UNIVERSITÉ DE NANTES
FACULTÉ DES
SCIENCES ET TECHNIQUES

Plus courts chemins multi-objectifs

FORGET Nicolas
Master 1 ORO
Université de Nantes
Année universitaire 2017-2018

Encadrant : Xavier Gandibleux

Table des matières

1 Introduction

Le problème de plus courts chemins multi-objectif est un problème qui est étudié depuis longtemps [5] [6]. De nombreuses méthodes de résolution existent, parmi lesquelles on trouve l'algorithme de Martins [2]. Il arrive qu'il soit nécessaire d'ajouter des éléments de modélisation supplémentaires au problème. Cela peut se traduire par l'introduction de contraintes spécifiques [4] ou encore par l'ajout de fonctions objectif de type autre que somme (initialement, les fonctions objectif sont de type somme dans le problème de plus courts chemins multi-objectif). Dans ce dernier cas notamment, il est alors nécessaire d'adapter les méthodes de résolution : l'algorithme de Martins a déjà été étendu à certaines d'entre elles [6]. Nous allons ici chercher à modéliser des mesures de fiabilité de chemins en intégrant des objectifs de type produit à la modélisation, puis nous adapterons l'algorithme de Martins à celles-ci.

En parallèle, des études ont été menées sur les tests de dominance et l'utilisation d'une structure de données particulière semble améliorer la rapidité d'exécution de ces tests [1]. Or, l'algorithme de Martins nécessite de réaliser des tests de dominance, et parfois en très grande quantité. Nous allons donc dans un second temps nous interroger sur l'intérêt d'utiliser une structure de données particulière pour réaliser les tests de dominance de l'algorithme de Martins dans le but de réduire le temps de résolution de problèmes de plus courts chemins multi-objectifs.

2 Définitions et notations

2.1 Théorie des graphes

2.1.1 Définition

Un graphe est une structure de données qui met en relation des objets entre eux [4]. Les objets seront appelés sommets du graphe et une relation sera représentée par une arête entre les deux objets concernés. Ainsi, pour définir un graphe G, il faut un ensemble de sommets (noté S) et un ensemble d'arêtes (noté A).

2.1.2 Sommets et arêtes

Soit $G(S, A)$ un graphe quelconque. L'ensemble S représente l'ensemble des sommets de G et on notera par la suite que $|S| = n$. Par exemple, un sommet peut représenter une localisation (ou point) particulière sur une carte.

L'ensemble A contient toutes les arêtes de G. Un arêtes est définie par un couple de sommets. Ainsi, une arête reliant un sommet i et un sommet j s'écrira (i, j), et on a $A \subseteq S \times S$. Par exemple, une arête peut représenter l'existence d'une route directe entre deux points sur une carte.

Il est possible d'orienter les arêtes pour montrer qu'une relation entre deux sommets existe dans un sens mais pas dans un autre. On parle alors non plus d'arêtes mais d'arcs. On appelle graphe orienté un graphe avec de tels arcs. Par exemple, il est possible qu'on ne puisse emprunter une route que dans un seul sens. On peut donc se retrouver à pouvoir aller d'un point x à un point y mais de ne pas pouvoir aller de y à x. Il est important de noter que l'on peut aisément transformer un graphe non-orienté G en un graphe orienté G' en remplaçant chaque arc de G par deux arcs dans G' (les deux seront attachées aux mêmes sommets, mais l'une le sera dans un sens et l'autre dans l'autre sens). De ce fait, nous n'utiliserons par la suite que des graphes orientés.

2.1.3 Successeurs et prédécesseurs

Soient $i, j \in S$ deux sommets d'un graphe $G(S, A)$. On dit que j est successeur de i si et seulement s'il existe un arc $(i, j) \in A$ qui part de i et qui va vers j. On dit que j est prédécesseur de i s'il existe un arc $(j, i) \in A$ qui part de j et qui va vers i.

2.1.4 Chemins dans un graphe

Soient $i, j \in S$ deux sommets d'un graphe G. On appellera $r_{i,j}$ un chemin allant du sommet i au sommet j. Celui-ci est défini par un ensemble d'arcs reliant i à j et où le sommet d'arrivée d'un arc est le sommet de départ de la suivante. Ainsi, soit $s_1, ..., s_t \in S$, on a $r_{i,j} = \{(i, s_1), (s_1, s_2), ..., (s_{t-1}, s_t), (s_t, j)\}$. On dit que $r_{f,h}$ est un sous chemin d'un chemin $r_{i,j}$ si $r_{f,h} \subset r_{i,j}$. Il nous arrivera par la suite de dire qu'un sommet est situé sur un chemin r. Cela signifie qu'un tel sommet est inclus dans au moins une des arcs de r. Par exemple, s_2 est un sommet situé sur $r_{i,j}$.

On notera $R_{i,j}$ l'ensemble des chemins reliant i à j et $R_{i,.}$ l'ensemble des chemins reliant i à tout les autres sommets du graphe.

2.1.5 Poids des arcs

Il est possible d'ajouter un poids (ou coût) sur chacune des arcs. Par exemple, celui-ci peut représenter la distance parcourue d'un point x à un point y en utilisant la route directe de x à y sur une carte. Les coûts des arcs seront donnés par une matrice C de taille $n \times n$ où l'élément $c_{i,j}$ nous donne le coût de l'arc allant du sommet i au sommet j.

Il sera parfois nécessaire d'ajouter plusieurs valeurs sur une même arc. Nous n'aurons alors plus un poids mais un vecteur de poids (ou coûts) sur chaque arc. S'il y a p valeurs sur chaque arc, les coûts seront donnés par p matrices C^k, $k \in \{1, ..., p\}$ et où l'élément $c^k_{i,j}$ représente le k^{eme} coût de l'arc allant du sommet i au sommet j.

Un graphe $G(S, A, C)$ muni de un ou plusieurs poids sur chaque arc est appelé graphe pondéré.

2.2 Optimisation Multi-Objectif

2.2.1 Présentation

L'optimisation multi-objectif cherche à optimiser plusieurs fonctions objectifs généralement en concurrence (c'est-à-dire que la solution optimale d'une des fonctions n'est pas la même que celle des autres) simultanément et sous un même ensemble de contraintes. Ainsi, soit $x \in D \subseteq \mathbb{R}^n$, un programme d'optimisation multi-objectif à p objectifs et à m contraintes s'écrit de la manière suivante :

$$\begin{array}{llll} \{\min,\max\} & z^k(x) & & \forall k \in \{1, ..., p\} \\ \text{s.c.} & g_i(x) & \{\leq, \geq, =\} \quad b_i & \forall i \in \{1, ..., m\} \\ & x & \in \quad D & \end{array}$$

Par la suite, nous noterons toujours les numéros d'objectif à l'exposant tandis que les indices divers seront mis en indice.

2.2.2 Dominance

En optimisation mono-objectif, on associe à une solution $x \in D$ une valeur $z(x)$ qui est la valeur de la fonction objectif. Afin de comparer des solutions, on compare leurs valeurs de fonction objectif. En optimisation multi-objectif à p objectifs, on associe à une solution $x \in D$ un vecteur de valeurs $z(x) = (z^1(x), ..., z^p(x))$ dont la k^{eme} valeur pour $k \in \{1, ..., p\}$ représente la valeur de la k^{eme} fonction objectif. L'image de l'espace des solutions réalisables par z est appelée espace des objectifs et sera notée Z. On a en particulier que $Z \subseteq \mathbb{R}^p$

Les outils de comparaison utilisés en mono-objectif ($\leq$,$\geq$,=,...) ne sont alors plus suffisants et il est nécessaire d'en définir de nouveaux : les relations de dominance. Considérons deux points de l'espace des objectifs $x, y \in Z$.

Définition 1. On dit que x domine y si et seulement si $\forall k \in \{1, ..., p\}$, $x^k \leq y^k$ si l'objectif k est à minimiser et $x^k \geq y^k$ si l'objectif k est à maximiser, avec au moins un objectif pour lequel l'inégalité est stricte. On notera cette relation $x >_d y$.

Définition 2. On dit que y est faiblement non-dominé par x si et seulement si $\forall k \in \{1, ..., p\}$, $x^k \leq y^k$ si l'objectif k est à minimiser et $x^k \geq y^k$ si l'objectif k est à maximiser et avec au moins un des objectifs à l'égalité et au moins un des objectifs à l'inégalité stricte. On notera cette relation $x \geq_d y$.

Définition 3. Soit $k^* = \min_{k \in \{1,...,p\}} \{k \mid x^k \neq y^k\}$. On dit que x est plus petit lexicographiquement que y si $x^{k^*} < y^{k^*}$ si l'objectif k est à minimiser ou si $x^{k^*} > y^{k^*}$ si l'objectif k est à maximiser. On notera cette relation $x <_{lex} y$.

On remarquera par ailleurs que l'opération $>_d$ est transitive. En effet, soit $x, y, z \in Z$ tels que $x >_d y >_d z$. On a pour un objectif k à minimiser $x^k \leq y^k$ et $y^k \leq z^k$. On a donc bien $x^k \leq z^k$. Pour un objectif k à maximiser, on a $x^k \geq y^k$ et $y^k \geq z^k$, et donc $x^k \geq z^k$. Ainsi, on a donc bien $x >_d z$: la relation est bien transitive.

2.2.3 Optimalité

En optimisation mono-objectif, nous n'avons qu'une solution optimale qui correspond à la solution donnant la meilleure valeur de la fonction objectif sous l'ensemble des contraintes. Il est possible d'avoir plusieurs solutions optimales mais celles-ci donneront toutes la même valeur pour la fonction objectif. En optimisation multi-objectif, nous n'aurons pas une solution optimale, mais un ensemble de solutions Pareto-optimales, aussi appelées ensemble des points non-dominés dans l'espace des objectifs.

Définition 4. On dit qu'un point $y \in Z$ est non-dominé s'il n'existe pas de point $y' \in Z$ tel que $y' >_d y$.

Il est parfois possible d'avoir pour un même point de l'espace des objectifs d'avoir plusieurs solutions associées. Soit $E \subset Z$ l'ensemble des points non-dominés de Z et $X(E)$ un l'ensemble des solutions associées.

Définition 5. On dit que $X(E)$ est minimum complet si pour chaque point de E il y a exactement une solution dans $X(E)$.

Définition 6. On dit que $X(E)$ est complet si pour chaque point de E il y a une ou plusieurs solutions correspondantes dans $X(E)$.

Définition 7. On dit que $X(E)$ est maximum complet si pour chaque point de E il y a toutes les solutions correspondantes dans $X(E)$.

2.2.4 Point nadir et point idéal

Le point nadir est le point de l'espace des objectifs Z tel qu'il est dominé par chaque point z appartenant à l'ensemble des points non-dominé de l'espace des objectifs Z_{nd} . Soient z_* le point nadir et $k \in \{1, ..., p\}$, on a donc que $z_*^k = \max_{z \in Z_{nd}} z^k$ si l'objectif k est à minimiser et $z_*^k = \min_{z \in Z_{nd}} z^k$ si l'objectif k est à maximiser.

Déduction 1. Le point nadir z_* est dominé ou faiblement non-dominé par tout point non-dominé $z \in Z_{nd}$.

Le point idéal est le point de l'espace des objectifs tel que l'on ne peut obtenir un point meilleur que celui-ci. Soient z^* le point nadir et $k \in \{1, ..., p\}$, on a donc que $z^{k*} = \min_{z \in Z} z^k$ si l'objectif k est à minimiser et $z^{k*} = \max_{z \in Z} z^k$ si l'objectif k est à maximiser.

Déduction 2. Tout point $z \in Z$ est dominé ou faiblement non-dominé par le point idéal z^*.

2.3 Problèmes de plus court chemin

2.3.1 Mono-objectif

Le problème de plus court chemin consiste à trouver dans un graphe $G(S, A, C)$ le plus court chemin reliant un sommet s à un sommet t. Le coût sur chaque arc peut par exemple représenter une distance ou un temps de trajet. L'objectif est de trouver le chemin qui minimise l'objectif qui est fixé. Mathématiquement, ce problème se modélise de la manière suivante [2] : soit $x_{ij} \in \{0, 1\}$ une variable telle que $x_{ij} = 1$ si et seulement si on emprunte l'arc (i, j) du graphe, $\forall i, j \in S$. On notera que si l'arc (i, j) n'existe pas, alors on considère que $c_{ij} = +\infty$. Ainsi, on a

$$
\begin{array}{lllll}
\min & z(x) = \sum_{i \in S} \sum_{j \in S} c_{ij} x_{ij} & & & \\
\text{s.c.} & \sum_{j \in S} x_{sj} - \sum_{j \in S} x_{js} & = & 1 & \\
& \sum_{j \in S} x_{ij} - \sum_{j \in S} x_{ji} & = & 0 & \forall i \in S \backslash \{s,t\} \\
& \sum_{j \in S} x_{tj} - \sum_{j \in S} x_{ts} & = & \text{-1} & \\
& x_{ij} & \in & \{0,1\} & \forall i,j \in S
\end{array}
$$

Dans un graphe, cela revient à chercher le chemin $r \in R_{s,t}$ qui minimise $z(r) = \sum_{(i,j) \in r} c_{ij}$. On dira par la suite qu'un tel objectif est de type somme car on somme des coefficients pour avoir la valeur d'un chemin. Il existe différents algorithmes de résolution exacte, comme celui de Dijkstra ou de Bellman-Ford.

2.3.2 Multi-objectif

Parfois, il n'est pas suffisant de ne regarder qu'un seul objectif. Par exemple, il est possible que le chemin présentant la plus petite distance dans notre graphe présente un temps de trajet bien trop grand pour être réalisable. Il va falloir alors regarder 2 objectifs simultanément : la distance et le temps de trajet. Chaque arc sera alors muni de deux poids et notre problème de plus courts chemins se transformera en problème de plus courts chemins multi-objectifs. Le problème de plus courts chemins multi-objectif consiste à trouver l'ensemble des chemins Pareto-optimaux dans un graphe $G(S, A, C)$ reliant un sommet s à un sommet t. Par la suite, nous aurons p objectifs et nous aurons toujours pour point de départ le sommet s et pour point d'arrivée de sommet t. De la même manière qu'en mono-objectif, le problème se modélise mathématiquement comme ceci [2] :

$$
\begin{array}{lllll}
\min & z^k(x) = \sum_{i \in S} \sum_{j \in S} c^k_{ij} x_{ij} & & & \forall k \in \{1, ..., p\} \\
\text{s.c.} & \sum_{j \in S} x_{sj} - \sum_{j \in S} x_{js} & = & 1 & \\
& \sum_{j \in S} x_{ij} - \sum_{j \in S} x_{ji} & = & 0 & \forall i \in S \backslash \{s,t\} \\
& \sum_{j \in S} x_{tj} - \sum_{j \in S} x_{ts} & = & \text{-1} & \\
& x_{ij} & \in & \{0,1\} & \forall i,j \in S
\end{array}
$$

Dans un graphe, cela revient à trouver les chemins $r \in R_{s,t}$ qui sont non-dominés. Si on a p objectifs de type somme, on a $z(r) = (\sum_{(i,j) \in r} c^1_{ij}, ..., \sum_{(i,j) \in r} c^p_{ij})$. Nous verrons dans la section suivante qu'il est possible d'avoir d'autres types de fonctions objectif et d'étendre le problème de plus courts chemins à celles-ci.

Le problème de plus courts chemins multi-objectif est NP-dur, mais fait partie de ceux qui passent le mieux à l'échelle [5]. Il a été prouvé par ailleurs par Hansen en 1979 qu'il existe des instances de ce problème pour lesquelles le nombre de solutions Pareto-optimales est exponentiel. Cela a pour conséquence que les algorithmes de résolution exacte sont dans le pire des cas exponentiels [5], y compris celui que nous allons étudier par la suite.

3 Algorithme de résolution exacte : l'algorithme de Martins

3.1 Algorithme de Martins

3.1.1 Présentation

L'algorithme de Martins, comme présenté dans son papier de 1984 [2], est un algorithme de résolution exact de problèmes de plus courts chemins multi-objectifs à p objectifs de type somme.

Cet algorithme utilise un objet particulier : le label. Ce dernier est constitué de $p+2$ valeurs. Les p premières correspondent aux p valeurs d'un chemin $r \in R_{s,i}$, $i \in S$. La $p+1^{eme}$ valeur est un pointeur vers un sommet j et la dernière valeur est un pointeur vers un label de j. Un label peut être de deux types différents : temporaire (c'est-à-dire qu'il est susceptible d'être supprimé au cours de l'algorithme) ou permanent (c'est-à-dire qu'il ne peut être supprimé et correspondra donc à un chemin Pareto-optimal, cela est montré dans la preuve qui suit). L'algorithme de Martins est un algorithme dit de *label setting* [3] : à chaque itération de l'algorithme, de nouveaux labels sont construits. L'algorithme est donné par **Algorithm 1**.

Notation. Soit un label l, on notera la k^{eme} valeur du label l : $z^k(l)$, $\forall\ k \in \{1, ..., p\}$.

Algorithm 1 Algorithme de Martins

p : nombre d'objectif de type somme
C^k : matrice des coûts de l'objectif k, $k \in \{1, ..., p\}$

assigner à s le label temporaire $[(0, ..., 0), -, -]$
while $\exists$ label(s) temporaire(s) dans le graphe **do**
  $l \leftarrow$ plus petit label temporaire par ordre lexicographique
  passer l permanent
  // soit l le N^{eme} label du sommet i, on a :
  $i \leftarrow$ sommet de l
  $N \leftarrow$ numéro de label de l
  for chaque successeur j de i **do**
    $l' \leftarrow [(z^1(l) + c^1_{ij}, ..., z^p(l) + c^p_{ij}), i, N]$
    ajouter l' au sommet j en tant que label temporaire
    if l' est dominé par un label de j **then**
      Supprimer l'
    else if l' domine un label L de j **then**
      Supprimer L
    end if
  end for
end while
backtracker à l'aide des pointeurs depuis le sommet t pour retrouver tous les chemins Pareto-optimaux

Cet algorithme permet de trouver tous les chemins Pareto-optimaux d'un graphe et de restituer l'ensemble maximum complet des solutions efficaces.

3.1.2 Preuve

Nous allons reprendre ici la preuve réalisé dans le livre de M. Ehrgott [3]. Celle-ci se décompose en deux parties : nous montrerons dans un premier temps qu'un label permanent correspond toujours à un chemin Pareto-optimal puis nous montrerons que tous les chemins Pareto-optimaux seront trouvés.

Un label permanent correspond forcément à un chemin Pareto-optimal. Soit l le label temporaire le plus petit par ordre lexicographique dans le graphe, r_l le chemin décrit par l (r_l

peut être retrouvé par le backtracking sur les pointeurs) et m le sommet sur lequel est posé l. Selon l'algorithme, l va être passé permanent. Supposons maintenant que r_l n'est pas un chemin Pareto-optimal. Cela signifie qu'il existe un chemin $r'_l \neq r_l$ tel que $r'_l \in R_{s,m}$ et que $z^k(r'_l) \leq z^k(r_l) \quad \forall k \in \{1, ..., p\}$ avec au moins un k pour lequel l'inégalité est stricte. Soit m' un sommet situé sur le chemin r'_l, et l' un label du chemin r'_l posé sur m', cela implique que $\exists\, r_{m',m} \in R_{m',m}$ tel que $z^k(l') + \sum_{(i,j)\in r_{m',m}} c^k_{i,j} \leq z^k(l)$. Les c_{ij} étant positifs, on a que $z^k(l') \leq z^k(l)$ avec au moins un objectif ayant l'inégalité stricte. Cela implique donc que l' est plus petit lexicographiquement que l, ce qui entre en contradiction avec le fait que l soit le plus petit label par ordre lexicographique. Donc tout label permanent correspond à un label Pareto-optimal.

Tous les chemins Pareto-optimaux seront trouvés par l'algorithme de Martins. Nous avons pour cela besoin de montrer que tout sous-chemin d'un chemin Pareto-optimal est aussi Pareto-optimal. Soit $r_{s,i}$ un sous chemin d'un chemin Pareto-optimal $r_{s,t}$ et $r_{i,t}$ le reste du chemin (c'est-à-dire que $r_{s,i} \cup r_{i,t} = r_{s,t}$). Si $r_{s,i}$ n'est pas Pareto-optimal, cela signifie qu'il existe un chemin $r'_{s,i}$ qui domine $r_{s,i}$. Or, cela nous donne un nouveau chemin $r'_{s,i} \cup r_{i,t}$ qui domine $r_{s,t}$, ce qui entre en contradiction avec la fait que celui-ci soit Pareto-optimal. Ainsi, tout sous-chemin d'un chemin Pareto-optimal est aussi Pareto-optimal.

Supposons qu'un chemin Pareto-optimal n'ai pas été trouvé : soit il a été supprimé, soit il n'a jamais été trouvé. S'il a été supprimé, cela signifie que le chemin ou un de ses sous-chemin a été dominé par un autre, ce qui est impossible car le chemin est Pareto-optimal et tout sous chemin d'un tel chemin est aussi Pareto-optimal. Supposons donc qu'il n'a jamais été trouvé. Soit $r_{s,t} = \{(s, i_1), ..., (i_h, t)\}$ avec $h \in \mathbb{N}$ et $i_1, ..., i_h \in S$. Cela implique que r_{s,i_h} n'a pas été trouvé, ainsi que $r_{s,i_{h-1}}$, etc... Par récurrence, on arrive au fait que r_{s,i_1} n'a pas été trouvé et que donc $r_{s,s}$ non plus, ce qui est impossible car la première étape de l'algorithme pose un label sur s. Ainsi, tous les chemins Pareto-optimaux sont bien trouvés.

3.2 Extension de l'algorithme de Martins aux fonctions Min-Max

Cette extension de l'algorithme a été étudiée dans le papier de X.Gandibleux, F.Beugnies et S.Randriamasy en 2006 [6]. L'idée est d'introduire un objectif de type max(min(.)) (aussi souvent appelé *bottleneck*) en plus de p fonctions objectif de type somme. Un tel objectif peut par exemple être utilisé pour modéliser de la bande passante que l'on souhaite maximiser sur un circuit électronique.

Soit un chemin $r \in R_{s,t}$, on cherchera ici à maximiser le plus petit coût présent sur les arcs de r. La matrice des coûts de cet objectif $m = p + 1$ sera notée M et le coût d'un arc (i, j) sera noté m_{ij}. Ainsi, on a $z^m(r) = \min_{(i,j)\in r}(m_{ij})$ et on cherche à maximiser $z^m(r)$. Ainsi, nous avons un nouveau problème de plus courts chemins multi-objectif : nous souhaitons trouver tout les chemins non-dominés $r \in R_{s,t}$ avec $z(r) = (\sum_{(i,j)\in r} c^1_{ij}\ ,\ ...\ ,\ \sum_{(i,j)\in r} c^p_{ij}\ ,\ \min_{(i,j)\in r}(m_{ij}))$.

Dans le papier de 2006 [6], il est proposé de recycler l'algorithme de Martins pour y intégrer une fonction de type *bottleneck*. On appelera cette nouvelle version de l'algorithme la version $(p - \Sigma \mid 1 - M)$. Ici, deux éléments vont être impactés. Le premier est la propagation des coûts lors de la création d'un nouveau label temporaire. Pour les fonctions somme, la procédure reste inchangée. En revanche, pour la fonction max(min(.)), il faut vérifier si le coût de l'arc traversée est inférieur au coût du label de départ. Si c'est le cas, le coût du nouveau label sera le coût de l'arc, sinon le coût restera le même que celui du label de départ. La mise à jour du coût de l'objectif m se fera à l'aide de la fonction *UpdateMinMax*, décrite dans **Algorithm 2**.

Le second élément qui change avec ce type de fonction objectif est le test de dominance. En effet, il est possible que certains labels faiblement dominés par d'autres correspondent à des chemins Pareto-optimaux. Soient l et l' deux labels posés sur un même sommet i et tels que $z^k(l) = z^k(l') \ \forall\ k \in \{1, ..., p\}$ et $z^m(l) > z^m(l')$. On dira que l' est faiblement non-dominé sur la fonction max(min(.)). Soit j un sommet successeur de i et (i, j) un arc telle que $m_{ij} < z^m(l') < z^m(l)$. On remarque que si l'on passe par l'arc (i, j) depuis i, les deux nouveaux labels L et L' créés sur j respectivement à partir de l et l', auront pour m^{eme} valeur m_{ij}. Ainsi, on aura

Algorithm 2 UpdateMinMax

l : label du sommet précédent
i : sommet de l
j : sommet du label que l'on calcule
M : matrice des coûts de l'objectif max(min(.))

```
if m_ij < z^m(l) then
  return m_ij
else
  return z^m(l)
end if
```

$z^k(L) = z^k(L')\ \forall\ k \in \{1, ..., p, m\}$ et L et L' correspondront à deux chemin Pareto-optimaux à part entière . Par exemple, sur la figure 1 on a $m_{ij} = 2$, $z^m(l) = 6$ et $z^m(l') = 4$. Lors de la propagation des coûts, nous obtenons deux labels de même valeur sur j. Ainsi, là où l'algorithme de Martins initial supprime les labels faiblement non-dominés, celui-ci doit conserver ceux qui sont faiblement non-dominés par la fonction max(min(.)).

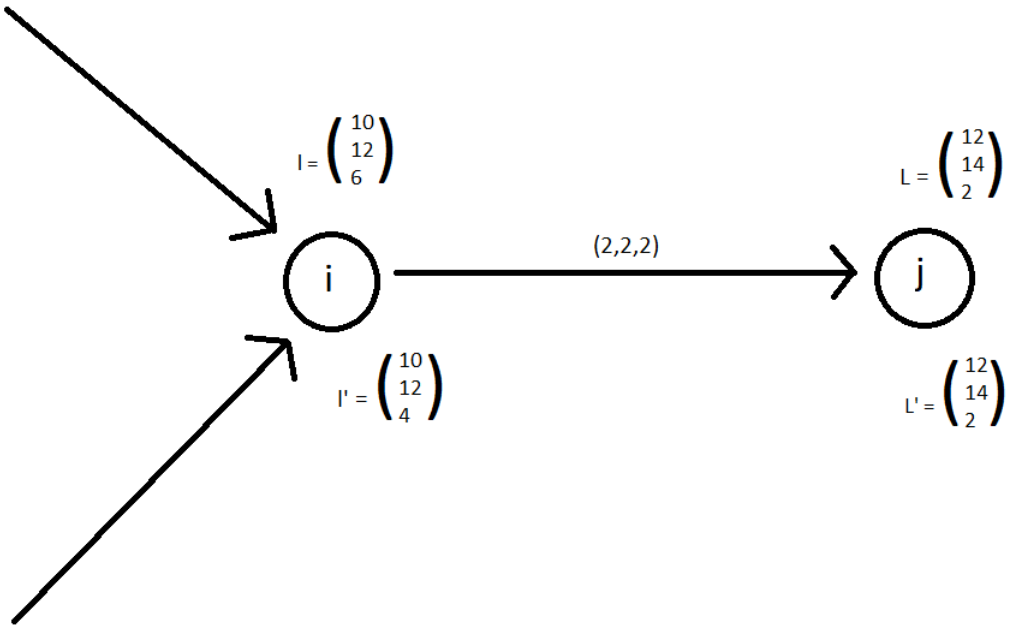

FIGURE 1 – exemple de label faiblement non-dominé par une fonction *bottleneck* étant un chemin Pareto-optimal en $(2 - \Sigma \mid 1 - M)$

En revanche, si $i = t$, c'est-à-dire que i est le sommet d'arrivée, le label l' ne représente pas un chemin Pareto-optimal. Il va donc être nécessaire d'ajouter une caractéristique supplémentaire aux labels : caché ou découvert. Ainsi, tout label sera à sa création découvert mais peut devenir caché s'il est faiblement non-dominé par la fonction max(min(.)). Pendant le déroulement de l'algorithme, les deux types de labels seront pris en compte mais lors de la restitution des chemins de s à t, nous ne considérerons que les labels découverts sur t. Le nouvel algorithme que nous obtenons est décrit dans **Algorithm 3**.

Comme pour l'algorithme initial, cet algorithme permet de trouver tous les chemins Pareto-optimaux d'un graphe et de restituer l'ensemble maximum complet des solutions efficaces.

3.3 Introduction de fonctions de type produit

Nous allons introduire dans cette section un troisième type de fonction objectif. L'intérêt de celles-ci est de réaliser une mesure de fiabilité sur un chemin d'un graphe. Puis, comme pour les

Algorithm 3 Algorithme de résolution de plus courts chemins $(p - \Sigma \mid 1 - M)$

p : nombre d'objectif de type somme
C^k : matrice des coûts de l'objectif k, $k \in \{1, ..., p\}$
M : matrice des coûts de l'objectif max(min(.))

assigner à s le label temporaire découvert $[(0, ..., 0, \infty), -, -]$
while $\exists$ label(s) temporaire(s) dans le graphe **do**
  $l \leftarrow$ plus petit label temporaire (découvert ou caché) par ordre lexicographique
  Passer l permanent
  // soit l le N^{eme} label du sommet i, on a :
  $i \leftarrow$ sommet de l
  $N \leftarrow$ numéro de label de l
  for chaque successeur j de i **do**
    $l' \leftarrow [(z^1(l) + c^1_{ij}\ ,\ ...\ ,\ z^p(l) + c^p_{ij}\ ,\ UpdateMinMax(l, i, j, M))\ ,\ i\ ,\ N]$
    ajouter l' au sommet j en tant que label temporaire découvert
    if l' est faiblement non-dominé sur la fonction max(min(.)) par un label de j **then**
      Cacher l'
    else if l' est dominé par un label de j **then**
      Supprimer l'
    else if l' domine un label L de j **then**
      Supprimer L
    else if un label L de j est faiblement non-dominé sur la fonction max(min(.)) **then**
      Cacher L
    end if
  end for
end while
backtracker sur les labels découverts de t à l'aide des pointeurs pour retrouver tous les chemins Pareto-optimaux

fonctions max(min(.)), nous chercherons à étendre l'algorithme de Martins à celles-ci.

3.3.1 Contexte et définition d'une fonction de type produit

Des fonctions de type produit ont déjà été définies par le passé [5], mais nous allons les redéfinir pour les faire concorder avec le contexte que l'on se fixe et dans le but de les introduire à l'algorithme de Martins.

Soit un graphe orienté $G(S, A, Q)$, Q représentant une matrice de coûts sur les arcs de G. Sur chaque arc, on sait qu'il peut y avoir une défaillance et l'on suppose ici que l'on peut quantifier la probabilité que cela arrive. L'objectif est ici de se rendre d'un sommet s à un sommet t du graphe en minimisant le risque qu'un problème apparaisse sur le chemin. On peut le transformer en un objectif similaire : déterminer le chemin de s à t qui maximise la probabilité d'arriver sans encombre à t.

On note p_{ij} la probabilité qu'une défaillance apparaisse sur l'arc (i, j) et q_{ij} la probabilité qu'aucun dysfonctionnement ne survienne sur l'arc (i, j). On remarque tout de suite que :

$$q_{i,j} = 1 - p_{ij} \tag{1}$$

On appellera D_{ij} l'évènement "une défaillance est apparue sur l'arc (i, j)". En particulier, on observe que :

$$p_{ij} = P(D_{ij}) \tag{2}$$

On suppose ici que les D_{ij} sont tous indépendants entre eux ainsi que leurs complémentaires $\overline{D_{ij}}$ ("Aucune défaillance n'est apparue sur l'arc (i, j)). A un instant donné, soit un composant a une défaillance, soit il n'en a pas. On a donc l'égalité suivante :

$$P(D_{ij}) = 1 - P(\overline{D_{ij}}) \tag{3}$$

En terme littéral, notre problème qui est de minimiser la probabilité qu'un risque apparaisse sur un chemin $r = \{(i_1, i_2), (i_2, i_3), ..., (i_{k-2}, i_{k-1}), (i_{k-1}, i_k)\} \in R_{s,t}$ peut se traduire de la manière suivante : on souhaite maximiser la probabilité qu'il n'y ait aucune défaillance sur l'arc (i_1, i_2) et qu'il n'y ait aucune défaillance sur l'arc (i_2, i_3) et ... et qu'il n'y ait aucune défaillance sur l'arc (i_{k-1}, i_k). Mathématiquement, cette probabilité serait : $P(\bigcap_{(i,j)\in r} \overline{D_{ij}})$. Or par indépendance des $\overline{D_{ij}}$, on a :

$$P(\bigcap_{(i,j)\in r} \overline{D_{ij}}) = \prod_{(i,j)\in r} P(\overline{D_{ij}})$$

De plus, par l'équation (3) puis la relation (2) et enfin la relation (1), on a :

$$\prod_{(i,j)\in r} P(\overline{D_{ij}}) = \prod_{(i,j)\in r} (1 - P(D_{ij})) = \prod_{(i,j)\in r} (1 - p_{ij}) = \prod_{(i,j)\in r} q_{ij}$$

Ainsi, afin de résoudre notre problème, nous allons chercher dans le graphe $G(S, A, Q)$ munit d'un poids q_{ij} donné par Q sur chaque arc $(i, j) \in A$ (q_{ij} étant la probabilité qu'aucune défaillance n'apparaisse sur l'arc en question) le chemin $r \in R_{s,t}$ qui maximise la fonction objectif suivante :

$$z(r) = \prod_{(i,j)\in r} q_{ij}$$

On précise que les q_{ij} représentant des probabilités, on a $0 \leq q_{ij} \leq 1$. En particulier, cela aura pour conséquence que $0 \leq z(r) \leq 1, \forall r \in R_{s,t}$.

3.3.2 Introduction des fonctions produit dans l'algorithme de Martins

Nous allons maintenant intégrer q fonctions objectif de type produit à maximiser dans l'algorithme de Martins en plus des p fonctions somme et d'une fonction max(min(.)). Les coûts de la h^{eme} fonction produit pour l'ensemble du graphe seront donnés par la matrice Q^h, et ce pour tout $h \in \{1, ..., q\}$. Nous disposerons les objectifs selon l'ordre suivant : les p fonctions somme, les q fonctions produit, puis la fonction max(min(.)). De ce fait, le numéro d'objectif associé à cette dernière deviendra $m = p + q + 1$. Ainsi, nous avons un nouveau problème qui consiste à trouver l'ensemble des chemins non-dominés $r \in R_{s,t}$ avec $z(r) = (\sum_{(i,j)\in r} c^1_{ij}\ ,\ ...\ ,\ \sum_{(i,j)\in r} c^p_{ij}\ ,\ \prod_{(i,j)\in r} q^1_{ij}\ ,\ ...\ ,\ \prod_{(i,j)\in r} q^q_{ij}\ ,\ \min_{(i,j)\in r}(m_{ij}))$.

Repartons de la version $(p - \Sigma \mid 1 - M)$. L'aspect qui va être ici impacté est la propagation des coûts des q fonctions produit. Soient l un label posé sur un sommet i et (i, j) un arc entre i et un autre sommet j. Le label L créé sur j depuis le label l sera tel que $z^h(L) = z^h(l) \cdot q^h_{ij}$, $\forall\ h \in \{1, ..., q\}$. Le nouvel algorithme est décrit dans **Algorithm 4**.

En théorie, il y a un second élément qui est impacté par l'introduction de ces fonctions. Soient deux labels l et l' posés sur un même sommet i et tels que $z^k(l) = z^k(l')\ \forall\ k \in \{1, ..., p\}$, $z^h(l) \geq z^h(l')\ \forall\ h \in \{1, ..., q\}$ et $z^m(l) = z^m(l')$ et avec au moins un h pour lequel l'inégalité est stricte. On remarque que l' est faiblement non-dominé par l et ce, par les fonctions produit. Prenons maintenant un sommet j et un arc $(i, j) \in A$ telle que $q_{ij} = 0$, $\forall\ h \in \{1, ..., q\}$. Lors de la création des labels L et L' sur le sommet j depuis respectivement l et l', on observe que $z^h(L) = q^h_{ij} \cdot z^h(l) = 0 \cdot z^h(l) = 0$, $\forall\ h \in \{1, ..., q\}$ et qu'il en va de même pour L'. Ainsi, on a que $z(L) = z(L')$. Un exemple est donné sur la figure 2. On en conclut donc qu'un label faiblement non-dominé par les fonctions produit peut lui aussi correspondre à un chemin Pareto-optimal, et qu'il serait nécessaire de développer une stratégie similaire à la fonction max(min(.)) mais étendue à plusieurs fonctions nécessitant ce type de système. Cependant, ce cas se présentera si et seulement s'il y a un coût nul sur un des $q^h_{ij} \in Q^h$. Cela aura donc pour conséquence que ce cas ne se présentera que pour des chemins avec un coût nul pour au moins une des fonctions produit. Or, dans notre contexte, une fonction produit représente une mesure de fiabilité. Ainsi, un chemin avec un coût nul est un chemin où quelque chose se passera forcément mal : il n'est pas fiable. Le décideur ne retiendra donc jamais un tel chemin dans notre contexte. Nous avons donc décidé

d'ignorer ces cas-ci dans l'algorithme. Cela aura pour conséquence que l'algorithme ne trouvera pas certains chemins Pareto-optimaux dont le coût est égal à un autre chemin Pareto-optimal. Ainsi, cet algorithme restituera un ensemble complet des solutions efficaces plutôt que l'ensemble maximum complet.

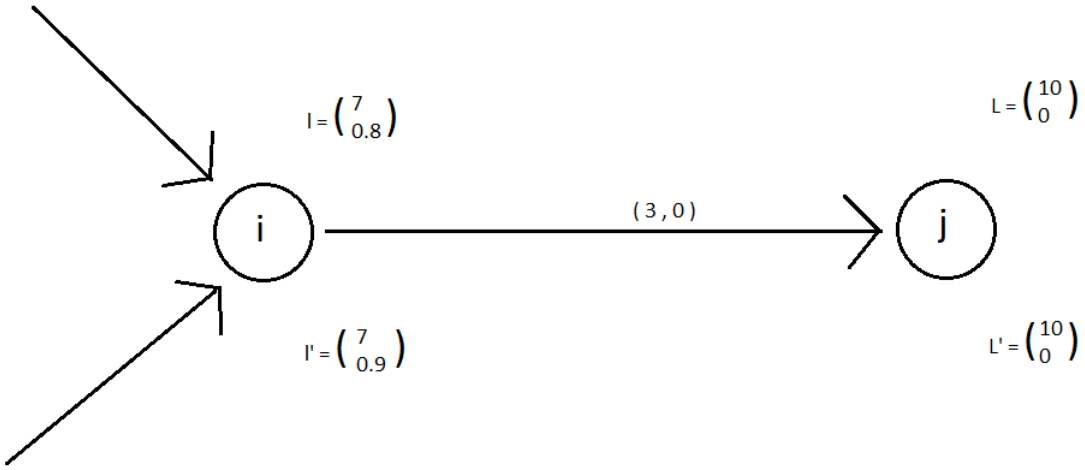

FIGURE 2 – exemple de label faiblement non-dominé par une fonction produit étant un chemin Pareto-optimal en $(1 - \Sigma \mid 1 - \Pi)$

Algorithm 4 Algorithme de résolution de plus courts chemins $(p - \Sigma \mid q - \Pi \mid 1 - M)$

p : nombre d'objectif de type somme
C^k : matrice des coûts de l'objectif k, $k \in \{1, ..., p\}$
Q^h : matrice des coûts de l'objectif $p + h$, $h \in \{1, ..., p\}$
M : matrice des coûts de l'objectif max(min(.))

assigner à s le label temporaire découvert $[(0, ..., 0, 1, ..., 1, \infty), -, -]$
while $\exists$ label(s) temporaire(s) dans le graphe **do**
  $l \leftarrow$ plus petit label temporaire (découvert ou caché) par ordre lexicographique
  Passer l permanent
  // soit l le N^{eme} label du sommet i, on a :
  $i \leftarrow$ sommet de l
  $N \leftarrow$ numéro de label de l
  for chaque successeur j de i **do**
    $l' \leftarrow [(z^1(l) + c^1_{ij}, ..., z^p(l) + c^p_{ij}, z^{p+1}(l) \cdot q^1_{ij}, ..., z^{p+q}(l) \cdot q^q_{ij}, UpdateMinMax(l, i, j, M)), i, N]$
    ajouter l' au sommet j en tant que label temporaire découvert
    if l' est faiblement non-dominé sur la fonction max(min(.)) par un label de j **then**
      Cacher l'
    else if l' est dominé par un label de j **then**
      Supprimer l'
    else if l' domine un label L de j **then**
      Supprimer L
    else if un label L de j est faiblement non-dominé sur la fonction max(min(.)) **then**
      Cacher L
    end if
  end for
end while
backtracker sur les labels découverts de t à l'aide des pointeurs pour retrouver tous les chemins Pareto-optimaux

3.3.3 Preuve

Nous allons ici nous limiter à p fonctions somme et q fonctions produit par souci de simplicité. L'ajout d'une fonction de type max(min(.)) se fait comme décrit dans le papier de X.Gandibleux, F.Beugnies et S.Randriamasy [6]. Nous allons donc reprendre les arguments avancés dans le livre de M.Ehrgott [3] (et décrits dans la partie 3.1.2) qui restent valables pour les fonctions de type somme afin d'observer ce qu'il se passe pour les fonctions de type produit.

Montrons d'abord qu'un label permanent correspond forcément à un chemin Pareto-optimal. Le premier argument était le suivant : « *Soit l le label temporaire le plus petit par ordre lexicographique dans le graphe, r_l le chemin décrit par l (r_l peut être retrouvé par le backtracking sur les pointeurs) et m le sommet sur lequel est posé l. Selon l'algorithme, l va être passé permanent. Supposons maintenant que r_l n'est pas un chemin Pareto-optimal. Cela signifie qu'il existe un chemin $r'_l \neq r_l$ tel que $r'_l \in R_{s,m}$ et que $z^k(r'_l) \leq z^k(r_l) \ \ \forall k \in \{1, ..., p\}$ avec au moins un k pour lequel l'inégalité est stricte. Soit m' un sommet situé sur le chemin r'_l, et l' un label du chemin r'_l posé sur m', cela implique que $\exists\ r_{m',m} \in R_{m',m}$ tel que $z^k(l') + \sum_{(i,j)\in r_{m',m}} c^k_{i,j} \leq z^k(l)$. Les c_{ij} étant positifs, on a que $z^k(l') \leq z^k(l)$ avec au moins un objectif ayant l'inégalité stricte. Cela implique donc que l' est plus petit lexicographiquement que l, ce qui entre en contradiction avec le fait que l soit le plus petit label par ordre lexicographique.* ». On remarque que pour les fonctions produit, on aurait $z^k(l') + \prod_{(i,j)\in r_{m',m}} q_{ij} \geq z^k(l)$, ce qui là encore entre en contradiction avec le fait que l soit le plus petit label par ordre lexicographique. Donc tout label permanent correspond toujours à un chemin Pareto-optimal.

Montrons maintenant qu'un ensemble complet des solutions efficaces est restitué. Considérons dans un premier temps uniquement des q_{ij} strictement positifs. Dans ce cadre là, les arguments utilisés dans la preuve 3.1.2 restent valables et tous les chemins Pareto-optimaux seront trouvés, y compris ceux ayant des coûts similaires.

Considérons maintenant des coûts tels que $q_{ij} \in [0, 1]$. Précédemment, nous avons expliqué qu'un label l faiblement non-dominé sur des fonctions produit pouvait correspondre à un chemin non-dominé, mais nous avons choisi de les ignorer. C'est donc le seul cas où l'algorithme peut omettre un chemin. Or, dans ce cas là, on a nécessairement que exactement tous les coûts de l sont égaux à un autre label l' qui lui est Pareto-optimal. Si un seul coût est différent, soit nous ne somme pas dans le cas ici décrit, auquel cas les arguments précédemment explicités restent valables, soit r_l, qui sera omis dans tout les cas, n'est pas un chemin Pareto-optimal. Donc l'algorithme trouvera nécessairement tous les chemins Pareto-optimaux de coûts différents. Ainsi, un ensemble complet de solutions efficaces est restitué par l'algorithme.

3.4 Exemple

Soit le graphe de la figure 3 avec un objectif somme, un objectif produit et un objectif de type max(min(.)). Les coûts de chaque arc sont donnés par le vecteur qui est collé dessus. Sur la figure 3, l'algorithme a déjà été appliqué. Les labels sont représentés avec un fond de couleur. Les éléments en noir sont les $p + q + 1 + 2$ valeurs d'un label normal. Les éléments en rouge sont simplement de l'information supplémentaire pour le bon déroulement de l'exemple. Le détail de ces éléments ainsi que le code couleur sont donnés dans la légende en bas à gauche de la figure 3. On appellera par ailleurs TL la liste des labels temporaires.

Itération 1

Initialisation de l'algorithme : le label $[(0, 1, \infty), -, -]$ est créé sur le sommet s.

$TL = \{[(0, 1, \infty), -, -]\}$

Itération 2

Le plus petit label par ordre lexicographique est celui sur s : on le passe permanent. On propage les coûts :

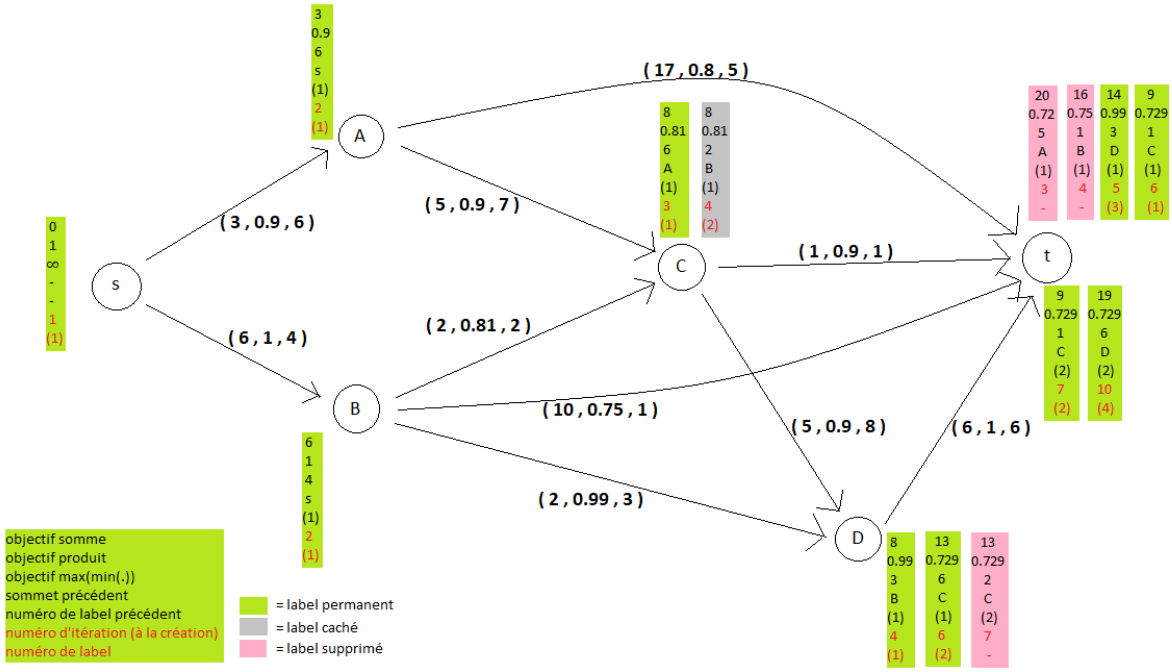

FIGURE 3 – exemple de l'algorithme 4 de résolution en $(1 - \Sigma \mid 1 - \Pi \mid 1 - M)$

— le label $[(3, 0.9, 6), s, 1]$ est créé sur A.
— le label $[(6, 1, 4), s, 1]$ est créé sur B.

$TL = \{[(3, 0.9, 6), s, 1] \quad , \quad [(6, 1, 4), s, 1]\}$

Itération 3

Le plus petit label par ordre lexicographique est $[(3, 0.9, 6), s, 1]$, sur le sommet A : on le passe permanent. On propage les coûts :
— le label $[(20, 0.72, 5), A, 1]$ est créé sur t.
— le label $[(8, 0.81, 6), A, 1]$ est créé sur C.

$TL = \{[(6, 1, 4), s, 1] \quad , \quad [(20, 0.72, 5), A, 1] \quad , \quad [(8, 0.81, 6), A, 1]\}$

Itération 4

Le plus petit label par ordre lexicographique est $[(6, 1, 4), s, 1]$, sur le sommet B : on le passe permanent. On propage les coûts :
— le label $[(8, 0.81, 2), B, 1]$ est créé sur le sommet C. On remarque qu'il est faiblement non-dominé sur la fonction max(min(.)) : on le passe caché.
— Le label $[(8, 0.99, 3), B, 1]$ est créé sur le sommet D.
— Le label $[(16, 0.75, 1), B, 1)$ est créé sur le sommet t. Il est non-dominé par le label $[(20, 0.72, 5), A, 1]$: on ne fait rien.

$TL = \{[(20, 0.72, 5), A, 1] \quad , \quad [(8, 0.81, 6), A, 1] \quad , [(8, 0.81, 2), B, 1] \quad , \quad [(8, 0.99, 3), B, 1] \quad , \quad [(16, 0.75, 1), B, 1)\}$

Itération 5

Le plus petit label par ordre lexicographique est $[(8, 0.99, 3), B, 1]$, sur le sommet D : on le passe permanent. On propage les coûts :
— le label $[(14, 0.99, 3), D, 1]$ est créé sur le sommet t. On remarque qu'il domine le label $[(16, 0.75, 1), B, 1)$ que l'on supprime.

$TL = \{[(20, 0.72, 5), A, 1] \quad , \quad [(8, 0.81, 6), A, 1] \quad , \quad [(8, 0.81, 2), B, 1] \quad , \quad [(14, 0.99, 3), D, 1]\}$

Itération 6

Le plus petit label par ordre lexicographique est $[(8, 0.81, 6), A, 1]$ sur le sommet C : on le passe permanent. On propage les coûts :

— le label $[(9, 0.729, 1), C, 1]$ est créé sur le sommet t. Il est non-dominé par les deux autres présents, on ne fait rien.

— le label $[(13, 0.729, 6), C, 1]$ est créé sur le sommet D. Il est non-dominé par le label $[(8, 0.99, 3), B, 1]$, on ne fait rien.

$TL = \{[(20, 0.72, 5), A, 1]\ ,\ [(8, 0.81, 2), B, 1]\ ,\ [(14, 0.99, 3), D, 1]\ ,\ [(9, 0.729, 1), C, 1]\ ,\ [(13, 0.729, 6), C, 1]\}$

Itération 7

Le plus petit label par ordre lexicographique est $[(8, 0.81, 2), B, 1]$, sur le sommet D : on le passe permanent. On propage les coûts :

— le label $[(9, 0.729, 1), C, 2]$ est créé sur le sommet t. Il est non-dominé par les autres labels présents, on ne fait rien.

— le label $[(13, 0.729, 2), C, 2]$ est créé sur le sommet D. On observe qu'il est dominé par le label permanent $[(8, 0.99, 3), B, 1]$. On supprime donc le label créé.

$TL = \{[(20, 0.72, 5), A, 1]\ ,\ [(14, 0.99, 3), D, 1]\ ,\ [(9, 0.729, 1), C, 1]\ ,\ [(13, 0.729, 6), C, 1]\ ,\ [(9, 0.729, 1), C, 2]\}$

Itération 8

Le plus petit label par ordre lexicographique est $[(9, 0.729, 1), C, 1]$, sur le sommet t : on le passe permanent. Ce sommet n'a aucun successeur, il n'y a donc pas de coûts à propager : l'itération s'arrête ici.

$TL = \{[(20, 0.72, 5), A, 1]\ ,\ [(14, 0.99, 3), D, 1]\ ,\ [(13, 0.729, 6), C, 1]\ ,\ [(9, 0.729, 1), C, 2]\}$

Itération 9

Le plus petit label par ordre lexicographique est $[(9, 0.729, 1), C, 2]$, sur le sommet t : on le passe permanent. Ce sommet n'a aucun successeur, il n'y a donc pas de coûts à propager : l'itération s'arrête ici.

$TL = \{[(20, 0.72, 5), A, 1]\ ,\ [(14, 0.99, 3), D, 1]\ ,\ [(13, 0.729, 6), C, 1]\}$

Itération 10

Le plus petit label par ordre lexicographique est $[(13, 0.729, 6), C, 1]$, sur le sommet D : on le passe permanent. On propage les coûts :

— le label $[(19, 0.729, 6), D, 2]$ est créé sur le sommet t. On observe qu'il domine le label $[(20, 0.72, 5), A, 1]$ que l'on supprime.

$TL = \{[(14, 0.99, 3), D, 1]\ ,\ [(19, 0.729, 6), D, 2]\}$

Itération 11

Le plus petit label par ordre lexicographique est $[(14, 0.99, 3), D, 1]$, sur le sommet t : on le passe permanent. Ce sommet n'a aucun successeur, il n'y a donc pas de coûts à propager : l'itération s'arrête ici.

$TL = \{[(19, 0.729, 6), D, 2]\}$

Itération 12

Le plus petit label par ordre lexicographique est $[(19, 0.729, 6), D, 2]$, sur le sommet t : on le passe permanent. Ce sommet n'a aucun successeur, il n'y a donc pas de coûts à propager : l'itération s'arrête ici.

$TL = \emptyset$

Il n'y a plus de label temporaire dans le graphe, l'algorithme s'arrête donc ici. Nous allons maintenant utiliser les pointeurs des labels pour reconstituer les chemins Pareto-optimaux.

Reconstitution des chemins

Le premier label permanent de t est $[(14, 0.99, 3), D, 1]$. Celui-ci nous mène au label permanent numéro 1 du sommet D qui est $[(8, 0.99, 3), B, 1]$. Ce dernier nous mène au label numéro 1 du sommet B qui est $[(6, 1, 4), s, 1]$, qui lui nous mène au label $[(0, 1, \infty), -, -]$. Ainsi, un premier chemin Pareto-optimal est $\{(s, B), (B, D), (D, t)\}$ de coûts $(14, 0.99, 3)$.

Le second label permanent de t est $[(9, 0.729, 1), C, 1]$. Celui-ci nous mène au label permanent numéro 1 du sommet C, qui est $[(8, 0.81, 6), A, 1]$. Cela nous mène au label numéro 1 du sommet A, qui nous mène au label 1 du sommet s. Ainsi, nous avons le chemin $\{(s, A), (A, C), (C, t)\}$ de coûts $(9, 0.729, 1)$.

Le troisième label permanent de t est $[(9, 0.729, 1), C, 2]$. Celui-ci nous mène au label permanent numéro 2 du sommet C, qui nous mène au label numéro 1 de B, qui nous mène au label 1 de s. Ainsi, nous avons le chemin $\{(s, B), (B, C), (C, t)\}$ de coûts $(9, 0.729, 1)$.

Le dernier label permanent de t est $[(19, 0.729, 6), D, 2]$ qui nous mène au label numéro 2 du sommet D, qui nous mène au label 1 de C, qui nous mène au label 1 de A, qui nous mène au label 1 de s. Ainsi, nous avons le chemin $\{(s, A), (A, C), (C, D), (D, t)\}$ de coûts $(19, 0.729, 6)$.

4 Implémentation de l'algorithme

L'implémentation de l'algorithme précédemment décrit a été réalisée en Julia, version 0.6. Nous allons ici décrire les différentes structures de données utilisées et nous détaillerons l'implémentation de certaines fonctions. De nombreux éléments qui vont être décrits reposent sur les objets "Vector" de Julia, qui sont des tableaux dotés de certaines opérations comme push!(), qui ajoutent un élément à la fin d'un Vector. Les Vectors sont indicés à partir de 1.

Attention, nous allons ici changer la signification de m : on aura que $m = 1$ s'il y a une fonction de type max(min(.)), $m = 0$ s'il n'y en a pas. De plus, nous allons introduire deux notations : B_i représentera le nombre de labels sur un sommet i et b_i sera le nombre de labels temporaires sur un label i.

Enfin, par abus de langage, nous appellerons « fonction » toutes les fonctions et procédures, qu'elles renvoient quelque chose ou non. Lorsque quelque chose est retourné, cela sera précisé à l'aide d'un « Return » dans l'algorithme et dans le texte explicatif qui l'accompagne. Les paramètres des fonctions sont donnés en haut des algorithmes.

4.1 Structures de données

4.1.1 graphe

Cette structure de données représente un graphe. Celle-ci est composée des éléments suivants :

- S : un Vector de sommets. Dans chaque case sera rangé un sommet du graphe. Chaque sommet se verra donc attribué un indice. Nous décrirons un sommet par la suite.
- C : un Vector de matrices $n \times n$ de taille p dont le k^{eme} élément représente la matrice de coût du k^{eme} objectif de type somme, pour $k \in \{1, ..., p\}$.
- Q : un Vector de matrices $n \times n$ de taille q dont le h^{eme} élément représente la matrice de coût du h^{eme} objectif de type produit, pour $h \in \{1, ..., q\}$.
- M : une matrice $n \times n$ qui représente la matrice de coût de l'objectif de type max(min(.)).
- *tempo* : un Vector de labels qui contient tous les labels temporaires du graphe, ordonnés par ordre lexicographique.
- *nbtempo* : un entier représentant le nombre de labels temporaires dans le graphe.

On appellera par la suite b la taille de la liste des labels temporaires dans le graphe.

4.1.2 sommet

Cette structure de données représentera un sommet d'un graphe. Celle-ci est composée des éléments suivants :

- *sucs* : un Vector d'entiers représentant les successeurs de ce sommet. Chaque élément représente l'indice d'un sommet.
- *tempo* : un Vector de labels représentant liste des labels temporaires d'un sommet.
- *nbtempo* : un entier représentant le nombre de labels temporaires sur un sommet.
- *perma* : un Vector de labels représentant liste des labels permanents d'un sommet.
- *nbperma* : un entier représentant le nombre de labels permanents sur un sommet.

On peut remarquer qu'un label temporaire est toujours rangé à deux endroits en même temps : dans la liste *tempo* d'un sommet et dans la liste *tempo* d'un graphe. Les deux se réfèrent à la même adresse en mémoire, ce qui a pour conséquence que lorsque l'on modifie un label dans une liste, il sera modifié dans l'autre. Cette propriété sera utile par la suite.

4.1.3 label

Cette structure de données représentera un label. Celle-ci est composée des éléments suivants :

— *val* : un Vector de float64 à $p + q + m$ éléments, qui représentera le vecteur de coûts du label.
— *prec* : un Vecteur d'entiers à 2 éléments, qui représentent les pointeurs du label. Un Vector $[2, 4]$ signifie que le label a été créé depuis le label permanent d'indice 2 du sommet d'indice 4.
— *som* : un entier qui représente l'indice du sommet sur lequel est posé le label.
— *aSupprimer* : un booléen qui vaut true si le label a été déterminé comme dominé par un autre label, false sinon. Nous redécrirons son utilité par la suite.
— *hidden* : un booléen qui vaut true si le label est caché, false s'il est découvert.

4.2 Fonctions de l'algorithme

4.2.1 AjoutTempo

Cette fonction permet d'ajouter un label dans la liste ordonnée par ordre lexicographique des labels temporaires d'un graphe. Dans l'implémentation, i représentera l'indice d'un label dans G.tempo et k représentera un indice d'objectif. L'algorithme correspondant est l'**Algorithm 5**.

Algorithm 5 AjoutTempo
G : graphe
lab : label que l'on ajoute

```
t ← taille de G.tempo
if t = 0 then
  push !(G.tempo,lab)
else
  i ← 1
  k ← 1
  while (k ≤ p + q + m) et (i ≤ t) do
    if k ≤ p then
      if G.tempo[i].val[k] < lab.val[k] then
        i ← i + 1
        k ← 1
      else if G.tempo[i].val[k] = lab.val[k] then
        k ← k + 1
      else
        k ← p + q + m + 1
      end if
    else
      if G.tempo[i].val[k] > lab.val[k] then
        i ← i + 1
        k ← 1
      else if G.tempo[i].val[k] = lab.val[k] then
        k ← k + 1
      else
        k ← p + q + m + 1
      end if
    end if
  end while
  ajouter le label lab à l'indice i de G.tempo
end if
```

La boucle while fonctionne de la manière suivante : tant que l'on n'a pas trouvé un moins bon label lexicographiquement parlant que *lab* ou que l'on n'est pas au bout de la liste, on continue à parcourir celle-ci. Il est par ailleurs nécessaire de séparer les comparaison lorsques $k \leq p$ et lorsque $k > p$ car les p premiers objectifs sont à minimiser et les suivants sont à maximiser. L'ajout du

label à l'indice i se fait très rapidement en Julia grâce à la fonction vcat() qui peut concaténer des Vectors.

Dans le pire des cas, tous les labels de la liste seront de valeur égale sauf pour le dernier objectif où *lab* aura la pire des valeurs. Ainsi, *lab* sera le plus grand label du graphe par ordre lexicographique et sera placé en dernier après avoir comparé tous les objectifs de chacun des labels temporaires du graphe. La complexité de la fonction est donc en $O((p+q+m)\cdot b)$.

4.2.2 TestDomi

Cette fonction nous permet d'effectuer le test de dominance de l'algorithme de Martins. Lorsque nous ajoutons un label l sur un sommet j, il va falloir regarder si parmi les B_j labels du sommet j il y en a qui dominent ce label l, auquel cas on le supprime et on arrête les comparaisons ; ou s'il y en a qui sont dominés par l, auquel cas on supprime ceux qui le sont. Pour cela, nous allons parcourir les B_j labels jusqu'à tomber sur un de ces deux cas.

Dans l'algorithme, il sera nécessaire de supprimer des labels. Il est possible de supprimer l'élément i d'un Vector V à l'aide de la fonction deleteat !(V,i) de Julia. Par ailleurs, lorsque l'on supprime un label de la liste des labels temporaires d'un sommet, il est nécessaire de le supprimer dans la liste tempo du graphe. On notifiera cela à l'aide du champs *aSupprimer* d'un label. Cette opération est décrite dans **Algorithm 6**.

Algorithm 6 SupprimerLabel

G : graphe
j : indice de sommet
nlab : indice du label à supprimer

G.S[j].tempo[nlab].aSupprimer ← true
deleteat !(G.S[j].tempo,nlab)
G.S[j].nbtempo ← G.S[j].nbtempo -1
G.nbtempo ← G.nbtempo - 1

Cette fonction réalise toujours le même nombre d'opérations : elle est en $O(1)$ (à supposer que la fonction deleteat !() est en $O(1)$ aussi).

Nous allons maintenant décrire l'algorithme utilisé pour regarder si un label z_1 en domine un autre z_2. Si on a $z_1 >_d z_2$, la fonction retournera un booléen domi qui vaudra true, sinon elle retournera false. Nous allons partir du principe que $z_1 >_d z_2$, et donc domi est initialisé à true. Dès qu'une valeur meilleure pour z_2 que pour z_1 est trouvée, on s'arrête. Sinon, on continue à chercher. Il est aussi nécessaire de regarder si z_1 n'est pas égal à z_2 pour tous les objectifs, auquel cas z_2 n'est pas dominé. On comptera donc le nombre d'objectifs égaux dans l'opération. On utilisera aussi ce compteur pour vérifier si z_2 n'est pas faiblement dominé par z_1 sur la fonction $\max\min(.)$). Ces éléments sont effectués dans la fonction *Domine*, décrite dans **Algorithm 7**.

Cette fonction compare dans le pire des cas tous les objectifs des deux labels. Ainsi, elle est en $O(p+q+m)$. Dans le test de dominance, on applique l'algorithme 7 entre chacun des labels déjà présents sur le sommet et le nouveau label créé pour vérifier s'il n'est pas dominé. Si on conclut que le nouveau label est dominé, on arrête le test : celui-ci est terminé, on ne garde pas le label. Si ce n'est pas le cas, il faut réappliquer l'algorithme 7 entre le nouveau label créé et chacun des labels temporaires présents sur le sommet pour vérifier si le nouveau label n'en domine pas un. Si l'un d'entre eux est dominé, on appelle *SupprimerLabel*. On ne regarde pas si un label permanent est dominé car il est nécessairement non-dominé (Preuve 3.3.3). Nous allons regrouper tous ces éléments dans une fonction *TestDomi* qui réalise le test de dominance. Celle-ci renverra un booléen : true si le nouveau label a été supprimé, false sinon.

Ainsi, le test de dominance appelle $B_j + b_j$ fois la fonction Domine qui est en $O(p+q+m)$. Or, on a que $B_j + b_j \leq 2 \cdot B_j = O(B_j)$. De plus, *SupprimerLabel* étant en $O(1)$, on la considère ici comme négligeable. On a donc que le test de dominance d'un label sur un sommet se réalise en

Algorithm 7 Domine

z_1 : label
z_2 : label

$k \leftarrow 1$
domi $\leftarrow$ true
cptEgaux $\leftarrow 0$
while $(k \leq p)$ et (domi) **do**
  if z_1.val[k] $>$ z_2.val[k] **then**
    domi $\leftarrow$ false
  else if z_1.val[k] $=$ z_2.val[k] **then**
    cptEgaux $\leftarrow$ cptEgaux $+1$
  end if
  $k \leftarrow k + 1$
end while
while $(k \leq p + q)$ et (domi) **do**
  if z_1.val[k] $<$ z_2.val[k] **then**
    domi $\leftarrow$ false
  else if z_1.val[k] $=$ z_2.val[k] **then**
    cptEgaux $\leftarrow$ cptEgaux $+1$
  end if
  $k \leftarrow k + 1$
end while
if $(m = 1)$ et (domi) **then**
  if z_1.val[k] $<$ z_2.val[k] **then**
    domi $\leftarrow$ false
  else if z_1.val[k] $=$ z_2.val[k] **then**
    cptEgaux $\leftarrow$ cptEgaux $+1$
  else if cptEgaux $= p + q$ **then**
    z_2.hidden $\leftarrow$ true
    domi $\leftarrow$ false
  end if
end if
if cptEgaux $= p + q + m$ **then**
  domi $\leftarrow$ false
end if
return domi

$O((p + q + m) \cdot B_j)$.

4.2.3 Propagation des coûts

Comme décrit dans l'algorithme 4, dès qu'un label est passé permanent, il faut créer de nouveaux labels temporaires sur chacun des successeurs du sommet concerné puis réaliser le test de dominance. Par souci de simplicité, « *coûts de l'algo 4* » se référera aux $p + q + m$ premiers éléments du label de la ligne « $l' \leftarrow [(z^1(l) + c^1_{ij}\ ,\ ...\ ,\ z^p(l) + c^p_{ij}\ ,\ z^{p+1}(l) \cdot q^1_{ij}\ ,\ ...\ ,\ z^{p+q}(l) \cdot q^q_{ij}\ ,\ UpdateMinMax(l, i, j, M))\ ,\ i\ ,\ N]$ » de l'algorithme 4. De plus, en Julia, l'indice *end* d'un tableau correspond au dernier élément de celui-ci. Nous utiliserons cela pour aller chercher le dernier label permanent créé sur un sommet. Nous réaliserons cela à l'aide de **Algorithm 8**.

Soit d_i le nombre de successeurs d'un sommet i. Pour chaque successeur, on réalise dans le pire des cas $(p + q + m) \cdot B_i + (p + q + m) \cdot b$ opérations (test de dominance et ajout à la liste tempo du graphe). Nous avons donc une propagation des coûts qui est en $O(d_i \cdot (p + q + m) \cdot (B_i + b))$.

Algorithm 8 Propagation

G : graphe
i : indice de sommet

```
l ← G.S[i].perma[end]
for j ∈ G.S[i].sucs do
   nvlab ← new label(coûts de l'algo 4,[i,G.S[i].nbperma],j,false,false)
   push !(G.S[j].tempo,nvlab)
   G.S[j].nbtempo ← G.S[j].nbtempo +1
   if ¬(TestDomi(G,nvlab,j)) then
      AjoutTempo(G,nvlab)
   end if
end for
```

4.2.4 Récupération du plus petit label par ordre lexicographique

Il reste maintenant un dernier élément à éclaircir : comment récupérer le plus petit label par ordre lexicographique ? Nous allons ici utiliser la liste ordonnée des labels temporaires par ordre lexicographique *tempo* du graphe. En effet, le premier élément de cette liste correspond à priori au plus petit label par ordre lexicographique. Il restera juste à vérifier si celui-ci n'a pas été précédemment supprimé (grâce au champs *aSupprimer* du label), auquel cas il ne peut être accepté comme plus petit label, car il n'est plus censé exister. On se rabattra alors sur le second où on réalisera la même vérification. Au final, on finira par retrouver le plus petit label temporaire non supprimé.

Dans le pire des cas, il faudra parcourir $b - 1$ labels supprimés dans la liste avant de tomber sur le dernier qui lui n'a pas été supprimé. La fonction est donc en $O(b)$.

5 Utilisation d'une structure de données pour le test de dominance dans l'algorithme de Martins

5.1 Motivations

On sait qu'il y a parfois un grand nombre de chemins Pareto-optimaux dans un graphe. Cela implique qu'à la création d'un label sur un sommet, il faut éventuellement réaliser un grand nombre de tests de dominance. En effet, dans l'implémentation précédemment réalisée, nous comparons le nouveau label créé à tous les autres labels déjà présents sur le sommet sauf dans le cas où il est dominé, auquel cas on s'arrête dans le test de dominance. Soit B_i le nombre de labels présents sur un sommet i, notre test de dominance est en $O((p+q+m)B_i)$. Ainsi, si B_i est très grand, pour un ou plusieurs sommets, cela risque d'influer grandement sur le temps d'exécution de l'algorithme. L'idée est donc d'utiliser une structure de données sur chaque sommet d'un graphe $G(S, A, C, Q, M)$ afin de diminuer la complexité en fonction de B_i de la résolution. Pour cela, nous allons essayer d'introduire des ND-trees dans l'algorithme.

5.2 Présentation du ND-tree

Le ND-tree est une structure de données proposée par Andrzej Jaszkiewicz et Thibaut Lust pour réaliser des tests de dominance plus efficacement [1]. Tout ce qui est ici expliqué est tiré de leurs travaux. L'idée est de diviser l'espace des objectifs en hyperrectangles. En effet, ces derniers présentent des propriétés intéressantes : nous allons les présenter avec deux objectifs à minimiser (nous continuerons à parler d'hyperrectangles plutôt que de rectangles pour rester général), sans perte de généralité.

Soit Z_i un sous ensemble de points de l'espace des objectifs, l'hyperrectangle qui les regroupe sera déterminé à l'aide d'une estimation des points nadir et idéal de Z_i. Nous avons donc la configuration décrite par la Figure 1.

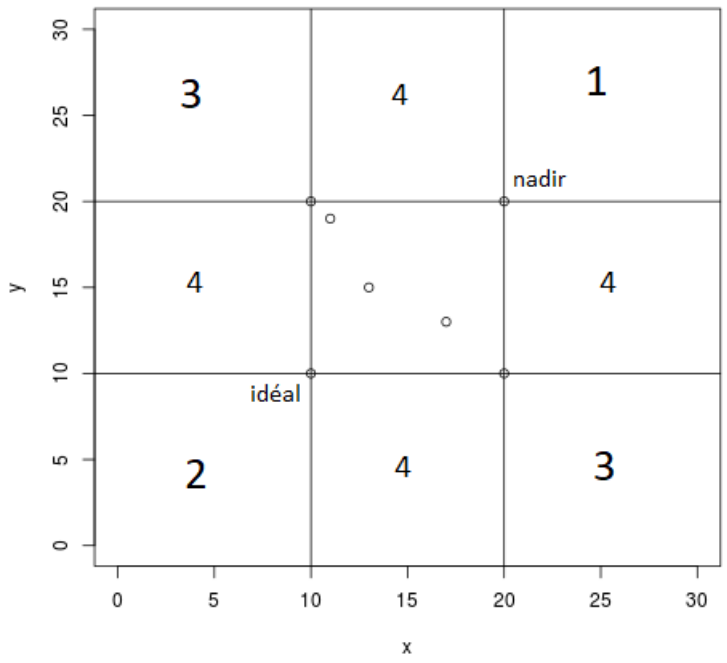

FIGURE 4 – Hyperrectangle et ses zones

Admettons qu'un nouveau point arrive pour réaliser le test de dominance. Si celui-ci est dans

la zone 1, on remarque qu'il est dominé par le point nadir de l'hyperrectangle. Le point sera alors grâce à la déduction 1 et à la transitivité de la relation de dominance dominé par tous les points contenus dans l'hyperrectangle. Si le nouveau point est dans la zone 2, on observe qu'il domine le point idéal de l'hyperrectangle, et donc par la déduction 2 et par transitivité, il domine tous les points contenus dans l'hyperrectangle. Si le nouveau point est dans région 3, on remarque qu'il est meilleur que le point idéal (et donc que tout les autres points de Z_i) sur un objectif et qu'il est moins bon que le nadir (et donc que tout les autres points de Z_i) sur un objectif. Ainsi, s'il est non-dominé par le nadir et par l'idéal, il est non-dominé par tous les points de l'hyperrectangle. Si le point n'est dans aucun de ces trois cas, il faudra alors le comparer à chaque point de l'hyperrectangle pour savoir s'il est dominé ou non. Si un point non-dominé est trouvé dans la région 3 ou dans la région 4, nous allons l'ajouter à l'hyperrectangle en élargissant celui-ci suffisamment pour qu'il englobe le nouveau point. Pour cela, il suffit de mettre à jour les points nadir et idéal (figure 2). Il est important de noter que l'on ne réduira jamais l'hyperrectangle car ceci est trop coûteux : il faudrait regarder les coûts de tous les points présents pour mettre à jour correctement les points idéal et nadir. De ce fait, nous travaillons avec une estimation de ces points pour chaque hyperrectangle. Par exemple, sur la figure 2, le points nadir correspondant aux points non-dominés de l'hyperrectangle devrait se situer en $(19, 20)$, mais comme nous sommes sur un cas de rétrécissement d'hyperrectangle, celui-ci restera en $(20, 20)$. Par abus de langage, nous parlerons par la suite de points nadir et idéal pour parler de l'estimation des points nadir et idéal.

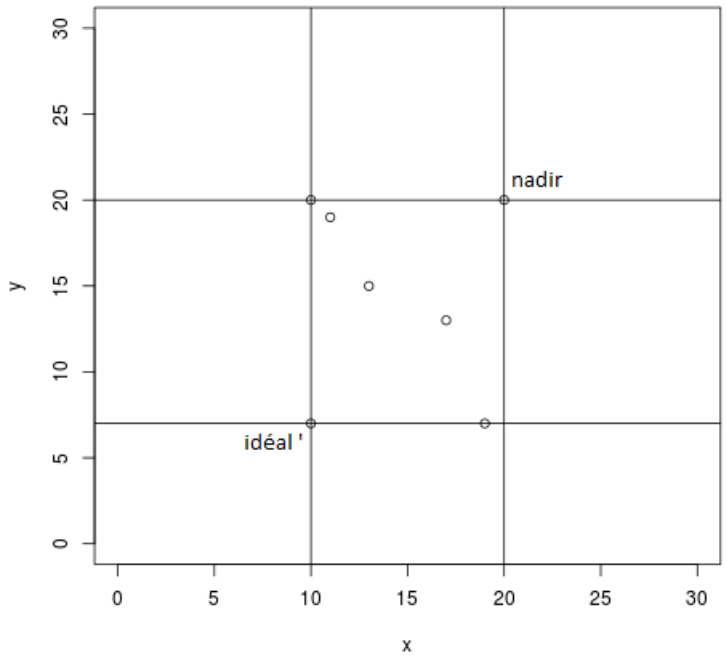

FIGURE 5 – Exemple de mise à jour du nadir ou de l'idéal

Si le nombre de points présents dans l'hyperrectangle est supérieur à un certain seuil σ, nous allons le diviser en plusieurs sous-hyperrectangles. La procédure à suivre pour le test de dominance sera alors la suivante :

— regarder si le point est dans une des régions 1, 2 ou 3. Si oui, conclure, sinon continuer,
— regarder pour chaque sous-hyperrectangle si on est dans une des régions 1, 2 ou 3. Si oui, conclure, sinon continuer (si certains hyperrectangles ont été à leur tour divisés), etc.

La structure de données retenue pour réaliser tous ces tests de dominance est un arbre où chaque noeud correspond à un hyperrectangle et chaque feuille contient au plus σ éléments. Pour un noeud donné, ses fils représentent les sous-hyperrectangles créés lorsqu'il a été divisé et il est un sous-hyperrectangle de son père. La racine est donc l'hyperrectangle qui contient tous les autres hyperrectangles. Ainsi, chaque noeud se verra attribué deux valeurs : le point idéal et le point nadir

correspondants. Un nouveau point arrivant pour le test de dominance réalisera les comparaisons nécessaires (zones 1, zone 2, zone 3, autre ?). S'il n'est pas possible de conclure, on va réaliser les même comparaisons avec les fils du noeuds et ainsi de suite. Si à un moment, on arrive à la conclusion qu'un hyperrectangle est dominé (ex : point dans la zone 1), on supprime simplement le noeud correspondant et tout l'arbre qui en découle.

L'algorithme a donc deux paramètres à régler :

— le nombre maximal de points dans une feuille (ce que l'on a appelé précédemment σ).
— le nombre de fils a créer à chaque fois que l'on divise un noeud (ou hyperrectangle), que l'on appelera δ.

5.3 Complexité

Dans l'article présentant le ND-tree comme structure pour le test de dominance [1], les auteurs prouvent que la complexité au pire reste linéaire en la taille de l'archive. Dans notre contexte, le ND-tree est donc en $O((p+q+m) \cdot B_i)$. En revanche, ils montrent aussi qu'il est possible d'avoir une complexité en moyenne en $\Theta((p+q+m) \cdot B_i^c)$, où $c < 1$. La complexité moyenne étant difficile à étudier pour cet algorithme, elle a été montrée uniquement avec $\delta = 2$. De plus, les expérimentations numériques menées semblent démontrer une certaine efficacité de la structure de données lorsqu'un très grand nombre de comparaisons est effectué. Nous mènerons à la fin une expérimentation numérique pour déterminer si l'utilisation du ND-tree permet empiriquement d'avoir un gain en temps d'exécution par rapport à la première implémentation ici proposée (que nous appelerons implémentation linéaire, du fait de sa complexité).

5.4 Implémentation et introduction dans l'algorithme de Martins

Nous allons dans cette partie conserver certaines structures de données (label, graphe) et fonctions (AjoutTempo, Récupération du plus petit label par ordre lexicographique, Propagation des coûts, Domine, SupprimerLabel) utilisées dans l'implémentation précédente. Certaines structures de données vont êtres modifiées et le test de dominance va être entièrement revu.

Les algorithmes décrits dans cette section sont ceux fournis dans le papier de Andrzej Jaszkiewicz et Thibaut Lust [1]. Nous les compléterons avec une présentation de notre implémentation de certains points décrits moins précisément dans leur article, comme la suppression d'un arbre ou la séparation d'un noeuds en plusieurs fils. Une autre différence est que nous ne manipulerons pas des points mais des labels. Nous utiliserons le champs *val* des labels pour effectuer les comparaisons nécessaires, celui-ci correspondant à un point de l'espace des objectifs.

5.4.1 Structures de données

Une nouvelle structure de données va être nécessaire pour créer les noeuds de l'arbre, qu'ils soient une feuille ou non. Celle-ci s'appellera *noeud*. Sur le même schéma que pour les sommets, l'arbre sera représenté sous forme d'un Vector de noeuds. Chaque noeud aura un indice unique (son indice dans le Vector) et il sera possible d'y accéder via celui-ci.

La structure *noeud* contiendra les éléments suivants :

— *parent* : un entier qui correspond à l'indice du noeud parent de celui-ci. Si le noeud est la racine, cet élément prendra la valeur 0.
— *enfants* : un Vector d'entiers qui contient les indices des fils du noeud. Si le noeud est une feuille, celui-ci est un Vector vide (0 éléments).
— *nadir* : un Vector de réels à $p+q+m$ éléments qui représente le point nadir de l'hyperrectangle correspondant au noeud.
— *ideal* : un Vector de réels à $p+q+m$ éléments qui représente le point idéal de l'hyperrectangle correspondant au noeud.

— *list* : un Vector de labels, qui contient la liste des labels présents dans l'hyperrectangle. Si le noeud n'est pas une feuille, cet élément est vide : les labels ne seront stockés que dans les feuilles.
— *tList* : un entier représentant la taille du champs *list* du noeud.
— *leaf* : un booléen tel qu'il vaut true si le noeud est une feuille, false sinon.

La structure de données *sommet* sera modifiée. Un ND-tree sera placé sur chaque sommet afin de réaliser le test de dominance. On a donc :

— *sucs* : inchangé
— *ndtree* : un Vector de noeuds représentant le ND-tree qui sera utilisé pour le test de dominance.
— *perma* : inchangé
— *nbperma* : inchangé

Le Vector contenant les labels temporaires a été supprimé et remplacé par le ND-tree car celui-ci ne nous servait que dans le test de dominance. Le Vector des labels permanents est conservé afin de restituer rapidement les chemins Pareto-optimaux à la fin.

5.4.2 Fonctions du test de dominance

Dans cette partie de l'algorithme, nous allons regarder si un label est dominé ou s'il domine d'autres labels. Nous allons utiliser pour cela les propriétés des régions 1, 2 et 3. Cela sera réalisé dans la fonction *UpdateNode* (**Algorithm 9**), celle-ci étant directement tirée du papier d'origine [1]. Elle renverra à un booléen *maj* qui vaut true si le label est non-dominé, false sinon.

Il a été nécessaire de modifier légèrement la fonction *Domine* pour qu'elle ne compare non plus des labels mais des points de l'espace des objectifs, étant donné que les points idéal et nadir ne sont pas des labels mais des points de l'espace des objectifs.

Il y a aussi un mécanisme ajouté dans la fonction UpdateNode qui fait que si tous les labels d'une feuille sont supprimés, alors on supprime ce noeud. De plus, s'il ne reste qu'un seul enfant nd' à un noeud nd, alors on remplace nd par nd'.

Cette fonction nécessite parfois de supprimer un sous-arbre. Il faut bien penser à aller marquer tous les labels du sous-arbre supprimé en tant que « supprimé » afin de ne pas récupérer des labels inexistants lorsque l'on cherche le plus petit label par ordre lexicographique. Nous le faisons dans la fonction *SupprimerArbre* (**Algorithm 10**).

La fonction *SupprimerArbre* est en $O(B_i)$ car dans le pire des cas, il faudra supprimer tous les labels du sommet. Il a été montré que la fonction UpdateNode est aussi en $O(B_i)$ [1].

5.4.3 Insertion d'un label dans le ND-tree

Si la conclusion de l'algorithme 9 est que le label est non-dominé, c'est à dire que true est retourné, il faut placer ce nouveau label dans l'arbre pour les futures comparaisons. À chaque niveau de l'arbre, nous allons choisir de descendre dans le fils le plus proche du label, jusqu'à tomber sur une feuille. La mesure de proximité utilisée pour comparer des noeuds est la distance euclidienne entre le label et le milieu de l'hyperrectangle correspondant au noeud. Cette opération sera réalisée dans la fonction *FindClosest* (**Algorithm 11**). Une illustration numérique est donnée dans l'exemple 1.

Exemple 1. Sur la figure 6, le label de valeurs $(14, 3, 8)$ arrive sur un noeud et doit être envoyé dans un des deux fils présents. On utilise *FindClosest* pour savoir dans lequel envoyer. Le milieu du premier fils est $(\frac{2+10}{2}, \frac{8+16}{2}, \frac{8+16}{2}) = (6, 12, 12)$. Le milieu du second est $(\frac{6+12}{2}, \frac{4+6}{2}, \frac{2+8}{2}) = (9, 5, 5)$. Une fois ces deux éléments obtenus, il faut calculer la distance euclidienne entre le label et le milieu de premier fils, qui est : $\sqrt{(14-6)^2 + (3-12)^2 + (8-12)^2} = \sqrt{161} \simeq 12.69$. La distance euclidienne entre le milieu du second fils et le label est : $\sqrt{(14-9)^2 + (3-5)^2 + (8-5)^2} = \sqrt{38} \simeq 6.16$. On observe que le label est plus proche du milieu du second fils que du premier, il sera donc envoyé

Algorithm 9 UpdateNode

Y : un ND-tree, sous forme d'un tableau de noeuds (comme décrit auparavant).
nd : un entier représentant l'indice du noeud courant.
lab : le label sur lequel on effectue le test de dominance.

```
if Domine(Y[nd].nadir,lab.val) then
  // le nadir domine le label, région 2 => lab est dominé
  maj ← false
else if Domine(lab.val,Y[nd].ideal) then
  // lab domine le point ideal, région 1 => hyperrectangle dominé
  SupprimerArbre(Y,nd)
else if Domine(Y[nd].ideal,lab.val) OU Domine(lab.val,Y[nd].nadir) then
  // région où il faut faire le test sur chaque label
  if Y[nd].leaf = true then
    for z ∈ Y[nd].list do
      if Domine(z.val,lab.val) then
        // lab dominé par un des labels de l'hyperrectangle
        maj ← false
      else if Domine(lab.val,z.val) then
        // lab domine un des labels de l'hyperrectangle, on le supprime
        deleteat !(Y[nd].list,z)
      end if
    end for
  else
    // Si le noeud nd n'est pas une feuille, on regarde ses sous-hyperrectangles
    for child ∈ Y[nd].enfants do
      if ¬(UpdateNode(Y,child,lab) then
        // Dans un des sous-hyperrectangles on a conclu que lab était dominé
        maj ← false
      end if
    end for
  end if
else
  // lab est dans la région 3 et est donc non-dominé dans cet hyperrectangle
end if
return maj
```

Algorithm 10 SupprimerArbre(Y,nd)

Y : un ND-tree, sous forme d'un tableau de noeuds (comme décrit auparavant).
nd : un entier représentant l'indice du noeud courant.

```
if Y[nd].leaf = true then
  for i allant de 1 à Y[nd].tList do
    Y[nd].list[i].aSupprimer = true
  end for
else
  for enf ∈ Y[nd].enfants do
    SupprimerArbre(Y,enf)
  end for
end if
```

sur le second fils.

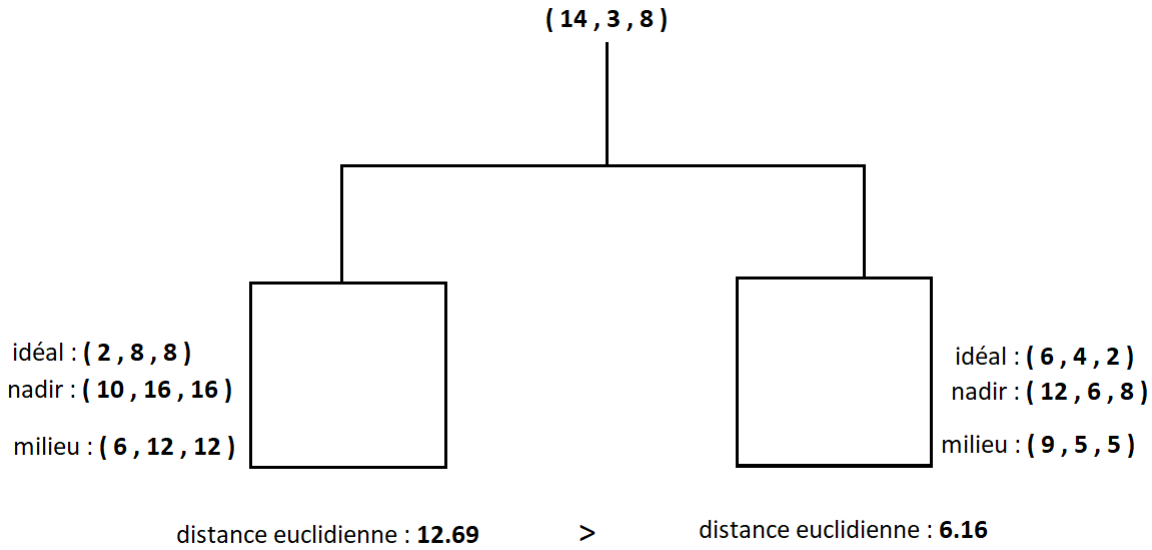

FIGURE 6 – Exemple de fonctionnement de *FindClosest*

Algorithm 11 FindClosest

Y : un ND-tree, sous forme d'un tableau de noeuds (comme décrit auparavant).
candidats : un Vector d'entiers contenant les indices des noeuds parmi lesquels on cherche le plus proche.
lab : le label que l'on souhaite ajouter dans l'arbre.

```
distMin ← +∞
for c ∈ candidats do
  milieu = (0, ..., 0)
  for i allant de 1 à p + q + m do
    milieu[i] ← (Y[c].nadir[i] + Y[c].ideal[i]) / 2
  end for
  dist ← DistanceEuclidienne(lab.val,milieu)
  if distMin > dist then
    distMin ← dist
    bestCandidat ← c
  end if
end for
return bestCandidat
```

L'algorithme 11 réalise $p + q + m$ opérations pour le calcul du milieu et $p + q + m$ opérations pour la distance euclidienne. De plus, le Vector de candidats comme nous le verrons par la suite correspondra toujours à un ensemble de fils d'un noeud. Il n'y en aura donc jamais plus de δ, mais il peut y en avoir moins si un ou plusieurs ont été supprimés dans *UpdateNode*. L'algorithme 11 est donc en $O((p + q + m) \cdot \delta)$. Cependant, δ est un paramètre constant de l'algorithme. On est donc en $O(p + q + m)$.

Une fois que l'on tombe sur une feuille en descendant dans l'arbre, on va y placer le label. Si celui-ci est dans la région 4 de l'hyperrectangle de la feuille, il est nécessaire de mettre à jour le point nadir et le point idéal. Il est alors nécessaire, si besoin, de mettre à jour les points des hyperrectangles des noeuds supérieurs. Nous ferons cela dans la fonction *UpdateIdealNadir* (**Algorithm 12**). Nous savons qu'un sous-hyperrectangle h est contenu dans l'hyperrectangle supérieur H, c'est-à-dire que h est un des sous-hyperrectangles obtenus de la division de H. Ainsi, si les points idéal et nadir de h ne sont pas mis à jour, ceux de H ne le seront pas non plus. Nous allons donc utiliser un booléen *changement* qui captera si une mise à jour des points a été faite et

qui éventuellement nous permettra d'éviter de réaliser des récursions inutiles et ainsi de remonter tout l'arbre. S'il est tout de même nécessaire de remonter tout l'arbre, on s'arrête lorsque l'on est à la racine.

Algorithm 12 UpdateIdealNadir

Y : un ND-tree, sous forme d'un tableau de noeuds (comme décrit auparavant).
nd : un entier représentant l'indice du noeud courant.
z : le label ajouté.

```
changement ← false
// fonctions somme
for i allant de 1 à p do
  if z.val[i] > Y[nd].nadir[i] then
    Y[nd].nadir[i] = z.val[i]
    changement ← true
  else if z.val[i] < Y[nd].ideal[i] then
    Y[nd].ideal[i] = z.val[i]
    changement ← true
  end if
end for
// autres fonctions
for i allant de p + 1 à p + q + m do
  if z.val[i] < Y[nd].nadir[i] then
    Y[nd].nadir[i] = z.val[i]
    changement ← true
  else if z.val[i] > Y[nd].ideal[i] then
    Y[nd].ideal[i] = z.val[i]
    changement ← true
  end if
end for
if changement ET nd ≠ 1 then
  UpdateIdealNadir(Y,Y[nd].parent,z)
end if
```

Lorsque le seuil σ de labels dans une feuille est dépassé, nous allons la diviser en δ feuilles auxquelles nous attribuerons les $\sigma + 1$ labels. La stratégie est la suivante : nous déterminons le label avec la plus grand distance euclidienne moyenne par rapport aux autres et nous le mettons dans une première feuille. Ensuite, nous choisissons le label l qui a la plus grand distance euclidienne moyenne par rapport aux d labels déjà placés dans une feuille et nous plaçons l dans une $d+1^{eme}$ feuille, pour tout $d \in \{1, ..., \delta - 1\}$. La fonction *FurthestLabel* (**Algorithm 13**) permettra de déterminer les labels les plus éloignées en utilisant ce critère. Une fois que les δ feuilles ont été créées, il faut assigner chacun des $\sigma - \delta$ labels restants à sa feuille la plus proche (en utilisant la fonction *FindClosest*). Il faut là aussi mettre à jour le point idéal et le point nadir lorsqu'un label est ajouté à une feuille (cas où le point est dans la région 4, mais aussi la région 3). Toutes ces opérations seront réunies dans la fonction *Split* (**Algorithm 14**). Nous y utiliserons par ailleurs la fonction *length()* de Julia qui renvoie la taille d'un Vector. Cela nous sera utile pour savoir combien d'enfants ont été créés, mais aussi pour connaître l'indice final d'un Vector. Un exemple de séparation d'un noeud est donné dans l'exemple 2.

Exemple 2. Prenons un exemple de fonctionnement de la séparation d'un noeud en δ fils avec $\sigma = 2$ et $\delta = 2$. La situation initiale est décrite dans la figure 7 : les deux labels $l_1 = (10, 12, 9)$ et $l_2 = (20, 8, 4)$ sont présents dans le noeuds et le label $l_3 = (14, 5, 12)$ doit être inséré ici. On observe que l'on a 3 labels dans le noeud, ce qui est supérieur à σ, donc on va séparer le noeuds en 2 fils. La première étape consiste à trouver le label le plus éloigné des autres. Après avoir calculé la distance euclidienne entre chaque, nous avons la table 1 qui est la table des distances. La valeur $d_m(l)$ représente la valeur moyenne de la distance euclidienne avec les autres labels. Ainsi, on a $d_m(l_1) = \frac{0+11.87+8.60}{3} \simeq 6.82$, $d_m(l_2) = \frac{11.87+0+10.44}{3} \simeq 7.44$ et $d_m(l_3) = \frac{8.60+10.44+0}{3} \simeq 6.35$. Le label le plus éloigné des autres est donc l_2 : un premier noeud est créé avec l_2 dedans.

distance	l_1	l_2	l_3
l_1	0	11.87	8.60
l_2	11.87	0	10.44
l_3	8.60	10.44	0

TABLE 1 – table des distances dans l'exemple de séparation

N'ayant qu'un seul label, les points nadir et idéal de la feuille valent tout deux $l_2 = (20, 8, 4)$ et le milieu du noeud vaut aussi par conséquent $(20, 8, 4)$. Il faut maintenant chercher le label le plus éloigné du premier fils, ce qui revient à chercher le noeud le plus éloigné de l_2. On a $d(l_2, l_1) = 11.87 > 10.44 = d(l_2, l_3)$: on créer donc un second noeud avec le label l_1. Pour les mêmes raisons que précédemment, ses points nadir, idéaux ainsi que son milieu valent $l_1 = (10, 12, 9)$. Les deux fils ont été créés, il faut maintenant ajouter le label restant à la bonne feuille, à l'aide de la fonction *FindClosest*. On a $d(l_3, (20, 8, 4)) = 10.44 > 8.60 = d(l_3, (10, 12, 9))$, donc le noeud le plus proche est le second : on y ajoute l_3. Une fois ceci fait, on met à jour les valeurs du point nadir et du point idéal. La situation finale est décrite dans la figure 7.

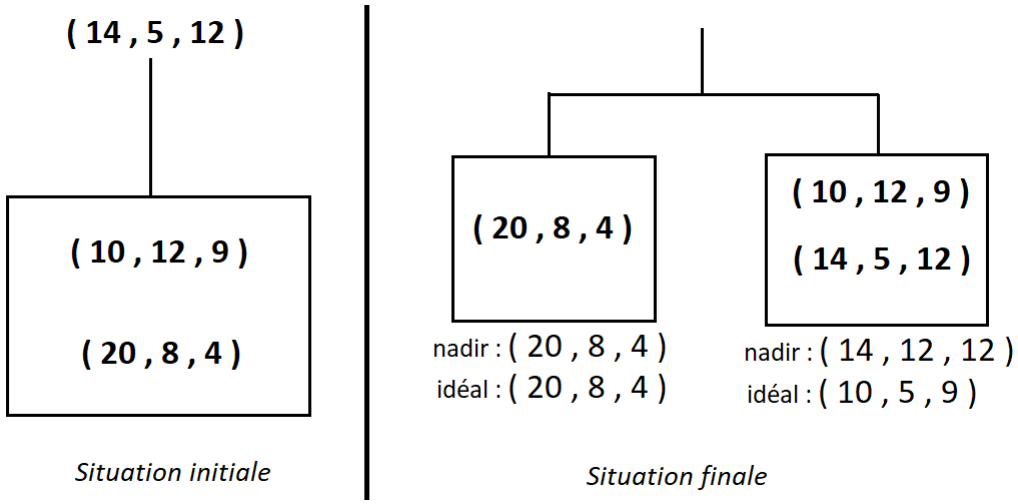

FIGURE 7 – Exemple de séparation d'un noeud

Algorithm 13 FurthestLabel

Candidats : un Vector contenant les labels parmi lesquels on recherche le plus éloigné
lc : un Vector contenant les labels sur lesquels on compare l'éloignement
$t_{Candidats}$: un entier représentant la taille de *Candidats*
t_{lc} : un entier représentant la taille de *lc*

```
distMoyMax ← 0
for i allant de 1 à t_Candidats do
  distAvg ← 0
  for j allant de 1 à t_lc do
    distAvg ← distAvg + DistanceEuclidienne(Candidats[i].val,lc[j].val)
  end for
  distAvg ← distAvg /t_lc
  if distAvg ≥ distMoyMax then
    bestC ← Candidats[i]
    distMoyMax ← distAvg
  end if
end for
return bestC
```

On remarque dans l'algorithme *Split* que la fonction FurthestLabel est appelée soit avec deux listes de taille σ (à l'initialisation), soit avec une liste de taille σ et une liste de taille δ. Or on

Algorithm 14 Split

Y : un ND-tree, sous forme d'un tableau de noeuds (comme décrit auparavant)
nd : un entier représentant l'indice du noeud que l'on divise

```
z ← FurthestLabel(Y[nd].list,Y[nd].list,Y[nd].tList,Y[nd].tList
// création d'un nouveau noeud, ajouté à la fin de Y
push !(Y, new noeud(nd,[],z.val,z.val,[z],1,true))
// ajout du nouveau noeud en tant qu'enfant de celui qu'on divise
push !(Y[nd].enfants,length(Y))
// Suppression du label dans la liste de nd
deleteat !(Y[nd].list,z)
Y[nd].tList ← Y[nd].tList - 1
while length(Y[nd].enfants) < δ do
   // On répète les mêmes opérations δ fois
   z ← FurthestLabel(Y[nd].list,labels des nouveaux noeuds,Y[nd].list,length(Y[nd].enfants))
   push !(Y, new noeud(nd,[],z.val,z.val,[z],1,true))
   push !(Y[nd].enfants,length(Y))
   deleteat !(Y[nd].list,z)
   Y[nd].tList ← Y[nd].tList - 1
end while
// On assigne chaque label à sa feuille la plus proche
while Y[nd].tList ≠ 0 do
   z ← Y[nd].list[1]
   c ← FindClosest(Y,Y[nd].enfants,z)
   push !(Y[c].list,z)
   Y[c].tList ← Y[c].tList +1
   deleteat !(Y[nd].list,z)
   Y[nd].tList ← Y[nd].tList −1
   UpdateIdealNadir(Y,c,z)
end while
Y[nd].leaf ← false
```

a nécessairement que $\delta \leq \sigma$, sinon nous n'aurons pas assez de labels à répartir dans les feuilles. Dans le pire des cas, on appelle σ^2 fois la DistanceEuclidienne qui est en $\Theta(p+q+m)$. Nous avons donc que FurthestLabel est en $O((p+q+m)\cdot\sigma^2)$. Cependant, σ est un paramètre constant de l'algorithme. On est donc en $O(p+q+m)$.

L'algorithme *Split* appelle δ fois la fonction *FurthestLabel* qui est en $O(p+q+m)$ et $\sigma-\delta$ fois *FindClosest* qui est en $O(p+q+m)$. Ainsi, nous sommes ici en $O((p+q+m)\cdot\sigma) = O(p+q+m)$ pour les mêmes raisons que précédemment.

6 Expérimentation numérique

6.1 Instances utilisées

Nous allons dans cette expérimentation numérique utiliser des graphes représentant des grilles orientées : les noeuds seront rangés en n_l lignes et n_c colonnes et chacun, excepté ceux du bord qui n'ont pas forcément tous les voisins nécessaires, sera relié par un arc à celui au dessous et celui à droite. Les sommets sur le bord (du bas et de droite) seront reliés aux voisins disponibles uniquement. Le sommet de départ s sera toujours le sommet le plus en haut à gauche du graphe et celui d'arrivée sera le sommet le plus en bas à droite de la grille.

Ce type de graphe présente un avantage : d'une instance d'une taille n à une instance de même taille n, la structure du graphe reste la même et ainsi, le nombre d'arc entre le sommet de départ s et le sommet d'arrivée t reste inchangé : au moins $n_l + n_c$ arcs seront nécessaires. Seuls les coefficients des fonctions objectifs changent. Cela nous permet de nous prémunir contre des cas où par exemple il existe un arc entre s et t. Un chemin d'un arc permet d'aller du point de départ au point d'arrivée et celui-ci risque d'en dominer de nombreux autres et le nombre de chemin Pareto-optimal moyen risque d'être très faible Cela peut ainsi biaiser les résultats si par exemple pour toutes les autres instances il faut au moins 5 arcs pour aller de s à t.

Nous allons nommer ces instances avec le code suivant : G-$n_l \times n_c$-$objectifs$. Dans la partie objectif, le code p renverra à des objectifs de type sommet et le code q à des objectifs de type produit. Par exemple, le code G-10x20-2p1q correspond à une grille avec 10 lignes, 20 colonnes, 2 objectifs de type somme et 1 objectif de type produit. Les valeurs prises par les différents objectifs seront précisées avant chaque expérimentation.

Les expérimentations sont menées sur 25 à 100 runs et les résultats sont agrégés sous forme de moyenne. Pour les plus grosses instances (avec un temps d'exécution considérablement plus grand), nous nous sommes limités à 10, 5 ou 3 runs.

6.2 Étude sur les fonctions produit

Nous allons ici nous intéresser au nombre de chemins Pareto-optimaux obtenus lorsqu'une fonction objectif de type produit est introduite dans le modèle, ainsi que les répercutions sur le temps de résolution (exprimé en secondes). Pour cela, nous allons comparer des instances avec deux fonctions sommes et une fonction produit à des instances à trois fonctions sommes. Les coefficients des fonctions sommes prendront leurs valeur dans $\{0, ..., 100\}$.

Dans un premier temps, nous nous intéresserons à une fonction produit avec des coefficients tels que $q_{ij} \in]0; 1]$, $\forall\ (i, j) \in A$. Les résultats que nous obtenons sont dans la Table 2. Nous nous sommes affranchis de la valeur 0 car si celle-ci est présente, nous n'aurions pas eu un ensemble maximum complet des solutions efficaces pour la fonction produit, ce qui aurait pu biaiser les résultats.

données / Instances	Nombre moyen de chemins Pareto-optimaux	Temps moyen d'exécution
G-5x10-3p	35.895	0.01
G-5x10-2p1q	37.41	0.01
G-10x10-3p	118.67	0.1
G-10x10-2p1q	116.37	0.1
G-20x10-3p	352.86	1.28
G-20x10-2p1q	330.00	1.25
G-25x10-3p	440.32	2.45
G-25x10-2p1q	457.24	2.75

TABLE 2 – Comparaison du nombre de chemins Pareto-optimaux : $(3 - \Sigma)$ et $(2 - \Sigma \mid 1 - \Pi)$

On observe que le nombre de chemins Pareto-optimaux moyen entre une instance $(3-\Sigma)$ et une instance $(2-\Sigma|1-\Pi)$ n'est jamais significativement différent. Aucun des deux types d'instances ne prend le dessus sur l'autre concernant cette caractéristique, et ce quel que soit la taille du graphe. Il en va de même pour le temps d'exécution : aucune différence significative n'est notable.

Intéressons nous maintenant aux valeurs des coefficients de la fonction de type produit. Précédemment, nous avions $q_{ij} \in]0;1]$. On va ici s'interroger sur l'impact éventuel de la réduction de l'intervalle des coefficients. Nous allons prendre $q_{ij} \in [0.9;1]$. Les résultats sont recensés dans la Table 3.

Instances \ données	$q_{ij} \in]0;1]$	$q_{ij} \in [0.9;1]$
G-5x10-2p1p	37.41	35.51
G-10x10-2p1q	116.37	118.68
G-20x10-2p1q	330.00	337.86
G-25x10-2p1q	457.24	450.64

TABLE 3 – Comparaison du nombre de chemins Pareto-optimaux en fonction des coefficients de la fonction produit

Ici encore, il n'y a aucune différence majeure dans le nombre de chemins Pareto-optimal lorsque l'on fait varier l'intervalle des valeurs des coefficients q_{ij}.

6.3 Étude sur les ND-trees

Dans cette section, nous allons étudier empiriquement l'impact de l'utilisation de ND-trees dans l'algorithme de Martins. Nous allons conduire notre étude sur p fonctions de type somme uniquement. Le jeu de paramètres que nous allons utiliser sont ceux conseillés par l'expérimentation du papier de Andrzej Jaszkiewicz et Thibaut Lust [1], à savoir $\sigma = 20$ et $\delta = p + 1$. Dans la suite de l'expérimentation, nous allons fixer n_c à 10 et nous ferons varier n_l pour faire varier la taille du graphe. De plus, on aura $c_{ij}^k \in \{0, ..., 100\}\ \forall\ k \in \{1, ..., p\}, \forall\ (i,j) \in A$.

6.3.1 Nombre de label maximal posé sur un sommet

Nous allons ici étudier le nombre de label maximal posé sur un seul sommet pendant l'algorithme (c'est-à-dire $B = \max_{t \in \{1,...N\}} \max_{i \in S}(B_{i,t})$, avec N le nombre d'itérations de l'algorithme). En effet, au vu de l'analyse de complexité précédemment réalisée, si B_i est grand, cela risque d'influencer négativement la rapidité d'exécution. L'enjeu est donc ici de savoir à quel point B_i peut-être grand. Sur les figures 8, 9 et 10, nous pouvons observer la valeur de cette donnée en fonction de la taille du graphe sur respectivement 3, 4 et 5 objectifs. Sur ces figures, la droite rouge représente la droite d'équation $y = x$. On remarque que non seulement sur chacune des figures, B augmente de plus en plus vite lorsque la taille augmente, mais aussi que cette vitesse de croissance est de plus en plus rapide lorsque le nombre d'objectifs augmente. On arrive au constat que sur 5 objectifs, B croît ridiculement plus rapidement qu'une relation linéaire de type $y = x$: on a alors une relation exponentielle entre B et la taille du graphe.

6.3.2 Nombre d'objectifs

Nous allons nous intéresser ici à l'impact du nombre d'objectifs sur l'efficacité de l'utilisation des ND-trees. Pour cela, nous allons nous munir de 3 graphes de tailles différents : 7×10 (70 noeuds, figure 11), 12×10 (120 noeuds, figure 12) et 20×10 (200 noeuds, figure 13). Nous allons y faire varier le nombre d'objectifs et nous allons observer les temps d'exécution des deux algorithmes exprimés en secondes. Une échelle logarithmique est utilisée en ordonnées afin de pouvoir mieux observer la point de coupure où l'utilisation du ND-tree devient plus intéressante que l'implémentation linéaire, excepté sur la figure 9, qui est utilisée pour observer le gain obtenu.

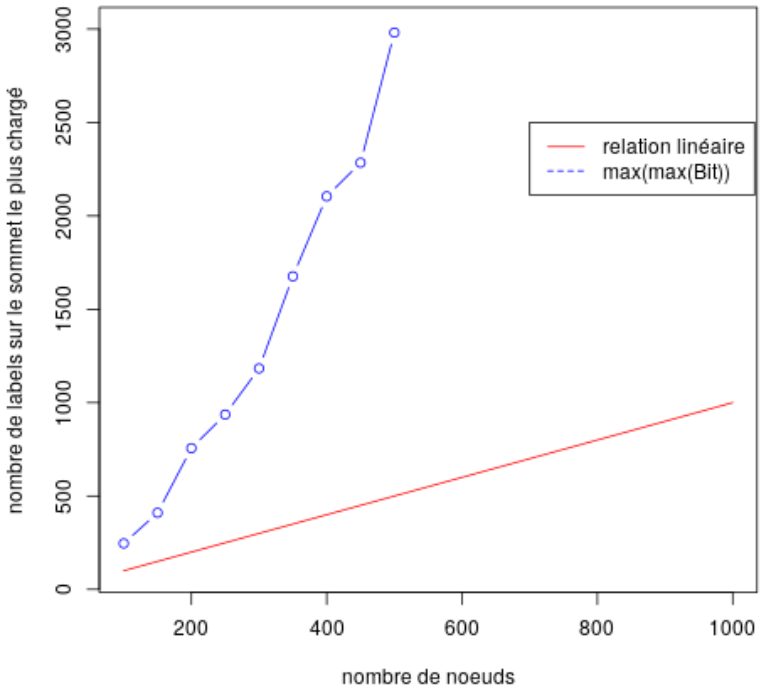

FIGURE 8 – $\max_{t\in\{1,\dots N\}} \max_{i\in S}(B_{i,t})$ en tri-objectif

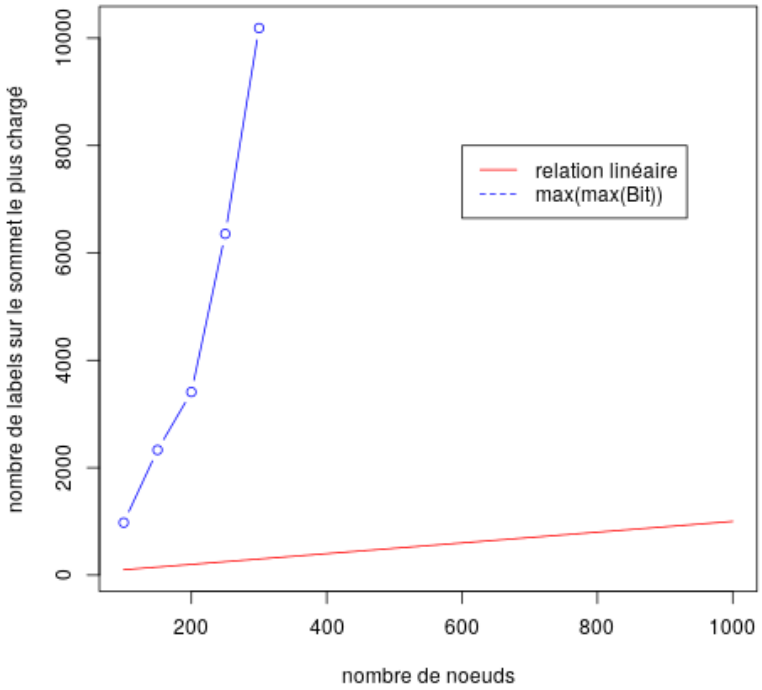

FIGURE 9 – $\max_{t\in\{1,\dots N\}} \max_{i\in S}(B_{i,t})$ avec 4 objectifs

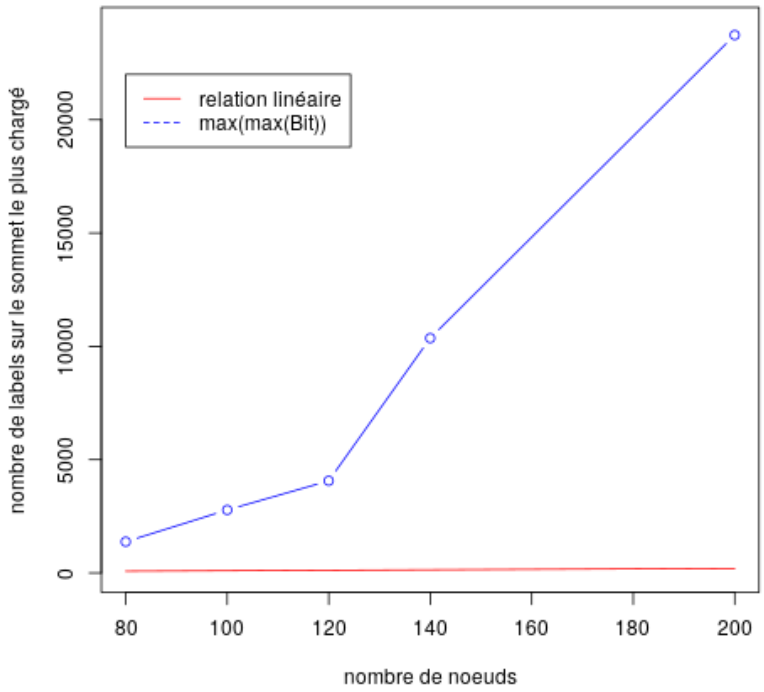

FIGURE 10 – $\max_{t\in\{1,\dots N\}} \max_{i\in S}(B_{i,t})$ avec 5 objectifs

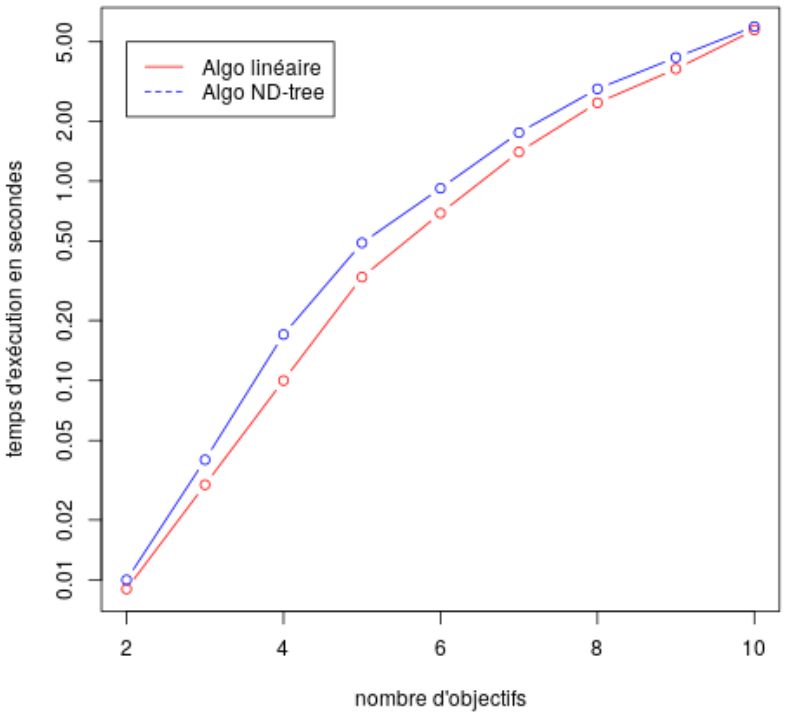

FIGURE 11 – Temps d'exécution en fonction du nombre d'objectifs sur une grille de taille 7×10

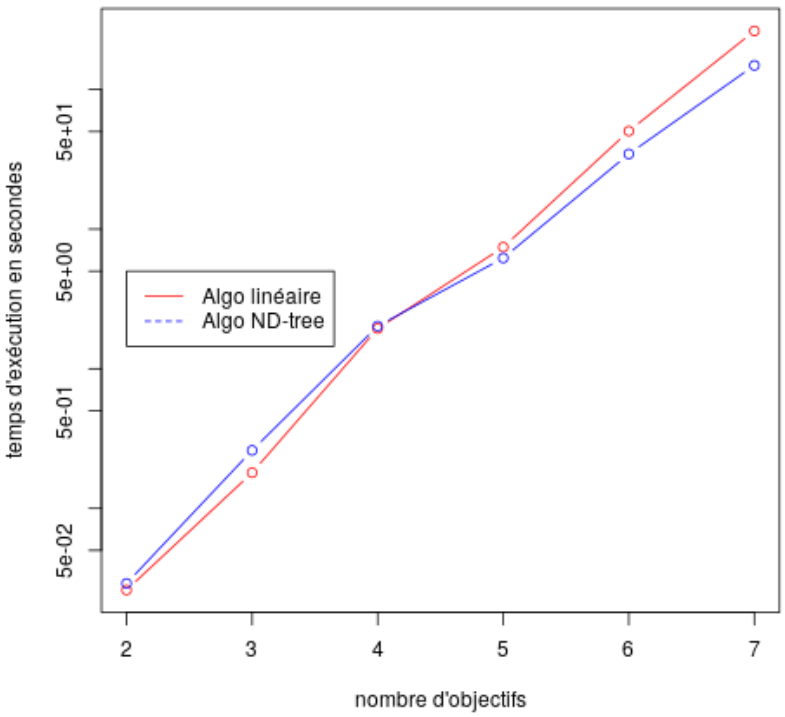

FIGURE 12 – Temps d'exécution en fonction du nombre d'objectifs sur une grille de taille 12×10

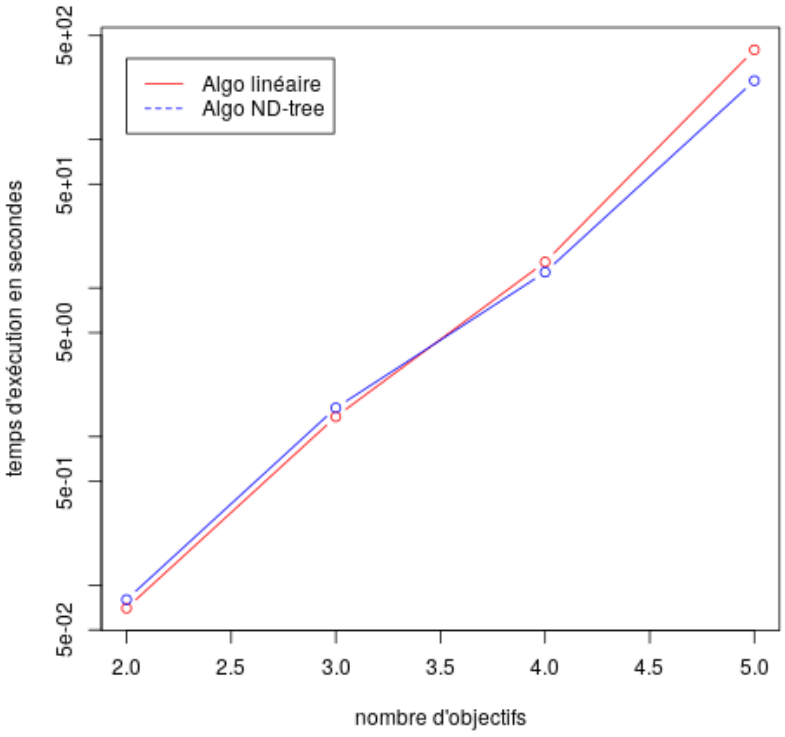

FIGURE 13 – Temps d'exécution en fonction du nombre d'objectifs sur une grille de taille 20×10

On remarque que pour les plus petits graphes, l'algorithme utilisant les ND-trees peine à réaliser de meilleures performances que l'implémentation linéaire et ce, quel que soit le nombre d'objectifs du modèle (figure 11). En revanche, lorsque l'on augmente la taille du graphe, le nombre d'objectifs minimal P pour lequel l'algorithme ND-tree est plus efficace que l'algorithme linéaire diminue (figures 12 et 13). Sur les graphes de taille 20×10, on voit un grand écart se dessiner entre les deux implémentations, en faveur du ND-tree (figure 14).

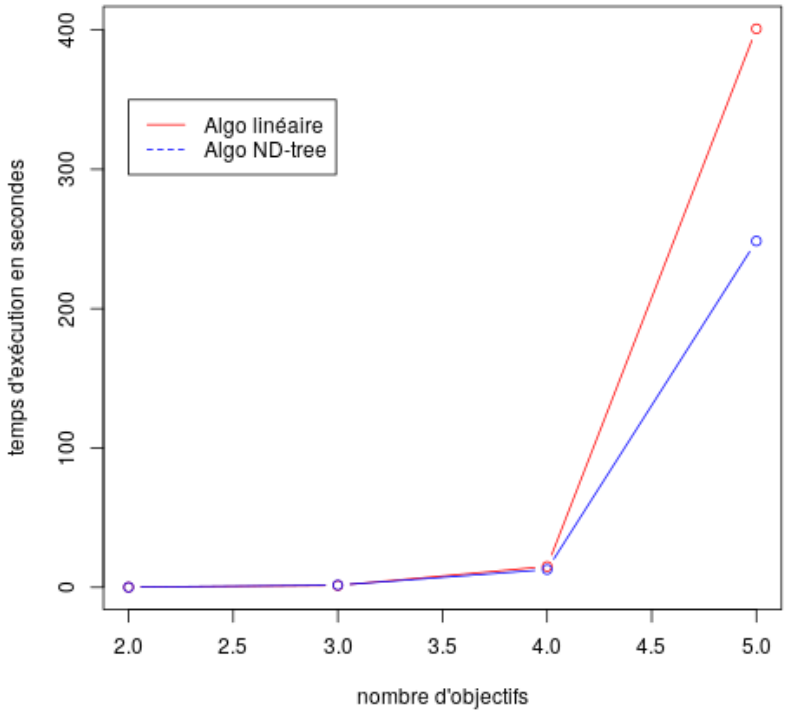

FIGURE 14 – Temps d'exécution en fonction du nombre d'objectifs sur une grille de taille 20×10 sur une échelle normale

6.3.3 Taille du graphe

Nous avons vu que l'efficacité de l'algorithme ND-tree est déterminée notamment par la taille du graphe et le nombre d'objectifs. L'enjeu va être ici pour un nombre d'objectif p fixé de déterminer à partir de quelle taille l'utilisation de ND-trees permet d'être plus efficace que l'implémentation linéaire. Nous allons effectuer cette analyse pour $p \in \{2, 3, 4, 5\}$. Une échelle logarithmique est utilisée en ordonnée afin de pouvoir mieux observer le point de coupure où l'utilisation du ND-tree devient plus intéressante que l'implémentation linéaire, les temps d'exécution variant énormément en fonction de la taille.

Analyse sur 2 objectifs

On observe qu'en bi-objectif, l'algorithme ND-tree ne parvient pas à avoir de meilleures performances que l'algorithme linéaire, y compris sur de grandes tailles de graphes (figure 15). Cela peut s'expliquer par le fait qu'en bi-objectif, le nombre de labels posé sur le sommet le plus chargé reste assez peu élevé (toujours inférieur à la droite d'équation $y = x$, dessinée en rouge)(figure 16).

Analyse sur 3 objectifs

En tri-objectif, c'est sur des graphes de $35 \times 10 = 350$ sommets et plus que l'algorithme ND-tree devient plus efficace (figure 17). Pour ce type d'instance, le nombre de chemins Pareto-optimaux moyen est de 722 et le nombre maximal moyen de labels posés sur un sommet est de 1676. Les

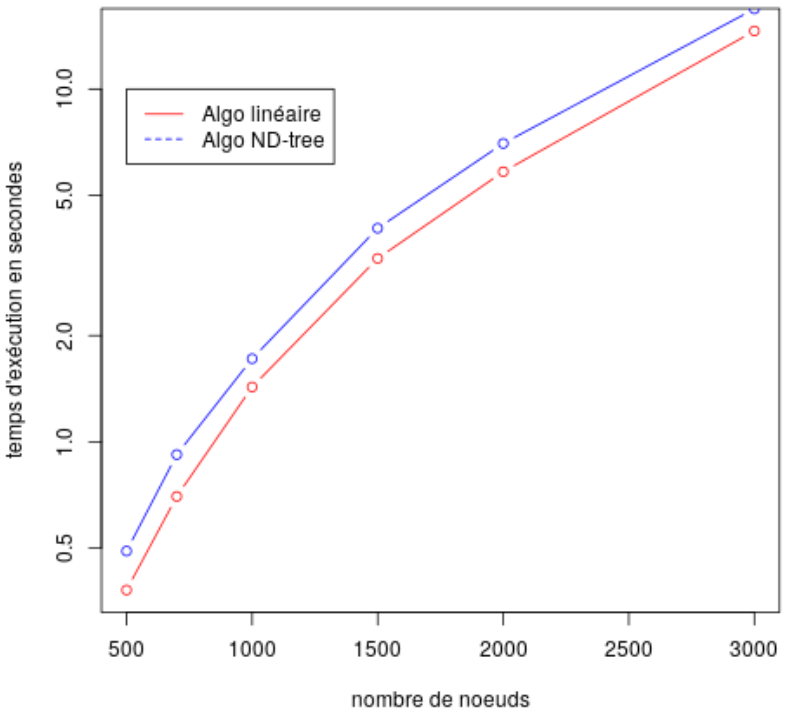

FIGURE 15 – Analyse 2 objectifs

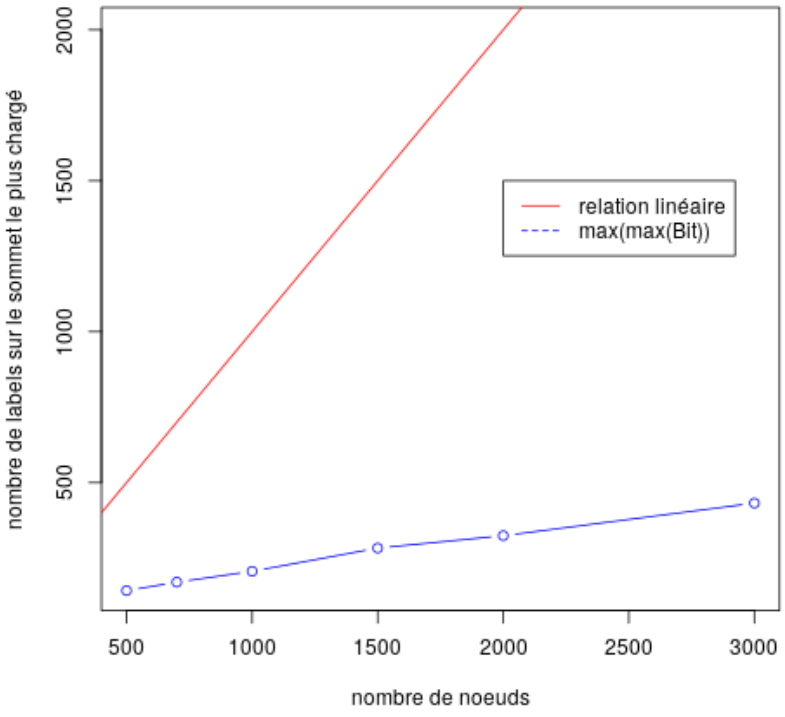

FIGURE 16 – Valeur de B en bi-objectif

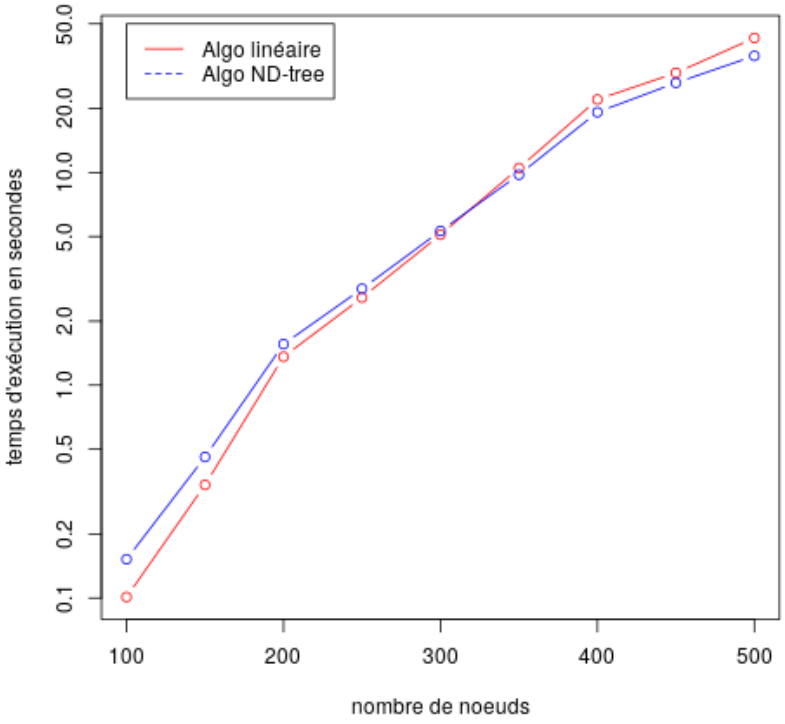

FIGURE 17 – Analyse 3 objectifs

gains en tri-objectif sont présentés dans la Table 4. Le gain représente le gain en pourcentage de temps d'exécution : de combien de pourcent avons-nous réduit le temps d'exécution grâce à l'utilisation des ND-trees ?

Instance	Taille (nombre de sommets)	Gain
G-35x10-3p	350	6.96%
G-40x10-3p	400	12.85%
G-45x10-3p	450	10.24%
G-50x10-3p	500	17.65%

TABLE 4 – Table des gains avec 3 objectifs

Analyse sur 4 objectifs

Avec 4 objectifs, c'est autour de $15 \times 10 = 150$ sommets et plus que l'algorithme ND-tree devient plus efficace en moyenne (figure 18). Pour cette taille, le nombre de labels maximum moyen sur un sommet est de 2330 et le nombre moyen de chemins Pareto-optimaux est de 1063. La table est gains au delà de 150 sommets est donnée dans la Table 5. On remarque que plus la taille augmente, plus le gain est grand.

Instance	Taille (nombre de sommets)	Gain
G-15x10-4p	150	8.45%
G-20x10-4p	200	14.64%
G-25x10-4p	250	28.45%
G-30x10-4p	300	33.23%

TABLE 5 – Table des gains avec 4 objectifs

Analyse sur 5 objectifs

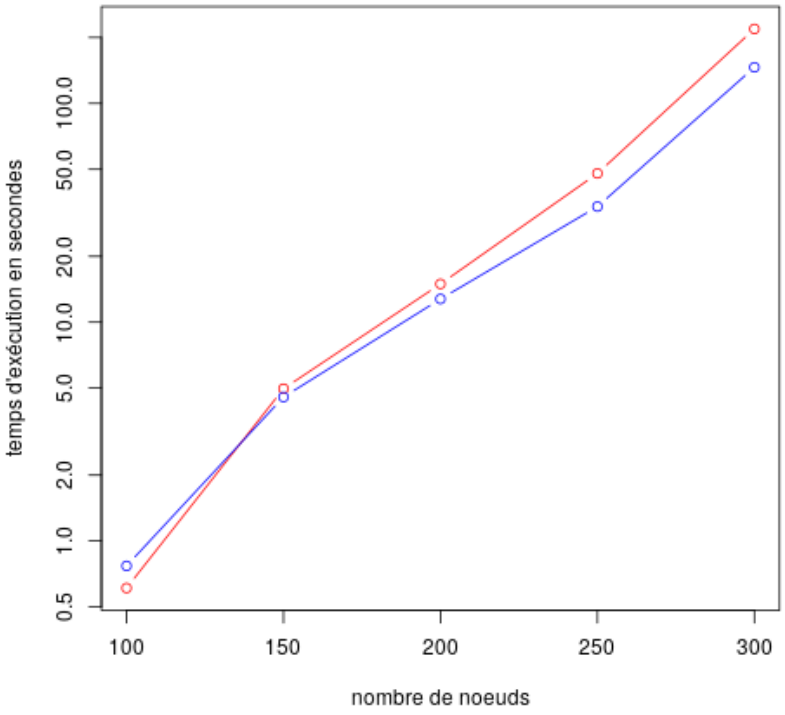

FIGURE 18 – Analyse 4 objectifs

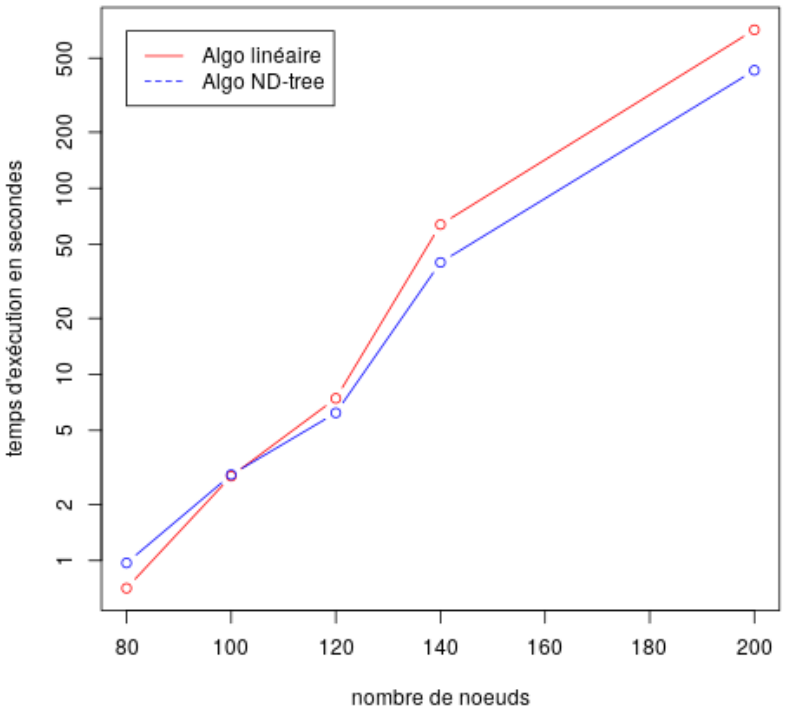

FIGURE 19 – Analyse 5 objectifs

Avec 5 objectifs, c'est autour de $10 \times 10 = 100$ sommets et plus que l'algorithme ND-tree devient plus efficace (figure 19). Pour cette taille, le nombre de chemins Pareto-optimaux moyen y est de 2466 et le sommet le plus chargé au cours de l'algorithme a 4066 labels. Les gains sont donnés dans la Table 6. On remarque qu'en 5 objectifs, le gain commence à être très intéressant, en diminuant jusqu'à 39.46% le temps de résolution grâce à l'utilisation des ND-trees.

Instance	Taille (nombre de sommets)	Gain
G-12x10-5p	120	16.42%
G-14x10-5p	140	37.36%
G-20x10-5p	200	39.46%

TABLE 6 – Table des gains avec 5 objectifs

6.3.4 Conclusions de l'expérimentation

L'utilisation des ND-trees dans l'algorithme de Martins semble intéressante lorsque l'on passe sur de grands graphes, ou lorsque que notre modélisation a de nombreux objectifs. Cette pratique présente en revanche un manque d'efficacité face à l'implémentation linéaire lorsque le nombre de labels sur les sommets ne monte pas suffisamment haut, comme c'est le cas en bi-objectif dans notre contexte. Sur les petits graphes, l'utilisation de ND-tree semble moins efficace, même si le nombre de labels sur le sommet le plus chargé est élevé. Par exemple, sur des instances de type G-8x10-5p, le sommet le plus chargé monte en moyenne à 1379 labels mais le ND-tree reste moins efficace alors que sur des instances de type G-35x10-3p, le sommet le plus chargé monte à 722 labels et l'implémentation linéaire est ici moins efficace. Ce type de constat s'est retrouvé lors de l'étude des instances de type G-7x10 de la figure 6.

7 Conclusion

Nous avons tout d'abord cherché à introduire des mesures de fiabilité des chemins dans un problème de plus courts chemins multi-objectif. Nous avons vu qu'il était possible de le faire en ajoutant un objectif de type produit dans la modélisation, à condition que les fiabilités sur chaque arc restent indépendantes. L'enjeu était ensuite de pouvoir résoudre ce nouveau problème en incluant ce nouveau type de fonction objectif. Cela a pu se faire en étendant l'algorithme de Martins à celles-ci grâce auquel nous obtenons au moins un ensemble complet des solutions efficaces. Après implémentation, nous avons pu observer que l'introduction d'une fonction produit n'a pas d'impact majeur sur le temps de résolution ou sur le nombre de solutions efficaces par rapport à l'introduction d'une fonction somme. Il serait intéressant à l'avenir d'introduire cet algorithme de résolution de ce nouveau type de plus courts chemins à un solveur multi-objectif tel que vOptSolver. Une autre amélioration possible est l'extension et la résolution du problème avec $(p - \Sigma \mid q - \Pi \mid m - M)$ et $m > 1$.

Nous avons ensuite porté notre attention sur le test de dominance. En effet, au cours de l'implémentation, nous avons observé une complexité linéaire en le nombre de labels présents sur un sommet. Or, nous avons vu qu'empiriquement, le nombre de labels présents sur un sommet peut monter très vite. Lorsqu'il y a beaucoup d'objectifs, on a pu observer une relation exponentielle entre la taille du graphe étudié et le nombre maximal de labels présents sur le sommet le plus chargé. Cela a pour conséquence d'influer négativement sur la vitesse de résolution. Afin d'atténuer le problème et d'obtenir de meilleures performances, nous avons cherché à mettre un ND-tree sur chaque sommet pour réaliser ce test de dominance. Cette stratégie s'est avérée le plus efficace lorsque le nombre d'objectif était grand et lorsque le graphe était suffisamment gros en terme de nombre de sommets, mais moins intéressante sur de petits graphes ainsi qu'en problème bi-objectif. L'étude numérique a été menée sur des graphes formant des grilles. Un axe d'étude futur possible serait de faire varier la topologie de graphe et d'observer si sur certaines structures l'utilisation de ND-trees est plus, moins ou aussi efficace. Il serait aussi intéressant d'étudier si les paramètres σ et δ conseillés restent les plus efficaces dans l'utilisation que nous faisons des ND-trees.

Références

[1] A.Jaszkiewicz and T.Lust. *ND-Tree-based update : a Fast Algorithm for the Dynamic Non-Dominance Problem.* Cornell University Library, 2017.

[2] E. Martins. On a multi criteria shortest path problem. *European Journal of Operation Reasearch, vol 16*, pages 236–245, 1984.

[3] M.Ehrgott. *Multicriteria Optimization.* Springer, 2005.

[4] O.Dib. *Reroutage dynamique des passagers dans les réseaux de transport multimodaux.* PhD thesis, Université de Bourgogne Franche Comté, 2017.

[5] Z. Tarapata. Selected multicriteria shortest path problems : an analysis of complexity, models and adaptation of standard algorithms. *Int. J. Appl. Math. Comput. Sci., vol 17*, pages 269–287, 2007.

[6] X.Gandibleux, F.Beugnies, and S.Randriamasy. Martins' algorithm revisited for multi-objective shortest path problems with a maxmin cost function. *4OR*, pages 47–59, 2006.