Compilation

Jules Chouquet

SOM2IF15 – 2026

Point d'étape

On a fini de décrire les étapes principales du compilateur. C'est-à-dire qu'avec ce qu'on a vu on est en mesure de compiler un langage simple vers de l'assembleur.
Mais, on a laissé plein de choses sur le côté :

- Compiler des langages moins simples (objet, héritage, fonctionnel, réactif...)
- Gérer des dépendances entre fichiers source (édition de liens)
- Optimisation de l'utilisation de la mémoire
  - Ramasse-miettes
  - Allocation de registres

Dans ce cours

On va parler de l'allocation de registres.

De quoi s'agit-il ?

Rappelez-vous de la traduction des cadres d'activation :

Dans le bébé compilateur qu'on a construit

On a utilisé la pile pour stocker le contenu des variables temporaires.

De quoi s'agit-il ?

Rappelez-vous de la traduction des cadres d'activation :

Dans le bébé compilateur qu'on a construit

On a utilisé la pile pour stocker le contenu des variables temporaires.

Dans la vraie vie

Il y a un certain nombre de registres qui ne sont pas utilisés par le code assembleur. C'est dommage. (pourquoi ?)

Comment régler ça simplement ?

Idée : placer les valeurs des variables temporaires dans les registres.

Approche basique

- quand une valeur est affectée à une variable, on place son contenu dans un registre (s'il en reste de dispo, sinon sur la pile, tant pis. . .)
- quand elle n'est plus utile, on libère le registre.
- Comme ça on utilise la pile le moins possible.

Comment régler ça simplement ?

Idée : placer les valeurs des variables temporaires dans les registres.

Approche basique

- quand une valeur est affectée à une variable, on place son contenu dans un registre (s'il en reste de dispo, sinon sur la pile, tant pis. . .)
- quand elle n'est plus utile, on libère le registre.
- Comme ça on utilise la pile le moins possible.

Qu'est-ce qui ne va pas avec cette approche ?

Rien, c'est exactement ce qu'il faut faire, mais c'est pas si évident.

Où cela se passe-t-il ?

Dans le code du cours précédent, on a traduit les frames de la repr. int. en utilisant parfois des registres, parfois la pile.
$\rightarrow$ il faut donc une étape intermédiaire à cette traduction.

Quel langage ?

Représentation intermédiaire + résolution des appels/frames et évaluation des expressions
Assembleur - références explicites à la pile ou aux registres.

Code trois adresses

a.k.a. pré-assembleur

`delta = b * b - 4 * a * c`
sera décomposée en :
`01 : t0 = b × b`
`02 : t1 = 4 × a`
`03 : t2 = t1 × c`
`04 : delta = t0 - t2`
(plein de nouveaux temporaires sont crées pendant ce type de décompositions)

En résumé

- Affectations et opérations arithmétiques (supportant au plus deux opérandes)
- Sauts conditionnels et inconditionnels
- Étiquettes délimitant des blocs de code

Première difficulté

Savoir quand le contenu d'une variable est utile ou non

Analyse de vie

Donnée : Le programme (code trois adresses) sous forme de flot de contrôle.
Sortie : Le graphe de flot de contrôle étiqueté (arcs) par les variables vivantes.

Ensuite

Mais la c'est facile

L'analyse de vie nous permet d'extraire une information cruciale : Pour deux variables temporaires x et y, on peut savoir si elles sont utiles (vivantes) en même temps.
$\rightarrow$ dans ce cas on ne leur associe pas le même registre, évidemment.

Ensuite

Mais la c'est facile

L'analyse de vie nous permet d'extraire une information cruciale : Pour deux variables temporaires x et y, on peut savoir si elles sont utiles (vivantes) en même temps.
$\rightarrow$ dans ce cas on ne leur associe pas le même registre, évidemment.

Le graphe d'interférences

$V =$ Temporaires du programme.
$E = \{(x, y) \mid$ x et y sont vivantes en même temps

[exemple]

Deuxième difficulté

La pire en fait

Si on reformule notre problème maintenant : il faut associer un registre à chaque variable, mais deux variables adjacentes ne peuvent pas être associées au même registre.

Autrement dit

Colorer le graphe (registre = couleur)
$\rightarrow$ chaud, car problème NP-complet

Et du coup ?
$\rightarrow$ on va utiliser des algos de coloration qui ne sont pas parfaits, forcément, mais qui fonctionnent dans les grandes lignes.

Plan d'action

- Représenter le programme sous forme de graphe de flot de contrôle
- Procéder à l'**analyse de vie** des variables
- Construire le graphe d'interférence
- Établir une coloration du graphe
- En déduire une affectation des registres réels aux registres fictifs.

Le graphe de flot de contrôle [1]

1. cfg : control-flow graph

On part de la représentation intermédiaire (pré-assembleur)

Idée

$V =$ instructions.
$E = \{(i, j) | \text{j peut suivre immédiatement i dans le programme}\}$
$\rightarrow$ par séquence ou par `goto`

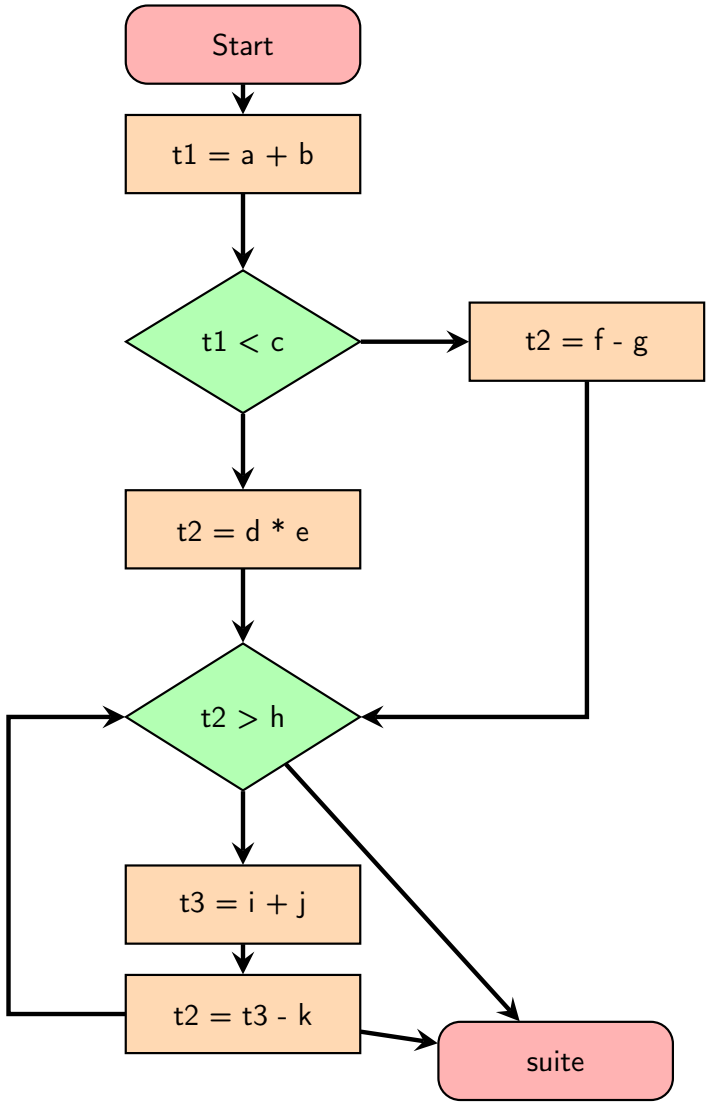
Attention

Parfois on définit le CFG à partir de "blocs" délimités par des étiquettes et des "goto". (i.e. on amalgame les arcs successifs). Pas nous, ça pourrait compromettre la pertinence de l'analyse de vie.

Question : quelles propriétés de graphe ?

Exemple

```
 1: t1 = a + b
2: if t1 < c goto 5
3: t2 = d * e
4: goto 6
5: t2 = f - g
6: if (t2 > h) goto 10
7: t3 = i + j
8: t2 = t3 - k
9: goto 6
10: -suite-
```



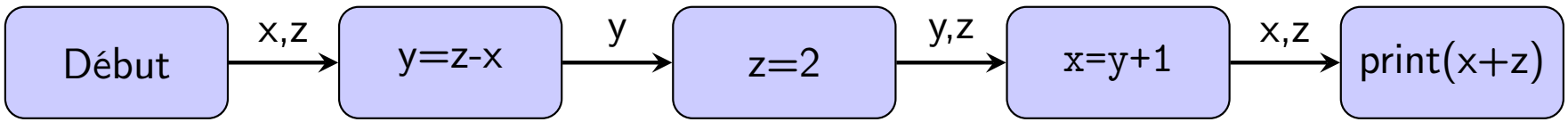
Analyse de vie

Principe — rappel

- Analyser le cfg pour savoir quand deux variables sont utiles en même temps.
- On dit qu'une variable est vivante le long d'un arc du cfg si elle doit être lue par une instruction future avant d'être (ré)-affectée

Idée principale : une variable est vivante entre sa définition et son utilisation. Elle est morte après sa dernière utilisation, et avant ses (ré)affectations.

Lire le graphe avec la technique du rebrousse-chemin



Résurrection de x
Mort de y
Naissance de z

Mort de z

mort de x
naissance de y

Définition et utilisation des variables

Variables définies

`def(s)` est l'ensemble des variables qui sont définies au sommet `s` du cfg

Variables utilisées

`use(s)` est l'ensemble des variables qui sont utilisées au sommet `s` du cfg

Les ensembles *in* et *out*

Pour tout sommet s du graphe de flot de contrôle, on définit :

- $in(s)$ l'ensemble des variables qui sont vivantes sur les arcs entrants dans s
- $out(s)$ l'ensembles des variables qui sont vivantes sur les arcs sortants de s.

Le but de l'analyse de vie est donc de calculer ces ensembles pour tous les sommets du graphe.

Définition équationnelle

- $in(s) = use(s) \cup out(s) \backslash def(s)$
- $out(s) = \bigcup_{s' \in suc(s)} in(s')$

Analyse de Vie des Variables

Algorithm 1: Analyse de Vie des Variables

Input: CFG = (V,E)

Output: Les intervalles de vie des variables

```
1 for s in V do
2     in(s)={}
3     out(s)={}
4 while in'≠in do
5     for s in V do
6         in'(s)=in(s)
7         in(s)=use(s)∪(out(s) - def(s))
8         out(s)=∪_{s'∈suc(s)} in(s')
```

Exemple

		def	use
1	t4 = 10	t4	
2	L0 :		
3	t3=5	t3	
4	t1 = t4 - 1	t2	t4
5	t0 = 1	t0	
6	if t3 < t1 goto L2		t3,t1
7	t0 = 0	t0	
8	L2 :		
9	if t0 = 0 goto L1		t0
10	write t3		t3
11	t2 = t3 + 1	t2	t3
12	t3 = t2	t3	t2
13	goto L0		
14	L1 : end		

Construction du Graphe d'Interférence

Définition

Deux variables x,y **interfèrent** si elles sont vivantes sur un même arc.

Construction du Graphe d'Interférence

Définition

Deux variables x,y **interfèrent** si ~~elles sont vivantes sur un même arc.~~
S'il existe s tel que $x \in$def(s) et $y \in$out(s).

Construction du Graphe d'Interférence

Définition

Deux variables x,y **interfèrent** si ~~elles sont vivantes sur un même arc.~~
S'il existe s tel que $x \in$def(s) et $y \in$out(s).

Construction du graphe d'interférence G=(V,E) :
V=variables, E=$\{\{x, y\} \mid$ x et y interfèrent $\}$
$\rightarrow$ on peut passer à la coloration

Algorithmes de Coloration

Faute de mieux...

Quasi-coloration de G

k-coloration d'un sous-graphe de G. Les sommets non colorés se voient attribuer une quasi-couleur spécifique π. Les sommets quasi-colorés peuvent être adjacents.

Idée :

- les sommets colorés utiliseront les k registres.
- Les sommets quasi-colorés utiliseront la pile.

Utile dans les cas où :

- Le nombre chromatique de G est inférieur au nombre de registres disponibles.
- On utilise un algorithme linéaire ou polynomial, qui ne trouvera pas à tous les coups la coloration optimale.

Coloration Gloutonne

Algorithm 2: Coloration Gloutonne

Input: Un graphe G avec un ensemble de sommets V et des arêtes E
Output: Une coloration valide du graphe

```
1 for chaque sommet v ∈ V do
2     Colorer le sommet v avec la première couleur disponible qui n'est
        pas utilisée par ses voisins;
```

Coloration Gloutonne

Algorithm 3: Coloration Gloutonne

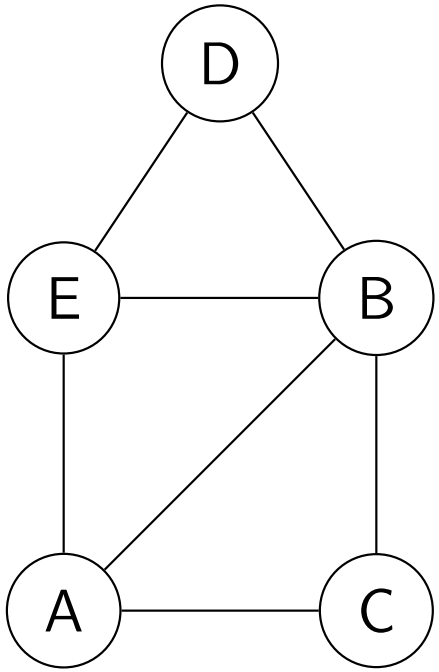
Input: Un graphe G avec un ensemble de sommets V et des arêtes E
Output: Une coloration valide du graphe

```
1 for chaque sommet v ∈ V do
2     Colorer le sommet v avec la première couleur disponible qui n'est
        pas utilisée par ses voisins;
```

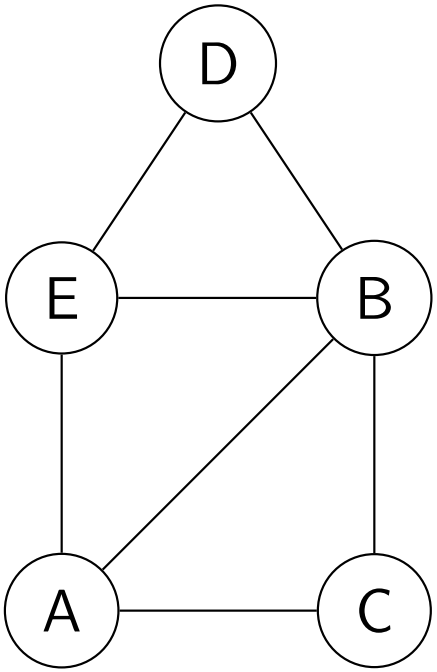
Quelques questions se posent :

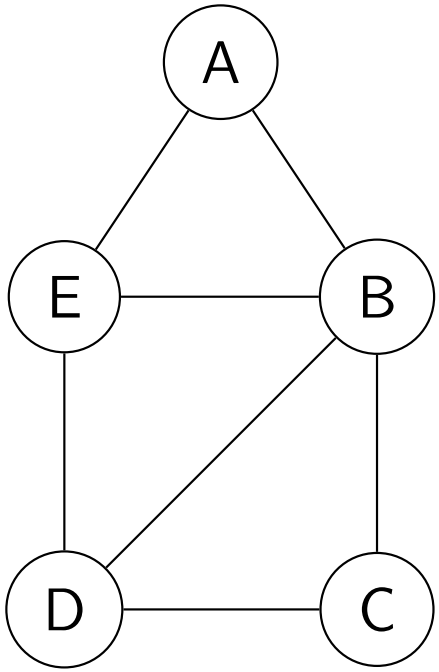
- Dans quel cas est-il efficace ?
- Dans quel ordre parcourir les sommets ?

Exemple



Exemple





Autre approche : simplification

Observation (Chaitin)

Si $v \in V$ a un degré $< k$, et que l'on parvient à k-colorier $G \backslash \{v\}$, alors G est k-coloriable.

Algorithme de Chaitin-Briggs - Pseudocode

Algorithm 4: Algorithme de Chaitin-Briggs

Input: Un graphe d'interférence $G = (V, E)$, $k \in \mathbb{N}$

Output: Une quasi-coloration

```
1  Colorer(G) :
2      Si V = ∅ alors
3          retour appelant
4      Si ∃v ∈ V t.q. deg(v) < k alors
5          Colorer(G\{v})
6          Affecter couleur à v // toujours possible car deg(v) < k
7      Sinon
8          Choisir v ∈ V
9          Colorer(G\{v})
10         Affecter π à v
```

Algorithme de Chaitin-Briggs - Pseudocode

Algorithm 5: Algorithme de Chaitin-Briggs

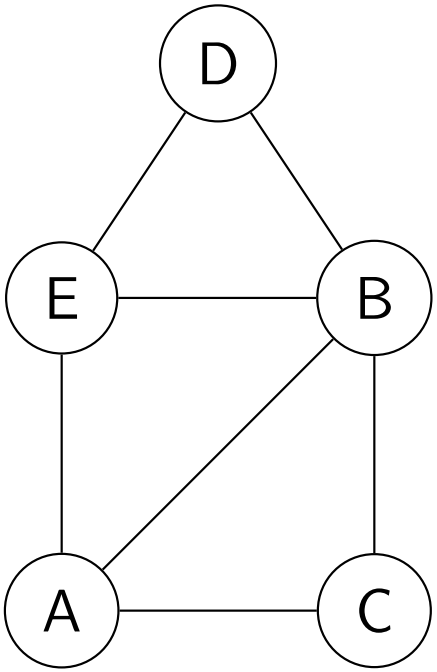
Input: Un graphe d'interférence $G = (V, E)$, $k \in \mathbb{N}$

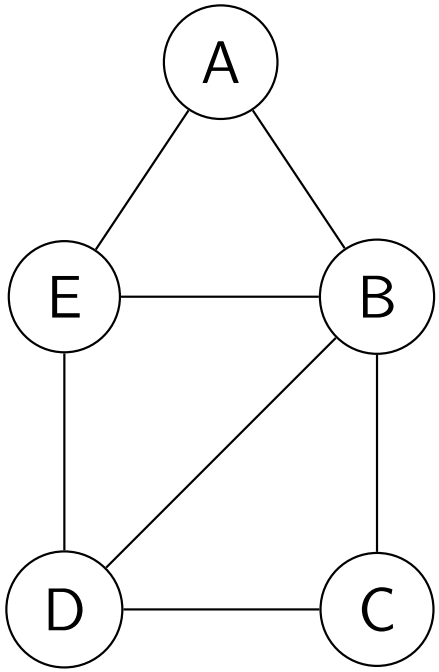
Output: Une quasi-coloration

```
1  Colorer(G) :
2      Si V = ∅ alors
3          retour appelant
4      Si ∃v ∈ V t.q. deg(v) < k alors
5          Colorer(G\{v})
6          Affecter couleur à v // toujours possible car deg(v) < k
7      Sinon
8          Choisir v ∈ V
9          Colorer(G\{v})
10         Affecter π à v
```

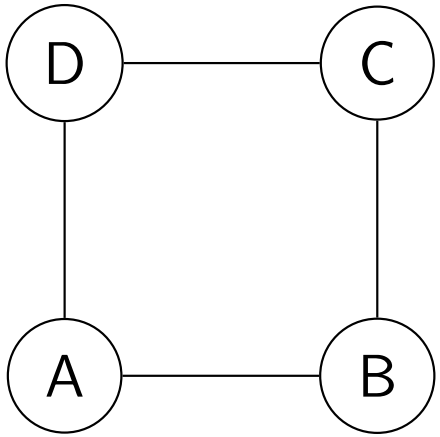
Complexité ?

Exemple





Exemple

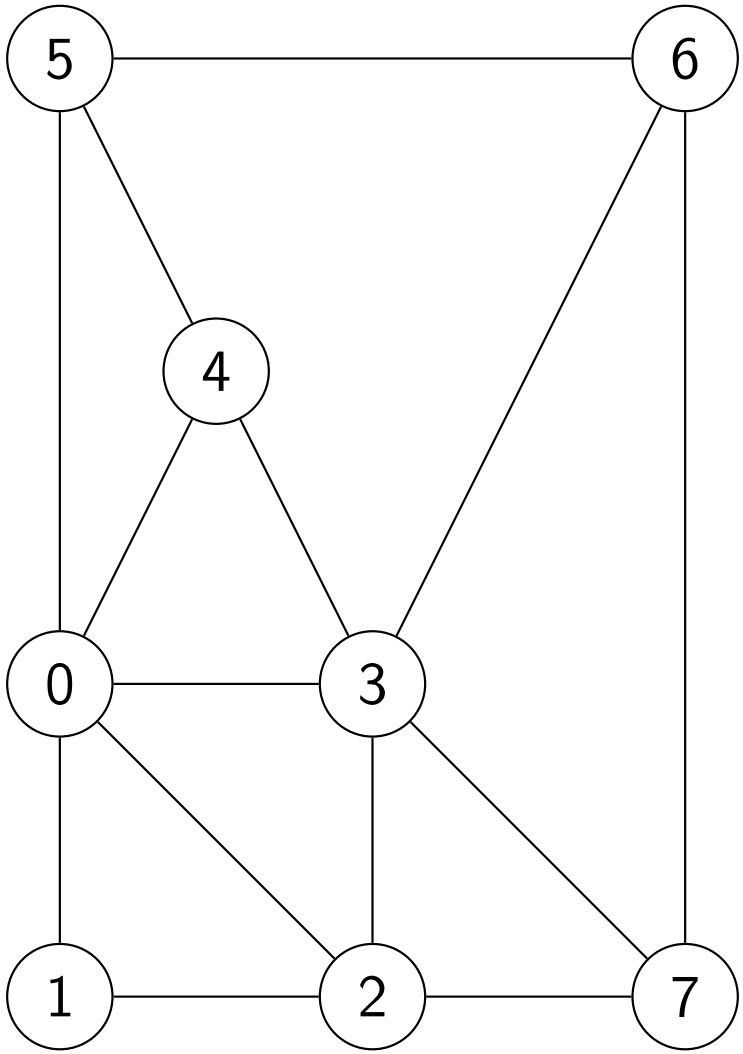


Algorithme de Chaitin-Briggs — Optimiste

Input: Un graphe d'interférence $G = (V, E)$, $k \in \mathbb{N}$
Output: Une quasi-coloration

```
1  Colorer(G) :
2      Si V = ∅ alors
3          retour appelant
4      Si ∃v ∈ V t.q. deg(v) < k alors
5          Colorer(G\{v})
6          Affecter couleur à v // toujours possible car deg(v) < k
7      Sinon
8          Choisir v ∈ V
9          Colorer(G\{v})
10         Si ∃ couleur c dispo pour v alors
11             Affecter c à v
12         Sinon
13             Affecter π à v
```

k=4 ? k=3 ?



Gestion des échecs de coloriage

Une possibilité :

- Si le temporaire x finit malgré tout avec la pseudo-couleur π, on choisit un emplacement mémoire m_x pour y associer la valeur de x.
- À chaque fois que x est utilisé dans le code originel, on introduit une **nouvelle** variable x_i. On chargera dans x_i la valeur dans m_x pour les utilisations de x, et on enregistrera x_i dans m_x pour les écritures dans x.

Les variables x_i ont une durée de vie très courtes, elles n'interfèrent pas avec les autres.
On peut relancer l'analyse de vie et la coloration...

Exemple

A. Appel

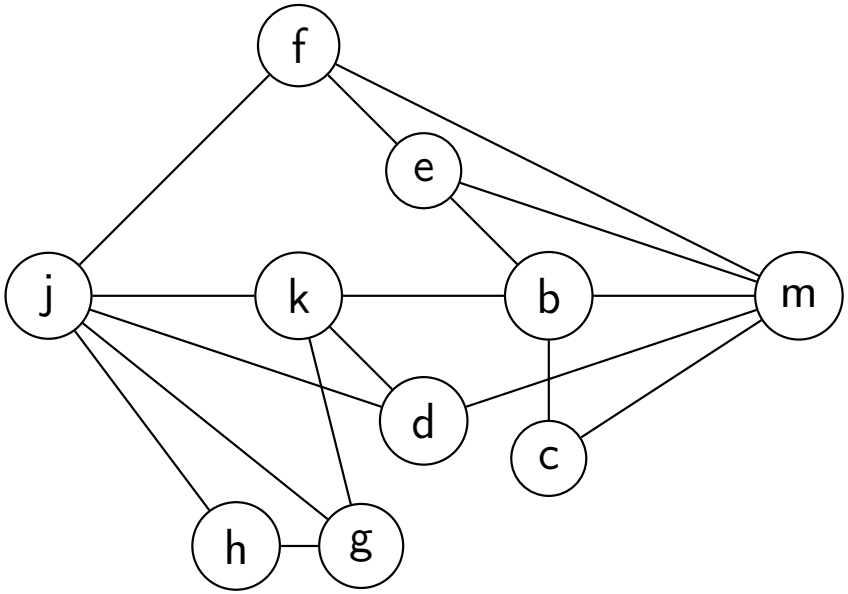
On n'a pas réussi à colorier m dans le code suivant :

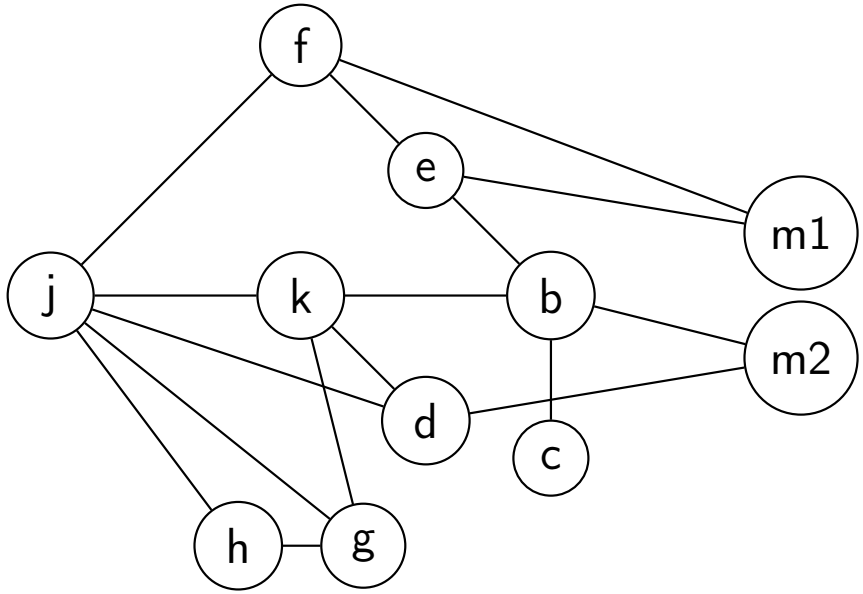
```
 g := mem[j+12]
h := k - 1
f := g * h
e := mem[j+8]
m := mem[j+16]
b := mem[f]
c := e + 8
d := c
k := m + 4
j := b
```

On réécrit le code, après avoir alloué l'espace mémoire M pour `m` :

```
 g := mem[j+12]
h := k - 1
f := g * h
e := mem[j+8]
m1 := mem[j+16]
mem[M]:= m1
b := mem[f]
c := e + 8
d := c
m2 := mem[M]
k := m2 + 4
j := b
```

Graphes d'interférence pour l'exemple précédent





Il y a quand même un problème : on introduit de nouveaux temporaires dans le graphe. Comment être sûr qu'on parviendra in fine toujours à terminer cette procédure ?

Deux approches

- Plutôt que de créer des nouveaux registres m_1 et m_2, on alloue **physiquement** deux registres destinés à cette tâche. On les déclare non allouables, ainsi ils n'interfèreront jamais.
- On fait comme on a dit, mais on interdit de colorier les registres crées avec π (terminaison)

Est-ce que c'est fini ?

Est-ce que c'est fini ?

Bien sûr que non.

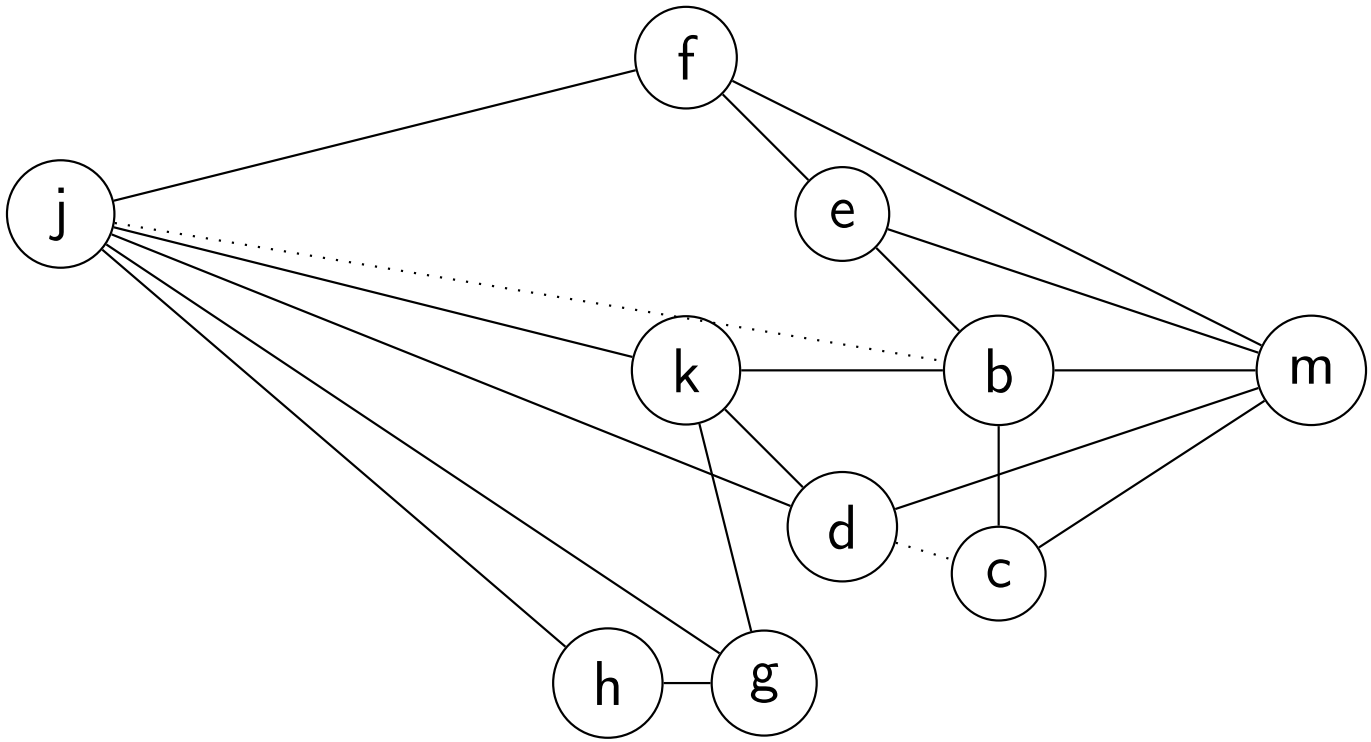
Est-ce que c'est fini ?

Bien sûr que non.

Observations sur les lignes suivantes

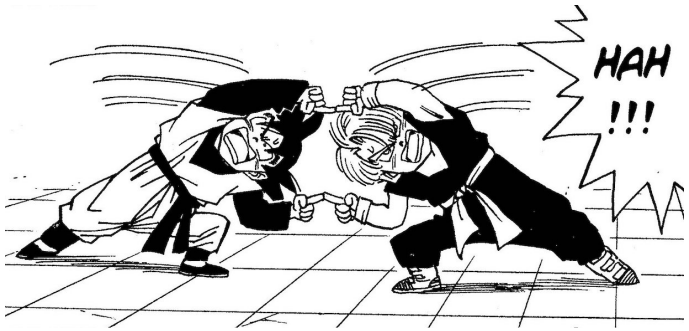
```
d := c
j := b
```

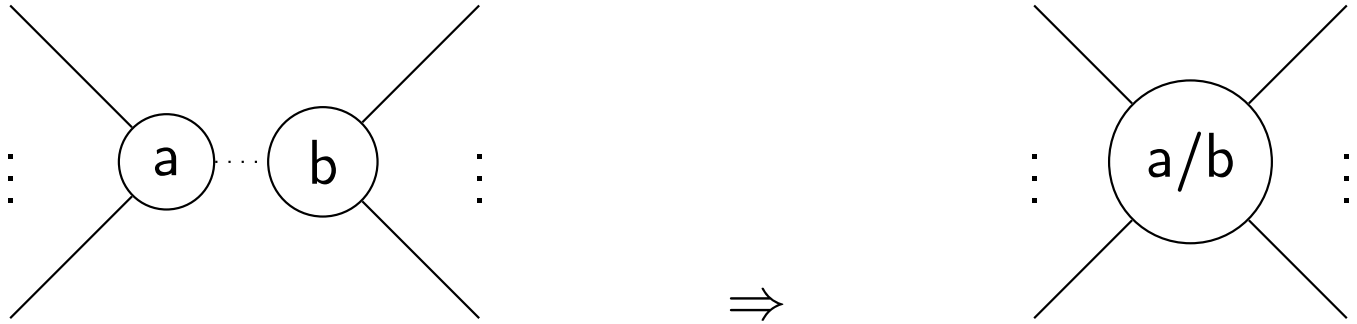
Le **vrai** graphe d'interférences



Coloration avec fusion

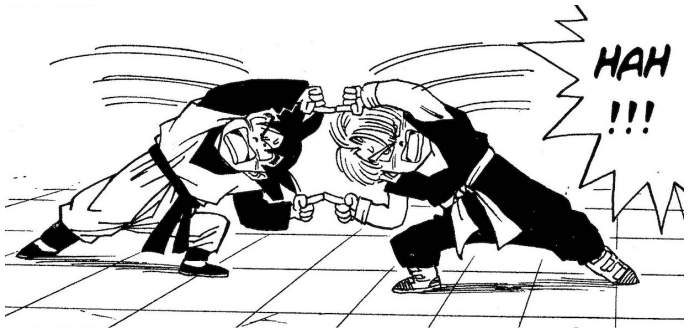
coalescing

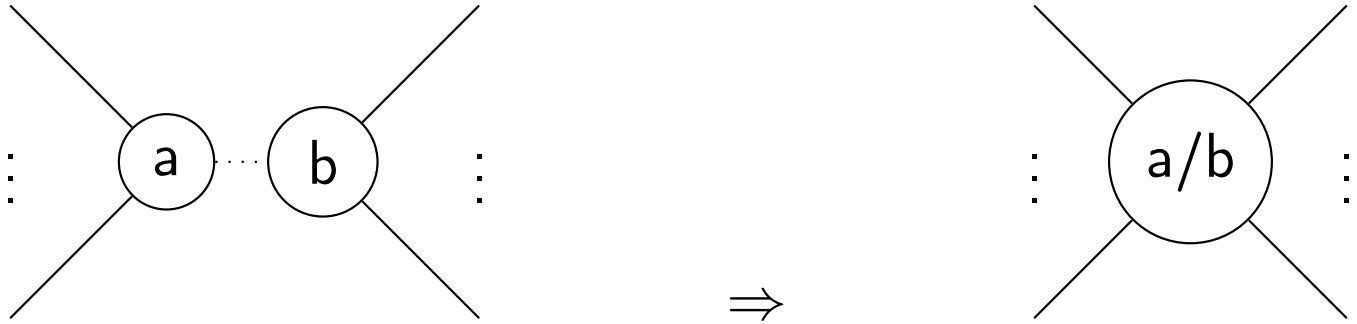




Coloration avec fusion

coalescing





Difficultés de la fusion agressive

- Augmentation du degré
- Chevauchement des arêtes de préférence et d'interférence

Fusion douce

Idée : on ne fusionne que si le sommet résultant reste trivialement coloriable (critère de Briggs), ou alors on ne fusionne x et y que si tout voisin non trivialement coloriable de x est également voisin de y (critère de George).
$\rightarrow$ dans les deux cas, la fusion préserve la k-colorabilité du graphe.

Fusion douce

Idée : on ne fusionne que si le sommet résultant reste trivialement coloriable (critère de Briggs), ou alors on ne fusionne x et y que si tout voisin non trivialement coloriable de x est également voisin de y (critère de George).
$\rightarrow$ dans les deux cas, la fusion préserve la k-colorabilité du graphe.

Remarque

La fusion peut aussi rendre un sommet trivialement coloriable.
Cela peut permettre de simplifier le graphe comme dans le déroulement de l'algorithme de Chaitin.

Fusion douce

Idée : on ne fusionne que si le sommet résultant reste trivialement coloriable (critère de Briggs), ou alors on ne fusionne x et y que si tout voisin non trivialement coloriable de x est également voisin de y (critère de George).
$\rightarrow$ dans les deux cas, la fusion préserve la k-colorabilité du graphe.

Remarque

La fusion peut aussi rendre un sommet trivialement coloriable.
Cela peut permettre de simplifier le graphe comme dans le déroulement de l'algorithme de Chaitin.

Remarque

La simplification peut rendre deux sommets non fusionnables.
Mais, il n'est pas intéressant de simplifier le graphe en supprimant une arête susceptible d'être fusionnée.

Comment on s'en sort ?

On alterne. Algorithme de George et Appel

```
1 Simplifier(G) :
2     Si x trivialement coloriable et ne porte aucune arête de préférence
       alors
3         Simplifier(G\{x})
4         Attribuer couleur à x
5     Sinon
6         Fusionner(G)
```

```
1 Fusionner(G) :
2     Si x et y peuvent être fusionnés alors
3         G' = G avec fusion de x et y.
4         Simplifier(G')
5     Sinon
6         Geler(G)
```

```
1 Geler(G) :
2     Si x est trivialement coloriable alors
3         G' = G privé des arêtes de préférence de x Simplifier(G')
4     Sinon
5         Spill(G)
```

```
1 Spill(G) :
2     Choisir x de coût minimal // peu utilisé, ou haut degré.
3     Simplifier(G\{x})
4     Si il reste couleur dispo pour x alors
5         La lui attribuer.
6     Sinon
7         attribuer π à x.
```