

Data: Graph
Result: Graph ohne Knoten, die nicht zum bestem Walk gehören

```
1  Markiere alle Knoten schwarz;
2  Berechnung aller Connected Components;
3  foreach Connected Component do
4  |  Berechne Menge der terminalen Knoten (mit ausschließlich SENSE oder
   |  ANTISENSE Kanten)
5  |  foreach Terminaler Knoten do
6  |  |  Berechne alle Walks;
7  |  |  foreach Walk do
8  |  |  |  if Contig-Gesamtlänge > bislang beste Contig-Gesamtlänge then
9  |  |  |  |  Setze aktuellen Walk als bestWalk;
10 |  |  |  end
11 |  |  end
12 |  end
13 end
14 Setze alle Kanten des bestWalk weiß;
15 Lösche alle schwarzen Kanten;
```

Algorithm 4: Berechnung der Walks (Schritt 6.1)

Data: terminaler Startknoten
Result: Alle von diesem Knoten möglichen Walks

```
1  Konstruktionsrichtung = Richtung der vom terminalen Knoten ausgehenden Kanten;
2  foreach Alle vom Knoten ausgehenden Kanten (in Konstruktionsrichtung) do
3  |  Speichere Startkante und Distanz in Map (für spätere Traversierung)
4  |  Schiebe Startkante und Distanz in Queue
5  end
6  while BFS über Queue nicht beendet do
7  |  if Distanz zu aktuell betrachtetem Knoten z < bisher ermittelte Distanz zu diesem
   |  Knoten OR Knoten noch unbetrachtet then
8  |  |  if Keine Kanten mehr zu betrachten then
9  |  |  |  Schiebe Knoten in terminalSet
10 |  |  end
11 |  |  Speichere Kante und Distanz in Map;
12 |  |  Schiebe Kante und Distanz in Queue;
13 |  end
14 end
15 foreach Knoten in terminalSet do
16 |  Erzeuge Walk mithilfe einer Traversierung über die Map
17 end
```

C-Datei anlegen + Implementation
Pseudocode verbessern
Test über SGA-Eingabedatei (AStatistik, BAM)
Beispiel erzeugen (ART, Virussequenz)