**Projet Info LDD2**

État des lieux du projet

Noms, prénoms des membres du groupe :­ GUEYE Maram Sall et AL MASRI Reina

***Remarque :*** *Cet état des lieux a été établi à partir de nos connaissances actuelles ; il est possible que certains éléments aient échappé à notre attention*.

→ Expliciter l’approche choisie et l’état actuel pour chaque thème :

Fonctionnel / fonctionnel sur certains exemples / non fonctionnel

→ Les erreurs et points à améliorer sont à détailler dans la colonne Remarques.

→ Dans Remarque, préciser si les question bonus ont été abordées.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **TD** | **Thèmes** | Code : classe / méthode | Tests | Remarques |
| 1 | Implémentation de graphes, manipulation | Deux classes principales sont définies : Node, pour représenter les nœuds individuellement, et OpenDigraph, pour représenter le graphe complet. Les méthodes sont réparties entre des méthodes locales (dans Node) et globales (dans OpenDigraph), afin de faciliter les tests unitaires et la réutilisabilité. Chaque nœud gère lui-même ses parents et ses enfants via des dictionnaires de la forme {id : multiplicité}, permettant ainsi la prise en charge d’arêtes multiples et une gestion fine de celles-ci. Fonctionnel. | Les fonctionnalités sont validées par des tests unitaires, garantissant leur bon fonctionnement et leur robustesse. |  |
| 2 | remove (parent, edge…), is\_well\_formed, add\_input\_node, add\_output\_node | Avant chaque opération critique (comme remove ou add\_input\_node), des vérifications de l’existence de l’ID sont effectuées afin d’éviter des erreurs inattendues. Les fonctions internes sont également réutilisées pour favoriser la cohérence et limiter la redondance du code.  Fonctionnel | Les fonctionnalités sont validées par des tests unitaires, garantissant leur bon fonctionnement et leur robustesse. |  |
| 3 | Traductions graphe/matrice d'adjacence ; génération  aléatoire de matrice / graphe | **Une correspondance bidirectionnelle est établie entre les graphes orientés et leurs matrices d’adjacence, permettant une conversion efficace dans les deux sens. Cette abstraction autorise la génération de graphes aléatoires sous contraintes à partir de manipulations matricielles, évitant ainsi les opérations directes sur les structures de nœuds et d’arêtes. Cette approche s’inscrit dans une logique de séparation des responsabilités : les mécanismes liés à la représentation et à la manipulation du graphe sont découplés des algorithmes de génération aléatoire, favorisant une architecture plus modulaire, testable et extensible.**  Fonctionnel | Fonctionnalité(s) validée(s) par des tests unitaires Les fonctionnalités sont validées par des tests unitaires, garantissant leur bon fonctionnement et leur robustesse. | La question bonus a également été traitée et intégrée à la solution proposée. |
| 4 | Sauvegarde (graphe vers .dot ; .dot vers graph) et représentation de graphes. Affichage (display) | L’approche retenue est conforme aux recommandations du sujet, en s’appuyant sur le format standard .dot pour représenter les graphes, tout en intégrant des attributs personnalisés. Le graphe est d’abord exporté au format digraph G, avec des nœuds nommés et annotés selon les besoins. L’affichage est ensuite pris en charge via des outils système (os, webbrowser), permettant une visualisation directe. Par ailleurs, un paramètre verbose a été introduit pour faciliter le débogage et le suivi des opérations.  Fonctionnel | La sauvegarde et l’affichage des fichiers .dot font l’objet d’une vérification directe, garantissant la validité du fichier généré ainsi que la bonne ouverture via les outils système. |  |
| 5 | Classe pour les circuitds booléens, méthode is\_cyclic, is\_well\_formed de  bool\_circ, shift\_indices | La classe bool\_circ hérite de open\_digraph et représente un circuit booléen. Son instanciation à partir d’un graphe arbitraire n’impose pas immédiatement de contraintes structurelles spécifiques, offrant ainsi une certaine souplesse initiale. La validité en tant que circuit booléen est ensuite assurée par la méthode is\_well\_formed(), qui vérifie que la structure respecte un ensemble de règles fondamentales.  La détection de cycles est gérée par la méthode is\_cyclic(), qui repose sur un tri topologique basé sur les degrés entrants. Afin de préserver l’intégrité du graphe d’origine, l’algorithme s'exécute sur une copie du graphe.  Un aspect crucial de l’implémentation concerne la fusion de graphes, notamment à travers l’utilisation des fonctions shift\_indices, min\_id, et max\_id, permettant de gérer correctement les identifiants lors de combinaisons multiples.  Enfin, la méthode \_\_init\_\_ de bool\_circ intègre une validation systématique via un appel à is\_well\_formed(), garantissant que tout objet instancié respecte les contraintes requises d’un circuit booléen valide.  Fonctionnel | Chaque fonctionnalité est validée par des tests unitaires Chaque fonctionnalité développée fait l’objet de tests unitaires dédiés, assurant leur validité. |  |
| 6 | Compositions & connectivité ; iparallel, icompose, identity, connected\_components d'open\_digraph | L’approche adoptée repose sur une décomposition des fonctionnalités, avec un objectif principal de préserver les graphes originaux. En effet, à l'exception des méthodes spécifiées, chaque opération retourne une nouvelle instance sans altérer les graphes d’origine. Pour éviter toute collision d’identifiants, les indices des nœuds sont ajustés à l’aide de la fonction shift\_indices. Les graphes sont ensuite parcourus en profondeur (DFS) afin d’identifier les sous-ensembles disjoints. Chaque composant connexe ainsi détecté est extrait et représenté sous forme de sous-graphe autonome.  Fonctionnel | Chaque fonctionnalité est validée par des tests unitaires Chaque fonctionnalité développée fait l’objet de tests unitaires dédiés, assurant leur validité. | Le bonus sur l'identité a été abordé. |
| 7 | Longueur de chemins : l'algorithme de Dijkstra, shortest\_path, associer à chaque ancêtre commun des deux nœuds sa  distance à chacun des deux nœuds | L'algorithme de Dijkstra a été personnalisé pour s'adapter spécifiquement à un graphe orienté, en introduisant un paramètre de direction. De plus, un paramètre tgt a été ajouté afin d'arrêter l'algorithme dès que la distance vers le nœud cible est déterminée, optimisant ainsi les performances. En ce qui concerne les ancêtres communs, l'algorithme conserve les nœuds accessibles depuis les deux sources et associe à chaque nœud la distance respective depuis les points de départ u et v.  Fonctionnel | Chaque fonctionnalité est validée par des tests unitaires Chaque fonctionnalité développée fait l’objet de tests unitaires dédiés, assurant leur validité. |  |
| 8 | Tri topologique, chemin le plus long, mixins. | **L’approche repose sur l’idée des** co-feuilles **: nous supprimons récursivement les co-feuilles et formons des couches successives. Ensuite, nous appliquons un tri topologique pour visiter les nœuds dans l’ordre, et nous stockons, pour chaque nœud, la distance maximale depuis la source. La reconstruction du chemin est réalisée à l’aide d’un dictionnaire** prev**, qui permet de retracer les étapes du parcours.**  L’utilisation de mixins a facilité la séparation des responsabilités, améliorant ainsi la modularité et la clarté du code.  Fonctionnel | Chaque fonctionnalité est validée par des tests unitaires Chaque fonctionnalité développée fait l’objet de tests unitaires dédiés, assurant leur validité. |  |
| 9 | Synthèse de circuit via une formule propositionnelle. | La formule est analysée caractère par caractère en utilisant une pile pour gérer les niveaux d'imbrication des parenthèses. Les variables sont créées comme nœuds d'entrée. Les opérateurs sont ajoutés comme nœuds intermédiaires. Les parenthèses définissent la structure du circuit avec la priorité des opérations.  Fonctionnel | Une vérification directe est effectuée pour garantir l'intégrité du processus. | La partie bonus a été effectuée |
| 10 | Circuits booléens aléatoires ; Additionneur. | Utilisation de random\_graph pour générer un DAG avec des nœuds étiquetés aléatoirement comme portes logiques. Et Ajustement des entrées/sorties pour respecter les contraintes  Fonctionnel | Une vérification directe est effectuée pour garantir l'intégrité du processus. | Les deux exercices du bonus ont étés fait.  La profondeur linéaire du half adder doit être optimisée. |
| 11 | Réécriture : évaluation de circuits. | **L'évaluation des circuits se fait via deux méthodes principales : la simplification itérative et l'évaluation directe. On utilise le tri topologique pour l'ordre d'évaluation, avec des raccourcis pour les cas particuliers et une gestion des portes complexes.**  fonctionnel | Chaque fonctionnalité est validée par des tests unitaires Chaque fonctionnalité développée fait l’objet de tests unitaires dédiés, assurant leur validité. |  |
| 12 | Réécriture : vérification d'un code de Hamming. | Deux circuits sont implémentés pour le code de Hamming , l’**Encodeur** Et le **Décodeur. Utilisation de Portes XOR pour l’Efficacité. Et Détection d’Erreur via Syndromes car** Les syndromes forment un mot binaire (s0s1s2) donnant la position du bit erroné. Et g**énéricité et Extensibilité**  fonctionnel | Chaque fonctionnalité est validée par des tests unitaires Chaque fonctionnalité développée fait l’objet de tests unitaires dédiés, assurant leur validité. | On peut optimiser le code de Hamming en respectant les limites théoriques confirmées. |