## Library

Boost library

## Graph Definition

The graph is defined as below:

typedef boost::adjacency\_list<boost::listS, boost::listS, boost::bidirectionalS, VertexType, EdgeType> BoostGraph;

Meaning we are using the list-implementation of adjacent list to represent the graph, and the graph object support bi-directional operation.

The Vertex is defined as below:

typedef unsigned long long IDType;

class vertexType

{

public:

char\* p\_seq\_; // the sequence associated with current vertex

IDType id\_; // the ID of current vertex, in case of huge graph

SeqLenType len\_; // the length of sequence

bool deleted\_; // set true when current vertex is deleted

};

The Edge is defined as below:

typedef unsigned short int SeqLenType;

class edgeType

{

public:

SeqLenType op\_; // overlap length

bool deleted\_; // set true when current edge is deleted

};

The main data maintained by the graph object is the pointer to BoostGraph. The map from int to BoostVertex is used for random access to node in the graph.

BoostGraph\* p\_graph\_;

std::unordered\_map<int, BoostVertex> vertex\_map\_;

## Graph File format

For read/write the graph from/to hard disk, the graph should be stored in following format: It follows FASTA format.

>source\_id:target\_id1,OP1,target\_id2,OP2,...,target\_idn,OPn

SEQUENCE

>source\_id

SEQUENCE

The SEQUENCE is the sequence stored in current vertex with ID *source\_id*. If current vertex has any neighbor(s), a ‘:’ is used to indicate this case. And the neighbor information is stored as the first neighbor ID

For the header, there is ‘:’ after *source\_id*, followed by all the *target\_ids* that are connected to the source. All the *target\_ids* are separated by comma, except the last target. For the sequence, the OVERLAP is the region where the source overlaps with the targets.

Chart, line chart

Description automatically generated

Figure 1. An example graph of three nodes.

Figure 1 shows an example, and the corresponding content saved on hard disk is:

>0:1,2

ATTTTG[TGTTTT]

>1

[TGTTTT]GACATT

>2

[TGTTTT]CCGCAG

## Read info format

The read information should be saved in a separate file in following format: Reads are sorted by their presence in the sequence.

vertex\_id,read1,length1,pos1,read2,length2,pos2,...

What do you mean by “source\_id”? Is the information relating to an edge or a node?

where,

* vertex\_id is the ID of node.
* *read1*, *read2*, ... are the read header.
* *length1, length2*, ... are the length of each read.
* *pos1, pos2*, ... are the position of each read on the edge, starting from 1.

## API

* constructor of the graph object: AssemblyGraph() that allocate memory to the pointer BoostGraph\* that creates an empty graph.
* constructor of the graph object: AssemblyGraph(const std::string& graph\_file, const std::string& read\_info\_file) that loads the graph from file in the format defined in 3, and loads the read info from file in the format defined in 4. Returns true when succussed.
* destructor of the graph object: ~AssemblyGraph() that free the pointer BoostGraph\* and every dynamic allocated memory.
* bool readGraph(const std::string& graph\_file, const std::string& read\_info\_file = nullptr) which loads the graph from file in the format defined in 3, and loads the read info from file in the format defined in 4. Returns true when succussed.
* bool writeGraph(const std::string& graph\_file, const std::string& read\_info\_file = nullptr) which writes the graph to graph file in the format defined in 3, and writes the read info to file in the format defined in 4. returns true when succussed.
* bool deleteNode(const IDType& node\_id) that checks if node\_id exists in vertex\_map\_, return false if not exists. Then this function deletes one vertex and all edges of that vertex, update vertex\_map\_, free allocated memory for sequence, return true when succussed.
* bool deleteEdge(const IDType& source\_id, const IDType& target\_id) that checks if source\_id and target\_id exist in vertex\_map\_, return false if not exist. Then this function deletes one edge, both its source and target vertices and all edges of its source and target vertices, update vertex\_map\_, free allocated memory for sequence; return true when succussed.
* bool addNode(const VertexType& v) that (what if boost fail to add vertex)
* checks if the graph size will be too large creates a new node, the graph object assigns a new node\_id to this node. Same questions as above.
* bool addEdge(const IDType& source, const IDType& target, const EdgeType& e) that create a new edge from source to target. source and target must be valid node IDs. What is the third parameter? - the overlap length between source and target vertices.
* bool modifyNode(const VertexType& v)
* bool modifyEdge(const IDType& source, const IDType& target, const EdgeType& e)
* void DFSRight(int anchor\_id, int right, order) that traverses the graph using depth-first search from vertex with *anchor\_id*. The maximum traverse length to upstream is *left*, and the maximum traverse length to downstream is *right*. What is the exact behavior? output ordered traversed vertices.
* void DFSLeft(int anchor\_id, int left, order).
* void condense() which condense the overlap graph from ASQG file in the way that all the one-after-one overlapping reads are condensed into one vertex. Why do you want to separate this function from readASQGFIle? Is there any reason you want to keep the uncondensed graph? - This is a large individual function. I leave the option of uncondensed graph for debugging purpose.
* bool readGraphFromAsqg(const std::string& filename) which loads the overlap graph from ASQG file described in 6. After loading the overlap graph, returns true when succussed. How do you define success or fail? Do you perform transitive edge removal? - This program will fail if the provided file name is incorrect. SGA didn’t output transitive edge in ASQG file, I tested using a small example.
* bool readGraphFromFq(const std::string& filename)

## ASQG format

Current graph object also accept the SGA overlap graph, which is stored in the ASQG format. ASQG file consists of three sections, denoted by the token in the front of each line: contents in each section are separated by white space ‘ ’ or tab ‘\t’.

1. **HT**: Header record. This record contains metadata tags for the file version (VN tag) and parameters associated with the graph (for example the minimum overlap length). Ignored by graph object.
2. **VT**: Vertex records. The first field will always be ‘VT’. The second field contains the vertex(read) identifier, and the third field contains the sequence. Subsequent fields contain optional tags, which are ignored by graph object.
3. **ED**: Edge description records. The first field will always be ‘ED’. The following two fields contain the identifiers of two overlapping sequences stored in vertices. The next 8 fields describe the overlap details: the fourth and fifth fields indicate the starting and ending of the overlap region in the first sequence, (0-based and inclusive). The sixth field is the length of first sequence. The seventh and eighth fields indicate the starting and ending of the overlap region in the second sequence. The ninth field is the length of second sequence. The tenth field is either 0 or 1, 1 indicating that the reverse and complimentary of the second sequence overlaps with the first sequence (which is always original sequence). The eleventh field contains the number of mismatches in the overlap region.
4. **Example**:

HT VN:i:1 ER:f:0 OL:i:45 IN:Z:reads.fa CN:i:1 TE:i:0

VT read1 GATCGATCTAGCTAGCTAGCTAGCTAGTTAGATGCATGCATGCTAGCTGG

VT read2 CGATCTAGCTAGCTAGCTAGCTAGTTAGATGCATGCATGCTAGCTGGATA

VT read3 ATCTAGCTAGCTAGCTAGCTAGTTAGATGCATGCATGCTAGCTGGATATT

ED read2 read1 0 46 50 3 49 50 0 0

ED read3 read2 0 47 50 2 49 50 0 0

Current graph object only loads ASQG format and never write in ASQG format.