Rapport de projet
Outils décisionnels

Médéric HURIER <mederic.hurier@etu.univ-lorraine.fr>

Valentin LACAVE <valentin.lacave@etu.univ-lorraine.fr>

Mouchtali SOILIHI <mouchtali.soilihi@etu.univ-lorraine.fr>

Geoffrey GULDNER <geoffrey.guldner@etu.univ-lorraine.fr>

2013-2014

Sommaire

1. Introduction

1.1 Résumé du problème

La sécurité absolue n'existe pas, surtout lorsqu'il s'agit de sécurité informatique où les moyens technologiques évoluent rapidement dans des infrastructures de plus en plus complexes. Dans cet environnement changeant, le responsable de la sécurité du système d'information (RSSI) se doit d'agir de plus en plus rapidement pour prevenir au mieux les risques, et plus rapidement encore lorsqu'il s'agit de réagir aux accidents. S'ajoute à cela les contraintes budgétaires qui lui imposent des compromis pas toujours bien mesurés. Dans cet article, nous proposons de modéliser la sécurité d'une infrastructure comme un graphe orienté pondéré. Les noeuds sont les états de sécurité de l'infrastructure et les arcs représentent les actions possibles pour l'attaquant. Certaines feuilles du graphe sont des états de vulnérabilité que l'on doit protéger. Le chemin le plus court correspond au risque majeur. Nous proposons un algorithme génétique pour affecter aux arcs des mesures de sécurité qui vont rallonger le chemin le plus court tout en recherchant à s'approcher au mieux d'un budget cible.

1.2 Contenu du rapport

Ce document est le compte-rendu de nos travaux de recherche sur l'usage **des outils décisionnels appliqués à la sécurité de l'information**. Nous avons orienté cette étude en suivant les recommandations de notre tuteur. Grâce à **un algorithme génétique et la décomposition par niveau,** nous pouvons effectuer **une optimisation multicritère** pour identifier la meilleure façon d'affecter des mesures de sécurité.

Ce rapport est organisé de la façon suivante:

- **Les structures** sont expliquées dans la partie 2. Nous avons séparé la modélisation haut-niveau dans la partie 2.1 et l'implémentation technique dans les la partie 2.2 et la partie 2.3.
- **Les traitements** associés aux structures sont dans la partie 3. Cette partie comprend la décomposition par niveau (partie 3.1), la recherche du plus court chemin (partie 3.2) et la base de l'algorithme génétique (partie 3.3).
- **L'algorithme génétique** est développé en détails dans la partie 4.
- **La conclusion** de la partie 5 reprend les résultats de l'étude sous forme graphique (partie 5.1), et notre bilan du projet (partie 5.2).

Vous trouverez également en annexe externe les éléments suivants:

- **le code source du programme C/GCC** qui permet de compiler et de lancer le solveur.
- **le code source du prototypez Python** qui nous a servi à tester les algorithmes et à générer les graphiques pour étudier les paramètres de l'algorithme génétique.

2. Structures

2.1 Jeu de données

Comme proposé dans le résumé du problème, nous devons créer un graphe connexe, orienté et pondéré pour modéliser l'infrastructure d'un système d'information. **Les noeuds** représentent les actifs ciblés (pare-feu, commutateur, service …) et **les arcs** des vecteurs d'attaque qu'un utilisateur malveillant peut emprunter pour compromettre le système. La **pondération** correspond à la difficulté de l'attaquant à traverser l'arc, ou pour la défense, l'efficacité des mesures de sécurité.

En annexe du graphe, nous avons également des **mesures de sécurité** qui peuvent s'appliquer aux arcs pour augmenter la pondération. Cette pondération dépend de l'arc auquel la mesure s'applique, mais **le coût de déploiement de la mesure est fixe**.

Certains points du modèle étant libre, nous avons fais plusieurs **choix d'implémentation**:

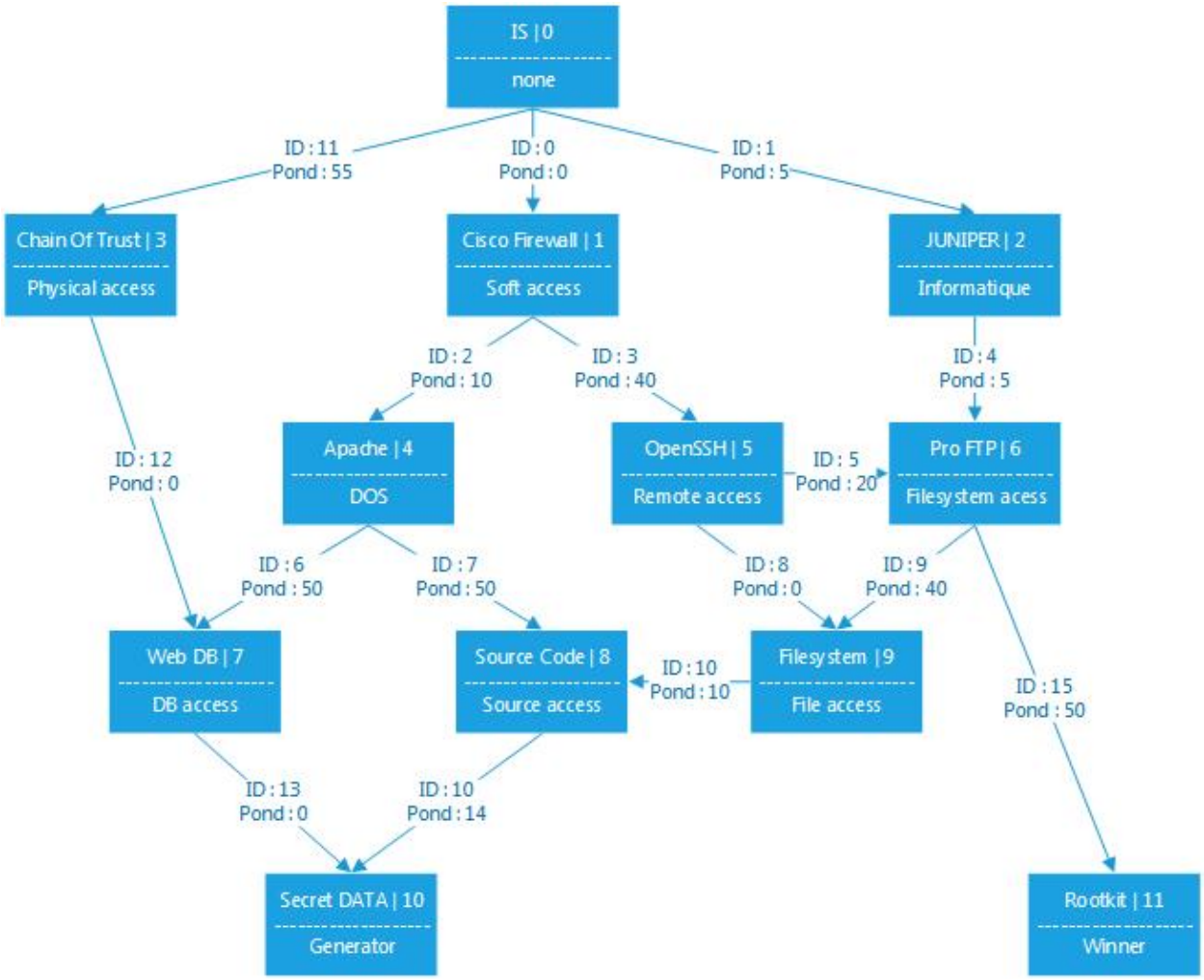
- **la matrice** est représenté par des tableaux dynamiques et des listes plutôt que par une matrice. Ce choix nous paraissait le plus souple et le plus évolutif.
- **les états de vulnérabilités** sont placés à n'importe quel niveau du graphe et correspondent aux noeuds feuilles. De cette façon, un RSSI peut facilement représenter son infrastructure, sans se soucier de la place de ses actifs dans le graphe.
- **la pondération et la longueur des chemins** s'expriment en % et correspondent à l'efficacité des protections (0% = absence de mesure, 100% = sécurité presque parfaite).
- **le budget** s'exprime en k€, mais nos valeurs que nous avons choisi sont imaginaires.

Le diagramme qui va suivre correspond au **jeu de données** que nous avons retenu pour cette étude. Nous avons essayé d'être le plus concret possible, en donnant des noms et des valeurs pertinents aux noeuds, aux arcs et aux mesures.

Les tests que nous avons fait se basent directement sur ce jeu de données, de même que **les graphiques et les résultats** que nous donnons en conclusion.

Compte-tenu du contexte du projet, nous n'avons pas réalisé **d'interface graphique** pour composer manuellement le jeu de données. Un utilisateur doit créer un programme principale et inclure les bibliothèques correspondantes pour modéliser une nouvelle infrastructure (dans la version en c, il est possible de rentrer le graphe et les mesures à la main).

Néanmoins, nous pensons que l'ajout cette fonctionnalité peut se faire **facilement** car toutes nos fonctions sont couvertes par **des tests unitaires et documentées**.



Le diagramme se lit de la manière suivante:

- **les noeuds** sont matérialisés par des carrés bleus. Sur la première ligne, on trouve le **nom de l'actif** et son **ID**. Sur la seconde ligne, c'est le privilège de l'attaquant sur l'actif.
- **les arcs** sont des flèches bleues qui partent d'un noeud d'origine et pointe vers le noeud de destination. Elles portent 2 informations: **l'ID** unique de l'arc et la **pondération initiale**.
- sur ce graphe, **le noeud racine** a l'ID 0 (IS) et **les noeuds feuilles** (états vulnérables) ont pour ID 10 (Secret DATA) et 11 (Rootkit).

*Pour ne pas surcharger le graphe, nous avons regroupé **les informations des arcs** dans le tableau suivant.*

N°	Nom	Description
0	0-day	corrige une faille de type 0 day
1	Scan port	prévient le scan des ports ouverts sur la machine
2	Dos	tentative de déni de service (DoS)
3	Brute force	tentative de récupération de MDP par force brute
4	Heap overflow	attaque par dépassement de pile
5	Lack of privilege	les droits de l'attaquant sont trop permissifs
6	SQLi	injection de code SQL par l'application
7	Wget	extraction des sources via la commande wget
8	scp	extraction des sources via la commande scp
9	DNS Dump	extraction des sources via le protocole DNS
10	DNS Dump	extraction des sources via le protocole DNS
11	Physical breach	tentative d'intrusion physique (batîment)
12	Hard Drive Robery	vol du disque dur
13	GG	l'attaquant a accès à une donnée secrète !!!
14	GG	l'attaquant à accès à une donnée secrète !!!

*De même que pour les arcs, nous avons regroupé **les informations des mesures** dans le tableau suivant.*

N°	Nom	Coût	Pondérations
1	Patch Cisco FW	+30	Pour l'arc 0: +50
2	PSAD	+10	Pour l'arc 0: +20 Pour l'arc 1: +25
3	Fail2Ban	+20	Pour l'arc 2: +15 Pour l'arc 3: +40
4	Patch PROFTP	+5	Pour l'arc 4: +60 Pour l'arc 5: +25
5	SQL Firewall	+40	Pour l'arc 6: +50
6	MODSec	+25	Pour l'arc 2: +30 Pour l'arc 7: +30 Pour l'arc 6: +30
7	IPS	+15	Pour l'arc 8: +20 Pour l'arc 9: +20 Pour l'arc 10: +20
8	Encryption	+35	Pour l'arc 12: +70
9	Ultimate Def	+200	Pour l'arc 0: +100 Pour l'arc 11: +100
10	Chmod	+40	Pour l'arc 13: +40

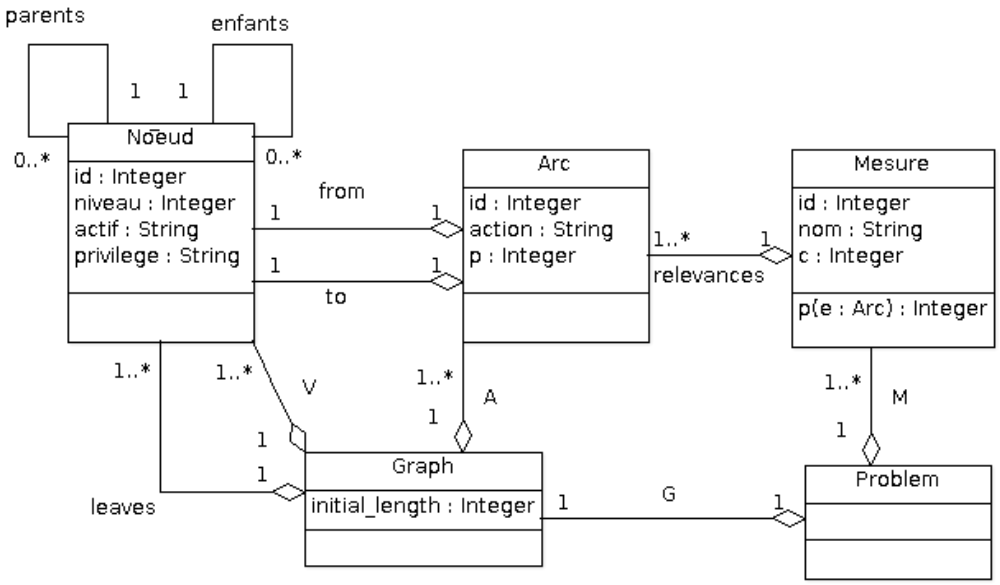
Les informations sont de deux types:

- **un coût fixe** qui correspond à l'argent nécessaire pour déployer la mesure.
    - nous pensons que le coût est le même quelque soit l'actif. En effet, les procédures d'installation et de configurations sont généralement semblables,.
    - quand bien même, les diférences en terme de coût seraient minimes
- **la pondération supplémentaire** qui dépend de l'arc.
    - selon le type et le niveau de sécurité de l'actif, l'effet de la mesure ne sera pas aussi efficace.
    - dans le cas de la mesure 1 par exemple, la protection des 2 pare-feux qu'offre la solution PSAD (bloque les tentatives de scan) dépend du nombre de port ouvert sur chaque pare-feux et des solutions constructeurs déjà en place.

2.2 Structures liées aux graphes

Un graphe est **une structure standard** en programmation informatique, mais dans le cadre de notre projet, nous avons dû inclure **l'orientation** et **la pondération** sur les arcs.

Le diagramme de classe ci-dessous présente la modélisation que nous avons retenu:



Un noeud est défini par un numéro d'identifiant unique (**ID**) et le **niveau** qu'il occupe dans le graphe. Pour le décrire, nous avons utilisé 2 champs textes: **actif** (bien ou service à protéger) et **privilège** (possibilités de l'attaquant sur l'actif). Un noeud peut avoir plusieurs prédécesseurs (**parents**) et successeurs (**enfants**). Les noeuds sans successeurs sont les états vulnérables.

Un arc a également un **ID** et est décrit par un champ action correspondant à une vulnérabilité. L'attribut p est une pondération (de 0 à 100) qualifiant la facilité de l'attaquant à traverser l'arc. Pour relier le noeud d'origine et de destination, l'arc a deux champs, respectivement from et to.

Une mesure est une sécurité qui peut s'appliquer à un arc selon sa pertinence (**relevances**). Elle dispose d'un attribut **ID**, d'un champ description **nom** et d'un coût fixe **c**. Pour obtenir la pondération supplémentaire, il faut utiliser la méthode **p** qui prend un arc en paramètre.

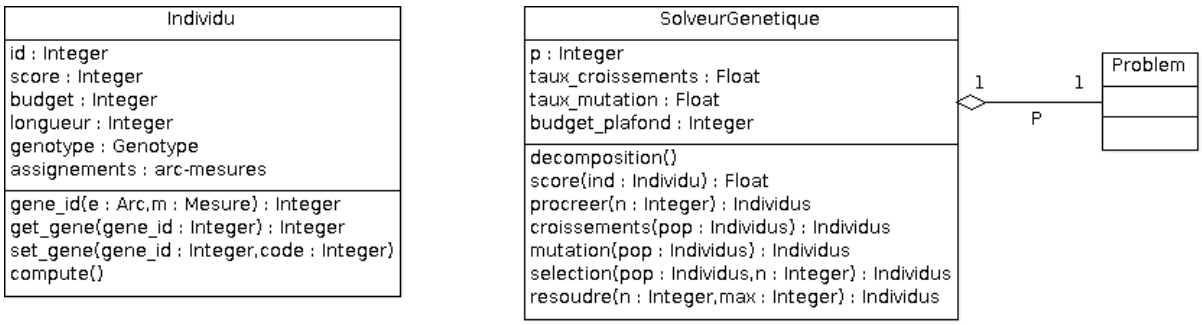
Les noeuds et les arcs sont associés à une classe **Graphe** qui les place dans des listes, respectivement **V** et **A**. Les noeuds feuilles sont également stockées dans l'association **leaves**.

Enfin, une classe **Problème** regroupe un graphe unique **G** et un ensemble de mesure **M**.

2.3 Structures liées à l'algorithme génétique

Les structures liées à l'algorithme génétique sont quant à elles **spécifiques** à ce type de problème. Après plusieurs tentatives, nous avons réussi à trouver la bonne modélisation.

Le diagramme de classe ci-dessous présente la modélisation que nous avons retenu:



Un **individu** représente une solution possible pour le problème d'optimisation. Son attribut le plus important est le **génotype**, que nous détaillons par la suite dans une partie consacrée. L'individu dispose également d'un **ID** et de plusieurs champs permettant de qualifier sa qualité:

- **le score**: pourcentage renvoyé par la fonction d'évaluation (voir partie suivante)
- **le budget**: le coût total des nouvelles mesures à déployer
- **la longueur**: la longueur du plus court chemin (PCC) vers un état de vulnérabilité

L'assignement correspond a une association entre un arc et des mesures à appliquer sur cette arc. C'est un champ utilisé en autres pour calculer le chemin le plus court (voir partie suivante).

Pour résoudre le problème, nous avons également **un solveur génétique** qui utilise ces individus pour fonctionner. Il prend plusieurs paramètres (voir partie suivante):

- **p**: pondération entre l'optimisation du PCC ou du budget (compris entre 0 et 1)
- **taux_croisements**: le taux de croisement effectuée par itération (compris entre 0 et 1)
- **taux_mutation**: le taux de mutation effectuée par itération (compris entre 0 et 1)
- **budget_plafond**: le montant disponible pour améliorer la sécurité (en k€)

Plusieurs fonctions propres à l'algorithme génétique sont égalements liées à cette classe. Pour plus de détails, nous vous invitons à consulter la partie dédiée à l'algorithme génétique.

3. Traitements

3.1 Décomposition par niveaux

L'algorithme de décomposition par niveau est utilisé dans notre programme pour:

- **vérifier que le graphe ne comporte pas de cycle**.
- **déterminer, lorsque deux chemins ont la même longueur, quel est le plus direct**.

Dans le premier cas, nous cherchons à vérifier que **le graphe est bien formé**. En effet, notre modèle ne prévoit pas qu'un attaquant puisse revenir vers un état précédent. Les actions qu'il entreprend doivent donc être modélisées **sans cycles**.

Dans le second cas, nous avons dû définir ce qu'est **la notion** "**facilité"** pour un attaquant de compromettre un système. Son objectif n'est pas réaliser son attaque avec le moins d'étape possible, mais bien de **limiter son effort global.** Ainsi, il préfera toujours prendre le **chemin le plus court**, même si celui-ci implique plus d'étape pour atteindre ses objectifs.

Néanmoins, nous avons voulu tenir compte du nombre d'étape dans le cas il existe **plusieurs plus courts chemins de même longueur**. Dans notre modèle, l'attaquant aura tendance à **privilégier le chemin le plus court, et ensuite le plus direct**.

Nous profitons également du parcours de tous les noeuds pour **identifier les noeuds feuilles**. Ces noeuds n'ont pas de successeurs. Dans notre modèle, ils correspondent aux **états cibles** que l'attaquant cherche à atteindre pour réussir sa compromission.

Le pseudo-code suivant vous présente l'algorithme tel que nous l'avons implémenté:

```
# remise à zéro des résultats précédents
L = list()            # table associative entre un niveau et une liste de noeud
leaves = list()     # tableau pour stocker les noeuds feuilles

# initialisation des variables
my_vertices = list(V)   # copie du tableau contenant les noeuds
tagged = list()             # tableau contenant les noeuds marqués
degree = 0                  # niveau du graphe (incrémenté)

# boucle de traitement principale: tant que les noeuds ne sont pas tous traités
while len(my_vertices) != 0:
        # tableau contenant les noeuds associés à ce niveau
        degree_vertices = list()

        # parcours les noeuds restants pour identifier ceux du niveau courant
        for v in my_vertices:
                # vérifie la présence de précédesseurs non marqués
                has_untagged_predecessors = False
```

```
            for u in v.parents.keys():          # parcours des prédécesseurs
                if not u in tagged:             # un prédécesseur non marqué à été trouvé !
                    has_untagged_predecessors = True
                    break

            # si il n'a pas de prédécesseurs non marqués, on ajoute le noeud au niveau
            if not has_untagged_predecessors:
                degree_vertices.append(v)
                v.degree = degree

    # marque les noeuds du niveau comme traité et marqué
     # doit être fait APRES la boucle pour ne pas fausser le test des prédecesseurs
    for v in degree_vertices:
            tagged.append(v)
            my_vertices.remove(v)

            # on profite de la boucle pour tester si il s'agit d'un noeud feuille
            if len(v.children) == 0:
                leaves.append(v)

    # on ajoute une nouvelle entrée uniquement si des noeuds sont trouvés
    if len(degree_vertices) > 0:
            L.insert(degree, degree_vertices)

    # on incrément le niveau
    degree += 1
```

3.2 Recherche du plus court chemin

Grâce à la décomposition par niveau que nous avons effectué dans la partie précédente, nous pouvions trouver le plus court chemin rapidement en calculant la distance de chaque noeuds à partir de ces prédécesseurs directs. Cependant, **cette méthode n'aurait pas été compatible avec notre modèle que nous avons retenu**.

En effet, **nos états de vulnérabilité ne sont pas tous au dernier niveau du graphe**. Cette représentation aurait imposé des contraintes supplémentaires pour le RSSI. Aussi, nous avons préféré lui laissé la liberté de placer ses états de vulnérabilité où il le souhait
Pour calculer le PCC, nous nous sommes donc basé **l'algorithme de Dijkstra**, qui est une référence pour ce type de problème. L'implémentation suit les prescriptions de la page de Wikipédia suivante: http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm#Pseudocode

Cependant, nous avons dû adapter l'algorithme pour tenir compte de l'**affectation des mesures sur les arcs**, qui a pour effet d'impacter l'effort de l'attaquant. Nous avons également tenu compte du **niveau du chemin** pour préférer les parcours les plus directs.

Le pseudo-code suivant vous présente l'algorithme tel que nous l'avons implémenté:

```
# Fonction retournant la plus petite distance d'un noeud parmi la liste des noeuds non-traités (Q)
# et la liste des distances par noeud (dist)
def smallest_dist(Q, dist):
        min = float('Inf')          # affecte la distance minimal à l'infini
        vmin = None                 # affecte le noeud dont la distance est minimal à nul
                for v in Q:         # parcours chaque noeud non traité
                        if dist[v] < min:           # un nouveau minimum est trouvé
                                min = dist[v]
                                vmin = v
                        # distance égale, le chemin est plus direct (niveau inférieur)
                        elif dist[v] == min and (vmin is not None and v.degree < vmin.degree):
                                min = dist[v]
                                vmin = v
        return vmin

# Paramètres de la fonction
G <= Graph()                # structure Graphe initialisée et décomposée
assignements <= dict()    # tableau associatif entre un arc et des mesures

# variables de la fonction
dist = dict()          # tableau associatif entre un noeud et sa distance
previous = dict()   # tableau associatif entre un noeud et son prédécesseur sur le chemin
source = G.V[0]    # noeud de départ (racine)
target = None      # noeud d'arrivé (premier état de vulnérabilité trouvé)
Q = set(G.V)         # ensemble des noeuds non traités

# initialise les variables
for v in G.V:
        dist[v] = float('Inf')
        previous[v] = None
dist[source] = 0

# boucle de traitement principale: tant que des noeuds ne sont pas traités
while len(Q) != 0:
        # récupère le noeud non traité dont la distance est la plus courte
        u = smallest_dist(Q, dist)
        # il est retiré de la liste (donc considéré comme traité)
        Q.remove(u)

        # arrête le parcours car il s'agit d'un état de vulnérabilité (noeud feuille)
        if u in G.leaves:
                target = u
                break

        for v, e in u.children.items():          # parcours des noeuds successeurs (v=noeud fils, e=arc)
                alt = dist[u] + e.p               # calcule la distance alternative en passant par le fils
                if e in assignments.keys():   # tiens compte de l'affectation des mesures sur les arcs
                        for m in assignments[e]:
                                alt += m.p(e)
        # nos distances ne peuvent pas dépasser la valeur 100
        if alt > 100:
                alt = 100
```

```
        # la distance est plus courte par ce chemin, on met à jour le chemin et la distance
        if alt < dist[v]:
                dist[v] = alt
                previous[v] = u

# après calcul, on récupère le chemin le plus court à partir de variables
u = target
shortest_path = list()
while u in previous.keys():
        shortest_path.insert(0, u)
        u = previous[u]

# on retourne la longueur du plus court chemin, et son tracé (liste de noeud)
return dist[target], shortest_path
```

3.3 Description de l'algorithme génétique

Suite à la décomposition par niveau, nous avons utilisé **l'algorithme génétique** que nous a donné notre tuteur pour réaliser le problème. **L'algorithme de base** est donné ci-dessous, et vous pouvez obtenir **plus de détails** sur les fonctions associées dans la partie suivante.

Le pseudo-code suivant vous présente l'algorithme tel que nous l'avons implémenté:

```
pop = procreate(n)        # création d'une population initiale de n individus
i = 1                     # enregistre le numéro de génération

# boucle de traitement principale
while i < max:            # la boucle continue jusqu'à atteindre un niveau maximum de génération
        pop += crosses(pop)      # ajout d'individus suite à des croissements
        pop += mutations(pop)    # ajout d'invididus suite à des mutations
        pop = selection(pop, n)  # sélection des N meilleurs individus parmi la population
        i += 1

return pop        # retour de la population calculée
```

4. L'algorithme génétique en détails

4.1 Génotype

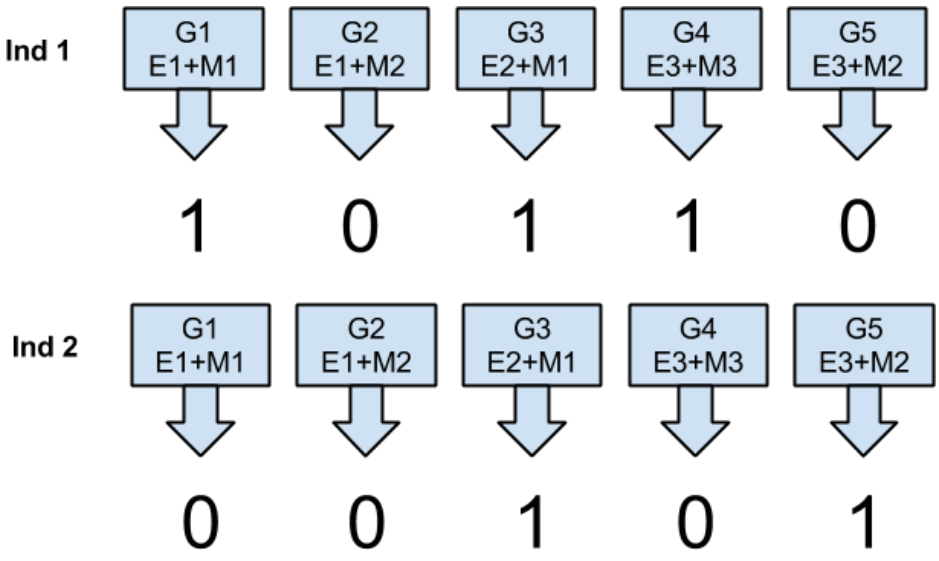
Le but d'un algorithme génétique est de **créer et d'évaluer des solutions possibles**, qui au fur des itérations, doivent être de plus en plus pertinentes et efficaces. Dans le cadre de notre problème, nous souhaitons affecter des mesures de sécurité à des actifs vulnérables de notre système. Chaque solution se caractérise donc par un ensemble d'affectation de **mesure (m)** sur des des **arcs (e)** dans le graphe.

Pour représenter ces solutions, nous avons crée une structure de type **génotype** qui code les affectations. Chaque affectation est représentée par un **gêne** marqué comme "affecté" (code 1) ou "ignoré" (code 0). **Le nombre de gêne est fixe pour chaque génotype**.

Pour que nous puissions évaluer ces solutions, nous avons associé à ce génotype différentes données liées au problème (budget total, longueur de son plus court chemin, score). Nous avons ainsi formé une nouvelle structure, que nous avons appelé un **Individu**.

La structure Individu est donnée dans la partie 2.3 de ce document.

Le schéma suivant permet d'expliquer la représentation d'un génotype:



Comme vous pouvez le voir sur cette figure, **les gênes sont identiques pour chaque individu** (g1 = e1 + m1, g2 = e1 + m2 …), seul **les codes des affectations sont différents**.

La stabilité des numéros de gêne est très importante pour pouvoir **comparer et mélanger** les génotypes (voir parties suivantes).

4.2 Procréation

L'opération de procréation permet de créer une population initiale selon un paramètre n.

Le pseudo-code suivant vous présente l'algorithme tel que nous l'avons implémenté:

```
# Paramètres de la fonction
n <= int                        # nombre d'individus souhaités

# Variables de la fonction
pop = list()                    # liste servant à convenir les individus

# fonction principale de traitement
for i in range(n):
        ind = Individual()              # création d'un nouvel individu

        # création d'un génotype original (aléatoire)
        for gene_id in genes():
                ind.set_gene(gene_id, random.choice([False,True]))      # choix du code (True= 1, False=0)

        ind.compute()           # calcule les données associées à l'individu
        pop.append(ind)                 # ajoute l'individu à la population

return pop                      # retourne la population créée
```

4.3 Évaluation

Pour qu'une solution soit retenue, elle doit être évaluée selon 2 critères:

- **la longueur du plus court chemin**: permet d'apprécier l'amélioration vis à vis de l'état de sécurité actuel du système d'information. Dans notre modèle, le niveau de sécurité globale correspond à la **longueur du PCC** vers un état vulnérable.
- **le budget total**: c'est le montant de l'amélioration, qui correspond à la somme des coûts des différentes mesures affectées.

Ses critères sont relativisés selon 3 paramètres du programme:

- **la longueur initiale du plus court chemin (iPCC)**: cette longueur correspond au niveau de sécurité initial du système. Les autres longueurs de PCC seront forcément supérieures ou égales à cette valeur.
- **le budget plafond**: les ressources d'une organisation étant limitées, le RSSI se voit imposer un budget à ne pas dépasser (même si l'étude n'est pas exclue).
- **la pondération amélioration/budget (p)**: correspond à l'importance que souhaite donner le RSSI à l'un ou l'autre des facteurs. Cette valeur est exprimée entre 0 et 1.

L’objectif du programme est de maximiser le PCC tout en minimisant le budget total.

Cependant, **nous ne pouvons pas comparer ces valeurs directement.** En effet, les dimensions de ces facteurs (moyenne et écart-type) sont très différentes par nature:

- les longueurs des chemins et des arcs sont comprises entre 0 et 100. Ces valeurs correspondent à la couverture apportée par les mesures de sécurité. Une longueur de 0 correspond à une absence totale de sécurité, et une longueur de 100 à une sécurité absolue (hypothétique).
- le budget quant à lui s’exprime en milliers d’euro (k€).

Pour que nous puissions utiliser ces mesures, nous devons donc les **normaliser**. C’est le but de **notre fonction d’évalutation**, qui renvoit **un score comparable et reproductible** pour chaque individu.

Initialement, nous avons implémenté le calcul tel qu’il était fourni dans le descriptif:

score = p * PCC / iPCC + (1-p) * budget plafond / budget total

Cette fonction s’est révélée inapaptée compte-tenu des unités de mesures que nous avons choisi. En effet, le score de la solution initiale (aucune affectation) était toujours supérieur aux autres. Prenons un exemple avec p = 0.9, iPCC = 50 et budget plafond = 300:

- score initiale = 0.9 * 50/50 + 0.1 * 300 / 1 = 30.9
- score intermédiaire 1 = 0.9 * 85/50 + 0.1 * 300 / 150 = 1.73
- score intermédiaire 2 = 0.9 * 95/50 + 0.1 * 300 / 50 = 2.31
- score intermédiaire 3 = 0.9 * 100/50 + 0.1 * 300 / 450 = 1.86

Pour palier à ce problème, nous avons créer notre propre formule de calcul qui ramène les valeurs du facteur budget entre 0 et 100 et fais un calcul de moyenne avec la longueur:

score = p * PCC + (1-p) * (100 - budget total / budget plafond *100)

Si l’on recalcule les données de l’exemple précédent avec cette formule:

- score initiale = 0.9 * 50 + 0.1 * (100 - 0 / 300 * 100) = 55
- score intermédiaire 1 = 0.9 * 85 + 0.1 * (100 - 150 / 300 * 100) = 86.5
- score intermédiaire 2 = 0.9 * 95 + 0.1 * (100 - 50 / 300 * 100) = 95.5
- score intermédiaire 3 = 0.9 * 100 + 0.1 * (100 - 450 / 300 * 100) = 90

On voit que les résultats s’expriment maintenant en **pourcentage** et qu’ils sont donc **plus facilement comparable** entre eux. De plus, **les dépassements de budgets** (budget total > budget plafond) ont maintenant un impact négatif sur le score total. Dans l’exemple précédent, un RSSI préférera prendre la solution 2 à la solution 3, qui bien que cette dernière soit plus efficace, entraîne un dépassement de budget.

4.4 Croisements

Le croisement est une opération de l'algorithme génétique qui permet de créer 2 nouveaux individus (enfants) à partir de 2 individus sélectionnés (parents) dans la population initiale.

Pour que les croisements puissent s'effectuer en nombre assez important, nous avons ajouté un paramètre "**crosses_rate**" qui correspond au taux de croisement dans une population. Une population de 100 individus avec un taux de croisement de 0.25 passera à 125 individus après l'opération.

Le pseudo-code suivant vous présente l'algorithme tel que nous l'avons implémenté:

```
# Paramètres de la fonction
pop <= list()                    # population courante
crosses_rate <= float            # taux de croisement (entre 0 et 1)

# Variables de la fonction
new_pop = list()                                    # liste stockant les nouveaux individus (enfants)
n = len(pop) * crosses_rate/2                       # calcul le nombre de familles (2 parents et 2 enfants)

for i in range(n):                                  # pour chaque famille
        father, mother = random.sample(pop, 2)      # sélection de 2 individus dans la population initiale

        # création de 2 nouveaux enfants
        son = Individual()
        daugther = Individual()

        for i, gene_id  in enumerate(random.shuffle(genes())):          # parcours tous les gênes au
hasard
                # selon le nombre d'itération, les codes sont affectés depuis le père ou la mère
                if i%2 == 0:
                        # gène fils <= gène père et gène fille <= gène mère
                        son.set_gene(gene_id, father.get_gene(gene_id))
                        daughter.set_gene(gene_id, mother.get_gene(gene_id))
                else:
                        # gène fils <= gène mère et gène fille <= gène père
                        son.set_gene(gene_id, mother.get_gene(gene_id))
                        daughter.set_gene(gene_id, father.get_gene(gene_id))

                # calcul les données des nouveaux individus et ajoute les individus à la population
                son.compute()
                daugther.compute()
                new_pop.append(son)
                new_pop.append(daughter)

return new_pop          # retourne les nouveaux individus
```

4.5 Mutations

Le croisement est une opération de l'algorithme génétique qui permet de modifier aléatoirement le génotype d'un individu. Le processus est répété plusieurs fois sur plusieurs individus de la population.

Pour réguler ces changements, nous fixons le paramètre "**mutation_rate**" qui correspond au taux de mutation dans la population et sur le génotype des individus. Une population de 100 individus avec un taux de mutation de 0.25 passera à 125 individus après opération, et pour chacun des 25 nouveaux individus, 25 % de leurs gênes seront modifiés.

Le pseudo-code suivant vous présente l'algorithme tel que nous l'avons implémenté:

```
# Paramètres de la fonction
pop <= list()                    # population courante
mutation_rate <= float           # taux de mutation (entre 0 et 1)

# Variables de la fonction
mutants = list()                                    # liste stockant les nouveaux individus (mutants)
n = len(pop) * mutation_rate                        # nombre d'individus à muter
m = len(genes()) * mutation_rate                    # nombre de gène à muter par individu

for i in range(n):                                  # pour chaque nouvel individu à muter
        ind = random.choice(pop)                            # choisi un individu au hasard parmi la population
        mutant = Individual()                       # créer un nouvel individu (mutant)
        mutated_genes = random.sample(random.shuffle(genes()), m)      # sélectionne m gènes à muter

        # effectue les mutations sur le génotype du mutant
        for gene_id in genes():
                if em in mutate_ass:        # si il appartient aux gènes à muter, on change le code
                        # passe le code de True à False ou inversement
                        code = True if ind.get_gene(gene_id) == False else False
                else:
                        code = ind.get_gene(em[0], em[1])          # conserve le même code

                mutant.set_gene(gene_id, code)              # renseigne le code du gène

        # calcul les données et ajoute le mutant à la nouvelle population
        mutant.compute()
        mutants.append(mutant)

return mutants              # retourne la liste des mutants
```

4.6 Sélection

La sélection est une opération de l'algorithme génétique qui permet de sélectionner les meilleures individus d'une population pour qu'ils puissent continuer à se développer.

Notre version de l'opération est très simple: nous nous contentons de calculer le score pour chaque individu, de trier la population en fonction du score obtenu et de retourner les n meilleurs individus. **La valeur de n est égale à la taille de la population initiale**.

Le pseudo-code suivant vous présente l'algorithme tel que nous l'avons implémenté:

```
# Paramètres de la fonction
pop <= list()                    # population courante
n <= int                         # nombre d'individus à sélectionner

# calcul le score pour chaque individu (utilise la fonction d'évaluation)
for ind in pop:
        score(ind)

# tri la population en fonction du score des individus
pop = sorted(pop, key=lambda ind: ind.score, reverse=True)        # du plus grand au plus petit score

return pop[0:n]          # retourne la liste des m meilleurs individus
```

4.7 Paramètres

Au cours de l'élaboration de l'algorithme, nous avons identifié plusieurs paramètres:

- **p**: pondération entre l'amélioration de la sécurité (maximisation du PCC) et la réduction du coût total (minimisation du budget) (voir partie évaluation).
- **crosses_rate**: taux de croisement pour une population (voir croisements)
- **mutation_rate**: taux de mutation pour une population et le génotype (voir mutation)
- **target_budget**: budget que le RSSI ne souhaite pas dépasser (voir évaluation).

L'un des difficultés des algorithmes génétiques est de trouver les paramètres les plus efficaces pour atteindre la solution optimale le plus rapidement possible. C'est une recherche empirique, car il n'existe aucune formule pour trouver ces facteurs. Dans le cadre de notre étude, nous avons fait le choix d'utiliser les paramètres suivants:

- p = 0.75
- crosses_rate = 0.25
- mutation_rate = 0.25
- target_budget = 300

Nous avons invitons à découvrir les résultats de la résolution dans la partie suivante.

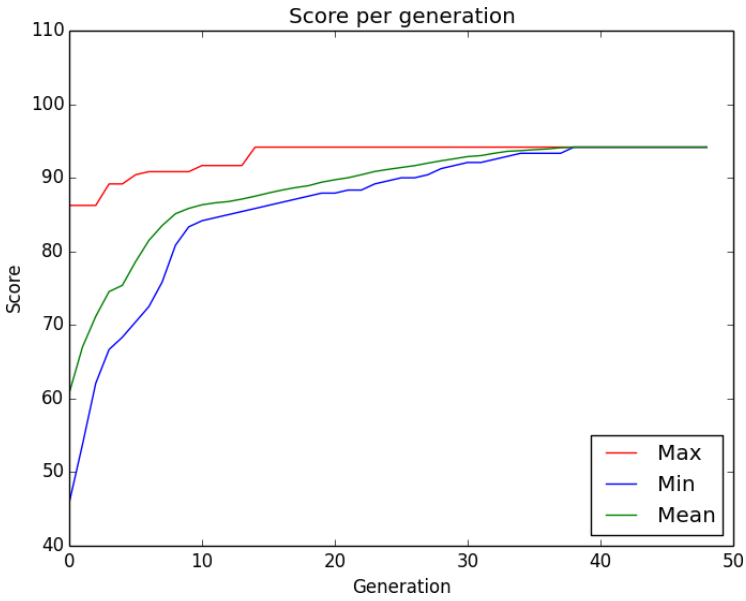
5. Conclusion

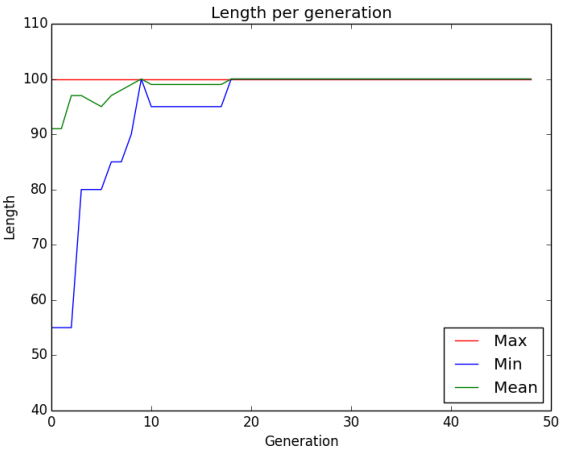
5.1 Résultat

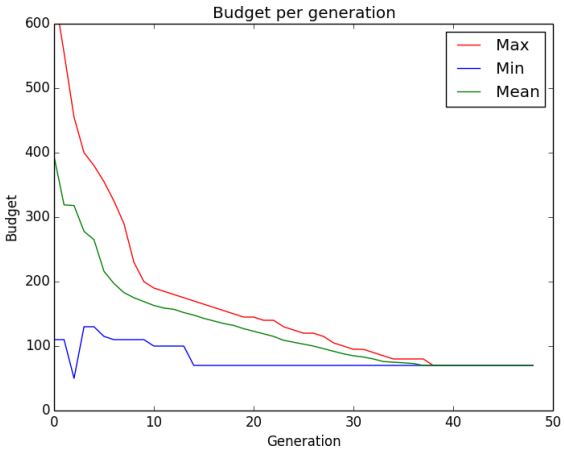
Les résultats d'un algorithme génétique sont très variables. En effet, de nombreuses fonctions sont réalisées **de manière aléatoire**, et il est impossible de prédire le nombre de génération nécessaire pour atteindre la meilleure solution(surtout en c, l'aléatoire est plus mauvais qu'en python, il nous faut donc plus de génération pour obtenir le même résultat (du au modulo)).

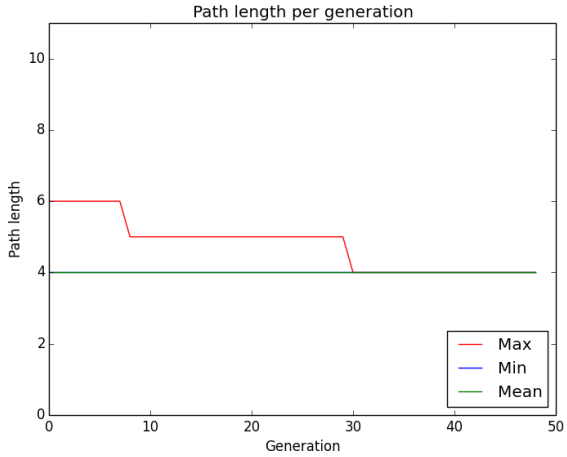
Pour que nous puissions vérifier nos paramètres, nous avons lancé 2 simulations basées sur le même jeu de test. Les évolutions sont retranscrites dans les graphiques suivants:

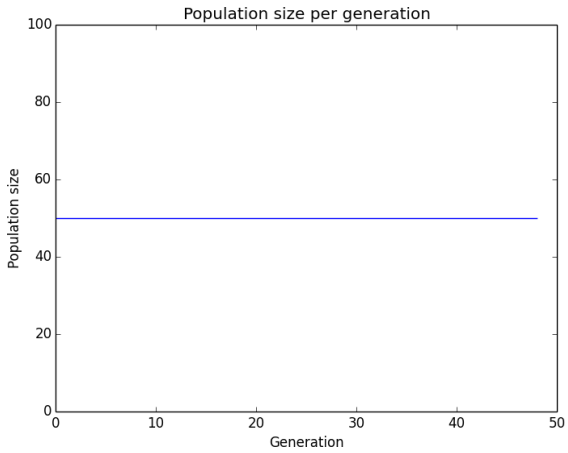
Première simulation



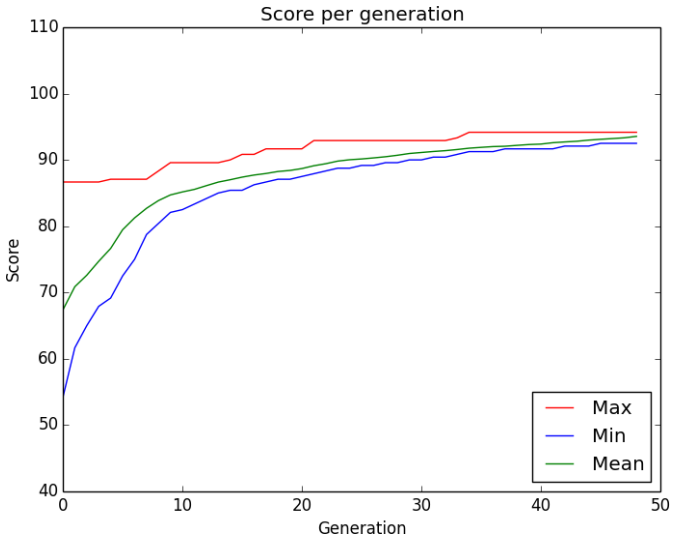


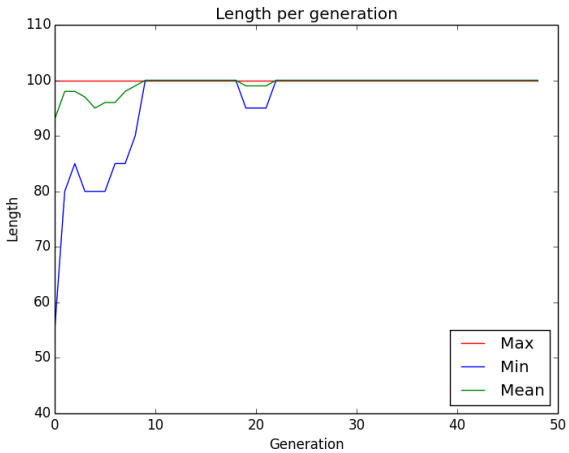


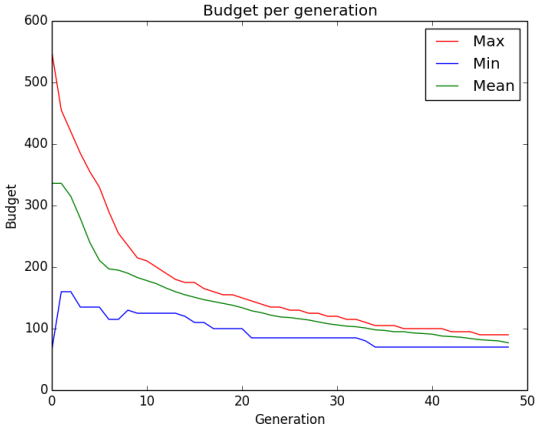




Deuxième simulation







Dans la première simulation, on voit que le score maximal est atteint vers la 12e génération alors que dans la deuxième simulation, l'optimal n'est atteint qu'au bout de la 32e génération. De plus, on peut également constater que les graphiques ont des formes différentes (scores, budgets, longueurs du PCC …).

Le graphique "taille de la population" de la première simulation montre également qu'entre chaque génération, le nombre d'individus est constant. Cela est dû à la fonction de sélection qui vérifie que la taille de la population reste identique à celle initiale (voir partie).

A la fin des simulations, nous obtenons les résultats suivants (TOP 3):

- 1er choix: e0 + m1, e12 + e8, e4+e4
  - score: 94.17, longueur du PCC: 100, PCC: {v0, v2, v6, v11}, budget: 70 k€
- 2e choix: e0 + m1, e12 + e8, e4+e4
  - score: 94.17, longueur du PCC: 100, PCC: {v0, v2, v6, v11}, budget: 70 k€
- 3e choix: e0 + m1, e12 + e8, e4+e4
  - score: 94.17, longueur du PCC: 100, PCC: {v0, v2, v6, v11}, budget: 70 k€

Les 3 meilleurs individus de la population finale sont **identiques** et montrent qu'**il est possible d'atteindre une sécurité absolue pour un budget de seulement 70 k€.**

5.2 Bilan

Les résultats que nous avons obtenu par l'algorithme sont **très positifs**. En effet, nos calculs manuels n'ont pas permis d'obtenir une amélioration si grande pour un budget si faible. **C'est la preuve que l'algorithme génitique est un bon outil d'aide à la décision**.

Même si les tendances de l'algorithme sont **incertaines**, l'implémentation est **facile** à réaliser et les résultats obtenus **suffisamment fiables** (mais pas forcément optimaux) pour être utilisé par un agent de production. Nous pensons toutefois qu'il est nécessaire de **mettre à l'epreuve le programme** par des méthodes statistiques et des cas concrets.

Concernant la conception de l'algorithme génétique, **nous avons dû faire plusieurs modifications par rapport aux documents fournis**. Notre modèle étant moins normé que celui prévu (état de vulnérabilité à tous niveaux du graphe, rapport longueur/budget), ces adaptations ont été ajoutées au fur et à mesure sans trop impacter les performances.

Pour conclure, nous pensons que l'utilisation des algorithmes génétiques dans le cadre des problèmes de décision liés à la sécurité informatique est intéressante, mais demande plus de développement pour être exploitée de manière industriel. De nombreux cas de décisions restent difficiles à représenter (fonction d'évaluation normée, identification des noeuds selon des zones …) et ces méthodes méritent d'être intégrées dans le cadre d'une norme, tel que celle de la famille ISO/IEC 27000.