



SUPER MARIO GRAPHE

Algorithme de Ford-Fulkerson 22/06/19



SAADA KHELKHAL ADEL
TOUHAMI RABAH

Table des matières

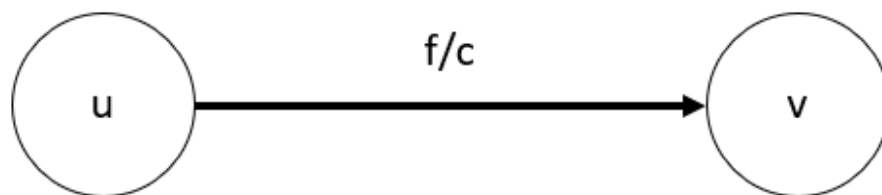
I. Généralité	2
a. Qu'est-ce qu'un réseau ?	2
b. Applications possibles	4
c. Problème du flot maximum	5
II. Algorithme de Ford-Fulkerson	6
a. Définition	6
b. Principe de l'Algorithme	7
c. Méthode	8
d. Exemple	8
f. Complexité	11
III. Projet Mario	12
a. Concept	12
b. Commandes du jeu	14
c. Eléments interactifs du jeu	15
d. Architecture du projet	16
e. Création 1/2 [Forme]	17
f. Création 2/2 [Fond]	18
g. Implémentation de l'algorithme	20
h. Répartition des tâches	24
IV. Conclusion	25
V. Documentation / Annexes	26
a. Ressources Informationnelles	26
b. Ressources Visuelles	26
c. Outils/Logiciels	26
d. Annexes	26

I. Généralité

a. Qu'est-ce qu'un réseau ?

Un réseau de flot est un **graphe orienté** où chaque arête possède une **capacité** (positive ou nulle) et peut recevoir un **flot**.

*On notera la capacité d'une arête : $c(u,v)$
Avec u et v deux sommets de l'arête.*



Le cumul des flots sur une arête ne peut pas excéder sa capacité.

Les sommets sont alors appelés des **nœuds** et les arêtes des **arcs**.

Dans ce réseau, il y a aussi une **source** s et un **puits** t . Aucun arc n'arrive à la source et aucun arc ne quitte le puits.

Un flot dans le réseau est une fonction à valeur réelle f : qui pour tous sommets u et v , vérifie les **3 propriétés** suivantes :

Contraintes de capacité

Le flot sur une arête ne peut excéder sa capacité

$$f(u, v) \leq c(u, v)$$

Antisymétrie

Le flot du sommet u vers le sommet v doit être l'opposé du flot de v vers u

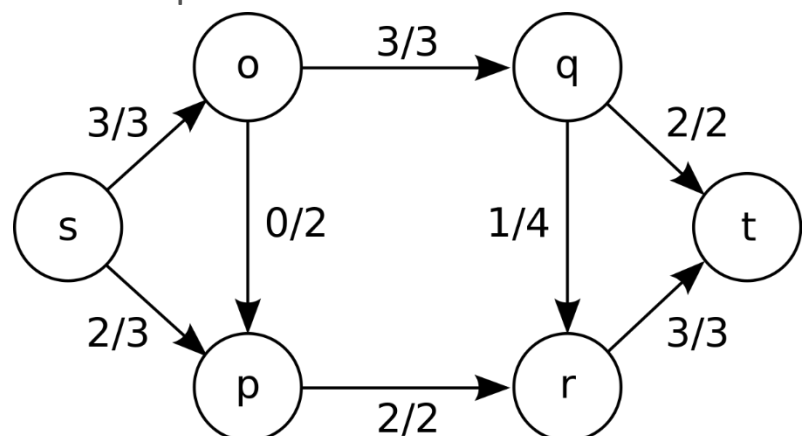
$$f(u, v) = -f(v, u)$$

Conservation de flot

Le cumul signé des flots entrant et sortant d'un nœud est nul, sauf pour la source qui en produit, ou pour le puits, qui en consomme.

$$\sum_{u \in V} f(u, v) = \sum_{z \in V} f(v, z)$$

Voici ci-dessous une représentation d'un réseau de flot :



La **capacité résiduelle** d'une arête est :

$$c_f(u, v) = c(u, v) - f(u, v)$$

Elle permet d'indiquer la quantité de capacité disponible.

si $f(x, y) < c(x, y)$:

Créer une arête dans le sens positif (x,y) avec une capacité égale à :

$$c_f = c(x, y) - f(x, y).$$

si $f(x, y) > 0$:

Créer une arête dans le sens négatif (y,x) avec une capacité égale à :

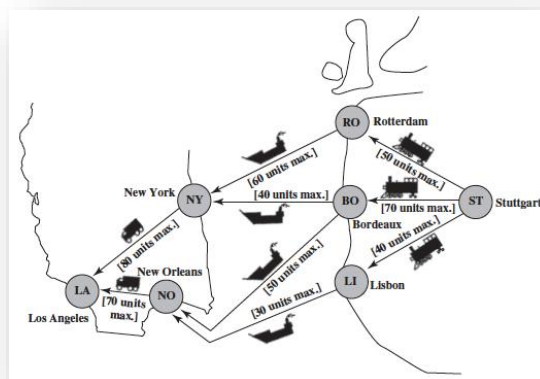
$$c_f = f(x, y)$$

b. Applications possibles

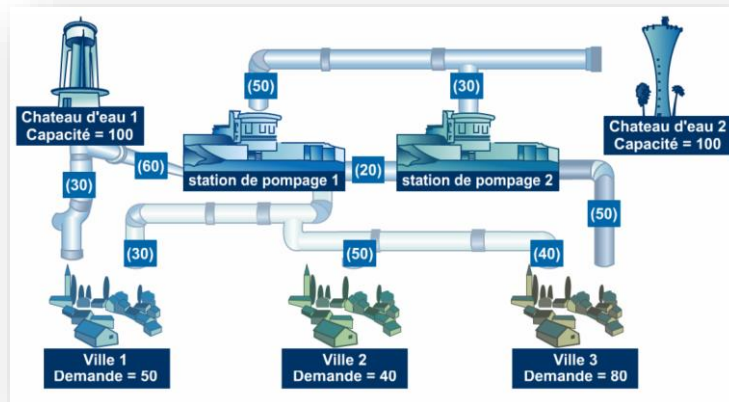
Un réseau peut être utilisé pour modéliser le trafic dans plusieurs domaines.

Réseau de transport : *trafic dans un réseau routier*

Logistique : *transport de marchandises, import/export*



Distribution d'eau : *La circulation de fluides dans des canalisations.*



Transport de pétrole : *réseau de pipelines*

Energie : *réseau EDF, centrales*

Information : *réseau téléphonique, d'entreprise...*

Flux démographiques : *flux migratoires...*

Financiers ...

c. Problème du flot maximum

Le problème du flot maximal consiste à **trouver**, dans un réseau, **un flot réalisable** depuis une source unique et vers un puit unique qui soit **maximum**.

Il existe plusieurs algorithmes permettant de résoudre ce problème : c'est-à-dire calculer la quantité maximale de matière que l'on peut faire transiter entre un point A et un point B en empruntant un réseau de routes ou de conduits dont les caractéristiques sont connues.

Nous nous concentrerons principalement sur l'**algorithme de Ford-Fulkerson**, puisque c'est à travers cet algorithme qu'a été conçu le jeu lié au projet.

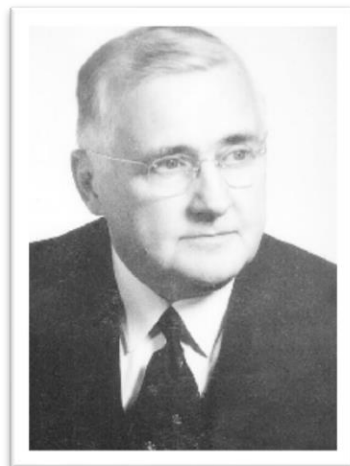
II. Algorithme de Ford-Fulkerson

a. Définition

L'algorithme de Ford-Fulkerson est un algorithme d'optimisation dans le domaine de la recherche opérationnelle.

Il répond au problème du flot maximum abordé précédemment.

Les auteurs de l'algorithme sont Lester Randolph **Ford** junior et Delbert Ray **Fulkerson**, deux mathématiciens américains.



Lester Randolph Ford junior



Delbert Ray Fulkerson

L'algorithme est paru la première fois dans des rapports techniques en 1954. (Puis dans un périodique public en 1956, voir [lien article](#) dans les sources).

MAXIMAL FLOW THROUGH A NETWORK

L. R. FORD, JR. AND D. R. FULKERSON

Introduction. The problem discussed in this paper was formulated by T. Harris as follows:

“Consider a rail network connecting two cities by way of a number of intermediate cities, where each link of the network has a number assigned to it representing its capacity. Assuming a steady state condition, find a maximal flow from one given city to the other.”

While this can be set up as a linear programming problem with as many equations as there are cities in the network, and hence can be solved by the simplex method **(1)**, it turns out that in the cases of most practical interest, where the network is planar in a certain restricted sense, a much simpler and more efficient hand computing procedure can be described.

In §1 we prove the minimal cut theorem, which establishes that an obvious upper bound for flows over an arbitrary network can always be achieved.

Cet algorithme étant une variante de l'algorithme de Busacker et Gowen.

b. Principe de l'Algorithme

1. On démarre avec un flot valide (par exemple un flot nul à chaque arc).
2. On cherche une chaîne augmentante qui est une chaîne qui va de s (source) à p (puits) le long de laquelle on peut faire passer une valeur de flux plus importante.
3. S'il existe une chaîne augmentante on augmente le flot le long de cette chaîne et on reprend à l'étape précédente. Sinon on a trouvé le flot maximal et on s'arrête.

c. Méthode

Processus **d'étiquetage** et de **scannage** en démarrant par la source et en terminant par le puit.

Au début tous les nœuds sont marqués non étiquetés et non scannés.

Scanner un nœud v signifie on se tient en v et on regarde les **voisins** de v qui ne sont pas étiquetés.

Une **étiquette** est un triplet de la forme $(u, +/-, z)$ où :

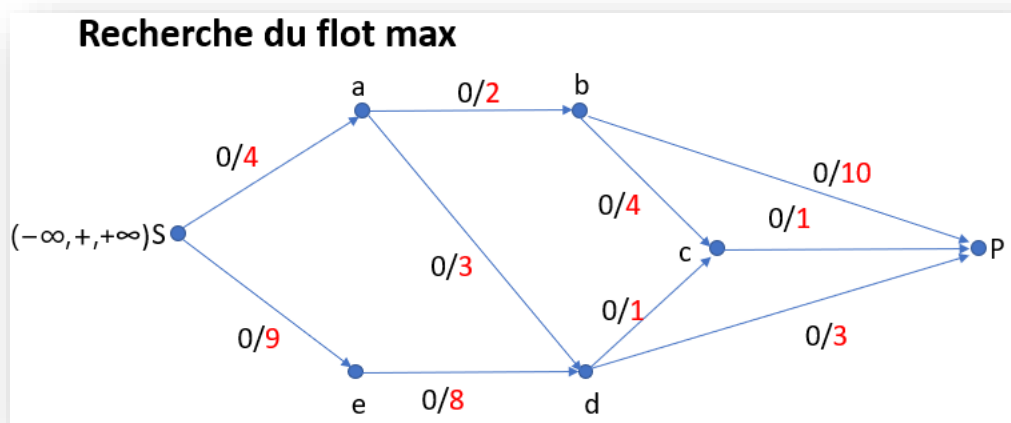
+/- c'est le sens utilisable

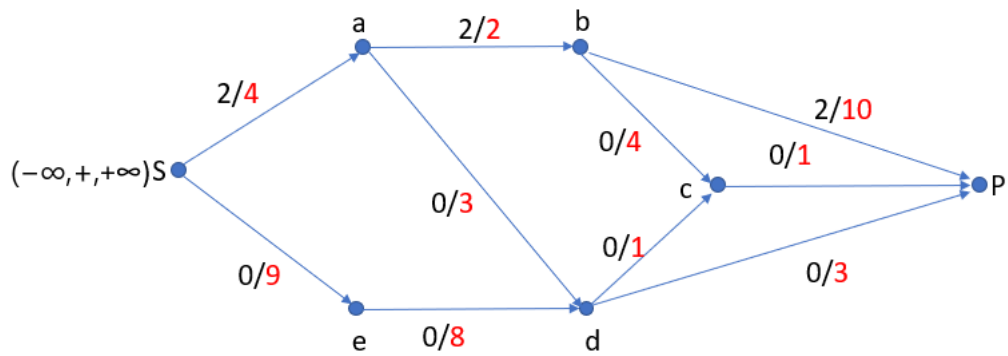
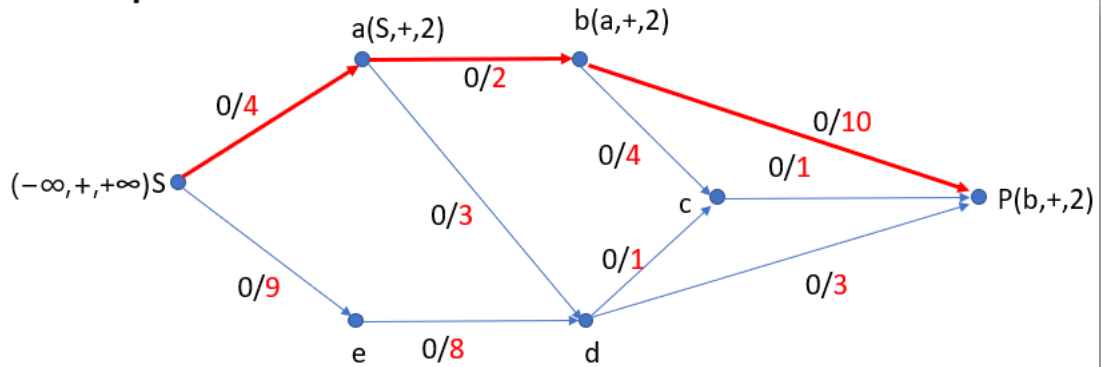
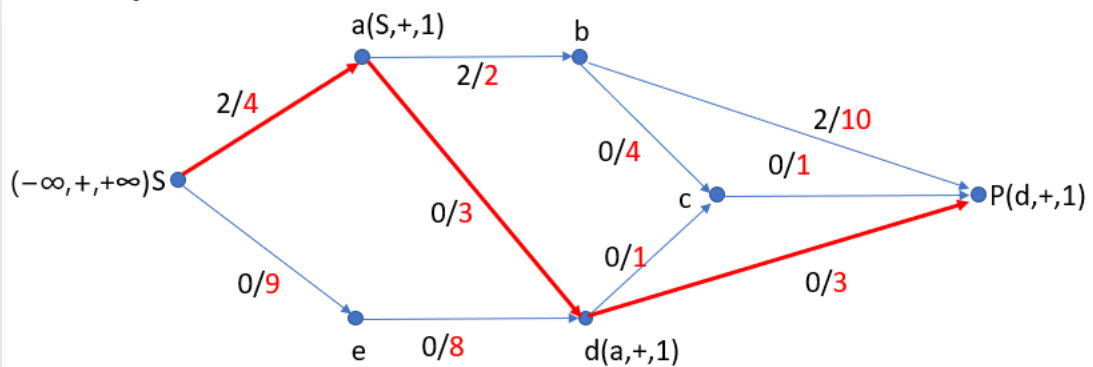
+ c'est de u vers v

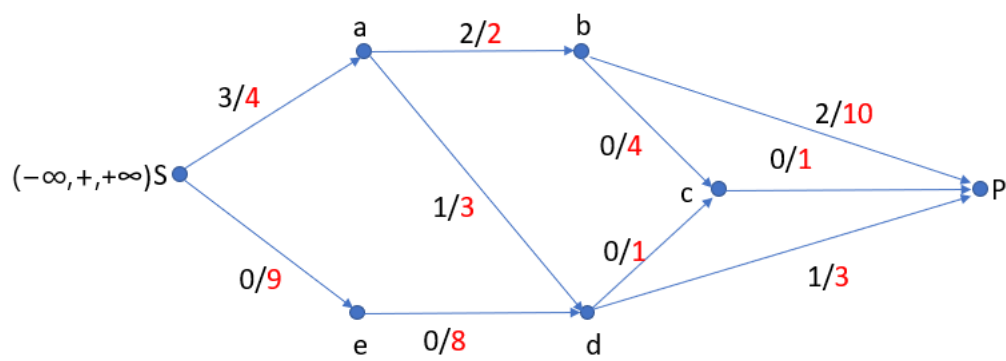
- C'est de v vers u

z la plus grande valeur qui peut être ajoutée au flot de la source jusque v .

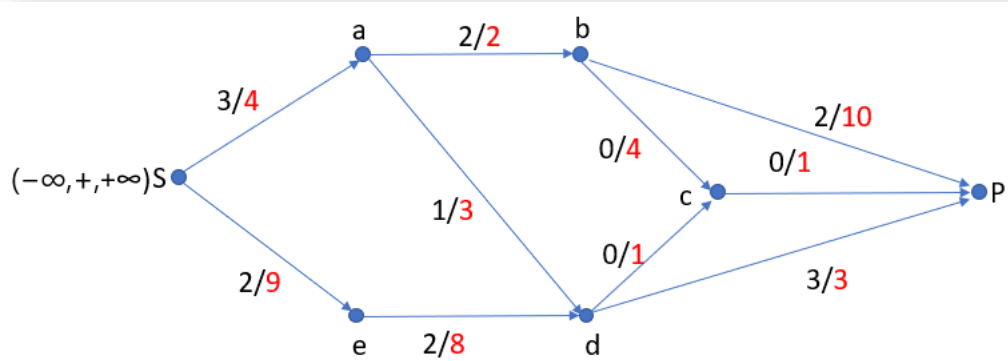
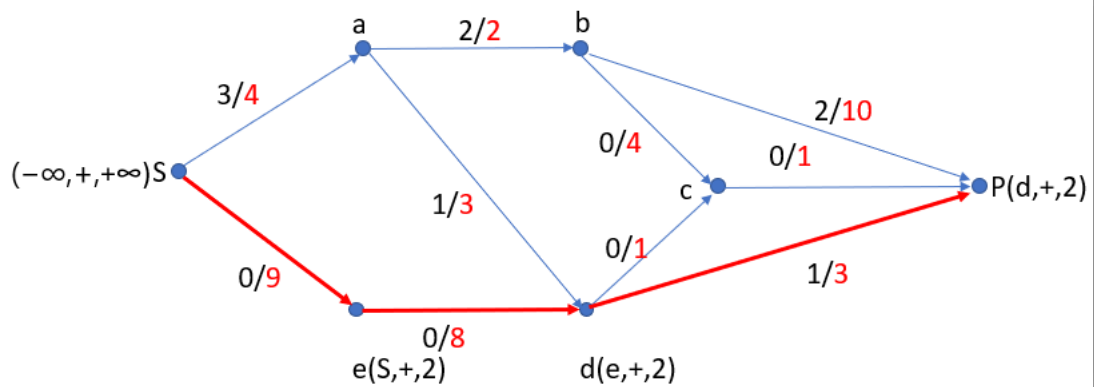
d. Exemple

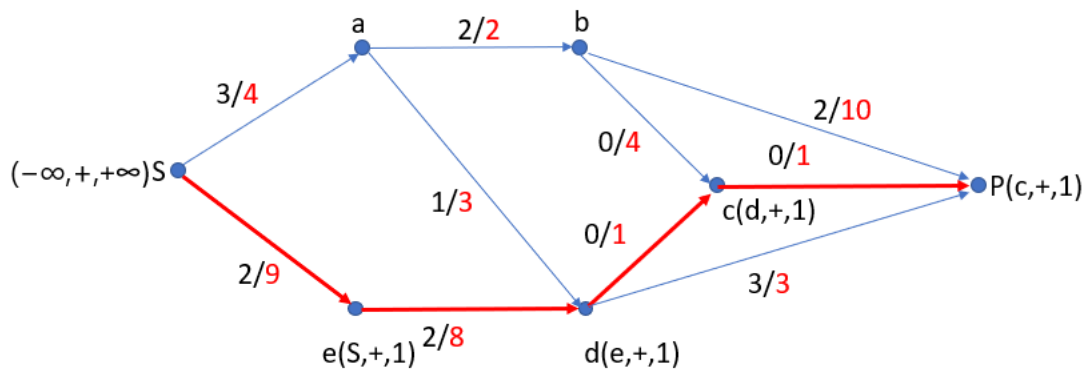
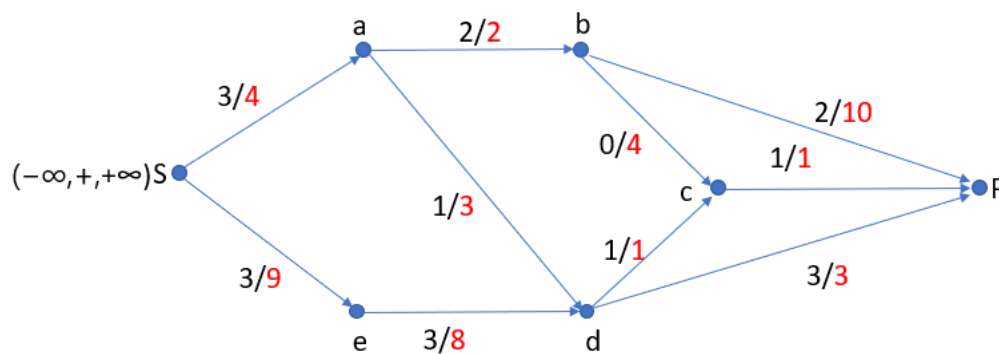


Etape 1**Etape 2**



Etape 3



Etape 4**Le flot max est atteint**

Flot max = 6

f. Complexité

La complexité de l'algorithme est en **$O(m * \text{Flotmax})$**

Pour un graphe avec m arcs.

Avec un $\text{flotmax} = 6$ et $m=7$, la complexité est de : 42.

Avec la variante de Ford-Fulkerson, (Edmonds-Karp), la

complexité est en **$O(m^2 n)$**

Pour un graphe avec n nœuds(sommets) et m arcs.

III. Projet Mario

a. Concept

Le principe est d'utiliser un jeu qui se base sur l'algorithme de Ford-Fulkerson.

L'idée imposée était de voir le personnage Mario se déplacer dans une cave et parcourir les chemins d'un graphe pour atteindre le flux maximal.

Bien que nous ayons respecté l'idée principale, nous avons légèrement modifié le concept du jeu proposé en laissant la possibilité au joueur de choisir le chemin souhaité.

Ici le joueur peut déplacer le personnage et choisir le chemin de la source jusqu'au puit.

Arrivé jusqu'au puit, il reprend le graphe au début pour continuer à chercher le flux optimisé.

Pour le **langage** nous avons naturellement opté pour Java qui est celui que nous avons le plus appris au cours de notre formation.

A la manière d'une grande partie des franchises de Nintendo, nous avons cherché un scénario simpliste et totalement secondaire pour le jeu.



-----[Début du scénario] -----

Yoshi informe Mario que la princesse Peach a été enlevée par Bowser.

Mario doit réussir à obtenir le flot maximal pour pouvoir vaincre Bowser et ainsi libérer Peach...

-----[Fin du scénario] -----

La difficulté du jeu est qu'il n'y a – *volontairement* - aucune indication sur le flot maximal à obtenir. Si vous n'arrivez plus à atteindre le puit parce que tous les chemins qui y mènent sont bouchés, et que Bowser est toujours présent dans l'écran, alors il s'agit d'une défaite.

Il ne vous restera plus qu'à recommencez la partie et être plus vigilant et stratégique.

Si en revanche vous arrivez à obtenir le flot maximal, Yoshi vous félicitera et Bowser sera vaincu.

Le jeu se joue à la fois avec le clavier et la souris.

Vous devez alterner l'écran d'affichage et la console pour le choix des chemins.

Vous trouverez dans les parties suivantes, la description des commandes et des éléments interactifs du jeu.

b. Commandes du jeu

COMMANDES DU JEU



Les touches directionnelles pour déplacer Mario.



La touche 'G' pour revenir à la source.

(Uniquement quand vous êtes coincé dans une route sans chemin accessible)



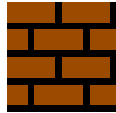
Le clavier numérique pour la saisie des numéros liés aux chemins proposés.



La touche 'Entrée' pour interagir avec les étoiles (sommets/nœuds) et avec Yoshi.

c. Éléments interactifs du jeu

ELEMENTS INTERACTIFS DU JEU

**Brique Marron**

Délimite le cadre du jeu.

**Brique Bleue**

Bloque le chemin entre deux sommets d'une arête.

**Brique Rouge**

Chemin inaccessible car Flot a atteint seuil de capacité.

**Etoile**

Symbolise le nœud / sommet






























**Yoshi**

Lance le graphe à chaque tour
[saisir : touche 'Entrée']

**Peach**

Donne des informations vis-à-vis
de votre score.

d. Architecture du projet

- ▼  grapheOptimi
 - >  JRE System Library [JavaSE-10]
 - ▼  src
 - ▼  audio
 - >  Audio.java
 - >  themeMarioBros.wav
 - ▼  fordFulkerson
 - >  Graphe.java
 - >  GrapheException.java
 - >  TestGraphes.java
 - >  images
 - ▼  jeu
 - >  Chrono.java
 - >  Clavier.java
 - >  Main.java
 - >  Saisie.java
 - >  Scene.java
 - ▼  objets
 - >  Bowser.java
 - >  Brique.java
 - >  BriqueBloque.java
 - >  Etoile.java
 - >  Objet.java
 - >  Peach.java
 - >  Yoshi.java
 - ▼  personnages
 - >  Mario.java
 - >  Personnage.java
 - >  Position.java

Package audio :

- **Audio** : gestion de la musique
- Le morceau de Mario bros format .wav

Package FordFulkerson :

Graphe : une classe qui permet de créer et utiliser un graphe.

TestGraphes : Contient l'algorithme de Ford Fulkerson.

GrapheException : classe qui étend la classe Exception. (Pour les erreurs)

Package images :

Les images du jeu (les personnages, la map, le décor...)

Package jeu :

Chrono : réaffiche la scène toutes les 8 millisecondes.

Clavier : contient les événements liés aux saisies du clavier. (Touche entrée, touches multidirectionnelles...)

Main : instancie la scène

Saisie : contient la gestion des erreurs de saisies clavier.

Scene : Le cœur du programme. Il affiche le décor et utilise les autres classes pour le fonctionnement du jeu.

Package objets :

Objet : tous les éléments qui peuvent interagir avec le joueur.


A part pour Bowser, toutes les autres classes étendent la classe Objet.

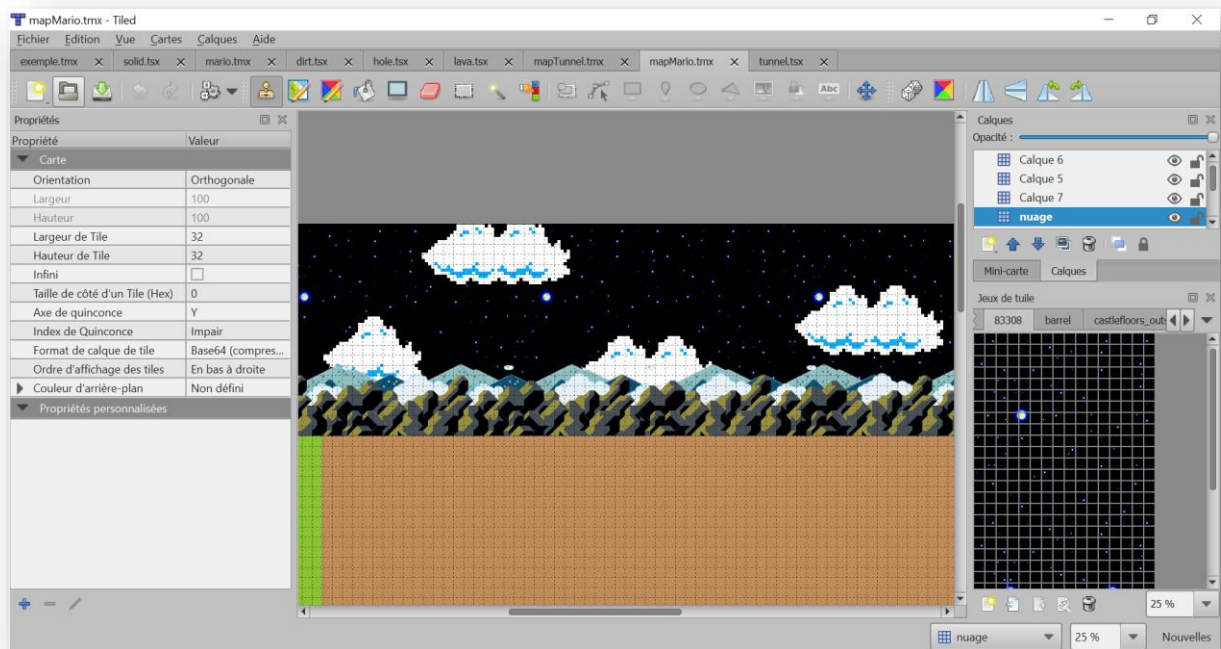
Package personnages :

Personnage : classe avec détection objet. Mario étend cette classe + détection collision + événements selon l'objet en contact.

Position : un simple enum avec des directions.

e. Création 1/2 [Forme]

Pour l'image de fond, nous avons utilisé le logiciel **Tiled** 
Un éditeur de map 2D, très pratique.



Ce logiciel nous a permis d'élaborer notre décor en associant des « tuiles » (tiles) de 32 x 32 pixels.

La grande utilité de ce logiciel c'est la superposition de plusieurs éléments assez facilement.

Nous n'avons pas utilisé de bibliothèque spécialisée pour le jeu 2D avec Java, donc n'avons pas pu maximiser l'utilisation de ce logiciel, mais il a été tout de même utile pour obtenir un décor qui nous convenait.

Concernant les éléments du jeu, nous avons récupéré les ressources via différents sites de ressources d'images 2D

« sprites ». Les gens s'occupent de récupérer les décors sur des jeux vidéo et permettent aux utilisateurs de bénéficier de ces éléments. Voici un exemple ci-dessous :



Nous avons aussi utilisé **Photoshop** pour le redimensionnement, le redécoupage, la modification et la transparence de certains éléments.

f. Création 2/2 [Fond]

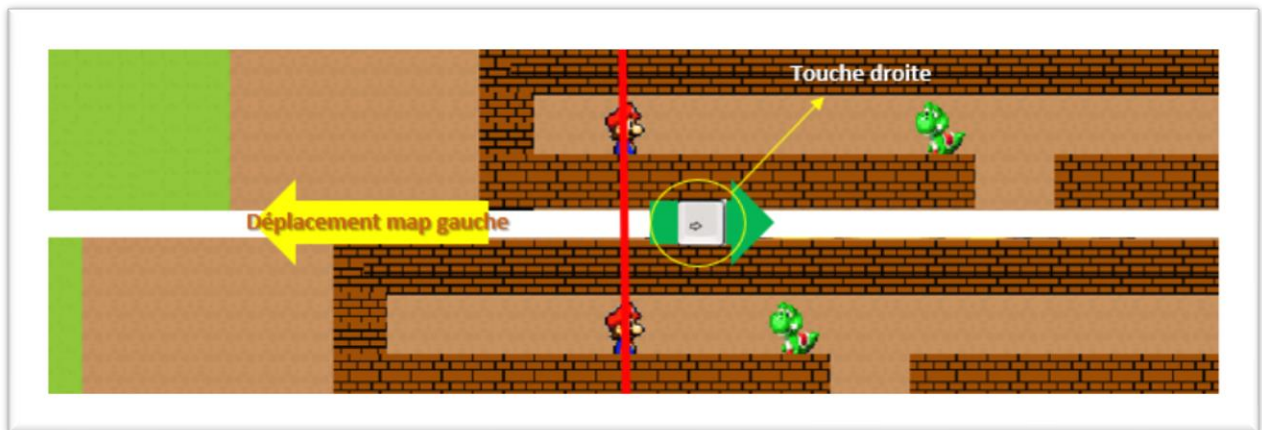
Pour le fond, nous nous sommes basés sur le tutoriel de Stefan Maurer sur YouTube, avec Mario Bros sur Java ainsi qu'un ancien projet que nous avons fait – à titre personnel - sur Pokémon.

Mais plutôt que de rester sur un décor qui ne bouge qu'à la verticale [avec deux directions : droite ou gauche] , nous avons opté pour 4 directions.

La technique est la même, il suffit d'adapter aussi à l'horizontal.

La technique de déplacement de Mario est simple, c'est le fond qui se déplace dans le sens contraire de la touche directionnelle saisie.

Il faut ensuite faire changer l'image de Mario pour simuler une marche.

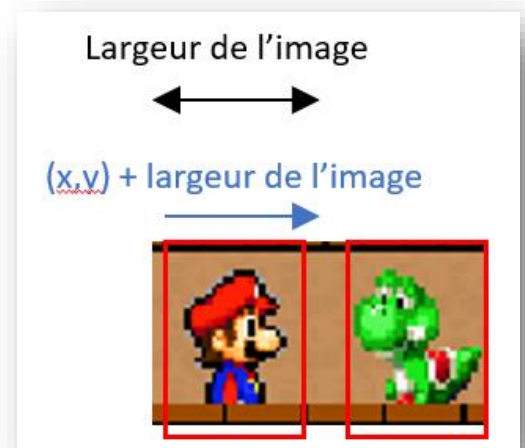


Mario **reste toujours au centre de l'écran**, c'est un effet d'illusion très connu des créateurs de jeu 2D.

Pour la détection de collision (par exemple pour une détection de collision à droite) :

La fonction détermine si l'image du personnage (point x, point y) + la largeur de l'image du personnage n'est pas supérieure ou égale à l'image de l'objet (point x, point y).

La fonction cherche **une collision** à droite, gauche, haut et bas.



Ensuite tous les éléments du décor sont positionnés dans la map. La détection entre Mario et l'un de ces éléments

se fera « *automatiquement* » puisque la **recherche de détection** est lancée en **boucle** grâce au réaffichage constant de la scène par la classe Chrono.

Nous aurions pu proposer un Mario qui court en l'associant à une touche, il aurait suffi de modifier la vitesse d'affichage de la scène. Il s'agit toujours d'une question d'illusion, finalement.

g. Implémentation de l'algorithme

Le package FordFulkerson est composé de deux classes (+ classe exception) :

- La classe Graphe
- La classe TestGraphes

Nous détaillerons d'une part la classe Graphe, puis d'autres part la Classe TestGraphes.

----- La classe Graphe -----

La classe **Graphe** est constitué de **2 variables** :

Le nombre de sommets et le tableau des adjacences :

```
private int nbrSommets;  
private int[][] adj;
```

(Nous faisons abstraction des 2 autres variables qui servent uniquement à l'animation du jeu.)

Le constructeur de l'objet reçoit en paramètre le nombre de sommet, le transmet à la variable d'objet et initialise le tableau des adjacences à 0 :

```
public Graphe(int nbrSommets) {
    this.nbrSommets = nbrSommets;
    // initialisation du tableau adjacents
    adj = new int[nbrSommets][nbrSommets];
    for (int i = 0; i < nbrSommets; i++) {
        for (int j = 0; j < nbrSommets; j++) {
            adj[i][j] = 0;
        }
    }
}
```

La méthode pour ajouter une arête (un arc)

```
/**
 * Ajoute une arête en indiquant la valeur de capacité entre 2 sommets
 * dans le tableau d'adjacence.
 * @param i (le numéro du sommet de départ de l'arête)
 * @param j (le numéro du sommet de fin de l'arête)
 * @param capacite (la capacité de l'arête)
 */
public void ajouterArete(int i, int j, int capacite) {
    adj[i][j] = capacite;
}
```

La méthode pour supprimer une arête (un arc)

```
/**
 * Supprime une arête en indiquant la valeur 0 au tableau
 * [ligne sommet D Arete] [ligne sommet A Arete]
 * @param i (le numéro du sommet de départ de l'arête)
 * @param j (le numéro du sommet de fin de l'arête)
 *
 * détail : i-----()-----j
 * |signification : pas d'existence d'une arête entre sommet i et j
 */
public void supprimerArete(int i, int j) {
    adj[i][j] = 0;
}
```

La méthode pour vérifier l'existence d'une arête (un arc)

```
/**
 * Vérifie l'existence d'une arête entre un sommet i et un sommet j
 * @param i (le numéro du sommet de départ de l'arête)
 * @param j (le numéro du sommet de fin de l'arête)
 *
 * détail : SI i------(0)-----j
 * ALORS il n'y a pas d'arête entre sommet i et j
 * @return un booléen qui vérifie l'existence.
 */
public boolean aUneArete(int i, int j) {
    if (adj[i][j] != 0) {
        return true;
    }
    return false;
}
```

La méthode pour retourner la liste des sommets/nœuds voisins d'un sommet reçu en paramètre :

```
/**
 * Retourne la liste des sommets voisins du sommet en paramètre.
 * @param sommet (pour lequel on cherche les voisins)
 * @return une liste d'entiers
 */
public List<Integer> voisins(int sommet) {
    List<Integer> listeArretes = new ArrayList<Integer>();
    for (int i = 0; i < nbrSommets; i++)
        if (aUneArete(sommet, i))
            listeArretes.add(i);
    return listeArretes;
}
```


La classe TestGraphes

Il y a deux méthodes :

- Parcours => algorithme de parcours en largeur
- FordFulkerson =>> méthode principale

La classe TestGraphes permet de retourner le « maxFlow » selon le graphe reçu en paramètre.

Il sert – *principalement* - dans le jeu à vérifier si le joueur a atteint le flot maximum.

```
public static float fordFulkerson(Graphe g, int source, int puit) {
    if (source == puit) {return 0;}
    int sommets = g.getNbrSommets();
    // creation graphe résiduel
    Graphe rg = new Graphe(sommets);
    for (int i = 0; i < sommets; i++) {
        for (int j = 0; j < sommets; j++) {
            rg.getAdj()[i][j] = g.getAdj()[i][j];
        }
    }
    // sera rempli par l'algorithme de parcours pour stocker le chemin
    int parent[] = new int[sommets];

    float maxFlow = 0; // valeur max flow
    // Tant qu'un chemin existe de la source vers le puit
    while (parcours(rg, source, puit, parent)) {
        // pour stocker le flux du chemin
        float fluxChemin = Float.MAX_VALUE;

        // trouver flow max de chemin rempli par l'algorithme de parcours
        for (int i = puit; i != source; i = parent[i]) {
            int j = parent[i];
            fluxChemin = Math.min(fluxChemin, rg.getAdj()[j][i]);
        }
        // mettre à jour les capacités du graphe résiduel
        for (int i = puit; i != source; i = parent[i]) {
            int j = parent[i];
            rg.getAdj()[j][i] -= fluxChemin;
            rg.getAdj()[i][j] += fluxChemin;
        }
        // ajout du fluxChemin au maxFlow
        maxFlow += fluxChemin;
    }
    return maxFlow;
}
```



```

/* Algorithme de parcours en largeur */
public static boolean parcours(Graphe rg, int source, int dest, int parent[]) {
    // tableau pour stocker les sommets visités.
    boolean[] sommetsVisites = new boolean[rg.getNbrSommets()];
    for (int i = 0; i < rg.getNbrSommets(); i++)
        sommetsVisites[i] = false;

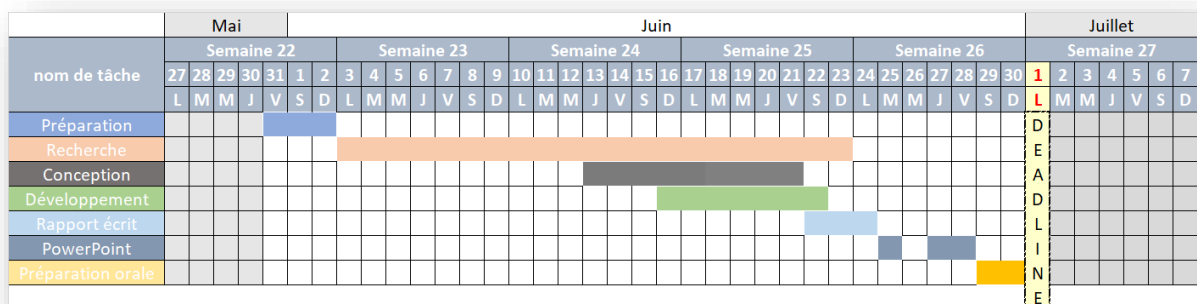
    LinkedList<Integer> q = new LinkedList<Integer>(); // une file

    // visite source
    q.add(source);
    sommetsVisites[source] = true;
    parent[source] = -1;

    // boucle sur tous les sommets
    while (!q.isEmpty()) {
        int i = q.poll();
        // vérifier les voisins du sommet i
        for (Integer j : rg.voisins(i)) {
            // Si pas visité et valeur positive alors visité
            if ((sommetsVisites[j] == false) && (rg.getAdj()[i][j] > 0)) {
                q.add(j);
                sommetsVisites[j] = true;
                parent[j] = i;
            }
        }
    }
    // retourne un boolean qui nous dit si nous sommes arrivé à destination
    return sommetsVisites[dest];
}

```

h. Répartition des tâches



(En annexe vous trouverez ce diagramme GANTT avec une plus grand dimension)

IV. Conclusion

Nous avons expliqué l'algorithme de Ford Fulkerson de manière ludique, en s'appuyant sur le monde de Mario.

Parmi les éléments que nous aurions pu améliorer si nous avions eu plus de temps, il y a notamment le fait de pouvoir partir dans le sens négatif lors de la recherche d'une chaîne augmentante.

Nous aurions pu faire en sorte que la saisie des commandes se fasse directement via l'interface graphique et non pas par console.

Concernant le graphe, nous avons un algorithme qui nous permettait d'avoir un nombre de sommets variés mais étant donné que nous fixions les positions des étoiles (images représentant les sommets) manuellement, il aurait été un peu plus compliqué de générer convenablement le visuel du graphe.

En dehors de ça, nous avons pris beaucoup de plaisir à faire ce projet qui nous a permis de mieux comprendre l'algorithme et d'évoluer dans la programmation orientée jeux-vidéo 2D.

Ce projet pourrait servir à sensibiliser un public plus jeune à l'initiation de la théorie des graphes.



V. Documentation / Annexes

a. Ressources Informationnelles

https://fr.wikipedia.org/wiki/R%C3%A9seau_de_flot

https://fr.wikipedia.org/wiki/Probl%C3%A8me_de_flot_maximum

https://fr.wikipedia.org/wiki/Algorithme_de_Ford-Fulkerson

<http://images.math.cnrs.fr/Au-feu-les-pompiers.html>

http://www.cs.yale.edu/homes/lans/readings/routing/ford-max_flow-1956.pdf

https://www.youtube.com/channel/UCWu3bBhZECqD-IfDuV_OApQ

b. Ressources Visuelles

<https://www.sprites-resource.com>

<https://opengameart.org/>

<https://fontmeme.com/fr/police-super-mario/>

c. Outils/Logiciels

<https://www.mapeditor.org/>

Photoshop

Java

d. Annexes

DIAGRAMME DE GANTT

