Control Dependence, Program Analyses and The Roman Chariots Problem

Keshav Pingali
Cornell University

Gianfranco Bilardi
Universita di Padova, Italy

Organization

1. Optimal Representation of Control dependence

- Definition
- Is the control dependence graph (O(|E|*|V|) space/time) optimal?

2. Our approach:

- Reduce problem to ROMAN CHARIOTS PROBLEM
- Build ***APT*** data structure in O(|E| + |V|) space/time

=> ***APT*** is an optimal representation of control dependence

3. Other applications of APT:

- SSA computation in linear time per variable
- SDEG computation in linear time per problem
- DFG computation in linear time per variable

4. Conclusions:

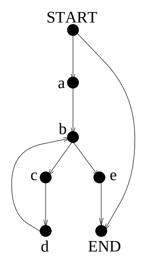
- ***APT*** is a factored form of the CDG which requires ‘filtered search’ to answer queries

Part 1:

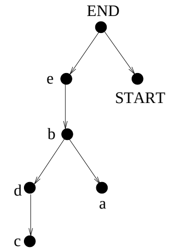
What is an
Optimal Representation
of
Control Dependence?

Control dependence: (Ferrante,Ottenstein,Warren 1987)

Node w is control dependent on edge (u -> v) if
- w postdominates v
- if w ≠ u, w does not postdominate u.



Control Flow Graph

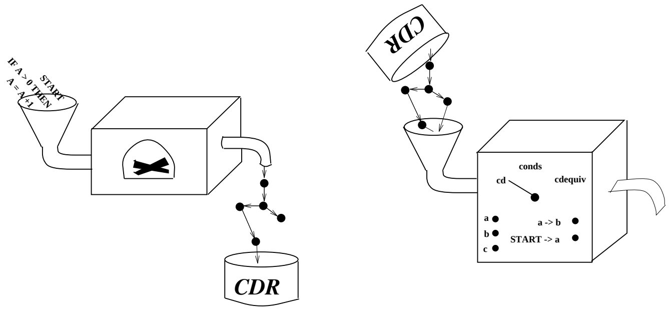


Postdominator Tree

E \ V	a	b	c	d	e
START -> a	√	√			√
b -> c		√	√	√	

Control Dependence Relation

Optimal Control Dependence Computation



Preprocessing ***Query***

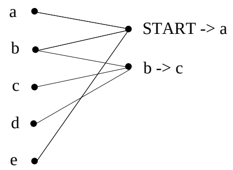
Query time for CD, CONDS,CDEQUIV sets is proportional to set size
Space and time for preprocessing should be minimal.

Control Dependence Graph (CDG)

- bipartite graph between edges and nodes
- connect node v to edge e if node v is control dependent on edge e
- connect nodes in same CDEQUIV class into rings (not shown)

E \ V	a	b	c	d	e
START -> a	✓	✓			✓
b -> c		✓	✓	✓	

Control Dependence Relation



Control Dependence Graph

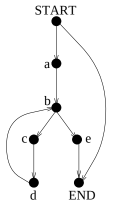
Query time: Proportional to size of output
Preprocessing : O(|E|*|V|) space and time

Queries on Control Dependence Relation:

- ***cd(e):*** set of nodes control dependent on edge e
- ***conds(v):*** set of control dependences of node v
- ***cdequiv(v):*** set of nodes with same control dependences as node v (in same equivalence class as v)

E \ V	a	b	c	d	e
START -> a	✓	✓			✓
b -> c		✓	✓	✓	

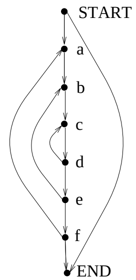
Control Dependence Relation



Control Flow Graph

Applications: program analysis, scheduling for pipelines, parallelization

Worst-case size of control dependence relation:



E \ V	a	b	c	d	e	f
START -> a	✓	✓	✓	✓	✓	✓
f -> a	✓	✓	✓	✓	✓	✓
e -> b		✓	✓	✓	✓	
d -> c			✓	✓		

n nested repeat-until loops => size of CDR is n(n+3)

The size of the CDR can grow quadratically with program size.

There have been many unsuccessful efforts
to reduce the size of the CDG.

" We therefore conjecture that to enumerate [conds sets] in time proportional to [the size of the set] requires a data structure of quadratic size."

[Cytron,Ferrante,Sarkar, PLDI 1990]

Part II:

APT

and the

Roman Chariots Problem

Our Solution:

- reduce control dependence computation to a graph problem called ***Roman Chariots Problem***

- design a data structure called ***APT*** (augmented postdominator tree)

  (a) which can be built in O(|E|) space and time, and
  (b) which can be used to answer CD,CONDS and CDEQUIV queries in time proportional to output size.

APT is a data structure for optimal control dependence computation.

Key Idea (I): Exploit structure of relation

Analogy: Postdominator relation

- queries: immediate pdom of node, all pdoms of node
- size of relation is $O(|V^2|)$
- relation is transitive, so build transitive reduction (pdom tree) in O(|E|) time [Harel,Tarjan]
- query time using pdom tree is optimal

=> There is no point in constructing the entire relation

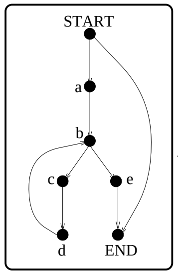
What structure is there in the control dependence relation?

Control dependence relation:

- nodes that are control dependent on an edge e form a simple path in the postdominator tree
- in a tree, a simple path is uniquely specified by its endpoints

Postdominator tree + endpoints of each control dependence path can be built in O(|E|) space and time

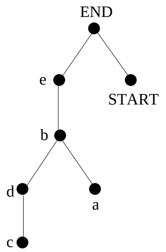
Example:



Control Flow Graph

E \ V	a	b	c	d	e
START -> a	✓	✓			✓
b -> c		✓	✓	✓	

Control Dependence Relation



Postdominator Tree

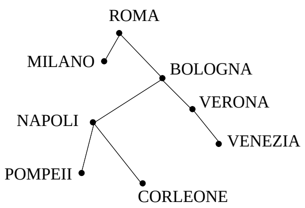
E \ Path	
START -> a	[a,e]
b -> c	[c,b]

Path Array A

O(|E|) Representation of the Control Dependence Relation

How can we use the compact representation of the CDR to answer queries for CD,CONDS and CDEQUIV sets in time proportional to output size?

Roman Chariots Problem



Tree: - nodes are cities
- edges are roads

Route # \ Path	
I	[MILANO,ROMA]
II	[POMPEII,BOLOGNA]
III	[VENEZIA,ROMA]

Cities on route ordered by ancestor relation
In route [x,y], x is descendant of y

Given a tree T, and an array A of chariot routes specified by endpoints, design a data structure to answer the following queries in optimal time.

(a) CD(n): Which cities are served by chariot n?
(b) CONDS(w): Which chariots serve city w?
(c) CDEQUIV(w): Which cities are served by the same chariots that serve w?

CD(n): Which cities are served by chariot n?

Query procedure: (similar to FOW 87)

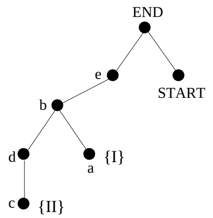
- Look up entry for chariot n in Route Array (say it is [x,y])
- Traverse nodes in tree T, starting at x and ending at y
- Output all nodes encountered in traversal

(cf. CDG: many routes can share tree nodes/edges)

CD query time is proportional to output size.

Key Idea (II): Cache route information in tree

At each node n in the tree, keep a list of chariot # s whose bottom node is n.



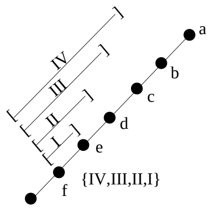
Chariot # \ Route	
I	[a,e]
II	[c,b]

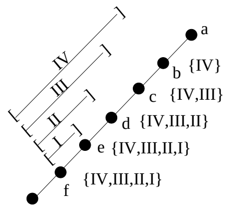
Query procedure: CONDS(w)

```
for each descendant d of w do
    for each route c = [x,y] in list at d do
        if w is a descendant of y
        then output c; fi
    od
od
```

Query time is proportional to # of descendants + size of all lists at descendants

Step 3: Cache route at multiple nodes.





Two extremes:

(1) Chariot # stored only at bottom node of route

Space : O(|V| + |A|)

Query Time: O(|V| + |Output|)

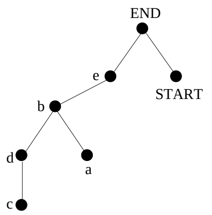
(2) Chariot # stored at all nodes on route

Space: O(|V|*|A|)

Query Time: O(|Output|)

Can we have a disciplined caching policy to have linear space and optimal query time?

CONDS(w): Which chariots serve city w?



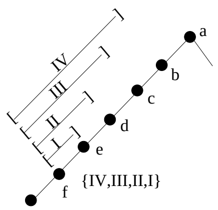
Chariot # \ Route	
I	[a,e]
II	[c,b]

Query procedure:

```
for each chariot c in Route Array do
    let route of c be [x,y];
    if w is an ancestor of x
       and w is a descendant of y
    then output c; fi
od
```

Can we avoid examining all routes in Route Array?

Refinement: Sort each list by decreasing length.



Chariot # \ Route	
I	[f,e]
II	[f,d]
III	[f,c]
IV	[f,b]

Query procedure: CONDS(w)

```
for each descendant d of w do
    for each route c = [x,y] in list at d do
        if w is a descendant of y
        then output c;
         else BREAK; fi od
od
```

At most one ‘non-overlapping’ path is examined at a descendant =>

Query time is proportional to size of output + # of descendants

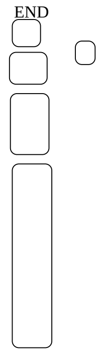
How do we construct zones?

(I) Invariant: For any node v, $|Z_v| \leq \alpha |A_v| + 1$
where α is a design parameter.

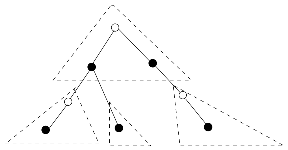
Query time for CONDS(v) $= O(|A_v| + |Z_v|)$
$= O((\alpha + 1) |A_v| + 1)$
$= O(|A_v|)$

(II) Build zones bottom-up, making them as large as possible w/o violating invariant

v is a leaf node => make v a boundary node
v is an interior node =>
if $(1 + \sum_{u \in \text{children}(v)} |Z_u|) > \alpha |A_v| + 1$
then make v a boundary node
else make v an interior node



Key idea (III): Cache a route at multiple nodes



Divide tree into ZONES

Query procedure:
Visit only nodes below query node and in the same zone as query node

Zone construction: For all nodes v, $|Z_v| \leq \alpha |A_v| + 1$

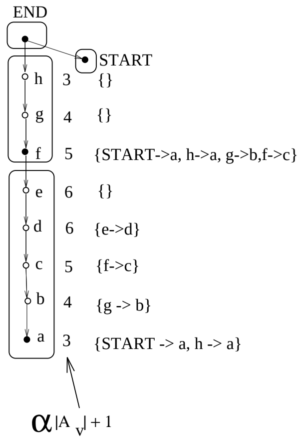
=> Query time $|A_v| + |Z_v| \leq (\alpha + 1) |A_v|$

Caching Rule:

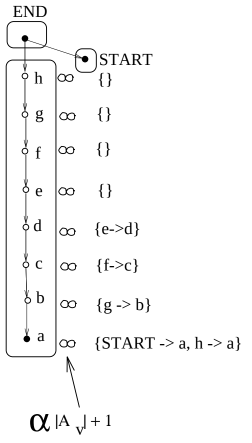
- Nodes are partitioned into
  - boundary nodes: lowest nodes in zone
  - interior nodes: all other nodes
- Caching rule:
  - boundary node: store all chariots serving node
  - interior node: store all chariots whose bottom node is that node
- Our algorithm: bottom-up, greedy zone construction

=> space requirements $< |A| + |V| / \alpha$

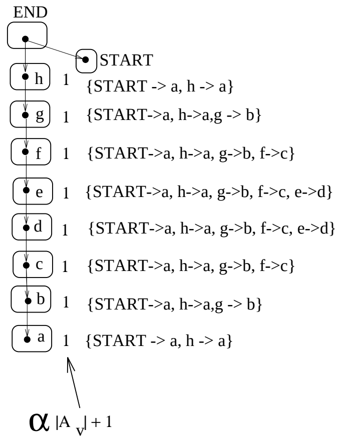
$\alpha = 1$ (some caching)



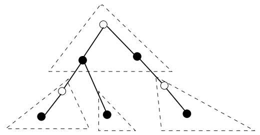
$\alpha = >>$ (no caching)



$\alpha = <<$ (full caching)



Summary of CONDS Approach:



Query Time: $(\alpha + 1)\,|A_V|$

Space : $|A| + |V| / \alpha$

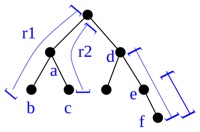
- Parameter α is used to partition tree into zones
  - α << : lower query time, increased space requirements
  - α >> : higher query time, lower space requirements
- Nodes are partitioned into
  - boundary nodes: lowest nodes in zone
  - interior nodes: all other nodes
- Caching rule:
  - boundary node: store all chariots serving node
  - interior node: store all chariots whose bottom node is that node
- Query procedure:
  Visit only nodes below query node and in the same zone as query node

CDEQUIV(v): Which cities are served by same chariots that serve v?

- Ferrante, Ottenstein, Warren 87: $O(|E|^3)$ using hashing for set equality
- Cytron, Ferrante, Sarkar 90: $O(|E|^2)$
- Ball 92: O(|E|) for structured programs
- Podgurski 93: O(|E|) for forward control dependence in general graphs
- Johnson, Pearson, Pingali 94: O(|E|) for general graphs (optimal)

CDEQUIV for Roman Chariots Problem

- cleaned-up version of JPP94 algorithm
- compute two finger prints for CONDS sets
  - . size of CONDS set
  - . Lo:lowest node contained in all routes of CONDS set



Lo(CONDS(a)) = a
Lo(CONDS(d)) = f
Lo(CONDS(e)) = f

Two CONDS sets are equal iff they have the same finger-prints.
Can compute finger-prints in O(|V| + |A|) space and time

APT

1. Postdominator tree with bidirectional edges

2. dfs-number[v]: integer

- used for ancestorship determination in CONDS query

3. boundary?[v]: boolean

- true if v is a boundary node, false otherwise
- used in CONDS query

4. L[v]: list of chariots #'s/control dependences

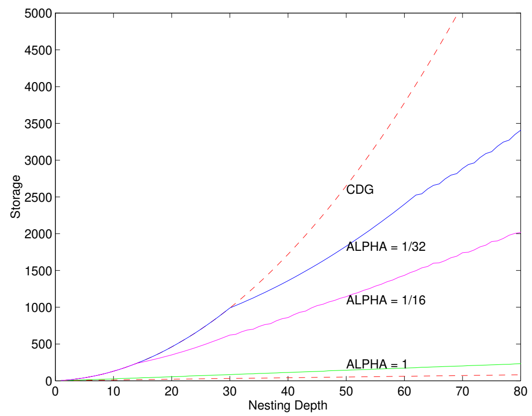
- boundary node: all chariots serving v (all control dependences of v)
- interior node: all chariots whose bottom node is v (all immediate control dependences of v)
- used in CONDS query

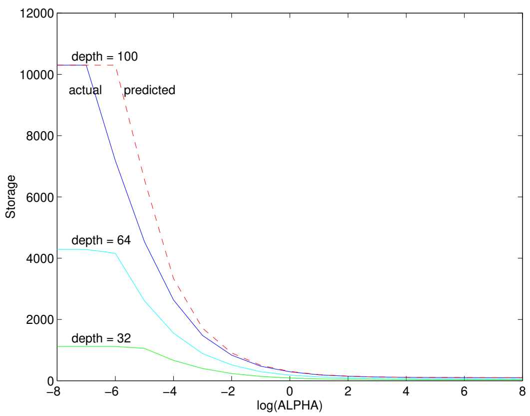
5. R[v]: pointer to CDEQUIV equivalence class

- used in CDEQUIV query

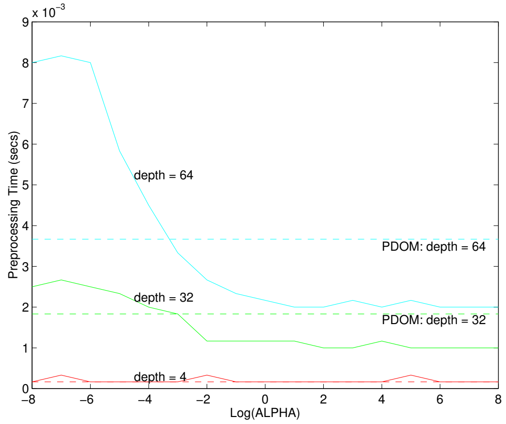
Query time: $(\alpha+1)$ **** output-size***
Space: $|E| + |V| / \alpha$

Experimental Results

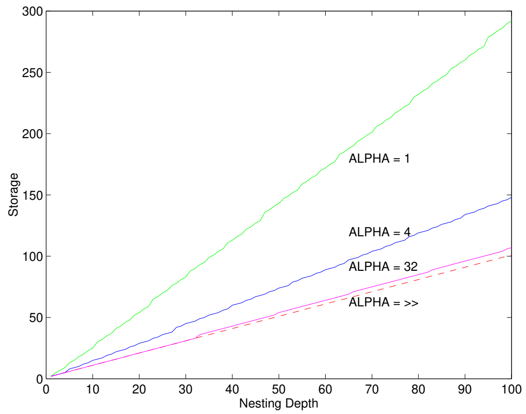



depth = 100
actual
predicted
depth = 64
depth = 32
Storage
log(ALPHA)

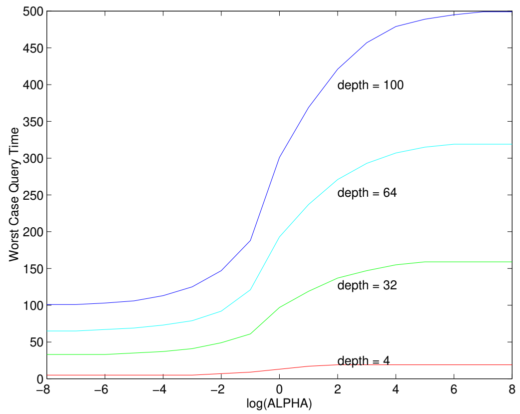
34


x 10^-3
depth = 64
PDOM: depth = 64
depth = 32
PDOM: depth = 32
depth = 4
Preprocessing Time (secs)
Log(ALPHA)

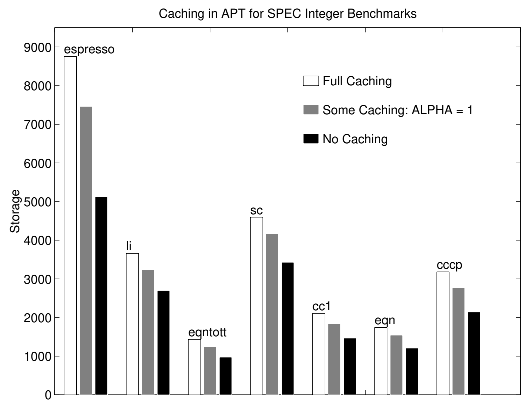
36


ALPHA = 1
ALPHA = 4
ALPHA = 32
ALPHA = >>
Storage
Nesting Depth

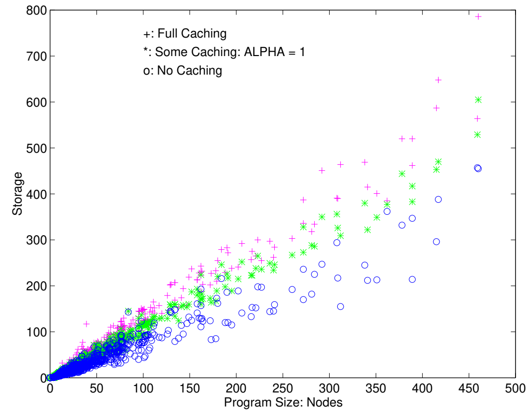
33


depth = 100
depth = 64
depth = 32
depth = 4
Worst Case Query Time
log(ALPHA)

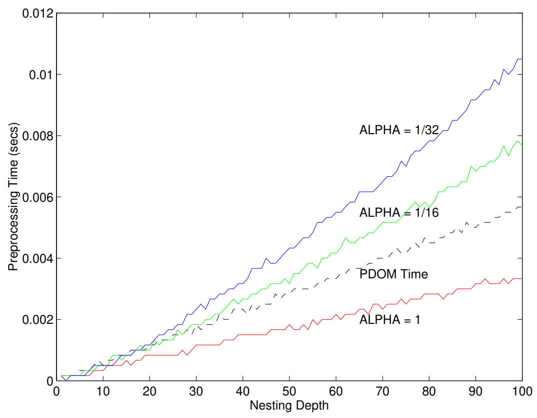
35


Caching in APT for SPEC Integer Benchmarks
Storage
espresso
li
eqntott
sc
cc1
eqn
cccp
Full Caching
Some Caching: ALPHA = 1
No Caching

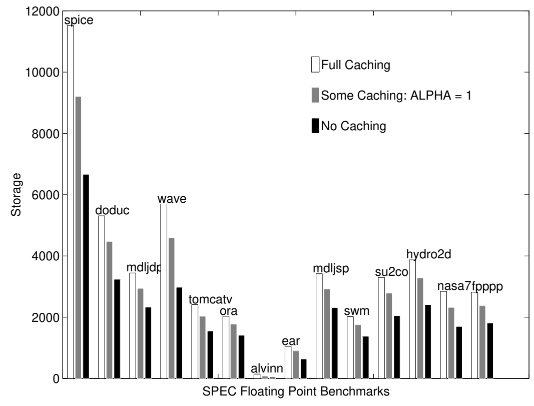
38


+: Full Caching
*: Some Caching: ALPHA = 1
o: No Caching
Storage
Program Size: Nodes

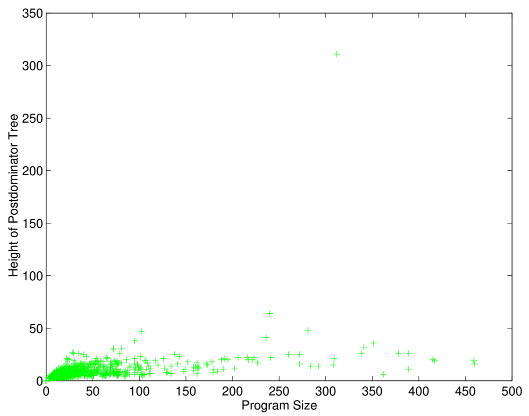
40


Preprocessing Time (secs)
Nesting Depth
ALPHA = 1/32
ALPHA = 1/16
PDOM Time
ALPHA = 1

37

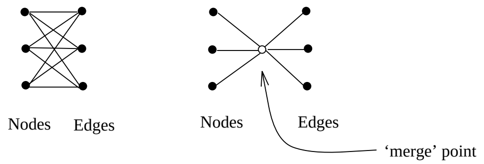

Storage
SPEC Floating Point Benchmarks
Full Caching
Some Caching: ALPHA = 1
No Caching
spice
doduc
mdljdp
wave
tomcatv
ora
alvinn
ear
mdljsp
swm
su2co
hydro2d
nasa7
fpppp

39

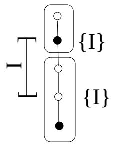


Comparison with factoring:

- Factoring attempts to reduce size of CDG by making nodes 'share' control dependences in the representation (CFS 90)



- Our caching approach can be viewed as factoring in which 'filtered search' is used to answer queries (Chazelle)



Other Applications of APT

Control Dependence		Dataflow Analysis
CONDS	iterate →	SSA,GSA
CDEQUIV	iterate →	DFG,PDW,VDG,....
CD		

ADT : augmented dominator tree (APT on reverse CFG)

ADT and APT

- can be used to build SSA form in O(|E|) per variable
  - subsumes algorithm of Cytron et al (α <<)
  - subsumes algorithm of Sreedhar and Gao (α >>)

- can be used to build DFG in O(|E|) time per variable
  - SESE determination in O(|E|) time
  - see Johnson, Pearson, Pingali (PLDI 94)
    Johnson's thesis at Cornell

SSA Computation

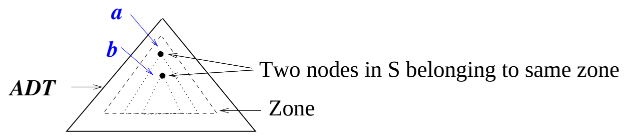
- phi-placement = iterated dominance frontier computation

- exploit the fact that conds relation is same as edge dominance frontier relation in reverse graph

Solution: Use APT on reverse graph = ADT on CFG

- First, look at DF(S) where S is given offline

  Algorithm: Sort S by level, and query in bottom-up order



- to compute DF(b), visit sub-zone below b
- after this, to compute DF(a), no need to visit subzone below a !

Algorithm:

- Sort nodes in S by level.
- Remove nodes from sorted list by decreasing level order, and query in ***ADT***
- After a node is queried, mark it in ***ADT*** so further queries that reach v do not look below v.

Time = O(|V| + |A|) (O|E|) in CFG terms

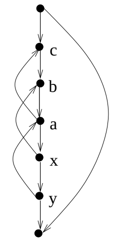
What if set for querying is given online?

- We can use same strategy provided nodes are presented for querying in bottom-up order.
- Happily, if n is in DF(m), then level(n) <= level(m) !!

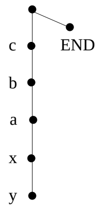
=> use a priority queue for 'dynamic sorting'

- Priority queue implementation: (k = # of keys = height of ***ADT***)
  - van Emde Boas: O(log(log(k))) per insertion and deletion
  - Sreedhar and Gao: use an array of size k

Example:



CFG

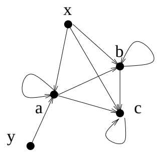


Dominator tree

E \ V	a	b	c	x	y
y -> a	√			√	√
x -> b	√	√		√	
a -> c	√	√	√		
y -> END	√	√	√	√	√

DF(node) = destination(EDF(node))
DF({a}) = {a,b,c,END})
DF({b}) = {b,c,END}
DF({c}) = {c,END}

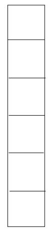
EDF



phi({a,x}) = {a,b,c}

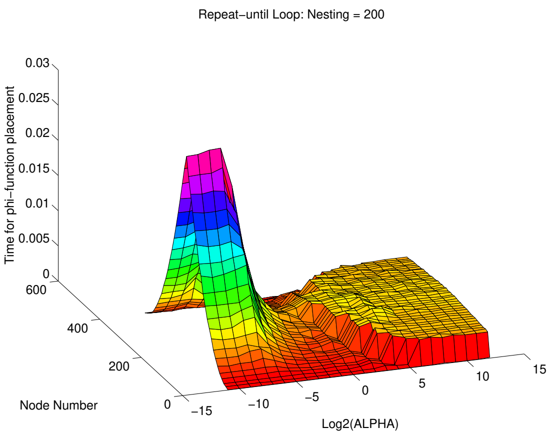
phi({c}) = {c}

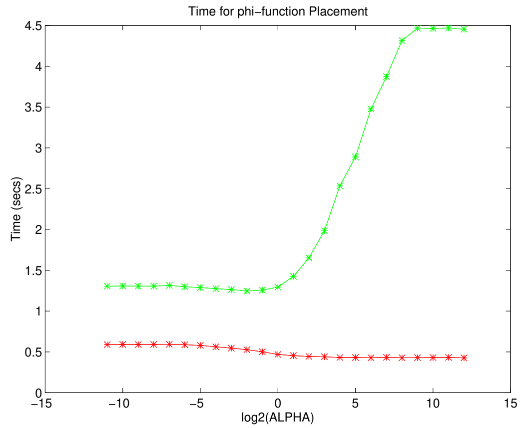
Dominance Frontier



Remarks:

- Time to build SSA form: O(|E|) per variable
- Subsumes algorithms of Cytron etal and Sreedhar and Gao
  - α << : Cytron et al [91] - O(|E|*|V|) per variable
  - α >> : Sreedhar and Gao (PLDI 95) - O(|E|) per variable
- Same idea can be used to build sparse dataflow evaluator graphs for other dataflow problems
- What is best value of α ? Interesting tradeoff
  - small value: repeatedly discover that some node is in transitive closure
  - large value: time to compute individual DF sets may be large
  - intermediate value may be best!





Conclusions

1. APT data structure:

Query time: (α+1) * output-size
Preprocessing Space and Time: O(|E| + |V| / α)

Control Dependence

CONDS (v): optimal
CDEQUIV(v): optimal
CD(e): optimal

Dataflow Analysis

SSA: O(|E|) per variable
SDEG: O(|E|) per problem
DFG: O(|E|) per variable

2. Key concepts

- exploit structure of control dependence relation
- intelligent caching of information

Applications of Technology

- *DCPI: Digital Continuous Profiling Infrastructure* uses control dependence equivalence algorithm to reduce overhead of program profiling http://www.research.digital.com/SRC/dcpi/
- *IBM VLIW Compiler*: Ebcioglu et al use Dependence Flow Graph (DFG) as their IF in VLIW compiler work http://www.research.ibm.com/vliw/
- *Aristotle Analysis System*: Ohio State University // uses weak control dependence algorithms
- *Toby compiler (IBM), Intel,...*: use some of the control dependence algorithms