Projet de Génie Logiciel

Système de compression de données

Master 1 - Informatique
Année 2025 - 2026

Antoine de Chabannes Curton la Palice

2 Novembre 2025

Encadrant : Jean-Charles Régin

Table des matières

1. Introduction

a. Contexte

Dans le cadre du cours de Génie Logiciel du Master 1 informatique à l'Université Côte d'Azur, il nous est demandé de concevoir un projet en appliquant les principes de la programmation orientée objet ainsi que les bonnes pratiques de conception logicielle.
Le projet porte sur la compression d'entiers dans le but d'accélérer la transmission de données.
Dans de nombreux systèmes, notamment ceux liés au réseau, aux bases de données ou au traitement massif d'informations, la taille des messages échangés représente un facteur déterminant en termes de temps de réponse et de coût énergétique.
Réduire le volume de données transmises sans perte d'information constitue donc un enjeu central de l'informatique moderne.
C'est donc dans ce contexte qu'intervient la méthode de Bit Packing, une technique consistant à représenter chaque entier sur un nombre minimal de bits.

b. Objectif du projet

L'objectif du projet est de concevoir et d'implémenter un système de compression d'entiers appelé Bit Packing, permettant de réduire la taille des données tout en conservant un accès direct à chaque élément.

Le programme doit proposer plusieurs modes de compression, mesurer leurs performances, et déterminer à partir de quelle latence réseau la compression devient avantageuse.

Enfin, le projet a pour but d'appliquer concrètement les principes du génie logiciel : conception modulaire, utilisation de design patterns, protocoles de mesure, tests et documentation claire.

2. Analyse

a. Contraintes

Le cahier des charges impose un ensemble de contraintes techniques et conceptuelles destinées à encadrer la conception et à garantir la validité du système développé.

Contraintes techniques :

- Le projet doit être implémenté en Java ou Python.
- L'algorithme de compression doit garantir un accès direct à l'élément i du tableau compressé via une méthode get(i).
- Les entiers sont stockés sur 32 bits, et la compression doit réduire cet espace à k bits par valeur, où k est calculé en fonction de la valeur maximum.
- Le système doit gérer plusieurs variantes :
    - Without Overlap : chaque valeur tient dans un mot 32 bits distinct.
    - Overlap : une valeur peut s'étendre sur deux mots consécutifs.
    - Overflow : introduction d'une zone de débordement (overflow area) pour les valeurs dépassant k.
- Une factory doit être utilisée pour gérer la création du bon type de compression à partir d'un paramètre unique (Mode)
- Des benchmarks doivent être exécutés pour mesurer les temps d'exécution afin de déterminer à partir de quel seuil la compression devient intéressante.

Contraintes d'ingénierie logicielle :

- Respect des principes SOLID et d'une architecture modulaire
- Intégration de Design Patterns
- Code clair et commenté avec documentation (Markdown + Rapport PDF)
- Structure Github organisée

b. Problématiques

Le problème à résoudre ne consiste pas simplement à implémenter un algorithme de compression. Il s'agit avant tout d'un problème d'ingénierie logicielle où il faut assembler :

- Performance : minimiser les temps de compression et de décompression
- Fiabilité : garantir la fidélité de la décompression et des accès get(i)
- Maintenabilité : structure de code claire et documenté
- Modularité : permettre l'ajout de nouveaux modes de compression facilement

La problématique centrale peut être formulée ainsi :

Comment concevoir un système de compression d'entiers efficace, modulaire et extensible, permettant à la fois d'améliorer la vitesse de transmission et de préserver l'accès direct aux données ?

Cette question constitue le fil conducteur du projet et permet de découper la démarche en quatre grands axes de travail :

1. Concevoir un système modulaire et extensible, structuré selon les principes SOLID et intégrant les Design Patterns
2. Mesurer précisément les performances de chaque variante de Bit Packing à l'aide de benchmarks fiables et reproductibles.
3. Comparer les résultats expérimentaux entre les différentes approches afin d'identifier leurs avantages respectifs.
4. Interpréter les résultats pour déterminer les conditions dans lesquelles la compression devient réellement bénéfique, et proposer des pistes d'amélioration ou d'extension.

c. Exigences

Avant toute conception, il est nécessaire d'identifier les exigences du projet. Elles se divisent en deux grandes catégories : les exigences fonctionnelles, qui décrivent les fonctions à réaliser, et les exigences non fonctionnelles, qui décrivent les qualités et contraintes du système.

Exigences fonctionnelles :

- Implémenter les trois opérations principales :
    - compress(int[] input) : compresser un tableau d'entiers
    - decompress(int[] output) : décompresser les données.
    - get(int i) : accéder à la i-ème valeur.
- La sélection du mode de compression via une factory
- la mesure automatique des performances (temps de compression, décompression, lecture)
- Génération d'un fichier csv récoltant les résultats
- Visualisation graphique du seuil de rentabilité
- Exécution en ligne de commande simplement

Exigences non-fonctionnelles :

- Être modulaire et extensible
- Être lisible et documenté

- Architecture claire selon les principes SOLID
- Le code doit être testé

3. Conception

a. Structure et organisation

Le projet est structuré en plusieurs packages cohérents et spécialisés, chacun répondant à un rôle bien défini dans le système.
Cette organisation vise à séparer les responsabilités, à faciliter les tests et à permettre l'ajout futur de nouveaux modes de compression sans modifier le code existant.

L'arborescence principale du projet est la suivante :

Dans le package main :

- **api :** Regroupe les interfaces et types du système :
  - IPacking : interface définissant les méthodes
  - Mode : enum qui liste tous les modes disponibles

- **core :** Contient les algorithmes de compression :
  - BitPacking : classe commune à :
    - BitPackingWithoutOverlap
    - BitPackingOverlap
  - BitPackingOverflow : rajout de la zone de débordement

- **factory:** Contient BitPackingFactory permet la création dynamique des objets selon le mode choisi. Application du Factory Pattern, centralise la logique de construction.

- **utils :** Regroupe les classes utilitaires telles que BitOps, qui fournit les opérations binaires bas niveau (masques, décalages, calcul de bits).

- **bench :** Dédié à l'évaluation des performances :
  - DataGenerator : génère les input à compresser.
  - BenchConfig : paramètres et profils de test.
  - BenchRunner : exécute les benchmarks et exporte les résultats au format CSV.
  - bench_plots.py : script Python pour la visualisation graphique

Dans le package test (**coverage** et **unitaire) :**

Contient les tests JUnit de couverture et de validation des implémentations. Chaque variante de BitPacking dispose de ses propres tests pour assurer la fidélité de la compression/décompression.

Et les répertoires :

- **results :** le fichier de résultat et les graphiques.
- **ressources :** les jeux d'entrée générés par DataGenerator
- **docs :** la documentation et les supports du cours

b. Diagramme UML

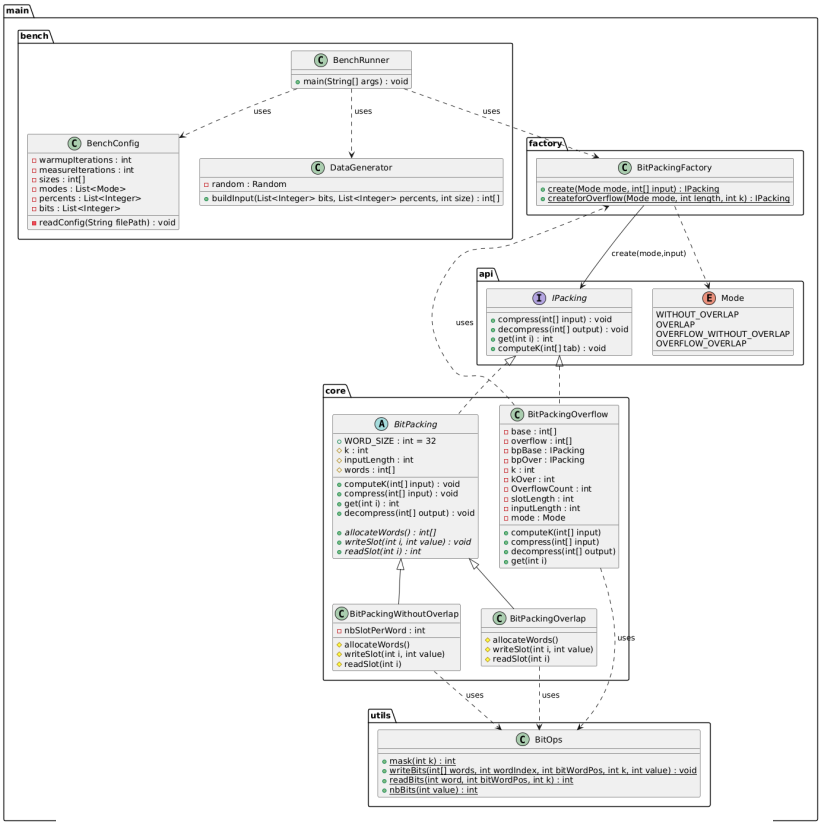


Figure 3.1 : Diagramme UML - package main

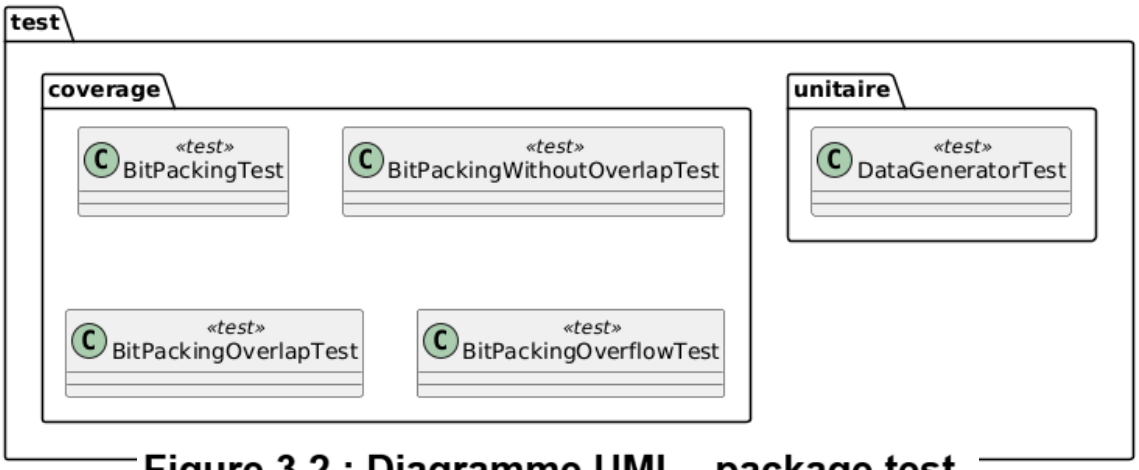


Figure 3.2 : Diagramme UML - package test

c. Justification de l'architecture

L'architecture choisie repose sur une organisation modulaire, hiérarchisée et conforme aux principes SOLID.
Elle vise à assurer la clarté, la maintenabilité et la réutilisabilité du code tout en facilitant les extensions futures (nouveaux modes de compression, nouveaux benchmarks, etc.).

Application des principes SOLID :

- S : Single Responsibility Principle (Responsabilité unique)
  Chaque classe a un rôle bien défini :
  - les classes du package core implémentent la logique de compression
  - la BitPackingFactory gère uniquement l'instanciation
  - BenchRunner et BenchConfig assurent la mesure de performance,
  - BitOps encapsule les opérations binaires.

- O : Open/Closed Principle (Ouvert/Fermé)
  Le système est ouvert à l'extension mais fermé à la modification :
  - Il est possible d'ajouter une nouvelle variante de compression (par exemple BitPackingNegatif) sans modifier le reste du code, simplement en l'enregistrant dans la BitPackingFactory.

  Le comportement global du programme reste inchangé.

- L : Liskov Substitution Principle (Substitution de Liskov)
  Toutes les variantes (WithoutOverlap, Overlap, Overflow) héritent du même contrat défini par l'interface IPacking.
  Elles peuvent donc être manipulées de manière interchangeable, ce qui simplifie le code des benchmarks et des tests.

- I : Interface Segregation Principle (Ségrégation d'interface)

L'interface IPacking définit les trois méthodes essentielles : compress, decompress et get.
Les classes clientes n'ont pas à implémenter de méthodes inutiles, assurant ainsi un découplage maximal.

- D : Dependency Inversion Principle (Inversion des dépendances)

Les classes de haut niveau comme BenchRunner, BenchConfig ou main.py ne manipulent jamais directement les classes BitPackingWithoutOverlap, BitPackingOverlap ou BitPackingOverflow.
Elles interagissent uniquement avec des objets de type IPacking, c'est-à-dire une abstraction commune

L'ensemble de ces choix architecturaux permet d'obtenir un système cohérent

- robuste : chaque composant est isolé et testable,
- évolutif : de nouvelles stratégies de compression peuvent être ajoutées sans modification des modules existants,
- réutilisable : les classes (BenchRunner, BitOps, BitPackingFactory) sont génériques et peuvent être adaptées à d'autres contextes,
- conforme au génie logiciel, illustrant concrètement l'application des principes vus en cours : abstraction, modularité, séparation des responsabilités et design patterns.

4. Design Patterns

a. Factory Pattern

Centraliser la création d'objets en fonction d'un paramètre (ici Mode) afin de découpler le code client de toute connaissance des classes concrètes.
Ce pattern permet d'éviter que le reste du programme (notamment les benchmarks et les tests) ait à connaître les détails d'instanciation des différentes variantes de Bit Packing.

Implémentation dans le projet :
la méthode principale create, qui retourne une instance de IPacking parmi :

- BitPackingWithoutOverlap
- BitPackingOverlap
- BitPackingOverflow (en Overlap ou WithoutOverlap).

Une seconde méthode, createForOverflow, est utilisée par BitPackingOverflow.
Elle permet de créer directement une instance de BitPacking adaptée à la sous-zone (base ou overflow) sans répéter les étapes déjà effectuées par la compression principale.

Problème :
Une tentative d'implémentation d'un mode AUTO, qui sélectionne la variante la plus optimale selon le contexte d'entrée a été explorée.
La seule solution trouvée consistait à pré-calculer le nombre total de bits pour chaque mode avant compression, afin d'en choisir le plus compact.
Cependant, cette approche s'est révélée difficile à rendre efficace dans la factory.
Faute de compromis satisfaisant entre complexité et performance, cette fonctionnalité a été abandonnée.

Bénéfices :

- Le haut niveau dépend de l'abstraction IPacking, et non des classes concrètes.
- L'ajout d'un nouveau mode ne nécessite aucun refactoring du code existant : il suffit d'étendre la factory.
- La logique de création est isolée, ce qui facilite les tests unitaires ciblés sur la sélection de mode.

Limites :

- Une factory trop "intelligente" devient rapidement difficile à maintenir si elle embarque de la logique d'optimisation complexe (comme le mode AUTO).
- Risque de violation du principe de responsabilité unique si la création commence à intégrer des heuristiques de performance.

Solution :
La factory a été volontairement maintenue simple, elle se limite à la création d'instances selon le mode demandé.
Si des comportements d'auto-sélection ou d'optimisation doivent être ajoutés plus tard (par exemple pour la gestion des nombres négatifs ou de nouveaux profils de données), ils seront implémentés dans un sélecteur externe dédié, plutôt que dans la factory elle-même.

b. Strategy Pattern

Encapsuler plusieurs algorithmes interchangeables derrière une même interface pour pouvoir les substituer dynamiquement.

Implémentation dans le projet :
L'interface IPacking du package api définit le contrat commun (compress, decompress, get) des variantes du package main.core

Problème :
Obligation d'avoir les mêmes scénarios (données, protocole), en évitant de parsemer le code de branches spécifiques à chaque implémentation.

Solution :
Le benchmark manipule exclusivement des références IPacking, et traite chaque instance de la même façon. Le choix est délégué à la BitPackingFactory.

Bénéfices :
- Polymorphisme : le benchmark ne change pas quand on alterne les variantes.
- Lisibilité : chaque stratégie est testée isolément
- Extensibilité : ajout d'une nouvelle variante sans changer le code existant

Risques :
- Se retrouver avec plein de classes trop proches

Mitigation :
- Factoriser ce qui est en commun (voir Template Method ci-dessous), en isolant ce qui diffère réellement.

c. Template Method Pattern

Factoriser le squelette d'un algorithme dans une classe de base, en laissant aux sous-classes l'implémentation des étapes variables.

Implémentation dans le projet :
La classe abstraite BitPacking définit le flux général des opérations de compression et décompression. Elle encapsule les invariants (calculs communs, utilitaires) et délègue les étapes spécifiques (écriture/lecture de slots, gestion overflow) aux sous classes.

Problème :
Lors des premières versions, la duplication du code entraînait des incohérences où chaque correction devait être répliquée dans plusieurs classes.

Solution :
Création d'une classe abstraite BitPacking qui :

- fixe le contrat commun
- centralise les outils partagés
- définit un squelette algorithmique clair

Ainsi, chaque variante ne redéfinit que sa logique spécifique tout en profitant du cadre général commun.

Bénéfices :

- Réduction de la duplication et cohérence accrue entre variantes.
- Lisibilité améliorée : le déroulement global de la compression/décompression est visible dans la classe mère.

Risque :

- Si trop de logique dans la classe abstraite, on perd la souplesse de ce qu'elle est censé nous apporter

Mitigation :

- Limiter la classe BitPacking aux invariants et à la structure, et déléguer le bas niveau à des utilitaires (BitOps) pour conserver la souplesse du design.

d. Facade Pattern

Consiste à fournir une interface simple et unifiée à un ensemble de composants complexes. L'utilisateur n'a pas besoin de connaître tous les détails internes, on met en place une façade qui sert de point d'entrée unique. Elle masque la complexité interne, simplifie l'exécution, et rend le système plus facile à utiliser.

Implémentation dans le projet :
Le pattern Façade est appliqué à deux niveaux :

- Côté Java - BenchRunner : Joue le rôle de façade du module de benchmarks. Elle regroupe tout le processus, du lancement du benchmark à l'enregistrement des données.
- Côté Python - Main
  Le script python agit comme une façade globale du projet : il relie les deux environnements (Java et python) et permet d'exécuter le projet avec des commandes simples.

L'utilisateur n'a pas besoin de connaître la structure interne du projet ni d'exécuter plusieurs commandes distinctes : une seule interface gère l'ensemble.

Problème :
Au départ, lancer une série de tests nécessite de :

- d'exécuter manuellement chaque mode de compression
- d'écrire les résultats à la main dans un fichier Excel
- puis d'afficher les courbes de performance séparément

Cette fragmentation rendait les tests fastidieux, sources d'erreurs et peu reproductibles.

Solution :
Mettre en place une façade combinée (BenchRunner + main.py) qui :

- orchestrent automatiquement toutes les étapes
- masquent la complexité de la configuration
- fournissent une commande unique pour l'utilisateur

Bénéfices :

- Simplicité d'utilisation : le système complet peut être exécuté sans connaissance technique avancée.
- Lisibilité du code : la logique de haut niveau est concentrée, tandis que les détails techniques restent encapsulés dans leurs packages respectifs.
- Réplicabilité : même environnement d'exécution et même protocole garantis pour tous les benchmarks.

Risque :

- Si la façade accumule trop de responsabilités (paramètres, logs, fichiers), elle peut devenir un "god object", c'est-à-dire une classe trop volumineuse, difficile à maintenir et à tester.

Mitigation :

- conserver BenchRunner léger
- déléguer la configuration à BenchConfig
- isoler la logique graphique dans bench_plots.py

5. Implémentation

a. BitPacking

Cette classe joue le rôle de classe abstraite commune à toutes les variantes du Bit Packing. Elle ne réalise pas la compression elle-même, elle structure le travail et définit les points à spécialiser pour les variantes WithoutOverlap et Overlap.
Elle fixe le comportement commun et le contrat API partagé entre toutes les implémentations :

- Comment on calcule la largeur minimale en bits k
- Comment on parcourt les données pour compresser ou décompresser
- Quelles méthodes doivent être obligatoirement fournies par les sous-classes

Autrement dit, elle décrit le "comment global", mais laisse les détails du "comment exact" à ses enfants.

Structure du code :

Attributs :

```
protected static final int WORD_SIZE = 32; // Taille d'un mot (taille "Integer" = 32)
protected int k;                           // Nombre de bits par valeur (k) (1 <= k <= WORD_SIZE)
protected int inputLength;                 // Nombre d'éléments à compresser (taille du tab "input")
protected int[] words;                     // Tableau compressée (composé de mots) (null avant compress, !null après)
```

Les variables sont communes à toutes les implémentations : c'est pour cela qu'elles sont définies ici et non répétées dans les sous-classes.

Méthodes principales :

- computeK(int[] input) : Parcourt le tableau pour déterminer la valeur maximale et donc le nombre minimal de bits nécessaires k.
  Cette méthode standardise le calcul de k pour toutes les variantes, garantissant une cohérence entre elles.

- compress(int[] input) : crée words et appelle writeSlot (implémenter dans les classes filles) pour chaque entier.

- decompress(int[] output) : fait l'opération inverse via readSlot en appelant get(i) sur chaque entier

- get(int i) : permet d'obtenir un élément (accès direct en O(1)).
  Ces méthodes sont finalisées ici : les sous-classes n'ont qu'à se concentrer sur la gestion des bits (pas sur la boucle générale).

Méthodes abstraites : chaque variante implémente sa propre logique

- allocateWords()

- writeSlot(int i, int value)
- readSlot(int i)

Méthodes utilitaires (getK, getWords) :
Destinées au debug et aux tests unitaires, elles permettent de vérifier l'état interne de la compression (valeur de k, contenu du flux compressé).

b. BitPackingWithoutOverlap

Cette classe implémente la version sans chevauchement du Bit Packing (Without Overlap). Chaque entier compressé est entièrement contenu dans un seul mot de 32 bits où aucune valeur ne déborde sur le mot suivant.

Structure du code :

Attributs :
nbSlotPerWord : nombre de valeurs stockées dans un mot ⇒ pour un k = 8, WORD_SIZE / k = nbSlotPerWord = 4

Constructeurs :
- Auto :
  - Appelé lorsqu'on compresse directement un tableau d'entiers.
  - Calcule k à partir du tableau, fixe inputLength, et déduit nbSlotPerWord.
- Fixé
  - Utilisé par la variante Overflow, où k est imposé par la zone principale (base) ou la zone de débordement (overflow).
  - Initialise directement les valeurs sans recalculer k.

Méthodes principales :
- allocateWords() :
  1. Calcule la taille nécessaire du tableau words[]
  2. Utilise la formule :
     wordsLength = (inputLength + nbSlotPerWord - 1) / nbSlotPerWord
  3. Retourne le tableau words à la taille wordsLength

- writeSlot(int i, int value) :
  1. Calcule la position absolue en bits où stocker la valeur i
  2. Utilise la petite classe interne SlotPosition pour obtenir :
     - wordIndex : numéro du mot concerné,
     - offset : position de départ du slot à l'intérieur du mot.
  3. Écrit la valeur compressée dans le bon mot avec BitOps.writeBits.

- readSlot(int i) :
    1. Calcule la même position que pour l'écriture.
    2. Lit les bits correspondants avec BitOps.readBits, puis retourne la valeur entière.

  L'accès get(i) reste direct (O(1)) et cohérent avec le protocole d'écriture.

Classe interne SlotPosition :
- Sert à factoriser le calcul d'adresse d'un slot dans le tableau compressé.
- Calcule :
    - wordIndex = i / nbSlotPerWord
    - bitWordPos = i % nbSlotPerWord
    - offset = bitWordPos * k

Cette approche est la plus simple et la plus rapide à décompresser, au prix d'une légère perte de compacité lorsque k ne divise pas exactement 32.

c. BitPackingOverlap

Cette variante du Bit Packing autorise les valeurs à chevaucher deux mots de 32 bits. Autrement dit, la compression ne s'aligne pas sur les limites des mots : chaque entier est écrit dans un flux de bits continu, ce qui maximise la compacité du stockage.
Concrètement :
- chaque slot commence à la position absolue bitPos = index * k,
- et s'il n'y a plus assez de bits dans le mot courant, la valeur est découpée entre deux mots successifs.

Structure du code :

Constructeurs :
- Auto :
    - Appelé lorsqu'on compresse directement un tableau d'entiers
    - Calcule k à partir du tableau et fixe inputLength
- Fixé :
    - Utilisé par la variante Overflow, où k est imposé par la zone principale (base) ou la zone de débordement (overflow)
    - Initialise directement les valeurs sans recalculer k

Méthodes principales :

- allocateWords() :
    1. Calcule le nombre total de bits nécessaires (k × n), puis le nombre de mots de 32 bits nécessaires pour les contenir.
    2. Le calcul est fait en long pour éviter tout risque d'overflow sur des entrées très longues.
    3. Retourne le tableau words à la taille wordsLength

  Cette allocation est exactement ajustée au flux de bits il n'y a aucun gaspillage de place contrairement à la version WithoutOverlap.

- writeSlot(int i, int value) :
    - Cas simple : le slot tient entièrement dans un seul mot. BitOps.writeBits écrit directement les k bits au bon offset.
    - Cas chevauchant : Si bitWordPos + k > WORD_SIZE
        1. on découpe value en deux parties :
            - lowerPartValue (bits restants du mot courant),
            - upperPartValue (bits à écrire au début du mot suivant),
        2. puis on écrit chaque partie dans son mot respectif.

- readSlot(int i) : La lecture reconstruit la valeur selon la même logique
    - Cas simple : la valeur est entièrement contenue dans un mot → lecture directe.
    - Cas chevauchant :
        - la fin du premier mot (lowerBits),
        - le début du suivant (upperBits),

      puis on recombine les deux morceaux

Cette version maximise la densité de compression en autorisant le chevauchement des valeurs entre deux mots.
Aucun bit n'est gaspillé : la compression est quasi optimale.
Elle constitue un excellent compromis entre compacité et performance.

d. BitPackingOverflow

Cette version est la plus avancée du système. Elle introduit la gestion d'une zone de débordement (overflow area), qui améliore la compression lorsque certaines valeurs du tableau sont beaucoup plus grandes que les autres.
Le flux compressé est donc scindé en deux parties :

- ZONE BASE : stocke la majorité des valeurs "normales" sur k bits
- ZONE OVERFLOW : stocke uniquement les valeurs trop grandes pour tenir sur k bits, référencées depuis la base.

Chaque valeur de la zone base contient un flag indiquant :

- 0 : la valeur est stockée directement dans la base
- 1 : la valeur réelle se trouve dans la zone overflow, à l'indice donné

Structure du code :

Méthodes principales :

- computeK(int[] input) : Calcul du k optimal repose sur du brute-force
    1. Construction d'un histogramme des tailles en bits nécessaires à chaque valeur freqBits[k]
    2. Calcul cumulatif du nombre de valeurs en overflow pour chaque k overflowCountByK
    3. Evaluation du coût total pour chaque candidat k :
       TotalBits(k) = n x slotLength(k) + overflowCount * kMax
       où :
        - slotLength(k) = 1 + max(k, nbBits(over-1))
          (flag + valeur ou index)
        - overflowCount = nombre d'éléments dont nbBits(val) > k
        - kMax = largeur maximale rencontrée
    4. On choisit alors le meilleur k minimisant la taille totale

- compress(int[] input) :
    1. Allocation des flux :
        - base[] de taille inputLength
        - overflow[] de taille overflowCount
    2. Parcourt le tableau d'entrée et pour chaque valeur :
        - Si elle tient sur k bits : écrit dans la base avec un flag 0
        - Sinon : écrit un flag 1 et place la valeur réelle dans overflow[]
    3. Après remplissage, délègue la compression des deux flux :
        - création via createForOverflow
        - remplissage via le compress du mode choisi

- get(int i) :
    1. Lit la valeur compressée dans la base (bpBase.get(i))
    2. Extrait le flag et la valeur/index via un masque :
        - Si flag == 0 : retourne valueOrIndex,
        - Sinon : va chercher la vraie valeur dans bpOver.get(valueOrIndex)

- decompress(int[] output) :
  Se contente de boucler sur les get(i) pour reconstruire tout le tableau

Cette approche réduit le nombre de bits moyens par valeur et rend la compression adaptative. Elle combine les variantes précédentes pour obtenir le meilleur ratio entre taille et vitesse, tout en restant totalement modulaire via la factory.

e. BenchRunner

BenchRunner est la classe de mesures : il lit chaque config de ressources/, génère des entrées représentatives, exécute les variantes via la Factory, mesure compress, decompress, et get(i), calcule les métriques réseau (paquets, latence), puis rempli et exporte un CSV unique results/results.csv.

Protocole de mesure :

1. Données : génération contrôlée par BenchConfig (distributions bits/percents, tailles sizes), reproductible via seed (DataGenerator(42)).
2. Warmup : warmupIterations appels à compress puis decompress pour stabiliser la JVM.
3. Mesures :
    - compress et decompress : moyenne de measureIterations chronométrées avec System.nanoTime().
    - get(i) : coût moyen par accès sur Q = min(1e6, size*10) tirages pseudo-aléatoires (seed fixé), avec un blackhole pour éviter les optimisations mortes.
4. Tailles :
    - Non compressé : S0bits = 32 * size.
    - Compressé : Scbits = 32 * bp.getWords().length.
5. Réseau :
    - MTU (en bits) et latence par paquet LATENCY_NS (en ns).
    - Nombre de paquets : n0 = ceil(S0bits / MTU), nC = ceil(Scbits / MTU).
6. Décision :
    - Temps total sans compression : T0 = n0 * LATENCY_NS.
    - Temps total avec compression : Tc = tComp + tDecomp + nC * LATENCY_NS.
    - Gain si Tc < T0.
    - Seuil de latence (si applicable) : t* = (tComp + tDecomp) / (n0 - nC)

BenchRunner orchestre la génération des données, l’exécution des algorithmes, la mesure des performances et l’export des résultats.

f. bench_plots.py

Ce fichier permet l'analyse et la visualisation complémentaire à la partie Java. Il permet de représenter graphiquement les résultats produits par BenchRunner, afin d'identifier le seuil de rentabilité de la compression en fonction de la latence réseau.

L'objectif est de comparer, pour un même jeu de données et un même mode de compression, le temps total de transmission :

- Sans compression : dépend uniquement du nombre de paquets à envoyer (nO) et de la latence du réseau.
- Avec compression : inclut les temps de compression, décompression, et le nombre réduit de paquets (nC).

Le script met ainsi en évidence le point où la compression devient réellement avantageuse.

Étapes principales :

1. Lecture du CSV

Le script lit le fichier results/results.csv généré par BenchRunner à l'aide de la bibliothèque pandas.

2. Extraction des paramètres

À partir de la lecture des paramètres lors de l'exécution, le script récupère :

- les temps moyens de compression et décompression (tComp_ms, tDecomp_ms)
- le nombre de paquets non compressés nO et compressés nC,
- la taille d'entrée (size).

3. Calcul du seuil de rentabilité

Le script détermine la latence t* selon la formule : t* = (tcomp + tdecomp) / (nO - nC)
C'est le point où les courbes "avec" et "sans compression" se croisent.

4. Tracé des courbes

Deux courbes sont tracées avec matplotlib :

- Rouge : transmission sans compression,
- Bleue : transmission avec compression.
- Une ligne verte en pointillés marque le seuil t*.

L'échelle du graphique est automatiquement ajustée selon la valeur du seuil (jusqu'à 200 000 ms si nécessaire).

Ce script constitue la phase finale du protocole expérimental :

- Il relie directement les mesures de performance (Java) à une interprétation visuelle .
- Il met en évidence l'impact de la latence réseau sur l'intérêt de la compression.
- Il facilite la comparaison entre les différents modes (WithoutOverlap, Overlap, Overflow…) grâce à une représentation cohérente et homogène.

Il complète ainsi le projet en offrant un outil de visualisation reproductible et conforme à la méthodologie du Software Engineering Project.

g. main.py

Le fichier main.py constitue une interface en ligne de commande (CLI) permettant d'exécuter facilement l'ensemble du projet BitPacking sans interaction manuelle complexe.
Il agit comme un point d'entrée unique pour :

- lancer les benchmarks Java et génère le CSV via l'option --bench
- afficher les résultats graphiques via l'option --plot avec --mode et --input
- générer automatiquement tous les graphiques via l'option --all

Choix de conception :

- Séparation claire entre Java et Python :

Le cœur du calcul (compression, mesure, CSV) reste en Java, tandis que la partie analyse et visualisation est gérée en Python.

- Automatisation des expérimentations :

Grâce à --all, il est possible de générer en une seule commande tous les benchmarks et graphiques du projet, garantissant une reproductibilité totale des résultats.

- Robustesse :

L'usage de subprocess.run() offre une gestion propre des processus et un contrôle clair du flux d'exécution.

L'interface en ligne de commande (main.py) a été conçue pour simplifier l'utilisation du projet et uniformiser l'exécution des différentes étapes.

6. Tests

a. Tests unitaires

Les tests unitaires ont été implémentés à l'aide du framework JUnit 5, dans le package test.unitaire.
Le fichier principal BitPackingAllTest.java vérifie le bon fonctionnement de toutes les variantes de Bit Packing.

Chaque test suit le même protocole :

1. Génération aléatoire d'un tableau d'entiers (input[]) de 1000 éléments, avec des valeurs comprises entre 2^4 et 2^{16}.
2. Création de l'implémentation appropriée via la factory (BitPackingFactory.create(mode, input)), garantissant que le test reste indépendant des classes concrètes.
3. Appel successif à compress(input) puis decompress(output) pour vérifier la bijectivité de la compression :

L'assertion assertArrayEquals(output, input), confirme que les données décompressées sont strictement identiques à celles d'origine, assurant la fidélité des opérations.

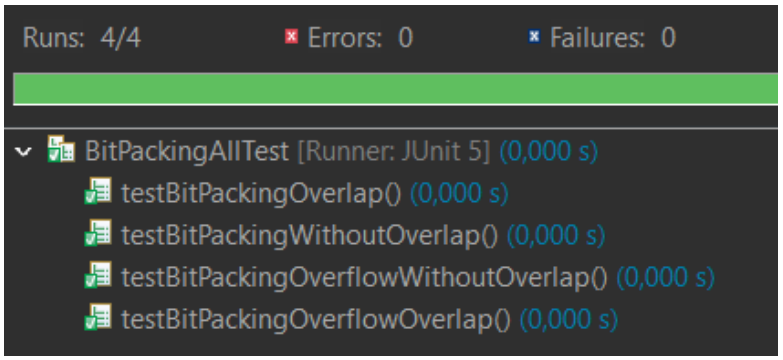
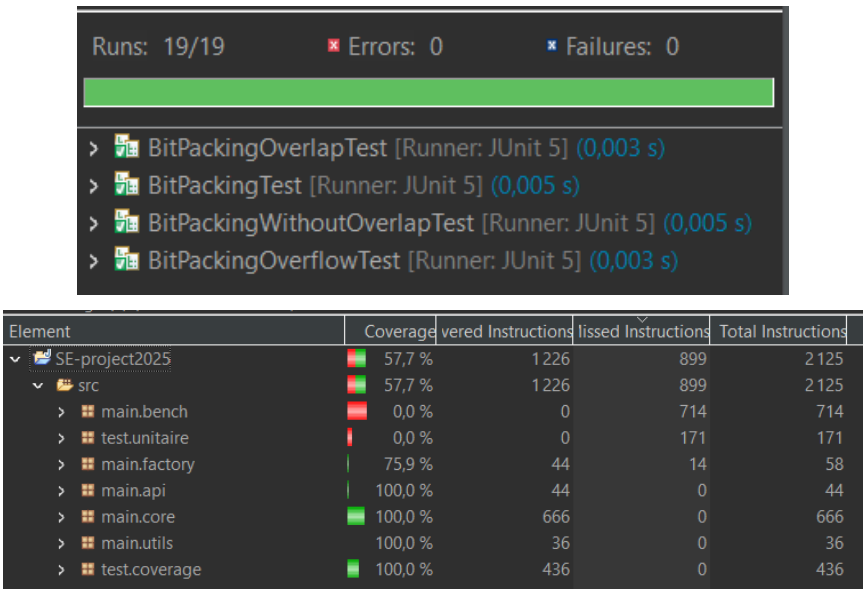


Figure 6.1 Tests unitaires

Ces tests couvrent l'intégralité des combinaisons possibles entre chevauchement et overflow, garantissant la cohérence du comportement global du système.

b. Couverture du code

Un second niveau de validation a été réalisé à l'aide des tests de couverture, situés dans le package test.couverture. Ces tests visent à mesurer quantitativement la proportion de code du package main.core réellement exécutée.



Element	Coverage	vered Instructions	lissed Instructions	Total Instructions
SE-project2025	57,7 %	1 226	899	2 125
src	57,7 %	1 226	899	2 125
main.bench	0,0 %	0	714	714
test.unitaire	0,0 %	0	171	171
main.factory	75,9 %	44	14	58
main.api	100,0 %	44	0	44
main.core	100,0 %	666	0	666
main.utils	100,0 %	36	0	36
test.coverage	100,0 %	436	0	436

Figure 6.2 Couverture du code

Les rapports de couverture montrent que l'ensemble des lignes du package core sont exécutées.

Cette couverture élevée garantit que :

- les méthodes (compress, decompress, get) ont été testées sur une variété de valeurs
- les différentes conditions (cas simples, cas chevauchants, valeurs en overflow) sont exercées
- et aucune partie du cœur algorithmique n'est laissée non testée.

7. Analyse des performances

a. Mesures de performances

Les mesures de performance ont été réalisées automatiquement grâce à la classe BenchRunner. Chaque test exécute plusieurs itérations de compression, décompression et accès direct (get(i)), sur des tableaux d'entiers de tailles variées et pour l'ensemble des modes de compression.

Le protocole adopté garantit des mesures fiables et reproductibles :

- Warmup : plusieurs exécutions "à blanc" sont effectuées avant la mesure, afin de neutraliser l'effet de la compilation Just-In-Time (JIT).
- Moyennage des mesures : chaque temps est calculé sur measureIterations répétitions, puis moyenné.
- Chronométrage : toutes les mesures utilisent System.nanoTime() pour obtenir une précision en nanosecondes.
- Paramètres constants : les tailles d'entrée, les distributions de valeurs et les seeds aléatoires sont fixées via le fichier BenchConfig, garantissant la reproductibilité totale des résultats.

Les temps mesurés sont ensuite enregistrés dans un fichier CSV unique results/results.csv

Choix des jeux d'entrée pour le benchmark :

Les jeux de données utilisés pour les benchmarks ont été conçus de manière à représenter différents profils de compression et à tester la robustesse de chaque variante.
Ils sont générés automatiquement par la classe DataGenerator, à partir des paramètres décrits dans les fichiers de configuration du dossier ressources/.

Pertinence des jeux d'entrée :

- small : contient 100% des petits entiers (k = 1) ⇒ cas idéal
- large : contient 100% de très grands entiers (k = 30) ⇒ cas extrême
- overlap : favorise la version overlap (k=10)
- without_overlap : favorise la version without_overlap (k=4)
- overflow : favorise la version overflow (k=10 80%, k=25 20%)

En combinant ces cinq jeux de tests, on obtient une vision complète du comportement du système :

- depuis les cas limites (meilleur et pire scénario),
- jusqu'à des configurations représentatives de situations réelles.

b. Résultats

Structure du fichier CSV :

- input : Nom du fichier d'entrée (jeu de test)
- mode : Variante utilisée (e.g. OVERLAP, OVERFLOW_OVERLAP, etc.)
- size : Taille du tableau compressé
- tComp_ms : Temps moyen de compression (ms)
- tDecomp_ms : Temps moyen de décompression (ms)
- nO : Nombre de paquets transmis sans compression
- nC : Nombre de paquets transmis avec compression
- gainT Booléen : compression rentable (true) ou non (false)

Ce CSV constitue la base des analyses et graphiques de performance.

Structure des graphiques :

- Axe X : latence réseau simulée (en millisecondes).
- Axe Y : temps total de transmission (en ms).
- Courbe rouge : transmission sans compression TO = nO × latence.
- Courbe bleue : transmission avec compression TC = tComp + tDecomp + nC x latence.
- Ligne verte : seuil de rentabilité t*, c'est-à-dire la latence à partir de laquelle la compression devient bénéfique.

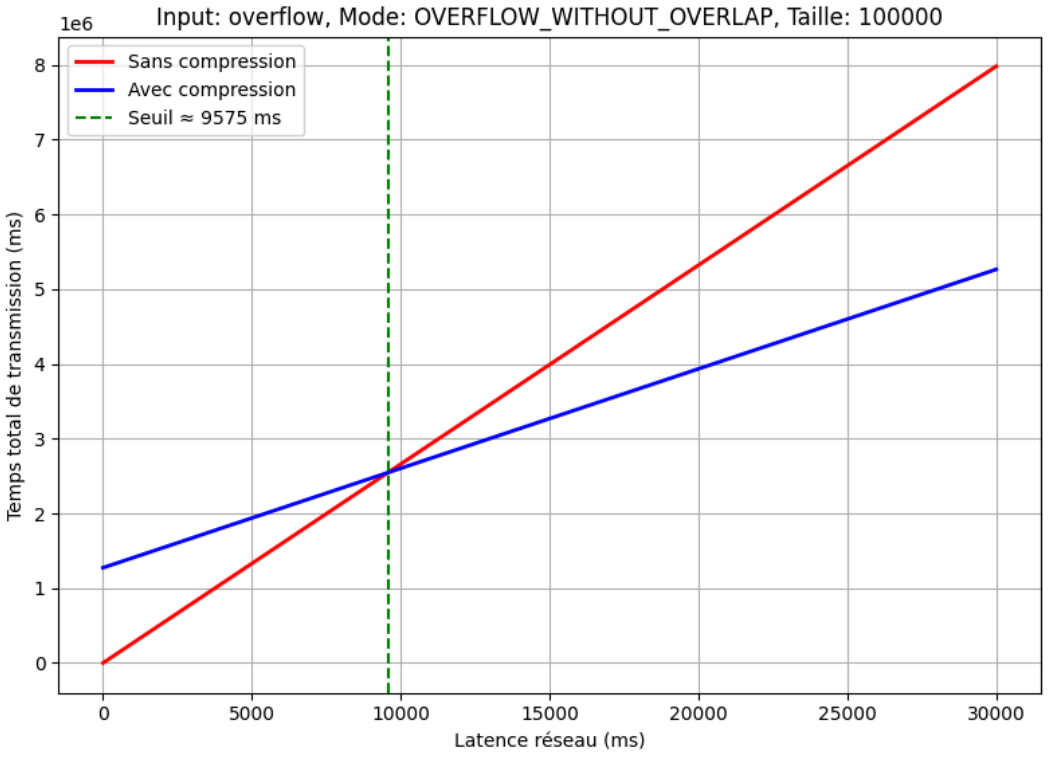


Figure 7.1 Graphique

c. Interprétation des résultats

Les graphiques générés par le script bench_plots.py illustrent l'évolution du temps total de transmission en fonction de la latence réseau pour chaque mode de compression.
Deux courbes principales y figurent :

- Rouge : Temps de transmission sans compression
  T0 = n0 x latence
- Bleue : Temps de transmission avec compression
  Tc = tcomp + tdecomp + nC x latence

Une ligne verte pointillée marque le seuil de rentabilité t*, point où les deux courbes se croisent.

Lecture du graphique :

- Pour des latences faibles, la compression est contre-productive : le coût CPU (compression + décompression) dépasse le gain en taille.
- À partir d'une certaine latence critique, la courbe bleue devient inférieure à la rouge : la compression réduit effectivement le temps total de transmission.
- Le taux de compression (écart entre nO et nC) détermine la pente de la courbe "avec compression" : plus la compression est efficace, plus la pente est faible.

Aperçu global :

Les résultats issus du benchmark confirment les comportements attendus des différentes variantes :

- les modes sans overflow sont rapides mais peu compressants,
- les modes avec overflow obtiennent les meilleurs taux de compression,
- le mode Overlap reste le meilleur compromis entre vitesse et compacité,

L'analyse des temps moyens (tComp, tDecomp) et du nombre de paquets (nO, nC) permet de distinguer trois profils de comportement.

Cas simple :

- small : Tous les modes obtiennent un taux de compression maximal (nC = 8 contre nO = 266), mais le temps CPU est élevé. Les seuils de rentabilité sont compris entre 1 et 2s. La compression est donc vraiment intéressante.
- large : Toutes les variantes présentent nO = nC, donc aucun gain en taille. Cela correspond au cas défavorable où la compression ne sert à rien car les données occupent déjà 30 bits sur 32.

Cas intermédiaires :

- overlap : Les valeurs de taille moyenne (k = 10) mettent en évidence le bénéfice du chevauchement :
    - Overlap : nC = 83, Seuil = 2s
    - WithoutOverlap : nC = 88, Seuil = 2s
    - Overflow : nC entre 91 et 133, Seuils entre 3 et 4s

  Le mode Overlap reste le plus efficace sur ce profil : il compresse davantage sans coût CPU excessif.

  Les modes Overflow deviennent utiles uniquement pour des latences plus élevées, car leur logique adaptative induit un surcoût.
- without_overlap : Seuils de rentabilité autour de 1 à 2s selon les variantes.

  Les écarts entre modes sont faibles : les performances sont stables car les données sont faciles à compresser et ne nécessitent pas de traitement complexe.

Cas réaliste overflow :

- Sans Overflow (Overlap, WithoutOverlap), nC > 200, la compression est peu utile.
- Avec Overflow : nC = 133, seuil ≈ 9.5 s, on y gagne vraiment elle devient intéressant.

Ces valeurs plus élevées reflètent le coût additionnel de la double compression (base + overflow).
Cependant, elles montrent aussi que la compression devient rentable sur des réseaux à forte latence (satellite, longue distance).

Conclusion de l'analyse des résultats :

- Les modes simples (WithoutOverlap, Overlap) sont efficaces à faible latence grâce à leur rapidité d'exécution.
- Les modes Overflow montrent tout leur intérêt lorsque la latence devient le facteur limitant et que la distribution des valeurs est irrégulière.
- Les jeux large et overflow démontrent que l'algorithme reste correct et stable, même lorsque la compression n'apporte aucun gain.
- Les variantes Overlap sont globalement les plus équilibrées.

8. Perspectives d'amélioration

a. Points forts de la solution actuelle

Le système développé constitue une base solide et modulaire pour la compression d'entiers par BitPacking.
Plusieurs aspects démontrent cela :

- Architecture claire et extensible :
  L'utilisation combinée des Design Patterns a permis d'obtenir un code hautement réutilisable et adaptable.
  Chaque composant peut évoluer indépendamment (nouvelles stratégies, nouveaux benchmarks, nouveaux formats d'entrée).

- Robustesse et fiabilité :
  Les tests unitaires et de couverture assurent que les résultats sont corrects, quelle que soit la configuration utilisée.
  Les benchmarks automatisés (BenchRunner) garantissent des mesures reproductibles et cohérentes.

- Analyse expérimentale complète :
  L'intégration des graphiques et sorties consoles rendent le projet expérimentalement exploitable : l'utilisateur peut observer en une commande l'impact de la latence réseau sur la rentabilité de la compression.

- Respect des principes de Génie Logiciel :
  L'ensemble du code respecte les principes SOLID, la séparation des responsabilités et une documentation claire.
  Cette rigueur rend le projet maintenable et évolutif.

b. Extensions possibles

1. Gestion des entiers négatifs :

Le Bit Packing actuel ne traite que des entiers positifs (valeurs non signées).
Une extension naturelle serait d'intégrer un codage signé :

- en utilisant un encodage zigzag (comme dans Protocol Buffers) pour transformer les entiers signés en non signés,
- en réservant un bit de signe explicite.

Cela permettrait de rendre le système compatible avec un plus large éventail d'applications (valeurs différentielles, données compressées d'images ou signaux).

2. Automatisation du choix des paramètres

Actuellement, le Mode et les configurations sont fournis manuellement.
Une évolution consisterait à :

- analyser automatiquement la distribution des données d'entrée,
- choisir le mode et le k optimal à l'exécution

Cela transformerait le système en un compresseur intelligent, capable de s'adapter sans intervention humaine.

3. Parallélisation du traitement

Le système pourrait bénéficier d'une parallélisation :

- soit au niveau des blocs de données (par chunk de taille fixe),
- soit au niveau des appels compress et decompress via des threads.

Cette amélioration augmenterait fortement la scalabilité sur les grandes entrées (large, overflow) et permettrait de réduire les temps CPU mesurés.

9. Conclusion

Ce projet de Génie Logiciel a permis de concevoir, implémenter et évaluer un système complet de compression d'entiers basé sur la méthode du Bit Packing.
L'objectif était de réduire la taille des données tout en maintenant un accès direct à chaque élément, et d'en mesurer la rentabilité en fonction de la latence réseau.

La solution développée repose sur une architecture modulaire et extensible, intégrant plusieurs Design Patterns pour séparer clairement les responsabilités et faciliter les évolutions futures.
Grâce à cette conception, le système peut facilement accueillir de nouvelles variantes de compression ou de nouveaux modes de mesure.

Les tests unitaires ont validé la fidélité de la compression et la robustesse des implémentations, tandis que la classe BenchRunner et le script Python ont permis de mettre en place un protocole de mesure automatisé, reproductible et visualisable.
Les résultats montrent que la compression devient réellement avantageuse lorsque la latence de transmission dépasse un certain seuil, et que les modes Overlap et Overflow offrent les meilleurs compromis.

Enfin, plusieurs pistes d'amélioration ont été identifiées, ouvrant la voie à une version encore plus performante et généralisable du compresseur.
Ce travail constitue ainsi une réalisation complète et aboutie, à la fois sur le plan technique, méthodologique et pédagogique.