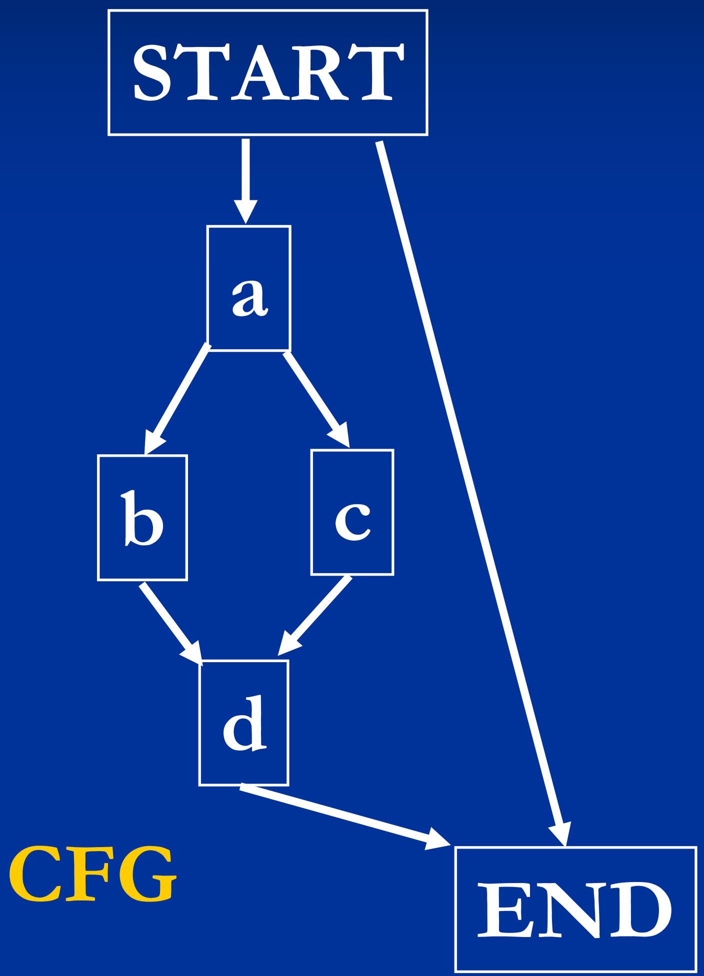
Control Dependence

Control Dependence

- *Introduction*
- Formal Definition
- Optimal Control Dependence Computation

Example



- *b* is control dependent on *a*
- *c* is control dependent on *a*
- *a* and *d* are control dependent on *START*

Applications of CD

- Dead code elimination
- Scheduling (hyper-block formation)
- Predication
- …

Simple Dead Code Elimination

- Mark inherently live statement live
  - Store to memory, print, …
- For each variable in these live statements, mark its definition statement live.
- For each live statement, mark it live the node that it is control dependent on.
- Remove everything that is not marked.

Example

- if (x > 0) {
-     printf(“greater than zero”);
- }

- The printf statement is inherently live. You also need to mark the “if (x>0)” live because the ‘print’ statement is control dependent on the ‘if’.

Control Dependence

- Introduction
- *Formal Definition*
- Optimal Control Dependence Computation

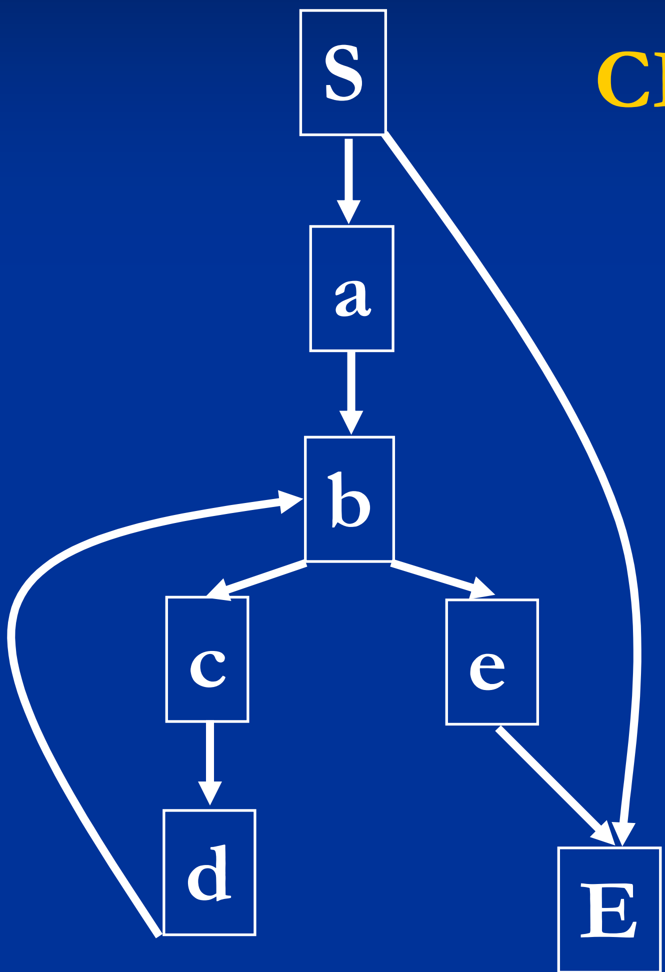
Post-dominator Relation

- If X appears on every path from START to Y, then X ***dominates*** Y.
- If X appears on every path from Y to END, then X ***postdominates*** Y.
- Postdominator Tree
  - END is the root
  - Any node Y other than END has ipdom(Y) as its parent
  - Parent, child, ancestor, descendant

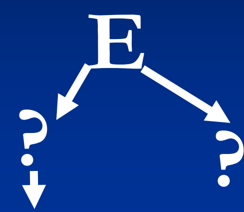
Control Dependence Relation

- There are two possible definitions.
- Node w is control dependent on edge (u→v) if
  - w postdominates v
  - If w ≠ u, w does not postdominate u
- Node w is control dependent on node u if there exists an edge u→v
  - w postdominates v
  - If w ≠ u, w does not postdominate u

Example



CFG

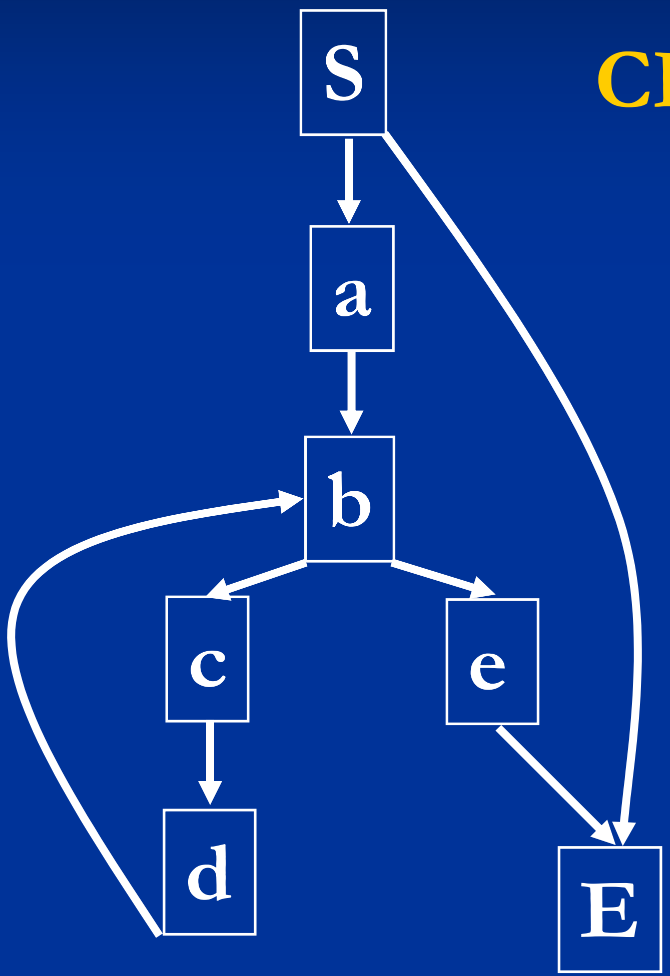


Pdom Tree

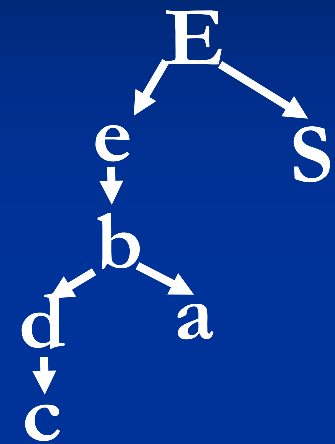
	a	b	c	d	e
S→a					
b→c					

Control Dep Relation

Example



CFG

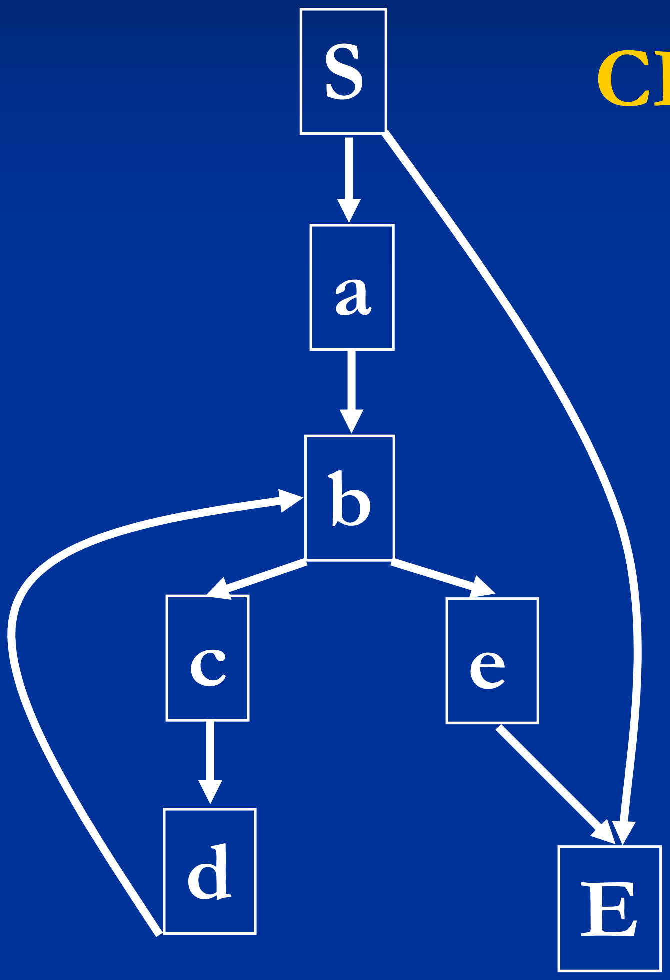


Pdom Tree

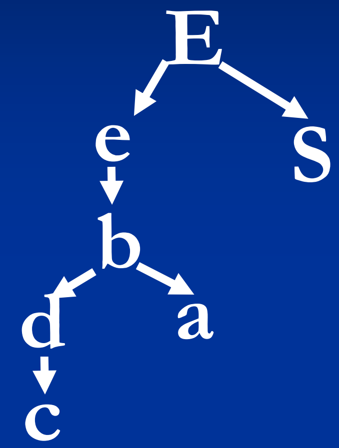
	a	b	c	d	e
S→a					
b→c					

Control Dep Relation

Example



CFG



Pdom Tree

	a	b	c	d	e
S→a	√	√			√
b→c		√	√	√	

Control Dep Relation

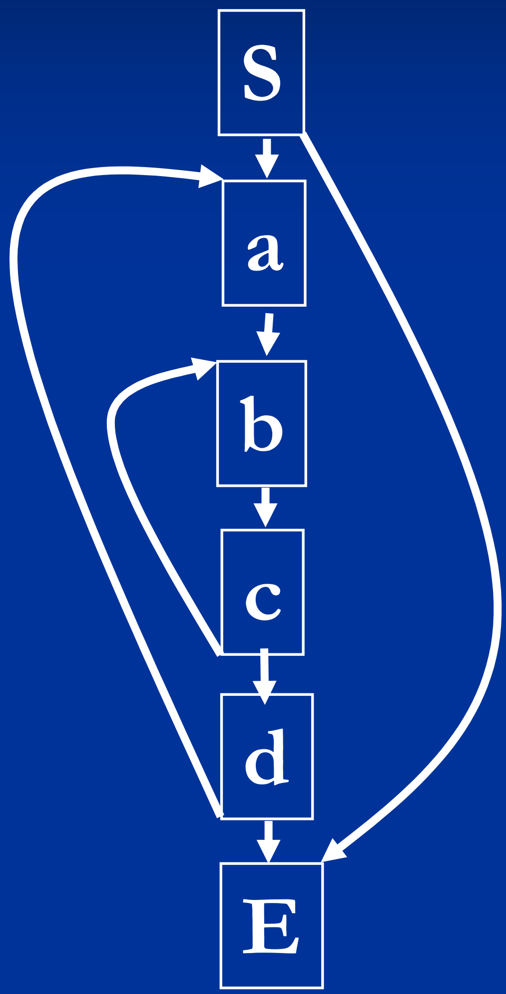
Control Dependence Queries

- CD(e): set of nodes control dependent on edge e
- CONDS(v): set of edges that node v is control dependent on

Dominance Frontier

- Reverse control flow graph (RCFG)
- Let X and Y be nodes in CFG. X in DF(Y) in CFG iff Y is control dependent on X in RCFG.
- DF(Y) in CFG = conds(Y) in RCFG, where conds(Y) is the set of nodes that Y is control dependent on.

Worst-case Size of CDR

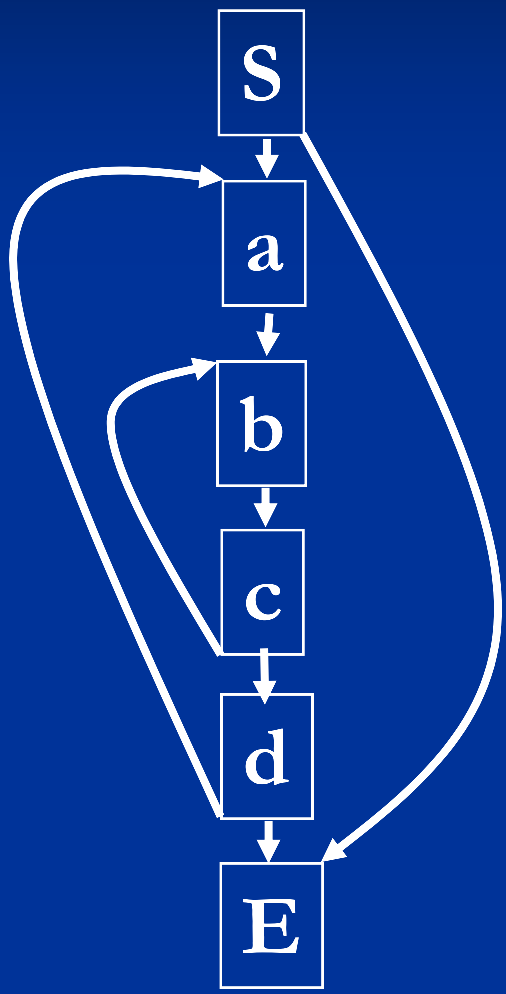


CFG

	a	b	c	d
S→a				
d→a				
c→b				

Control Dependence Relation

Worst-case Size of CDR



CFG

	a	b	c	d
S→a	√	√	√	√
d→a	√	√	√	√
c→b		√	√	

Control Dependence Relation

Size = $O(n^2)$

Control Dependence

- Introduction
- Formal Definition
- *Optimal Linear Control Dependence Computation*

APT

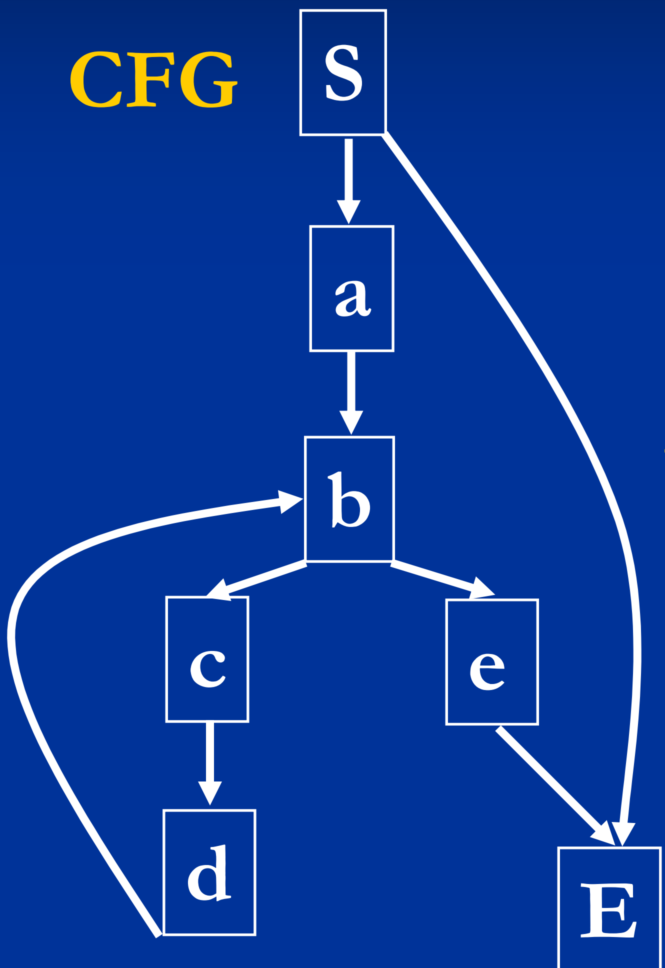
- APT: Augmented Postdominator Tree
  - which can be built in O(|E|) space and time
  - which can be used to answer CD and CONDS queries in time proportional to output size
- Optimal control dependence computation
- Solution: reduced control computation to a graph problem called **Roman Chariots Problem**

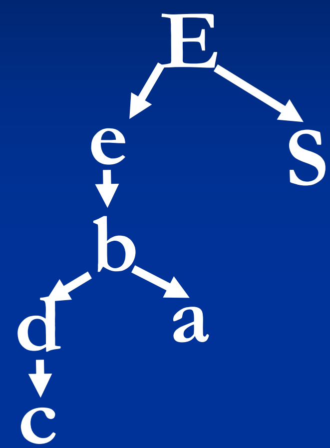
Key Idea (I): Exploit Structure

- How to avoid building the entire control dependence relation ($O(n^2)$)?
  - 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.
- Pdom tree + endpoints of each control dependence path can be built in $O(|E|)$ space and time

Example

CFG





Pdom Tree

E	path
S→a	?
b→c	?

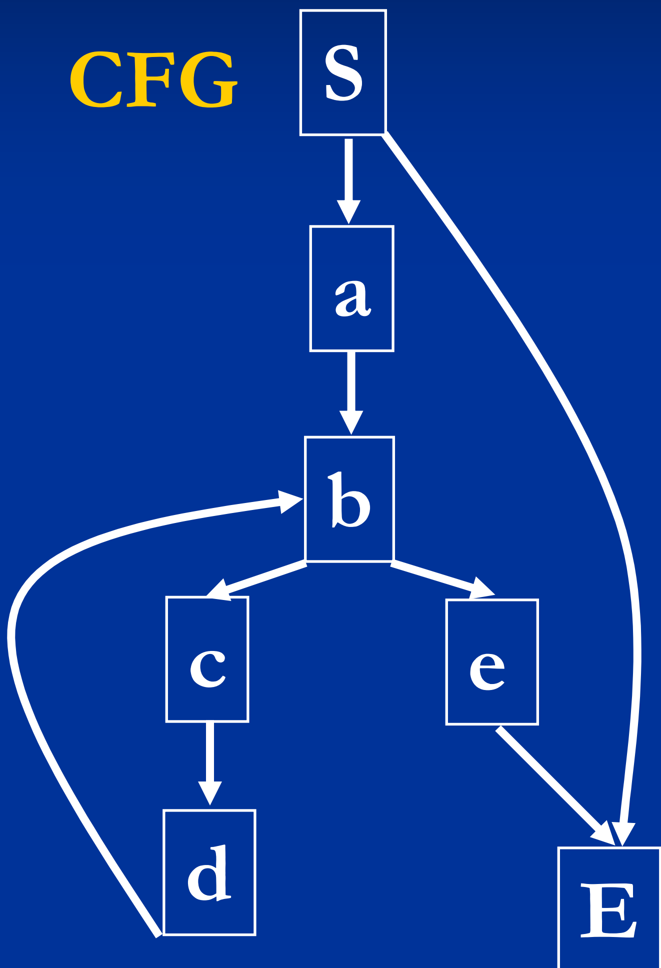
Path Array

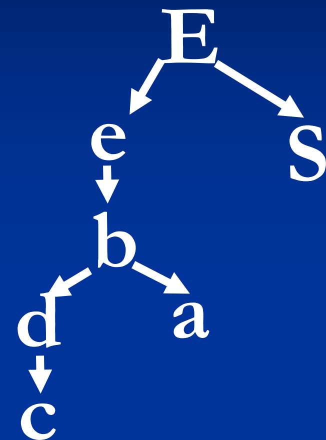
	a	b	c	d	e
S→a	√	√			√
b→c		√	√	√	

Control Dep Relation

Example

CFG





Pdom Tree

E	path
S→a	[a,e]
b→c	[c,b]

Path Array

	a	b	c	d	e
S→a	√	√			√
b→c		√	√	√	

Control Dep Relation

CD Queries

- How can we use the compact representation of the CDR (Control Dependence Relation) to answer queries for CD and CONDS sets in time proportional to output size?

Roman Chariots Problem

Route #	path
I	[Milano,Roma]
II	[Pompeii,Bologna]
III	[Venezia,Roma]

- CD(n): which cities are served by chariot n?
- CONDS(w): which chariots serve city w?

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

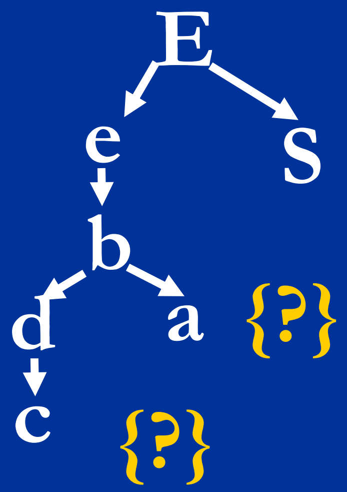
- Look up entry for chariot n in Route Array (say [x,y])
- Traverse nodes in tree T, starting at x and ending at y
- Output all nodes encountered in traversal
- Query time is proportional to output size

CONDS(w): which chariots serve city w?

- 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;
- Can we avoid examining all routes?

Key Idea (II): Caching

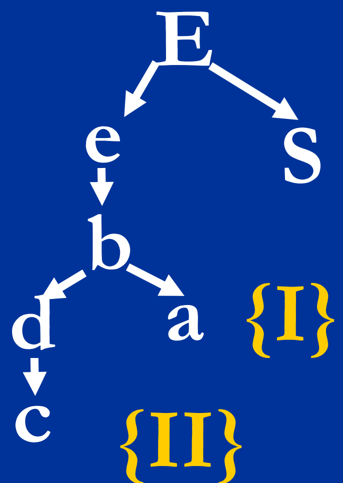
- At each node in the tree, keep a list of chariot #'s whose bottom node is n.



Chariot #	route
I	[a,e]
II	[c,b]

Key Idea (II): Caching

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



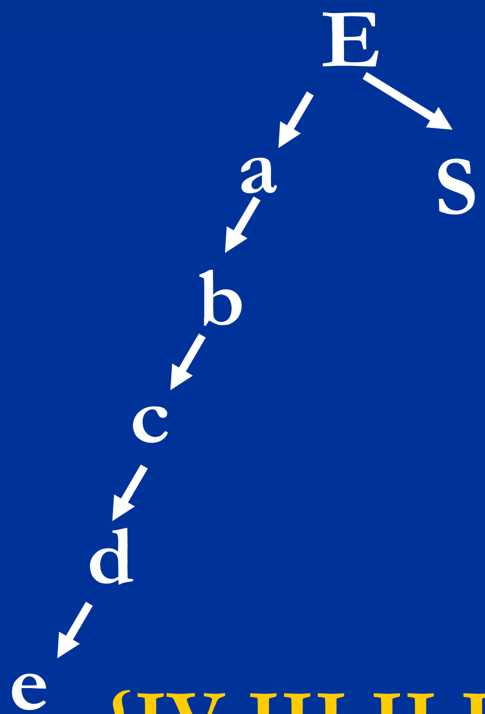
Chariot #	route
I	[a,e]
II	[c,b]

CONDS(w): which chariots serve city 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;
- Query time is proportional to # of descendants + size of all lists at d

Sorting Lists

- Sort each list by decreasing length



{IV,III,II,I}

Chariot #	route
I	[e,d]
II	[e,c]
III	[e,b]
IV	[e,a]

CONDS(w): which chariots serve city 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
- Query time is proportional to size of output + # of descendants

Caching at All Nodes on Route

- Sort each list by decreasing length

E
a
b
c
d
e {IV,III,II,I}

E
a {IV}
b {IV,III}
c {IV,III,II}
d {IV,III,II,I}
e {IV,III,II,I}

Space Time Tradeoff

- Chariot # stored only at bottom node of the route
  - Space: O(|V| + |A|)
  - Query Time: O(|V| + |Output|)
- Chariot # stored at all nodes on route
  - Space: O(|V| * |A|)
  - Query Time: O(|Output|)
- V is the set of tree nodes, and A is the Route Array.

Key idea (III): Caching Zones

- Divide tree into ZONES
- Nodes in Zone:
  - Boundary nodes: lowest nodes in zone
  - Interior nodes: all other nodes
- Query procedure:
  - **Visit only nodes below query node and in the same zone as query node**

Caching Rule

- Boundary node: store all chariots serving node
- Interior node: store all chariots whose bottom node is that node
- Algorithm: bottom-up, greedy zone construction
  - Query time: $|A_v| + |Z_v| \leq (\alpha + 1)\ |A_v|$
  - Space requirements $\leq |A| + |V| / \alpha$

Constructing Zones (I)

- Invariant: for any node v, $|Z_v| \le \alpha\, |A_v| + 1$, where α is a design parameter.
- Query time for CONDS(v)

  $= O(|A_v| + |Z_v|)$

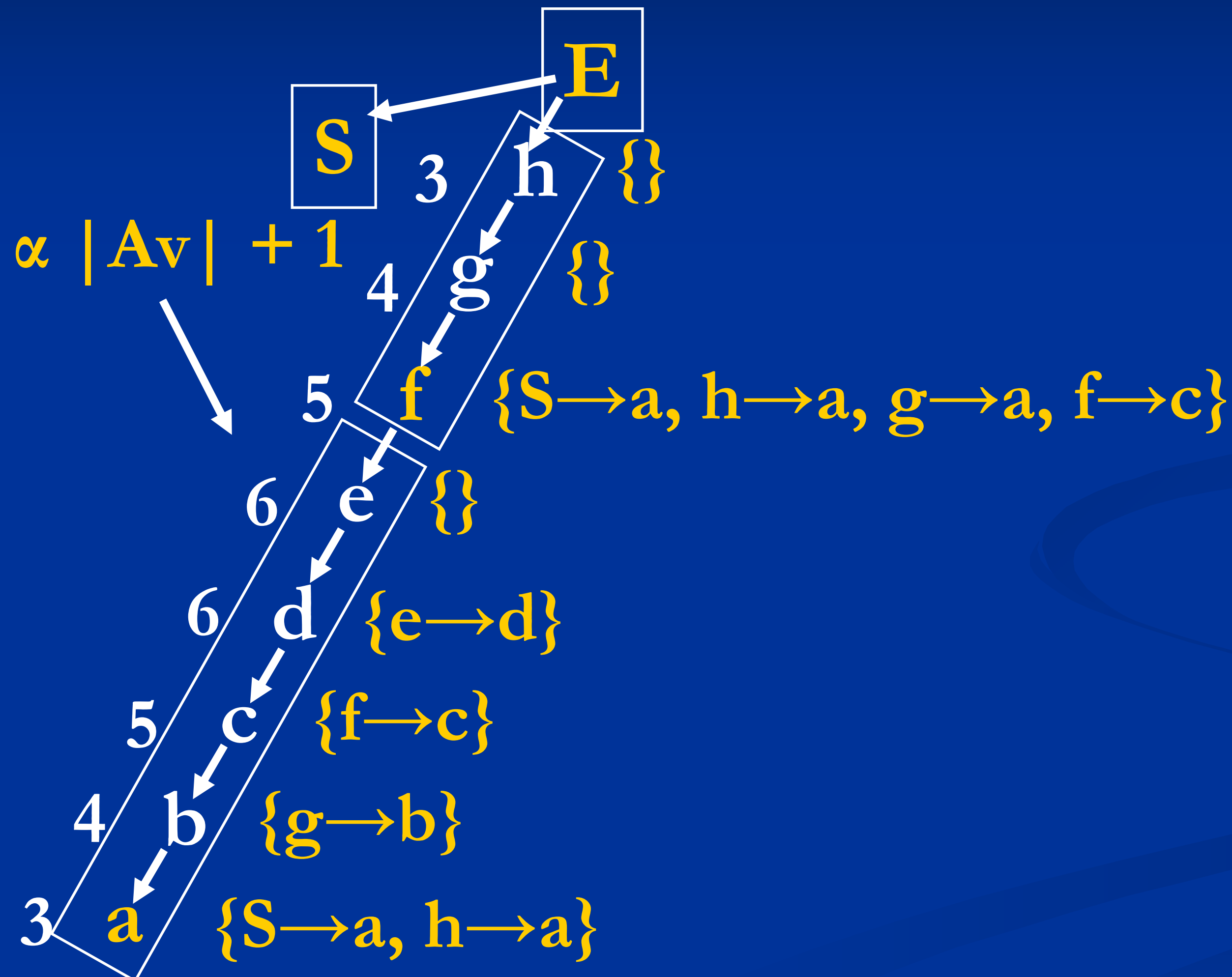
  $= O((\alpha + 1)|A_v| + 1)$

  $= O(|A_v|)$

Constructing Zones (II)

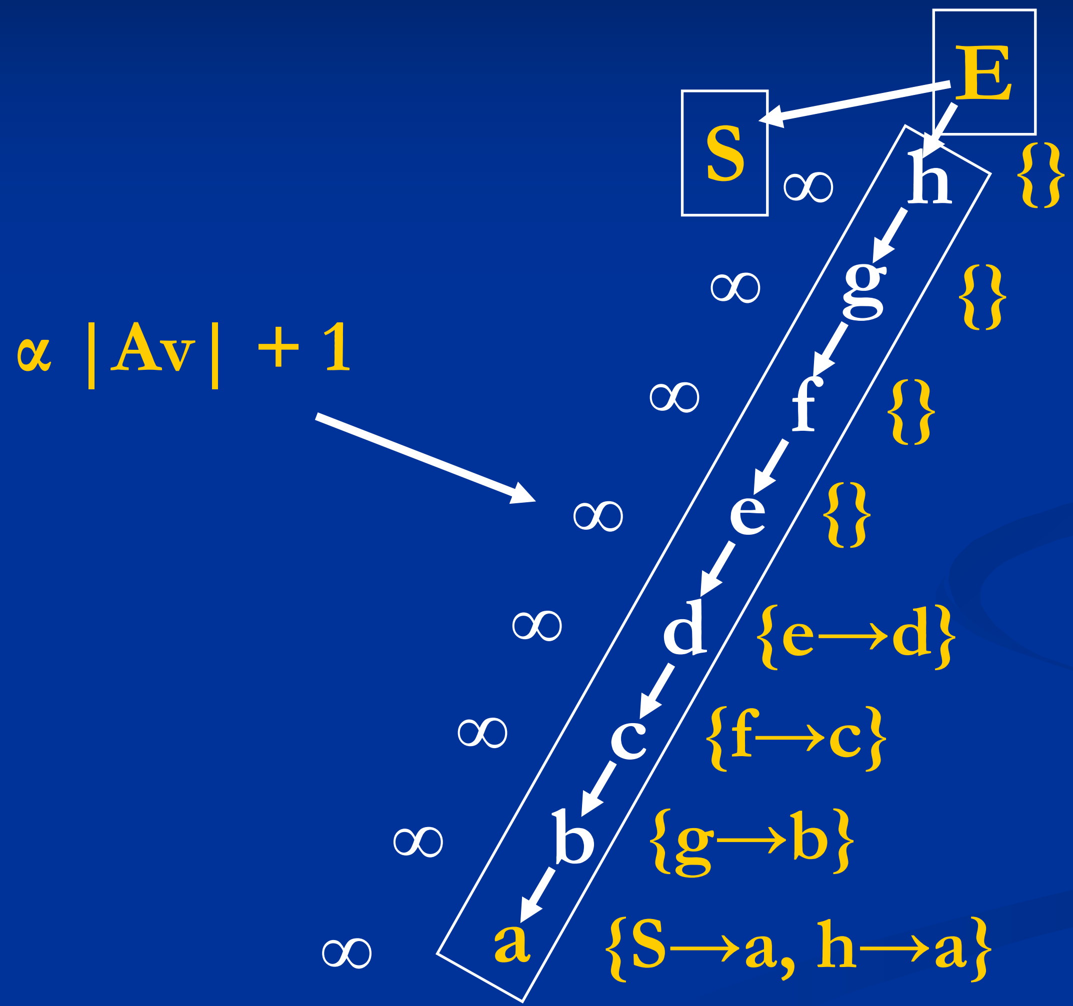
- Build zones bottom-up, making them as large as possible without violating invariant
- V is a leaf node, then make v a boundary node
- V is an interior node then
  - If $(1 + \sum_{u \in children(v)} |Z_u|) > \alpha |A_v| + 1$
    - then make v a boundary node
    - else make v an interior node

α = 1 (some caching)



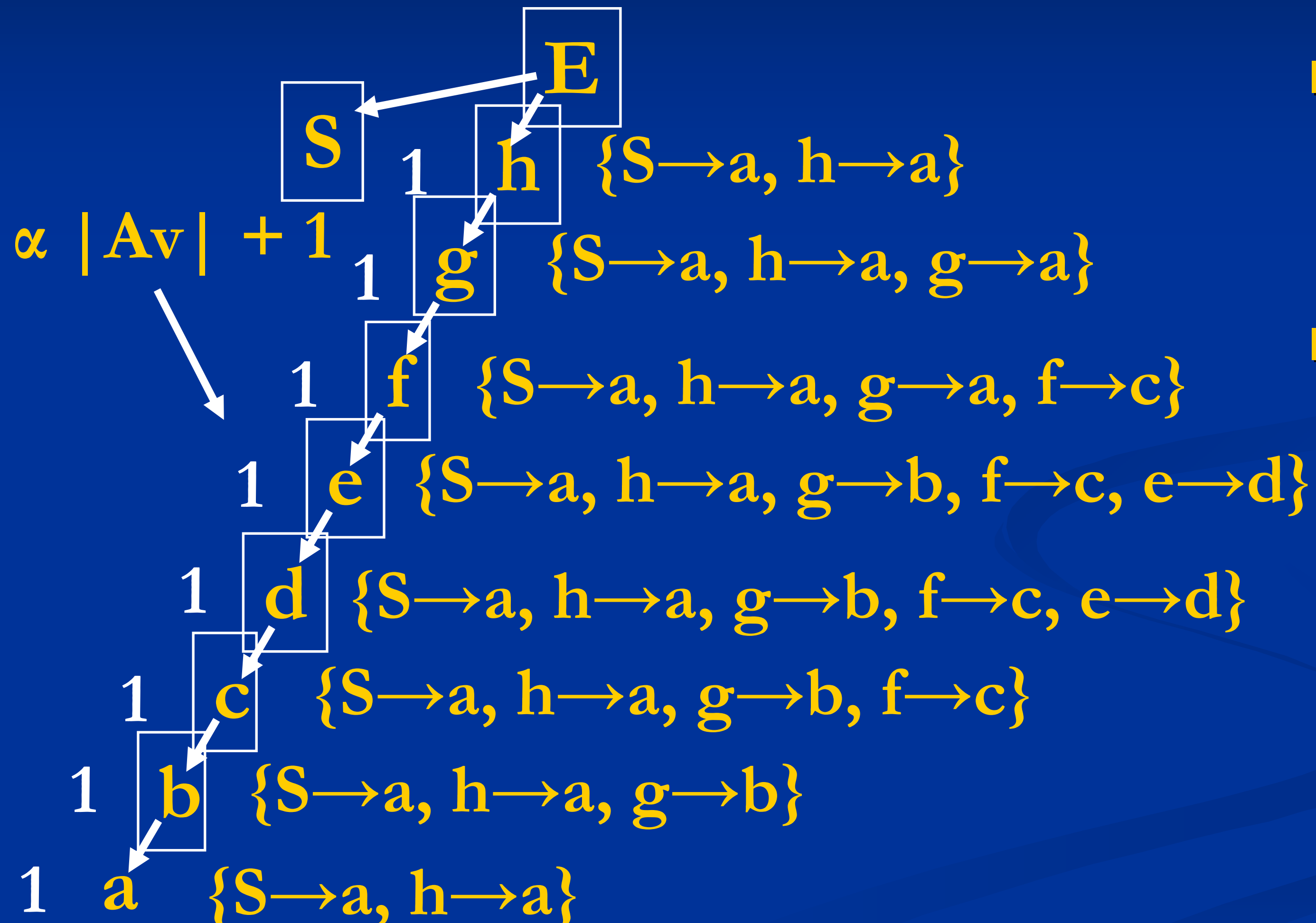
- Zones: {a,b,c,d,e}, {f,g,h}, {E}, {S}
- **Boundary nodes: a, f, E, S**

α = >>(no caching)



- Zones: {a,b,c,d,e,f,g,h}, {E}, {S}
- **Boundary nodes: a, E, S**

α = << (full caching)



- Zones: every node
- **Boundary nodes: every node**

APT (I)

- Postdominator tree with bidirectional edges
- dfs-number[v]: integer
  - Used for ancestorship determination in CONDS query
- Boundary?[v]: boolean
  - True if v is a boundary node, false otherwise
  - Used in CONDS query

APT (II)

- L[v]: list pf 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

Computational Complexity

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

Reference

- "Optimal Control Dependence Computation and the Roman Chariots Problem", Keshav Pingali, Gianfranco Bilardi, ACM Transactions on Programming Languages and Systems (TOPLAS), May 1997. http://iss.cs.cornell.edu/Publications/Papers/TOPLAS1997.pdf