TIPE 2010/2011
Thème "Mobilité, mouvement"
Nicolas Klarsfeld

"Flots dans des graphes"

Introduction

Les problèmes de flots, ainsi que l'optimisation en général, allie un intérêt pratique et une grande richesse théorique.

Les deux pemière partie de ce dossier traitent de l'aspect théorique des flots.
Il existe deux sortes de modélisation : celles qui prennent en compte les coûts de passage des arcs, et les autres.
On étudiera d'abord le cas plus simple sans les coûts, puis le cas avec coûts.

Dans la 3ème partie et quatrième partie on se penchera sur l'aspect concret des flots : la partie 3 fournit des exemples d'application, et la partie 4 des exemples de résolution pour comprendre concrètement la mise en oeuvre des algorithmes.

Mais le principal apport personnel est le codage en caml que nous avons réalisé. Nous avons codé les algorithmes et surtout une interface graphique. Celle-ci permet de rentrer un graphe valué ou même doublement valué dans l'ordinateur tout en le visualisant à l'écran, et avec la possibilité de déplacer les sommets de manière interactive . La solution de problème de flot peut aussi être affichée. En raison de la longueur importante du code, nous avons préféré le mettre dans une annexe, qui contient également d'autres choses (cf sommaire page suivante).

Enfin les textes projetés pendant la présentation orale peuvent donner un aperçu rapide des problèmes étudiés.

Sommaire détaillé

1) Flot maximum

1.1) Définitions et propriétés préalables

- Notation pratique: Si (X,A) un graphe orienté simple, ie sans boucles, alors on note, pour X1 et X2 des ensembles de sommets, (X1,X2) = $(X_1 \times X_2) \cap A$ l'ensemble des arcs allant d'un sommet de X1 vers un sommet de X2 dans A.
- Si $f : A \to \mathbb{R}^+$, une valuation des arcs, alors on note f(X1,X2) = $\sum_{a \in (X_1, X_2)} f(a)$.
- On notera le complémentaire dans X avec une barre, ie si R est sous-ensemble de sommets de X, $\forall R \subseteq X, \bar{R} = X \setminus R$.
- Soit s et p deux sommets distincts. Alors on appellera une **s-p coupe**, ou coupe entre s et p, un ensemble d'arcs de la forme $(S, \bar{S})$ où $s \in S \text{ et } p \in \bar{S}$.
  Dans la suite, on omettra parfois le préfixe "s-p".
- Propriété de f(X1,X2) :
  $\forall (f : A \to \mathbb{R}), \forall R, S, T \subseteq X, f(R, S \cup T) = f(R, S) + f(R, T) - f(R, S \cap T)$
- Définition d'un s-p flot, ou **flot de s à p** : étant donné (X,A) un graphe, u une valuation positive de ses arcs (la fonction capacité), et s et p deux sommets distincts (la source et le puit), alors une valuation des arcs f est un s-p flot ssi on a les deux propriétés suivantes sur f :
  1) Conservation du flot (ou loi des noeuds) : $\forall x \in X \setminus \{s, p\}, f(x, X) - f(X, x) = 0$
  2) Contrainte des capacités : $\forall a \in A, 0 \le f(a) \le u(a)$
  Dans la suite, on appellera simplement "flot" un "s-p flot".
- La **valeur d'un flot** est : $v(f) = f(s, X) - f(X, s)$
  On la notera v pour simplifier quand f est clairement défini.
  Le problème du flot maximum est de trouver un flot dont la valeur est maximale parmi tous les flots possibles de s à p.
- On a aussi $v(f) = f(X, p) - f(p, X)$ car
  $f(s, X) - f(X, s) + f(p, X) - f(X, p) = \sum_{x \in X} f(x, X) - f(X, x) = f(X, X) - f(X, X) = 0$
- Propriété d'une coupe : si f est un s-p flot, et $(S, \bar{S})$ une s-p coupe, alors
  $v(f) = f(S, \bar{S}) - f(\bar{S}, S)$ Preuve :
  $v = \sum_{x \in S} f(x, X) - f(X, x) = f(S, X) - f(X, S)$ et X= $S \cup \bar{S}$, d'où :
  $v = f(S, S) + f(S, \bar{S}) - f(S, S) - f(\bar{S}, S) = f(S, \bar{S}) - f(\bar{S}, S)$
- Corollaire : si on appelle "capacité d'une coupe $(S, \bar{S})$ " le nombre $u(S, \bar{S})$, alors la valeur de n'importe quel s-p flot est inférieure ou égale à la capacité de n'importe quelle s-p coupe coupe. En effet il suffit d'ajouter au résultat précédent le fait que
  $f(S, \bar{S}) \le u(S, \bar{S})$ et $f(\bar{S}, S) \ge 0$.
  Autrement dit, on a $\max_{f \text{ flot}} v(f) \le \min_{(S, \bar{S}) \text{coupe}} u(S, \bar{S})$
  On va en fait voir que ceci est une égalité : c'est le théorème de Ford et Fulkerson.

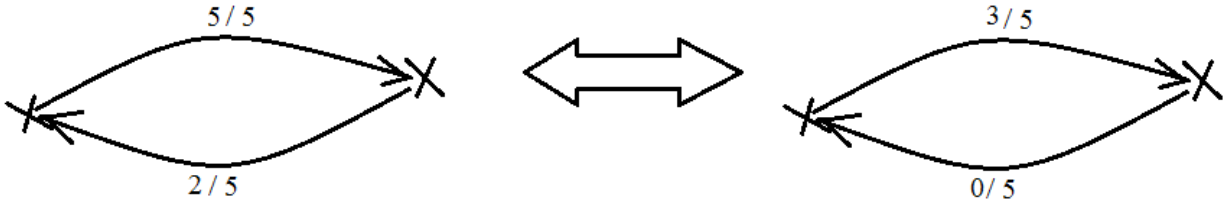
1.2) Théorème de Ford et Fulkerson (1962)

- Ce théorème est aussi appelé "théorème de flot max/coupe min".
  Supposons qu'un s-p flot a une valeur strictement inférieure à toutes les s-p coupes, et

prouvons qu'il existe alors un flot de valeur strictement supérieure. On prouvera ainsi par la contraposée que si un flot a une valeur maximale, celle-ci est égale à la capacité d'une s-p coupe au moins, ce qui prouvera le théorème.
Preuve : Définissons, pour un s-p flot f donné , l' écart d'augmentation possible suivant l'arc (x,y) : **e(x,y) = u(x,y) – f(x,y) + f(y,x)**. e(x,y) est appelée capacité résiduelle de l'arc (x,y) par rapport au flot f.
Par exemple pour le flot nul, e(x,y) = u(x,y).
On définit le **graphe d'écart** valué associé au flot f par (X,A',e) où A' est l'ensemble des arcs (x,y) tels que e(x,y) > 0. On remarque que ces arcs ne sont pas forcément présents dans A.
Alors on remarque que $v(f)=c(S,\bar{S})$ ssi $f(S,\bar{S})=u(S,\bar{S}) et\ f(\bar{S},S)=0$, soit $\forall(x,y)\in(S,\bar{S}), e(x,y)=0$, ie $(S,\bar{S})\cap A'=\emptyset$.
Ainsi, supposer que $v(f)<c(S,\bar{S})$ pour toute s-p coupe $(S,\bar{S})$ équivaut à dire que pour tout ensemble S contenant s et pas p, il existe dans A' un arc partant de S vers un sommet qui n'est pas dans S.
Ainsi, on peut partir de $S_0=\{s\}$ et, tant que S ne contient pas p, ajouter le sommet accessible dans A' depuis un sommet de S. Par récurrence, tous les sommets de S_k sont accessibles depuis s. Le processus ne peut durer indéfiniment car il y a un nombre fini de sommets Cela signifie donc que S finit par contenir p, et on a donc montré que p est accessible depuis s dans A'.
A présent il est facile d'obtenir un flot de valeur strictement supérieure à celle du flot de départ : si le chemin de s à p dans A' est $s=x_0,x_1,...,x_{n-1},x_n=p$ (on parle de **chemin augmentant**) , alors définissons la capacité résiduelle du chemin e comme valant $e=\min_{i\in[\![0;n-1]\!]} e(x_i,x_{i+1})$, qui est bien strictement positive. Alors on a sur tous les arcs (x,y) du chemin $e(x,y)=(u(x,y)-f(x,y))+f(y,x)\geq e$, donc on peut augmenter f(x,y) et éventuellement diminuer f(y,x) de façon à faire $e(x,y):=e(x,y)-e$, qu'on peut noter $e(x,y)-=e$. On remarque que cela fait disparaitre au moins un des arcs (x_i,x_{i+1}) du graphe d'écart en faisant $e(x_i,x_{i+1}):=0$.
Alors il est clair que la loi des noeuds est toujours respectée par le nouveau flot obtenu, et que la valeur de celui-ci est augmentée de e, puisque $[f(s,x_1)-f(x_1,s)]+=e\Rightarrow v+=e$.
D'où le résultat.

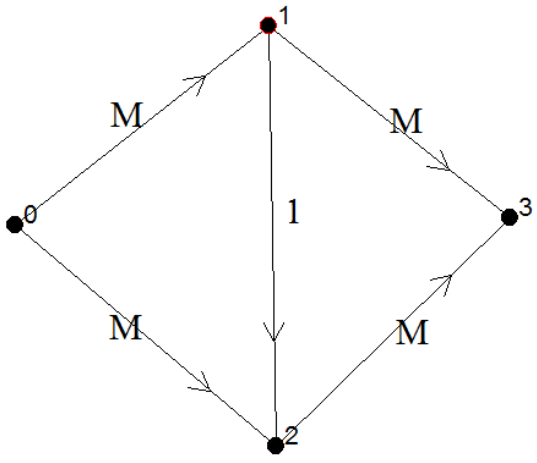
1.3) Algorithme de Ford et Fulkerson (FF)

- La preuve du théorème de Ford et Fulkerson nous fournit en plus un algorithme pour trouver un flot maximum :
- **FF :**
  { Prendre un flot nul au départ ;
  Tant que (p accessible depuis s dans le graphe d'écart (X,A'))
  { Trouver un chemin de s à p dans le graphe d'écart;
  Modifier le graphe d'écart suivant ce chemin comme il est précédemment décrit ; }
  Renvoyer le graphe d'écart }
  L'algorithme, si il se fini, nous donne un graphe d'écart final tel que p ne soit pas accessible depuis s. Alors si S est l'ensemble des sommets accessibles dans le graphe d'écart, $(S,\bar{S})$ est une s-p coupe de valeur minimale puisque $v(f)=u(S,\bar{S})$
  Le flot résultant est bien de valeur maximale d'après le théorème de Ford et Fulkerson.
- Pour récupérer la fonction f à partir du graphe d'écart, il faut tout d'abord remarquer que si l'arc (i,j) et l'arc (j,i) sont tous deux présents dans le graphe des capacités, alors certaines configurations de flots sont équivalentes :



Donc si par exemple f(x,y)<f(y,x), alors on peut poser aussi bien f ' (x,y) = 0 et f ' (y,x) = f(y,x)-f(x,y) : un des deux flots au moins peut toujours être ramené à zéro.
A partir de là, sachant que, par définition, e(x,y) = u(x,y) – f(x,y) + f(y,x), soit f(x,y)-f(y,x) = u(x,y)-e(x,y), on peut calculer k(x,y)= u(x,y)-e(x,y) et poser
* f(x,y) = k(x,y) et f(y,x)=0 si k(x,y)>=0
* f(x,y)=0 et f(y,x)= - k (x,y) si k(x,y)<0.
Alors les valeurs de f obtenues respectent bien les contraintes de capacité car on sait qu'il existe au moins une configuration de f(x,y) et f(y,x) qui les respecte d'après les modifications successives de e dans l'algorithme, et que si celle-ci ne vérifie pas les contraintes de capacités, alors toutes les configurations qui lui sont équivalentes , ie f '(x,y) = f (x,y)+a ; f '(y,x) = f(y,x)+a, ne respectent pas non plus les contraintes de capacités.

- **Terminaison et complexité** : Si les capacités des arcs sont entières, alors le flot sera atteint en au plus v_max étapes de recherche de chemin, car à chaque chemin trouvé le flot est augmenté de au moins 1. La terminaison est également assurée dans le cas où les capacités sont rationnelles (on se ramène au cas entier en multipliant les capacités par le pgcd des dénominateurs de toutes les capacités). Cependant Ford et Fulkerson on exhibé un graphe avec des capacités non rationnelles, où leur algorithme peut ne jamais se terminer et ne même pas converger vers un flot de valeur maximale.
  Dans le cas de capacités entières, puisque la recherche de chemin de s à p se fait en $O(|A|)$, la complexité de FF est donc de $O(v_{max}.|A|)$.
  On voit donc que l'algorithme n'est pas polynomial en fonction de |X|, et |A| , ie il n'existe pas de fonctions de la forme $g(|X|,|A|)=|X|^n.|A|^m$ pour m et n entiers positifs, tels que l'algorithme s'éxécute en $O(g(|X|,|A|))$.
  Ceci est bien illustré par le graphe des capacités suivant, qui peut se finir en 2M étapes (pour M entier naturel arbitrairement grand) si on choisit successivement des chemins augmentant de la forme (0,1,2,3) et (0,2,1,3), qui ont une capacité résiduelle de 1 à chaque fois.



Graphe 1

1.4) Amélioration de FF par Edmond et Karp

- Pour résoudre ce problème et rendre l'algorithme polynomial, Edmonds et Karp ont eu une idée : **choisir les chemins augmentant les plus courts possible**, ie ayant le moins d'arcs possibles (ce qui peut se faire en utilisant l'algorithme de recherche de chemin en largeur, appelée breast-first-search ou BFS en anglais).
  L'algorithme résultant a une complexité de $O(|X|.|A|^2)$. Par exemple, dans le cas du graphe 1, le nombre d'étape de la nouvelle version de FF (qu'on pourra noter EK) est seulement de 2 au lieu de 2M : les deux chemins augmentants sont ici (0,1,3) et (0,2,3).
- Pour une description et une justification de l'algorithme **BFS**, se référer à l'annexe 4.
  On y montre en particulier que sa complexité est en $O(|A|)$.
- Enfin il reste à montrer que le nombre de recherche de chemins augmentants est en $O(|X|.|A|)$ si on utilise le BFS pour les trouver, ce qui montrera le résultat d'Edmond et Karp.
  - Remarquons d'abord que la taille de (A') n'excède jamais deux fois celle de A. Un $O(|A'|)$ est donc aussi un $O(|A|)$.
  - Ensuite, on remarque que la suppression d'un arc dans un graphe ne peut pas diminuer la distance entre deux points dans un graphe : en effet si un chemin existe dans le graphe amputé d'un arc, il existe aussi dans le graphe initial.
  - On voit aussi que la modification du graphe d'écart qu'occasionne une étape de l'algorithme équivaut à : * rajouter éventuellement quelques arcs (x,y) où d(s,y)=d(s,x) – 1,
  * et supprimer au moins un arc (x,y) où d(s,y) = d(s,x)+1.
- Montrons que la distance de s à p ne peut pas rester la même après plus de $|A'|$ étapes de chemins augmentant, et qu'au delà la distance entre s et p augmente strictement. Puisque la distance entre s et p est comprise entre 1 et $|X|-1$ si p est accessible depuis s, on aura le résultat.
  Tout d'abord il est clair que le rajout des arcs (x,y) où d(s,y) = d(s,x) – 1 ne modifie pas l'algorithme de BFS, car l'arc y aura déjà été visité quand on arrivera à x. Les valeurs de d(s,x) restent donc les mêmes après avoir rajouté ces arcs, et en particulier d(s,p) reste le même. De plus on a vu qu'enlever des arcs ne pouvait pas diminuer la distance séparant deux sommets. Donc la distance d(s,p) est croissante.
  Pour des raisons de temps nous ne pouvons finir cette démonstration.

1.5) Autres algorithmes existants pour le flot maximal

Il existe un autre algorithme pour la recherche de flot maximum, créé par Goldberg et Tarjan.
Il fait progresser le flot en poussant le flot, et non par recherches successives de chemin dans le graphe des écarts comme dans FF.
Il existe aussi une autre amélioration de FF que celle proposée par Edmond et Karp, qui a été inventée par Dinic.

2) Flot de coût minimum

2.1) Définition et propriétés préalables

- Définition : Soit (X,A) un graphe orienté sans boucles, u une valuation (les capacités) des arcs, c une autre valuation (les coûts unitaire de passage) des arcs, et s et p sont deux sommets distincts de X. On peut imposer aux coûts d'être positifs, mais pas nécéssairement.
- Coût d'un flot : on appelle coût d'un flot f le nombre c(f) = $\sum_{a \in A} c(a).f(a)$
  Un coût négatif est équivalent à un gain.
- Alors pour une flot de valeur donnée, 0<=v<=v_max, on veut connaitre le flot qui est le moins cher, ie dont le coût est minimal, parmi tous les flots de valeur v.

2.2) Cycles diminuants

Un cycle diminuant est un cycle du graphe des écarts dont le coût est négatif.
Il est clair qu'un flot qui dispose d'un tel cycle est améliorable : il suffit d'augmenter au maximum le flot suivant ce cycle, et on dépense alors moins d'argent, et le flot reste un flot. Donc un flot qui a un circuit diminuant n'est pas de coût minimum.
De plus, un circuit qui n'a pas de tels circuit est forcément de coût minimum. En effet, supposons qu'il existe un flot de même valeur et de coût strictement inférieur, et montrons alors qu'il existe un cycle diminuant dans le graphe.
Le flot différence du moins cher moins le plus cher est de coût négatif et est une circulation (tous les sommets même s et p ont autant de flot rentrant que sortant).
On peut prouver facilement que toute circulation peut se décomposer en cycles élémentaires ayant un coût donné, en en cherchant un après l'autre et en supprimant ainsi du graphe au moins un arc. Alors un de ces cycles a forcément un coût strictement négatif, et donc cela prouve qu'on peut ajouter ce cycle au flot de départ : il comporte bien un cycle diminuant.
On obtient ainsi un algorithme pour trouver un flot de coût minimum : partir d'un flot de la valeur désirée, et trouver un cycle diminuant, et modifier le graphe des écarts en fonction de celui-ci. Continuer ainsi jusqu'à ce qu'il n'y ait plus de cycle diminuant.

2.3) Méthode "à la FF"

En se servant du lemme de la partie 2.2, ie qu'un flot de valeur fixée est de coût minimum ssi il n'existe pas de cycles diminuant dans le graphe des écarts, on peut démontrer qu'un algorithme semblable à Ford et Fulkerson, mais recherchant les chemins les plus courts possibles de la source au puit, fonctionne. Pour des raisons de temps nous ne détaillons pas la démonstration.

2.4) Autres algorithmes existants

Il existe une autre méthode : la méthode du simplexe.

3) Exemples d'utilisation

Quelques exemples d'utilisations, pour lesquels il existe parfois des algorithmes spécifiques plus efficaces que ceux présentés. Pour des raisons de temps nous n'avons pas pu détailler les exemples donnés.

3.1) Cas des graphes bipartis

3.1.1) Couplages

3.1.2)Assignements de travail, ou de salle de classe

Variation : bottleneck problem.

3.1.3) Transports par bateau de coût minimum et respectant les capacités des cargos

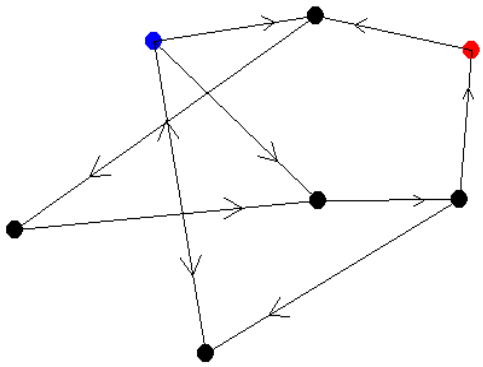
3.2) Problèmes dans des graphes quelconques

3.2.1) Circuits électriques

3.2.2) Circulation d'eau

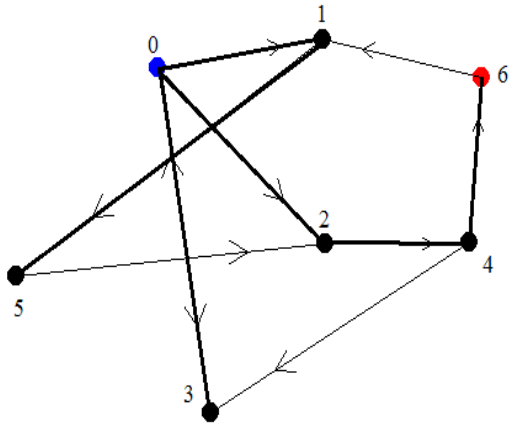
3.2.3) Problème du postier

4) Application des algorithmes pour mieux les comprendre



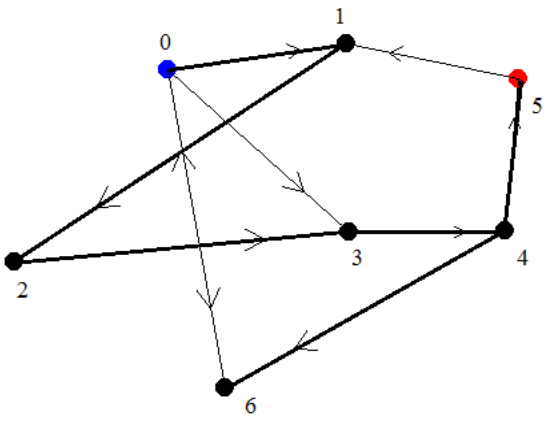
Graphe 1 (non valué)

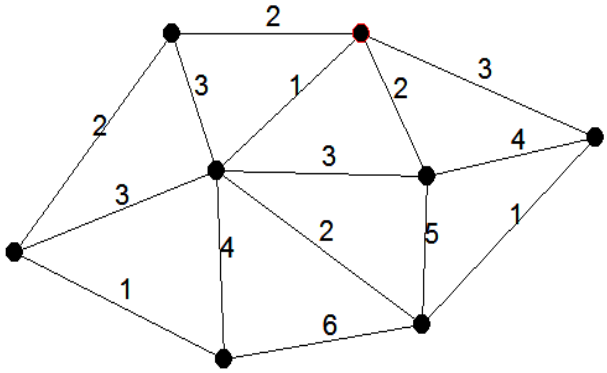
4.1) Algorithme de parcours en largeur



Le sommet de départ est en bleu, celui d'arrivée en rouge.
Les sommets sont rencontrés dans l'ordre numéroté avec le parcours en largeur (BFS).

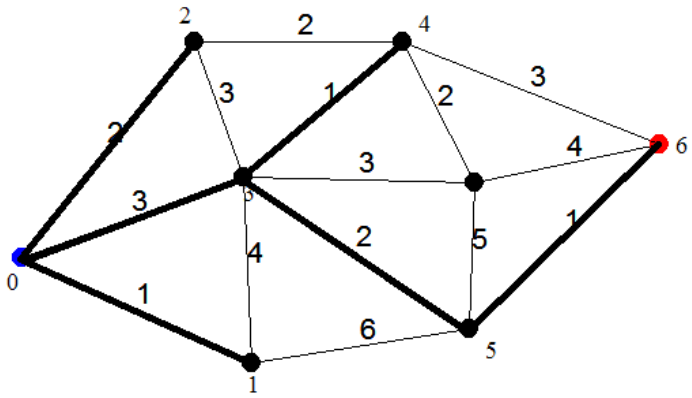
4.1) Algorithme de parcours en profondeur





Voici un graphe valué pour donner un exemple des algorithmes de Djikstra et de Bellman.

4.3) Algorithme de Djikstra

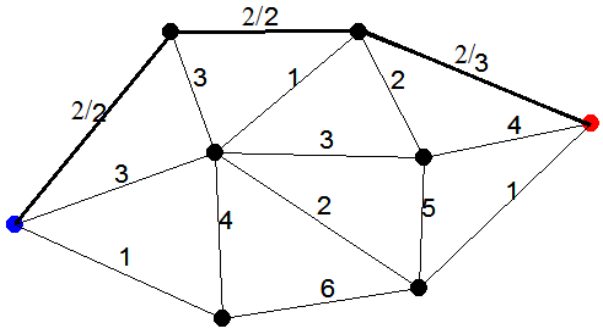


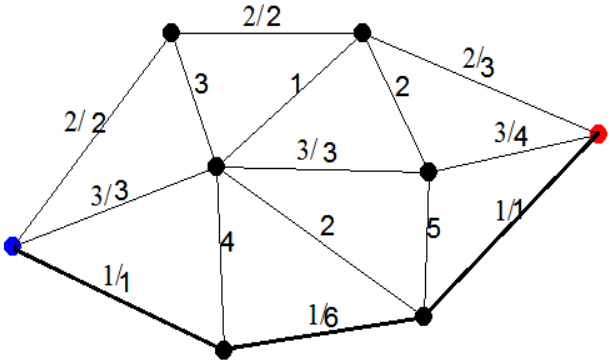
4.4) Algorithme de Bellman

Sommet --> étapes\	1	2	3	4	5	6	7
1	1	2	3	infini	infini	infini	infini
2	1	2	3	4	5	infini	6
3	1	2	3	4	5	6	6

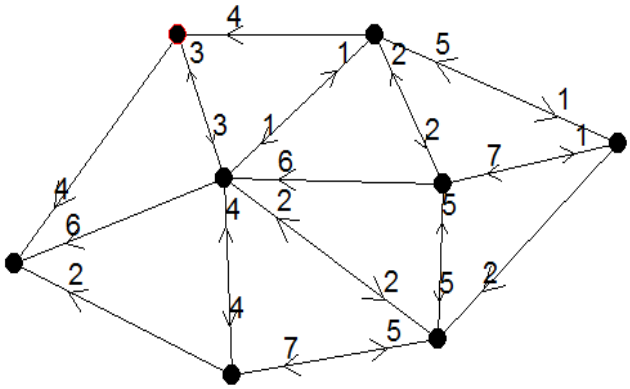
4.5)Algorithme de Ford et Fulkerson

Supposons maintenant que le graphe précédent représente des capacités.
Alors voici 3 étapes possibles de l'algorithme de FF :





Et le graphe d'écart final est :



4.6) Algorithme de Ford et Fulkerson avec coûts

Il faut faire comme Ford et Fulkerson, mais en choisissant des chemins les plus courts possible avec l'algorithme de Bellman.

Annexe

A1) Définition et notation de base de théorie des graphes

Définition de base

Un **graphe orienté G = (X,A)** (noté aussi (S,A), ou (X,U), ou (S,U), ou (N,A) et souvent noté (V,E) en anglais) est constitué :

- d'**un ensemble X de points** (ou sommets, ou vertices ou node en anglais)
- d'**un ensemble A d'arcs** (ou flèches, ou traits ou arêtes si le graphe n'est pas orienté, ou directed edge ou link en anglais).

Sortes de graphes

- Valuation(s) ou + généralement étiquettes : Oui : Graphe valué ou étiquetté. Existence de fonction(s) de X ou/et de A dans un ensemble de nombres, souvent $\mathbb{R}^+$, $\mathbb{N}$ avec parfois aussi $+\infty$, et parfois [0;1]. Elles peuvent représenter des longueurs de routes, des capacités de tuyaux, ou des degré de fiabilité par exemple. Parfois on associe des ensembles de lettres aux arcs : c'est le cas des automates / Non : graphe.
- Orientation : Oui $A \subseteq X^2$ (digraph en anglais) / Non $a \in A \Rightarrow (a \in P(X) tq |a| = 2)$
- Possibilité d'arcs d'un sommet vers lui même (boucle) : Oui : pseudo-graphe / Non : graphe simple
- Possibilité de plusieurs arcs reliant deux villes : Oui : hypergraphes (parfois aussi appelés multigraphes) / Non : graphe.
- Classe d'isomorphisme d'un graphe : Seul la structure du graphe importe, pas le nom de ses sommets.

Définitions de base

- Chemin : Suite d'arcs se touchant l'un après l'autre (déf. de "se touchant" variant comme on le pense selon le cas où G est orienté ou pas, le premier cas étant le plus restrictif).
- Cycle : chemin qui commence et finit sur un même point.
- Degré d'un sommet: nombre d'arcs qui l'ont pour extrémité.
  On distingue degré entrant et sortant si le graphe est orienté.
- Sommets fils : L'ensemble des fils de x est noté $\Gamma(x) = \{y \mid (x,y) \in A\}$
- Sommets descendants de degré k : l'ensemble des descendants de degré k est défini par récurrence : $\Gamma_{k+1}(x) = \Gamma(\Gamma_k(x))$
- L'ensemble des descendants de x est noté : $\hat{\Gamma}(x) = \bigcup_{k \in \mathbb{N}} \Gamma_k(x)$
- Pour noter les parents de x, on utilise $\Gamma^{-1}(x)$. De même pour Γ^{-k} et $\hat{\Gamma^{-1}}$.

Caractéristiques de graphes

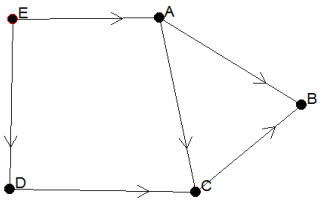
- Densité : Oui $|A| \approx |X|^2$ / Non $|A| \approx |X|$
- Connexité : Existe-t-il un chemin entre deux sommets quelconques ?

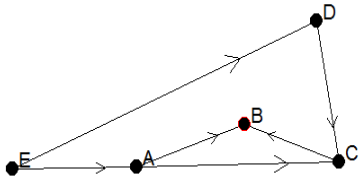
Appelée forte connexité si G est orienté.

- Existence de cycles : Oui : pas de nom spécifique / Non : si non orienté, et si graphe connexe, arbre. Si graphe non connexe, forêt. Si orienté, graphe progressivement fini : il représente une relation d'ordre.
- Fini : Oui : graphe normal / Non : graphe infini.
- Complet : Les sommets sont tous reliés entre eux. Noté Kn si |X| = n.
- Clique de G : sous-graphe complet de G.
- **Biparti** : Si on peut partitionner X en deux ensembles disjoints S et T, tels qu'aucun arcs ne soit interne à un de ces ensembles.
- Régulier de degré k : ssi $\forall x \in X$, $deg\, x = k$
- Eulérien : un graphe G non-orienté est eulérien ssi il existe un chemin de G passant par tous les arcs une seule fois.
  Ssi le nombre de sommets de degré impair est 0 ou 2.
- Hamiltonien : G non orienté est hamiltonien ssi il existe un chemin de G passant par tous les points 1 seule fois.

Représentation de graphes orientés simples

- Formelle : X={A,B,C,D,E} et Arcs={ (E,A), (E,D), (D,C), (A,C), (A,B), (C,B) }
- Dessins (une infinité possible) :





- Matrice d'adjacence avec des 0 sur la diagonale (graphe simple):

0	1	1	0	0
0	0	0	0	0
0	1	0	0	0
0	0	1	0	0
1	0	0	1	0

- Vecteur de listes d'adjacences : [| A : [B,C] ; B : [] ; C : [B] ; D: [C] ; E : [A,D] |]. Economise de la place pour les graphes peu denses par rapport à la matrice qui gaspille beaucoup de cases remplies de 0. Par contre accessibilité linéaire au lieu d'en temps constant si le graphe est dense.
- Utilisation d'arbre binaire de recherche AVL à la place des listes : utile uniquement pour les graphes denses, pour savoir plus rapidement si un sommet j est successeur d'un sommet i.
- Représentation en caml : voir Annexe 5.

A2) Code source, partie algorithmique

A2.1) Typages

A2.1.1) Nombres

Pour que les capacités puisse être des entiers (int) ou des flottants (float) de manière indifférente, sans avoir besoin de coder les algorithmes pour les entiers et pour les flottants de manière indépendante, j'ai eu l'idée de créer un type "nombre" permettant de manipuler ensemble des entiers et des flottants. Ce type permet aussi de coder les nombres $+\infty$, $-\infty$, et "indefini".

```
let intof = int_of_float and floatof = float_of_int;;

type nombre = ent of int |flo of float |omega |omega_m |indefini;;
(*|fra of (int*int);; qu'on utilise pas pour éviter les complications, dues au risque
de
division par zéro. L'avantage n'est pas vraiment suffisant par rapport à la
complication*)
(*Alternative à omega et omega_m : omega of bool, si true -> plus l'infini, sinon
moins*)

let est_nul a = match a with
|ent 0->true|flo 0.->true |_->false;;
let est_infini a = a = omega || a=omega_m;;

let rec prefix +& a b = match a,b with
|ent(i),ent(j)->ent(i+j)
|ent i,flo j-> flo(floatof i +. j)
|flo i,flo j-> flo(i+.j)
|omega,omega_m->indefini
|omega_m,omega->indefini
|_,omega-> omega
|_,omega_m-> omega_m
|_,indefini->indefini
|_->b+&a;;

let moins_nombre a = match a with
|ent a -> ent (-a)
|flo a -> flo (0.-.a)
|omega-> omega_m
|omega_m->omega
|indefini->indefini;;
let prefix -& a b = a+&(moins_nombre b);;

let positif a = match a with
ent k->k>=0
|flo k->k>=0.
|omega->true
|omega_m->false
|indefini->false(*ou true ?*);;
let prefix >=& a b = positif (a-&b);;
let prefix <=& a b = positif (b-&a);;
let prefix >& a b = if b = omega || a=omega_m then false else not a<=& b;; (*positif
(a-&b) && not vaut_zero(a-&b)*)
let prefix <& a b = if a = omega || b = omega_m then false else not a>=& b;;

let est_environ_nul precision a =
if a <& precision && a >& moins_nombre precision then true else false;;
```

```
let mini a b = if a<=&b then a else b;;
let maxi a b = if a>=&b then a else b;;

let rec prefix *& a b = match a,b with
|ent(i),ent(j)->ent(i*j)
|flo i,flo j-> flo(i*.j)
|ent i,flo j-> flo(floatof i *. j)
|j,k when est_nul j && est_infini k->indefini
|(u,k) when k>&ent 0 && est_infini u-> u
|u,k when k<&ent 0 && est_infini u-> moins_nombre u
|_,indefini->indefini
|_-> b*&a;;

let inverse a =
match a with
|ent a-> if a<>0 then flo (1./. floatof a) else indefini
|flo a-> if a<>0. then flo (1./.a) else indefini
|x when est_infini x-> ent 0
|_->indefini;;
let prefix /& a b = a*&(inverse b);;
```

A2.1.2) Listes LIFO (files) et FIFO (piles)

Les listes LIFO et FIFO sont utilisées pour les algorithmes de recherche de chemin BFS et DFS.

```
(*def_file*)

type 'a file ={mutable a: ('a list) ; mutable b: 'a list};;

let creer_file() ={a=[];b=[]};;
let est_vide_f f = f = {a=[];b=[]};;
let enfiler f r = f.b<-r::f.b;;
let inverse l =
let rec aux acc l = match l with [] -> acc
|_-> aux ((hd l) :: acc) (tl l) in aux [] l;;
let reactu f = f.a<- (*a@*)inverse f.b;f.b<-[];;
(*push*)
let rec defiler f = match f.a,f.b with
|[],[]->failwith "file vide"
|[],_-> reactu f; defiler f
|_->let k = hd f.a in f.a<-tl (f.a); k;;

(*-----------------*)

type 'a pile = {mutable c:'a list};;
(*plus pratique à utiliser que les ref list parfois *)

let creer_pile() = {c=[]};;
let est_vide_p p = p = {c=[]};;
let empiler p a = p.c<-a::p.c;;
let depiler p = if p.c = [] then failwith "pile vide"
else let k = hd p.c in p.c<-tl p.c;k;;
```

A2.1.3) Graphes (doublement valués)

Nous avons codé les graphes doublement valués avec le type "(int * (nombre vect)) list vect" :

```
type 'a graphe ==((int* 'a) list) vect;;
include "biblio2/TIPE/Programmes/Types/def_nombre";;
```

```
(*type graphe_nomb ==(int * (nombre vect)) list vect;;*)
#open "graphics";;

type dessin_nomb = {mutable tai: int; mutable arc : ((int * (nombre vect)) list) vect;
mutable pos : (int*int) vect;mutable nom : string vect;mutable col : color vect;};;

let gra_of_des d = d.arc;;
```

Nous avons aussi pensé à utiliser un type « étiquette », mais le codage aurait pu être plus lourd. Voici le type qui aurait été utilisé :

```
(* type etiquette = {mutable cap : nombre ; mutable dis : nombre };;
type graphe_nomb_avecetiq = etiquette graphe;; *)
```

A2.1.4) Arbre binaire de recherche AVL

Les arbres AVL peuvent être utilisés pour coder les graphes "(int* étiquette) avl vect" (ou multigraphes).
En particulier nous avons utilisé des graphes doublement valués, donc l'étiquette d'un arc était un "nombre vect", ce qui donne le type de graphe "(int*(nombre vect)) avl vect", un avl du vecteur principal représentant les sommets accessibles depuis le sommet k, ainsi que les étiquettes associées.
Cette manière de coder un graphe est en particulier utile pour des graphes denses : le coût d'accès à un sommet est ainsi considérablement réduit (en $O(\log(|\Gamma(x)|))$ au lieu de $O(|\Gamma(x)|)$ pour un sommet x donné). Cependant pour les graphes peu denses, ie pour lesquels, en moyenne, $|A| \ll |X|^2$, et donc $|\Gamma(x)| \ll X$ pour la plupart des sommets x, et donc log (|gamma (x)|) n'est pas très inférieure à |gamma(x)|.

- Nous avons d' abord codé des fonctions préliminaires, de rotation en particulier :

```
include "biblio2/TIPE/programmes/types/def_nombre";;

type balance == int;;
type 'a avl = V | N of (('a * balance) * ('a avl) * ('a avl));;
type avl_ici == (int * (nombre vect)) avl ;;

let inférieur a b = fst a<fst b || (fst a=fst b && (snd a).(1)<&(snd b).(1) );;
let pareil a b = fst a = fst b && (snd a).(1)=(snd b).(1) ;;
let supérieur a b = fst a>fst b || (fst a=fst b && (snd a).(1)>&(snd b).(1) );;

let inférieur2 a b = fst a<fst b ;;
let pareil2 a b =fst a = fst b;;
let supérieur2 a b = fst a>fst b;;

let rot_d z = match z with N (q, N (p, a, b), c) -> N (p, a, N (q, b, c))
    | _ -> V;;

let rot_g z =match z with N (q, c, N (p, a, b)) -> N (p, N (q, c, a), b)
    | _ -> V;;

let rot_gd z=match z with N(a,b,c)->rot_d(N(a,rot_g b,c))
|_->V;;

let rot_dg z= match z with N(a,b,c)->rot_g(N(a,b,rot_d c))
|_->V;;

let equil_d a =
match a with N ((b, 2), d, e) ->begin
```

```
match d with
| N ((f, 1), h, i) -> (rot_d (N ((b, 0), N ((f, 0), h, i), e))), (- 1)
| N ((f, 0), h, i) -> (rot_d (N ((b, 1), N ((f, (- 1)), h, i), e))), 0
| N ((f, - 1), h, i) ->
begin
match i with
| N ((j, - 1), l, m) -> (rot_gd (N ((b, 0), N ((f, 1), h, N ((j, 0), l, m)), e))), (-
1)
| N ((j, 0), l, m) -> (rot_gd (N ((b, 0), N ((f, 0), h, N ((j, 0), l, m)), e))), (- 1)
| N ((j, 1), l, m) -> (rot_gd (N ((b, -1), N ((f, 0), h, N ((j, 0), l, m)), e))), (- 1)
end
end;;


let equil_g a =
  match a with N ((b, -2), d, e) ->
        begin
          match e with
            | N ((f, -1), h, i) -> (rot_g (N ((b, 0), d, N ((f, 0),     h, i)))), (- 1)
            | N ((f, 0), h, i) ->  (rot_g (N ((b, 1), d, N ((f, (- 1)), h, i)))), 0
            | N ((f, 1), h, i) ->
                begin
                  match h with
                    | N ((j, 1), l, m) ->(rot_dg (N ((b, 0), d, N ((f,-1), N ((j, 0),
l, m),i)))), (- 1)
                    | N ((j, 0), l, m) ->(rot_dg (N ((b, 0), d, N ((f, 0), N ((j, 0),
l, m),i)))), (- 1)
                    | N ((j,-1), l, m) ->(rot_dg (N ((b, 1), d, N ((f, 0), N ((j, 0),
l, m),i)))), (- 1)
                end
        end;;

(*-----------------------------*)

let rec cherche_petit a =
  match a with V -> failwith "arbre vide"
    | N (b, V, d) -> (d, fst b, - 1)
    | N (b, c, d) -> let arb, el, varprof = cherche_petit c in
          let e = snd b + varprof in
            let b2 = fst b, e in
              if varprof = - 1 && snd b = - 1 then
                let f, g = equil_g (N (b2, arb, d)) in (f, el, g)
              else if varprof = - 1 && snd b = 1 then ((N (b2, arb, d)), el, - 1)
              else ((N (b2, arb, d)), el, 0);;
```

- Ensuite, nous avons codé l'algorithme de recherche d'un élément :

```
let recherche_avl a k =(*k est le numéro du snd sommet, plus le vect des capa et des
dist*)
  let rec aux a =
    match a with V -> false
      | N (u, w, x) -> if pareil k (fst u) then true
          else if supérieur k (fst u) then aux x
          else aux w in aux a;;
```

- Nous avons aussi codé l'algorithme de suppression d'un élément :

```
let rec enleve_aux_avl a b =
  match a with
V -> V, 0,(-1,[||])
| N (c, z, V) when pareil2 b (fst c) -> z, - 1,(fst c)
```

```
| N (d, e, f) when inférieur2 b (fst d) ->
   let g, h,z = enleve_aux_avl e b in
   let i = snd d + h in
   let d2 = fst d, i in
     if snd d = 1 && h = - 1 then N (d2, g, f), - 1, z
     else if i > - 2 then N (d2, g, f), 0, z
     else (*if i =-2 then*) let u,v= equil_g (N (d2, g, f)) in u,v,z
| N (d, e, f) when supérieur2 b (fst d) ->
   let g, h,z = enleve_aux_avl f b in
   let i = snd d - h in
   let d2 = fst d, i in
     if snd d = - 1 && h = - 1 then N (d2, e, g), - 1,z
     else if i < 2 then N (d2, e, g), 0,z
     else (*if i =2 then*)let u,v = equil_d (N (d2, e, g)) in u,v,z
| N (d, e, f) when pareil2 b (fst d) ->
   let arb, el, varprof = cherche_petit f in
   let j = snd d - varprof in
   let d2 = el, j in
     if j < 2 && snd d >= 0 then N (d2, e, arb), 0,(fst d)
     else if j < 2 && snd d = - 1 then N (d2, e, arb), varprof,(fst d)
     else (*if j = 2 then*) let u,v = equil_d (N (d2, e, arb)) in u,v,fst d;;

let enleve_avl a b = let c,d,e = enleve_aux_avl a b in c,e;;
```

- Nous avons enfin codé l'algorithme d'insertion d'un élément dans l'avl.

En particulier la différence avec un algorithme d'insertion habituel dans un avl est que si un élément de l'avl de la forme (int * [| cap ; coût |]) est inséré, est que l'avl contient déjà un élément ayant le même entier et le même coût de passage, on ajoute les coûts (et si le coût obtenu est négatif ou nul, on supprime l'élément correspondant).
Cependant si le coût est différent (pour un même entier) , on ajoutera le sommet en utilisant l'ordre lexicographique : l'avl peut donc coder des multigraphes.
Si on voulait se limiter aux graphes, il faudrait utiliser une procédure d'insertion qui remplacerait l'élément (int*étiquette 1) par le nouvel élément (int*étiquette 2) au lieu de les garder tous les deux.
Cependant le choix de coder les multigraphes se justifie, car on peut ainsi prendre en compte des situations impossible sinon pour le cas des flots de coûts minimum.

```
let rec aux_insere_avl a k = (*renvoie l'arbre, et la différence de profondeur avec
avant*)
  match a with
| V -> N ((k, 0), V, V), 1
| N (b, c, d) ->
if pareil k (fst b) then begin
   let cap = (snd (fst b)).(0) +& (snd k).(0) in
   let b2 = ((fst (fst b), [|cap; (snd (fst b)).(1)|]), snd b) in
     if cap >& ent 0 then (N (b2, c, d), 0)
     else let z,y,x = enleve_aux_avl a (fst b) in z,y end
else if inférieur k (fst b) then begin
   let (t, s) = aux_insere_avl c k (*s=1 ou 0*) in
   let h = (snd b) + s in let b2 = (fst b, h) in
     if snd b >= 0 then begin if h < 2 then ((N (b2, t, d)), s)
     else let (p, q) = (equil_d (N (b2, t, d))) in (p, q + s) end
     else N (b2, t, d), 0
end
else begin
   let (t, s) = aux_insere_avl d k in
   let h = (snd b - s) in let b2 = (fst b, h) in
     if snd b <= 0 then begin
       if h > (- 2) then ((N (b2, c, t)), s)
       else let (p, q) = (equil_g (N (b2, c, t))) in (p, q + s) end
     else N (b2, c, t), 0 end;;
```

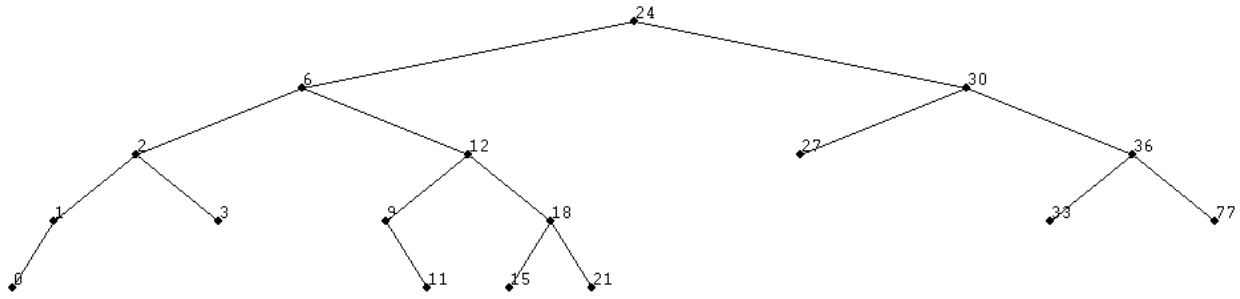
```
let rec insere_avl a k = let (p,m) = aux_insere_avl a k in p;;
```

• Finalement, nous avons codé une fonction permettant de visualiser un arbre binaire de recherche (uniquement les numéros des nœuds, pas les étiquettes). Ceci a permis en particulier de contrôler le bon fonctionnement des algorithmes sur des cas particuliers.

```
#open "graphics";;

let rec mettre l a =
match l with
[]->a
|_-> mettre (tl l) (insere_avl a ((hd l),[|ent 1 ;ent 1|]));;

let rec representer_avl1 a k = match a with V -> ()
    | N (p, q, s) -> let r = current_point () in fill_circle (fst r) (snd r) 3;
          draw_string (string_of_int (fst (fst p)));
          moveto (fst r) (snd r);
          if q <> V then begin lineto ((fst r) - k) ((snd r) - 50) end;
          representer_avl1 q (k / 2);
          moveto (fst r) (snd r);
          if s <> V then begin lineto ((fst r) + k) ((snd r) - 50) end;
          representer_avl1 s (k / 2);;

let representer_avl a = open_graph "1000x800"; clear_graph();moveto 500 750;
representer_avl1 a 250;;
```

Voici la représentation de l'avl contenant les éléments suivants :

```
representer_avl ((mettre [3;6;9;12;15;18;21;24;27;30;33;36;77;11;2;1;0] V));;
```



A2.2) Algorithmes de recherche de chemin

Il est nécessaire de disposer d'algorithmes de recherche de chemin pour coder FF.
Nous les avons donc codé.

A2.2.1) Algorithmes préliminaires

Pour avoir les chemins présentés de façon simple à utiliser, ie de la forme (x1,x2,...,xn) pour la suite des arcs (x1,x2),(x2,x3)...,x(n-1),xn, il nous faut une fonction qui prend la forme des chemins où chaque sommet est étiqueté par celui qui l'a atteint, et qui la simplifie :

```
let chemin v b = (*renvoie [a...b] si possible, [b] sinon*)
  let rec aux l =
```

```
    match l with
      | x :: y -> if v.(x) = (- 2) || v.(x) = (-1) then l
            (*<- y si on veut mettre [] au lieu de b et [a+1;...;b] sinon*)
          else aux (v.(x) :: l) | _ -> [] in aux [b];;
```

De plus l'algorithme de Bellman ne permet pas de connaître la capacité du chemin trouvé au cours de l'algorithme, il faut donc le trouver dans la fonction « chemin ». h est un vecteur tq h.(j) contient la capcité de l'arc emprunté pour aller jusqu'à j.

```
let chemin2 v h b = (*renvoie [a...b], puis la cap min, si possible, [b] et (ent 0)
sinon*)
(*utiliser uniquement pour Bellman*)
  let rec aux l k=
    match l,k with
      | (x :: y),_ -> if v.(x) = (- 2) || v.(x) = (-1) then l,k
            (*<- y,k si on veut mettre [] au lieu de [b] et [a+1;...;b] sinon*)
          else aux (v.(x) :: l) (mini k h.(x)) | _ -> [], (ent 0) in aux [b] (h.(b));;
```

A2.2.2) BFS (parcours en largeur)

Nous avons codé 2 versions : PL_bout, qui calcule les chemins de a à tous les sommets qu'on peut atteindre deupis a, et une seconde version qui s'arrête dès que b est atteint.
Il en est de même pour les autres algorithme de recherche de chemin (sauf Bellman) : la version "bout" va jusqu'au bout au lieu de s'arrêter quand b est atteint.
On ne peut le faire avec Bellman car la condition de l'arrêt de l'algorithme est d'avoir calculé toute les distances d(a,x) (pour tous sommets x atteignables depuis a).

```
let PL_bout g n a =
  let v = make_vect n (- 2) and h = make_vect n omega in
    v.(a) <- (- 1);
    let f = creer_file() in
      enfiler f a;
      while not est_vide_f f do
        let t = defiler f in
          let l = ref g.(t) in
            while !l <> [] do
              let k = fst (hd !l) and r = (snd (hd !l)).(0) in l := tl !l;
                if v.(k) = (- 2) then begin v.(k) <- t; h.(k) <- mini h.(t) r; enfiler
f k end
            done done;v,h;;

let PL2 g n a b = let v,h=PL_bout g n a in (chemin v b, h.(b));;

let PL g n a b (*parcours en largeur, avec file*) =
  (*en particulier nombre d'arêtes minimum*)
  (*on aurait aussi pu utiliser djikstra avec des distances de 1*)
  let v = make_vect n (- 2) and h = make_vect n omega in
v.(a)<-(-1);
    let f = creer_file() in
      enfiler f a;
      while not est_vide_f f do
        let t = defiler f in
          let l = ref g.(t) in
            while !l <> [] do
              let k = fst (hd !l) and r = (snd (hd !l)).(0) in l := tl !l;
                if k = b then begin l := []; v.(b) <- t; h.(b) <- mini (h.(t)) r; f.a
<- []; f.b <- [] end
                else if v.(k) = (- 2) then begin v.(k) <- t; h.(k) <- mini h.(t) r;
enfiler f k end
```

```
            done done; chemin v b, h.(b);;
```

A2.2.3) DFS (parcours en profondeur)

```
let PP_bout g n a =
  let v = make_vect n (- 2) and h = make_vect n omega in
v.(a)<-(-1);
    let p = ref [a] in
      while !p <> [] do let t = hd !p in p := tl !p;
          let l = ref g.(t) in
            while !l <> [] do
              let k = fst (hd !l) and r = (snd (hd !l)).(0) in l := tl !l;
                  if v.(k) = (- 2) then
                  begin v.(k) <- t; h.(k) <- mini h.(t) r;
                    p := k :: !p; end;
            done; done; v,h;;

let PP2 g n a b = let (v,h) = PP_bout g n a in (chemin v b, h.(b));;


let PP g n a b (*version itérative avec pile*) = (*a<>b*)
  let v = make_vect n (- 2) and h = make_vect n omega in
    v.(a) <- (- 1);
    let p = ref [a] in
      while !p <> [] do let t = hd !p in p := tl !p;
          (*qui n'est pas b car a<>b et on stop dès que b est devant aperçu*)
          let l = ref g.(t) in
            while !l <> [] do
              let k = fst (hd !l) and r = (snd (hd !l)).(0) in l := tl !l;
                if k = b then begin l := []; v.(b) <- t; h.(b) <- mini (h.(t)) r; p :=
[] end
                else if v.(k) = (- 2) (*si pas encore vu, marquer*) then
                  begin v.(k) <- t; h.(k) <- mini h.(t) r;
                    p := k :: !p; end;
            done; done; chemin v b, h.(b);;
```

A2.2.4) Djikstra

```
let djikstra_bout g n a =
  let v = make_vect n (- 2) and G = copy_vect g and
  w = make_vect n omega and v2 = make_vect n false and
  h = make_vect n omega and minimum = ref (ent 0) and k = ref 1 and
  t = ref (- 1) in w.(a) <- ent 0; v.(a) <- (- 1); v2.(a) <- true;

    while G.(a) <> [] do let c, d = hd G.(a) in G.(a) <- tl G.(a);
        w.(c) <- d.(1); h.(c) <- d.(0);v.(c)<-a done;

    while !minimum <> omega && !k < n do
      minimum := omega;
      t := (- 1);
      for i = 0 to n - 1 do
        if v2.(i) = false && w.(i) <& !minimum then begin
            t := i; minimum := w.(i) end
      done;

      if !t <> (- 1) then begin
          v2.(!t) <- true;
          while G.(!t) <> [] do let x, y = hd G.(!t) in G.(!t) <- tl G.(!t);
              if not v2.(x) then begin
```

```
                if (y.(1) +& w.(!t)) <& w.(x) then begin
                    v.(x) <- !t;
                    w.(x) <- (y.(1) +& w.(!t));
                    h.(x) <- mini h.(x) h.(!t) end
            end;
          done;
        end;
        k := !k + 1;
    done;
    v, w, h;;

let djikstra2 g n a b =
let v,w,h = djikstra_bout g n a in (chemin v b, h.(b),w.(b));;

let djikstra g n a b =
  let v = make_vect n (- 2) and G = copy_vect g and
  w = make_vect n omega and v2 = make_vect n false and
  h = make_vect n omega and minimum = ref (ent 0) and k = ref 1 and
  t = ref (- 1) in w.(a) <- ent 0; v.(a) <- (- 1); v2.(a) <- true;

    while G.(a) <> [] do let c, d = hd G.(a) in G.(a) <- tl G.(a);
        w.(c) <- d.(1); h.(c) <- d.(0);v.(c)<-a done;

    while !minimum <> omega && !k < n && !t<> b do
      minimum := omega;
      t := (- 1);
      for i = 0 to n - 1 do
        if v2.(i) = false && w.(i) <& !minimum then begin
            t := i; minimum := w.(i) end
      done;

      if !t <> (- 1) then begin
          v2.(!t) <- true;
          while G.(!t) <> [] do let x, y = hd G.(!t) in G.(!t) <- tl G.(!t);
              if not v2.(x) then begin
                  if (y.(1) +& w.(!t)) <& w.(x) then begin
                      v.(x) <- !t;
                      w.(x) <- (y.(1) +& w.(!t));
                      h.(x) <- mini h.(x) h.(!t) end
              end;
          done;
        end;
        k := !k + 1;
    done;
    chemin v b, h.(b), w.(b);;
```

A2.2.5) Bellman

```
let bellman g n a b =
  let v = make_vect n (- 2) in(*D'où viennent les noeuds dans le chemin*) v.(b)<-(-1);
  let w = make_vect n omega in w.(b) <- ent 0;(*distances*)
  let h = make_vect n omega in
   let k = ref 1 (*nb de sommets rencontrés pour l'instant au minimum, au depart juste
b donc 1*)
    and stabilité = ref false in
    (*on calcule le chemin le plus court pour aller jusqu'à b en k pas*)
      while not (!stabilité || !k = n) do
        stabilité := true;
        for i = 0 to n - 1 do
```

```
          let l = ref g.(i) in
            while !l <> [] do
              let r, s = hd !l in l := tl !l;
                if w.(r) +& s.(1) <& w.(i) then begin
                    w.(i) <- w.(r) +& s.(1);
                    v.(i) <- r;
                    h.(i)<- s.(0);
                    stabilité := false end done done;
        k := !k + 1;
      done; let dist = w.(a) in
 let (chem1,cap)= chemin2 v h a in (*car bellman a été fait en partant de b*)
 let chem = rev chem1 in if chem <>[a] then
 (chem, cap, dist) else ([b],cap,dist);;
```

A2.3) Algorithmes de flots

Nous avons ensuite codé les algorithmes de recherche de flots.

A2.3.1) Fonctions préliminaires

Nous avons d'abord codé les algorithmes permettant de modifier les graphes comme il le faut après avoir trouvé un chemin augmentant :

```
let nb_vect_default =
(-1,[|ent 0;ent 0|]);;

let enlever_arc G i j =
  let rec enlever l l2 k =
    match l with |[] -> l2, k
      | a :: b -> if fst a = j then enlever b l2 a else
            enlever b (a :: l2) k in let t = enlever G.(i) [] nb_vect_default
    in G.(i) <- fst t; snd t;;

let augmenter G i j p =
  let a, v = enlever_arc G i j in
    if a = - 1 then G.(i) <- (j, [|p|]) :: G.(i)
    else (*if a = j*) G.(i) <- (j, [|v.(0) +& p|]) :: G.(i);;

let diminuer G i j p =
  let a, v = enlever_arc G i j in
  (*on suppose l'absence d'erreur, ie on suppose a = j*)
    let r = v.(0) -& p in if r >& ent 0 then G.(i) <- (j, [|r|]) :: G.(i);;

let rec tout_changer G l p = match l with
    | [] -> ()
    | [a] -> ()
    | a :: b :: c -> diminuer G a b p; augmenter G b a p; tout_changer G (tl l) p;;

let valeur_flot G a =
  let k = ref (ent 0) in
    let l = ref G.(a) in
      while !l <> [] do
        k := !k +& (snd (hd !l)).(0); l := tl !l
      done; !k;;

let flot_of_ecart n g_ec2 g_cap2 =
  let g_cap = copy_vect g_cap2
  and g_ec = copy_vect g_ec2 in
    for i = 0 to (n - 1) do
      let l = ref [] in
```

```
        while g_cap.(i) <> [] do
          let (j, capa) = hd g_cap.(i) in
            g_cap.(i) <- tl g_cap.(i);
            let (p, q) = enlever_arc g_ec i j in
              if p = (- 1) (*route pleine*) then l := (j, [|capa.(0); capa.(1)|]) :: !l
              else let r = capa.(0) -& q.(0) in
                  if r >& ent 0 && r <> indefini && r <> omega && r <> omega_m
                  then l := (j, [|r; capa.(1)|]) :: !l
                  else if capa.(0) = omega then let (p, q) = enlever_arc g_ec j i in
                      if p <> (- 1) then l := (j, [|q.(0); capa.(1)|]) :: !l
        done;
        g_cap.(i) <- !l done; g_cap;;

let simplifier G g n a b (*renvoie la partition de la coupe min avec juste ceux
accessibles
depuis b dans le graphe d'écart final, et puis renvoie la valeur totale du flot*)=
      let v = make_vect n false in
        let rec explorer x =
          if not v.(x) then begin v.(x) <- true;
              let l1 = ref G.(x) in while !l1 <> [] do
                  let u = fst (hd !l1) in l1 := tl !l1; explorer u done;
            end in explorer a; let r = flot_of_ecart n G g in
(v, valeur_flot r a, r, G );;
(*coupe, valeur_flot, graphe_flot, graphe_écarts*)

let string_of_list l=
let rec string_of_list2 l=
match l with
|[]->""
|[a]-> string_of_int a
|a::b-> string_of_int a ^ ";" ^ string_of_list2 b
in "["^string_of_list2 l^"]";;

(*-----------------------------------------------*)
(*Pour mincost :*)

let rec tout_changer3 G l p =
  match l with
    | [] -> ()
    | [a] -> ()
    | a :: b :: c ->
        let r, r2 = enlever_arc G a b and s, s2 = enlever_arc G b a in
            G.(b) <- (a, [|s2.(0) +& p; moins_nombre r2.(1)|]) :: G.(b);
            if r2.(0) >& p then G.(a) <- (b, [|r2.(0) -& p; r2.(1)|]) :: G.(a);
            tout_changer3 G (tl l) p;;
```

A2.3.2) Max flow

Nous avons ensuite codé l'algorithme de Ford et Fulkerson :

```
let maxflow (*fonc*) g a b (*e&k si fonc = PL, f&f si fonc = PP*) =
  let n = vect_length g in
    let G = copy_vect g and c = ref true in
      while !c do
        let (chem, cap) = (*fonc*) PL G n a b in if chem <> [b] && cap <> omega then
            begin tout_changer G chem cap; end
          else if chem <> [b] && cap = omega then begin
              failwith ("Flot infini passant par " ^ (string_of_list chem)) end;
          c := chem <> [b] done;simplifier G g n a b;;
```

A2.3.3) Mincost

Nous avons aussi codé les 3 versions du problème de flot de coût minimum :

```
let maxflow_mincost g a b =
  let n = vect_length g in
    let G = copy_vect g and c = ref true and cost = ref (ent 0) in
      while !c do
        let (chem, cap, dist) = bellman G n a b in
          if chem <> [b] && cap <> omega then
            begin tout_changer3 G chem cap; cost := !cost +& (cap *& dist); end
          else if chem <> [b] && cap = omega then begin
              failwith ("Flot infini passant par " ^ (string_of_list chem)); end;
          c := chem <> [b] done;
      !cost, (simplifier G g n a b);;

let mincost_flot_fixé (*fonc*) g a b flot =(*pour un flot fixé*)
  let n = vect_length g in
    let G = copy_vect g and c = ref true and f = ref (ent 0) and cost = ref (ent 0) in
      while !c do
        let (chem, cap, dist) = (*fonc*) bellman G n a b in
          if chem <> [b] && !f<&flot then begin
                  let augm = mini cap (flot -& !f) in
                  f := !f +& augm; tout_changer3 G chem augm;
                  cost := !cost +& (augm *& dist);
                   end
          else c:=false;
      done;
      (!cost, simplifier G g n a b);;

let mincost_coût_fixé g a b cout =
  let n = vect_length g in
    let G = copy_vect g and c = ref true and f = ref (ent 0) and cost = ref (ent 0) in
      while !c do
        let (chem, cap, dist) = (*fonc*) bellman G n a b in
          if chem <> [b] && !cost <& cout then begin
              let augm = mini (cap*&dist) (cout -& !cost) in
                f := !f +& (augm/&dist); tout_changer3 G chem (augm/&dist);
                cost := !cost +& (augm);
            end
          else c := false;
      done;
```

A2.3.4) Cas des graphes bipartis

Nous avons enfin codé un programme permettant de traiter facilement quelques problèmes de flots particuliers (couplages, assignement et transport) sur des graphes bipartis (mais ils utilisent la partie graphique également).

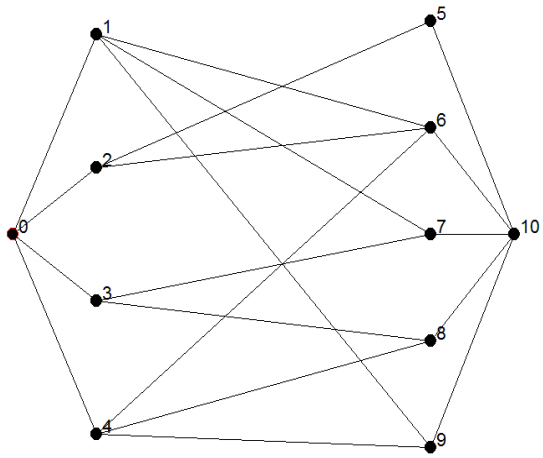
```
let preparer k n=
let v = make_vect (k+n+2) [] in
for i = 1   to k   do v.(0)<-(i,[|ent 1; ent 0|])::v.(0) done;
for j = k+1 to k+n do v.(j)<-[(k+n+1,[|ent 1; ent 0|])] done;v;;

let graphe_couplage_of_mat m =
let k = vect_length m and n = vect_length m.(0) in
let v = preparer k n in
for i = 1 to k do
for j = 1 to n do
if m.(i-1).(j-1) then v.(i)<- (k+j, [|ent 1; ent 0|]) ::v.(i)
done done;v;;
```

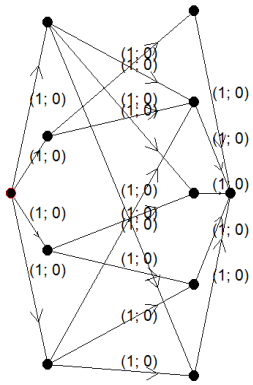
```
let graphe_maxflow_mincut_assignment m=
let k = vect_length m and n = vect_length m.(0) in
let v = preparer k n in
for i = 1 to k do
for j = 1 to n do
if m.(i-1).(j-1) >= 0 (*si <0, signifie non relié*)
 then v.(i)<- (k+j, [|ent 1; ent m.(i-1).(j-1)|]) ::v.(i)
done done;v;;

let preparer_transport k n v1 v2 =
let v = make_vect (k+n+2) [] in
for i = 1   to k   do v.(0)<-(i,[|ent v1.(i-1); ent 0|])::v.(0) done;
for j = k+1 to k+n do v.(j)<-[(k+n+1,[|ent v2.(j-k-1); ent 0|])] done;v;;

let graphe_maxflow_min_cut_transport v1 v2 m=
let k = vect_length m and n = vect_length m.(0) in
let v = preparer_transport k n v1 v2 in
for i = 1 to k do
for j = 1 to n do
if m.(i-1).(j-1) >= 0 then
v.(i)<- ( k+j, [|omega; ent m.(i-1).(j-1) |] ) ::v.(i)
done done;v;;

let graphe_maxflow_min_cut_transport_avec_capa v1 v2 m=
let k = vect_length m and n = vect_length m.(0) in
let v = preparer_transport k n v1 v2 in
for i = 1 to k do
for j = 1 to n do
if snd m.(i-1).(j-1) >= 0 then
v.(i)<- ( k+j, [|ent (fst m.(i-1).(j-1));
ent (snd m.(i-1).(j-1) ) |] ) ::v.(i)
done done;v;;

(*---------------------------------*)

let faire_pos k n = let r = 600 / k and s = 600 / n and
  v = make_vect (k + n + 2) (100, 300) in
    v.(k + n + 1) <- (700, 300);
    for i = 1 to k do v.(i) <- (200, 600- (r * i) + (r/2)) done;
    for j = 1 to n do v.(k + j) <- (600, 600 -(s * j)+ (s/2)) done; v;;

let faire_biparti g k n= let v = faire_pos k n in
let dessin = des_of_gra2 g v in
menu_principal dessin;;

(*---------------------------------*)

let couplage_of_flot g k n =
  (*détruit g sous sa forme originelle
 mais on vient de le sauvegarder donc ce n'est pas grave*)
  let m = make_matrix k n false in
    for i = 1 to k do
      while g.(i) <> [] do
        let j, v = hd g.(i) in g.(i) <- tl g.(i);
          m.(i - 1).(j - k - 1) <- true done; done; m;;

let transport_of_flot g k n=
let m = make_matrix k n (ent 0) in
    for i = 1 to k do
      while g.(i) <> [] do
        let j, v = hd g.(i) in g.(i) <- tl g.(i);
```

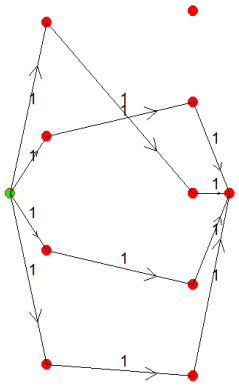
```
          m.(i - 1).(j - k-1) <- v.(0) done; done; m;;

(*--------------------------------*)

let ouvrir_couplage m = let k = vect_length m and n = vect_length m.(0) in
    let g = graphe_couplage_of_mat m in
      let coût,(coupe,valeur_flot,g_flot,g_écart) = faire_biparti g k n in
valeur_flot, couplage_of_flot g_flot k n;;

let ouvrir_assignment m = let k = vect_length m and n = vect_length m.(0) in
let g = graphe_maxflow_mincut_assignment m in
let coût,(coupe,valeur_flot,g_flot,g_écart) = faire_biparti g k n in
coût, valeur_flot, couplage_of_flot g_flot k n;;

let ouvrir_transport v1 v2 m=
let  k = vect_length m and n = vect_length m.(0) in
let g = graphe_maxflow_min_cut_transport v1 v2 m in
let coût,(coupe,valeur_flot,g_flot,g_écart) = faire_biparti g k n in
coût, valeur_flot, transport_of_flot g_flot k n;;

let ouvrir_transport_avec_capa v1 v2 m =
let  k = vect_length m and n = vect_length m.(0) in
let g = graphe_maxflow_min_cut_transport_avec_capa v1 v2 m in
let coût,(coupe,valeur_flot,g_flot,g_écart) = faire_biparti g k n in
coût, valeur_flot, transport_of_flot g_flot k n;;
```

Voici 4 exemples, ainsi que les copies d'écran associées (à gauche) et les copies d'écran des solutions (à droite) :

- ouvrir_couplage [|
[|false;true;true;false;true|];
[|true;true;false;false;false|];
[|false;false;true;true;false|];
[|false;true;false;true;true|]
|];;

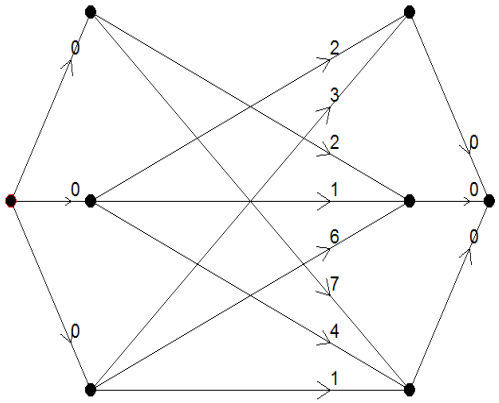


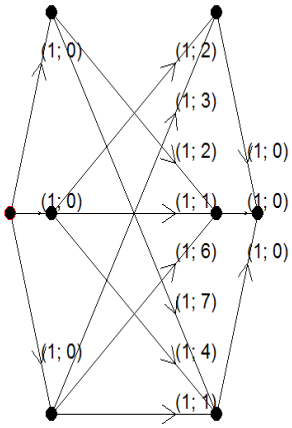


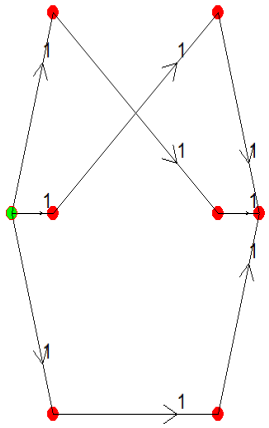


- ouvrir_assignment [|
[|-1;2;7|];
[|2;1;4|];
[|3;6;1|]

```
|];;
```

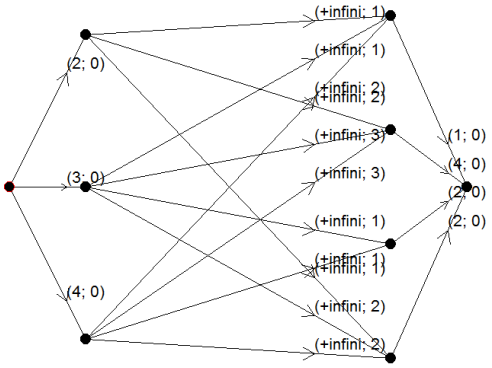


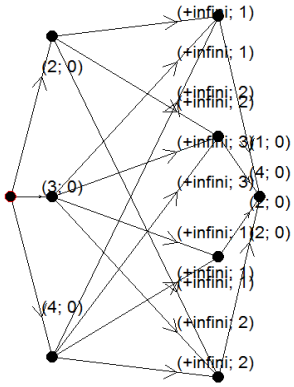


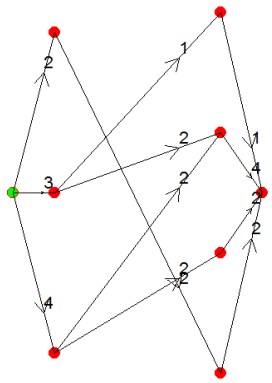


- ouvrir_transport

```
[|2;3;4|]
[|1;4;2;2|]
[|
[|1;2;-1;1|];
[|1;3;1;2|];
[|2;3;1;2|]
|];;
```

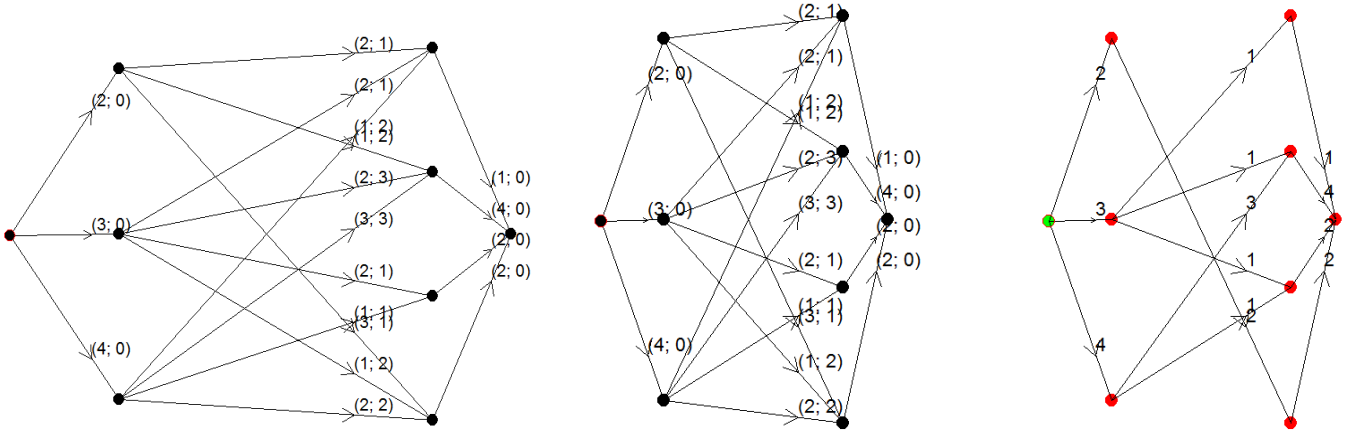






- ouvrir_transport_avec_capa

```
[|2;3;4|]
[|1;4;2;2|]
[|
[|2,1;1,2;0,-1;3,1|];
[|2,1;2,3;2,1;1,2|];
[|1,2;3,3;1,1;2,2|]
|];;
```



A3) Partie graphique, code source et copies d'écran

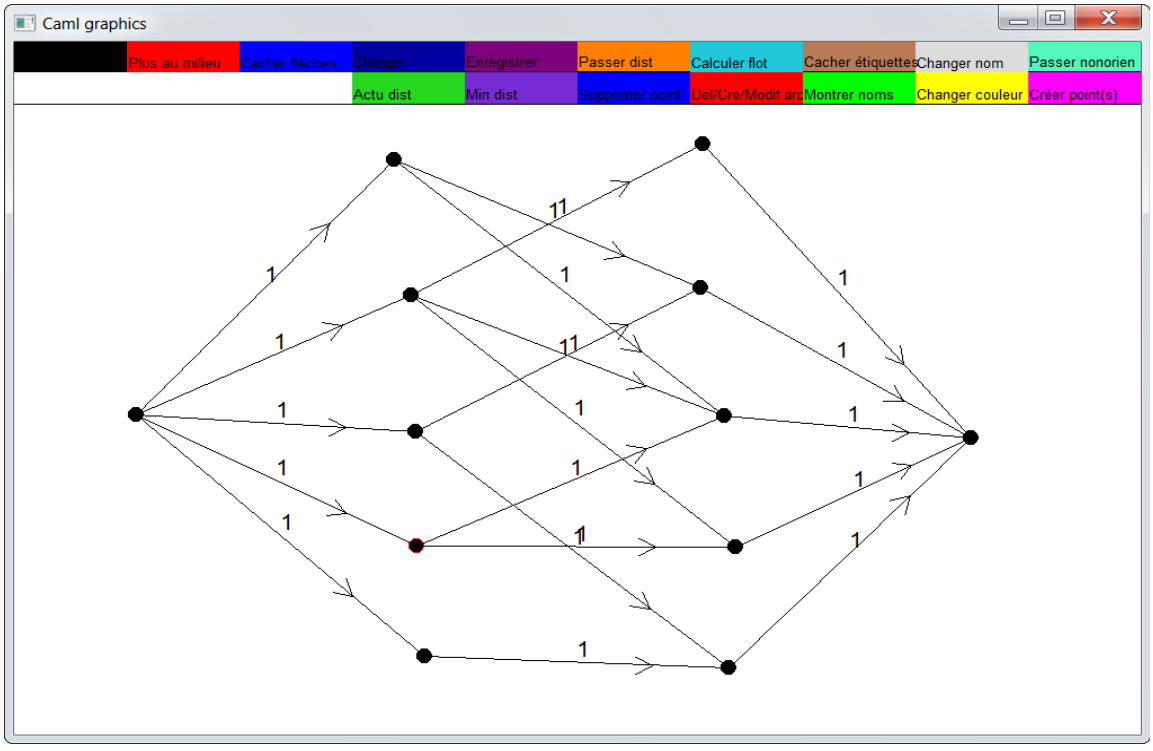
La partie la plus longue à coder fut la partie graphique. Nous avons utilisé la bibliothèque de fonctions de Caml-light appelée "graphics".

Nous montrons d'abord quelques copie d'écran et une description des fonctionnalités de notre programme, puis nous montrons le code source.

A31) Présentation de la partie graphique

Voici le "dessin" (graphe + informations sur sa représentation graphique) suivant, à partir duquel nous allons exposer les différentes fonctionnalités de notre programme.

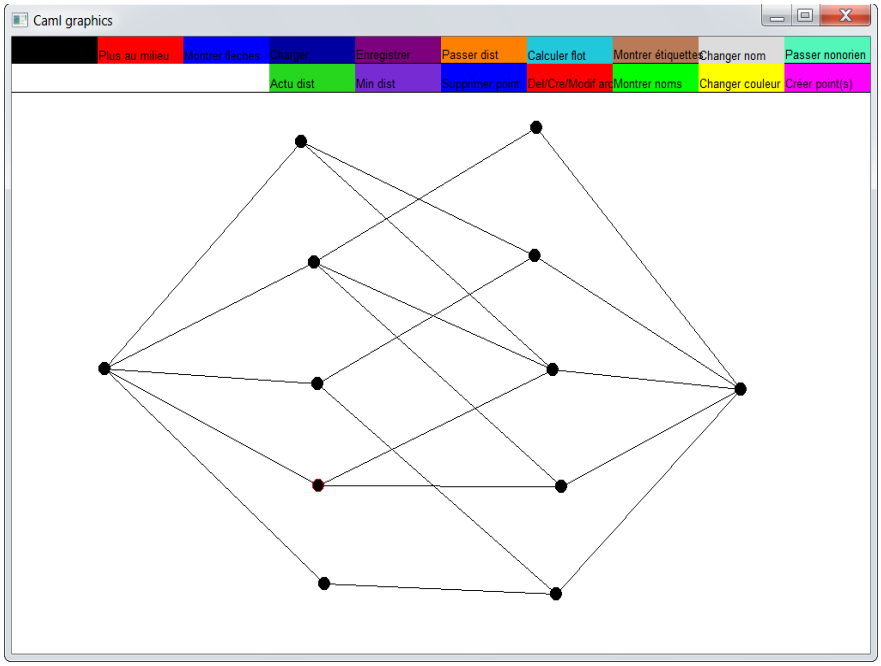
- Copie d'écran 1 :



- Codage du dessin :

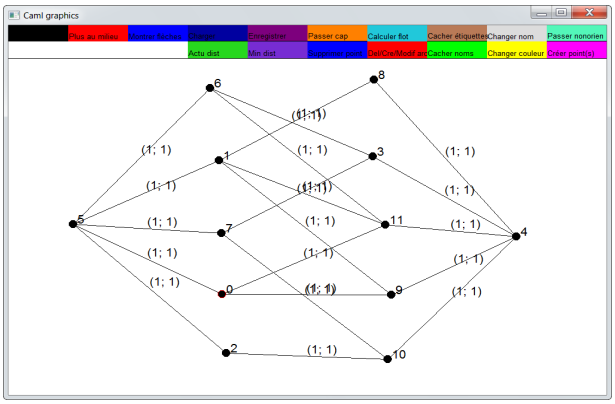
```
{tai = 12;
  arc =
   [|[9, [|ent 1; ent 1|]; 11, [|ent 1; ent 1|]];
     [9, [|ent 1; ent 1|]; 11, [|ent 1; ent 1|]; 8, [|ent 1; ent 1|]];
     [10, [|ent 1; ent 1|]]; [4, [|ent 1; ent 1|]]; [];
     [2, [|ent 1; ent 1|]; 0, [|ent 1; ent 1|]; 7, [|ent 1; ent 1|];
      1, [|ent 1; ent 1|]; 6, [|ent 1; ent 1|]];
     [11, [|ent 1; ent 1|]; 3, [|ent 1; ent 1|]];
     [3, [|ent 1; ent 1|]; 10, [|ent 1; ent 1|]]; [4, [|ent 1; ent 1|]];
     [4, [|ent 1; ent 1|]]; [4, [|ent 1; ent 1|]]; [4, [|ent 1; ent 1|]]|];
  pos =
   [|357, 179; 352, 418; 364, 74; 609, 425; 849, 282; 108, 304; 337, 547;
     356, 288; 611, 562; 640, 178; 634, 63; 630, 303|];
  nom = [|"0"; "1"; "2"; "3"; "4"; "5"; "6"; "7"; "8"; "9"; "10"; "11"|];
  col = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]}
```

- Copie d'écran 2 :



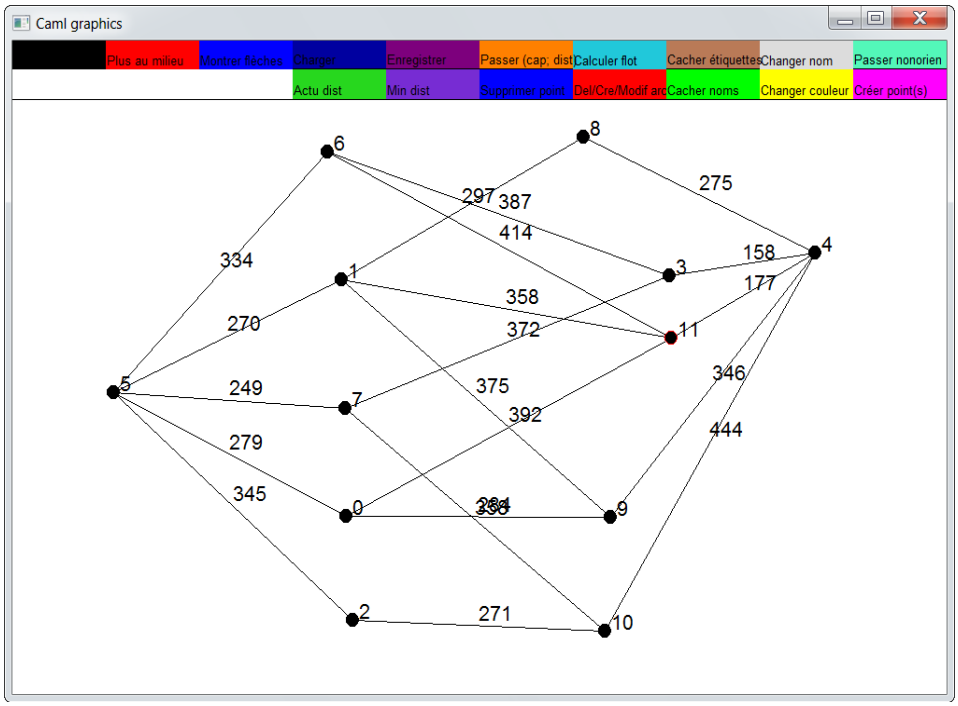
Sur la copie d'écran 2, ci dessus, l'utilisateur a appuyé sur le bouton "Cacher étiquettes" et sur sur le bouton "Cacher flèches" de la figure 1. Cela a enlevé de la figure, respectivement les nombres et les flèches montrant l'orientation des arcs.
Le nom des boutons en question (boutons entourés en noir, à l'aide de Paint) a été modifié : "Cacher étiquettes" est devenu "Montrer étiquettes" tandis que "Cacher flèches" est devenu "Montrer flèches". L'utilisateur pourra ainsi revenir à un autre mode d'affichage plus tard si il le désire.

- Copie d'écran 3 :



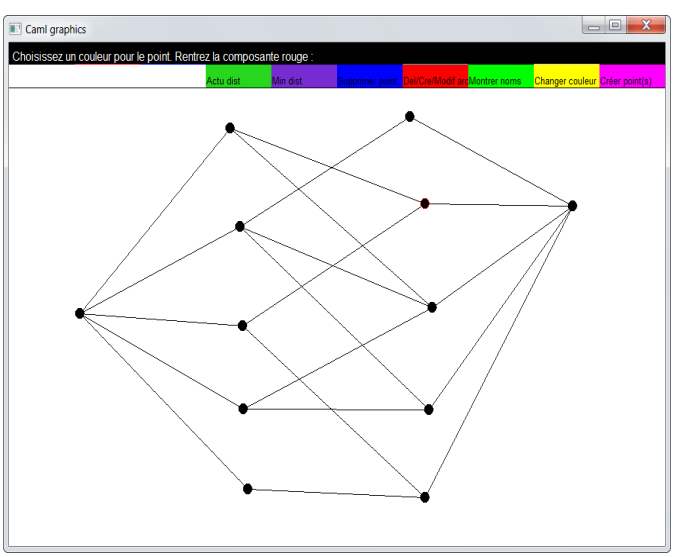
Cette copie d'écran (numéro 3) montre ce qui se passe si l'utilisateur a réactivé l'affichage des étiquettes, et a cliqué sur "Passer dist" deux fois de suite. Alors on voit s'afficher à la fois les capacités des arcs et leurs coûts unitaire. Il a également cliquer sur "montrer noms" et les noms des sommets sont maintenant affichés.

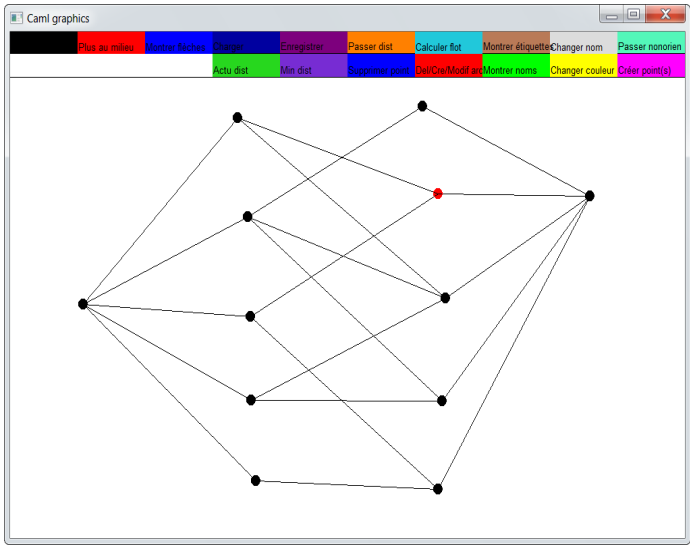
- Copie d'écran numéro 4 :



Cette fois, l'utilisateur a déplacé plusieurs points en les sélectionnant avec la souris puis en cliquant sur leur nouvelle positions. Il a aussi cliqué sur "actu dist" qui réactualise les distances en calculant la distance réelle entre les points, en pixels. Il s'est également placé en mode d'affichage "dist" (et on voit maintenant l'option "passer (cap,dist)" affichée).

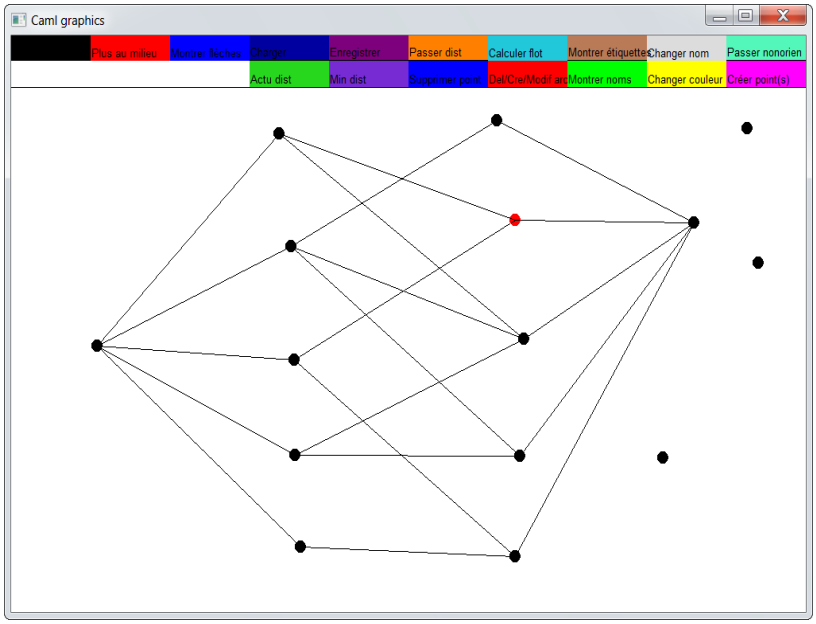
- Copies d'écran 5 et 6 :





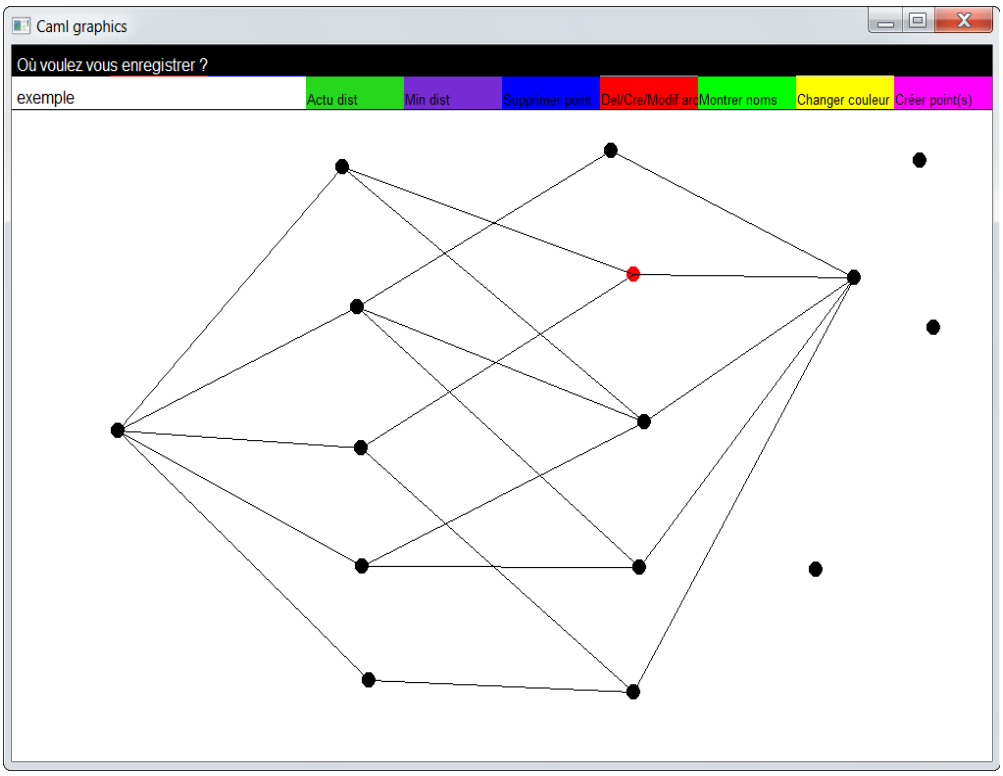
L'utilisateur a maintenant redéplacé les points à leur place de départ, a sélectionné un point et a cliqué sur "changer couleur". Le programme affiche alors "Choisissez une couleur pour le point. Rentrez la composante rouge". L'utilisateur rentrera 255, et le programme demandera les composantes verte et bleu, ausquelles l'utilisateur répondra 0. A la fin de l'opération, la couleur du point a changé : il est bien rouge.

- Copies d'écran 7 :



L'utilisateur a ici cliqué sur le bouton "créer points" et a cliqué avec la souris là où il voulait créer les points (il en a créé 3).

- Copies d'écran 8 :



Enfin l'utilisateur veut fermer le graphe et le sauvegarder. Il clique donc sur le bouton "sauvegarder". Le programme demande alors (cf copie d'écran) "Où voulez vous enregistrer ?". L'utilisateur écrit le nom du fichier où le dessin va être sauvegardé (ici "exemple"), et valide en appuyant sur enter. Instantanément un fichier "exemple" est créé dans un dossier prédéfini.
L'utilisateur pourra ensuite, même après avoir éteint l'ordinateur, recharger le graphe en cliquant sur "charger" et en tapant le nom du fichier.
L'utilisateur ferme finalement la fenêtre et on lui renvoie le dessin obtenu sous forme abstraite :

```
{tai = 15;
  arc =
   [|[9, [|ent 1; ent 284|]; 11, [|ent 1; ent 392|]];
     [9, [|ent 1; ent 375|]; 11, [|ent 1; ent 358|]; 8, [|ent 1; ent 297|]];
     [10, [|ent 1; ent 271|]]; [4, [|ent 1; ent 158|]]; [];
     [2, [|ent 1; ent 345|]; 0, [|ent 1; ent 279|]; 7, [|ent 1; ent 249|];
      1, [|ent 1; ent 270|]; 6, [|ent 1; ent 334|]];
     [11, [|ent 1; ent 414|]; 3, [|ent 1; ent 387|]];
     [3, [|ent 1; ent 372|]; 10, [|ent 1; ent 358|]];
     [4, [|ent 1; ent 275|]]; [4, [|ent 1; ent 346|]];
     [4, [|ent 1; ent 444|]]; [4, [|ent 1; ent 177|]]; []; []; []|];
  pos =
   [|357, 179; 352, 418; 364, 74; 634, 448; 859, 445; 108, 304; 337, 547;
     356, 288; 611, 562; 640, 178; 634, 63; 645, 312; 926, 553; 940, 399;
     820, 176|];
  nom =
   [|"0"; "1"; "2"; "3"; "4"; "5"; "6"; "7"; "8"; "9"; "10"; "11"; "12";
     "13"; "14"|];
  col = [|0; 0; 0; 16646400; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]}
```

Toute les fonctionnalités n'ont pas été exposée ici. Voici le sens des boutons qui n'ont pas encore été décrits :

- le bouton "calculer flot" demande une source et un puit à l'utilisateur, et alors il y a deux cas qui se présentent :
  - Si le mode est "onlycapa", alors le programme calcule et affiche un flot de valeur maximale.

- Si le mode est "dist" ou "(cap;dist)", alors le programme propose à l'utilisateur 3 choix : le calcul de flot maximum de coût minimum, le calcule de flot maximum à coût fixé, et le calcul de flot de coût minimum à valeur fixée.

- Le bouton "Cre/Del/Modif arc" permet de créer des arcs, de modifier leurs capacités ou leur distances, et de les supprimer. Il suffit de cliquer sur le point de départ de l'arc, puis sur le point d'arrivée, et enfin de dire au programme
  - si cet arc doit être supprimé ou modifié dans le cas où il existe déjà
  - rentrer directement les caractéristiques de l'arc si l'arc n'existe pas encore.
- Si l'utilisateur clique sur "Passer nonorien", les arcs qui seront créés à partir de cet instant le seront de manière symétrique ((x,y) en même temps que (y,x) et avec les mêmes caractéristiques).
- L'utilisateur peut supprimer un point en cliquant sur "supprimer point".
- Le bouton "min dist" permet de calculer un chemin de longueur minimale entre deux points.
- Enfin le bouton "Plus au milieu" permet de mettre les étiquettes des arcs plus vers l'extrémité, ce qui permet que les étiquettes de deux arcs opposés ne se chevauchent pas, mais qui permet aussi dans l'option "au milieu" d'éviter les redondances si tous les arcs sont symétriques par exemple.

A32) Code source de la partie graphique

```
(*Graphes orientés aux arêtes étiquetées*)

(*Structure du programme principal : modif_dessin :
découpé en menus correspondant aux effets de chaque boutons.
Attention la suppression, la création et la modification d'arcs en général sont liés au
même bouton :
"Del/Cre/Modif arc" *)
(*les graphes représentés sont sans boucles, ie pas d'arc d'un sommet vers lui-même *)

include "biblio2/TIPE/Programmes/Algorithmes/maxflow et mincost";;

let new_gra n = let a = make_vect n [] in a;;
(*pour un vide, n=0*)
(*---------------------------------------*)

let taille = 7;;
let taille2 = taille * taille;;
let taille_texte = 24;;

let new_col n = make_vect n black;;
let new_nom n = let v = make_vect n "" in for i = 0 to n-1 do v.(i) <- string_of_int i
done; v;;
let new_pos n = let v = make_vect n (-2,-2) in for i = 0 to n-1 do
      let x = random__int 1000 and y = random__int 600 in v.(i) <- (x, y) done; v;;
let new_des n = {tai = n; arc=new_gra n;pos=new_pos n;nom=new_nom n;col=new_col n};;
let k = new_des 0;;

let des_of_gra g = let n = vect_length g in
{tai = n;arc=g;pos=new_pos n;nom=new_nom n;col=new_col n};;
let des_of_gra2 g v = let n = vect_length g in
```

```
{tai = n;arc=g;pos=v;nom=new_nom n;col=new_col n};;
let des_of_gra3 g v w = let n = vect_length g in
{tai = n;arc=g;pos=(copy_vect v);nom=new_nom n;col=w};;

let dessiner (a,b) = fill_circle a b taille;;
let entourer k (a,b) = set_color k; draw_circle a b taille; set_color black;;
let plus_entouré v n = set_color black;
  for i = 0 to n - 1 do let (a, b) = v.(i) in draw_circle a b taille done;;

let ecrire (droite,haut) = let k = current_point() in lineto (fst k + droite) (snd k +
haut);
moveto (fst k) (snd k);;
let bouger (droite,haut) = let k = current_point() in moveto (fst k + droite) (snd k +
haut);;
let tourner30 c (a,b) = if c then (87*a/100 - b/2, a/2 + 87*b/100) else (87*a/100 +
b/2, - a/2 + 87*b/100);;

let unit (a, b) (c, d) = let intof = int_of_float and floatof = float_of_int in
    let u = floatof (c - a) and v = floatof (d - b) in
      let e = (sqrt (u ** 2. +. v ** 2.)) in
        if e >= 200. then let w = 20. *. u /. e and x = 20. *. v /. e in
            (- intof w, - intof x) else (((a - c) / 10), ((b - d) / 10));;

let segment montrer_flèches (a, b) (c, d) = moveto a b; lineto c d;
if montrer_flèches then begin
  let e = ((3 * c + a) / 4) and f = ((3 * d + b) / 4) in moveto e f;
    let j, k = unit (a, b) (c, d) in
      let (u, v) = tourner30 true (j, k) and (w, x) = tourner30 false (j, k) in
        ecrire (u, v); ecrire (w, x) end;;

let openg (tailx,taily) = let tailx1 = string_of_int tailx and taily1 = string_of_int
taily in
 open_graph (tailx1^"x"^taily1) ;;

(*---------------------*)

let mettre_bouton texte couleur a b = set_color couleur; set_text_size 16; set_font
"Arial";
  (*let u, v = text_size texte in*) fill_rect a b 100 30;
    set_color black; moveto (a + 1) (b + 1); draw_string texte; set_text_size 24;;

let affiche_orien orien = if orien then mettre_bouton "Passer nonorien" (rgb 83 247
185) 900 631
else mettre_bouton "Passer orien" (rgb 255 255 0) 900 631;;

let affiche_capa etiq = if etiq then mettre_bouton "Cacher étiquettes" (rgb 185 122 87)
700 631
else mettre_bouton "Montrer étiquettes" (rgb 185 122 87) 700 631;;

let affiche_dist dist = if dist=0 then mettre_bouton "Passer dist" (rgb 255 128 0) 500
631
else if dist = 1 then mettre_bouton "Passer (cap; dist)" (rgb 255 128 0) 500 631
else mettre_bouton "Passer cap" (rgb 255 128 0) 500 631;;

let affiche_noms noms = if noms then mettre_bouton "Cacher noms" green 700 600
else mettre_bouton "Montrer noms" green 700 600;;

let affiche_montrer_flèches montrer_flèches = if montrer_flèches then
mettre_bouton "Cacher flèches" blue 200 630
else mettre_bouton "Montrer flèches" blue 200 630;;

let affiche_au_milieu au_milieu = if au_milieu then
```

```
mettre_bouton "Plus au milieu" red  100 630
else mettre_bouton "Au milieu" red 100 630;;

(*On affiche les boutons :*)
let afficher_dessus orien etiq dist noms montrer_flèches au_milieu=
  moveto 0 600; lineto 1000 600; set_color black;
  fill_rect 0 630 1000 30; mettre_bouton "Supprimer point" blue 500 600;
mettre_bouton "Del/Cre/Modif arc" red 600 600;
mettre_bouton "Changer couleur" yellow 800 600;
mettre_bouton "Créer point(s)" magenta 900 600;
mettre_bouton "Changer nom" (rgb 220 220 220) 800 630;
mettre_bouton "Calculer flot" (rgb 33 200 218) 600 630;
mettre_bouton "Enregistrer" (rgb 125 0 125 ) 400 631;
mettre_bouton "Charger" (rgb 0 0 160) 300 631;
mettre_bouton "Actu dist" (rgb 39 215 30) 300 600;
mettre_bouton "Min dist" (rgb 119 44 211) 400 600;
affiche_orien orien;
affiche_capa etiq;
affiche_dist dist;
affiche_noms noms;
affiche_montrer_flèches montrer_flèches;
affiche_au_milieu au_milieu;;

let afficher_noms d = for i = 0 to d.tai-1 do set_color d.col.(i);
        let u, v = d.pos.(i) in moveto (u + taille) (v - 4); draw_string d.nom.(i)
done;;

let string_of_nombre n = match n with
omega->"+infini"
|omega_m->"-infini"
|indefini->"indefini"
|ent k->string_of_int k
|flo h->string_of_float h;;

let afficher_capacités dessin dist au_milieu= set_color black;
  for i = 0 to dessin.tai - 1 do let (a, b) = dessin.pos.(i) and l = ref dessin.arc.(i)
in
      while !l <> [] do let (u, v) = hd !l in let (c, d) = dessin.pos.(u) in
            if not au_milieu then moveto ((3 * c + a) / 4) ((3 * d + b) / 4)
            else  moveto ((c + a) / 2) ((d + b) / 2);
            if (v.(0)) >=& (ent 0) then
              if dist=0 then draw_string (string_of_nombre v.(0))
              else if dist = 1 then draw_string (string_of_nombre v.(1))
else draw_string ("(" ^ string_of_nombre (v.(0)) ^ "; " ^ string_of_nombre (v.(1)) ^
")");
            l := tl !l done; done;;

let afficher_dessous d orien optdist affnoms affdistances num montrer_flèches
au_milieu=
  for i = 0 to d.tai-1 do set_color d.col.(i); let u,v = d.pos.(i) in dessiner (u,v);
done;
  set_color black; let rec tracer m i =
match m with []->() |(a,b)::c-> segment montrer_flèches d.pos.(i) d.pos.(a); tracer c i
in
  for i = 0 to d.tai-1 do tracer d.arc.(i) i done;
if affnoms then afficher_noms d; if affdistances then afficher_capacités d optdist
au_milieu;
 if num>=0 then entourer red d.pos.(num);;

let afficher_tout d orien optdist affnoms affdistances num montrer_flèches au_milieu =
openg (1000, 660); clear_graph();
afficher_dessous d orien optdist affnoms affdistances num montrer_flèches au_milieu;
```

```
afficher_dessus orien affdistances optdist affnoms montrer_flèches au_milieu;;

let ecrire_message (mes: string) = set_color black; set_font "Arial"; set_text_size 18;
  fill_rect 0 631 1000 30; set_color white; moveto 5 632;
  draw_string mes; set_color black; set_text_size 24;;

let pret_pour_reponse () = set_color white; fill_rect 0 600 250 30; moveto 5 602;
set_color black;;

let questionner (qu: string) (var: string ref) = ecrire_message qu;
pret_pour_reponse(); set_text_size 18;
      let m = ref "" and c = ref true and n = ref 0 in
        try
          while !c do
            let e = wait_next_event [Button_down; Key_pressed] in
              if e.keypressed then let j = e.key in
                  begin
                    if j <>`\013`&& j <> (char_of_int 8) && !n <= 26 then
                      begin m := (!m) ^ (string_of_char j); draw_char j; n := !n + 1
end
                    else if j = (char_of_int 8) then
                      begin pret_pour_reponse(); n := 0; m := ""; end
                    else if j =`\013`then
                      begin if !n <> 0 then begin var :=!m end;
                        c := false; set_color white; fill_rect 0 600 250 30; set_color
black;
                        fill_rect 0 630 1000 30; end
                  end;
          done;set_text_size 24;
        with Graphic_failure "graphic screen not opened" -> ();;

let en_bas (a, b) = a < 1000 && b < 600;;
(*let verifier (a, b) (c, d) = max (abs (a - c)) (abs (b - d)) < taille;;*)

let verifier (a, b) (c, d) = let p = (a-c) and q = (d-b) in (p*p + q*q) <= taille2;;

let appartient (*pour savoir si le point ab est dans v*) (a, b) v n =
  if v = [||] then false, -1 else begin let k = ref 0 and c = ref false in while !k < n
&& not !c do
          c := verifier (a, b) v.(!k); if not !c then k := !k + 1 done; !c, !k end;;

let est_dans a b (i, j) (k, l) = a >= i && a <= k && b >= j && b <= l;;

(*-----------------------------------------------*)

let question_couleur dessin orien num = if num>=0 then begin
let couleur = ref black and a = ref "" in(*lancer les 2 questions couleurs et nom.*)
questionner "Choisissez un couleur pour le point. Rentrez la composante rouge :" a;
couleur := 65280*int_of_string !a;
questionner "Rentrez maintenant la composante verte :" a; couleur:= !couleur +
256*int_of_string !a;
questionner "Enfin, rentrez la composante bleu :" a; couleur:= !couleur + int_of_string
!a;
dessin.col.(num)<- !couleur; set_color !couleur; let u,v = dessin.pos.(num) in dessiner
(u,v); set_color black
end;;

let question_nom dessin orien onlycapa num montrer_flèches au_milieu= if num >=0 then
begin
let nom = ref "" in
questionner ("Le nom actuel du point est '"^dessin.nom.(num)^"'. Rentrez son nouveau
nom (moins de 26 lettres):") nom;
```

```
dessin.nom.(num)<- !nom;
afficher_tout dessin orien onlycapa true true num montrer_flèches au_milieu end;;

let changer_position dessin orien onlycapa num affnoms affdist (x,y) montrer_flèches
au_milieu=
 dessin.pos.(num)<- (x,y); clear_graph();
afficher_tout dessin orien onlycapa affnoms affdist num montrer_flèches au_milieu;;

let creer_point dessin (x,y) =   dessin.tai<-dessin.tai+1;
                      dessin.arc<- concat_vect dessin.arc [|[]|];
                    dessin.pos<- concat_vect dessin.pos [|(x,y)|];
                      dessin.nom<- concat_vect dessin.nom [|string_of_int (dessin.tai
-1)|];
                    dessin.col<-concat_vect dessin.col [|black|];
                    dessiner (x,y);;

let mode_creation dessin= ecrire_message "Cliquez sur une touche quand vous avez fini";
  let c = ref true and x = ref 0 and y = ref 0 in while (!c) do
      let e = wait_next_event [Button_down; Key_pressed] in
        if e.button then begin
            let t = (mouse_pos()) in x := fst t; y := snd t;
              creer_point dessin (!x, !y) end
        else if e.keypressed then c := false;
    done;;

let nombre_of_string a =try( try match a with "infini"-> omega
 |_-> ent(int_of_string a) with Failure "int_of_string"->flo(float_of_string a)) with
 Failure "float_of_string"-> omega;;

let demander_distance optdist noma nomb =
  let d = ref "infini" and e = ref "1" in
    if optdist = 0 || optdist = 2 then questionner
      ("Choisissez une capacité positive, nulle ou négative (pour infinie) du point " ^
noma ^ " au point " ^ nomb ^ " :") d;
    if optdist = 1 || optdist = 2 then questionner
      ("Choisissez une distance positive, nulle, négative ou infinie (taper infini) du
point " ^ noma ^ " au point " ^ nomb ^ " :") e;
    let r = nombre_of_string !d
    and s = nombre_of_string !e in if r >=& (ent 0) then (r, s) else (omega, s);;

let creer_arc (*orienté*)dessin optdist a b montrer_flèches =
segment montrer_flèches dessin.pos.(a) dessin.pos.(b);
let d,e = demander_distance optdist dessin.nom.(a) dessin.nom.(b) in if (optdist=0)
then
dessin.arc.(a)<- (b,[|d;e|])::dessin.arc.(a)
else dessin.arc.(a)<- (b,[|d;e|])::dessin.arc.(a);;

let selectionnez (*avec confirmation par enter*) dessin num txt = ecrire_message txt;
  let c = ref true and x = ref 0 and y = ref 0 and f = ref num in while (!c) do
      let e = wait_next_event [Button_down; Key_pressed] in
        if e.button then begin
            let t = (mouse_pos()) in x := fst t; y := snd t;
              let u, v = appartient (!x, !y) dessin.pos dessin.tai in
              (*point touché*) if u = true then f := v; end
        else if e.keypressed then begin
            if e.key = `4` then begin if !f <> 0 then f := !f - 1 else f := dessin.tai
- 1; end
            else if e.key = `6` then begin if !f = dessin.tai - 1 then f := 0 else f :=
!f + 1; end
            else if e.key = `\013` then c := false;
          end; (*<-fin du wait*)
          plus_entouré dessin.pos dessin.tai;
```

```
          entourer red dessin.pos.(!f);
          pret_pour_reponse(); draw_string (string_of_int !f);
    done; !f;;

let selectionnez2 (*sans confirmations, le premier selectionné*) dessin num txt =
  ecrire_message txt;
  let c = ref true and x = ref 0 and y = ref 0 and v = ref num in
    while (!c) (*si point non encore trouvé*) do
      let e = wait_next_event [Button_down; Key_pressed] in
        if e.button then begin
            let t = (mouse_pos()) in x := fst t; y := snd t;
              let (a,b) = appartient (!x, !y) dessin.pos dessin.tai in
                c := not a; v := b end
    done; !v;;

let rec existe_lien a = function [] -> false | (u, v) :: y -> a = u || existe_lien a
y;;

let rec enleve b l = match l with [] -> []
    | (c, d) :: e -> if c = b then enleve b e else (c, d) :: enleve b e;;

let supprimer_arc dessin a b montrer_flèches=
dessin.arc.(a)<-enleve b dessin.arc.(a);
set_color white; segment montrer_flèches dessin.pos.(a) dessin.pos.(b); set_color
black;;

let rec distance_2 optdist b l u v=
(*remplace la 1ere incidence de b dans l par la
 mm chose mais avec la cap ou la dist remplacée par u *)
match l with [] -> []
    | (c, d) :: e when c = b -> (c,[|u;v|])::e
|_-> (hd l) :: distance_2 optdist b (tl l) u v;;

let creer_arc_non_orien dessin optdist a b montrer_flèches=
segment montrer_flèches dessin.pos.(a) dessin.pos.(b);
segment montrer_flèches dessin.pos.(b) dessin.pos.(a);
let (d,e) = demander_distance optdist dessin.nom.(a) dessin.nom.(b) in
dessin.arc.(a)<- (b,[|d;e|])::dessin.arc.(a);
dessin.arc.(b)<- (a,[|d;e|])::dessin.arc.(b);;

let menu_arcs dessin num orien optdist montrer_flèches(*mode arcs*)=
  let a = ref 0 and b = ref 0 and c = ref false and d = ref "" in
    a:=selectionnez2 dessin num "Choisissez le point de départ";
    b:=selectionnez2 dessin num "Choisissez le point d'arrivée" ;
if orien then begin
    c := existe_lien !b dessin.arc.(!a);
    if not !c then creer_arc dessin optdist !a !b montrer_flèches
    else begin questionner "Voulez-vous supprimer cet arc : 1:oui / autre que 1:non" d;
        if !d = "1" then supprimer_arc dessin !a !b montrer_flèches
        else let u,v = demander_distance optdist dessin.nom.(!a) dessin.nom.(!b) in
         dessin.arc.(!a)<-distance_2 optdist !b dessin.arc.(!a) u v end
end
else begin
(*propose de supprimer les deux si un des deux au moins existe déjà, sinon crée les
deux avec la mm dist*)
(*dans le cas où au - un des deux et pas supprimer, il modifie les longueurs de ce qui
existe déjà*)
(*donc si on veut mettre les 2 et qu'il n'y en a qu'un, on doit supprimer et refaire*)
    c:= ( existe_lien !b dessin.arc.(!a) || existe_lien !a dessin.arc.(!b) );
    if not !c then creer_arc_non_orien dessin optdist !a !b montrer_flèches
    else begin questionner "Voulez-vous supprimer cet arc : 1:oui / autre que 1:non" d;
        if !d = "1" then begin supprimer_arc dessin !a !b montrer_flèches;
```

```
supprimer_arc dessin !b !a montrer_flèches end
        else begin let t,s= demander_distance optdist dessin.nom.(!a) dessin.nom.(!b)
in
          dessin.arc.(!a)<- distance_2 optdist !b dessin.arc.(!a) t s;
          dessin.arc.(!b)<- distance_2 optdist !a dessin.arc.(!b) t s end end
end;;

(*------------------------------*)
(*suppression de points*)

let coupe_vect k n v = if k <= n-1 then
  concat_vect (sub_vect v 0 k) (sub_vect v (k+1) (n-k-1)) else v;;

let dernieres_modif k m v =
   for i = 0 to m - 1 do
    let l = ref v.(i) and l2 = ref [] in
     while !l <> [] do let r, s = (hd !l) in
          if r < k then l2 := (hd !l) :: !l2
          else if r > k then l2 := (r - 1, s) :: !l2;
          l := tl !l; done;
     v.(i) <- !l2 done;
v;;

let appartient (*pour savoir si le point ab est dans v*) (a, b) v n =
  if v = [||] then false, -1 else begin let k = ref 0 and c = ref false in while !k < n
&& not !c do
          c := verifier (a, b) v.(!k); if not !c then k := !k + 1 done; !c, !k end;;

let delete dessin k = (*if k<dessin.gra.tai then begin ...*) let n = dessin.tai in
dessin.tai<-n-1;
dessin.arc<-coupe_vect k n dessin.arc;
 dessin.pos<-coupe_vect k n dessin.pos;
 dessin.nom<-coupe_vect k n dessin.nom;
 dessin.col<-coupe_vect k n dessin.col;
dessin.arc<-dernieres_modif k (n-1) dessin.arc;;

let supprimer_point dessin num = let a = ref num in
    a:=selectionnez2 dessin num "Quel point voulez vous supprimer ?";
    delete dessin !a;;

(*-----------------*)
(*enregistrer et charger*)

let enregistrer d fichier =
    let canal_out = open_out_bin fichier
    in output_value canal_out d;
       close_out canal_out;;

let ouvrir fichier=
let canal_in = open_in_bin fichier in
let d = (input_value canal_in) in close_in canal_in;d;;

let chemin_des = "biblio2/TIPE/Programmes/sauvegardes/dessins/";;
let chemin_gra = "biblio2/TIPE/Programmes/sauvegardes/graphes/";;

let enregistrer_dessin d = let rep = ref "" in
questionner "Où voulez vous enregistrer ?" rep;
let fichier1 = chemin_des^(!rep) in
enregistrer d fichier1 ;;

let charger_dessin d = let rep = ref "" in
questionner "Quel fichier voulez vous charger ?" rep;
```

```
let fichier1= chemin_des^(!rep) in
let d2 = ouvrir fichier1 in
d.tai<-d2.tai; d.arc<-d2.arc;d.pos<-d2.pos;d.col<-d2.col;d.nom<-d2.nom;;

let vect_couleurs u c1 c2 =let n = (vect_length u) in
let v = make_vect n black in
for i = 0 to n - 1 do if u.(i) then v.(i)<-c1 else v.(i)<-c2 done;v;;


(*----------------------------------------*)
(*Distances*)

let sqrt_int a = (*arrondie au supérieur,
sqrt_int 9 = 3 mais sqrt_int 10 = 4*)
  let i = ref 1 and k = ref 0 in
    while !k < a do
      k := !k + !i;
      i := !i + 2 done; !i / 2;;

let dist_int (a, b) (c, d) =
  let k = a - c and t = b - d in sqrt_int (k * k + t * t);;

let dist_float (a, b) (c, d) =
  let k = floatof (a - c) and t = floatof (b - d) in sqrt (k ** 2. +. t ** 2.);;

let vrai_dist (g:((int*nombre vect) list) vect) pos n e =
(*renvoie le graphe g avec les distances réelles en pixel*)
let G = make_vect n [] in
  (* /!\  e si on veut une distance entière *)
    let rec aux v k l =
      match l with [] -> []
        | _ -> let u, w = hd l in let w2 = ref (ent 0) in
               if e then w2 := ent (dist_int v.(k) v.(u) (*/100*))
               else w2 := flo (dist_float v.(k) v.(u)(*/.100.*));
               (u, [|w.(0); !w2|]) :: (aux v k (tl l)) in
      for i = 0 to n - 1 do
        G.(i) <- (aux pos i g.(i)) done;
      G;;

let actu_dist dessin =
let gr = vrai_dist dessin.arc dessin.pos dessin.tai true in
dessin.arc <- gr;;

let afficher_bizarre dessin r couleurs optdist montrer_flèches au_milieu= clear_graph
();
  let n = vect_length dessin.pos in
    for i = 0 to n - 1 do let (a, b) = dessin.pos.(i) in dessin.pos.(i) <- ((a / 2),
b); done;
    let v = copy_vect dessin.pos in for i = 0 to (n - 1) do
        v.(i) <- (500 + fst v.(i), snd v.(i)) done;
      let dd = {tai = n; arc = r; pos = v; nom = dessin.nom; col = couleurs} in
        afficher_dessous dd true 0 false true 0 montrer_flèches au_milieu;
        afficher_dessous dessin true 2 false true 0 montrer_flèches au_milieu;
        for i = 0 to n - 1 do let (a, b) = dessin.pos.(i) in dd.pos.(i) <- (2 * a, b);
done;
        dd.col <- dessin.col;
        dd;;
(*afficher_dessous d orien onlycapa affnoms affdistances num montrer_flèches*)

(*--------------------------------------*)

let afficher_gras (*ep*) pos chemin =
```

```
  (*affiche un chemin en gras, en fonction de l'épaisseur ep*)
  set_line_width 3 (*ep*);
  let rec continuer l = match l with
      a :: b -> lineto (fst pos.(a)) (snd pos.(a)); continuer b
      | _ -> set_line_width 1 in
    match chemin with a :: b -> moveto (fst pos.(a)) (snd pos.(a));
          continuer b | _ -> set_line_width 1;;

let plus_court_graphique dessin num a b =
  a := selectionnez2 dessin !num "Selectionnez le départ";
  b := selectionnez2 dessin !num "Selectionnez l'arrivée";
  let r = ref "" in questionner "1:PP // 2:PL // 3: Bellman // 4:Djikstra" r;
    if !r = "1" then let (chem, _) = (PP dessin.arc dessin.tai !a !b) in afficher_gras
dessin.pos chem
    else if !r = "2" then let (chem, _) = (PL dessin.arc dessin.tai !a !b) in
afficher_gras dessin.pos chem
    else if !r = "3" then let (chem, _, _) = bellman dessin.arc dessin.tai !a !b in
afficher_gras dessin.pos chem
    else let (chem, _, _) = djikstra dessin.arc dessin.tai !a !b in afficher_gras
dessin.pos chem;;

let demander_mincost arcs a b =
  let r = ref "" in questionner "1:maxflow_mincost // 2:coût fixé // 3: flot fixé" r;
    if !r = "1" then (maxflow_mincost arcs a b)
    else begin let s = ref "" in questionner "Valeur max :" s;
          let t = nombre_of_string !s in
            if !r = "2" then mincost_coût_fixé arcs a b t
            else mincost_flot_fixé arcs a b t; end;;

(*--------------------------------------*)

let menu_principal dessin =
  let c = ref true and num = ref 0 and x = ref 0 and y = ref 0 and affnoms = ref false
and optdist = ref 0
  and a = ref 0 and b = ref 0 and affetiq = ref false and orien = ref true and
montrer_flèches = ref true
  and au_milieu = ref true in
    while (!c) do
      afficher_tout dessin !orien (!optdist) !affnoms !affetiq !num !montrer_flèches !
au_milieu;
      let e = wait_next_event [Button_down; Key_pressed] in
        if e.button then (*1*) begin

            let t = (mouse_pos ()) in x := fst t; y := snd t;
              if en_bas (!x, !y) then begin let u, v = appartient (!x, !y) dessin.pos
dessin.tai in
                  (*point touché*) if u then begin num := v; afficher_tout dessin !
orien (!optdist) !affnoms !affetiq !num !montrer_flèches !au_milieu end
                      (*change pos*) else changer_position dessin !orien (!optdist) !
num !affnoms !affetiq (!x, !y) !montrer_flèches !au_milieu end

                (*bouton modifier param*)
              else if est_dans !x !y (500, 600) (600, 630) then begin num := 0;
supprimer_point dessin !num end
                (*attention : le point selectionné sera, après la suppression, le
nouveau point 0,
                  donc on ne peut pas enlever tous les points, il doit en rester au
moins 1.*)
              else if est_dans !x !y (600, 600) (700, 630) then menu_arcs dessin !num !
orien !optdist !montrer_flèches
              else if est_dans !x !y (700, 600) (800, 630) then affnoms := not !affnoms
              else if est_dans !x !y (700, 631) (800, 660) then affetiq := not !affetiq
```

```
              else if est_dans !x !y (800, 600) (900, 630) then question_couleur dessin
!orien !num
              else if est_dans !x !y (800, 631) (900, 660) then question_nom dessin !
orien (!optdist) !num !montrer_flèches !au_milieu
              else if est_dans !x !y (900, 631) (1000, 660) then orien := not !orien
              else if est_dans !x !y (900, 600) (1000, 630) then mode_creation dessin
              else if est_dans !x !y (500, 631) (600, 660) then optdist := (!optdist +
1) mod 3
              else if est_dans !x !y (400, 631) (500, 660) then enregistrer_dessin
dessin
              else if est_dans !x !y (300, 631) (400, 660) then charger_dessin dessin
              else if est_dans !x !y (400, 600) (500, 630) then plus_court_graphique
dessin num a b
              else if est_dans !x !y (300, 600) (400, 630) then actu_dist dessin
              else if est_dans !x !y (600, 631) (700, 660) then c := false
              else if est_dans !x !y (200, 631) (300, 660) then montrer_flèches :=
not !montrer_flèches
              else if est_dans !x !y (100, 631) (200, 660) then au_milieu := not !
au_milieu
          end (*cas du clic fini fini*)
        else if e.keypressed then begin
            if e.key = `4` then begin if !num <> 0 then num := !num - 1 else num :=
dessin.tai - 1; end
            else if e.key = `6` then begin if !num = dessin.tai - 1 then num := 0 else
num := !num + 1; end;
          end;
    done;

    a := selectionnez2 dessin !num "Selectionnez la source";
    b := selectionnez2 dessin !num "Selectionnez le puit";
    if (!optdist) = 0 then begin
        let (p, q, r, s) = (maxflow dessin.arc !a !b) in
          let couleurs = vect_couleurs p green red in
            let flot = afficher_bizarre dessin r couleurs !optdist !montrer_flèches !
au_milieu in
              enregistrer_dessin flot; (ent 0), (p, q, r, s); end
    else begin let (m, (p, q, r, s)) = demander_mincost dessin.arc !a !b in
          let couleurs = vect_couleurs p green red in
            let flot = afficher_bizarre dessin r couleurs !optdist !montrer_flèches !
au_milieu in
              enregistrer_dessin flot; (m, (p, q, r, s)); end;;
(*<-flot, (coupe, valeur du flot, graphe des écarts, graphe du flot max) *)

let essai n = let des = (new_des n) in (menu_principal des);;

let essai2 n = let des = (new_des n) in
try (menu_principal des), des; with
Graphic_failure "graphic screen not opened"->(ent 0, ([||], ent 0,[||],[||])),des;;

(*essai2 5;;*)

(*------------------------------*)
```

A4) Définition et notation de base de théorie des graphes

A5) Algorithmes d'accessibilités et de recherche de plus court chemin

A5.1) Recherche de chemin dans des graphes non valués

A5.1.1) Recherche en largeur (BFS)

- On note "d" le numéro du sommet de départ et "a" celui du sommet d'arrivée. Le graphe (X,A) est supposé simple, ie sans boucles.
  L'algorithme va en fait chercher des chemins les plus court possibles pour aller de "d" à n'importe quel sommet accessible depuis d. On peut éventuellement l'arrêter dès que "a" est atteint, en rajoutant les lignes de code qui sont en bleu. Si on veut également stocker les distances de "d" à chaque sommet, il faut rajouter les lignes en rouge.
  On stocke les marquages des sommets dans un vecteur V.
  n est le nombre de sommets du graphe. Ils sont numérotés de 0 à (n-1).
  La liste L est ici une liste LIFO (last in first out), qui fonctionne comme une file d'attente : le premier élément empilé est le premier dépilé. On pourrait appeler ces files des files "justes" car elles avantagent les premiers arrivés.

- **BFS** ((X,A) , d) :
  { L := [d] ; V := make_vect n (-1); V.d:=(-2); W := make_vect n (infini); W.d:=0;
  Tant que (L != [ ])
  { x := dépiler L ;
  Pour tout y tel que $(x, y) \in A \text{ et } V.y = (-1)$,
  faire { V.y := x; W.y := W.x + 1; si y != a empiler y; else L:= [ ]; }
  }
  Retourner (décoder V a) , W.b; }

- **Décoder** V a : { L := [a]; T:=V.a
  Tant que T != (-1) && T!=(-2) { empiler T; T:=V.T; }; L }

- **Remarques :** Alors on remarque que si aucun chemin n'existe jusqu'à "a", on obtiendra la liste [a] car V.a sera égal à (-1).
  On comprend aussi mieux pourquoi l'absence de boucle est demandée : si l'arc (d,d) appartient à A, alors on aura V.d = d et au décodage cela créera une boucle infinie.
  Cependant si il s'agit d'une boucle (x,x) où x!=d, alors cela n'a pas d'incidence.
  Pour éviter ce bug on peut écrire les lignes de code en vert.
- **Complexité :** On voit tout d'abord que la complexité de BFS est en $O(|X|+|A|)$ car chaque sommet ne peut être empilé au maximum qu'une fois, et qu'à chaque fois qu'on dépile un sommet x il faut passer en revue tous les arcs (x,y) pour savoir si V.y est égal à (-1) ou pas. Dans le cas d'un graphe connexe on a |A| >= |X|-1 donc on a une complexité de $O(|A|)$. Si le graphe n'est pas connexe, on peut raisonner sur le sous-graphe accessible depuis "d", qui est de taille inférieure, donc dans tous les cas, l'algorithme est en $O(|A|)$.
- **Correction :** Enfin, montrons par récurrence sur les étapes de l'algorithme que le BFS trouve les chemins les plus courts :
  On suppose que tous les sommets marqués jusqu'ici le sont bien par un chemin de longueur

minimale, et que les distances des sommets de L sont croissantes : les sommets qui vont bientôt sortir sont les plus proches de "d". Ceci est vrai au départ, où L=[d].
Si c'est vrai à une étape de l'algorithme, prouvons que c'est vrai à la suivante :
Si on dépile un sommet x de distance k, et qu'on le traite en empilant les sommets qui sont accessibles alors on va conserver la croissance des distances dans L d'après l'hypothèse de récurrence. De plus si il existait un trajet plus court vers un des sommets empilés, on l'aurait déjà marqué depuis un sommet de distance < k, ce qui est faux puisqu'il n'est pas marqué.
Donc par récurrence la propriété est vraie tout au long de l'algorithme, donc les chemins obtenus sont bien les plus courts.

- On peut reformuler le BFS de manière plus élégante sans utiliser de structures de données : On pose $E_0=\{a\}$ et $E_{k+1}=\Gamma(E_k)\setminus(\bigcup_{i\in[\![1;k]\!]} E_i)$. Alors E_k est l'ensemble des sommets à distance k de "d".

A5.1.2) Recherche en profondeur (DFS)

- On peut coder la recherche en profondeur de deux manières, même si l'algorithme réalisera finalement exactement la même chose.
- La première manière est presque la même que pour le BFS, à la différence que la liste L est cette fois une liste LIFO (last in first out) : elle fonctionne comme une pile d'assiettes, c'est à dire que l'élément rentré en dernier est le premier à sortir.
- La seconde manière de coder est récursive (alors que la première est itérative), mais nous ne l'avons pas codé en caml.

A5.2) Recherche de chemin dans des graphes valués

Il faut tout d'abord souligner que parfois, on ne peut définir de plus court chemin entre deux points : c'est le cas si on peut atteindre un cycle absorbant en partant du sommet de départ . Un cycle absorbant est un cycle dont la distance totale est négative. Il ne peut donc pas exister si les arcs ont des coûts négatifs.
Il existe principalement trois algorithmes (qui ont chacun des variantes) pour calculer des plus courts chemins entre points dans les graphes valués : l'algorithme de Djikstra, l'algorithme de Bellman, et l'algorithme de Floyd.
Cependant, l'algorithme de Floyd calcule les distances entre tous les points, ce qui est inutile ici car on a juste besoin de connaitre la distance entre la source et le puit.
Nous avons donc simplement codé les algorithmes de Djikstra et de Bellman. Ils calculent en fait les distances du point de départ à tous les autres points qui sont accessible depuis celui-ci.
L'algorithme de Djikstra est plus efficace algorithmiquement, mais il ne peut calculer des chemins les plus courts que si les distances sont positives. C'est un algorithme glouton, c'est à dire qu'à chaque étape, il ne modifie pas ce qui a été trouvé avant.
L'algorithme de Bellman est un algorithme dynamique : il calcule d'abord les chemins les plus courts parmi les chemins de longueur 1 ou moins partant du point de départ d. Puis il calcule les plus court chemins de longueur 2 ou moins à) partir des précédents, etc jusqu'à ce que les distances se stabilisent où que le nombre d'étapes dépasse le nombre de points du graphe (ce qui signale la présence d'un graphe absorbant.

Pour des raisons de temps, nous ne pouvons pas rentrer ici dans les détails de la démonstration de ces algorithmes.

A6) Suggestions d'amélioration

A6.1) Discussion sur les codages possibles d'un graphe en caml

Voici les possibilités pour un graphe :

1) Orienté ? 2 possibilités
2) Avec combien de nombres ? (0,1 ou 2 ici, mais pourquoi pas n entier quelconque) 4 possibilités
3) Multigraphe ? 2 possibilités
4) Utilisation des matrices, des listes ou des avl ? 3 possibilités

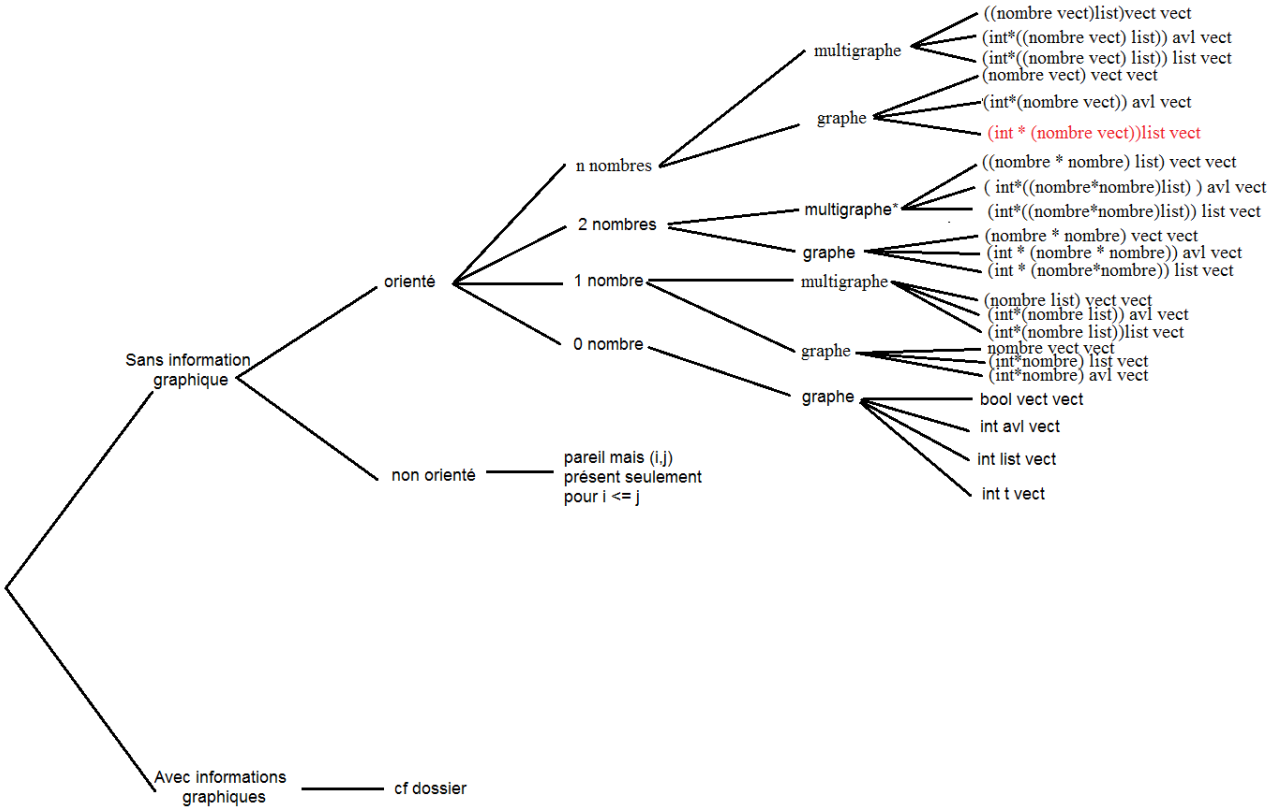
Ce qui donne 48 possibilités de sorte de graphes, et si on rajoute la question : Informations sur la représentation graphique ? (2 possibilités) cela fait 96.
Cependant ces 48 autres peuvent être déduite de manière automatique : si la structure du graphe est 'a , alors la structure avec les information graphique est :
type 'a structure_dessin = {**tai** : int; **arc** : 'a; **nom**: string vect; **pos** : (int * int) vect; **col** : color vect};;
On pourrait enfin se poser la question de la présence de boucle, mais on supposera qu'il n'y en a pas.

Ce qui fait donc 48 structures.
Mais celle orientées et non orientées se codent en fait de la même manière, si ce n'est le fait que l'arc (i,j) ne sera présent qu'une fois si le graphe est non orienté, dans l'emplacement où i<=j par exemple.
Il faudra donc définir les structures de données ET les fonctions de base qui s'y ramènent, qui changeront selon le fait que le graphe est orienté ou pas.
De plus il est inutile de coder les multigraphes sans nombres car on peut le coder avec des graphes valués par un nombre entier comme étiquette.
De plus, grâce à la bibliothèque caml-light "set", on peut faire : #open "set";; let 'a ensemble = 'a t;; (mais ceci n'est utile que dans le cas des arêtes non valuées, car il n'existe pas de fonctions de la bibliothèque "set" qui permettent de renvoyer tous les éléments d'un ensemble vérifiant une propriété donnée, plus fine que l'égalité).
Enfin, pour coder une étiquette contenant deux nombres, on peut hésiter entre un vecteur, qui pourra éventuellement ne contenir en fait qu'un nombre ou plus que deux en cas de modification ultérieure de l'algorithme, et un (nombre * nombre), qui est plus rigide. L'avantage du vecteur est qu'il est mutable, et je l'ai donc choisi, mais c'est discutable. Surtout dans le cas des multigraphes à deux nombres, où il serait avantageux d'utiliser la structure

(int * ((nombre * nombre)list)) list vect
ce qui permettrait de regrouper tous les arcs (i,j) ensembles.
On pourrait aussi stocker les étiquettes dans des structures séparées des direction des arcs, mais cela rendrait des erreurs trop faciles et augmenterait les déplacements d'informations inutiles.

Il y a en fait beaucoup de contraintes : justesse du programme, clarté du code, rapidité d'éxécution, généralité du programme...
Il est souvent dur de faire quelque chose d'efficace et de général à la fois, surtout qu'il faut s'adapter à la situation : le fait que le graphe soit dense ou non change par exemple beaucoup l'intérêt de telle ou telle structure.

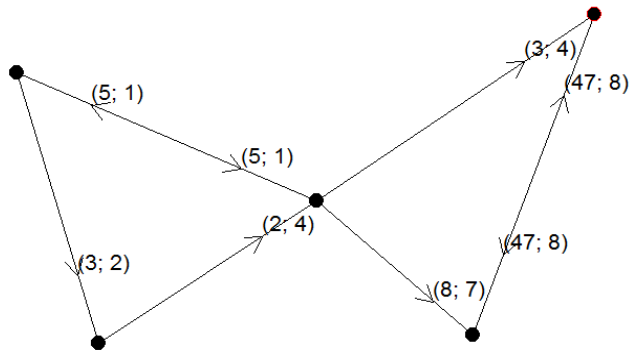
Voici au final l'arbre des structures possibles (22 types différents, mais des fonctions différentes



dans le cas non orienté).

Nous avons pensé à généraliser les structures trop tard dans le projet, donc les algorithmes ont uniquement été codés avec la structure en rouge, c'est à dire "(int * (nombre vect))list vect".
Cependant nous avons commencé à coder les autres types de codages possibles.
Le premier nombre d'un vecteur représente la capacité de l'arc, le second nombre représente le coût unitaire de passage. Les vecteurs utilisés comme étiquettes sont toujours de taille 2, même si la seconde case n'est utilisée que pour les algorithmes prenant le coût en compte.

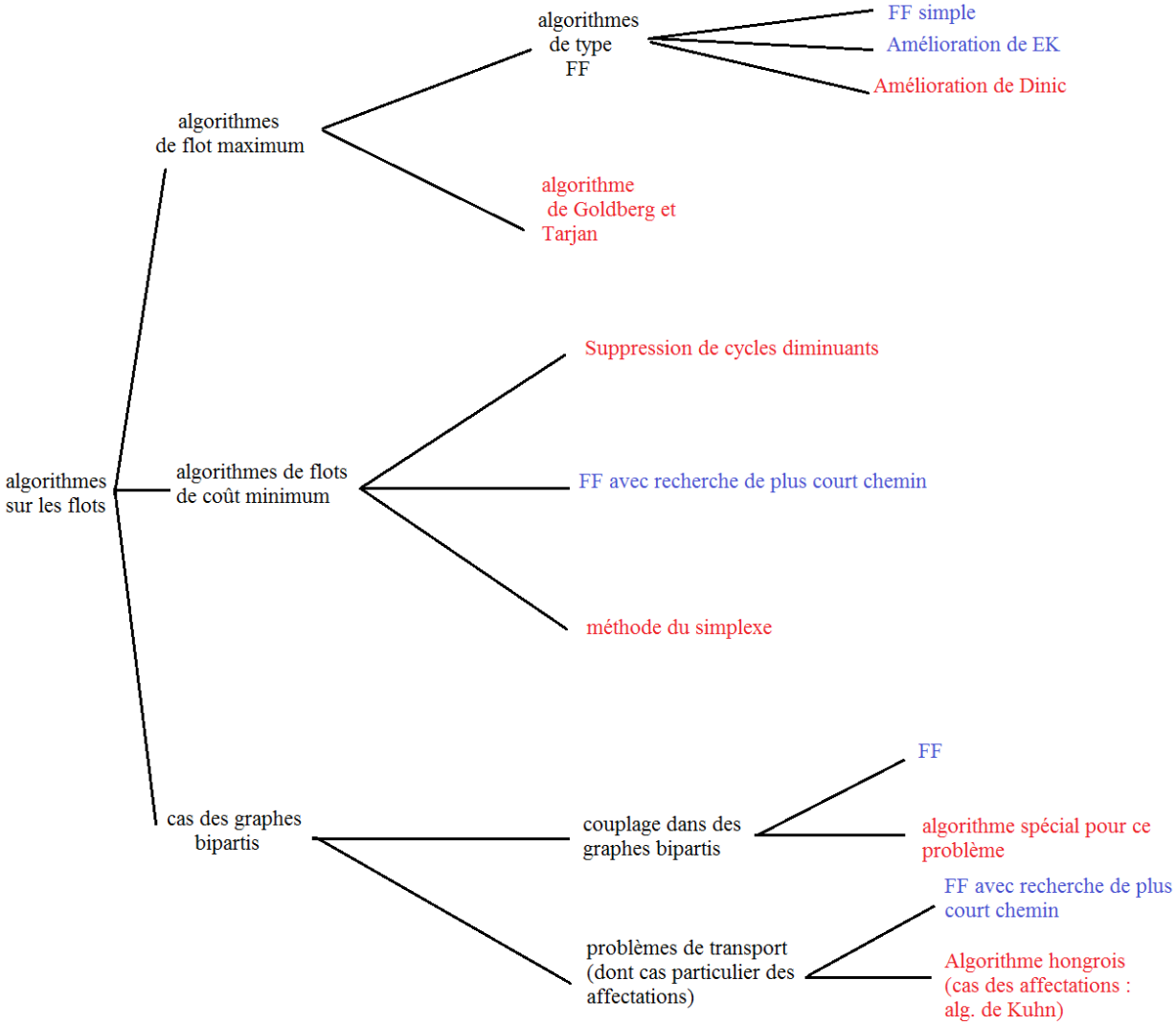
Comme exemple, le graphe suivant a pour représentation en tant que dessin (avec les données graphiques) :




```
{tai = 5;
  arc =
   [|[1, [|ent 8; ent 7|]; 3, [|ent 5; ent 1|]; 4, [|ent 3; ent 4|]];
     [4, [|ent 47; ent 8|]]; [0, [|ent 2; ent 4|]];
     [2, [|ent 3; ent 2|]; 0, [|ent 5; ent 1|]]; [1, [|ent 47; ent 8|]]|];
  pos = [|416, 298; 565, 167; 208, 159; 130, 423; 681, 480|];
  nom = [|"0"; "1"; "2"; "3"; "4"|]; col = [|0; 0; 0; 0; 0|]}
```

A6.2) Algorithmes non codés :

Pour résoudre les problèmes de flots de valeur maximale ou de coût minimum, il pourrait être intéressant de coder d'autres algorithmes, que nous n'avons pas pu réaliser faute de temps : voici un arbre de tous les algorithmes existants pour les problèmes de flots ou résolubles en utilisant les flots, ceux en bleu ayant été codés et pas ceux en rouge.
En particulier certains algorithmes comme utilisent la programmation linéaire, comme la méthode hongroise qui sert pour résoudre le problème des transports plus vite qu'avec FF.


algorithmes sur les flots
algorithmes de flot maximum
algorithmes de type FF
FF simple
Amélioration de EK
Amélioration de Dinic
algorithme de Goldberg et Tarjan
algorithmes de flots de coût minimum
Suppression de cycles diminuants
FF avec recherche de plus court chemin
méthode du simplexe
cas des graphes bipartis
couplage dans des graphes bipartis
FF
algorithme spécial pour ce problème
problèmes de transport (dont cas particulier des affectations)
FF avec recherche de plus court chemin
Algorithme hongrois (cas des affectations : alg. de Kuhn)

A6) Sources

- Sources utilisées

- Kenneth H. Rosen, John G. Michaels, "Handbook of discrete and combinatorial mathematics" (p.640-679, surtout 663-679)
- F. Droesbeke, "Les graphes par l'exemple" (p.179-251 et 124-125)
- Ford, Fulkerson , "Flows in Networks", Aout 1962, www.rand.org/pubs/reports/2007/R375.pdf
- Jørgen Bang-Jensen and Gregory Gutin, "Digraphs: Theory, Algorithms and Applications" (p.95-170 de la 1ère édition), http://www.math.cmu.edu/ãdudek/21-801/jensen_gutin.pdf
- Alexander Schrijver (= Lex Schrijver), "A Course in Combinatorial Optimization", http://homepages.cwi.nl/~lex/files/agtco.pdf
- Bernhard H. Korte, Jens Vygen, "Combinatorial optimization: theory and algorithms" (p.157-213 ou 165-226 dans la 4ème édition) www.averescu.com/facultate/ag/fac/Combinatorial_Optimization.pdf
- Claude Berge, "Théorie des graphes et ses applications"
- http://courses.csail.mit.edu/6.854/current
  Lectures 6,7,8,9. Surtout les "raw notes" des lectures 6 et 8.

- Articles de référence (non tous utilisés)

- Ford, Fulkerson , "Flows in Networks", Aout 1962
- A. V. Goldberg, R. E. Tarjan, "A new approach to the maximum flow problem" , 1986
- Ravindra K. Ahuja, Thomas L. Magnanti, James B. Orlin, "Network flows: theory, algorithms, and applications", 1993