Plan de test

Projet : Conception d'un outil d'expérimentation de scénarios et de génération d'emploi du temps

Arbaut Jean-Baptiste

Deschamps Kylian

Duez-Faurie Valentine

Eramil Kadir

Pilloud Aubry

Tropel Célia

Université Grenoble Alpes

Plan de test

Conception d'un outil d'expérimentation de scénarios et de génération d'emploi du temps

Les informations d'identification du document :

Référence du document :	Plan de test
Version du document :	1.0
Date du document :	11/06/2025
Auteur(s) :	Arbaut Jean-Baptiste Deschamps Kylian Duez-Faurie Valentine Eramil Kadir Pilloud Aubry Tropel Célia

Les éléments de vérification du document :

Validé par :	Aurélie Landry
Validé le :	17/06/2025
Soumis le :	11/06/2025
Type de diffusion :	Document électronique (.pdf)
Confidentialité :	/

Les éléments d'authentification :

Maître d'ouvrage:	Aurélie Landry	**Chef de projet :**	/
Date / Signature :	17/06/2025 	**Date / Signature :**	/

Sommaire

1. Introduction

Ce document a pour objectif de présenter la vision globale et la structure des tests unitaires mis en place pour le projet “Conception d'un outil d'expérimentation de scénarios et de génération d'emploi du temps”. Il décrit les différents scénarios de tests, leurs objectifs, méthodes, dépendances et critères de validation. L’enjeu est de garantir la fiabilité et la robustesse de la génération automatique d’emplois du temps en respectant l’ensemble des contraintes imposées dans la création du solveur.

1.1 Objectifs et méthodes

L’application développée s’inscrit dans le cadre d’un projet de TER (Travail d’Étude et de Recherche) et vise à proposer une solution complète et accessible pour la génération automatique d’emplois du temps dans les établissements scolaires de type collège. Cette application a pour objectif de réduire significativement le temps et la complexité liés à la création manuelle des plannings hebdomadaires, tout en garantissant la prise en compte de l’ensemble des contraintes pédagogiques, matérielles et humaines propres à chaque établissement.

Le projet repose sur deux composantes principales :

- Une interface web interactive, conçue avec le framework Dash (Python), permettant à l’utilisateur de renseigner ou importer les données nécessaires : enseignants, classes, disciplines, salles, contraintes horaires, options pédagogiques… Cette interface se veut claire, et accessible à des utilisateurs sans compétences en programmation. Elle permet également de visualiser, modifier et exporter les résultats.
- Un moteur de résolution de contraintes, basé sur la bibliothèque OR-Tools développée par Google, appelé “solver”, qui modélise le problème de l’emploi du temps sous forme d’un problème d’optimisation à contraintes. Ce solver prend en compte les règles définies (volumes horaires, incompatibilités, préférences, simultanéités, etc.) et produit des plannings faisables et équilibrés, en minimisant les conflits.

Le développement a suivi une approche itérative en plusieurs étapes :

1. Recueil des besoins auprès d’exemples réels d’emplois du temps scolaires et définition des types de contraintes à prendre en charge.
2. Modélisation du problème et du plan de développement.
3. Implémentation du solver avec OR-Tools, incluant la gestion des sous-groupes, des salles, des volumes horaires par matière, et des conflits potentiels. Conception en parallèle de l’interface Dash en modules fonctionnels : saisie des données, gestion des contraintes, visualisation des résultats et export. Avec des fonctionnalités telles que le système de sauvegarde automatique.

4. Validation continue par des jeux de données réalistes et ajustements selon la faisabilité et la lisibilité des emplois du temps générés.
5. Création des tests et de la documentation associée.
6. Documentation : les commentaires du code source ainsi que la documentation interne ont été réalisés tout le long du projet. Le rapport final et les guides ont été rédigés en fin de projet.

L'ensemble du système est conçu pour fonctionner localement, sans dépendre d'un serveur distant, et pour pouvoir être adapté selon les spécificités des établissements. Le projet met l'accent sur la modularité, la fiabilité des résultats, et l'accessibilité pour des non-informaticiens. L'un des objectifs était de fournir un export compatible avec Monoposte d'Index Education, ce qui n'a pas pu être effectué.

1.2 Documents de référence

La lecture du guide d'installation est indispensable afin de s'assurer que tous les modules et que Python soit correctement installés sur la machine de l'utilisateur. Sans cela, le fichier de test ne fonctionnera pas. Ce document se base sur les fichiers des données du projet (.JSON).

2. Guide de lecture

Ce document a été conçu pour répondre aux besoins de différents profils de lecteurs impliqués dans le développement, le test ou l'utilisation de l'application de génération d'emploi du temps. Chaque section du plan de test correspond à une étape clé du processus de validation.

2.1 Utilisateur final et maître d'ouvrage

Objectif : Comprendre les cas testés pour utiliser l'outil en toute confiance.

À lire en priorité :

- Section 6 : Vérification de la documentation : pour s'assurer que les procédures d'installation et d'utilisation sont fiables.
- Section 7 : Glossaire : pour clarifier les termes techniques du document.

L'utilisateur final peut ainsi se repérer dans les fonctionnalités testées, sans entrer dans les détails techniques du code.

2.2 Développeur (maître d'œuvre)

Objectif : Maintenir, adapter ou enrichir les tests du projet.

À lire en priorité :

- Section 5 : Tests unitaires : pour comprendre les validations spécifiques liées aux contraintes du solveur OR-Tools.
- Section 4 : Tests d'intégration : pour comprendre l'enchaînement des modules côté interface.
- Section 1.1 et 1.2 : Objectifs et documents de référence : pour disposer du contexte technique du projet.

Ce profil pourra utiliser ce document pour ajouter de nouveaux tests, modifier ceux existants ou valider des modifications du code source.

3. Concepts de base

Cette section présente les notions essentielles à connaître pour comprendre le fonctionnement de l'application et suivre efficacement les étapes décrites dans ce document. Elle permet d'aborder la suite du plan de test avec une vision claire des composants et mécanismes techniques du système.

Certaines actions de test ou de configuration (comme le lancement de l'application ou la vérification des fichiers) nécessitent l'utilisation d'un terminal (sous Linux / macOS) ou d'une invite de commande (sous Windows). Il s'agit d'une interface textuelle permettant d'exécuter des instructions telles que "python app.py".

L'application repose sur Dash, un framework Python utilisé pour concevoir des interfaces web interactives. L'interface se lance dans un navigateur et permet à l'utilisateur de renseigner des données, de configurer des contraintes pédagogiques, puis de visualiser ou exporter les résultats. Aucun logiciel supplémentaire n'est requis, et aucune compétence en programmation n'est nécessaire pour interagir avec l'application.

Toutes les données nécessaires au fonctionnement (enseignants, classes, matières, contraintes horaires…) sont stockées dans des fichiers JSON (.json). Ce format, lisible à la fois par des machines et des humains, organise les informations sous forme de paires clé / valeur. L'interface se charge automatiquement de la lecture, de la mise à jour et de la sauvegarde de ces fichiers : l'utilisateur n'a pas à les modifier manuellement.

Le solver de génération des emplois du temps s'appuie sur OR-Tools, une bibliothèque d'optimisation développée par Google. Cette bibliothèque permet de modéliser un problème sous forme de contraintes logiques et numériques (ex : 'deux groupes ne peuvent pas être dans la même salle en même temps'), et de rechercher une solution optimale qui respecte au mieux l'ensemble des contraintes définies par l'utilisateur.

Enfin, l'application fonctionne en local, c'est-à-dire qu'elle s'exécute entièrement sur l'ordinateur de l'utilisateur. Aucune connexion à internet, serveur distant ou base de données externe n'est requise. L'intégralité des opérations de la saisie des données au calcul de l'emploi du temps, jusqu'à l'export des résultats est réalisée en local, ce qui garantit la confidentialité des données.

4. Tests d'intégration

TI001 - Chargement de données JSON dans l'interface Dash

Description

- Ce test vérifie que les données du fichier data_interface.json sont correctement lues et injectées dans les composants de l'interface Dash au chargement de chaque page. Il s'agit d'une interface interne entre le backend de chargement JSON et le layout Dash.
- **Environnement** : Application Dash locale, fichier data_interface.json prérempli.
- **Principe** : Vérifier que les champs sont automatiquement remplis avec les données du fichier.

Contraintes

- Le fichier JSON doit exister et être bien formé.
- L'interface doit être démarrée dans un environnement Dash fonctionnel.
- Intervention humaine pour vérifier l'affichage visuel.

Dépendances

- Test U001 : Vérification de la structure JSON (data_interface.json).
- Test U002 : Initialisation des composants Dash (layout, callback de chargement).

Procédure de test

- **Entrée** : fichier data_interface.json avec des valeurs spécifiques ("LV1": ["Anglais"])
- **Résultat attendu** : la liste déroulante "LV1" est préremplie avec "Anglais"

Critère de validation

L'utilisateur voit les champs remplis correctement sans action manuelle.

TI002 - Sauvegarde automatique des données lors de la fermeture d'un volet

Description

Ce test vérifie que les données saisies dans un volet sont automatiquement sauvegardées dans data_interface.json lors de la fermeture (collapse) du volet.

Contraintes

- Il faut déclencher un changement d'état (volet ouvert ou fermé).

Dépendances

- TI001 : le chargement de données JSON dans l'interface Dash doit être correct.

Procédure de test

- **Entrée** : L'utilisateur saisit "Espagnol" en LV2 puis ferme le volet.
- **Résultat attendu** : data_interface.json contient "LV2": ["Espagnol"]

Critère de validation

À la réouverture de l'interface, "Espagnol" est bien rechargé.

TI003 - Lancement du solveur OR-Tools depuis l'interface

Description

Vérifie que le solveur est lancé, que les données sont transmises, et que le résultat est affiché.

Contraintes

- OR-Tools doit être installé.
- Données cohérentes requises dans data_interface.json.
- L'environnement de test peut être lourd s' il y a un gros volume de données dans data_interface.json.

Dépendances

- TI001 et TI002 doivent avoir été validés.

Procédure de test

- **Entrée** : Clic sur le bouton "Lancer le calcul".
- **Résultat attendu** : Barre de chargement puis affichage d'un emploi du temps sur la page de résultats.

Critère de validation

Affichage correct et sans plantage, contenu du résultat non vide.

TI004 - Génération dynamique des dropdowns selon les données JSON

Description

Vérifie que les listes déroulantes de classes, matières, salles, etc., s'adaptent dynamiquement au contenu de data_interface.json. Cela teste l'interface interne entre le module de lecture JSON et le front-end Dash.

Contraintes

- Le fichier JSON doit contenir des entrées valides.
- L'utilisateur doit avoir interagi avec la page pour déclencher le callback.

Dépendances

- TI001 : chargement de données JSON dans l'interface Dash doit être validé.

Procédure de test

- **Entrée** : Ajout dans data_interface.json de nouvelles classes : "classes": ["6e1", "6e2"]
- **Résultat attendu** : Les dropdowns affichent automatiquement "6e1", "6e2" sur la page de contraintes.

Critère de validation

Aucun redémarrage nécessaire, affichage du résultat attendu dans la page de contraintes.

TI005 - Export HTML et PDF du résultat

Description

Ce test vérifie l'intégration entre l'interface de résultats Dash et les modules d'export HTML/PDF.

Contraintes

- L'écriture de fichiers doit être fonctionnelle.

Dépendances

- TI003 : lancement du solveur OR-Tools depuis l'interface doit être validé.

Procédure de test

- **Entrée** : Clic sur le bouton "Exporter en PDF".
- **Résultat attendu** : Fichier emploi_du_temps.pdf généré dans le dossier.

Critère de validation

Le contenu du fichier doit être lisible.

5. Tests unitaires

TU001 - test_indisponibilites_respectees

Description

Vérifier que les indisponibilités des professeurs et des salles, telles que définies dans le fichier config_test.json, sont bien respectées dans la génération d'emploi du temps.

- **But du test** : s'assurer qu'aucune classe n'est programmée pour une matière lorsque le professeur ou la salle associés sont indisponibles.
- **Environnement de test** :
    - Python 3.x
    - Bibliothèques : pytest, ortools
    - Fichiers :
        - test.py (définition du test),
        - config_test.json (configuration utilisée).
- **Principe** :
    - Pour chaque itération (50 exécutions), on appelle creer_modele(config) pour construire le modèle CP-SAT à partir de config_test.json.
    - On résout le modèle (status FEASIBLE ou OPTIMAL) puis on vérifie, pour chaque indisponibilité spécifiée :
        - **Professeurs** : pour chaque professeur "prof" et chaque créneau (jour, heure) mentionné dans config["indisponibilites_profs"][prof], on vérifie que l'emploi du temps assigné n'affecte jamais à une classe une matière nécessitant ce professeur à ce créneau.
        - **Salles** : de même, pour chaque salle "salle" et chaque créneau interdit, on identifie les matières qui doivent avoir lieu dans cette salle (par affectation_matiere_salle) et on contrôle qu'aucune classe n'a cette matière à l'heure interdite.

Contraintes

- Lecture correcte de config_test.json (codage UTF-8).
- Lancer le solver sans modification manuelle du modèle avant vérification.
- Nécessité d'un temps de calcul suffisant pour 50 réitérations (paramétrées dans @pytest.mark.parametrize("n_iter", [50])).

Dépendances

- Aucune exécution préalable requise. Ce test repose uniquement sur la fonction creer_modele et sur la méthode de résolution resoudre_modele.

Procédure de test

1. **Chargement de la configuration :**
   - Lecture de config_test.json.
2. **Construction du modèle :**
   - Appel creer_modele(config) puis obtention de (model, emploi_du_temps, JOURS, HEURES, MATIERES, PROFESSEURS, CLASSES).
3. **Résolution :**
   - Appel resoudre_modele(model) → solver.
4. **Vérification indisponibilités profs :**
   - Pour chaque prof dans config["indisponibilites_profs"], pour chaque jour "jour_str" et liste d'heures heures :
     - Déterminer l'indice jour_index = JOURS.index(jour_str).
     - Pour chaque classe "classe" de CLASSES, déterminer si prof intervient dans une matière matiere.
     - Pour chaque h dans heures, récupérer val = solver.Value(emploi_du_temps[(classe, jour_index, h)]) et s'assurer que val != MATIERES.index(matiere)+1.
5. **Vérification indisponibilités salles :**
   - Pour chaque salle dans config["indisponibilites_salles"], identifier la liste matieres_concernees = [m for m, s in affectation_matiere_salle.items() if s == salle].
   - Pour chaque jour, chaque heure interdite, chaque classe, chaque matière dont le professeur ou la structure est associée à cette salle, vérifier que la valeur du créneau n'est pas l'indice de la matière.

Critères de validation

Aucune assertion ne doit échouer ; en cas d'indisponibilité violée, pytest lève un message détaillé.

TU002 - test_volume_horaire_respecte

Description

S'assurer que, pour chaque niveau et matière, le nombre d'heures alloué dans l'emploi du temps correspond strictement au volume horaire spécifié dans config_test.json.

- **But du test** : vérifier la contrainte de volume horaire dans le modèle CP-SAT.
- **Environnement de test** : identique à TU001 (Python 3.x, pytest, ortools, fichiers test.py et config_test.json).
- **Principe** : pour 50 réitérations, on génère un nouvel emploi du temps (via creer_modele), on le résout, puis on compte, pour chaque classe et chaque matière de config["volume_horaire"][niveau] :

- On parcourt tous les jours (JOURS) et toutes les heures (HEURES).
- On incrémente un compteur chaque fois que la valeur de emploi_du_temps[(classe,j,h)] égale l'indice de la matière.
- On compare ce compteur au nombre d'heures attendues (heures_attendues).

Contraintes

- Nécessite que la création du modèle inclut déjà la contrainte de volume horaire (implémentée dans creer_modele).
- Recourir à la même configuration (fichier JSON) pour chaque itération.
- Le solver doit être en mesure de trouver une solution satisfaisant la contrainte de volume horaire.

Dépendances

- Aucune exécution préalable autre que le fonctionnement correct de creer_modele et de la contrainte de volume horaire.

Procédure de test

1. **Chargement de la configuration :**
   - Lecture de config_test.json.
2. **Boucle d'itérations (50 fois) :**

   a. Appeler creer_modele(config) pour construire le modèle.

   b. Résoudre avec la fonction resoudre_modele()

   c. Pour chaque classe dans CLASSES :

   i. Pour chaque paire (matiere, heures_attendues) dans config["volume_horaire"][classe[:2]] :

   • mat_index = MATIERES.index(matiere)+1

   • Initialiser compte = 0.

   • Pour chaque j (indice de jour) et chaque h (indice d'heure) :

   - Récupérer val = solver.Value(emploi_du_temps[(classe,j,h)]).

   - Si val == mat_index, faire compte on incrémente de 1.

   • **Assert** : compte == heures_attendues; sinon levée d'erreurs indiquant la différence.

Critère de validation

Pour chaque matière de chaque classe, le nombre d'occurrences doit correspondre exactement à la valeur de config["volume_horaire"] (aucune différence autorisée).

TU003 - test_matExcluSuite

Description

Vérifier que la contrainte d'exclusion immédiate forte (pas de suite "Matière A" → "Matière B") fonctionne correctement :

- **But du test** : s'assurer qu'aucun créneau où "Maths" est programmé n'est jamais immédiatement suivi par un créneau "SVT" dans la journée, pour chaque classe et pour chaque jour. Cette contrainte est codée en dur dans le fichier contrairement au solveur par souci de facilité d'exécution. On précise que la contrainte est « forte » pour forcer tous les cours de maths à être suivi par des cours de SVT.
- **Environnement de test** :
    - Même que précédemment (Python 3.x, pytest, ortools, fichiers test.py et config_test.json).
- **Principe** :
    1. On reconstruit un nouveau modèle via creer_modele(config).
    2. On applique explicitement la contrainte matExcluSuite()
    3. On résout le modèle.
    4. On parcourt chaque classe, chaque jour, chaque heure "h" (sauf dernière heure) :
        - Si emploi_du_temps[(classe,j,h)] == indice("Maths"), alors on contrôle que emploi_du_temps[(classe,j,h+1)] != indice("SVT").

Contraintes

- La méthode matExcluSuite doit être implémentée correctement dans test.py.
- Nécessité d'un solver capable de trouver une solution même avec cette contrainte d'exclusion.
- L'appel à matExcluSuite doit précéder la résolution (et ne doit pas être involontairement écrasé).

Dépendances

- Le test suppose que la fonction creer_modele définit correctement les variables d'emploi du temps (emploi_du_temps) et que la liste MATIERES contient bien "Maths" et "SVT".
- Aucune dépendance à un autre test particulier.

Procédure de test

1. **Chargement de la configuration**.
2. **Création du modèle de base** : model, emploi_du_temps, JOURS, HEURES, MATIERES, PROFESSEURS, CLASSES = creer_modele(config).

3. **Résolution** : solver = resoudre_modele(model).
4. **Vérification :**
   Pour chaque classe, j et h ∈ [0, len(HEURES)-2] :

   val1 = solver.Value(emploi_du_temps[(classe,j,h)]).
   val2 = solver.Value(emploi_du_temps[(classe,j,h+1)]).
   Si val1 == MATIERES.index("Maths")+1, alors **assert** val2 == MATIERES.index("Français")+1.

Critère de validation

Aucune occurrence de “Maths” suivie d’autre chose que “SVT” ne doit exister.

TU004 - test_matIncluSuite

Description

Vérifier que la contrainte d’inclusion immédiate forte (chaque occurrence de “Matière A” doit être suivie de “Matière B”) fonctionne correctement. Cette contrainte est également codée en dur dans le fichier contrairement au solveur par souci de facilité d’exécution.
But du test : s’assurer que, pour chaque créneau où “Maths” apparaît, le créneau suivant de la même journée est “Français”.
Environnement de test : identique aux tests précédents (Python 3.x, pytest, ortools, fichiers test.py et config_test.json).
Principe :

1. Création du modèle via creer_modele(config).
2. Application de la contrainte d’inclusion forte avec matIncluSuite().
3. Résolution du modèle.
4. Pour chaque classe, pour chaque jour et pour chaque heure h (sauf la dernière heure) :
   - Si “Maths” est programmé au créneau (j, h), alors vérifier que “Français” figure au créneau (j, h+1).

Contraintes

- La fonction matIncluSuite doit créer correctement les BoolVar et les implications nécessaires.
- Les matières “Maths” et “Français” doivent exister dans la liste MATIERES extraite du JSON.
- Le paramètre pNBFois=1 impose qu’au moins une occurrence de “Maths” soit suivie de “Français” ; en mode “forte”, toutes les occurrences doivent satisfaire cette règle.
- Cette contrainte est donc obligatoire.

Dépendances

- Même dépendances que TU003 :
  Le bon fonctionnement de la fonction creer_modele pour initialiser le modèle et la variable emploi_du_temps.
- La présence de “Maths” et “Français” dans la configuration (config_test.json).

Procédure de test

1. **Chargement de config_test.json**.
2. **Création du modèle** d’emploi du temps en appelant creer_modele(config), ce qui retourne notamment le modèle CP-SAT, la structure emploi_du_temps, la liste JOURS, HEURES, MATIERES, PROFESSEURS et CLASSES.
3. **Ajout de la contrainte d’inclusion forte** en appelant matIncluSuite avec les paramètres suivants :
   - model
   - emploi_du_temps
   - JOURS
   - HEURES
   - MATIERES
   - CLASSES
   - pMatiere1 = “Maths”
   - pMatiere2 = “Français”
   - pContrainte = “forte”
   - pNBFois = 1
4. **Résolution du modèle** : appeler resoudre_modele(model) pour obtenir un objet solver.
5. **Vérification post-solution** : pour chaque classe, pour chaque index de jour j et pour chaque index d’heure h allant de 0 à len(HEURES) - 2 :
   - Récupérer la valeur val1 = solver.Value(emploi_du_temps[(classe, j, h)]).
   - Récupérer la valeur val2 = solver.Value(emploi_du_temps[(classe, j, h + 1)]).
   - Si val1 correspond à l’indice de “Maths” (c’est-à-dire MATIERES.index(“Maths”) + 1), alors vérifier que val2 correspond à l’indice de “Français” (MATIERES.index(“Français”) + 1). Sinon, lever une erreur.

Critère de validation

Toutes les occurrences de “Maths” doivent être suivie de “Français” sur la même journée. Toute violation entraîne un échec du test.

TU005 – test_memNivMemCours

Description

Vérifier que la fonction memNivMemCours impose correctement la contrainte “Même Niveau, Même Cours” pour le niveau 6e et la matière Maths, sans rendre le modèle insoluble. On initialise un modèle OR-Tools avec la configuration de test, on applique la contrainte pour les classes de 6e sur le cours de Maths, puis on résout. Enfin, on s’assure que les créneaux Maths des deux premières classes de 6e sont toujours synchronisés et que leur nombre correspond exactement au volume horaire attendu.

Contraintes

- Le fichier config_test.json doit contenir la clé “volume_horaire” avec au moins une entrée pour le niveau 6e et la matière Maths.
- Doit exister au moins deux classes dont l’identifiant commence par “6e” dans la liste CLASSES.
- Nécessite Python 3.x, pytest et OR-Tools installés, sans intervention manuelle pendant l’exécution.

Dépendances

- La fonction creer_modele(config) validée (génération correcte de model, emploi_du_temps, JOURS, HEURES, MATIERES, PROFESSEURS, CLASSES).
- Le chargeur de configuration charger_config opérationnel.
- La fonction resoudre_modele(model) fonctionnant et retournant un statut valide.

Procédure de test

1. **Charger le fichier de configuration** config_test.json à l’aide de la fonction de chargement de config.
2. **Pour 50 itérations,** répéter les opérations suivantes :
    - Créer le modèle de base (appel à creer_modele) et obtenir les structures emploi_du_temps, listes JOURS, HEURES, MATIERES, PROFESSEURS, CLASSES ainsi que VOLUME_HORAIRE extrait de la config.
    - Appliquer la contrainte “Même Niveau, Même Cours” en ciblant le niveau “6e” et la matière “Maths” (appel à memNivMemCours).
    - Résoudre le modèle (appel à resoudre_modele) et vérifier que le statut retourné est FEASIBLE ou OPTIMAL.

- Identifier dans la liste des classes celles dont le nom commence par “6e” et s’assurer qu’il y en a au moins deux. Sélectionner les deux premières classes de 6e pour les vérifications suivantes.
- Déterminer l’indice de la matière “Maths” dans la liste des matières et récupérer le volume horaire attendu pour les 6e depuis VOLUME_HORAIRE.
- Pour chaque jour et chaque créneau horaire, vérifier que si l’une des deux classes de 6e a “Maths” à cet horaire, l’autre aussi.
- Compter le nombre total de créneaux où la première classe de 6e reçoit “Maths” et s’assurer que ce nombre correspond exactement au volume horaire attendu.

Critère de validation

- Le modèle se résout systématiquement avec un statut FEASIBLE ou OPTIMAL.
- Pour chacun des créneaux étudiés, les deux classes de 6e partagent exactement les mêmes affectations pour la matière “Maths”.
- Le nombre total de séances de “Maths” pour chaque classe de 6e est égal au volume horaire défini dans la configuration.

TU006 - test_contrainte_cantine

Description

Vérifier que la contrainte cantine interdit bien la présence de cours sur le créneau attribué au déjeuner pour chaque classe, en fonction de la capacité de la cantine et de la répartition prioritaire ou cyclique.

- **But du test** :
  1. Si total_eleves ≤ cap_cantine : tout le monde doit déjeuner à “12h-13h” ; aucun cours autorisé à ce créneau pour aucune classe sur tous jours.
  2. Sinon (capacité insuffisante) : chaque classe doit se voir attribuer un créneau dans creneaux_dejeuner en respectant assignation_niveaux si priorite_active=True, ou une répartition circulaire si False. Ensuite, aucun cours ne doit être programmé dans le créneau cantine de chaque classe.
- **Environnement de test** :
  - Python 3.x, pytest, ortools.
  - Fichiers : test.py (implémentation de appliquer_contrainte_cantine et du test), config_test.json.

Contraintes

- config_test.json doit contenir au moins :

  - "capacites_classes",
  - "cantine" avec clés "capacite", "proportion_demi_pensionnaire", "creneaux_dejeuner", "assignation_niveaux", "priorite_active".
- Le créneau "12h-13h" doit figurer parmi les heures (HEURES).
- Si total_eleves > cap_cantine et priorite_active=True, tous les niveaux doivent avoir un créneau défini dans assignation_niveaux, sinon appliquer_contrainte_cantine lève ValueError.

Dépendances

- Nécessite que creer_modele(config) génère un modèle correct sans contrainte cantine.
- Utilisation de HEURES.index("12h-13h").
- Existence de décisions dans SALLES_GENERALES et capacites_classes.

Procédure de test

1. **Chargement** du fichier config_test.json.
2. **Création du modèle de base** (sans contrainte cantine) : model, emploi_du_temps, JOURS, HEURES, MATIERES, PROFESSEURS, CLASSES = creer_modele(config).
3. **Lecture des paramètres cantine** à partir du JSON :

```
cap_cantine = config["cantine"]["capacite"]
ratio_demi = config["cantine"]["proportion_demi_pensionnaire"]
creneaux_dej = config["cantine"]["creneaux_dejeuner"]
assignation_niveaux = config["cantine"]["assignation_niveaux"]
priorite_active = config["cantine"]["priorite_active"
```

4. **Résolution :**
   - Si total_eleves = sum(capacites_classes[classe] * ratio_demi) ≤ cap_cantine : toutes les classes ont créneau cantine à “12h-13h”.
   - Sinon, si priorite_active=False : répartition circulaire (cycle) sur creneaux_dej.
   - Sinon, on affecte chaque classe à son créneau prioritaire défini dans assignation_niveaux; si un niveau n’est pas mentionné, levée de ValueError.
   - Ajout, pour chaque classe et chaque jour, de la contrainte emploi_du_temps[(classe,j,h_dej)] == 0.
5. **Vérifications** :
   - Si total_eleves ≤ cap_cantine : pour chaque classe et chaque j, val = solver.Value(emploi_du_temps[(classe,j, HEURES.index("12h-13h"))]) doit être égal à 0. Sinon, levée d’erreur.
   - Sinon (en cas de capacité insuffisante) :

- Construire classe_creneau_dej en reprenant la même logique que dans appliquer_contrainte_cantine (vérifier que chaque niveau dispose d'un créneau).
- Pour chaque classe et j, val = solver.Value(emploi_du_temps[(classe,j, h_dej)]) avec h_dej = classe_creneau_dej[classe] doit être égal à 0.

Critère de validation

Aucun cours ne doit être programmé sur le créneau cantine de chaque classe.

TU007 - test_poids_cartable

Description

Vérifier que la contrainte "poids du cartable" (limitation du poids total des matières par jour et par classe) est construite correctement et n'empêche pas la résolution du modèle.

- **But du test** : s'assurer que, lorsqu'on ajoute la contrainte de poids cartable, le modèle reste solvable (statut FEASIBLE ou OPTIMAL).
- **Environnement de test** :
  - Python 3.x, pytest, ortools.
  - Fichiers : test.py (implémentation de appliquer_contrainte_poids_cartable), config_test.json.
- **Principe** :
  1. On crée le modèle de base via creer_modele(config).
  2. On appelle appliquer_contrainte_poids_cartable(model, emploi_du_temps, JOURS, HEURES, MATIERES, CLASSES, config).
  3. On résout le modèle.
  4. On vérifie simplement que le solver renvoie un statut FEASIBLE ou OPTIMAL ; le test n'effectue pas de validation plus fine sur la valeur des pénalités.

Contraintes

- Le JSON doit contenir :
  - "poids_matieres" : dictionnaire poids par matière (entiers ou flottants).
  - Soit "poids_cartable_max_somme_par_niveau" (poids max par niveau) ou, à défaut, "poids_cartable_max_somme".
  - Éventuellement "jours_sans_apres_midi", "sous_groupes_suffixes", même si leur absence n'empêche pas la contrainte principale.
- La fonction crée des variables BoolVar et IntVar pour chaque (classe, jour, heure, matière), puis un IntVar pour la somme de ces poids.

- Le solver doit être capable d'optimiser (Minimize) la somme des pénalités ; on vérifie seulement la solvabilité.

Dépendances

- Dépend de la présence correcte des clefs JSON citées ci-dessus.
- Fonctionnement correct du code appliquer_contrainte_poids_cartable.

Procédure de test

1. **Chargement** du fichier config_test.json.
2. **Création du modèle de base** : model, emploi_du_temps, JOURS, HEURES, MATIERES, PROFESSEURS, CLASSES = creer_modele(config).
3. **Application de la contrainte de poids** :
   model = appliquer_contrainte_poids_cartable(model, emploi_du_temps, JOURS, HEURES, MATIERES, CLASSES, config)
   Le code parcourt chaque classe, chaque jour, chaque heure (en sautant l'après-midi si le jour est dans jours_sans_apres_midi), et crée :
   - Des BoolVar "poids_<classe><*j*><h>_<matiere>" pour chaque matière possible à cet horaire.
   - Un IntVar "poids_total_<classe><*j*><h>" égal à la somme des poids (×10) des matières sélectionnées.
   - Un IntVar "somme_poids_jour_<classe>_<j>" égal à la somme de tous les poids_total de la journée.
   - Un IntVar "surcharge_poids_jour_<classe>_<j>" = max(somme_poids_jour - seuil_tolerant, 0).

   Enfin, ajoute model.Minimize(sum(penalites_surcharge_poids)).
4. **Résolution** : solver = resoudre_modele(model).
5. **Vérification** :
   - **Assert** que solver.StatusName() est dans ["FEASIBLE", "OPTIMAL"].

Critère de validation

Le modèle doit être soluble avec la contrainte de poids cartable activée, en prenant en compte un seuil de tolérance.

TU008 - test_max_heures_par_etendue

Description

S'assurer que la contrainte "maximum d'heures par étendue" (journée ou demi-journée) pour une matière et un niveau est correctement appliquée.

- **But du test** : vérifier, pour chaque règle spécifiée dans config["max_heures_par_etendue"], que le nombre d'heures alloué à la matière concernée, soit par journée entière, soit par demi-journée (matin ou après-midi), ne dépasse pas la valeur max_heures. Dans notre cas, il faut qu'il y ait au minimum une heure de français par demi-journée.
- **Environnement de test** :
  - Python 3.x, pytest, ortools.
  - Fichiers : test.py (implémentation de appliquer_contrainte_max_heures_etendue), config_test.json.
- **Principe** :
  1. Chargement de la configuration et extraction de MAX_HEURES_PAR_ETENDUE = config.get("max_heures_par_etendue", []).
  2. Pour chaque itération (50 fois), création d'un modèle de base via creer_modele.
  3. Si la liste MAX_HEURES_PAR_ETENDUE n'est pas vide, appeler appliquer_contrainte_max_heures_etendue(model, emploi_du_temps, ..., MAX_HEURES_PAR_ETENDUE).
  4. Résolution du modèle.
  5. Pour chaque règle :
     - Pour chaque classe correspondant au niveau (classe.startswith(niveau)), pour chaque jour :
       - Si etendue == "journee" : compter le nombre d'heures où la matière apparaît dans tous créneaux du jour ; vérifier que compte ≤ max_heures.
       - Si etendue == "demi-journee" : pour les indices h du matin (0 à 3) et de l'après-midi (5 à len(HEURES)-1), compter séparément et vérifier que, dans chaque bloc (étiquette "matin" ou "apres"), le compte ≤ max_heures.

Contraintes

- Dans config_test.json, chaque élément de "max_heures_par_etendue" doit contenir les clés :
  - "niveau" (par ex., "6e").
  - "matiere" (par ex., "Français").
  - "max_heures" (entier).

  - "etendue" (valeur “journee” ou “demi-journee”).
- Les matières référencées doivent exister dans MATIERES, et les niveaux doivent correspondre à une (ou plusieurs) classes.
- L’indice de l’heure du repas (“12h-13h”) n’a pas d’impact ici, car ce test ne traite pas de la cantine.

Dépendances

- Dépend du bon fonctionnement de creer_modele (variables emploi_du_temps).
- Nécessite la présence de MAX_HEURES_PAR_ETENDUE dans le JSON.

Procédure de test

1. **Chargement** de config_test.json.
2. **Extraction** de MAX_HEURES_PAR_ETENDUE = config.get("max_heures_par_etendue", []).
3. **Boucle d’itérations** (50 fois) :
    1. Créer le modèle : model, emploi_du_temps, JOURS, HEURES, MATIERES, PROFESSEURS, CLASSES = creer_modele(config).
    2. S’il existe au moins une règle : appliquer_contrainte_max_heures_etendue(model, emploi_du_temps, JOURS, HEURES, MATIERES, CLASSES, MAX_HEURES_PAR_ETENDUE)
    3. Résoudre : solver = resoudre_modele(model).
    4. **Vérification post-solution** :
       Pour chaque regle dans MAX_HEURES_PAR_ETENDUE :

```
niveau = regle["niveau"]
matiere = regle["matiere"]
mat_index = MATIERES.index(matiere)+1
max_heures = regle["max_heures"]
etendue = regle["etendue"]
```

- Pour chaque classe dans CLASSES tel que classe.startswith(niveau), pour chaque j, jour dans JOURS :
  - **Si** etendue == "journee" :

```
compte = sum(1 for h in range(len(HEURES)) if solver.Value(emploi_du_temps[(classe,j,h)]) ==mat_index).
Assert compte ≤ max_heures.
```

  - **Si** etendue == "demi-journee" :

Définir matin_heures = [0,1,2,3], apres_heures = [5,6,…,len(HEURES)-1]. compte_matin = sum(1 for h in matin_heures if solver.Value(emploi_du_temps[(classe,j,h)]) == mat_index). compte_apres = sum(1 for h in apres_heures if solver.Value(emploi_du_temps[(classe,j,h)]) == mat_index). **Assert** compte_matin ≤ max_heures et compte_apres ≤ max_heures.

Critères de validation

Aucune règle de MAX_HEURES_PAR_ETENDUE ne doit être violée (aucun bloc d'étendue ne contient plus d'heures que max_heures).

TU009 - test_jours_sans_apres_midi

Description

Vérifier que la liste jours_sans_apres_midi dans config_test.json est valide (liste de jours existants) et que, pour chaque jour mentionné, aucun cours n'est programmé l'après-midi (heures index ≥ 5).

- **But du test** :
    1. Contrôler la validité relative du JSON (jours_sans_apres_midi doit être une liste et chaque jour doit figurer dans config["jours"]).
    2. Vérifier, pour chaque jour sans après-midi, qu'aucun cours n'apparaît après le 5e index d'heure (heures 5, 6, …, fin).
- **Environnement de test** : identique aux précédents.
- **Principe** :
    - On commence par valider la structure de jours_sans_apres_midi.
    - Ensuite, pour 50 itérations, on construit le modèle via creer_modele, puis on ajoute, pour chaque classe et chaque j tel que JOURS[j] appartient à jours_sans_apres_midi, la contrainte emploi_du_temps[(classe,j,h)] == 0 pour tout h ≥ 5. On résout ensuite et vérifie qu'effectivement pas de cours n'apparaît pour ces créneaux.

Contraintes

- config["jours_sans_apres_midi"] doit être une liste JSON.
- Chaque élément de cette liste doit apparaître dans config["jours"], sinon levée d'AssertionError.

- Les indices horaires choisis (h ≥ 5) correspondent à l'après-midi.

Dépendances

- Dépendance à creer_modele.
- Validation du JSON en tête de test avant toute création de modèle.

Procédure de test

1. **Validation du JSON** :

   - JOURS_SANS_APRES_MIDI = config.get("jours_sans_apres_midi", [])
   - **Assert** isinstance(JOURS_SANS_APRES_MIDI, list).
   - Pour chaque jour dans JOURS_SANS_APRES_MIDI : **Assert** jour in config["jours"].

2. **Boucle d'itérations** (50 fois) :
   - Créer le modèle de base : model, emploi_du_temps, JOURS, HEURES, MATIERES, PROFESSEURS, CLASSES = creer_modele(config).
   - **Ajout de la contrainte** pour interdire les cours en après-midi :

```
for classe in CLASSES:
        for j, jour in enumerate(JOURS):
                if jour in JOURS_SANS_APRES_MIDI:
                        for h in range(5, len(HEURES)):
                                model.Add(emploi_du_temps[(classe, j, h)] == 0)
```

3. **Résolution** : solver = resoudre_modele(model).
4. **Vérification** :
   Pour chaque classe et chaque j, jour dans JOURS tel que jour in JOURS_SANS_APRES_MIDI, pour chaque h ≥ 5 :

```
val = solver.Value(emploi_du_temps[(classe,j,h)])
Assert val == 0.
```

Critère de validation

Aucun cours n'est programmé l'après-midi pour les jours spécifiés dans jours_sans_apres_midi.

TU010 - test_matHorairDonneV2

Description

Vérifier la contrainte “matières scientifiques sur un créneau donné” implémentée par matHorairDonneV2. Le test se concentre sur le cas où, pour le niveau “6e”, on veut allouer exactement 2 heures de matières du sous-groupe “Scientifique” soit les matières (“Maths”, “SVT”, “Physique”, “Techno”) le mercredi entre 13h et 17h.

- **But du test** : s’assurer que la somme des occurrences de n’importe quelle matière du sous-groupe “Scientifique” (présent dans la liste config["sousGroupes_matieres"]["Scientifique"]) pendant le créneau “Mercredi” entre 13h et 17h est égale à 2.
- **Environnement de test** : Pareil que les précédents
- **Principe** :
    1. On initialise le modèle via creer_modele.
    2. On récupère MATIERES_SCIENTIFIQUE = config["sousGroupes_matieres"]["Scientifique"].
    3. On appelle matHorairDonneV2("6e", MATIERES_SCIENTIFIQUE, "Mercredi", "13h", "17h", 2).
    4. On résout le modèle.
    5. On parcourt, pour chaque classe de niveau “6e”, toutes les heures h comprises entre l’indice de “13h” et l’indice de “17h” (inclus), et on compte le nombre d’occurrences où la valeur de emploi_du_temps[(classe, jourdex_mercredi, h)] est dans la liste des indices des matières scientifiques.
    6. On compare cette somme à 2.

Contraintes

- “Mercredi” doit être présent dans config["jours"].
- “13h-14h”, “14h-15h”, “15h-16h”, “16h-17h” doivent figurer dans config["heures"].
- Le JSON doit définir config["sousGroupes_matieres"]["Scientifique"].
- Les matières listées doivent exister dans MATIERES.
- pNBFois=2 indique qu’on veut exactement 2 créneaux scientifiques sur ce bloc horaire.

Dépendances

- creer_modele doit fournir correctement emploi_du_temps, JOURS et HEURES.
- config["volume_horaire"] doit être cohérent avec les matières scientifiques (sinon, le test peut bouleverser le volume horaire normal, mais on se concentre sur l’existence d’au moins 2 créneaux disponibles).

Procédure de test

1. **Chargement** de config_test.json.
2. **Boucle d’itérations** (50 fois) :

- model, emploi_du_temps, JOURS, HEURES, MATIERES, PROFESSEURS, CLASSES_BASE = creer_modele(config).

```
MATIERES_SCIENTIFIQUE=
config["sousGroupes_matieres"].get("Scientifique",
["Maths","Physique","Chimie"]).
```

- **Appel de la contrainte** : matHorairDonneV2("6e", MATIERES_SCIENTIFIQUE, "Mercredi", "13h", "17h", 2)
- **Résolution** : solver = resoudre_modele(model).
- **Comptage** :

```
jourdex = JOURS.index("Mercredi")
heures_13_17 = [i for i, h in enumerate(HEURES) if "13h" <= h <= "17h"]
mat_indices = [MATIERES.index(m)+1 for m in
MATIERES_SCIENTIFIQUE if m in MATIERES]
total_occurrences = 0
Pour chaque classe dans CLASSES_BASE tel que "6e" est dans le nom :
for h in heures_13_17:
val = solver.Value(emploi_du_temps[(classe, jourdex, h)])
if val in mat_indices:
total_occurrences += 1
Assert total_occurrences == 2.
```

Critère de validation

Le nombre total d'heures (quel que soit la matière scientifique) occupées entre 13h et 17h le mercredi pour le niveau "6e" doit être exactement égal à 2.

TU011 - test_affectation_salles

Description

- Vérifier qu'un algorithme d'affectation de professeurs à des salles générales respecte :
  1. Les préférences de salle spécifiées dans config["preferences_salle_prof"].
  2. L'absence de double-affectation initiale (tout professeur sans affectation initiale).
  3. La répartition en rotation pour les professeurs sans préférence.
  4. Tous les professeurs doivent recevoir une salle parmi config["salles_generales"] (s'il n'en reste plus, on recycle la liste circulairement).

- **But du test** : garantir que chaque professeur listé dans config["professeurs"] se voit attribuer exactement une salle, en respectant d'abord ses préférences, puis le principe de rotation parmi les salles disponibles.
- **Environnement de test** : Pareil que les précédents

Contraintes

- config["professeurs"] peut être :
    - Un dictionnaire {matiere: nom_professeur} (chaque matière avec un professeur) ou un dictionnaire {matiere: {niveau: [liste_profs]}}.
- config["salles_generales"] doit être une liste non vide.
- config["preferences_salle_prof"] est un dictionnaire {professeur: salle_preferee}.
- On considère qu'il n'y a **aucune** affectation initiale (AFFECTATION = {} vide).
- Pour la rotation, on utilise l'ordre de available_salles = SALLES_GENERALES.copy() ; si on épuise la liste, on revient au début.

Dépendances

- Nécessite simplement la lecture de config_test.json.

Procédure de test

3. **Chargement** de config_test.json.
4. **Initialisation** :

```
PROFESSEURS = config["professeurs"]
SALLES_GENERALES = config["salles_generales"]
PREFERENCES_SALLE_PROF = config.get("preferences_salle_prof", {})
AFFECTATION = {} (vide).
```

5. **Constitution de la liste complete des professeurs sans salle** :
    - Parcourir PROFESSEURS :
        1. Si le type de prof est dict, récupérer chaînes et listes de professeurs.
        2. Sinon (str), ajouter ce nom s'il n'est pas dans AFFECTATION.
    - Ainsi on obtient professeurs_sans_salle (liste unique de noms de profs).
6. **Préparation du pool de salles** : available_salles = SALLES_GENERALES.copy(), affectation_prof = {}.
7. **Affectation des préférences** :
    - Pour chaque prof dans professeurs_sans_salle :
        1. Si prof est clef de PREFERENCES_SALLE_PROF, on fait :

```
affectation_prof[prof] = salle_pref
available_salles.remove(salle_pref)
professeurs_sans_salle.remove(prof)
```

8. **Rotation pour les salles :**
   Pour chaque (index, prof) dans enumerate(professeurs_sans_salle) :
   - salle_assignee = available_salles[index % len(available_salles)] si available_salles non vide, sinon on recycle SALLES_GENERALES.
   - affectation_prof[prof] = salle_assignee.
9. **Vérifications** :
   1. **Tous les professeurs doivent être présents** :
      - Construire all_profs : ensemble de tous les noms de professeurs (chaînes ou listes) dans config["professeurs"].
      - **Assert** que chaque élément de all_profs figure dans affectation_prof.keys().
   2. **Préférences respectées** :
      - Pour chaque (prof, salle) dans PREFERENCES_SALLE_PROF.items(), **assert** affectation_prof[prof] == salle.
   3. **Pour les profs sans préférence**, vérifier que affectation_prof[prof] ∈ SALLES_GENERALES.

Critère de validation

- Toutes les assertions doivent réussir :
  1. Chaque professeur a une salle associée.
  2. Les préférences explicites sont satisfaites.
  3. Aucun professeur sans préférence ne se voit affecter une salle hors de salles_generales.

TU012 - test_exclusivite_salles

Description

- Vérifier que la contrainte « exclusivité d'une salle pour au plus une classe par créneau » est bien respectée. Dans ce test minimal, on considère deux classes C1 et C2 pour le même jour et la même heure :
  - On crée deux variables entières « salle_C1 » et « salle_C2 » à domaine {0, 1, 2}, où la valeur 1 représente « Salle1 ».
  - On définit deux BoolVar « b1_C1 » et « b1_C2 » indiquant respectivement si C1 ou C2 se trouve en Salle1 à ce créneau.

    - On impose la contrainte d'exclusivité « b1_C1 + b1_C2 ≤ 1 » (au plus une des deux classes peut être dans Salle1).
    - On force un conflit volontaire en ajoutant « b1_C1 = 1 » et « b1_C2 = 1 » (les deux classes voudraient toutes deux avoir la Salle1 au même créneau).
- **But du test** : s'assurer que le modèle devient INFEASIBLE, car deux classes ne peuvent pas partager la même salle au même créneau.
- **Environnement de test** : Pareil que les précédent.

Contraintes

- Les salles sont codées par des entiers 0, 1 ou 2, où 1 correspond explicitement à « Salle_1 ».
- La contrainte clé est « b1_C1 + b1_C2 ≤ 1 ».
- Le scénario de violation est obtenu en ajoutant les deux clauses « b1_C1 = 1 » et « b1_C2 = 1 ».

Dépendances

- Aucune dépendance externe (pas de JSON, pas d'appels à d'autres fonctions).
- Il suffit du module ortools pour créer le modèle, les variables et les contraintes.

Procédure de test

1. **Initialisation du modèle** : créer un objet CpModel.
2. **Déclaration des variables** « salle_C1 » et « salle_C2 » avec domaine entier [0, 2].
3. **Création des BoolVar** « b1_C1 » et « b1_C2 » pour indiquer que C1 ou C2 est en Salle1 :
    o Pour chaque variable « b1_<classe> », on ajoute une contrainte « salle_<classe> == 1 » renforcée uniquement si le BoolVar vaut vrai, et « salle_<classe> != 1 » si le BoolVar vaut faux.
4. **Ajout de la contrainte d'exclusivité** : « b1_C1 + b1_C2 ≤ 1 ».
5. **Forçage du conflit** : ajouter simultanément « b1_C1 = 1 » et « b1_C2 = 1 ».
6. **Appel du solveur** (CpSolver).
7. **Vérification** : s'assurer que le statut retourné est INFEASIBLE. Si le solveur renvoie tout autre statut, le test échoue.

Critère de validation

Le solveur doit renvoyer INFEASIBLE, prouvant que deux classes ne peuvent pas occuper la même salle au même créneau.

TU013 - test_double_affectation_profs

Description

- Vérifier la contrainte « un même professeur ne peut pas enseigner dans deux classes différentes simultanément ». On construit un mini-modèle manuel avec :

  JOURS = [« Lundi »] (un seul jour, index 0)
  HEURES = [« 8h-9h »] (une seule plage horaire, index 0)
  MATIERES = [« MatiereX »]
  PROFESSEURS = { « MatiereX » : « M. X » }
  CLASSES = [« C1 », « C2 »]

  On souhaite forcer C1 et C2 à avoir toutes deux « MatiereX » au créneau (0, 0). Ensuite, on crée deux BoolVar (« b_C1 » et « b_C2 ») indiquant si la matière « MatiereX » est bien affectée pour chaque classe au créneau donné. On impose la contrainte « b_C1 + b_C2 ≤ 1 ». Or, une contrainte de volume horaire élémentaire (v1 + v2 = 2) force v1 = 1 et v2 = 1, ce qui équivaut à « MatiereX » présent dans C1 et C2 simultanément. Dès lors, la somme des deux BoolVar vaut 2, ce qui viole « b_C1 + b_C2 ≤ 1 ».
- **But du test** : s'assurer que le solveur renvoie INFEASIBLE, car le même professeur ne peut pas être présent dans deux classes en même temps.
- **Environnement de test** : Pareil que les précédents

Contraintes

- Les variables d'emploi du temps pour C1 et C2 au créneau (0, 0) sont des IntVar à domaine {1} (pour indiquer qu'elles doivent obligatoirement valoir l'indice 1 correspondant à « MatiereX »).
- On crée deux BoolVar v1 et v2 qui forcent ces deux IntVar à 1 (via OnlyEnforceIf), puis on ajoute la contrainte « v1 + v2 = 2 » pour garantir que les deux classes ont la matière en même temps—simulant un double emploi de M. X.
- Pour la contrainte de prof, on crée deux BoolVar « b_C1 » et « b_C2 » indiquant si « MatiereX » est présent pour C1 ou C2, et on ajoute « b_C1 + b_C2 ≤ 1 ».

Dépendances

- Aucune dépendance à un JSON ou à une fonction externe.
- L'unique dépendance est le module ortools.cp_model.

Procédure de test

1. **Définition d'un modèle CpModel**.
2. **Initialisation des listes** JOURS, HEURES, MATIERES, PROFESSEURS et CLASSES comme indiqué.
3. **Création de deux IntVar** pour « emploi_du_temps[("C1",0,0)] » et « emploi_du_temps[("C2",0,0)] » avec domaine {1}.
4. **Création des BoolVar v1 et v2** :
    - Pour v1, on contraint « emploi_du_temps[("C1",0,0)] == 1 » quand v1 = 1, et « emploi_du_temps[("C1",0,0)] != 1 » quand v1 = 0.
    - Pour v2, de même sur C2.
5. **Ajout de la contrainte « v1 + v2 = 2 »** (forçant v1 = 1 et v2 = 1).
6. **Création des BoolVar b_C1 et b_C2** pour la contrainte prof :
    - b_C1 vaut 1 si et seulement si « emploi_du_temps[("C1",0,0)] == 1 ».
    - b_C2 vaut 1 si et seulement si « emploi_du_temps[("C2",0,0)] == 1 ».
7. **Ajout de la contrainte « b_C1 + b_C2 ≤ 1 »**.
8. **Appel du solveur** (CpSolver).
9. **Vérification** : s'assurer que le statut retourné est INFEASIBLE. Si ce n'est pas le cas, le test échoue.

Critère de validation

Le solveur doit renvoyer INFEASIBLE, prouvant que plusieurs professeurs ne peuvent pas être affecté à deux classes simultanément.

6. Vérification de la documentation

L'objectif de cette phase est de s'assurer que la documentation fournie à l'utilisateur (manuel d'installation, manuel d'utilisation, description des contraintes, messages d'erreur…) est à jour et complète pour le fonctionnement de la version de l'application.

6.1 Méthodologie

10. Lecture systématique de la documentation :
    - Parcours complet du manuel d'installation.
    - Parcours du manuel d'utilisation et des différentes procédures décrites.
    - Lecture des messages d'erreur et les solutions proposées dans le manuel d'utilisation.
11. Exécution réelle des procédures :
    - Suivi pas à pas des instructions depuis un environnement vierge (installation, lancement, saisie de données, arrêt).
    - Vérification que toutes les commandes indiquées fonctionnent comme prévu (ex : python app.py, CTRL+C, accès à http://127.0.0.1:8050...).
12. Identification des écarts :
    - Signalement de toute procédure incorrecte, obsolète ou incomplète.
    - Comparaison entre les messages d'erreur réels et ceux décrits dans la documentation.
    - Vérification que les noms de fichiers, chemins et commandes sont cohérents entre l'interface, le code, et la documentation.

6. 2 Critères de validation

- Toutes les procédures décrites fonctionnent sans erreur dans un environnement réel.
- Aucune étape n'est manquante ou incorrecte.
- Les messages d'erreur documentés correspondent à ceux rencontrés en pratique.
- L'utilisateur est capable d'installer, lancer, utiliser et arrêter l'application uniquement à partir de la documentation.

7. Glossaire

Ce glossaire définit les principaux termes techniques ou spécifiques utilisés dans ce document afin d'en faciliter la lecture pour tous les profils de lecteurs.

Terme	Définition
1. Bibliothèque (ou package)	Ensemble de fonctions Python préprogrammées, installables via pip, utilisées pour étendre les capacités d'un projet.
2. Contrainte (Constraint)	Règle définie dans le solveur pour garantir la validité de l'emploi du temps (ex : « une salle ne peut être occupée que par une seule classe à la fois »).
3. Dash	Framework Python permettant de créer des interfaces web interactives accessibles via un navigateur.
4. Dépendance	Bibliothèque externe nécessaire au bon fonctionnement du programme (ex : Dash, OR-Tools).
5. Fichier JSON (.json)	Format texte structuré, utilisé ici pour stocker les données (classes, enseignants, matières, contraintes…).
6. Framework	Ensemble d'outils et de bibliothèques facilitant le développement d'une application structurée (ex : Dash).
7. Interface web	Interface utilisateur graphique accessible via un navigateur internet.
8. Invite de commande	Terme utilisé sous Windows pour désigner le terminal système.
9. OR-Tools	Bibliothèque d'optimisation par contraintes développée par Google, utilisée ici pour construire l'emploi du temps.
10. Plan de test	Document décrivant l'ensemble des scénarios de tests nécessaires à la validation fonctionnelle et technique d'un logiciel.
11. Projet TER	Travail d'Étude et de Recherche réalisé dans le cadre d'un cursus universitaire.
12. Python	Langage de programmation utilisé pour développer l'ensemble du projet.

13. Résolution de contraintes	Méthode algorithmique consistant à satisfaire un ensemble de règles pour produire une solution valide (emploi du temps, planning…).
14. Serveur local	Programme exécuté sur l'ordinateur de l'utilisateur, servant l'interface web localement.
15. Terminal	Interface en ligne de commande (sous macOS ou Linux) permettant d'exécuter des instructions système.
16. Test unitaire	Test automatique vérifiant le bon fonctionnement d'une fonction ou d'un composant isolé du programme.
17. Test d'intégration	Test vérifiant que plusieurs composants interagissent correctement ensemble.
18. Visualisation	Représentation graphique interactive des emplois du temps générés.

8. Références

Cette section présente l'ensemble des documents et ressources qui ont contribué à la conception de l'application, ainsi que des références utiles pour approfondir certains aspects techniques ou fonctionnels, que ce soit du point de vue de l'utilisateur ou du développeur.

- Documentation Python : utile pour comprendre l'environnement, les commandes de base, la gestion des paquets et la structure générale du code.
- Documentation Dash : pour le développement des interfaces web interactives en Python avec le framework Dash.
- Documentation OR-Tools (Google) : Documentation de la bibliothèque OR-Tools, utilisée pour la résolution de contraintes et l'optimisation des emplois du temps.
- Dépôt GitHub du projet Emploi du Temps - TER : Dépôt source contenant l'intégralité du code de l'application, les fichiers de test, les jeux de données, les scripts d'installation ainsi que la documentation interne.
- Cahier des charges.
- Guide d'installation : décrivant les étapes nécessaires pour installer et configurer l'environnement de développement (Python, dépendances, lancement local de l'application).
- Manuel d'utilisation : décrivant le fonctionnement de l'interface, les procédures courantes (saisie, calcul, export), ainsi que les messages d'erreur et leur résolution.
- Fichiers de configuration (data_interface.json, config_test.json)
- Fichiers structurés au format JSON servant respectivement à la saisie utilisateur (via l'interface) et aux tests automatiques (unitaires et d'intégration).

9. Index

Voici la liste les mots-clés du document et où les trouver dans celui-ci :