Efficient synchronization of persistent DAGs

Matthieu JOURNAULT

24/06/2013

Contents

Chapter 1

Introduction and scientific context

This report gives the headline and the results of my two months of internship with the NetOS[1] (Networks and Operating Systems group) group in the Computer Lab in Cambridge. My work has been supervised by Thomas Gazagnaire. The NetOS group is included in the System Research Group of the Computer Lab. One of the project of the NetOS team is MirageOS[2], which is a library operating system that constructs unikernels for secure, high-performance network applications across a variety of cloud computing and mobile platforms. As a part of the MirageOS project, I worked on Irmin, a Git-like distributed, branchable storage.

The Irmin storage enables the user to synchronise distributed data structures. Those data structures are persistent graphs that are Merkle DAGs (see [1]) which are used by many systems (such as Git, Ori (see [10]) , etc ...). My contribution to the project was the elaboration of an efficient algorithm that enables users to synchronise their history in an asymetric way (meaning that a client pulls data from a server) which is to compute the difference $G \setminus G'$ for 2 graphs described as before. We did not make any assumption on the shape of the DAGS, meaning that the detailed algorithm can be used for any number of processes working simultaneously.

In order to underline the results of the intership as well as the research that has been done, this report starts with a presentation on the state of the art in terms of DAGs synchronization (Vector Clock) followed by a presentation of Bloom filters (as they are used in the algorithm). This introduction is followed by a presentation of the results of the internship : some preliminary results, the main algorithm (which signature can be found in the Appendix), some considerations on the hash family that are used for the Bloom Filters and an evaluation of the algorithm.

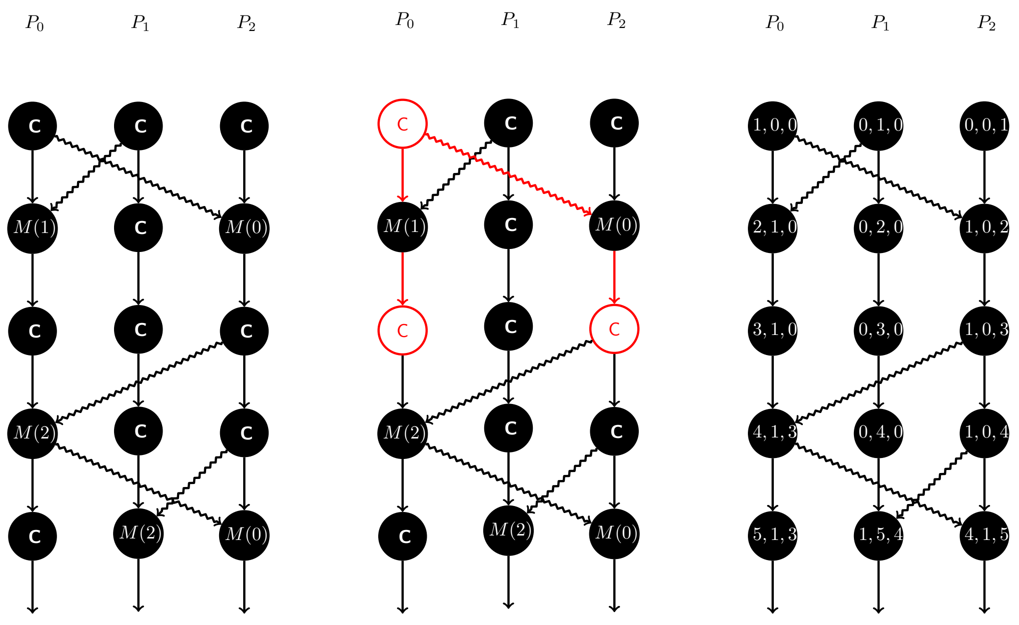
[1]http://www.cl.cam.ac.uk/research/srg/netos/
[2]http://www.openmirage.org/

Chapter 2

Research subject

2.1 Introduction to the problem

We assume that we have a collection of processes $P_0, \cdots, P_{n-1}$ that can communicate by exchanging messages. In our work we will assume that the processes can do two main actions that are "commiting" local data and "merging" remote data with an other process. For simplicity, we assume merge is always between 2 processes, we could easily generalise it to n-process merge. Each of the process P_i is doing a serie of action in a certain total order $\prec_i$, each nodes resulting in a new state for the process.



(a) Three processes sharing information

(b) Biggest common ancestor between two nodes

(c) Labeling Nodes with partial order

Figure 2.1: A known number of processes sharing information over time

Figures 2.1a shows an example with three processes commiting (C) and Merging with other process ($M(i)$). This figure shows an underlying partial order between elements as defined by Leslie Lamport (see [9]).

Definition 1. We define the relation $<$ as the smallest relation on the nodes following the conditions :

1. If a and b are two nodes of the same process P_i and $a \prec_i b$ then $a < b$

2. If b is a merge state and a is the state from which the merge occured on another process then $a < b$

3. If $a < b$ and $b < c$ then $a < c$

In our case we assume the relation $<$ to be a strict partial order, ie there is no cycle. We define the set of "Biggest common ancestors of a and b" (BCA) as : $\{c, c < a \wedge c < b \wedge (\forall d\ , c < d \Rightarrow (d \not< a \vee d \not< b))\}$ see Figure 2.1b. This set can be of any size, however Figure 2.3 underlines what having two biggest common ancestors means in term of merging.

As explained in the introduction, our aim is to synchronise DAGs in an asymetric way, meaning that a client pulls information from a server. Considering that the history of the server is a graph G and the history of the client is a graph G', we want to compute the difference $G \setminus G'$ so that the client is able to add the history of the server to its history. If we consider the complete graph $G \cup G'$ and compute in this graph the set of Biggest Common Ancestors of the two most "recent" nodes in G and G' we are able to compute the difference $G \setminus G'$ by taking all the descendants of the BCA that belong to G. Therefore it is interesting to find a way to discover quickly and without exchanging too much informations between two processes what are the biggest common ancestors of two nodes. For this purpose it is a common practice to use vector clock (see [5] and [11]), we label each of the states with a vector of integers of size n, n being the total number of processes exchanging information. δ_i denotes the vector having zeros everywhere except a 1 at the i-th position. We build the label of the states in the "$<$" order:

1. If a state a is a commit state on a process P_i and has no predecessor for the $\prec_i$ relation then a is labeled with δ_i

2. If a state a is a commit state on a process P_i and it has a biggest predecessor b then a is labeled with $\text{label}(b) + \delta_i$

3. If a state a is a merge state with a state b and a as no predecessor for the $\prec_i$ relation then a is labeled with $\text{label}(b) + \delta_i$

4. If a state a is a merge state with a state b and it has a biggest predecessor c then a is labeled with $\max(\text{label}(b), \text{label}(c)) + \delta_i$. Where max is the max on each components of the vector

Such a labeling can be seen on Figure 2.1c. We define the $\preccurlyeq$ relation on integer vector of size n by : $u \preccurlyeq v \Leftrightarrow \forall i \in \{0, \cdots, n-1\}\ u(i) \leq v(i)$ and $\prec$ defined by $u \prec v \Leftrightarrow u \preccurlyeq v \wedge u \neq v$. As shown in [4] and [7] we have the following invariants regarding the labels of the states :

Proposition 1. $a < b \Leftrightarrow \text{label}(a) \prec \text{label}(b)$

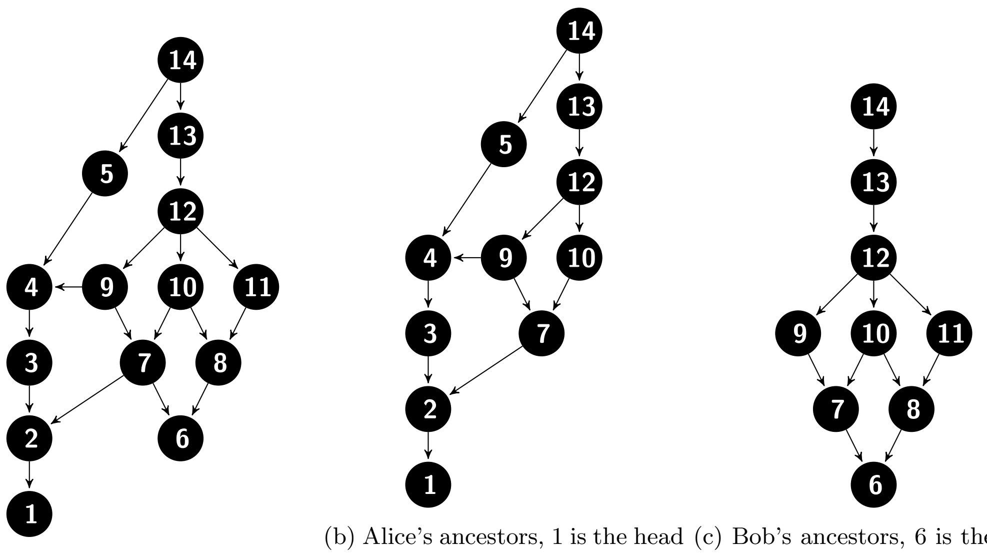
Proposition 2. If c is the only biggest common ancestor of a and b then $\text{label}(c) = \min(\text{label}(a), \text{label}(b))$

These two propositions allow processes to find their common ancestor just by computing the min with the label received. Once the biggest common ancestor is found, the difference of history between the two nodes can easily be computed and shared so that one node know the history of the other.

In all of the previous remarks we considered the total number of processes known in advance, however in many cases it is not true, for example in Git one does not know how many contributors will be part of a project at the end. Vector clocks are nice and easy to use : they can be built incrementally when merging or commiting, a minimum computation gives the BCA when there is only one, however there is no bound on the indexes in the vector which size must be set at the beginning. In section 3.1 we tried to adapt the work on Vector Clocks to our problem. Therefore we are working on some more generic assumptions :

- **two nodes can have more than one biggest common ancestor**
- **there can be any (possibly large) number of processes involved**

The graphs we are working on are persistent in the sense that the past does not change, we only add elements on the top, and computing `pred`(x) is cheap. We will pay a particular attention towards optimizing the cases where the difference between the two DAGs we want to synchronise is small. The figure 2.2a gives an example of DAGs we will be considering, and of the history of two processes Alice (Figure 2.2b) and Bob (Figure 2.2c) that have some history in common.



(a) Synchronised Graph (b) Alice's ancestors, 1 is the head of Alice (c) Bob's ancestors, 6 is the head of Bob

Figure 2.2: two processes Alice and Bob that share a part of their history

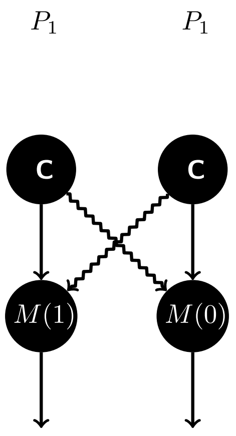


Figure 2.3: Two biggest common ancestors

2.2 Introduction to Bloom filters

The previous remarks underline the idea of using arrays labeling the nodes of the DAGs, which is the reason why we tried to use Bloom filter to elaborate our algorithm. A Bloom filter (BF) is a space-efficient probabilistic data structure that was conceived in 1970 by Burton Howard Bloom (see [2]). This data structure is particularly efficient for adding an element or testing wether an element is in a set or not. This efficiency is at the cost of having some false positive but no false negative in the membership test. Here we will present the Bloom filters as they were used during my internship, however a lot of different variations have been used on Bloom filters (see [3] or [6]). Let us consider a set S, a subset $S' = \{x_0, \cdots, x_{n-1}\} \subset S$, we assume moreover that we have k hash functions $\mathcal{H} = \{h_0, \cdots, h_{k-1}\}$ from

S to $\{0, \cdots, m-1\}$ and a matrix T of size $k \times m$ bits initially to zero. The BF of S' is :

$$T(i)(j) = \begin{cases} 1 & \text{when } \exists x \in S',\ j = h_i(x) \\ 0 & \text{elsewhere} \end{cases}$$

To test the membership of $x \in S$ in the Bloom filter T we check wether $\forall j \in \{0, \cdots, k-1\}$, $T(j)(h_j(x)) = 1$. This test can however raise false positive, as shown on Figure 2.4.

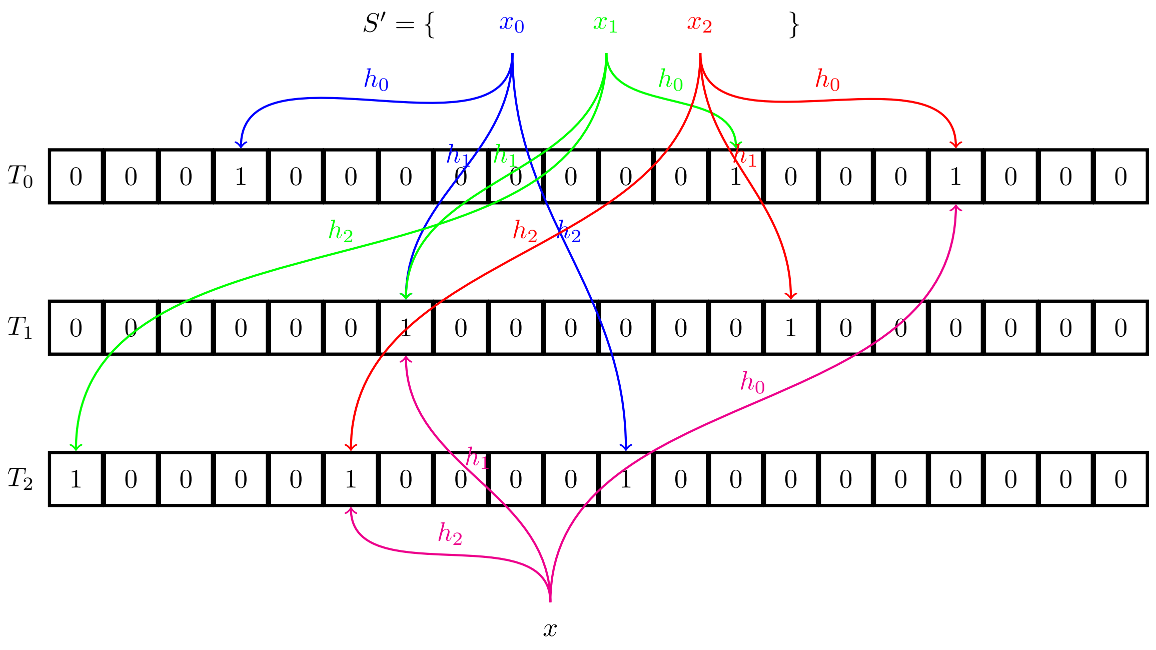


Figure 2.4: Inserting and testing the membership in a Bloom Filter, x is a false positive

Proposition 3. Under the assumption that the set $\mathcal{H}$ is a set of size k independant identically distributed hash functions ranging in $\{0, \cdots, m-1\}$, a BF T and a set $S' \subset S$ of size n, then the probability $\mathbb{P}$ of having a false positive while testing $x \in S'$ is :

$$\mathbb{P} = \left(1 - \left(1 - \frac{1}{m}\right)^n\right)^k$$

Proof. see section 5.2 $\square$

Chapter 3

Results

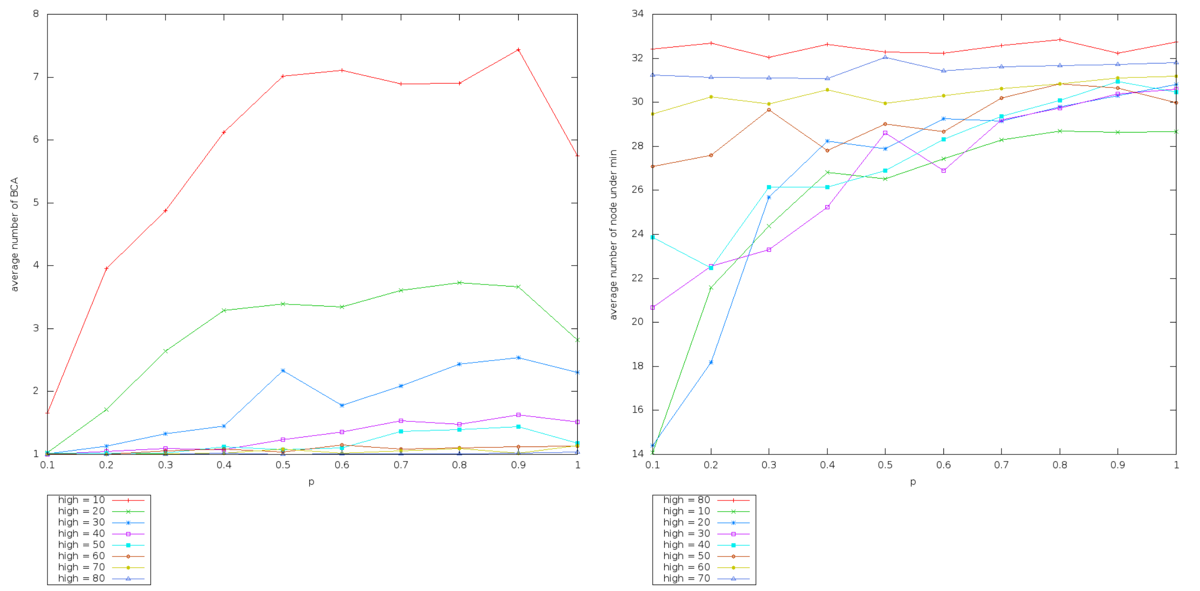
3.1 Preliminary result on DAGS and Bloom filters

As we change the problem from a finite and known number of processes with at most one BCA to an unknown number of processes with no restriction on the number of BCA, it is important to look for what is likely to change. Therefore we implemented a DAG random generator (see section 5.1) in order to test the possible number of BCA.

At the same time we implemented the same algorithm that the one used on a finite and known number of process, except that we now assume that each node a of the DAG contains a hash $h(a)$ of size k. We label every node a of the tree with a vector $v(a)$ corresponding to a BF of the predecessors, hence $v(a)_i = \sum_{b \in \text{ancestor}(a)} h(b)_i$, with this labeling we test wether the proposition 2 can still be used in some way. From the definition of the labeling we have a weaker version of proposition 2 and we wanted to experiment wether we could use it or not.

Proposition 4. If c is a biggest common ancestor of a and b then $v(c) \preccurlyeq \min(v(a), v(b))$.

In our DAG algorithm nodes are placed in different layers, each layer having links only with the previous and the next one. An important parameter to consider is the probability for a node that a node from the previous layer is one of its parents. Examples of the influence of this parameter (called p) can be found in section 5.1. The "height" of a DAG is the length of the biggest path between two nodes of the DAG. The next graphs show the influence of p and the high of the DAG on the number of BCA and on the number of element verifying the proposition 4.



(a) Number of Biggest Common Ancestor in a DAG with 100 nodes

(b) Given a and b, representation of the number of nodes c verifying $v(c) \preccurlyeq \min(v(a), v(b))$

Figure 3.1

Figure 3.1a shows that the average number of BCA is 2, this underlines that the assumption on the number of biggest common ancestors was too strong and that we can not rely on the case where there is only one BCA. Figure 3.1b shows that the inequality $v(c) \preccurlyeq \min(v(a), v(b))$ holds (in average) for $\frac{1}{3}$ of the nodes, hence adapting the algorithm from the case where we only have a finite and known number of processes might not be a good idea, indeed we reduce the search of the biggest comman ancestor to $\frac{1}{3}$ of the DAG, but the size of the area to search remains linear in the size of the complete DAG.

3.2 Algorithm

3.2.1 General algorithm

In the following we will consider two DAGs, each with some particular nodes called head. The set of heads is the smallest set so that every node in the considered DAG is an ancestor of one of the head. One of the DAG will be refered to as Client or *Bob* and the other one as Server or *Alice*. Ancestor($Alice$) and Ancestor(Bob) are the considered DAGs and the aim of the algorithm is to be able to compute Ancestor($Alice$) $\setminus$ Ancestor(Bob) $= \{x \ , \ x \in \text{Ancestor}(Alice) \wedge x \notin \text{Ancestor}(Bob)\}$. The complexity of the algorithm will be expressed in terms of $n = \#(\text{Ancestor}(Alice) \setminus \text{Ancestor}(Bob))$. In terms of application, the client and the server are two entities that are physically distinct, therefore it was important during the design of the algorithm to keep in mind and to be aware that :

1. The quantity of information exchanged between the two processes shall be the smallest possible
2. We want to minimise the complexity on both processes
3. There can be no no pre-assumptions on the shape of the graph, because the shape of the global DAG (with multiple head) is evolving in time and is not known at the beginning.
4. There is no high authority that has a memory of everything.

The section 3.1 underlined that we were not able to find a way to compute easily the set of BCA, moreover if the server has to send the difference Ancestor($Alice$) $\setminus$ Ancestor(Bob) then at some point we will have to go all over this difference, therefore we decided that the server could cross the part of its history corresponding to the difference. I now describe the algorithm that I developed during my internship. Starting with some definitions :

Definition 2. Given a DAG $\mathcal{D}$ and a partition of a the nodes of $\mathcal{D}$, we call a set of equivalent nodes for this partition a "slice of $\mathcal{D}$"

Definition 3. Given a DAG $\mathcal{D}$ and a slice $\mathcal{S}$ we call the border of $\mathcal{S}$ (noted $\overline{\mathcal{S}}$) the smallest set such that $\forall x \in \mathcal{D},\ \forall y \in \text{predecessor}(x)\ y \in \mathcal{S} \vee y \in \overline{\mathcal{S}}$.

Idea of the algorithm We assume that a partition of its DAG of ancestors is known by the client as well as a border for every slice of the partition.

1. The client sends the list of its head, one of its slice and the union of all the border to the server
2. The server goes up in its history and stop whenever it crosses a node in the slice or in the border.
   (a) If it crosses a slice, it means the server found a common ancestor with the client
   (b) It it crosses a border, it means the exploration is pointless and should be stopped
3. The server computes the DAG D (which is part of the difference) of the successors of every nodes it found in slice.
4. The server computes a list of "nodes of interest" from which it shall start again the exploration (those "nodes of interest" are the last successors of the nodes found in the border that are not in D, they are the node from which the server should start again the exploration later).
5. The server sends back D and the list of "nodes of interest"
6. The client starts again with anoter slice and instead of the list of its head it sends the list of "nodes of interest" it received from the server, except if there is no "node of interest" in which case the algorithm stops.

This algorithm has the property that the server has absolutely no memory of the previous client requests, hence it can be done in parallel for a large number of concurrent clients. The figure 3.2 shows a partition of two dags in slices.

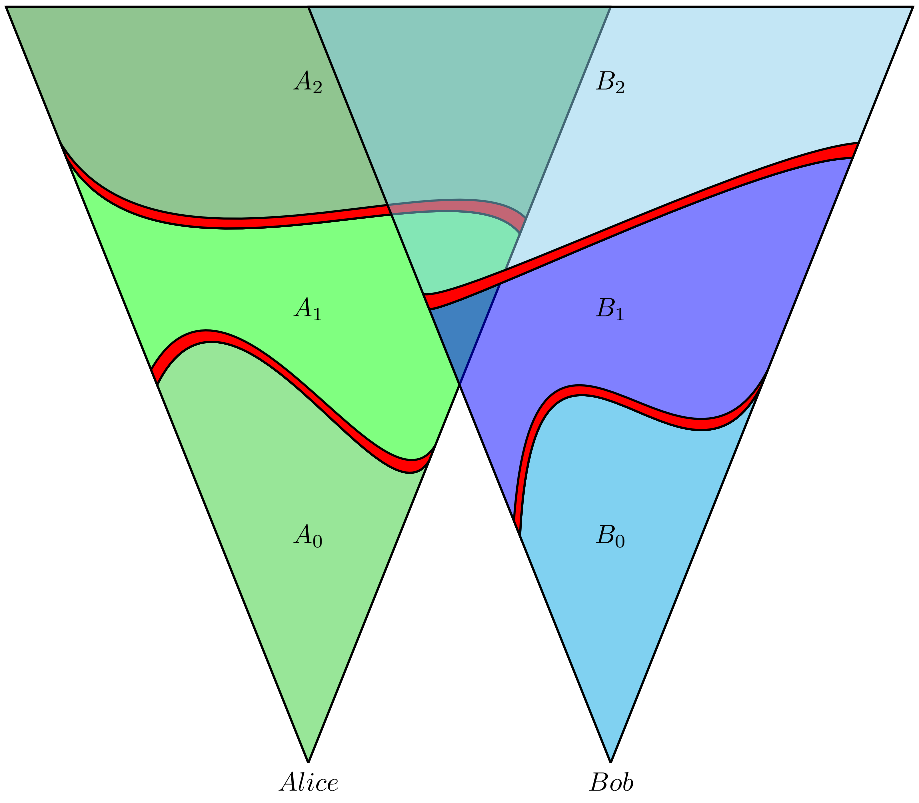


Figure 3.2: Two processes sharing a part of their history and their decomposition in Slices and Borders

So far, we have made no assumption on how to choose slices and borders. The figure 3.2 underlines the fact that it is interesting to have a "good" partition of the node, in the sense that :

1. We want the borders to be small so that we do not have to send a lot of information
2. We want to do the least sending/receiving data to/from the server possible
3. We want those slices/border to be easy to compute, in the sense that we do not want to cross all of the history of the client each time we want to merge
4. We want the slices to be all of the same size

On the client side, slides and borders can be computed incrementally. Let us assume we decided that the slices should be of size n and that a process already knows a partition $\mathcal{P} = \{A_0, \cdots, A_m\}$ in slices of its ancestors and the paired borders. This process needs to add some nodes S in its history, wether because an event happened locally or because a server sent a DAG of nodes. We start by filling the set in the partition with a size smaller that n, by adding the element from S, if S is a single element we add it, but if S is a DAG of nodes, we add them in the order of the DAG. Therefore if S contains two nodes a and b, a being a predecessor of b then a will be added prior to b. Once every partition is of size n we add a new slice to $\mathcal{P}$. While adding a node in one of the slices we add it in the border if one of its parents is not in the slice. The order in which we add the elements ensures us that the border will be the small (regarding the number of elements in the slice). Hence we can build a new partition and the corresponding borders quite easily from the previous one. Finally in order to ensure that there is not a lot of exchanges between the client and the server, the client sends its slices from the newest to the oldest one. For example in figure 3.2 Bob will be sending the slices in the order he built them, here that is : B_0 then B_1 then B_2. As for now we still have not made any assumptions on the way elements are stored, except that we need membership queries and merges. We could use normal sets but also BF (both case have been implemented). In the case of BF we have to take care of false positive (see section 3.2.2).

As this algorithm is designed for multi processes, we wrote functions that can be used on the client side and on the server sides. Such functions are placed in a functor with the following signature :

Arguments A module for DAGs signature and a datastructure module that enables merge and membership queries[1].

Invariants In order to explain the way these functions work, we detailed some invariants on those functions.

1.
   ```
   val next_ring : graph -> vertex list -> slice -> borders -> (graph *
       (vertex list))
   ```

   If `il` is a list of nodes in the history of A, `ia` is the list of the ancestor DAGs of the elements in `il`, if `bf` is a datastructure containing a slice of B and `bd` is the union of all the borders of B, if `g,l` = `next_ring il ia bf bd` then

   $$g = \{x, \exists x_0 < x_1 < \cdots < x_j = x < \cdots < x_n,\ (\forall i \in \{1, \cdots, n-1\},\ x_i \notin \texttt{bd} \cup \texttt{bf} \wedge x_0 \in \texttt{bf} \wedge x_n \in \texttt{il})\}$$
   $$l = \overline{g}$$

2.
   ```
   val increase_high :  state ->  vertex list -> vertex -> state
   ```

   If `s` is a state containing a DAG D of the ancestors of a list `old_hd_l` of nodes, then for all `new_hd`, `s'` = `increase_high s old_hd_l new_hd` is a state containing the DAG with head `new_hd`, which fathers are `old_hd_l` and D

3.
   ```
   val increase_width : state -> ( vertex list -> slice -> border ->
       (vertex list * graph)) -> vertex -> state
   ```

   If `s` is a state containing a DAG D, `f` is a function verifying the invariant of item 1 and `t` is a node in B then `s'` = `increase_width s f t` is a state containing the DAG D as well as t and all of its ancestors in B.

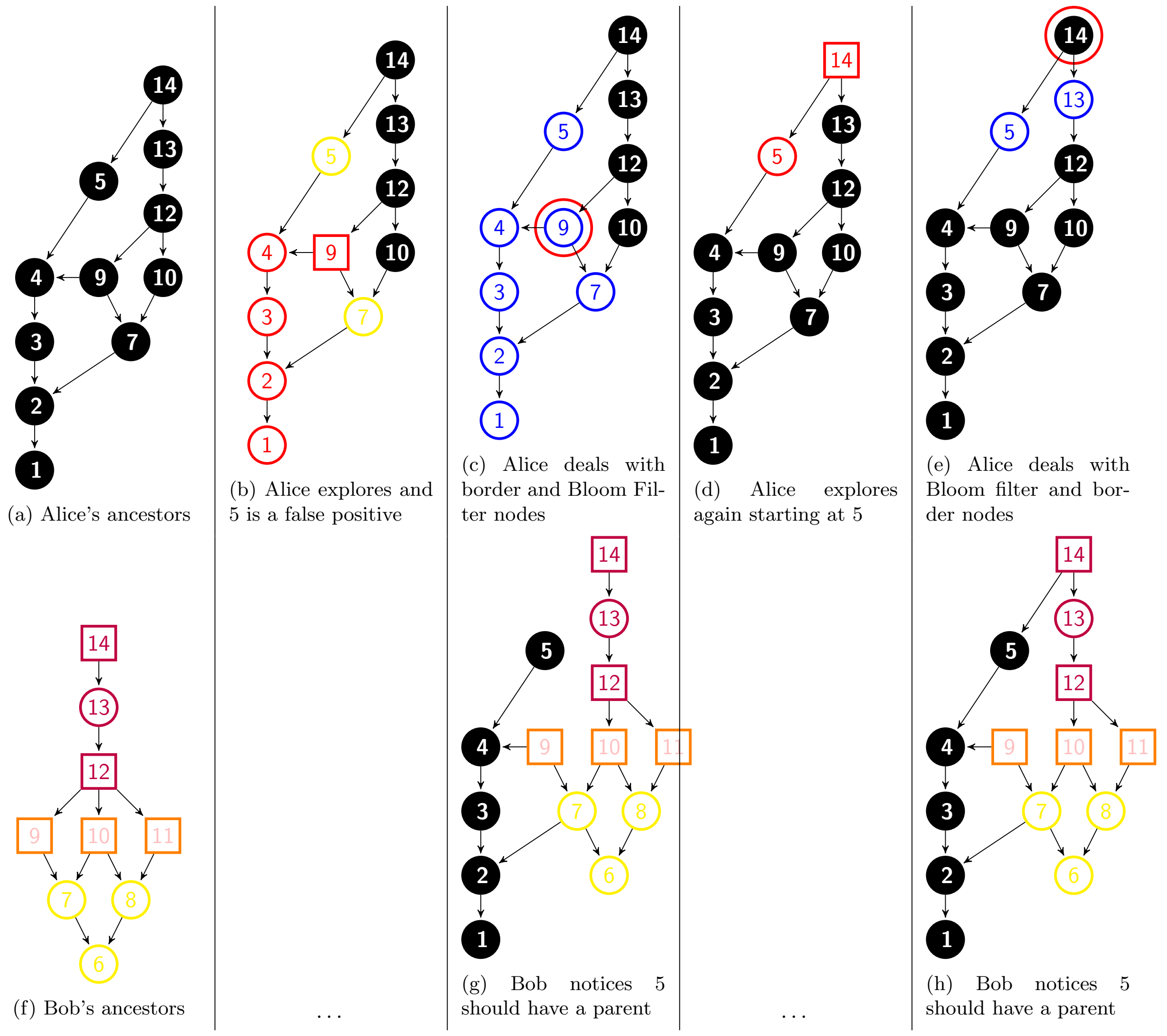
[1]The complete signature can be found on https://github.com/samoht/ocaml-bloom-filters

3.2.2 The False-positive case

Our algorithm uses Bloom filters, therefore some false positive membership queries can happen, even if we chose the hash family so that the probibility that a false positive occurs is small. In the algorithm, membership queries only occur when the server is computing the difference between its history and the client one using Bloom filter. Let us consider that the server just received a Bloom filter and a list of ”interesting nodes” from which it shall start exploring its ancestors, two different false positive cases can occur :

1. A false positive occurs in a slice, therefore the server will stop exploring up at this node and will add all of its sons to the ring to send to the client. However the client will receive the ring and some node will not have any fathers in its history, those nodes shall be those in the ”interesting nodes” list otherwise it means that it is a node the server considered in the history of the client and therefore can detect that a false positive occured. In such case the client can send back the same slice while signaling to the server which nodes were false-positive, hence the server will go accross the false-positive node and continue the exploration with the correct slice, for an example see 3.3.

2. A false positive occurs in a border, there is no way to discover such a false positive until the end of the algorithm, when the client has sent all of its Bloom filters. At this point if the algorithm occured without any false positive, there should be no more ”interesting nodes”, however in the case of a false positive in a border, there remains some ”interesting nodes”. Considering that the border can not be trusted to test the membership, the client sends its border to the server that sends back to the client every nodes in its history that are ancestors of the remaining ”interesting nodes” and that belongs to the client’s border. The client verifies that the received nodes are in its history. If a node is not in its history, then it raised a false positive in the border, once we know all the false positive nodes, we send them to the server that will ignore them while starting over all of the exploration. We can notice that if a node was in the history of the client but was raising a false positive in the border, then we do not have a problem because the node was already in the client history, as were all of its ancestors, therefore we do not need this node to be further explored on the server side. Having a false positive in a border is a very rare case because the borders are rather small.


(a) Alice's ancestors

(b) Alice explores and 5 is a false positive

(c) Alice deals with border and Bloom Filter nodes

(d) Alice explores again starting at 5

(e) Alice deals with Bloom filter and border nodes

(f) Bob's ancestors

. . .

(g) Bob notices 5 should have a parent

. . .

(h) Bob notices 5 should have a parent

Figure 3.3: How a false positive in a Bloom filter is detected and dealt with

3.2.3 Assymptotic behaviour

We assumed that the client and the server had a bounded memory, however the accumulation of slices and borders make the size of the history of the client and the servor grow linearly in the number of nodes added. To keep a memory bounded by a constant M the following algorithm ensures that merging two processes with a small difference in their history will still be efficient but we may lose efficiency when trying to find older nodes : When adding nodes we check wether the size is greater than M or not, if it is we partition the sorted list of slices in two and we remove half of the slices and according borders of the oldest half (we remove one slice out of two in the chronological order (which is the order in which the slices and borders are saved)), thus reducing the size to $\frac{3M}{4}$ and forgetting a part of the history. This algorithm will still be effective but we loose in efficiency beacause the server will assume that the client does not know some nodes it actually knew and the server will send them to the client. We keep the first half intact because in real-life applications most of the merging happens between the last nodes added to the history, ensuring that the algorithm will remain efficient on such merges.

3.3 Hash Functions

The question of choosing an interesting hash function family is very important in our problem because we often have to hash elements, in order to be able to store data or to test the membership of an element in a set. Therefore we want the hash functions to be fast to compute while being independant. In particular we will look for a set of hash functions that hashes integers in the set $\{0, \cdots, m-1\}$:

1. The multiplication method : we assume that we have a real number generator. We generate the hash family $h_\theta : x \to \lfloor m \text{frac}(x \times \theta) \rfloor$ where $0 < \theta < 1$ is a random real number.

2. The less hashing method : we assume that we have two hash functions h_a and h_b hashing integers to the set $\{0, \cdots, m-1\}$, we can generate the hash function family $\mathcal{H} = \{x \to h_a(x) + ih_b(x) \bmod m, i \in \{0, \cdots, k-1\}\}$.

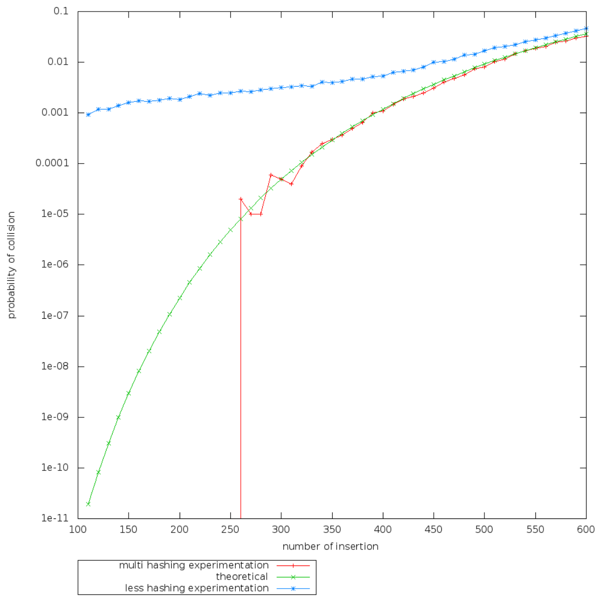


Figure 3.4: Testing the probability of getting a false positive with two different families of Hash functions

The graphic above shows that the less hashing method, even if easy to compute, can not be used in our case because of the high probability of getting false positive. However this hashing method could be really interesting when hashing in a space of size m growing linearly in the number of added element, because then it has the same asymptotic behaviour than an independant hash family (see [8]). Such a growth of the size of the hashing space can not be used in our case where we assumed that we have no bound on the size of the history. Therefore we shall use the multiplication method which needs more computation but is more reliable, and the figure 3.4 enables us to choose the number of elements we can insert in a slice given a bound on the probability of false positive.

3.4 Evaluation

There are three important evaluations that need to be made :

1. The complexity on the client side : it can be computed and it is linear in the size of the difference we want to compute.

2. The complexity on the server side : harder to compute because it depends on the Bloom filter false positive rates and on the shape of the DAG, therefore we tried to give an average estimation of the complexity

3. The quantity of exchanged information : directly linked to the complexity on the server side, therefore we want an evaluation of the number of pairs (slice,border) that need to be sent.

Therefore in this last part we will show some experimentations that have been done on the algorithm, mainly on the `next_ring` function.

The example that is detailed for the algorithm in Appendix (see section 5.3) uses three slices in order to fully discover the difference. The number of slices needed to synchronize is important as it is the number of times the algorithm will be restarted on the server side and as it will be exchanged over the network. The following curves give the average number of slices needed to synchronize regarding the height of the dag and the p factor (see section 3.1)

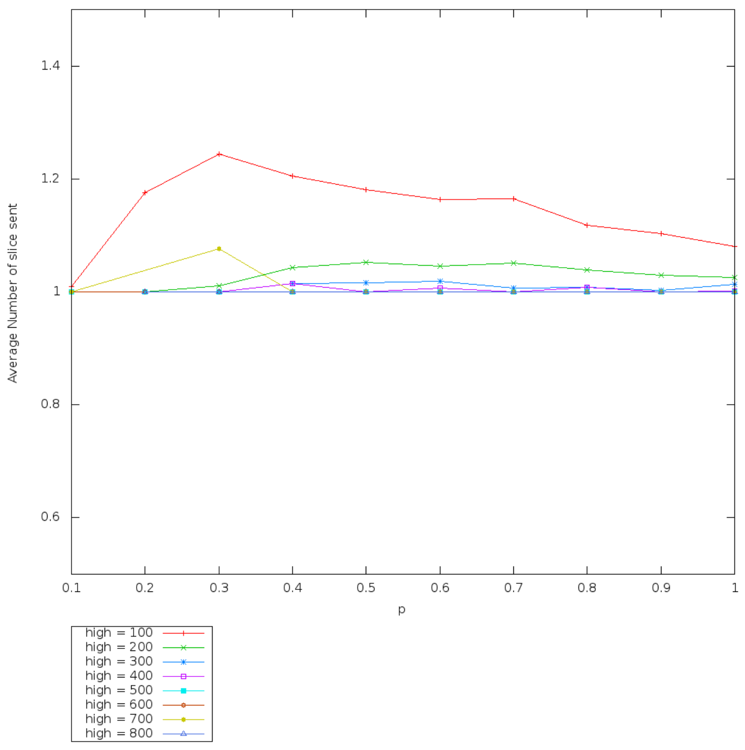


Figure 3.5: Graph of 1000 nodes with Slices of size 200

This graph underlines the fact that, whatever the shape of the DAG is, not a lot of slices are needed to synchronize. Therefore in this example the information sent by the client to synchronize fits in $20 \times 320 \times 1.1 = 7040$ bits. 20 is the number of hash functions we used, 320 is the size of each table, and 1.1 is the average number of slices.

The complexity of the `next_ring` algorithm is difficult to assert, given that it mainly depends on the shape of the tree. However we counted the number of calls to the functions `pred` and `succ` as well as the number of visits to each nodes. We chose to count those calls because they are the one with the biggest complexity. The algorithm can be implemented without any call to the `succ` function, but the complexity of the algorithm is the same and it is more complicated to implement. The following curves show the average values of these calls divided by the size of the difference between the DAGs depending on different values. Each DAG had 1000 nodes labelled with 160 bits hash.

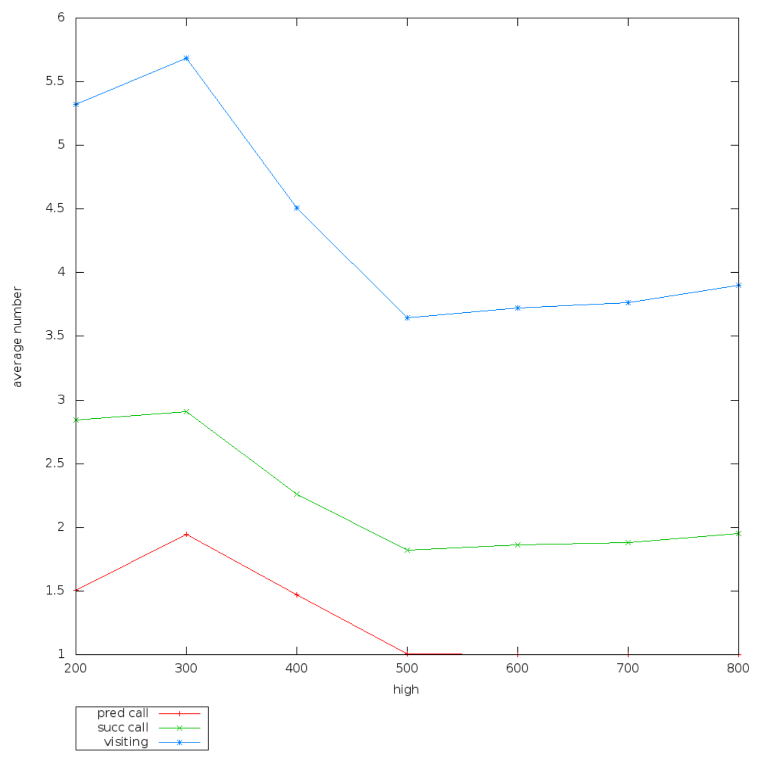


Figure 3.6: Number of operations depending on the height of the DAG

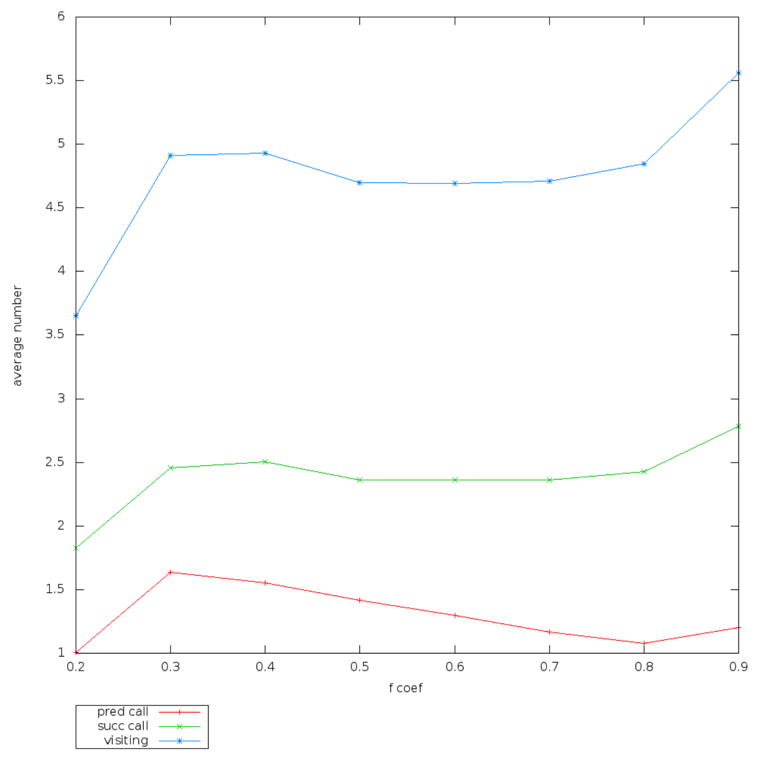


Figure 3.7: Number of operations depending on average number of predecessors

The two previous graphs (figure 3.6 and figure 3.7) underline that the complexity in terms of call to the `pred` and `succ` function is linear in average on the size of the difference between the two DAGs that are being synchronized, which is the main result we wanted. Indeed in the case where we want our algorithm to work it is important that the complexity does not increase with the size of the history but only with the size of the difference.

Chapter 4

Conclusion

During the months of June and July I decided to do my internship at the Computer Laboratory of Cambridge with Thomas Gazagnaire (former student from the ENS of Lyon and Rennes). I chose to do my internship with Thomas Gazagnaire and his team as much for the programming language they work with (OCaml) as for the projects of the team (MirageOS, Irmin, ...).

Thomas and I chose the subject of the internship before the beginning of it which allowed me to document myself on the field I was going to work on. We chose the aim of the internship at my arrival and therefore decided that I was going to try to adapt existing algorithms (as described in [9]) to other cases. However after some work we realized it was not going to work and therefore decided that we were going to try to establish a new algorithm using Bloom Filters, the idea of using Bloom Filters coming from the work we had already done, while trying to adapt algorithms. Once the algorithm established I had to implement, test it and shape it so that it could be used in a library, while documenting myself on the state of the art in terms of Bloom Filters and Hash function families (this part was made easier thanks to the help of Magnus Skjegstad, a colleague of Thomas). Once the code implemented we defined what were the invariants of the functions in the library we wrote.

In conclusion we underlined the difficulties of adapting the work already done in the cases where there are a finite number of processes and a unique biggest common ancestor. We wrote a library[1] of functions enabling a user to synchronize persistent DAGs, with a low cost in term of data exchanged and of complexity. We dealt with the side cases due to false positive in Bloom Filters and tried to find a "good" hash family function that ensures the user that the false positive rate will be low.

[1]https://github.com/samoht/ocaml-bloom-filters

Bibliography

[1] Georg Becker and Ruhr universität Bochum. Merkle signature schemes, merkle trees and their cryptanalysis.

[2] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.

[3] Benoit Donnet, Bruno Baynat, and Timur Friedman. Improving retouched bloom filter for trading off selected false positives against false negatives. *Computer Networks*, 54(18):3373–3387, 2010.

[4] Marcel Erné. Posets isomorphic to their extensions. *Order*, 2(2):199–210, 1985.

[5] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):56–66, 1988.

[6] Michael T. Goodrich and Michael Mitzenmacher. Invertible bloom lookup tables. In *Allerton*, pages 792–799, 2011.

[7] Denis Higgs. Lattices isomorphic to their ideal lattices. *algebra universalis*, 1(1):71–72, 1971.

[8] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. *Random Struct. Algorithms*, 33(2):187–218, 2008.

[9] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[10] Ali José Mashtizadeh, Andrea Bittau, Yifeng Frank Huang, and David Mazières. Replication, history, and grafting in the ori file system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 151–166, New York, NY, USA, 2013. ACM.

[11] Friedemann Mattern. Virtual time and global states of distributed systems. In Cosnard M. et al., editor, *Proc. Workshop on Parallel and Distributed Algorithms*, pages 215–226, North-Holland / Elsevier, 1989. (Reprinted in: Z. Yang, T.A. Marsland (Eds.), "Global States and Time in Distributed Systems", IEEE, 1994, pp. 123-133.).

Chapter 5

Appendix

5.1 DAG Generator

I wrote the DAG generator that we used thanks to the OCamlgraph Librairy. The Dag Generator is a functor with signature :

```
module type Elem =
sig
  type t
  (** [init n] initializes the Random element generator where [n] is a seed
  *)
  val init : int -> unit
  (** [next_item n] gives back a new element where [n] is the size of the new
      element
  *)
  val next_item : int -> t
end
module Make :
  functor (B : Graph.Sig.I) ->
    functor (L : Elem with type t = B.V.t) ->
      sig
  (** [alea n h t seed p] is the function that produces a dag
  and the biggest element of this dag where [n] is the
  number of nodes of the dag and [h] is the high of the dag
  and [t] is the size of the element in the dag and [seed]
  is the seed used to initialize the element generator and
  [p] is the probability that a node of high i is the son of
  a node of high (i-1)
  *)
  val alea : int -> int -> int -> int -> float -> B.t * B.V.t
      end
```

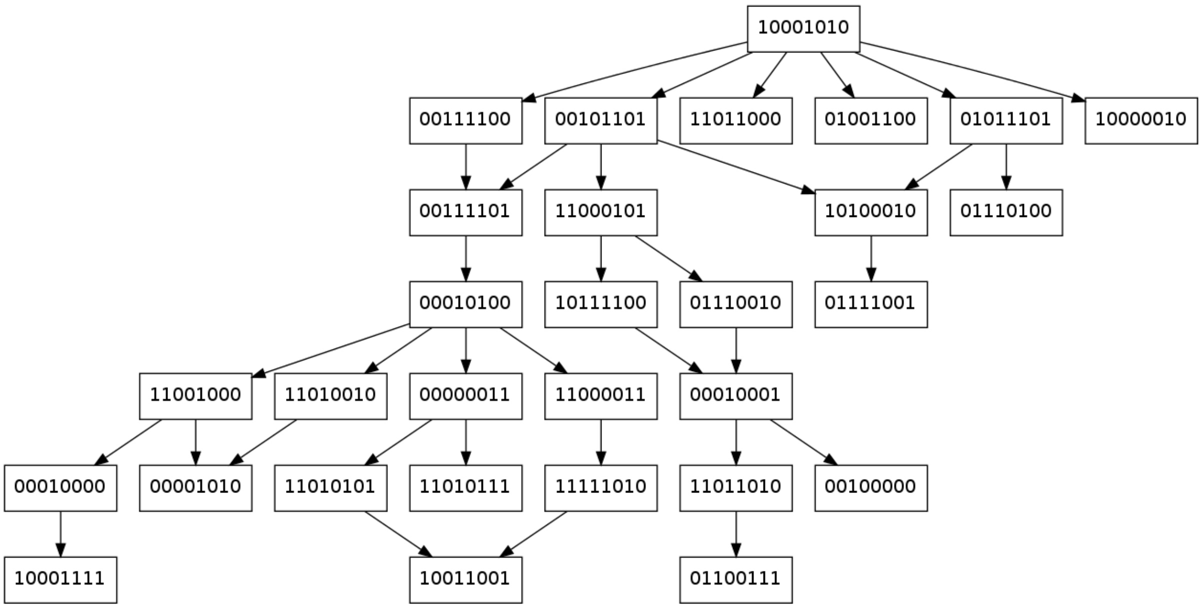


Figure 5.1: DAG generated with $p = 0.25$

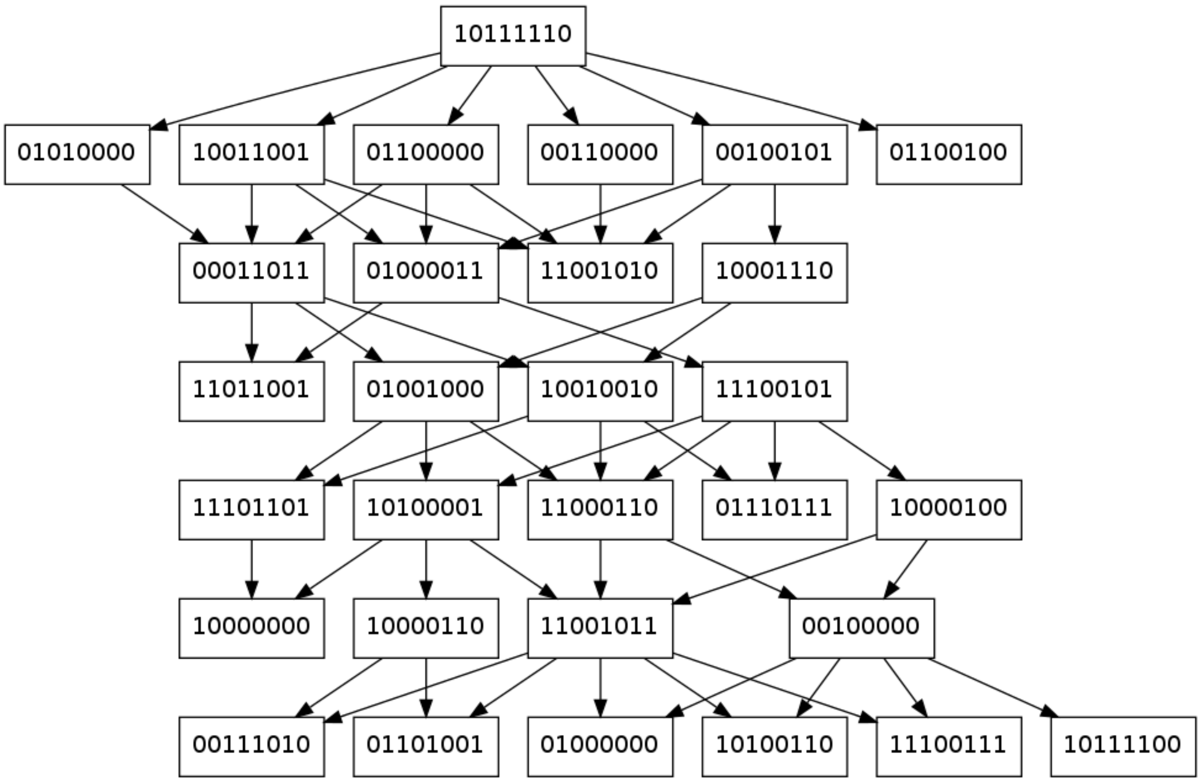


Figure 5.2: DAG generated with $p = 0.5$

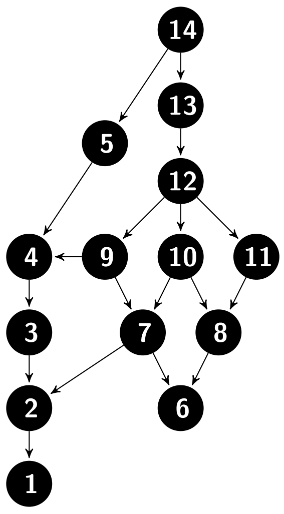
5.2 Probability of false positive

Proof. With the same notation than in the proposition :

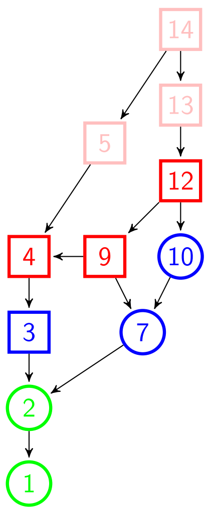
$$
\begin{aligned}
\mathbb{P} &= \mathbb{P}(\forall j \in \{0, \cdots, k-1\}, T(j)(h_j(x)) = 1) \\
&= (\mathbb{P}(T(0)(h_0(x)) = 1))^k \\
&= \left(\sum_{i=0}^{m-1} \mathbb{P}(T(0)(i) = 1)\mathbb{P}(h_0(x) = i) \right)^k \\
&= \left(\sum_{i=0}^{m-1} \mathbb{P}(T(0)(0) = 1)\frac{1}{m} \right)^k \\
&= (1 - \mathbb{P}(T(0)(0) = 0))^k \\
&= (1 - \mathbb{P}(\forall p \in \{0, \cdots, n-1\} h_0(p) \neq 0))^k \\
&= \left(1 - \left(\frac{m-1}{m} \right)^n \right)^k
\end{aligned}
$$

□

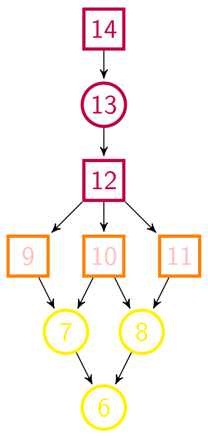
5.3 Example of application of the algorithm



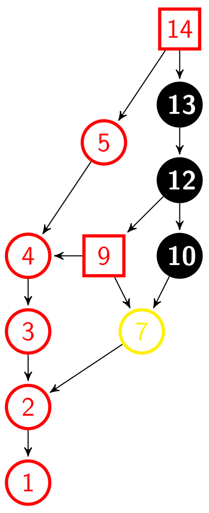
(a) Main Graph

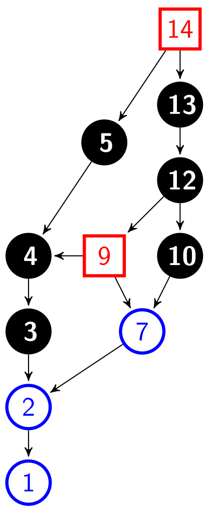


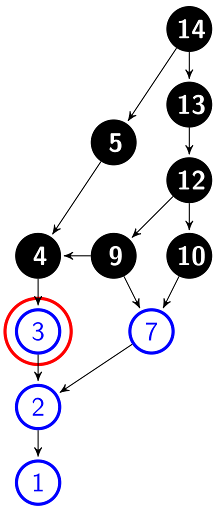
(b) Alice's ancestors

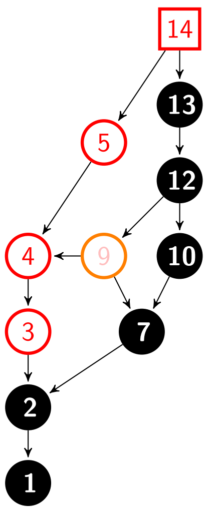


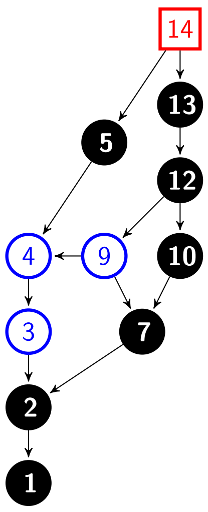
(c) Bob's ancestors

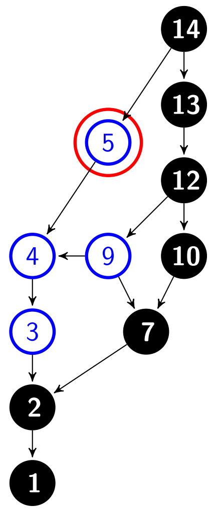

[1] Explore

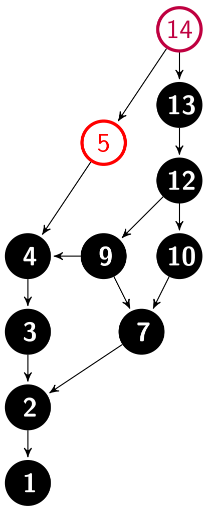

(a) deal with bloomfilters

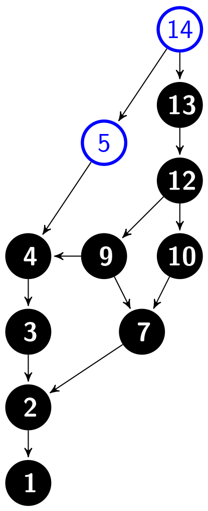

(b) deal with border

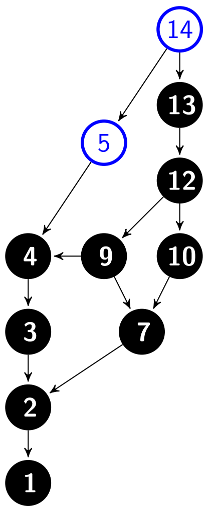

(a) explore


(b) deal with bloomfilters


(c) deal with borders


(a) explore


(b) deal with bloomfilters


(c) deal with borders

5.4 Detailed Algorithm

Algorithm 0: Finding some of the ancestors of *Alice* not known by *Bob*

```
Data: Alice_Heads : some nodes in the history of Alice, Ancestor : a dag of the ancesters of
      Alice, G_In : a graph of ancestors of Alice not known by Bob, Bf : The Bloom filter of
      Bob, Bd : The Border of Bob
Result: G_Out : a graph of ancestors of Alice not known by Bob, node_to_check : a list of
        ancestors of Alice that shall be revisited with the next Bloomfilters
explored = ∅
in_bf = ∅
in_border = ∅
to_further_explore = ∅
Procedure explore(node)
    if node ∉ explored then
        explored = node ∪ explored
        if belong(node,Bf) then
            in_bf = node ∪ in_bf
        else
            if belong(node,Bd) then
                in_border = node ∪ in_border
            else
                for pere ∈ predecessor(Ancestor,node) do
                    explore(pere)
                end
            end
        end
    end
Procedure find_in_bf(node)
    if node ∉ explored ∧ node ∉ G_In then
        add_vertex(G_In,node)
        for fils ∈ successor(Ancestor,node) do
            add_edge(G_In,node,fils)
            find_in_bf(fils)
        end
    end
Procedure find_in_border(node)
    if node ∉ explored then
        if node ∈ G_In then
            to_further_explore = node ∪ to_further_explore
        else
            for fils ∈ successor(Ancestor,node) do
                find_in_border(fils)
            end
        end
    end
for node ∈ Alice_Heads do
    explore(node)
end
explored = ∅
for node ∈ in_bf do
    find_in_bf(node)
end
explored = ∅
for node ∈ in_border do
    find_in_border(node)
end
return (G_In, to_further_explore)
```