Sorbonne Université
Faculté de Science et d'ingénierie
Département Informatique

Rapport du PSTL

Informatique

Spécialité :
Science et Technologie Logiciel

Thème

Génération et réparation d'instances pour JSON Schema

Encadré par

- Mohammed-Amine Baazizi
- Lyes Attouche

Réalisé par

- Tabellout Salim
- Tabellout Yanis
- Bouzourine Hichem

Soutenu le : **21/05/2024**

TABLE DES MATIÈRES

TABLE DES FIGURES

INTRODUCTION

JSON Schema est un standard incontournable pour décrire et valider les données JSON. Sa simplicité et sa flexibilité en font un outil précieux pour garantir la cohérence et l'intégrité des données échangées ou traitées. Si de nombreux outils existent pour générer des instances JSON à partir d'un schéma, la correction du résultat n'est jamais garantie. Le risque de se retrouver avec des instances non conformes est réel, ce qui peut causer des problèmes lors de l'échange ou du traitement des données.

La réparation d'instances JSON s'attaque à ce problème en proposant des techniques pour modifier les instances non conformes et les rendre valides. L'objectif est de minimiser les modifications nécessaires tout en garantissant la cohérence des données.

Le but de ce travail est d'étudier la réparation d'une instance J relativement à un schéma S avec l'hypothese que J satisfait partiellement S. Le probleme est celui d'identifier les fragements de S qui ne sont pas satisfaits par J puis de réparer J en conséquence. Comme il n'existe pas d'instance J' unique obtenue de la réparation de J, le but est de produire un J' qui soit le plus proche de J. Pour ce faire, nous exploitons la distance d'édition entre documents JSON proposée dans [1].

Intégration avec les Objectifs du Projet

Dans le cadre du projet, les objectifs visent la génération et la correction d'instances JSON conformes à un schéma initial, tout en minimisant les modifications nécessaires.

1. **Validation initiale :** Les générateurs d'instances identifiés dans l'objectif 1 produisent des données JSON à partir des schémas. La première étape consiste à valider ces instances par rapport au JSON Schema, identifiant ainsi les non-conformités.
2. **Analyse des erreurs de validation :** L'objectif consiste à étudier le lien entre les erreurs de validation, détectées à l'étape 1, et la distance d'édition entre les instances non conformes et l'instance valide. Cette analyse contribue à une compréhension approfondie des types d'erreurs et guide le processus de réparation.
3. **Réparation des instances :** L'objectif global du projet est de développer des approches de réparation permettant de minimiser les modifications nécessaires pour rendre une instance non conforme conforme au schéma initial.

Le travail réalisé et présenté dans ce mémoire est structuré comme suit :

- **Chapitre 1** : Ce chapitre expose des généralités sur le JSON schema, la génération d'instances, et la validation des instances.
- **Chapitre 2 :** Ce chapitre représente les différentes études expérimentales menées durant notre étude.
- **Chapitre 3 :** Ce chapitre met en avant une approche des réparation d'instance.

Approche de travail

Dans cette initiative, nous cherchons à examiner différentes approches pour améliorer la réparation des instances. Voici les principaux axes que nous souhaitons explorer :

1. Maîtrise du langage de JSON Schema et de la validation associée.
2. Compréhension de la distance d'édition sur les documents Json.
3. Analyse des liens entre la distance d'édition et les erreurs de validation.
4. Proposition d'une méthode de réparation d'instance efficace.
5. Proposition des éventuelles pistes à explorer pour la réparation.

CHAPITRE 1

ETAT DE L'ART

1. JSON Schema

Le JSON Schema [2] est une norme permettant de décrire la structure et les contraintes des données au format JSON (JavaScript Object Notation). Il spécifie la manière dont les données JSON doivent être organisées, les types de données autorisés, les valeurs par défaut, etc. Le langage est basé sur le format JSON et utilise des mots clés dédiés pour définir des contraintes sur la structure des données. Ces mots-clés sont regroupés en familles et permettent de décrire pour chaque type de données les informations suivante [3] :

- Pour les nombres, il est possible de contraindre leur intervalle de valeur en utilisant les mots-clés `minimum` et `maximum`. On peut également imposer qu'ils soient multiples d'un certain nombre avec le mot-clé `multipleOf`.
- Pour les chaînes de caractères, on dispose de plusieurs possibilités. Le mot-clé `minLength` permet de spécifier la longueur minimale d'une chaîne, tandis que `maxLength` définit la longueur maximale. On peut également imposer des motifs de caractères à respecter en utilisant le mot-clé `pattern`. De plus, il est possible de définir une liste de valeurs acceptées avec le mot-clé `enum`.
- Pour les objets, on peut décrire les propriétés attendues en utilisant `required` et préciser le type de chaque propriété en utilisant `properties` et `patternProperties`. On peut imposer une longueur au moyen de `minProperties` et `maxProperties`.
- Pour les tableaux, le mot-clé `items` est utilisé pour définir le type des éléments du tableau. On peut également spécifier la taille minimale et maximale du tableau avec les mots-clés `minItems` et `maxItems`, ainsi que d'autres contraintes telles que l'existence d'un élément satisfaisant une contrainte `contains`.
- L'assertion `type` permet d'imposer le type de l'instance à valider. Par défaut, les autres assertions expriment une implication implicite.

1.1. Utilité du JSON Schema

1. **Validation des données :** Il permet de valider si une instance JSON est conforme à un schéma prédéfini, assurant ainsi la qualité et la cohérence des données.
2. **Documentation :** En décrivant la structure des données attendues, le JSON Schema sert également de documentation explicite pour les utilisateurs et les développeurs.
3. **Communication :** En partageant un schéma, différentes parties prenantes peuvent avoir une compréhension commune de la structure des données, facilitant ainsi l'échange d'informations.
4. **Génération de données de test :** Il peut être utilisé pour générer des jeux de données de test conformes au schéma, ce qui est utile lors de la phase de développement et de tests.

1.2. Grammaire JSON Schema

La syntaxe de JSON Schema a été formalisée dans [4] pour le Draft 06. Nous la rappelons ci-dessous et illustrons, avec un exemple, son utilsation pour décrire des objets complexes.

- **JSDoc :=** {(defs,) ? JSch}
- **Defs :=** "definitions : {string :{JSch} (,string : JSch)* }
- **JSch :=** strSch | numSch |intSch | objSch | arrSch | refSch | not | allOf | anyOf | enum
- **not :=** "not" : {JSch}
- **strSch :=** "type" : "string" (, strRes)*
- **numSch :=** "type" : "number" (, numRes)*
- **objSch :=** "type" : "object" (, objRes)*
- **arrSch :=** "type" : "array" (, arrRes)*
- **refSch :=** "$ref" : "# JPointer"

En ce qui concerne les règles complémentaires, il est nécessaire de se référer à l'article [4].

Exemple

```json
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Exemple de schema JSON",
  "type": "object",
  "properties": {
    "nom": {
      "type": "string",
      "description": "Le nom de la personne.",
      "minLength": 1,
      "maxLength": 255
    },
    "prenom": {
      "type": "string",
      "description": "Le prenom de la personne.",
      "minLength": 1,
      "maxLength": 255
    },
    "adresse": {
      "type": "object",
      "description": "L'adresse de la personne.",
      "properties": {
        "code_postal": {
          "type": "string",
          "description": "Le code postal de la ville.",
          "pattern": "^[0-9]{5}$"
        }
      }
    }
  },
  "required": ["nom", "prenom", "age"]
}
```

FIGURE 1.1 – Exemple de Schéma JSON

Ci-dessous la description du schéma JSON, tel que :

- **$schema :** Indique la version du schéma JSON utilisée.
- **title :** Le titre du schéma.
- **description :** Une description du schéma.
- **type :** Le type de l'objet JSON (ici, un objet).
- **properties :** Définit les propriétés de l'objet JSON.

- **required :** Indique les propriétés obligatoires.

Le **schéma 1.1** décrit des objets ayant obligatoirement les propriétés : *nom, prenom, age*, mais la propriété *addresse* n'est pas requise . L'exemple ci-dessous représente instance JSON qui respecte le schema 1.1, où le nom, prenom et l'age sont présent et bien typé, ainsi qu l'addresse qui n'est pas requise :

```
{
     "nom": "Dupont",
     "prenom": "Jean",
     "age": 30,
     "adresse": {
       "code_postal": "75001",
     }
}
```

FIGURE 1.2 – Exemple de document respectant le schema

2. Validation

La **validation** d'une instance J pour un schéma S consiste à verifier la conformité de J par rapport aux contraintes définies dans S. Le probleme de validation a été étudié dans [4] et plus récemment dans [5] pour un draft plus récent. La complexité de la validation pour les draft 7 est polynomiale en la taille de l'instance et du schéma.

La specification de JSON Schema décrit la sortie standard d'un validateur conforme à cette specification. Cette sortie est exhaustive et indique la validité de chaque expression du schéma S pour l'instance J. De ce fait, toute violation des règles spécifiées dans le schéma est *détectée et signalée*.

Exemples

Soit le schema json suivant :

```json
{
   "type": "object",
   "properties": {
       "name": { "type": "string" },
       "birthday": { "type": "string","format": "date" },
       "age": { "type": "integer" ,"minimum" : 0 , "maximum" : 100},
       "federation":{
           "type":"object",
           "properties":{
               "name":{"type":"string"},
               "role":{"type":"string"}
           },
           "required":["name"]
       }
   },
   "required":["name","birthday","age","federation"]
 }
```

FIGURE 1.3 – Schema JSON

Le schéma JSON ci-dessus représente la structure des instances JSON d'athlètes. Les informations pour chaque athlète incluent son nom, sa date de naissance, son âge et sa fédération, laquelle est définie par son nom et son rôle.

```json
{
  "name" : "Chris Bumstead",
  "birthday" : "1995-02-05",
  "age" : 29,
  "federation" :{
    "name" : "IFBB",
    "role" :"bodybuilding"
  }
}
```

(a) Instance Valide

```json
{
  "name" : "Chris Bumstead",
  "birthday" : "1995-02-05",
  "age" : 29
}
```

(b) Instance Invalide

FIGURE 1.4 – Instance valide et invalide par rapport au schema

L'instance **(b)** de la **Figure 1.4** est invalide. En effet, elle ne respecte pas le schéma **(1.3)** car l'attribut "federation" est manquant.Voici plus précisément ce que pourrait indiquer le résultat d'un validateur. :

```json
{
    "valid": false,
    "instanceLocation": "",
    "keywordLocation": "",
    "errors": [
        {
            "instanceLocation": "",
            "keywordLocation": "/required",
            "error": "The object is missing required properties ['federation']"
        }
    ]
}
```

FIGURE 1.5 – Output du validateur

3. La similarité des documents JSON

La similarité des documents JSON est une mesure de la similarité entre deux documents JSON. Elle est généralement utilisée pour comparer des documents JSON qui représentent des objets ou des données similaires.

3.1. Approches existantes

Une des approches existantes pour calculer la similarité des documents JSON est :

- **Approche top-down [6] :** Cet approche top-down pour un comparateur de similarité dans le contexte JSON consiste à examiner la similarité entre deux structures JSON en commençant par les éléments les plus généraux et en descendant progressivement vers les détails spécifiques. Cela implique une comparaison basée sur la hiérarchie des éléments plutôt que sur les valeurs individuelles. ensuite les valeurs des propriétés et des éléments des deux documents.

3.2. Limitations :

1. **La structure du document est ignorée :** les approches top-down ignorent la structure du document, ce qui peut conduire à des résultats inexacts car dans un document JSON, on retrouve des éléments ordonnées (*array*), et non ordonnées (*objects*).
2. **Aucune garantie de qualité n'est donnée :** les approches existantes ne fournissent généralement aucune garantie de qualité pour leurs résultats.

4. JEDI

JEDI [1] est un algorithme de calcul de la similarité entre deux documents JSON. Il fonctionne en comparant les deux documents en tant qu'arbres en prenant en compte la structure du document (*ordonnée vs non ordonnée*). La similarité entre les deux documents est définie comme le nombre minimum d'opérations d'édition (Ajout, Suppression, Modification) nécessaires pour transformer un arbre en l'autre.

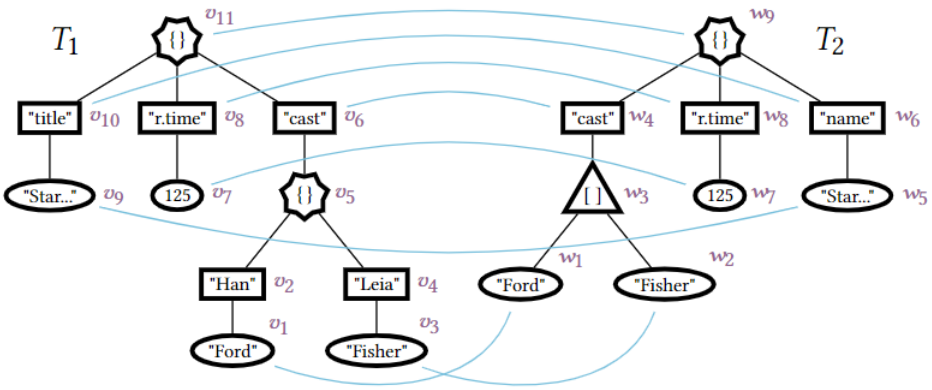


FIGURE 1.6 – Exemple de nombre d'opération de transformation [1]

4.1. Représentation d'Arbres JSON

JEDI introduit la notion d'arbre JSON qui est une représentation arborescente d'un document où chaque valeur du document est représenté par un noeud dans l'arbre, nous permettant ainsi d'exploiter certaines propriétés des arbres qui nous sera utile par la suite. La figure 1.7 illustre cette notion :

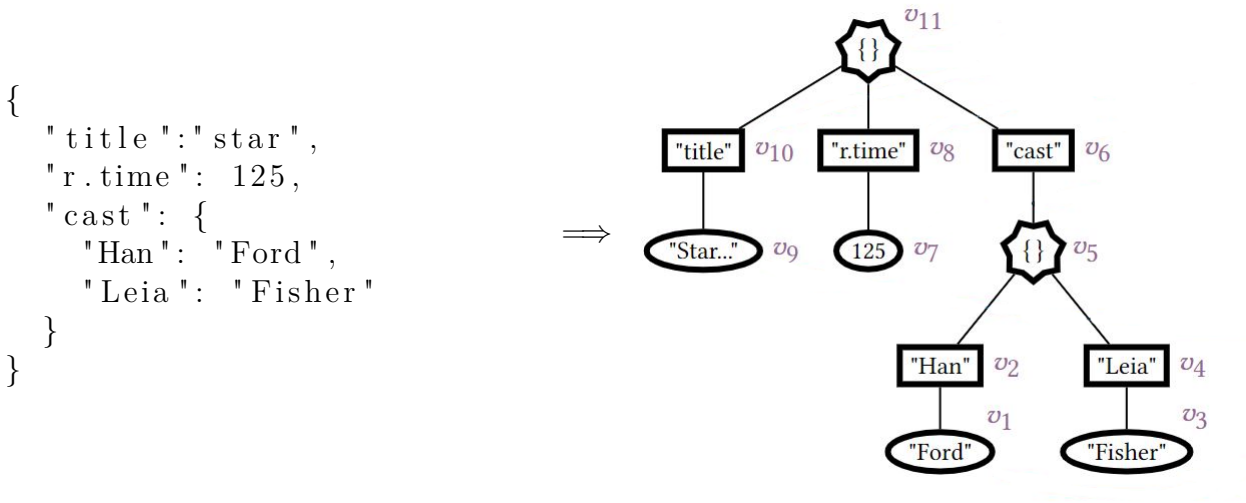


FIGURE 1.7 – Transformation d'un document json en arbre Json

4.2. Distance d'édition (Tree Edit Distance : TED)

JEDI permet de calculer la **distance d'édition** entre **deux arbres JSON**. On définit **TED** comme étant le nombre d'opération minimale pour transformer un arbre T_1 à l'arbre T_2. Ces opérations sont : *ajout, suppression, renomer un noeud.* désignent les opérations qui permettent le **Json Edit Mapping**, c'est à dire faire correspondre chaque noeud du premier arbre vers le deuxième soit :

- Suppression : Les noeuds de T_1 qui ne figurent pas dans T_2 sont supprimés
- Ajout : Les noeuds de T_2 qui ne figurent pas dans T_1 sont ajoutés
- Renommer : Les noeuds de T_1 qui figurent dans T_2 sont renommés

Chaque opération est associé à un coût, sauf le cas de renommer le même noeud [1]

4.3. Distance et Matrice d'édition

Jedi permet de construire une matrice appelé **Matrice d'édition** qui sauvegarde la distance d'édition entre chaque noeuds de T_1 avec les noeuds de T_2, la figure suivante illustre la matrice de l'exemple précédent :

1. Le coût de renommer $v8$ en $w8$ est nulle **(Figure 1.6)** car il s'agit du même noeud

dt	ϵ	w_1	w_2	...	w_6	w_9
ϵ	0	1	1	...	2	9
v_9	1	1	1	...	1	8
v_{10}	2	2	2	...	1	8
⋮	⋮	⋮	⋮	⋱	⋮	⋮
v_6	6	5	5	...	6	8
v_{11}	11	10	10	...	10	5

FIGURE 1.8 – Matrice d'édition entre T_1 et T_2

Exemple : Voici deux instances qui décrivent les caractéristiques d'un arbre, notamment sa hauteur et sa largeur. La matrice d'édition permet d'estimer le coût de transition du premier document vers le deuxième, avec un coût estimé à 4, qui se décompose comme suit :

- Suppression des accolades : coût de 2 [2]
- Suppression des attributs "height" et "width" : coût de 2

```
{
    "tree":[
        {"height" : 300},
        {"width" : 20}
    ]
}
```

(a) Json T_1

```
{
    "tree":[
        300,
        20
    ]
}
```

(b) Json T_2

dt	{}	tree	[ ]	300	20
{}	4	5	6	8	8
tree	5	4	5	7	7
[ ]	6	5	4	6	6
{}	3	2	2	2	2
height	3	2	1	1	1
300	4	3	2	0	0
{}	3	2	2	2	2
width	3	2	1	1	1
20	4	3	2	0	0

FIGURE 1.9 – Matrice d'édition

Remarque *Jedi permet de déterminer le coût de modification, mais il ne fournit pas les opérations à effectuer.*

2. une pour chaque élément de la liste

5. Réparation d'instances

Soit un schéma JSON et une instance qui ne le respecte pas. La validation de l'instance produit un arbre d'erreurs indiquant les parties de l'instance qui ne correspondent pas au schéma. L'objectif de la réparation d'instance est de proposer des modifications à l'instance pour la rendre conforme au schéma. L'idée pour réparer l'instance, est de considérer l'arbre d'erreurs comme un arbre dont les feuilles sont des erreurs simple à corriger. En partant de la racine de l'arbre, on parcourt chaque noeud et on applique une transformation appropriée à l'instance. Les transformations dépendent de le type d'erreur, par exemple :

- **Valeur non conforme.**
- **Propriété manquante.**
- **Structure incorrecte.**

5.1. Problème de répération

La réparation d'instance nous force à vérifier à chaque réparation d'une feuille de réverifier la conformité au schéma si nous n'avons pas cassé cette dernière. Soit l'exemple suivant

```
{
    "type":"object",
    "required":["a"],
    "properties":{
        "a":{
          "type":"number",
          "allOf" : [
             {"minimum":7},
             {"multipleOf":3}
           ]
        }
    }

}
```

(a) Schema Json

```
{
  "a":5
}
```

(b) Instance non valide

Dans ce schéma, deux erreurs sont identifiées : une concernant la valeur minimale et l'autre le multipleOf. Étant donné que le schéma est de forme conjonctive (allOf), en tentant de réparer cette instance en traitant les erreurs séparément (chaque erreur correspond à une branche dans le schéma), il est possible de générer un nombre supérieur à 7 qui n'est pas un multiple de 3. Une solution envisageable serait de produire une instance qui respecte à la fois le minimum de 7 et qui soit un multiple de 3, ce qui implique de transformer le schéma en forme disjonctive.

Conclusion

Dans ce chapitre, nous avons introduit des notions de base ainsi que des études récentes qui seront essentielles pour explorer de nouvelles approches de réparation des instances JSON. Dans la section suivante, nous entreprendrons une étude expérimentale afin d'identifier un lien potentiel entre les erreurs de validation et la distance d'édition.

CHAPITRE 2

ETUDE EXPÉRIMENTALE

Introduction

Dans le chapitre précédent, nous avons introduit certaines notions essentielles pour appréhender la suite de notre étude expérimentale. Dans la section suivante, nous cherchons à explorer une nouvelle piste qui semble prometteuse pour l'approche de réparation. Nous étudierons le lien entre la distance d'édition et les erreurs relevées lors de la validation, en envisageant la possibilité d'effectuer des réparations à l'aide de la matrice d'édition.

1. Génération d'instance

1.1. Defintion

La génération d'instance dans le contexte de JSON schema est la génération des instances JSON conforme à un schéma de référence. Cette tâche est complexe du à la forme non algébrique du JSON schema. Un langage est dit **algébrique** quand l'application et la séméntique de ses opérateurs dépéndent de la sémantique de leurs opérandes, pour json nous avons l'interaction syntaxique ainsi que sémantique.[7] La génération d'instances JSON Schema permet aussi de créer des ensembles de données valides à partir d'un schéma JSON défini. Ce processus est crucial pour divers cas d'utilisation, tels que la création de jeux de test, le remplissage de bases de données et l'exploration de l'espace de solutions défini par le schéma [3]. Pour celà on dispose des schémas, ainsi que des instances correctes, et nous utilisons trois générateurs d'instance, ces instances seront appelé **witness**. Voici certains générateurs d'instance dont on dispose :

- **json-schema-faker (JSF) :** pour une génération rapide et simple de données fictives[8].

- **json-everything (JE) :** écrite en C#, est extension de **System.Text.Json**, limitée en termes d'expressivité sur la partie JSON[9].
- **json-data-generator (DG) :** pour une prise en charge complète de JSON Schema Draft 7 et la génération de données aléatoires[10].

1.2. Tableau comparatifs des génerateurs

Générateur	**Dataset**	**Total** (schémas)	**Instances** (Valid/Invalid)	**Taux de validité** / Dataset (%)	**Taux de validité** général (%)
json-schema-faker	Snowplow	409	402/7	98.28%	
	Washington Post	125	102/23	81.60%	
	Kubernetes	1082	984/98	90.94%	89.75%
	GitHub	5957	5254/703	88.19%	
json-everything	Snowplow	409	307/102	75.06%	
	Washington Post	125	48/77	38.40%	
	Kubernetes	1082	699/383	64.60%	59.30%
	GitHub	5957	3524/2433	59.15%	

TABLE 2.1 – Taux de Validité des instances générées [3]

Remarque

Le dataset contenant les witness a été déjà construit précédemment, donc on mènera nos tests dessus.

2. Validateur

Plusieurs possibilité s'offrent pour le choix du validateur, cependant nous avons trouvé que le **Jschon** [11]validateur nous permet d'exploiter les erreurs de manière arborescente.

3. Dataset

Le dataset déjà conçu contient 6000 schéma ainsi que les witness pour chacun des trois générateurs. Certains witness n'ont pas été généré correctement c'est pour celà durant notre étude experimentale, nous allons essayer d'ignorer certaines valeurs aberrantes.

4. Prétraitement

4.1. Génération de l'exec TED

L'outil JEDI a été développé en *C++*. Bien que la génération de l'exécutable ait initialement manqué de documentation, une fois que nous avons réussi à la générer, nous avons soumis une demande de modification (pull request) pour améliorer la documentation de l'outil. Cette demande a été acceptée [12].

4.2. Bracket Notation

L'outil JEDI permet de calculer la distance d'édition entre deux arbres, cependant un certain prétraitement nécessaire pour son utilisation. C'est pour celà JEDI propose d'utiliser une notation "Bracket", ce qui nous pousse à transformer les fichiers JSON en des fichiers bracket qui nous sera utile par la suite. Le script de migration du JSON vers Bracket a été déjà fourni.

4.3. Migration de Draft

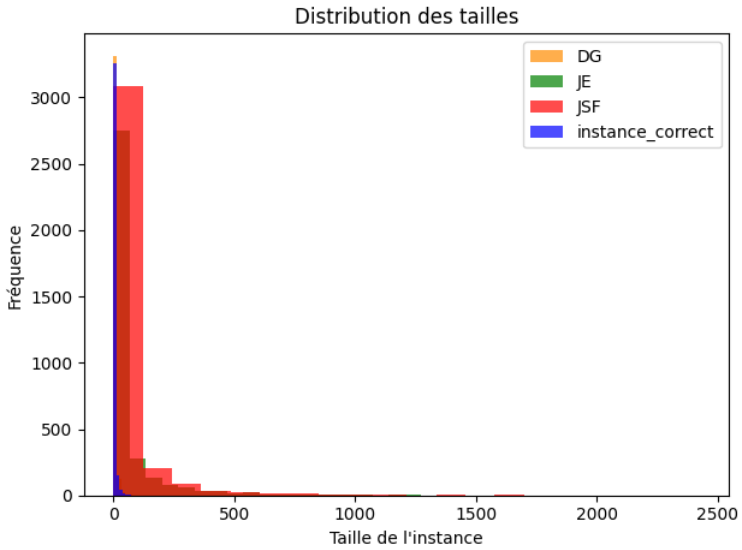
JSON Schema se base sur des versions ou des **draft** ou chaque version diffère de celle qui l'a précède. Étant donné que le validateur qu'on utilisera nécessite une draft plus récente que celle de notre dataset, nous devons donc faire migrer nos schémas du draft 4 vers le draft 2019 ou 2020. Pour celà nous avons utilisé la librairie [13] a été fourni qui fait la migration

5. Distance d'édition et nombre erreurs des instances

Une première approche serait de voir si il existe un lien entre la distance d'édition et le nombre d'erreurs pour chaque witness, celà en calculant la distance d'édition entre chaque instance correct et les witness de chaque générateurs

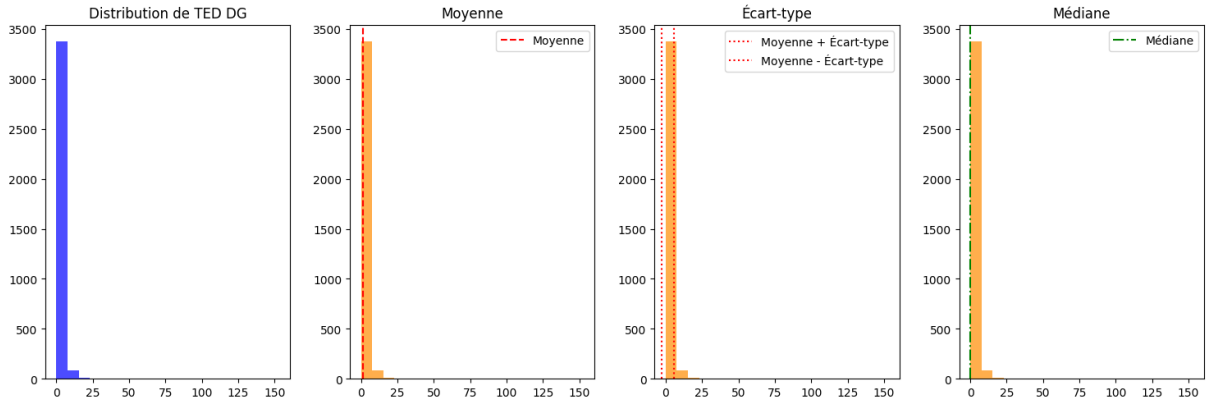
5.1. Distribution des tailles des instances

Pour cette partie on s'intéresse à la distribution des tailles des instances corrects ainsi que les witness.

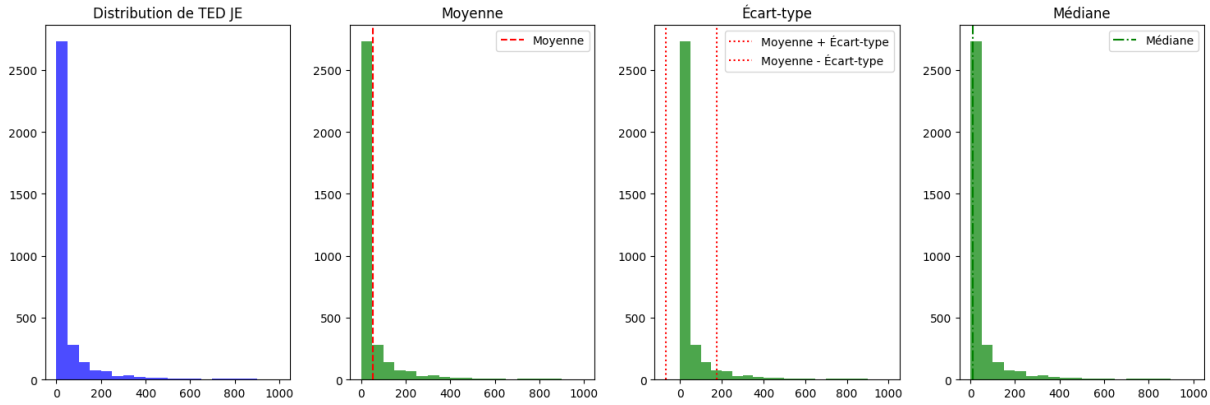


5.2. Distribution des distances d'édition

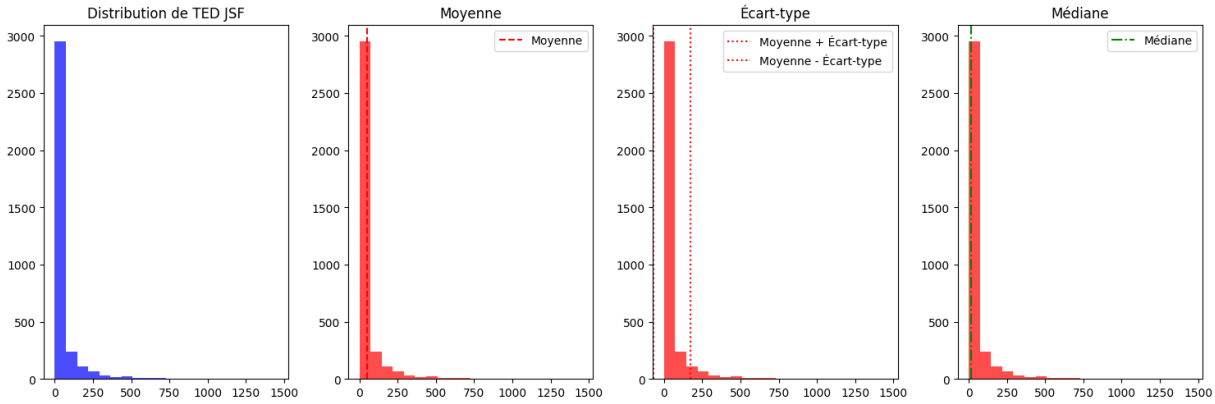
On s'intéresse dans cette partie à savoir connaitre certaines statistiques concernant les distances d'édition pour chacun des générateurs par rapport à la taille des instances.



(a) Distrubtion TED DG



(b) Distrubtion TED JE



(c) Distrubtion TED JSF

FIGURE 2.1 – Distribution TED par rapport à la des tailles des instances

5.3. Relation entre distance d'édition et nombre d'erreurs

On s'intéresse dans cette partie à savoir connaitre certaines statistiques concernant les distances d'édition pour chacun des générateurs

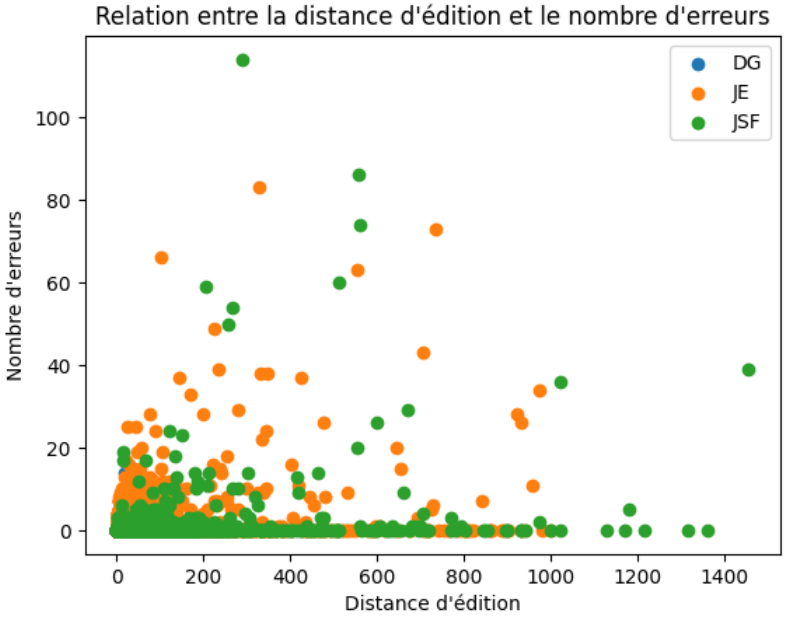
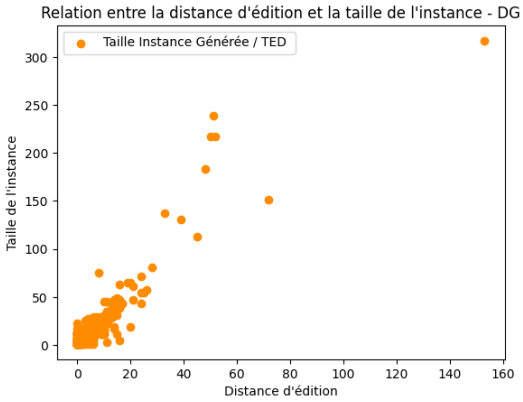
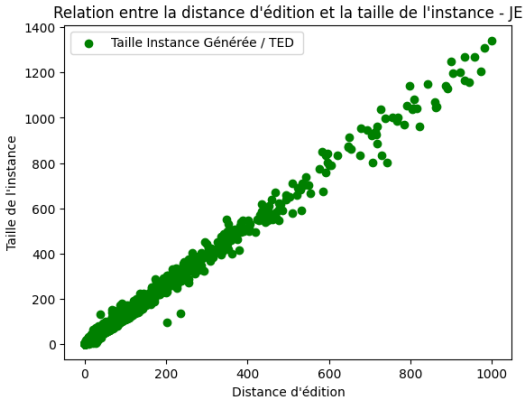


FIGURE 2.2 – Nombre d'erreurs et TED

On s'intéresse aussi à la relation entre la taille des witness par rapport à leurs TED

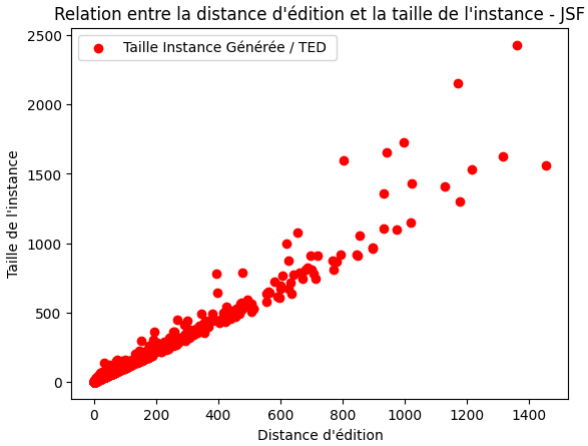


(a) Taille d'instances et TED (DG)



(b) Taille d'instances et TED (JE)

FIGURE 2.3 – Distribution des TED



(c) Taille d'instances et TED (JSF)

FIGURE 2.3 – Distribution des TED

Observations

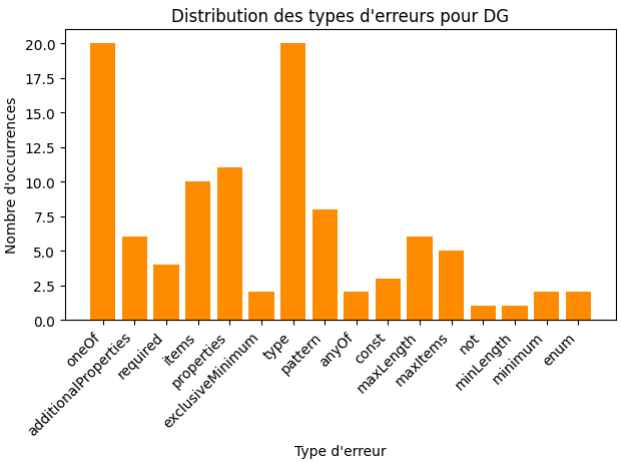
D'après notre étude expérimentale, nous constatons qu'il n'y a pas de lien entre le nombre d'erreurs et la distance d'édition. Nous remarquons également que parfois, la distance d'édition peut être considérable sans que le nombre d'erreurs ne soit significatif. Ce résultat est attendu, car les générateurs produisent des instances conformes au schéma mais qui ne sont pas des répliques exactes de l'instance correcte. En revanche, nous pouvons identifier une corrélation entre la taille des instances et la distance d'édition.

Conclusion

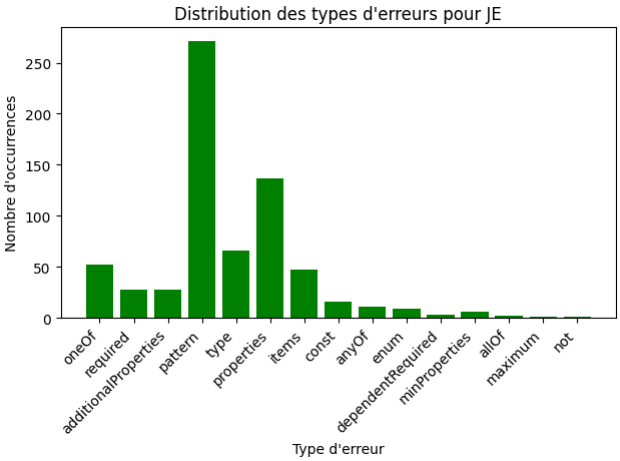
À ce stade, la distance d'édition ne nous fournit que peu d'informations utiles pour réparer les instances. La principale raison en est que JEDI ne fournit qu'une **mesure quantitative** du nombre d'opérations nécessaires, sans nous donner d'informations sur la nature spécifique de ces opérations.

6. Types d'erreurs des instances

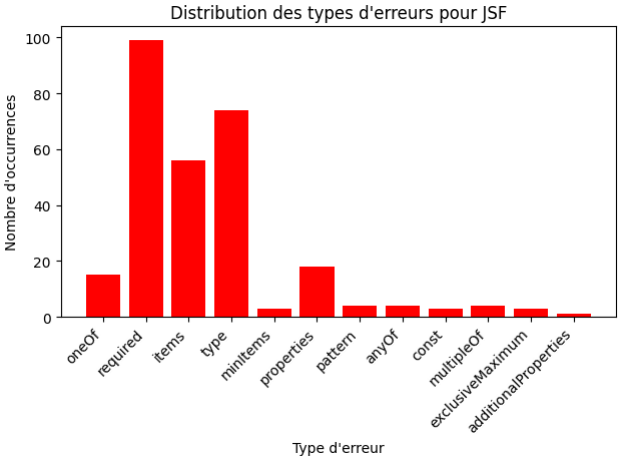
Dans cette section, nous examinons les types d'erreurs courantes présentes dans l'ensemble de données, ce qui nous permet de mieux définir et choisir l'approche à adopter pour la réparation des instances JSON.



(a) DG



(b) JE



(c) JSF

FIGURE 2.4 – Distribution des types d'erreurs

Le tableau ci-dessous présente un résumé des statistiques obtenues au cours de notre expérimentation.

Générateurs	Erreurs	Taux d'occurrence
DG	type	19.41 %
	oneOf	19.41 %
	properties	10.57 %
	maxLength	5.82 %
	...	44.79 %
JE	pattern	39.97 %
	properties	20.20 %
	type	9.73 %
	required	4.35 %
	...	33.41 %
JSF	required	34.85 %
	type	26.05 %
	items	19.71 %
	properties	6.33 %
	...	13.06 %

TABLE 2.2 – Taux de Validité des instances générées

Observations

Il est notable que les erreurs les plus fréquentes sont souvent des erreurs fondamentales telles que les erreurs de type, les champs requis, les valeurs minimales et maximales. Cependant, du fait de la structure arborescente du schéma JSON, les erreurs sur les feuilles ou les nœuds les plus profonds de l'arbre peuvent avoir des répercussions sur le parent.Par exemple, les erreurs de type AnyOf, OneOf ou Not sont généralement causées par des sous-schémas invalides. Deux types d'erreurs se distinguent dans notre étude : les erreurs simples à corriger, telles que les erreurs de type et les erreurs "required", et les erreurs liées aux sous-schémas, classées en deux catégories distinctes :

- **Erreurs de schéma de forme disjonctive** : Ces erreurs peuvent être corrigées facilement en rectifiant l'erreur dans le sous-schéma correspondant.
- **Erreurs de schéma de forme conjonctive** : La correction de ces erreurs implique davantage de contraintes, ce qui rend ce type de schéma moins réparable, du moins sans adopter d'autres approches.

Conclusion

La structure arborescente du schéma JSON signifie que la réparation d'une instance ne se limite pas à des corrections locales, mais plutôt à une approche ascendante où à chaque

validation, nous vérifions si la conformité de l'instance par rapport au schéma a été respectée. En outre, nous avons constaté que la distance d'édition n'est pas particulièrement utile en tant qu'approche de réparation, en raison de sa nature quantitative.

CHAPITRE 3

APPROCHE DE RÉPARATION

1. Algorithme de réparation

Comme indiqué dans le chapitre précédent, nous avons conclu que la distance d'édition ne nous permet pas de tirer des conclusions particulièrement utiles pour la réparation, et que les schémas de forme disjonctive sont réparables, contrairement aux schémas conjonctifs pour lesquels des solutions spécifiques doivent être trouvées. Par conséquent, la réparation sera guidée par les erreurs. Pour cela, nous décomposons l'algorithme de réparation en :

- Réparation des **type assertions** : Type, Required, Minimum, ...
- Réparation des **expressions booléenne** : Il s'agit d'une forme de schéma un peu complexe, et on trouve :
  - Forme disjonctive : anyOf
  - Forme conjonctive : allOf, not$\cdots$.

Ci-après, nous focalisons exclusivement sur la forme disjonctive ainsi que sur les divers types d'assertions, étant donné que la forme conjonctive présente des difficultés de manipulation. Bien que le prototype que nous avons implémenté soit correct, il demeure incomplet, car nous ne traitons que certaines des erreurs que nous jugeons pertinentes : *type, minimum, multiple, required, anyOf, oneOf, noAdditionnalProperties*

1.1. Exemple de réparation

Considérons le schéma ci-dessous qui définit une personne. Ce schéma spécifie qu'un nom est obligatoire, pouvant être soit une chaîne de caractères, soit un objet contenant la propriété "first", qui à son tour contient une propriété appelée "surname".

```json
{
  "type":"object",
  "required": [
    "person"
  ],
  "properties": {
    "person": {
      "required": [
        "name"
      ],
      "properties": {
        "name": {
          "anyOf": [
            {
              "type": "object",
              "required": [
                "first"
              ],
              "properties": {
                "first": {
                  "type": "object",
                  "properties": {
                    "surname": {
                      "type": "string"
                    }
                  },
                  "required": [
                    "surname"
                  ],
                  "minProperties": 2
                }
              }
            },
            {
              "type": "string"
            }
          ]
        }
      }
    }
  },
  "$schema": "https://json-schema.org/draft/2019-09/schema"
}
```

(a) Schema Json

```
{
    "person":{
        "name":{
            "first":{
                "surname":12
            }
        }
    }
}

```

(b) Instance non valide

Lors de la validation de l'instance par rapport au schéma nous obtenons les erreurs suivante :

```
{"valid": false,
  "instanceLocation": "",
  "keywordLocation": "",
  "errors": [
    {
      "instanceLocation": "/person/name",
      "keywordLocation": "/../name/anyOf",
      "errors": [
        {
          "instanceLocation": "/person/name/first",
          "keywordLocation": "/../name/anyOf/0/properties/first",
          "errors": [
            {
              "instanceLocation": "/person/name/first/surname",
              "keywordLocation": "/../name/anyOf/0/../surname/type",
              "error": "The instance must be of type \"string\""
            },
            {
              "instanceLocation": "/person/name/first",
              "keywordLocation": "/../name/anyOf/0/../first/minProperties",
              "error": "The object has too few properties (minimum 2)"
            }
          ]
        },
        {
          "instanceLocation": "/person/name",
          "keywordLocation": "/../name/anyOf/1/type",
          "error": "The instance must be of type \"string\""
```

```
        }
      ]
    }
  ]
}

```

Dans l'exemple précédent, nous avons identifié quatre erreurs : : *anyOf, type (*deux fois [1]*),* *minProperties*. Dans ce cas, la réparation implique de sélectionner une branche du "anyOf", puis de corriger les erreurs du sous-schéma. Par exemple, une solution de réparation pourrait consister à sélectionner le premier sous-schéma de anyOf, corriger l'erreur de type pour l'attribut "surname" et ajouter l'attribut "end" requis.

```
{
    "person":{
        "name":{
            "first":{
                "surname":"Yanis",
                "end":null
            }
        }
    }
}
```

(c) Instance valide

1.2. Contraintes de réparations

Comme mentionné à plusieurs reprises, la réparation est un processus assez complexe et certaines instances ne peuvent pas être réparées. Prenons par exemple le cas suivant, où l'instance est valide par rapport aux deux sous-schémas du **oneOf**, mais où il est difficile de générer une valeur car les deux sous-schémas indiquent que l'élément est un objet dont aucune propriété n'est requise. Dans ce cas, le générateur ne produit qu'une instance vide.

1. une à cause de la première branche du anyOf dans l'objet, et la seconde de la deuxième branche du anyOf

```json
{
  "type": "object",
  "oneOf": [
    {
      "additionalProperties": false,
      "properties": {
        "key": {
          "maxLength": 12,
          "pattern": "^[a-zA-Z0-9_\\.]+$",
          "type": "string"
        }
      }
    },
    {
      "additionalProperties": false,
      "properties": {
        "keys": {
          "items": {
            "maxLength": 12,
            "pattern": "^[a-zA-Z0-9_\\.]+$",
            "type": "string"
          },
          "type": "array"
        }
      }
    }
  ]

}
```

(a) Schema Json

```
{ }
```

(b) Instance générée

1.3. Réparation des types assertions

Ces erreurs surviennent généralement dans des sous-schémas ou des feuilles mais sont répercutées sur les parents. Pour simplifier, supposons que le sous-schéma concerné a une forme disjonctive.

Algorithm 1 Réparation type assertions

```
procedure RepairTypeAssertionError(error, schema, instance)
    if error.type is TypeError then
        instance[error.instancePath] ← GenerateType(schema[error.path])
    end if
    if error.type is RequiredError then
        missingProperty ← error.key
        subSchema ← schema[error.path]
        subInstance ← instance[error.instancePath]
        instance[error.instancePath] ←
                GenerateValue(schema.properties[missingProperty])
    end if
    if error.type is MinimumOfError then
        instance[error.instanceError] ← GenerateMinValue(schema[error.path])
    end if
     ....
    if error.type is NoAdditionnalPropertiesError then
        delete instance[error.instanceError]
    end if
    if error.type is MaximumProperties then
        numMaxProperties ← error.maxProperties
        numProperties ← error.properties.length
        notRequiredProps ← ExtractNotRequiredProp(error.properties, error.required)
        /*remove only min unrequired properties*/
        minPropsRemove ← Min(notRequiredProps.length, (numProperties −
numMaxProperties))
        for i in minPropsRemove do
            delete instance[notRequiredProps[i].path]
        end for
    end if
end procedure
```

1.4. Réparation des expressions booléennes

Les erreurs de anyOf, items ... sont causés par un sous schémas non valide, par exemple pour corriger une instance non valide à un anyOf, il suffit de corriger une de branche qui n'est pas valide.

— **anyOf** : Nous sélectionnons la branche la plus appropriée [2] du 'anyOf' à réparer en priorité, puis nous appelons de manière récursive le sous-schema concerné.

— **oneOf** : La réparation d'un oneOf est possible lorsque l'instance n'est valide selon aucun des schémas. Dans ce cas précis, la démarche à suivre est similaire à celle d'un 'anyOf'. Toutefois, il est important de noter que même après cette démarche,

2. dans notre implémentation nous avons choisi le premier sous schéma pour simplifier le prototype

le résultat peut ne pas être valide

— **not/allOf** : Le not ne peut pas être réparé car c'est une forme conjonctive.

Algorithm 2 Réparation des expressions booléennes

```
procedure RepairBooleanExpressionsError(error, schema, instance)
    if error.type is AnyOfError then
        pathAnyOf ← IdentifyBranchPath(schema, instance)
        subSchema ← schema[pathAnyOf]
        subInstance ← instance[error.instancePath]
        subError ← Validate(subSchema, subInstance)
        instance[error.instancePath] ← RepairInstance(subError, subSchema, subInstance)
    end if
    if error.type is OneOfError then
        if instance valid on multiple schema then
            return Can not
        end if
        /*Case where schema is not valid at any branch of oneOf*/
        pathOneOf ← IdentifyBranchPath(schema, instance)
        subSchema ← schema[pathOneOf]
        subInstance ← instance[error.instancePath]
        subError ← Validate(subSchema, subInstance)
        instance[error.instancePath] ← RepairInstance(subError, subSchema, subInstance)
    end if
end procedure
```

1.5. Algorithme de réparation d'une erreur

L'algorithme de répration d'une erreur permet donc de gérer l'erreur de validation

Algorithm 3 Réparation d'une erreur générique

```
procedure RepairInstance(error, schema, instance)
    if error.type is typeAssertion then
        RepairTypeAssertionError(error, schema, instance)
    else
        RepairBooleanExpressionsError(error, schema, instance)
    end if
end procedure
```

1.6. Algorithme de réparation

L'algorithme de répration répare une erreur, puis revalide les nouvelles données

Algorithm 4 Réparation de erreur de schéma

```
procedure REPAIR(schema, instance)
    errors ← VALIDATE(schema, instance)
    while errors.length !=0 do
        error ← errors[0]
        if error is Repairable then
            res ← REPAIRINSTANCE(error, schema, instance)
            /*if we return can not then remove the error*/
            if res is not null then
                errors.Pop()
            else
                errors ← VALIDATE(schema, instance)
            end if
        else
            errors.Pop()
        end if
    end while
end procedure
```

2. Expérimentation

Nous avons implémenté un algorithme minimaliste en TypeScript pour deux raisons :

- Le générateur JSF qui enregistre le taux de réussite le plus élevé lors de la génération est développé en TypeScript.
- Notre familarité avec le langage de programmation.

2.1. Dataset

Le dataset comprend neuf schémas, dont cinq que nous avons définis manuellement avec leurs instances non valides, et quatre schémas provenant du dataset utilisé pour l'expérimentation avec JEDI.

2.2. Type d'erreurs à traiter

Les types d'erreurs qu'on traite sont : *type, minimum, multiple, required, anyOf, oneOf, noAdditionnalProperties*

2.3. Validateur et générateur

Nous avons utilisé la librarie de validation **Json Schema library** [**14**], ainsi que le générateur **JSF** [**8**]

2.4. Résultat

L'algorithme a réussi à corriger **huit instance** sur les **neufs instance invalides**. La seule instance que nous n'avons pas pu corriger est un schéma de type OneOf, où le générateur génère une instance correcte par rapport aux sous-schémas.

3. Conclusion et perspectives

Nous avons exploré l'utilisation d'un langage très puissant, la bibliothèque JSON Schema, pour la réparation des instances. Malgré les contraintes de temps, nous avons pu implémenter une partie de l'algorithme. Cependant, notre approche demeure fiable. Nous avons notamment supposé que les schémas étaient sous une forme disjonctive. Pour les schémas conjonctifs, nous ne pouvons pas nous limiter à notre solution actuelle. Nous envisageons donc de transformer les schémas conjonctifs en une forme intermédiaire équivalente disjonctive. Cependant, cela pose toujours un défi en ce qui concerne l'opérateur de négation 'not', pour celà on propose de choisir des méthodes heuristiques pour choisir quelle méthode est la mieux adapté pour réparer ce type de schéma.

BIBLIOGRAPHIE

[1] Thomas Hütter, Nikolaus Augsten, Christoph Kirsch, Michael Carey, and Chen Li. Jedi : These aren't the json documents you're looking for ? pages 1584–1597, June 2022.

[2] Json schema. `https://json-schema.org`.

[3] Benchmarking de solutions optimistes pour génération de données test à partir de json schema. Sorbonne Universite - Faculté de Science et ingénierie Master Informatique parcours STL, 2023.

[4] Felipe Pezoa, Juan Reutter, Fernando Suarez, Martin Ugarte, and Domagoj Vrgoč. Foundations of json schema. pages 263–273, April 2016.

[5] Lyes Attouche, Mohamed-Amine Baazizi, Dario Colazzo, Giorgio Ghelli, Carlo Sartiani, and Stefanie Scherzinger. Validation of modern json schema : Formalization and complexity. *Proceedings of the ACM on Programming Languages*, 8 :1451–1481, January 2024.

[6] Json similarity comparitor. `https://github.com/Geo3ngel/JSON-Similarity-comparitor`.

[7] Mohamed Amine Baazizi et al. Not elimination and witness generation for json schema. bda, 2020.

[8] json-schema-faker. `https://github.com/json-schema-faker/jsonschema-faker`, 2023.

[9] json-everything. `https://github.com/gregsdennis/json-everything`, 2023.

[10] json-data-generator. `https://github.com/jimblackler/jsongenerator`.

[11] Jschon validator. `https://github.com/marksparkza/jschon`, 2023.

[12] Json schema. `https://github.com/DatabaseGroup/tree-similarity/pull/31`.

[13] json schema migration. `https://github.com/ajv-validator/json-schema-migrate`, 2023.

[14] Json schema library. `https://github.com/sagold/json-schema-library`.