PaDeNA: Parallel De Novo Assembler

Technical Design Document – Microsoft Biology Foundation

Version 1.0 - June, 2010

Abstract

This document describes the ParallelDeNovoAssembler (PaDeNA) class, an implementation of a DNA sequence assembler algorithm. This assembler algorithm falls under the class of DeNovo assembly algorithms that are based on the de Bruijn graph.

Contents

[Overview 2](#_Toc253398615)

[Design 3](#_Toc253398616)

[ParallelDeNovoAssembler class [namespace MBF.Algorithms.Assembly] 3](#_Toc253398617)

[Step 1: k-mer generation [namespace MBF.Algorithms] 6](#_Toc253398618)

[Step 2: Graph Construction [namespace MBF.Algorithms.Assembly.Graph] 7](#_Toc253398619)

[Steps 3,4: Error Correction [namespace MBF.Algorithms.Assembly.PaDeNA] 9](#_Toc253398620)

[Step 5: Building Contigs [namespace MBF.Algorithms.Assembly.PaDeNA] 12](#_Toc253398621)

[Step 6: Building Scaffolds [namespace MBF.Algorithms.Assembly.PaDeNA.Scaffold] 13](#_Toc253398622)

[Appendix 26](#_Toc253398623)

**Disclaimer:** This document is provided “as-is”. Information and views expressed in this document, including URL and other Internet Web site references, may change without notice. You bear the risk of using it.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes.

© 2010 Microsoft Corporation. All rights reserved.

Microsoft, Visual Basic, Visual Studio, and Windows are trademarks of the Microsoft group of companies.  All other trademarks are property of their respective owners.

Document History

|  |  |
| --- | --- |
| Date | Change |
| February 12, 2010 | First publication |

# Overview

DNA sequencing machines are increasing their throughput at an exponential rate.  They typically employ techniques of splitting the DNA sequence into 'short read' fragments, which allows them to sequence multiple lanes of DNA at a time to a high degree of accuracy.  Short read lengths are typically 25-100bp, though there are some efforts underway attempting to perform longer strands of reads as well.  Once the DNA fragments are sequenced, they need to be re-assembled into a fully contiguous DNA strand to represent the original specimen.

There are two broad approaches to this problem:

* Use of a map, or ordered set of markers along a sequence (assembly with reference sequence)
* Use of many overlapping sequences (high coverage) to infer ordering directly from the sequences themselves (de-novo assembly)

De Novo assembly in turn has two popular approaches:

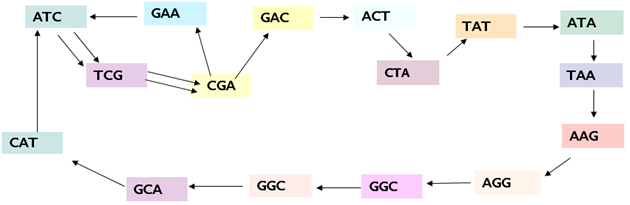
* Overlap-layout-consensus
* De Bruijn graph based approaches that model the assembly problem into a (eulerian) graph path problem.

Most of the widely used DeNovo assembly algorithms are based on de Bruijn graphs.

De Bruijn Graph is a directed graph with vertices that represent sequences of symbols from an alphabet, and edges that indicate where the sequence may overlap.

Example: The strand ATCGACTATAAGGCATCGAA

* De bruijn graph has vertices of length 3 (k-mers)
* There is a directed edge between two vertices if there is a (k+1)-mer such that it spans both the vertices. (Adjacent nodes always have an overlap of (k-1) bases)



Assembly algorithms such as ABySS, Velvet, EULER-SR use the de Bruijn graph. At a high level, these algorithms also share steps.

# Design

**Note:** For each step, there is a section ‘Parallelization’ that mentions the parts of that step that are parallelizable. For every parallelized step, wherever we need to collate information, we will be using thread-safe data structures provided in System.Collections.Concurrent namespace.

## ParallelDeNovoAssembler class [namespace MBF.Algorithms.Assembly]

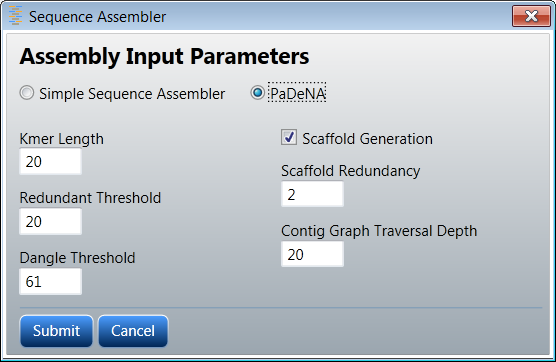
ParallelDeNovoAssembler class implements assembler algorithm. This assembler algorithm falls under the class of denovo assembly algorithms that are based on de Bruijn graph. It supports the assembly of DNA sequences only.

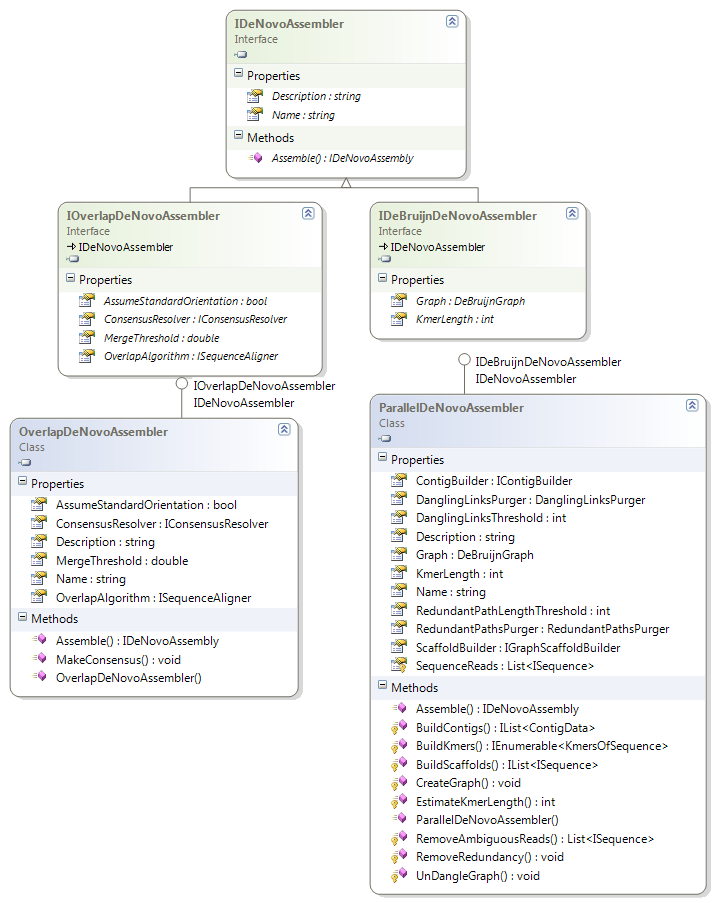
In our implementation of sequence assembler (PaDeNA), we are combining ideas from multiple sources:

* The first five steps (building contigs from input sequence reads) of our de-novo implementation are borrowed from ABySS algorithm as described in their publication: <http://genome.cshlp.org/content/19/6/1117.full>
* For the last step of de-novo implementation (generation of scaffolds from contigs and paired reads), we combine ideas mainly from 2 public sources:
* ABySS algorithm
* Greedy Path Merging Algorithm  
   <http://research.janelia.org/myers/Papers/greedy.path.merging.pdf>

List of Inputs for assembly process in PaDeNA:

* List of sequence reads.
* Paired read information: this gives mate-pair relationship between sequence reads, which represents the mapping between forward and backward sequence reads.
* User Input parameters:
* Kmer Length:
  + Each input sequence is transformed into sequences of length ‘k’ (k-mers).
  + An estimate input value should be based on the lengths of the input reads. Roughly, for estimation the range of allowed kmer values is from the length of longest sequence divided by 2, to length of shortest sequence. A median value in this range should be chosen as a first approximation.
  + If the length of the longest sequence divided by 2 is longer than the length of the shortest sequence, the length of the shortest sequence divided by 2 should be used.
* Scaffold Generation
  + Select this box when input sequences contain paired-end read data.
* Redundant Threshold
  + Threshold for redundant path error removal step
  + First order estimate is to use 1.5 x Kmer Length.
* Dangle Threshold
  + Threshold for dangling link error removal step
  + First estimate is to use Kmer Length +1
* Scaffold Redundancy
  + Scaffolds are an ordered sequence of contigs
  + Redundancy is the number of mate pairs to be considered to make a mate pair connection between contigs.
* Contig Graph Traversal Depth
  + Number of levels (depth) of traversal in the contig graph before backtracking.



Class Diagram:  
Namespace MBF.Algorithms.Assembly.PaDeNA 

#### Explanation of Class Diagram:

At the top is the interface for all de novo assemblers – IDeNovoAssembler. This is futher derived by 2 interfaces that represent that two broad classes of assembly process:

* IOverlapDeNovoAssembler: interface for overlap-layout-consensus approach.
* IDeBruijnDeNovoAssembler: interface for De Bruijn graph based De Novo Assembly approach.

Class ParallelDeNovoAssembler represents the class that implements our de-novo algorithm.

* Fields that hold values of input parameters (\_kmerLength, \_dangleThreshold, \_\_redundantPathLengthThreshold, \_sequenceReads)
* Each step in the de novo algorithm is handled by a separate class. ‘ParallelDeNovoAssembler’ declares a field that refers to the class implementing each of these steps for this algorithm: (\_danglingLinksPurger, \_redundantPathsPurger, \_contigBuilder, \_scaffoldBuilder). More details on each of these classes are further below
* Method ‘Initialize’ takes care of initializations and setting up things for assembler to start:
* In case \_kmerLength is not set by user, an estimate based on the lengths of the input reads is made to arrive at a default value. Roughly, for estimation the range of allowed kmer values is from (length of longest sequence/2) to length of shortest sequence. A median value in this range is chosen for \_kmerLength.
* Threshold values that are uninitialized are set to default values based on the \_kmerLength:  
  \_dangleThreshold = \_kmerLength + 1  
  \_redundantPathLengthThreshold = (int)Math.Ceiling(1.5f \* \_kmerLength)
* Graph modifying classes such as \_danglingLinksPurger, \_redundantPathsPurger are set to default values based on the particular implementation that the algorithm uses for each step.
* Additionally, this method also removes any reads that have ambiguous characters in their sequence.
* Implements interface method**:**  
  public IDeNovoAssembly Assemble(List<ISequence> inputSequences)

This method calls sequence of steps for de novo assembly, and returns **PaDeNAAssembly** object that holds the result of assembly process.

* Each step of the assembly process is carried out by a protected method. This enables deriving classes to override specific steps with custom implementation:

Step 1: BuildKmers()  
Step 2: CreateGraph()  
Step 3: UnDangleGraph()  
Step 4: RemoveRedundancy()  
Step 5: BuildContigs()  
Step 6: BuildScaffolds()

#### Parallelization:

There is no parallelization at this level. Parallelization is taken care of within each step in the assembly process (explained for each step).

#### Output:

Results of assembly process are returned within a **PaDeNAAssembly** object:

* List of assembled contig sequences
* List of assembled scaffold sequences

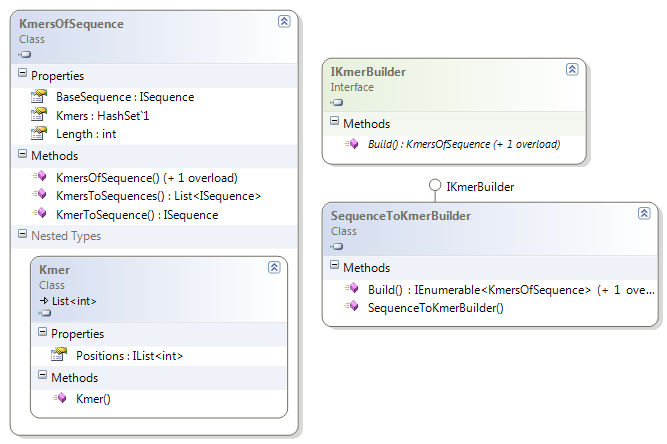
## Step 1: k-mer generation [namespace MBF.Algorithms]

The first step is to break each input sequence read into sequences of length ‘k’ (k-mers).

**Note:** This set of classes reside in the MBF namespace, since they are used by both PaDeNA and DiffSeq (EMBOSS).

#### Input:

List of input sequence reads, k-mer length ‘k’

Class Diagram:  
Namespace MBF.Algorithms   


#### Explanation of Class Diagram:

* Each instance of ***KmersOfSequence*** class represents one input sequence and the list of k-mers that are part of this sequence.
* Nested Type Class ***Kmer:*** Represents a k-mer within base sequence. Storing the ISequence for each Kmer is very wasteful of memory, since there is a lot of overlap between k-mer sequences. Hence, we only store the list of start positions with respect to the base sequence here.
* ***KmersOfSequence***:
* Stores input sequence in ‘baseSequence’.
* Holds set of Kmer class instances that represent the k-mers associated with the baseSequence. The k-mers are defined in terms of starting position in the base sequence. The k-mers along with the base sequence thus completely define the set of k-mers associated with a given sequence.
* Since we store only start positions inside Kmer, Kmer by itself cannot construct the ISequence / string for the k-mer. We provide helper methods in KmersOfSequence to allow access to k-mers in string / sequence form:  
    
  public ISequence KmerToSequence(Kmer k)  
  public List<ISequence> KmerToSequences()

We follow the Builder design pattern here. The process of building k-mers from a sequence is abstracted from its representation, to enable different implementations of the construction process.

* Interface ***IKmerBuilder***: provides interface for building k-mers from sequence.
* Class ***SequenceToKmerBuilder*** implements above interface. Given a sequence, it slides a window of size \_length along the sequence, extracting the sequence items within the range to construct k-mers.

Implements methods for building k-mers from one sequence or a list of input sequences. There are 2 overloads – one that works with a list of sequences and other that works with a single sequence.

#### Parallelization:

For each input sequence read, a separate ‘Task’ (.NET parallel construct) is spawned. Each task will populate and return the corresponding KmersFromSequence.

#### Output:

List<KmersOfSequence>

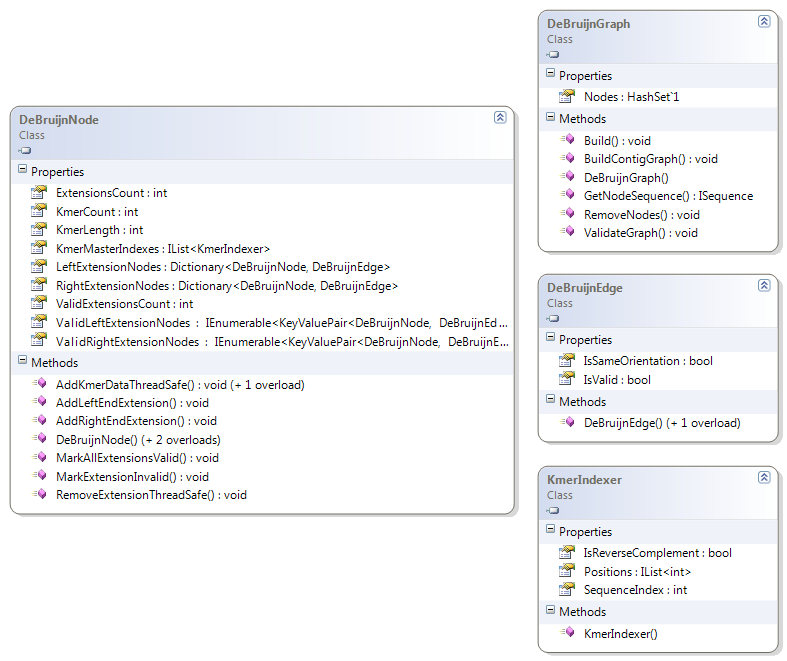
For each input sequence read, a KmersOfSequence is constructed that provides the set of k-mers that are associated with that sequence.

## Step 2: Graph Construction [namespace MBF.Algorithms.Assembly.Graph]

The next step in assembly process is to construct de Bruijn graph based on the k-mers. This step computes the nodes and edges of the graph to be used in the rest of the assembly process.

#### Input:

List of sequence k-mers

Class Diagram:   
Namespace MBF.Algorithms.Assembly.Graph  


#### Explanation of Class Diagram:

For de bruijn graph, we again follow the Builder design pattern template for graph construction. Three classes are provided:

* Class ***DeBruijnNode*** represents a node in the graph.
* Each node is associated with a k-mer. Node holds starting position of kmer associated with node. This index references into base sequence stored in the DeBruijnGraph graph containing this node.
* LeftExtensionNodes, RightExtensionNodes represent the nodes that are connected to the left and right of current node, and the associated edge.
* A left-end extension edge will be added between nodes A and B if the last (k-1) sequence items of node B is same as the first (k-1) items in node A. Similarly, a right-end extension edge will be added from node A to node B, if the last (k-1) sequence items in sequence A is same as the first (k-1) items in sequence B.
* \_countNormalOrientation, \_countReverseComplement stores number of times the k-mer associated with this node or its reverse complement occurs in input sequence list. Property ‘KmerCount’ exposes the sum of these values.
* Methods are provided to allow addition / removal of left / right extension nodes, and allow updation of associated count information. These methods are used during building of the graph.
* Class ***KmerIndexer*** maintains index of k-mer as sequence index, number of times the k-mer occurs within that sequence and orientation information (reverse complement or not). This information is combined with list of positions to get the exact location of k-mer in reads. This k-mer to read map is used in the 6th step of de-novo algorithm.
* Class ***DeBruijnGraph*** represents graph structure formed out of the DeBruijnNodes.
* Holds a set of graph nodes in \_kmerNodes
* Has a field \_baseSequence that stores segmented sequence that is built from concatenation of all the input sequences. Positions within KmerIndexer in DeBruijnNode references into this sequence. The method GetNodeSequence is used to construct the sequence of a node from the base sequence and index information
* Methods are provided to enable graph modification through node removal. (RemoveNodes)
* This method deletes the node from graph.
* Build() creates a node in the graph for each k-mer and also populates adjacency information between nodes.
* BuildContigGraph() is part of the 6th step. This builds a contig graph from the kmer graph for scaffold generation to work with.

#### Parallelization:

The node creation and adjacency generation are parallelized. For each KmersOfSequence, a separate task is started, that takes care of creating nodes for kmers in that sequence. Following node generation, each k-mer node checks to see which all of its neighboring k-mers exist. Tasks can thus be used here again – a separate task is set up for each k-mer node.

#### Output:

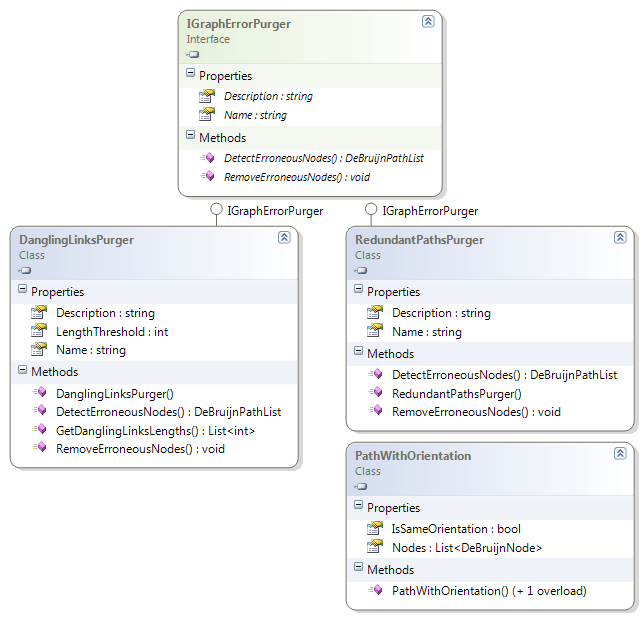
Complete de-Bruijn graph with nodes and edges.

## Steps 3,4: Error Correction [namespace MBF.Algorithms.Assembly.PaDeNA]

These set of classes provide functionality for graph-based error correction. This corresponds to trimming, bubble popping steps (Steps 3, 4) in ABySS.

#### Input:

DeBruijnGraph graph

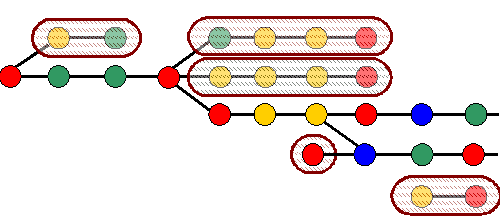
Class Diagram:  
Namespace MBF.Algorithms.Assembly.PaDeNA   


#### Explanation of Class Diagram:

IGraphErrorCorrection provides interface for classes that perform error corrections based on graph structure. It provides abstract methods for detection of nodes that fall under some error criteria (DetectErroneousNodes) and subsequent removal of these nodes from the graph (RemoveErroneousNodes).

Two types of error correction are currently supported. Each of these classes implement above interface.

* DanglingLinksPurger: This class implements error removal algorithm that detects and removes dangling ends in the graph. This implements ABySS’s step for trimming dead-end branches.

  
**Figure: At the end of this step, all graph parts inside hashed ovals are deleted** [**[4]**](#_References)

* **LengthThreshold** stores threshold value (input parameter) required for identification of dangling links
* **GetDanglingLinksLengths** method is used to first determine the lengths of the dangling links that exist in the graph. This information is used to remove dangling links in increasing order of their lengths.
* **DetectErroneousNodes** method identifies nodes of the graph that fall on dangling links, whose length is lesser than a given threshold.
* **RemoveErroneousNodes** method removes nodes identified in previous step from the graph, while updating adjacent node’s edge information.
* RedundantPathsPurger: Detects and removes redundant paths. This implements ABySS’s step for popping bubbles.

|  |  |
| --- | --- |
| **Two examples of input de Bruijn graphs to this step** [**[4]**](#_References) | **Output at end of this step,  for the two input graphs to the left** [**[4]**](#_References) |

* **\_lengthThreshold** stores threshold input parameter
* **DetectErroneousNodes** method identifies branches in the graph that have a common start and end points. Using graph parameters, it then identifies branches (and corresponding nodes on those branches) that have to be deleted.
* **RemoveErroneousNodes** method deletes nodes identified in the previous step from the graph.

Class ***PathWithOrientation***: Each path has an orientation, which gets updated depending on the orientation of the graph edge being added. This class holds a list of path nodes, along with the orientation.

#### Parallelization:

Both detection and deletion methods can be parallelized. Separate tasks will be set up for each node that can potentially lead to a dangling link or redundant path. Each task then tries to extend these and check whether it can be removed. During the deletion step, tasks again deal with sets of nodes to be removed – here updating the adjacent nodes of the deletion is parallelized.

#### Output:

Error corrected DeBruijn Graph

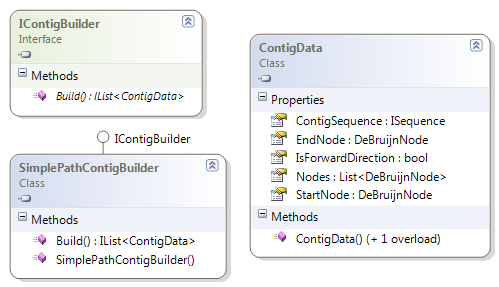
## Step 5: Building Contigs [namespace MBF.Algorithms.Assembly.PaDeNA]

This maps to the ‘Vertex Merging’(Assembly – SET) step in ABySS algorithm:

* Remove ambiguous edges (whenever the vertex associated with an edge has multiple incoming / outgoing edges the edge is ambiguous) from the graph.
* Merge vertices along unambiguous edges to generate initial contigs. Build sequences along simple paths in the graph to create contigs.

#### Input:

De Bruijn graph

Class Diagram:  
**Namespace MBF.Algorithms.Assembly.PaDeNA**

#### Explanation of Class Diagram:

* Interface ***IContigBuilder*** provides the framework for building contigs from the graph. It provides an abstract method that build contigs from the de bruijn graph.
* Class ***SimplePathContigBuilder*** implements above interface. Build method identifies simple paths in the graph (paths that have a single source and single sink). It then collapses these to generate corresponding contig sequences.

This exposes the contigs as a list of de bruijn nodes (ContigNodes). This is used to reduce re-work in the 6th step of the algorithm.

* Class ***ContigData*** represents a contig. It holds the contig sequence, the corresponding list of nodes, and direction information (whether contig is built by extending in right or left direction).

#### Parallelization:

For every node in the graph that has a single out-going edge, a separate task is started. Each task then check if the path from that node is simple, and if yes, construct and populate the corresponding contig data.

#### Output:

List of contigs

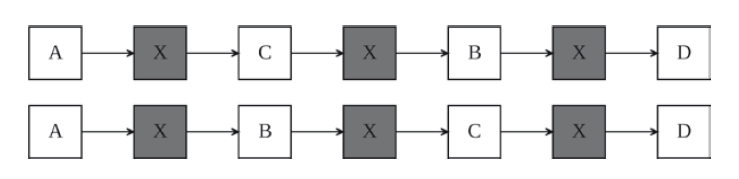
## Step 6: Building Scaffolds [namespace MBF.Algorithms.Assembly.PaDeNA.Scaffold]

#### Background

Challenge of genome sequencing lies in the fact that only a small portion of the genome encodes genes, and that these genes are often surrounded by repetitive DNA which is comparably information poor. Large repeats can cause ambiguity with fragment assembly and thus pose the greatest challenge when assembling genomic data.

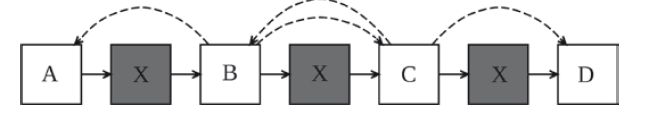
#### Scenario

We know that regions B and C are surrounded by identical repetitive regions X and that both regions lie between regions A and D, but without more information, it is impossible to know the correct ordering of B and C. [[3]](#_References)



Using paired end sequencing, where DNA fragments of known approximate size are generated and sequenced from both ends. Information about these pairs such as average fragment size and the orientation of reads with respect to the read pair is included in the assembly process. If the distance between the paired ends, known as the insert size, is large enough, then there is a high probability that repeats will be spanned by a pair of reads (or mates) which can remove ambiguity from the assembly.

For example, if paired end data is analyzed and region B is found to have mates in regions A and C but not D, while region C has mates in regions B and D but not A, then an ordering can be inferred.



**Where dashed edges indicate paired read information.** [**[3]**](#_References)

#### Input:

* List of sequence reads
* Graph (DeBruijnGraph graph)
* List of Contigs

Class Diagram:  
   


#### Explanation of Class Diagram:

Interface ***IScaffoldBuilder*** provides the framework for building scaffolds (ordered sequence of contigs).

It provides a method to BuildScaffold to build scaffolds.

Class ***GraphScaffoldBuilder*** implements above interface.

* Fields that hold values of input parameters (\_redundancy, \_depth, \_contigData, \_contigGraph, \_kmerLength )
* Each step in the de novo algorithm is handled by a separate class. ‘***GraphScaffoldBuilder***’ have fields that refer to the class implementing each of these steps for this algorithm and their return types. More details on each of these classes are further below
* Method ‘Initialize’ takes care of initializations initializes filed variables with input parameters after validataion.
* Implements interface method**:**

public IList<ISequence> BuildScaffold(IList<ISequence> reads, DeBruijnGraph graph, IList<ContigData> contig)

This method calls sequence of steps for Scaffold Generation, and returns IList<ISequence> object that holds the sequence of Scaffold.

* Each step of the assembly process is carried out by a protected method. This enables deriving classes to override specific steps with custom implementation:

Step 1: MapPairedReads()  
Step 2: ReadContigMap()  
Step 3: ModifyGraph()  
Step 4: FilterReadsBasedOnOrientation()  
Step 5: CalculateDistanceBetweenContigs()  
Step6: TracePath()  
Step 7: PathPurger()  
Step 8: GenerateScaffold()

* Properties:

Redundancy: Number of mate pairs to be considered to make a mate pair connection between contigs.

#### Parallelization:

Alignment of reads to contigs and mapping of reads to paired reads is parallelized by at step levels too. These tasks are independent of each other.

#### Output:

List of scaffold sequences (List<ISequence>)

### Step 1: Map Reads to Paired Reads

In pair reads sequencing, the sequence of insert is read from both directions with have known spacing and orientation.



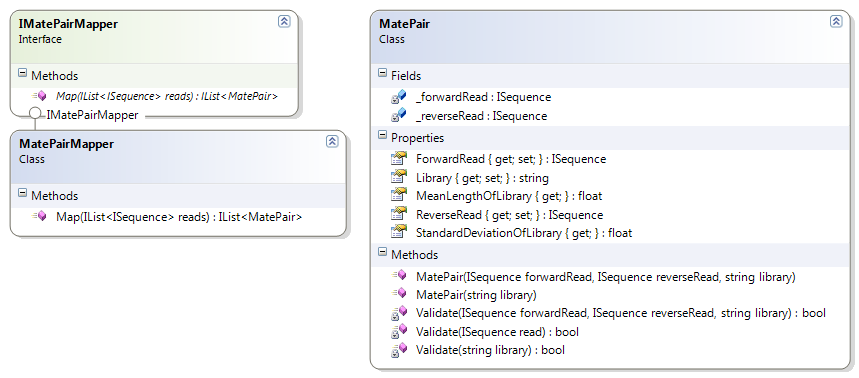
Sequence Vector. A schematic drawing of a sequencing vector, such as a BAC (Bacterial Artificial Chromosome). The insert can be a genomic fragment, or ancDNA (for EST sequencing). In both cases sequencing from each end will produce a read pair that can provide additional information for assemblers. [[1]](#_References)

#### Input:

IList<ISequence> : Input List of reads

#### Output:

IList<MatePair> : List of Mate pairs

Class Diagram:  


#### Explanation of Class Diagram:

* Read Pair class store read pairs with library information and Map Read Pairs class converts input list of reads into paired reads using information available in FASTA header.
* Interface ***IMatePairMapper*** provides the framework for mapping reads to mate pairs.

It provides a method to Map to map reads to mate pairs.

* Class ***MatePairMapper*** implements above interface.
* Supported mate pair formats:

>chrI0.X1:abc  
ATGC  
>chrI0.Y1:abc  
TACG  
>chrI0.F:abc  
ATGC  
>chrI0.R:abc  
TACG  
>chrI0.1:abc  
ATGC  
>chrI0.2:abc  
TACG

Where:

X1,F,1 denotes forward reads and Y1,R,2 denotes reverse reads

abc denotes library name

chrI0 is the sequence id

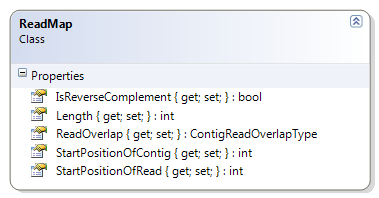
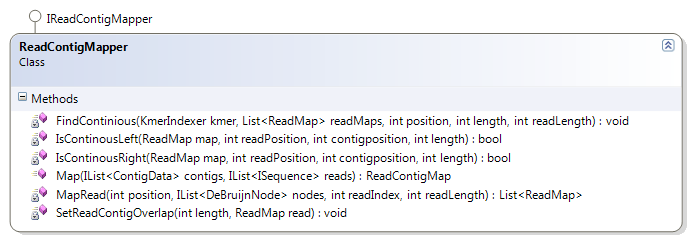
### Step 2: Map Reads to contigs

#### Input:

IList<ISequence>: Input List of reads, IList<ContigData> : Input List of contigs

#### Output:

ReadContigMap

Class Diagram:   
  
  
  
  


#### Explanation of Class Diagram:

* ReadContigMap class stores mapping between reads and contigs.
* Interface ***IReadContigMapper*** provides the framework for mapping reads to contigs.

It provides a method to Map to map reads to contigs.

* Class ***MatePairMapper*** implements above interface. Map method calls different methods to generate alignments between reads and contigs using information stored in DeBruijn Nodes.

#### Parallelization:

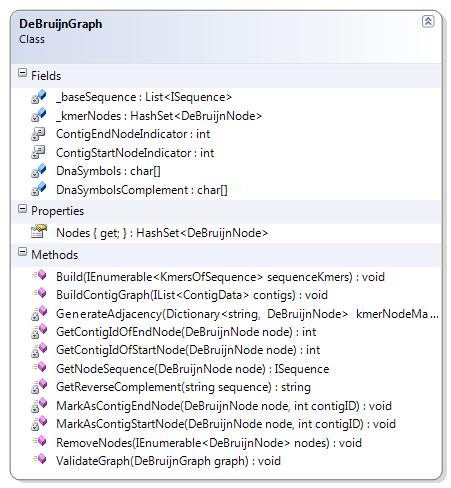
Each contig is mapped separately to reads.

### Step 3: Modify Graph:

#### Input:

IList<ContigData>: Input List of contigs

#### Class Diagram:



Method BuildContigGraph modifies the existing graph into contig overlap graph using information available in contig data. Now, each node in graph represents single contig.

#### Parallelization:

Nodes of Graphs are updated independently and are merged together to form contig nodes.

### Step 4: FilterReadsBasedOnOrientation

For each pair of contigs, perform clustering of mate pairs based on orientation of contigs (Four possible clusters as shown in fig below). Cluster with largest numbers of mate pairs will be used for further steps and rest all mate pairs will be removed from consideration. If the cluster with largest number of mate pairs is below threshold (user defined: Default =2), then no contig edge will be created and all mate pairs will be removed from further consideration. If two clusters have equal number of mate pairs and both cluster passes threshold, then we retain mate pairs from both clusters for further consideration.

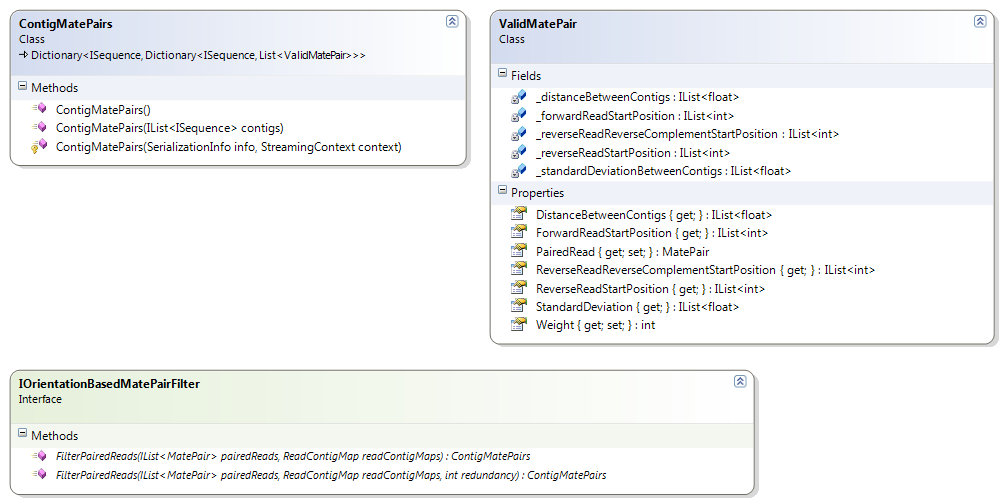
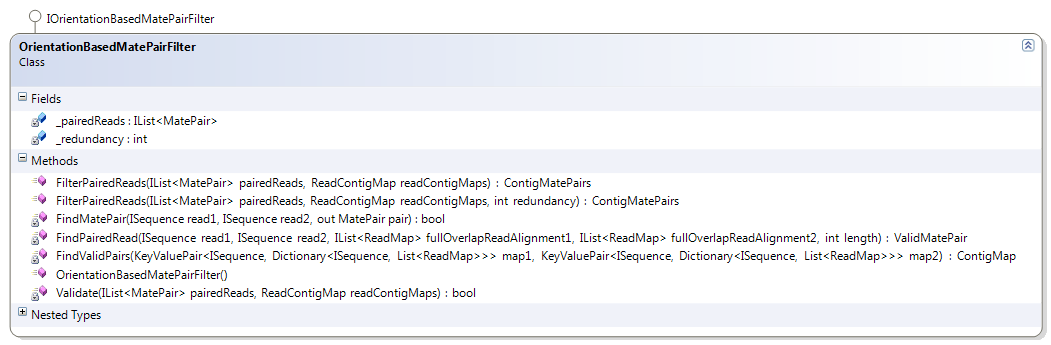
  
**Four possible orientation of a contig pair** [**[2]**](#_References)

#### Input:

ReadContigMap and IList<matePair>:

#### Output:

ContigMatePairs

Class Diagram:  
  
  


* Interface ***IOrientationBasedMatePairs*** provides the framework for mapping paired reads to contigs and filtering paired reads based on orientation of mate pairs.
* Class ***OrientationBasedMatePairFilter*** implements above interface. It provides a method to FilterPairedReads to map paired reads to contigs and filter paired reads based on orientation.

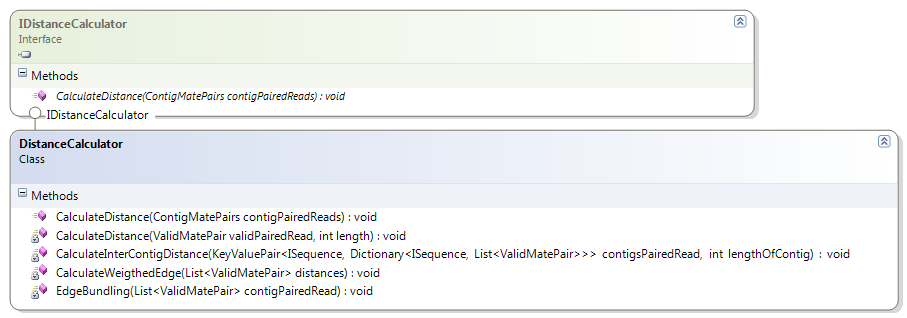
#### Parallelization:

For Each pair of contig the mate pair map and filtering is performed separately.

### Step 5: CalculateDistanceBetweenContig

#### Input:

ContigMatePairs

Class Diagram:  


* Interface ***IDistanceCalculator*** provides the framework for paired read distances between contigs.
* Class ***OrientationBasedMatePairFilter*** implements above interface. It provides a method to calculate to calculate distance between contigs using paired read connections.

#### Parallelization:

For each pair of contigs distance calculation is performed independently.

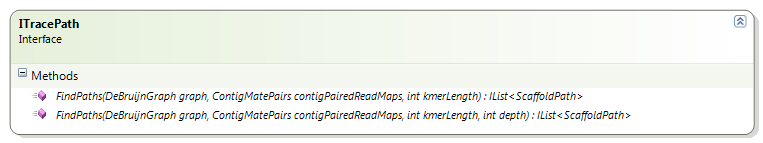
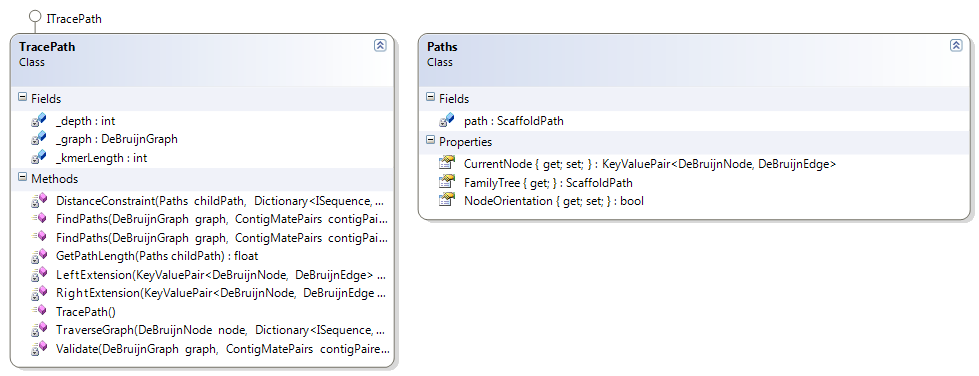
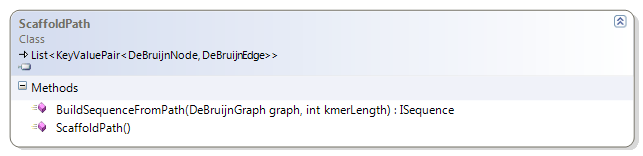
### Step 6: TracePath

#### Input:

ContigMatePairs and DeBruijnGraph

#### Output:

IList<ScaffoldPaths>

Class Diagram:  
  
  


* Interface ***ITracePath*** provides the framework for traversing in contig overlap graph using paired read connections between contigs.
* Class TracePath implements above interface. It provides a method find paths to find paths in graph using depth first search. It returns list of paths.
* Paths is a internal class which stores information about path followed while performing depth first search.

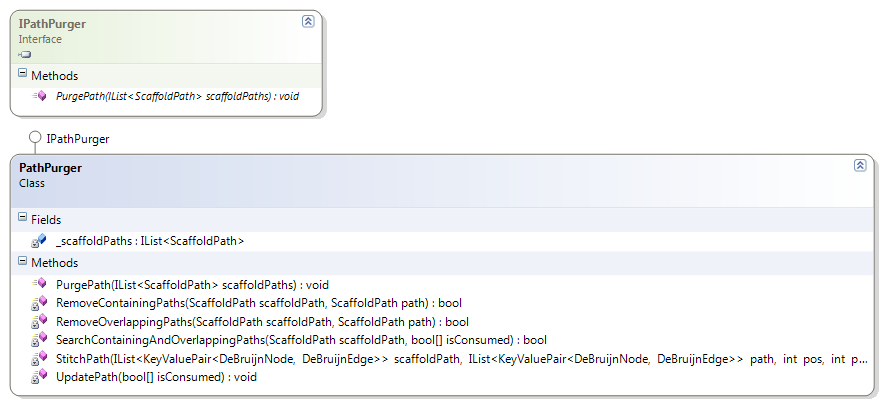
#### Parallelization:

Depth first search is performed starting from nodes having forward contigs independently.

### Step 7: PathPurger

#### Input:

IList<ScaffoldPaths>

Class Diagram:  


* Interface ***IPathPurger*** provides the framework for removing redundant paths and stitching paths to make longer paths.
* Class PathPurger implements above interface. It provides a method PurgePath to remove redundant paths and stitch paths.

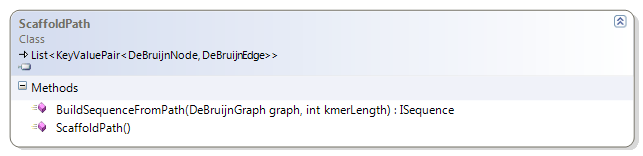
### Step 8: GenerateScaffold

#### Input:

IList<ScaffoldPaths>

#### Output:

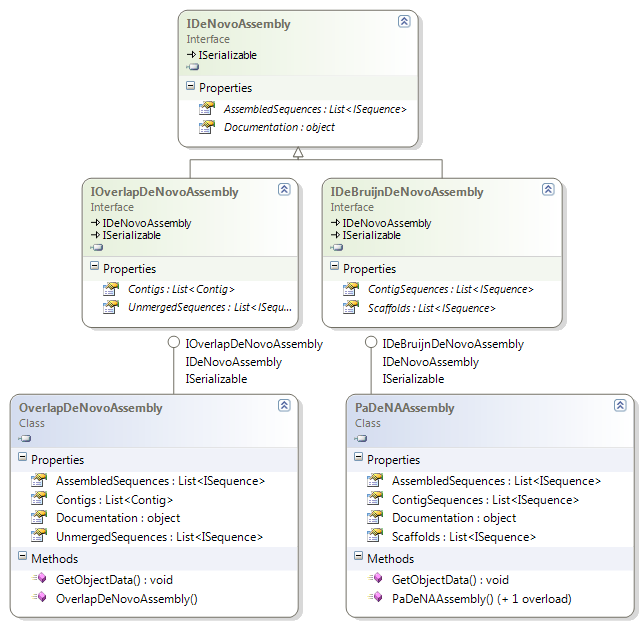
IList<ISequence>

Class Diagram:  


BuildSequenceFromPaths: Build sequence from paths using contig sequences in nodes of paths.

#### Output:

The structure of output is very similar to the way Assembler structure was organized.

Namespace MBF.Algorithms.Assembly   


At the top is interface for output from all de novo assemblers – IDeNovoAssembly. This gives list of assembled sequences. This is derived by 2 interfaces that represent output from the two types of assembly process:

1. IOverlapDeNovoAssembly: interface for output of overlap-layout-consensus based assembly.

* This additionally exposes the list of contigs (along with information about the sequences merged for each contig sequence) and list of unmerged sequences.
* This is implemented by class ***OverlapDeNovoAssembly*** that is the result of overlap-based simple sequence assembly (previously implemented **OverlapDeNovoAssembler**).

2. IDeBruijnDeNovoAssembly: interface for output of De Bruijn graph based assembly.

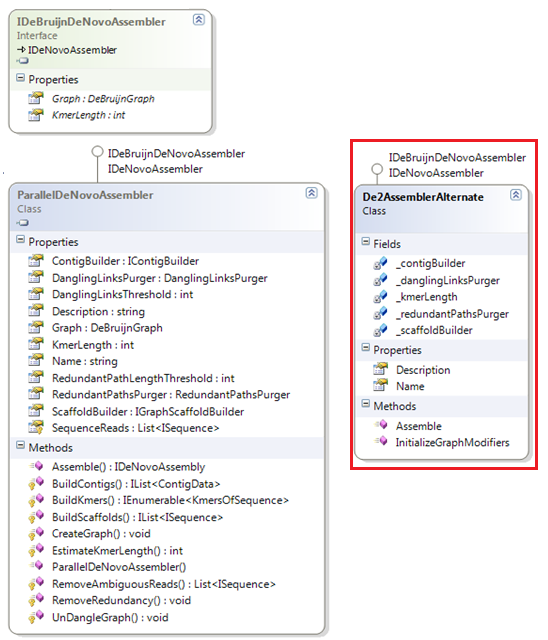
* This exposes a list of contigs and a list of scaffold sequences.
* Class ***PaDeNAAssembly*** is the output of PaDeNA assembly process.

# Appendix

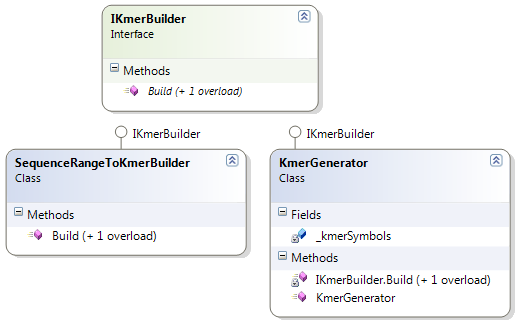
Once we have an end-to-end implementation of de novo assembler, alternate implementations for different steps can be added. This gives the user multiple options when he/she wants to do the assembly process using plug-and-play model for the different steps. Parts of class diagram below are highlighted in red. These show how the alternate implementations fit into the current design.

For example, we will consider implementing the steps similar to those described in the Velvet algorithm paper. Velvet has different algorithms for error corrections, and also combines the steps in a slightly different manner. The following diagrams show how to extend the current design to accommodate these differences.

The following figure (class De2AssemblerAlternate) illustrates how the design will be extended if we implement another algorithm that requires the steps to be combined or called in a different order.

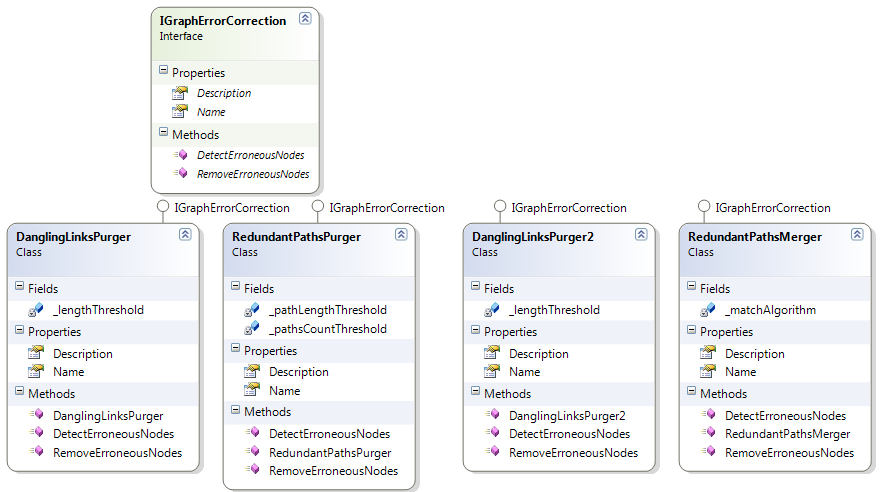
class De2AssemblerAlternate   


In Kmer building stage, alternate implementation can be provided. For example, one other popular approach for this step is to generate the complete list of all possible k-mers for that length – that is, for DNA generate (4^l) strings (where l = length of k-mer). Once this is done, check for existence and positions of occurrence for these k-mers within input sequence(s).



For error correction methods, Velvet follows approaches different from ABySS. Velvet, for example, uses a pairwise algorithm for identifying and merge redundant paths.

The following figure illustrates how alternate implementations can be added for these error correction measures. In the same manner, any other graph correction measures can be added and used.



# References

1. K. Scheibye-Alsing, S. Hoffmann, A. Frankel, P. Jensen, P. F. Stadler, Y. Mang, N. Tommerup, M. J. Gilchrist, A. B. Nygård, S. Cirera 2009. Sequence assembly. Computational Biology and Chemistry, Vol. 33, No. 2. (April 2009), pp. 121-136.
2. Mihai Pop, Daniel S. Kosack and Steven L. Salzberg. Hierarchical Scaffolding With Bambus. Genome Res. 2004 14: 149-159
3. Imelfort M, Sequence Comparison Tools, In: Applied Bioinformatics. Ed Edwards D, Hanson D and Stajich J. Springer October 2008.
4. Simpson JT, Wong K, Jackman SD, Schein JE, Jones SJ, Birol I. ABySS: A parallel assembler for short read sequence data. Genome Research, 2009-June
5. Daniel H. Huson, Knut Reinert, Eugene Myers 2001. The greedy path-merging algorithm for sequence assembly . RECOMB '01: Proceedings of the fifth annual international conference on Computational biology (2001), pp. 157-163.
6. D. Hernandez, P. François, L. Farinelli, M. Osteras, and J. Schrenzel. 2008. De novo bacterial genome sequencing: millions of very short reads assembled on a desktop computer. Genome Research. 18:802-809, 2008.
7. Chaisson MJ, Pevzner PA, "Short read fragment assembly of bacterial genomes", Genome Research Jan 2008.
8. D.R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. Genome Research 18:821-829.