Jurnal Teknologi Informasi dan Ilmu Komputer (JTIIK) DOI: 10.25126/jtiik.202072070
Vol. 7, No. 2, April 2020, hlm. 407-416 p-ISSN: 2355-7699
AkreditasiKEMENRISTEKDIKTI, No.30/E/KPT/2018 e-ISSN: 2528-6579

PENGKONSTRUKSIAN *BIDIRECTED OVERLAP GRAPH* UNTUK PERAKITAN SEKUENS DNA

Wisnu Ananta Kusuma*[1], Albert Adrianus[2]

[1,2] Departemen Ilmu Komputer, Fakultas Matematika dan Ilmu Pengetahuan Alam, Institut Pertanian Bogor
Email: [1]ananta@apps.ipb.ac.id, [2] albertadrianus@gmail.com
*Penulis Korespondensi

(Naskah masuk: 01 Juni 2019, diterima untuk diterbitkan: 11 Februari 2020)

Abstrak

De novo DNA (*Deoxyribonucleic Acid*) *sequence assembly* atau perakitan sekuens DNA secara *De novo* adalah tahapan yang sangat penting dalam analisis sekuens DNA. Tahapan ini diperlukan untuk merakit atau menyambungkan kembali fragmen-fragmen DNA (*reads*) yang dihasilkan oleh *Next Generation Sequencer* menjadi genom yang utuh. Masalah perakitan DNA ini dapat direpresentasikan sebagai masalah Shortest Common Superstring (SCS). Perakitan ini memerlukan bantuan perangkat lunak untuk mendeteksi daerah yang sama pada *reads* DNA (*overlap*), mengkonstruksi *overlap graph,* dan kemudian mencari *shortest path* dari graf yang terbentuk. Metode ini dinamakan *Overlap Layout Consensus* (OLC). Hal terpenting dalam metode OLC adalah pendeteksian *overlap* dari masing-masing *reads.* Pada penelitian ini dikembangkan suatu teknik untuk membuat *bidirected overlap graph*. *Suffix array* digunakan untuk menentukan bagian *overlap* dari setiap *reads* dengan melakukan pengindeksan setiap *suffix* dari *reads*. Proses perakitan sekuens DNA merupakan suatu proses komputasi yang intensif. Untuk mengefisiensikan proses dilakukan perubahan masing-masing *suffix* dan *prefix* menjadi suatu nilai tertentu yang bersifat tunggal dan mencari *overlap* dengan membandingkan nilai yang merupakan representasi dari setiap *reads*. Cara ini lebih efisien dibandingkan melakukan pendeteksian *overlap* dengan metode pencocokan *string*. Hasil perbandingan menunjukkan bahwa waktu yang diperlukan untuk mengeksekusi metode yang diusulkan (perbandingan nilai) jauh lebih singkat dibandingkan dengan menggunakan metode pencocokan *string*. Untuk jumlah *reads* 2000 dan 5000 *reads,* teknik yang diusulkan ini dapat merepresentasikan semua *reads* menjadi *overlap* graph, di mana semua *reads* dapat direpresentasikan ke dalam *node* dan semua *overlap* dapat direpresentasikan ke dalam *edge*.

Kata kunci: *Bidirected overlap graph*, DNA *assembly, Overlap Layout Consensus, Shortest Commong Superstring Problem*

BIDIRECTED OVERLAP GRAPH FOR DNA SEQUENCE ASSEMBLY (OVERLAP LAYOUT CONSENSUS)

Abstract

De novo DNA sequence assembly is the important step in DNA sequence analysis. This step is required for assembling fragments or reads produced by Next Generation Sequencer to yield a whole genome. The problem of DNA assembly could be represented as the Shortest Common Superstring (SCS) problem. The assembly requires a software for detecting the overlap region among reads, constructing an overlap graph, and finding the shortest path from the overlap graph.. This method is popular as The Overlap Layout Consensus (OLC). The most important step in OLC is detecting overlaps among reads. This study developed a new approach to construct bidirected overlap graph. Suffis array was used for detecting overlap region from each reads by indexing suffix of each reads. DNA assembly process is computational intensive. To reduce the execution time, suffix and prefix was converted into the single value so that the detection of overlap could be done by comparing the values. This method is much more efficient compared to that of using string matching.With 5000 and 10000 reads, the proposed method (value comparison) could yield the perfect overlap graph, in which almost all reads and overlap could be represented as nodes and edges, respectively.

Keywords: Bidirected *overlap* graph, DNA assembly, *Overlap* Layout Consensus, *Shortest Commong Superstring Problem*

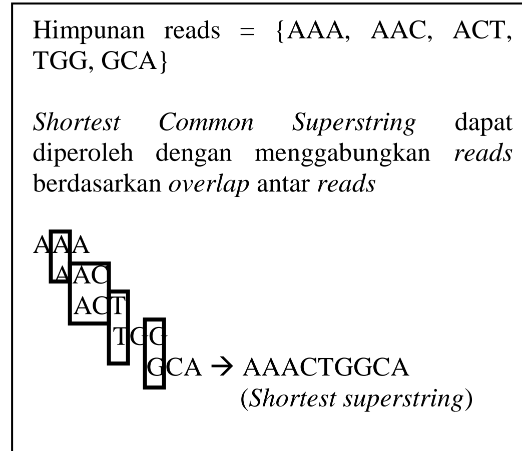
1. PENDAHULUAN

Teknologi DNA perakitan sekuens DNA (DNA *sequence assembly*) telah berkembang pesat dan mempunyai peran penting dalam analisis genom. Teknik perakitan sekuens DNA muncul karena hingga saat ini belum ada teknologi yang dapat membaca seluruh genom (DNA) suatu organisme dalam satu kali *sequencing*. Sebelumnya teknologi yang digunakan untuk membaca sekuens DNA adalah teknologi Sanger *sequencing*. Dengan menggunakan teknologi Sanger *sequencing*, dihasilkan potongan-potongan *reads* dengan panjang 1000-2000 bp namun membutuhkan waktu yang lama dan biaya yang besar (Kae, 2003) (Pop, 2009).

Seiring berjalannya waktu, ditemukan suatu teknologi baru yaitu *high throughput sequencing technologies* (Zhou, 2010). Dengan teknologi tersebut, genom bakteri dimungkinkan untuk dapat dibaca hanya dalam satu eksperimen dan dengan biaya yang tidak mahal (Brenner *et al*., 2000). Hasil dari penguraian ini berupa jutaan potongan *reads* dengan panjang 35-50 bp. Potongan *reads* ini selanjutnya disambungkan dengan *reads* lain menjadi potongan-potongan yang lebih panjang (*contigs*), sebelum akhirnya dengan mekanisme *layout* dan consensus dihasilkan genom utuh (*whole genome*). Pendekatan ini memerlukan metode untuk menyambung atau merakit (*assemble*) *reads* yang ada.

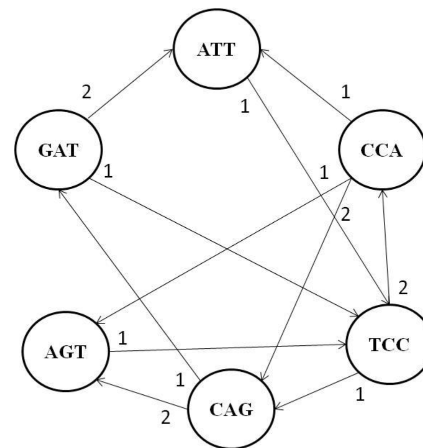
Probrem perakitan kembali *reads* menjadi *contigs* sampai akhirnya diperoleh genom utuh ini dapat diformulasikan sebagai problem *Shortest Common Superstring*. Perumusan problem ini berdasarkan asumsi bahwa setiap *reads* harus ada di dalam genom aslinya. Hal ini karena *reads* pada dasarnya diperoleh dengan melakukan *sequencing* atau pembacaan secara acak dari genom yang utuh dengan menggunakan *sequencer* sehingga dihasilkan potongan-potongan fragmen yang saling *overlap* yang disebut *reads.* Dengan demikian, jika akan direkonstruksi kembali genom yang tersebut, maka dapat diperoleh dari *shortest sequence* atau string yang mengandung semua *reads* sebagai substringnya (Medvedev, 2007) (Gambar 1).

Secara garis besar metode yang umum digunakan untuk melakukan perakitan kembali *reads* ada dua yaitu metode *Overlap* Layout Consensus (OLC) dan dengan menggunakan graf De Bruijn (Kusuma *et al*., 2011) (Simpson and Pop, 2015). Meskipun demikian ada juga peneliti yang menggunakan pendekatan probalistik sebagai alternatif, seperti pada GAML (Boza, 2014) Pendekatan OLC adalah pendekatan yang intuitif di mana problem perakitan sekuens ini direprentasikan menjadi *overlap graph*. Pada *overlap graph* ini *node* merepresentasikan *reads* dan *edge* merepresentasikan *overlap* antar *reads* (Gambar 2). Pembentukan *overlap* graph ini adalah tahap terpenting dalam proses perakitan sekuens DNA.



Gambar 1 Ilustrasi Problem *Shortest Common Superstring*

reads = {ATT, CCA, CAG, TCC, AGT, GAT}



Gambar 2 Ilustrasi *Overlap Graph*

Sampai saat ini, pengembangan perangkat lunak untuk melakukan perakitan *reads* terus dilakukan. Beberapa perangkat yang menggunakan metode OLC untuk merakit *reads* tersebut antara lain Celera (Myers, 2000), ARACHNE (Batzouglou, 2002), PCAP (Huang, 2003), Phusion (Mullikin, 2003), dan EDENA (Hernandez *et al.*, 2008). Adapun contoh yang dibangun berdasarkan pendekatan *De Bruijn Graph* adalah Velvet (Zerbino, 2008), SOAPDenovo (Li et. al., 2009), dan SPAdes (Bankevich et. al., 2012).

Pada EDENA, data *input* yang berupa potongan-potongan *short reads* akan diproses menjadi *output* yang berupa potongan-potongan *contigs* serta jumlah *node* dan *edge* yang terbentuk dari proses-proses yang digunakan di dalam metode OLC. Proses-proses tersebut terdiri atas proses penghilangan data *reads* yang redundan, pembuatan *bidirected overlap graph*, penghilangan *transitive edges,* serta pembersihan *graph* (Hernandez *et al*., 2008). Semua proses tersebut harus dilakukan secara berurutan.

Dari semua proses atau tahapan tersebut, proses yang paling penting adalah pembuatan *bidirected overlap graph*. Proses ini digunakan untuk mendeteksi *overlap* dari masing-masing *reads*. Jika

mekanisme pengkonstruksian *bidirected overlap graph* dapat disederhanakan, maka proses perakitan DNA secara keseluruhan akan menjadi lebih efisien. Hal ini karena kompleksitas dari proses perakitan DNA dengan metode OLC ini dipengaruhi secara signifikan oleh kompleksitas saat dilakukannya pengkonstruksian *bidirected overlap graph*, khususnya saat pendeteksian *overlap* antar *reads*

Pada umumnya proses pendeteksian *overlap* antar *reads* untuk mengkonstruksi *overlap* graph dilakukan dengan algoritme *string matching* atau pencocokan string dalam hal ini adalah *reads*, seperti yang dilakukan pada Edena dengan menggunakan prefix tree (Hernandez *et al.*, 2008). Cara lain adalah dengan menggunakan suffix array.

Pada penelitian ini algoritme pencocokan string dimodifikasi dengan menkonversikan tiap basa DNA yang dalam hal ini direpresentasikan sebagai satu karakter A, T, G, atau C ke dalam suatu nilai atau nilai. Dengan demikian proses pendeteksian *overlap* tidak dilakukan dengan melakukan pencocokan string tapi dengan membandingkan nilai atau nilai yang merepresentasikan basa DNA.

Input yang digunakan adalah potongan-potongan *reads* dengan panjang yang sama dan *output* yang dihasilkan adalah jumlah *node* dan *edge* yang dihasilkan dari proses yang ada. Hasil dari pendekatan menggunakan perbandingan nilai dibandingkan dengan pendekatan pencocokan string yang telah umum digunakan oleh peneliti-peneliti sebumnya.

2. METODOLOGI

Data

Data yang digunakan merupakan potongan-potongan *reads* dengan panjang 35 base pairs (bp) yang dibangkitkan menggunakan perangkat lunak simulasi *sequencing*, yaitu MetaSim (Richter *et al.,* 2008). *Reads* dari DNA dapat direpresentasikan sebagai string yang terdiri atas empat karakter A, C, G, dan T. Tiap karakter mewakili basa-basa nitrogen, yaitu A untuk Adenine, C untuk Cytosine, G untuk Guanine dan T mewakili Thymine. *Reads* ini disimpan dalam file dengan format FASTA. Contoh formal FASTA dapat dilihat pada Gambar 3.

Data yang diperlukan merupakan data yang tidak mengandung redundansi dan bebas dari *sequencing error*. Informasi selengkapnya mengenai data *reads* yang digunakan pada penelitian ini dapat dilihat pada Tabel 1.

```
>AB000263  |acc=AB000263|descr=Homo
sapiens mRNA for prepro cortistatin
like peptide, complete cds.|len=35
ACAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTG
```

Gambar 3 Contoh *Reads* dengan Format FASTA

Pada penelitian ini digunakan dua data DNA, yaitu yang berasal dari Acidiphilium multivorum AIU301 plasmid pACMV4, yang memiliki panjang total sekitar 40.6 *Kilo basepair* (Kbp) dan Saccharomyces cerevisiae EC1118 chromosome V, yang memiliki panjang total genomnya sekitar 12 Mbp. Keduanya dipilih agar data DNA yang digunakan merepresentasikan ukuran genom yang pendek dan panjang.

Untuk genom dengan ukuran yang pendek, yaitu Acidiphilium multivorum AIU301 plasmid pACMV4, dibangkitkan *reads* dengan menggunakan perangkat lunak simulasi *sequencer* Metasim dengan jumlah *reads* 2000, 5000, dan 10000. Adapun untuk Saccharomyces cerevisiae EC1118 chromosome V karena ukurannya jauh lebih panjang, maka jumlah *reads* yang dibangkitkan lebih banyak, yaitu 5000 dan 1000. Hal ini dilakukan agar jumlah *reads* yang dibangkitkan memiliki coverage yang memadai untuk membentuk *bidirected overlap graph.*

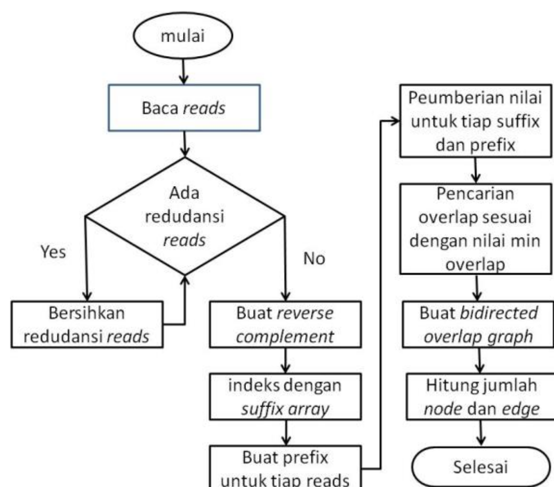
Tabel 1 Data Sekuens DNA yang Digunakan dalam Penelitian

No	Sekuens DNA (Organisme)	Panjang Genom	Jumlah *Reads*
1.	Acidiphilium multivorum AIU301 plasmid pACMV4	40.6 Kbp	2000, 5000 dan 10000
2	Saccharomyces cerevisiae EC1118 chromosome V	12 Mbp	5000 dan 10000

Dalam simulasi menggunakan MetaSim, digunakan *error* model dengan pilihan exact. Hal tersebut dilakukan agar data yang dihasilkan nantinya merupakan data yang bebas kesalahan (*error free*). Data yang *error free* berarti dismulasikan tidak ada kesalahan pembacaan oleh *sequencer*. Dengan demikian akan dihasilkan *reads* di mana urutan basa nitrogennya sama dengan organism aslinya. Hasil simulasi dari perangkat MetaSim masih memungkinkan memiliki data yang redundan. Data redundan yang dimaksud adalah adanya *read* yang sama persis satu dengan yang lain, mengingat *reads* dibangkitkan secara acak. Pada penelitian ini dilakukan praproses dengan menghilangkan data yang redundan. Hal tersebut diperlukan agar graf yang terbentuk tidak kompleks. Selain itu juga dilakukan pembuatan *reverse complement* dari setiap *reads*. Hal ini disebabkan proses *sequencing* dilakukan dari dua sesuai dengan karakteristik struktur DNA yang berbentuk *double helix.*

Metode

Penelitian terdiri atas beberapa tahap antara lain pembacaan *reads* dan pembuatan *reverse complement*, pengindeksan dengan menggunakan suffix array, pencarian *overlap* antar *reads* dan *reverse complement*, serta perhitungan *node* dan *edge* sebagai hasil keluaran dari sistem. Tahapan atau proses yang dilakukan oleh perangkat lunak pengkonstruksi *bidirected overlap graph* secara ringkas dapat dilihat pada Gambar 4.



Gambar 4 Tahapan Pembentukan *Bidirected Graph*

Pembacaan *input reads*

Pertama-tama sistem akan membaca data masukan yang berupa potongan-potongan *reads* hasil simulasi dari software MetaSim. Data yang akan dibaca oleh sistem hanyalah data input yang merupakan potogan dari DNA, sedangkan data yang berupa keterangan atau header tidak dibaca oleh sistem. Jika dilihat dari Gambar 1, data yang digunakan adalah data yang memuat informasi DNA atau basa atau nukleotida, sebagai berikut `ACAAGATGCCATTGTCCCCCGGCCTCCTGCTGCTG`.

Potongan-potongan *reads* tersebut mempunyai panjang 35 bp (base pair). Data yang digunakan sebanyak 2000 dan 5000 buah.

Pembersihan *input reads* redundan

Reads yang telah dibaca oleh sistem akan diperiksa satu persatu. Apabila ada *reads* identik yang berjumlah lebih dari satu maka sistem hanya akan mengambil salah satu *read*. Hal ini agar data *input* yang akan digunakan untuk proses selanjutnya bebas dari data yang redundan. Data yang bebas redundan tersebut kemudian akan dimasukan ke dalam sebuah *vector* untuk digunakan pada proses selanjutnya.

Pembuatan *reverse complement*

Setelah itu sistem akan membuat *reverse complement* dari masing-masing read yang menjadi masukan. *Reverse complement* itu sendiri merupakan *reads* yang isinya ditukar menurut pasangan basa nitrogennya (A menjadi T, C menjadi G, G menjadi C, dan T menjadi A) dan setelah itu urutan dari basa-basa nitrogen tersebut diurutkan terbalik. Hal ini dikarenakan high throughput sequencing technologies menguraikan DNA dari 2 sisi, dari sisi kiri dan sisi kanan. Karena itu, hasil potongan-potongan *reads* tidak diketahui apakah merupakan potongan dari rantai primer atau merupakan bagian dari rantai sekunder.

Penentuan jumlah minimum *overlap*

Poin penting dalam penelitian ini adalah jumlah minimum *overlap*, karena akan menentukan banyaknya jumlah *reads overlap* satu sama lain (*node*) dan jumlah *overlap* secara keseluruhan (*edge*). Hal ini juga dikarenakan apabila jumlah minimum *overlap* terlalu sedikit akan membuat graph yang dibentuk menjadi semakin kompleks. Graph yang terbentuk semakin kompleks karena semakin kecil jumlah minimum *overlap* maka jumlah read yang saling *overlap* satu sama lain akan semakin banyak. Hal tersebut akan membuat jumlah *node* dan *edge* yang dihasilkan akan semakin banyak. Adapun jika jumlah minimum *overlap* terlalu besar, maka akan menyebabkan semakin sedikit *reads* yang saling *overlap*. Hal tersebut akan menyebabkan terjadinya dead end path pada graph ketika proses penyambungan kumpulan *reads* yang saling *overlap*. Karena jumlah minimum *overlap* menjadi penting maka selanjutnya sistem akan meminta jumlah minimum *overlap* yang diharapkan oleh user.

Jumlah minimum *overlap* yang digunakan dalam rentang 20 sampai 30. Apabila jumlah yang dimasukan user tidak dalam rentang tersebut maka sistem akan meminta user untuk memasukkan nilai lagi sampai nilai masukan user berada dalam rentang yang telah ditetapkan.

Pengindeksan dengan *suffix array*

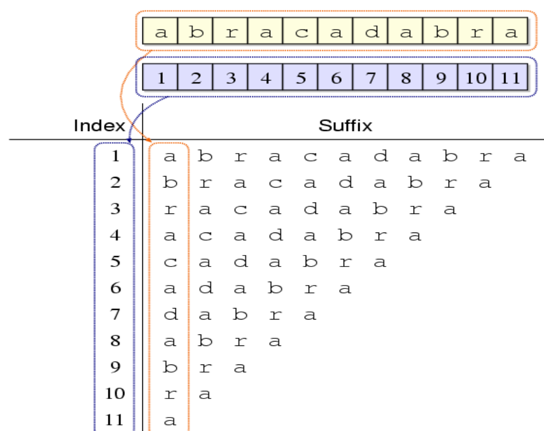
Tahap selanjutnya yaitu masing-masing *reads* akan diindeks dengan menggunakan suffix array. Suffix array merupakan perkembangan dari suffix tree. Suffix array adalah sebuah list terurut dari semua suffix suatu kata (Manber dan Myers 1993).

Misalkan terdapat sebuah string “abracadabra”. Hal pertama yang akan dilakukan adalah proses pemasangan indeks ke masing-masing karakter dari string tersebut. Sebagai contoh pada string “abracadabra”, a akan diberi indeks 1, b diberi indeks 2, r diberi indeks 3, a diberi indeks 4, c diberi indeks 5, a diberi indeks 6, d diberi indeks 7, a diberi indeks 8, b diberi indeks 9, r diberi indeks 10, dan a diberi indeks 11. Setelah pemasangan indeks terhadap masing-masing karakter dari string tersebut kemudian dibuat suffix dari string tersebut sesuai dengan indeks yang diberikan. Dalam kasus dengan string “abracadabra” maka akan terdapat 11 suffix sesuai indeks yang ada yaitu “abracadabra”, “bracadabra”, “racadabra”, “acadabra”, “cadabra”, “adabra”, “dabra”, “abra”, “bra”, “ra”, dan “a”. Untuk lebih jelasnya dapat dilihat pada Gambar 5.

Setelah didapatkan indeks seperti di atas maka kemudian akan diurutkan secara lexicographical atau berdasarkan abjad seperti pada Tabel 2.

Karena itu *suffix array* untuk string “abracadabra” adalah {10 7 0 3 5 8 1 4 6 9 2}. Pada penelitian ini, *suffix array* digunakan untuk mendata *overlap* dari masing-masing *reads* menurut nilai jumlah minimum *overlap* yang telah ditetapkan oleh

user sebelumnya. Bagian *suffix* yang tidak memenuhi jumlah minimum *overlap* akan langsung dihilangkan agar tidak terproses ke tahap selanjutnya. Selain itu dilakukan juga hal yang sama pada setiap *reverse complement* dari masing-masing read. Pada tahap yang sama juga dilakukan pengindeksan *prefix* dari masing-masing *reads* dan *reverse complement* yang ada. Sehingga pada akhirnya didapat *prefix* dan *suffix* dari masing-masing *reads* dan *reverse complement* untuk diproses pada tahap selanjutnya



Gambar 5 Pengindeksan Suffix Array

Tabel 2 *Suffix Array* Setelah Diurutkan

Sorted Suffix	Index
a	10
a b r a	7
a b r a c a d a b r a	0
a c a d a b r a	3
a d a b r a	5
b r a	8
b r a c a d a b r a	1
c a d a b r a	4
d a b r a	6
r a	9
r a c a d a b r a	2

Pembuatan *prefix*

Pada tahap ini akan dicari *prefix* dari setiap read dan *reverse complement* yang ada. *Prefix* yang dicari juga mempunyai ketentuan tertentu. *Prefix* akan didata apabila memiliki jumlah karakter lebih besar dari jumlah minimum *overlap* yang dimasukan oleh user sebelumnya. Proses yang dilakukan sebenarnya sama dengan proses pencarian suffix, namun demikian bedanya pada tahap ini yang dicari adalah bagian *prefix*.

Pemberian nilai pada suffix dan *prefix*

Setelah masing-masing *reads* dan *reverse complement* telah memiliki *prefix* dan *suffix*, setiap *prefix* dan *suffix* tersebut diubah menjadi nilai. Adapun pengubahan ini dilakukan untuk membuat waktu yang dibutuhkan pada proses selanjutnya menjadi lebih singkat. Proses pengubahan dari string menjadi double dilakukan dengan ketetapan tertentu. Aturan pertama adalah dengan cara membuat nilai tertentu bagi masing-masing huruf dari basa nitrogen yang ada. Penentuan nilai bagi masing-masing basa nitrogen dapat dilihat pada Tabel 3.

Tabel 3 Nilai Masing-masing Basa Nitrogen

Basa Nitrogen	Nilai
A	1.000000
T	0.110000
G	0.111100
C	0.111111

Setelah itu dari masing-masing nilai dari huruf basa nitrogen yang ada dikalikan dengan posisinya dan dengan suatu nilai tertentu. Lalu semua nilai tersebut dijumlahkan hingga menjadi sebuah nilai tunggal yang merepresentasikan *suffix* dan *prefix* dari masing-masing *reads* dan *reverse complement* yang ada (aturan pertama). Nilai hasil tersebut memungkinkan adanya suatu *collision* yang yaitu keadaan di mana ada dua *suffix/prefix* yang memiliki nilai yang sama namun bentuk fisik yang berbeda, karena itulah diperlukan nilai kedua yaitu nilai jumlah karakter suatu *suffix/prefix* (aturan kedua). Nilai suatu *suffix/prefix* (nilai suatu *overlap*) dapat dicari dengan formula seperti dit.unjukkan pada Persamaan 1-4 berikut

Basa nitrogen "A" :

Nilai suatu *suffix/prefix* = $\sum_{i=1}^{max} i \bullet a \bullet 0.11$ (1)

Basa nitrogen "T" :

Nilai suatu *suffix/prefix* = $\sum_{i=1}^{maks} i \bullet a \bullet 0.1111$ (2)

Basa nitrogen "G" :

Nilai suatu *suffix/prefix* = $\sum_{i=1}^{maks} i \bullet a \bullet 0.111111$ (3)

Basa nitrogen "C" :

Nilai suatu *suffix/prefix*=

$$\sum_{i=1}^{maks} i \bullet a \bullet 0.11111111 \quad (4)$$

Keterangan :
i : posisi suatu variabel
a : nilai variabel pada posisi *i*
max : jumlah maksimal variabel suatu *suffix* atau *prefix*

Aturan kedua digunakan untuk mengetahui jumlah tiap basa nitrogen pada masing-masing *suffix* dan *prefix*. Pencarian nilai tiap jenis basa nitrogen berbeda. Setiap basa nitrogen "A" akan menambah nilai sebesar 1, basa nitrogen "T" akan menambah nilai jumlah karakter *suffix/prefix* sebelumnya sebesar 0.11, basa nitrogen "G" akan menambah nilai jumlah karakter *suffix/prefix* sebelumnya sebesar 0.1111, dan basa nitrogen "C" akan

menambah nilai jumlah karakter *suffix/prefix* sebelumnya sebesar 0.111111. Secara ringkas dapat dilihat pada Persamaan 5 berikut.

$$\text{Nilai Jumlah karakter } \textit{suffix/prefix} = \sum_{i=1}^{maks} a \qquad (5)$$

Keterangan :
a : nilai variabel pada posisi *i*
max : jumlah maksimal variabel suatu *suffix* atau *prefix*

Cuplikan kode program untuk pemberian nilai suatu *suffix/prefix* dan nilai jumlah karakter *suffix* dan *prefix* dapat dilihat pada Gambar 6 dan Gambar 7.

```
double ubah_angka(string read) {
  double hasil=0;
  for(int i = 0 ; i < read.length() ; i++){
        if(read[i] =='A')
            hasil+=(((i+1)*1)*0.11);
        if(read[i] =='T')
            hasil+=(((i+1)*0.11)*0.1111);
        if(read[i] =='G')
            hasil+=(((i+1)*0.1111)*0.111111);
        if(read[i] =='C')
            hasil+=(((i+1)*0.111111)*0.11111111);
  }
  return hasil;
}
```

Gambar 6 *Pseudocode* Pemberian Nilai *Suffix/Prefix*

Pencarian bagian *overlap*

Setelah didapatkan nilai dari tiap *prefix* dan *suffix* menurut aturan pertama dan kedua, maka akan dilakukan proses pencarian *overlap*. Proses pencarian *overlap* dilakukan dengan membandingkan nilai dari masing-masing *prefix* dan *suffix* dari *reads* serta *reverse complement* satu sama lain.

```
double cek_karakter(string read) {
  double cek=0;
  for(int i = 0 ; i < read.length() ; i++){
      if(read[i] =='A')
          cek+=1;
      if(read[i] =='T')
          cek/=0.11;
      if(read[i] =='G')
          cek*=0.1111;
      if(read[i] =='C')
          cek+=0.111111;
  }
  return cek;
}
```

Gambar 7 *Pseudocode* Perhitungan Jumlah Karakter *Suffix/Prefix*

Proses perbandingan dilakukan dengan mengurangkan nilai dari masing-masing *suffix* yang ada dengan nilai dari masing-masing *prefix* yang ada. *Suffix* yang digunakan bukan hanya berasal dari bagian *reads* saja tetapi juga dari bagian *reverse complement*. Demikian juga *prefix* yang digunakan merupakan *prefix* dari bagian *reads* dan bagian *reverse complement*. Hal ini menyebabkan adanya 4 proses perbandingan yang dilakukan oleh sistem untuk menemukan bagian *overlap* dari masing-masing *reads*. Dalam hal ini, apabila nilai aturan pertama dan aturan kedua bernilai 0 maka *reads/reverse complement* dengan *suffix* tersebut memiliki *overlap* dengan *reads/reverse complement* dengan *prefix* yang bersangkutan. Untuk lebih jelasnya dapat dilihat pada Tabel 3.

Tabel 3 Proses Pencarian *Overlap*

Jenis	Keterangan
S *reads* - P *reads*	Nilai *suffix* dari masing-masing *read* dikurangkan dengan nilai *prefix* dari masing-masing *reads*
S *reads* - P RC	Nilai *suffix* dari masing-masing *read* dikurangkan dengan nilai *prefix* dari masing-masing *reverse complement*
S RC - P *reads*	Nilai *suffix* dari masing-masing *reverse complement* dikurangkan dengan nilai *prefix* dari masing-masing *reads*
S RC - P RC	Nilai *suffix* dari masing-masing *reverse complement* dikurangkan dengan nilai *prefix* dari masing-masing *reverse complement*

Evaluasi jumlah *node* dan *edge* serta waktu eksekusi

Pada tahap ini dilakukan proses evaluasi pada jumlah *node* dan *edge* yang dihasilkan oleh sistem serta waktu yang dibutuhkan untuk memprosesnya. Setelah semua *reads* (termasuk *reverse complement*) yang saling *overlap* telah terdata, maka dilakukan proses selanjutnya yaitu proses perhitungan *node* dan *edge* yang merupakan output dari sistem ini. *Node* merupakan perwakilan dari jumlah *reads* yang saling *overlap* satu sama lain sedangkan *edge* merupakan perwakilan dari jumlah *overlap* yang ada. Setelah proses pencarian *overlap* dilakukan maka setiap *reads* yang saling *overlap* dimasukan ke dalam suatu matriks yang mempunyai ukuran n × n, di mana n adalah jumlah *reads* keseluruhan.

Pada sistem ini digunakan 2 matriks. Matriks pertama digunakan untuk menandai bagian *overlap* pada *reads* dan matriks kedua digunakan untuk menampung bagian *overlap* dari *reverse* komplemen. Pada matriks 1, apabila *reads* i memiliki *overlap* dengan *reads* j maka matriks 1 baris i dan kolom j akan berisi nilai integer 1. Apabila *reads* i *overlap* dengan *reverse complement* j maka matriks 1 baris i dan kolom j akan berisi nilai integer 1. Adapun apabila *reads* i memiliki *overlap* dengan *reads* j dan *reverse complement* j maka nilainya akan menjadi 3. Hal yang sama juga diberlakukan pada matriks 2. Apabila *reverse complement* i memiliki *overlap* dengan *reads* j maka matriks 2 baris i dan kolom j akan berisi nilai integer 1. Apabila apabila *reverse complement* i memiliki *overlap* dengan *reverse complement* j maka matriks 2 baris i dan kolom j akan berisi nilai integer 1. Adapun apabila *reverse complement* i memiliki *overlap* dengan *reads* j dan *reverse complement* j maka nilainya akan menjadi 3. Setelah semua

matriks terisi maka dicari jumlah *node* dan *edge* dari kedua matriks tersebut. Secara ringkas dapat dilihat pada Tabel 4.

Tabel 4 Matriks untuk Perhitungan Jumlah *Node* dan *Edge*

Matriks	Baris i	Kolom j	*overlap*	Nilai
Matriks 1	*read i*	*read j*	Ya	1
	read i	*reverse complement j*	Ya	2
	read i	*read j dan reverse complement j*	Ya	3
Matriks 2	*reverse complement i*	*read j*	Ya	1
	reverse complement i	*reverse complement j*	Ya	2
	reverse complement i	*read j dan reverse complement j*	Ya	3

Selain jumlah *node* dan *edge*, salah satu hal yang diperhatikan dalam penelitian ini adalah waktu eksekusi yang diperlukan oleh metode perbandingan nilai dan metode pencocokan string untuk jumlah input yang sama. Waktu eksekusi dihitung setelah user memasukan input berupa nilai panjang minimum *overlap* yang diinginkan. Waktu ekseskusi digunakan untuk membandingkan antara metode perbandingan nilai dan metode pencocokan string. Satuan waktu yang digunakan adalah detik (s).

Spesifikasi Perangkat Lunak dan Perangkat Keras

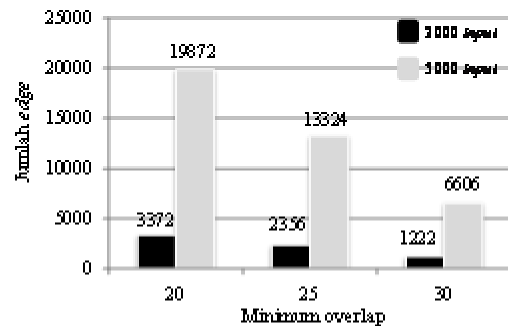
Perangkat lunak yang digunakan untuk mengembangkan sistem ini adalah Dev-C++ 5.4.1. Adapun perangkat keras yang digunakan untuk mengembangkan sistem ini adalah notebook bertipe Lenovo ThinkPad, CPU Intel Core i5-5200 2.20 GHz dengan memori 4 GB.

3. HASIL DAN PEMBAHASAN

Pada penelitian ini panjang minimum *overlap* yang digunakan adalah 20, 25, dan 30. Evaluasi dilakukan dengan melihat jumlah *node* dan *edge* serta waktu eksekusi yang dibutuhkan untuk mengkonstruksi *overlap graph*. Semakin sedikit *node* dan *edge* yang dihasilkan berarti *overlap graph* yang dihasilkan semakin sederhana. Semakin sedikit waktu yang dibutuhkan untuk mengkonstruksi *overlap graph* semakin efisien algoritme atau metode yang digunakan.

Ketepatan hasil *node* dan *edge*

Perbandingan hasil jumlah *edge* antara *input* 2000 *reads* dan 5000 *reads* dapat dilihat pada Gambar 8. Semakin besar nilai minimum *overlap* yang digunakan akan semakin sedikit jumlah *edge* yang dibentuk. Hal ini disebabkan parameter minimum *overlap* ini menunjukkan panjang substring yang sama antar dua *reads*. Dengan demikian, semakin kecil nilai minimum *overlap*, semakin kecil pula panjang substring pada *reads*, di mana menyebabkan semakin meningkatnya peluang ditemukannya substring yang sama (*overlap*) pada *reads* lainnya.



Gambar 8 Perbandingan Jumlah *Edge* untuk Input Jumlah *Reads* 2000 dan 5000

Hasil *node* dan *edge* yang dihasilkan pada penelitian ini dengan menggunakan metode perbandingan nilai dibandingkan dengan hasil *node* dan *edge* yang dihasilkan dengan metode konvensional (*pencocokan string*). Perbandingan dilakukan dengan jumlah *reads* 5000 dan 10000 dari dua sekuens DNA, yaitu dari Acidiphilium multivorum AIU301 plasmid pACMV4 dan Saccharomyces cerevisiae EC1118 chromosome V.

Tabel 5 menunjukkan perbandingan antara jumlah *node* dan *edge* dari metode perbandingan nilai dan metode *pencocokan string* dengan menggunakan sekuens DNA dari Acidiphilium multivorum AIU301 plasmid pACMV4. Dari hasil yang diperoleh, jumlah *node* dan *edge* yang dihasilkan dengan menggunakan metode perbandingan nilai sama persis dengan jumlah *node* dan *edge* yang dihasilkan dengan menggunakan metode *pencocokan string* untuk nilai minimum *overlap* 20, 25, dan 30 dengan jumlah *reads* 5000 dan 10000. Dengan demikian akurasi pembentukan *node* dan *edge* dari metode yang diusulkan dibandingkan dengan jumlah *node* dan *edge* yang dihasilkan dengan metode pencocokan string adalah sebesar 100%

Adapun dengan menggunakan sekuens Saccharomyces cerevisiae EC1118 chromosome V, jumlah *node* dan *edge* yang dihasilkan oleh kedua metode hampir sebagian besar sama, hanya terdapat sedikit perbedaan, yaitu pada nilai minimum *overlap* 20 dan 25 untuk jumlah *reads* 10000. Perbedaan ini tidak signifikan, yaitu pada nilai *overlap* 20, terdapat selisih jumlah *node* sebesar 1 (0.0023%) dan jumlah *edge* sebesar 5 (0.091%). Adapun pada nilai *overlap* 25 hanya terdapat perbedaan jumlah *edge* dengan selisih 3 (0.082%). Akurasi atau ketepatan jumlah *node* yang dihasilkan dengan menggunakan sekuens Saccharomyces cerevisiae EC1118 chromosome V adalah sebesar 99.9%. Dari percobaan menggunakan dua sekuens DNA tersebut dapat dihitung akurasi rata-rata pembentukan *node* dan *edge* dengan menggunakan metode perbandingan nilai adalah sekitar 99.99%.

Hal ini menunjukkan bahwa metode yang diusulkan pada penelitian ini (perbandingan nilai) efektif untuk menghasilkan *overlap graph*. Kesamaan atau ketepatan dari sistem ini tak lepas dari nilai suatu *overlap* dan nilai jumlah karakter *overlap* yang digunakan. Nilai dari suatu *overlap*, *prefix* maupun *suffix* diharapkan mempunyai nilai yang unik, yaitu nilai suatu *overlap* dari suatu *suffix* atau *prefix* sama satu sama lain apabila *suffix* dan *prefix* tersebut mempunyai karakter yang terdiri atas basa-basa nitrogen dengan jumlah yang sama dan dengan urutan yang sama. Karena hasil *suffix* dan *prefix* yang dihasilkan dari *reads* dan *reverse complement* sangat banyak maka dapat menyebabkan ada beberapa nilai yang tidak unik, yaitu ada *suffix* atau *prefix* yang mempunyai nilai suatu *overlap* yang sama tetapi sebenarnya mempunyai karakter penyusun yang berbeda baik urutan maupun jumlahnya. Karena itulah digunakan nilai kedua. Nilai kedua(nilai jumlah karakter *overlap*) digunakan ketika sistem menemukan dua nilai suatu *overlap* yang sama. Jadi setiap *suffix* dan *prefix* dibandingkan dengan menggunakan 2 nilai dengan harapan bahwa hasil yang didapat maksimal. Untuk jumlah *reads* sebanyak 5000 dan 10000 buah, sistem dapat memberikan ketepatan sekitar 99.99% (Tabel 5-6).

Tabel 5 Jumlah *Node* dan *Edge* Menggunakan Sekuens Acidiphilium multivorum AIU301 Plasmid pACMV4

Perbandingan nilai			***Pencocokan string***		
Min. *overlap*	Jumlah *node*	Jumlah *edge*	Min. *overlap*	Jumlah *node*	Jumlah *edge*
Jumlah *reads* 5000					
20	4704	19872	20	4704	19872
25	4436	13234	25	4436	13234
30	3469	6606	30	3469	6606
Jumlah *reads* 10000					
20	4291	73896	20	4291	73896
25	9232	49080	25	9232	49080
30	8589	24270	30	8589	24270

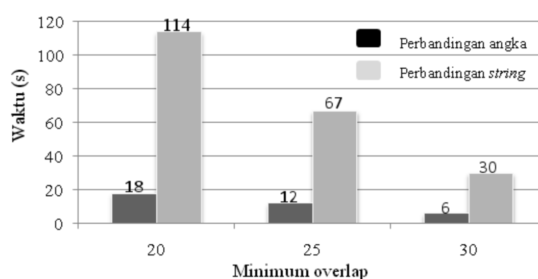
Tabel 6 Jumlah *Node* dan *edge* Menggunakan Sekuens Saccharomyces cerevisiae EC1118 Chromosome V

Perbandingan nilai			*Pencocokan string*		
Min. *overlap*	Jumlah *node*	Jumlah *edge*	Min. *overlap*	Jumlah *node*	Jumlah *edge*
Jumlah *reads* 5000					
20	1236	1466	20	1236	1466
25	819	932	25	819	932
30	422	446	30	422	446
Jumlah *reads* 10000					
20	4237	5488	20	4236	5483
25	3088	3645	25	3088	3642
30	1721	1872	30	1721	1872

Dari Gambar 9 dapat dilihat perbedaan yang signifikan antara waktu eksekusi yang diperlukan antara kedua metode. Waktu eksekusi yang digunakan untuk metode perbandingan nilai secara keseluruhan jauh lebih cepat daripada waktu eksekusi metode pencocokan string. Bahkan dapat dilihat secara keseluruhan waktu yang digunakan metode pencocokan string 5 kali lipat lebih lama dari waktu eksekusi dengan metode perbandingan nilai.

Hal ini dapat terjadi karena metode proses perbandingan nilai menggunakan proses pengurangan sedangkan metode pencocokan string menggunakan proses penyamaan karakter dari masing-masing *prefix* dan *suffix*. Proses pengurangan membutuhkan waktu yang lebih cepat daripada proses penyamaan masing-masing string *suffix-prefix*.

Dengan demikian, penelitian ini memberikan hasil positif karena dapat membentuk bidirected *overlap* graph dengan waktu yang lebih cepat dan dengan ketepatan 99.99%. Metode perbandingan nilai menghasilkan output lebih cepat dibandingkan dengan pencocokan string dikarenakan untuk membandingkan 2 buah string, komputer akan merubahnya dahulu menjadi nilai yang kemudian akan dibandingkan satu sama lain. Dengan metode perbandingan nilai, waktu yang diperlukan untuk mengubah string menjadi nilai dapat dihemat. Sistem ini mempunyai kompleksitas $O(n^2)$. Hal tersebut dikarenakan sistem menggunakan dua pengulangan dalam bentuk *nested loop*.



Gambar 9 Perbandingan Waktu Eksekusi Metode Perbandingan Nilai dan Pencocokan string untuk 5000 input *reads*

Error Collision Rate

Telah ditunjukkan pada pembahasan sebelumnya bahwa metode perbandingan nilai ini dapat menghasilkan ketepatan jumlah *node* dan *edge* yang saling *overlap* hingga 99.99%. Namun demikian, dengan semakin banyaknya data input yang digunakan dapat menyebabkan suatu *collision*. Pada penelitian ini dicari *error collision rate* sebagai acuan sampai jumlah data input berapakah metode ini dapat digunakan dan tetap menghasilkan output dengan ketepatan yang tinggi.

Error collision rate ini dicari dengan membandingkan antara jumlah *edge* yang dihasilkan oleh metode perbandingan nilai dan jumlah *edge* yang dihasilkan oleh metode pencocokan string. Untuk itu dilakukan pengujian untuk mencari nilai ketepatan metode perbandingan nilai sampai dengan jumlah input / masukan 80 000 *reads* dengan panjang setiap string yang dibandingkan satu sama lain memiliki panjang maksimum 34 bp dan panjang minimum adalah 30 bp. Hal ini dikarenakan untuk pengujian yang dilakukan digunakan panjang minimum *overlap* 30 bp. Jumlah input yang dimasukkan berjumlah 80 000 buah. Namun setelah

dilakukan penghilangan *reads* yang redundan, diperoleh *reads* sebanyak 69 936 buah. Karena setiap *reads* akan memiliki *suffix* berjumlah 5 yaitu dengan panjang 34 bp, 33 bp, 32 bp, 31 bp, dan 30 bp, maka secara keseluruhan jumlah string dari *reads* berjumlah 349 680 buah. Akan tetapi karena sistem juga memakai *reverse complement* yang memiliki jumlah yang sama dengan jumlah *reads* unik yang ada, maka jumlah string total yang dibandingkan satu sama lain berjumlah 699 360 buah. Untuk jumlah masukan tersebut, metode perbandingan nilai menghasilkan jumlah *edge* yang sama persis dengan metode pencocokan string yaitu berjumlah 336 414 buah. Sampai dengan jumlah input 80 000, metode perbandingan nilai memiliki nilai *error collision rate* 0%, atau tidak ada *collision*, dengan menggunakan sekuens Acidiphilium multivorum AIU301 plasmid pACMV4. Dengan melihat hasil percobaan sebelumnya, dengan menggunakan sekuens yang berbeda, yaitu Saccharomyces cerevisiae EC1118 chromosome V, kemungkinan diperoleh nilai *error collision rate* tidak lebih dari 1%.

4. KESIMPULAN

Penelitian ini berhasil membentuk *bidirected overlap graph* dengan menggunakan teknik perbandingan nilai. Diperoleh informasi pula bahwa semakin kecil nilai *overlap* antar *reads* yang dipilih akan menyebabkan semakin lama waktu eksekusinya. Dari segi ketepatan jumlah *node* dan *edge* yang dihasilkan, metode perbandingan nilai memiliki akurasi sebesar 99.99% hampir sama dengan menggunakan metode pencocokan string. Adapun dari segi waktu eksekusi, metode perbandingan nilai lebih efisien dibandingkan dengan pencocokan string, yaitu lima kali lebih cepat. Dengan demikian, metode perbandingan nilai cukup efektif dan efisien untuk digunakan sebagai alternatif teknik dalam pembentukan *bidirected overlap graph.*

DAFTAR PUSTAKA

ABEGUNDE, T., 2010. Comparison of DNA sequence assembly algorithms using mixed data sources [tesis]. Saskatoon (CA): University of Saskatchewan.

BANKEVICH, A. et. al. 2012. SPAdes: a new genome assembly and its application to single cell sequencing. Journal of Computational Biology, vol 19, no. 5, pp 455-477

BATZOUDLOU, S., JAFFE, D.B., STANLEY, K. et. al., 2002. ARACHNE: A whole genome shotgun assembler. Genome Res. 12: 177-189

BOZA, V., Brejova, B., Vinar, T. 2014. GAML: genome assembly by maximum likelihood. Algorithm in Bioinformatics, Lecture Notes Bioinformatics, 8701, pp 122-134.

BRENNER, S., JOHNSON, M., BRIDGHAM, J., GOLDA, G., LLOYD D.H., JOHNSON, D., LUO, S.J., McCURDY, S., FOY, M., EWAN, M., et al., 2000. Gene expression analysis by massively parallel signature sequencing (MPPS) on microbead arrays. Nat Biotechnol. 18:630-634.

CHAISSON, M., PEVZNER, P., TANG, H.. 2004. Fragment assembly with short *reads*. Bioinformatics. 20(13):2067-2074.

COMMINS, J., TOFT, C., FARES, M.A., 2009. Computational Biology Methods and Their Application to the Comparative Genomics of Endocellular Symbiotic Bacteria of Insects. Biol Proced Online. 11:52-78. doi: 10.1007/s12575-009-9004-1.

HERNANDEZ, D., FRANCOIS, P., FARINELLIi, Østerås M., SCHRENZEL, J., 2008. De novo bacterial genome sequencing: Millions of very short *reads* assembled on a desktop computer. Genome Res. 18:802-809

HUANG, X., WANG, J., ALURU, S., YANG, S., HILLIER, D., 2003. PCAP: A whole-genome assembly program. Genome Res. 13: 2164-2170

KAE, H. 2003. Genome Project: uncovering the blueprints of biology. The Science Creative Quarterly.

KUSUMA, W.A., ISHIDA, T., AKIYAMA, Y., 2011. A combined approach for de novo DNA sequence assembly of very short *reads*. IPSJ Transaction on Bioinformatics. 3(10):21-33. doi: 10.2197/ipsjtbio.4.21.

MANBER, U., MYERS, E.W., 1993. *Suffix* arrays: a new method for on-line string searches. SICOMP. 22(5):935-948.

MEDVEDEV, P. et. al. 2007. Computability of Models for Sequence Assembly. Algorithms in Bioinformatics, Lecture Notes in Computer Science, vol. 4645, pages 289-301.

MULLIKIN, J.C., NING, Z., 2003. The Phusion assembler. Genome Res. 13:81-90

MYERS, E.W., SUTTON, G.G., DELCHER, A.L., et. al., 2000. A whole-genome assembly of Drosophila. Science. 287: 2196-2204

POP, M., 2009. Genome assembly reborn: recent computational challenges. Briefing in Bioinformatics. 3(7):47-54

RICHTER, D.C., OTT, F., AUCH, A.F., SCHMID, R., HUSON, D.H., 2008. Metasim-A sequencing simulator for genomics and metagenomics. PLoS ONE. 3(10): e3373

SIMPSON, J. T. and POP, M. 2015. The Thory and Practice of Genome Sequence Assembly. Annu. Rev. Genomics Hum. Genet. 2015. 16:153–72

ZHOU, X., REN, L., MENG, Q., LI, Y. YU, Y., YU, J., 2010. The next-generation sequencing techmology and application. Protein Cell. 1(6): 520-536