
Programmation impérative (avancé)

Projet

Licence informatique 2^{ème} année

Université de La Rochelle



Ce document est distribué sous la licence CC-by-nc-nd

<https://creativecommons.org/licenses/by-nc-nd/4.0/deed.fr>

© 2024-2025 Christophe Demko <christophe.demko@univ-lr.fr>



2024-2025_1



Les lignes commençant par \$ représente une commande *unix*. Le \$ ne fait pas partie de la commande proprement dite mais symbolise l’invite de commande.

Table des matières

| | | |
|----------|-------------------------------------|----------|
| 1 | Consignes | 1 |
| 1.1 | Groupes | 1 |
| 1.2 | Langue utilisée | 1 |
| 1.3 | Nommage | 2 |
| 1.3.1 | Fichiers | 2 |
| 1.3.2 | Types | 2 |
| 1.3.3 | Macros | 2 |
| 1.3.4 | Variables et fonctions | 2 |
| 1.4 | Style | 2 |
| 1.4.1 | Marque d’inclusion unique | 2 |
| 1.4.2 | Ordre des inclusions | 2 |
| 1.4.3 | Indentation | 3 |
| 1.5 | Structuration du code | 3 |
| 1.6 | Documentation | 3 |
| 1.7 | Tests | 3 |
| 2 | Sujet | 3 |
| 2.1 | Étapes du Projet | 4 |
| 2.2 | Livrables | 5 |
| 2.3 | Évaluation | 5 |
| | Historique des modifications | 5 |

1 Consignes

1.1 Groupes

-  **Le projet se fait par groupe de maximum 6 étudiants et est à rendre par un dépôt *moodle* le vendredi 23 mai 2025 à 23h00. Un retard raisonnable est toléré mais il sera pénalisé.**
-  **L'utilisation des générateurs de code de type *copilot* est fortement encouragée.**

1.2 Langue utilisée

La langue utilisée dans le code et la documentation devra être exclusivement l’anglais. Si vous avez des difficultés dans la langue de Shakespeare, vous pourrez utiliser les traducteurs automatiques :

- <https://www.deepl.com/translator>
- <https://translate.google.fr/>

- <https://chat.openai.com/>

1.3 Nommage

1.3.1 Fichiers

- les noms de fichiers du langage C seront tous en minuscules et en anglais. S'ils sont composés de plusieurs mots, ils devront être séparés par un tiret (–) (notation https://en.wikipedia.org/wiki/Letter_case#Kebab_case);
- les fichiers contenant le code devront avoir l'extension `.c` ;
- les fichiers d'en-têtes (exportables) devront avoir l'extension `.h` ;
- les fichiers destinés à être inclus dans votre code mais non exportables devront avoir l'extension `.inc`

1.3.2 Types

Les noms de types devront faire commencer chaque mot qui les compose par une majuscule. Il n'y a pas de sous-tirets (notation https://en.wikipedia.org/wiki/Letter_case#Camel_case). Les structures devront commencer par un sous-tiret (–) puis suivre la notation *Camel case* pour ne pas les confondre avec les noms de types.

1.3.3 Macros

Les macros (avec ou sans arguments) s'écrivent tout en majuscule en séparant les mots par des sous-tirets (–) (notation https://en.wikipedia.org/wiki/Letter_case#Snake_case).

1.3.4 Variables et fonctions

Les variables et les fonctions s'écrivent toutes en minuscules en séparant les mots par des sous-tirets (notation https://en.wikipedia.org/wiki/Letter_case#Snake_case).

1.4 Style

1.4.1 Marque d'inclusion unique

Chaque fichier d'en-tête devra posséder une marque permettant d'éviter les conséquences d'un fichier inclus plusieurs fois. Voir https://google.github.io/styleguide/cppguide.html#The__define_Guard

1.4.2 Ordre des inclusions

L'inclusion des fichiers d'en-tête devra respecter la logique suivante :

1. Inclusion du fichier directement lié au fichier `.c` qui l'inclut suivi d'une ligne vide ;
2. inclusion des fichiers d'en-tête du C standard suivis d'une ligne vide ;
3. inclusion des fichiers d'en-tête provenant d'autres bibliothèques suivis d'une ligne vide ;
4. inclusion des fichiers d'en-tête du projet suivi d'une ligne vide ;
5. inclusion des fichiers d'inclusion (extension `.inc`)

1.4.3 Indentation

Le style d'indentation devra être celui préconisé par Google <https://google.github.io/styleguide/cppguide.html#Formatting>. L'utilitaire `clang-format` (<https://clang.llvm.org/docs/ClangFormat.html>) supporte le style Google.

Vous pourrez utiliser l'utilitaire `cclint` pour vérifier votre code.

1.5 Structuration du code

Les champs des structures seront protégés à la manière de la bibliothèque `fraction` vue en travaux pratiques. Un soin sera tout particulièrement apporté à la structuration du code notamment en ce qui concerne les structures de données utilisées.

1.6 Documentation

La documentation sera générée avec l'outil `sphinx` et les fonctions seront documentées avec la norme de `doxygen`.

1.7 Tests

Des tests unitaires devront être implémentés, ils testeront chaque fonction et s'efforceront de vérifier que la mémoire est bien libérée au moyen de l'utilitaire `valgrind`.

Vous pourrez vous inspirer du projet <https://github.com/chdemko/c-arithmetic>.

D'une manière générale, toutes les options possibles décrites dans ce projet devront être implémentées.

2 Sujet

L'objectif de ce projet est d'étendre la bibliothèque CSP existante (**sans changer les types déjà définis**) pour inclure des algorithmes avancés de résolution de contraintes, notamment le "forward checking" et l'ordonnancement des variables et des valeurs. Vous devrez appliquer ces extensions à des problèmes classiques tels que les n -reines et le sudoku, puis comparer les performances des différentes approches.

Le projet pourra fournir un fichier README .md expliquant les instructions à exécuter avant de le configurer. Il devra fournir une documentation produite avec

> | \$ make docs

et devra contenir votre approche ainsi que vos difficultés et le cas échéant, ce qui ne fonctionne pas.

Il pourra être installé avec

> | \$ make install

2.1 Étapes du Projet

1. Compréhension de la Bibliothèque Existante :

- Étudier le code source de la bibliothèque CSP actuelle.
- Comprendre les structures de données et les algorithmes utilisés pour la résolution de base des CSP.

2. Implémentation du Forward Checking :

- Modifier l'algorithme de backtracking pour inclure le forward checking.
- Le forward checking consiste à vérifier les contraintes dès qu'une variable est assignée, afin d'éliminer les valeurs impossibles pour les variables non assignées.

3. Ordonnancement des Variables et des Valeurs :

- Implémenter des heuristiques pour l'ordonnancement des variables (par exemple, choisir d'abord la variable avec le moins de valeurs possibles restantes).
- Implémenter des heuristiques pour l'ordonnancement des valeurs (par exemple, essayer d'abord la valeur la moins contraignante).

4. Application aux Problèmes du Sudoku :

- Modéliser le problème du sudoku comme un CSP en s'inspirant du problème des n-reines dont la correction est donnée.
- Appliquer les extensions de forward checking et d'ordonnancement aux deux problèmes.

5. Comparaison des Performances :

- Mesurer le temps de résolution et le nombre de nœuds explorés pour chaque problème (n-reines et sudoku) avec et sans les extensions.
- Comparer les résultats pour évaluer l'impact des extensions sur l'efficacité de la résolution.

6. Rapport :

- Rédiger un rapport détaillant les modifications apportées à la bibliothèque, les résultats des tests de performance, et les conclusions tirées.

2.2 Livrables

- Code source modifié de la bibliothèque CSP avec les extensions.
- Implémentations des problèmes des n-reines et du sudoku.
- Rapport détaillant les modifications, les résultats de performance, et les conclusions.

2.3 Évaluation

Le projet sera évalué sur la base de :

- La correctitude et l'efficacité des implémentations.
- La clarté et la précision du rapport.
- La capacité à expliquer et à justifier les choix d'implémentation.
- L'impact mesuré des extensions sur les performances de résolution.

Ce projet vous permettra de comprendre en profondeur les algorithmes de résolution de contraintes et d'appliquer ces connaissances à des problèmes concrets.

Historique des modifications

2024-2025_1 *Lundi 17 mars 2025*

Dr Christophe Demko <christophe.demko@univ-lr.fr>

- Version initiale