SSA-based Compiler Design

EJCP 2014 — Rennes

F. Rastello*
F. Bouchez Tichadou†

*Inria
†Université Joseph Fourier

What's in a name?

Cytron & Ferrante, 1987

What's in a name?

Or,
the value of renaming for parallelism detection and storage allocation

Cytron & Ferrante, 1987

What's in a compiler?

Goals for this lecture

- Understand the importance of "names" ☞ SSA Form
- Introduce an interesting optimization problem ☞ register allocation

Outline

Outline

Static Single Assignment (SSA)

¿SSA?

- Assignment: variable's definition (e.g., `x` in ``x=y+1'')
- Single: only one definition per variable
- Static: in the program text

Referential transparency

Example (y and z are not equal)

opaque (context dependent)	referentially transparent SSA form
$x = 1;$	$x_1 = 1;$
$y = x + 1;$	$y = x_1 + 1;$
$x = 2;$	$x_2 = 2;$
$z = x + 1;$	$z = x_2 + 1;$

Referential transparency

Example (y and z are not equal)

opaque (context dependent)	referentially transparent SSA form
$x = 1;$	$x_1 = 1;$
$y = x + 1;$	$y = x_1 + 1;$
$x = 2;$	$x_2 = 2;$
$z = x + 1;$	$z = x_2 + 1;$

Referential transparency

- value of variable independent of its position
- may refine our knowledge (e.g., ``if (x==0)'') but underlying value of x does not change

Informal Semantics

Each variable `v` is:

- used only once as `v = ...` (target/definition/left-hand-side)
- can be many times as `... = v` (source/use/right-hand side)

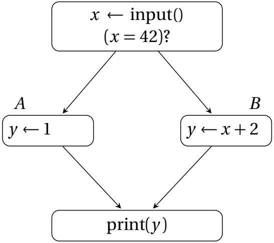
Informal Semantics

Each variable v is:

- used only once as `v = ...` (target/definition/left-hand-side)
- can be many times as `... = v` (source/use/right-hand side)

x = input();
if (x == 42) {
  y = 1;
} **else** {
  y = x + 2;
}

print(y);



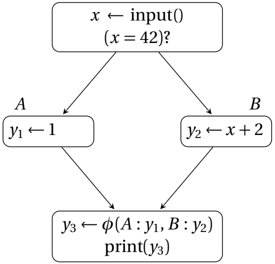
▸ CFG

Informal Semantics

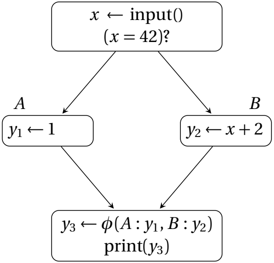
Each variable `v` is:

- used only once as `v = ...` (target/definition/left-hand-side)
- can be many times as `... = v` (source/use/right-hand side)

$$
\begin{array}{l}
x = \text{input}();\\
\textbf{if } (x == 42)\ \{\\
\quad y_1 = 1;\\
\}\ \textbf{else}\ \{\\
\quad y_2 = x + 2;\\
\}\\
y_3 = \phi(y_1, y_2);\\
\text{print}(y_3);
\end{array}
$$



Informal Semantics



Introduction of ϕ-functions:

- to fix the ambiguity; introduces y_3 which takes either y_1 or y_2
- placed at control-flow merge points i.e., head of basic-blocks that have multiple predecessors

- n parameters if it has n incoming CFG paths
- represented as $a_0 = \phi(a_1, \ldots, a_n)$

Questions on SSA

$\approx$ 5 min to answer the following questions:

Is it possible to have more than one ϕ-function in a basic block?

How can you execute code containing ϕ-functions on a machine?

Informal Semantics

- multiple ϕ-functions executed simultaneously:
  $a = \phi(a, b)$
  $b = \phi(b, a)$
- ϕ-functions not directly executable (IR only: for static analysis)
- ϕ-functions removed before assembly code generation
  ☞ copy instructions insertion

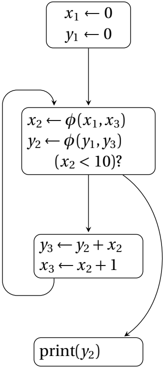
Informal Semantics

- multiple ϕ-functions executed simultaneously:
  $a = \phi(a, b)$
  $b = \phi(b, a)$
- ϕ-functions not directly executable (IR only: for static analysis)
- ϕ-functions removed before assembly code generation
  ☞ copy instructions insertion
- exists extensions of ϕ-functions (e.g., ϕ_{if}, γ, etc.) that take an additional predicate parameter

Informal Semantic

- SSA is not Dynamic Single Assignment (DSA or SA)
- Construction: insert ϕ-function where multiple reaching defs converge; version variables x and y (integer subscripts);

$x = 0$;
$y = 0$;
while $(x<10)$ {
  $y = y + x$;
  $x = x + 1$;
}
print (y);



Comparison with Classical Data Flow Analysis

- During actual program execution, information flows between variables
- Static analysis captures this behavior by propagating abstract information along CFG
- Can be propagated more efficiently using a functional or sparse representation such as SSA
- Constant propagation: definitions $\equiv$ set of points where information may change; associate information with variable names rather than variables $\times$ program points

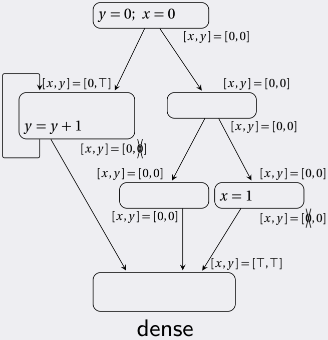
Comparison with Classical Data Flow Analysis

Null pointer analysis

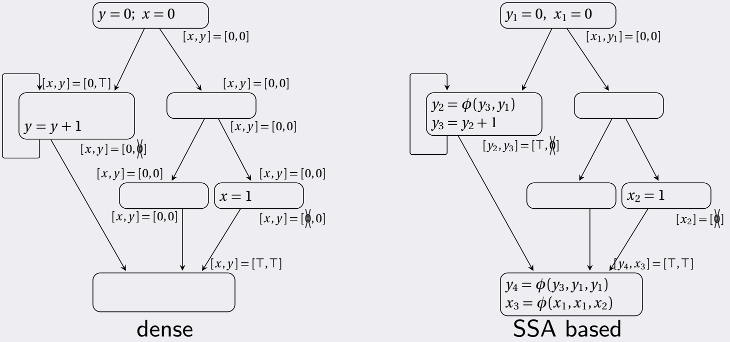
Determine statically if variable can contain null value at run-time.

Comparison with Classical Data Flow Analysis

Null pointer analysis



Null pointer analysis



- Propagates from defs to uses (via def-use links); avoid program points where information does not change or not relevant
- Results are more compact

SSA in Context

- 1980s developments of IRs to encapsulate data dependences to expose direct link between definitions and uses (def-use chains). eg PDG, PD-web... SSA developed at IBM and published late 80s
- GCC, Open64, HotSpot, Jikes, V8, Mono, LLVM... use SSA
- more and more popular for JIT compilation on Java byte-code, CLI byte-code, LLVM bitcode...
- because of favorable properties (simplification and reduced complexity) recently adopted back-end level even register allocation phase
- also for high-level language impose referential transparency e.g. SISAL; on a per-variable basis final in Java, const or readonly in C#. Immutability simplifies concurrent programming

Benefits of SSA

- Compile time benefit (e.g., sparse data flow analysis)
- Compiler development benefit (e.g., dead-code in GCC 4.x 40% of GCC 3.x)
- Program run-time benefit (simpler to develop more efficient analysis)

Myth	Reality
Greatly increases number of vars	$\approx$10% expansion
Destruction generates many copy ops	not more than original prog.
SSA property difficult to maintain	$\equiv$ SSA construction restricted to some variables / code region

Outline

1 Vanilla SSA (J. Singer)

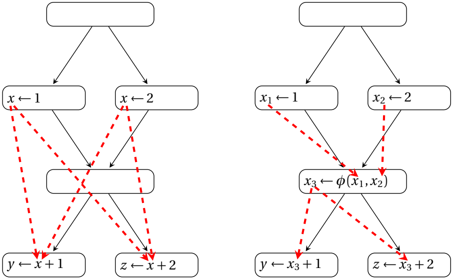
2 Properties and Flavors (P. Brisk, F. Rastello)

3 Register Allocation (F. Bouchez Tichadou)

4 Static Single Information Form (F. Pereira, F. Rastello)

Def-Use and Use-Def Chains

- Def-Use chains: for a definition the set of all its uses
- Use-Def chain: for a use the (unique under SSA) definition that reaches the use
- direct connections to propagate data-flow information



- Information is combined as early as possible
- Use-Def Chains for free; Def-Use Chains almost for free.

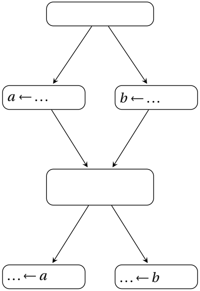
Minimality

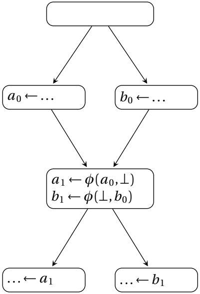
- Construction: place ϕ-functions (e.g., $a = \phi(a, a)$); rename variables. Minimality is a property of the code before renaming.
- Single reaching-definition property: no program point can be reached by two definitions of the same variable
- Minimality property: minimality of the number of inserted ϕ-functions
- n_3 is a join node of n_1 and n_2 ($n_3 \in \mathcal{J}(n_1, n_2)$) if $\exists$ two non-empty path (at least one edge), from n_1 to n_3 and from n_2 to n_3.
- necessary: place ϕ-functions of var v at $\mathcal{J}(\mathrm{Defs}_v)$
- sufficient: $\mathcal{J}(S \cup \mathcal{J}(S)) = \mathcal{J}(S)$
- strictness: in practice place at $\mathcal{J}(\mathrm{Defs}_v \cup r)$

Minimality not a requirement. Copy-propagation is enough.

Strict SSA Form and Dominance Property

- strict: if every variable is defined before it is used. Java imposes strictness, C++ does not.
- n_1 dominates basic block n_2 if every path from r to n_2 includes n_1





- dominance property: each use of a variable is dominated by its definition
- Add (undefined) pseudo-definition of each variable at r

Dominator Tree 2.5cm

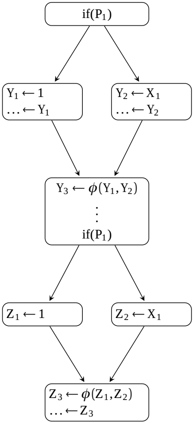
Strict SSA Form and Dominance Property

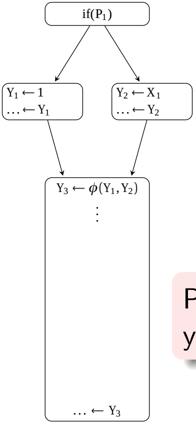
- So called "Minimal SSA" (minimality and dominance property) can be efficiently built using formalism of dominance frontier ($\mathcal{J}(\mathrm{Def}_v, r) = \mathrm{DF}^+(v)$)
- Structural properties of variables' live-range: sub-tree of the dominator tree ▸ Chordal 1.4cm
  1. Fast liveness-check and iteration free liveness-set
  2. Graph Coloring in linear time using tree scan (register allocation) ▸ Tree scan 1.6cm
- break strictness: e.g. copy-propagation
- make it strict: use standard incremental update

Strictness is good especially for JIT

Pruned SSA Form

- Minimal SSA: ϕ-functions where var not live prior to SSA construction



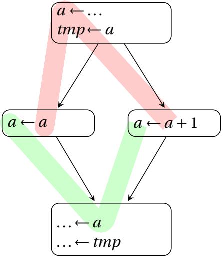


- bad for register allocation
  good for value numbering
- pruned SSA: without non-live ϕ-functions
- make it pruned: dead-code elimination

> Prune it (semi-pruned is ok) unless you really need it

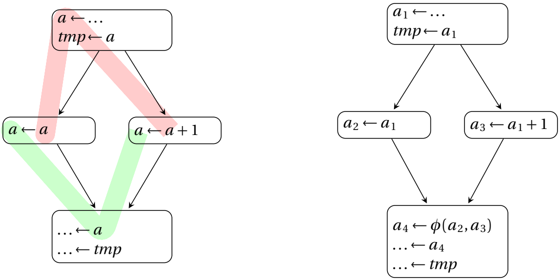
Conventional and Transformed SSA Form

- register web: maximum unions of def-use chains
- ϕ-webs: transitive closure of ϕ-related variables



Conventional and Transformed SSA Form

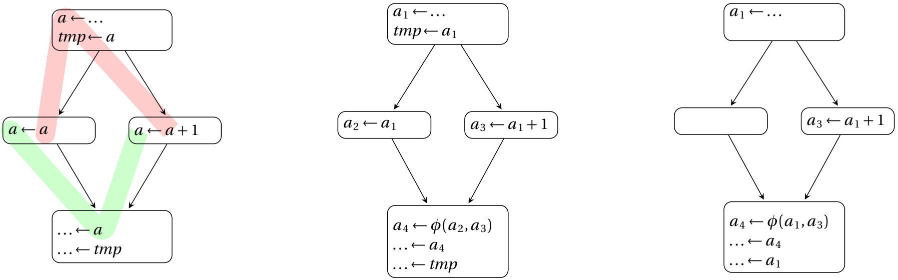
- register web: maximum unions of def-use chains
- ϕ-webs: transitive closure of ϕ-related variables



- Conventional SSA (C-SSA): each ϕ-web is interference free.

Conventional and Transformed SSA Form

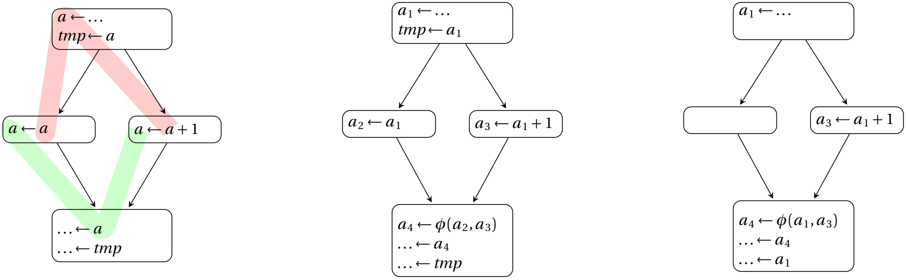
- register web: maximum unions of def-use chains
- ϕ-webs: transitive closure of ϕ-related variables



- Conventional SSA (C-SSA): each ϕ-web is interference free.
- Transformed SSA (T-SSA): non-conventional SSA

Conventional and Transformed SSA Form

- freshly constructed SSA: ϕ-web $\equiv$ register web of original non-SSA
- destructing C-SSA: replace each ϕ-web by a single variable
- make it conventional: as “difficult” as destructing SSA: insert copies to dissociate interfering vars from the connecting ϕ-functions.



Don’t try to enforce conventional property but for the really last phases of code generation

Outline

What is register allocation?

Assign variables (unbounded) to:

Registers (■ ■ ■) or memory (infinite)

Architectural subtleties

Specific registers (sp, fp, r0), variable affinities (auto-inc), register pairing (64 bits ops), distributed register banks, etc.

Rules of the game

- Fixed instruction schedule
- Spill: insert LOADS and STORES
- Coalesce: delete register-to-register MOVES
- Split: add register-to-register MOVES

What is register allocation (cont'd)?

Allocation versus assignment

- Allocation: which variables are allocated to memory (spilled), which ones are allocated to registers?
- Assignment: assign non spilled variables to registers

Allocation problem

- The spill test: is spilling necessary (assignment feasible)?
- What to spill and where to insert loads & stores?

What is register allocation (cont'd)?

Allocation versus assignment

- Allocation: which variables are allocated to memory (spilled), which ones are allocated to registers?
- Assignment: assign non spilled variables to registers

Allocation problem

- The spill test: is spilling necessary (assignment feasible)?
- What to spill and where to insert loads & stores?

What is register allocation (cont'd)?

Allocation versus assignment

- Allocation: which variables are allocated to memory (spilled), which ones are allocated to registers?
- Assignment: assign non spilled variables to registers

Allocation problem

- The spill test: is spilling necessary (assignment feasible)?
- What to spill and where to insert loads & stores?

In-Two-Phases Register Allocation and Tree-Scan

Spill test:
(NP-complete?

Chaitin's reduction proves the NP-completeness of the spill test...

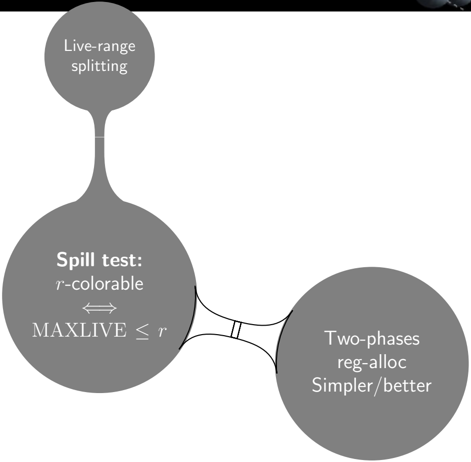
In-Two-Phases Register Allocation and Tree-Scan

Live-range splitting

Spill test:
r-colorable
$\iff$
$\mathrm{MAXLIVE} \leq r$

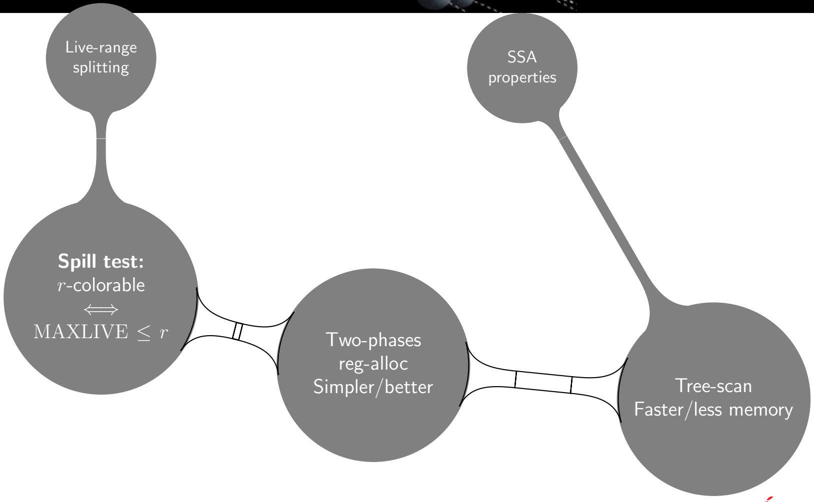
...under the assumption that live-range splitting is not possible.

In-Two-Phases Register Allocation and Tree-Scan



Spill test should not be coupled to global coloration anymore

In-Two-Phases Register Allocation and Tree-Scan



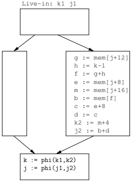
Tree-shaped SSA live-ranges allows for tree-based allocation schemes

“Spilling easier on a BB than on a general CFG”

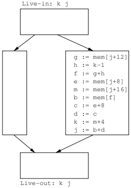
Basic block

```
Live-in: 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+d
Live-out: k' j'
```

SSA code



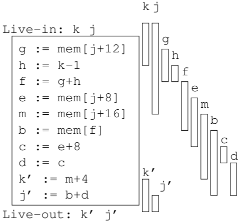
General CFG



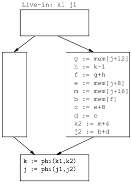
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

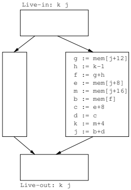
Basic block



SSA code



General CFG

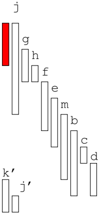


- MAXLIVE $\leq r$
- Linear scan

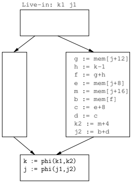
“Spilling easier on a BB than on a general CFG”

Basic block

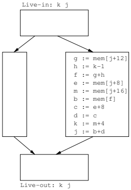
```
Live-in: k j
 g  := mem[j+12]
 h  := k-1
 f  := g+h
 e  := mem[j+8]
 m  := mem[j+16]
 b  := mem[f]
 c  := e+8
 d  := c
 k'  := m+4
 j'  := b+d
Live-out: k' j'
```



SSA code



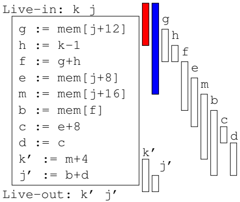
General CFG



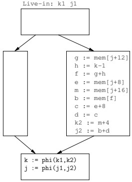
- MAXLIVE $\leq r$
- Linear scan

"Spilling easier on a BB than on a general CFG"

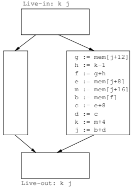
Basic block



SSA code



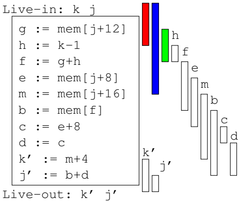
General CFG



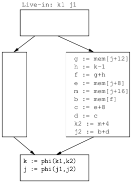
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

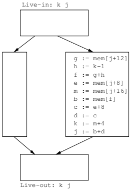
Basic block



SSA code



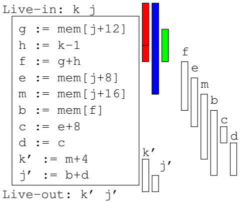
General CFG



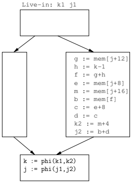
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

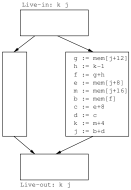
Basic block



SSA code



General CFG

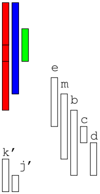


- MAXLIVE $\leq r$
- Linear scan

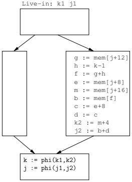
“Spilling easier on a BB than on a general CFG”

Basic block

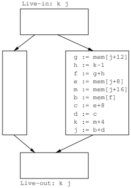
```
Live-in: k j
g  := mem[j+12]
h  := k-1
f  := g+h
e  := mem[j+8]
m  := mem[j+16]
b  := mem[f]
c  := e+8
d  := c
k'  := m+4
j'  := b+d
Live-out: k' j'
```



SSA code



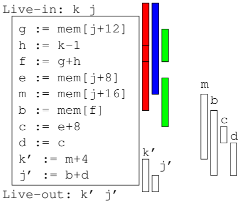
General CFG



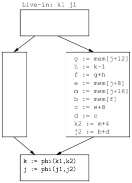
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

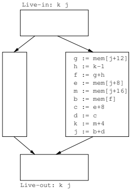
Basic block



SSA code



General CFG

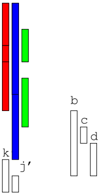


- MAXLIVE $\leq r$
- Linear scan

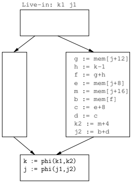
“Spilling easier on a BB than on a general CFG”

Basic block

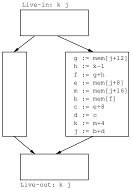
```
Live-in: k j
g  := mem[j+12]
h  := k-1
f  := g+h
e  := mem[j+8]
m  := mem[j+16]
b  := mem[f]
c  := e+8
d  := c
k'  := m+4
j'  := b+d
Live-out: k' j'
```



SSA code



General CFG

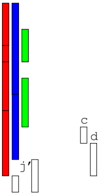


- MAXLIVE $\leq r$
- Linear scan

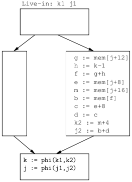
“Spilling easier on a BB than on a general CFG”

Basic block

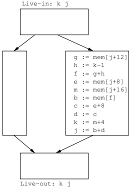
```
Live-in: k j
g  := mem[j+12]
h  := k-1
f  := g+h
e  := mem[j+8]
m  := mem[j+16]
b  := mem[f]
c  := e+8
d  := c
k'  := m+4
j'  := b+d
Live-out: k' j'
```



SSA code



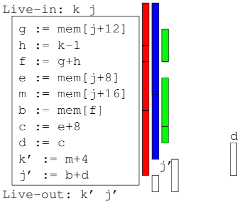
General CFG



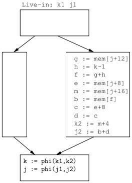
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

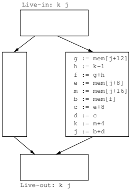
Basic block



SSA code



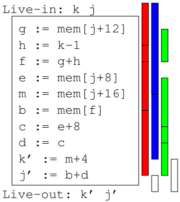
General CFG



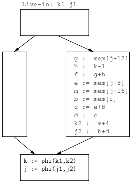
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

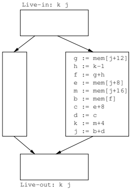
Basic block



SSA code



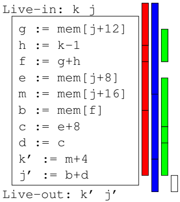
General CFG



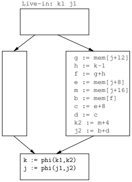
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

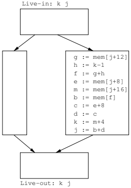
Basic block



SSA code



General CFG



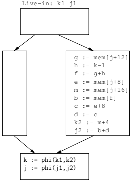
- MAXLIVE $\leq r$
- Linear scan

"Spilling easier on a BB than on a general CFG"

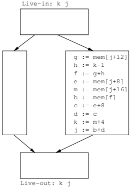
Basic block

```
Live-in: 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+d
Live-out: k' j'
```

SSA code



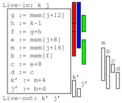
General CFG



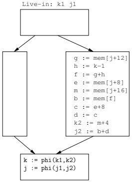
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

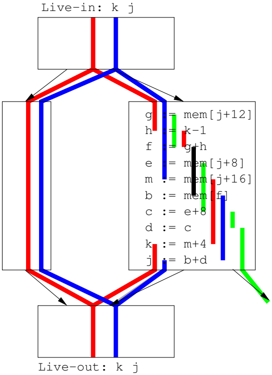
BB



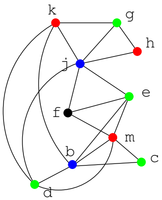
SSA



General control flow graph



Interference graph

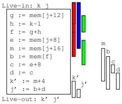


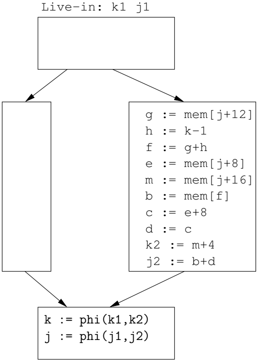
- Coloring test
- Greedy coloring

“Under SSA: the dominance tree”

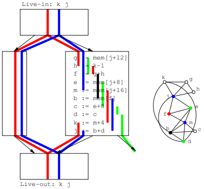
Static single assignment form

BB





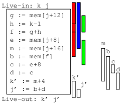
CFG

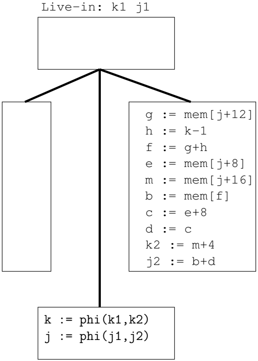


- MAXLIVE $\leq r$
- Tree scan

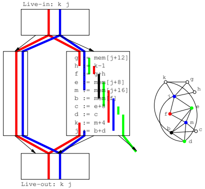
Static single assignment form

BB





CFG

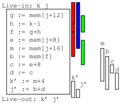


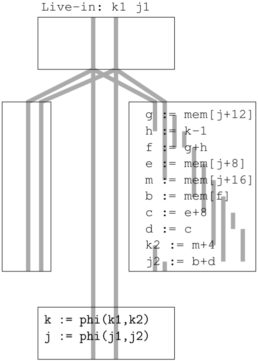
- MAXLIVE $\leq r$
- Tree scan

"Under SSA: the dominance tree"

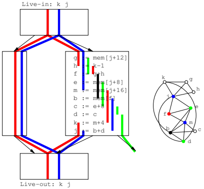
Static single assignment form

BB





CFG

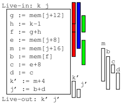


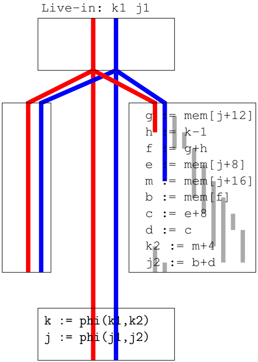
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

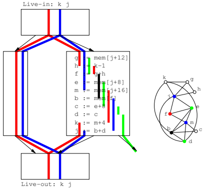
Static single assignment form

BB





CFG

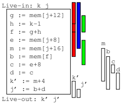


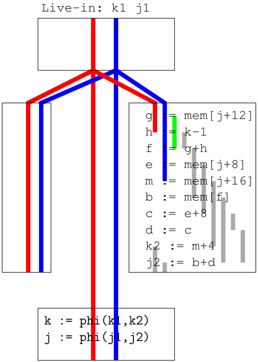
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

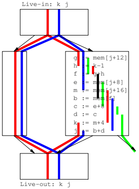
Static single assignment form

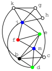
BB





CFG



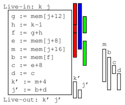


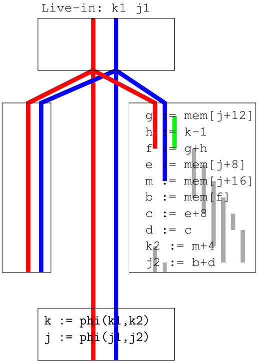
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

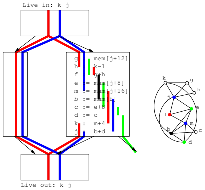
Static single assignment form

BB





CFG



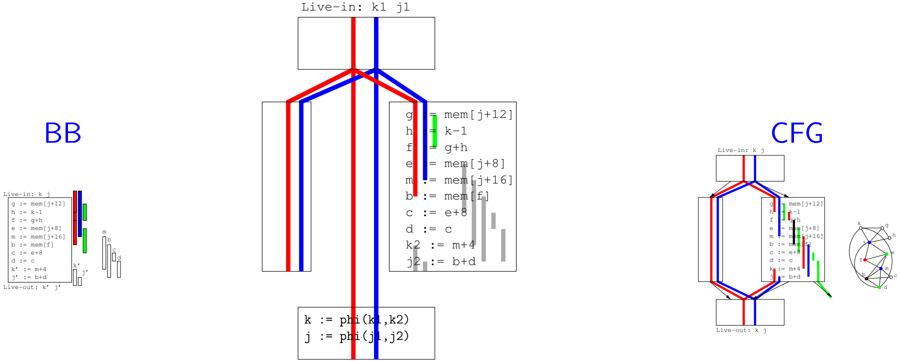
- MAXLIVE $\leq r$
- Tree scan

"Under SSA: the dominance tree"

Static single assignment form

BB

CFG

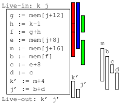


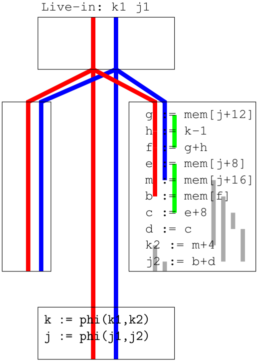
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

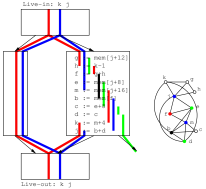
Static single assignment form

BB





CFG

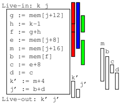


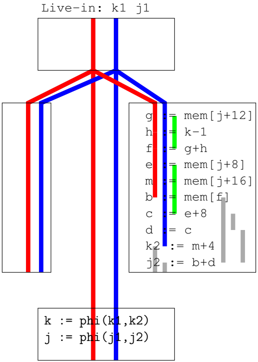
- MAXLIVE $\leq r$
- Tree scan

"Under SSA: the dominance tree"

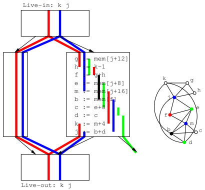
Static single assignment form

BB





CFG

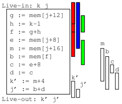


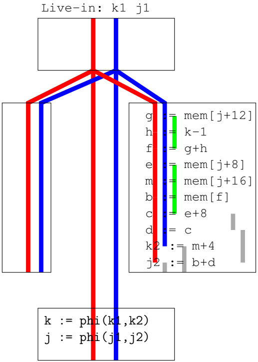
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

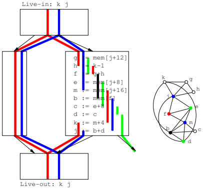
Static single assignment form

BB





CFG

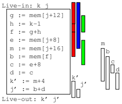


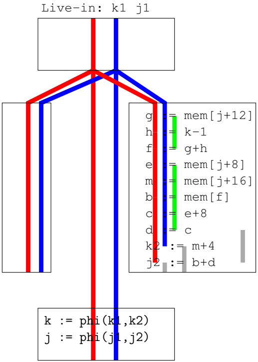
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

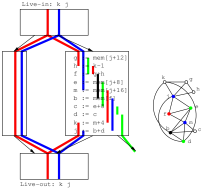
Static single assignment form

BB





CFG

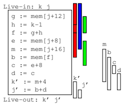


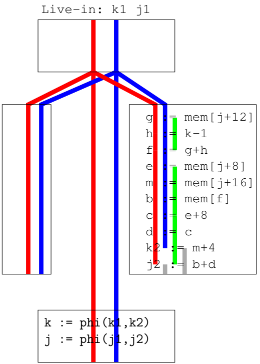
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

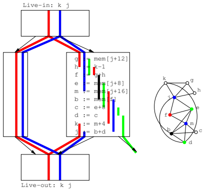
Static single assignment form

BB





CFG

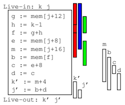


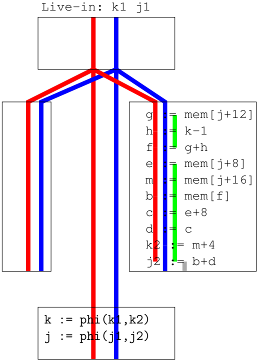
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

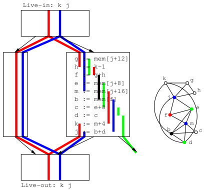
Static single assignment form

BB





CFG

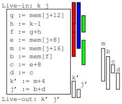


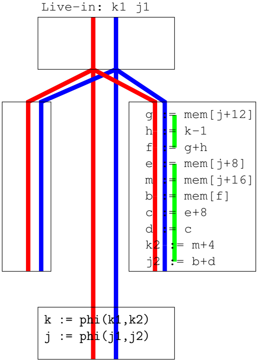
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

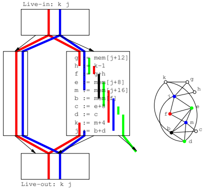
Static single assignment form

BB





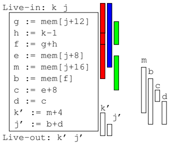
CFG



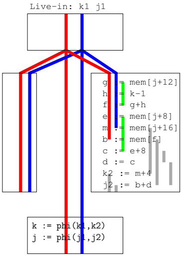
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

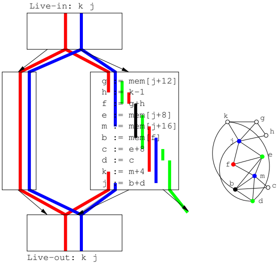
Basic block



SSA form



General CFG



- MAXLIVE $\leq r$
- Linear scan

- MAXLIVE $\leq r$
- Tree scan

- Coloring test
- Greedy coloring

Why coalescing?

Goal of coalescing

Removing the register-to-register copies [`move` $a \leftarrow b$]

Numerous copies due to:

- live-range splitting to avoid spilling

Why coalescing?

Goal of coalescing

Removing the register-to-register copies [`move` $a \leftarrow b$]

Numerous copies due to:

- live-range splitting to avoid spilling
- register constraints

$$\boxed{\begin{array}{l} a \leftarrow \ldots \\ b \leftarrow \ldots \\ c \leftarrow f(a,b) \end{array}} \rightsquigarrow \boxed{\begin{array}{l} a \leftarrow \ldots \\ b \leftarrow \ldots \\ \text{move } \mathrm{R}_0, a \\ \text{move } \mathrm{R}_1, b \\ \text{call } f \\ \text{move } c, \mathrm{R}_0 \end{array}}$$

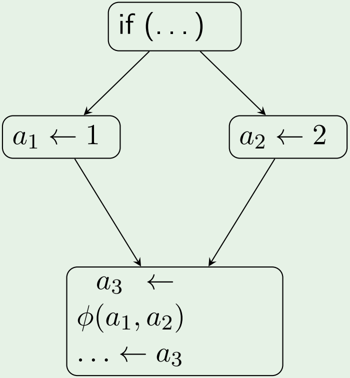
Why coalescing?

Goal of coalescing

Removing the register-to-register copies [`move` $a \leftarrow b$]

Numerous copies due to:

- live-range splitting to avoid spilling
- register constraints
- SSA destruction



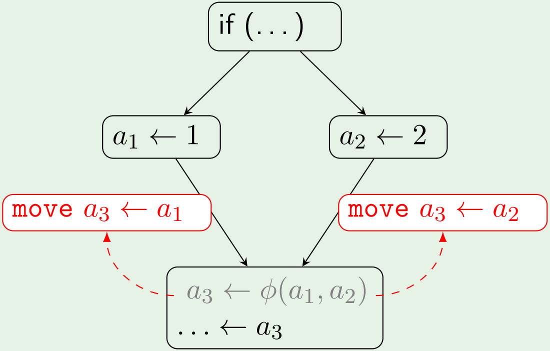
Why coalescing?

Goal of coalescing

Removing the register-to-register copies [`move` $a \leftarrow b$]

Numerous copies due to:

- live-range splitting to avoid spilling
- register constraints
- SSA destruction



The past

Live-range splitting already considered in the past:

- MAXLIVE registers are sufficient if aggressive splitting is performed: already pointed out by Fabri, and Cytron & Ferrante (but did not mention the possible need of critical edge splitting).
- Briggs tried SSA-based splitting (PhD thesis)

The past

Live-range splitting already considered in the past:

- MAXLIVE registers are sufficient if aggressive splitting is performed: already pointed out by Fabri, and Cytron & Ferrante (but did not mention the possible need of critical edge splitting).
- Briggs tried SSA-based splitting (PhD thesis)

But,

- These ideas were not exploited (coalescing not good enough?)
- Instead: sophisticated algorithms that split on demand when the coloring fails, e.g., live-range splitting (Cooper and Simpson), optimistic (Park and Moon), priority based (Chow and Hennessy, etc.

The present

Lower MAXLIVE → split live-ranges → Assign & coalesce

1. Spill so as to lower register pressure to $\leq$ #registers
   - Split so that interference graph is greedy-k-colorable ($\sim$ k-colorable *à la* Chaitin)
2. Color + Coalesce to remove useless copies

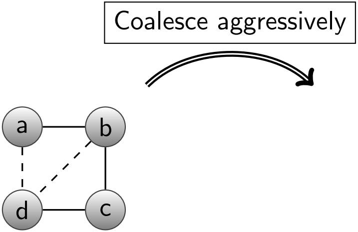
Exploit Greedy-k-colorable graph properties

SSA based live-range splitting leads to an interference graph that can be k-colored ($k =$ MAXLIVE) using Chaitin's simplification scheme.

Optimize coalescing separately from spilling

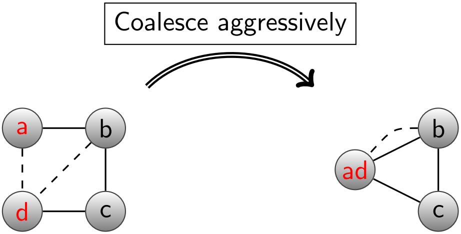
Aggressive coalescing

Aggressive coalescing may lead to spilling. Conservative coalescing takes colorability into account.



Aggressive coalescing

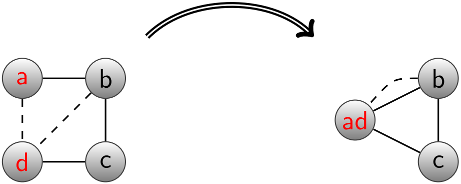
Aggressive coalescing may lead to spilling. Conservative coalescing takes colorability into account.



Aggressive coalescing

Aggressive coalescing may lead to spilling. Conservative coalescing takes colorability into account.

Coalesce aggressively

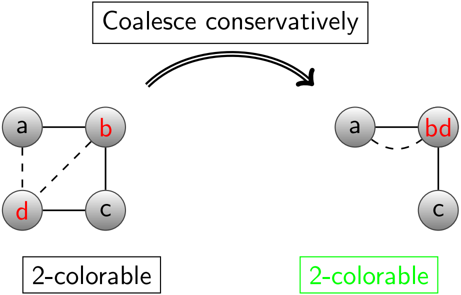


2-colorable

Not 2-colorable

Conservative coalescing

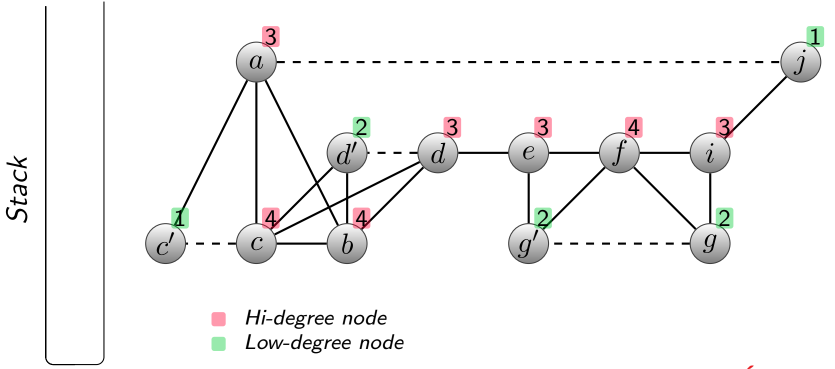
Aggressive coalescing may lead to spilling. Conservative coalescing takes colorability into account.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

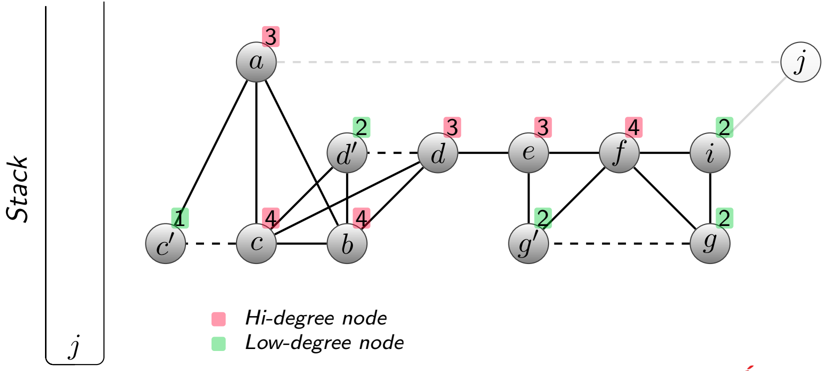
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

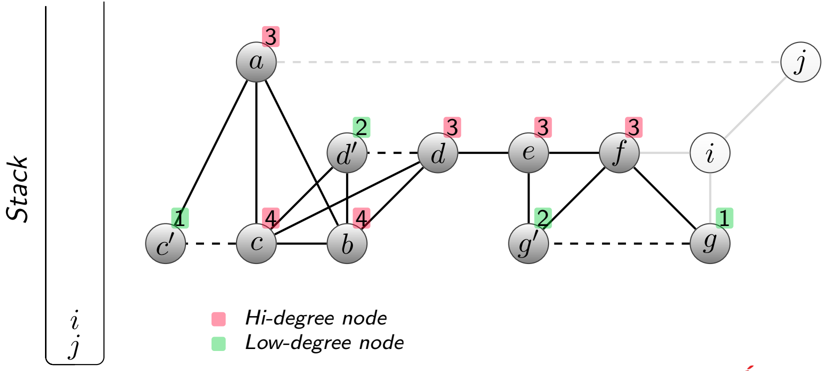
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

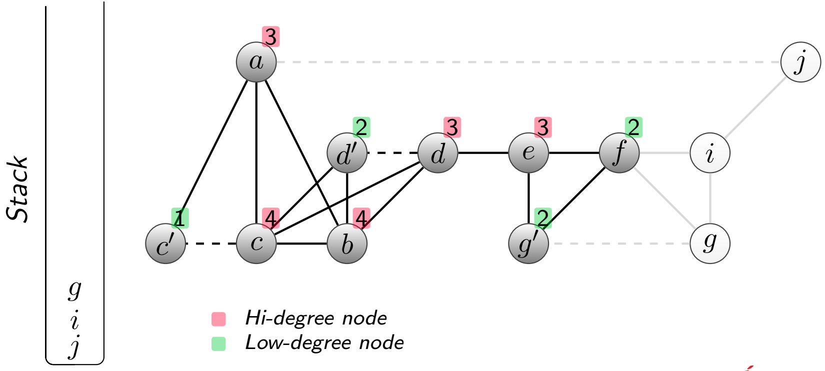
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

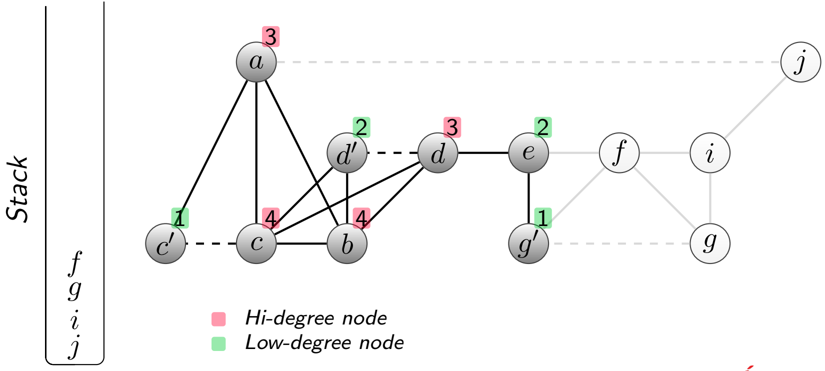
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

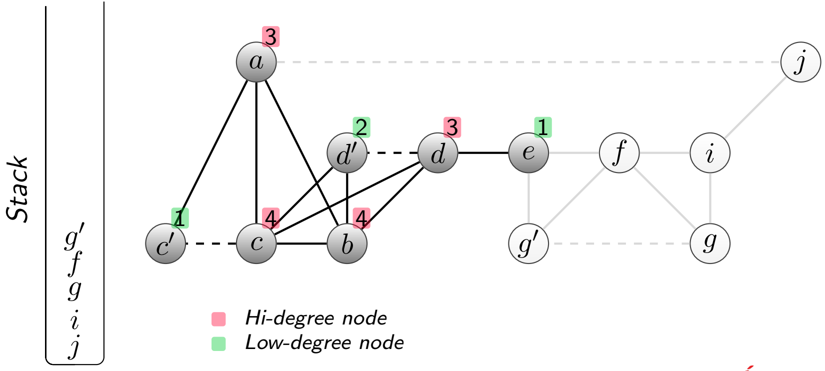
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

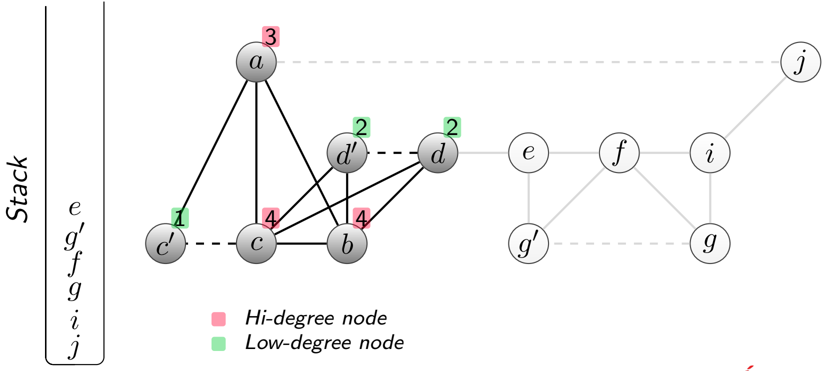
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

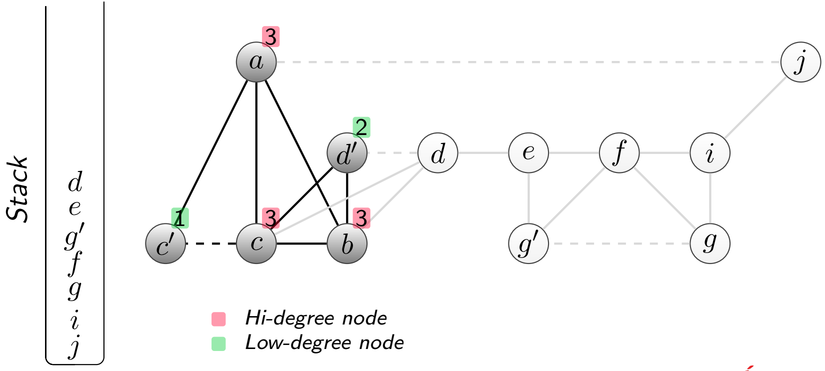
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

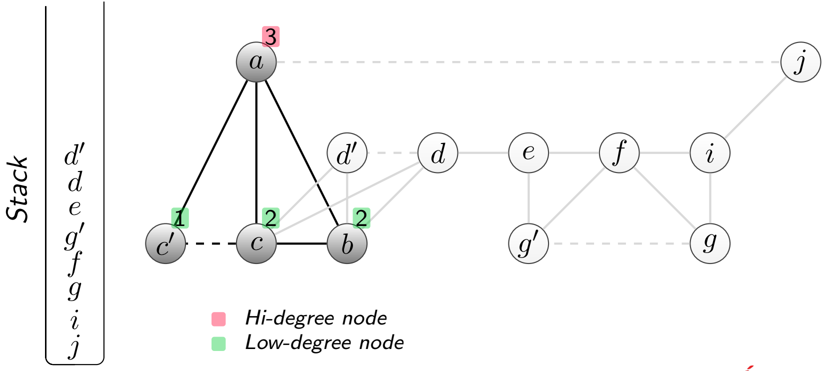
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

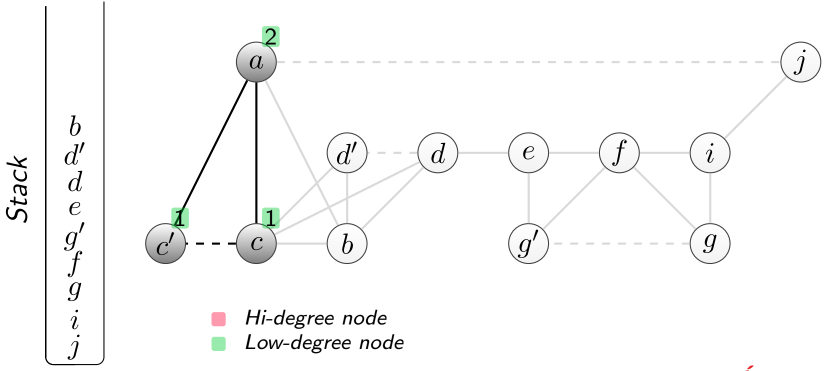
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

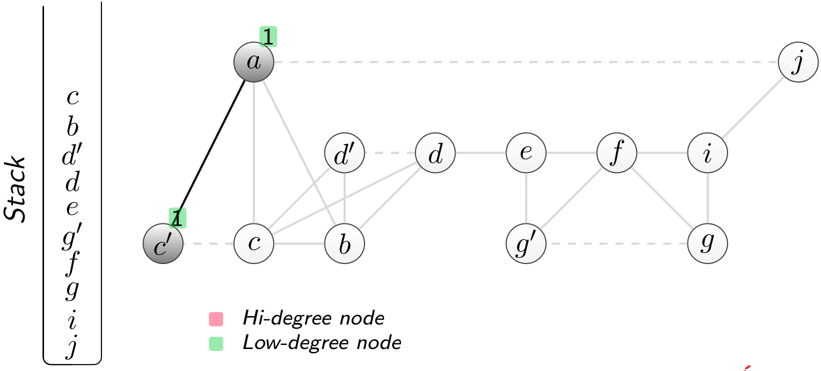
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

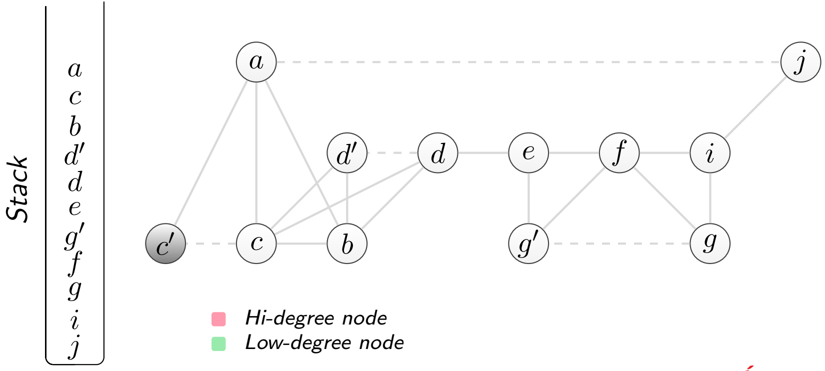
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

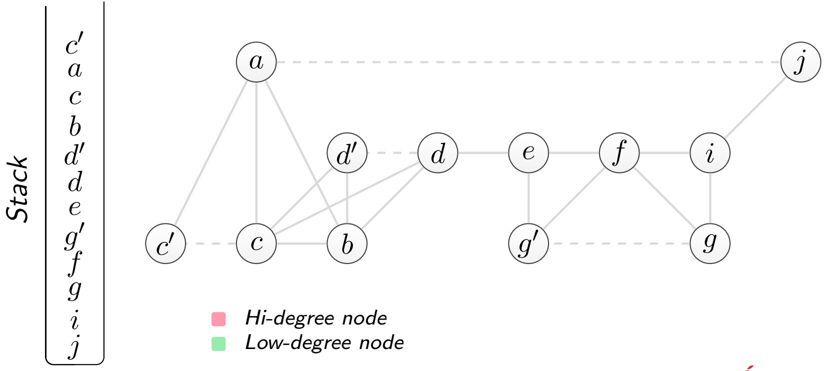
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

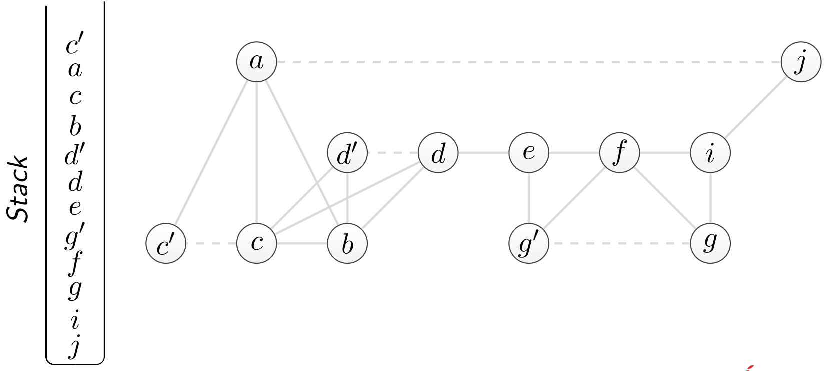
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

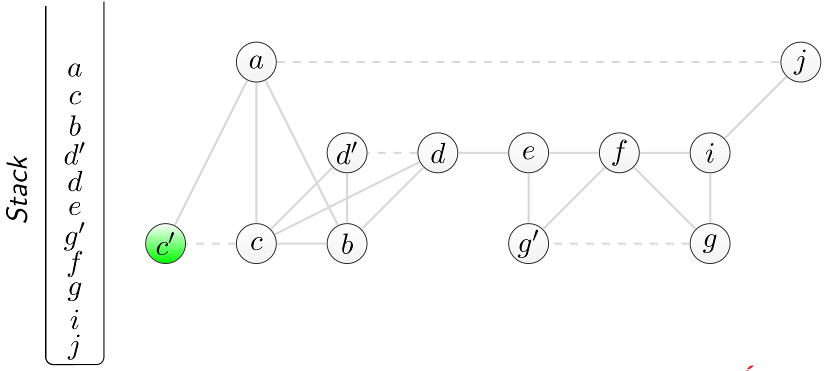
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

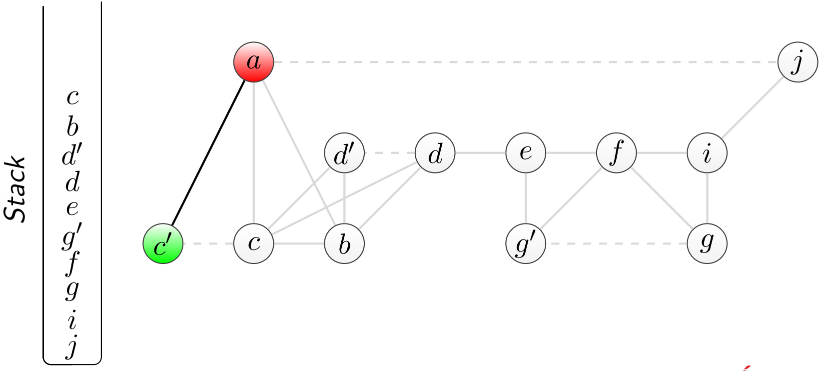
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

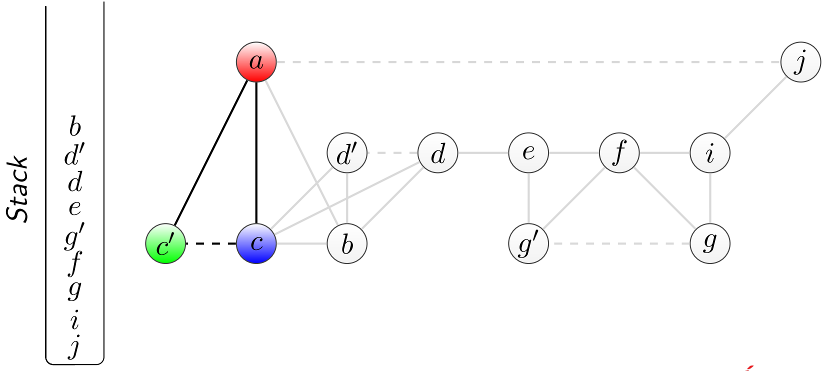
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

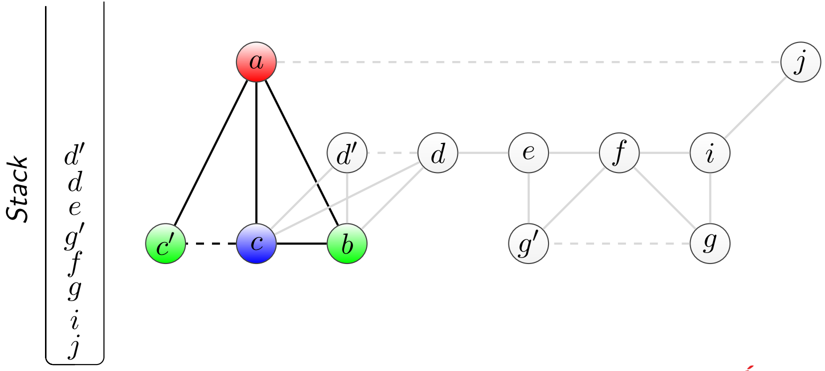
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

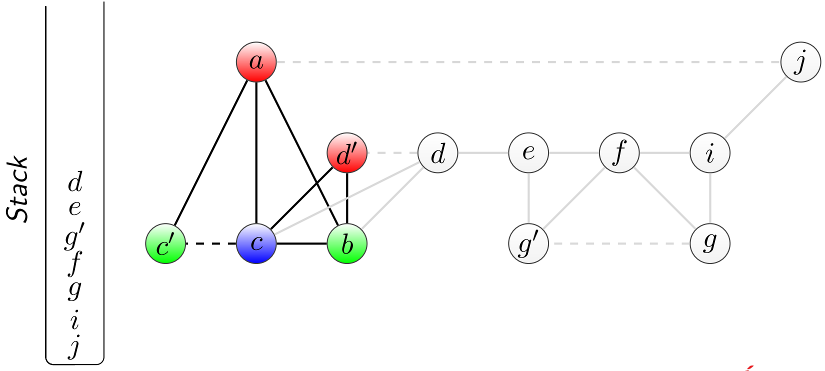
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

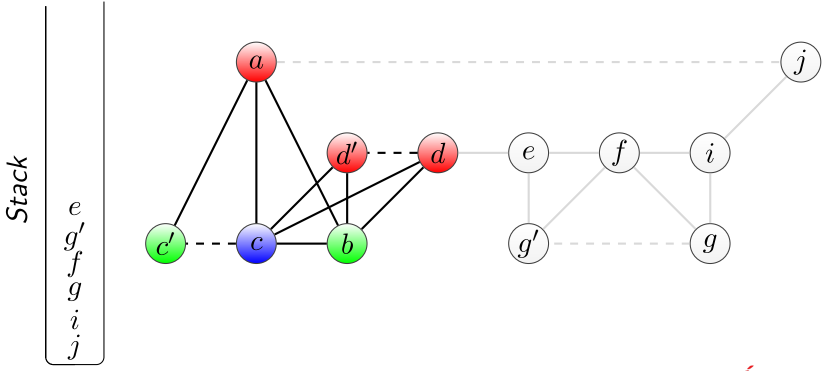
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

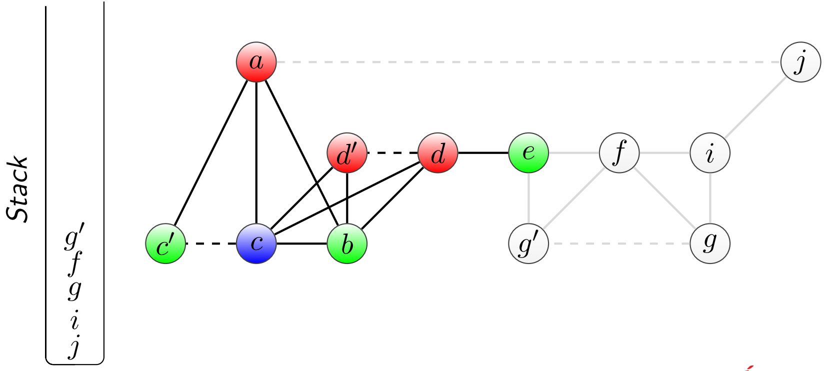
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

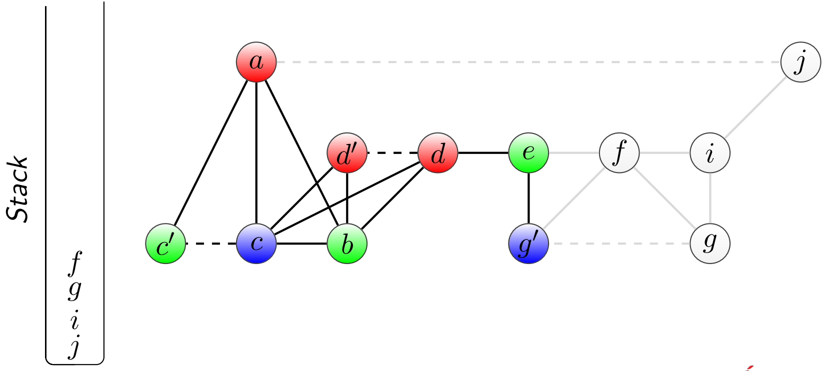
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

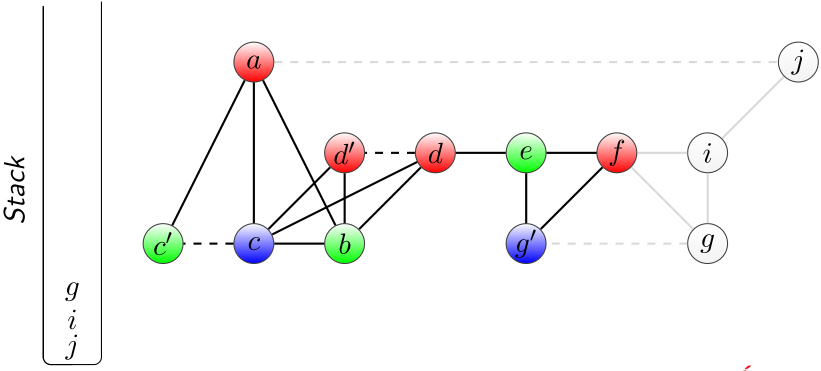
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

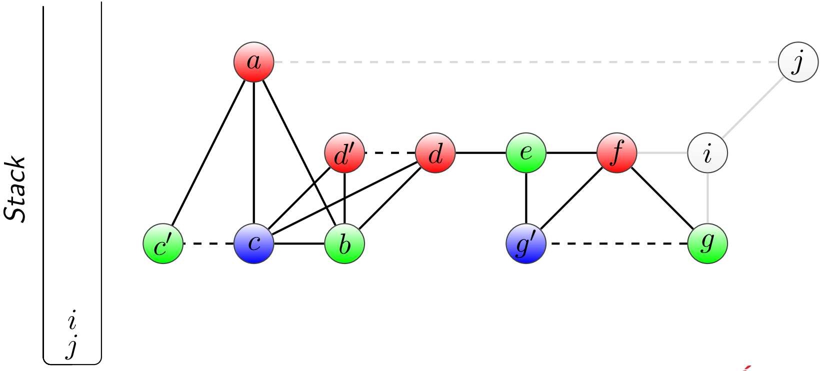
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

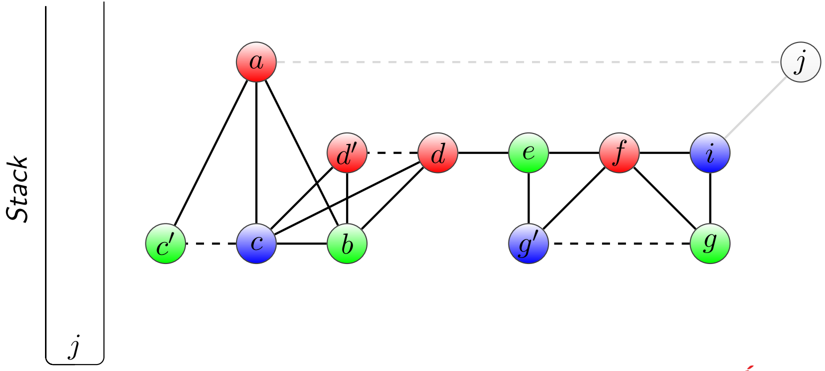
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

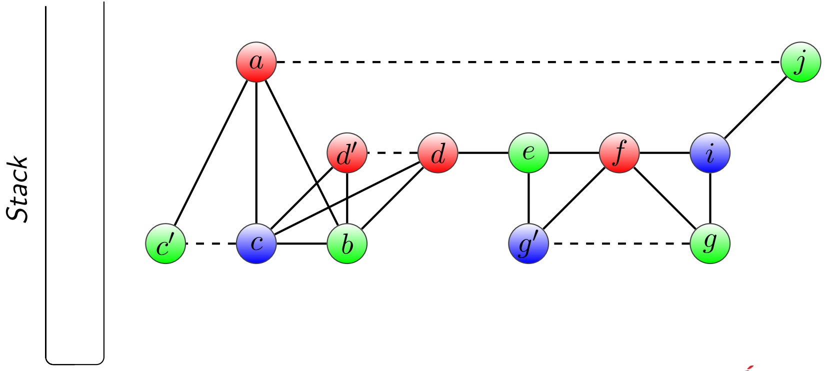
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

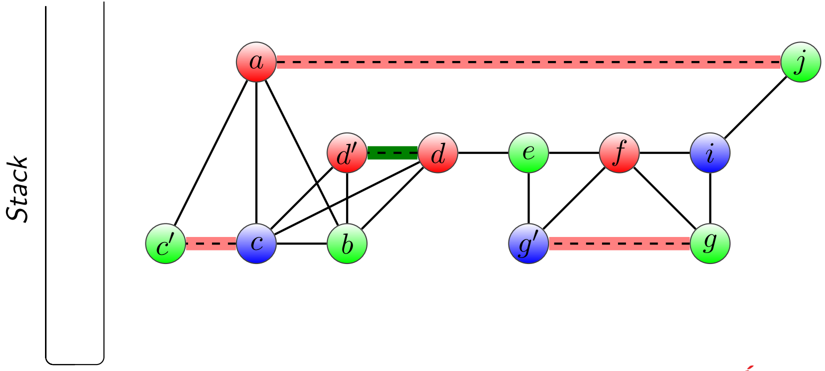
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Greedy-k-colorable graphs

k-colorability is hard to check, but greedy-k-colorability is easy.

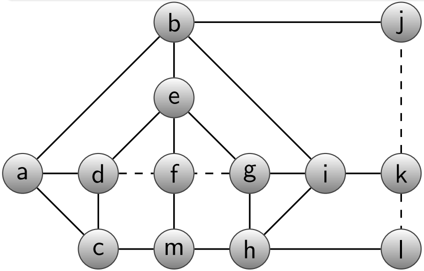
Check greedy-k-colorability: simplify nodes with $< k$ neighbors.



Incremental conservative coalescing

Finding the optimal subset of affinities is hard.

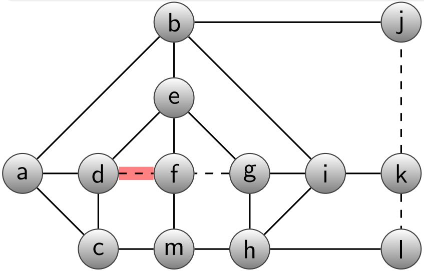
Algorithms do incremental conservative coalescing.



Incremental conservative coalescing

Finding the optimal subset of affinities is hard.

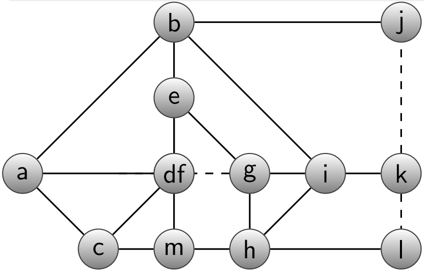
Algorithms do incremental conservative coalescing.



Incremental conservative coalescing

Finding the optimal subset of affinities is hard.

Algorithms do incremental conservative coalescing.

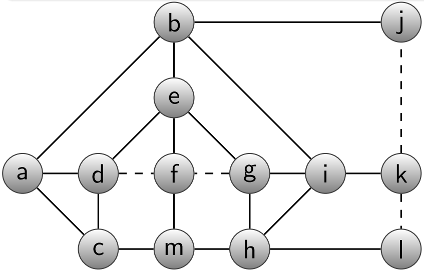


Not greedy-3-colorable

Incremental conservative coalescing

Finding the optimal subset of affinities is hard.

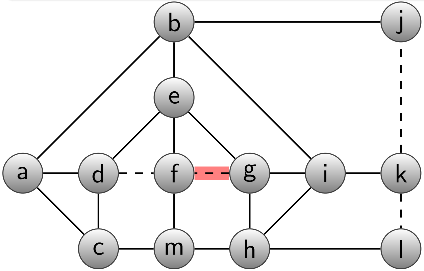
Algorithms do incremental conservative coalescing.



Incremental conservative coalescing

Finding the optimal subset of affinities is hard.

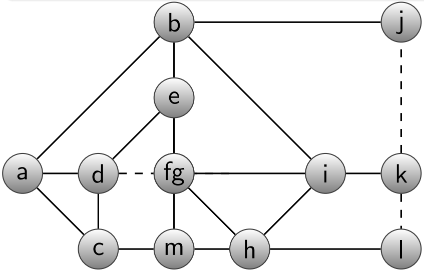
Algorithms do incremental conservative coalescing.



Incremental conservative coalescing

Finding the optimal subset of affinities is hard.

Algorithms do incremental conservative coalescing.

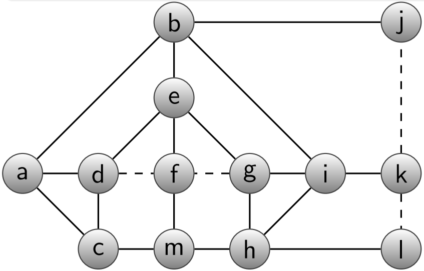


Not greedy-3-colorable

Incremental conservative coalescing

Finding the optimal subset of affinities is hard.

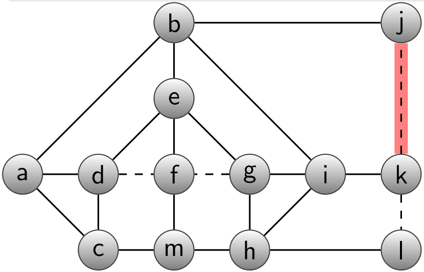
Algorithms do incremental conservative coalescing.



Incremental conservative coalescing

Finding the optimal subset of affinities is hard.

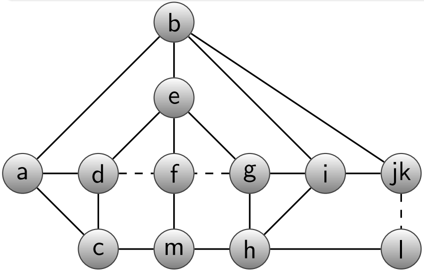
Algorithms do incremental conservative coalescing.



Incremental conservative coalescing

Finding the optimal subset of affinities is hard.

Algorithms do incremental conservative coalescing.

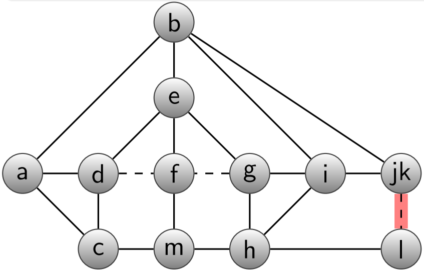


greedy-3-colorable

Incremental conservative coalescing

Finding the optimal subset of affinities is hard.

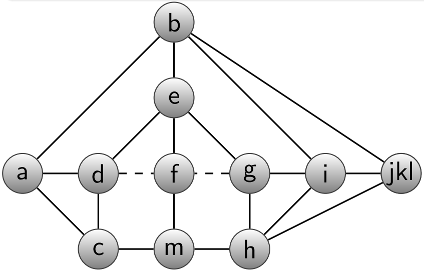
Algorithms do incremental conservative coalescing.



Incremental conservative coalescing

Finding the optimal subset of affinities is hard.

Algorithms do incremental conservative coalescing.

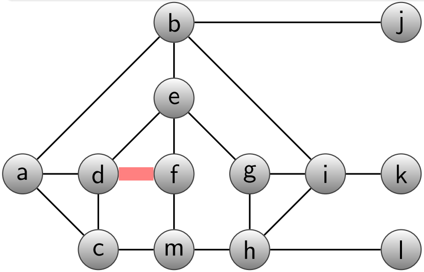


greedy-3-colorable

A gap

Incremental conservative is not optimal.

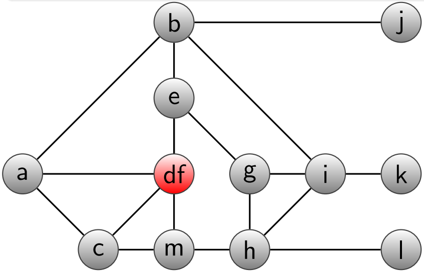
Greedy-k-colorable test might be stuck. Multiple node merging necessary to stay Greedy-k-colorable.



A gap

Incremental conservative is not optimal.

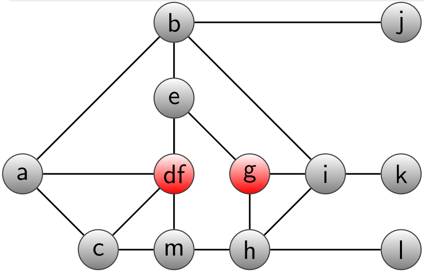
Greedy-k-colorable test might be stuck. Multiple node merging necessary to stay Greedy-k-colorable.



A gap

Incremental conservative is not optimal.

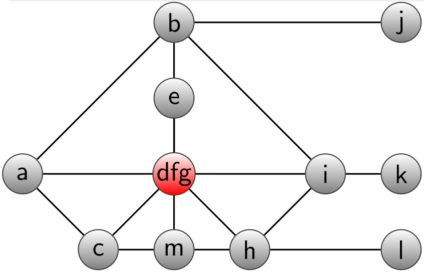
Greedy-k-colorable test might be stuck. Multiple node merging necessary to stay Greedy-k-colorable.



A gap

Incremental conservative is not optimal.

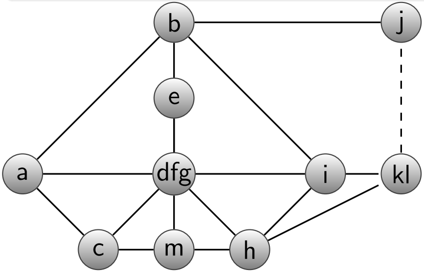
Greedy-k-colorable test might be stuck. Multiple node merging necessary to stay Greedy-k-colorable.



A gap

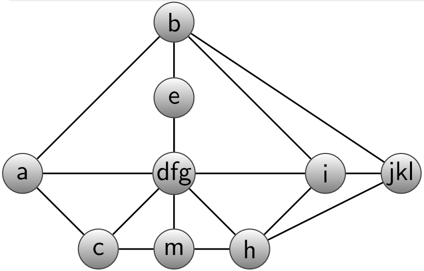
Incremental conservative is not optimal.

Greedy-k-colorable test might be stucked. Multiple node merging necessary to stay Greedy-k-colorable.



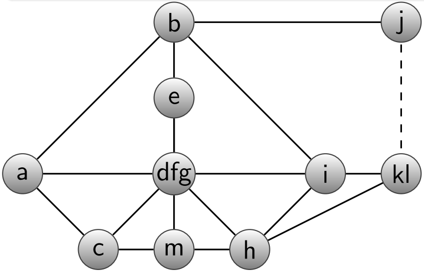
Aggressive + decoalescing

Aggressive + de-coalescing scheme: start from a completely aggressively coalesced graph, give up with some move until it gets Greedy-k-colorable again.



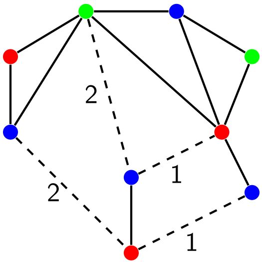
Aggressive + decoalescing

Aggressive + de-coalescing scheme: start from a completely aggressively coalesced graph, give up with some move until it gets Greedy-k-colorable again.



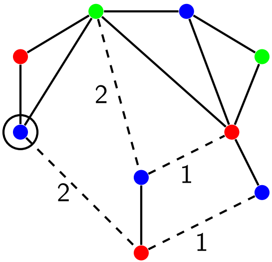
Recoloring

Coalescing (coloring) + recoloring scheme: start from a colored graph, try to assign move-related nodes the same color by recoloring its neighborhood.



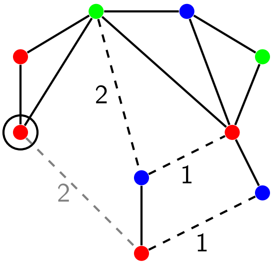
Recoloring

Coalescing (coloring) + recoloring scheme: start from a colored graph, try to assign move-related nodes the same color by recoloring its neighborhood.



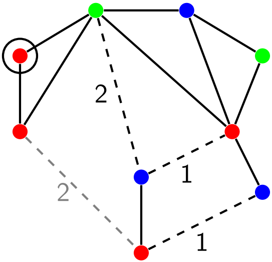
Recoloring

Coalescing (coloring) + recoloring scheme: start from a colored graph, try to assign move-related nodes the same color by recoloring its neighborhood.



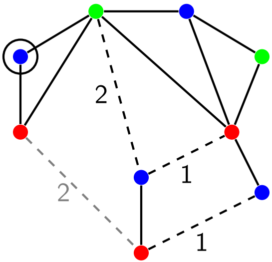
Recoloring

Coalescing (coloring) + recoloring scheme: start from a colored graph, try to assign move-related nodes the same color by recoloring its neighborhood.



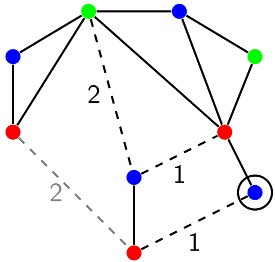
Recoloring

Coalescing (coloring) + recoloring scheme: start from a colored graph, try to assign move-related nodes the same color by recoloring its neighborhood.



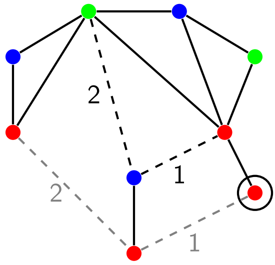
Recoloring

Coalescing (coloring) + recoloring scheme: start from a colored graph, try to assign move-related nodes the same color by recoloring its neighborhood.



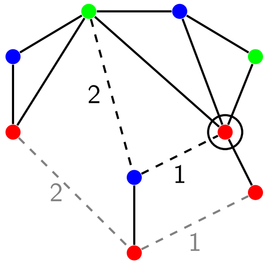
Recoloring

Coalescing (coloring) + recoloring scheme: start from a colored graph, try to assign move-related nodes the same color by recoloring its neighborhood.



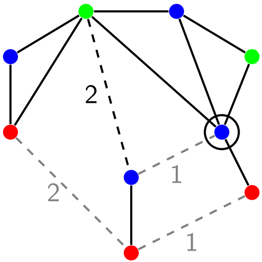
Recoloring

Coalescing (coloring) + recoloring scheme: start from a colored graph, try to assign move-related nodes the same color by recoloring its neighborhood.



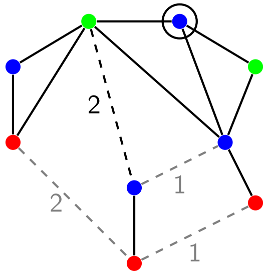
Recoloring

Coalescing (coloring) + recoloring scheme: start from a colored graph, try to assign move-related nodes the same color by recoloring its neighborhood.



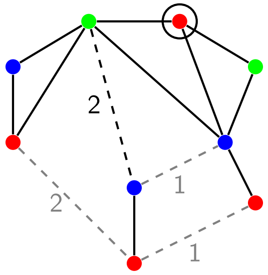
Recoloring

Coalescing (coloring) + recoloring scheme: start from a colored graph, try to assign move-related nodes the same color by recoloring its neighborhood.



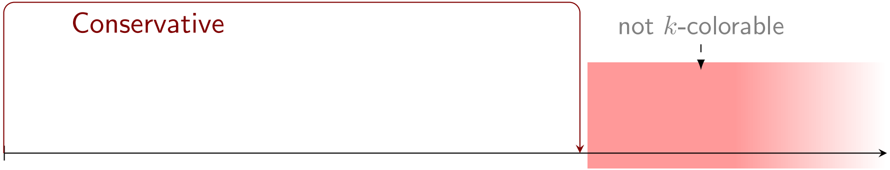
Recoloring

Coalescing (coloring) + recoloring scheme: start from a colored graph, try to assign move-related nodes the same color by recoloring its neighborhood.

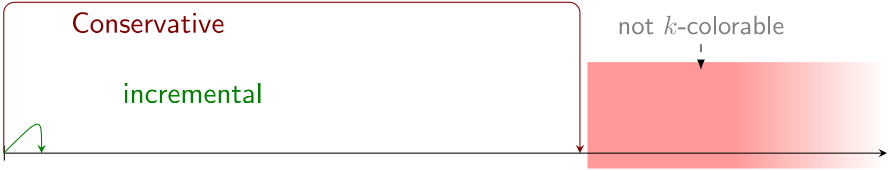


Different coalescing schemes

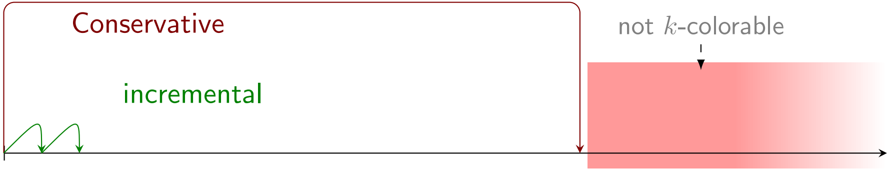
Different coalescing schemes



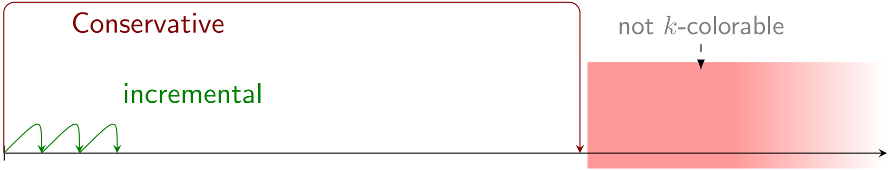
Different coalescing schemes



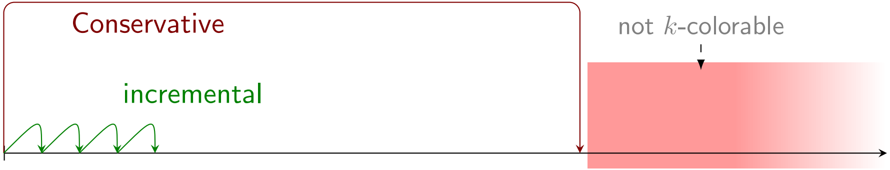
Different coalescing schemes



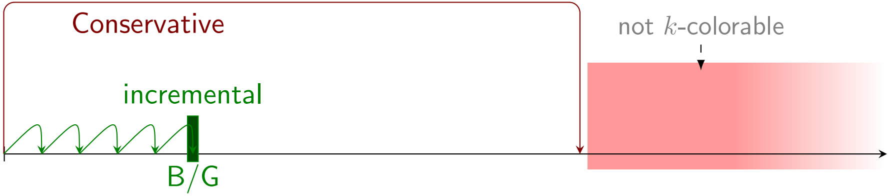
Different coalescing schemes



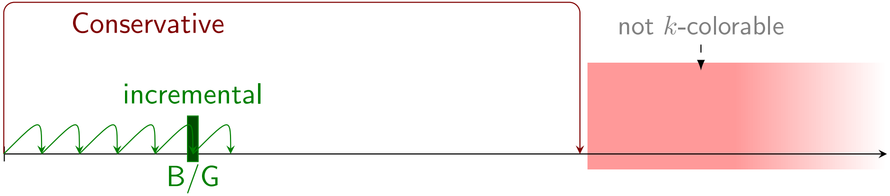
Different coalescing schemes



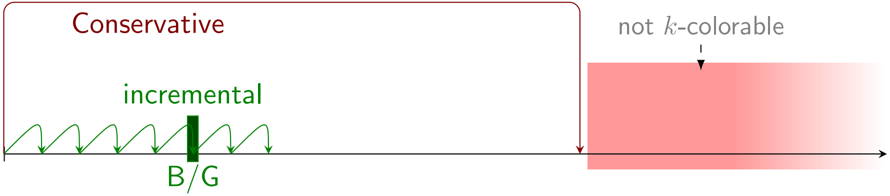
Different coalescing schemes



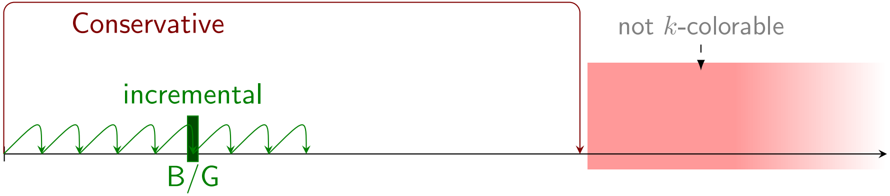
Different coalescing schemes



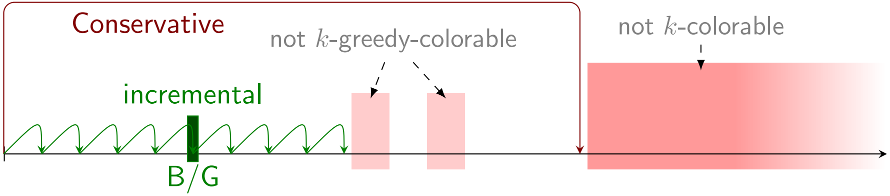
Different coalescing schemes



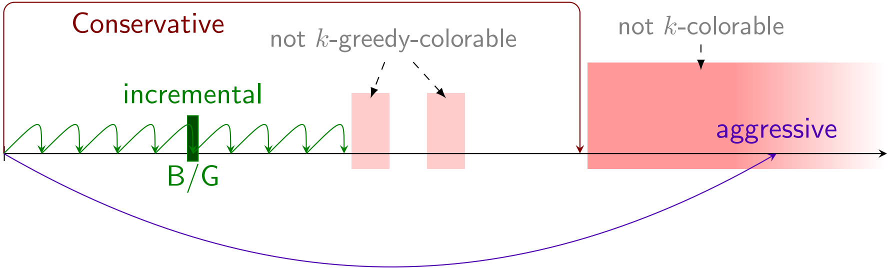
Different coalescing schemes



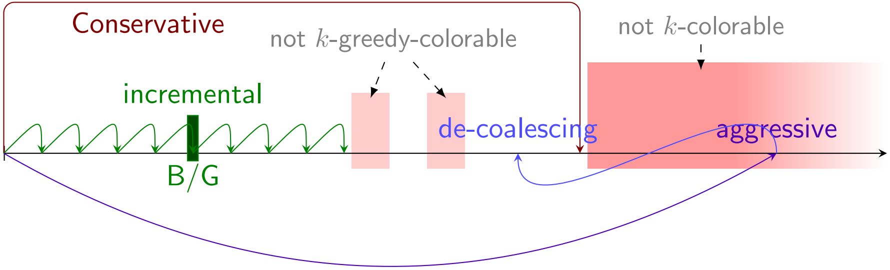
Different coalescing schemes



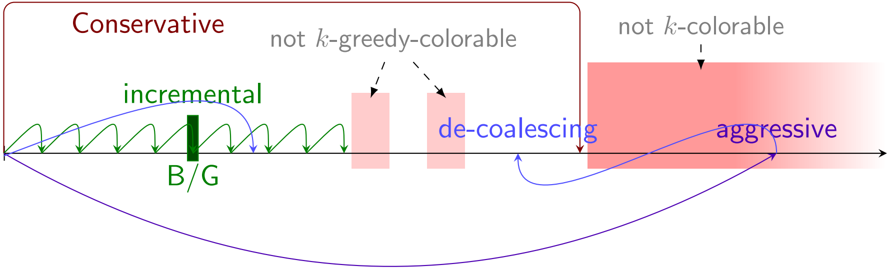
Different coalescing schemes



Different coalescing schemes

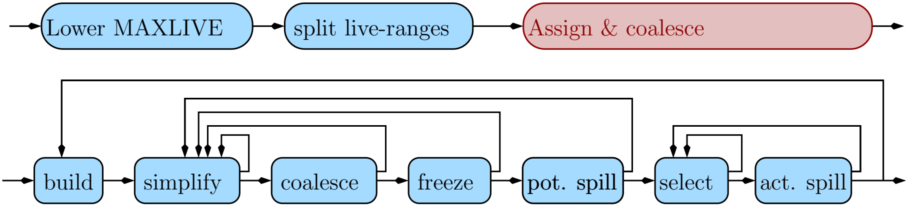


Different coalescing schemes



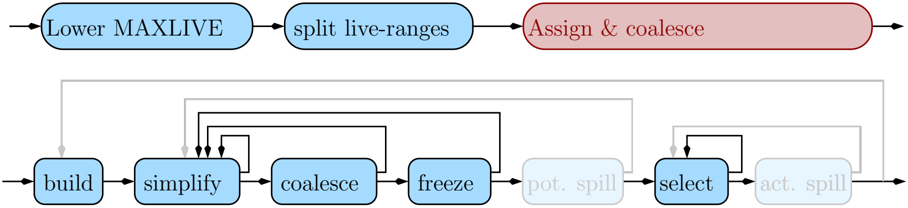
Iterated register coalescing

Iterated register coalescing uses an incremental scheme.



Iterated register coalescing

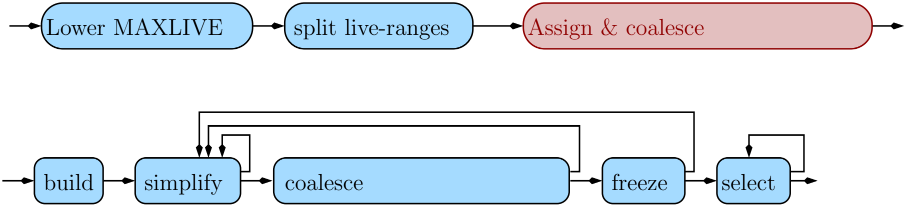
Iterated register coalescing uses an incremental scheme.



- spilling and coalescing are not intermixed

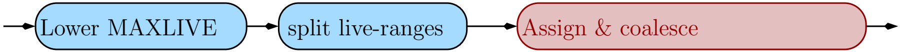
Iterated register coalescing

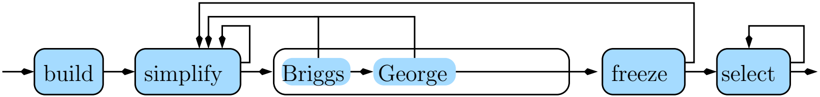
Iterated register coalescing uses an incremental scheme.



- spilling and coalescing are not intermixed

Iterated register coalescing uses an incremental scheme.

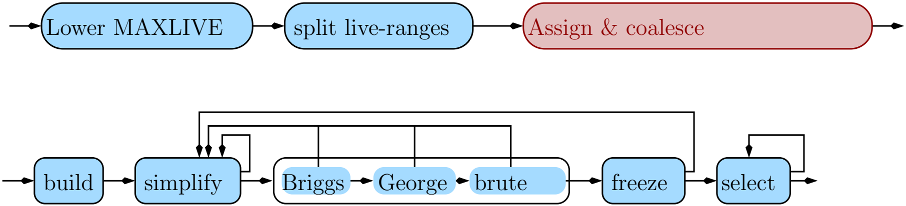




- spilling and coalescing are not intermixed

Iterated register coalescing

Iterated register coalescing uses an incremental scheme.



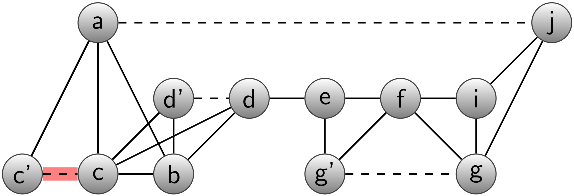
- spilling and coalescing are not intermixed
- if local coalescing tests (B/G) fail, run brute-force test

Incremental conservative local rules

- Briggs

Briggs

Resulting node has $< k$ high-degree neighbours

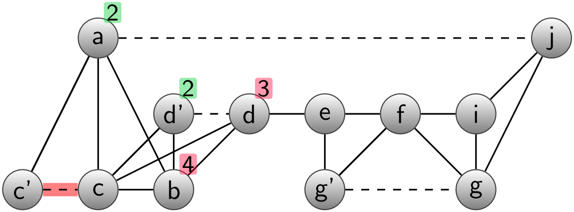


Incremental conservative local rules

- Briggs

Briggs

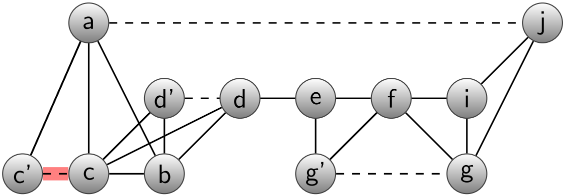
Resulting node has $< k$ high-degree neighbours



- Briggs

Briggs

Resulting node has $< k$ high-degree neighbours

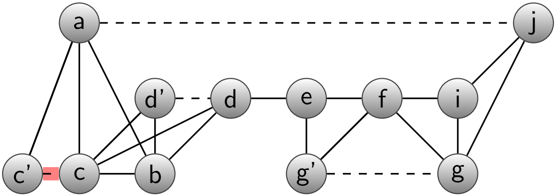


Incremental conservative local rules

- Briggs

Briggs

Resulting node has $< k$ high-degree neighbours

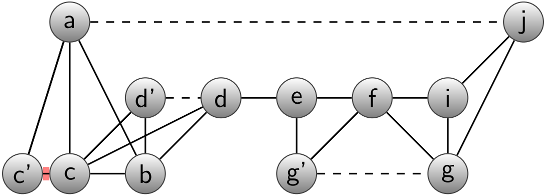


Incremental conservative local rules

- Briggs

Briggs

Resulting node has $< k$ high-degree neighbours

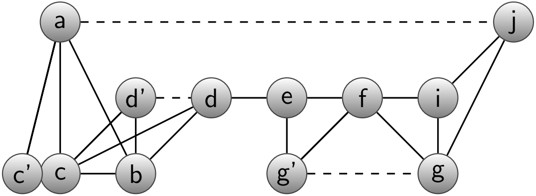


Incremental conservative local rules

- Briggs

Briggs

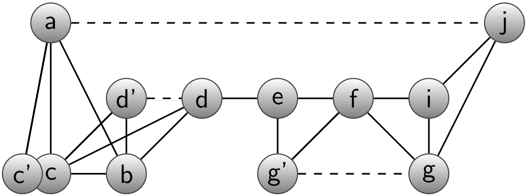
Resulting node has $< k$ high-degree neighbours



- Briggs

Briggs

Resulting node has $< k$ high-degree neighbours

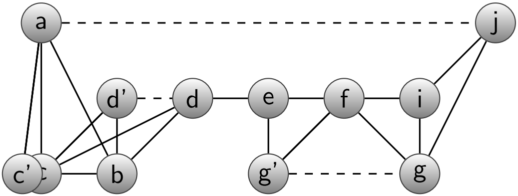


Incremental conservative local rules

- Briggs

Briggs

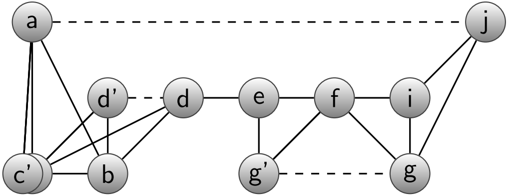
Resulting node has $< k$ high-degree neighbours



- Briggs

Briggs

Resulting node has $< k$ high-degree neighbours

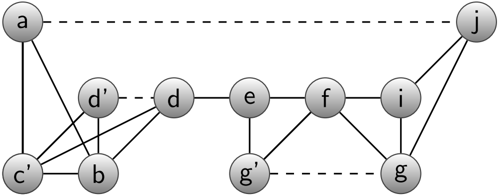


Incremental conservative local rules

- Briggs

Briggs

Resulting node has $< k$ high-degree neighbours

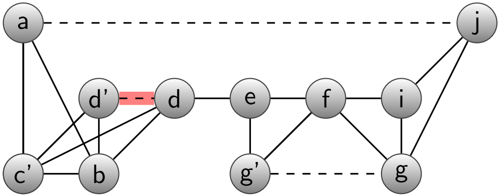


Incremental conservative local rules

- Briggs
- George

George

All high-degree neighbours are neighbours of the other node

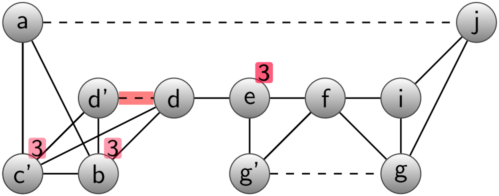


Incremental conservative local rules

- Briggs
- George

George

All high-degree neighbours are neighbours of the other node

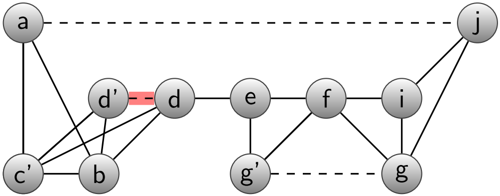


Incremental conservative local rules

- Briggs
- George

George

All high-degree neighbours are neighbours of the other node

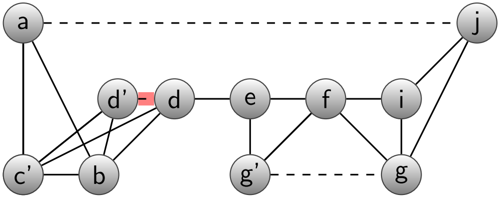


Incremental conservative local rules

- Briggs
- George

George

All high-degree neighbours are neighbours of the other node

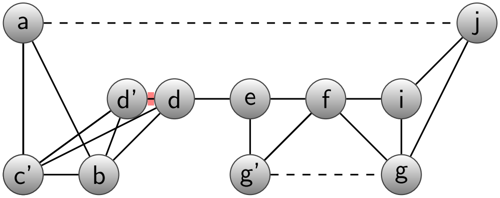


Incremental conservative local rules

- Briggs
- George

George

All high-degree neighbours are neighbours of the other node

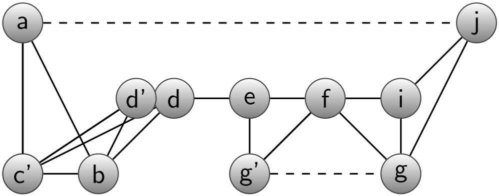


Incremental conservative local rules

- Briggs
- George

George

All high-degree neighbours are neighbours of the other node

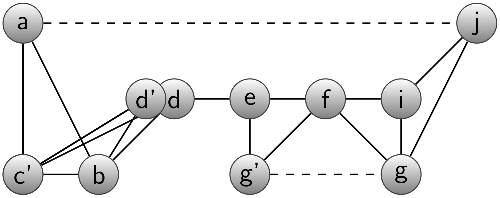


Incremental conservative local rules

- Briggs
- George

George

All high-degree neighbours are neighbours of the other node

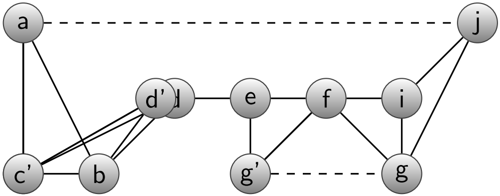


Incremental conservative local rules

- Briggs
- George

George

All high-degree neighbours are neighbours of the other node

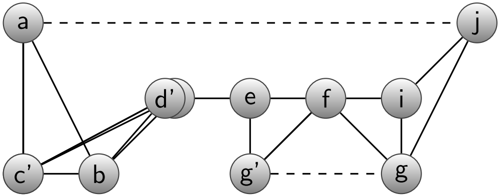


Incremental conservative local rules

- Briggs
- George

George

All high-degree neighbours are neighbours of the other node

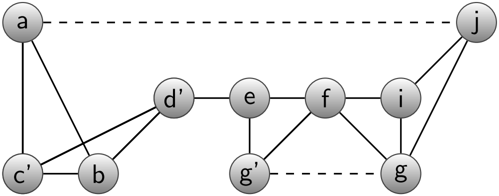


Incremental conservative local rules

- Briggs
- George

George

All high-degree neighbours are neighbours of the other node

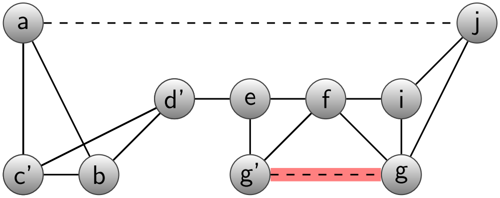


Incremental conservative local rules

- Briggs
- George
- Brute-force

Brute-force

Merge the nodes and check if resulting graph is greedy-$k$colorable

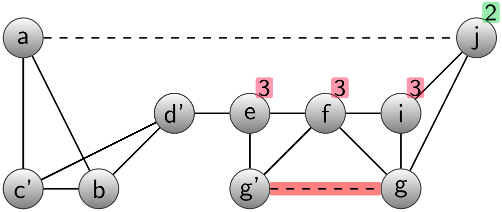


Incremental conservative local rules

- Briggs
- George
- Brute-force

Brute-force

Merge the nodes and check if resulting graph is greedy-$k$colorable

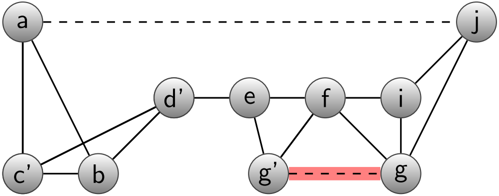


Incremental conservative local rules

- Briggs
- George
- Brute-force

Brute-force

Merge the nodes and check if resulting graph is greedy-$k$colorable

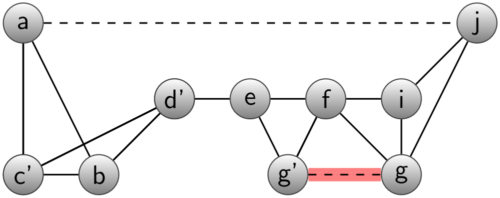


Incremental conservative local rules

- Briggs
- George
- Brute-force

Brute-force

Merge the nodes and check if resulting graph is greedy-$k$colorable

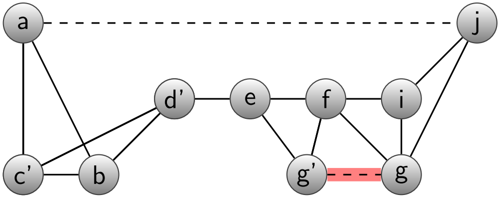


Incremental conservative local rules

- Briggs
- George
- Brute-force

Brute-force

Merge the nodes and check if resulting graph is greedy-$k$colorable

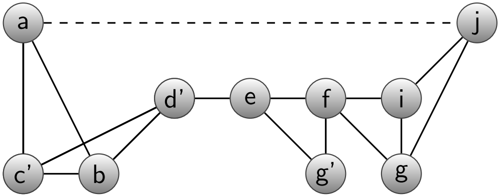


Incremental conservative local rules

- Briggs
- George
- Brute-force

Brute-force

Merge the nodes and check if resulting graph is greedy-$k$colorable

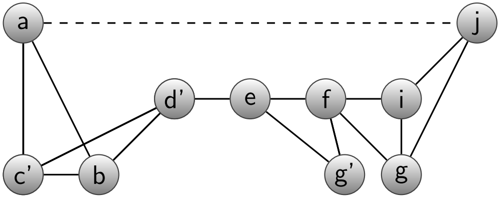


Incremental conservative local rules

- Briggs
- George
- Brute-force

Brute-force

Merge the nodes and check if resulting graph is greedy-$k$colorable

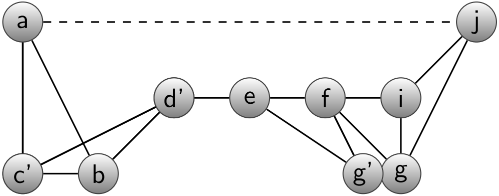


Incremental conservative local rules

- Briggs
- George
- Brute-force

Brute-force

Merge the nodes and check if resulting graph is greedy-$k$colorable

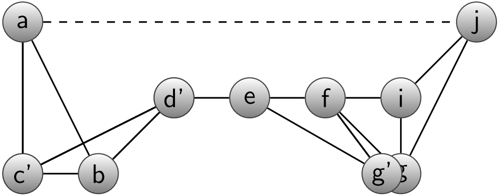


Incremental conservative local rules

- Briggs
- George
- Brute-force

Brute-force

Merge the nodes and check if resulting graph is greedy-$k$colorable

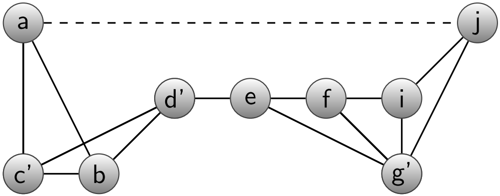


Incremental conservative local rules

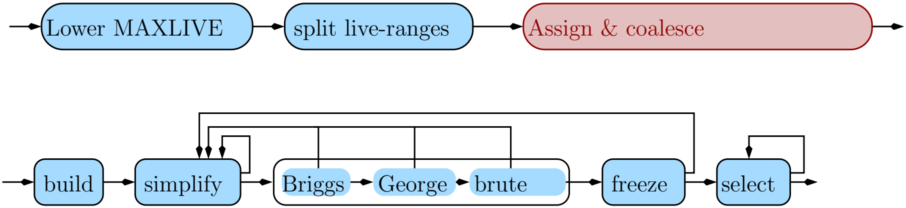
- Briggs
- George
- Brute-force

Brute-force

Merge the nodes and check if resulting graph is greedy-$k$colorable

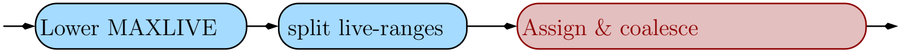


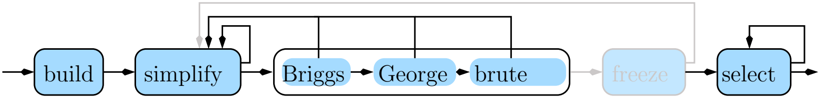
The brute-force solution. HOWTO



- spilling and coalescing are not intermixed
- if local coalescing tests (B/G) fail, run brute-force test

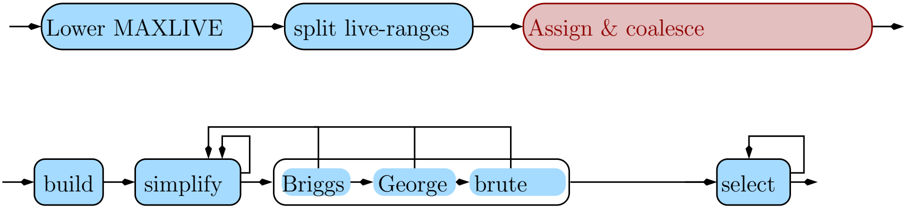
The brute-force solution. HOWTO





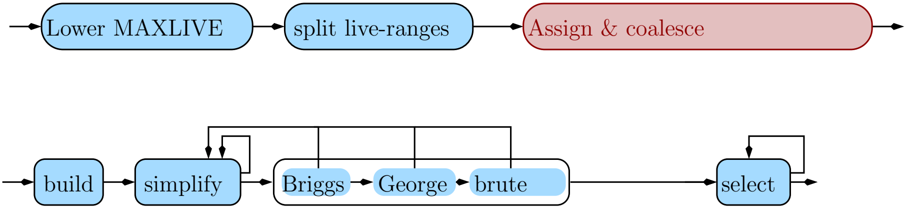
- spilling and coalescing are not intermixed
- if local coalescing tests (B/G) fail, run brute-force test
- each affinity is considered only once

The brute-force solution. HOWTO



- spilling and coalescing are not intermixed
- if local coalescing tests (B/G) fail, run brute-force test
- each affinity is considered only once

The brute-force solution. HOWTO

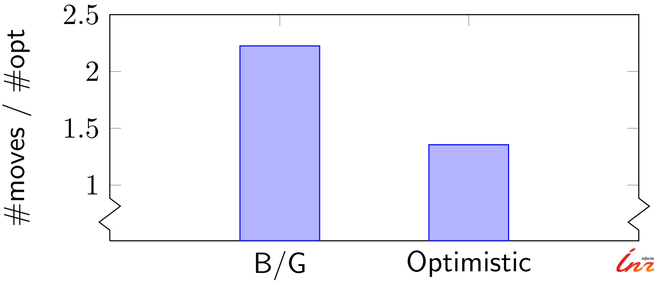


- spilling and coalescing are not intermixed
- if local coalescing tests (B/G) fail, run brute-force test
- each affinity is considered only once
- less work-lists to manipulate

Appel & George's experiments

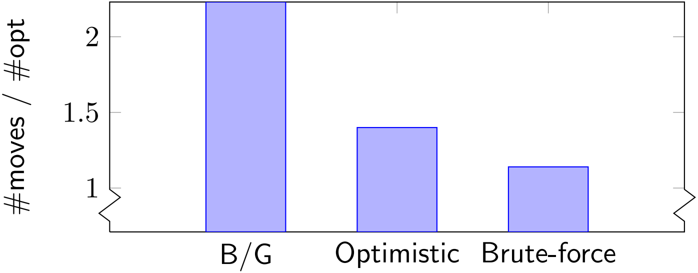
From the standard ML new Jersey benchmark suite:

- Split everywhere. Compute and apply optimal spilling.
- B/G: Iterated conservative coalescing (Briggs/George rules).
- ➠ Coalescing challenge.
- Optimistic: adapted optimistic (only for graphs with $degree < k$).
- Optimal: ILP-based solution (Grund and Hack).



Experimental results. Quality

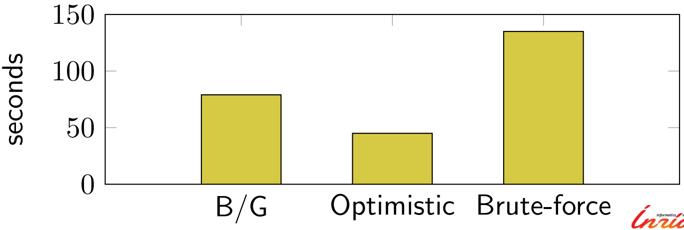
Quality:



Experimental results. Runtime

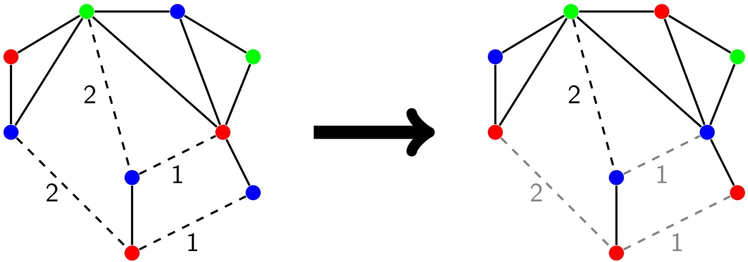
Total time (for all graphs):

- Briggs/George's test: $O(\frac{E}{V})$
- Brute-force test: $O(E)$
- In iterated scheme, if Briggs/George test fails: affinity inserted many times in the (ordered) work-list to be tested again.
- In our simplified scheme with brute-force coalescing, if Briggs/George test fails: only one brute-force test.



Recoloring

→ Lower MAXLIVE → split live-ranges → color / coalesce → coalesce (recolor)



- Optimistically try to assign move-related nodes the same color
- Resolve color clashes recursively through the graph

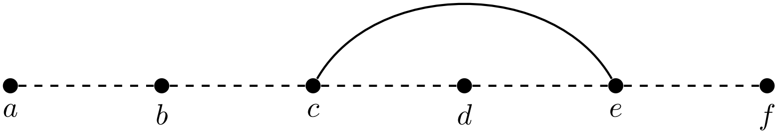
Recoloring

Conduct recoloring as a transaction

- rollback if color clash cannot be resolved.
- In a transaction:
  - Do not recolor already recolored nodes to avoid recursion
  - Look at all interference neighbors
  - Try to find a color that is not used in the neighborhood
  - If no such color is available: Pick the least used one
  - If node is constrained: Set of available colors restricted

Recoloring

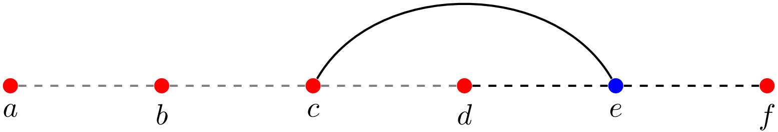
Conduct transaction affinity component by affinity component



- Before recoloring: segregate components into interference-free chunks: already agree upon a set of lost affinities (aggressive coalescing)
- Try to satisfy affinities chunk by chunk

Recoloring

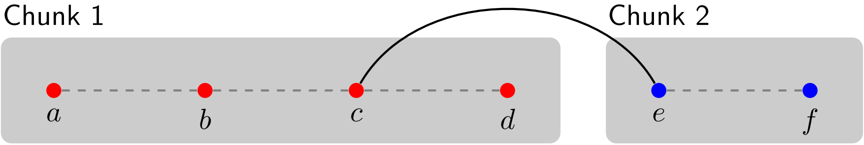
Conduct transaction affinity component by affinity component



- Before recoloring: segregate components into interference-free chunks: already agree upon a set of lost affinities (aggressive coalescing)
- Try to satisfy affinities chunk by chunk

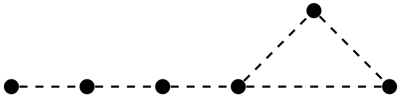
Recoloring

Conduct transaction affinity component by affinity component



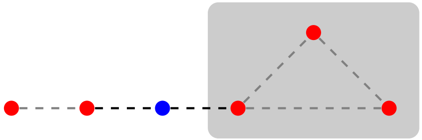
- Before recoloring: segregate components into interference-free chunks: already agree upon a set of lost affinities (aggressive coalescing)
- Try to satisfy affinities chunk by chunk

Recoloring a chunk



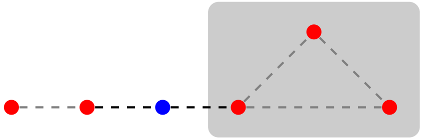
- Define a sequence of colors to try for the chunk
- For each color c in that sequence
  - Try to recolor each node in the chunk to c

Recoloring a chunk



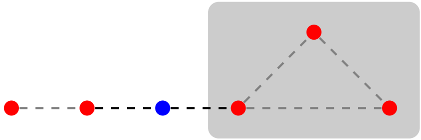
- Define a sequence of colors to try for the chunk
- For each color c in that sequence
  - Try to recolor each node in the chunk to c
  - Memorize the best sub-chunk in the chunk

Recoloring a chunk



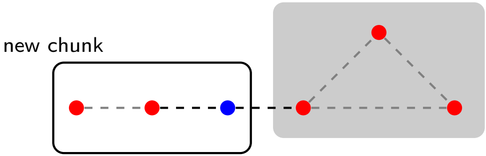
- Define a sequence of colors to try for the chunk
- For each color c in that sequence
  - Try to recolor each node in the chunk to c
  - Memorize the best sub-chunk in the chunk
- Select the color with the best sub-chunk

Recoloring a chunk



- Define a sequence of colors to try for the chunk
- For each color c in that sequence
  - Try to recolor each node in the chunk to c
  - Memorize the best sub-chunk in the chunk
- Select the color with the best sub-chunk
- Make the color of those nodes permanent

Recoloring a chunk



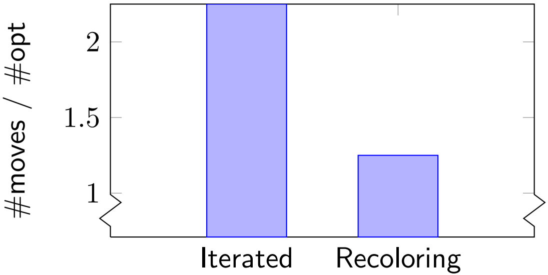
- Define a sequence of colors to try for the chunk
- For each color c in that sequence
  - Try to recolor each node in the chunk to c
  - Memorize the best sub-chunk in the chunk
- Select the color with the best sub-chunk
- Make the color of those nodes permanent
- Create new chunks from the rest and add them to the queue

Libfirm compiler (`http://www.libfirm.org`)

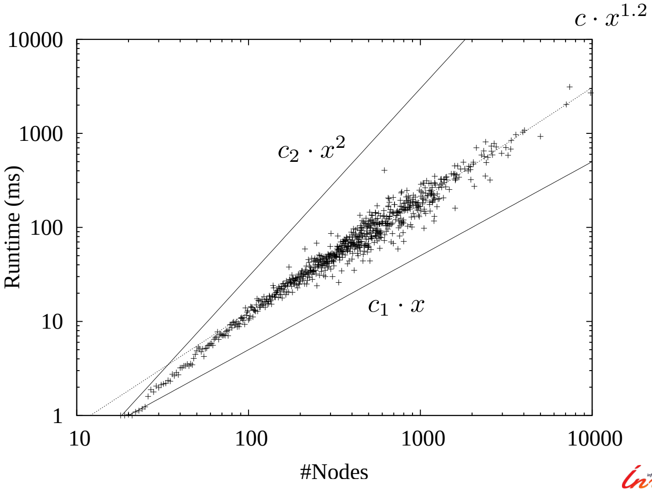
- CINT2000 benchmark suite
  - Graph-based SSA IR
  - SSA-based register allocator
- Pentium 4 2.4GHz, 1GB RAM
- Affinity weights given by estimated execution frequencies
- Comparison to optimal solutions obtained from an ILP solver

Experimental results. Quality

Quality:



Experimental results. Runtime



Outline

Introduction

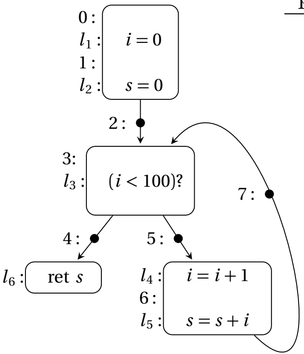
- Data-flow analysis: discover facts (information) that are true about a program. Bind to $\mathit{Variables} \times \mathit{PogramPoints}$.
- Static Single Information (SSI) property: IR such that information of a variable invariant along its whole live-range
- ϕ-functions split live-ranges where reaching definitions collide: SSA fulfills SSI property for constant analysis. Not for class inference (backward from uses).
- Extended SSA: SSI property for forward analysis flowing from definitions and conditional tests.
- SSU: SSI property for backward analysis flowing from uses

Can we generalize?

Sparse Analysis

Non-relational (dense) analysis: bind information to pairs *Variables* × *ProgPoints*

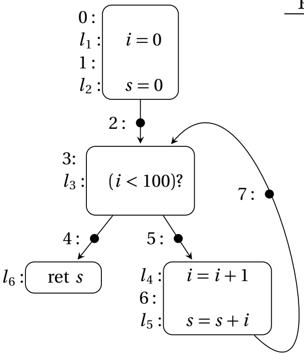
$i = 0;$
$s = 0;$
`while` $(i < 100)$
  $i = i + 1;$
  $s = s + i;$
`ret`



prog. point	$[i]$	$[s]$
0	$\top$	$\top$
1	$[0, 0]$	$\top$
2	$[0, 0]$	$[0, 0]$
3	$[0, 100]$	$[0, +\infty[$
4	$[100, 100]$	$[0, +\infty[$
5	$[0, 99]$	$[0, +\infty[$
6	$[0, 100]$	$[0, +\infty[$
7	$[0, 100]$	$[0, +\infty[$

Range Analysis: $[v]^p$ intervals of possible values variable v might assume at program point p

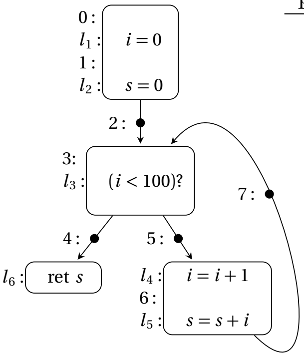
$i = 0;$
$s = 0;$
`while` $(i < 100)$
  $i = i + 1;$
  $s = s + i;$
`ret`



prog. point	$[i]$	$[s]$
0	$\top$	$\top$
1	$[0, 0]$	$\top$
2	$[0, 0]$	$[0, 0]$
3	$[0, 100]$	$[0, +\infty[$
4	$[100, 100]$	$[0, +\infty[$
5	$[0, 99]$	$[0, +\infty[$
6	$[0, 100]$	$[0, +\infty[$
7	$[0, 100]$	$[0, +\infty[$

Redundancies: e.g. $[i]^1 = [i]^2$; because identity transfer function for $[i]$ from 1 to 2.

$i = 0;$
$s = 0;$
`while` $(i < 100)$
  $i = i + 1;$
  $s = s + i;$
`ret`

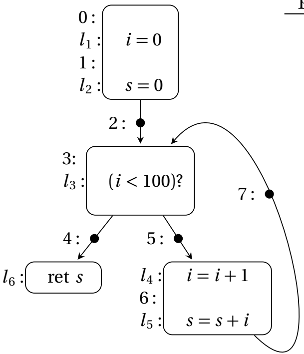


prog. point	$[i]$	$[s]$
0	$\top$	$\top$
1	$[0, 0]$	$\top$
2	$[0, 0]$	$[0, 0]$
3	$[0, 100]$	$[0, +\infty[$
4	$[100, 100]$	$[0, +\infty[$
5	$[0, 99]$	$[0, +\infty[$
6	$[0, 100]$	$[0, +\infty[$
7	$[0, 100]$	$[0, +\infty[$

Sparse Analysis

Sparse data-flow analysis: shortcut identity transfer functions by grouping contiguous program points bound to identities into larger regions

$i = 0;$
$s = 0;$
while $(i < 100)$
  $i = i + 1;$
  $s = s + i;$
ret

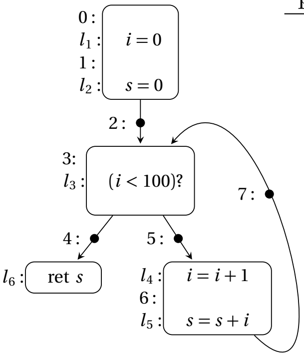


prog. point	$[i]$	$[s]$
0	$\top$	$\top$
1	$[0, 0]$	$\top$
2	$[0, 0]$	$[0, 0]$
3	$[0, 100]$	$[0, +\infty[$
4	$[100, 100]$	$[0, +\infty[$
5	$[0, 99]$	$[0, +\infty[$
6	$[0, 100]$	$[0, +\infty[$
7	$[0, 100]$	$[0, +\infty[$

Sparse Analysis

Sparse data-flow analysis: replace all $[v]^p$ by $[v]$ $(\forall v,\ p \in \mathrm{live}(v))$; propagate along def-use chains.

$i = 0;$
$s = 0;$
$\texttt{while}\ \ (i < 100)$
$\quad i = i + 1;$
$\quad s = s + i;$
$\texttt{ret}$



prog. point	$[i]$	$[s]$
0	$\top$	$\top$
1	$[0, 0]$	$\top$
2	$[0, 0]$	$[0, 0]$
3	$[0, 100]$	$[0, +\infty[$
4	$[100, 100]$	$[0, +\infty[$
5	$[0, 99]$	$[0, +\infty[$
6	$[0, 100]$	$[0, +\infty[$
7	$[0, 100]$	$[0, +\infty[$

Partitioned Variable Lattice Data-Flow Problems

Partitioned Variable Lattice (PVL) Problem

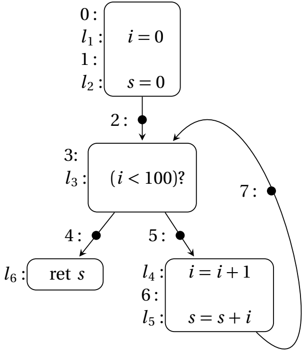
- program variables: v_i; program points: p; lattice: $\mathcal{L}$
- abstract state associated to prog. point p: x^p
- transfer function associated with $s \in$ *preds*(p): $F^{s,p}$
- constraint system: $x^p = x^p \wedge F^{s,p}(x^s)$ (or eq. $x^p \sqsubseteq F^{s,p}(x^s)$)

The corresponding Max. Fixed Point (MFP) problem is a PVL problem iff $\mathcal{L} = \mathcal{L}_{v_1} \times \cdots \times \mathcal{L}_{v_n}$ where each $\mathcal{L}_{v_i}$ is the lattice associated with v_i i.e. $x^s = ([v_1]^s, \ldots, [v_n]^s)$. Thus $F^{s,p} = F^{s,p}_{v_1} \times \cdots \times F^{s,p}_{v_n}$ and $[v_i]^p = [v_i]^p \wedge F^{s,p}_{v_i}([v_1]^s, \ldots, [v_n]^s)$.

Partitioned Variable Lattice Data-Flow Problem

Range analysis

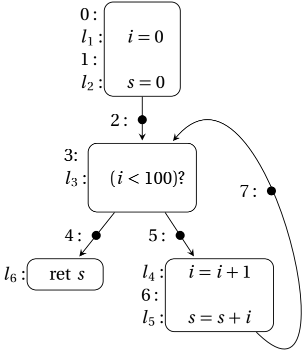
- $[i]^0 = [i]^0 \wedge F_i^{r,0}([i]^r, [s]^r)$
- $[i]^1 = [i]^1 \wedge F_i^{l_1}([i]^0, [s]^0)$
- $[i]^2 = [i]^2 \wedge F_i^{l_2}([i]^1, [s]^1)$
- $[i]^3 = [i]^3 \wedge F_i^{2,3}([i]^2, [s]^2)$
- $[i]^3 = [i]^3 \wedge F_i^{7,3}([i]^7, [s]^7)$
- $[i]^4 = [i]^4 \wedge F_i^{\overline{l_3}}([i]^3, [s]^3)$
- $[i]^5 = [i]^5 \wedge F_i^{l_3}([i]^3, [s]^3)$
- $[i]^6 = [i]^6 \wedge F_i^{l_4}([i]^5, [s]^5)$
- $[i]^7 = [i]^7 \wedge F_i^{l_5}([i]^6, [s]^6)$



Partitioned Variable Lattice Data-Flow Problem

Range analysis

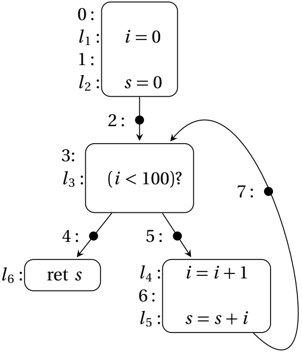
- $[i]^0 = [i]^0 \cup F_i^{r,0}([i]^r, [s]^r)$
- $[i]^1 = [i]^1 \cup F_i^{l_1}([i]^0, [s]^0)$
- $[i]^2 = [i]^2 \cup F_i^{l_2}([i]^1, [s]^1)$
- $[i]^3 = [i]^3 \cup F_i^{2,3}([i]^2, [s]^2)$
- $[i]^3 = [i]^3 \cup F_i^{7,3}([i]^7, [s]^7)$
- $[i]^4 = [i]^4 \cup F_i^{\overline{l_3}}([i]^3, [s]^3)$
- $[i]^5 = [i]^5 \cup F_i^{l_3}([i]^3, [s]^3)$
- $[i]^6 = [i]^6 \cup F_i^{l_4}([i]^5, [s]^5)$
- $[i]^7 = [i]^7 \cup F_i^{l_5}([i]^6, [s]^6)$



Partitioned Variable Lattice Data-Flow Problem

Range analysis

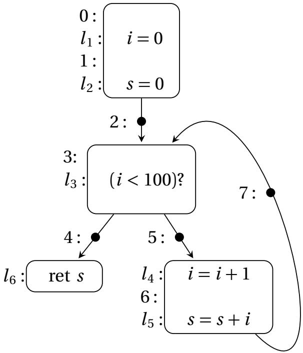
- $[i]^0 = [i]^0$
- $[i]^1 = [i]^1 \cup [0, 0]$
- $[i]^2 = [i]^2 \cup [i]^1$
- $[i]^3 = [i]^3 \cup [i]^2$
- $[i]^3 = [i]^3 \cup [i]^7$
- $[i]^4 = [i]^4 \cup \left([i]^3 \cap [100, +\infty[\right)$
- $[i]^5 = [i]^5 \cup \left([i]^3 \cap]-\infty, 99[\right)$
- $[i]^6 = [i]^6 \cup \left([i]^5 + 1\right)$
- $[i]^7 = [i]^7 \cup [i]^6$

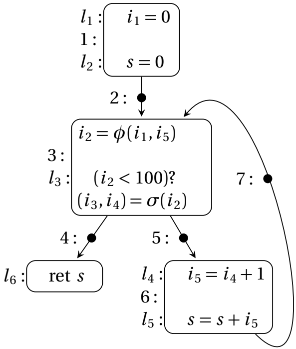


The Static Single Information Property

SSIfy (forward)

Modify the code (split live-ranges) without modifying its semantic s.t. fullfils SSI property

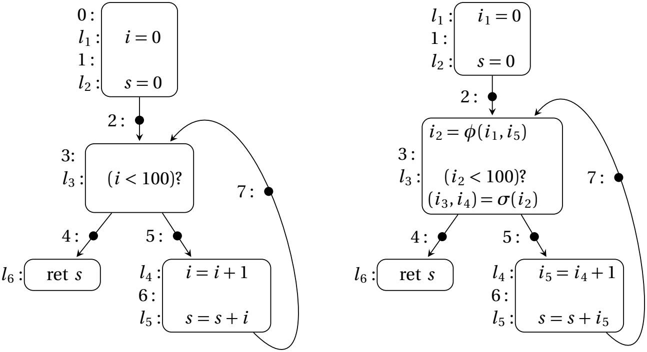




The Static Single Information Property

SPLIT

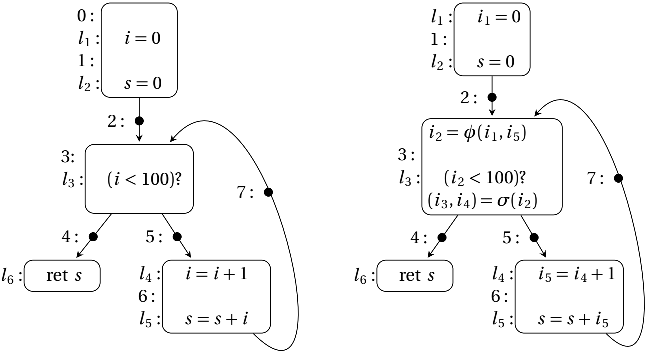
if s unique pred. of $p \in \mathrm{live}(v)$ and such that $F_v^{s,p} \neq \lambda x.\top$ is non-trivial, then s should contain a definition of v



The Static Single Information Property

SPLIT

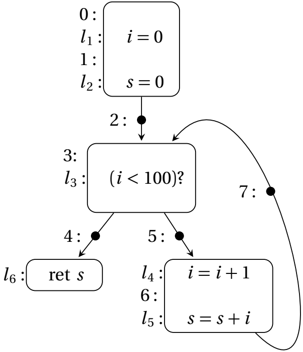
if s and t two preds of p such that $F_v^{s,p}(Y) \neq F_v^{t,p}(Y)$ (Y a MFP solution), then there must be a ϕ-function at entry of p

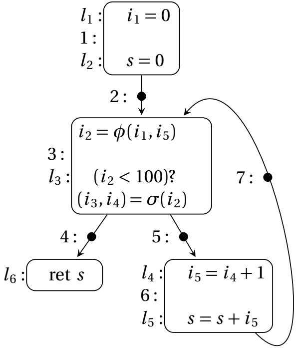


The Static Single Information Property

INFO

if $F_v^{s,p} \neq \lambda x.\top$,
then $v \in \text{live}(p)$

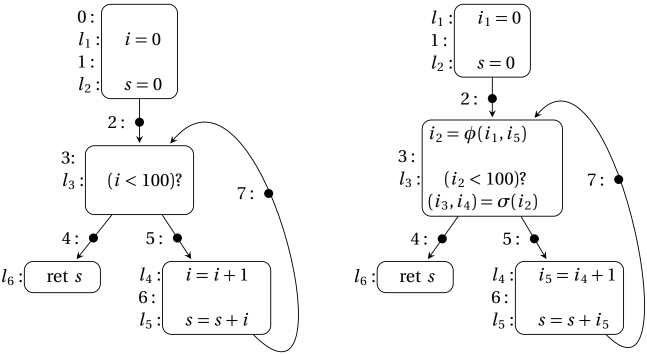




The Static Single Information Property

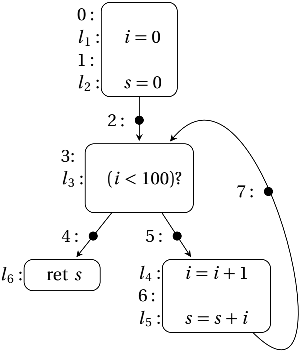
VERSION

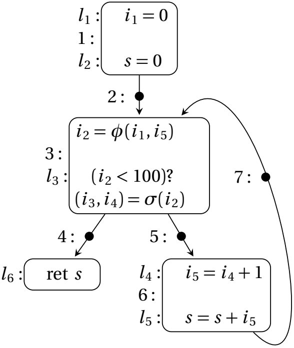
for each variable v, $\mathrm{live}(v)$ is a connected component of the CFG.



LINK

if F_v^{inst} depends on some $[u]^s$,
then *inst* should contain an use of u live-in at *inst*.

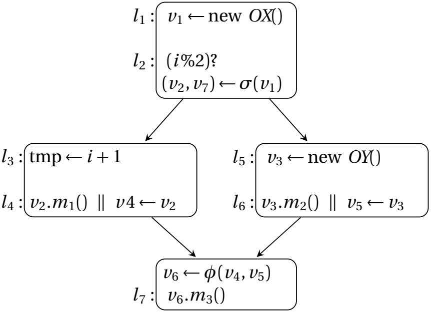




Special instructions used to split live ranges

Interior nodes (unique predecessor, unique successor)

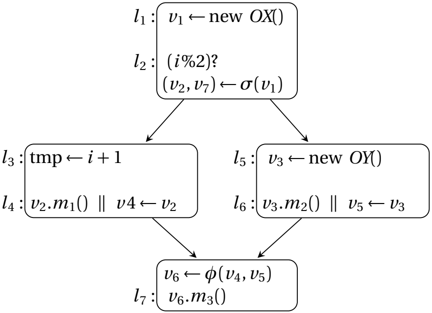
$$\textit{inst} \;\|\; v_1 = v_1' \;\|\; \ldots \;\|\; v_m = v_m'$$



Special instructions used to split live ranges

joins (multiple predecessors, one successor)

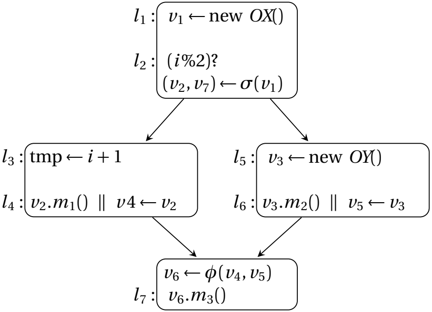
ϕ-functions



Special instructions used to split live ranges

branch points (one predecessor, mulitple successors)

$$(l^1 : v_1^1, \ldots, l^q : v_1^q) = \sigma(v_1) \;\; \| \;\; \ldots \;\; \| \;\; (l^1 : v_m^1, \ldots, l^q : v_m^q) = \sigma(v_m)$$



Propagating Information Forwardly and Backwardly

Dense constrained system

$[v]^p = [v]^p \wedge F_v^{s,p}([v_1]^s, \ldots, [v_n]^s)$

Sparse SSI constrained system

$[v] = [v] \wedge G_v^i([a], \ldots, [z])$ where $a, \ldots, z$ are used (resp. defined) at i

Proof

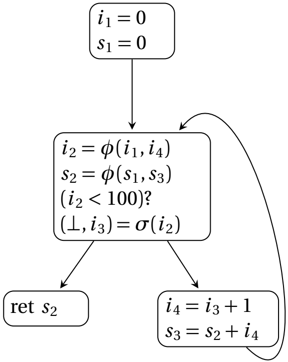
- coalesce all $[v]^p$ such that $v \in \mathrm{live}(p)$ into $[v]$; replace all $[v]^p$ such that $v \notin \mathrm{live}(p)$ by $\top$
- for each instruction *inst* with uses $a \ldots z$, let $G_v^i([a], \ldots, [z]) = F_v^i([v_1], \ldots, [v_n])$
- remove redundancies

Propagating Information Forwardly and Backwardly

Backward propagation engine under SSI

function back_propagate(transfer_functions $\mathcal{G}$)
    $\mathit{worklist} = \emptyset$
    foreach $v \in \mathrm{vars}$: $[v] = \top$
    foreach $i \in \mathrm{insts}$: $\mathit{worklist} \mathrel{+}= i$
    while $\mathit{worklist} \neq \emptyset$:
        let $i \in \mathit{worklist}$; $\mathit{worklist} \mathrel{-}= i$
        foreach $v \in i.\mathit{uses}()$:
            $[v]_{new} = [v] \wedge G_v^i([i.\mathit{defs}()])$
            if $[v] \neq [v]_{new}$:
                $\mathit{worklist} \mathrel{+}= v.\mathit{defs}()$
                $[v] = [v]_{new}$

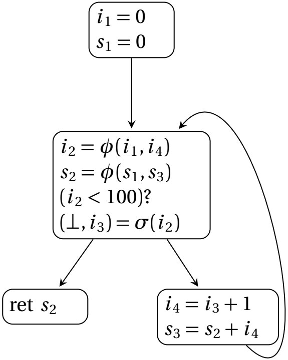
Examples of sparse data-flow analyses



Range analysis (forward from defs & conds)

$$
\begin{array}{lcl|l}
[i_1] & \cup= & [0,0] & \emptyset \\
[s_1] & \cup= & [0,0] & \emptyset \\
[i_2] & \cup= & [i_1] \cup [i_4] & \emptyset \\
[s_2] & \cup= & [s_1] \cup [s_3] & \emptyset \\
[i_3] & \cup= & ([i_2] \cap]-\infty, 99]) & \emptyset \\
[i_4] & \cup= & ([i_3] + 1) & \emptyset \\
[s_3] & \cup= & ([s_2] + [i_4]) & \emptyset
\end{array}
$$

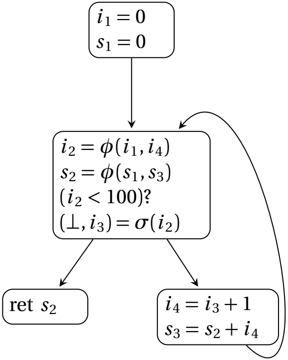
Examples of sparse data-flow analyses



Range analysis (forward from defs & conds)

$$
\begin{array}{l|l}
\color{red}{[i_1] \cup= [0,0]} & \emptyset \\
[s_1] \cup= [0,0] & \emptyset \\
[i_2] \cup= [i_1] \cup [i_4] & \emptyset \\
[s_2] \cup= [s_1] \cup [s_3] & \emptyset \\
[i_3] \cup= ([i_2] \cap]-\infty, 99]) & \emptyset \\
[i_4] \cup= ([i_3] + 1) & \emptyset \\
[s_3] \cup= ([s_2] + [i_4]) & \emptyset
\end{array}
$$

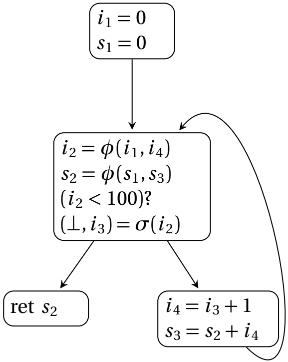
Examples of sparse data-flow analyses



Range analysis (forward from defs & conds)

$$
\begin{array}{l|l}
[i_1] \cup= [0,0] & [0,0] \\
[s_1] \cup= [0,0] & \emptyset \\
[i_2] \cup= [i_1] \cup [i_4] & \emptyset \\
[s_2] \cup= [s_1] \cup [s_3] & \emptyset \\
[i_3] \cup= ([i_2] \cap]-\infty, 99]) & \emptyset \\
[i_4] \cup= ([i_3] + 1) & \emptyset \\
[s_3] \cup= ([s_2] + [i_4]) & \emptyset
\end{array}
$$

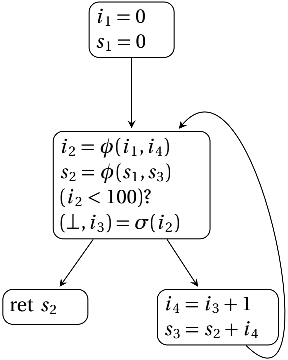
Examples of sparse data-flow analyses



Range analysis (forward from defs & conds)

$[i_1] \cup= [0,0]$	$[0,0]$
$[s_1] \cup= [0,0]$	$\emptyset$
$[i_2] \cup= [i_1] \cup [i_4]$	$\emptyset$
$[s_2] \cup= [s_1] \cup [s_3]$	$\emptyset$
$[i_3] \cup= ([i_2] \cap]-\infty, 99])$	$\emptyset$
$[i_4] \cup= ([i_3] + 1)$	$\emptyset$
$[s_3] \cup= ([s_2] + [i_4])$	$\emptyset$

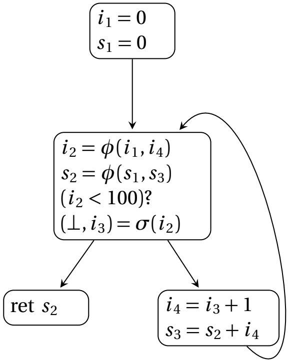
Examples of sparse data-flow analyses



Range analysis (forward from defs & conds)

$[i_1] \cup= [0,0]$	$[0,0]$
$[s_1] \cup= [0,0]$	$[0,0]$
$[i_2] \cup= [i_1] \cup [i_4]$	$\emptyset$
$[s_2] \cup= [s_1] \cup [s_3]$	$\emptyset$
$[i_3] \cup= ([i_2] \cap]-\infty, 99])$	$\emptyset$
$[i_4] \cup= ([i_3] + 1)$	$\emptyset$
$[s_3] \cup= ([s_2] + [i_4])$	$\emptyset$

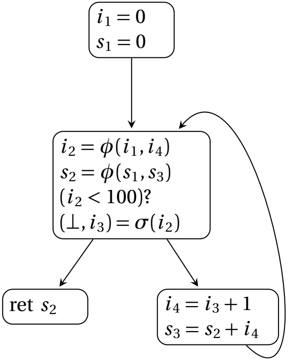
Examples of sparse data-flow analyses



Range analysis (forward from defs & conds)

$[i_1] \cup= [0,0]$	$[0,0]$
$[s_1] \cup= [0,0]$	$[0,0]$
$[i_2] \cup= [i_1] \cup [i_4]$	$\emptyset$
$[s_2] \cup= [s_1] \cup [s_3]$	$\emptyset$
$[i_3] \cup= ([i_2] \cap]-\infty, 99])$	$\emptyset$
$[i_4] \cup= ([i_3] + 1)$	$\emptyset$
$[s_3] \cup= ([s_2] + [i_4])$	$\emptyset$

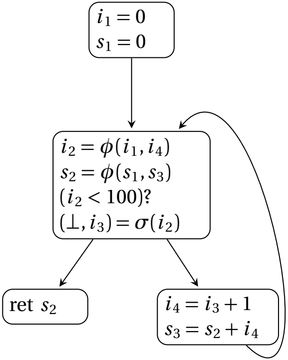
Examples of sparse data-flow analyses



Range analysis (forward from defs & conds)

$$
\begin{array}{lll|l}
[i_1] & \cup= & [0,0] & [0,0] \\
[s_1] & \cup= & [0,0] & [0,0] \\
[i_2] & \cup= & [i_1] \cup [i_4] & [0,0] \\
[s_2] & \cup= & [s_1] \cup [s_3] & \emptyset \\
[i_3] & \cup= & ([i_2] \cap]-\infty, 99]) & \emptyset \\
[i_4] & \cup= & ([i_3] + 1) & \emptyset \\
[s_3] & \cup= & ([s_2] + [i_4]) & \emptyset
\end{array}
$$

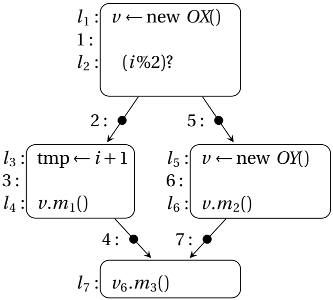
Examples of sparse data-flow analyses



Range analysis (forward from defs & conds)

$$
\begin{array}{l|l}
[i_1] \cup= [0,0] & [0,0] \\
[s_1] \cup= [0,0] & [0,0] \\
[i_2] \cup= [i_1] \cup [i_4] & [0,100] \\
[s_2] \cup= [s_1] \cup [s_3] & [0,+\infty[\\
[i_3] \cup= ([i_2] \cap]-\infty, 99]) & [0,99] \\
[i_4] \cup= ([i_3] + 1) & [1,100] \\
[s_3] \cup= ([s_2] + [i_4]) & [1,+\infty[
\end{array}
$$

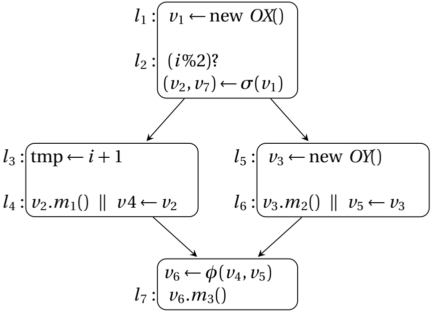
Examples of sparse data-flow analyses



Class inference (backward from uses)

prog. point	$[v]$
1	$\{m_1, m_3\}$
2	$\{m_1, m_3\}$
3	$\{m_1, m_3\}$
4	$\{m_3\}$
5	$\top$
6	$\{m_2, m_3\}$
7	$\{m_3\}$

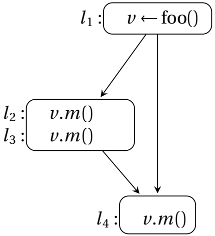
Examples of sparse data-flow analyses



Class inference (backward from uses)

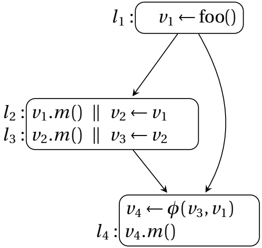
$[v_7]$	$\top$
$[v_6] \cup= \{m_3\}$	$\{m_3\}$
$[v_5] \cup= [v_6]$	$\{m_3\}$
$[v_4] \cup= [v_6]$	$\{m_3\}$
$[v_2] \cup= (\{m_1\} \wedge [v_4])$	$\{m_1, m_3\}$
$[v_3] \cup= (\{m_2\} \wedge [v_4])$	$\{m_2, m_3\}$
$[v_1] \cup= [v_2]$	$\{m_1, m_3\}$
$[v_1] \cup= [v_7]$	$\{m_1, m_3\}$

Examples of sparse data-flow analyses



Null pointer (forward from defs & uses)

Examples of sparse data-flow analyses



Null pointer (forward from defs & uses)

$$\begin{array}{lll|l}
[v_1] & \wedge= & 0 & 0 \\
[v_2] & \wedge= & \cancel{0} & \cancel{0} \\
[v_3] & \wedge= & \cancel{0} & \cancel{0} \\
[v_4] & \wedge= & ([v_3] \wedge [v_1]) & 0
\end{array}$$

Splitting strategy

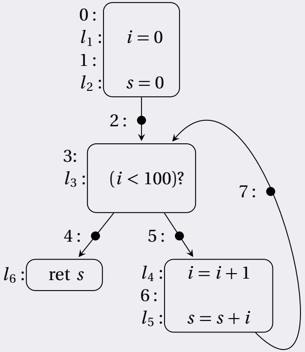
Live range splitting strategy $\mathcal{P}_v = I_\uparrow \cup I_\downarrow$

$I_\downarrow$: set of points i with forward direction
$I_\uparrow$: set of points i with backward direction

```
function SSIfy(var v, Splitting_Strategy P_v)
    split(v, P_v)
    rename(v)
    clean(v)
```

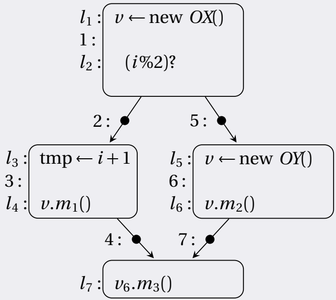
Splitting strategy

Range analysis: $\mathcal{P}_i = \{l_1, \mathrm{Out}(l_3), l_4\}_\downarrow$



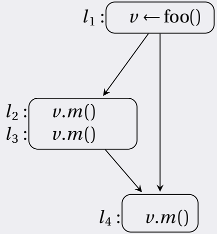
Splitting strategy

Class inference: $\mathcal{P}_v = \{l_4, l_6, l_7\}_\uparrow$



Splitting strategy

Null pointer: $\mathcal{P}_v = \{l_1, l_2, l_3, l_4\}_\downarrow$



Splitting strategy

Client	Splitting strategy $\mathcal{P}$
Alias analysis, reaching definitions cond. constant propagation	$\mathit{Defs}_{\downarrow}$
Partial Redundancy Elimination	$\mathit{Defs}_{\downarrow} \bigcup \mathit{LastUses}_{\uparrow}$
ABCD, taint analysis, range analysis	$\mathit{Defs}_{\downarrow} \bigcup \mathcal{Out}(\mathit{Conds})_{\downarrow}$
Stephenson's bitwidth analysis	$\mathit{Defs}_{\downarrow} \bigcup \mathcal{Out}(\mathit{Conds})_{\downarrow} \bigcup \mathit{Uses}_{\uparrow}$
Mahlke's bitwidth analysis	$\mathit{Defs}_{\downarrow} \bigcup \mathit{Uses}_{\uparrow}$
An's type inference, Class inference	$\mathit{Uses}_{\uparrow}$
Hochstadt's type inference	$\mathit{Uses}_{\uparrow} \bigcup \mathcal{Out}(\mathit{Conds})_{\uparrow}$
Null-pointer analysis	$\mathit{Defs}_{\downarrow} \bigcup \mathit{Uses}_{\downarrow}$

Splitting live ranges

- Split live range of v at each $p \in \mathcal{P}_v$
- Split live range where the information collide (join set $\mathcal{J}(I_\downarrow)$ and split set $\mathcal{S}(I_\uparrow)$)
- Iterated dominance frontier $\mathrm{DF}^+(S) = \mathcal{J}(S \cup \{r\})$ can be computed efficiently (as opposed to $\mathcal{J}(S)$)
- Iterated post dominance frontier $\mathrm{pDF}^+(S) = \mathcal{J}(S \cup \{r\})$ for the reverse CFG

function split(var *v*, Splitting_Strategy $\mathcal{P}_v = I_\downarrow \cup I_\uparrow$)

$$\left[I_\downarrow \cup \mathrm{In}(\mathrm{DF}^+(I_\downarrow))\right]$$

Splitting live ranges

- Split live range of v at each $p \in \mathcal{P}_v$
- Split live range where the information collide (join set $\mathcal{J}(I_\downarrow)$ and split set $\mathcal{S}(I_\uparrow)$)
- Iterated dominance frontier $\mathrm{DF}^+(S) = \mathcal{J}(S \cup \{r\})$ can be computed efficiently (as opposed to $\mathcal{J}(S)$)
- Iterated post dominance frontier $\mathrm{pDF}^+(S) = \mathcal{J}(S \cup \{r\})$ for the reverse CFG

function split(var *v*, Splitting_Strategy $\mathcal{P}_v = I_\downarrow \cup I_\uparrow$)

$$\left[I_\downarrow \cup \mathrm{In}(\mathrm{DF}^+(I_\downarrow))\right] \cup \left[I_\uparrow \cup \mathrm{Out}(\mathrm{pDF}^+(I_\uparrow))\right]$$

Splitting live ranges

- Split live range of v at each $p \in \mathcal{P}_v$
- Split live range where the information collide (join set $\mathcal{J}(I_\downarrow)$ and split set $\mathcal{S}(I_\uparrow)$)
- Iterated dominance frontier $\mathrm{DF}^+(S) = \mathcal{J}(S \cup \{r\})$ can be computed efficiently (as opposed to $\mathcal{J}(S)$)
- Iterated post dominance frontier $\mathrm{pDF}^+(S) = \mathcal{J}(S \cup \{r\})$ for the reverse CFG

function split(var *v*, Splitting_Strategy $\mathcal{P}_v = I_\downarrow \cup I_\uparrow$)

$$\mathcal{P}_v \cup \mathrm{In}\left[\mathrm{DF}^+(I_\downarrow \cup \left[I_\uparrow \cup \mathrm{Out}(\mathrm{pDF}^+(I_\uparrow))\right])\right]$$

Variable Renaming

function rename(var v)

- traverses the CFG along topological order
- give a unique version to each definition of v
- stack the versions that dominates the current program point
- rename each use of v with the version of immediately dominating definition

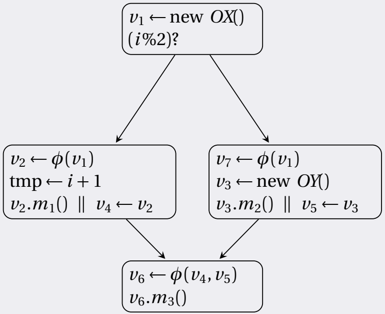
Dead and Undefined Code Elimination

clean(var v)

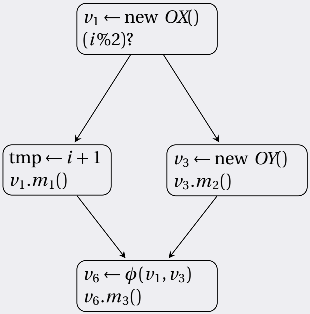
- actual instructions: instructions originally in the code
- SSA graph: nodes are instructions; edges are def-use chains
- active instructions: instructions connected to an actual instruction
- simple traversal of the SSA graph from actual instructions that mark active ones
- remove non-active instructions (inserted phi and sigma functions)

Implementation Details

Implementing σ-functions



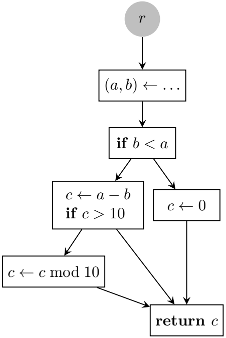
SSI destruction



Control-flow graph (CFG)

Basic blocks sequence of consecutive statements

Edges control flow (jumps or fall-through)



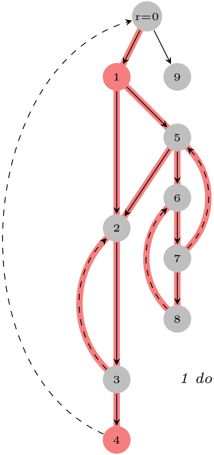
$(a, b) \leftarrow \ldots$
if $b < a$ **then**
  $c \leftarrow a - b$
  if $c > 10$ **then**
    $c \leftarrow c \bmod 10$
  endif
else
  $c \leftarrow 0$
endif
return c

Back

Tree-shape. Dominance

Dominance relation

- a single entry node r.
- each node reachable from r.
- a dominates b if every path from r to b contains a.

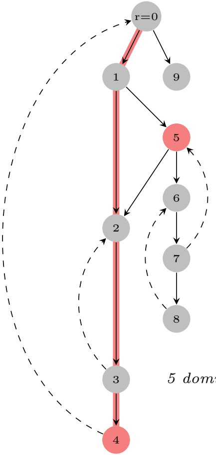


1 dominates 4? YES

Tree-shape. Dominance

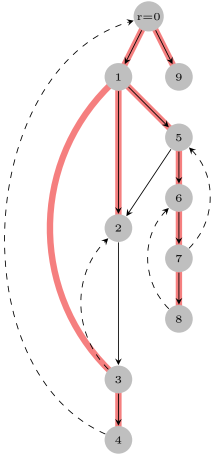
Dominance relation

- a single entry node r.
- each node reachable from r.
- a dominates b if every path from r to b contains a.



5 dominates 4? NO

Tree-shape. Dominance



Dominance relation

- a single entry node r.
- each node reachable from r.
- a dominates b if every path from r to b contains a.

Properties

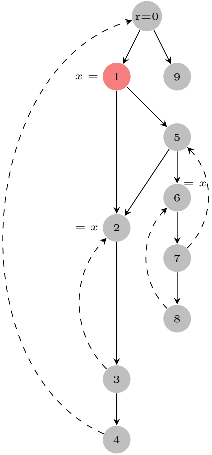
- The dominance relation induces a tree.

Strict code

Every path from r to a use traverses a definition

Strict SSA

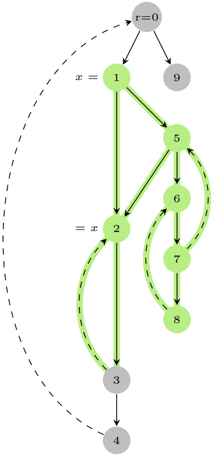
- SSA: only one definition textually per variable
- Strict: the definition dominates all uses



Liveness: sub-tree of a tree

The live-range of an SSA variable is

the set of program points
between the definition and a use
(without going through the definition again)

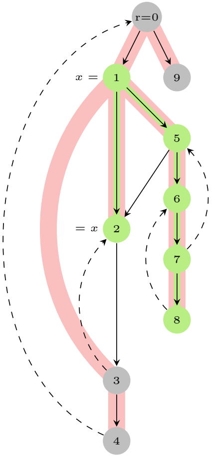


Liveness: sub-tree of a tree

The live-range of an SSA variable is

the set of program points
between the definition and a use
(without going through the definition again)

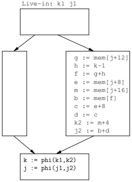
- the definition dominates the entire live-range
- the live-range is a sub-tree of the dominance-tree



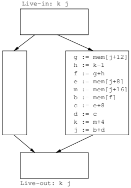
Basic block

```
Live-in: 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+d
Live-out: k' j'
```

SSA code



General CFG

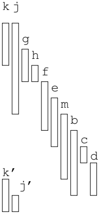


- MAXLIVE $\leq r$
- Linear scan

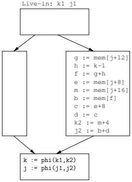
“Spilling easier on a BB than on a general CFG”

Basic block

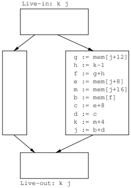
```
Live-in: k j
g  := mem[j+12]
h  := k-1
f  := g+h
e  := mem[j+8]
m  := mem[j+16]
b  := mem[f]
c  := e+8
d  := c
k'  := m+4
j'  := b+d
Live-out: k' j'
```



SSA code



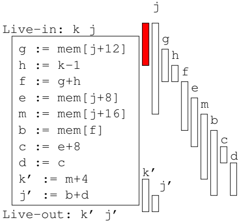
General CFG



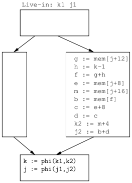
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

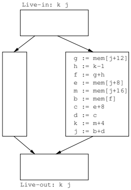
Basic block



SSA code

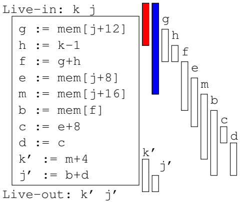


General CFG

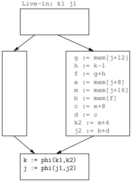


- MAXLIVE $\leq r$
- Linear scan

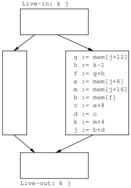
Basic block



SSA code



General CFG

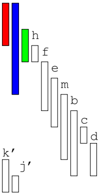


- MAXLIVE $\leq r$
- Linear scan

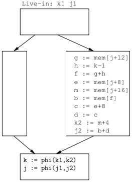
“Spilling easier on a BB than on a general CFG”

Basic block

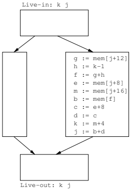
```
Live-in: k j
g  := mem[j+12]
h  := k-1
f  := g+h
e  := mem[j+8]
m  := mem[j+16]
b  := mem[f]
c  := e+8
d  := c
k'  := m+4
j'  := b+d
Live-out: k' j'
```



SSA code



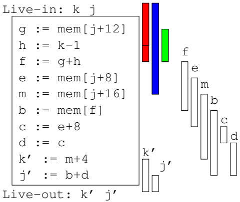
General CFG



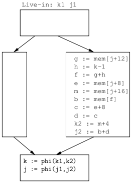
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

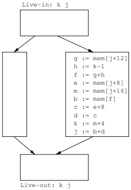
Basic block



SSA code



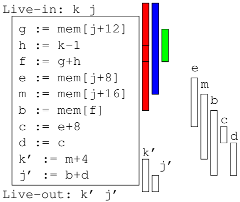
General CFG



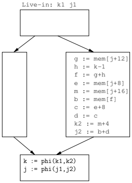
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

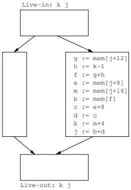
Basic block



SSA code



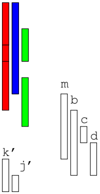
General CFG



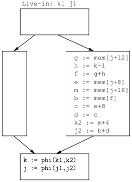
- MAXLIVE $\leq r$
- Linear scan

Basic block

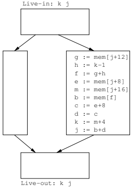
```
Live-in: k j
g  := mem[j+12]
h  := k-1
f  := g+h
e  := mem[j+8]
m  := mem[j+16]
b  := mem[f]
c  := e+8
d  := c
k'  := m+4
j'  := b+d
Live-out: k' j'
```



SSA code



General CFG

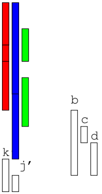


- MAXLIVE $\leq r$
- Linear scan

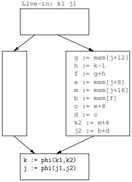
“Spilling easier on a BB than on a general CFG”

Basic block

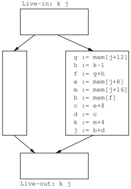
```
Live-in: k j
  g  := mem[j+12]
  h  := k-1
  f  := g+h
  e  := mem[j+8]
  m  := mem[j+16]
  b  := mem[f]
  c  := e+8
  d  := c
  k'  := m+4
  j'  := b+d
Live-out: k' j'
```



SSA code



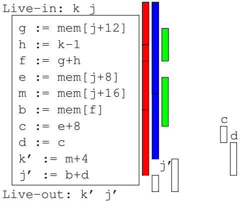
General CFG



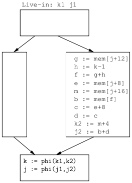
- MAXLIVE $\leq r$
- Linear scan

"Spilling easier on a BB than on a general CFG"

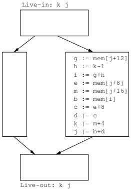
Basic block



SSA code



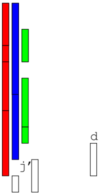
General CFG



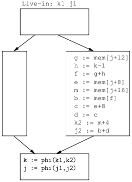
- MAXLIVE $\leq r$
- Linear scan

Basic block

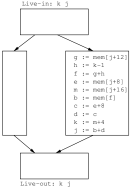
```
Live-in: k j
g  := mem[j+12]
h  := k-1
f  := g+h
e  := mem[j+8]
m  := mem[j+16]
b  := mem[f]
c  := e+8
d  := c
k'  := m+4
j'  := b+d
Live-out: k' j'
```



SSA code

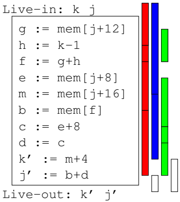


General CFG

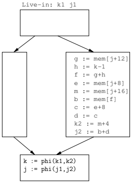


- MAXLIVE $\leq r$
- Linear scan

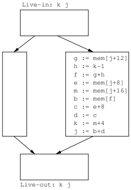
Basic block



SSA code



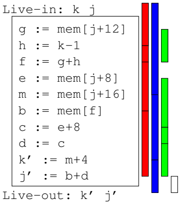
General CFG



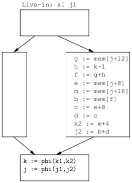
- MAXLIVE $\leq r$
- Linear scan

"Spilling easier on a BB than on a general CFG"

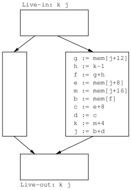
Basic block



SSA code



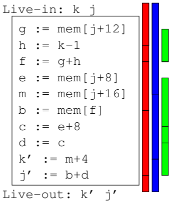
General CFG



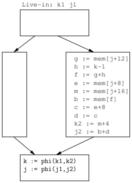
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

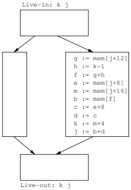
Basic block



SSA code



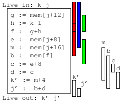
General CFG



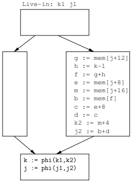
- MAXLIVE $\leq r$
- Linear scan

“Spilling easier on a BB than on a general CFG”

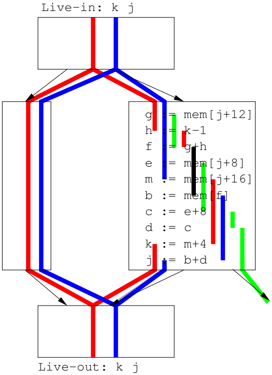
BB



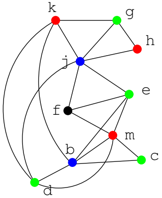
SSA



General control flow graph



Interference graph

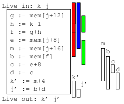


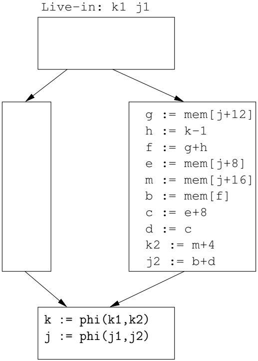
- Coloring test
- Greedy coloring

"Under SSA: the dominance tree"

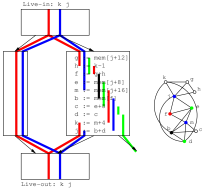
Static single assignment form

BB





CFG

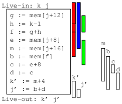


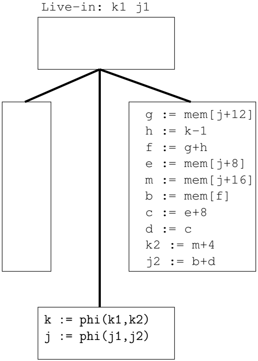
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

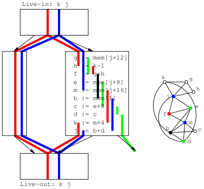
Static single assignment form

BB





CFG

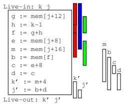


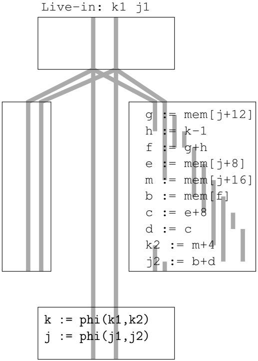
- MAXLIVE $\leq r$
- Tree scan

"Under SSA: the dominance tree"

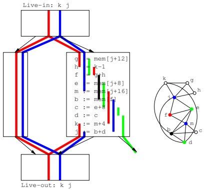
Static single assignment form

BB





CFG

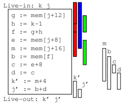


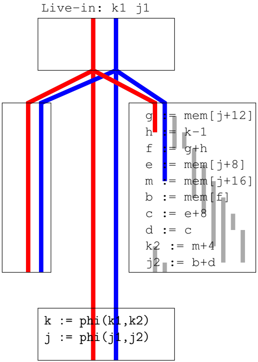
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

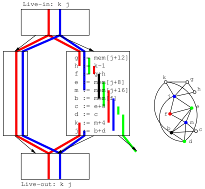
Static single assignment form

BB





CFG

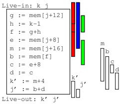


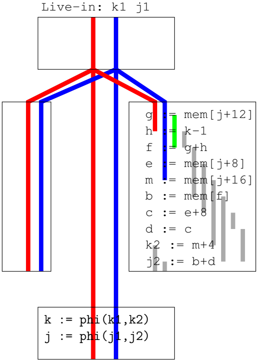
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

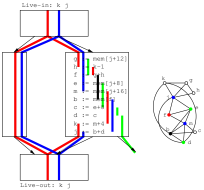
Static single assignment form

BB





CFG

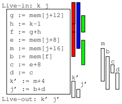


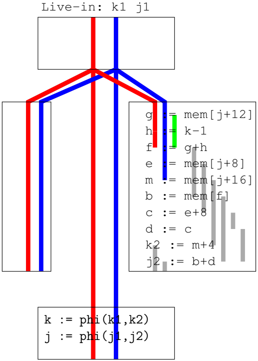
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

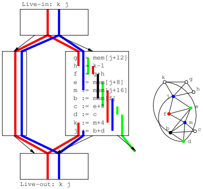
Static single assignment form

BB





CFG

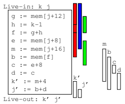


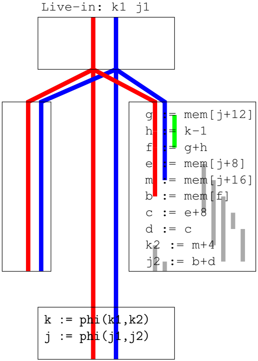
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

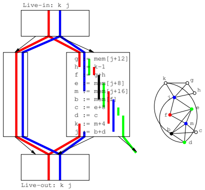
Static single assignment form

BB





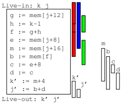
CFG

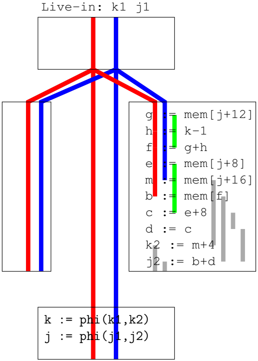


- MAXLIVE $\leq r$
- Tree scan

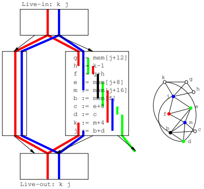
Static single assignment form

BB





CFG

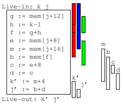


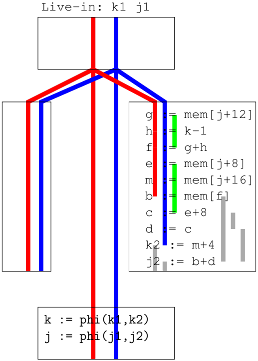
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

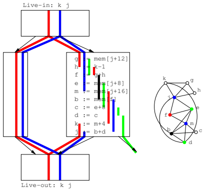
Static single assignment form

BB





CFG

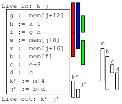


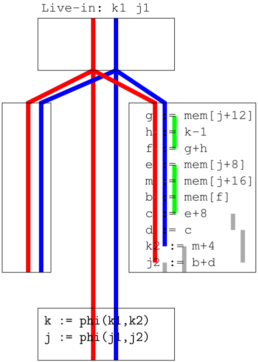
- MAXLIVE $\leq r$
- Tree scan

"Under SSA: the dominance tree"

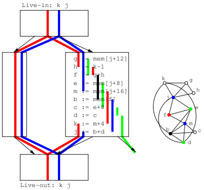
Static single assignment form

BB





CFG

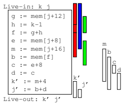


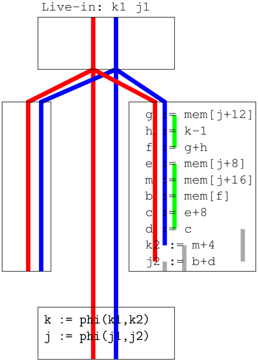
- MAXLIVE $\leq r$
- Tree scan

"Under SSA: the dominance tree"

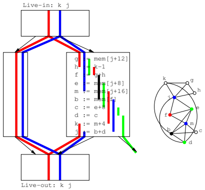
Static single assignment form

BB





CFG

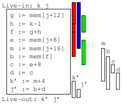


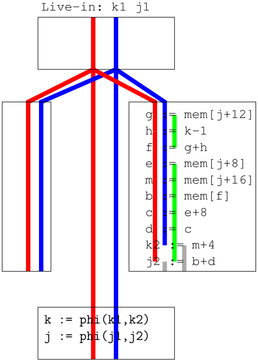
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

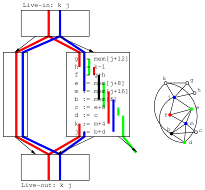
Static single assignment form

BB





CFG

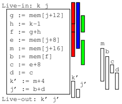


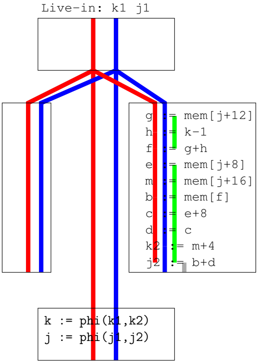
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

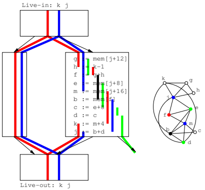
Static single assignment form

BB





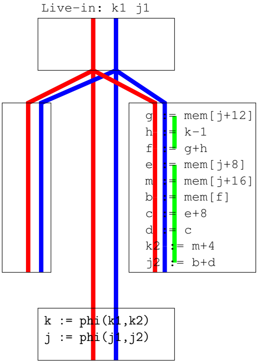
CFG



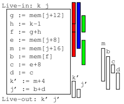
- MAXLIVE $\leq r$
- Tree scan

“Under SSA: the dominance tree”

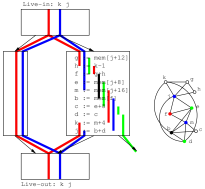
Static single assignment form



BB



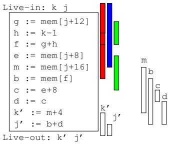
CFG



- MAXLIVE $\leq r$
- Tree scan

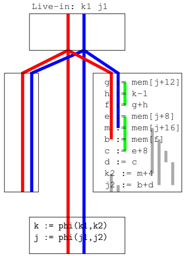
“Under SSA: the dominance tree”

Basic block



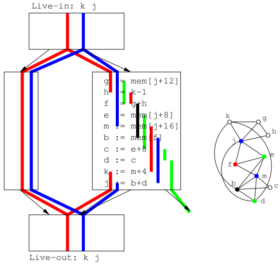
- MAXLIVE $\leq r$
- Linear scan

SSA form



- MAXLIVE $\leq r$
- Tree scan

General CFG



- Coloring test
- Greedy coloring