Research Internship Report

Loric ANDRE

Study of the competitiveness of an algorithm for online list update with precedence constraints

Under the supervision of :
Maciej PACUT, *Universität Wien*
Arash POURDAMGHANI, *Universität Wien*
Bruno MONSUEZ, *ENSTA Paris*

Special Thanks

I want to warmly thank Maciej PACUT and Arash POURDAMGHANI for their guidance, help, patience and understanding during this internship. They have welcomed me with open arms and have supported me throughout the internship, and learning from them has been a pleasure.

I also want to thank Bruno MONSUEZ for his reactivity and efficiency, and Stefan Schmid for directing me to what has been an extremely interesting team and subject.

Abstract

The list update problem is a classic of computer science, where we have a list of items and we try to optimize the cost of accessing a sequence of these items, reorganizing (or not) the list in real time (online) or in advance (offline). We had results as early as 1976 [Rivest, 1976], and a deeper, more general and systematical study in Borodin and El-Yaniv [2005], proving competitiveness of many algorithms.

This problem has applications in various domains such as compression or caching, where fast access to items in linked structures is crucial.

We will here study the competitiveness of an algorithm in the context of the list update problem with *precedence constraints*. This means that we introduce the concept of *dependencies*, enforcing a partial order to the items in our list.

Contents

List of Figures

Chapter 1

Introduction

1.1 Context of the internship

This internship was an amazing immersion in the world of research at the University of Vienna. I was supervised by Stefan Schmid but most of my work was done in collaboration with Maciej Pacut and Arash Pourdamghani, through text messages most of the time and voice calls twice a week. This allowed them to help me as much as I needed and give me directions.

1.2 The list update problem

1.2.1 Basic context

The list update problem is as follows : We have a set of n items contained in a list L, indexed from 1 to n. We also have a *request sequence* $\sigma = (\sigma_1, \dots \sigma_m)$ consisting of accesses to items in the list L.

Example :
$L = [1, 2, 3, 4, 5]$
$\sigma = [3, 2, 4, 4, 1, 3, 4, 2, 1, 5]$

The goal of our research is to design and analyze algorithms that minimize the overall cost.

1.3 Cost

The total cost of running an algorithm ALG through a request sequence σ is denoted by $ALG(\sigma)$ and is the sum of the *access cost* and the *reconfiguration cost.*

1.3.1 Access Cost

The access cost of a request σ_j equals the number of comparisons needed to find σ_j in the list L. This means that the access cost of the item at position i in L is i. Note : Later in this report, we use the *Partial Cost* of an algorithm, $ALG^*(\sigma)$. In this cost model, the last comparison is free, which means that accessing the item at position i costs $i - 1$.

1.3.2 Reconfiguration Cost

In addition to access cost, we define the *reconfiguration cost*, the cost of moving items in our list. There are two models used to compute the reconfiguration cost:

Firstly, the *free exchange* model, in which the the accessed item can move (up to in front of the list) without any additional cost. As the name implies, these operations have no impact on the cost

Secondly, we consider the case that each swap of two items costs 1, the *paid exchange* model. A reconfiguration in paid exchange model can be moving the accessed item before accessing it, or moving items different from the one accessed.

1.4 Types of algorithms

The algorithms solutions to this problem are split into two categories : online and offline algorithms.

1.4.1 Offline Algorithms

An offline algorithm knows the entire sequence as it runs, while an online algorithm only knows the requests up until the current one.

1.4.2 Online Algorithms

Since the goal is to work on arbitrarily long request sequences and to deal with problems where we continuously treat requests as they come, we try to design the best possible online algorithms.

1.4.3 Optimal Algorithm

The optimal algorithm is a baseline we use to compute the efficiency of our algorithms. Reingold and Westbrook [1996] and Divakaran [2014] exposed optimal algorithms running in $\Theta(2^n(n-1)!l)$ time, where n is the length of the sequence and l is the number of nodes (the running time is not to be confused with the cost). This shows that we generally have no way to express this algorithm, which requires us to designing methods to get bounds on its cost.

1.5 Competitive ratio

To characterize the efficiency of an algorithm, we use the *competitive ratio*. We say that an algorithm ALG has a competitive ratio c if :

$$\exists a, \forall \sigma, ALG(\sigma) \leq c \cdot OPT(\sigma) + a$$

ALG is *strictly competitive* if $a = 0$.

1.6 Some Known Algorithms

We present a few classic algorithms

1.6.1 Move To Front

The algorithm Move To Front (MTF) is one of the simplest algorithms there are : It simply moves the accessed item to the head of the list, using free exchanges.

1.6.2 Transpose

The algorithm Transpose (TRANS) simply transposes the accessed item with the previous one in the list.

1.6.3 Frequency Count

We need to use counters for each item, increasing it by one after every access. We then move the item to keep the list sorted by frequency. This algorithm is obviously more complicated to implement and more costly (in terms of memory) to run.

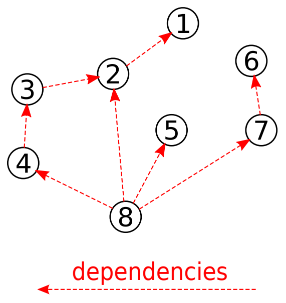
Chapter 2

Adding precedence constraints

In some cases, like firewall rules, some items depend on one another (for firewall rules, we often have a general rule over a subnet, then some more specific rules for some IPs or protocols, and the general rule needs to be applied before the specific ones). This requires us to introduce *precedence constraints*.

2.1 Model

Let G be a Directed Acyclic Graph with our items $1, ..n$ as nodes. We say that node i depends on node j if there is an edge between i and j.



N	Proto	Src IP	Dst IP	Src Port	Dst Port	Action
1	TCP	10.12.12.0/24	20.0.0.1/32	ANY	80	DENY
2	TCP	0.0.0.0/0	20.0.0.1/32	ANY	80	ACCEPT
3	IP	0.0.0.0/0	20.0.0.1/32			DENY
4	UDP	0.0.0.0/0	0.0.0.0/0	1000-2000	1000-2000	ACCEPT
5	UDP	20.0.0.0/24	10.0.10.0/24	ANY	3306	ACCEPT
6	TCP	10.12.12.0/24	0.0.0.0/0	21	21	DENY
7	IP	10.0.0.0/16	20.0.0.0/20			ACCEPT
8	IP	0.0.0.0/0	0.0.0.0/0			DENY

Figure 2.1: An example of a set of firewall rules and the associated dependency graph.
Credits : Pacut et al. [2021], page 4

This implies constraints on our list, mainly that any node **must be** before all of its dependencies, enforcing a partial order in our list.

2.2 Algorithm MRF

We need to adapt the list update algorithms to comply with the constraints. One of the (competitive) solutions found is the algorithm **Move Recursively Forward** :
Pacut et al. [2021] defines MRF as :
The algorithm uses a recursive procedure. The procedure MRF(y) moves the node y forward (by transposing it with the preceding nodes) until it encounters any of its dependency nodes, say z, and recursively calls MRF(z). Upon receiving an access request to a node σ_t, MRF locates σ_t on the list and invokes the procedure MRF(σ_t). We present the pseudocode of MRF in Algorithm 1. We say that a node y is a *direct dependency* of a node z if y is the dependency of z that is located at the furthest position on the list. By $\mathsf{pos}(z)$ we denote the position of node z in

the list maintained by the algorithm, counting from the head of the list (recall that the position of the first node is 1).

Algorithm 1: The algorithm MRF.

```
  Input: An access request to node $\sigma_t$
1 Access $\sigma_t$
2 Run the procedure MRF($\sigma_t$)

  Procedure MRF($y$):
3     if $y$ has no dependencies then
4         Move $y$ to the front of the list
5     else
6         Let $z$ be the direct dependency of $y$
7         Move node $y$ to $\mathsf{pos}(z) + 1$
8         Run the procedure MRF($z$)
```

Chapter 3

Stochastic Model

3.1 Model

Rivest [1976] introduces the stochastic model for computing competitive ratios in the list access problem. The basic premise of this model is that each item x is associated with a probability p_x. This allows us to use the average cost of each access instead of the total cost of the request sequence (which we assume is arbitrarily long in this model). This makes computing costs way easier using a probabilistic approach.

3.2 Competitive ratio for MTF

3.2.1 Cost for MTF

Rivest [1976] gives a proof for the competitive ratio of MTF in the stochastic setting by introducing $b(i, j)$, the probability that item i is in front of item j in our list :
Note : The law of large numbers allows us to get rid of the timing dependency and assume that all our expressions are time-independent.
Note 2 : This proof is given in the *Free exchange model*, in which all moves of the **currently accessed node** are **free**. In this model, in particular, the cost of MTF is exactly its access cost.

We have the average number of items in front of item j : $\sum_{i \neq j} b(i, j)$. Then, the access cost, which is the position of the item in the list, is:

$$pos(j) = 1 + \sum_{i \neq j} b(i, j)$$

We then simply need to compute $b(i, j)$.

By definition, $b(i, j)$ is the probability that item i is in front of item j in our list. We know that, according to the algorithm MTF, item i will be in front of item j if, and only if, item i was accessed after item j. This means that item i was accessed once at some point, and then neither i nor j was accessed afterwards. This gives us the following probability:

$$\begin{aligned} b(i, j) &= \sum_{k=1}^{\infty} p_i (1 - p_i - p_j)^{k-1} \\ &= \frac{p_i}{p_i + p_j} \end{aligned}$$

Putting this all together gives us the following cost for MTF:

$$\begin{aligned}
E_{MTF} &= \sum_{j=1}^{n} p_j pos(j) \\
&= \sum_{j=1}^{n} p_j (1 + \sum_{i \neq j} b(i,j)) \\
&= \sum_{j=1}^{n} p_j (1 + \sum_{i \neq j} \frac{p_i}{p_i + p_j}) \\
&= 1 + \sum_{j=1}^{n} p_j \cdot 2 \sum_{i=1}^{j-1} \frac{p_i}{p_i + p_j} \\
&\leq 1 + 2 \sum_{j=1}^{n} p_j (j-1)
\end{aligned} \tag{3.1}$$

3.2.2 Cost for OPT

Without any dependencies, the optimal algorithm creates a list in which items are ordered with decreasing probability. Therefore, the expected cost of OPT is:

$$E_{OPT} = \sum_{j=1}^{n} j p_j$$

3.2.3 Competitive ratio

Now that we have the expected cost for MTF and OPT, we can get the competitive ratio for MTF:

$$\begin{aligned}
R_{MTF} &= \frac{E_{MTF}}{E_{OPT}} \\
&\leq \frac{1 + 2\sum_{j=1}^{n} j p_j}{\sum_{j=1}^{n} j p_j} \\
&= \frac{2x - 1}{x} \\
&= 2 - \frac{1}{x}
\end{aligned} \tag{3.2}$$

Where $x = \sum_{j=1}^{n} j p_j$.
This gives us a competitive ratio of 2.

3.3 Competitive ratio for MRF

3.3.1 Preliminaries

We use a similar method for MRF, with a few changes. We are computing the cost of our algorithm in the *paid exchange model*, where all moves are costly. Of course, the dependencies also make this harder. We give all our results under Assumption 3.3.1 :

Assumption 3.3.1. *The DAG G is a forest.*

This assumption implies that all nodes depend on one other node at most.
We define the following notations:

– $MRF(\sigma)$, $MRF_A(\sigma)$ and $MRF_M(\sigma)$ are respectively the cost of MRF, the access cost of MRF and the cost of the reconfigurations in running MRF on the sequence σ.

– STATD (resp. STAT) is the optimal static algorithm with dependencies (resp. without).

– $b(i, j)$ is the asymptotic probability that i is before j in our list.

– γ_x is the sum of the probabilities for all the nodes depending on x (including x itself)

– $\mathcal{D}(x)$ are the nodes of the subtree of all nodes depending on x (the descendants of x).

– $\mathcal{A}(x)$ is the set of all the nodes x depends on (the ancestors of x and x).

– ψ_x is the number of nodes in x's complete dependency tree.

– δ_x is the number of nodes x depends on.

– $par(i)$ is the parent of i in its complete dependency tree. If i does not have a parent, we give him a 'virtual' one that we place at position 0 in our list.

– $par^{(m)}(i)$ is the m-th ancestor of i

– $\mathcal{R}$ is the set of all the roots in the forest.

Theorem 3.3.2. *If the input consist of independent and identically distributed random variables and the dependency graph G is a forest, then the expected cost for Move Recursively Forward cost is*

$$MRF(\sigma) \leq 2(1+\sum_{j=1}^{n} p_j(\delta_j + \sum_{i \notin \mathcal{A}(x) \cup \mathcal{D}(x)} \prod_{l \in \mathcal{A}(i)} \gamma_l \left(\frac{1}{\gamma_j + \gamma_l} + \frac{1}{\gamma_l(1-\gamma_j)}(1 + \delta_j - \frac{1-(\gamma_j)^{\delta_j+1}}{1-\gamma_j} \right))).$$

Lemma 3.3.3. *In the dependency graph is a forest, we have the following values for $b(i, j)$*

$$b(i,j) = \begin{cases} 1 & \textit{if } j \textit{ depends on } i \textit{ (1)} \\ 0 & \textit{if } i \textit{ depends on } j \textit{ (2)} \\ \prod_{l \in \mathcal{A}(i)} \gamma_l \left(\frac{1}{\gamma_j + \gamma_l} + \frac{1}{\gamma_l} \cdot \sum_{m=0}^{\delta_j} \sum_{k=1}^{m} (\gamma_j)^{k-1} \cdot b(l, par^m(j)) \right) & \textit{else (3)} \end{cases} \tag{3.3}$$

Proof. The dependency relation in G enforces order between nodes, thus the cases (1) and (2) follow. In the remaining of this proof we consider the case where the nodes are independent in G.

The node i can be before j in the list if all four of the following conditions are met:

(a) i itself or a node depending on it has been accessed

(b) i could be moved ahead of j

(c) j has not been moved ahead of i afterwards

(d) i was not moved afterwards

(a) happens with probability γ_i.
(b) is exactly b(par(i), j) : we can use the strong law of large numbers to get rid of the timing dependency in our case
(c) and (d) can be combined, forming the event : "neither i nor j have been moved at any point afterwards or i was not moved and j did not overtake i". The first part can be translated to :

$$\sum_{k=1}^{\infty} (1 - \gamma_j - \gamma_i)^{k-1} = \frac{1}{\gamma_j + \gamma_i}$$

"j did not overtake i" can be expressed as "j was not moved enough to overtake i". To overtake i, j would need to be moved m times, where m is the number of ancestors of j between i and j. This gives us the following probability for the whole second part :

$$\begin{aligned}&\sum_{k=1}^{\infty}(1-\gamma_i)^{k-1}\cdot\sum_{m=0}^{\delta_j}\sum_{k=1}^{m}(\gamma_j)^{k-1}\cdot b(i,par^m(j))\\&=\frac{1}{\gamma_i}\cdot\sum_{m=0}^{\delta_j}\sum_{k=1}^{m}(\gamma_j)^{k-1}\cdot b(i,par^m(j))\end{aligned}\tag{3.4}$$

This gives us the combined probability in case (3):

$$\begin{aligned}b(i,j)&=\gamma_i\cdot b(par(i),j)\cdot\left(\frac{1}{\gamma_j+\gamma_i}+\frac{1}{\gamma_i}\cdot\sum_{m=0}^{\delta_j}\sum_{k=1}^{m}(\gamma_j)^{k-1}\cdot b(i,par^m(j))\right)\\&=\prod_{l\in\mathcal{A}(i)}\gamma_l\left(\frac{1}{\gamma_j+\gamma_l}+\frac{1}{\gamma_l}\cdot\sum_{m=0}^{\delta_j}\sum_{k=1}^{m}(\gamma_j)^{k-1}\cdot b(l,par^m(j))\right)\end{aligned}\tag{3.5}$$

□

Proof of Theorem 3.3.2. The proof is similar in structure to Rivest [1976]. We first calculate the value of $b(i,j)$ in Lemma 3.3.3 and then use this value to get the access cost of any element of our list using the results from Rivest [1976]. From this, we use the result from Pacut et al. [2021] that the rearrangement cost is bound by the access cost and get the bound total expected cost for MRF.

From $b(i,j)$, we can get the expected position of node j :

$$\begin{aligned}E[pos(j)]&=1+\sum_{i\neq j}P(pos(i)<pos(j))\\&=1+\sum_{i\neq j}b(i,j)\end{aligned}\tag{3.6}$$

From this, the expected access cost for MRF $MRF_A(\sigma)$ comes naturally :

$$\begin{aligned}
MRF_A(\sigma) &= \sum_{j=1}^{n} p_j E[pos(j)] \\
&= \sum_{j=1}^{n} p_j (1 + \sum_{i\neq j} b(i,j)) \\
&= \sum_{j=1}^{n} p_j (1 + \sum_{l\in\mathcal{A}(j)-\{j\}} 1 + \sum_{i\notin\mathcal{A}(x)\cup\mathcal{D}(x)} b(i,j)) \text{ \textit{(see Lemma 3.3.3)}} \\
&= 1 + \sum_{j=1}^{n} p_j (\delta_j + \sum_{i\notin\mathcal{A}(x)\cup\mathcal{D}(x)} b(i,j)) \\
&= 1 + \sum_{j=1}^{n} p_j (\delta_j + \sum_{i\notin\mathcal{A}(x)\cup\mathcal{D}(x)} \prod_{l\in\mathcal{A}(i)} \gamma_l \left(\frac{1}{\gamma_j+\gamma_l} + \frac{1}{\gamma_l}\cdot\sum_{m=0}^{\delta_j}\sum_{k=1}^{m} (\gamma_j)^{k-1}\cdot b(l, par^m(j)) \right)) \\
&\leq 1 + \sum_{j=1}^{n} p_j (\delta_j + \sum_{i\notin\mathcal{A}(x)\cup\mathcal{D}(x)} \prod_{l\in\mathcal{A}(i)} \gamma_l \left(\frac{1}{\gamma_j+\gamma_l} + \frac{1}{\gamma_l}\cdot\sum_{m=0}^{\delta_j}\sum_{k=1}^{m} (\gamma_j)^{k-1} \right)) \\
&= 1 + \sum_{j=1}^{n} p_j (\delta_j + \sum_{i\notin\mathcal{A}(x)\cup\mathcal{D}(x)} \prod_{l\in\mathcal{A}(i)} \gamma_l \left(\frac{1}{\gamma_j+\gamma_l} + \frac{1}{\gamma_l}\cdot\sum_{m=0}^{\delta_j} \frac{1-(\gamma_j)^{k-1}}{1-\gamma_j} \right)) \\
&= 1 + \sum_{j=1}^{n} p_j (\delta_j + \sum_{i\notin\mathcal{A}(x)\cup\mathcal{D}(x)} \prod_{l\in\mathcal{A}(i)} \gamma_l \left(\frac{1}{\gamma_j+\gamma_l} + \frac{1}{\gamma_l(1-\gamma_j)}(1+\delta_j - \sum_{m=0}^{\delta_j} (\gamma_j)^{k-1} \right)) \\
&= 1 + \sum_{j=1}^{n} p_j (\delta_j + \sum_{i\notin\mathcal{A}(x)\cup\mathcal{D}(x)} \prod_{l\in\mathcal{A}(i)} \gamma_l \left(\frac{1}{\gamma_j+\gamma_l} + \frac{1}{\gamma_l(1-\gamma_j)}(1+\delta_j - \frac{1-(\gamma_j)^{\delta_j+1}}{1-\gamma_j} \right))
\end{aligned} \tag{3.7}$$

Pacut et al. [2021] proves that the rearrangement cost for MRF is bound by its access cost, hence the final bound. □

3.3.2 Bound on the cost for MRF

Theorem 3.3.4. *In the case of dependency trees, we have the following bound on the cost of MRF :*

$$MRF(\sigma) \leq 2\left(\sum_{j=1}^{n} p_j(n - |\mathcal{D}(j)|) - \sum_{j\in\mathcal{R}} p_j \sum_{i\notin\mathcal{A}(x)\cup\mathcal{D}(x)} \left(\frac{\gamma_j}{\gamma_j+\gamma_i}\right) \right)$$

Lemma 3.3.5. *In the case of dependency trees, $b(i,j)$ can be bounded by:*

$$b(i,j) \leq \begin{cases} \frac{\gamma_i}{\gamma_i+\gamma_j} & \textit{if } j \textit{ is a root and if } i \textit{ does not depend on } j \\ 0 & \textit{if } j \textit{ depends on } i \\ 1 & \textit{if } i \textit{ depends on } j \textit{ or if } j \textit{ is not a root} \end{cases}$$

Proof of Lemma 3.3.5. We know that $b(i,j)$ is bounded by 1. We can then replace in case 3 of Lemma 3.3.3:

$$b(i,j) \leq \frac{\gamma_i}{\gamma_i+\gamma_j} + \sum_{m=0}^{\delta_j}\sum_{k=1}^{m} \gamma_j^{k-1} \tag{3.8}$$

The second term is greater than 1 if δ_j is positive. Hence the bound. □

Proof of Theorem 3.3.4. By using the bound for the cost, we get :

$$\begin{aligned}
MRF_A(\sigma) &= 1 + \sum_{j=1}^{n} p_j \sum_{i \neq j} b(i,j) \\
&\leq 1 + \sum_{j=1}^{n} p_j \sum_{i \in \mathcal{A}(j)-\{j\}} 1 + \sum_{j \in \mathcal{R}} p_j \sum_{i \notin \mathcal{A}(x) \cup \mathcal{D}(x)} \frac{\gamma_i}{\gamma_i + \gamma_j} + \sum_{j \notin \mathcal{R}} p_j \sum_{i \notin \mathcal{D}(j) \bigcup \mathcal{A}(j)} 1 \\
&= 1 + \sum_{j=1}^{n} p_j \delta_j + \sum_{j \in \mathcal{R}} p_j \sum_{i \notin \mathcal{A}(x) \cup \mathcal{D}(x)} \frac{\gamma_i}{\gamma_i + \gamma_j} + \sum_{j \notin \mathcal{R}} p_j (n - |\mathcal{D}(j)| - \delta_j - 1) \\
&= 1 + \sum_{j=1}^{n} p_j \delta_j + \sum_{j \in \mathcal{R}} p_j \sum_{i \notin \mathcal{A}(x) \cup \mathcal{D}(x)} (1 - \frac{\gamma_j}{\gamma_i + \gamma_j}) + \sum_{j \notin \mathcal{R}} p_j (n - |\mathcal{D}(j)| - \delta_j - 1) \\
&= 1 + \sum_{j=1}^{n} p_j \delta_j + \sum_{j \in \mathcal{R}} p_j (n - |\mathcal{D}(j)| - 1) + \sum_{j \notin \mathcal{R}} p_j (n - |\mathcal{D}(j)| - \delta_j - 1) - \sum_{j \in \mathcal{R}} p_j \sum_{i \notin \mathcal{A}(x) \cup \mathcal{D}(x)} (\frac{\gamma_j}{\gamma_j + \gamma_i}) \\
&= 1 + \sum_{j=1}^{n} p_j \delta_j + \sum_{j \in \mathcal{R}} p_j (n - |\mathcal{D}(j)| - \delta_j - 1) - \sum_{j \in \mathcal{R}} p_j \sum_{i \notin \mathcal{A}(x) \cup \mathcal{D}(x)} (\frac{\gamma_j}{\gamma_j + \gamma_i}) \\
&= 1 + \sum_{j=1}^{n} p_j (n - |\mathcal{D}(j)| - 1) - \sum_{j \in \mathcal{R}} p_j \sum_{i \notin \mathcal{A}(x) \cup \mathcal{D}(x)} (\frac{\gamma_j}{\gamma_j + \gamma_i})
\end{aligned} \tag{3.9}$$

□

3.3.3 Bound for the cost of STATD

Theorem 3.3.6. *For dependency trees, we have the following lower bound for the cost of the static optimal algorithm:*

$$E_{STATD} \geq 1 + max(\sum_{j=1}^{n} (j-1) p_j, \sum_{j=1}^{n} \delta_j p_j)$$

Proof of Theorem 3.3.6. Each node is behind all the nodes it depends on. This means that $pos(x) \geq \delta_x + 1$. Then, we have $E_{STATD} \geq 1 + \sum_{j=1}^{n} \delta_j p_j$ on average. Furthermore, we also know that this algorithm is more expensive than the optimal static algorithm without dependencies, hence the final bound. □

3.3.4 Competitive ratio

We can see that the cost of MRF, even after as many simplications as possible, is still extremely complicated.
A few directions were explored on simplifying it, but the bounds always became loose in some cases, preventing us from getting a good bound (Pacut et al. [2021] proved that a bound of 4 exists).

3.3.5 Conclusion

This did not yield results, but the code in 6.1.2 seemed to indicate an upper bound of π in the stochastic setting, which would be consistent with Chung et al. [1988]'s Theorem 1, stating a result of $\frac{\pi}{2}$ for MRF in this setting.
The major issues we had at the end of this research were, first, the complexity of our equations, and secondly the bound on STATD. This is the bound that we could not optimize for all request sequences and this is the part that needs more time or a different way to dela with.

Chapter 4

Non-stochastic bound

As mentioned, Pacut et al. [2021] proved a ratio of 4 for MRF in the paid exchange model. However, that proof is fairly long and complicated, and not easily applicable to other algorithms. We present here a more replicable way to prove the same ratio of 4, based on Borodin and El-Yaniv [2005].

4.1 Preliminaries

Similarily to Borodin's book, we define the following notations :

- $B(x, j)$ is equal to one if node x is in front of node σ_j (at the time of accessing j), zero otherwise
- $pos_t(x)$ is the position of node x in our list at time t.
- σ_{xy} is the request sequence stripped of all requests to nodes different from x and y.
- $a(x)$ is the number of ancestors of node x
- $occ(x, \sigma)$ is the number of occurences of node x in the request sequence σ

4.2 List Factoring

4.2.1 Original List Factoring Technique

The objective of the list factoring technique for competitive analysis, as defined in Borodin and El-Yaniv [2005] in section 1.6, is to simplify the problem and only deal with **pairs of items**. Most algorithms then become very simple to characterize and the costs are easy to compute.

To then come back to the actual cost of our algorithm, we need a property called the *pairwise property*. This is satisfied if, at all times, the relative positions of nodes x and y in the list generated by running the algorithm on the complete sequence is the same as their position in the list generated by running the algorithm on the striiped sequence, with only accesses to x and y.

4.2.2 In our Case

We cannot use the list factoring technique as-is because MRF obviously does not respect the Pairwise property lemma (take the case where x and y depend on one another, for example). This means that we need to develop an alternative way to use the list factoring technique, that will take the dependency relationships into account.

Lemma 4.2.1. *ALG satisfies the weak pairwise property if and only if :*

$$\forall x \neq y, ALG_{xy}(\sigma) \leq ALG(\sigma_{xy})$$

Where

$$ALG_{xy}(\sigma) = \sum_{j, \sigma_j \in \{x,y\}} [B(x,j) + B(y,j)]$$

Lemma 4.2.2. *If ALG satisfies the weak pairwise property, then the ratio over the stripped sequences holds over the full sequence. This means the following :*

$$ALG(\sigma_{xy}) \leq c \cdot OPT(\sigma_{xy}) \implies \sum_{x \neq y} ALG_{xy}(\sigma) \leq c \cdot \sum_{x \neq y} OPT_{xy}(\sigma)$$

Proof. Borodin and El-Yaniv [2005] showed this result in the case without precedence constraints. His results can be adapted to the case of an algorithm satisfying the weak pairwise property :
First, we want to prove the following : $OPT(\sigma_{xy}) \leq OPT_{A,xy}(\sigma) + OPT_{M,xy}(\sigma)$.
To get this result, we notice that the left hand side of the inequality is the cost of an algorithm acting on x and y, which means that its cost is higher than that of OPT. We also know that $OPT(\sigma) = \sum_{x \neq y}(OPT_{A,xy} + OPT_{M,xy})$, which means that the ratio on the stripped sequences holds over the ratio for the pairs

$$\begin{aligned} & ALG(\sigma_{xy}) \leq c \cdot OPT(\sigma_{xy}) \\ \implies & ALG_{xy}(\sigma) \leq c \cdot OPT_{xy}(\sigma) \end{aligned} \tag{4.1}$$

This gives us, after summing :

$$\begin{aligned} & ALG(\sigma_{xy}) \leq c \cdot OPT(\sigma_{xy}) \\ \implies & \sum_{x \neq y} ALG_{xy}(\sigma) \leq c \sum_{x \neq y} \cdot OPT_{xy}(\sigma) \end{aligned} \tag{4.2}$$

□

4.3 Case of MRF

Theorem 4.3.1 (Cost of MRF split by pairs of nodes)**.** *The cost of MRF can be expressed as :*

$$MRF(\sigma) = 2 \sum_{x \neq y} MRF_{xy}(\sigma) - \sum_{j=1}^{n} a(\sigma_j)$$

Proof. We can notice that the access cost at iteration j is $pos_(\sigma_j)$. Also, the exchanges at that time occur when we move the node and its parents forward. This gives us a cost for the reconfiguration at iteration j of $pos_j(\sigma_j) - a(\sigma_j)$.
Then, the total cost can be expressed as :

$$\begin{aligned} MRF(\sigma) &= MRF_A(\sigma) + MRF_M(\sigma) \\ &= \sum_{j=1}^{n} (2pos_j(\sigma_j) - a(\sigma_j)) \\ &= \sum_{j=1}^{n} \left(2 \sum_{x \in L} B(x,j) \right) - \sum_{j=1}^{n} a(\sigma_j) \\ &= \sum_{x \in L} \sum_{y \in L} 2 \sum_{j, \sigma_j = y} B(x,j) - \sum_{j=1}^{n} a(\sigma_j) \\ &= \sum_{x,y \in L} 2 \sum_{j : \sigma_j \in \{x,y\}} B(x,j) - \sum_{j=1}^{n} a(\sigma_j) \end{aligned} \tag{4.3}$$

□

Theorem 4.3.2 (Weak pairwise property for MRF)**.** *MRF satisfies the weak pairwise property.*

This assures the following property for MRF :

$$MRF(\sigma) \leq 2 \sum_{x \neq y} MRF(\sigma_{xy})$$

Proof. $MRF(\sigma_{xy})$ is the cost of running the algorithm on the stripped sequence. This will be, as shown by Theorem 4.3.1, equal to :

$$MRF(\sigma_{xy}) = 2MRF_{xy}(\sigma) - a(x)occ(x,\sigma) - a(y)occ(y,\sigma)$$

Furthermore, $a(x)$ is the number of ancestors of x, which means that $pos(x) > a(x)$ at all times. Then, $a(x)occ(x,\sigma) + a(y)occ(y,\sigma)$ is less than even the optimal cost of any algorithm for treating the requests to x and y in σ. This means that, in particular, it is less than $MRF_{xy}(\sigma)$.
Then, we can write :

$$MRF(\sigma_{xy}) \geq MRF_{xy}(\sigma)$$

Hence the result, by replacing in Theorem 4.3.1. □

Theorem 4.3.3 (Competitive ratio for MRF)**.** *MRF is 4-competitive in the paid exchange model.*

Proof. For an unrelated pair of nodes, serving a request can cost these values for MRF and OPT:

	Front node	Back node
MRF	1	3
OPT	1 or 2	2 or 3

On such a pair, let i, j be two indexes of requests in σ_{xy} such as these requests cost 3 to treat with MRF, and there are no requests with cost 3 in between, with $i < j$.

Let's also assume that x is in front before treating request i. Then, MRF will move y in front, costing 3 (2 for access, one for the move). All following requests then cost 1 until j, which costs 3 again. This gives us a total cost of $3 + 3 + (j - i - 1) = 5 + j - i$ for MRF.

While treating this request, the nodes are both accessed by OPT, meaning that at some point, it will cost 2 to treat a request. This gives us a lower bound on the cost for OPT of $2 + j - i$.

Thus, we have the following competitive ratio for MRF over two unrelated nodes :

$$\frac{MRF(\sigma_{xy})}{OPT(\sigma_{xy})} \leq \frac{5 + j - i}{2 + j - i}$$

What's more, we know that $i < j$, and this means that $j - i \geq 1$.
Then, we have the following ratio :

$$\frac{MRF(\sigma_{xy})}{OPT(\sigma_{xy})} \leq \frac{5 + 1}{2 + 1} = 2$$

For related nodes, the reasoning is simple : the list is fixed, which means that both MRF and OPT have the same cost over these nodes, and the ratio for these nodes is one.

This gives us a ratio of 2 for the stripped sequences. In the case of MRF, we know that $MRF(\sigma) \leq 2 \sum_{x \neq y} MRF(\sigma_{xy})$, hence the result. □

Note : The full ratio of $2 \cdot \frac{5+j-i}{2+j-i}$ decreases with locality, down to two as the number of requests to the same node between indexes i and j increases.

4.4 Conclusion

We have shown that, as long as an algorithm satisfies the waek pairwise property, we can easily switch from the competitive ratio over pairs of nodes to the full competitive ratio.

This is, as announced a repeatable way to compute the competitive ratio of algorithms in the paid exchange model even for algorithms in the precedence constraint setting.

Note : For MRF itself, the ratio could maybe be improved by reasoning differently in the cases where $j - i$ is greater than 1 or not. This was pointed ouut by Arash at the end of the internship and we were not able to dedicate it the time it needed unfortunately.

Chapter 5

Locality

5.1 Preliminaries

This work is based on Albers and Lauer [2016]. We define the following notations :

- $ALG_M(t, \sigma)$ is the cost of moving the items at time t for algorithm ALG
- $ALG_{M,xy}(\sigma)$ is the cost of the moves involving both x and y for algorithm ALG
- $l(\sigma_{xy})$ is the number of long runs in the stripped sequence
- $l_c(\sigma_{xy})$ is the number of long run changes in sequence σ_{xy}
- $r(\sigma_{xy})$ is the number of runs in $\sigma_x y$
- $f_e(\sigma_{xy})$ is equal to one if and only if the sequence ends with a long run change.
- $f_b(\sigma_{xy})$ is equal to one if and only if the first requested item is in front in the list at that time.
- for all these notations $s(\sigma_{xy})$, $s(\sigma)$ is the sum of $s(\sigma_{xy})$ over all pairs of nodes.
- A subscript U means that it is the sum of all the considered objects over unrelated nodes, and a subscript R will be sum sum over the related nodes. Also, |U| is the number of pairs of unrelated nodes and |R| is the number of pairs of related nodes.
- $\eta(\sigma)$, where x and y are related and x is an ancestor of y is equal to the sum of $\frac{occ(y,\sigma)}{|\sigma|}$ over all pairs of related nodes.

5.2 A definition of locality

Locality is a way to characterize the way nodes are accessed in the list update problem. We say that a sequence has *high locality* if an accessed node has higher chances to be accessed again. This has been expressed by Albers and Lauer [2016] using a number called λ. This is a parameter that grows closer to 1 with high locality. Its definition in Albers and Lauer [2016] is as follows : A class Σ of requests satisfies λ-locality if for all requests $\sigma \in \Sigma$,

$$\lambda = \frac{l_c(\sigma)}{r(\sigma)}$$

This means that this parameter grows closer to one as the proportion of long runs increases, and gets down to 0 as it decreases.

As we do with other notations, we adapt this to our model of precedence constraints, giving us the parameter λ_U : A class Σ of requests satisfies λ-locality if for all requests $\sigma \in \Sigma$,

$$\lambda = \frac{l_c(\sigma)}{r(\sigma)}$$

Locality itself is often observed, again in the case of firewall rules, a user using an API or browsing a website will often be requesting content from the same server multiple times in a row (for a website, the user need the basic page as well as all the images, scripts and stylesheets).

5.3 A new way to write the cost

We write the cost of any algorithm ALG as :

$$\begin{aligned} ALG(\sigma) &= \sum_{t=1}^{n} \left(\sum_{x \in L} B(x,t) + ALG_M(t,\sigma) \right) + n \\ &= \sum_{x \neq y} \sum_{t:\sigma_t \in \{x,y\}} B(x,t) + \sum_{t=1}^{n} ALG_M(t,\sigma) + n \\ &= \sum_{x \neq y} ALG^*_{xy}(\sigma) + |\sigma| \end{aligned} \tag{5.1}$$

Where $ALG^*_{xy}(\sigma) = \sum_{t:\sigma_t \in \{x,y\}} (B(x,t) + B(y,t)) + ALG_{M,xy}(\sigma)$

5.4 Analysis of the phases

We split σ_{xy} into phases $\pi(1), ..\pi(p_{xy})$ ending with either a long run or the last request of σ_{xy} if it is a short run. If $\pi(i)$ starts with x, it has one of the 2 following forms :

(a) $(xy)^k x^l$, $k \geq 0$, $l \geq 1$

(b) $(xy)^k y^l$, $k \geq 1$, $l \geq 0$

The phases starting with y can be obtained by exchanging the roles of x and y. We will study phases starting with x from now on.

Theorem 5.4.1. *The full cost of OPT over σ is :*

$$\begin{aligned} OPT(\sigma) &\geq \sum_{x \neq y} OPT^*(\sigma_{xy}) + |\sigma| \\ &\geq \frac{r_U(\sigma) + 3l_{c,U}(\sigma) - 2f_{e,U}(\sigma) - f_{b,U}(\sigma)}{2} + OPT^*_R(\sigma) + |\sigma| \end{aligned} \tag{5.2}$$

Proof. An optimal algorithm over two nodes is easy to state :

– If the two nodes are related, the list is fixed and the algorithm doesn't do anything

– If not, the algorithm will move the requested node at the beginning of every long run

An intuitive proof is as follows :

– Accessing the back node twice in a row costs $2 + 2 = 4$, and each subsequent access costs 2 too.

– Accessing the back node and moving it to the front costs $2 + 1 + 1 = 4$, and each subsequent access costs 1.

This means that if we don't move nodes, a long run of length k to the back node costs $2k$ while the same long run costs $3 + k$ if we move the node.

Given that the previous phase ended with a long run, and that the current one starts with x, we know that the previous phase ended with a long run to y. This assures us that at the beginning of phase $\pi(i), i \neq 1$, y is in front of the list generated by our OPT.

We will deal with phases that are neither the first nor the last one. This assures us two things :

– y is in front at the beginning of the phase

– the phase ends with either a long run of x or y.

Then, the cost of sequence (b) is easy to determine, since no node will be moved. Since we are using the partial cost model, each access to y is free and each access to x costs 1. We have $2k$ runs and k accesses to x. Hence a cost of $\frac{r(\pi(i))}{2}$.

For (a), we have $2k + 1$ runs, $k + 1$ requests to x and 1 move, which gives us a cost of $\frac{r(\pi(i))+1}{2} + 1$.

This gives us the followinng cost for these phases :

$$\forall i \in [2, p_{xy} - 1], OPT^*(\pi(i)) = \frac{r(\pi(i)) + 3l_c(\pi(i))}{2}$$

For the last run, the only difference will be that we don't move the node in sequence (a) if it doesn't end in a long run change. Then, we get a cost of :

$$OPT^*(\pi(p_{xy})) = \frac{r(\pi(p_{xy})) + 3l_c(\pi(p_{xy}))}{2} - f_e(\sigma_{xy})$$

For the first one, the cost depends on which node is in front of the list at the beginning. If y is in front, the cost is the same as for all phases but the last.
If x is in front, then (a) costs $k = \frac{r(\pi(1))-1}{2} = \frac{r(\pi(1))+3l_c(\pi(1))}{2} - \frac{1}{2}$. (b) costs $k+2 = \frac{r(\pi(1))+3l_c(\pi(1))}{2} + \frac{1}{2}$.
This means that, in this case,

$$OPT^*(\pi(1)) \geq \frac{r(\pi(1)) + 3l_c(\pi(1)) - f_b(\sigma_{xy})}{2}$$

This gives us, for unrelated nodes :

$$OPT^*(\sigma_{xy}) \geq \frac{r(\sigma_{xy}) + 3l_c(\sigma_{xy}) - 2f_e(\sigma_{xy}) - f_b(\sigma_{xy})}{2}$$

□

Theorem 5.4.2. *The full cost of MRF is :*

$$\begin{aligned} MRF(\sigma) &\leq MRF_U^*(\sigma) + MRF_R^*(\sigma) + |\sigma| \\ &\leq 2r_U(\sigma) - 2f_{b,U}(\sigma) + OPT_R^*(\sigma) + |\sigma| \end{aligned} \tag{5.3}$$

Proof. The cost of MRF for pairs of unrelated nodes is easy to compute : Every run other than the first one costs 2 (since we access the back node and move it to the front, then we don't pay for the rest of the run). The first run will cost 2 if the first accessed node is in the back and 0 otherwise. This gives us the following cost for a pair of unrelated nodes :

$$MRF_U^*(\sigma_{xy}) = 2r(\sigma_{xy}) - 2f_b(\sigma_{xy})$$

Furthermore, the cost of running the algorithm on pairs of related nodes is thee same as for OPT:

$$OPT_R^*(\sigma) = MRF_R^*(\sigma)$$

□

Theorem 5.4.3. *Cost over related nodes The partial cost of these algorithms over the related nodes is :*

$$OPT^*_R(\sigma) = \eta(\sigma)|\sigma|$$

Proof. The partial cost over any pair is equal to the number of accesses to the back node. Since the ancestor node is always in front in this case, the result simply comes from summing over all pairs. □

5.5 Strict competitive ratio

Theorem 5.5.1. *MRF is strictly 4-competitive and the ratio goes as low as 2 as locality over unrelated nodes increases :*

$$\frac{MRF(\sigma)}{OPT(\sigma)} \leq \frac{4}{1+\lambda_U}$$

Proof. Let's introduce the following notations :

- $\alpha(\sigma) = \frac{(1+\eta(\sigma))|\sigma| - f_{b,U}}{r_U(\sigma)}$
- $\beta(\sigma) = \frac{3l_{c,U}(\sigma) - 2f_{e,U}(\sigma)}{r_U(\sigma)}$

This gives us the following competitve ratio for MRF :

$$\begin{aligned}
\mathcal{R}(\sigma) &= \frac{MRF(\sigma)}{OPT(\sigma)} \\
&\leq \frac{MRF^*_U(\sigma) + \eta(\sigma)|\sigma| + |\sigma|}{OPT^*_U(\sigma) + \eta(\sigma)|\sigma| + |\sigma|} \\
&\leq 4\frac{r_U(\sigma) - f_{b,U}(\sigma) + (\eta(\sigma)+1)|\sigma|}{r_U(\sigma) + 3l_{c,U}(\sigma) - 2f_{e,U}(\sigma) - f_{b,U}(\sigma) + 2(1+\eta(\sigma))|\sigma|} \\
&\leq 4\frac{1+\alpha(\sigma)}{1+2\alpha(\sigma)+\beta(\sigma)} \\
&\leq \frac{4}{1+\frac{\beta(\sigma)+\alpha(\sigma)}{1+\alpha(\sigma)}}
\end{aligned} \tag{5.4}$$

To get our result, we now study two cases :

- If $\alpha < 0$, then $\frac{\beta(\sigma)+\alpha(\sigma)}{1+\alpha(\sigma)} \geq \frac{l_{c,U}(\sigma)}{r_U(\sigma)}$ and we get the result from the last line
- If $\alpha \geq 0$, the result comes directly from the second to last line

□

5.6 Competitive ratio

Theorem 5.6.1. *MRF is* $\frac{4}{1+3\lambda_U}$ *competitive.*

Proof. We introduce the following notations :

- $\alpha'(\sigma) = \frac{(1+\eta(\sigma))|\sigma| - f_{b,U}(\sigma) - f_{e,U}(\sigma)}{r_U(\sigma)}$
- $\beta'(\sigma) = \frac{3l_{c,U}(\sigma)}{r_U(\sigma)}$

We can write :

$$
\begin{aligned}
MRF(\sigma) &\leq 4OPT(\sigma)\frac{1+\alpha(\sigma)}{1+2\alpha(\sigma)+\beta(\sigma)} \\
&\leq 4OPT(\sigma)\left(\frac{1+\alpha(\sigma)-2f_{e,U}(\sigma)/r_U(\sigma)}{1+2\alpha(\sigma)+\beta(\sigma)}+\frac{2f_{e,U}(\sigma)/r_U(\sigma)}{1+2\alpha(\sigma)+\beta(\sigma)}\right) \\
&\leq 4OPT(\sigma)\left(\frac{1+\alpha'(\sigma)}{1+2\alpha'(\sigma)+\beta'(\sigma)}+\frac{2f_{e,U}(\sigma)/r_U(\sigma)}{1+2\alpha'(\sigma)+\beta'(\sigma)}\right) \\
&\leq 4OPT(\sigma)\left(\frac{1}{1+\frac{\beta'(\sigma)+\alpha'(\sigma)}{1+\alpha'(\sigma)}}+\frac{2f_{e,U}(\sigma)/r_U(\sigma)}{1+2\alpha'(\sigma)+\beta'(\sigma)}\right) \\
&\leq \frac{4}{1+3\lambda_U}OPT(\sigma)+4\frac{2f_{e,U}(\sigma)OPT(\sigma)/r_U(\sigma)}{1+2\alpha(\sigma)+\beta(\sigma)}
\end{aligned}
\tag{5.5}
$$

Furthermore, the proof for the cost of OPT also shows that (by looking closely at the proof for the first run of the phase) :

$$
OPT(\sigma) \leq \frac{r_U(\sigma)+3l_{c,U}(\sigma)-2f_{e,U}(\sigma)+f_{b,U}(\sigma)+2(1+\eta(\sigma))|\sigma|}{2}
$$

This means that :

$$
\frac{OPT(\sigma)}{r_U(\sigma)} \leq \frac{1+2\alpha(\sigma)+\beta(\sigma)+2f_{b,U}(\sigma)}{2}
$$

Adding that into Equation 5.5 shows that :

$$
\begin{aligned}
MRF(\sigma) &\leq \frac{4}{1+3\lambda_U}OPT(\sigma)+4\frac{2f_{e,U}(\sigma)OPT(\sigma)/r_U(\sigma)}{1+2\alpha(\sigma)+\beta(\sigma)} \\
&\leq \frac{4}{1+3\lambda_U}OPT(\sigma)+4f_{e,U}(\sigma)\frac{1+2\alpha(\sigma)+\beta(\sigma)+2f_{b,U}}{1+2\alpha(\sigma)+\beta(\sigma)} \\
&\leq \frac{4}{1+3\lambda_U}OPT(\sigma)+4f_{e,U}(\sigma)\left(1+\frac{2f_{b,U}}{1+2\alpha(\sigma)+\beta(\sigma)}\right) \\
&\leq \frac{4}{1+3\lambda_U}OPT(\sigma)+4f_{e,U}(\sigma)\left(1+2f_{b,U}\right)
\end{aligned}
\tag{5.6}
$$

Both $f_{b,U}(\sigma)$ and $f_{e,U}(\sigma)$ are depending only on the number of nodes, and quadratically. This means that : $MRF(\sigma) \leq \frac{4}{1+3\lambda_U} + O(l^4)$ □

5.7 Conclusion

Once again, we showed a competitive ratio of 4. However, this is more specific than the previous results, since this ratio now gets better as λ_U and locality grow.
We showed a competitive ratio going down to the ideal ratio of one and a strict ratio going down to two. As the goal of these algorithms is to work indefinitely on a fixed set of nodes (take firewall rules for example) l quickly gets very small compared to the current length of the sequence, hence the usefulness of a non-strict ratio.

Chapter 6

Conclusion

We have worked to show a competitive ratio for MRF 3 different ways.
The first way involved working in the case where the accesses follow a distribution but did not yield definite results as the calculations quickly grew complicated and we did not have a good enough bound on the cost of the optimal algorithm. However, implementing our equations seemed to suggest that a bound of π might be reachable, similarily to the bound of $\frac{\pi}{2}$ from Chung et al. [1988].
The second way allowed us to design a systematic way to prove bounds for algorithms in the paid exchange model with precedence constraints using the weak pairwise property. This should prove useful as it is linked to other research in the department.
The last way was the actual goal of the internship, working with locality of reference. In this case, we were able to show the ratio of 4, but with a twist as we have a strict ratio going down to 2 and a non-strict one going down to the ideal ratio of one.

Appendix

6.1 Code

6.1.1 Optimal algorithm in the stochastic setting

```python
#!/usr/bin/env python3

from ratio_MRF import * # Nodes, generators etc.

# We have the following relation :
#    E_{STATD}^0 = 0
#    E_{STATD}^{n + 1} = E_{STATD}^n + p_{n+1} pos(n+1) + sum_{i behind n+1 in
    the new list} p_i
# We then simply need to calculate the new value for all possible positions of
    n+1 and take the minimum

def next_iter(g, prev, new_node, prev_cost):
    # prev needs to be a list of (index, probability) tuples
    # new_node should be a node
    n = len(prev)
    # get partial costs
    part_cost = [0] * (n + 1)
    for i in range(n-1, -1, -1):
        part_cost[i] = part_cost[i+1] + (i + 1) * prev[i].p


    next = []
    for i in range(n + 1):
        next.append(prev.copy())
    next_cost = [-1] * (n + 1)
    # We need to find the parent of our nodes and the list of nodes in its tree
    # locate the node in the tree
    t = 0
    for u in g:
        if new_node.i in u:
            t = u
            break

    # Get children and porent
    try:
        par = t.lambda_l(new_node.i)[1]
        found_parent = False
    except IndexError:
        par = 0
        found_parent = True
    children = new_node.children

    i_par = -1
    i_child = -1

    for i in range(n + 1):
        if i == n:
            next[n].append(new_node)
            next_cost[n] = prev_cost + new_node.p * (n + 1)
        elif prev[i].i == par:
            found_parent = True
            i_par = i
        elif found_parent: # We insert before the i-th node or after the last one
            if prev[i].i in children:
                i_child = i
                break
            else:
                next[i].insert(i, new_node)
                next_cost[i] = prev_cost + part_cost[i] + new_node.p * (i + 1)
    if i_child < 0:
        return next[i_par+1:], next_cost[i_par+1:]
    return next[i_par+1:i_child], next_cost[i_par+1:i_child]

def main(g, n):
    nodes = flatten([t.nodes() for t in g])
    print("Graph: ")
    for t in g:
        print(t)
    curr = [[]]
    curr_cost = [0]
    for node in nodes:
        all_next = []
        all_cost = []
        for i in range(len(curr)):
            next, next_cost = next_iter(g, curr[i], node, curr_cost[i])
            all_next += next
            all_cost += next_cost
        curr = all_next
        curr_cost = all_cost

    m = -1
    mi = 0
    for i in range(len(curr_cost)):
        c = curr_cost[i]
        if c > 0 and (m < 0 or c < m):
            m = c
            mi = i
    return m, curr[mi]

def delta_bound(g):
    bound = 0
    max_bound = 0
    for t in g:
        for node in t.nodes():
            bound += node.p * t.delta(node.i)
            max_bound += node.p * max(t.delta(node.i) + 1, node.i)
    return bound + 1, max_bound


if __name__ == '__main__':
    if len(sys.argv) != 3:
        print("Usage: ./statd.py <number of nodes> <sample size>")
        exit()
    n = int(sys.argv[1])
    N = int(sys.argv[2])
    avg = 0
    mini = -1
    maxi = -1
    for _ in range(N):
        g, p = gen(n)
        c, l = main(g, n)
        rivest = sum([(i + 1)*p[i] for i in range(n)])
        print("Cost: %s\nList: %s\nRivest: %s\nDeltas: %s" % (c, l, rivest,
    delta_bound(g)))
        if mini < 0:
            mini = c
        mini = min(c, mini)
        maxi = max(maxi, c)
        avg += c
    avg /= N
    print("avg: %s, min: %s, max: %s" % (avg, mini, maxi))
```

6.1.2 Code for MRF in the stochastic setting

This code has been modified often to test different formulas. It is parallelized to be able to run as fast as possible, since simulations without it can take hours with 100 to 1000 nodes.

```python
#!/usr/bin/env python3
import numpy as np
import sys
import matplotlib.pyplot as plt
from joblib import Parallel, delayed
from tqdm import tqdm

def flatten(l):
    res = []
    for x in l:
        try:
            iter(x)
            res += x
        except:
            res += [x]
    return res


class DepNode:
    def __init__(self, prob, i):
        self.i = i
        self.p = prob
        self.children = []
        self.n = 1

    def gamma(self):
        return self.p + sum([a.gamma() for a in self.children])

    def lambda_l(self, i):

        def aux(t, i):
            if t.i == i:
                return [t.i]
            elif t.is_leaf():
                return False
            else:
                for x in t.children:
                    rest = aux(x, i)
                    if not rest:
                        continue
                    else:
                        return rest + [t.i]
                return []

        return aux(self, i)

    def anc(self, i):

        def aux(t, i):
            if t == i:
                return [t]
            elif t.is_leaf():
                return False
            else:
                for x in t.children:
                    rest = aux(x, i)
                    if not rest:
                        continue
                    else:
                        return rest + [t]
                return []

        return aux(self, i)

    def delta(self, i):
        if self.i == i or self.is_leaf():
            return 0
        else:
            return 1 + max([c.delta(i) for c in self.children])

    def l(self):
        # get list of nodes in tree
        return [self.i] + flatten([a.l() for a in self.children])

    def nodes(self):
        return [self] + flatten([a.nodes() for a in self.children])


    def add_child(self, node):
        self.children.append(node)
        self.n += 1

    def insert(self, node, i):
        # Insert node as child of node i
        def aux(t):
            if t.i == i:
                t.children.append(node)
            else:
                [aux(subt) for subt in t.children]
        aux(self)
        self.n += 1

    def subtree(self, i):
        if self.i == i:
            return self
        elif self.is_leaf():
            return False
        else:
            for child in self.children:
                sub = child.subtree(i)
                if not sub:
                    continue
                else:
                    return sub

    def is_leaf(self):
        return self.children == [] or self.children[0] is None

```

```python
    def __contains__(self, i):
        def aux(t):
            if t.i == i:
                return True
            elif t.is_leaf():
                return False
            else:
                return any([aux(c) for c in t.children])
        return aux(self)

    def __str__(self):
        # for debugging (used in print)
        return ("%d : %f -> %s" % (self.i, self.p, [a.__str__() for a in self.children])).replace("'", "").replace('"', '')
    def __repr__(self):
        return self.__str__()

def gen(n):
    rng = np.random.default_rng()
    raw_probs = sorted(rng.random(n), reverse=True)
    raw_probs = [x/sum(raw_probs) for x in raw_probs]
    probs = [(i + 1, raw_probs[i]) for i in range(n)]
    ntrees = int(rng.integers(low = 1, high = n, size = 1))
    g = []
    for k in range(ntrees):
        if k == ntrees - 1:
            size_tree = len(probs)
        else:
            size_tree = rng.integers(low = 1, high = len(probs) - (ntrees - k) + 1, size = 1)
        pindex = int(rng.integers(low = 0, high = len(probs), size = 1))
        i, p = probs.pop(pindex)
        tree = DepNode(p, i)
        for _ in range(1, int(size_tree)):
            pos = int(rng.integers(low = 0, high = tree.n, size = 1))
            pindex = int(rng.integers(low = 0, high = len(probs), size = 1))
            i, p = probs.pop(pindex)
            tree.insert(DepNode(p, i), tree.l()[pos])
        g.append(tree)
    return g, raw_probs

def gen_uni_list(n):
    probs = [1/n] * n
    g = [DepNode(1/n, 1), DepNode(1/n, 2), DepNode(1/n, 3)]
    for i in range(3, n):
        g[i % 3].insert(DepNode(1/n, i + 1), i - 2)
    print(g[0], g[1], g[2])
    return g, probs

def gen_12(n):
    g = [DepNode(1/3, 1), DepNode(1/3, 2)]
    g[1].insert(DepNode(1/3, 3), 2)
    return g, [1/3]*3

def gen_no_deps(n):
    rng = np.random.default_rng()
    # raw_probs = sorted(rng.random(n), reverse=True)
    raw_probs = [1000000000] + [1] * (n - 1)
    raw_probs = [x/sum(raw_probs) for x in raw_probs]
    probs = [(i + 1, raw_probs[i]) for i in range(n)]
    g = []
    for i in range(n):
        i = int(rng.integers(low = 0, high = len(probs), size = 1))
        index, p = probs.pop(i)
        g.append(DepNode(p, index))
    return g, raw_probs

def gen_sqrt(n):
    s = int(np.sqrt(n))
    raw_probs = [s*1000]*s + [1]*(n-s)
    raw_probs = [x/sum(raw_probs) for x in raw_probs]
    p0 = raw_probs[0]
    eps = raw_probs[-1]
    g = [DepNode(eps, i) for i in range(1, s+1)]
    for i in range(s+1, n+1):
        if i > n - s:
            g[(i-1) % s].insert(DepNode(p0, i), i - s)
        else:
            g[(i-1) % s].insert(DepNode(eps, i), i - s)
    return g, raw_probs




def b(i_tree, j_tree, i, j):
    res = 1
    for l in i_tree.lambda_l(i):
        gamma_j = j_tree.subtree(j).gamma()
        gamma_l = i_tree.subtree(l).gamma()
        lambda_j = j_tree.lambda_l(j)
        delta_j = len(lambda_j) - 1
        alpha = 0
        # bpar = [b(i_tree, j_tree, l, lambda_j[k]) for k in range(1, delta_j +
         1)] # duplicate
        bpar = [1] * delta_j # This is to bound b(l, par^(k-1)(j)) by 1
        for m in range(delta_j + 1):
            for k in range(1, m):
                alpha += bpar[k - 1] * gamma_j ** (k - 1)
        res *= (gamma_l/(gamma_l + gamma_j) + alpha)
        print(i, j, res)
    if res > 1:
        return 1
    return res

def all_b(n, g, anc_bound = False):
    # Get the values for all b(i, j)
    nodes = [t.nodes() for t in g]
    b = []
    for i in range(n + 1):
        b.append([])
        for j in range(n + 1):
            b[i].append(int(i == 0))
    # With the way lists are generated, children will be after their parents
    for l in nodes:
        for i in range(len(l)):
            for j in range(i+1, len(l)): # j is behind i, b(i, j) = 1
                b[l[i].i][l[j].i] = 1
    # Now we need to compute b for i !=j and b(i, j) != 1 and b(j, i) != 1
    def aux(i_tree, j_tree, i, j):
        if b[i][j] + b[j][i] != 0:
            return b[i][j]
        else:
            gamma_i = i_tree.subtree(i).gamma()
            gamma_j = j_tree.subtree(j).gamma()
            anc_j = j_tree.lambda_l(j)
            delta_j = len(anc_j) - 1
            try:
                par_i = i_tree.lambda_l(i)[1]
            except IndexError:
                par_i = 0
            res = gamma_i / (gamma_i + gamma_j)
            for m in range(delta_j + 1):
                for k in range(1, m):
                    if anc_bound:
                        res += gamma_j ** (k - 1)
                    else:
                        res += b[i][anc_j[k]] * gamma_j ** (k - 1)
            if anc_bound:
                return min(res, 1)
            return min(res * b[par_i][j], 1)
    for n_li in range(len(nodes)):
        for n_lj in range(n_li):
            li = nodes[n_li]
            lj = nodes[n_lj]
            for i in li:
                for j in lj:
                    curr = aux(li[0], lj[0], i.i, j.i)
                    b[i.i][j.i] = curr
                    b[j.i][i.i] = 1 - curr
    return b

def last_b(n, g, anc_bound = False):
    # Get the values for all b(i, j)
    nodes = [t.nodes() for t in g]
    b = []
    for i in range(n + 1):
        b.append([])
        for j in range(n + 1):
            b[i].append(int(i == 0))
    # With the way lists are generated, children will be after their parents
    for l in nodes:
        for i in range(len(l)):
            for j in range(i+1, len(l)): # j is behind i, b(i, j) = 1
                b[l[i].i][l[j].i] = 1
    # Now we need to compute b for i !=j and b(i, j) != 1 and b(j, i) != 1
    def aux(i_tree, j_tree, i, j):
        if b[i][j] + b[j][i] != 0:
            return b[i][j]
        else:
            gamma_j = j_tree.subtree(j).gamma()
            anc_j = j_tree.lambda_l(j)
            delta_j = len(anc_j) - 1
            try:
                par_i = i_tree.lambda_l(i)[1]
            except IndexError:
                par_i = 0
            res = 1
            for m in range(1, delta_j + 1):
                res -= (b[i][anc_j[m]] * gamma_j ** m) / (1 - gamma_j)
            if anc_bound:
                return min(res, 1)
            return min(res * b[par_i][j], 1)
    for n_li in range(len(nodes)):
        for n_lj in range(n_li):
            li = nodes[n_li]
            lj = nodes[n_lj]
            for i in li:
                for j in lj:
                    curr = aux(li[0], lj[0], i.i, j.i)
```

```python
                    b[i.i][j.i] = curr
                    b[j.i][i.i] = 1 - curr
    return b
def e_mrf(n, g, p, b):
    res = 1
    for j in range(1, n+1):
        # locate j
        t = 0
        for i in range(len(g)):
            if j in g[i]:
                t = i
                break
        # j is in tree t
        subsum = 0
        for k in range(len(g)):
            if k != t:
                subsum += sum([b[i][j] for i in g[k].l()])
        res += p[j - 1] * (len(g[t].lambda_l(j)) - 1 + subsum)
    return 2 * res

def e_mrf_pessimistic(n, g):
    res = 0
    for t in g:
        for node in t.nodes():
            res += node.p * (n - len(t.l()) + 1)
    return 2 * res

def e_mrf_simple(n, g):
    def aux(i, ti, gamma_j):
        res = 1
        for l in ti.anc(i):
            res *= l.gamma() / (l.gamma() + gamma_j)
        return 1 - res + sum([aux(c, ti, gamma_j) for c in i.children])
    res = e_mrf_pessimistic(n, g)
    for j in g:
        sub = 0
        gamma_j = j.gamma()
        for ti in g:
            # iterate over other trees
            if j != ti:
                sub += aux(ti, ti, gamma_j)
        res += 2 * j.p * sub
    return res

def e_mrf_last(n, g):
    def aux_p(i, ti, j):
        res = 1
        gamma_j = j.gamma()
        for l in ti.anc(i):
            gamma_l = l.gamma()
            res *= gamma_l / (gamma_l + gamma_j)
        return res
    res = n
    for t in g:
        for j in t.nodes():
            delta_j = t.delta(j.i)
            gamma_j = j.gamma()
            for ti in g:
                for i in ti.nodes():
                    delta_i = ti.delta(i.i)
                    try:
                        p_par_j = aux_p(t.anc(j)[1], t, i)
                    except:
                        p_par_j = 1
                    res -= j.p * (aux_p(j, t, i) + (p_par_j * (delta_i + 1)) ** (delta_j + 1))
    return 2 * res

def e_statd(n, g, p):
    return sum([j*p[j - 1] for j in range(1, len(p) + 1)])

def ratio(val, generator=gen, anc_bound=False):
    g, p = generator(val)
    b = all_b(val, g, anc_bound)
    print(e_mrf(val, g, p, b), e_mrf_last(val, g), e_mrf_simple(val, g))
    return e_mrf(val, g, p, b) / e_statd(val, g, p)



if __name__ == '__main__':
    values = [9, 49, 100, 400] # range(10, 100, 2)
    n_iter = len(values)
    avg = []
    maxi = []
    mini = []
    if len(sys.argv) != 2:
        print("Usage: ./ratio_MRF.py <sample_size>")
        exit()
    N = int(sys.argv[1])
    for n in range(n_iter):
        results = Parallel(n_jobs=-1)(delayed(ratio)(values[n], generator=gen_sqrt, anc_bound=False) for _ in tqdm(range(N)))
        avg.append(np.mean(results))
        maxi.append(np.max(results) - avg[-1])
        mini.append(avg[-1] - np.min(results))
        print("%s: %s avg, %s min, %s max" % (values[n], avg[n], -mini[n] + avg[n], maxi[n] + avg[n]))
    plt.errorbar(values, avg, [mini, maxi])
    plt.show()
```

Bibliography

Susanne Albers and Sonja Lauer. On list update with locality of reference. *Journal of Computer and System Sciences*, 82(5):627–653, 2016. ISSN 0022-0000. doi:https://doi.org/10.1016/j.jcss.2015.11.005. URL `https://www.sciencedirect.com/science/article/pii/S0022000015001166`.

Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. cambridge university press, 2005.

Fan R. K. Chung, D. J. Hajela, and Paul D. Seymour. Self-organizing sequential search and hilbert's inequalities. *J. Comput. Syst. Sci.*, 36(2):148–157, 1988. doi:10.1016/0022-0000(88)90025-6. URL `https://doi.org/10.1016/0022-0000(88)90025-6`.

Srikrishnan Divakaran. An optimal offline algorithm for list update. *CoRR*, abs/1404.7638, 2014. URL `http://arxiv.org/abs/1404.7638`.

Maciej Pacut, Juan Vanerio, Vamsi Addanki, Arash Pourdamghani, Gábor Rétvári, and Stefan Schmid. Online list access with precedence constraints. *CoRR*, abs/2104.08949, 2021.

Nick Reingold and Jeffery Westbrook. Off-line algorithms for the list update problem. *Information Processing Letters*, 60(2):75–80, 1996. ISSN 0020-0190. doi:https://doi.org/10.1016/S0020-0190(96)00144-5. URL `https://www.sciencedirect.com/science/article/pii/S0020019096001445`.

Ronald L. Rivest. On self-organizing sequential search heuristics. *Commun. ACM*, 19(2):63–67, 1976.