Learn to Build Automated Software Analysis Tools with Graph Paradigm and Interactive Visual Framework

GIAN Jaipur, September 12-16, 2016

Suresh C. Kothari
Richardson Professor
Department of Electrical and Computer Engineering

Ben Holland, Iowa State University

Module I: Graph Models to Solve Software Problems

Acknowledgement: **Team members at Iowa State University and EnSoft, DARPA contracts FA8750-12-2-0126 & FA8750-15-2-0080**

Module Outline

- Efficient debugging requires a graph model
- *Micro* and *Macro* models
- Micro model examples – (a) backward dataflow slice, (b) the control flow graph (CFG)
- Macro model examples – (a) the call graph (CG), (b) the reverse call graph (RCG).
- Graph models as abstractions for solving software problems

Presenters

- Suresh(Suraj) Kothari
  - Richardson Professor, Iowa State University (ISU)
  - President and Founder, EnSoft
  - Principal Investigator (PI), DARPA Projects Space/Time Analysis for Cyber Security (STAC) and Automated Program Analysis for Cybersecurity (APAC)
- Ben Holland
  - MITRE, Rockwell Collins (Government Systems), Wabtec Railway Electronics
  - Cybersecurity Analyst on the DARPA projects STAC and APAC

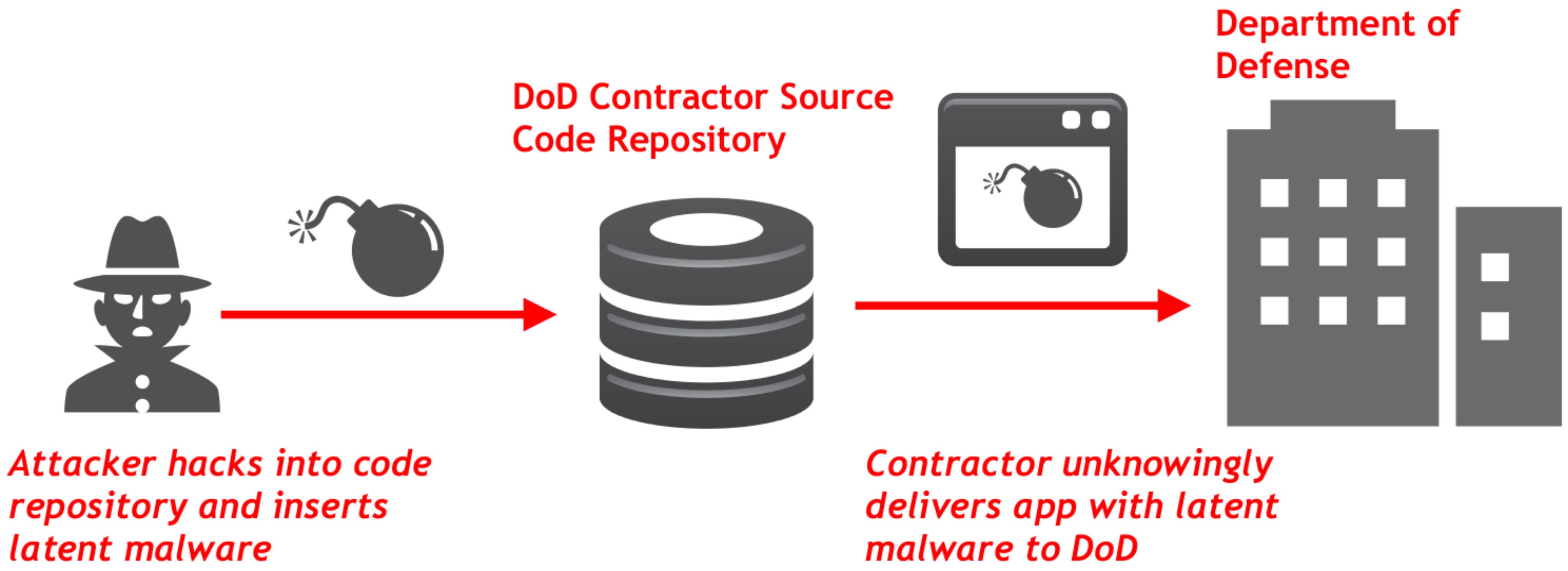
DARPA APAC Project

- **Automated Program Analysis for Cybersecurity (APAC):** Detect sophisticated vulnerabilities in Android apps.
- Requirement: Analyze Java code, the resource and GUI files, and the Android APIs used by the app.
- This project finished in February 2015: ISU-EnSoft the top performing Blue team in Phase I, and among the top 3 teams in Phase II.

DARPA STAC Project

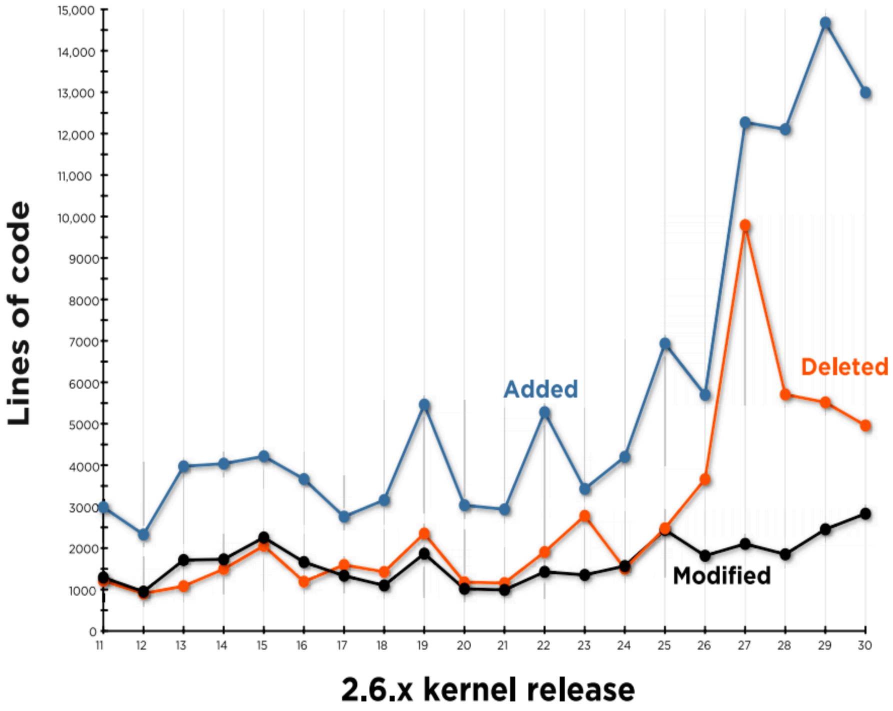
- Space/Time Analysis for Cybersecurity (STAC): attacks use the knowledge of variations in space-time complexities along different execution paths to design denial of service or side channel attacks.
- Requirement: Analyze Java byte code to detect *algorithmic complexity* (AC) and *side channel* (SC) vulnerabilities.

DARPA's Challenge



Hardened devices, untrusted contractors, expert adversaries

Humongous Software



Added LOC/day	Deleted LOC/day	Modified LOC/ day
12993	4958	2830

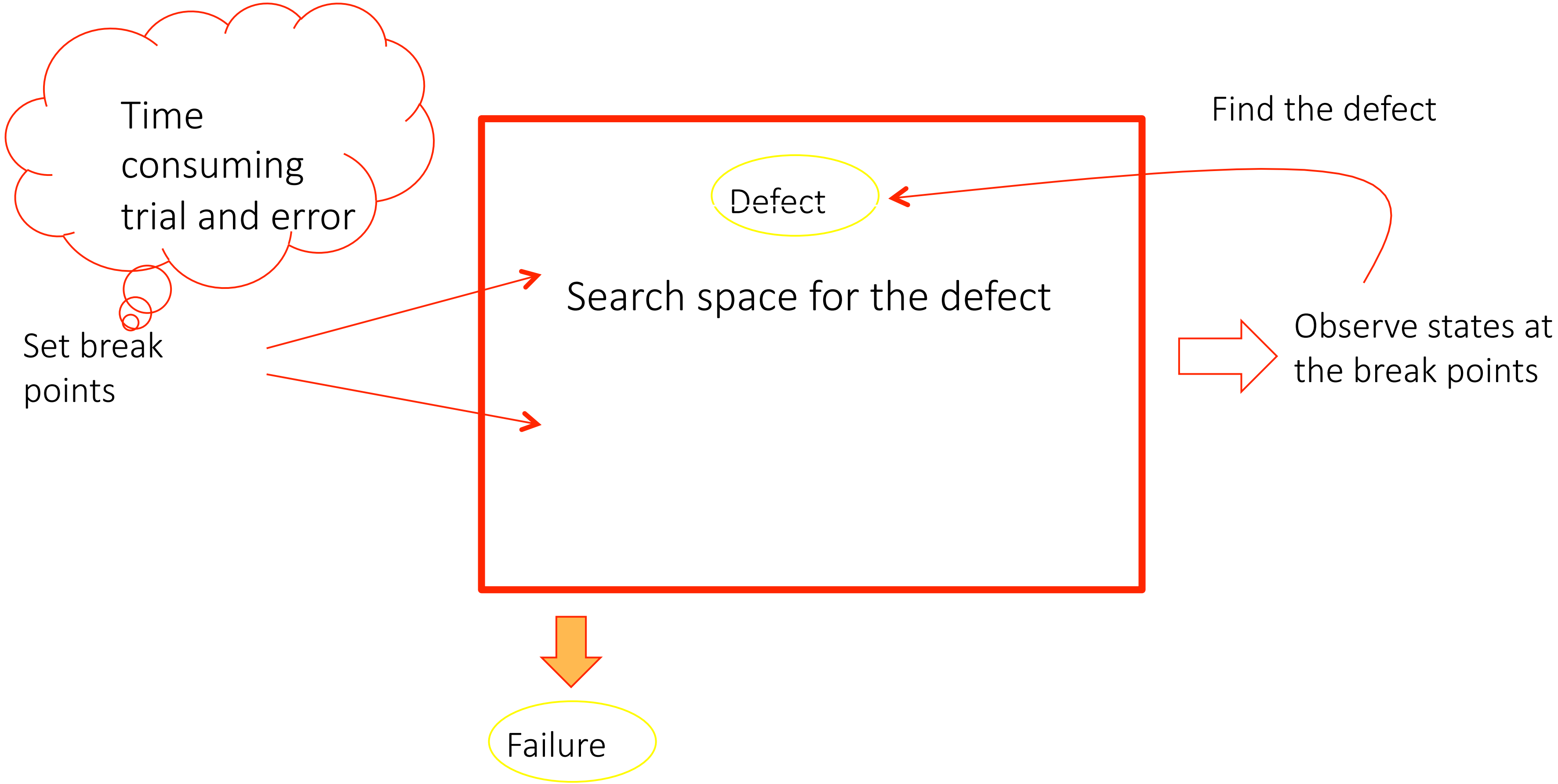
Linux 1.0.0 – 176,250 LOC
Linux 4.0 – 13 M LOC

A printed stack of paper 136 feet high!

Visual Graph Models – What and Why

- What: Extract and abstract the problem-relevant knowledge from humongous software. Have a 2-way correspondence to source.
- Visual models are needed to:
  - reason about large software.
  - make automation computationally scalable and efficient.
  - define hardness of the problem.
  - enable man-machine collaboration for efficient and accurate problem solving.
- We will discuss a graph paradigm to create and refine visual models through interactive and programmable queries using a graph database of program artifacts and relationships.

Efficient Debugging

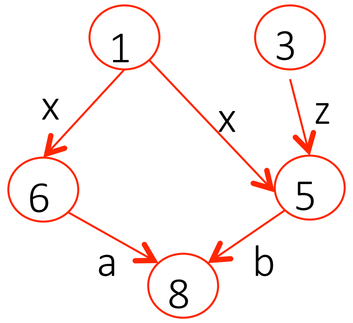


A graph model for debugging

1. x = 2;
2. y = 3;
3. z = 7;
4. a = x + y;
5. b = x + z;
6. a = 2 * x;
7. c = y + x + z;
8. t = a + b; ← detected failure
9. Print t;

Relevant lines:

1,3,5,6,8



The graph abstraction captures all that is needed, no more no less.

This graph model, called the *backward dataflow slice*, was introduced by Mark Weiser in early 1980's.

Use-Def (UD) Chain

The *backward dataflow slice* is constructed by applying UD chains.

1. x = 2;
2. y = 3;
3. z = 7;
4. a = x + y;
5. b = x + z;
6. a = 2 * x;
7. c = y + x + z;
8. t = a + b;
9. Print t;

Statement 8 *defines* **t** and *uses* **a** and **b**

Equivalently, *write-set*(8) = {**t**} and *read-set*(8) = {**a**, **b**}

A *UD chain* consists of a use of a variable, and *all the definitions* of that variable that can reach that use.

Statement 4 and 6 provide definitions of the variable **a**.

The definition 6 reaches the use of **a** at statement 8

The definition 4 is *killed* by the definition 6, thus it *cannot* reach the use at 8.

How can we have multiple definitions reaching the same use?

What makes software problems hard?

- *Global interactions* across functions get complicated because of the varied ways the functions communicate with each other.
- *Local interactions* within a function get complicated because of many paths, and complex data flows.

Global Hardness

- Relevant functions – the functions necessary and sufficient to solve a problem (e.g. check if each memory allocation is followed by a deallocation).
- Butterfly Effect – A small change in one function causes unforeseen effects in a far away function.
- How do we find relevant functions? Answer: Macro Models

Communication Mechanisms

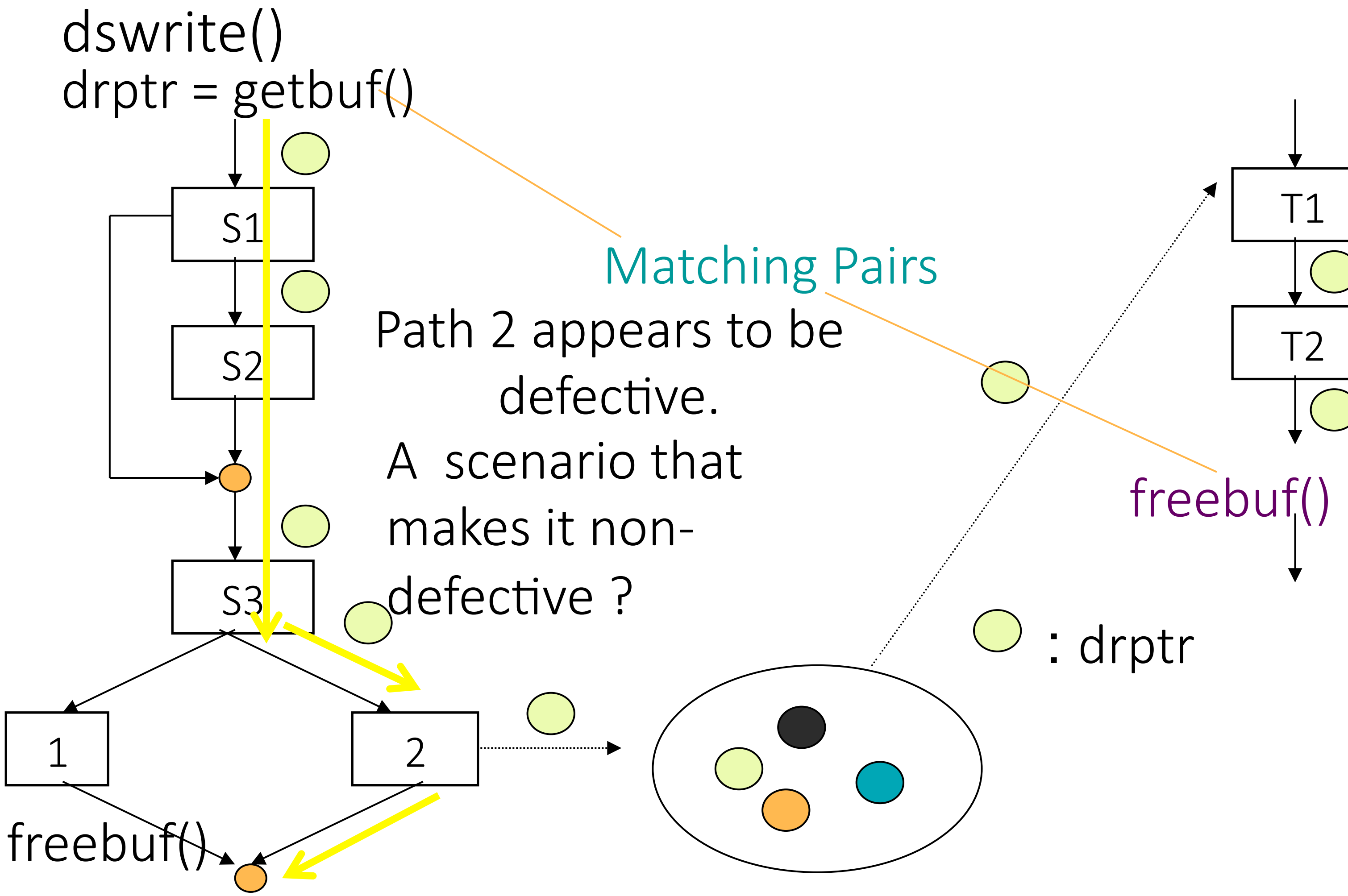
- Relevant data: Data (`D`) relevant to a problem instance (e.g. pointers to an allocated memory).
- Fundamental mechanisms of data flow:
  - `f` passes `D` as a parameter to a callee function `g`.
  - `f` passes `D` as a parameter or return to a caller function `g`.
  - `f` and `g` share `D` through a global variable.

Harder

- Fundamental mechanisms of control flow:
  - `f` calls `g` directly.
  - `f` calls `g` indirectly (e.g. using a function pointer).
  - `f` and `g` operate asynchronously (control transfer happens through interrupts or context switches).

Harder

An illustration of global hardness



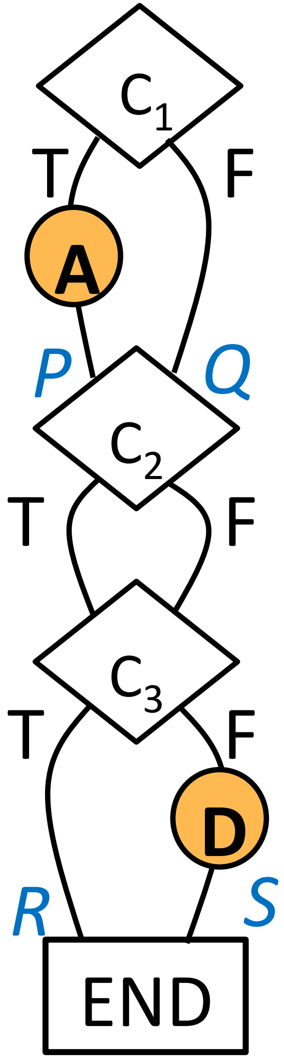
Path 2 appears to be defective.
A scenario that makes it non-defective ?

execution path *invisible* (not reachable through control flow) from the function where memory is allocated.

Local Hardness

- Exponentially many control flow paths – n non-nested If conditions create 2^n paths.
- Satisfiability of branch conditions – a control flow path may not be feasible
- Complex data flows – especially through pointers
- How do we address the local hardness? Answer: Micro Models

An illustration of local hardness



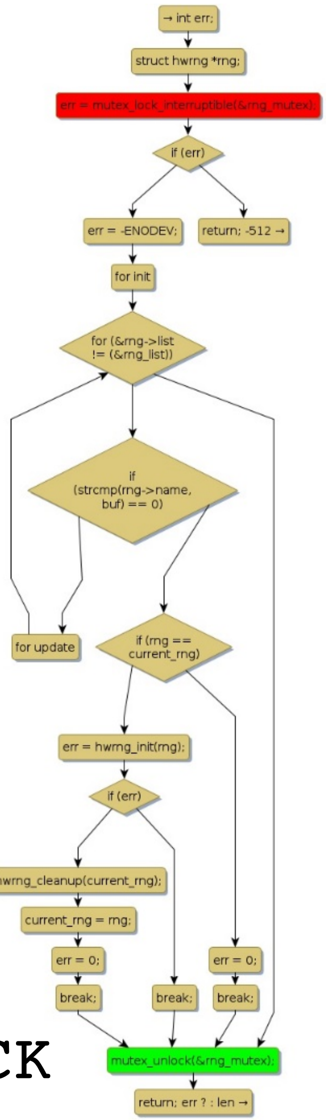
8 paths due to three non-nested branch nodes

The P-R path is *not* feasible if the condition ((c1==T) and (c3==T)) is *not* satisfiable.

Note that A is followed by D on all feasible paths if the P-R path is *not* feasible.

Micro Model: Control Flow Graph (CFG)

LOCK



UNLOCK

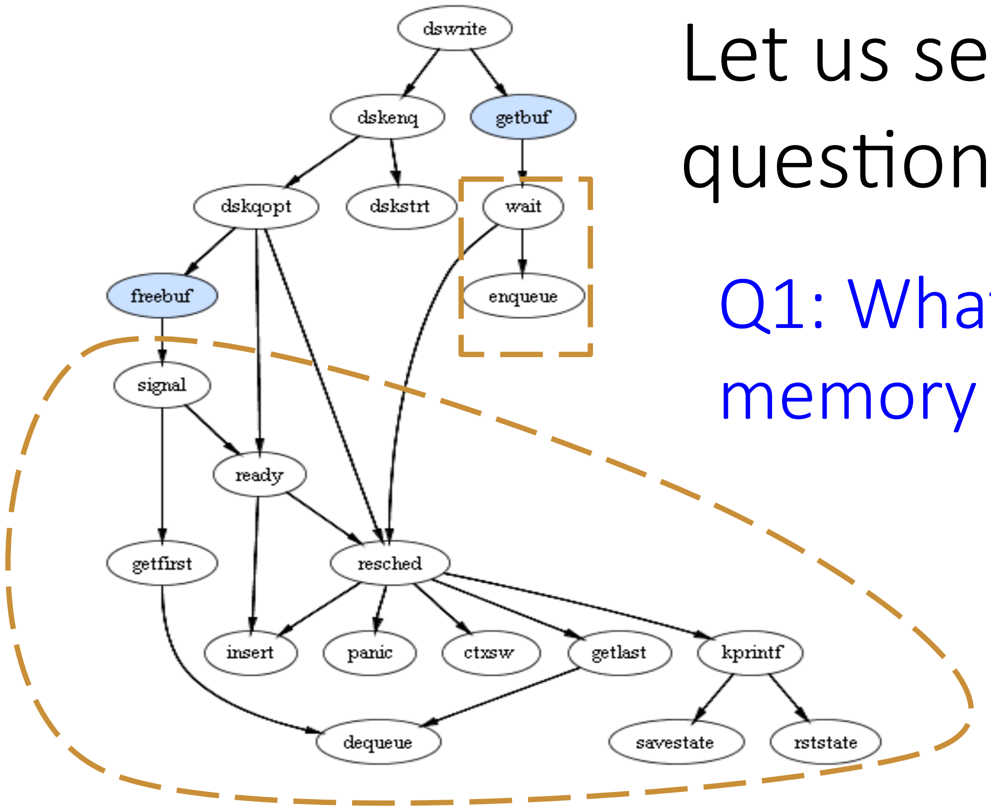
Let us see how the CFG is useful. Here are some questions to illustrate the use of CFG.

Q1: Does the program have a loop?

Q2: Does the loop have a `break`?

Q3: Is the LOCK followed by UNLOCK on all paths?

Macro Model: Call Graph (CG)

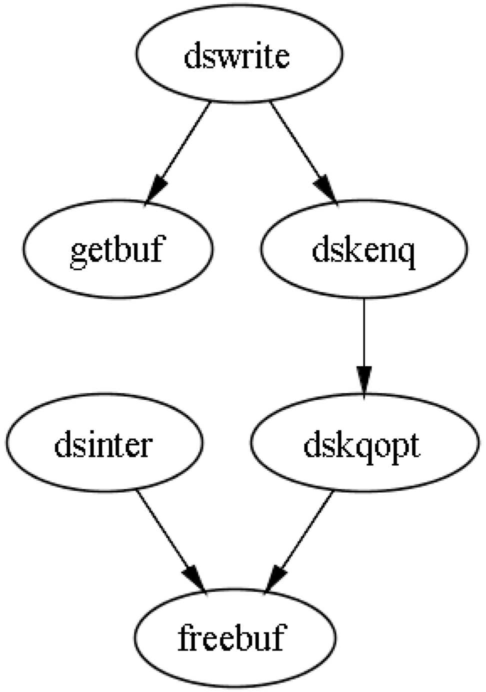


Let us see how the CG is useful. Here is question to illustrate the use of CG.

Q1: What could be the function that releases memory allocated in `dswrite`?

Overarching Question: **`getbuf`** and **`freebuf`** are respectively the calls to *allocate* and *deallocate* memory. The function **`dswrite`** allocates memory by calling **`getbuf`**, but it does not directly release it by calling **`freebuf`**. Is the memory released by another function that interacts with **`dswrite`**.

Macro Model: Reverse Call Graph (RCG)



Let us see how the RCG is useful. Here is question to illustrate the use of RCG.

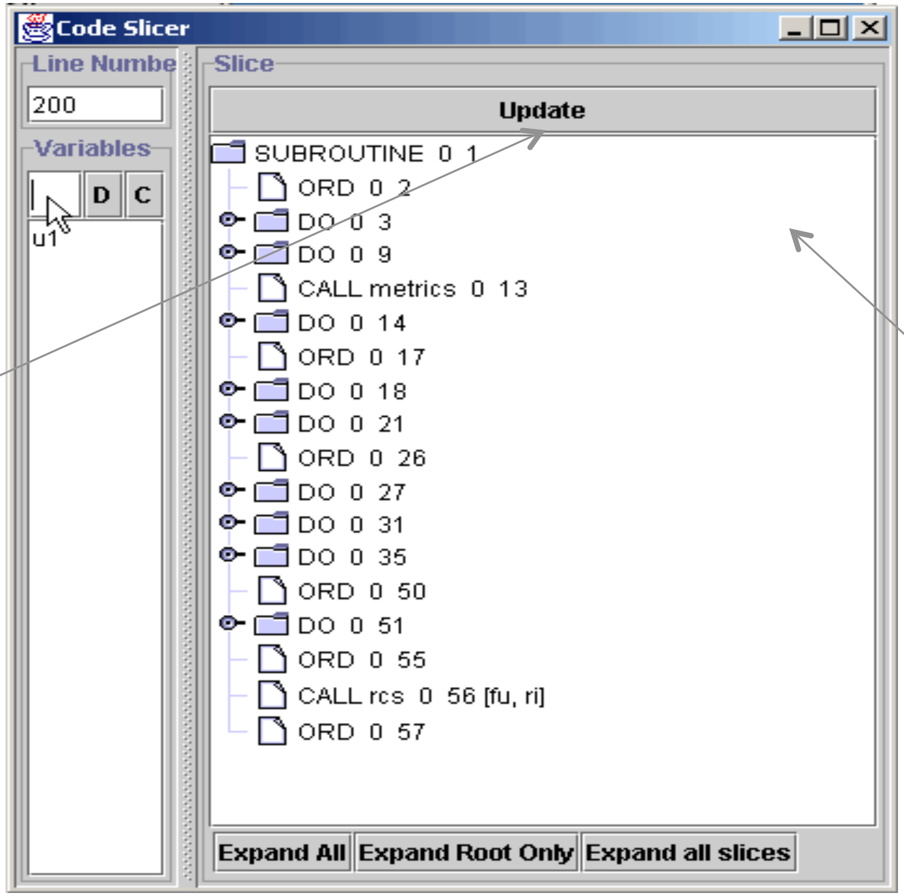
Q1: What could be the function that releases memory allocated in **`dswrite`**?

Q2: How could **`dsinter`** get the pointer to allocated memory if it is not called by **`dswrite`**?

Overarching Question: `getbuf` and `freebuf` are respectively the calls to *allocate* and *deallocate* memory. The function `dswrite` allocates memory by calling `getbuf`, but it does not directly release it by calling `freebuf`. Is the memory released by another function that interacts with `dswrite`.

Creating *backward dataflow slices* with the ParAgent tool

Specify line number, variable, and click the Update bar

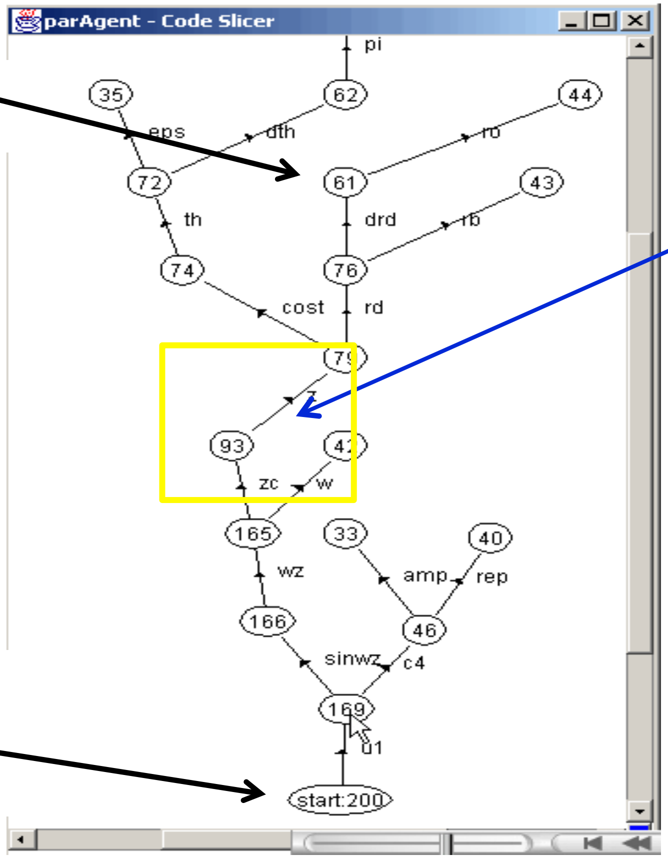


The control structure of the code: the DO blocks, ordinary blocks (no branches), and CALLs.

The nested control structure can be viewed by clicking the Expand button..

Optimal break points using the backward slice

defect



Set the break point here.

detected failure

When software is viewed as lines of code, the visibility limited to a small neighborhood of a node. As a result, debugging is inefficient.

Graph algorithms can be leveraged to find optimal break points.

Backbone operation on slices

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

Backbone = intersection of two slices

Backward slice of sum

Backward slice of mul

Dice operation on slices

```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

Dice = difference of two slices

Backward slice of sum

Backward slice of mul

Chop operation on slices

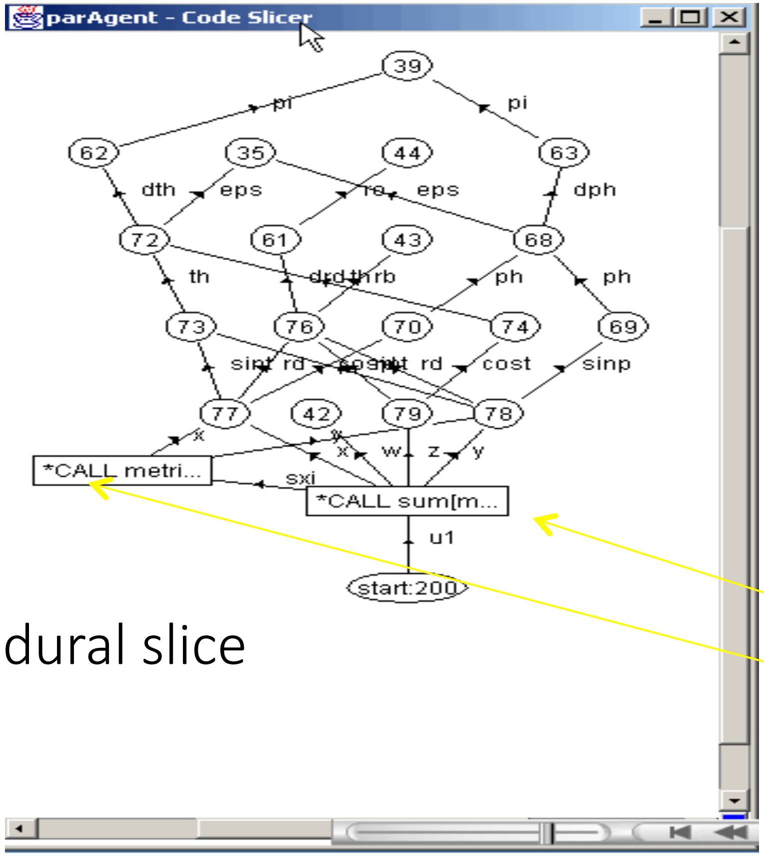
```
int main() {
  int a, b, sum, mul;
  sum = 0;
  mul = 1;
  a = read();
  b = read();
  while (a <= b) {
    sum = sum + a;
    mul = mul * a;
    a = a + 1;
  }
  write(sum);
  write(mul);
}
```

Chop = Intersection of backward and forward slices

Forward slice of b

Backward slice of mul

Variants of slices to address complex debugging



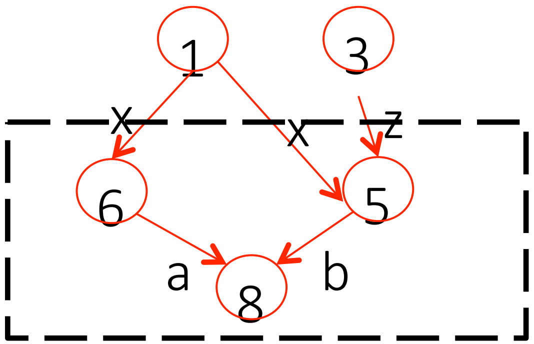
Where to set break points?

Interprocedural slice

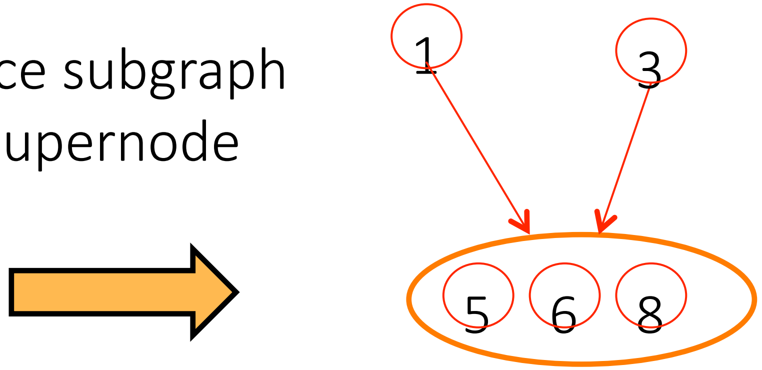
Involves two subroutine calls

An idea: use a transform to simplify the graph

1. x = 2;
2. y = 3;
3. z = 7;
4. a = x + y;
5. b = x + z;
6. a = 2 * x;
7. c = y + x + z;
8. t = a + b;
9. Print t;



Replace subgraph with supernode



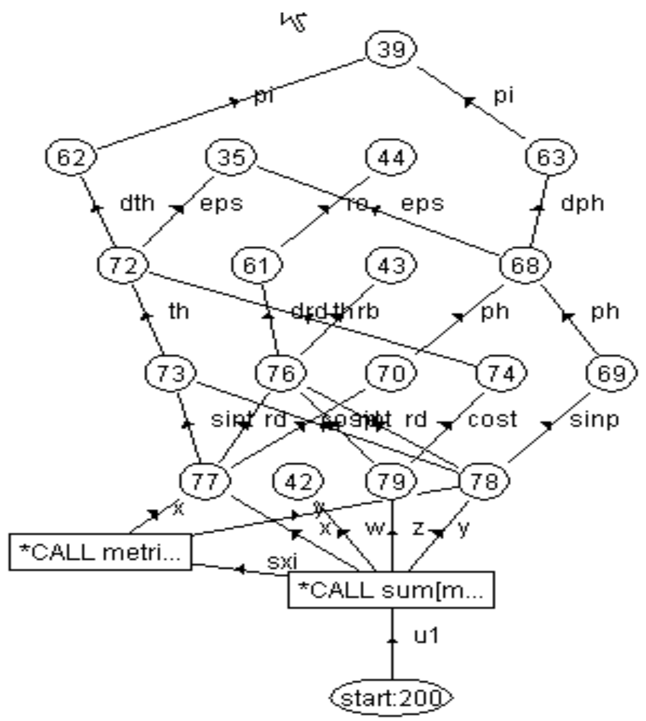
What would be meaningful ways to transform the backward data flow for the purpose of debugging? Which subgraphs to replace?

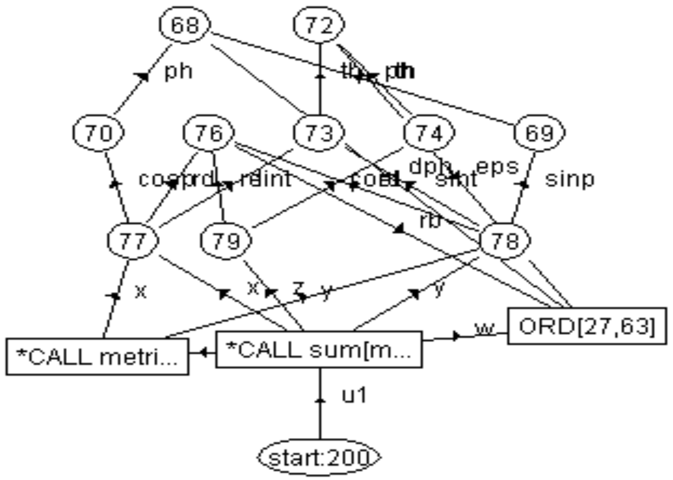
A roadmap idea

- We view maps at different levels of granularity –major highways and the local streets.
- We first chart the highway route and then the local streets.

Can we can do something similar, view the slice at different levels of granularity?

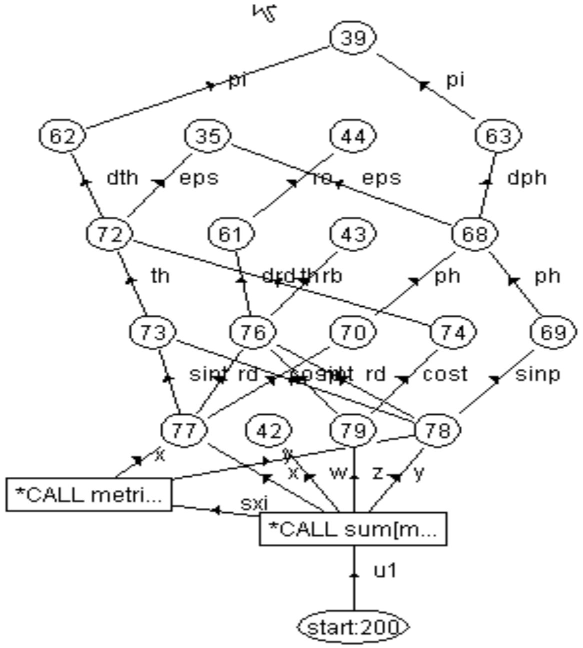
Using the control blocks as nodes



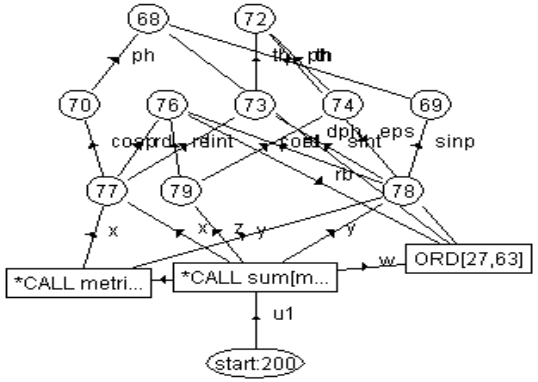


A transformation with one super-node for the selected control bock. The nodes for the statements from 27 to 63 are replaced by one super-node.

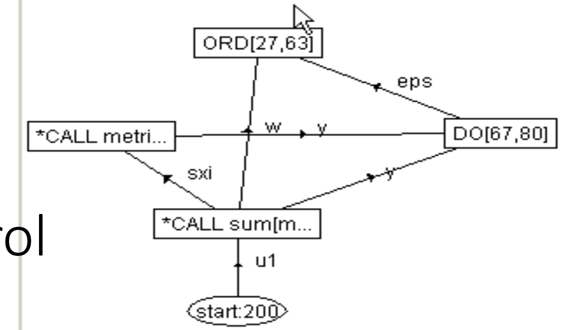
A transformed slice for debugging



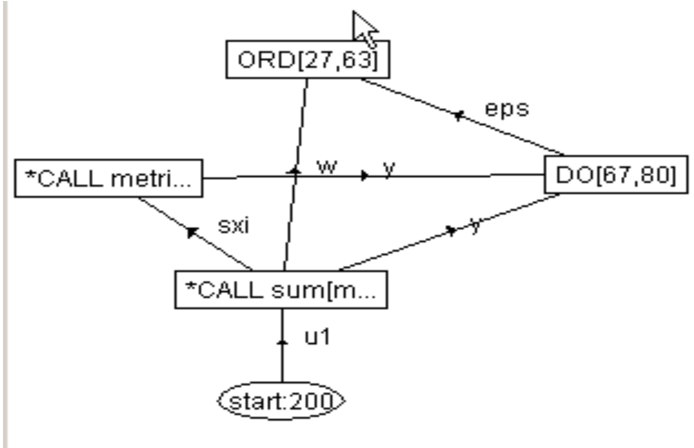
The graph with statements as nodes



The graph with control blocks as nodes



Efficient debugging with the transformed graph

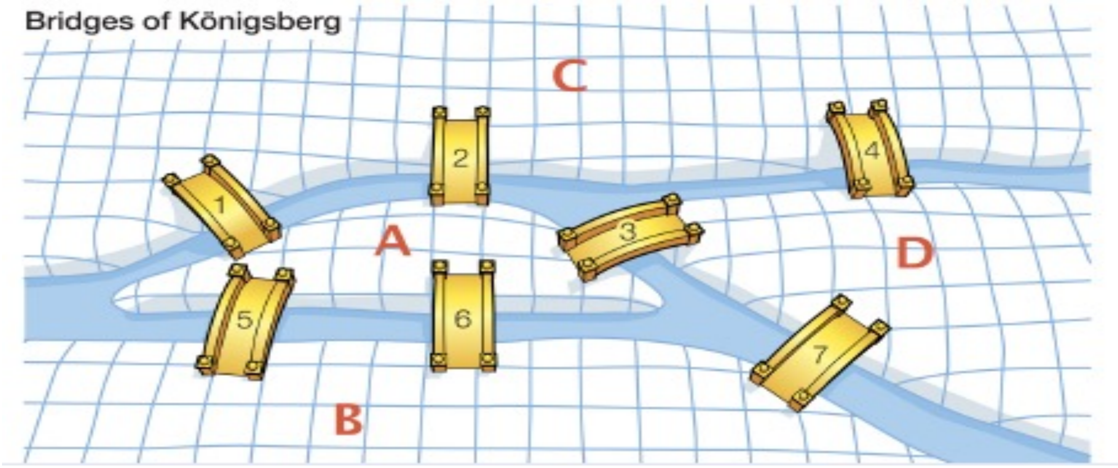


- Set the break points using the control blocks as nodes.
- Locate the control blocks with defects.
- Drill deeper inside the defective control blocks to find the defective statements

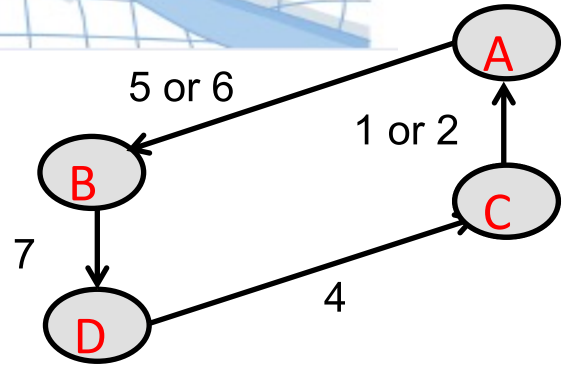
Deriving Visual Graph Models

- Need a programming language to:
  - Extract and abstract visual models from large software.
  - Perform computations on models to solve problems.
- Need interactive querying for a human to experiment with visual models.
- Concept-empowered: A human can discover powerful graph abstractions to solve difficult problems.
- We can leverage graph theory and the technology of graph databases.

The beginning of the graph theory

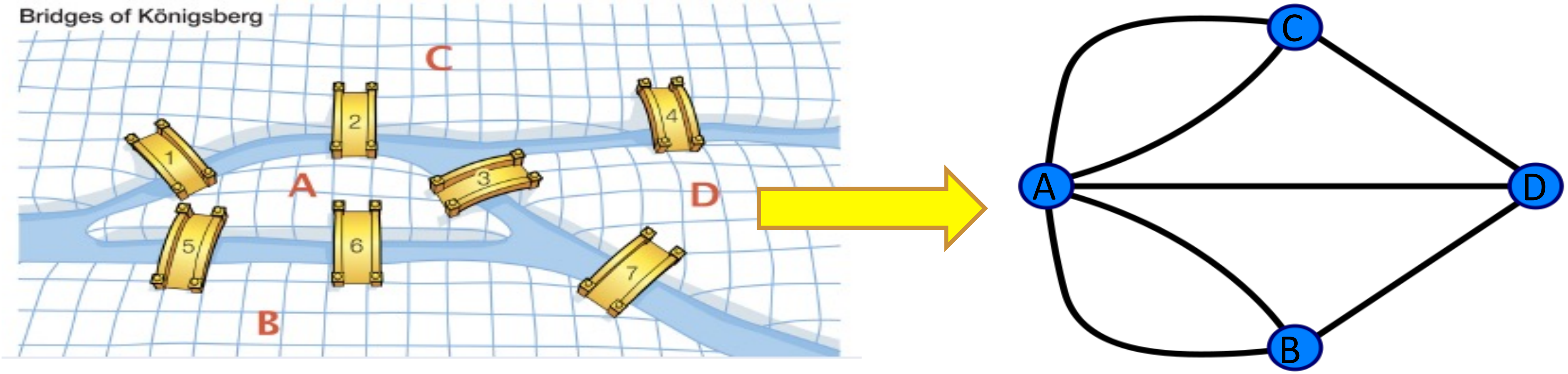


Problem: Is it possible to make a loop starting and ending at the point A and crossing every bridge exactly once?



The loop ABDCA misses three bridges.

Euler introduced the graph abstraction (1735)



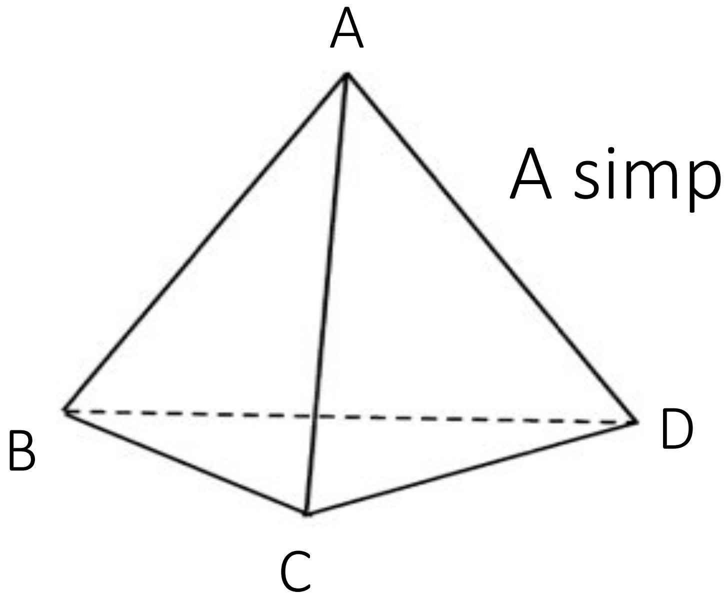
A loop that goes through all edges exactly once is called an *Euler loop.*

Theorem: A Euler loop exists if and only if each vertex has even degree.

Hard Graph Problems: Hamilton's Icosian Game (1857)

Dodecahedron – a polyhedron with 12 faces

Problem: Find a loop through the edge graph of the dodecahedron visiting every vertex exactly once.



A simple case.

Loop: ABCDA

Euler: Compute a loop without repeating edges.

Hamilton: Compute a loop without repeating vertices.

Hamilton problem *unsolved* to date – no one has found an efficient algorithm.

Concluding Remarks

- Graph models and algorithms based on those models help us solve difficult problems of large software.
- Almost three hundred years after the beginning of the graph theory, we have arrived at the modern age of large graphs (e.g. software engineering, bioinformatics, internet, social networks).
- This tutorial introduces the interactive graph database technology and the applications of graph theory for solving software problems.

Demo Videos

Atlas Platform Demo Video:

https://www.youtube.com/watch?v=cZOWlJ-IO0k&feature=youtu.be