

# Optimisations

## CS410 - Langages et Compilation

Julien Henry  
Catherine Oriat

Grenoble-INP Esisar

2012-2013

# Summary

- 1 Optimisations : introduction
- 2 Graphe de flot de contrôle
- 3 Optimisation des boucles
- 4 Allocation de registres

# Optimisations : challenge

Idéalement, la compilation d'un programme d'un langage vers un autre doit :

- Éliminer le coût des abstraction du langage d'entrée (de plus haut niveau)
- Utiliser les forces du langage cible : en assembleur, instructions spéciales, etc.

# Différentes contraintes

On peut imposer différentes contraintes au code produit :

- Code le plus rapide
- Code le plus compact
- Code le plus proche du source (débugage)
- ...

L'optimisation permet de transformer le code (ou la structure de données représentant le code) pour atteindre l'un de ces objectifs.

# Pourquoi étudier les optimisations ?

C'est important de comprendre ce que le compilateur va faire du code qu'on lui donne en entrée :

- Comprendre l'impact possible de l'optimisation sur sa production de code.
- Comprendre l'impact sur la taille du code (important en embarqué lorsqu'on a très peu de mémoire).
- Comprendre un certain nombre d'optimisations courantes et leurs conséquences.

# Phases d'optimisations

arbre abstrait décoré

Optimisations

arbre abstrait optimisé

Génération de code

→ Erreurs

code cible

Optimisations

(liées au langage cible)

code cible optimisé

Traduction code machine

code machine

# Optimisations 1

Après avoir construit l'arbre abstrait du programme et l'avoir décoré :

- Richesse maximale d'informations
- Identification et fusion de code
- Transformations de la structure du programme
- Déplacement/suppression d'instructions
- etc.

# Optimisations 2

Optimisations liées au langage cible utilisé :

- Sélection d'instructions
- Optimisations locales des séquences d'instructions
- etc.



# Exemples d'optimisations d'un compilateur moderne

## Optimisations 1 :

- Inlining de fonctions
- Elimination de code mort
- Optimisation des boucles imbriquées
- Transformation de boucles (déroutement, fusion, etc)
- propagation de constantes
- ...

## Optimisations 2 :

- Conversion des conditionnelles
- Déplacement de code
- Allocation des registres
- Optimisation par fenêtre

## Exemple d'optimisations 1 : l'inlining

Optimisation très courante :

Remplace un appel de fonction par le code de la fonction :

- Pas d'appel de fonction donc :
  - Pas de gestion de pile
  - Pas de passage de paramètres
  - Meilleure optimisation du code localement
  - Prédiction des processeurs meilleure
- Mais : augmentation de la taille du code

L'inlining

- est sûr : le comportement du code après inlining doit être équivalent au code avant inlining
- doit à priori apporter un gain de vitesse
- est opportun pour les fonctions de petite taille ou appelées à un seul endroit du code.

## Exemple d'optimisations 2 : optimisation par fenêtre

- On génère dans un premier temps de l'assembleur non optimisé
- On fait une passe sur cet assembleur généré en optimisant localement dans une fenêtre

Exemple :

$a = b + c ;$

$d = a + e ;$

MOV b, R0

ADD c, R0

MOV R0, a

MOV a, R0

ADD e, R0

MOV R0, d

MOV b, R0

ADD c, R0

MOV R0, a

ADD e, R0

MOV R0, d

# Summary

- 1 Optimisations : introduction
- 2 Graphe de flot de contrôle**
- 3 Optimisation des boucles
- 4 Allocation de registres

# Analyses nécessaires

De nombreuses optimisations requièrent des analyses **statiques** sur le code :

- Pour supprimer le code mort, il faut faire une analyse statique le détectant.
- Pour allouer les registres de façon optimisée, il faut connaître les variables **vivantes**.
- etc.

On effectue ces analyses sur le **Graphe de flot de contrôle** du programme.

# Graphe de flot de contrôle

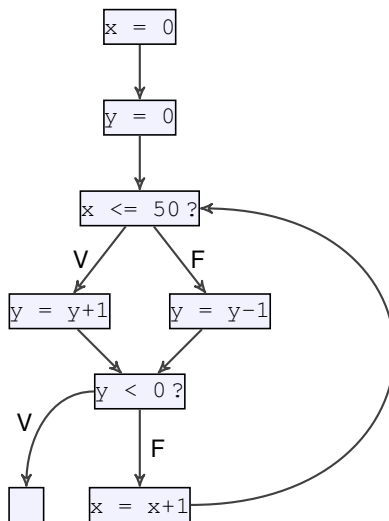
Représentation du programme sous forme de graphe :

- Les états sont les différents points du programme (instructions)
- Les arcs sont les transitions élémentaires entre 2 instructions qui peuvent s'exécuter à la suite.
- Un état a 0, 1 ou 2 arcs sortants (0 si état final, 2 si l'instruction est un branchement conditionnel et 1 sinon)

Ce graphe s'appelle un **graphe de flot de contrôle (CFG)** : il respecte et permet de travailler sur l'ordre d'exécution du programme.

## Exemple

```
x = 0;  
y = 0;  
while (true) {  
    if (x <= 50)  
        y++;  
    else  
        y--;  
  
    if (y < 0) break;  
    x++;  
}
```



# Blocs de base

En pratique, les états du graphe de flot de contrôle ne sont pas de simples instructions, mais une suite d'instructions qui s'exécutent nécessairement à la suite (sans saut).  
Ces états s'appellent des **blocs de base**.

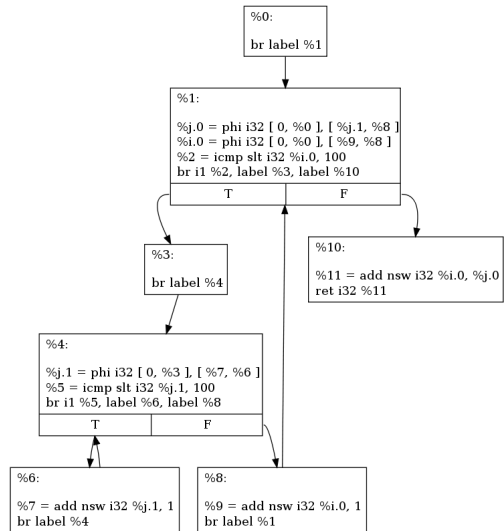


# Exemple de CFG

LLVM est l'infrastructure de compilateur utilisée par Apple.

```
int f() {
    int i = 0;
    int j = 0;

    while (i < 100) {
        j = 0;
        while (j < 100) {
            j++;
        }
        i++;
    }
    return i+j;
}
```



CFG for 'f' function

# Détection de code mort

Trouver les blocs de base inaccessibles :

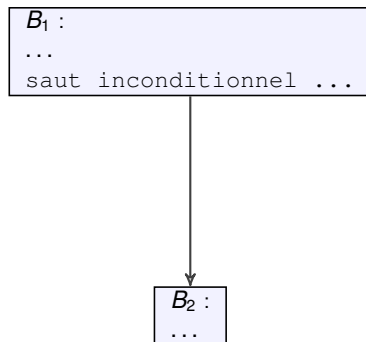
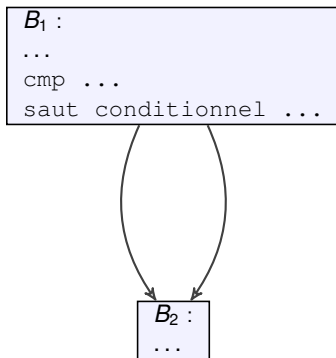
- Soit  $I$  le bloc de base initial du programme.
- Soit  $B$  un bloc de base dans le graphe
- $B$  est un bloc mort si il n'existe aucun chemin valide dans le graphe entre  $I$  et  $B$ .

Exemple :

- Si tous les chemins entre le bloc initial et le bloc  $B$  passent par les branchements  $(i < 100) = T$  et  $(i == 100) = T$ , alors le bloc  $B$  n'est pas atteignable.
- Très souvent, ce n'est pas aussi simple et il faut faire une analyse statique plus avancée pour espérer découvrir du code mort.

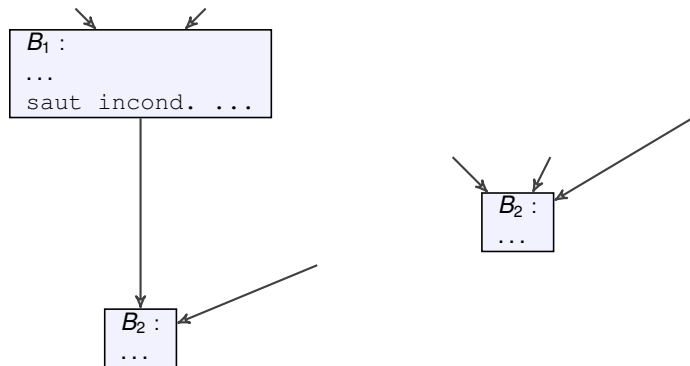
# Simplifications du graphe

Branchement conditionnel vers le même bloc :



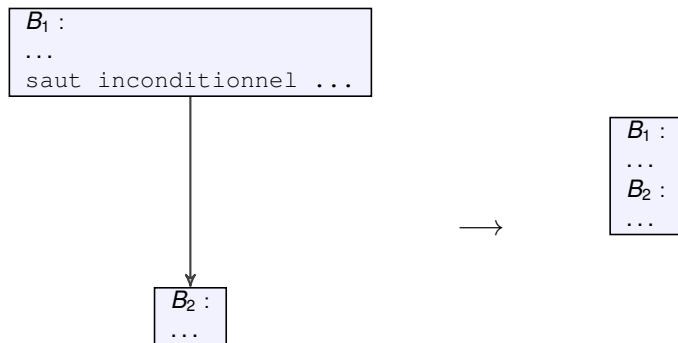
# Simplifications du graphe

Suppression d'un bloc vide :



- $B_1$  est un bloc vide (sans instruction)
- se produit suite à des optimisations dans  $B_1$

# Simplifications du graphe



- $B_1$  et  $B_2$  non vides
- $B_1$  a un saut inconditionnel vers  $B_2$
- $B_2$  a un seul antécédent (qui est  $B_1$ )

# Variables vivantes (Live)

Certaines optimisations s'appuient sur la définition de variable **vivante** :

## Definition

Une variable est vivante en un point du programme (une instruction) si sa valeur peut être utilisée par la suite.

Exemple :

```
int x = 12;  
... (x vivante)  
y = x*2;  
... (x non vivante)
```

```
int x = 12;  
... (x non vivante)  
x = f(5);  
y = x*2;  
... (x non vivante)
```

# Optimisations sur les variables vivantes

Principales optimisations liées à la notion de variables vivantes :

- Pour l'allocation de registres, on s'appuie sur les variables vivantes en un point du programme pour choisir les variables que l'on garde en registre.
- On peut éliminer du code inutile si on se rend compte qu'une variable qui reçoit une affectation est morte.

# Calcul des variables vivantes

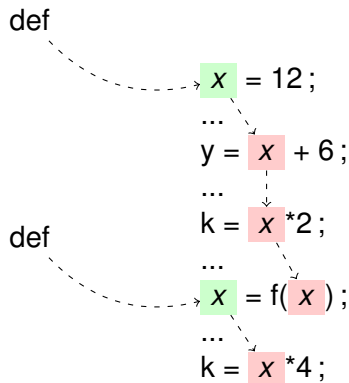
Pour déterminer l'ensemble des variables vivantes à un point du programme, il faut connaître :

- 1 Les points de **définition** des variables : endroits où elles sont affectées.
- 2 Les points d'**utilisation** de ces variables : endroits où on utilise leurs valeurs.

Il est essentiel de pouvoir naviguer facilement entre les utilisations des variables. La représentation intermédiaire du programme inclut donc une chaîne **def-use** : pour une définition de variable, on a la liste de toutes ses utilisations.



## Exemple : def-use chaîne



# Elimination de code inutile

Si, en calculant les **def-use** chaînes, on obtient des chaînes sans utilisations, alors on peut optimiser en supprimant la définition.

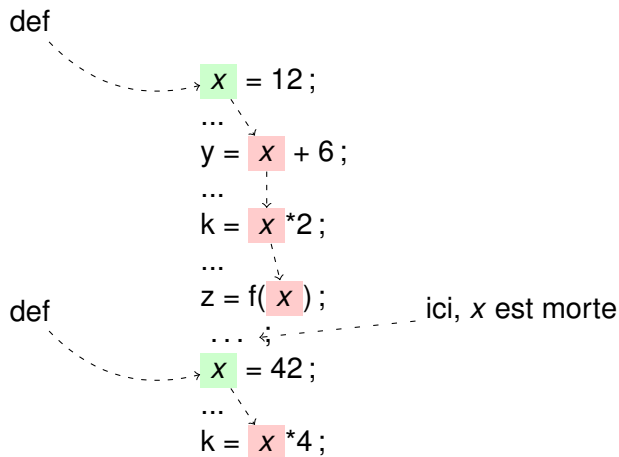
Remarque : ceci n'est vrai que si la définition n'a pas d'effets de bord.

# Variables vivantes

## Definition

Une variable  $x$  est **vivante** à un point  $l$  du programme (instruction) si il existe un chemin dans le CFG entre la définition de  $x$  et une utilisation de  $x$  (dans la même def-use chaîne) qui passe par  $l$ .

# Exemple



# Calcul de l'ensemble des variables vivantes

L'analyse des variables vivantes est en fait **approximative** :

- On vérifie s'il existe dans le CFG un chemin menant vers un site d'utilisation, mais on ne se demande pas à quelle condition ce chemin est faisable en réalité.

En conséquence :

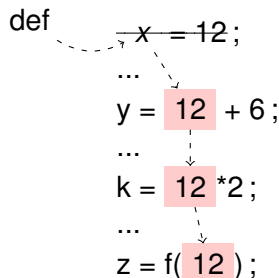
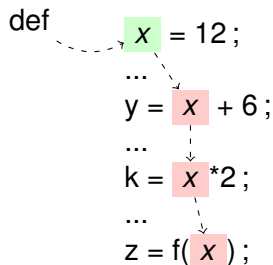
- **vivante** signifie en fait potentiellement vivante
- **morte** signifie morte de façon certaine.

Au pire, on a donc le droit de dire que toutes les variables sont vivantes : les optimisations seront alors très inefficaces mais correctes...

# Propagation de constantes

Une optimisation possible est la **propagation de constantes** :

- On prend une **def-use** chaîne.
- On remplace chaque utilisation par la définition.



# Summary

- 1 Optimisations : introduction
- 2 Graphe de flot de contrôle
- 3 Optimisation des boucles**
- 4 Allocation de registres

# Optimisation des boucles

De très nombreuses optimisations peuvent être faites sur les boucles :

- Pour optimiser l'utilisation du cache
- Pour ne pas répéter inutilement des calculs dans la boucle
- ...

Les quelques slides suivants présentent un certain nombre d'optimisations possibles sur les boucles.



# Fusion de boucles

```
for (int i = 0; i < 50; i++) {  
    t[i] = t[i] + 1;  
}  
...  
for (int i = 0; i < 50; i++) {  
    u[i] = u[i] + 1;  
}
```

Transformé en :

```
for (int i = 0; i < 50; i++) {  
    t[i] = t[i] + 1;  
    u[i] = u[i] + 1;  
}
```

# Changement de l'ordre des boucles imbriquées

Principe de localité : éviter les cache-miss

```
for (int i = 0; i < 50; i++) {  
    for (int j = 0; j < 100; j++) {  
        t[i][j] = t[i][j]+1;  
    }  
}
```

Transformé en :

```
for (int j = 0; j < 100; j++) {  
    for (int i = 0; i < 100; i++) {  
        t[i][j] = t[i][j]+1;  
    }  
}
```

Dépend de l'architecture (technique de cache) et disposition du tableau dans la mémoire.

# Loop unwinding

```
for (int i = 0; i < 50; i++) {  
    f(i);  
}
```

Transformé en :

```
for (int i = 0; i < 50; i+=5) {  
    f(i);  
    f(i+1);  
    f(i+2);  
    f(i+3);  
    f(i+4);  
}
```

- 5 fois moins de tests / branchements
- code 5 fois plus grand (en taille)...

## Loop unswitching

```
for (int i=0; i<1000; i++) {  
    x[i] = x[i] + y[i];  
    if (w) { y[i] = 0 };  
}
```

Transformé en :

```
if (w) {  
    for (int i=0; i<1000; i++) {  
        x[i] = x[i] + y[i];  
        y[i] = 0;  
    }  
} else {  
    for (int i=0; i<1000; i++) {  
        x[i] = x[i] + y[i];  
        y[i] = 0;  
    }  
}
```

# Summary

- 1 Optimisations : introduction
- 2 Graphe de flot de contrôle
- 3 Optimisation des boucles
- 4 Allocation de registres**

# Intro

Dans les compilateurs modernes, on commence par sélectionner les instructions assembleur que l'on va utiliser, en considérant qu'on a un nombre infini de registres.

On obtient alors une représentation intermédiaire appelée ERTL.

# Exemple : factorielle

```

procedure f(1)
var %0,%1,%2,%3,%4,%5,%6
entry f11
f11: newframe           →
f10
f10: move %6, $ra        →f9
f9 : move %5, $s1        →f8
f8 : move %4, $s0        →f7
f7 : move %0, $a0        →f6
f6 : li %1, 0            →f5
f5 : blez %0             →
f4,f3
f3 : addiu %3, %0, -1    →f2
f2 : j → f20

f20: move $a0, %3        →f19
f19: call f(1)           →f18
f18: move %2, $v0        →f1
f1 : mul %1, %0, %2      →f0
f0 : j                  →f17
f17: move $v0, %1        →f16
f16: move $ra, %6        →f15
f15: move $s1, %5        →f14
f14: move $s0, %4        →f13
f13: delframe          →f12
f12: jr $ra
f4 : li %1, 1           →f0

```

## Exemple : factorielle, après analyse des variables vivantes

```
procedure f(1)
```

```
var %0,%1,%2,%3,%4,%5,%6
```

```
entry f11
```

f11: <b>newframe</b>	→f10	\$a0, \$s0, \$s1, \$ra
f10: <b>move</b> %6, \$ra	→f9	%6, \$a0, \$s0, \$s1
f9 : <b>move</b> %5, \$s1	→f8	%5, %6, \$a0, \$s0
f8 : <b>move</b> %4, \$s0	→f7	%4, %5, %6, \$a0
f7 : <b>move</b> %0, \$a0	→f6	%0, %4, %5, %6
f6 : <b>li</b> %1, 0	→f5	%0, %4, %5, %6
f5 : <b>blez</b> %0	→f4, f3	%0, %4, %5, %6
f3 : <b>addiu</b> %3, %0, -1	→f2	%0, %3, %4, %5, %6
f2 : <b>j</b> → f20		%0, %3, %4, %5, %6
f20: <b>move</b> \$a0, %3	→f19	%0, %4, %5, %6, \$a0
f19: <b>call</b> f(1)	→f18	%0, %4, %5, %6, \$v0
f18: <b>move</b> %2, \$v0	→f1	%0, %2, %4, %5, %6
f1: <b>mul</b> %1, %0, %2	→f0	%1, %4, %5, %6
f0: <b>j</b>	→f17	%1, %4, %5, %6
f17: <b>move</b> \$v0, %1	→f16	%4, %5, %6, \$v0
f16: <b>move</b> \$ra, %6	→f15	%4, %5, \$v0, \$ra
f15: <b>move</b> \$s1, %5	→f14	%4, \$v0, \$s1, \$ra
f14: <b>move</b> \$s0, %4	→f13	\$v0, \$s0, \$s1, \$ra
f13: <b>delframe</b>	→f12	\$v0, \$s0, \$s1, \$ra
f12: <b>jr</b> \$ra		
f4: <b>li</b> %1, 1	→f0	%1, %4, %5, %6



# Interférence

- On a maintenant un ensemble de pseudo-registres %0, %1, ... auxquels il faut associer un vrai registre.
- On connaît les propriétés de vivacité de chaque pseudo-registre.

**Question :** Quels sont les pseudo-registres qui peuvent être associés au même registre ?

## Definition

Deux pseudo-registres peuvent être associés au même registre si ils n'**interfèrent** pas : on n'écrit jamais dans l'un si l'autre est vivant.

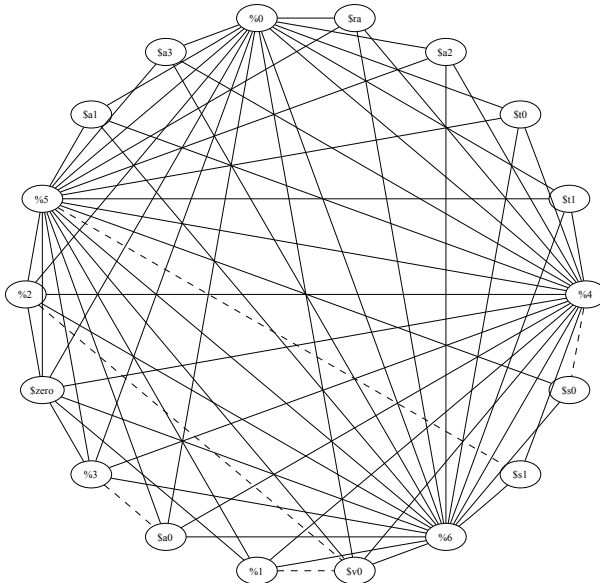
Les instructions **move** fournissent des emplacements préférentiels (pour essayer de les supprimer).

# Graphe d'interférence

On construit pour l'allocation de registres un graphe d'interférences :

- Les sommets sont les pseudo-registres
- Les arêtes représentent les relations d'interférence
- Les arêtes en pointillé représentent les arêtes de préférence (lorsqu'on a une instruction **move %0, %1**, on choisit de préférence d'associer le même registre à %0 et %1)

# Exemple : graphe d'interférence pour factorielle



# Coloriage du graphe d'interférence

Supposons que l'on dispose de  $k$  registres dans notre architecture cible. Le problème de l'allocation de registre se réduit à :

- Soit  $k$  couleurs différentes.
- Attribuer une couleur à chaque sommet du graphe d'interférence (pseudo-registre), tel que
  - 1 deux sommets reliés par une arête d'interférence n'ont jamais la même couleur.
  - 2 si possible, deux sommets reliés par une arête de préférence sont de même couleur.

Si une solution à ce problème existe, on dit que le graphe est  $k$ -colorable. Sinon, notre programme devra utiliser des variables temporaires sur la pile (**spill**)...

# Difficultés et limitations

Premier problème :

- Le problème de coloriage de graphe est **NP-complet**, donc impossible en pratique de colorier le graphe de façon optimale.

On s'appuie donc sur des heuristiques pour colorier le graphe de façon linéaire ou quasi-linéaire.

# Difficultés et limitations

Deuxième problème :

- Si le graphe n'est pas  $k$ -colorable, ou si on ne trouve pas de  $k$ -coloriage, que faire ?

On laisse certains sommets non coloriés. Les sommets non-coloriés seront des pseudo-registres qui seront en fait placés dans la pile et non pas de façon permanente dans un registre.

# Difficultés et limitations

Troisième problème :

- Sur certaines architectures, les registres ne sont pas tous interchangeables.

Exemple : sur les Motorola 68k, l'instruction d'addition accepte un registre d'**adresse** ou de **données** en tant qu'opérande, mais l'instruction de multiplication n'accepte qu'un registre de **données**.

Dans ce cas, il faut adapter l'algorithme de coloration...

# Un algorithme de coloriage

Algorithme de Chaitin :

- Un sommet  $s$  de degré strictement inférieur à  $k$  est **trivialement colorable** : le graphe  $G$  est  $k$ -colorable si et seulement si  $G \setminus \{s\}$  est  $k$ -colorable.
- On peut répéter cette simplification autant de fois que possible.
- Après simplification, de nouveaux sommets peuvent devenir **trivialement colorables**.



# Un algorithme de coloriage

Algorithme de Chaitin :

```
procedure COLORIER (G)
  Si le graphe  $G$  n'a aucun sommet alors terminer
  Si il existe un sommet  $s$  trivialement colorable :
    COLORIER ( $G \setminus \{s\}$ )
    Attribuer une couleur disponible à  $s$ 
  Sinon
    Choisir un sommet  $s$ 
    COLORIER ( $G \setminus \{s\}$ )
    Spiller  $s$ 
```

# Un algorithme de coloriage

Problème **NP-complet**  $\Rightarrow$  on utilise des **heuristiques** !

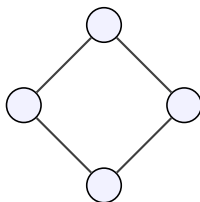
Le choix des sommets à **spiller** est très important :

- Il faut spiller les registres **peu utilisés** ou utilisé en des points non critiques du code (hors boucle, etc).
- Pour faciliter la suite du coloriage, il vaut mieux choisir un sommet de fort degré pour faire baisser le degré des autres. . .

On crée donc une fonction de coût qui utilise ces critères.

## Exemple

Pour  $k = 2$ , que donne l'algorithme de coloriage ?



Comment améliorer l'algorithme ?

# Un algorithme de coloriage

Algorithme de Chaitin :

```
procedure COLORIER (G)
  Si le graphe  $G$  n'a aucun sommet alors terminer
  Si il existe un sommet  $s$  trivialement colorable :
    COLORIER ( $G \setminus \{s\}$ )
    Attribuer une couleur disponible à  $s$ 
  Sinon
    Choisir un sommet  $s$ 
    COLORIER ( $G \setminus \{s\}$ )
    Si une couleur est disponible pour  $s$ 
      la lui attribuer
    Sinon
      Spiller  $s$ 
```