# TP: Mise en œuvre d'une intégration continue pour de la production de graphe réseaux avec Graphviz

L'objectif de ce TP est de :

- vous initier aux outils de générations de graphiques à partir d'une description textuelle, via une syntaxe spécifique ;
- développer vos compétences sur une chaine CI/CD (récupération des *artefacts*) ;
- vous initier au traitement de données JSON.

La définition d'un graphique via une description textuelle s'appuie sur un "*domain-specific language*" (DSL), c'est à dire un langage de programmation dédié à une tâche précise, par opposition à un langage de programmation généraliste. Voir `https://fr.wikipedia.org/wiki/Langage_d%C3%A9di%C3%A9`

Cette pratique peut semble rébarbative a priori, et de nombreux étudiants préfèrent utiliser des outils de dessin soit sous forme d'appli web, soit sous la forme d'utilitaires installés sur la machine. Ceci semble plus accessible et/ou rapide. Mais l'utilisation d'un DSL a cependant l'avantage de pouvoir être intégré à une chaîne CI/CD d'un projet, garantissant que le graphique pourra être mis à jour automatiquement, simplement en modifiant le fichier de description. Alors que dans le cas d'un graphique généré de façon statique, il faudra reprendre le dessin "à la main".

Le logiciel Graphviz sera utilisé ici, il a vocation a produire des graphes, c'est à dire des graphiques composés de **nœuds** connectés par des **arcs**. Cette représentation est très adaptée à de nombreuses situations, et en particulier en informatique, télécommunications, réseaux, etc.

Graphviz dispose de plusieurs outils (commandes), qui utilisent tous la même syntaxe d'entrée, mais qui seront utilisés en fonction du type de graphe à produire. Des exemples de ce qu'il peut générer sont présentés ici : `https://graphviz.org/gallery/`, allez voir.

Vous pouvez aussi prendre quelques minutes pour explorer les liens suivants :

- `https://en.wikipedia.org/wiki/Graphviz`
- `https://graphviz.org/`
- `https://fr.wikipedia.org/wiki/Graphe_(math%C3%A9matiques_discr%C3%A8tes)`

## 1 Utilisation en local

Q1.1 - Installer le logiciel : `$ sudo apt install graphviz`

Puis vérifier le bon fonctionnement avec la commande suivante :

```
$ echo 'digraph { a -> b }' | dot -Tsvg > output.svg
```

Q1.2 - Vérifier la présence et le contenu du fichier `output.svg` : l'ouvrir avec l'application par défaut (en tant qu'image). Essayer de zoomer. Quel est l'intérêt principal des images au format SVG ?

Q1.3 - Ouvrir le fichier avec un éditeur texte et observer le contenu. Quelle est la nature de la syntaxe de ce fichier ?

Q1.4 - Créer le fichier `g1.dot`

```
graph {
0 [shape="box"];
0 -- 1;
2 -- 2;
3 -- 2;
4 -- 1;
0 -- 4;
4 -- 1;
}
```

(Note : le nœud n° 0 se voit ici assigner un attribut spécifiant une forme rectangulaire.)

Q1.5 - Créer un script `run1.sh` contenant ceci :

```
tool=dot
$tool -Tpng $1.dot >$1.png
$tool -Tsvg $1.dot >$1.svg
```

Le rendre exécutable, puis taper la commande `$ ./run1.sh g1`
Vérifier les deux fichiers générés, et donner leur taille :
svg : png :

Q1.6 - Copier le fichier `g1.dot` en `g2.dot` et le modifier comme suit :

```
digraph {
0 [shape="box"];
0 -> 1;
2 -> 2;
3 -> 2;
4 -> 1;
1 -> 4;
0 -> 4;
}
```

Q1.7 - Taper ensuite la commande `$ ./run1.sh g2` et vérifier les deux fichiers générés. Quelle est la différence avec le 1er graphe ?

Q1.8 - Créer un fichier `g3.dot` contenant une spécification du graphe ci-dessous, en laissant attributs de nœud et d'arc par défaut. Il faut néanmoins assigner des **labels** aux arcs, comme ceci : `B--G [label="4"];`

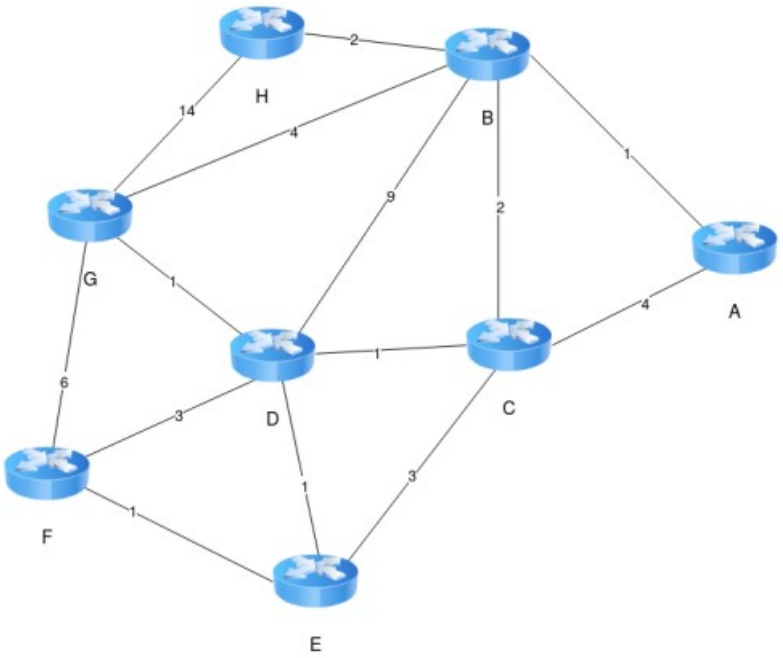



Figure 3: Astract graph model of a computer network

Le compiler, et vérifier le résultat.

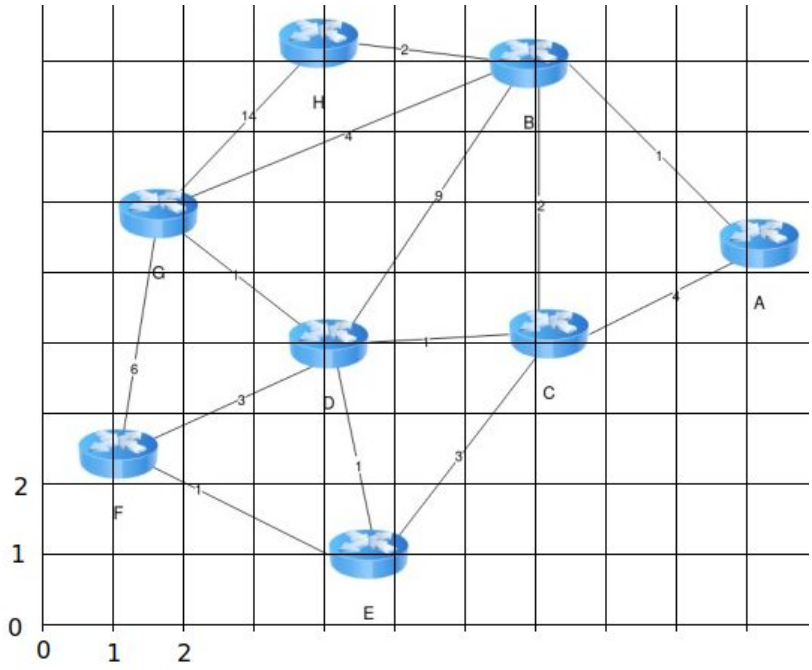
Q1.9 - De multiples personnalisations sont possible, aussi bien pour les arcs que pour les nœuds, via les "*attributes*". Voir la liste complète ici : `https://graphviz.org/doc/info/attrs.html`

Ces attributs peuvent être assignés individuellement pour chaque arc et nœud, mais il est possible de les assigner de façon globale pour tout le graphe. Par exemple :

```
graph {
node [shape="star" style="filled" fillcolor="aqua"];
edge [fontsize="20pt" color="red"];
...
```

Insérer ceci dans votre fichier, le recompiler, et observez le résultat.

Q1.10 - Graphviz permet de spécifier la position des nœuds en leur assignant un attribut "pos". Par exemple, ceci : `A [pos="1,5!"];` va placer le nœud A en $x = 1, y = 5$[1]. Le point d'exclamation indique que la position est impérative, à défaut il est possible que Graphviz l'adapte en fonction du graphe.
Copier le fichier `g3.dot` en `g4.dot` est assigner aux nœuds des positions. Vous pourrez utiliser cette grille pour obtenir les positions approximatives :




Pour assigner une position, il faut utiliser une autre commande de Graphviz : `neato`. Modifier le script `run1.sh` comme ceci :

```
#tool=dot
tool=neato
$tool -Tpng $1.dot >$1.png
$tool -Tsvg $1.dot >$1.svg
```

Compiler votre fichier et vérifier le résultat.

Q1.11 - Graphviz permet d'avoir des images pour les nœuds, via un attribut "image", voir ici :
`https://graphviz.org/docs/attrs/image/`
Attention, pour ne pas que l'image soit insérée dans une forme, il faut indiquer que pour l'attribut "shape", il n'y a rien : `[image="image.png" shape="none"];`
Ajouter de façon globale pour tous les nœuds, l'image de routeur accessible ici :
`https://github.com/skramm/but3_rt#2---fichiers-ressources-pour-tps-but3`

Q1.12 - Pour finir, augmenter la taille de la police (arcs & nœuds), et mettez le label des noeuds en rouge et le label des arcs en bleu (ceci se fait bien sur de façon globale pour tout le graphe). Faites valider par l'enseignant.

1. Les unités sont gérées en internes par Graphviz

# 2 Mise en place d'une intégration continue

## 2.1 Démarrage

Q2.1 - Créer sur votre compte Github un dépot public nommé `graphes` et le cloner dans votre "home".

Q2.2 - Y ajouter le fichier `run1.sh` ainsi que tous les fichiers "dot".

Q2.3 - De façon similaire à ce qui a été fait dans le TP "Mise en place de tests unitaires", créer un fichier d'intégration continue : `.github/workflows/graphviz.yml`, contenant ceci :

```
name: graphes
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - name: Installation de graphviz
      run: sudo ...
    - name: Check out repository code
      uses: actions/checkout@v3
    - name: Lancement graphviz
      run: ./NOM-SCRIPT.sh g1
```

Q2.4 - "Commiter" et propager au dépôt, puis examiner sur l'interface web le résultat (onglet "Actions"). Corriger les erreurs éventuelles, et recommencer jusqu'à ce qu'il n'y en ai plus.
En l'état, les graphiques sont générés, mais il n'est pas possible d'y accéder. Il faut préalablement présenter le concept d'"**artefact**".

## 2.2 Gestion des *artifacts* Github

Dans une chaîne CI/CD, certains "workflows" vont générer des fichiers lors de son exécution. On appelle ces fichiers des **artifacts**. Celà peut être des images, comme dans ce TP, ou des logs de compilation, des résultats de tests d'intégration, des screenshots, des fichiers pdf, etc.

La problématique est que, une fois le lancement du "workflow" terminé, ces images sont stockées sur le "runner", la VM que Github instancie au moment de l'exécution. Or, il n'est pas possible d'y accéder directement, il faut passer par une étape d'**upload**. Ceci consiste à transférer les artifacts depuis le "runner" jusqu'à un espace de stockage interne, géré par Github.

Certains de ces artifacts peuvent être nécessaires à l'exécution d'autres "jobs" appartenant au même "workflow". Par exemple des résultats de différents tests qui peuvent être agrégés pour produire un unique fichier HTML ou pdf, compilé dans un autre "job".

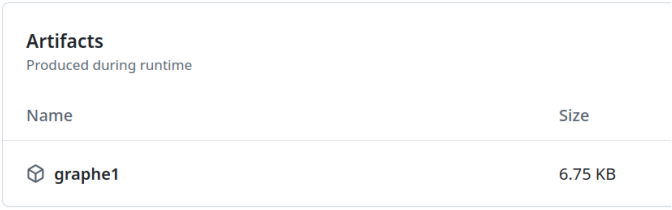
Pour pouvoir récupérer dans un "job" des artifacts produits par un autre "job", il faut passer par une action de **download**, qui va transférer le/les fichier(s) depuis l'espace de stockage évoqué ci-dessus jusqu'au "runner" dans lequel le workflow est exécuté.

Doc de référence :
`https://docs.github.com/fr/actions/writing-workflows/choosing-what-your-workflow-does/storing-and-`

Q2.5 - Pour mettre en place la fonction d'"upload", ajouter dans le fichier `.yml` les lignes ci-dessous dans les "steps", afin d'"uploader" le fichier `g1.svg` :

```
    - name: upload du resultat
      uses: actions/upload-artifact@v4
      with:
        name: (donner un nom arbitraire)
        path: chemin/vers/le/fichier
```

Q2.6 - Commiter et propager les changements. Puis, dans l'interface web, visualiser l'exécution du workflow. Une fois le "workflow" terminé, vous devez voir en dessous l'apparition d'un cadre "Artifacts", avec un lien de téléchargement :




Q2.7 - Télécharger le fichier en cliquant dessus et vérifier que le fichier zip contient bien l'image attendue.

Q2.8 - On veut maintenant que l'ensemble des graphes au format SVG soit générés, et "uploadés". Dupliquer le script `run1.sh` en `run2.sh` et modifier ce dernier :

```
for a in *.dot;
do
    a2="${a%.*}"
    $tool -Tsvg $a > $a2.svg
done
```

(Note : la ligne `a2="${a%.*}"` permet de récuperer dans a2 le nom de fichier sans extension.)

Q2.9 - Vérifier son fonctionnement en local : `$ ./run2.sh`, puis vérifier que tous les SVG ont été regénérés.

Q2.10 - Il faut aussi modifier le fichier `.yml` : d'une part, il faut demander l'exécution de `run2.sh` au lieu de `run1.sh`. D'autre part, il faut demander l'upload **tous** les fichiers SVG générés. Ceci se fait en indiquant dans "path" une spécification de fichiers au lieu d'un nom de fichier unique.
Par exemple : `path: un/chemin/*.extension`
Commiter tous les changements, les propager, et vérifier que vous pouvez bien récuperer l'ensemble des fichiers SVG (sous la forme d'une archive zip) dans le cadre "Artifacts" après l'exécution du "workflow" de Github-Actions.

## 2.3 Téléchargement direct

Github permet via son API REST de récupérer les artifacts stockés lors des exécutions de "workflow". Ceci permet de scripter le téléchargement des fichiers générés, sans avoir à passer par l'interface web. Mais ceci nécessite de disposer d'un "token", qu'il faut générer au préalable.

Doc : `https://docs.github.com/en/rest/actions/artifacts`

Q2.11 - Générer un token via cette procédure :

(a) Sur l'interface Web de Github, cliquer sur son avatar tout en haut à droite, et sélectionner "Settings".
(b) Dans le menu qui apparait à droite, tout en bas, sélectionner "Developper Settings".
(c) Dans le menu de droite qui apparaît, choisir "Personal access tokens" → "tokens (classic)".
(d) Sélectionner "Generate new token (classic)". Lui donner un nom (`TP-graphviz`, par exemple), et activer en dessous les cases "repo" et "workflow".
(e) Valider tout en bas (bouton "Generate token").
(f) Copier le token généré, et le coller dans un fichier `token.gh` placé dans votre "home" (**PAS dans votre depot ! Pour des raisons de sécurité, ce genre de fichier ne doit surtout pas être versionné**).

Q2.12 - Créer un fichier `get-artifacts1.sh`, lui donner les droits d'exécution, et y placer la requete Curl suivante :

```
curl \
   -H "..." \
   -H "..." \
   -H "..." \
   https://api.github.com/repos/OWNER/REPO/actions/artifacts
```

Ajouter les 3 entêtes HTTP nécéssaires (voir lien github ci-dessus) et compléter par les éléments à remplacer.

Q2.13 - Exécuter ce script. Vous devez observer dans la console toutes les données sur les artifacts, sous un format JSON.

Q2.14 - Modifier le script en ajoutant une redirection de la sortie vers un fichier `gh-artifacts.json`. Relancer le script et vérifier que vous avez bien le fichier. Examiner son contenu avec un éditeur texte.

Q2.15 - Pour télécharger le fichier désiré, il faut d'abord récuperer son ID, qui est affiché dans les données transmises par Github en réponse à la requete ci-dessus. Dans ce script, ajouter une deuxième requete curl en dessous, similaire à la première, mais en ajoutant au bout de l'URL l'ID de l'artifact que vous voulez récupérer, et "/zip".
Par exemple : `.../actions/artifacts/45789574/zip`
Ajouter également à la requete Curl l'option `--output artifact.zip`. En dessous, ajouter la commande pour dézipper le fichier : `unzip artifact.zip`.

Q2.16 - Relancer le script et vérifier que vous obtenez bien dans le dossier courant le fichier produit via Github.

**Problème :** en l'état, ce téléchargement du résultat ne peut pas être automatisé, puisqu'il faut aller manuellement récupérer l'ID dans le fichier JSON. Idem si plusieurs "artifacts" sont générés et qu'on veut en récupérer un en particulier.
La solution passe par une extraction automatique de l'ID via un traitement des données JSON.

## 2.4 Extraction d'informations automatisée dans du JSON

L'outil `jq` permet de traiter et d'extraire des données depuis un fichier JSON de façon scriptée :

```
$ jq ...COMMANDE... fichier.json
```

Il peut aussi s'utiliser comme "filtre Shell", par exemple :

```
$ curl ...options... URL | jq ...COMMANDE...
```

Vous trouverez en ligne de nombreux tutoriels (cliquer sur le lien !)

Q2.17 - Exécuter la commande suivante :

```
$ jq '.artifacts[]' gh-artifacts.json
```

et comparer avec le contenu du fichier d'origine : vous récupérez uniquement le contenu du tableau "artifacts"

Q2.18 - Dans le fichier .yml, vous avez donné un nom à l'artifact. Ce nom apparait dans le fichier JSON. Quelle est la clé sous laquelle ce nom apparait dans le tableau ?
Exécuter ensuite la commande suivante, en indiquant le nom de cette clé :

```
$ jq '.artifacts[].NOM-CLÉ' gh-artifacts.json
```

Q2.19 - Proposer un filtre pour jq qui permet de récupérer l'ensemble des ID de tous les artefacts, et rediriger vers un fichier `liste-id.txt`. Écrire ensuite un script `getall.sh` qui va itérer sur ce fichier et télécharger l'artefact référencé.