L1

```
     p ::= ((i ...) (label i ...) ...)
     i ::= (x <- s)
         | (x <- (mem x n4))
         | ((mem x n4) <- s)
         | (x aop= s)
         | (x sop= num)
         | (x sop= sx)
         | (cx <- s cmp s)
         | label
         | (goto label)
         | (cjump s cmp s label label)
         | (call s)
         | (tail-call s)
         | (return)
         | (eax <- (print s))
         | (eax <- (allocate s s))
aop= ::= += | -= | *= | &=
 sop ::= <<= | >>=
 cmp ::= < | <= | =
   s ::= x | num | label
x, y ::= cx | esi | edi | ebp | esp
  cx ::= eax | ecx | edx | ebx
  sx ::= ecx
```

L2

```
      p ::= ((i ...) (label i ...) ...)
      i ::= (x <- s)
          | (x <- (mem x n4))
          | ((mem x n4) <- s)
          | (x aop= s)
          | (x sop= num)
          | (x sop= sx)
          | (cx <- s cmp s)
          | label
          | (goto label)
          | (cjump s cmp s label label)
          | (call s)
          | (tail-call s)
          | (return)
          | (eax <- (print s))
          | (eax <- (allocate s s))
   aop= ::= += | -= | *= | &=
    sop ::= <<= | >>=
    cmp ::= < | <= | =
      s ::= x | num | label
   x, y ::= any-variable-at-all | reg
     cx ::= any-variable-at-all | reg
     sx ::= any-variable-at-all | reg
    reg ::= eax | ecx | edx | ebx | esi | edi | ebp | esp
```

L2 semantics: variables

L2 behaves just like L1, except that non-reg variables are function local, e.g.,

```
(define (f x)     ⇒   ((; :main
  (+ (g x) 1))           (eax <- 10)
                         (call :f))
(define (g x)           (:f (temp <- 1)
  (+ x 2))                  (call :g)
                            (eax += temp)
(f 10)                      (return))
                        (:g (temp <- 2)
                            (eax += 2)
                            (return)))
```

The assignment to `temp` in `g` does not break `f`, but if `temp` were a register, it would.

L2 semantics: esp & ebp

L2 programs must use neither `esp` nor `ebp`. They are in L2 to facilitate register allocation only, *not* for the L3 → L2 compiler's use.

From L2 to L1

Register allocation, in three parts; for each function body we do:

- **Liveness analysis** ⇒ interference graph (nodes are variables; edges indicate “cannot be in the same register”)
- **Graph coloring** ⇒ register assignments
- **Spilling** coping with too few registers
- Bonus part, **coalescing** eliminating redundant `(x <- y)` instructions

Example Function

```
;; f(x) = 2*x^2
;;        + 3x + 4
(:f
 (x2 <- edx)
 (x2 *= x2)
 (2x2 <- x2)
 (2x2 *= 2)
 (3x <- edx)
 (3x *= 3)
 (eax <- 2x2)
 (eax += 3x)
 (eax += 4)
 (return))
```

Example Function

```
;; f(x) = 2*x^2
;;       + 3x + 4
(:f
 (x2 <- edx)
 (x2 *= x2)
 (2x2 <- x2)
 (2x2 *= 2)
 (3x <- edx)
 (3x *= 3)
 (eax <- 2x2)
 (eax += 3x)
 (eax += 4)
 (return))
```

x2 and **2x2** are not both live at the same time. Ditto for **x2** and **3x**. But **3x** and **2x2** are.

Thus **3x** and **2x2** have to be in different registers, but **x2** can share a register with either.

Liveness analysis

Fixed point computation:

- Define which variables are set and which are used, for each instruction: **kill** & **gen** functions
- Specify how nearby instructions transmit live values around the program: **in** & **out** functions
- Iterate until nothing changes

kill, gen : i → register set

kill(s) = {all assigned registers in stm s}

gen(s) = {all referenced registers in stm s}

Gen & Kill

		gen	kill
1:	`:f`	()	()
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2)
5:	`(2x2 *= 2)`	(2x2)	(2x2)
6:	`(3x <- edx)`	(edx)	(3x)
7:	`(3x *= 3)`	(3x)	(3x)
8:	`(eax <- 2x2)`	(2x2)	(eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(eax)
11:	`(return)`	()	()

in, out : Nat → register set

in(n) = gen(n-th-inst) ∪ (out (n) - kill(n-th-inst))

out(n) = ∪{in(m) | m ∈ succ(n)}

Intuition behind the definitions: if x ∈ **in**(n) then we know that x is live between instructions n-1 and n. Similarly, if x ∈ **out**(n) then we know that x is live between instructions n and n+1.

in, out : Nat → register set

in(n) = gen(n-th-inst) ∪ (out (n) - kill(n-th-inst))

out(n) = ∪{in(m) | m ∈ succ(n)}

One possible solution: every variable is in **in**(n) and in **out**(n) for every n.

But that isn't super helpful... since we want to know what values don't need to be in registers at each point.

in, out : Nat → register set

in(n) = gen(n-th-inst) ∪ (out (n) - kill(n-th-inst))

out(n) = ∪{in(m) | m ∈ succ(n)}

Instead, compute **in** and **out** by iterating the equations until we reach a fixed point. Starting out with the assumption that they map everything to the empty set. Then repeatedly apply the right hand sides until nothing changes.

That result will satisfy the definitions of **in** and **out** but will it be the smallest solution?

Liveness

		in	**out**
1:	`:f`	()	()
2:	`(x2 <- edx)`	()	()
3:	`(x2 *= x2)`	()	()
4:	`(2x2 <- x2)`	()	()
5:	`(2x2 *= 2)`	()	()
6:	`(3x <- edx)`	()	()
7:	`(3x *= 3)`	()	()
8:	`(eax <- 2x2)`	()	()
9:	`(eax += 3x)`	()	()
10:	`(eax += 4)`	()	()
11:	`(return)`	()	()

Liveness

		in	out
1:	`:f`	()	()
2:	`(x2 <- edx)`	(edx)	()
3:	`(x2 *= x2)`	(x2)	()
4:	`(2x2 <- x2)`	(x2)	()
5:	`(2x2 *= 2)`	(2x2)	()
6:	`(3x <- edx)`	(edx)	()
7:	`(3x *= 3)`	(3x)	()
8:	`(eax <- 2x2)`	(2x2)	()
9:	`(eax += 3x)`	(3x eax)	()
10:	`(eax += 4)`	(eax)	()
11:	`(return)`	()	()

Liveness

		in	out
1:	`:f`	()	(edx)
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2)
5:	`(2x2 *= 2)`	(2x2)	(edx)
6:	`(3x <- edx)`	(edx)	(3x)
7:	`(3x *= 3)`	(3x)	(2x2)
8:	`(eax <- 2x2)`	(2x2)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	()
11:	`(return)`	()	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2)
5:	`(2x2 *= 2)`	(2x2 edx)	(edx)
6:	`(3x <- edx)`	(edx)	(3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	()
11:	`(return)`	()	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(edx)
6:	`(3x <- edx)`	(edx)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	()
11:	`(return)`	()	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(edx)
6:	`(3x <- edx)`	(2x2 edx)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	()
11:	`(return)`	()	()

Liveness

		in	**out**
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(x2)	(edx x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(2x2 edx)
6:	`(3x <- edx)`	(2x2 edx)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	()
11:	`(return)`	()	()

Liveness

		in	**out**
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(edx x2)	(edx x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(2x2 edx)
6:	`(3x <- edx)`	(2x2 edx)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	()
11:	`(return)`	()	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(edx x2)
3:	`(x2 *= x2)`	(edx x2)	(edx x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(2x2 edx)
6:	`(3x <- edx)`	(2x2 edx)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	()
11:	`(return)`	()	()

Graph coloring

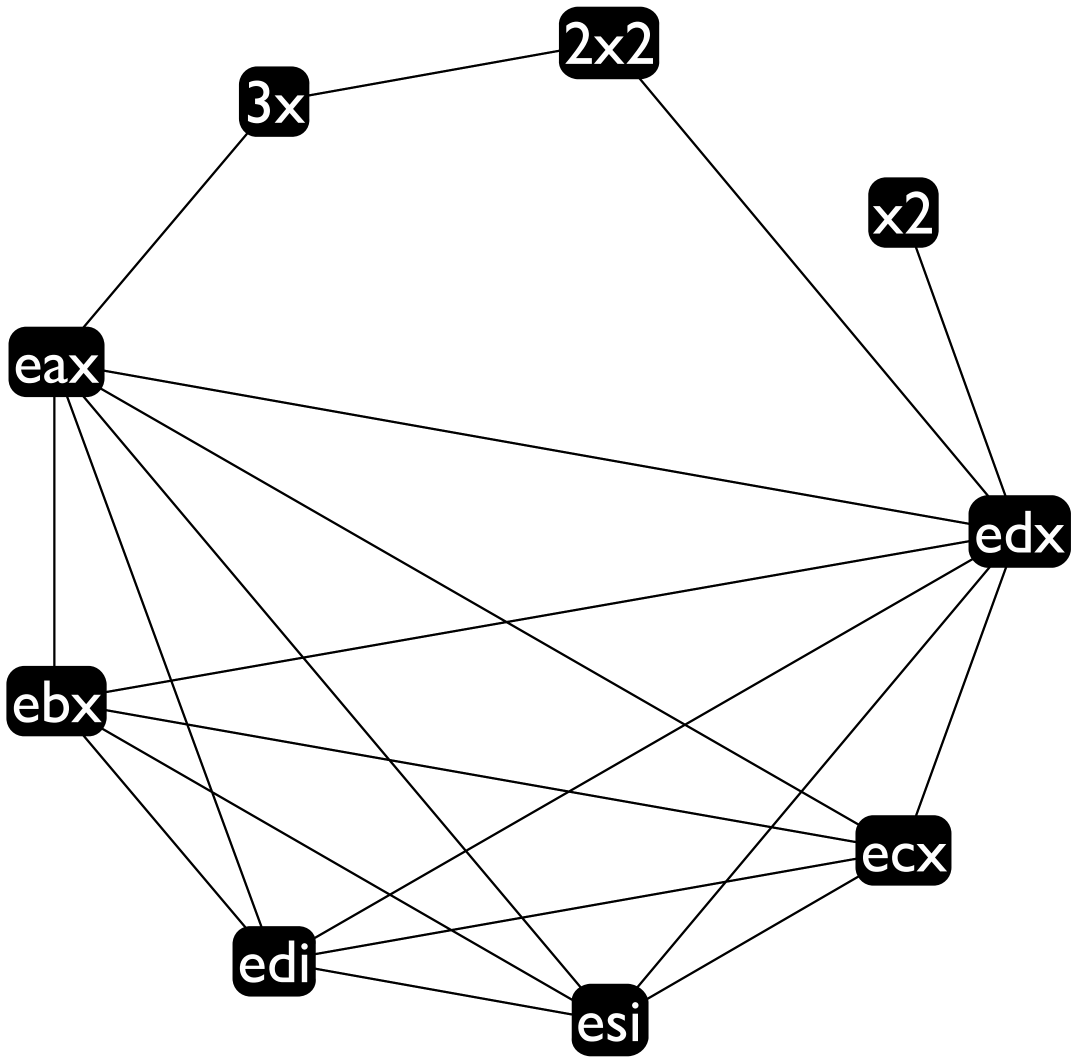
Reduce register allocation to the graph coloring problem

- Nodes represent variables
- Edges connect nodes that cannot share a register

Graph coloring

Build interference graph from the liveness information

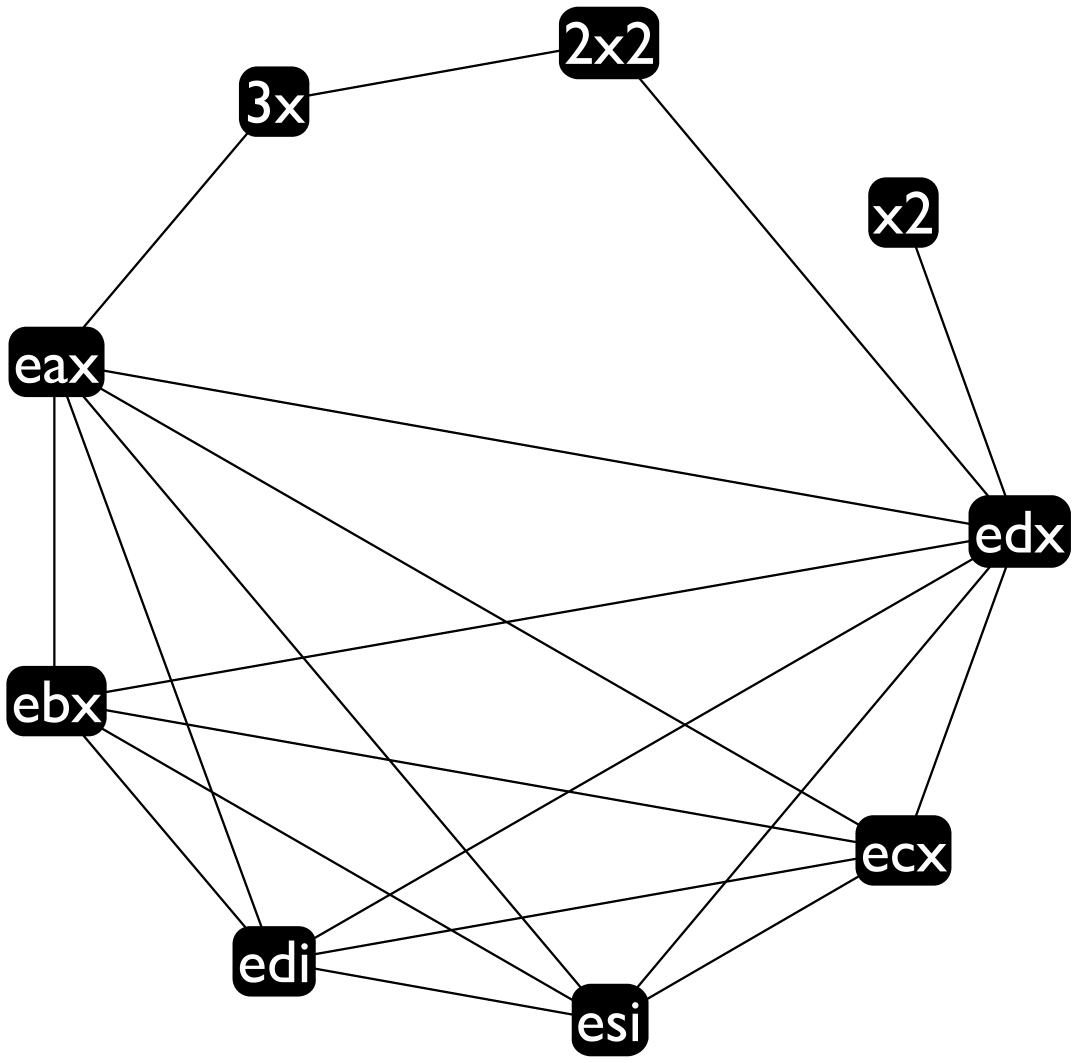
- Two variable live at the same time interfere with each other
- Killed variables interferes with all live variables at that point, unless it is a `(x <- y)` instruction (in which case it is fine if `x` and `y` share a register)
- All real registers interfere with each other

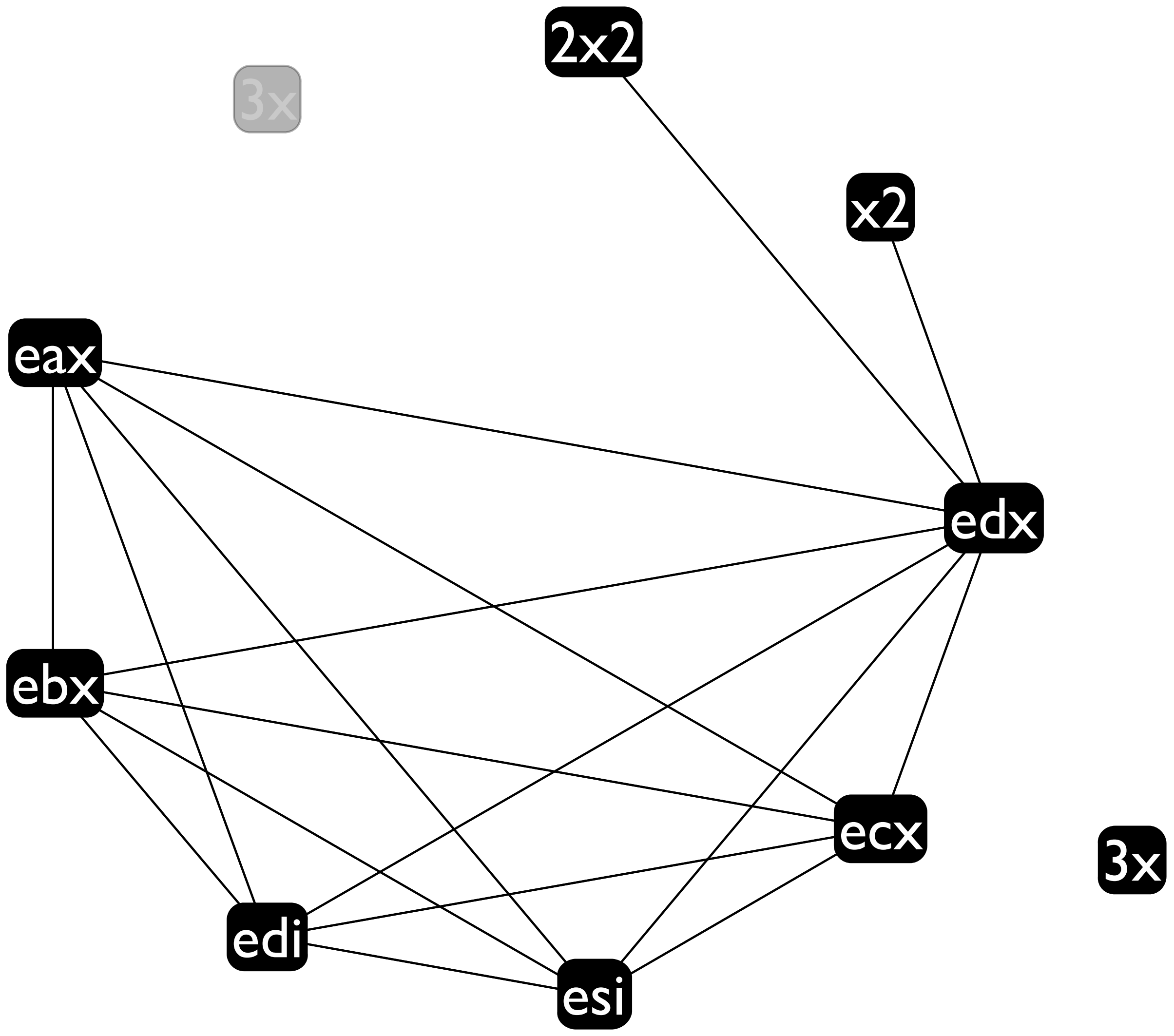

2x2
3x
x2
eax
edx
ebx
ecx
edi
esi

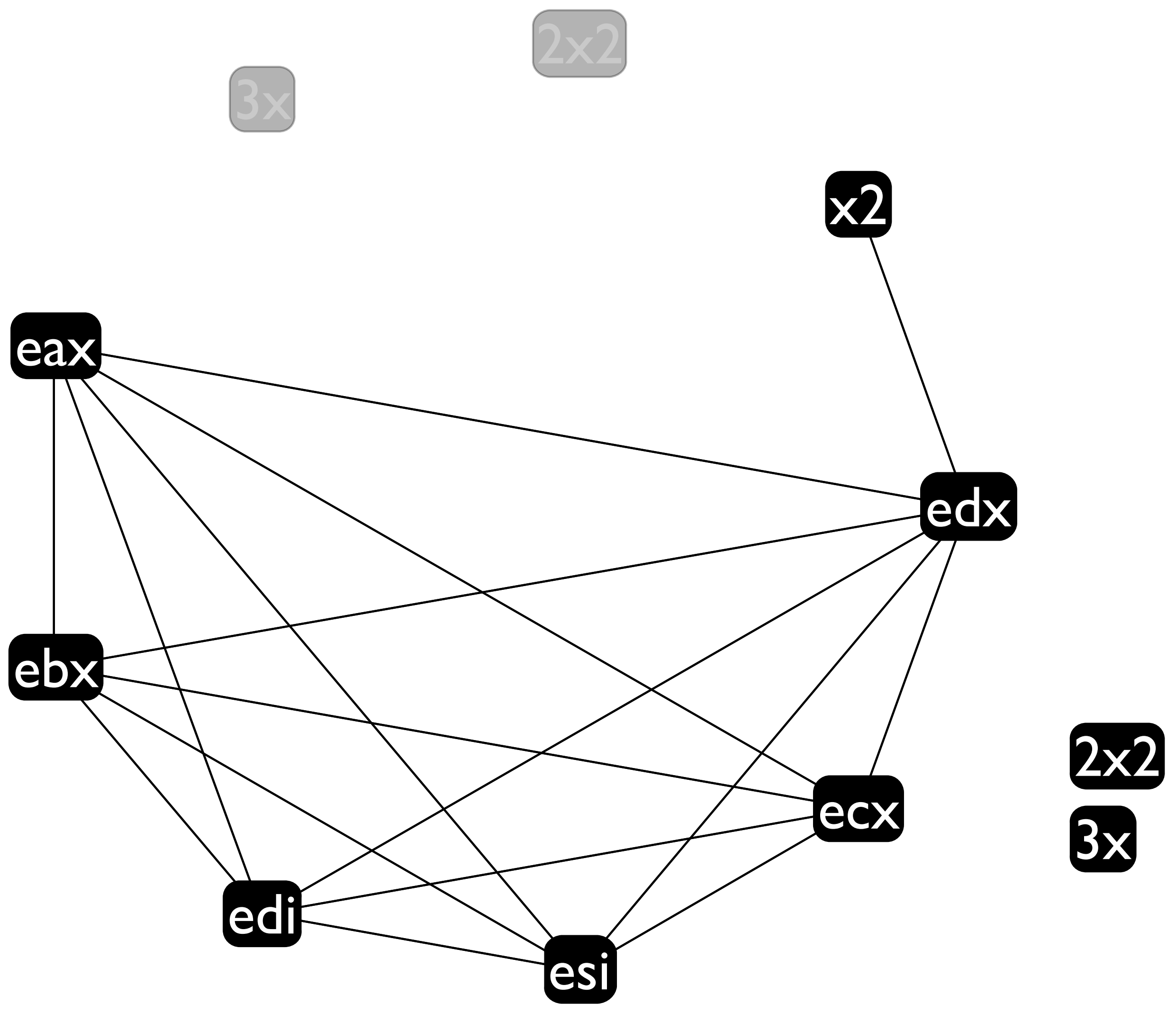
Graph coloring

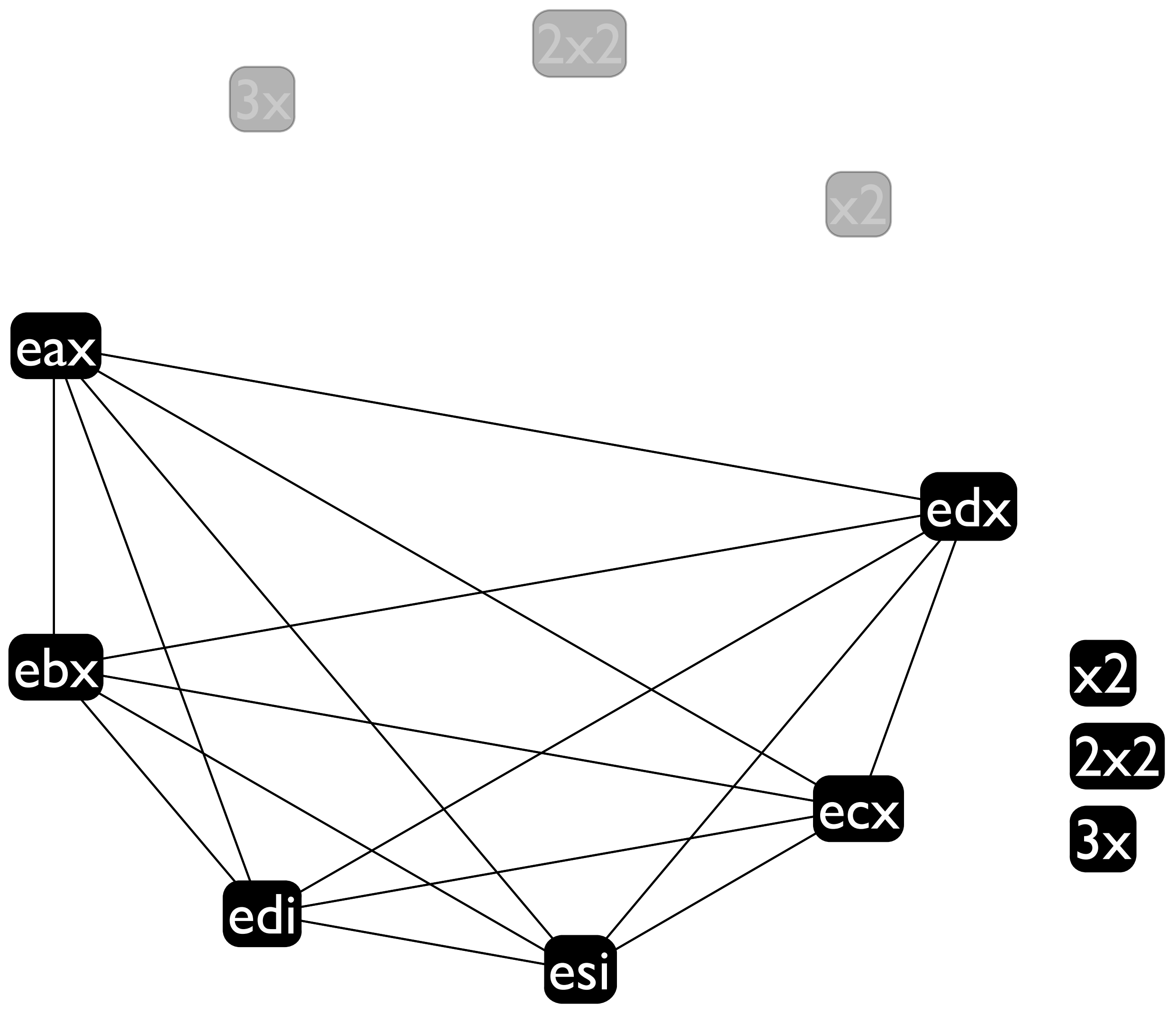
Algorithm:

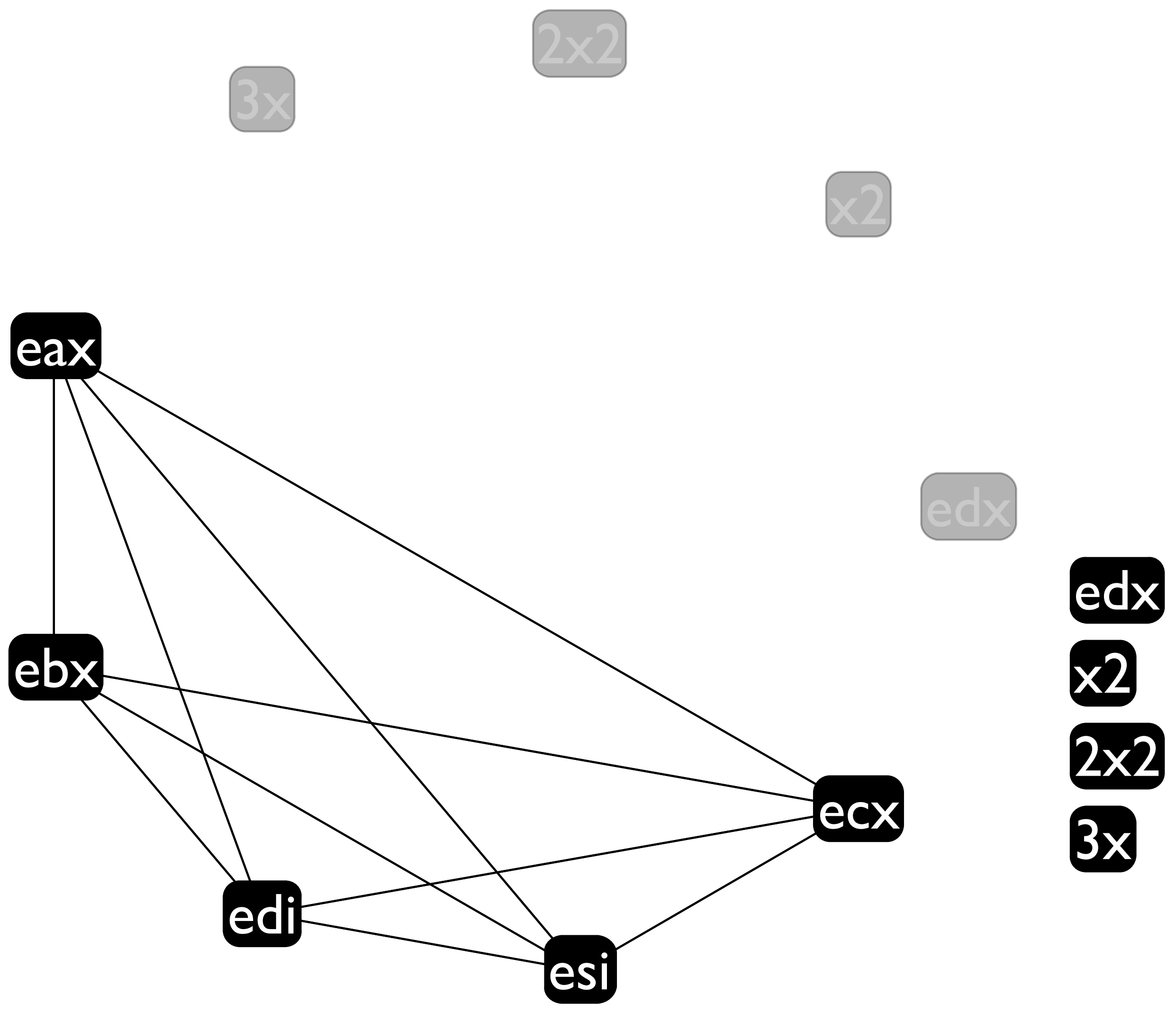
- Repeatedly select a node and remove it from the graph, putting it onto a stack
- When the graph is empty, rebuild it, putting a new color on each node as it comes back into the graph, making sure no adjacent nodes have the same color
- If there are not enough colors, the algorithm fails (spilling comes in here)
- Make sure real registers come out of the graph last so they get the first 6 colors

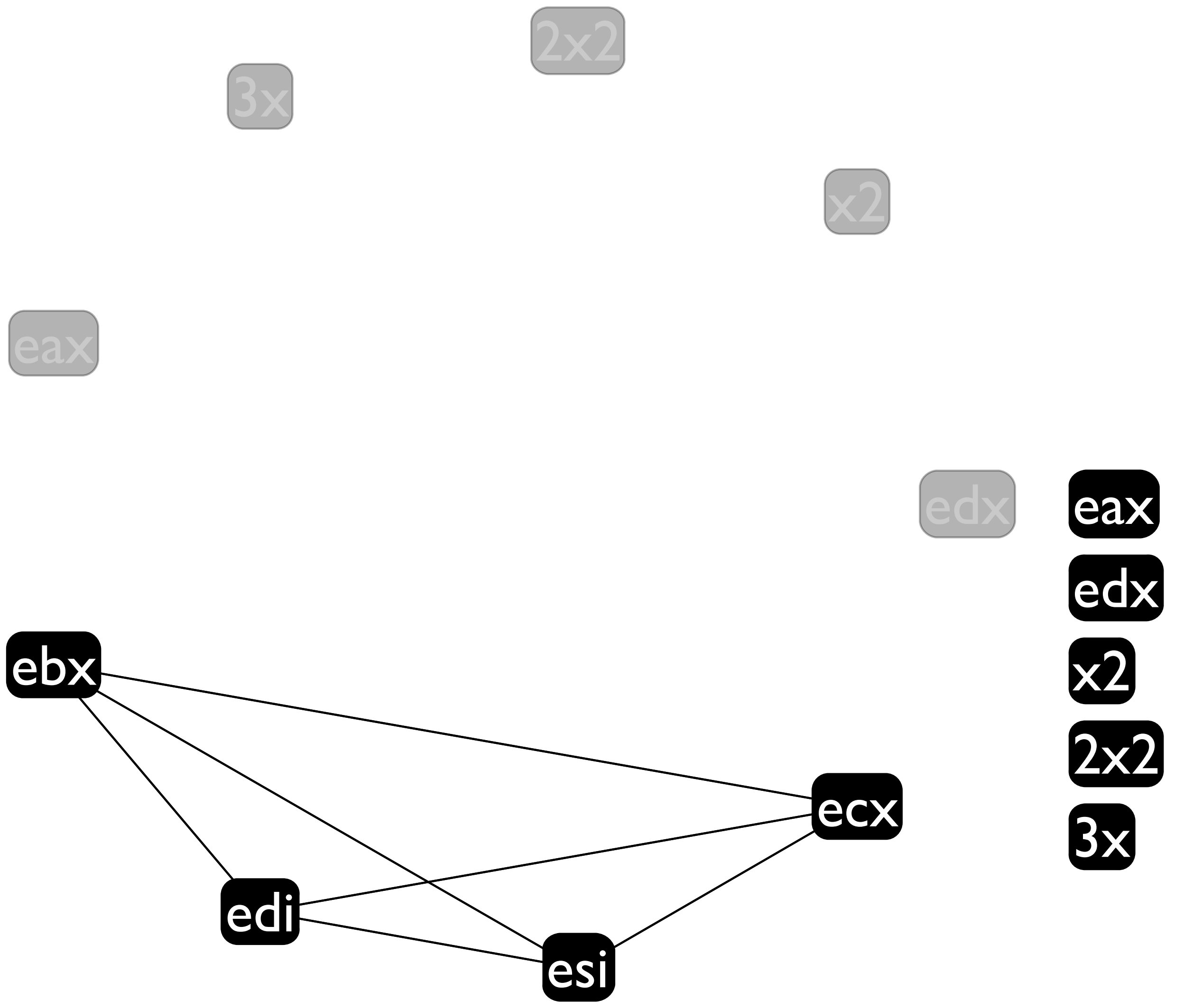

2x2
3x
x2
eax
edx
ebx
ecx
edi
esi

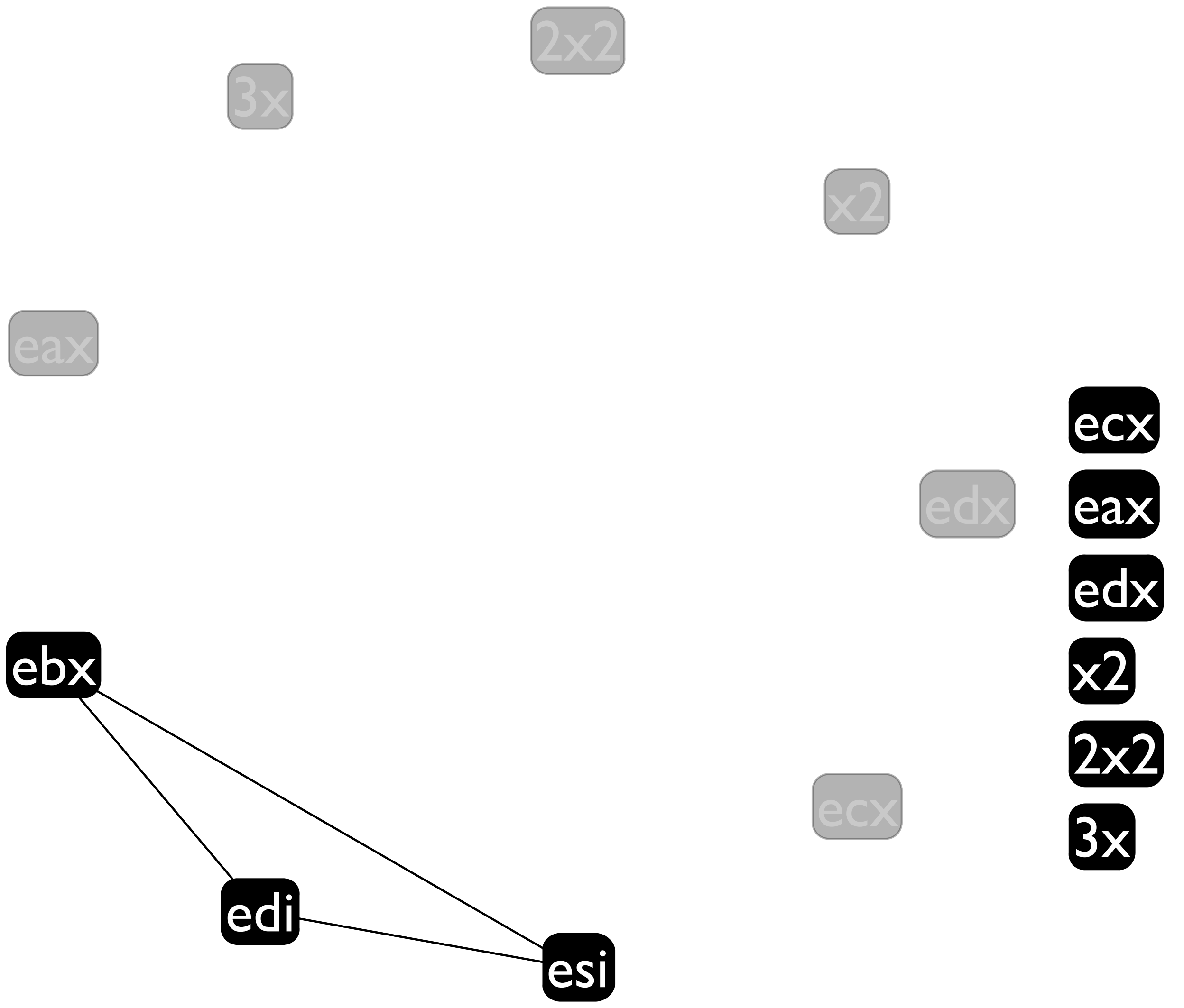

3x
2x2
x2
eax
edx
ebx
ecx
3x
edi
esi


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
2x2
3x

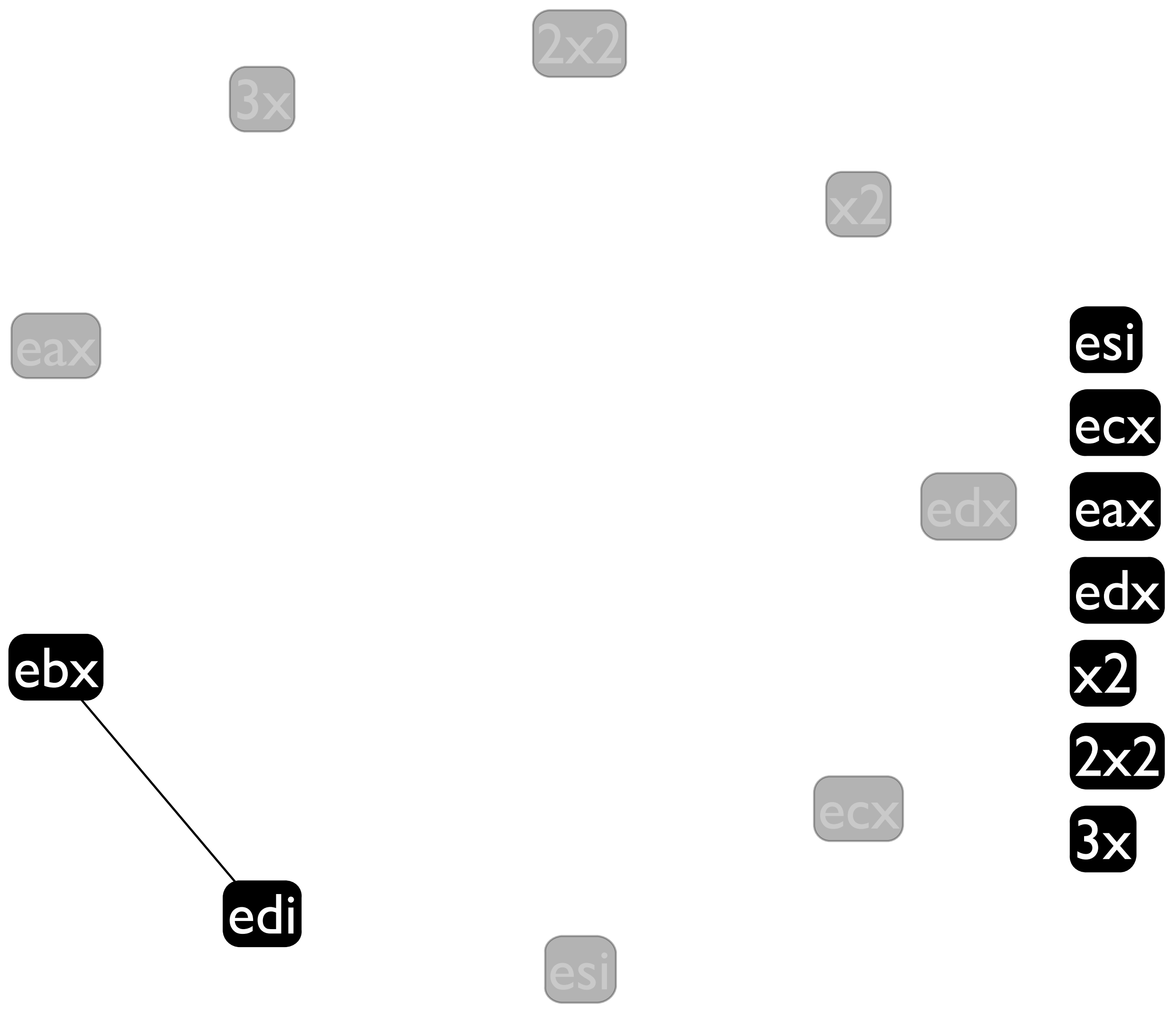

2x2
3x
x2
eax
edx
ebx
x2
2x2
3x
ecx
edi
esi

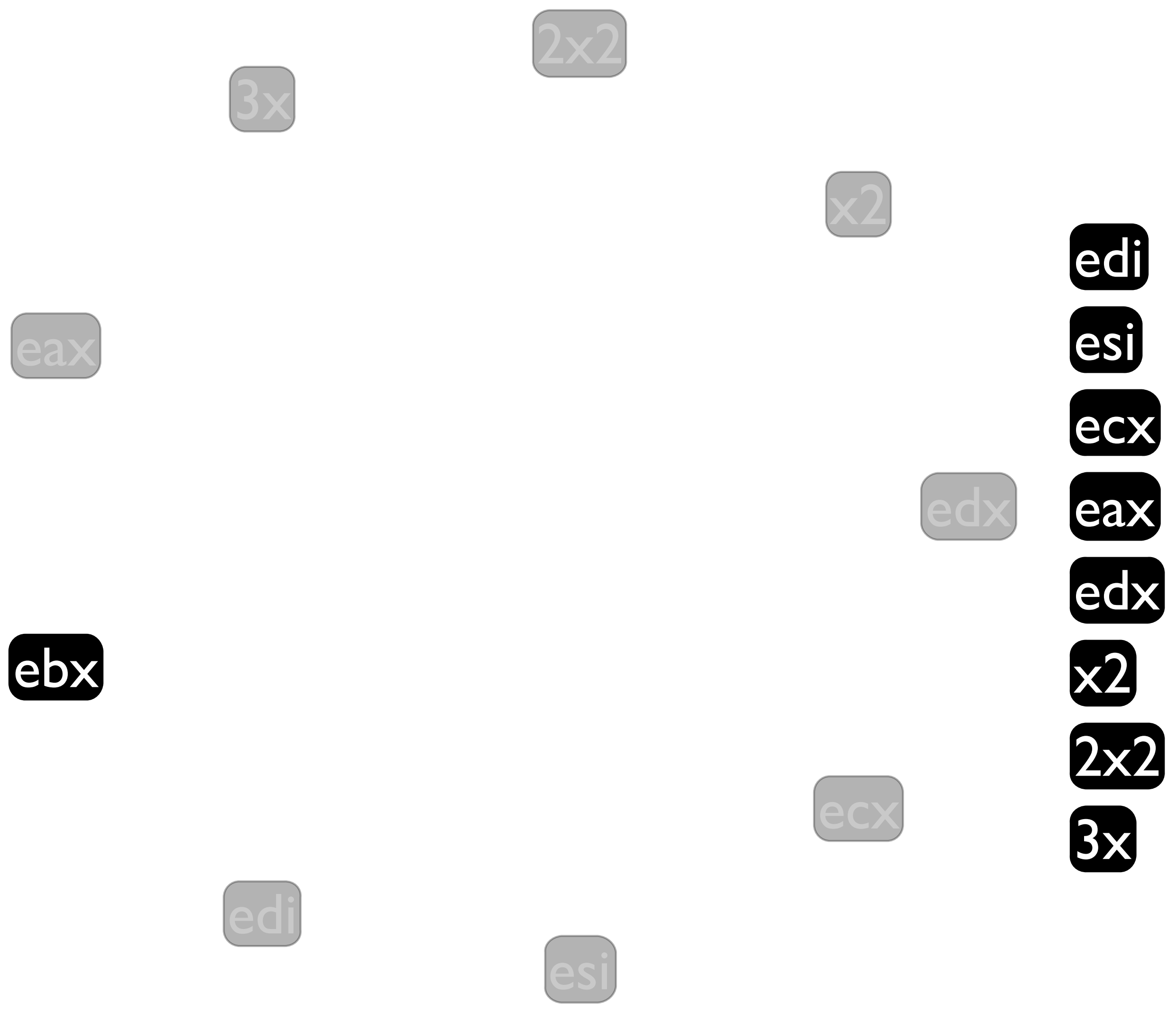

2x2
3x
x2
eax
edx
edx
ebx
x2
2x2
ecx
3x
edi
esi

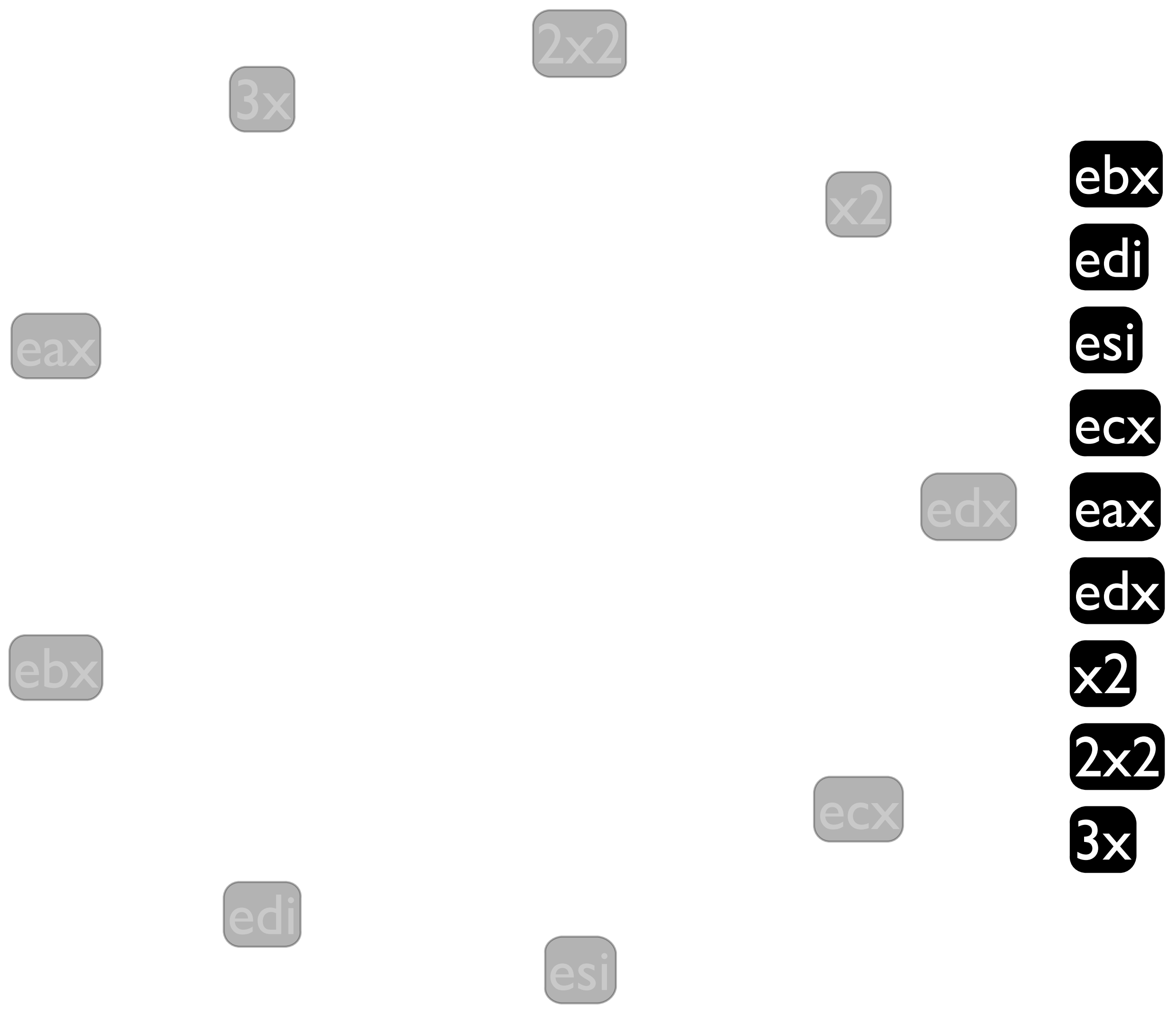

2x2
3x
x2
eax
edx
eax
edx
x2
2x2
3x
ebx
ecx
edi
esi

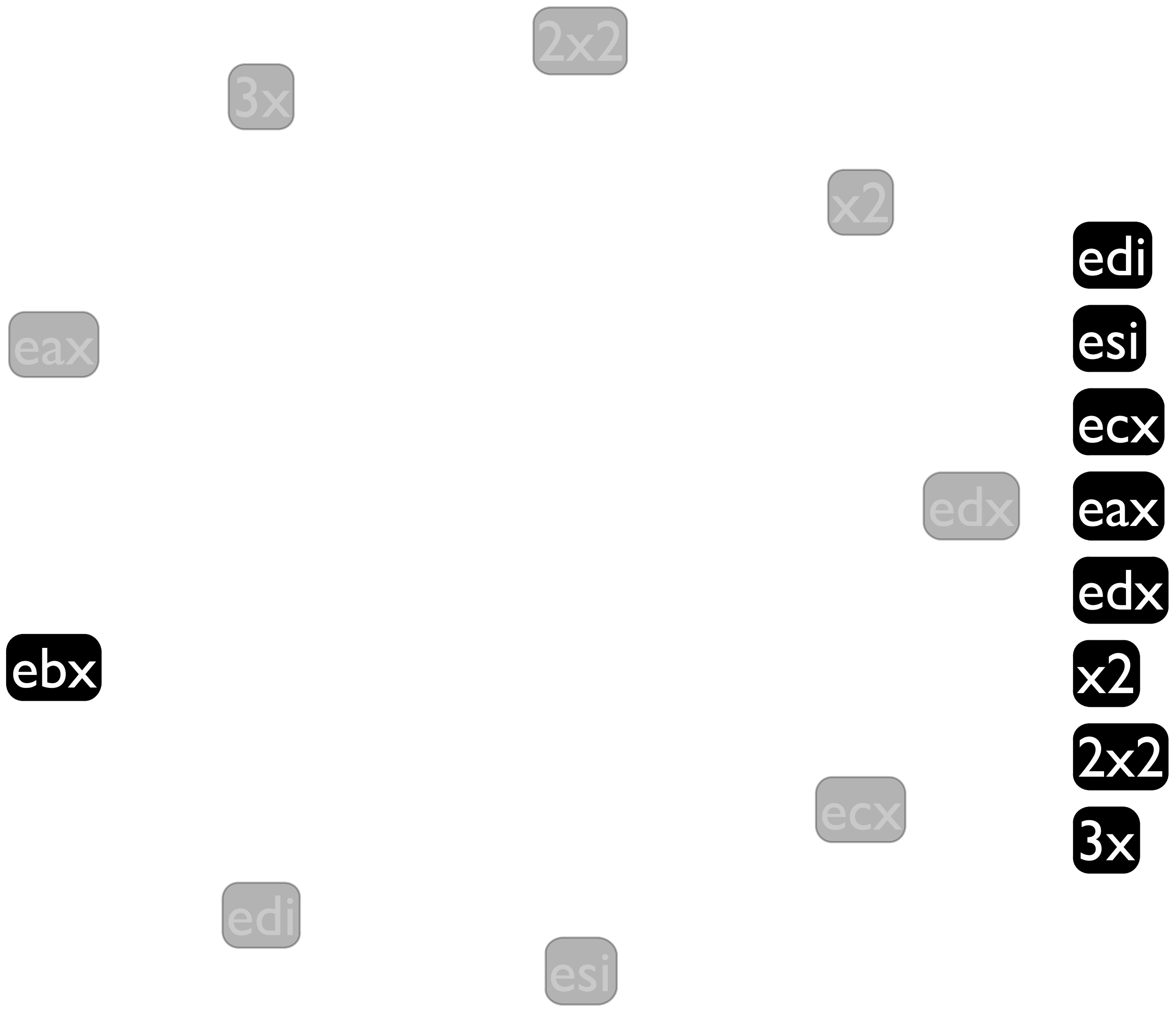


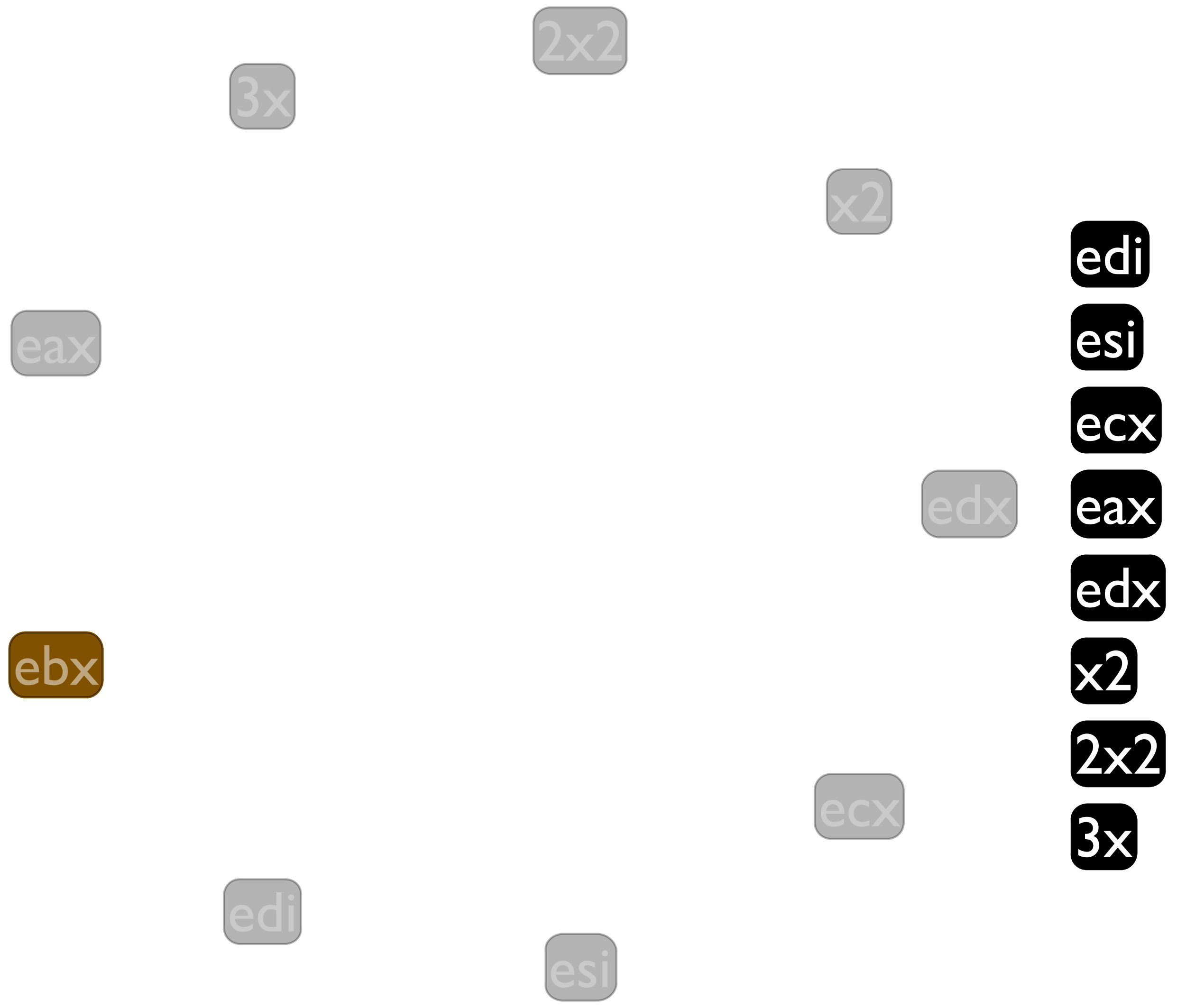
2x2
3x
x2
eax
edx
ecx
ebx
edi
esi
ecx
eax
edx
x2
2x2
3x

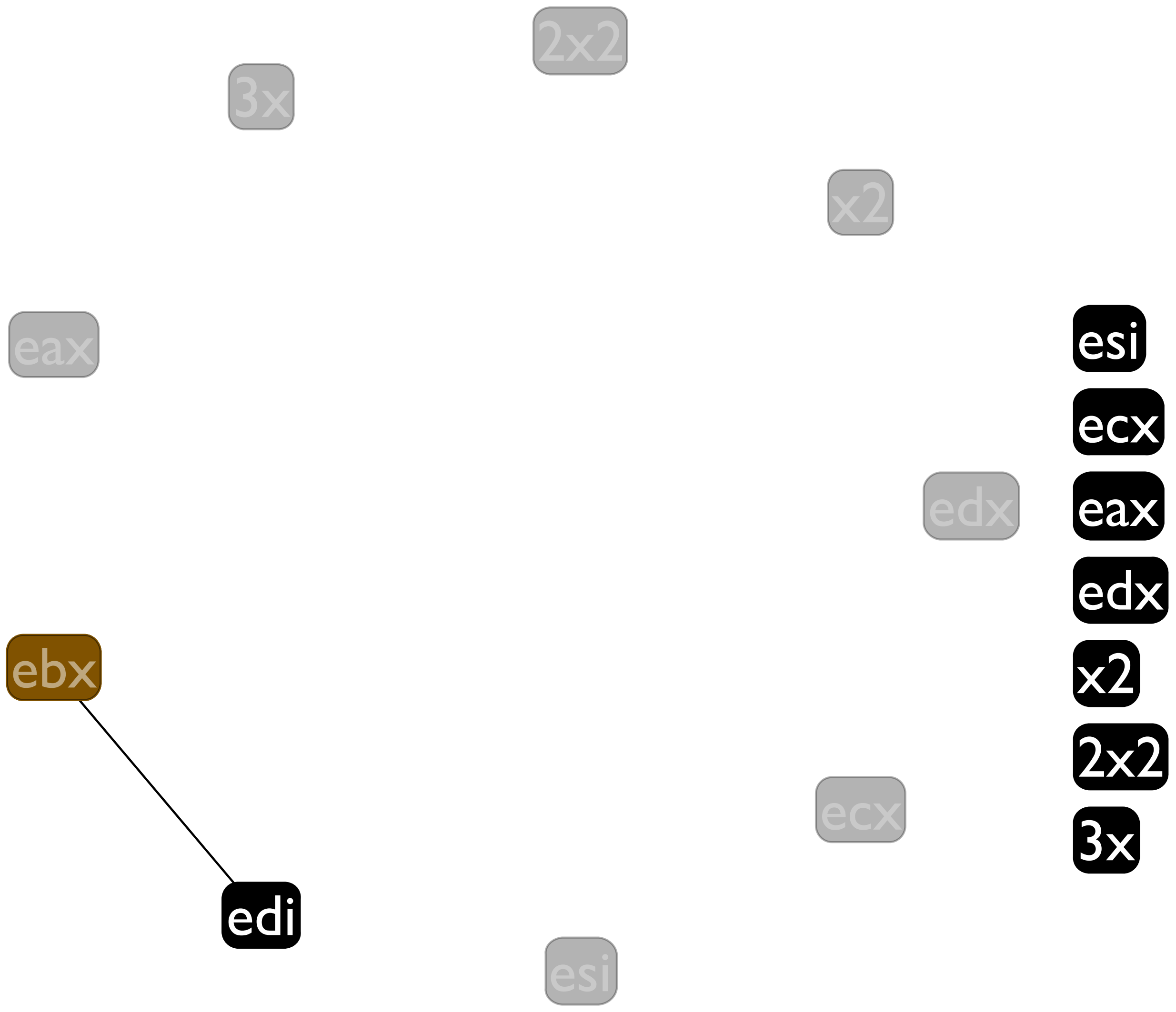

2x2
3x
x2
eax
esi
ecx
edx
eax
edx
ebx
x2
2x2
ecx
3x
edi
esi

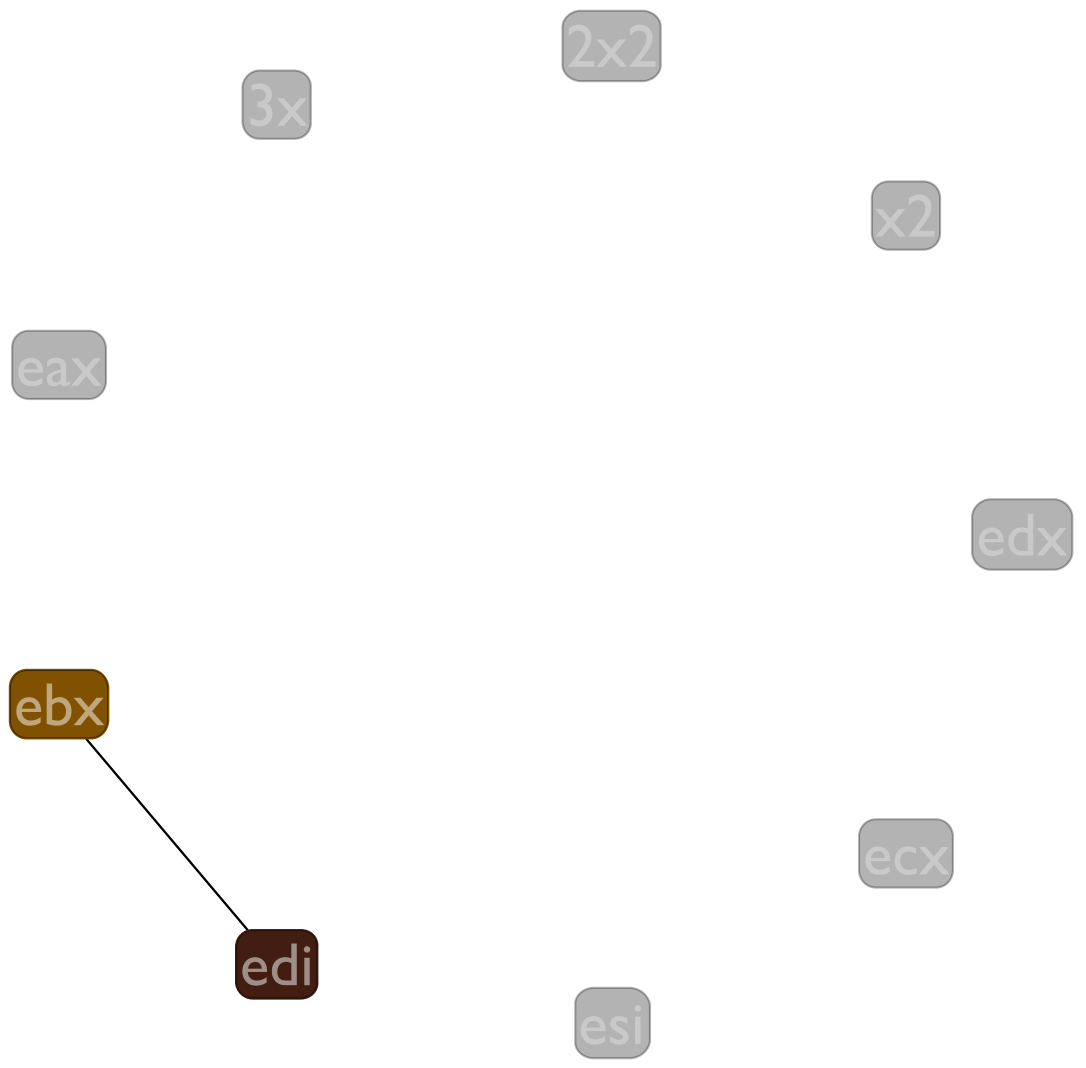

2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
edi
esi
ecx
eax
edx
x2
2x2
3x

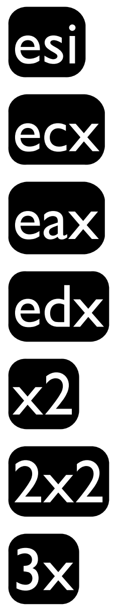

2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
ebx
edi
esi
ecx
eax
edx
x2
2x2
3x

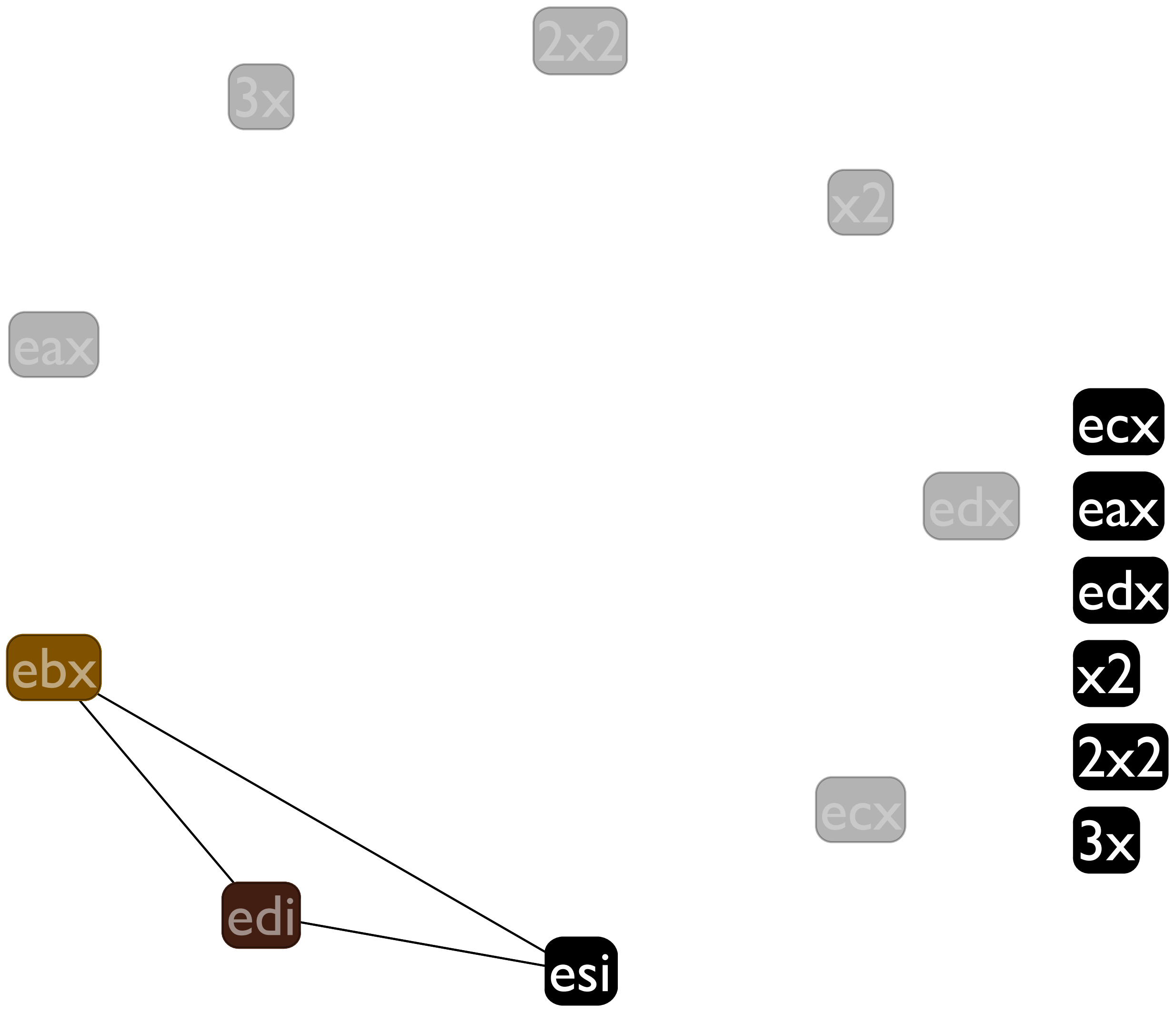

2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
edi
esi
ecx
eax
edx
x2
2x2
3x

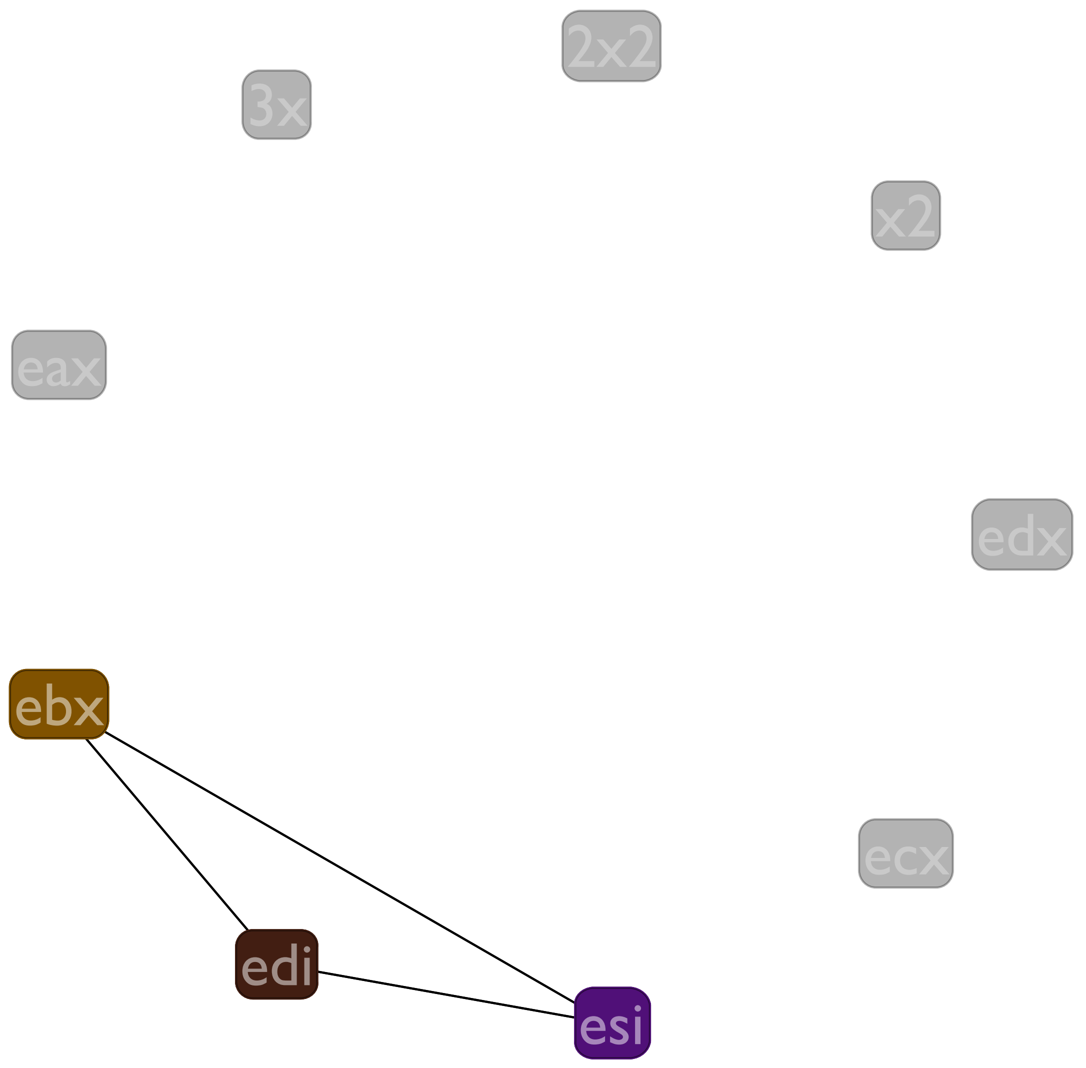

2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
edi
esi
ecx
eax
edx
x2
2x2
3x


2x2
3x
x2
eax
esi
ecx
edx
eax
edx
ebx
x2
2x2
ecx
3x
edi
esi


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi


esi
ecx
eax
edx
x2
2x2
3x


2x2
3x
x2
eax
edx
ecx
ebx
edi
esi
ecx
eax
edx
x2
2x2
3x



ecx

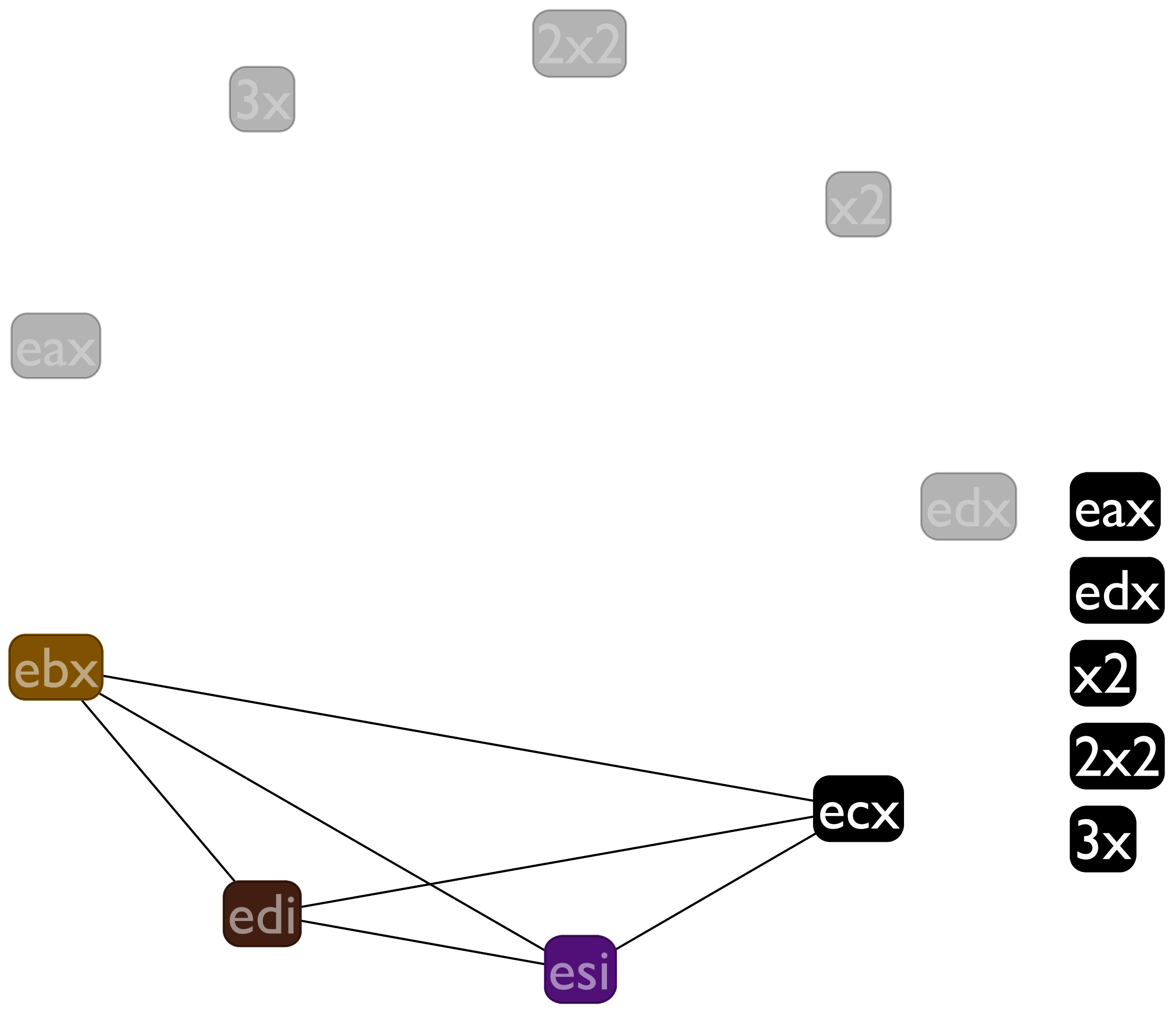
eax

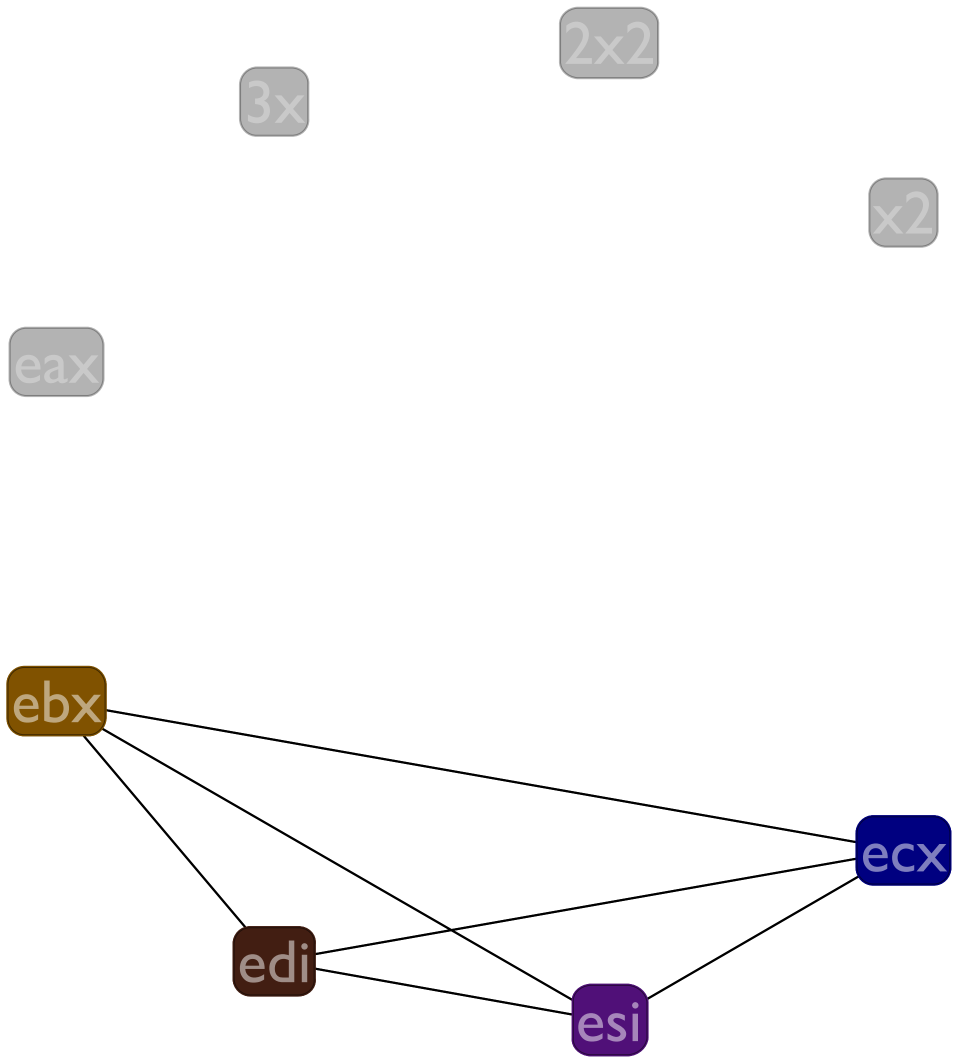
edx

x2

2x2

3x


2x2
3x
x2
eax
edx
eax
edx
x2
2x2
3x
ebx
ecx
edi
esi



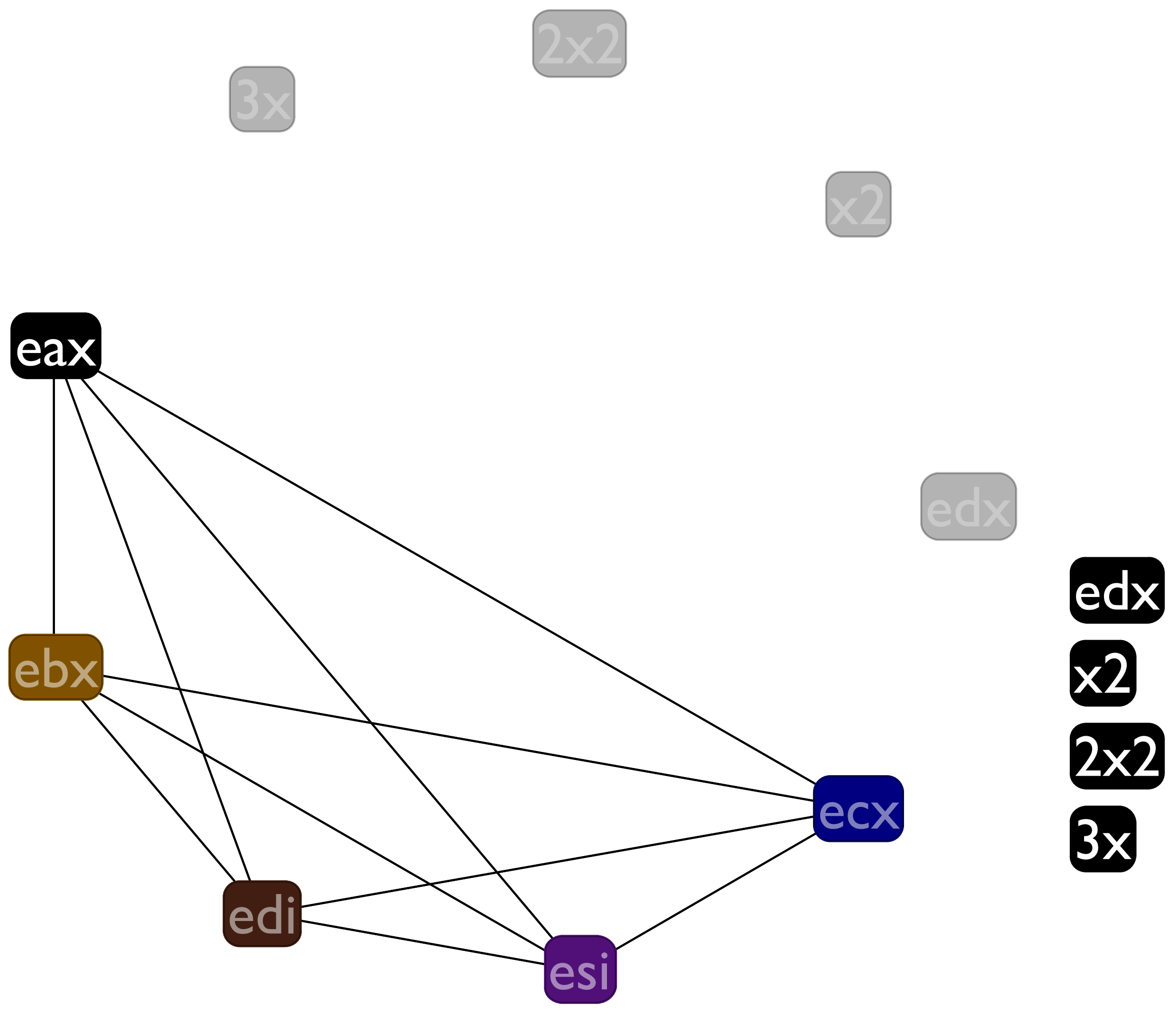
eax

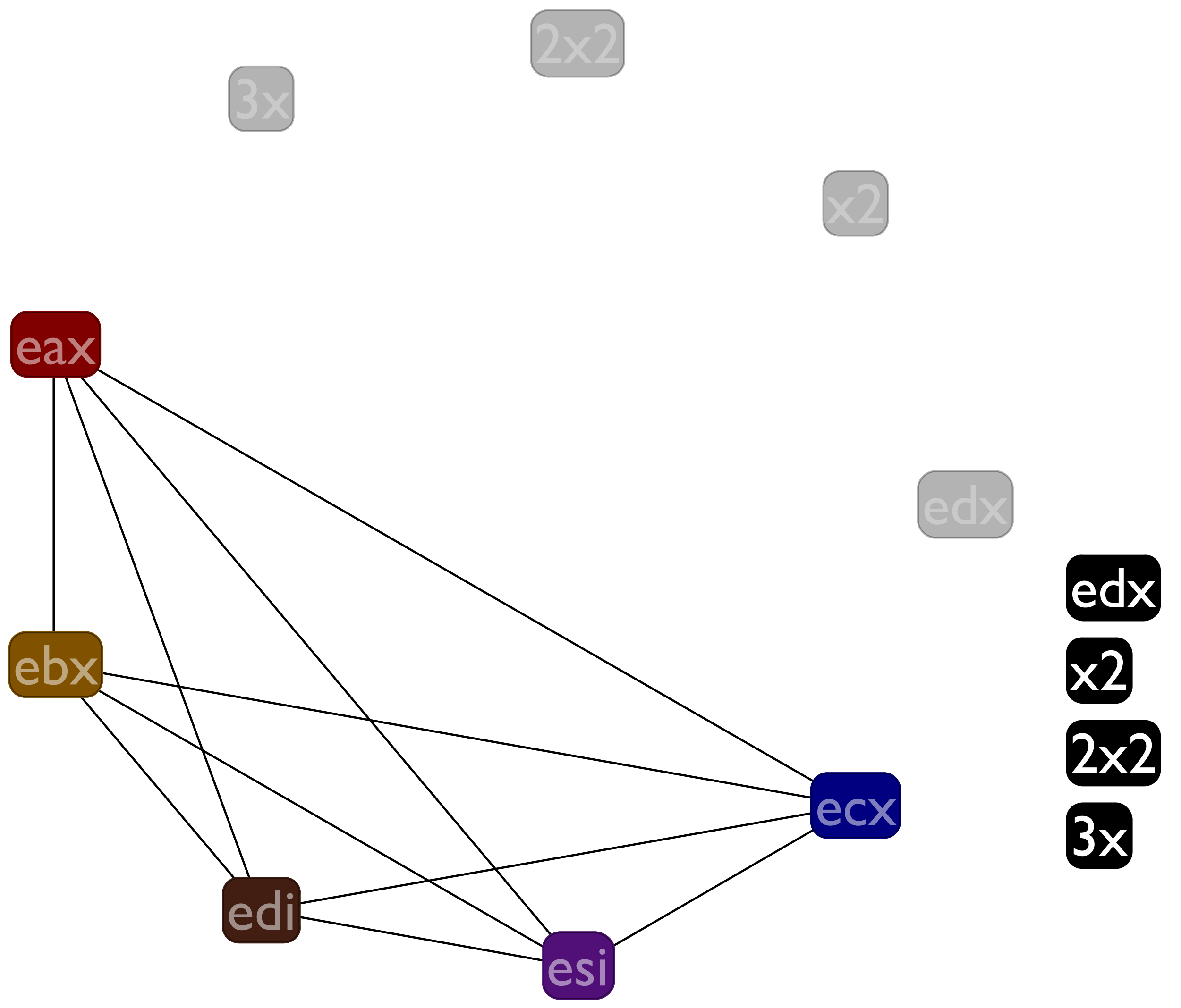
edx

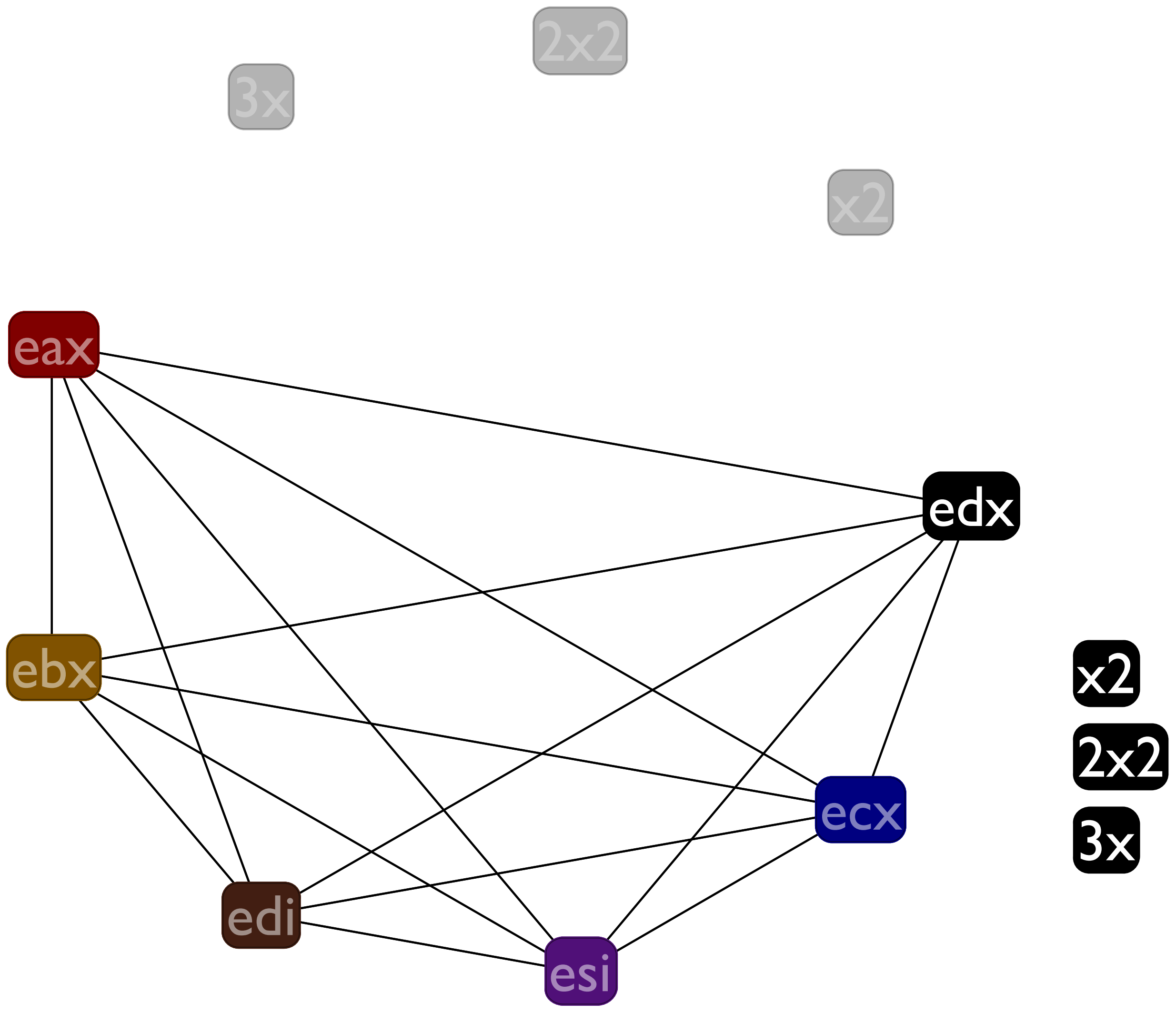
x2

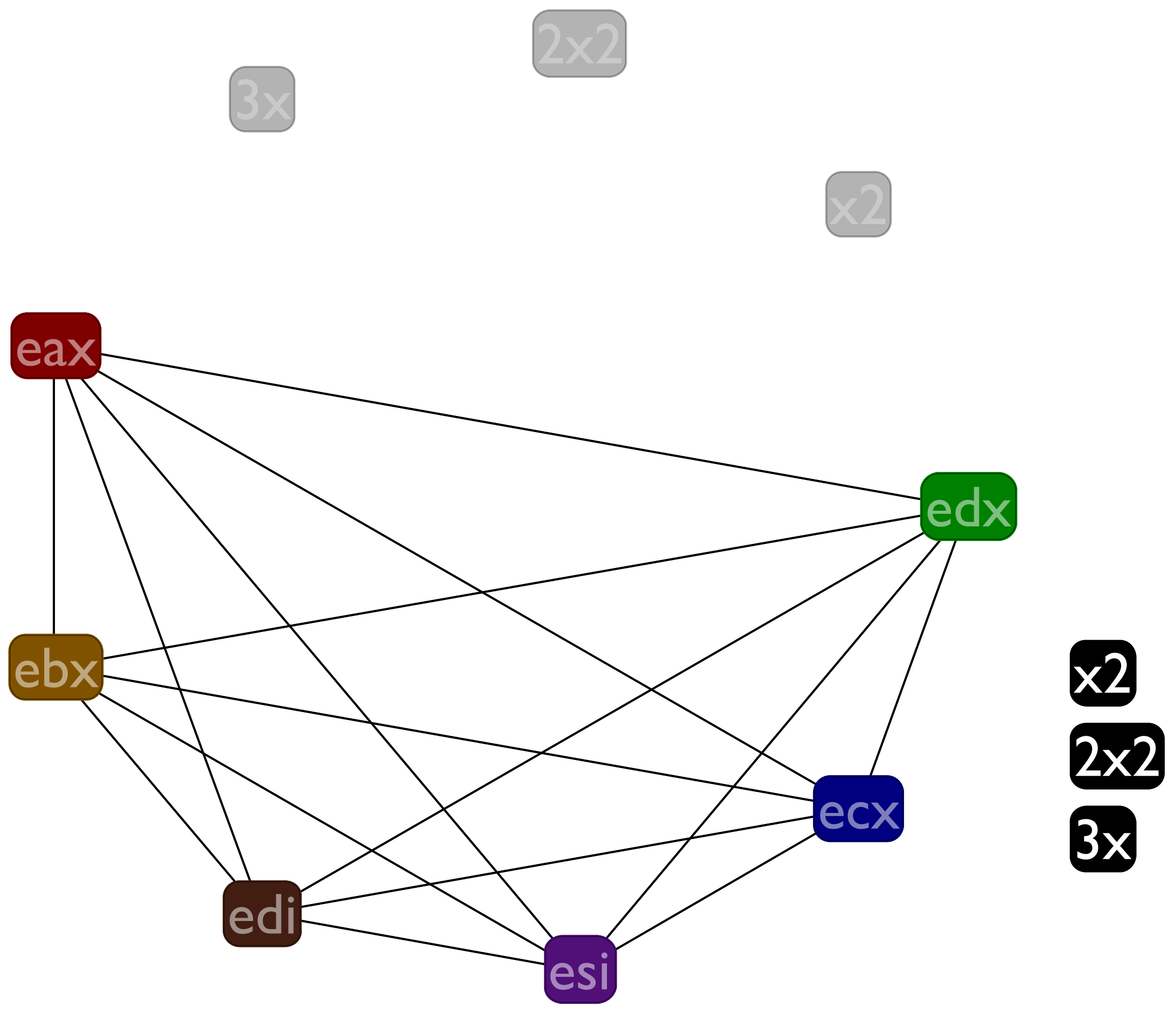
2x2

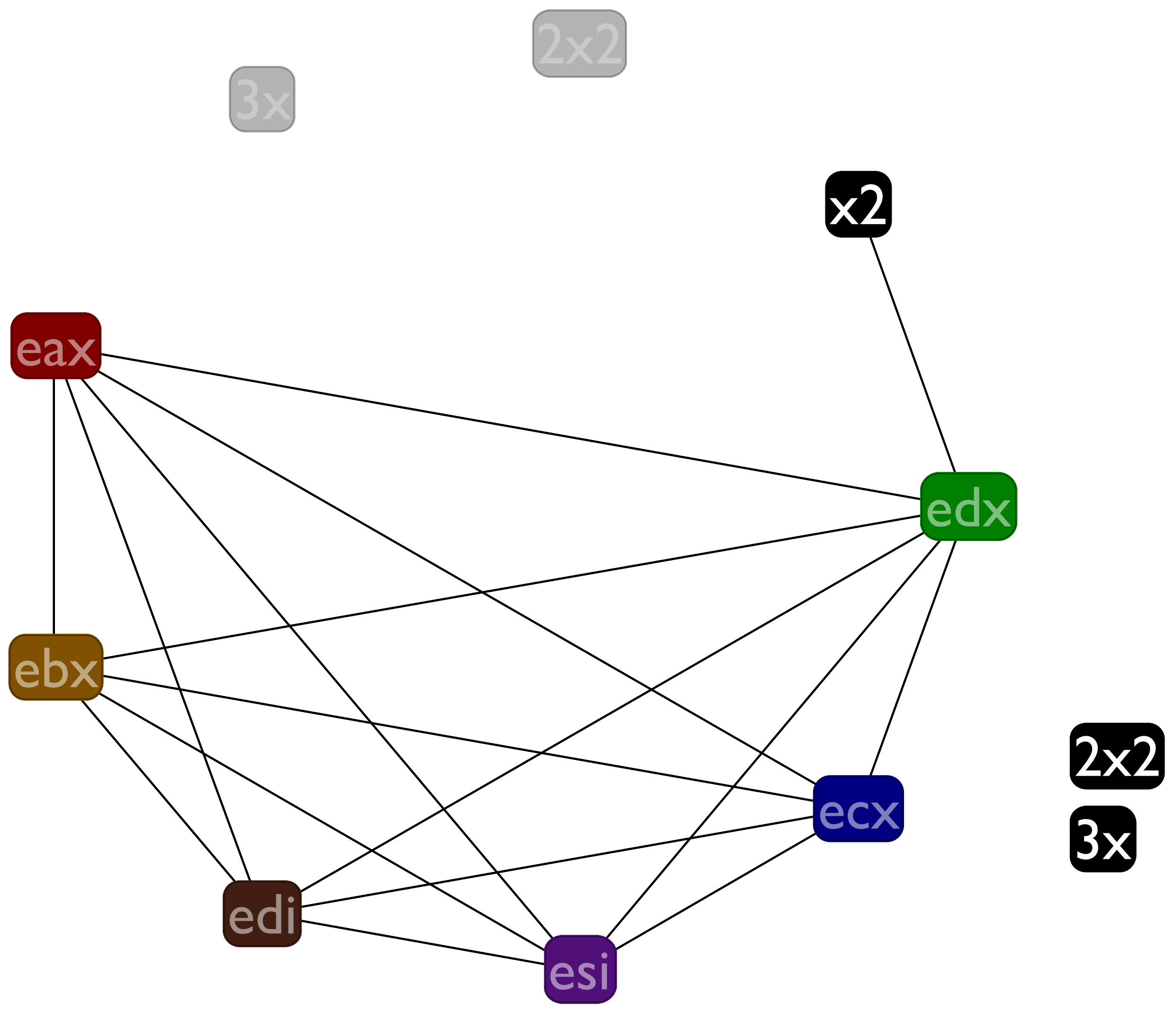
3x

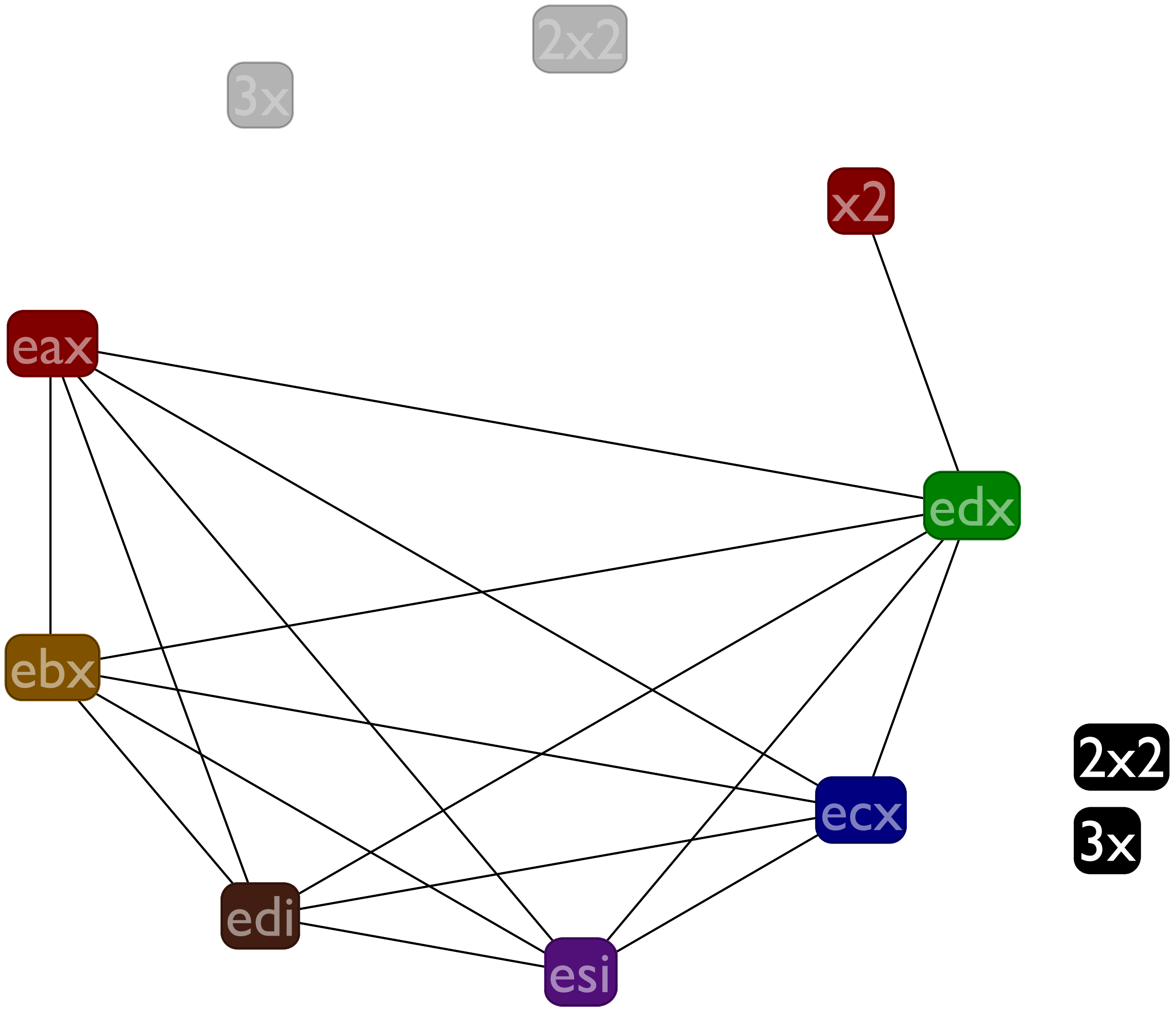

2x2
3x
x2
eax
edx
edx
ebx
x2
2x2
ecx
3x
edi
esi

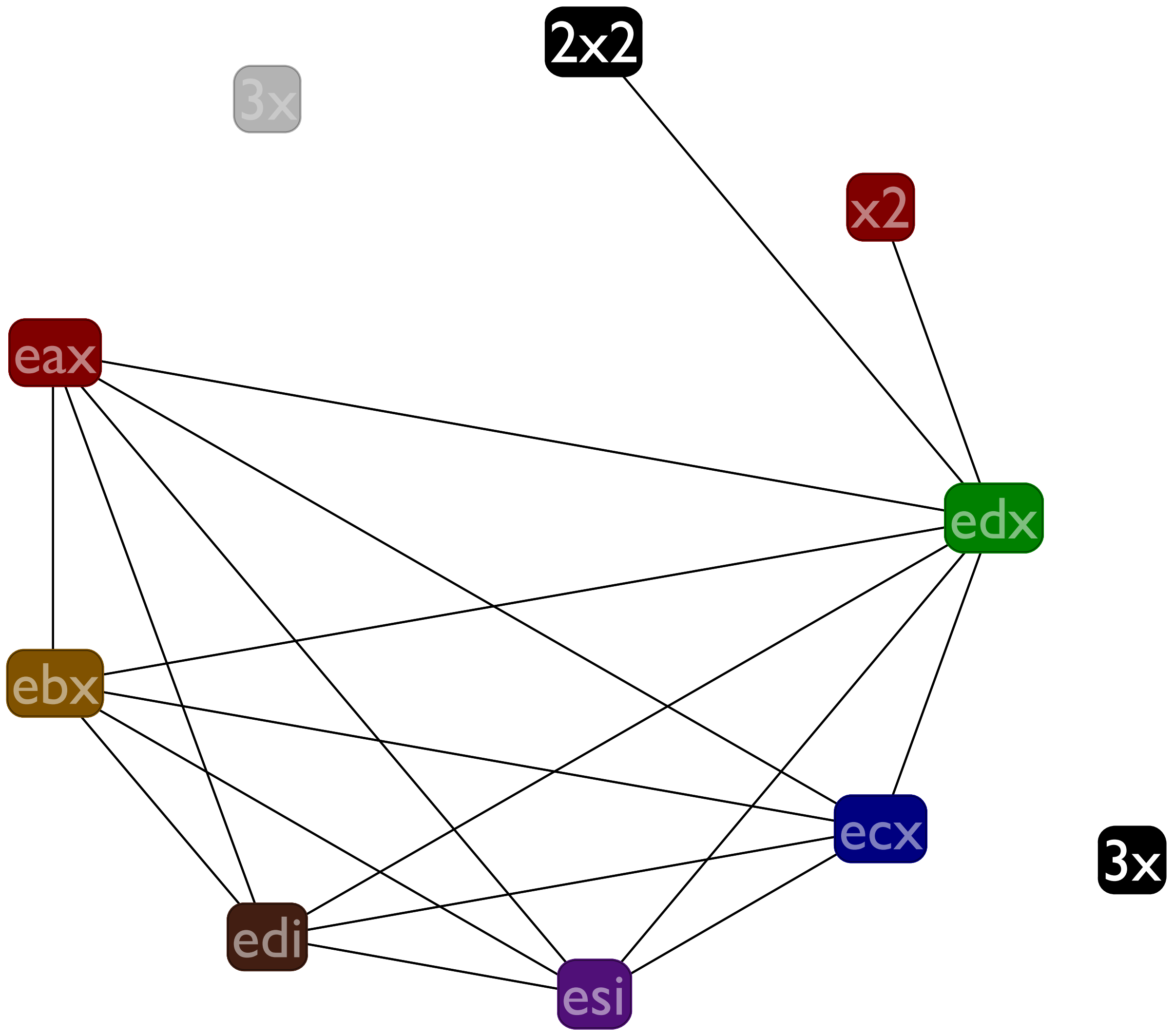

2x2
3x
x2
eax
edx
edx
x2
ebx
2x2
3x
ecx
edi
esi

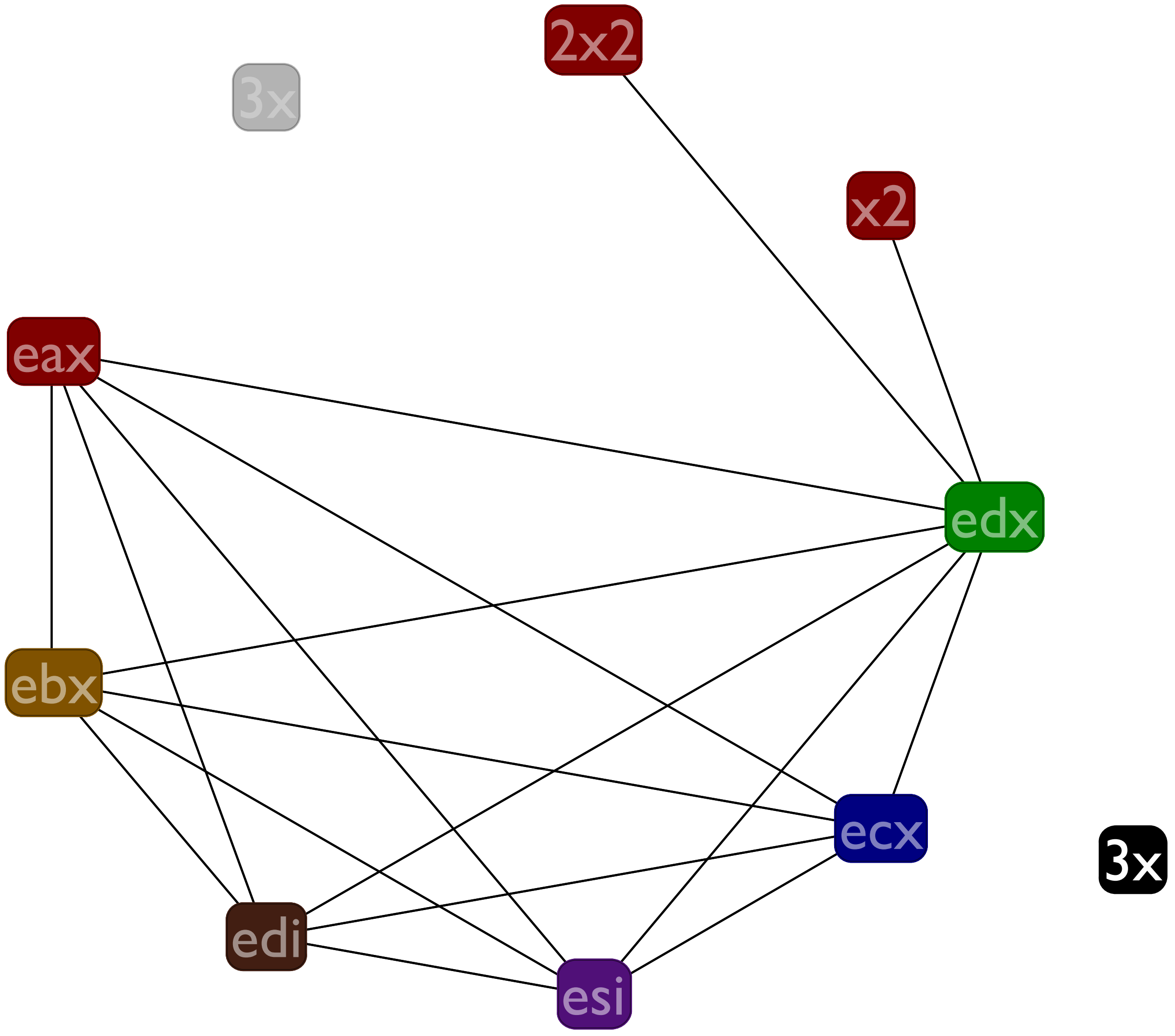

2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
x2
2x2
3x

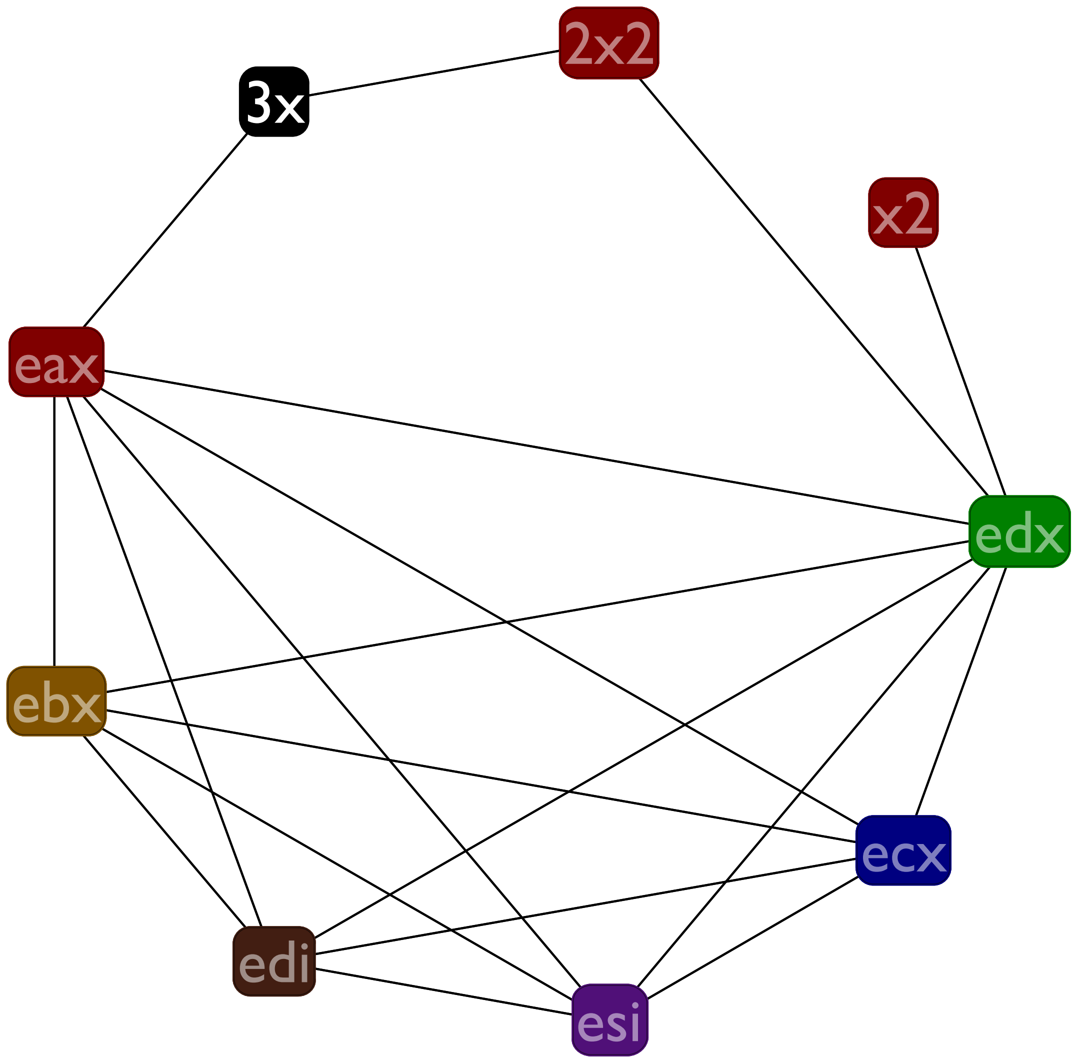

2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
x2
2x2
3x

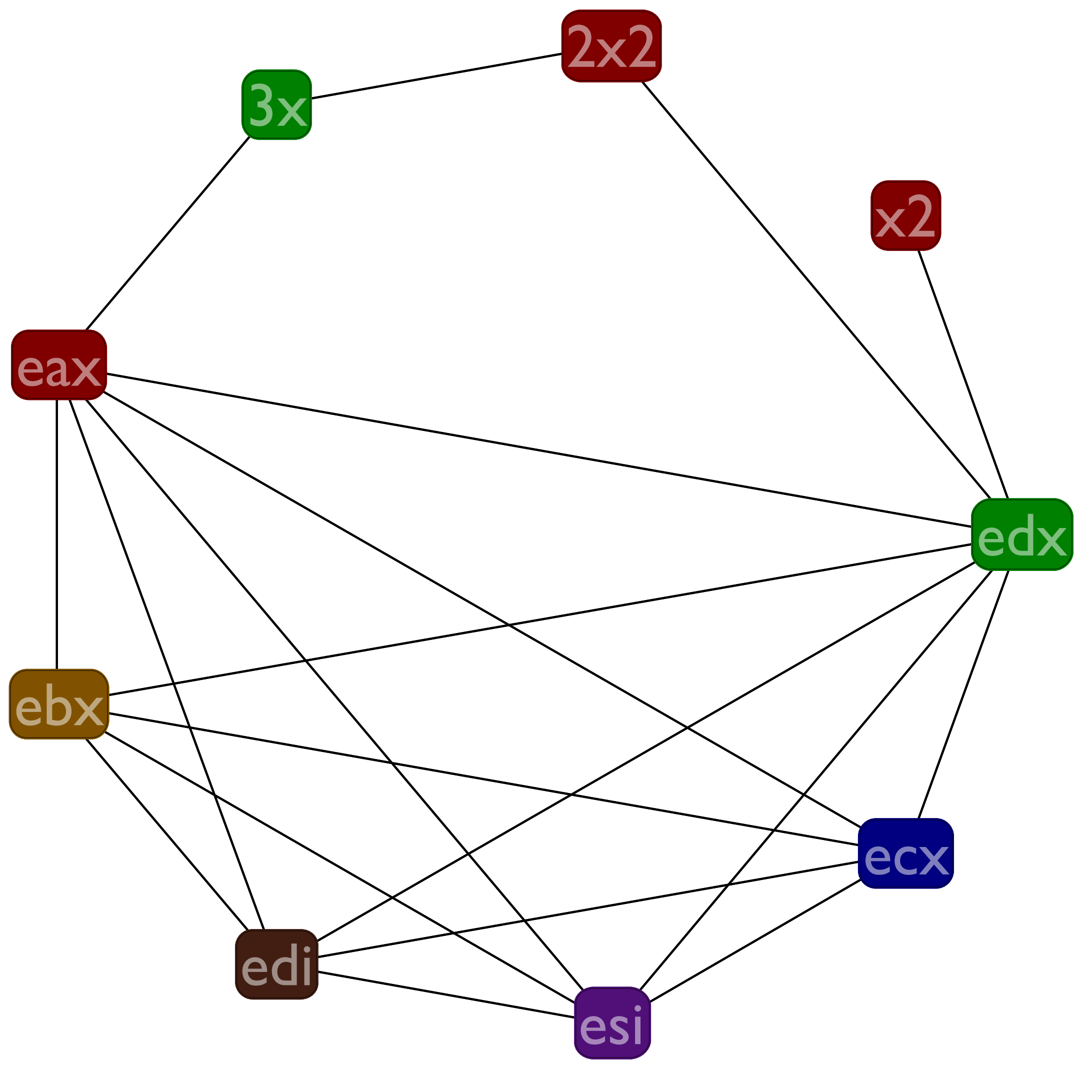

2x2
3x
x2
eax
edx
ebx
ecx
2x2
3x
edi
esi


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
2x2
3x


3x
2x2
x2
eax
edx
ebx
ecx
3x
edi
esi


2x2
3x
x2
eax
edx
ebx
ecx
3x
edi
esi


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi

Graph coloring algorithm

Some of the finer details of the graph coloring algorithm

- If possible, prefer to pull out nodes that have 5 or fewer edges at each stage
- When inserting & coloring nodes, always use the "smallest" color possible (according to any ordering on the colors you want). That is, imagine an ordering on the registers (say, alphabetical) and use the color of the first register in that order that works
- Ignore **`ebp`** and **`esp`** registers when building graph

Caller and callee save registers

But the result of coloring the example graph is wrong, since we didn't account for the callee save registers! Specifically, **`3x`** clobbers **`esi`**, breaking the calling convention.

Caller and callee save registers

- Treat the beginning of the function as an initialization of the callee save registers
- Treat each `(return)` as a reference to the callee save registers
- Treat each `(call)` as an assignment to the caller save registers

Constrained arithmetic operators

The `(cx <- s cmp s)` instruction in L1 is limited to only 4 possible destinations.

The `(x sop= sx)` instruction in L1 is limited to only shifting by a constant or the value of `ecx`.

So we just add interference edges to disallow the illegal registers when we build the interference graph, before starting the coloring.

Do over

Lets redo the coloring, now with the callee and caller save register information in the graph

Gen & Kill

		gen	kill
1:	`:f`	()	()
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2)
5:	`(2x2 *= 2)`	(2x2)	(2x2)
6:	`(3x <- edx)`	(edx)	(3x)
7:	`(3x *= 3)`	(3x)	(3x)
8:	`(eax <- 2x2)`	(2x2)	(eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(eax)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	**out**
1:	`:f`	()	()
2:	`(x2 <- edx)`	()	()
3:	`(x2 *= x2)`	()	()
4:	`(2x2 <- x2)`	()	()
5:	`(2x2 *= 2)`	()	()
6:	`(3x <- edx)`	()	()
7:	`(3x *= 3)`	()	()
8:	`(eax <- 2x2)`	()	()
9:	`(eax += 3x)`	()	()
10:	`(eax += 4)`	()	()
11:	`(return)`	()	()

Liveness

		in	out
1:	`:f`	()	()
2:	`(x2 <- edx)`	(edx)	()
3:	`(x2 *= x2)`	(x2)	()
4:	`(2x2 <- x2)`	(x2)	()
5:	`(2x2 *= 2)`	(2x2)	()
6:	`(3x <- edx)`	(edx)	()
7:	`(3x *= 3)`	(3x)	()
8:	`(eax <- 2x2)`	(2x2)	()
9:	`(eax += 3x)`	(3x eax)	()
10:	`(eax += 4)`	(eax)	()
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	()	(edx)
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2)
5:	`(2x2 *= 2)`	(2x2)	(edx)
6:	`(3x <- edx)`	(edx)	(3x)
7:	`(3x *= 3)`	(3x)	(2x2)
8:	`(eax <- 2x2)`	(2x2)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2)
5:	`(2x2 *= 2)`	(2x2 edx)	(edx)
6:	`(3x <- edx)`	(edx)	(3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(edx)
6:	`(3x <- edx)`	(edx)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(x2)	(x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(edx)
6:	`(3x <- edx)`	(2x2 edx)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(x2)	(edx x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(2x2 edx)
6:	`(3x <- edx)`	(2x2 edx)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	**out**
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(x2)
3:	`(x2 *= x2)`	(edx x2)	(edx x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(2x2 edx)
6:	`(3x <- edx)`	(2x2 edx)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(edx x2)
3:	`(x2 *= x2)`	(edx x2)	(edx x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(2x2 edx)
6:	`(3x <- edx)`	(2x2 edx)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(edx x2)
3:	`(x2 *= x2)`	(edx x2)	(edx x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(2x2 edx)
6:	`(3x <- edx)`	(2x2 edx)	(2x2 3x)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(edx x2)
3:	`(x2 *= x2)`	(edx x2)	(edx x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(2x2 edx)
6:	`(3x <- edx)`	(2x2 edx)	(2x2 3x ebx edi esi)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(edx x2)
3:	`(x2 *= x2)`	(edx x2)	(edx x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(2x2 edx)
6:	`(3x <- edx)`	(2x2 ebx edi edx esi)	(2x2 3x ebx edi esi)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(edx x2)
3:	`(x2 *= x2)`	(edx x2)	(edx x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 edx)	(2x2 ebx edi edx esi)
6:	`(3x <- edx)`	(2x2 ebx edi edx esi)	(2x2 3x ebx edi esi)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(edx x2)
3:	`(x2 *= x2)`	(edx x2)	(edx x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 edx)
5:	`(2x2 *= 2)`	(2x2 ebx edi edx esi)	(2x2 ebx edi edx esi)
6:	`(3x <- edx)`	(2x2 ebx edi edx esi)	(2x2 3x ebx edi esi)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(edx x2)
3:	`(x2 *= x2)`	(edx x2)	(edx x2)
4:	`(2x2 <- x2)`	(edx x2)	(2x2 ebx edi edx esi)
5:	`(2x2 *= 2)`	(2x2 ebx edi edx esi)	(2x2 ebx edi edx esi)
6:	`(3x <- edx)`	(2x2 ebx edi edx esi)	(2x2 3x ebx edi esi)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(edx x2)
3:	`(x2 *= x2)`	(edx x2)	(edx x2)
4:	`(2x2 <- x2)`	(ebx edi edx esi x2)	(2x2 ebx edi edx esi)
5:	`(2x2 *= 2)`	(2x2 ebx edi edx esi)	(2x2 ebx edi edx esi)
6:	`(3x <- edx)`	(2x2 ebx edi edx esi)	(2x2 3x ebx edi esi)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(edx x2)
3:	`(x2 *= x2)`	(edx x2)	(ebx edi edx esi x2)
4:	`(2x2 <- x2)`	(ebx edi edx esi x2)	(2x2 ebx edi edx esi)
5:	`(2x2 *= 2)`	(2x2 ebx edi edx esi)	(2x2 ebx edi edx esi)
6:	`(3x <- edx)`	(2x2 ebx edi edx esi)	(2x2 3x ebx edi esi)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(edx x2)
3:	`(x2 *= x2)`	(ebx edi edx esi x2)	(ebx edi edx esi x2)
4:	`(2x2 <- x2)`	(ebx edi edx esi x2)	(2x2 ebx edi edx esi)
5:	`(2x2 *= 2)`	(2x2 ebx edi edx esi)	(2x2 ebx edi edx esi)
6:	`(3x <- edx)`	(2x2 ebx edi edx esi)	(2x2 3x ebx edi esi)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(edx)	(ebx edi edx esi x2)
3:	`(x2 *= x2)`	(ebx edi edx esi x2)	(ebx edi edx esi x2)
4:	`(2x2 <- x2)`	(ebx edi edx esi x2)	(2x2 ebx edi edx esi)
5:	`(2x2 *= 2)`	(2x2 ebx edi edx esi)	(2x2 ebx edi edx esi)
6:	`(3x <- edx)`	(2x2 ebx edi edx esi)	(2x2 3x ebx edi esi)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

Liveness

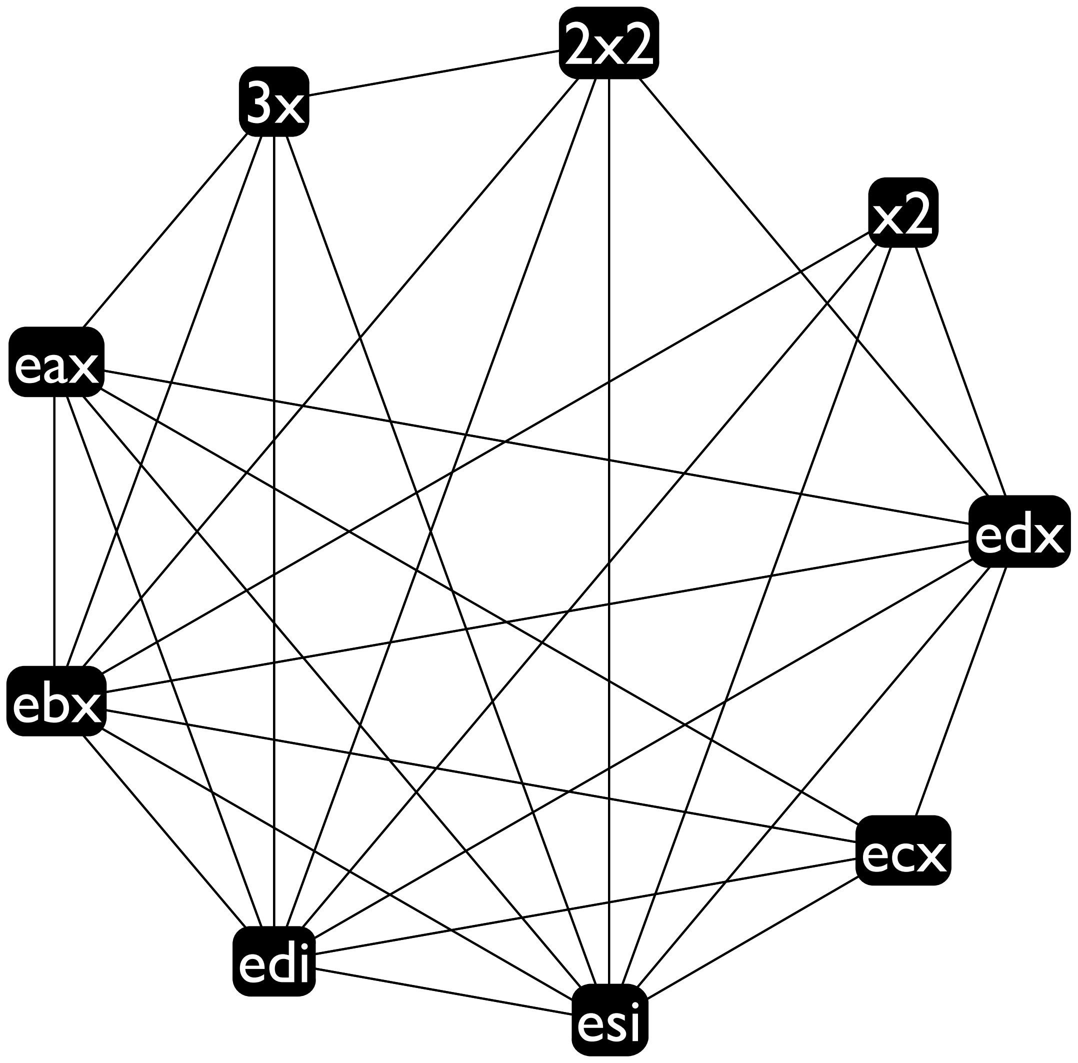
		in	out
1:	`:f`	(edx)	(edx)
2:	`(x2 <- edx)`	(ebx edi edx esi)	(ebx edi edx esi x2)
3:	`(x2 *= x2)`	(ebx edi edx esi x2)	(ebx edi edx esi x2)
4:	`(2x2 <- x2)`	(ebx edi edx esi x2)	(2x2 ebx edi edx esi)
5:	`(2x2 *= 2)`	(2x2 ebx edi edx esi)	(2x2 ebx edi edx esi)
6:	`(3x <- edx)`	(2x2 ebx edi edx esi)	(2x2 3x ebx edi esi)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

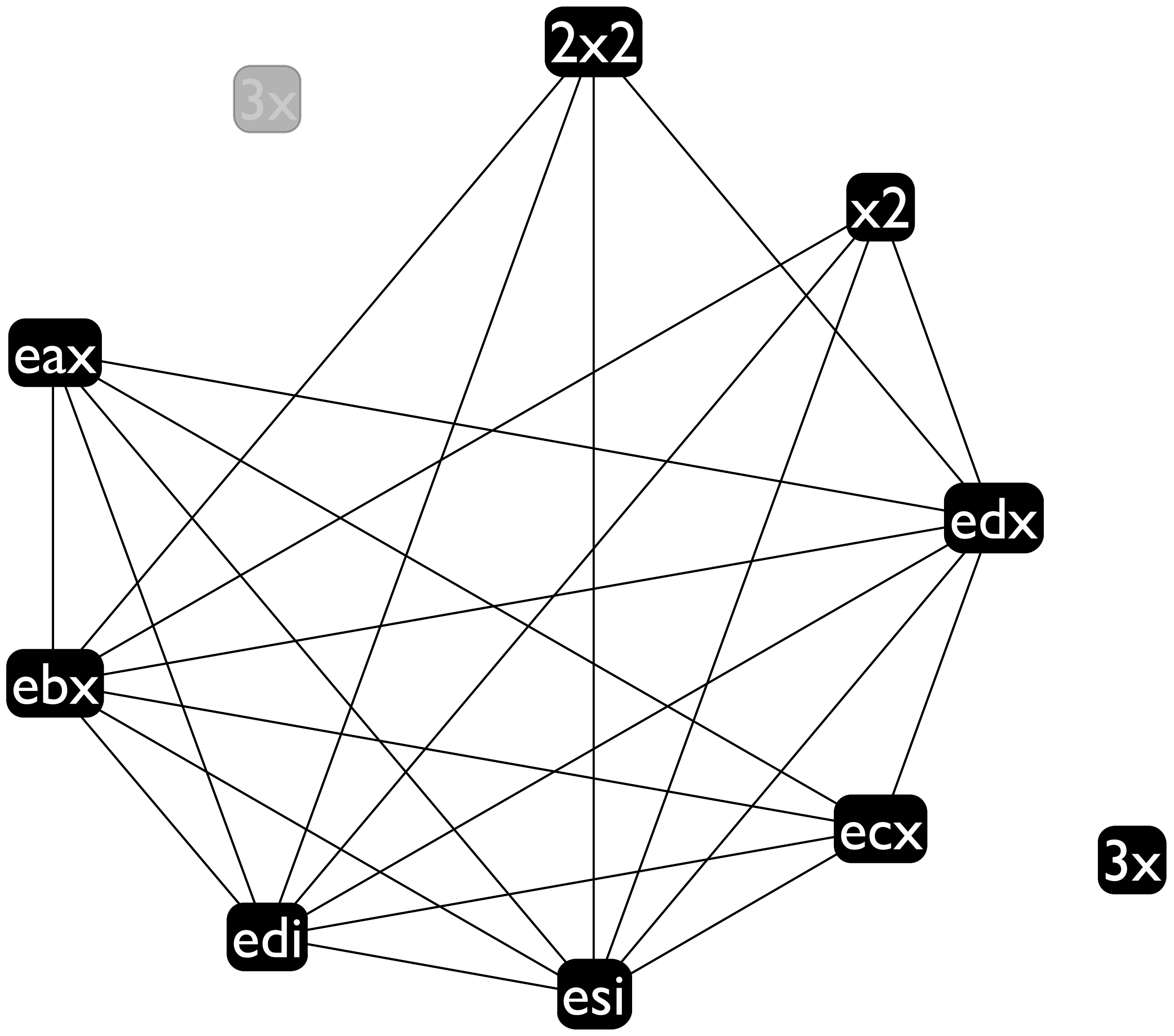
Liveness

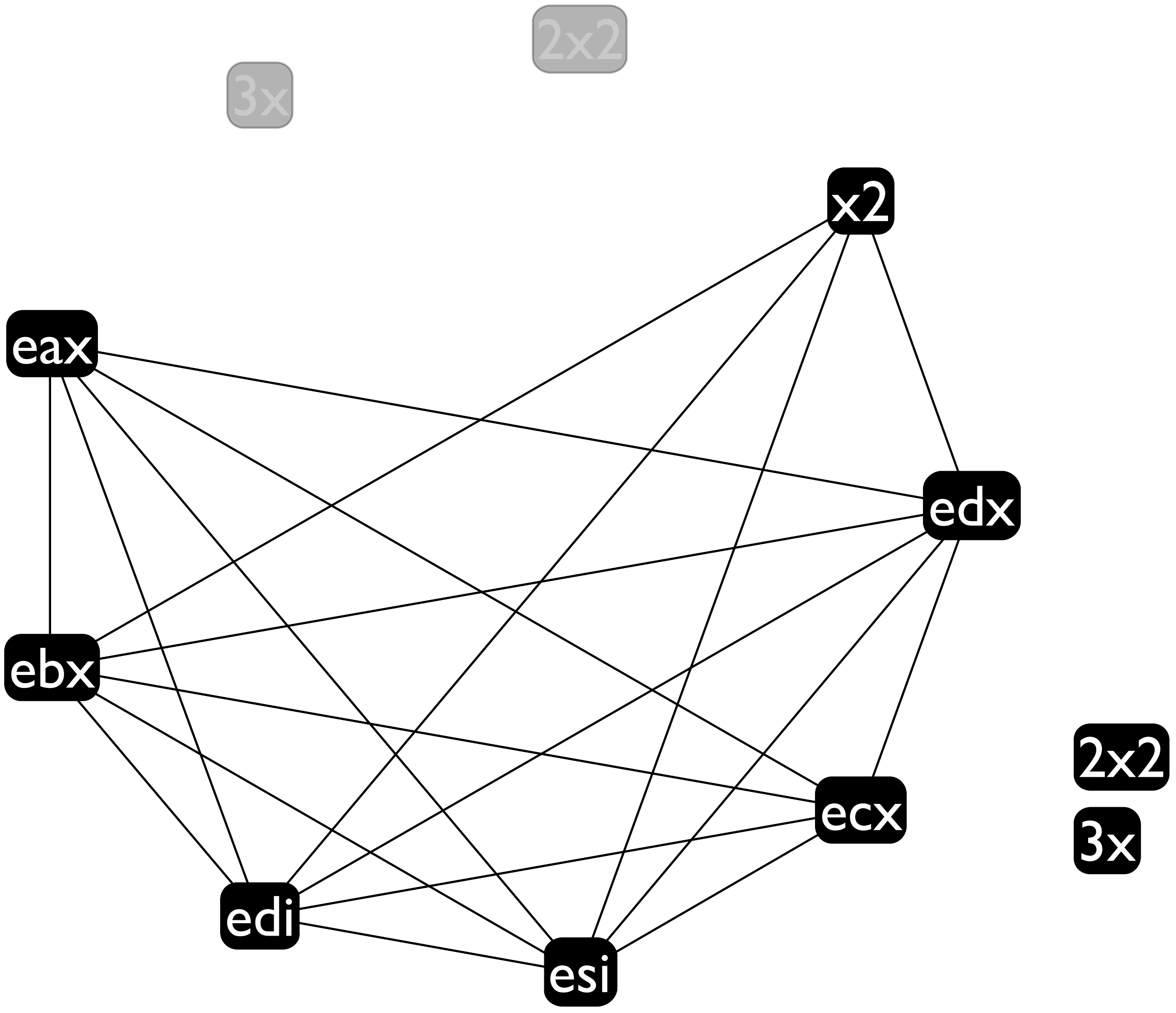
		in	out
1:	`:f`	(edx)	(ebx edi edx esi)
2:	`(x2 <- edx)`	(ebx edi edx esi)	(ebx edi edx esi x2)
3:	`(x2 *= x2)`	(ebx edi edx esi x2)	(ebx edi edx esi x2)
4:	`(2x2 <- x2)`	(ebx edi edx esi x2)	(2x2 ebx edi edx esi)
5:	`(2x2 *= 2)`	(2x2 ebx edi edx esi)	(2x2 ebx edi edx esi)
6:	`(3x <- edx)`	(2x2 ebx edi edx esi)	(2x2 3x ebx edi esi)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

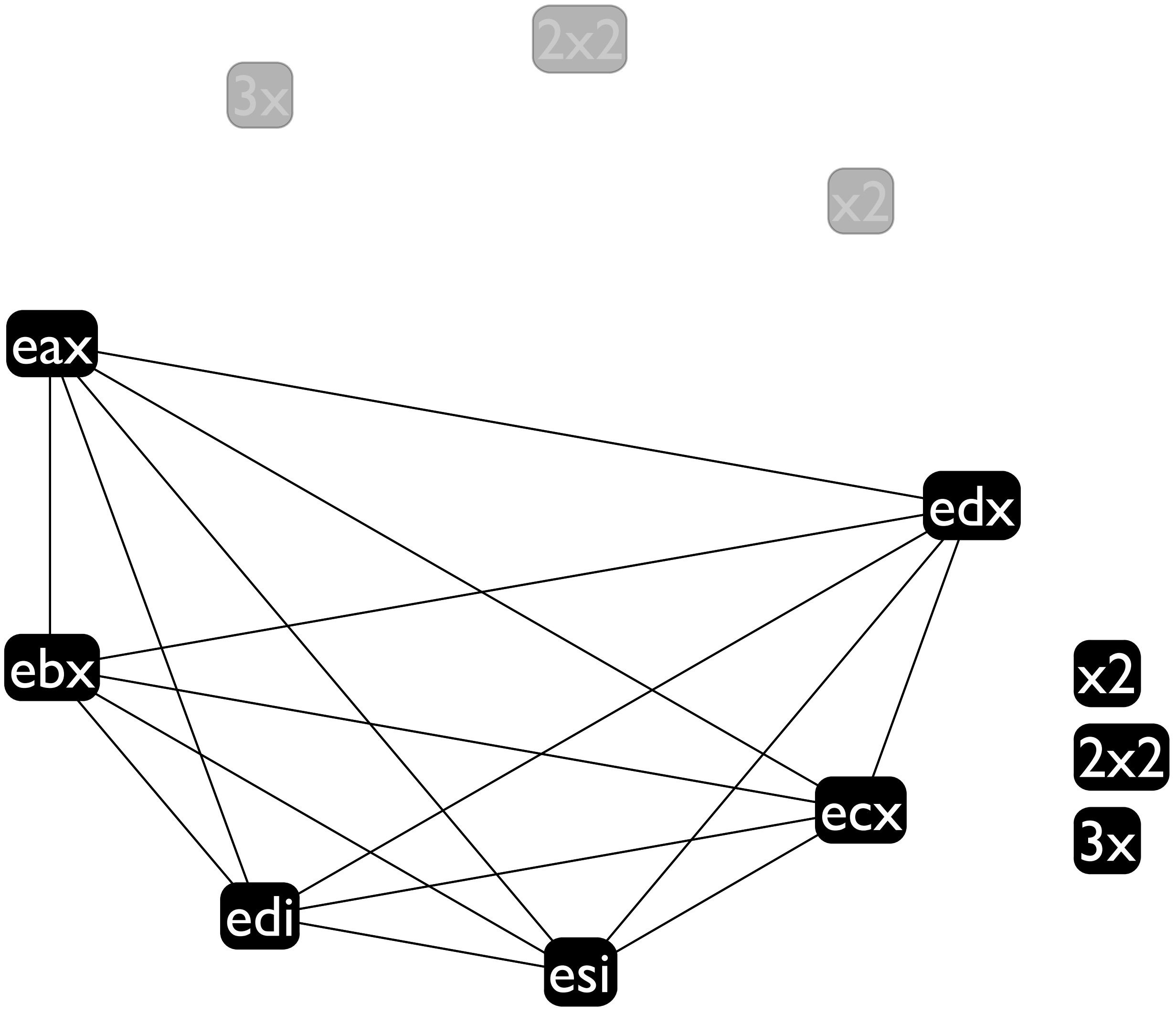
Liveness

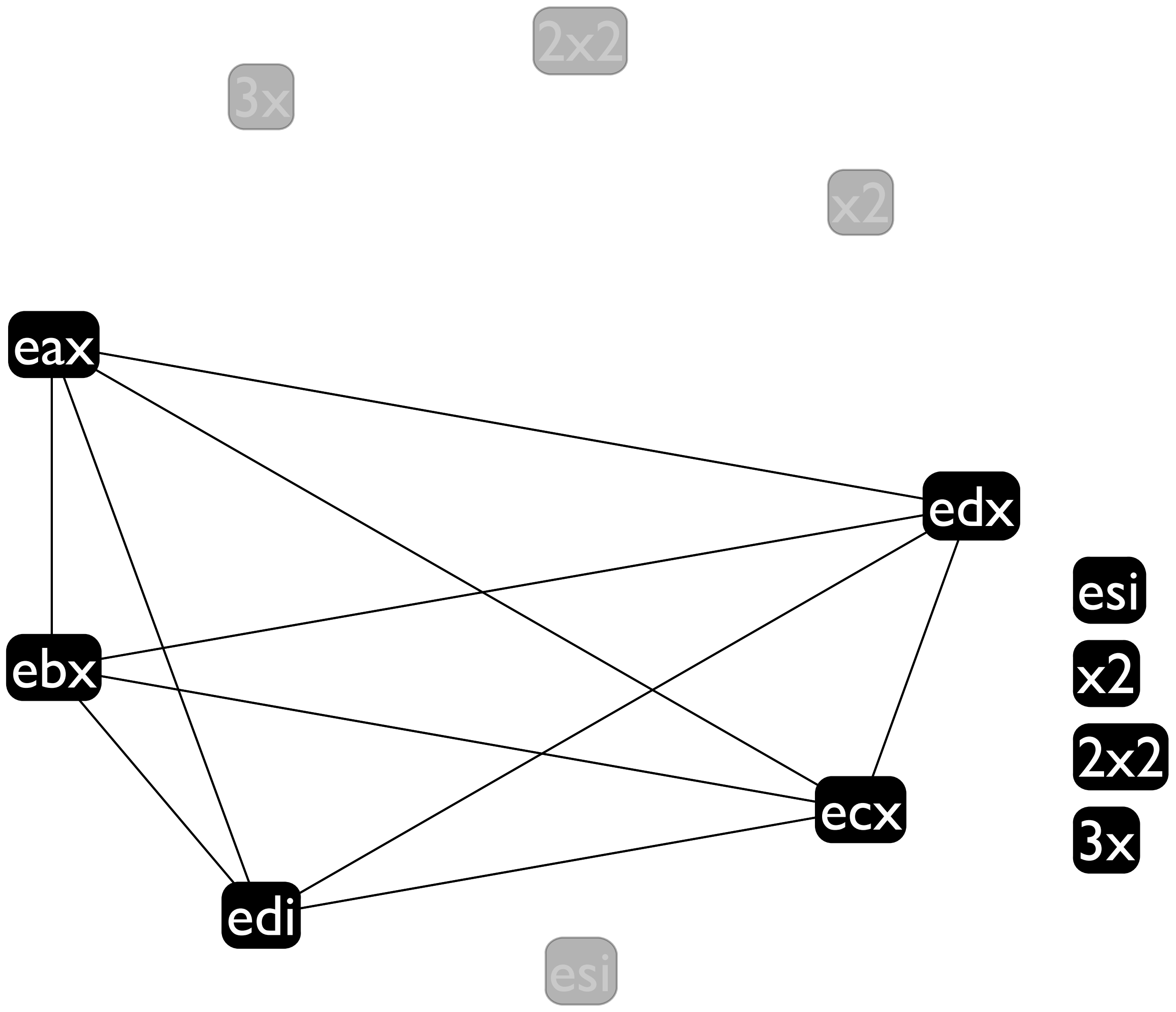
		in	out
1:	`:f`	(ebx edi edx esi)	(ebx edi edx esi)
2:	`(x2 <- edx)`	(ebx edi edx esi)	(ebx edi edx esi x2)
3:	`(x2 *= x2)`	(ebx edi edx esi x2)	(ebx edi edx esi x2)
4:	`(2x2 <- x2)`	(ebx edi edx esi x2)	(2x2 ebx edi edx esi)
5:	`(2x2 *= 2)`	(2x2 ebx edi edx esi)	(2x2 ebx edi edx esi)
6:	`(3x <- edx)`	(2x2 ebx edi edx esi)	(2x2 3x ebx edi esi)
7:	`(3x *= 3)`	(2x2 3x ebx edi esi)	(2x2 3x ebx edi esi)
8:	`(eax <- 2x2)`	(2x2 3x ebx edi esi)	(3x eax ebx edi esi)
9:	`(eax += 3x)`	(3x eax ebx edi esi)	(eax ebx edi esi)
10:	`(eax += 4)`	(eax ebx edi esi)	(ebx edi esi)
11:	`(return)`	(ebx edi esi)	()

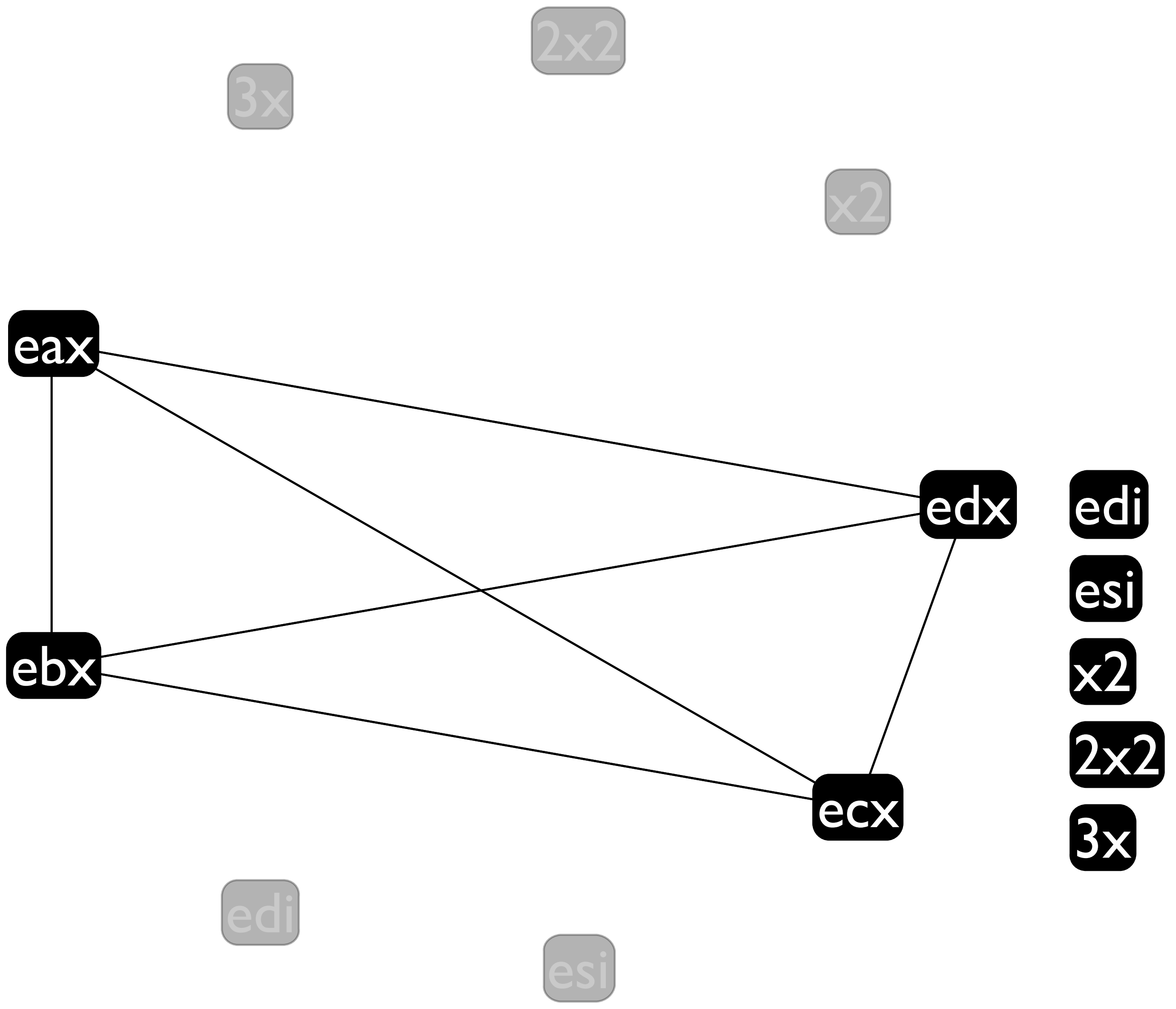

2x2
3x
x2
eax
edx
ebx
ecx
edi
esi

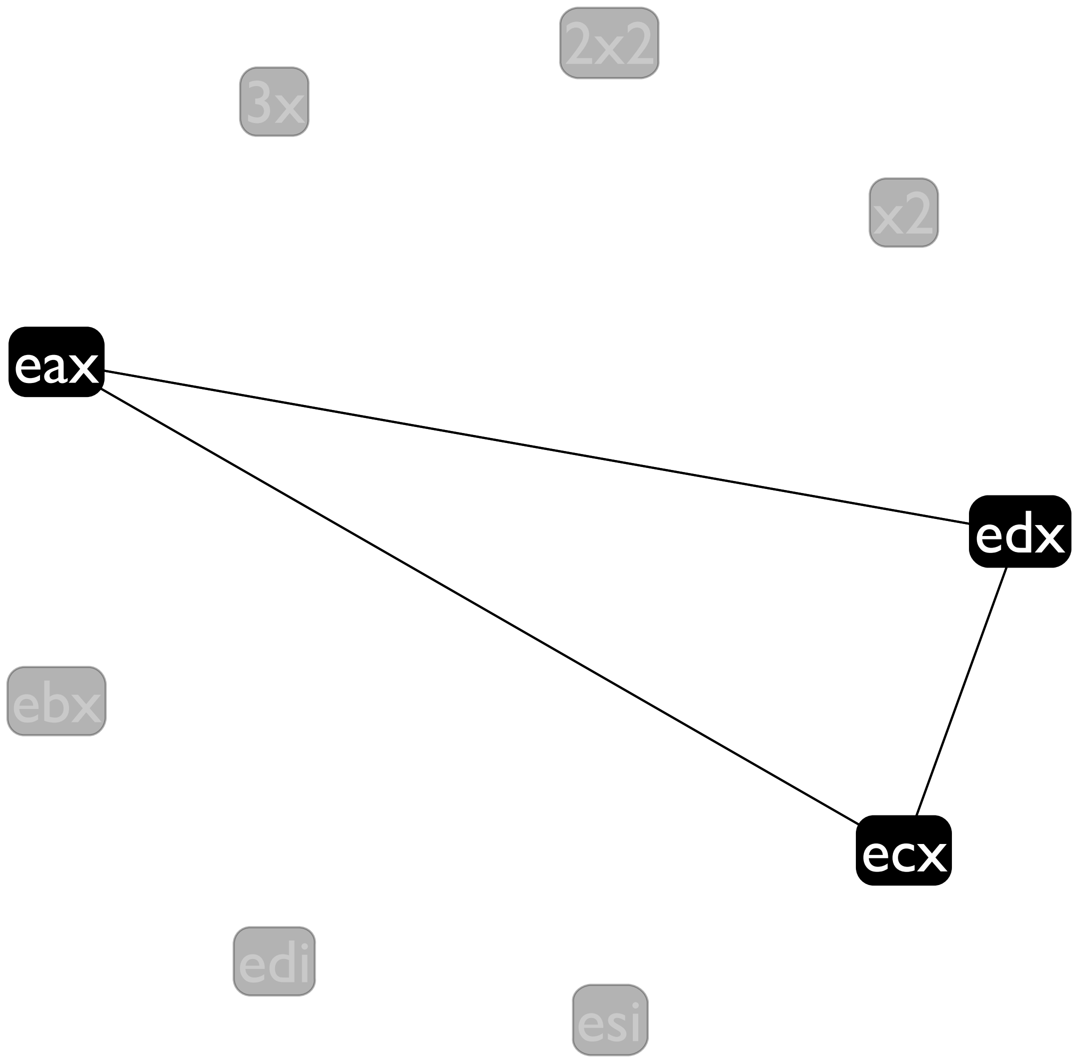

3x
2x2
x2
eax
edx
ebx
ecx
3x
edi
esi


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
2x2
3x

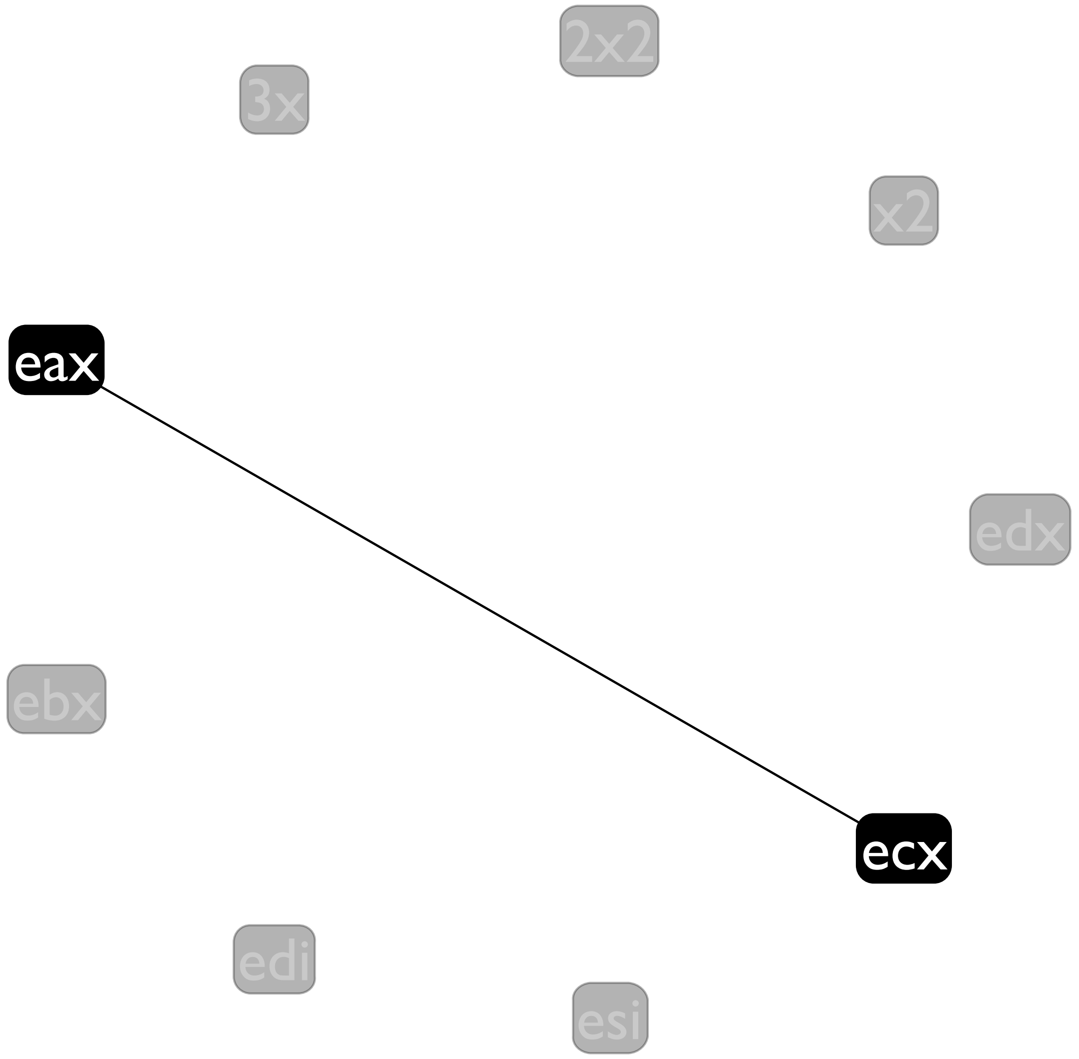

2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
x2
2x2
3x


2x2
3x
x2
eax
edx
esi
ebx
x2
2x2
ecx
3x
edi
esi


2x2
3x
x2
eax
edx
edi
esi
x2
2x2
3x
ebx
ecx
edi
esi



ebx
edi
esi
x2
2x2
3x



edx

ebx

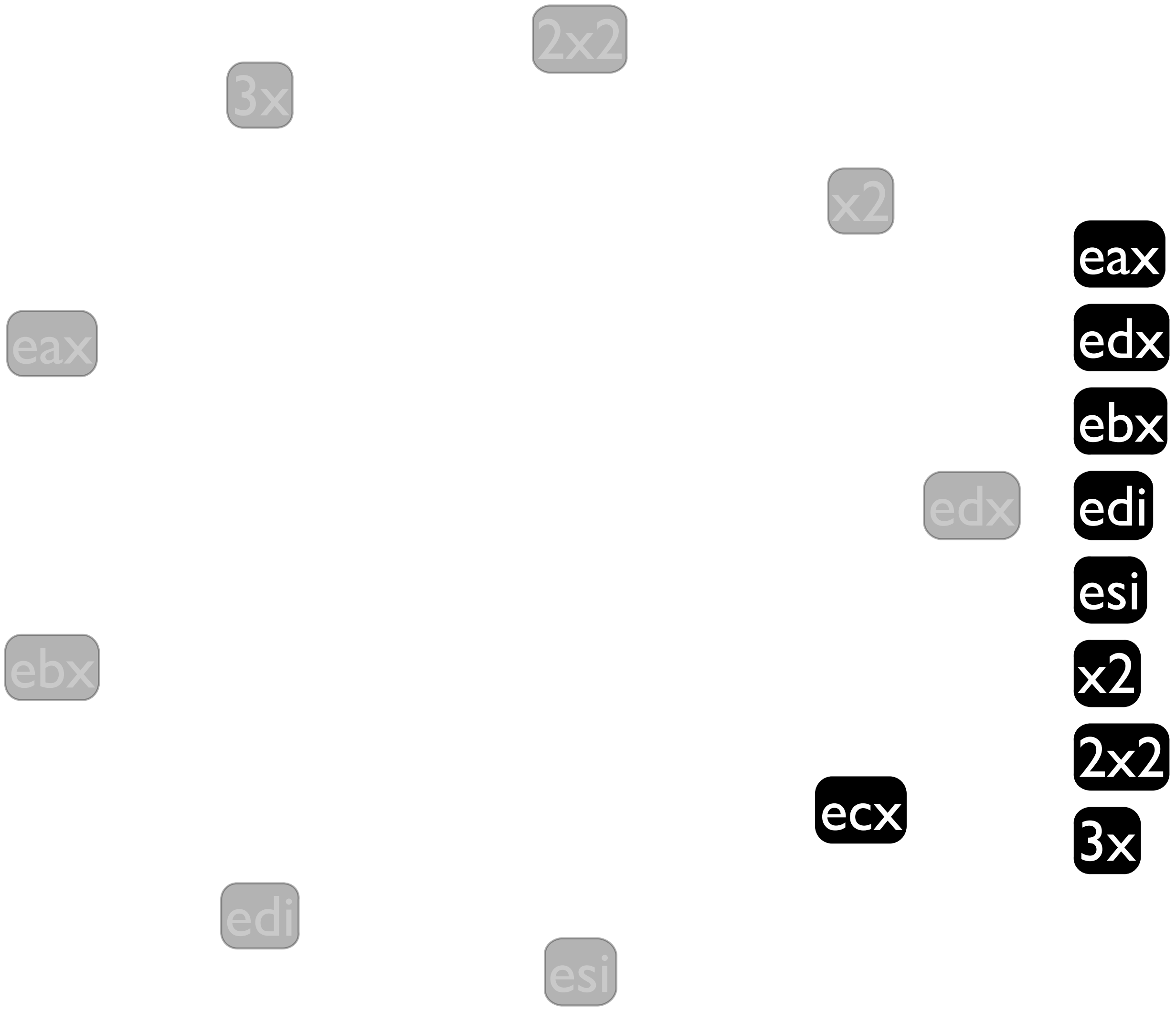
edi

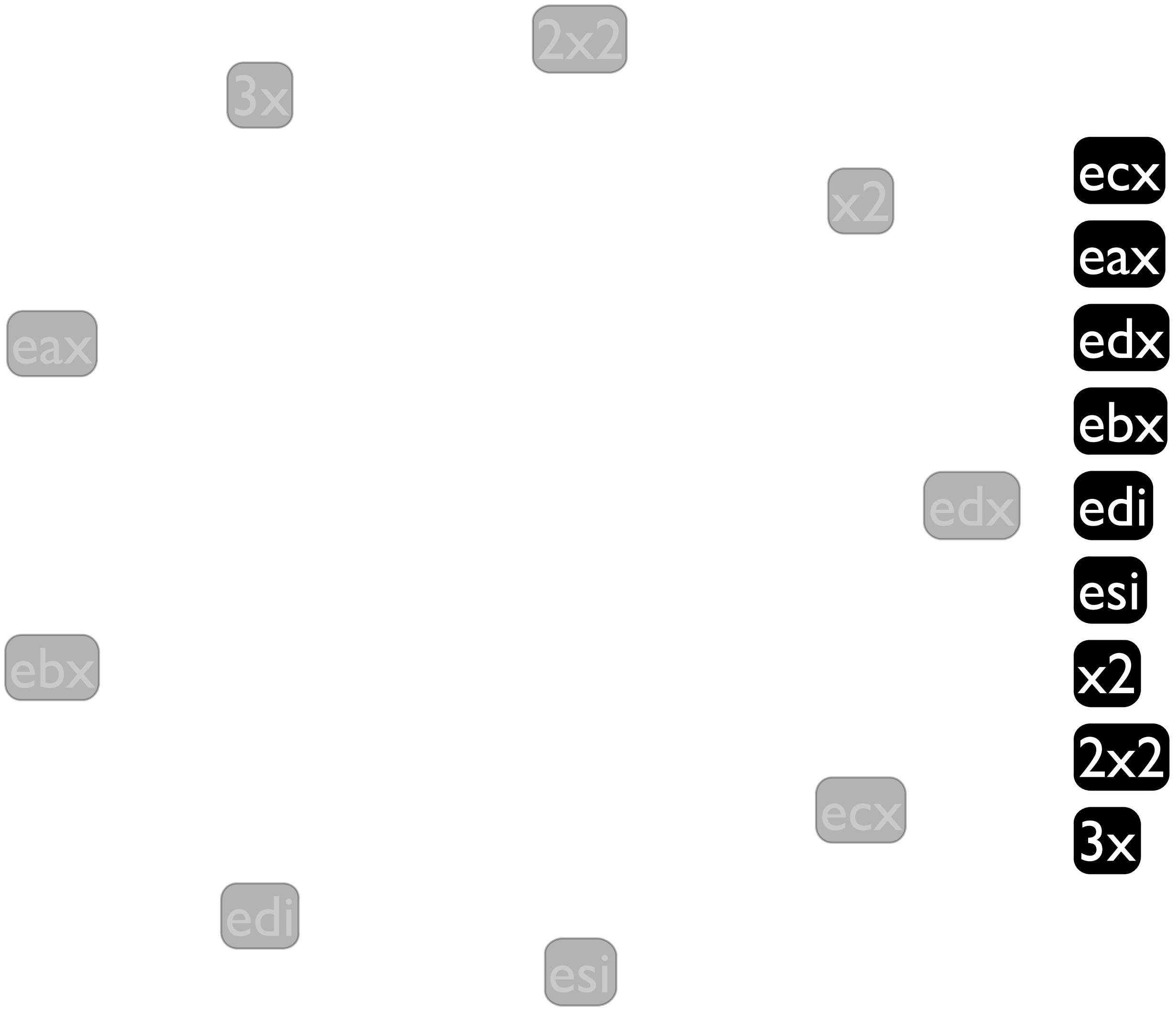
esi

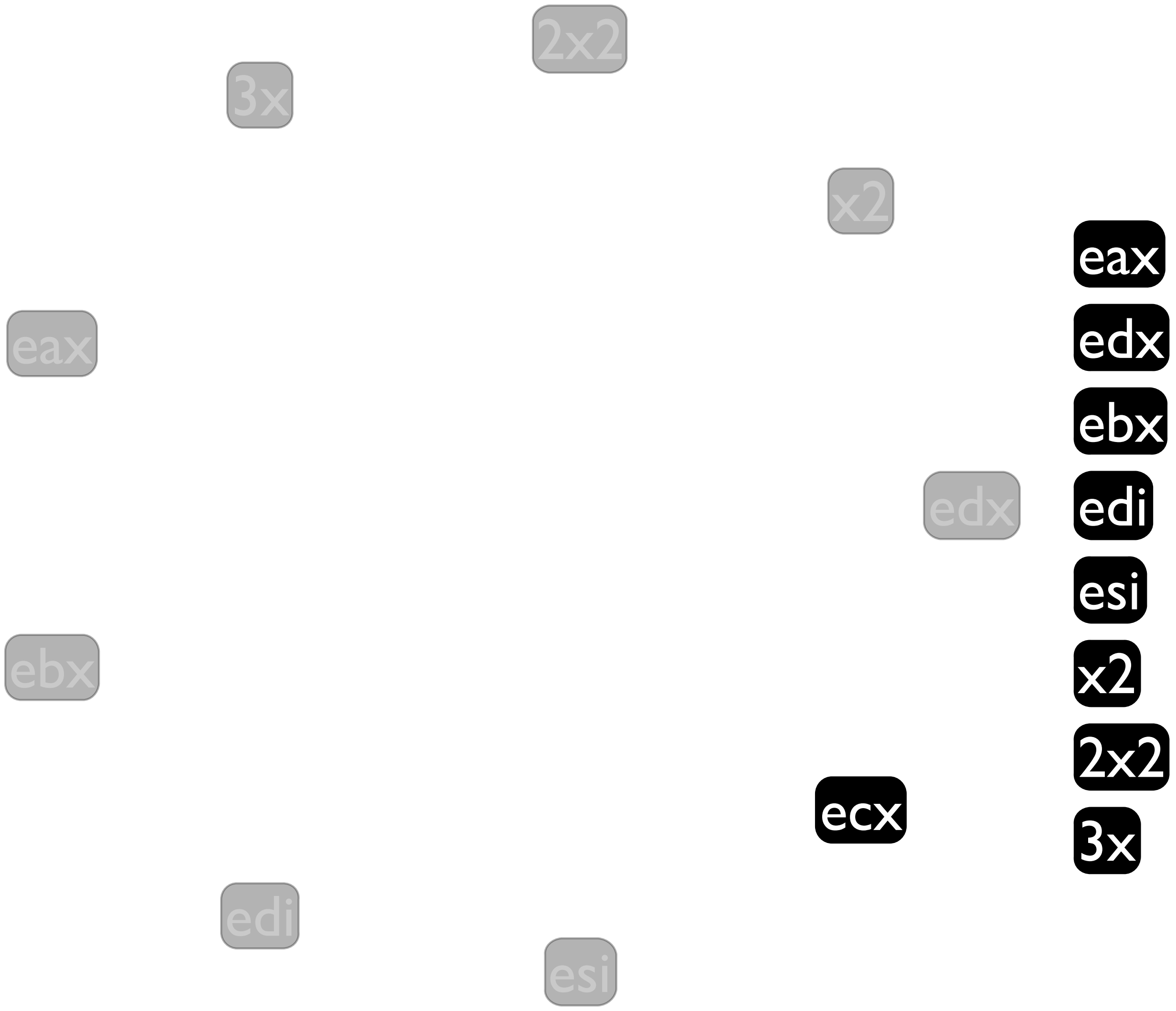
x2

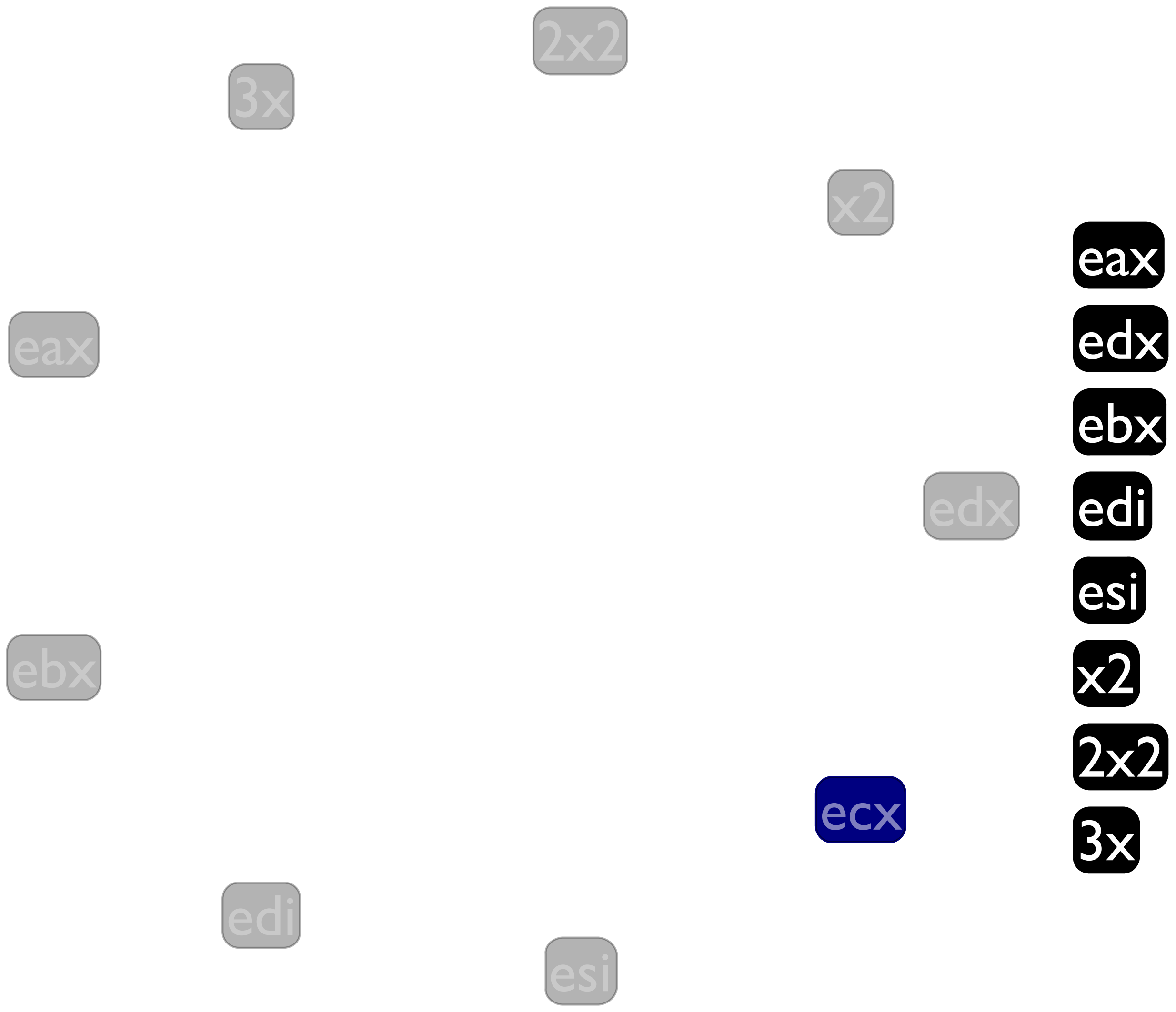
2x2

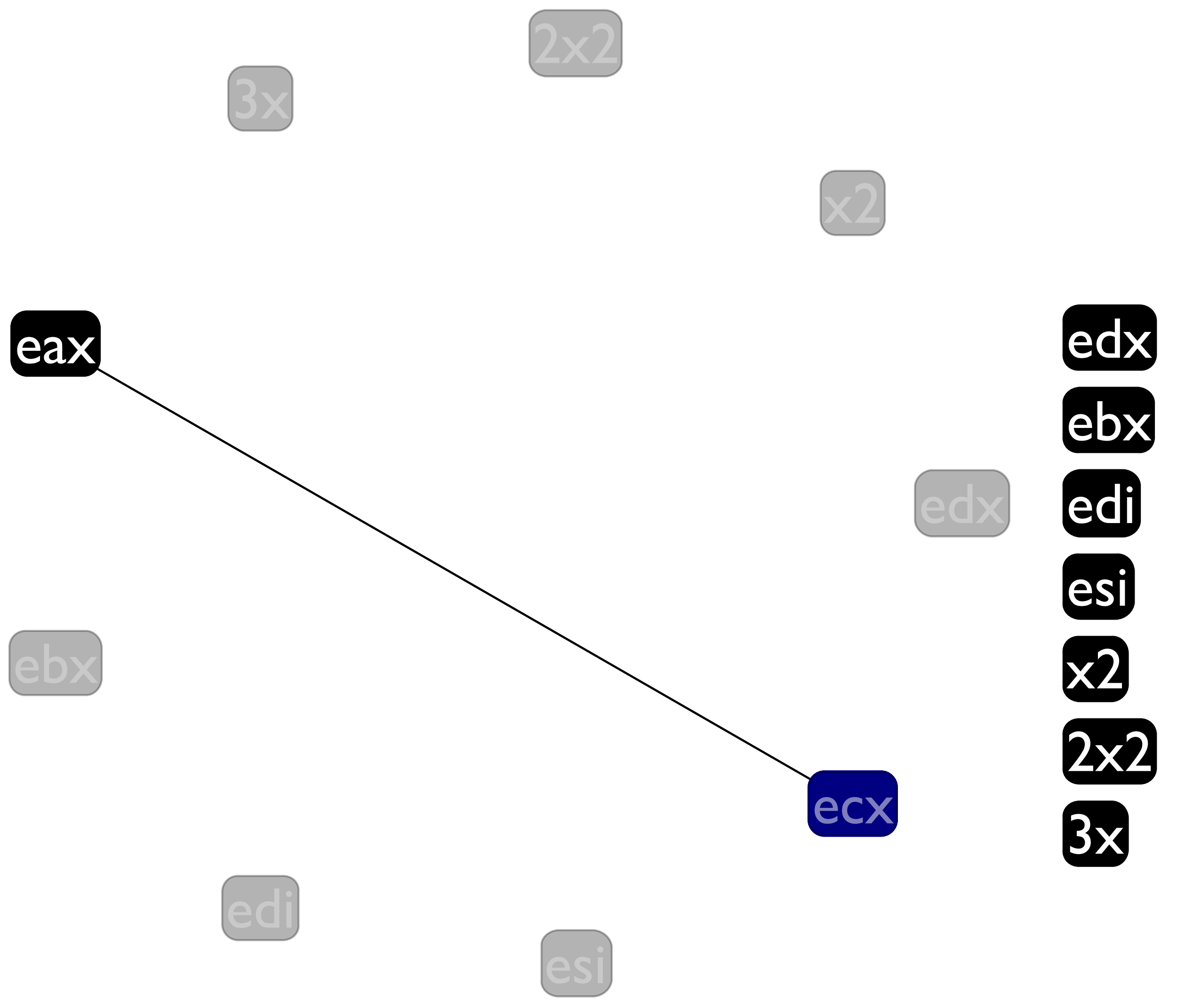
3x

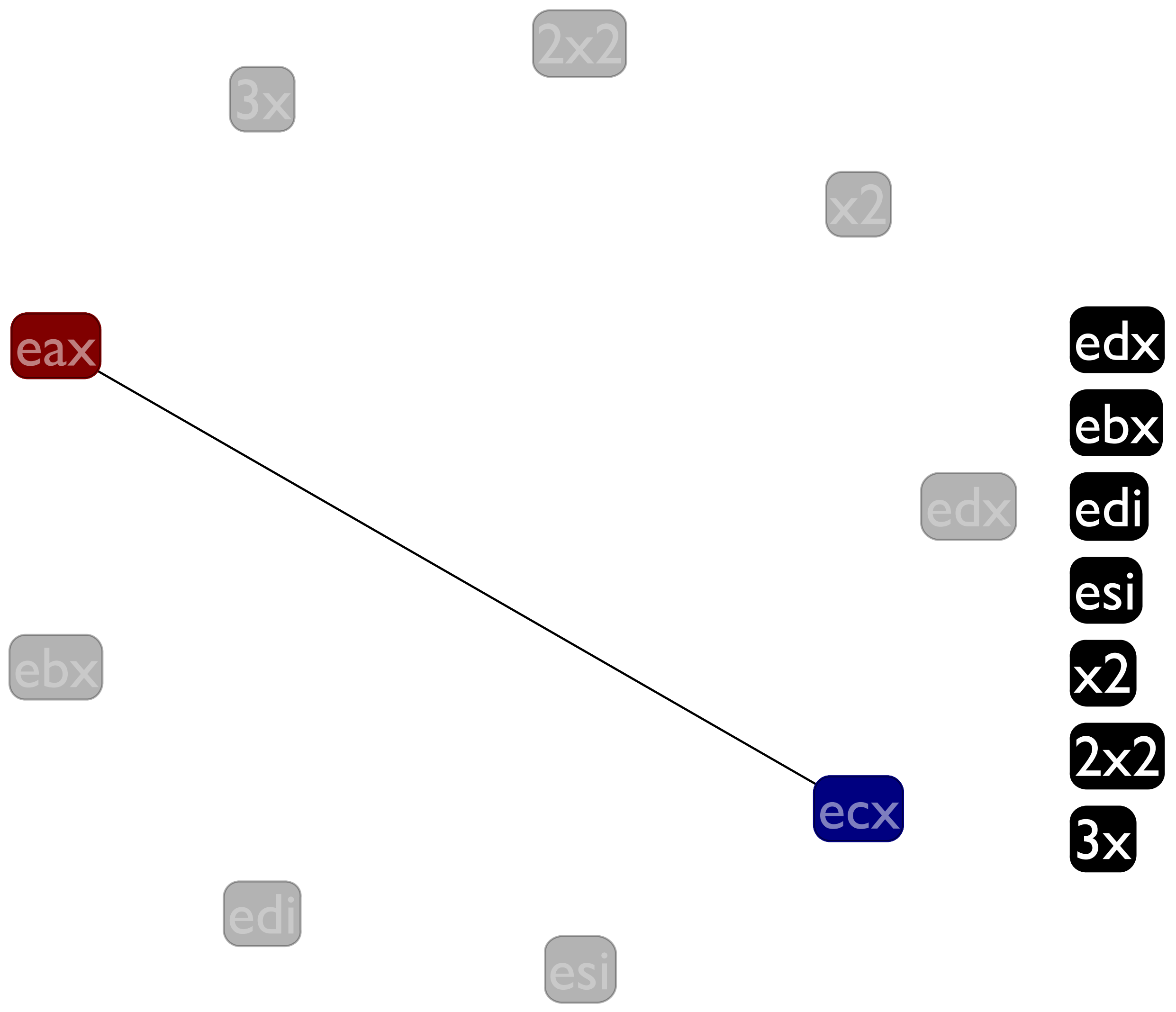

2x2
3x
x2
eax
edx
ebx
edi
esi
x2
2x2
3x
eax
edx
ebx
ecx
edi
esi

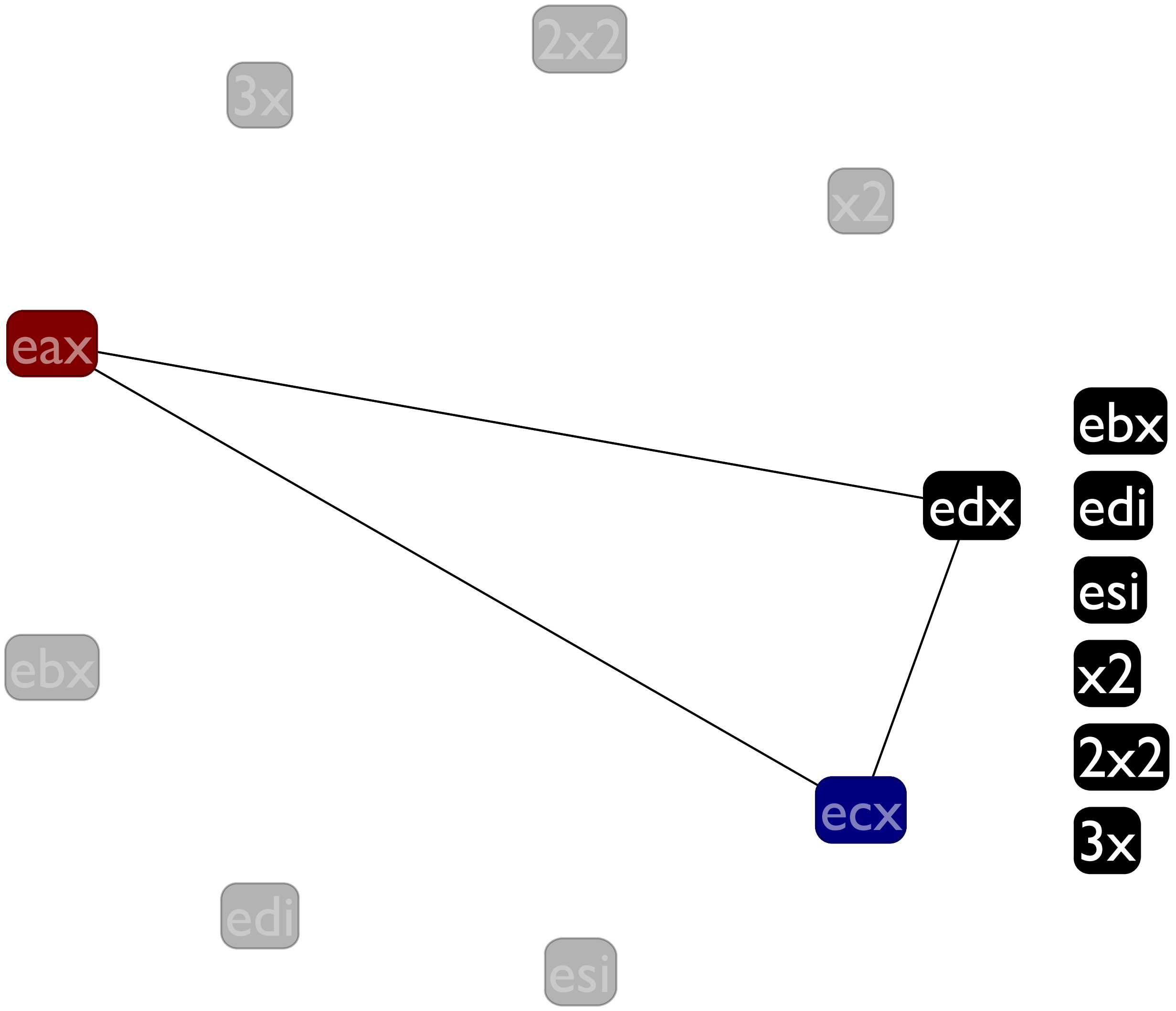

2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
ecx
eax
edx
ebx
edi
esi
x2
2x2
3x

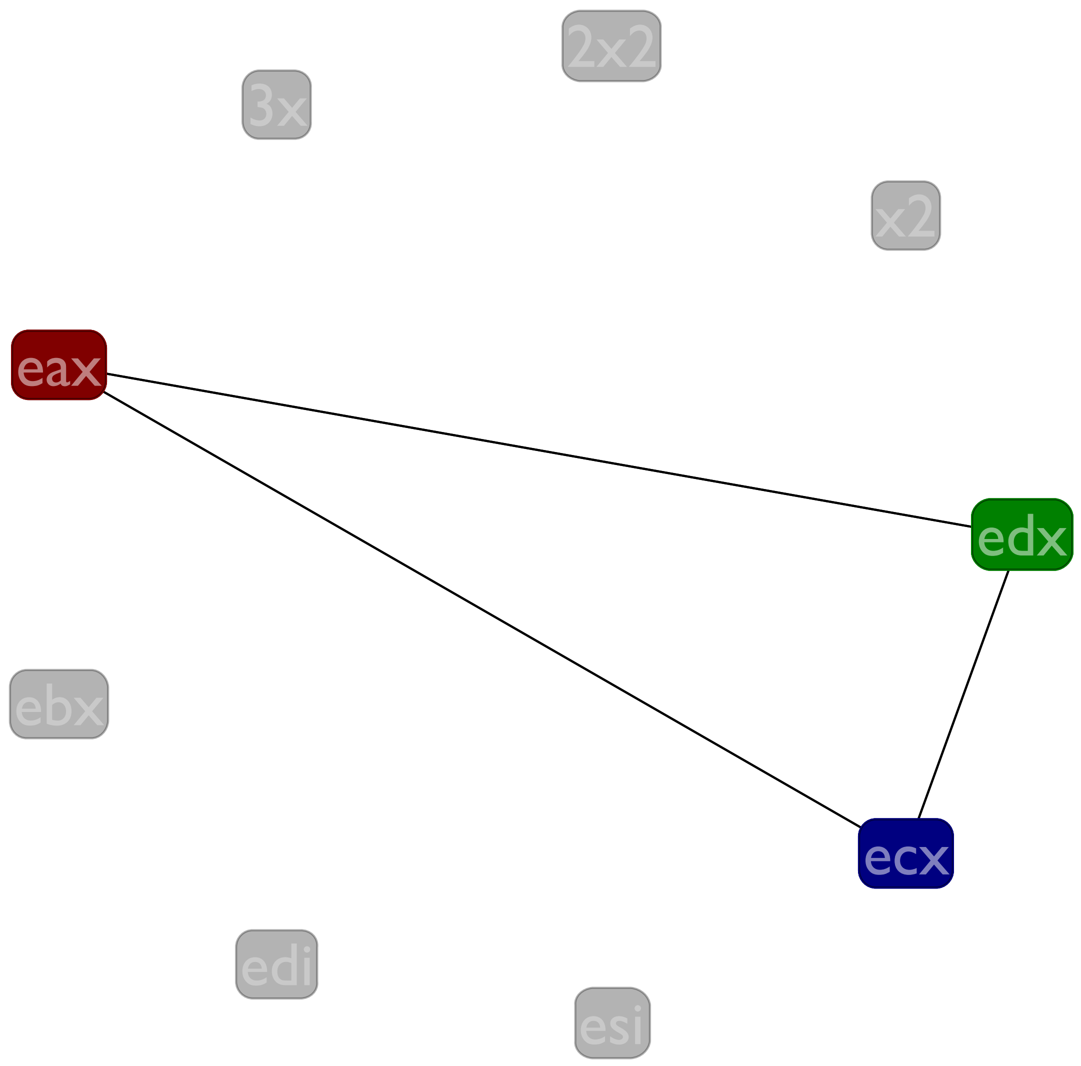

2x2
3x
x2
eax
edx
ebx
edx
edi
esi
ebx
x2
2x2
ecx
3x
edi
esi
eax


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
eax
edx
ebx
edi
esi
x2
2x2
3x


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
edx
ebx
edi
esi
x2
2x2
3x


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
edx
ebx
edi
esi
x2
2x2
3x


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
ebx
edi
esi
x2
2x2
3x



ebx

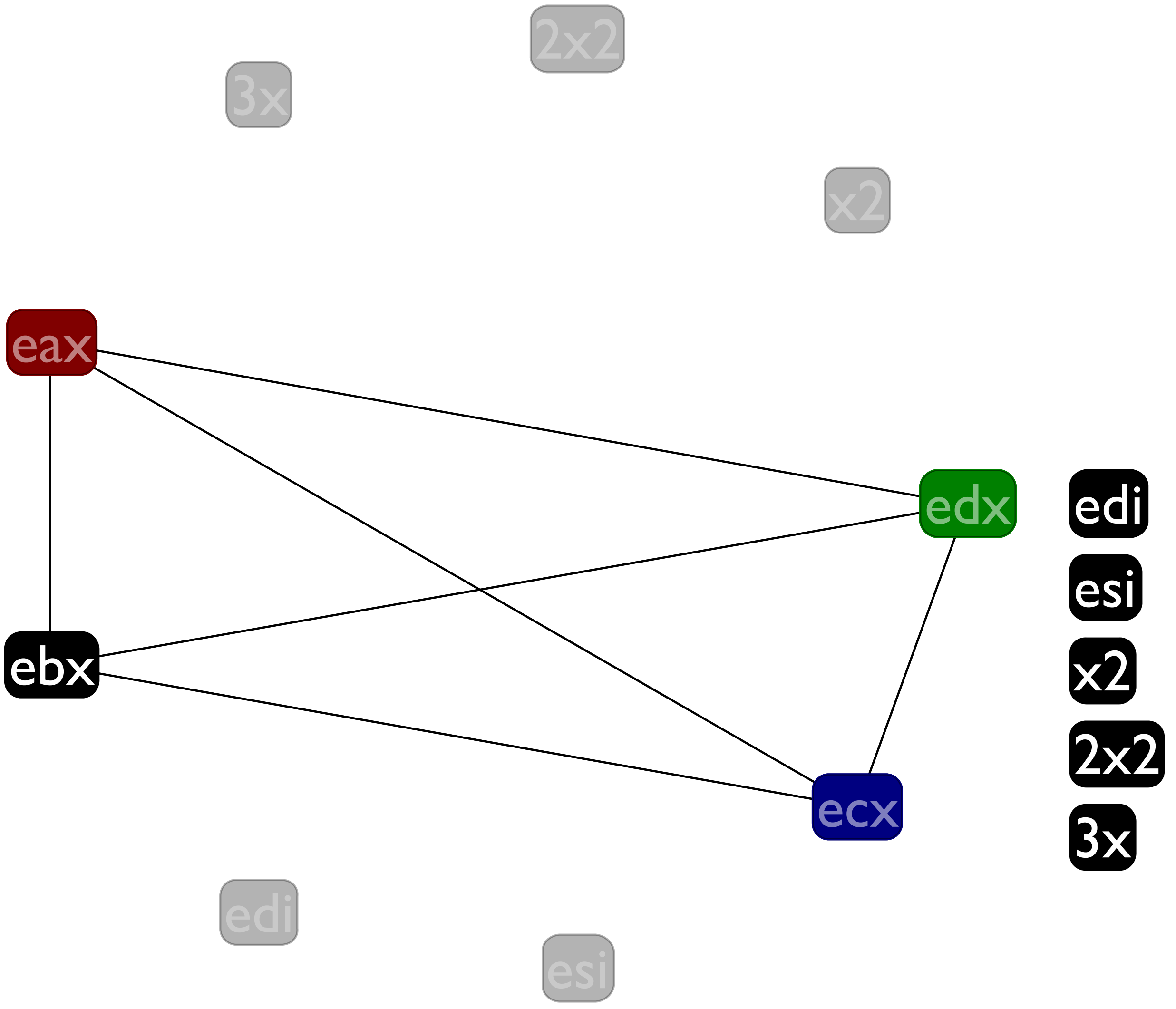
edi

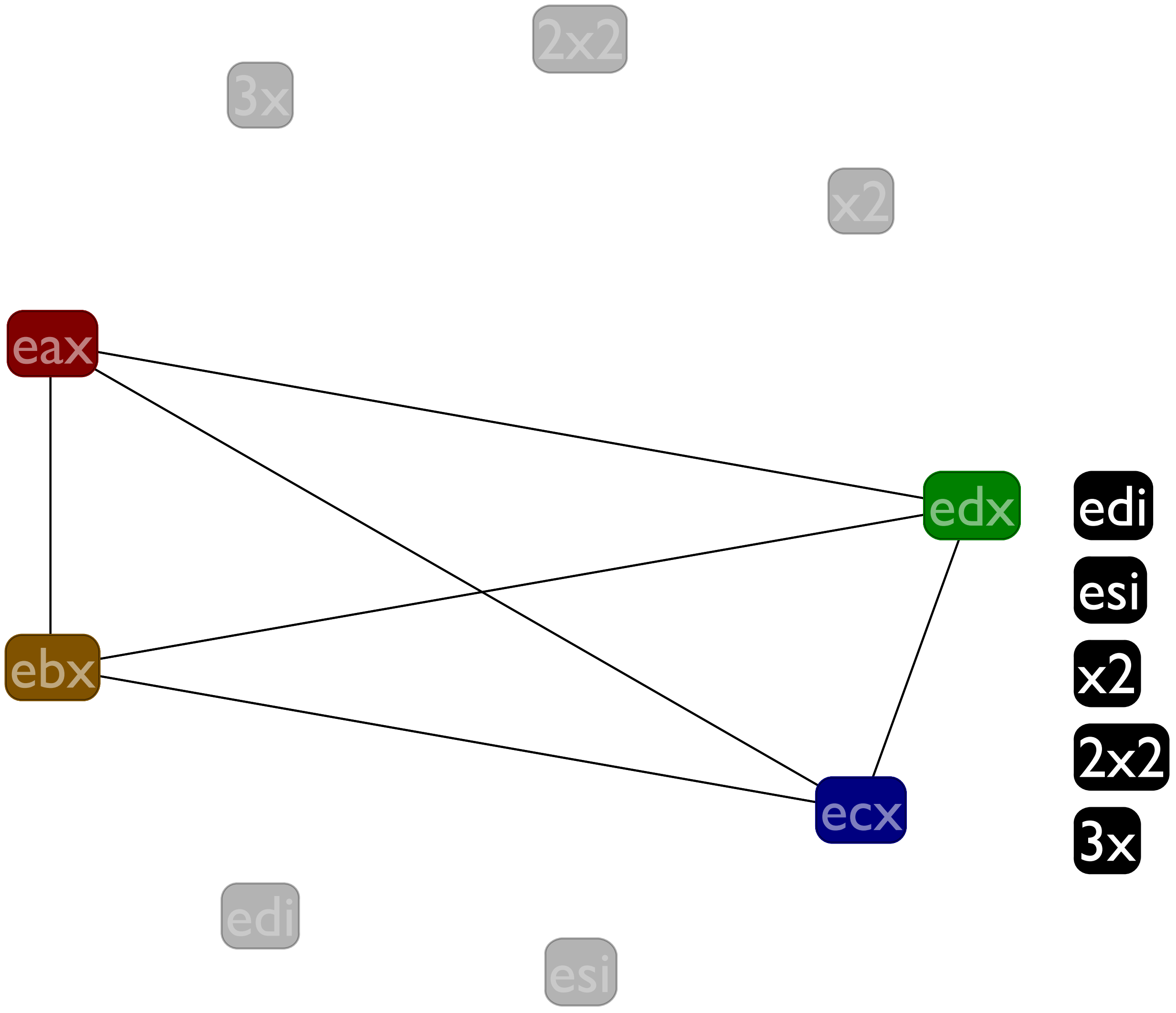
esi

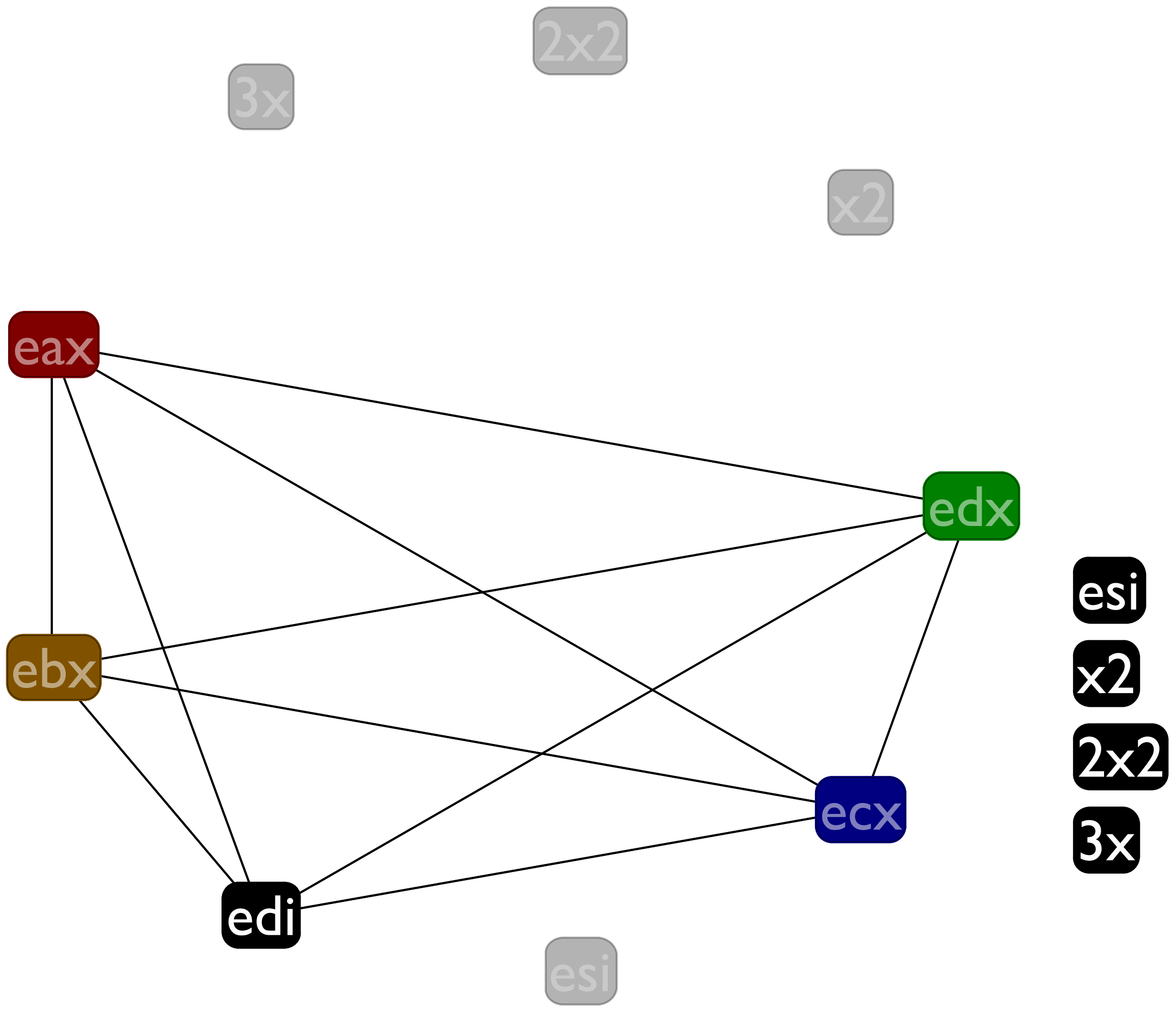
x2

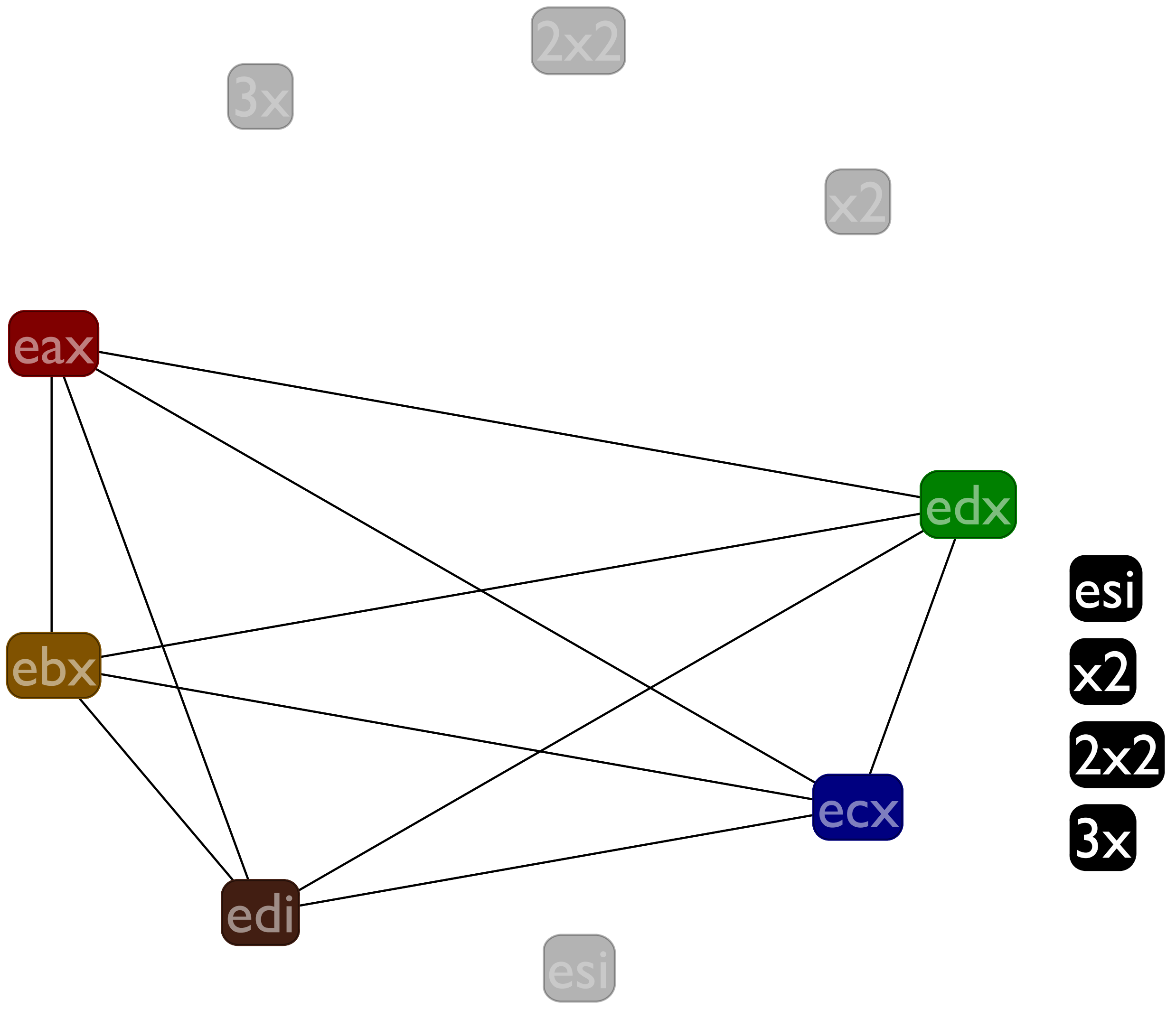
2x2

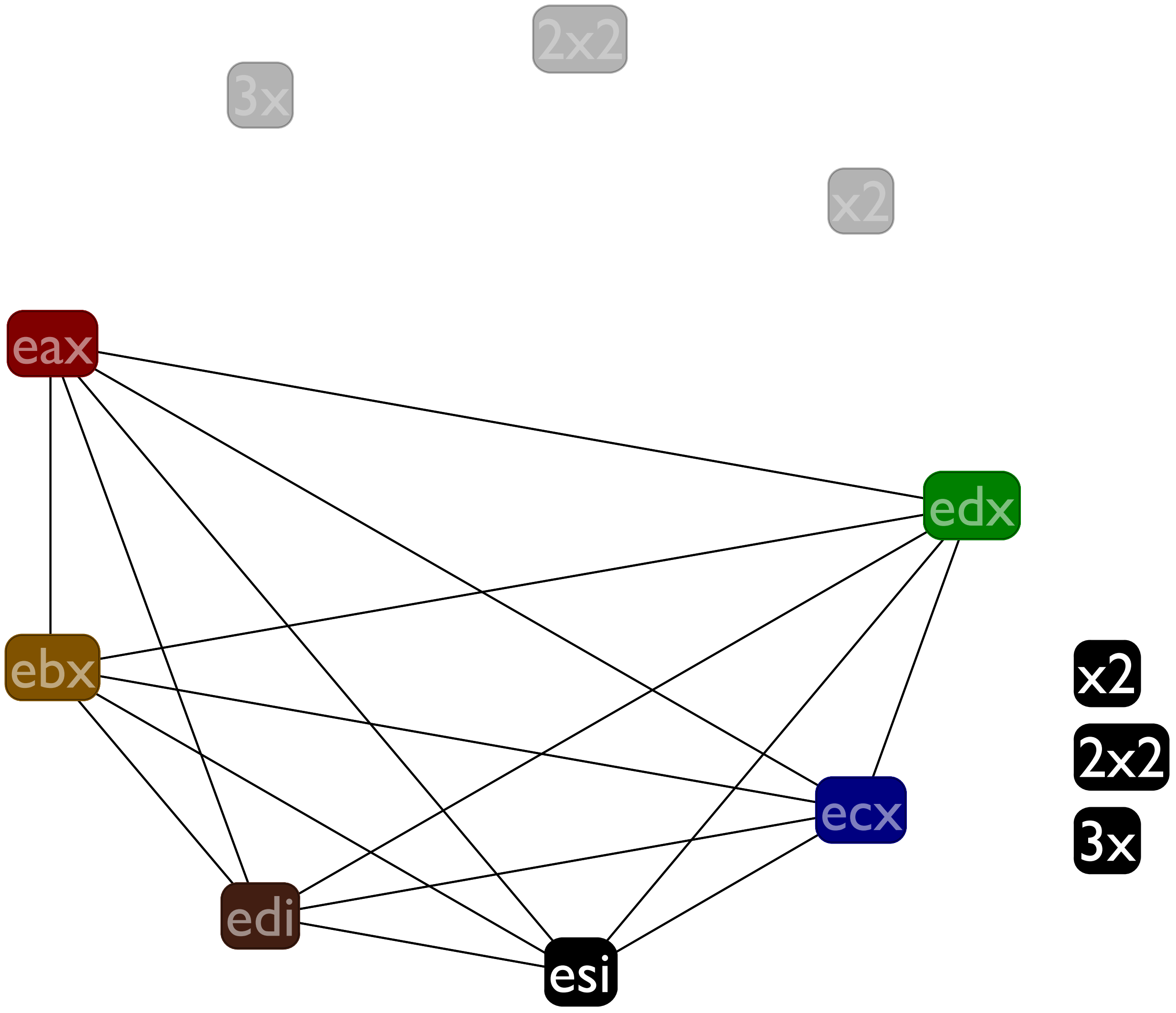
3x

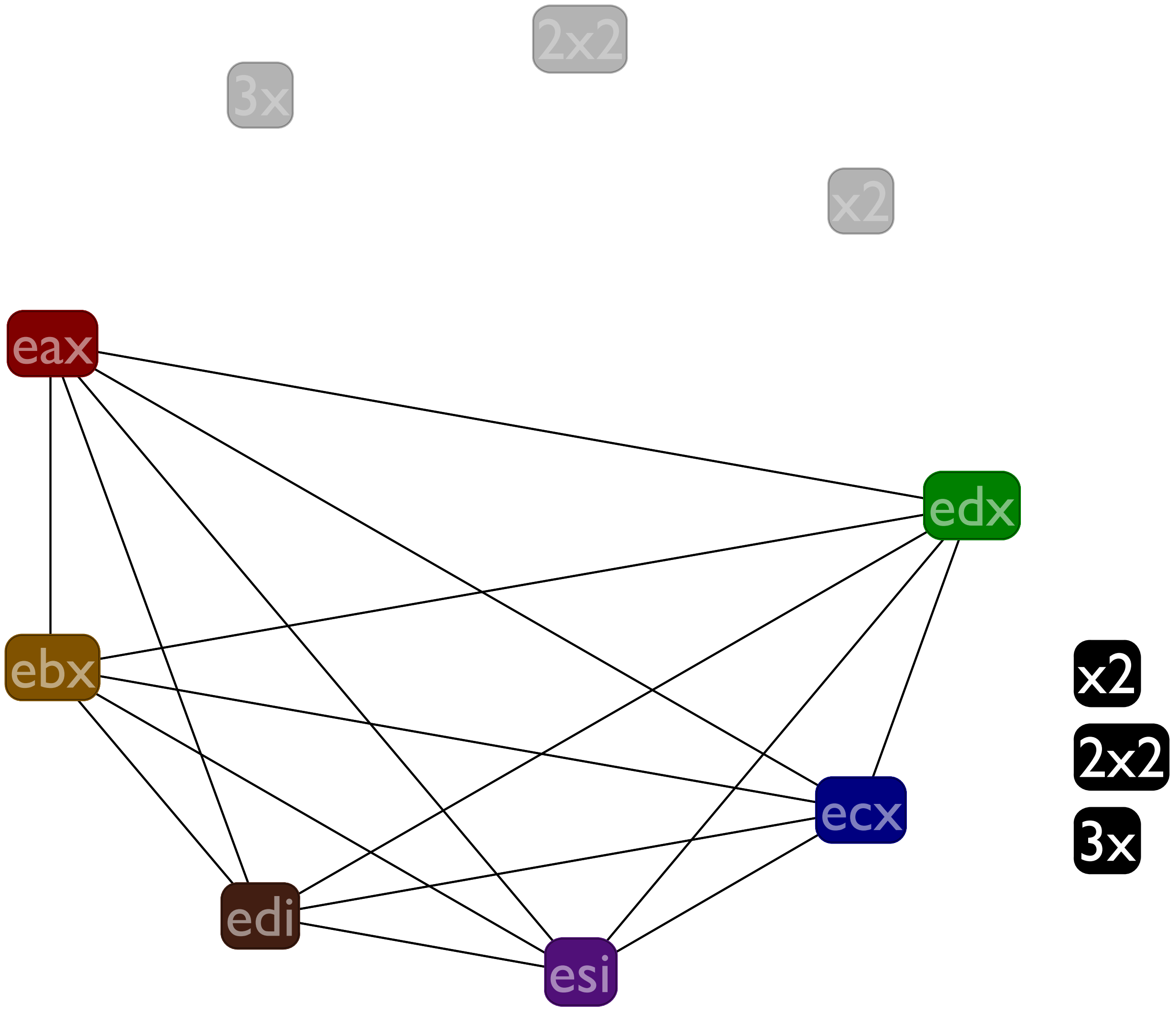

2x2
3x
x2
eax
edx
edi
esi
x2
2x2
3x
ebx
ecx
edi
esi

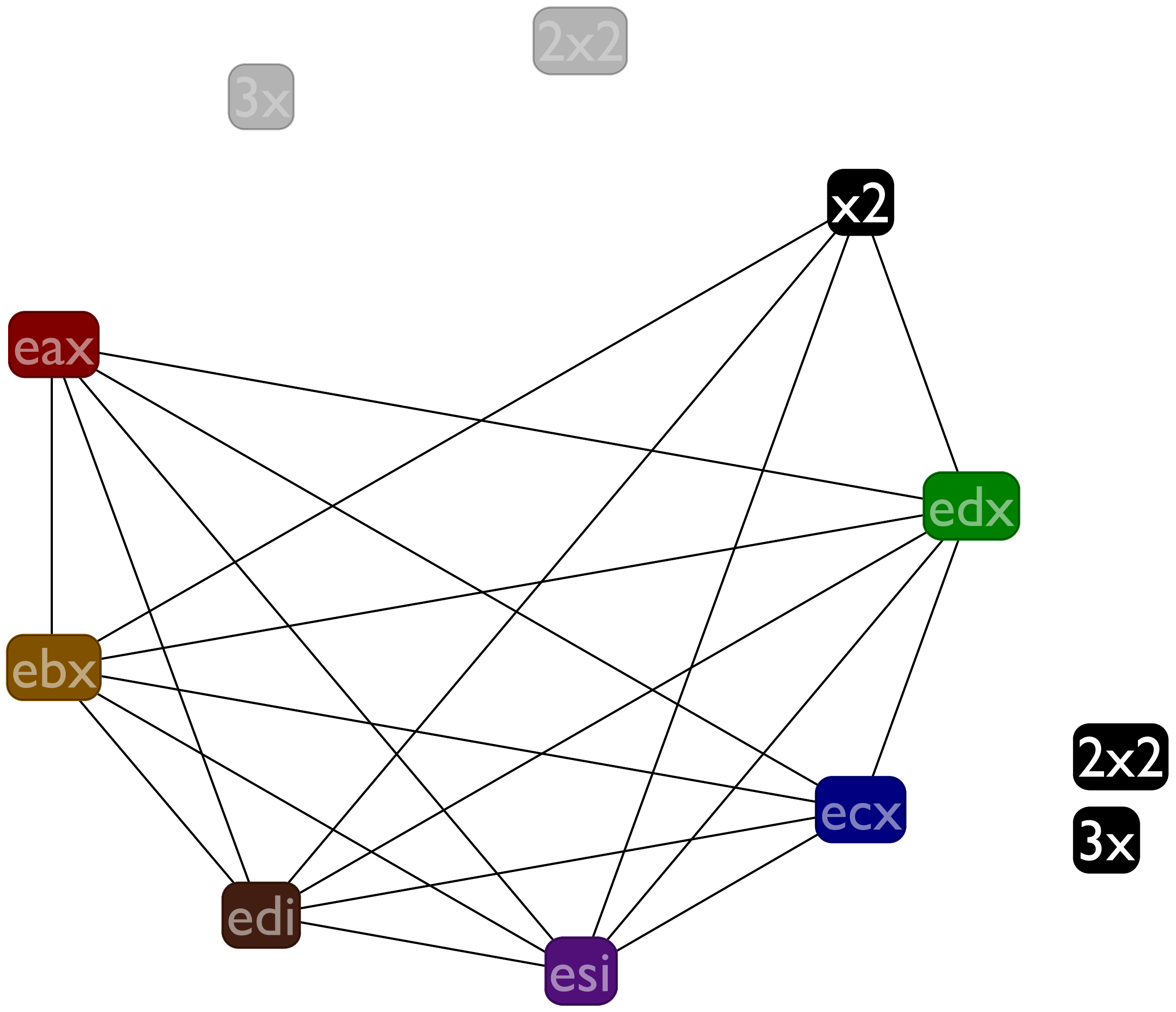

2x2
3x
x2
eax
edx
edi
esi
x2
2x2
3x
ebx
ecx
edi
esi

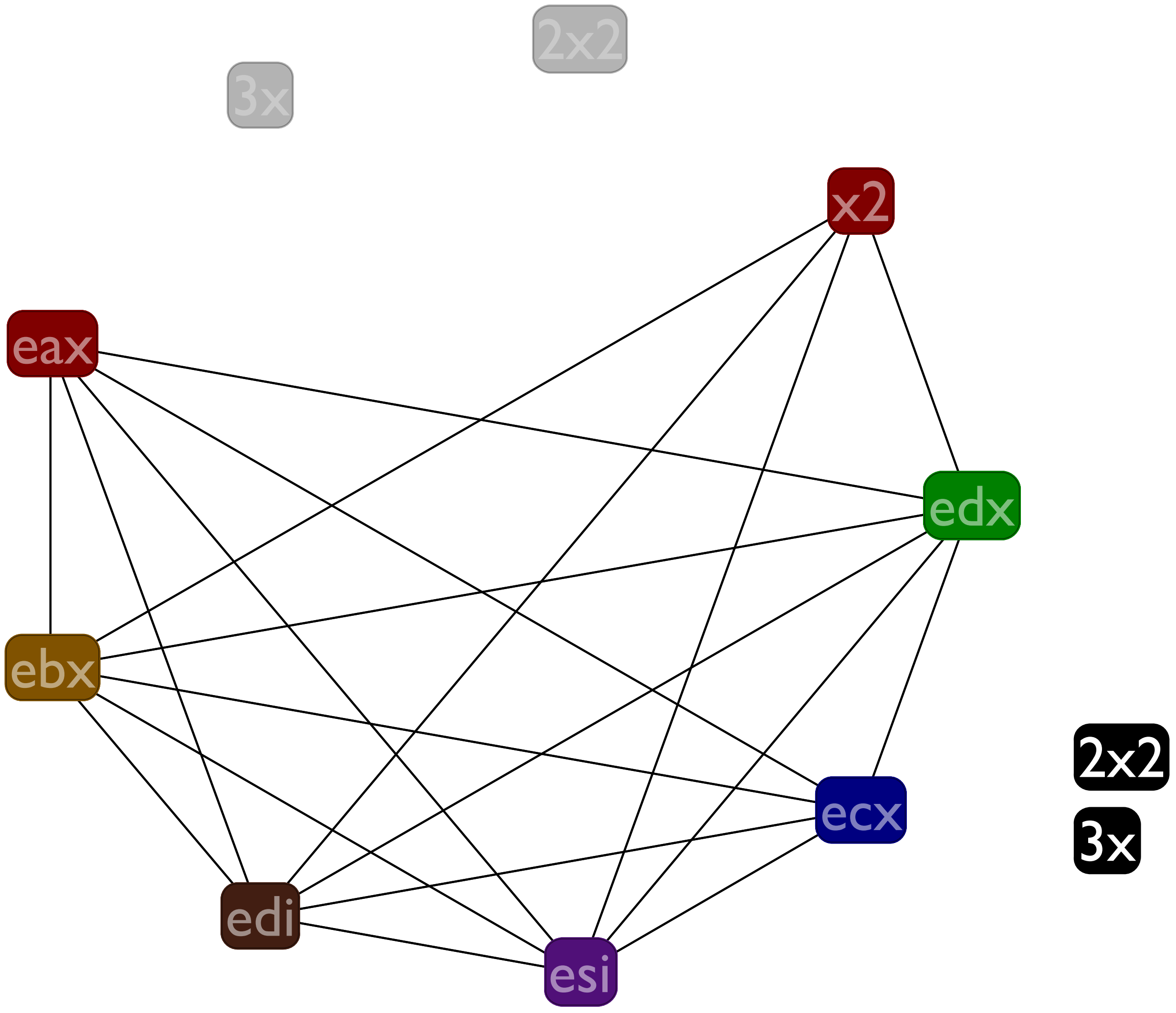

2x2
3x
x2
eax
edx
esi
x2
ebx
2x2
ecx
3x
edi
esi


2x2
3x
x2
eax
edx
esi
x2
2x2
3x
ebx
ecx
edi
esi

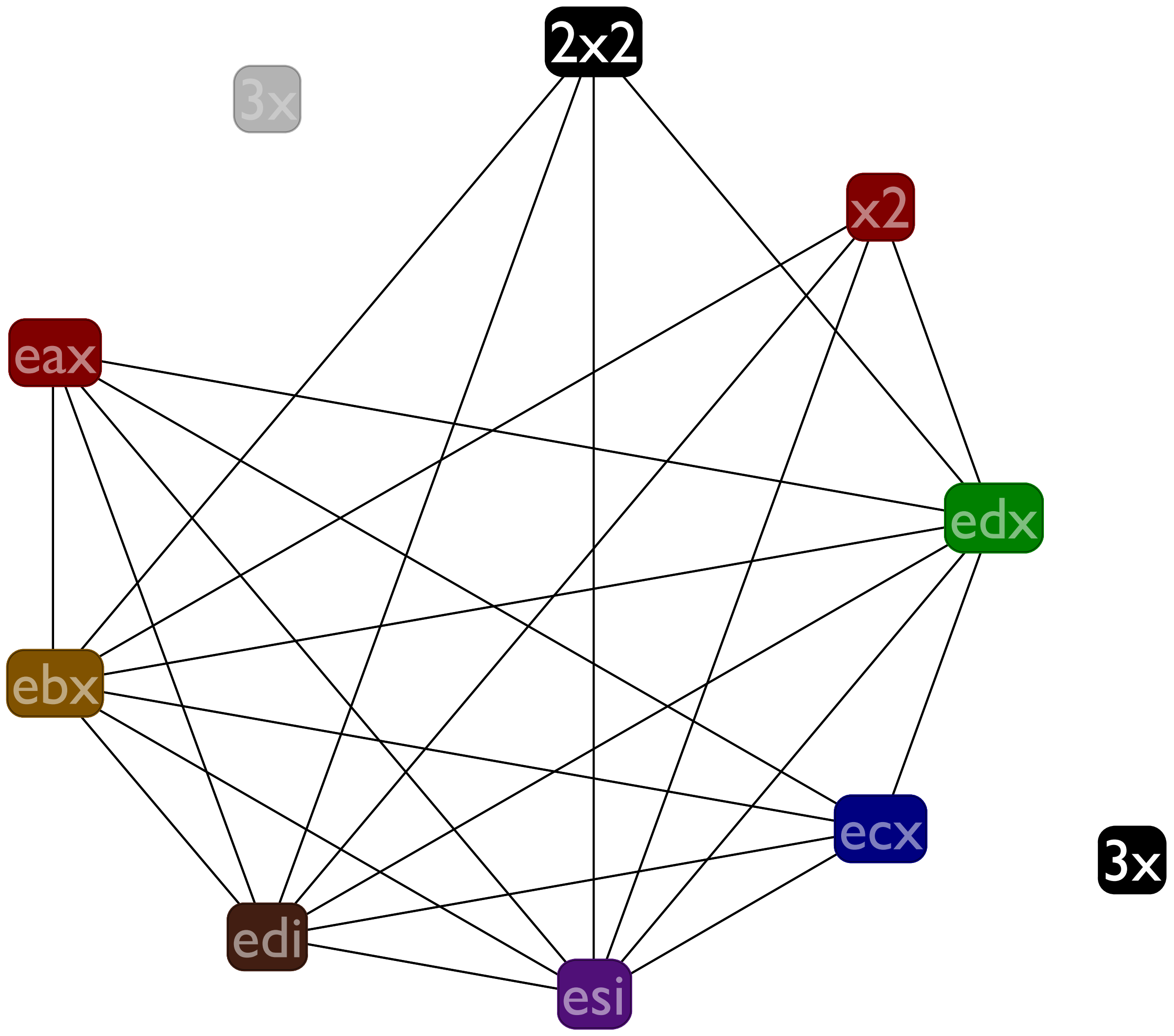

3x
2x2
x2
eax
edx
ebx
ecx
edi
esi
x2
2x2
3x

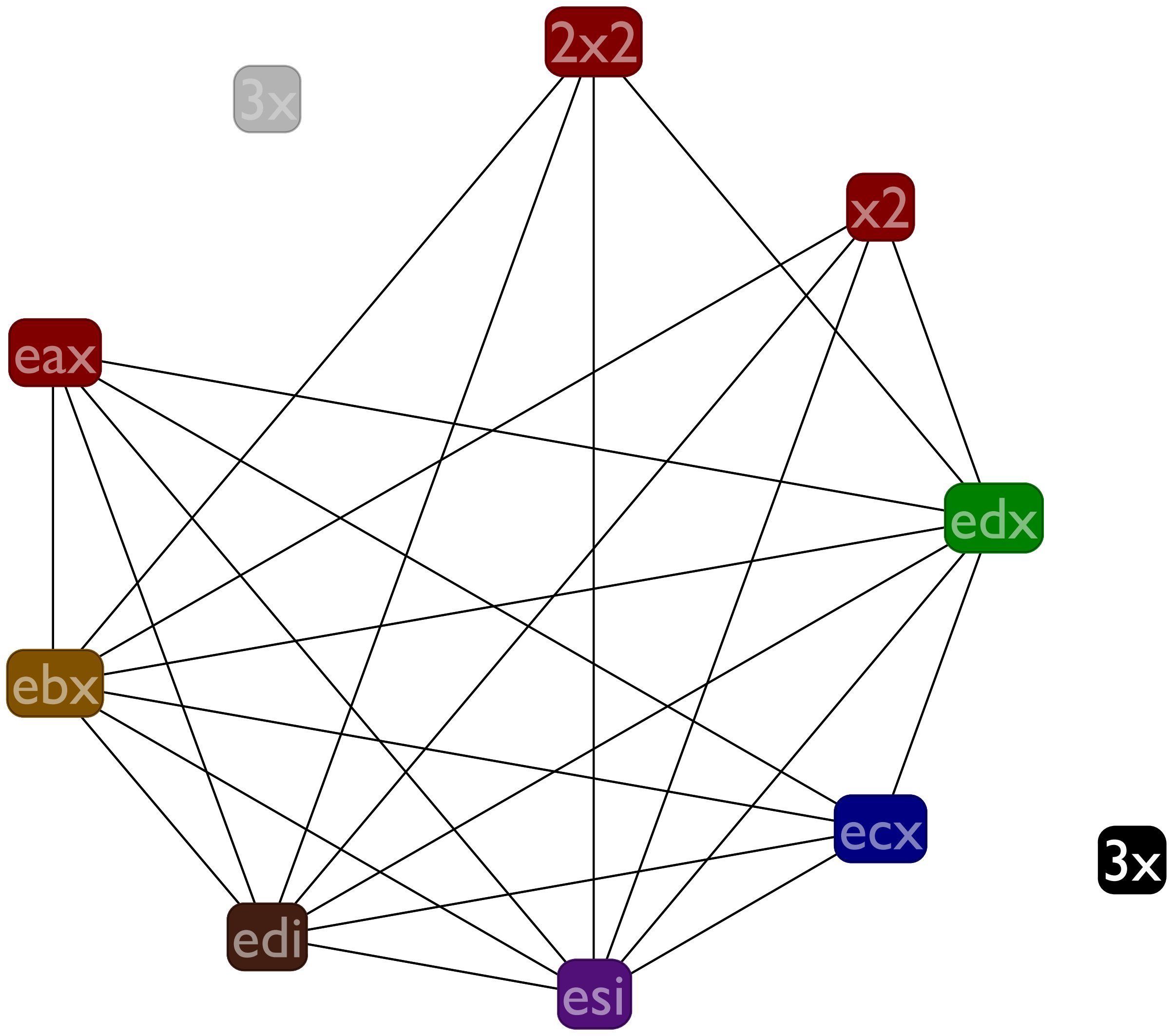

3x
2x2
x2
eax
edx
ebx
ecx
edi
esi
x2
2x2
3x

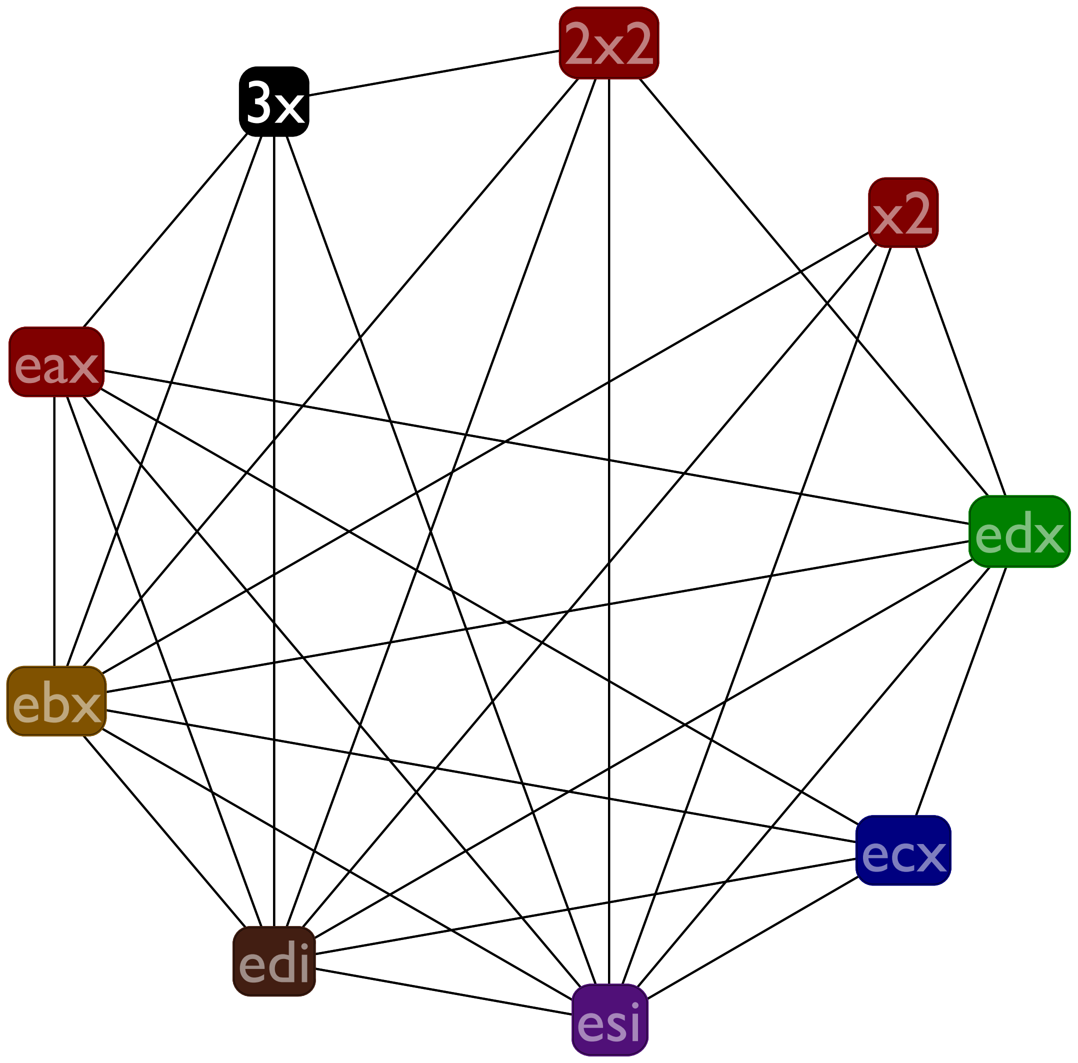

3x
2x2
x2
eax
edx
ebx
ecx
edi
esi
2x2
3x

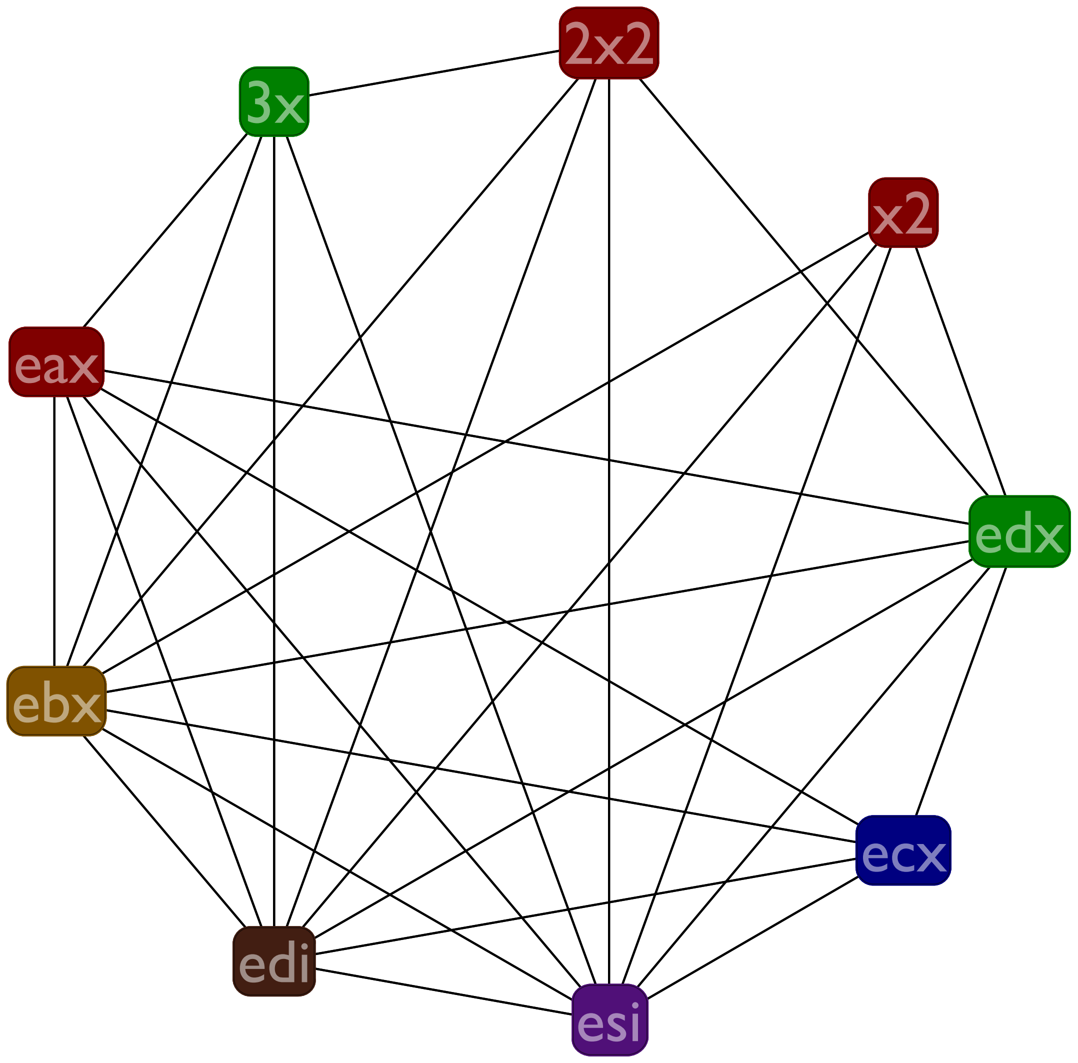


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi
2x2
3x


2x2
3x
x2
eax
edx
ebx
ecx
3x
edi
esi


3x
2x2
x2
eax
edx
ebx
ecx
3x
edi
esi


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi


2x2
3x
x2
eax
edx
ebx
ecx
edi
esi

Function Calls

```
;; f(x) = g(x) + h(x)
(:f
 (in <- edx)       ;; save our argument
 (call :g)         ;; call g with our argument
 (edx <- in)       ;; set up call to h
 (g-ans <- eax)    ;; save g's result
 (call :h)         ;; call h
 (eax += g-ans)    ;; add in saved result
 (return))         ;; and we're done.
```

Gen & Kill

		gen	kill
1:	`:f`	()	()
2:	`(in <- edx)`	(edx)	(in)
3:	`(call :g)`	()	(eax ecx edx)
4:	`(edx <- in)`	(in)	(edx)
5:	`(g-ans <- eax)`	(eax)	(g-ans)
6:	`(call :h)`	()	(eax ecx edx)
7:	`(eax += g-ans)`	(eax g-ans)	(eax)
8:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	()	()
2:	`(in <- edx)`	()	()
3:	`(call :g)`	()	()
4:	`(edx <- in)`	()	()
5:	`(g-ans <- eax)`	()	()
6:	`(call :h)`	()	()
7:	`(eax += g-ans)`	()	()
8:	`(return)`	()	()

Liveness

		in	out
1:	`:f`	()	()
2:	`(in <- edx)`	(edx)	()
3:	`(call :g)`	()	()
4:	`(edx <- in)`	(in)	()
5:	`(g-ans <- eax)`	(eax)	()
6:	`(call :h)`	()	()
7:	`(eax += g-ans)`	(eax g-ans)	()
8:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	()	(edx)
2:	`(in <- edx)`	(edx)	()
3:	`(call :g)`	()	(in)
4:	`(edx <- in)`	(in)	(eax)
5:	`(g-ans <- eax)`	(eax)	()
6:	`(call :h)`	()	(eax g-ans)
7:	`(eax += g-ans)`	(eax g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

Liveness

	in	out
1: `:f`	(edx)	(edx)
2: `(in <- edx)`	(edx)	()
3: `(call :g)`	(in)	(in)
4: `(edx <- in)`	(eax in)	(eax)
5: `(g-ans <- eax)`	(eax)	()
6: `(call :h)`	(g-ans)	(eax g-ans)
7: `(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8: `(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(in <- edx)`	(edx)	(in)
3:	`(call :g)`	(in)	(eax in)
4:	`(edx <- in)`	(eax in)	(eax)
5:	`(g-ans <- eax)`	(eax)	(g-ans)
6:	`(call :h)`	(g-ans)	(eax ebx edi esi g-ans)
7:	`(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(in <- edx)`	(edx)	(in)
3:	`(call :g)`	(in)	(eax in)
4:	`(edx <- in)`	(eax in)	(eax)
5:	`(g-ans <- eax)`	(eax)	(g-ans)
6:	`(call :h)`	(ebx edi esi g-ans)	(eax ebx edi esi g-ans)
7:	`(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

Liveness

	in	out
1: `:f`	(edx)	(edx)
2: `(in <- edx)`	(edx)	(in)
3: `(call :g)`	(in)	(eax in)
4: `(edx <- in)`	(eax in)	(eax)
5: `(g-ans <- eax)`	(eax)	(ebx edi esi g-ans)
6: `(call :h)`	(ebx edi esi g-ans)	(eax ebx edi esi g-ans)
7: `(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8: `(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(in <- edx)`	(edx)	(in)
3:	`(call :g)`	(in)	(eax in)
4:	`(edx <- in)`	(eax in)	(eax)
5:	`(g-ans <- eax)`	(eax ebx edi esi)	(ebx edi esi g-ans)
6:	`(call :h)`	(ebx edi esi g-ans)	(eax ebx edi esi g-ans)
7:	`(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(in <- edx)`	(edx)	(in)
3:	`(call :g)`	(in)	(eax in)
4:	`(edx <- in)`	(eax in)	(eax ebx edi esi)
5:	`(g-ans <- eax)`	(eax ebx edi esi)	(ebx edi esi g-ans)
6:	`(call :h)`	(ebx edi esi g-ans)	(eax ebx edi esi g-ans)
7:	`(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(in <- edx)`	(edx)	(in)
3:	`(call :g)`	(in)	(eax in)
4:	`(edx <- in)`	(eax ebx edi esi in)	(eax ebx edi esi)
5:	`(g-ans <- eax)`	(eax ebx edi esi)	(ebx edi esi g-ans)
6:	`(call :h)`	(ebx edi esi g-ans)	(eax ebx edi esi g-ans)
7:	`(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(in <- edx)`	(edx)	(in)
3:	`(call :g)`	(in)	(eax ebx edi esi in)
4:	`(edx <- in)`	(eax ebx edi esi in)	(eax ebx edi esi)
5:	`(g-ans <- eax)`	(eax ebx edi esi)	(ebx edi esi g-ans)
6:	`(call :h)`	(ebx edi esi g-ans)	(eax ebx edi esi g-ans)
7:	`(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(in <- edx)`	(edx)	(in)
3:	`(call :g)`	(ebx edi esi in)	(eax ebx edi esi in)
4:	`(edx <- in)`	(eax ebx edi esi in)	(eax ebx edi esi)
5:	`(g-ans <- eax)`	(eax ebx edi esi)	(ebx edi esi g-ans)
6:	`(call :h)`	(ebx edi esi g-ans)	(eax ebx edi esi g-ans)
7:	`(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

Liveness

		in	out
1:	`:f`	(edx)	(edx)
2:	`(in <- edx)`	(edx)	(ebx edi esi in)
3:	`(call :g)`	(ebx edi esi in)	(eax ebx edi esi in)
4:	`(edx <- in)`	(eax ebx edi esi in)	(eax ebx edi esi)
5:	`(g-ans <- eax)`	(eax ebx edi esi)	(ebx edi esi g-ans)
6:	`(call :h)`	(ebx edi esi g-ans)	(eax ebx edi esi g-ans)
7:	`(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

Liveness

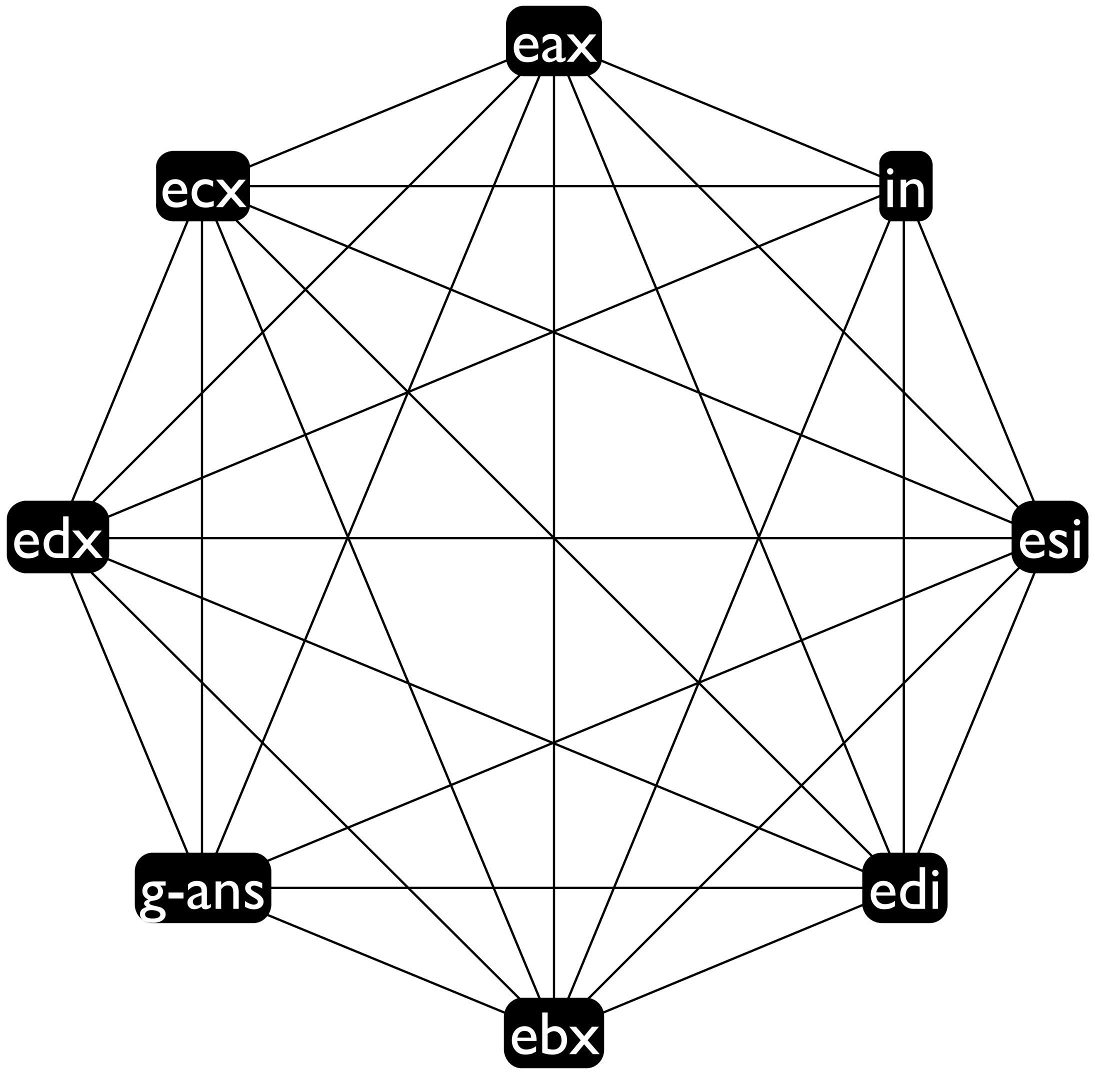
		in	out
1:	`:f`	(edx)	(edx)
2:	`(in <- edx)`	(ebx edi edx esi)	(ebx edi esi in)
3:	`(call :g)`	(ebx edi esi in)	(eax ebx edi esi in)
4:	`(edx <- in)`	(eax ebx edi esi in)	(eax ebx edi esi)
5:	`(g-ans <- eax)`	(eax ebx edi esi)	(ebx edi esi g-ans)
6:	`(call :h)`	(ebx edi esi g-ans)	(eax ebx edi esi g-ans)
7:	`(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

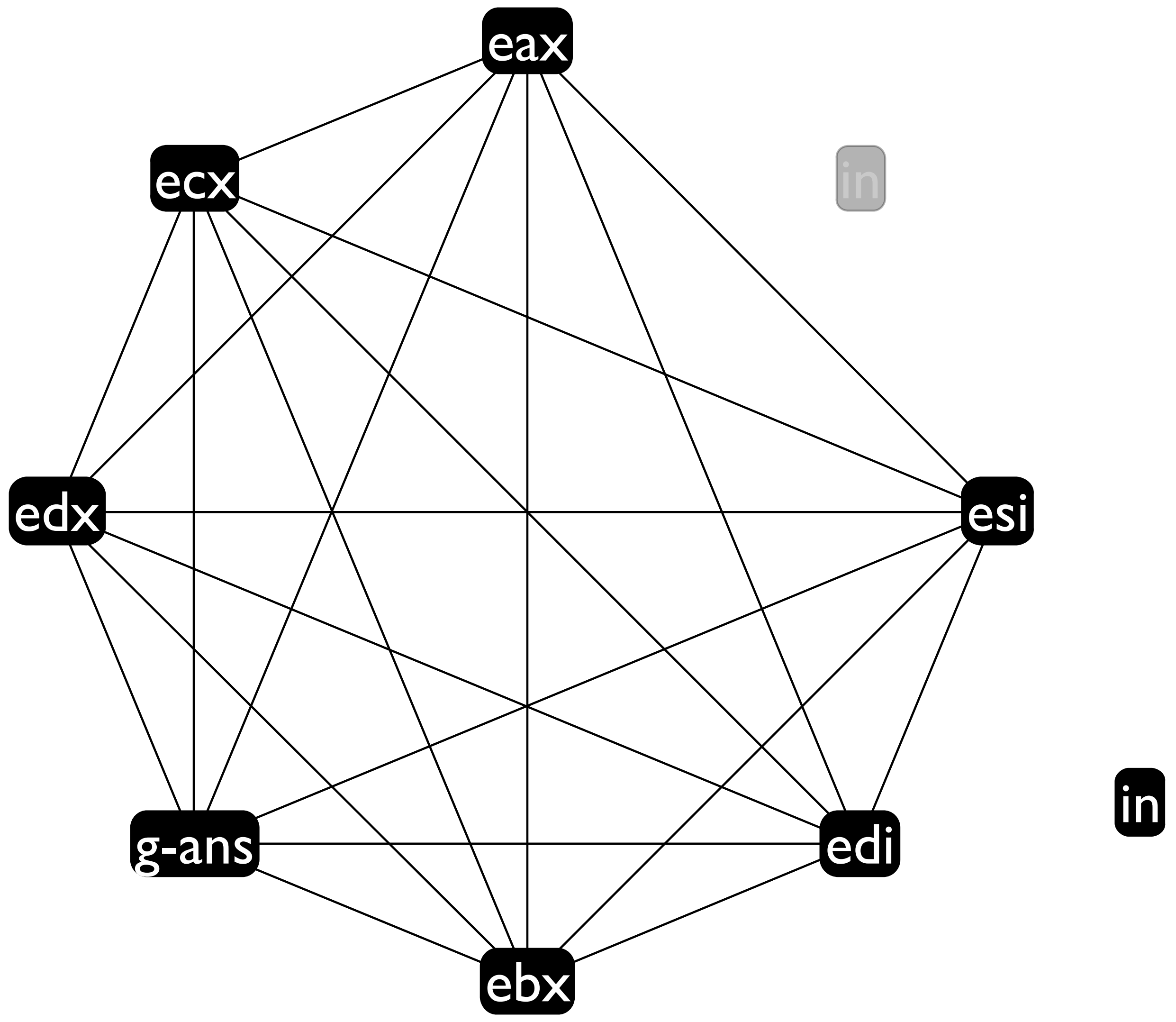
Liveness

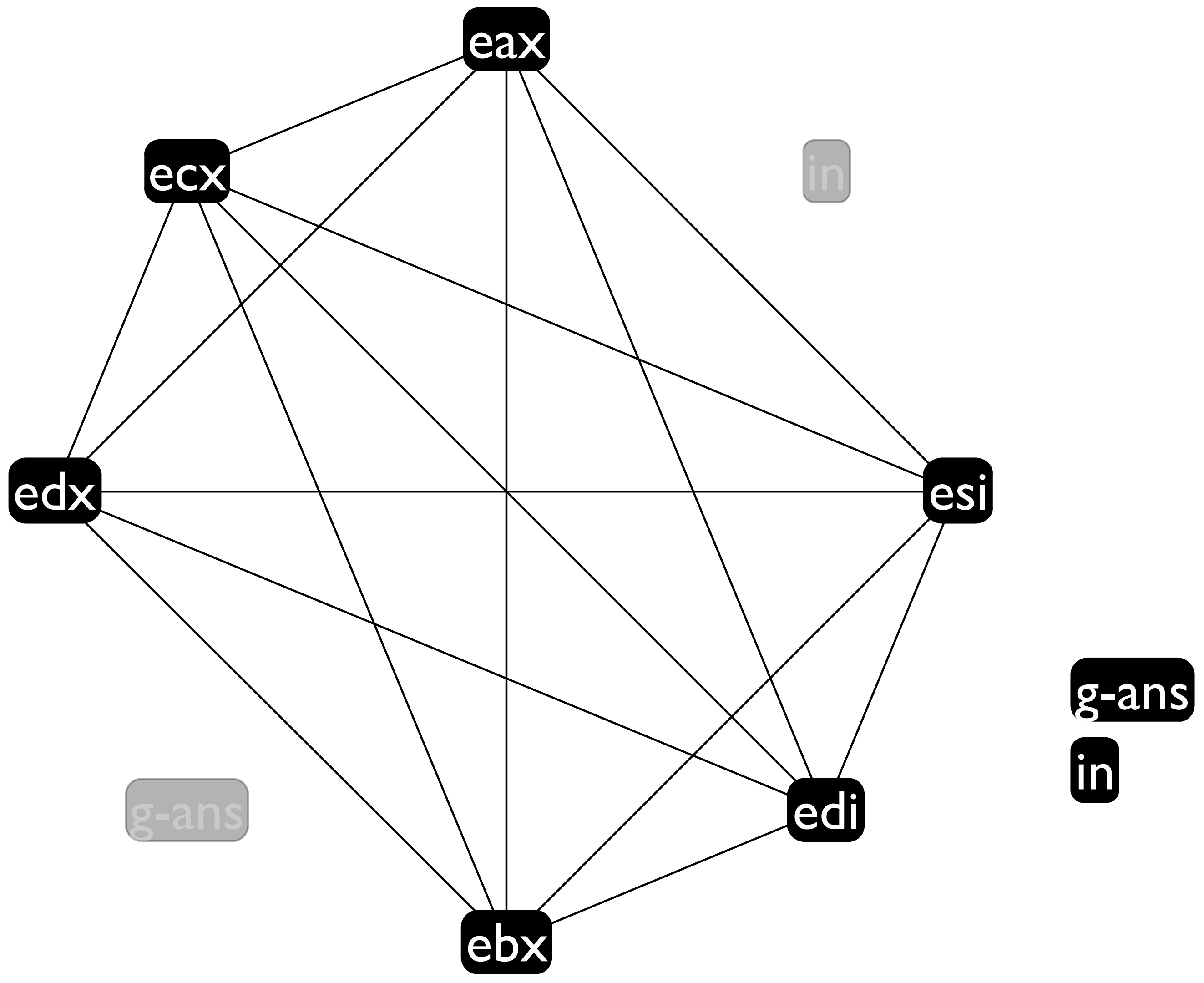
		in	out
1:	`:f`	(edx)	(ebx edi edx esi)
2:	`(in <- edx)`	(ebx edi edx esi)	(ebx edi esi in)
3:	`(call :g)`	(ebx edi esi in)	(eax ebx edi esi in)
4:	`(edx <- in)`	(eax ebx edi esi in)	(eax ebx edi esi)
5:	`(g-ans <- eax)`	(eax ebx edi esi)	(ebx edi esi g-ans)
6:	`(call :h)`	(ebx edi esi g-ans)	(eax ebx edi esi g-ans)
7:	`(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

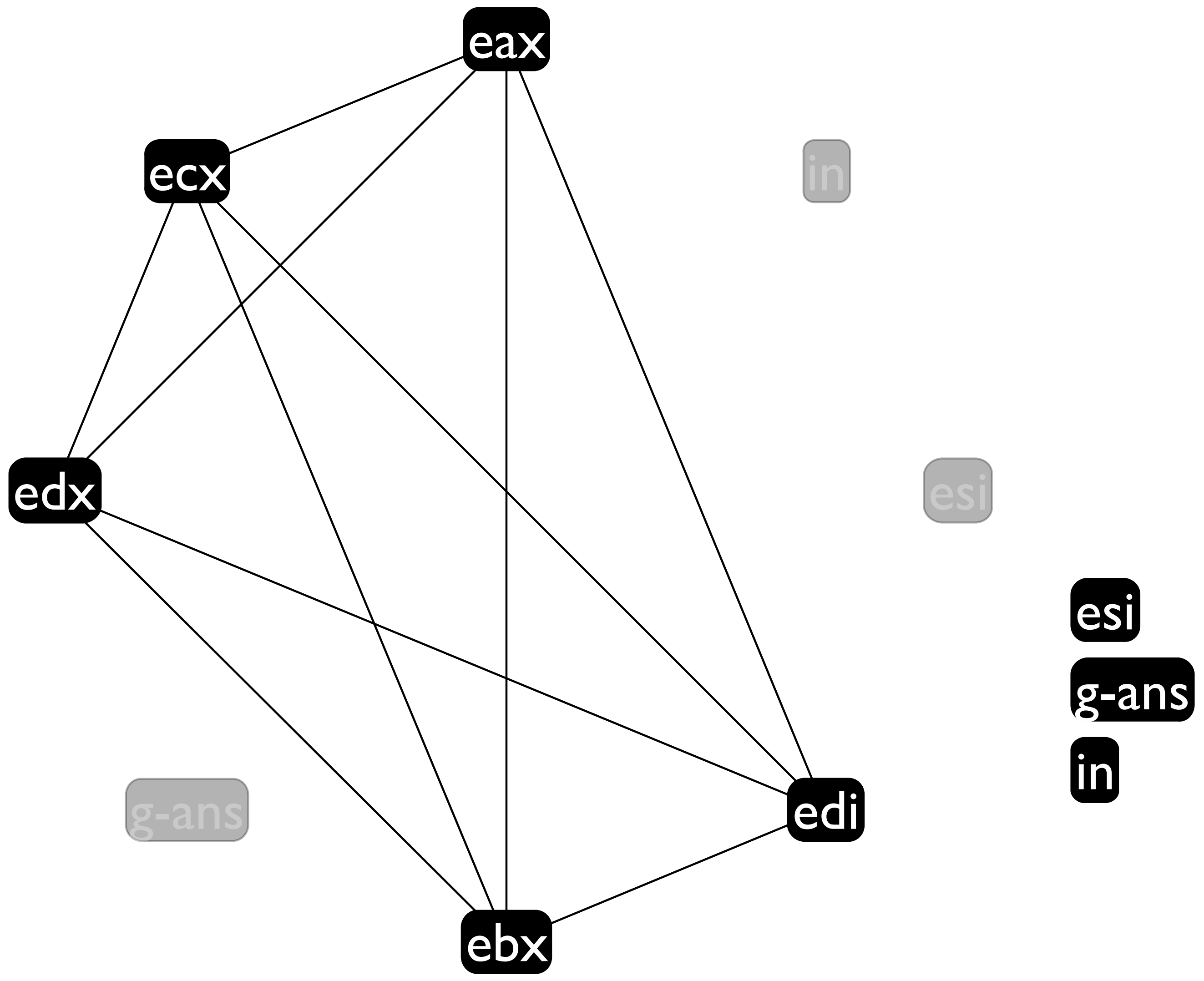
Liveness

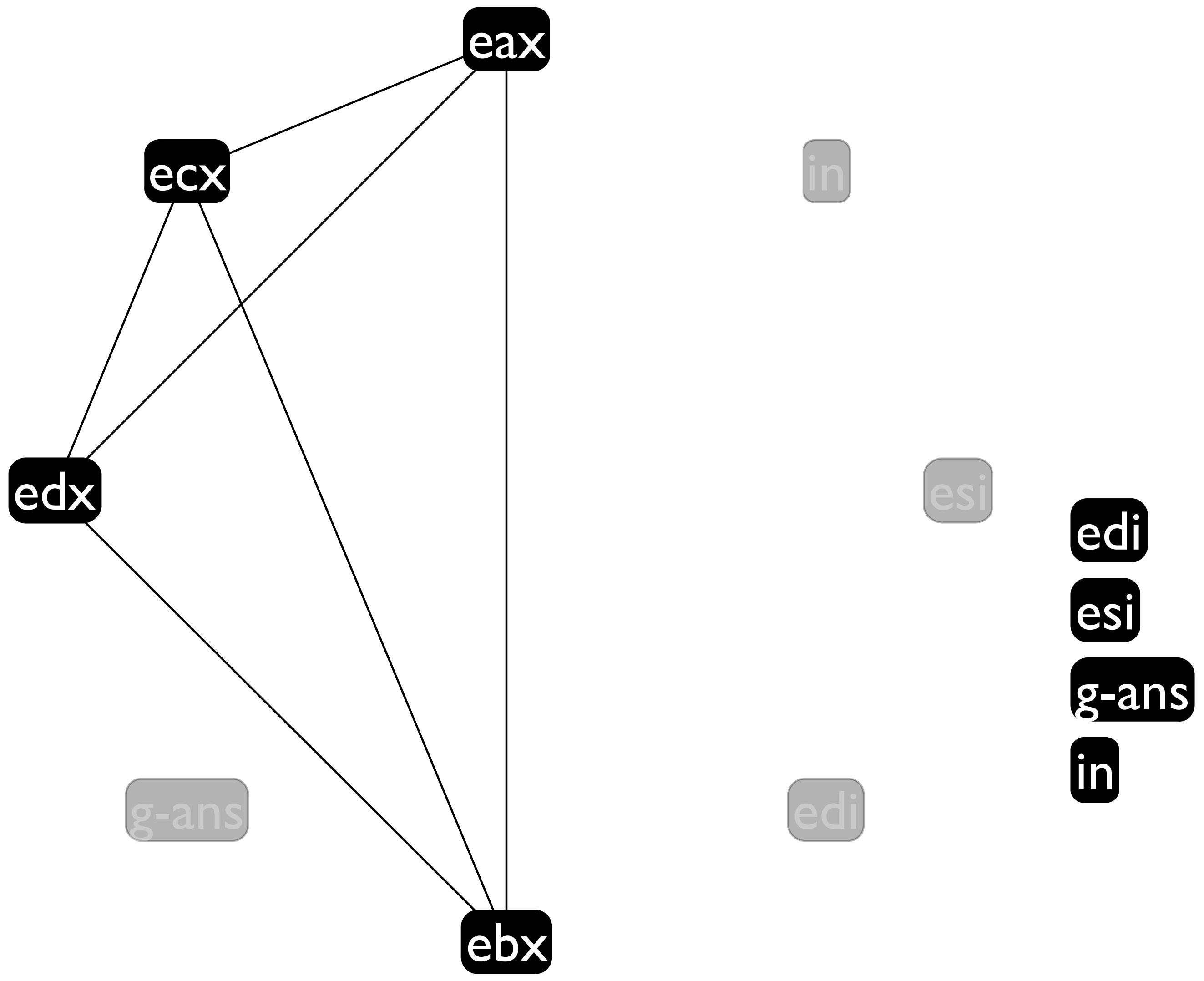
		in	out
1:	`:f`	(ebx edi edx esi)	(ebx edi edx esi)
2:	`(in <- edx)`	(ebx edi edx esi)	(ebx edi esi in)
3:	`(call :g)`	(ebx edi esi in)	(eax ebx edi esi in)
4:	`(edx <- in)`	(eax ebx edi esi in)	(eax ebx edi esi)
5:	`(g-ans <- eax)`	(eax ebx edi esi)	(ebx edi esi g-ans)
6:	`(call :h)`	(ebx edi esi g-ans)	(eax ebx edi esi g-ans)
7:	`(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

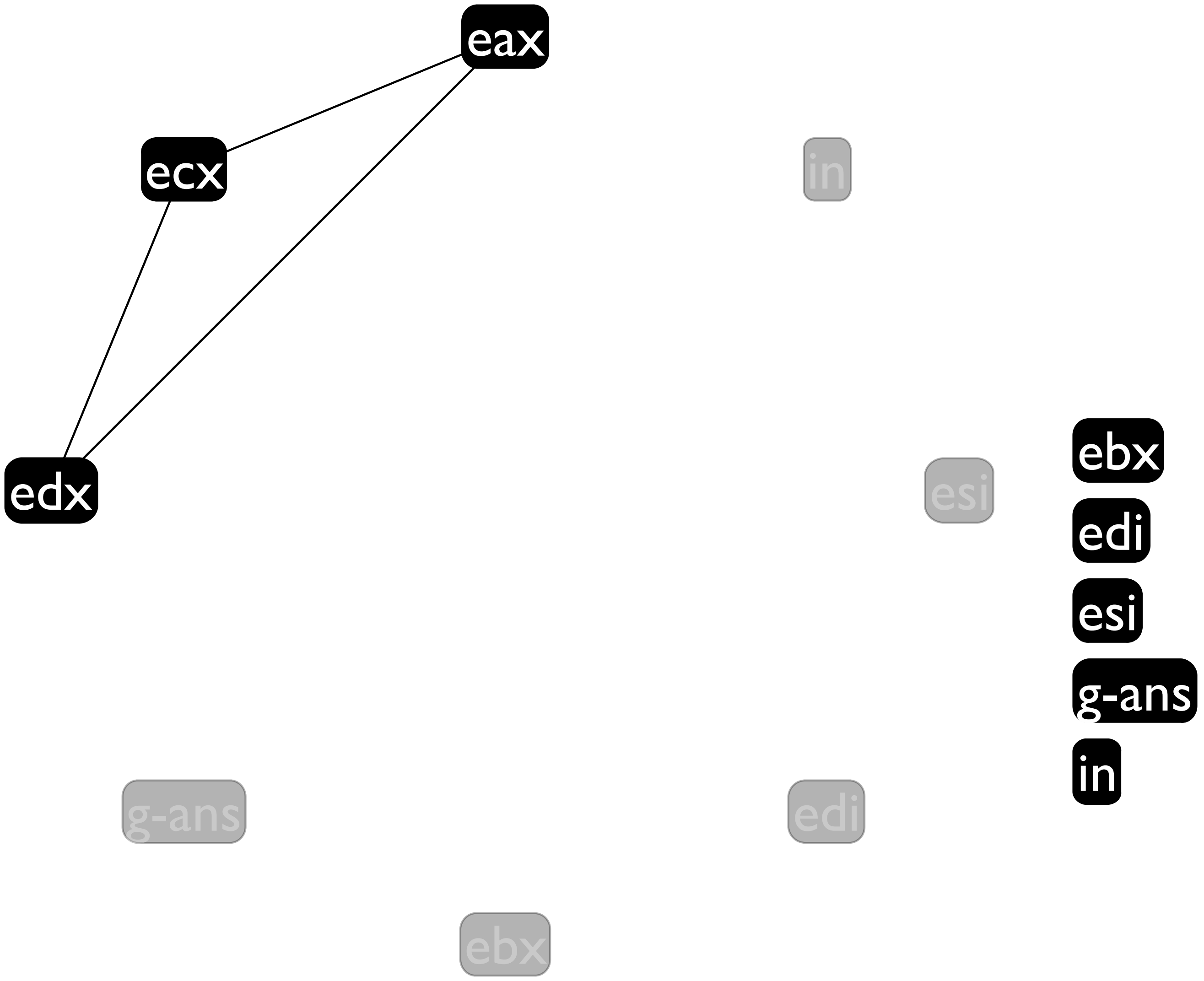

eax
ecx
in
edx
esi
g-ans
edi
ebx

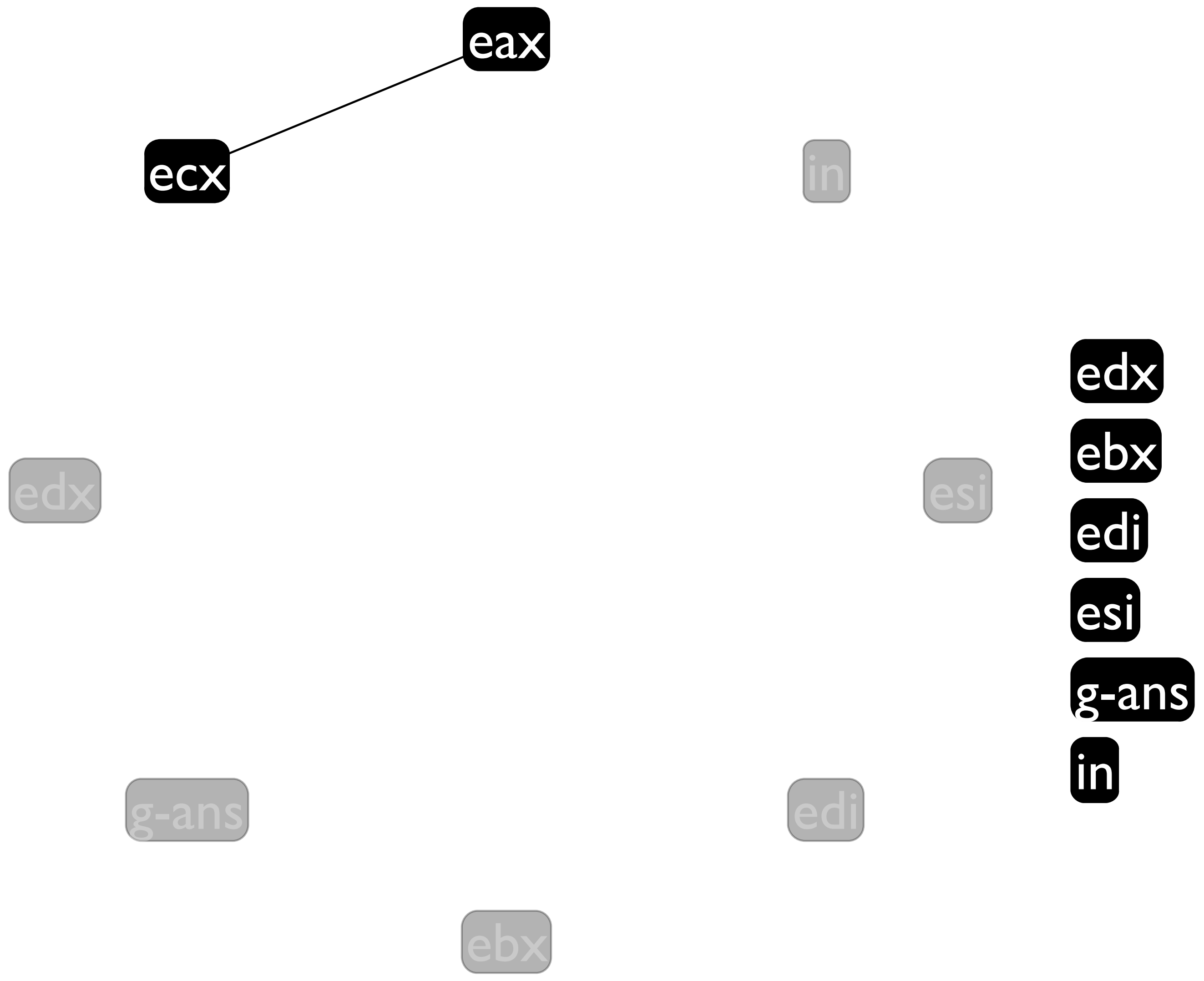

eax
ecx
in
edx
esi
in
g-ans
edi
ebx

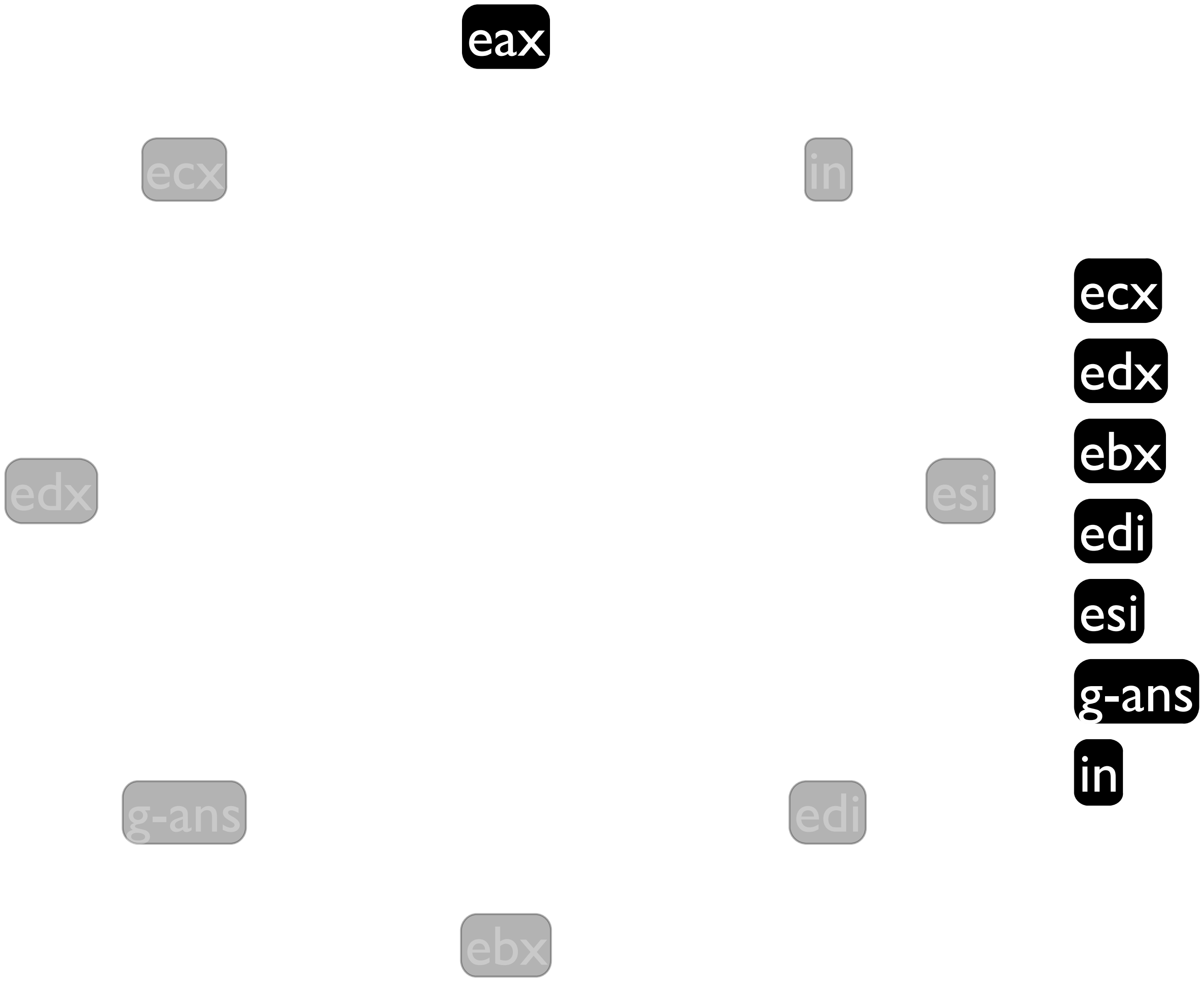

eax
ecx
in
edx
esi
g-ans
in
g-ans
edi
ebx

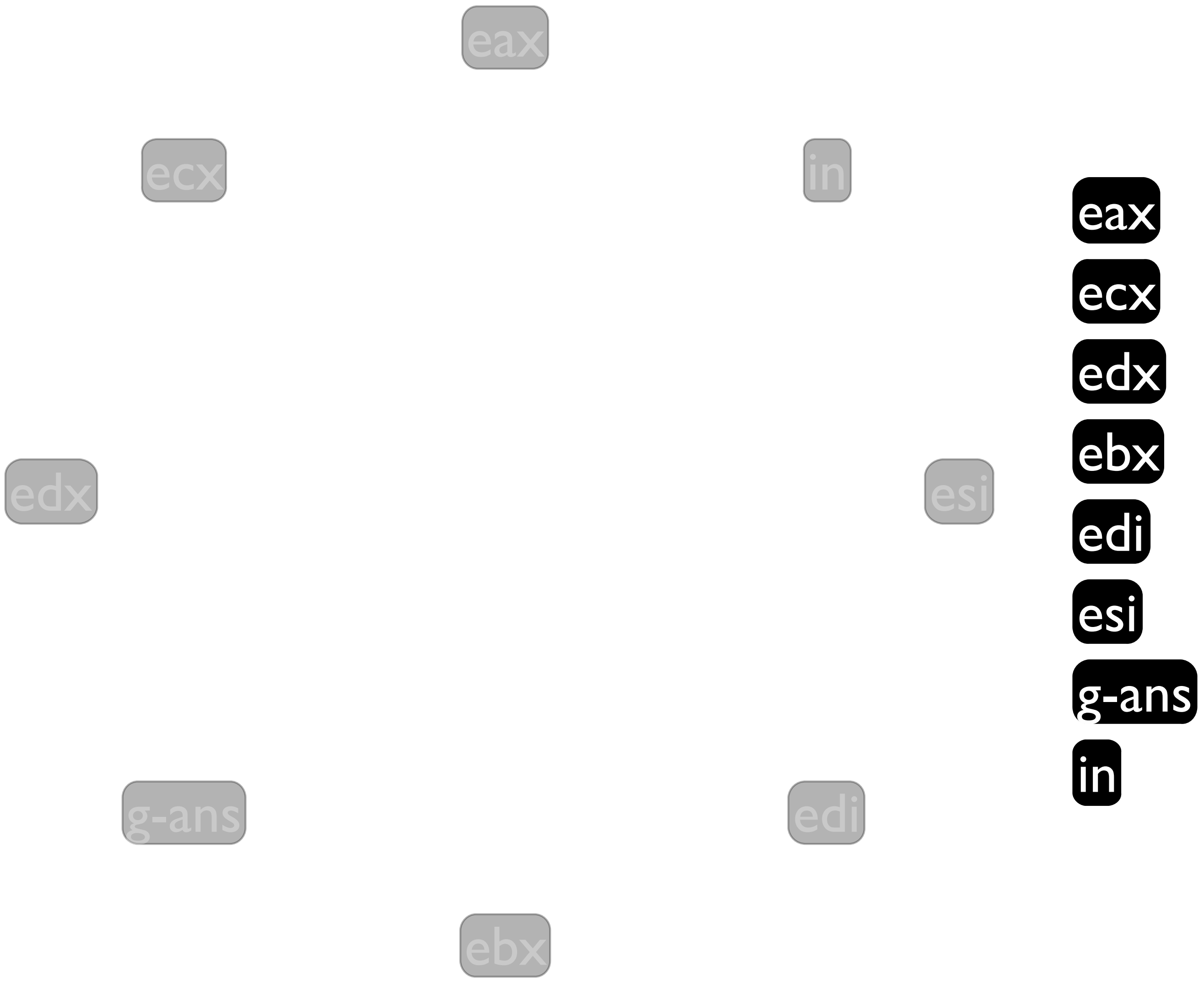

eax
ecx
in
edx
esi
esi
g-ans
in
g-ans
edi
ebx

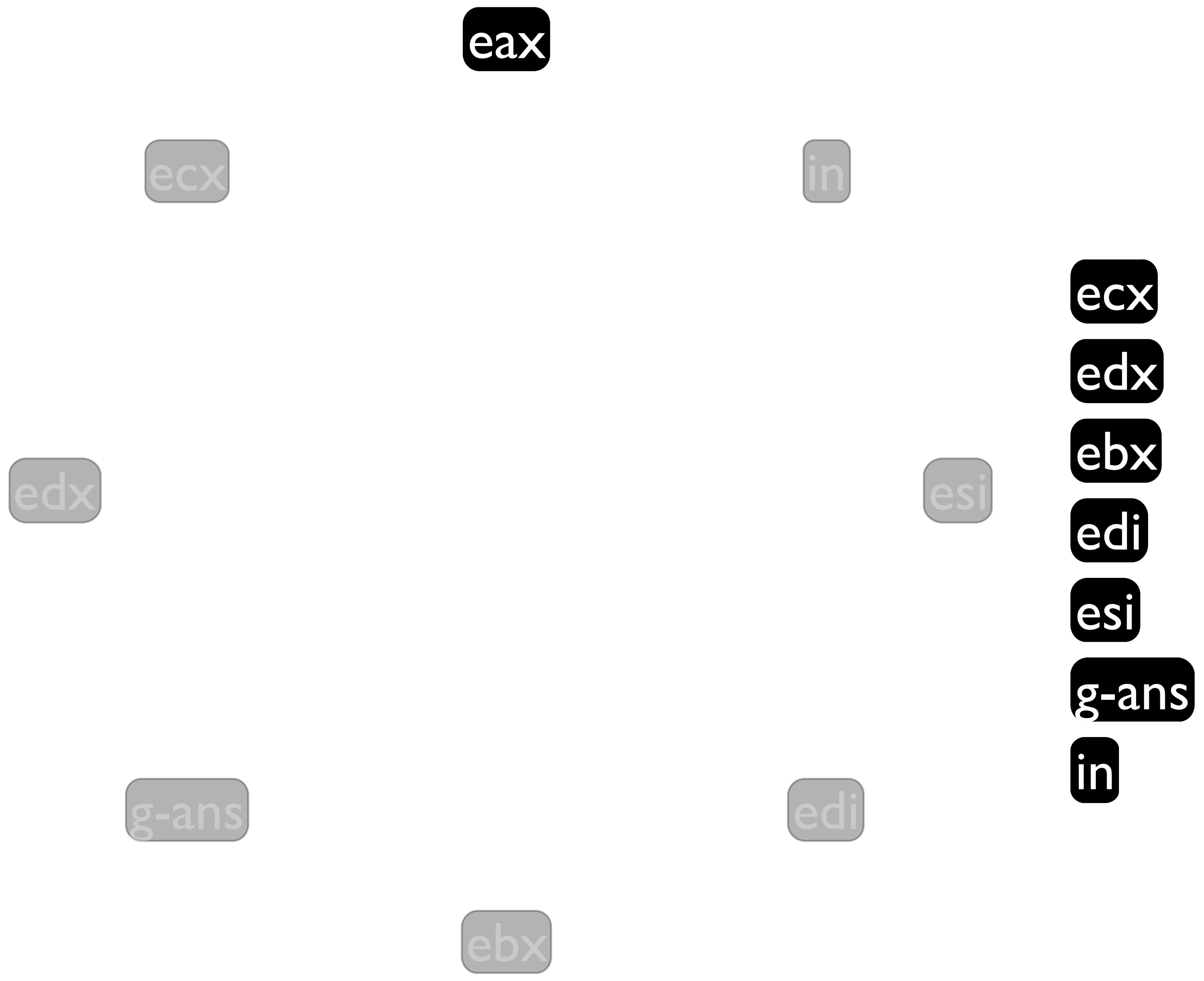

eax
ecx
in
edx
esi
edi
esi
g-ans
in
g-ans
edi
ebx

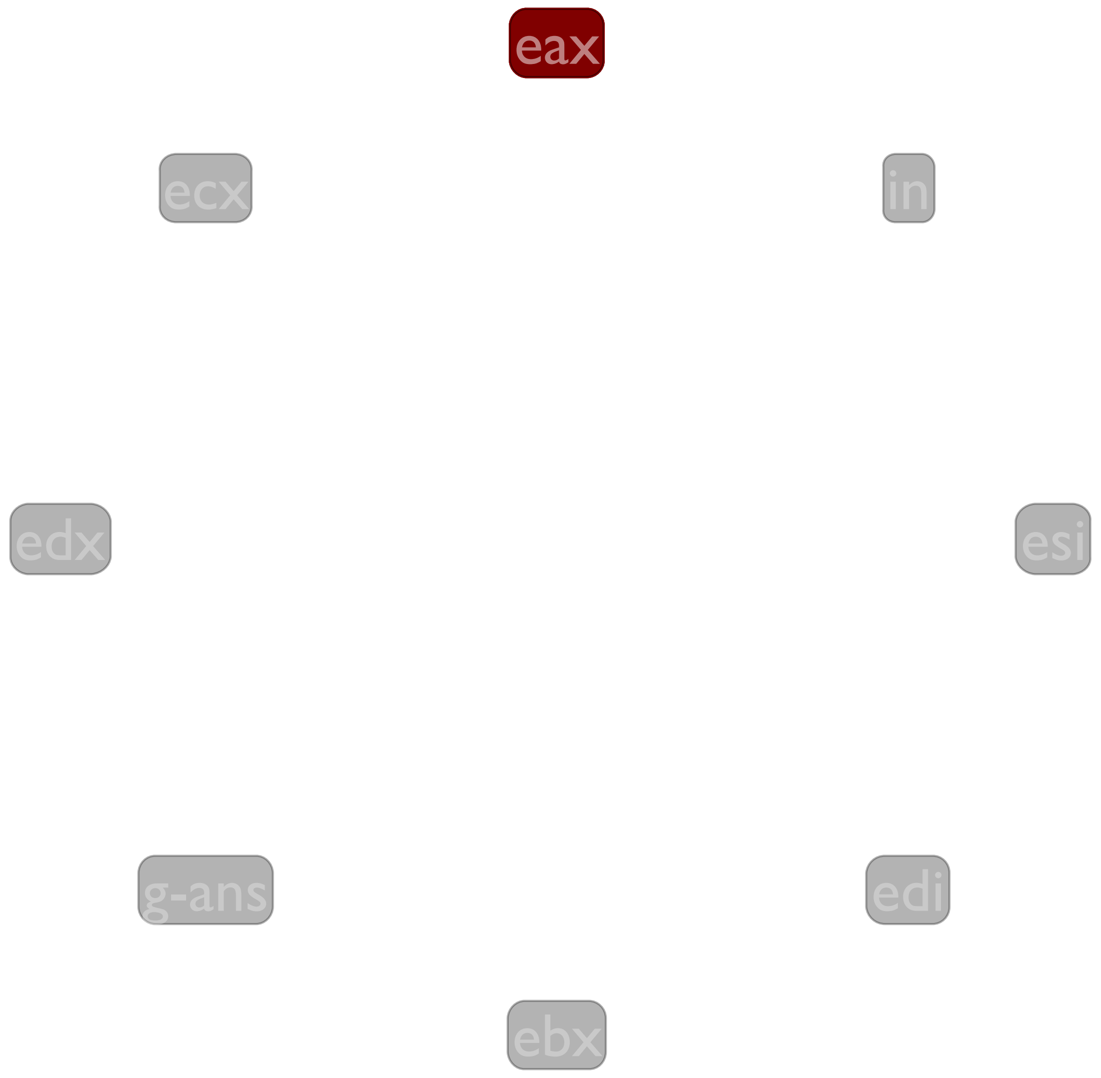

eax
ecx
in
edx
esi
ebx
edi
esi
g-ans
in
g-ans
edi
ebx


eax
ecx
in
edx
esi
g-ans
edi
ebx
edx
ebx
edi
esi
g-ans
in

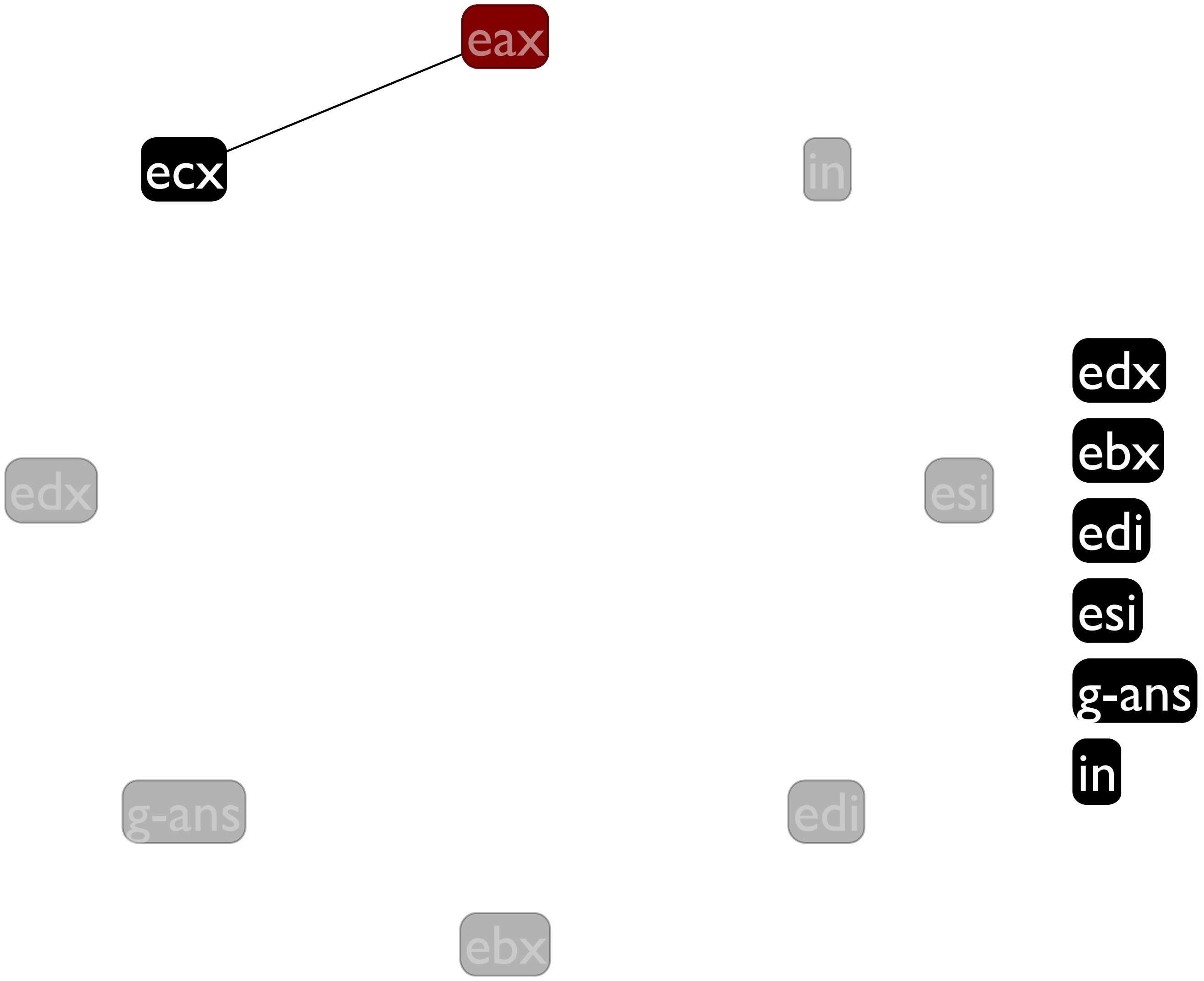

eax
ecx
in
edx
esi
g-ans
edi
ebx
ecx
edx
ebx
edi
esi
g-ans
in

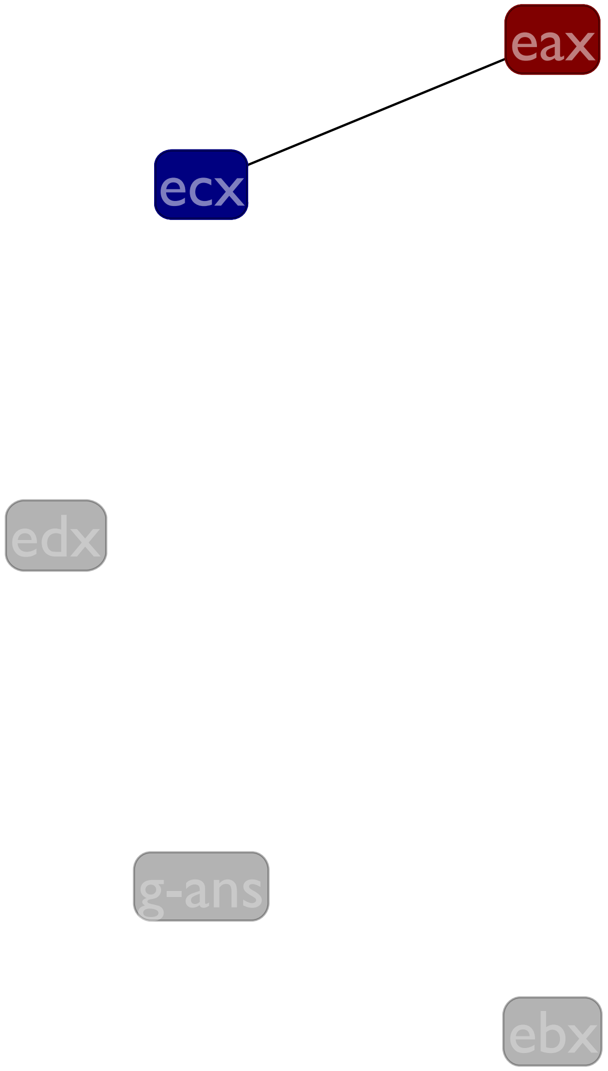

eax
ecx
in
edx
esi
g-ans
edi
ebx
eax
ecx
edx
ebx
edi
esi
g-ans
in


eax
ecx
in
edx
esi
g-ans
edi
ebx
ecx
edx
ebx
edi
esi
g-ans
in



ecx
edx
ebx
edi
esi
g-ans
in


eax
ecx
in
edx
esi
g-ans
edi
ebx
edx
ebx
edi
esi
g-ans
in



edx

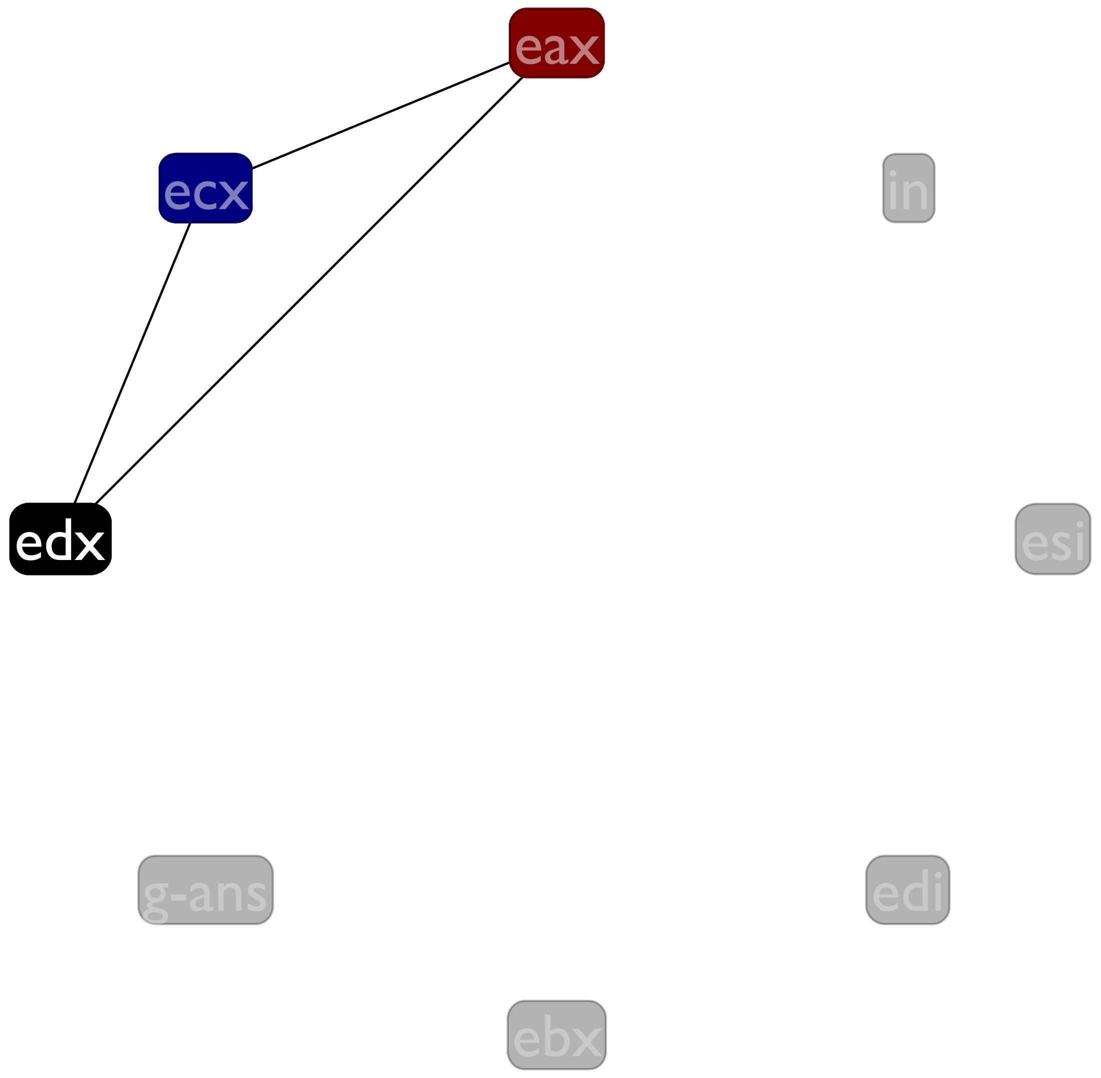
ebx

edi

esi

g-ans

in



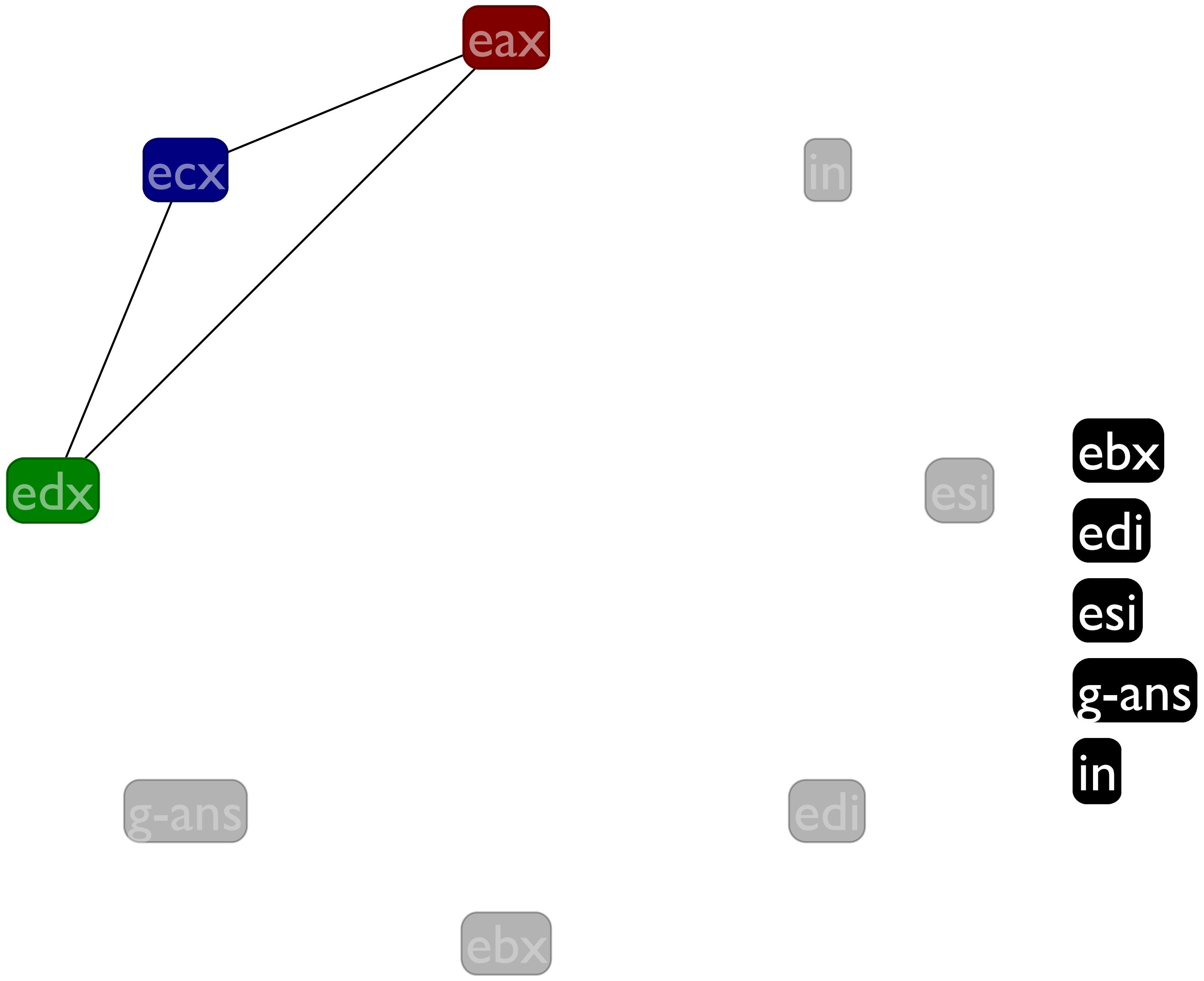
ebx

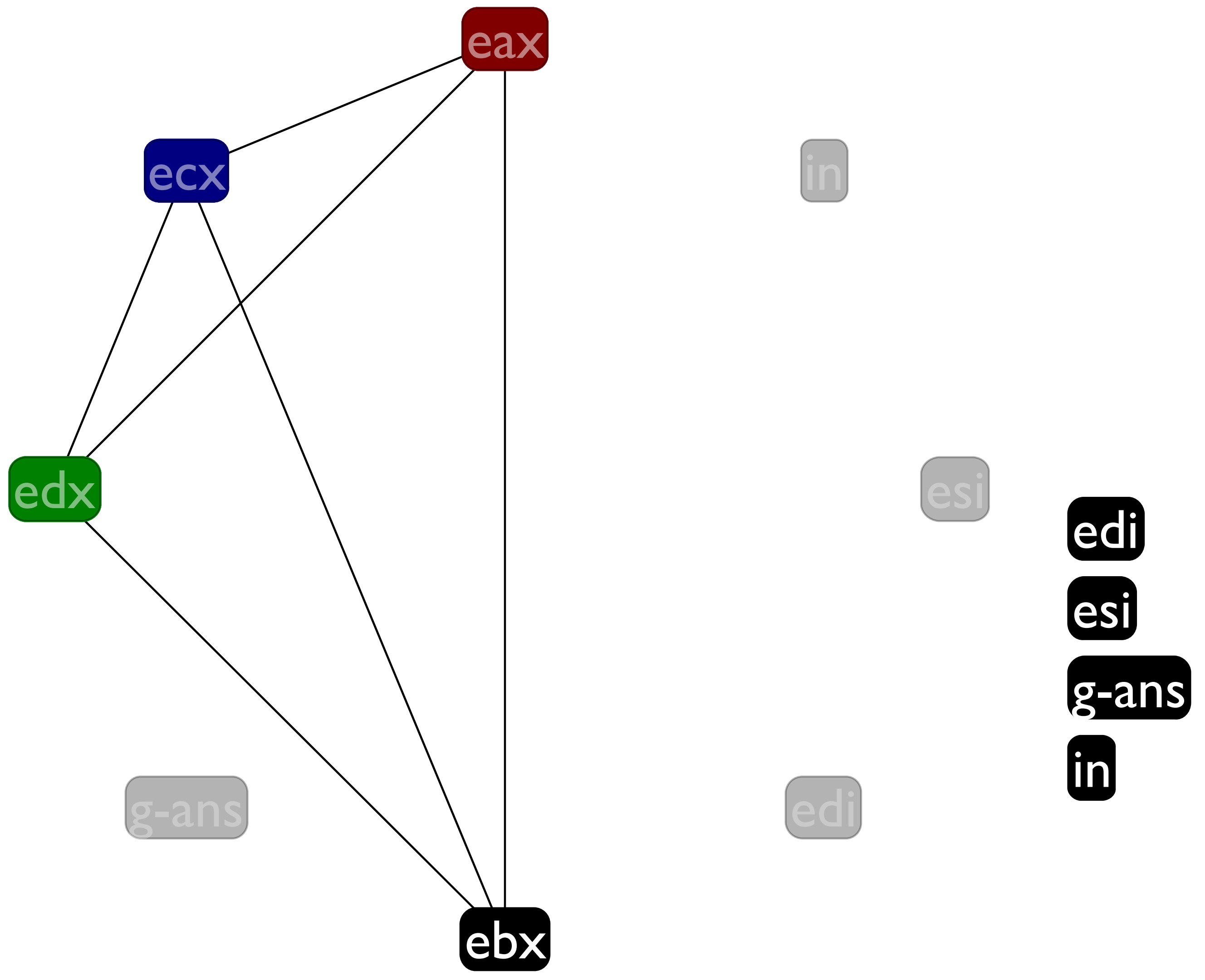
edi

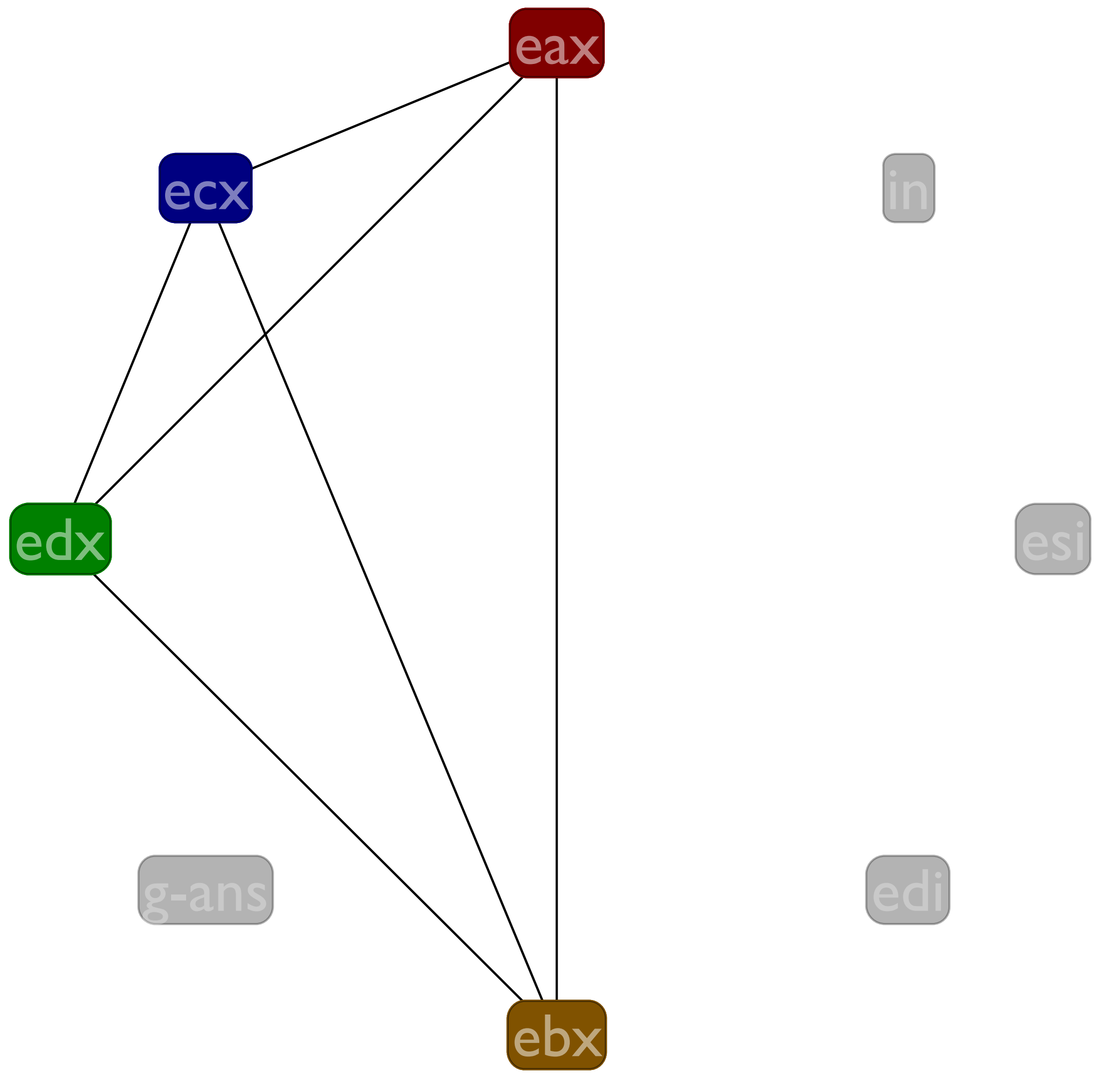
esi

g-ans

in


eax
ecx
in
edx
esi
g-ans
edi
ebx
ebx
edi
esi
g-ans
in


eax
ecx
in
edx
esi
edi
esi
g-ans
in
g-ans
edi
ebx

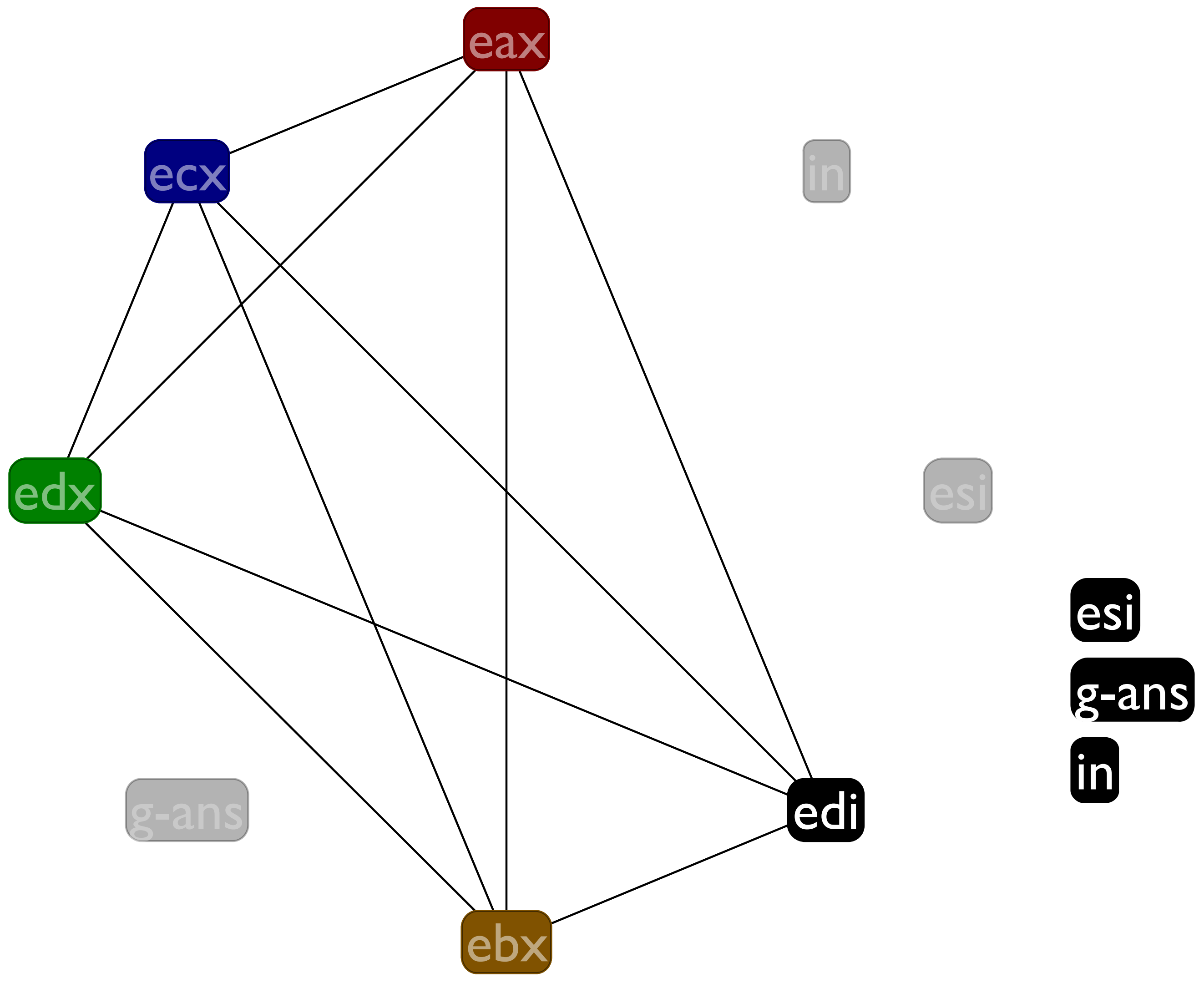


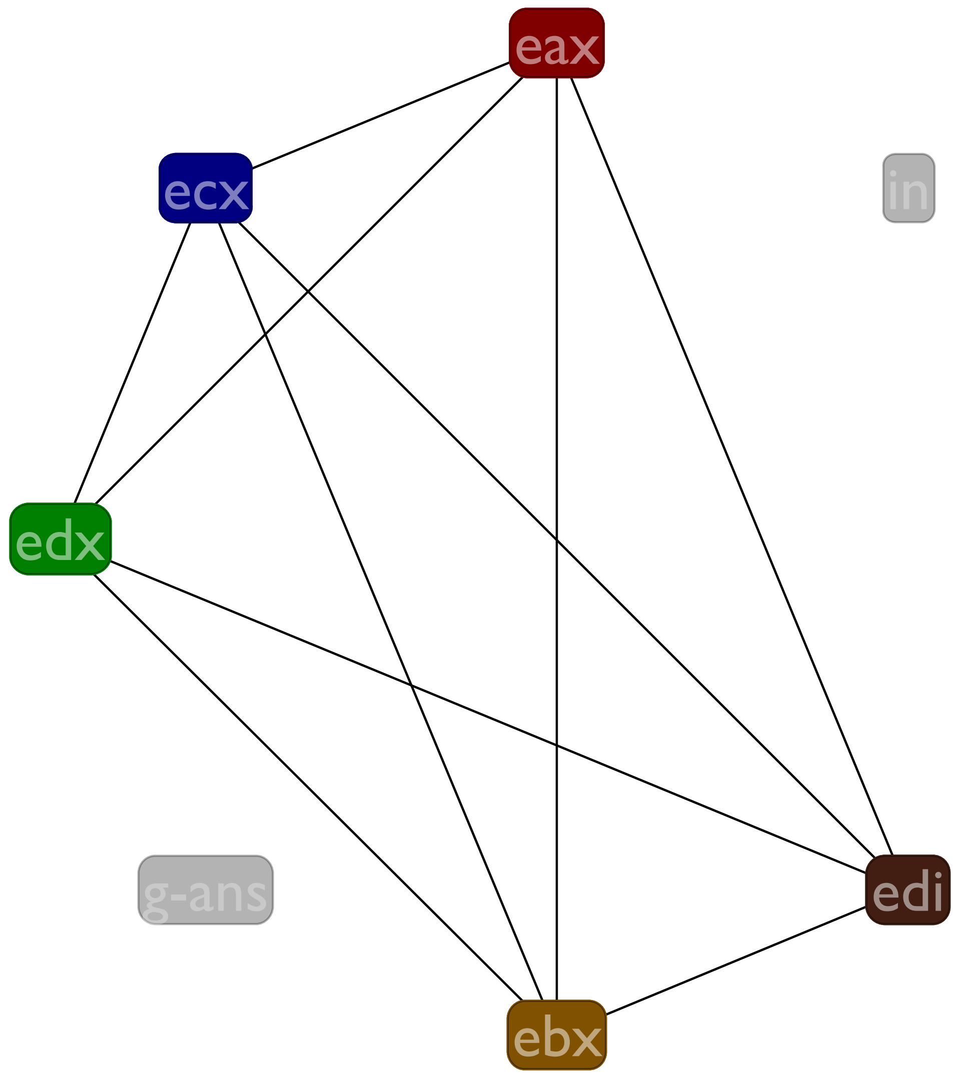
edi

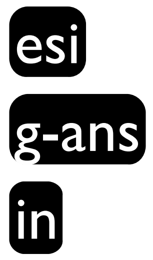
esi

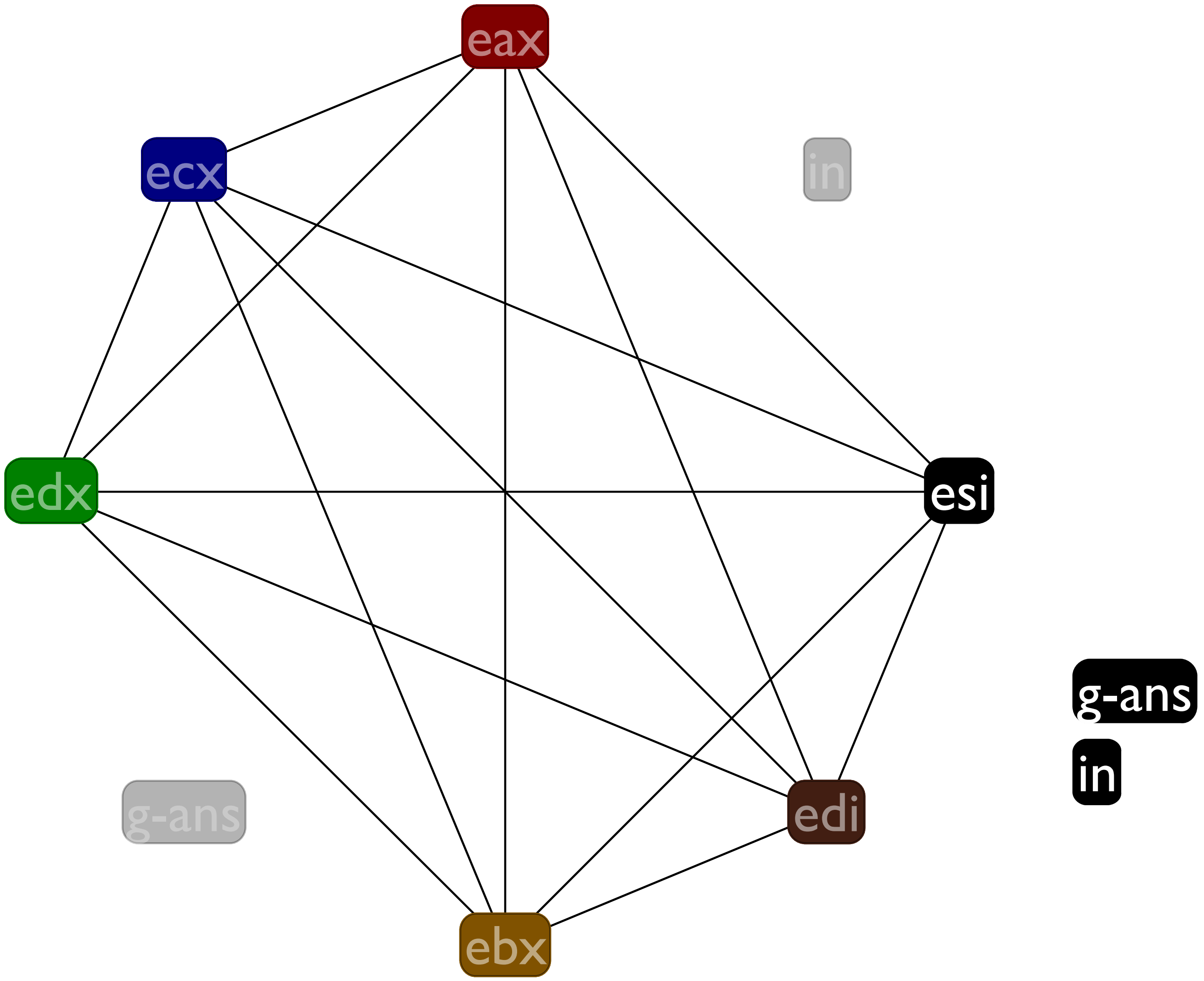
g-ans

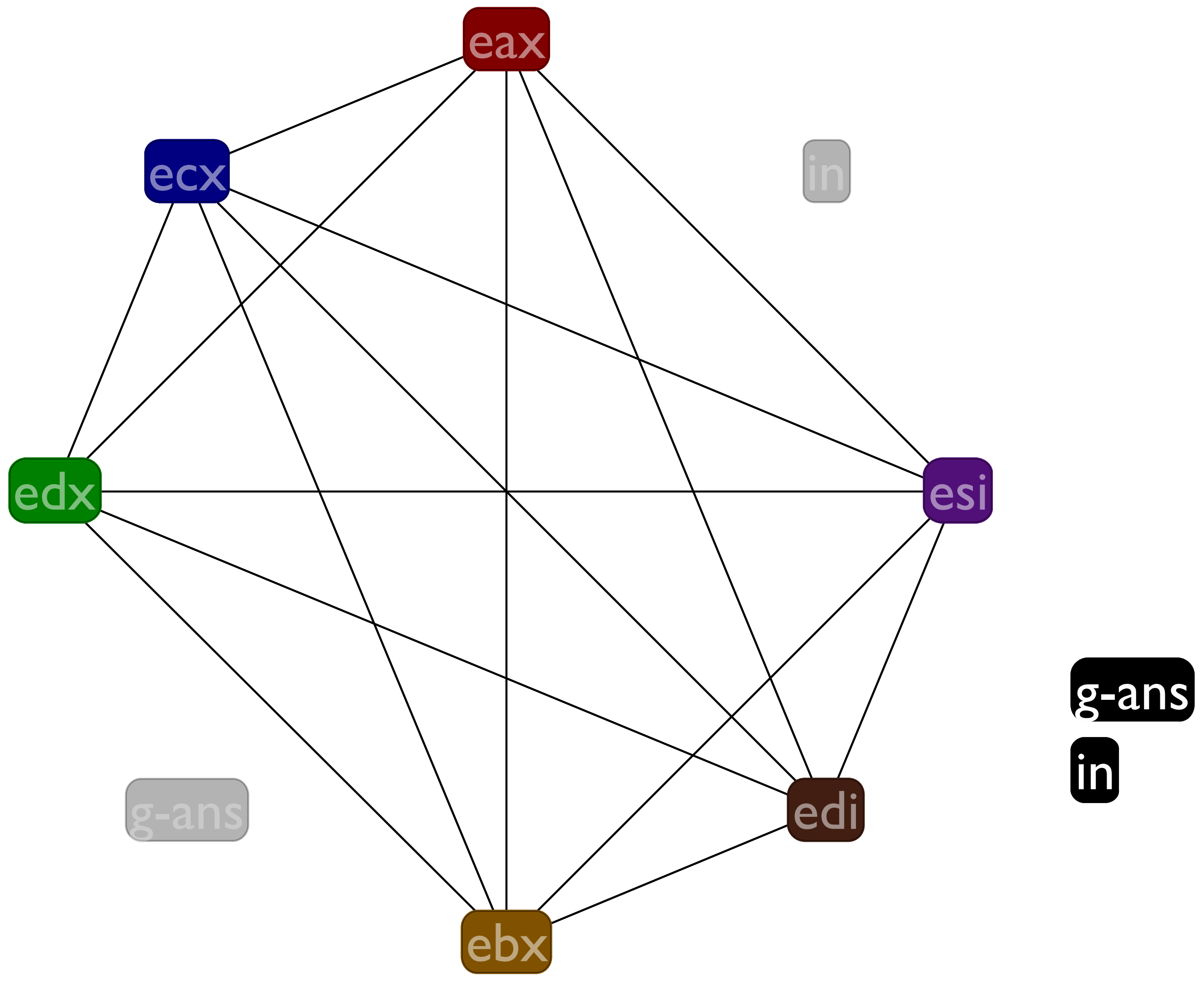
in

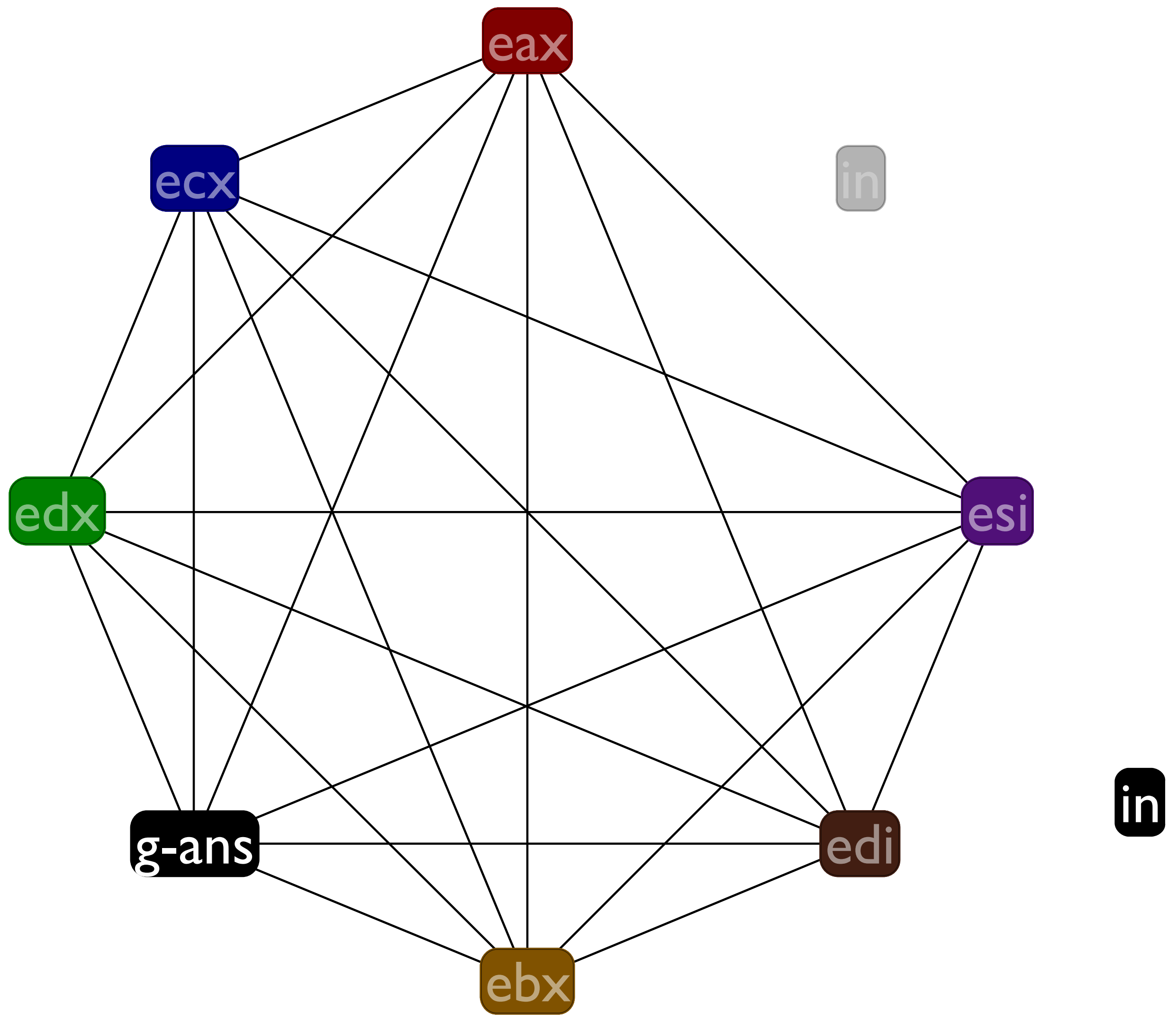

eax
ecx
in
edx
esi
esi
g-ans
in
g-ans
edi
ebx

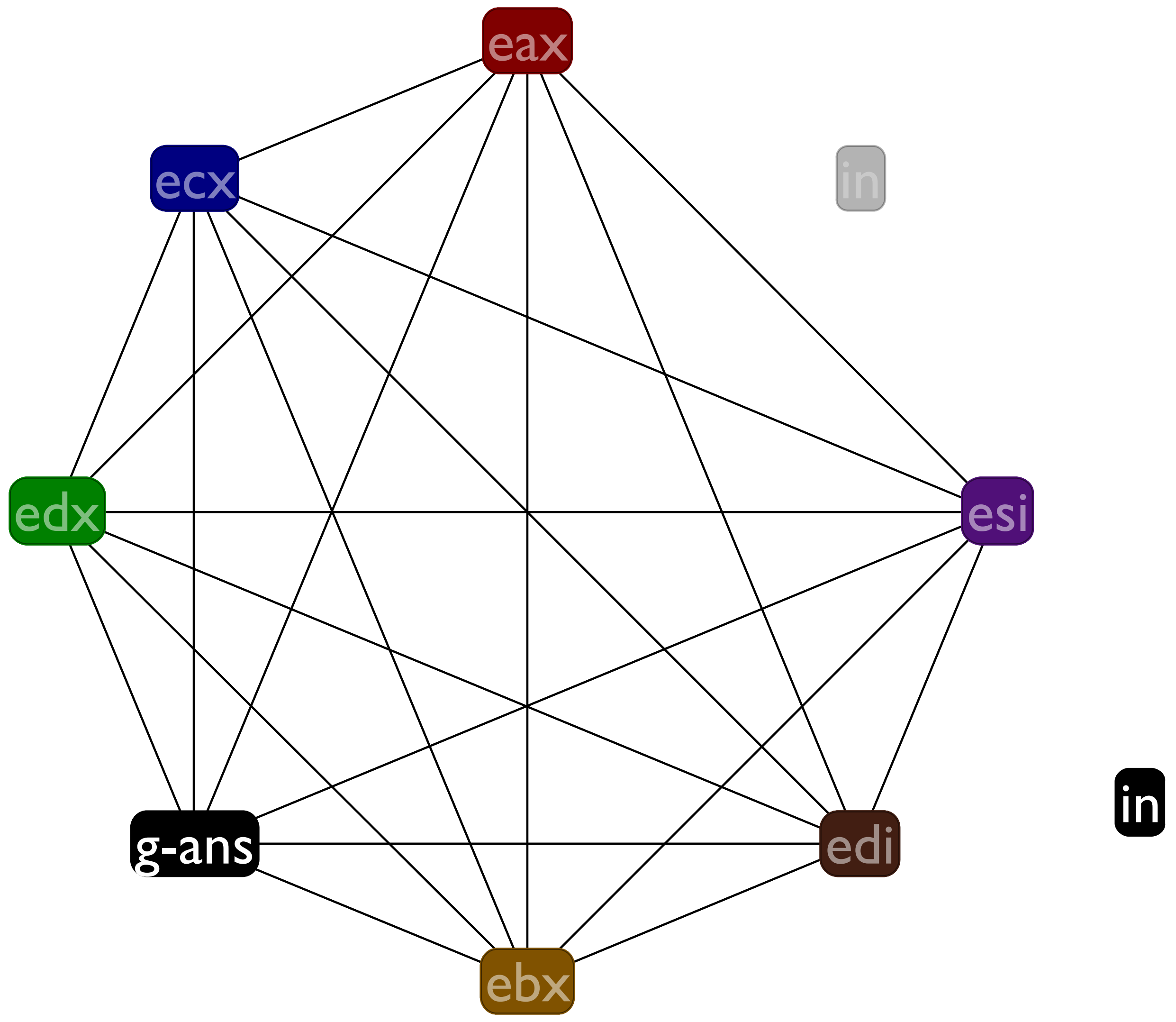

eax
ecx
in
edx
esi
g-ans
edi
ebx


esi
g-ans
in


eax
ecx
in
edx
esi
g-ans
in
g-ans
edi
ebx


eax
ecx
in
edx
esi
g-ans
in
g-ans
edi
ebx


eax
ecx
in
edx
esi
g-ans
edi
in
ebx


eax
ecx
in
edx
esi
in
g-ans
edi
ebx

Spilling

Spilling is a program rewrite to make it easier to allocate registers

- Pick a variable and a location on the stack for it
- Replace all writes to the variable with writes to the stack
- Replace all reads from the variable with reads from the stack

Sometimes that means introducing new temporaries

Spilling Example

Say we want to spill `s` to the location `(mem ebx -4)`.

Two easy cases:

```
(s <- 1)  ⇒  ((mem ebx -4) <- 1)

(x <- s)  ⇒  (x <- (mem ebx -4))
```

A trickier case:

```
(s *= s)  ⇒  (s_new <- (mem ebx -4))
              (s_new *= s_new)
              ((mem ebx -4) <- s_new)
```

In general, make up a new temporary for each instruction that uses the variable to be spilled

This makes for very short live ranges

Spilling in

Before:

```
:f
(in <- edx)
(call :g)
(edx <- in)
(g-ans <- eax)
(call :h)
(eax += g-ans)
(return)
```

After:

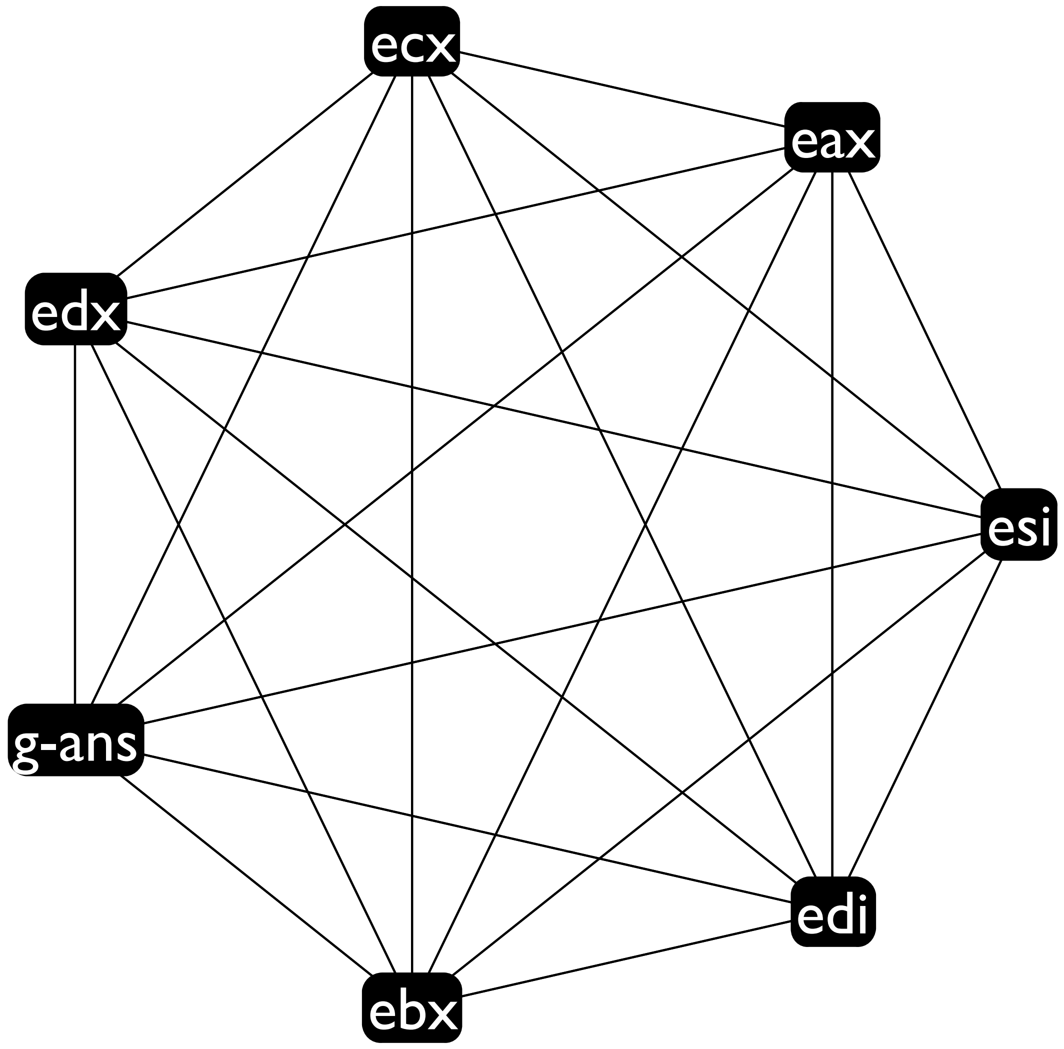
```
:f
((mem ebp -4) <- edx)
(call :g)
(edx <- (mem ebp -4))
(g-ans <- eax)
(call :h)
(eax += g-ans)
(return)
```

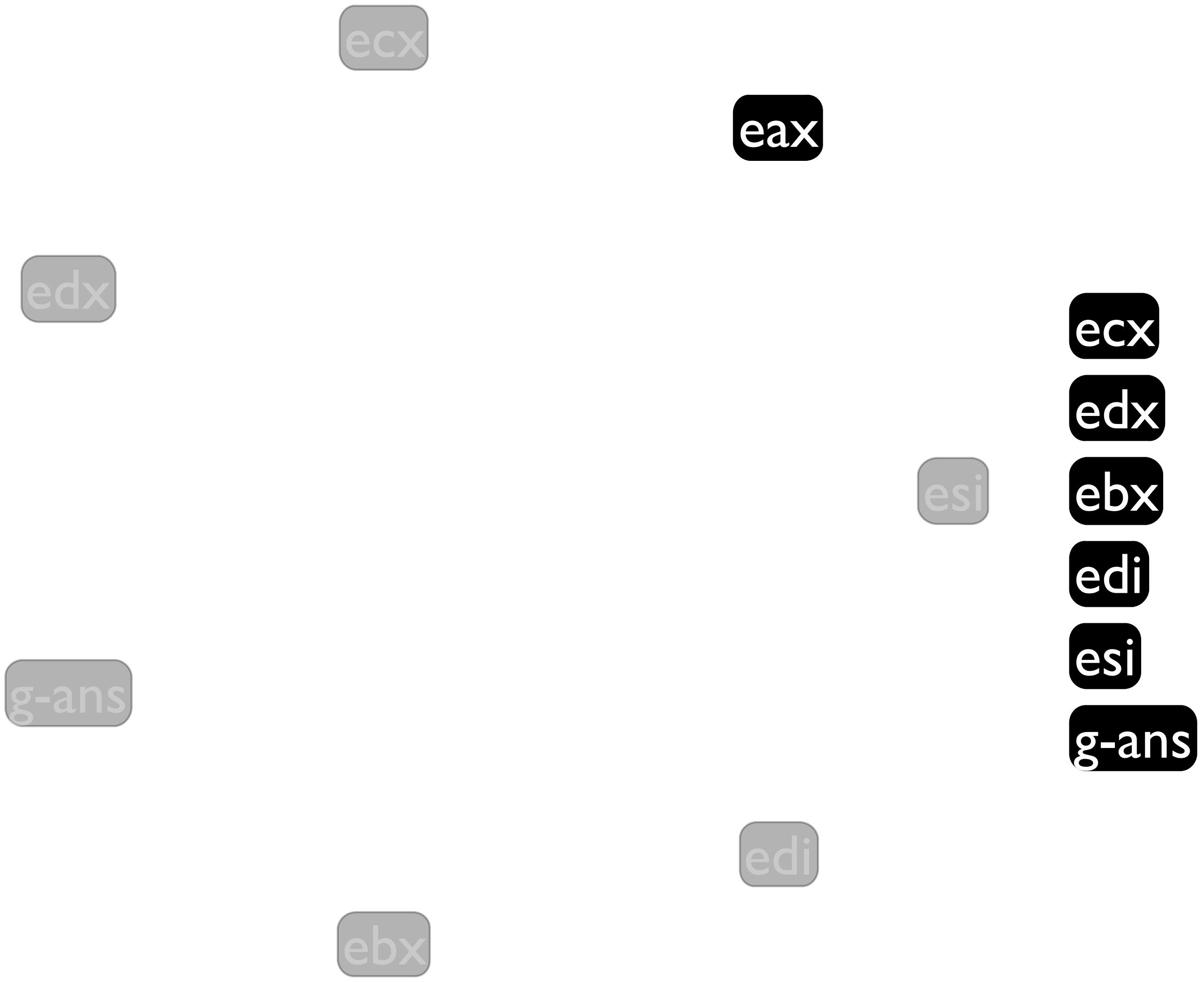
Spilling in

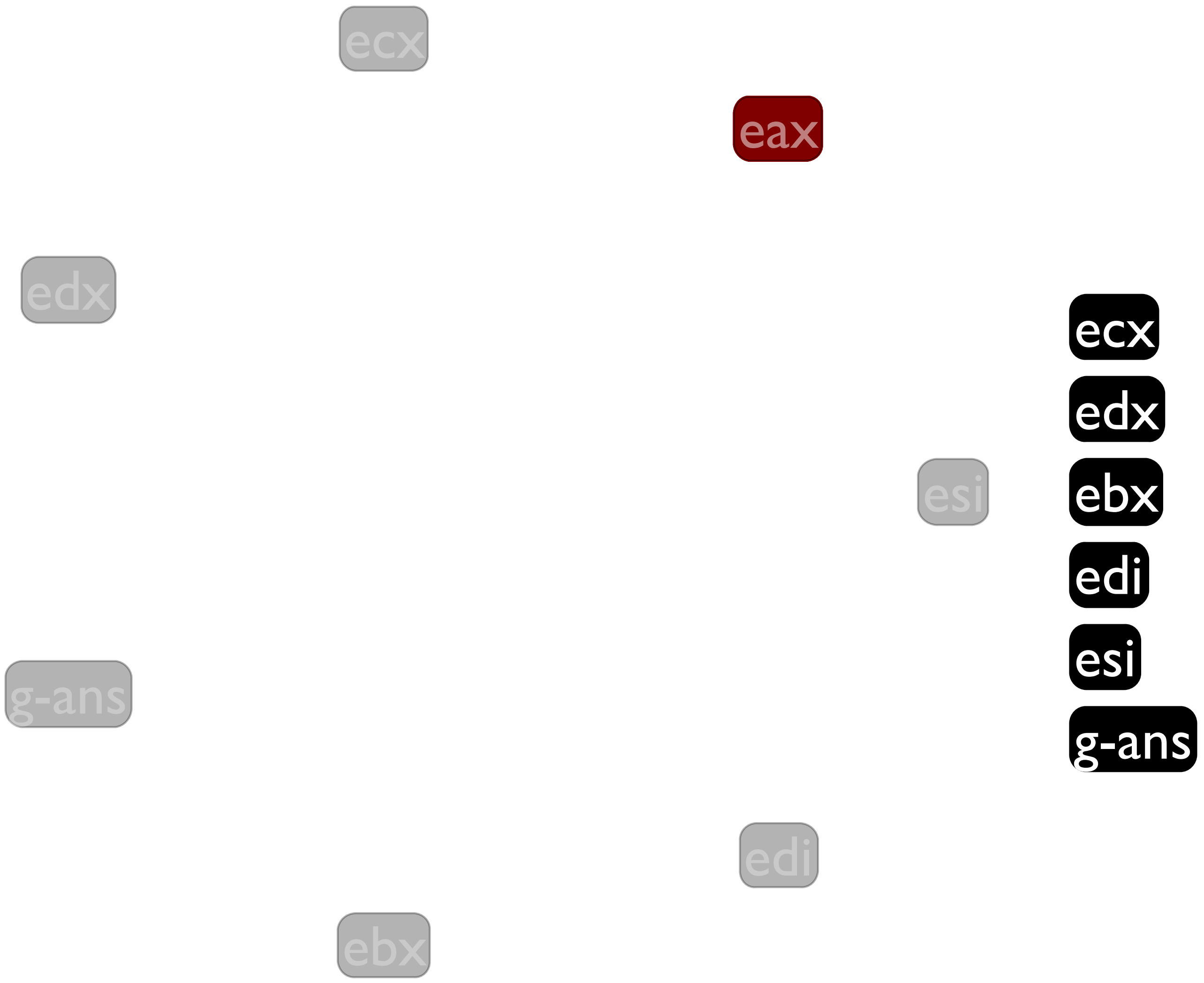
		in	out
1:	`:f`	()	()
2:	`((mem ebp -4) <- edx)`	()	()
3:	`(call :g)`	()	()
4:	`(edx <- (mem ebp -4))`	()	()
5:	`(g-ans <- eax)`	()	()
6:	`(call :h)`	()	()
7:	`(eax += g-ans)`	()	()
8:	`(return)`	()	()

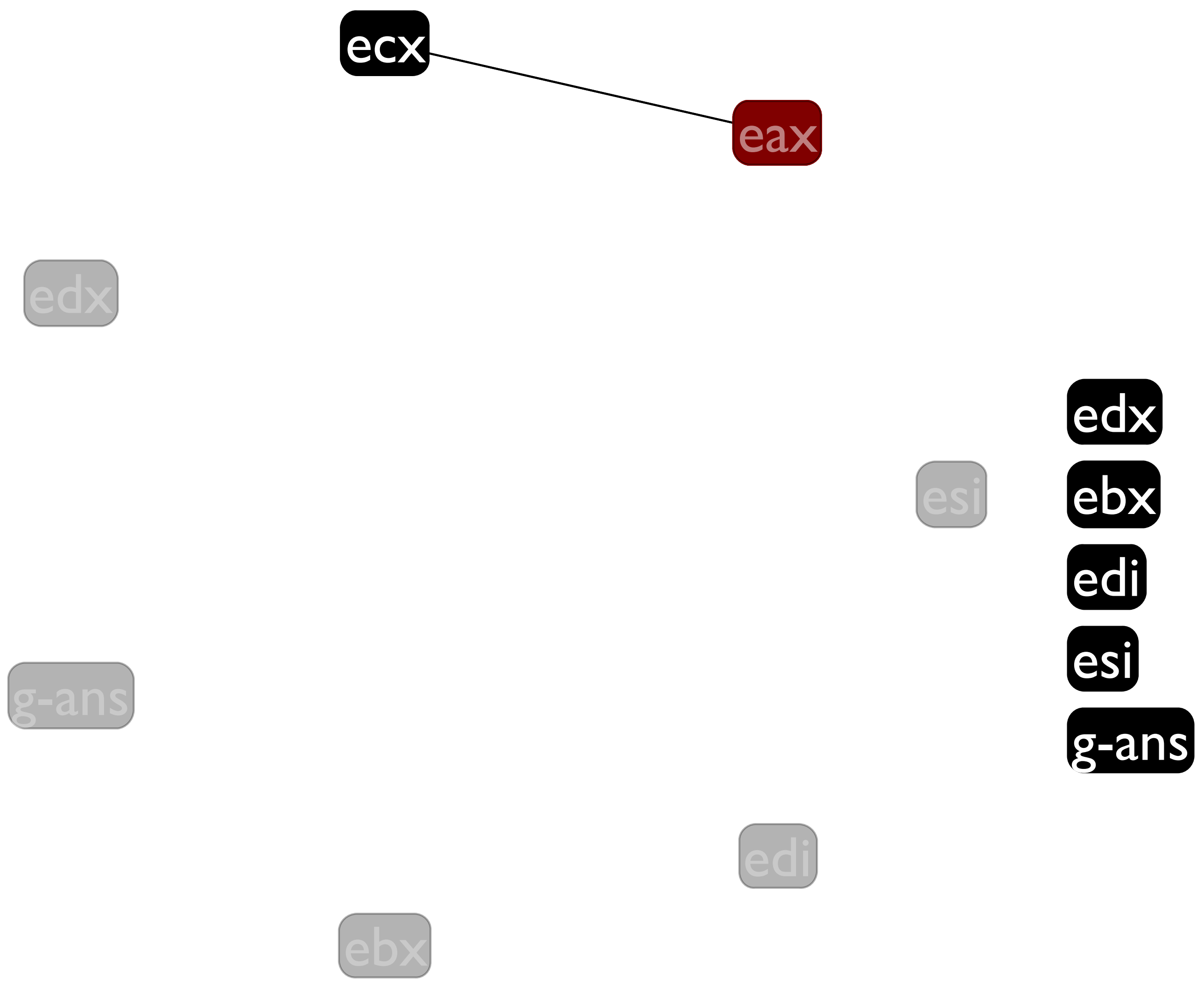
Spilling in

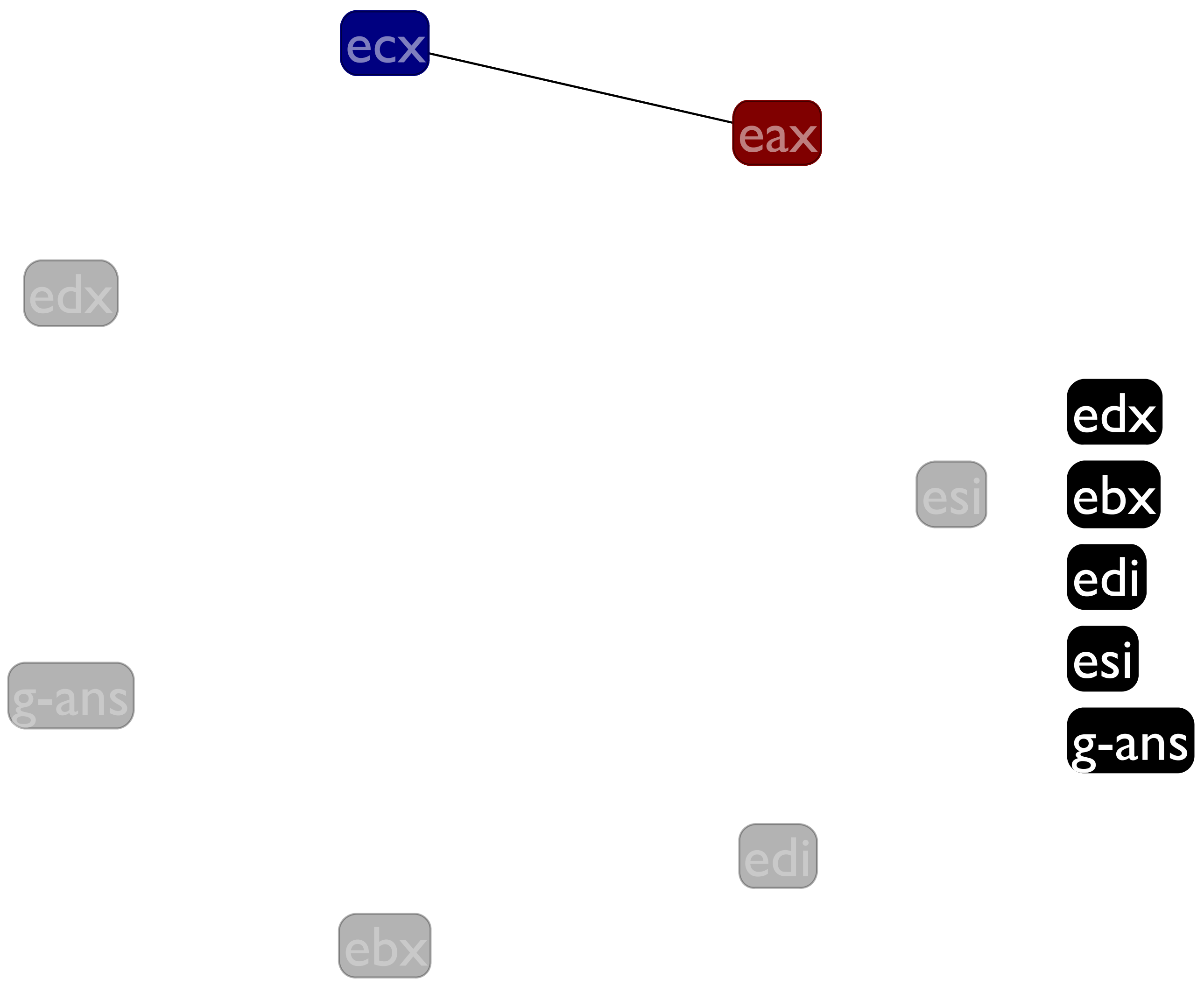
		in	out
1:	`:f`	(ebp ebx edi edx esi)	(ebp ebx edi edx esi)
2:	`((mem ebp -4) <- edx)`	(ebp ebx edi edx esi)	(ebp ebx edi esi)
3:	`(call :g)`	(ebp ebx edi esi)	(eax ebp ebx edi esi)
4:	`(edx <- (mem ebp -4))`	(eax ebp ebx edi esi)	(eax ebx edi esi)
5:	`(g-ans <- eax)`	(eax ebx edi esi)	(ebx edi esi g-ans)
6:	`(call :h)`	(ebx edi esi g-ans)	(eax ebx edi esi g-ans)
7:	`(eax += g-ans)`	(eax ebx edi esi g-ans)	(ebx edi esi)
8:	`(return)`	(ebx edi esi)	()

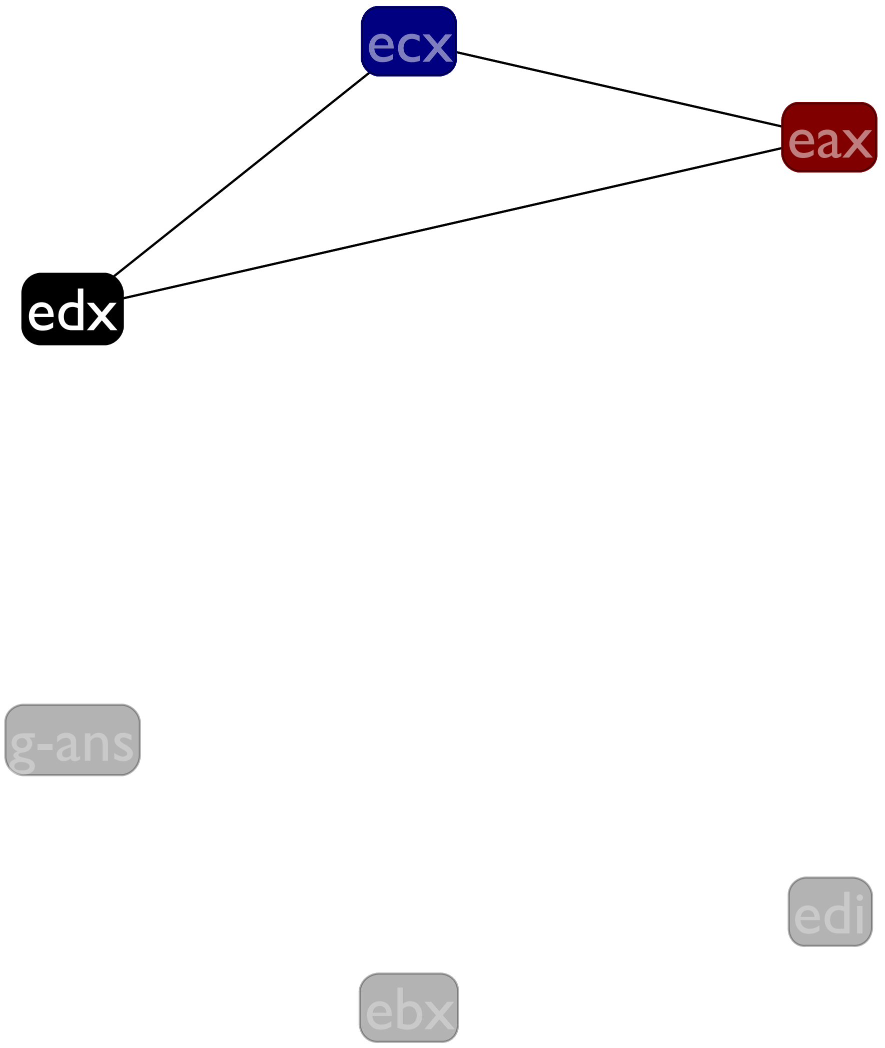

ecx
eax
edx
esi
g-ans
edi
ebx


ecx
eax
edx
ecx
edx
esi
ebx
edi
esi
g-ans
g-ans
edi
ebx


ecx
eax
edx
esi
g-ans
edi
ebx
ecx
edx
ebx
edi
esi
g-ans


ecx
eax
edx
esi
g-ans
edi
ebx
edx
ebx
edi
esi
g-ans


ecx
eax
edx
esi
g-ans
edi
ebx
edx
ebx
edi
esi
g-ans

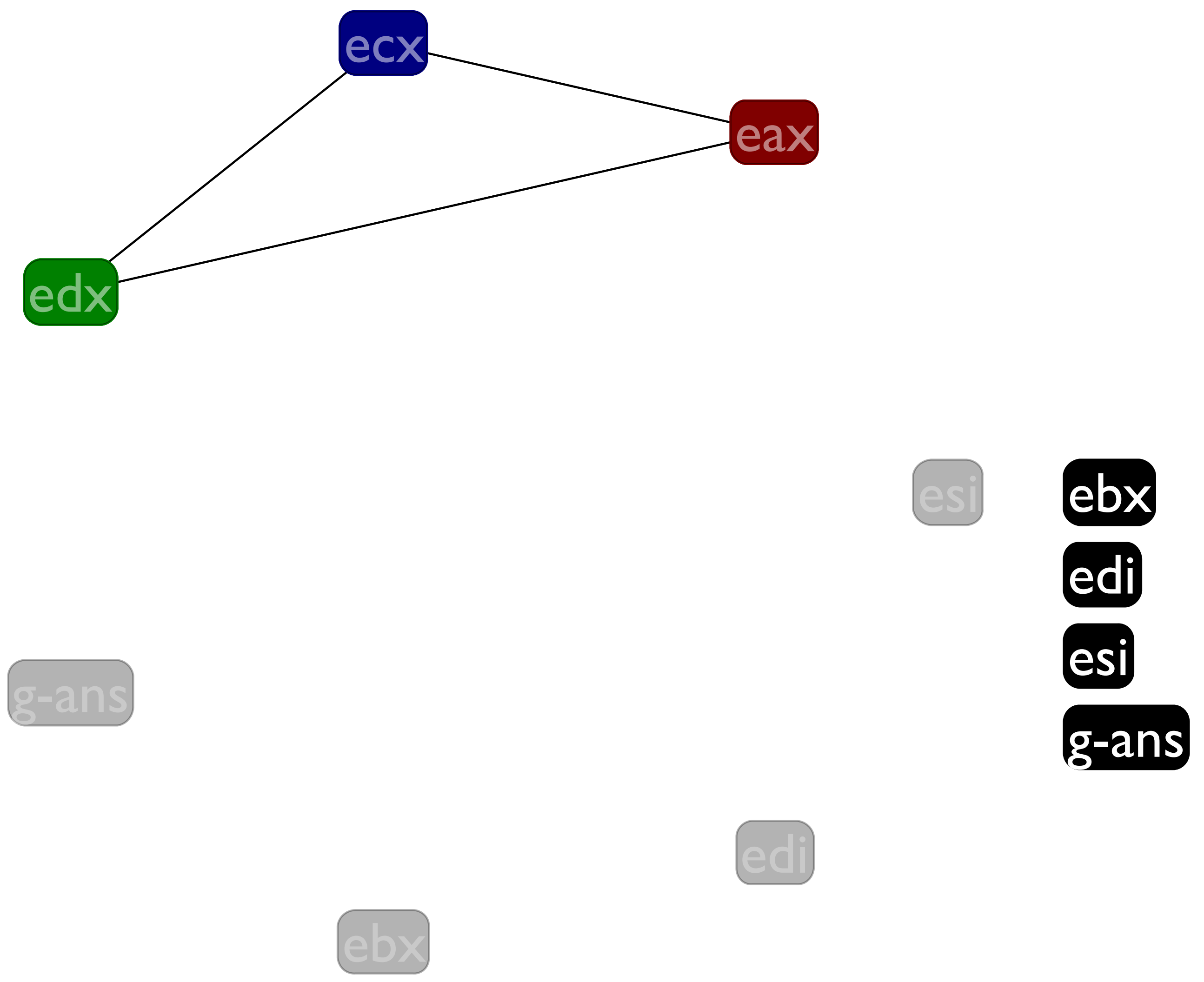


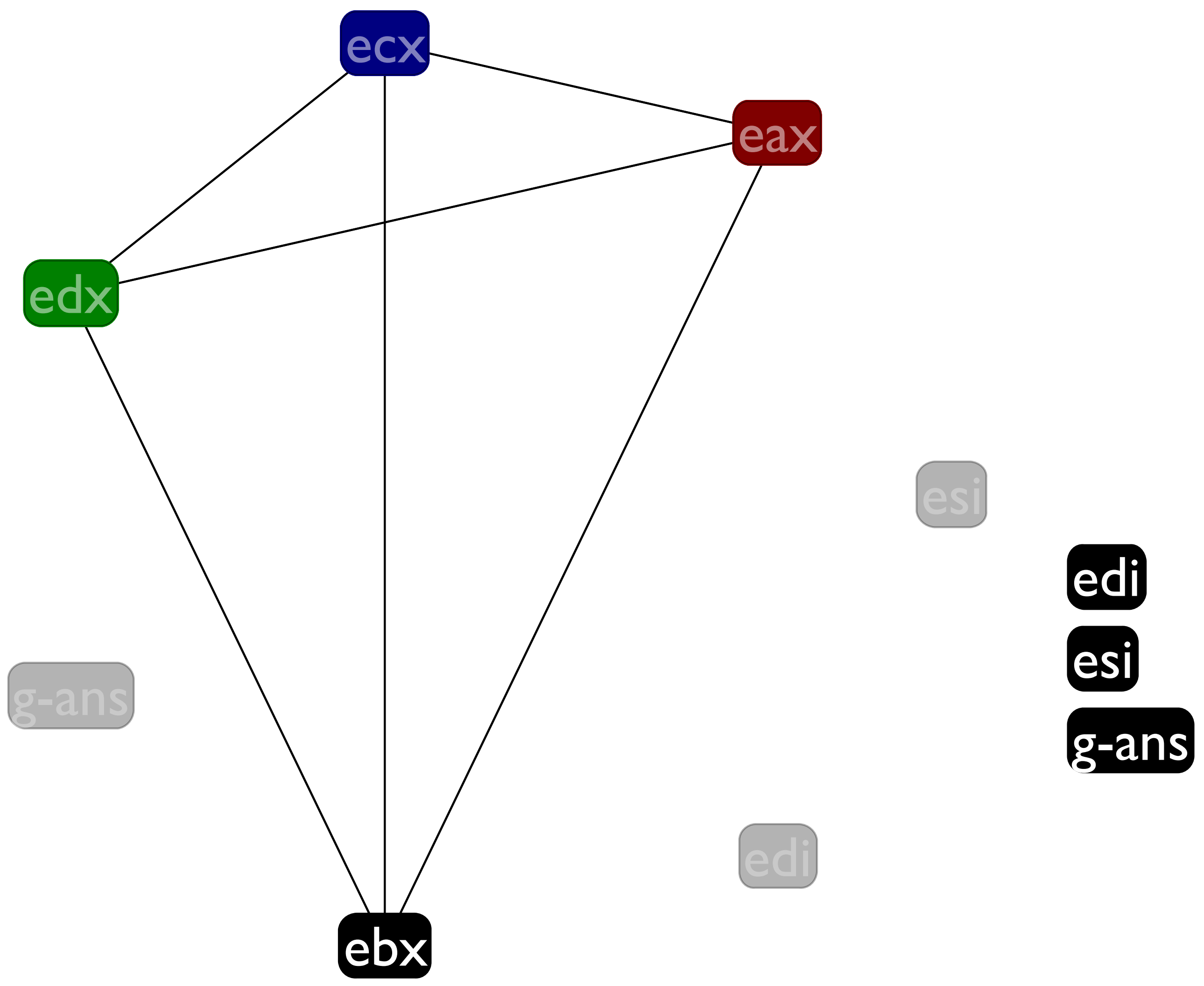
ebx

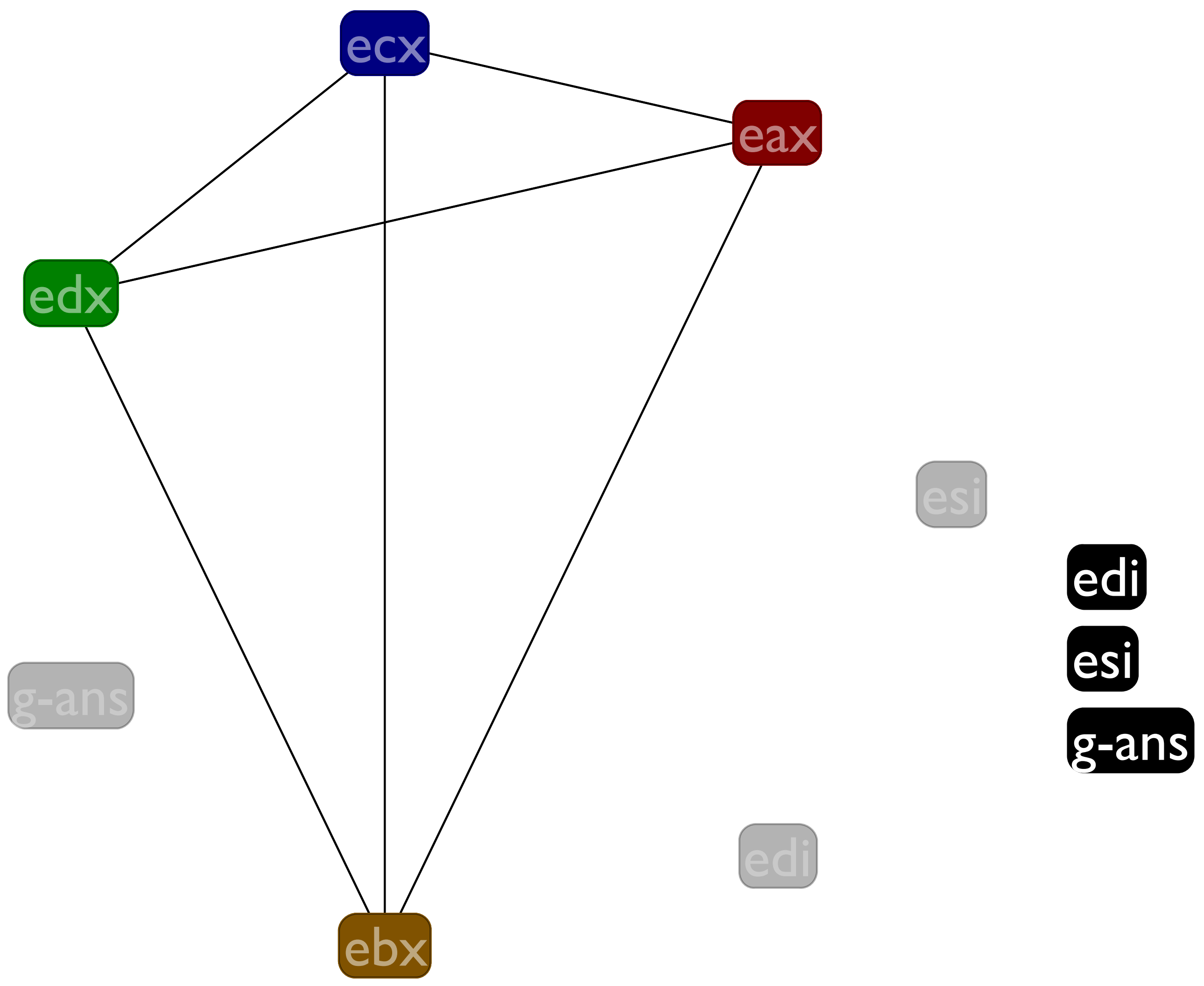
edi

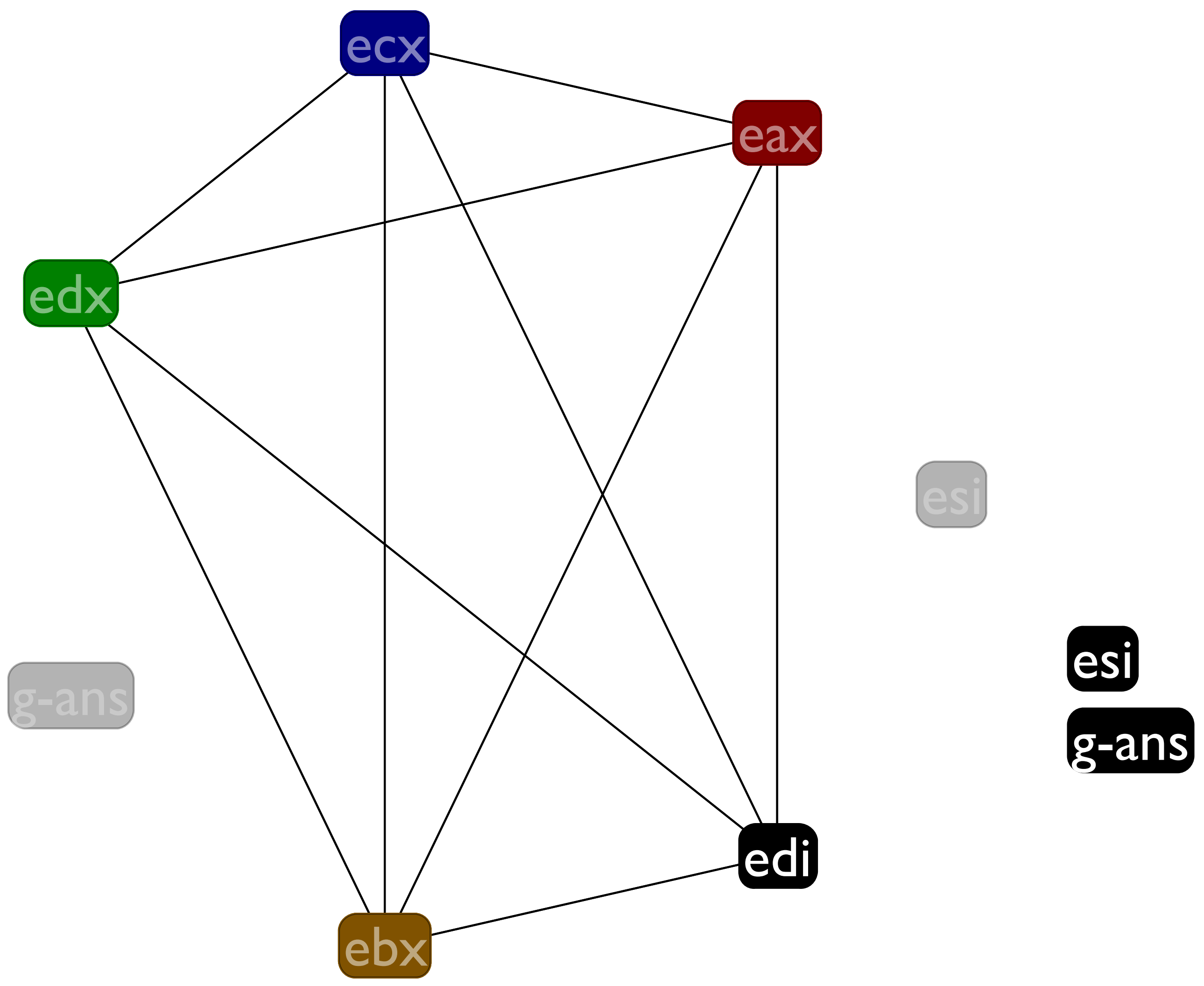
esi

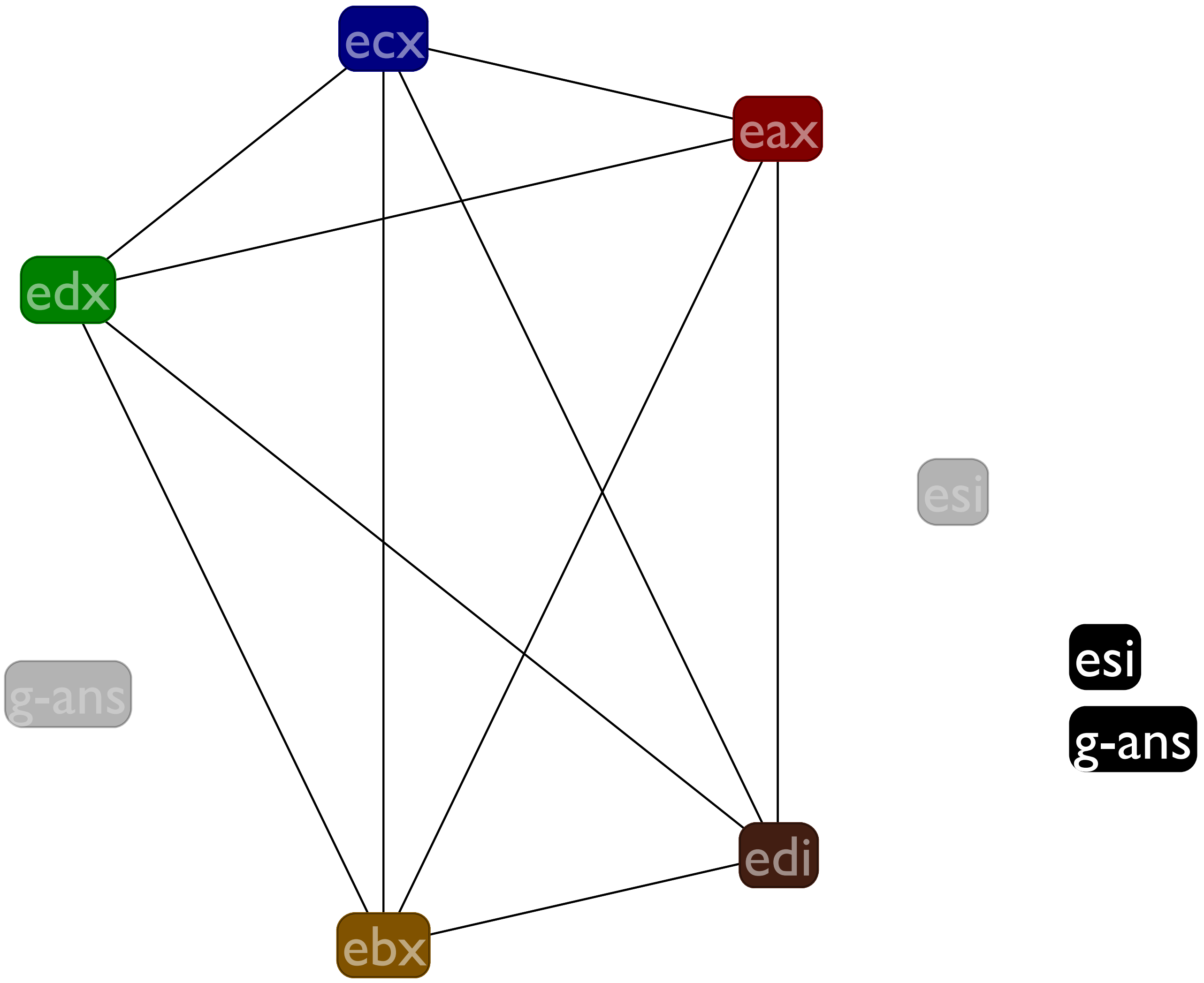
g-ans

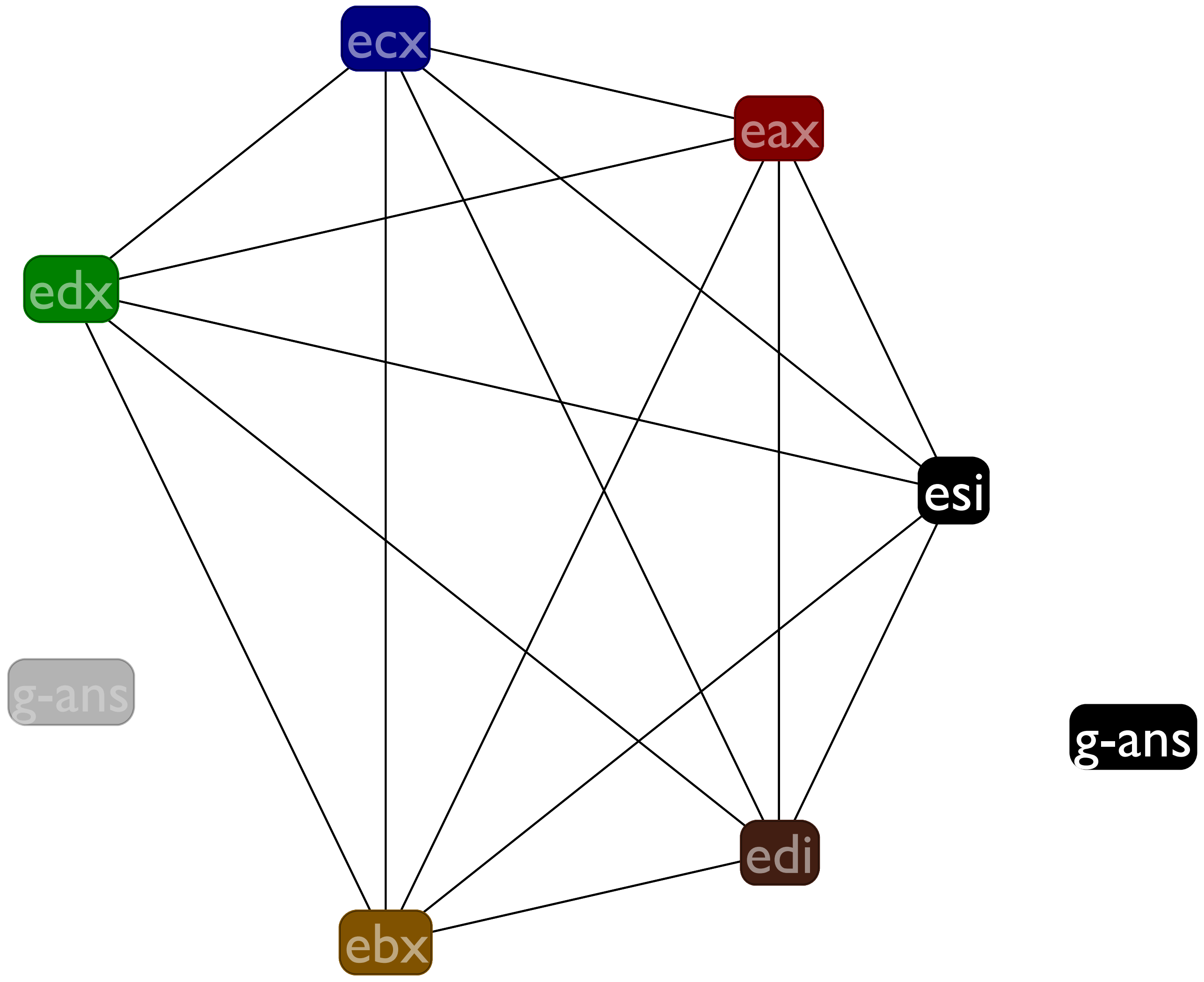

ecx
eax
edx
esi
ebx
edi
esi
g-ans
g-ans
edi
ebx

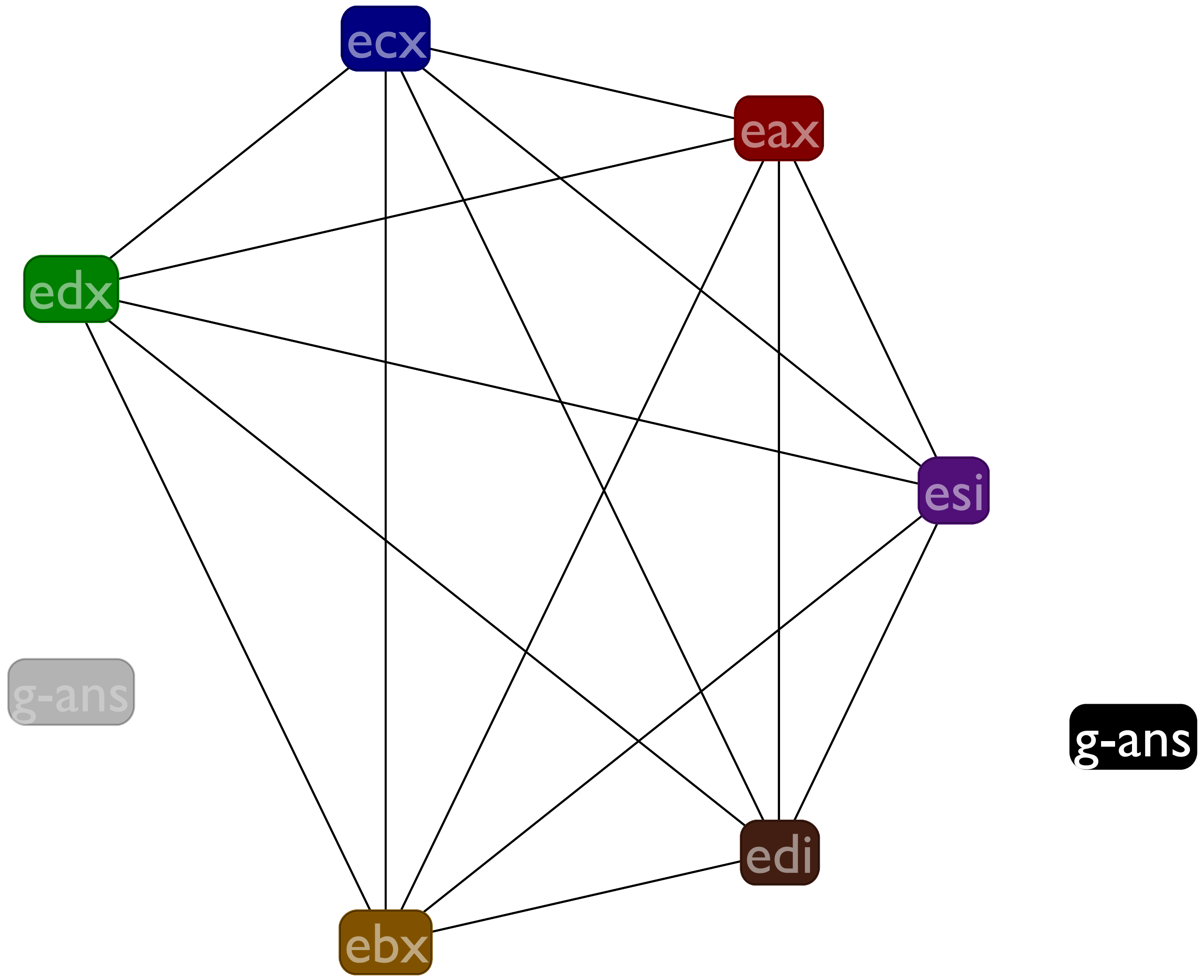

ecx
eax
edx
esi
edi
esi
g-ans
g-ans
edi
ebx

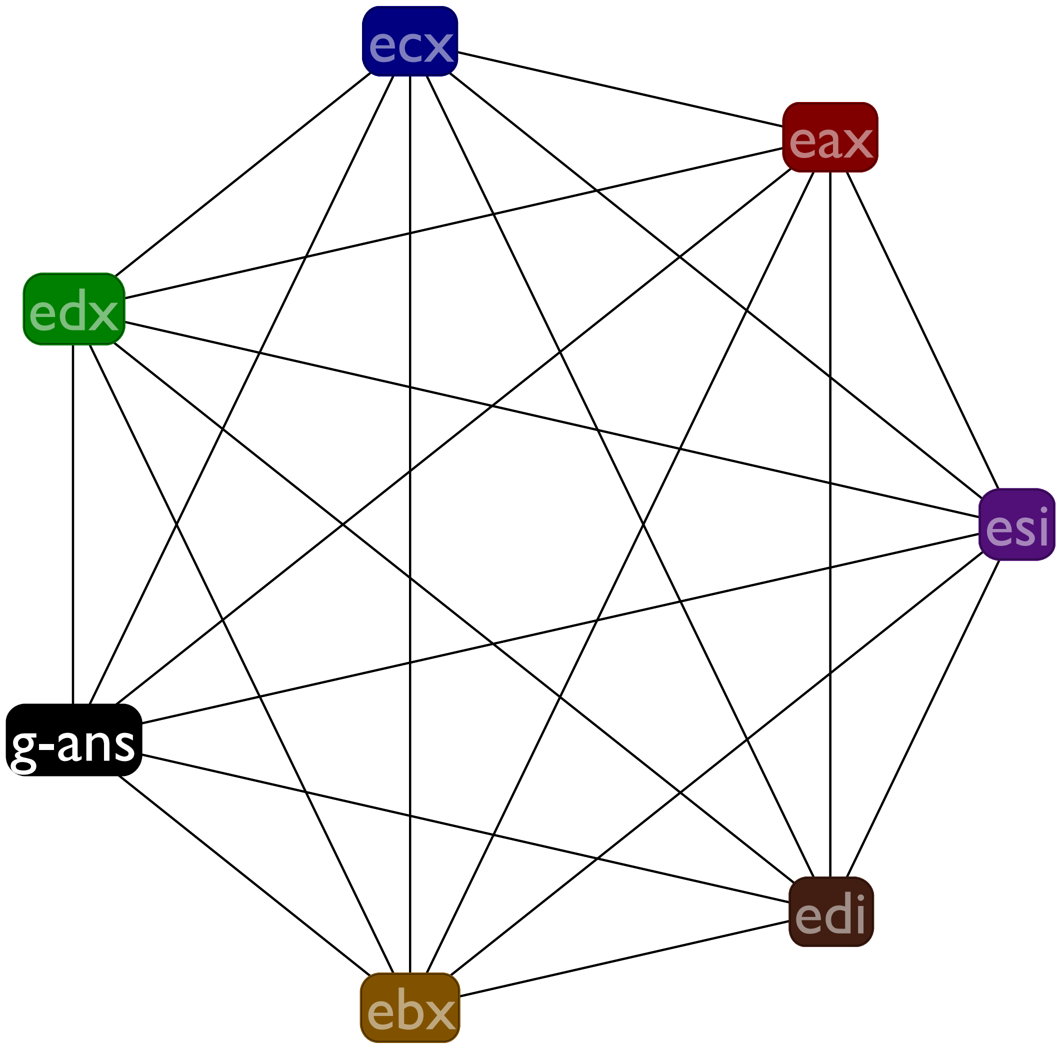

ecx
eax
edx
esi
edi
esi
g-ans
g-ans
edi
ebx

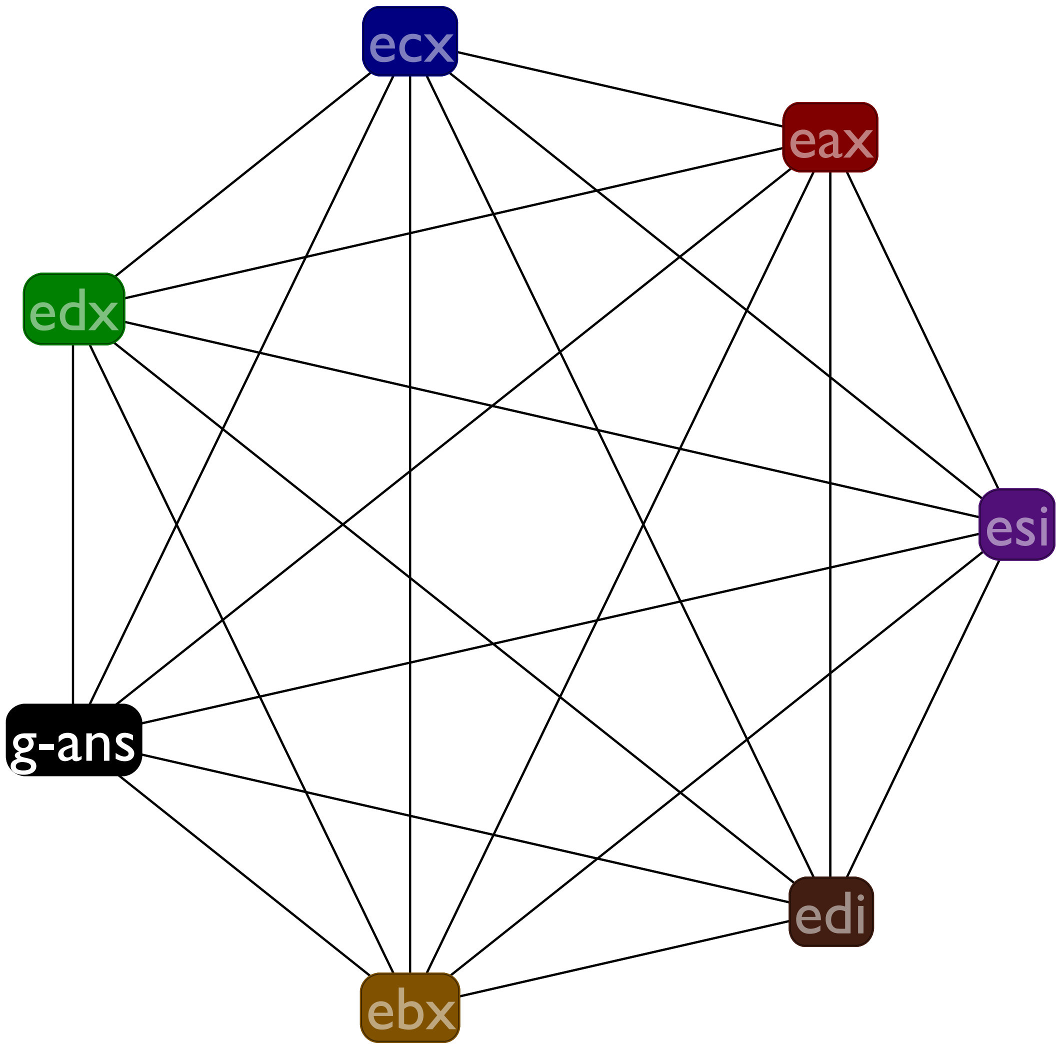

ecx
eax
edx
esi
g-ans
esi
g-ans
edi
ebx


ecx
eax
edx
esi
esi
g-ans
g-ans
edi
ebx


ecx
eax
edx
esi
g-ans
g-ans
edi
ebx


ecx
eax
edx
esi
g-ans
g-ans
edi
ebx


ecx
eax
edx
esi
g-ans
edi
ebx


ecx
eax
edx
esi
g-ans
edi
ebx

Spilling g-ans

Before:

```
:f
((mem ebp -4) <- edx)
(call :g)
(edx <- (mem ebp -4))
(g-ans <- eax)
(call :h)
(eax += g-ans)
(return)
```

After:

```
:f
((mem ebp -4) <- edx)
(call :g)
(edx <- (mem ebp -4))
((mem ebp -8) <- eax)
(call :h)
(s0 <- (mem ebp -8))
(eax += s0)
(return)
```

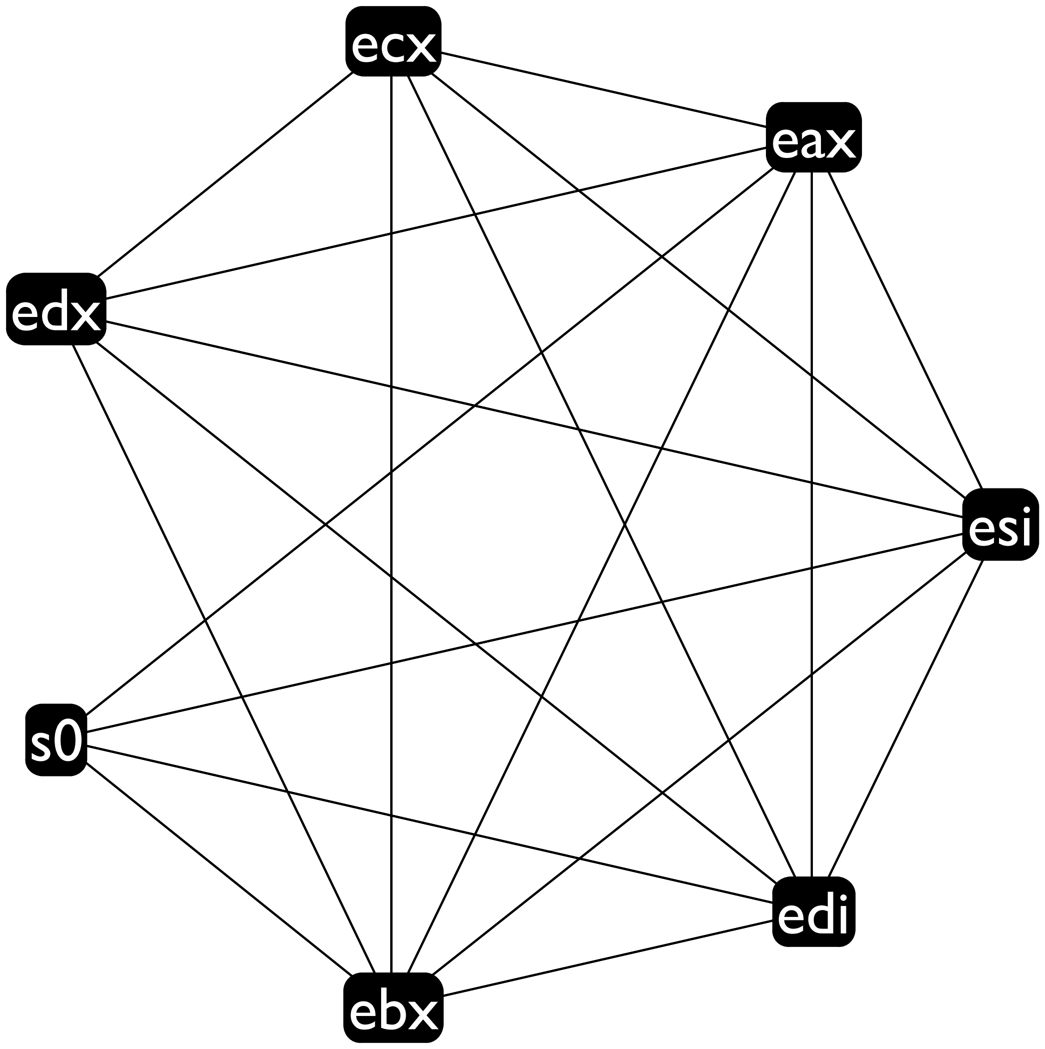
Note that this time we introduce a `s0`, but compare its live range to `g-ans`’s

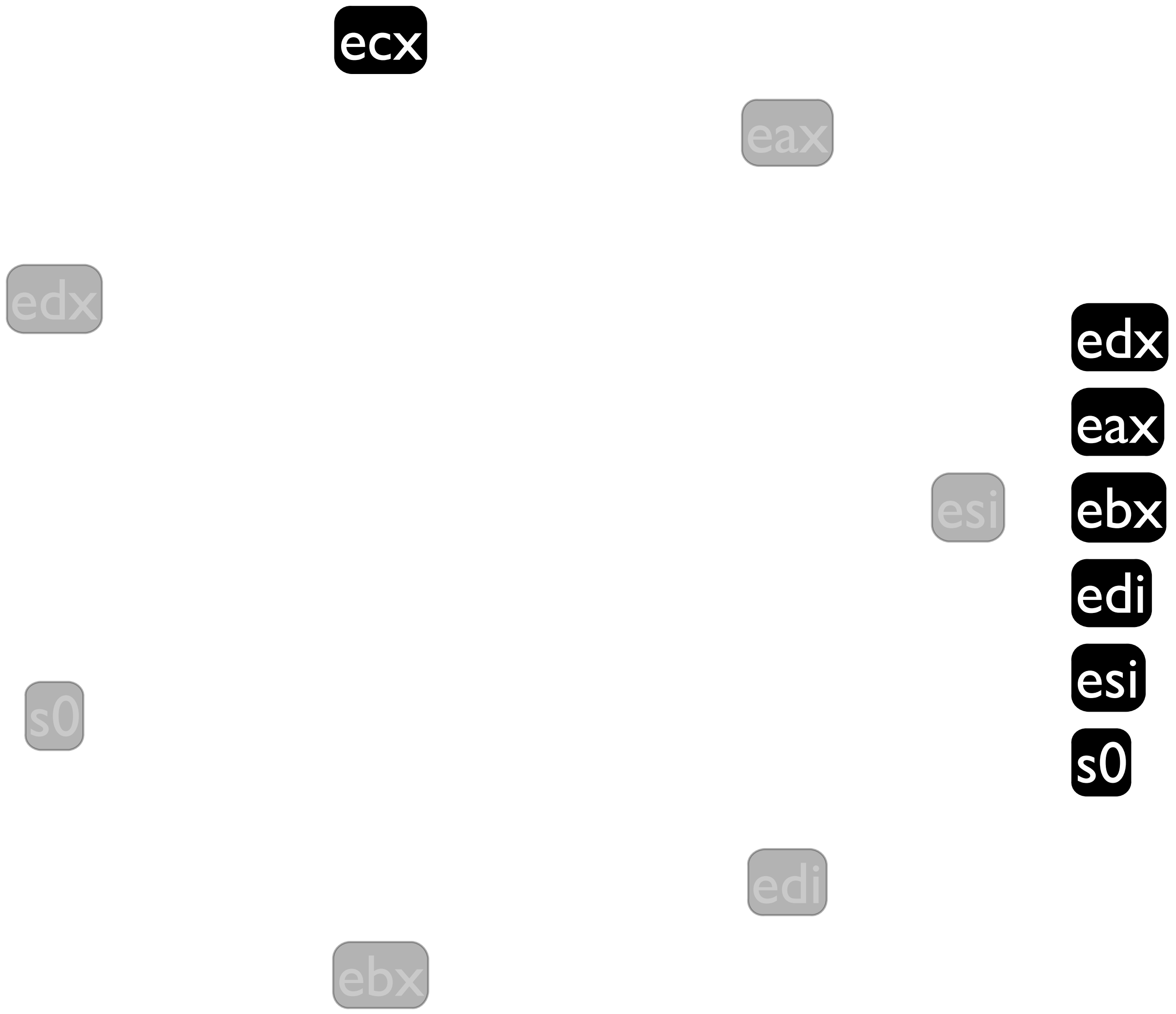
Spilling g-ans

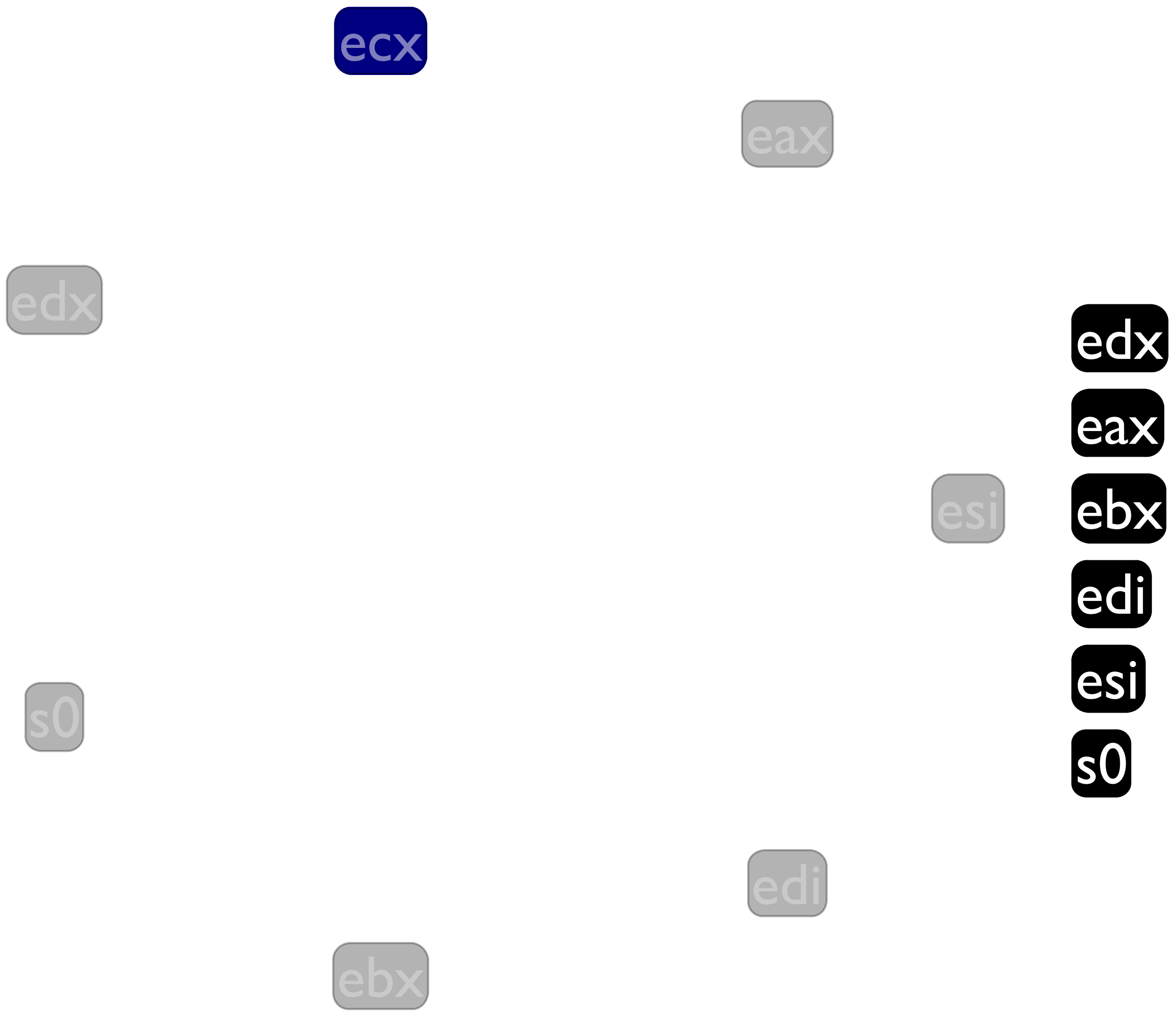
		in	out
1:	`:f`	()	()
2:	`((mem ebp -4) <- edx)`	()	()
3:	`(call :g)`	()	()
4:	`(edx <- (mem ebp -4))`	()	()
5:	`((mem ebp -8) <- eax)`	()	()
6:	`(call :h)`	()	()
7:	`(s0 <- (mem ebp -8))`	()	()
8:	`(eax += s0)`	()	()
9:	`(return)`	()	()

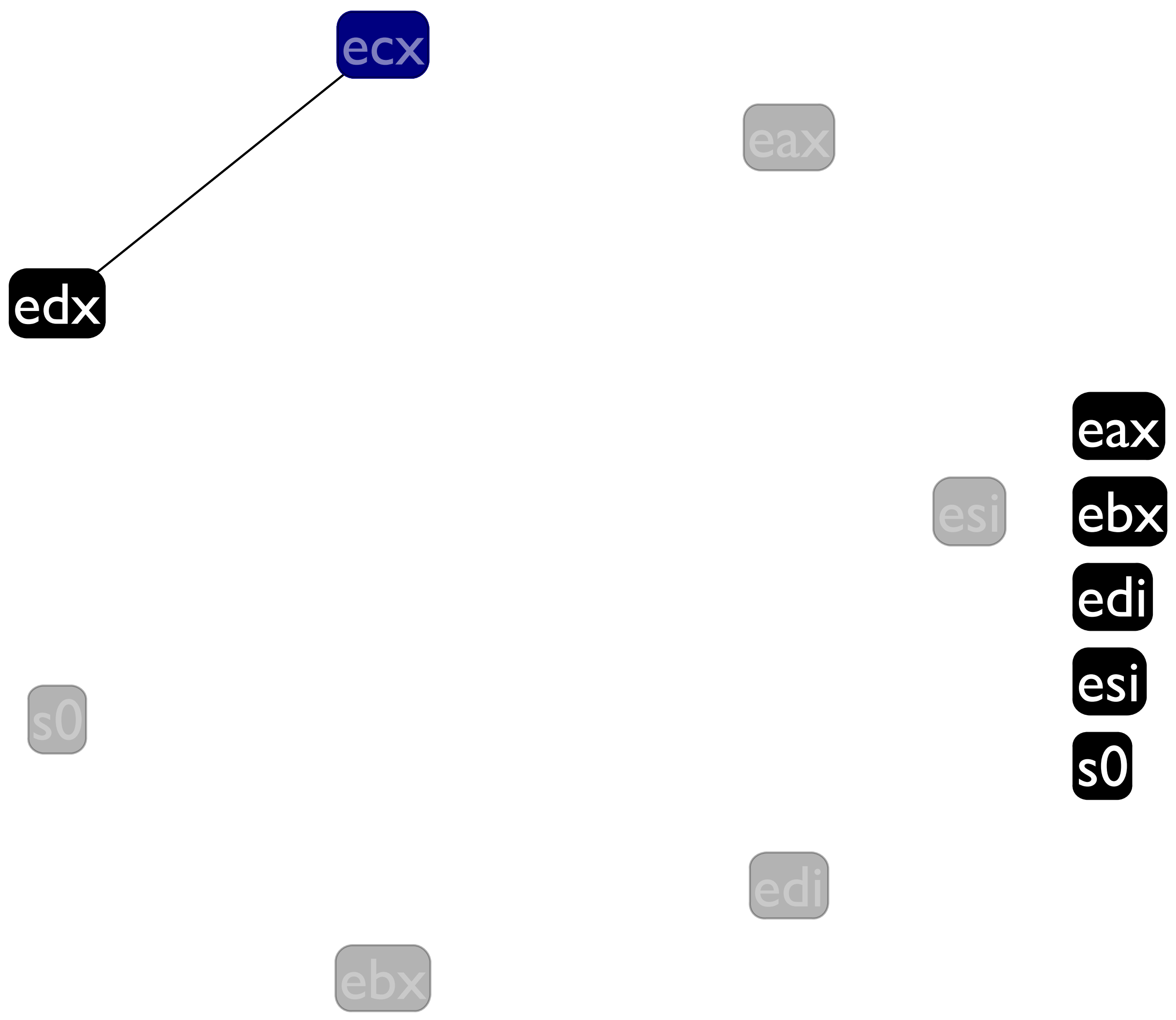
Spilling g-ans

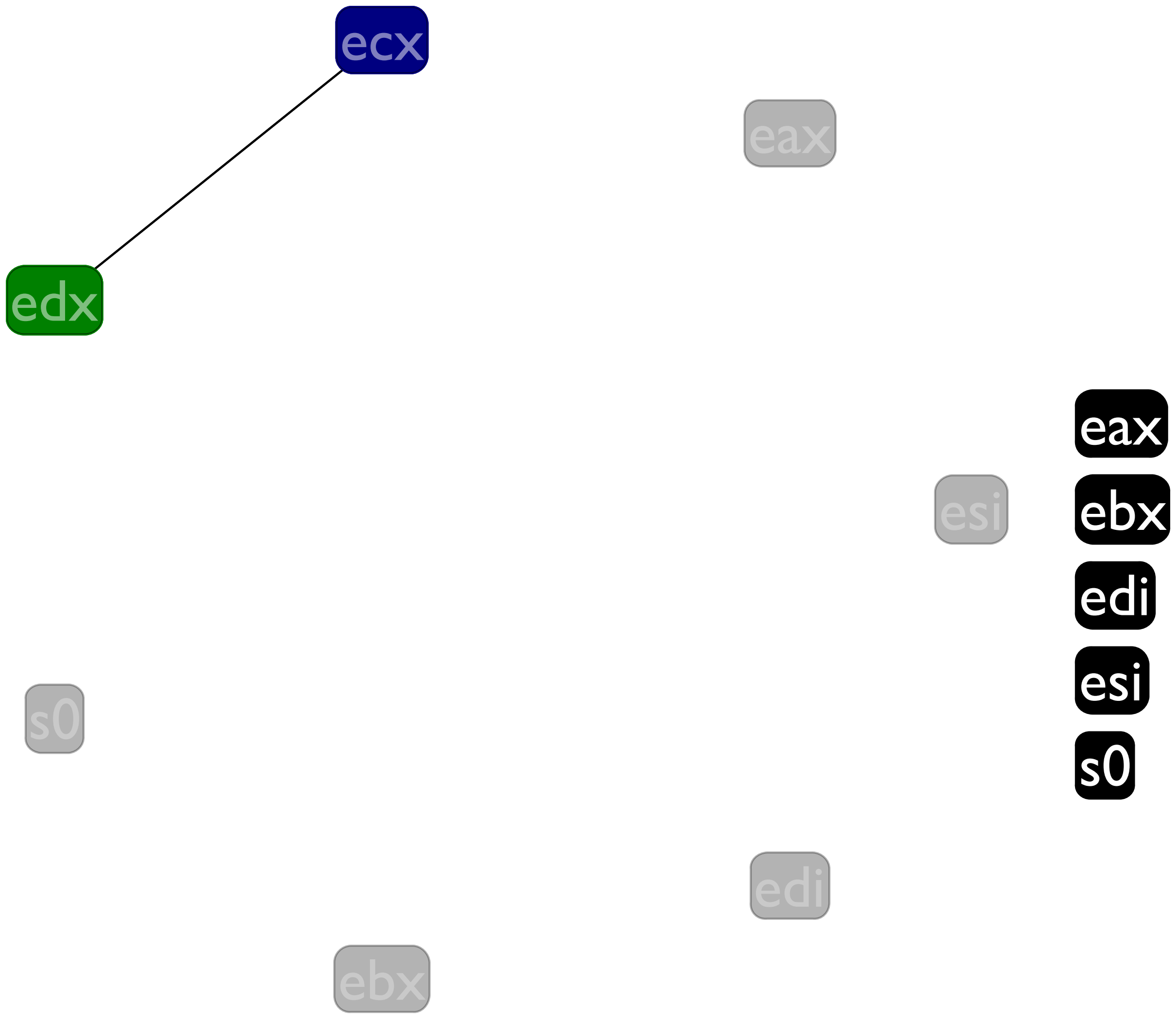
		in	out
1:	`:f`	(ebp ebx edi edx esi)	(ebp ebx edi edx esi)
2:	`((mem ebp -4) <- edx)`	(ebp ebx edi edx esi)	(ebp ebx edi esi)
3:	`(call :g)`	(ebp ebx edi esi)	(eax ebp ebx edi esi)
4:	`(edx <- (mem ebp -4))`	(eax ebp ebx edi esi)	(eax ebp ebx edi esi)
5:	`((mem ebp -8) <- eax)`	(eax ebp ebx edi esi)	(ebp ebx edi esi)
6:	`(call :h)`	(ebp ebx edi esi)	(eax ebp ebx edi esi)
7:	`(s0 <- (mem ebp -8))`	(eax ebp ebx edi esi)	(eax ebx edi esi s0)
8:	`(eax += s0)`	(eax ebx edi esi s0)	(ebx edi esi)
9:	`(return)`	(ebx edi esi)	()

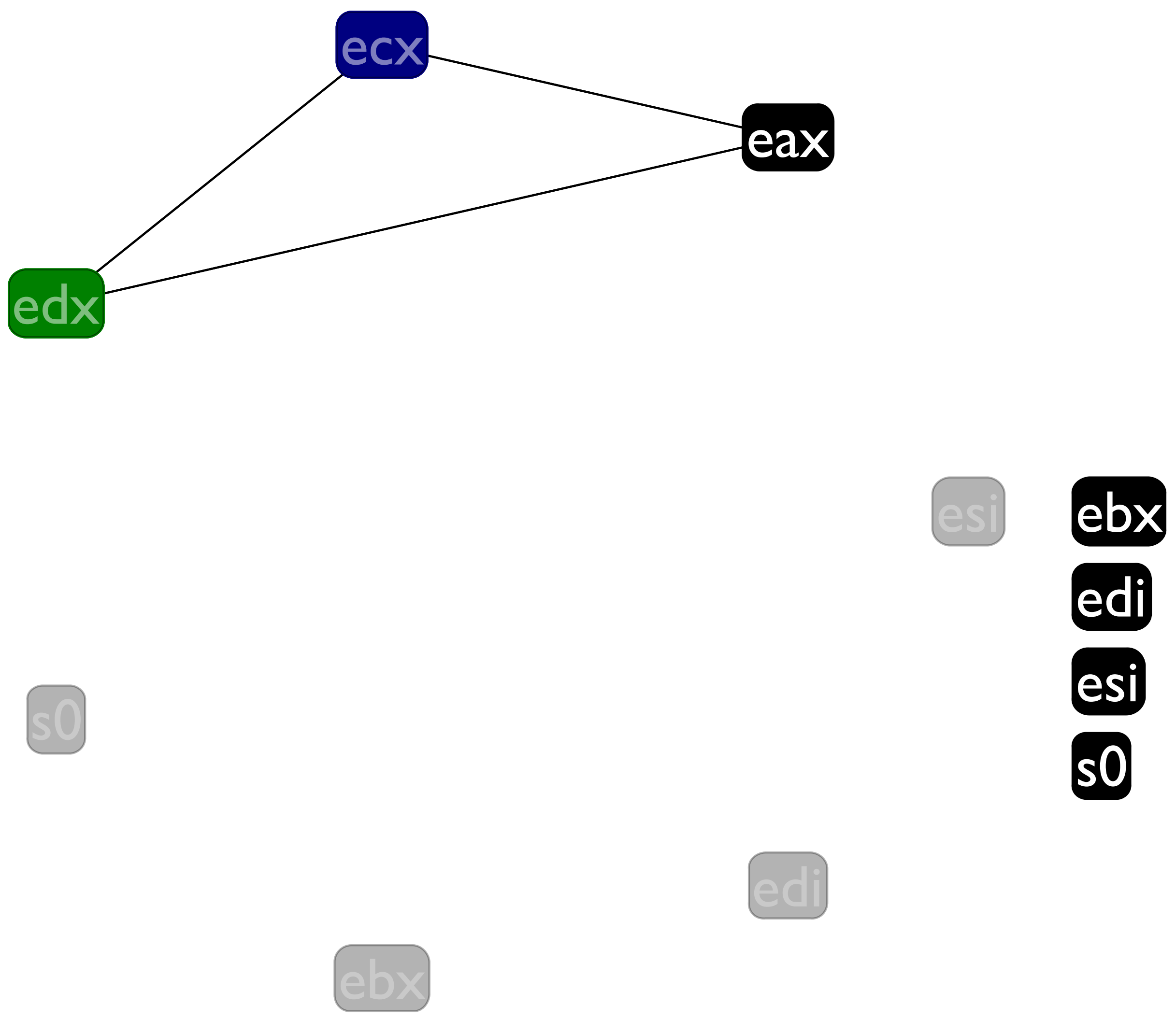

ecx
eax
edx
esi
s0
edi
ebx

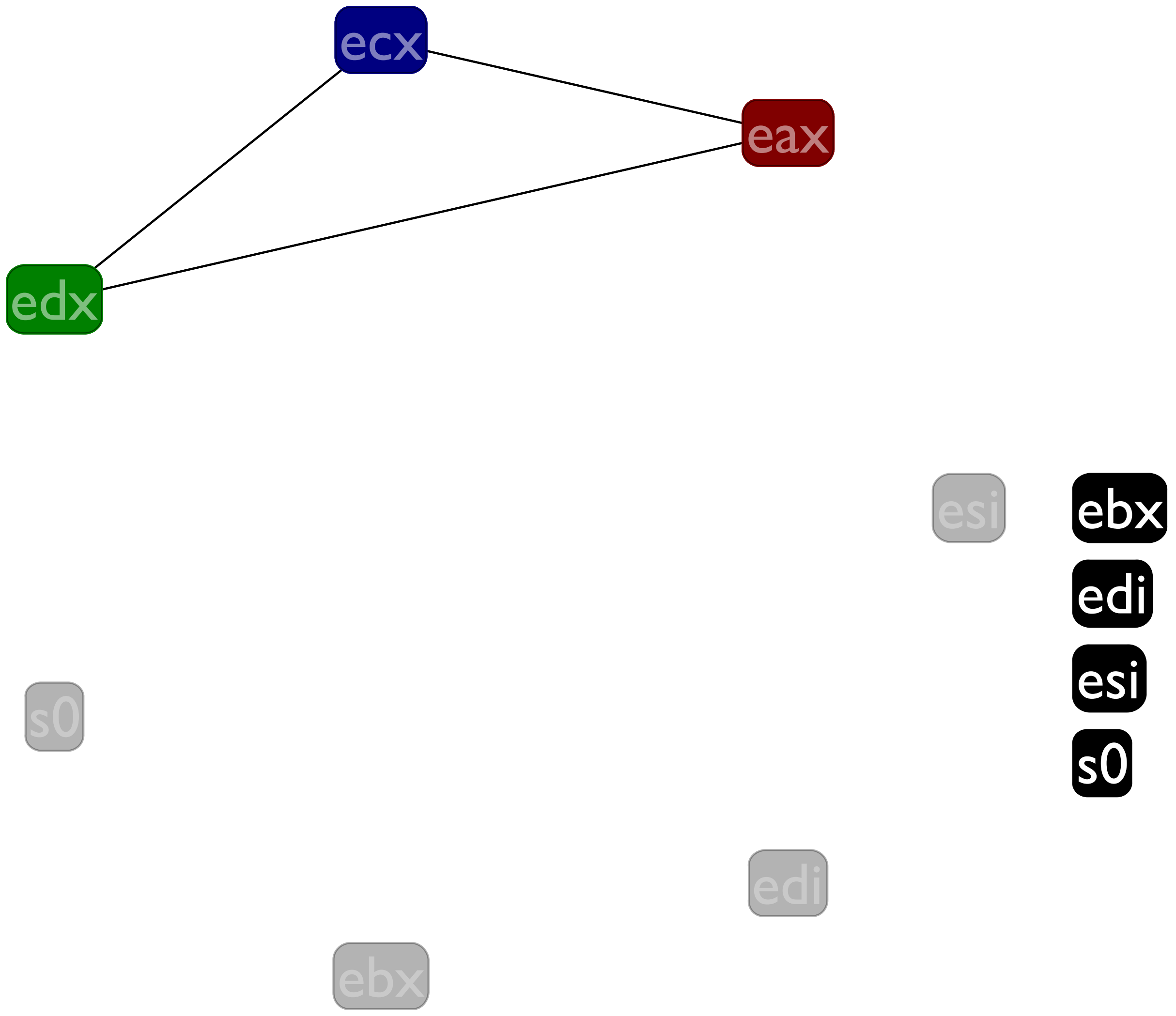

ecx
eax
edx
esi
s0
edi
ebx
edx
eax
ebx
edi
esi
s0

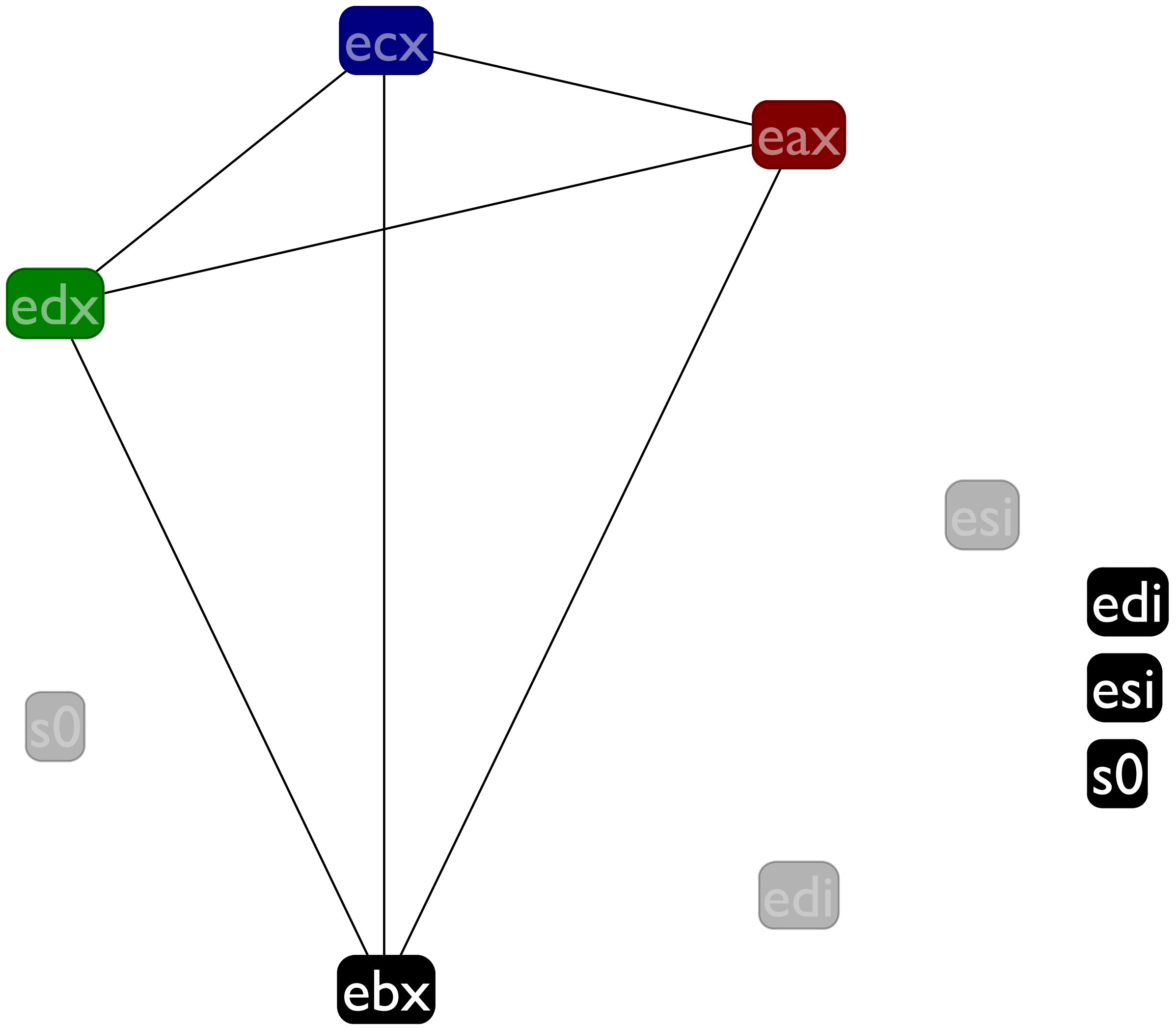

ecx
eax
edx
esi
s0
edi
ebx
edx
eax
ebx
edi
esi
s0

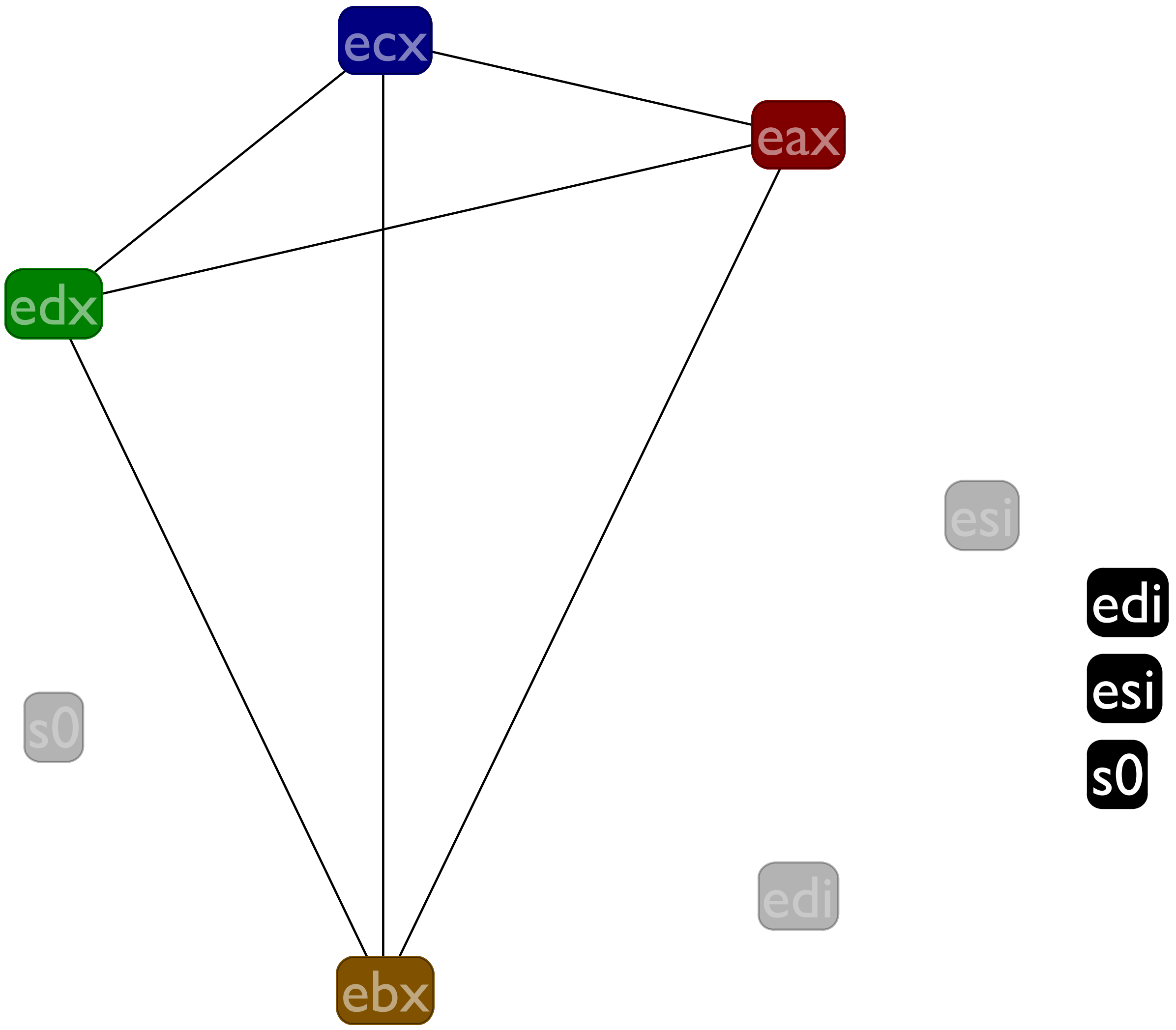

ecx
eax
edx
esi
s0
edi
ebx
eax
ebx
edi
esi
s0

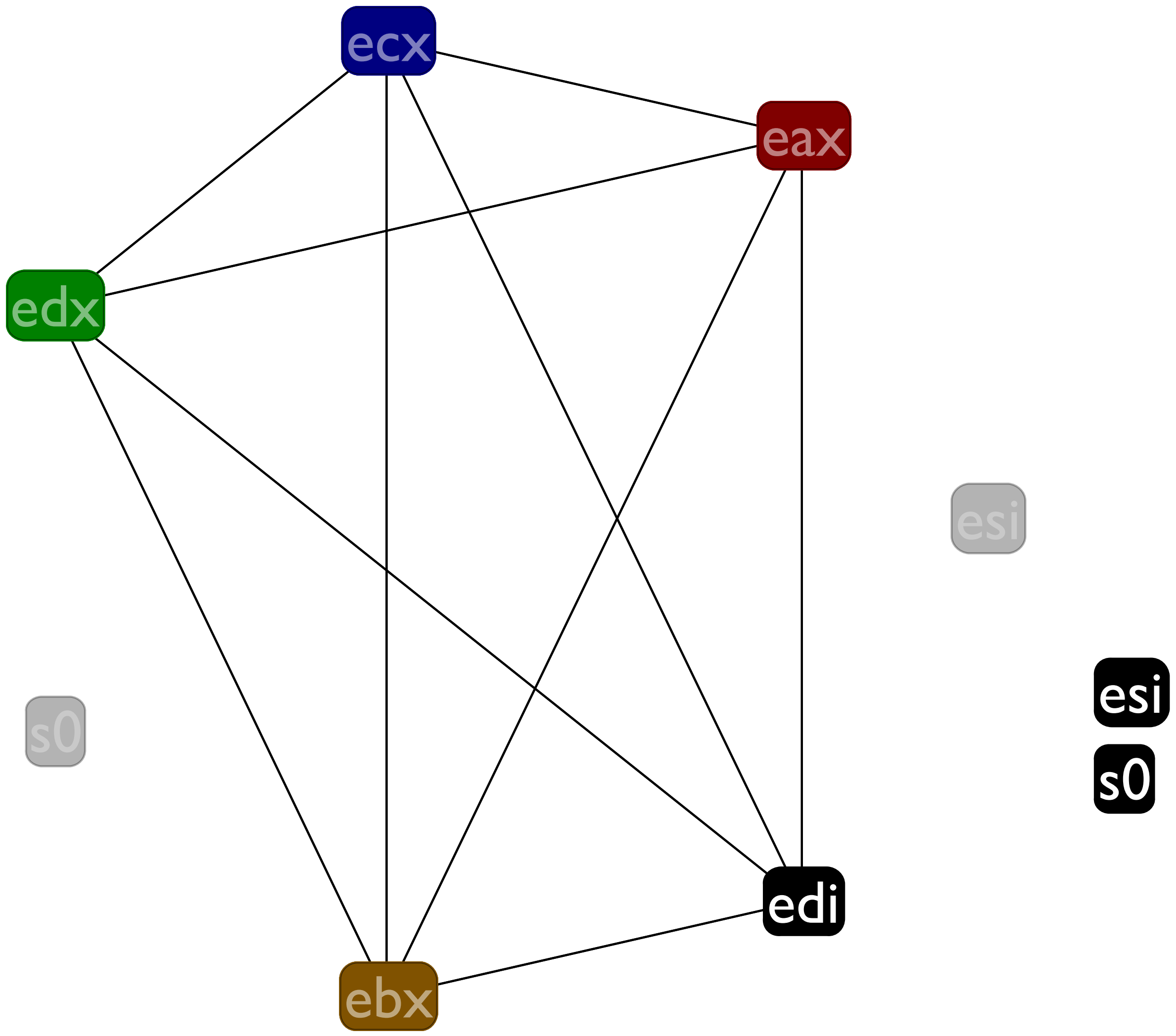

ecx
eax
edx
esi
s0
edi
ebx
eax
ebx
edi
esi
s0

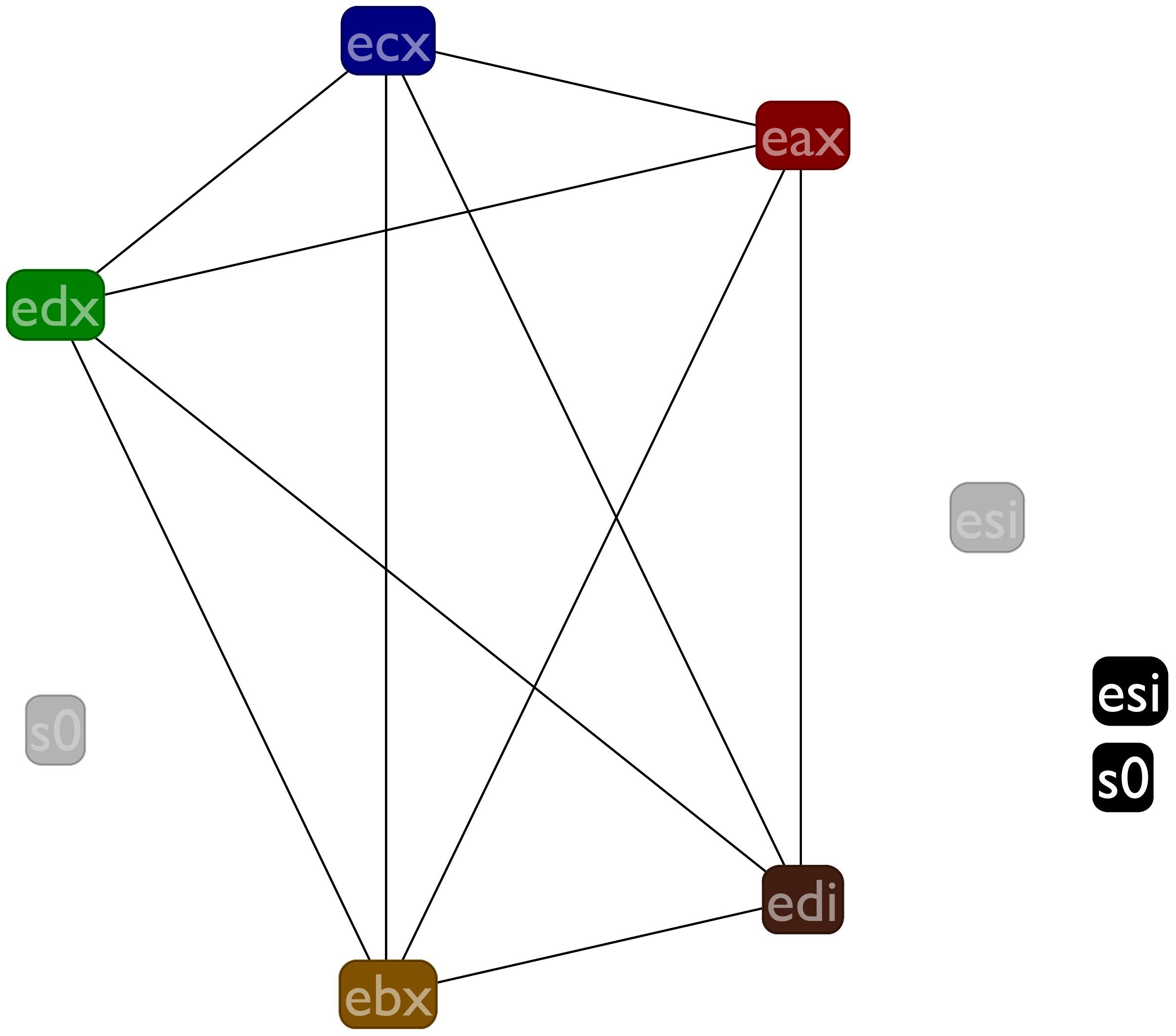

ecx
eax
edx
esi
ebx
edi
esi
s0
s0
edi
ebx

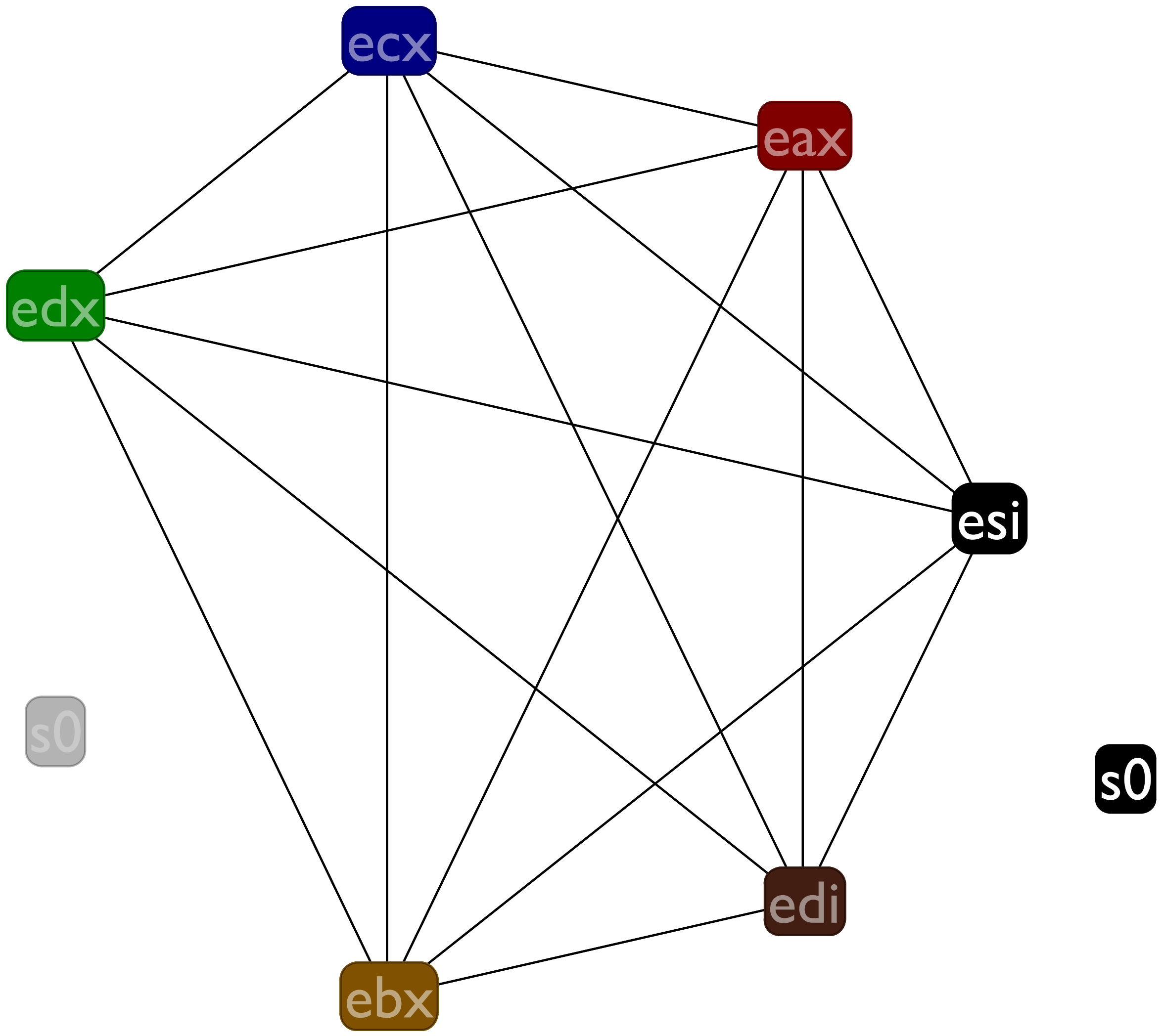

ecx
eax
edx
esi
ebx
edi
esi
s0
s0
edi
ebx

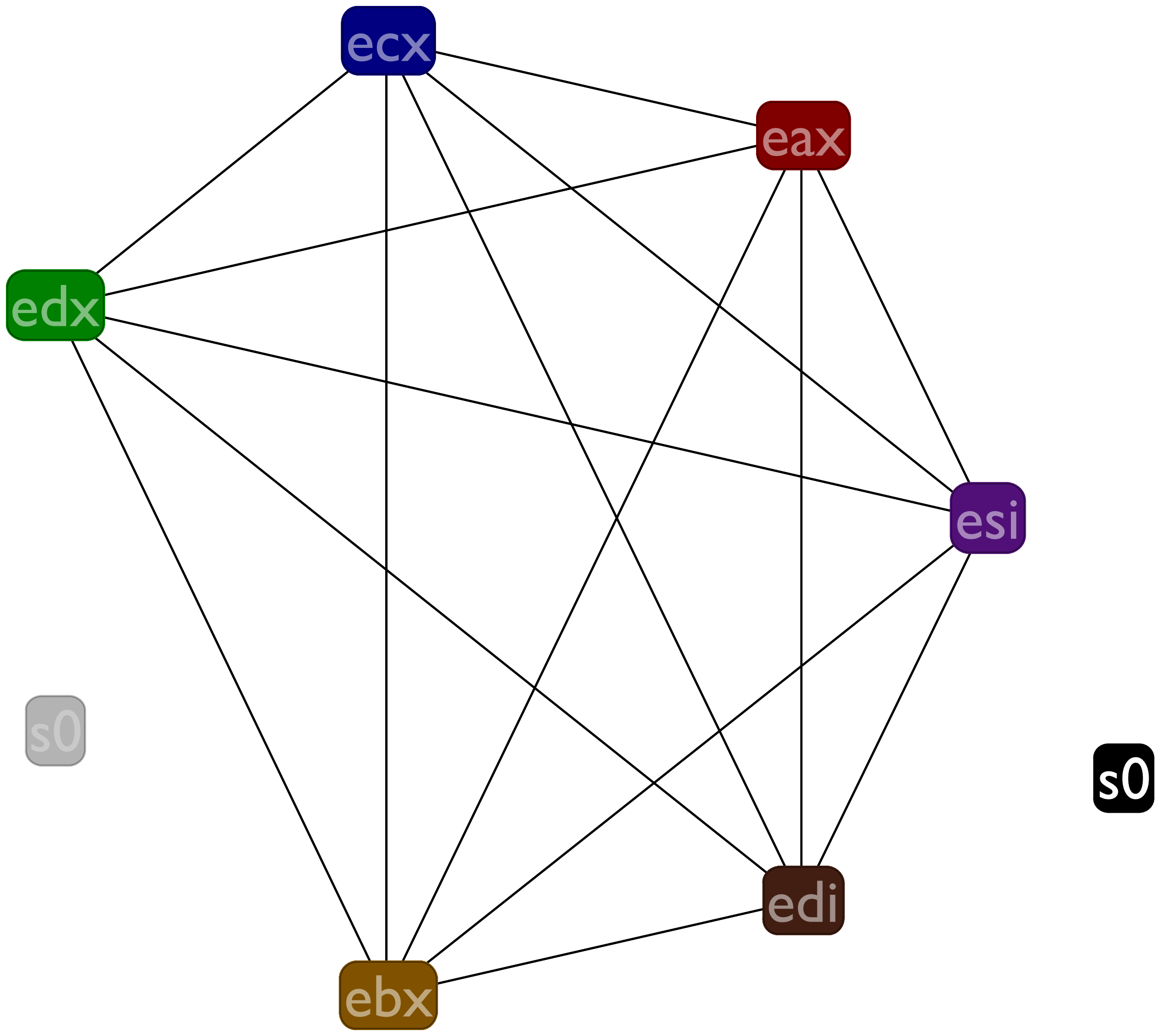

ecx
eax
edx
esi
edi
esi
s0
s0
edi
ebx

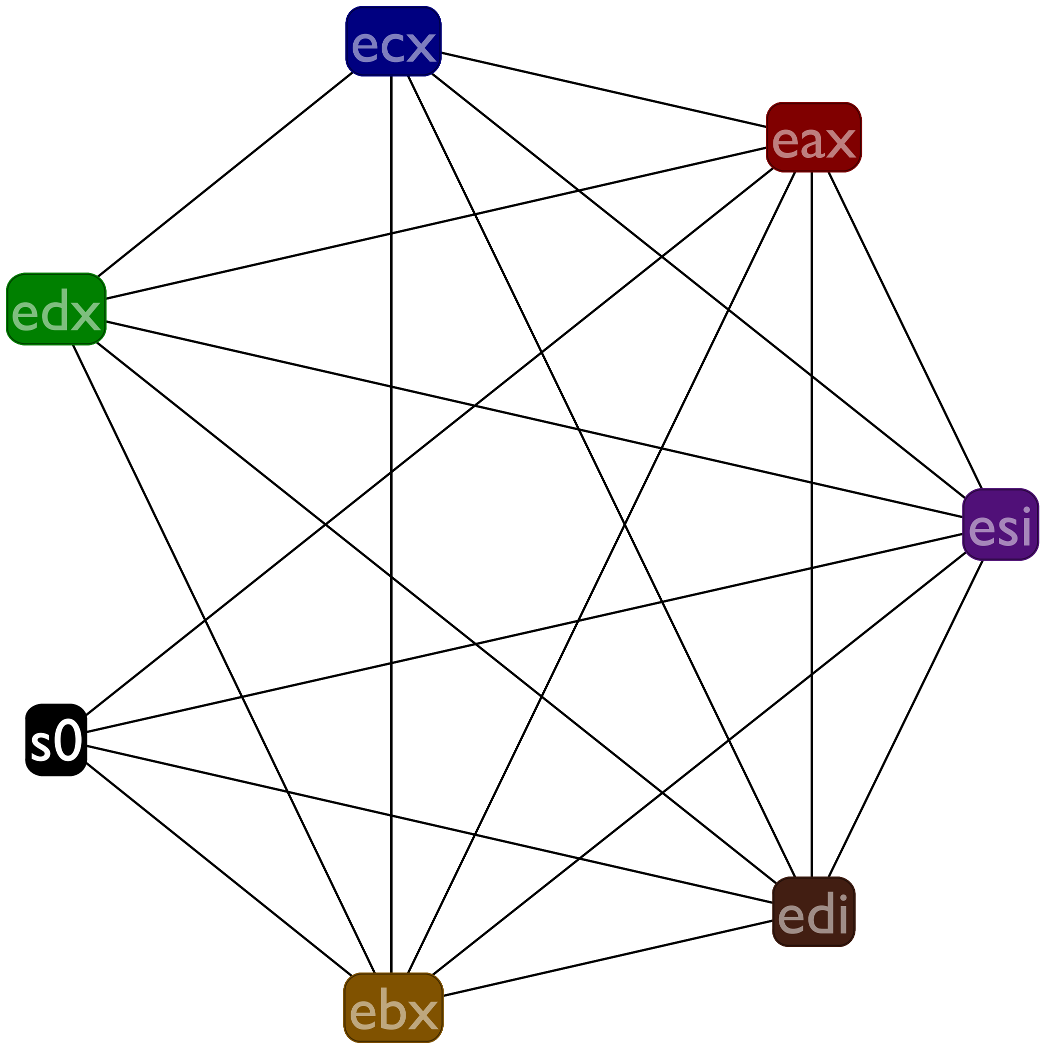

ecx
eax
edx
esi
edi
esi
s0
s0
edi
ebx

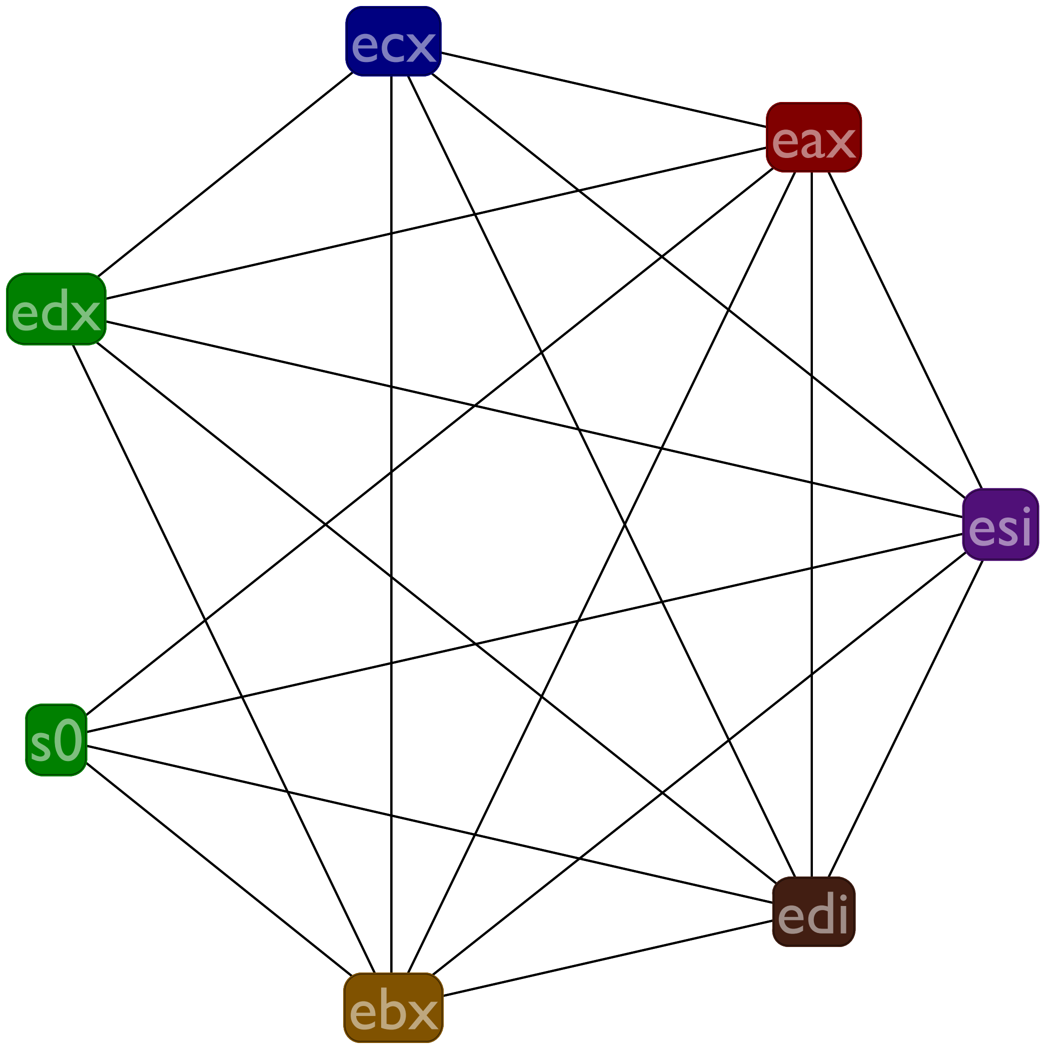

ecx
eax
edx
esi
esi
s0
s0
edi
ebx


ecx
eax
edx
esi
s0
esi
s0
edi
ebx


ecx
eax
edx
esi
s0
s0
edi
ebx

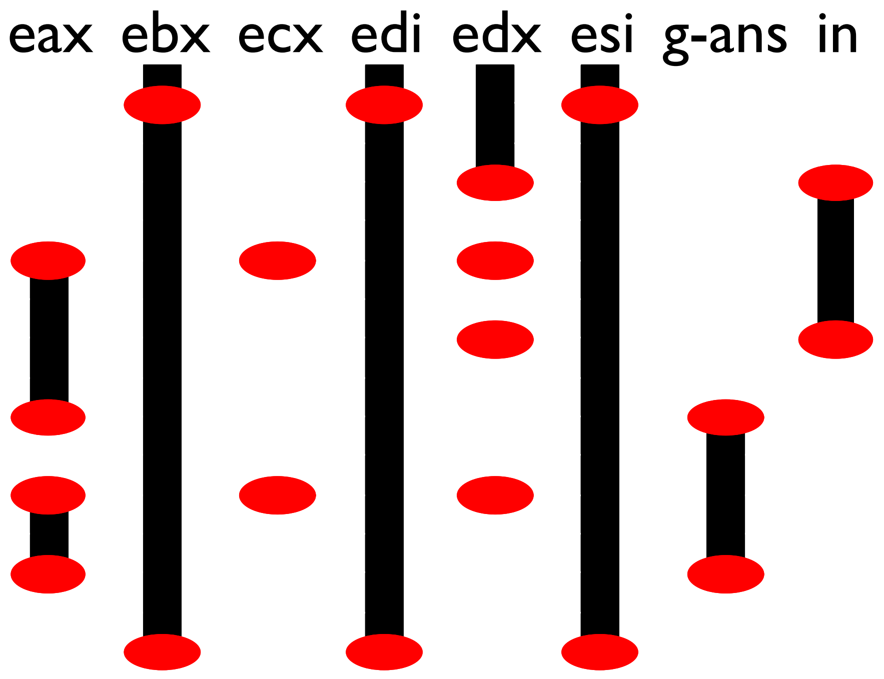

ecx
eax
edx
esi
s0
s0
edi
ebx


ecx
eax
edx
esi
s0
edi
ebx


ecx
eax
edx
esi
s0
edi
ebx

Live ranges

```
:f
(in <- edx)
(call :g)
(edx <- in)
(g-ans <- eax)
(call :h)
(eax += g-ans)
(return)
```



Live ranges

We spilled two variables that have relatively short live ranges, but look at those long live ranges with no uses of the variables that the callee save registers have. We'd rather spill them

Spilling callee saves

Unfortunately, it makes no sense to spill real registers. Instead, a trick: we just make up new variables to hold their values. Semantics of the program does not change, but:

- Now the real registers now have short live ranges, and
- New temporaries are spillable

Adding new variables g-ans

Before:

```
:f
(in <- edx)
(call :g)
(edx <- in)
(g-ans <- eax)
(call :h)
(eax += g-ans)
(return)
```

After:

```
:f
(z1 <- ebx)
(z2 <- edi)
(z3 <- esi)
(in <- edx)
(call :g)
(edx <- in)
(g-ans <- eax)
(call :h)
(eax += g-ans)
(ebx <- z1)
(edi <- z2)
(esi <- z3)
(return)
```

Init new variables at beginning of fun, restore them before returning or jumping to computed location (this is our tail calls; more later)

With new variables

		in	out
1:	`:f`	()	()
2:	`(z1 <- ebx)`	()	()
3:	`(z2 <- edi)`	()	()
4:	`(z3 <- esi)`	()	()
5:	`(in <- edx)`	()	()
6:	`(call :g)`	()	()
7:	`(edx <- in)`	()	()
8:	`(g-ans <- eax)`	()	()
9:	`(call :h)`	()	()
10:	`(eax += g-ans)`	()	()
11:	`(ebx <- z1)`	()	()
12:	`(edi <- z2)`	()	()
13:	`(esi <- z3)`	()	()
14:	`(return)`	()	()

With new variables

		in	out
1:	`:f`	()	()
2:	`(z1 <- ebx)`	(ebx)	()
3:	`(z2 <- edi)`	(edi)	()
4:	`(z3 <- esi)`	(esi)	()
5:	`(in <- edx)`	(edx)	()
6:	`(call :g)`	()	()
7:	`(edx <- in)`	(in)	()
8:	`(g-ans <- eax)`	(eax)	()
9:	`(call :h)`	()	()
10:	`(eax += g-ans)`	(eax g-ans)	()
11:	`(ebx <- z1)`	(z1)	()
12:	`(edi <- z2)`	(z2)	()
13:	`(esi <- z3)`	(z3)	()
14:	`(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	()	(ebx)
2:	`(z1 <- ebx)`	(ebx)	(edi)
3:	`(z2 <- edi)`	(edi)	(esi)
4:	`(z3 <- esi)`	(esi)	(edx)
5:	`(in <- edx)`	(edx)	()
6:	`(call :g)`	()	(in)
7:	`(edx <- in)`	(in)	(eax)
8:	`(g-ans <- eax)`	(eax)	()
9:	`(call :h)`	()	(eax g-ans)
10:	`(eax += g-ans)`	(eax g-ans)	(z1)
11:	`(ebx <- z1)`	(z1)	(z2)
12:	`(edi <- z2)`	(z2)	(z3)
13:	`(esi <- z3)`	(z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx)	(ebx)
2:	`(z1 <- ebx)`	(ebx edi)	(edi)
3:	`(z2 <- edi)`	(edi esi)	(esi)
4:	`(z3 <- esi)`	(edx esi)	(edx)
5:	`(in <- edx)`	(edx)	()
6:	`(call :g)`	(in)	(in)
7:	`(edx <- in)`	(eax in)	(eax)
8:	`(g-ans <- eax)`	(eax)	()
9:	`(call :h)`	(g-ans)	(eax g-ans)
10:	`(eax += g-ans)`	(eax g-ans z1)	(z1)
11:	`(ebx <- z1)`	(z1 z2)	(z2)
12:	`(edi <- z2)`	(z2 z3)	(z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx)	(ebx edi)
2:	`(z1 <- ebx)`	(ebx edi)	(edi esi)
3:	`(z2 <- edi)`	(edi esi)	(edx esi)
4:	`(z3 <- esi)`	(edx esi)	(edx)
5:	`(in <- edx)`	(edx)	(in)
6:	`(call :g)`	(in)	(eax in)
7:	`(edx <- in)`	(eax in)	(eax)
8:	`(g-ans <- eax)`	(eax)	(g-ans)
9:	`(call :h)`	(g-ans)	(eax g-ans z1)
10:	`(eax += g-ans)`	(eax g-ans z1)	(z1 z2)
11:	`(ebx <- z1)`	(z1 z2)	(z2 z3)
12:	`(edi <- z2)`	(z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx edi)	(ebx edi)
2:	`(z1 <- ebx)`	(ebx edi esi)	(edi esi)
3:	`(z2 <- edi)`	(edi edx esi)	(edx esi)
4:	`(z3 <- esi)`	(edx esi)	(edx)
5:	`(in <- edx)`	(edx)	(in)
6:	`(call :g)`	(in)	(eax in)
7:	`(edx <- in)`	(eax in)	(eax)
8:	`(g-ans <- eax)`	(eax)	(g-ans)
9:	`(call :h)`	(g-ans z1)	(eax g-ans z1)
10:	`(eax += g-ans)`	(eax g-ans z1 z2)	(z1 z2)
11:	`(ebx <- z1)`	(z1 z2 z3)	(z2 z3)
12:	`(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx edi)	(ebx edi esi)
2:	`(z1 <- ebx)`	(ebx edi esi)	(edi edx esi)
3:	`(z2 <- edi)`	(edi edx esi)	(edx esi)
4:	`(z3 <- esi)`	(edx esi)	(edx)
5:	`(in <- edx)`	(edx)	(in)
6:	`(call :g)`	(in)	(eax in)
7:	`(edx <- in)`	(eax in)	(eax)
8:	`(g-ans <- eax)`	(eax)	(g-ans z1)
9:	`(call :h)`	(g-ans z1)	(eax g-ans z1 z2)
10:	`(eax += g-ans)`	(eax g-ans z1 z2)	(z1 z2 z3)
11:	`(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12:	`(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx edi esi)	(ebx edi esi)
2:	`(z1 <- ebx)`	(ebx edi edx esi)	(edi edx esi)
3:	`(z2 <- edi)`	(edi edx esi)	(edx esi)
4:	`(z3 <- esi)`	(edx esi)	(edx)
5:	`(in <- edx)`	(edx)	(in)
6:	`(call :g)`	(in)	(eax in)
7:	`(edx <- in)`	(eax in)	(eax)
8:	`(g-ans <- eax)`	(eax z1)	(g-ans z1)
9:	`(call :h)`	(g-ans z1 z2)	(eax g-ans z1 z2)
10:	`(eax += g-ans)`	(eax g-ans z1 z2 z3)	(z1 z2 z3)
11:	`(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12:	`(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx edi esi)	(ebx edi edx esi)
2:	`(z1 <- ebx)`	(ebx edi edx esi)	(edi edx esi)
3:	`(z2 <- edi)`	(edi edx esi)	(edx esi)
4:	`(z3 <- esi)`	(edx esi)	(edx)
5:	`(in <- edx)`	(edx)	(in)
6:	`(call :g)`	(in)	(eax in)
7:	`(edx <- in)`	(eax in)	(eax z1)
8:	`(g-ans <- eax)`	(eax z1)	(g-ans z1 z2)
9:	`(call :h)`	(g-ans z1 z2)	(eax g-ans z1 z2 z3)
10:	`(eax += g-ans)`	(eax g-ans z1 z2 z3)	(z1 z2 z3)
11:	`(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12:	`(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

	in	out
1: `:f`	(ebx edi edx esi)	(ebx edi edx esi)
2: `(z1 <- ebx)`	(ebx edi edx esi)	(edi edx esi)
3: `(z2 <- edi)`	(edi edx esi)	(edx esi)
4: `(z3 <- esi)`	(edx esi)	(edx)
5: `(in <- edx)`	(edx)	(in)
6: `(call :g)`	(in)	(eax in)
7: `(edx <- in)`	(eax in z1)	(eax z1)
8: `(g-ans <- eax)`	(eax z1 z2)	(g-ans z1 z2)
9: `(call :h)`	(g-ans z1 z2 z3)	(eax g-ans z1 z2 z3)
10: `(eax += g-ans)`	(eax g-ans z1 z2 z3)	(z1 z2 z3)
11: `(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12: `(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13: `(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14: `(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx edi edx esi)	(ebx edi edx esi)
2:	`(z1 <- ebx)`	(ebx edi edx esi)	(edi edx esi)
3:	`(z2 <- edi)`	(edi edx esi)	(edx esi)
4:	`(z3 <- esi)`	(edx esi)	(edx)
5:	`(in <- edx)`	(edx)	(in)
6:	`(call :g)`	(in)	(eax in z1)
7:	`(edx <- in)`	(eax in z1)	(eax z1 z2)
8:	`(g-ans <- eax)`	(eax z1 z2)	(g-ans z1 z2 z3)
9:	`(call :h)`	(g-ans z1 z2 z3)	(eax g-ans z1 z2 z3)
10:	`(eax += g-ans)`	(eax g-ans z1 z2 z3)	(z1 z2 z3)
11:	`(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12:	`(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx edi edx esi)	(ebx edi edx esi)
2:	`(z1 <- ebx)`	(ebx edi edx esi)	(edi edx esi)
3:	`(z2 <- edi)`	(edi edx esi)	(edx esi)
4:	`(z3 <- esi)`	(edx esi)	(edx)
5:	`(in <- edx)`	(edx)	(in)
6:	`(call :g)`	(in z1)	(eax in z1)
7:	`(edx <- in)`	(eax in z1 z2)	(eax z1 z2)
8:	`(g-ans <- eax)`	(eax z1 z2 z3)	(g-ans z1 z2 z3)
9:	`(call :h)`	(g-ans z1 z2 z3)	(eax g-ans z1 z2 z3)
10:	`(eax += g-ans)`	(eax g-ans z1 z2 z3)	(z1 z2 z3)
11:	`(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12:	`(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx edi edx esi)	(ebx edi edx esi)
2:	`(z1 <- ebx)`	(ebx edi edx esi)	(edi edx esi)
3:	`(z2 <- edi)`	(edi edx esi)	(edx esi)
4:	`(z3 <- esi)`	(edx esi)	(edx)
5:	`(in <- edx)`	(edx)	(in z1)
6:	`(call :g)`	(in z1)	(eax in z1 z2)
7:	`(edx <- in)`	(eax in z1 z2)	(eax z1 z2 z3)
8:	`(g-ans <- eax)`	(eax z1 z2 z3)	(g-ans z1 z2 z3)
9:	`(call :h)`	(g-ans z1 z2 z3)	(eax g-ans z1 z2 z3)
10:	`(eax += g-ans)`	(eax g-ans z1 z2 z3)	(z1 z2 z3)
11:	`(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12:	`(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx edi edx esi)	(ebx edi edx esi)
2:	`(z1 <- ebx)`	(ebx edi edx esi)	(edi edx esi)
3:	`(z2 <- edi)`	(edi edx esi)	(edx esi)
4:	`(z3 <- esi)`	(edx esi)	(edx)
5:	`(in <- edx)`	(edx z1)	(in z1)
6:	`(call :g)`	(in z1 z2)	(eax in z1 z2)
7:	`(edx <- in)`	(eax in z1 z2 z3)	(eax z1 z2 z3)
8:	`(g-ans <- eax)`	(eax z1 z2 z3)	(g-ans z1 z2 z3)
9:	`(call :h)`	(g-ans z1 z2 z3)	(eax g-ans z1 z2 z3)
10:	`(eax += g-ans)`	(eax g-ans z1 z2 z3)	(z1 z2 z3)
11:	`(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12:	`(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

	in	out
1: `:f`	(ebx edi edx esi)	(ebx edi edx esi)
2: `(z1 <- ebx)`	(ebx edi edx esi)	(edi edx esi)
3: `(z2 <- edi)`	(edi edx esi)	(edx esi)
4: `(z3 <- esi)`	(edx esi)	(edx z1)
5: `(in <- edx)`	(edx z1)	(in z1 z2)
6: `(call :g)`	(in z1 z2)	(eax in z1 z2 z3)
7: `(edx <- in)`	(eax in z1 z2 z3)	(eax z1 z2 z3)
8: `(g-ans <- eax)`	(eax z1 z2 z3)	(g-ans z1 z2 z3)
9: `(call :h)`	(g-ans z1 z2 z3)	(eax g-ans z1 z2 z3)
10: `(eax += g-ans)`	(eax g-ans z1 z2 z3)	(z1 z2 z3)
11: `(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12: `(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13: `(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14: `(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx edi edx esi)	(ebx edi edx esi)
2:	`(z1 <- ebx)`	(ebx edi edx esi)	(edi edx esi)
3:	`(z2 <- edi)`	(edi edx esi)	(edx esi)
4:	`(z3 <- esi)`	(edx esi z1)	(edx z1)
5:	`(in <- edx)`	(edx z1 z2)	(in z1 z2)
6:	`(call :g)`	(in z1 z2 z3)	(eax in z1 z2 z3)
7:	`(edx <- in)`	(eax in z1 z2 z3)	(eax z1 z2 z3)
8:	`(g-ans <- eax)`	(eax z1 z2 z3)	(g-ans z1 z2 z3)
9:	`(call :h)`	(g-ans z1 z2 z3)	(eax g-ans z1 z2 z3)
10:	`(eax += g-ans)`	(eax g-ans z1 z2 z3)	(z1 z2 z3)
11:	`(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12:	`(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx edi edx esi)	(ebx edi edx esi)
2:	`(z1 <- ebx)`	(ebx edi edx esi)	(edi edx esi)
3:	`(z2 <- edi)`	(edi edx esi)	(edx esi z1)
4:	`(z3 <- esi)`	(edx esi z1)	(edx z1 z2)
5:	`(in <- edx)`	(edx z1 z2)	(in z1 z2 z3)
6:	`(call :g)`	(in z1 z2 z3)	(eax in z1 z2 z3)
7:	`(edx <- in)`	(eax in z1 z2 z3)	(eax z1 z2 z3)
8:	`(g-ans <- eax)`	(eax z1 z2 z3)	(g-ans z1 z2 z3)
9:	`(call :h)`	(g-ans z1 z2 z3)	(eax g-ans z1 z2 z3)
10:	`(eax += g-ans)`	(eax g-ans z1 z2 z3)	(z1 z2 z3)
11:	`(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12:	`(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

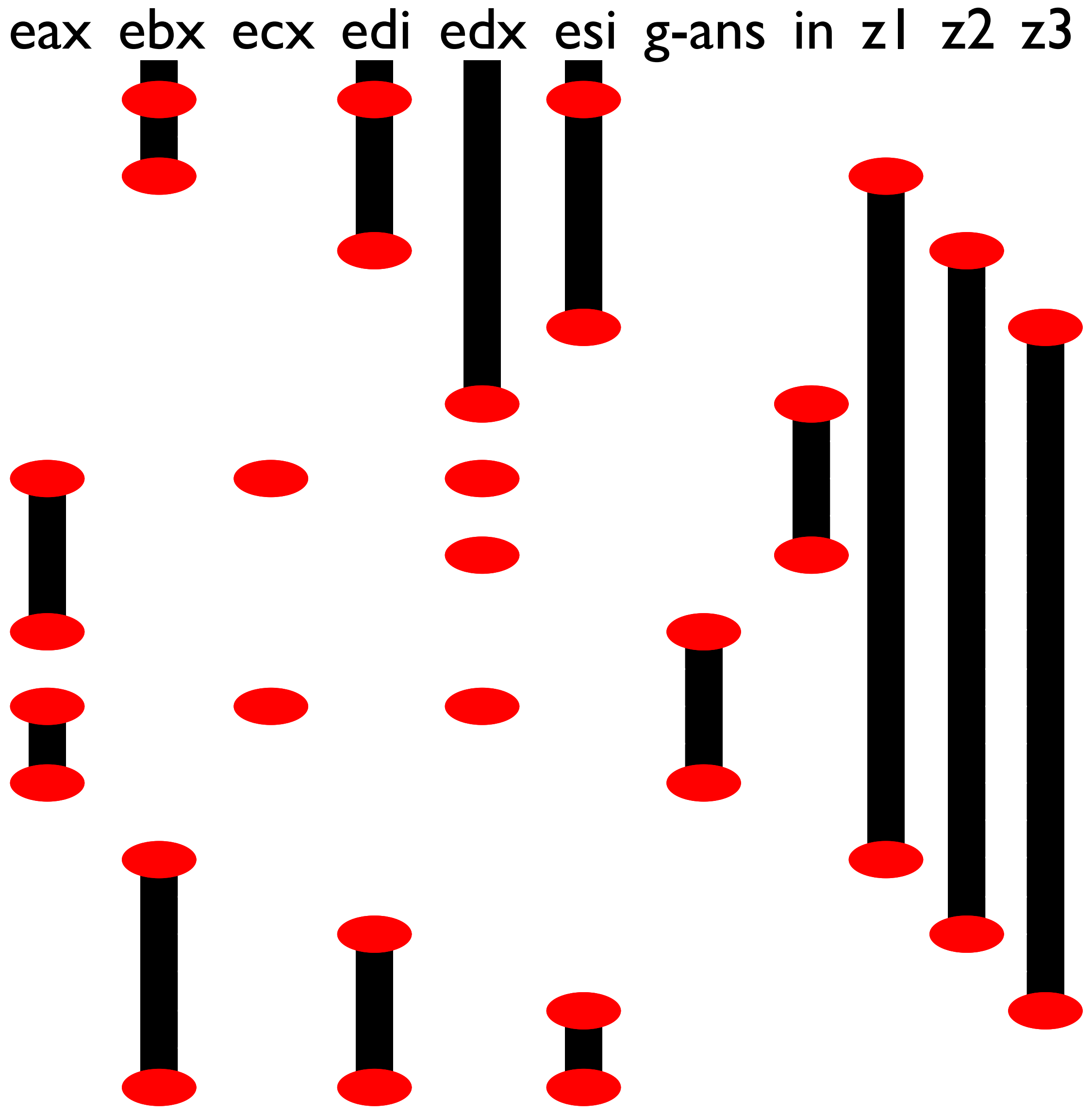
		in	out
1:	`:f`	(ebx edi edx esi)	(ebx edi edx esi)
2:	`(z1 <- ebx)`	(ebx edi edx esi)	(edi edx esi)
3:	`(z2 <- edi)`	(edi edx esi z1)	(edx esi z1)
4:	`(z3 <- esi)`	(edx esi z1 z2)	(edx z1 z2)
5:	`(in <- edx)`	(edx z1 z2 z3)	(in z1 z2 z3)
6:	`(call :g)`	(in z1 z2 z3)	(eax in z1 z2 z3)
7:	`(edx <- in)`	(eax in z1 z2 z3)	(eax z1 z2 z3)
8:	`(g-ans <- eax)`	(eax z1 z2 z3)	(g-ans z1 z2 z3)
9:	`(call :h)`	(g-ans z1 z2 z3)	(eax g-ans z1 z2 z3)
10:	`(eax += g-ans)`	(eax g-ans z1 z2 z3)	(z1 z2 z3)
11:	`(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12:	`(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

With new variables

		in	out
1:	`:f`	(ebx edi edx esi)	(ebx edi edx esi)
2:	`(z1 <- ebx)`	(ebx edi edx esi)	(edi edx esi z1)
3:	`(z2 <- edi)`	(edi edx esi z1)	(edx esi z1 z2)
4:	`(z3 <- esi)`	(edx esi z1 z2)	(edx z1 z2 z3)
5:	`(in <- edx)`	(edx z1 z2 z3)	(in z1 z2 z3)
6:	`(call :g)`	(in z1 z2 z3)	(eax in z1 z2 z3)
7:	`(edx <- in)`	(eax in z1 z2 z3)	(eax z1 z2 z3)
8:	`(g-ans <- eax)`	(eax z1 z2 z3)	(g-ans z1 z2 z3)
9:	`(call :h)`	(g-ans z1 z2 z3)	(eax g-ans z1 z2 z3)
10:	`(eax += g-ans)`	(eax g-ans z1 z2 z3)	(z1 z2 z3)
11:	`(ebx <- z1)`	(z1 z2 z3)	(ebx z2 z3)
12:	`(edi <- z2)`	(ebx z2 z3)	(ebx edi z3)
13:	`(esi <- z3)`	(ebx edi z3)	(ebx edi esi)
14:	`(return)`	(ebx edi esi)	()

Live ranges

```
:f
(z1 <- ebx)
(z2 <- edi)
(z3 <- esi)
(in <- edx)
(call :g)
(edx <- in)
(g-ans <- eax)
(call :h)
(eax += g-ans)
(ebx <- z1)
(edi <- z2)
(esi <- z3)
(return)
```



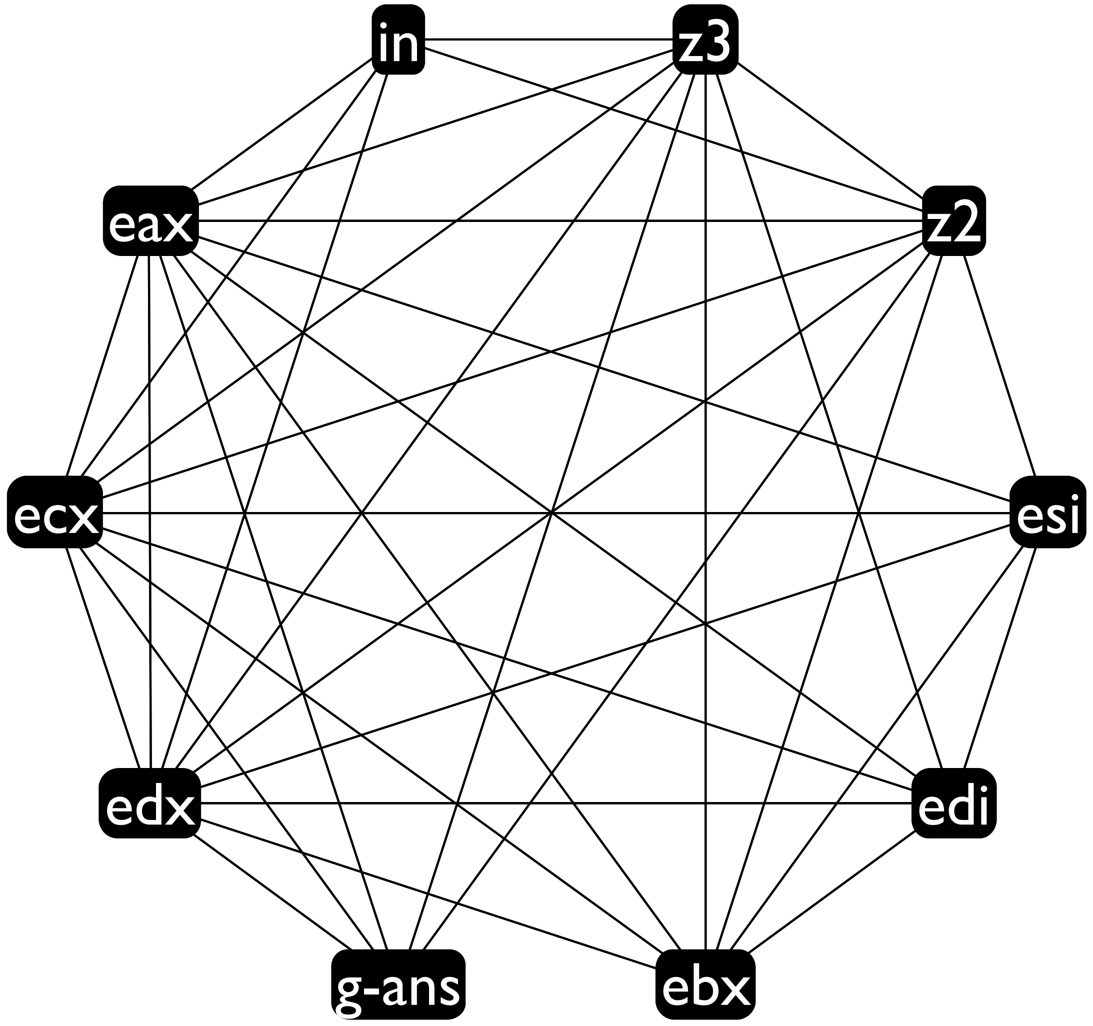
Spilling z1

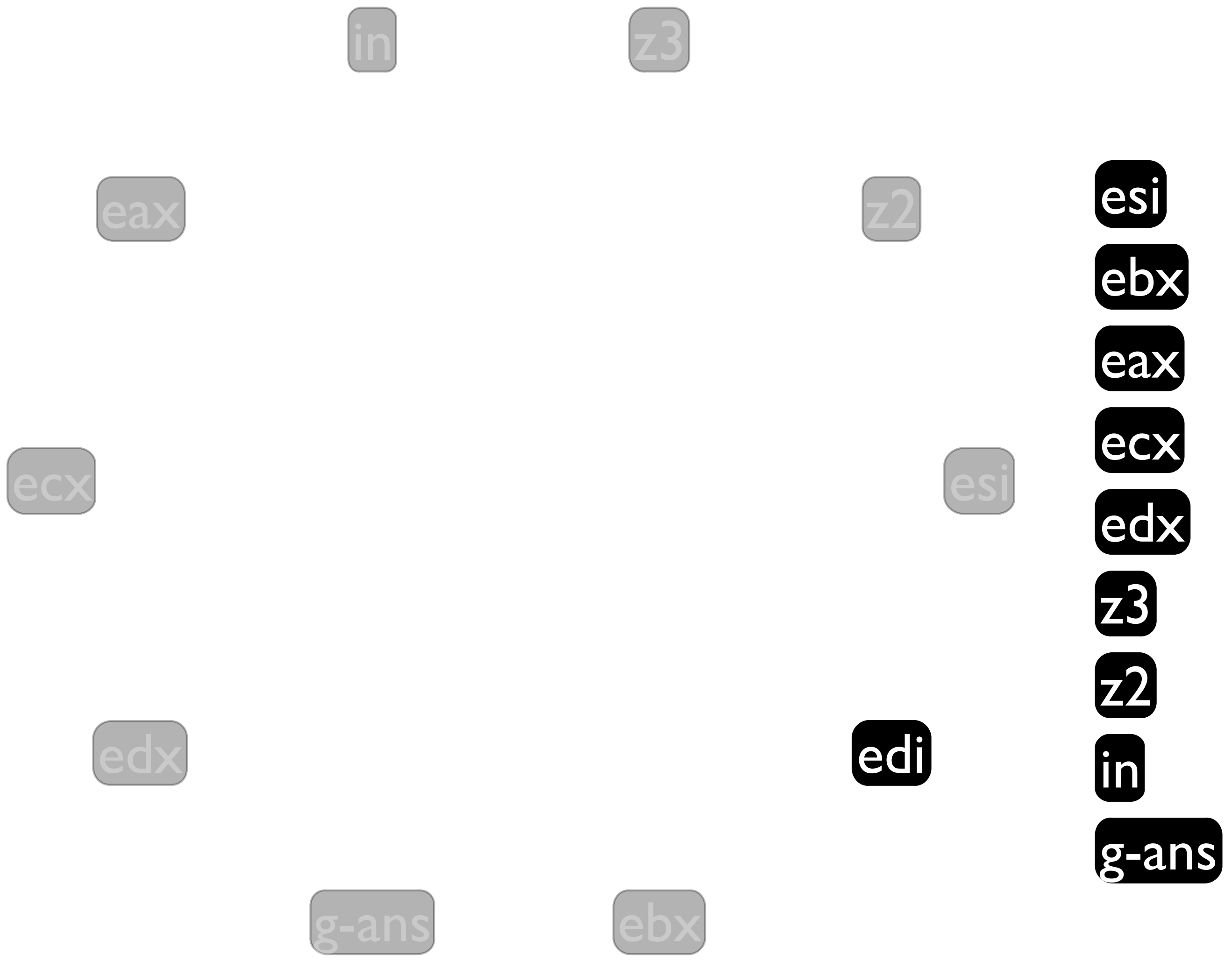
Before:

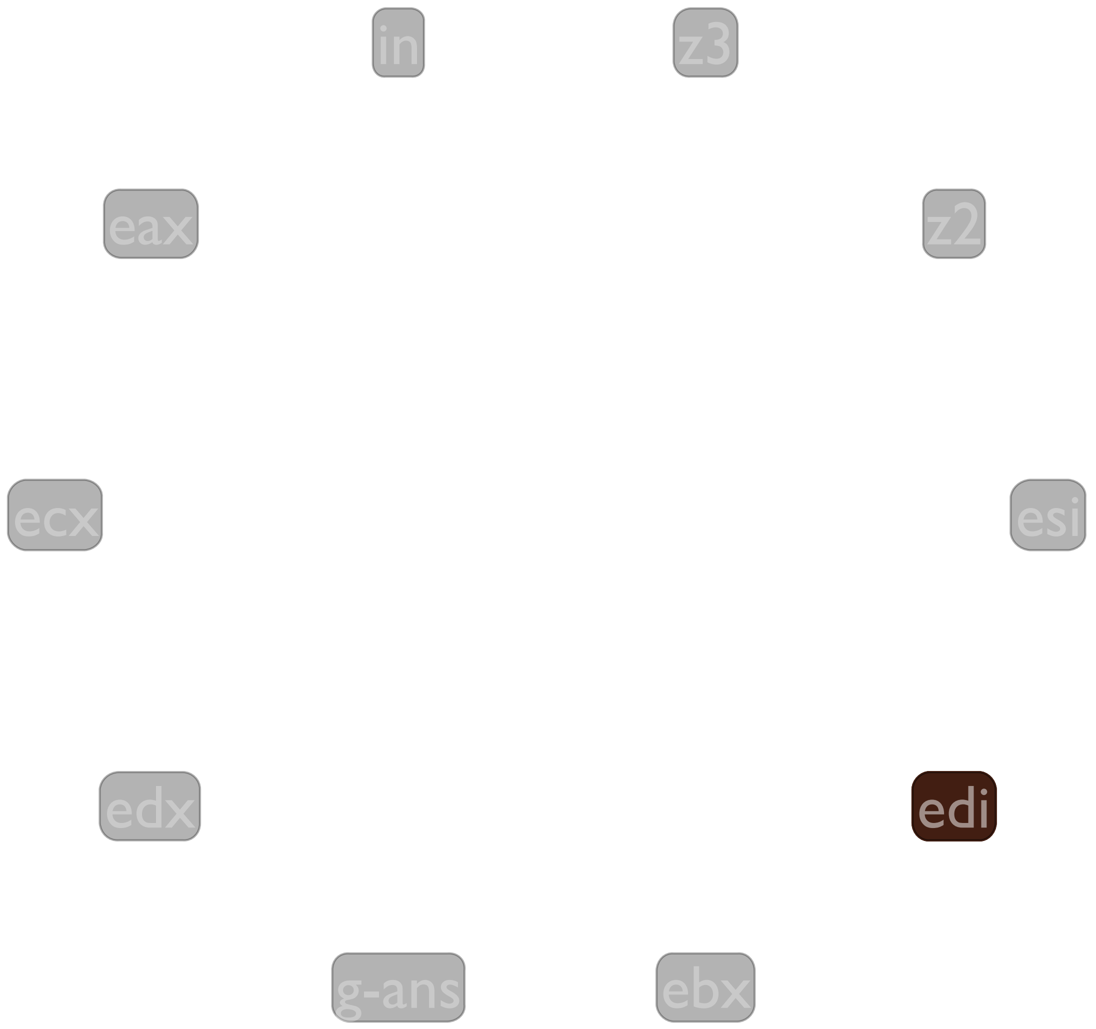
```
:f
(z1 <- ebx)
(z2 <- edi)
(z3 <- esi)
(in <- edx)
(call :g)
(edx <- in)
(g-ans <- eax)
(call :h)
(eax += g-ans)
(ebx <- z1)
(edi <- z2)
(esi <- z3)
(return)
```

After:

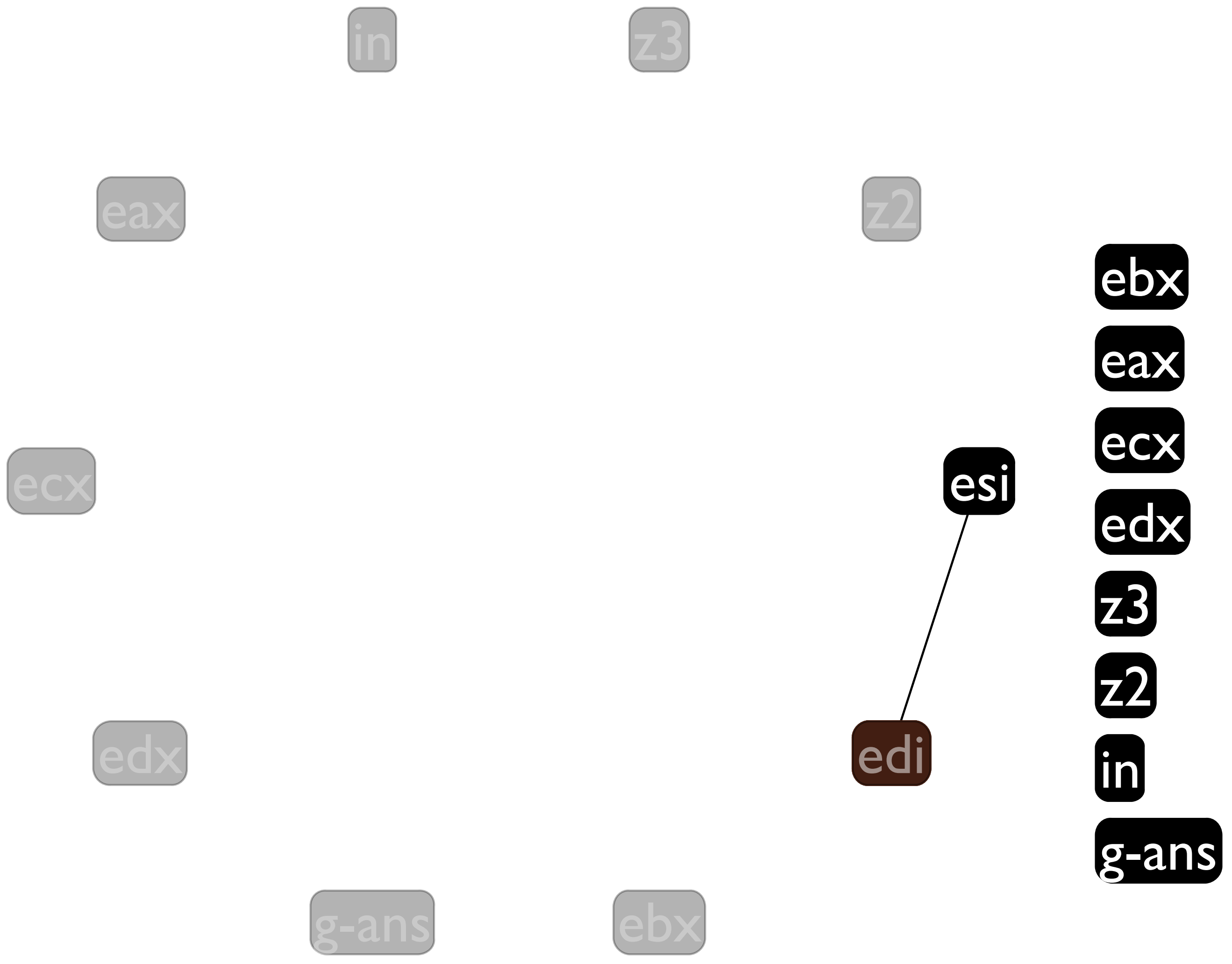
```
:f
((mem ebp -4) <- ebx)
(z2 <- edi)
(z3 <- esi)
(in <- edx)
(call :g)
(edx <- in)
(g-ans <- eax)
(call :h)
(eax += g-ans)
(ebx <- (mem ebp -4))
(edi <- z2)
(esi <- z3)
(return)
```

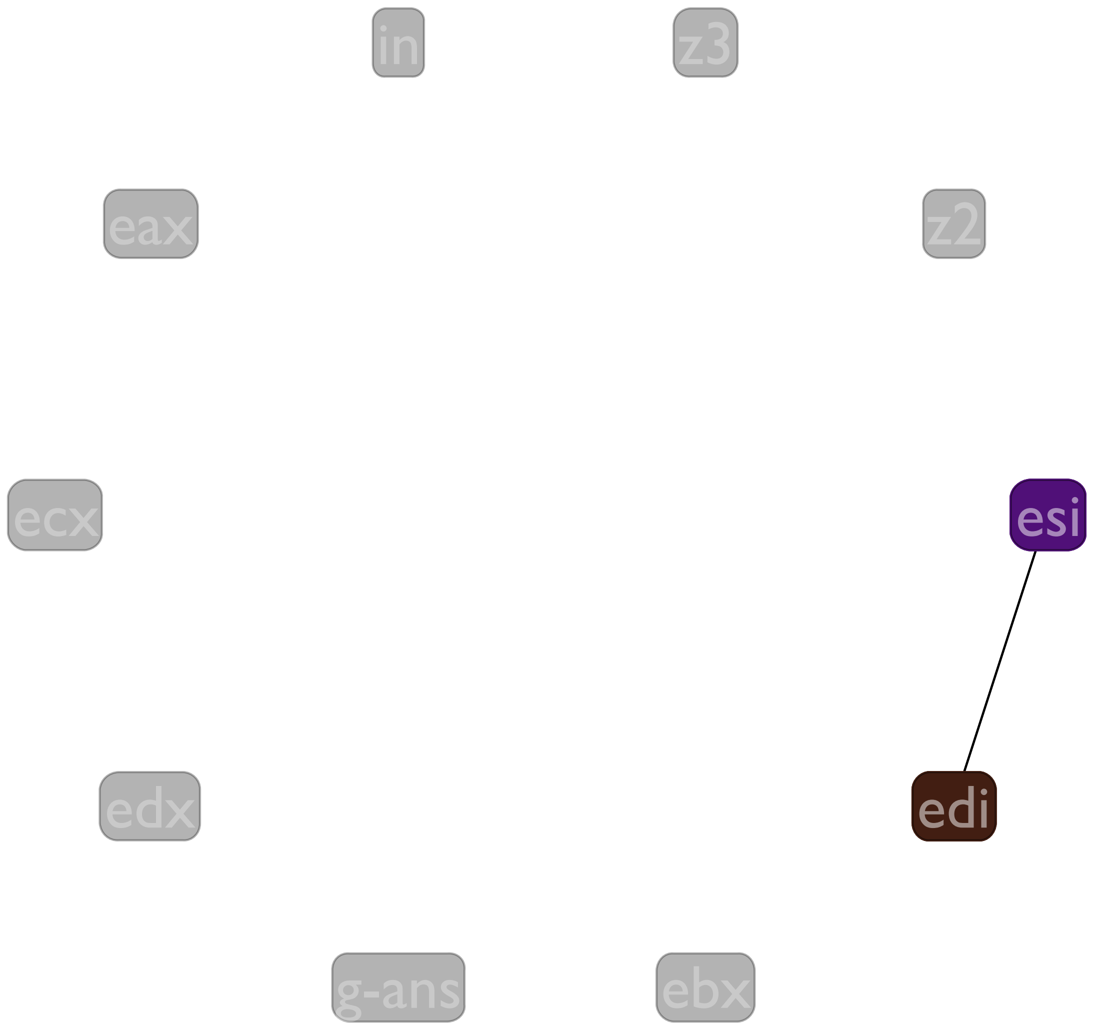

in
z3
eax
z2
ecx
esi
edx
edi
g-ans
ebx


in
z3
eax
z2
ecx
esi
edx
edi
g-ans
ebx
esi
ebx
eax
ecx
edx
z3
z2
in
g-ans

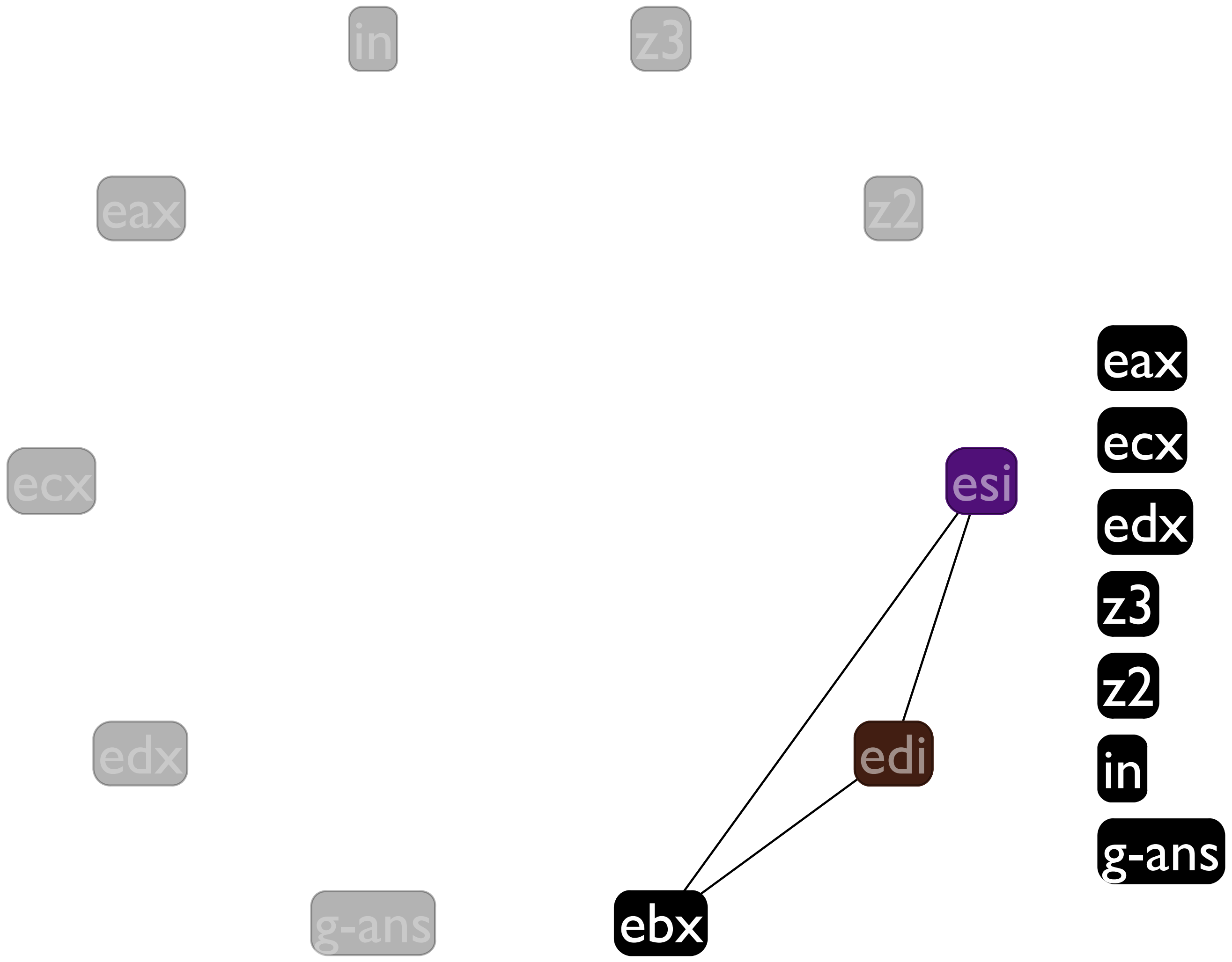


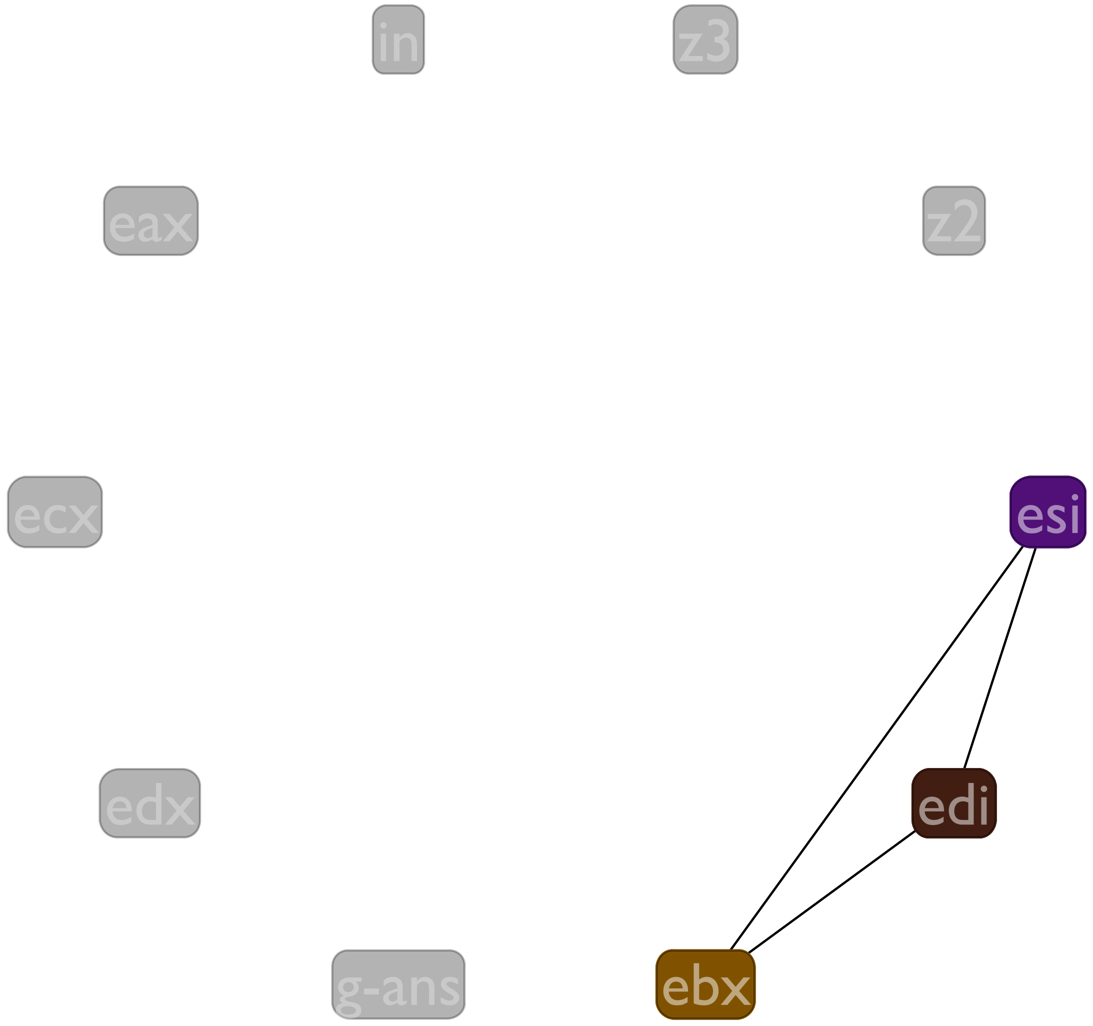
esi
ebx
eax
ecx
edx
z3
z2
in
g-ans


in
z3
eax
z2
ebx
eax
ecx
esi
edx
z3
z2
in
edi
edx
g-ans
g-ans
ebx

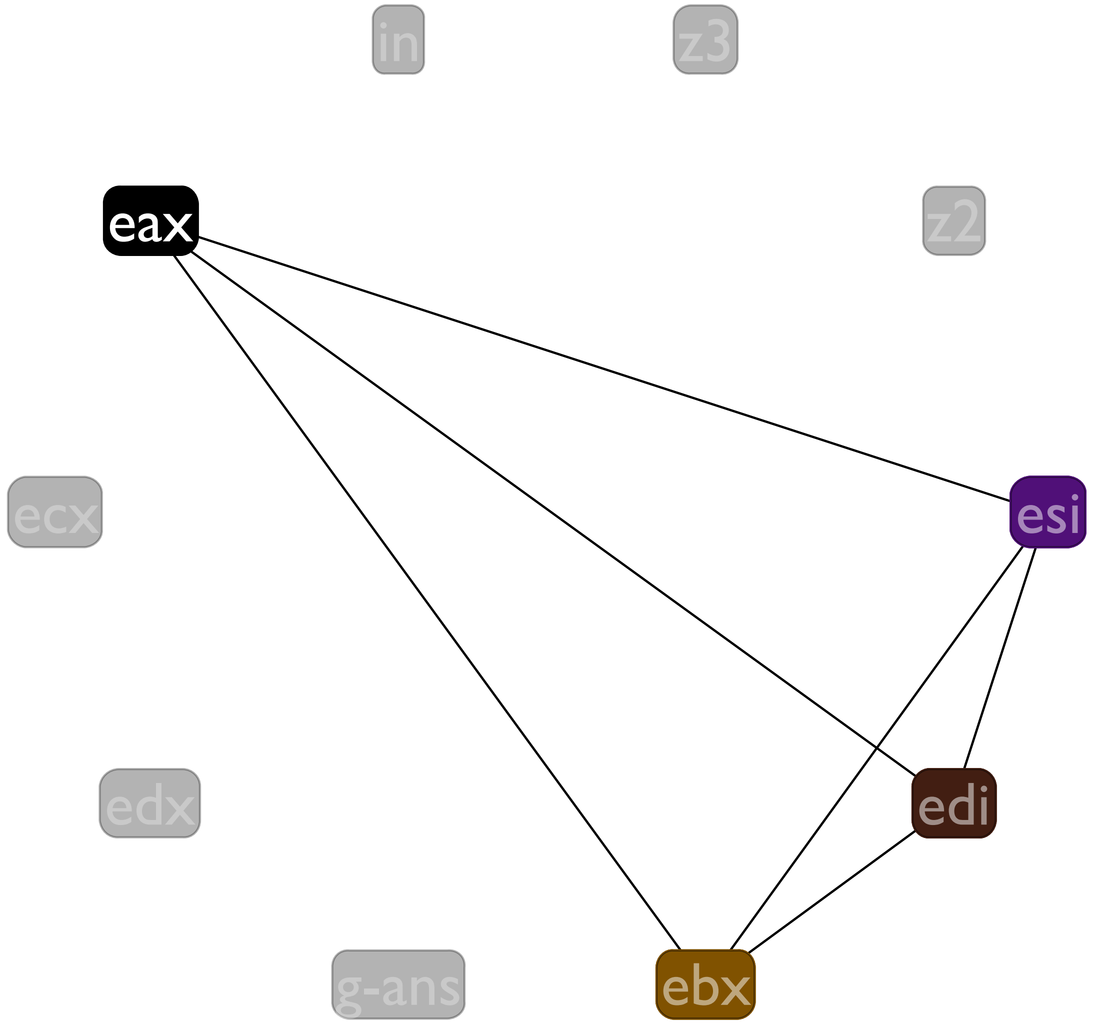


ebx
eax
ecx
edx
z3
z2
in
g-ans

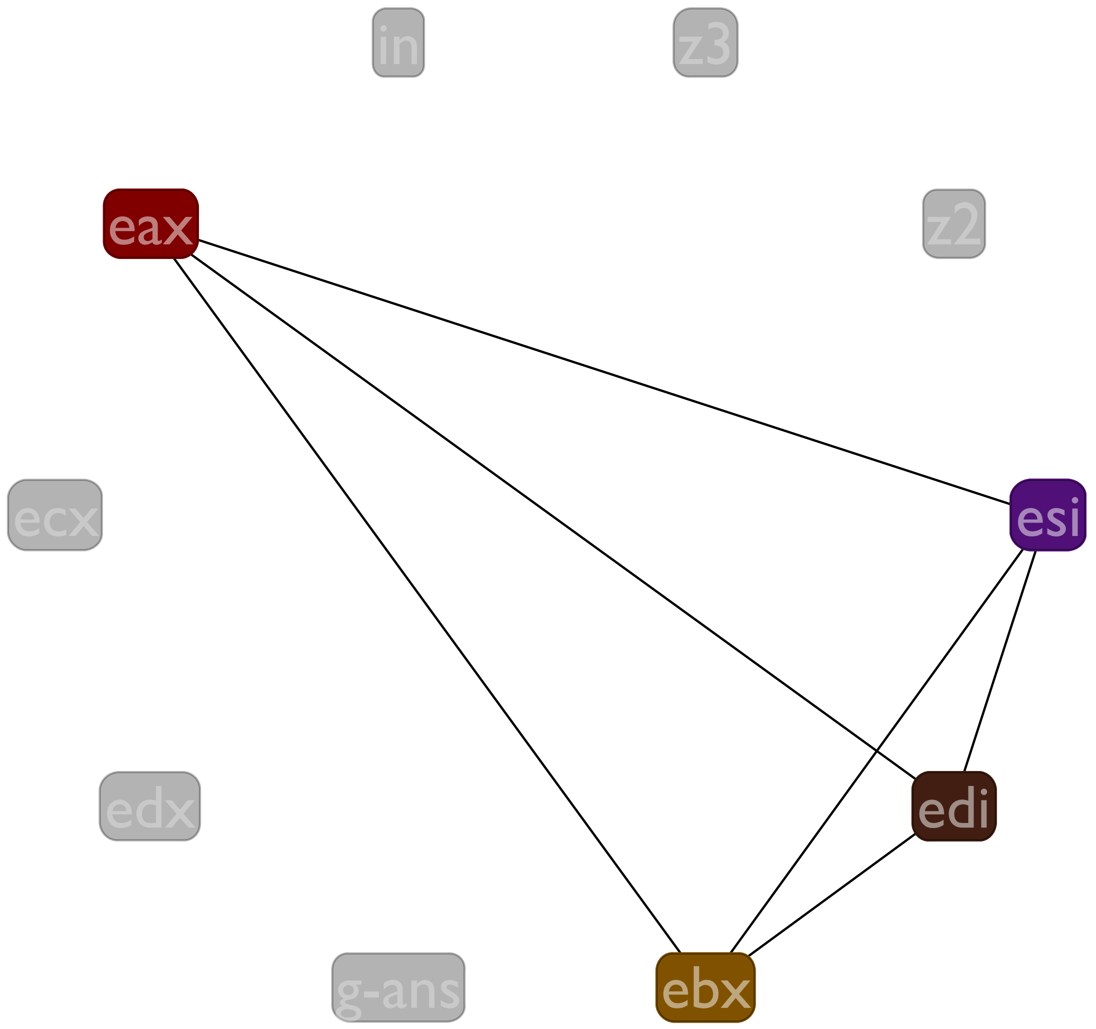

in
z3
eax
z2
ecx
esi
edx
edi
g-ans
ebx
eax
ecx
edx
z3
z2
in
g-ans



eax
ecx
edx
z3
z2
in
g-ans



ecx
edx
z3
z2
in
g-ans



ecx

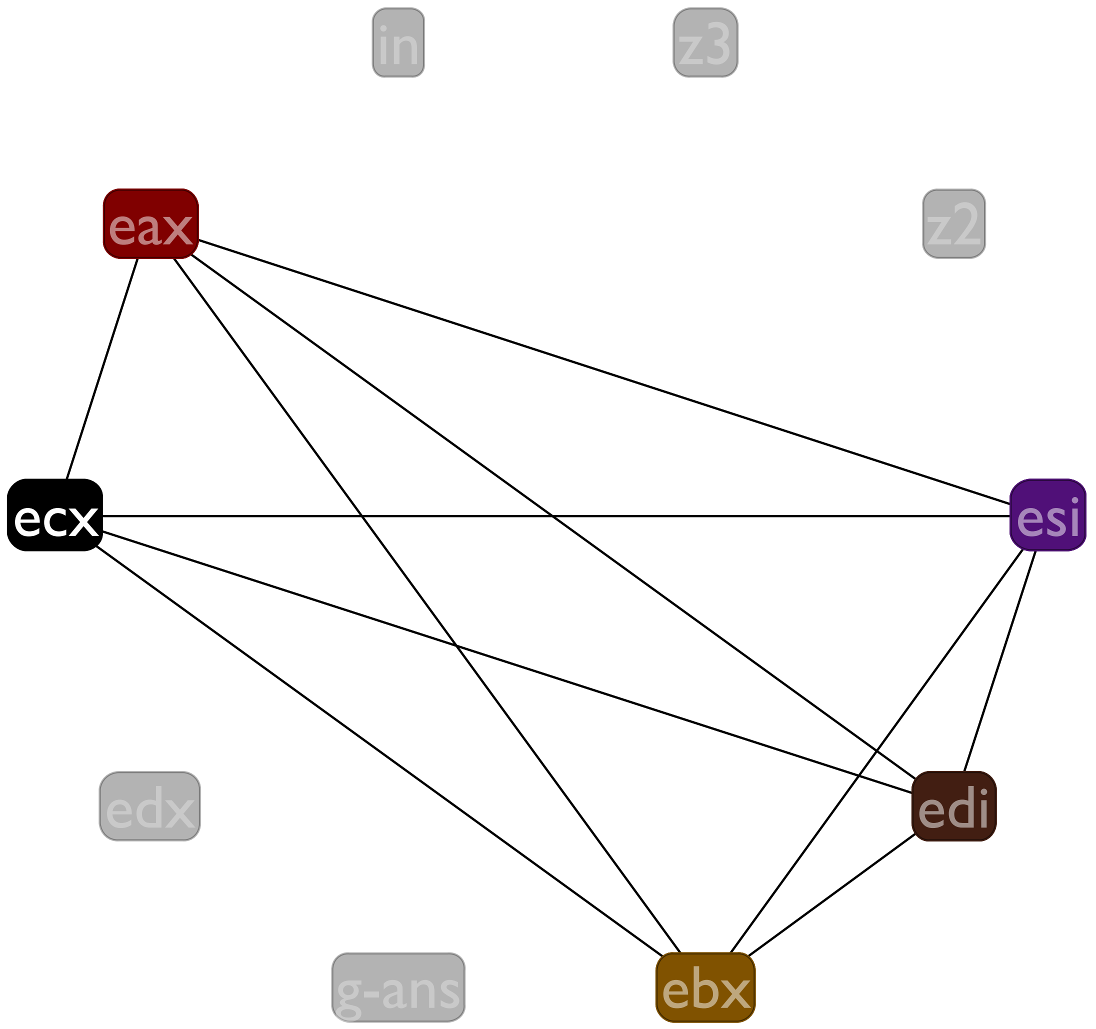
edx

z3

z2

in

g-ans



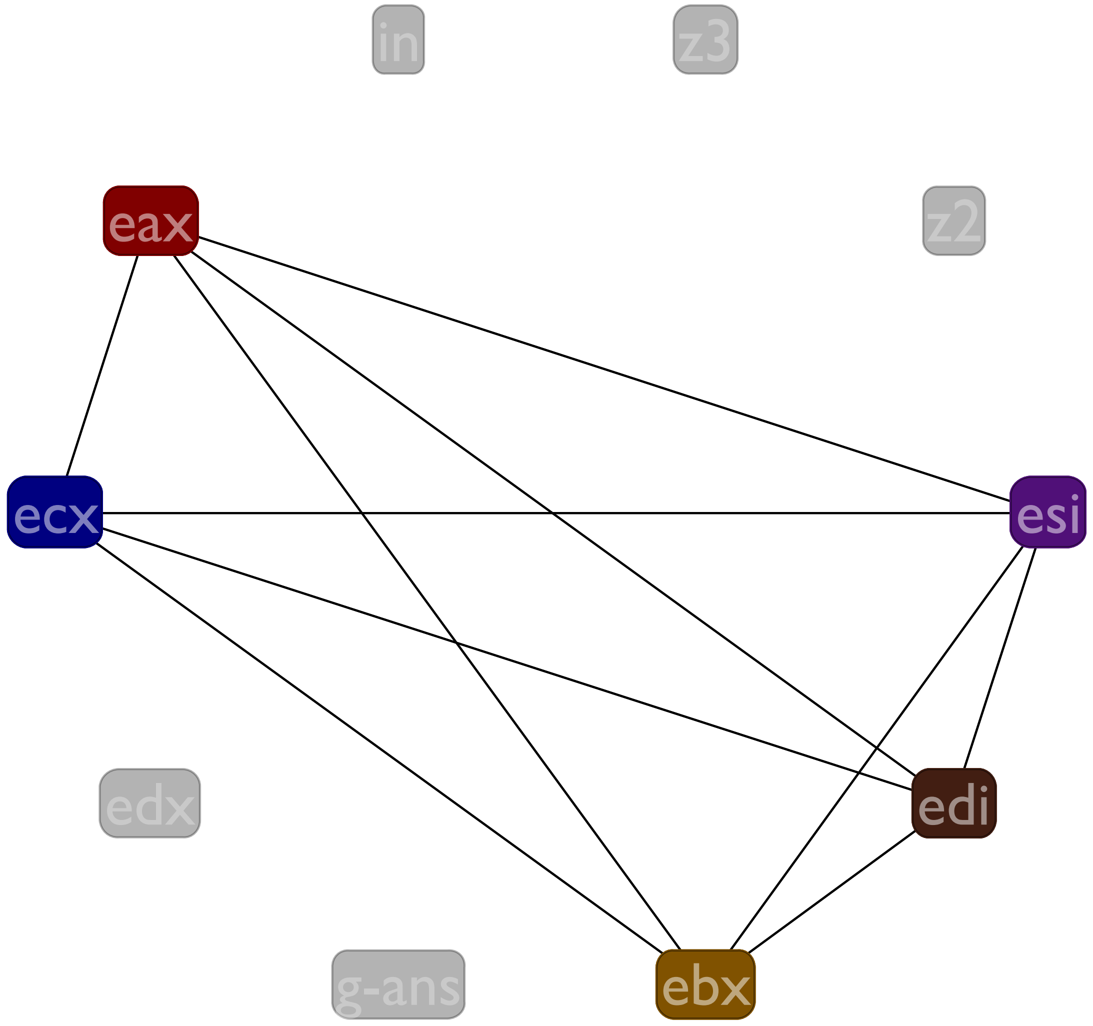
edx

z3

z2

in

g-ans



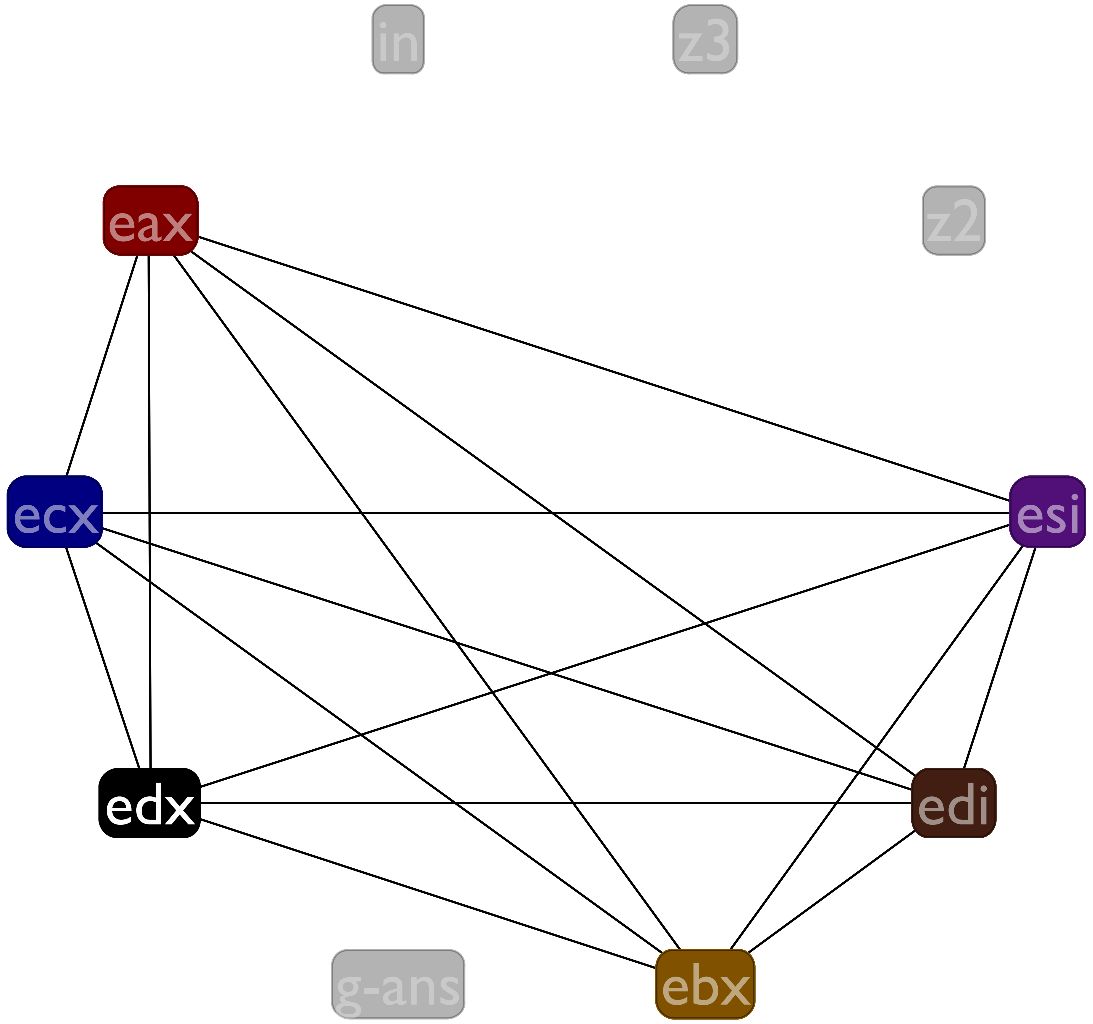
edx

z3

z2

in

g-ans

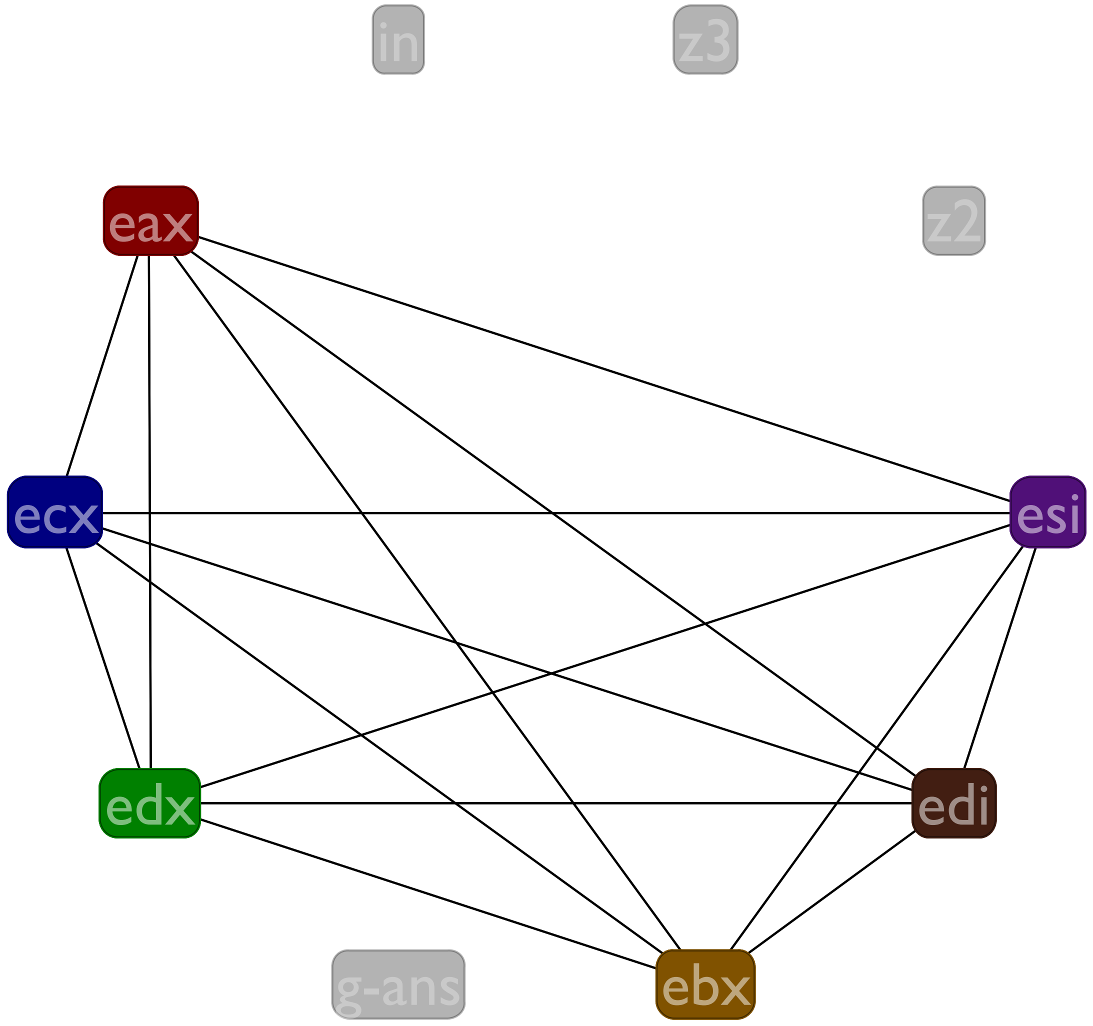


z3

z2

in

g-ans

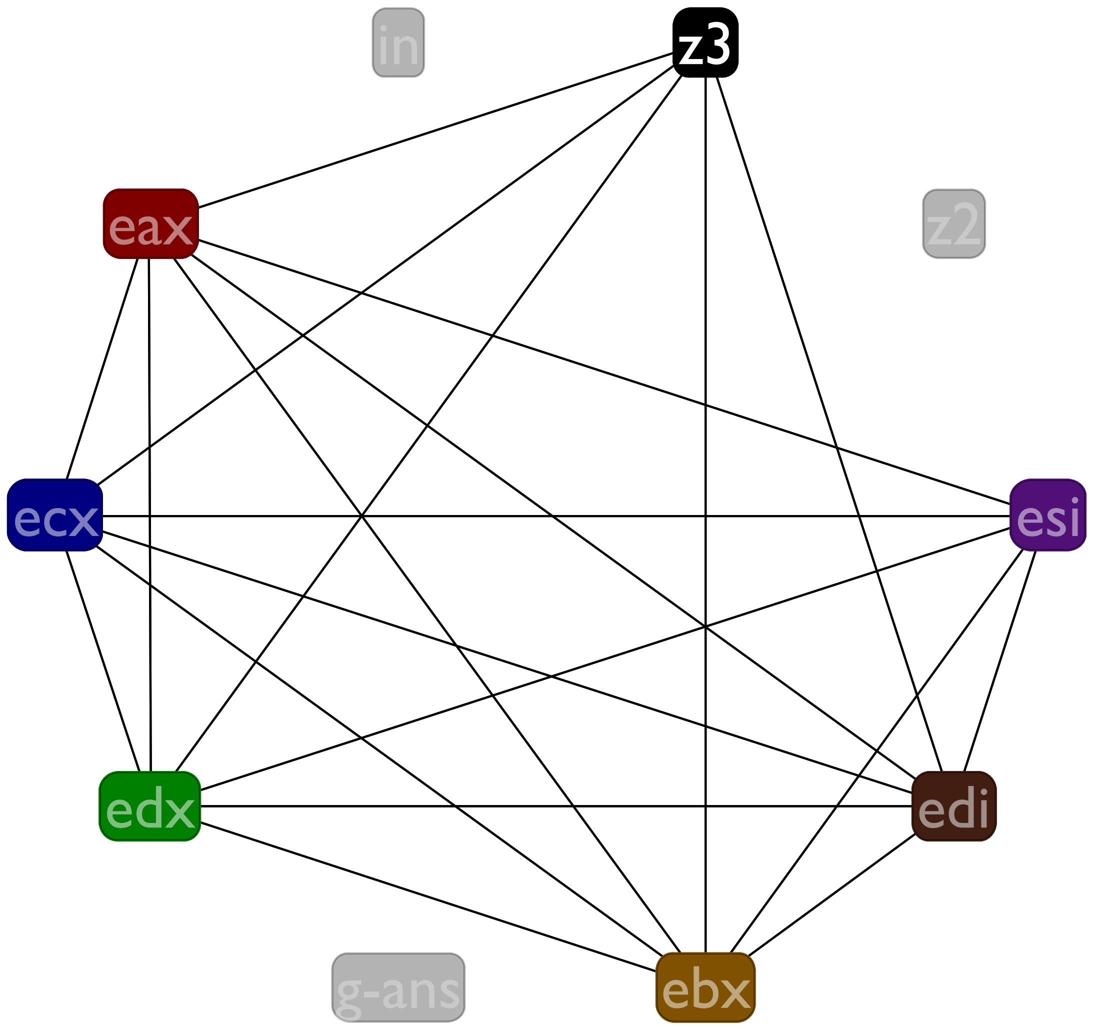


z3

z2

in

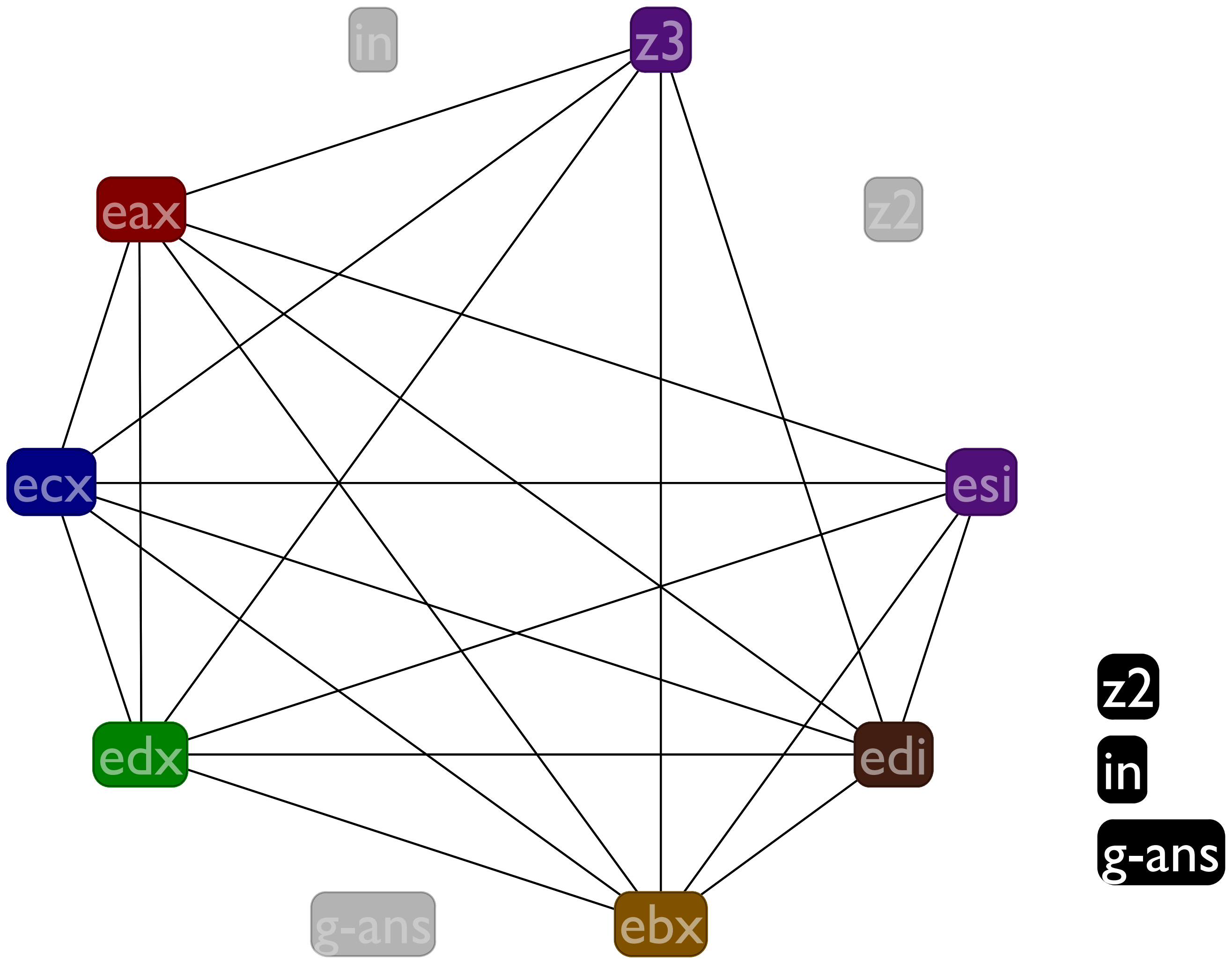
g-ans

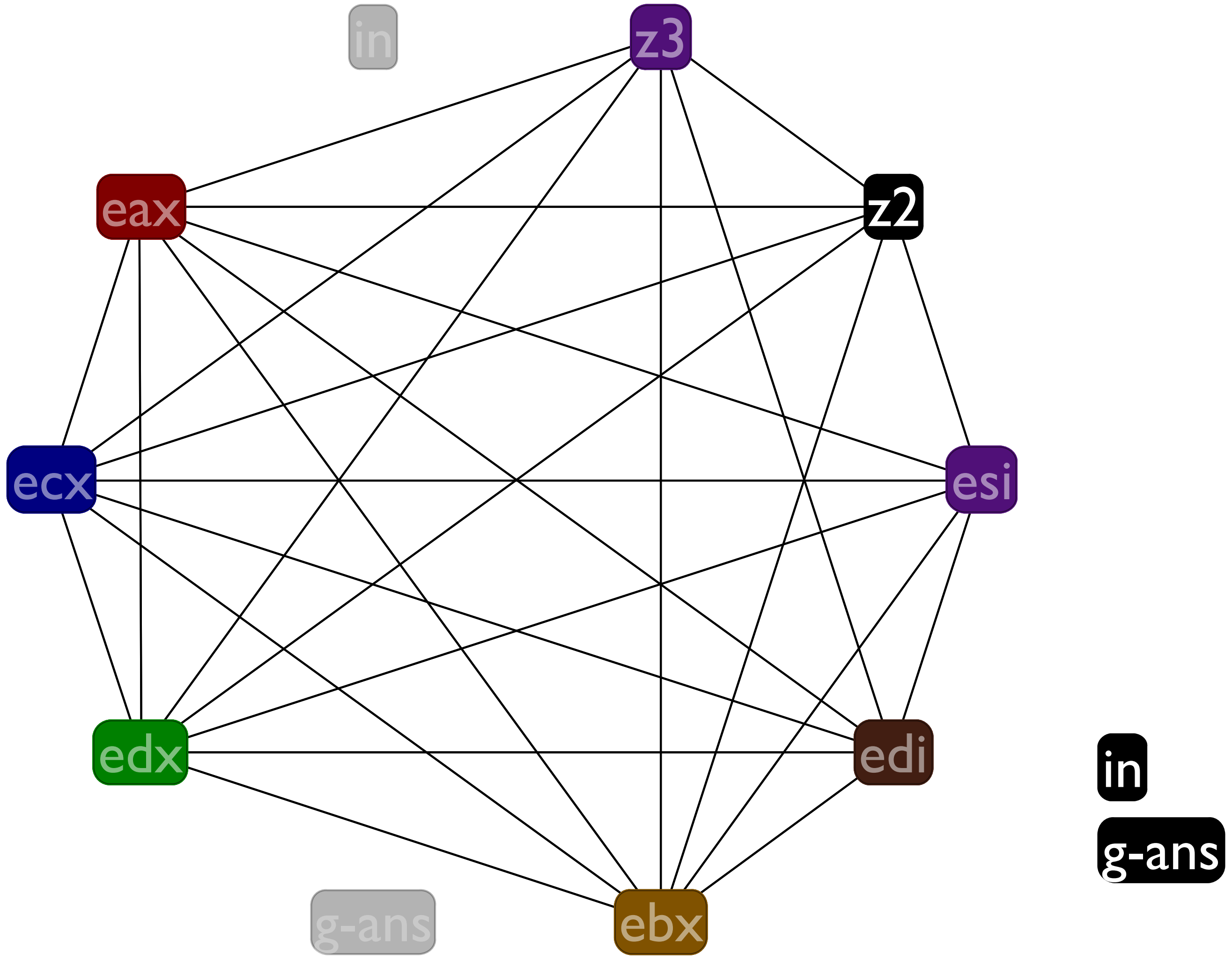


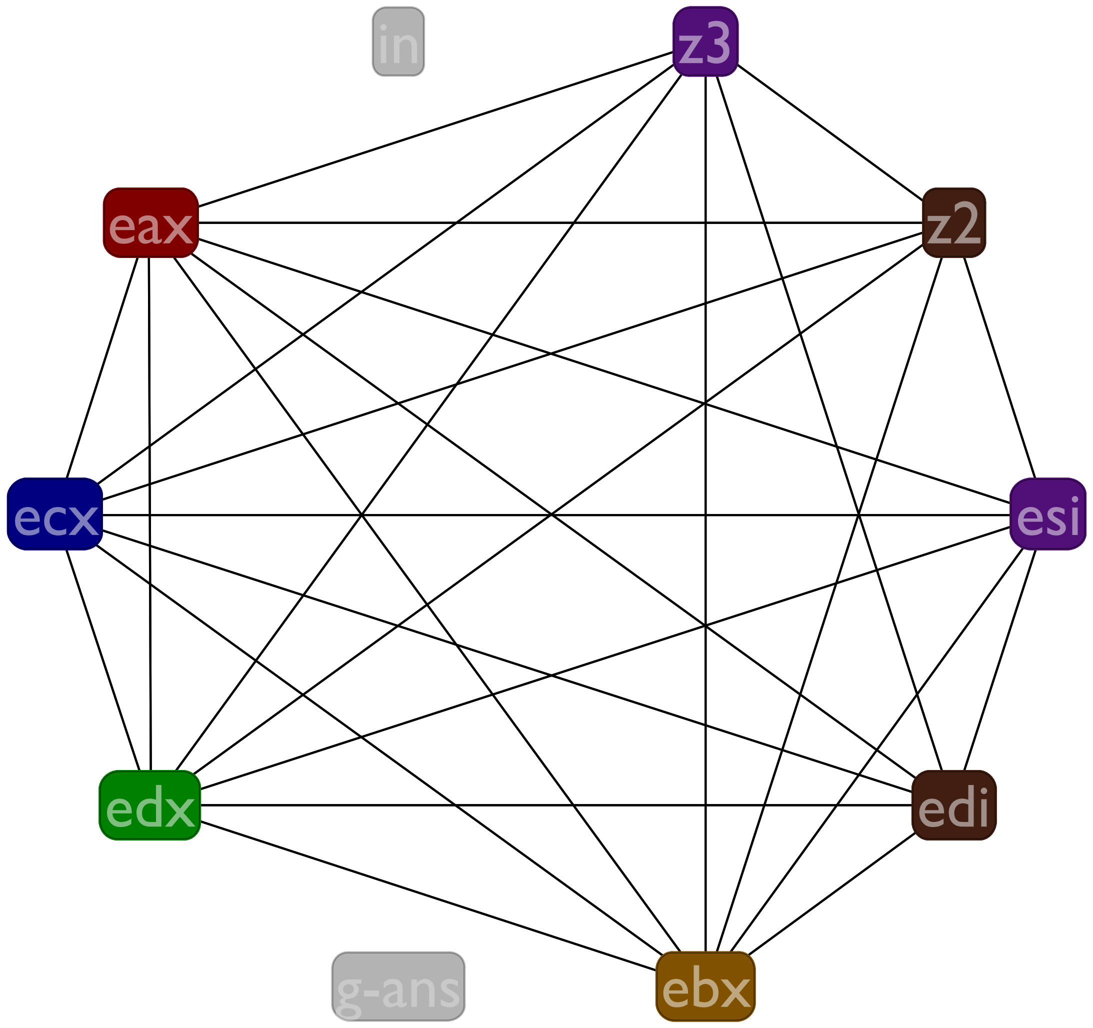
z2

in

g-ans

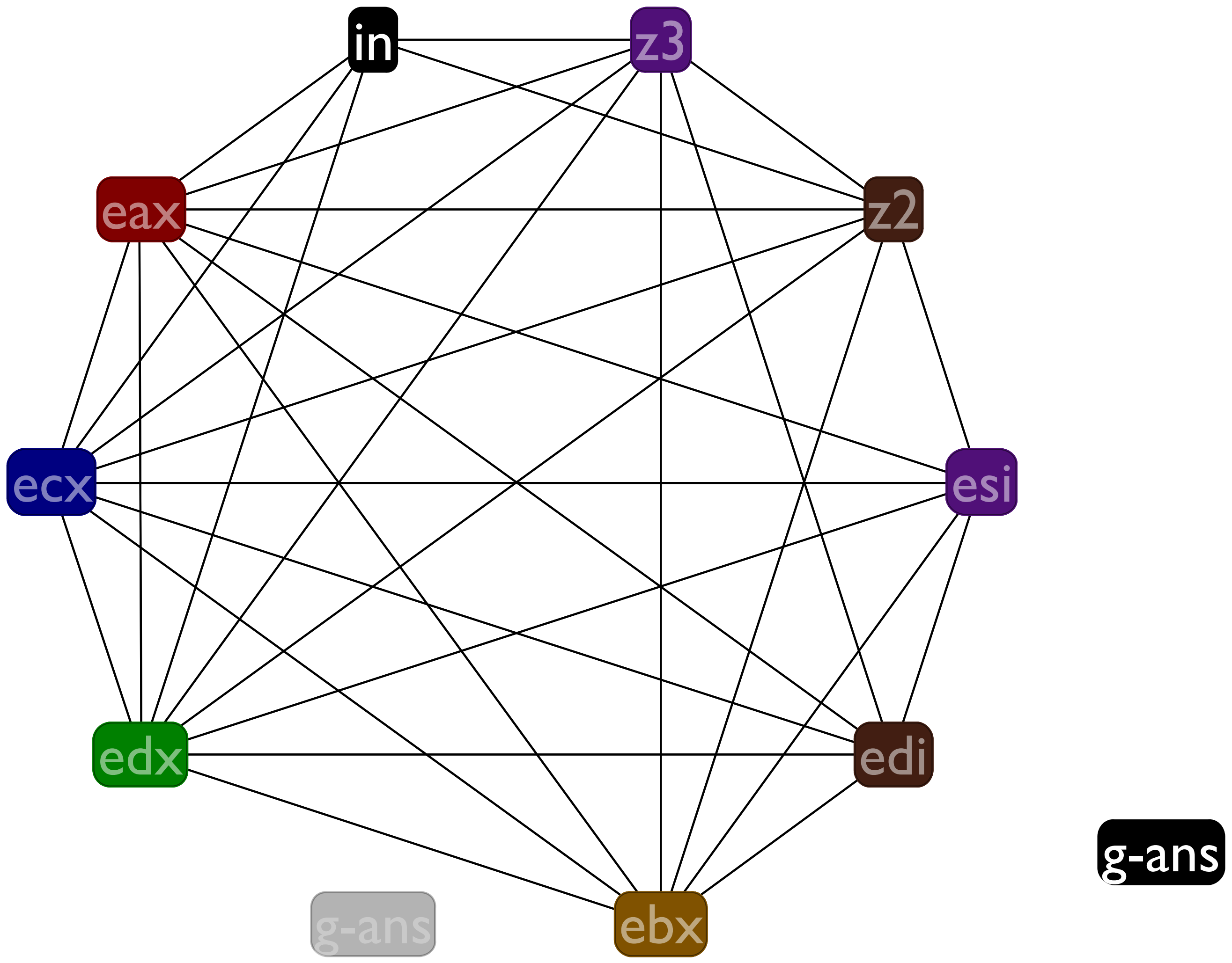

in
z3
eax
z2
ecx
esi
edx
edi
g-ans
ebx
z2
in
g-ans

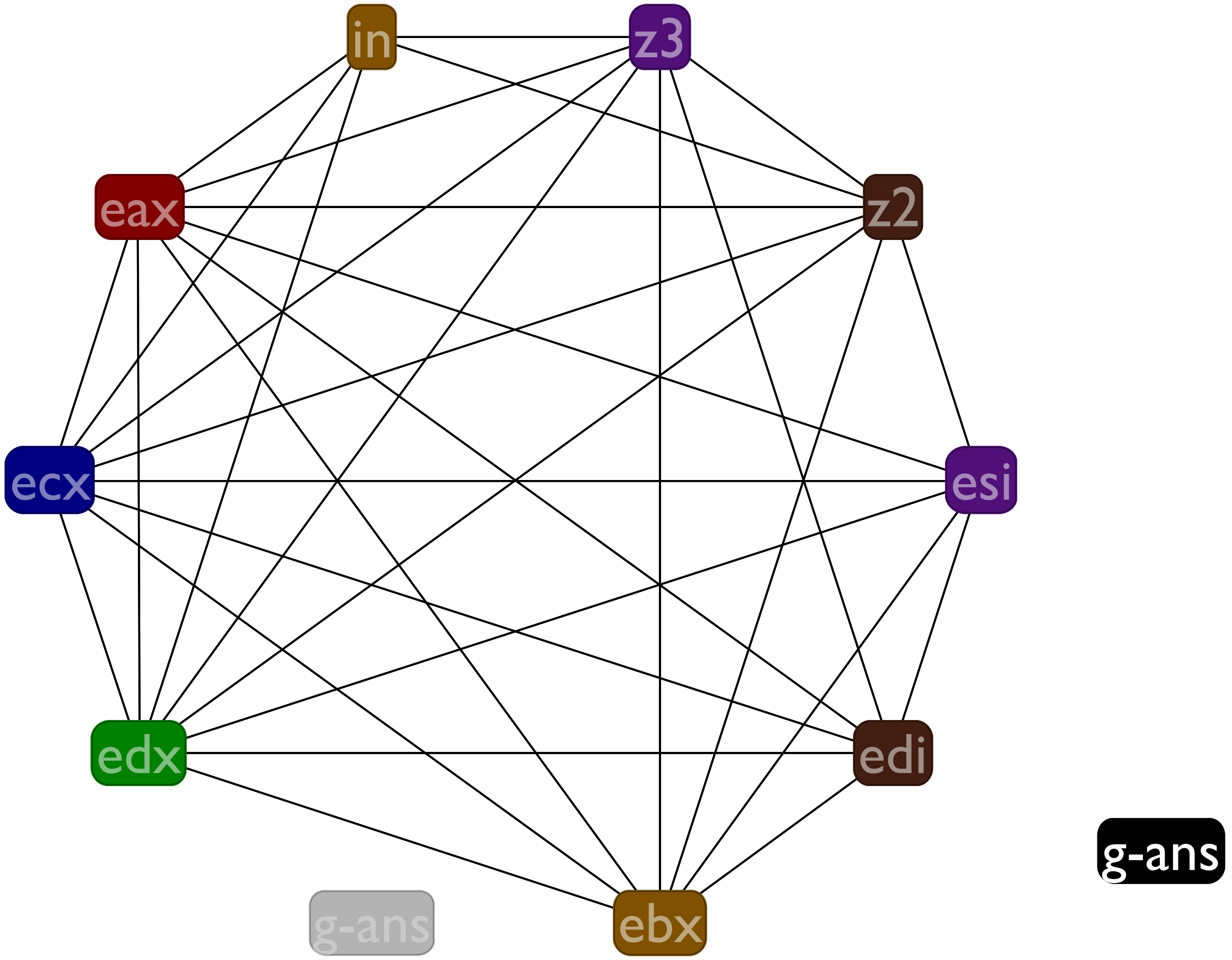

in
z3
eax
z2
ecx
esi
edx
edi
in
g-ans
g-ans
ebx

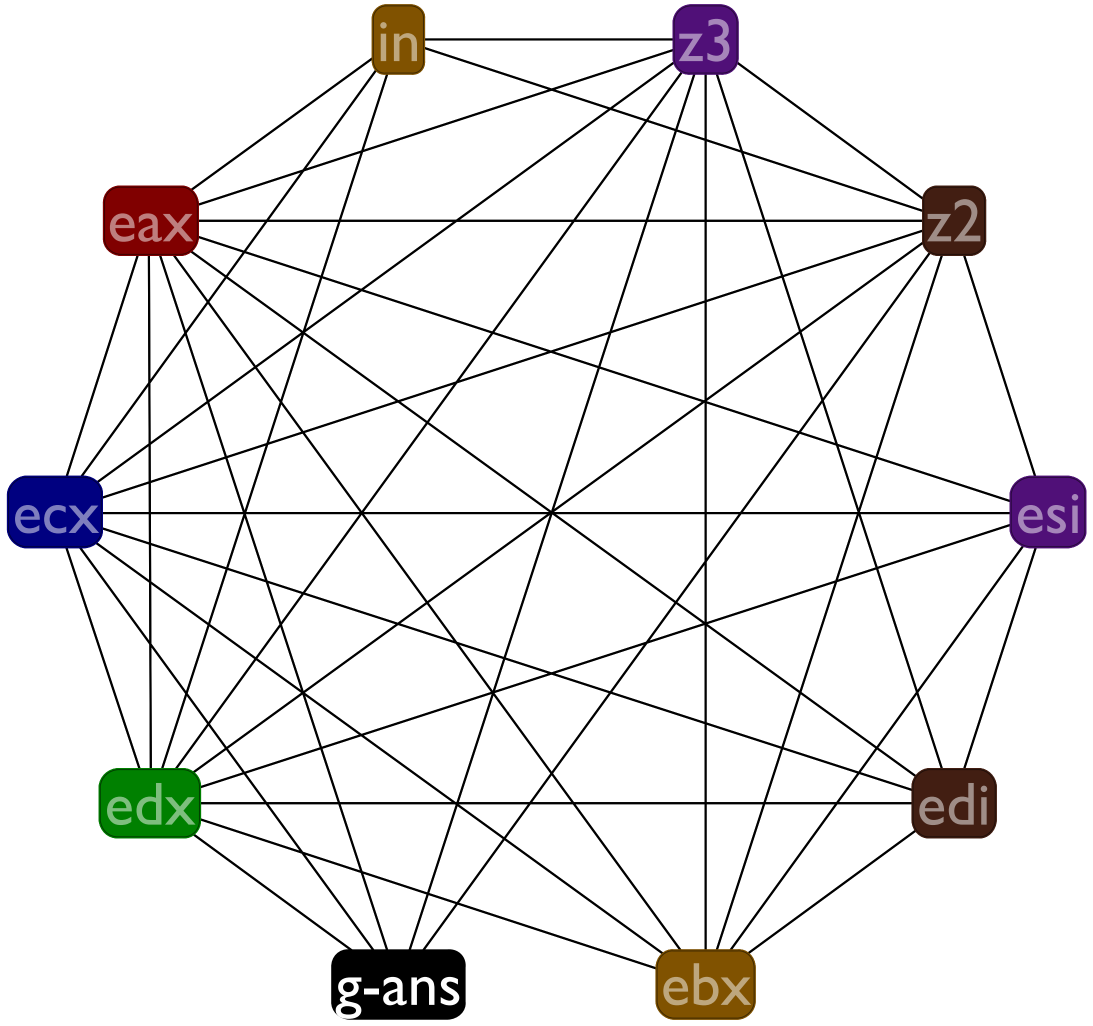


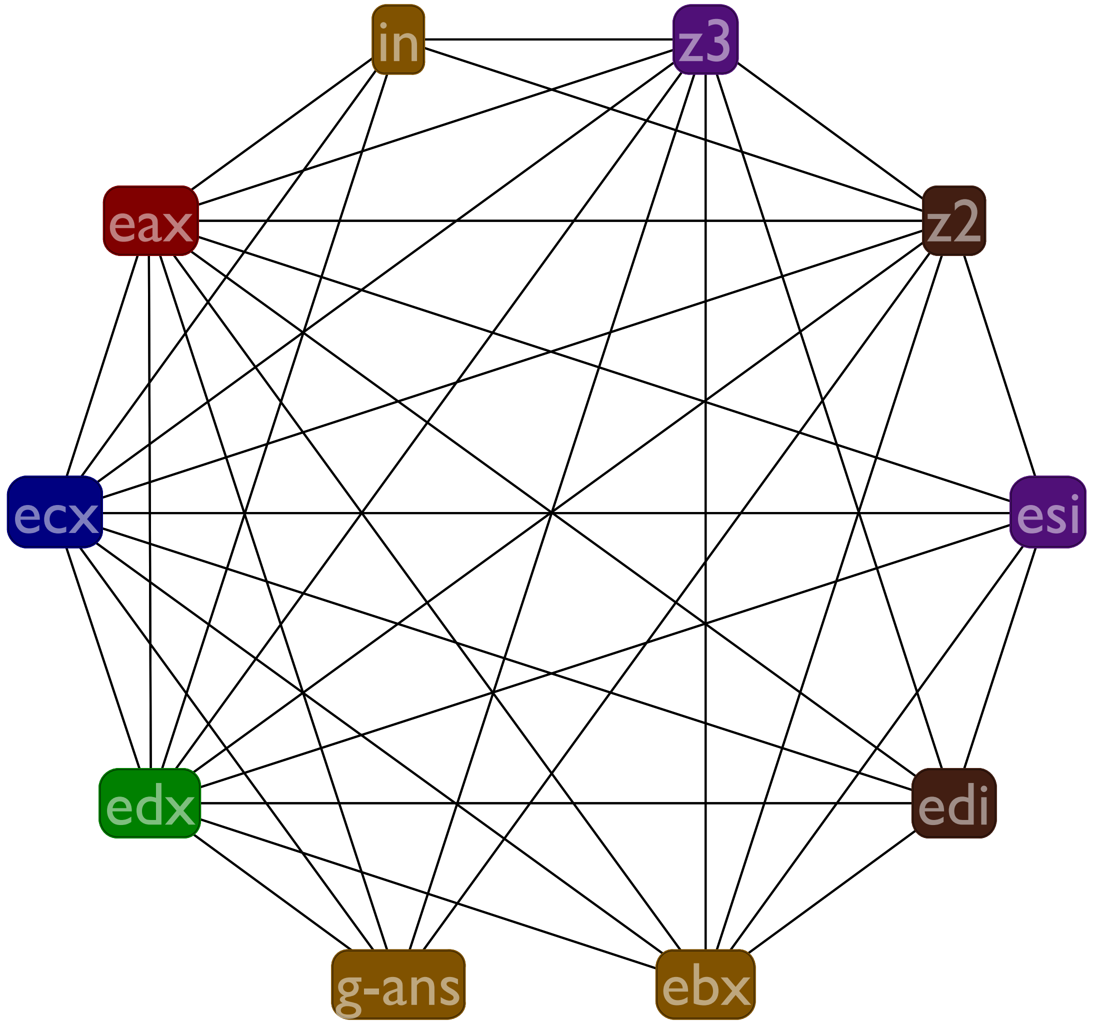
in

g-ans


in
z3
eax
z2
ecx
esi
edx
edi
g-ans
ebx
g-ans


in
z3
eax
z2
ecx
esi
edx
edi
g-ans
ebx
g-ans


in
z3
eax
z2
ecx
esi
edx
edi
g-ans
ebx


in
z3
eax
z2
ecx
esi
edx
edi
g-ans
ebx

How to choose spills

- Pick variables with long live ranges and few uses to spill (callee saves have this profile)
- In this case, saved us from spilling two variables; can just spill one

Coalescing

If we see a `(x <- y)` instruction, we might be able to just change all of the `x`'s into `y`'s

That's called coalescing `x` and `y`

Lets use coalescing to remove `z2` and `z3` from our example program

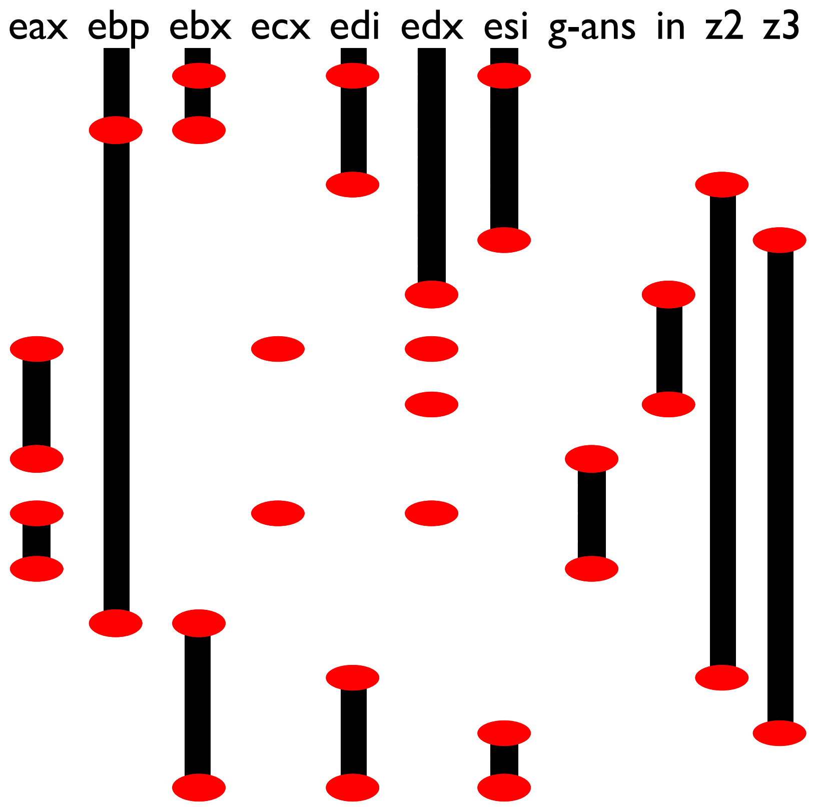
Coalescing example

```
:f
((mem ebp -4) <- ebx)
(z2 <- edi)
(z3 <- esi)
(in <- edx)
(call :g)
(edx <- in)
(g-ans <- eax)
(call :h)
(eax += g-ans)
(ebx <- (mem ebp -4))
(edi <- z2)
(esi <- z3)
(return)
```

```
:f
((mem ebp -4) <- ebx)
(edi <- edi)
(esi <- esi)
(in <- edx)
(call :g)
(edx <- in)
(g-ans <- eax)
(call :h)
(eax += g-ans)
(ebx <- (mem ebp -4))
(edi <- edi)
(esi <- esi)
(return)
```

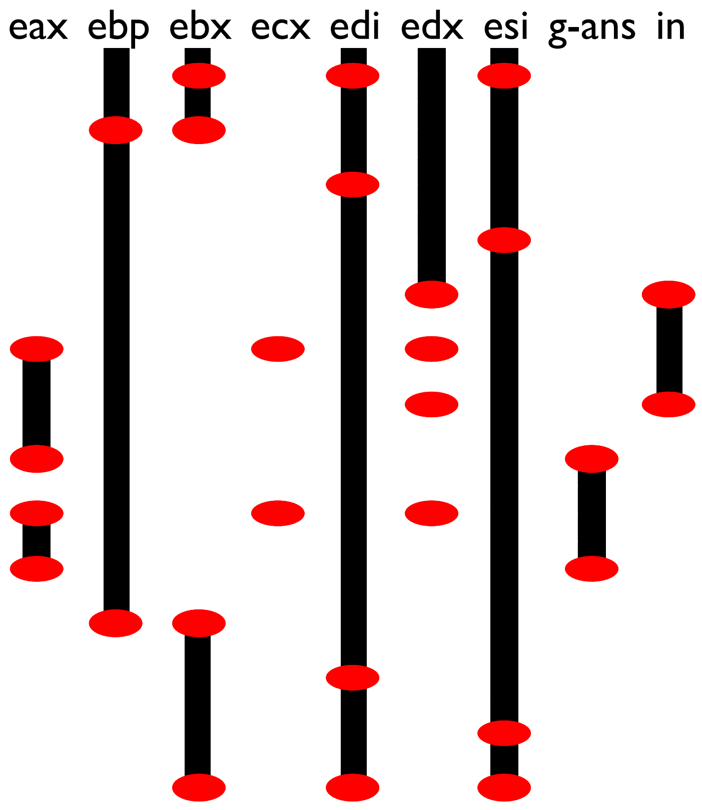
Live ranges before coalescing

```
:f
((mem ebp -4) <- ebx)
(z2 <- edi)
(z3 <- esi)
(in <- edx)
(call :g)
(edx <- in)
(g-ans <- eax)
(call :h)
(eax += g-ans)
(ebx <- (mem ebp -4))
(edi <- z2)
(esi <- z3)
(return)
```

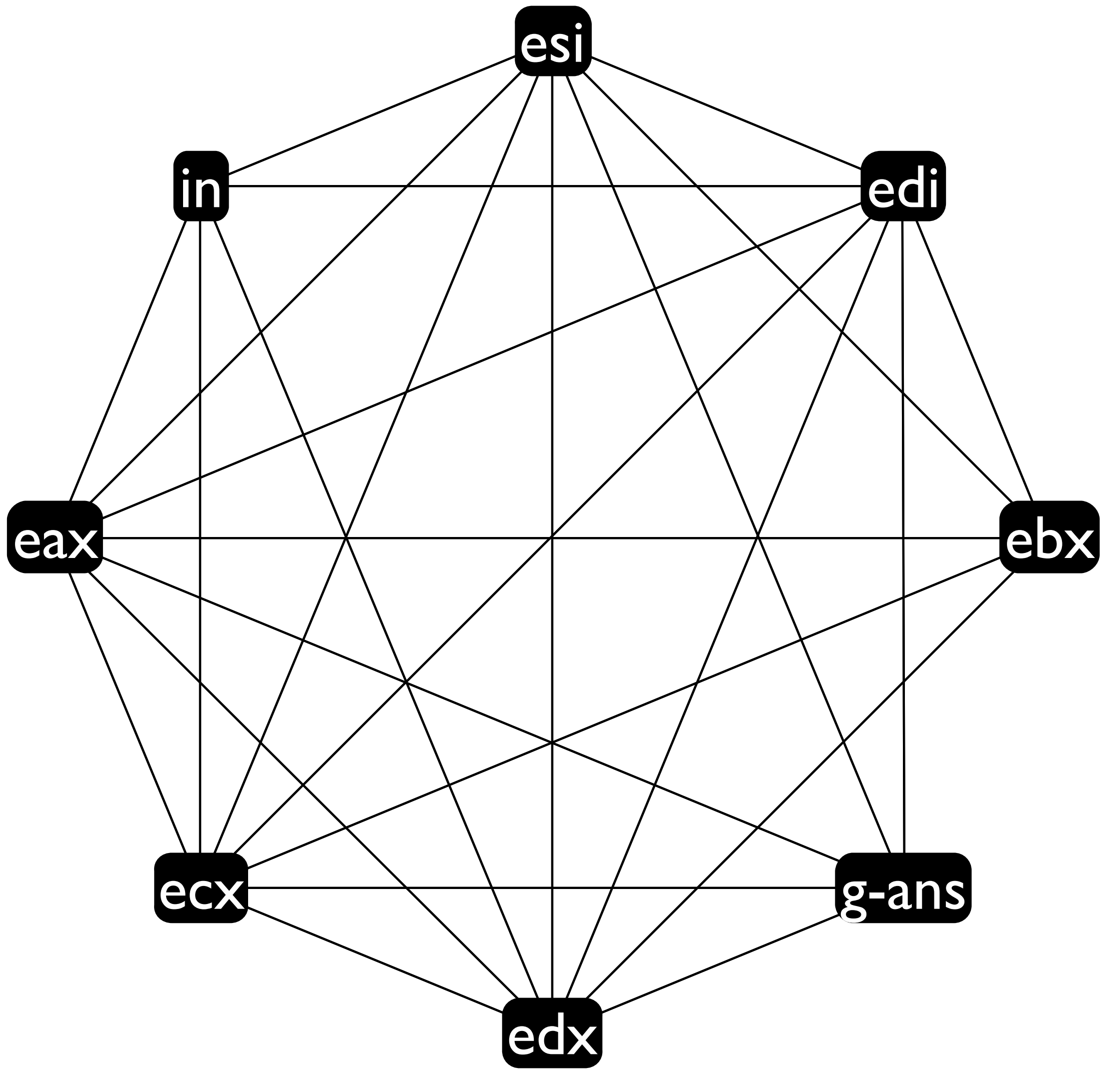


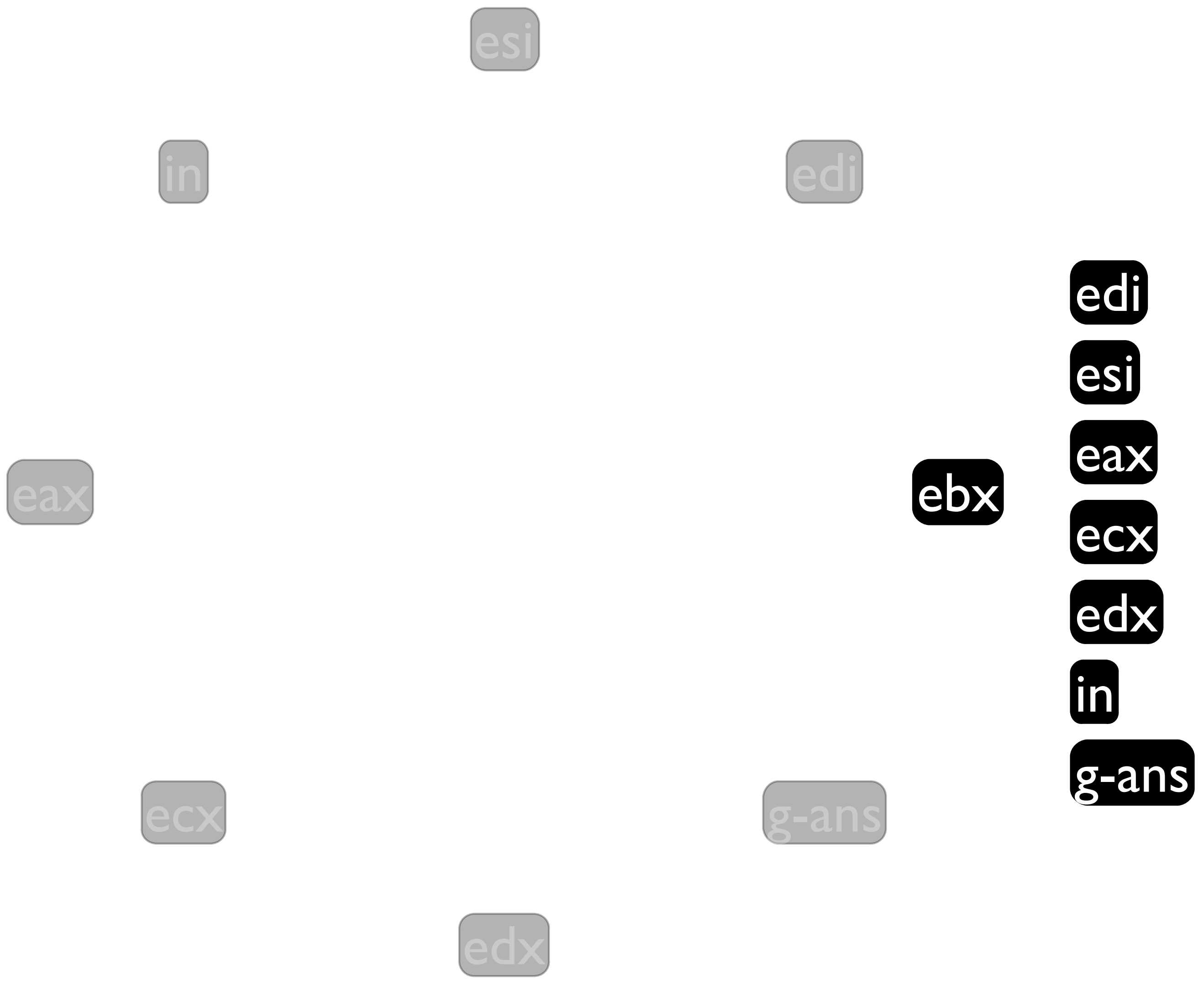
Live ranges after coalescing

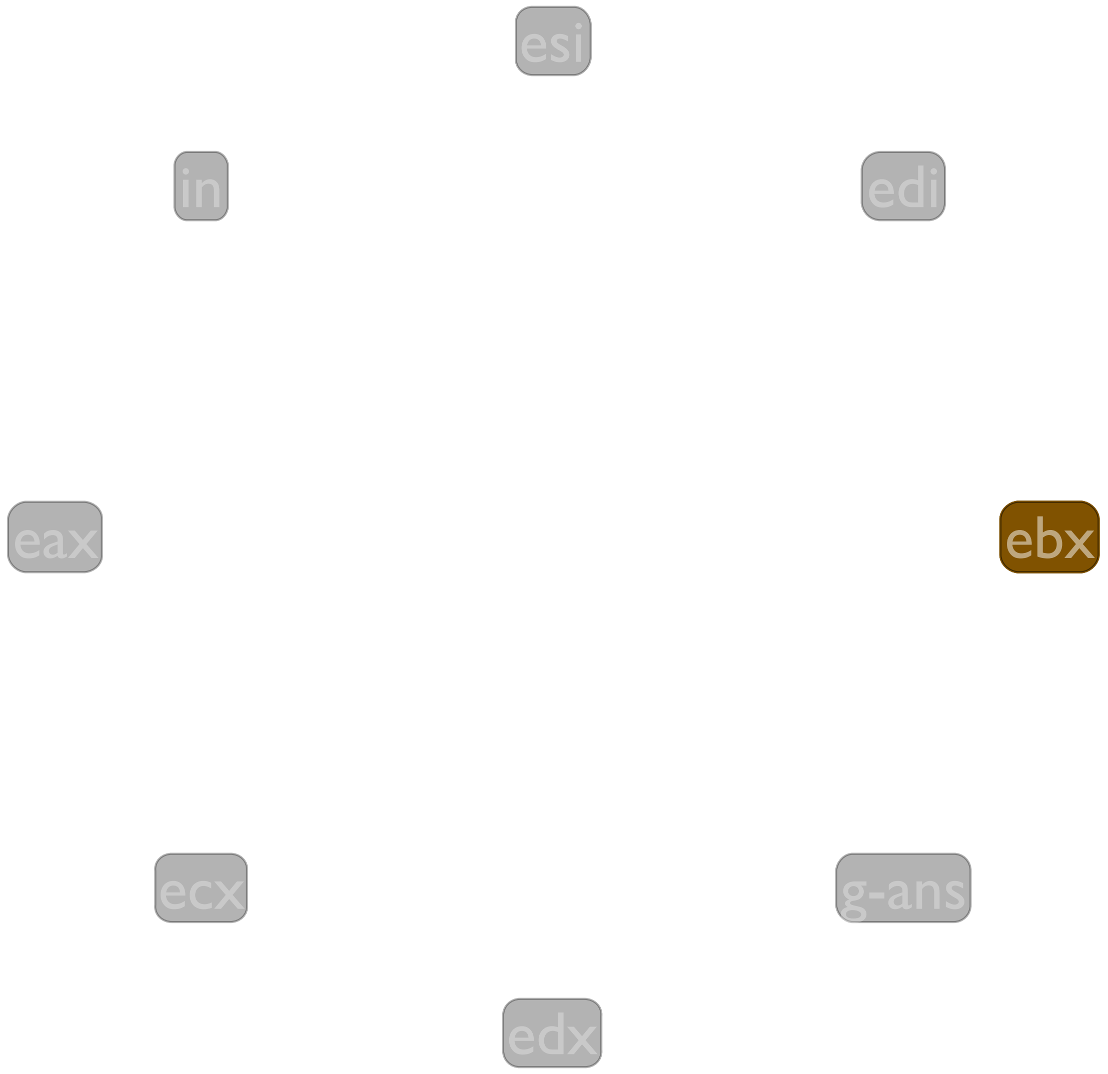
```
:f
((mem ebp -4) <- ebx)
(edi <- edi)
(esi <- esi)
(in <- edx)
(call :g)
(edx <- in)
(g-ans <- eax)
(call :h)
(eax += g-ans)
(ebx <- (mem ebp -4))
(edi <- edi)
(esi <- esi)
(return)
```



Lets try to register allocate the coalesced graph


esi
in
edi
eax
ebx
ecx
g-ans
edx


esi
in
edi
edi
esi
eax
eax
ebx
ecx
edx
in
g-ans
ecx
g-ans
edx



edi

esi

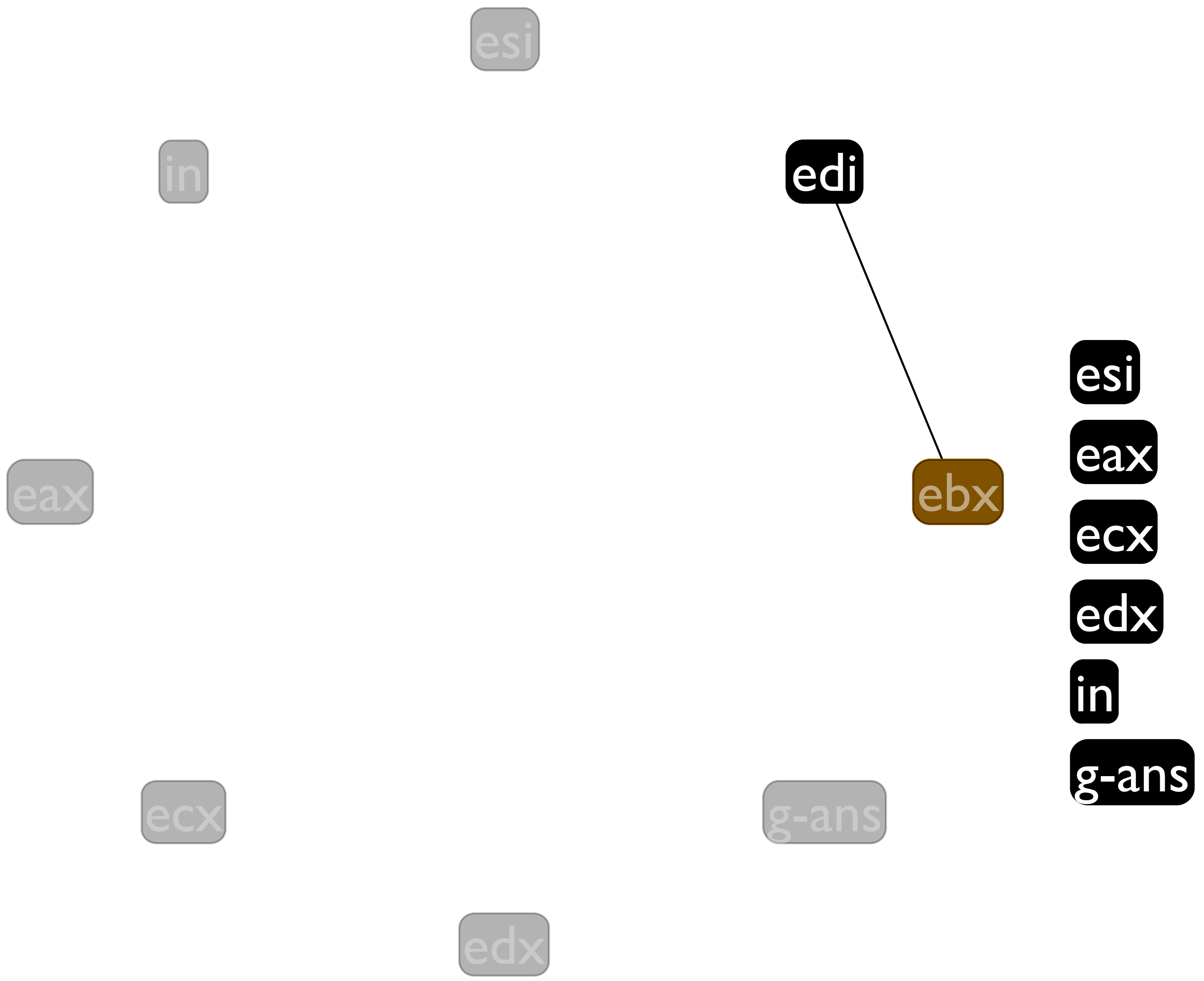
eax

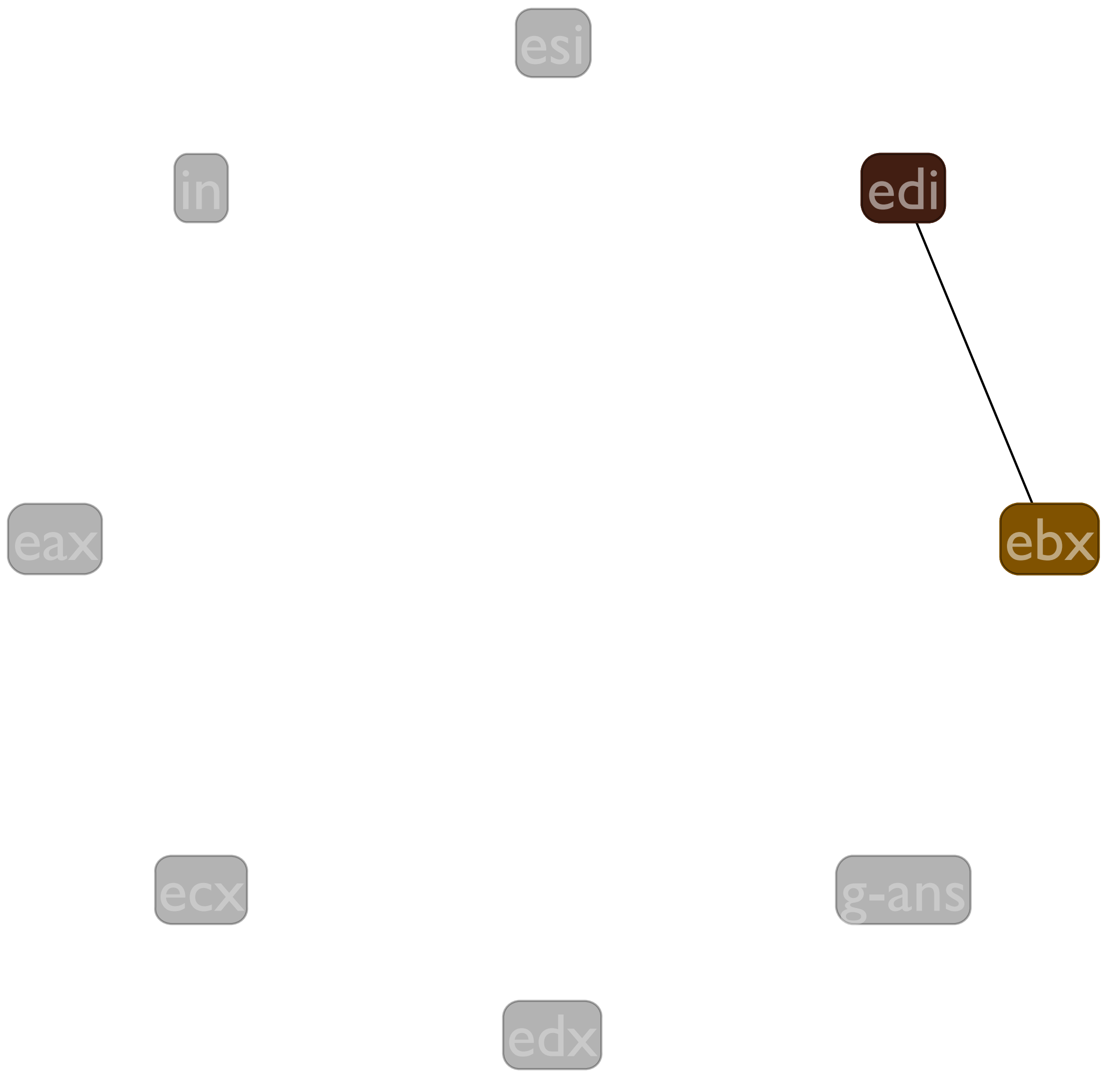
ecx

edx

in

g-ans


esi
in
edi
eax
ebx
ecx
g-ans
edx
esi
eax
ecx
edx
in
g-ans



esi

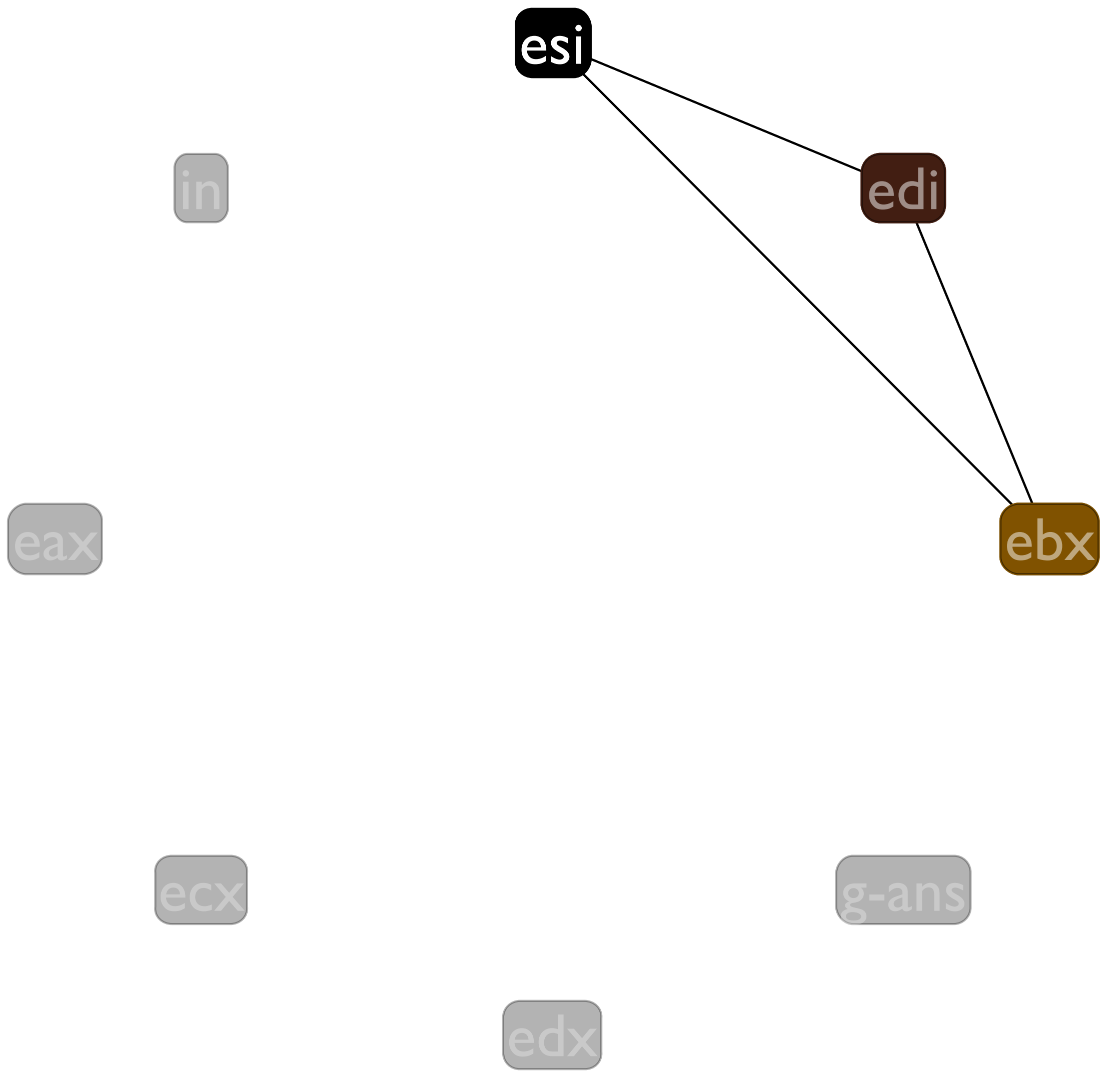
eax

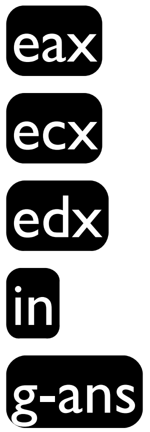
ecx

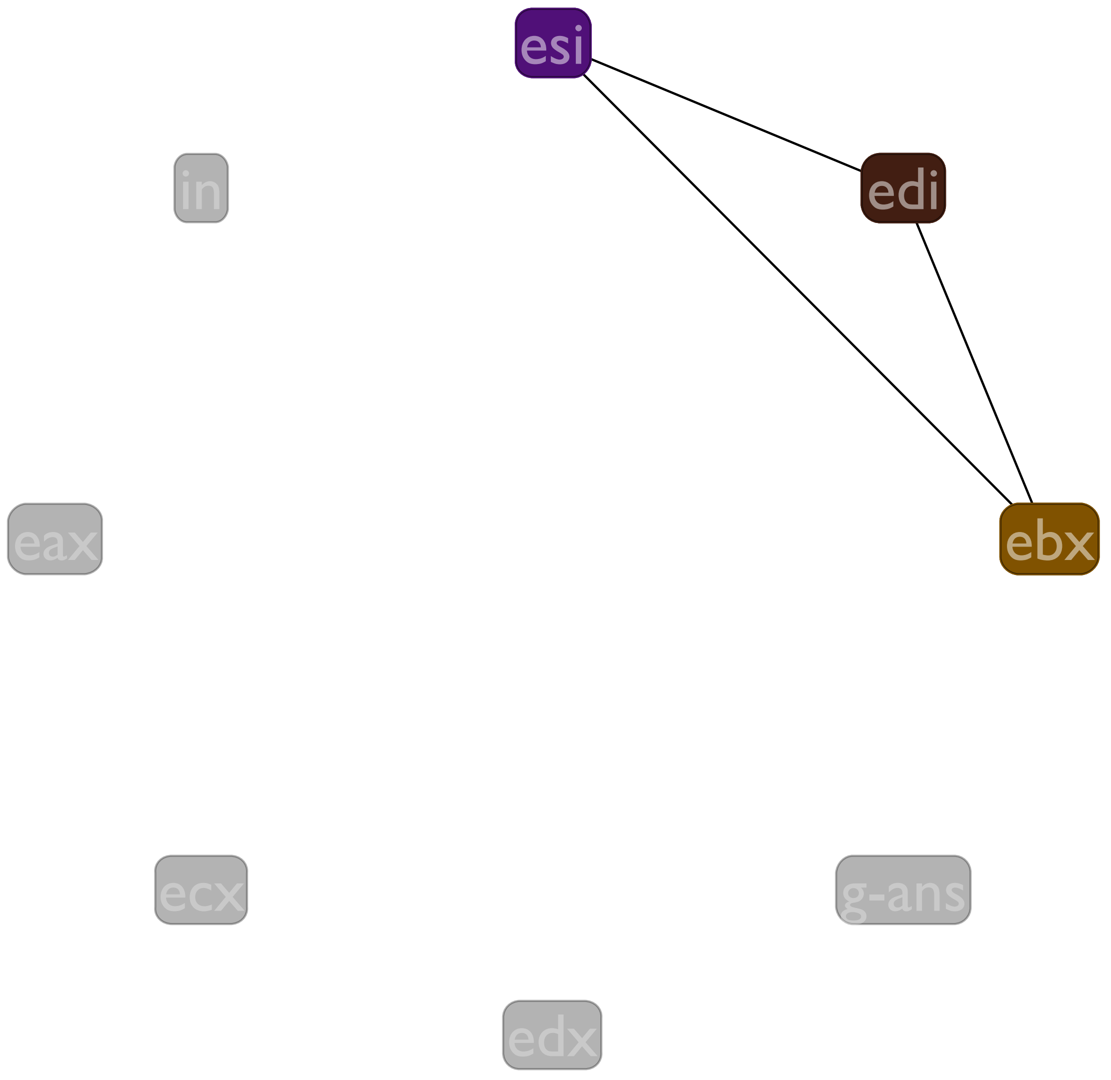
edx

in

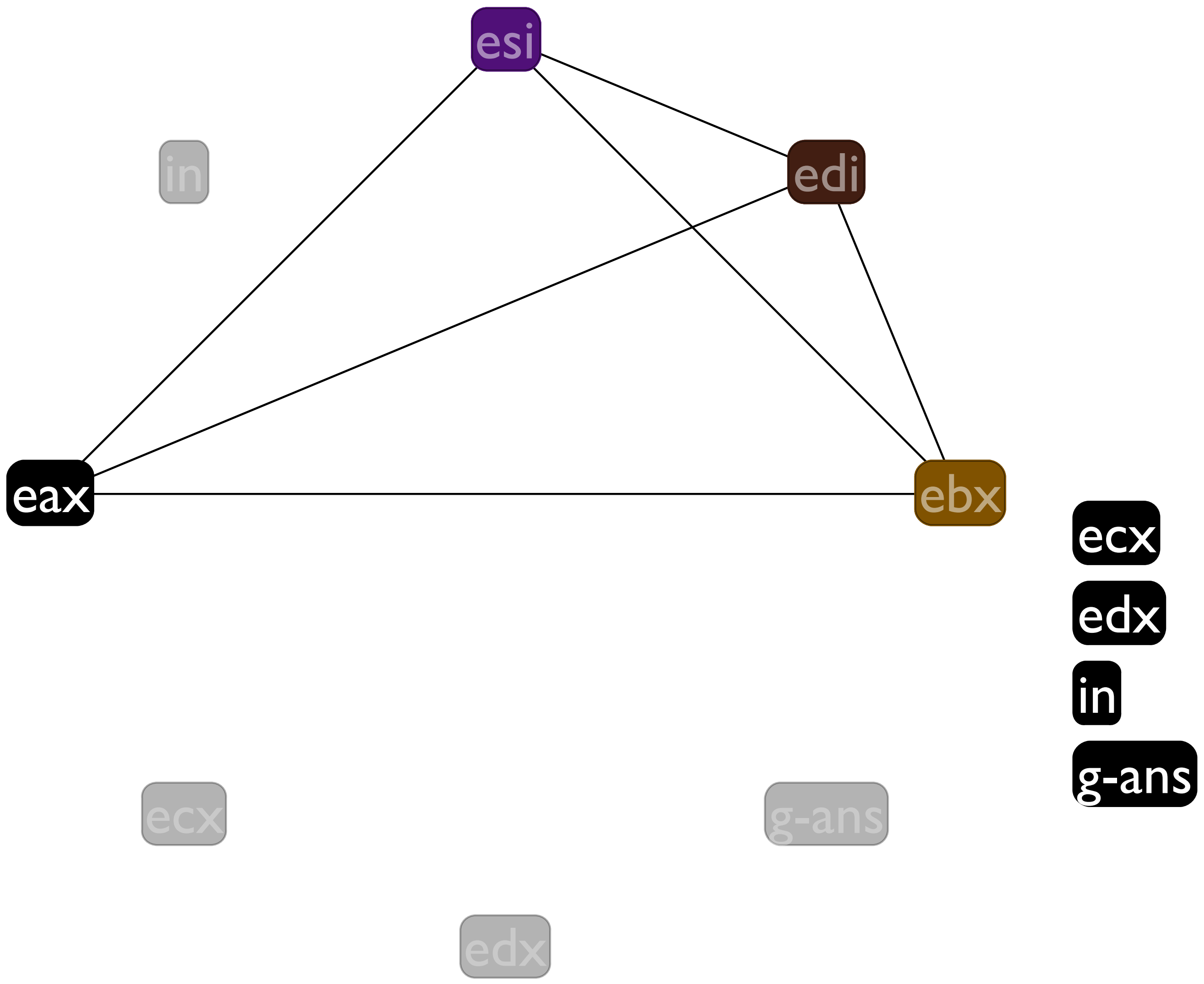
g-ans

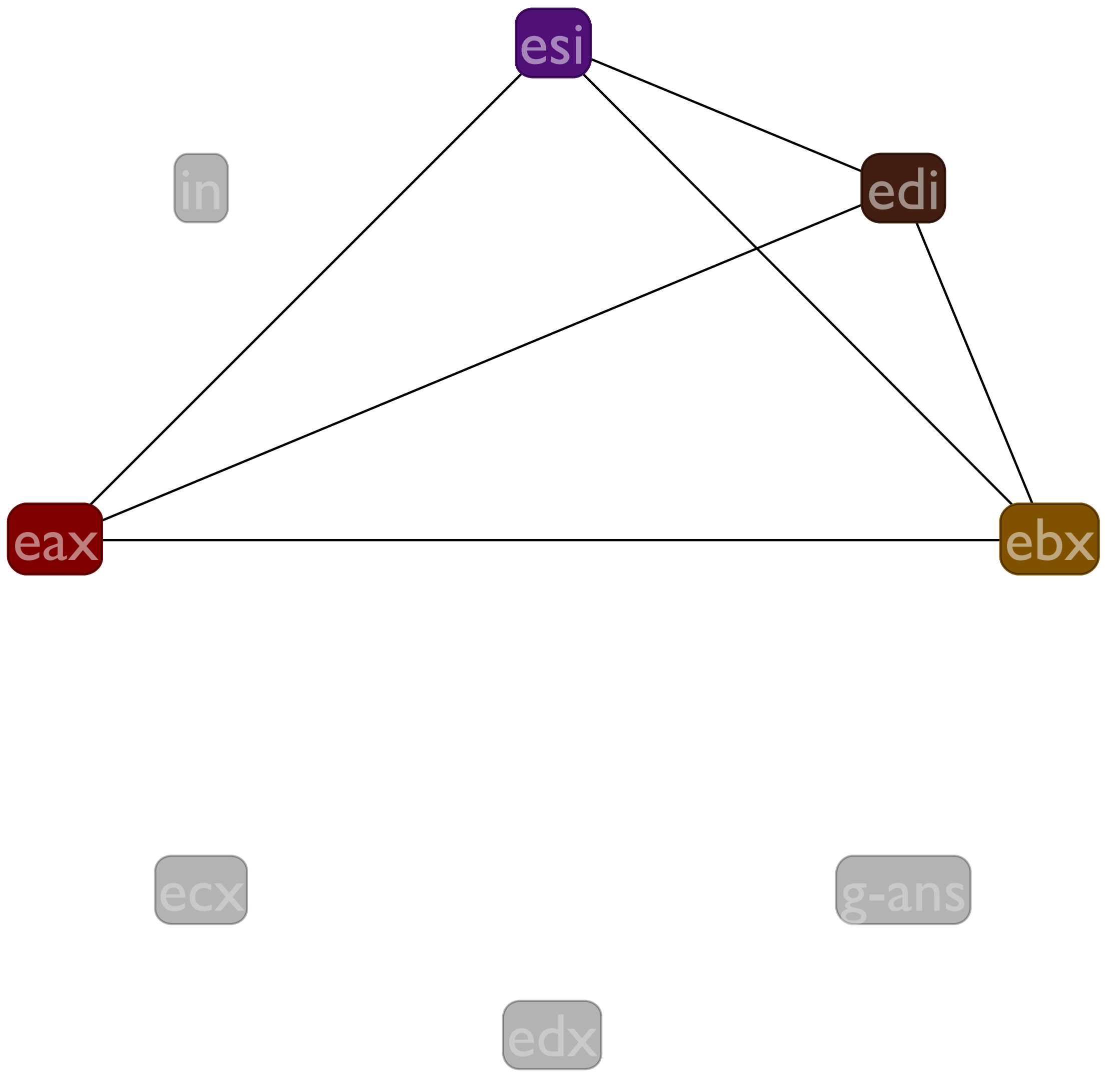

esi
edi
in
eax
ebx
ecx
g-ans
edx


eax
ecx
edx
in
g-ans


esi
edi
in
eax
ebx
ecx
g-ans
edx

eax
ecx
edx
in
g-ans


esi
in
edi
eax
ebx
ecx
edx
in
g-ans
ecx
g-ans
edx

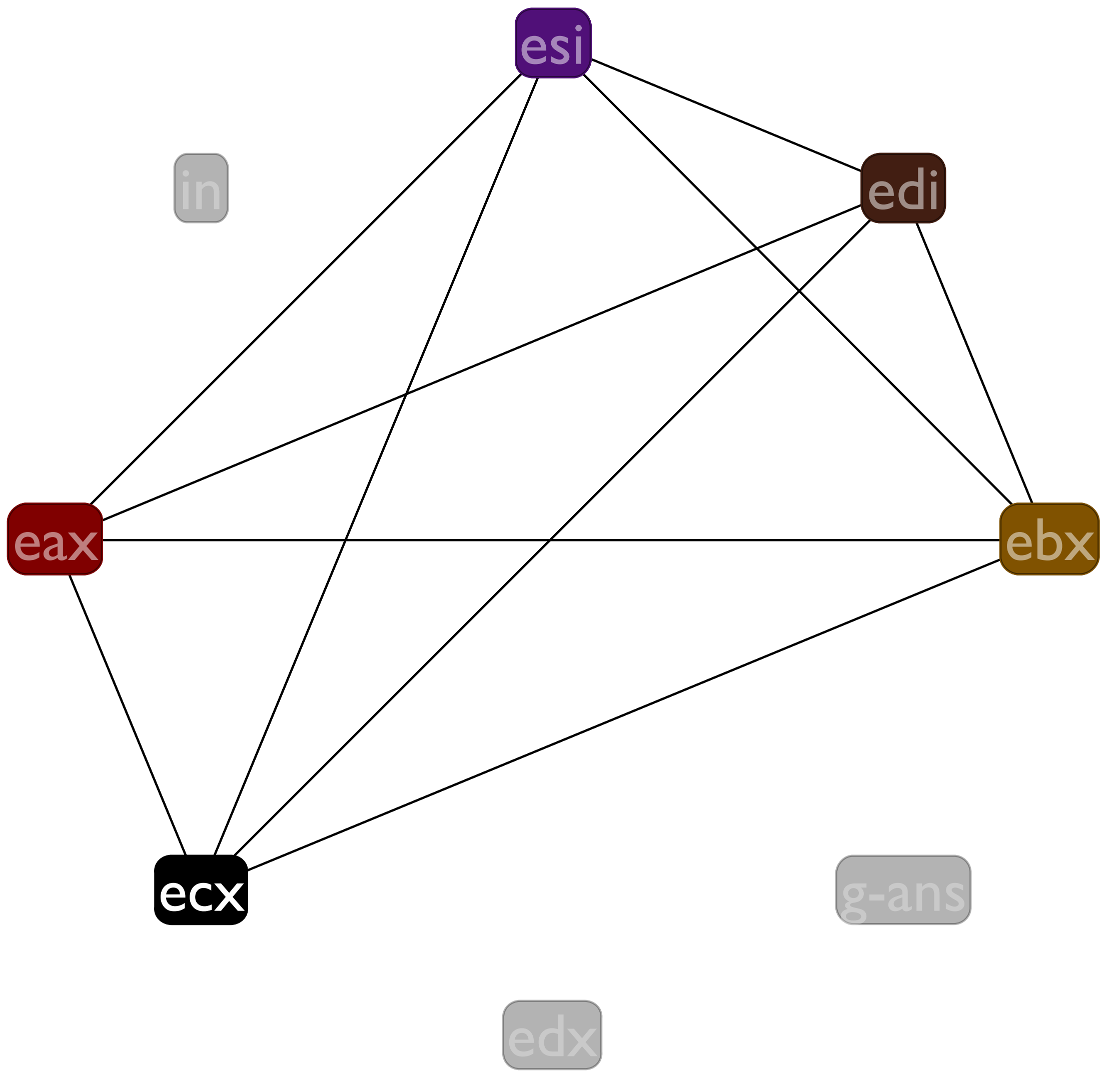


ecx

edx

in

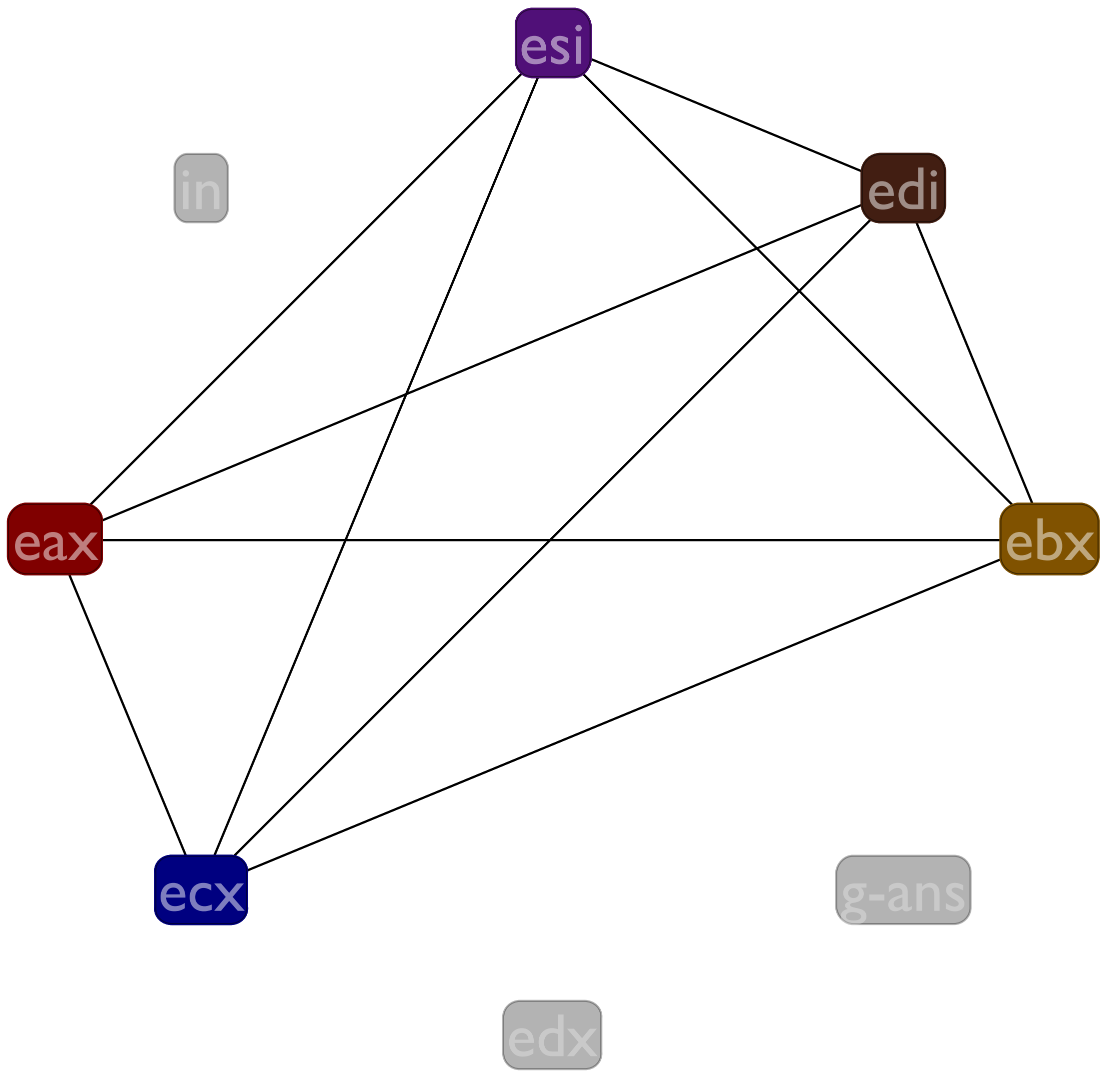
g-ans



edx

in

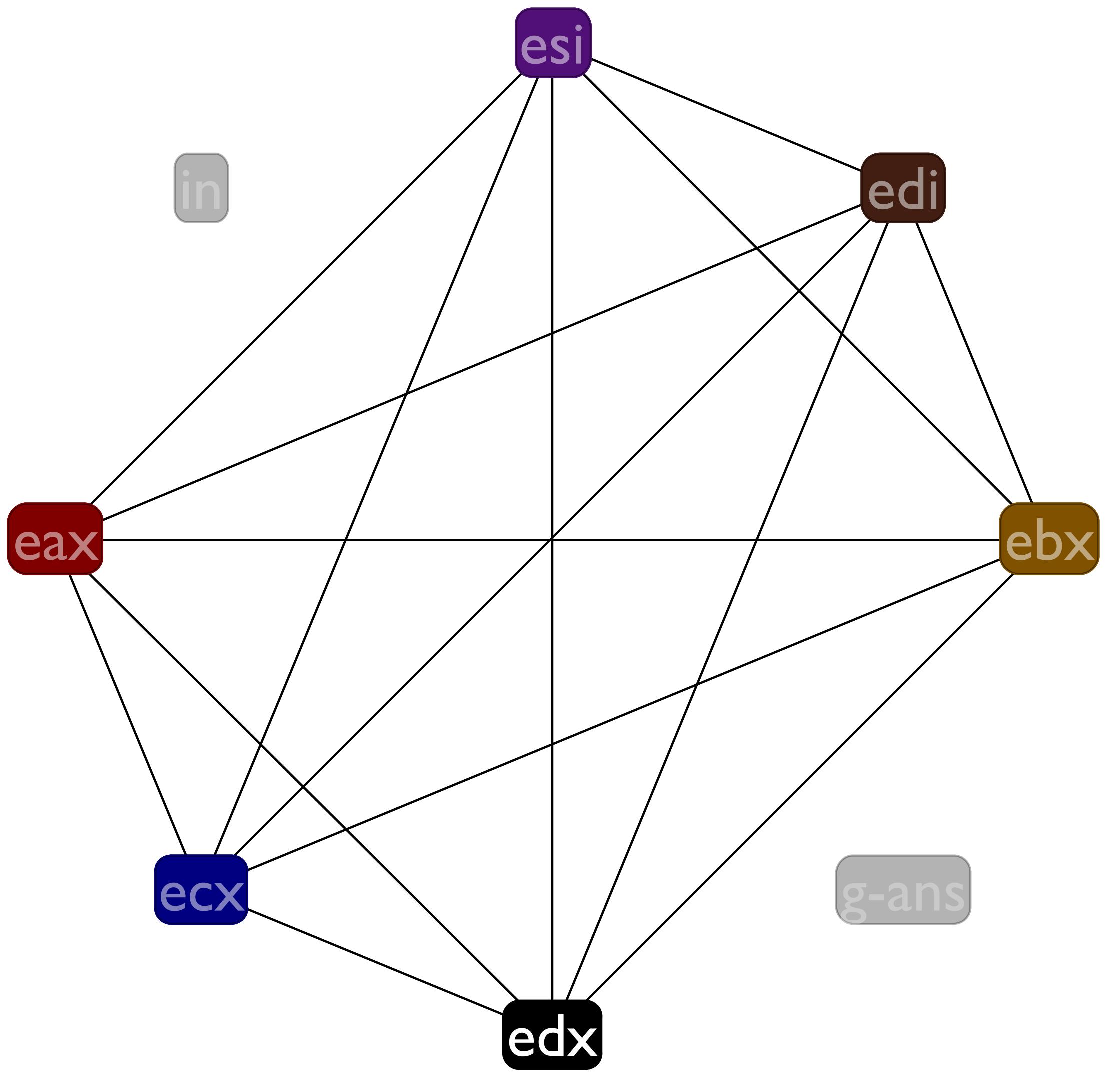
g-ans



edx

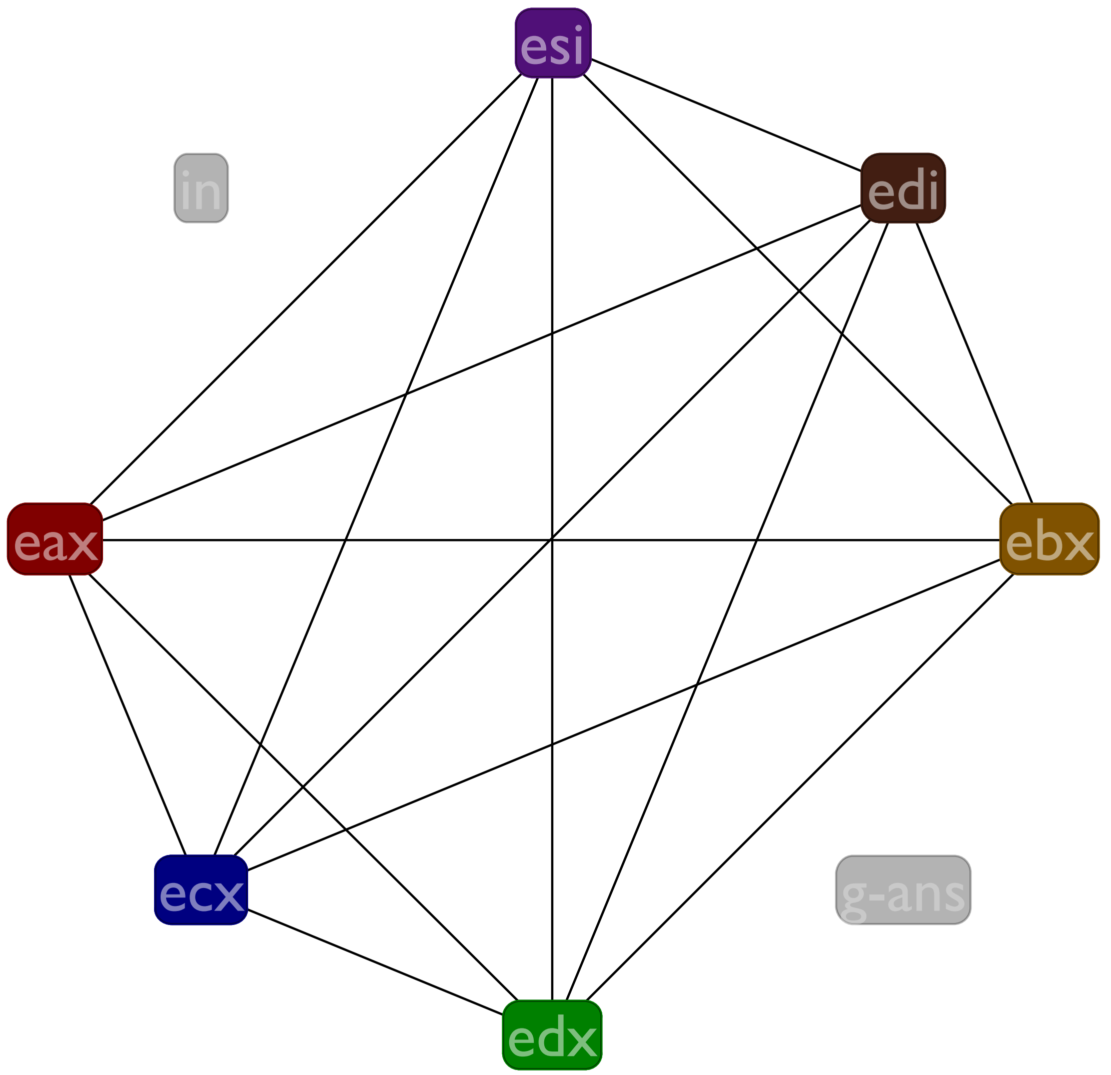
in

g-ans



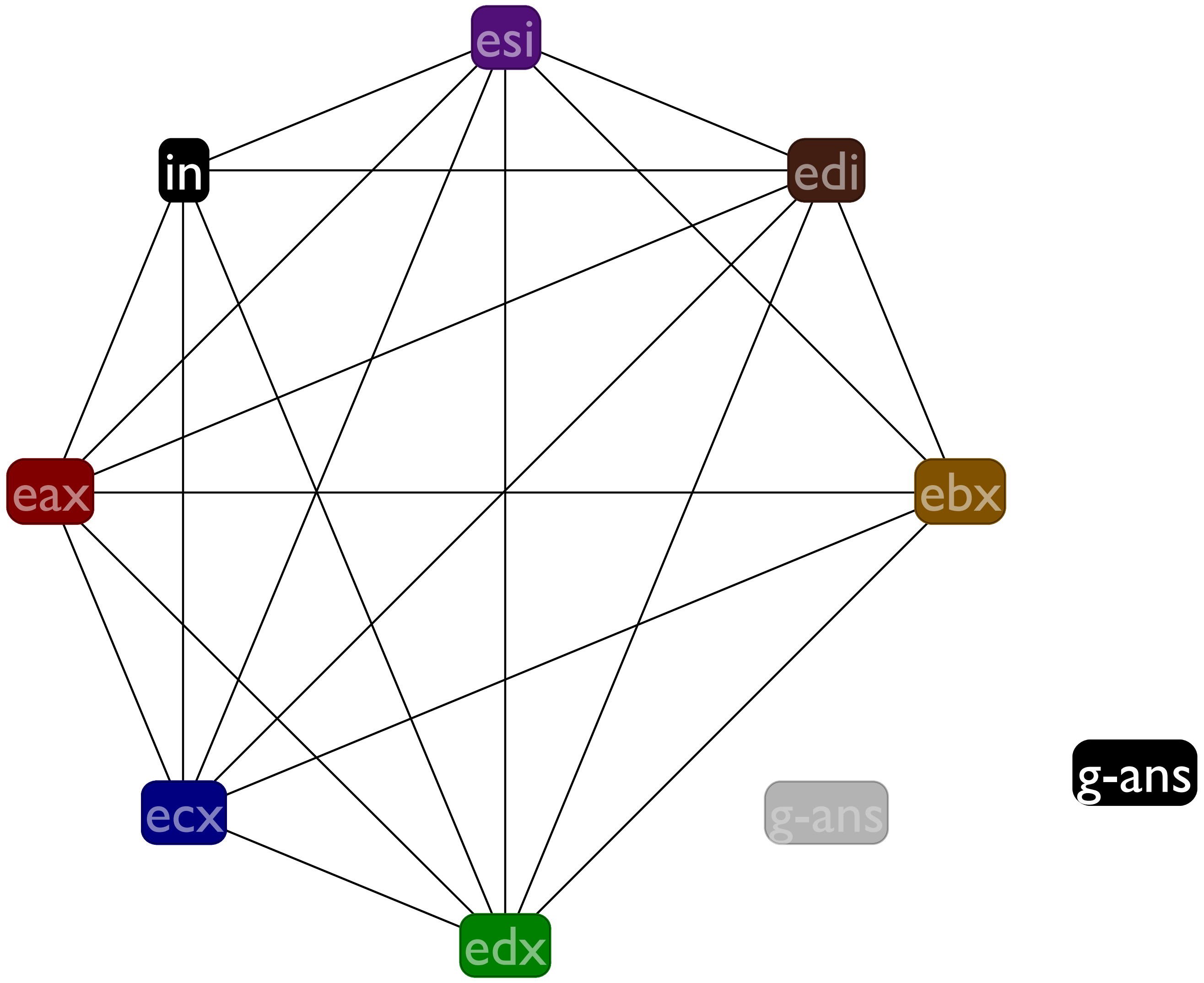
in

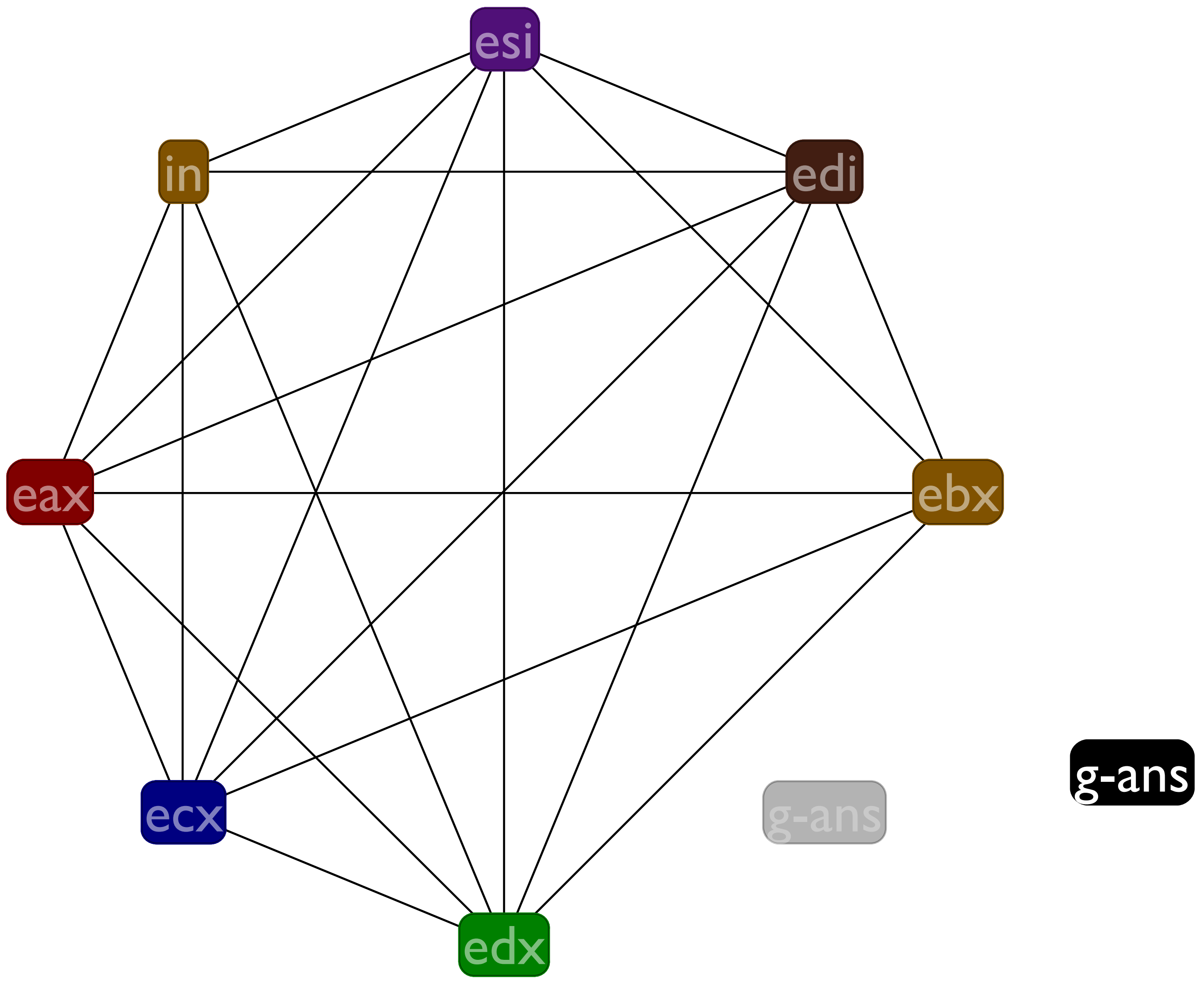
g-ans

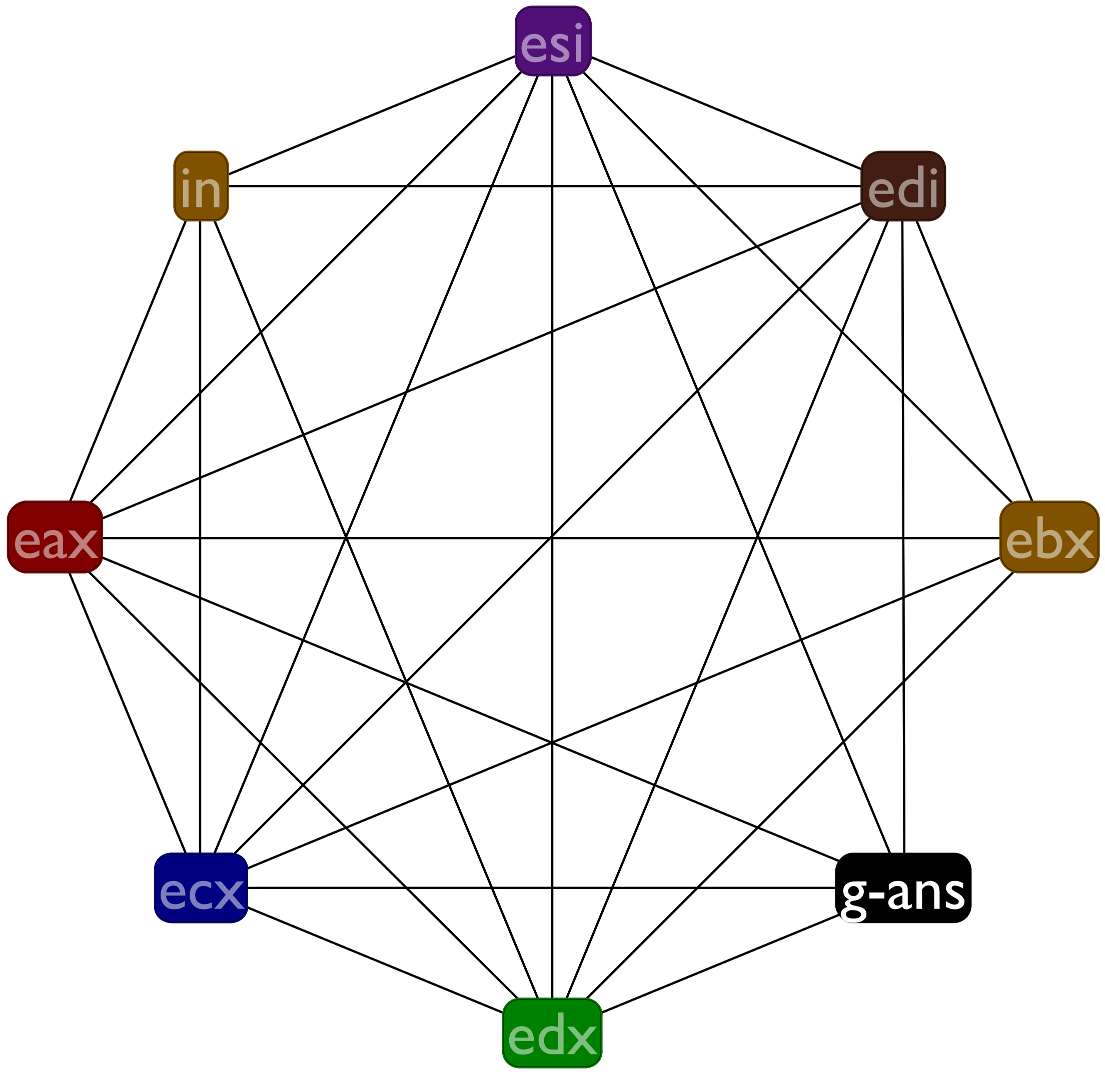


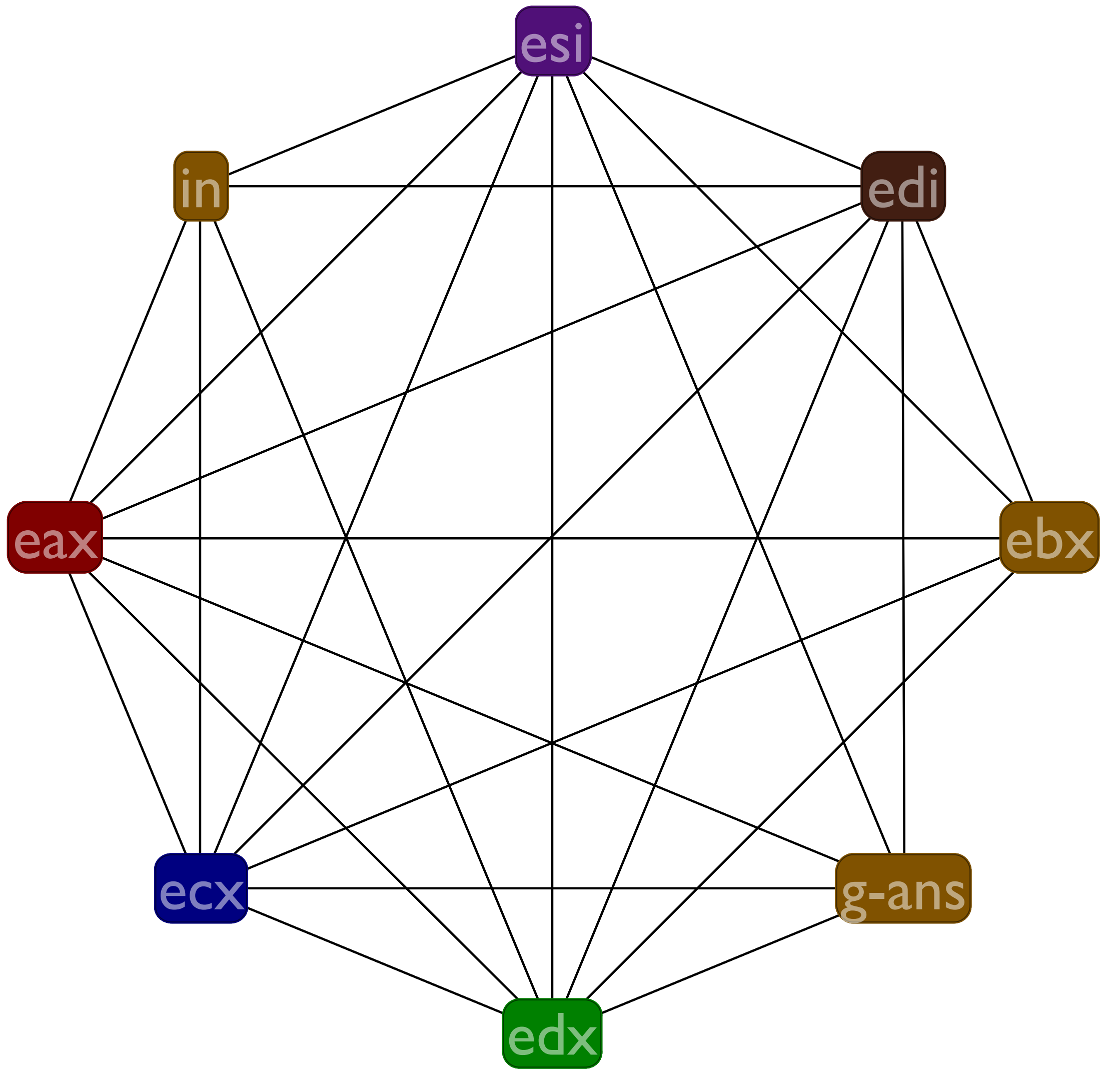
in

g-ans


esi
in
edi
eax
ebx
ecx
edx
g-ans
g-ans


esi
in
edi
eax
ebx
ecx
g-ans
edx
g-ans


esi
in
edi
eax
ebx
ecx
g-ans
edx


esi
in
edi
eax
ebx
ecx
g-ans
edx

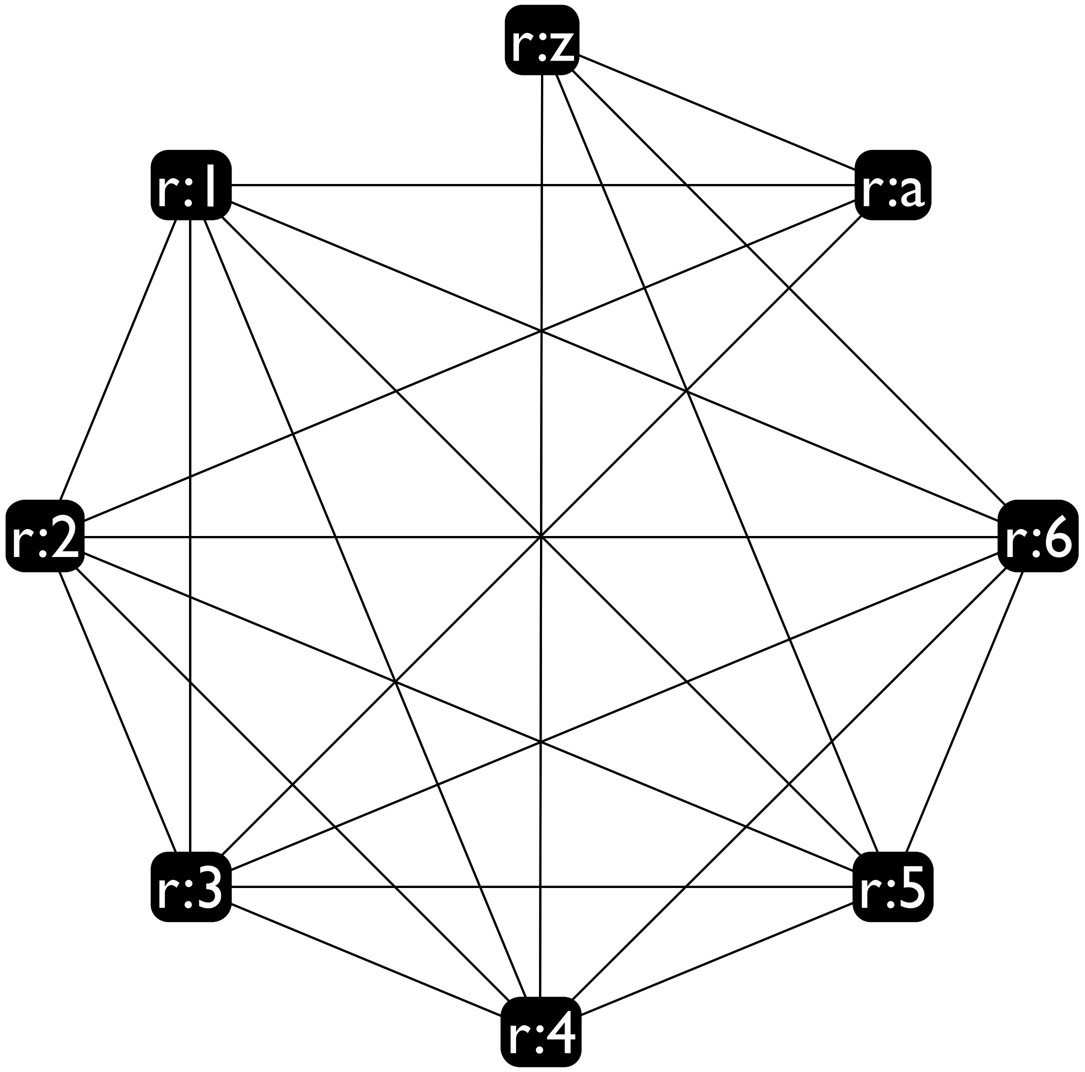
It worked that time, but this doesn’t always work

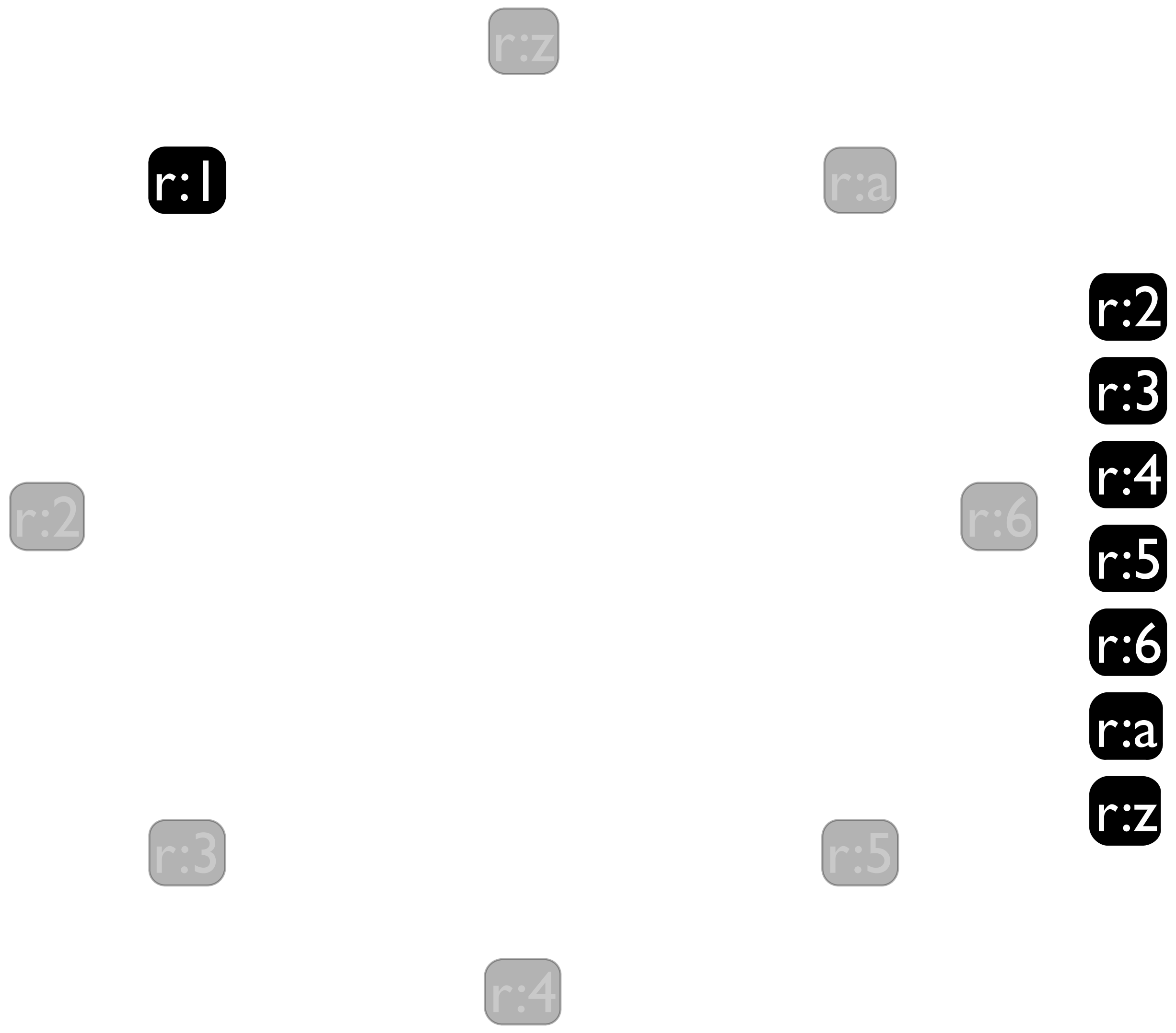
Coalescing Problem

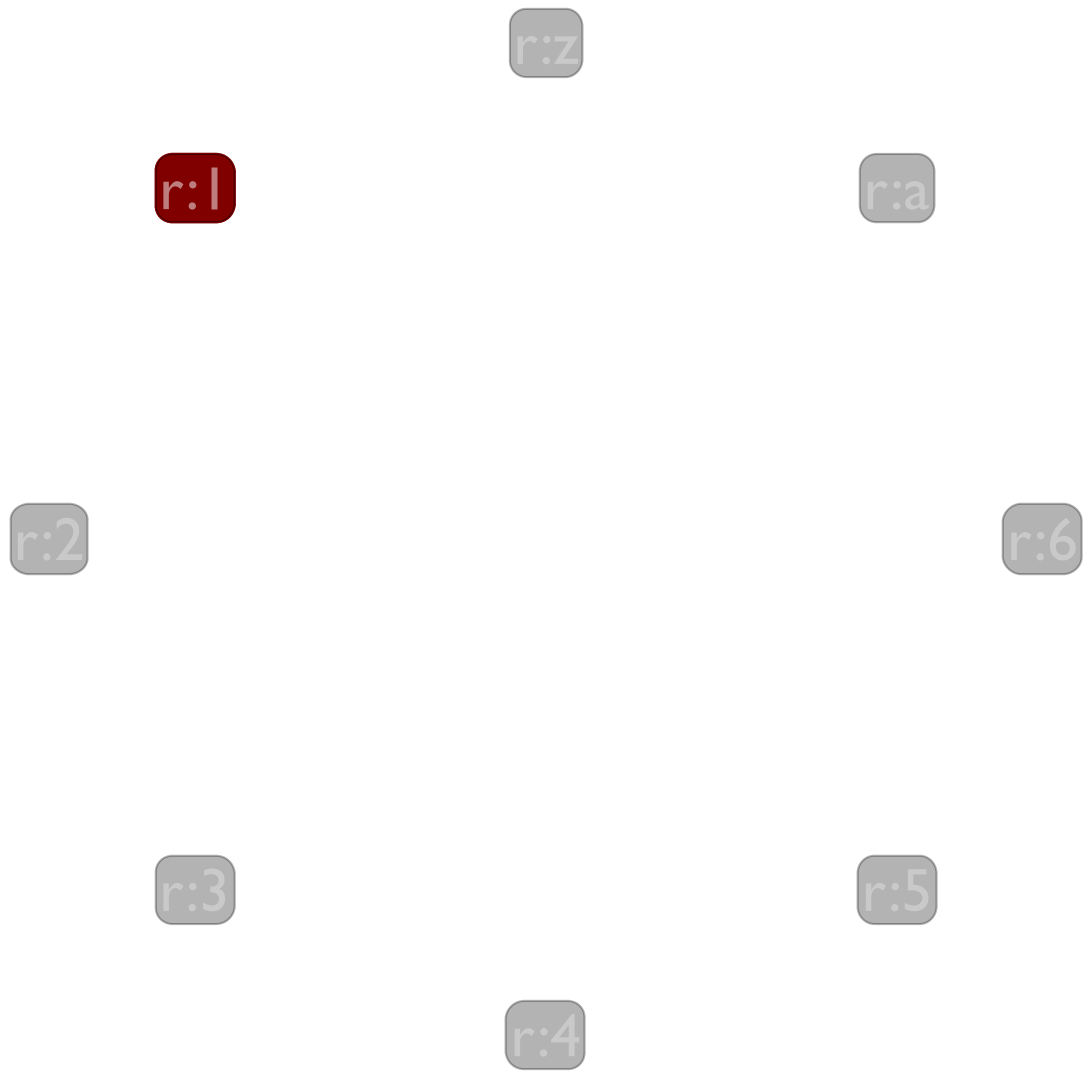
		in	out
1:	`(r:a <- 1)`	()	()
2:	`(r:z <- r:a)`	()	()
3:	`(r:1 <- 1)`	()	()
4:	`(r:2 <- 2)`	()	()
5:	`(r:3 <- 3)`	()	()
6:	`(r:a += 1)`	()	()
7:	`(r:4 <- 4)`	()	()
8:	`(r:5 <- 5)`	()	()
9:	`(r:6 <- 6)`	()	()
10:	`(r:1 += 1)`	()	()
11:	`(r:2 += 1)`	()	()
12:	`(r:3 += 1)`	()	()
13:	`(r:z <- 1)`	()	()
14:	`(r:4 += 1)`	()	()
15:	`(r:5 += 1)`	()	()
16:	`(r:6 += 1)`	()	()
17:	`(r:z += 1)`	()	()

Coalescing Problem

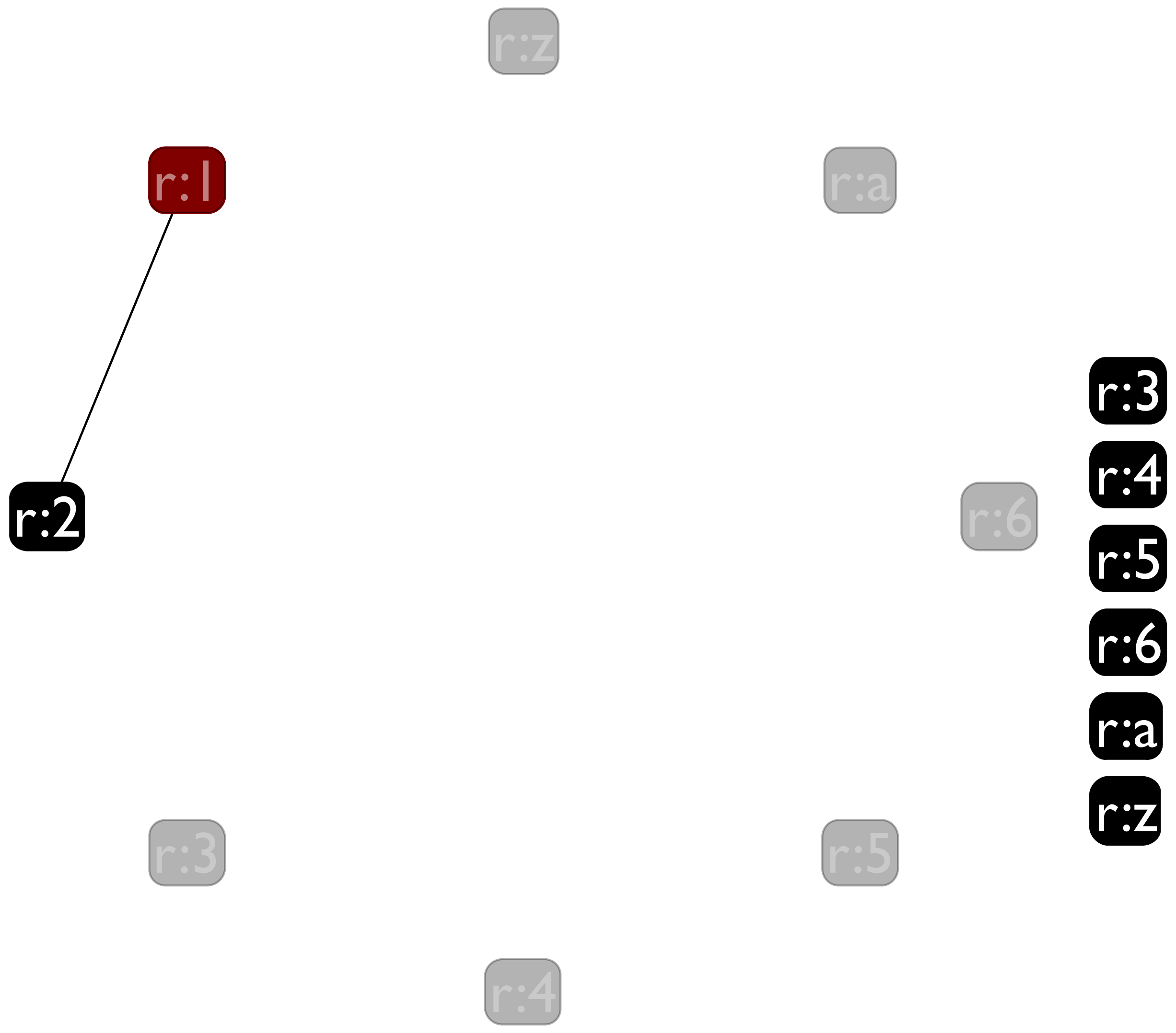
		in	out
1:	`(r:a <- 1)`	()	(r:a)
2:	`(r:z <- r:a)`	(r:a)	(r:a)
3:	`(r:1 <- 1)`	(r:a)	(r:1 r:a)
4:	`(r:2 <- 2)`	(r:1 r:a)	(r:1 r:2 r:a)
5:	`(r:3 <- 3)`	(r:1 r:2 r:a)	(r:1 r:2 r:3 r:a)
6:	`(r:a += 1)`	(r:1 r:2 r:3 r:a)	(r:1 r:2 r:3)
7:	`(r:4 <- 4)`	(r:1 r:2 r:3)	(r:1 r:2 r:3 r:4)
8:	`(r:5 <- 5)`	(r:1 r:2 r:3 r:4)	(r:1 r:2 r:3 r:4 r:5)
9:	`(r:6 <- 6)`	(r:1 r:2 r:3 r:4 r:5)	(r:1 r:2 r:3 r:4 r:5 r:6)
10:	`(r:1 += 1)`	(r:1 r:2 r:3 r:4 r:5 r:6)	(r:2 r:3 r:4 r:5 r:6)
11:	`(r:2 += 1)`	(r:2 r:3 r:4 r:5 r:6)	(r:3 r:4 r:5 r:6)
12:	`(r:3 += 1)`	(r:3 r:4 r:5 r:6)	(r:4 r:5 r:6)
13:	`(r:z <- 1)`	(r:4 r:5 r:6)	(r:4 r:5 r:6 r:z)
14:	`(r:4 += 1)`	(r:4 r:5 r:6 r:z)	(r:5 r:6 r:z)
15:	`(r:5 += 1)`	(r:5 r:6 r:z)	(r:6 r:z)
16:	`(r:6 += 1)`	(r:6 r:z)	(r:z)
17:	`(r:z += 1)`	(r:z)	()

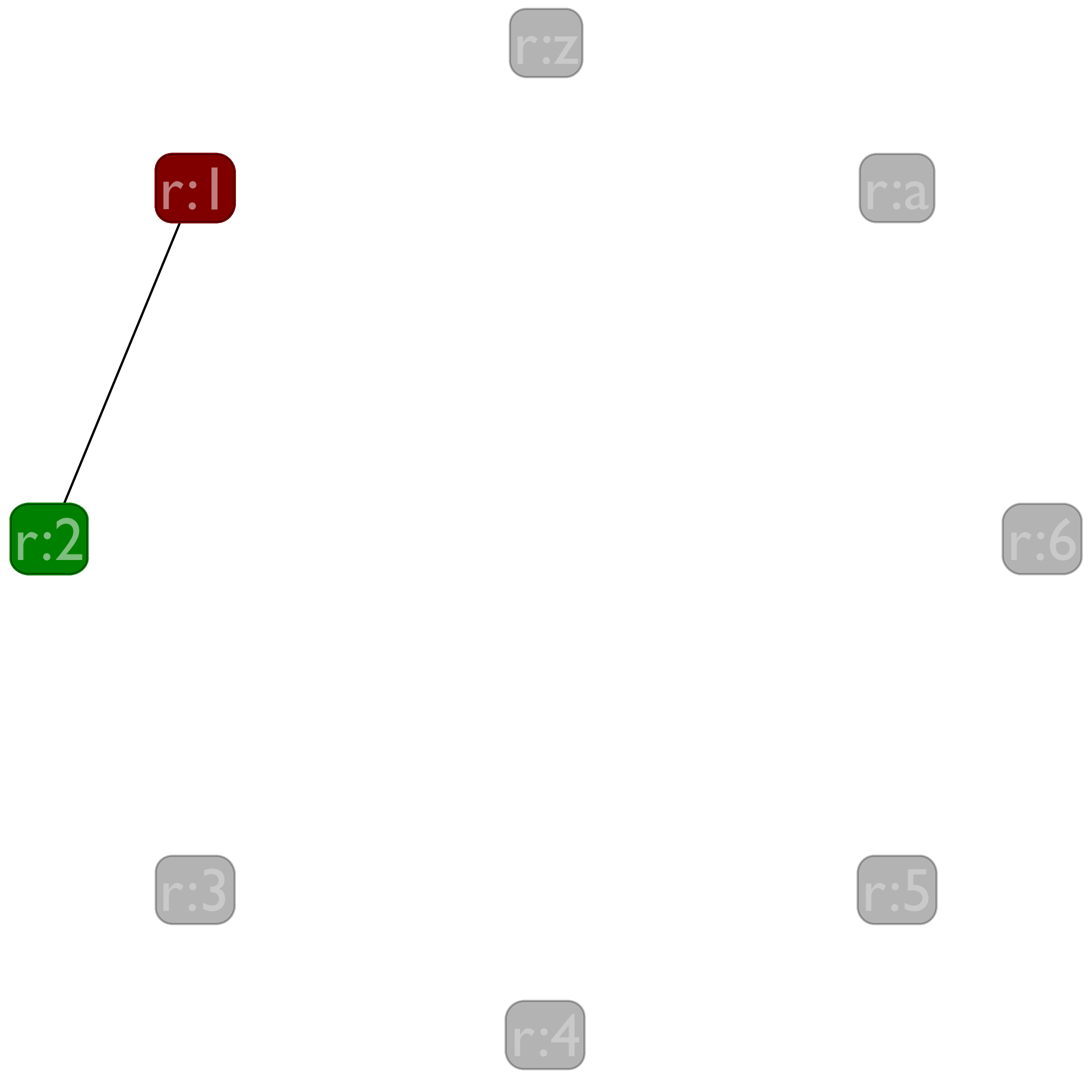

r:z
r:1
r:a
r:2
r:6
r:3
r:5
r:4


r:z
r:1
r:a
r:2
r:3
r:4
r:5
r:6
r:a
r:z
r:2
r:6
r:3
r:5
r:4



r:2
r:3
r:4
r:5
r:6
r:a
r:z


r:z
r:1
r:a
r:2
r:6
r:3
r:5
r:4
r:3
r:4
r:5
r:6
r:a
r:z



r:3

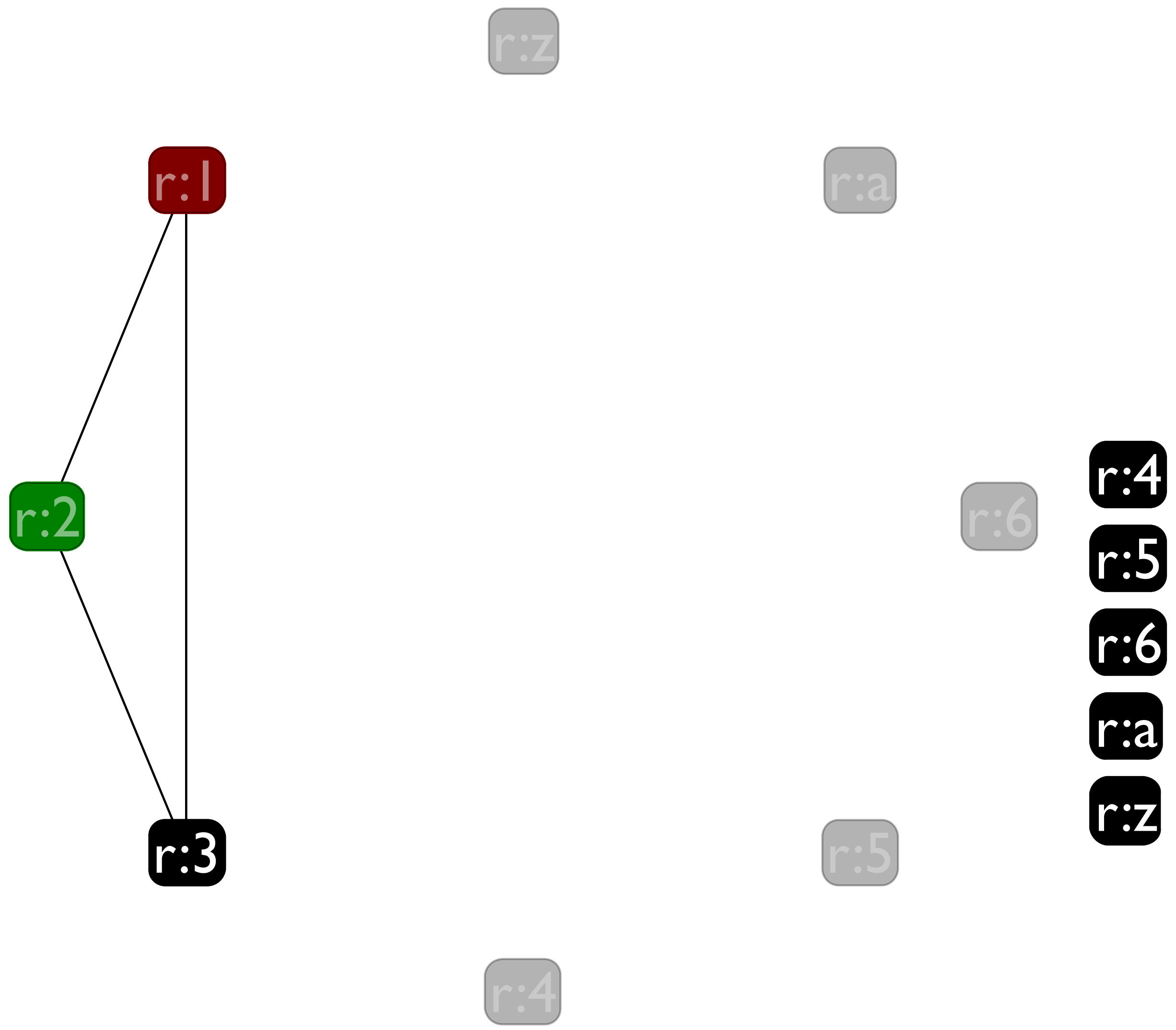
r:4

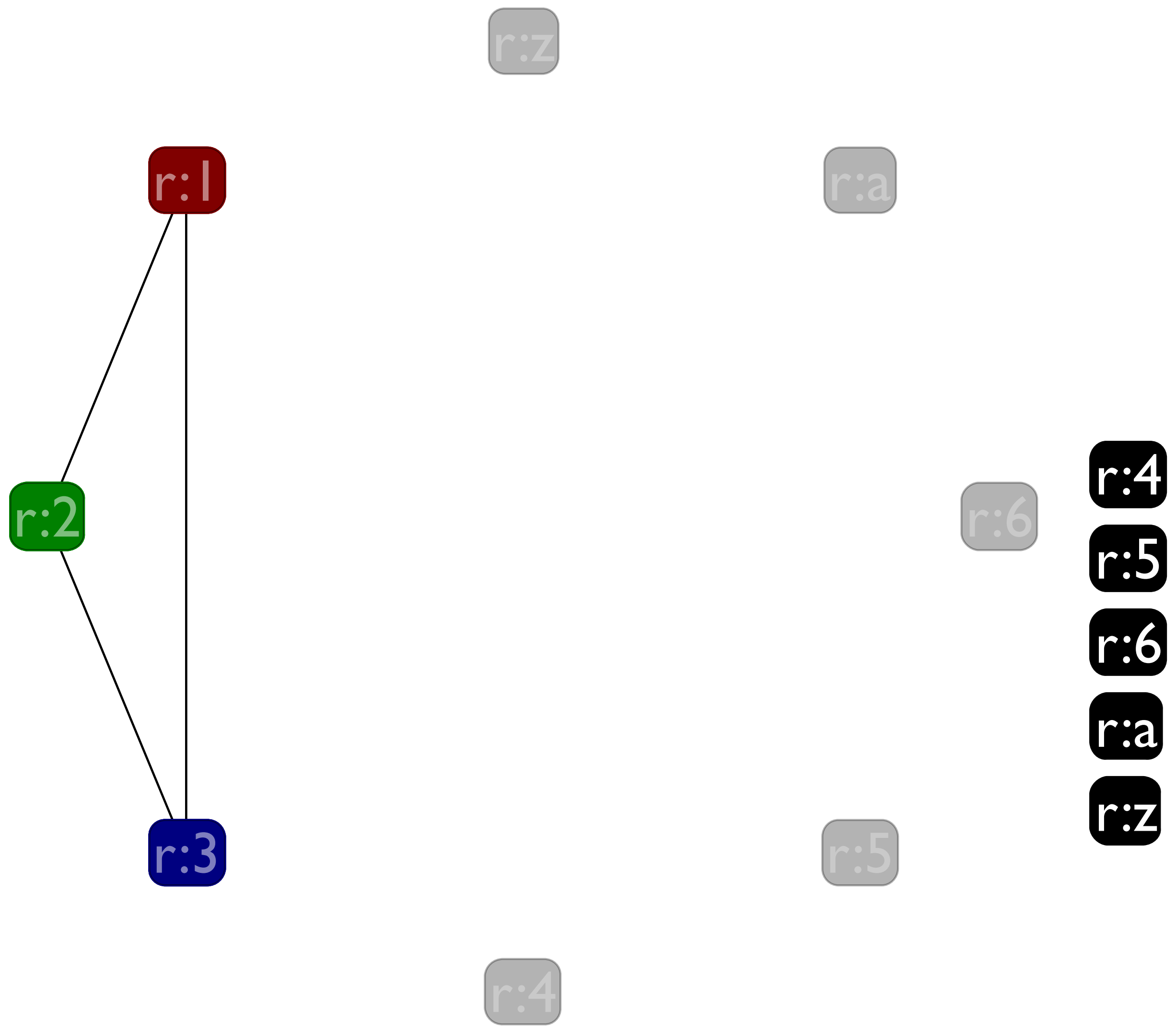
r:5

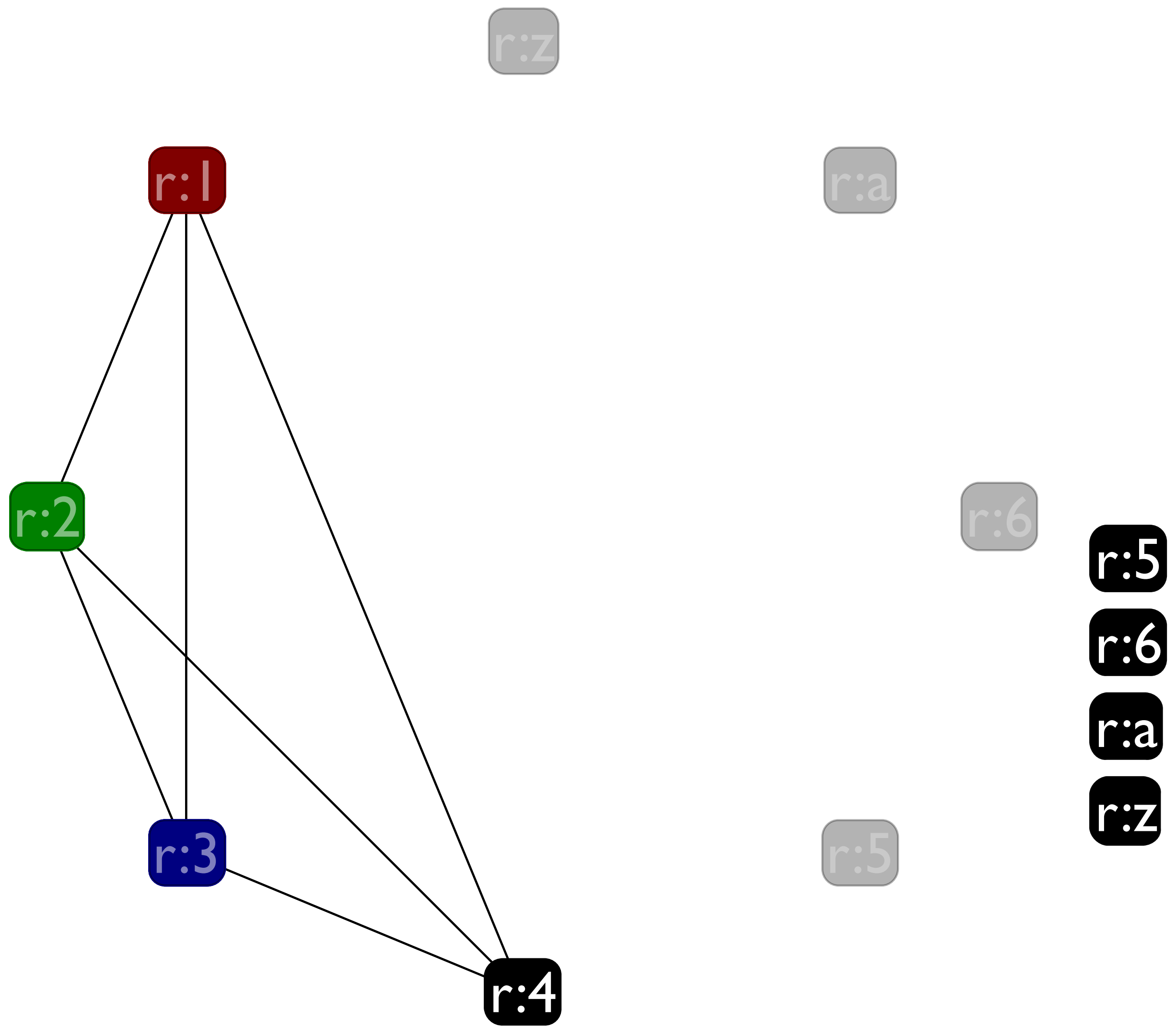
r:6

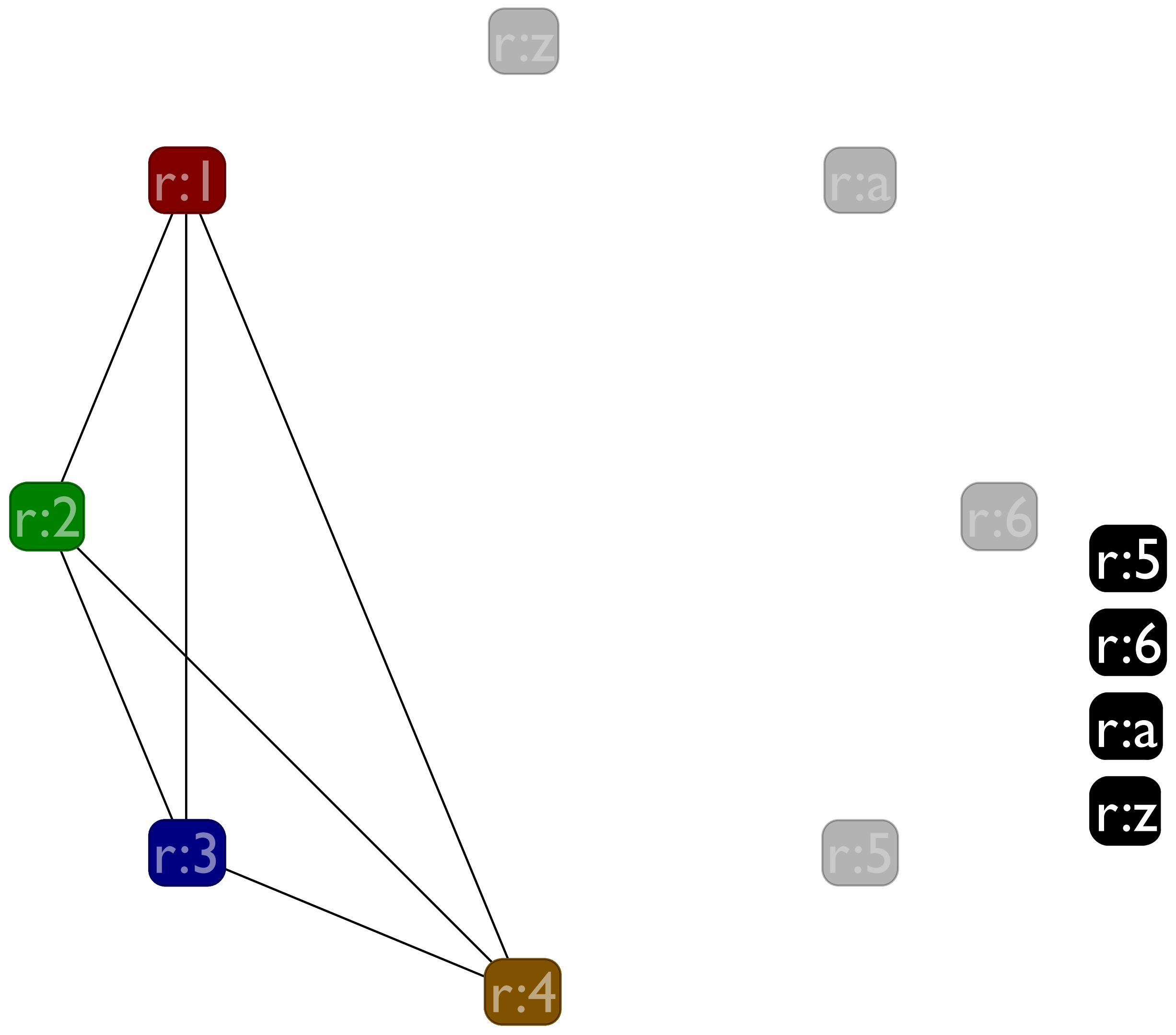
r:a

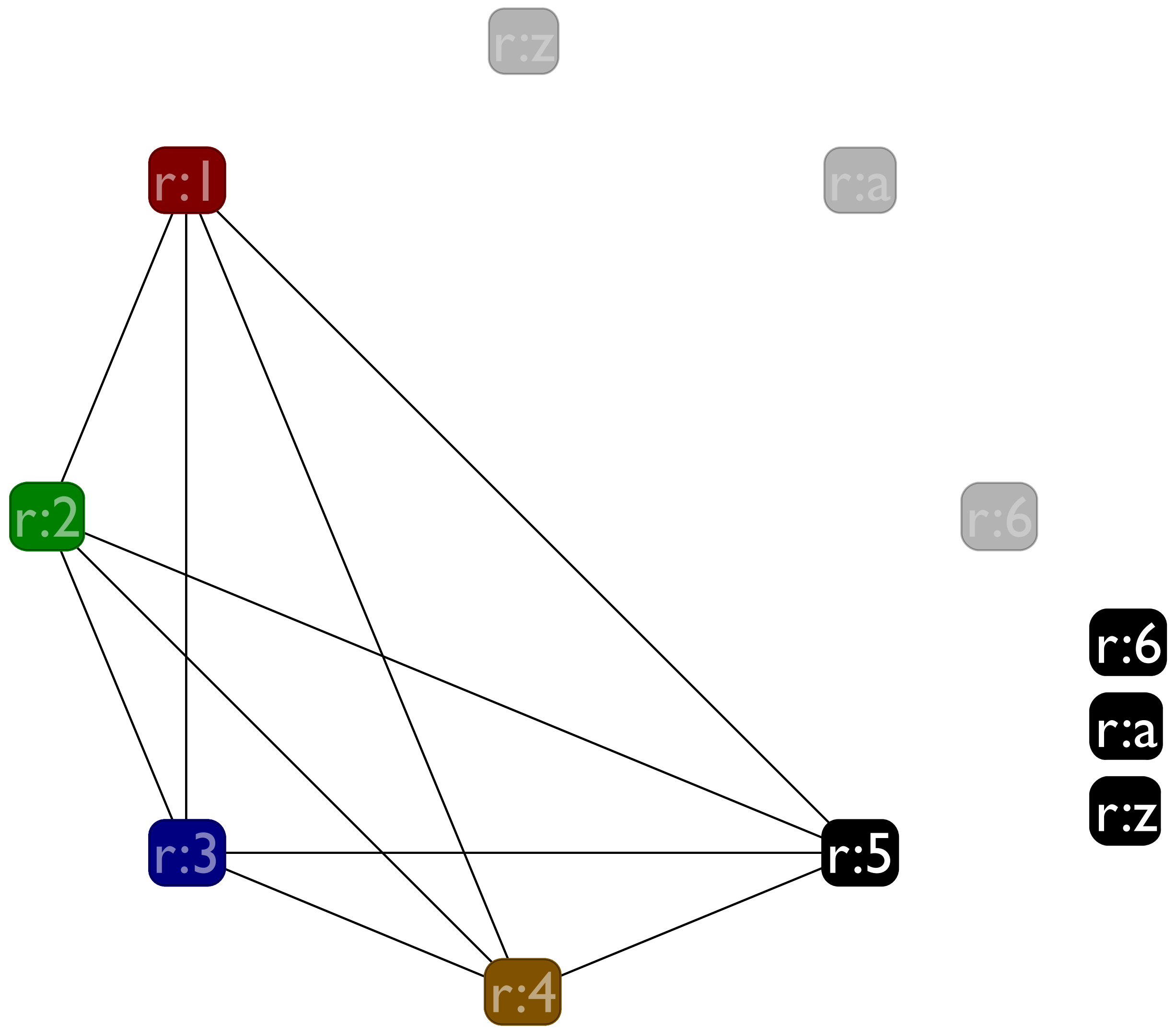
r:z

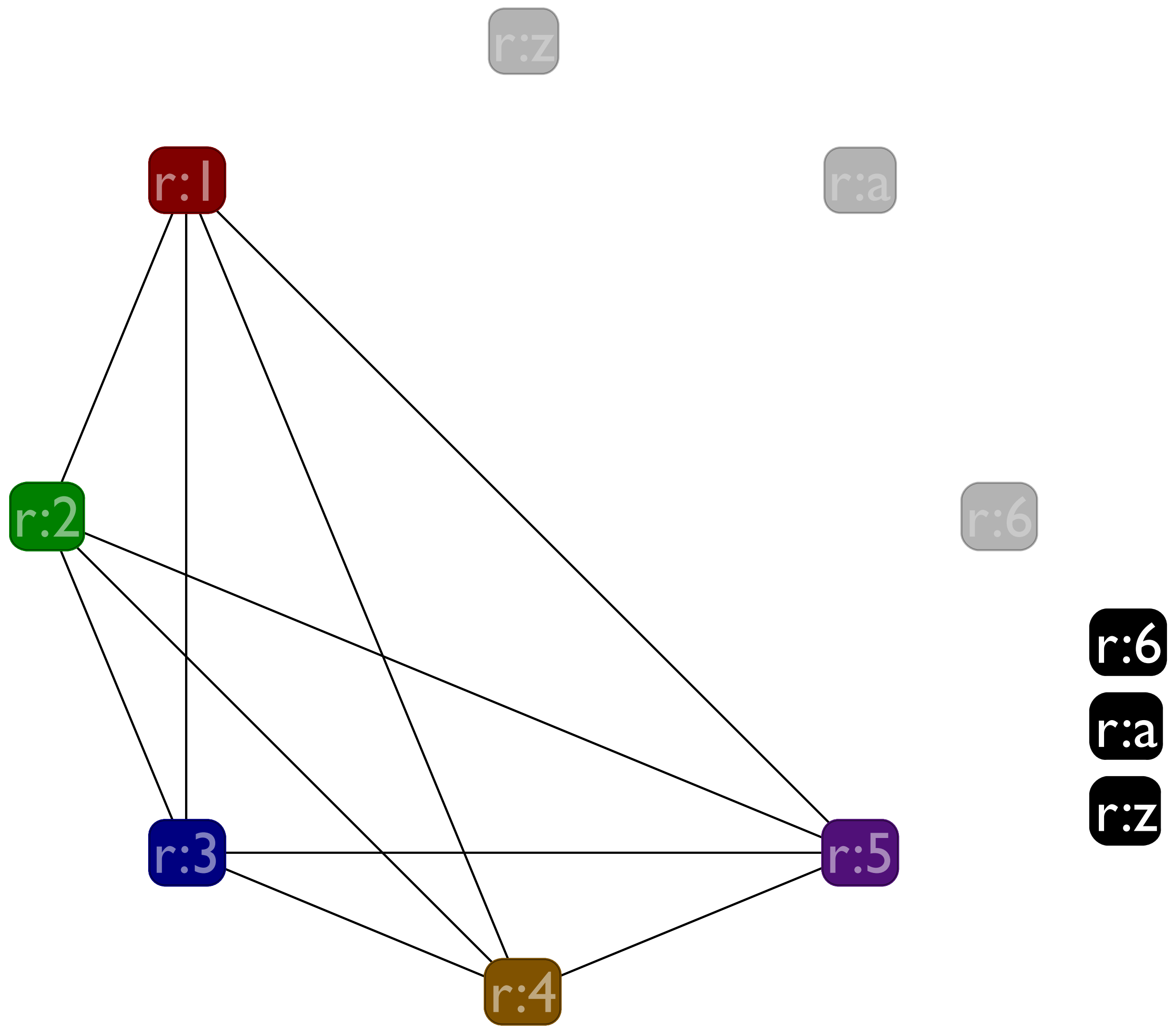

r:z
r:1
r:a
r:2
r:6
r:4
r:5
r:6
r:a
r:z
r:3
r:5
r:4

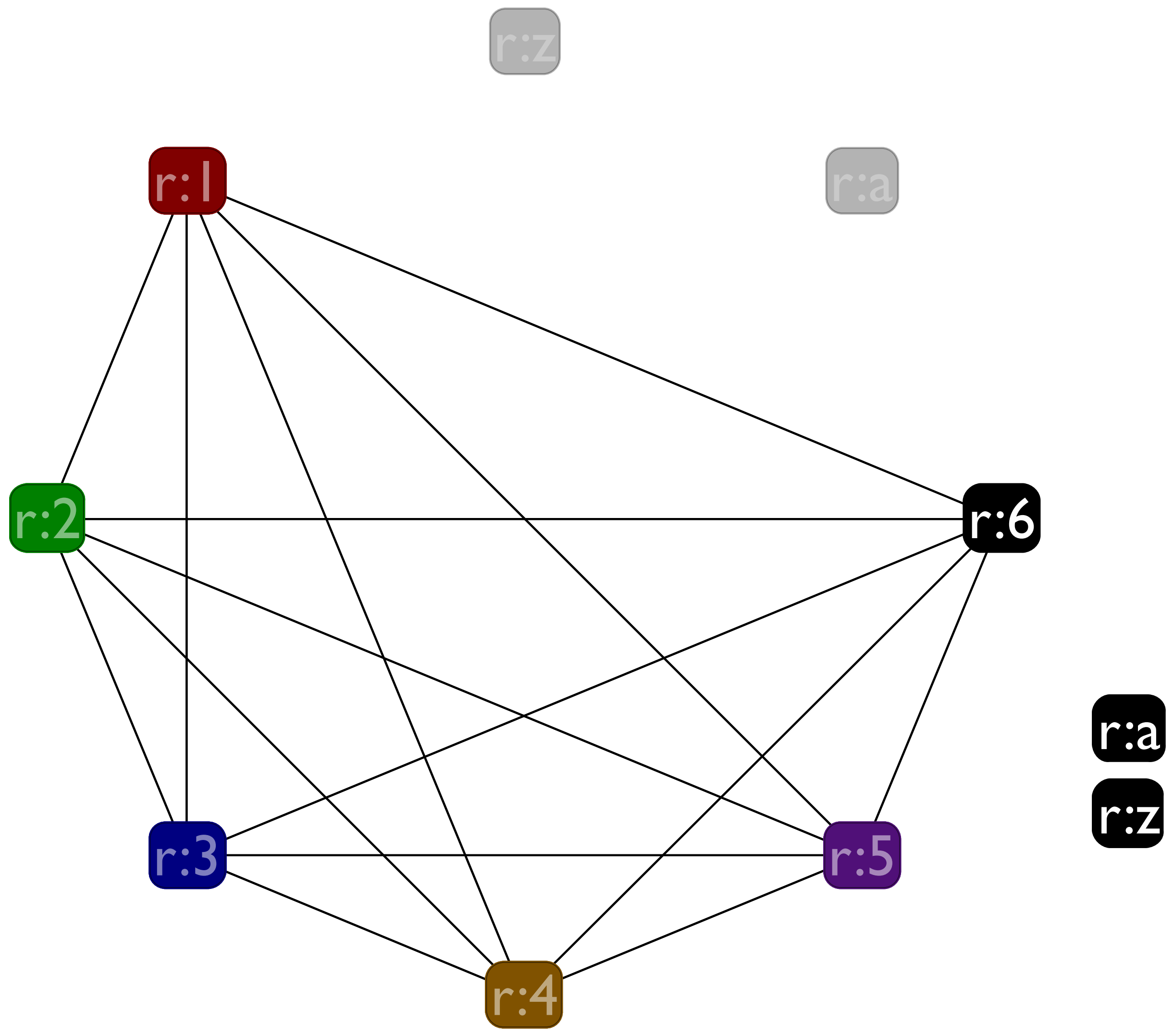

r:z
r:1
r:a
r:2
r:6
r:4
r:5
r:6
r:a
r:z
r:3
r:5
r:4

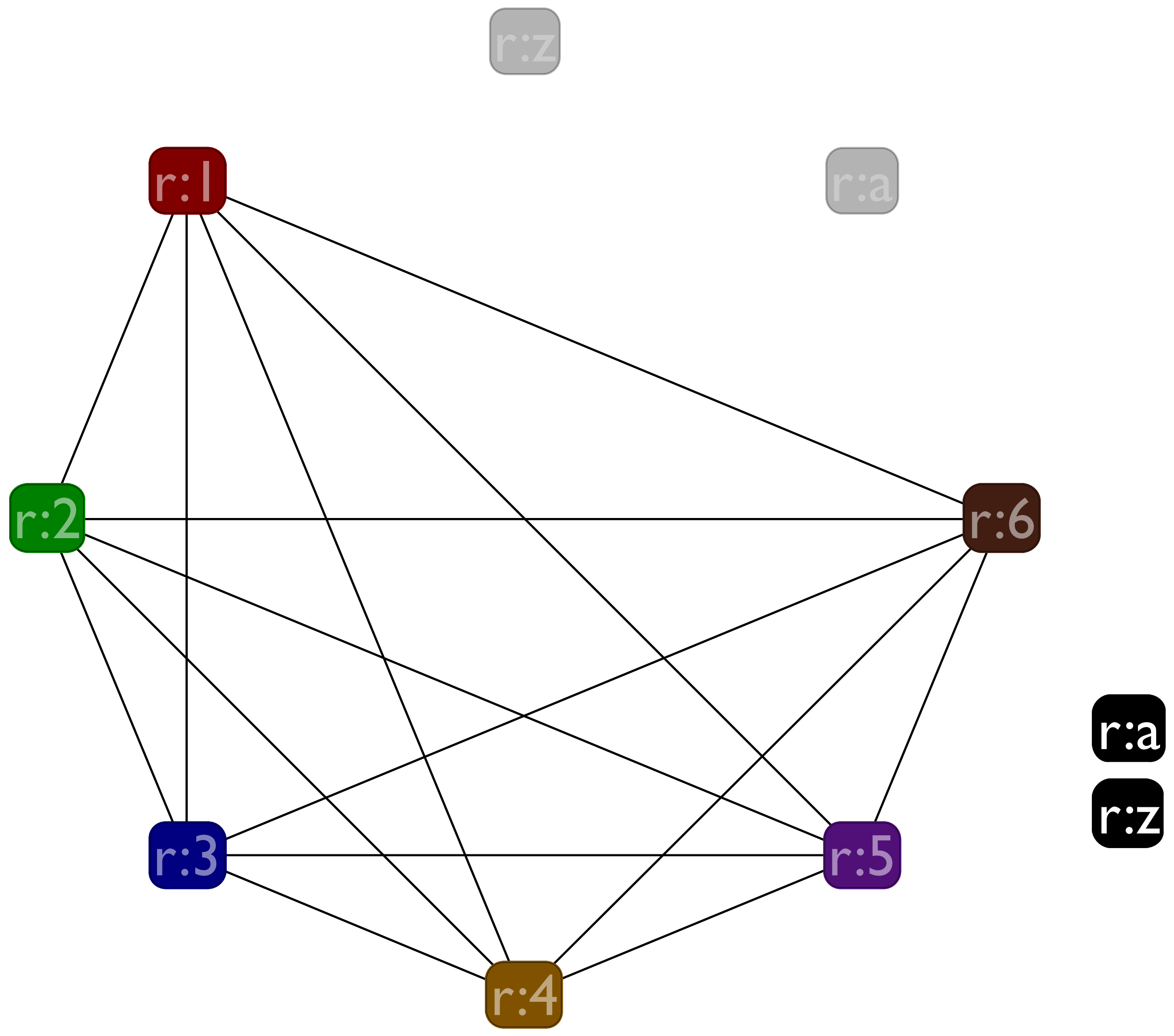

r:z
r:1
r:a
r:2
r:6
r:5
r:6
r:a
r:z
r:3
r:5
r:4

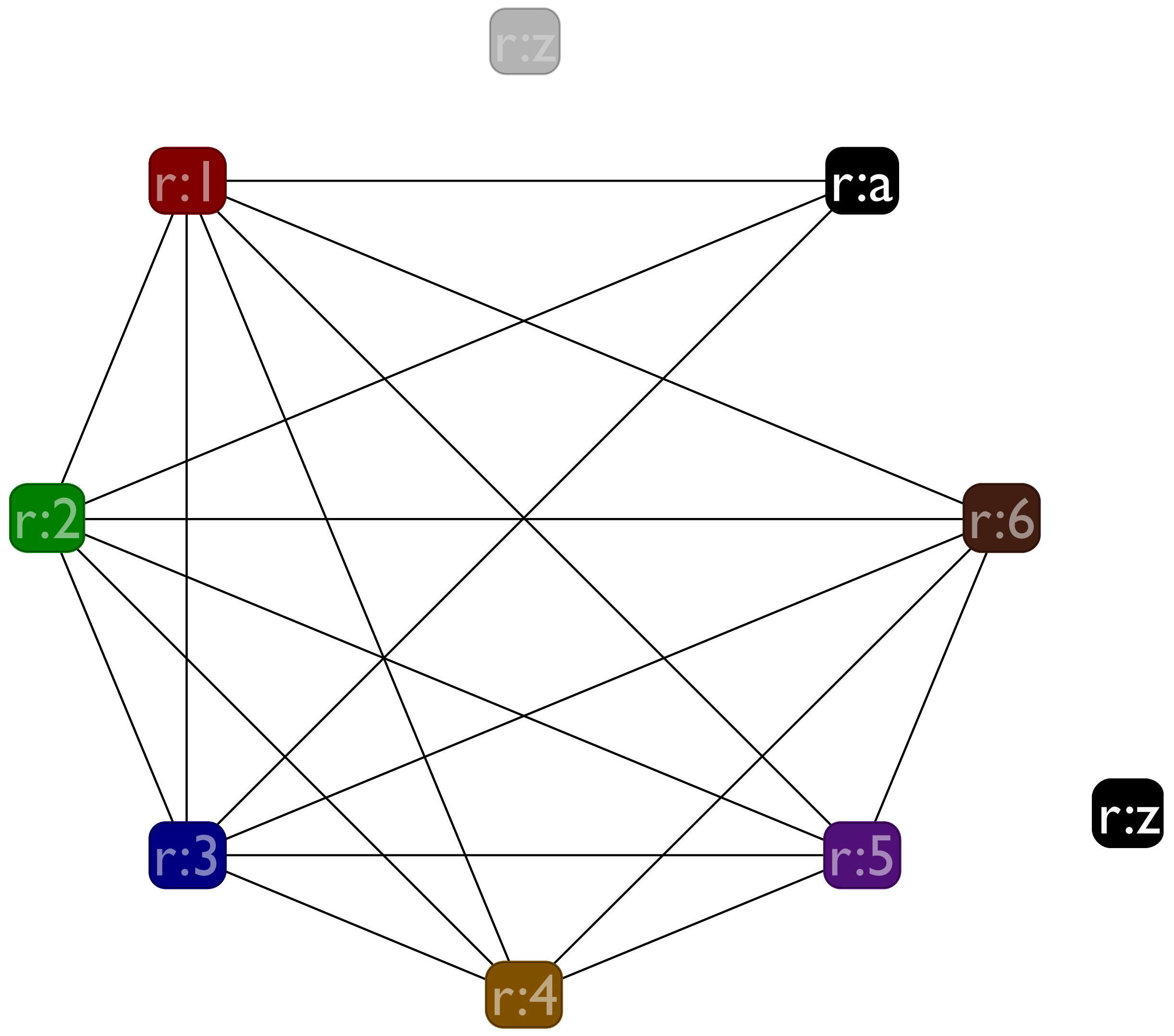

r:z
r:1
r:a
r:2
r:6
r:5
r:6
r:a
r:z
r:3
r:5
r:4

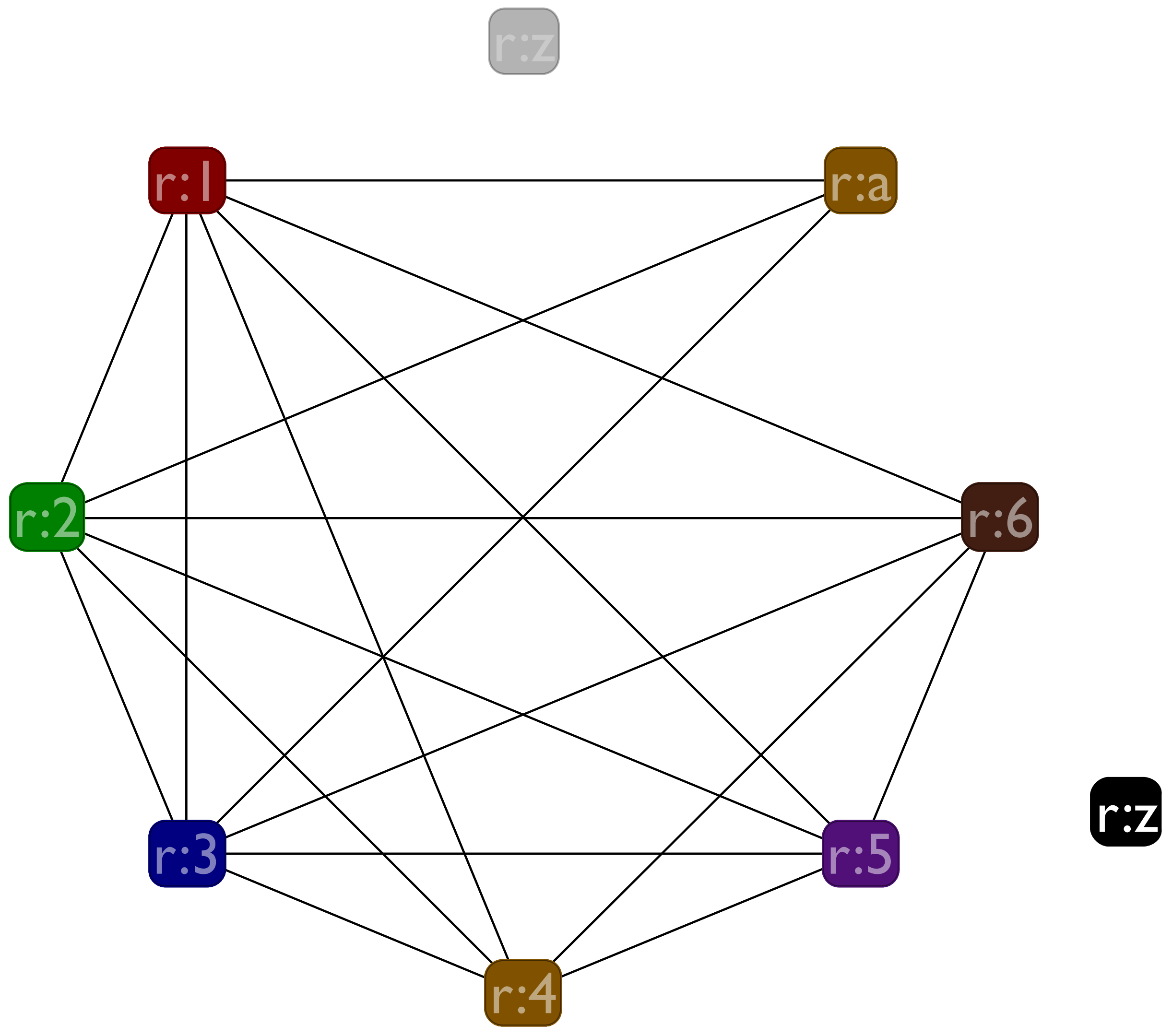

r:z
r:1
r:a
r:2
r:6
r:6
r:a
r:z
r:3
r:5
r:4

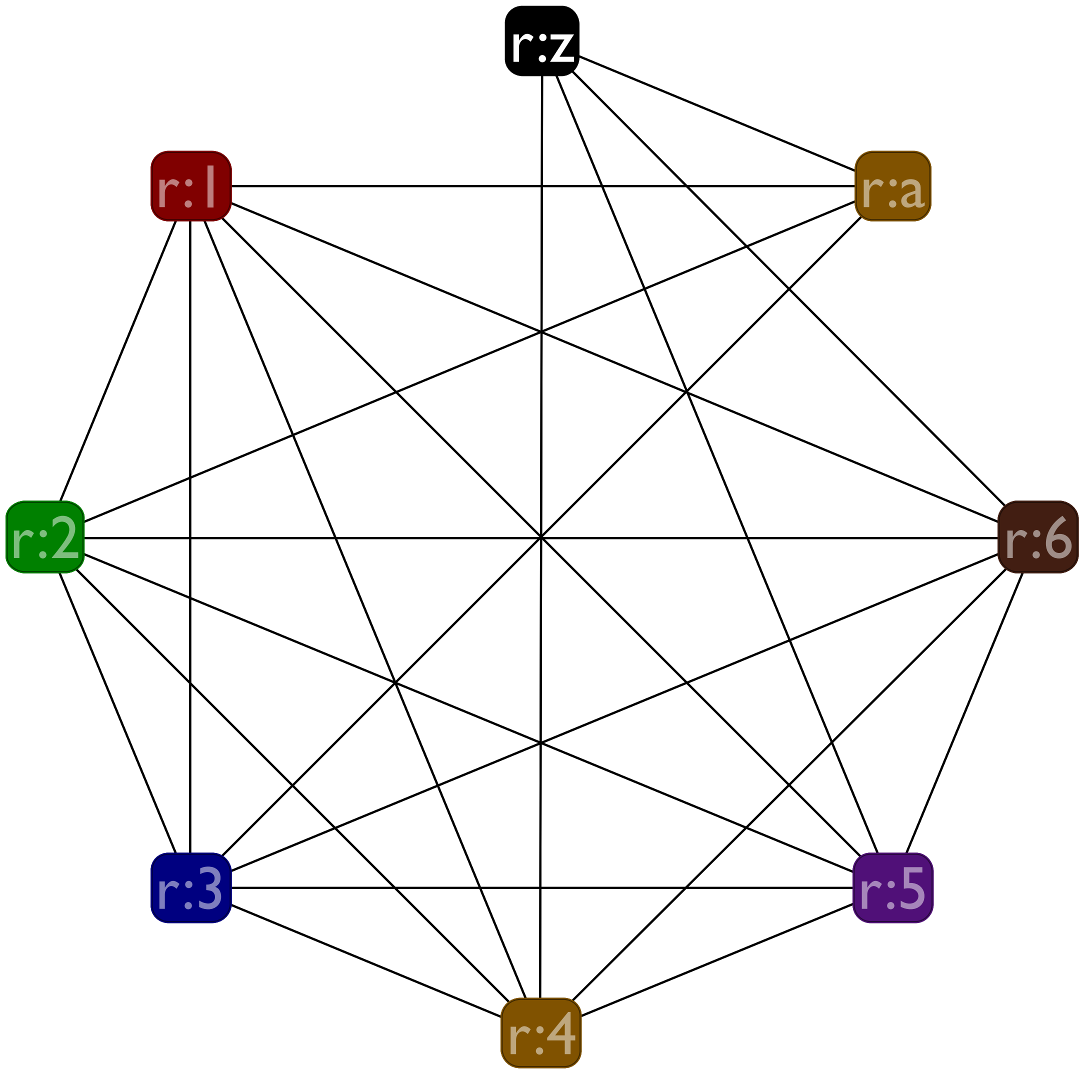

r:z
r:1
r:a
r:2
r:6
r:6
r:a
r:z
r:3
r:5
r:4

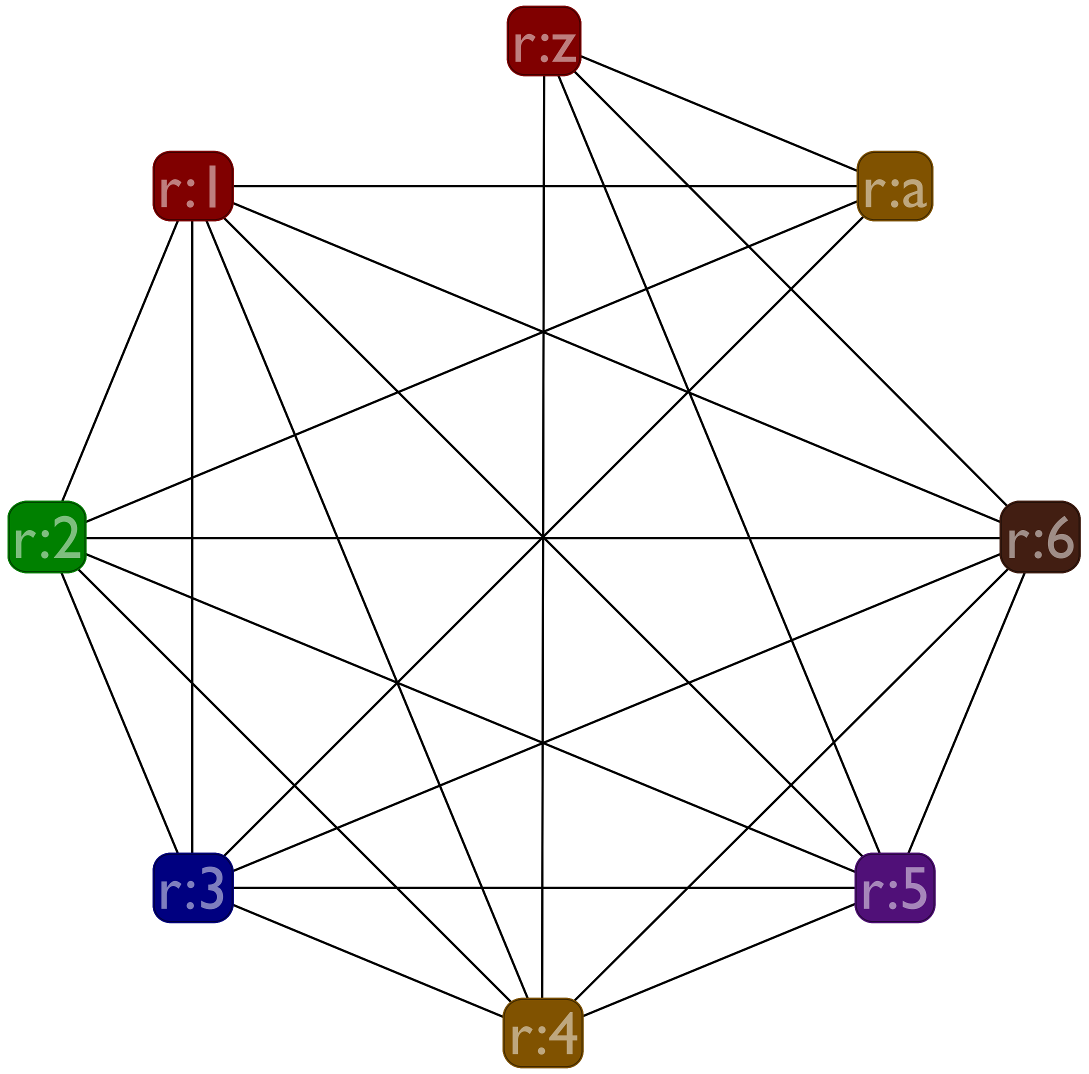

r:z
r:a
r:1
r:2
r:6
r:a
r:z
r:3
r:5
r:4


r:z
r:1
r:a
r:2
r:6
r:a
r:z
r:3
r:5
r:4


r:z
r:1
r:a
r:2
r:6
r:3
r:5
r:z
r:4


r:z
r:l
r:a
r:2
r:6
r:3
r:5
r:4
r:z


r:z
r:1
r:a
r:2
r:6
r:3
r:5
r:4

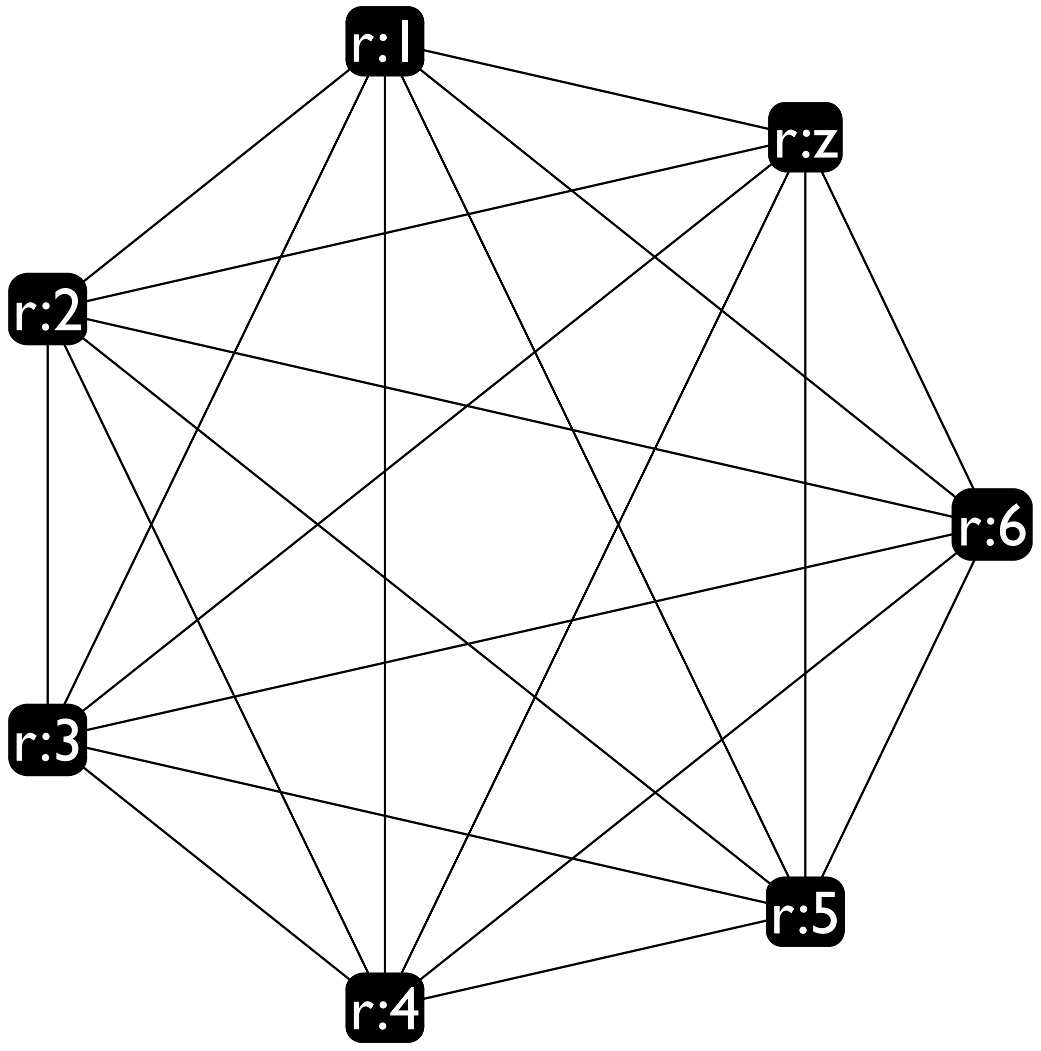

r:z
r:a
r:1
r:6
r:2
r:5
r:3
r:4

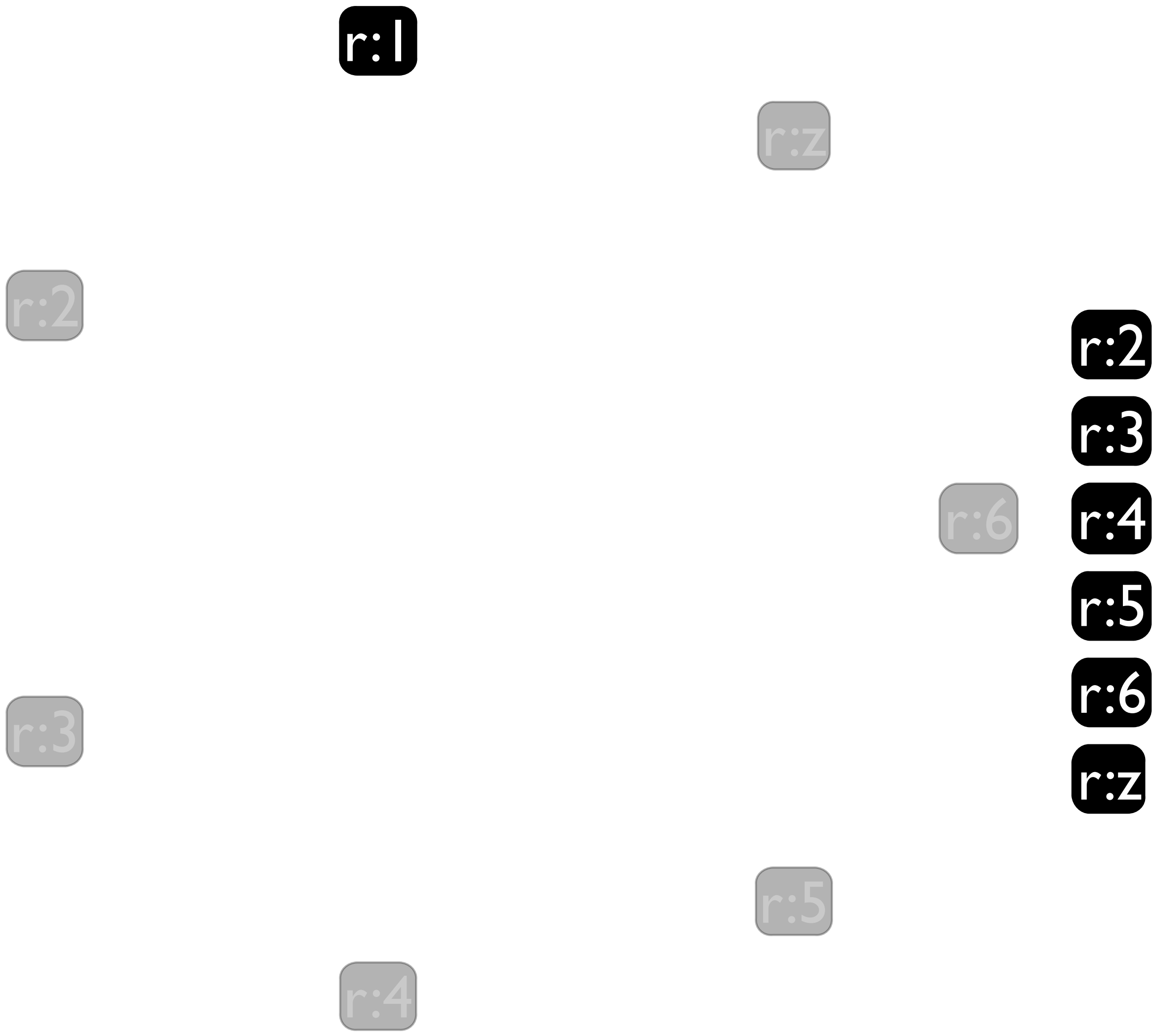
Coalescing Problem

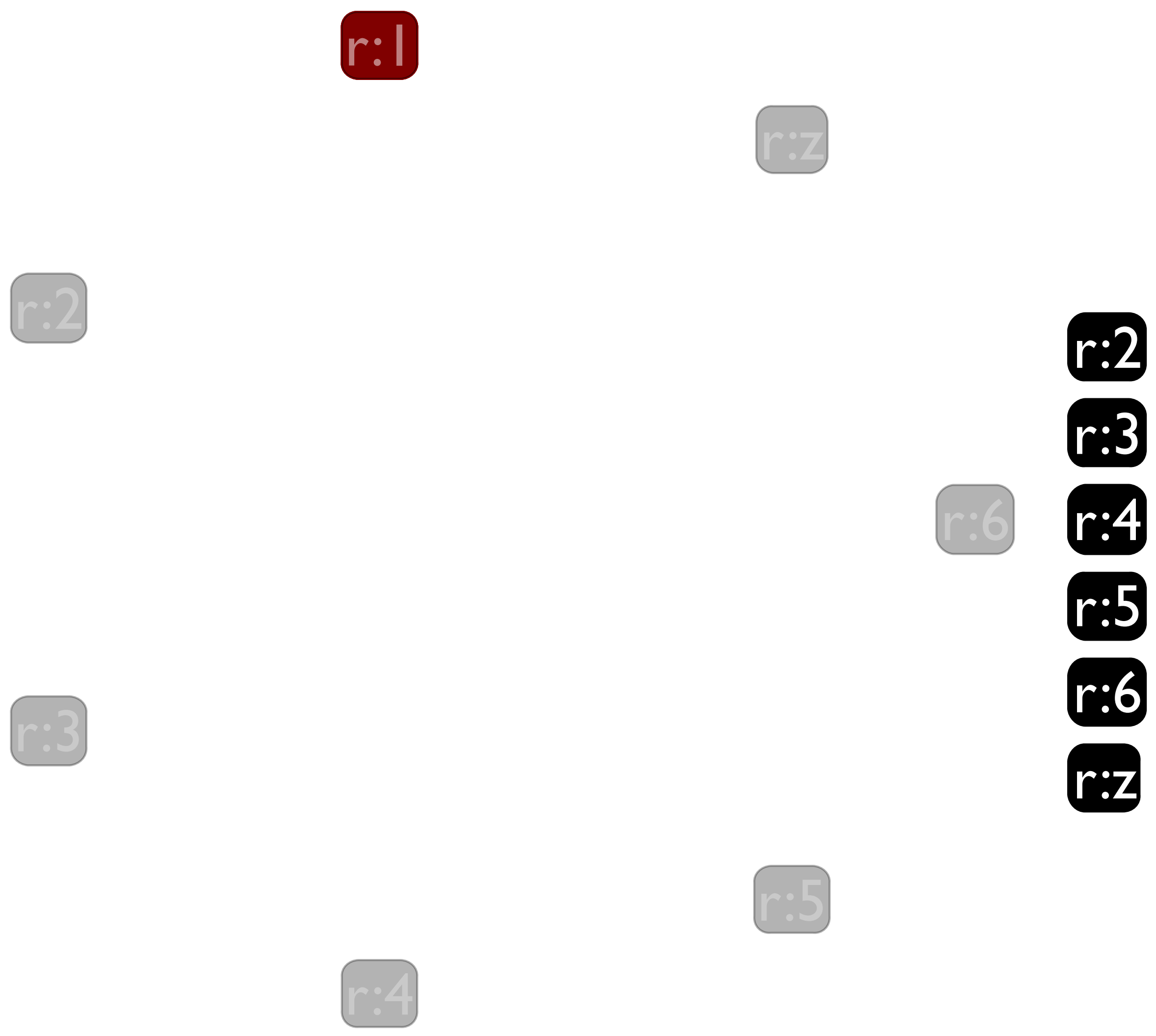
		in	out
1:	`(r:z <- 1)`	()	()
2:	`(r:z <- r:z)`	()	()
3:	`(r:1 <- 1)`	()	()
4:	`(r:2 <- 2)`	()	()
5:	`(r:3 <- 3)`	()	()
6:	`(r:z += 1)`	()	()
7:	`(r:4 <- 4)`	()	()
8:	`(r:5 <- 5)`	()	()
9:	`(r:6 <- 6)`	()	()
10:	`(r:1 += 1)`	()	()
11:	`(r:2 += 1)`	()	()
12:	`(r:3 += 1)`	()	()
13:	`(r:z <- 1)`	()	()
14:	`(r:4 += 1)`	()	()
15:	`(r:5 += 1)`	()	()
16:	`(r:6 += 1)`	()	()
17:	`(r:z += 1)`	()	()

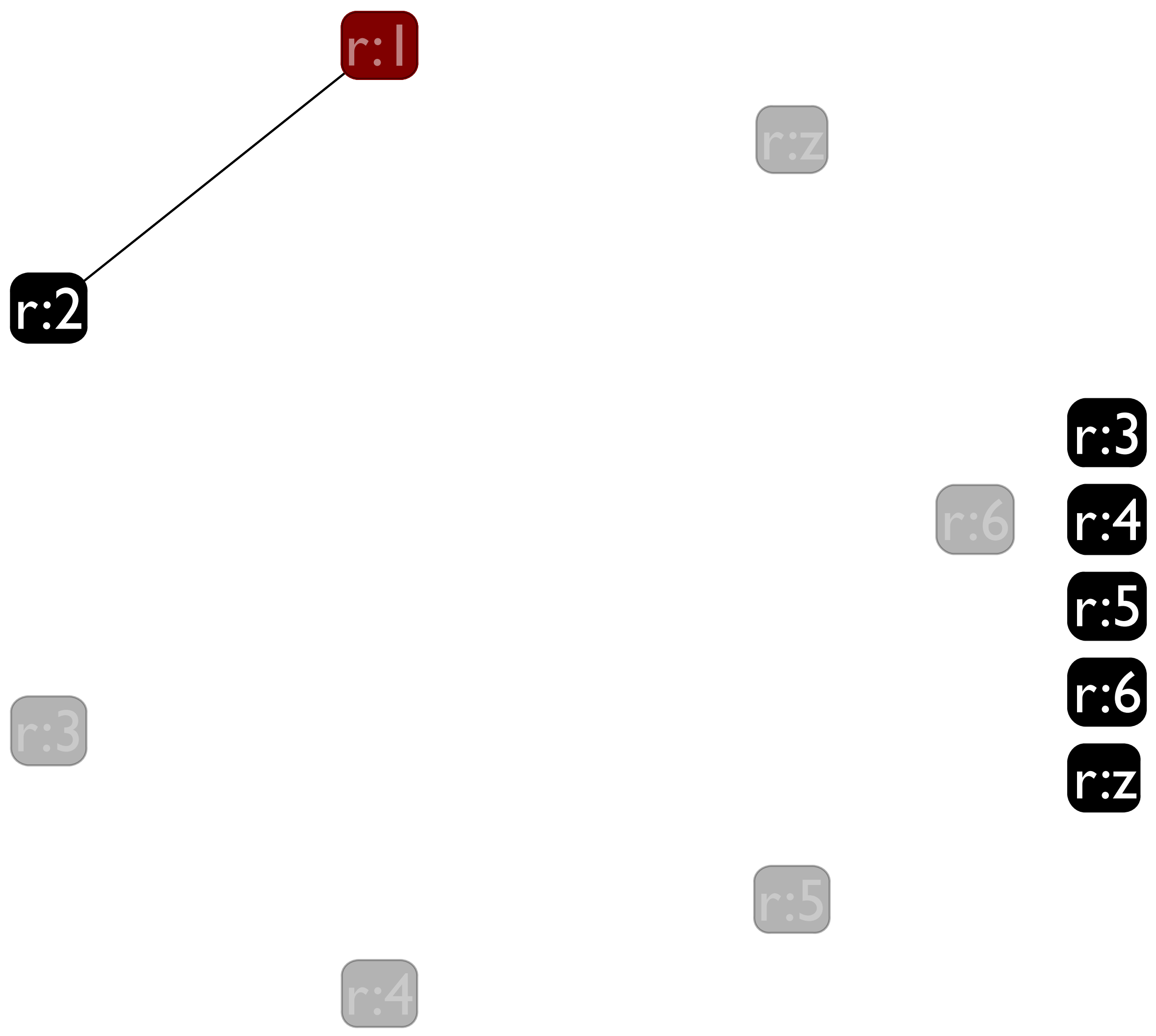
Coalescing Problem

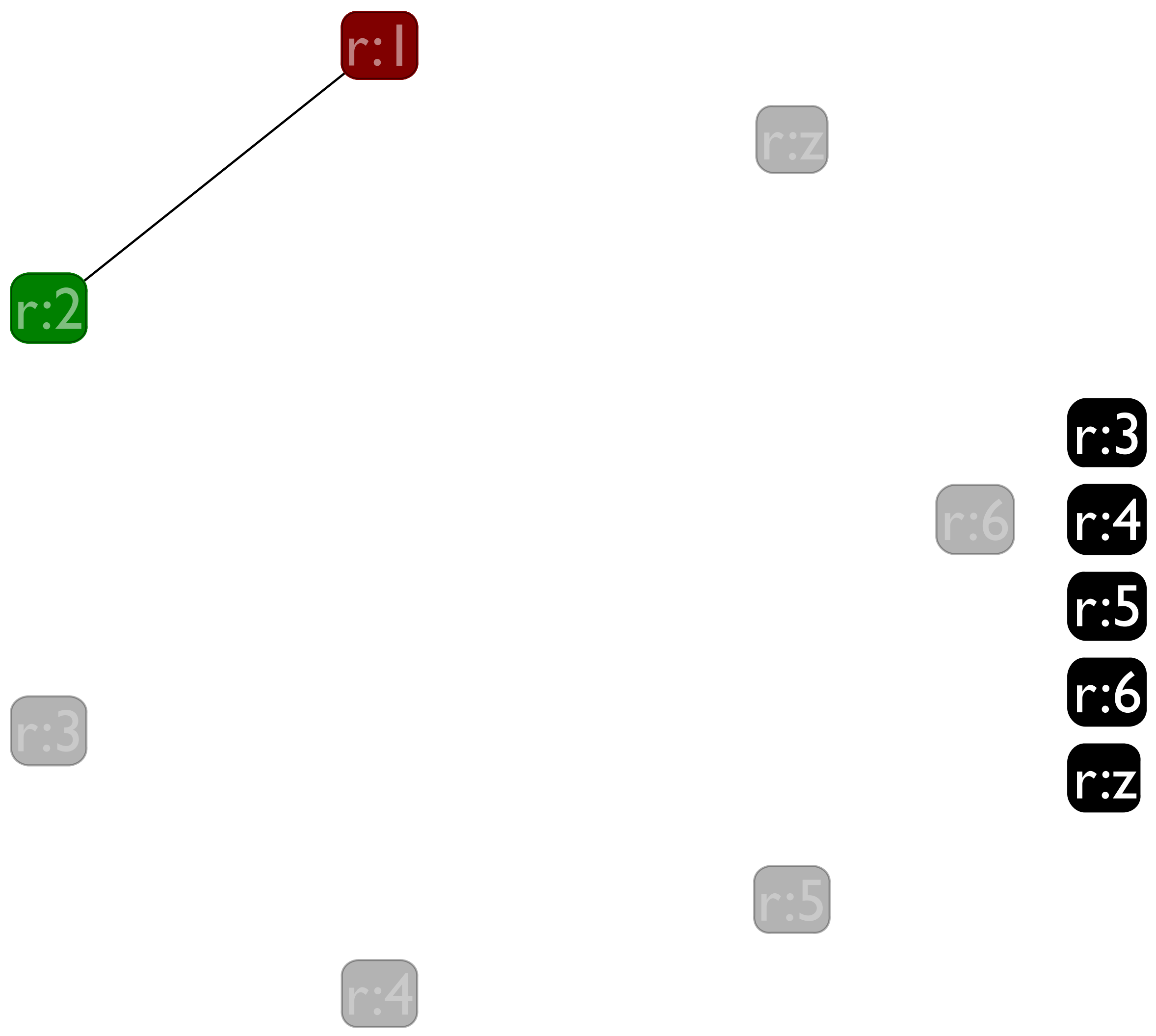
		in	out
1:	`(r:z <- 1)`	()	(r:z)
2:	`(r:z <- r:z)`	(r:z)	(r:z)
3:	`(r:1 <- 1)`	(r:z)	(r:1 r:z)
4:	`(r:2 <- 2)`	(r:1 r:z)	(r:1 r:2 r:z)
5:	`(r:3 <- 3)`	(r:1 r:2 r:z)	(r:1 r:2 r:3 r:z)
6:	`(r:z += 1)`	(r:1 r:2 r:3 r:z)	(r:1 r:2 r:3)
7:	`(r:4 <- 4)`	(r:1 r:2 r:3)	(r:1 r:2 r:3 r:4)
8:	`(r:5 <- 5)`	(r:1 r:2 r:3 r:4)	(r:1 r:2 r:3 r:4 r:5)
9:	`(r:6 <- 6)`	(r:1 r:2 r:3 r:4 r:5)	(r:1 r:2 r:3 r:4 r:5 r:6)
10:	`(r:1 += 1)`	(r:1 r:2 r:3 r:4 r:5 r:6)	(r:2 r:3 r:4 r:5 r:6)
11:	`(r:2 += 1)`	(r:2 r:3 r:4 r:5 r:6)	(r:3 r:4 r:5 r:6)
12:	`(r:3 += 1)`	(r:3 r:4 r:5 r:6)	(r:4 r:5 r:6)
13:	`(r:z <- 1)`	(r:4 r:5 r:6)	(r:4 r:5 r:6 r:z)
14:	`(r:4 += 1)`	(r:4 r:5 r:6 r:z)	(r:5 r:6 r:z)
15:	`(r:5 += 1)`	(r:5 r:6 r:z)	(r:6 r:z)
16:	`(r:6 += 1)`	(r:6 r:z)	(r:z)
17:	`(r:z += 1)`	(r:z)	()

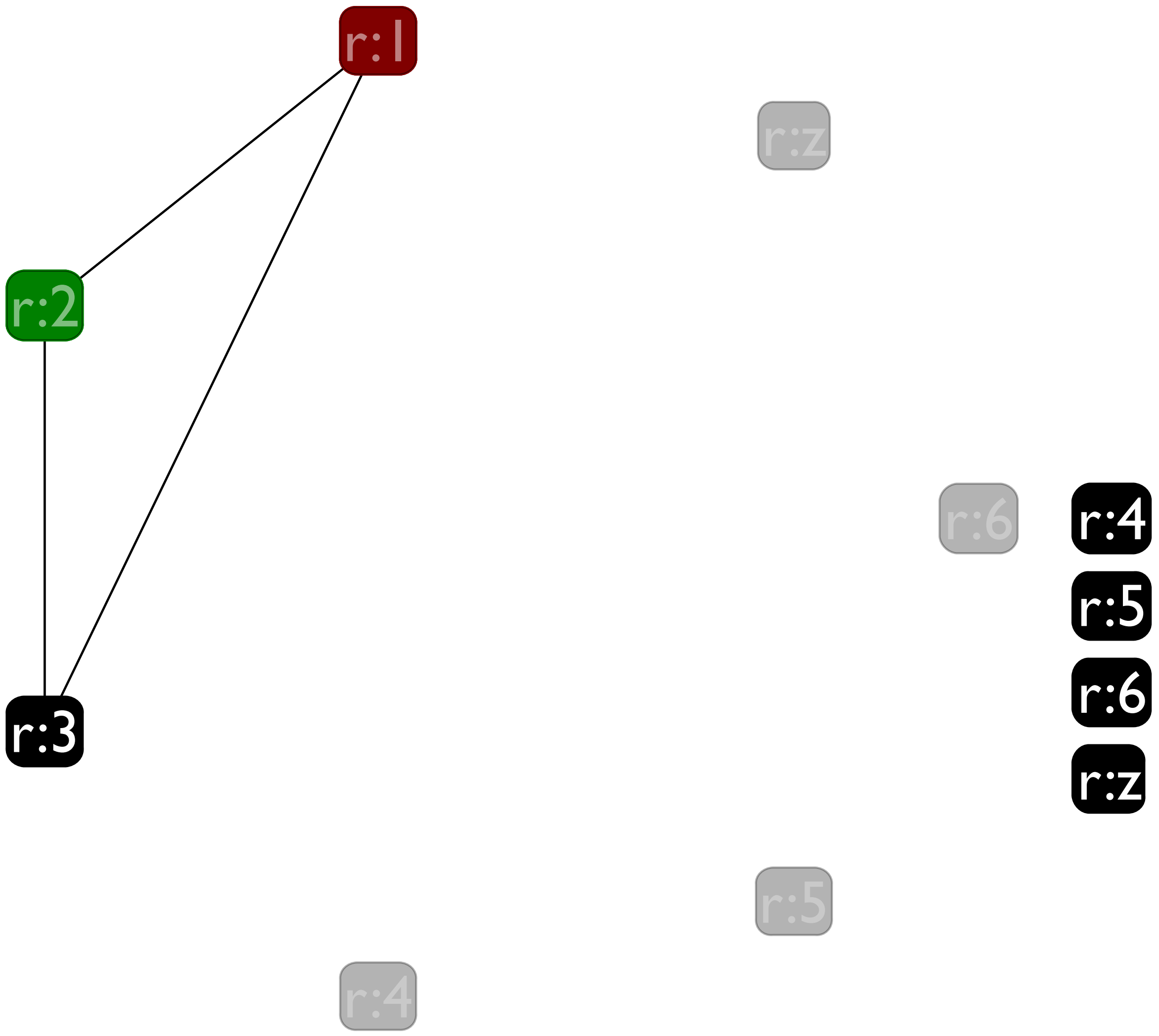

r:1
r:z
r:2
r:6
r:3
r:5
r:4

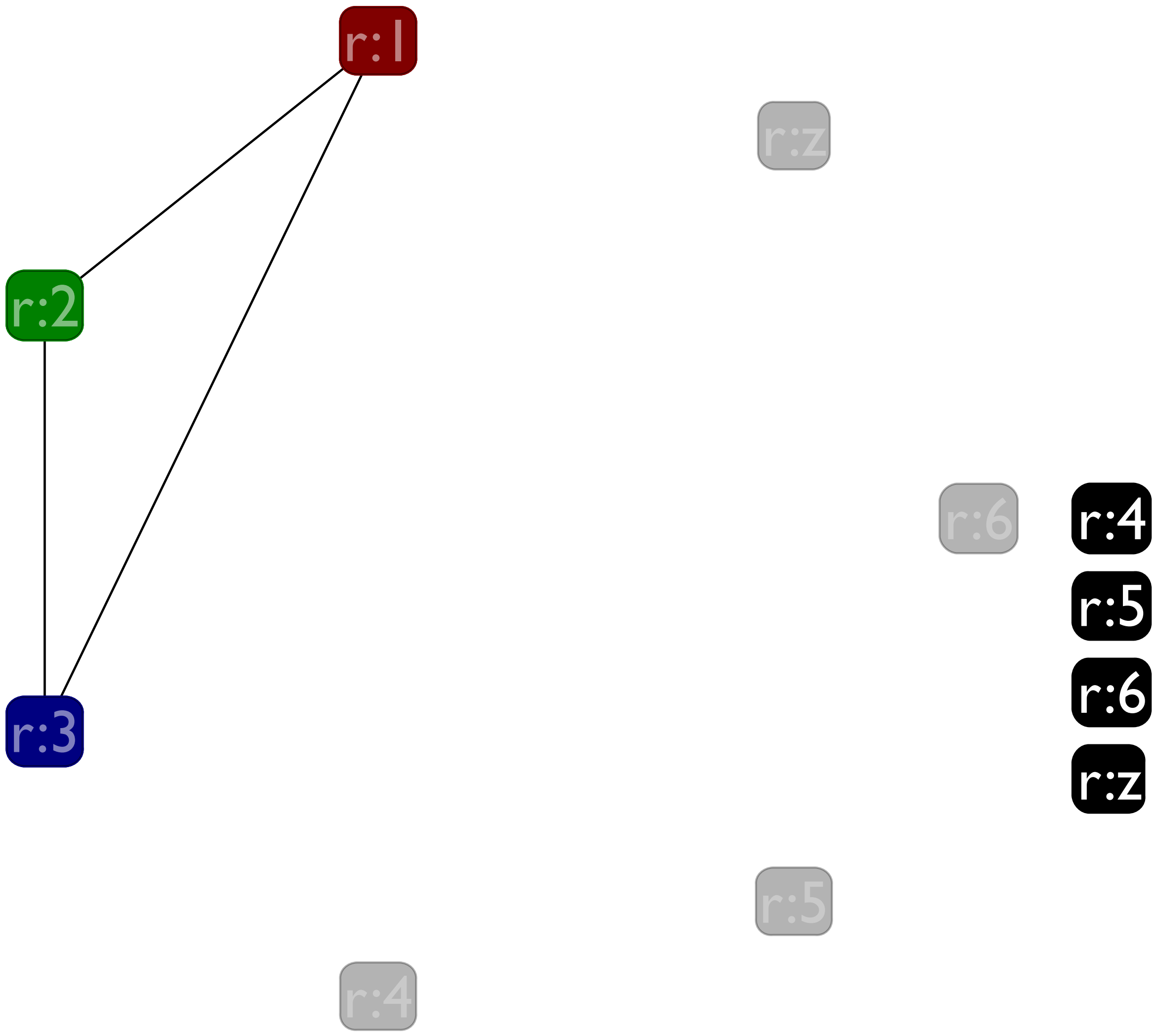

r:1
r:z
r:2
r:6
r:3
r:5
r:4
r:2
r:3
r:4
r:5
r:6
r:z

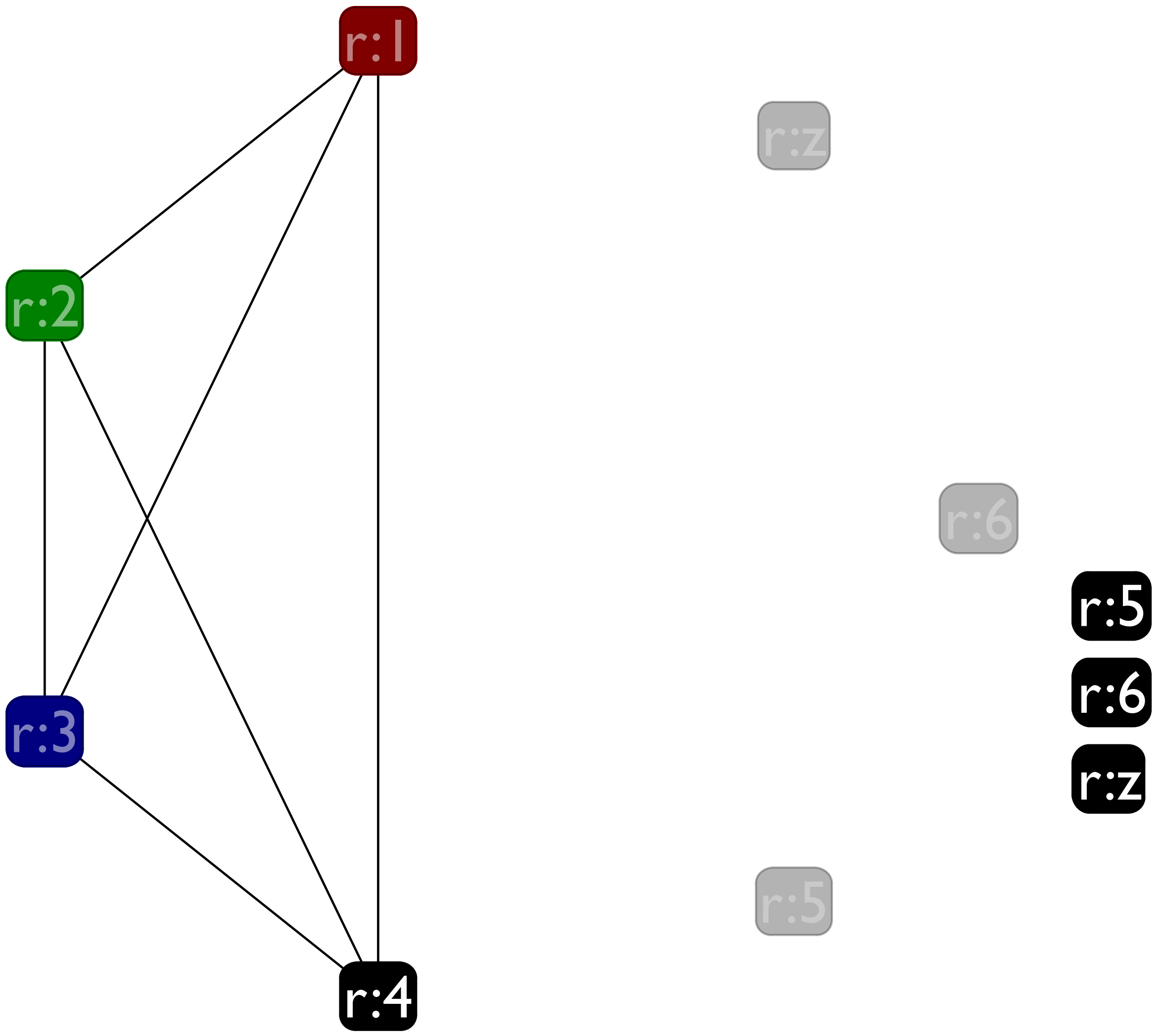

r:1
r:z
r:2
r:6
r:3
r:5
r:4
r:2
r:3
r:4
r:5
r:6
r:z

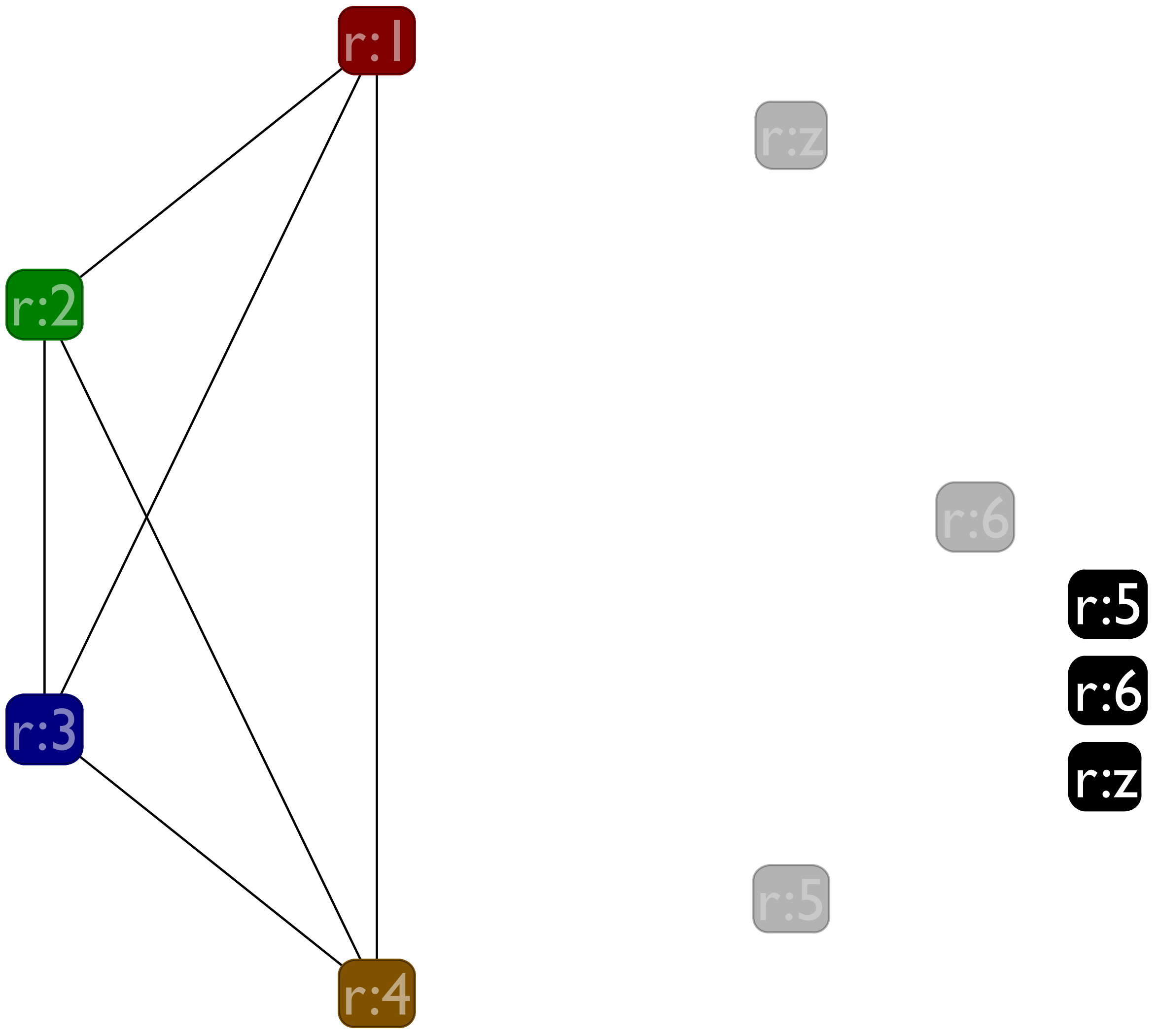

r:1
r:z
r:2
r:3
r:6
r:4
r:5
r:6
r:z
r:3
r:5
r:4

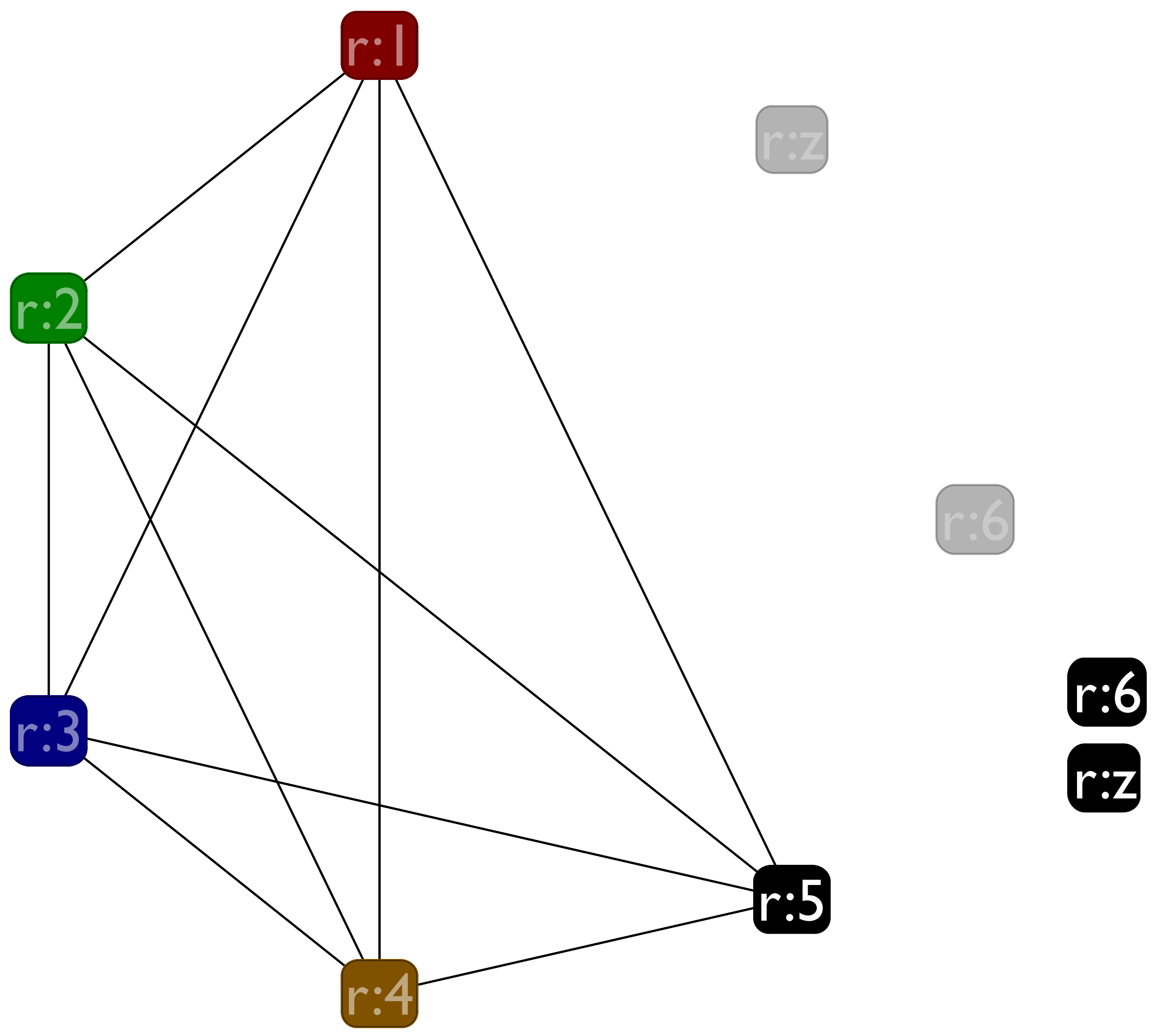

r:1
r:z
r:2
r:3
r:6
r:4
r:5
r:6
r:z
r:3
r:5
r:4

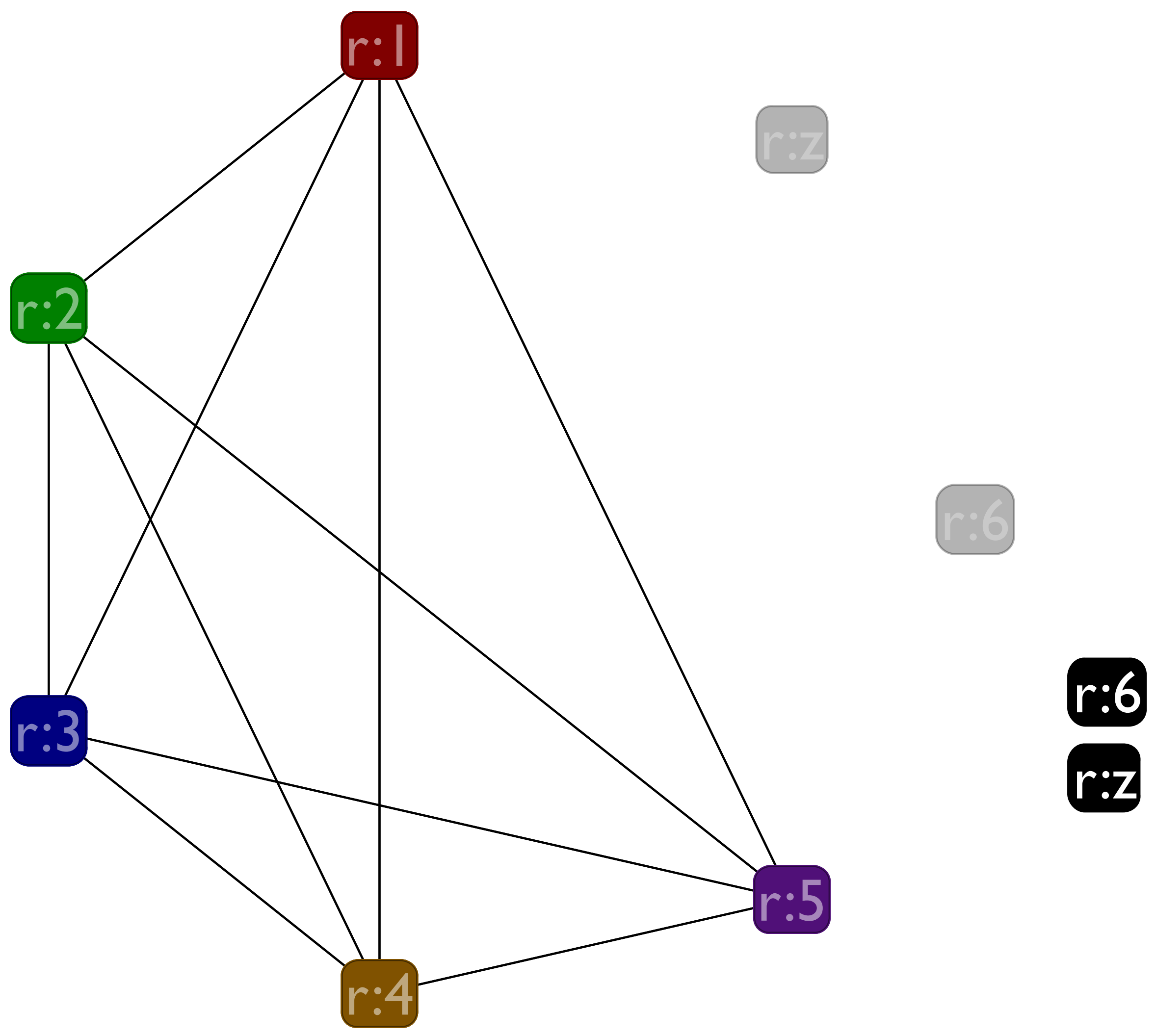

r:1
r:z
r:2
r:6
r:4
r:5
r:6
r:z
r:3
r:5
r:4

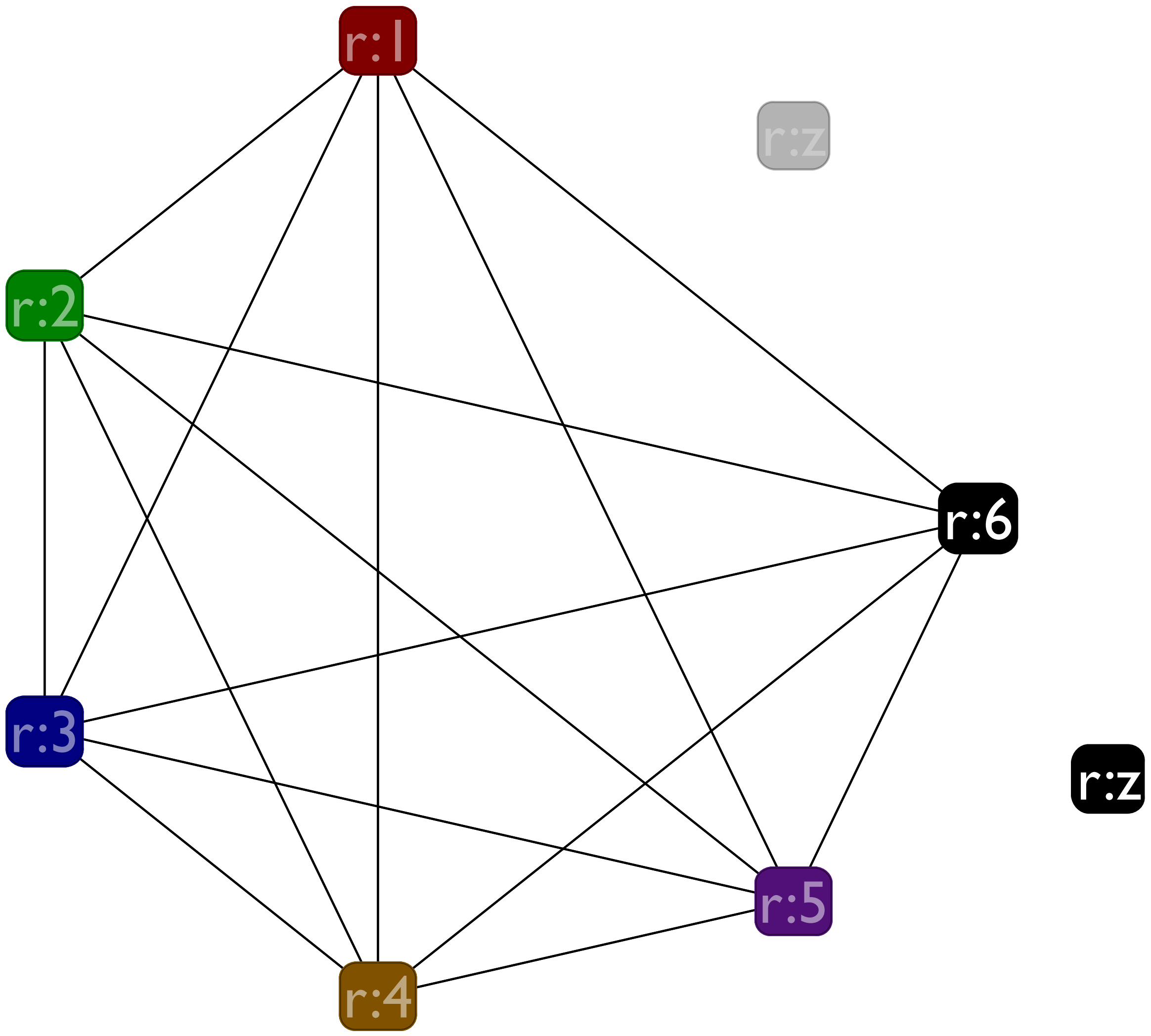

r:1
r:z
r:2
r:6
r:4
r:5
r:6
r:z
r:3
r:5
r:4

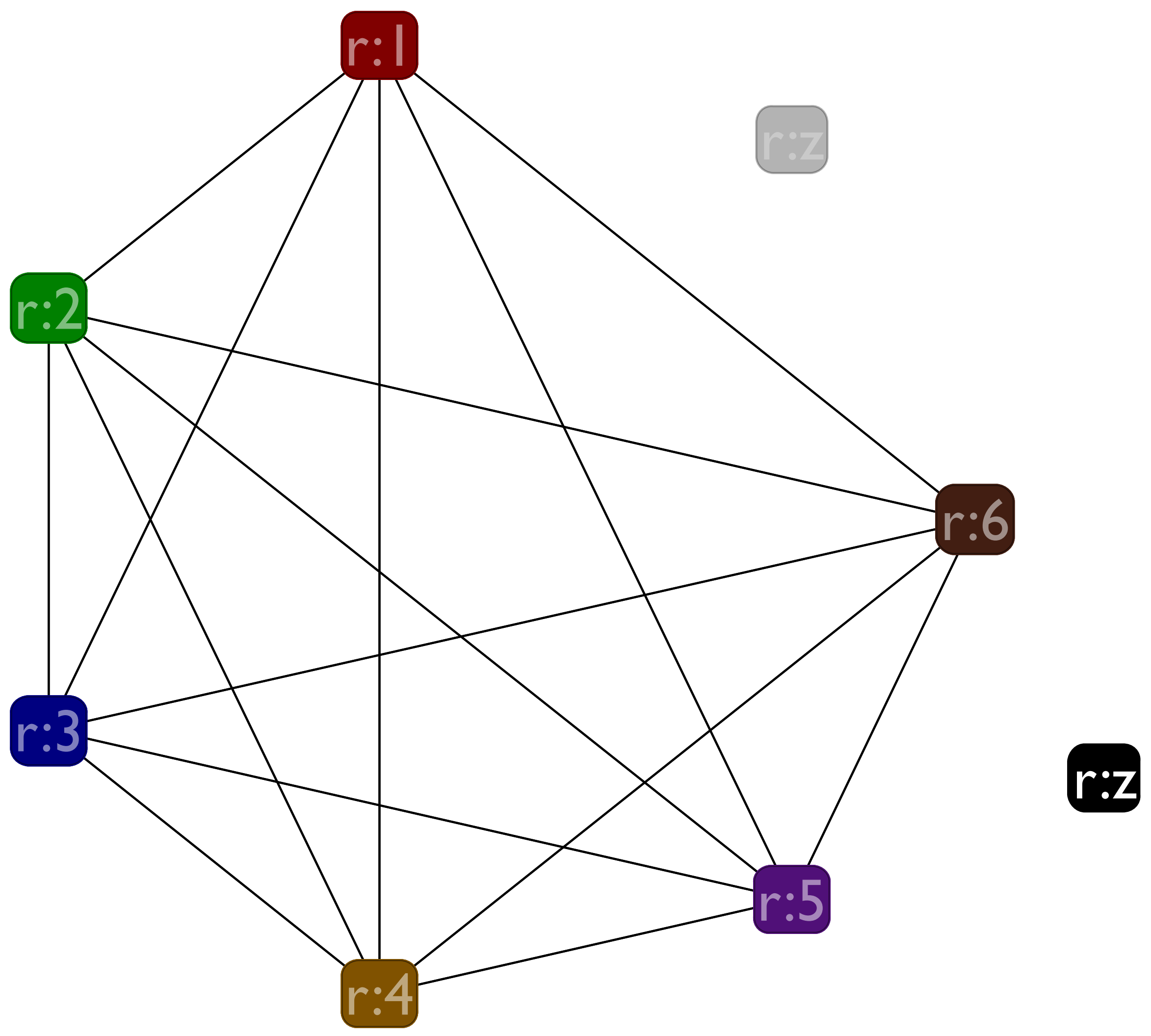

r:1
r:z
r:2
r:6
r:5
r:6
r:z
r:3
r:5
r:4

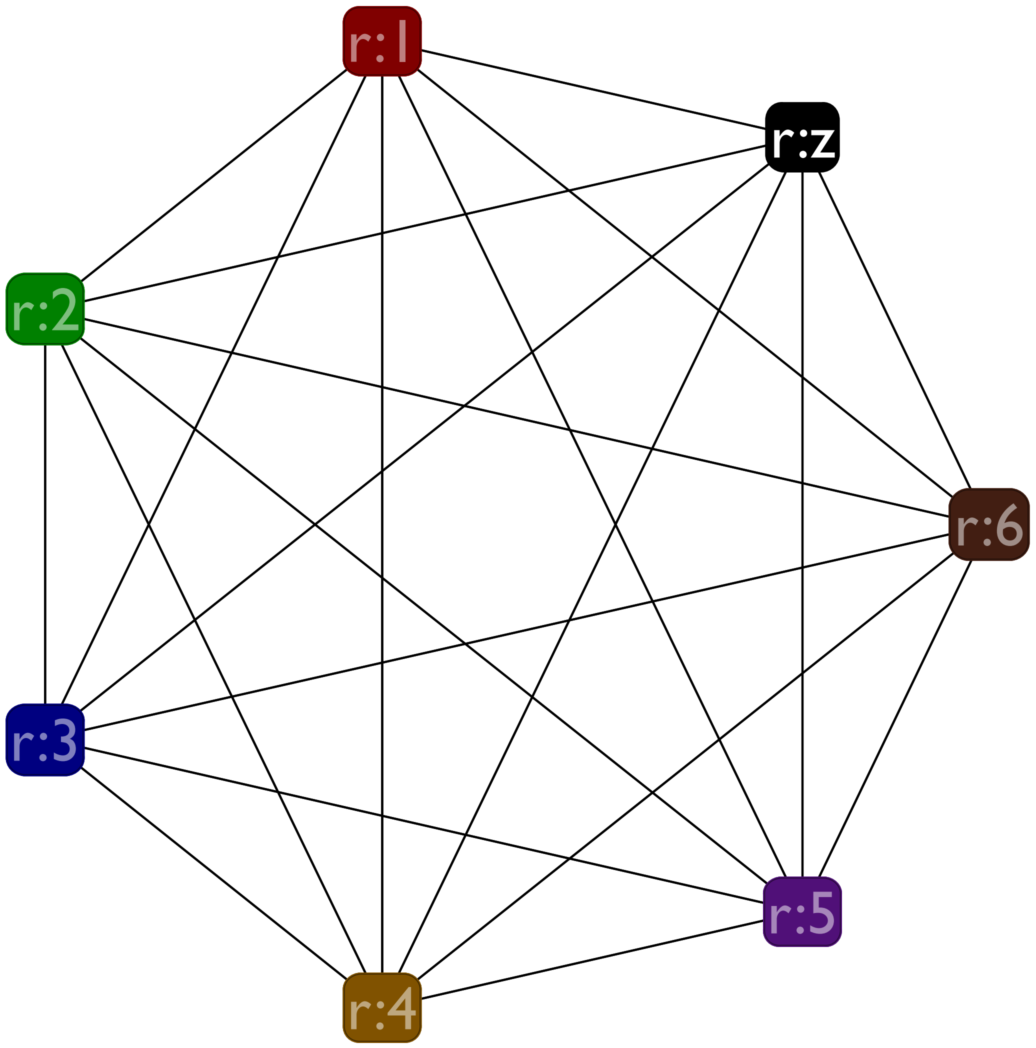

r:1
r:z
r:2
r:6
r:5
r:6
r:z
r:3
r:5
r:4

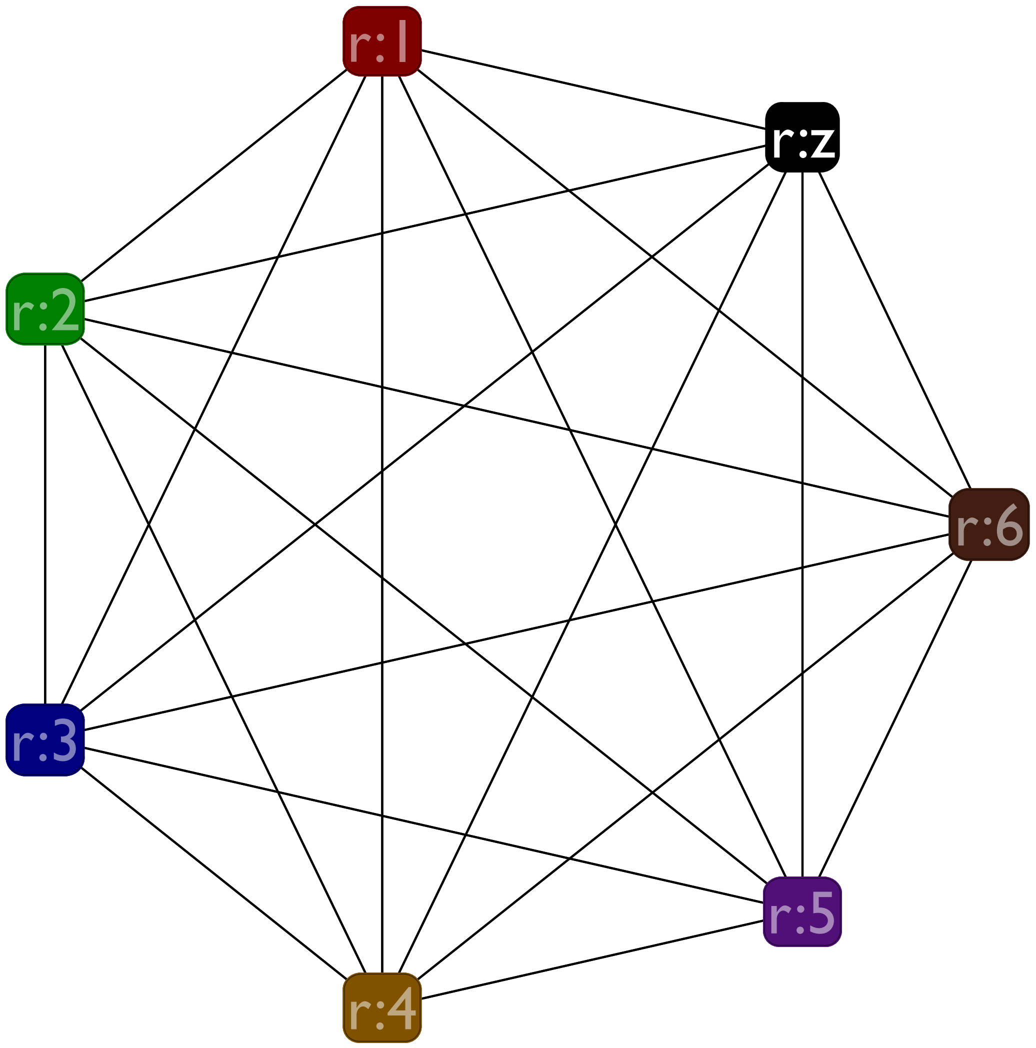

r:1
r:z
r:2
r:6
r:6
r:3
r:z
r:5
r:4


r:1
r:z
r:2
r:6
r:6
r:3
r:z
r:5
r:4


r:1
r:z
r:2
r:6
r:3
r:z
r:5
r:4


r:1
r:z
r:2
r:6
r:3
r:z
r:5
r:4


r:1
r:z
r:2
r:6
r:3
r:5
r:4


r:1
r:z
r:2
r:6
r:3
r:5
r:4

Coalescing during graph coloring

Extend the interference graph with a new kind of edge, called a move edge.

A move edge connects two nodes if there is a `(x <- y)` instruction in the program

Combine two move-edge connected nodes into a single node, if:

- They don’t interfere
- The resulting edge has fewer than 6 neighbors that each have 6 (or more) edges

This ensures the graph is colorable (if it was before)

Roadmap: putting it all together

Programming

There are a number of different modules to put together

- A liveness library: gen & kill functions on instructions; the in and out loop; going from liveness to interference
- A graph library: creating graphs, creating nodes, creating edges, iterating over edges and nodes
- An interference library: build a graph from the liveness information

Programming

There are a number of different modules to put together, cotd

- A coloring library: color a graph using the coloring algorithm
- A spilling library: given a variable and a stack position, rewrite the program to move the variable in and out of the stack right as it is used
- The final translation: When you have a valid coloring, rewrite the variables to use registers, turning the L2 program into an L1 one.

Test

Unit testing

The most underrated part of developing good software is testing it well.

- Build simple (unit) tests for the api for each module, as you design the api
- As you write the code, write a new test for each different case in the code
- Whenever you find a bug, always add a test case *before* fixing the bug; make sure the test case fails so you know you wrote it properly
- Testing only the overall L2 → L1 translation is a sure way to **fail**.