COS320 - Loop transforms

Essential Question:

What is a graph-theoretic def of a loop?
(cfg)

this def has to be syntax-invariant. (nonsensitive to choice of loop e.g. goto, while, for, etc)

First attempt: Strongly Connected Components & But this isn't ~~enogh~~ enough — Nested loops have only one SCC (but we have multiple loops nested) we want to transform separate loops. So this is too general — Not enough information! (In the sense that not a lot of local info within each SCC.

We want to at least capture loops that result from structured programming (e.g. w/o goto's).

What is a loop?

A loop of set of nodes S:

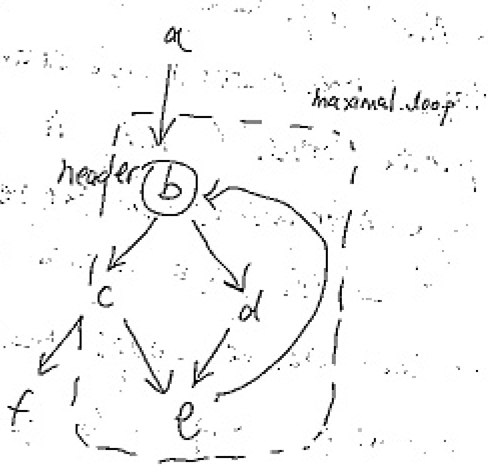
① S is SCC

② header node h that dominates other nodes in S

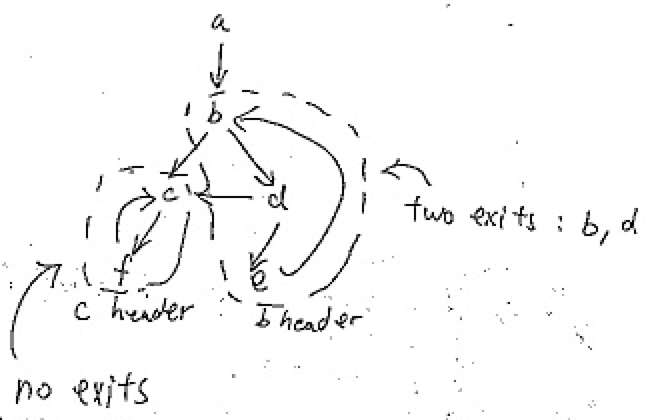
③ No edge from any node outside S to any node inside S except for h.

(Single entry, multiple exits)

ex

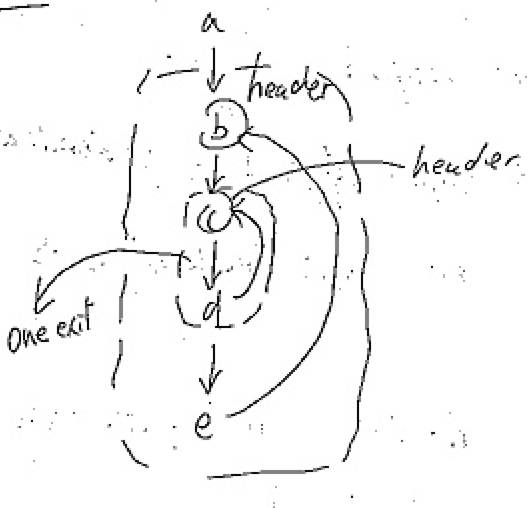


ex 2 loop with 2 nodes

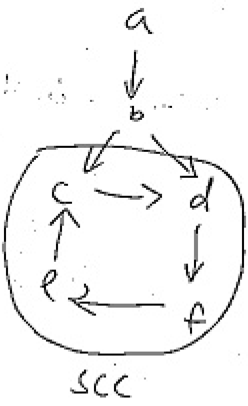


both loop have one entry

ex 3



ex 4 (counterexample)



but two headers (entries) c, e !

So not a loop per our definition.

Identification

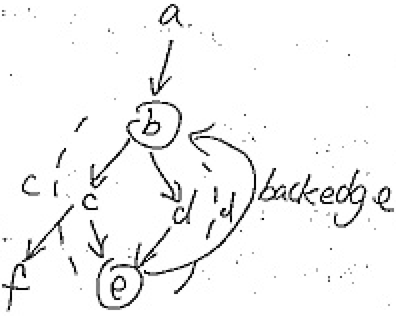
Only works for natural loops:

1. back edge is (u,v) edge s.t. v dominates u.

2. the nat loop of (u,v) back edge is set of nodes

$\{n;\ v$ dominates $n \wedge$ path from n to u w/o $v\}$

- Can do DFS at src of back edge to compute set of nodes for nat. loop:



natural loop $= \{c, d\} \cup \{b, e\}$

Proposition Every nat loop is a loop

We have 3 defs for loops:

① Strongly connected def for loop
- by DFS construction every node has path to u that doesn't pass thru v
- Every node has path from v (path from entry to node to u must incl. v).

But nat. loop obeys this definition.

② Header (v) dominates the loop

Pf by contradiction:

suppose from entry → $d \in$ nat loop that
path
doesn't pass thru v

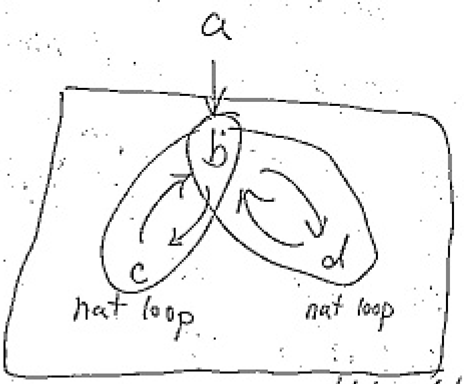
⟹ ∃ path from entry → v that doesn't pass thru v, contradiction.

③ Single entry

Pf by dfs construction, all predecessors of any node except v belongs to loop.

Q: Is every loop natural?

A: No: example:



Not nat loop
(but single entry scc)

→ So 

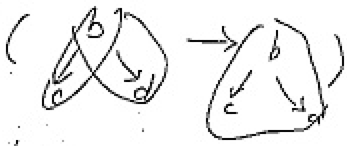
also loop

— but no back edge that encloses this.

but we are happy for this conservative definition.

Nested loops and intersecting loops

Def loop B nested within A if $B \subseteq A$

A node can be header of more than one natural ~~nested~~ loop (intersecting loops that aren't nested) → but we will merge nat loops w/ same header ()

No longer natural but is a loop.

Key property: every loops are either disjoint or nested

- form a forest
- leaves of trees are most deeply nested loops.

* SCC constructed as intersections of natural loops

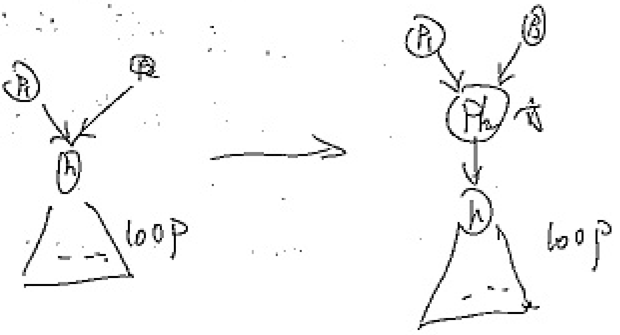
Apply loop transformations "inside-out",

. Start w/ leaf nodes (innermost loops) first.

Loop Preheaders

- Some optimizations (e.g. loop-invariant code motion) require inserting statements immediately before code executes.

- A loop preheader is basic block that is inserted immediately before loop header to serve as a place to store these statements.



loop-invariant Code Motion (LICM)

* Saves cost of recomputing expressions that are lhs-invariant (do not change) inside the loop.
  - Such computations can be moved into preheader as long as they're not side-effecting.

SSA-based LICM

operand is invariant in loop L if

1. it is const
2. it is gid
3. it is rid whose def is not in L
   - 2 holds based on SSA property — it is eval'd in some node that dominates loop preheader.

Now for each computation

$$\%x = opn_1 \ op \ opn_2$$

if opn_1 $\wedge$ opn_2 are invariant.

$\longrightarrow$ move $\%x = opn_1 \ op \ opn_2$ to loop pre-header.

this moves def of $\%x$. outside of the loop so $\%x$ is now invariant.

Induction Variables "等差数列"

<u>Def</u>: Induction var is var $\%x$ such ~~that differ~~ the diff between successive vals in loop is const.

<u>ex</u>: for(i=0; i<n ;i++) $\leftarrow$ Counter var

- Use $\%X(k)$ to denote x in kth iter.

$$\%X(k+1) = \%X(k) + \Delta(\%X)$$ 公差

Useful for several optimizations...

- strength reduction
- loop unrolling
- induction variable elimination
- parallelization
- array bounds-check

basic:

A variable %X is basic induction variable for loop L is increased/decreased by fixed loop ~~co~~ invariant quantity in any iteration of loop: $\Delta(\%X) = C$

$\%X(i+1) = \%X(i) + c$

this is a "syntactic" sequence of assignments.

Derived:

Y is a derived induction variable for loop L if it is an affine function of a basic induction variable:

$X = ax++$

depends on basic induct. var x

$y(i) = a \cdot x(i) + b \longrightarrow \Delta(y) = a \cdot c$

Finding Induction Variables in SSA form

- look for basic induction vars:
  look for ϕ statements in loop header

$$\%X = \phi(\%X_1 \ldots \%X_n)$$

- if value of some basic var is changed in loop then it appears in a ϕ-statement.

- Only look for X_i s.t. each X_i position corresponds to the same vid at the back edge of a loop.

Next find chain of $\%X_K$ leading back to $\%X$ such that each either adds or subtracts an invariant quantity.

Detecting derived induction variables

- Choose basic induction variable %x
- find assignments of form %y = opn, op opn$_2$

where op ∈ {+, −}

opn$_1$, opn$_2$ are either %x

derived induction variables of %x

or loop invariant quantities.

example of application: Strength Reduction

Idea : Replace expensive operation w/ cheaper ones (e.g. replace multiplication with addition).

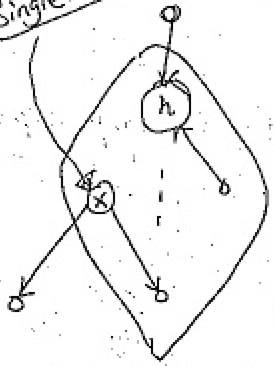
Loop Unrolling

- Some loops are so small that significant time is spent for branch instructions
- Unrolling helps (by trading code size for space

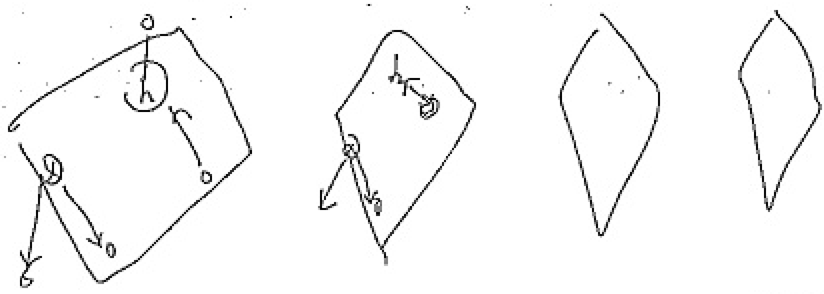
Graphical Illustration

Single exit: bgz t, in, out / $t \geq 0$ is induction variable with $\Delta(t) = c \geq 0$

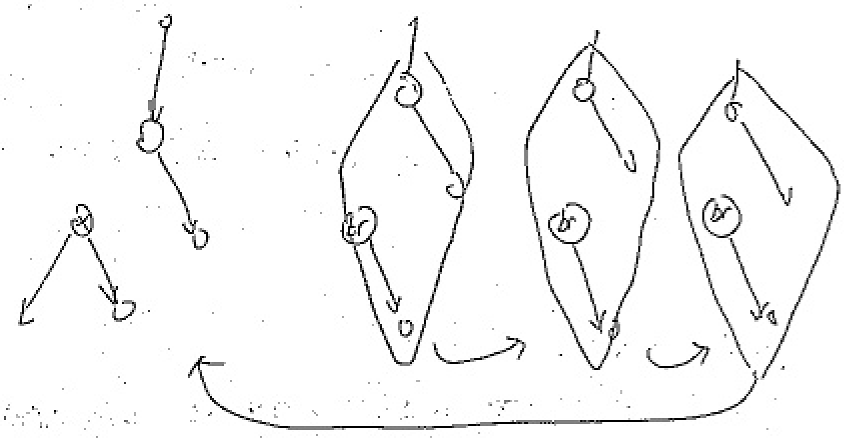
i.e. t is monotonically decreasing ... successive t's are equi difference.



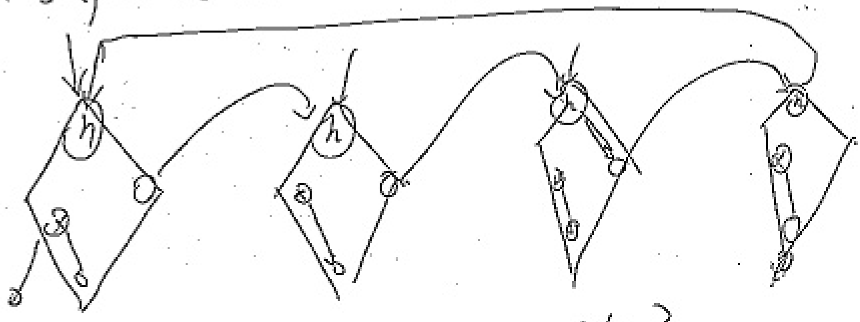
first step: copy loop k times



Next step: Alter conditionals to unconditional branch.



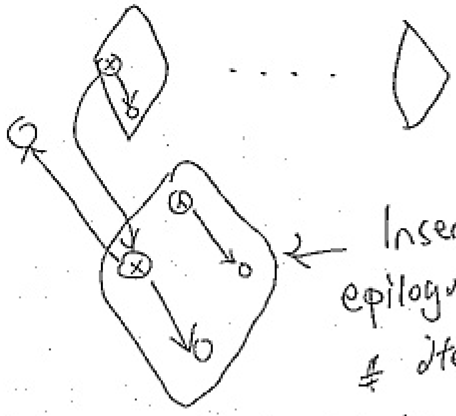
Next step: redirect back-edges to next loop copy



— Doesn't quite work —— why?

A: Still have to "clean up" at the end.

Final step:



Insert epilogue in case # iterations not divisible by 4.

Epilogue: wrap-up

Recap:

Optimizer is series of IR→IR transformations these transformations are typically supported by some analysis that proves the transformation is safe.

Each transformation is simple — when come in series they are mutually beneficial!