November 2019

G-PhoCS – Generalized Phylogenetic Coalescent Sampler

Version 1.40

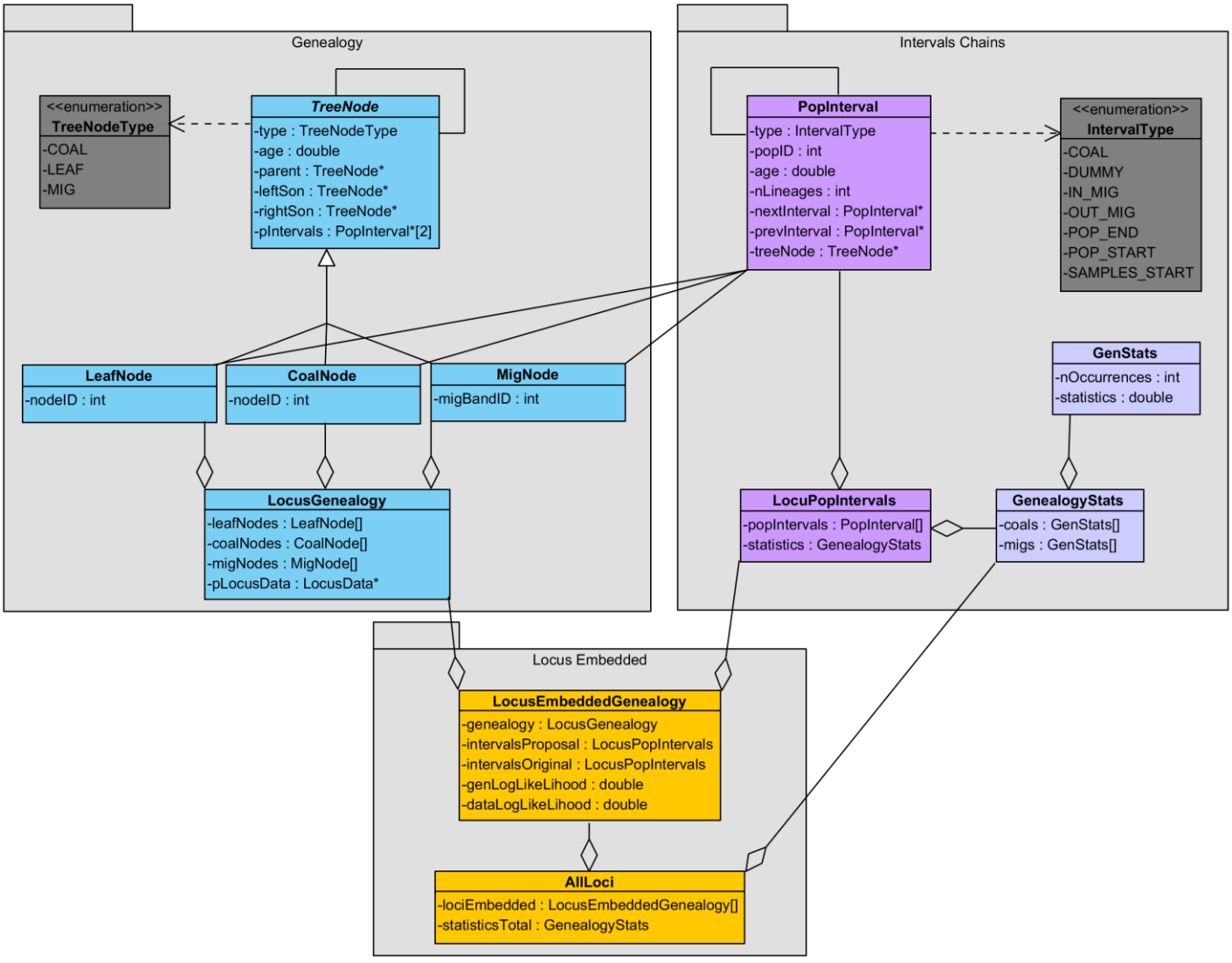
Latest updates

The main updates in version 1.4 include of redesigning fundamental data structures. New design is OOP oriented.

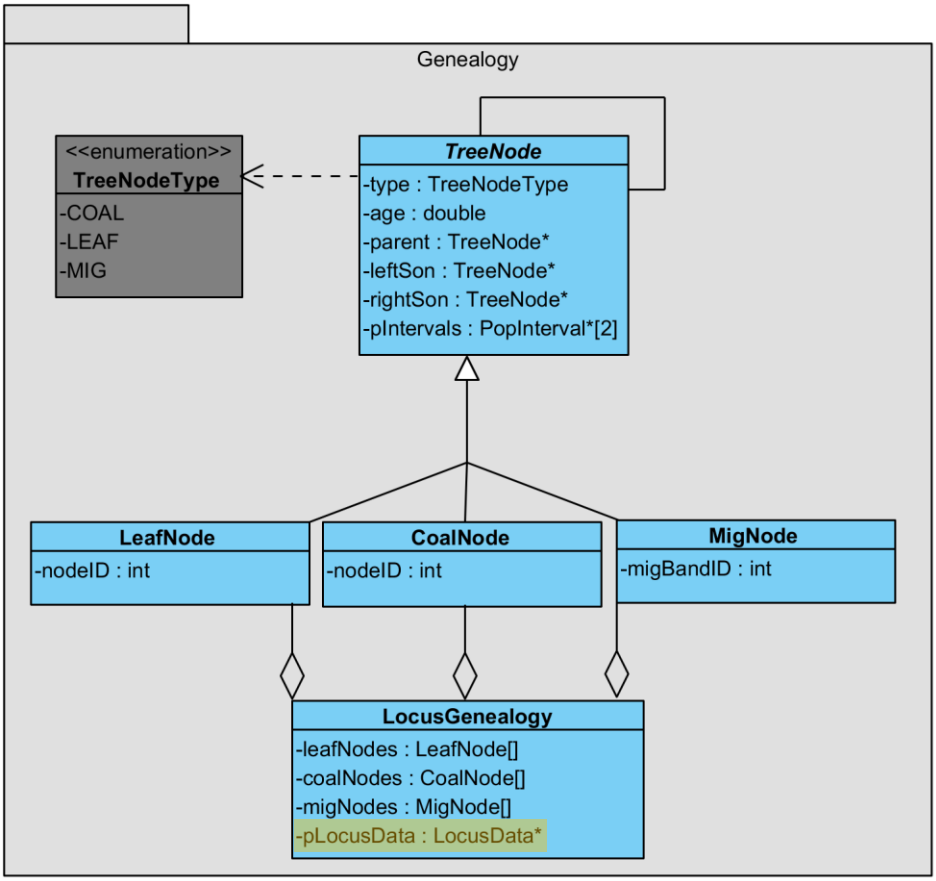
A. New data structures

UML diagram

Below a UML diagram shows the high level picture of the relationships between the new classes. In general, new data structure consists of two main components: genealogy and intervals chains (i.e., chains of events).



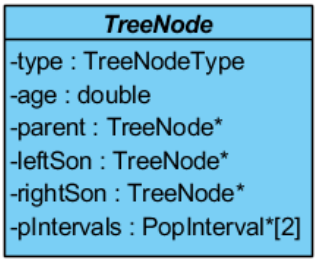
1. Genealogy Classes



1.1 *TreeNode* class

Tree node is an **abstract** class which represents a node in the genealogy tree. There are three types of nodes: **leaf, coalescence and migration**. Each type is represented by a separate (non-abstract) class, which **inherits** *TreeNode*.

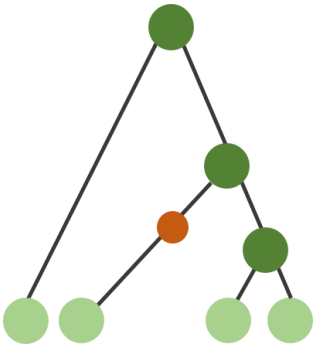
Class members (description – *variable type*):



1. type (*Enum class TreeNodeType*) - type of node.
2. age (*double*) - age of node.
3. parent (*pointer to TreeNode*) - parent of node.
4. rightSon (*pointer to TreeNode*) - right son of node.
5. leftSon (*pointer to TreeNode*) - Left son of node.
6. pIntervals (*list size two of pointer/s to PopInterval class*) - corresponding interval/s (see below).
    - **Leaf** node points to a samples-start interval.
    - **Coalescence** node points to a coalescence interval.
    - **Migration** node points to incoming and out-coming migration intervals.

1.2 *LeafNode, CoalNode* and *MigNode* classes

TreeNode is inherited by each of the three following classes:

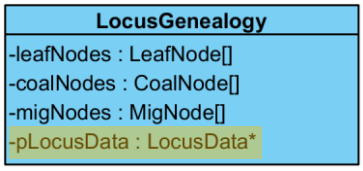


1. ***LeafNode* class -** represents a **leaf node**.
   - Member: `nodeID` (*int*) – ID number of node.
   - Sons point to *null*.
2. ***CoalNode* class -** represents a **coalescence node**.
   - Member: `nodeID` (*int*) – ID number of node.
   - Parent of the root points to *null*.
3. ***MigNode* class -** represents a **migration node**.
   - Member: `migBandID` (*int*) –ID number of migration band.
   - Right and left sons are identical.

1.3 *LocusGenealogy* class

LocusGenealogy class represents a genealogy tree of a single locus. A genealogy tree is simply represented by all its tree nodes.

Class members (description – *variable type*):



1. `leafNodes` (*vector of LeafNode objects*) - list of leaf nodes. Size: number of samples.
2. `coalNodes` (*vector of CoalNode objects*) - list of coalescence nodes. Size: number of samples minus one.
3. `migNodes` (*vector of MigNode objects*) - list of migration nodes. Size: variable.
4. `pLocusData` (*pointer to struct LocusData*) - likelihood data of locus.

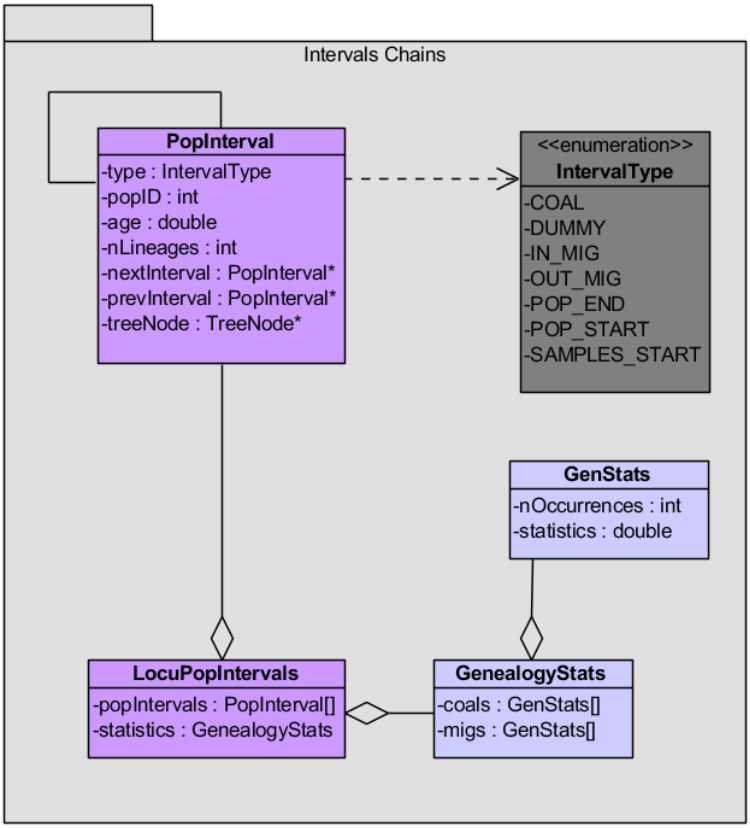
   Note that *LocuData* is not part of a new data structure, but an existing structure.

Class main methods:

- `constructBranches()` – constructs the genealogy tree's brunches by linking the tree nodes to each other (by setting parents/sons of leaves and coalescences. Migration nodes are added separately).
- `testLocusGenealogy()` – verifies that genealogy tree is consistent with the original genealogy of previous version.
- Functions with **wrap** suffix (e.g. `getNodeAgeWrap`) – those functions wrap old, global functions which get as an argument a pointer to locus data.

November 2019

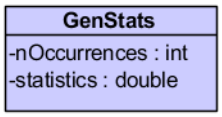
2. Intervals Classes



2.1 *GenStats* struct

GeneStats struct stores statistics of a population.

Class members:

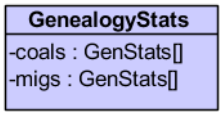


1. `nOccurrences` (int) - number of occurrences of coalescence or migration events.
2. `statistics` (*double*) - value of statistics.

2.2 *GenealogyStats* struct

GenealogyStats struct stores the statistics of a genealogy. For each genealogy, statistics of coalescence and migration events are computed.

Class members:



1. `coals` (*vector of GenStats structs*) - list of statistics of coalescence events.
   Size: number of populations.

2. `migs` (*vector of GenStats structs*) - list of statistics of migration events. Size: number of migration bands.

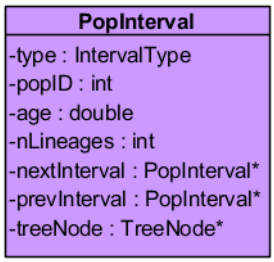
2.3 *PopInterval* class

PopInterval class represents a single interval in a chain of intervals. In a genealogy tree, the life time of a population is divided into time-intervals, where each interval ends in one of the events below. A chain of intervals is a linked list of those intervals, chronologically arranged.

Intervals types:

- **DUMMY:** A free interval.
- **POP_START:** Start of a population.
- **POP_END:** End of a population.
- **SAMPLES_START:** Start of genetic sequences samples.
- **COAL:** Coalescent event.
- **IN_MIG:** Incoming migration event.
- **OUT_MIG:** Out-coming migration event.

Class members (description – *variable type*):

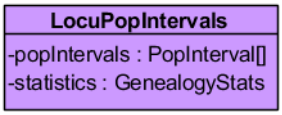


1. `type` (*Enum class IntervalType*) - type of interval.
2. `popID` (*int*) - ID number of interval's population.
3. `age` (*double*) - age of interval (i.e., when interval ends).
4. `nLineages` (*int*) - number of lineages.
5. `nextInterval` (*pointer to PopInterval*) - next interval.
6. `prevInterval` (*pointer to PopInterval*) - previous interval.
7. `treeNode` (*pointer to TreeNode class*) - corresponding tree node:
    - **Coalescence** interval points to a coalescence tree-node.
    - **In/out-coming migration** interval points to a migration tree-node.
    - **Samples-start** interval points to null (samples-start interval is associated with several leaf-nodes, thus can't point to all of them).
    - Intervals of rest types point to null.

2.4 *LocusPopIntervals* class

LocusPopIntervals class contains the intervals chains of a locus. Each population is associated with its own intervals chain.

Class members (description – *variable type*):

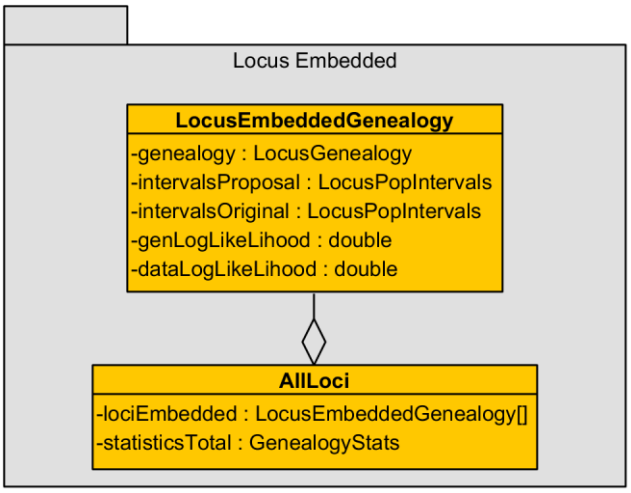


1. List of intervals – *array of PopInterval objects. Cells point to each other, as in a linked-list (see figure 2).* Size: fixed size.
2. Statistics of locus - *GenealogyStats struct.*

Class main methods:

- `createStartEndIntervals()` – initializes intervals array with POP_START and POP_END intervals. Let N denote the number of populations. The first N cells in the intervals array are of type POP_START, and the next N cells are of type POP_END. Total: 2N intervals (see figure 2).
- `getSamplesStart()` – returns an interval of type SAMPLES_START of a given population.
- `getPopStart()` – returns an interval of type POP_START of a given population.
- `getPopEnd()` – returns an interval of type POP_END of a given population.
- `getIntervalFromPool()` – returns an unused interval.
- `createInterval()` – adds a new interval in given population and time.
- `computeGeneTreeStats()` – computes statistics of a locus.
- `recalcStatsStats()` – recalculates statistics of a given population in a locus.
- `computeStatsDelta()` – computes the delta in statistics of a given population in a locus.
- `computeLogLikelihood()` – calculates log-likelihood of a locus.
- `copyIntervals()` – all class members are copied by a DEEP copy, except of the pointers to the tree nodes, which are copied by a SHALLOW copy.
- `testPopIntervals()` – verifies that chain intervals are consistent with the original chains in previous version.
- `testGenealogyStatistics()` – verifies that locus statistics are consistent with the original statistics in previous version.

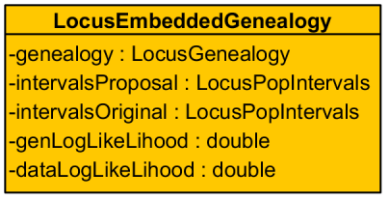
3. Locus Embedded Genealogy Classes



3.1 *LocusEmbeddedGenealogy* class

LocusEmbeddedGenealogy class combines a genealogy tree and the chain intervals associated with it. See figure below.

Class members:



1. `genealogy` (*LocusGenealogy object*) - genealogy.
2. `intervalsProposal` (*LocusPopIntervals object*) **- proposal** intervals, see proposals mechanism.
3. `intervalsProposal` (*LocusPopIntervals object*) - **original** intervals, see proposals mechanism.
4. `genLogLikeLihood` (*double*) - log-likelihood of genealogy, P(genealogy | model).
5. `dataLogLikeLihood` (*double*) - log-likelihood of data, P(data | genealogy).

Proposals mechanism:

Proposals are suggestions for a change in the genealogy (e.g., preceding a coalescence event). The proposals mechanism is implemented as following: *LocusEmbeddedGenealogy* class holds **two identical copies of intervals objects** - proposal-intervals (**A**) and original-intervals (**B**). When a proposal is suggested, it is first made on copy **A**. If the proposal is accepted, then **A** is copied to **B** (i.e., proposal-intervals overrides the original-intervals). Otherwise, if proposal is rejected, **B** is copied to **A**.

Note: Only intervals are copied. Genealogy has a single copy and changes are made directly on it.

Important: Evaluation of running times showed that copying is time consuming. This mechanism should be replaced in more efficient mechanism.

Class main methods:

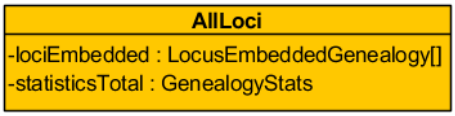
- `constructEmbeddedGenealogy()` – constructs both genealogy and corresponding intervals (figure 3). Creation of intervals is done by three loops: First, samples-start intervals are created for each non-ancient population. Second, coalescence intervals are created. Finally, migration intervals are added.
- `considerIntervalMove()` – computes the modifications required for changing a specific genealogy tree by moving an interval to a new position.
- `updateGB_InternalNode()` – perturbs times of all coalescence nodes. See more details later on.

3.2 *AllLoci* class

AllLoci class stores all loci, embedded with their genealogies. This class should have a single instance.

Class members:

1. `lociEmbedded` (*vector of LocusEmbeddedGenealog objects*) - list of loci. Size: number of loci.
2. `statisticsTotal` (*GenealogyStats struct*) - total statistics of populations.



Class main methods:

- *calcLociTotalStats()* – calculates statistics of all loci.

Figure 1: Illustration of locus embedded genealogy

Illustration shows an example of a genealogy and its corresponding intervals.

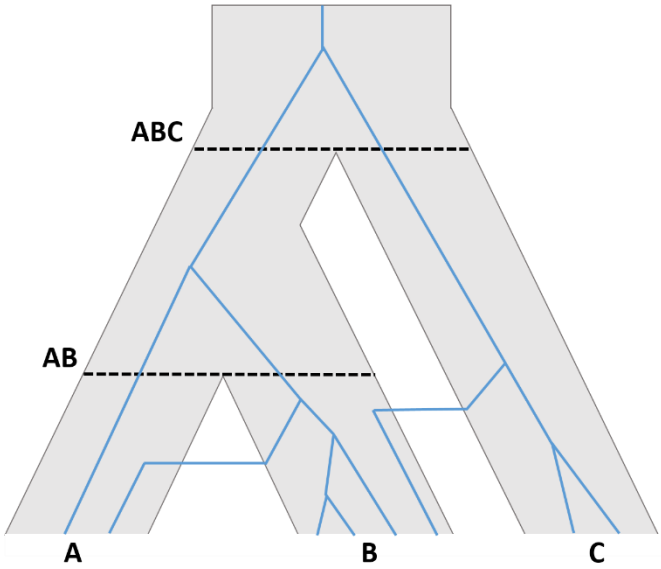


Figure 1a: Example genealogy embedded in the population phylogeny. There are 5 populations (A, B, C, AB, ABC), 8 samples, 7 coalescence events and 2 migration evens.

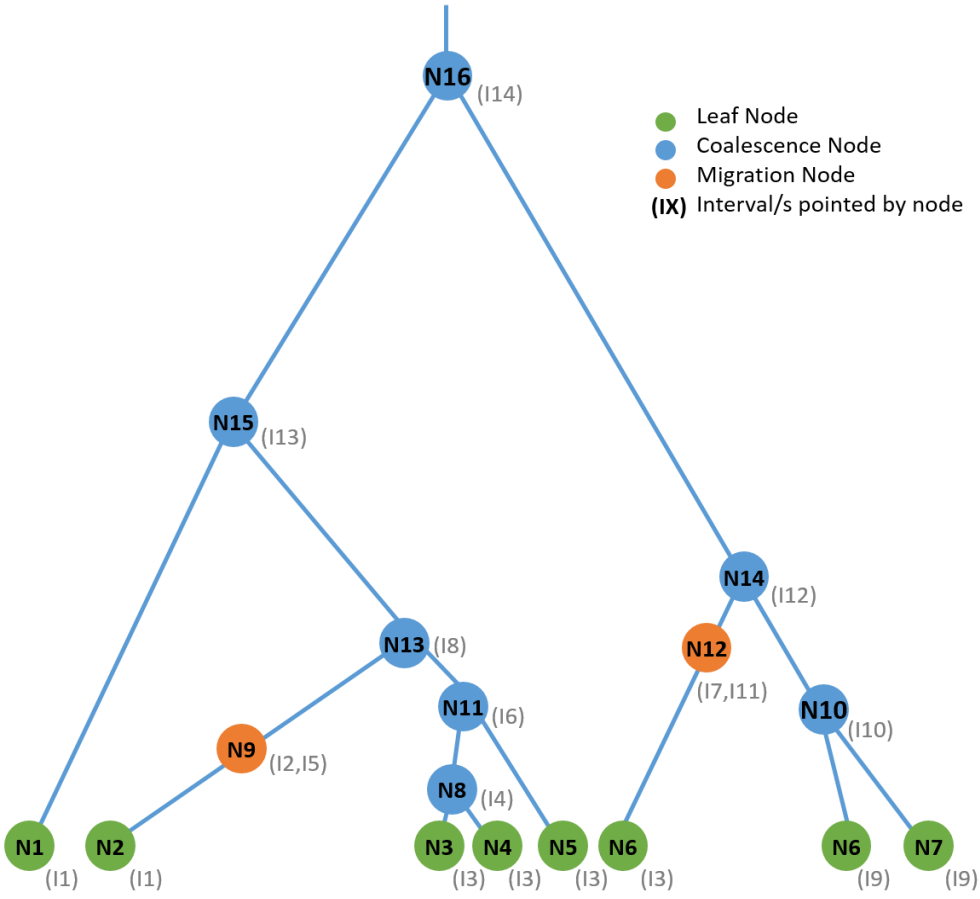


Figure 1b: Genealogy outside the context of the population phylogeny.

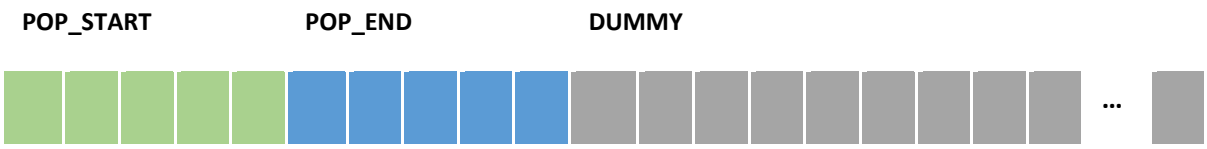


Figure 1c: In initialization of the intervals list, first five cells are defined as pop-start, and the next five cells are defined as pop-end. Those ten cells are fixed during running-time. The rest of the cells are free.

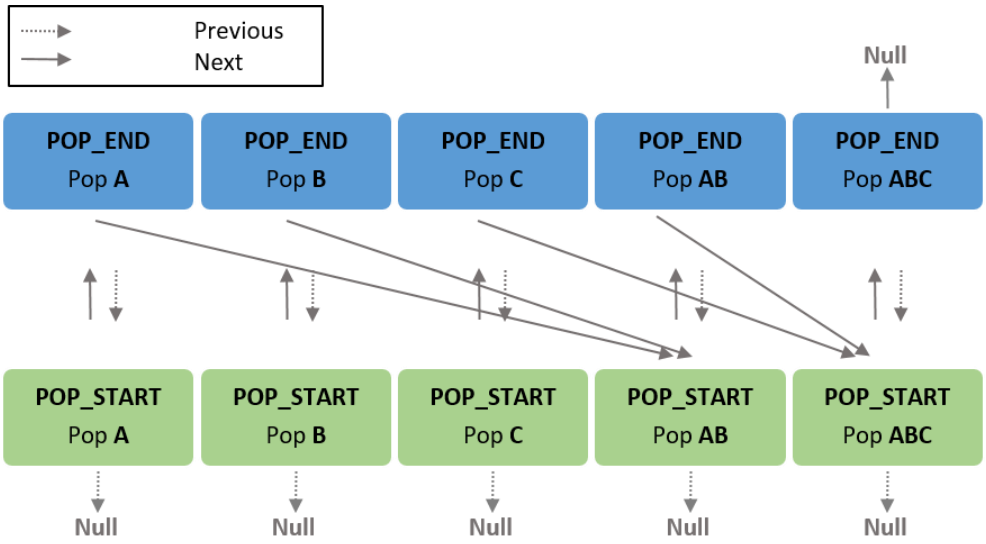


Figure 1d: Intervals' next and previous pointers point as shown below. It can be seen that a POP_END interval of a population points to the POP_START of the parent population (for example, pop B points to its parent pop AB).

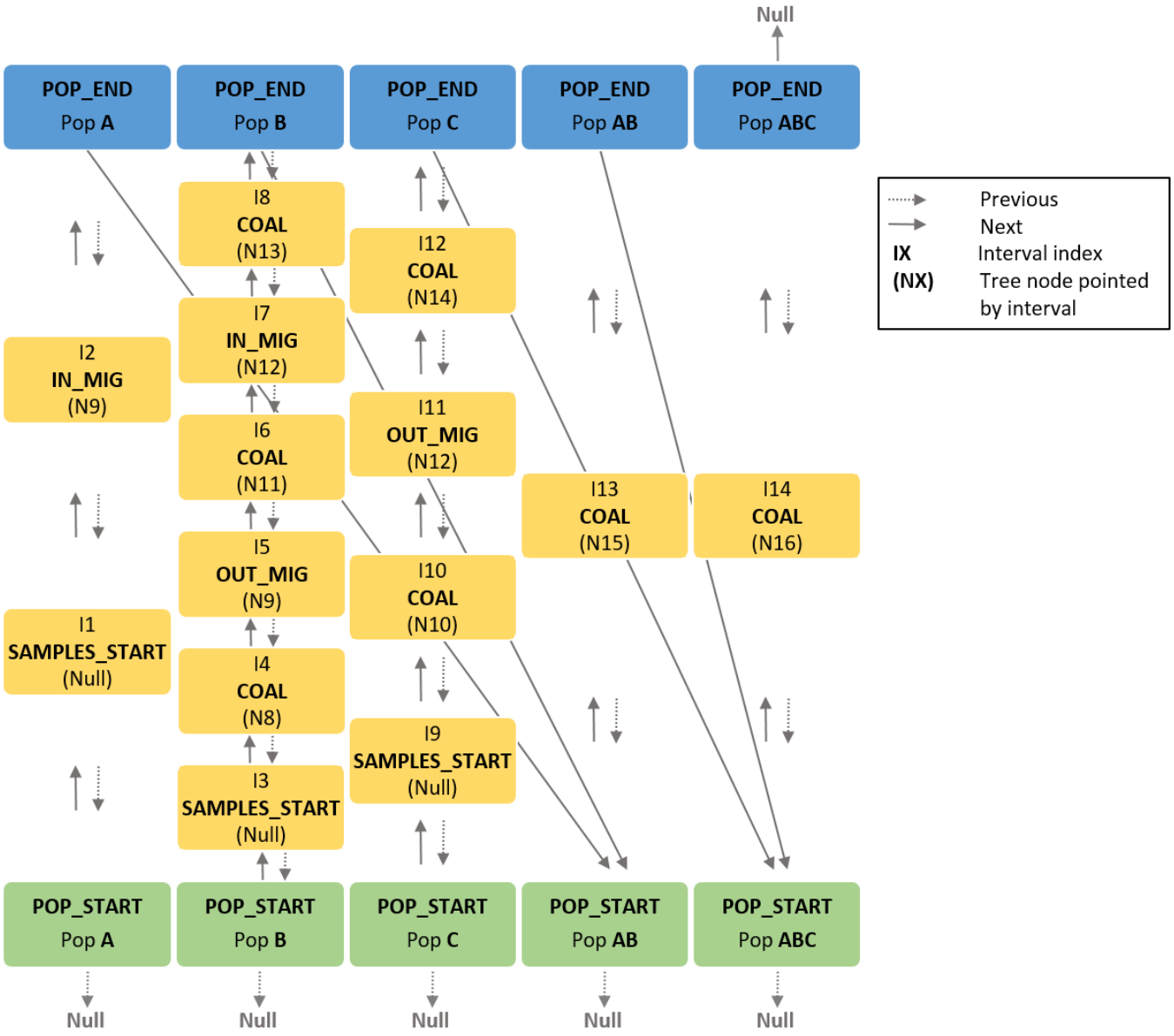
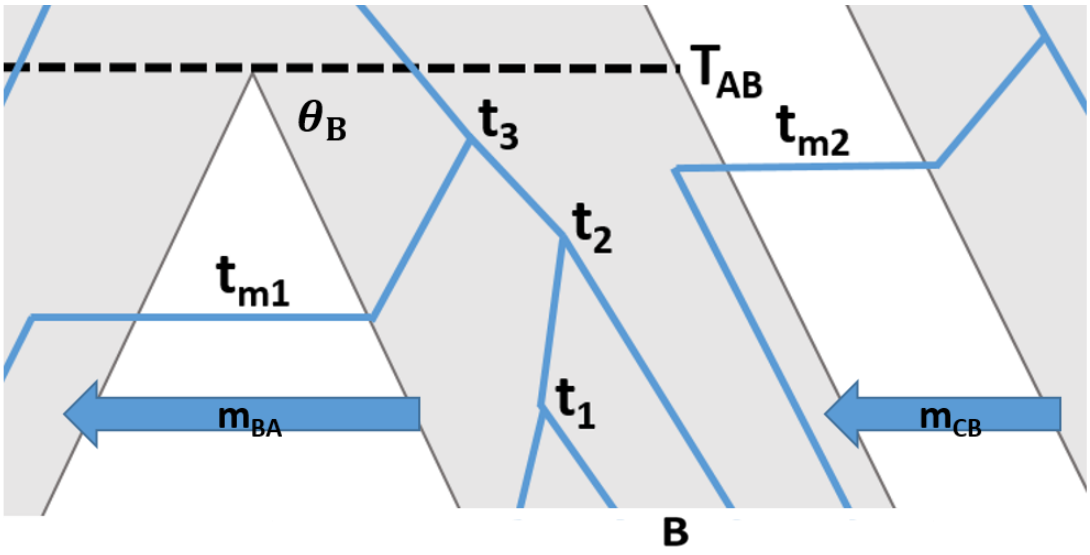


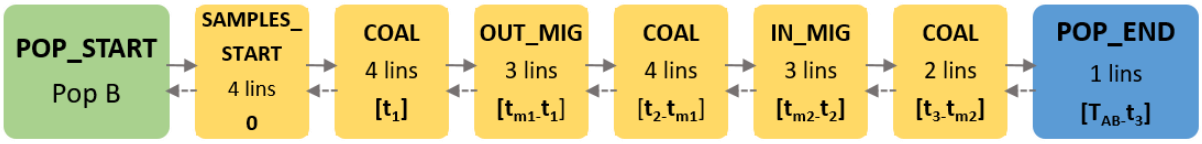
Figure 1e: Intervals are created while iterating genealogy nodes (function *constructEmbeddedGenealogy*). Pointers from tree nodes to intervals and vice versa are shown in parentheses, and behave as follow:

1. **Coalescence:** A coalescence tree-node points to intervals of type coalescence, and vice versa. E.g.: Node N15 points to interval I13, and interval I13 points to node N15.
2. **Migration:** A migration tree-node points to two intervals – interval of type in-migration and interval of type out-migration. Each of those intervals points back to this tree node. E.g.: Node N9 points to intervals I2 and I5, while those intervals point to node N9 each.
3. **Leaf:** Leaf nodes of same population point to an interval of type samples-start, while this interval points to null. E.g.: Nodes N3, N4, N5, N6, point to interval I3. I3 points to null.

B. Example of calculating statistics

This section demonstrates a computation of a population's statistics. Here we examine population B in the genealogy shown in previous sections.





Remainder: genealogy prior

An interval I contributes the following amounts to the genealogy prior:

$$p(I|\theta_p) = \left(\frac{2}{\theta_p}\right)^{coal(I)} \cdot \exp\{-\frac{(n^2-n)\tau}{\theta_p}\},$$

$$p(I|m_b) = m_b^{mig(I,b)} \cdot \exp\{-m_b\, n\, \tau\},$$

where -

τ is the interval length

n is the lineages number

$coal(I) \in \{0,1\}$ indicates whether I ends in a **coalescence** event or not

$mig(I,b) \in \{0,1\}$ indicates whether I ends in an (incoming) **migration** event in band b

θ_p, m_b are the model parameters

In blue: coalescence statistics

In orange: migration statistics

Note that the statistics themselves do not depend on the model parameters (theta, m). The statistics are simply the number of evens and the normalized sum of lineage counts. In our example, the statistics of population B are the following:

- **Coalescence statistics:**
  - $numCoals(B) = 3.$
  - $coalStats(B) = 12t_1 + 6(t_{m1} - t_1) + 12(t_2 - t_{m1}) + 6(t_{m2} - t_2) + 2(t_3 - t_{m2}) + 0(\tau_{AB} - t_3).$
- **Migration statistics:**
  - $numMigs(\mathrm{C} \to \mathrm{B}) = 1.$
  - $migStats(\mathrm{C} \to \mathrm{B}) = 4\mathrm{t}_1 + 3(t_{m1} - t_1) + 4(t_2 - t_{m1}) + 3(t_{m2} - t_2) + 2(t_3 - t_{m2}) + 1(\tau_{AB} - t_3).$

The likelihood is then computed as a function of the statistics and the model parameters. In our example, the total contributions of population B to the genealogy prior are:

$\left(\frac{2}{\theta_B}\right)^{numCoals(B)} \cdot \exp\{-\frac{coalStats(B)}{\theta_B}\}$, and -

$m_{C\to B}^{numMigs(\mathrm{C}\to\mathrm{B})} \cdot \exp\{-\mathrm{m}_{\mathrm{B}\to\mathrm{C}} migStats(\mathrm{C} \to \mathrm{B})\}$,

Note: For the calculations of migration statistics of a population *p*, only migration bands whose target population is *p* are considered (in this example, only band C→B is considered, and not band B→A).

Note2: in this example, **all** intervals contribute to the migration statistics, since the lifetime of population B is in the life span of migration band C→B. In general, interval contributes only when its lifetime is in the life span of the relevant migration band.

C. Old data structures

This sections provides a high level explanation of the old data structures (DSs) and how they relate to the new data structures. Old DS refers to DS appear in version 1.4 (and not 1.3V), and are not one of the classes mentioned above.

	Old data structure	New data structure
1	***LikelihoodNode* (*LIKELIHOOD_NODE*) struct** (*LocusDataLikelihood.cpp*)	***TreeNode* class** and inherited classes: ○ ***LeafNode* class** ○ ***CoalNode* class** ○ ***MigNode* class** *(TreeNode.h)*
	***LocusData* (*LOCUS_LIKELIHOOD*) struct:** ***LikelihoodNode* **nodeArray** member (*LocusDataLikelihood.h*)	**LocusGenealogy class** (*LocusGenealogy.h*)
2	***Event* class** *(DataLayer.h)*	***PopInterval* class** (*PopInterval.h*)
	***EventChain* class** *(DataLayer.h)*	***LocusPopIntervals* class** (*LocusPopIntervals.h*)
	***EventChains* class** *(DataLayer.h)*	---
3	---	***LocusEmbeddedGenealogy* class** *(LocusEmbeddedGenealogy.h)*
4	**DATA_STATE struct:** ***LocusData* **lociData** member *(GPhoCS.h)*	***AllLoci* class** *(AllLoci.h)*

Notice: The comparison shown in table is at the conceptual level. In details, the differences are great.

1. **Genealogy -** in old DSs, genealogy is saved in an array named **nodeArray**, which is a member of the struct *LocusData*. Each element of the array is of type *LikelihoodNode* struct, which is equivalent to the *treeNode* classes.

Notice The new *LocusGenealogy* class doesn't completely replace *LocusData*. It points to it. Maybe later on, the functionality of *LocusData* will be merged with *LocusGenealogy*. Use example: locusData->nodeArray[nodeID]->father

2. **Events/intervals chains -** in old DSs, events were saved in a global DS with three hierarchies:
    a. **First entry level:** genealogy [class *EventChains*].
    b. **Second entry level:** event id [class *EventChain*].
       Class contains pointers to first and last events for every population.
    c. **Third entry level:** event [class *Event*].

    Use example: *event_chains[genealogy_id].events[event_id].getType();*

    Notice: in new data structures, **events** terminology is replaced by **intervals**.

3. In old data structure there is no DS which integrates genealogy and events/intervals into one DS.
4. In old DS, loci are saved in an array named **lociData**, which is a member of the struct *LocusData*.

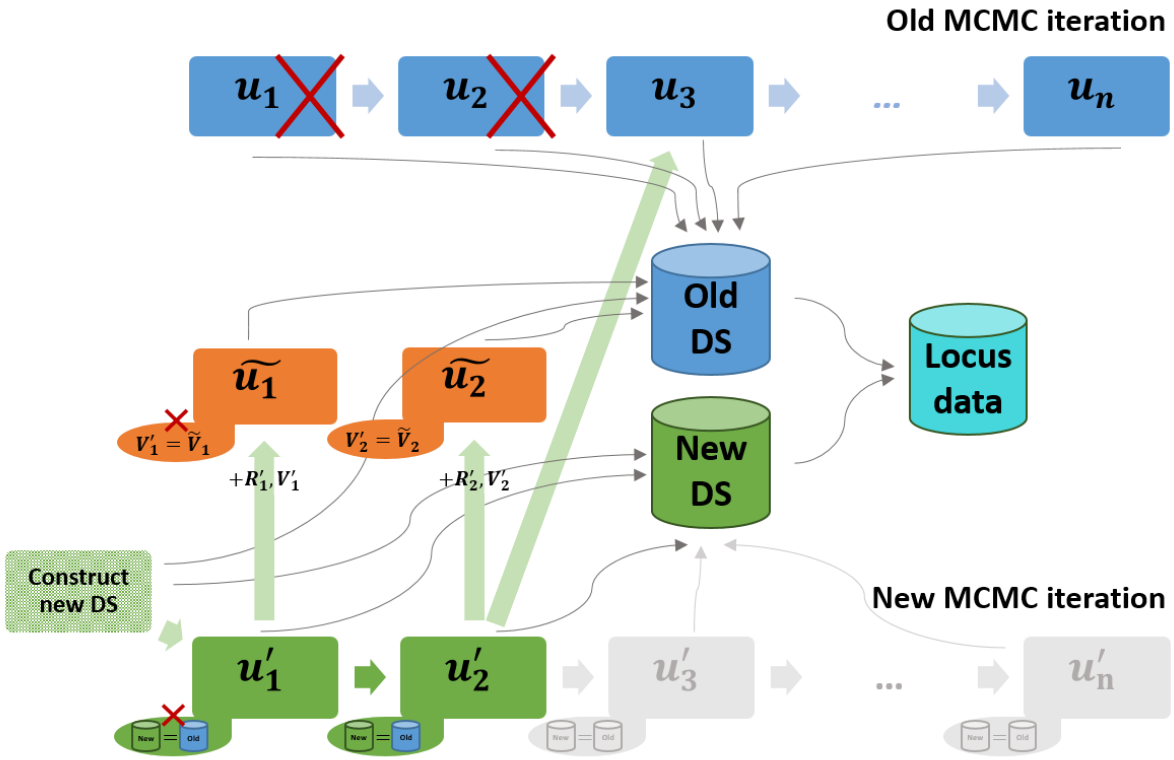
D. Integration with existing code

This sections shows how new data-structures (DSs) are integrated into the existing code.

1. An overview

- Let u_i denote the i'th proposal in a set of *n* proposals.
- An MCMC iteration is a chain of proposals $u_1 \to u_2 \to \cdots \to u_n$.
- Let V_i denote set of values computed by proposal u_i.
- Let R_i denote set of random values drawn by proposal u_i.

Following is a scheme demonstrating the gradual transition from the old DSs to the new DSs.



u_i	- an old proposal
u'_i	- a new proposal
$\tilde{u}_i$	- an intermediate version of the proposal
⇨	- indicates passing to the next proposal
⤳	- indicates a use (read/write) of DS
⬭	- validation

In figure, chain $u_1 \to \cdots \to u_n$ represents the original, old proposals, which use old DSs only. Chain $u'_1 \to \cdots \to u'_n$ represents the new implemented proposals, which use new DSs only.

Challenge description: One would think that the transition from the old DS to the new DS can be done as following: The chains u and u' run in parallel. At the end of the run, the two DSs are compared. If new DS is identical to the old DS, then transition is successful, and the old chain (u) can be removed. However, this can't be done. The reason is that the proposals draw random numbers (R). To be able to compare the DSs, the old and the new proposals must use the same values.

Solution: The transition between the two DSs is done as following (see figure):

1. Suppose we are in the beginning of an MCMC iteration. Before starting the new proposals chain u', the new DS is constructed, based on current state of the old DS.
2. Proposal u'_1 is called. During the proposal, it draws some random values R'_1, computes some values V'_1, and updates the new DS.
3. Proposal u'_1 calls an "intermediate" proposal, denoted by $\tilde{u}_1$, and passes it R'_1 and V'_1 values as arguments. Proposal $\tilde{u}_1$ is a copy of the old proposal u_1, except that:
    a. Proposal $\tilde{u}_1$ uses R'_1 values instead of drawing these values itself.
    b. Proposal $\tilde{u}_1$ makes validation to u'_1: It computes the values $\tilde{V}_1$, and verifies that values V'_1 equal to $\tilde{V}_1$.
4. The call to u_1 is muted. (Note that $\tilde{u}_1$ actually replaces u_1, which means it updates old DS only).
5. Proposal u'_1 validates that the new DS equals the old DS, for all benchmarks.
6. If both validations are o.k., the validation code in $\tilde{u}_1$ and in u'_1 is muted.
7. Proposal u'_2 is called. Repeat 2-4, with $i = 2$.
8. Proposals $u_3, \ldots, u_n$ are called.
9. A new MCMC iteration is starting. Note that as long as not all new proposals are implemented, the new DS must be constructed from scratch at each iteration beginning, to catch up with the old DS (which keeps getting updated by the old proposals).

To sum up, the order of the actions described above is as follow:

$$construct\ new\ DS \to u'_1 \to \tilde{u}_1 \to u'_2 \to \tilde{u}_2 \to u_3 \to \cdots \to u_n.$$

November 2019

Note: most of the proposals access *LocusData*. A new proposal u_i' access *LocusData* through the new DS, and an intermediate proposal $\tilde{u}_i$ access *LocusData* through the old DS.

Work plan:

To implement the rest of the new proposals, do the following steps:

1. Implement a new proposal u_i' . Proposal u_i' draws some random values R_i', and computes some other non-random values. Among the later, think what values, denoted by V_i', should be validated for correctness. Execute proposal $\tilde{u}_i$ (see next step), and pass V_i' and R_i' as arguments. Add validation which makes sure that the new DS equals the old DS (after both DSs were updated by u_i' and $\tilde{u}_i$). (For this validation use the tests methods of the new classes).
   Note: proposal u_i' updates new DS only.
2. Implement an intermediate proposal $\tilde{u}_i$, based on the old code of u_i. Replace lines which draw random numbers by the input values R_i'. Computes the corresponding values $\tilde{V}_i$.
   Add validation that V_i' equal to $\tilde{V}_i$.
   Note: proposal $\tilde{u}_i$ updates old DS only.
3. **Mute** the call to the old proposal u_i, which is now replaced by $\tilde{u}_i$.
4. Run G-PhoCS on all benchmarks, and verify that output traces are identical to the true traces.
5. If everything is o.k., **mute** the validation parts in u_i' and in $\tilde{u}_i$, and repeat 1-4 for $i + 1$.
6. After implementing last proposal, **mute** all intermediate proposals $\tilde{u}_i$. In addition, make sure the new DS is not constructed in the beginning of each MCMC iteration.

2. Code example

To illustrate previous section in more detail, we will see a pseudo code of proposals u_1' and $\tilde{u}_1$.

Proposal u_1' is ***updateGB_InternalNode*** (*updateIN,* for short). This proposal perturbs times for all coalescent nodes, and is a method of class *LocusEmbeddedGenealogy*.

Proposal $\tilde{u}_1$ is called ***updateGB_InternalNode*_oldDS.**

First, following is the context of the u_1' in the main MCMC loop in file *GPhoCS.cpp*. It can be seen that before calling proposal u_1', the new DS is constructed from scratch.

```
function performMCMC():
    ...
    AllLoci allLoci   //create an object of AllLoci

    for each MCMC iteration:        //main MCMC loop

        for each locus in allLoci:

            //construct migration bands intervals
            constructMigBandsGenealogy()

            //construct genealogy and intervals
            locus.constructEmbeddedGenealogy()

            //compute genealogy statistics
            locus.computeGenetreeStats()

            //update genealogy statistics
            locus.updateGenLogLikelihood()

            //copy intervals from proposal to original
            locus.copyIntervlas()

            //update internal nodes
            locus.updateGB_InternalNode()

            //end of inner loop
    ...
    //end of MCMC loop
...
//end of function
```

Construct new DS

u_1'

u'_1

```
function updateGB_InternalNode (finetune) {
    ...
    for each node in coalescence_nodes:
        //get lower and upper times bound for the new age
        lowerBound = ...
        upperBound = ...

        //get age of current node (of new DS)
        t = getAge()

        //calculate new age
        tnew = t + finetune * get_random_number()

        //consider interval move
        lnAcceptance = considerIntervalMove(tnew)

        //boolean if proposal is accepted
        isAccepted = lnAcceptance >= 0 or get_random_number() < exp(lnAcceptance)

        if isAccepted: //if proposal is accepted
            ...
            //copy original intervals into proposal intervals
            intervalsOri.copyIntervals(intervalsPro)
            ...
            //original function, works on LocusData
            resetSaved()
        else: // reject changes and revert to saved version
            ...
            //copy proposal intervals into original intervals
            intervalsPro.copyIntervals(intervalsOri)
            //original function, works on LocusData
            revertToSaved()

        //revert changes made above in old DS
        ...
        resetSaved()
        //update old DS
        updateGB_InternalNode_oldDS(lowerBound, upperBound, tnew,
                                    lnAcceptance, isAccepted, node)

        //validates new DS equals old DS
        testLocusEmbeddedGenealogy()

    //end of loop
    ...
//end of function
```

drawing

drawing

$\widetilde{u_1}$

R'_1 = { tnew, isAccepted }
V'_1 = { lowerBound, upperBound, lnAcceptance}

validation

In the above code of proposal u_1', we can see the computation of values V_1' and R_1'.
Then comes the condition testing if proposal is accepted. Inside the condition, structure *LocusData* is updated. In the end of the proposal, we call proposal $\tilde{u}_1$, with V_1' and R_1' as arguments. However, before calling $\tilde{u}_1$, the changes made in *LocusData* should be undone, since proposal $\tilde{u}_1$ is going to make the exact changes in *LocusData*. After returning from $\tilde{u}_1$, we validate that the new DS equals the old DS (i.e., that DSs are consistent after applying the proposals).

```
function updateGB_InternalNode_oldDS(lowerBound, upperBound, tnew,
                                     lnAcceptance, isAccepted, node):

    //get lower and upper times bound for new age
    lowerBound2 = …
    upperBound2 = …

    //assert lower and upper bounds calculated in updateGB_InternalNode
    //equal to values resulted by original code
    assert(lowerBound == lowerBound2)
    assert(upperBound == upperBound2)

    //get age of current node (of old DS)
    t = getAge()
    …
    //consider event move
    lnAcceptance2 = considerEventMove(tnew)

    assert(lnAcceptance == lnAcceptance2)

    if isAccepted: //if proposal is accepted
            ….
            //original function, works on LocusData
            resetSaved()
    else: // reject changes and revert to saved version
            …
            //original function, works on LocusData
            revertToSaved()

//end of function
```

validation $V_1' = \tilde{V}_1$

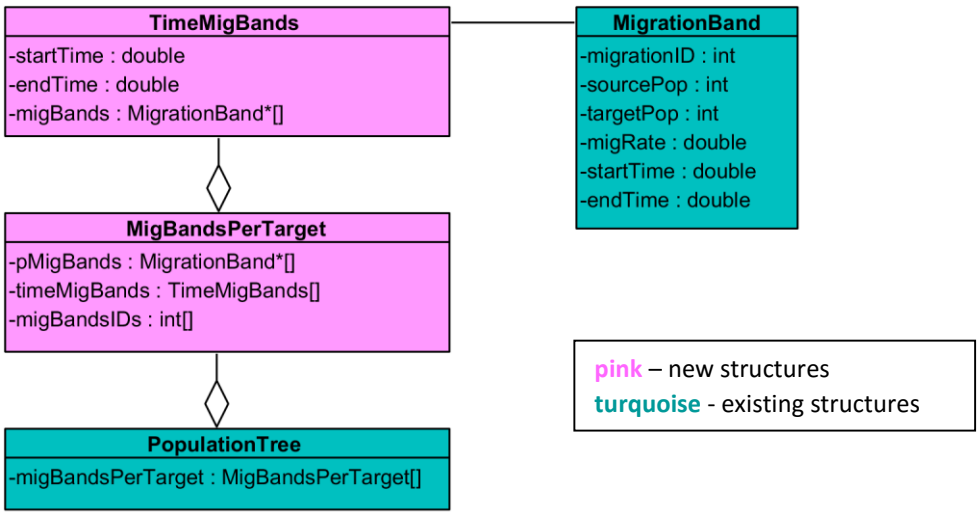
validation $V_1' = \tilde{V}_1$

In proposal $\tilde{u}_1$, we can see that there is no drawing of random values. Instead, $\tilde{u}_1$ uses the input values R_1' (*tnew* and *isAccepted*). Proposal $\tilde{u}_1$ computes $\tilde{V}_1$ by itself (*lowerBound*, *upperBound* and *lnAcceptance*), then validates that the input values V_1' equal to $\tilde{V}_1$.

Note thathere *LocusData* is updated through the old DS.

Also note that proposal $\tilde{u}_1$ runs per node (given as an argument), in contrast to u_1' which iterates all coalescence nodes.

E. Data structure for maintaining the time spans of migration bands



The time span of migration bands is a global attribute (does not change from one locus to another), and is a function of the set of migration bands and the divergence time parameters. The main purpose of these data structures is to enable queries of the sort: **"at a certain point in time – which migration bands are incoming into population *p*?"**. This is achieved by following two new *structures* (not classes).

Note that structures are an extension of the existing structure ***PopulationTree struct***, declared in file `PopulationTree.h`..

1. *TimeMigBands* struct

Struct holds a subset of migration bands incoming to a given pop living in a defined time period (defined by a starting and ending points).

Struct members:

- `startTime` (*double*) – start of time interval.
- `endTime` (*double*) – end of time interval.
- `migBands` (*vector of pointers to **MigrationBand structs***) – list of migration bands incoming to a given population in a given time period.

2. *MigBandsPerTarget* struct

Struct holds a list of ***TimeMigBands structs*** for all time intervals in a certain population with distinct sets of incoming migration bands.

Struct members:

- `pMigBands` (*vector of pointers to **MigrationBand structs***) – list of all migration bands incoming to the specific pop.
- `timeMigBands` (*vector of pointers to **TimeMigBands* struct**) – list of intervals with distinct subsets of bands incoming to the specific pop.
- `migBandIDs` (*vector of ints*) – list of migration band IDs according to their order in `pMigBands`.

The new data structure is accessed from the global ***PopulationTree struct*** via a new member:

- `migBandsPerTarget` (*vector of **MigBandsPerTarget* struct**) – list of *MigBandsPerTarget* partitioned according to target populations.

Global functions accessing the migration-band structures

Functions are implemented in file `PopulationTree.cpp`. They are global in the sense that they are not part of the migraiotn-band structures., but rather get as argument a pointer to ***PopulationTree struct***).

- `getLiveMigBands()` – This function implements the main objective of this DS – to determine the set of incoming migration bands for a given target population at a given time.
  Input: a target pop, a time point *t* (age)
  Output: a *TimeMigBands* structure containing *t* (`startTime` <= *t* < `endTime`). If not exists, returns null.

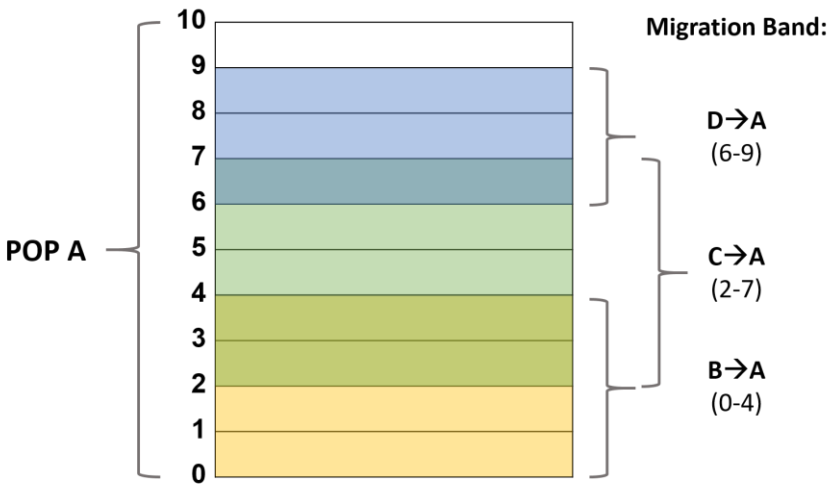
  This function is called by functions calculating the statistics of a locus genealogy (`recalcStats`, `computeStatsDelta`). In each time point *t*, we need to know which migration bands are alive in order to compute their statistics (recall that only active/live migration bands contribute to the migration statistics).

- `constructMigBandsTimes()` – function constructs the timed migration-bands structures based on the current divergence times. It is called as part of the new DS construction every time that a divergence time parameter is modified (see previous section, under "code example").

November 2019

Following is an illustration of the data-structures, through a demonstration of their construction process.

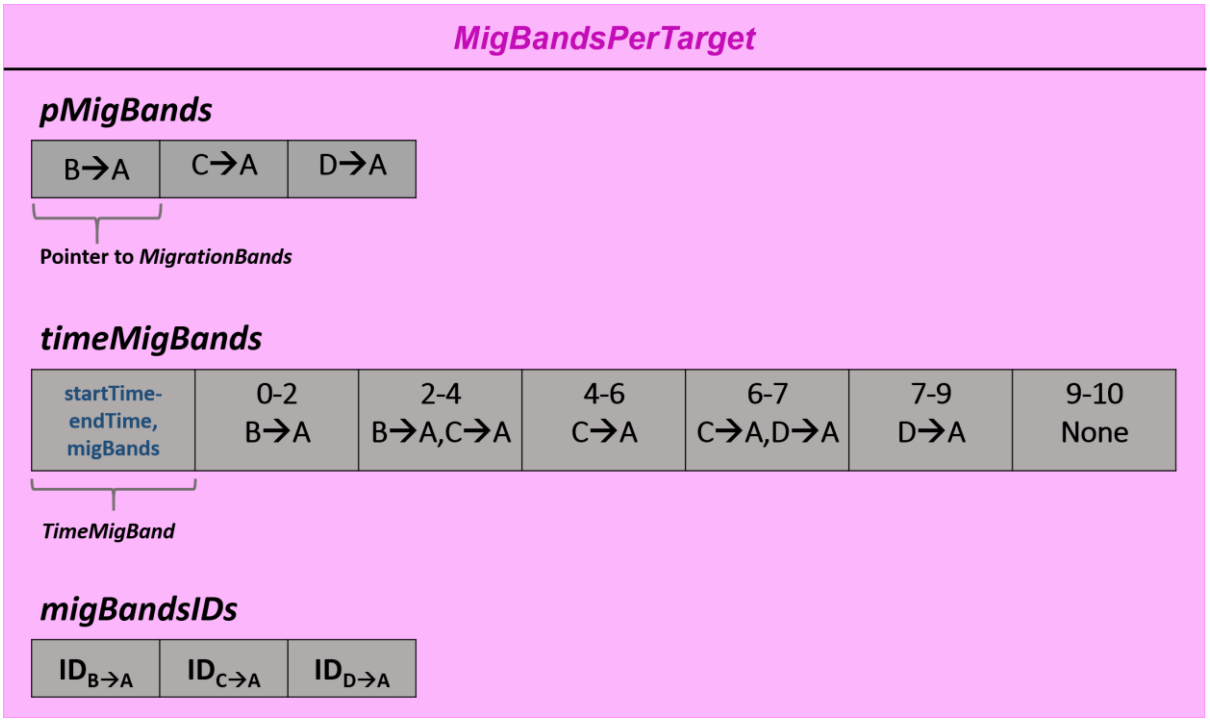
We will demonstrate for target-population **A**.
The lifetime of A: **0-10** (units time).
The migration bands which A is their target-pop, and their lifetimes (in parenthesis), are shown in the following figure:



Note that the life span of each migration band is the intersection of the life span of the target population (A) and the life span of the source population (B,C,D in this example). We start by determining the switch points between time intervals where the set of migration bands changes. This is done by taking all start and end times of the migration bands, and the start and end time of population A and sorting them, as follows:

0	2	4	6	7	9	10

Now, **each two successive time points** forms a struct of type *TimeMigBand*, together with the migration-bands which are active between these two time points. These *TimeMigBand* structures are stored in order in the list *timeMigBands* member of the struct *MigBandsPerTarget*.



Now, if we call function `getLiveMigBands()` with *pop=A* and *t=3*, the function will return a pointer to the following *TimeMigBand* structure, indicating that at time *t=3* the active migration-bands incoming to population A are B→A and C→A:

2-4 B→A,C→A

November 2019

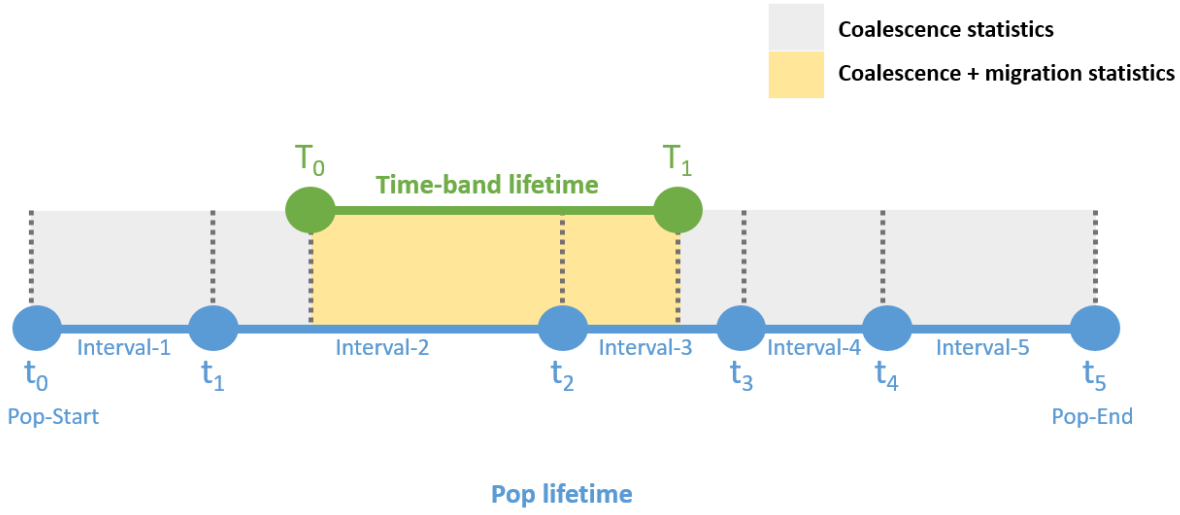
A demonstration of a use of function getLiveMigBands()

Following is a demonstration of a use of function `getLiveMigBands()`. We will see how it is used in function `recalcStats()`, which is a method of class *LocusPopIntervals*.

Function `recalcStats()` re-calculates the statistics of a given population. It does is by looping over all population's intervals. For each interval, it calculates the coalescence statistics, as a function of current time. The migration statistics, however, are calculated only for migration-bands living in that time. To know which migration-bands are living in a current point *t*, we call `getLiveMigBands()`, which returns an object of *TimeMigBand* struct (we will refer it as time-band), containing *t*.

However, as shown in the following figure, the start\end points of intervals and time-bands are not necessarily overlap. Thus, to know if migration statistics should be calculated or not, we want to partition the intervals into time segments.

Figure: blue line represents a timeline of a population, and the circles marks events. Green line represents a time-band lasting from T_0 to T_1. Time-band partially overlaps with interval-2 and interval-3. Migration statistics are calculated only for the two time-segments colored in yellow.



Pseudo-code is in next page:

First we get the POP-START interval of a given population, and get its age t. We also get the migration-bands living in t. Now we start iterating the intervals of population.

First we update t to be the minimum between the age of current interval, and the end time of time band, minus current t.

Consider a function that computes the migration and coalescence statistics in a given population by traversing the list of intervals associated with that population. Below is a textual description of such a function coupled with pseudocode with comments indicating different stages.

 # 1) The loop that traverses the population intervals keeps track of three main variables: the current interval, the current time and the current list of incoming migration bands. Before the loop starts, these variables are initialized to: (1) the first interval in the population (an empty population start interval with $\Delta t=0$), (2) the start time of the population, and (3) the list of migration bands incoming to that population at that time.

 # 2) In each iteration of the loop, the next time step is set to the earlier among the two events: (1) the next change in incoming migration bands, or the end time of the current interval.

 # 3) The main operation of the loop is executed. It depends on: (1) the number of lineages in the current intervals, (2) the time interval until the next step (as computed by #2 above), (3) the type of the current interval, and (4) the current list of incoming migration. The operations are carried out in the loop between step #2 and step #5.

 # 4) If the next event determined in step #2 is a change in the list of migration bands, then the list of migration bands is updated. If the list is null, it means that we should have reached the end of that population (note that an empty list is not a null list). The function makes the appropriate assertions, and if the end of the population was reached, the loop halts.

 # 5) If the next event determined in step #2 is a change in the number of lineages (interval change), then the current interval should not be the last one, and it is updated to the next one in interval list.

```
void recalcStats(int pop)
        ...
        // get pop start interval# 1 initialize main loop variables
        PopInterval *pInterval = getPopStart(pop)

        //get current age
        double currAge = pInterval->getAge()

        //get live migration bands
        TimeMigBands *timeBand = getLiveMigBands(pop, currAge)

        // follow intervals chain
        while (true):

                // #2 set next time stepget minimum between interval's age and end-time of time band
                double t = min(pInterval->getAge(),   timeBand->endTime)   -   currAge
                currAge += t[                          I1]
                ...

                //for each live mig band update mig statistics #3 main loop operations – e.g. update
                coalescent statistics and migration statistics
                for each migBand in timeBand->migBands:
                        …

//update current age
                currAge += t

                //if interval age is larger than end of time band, get next time band #4 update mig band list
                if (timeBand->endTime   <=   pInterval->getAge())
                        double prevEndTime = timeBand->endTime
                        timeBand = getLiveMigBands(pop, currAge)

                        if (!timeBand)
                                assert(pInterval->isType(POP_END))
                                assert(prevEndTime == pInterval->getAge())
                                break //here we get out of loopbreak out of loop !!

                        continue

                //assert #5 update interval
                assert(!pInterval->isType(POP_END))

                //switch-case according to interval type
                switch (pInterval->getType())
                        ...

                //get next interval
                pInterval = pInterval->getNext()

        //end of while loop
//end of function
```

November 2019

Appendix

Code

- Source code of version v140 can be found on GitHub, **branch v140**:
  https://github.com/gphocs-dev/G-PhoCS
- To **compile** files, enter directory and type:

```
cmake CMakeLists.txt
make
```

- New files added in version v140:
  - `AllLoci.h/.cpp`
  - `DbgErrMsgIntervals.h/.cpp`
  - `LocusEmbeddedGenealogy.h/.cpp`
  - `LocusGenealogy.h/.cpp`
  - `LocusPopIntervals.h/.cpp`
  - `PopInterval.h/.cpp`
  - `TreeNode.h/.cpp`

Development

- **IDE:** code was developed in **CLion** – an IDE for C and C++. Students and faculty staff members can get licenses by registering with their faculty e-mail.
- **Environment:** Ubuntu

Execution options

- **Execution with validation tests**

  New DS classes of v140 has validation functions (usually start with "test"), which verify the consistency of the new DS with the old DS. By default, these functions are muted (by surrounding them with `#ifdef`).

  To run with tests, uncomment command

  ```
  #define TEST_NEW_DATA_STRUCTURE
  ```

  In file `GPhoCS.h`.

- **Execution with time measurements**

  A version with time measurements can be found in (git) branch `v140_measure_times`.

This branch is version v140 + the addition time measurements. It evaluates the time duration of some of the main functions in the new DS, using the `std::chrono` library.

To run this option, checkout to branch `v140_measure_times` before running.

Output is written to the `benchmark*.out` file.

Scripts

Following is a description of scripts useful for development.

Scripts can be found on `compbio.idc.ac.il` server in path: `ROOT/benchmarks/scripts/`

ROOT = `/home/nomihadar`

- **`run-benchmark.sh -`** run a benchmark.

  Args:

  - `benchmarkID` – id of benchmark (e.g. "benchmark2b").
  - `GPhoCSBinary` – path to the binary file of GPhoCS (generated by `make` command).
  - `outputDir` – output directory.
  - `numIter (optional)` – number of MCMC iterations.
  - `benchmarkRoot (hard coded within script file)` – path to directory containing the benchmarks data (directories `ctlFiles, seqFiles`).

```
run-benchmarks.sh benchmarkID GPhoCSBinary outputDir (numIter)
```

- **`run-all-benchmarks.sh -`** run all benchmarks (each in a separate directory).

```
run-all-benchmarks.sh GPhoCSBinary outputDir (numIter)
```

- **`summaryBenchmarks.py`** – summarize into a table the results of benchmarks running (see below). Table columns:
  - `Errors:` whether benchmark terminated o.k. (by checking the `benchmark*.err` file).
  - `Num MCMC iterations:` requested number of MCMC iterations.
  - `Num iterations in trace:` number of completed MCMC iterations (number of lines in the `trace` file).
  - `Time`: running time in format *hours:min:sec*.

  Args:

- `benckmarksOutputDir` – path to root directory of benchmarks' outputs.

Output: CSV file "`benchmarks_summary.csv`".

(see examples in benchmarks section below).

```
python summaryBenchmarks.py benckmarksOutputDir
```

- **`compare-all-benchmarks.sh` –** compare traces from two separate runs, for all benchmarks. Mainly useful to compare development version to master version.

  Args:
  - `rootDir1` – root directory of benckmarks' outputs of first running.
  - `rootDir2` – root directory of benckmarks' outputs of second running.

```
compare-all-benchmarks.sh rootDir1 rootDir2
```

- **`compare-two-traces.sh` –** compare two traces files. First trace is a trace of an v130 execution (control trace), and second trace is of v140 execution. It checks if they are equal, and if not, outputs the difference.

  Args:
  - `trace1` – path to a trace file outputted by version v130.
  - `Trace2` – path to a trace file outputted by version v140.

```
compare-two-traces.sh trace1 trace2
```

November 2019

Benchmarks tests

Following is a summary of the benchmarks tests that were conducted.

ROOT=/home/nomihadar

Test 0 – control:

Path: ROOT/benchmarks/outputs/GPhoCS_master/

Version: v130 (branch master)

Result: all benchmarks terminated without error, and completed all required iterations.

File	Errors	Num MCMC iteration	Num iterations in trace	Time
benchmark1.out	no	300000	300000	33:48:02
benchmark2.out	no	50000	50000	04:18:56
benchmark2b.out	no	50000	50000	04:17:15
benchmark2c.out	no	50000	50000	04:16:31
benchmark2d.out	no	50000	50000	04:17:37
benchmark2e.out	no	50000	50000	01:48:45
benchmark2f.out	no	50000	50000	04:19:44
benchmark2g.out	no	50000	50000	04:24:26
benchmark3.out	no	50000	50000	01:48:26
benchmark3a.out	no	50000	50000	01:45:46
benchmark3b.out	no	50000	50000	01:48:28
benchmark3c.out	no	50000	50000	01:45:18
benchmark3d.out	no	50000	50000	01:48:37
benchmark3e.out	no	50000	50000	01:48:48
benchmark3f.out	no	50000	50000	01:48:08
benchmark3g.out	no	50000	50000	01:45:15

Test 1 – v140:

This is the standard execution of v140.

Path: ROOT/benchmarks/outputs/GPhoCS_v140/

Version: v140 (branch v140)

Result: all benchmarks terminated without error, and completed all required iterations. Traces are consistent with the traces of master version. Traces comparison result can be found here: ROOT/benchmarks/outputs/compare_traces_master_vs_v140/

File	Errors	Num MCMC iteration	Num iterations in trac	Time
benchmark1.out	no	300000	300000	149:03:38
benchmark2.out	no	50000	50000	13:22:49
benchmark2b.out	no	50000	50000	13:21:03
benchmark2c.out	no	50000	50000	13:20:00
benchmark2d.out	no	50000	50000	13:21:15
benchmark2e.out	no	50000	50000	05:28:00
benchmark2f.out	no	50000	50000	13:22:26
benchmark2g.out	no	50000	50000	13:31:31
benchmark3.out	no	50000	50000	05:21:27
benchmark3a.out	no	50000	50000	05:19:00
benchmark3b.out	no	50000	50000	05:21:03
benchmark3c.out	no	50000	50000	05:17:44
benchmark3d.out	no	50000	50000	05:21:13
benchmark3e.out	no	50000	50000	05:21:33
benchmark3f.out	no	50000	50000	05:19:26
benchmark3g.out	no	50000	50000	05:15:30

Test 2 – v140 + validation:

In this test, validation tests were activated (see " Execution options" above).

Path: `/home/nomihadar/benchmarks/outputs/GPhoCS_v140_with_tests/`

Version: v140 (branch v140)

Result: all benchmarks terminated without error, and completed all required iterations, except of benchmark1 (needed more time). This result means that the new DS is consistent with the old DS.

Test 3 – v140 + time measuring:

In this test, time measurements were taken (see " Execution options" above).

Path: `/home/nomihadar/benchmarks/outputs/GPhoCS_v140_measure_times/`

Version: v140 (branch `v140_measure_times`)

Result: all benchmarks terminated without error, and completed all required iterations (number of iteration was limited to 5000). Times are written to the end of files `benchmark*.out`. According to results, the most time consuming function is the function which copy the intervals (from proposal to original and vice versa (see " Proposals mechanism").

Others

November 2019

- UML diagrams in this document were created with **Visual Paradigm** software, which require a license (6$ per month).