Vérification de l'algorithme de calcul des ordres d'appel dans Parcoursup

Hugo Gimbert, Pierre Castéran, Claire Mathieu, Gérald Point

Vérification de l'algorithme de calcul des ordres d'appel dans Parcoursup

Pierre Castéran[1], Hugo Gimbert*[1], Claire Mathieu[2], and Gérald Point[1]

[1]LaBRI, CNRS, Université de Bordeaux, France
[2]IRIF, CNRS, France

November 4, 2021

Abstract

Parcoursup est la plateforme nationale de préinscription en première année de l'enseignement supérieur en France. Les méthodes formelles sont un ensemble de techniques permettant de garantir qu'un algorithme satisfait bien sa spécification. Ce document présente une partie des travaux de sûreté logicielle autour d'un des algorithmes de cette plateforme, appelé *calcul des ordres d'appel*. Ces travaux incluent la spécification de l'algorithme, la vérification à l'exécution de cette spécification, et la preuve formelle de correction de l'algorithme. Ces travaux sont réalisée à l'aide des outils de preuve `Why3` et `COQ` et de techniques de vérification à l'exécution du code Java. Ces travaux permettent d'atteindre un très haut niveau de confiance dans l'algorithme de calcul des ordre d'appel de Parcoursup.

Contents

*Les auteurs de ces travaux ont pu bénéficier de plusieurs subventions du MESRI sur la période 2018-2021.

1 Introduction

Parcoursup est la plateforme nationale de préinscription en première année de l'enseignement supérieur en France. Chaque année, plus de 1 000 000 candidats toutes procédures confondues formulent des vœux sur Parcoursup, où plus de 19 500 formations de l'enseignement supérieur sont proposées.

1.1 Calcul des ordres d'appel dans Parcoursup.

La procédure principale de Parcoursup se décompose en plusieurs phases. Initialement, les candidats font des voeux pour les différentes formations proposées. Ensuite chaque formation classe une partie des candidats ayant effectué des voeux dans cette formation [1]. Enfin les candidats reçoivent des propositions.

La loi ORE prévoit l'application de taux encadrant la proportion de candidats boursiers et de candidats hors-secteur dans les formations concernées [2]:

> "(...) lorsque le nombre de candidatures excède les capacités d'accueil d'une formation, l'autorité académique fixe un pourcentage minimal de bacheliers retenus bénéficiaires d'une bourse nationale de lycée, en fonction du rapport entre le nombre de ces bacheliers boursiers candidats à l'accès à cette formation et le nombre total de demandes d'inscription (...). Pour l'accès à ces mêmes formations et compte tenu du nombre de candidats à ces formations résidant dans

[1] plus précisément chaque formation dans laquelle le nombre de candidature est supérieur au nombre de places disponibles.
[2] cf sections V et VI de l'article L. 612-3 du code de l'éducation.

> l'académie, l'autorité académique fixe également, afin de faciliter l'accès des bacheliers qui le souhaitent aux formations d'enseignement supérieur situées dans l'académie où ils résident, un pourcentage maximal de bacheliers retenus résidant dans une académie autre que celle dans laquelle est situé l'établissement. (...) Seule l'obligation de respecter le pourcentage minimal de bacheliers boursiers retenus peut conduire à déroger au pourcentage maximal de bacheliers retenus résidant dans une autre académie. "

La mise en oeuvre des taux minimum de candidats boursiers et des taux minimum de candidats résidents dans l'académie est traitée en partie de manière automatisée, à l'échelle nationale, par un algorithme mis en oeuvre par le Service à Compétence Nationale Parcoursup (SCN Parcoursup). Le contexte de cette automatisation et le fonctionnement de cet algorithme sont détaillés dans le "Document de présentation des algorithmes de Parcoursup" [11, Section 4] et les algorithmes et leur implémentation open-source sont consultables sur le dépôt de code dédié [12].

Chaque recteur de région académique fixe les taux s'appliquant à chaque formation de son territoire, et l'algorithme applique les taux indépendamment dans chaque formation [3]. Pour cela, l'algorithme calcule un *ordre d'appel* des candidats, qui définit l'ordre dans lequel les propositions seront faites aux candidats. Si les taux minimum de boursiers et de résidents étaient de 0%, l'ordre d'appel serait tout simplement l'ordre du classement pédagogique des candidatures, tel que transmis par la formation. Dès lors qu'il convient de respecter ces taux, l'ordre d'appel est obtenu à partir du classement pédagogique en faisant remonter des boursiers et/ou des résidents dans le classement afin de garantir le respect desdits taux.

1.2 Vérification des logiciels critiques.

Les méthodes formelles sont un ensemble de techniques permettant de garantir qu'un algorithme satisfait bien sa spécification. Ces méthodes s'appliquent en particulier aux algorithmes et logiciels dits *critiques*, c'est-à-dire dont les défaillances ont un coût potentiel important, en terme humains ou financiers. Plusieurs logiciels de la plateforme Parcoursup sont des logiciels *critiques*. Par exemple une erreur de calcul des ordres d'appel pourrait nécessiter un redémarrage de tout ou partie de la procédure d'admission, avec un fort coût en terme de temps perdu pour les candidats et les formations.

La vérification logicielle permet de garantir qu'un algorithme est conforme à sa spécification, c'est-à-dire que le résultat de ses calculs est bien conforme à ce qui est attendu. Il faut distinguer un algorithme et son implémentation sous forme de programme: un algorithme désigne le principe de calcul lui-même, qui peut être décrit en langue naturelle, en pseudo code ou bien en langage mathématique, alors que l'implémentation désigne la traduction de l'algorithme en un langage de programmation, préalable à son exécution effective par un ordinateur. Pour augmenter le degré de confiance, il est intéressant de vérifier non seulement les algorithmes utilisés par les logiciels critiques, mais également leurs implémentations sous forme de programmes. Pour aller plus loin on peut même vouloir vérifier le processus de traduction du programme en langage machine, c'est-à-dire la compilation, point qui n'est pas abordé dans le document présent.

Dans ce document nous présentons plusieurs mesures mises en oeuvre pour atteindre un haut niveau de confiance dans l'algorithme de l'ordre d'appel et son implémentation:

- spécification des algorithmes,
- preuve mathématique de la conformité des algorithmes à leur spécification,
- preuve formelle de la conformité d'une implémentation à la spécification,
- vérification à l'exécution de la spécification.

[3] Dans le cas où une même formation ventile les différentes candidatures dans différents groupes de candidats, en fonction notamment du type de diplôme détenu par chaque candidat, alors l'algorithme est appliqué indépendamment dans chaque groupe.

1.3 Algorithme de calcul des ordres d'appels

Dans une formation soumise à deux taux q_B (boursiers) et q_R (résidents), l'algorithme consiste à énumérer les candidats à partir du candidat mieux classé, en les intégrant un à un dans l'ordre d'appel. Ce faisant, on surveille le respect des taux. Si le taux boursier devient contraignant, et que le prochain candidat du classement ne l'est pas, alors on fait remonter à sa place le prochain candidat boursier dans le classement. Même chose si le taux résident devient contraignant. Si les deux taux sont contraignants à la fois, alors on fait remonter un boursier résident, si il en reste un. Si aucun candidat n'est à la fois boursier et résident, priorité est donnée au taux de boursiers sur le taux de résidents, conformément à la loi ORE.

Algorithme de calcul de l'ordre d'appel

1. Notons n le nombre de candidats apparaissant dans le classement pédagogique de la formation.
2. Pour chaque entier k de 1 à n, dans cet ordre, le candidat C_k de rang k dans l'ordre d'appel est calculé de la manière suivante. On a déjà sélectionné les candidats $C_1, \ldots, C_{k-1}$ dans l'ordre d'appel, et parmi eux il y a b boursiers et r résidents. On dit que le taux minimum boursiers est contraignant si $b/k < q_B$ et que le taux résidents est contraignant si $r/k < q_R$. On considère tous les autres candidats, pris dans l'ordre pédagogique. Pour choisir C_k, parmi ceux-là:
   - Si aucun des deux taux n'est contraignant, on prend le premier candidat.
   - Si seul le taux minimum boursiers est contraignant, on prend le premier candidat boursier s'il y en a, le premier candidat sinon.
   - Si seul le taux résident est contraignant, on prend le premier candidat résident s'il y en a, le premier candidat sinon.
   - Si les deux taux sont contraignants, on prend le premier candidat qui soit à la fois résident et boursier s'il y en a; sinon, le premier candidat boursier s'il y en a; sinon, le premier candidat résident s'il y en a; sinon, le premier candidat.

Illustrons cet algorithme avec un exemple. Dans cette formation, le taux minimum de résidents est fixé à $q_R = 0$ et que le taux minimum de boursiers est de $q_B = 20\%$, et que la formation a classé ses candidats dans l'ordre $(C_1, C_2, B_3, C_4, C_5, C_6, C_7, C_8, C_9, B_{10}, C_{11})$, où seuls deux candidats B_3 et B_{10} sont boursiers. Alors l'ordre d'appel sera $(B_3, C_1, C_2, C_4, C_5, B_{10}, C_6, C_7, C_8, C_9, C_{11})$. En effet, d'une part le taux résident n'est jamais contraignant. D'autre part le taux boursier est contraignant lors du choix des candidats d'indices 1 et 6, ce qui permet à B_3 et B_{10} de "remonter" dans l'ordre d'appel. Remarquons que le choix du candidat d'indice 11 est également contraignant, mais à ce stade du calcul de l'ordre d'appel tous les boursiers ont déjà été sélectionnés.

1.4 Spécification

Le document de présentation des algorithmes de Parcoursup [11, Section 4] précise un certain nombre de propriétés que l'algorithme est tenu de respecter. Dans le domaine de la sûreté logicielle, ces propriétés sont appelées une *spécification* de l'algorithme. Une spécification peut être exprimée dans différents formalismes, soit informellement sous forme d'un cahier des charges rédigé en langue naturelle, soit formellement dans un langage de spécification logiciel tel que JML [10] ou ACSL [1]. Pour des raisons de lisibilité, la spécification disponible dans [11, Section 4] est exprimée de manière semi-formelle.

L'ordre d'appel calculé par l'algorithme est représenté par une permutation $d_1, d_2, \ldots, d_{|C|}$ de l'ensemble C des candidats apparaissant dans le classement pédagogique, qui doit vérifier les propriétés suivantes.

Spécification du calcul de l'ordre d'appel.

Q1) Pour tout $k \in 1 \dots |C|$, $d_1, \dots, d_k$ contient au moins $\lceil q_B * k \rceil$ candidats boursiers ; ou sinon, aucun candidat parmi $d_{k+1} \dots d_{|C|}$ n'est boursier.

Q2) Pour tout $k \in 1 \dots |C|$ au moins

$$\min\left(\lceil q_R * k \rceil, \lceil (1 - q_B) * k - 1 \rceil\right)$$

candidats parmi $d_1, \dots, d_k$ sont résidents ; ou sinon, aucun candidat parmi $d_{k+1} \dots d_{|C|}$ n'est résident.

Q3) Un candidat résident boursier qui a le rang r dans le classement pédagogique n'est jamais doublé par personne et a un rang inférieur ou égal à r dans l'ordre d'appel.

Q4.1) Un candidat résident non-boursier qui a le rang r dans le classement pédagogique ne peut être doublé que par des boursiers, résidents ou non, et a un rang dans l'ordre d'appel inférieur ou égal à $1 + \left\lfloor (1 + \frac{q_B}{1-q_B})r \right\rfloor$.

Q4.2) Un candidat non-résident boursier qui a le rang r dans le classement pédagogique ne peut être doublé que par des résidents, boursiers ou non, et a un rang dans l'ordre d'appel inférieur ou égal à $1 + \left\lfloor (1 + \frac{q_R}{1-q_R})r \right\rfloor$.

Q5) Un candidat non-résident et non-boursier ne double jamais personne. Si $q_B + q_R < 1$ alors un candidat non-résident non-boursier qui a le rang r dans le classement pédagogique aura un rang dans l'ordre d'appel inférieur ou égal à $\left\lceil \left(1 + \frac{q_R+q_B}{1-(q_R+q_B)}\right)(r+1) \right\rceil$.

Les propriétés Q1 et Q2 garantissent le respect des taux, tout en prenant en compte la priorité donnée au taux boursier en cas d'impossibilité de respecter les deux taux en même temps. Les propriétés Q3, Q4 et Q5 spécifient dans quels cas un candidat peut perdre des places lors du calcul de l'ordre d'appel, et dans quelles proportions.

1.5 Spécification supplémentaire dans les formations soumises au seul taux boursier

Dans Parcoursup, le taux minimum de résidents ne s'applique qu'aux formations dites *non-sélectives*, c'est-à-dire à une partie des licences. Dans la plupart des formations, le taux q_R est donc fixé à 0%, en d'autres termes il n'y a pas de contraintes sur ce taux, seul le taux boursier est pris en compte. Dans ce cas, la spécification se complète de deux propriétés supplémentaires.

Spécification du calcul de l'ordre d'appel dans les formations soumises au seul taux boursiers.

P4) Supposons $q_R = 0$. Comparé au classement pédagogique, l'ordre d'appel minimise le nombre d'inversions (distance de Kendall-tau), parmi ceux qui garantissent la propriété P1.

P5) Supposons $q_R = 0$. Si l'on munit l'ensemble des permutations de candidats de l'ordre lexicographique induit par les classements, alors l'ordre d'appel est le maximum parmi toutes les sélections qui garantissent la propriété P1.

P6) Supposons $q_R = 0$.

P6.0) Le résultat du calcul doit être indépendant du fait qu'un candidat soit résident ou non.

Les propriétés (Q1) à (Q5) définies ci-dessus sont satisfaites, en supposant tous les candidats résidents. Par conséquent:

P6.1) Pour tout k, l'ensemble de candidats $d_1, \ldots, d_k$ contient au moins $q_B * k$ candidats boursiers ; ou sinon, aucun candidat parmi $d_{k+1} \ldots d_{|C|}$ n'est boursier.

P6.2) Un candidat boursier qui a le rang r dans le classement pédagogique n'est jamais doublé par personne et aura donc un rang inférieur ou égal à r dans l'ordre d'appel.

P6.3) Un candidat non boursier qui a le rang r dans le classement pédagogique ne double jamais personne et aura un rang dans l'ordre d'appel compris entre r et $1+\lfloor r(1+q_B/(1-q_B))\rfloor$.

La propriété P6.0 garantit que le résultat du secteur est indépendante du fait que les candidats sont considérés comme étant du secteur ou non. La propriété P6.1 garantit le respect des taux boursiers. Les propriétés P6.2 et P6.3 spécifient dans quels cas et quelles proportions un candidat peut perdre des places lors du calcul de l'ordre d'appel, par rapport à son rang initial dans le classement pédagogique. Les propriétés P4 et P5 offrent une garantie de minimalité de la modification du classement sous contrainte de respecter le taux boursier.

1.6 Vérification de l'algorithme d'application des taux boursiers.

Les travaux de vérification logicielle effectué sur cet algorithme sont présentés dans la section 2.

La section 2.1.1 présente une *preuve mathématique* de la conformité de l'algorithme à sa spécification (Q1)-(Q5) et la section 3.1 la conformité à (P4)-(P6). La preuve s'applique à une formalisation mathématique de l'algorithme. Une preuve mathématique telle que celle présentée permet d'atteindre un bon niveau de confiance, mais laisse au lecteur le soin de vérifier un certain nombre de détails.

La section 2.2 présente une *preuve formelle* de la conformité de l'algorithme à sa spécification. La preuve s'applique à une implémentation de l'algorithme dans le langage `WhyML`, en s'appuyant sur la plateforme `Why3` pour la preuve de programme [4, 5]. L'implémentation `WhyML` et les preuves formelles associées sont disponibles sur le dépôt `https://gitub.u-bordeaux.fr/parcoursup/why3`. Une telle preuve formelle permet d'atteindre un très haut niveau de confiance, puisqu'elle garantit l'absence de bugs quelques soient les données d'entrée traitées par l'algorithme. En particulier le niveau de confiance est beaucoup plus haut que dans le cadre du test logiciel, qui ne couvre en général que quelques exécutions possibles. L'implémentation en Whyml de l'algorithme n'est pas l'implémentation Java effectivement exécutée par le SCN Parcoursup. Cette mesure garantit donc l'absence de bugs dans l'implémentaton WhyML mais pas dans l'implémentation Java. Le langage WhyML se prête lui même peu à une exploitation par le SCN Parcoursup, mais l'implémentation `WhyML` peut être automatiquement traduite en une implémentation dans le langage OCaml [13], certifiée sans bugs. Actuellement c'est l'implémentation Java qui est exploitée, avec une vérification en double-aveugle contre une autre implémentation en PL/SQL.

La section 2.3 présente les mesures de *vérification à l'exécution* intégrées dans l'implémentation Java de Parcoursup. Cette mesure permet de certifier que chaque exécution du logiciel est conforme à sa spécification,

tant qu'elle ne lève pas une exception. Le niveau de confiance est moindre que celui d'une preuve formelle, car cette mesure de vérification à l'exécution écarte la possibilité d'une erreur dans les exécutions passées du programme, mais ne garantit rien sur les exécutions futures avec de nouvelles données d'entrée. Toutefois, cette mesure a pu s'appliquer dès la première année à l'implémentation Java exploitée par le SCN Parcoursup et a pu être mise en place rapidement dès la première année, en avance de phase par rapport au développement des preuves formelles.

1.7 Travaux connexes.

Ce travail de preuve a débuté au LRI, en collaboration avec le second auteur de ce rapport et un soutien financier du MESRI. Les travaux réalisés au LRI ont permis de réaliser la preuve des propriétés P5) et P6) sur une implémentation `WhyML` de l'algorithme à un taux, ainsi qu'une preuve partielle d'une implémentation alternative en Java [2, 3]. La suite de ces travaux (propriété P4 de l'algorithme à un taux et preuve des propriétés l'algorithme à deux taux) a été effectuée au LaBRI et est l'objet du présent rapport. Pour cela nous avons effectué une nouvelle implémentation `WhyML` de l'algorithme à deux taux, la plus proche possible du code `Java`, et nous avons effectué une preuve la plus lisible et modulaire possible, qui suive la trame de la preuve mathématique et soit facilement rejouable.

Les méthodes formelles sont également appliquées à d'autres algorithmes de décision publiques, notamment la sécurisation du calcul des impôts et des allocations sociales grâce au langage CATALA [9] ou les systèmes de régulation des heures de conduite des chauffeurs routiers [8]. Plus globalement, une réflexion internationale appelée *Rules as Code* interroge les conditions de mise en oeuvre et d'élaboration des lois dont l'application est partiellement ou totalement automatisée [14].

2 Preuve des propriétés (Q1)-(Q5) du calcul des ordres d'appel

2.1 Preuve mathématique

Dans cette section on présente trois mesures garantissant la conformité de l'algorithme de calcul des ordres d'appel aux spécifications (Q1)-(Q5): preuve mathématique, preuve formelle et vérification à l'exécution.

2.1.1 Formalisation mathématique de l'algorithme du calcul de l'ordre d'appel

L'algorithme du calcul de l'ordre d'appel prend en entrée:

- un taux minimum de candidats boursiers $q_B \in [0,1]$
- un taux minimum de candidats résidents $q_R \in [0,1]$
- Un ensemble C de candidats partitionné d'une part entre boursiers et non-boursiers (partition $(B, \overline{B})$) et d'autre part entre candidats résidents et candidats non-résidents (partition $(R, \overline{R})$).
- un classement pédagogique rang : $C \to 1 \dots |C|$ des candidats.

On note $(c_i)_{i \in 1 \dots |C|}$ la suite des candidats triés par classement pédagogique. On suppose que les rangs dans le classement pédagogique sont des entiers consécutifs à partir de 1 sans ex-aequo, c'est-à-dire que $\forall i \in 1 \dots |C|, \text{rang}(c_i) = i$.

L'algorithme calcule une permutation $(d_k)_{k \in 1 \dots |C|}$ des candidats, appelée *l'ordre d'appel*.

L'algorithme calculant les listes d'appel a été implémenté en Java (lien vers le fichier Java) et est décrit dans [11, section 5].

On définit $BR = B \cap R$ et avec des notations similaires C est partitionné en $BR \cup \overline{B}R \cup B\overline{R} \cup \overline{BR}$.

L'ordre d'appel est calculé progressivement en $|C|$ étapes $1, \dots, |C|$. Au début de l'étape k l'algorithme a calculé les k premiers candidats dans l'ordre d'appel:

$$oa_k = d_1, \dots, d_{k-1}$$

ainsi que quatre files $q_{BR}, q_{\overline{B}R}, q_{B\overline{R}}, q_{\overline{BR}}$ qui maintiennent la liste des candidats de chaque type qui n'ont pas encore été intégrés dans l'ordre d'appel. Chacune des quatre files est triée selon les rangs croissants des candidats. Par exemple, à l'étape k la file q_{BR} contient le tri par rangs croissants des candidats de $BR \setminus \{d_1, \ldots, d_{k-1}\}$.

Le choix de d_k est fait à l'étape k comme suit.

On dit que le taux de boursiers est *contraignant* après $d_1 \ldots d_{k-1}$, si il reste au moins un boursier à sélectionner et que ajouter un non-boursier à la permutation ferait passer le taux de boursiers en dessous du minimum fixé c'est-à-dire si

$$\sharp_B(d_1 \ldots d_{k-1}) < |B| \text{ et } \sharp_B(d_1 \ldots d_{k-1}) < q_B * k \ .$$

On utilise une définition similaire pour le taux de résidents, qui est contraignant si:

$$\sharp_R(d_1 \ldots d_{k-1}) < |R| \text{ et } \sharp_R(d_1 \ldots d_{k-1}) < q_R * k \ .$$

L'algorithme utilise la propriété suivante.

Lemma 1. *Supposons que les deux taux soient contraignants après $d_1 \ldots d_{k-1}$ et que q_{BR} est vide. Alors $q_{B\overline{R}}$ est non-vide.*

Proof. Notons $C' = C \setminus \{d_1, \ldots, d_{k-1}\}$. Si le taux boursier est contraignant, alors $\sharp_B(d_1 \ldots d_{k-1}) < |B|$ donc $C' \cap B \neq \emptyset$. Donc $(C' \cap BR \neq \emptyset) \vee (C' \cap B\overline{R} \neq \emptyset)$. □

Le choix de d_k est fait en plusieurs étapes.

1. on sélectionne parmi les candidats en tête des quatre files $q_{BR}, q_{\overline{B}R}, q_{B\overline{R}}, q_{\overline{BR}}$ la liste des candidats éligibles. Pour qu'un candidat soit éligible il doit satisfaire les deux conditions suivantes.
   - si le taux boursier est contraignant alors les non-boursiers ne sont pas éligibles.
   - si le taux résident est contraignant alors les non-résidents ne sont pas éligibles.
2. si la liste des candidats éligibles est non-vide, on ajoute à oa_k celui des candidats éligibles qui a le plus petit rang et on le retire de la file correspondante,
3. si la liste des candidats éligibles est vide, alors c'est que les deux taux sont contraignants (si un seul des taux est contraignant alors par définition il y a au moins un candidat éligible). D'après le lemme supra, la file q_{BR} est vide et la file $q_{B\overline{R}}$ ne l'est pas. Dans ce cas on dit qu'il y a conflit entre les deux taux, et on ajoute à oa_k le candidat en tête de la file $q_{B\overline{R}}$.

2.1.2 Invariants

Pour démontrer les propriétés (Q1)-(Q5), il suffit de montrer que l'algorithme maintient les invariants suivants.

On fixe $1 \leq k \leq n$ et on suppose que l'ordre d'appel est $oa_k = d_1, \ldots, d_k$.

On dit qu'il reste des boursiers après $d_1 \ldots d_{i-1}$ si

$$\sharp_B(d_1 \ldots d_{i-1}) < |B| \ .$$

(IQ0) La concaténation de $d_1, \ldots, d_k$ et de $q_{BR}, q_{\overline{B}R}, q_{B\overline{R}}, q_{\overline{BR}}$ est une permutation de $(c_i)_{i \in 1 \ldots |C|}$.

(IQ1) Pour tout $i \leq k$, si il reste des boursiers après $d_1 \ldots d_{i-1}$ alors le taux boursier est respecté par $d_1 \ldots d_i$.

(IQ2) Soit Z_C l'ensemble des indices $\ell \in 1 \ldots k$ tels qu'il y a conflit entre les taux après $d_1 \ldots d_{\ell-1}$. Alors tous les candidats $d_\ell, \ell \in Z_C$ sont boursiers non-résidents. Soit $i \leq k$, alors $|Z_C \cap 1 \ldots i| < 1 + q_B * i$. Et $\sharp_R(d_1 \ldots d_i) \geq \min(q_R * i, i - |Z_C \cap 1 \ldots i|)$.

(IQ3.0) Soit $1 \leq i \leq k$. Fixons un type $t \in \{BR, B\overline{R}, \overline{B}R, \overline{BR}\}$. Tout candidat de $d_1, \ldots, d_i$ de type t est mieux classé que tout candidat du même type qui n'apparaît pas dans $d_1, \ldots, d_i$.

(IQ3.1) Soit $1 \leq i \leq k$. Tout candidat de $d_1, \ldots, d_i$ est mieux classé que tout candidat boursier résident qui n'apparaît pas dans $d_1, \ldots, d_i$.

(IQ3.2) Soit $1 \leq i \leq k$. Tout candidat non-boursier de $d_1, \ldots, d_i$ est mieux classé que tout candidat non-boursier résident qui n'apparaît pas dans $d_1, \ldots, d_i$.

(IQ3.3) Soit $1 \leq i \leq k$. Tout candidat non-résident de $d_1, \ldots, d_i$ est mieux classé que tout candidat boursier non-résident qui n'apparaît pas dans $d_1, \ldots, d_i$.

(IQ3.4) Soit $1 \leq i \leq k$. Tout candidat non-boursier non-résident de $d_1, \ldots, d_i$ est mieux classé que tout candidat qui n'apparaît pas dans $d_1, \ldots, d_i$.

(IQ4.1) Soit Z_R l'ensemble des indices $\ell \in 1 \ldots k$ tels que le taux de résidents était contraignant pour le choix de d_ℓ et que d_ℓ est résident. Soit $1 \leq i \leq k$. Alors $|Z_R \cap 1 \ldots i| \leq \lceil q_R * i \rceil$. Les candidats $(d_\ell)_{\ell \in 1 \ldots i \setminus Z_R}$ sont tous mieux classés que tous les boursiers n'apparaissant pas dans $d_1 \ldots d_i$.

(IQ4.2) Soit Z_B l'ensemble des indices $\ell \in 1 \ldots k$ tels que le taux de boursiers était contraignant pour le choix de d_ℓ (et donc d_ℓ est boursier). Soit $1 \leq i \leq k$. Alors $|Z_B \cap 1 \ldots i| \leq \lceil q_B * i \rceil$. Les candidats $(d_\ell)_{\ell \in 1 \ldots i \setminus Z_B}$ sont tous mieux classés que tous les résidents n'apparaissant pas dans $d_1 \ldots d_i$.

(IQ4.3) Les candidats $(d_\ell)_{\ell \in 1 \ldots i \setminus (Z_B \cup Z_R)}$ sont tous mieux classés que tous les candidats n'apparaissant pas dans $d_1 \ldots d_i$.

La preuve des invariants est disponible dans l'appendice A.1.

2.1.3 Preuve mathématique des propriétés (Q1)-(Q5)

Preuve de (Q1). (Q1) est impliquée par l'invariant (IQ1)

Preuve de (Q2). Immédiat d'après IQ2.

Preuve de (Q3). D'après (IQ3.1) tous les candidats de $d_1 \ldots d_{i-1}$ ont un meilleur classement que d_i. Donc il y en a au plus $\text{rang}(d_i) - 1$ donc $\text{rang}(d_i) \leq i$.

Preuve de (Q4.1). Soit $1 \leq k \leq |C|$ tel que d_k est non-boursier résident. Dans le préfixe $d_1 \ldots d_{k-1}$, d'après (IQ4.2) il peut y avoir soit des candidats mieux-classés que d_k, soit des candidats de $(d_\ell)_{\ell \in Z_B}$. Donc $k \leq 1 + (\text{rang}(d_k) - 1) + \lceil q_B * (k-1) \rceil = \text{rang}(d_k) + \lceil q_B * (k-1) \rceil$. Donc $\lfloor (1 - q_B) * k + q_B \rfloor \leq \text{rang}(d_k)$. Donc $k \leq 1 + \frac{1}{1-q_B} \text{rang}(d_k)$.

Preuve de (Q4.2). Similaire à (Q4.1), en s'appuyant sur (IQ4.1).

Preuve de (Q5). Soit $1 \leq k \leq |C|$ tel que d_k est non-boursier non-résident . Dans le préfixe $d_1 \ldots d_{k-1}$, d'après (IQ4.3) il peut y avoir soit des candidats mieux-classés que d_k, soit des candidats de $(d_\ell)_{\ell \in Z_B \cup Z_R}$. D'après (IQ4.1) et (IQ4.2), $k \leq 1 + (\text{rang}(d_k) - 1) + \lceil q_R * (k-1) \rceil + \lceil q_B * (k-1) \rceil = \text{rang}(d_k) + \lceil q_R * (k-1) \rceil + \lceil q_B * (k-1) \rceil$. Donc $k + \lfloor -q_R * (k-1) \rfloor + \lfloor -q_B * (k-1) \rfloor \leq \text{rang}(d_k)$ donc $k - q_R * (k-1) - q_B * (k-1) < 2 + \text{rang}(d_k)$ donc $k < 1 + \frac{1 + \text{rang}(d_k)}{1 - (q_R + q_B)}$ donc d'après le Lemme 7, $k \leq \left\lceil \frac{1 + \text{rang}(d_k)}{1 - (q_R + q_B)} \right\rceil$.

2.2 Preuve formelle de l'implémentation WhyML

Why3 est une plateforme pour la preuve de programme, qui s'appuie sur des prouveurs externes automatiques ou interactifs [4]. Elle permet de prouver des programmes écrits en WhyML, un langage de spécification et de programmation. Une fois prouvé, le programme peut être automatiquement traduit en un programme Caml correct par construction. Dans cette section on donne un aperçu de l'implémentation WhyML et des preuves formelles associées, qui sont disponibles sur le dépôt https://gitub.u-bordeaux.fr/parcoursup/why3.

L'implémentation de l'algorithme à deux taux en WhyML est une traduction assez proche de l'implémentation Java. La figure 1 donne la traduction de la fonction `calculerOrdreAppel`[4] en Why3. Nous rappelons ici les étapes de l'algorithme:

1. partitionner le classement pédagogique en quatre files stockées dans un objet `java.util.EnumMap` indexée par les identifiants symboliques (*enum*) des files; par exemple `BOURSIER_HORS_SECTEUR`.
2. Tant qu'il reste des candidats à insérer dans l'ordre d'appel:
   (a) sélectionner le meilleur candidat éligible;
   (b) retirer ce candidat de sa file;
   (c) ajouter ce candidat à l'ordre d'appel;
   (d) mettre à jour les compteurs relatifs aux nombres de candidats.

Nous décrivons ci-dessous, sans entrer dans les détails du langage, l'implémentation en WhyML de l'algorithme présentée figure 1:

- La ligne 1 est le prototype de la fonction: elle reçoit un classement pédagogique `g` et retourne l'ordre d'appel sous la forme d'un file.
- Les lignes 2 et 3 sont les pré-conditions de la fonction: le classement pédagogique `g` est trié et non vide.
- Les lignes 4 à 9 sont les propriétés de la spécification garanties par cette implémentation (voir la section 1.4).
- Les lignes 10 et 11 sont deux propriétés accessoires qui expriment que l'ordre d'appel contient le même nombre de candidats que le classement pédagogique (ligne 11) et que l'absence de taux implique que l'ordre d'appel est le classement pédagogique (ligne 10).
- Après les initialisations des variables locales (lignes 13 à 18), la ligne 19 est le partitionnement du classement pédagogique en quatre files triées. Nous revenons sur ce point dans la section 2.2.1.
- Enfin les lignes 21 à 49 forment la boucle principale de l'algorithme:
  - extraction du `meilleur` candidat (parmi ceux restant) par l'appel de la fonction `selectionner-MeilleurVoeu` (lignes 35 à 37);
  - insertion du `meilleur` candidat dans la file résultat (ligne 39);
  - mise à jour des compteurs (lignes 41 à 47).

Les lignes 22 à 26 ne font pas partie de l'implémentation; elles indiquent à l'outil Why3 des propriétés qui restent satisfaites à chaque itération de la boucle. Ces propriétés sont utilisées par l'outil pour prouver que l'algorithme vérifie bien les propriétés $Q1$ à $Q5$ données dans la spécification. Parmi ces *invariants*, l'expression `invariants_algo2 g ordreAppel filesAttente` exprime qu'à chaque itération, le classement pédagogique `g`, l'ordre d'appel courant `ordreAppel` et les quatre files courantes présentes dans `filesAttente`,

[4] voir le module algo.mlw

vérifient les invariants $IQ1, \ldots, IQ4.3$ utilisés dans la preuve mathématique. Le prédicat `invariants_algo2`[5] est une conjonction des propriétés spécifiées dans les sections 1.4 et 2.1.2[6].

Figure 1: Implémentation en `Why3` de la fonction calculerOrdreAppel

2.2.1 Partionnement du classement pédagogique

L'implémentation Java commence par le partitionnement du classement pédagogique en quatre files (`java.util.Queue`) de candidats stockées dans une table (`java.util.EnumMap`) indexée par les quatre types de candidats:

```
EnumMap<VoeuClasse.TypeCandidat, Queue<VoeuClasse>> filesAttente
```

La bibliothèque standard de l'outil `Why3`[3] possède des modules `Map` et `Queue` que nous aurions pu exploiter pour implémenter une structure de données proche de la version Java. Toutefois, dans le contexte de la preuve de programmes, une telle structure aurait certainement complexifié inutilement les preuves. Nous avons opté pour une structure plus *ad hoc* contenant explicitement les quatre files[7]:

```
type enummap =
{
  bds : Q.t voeu;
  bhs : Q.t voeu;
  nbds : Q.t voeu;
  nbhs : Q.t voeu;
}
```

Par exemple la file stockée dans le champ `bds` contient les boursiers du secteur, l'équivalent de la file Java

```
filesAttente[BOURSIER_DU_SECTEUR]
```

Nous avons aussi dû adapter des méthodes de la classe `java.util.EnumMap` à notre structure comme par exemple les méthodes `get` et `put`.

Le partitionnement du classement pédagogique est un algorithme qui nous assure certaines propriétés de la structure de données qu'il produit (`enummap`). Nous avons déjà mentionné le fait que les quatre files contiennent des candidats de même type et qu'elles sont triées selon les rangs du classement pédagogique. D'autres propriétés doivent être garanties afin de prouver l'algorithme global. Nous montrons par exemple que l'invariant IQ0 (spécifié page 8) est vrai ou que tous les éléments contenus dans les files sont bien tous distincts.

Le code de cet algorithme est donné dans le module algo_partition_groupe.mlw.

2.2.2 Preuve des invariants et propriétés de l'algorithme

L'essentiel de la preuve de l'algorithme consiste à prouver que les invariants et propriétés de l'algorithme restent satisfait à chaque itération de la boucle (figure 1, lignes 21 à 49).

La preuve est décomposée en plusieurs fichiers indépendants, correspondants aux différents invariants. Cette modularité permet de conserver une définition de l'algorithme la plus lisible possible (cf. algo.mlw). De plus cette isolation des différentes partie de la preuve améliore également la performance des solveurs automatiques en isolant les différents contextes dans lesquels les différents invariants sont prouvés.

Les prédicats vérifiés expriment des propriétés sur le classement pédagogique `g`, le résultat `ordreAppel` et les files de candidats stockées dans `filesAttente`.

Les variables `ordreAppel` et `filesAttente` ne sont modifiées que par les lignes 36 à 39 où un candidat, `meilleur`, est retiré de `filesAttente` (lors de l'appel à `selectionnerMeilleurVoeu`) pour être ajouté, ligne 39, à la fin de `ordreAppel`.

[5] voir le module algo_invariants.mlw
[6] D'autres invariants ont été ajoutés afin d'améliorer les temps de calcul de `Why3`.
[7] Cette structure est déclarée dans le module enummap.mlw de la preuve.

La préservation des prédicats est prouvée par le contrat de la fonction `selectionnerMeilleurVoeu`[8] qui spécifie que (voir la figure 2):

- si avant l'appel à cette fonction les invariants étaient vérifiés par `g`, `ordreAppel` et `filesAttente` (lignes 75-76);
- alors (lignes 77-78) le candidat choisi, `result`, est tel que les invariants restent satisfaits pour `g`, `ordreAppel` augmenté de `result` et `filesAttente` de laquelle `result` a été retiré.

Figure 2: Contrat de la fonction de selection du meilleur candidat.

La preuve de la post-condition du contrat par `Why3` nécessite l'utilisation du lemme `invariants_algo_snoc` déclaré dans le module algo_invariants.mlw. Ce lemme est un simple corollaire des lemmes qui prouvent que chaque propriété ou invariant est satisfait après l'appel à `selectionMeilleurVoeu`. Ces lemmes sont prouvés dans des modules séparés portant le nom de la propriété ou de l'invariant concerné; par exemple q1.mlw ou iq4_3.mlw.

2.2.3 Problème relatif aux alias d'objets modifiables

Le langage de programmation de `Why3` est un langage fonctionnel mais il permet l'utilisation d'objets modifiables (*mutable*). L'algorithme utilise de tels objets: les files utilisées pour partitionner le classement pédagogique. Afin de pouvoir réaliser ses preuves `Why3` n'autorise pas toujours les alias (références) d'objets modifiables[9] et dans certains cas le programme est refusé.

Souhaitant être proche du code Java, nos premières implémentations `Why3` provoquaient ce type d'erreurs. En effet, afin d'éviter des duplications de code, nous utilisions l'accesseur `get` (du module `enummap`) pour accéder à une file *via* une variable contenant un type de candidat. Ces appels à `get` provoquait des conflits d'alias rejetés par `Why3`.

Nous avons contourné le problème en remplaçant ces appels à `get` en explicitant le type des candidats dans les opérations qui posaient problème. Ces fonctions sont implémentées dans le module safeop.mlw.

2.3 Vérification à l'exécution de l'implémentation `Java`

La vérification à l'exécution (runtime Model-checking) consiste à vérifier chaque exécution d'un programme afin de garantir que cette exécution est conforme à sa spécification, ou déclencher une alerte si ce n'est pas le cas, typiquement en levant une exception. Cette technique ne permet pas de fournir une garantie définitive de correction d'un algorithme, comme c'est le cas des techniques de preuve de programme vues précédemment, mais elle permet de garantir qu'en l'absence d'alerte, la spécification a bien été respectée.

L'ensemble des vérifications à l'exécution du calcul des ordres d'appel est consultable en ligne sur le dépôt public du code de Parcoursup [12]. Les vérifications à l'exécution d'autres algorithmes sont également consultables dans le même dossier. La vérification des propriétés (Q1)(Q3)(Q4.2)(Q5) et (P6) du calcul des ordres d'appel sont implémentées. Remarquons que (P6.0) n'est pas implémentée car dans les formations appliquant le seul taux boursier, les voeux de tous les candidats sont marqués "du secteur". La traduction en Java de ces propriétés de la spécification est une source potentielle d'erreurs, donc cette traduction est effectuée de la manière la plus transparente possible, sans optimiser la complexité algorithmique de la vérification. Les propriétés (Q2) et (Q4.1), apparues dans la version 2021 de [11, Section 4] ont vocation à être ajoutées au module de vérification à l'exécution pour 2022. En revanche, la vérification à l'exécution des propriétés (P4) et (P5) n'est pas implémentée car une traduction directe en Java aurait un coût algorithmique trop élevé, dû à la quantification universelle sur toutes les permutations possibles.

Voici le code Java implémentant la vérification à l'exécution de la propriété (Q1). Ce code de vérification est exécuté après chaque calcul d'un ordre d'appel.

[8] Le code de la fonction `selectionnerMeilleurVoeu` est donné dans le module algo_selection_meilleur_voeu.mlw
[9] Voir le rapport *A Pragmatic Type System for Deductive Verification*[7].

```
/* Verif de P1. Pour tout k, au moins qB pourcents des k premiers candidats sont
    boursiers ;
 ou sinon, aucun candidat parmi Ck+1,...,Cn n'est boursier.
*/
    void verifierP1(GroupeClassement g) throws VerificationException {
        /* on reordonne les voeux par ordre d'appel */
        g.voeuxClasses.sort(Comparator.comparingInt(VoeuClasse::getRangAppel));

        /* on verifie P1 */
        Set<Integer> candidats = new HashSet<>();
        Set<Integer> boursiers = new HashSet<>();
        boolean seulementDesNonBoursiers = false;
        for (VoeuClasse v : g.voeuxClasses) {
            candidats.add(v.gCnCod);
            if (v.estBoursier()) {
                boursiers.add(v.gCnCod);
                if (seulementDesNonBoursiers) {
                    throw new VerificationException(
                        VerificationExceptionMessage
                        .VERIFICATION_RESULTATS_ALGO_ORDRE_APPEL_VIOLATION_P1
                    );
                }
            }
            int tauxEffectif = 100 * boursiers.size() / candidats.size();
            if (tauxEffectif < g.tauxMinBoursiersPourcents) {
                seulementDesNonBoursiers = true;
            }
        }
    }
```

Voici le code Java implémentant la vérification à l'exécution de la propriété (Q5). Ce code de vérification est exécuté après chaque calcul d'un ordre d'appel.

```
 /* Un candidat non-resident non-boursier ne double jamais personne.
    Si son rang dans le classement pedagogique est r et si qB + qR < 100 alors
        son rang dans l'ordre d'appel sera inferieur ou egal a l'arrondi superieur
        de  (1 + (qB + qR) / (100 - (qB + qR)) * (r + 1) .*/
    void verifierP5(GroupeClassement g) throws VerificationException {

        for (VoeuClasse v1 : g.voeuxClasses) {
            if (!v1.estBoursier() && !v1.estDuSecteur()) {
                if(g.tauxMinDuSecteurPourcents  + g.tauxMinBoursiersPourcents <
                    100) {
                    int sommeTaux = g.tauxMinDuSecteurPourcents  +
                        g.tauxMinBoursiersPourcents;
                    long rangOrdreAppelMax = Math.round(Math.ceil((v1.rang + 1) *
                        ( 1 + sommeTaux / (100.0 - sommeTaux))));
                    if (v1.getRangAppel() > rangOrdreAppelMax) {
                        throw new
                            VerificationException(VerificationExceptionMessage.
                        VERIFICATION_RESULTATS_ALGO_ORDRE_APPEL\
                        _NON_BOURSIER_HORS_SECTEUR_DIMINUE_TROP, v1.gCnCod,
                            g.cGpCod);
                    }
                }
                for (VoeuClasse v2 : g.voeuxClasses) {
                    if (v2.rang < v1.rang && v2.getRangAppel() >
```

```
                    v1.getRangAppel()) {
                    throw new
                        VerificationException(VerificationExceptionMessage.
                    VERIFICATION_RESULTATS_ALGO_ORDRE_APPEL\
                    _NON_BOURSIER_HORS_SECTEUR_DEPASSE_CANDIDAT, v2.gCnCod,
                        v1.gCnCod, g.cGpCod);
                }
            }
        }
    }
}
```

3 Preuve des propriétés (P4)-(P6) dans les formations appliquant le seul taux boursier

Dans cette section on se concentre sur les trois propriétés (P4) (P5) et (P6) de l'algorithme de calcul de l'ordre d'appel dans les formations soumises au seul taux boursier. Tout d'abord on fournit dans la section 3.1 une preuve mathématique que l'algorithme vérifie ces propriétés. Ensuite dans la section 3.2 on présente une preuve formelle de correction de l'implémentation de l'algorithme en `WhyML`. Dans la section 3.3, on commente deux preuves formelles d'un lemme clé servant à établir (P4), l'une réalisée avec Why3 et l'autre avec Coq.

Rappelons la spécification de l'algorithme dans les formations soumises au seul taux boursier.

Dans Parcoursup, le taux minimum de résidents ne s'applique qu'aux formations dites *non-sélectives*, c'est-à-dire à une partie des licences. Dans la plupart des formations, le taux q_R est donc fixé à 0%, en d'autres termes il n'y a pas de contraintes sur ce taux, seul le taux boursier est pris en compte. Dans ce cas, la spécification se complète de deux propriétés supplémentaires.

Spécification du calcul de l'ordre d'appel dans les formations soumises au seul taux boursiers.

P4) Supposons $q_R = 0$. Comparé au classement pédagogique, l'ordre d'appel minimise le nombre d'inversions (distance de Kendall-tau), parmi ceux qui garantissent la propriété P1.

P5) Supposons $q_R = 0$. Si l'on munit l'ensemble des permutations de candidats de l'ordre lexicographique induit par les classements, alors l'ordre d'appel est le maximum parmi toutes les sélections qui garantissent la propriété P1.

P6) Supposons $q_R = 0$.

P6.0) Le résultat du calcul doit être indépendant du fait qu'un candidat soit résident ou non.

Les propriétés (Q1) à (Q5) définies ci-dessus sont satisfaites, en supposant tous les candidats résidents. Par conséquent:

P6.1) Pour tout k, l'ensemble de candidats $d_1, \ldots, d_k$ contient au moins $q_B * k$ candidats boursiers ; ou sinon, aucun candidat parmi $d_{k+1} \ldots d_{|C|}$ n'est boursier.

P6.2) Un candidat boursier qui a le rang r dans le classement pédagogique n'est jamais doublé par personne et aura donc un rang inférieur ou égal à r dans l'ordre d'appel.

P6.3) Un candidat non boursier qui a le rang r dans le classement pédagogique ne double jamais personne et aura un rang dans l'ordre d'appel compris entre r et $1+\lfloor r(1+q_B/(1-q_B))\rfloor$.

La propriété P6.0 garantit que le résultat du secteur est indépendante du fait que les candidats sont considérés comme étant du secteur ou non. La propriété P6.1 garantit le respect des taux boursiers. Les propriétés P6.2 et P6.3 spécifient dans quels cas et quelles proportions un candidat peut perdre des places lors du calcul de l'ordre d'appel, par rapport à son rang initial dans le classement pédagogique. Les propriétés P4 et P5 offrent une garantie de minimalité de la modification du classement sous contrainte de respecter le taux boursier.

Quand le taux minimum de résidents q_R est fixé à 0%, l'algorithme de calcul des ordres d'appel se simplifie de la manière suivante. On note q_B le taux minimum de boursiers, avec $0 \leq q_B \leq 1$.

Algorithme de calcul de l'ordre d'appel dans les formations soumises au seul taux boursier

1. Notons n le nombre de candidats apparaissant dans le classement pédagogique de la formation.
2. Pour chaque entier k de 1 à n, dans cet ordre, le candidat C_k de rang k dans l'ordre d'appel est calculé de la manière suivante. On a déjà sélectionné les candidats $C_1, \ldots, C_{k-1}$ dans l'ordre d'appel, et parmi eux il y a b boursiers. On dit que le taux minimum boursiers est contraignant si $b/k < q_B$. On considère tous les autres candidats, pris dans l'ordre pédagogique. Pour choisir C_k, parmi ceux-là:
   - Si le taux n'est pas contraignant, on prend le premier candidat.
   - Si le taux est contraignant, on prend le premier candidat boursier s'il y en a, le premier candidat sinon.

Par exemple, dans une formation avec un taux minimum de boursiers de 20%, et qui a établi un classement $(C_1, C_2, B_3, C_4, C_5, C_6, C_7, C_8, C_9, B_{10}, C_{11})$, où seuls deux candidats B_3 et B_{10} sont boursiers, l'ordre d'appel sera $(B_3, C_1, C_2, C_4, C_5, B_{10}, C_6, C_7, C_8, C_9, C_{11})$. En effet, le taux est contraignant lors du choix des candidats d'indices 1 et 6, ce qui permet à B_3 et B_{10} de "remonter" dans l'ordre d'appel. Remarquons que le choix du candidat d'indice 11 est également contraignant, mais à ce stade du calcul de l'ordre d'appel tous les boursiers ont déjà été sélectionnés.

Clairement l'algorithme à un taux coïncide avec l'algorithme à deux taux dans le cas où $q_R = 0$. A l'inverse, on peut généraliser l'algorithme à un nombre arbitraire de taux, mais il faut définir une hiérarchie dans la priorité donnée à l'application des différents taux.

3.1 Preuve mathématique

Soit C l'ensemble des candidats partitionné en deux sous-ensembles B les candidats boursiers et $\overline{B}$ les candidats non-boursiers.

L'algorithme du calcul de l'ordre d'appel avec un seul taux prend en entrée un taux minimum de candidats boursiers $q_B \in [0, 1]$ et un classement pédagogique rang : $C \to 1 \ldots |C|$ des candidats. On suppose que les rangs dans le classement pédagogique sont des entiers consécutifs à partir de 1 sans ex-aequo, c'est-à-dire que $\text{rang}(C) = \{1 \ldots |C|\}$. Ce classement induit une permutation $(c_i)_{i \in 1 \ldots |C|}$, c'est l'unique permutation telle que $\forall i \in 1 \ldots |C|, \text{rang}(c_i) = i$.

3.1.1 Formalisation de l'algorithme à un taux

L'algorithme commence par extraire de $(c_i)_{i \in 1 \ldots |C|}$ les permutations correspondantes $(b_i)_{i \in 1 \ldots |B|}$ des boursiers et $(nb_i)_{i \in 1 \ldots |\overline{B}|}$ des non-boursiers. Elles sont caractérisées par

- $(b_i)_{i \in 1 \ldots |B|}$ est une permutation de B,
- $\forall 1 \leq i \leq j \leq |B|, \text{rang}(b_i) \leq \text{rang}(b_j)$.

- $(nb_i)_{i\in 1\dots|\overline{B}|}$ est une permutation de $\overline{B}$,
- $\forall 1 \leq i \leq j \leq |\overline{B}|, \text{rang}(nb_i) \leq \text{rang}(nb_j)$.

Le calcul est effectué en $n+1$ étapes $0, 1, \dots, n$. Au début de l'étape k, la permutation $d_1 \dots d_k$ est déjà calculée, ainsi que deux indices $r_k(B) \in 0 \dots |B|$ et $r_k(\overline{B}) \in 0 \dots |\overline{B}|$. Initialement $k = 0$ et $r_0(B) = 0$ et $r_0(\overline{B}) = 0$. Le calcul se termine au début de l'étape $k = n$.

Les deux indices $r_k(B)$ et $r_k(\overline{B})$ sont des pointeurs dans les permutations $(b_i)_{i\in 1\dots|B|}$ et $(nb_i)_{i\in 1\dots|\overline{B}|}$ qui sont mis-à-jour de manière à garantir les invariants

$$B \cap \{d_1, \dots, d_k\} = \{b_1, \dots, b_{r_k(B)}\}$$
$$\overline{B} \cap \{d_1, \dots, d_k\} = \{nb_1, \dots, nb_{r_k(\overline{B})}\} \ .$$

Le choix de d_{k+1} est fait à l'étape k comme suit.

On dit que le taux effectif de boursier est *contraignant* après $d_1 \dots d_k$, si ajouter un non-boursier à la permutation ferait passer le taux de boursiers en dessous du minimum fixé alors qu'il reste un boursier à sélectionner c'est-à-dire si

$$(r_k(B) < |B|) \text{ et } (r_k(B) < q_B * (k+1)) \ .$$

Le choix de d_{k+1} est fait parmi un ou deux candidats *éligibles*. L'ensemble des candidats éligibles est calculé comme suit. Si il reste un boursier non-sélectionné, i.e. si $r_k(B) < |B|$, alors le candidat $b_{r_k(B)+1}$ est éligible. De plus, si le taux n'est pas contraignant après $d_1 \dots d_k$ et qu'il reste un non-boursier non-sélectionné, i.e. si $r_k(\overline{B}) < |\overline{B}|$, alors le candidat $nb_{r_k(\overline{B})+1}$ est éligible.

On choisit pour d_{k+1} le candidat le mieux classé pédagogiquement parmi les candidats éligibles.

- Si c'est le boursier $d_{k+1} = b_{r_k(B)+1}$ alors $r_{k+1}(B) = r_k(B) + 1$ et $r_{k+1}(\overline{B}) = r_k(\overline{B})$.
- Si c'est le non-boursier $d_{k+1} = nb_{r_k(\overline{B})+1}$ alors $r_{k+1}(B) = r_k(B)$ et $r_{k+1}(\overline{B}) = r_k(\overline{B}) + 1$.

3.1.2 Invariants

Pour démontrer les propriétés, on montre tout d'abord que l'algorithme maintient les invariants suivants.

Pour tout $k \in 0 \dots |C|$ on note

$$g(k) = \mid B \cap \{d_1, \dots, d_k\} \mid \ .$$

Donc le taux est contraignant après $d_1, \dots, d_k$ si

$$(g(k) < |B|) \text{ et } (r_k(B) < q_B * (k+1)) \ .$$

L'algorithme satisfait les invariants suivants.

(I0) $\forall 1 \leq i \leq k$, $g(i) = r_i(B)$.

(I0') $\forall 1 \leq i \leq k$, si le taux n'est pas contraignant après $d_1, \dots, d_{i-1}$ alors d_i est le candidat le mieux classé parmi $C \setminus \{d_1 \dots d_{i-1}\}$. Si le taux est contraignant après $d_1, \dots, d_{i-1}$ alors d_i est le candidat le mieux classé parmi $B \setminus \{b_1 \dots b_{g(i-1)}\}$

(I1) $d_1 \dots d_k$ est une permutation de $\{b_1 \dots b_{r_k(B)}\} \cup \{nb_1 \dots nb_{r_k(\overline{B})}\}$,

(I2) $\forall 1 \leq i \leq k$, si d_i et d_k sont de même type alors $\text{rang}(d_i) \leq \text{rang}(d_k)$.

(I3) $\forall 1 \leq i \leq k$, si d_k est boursier alors $\text{rang}(d_i) \leq \text{rang}(d_k) \leq k$.

(I4) $\forall 1 \leq i \leq k$, si d_k est non-boursier et $\text{rang}(d_i) > \text{rang}(d_k)$ alors d_i est boursier et le taux est contraignant après $d_1, \dots, d_{i-1}$.

(I5) $\forall 1 \leq i \leq k$, si d_i est non-boursier et $x_i = \mid \{j \in 1 \dots i \mid \text{rang}(d_j) > \text{rang}(d_i)\} \mid$ alors $x_i < q_B * i + 1$.

(I6) $\forall 1 \leq i \leq k$, si le taux est contraignant après $d_1, \ldots, d_i$ alors $i < |C|$ et $d_{i+1} \in B$.

(I7) $\forall 1 \leq i \leq k$, si $g(i) < q_B * i$ alors $g(i) = |B|$.

La preuve de ces invariants est élémentaire.

3.1.3 Preuve des propriétés à partir des invariants

Propriété (P6.0). Cette propriété est obtenue par induction en utilisant l'invariant (I0').

Propriété (P6.1). Soit $k \in 1 \ldots |C|$ tel que $g(k) < q_B * k$. Alors $g(k) = |B|$ d'après (I7) . Donc d'après (I1),

$$\forall k \leq i \leq |C|, B = \{d_1 \ldots d_i\} \cap B \ .$$

donc par comptage, et puisque $(d_k)_{k \in 1 \ldots |C|}$ est une permutation,

$$\forall k < i \leq |C|, d_i \notin B \ .$$

Propriété (P6.2). Immédiate par (I3).

Propriété (P6.3). D'après (I5) $x_k < q_B * k + 1$. D'après (I2) et (I3), $1 \ldots \text{rang}(d_k) \subseteq \text{rang}(\{d_1 \ldots d_k\})$. Donc $k = \text{rang}(d_k) + x_k$. Finalement $k < \text{rang}(d_k) + q_B * k + 1$ donc $\text{rang}(d_k) \leq k < 1 + \text{rang}(d_k)/(1 - q_B)$, i.e. (P3).

Propriété (P5). Soit $(d'_k)_{k \in 1 \ldots |C|}$ une autre permutation de C différente de $(d_k)_{k \in 1 \ldots |C|}$ et qui respecte la propriété $(P1)$. Soit k le plus petit indice où les deux permutations diffèrent. On veut montrer que $\text{rang}(d_k) < \text{rang}(d'_k)$.

Il y a deux cas, selon que le taux est contraignant ou non après $d_1, \ldots, d_{k-1}$.

Si le taux n'est pas contraignant après $d_1, \ldots, d_{k-1}$, alors d'après (I0'), d_k est le candidat le mieux classé parmi $C \backslash \{d_1 \ldots d_{k-1}\}$ et puisque $\{d_1 \ldots d_{k-1}\} = \{d'_1 \ldots d'_{k-1}\}$ alors $d'_k \in C \backslash \{d_1 \ldots d_{k-1}\}$ donc en particulier d_k est mieux classé que d'_k i.e. $\text{rang}(d_k) < \text{rang}(d'_k)$.

Si le taux est contraignant après $d_1, \ldots, d_{k-1}$ alors d'après (I0'), d_k est le candidat le mieux classé parmi $B \setminus \{b_1 \ldots b_{g(k-1)}\}$. De plus d'_k est nécessairement boursier car le taux est contraignant et $(d'_k)_{k \in 1 \ldots |C|}$ respecte $(P1)$ donc puisque $\{d_1 \ldots d_{k-1}\} = \{d'_1 \ldots d'_{k-1}\}$ alors $d'_k \in B \setminus \{b_1 \ldots b_{g(k-1)}\}$ et en particulier d_k est mieux classé que d'_k i.e. $\text{rang}(d_k) < \text{rang}(d'_k)$.

Propriété (P4). On utilise le *nombre d'inversions* ou indice de Kendall-Tau défini pour toute permutation $(e_k)_{k \in 1 \ldots |C|}$ des candidats par

$$\text{Inversions}((e_k)_{k \in 1 \ldots |C|}) = \mid \{i \in 1 \ldots |C|, j \in 1 \ldots |C|, \\ (i < j \wedge \text{rang}(e_i) > \text{rang}(e_j)) \vee (i > j \wedge \text{rang}(e_i) < \text{rang}(e_j))\} \mid$$

Soit $(d''_k)_{k \in 1 \ldots |C|}$ qui minimise le nombre d'inversions parmi toutes les permutations qui respectent (P1).

On veut montrer que $(d_k)_{k \in 1 \ldots |C|} = (d''_k)_{k \in 1 \ldots |C|}$. A contrario, supposons que ce soit faux, et soit k le plus petit indice où les deux permutations diffèrent:

$$d_1, \ldots, d_{k-1} = d'_1, \ldots, d''_{k-1}$$
$$d_k \neq d''_k \ .$$

Montrons que $(d''_k)_{k \in 1 \ldots |C|}$ vérifie la propriété (I2) avec $k = |C|$. Remarquons qu'échanger deux boursiers ou deux non-boursiers préserve $(P1)$. Donc si (I2) n'était pas satisfaite on pourrait échanger les deux

candidats correspondants et diminuer ainsi le nombre d'inversions, contredisant de ce fait la minimalité de $(d''_k)_{k\in 1\dots|C|}$.

Et $(d''_k)_{k\in 1\dots|C|}$ vérifie la propriété (I3) avec $k = |C|$, par le même genre de raisonnement.

Posons $d = d_k$ et $d'' = d''_k$. Puisque les $(d_i)_{i\in 1\dots|C|}$ sont deux à deux distincts, que $d_1, \dots, d_{k-1} = d'_1, \dots, d''_{k-1}$ et que $d_k = d \neq d'' = d''_k$ alors il existe $k_0 \in k+1\dots|C|$ et $k''_0 \in k+1\dots|C|$ tel que

$$d = d_k = d''_{k''_0}$$
$$d'' = d''_k = d_{k_0} \ .$$

Il y a trois cas selon les caractéristiques des candidats d et d''.

- Supposons d et d'' de même type. D'après (I2), d est le mieux classé des candidats de sa catégorie dans $|C| \setminus \{d_1, \dots, d_{k-1}\}$. Et $d's$ est le mieux classé des candidats de sa catégorie dans $|C| \setminus \{d''_1, \dots, d''_{k-1}\} = |C| \setminus \{d_1, \dots, d_{k-1}\}$. Donc $d = d''$.

- Supposons d boursier. Puisque $(d_i)_{i\in 1\dots|C|}$ et $(d''_i)_{i\in 1\dots|C|}$ respectent toutes deux la contrainte $P1$ et puisqu'elles coincident sur leurs prefix de longueur $k-1$ alors d'après (P5) $\text{rang}(d_k) < \text{rang}(d''_k)$. Donc $\text{rang}(d''_{k''_0}) = \text{rang}(d_k) < \text{rang}(d_{k_0}) = \text{rang}(d''_k)$. C'est une contradiction avec (I3) car $k < k''_0$ et $d = d''_{k''_0}$ est boursier.

- Reste le cas où d n'est pas boursier et d'' est boursier. Que se passerait t'il si l'on échangeait le boursier $d'' = d''_k$ avec le non-boursier $d = d''_{k_0}$ et dans $(d''_k)_{k\in 1\dots|C|}$? ? Notons $(d'''_i)_{i\in 1\dots|C|}$ cette nouvelle permutation.

  Déjà, cela diminuerait le nombre d'inversions, i.e.

  $$\text{Inversions}((d'''_i)_{i\in 1\dots|C|}) < \text{Inversions}((d''_i)_{i\in 1\dots|C|}) \ .$$

  en effet, $\text{rang}(d''_{k_0}) = \text{rang}(d_k) < \text{rang}(d_{k_0}) = \text{rang}(d''_k)$ où l'inégalité centrale est obtenue en appliquant (I3) à $(d_i)_{i\in 1\dots|C|}$. On conclut en appliquant le lemme suivant:

  Lemma 2. *Soit $(e_k)_{k\in 1\dots n}$ un vecteur de E et $0 \leq i_0 < j_0 \leq n$ une inversion de ce vecteur. Soit e' le vecteur obtenu en échangeant e_{i_0} et e_{j_0}. Alors le nombre d'inversions de e' est inférieur à celui de e.*

  On montre maintenant que la nouvelle permutation $(d'''_i)_{k\in 1\dots|C|}$ respecterait $(P1)$, ce qui contredira la minimalité de $(d''_i)_{k\in 1\dots|C|}$ et terminera la preuve.

  Pour toute permutation des candidats $(e_i)_{i\in 1\dots|C|}$ et tout $\ell \in 1\dots|C|$, notons $CB((e_i)_{i\in 1\dots|C|}, \ell)$ la propriété

  > l'ensemble de candidats $e_1, \dots, e_i$ contient au moins $q_B * i$ candidats boursiers ; ou sinon, aucun candidat parmi $e_{i+1} \dots d_{|C|}$ n'est boursier.

  Ce qui fait que pour montrer que $(d'''_i)_{k\in 1\dots|C|}$ satisfait $(P1)$ il suffit de montrer $CB((d'''_i)_{i\in 1\dots|C|}, \ell)$ pour tout $\ell \in 1\dots|C|$.

  Le cas $\ell \in 1\dots k-1$ est trivial car $(d_i)_{k\in 1\dots|C|}$ satisfait $(P1)$ et $d_1 \dots d_{k-1} = d''_1 \dots d''_{k-1}$.

  Passons au cas $\ell = k$. Prouvons tout d'abord que l'ensemble de candidats $d_1 \dots d_{k-1}$ contient au moins $q_B*(k+1)$ candidats boursiers. En effet $(d_i)_{i\in 1\dots|C|}$ satisfait (P1) et que d_{k_0} est boursier donc l'ensemble de candidats $d_1, \dots, d_k$ contient au moins $q_B * (k+1)$ candidats boursiers et on conclut puisque $d = d_k$ est non-boursier. Puisque $d_1 \dots d_{k-1} = d'''_1 \dots d'''_{k-1}$ alors l'ensemble de candidats $d'''_1, \dots, d'''_k$ contient au moins $q_B * (k+1)$ candidats boursiers donc $CB((d'''_i)_{i\in 1\dots|C|}, \ell)$ est bien vérifiée.

  Passons maintenant au cas $\ell \in k+1\dots k_0 - 1$. Pour cela montrons au préalable qu'il n'y a que des boursiers dans la suite $d''_k, \dots, d''_{k''_0-1}$. Soit $\ell \in k\dots k''_0 - 1$ et supposons a contrario que d''_ℓ est

non-boursier. D'après (I2), appliqué à $(d_i)_{i\in 1\dots|C|}$ le candidat d est le non-boursier le mieux classé parmi $C \setminus \{d_1 \dots d_{k-1}\}$. Et par (I2) appliquée à $(d''_i)_{i\in 1\dots|C|}$, le candidat d''_ℓ est le non-boursier le mieux classé parmi $C \setminus \{d''_1 \dots d''_{\ell-1}\}$. Or $\{d_1 \dots d_{k-1}\} \subseteq \{d''_1 \dots d''_{\ell-1}\}$ donc $\text{rang}(d) \leq \text{rang}(d''_\ell)$. Donc $\text{rang}(d''_{k''_0}) \leq \text{rang}(d''_\ell)$ et $\ell < k''_0$, contradiction avec (I2). Finalement, il n'y a bien que des boursiers dans la suite $d''_k, \dots, d''_{k''_0-1}$

Montrons $CB((d'''_i)_{i\in 1\dots|C|}, \ell)$ pour $\ell \in k+1 \dots k_0 - 1$. Comme montré plus haut, l'ensemble de candidats $d'''_1, \dots, d'''_k$ contient au moins $q_B * (k+1)$ candidats boursiers donc l'ensemble de candidats $d'''_1, \dots, d'''_\ell$ contient au moins $q_B * (k+1) + (\ell - k)$ candidats boursiers et on conclut puisque $q_B * (k+1) + (\ell - k) \geq q_B * (k+1+\ell-k) = q_B * (1+\ell)$

On termine en montrant que $CB((d'''_i)_{i\in 1\dots|C|}, \ell)$ pour $\ell \in k_0 \dots |C|$. Puisque $(d''_i)_{i\in 1\dots|C|}$ satisfait (P1) alors $CB((d''_i)_{i\in 1\dots|C|}, \ell)$ est vraie. Les ensembles de candidats $d''_1, \dots, d''_\ell$ et $d'''_1, \dots, d'''_\ell$ coincident à permutation près, donc ils contiennent le même nombre de boursiers et de plus si il n'y a aucun boursier dans $d''_{\ell+1}, \dots, d''_{|C|}$ alors il n'y en a aucun dans $d'''_{\ell+1}, \dots, d'''_{|C|}$. Donc $CB((d'''_i)_{i\in 1\dots|C|}, \ell)$ est bien vérifiée.

3.2 Preuve formelle de l'implémentation `WhyML`

Why3 est une plateforme pour la preuve de programme, qui s'appuie sur des prouveurs externes automatiques ou interactifs [4]. Elle permet de prouver des programmes écrits en `WhyML`, un langage de spécification et de programmation. Une fois prouvé, le programme peut être automatiquement traduit en un programme `Caml` correct par construction.

L'implémentation en `WhyML` est celle présentée dans la section 2.2. Cette section commente la preuve de P4), la propriété établissant que l'ordre d'appel minimise le nombre d'inversions parmi les permutations du classement satisfaisant P1), qui est la plus difficile à établir. L'ensemble des preuves est disponible dans le dépôt https://gitub.u-bordeaux.fr/parcoursup/why3.

3.2.1 Preuve de P4) en `Why3`

La propriété (P4) de l'algorithme à un taux s'appuie sur un lemme clé concernant le nombre d'inversions dans un vecteur, le lemme 2 page 18. Ce lemme clé est reformulé sous le forme du lemme `inversions_dec` en `WhyML`, de la manière suivante (on regroupe ici plusieurs définitions `WhyML` disponibles dans différents fichiers):

```
type pair = {
  i: int;
  j: int;
}

let function mkp (i j : int) : pair = { i=i; j=j }

predicate inversion (s: seq voeu) (p : pair) =
  (0 <= p.i < p.j < length s) && (voeu_lt s[p.j] s[p.i])

function  inversions_count  (s: seq voeu) (p : pair) : int =
  if inversion s p then 1 else 0

function sum1d (f : pair -> int) (nj : int) : (int -> int)
  = fun i -> sum ( fun j ->  f (mkp i j)) 0 nj

function sum2d (f : pair -> int) (ni nj : int) : int =
    sum (sum1d f nj) 0 ni

function nb_inversions (s: seq voeu) : int =
```

```
  sum2d (inversions_count s) (length s) (length s)

let function swap (s:seq voeu) (p : pair) : seq voeu
    requires { 0 <= p.i < p.j < length s }
  = Seq.create (length s) (fun k -> s[if k = p.i then p.j else if k = p.j then
     p.i else k])

let lemma  inversions_dec (s : seq voeu) (p : pair)
  requires { 0 <= p.i && 0 <= p.j }
  requires { inversion s p }
  ensures { nb_inversions (swap s p) < nb_inversions s }
= ()
```

Pour démontrer ce lemme, on construit une fonction ϕ de l'ensemble des inversions du vecteur e' dans celles du vecteur e et on montre que cette fonction est une injection stricte (i.e. non surjective) d'où le lemme. Une autre preuve de ce même lemme, déclinée en COQ, est présentée dans la section suivante.

Définition de l'injection. On note $I = 1 \dots n$ l'ensemble des indices. On note également $I_0 = [0 \dots i_0 - 1]$ et $J_1 = [j_0 + 1 \dots n]$. Soit $\phi : I^2 \to I^2$ la fonction qui:

- permute les points du segment $I_0 \times \{i_0\}$ avec ceux du segment $I_0 \times \{j_0\}$, en conservant intacte la première coordonnée;
- permute les points du segment $\{i_0\} \times J_1$ avec ceux du segment $\{j_0\} \times J_1$, en conservant intacte la seconde coordonnée;
- et qui laisse les autres points en place.

La fonction ϕ a trois propriétés intéressantes:

i) c'est une involution (i.e. $\phi \circ \phi$ est l'identité), donc en particulier une bijection entre les inversions de e et leur image par ϕ.

ii) si (i, j) est une inversion de e' alors $\phi(i, j)$ est une inversion de e.

iii) le point fixe $(i_0, j_0) = \phi(i_0, j_0)$ est une inversion de e mais pas de e'

En WhyML la définition de la fonction et ses propriétés clés s'énoncent comme suit.

```
let function phi (p0 p: pair) : pair
  requires { p0.i >= 0 && p0.j >= 0 }
  =
  (
  if p.i < p0.i && p.j = p0.i then mkp p.i p0.j
  else if p.i < p0.i && p.j = p0.j then mkp p.i p0.i
  else if p.i = p0.i && p0.j < p.j then mkp p0.j p.j
  else if p.i = p0.j && p0.j < p.j then mkp p0.i p.j
  else p
  )

predicate in_rect (p : pair) (ni nj : int) = 0<= p.i < ni && 0<= p.j < nj

predicate is_involution (phi : pair -> pair ) (ni nj : int)
  = forall p. in_rect p ni nj ->
  (in_rect (phi p) ni nj /\ phi (phi p) = p)

let lemma phi_is_involution (p0 : pair) (n : int)
 requires { p0.i >= 0 && p0.j >= 0 }
```

```
 ensures {  is_involution (phi p0) n n }
  =()

let lemma inversions_inc (s : seq voeu) (p0 : pair)
  requires { p0.i >= 0 && p0.j >= 0 }
  requires { inversion s p0 }
  ensures { forall p. inversion (swap s p0) p -> inversion s (phi p0 p) }
  =()
```

Pour un mathématicien, le lemme se déduit immédiatement de ces propriétés: ϕ injecte strictement les inversions de e' dans celles de e d'où le lemme. En why3 il faut travailler un peu plus et prouver deux lemmes clés.

Deux lemmes clés sur les sommes doubles. Les lemmes concernent l'opérateur de somme sur le rectangle `sum2d` défini comme suit.

Le premier lemme établit une propriété de stricte monotonie de `sum2d`:

```
let lemma sum2d_lt (f g : pair -> int) (ni nj : int) (p0 : pair)
  requires { forall p. in_rect p ni nj -> f p <= g p  }
  requires { in_rect p0 ni nj /\ f p0 < g p0 }
  ensures { sum2d f ni nj < sum2d g ni nj }
```

Le second lemme établit l'invariance de `sum2d` quand on applique aux indices une involution du rectangle.

```
let  lemma sum_permut (f : pair -> int) (g : pair -> pair) (ni nj : int)
  requires { 1 <= ni /\ 1 <= nj }
  requires { is_involution g ni nj }
  ensures { sum2d f ni nj = sum2d ( f @ g ) ni nj }
```

Le lemme `sum_permut` s'établit à l'aide d'une induction qui a pour cas de base le cas d'une transposition:

```
let function (@) (f : pair -> int) (phi : pair -> pair)
  = fun p -> f (phi p)

let function transpose (p0 p1 : pair) : (pair -> pair)
  = fun p -> if pair_eq p p0 then p1 else if pair_eq p p1 then p0 else p

let function compose (phi1 phi2 : pair -> pair) : (pair -> pair)
  = fun p -> phi1 (phi2 p)

let lemma sum_transp (f f' : pair -> int) (p0 p1 : pair) (ni nj : int)
  requires { in_rect p0 ni nj }
  requires { in_rect p1 ni nj }
  requires { pair_lt p0 p1 }
  requires { f' = f @ transpose p0 p1 }
  ensures { sum2d f ni nj = sum2d f' ni nj }
```

Les lemmes `sum_lt` et `sum_transp` sont prouvés en utilisant un opérateur `sum_path` qui permet de manipuler les sommes en 2D de manière plus fine que ne le permet `sum2d`. Cet opérateur permet de réaliser une somme partielle d'une fonction définie sur le rectangle en restreignant la somme à un chemin défini par un point de départ (inclu) et un point d'arrivée (exclu) et qui énumère de manière lexicographique tout les points intermédiaires du rectangle:

```
(*** predecessor in the lexicographic ordering in the rectangle ni nj ***)
let function prec (p : pair) (nj : int) : pair =
  if p.j = 0 then (mkp (p.i-1) (nj-1))
  else (mkp p.i (p.j-1))

(*** partial 2d sum in the rectangle ni nj along the path
```

```
between pmin (included) and pmax (excluded) ***)
let rec function sum_path (f : pair -> int) (ni nj : int) (pmin pmax: pair)
requires { in_rect pmin ni nj }
requires { in_rect pmax ni nj \/ (pmax.i = ni /\ pmax.j = 0) }
requires { not pair_lt pmax pmin }
variant { pmax.i, pmax.j }
=
if pair_eq pmin pmax then 0
else f (prec pmax nj) + sum_path f ni nj pmin (prec pmax nj)
```

Remarques. La preuve complète en why3 fait environ 600 lignes et est rejouée en moins d'une minute. Si l'on exclut les preuves consacrées aux deux lemmes génériques `sum2d_lt` et `sum_permut` qui concernent les doubles sommes sur le rectangle (stricte monotonicité et invariance par permutation), alors la preuve restante spécifique au problème du nombre d'inversions fait moins de 100 lignes.

Concernant le lemme `sum_permut` on pourrait utiliser un lemme plus faible qui établit la même propriété pour les compositions de transpositions (données explicitement), avec une induction sur le nombre de transpositions.

Par curiosité et pour être complet, on peut mesurer le travail effectué par les SMT solveurs en prouvant "à la main" la propriété

- "si (i, j) est une inversion de e' alors $\phi(i, j)$ est une inversion de e

mentionnée plus haut (item ii) dans les propriétés de ϕ). Soit (i, j) une inversion de e'. Pour montrer que $\phi(i, j)$ est une inversion de e on doit considérer de manière un peu laborieuse six cas différents, ce qui montre l'efficacité des SMT solveurs dans ce genre de preuve par énumération de cas. Tout d'abord les quatre cas où (i, j) est un point fixe de ϕ:

- si ni i ni j n'est égal à i_0 ou j_0 alors $e_i = e'_i$ et $e_j = e'_j$ donc $(i, j) = \phi(i, j)$ est une inversion de e également;
- si $i = i_0$ et $j = j_0$ alors (i, j) n'est pas une inversion de e' (mais c'en est une dans e par hypothèse);
- si $i \in \{i_0, j_0\}$ et $j < j_0$ alors puisque $i < j$ nécessairement $i = i_0$ et on a bien $e_i = e_{i_0} > e_{j_0} = e'_{i_0} = e'_i > e'_j = e_j$;
- si $j \in \{i_0, j_0\}$ et $i_0 < i$ alors puisque $i < j$ nécessairement $j = j_0$ et on a bien $e_i = e'_i > e'_j = e'_{j_0} = e_{i_0} > e_{j_0} = e_j$;

Enfin les deux cas où (i, j) n'est pas un point fixe de ϕ. Pour $k \in \{i_0, j_0\}$ on note $k' = j_0$ si $k = i_0$ et $k' = i_0$ si $k = j_0$.

- si $i \in \{i_0, j_0\}$ et $j_0 < j$. Alors $\phi(i, j) = (i', j)$ et on a bin $i' < j$ et $e_{i'} = e'_i > e'_j = e_j$;
- si $j \in \{i_0, j_0\}$ et $i < i_0$. Alors $\phi(i, j) = (i, j')$ et on a bien $i < j'$ et $e_i = e'_i > e'_j = e_{j'}$.

3.3 Preuve alternative en COQ d'un lemme clé pour établir (P4)

Coq est un système de gestion des preuves formelles [6]. Il fournit un langage formel permettant d'écrire des définitions mathématiques, des algorithmes exécutables et des théorèmes, et fournit également un environnement de preuves semi-interactive vérifiées par ordinateur. Les applications typiques sont la certification des propriétés des langages de programmation (e.g. le projet de certification du compilateur CompCert, la "Verified Software Toolchain for verification of C programs", ou le framework Iris pour la logique de séparation concurrente), la formalisation des mathématiques (e.g. la formalisation complète du théorème de Feit-Thompson theorem), et l'enseignement.

La preuve COQ utilise un angle différent de la preuve en Why3: au lieu de construire une injection non-surjective de l'ensemble des inversions du vecteur e' dans celles du vecteur e, on définit explicitement une partition du rectangle. Les preuves complètes sont disponibles sur le dépôt https://gitub.u-bordeaux.fr/parcoursup/why3.

Le but est de prouver

```
Theorem p4_dec_inv'vc :
  forall (s:seq voeu) (i:Numbers.BinNums.Z) (j:Numbers.BinNums.Z),
  ((0%Z <= i)%Z /\ (i < j)%Z /\ (j < (length s))%Z) /\
  ((voeu_lt (get s j) (get s i)) = Init.Datatypes.true) ->
  ((nb_inversions (swap s i j)) < (nb_inversions s))%Z.
```

La preuve utilise une définition de `nb_inversions` comme somme de 6 fonctions décroissantes au cours d'un swap (une de ces fonctions étant strictement décroissante au cours de cet échange). Ces fonctions s'obtiennent en considérant des restrictions à des sous-ensembles de $\mathbb{Z} \times \mathbb{Z}$, complétées par un tas de zéros. On retrouve les six cas mentionnés à la fin de la section 3.2, pour prouver la propriété ii) de l'involution ϕ, et que les SMT solveurs de Why3 parviennent à énumérer automatiquement.

Cette preuve contient les parties suivantes :

- Définition de sommes partielles à partir de prédicats décidable sur $\mathbb{Z} \times \mathbb{Z}$.
- Définition d'une partition de l'ensemble des couples d'indices dans la suite considérée.
- Preuves de décroissance du nombre d'inversions.

Fonctions et sommes partielles

```
Class dec_set  (P : int -> int -> Prop) :=
  {decider:forall k l,  {P k l} + {~ P k l}}.

Arguments decider {P} dec_set _ _.
```

On peut alors définir des versions partielles des trois fonctions de comptage de nombre d'inversions.

```
Definition countP {P}(X : dec_set  P) s k l :=
  if decider X  k l then inversion_count s k l else 0.

Definition inversions_forP {P} (X: dec_set P) s k :=
  sum (fun l => countP X s  k l) 0 (length s).

Definition inversions_P {P} (X : dec_set P) s :=
  sum (fun k => inversions_forP X s k) 0 (length s).
```

Partitions Nous définissons de façon *ad-hoc* des partitions à deux et à trois éléments.

```
(** * Partitions of int * int *)

Class partition  (P Q R : int -> int -> Prop) :=
  {
    p_cases :forall k l,  P k l -> {Q k l} + {R k l};
    p_disj : forall k l,  P k l -> Q k l -> R k l -> False;
    p_inc1 : forall k l,  Q k l -> P k l;
    p_inc2 : forall k l,  R k l -> P k l
  }.
```

```
(** We use also a partition into 3 subsets *)

Class partition3 (P Q R S: int -> int -> Prop) :=
  {
    p3_cases :forall k l,  P k l -> {Q k l} + {R k l} + {S k l};
    p3_disj1 : forall k l,  P k l -> Q k l -> R k l -> False;
    p3_disj2 : forall k l,  P k l -> Q k l -> S k l -> False;
    p3_disj3 : forall k l,  P k l -> R k l -> S k l -> False;
    p3_inc1 : forall k l,  Q k l -> P k l;
    p3_inc2 : forall k l,  R k l -> P k l;
    p3_inc3 : forall k l,  S k l -> P k l;
  }.
```

Étant donnés une partition de P en Q et R et un décideur pour P, nous pouvons construire un décideur pour Q (resp. R). De même pour les partitions à 3.

Par exemple:

```
 (** deriving  deciders along partitions  *)

Instance p_dec1 {P Q R} (decP : dec_set P)
         (p: partition P Q R) : dec_set Q.
```

L'intérêt des partitions réside en les lemmes suivants, qui permettent de décomposer en somme le calcul du nombre d'inversions :

```
Lemma countpart {P Q R}(p: partition P Q R)(X : dec_set P) s :
  forall k l, countP X s k l =
              countP (p_dec1 X p ) s k l +
              countP (p_dec2 X p ) s k l.

Lemma inversionsforpart {P Q R}(p: partition P Q R)(X : dec_set P) s k:
  inversions_forP X s k =
  inversions_forP (p_dec1 X p ) s k +
  inversions_forP (p_dec2 X p ) s k .

Lemma inversionspart {P Q R}(p: partition P Q R)(X : dec_set P) s :
  inversions_P X s  =
  inversions_P (p_dec1 X p ) s  +
  inversions_P (p_dec2 X p ) s  .
```

L'étude de cas On considère données une suite s_0, et deux positions $i < j$ telles que $s_0[j] < s_0[i]$. Soit s_1 la suite obtenue par échange des contenus en i et j.

```
Section case_study.

  (** Let us consider inversion count before and after a swap on s0
      resulting in s1 (at positions i j) *)


  Variables (s0 : seq voeu)
            (i j : int).
  Hypotheses (Hi : 0 <= i < j)
             (Hj : j < length s0)
```

```
            (Hinv : voeu_lt (get s0 j) (get s0 i) = true).

  Let s1 := swap s0 i j.
```

On veut prouver l'inégalité suivante:

```
 Lemma global : nb_inversions s1 < nb_inversions s0.
```

Cas de base On considère les 7 ensembles de couples (k, l) suivants.
Nous ne nous intéressons qu'aux couples (k, l) d'indices vérifiant $0 \leq k < l < |s_0|$.

```
Definition full k l  := 0 <= k < l  /\ l < length s0.
```

Notons que ce prédicat permet de relier le comptage original des inversions et nos sommes partielles.

```
Instance  dec_full  : dec_set full.

Lemma inversions_full s  :
     length s = length s0 ->
     inversions_P  dec_full s  = nb_inversions s.
```

A est le singleton contenant uniquement le couple (i, j). Le nombre d'inversions decroit strictement dans la transformation de s_0 en s_1. Nous appelons `dec_A` le décideur associé à A.

```
Definition A k l := k = i /\ l = j.

Lemma L003 k l  : A k l ->
                    countP dec_A s1 k l <
                    countP dec_A s0 k l .

Lemma inversions_forA_le k  : 0 <= k < length s0 ->
                                inversions_forP dec_A s1 k <=
                                inversions_forP dec_A s0 k.

Lemma inversions_A : inversions_P dec_A s1 < inversions_P dec_A s0.
```

C contient tous les couples (k, l) tels que $\{k, l\} \cap \{i, j\} = \emptyset$. On prouve que le nombre d'inversions dans ce cas reste constant, donc décroissant au sens large.

```
Let B k l := full k l /\  k <> i /\ l <> j.
Definition C k l := B k l /\  (k <> j /\ l <> i).

Lemma count_C  k l : C k l ->
   countP dec_C s1 k l =  countP dec_C s0 k l.

Lemma inversions_for_C k :
   inversions_forP dec_C s1 k  = inversions_forP dec_C s0 k .

Lemma inversions_C :
   inversions_P dec_C s1 = inversions_P dec_C s0.
```

D contient tous les couples (k, l) de `full` tels que $k < i$ et $l = i \vee l = j$. Au cours d'un swap, le nombre d'inversions restreint à D reste constant.

```
Lemma L015 k : 0 <= k < length s0  ->
                 inversions_forP dec_D s1 k =
                 inversions_forP dec_D s0 k.

Lemma L016 : inversions_P dec_D s1  = inversions_P dec_D s0.
```

E voudrait jouer un rôle sysmétrique de celui de D, puisqu'il est décrit par la formule $j < l$ et $k = i \vee k = j$.

Le problème est que l'on ne peut avoir de lemme correspondant à `L015`, car `inversions_forP` compte les inversions à k constant.

Pour résoudre ce problème, on crée deux fonctions travaillant "à l constant".

```
Definition inversions_forX {P} (X : dec_set P) s l :=
  sum (fun k => countP X s  k l) 0 (length s).

Definition inversionsX {P} (X : dec_set P) s :=
  sum (fun l =>  inversions_forX X s l) 0 (length s).
```

Et on prouve (grâce à `double_sum`), l'équivalence des deux formes de comptage.

```
Lemma inversionsX_eq P (X : dec_set P) s :
  inversionsX X s = inversions_P X s.
```

On arrive donc à:

```
Lemma L0150 l : 0 <= l < length s0  ->
                  inversions_forX dec_E s1 l =
                  inversions_forX dec_E s0 l.

Lemma L0160 : inversionsX dec_E s1  = inversionsX dec_E s0.

Lemma L0161 : inversions_P dec_E s1 = inversions_P dec_E s0.
```

F

```
Definition F k l := full k l /\ l = j /\ i < k < j.
```

Dans ce cas, l'échange des éléments d'indice i et j (avec $s_0[j] < s_0[i]$) entraîne (par transitivité de l'ordre sur les voeux) que si $s_0[k] \leq s_0[l] = s_0[j]$, alors $s_1[k] = s_0[k] \leq s_1[i] < s_1[j] = s_1[l]$. Donc le nombre d'inversions entre k et l diminue (au sens large), car s'il est égal à 0 pour s_0, il sera aussi nul pour s_1.

```
Lemma F_global : inversions_P dec_F s1 <= inversions_P dec_F s0.
```

G Le traitement suivant est similaire à celui de F.

```
Definition G k l := full k l /\ k = i /\ i < l < j.
```

```
Lemma G_global : inversions_P dec_G s1 <= inversions_P dec_G s0.
```

Preuve finale Les 6 ensembles de A à G forment une partition de `full`. En appliquant les relations entre partitions et nombre d'inversions, on arrive au résultat suivant.

```
Lemma global : nb_inversions s1 < nb_inversions s0.
```

4 Conclusion

Dans ce document on a présenté les différentes mesures de vérification des spécifications de l'algorithme des ordres d'appel: preuve mathématique, preuve formelle et vérification à l'exécution. Ces mesures permettent d'atteindre un niveau élevé de confiance.

Un autre algorithme crucial pour le fonctionnement de Parcoursup est l'algorithme qui calcule quotidiennement quelles propositions doivent être envoyées aux candidats. Tout comme le calcul des ordres d'appel, cet algorithme est documenté dans le "Document de présentation des algorithmes de Parcoursup" [11, Sections 3,5,6] et l'implémentation `Java` est publiée sur le dépôt public du code de Parcoursup [12]. Des mesures de vérification à l'exécution ont été implémentées en Java pour garantir la sécurité du code de l'algorithme des propositions mais cet algorithme n'a pas encore été prouvé formellement, notamment l'unicité du résultat pour les formations avec internat ou de manière équivalente l'indépendance du résultat à l'ordre d'énumération des données d'entrée.

La preuve formelle de l'implémentation `Java` a été partiellement effectuée [3] mais pas encore finalisée. Pour avancer dans cette direction, il y a au moins deux possibilités. Premièrement contourner le problème en générant des implémentations de référence `Caml` ou `Java` à partir des implémentations `WhyML`. Cela pose un problème de maintenabilité du code. Deuxièmement définir un fragment syntaxique de Java qui se prête à une interprétation en WhyML avec une extension de l'outil `jmltowhy3` développé au LRI [3].

References

[1] ACSL. Ansi/iso c specification language. https://frama-c.com/html/acsl.html, 2021.

[2] Léo Andrès. Vérification par preuve formelle de propriétés fonctionnelles d'algorithme de classification. Research report, Université Paris Sud (Paris 11) - Université Paris Saclay, https://hal.inria.fr/hal-02421484/file/internship-report-parcoursup.pdf, August 2019.

[3] Benedikt Becker, Jean-Christophe Filliâtre, and Claude Marché. Rapport d'avancement sur la vérification formelle des algorithmes de ParcourSup. Technical report, Université Paris-Saclay, https://hal.inria.fr/hal-02447409/file/main.pdf, January 2020.

[4] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Why3. http://why3.lri.fr.

[5] François Bobot, Jean-Christophe Filliâtre, Claude Marché, and Andrei Paskevich. Let's Verify This with Why3. *International Journal on Software Tools for Technology Transfer*, 17(6):709–727, 2015.

[6] COQ. Site web de COQ. https://coq.inria.fr.

[7] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. A Pragmatic Type System for Deductive Verification. Technical report, https://hal.inria.fr/hal-01256434, February 2016.

[8] formalvindications. Site web de formalvindications. https://formalvindications.com.

[9] Liane Huttner and Denis Merigoux. Traduire la loi en code grâce au langage de programmation Catala. In *Intelligence artificielle et finances publiques*, Nice, France, October 2020.

[10] JML. Java modelling language. https://www.cs.ucf.edu/~leavens/JML/index.shtml, 2021.

[11] MESRI. Document de présentation des algorithmes de parcoursup. https://framagit.org/parcoursup/algorithmes-de-parcoursup/-/blob/master/doc/presentation_algorithmes_parcoursup_2021.pdf, 2021.

[12] MESRI. Dépôt public du code de parcoursup. https://framagit.org/parcoursup/algorithmes-de-parcoursup/, 2021.

[13] OCaml.org. Site web de OCaml. https://ocaml.org.

[14] OCDE. Rapport RaC de l'OCDE. https://oecd-opsi.org/projects/rulesascode/.

A Appendice

A.1 Preuve des invariants pour l'algorithme à deux taux

Cette partie présente la preuve des invariants pour l'algorithme à deux taux. On essaye de se placer à un niveau de détails de preuve qui est intermédiaire entre la compréhension d'un humain et la capacité des solveurs automatiques: la plupart des preuves paraîtront triviales au lecteur habitué à lire des preuves algorithmiques, mais leur traduction directe en Why3 nécessitera parfois de détailler des étapes supplémentaires pour que les solveurs puissent effectuer les preuves.

A.1.1 Preuve de la préservation de l'invariant (IQ0)

Pour chaque élément de c_i, exactement une des assertions suivantes est vraie:

- c_i apparaît une fois dans $d_1, \ldots, d_k$; ou bien
- c_i apparaît une fois dans exactement une des files $q_{BR}, q_{\overline{B}R}, q_{B\overline{R}}, q_{\overline{BR}}$.

A.1.2 Preuve de la préservation de l'invariant (IQ1)

On utilise trois lemmes.

Lemma 3. *Soit $i < k+1$. Si $i > 1$ et qu'il reste des boursiers après après $d_1 \ldots d_{i-1}$ alors il restait des boursiers après après $d_1 \ldots d_{i-2}$.*

Proof. trivial. □

Lemma 4. *Si on ajoute un boursier (resp. un résident) à une file $d_1 \ldots d_i$ respectant le taux boursier (resp. résident) alors la file résultant respecte le taux boursier (resp. résident) . Si on ajoute un candidat quelqu'il soit à une file $d_1 \ldots d_i$ dans laquelle le taux boursier (resp. résident) n'est pas contraignant, alors la file résultant respecte le taux boursier (resp. résident) , ou alors il ne reste plus de boursier (resp. de résident) après $d_1 \ldots d_i$.*

Proof. Le taux boursier est compris entre 0 et 1. □

Lemma 5. *Pour tout $1 \leq i \leq k$, Si le taux boursier est contraignant après $d_1 \ldots d_{i-1}$ alors le candidat d_i sélectionné par l'algorithme après $d_1 \ldots d_{i-1}$ est boursier*

Soit $i \in 1 \ldots k$. et supposons qu'il reste des boursiers à l'étape i.
Si $i < k$ alors on peut appliquer l'invariant.
Si $i = k$.

Premier cas: supposons que le taux boursier n'est pas contraignant après $d_1 \ldots d_{k-1}$. Alors d'après le lemme 4, le taux boursier est respecté par $d_1 \ldots d_k$.

Second cas: supposons que le taux boursier est contraignant après $d_1 \ldots d_{k-1}$. Par hypothèse il reste des boursiers après $d_1 \ldots d_{k-1}$. Donc d'après le lemme 3, il reste des boursiers après $d_1 \ldots d_{k-2}$. D'après (IQ1) à l'indice $k-1$, $d_1 \ldots d_{k-1}$ respecte le taux boursier.

Donc $d_1 \ldots d_k$ aussi d'après les lemmes 5 et 4.

A.1.3 Preuve de la préservation de l'invariant (IQ2)

On s'appuie sur le lemme suivant

Lemma 6. *Soit $1 \leq i \leq k$. Supposons que le taux résident est contraignant après $d_1 \dots d_{i-1}$. Si il n'y a pas conflit entre les taux alors le candidat d_i sélectionné par l'algorithme après $d_1 \dots d_{i-1}$ est résident. Sinon d_i est boursier non-résident.*

Prouvons (IQ2). Soit Z_C l'ensemble des indices $\ell \in 1 \dots k$ tels qu'il y a conflit entre les taux après $d_1 \dots d_{\ell-1}$. Soit $i \leq k$,

On considère trois cas distincts. Dans les deux premiers cas, $i \notin Z_C$, donc $|Z_C \cap 1 \dots i| = |Z_C \cap 1 \dots i-1|$ et la borne sur Z_C reste vraie par induction.

Premier cas: supposons que le taux de résidents n'est pas contraignant après $d_1 \dots d_{i-1}$. Par hypothèse il restait des résidents à l'étape i, donc quelque soit le choix de d_i, résident ou non, la file $d_1 \dots d_i$ respecte le taux résident et $\sharp_R(d_1 \dots d_i) \geq q_R * i$.

Second cas, supposons que le taux de résident est contraignant après $d_1 \dots d_{i-1}$ et qu'il n'y a pas conflit entre les taux à l'étape i. Alors d'après le Lemme 6, d_i est résident. Et $\sharp_R(d_1 \dots d_i) = 1 + \sharp_R(d_1 \dots d_{i-1}) \geq 1 + \min(q_R * (i-1), i - 1 - |Z_C \cap 1 \dots i-1|) \geq \min(q_R * i, i - |Z_C \cap 1 \dots i|)$.

Troisième cas, supposons qu'il y a conflit entre les taux après $d_1 \dots d_{i-1}$. Alors le taux boursier est contraignant, donc $\sharp_B(d_1 \dots d_{i-1}) < q_B i$. Or tous les candidats de Z_C sont boursiers donc $|Z_C \cap 1 \dots i-1| < q_B i$. Par conséquent, $|Z_C \cap 1 \dots i| = 1 + |Z_C \cap 1 \dots i-1| < 1 + q_B i$ D'autre part, $\sharp_R(d_1 \dots d_i) = \sharp_R(d_1 \dots d_{i-1}) \geq i - 1 - |Z_C \cap 1 \dots i-1| = i - |Z_C \cap 1 \dots i|$.

A.1.4 Preuve de préservation de l'invariant (IQ3.0)

Un candidat éligible a le plus petit rang parmi les candidats de même type n'apparaissant pas encore dans la file.

A.1.5 Preuve de préservation de l'invariant (IQ3.1)

Si $i < k$ on applique par induction l'invariant (IQ3.1). Sinon $i = k$. Si tous les boursiers résidents apparaissent dans $d_1, \dots, d_i$ il n'y a rien à prouver. Sinon il restait des boursiers résidents après $d_1 \dots d_{k-1}$, donc pas de conflit entre les taux après $d_1 \dots d_{k-1}$. Et un boursier résident fait toujours parti des candidats éligibles. Or d_k était le mieux classé des candidats éligibles, donc aussi bien classé que tous les boursiers résidents restant.

A.1.6 Preuve de préservation de l'invariant (IQ3.2)

Dans le cas $i < k$ on conclut en appliquant l'hypothèse d'induction pour (IQ3.2). Supposons $i = k$. Soit $\ell \in 1 \dots k$ tel que d_ℓ est un non-boursier résident qui n'apparaît pas dans $d_1 \dots d_k$. Si $\ell < k$ on conclut en appliquant l'hypothèse d'induction pour (IQ3.32) et l'indice $k-1$. Supposons $\ell = k$. Si d_k est boursier il n'y a rien à prouver. Sinon d_k est non-boursier. Puisque d_k est non-boursier, d'après le Lemme 5 le taux boursier n'était pas contraignant pour choisir d_k. Puisque d est résident, il faisait partie des candidats éligibles. Donc d n'a pas été sélectionné et d_k est mieux classé que d.

A.1.7 Preuve de préservation de l'invariant (IQ3.3)

Dans le cas $i < k$ on conclut en appliquant l'hypothèse d'induction pour (IQ3.3) et l'indice i. Supposons $i = k$. Soit $\ell \in 1 \dots k$ tel que d_ℓ est un boursier non-résident qui n'apparaît pas dans $d_1 \dots d_k$. Si $\ell < k$ on conclut en appliquant l'hypothèse d'induction pour (IQ3.3) et l'indice $k-1$. Supposons $\ell = k$. Si d_k est résident il n'y a rien à prouver. Sinon d_k est non-résident. Si il y avait conflit entre les taux alors d_k était le mieux classé des boursiers non-résidents, donc mieux classé que d. Sinon le taux résident n'était pas contraignant pour choisir d_k. Donc d faisait partie des candidats éligibles. Donc d_k est mieux classé que d.

A.1.8 Preuve de préservation de l'invariant (IQ3.4)

Dans le cas $i < k$ on conclut en appliquant l'hypothèse d'induction pour (IQ3.4) et l'indice i. Soit $\ell \in 1 \dots k$ tel que d_ℓ est un non-boursier non-résident mieux classé qu'un candidat d qui n'apparaît pas dans $d_1 \dots d_k$. Si $\ell < k$ on conclut en appliquant l'hypothèse d'induction pour (IQ3.4) et l'indice $k-1$. Supposons $\ell = k$. Puisque d_k est non-boursier non-résident alors aucun taux n'était contraignant après $d_1 \dots d_{k-1}$. Donc d_k était le mieux classé de tous les candidats n'apparaissant pas dans $d_1 \dots d_{k-1}$, en particulier il était mieux classé que d.

A.1.9 Preuve de préservation de l'invariant (IQ4.1)

On montre dans le cas $i = k$, car dans les cas $i < k$ on conclut immédiatement par hypothèse d'induction.

Premier cas: supposons $k \notin Z_R$. Alors $|Z_R \cap 1 \dots k| = |Z_R \cap 1 \dots (k-1)| \leq \lceil q_R * i \rceil \leq \lceil q_R * i + 1 \rceil$.

Second cas: supposons $k \in Z_R$. Donc le taux résident est contraignant après $d_1 \dots d_{k-1}$. Donc $\sharp_R(d_1 \dots d_{k-1}) < q_R * k$. Puisque tous les candidats aux indices de Z_R sont résidents, $|Z_R \cap 1 \dots (k-1)| \leq \sharp_R(d_1 \dots d_{k-1}) < q_R * k$. Et $|Z_R \cap 1 \dots k| = 1 + |Z_R \cap 1 \dots (k-1)| < 1 + q_R * k$ donc $|Z_R \cap 1 \dots k| \leq \lceil q_R * k \rceil$, en appliquant:

Lemma 7. *Soit $n \in \mathbb{N}, x \in \mathbb{R}$ tel que $n < 1 + x$. Alors $n \leq \lceil x \rceil$.*

Proof. $n < 1 + x \implies n < 1 + \lceil x \rceil \implies n \leq \lceil x \rceil$. □

Pour la seconde partie de (IQ4.1), remarquons que si le taux résidents n'est pas contraignant après $d_1 \dots d_{k-1}$ alors pour chaque candidats boursier n'apparaissant pas dans $d_1 \dots d_{k-1}$ il y a un candidat de même type parmi les candidats éligibles. Donc d_k est mieux classé que tous les candidats boursiers n'apparaissant pas dans $d_1 \dots d_{k-1}$. Et si le taux résidents est contraignant mais en conflit avec le taux boursier, c'est que les seuls boursiers restant sont non-résidents, et d_k est le meilleur d'entre eux.

A.1.10 Preuve de préservation de l'invariant (IQ4.2)

Similaire à la preuve de (IQ4.1), en s'appuyant sur (IQ4.2) et (IQ3.2).

A.1.11 Preuve de préservation de l'invariant (IQ4.3)

En l'absence de contrainte sur les taux, c'est le meilleur qui gagne.

A.2 Optimalité des bornes données dans (Q2), (Q4) et (Q5)

A.2.1 optimalité de la borne donnée dans (Q2)

On considère le cas où le premier candidat du classement, nommé c, est un résident non-boursier, et aucun résident n'est boursier.

La borne minimum donnée par (Q2) sur le nombre de résidents parmi $d_1, \dots, d_k$ est

$$\min\left(\lceil q_R * k \rceil, \lceil (1 - q_B) * k - 1 \rceil\right)$$

Dans le cas extrême où $q_B = 1$ alors c est envoyé arbitrairement loin dans l'ordre d'appel, après tous les boursiers, ce qui correspond à la borne donnée, égale à 0.

Dans le cas où $q_B = 1\%$ et $k = 1$ alors il y a conflit entre les taux lors du choix de d_1 donc d_1 est boursier non-résident et le nombre de résidents dans les préfixes d_1 et $d_1 d_2$ sont respectivement 0 et 1, ce qui correspond à la borne donnée.

Dans le cas où $q_B = 98.99\%$ et $k = 100$ alors il y a conflit entre les taux lors du choix de $d_1, \dots, d_{99}$, qui sont donc tous boursiers non-résidents, et c est 100ième dans l'ordre d'appel . Le nombre de résident dans les préfixes $d_1, \dots, d_{99}$ et $d_1, \dots, d_{100}$ est respectivement 0 et 1. Cela correspond exactement aux bornes données par la formle supra.

A.2.2 optimalité de la borne donnée dans (Q4.1)

Dans le cas où $q_B = 1\%$ et $q_R = 0\%$ et $k = 2$ et le candidat classé premier est un non-boursier résident alors il est doublé par un boursier et second dans l'ordre d'appel. Cela correspond à la borne donnée qui est de $1 + \left\lfloor 1 + \frac{1}{99} \right\rfloor = 2$.

Dans le cas où $q_B = 98.99\%$ et $q_R = 0\%$ et $k = 100$ et le candidat classé premier est un non-boursier résident alors il est doublé par 99 boursiers et se retrouve 100ième dans l'ordre d'appel. Dans ce cas la borne donnée est $1 + \left\lfloor (1 + \frac{q_B}{1-q_B})r \right\rfloor = 1 + \lfloor 99.00990099\ldots \rfloor = 100$ et est donc optimale.

A.2.3 optimalité de la borne donnée dans (Q4.2)

Similaire au cas de la borne donnée dans (Q4.1), en inversant q_R et q_B (comme $q_B = 0\%$ il n'y a pas de conflit entre les taux).

A.2.4 optimalité de la borne donnée dans (Q5)

Dans le cas où $q_B = 49.001\%$ et $q_R = 49.001\%$ et les 100 premiers candidats sont des non-boursier non-résident et les autres candidats sont ou bien boursier ou bien résidents. Soit C le 100ième candidat. Il apparaitra après les 99 candidats devant lui. Pour respecter les taux, il faudra au préalable avoir classé $n = 99 + n_B + n_R$ candidats avec $n_B \geq q_B * (n+1)$ et $n_R \geq q_R * (n+1)$ donc $n \geq 99 + q_B * (n+1) + q_R * (n+1)$ donc $n \geq (99 + q_B + q_R)/(1 - q_B - q_R)$ donc $n \geq 5004$.

Alors le candidat C est classé au mieux 5005ième dans l'ordre d'appel. La borne données par $(Q5)$ est $\left\lceil \left(1 + \frac{q_R+q_B}{1-(q_R+q_B)}\right) * 101 \right\rceil = \left\lceil \frac{101}{1-(q_R+q_B)} \right\rceil \approx 5055$.