CS 516: COMPILERS

Lecture 23

Topics

- Register Allocation

Remaining

- HW9: Dataflow Analysis
  - Due Thursday, April 27th at 11:59pm
  - Alias analysis, dead code elimination
- HW10: Register Allocation
  - Due Thursday, May 4th at 11:59pm
  - Constant propagation, register allocation, experiments
- Lectures:
  - L22: Generalized Analysis (some today, some next week)
  - L23: Register Allocation (today)
  - L24: Wrap up (next week)
- Final Quiz, next week
  - Lec20: Optimizations
  - Lec21: Analysis
  - Lec22: Generalized Analysis

Today, April 25

- Remaining Aspects of the course
- Register Allocation

REGISTER ALLOCATION

Register Allocation Problem

- Given: an IR program that uses an unbounded number of temporaries
  - e.g. the uids of our LLVM programs
- Find: a mapping from temporaries to machine registers such that
  - program semantics is preserved (i.e. the behavior is the same)
  - register usage is maximized
  - moves between registers are minimized
  - calling conventions / architecture requirements are obeyed
- Stack Spilling
  - If there are k registers available and m > k temporaries are live at the same time, then not all of them will fit into registers.
  - So: "spill" the excess temporaries to the stack.

Linear-Scan Register Allocation

Simple, greedy register-allocation strategy:

Node: `%x = OP y z`

1. Compute liveness information: `live(x)`
   - recall: `live(x)` is the set of uids that are live on entry to `x`'s definition
2. Let `pal` be the set of usable registers
   - usually reserve a couple for spill code [our implementation uses rax,rcx]
3. Maintain "layout" `uid_loc` that maps uids to locations
   - locations include registers and stack slots n, starting at n=0
4. Scan through the program. For each instruction that defines a uid `x`
   - `used` = `{r | reg r = uid_loc(y)` s.t. `y` ∈ `live(x)}`
   - `available = pal - used`
   - If `available` is empty: *// no registers available, spill*
     `uid_loc(x) := slot n` ; n = n + 1
   - Otherwise, pick `r` in `available`: *// choose an available register*
     `uid_loc(x) := reg r`

For HW10

- HW 10 implements two naive register allocation strategies:
- `no_reg_layout`: spill all registers

- `simple_layout`: use registers but without taking liveness into account

- Your job: do "better" than these.
- Quality Metric:
  - registers other than rbp count positively
  - rbp counts negatively (it is used for spilling)
  - shorter code is better (each line counts as 2 registers)

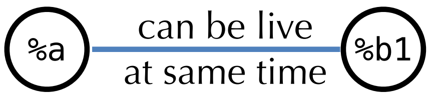
- Linear scan register allocation should suffice
  - but… can we do better?

GRAPH COLORING

Register Allocation

Basic process:

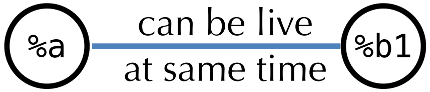
1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   - Nodes are temporary variables.
   - There is an edge between node n and m if n is *live at the same time as* m



Register Allocation

Basic process:

1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   - Nodes are temporary variables.
   - There is an edge between node n and m if n is *live at the same time as* m

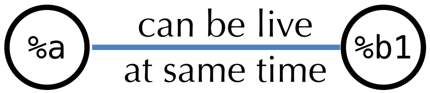


3. Try to color the graph

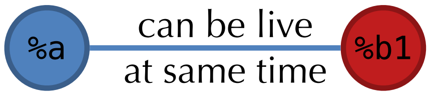
Register Allocation

Basic process:

1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   - Nodes are temporary variables.
   - There is an edge between node n and m if n is *live at the same time as* m



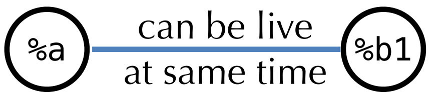
3. Try to color the graph



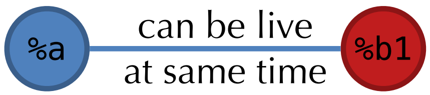
Register Allocation

Basic process:

1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   – Nodes are temporary variables.
   – There is an edge between node n and m if n is *live at the same time as* m



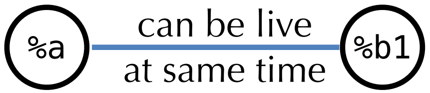
3. Try to color the graph
   – Each color corresponds to a register



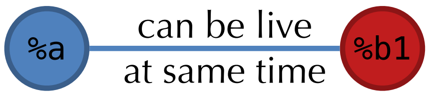
Register Allocation

Basic process:

1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   - Nodes are temporary variables.
   - There is an edge between node n and m if n is *live at the same time as* m



3. Try to color the graph
   - Each color corresponds to a register

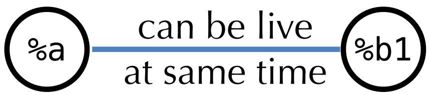


4. In case step 3 fails, "spill" a register to the stack and repeat the whole process.

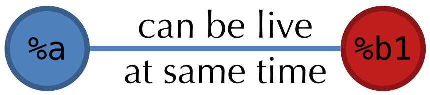
Register Allocation

Basic process:

1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   - Nodes are temporary variables.
   - There is an edge between node n and m if n is *live at the same time as* m



3. Try to color the graph
   - Each color corresponds to a register



4. In case step 3 fails, "spill" a register to the stack and repeat the whole process.
5. Rewrite the program to use registers

Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their *live ranges intersect* (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

```
%b1 = add i32 %a, 2

%c = mult i32 %b1, %b1

%b2 = add i32 %c, 1

%ans = mult i32 %b2, %a

return %ans;
```

Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their *live ranges intersect* (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

What temps are live?

```
%b1 = add i32 %a, 2

%c = mult i32 %b1, %b1

%b2 = add i32 %c, 1

%ans = mult i32 %b2, %a

return %ans;
```

Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their *live ranges intersect* (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

What temps are live?

```
// live = {%a}
%b1 = add i32 %a, 2

%c = mult i32 %b1, %b1

%b2 = add i32 %c, 1

%ans = mult i32 %b2, %a

return %ans;
```

Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their *live ranges intersect* (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

What temps are live?

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1

%b2 = add i32 %c, 1

%ans = mult i32 %b2, %a

return %ans;
```

Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their *live ranges intersect* (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

What temps are live?

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1

%ans = mult i32 %b2, %a

return %ans;
```

Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their *live ranges intersect* (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

What temps are live?

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1
// live = {%a,%b2}
%ans = mult i32 %b2, %a

return %ans;
```

Interference Graphs

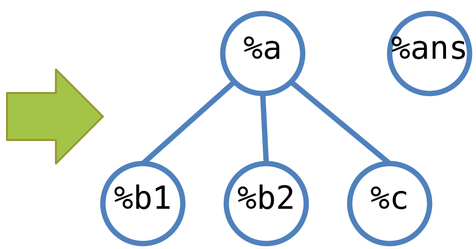
- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their *live ranges intersect* (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1
// live = {%a,%b2}
%ans = mult i32 %b2, %a
// live = {%ans}
return %ans;
```

Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their *live ranges intersect* (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1
// live = {%a,%b2}
%ans = mult i32 %b2, %a
// live = {%ans}
return %ans;
```

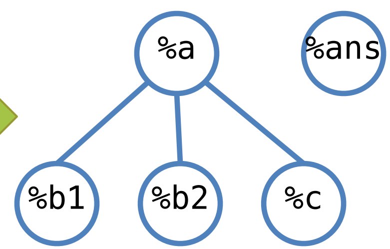


Interference Graph

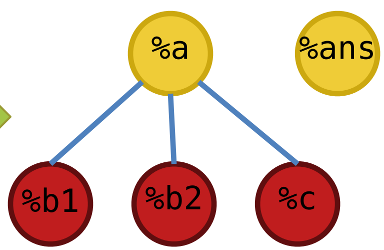
Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their *live ranges intersect* (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1
// live = {%a,%b2}
%ans = mult i32 %b2, %a
// live = {%ans}
return %ans;
```



Interference Graph



2-Coloring of the graph
red = r8
yellow = r9

Register Allocation Questions

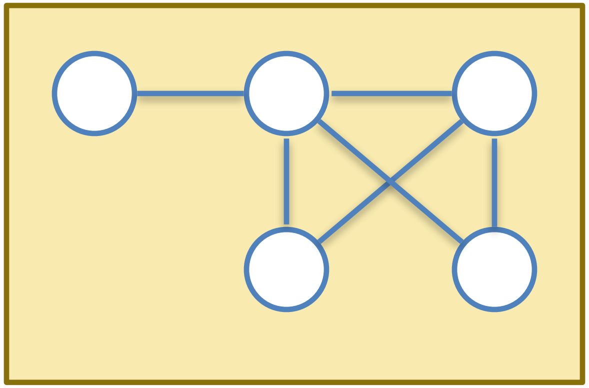
- Can we efficiently find a k-coloring of the graph whenever possible?
  - Answer: in general the problem is NP-complete (it requires search)
  - But, we can do an efficient approximation using heuristics.
- How do we assign registers to colors?
  - If we do this in a smart way, we can eliminate redundant MOV instructions.
- What do we do when there aren't enough colors/registers?
  - We have to use stack space, but how do we do this effectively?

Coloring a Graph: Kempe's Algorithm

- Kempe [1879] provides this algorithm for K-coloring a graph.

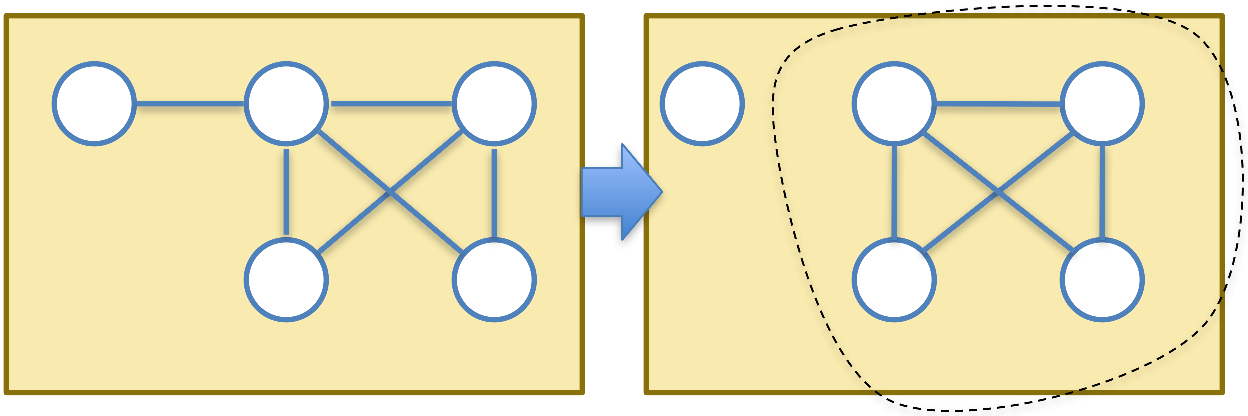
- It's a recursive algorithm that works in three steps:
  - **Step 1**: Find a node with degree < K and cut it out of the graph.
    - Remove the nodes and edges.
    - This is called "*simplifying*" the graph
  - **Step 2**: Recursively K-color the remaining subgraph
  - **Step 3**: When remaining graph is colored, there must be at least one free color available for the deleted node (since its degree was < K). Pick such a color.

Example: 3-color this Graph



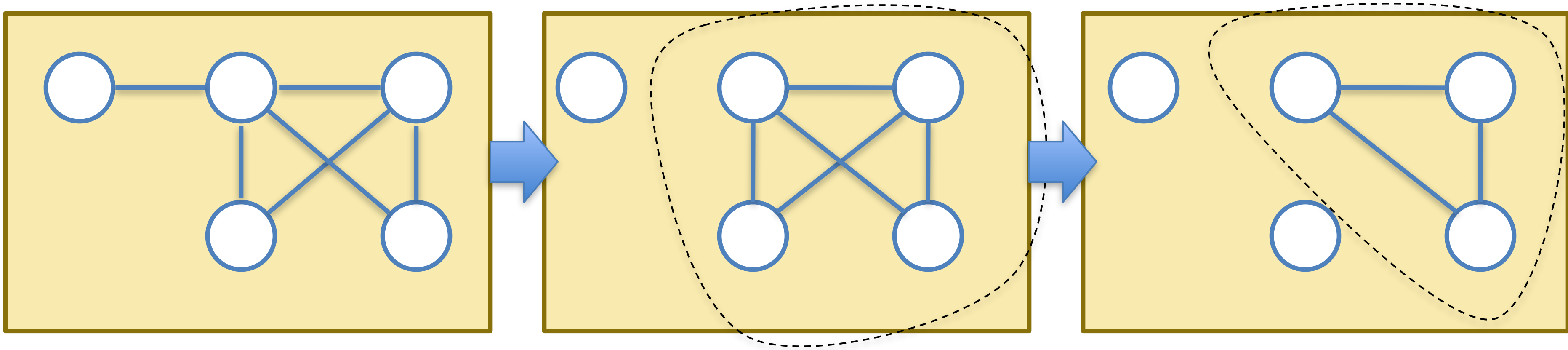
Recursing Down the Simplified Graphs

Example: 3-color this Graph



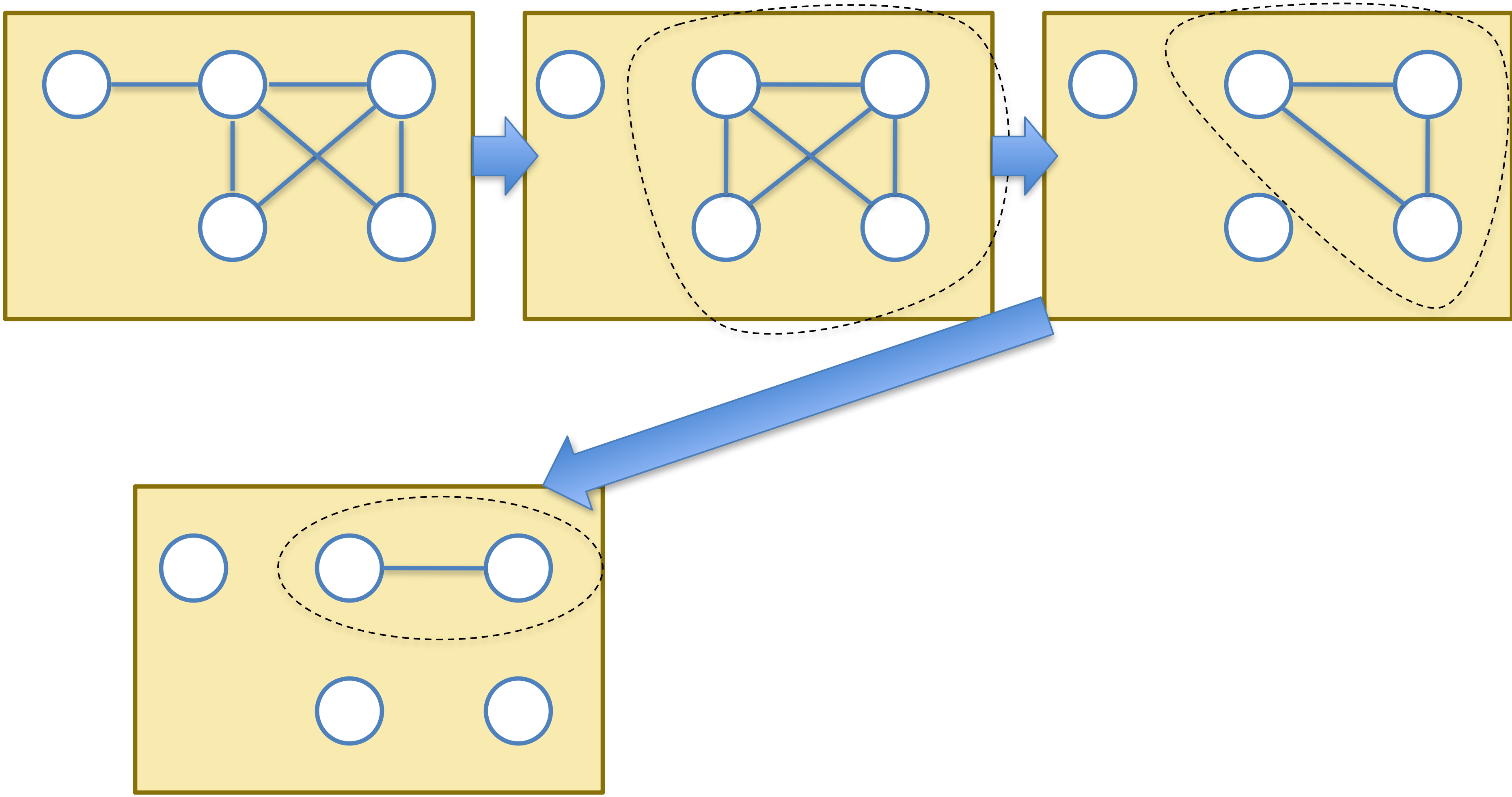
Recursing Down the Simplified Graphs

Example: 3-color this Graph



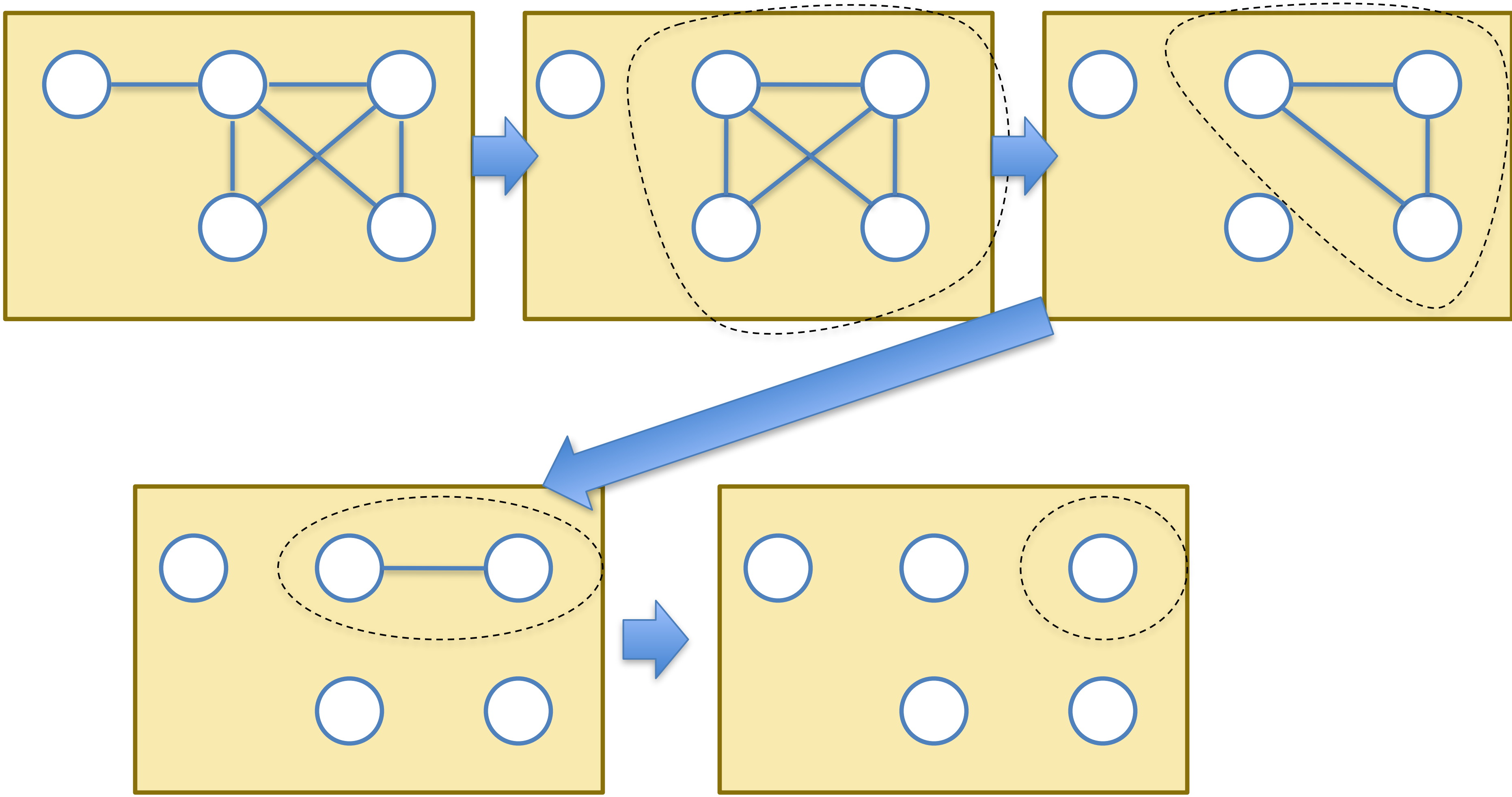
Recursing Down the Simplified Graphs

Example: 3-color this Graph



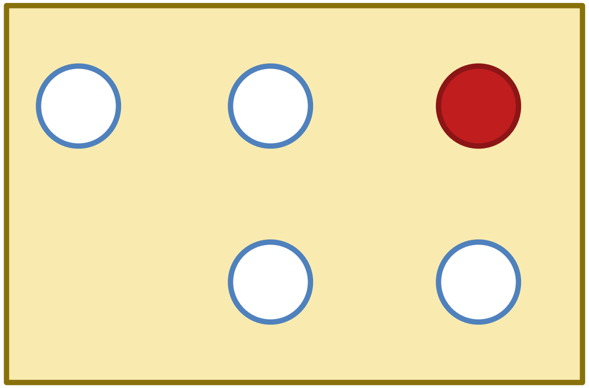
Recursing Down the Simplified Graphs

Example: 3-color this Graph



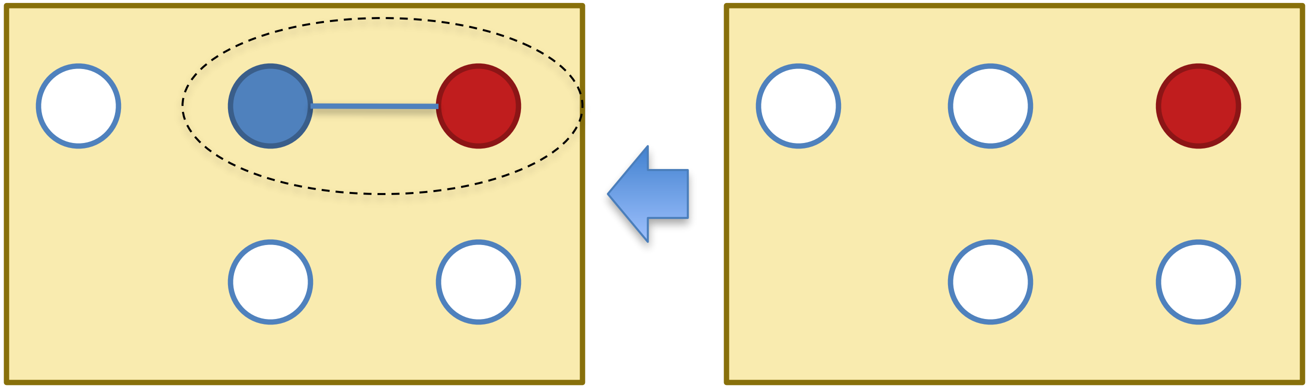
Recursing Down the Simplified Graphs

Example: 3-color this Graph



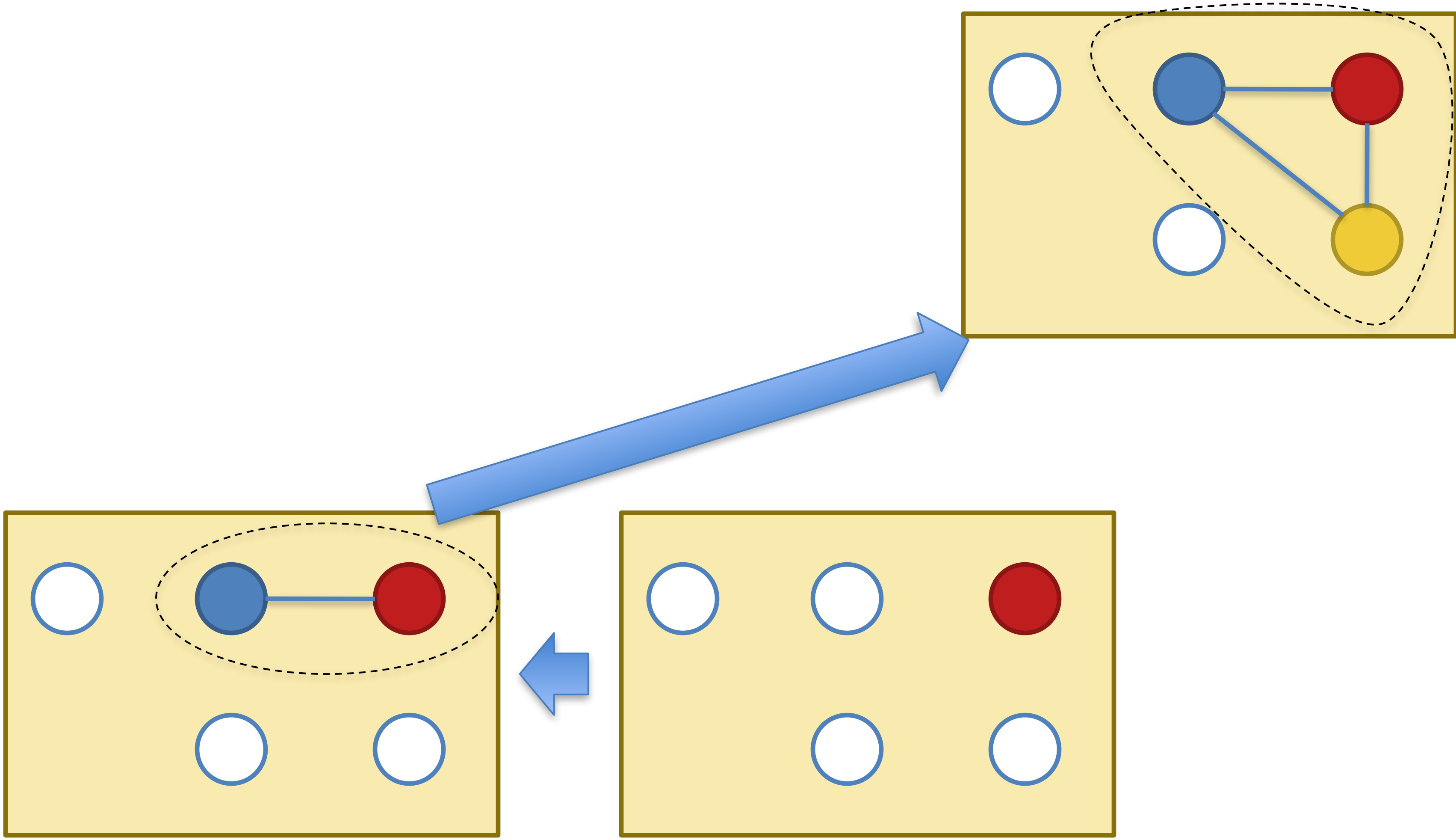
Assigning Colors on the way back up.

Example: 3-color this Graph



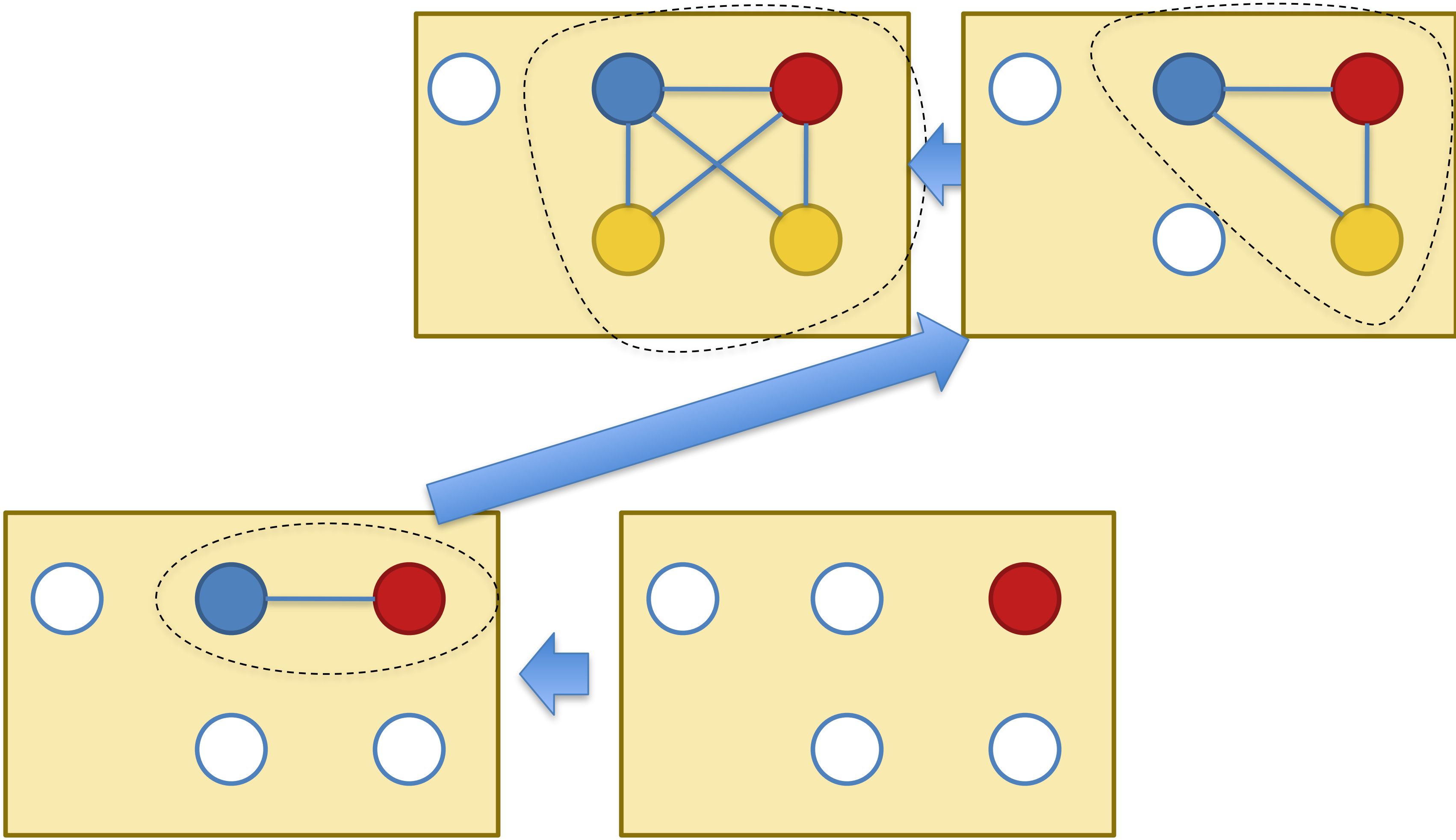
Assigning Colors on the way back up.

Example: 3-color this Graph



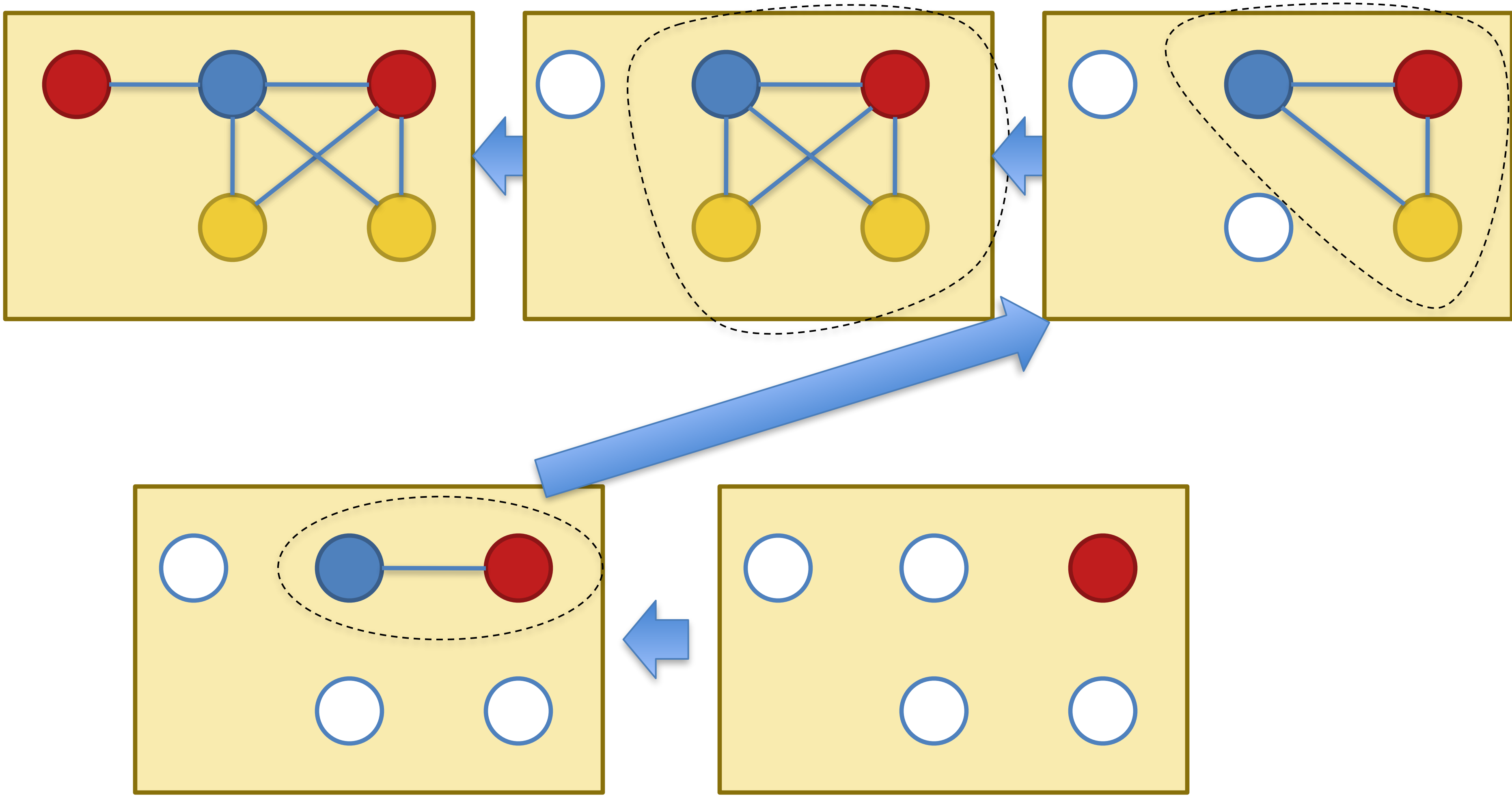
Assigning Colors on the way back up.

Example: 3-color this Graph



Assigning Colors on the way back up.

Example: 3-color this Graph

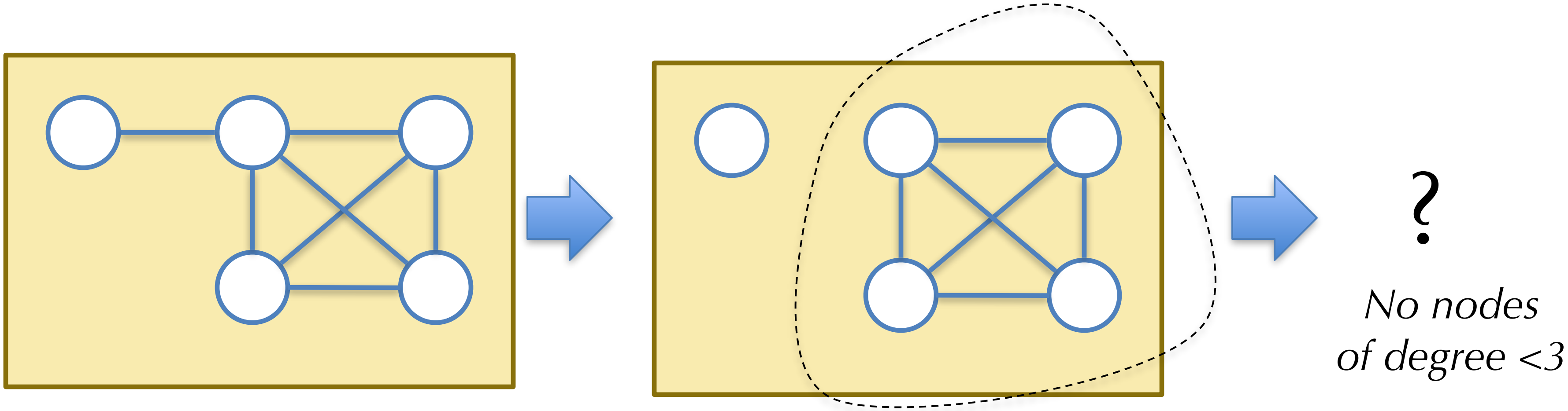


Assigning Colors on the way back up.

Failure of the Algorithm

- If the graph cannot be colored, it will simplify to a graph where every node has at least K neighbors.
  - This can happen even when the graph is K-colorable!
  - This is a symptom of NP-hardness (it requires search)

- Example: When trying to 3-color this graph:



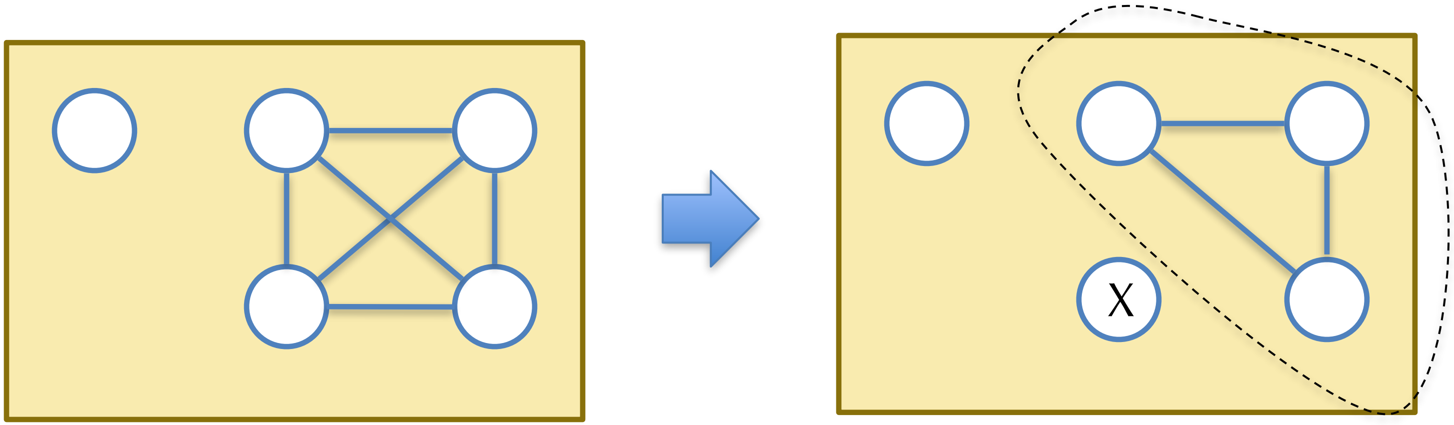
Spilling

- Idea: If we can't K-color the graph, we need to store one temporary variable on the stack.
- Which variable to spill?
  - Pick one that isn't used very frequently
  - Pick one that isn't used in a (deeply nested) loop
  - Pick one that has high interference (since removing it will make the graph easier to color)
- In practice: some weighted combination of these criteria

- When coloring:
  - Mark the node as spilled
  - Remove it from the graph
  - Keep recursively coloring

Spilling, Pictorially

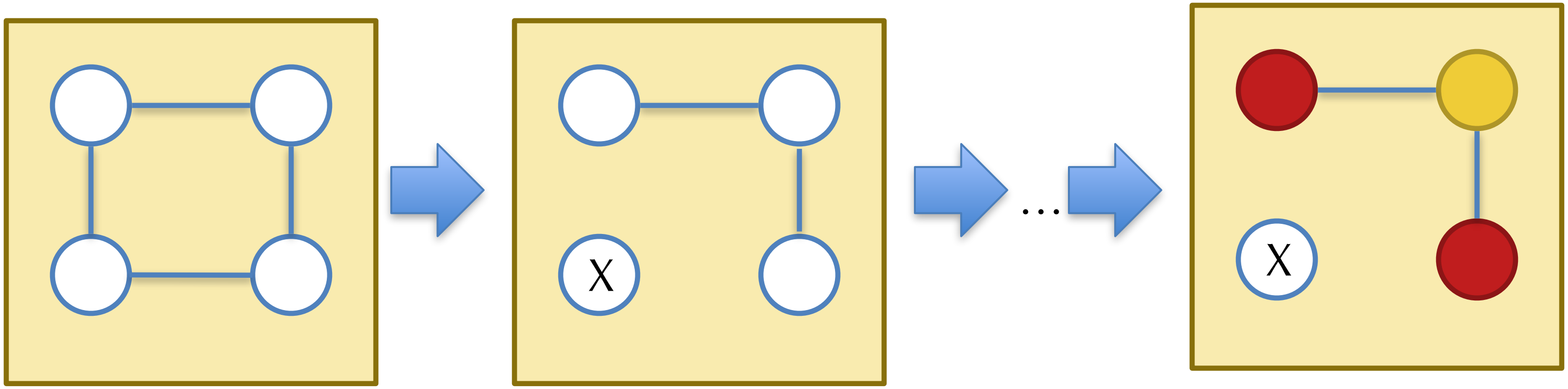
- Select a node to spill
- Mark it and remove it from the graph
- Continue coloring



Optimistic Coloring

- Sometimes it is possible to color a node marked for spilling.
  - If we get "lucky" with the choices of colors made earlier.

- Example: When 2-coloring this graph:



- Even though the node was marked for spilling, we can color it.
- So: on the way down, mark for spilling, but don't actually spill…

Accessing Spilled Registers

- If optimistic coloring fails, we need to *generate code* to move the spilled temporary to & from memory.
- **Option 1**: Reserve registers specifically for moving to/from memory.
  - Con: Need at least two registers (one for each source operand of an instruction), so decreases total # of available registers by 2.
  - Pro: Only need to color the graph once.
  - Not good on X86 (especially 32bit) because there are too few registers & too many constraints on how they can be used.
- **Option 2**: Rewrite the program to use a new temporary variable, with explicit moves to/from memory.
  - Pro: Need to reserve fewer registers.
  - Con: Introducing temporaries changes live ranges, so must recompute liveness & recolor graph

Spilling with explicit re-writing

- **Option 2**: Rewrite the program to use a new temporary variable, with explicit moves to/from memory.

- Suppose temporary `t` is marked for spilling to stack slot `[rbp+offs]`
- Rewrite the program like this:

```
t = a op b; // defn. of t
…
x = t op c; // use 1 of t
…
y = d op t; // use 2 of t
```

Spilling with explicit re-writing

- **Option 2**: Rewrite the program to use a new temporary variable, with explicit moves to/from memory.

- Suppose temporary `t` is marked for spilling to stack slot `[rbp+offs]`
- Rewrite the program like this:

```
t = a op b;  // defn. of t
…

x = t op c;  // use 1 of t
…

y = d op t;  // use 2 of t
```

```
t = a op b;
Mov [rbp+offs], t
…

Mov t37, [rbp+offs]
x = t37 op c
…

Mov t38, [rbp+offs]
y = d op t38
```

Spilling with explicit re-writing

- **Option 2**: Rewrite the program to use a new temporary variable, with explicit moves to/from memory.

- Suppose temporary `t` is marked for spilling to stack slot `[rbp+offs]`
- Rewrite the program like this:

```
t = a op b;   // defn. of t
…
x = t op c;   // use 1 of t
…
y = d op t;   // use 2 of t
```

```
t = a op b;
Mov [rbp+offs], t
…
Mov t37, [rbp+offs]
x = t37 op c
…
Mov t38, [rbp+offs]
y = d op t38
```

- `t37` and `t38` freshly generated tmps that replace `t` for different uses of `t`.
- Rewriting the code in this way breaks `t`'s live range up:

  `t`, `t37`, `t38` are only live across one edge

Precolored Nodes

- Some variables must be pre-assigned to registers.
  - E.g. on X86 the multiplication instruction: IMul must define %rax
  - The "Call" instruction should kill the caller-save registers %rax, %rcx, %rdx.
  - Any temporary variable live across a call interferes with the caller-save registers.

- To properly allocate temporaries, we treat registers as nodes in the interference graph with pre-assigned colors.
  - Pre-colored nodes *can't be removed* during simplification.
  - Trick: Treat pre-colored nodes as having "infinite" degree in the interference graph – this guarantees they won't be simplified.
  - When the graph is *empty except the pre-colored nodes*, we have reached the point where we start coloring the rest of the nodes.

Picking Good Colors

- When choosing colors during the coloring phase, *any* choice is semantically correct, but some choices are better for performance.
- Example:

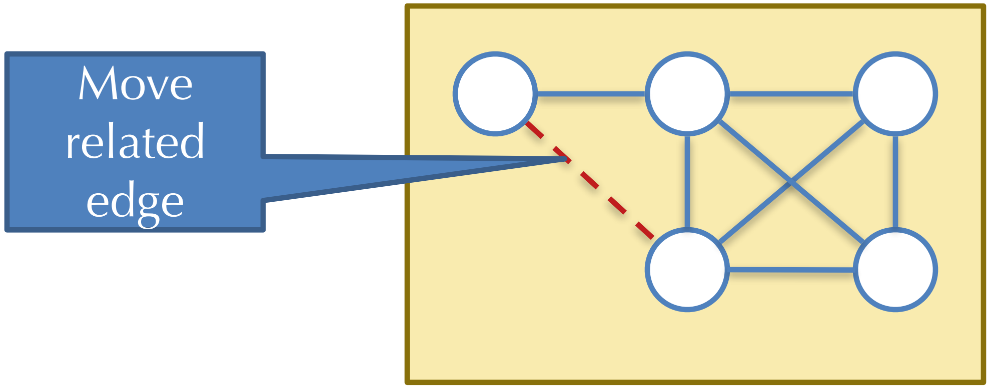
```
movq t1, t2
```

  - If t1 and t2 can be assigned the same register (color) then this move is redundant and can be eliminated.

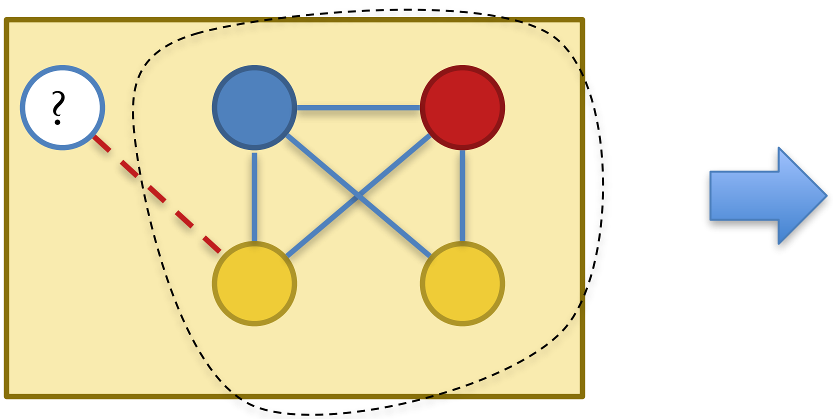
- A simple color choosing strategy that helps eliminate such moves:
  - Add a new kind of "move related" edge between the nodes for t1 and t2 in the interference graph.
  - When choosing a color for t1 (or t2), if possible pick a color of an already colored node reachable by a move-related edge.

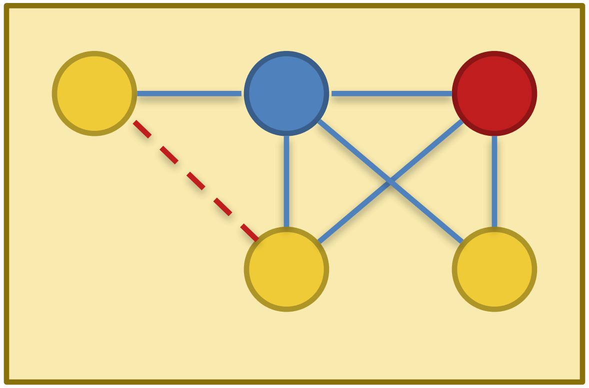
Example Color Choice

- Consider 3-coloring this graph, where the dashed edge indicates that there is a Mov from one temporary to another.



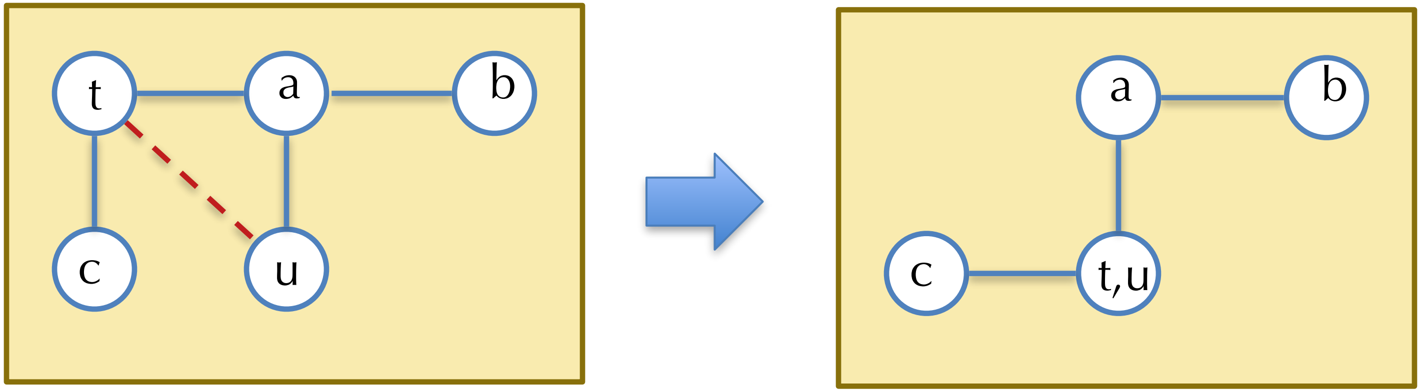
- After coloring the rest, we have a choice:
  - Picking yellow is better than red because it will eliminate a move.



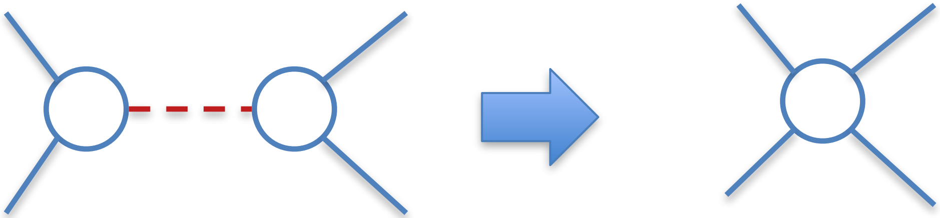


Coalescing Interference Graphs

- A more aggressive strategy is to *coalesce* nodes of the interference graph if they are connected by move-related edges.
  - Coalescing the nodes *forces* the two temporaries to be assigned the same register.



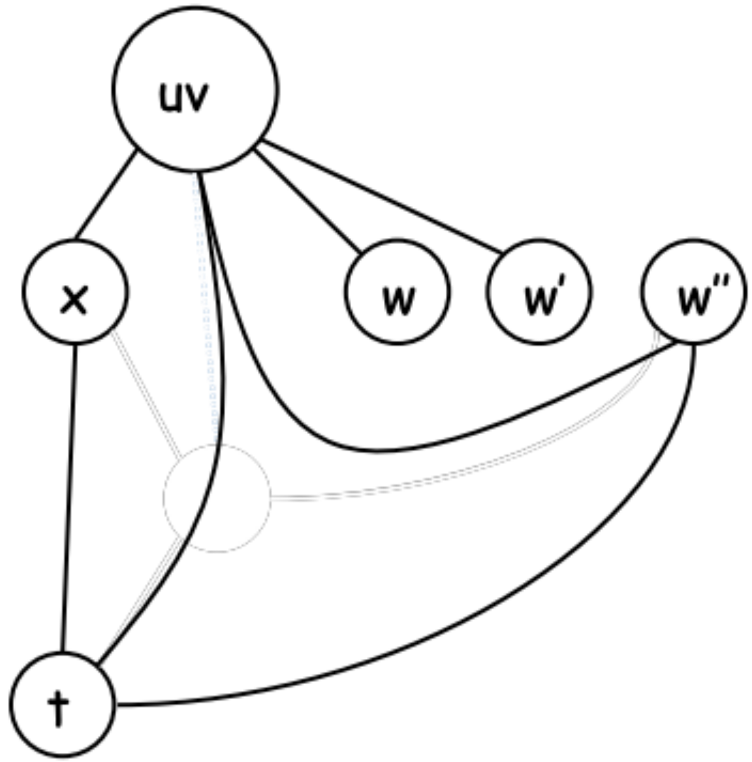
- Idea: interleave simplification and coalescing to maximize the number of moves that can be eliminated.
- Problem: coalescing can sometimes increase the degree of a node.



Conservative Coalescing

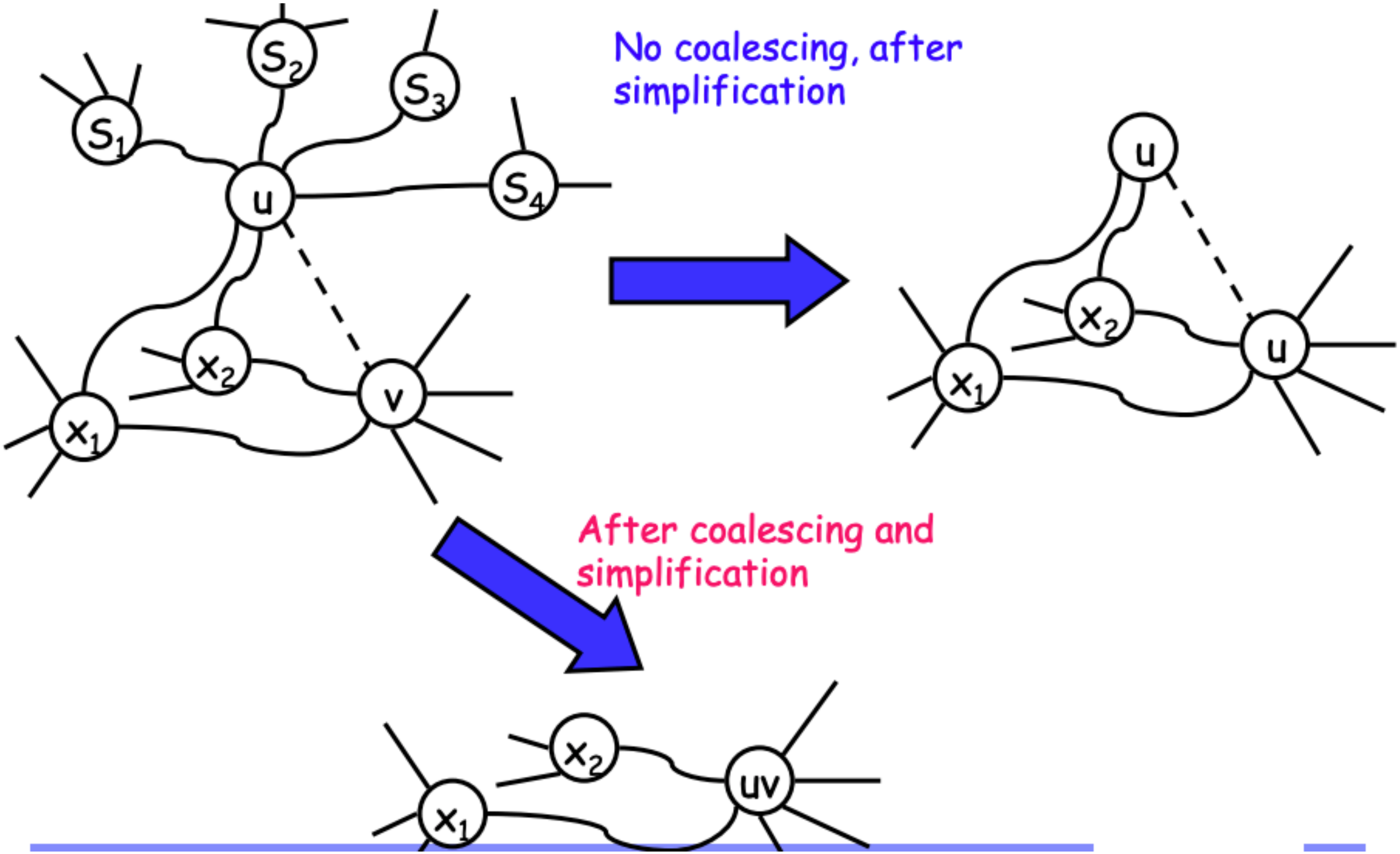
- Two strategies are guaranteed to preserve the k-colorability of the interference graph.

- *Brigg's strategy*: It's safe to coalesce u & v **if**:
  - the resulting node will have *fewer than k neighbors* (with degree $\geq$ k).



Conservative Coalescing

George's strategy: We can safely coalesce u & v if for every neighbor t of u, either t already interferes with v or t has degree < k.



Complete Register Allocation Algorithm

1. Build interference graph (precolor nodes as necessary).
   - Add move related edges
2. Reduce the graph (building a stack of nodes to color).
   1. **Simplify** the graph as much as possible without removing nodes that are move related (i.e. have a move-related neighbor). Remaining nodes are *high degree* or *move-related*.
   2. **Coalesce** move-related nodes using Brigg's or George's strategy.
   3. Coalescing can reveal more nodes that can be *simplified*, so repeat 2.1 and 2.2 until no node can be simplified or coalesced.
   4. **Freeze:** If no nodes can be coalesced, *freeze* (remove) a move-related edge and keep trying to simplify/coalesce.
3. If there are non-precolored nodes left, *mark one for spilling*, remove it from the graph and continue doing step 2.
4. When only pre-colored node remain, *start coloring* (popping simplified nodes off the top of the stack).
   1. If a node must be spilled, insert spill code as on slide 14 and rerun the whole register allocation algorithm starting at step 1.