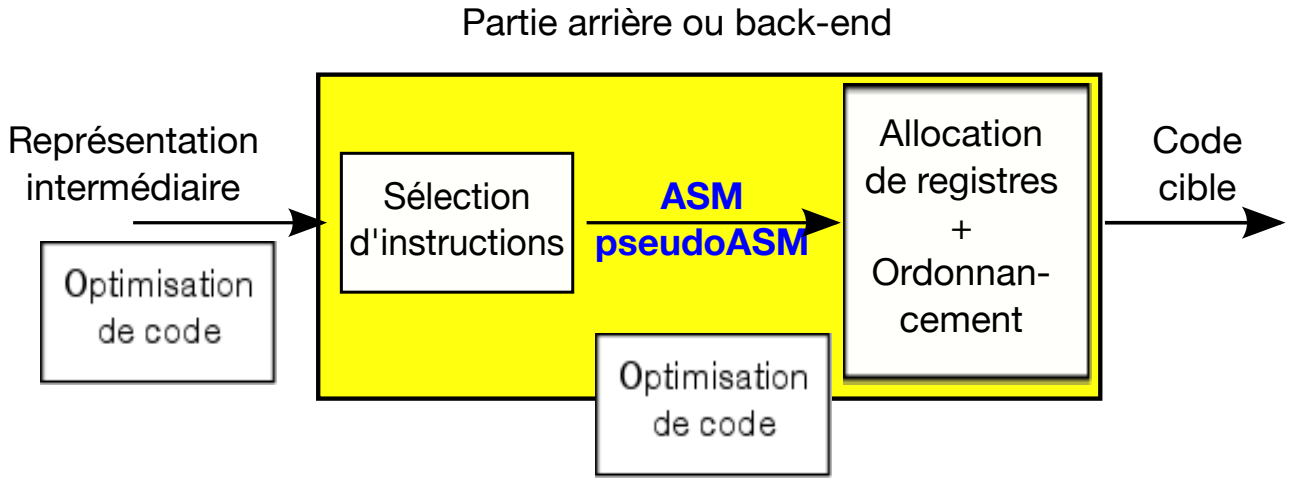
Allocation de registres

Cours 5. UE Compilation Avancée

Karine Heydemann
karine.heydemann@lip6.fr

Génération de code, allocation de registres et ordonnancement



- La génération de code produit une suite d'instructions (ou pseudo-instr) dans un ordre qui vérifie simplement la sémantique du programme
- Puis étapes clé dépendantes de l'architecture cible
  - L'allocation de registres qui assigne un registre de la cible aux opérandes source et destination des instructions
  - Ordonnancement des instructions qui consiste à changer l'ordre des instructions pour mieux exploiter le matériel disponible.

Sélection d'instructions et allocation de registres

- Génération de pseudo-code machine à partir du code intermédiaire : sélection des instructions parmi celles disponibles sur la cible, assignation de registres symboliques, nb illimité.
- Il est possible de produire un code machine utilisant peu de registres mais, nécessite beaucoup de transferts mémoire et de place mémoire (pile) : code volumineux et lent.

```
(1) t1 := x + y
(2) t2 := t1 + z
```

```
(1) la r1, @x
(2) load r2, 0(r1)   // y
(3) la r1, @y
(4) load r3, 0(r1)   // x
(5) add r1, r2, r3   // x + y
(6) la r2, @z
(7) load r3, 0(r1)   // z
(8) add r2, r1, r3
...
```

- L'allocation de registres a pour but d'assigner les registres de l'architecture cible de façon à limiter les transferts inutiles avec la mémoire.

Allocation de registres

- L'allocation de registres a pour but d'assigner les registres de l'architecture cible de façon à limiter les transferts avec la mémoire.
- Idéalement les variables vivantes du code assembleur abstrait en tout point du programme résident dans des registres (après leur définition/leur chargement on ne veut pas relire leur valeur en mémoire)
- Problème si plus de variables vivantes en un point que de registres => stockage en mémoire nécessaire, lecture au moment où on en a besoin (spill code).
- De nombreuses opérations de copie (move) sont générées dans le code (élimination redondance cf. cours 3), l'allocation de registres peut les éliminer en assignant le même registre aux opérandes source et destination d'un move.

Principe de l'allocation de registres par coloriage de graphe

- Approche globale à une procédure
- Construction du CFG puis analyse de la vivacité des variables en tout point du programme
- Annotation des arcs entre les instructions avec l'ensemble des variables vivantes sur cet arc
- Construction d'un graphe d'interférence : les noeuds sont les variables et les arêtes relient des variables qui interfèrent :
  - variables vivantes en même temps
  - contrainte exprimant qu'une variable ne peut être mise dans un registre `Ri`
- On tente de mettre dans des registres différents les variables qui interfèrent : coloriage de graphe avec moins de K couleurs si K registres disponibles
- Les noeuds qui interfèrent doivent avoir des couleurs différentes
- Si le graphe est K-coloriable pas de problème, sinon introduction de spill code

Coloring by simplification

- L'allocation de registres est un problème NP-complet (sauf arbre d'expression, sauf si SSA, sauf... mais cas général oui)
- Le coloriage de graphe aussi
- Mais heuristique linéraire donne de bons résultats en pratique
- Plusieurs phases :
  1. Build : contruction du graphe d'interférence
  2. Simplify : réduction de la taille du pb
  3. Spill : choix d'une variable à mettre en mémoire (spill code)
  4. Select : assignation des couleurs aux noeuds du graphe

Build : construction du graphe d'interférence

- Un noeud par variable
- Un arc entre 2 variables si elles "interfèrent", c'est-à-dire si elles sont vivantes en même temps en un point du programme
- Deux variables impliquées dans une copie sont reliées par un arc dédié

```
LIVEin = k, j
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
LIVEout = d, k, j
```

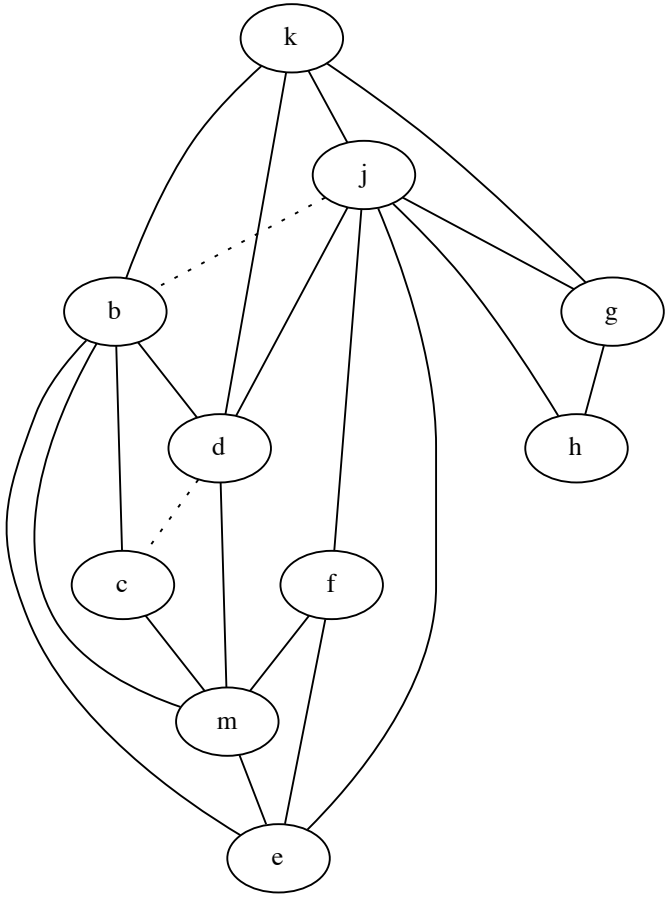
Build : exemple

```
LIVEin = k, j
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
LIVEout = d, k, j
```

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

Build : exemple

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



Le pourquoi de l'heuristique/simplify

- Coloriage : étant donné un graphe G et K couleurs, attribuer une couleur à chaque noeud de telle sorte que 2 noeuds adjacents n'aient pas la même couleur
- Si un noeud n est de degré (nb de voisins) $< K$ et si G - n est K-coloriable alors G est K-coloriable :
  il existe forcément une couleur disponible pour n qui a au plus $K - 1$ voisins on peut donc toujours le colorier
- Heuristique d'allocation de registres utilise ce résultat
- NB : si un noeud a plus de $K - 1$ voisins, K couleurs peuvent suffire : les voisins peuvent avoir la même couleur, cela dépend s'ils interfèrent ou non !

Simplify

- Retirer du graphe à colorier un noeud de degré $< K$ et l'empiler
- Cela réduit le dégré des noeuds restant, itération tant que c'est possible
- Si le graphe restant est vide, alors il est coloriable. On passe à l'étape Select qui va assigner une couleur aux registres
- Sinon, étape de Spill.
- Select : on reconstruit le graphe avec les noeuds qu'on retire un à un de la pile ; une couleur existe pour chacun sinon ils ne seraient pas sur la pile

Spill

- Si tous les noeuds ont un degré $\geq K$, la simplication s'arrête. On choisit un noeud pour le spilling.
- Spill optimiste : on enlève ce noeud, et comme un noeud spillé (toujours lu depuis la mémoire/écrit) n'interfèrent plus, on le met sur la pile et la simplication peut continuer.
- Lors de la phase Select, on trouve donc des noeuds sur la pile choisis pour le spilling...

Spill et Select

- Si un noeud qui avait été choisi pour le spilling a des voisins utilisant moins de K couleurs alors il peut être colorié, d'où le nom de spill optimiste !
- Mais un noeud choisi n'est pas forcément coloriable (ses voisins utilisent les K couleurs), dans ce cas c'est un spill réel
- La sélection continue pour identifier d'autres spills réels
- Si pas de coloriage des noeuds, réécriture du prog avec lecture mémoire avant toute utilisation et écriture après toute définition
- Cela introduit de nouveaux temporaires (avec durée de vie courte) qui interfèrent avec les noeuds du graphe : il faut recommencer à l'étape Build en prenant en compte le nouveau programme
- En pratique 1 à 2 itérations suffisent

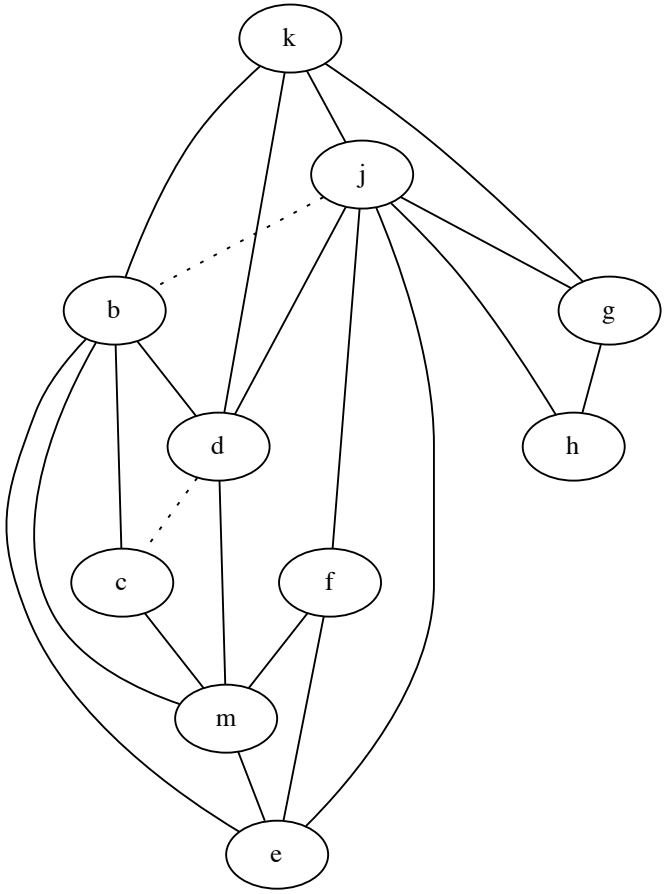
Coloriage de graphe et spill code

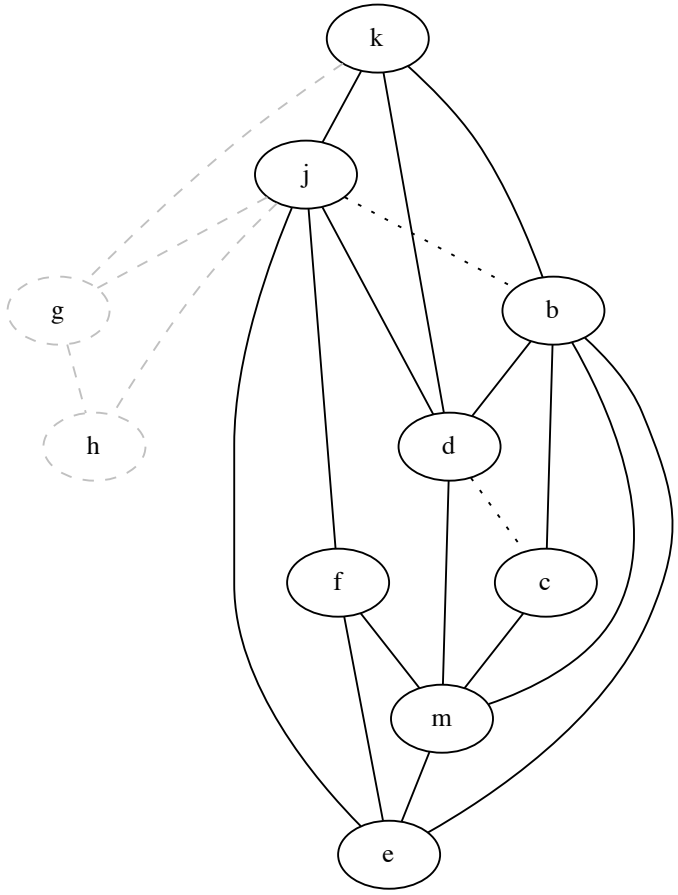
- Le choix du noeud à mettre sur la pile est important
- Pour que le choix soit efficace, noeud à fort degré
- Pour que le choix ne soit pas trop couteux, choisir un noeud pas trop utilisé (pb des boucles).
- Compromis à faire entre coût du spill et son efficacité.
- La durée de vie et la fréquence d'utilisation ne sont pas reliées ! On ne connait pas le nombre d'itérations des boucles par exemple, ni la fréquence des chemins résultant d'un saut conditionnel
- Heuristique basée sur nombre d'utilisations et définitions avec poids pour celles apparaissant dans des boucles

Simplify : exemple

Machine avec 4 registres
Moins de 4 voisins : g, h, f, c

Choix de g puis h
Pile = g, h



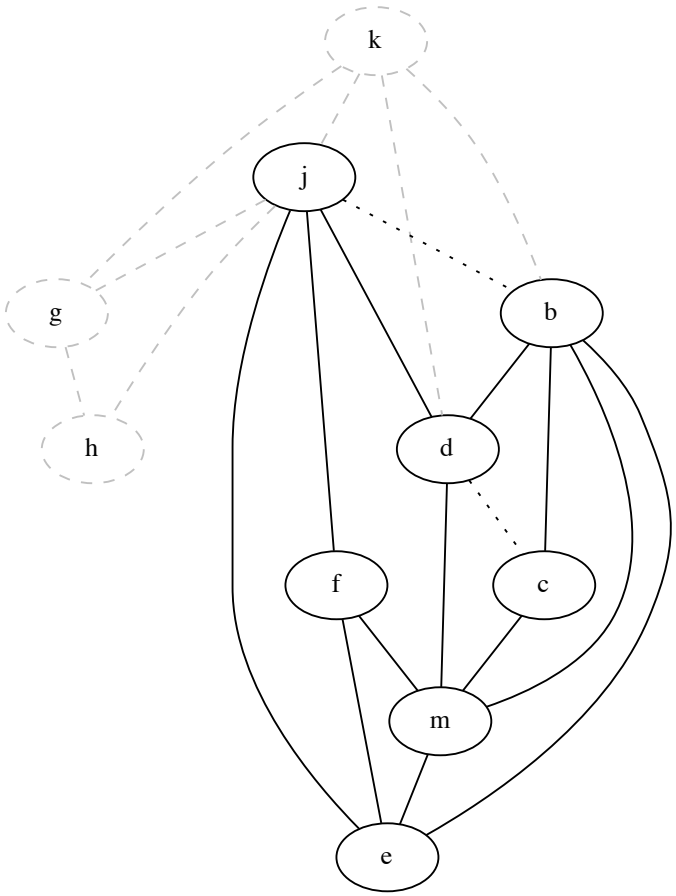


Simplify : exemple

Moins de 4 voisins : k, f, c

Choix de k

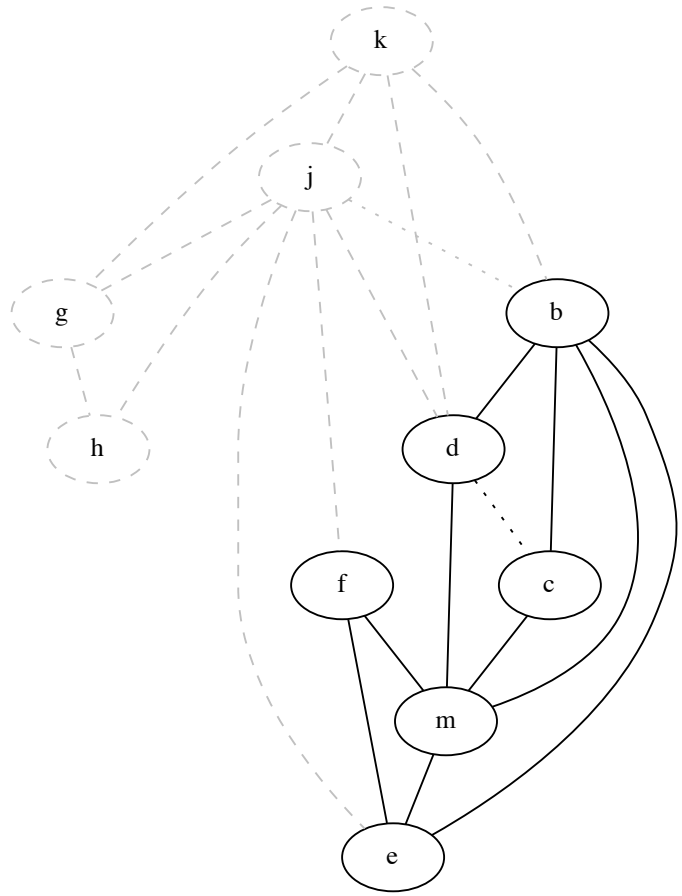
Pile = g, h, k



Moins de 4 voisins : j, f, c

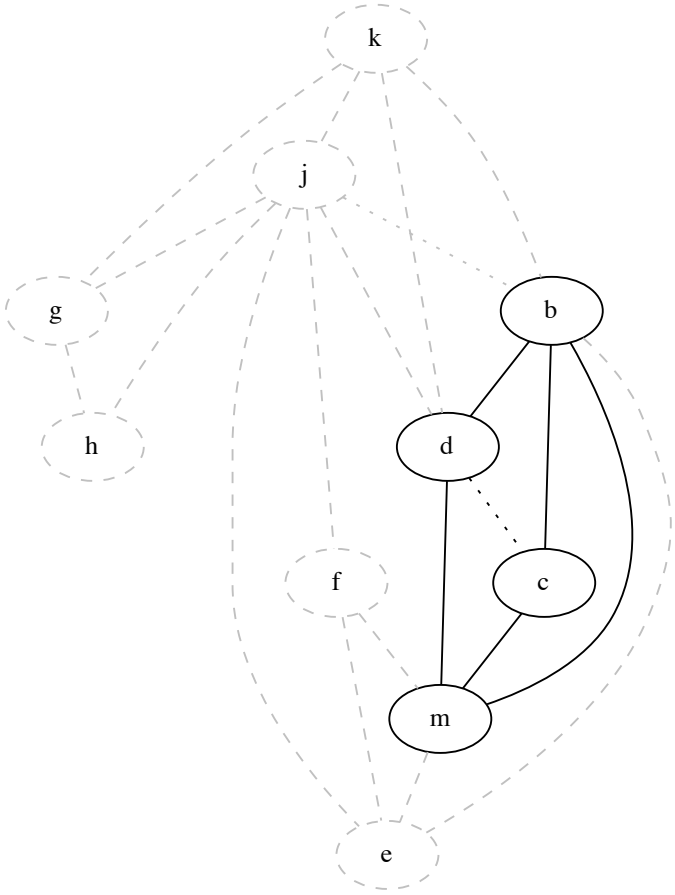
Choix de j

Pile = g, h, k, j

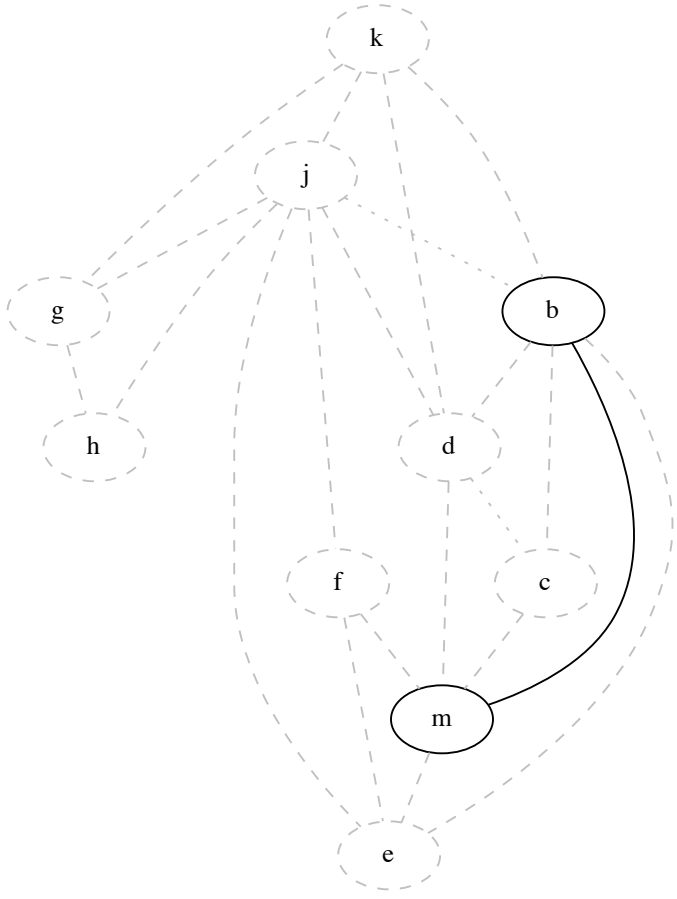


Simplify : exemple

Moins de 4 voisins : f, e, c, d
Choix de f, e
Pile = g, h, k, j, f, e,



Moins de 4 voisins : c, d
Choix de c, d
Pile = g, h, k, j, f, e, c, d

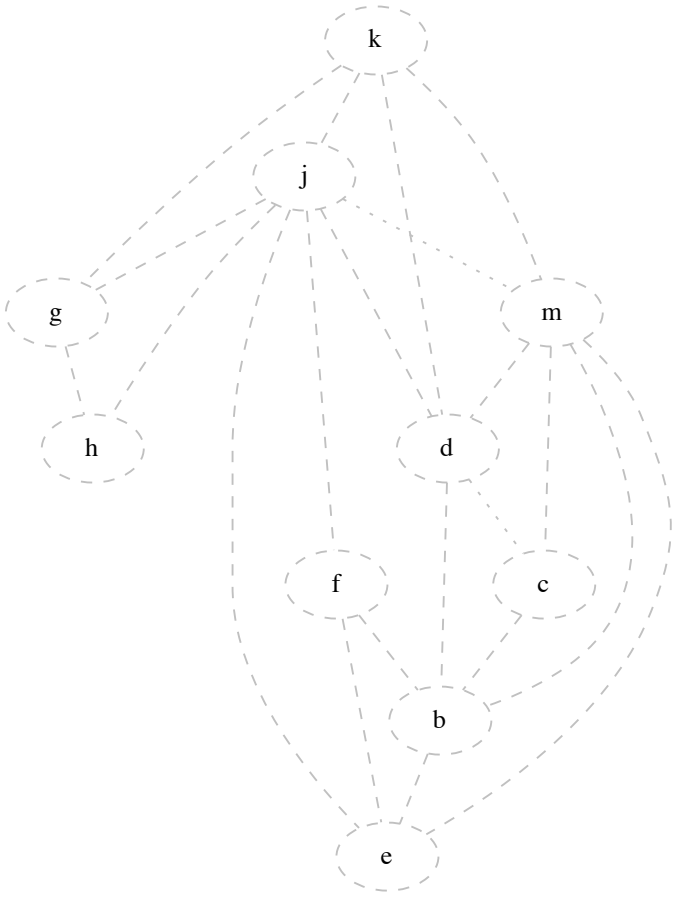


Simplify : exemple

Moins de 4 voisins : b, m

Choix de b, m

Pile = g, h, k, j, f, e, c, d, b, m



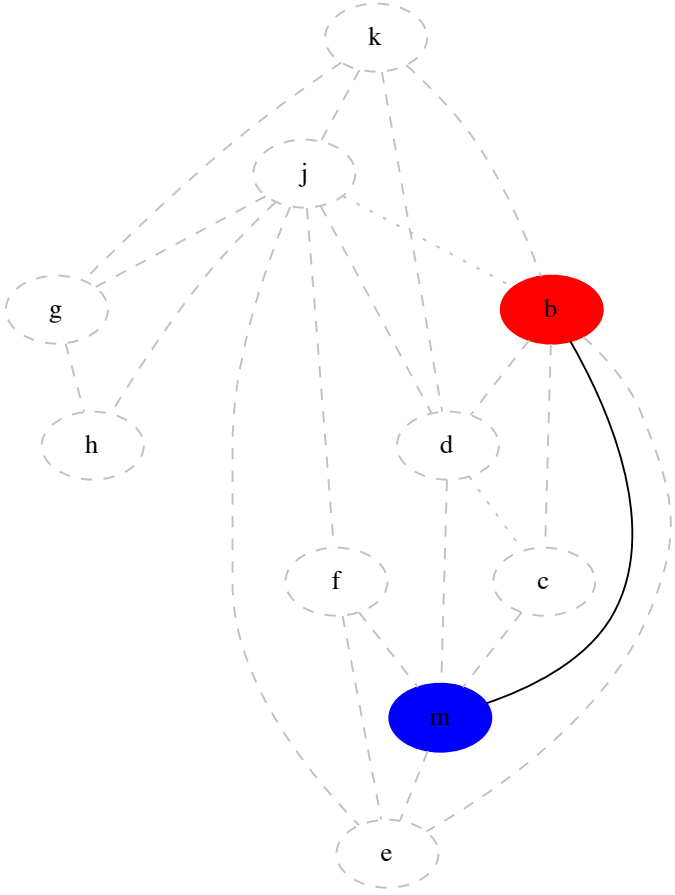
On passe à l'étape d'attribution des couleurs :

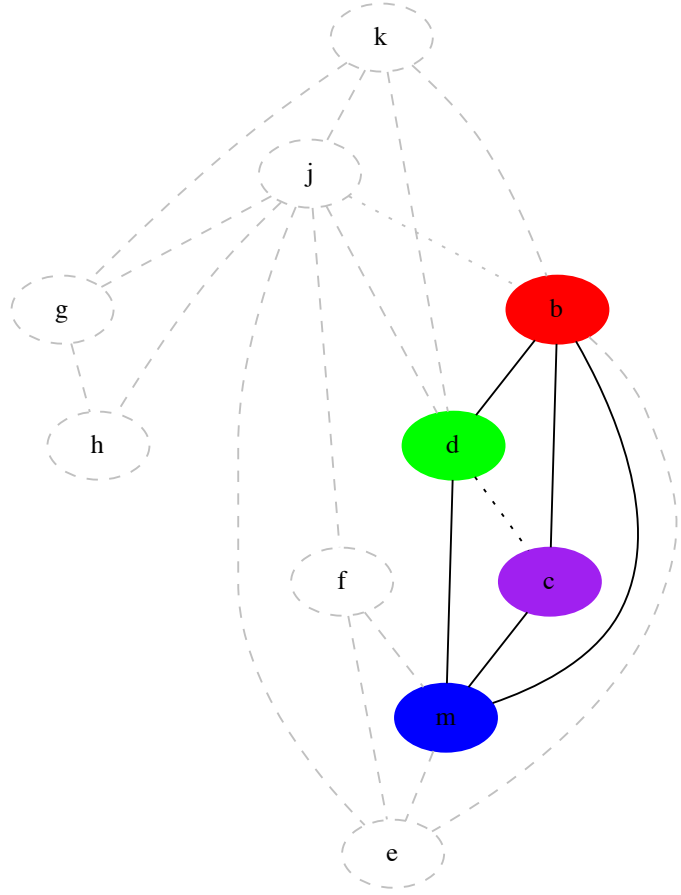
- On dépile un noeud
- On lui attribut une couleur compatible avec celles de ses voisins déjà coloriés

Select : exemple avec 4 couleurs (B, R, G, P)

Pile = g, h, k, j, f, e, c, d, b, m
m ← B, b ← R

Pile = g, h, k, j, f, e, c, d
d ← G, c ← P

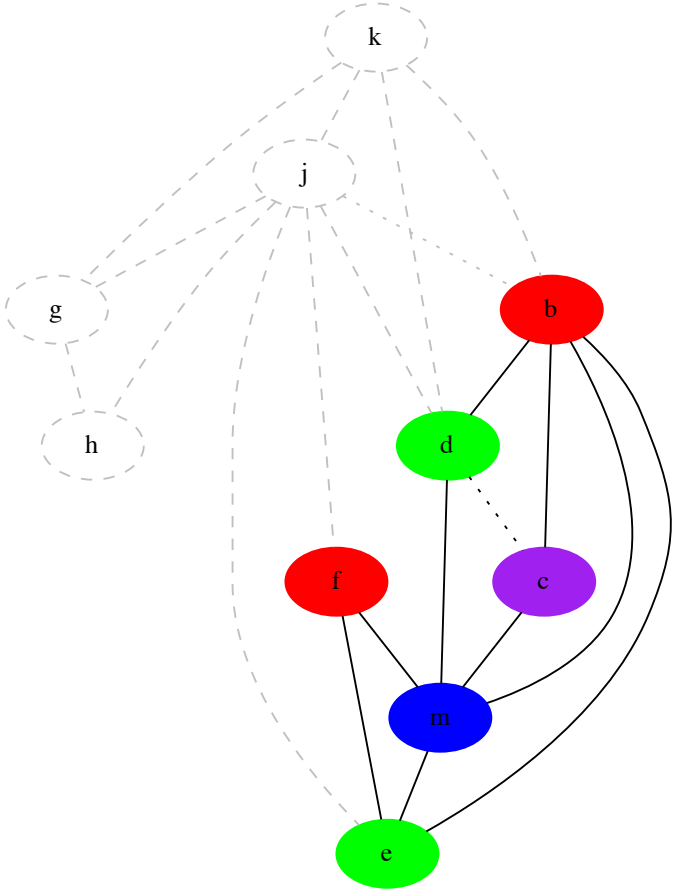




Select : exemple avec 4 couleurs (B, R, G, P)

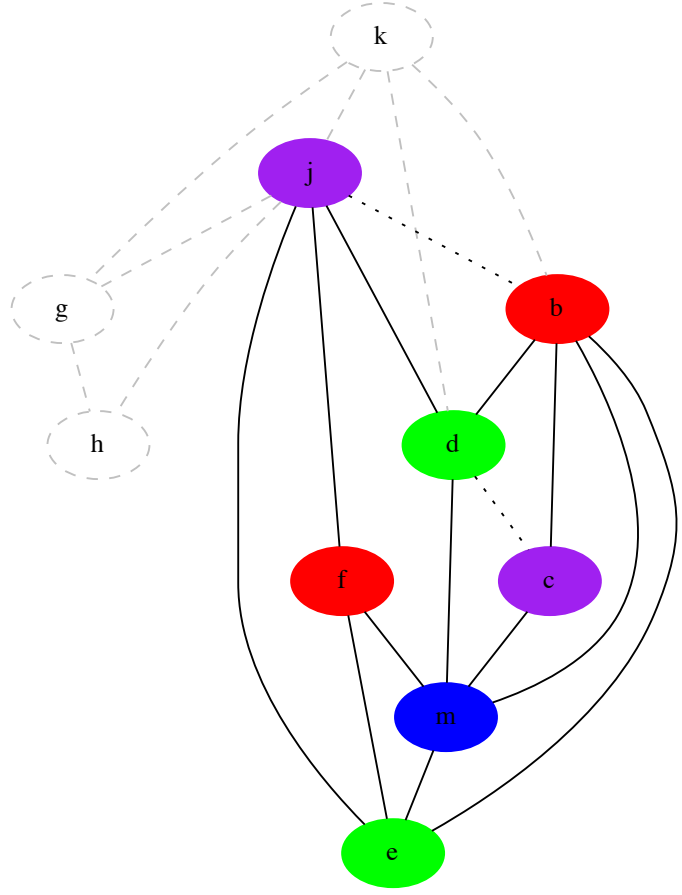
Pile = g, h, k, j , f, e

e ← G, f ← R



Pile = g, h, k, j

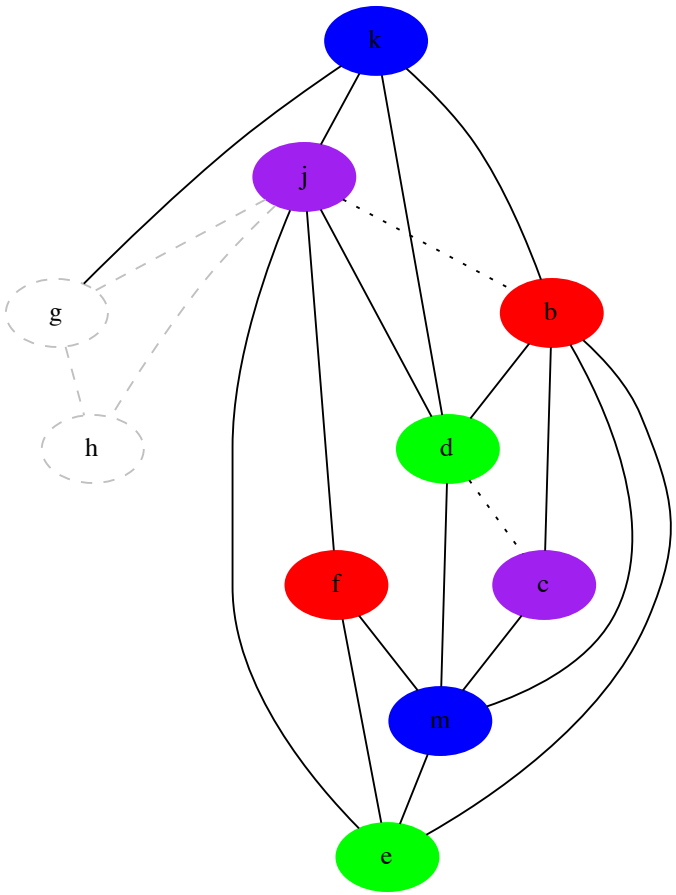
j ← P



Select : exemple avec 4 couleurs (B, R, G, P)

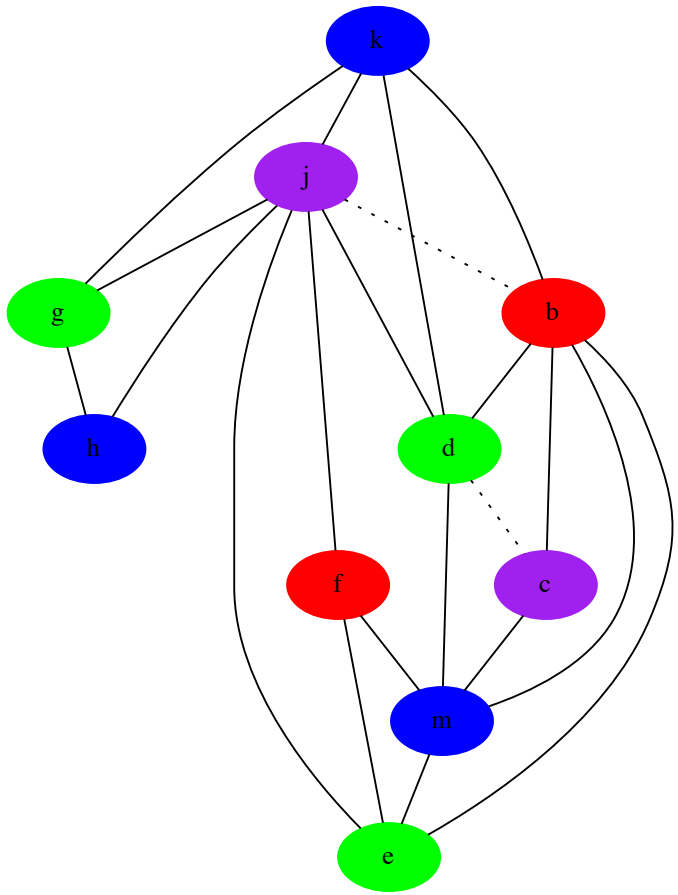
Pile = g, h, k

k ← B



Pile = g, h

h ← B, g ← G



Mais on n'a rien fait pour éliminer les instructions de copie...

Coalescing

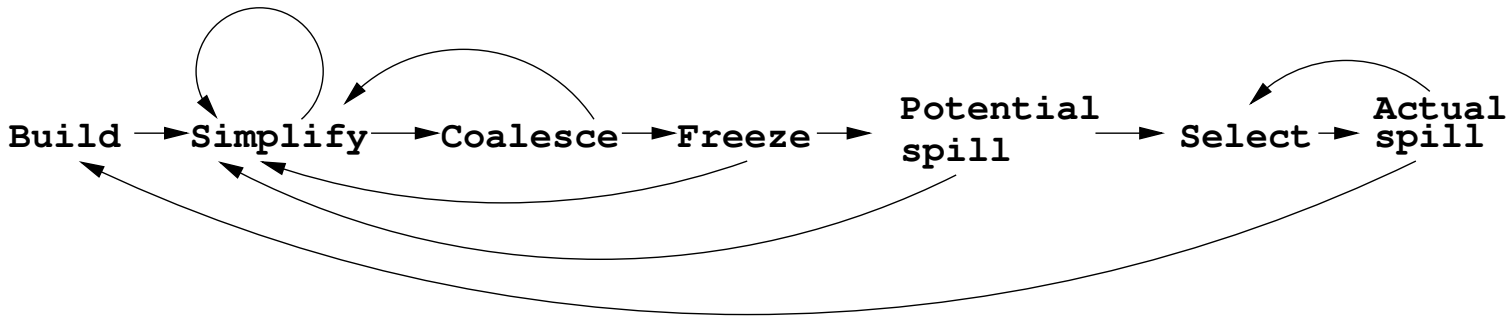
- S'il n'y a pas de lien entre Src et Dst d'une copie, on peut les réunir et éliminer la copie.
- Réunion en un nouveau noeud, les voisins sont l'union des ens. de voisins des noeuds initiaux
- En principe on peut toujours le faire, efficace pour éliminer les copies/instructions move
- Mais un noeud "union" est plus contraint, risque de ne plus pouvoir colorier le graphe résultant
- Objectif : réunion uniquement lorsqu'on est sûr que cela ne rendra pas le graphe non coloriable

Stratégies sures de coalescing

- Briggs
  - Réunion de *a* et *b*, ssi le noeud *ab* a moins de *K* voisins de degré $\geq K$
  - Raison : le graphe résultant sera aussi coloriable que celui de départ
  - Après simplify, le noeud *ab* sera connecté aux noeuds de degré $\geq$ K qui sont moins de *K*, donc il pourra être éliminé
  - Le graphe est donc coloriable s'il l'était au départ
- Georges
  - Réunion de *a* et *b* ssi tous les voisins de *a* interfèrent avec *b* ou sont de degré $< K$.
  - Raison : soit *S* est l'ensemble des voisins de a avec degré $<$ K dans le graphe *G*
  - Après simplify sans coalescing il reste un graphe *G*1 sans *S*
  - S'il y a du coalescing alors simplify enlève aussi *S* et résulte en *G*2
  - *G*2 est un sous-graphe de *G*1 (noeud *ab* dans *G*2 correspond à *b* dans *G*1)
  - Il est au moins aussi facile à colorier
- Stratégies conservatives (certains cas "safe" ne seront pas traités) mais mieux que spill !

Allocation avec coalescing

- Entrelacer les étapes de simplification et de coalescing conservatif permet d'éliminer la plupart des instr move tout en évitant spill code
- Schéma de l'algorithme complet



- Itération de phases de simplification et de coalescing
- Si plus de possibilité, phase de "freeze", au pire spilling

Allocation avec coalescing

- Build : construction du graphe, les noeuds sont soit move-related, soit non-move-related
- Simplify : 1 à la fois, noeud non-move-related de degré $< K$
- Coalesce : approche conservative sur graphe réduit (donc plus de possibilités), un noeud réunion qui devient non-move-related sera mis sur la pile au prochain Simplify.
- Répétition Simplify et Coalesce jusqu'à ce qu'il n'y ait plus de noeuds ou que des noeuds de degré $> K$ ou move-related
- Freeze : recherche d'un noeud move-related de faible degré pour le "geler", pas de coalescing pour lui, devient non-move-related, peut être ses voisins aussi : Simplify et Coalesce reprennent.
- S'il n'y a plus que des noeuds de degré $> K$, choix d'un noeud pour spill potentiel, mis sur la pile, et retour en Simplify
- Select : assigne les couleurs en dépilant les noeuds, si spill réel, reconstruction du graphe et retour au Build.

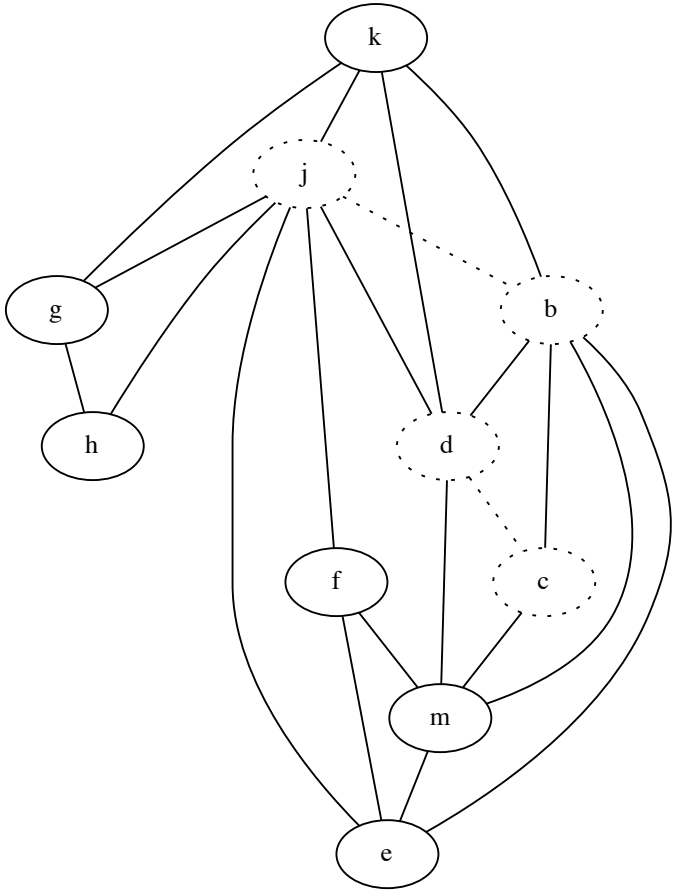
Allocation avec coalescing : exemple

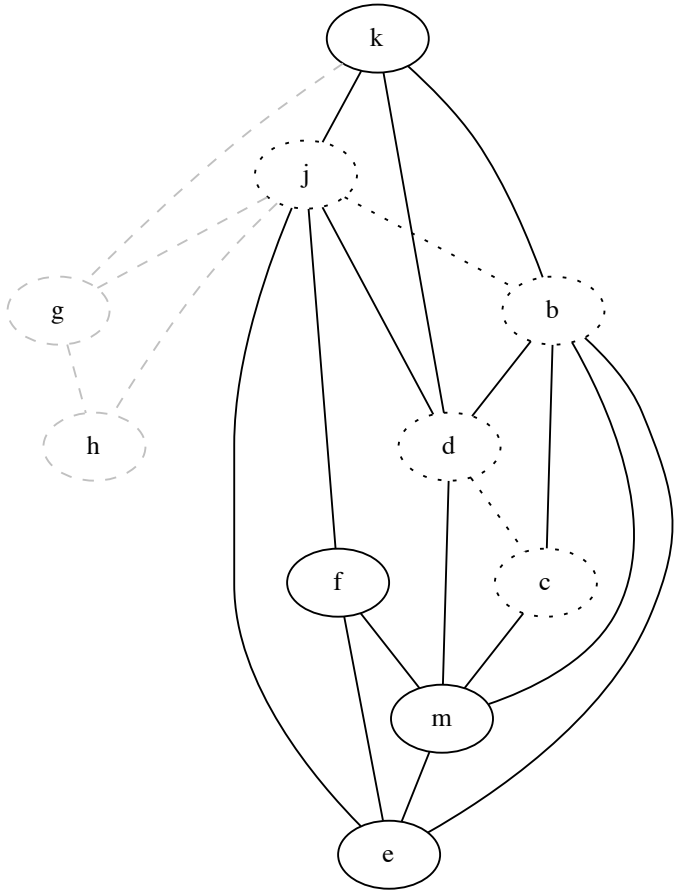
Move-related : j, b, d, c

Moins de 4 voisins : g, h

Choix de g puis h

Pile = g, h



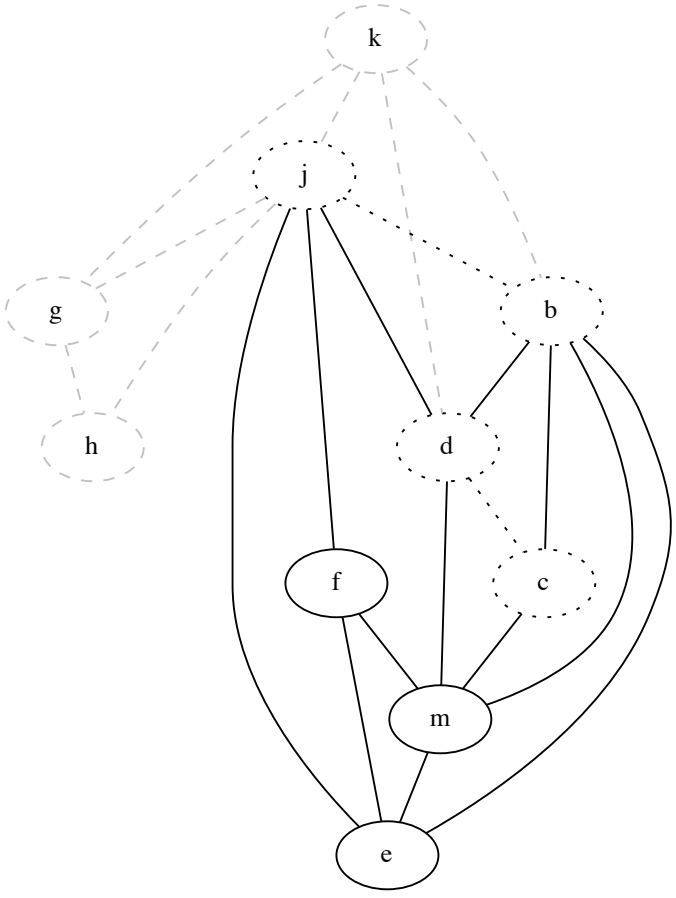


Allocation avec coalescing : exemple

Moins de 4 voisins : k, f

Choix de k

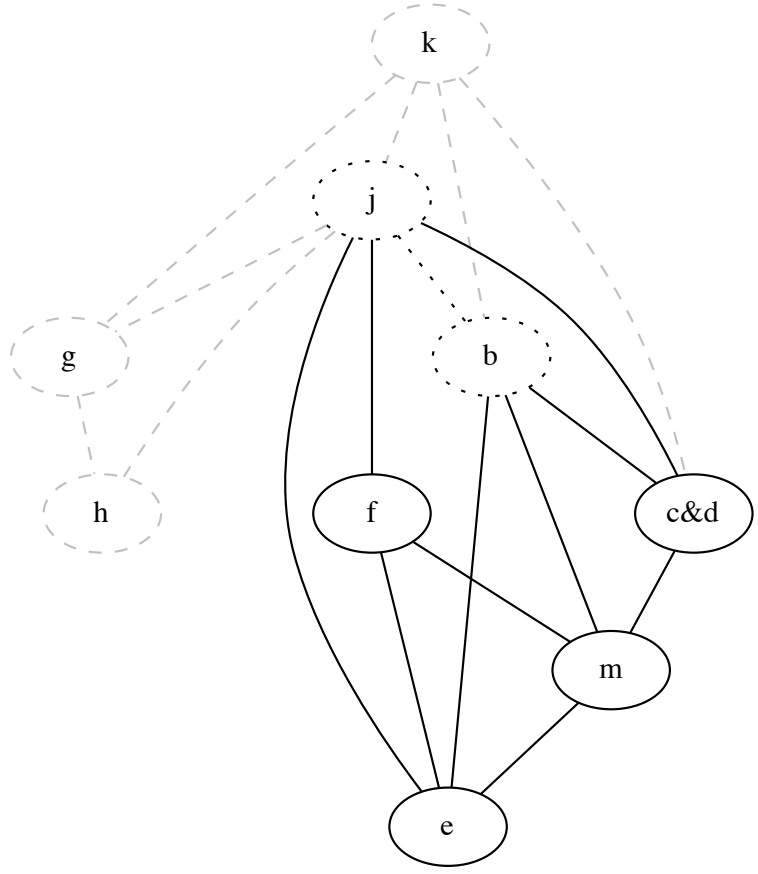
Pile = g, h, k



Coalescing de c et d possible

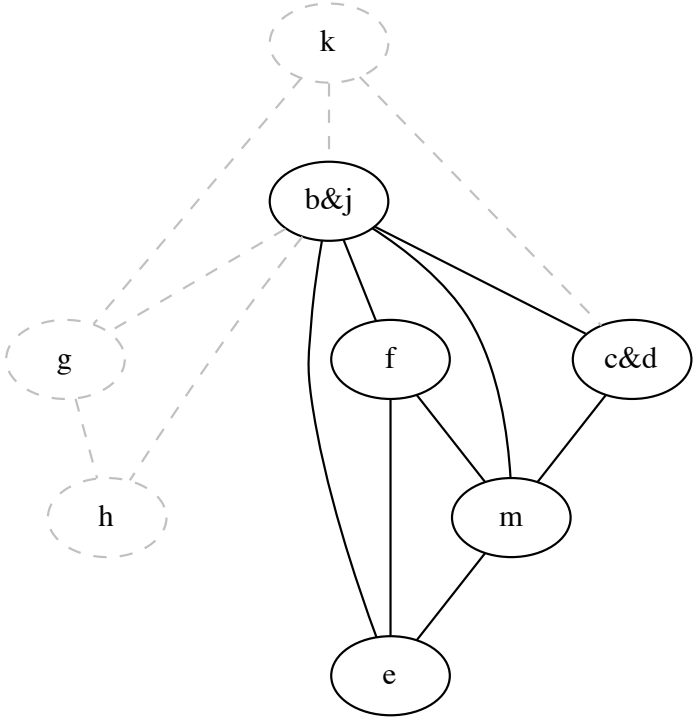
Noeud c&d non-move-related

Pile = g, h, k

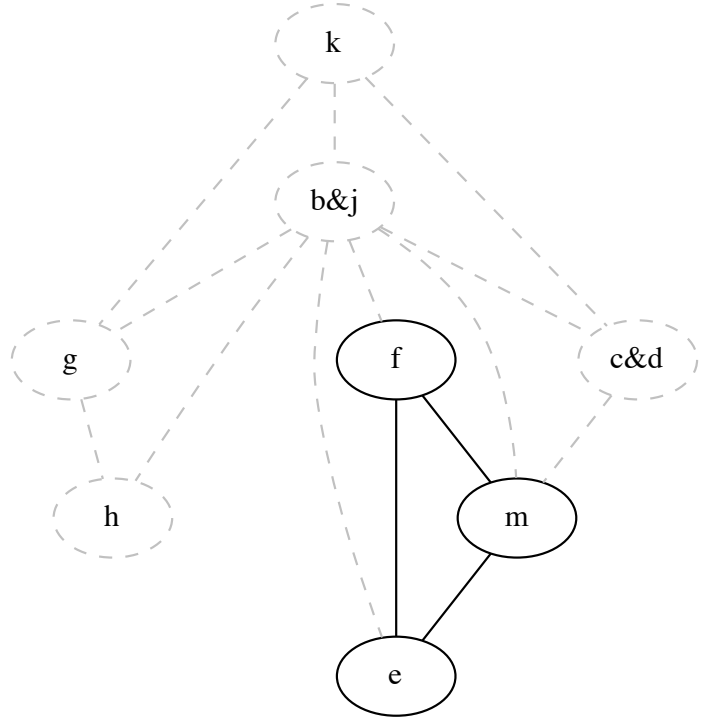


Allocation avec coalescing : exemple

Coalescing de b et j possible
Noeud b&j non-move-related
Pile = g, h, k



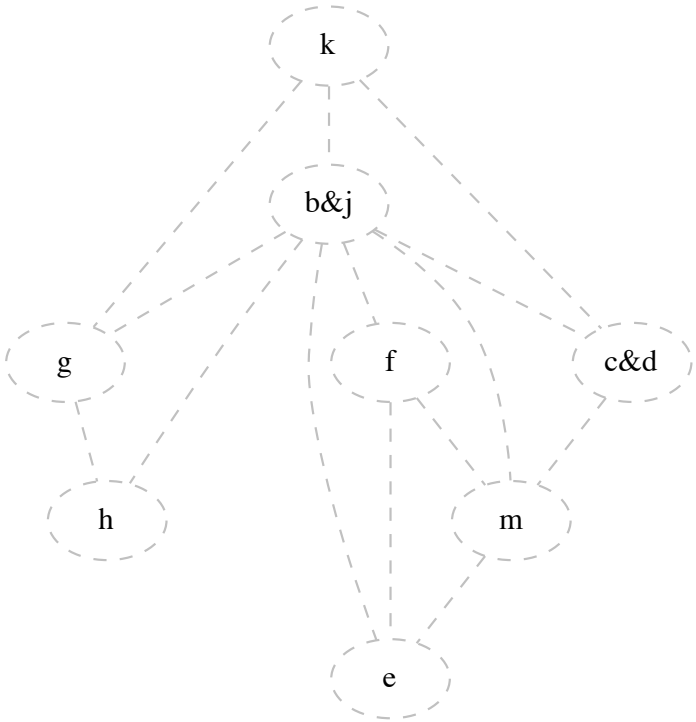
Moins de 4 voisins : c&d, b&j
Pile = g, h, k, c&d, b&j

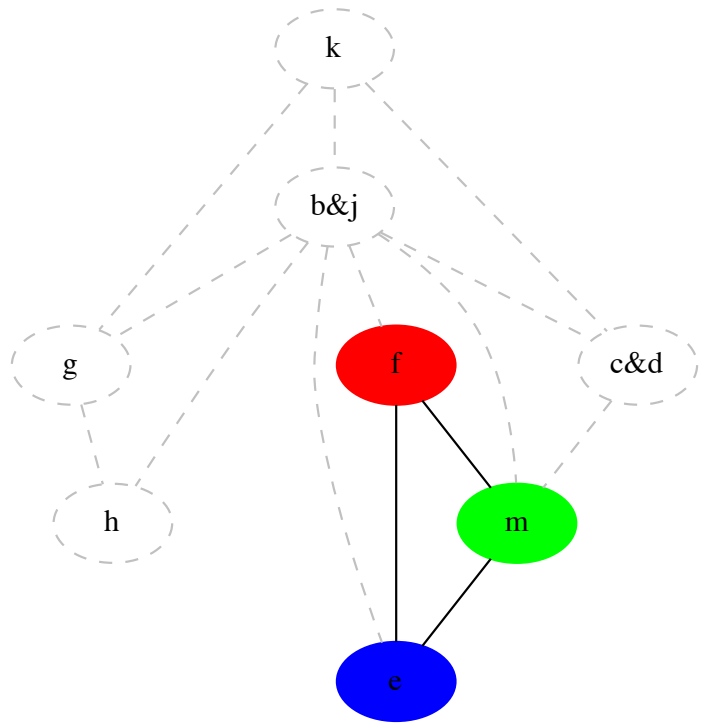


Allocation avec coalescing : exemple

Choix de f, m puis e
Pile = g, h, k, c&d, b&j, f, m, e

Coloriage possible
e ← B, m ← G, f ← R

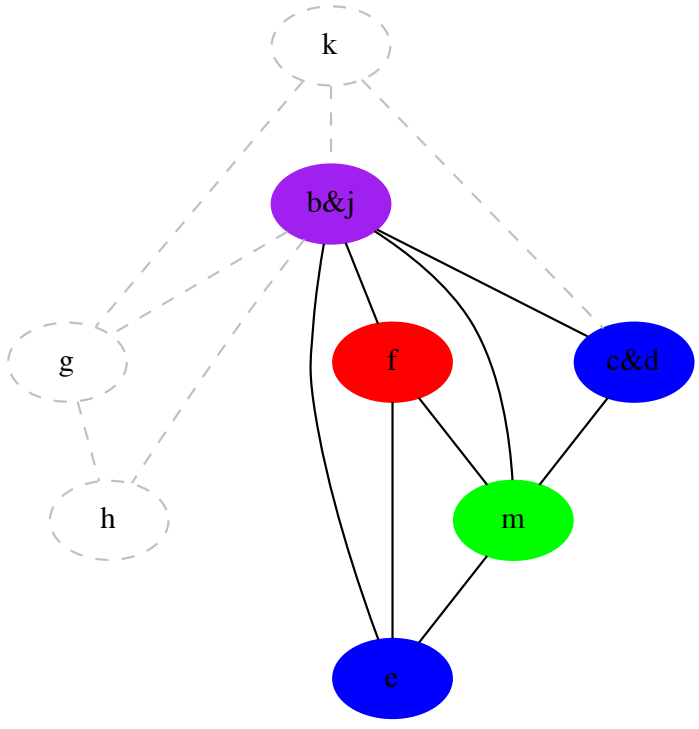


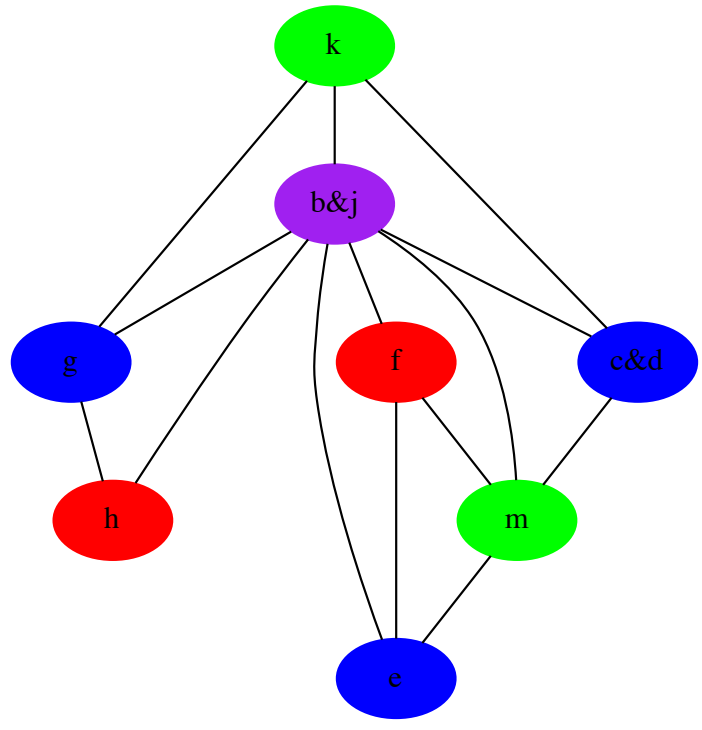


Allocation avec coalescing : exemple

Pile = g, h, k, c&d, b&j
b&j ← P, c&d ← B

Pile = g, h, k
k ← G, h ← R, g ← B

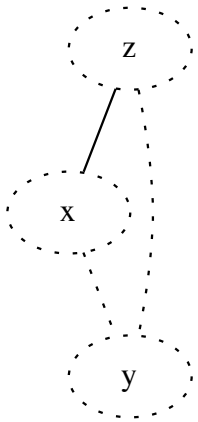


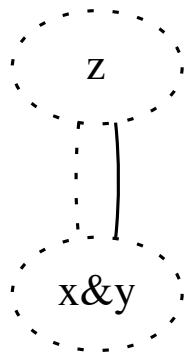


Les 2 instructions de copie ont été éliminées.

Coalescing et contrainte

- Certains move ne sont ni réunis, ni gelés
- Ils sont "contraints"





- Réunion impossible à cause de l'interférence x&y avec z
- Le move est contraint, les noeuds ne sont donc plus move-related

Noeuds précoloriés

- Certains noeuds sont précoloriés : ils représentent des registres machine
- Registres avec utilisation spécifique (stack pointer, frame pointer, arg1, arg2, resultat, ...) doivent être utilisés via des temporaires particuliés liés à ces registres
- Pour chaque couleur (registre) au plus un noeud précolorié
- Select (via Coalesce ou non) peut donner à un temporaire une couleur d'un registre précolorié si pas d'interférence (on peut réutiliser registre utilisé dans convention d'appel au sein d'une procédure)
- Noeuds précoloriés peuvent être réunis avec les noeuds non coloriés avec le coalescing conservatif

Noeuds précoloriés

- Si K registres, alors K noeuds précoloriés
- Les noeuds précoloriés interfèrent entre eux, ceux non utilisés explicitement n'interfèreront pas avec noeuds non précoloriés
- Registre utilisé explicitement interferera avec var vivantes en même temps
- On ne peut pas enlever du graphe un noeud précolorié : signifie le mettre sur la pile pour espérer le colorier, alors que 0 degré de liberté sur la couleur
- On ne peut pas les considérer pour le spill (ce sont des registres !), ils sont vus avec un degré infini

Rappels / définitions

- Registre callee-save est un registre qui doit être sauvegardé par une fonction si elle l'utilise (contenu doit rester cohérent pour l'appelant)
- Registre caller-save est un registre qui doit être sauvegardé par une fonction avant un appel de fonction et restauré après si contenu utilisé après l'appel de la fonction

Noeuds précoloriés et copie temporaire des registres

- Application de Simplify, Coalesce et Spill jusqu'à ce qu'il ne reste que des noeuds précoloriés, ensuite Select qui colorie les noeuds de la pile
- Comme les noeuds précoloriés ne sont pas candidats au Spill, durée de vie courte importante : le compilateur (front-end ou middel-end) génèrer des instructions de move to/from ces noeuds pour réduire leur durée de vie
- Exemple : Si r7 est un registre "callee-save" il est vivant dans le callee, pour cela "défini" à l'entrée de la procedure et "utilisé" à la sortie.
- Au lieu de le conserver dans un noeud précolorié pendant toute la procédure, mis dans un nouveau temporaire au début puis remis dans r7 à la fin.
- En fonction de la pression sur les registres, le temporaire pourra soit être spillé, soit être réuni avec r7 et le move éliminé

```
enter : def(r7)
          ...
          ...
exit :   use(r7)
```

```
enter : def(r7)
        t231 := r7
          ...
          ...
        r7 := t231
exit :  use(r7)
```

Registres callee-save et caller-save

- Registre callee-save : si l'appelé l'utilise, il doit sauvegarder son contenu, utilisation de registre caller-save de préférence dans l'appelé
- Registre caller-save : si l'appelant l'utilise il doit sauvegarder son contenu avant un appel si besoin du contenu après, utilisation de registre callee-save de préférence
- Ainsi dans l'appelant, intérêt à utiliser des registres caller-save pour des variables non vivantes au travers d'appel de procédure
- Dans l'appelant, intérêt à utiliser des registres callee-save pour les variables vivantes avant et après un appel : 1 seul save/restore dans la fonction appelée au début et à la fin au pire
- Allocation de registre doit suivre ces critères, facile de le faire faire à l'algo

Registres callee-save et caller-save

- Instructions d'appel de fonction annotées comme définissant les registres caller-save (interférence donc) : une variable non vivante sera mise dans caller-save
- Si x est vivante au travers de l'appel, elle interfère avec les registres caller-save (précoloriés) et interfèrera avec les temporaires créés pour les callee-save, 1 spill apparaîtra
- L'heuristique de choix du registre à spiller choisit noeud avec degré élevé et peu d'utilisations : les temporaires introduits pour les copies seront choisis, r7 sera alors disponible pour x

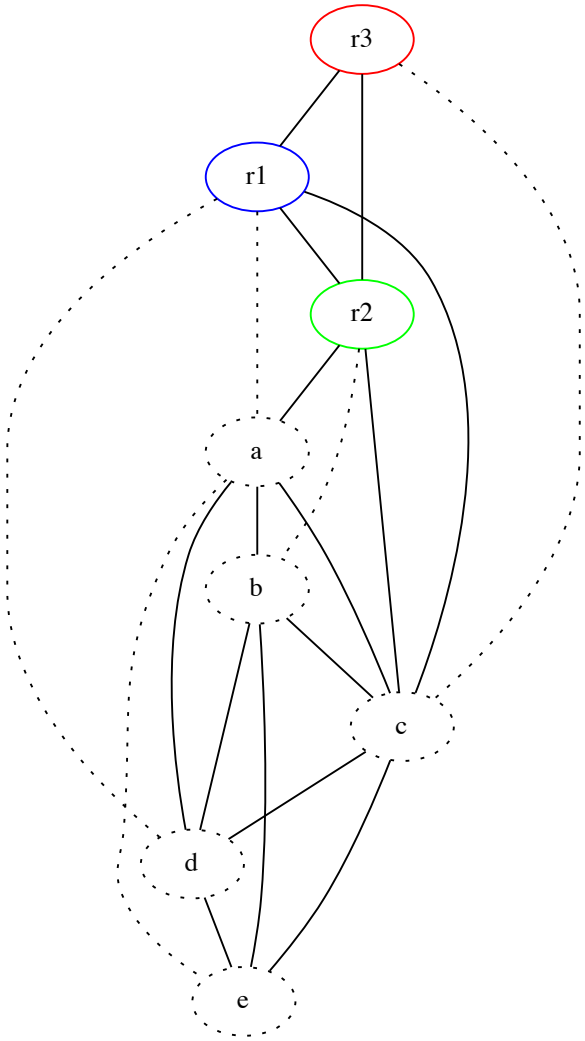
Exemple

```
// r3 callee-save
// r1, r2 caller-save
Enter :
      c := r3
      a := r1 // arg1
      b := r2 // arg2
      d := 0
      e := a

Loop  d := d + b
      e := e - 1
      if (e > 0) goto Loop

      r1 := d // res
      r3 := c
      return
           LIVEout : r1, r3
```

Exemple

Graphe d'interference

K = 3



- Aucune possibilité pour Simplify (move-related) et Freeze (noeud non précolorié degré $>$ K)
- Coalescing impossible (degré trop élevé)
- Spill avec priorité = $((\text{NbUse+NbDef})_{horsloop} + 10 * (\text{NbUse} + \text{NbDef})_{dansloop})$ / degré du noeud
- A calculer pour chaque variable

Exemple

```
// r3 callee-save
// r1, r2 caller-save
Enter :
      c := r3
      a := r1 // arg1
      b := r2 // arg2
      d := 0
      e := a

Loop  d := d + b
      e := e - 1
      if (e > 0) goto Loop

      r1 := d // res
      r3 := c
      return

// LIVEout : r1, r3
```

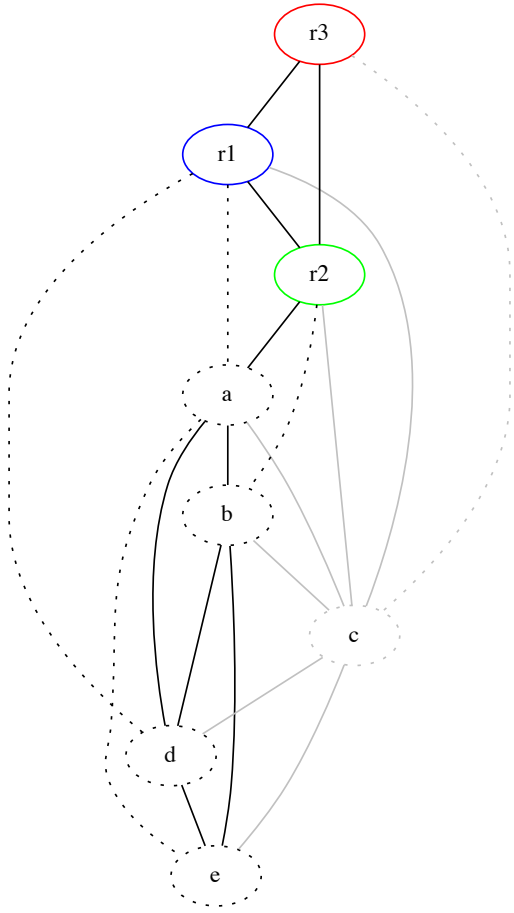
var	use-def	use-def in loop	d	p
a	2	0	4	0.5
b	1	1	4	2.75
c	2	0	6	0.33
d	2	2	4	5.5
e	1	3	3	10.33

Choisi pour le spill : c

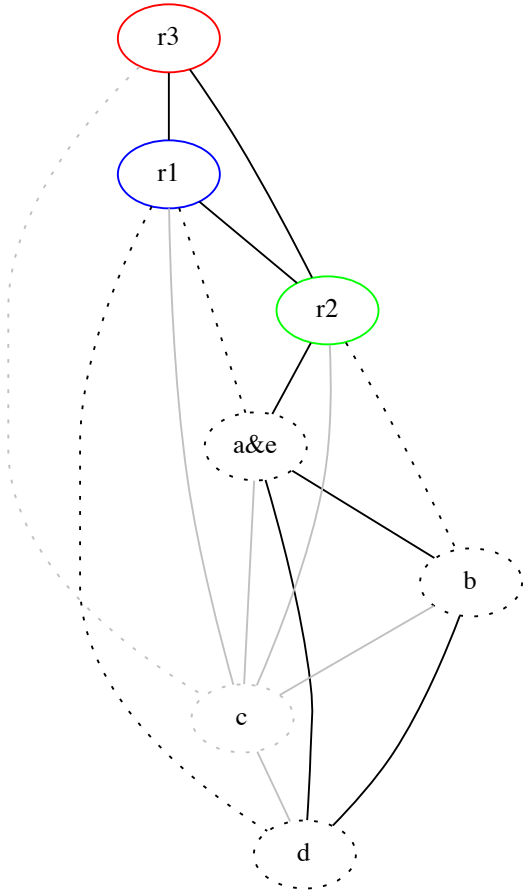
Mis sur la pile

Exemple

Pile = c



Pas de simplify possible
Coalescing de a et e

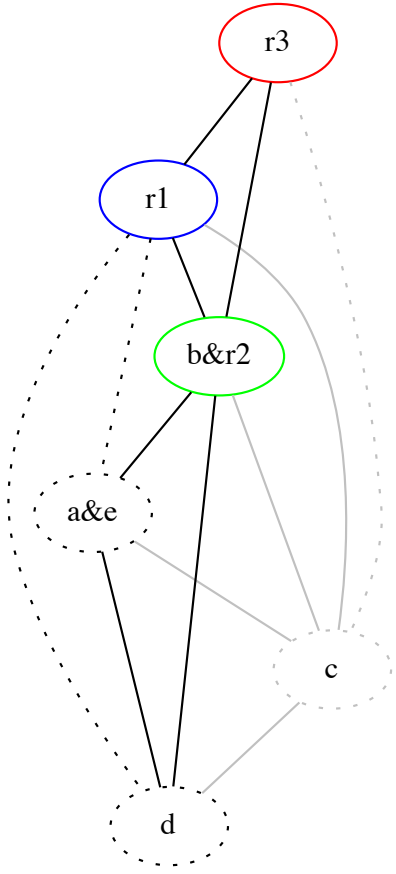


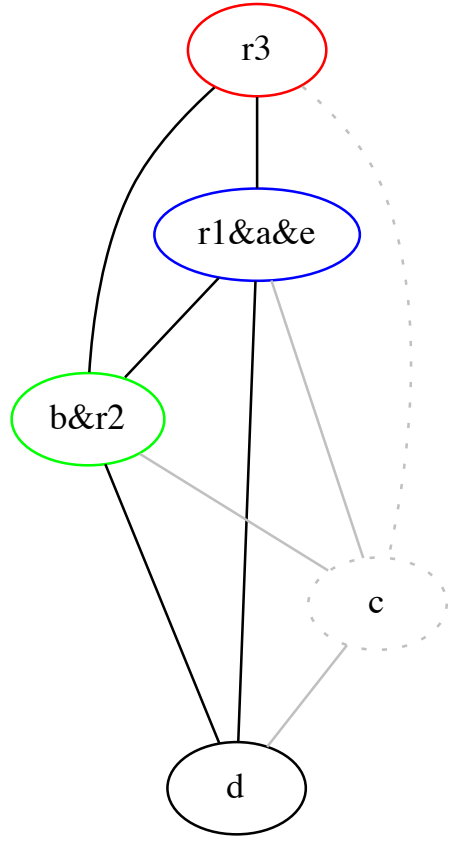
Exemple

Pile = c

Coalescing possible : ae&r1 ou b&r2 Coalescing de ae et r1

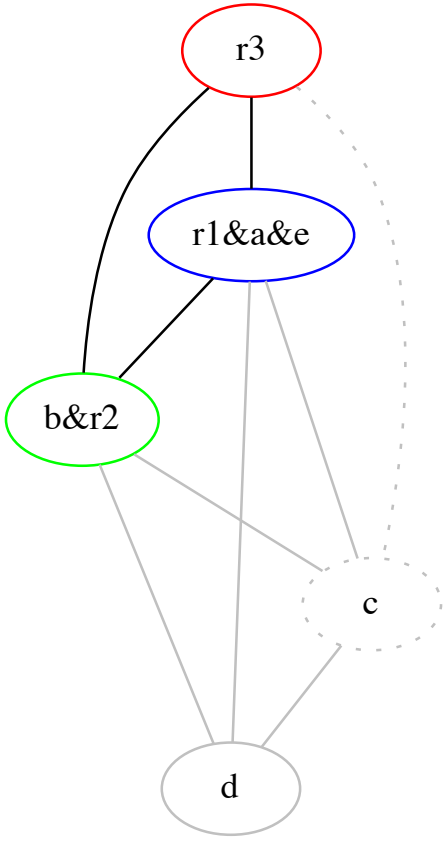
Choix b&r2



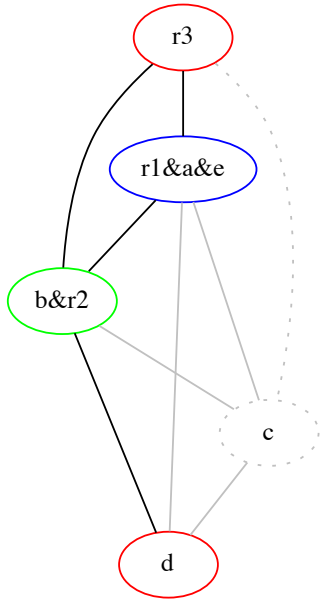


Exemple

Pile = c
Simplify : d



- Que des noeuds précoloriés
- Pill = c, d
- Select : d prend couleur r3
- Aucune couleur pour c
- Spill réel



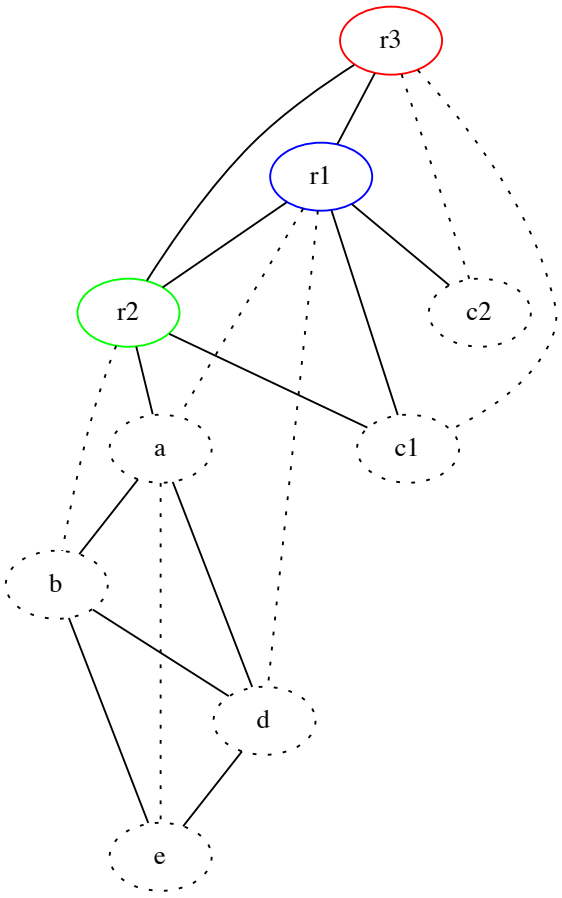
Exemple

```
// r3 callee-save
// r1, r2 caller-save
Enter :
      c1 := r3
      M[c_loc] := c1
      a := r1 // arg1
      b := r2 // arg2
      d := 0
      e := a

Loop  d := d + b
      e := e - 1
      if (e > 0) goto Loop

      r1 := d // res
      c2 := M[c_loc]
      r3 := c2
      return
           LIVEout : r1, r3
```

```
// r3 callee-save
// r1, r2 caller-save
      LIVE = r2, r1, r3
      LIVE = r2, r1, c1
      LIVE = r2, r1
      LIVE = a, r2
      LIVE = a  b
      LIVE = d, b, a

      LIVE = e, d, b
Loop  LIVE = d, e
      LIVE = d, e

      LIVE = d
      LIVE = r1
      LIVE = r1, c2
```

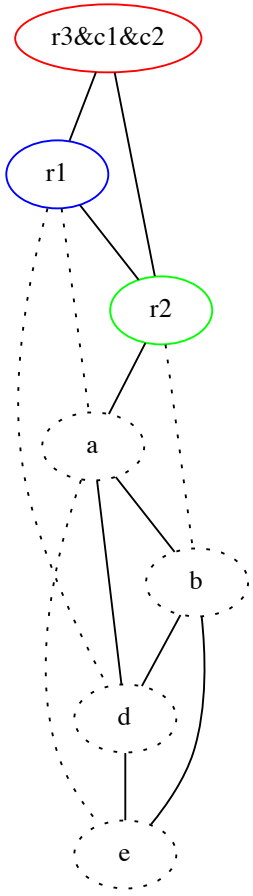
Exemple

Graphe d'interférence



Coalescing possible : c1&r3, c2&r3

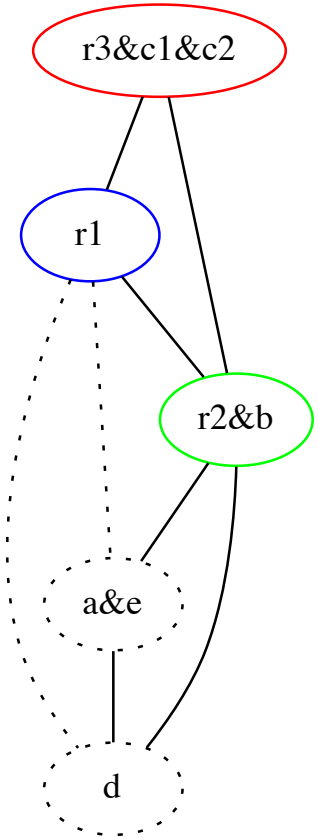
Choix c1&r3, c2&r3



Coalescing possible : a&e, b&r2

Exemple

Choix a&e, b&r2

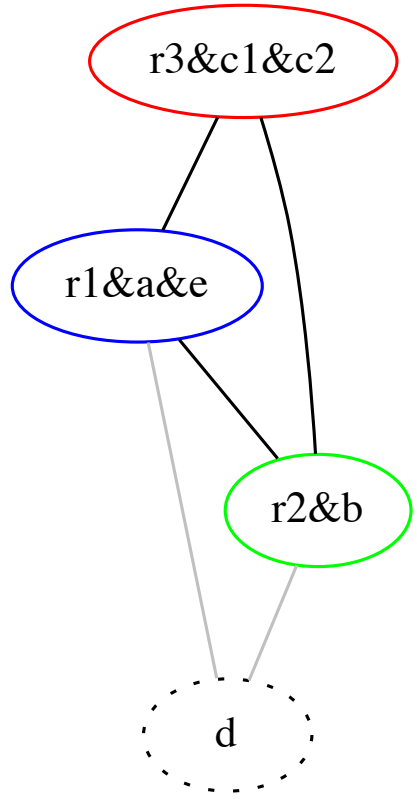


Coalescing possible : a&e et r1

Simplify : d

Coalescing a&e et r1

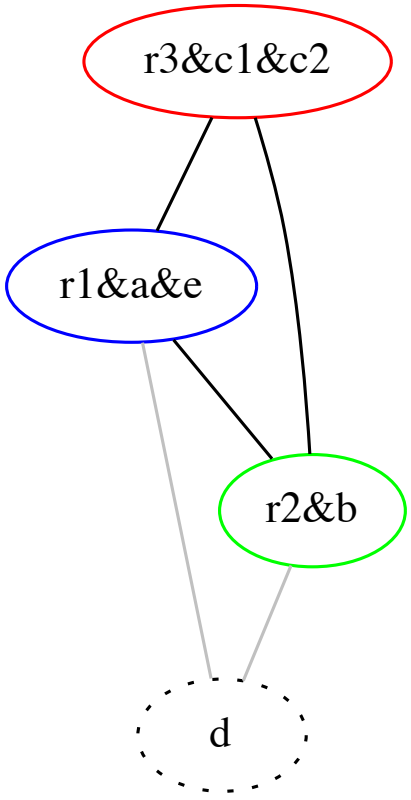
Puis d sur la pile



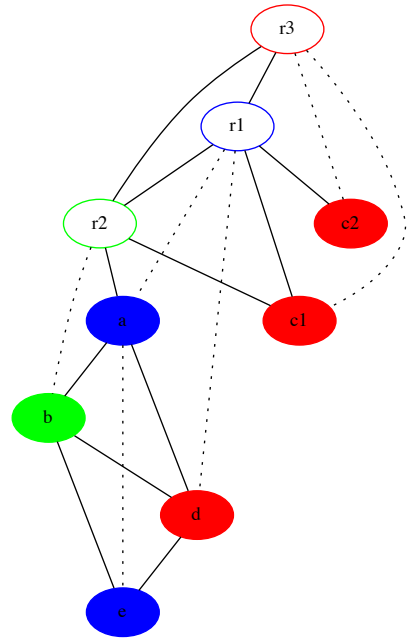
Que des noeuds précoloriés

Exemple

Selection des couleurs
Pile = d



- a couleur de r1
- b couleur de r2
- c1, c2 couleur de r3
- d couleur de r3
- e couleur de r1



Exemple

```
// r3 callee-save
// r1, r2 caller-save
Enter :
      r3 := r3
      M[c_loc] := r3
      r1 := r1 // arg1
      r2 := r2 // arg2
      r3 := 0
      r1 := r1

Loop  r3 := r3 + r2
      r1 := r1 - 1
      if (r1 > 0) goto Loop

      r1 := r3 // res
      r3 := M[c_loc]
      r3 := r3
      return
```

On peut éliminer les copies avec le même registre

```
Enter :
      M[c_loc] := r3
      r3 := 0

Loop  r3 := r3 + r2
      r1 := r1 - 1
      if (r1 > 0) goto Loop

      r1 := r3 // res
      r3 := M[c_loc]
      return
```

Une seule copie non réunie !

Spill et emplacements mémoire

- Une variable choisie pour du spill code nécessite un emplacement mémoire
- Même si quantité infinie, on peut limiter la quantité nécessaire
- On peut utiliser le coloriage de graphe entre les variables choisies pour le spill
- Objectif : limiter le nombre d'emplacements mémoire nécessaires pour le spill code
- Coloriage réalisé avec un nb infini de couleurs