CSC207H: Software Design
Lecture 9

Wael Aboelsaadat

wael@cs.toronto.edu
http://ccnet.utoronto.ca/20075/csc207h1y/
Office: BA 4261
Office hours: R 5-7

Acknowledgement: These slides are based on material by Prof. Karen Reid

Programming/scripting languages technologies

- Regular Expressions
- XML
- Parsers

Parsers


Schema
XML
True/False
Match/no-match


Regular
Expression
Pattern
Text
to Search
(aka document)
True/False
Found-it/
didn’t find it

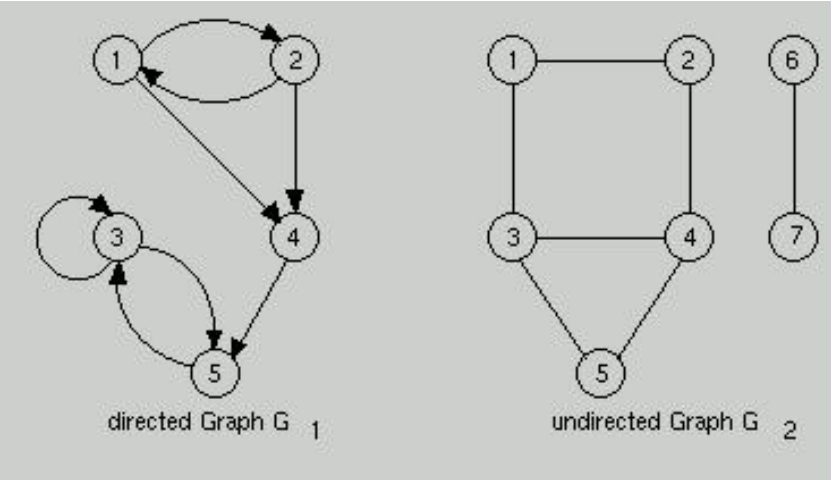
Graphs

Graph

- You have seen some specialized graphs
  - Trees are graphs
  - Linked lists are (simple) graphs
- Parsing uses graph & graph theory
- What is graph theory?

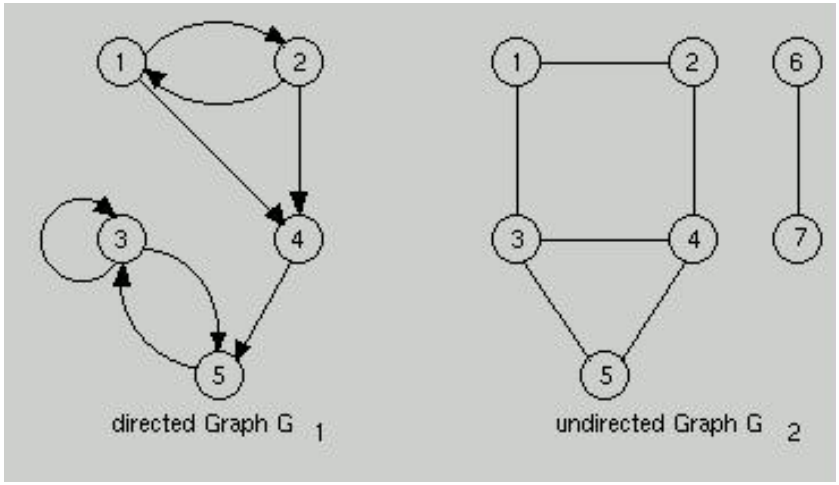
Graph

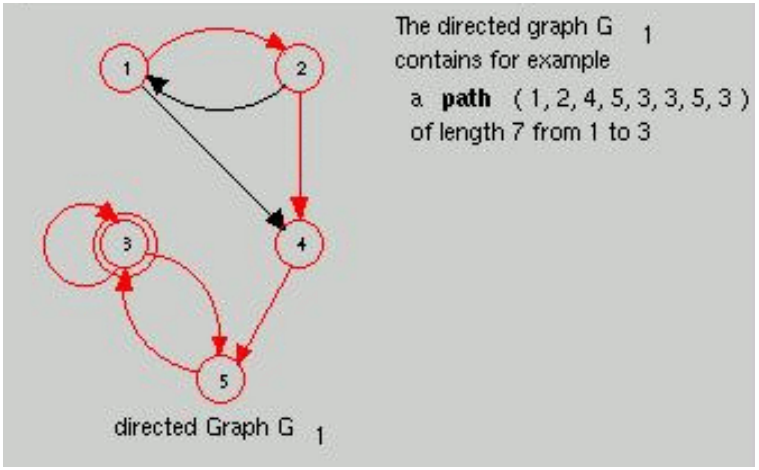
- A graph is a set of nodes connected by arcs
  - Directed graph if the arcs have direction
  - Undirected graph if the arcs simply show connection



Graph

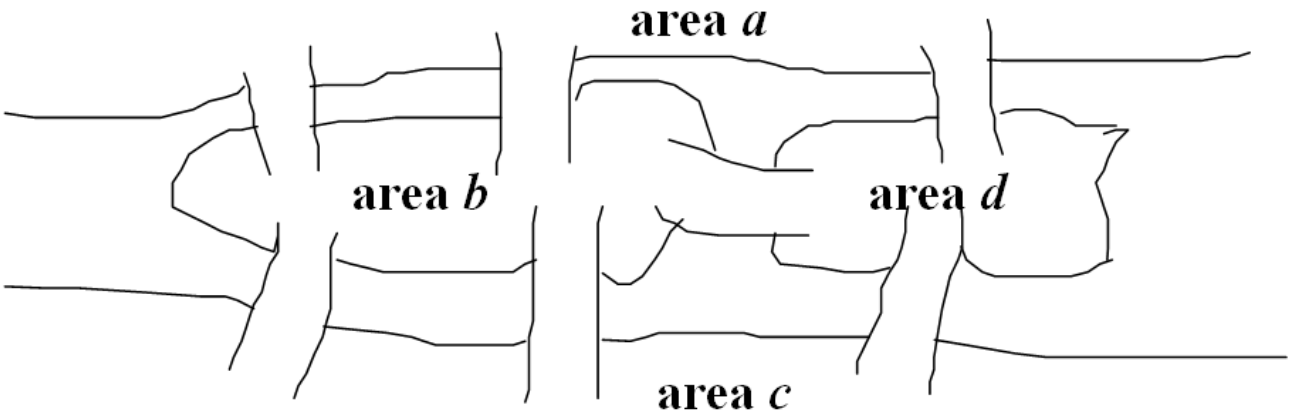
- A graph is a set of nodes connected by arcs
  - Directed graph if the arcs have direction
  - Undirected graph if the arcs simply show connection





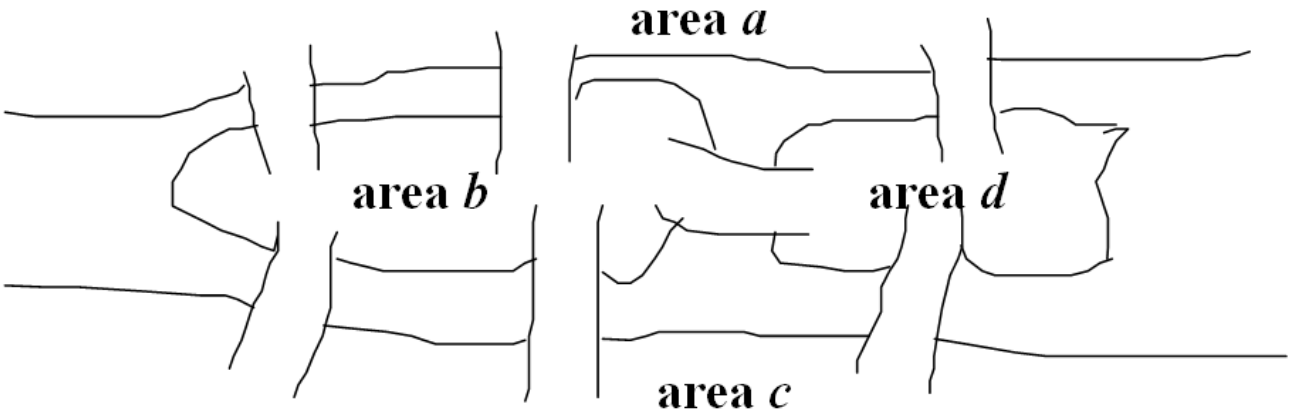
Graph: Königsberg

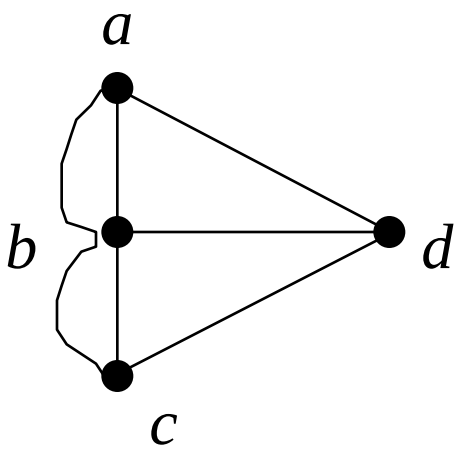
- Königsberg (in Germany) had seven bridges. The townspeople wondered if was possible to take a walk around the town in such a way as to cross each of the seven bridges exactly once.



Graph: Königsberg

- Königsberg (in Germany) had seven bridges. The townspeople wondered if was possible to take a walk around the town in such a way as to cross each of the seven bridges exactly once.



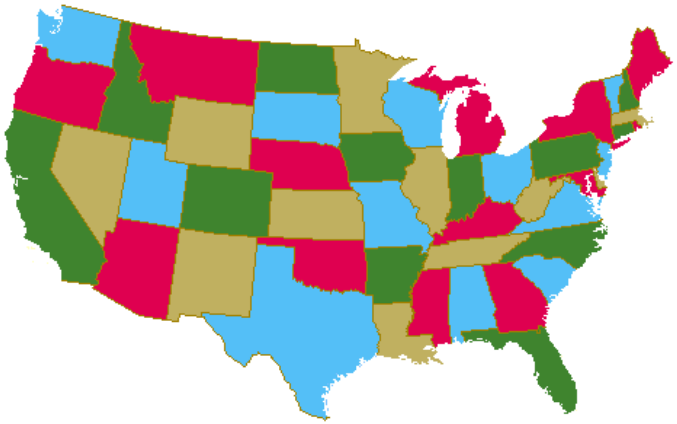


Graph: map coloring

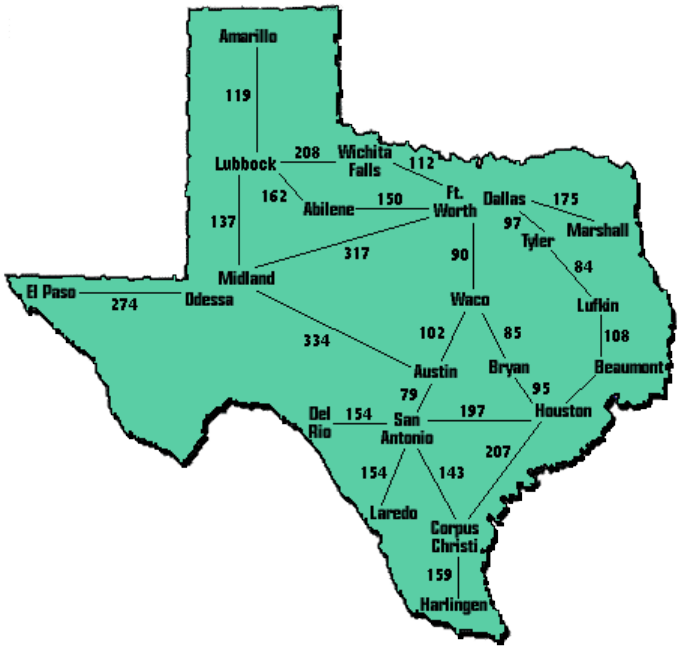
- How many colors do we need to color a map so that every pair of states with a border in common have different colors?

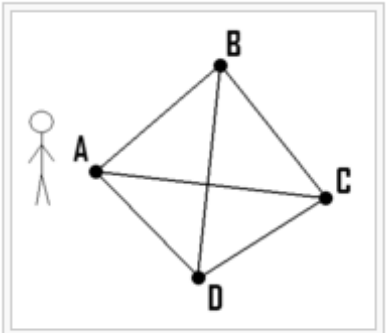
Graph theory: map coloring

- How many colors do we need to color a map so that every pair of states with a border in common have different colors?



Graph: traveling salesman problem

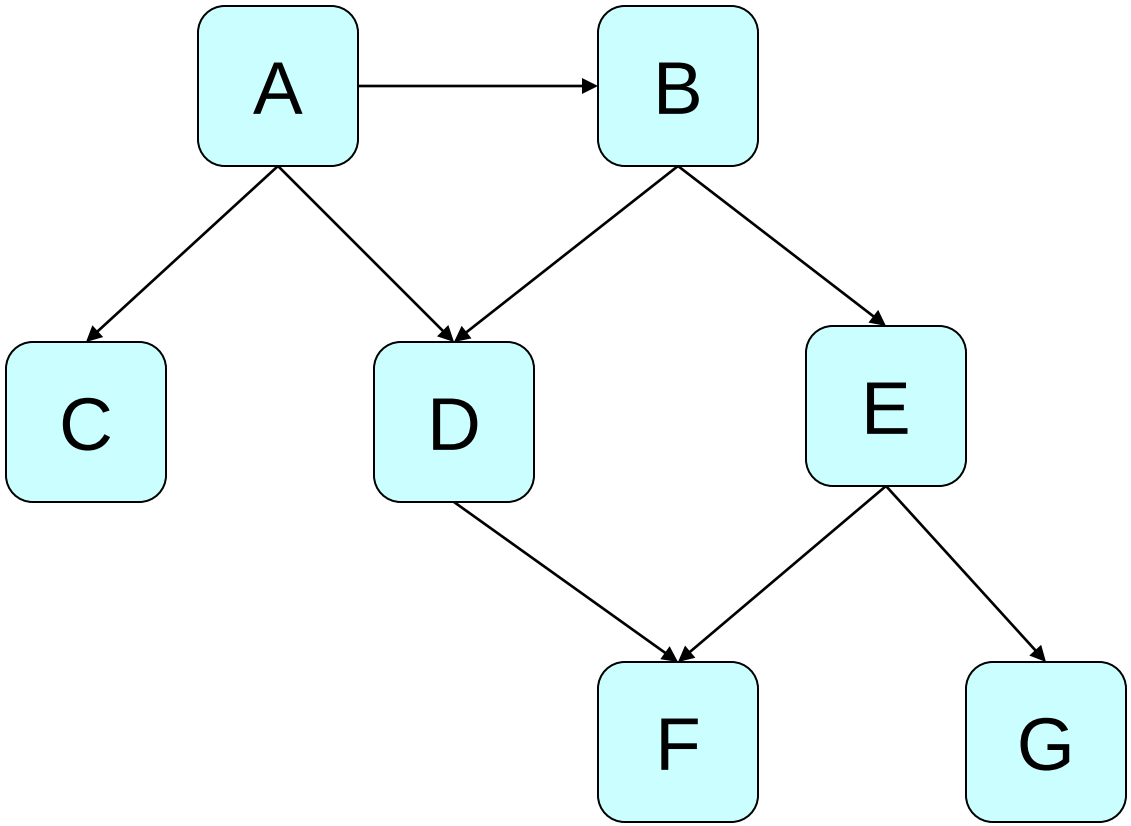




If a salesman starts at point A, and if the distances between every pair of points are known, what is the shortest route which visits all points and returns to point A?

- http://en.wikipedia.org/wiki/Traveling_salesman_problem

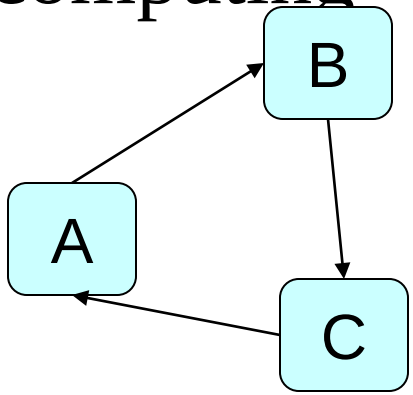
Example



Key	Value
A	{D, B, C}
B	{D, E}
C	{}
D	{F}
E	{F, G}
F	{}
G	{}

Graph Algorithms

- Crop up *everywhere* in computing
- Most are recursive
- Must handle circularity



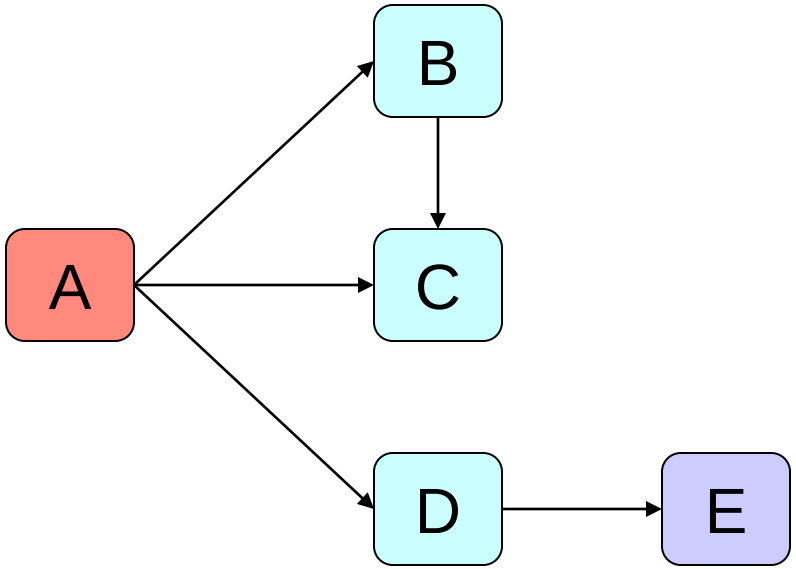
- Common solution is to keep track of nodes already visited using a set/stack

Stack??

- First in, last out (FILO)…
- How do you implement that?

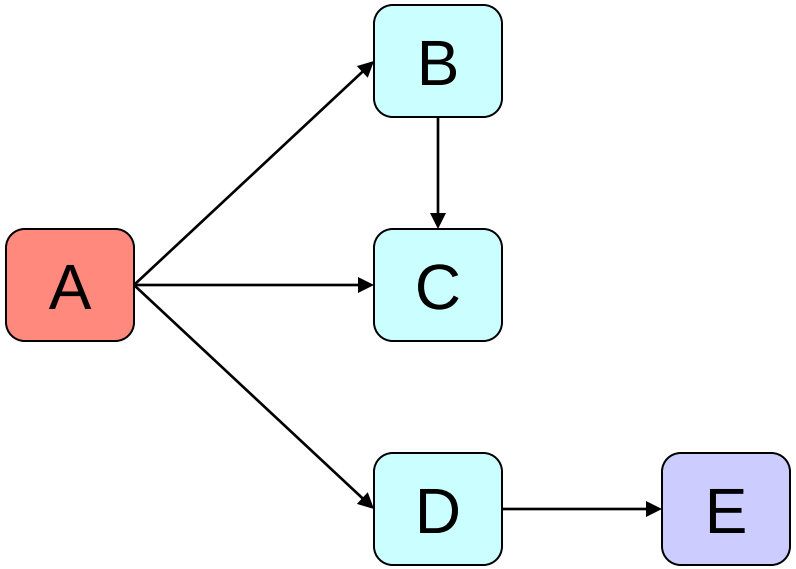
Example: Reachability

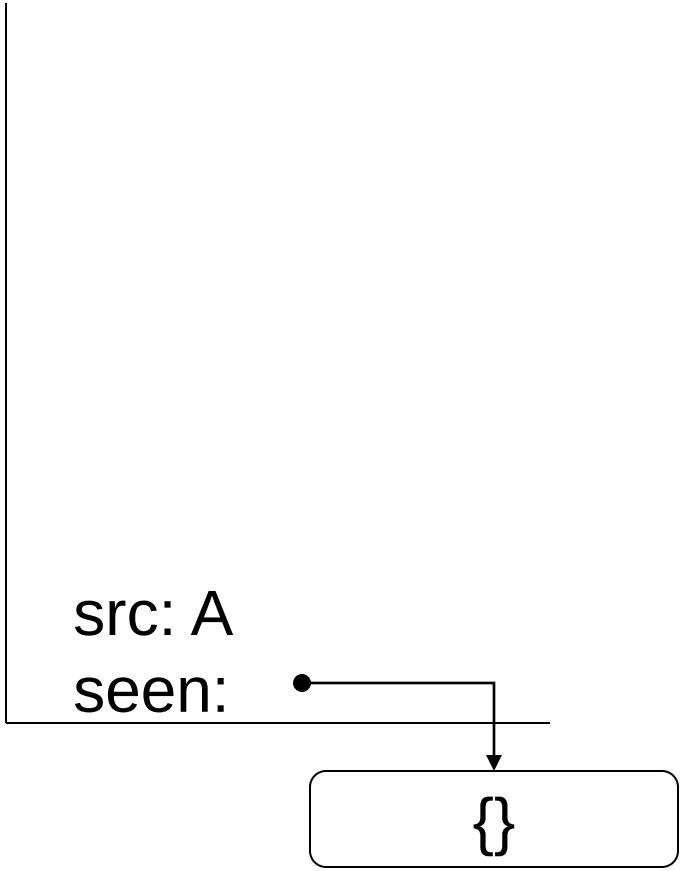
- Can we get to E from A?



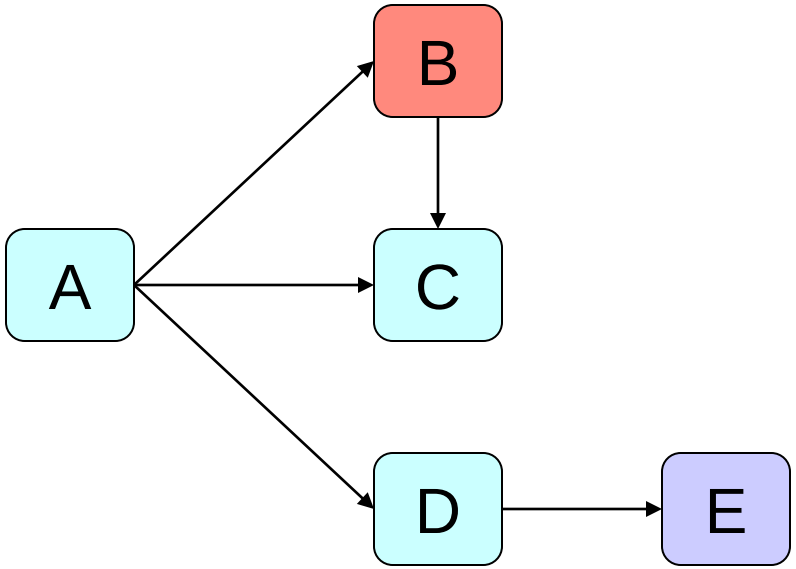
Example: Reachability

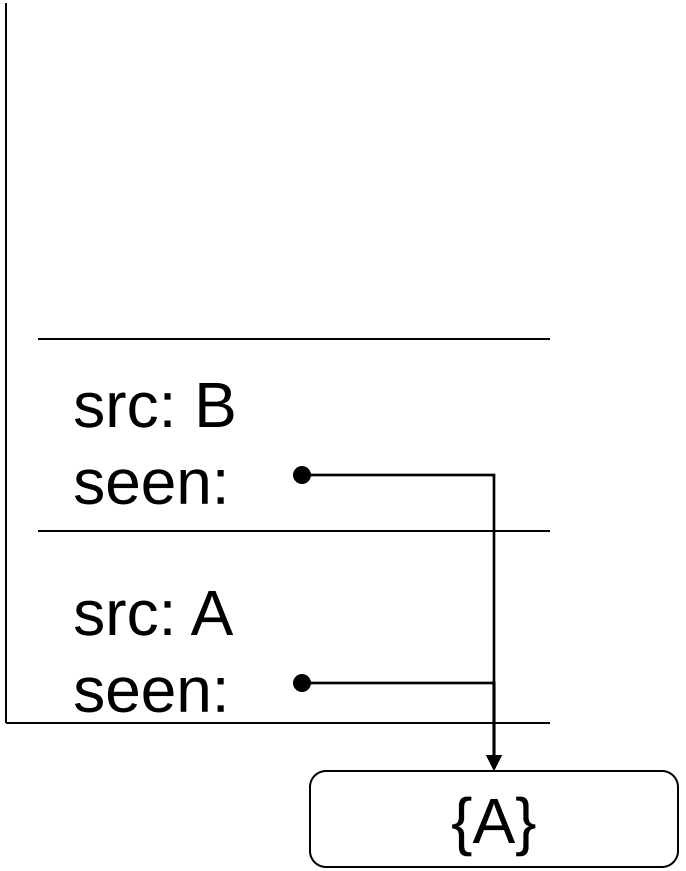
For each child of the src, check if dst is reachable from that child



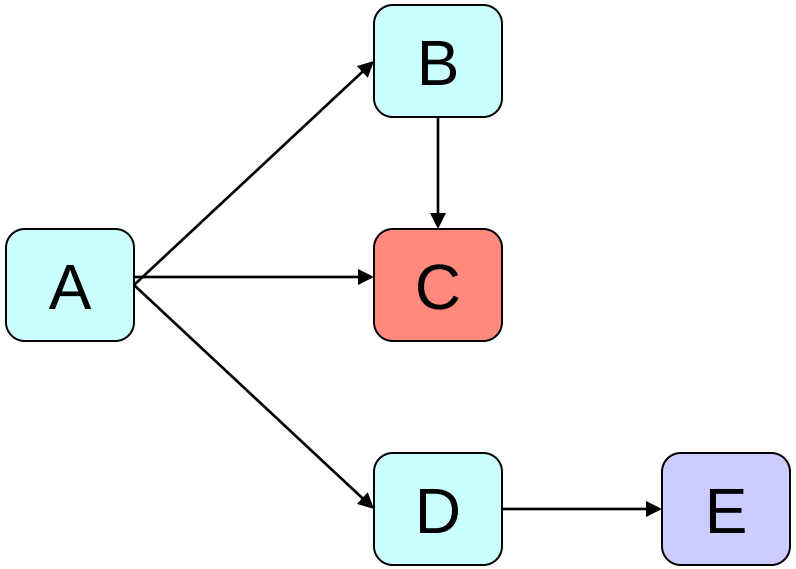


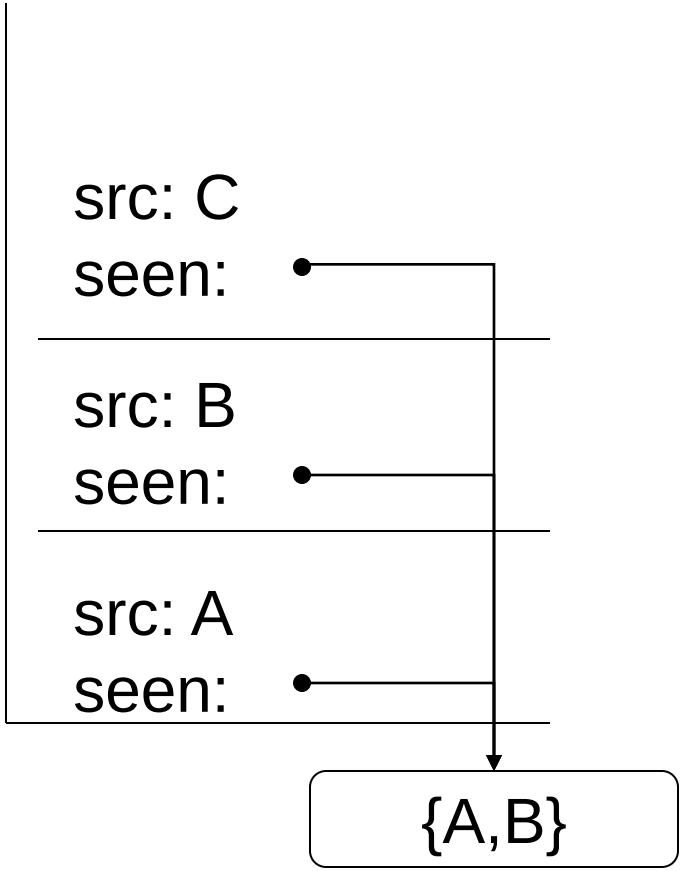
Example: Reachability



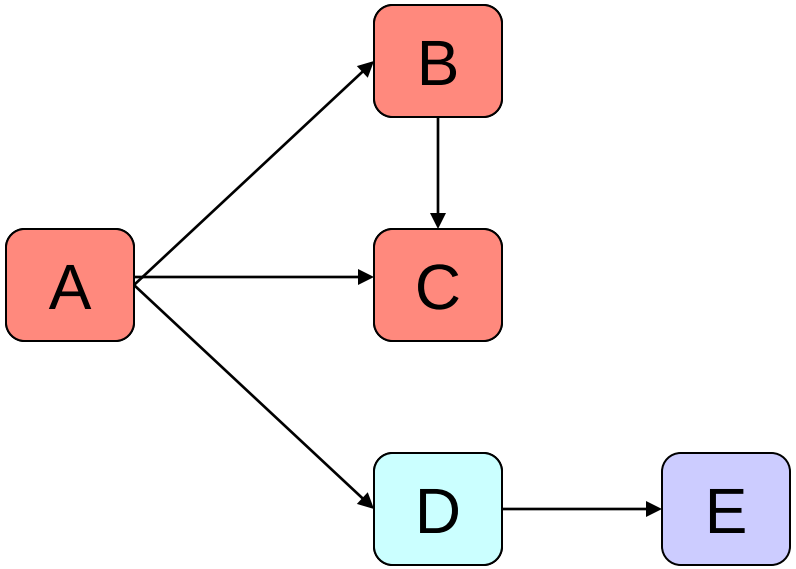


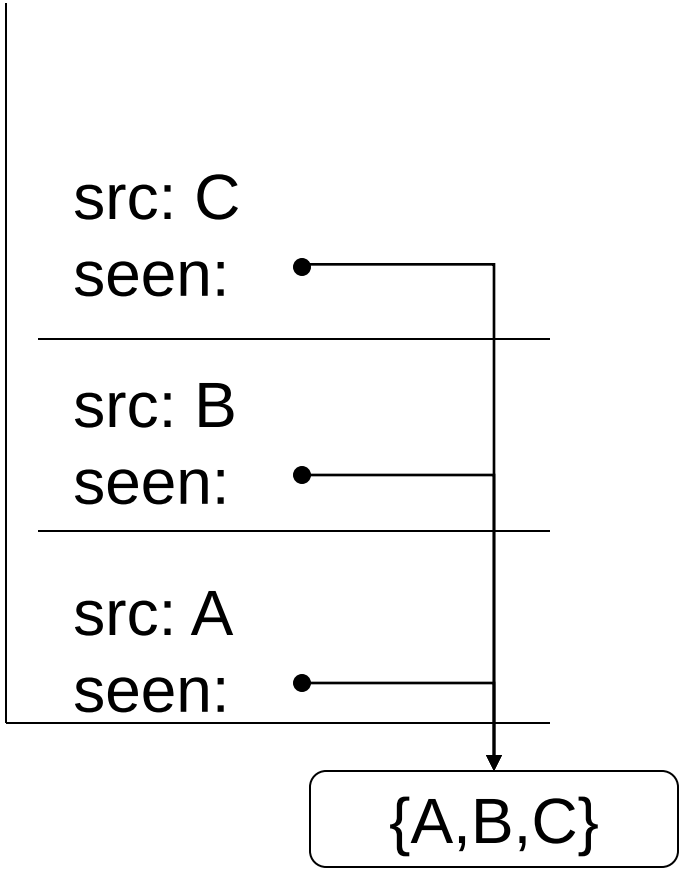
Example: Reachability



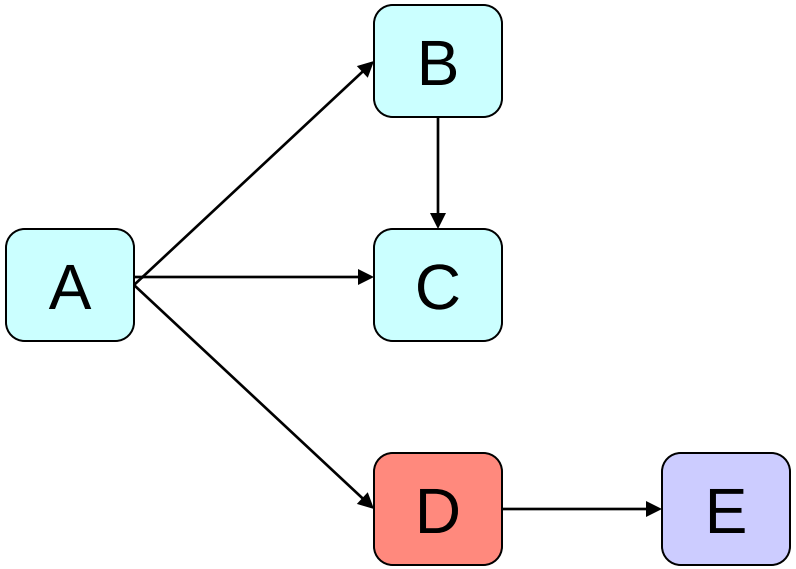


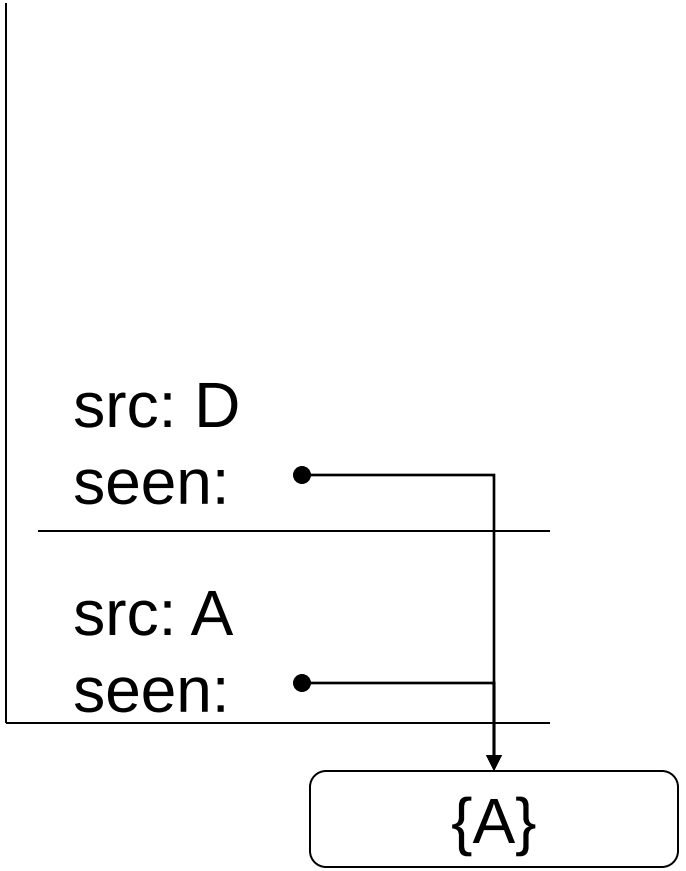
Example: Reachability



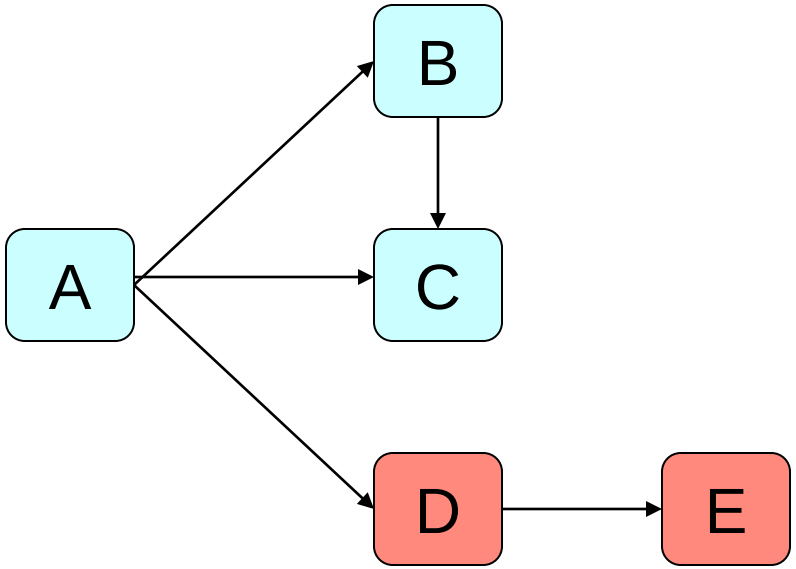


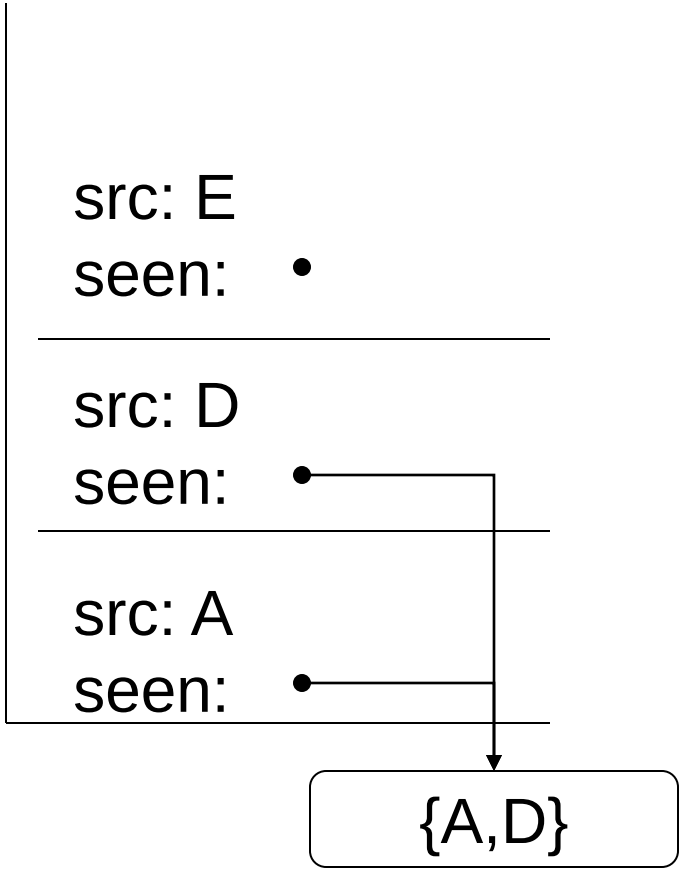
Example: Reachability



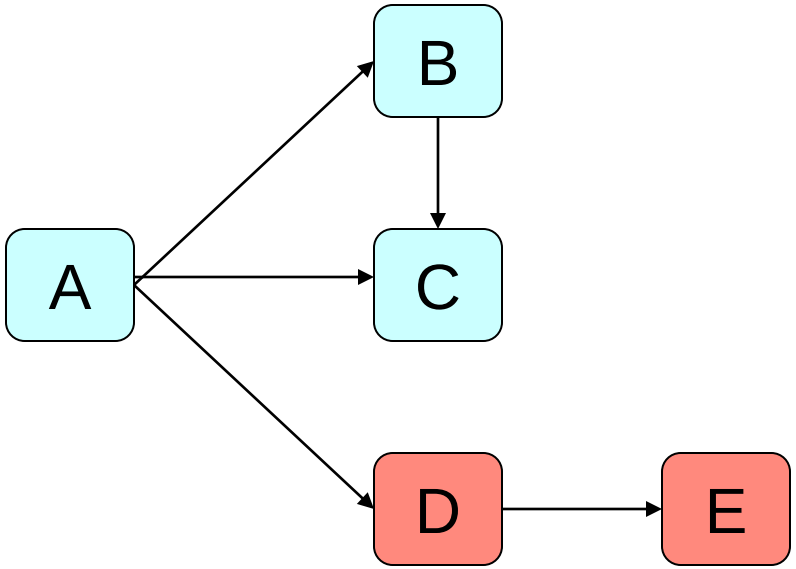


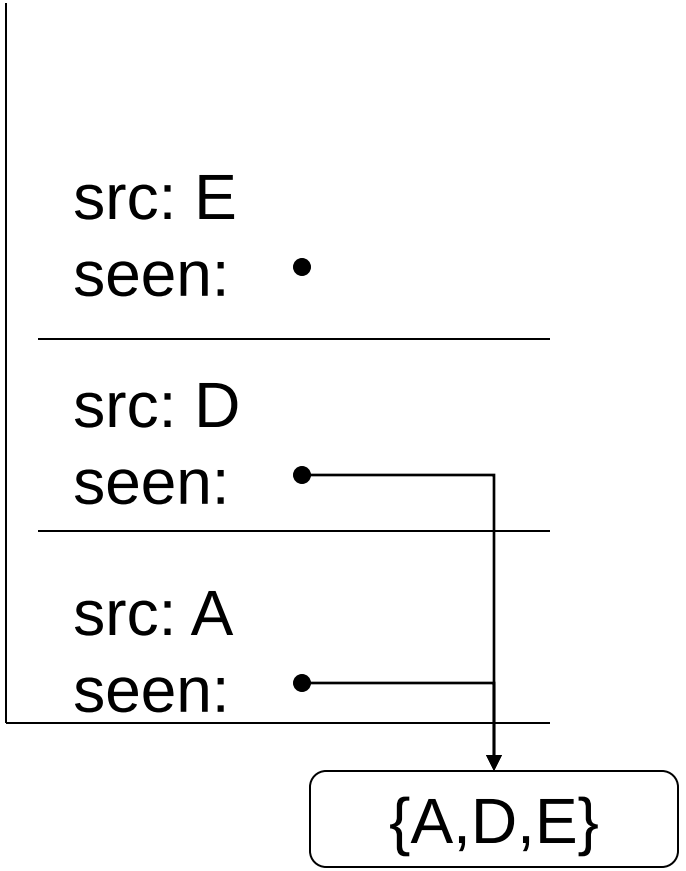
Example: Reachability





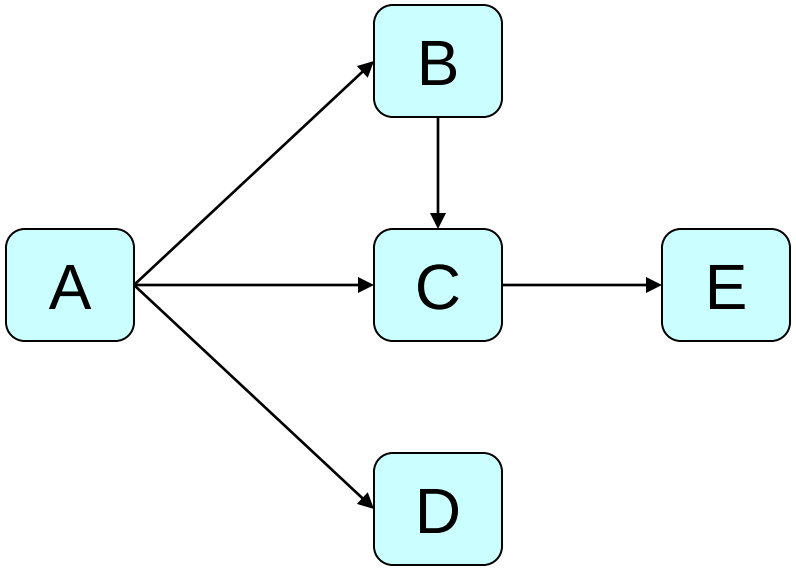
Example: Reachability



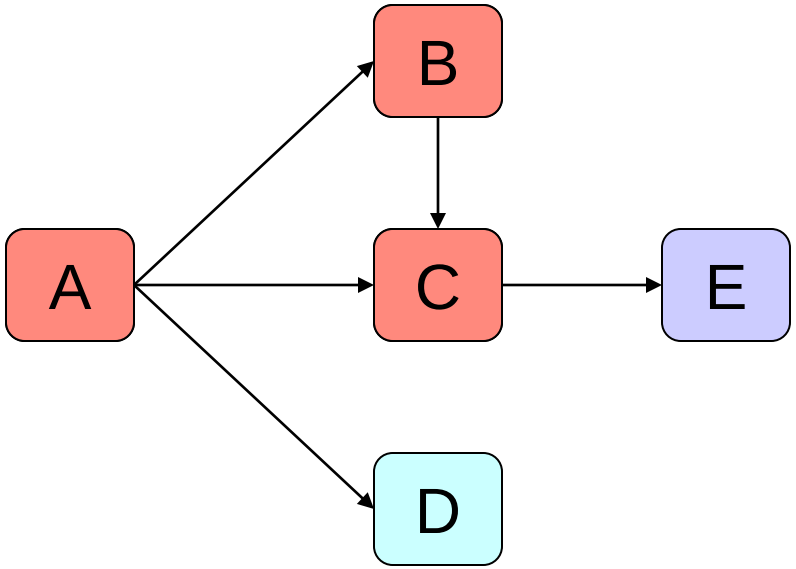


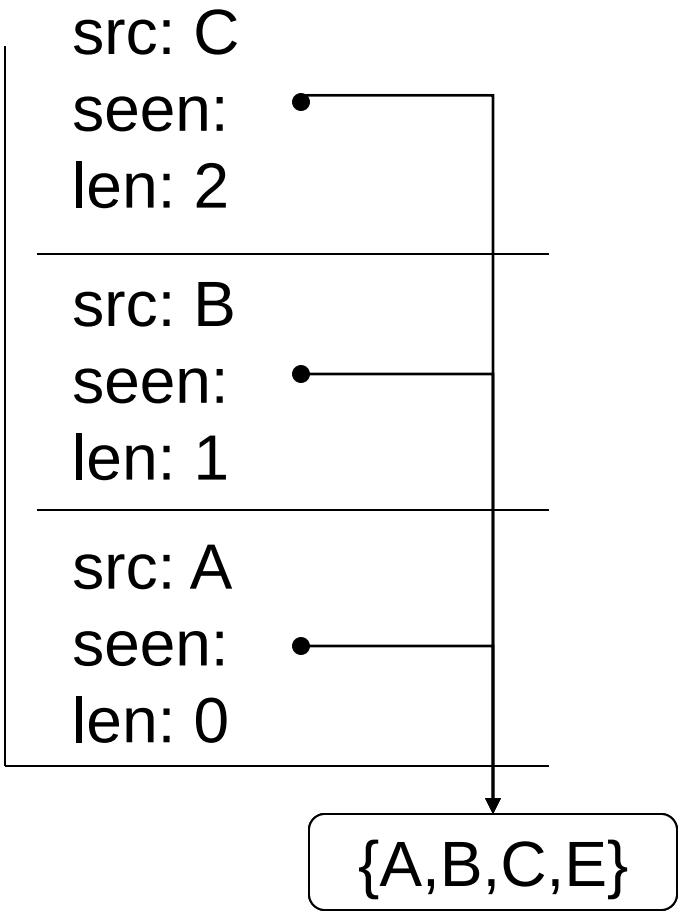
Shortest Path

- What is the shortest path from A to E?

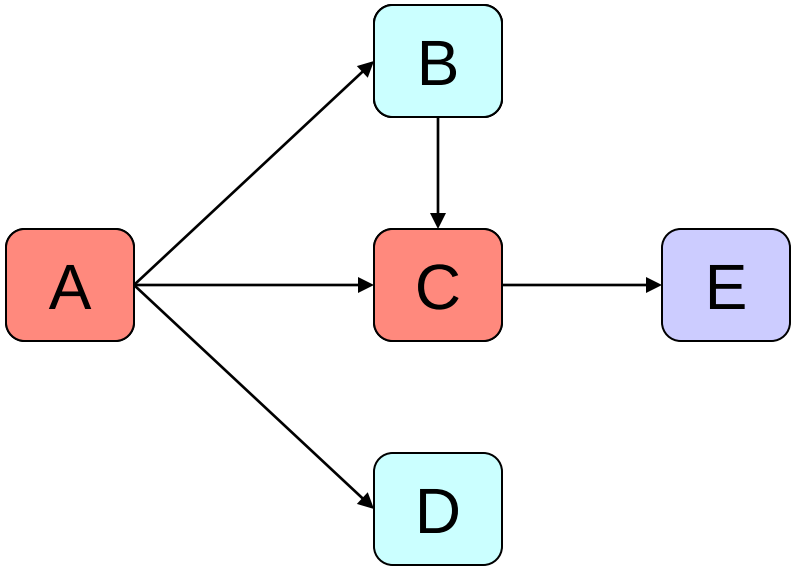


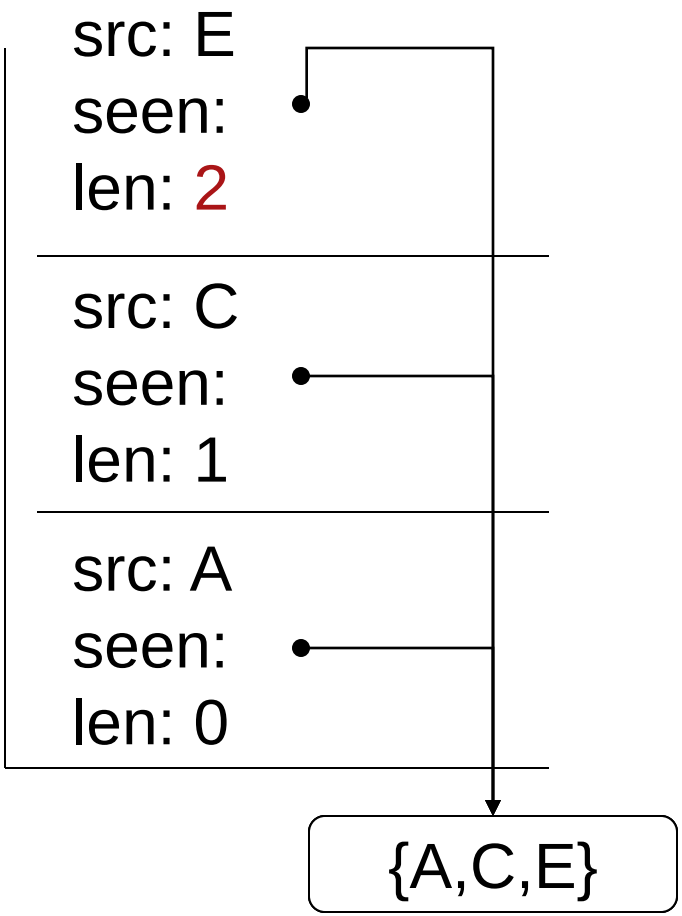
Shortest Path



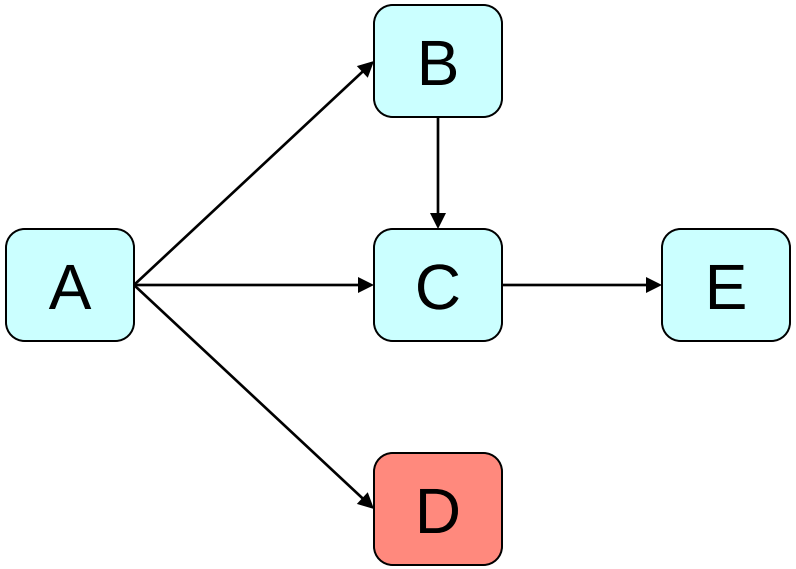


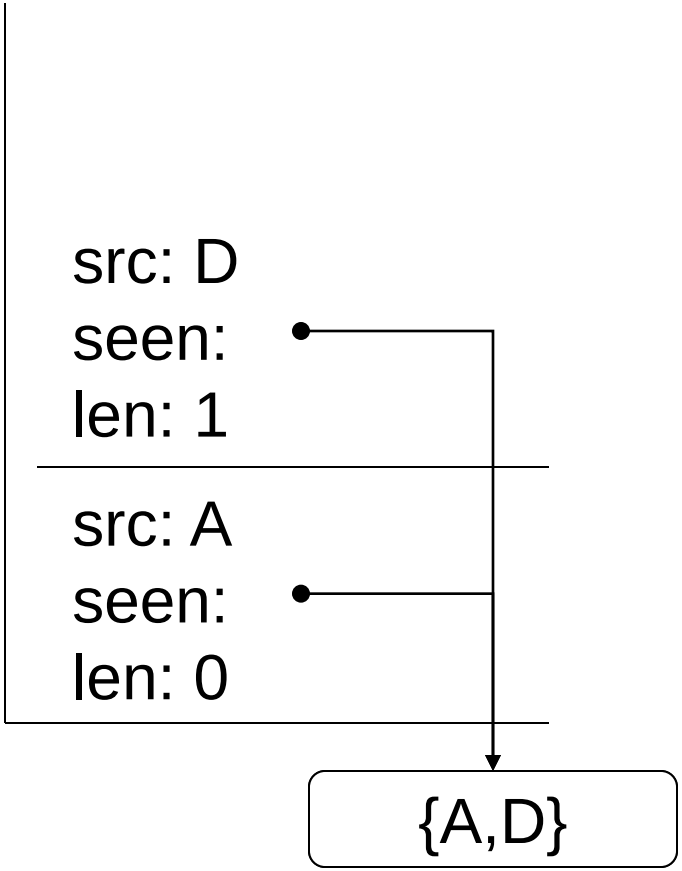
Shortest Path





Shortest Path





Simple Graph Implementation

- Graphs are stored as maps
- Keys are the names of nodes
- Values are the set of nodes
  - A node B is in the set of nodes for node A if there is an arc from A to B

Graphs

- class DirectedGraph {
- public DirectedGraph() {...}
- public Set getNodes() {...}
- public boolean hasNode(Object node) {...}
- public void addNode(Object node) {...}
- public void addNode(Object node, Collection arcs) {...}
- public void removeNode(Object node) {...}

- public Set getArcs(Object node) {...}
- public boolean hasArc(Object src, Object dst) {...}
- public void addArc(Object src, Object dst) {...}
- public void addArcs(Object src, Collection allDst) {...}
- public void removeArc(Object src, Object dst) {...}
- public String toString() {...}
- protected Map fNodes;
- }

Design Choices

- What happens if we try to add a node that is already present?
  - Or duplicate an arc?
  - Or remove a node or arc that doesn’t exist?

Design Choices

- What happens if we try to add a node that is already present?
  - Or duplicate an arc?
  - Or remove a node or arc that doesn't exist?
- Could throw an exception
  - Means that users have to write code like
    - if (not present) then add node
- Could just ignore operations that don't make sense
  - Results in late failures

More Design Choices

- What should a graph return when asked for its nodes?
- What if it is asked for a node’s arcs?
- What if a user wants all arcs?

More Design Choices

- What should a graph return when asked for its nodes?
  - Return a map's keys as a set
- What if it is asked for a node's arcs?
  - The set used to store the arcs?
    - Efficient, but allows users to mess up data structure
  - A copy of the set?
    - Less efficient for large graphs with many arcs, but safer
- What if a user wants all arcs?
  - Not directly available, but easy to construct
    - A set or a list of two-element lists (arrays)?

Reachable method

- public static boolean reachable(DirectedGraph graph,
- Object src, Object dst, Set seen) {

- }

Reachable method

```
public static boolean reachable(DirectedGraph graph,
                    Object src, Object dst, Set seen) {
    // Are we there yet?
    if (src == dst) {
        return true;
    }
    // Try to get there indirectly
    Iterator ia = graph.getArcs(src).iterator();
    while (ia.hasNext()) {
        Object next = ia.next();
        if (!seen.contains(next)) {
            seen.add(next);
            if (reachable(graph, next, dst, seen)) {
                return true;
            }
        }
    }
    return false;
}
```

Parsing Text Files

How to build a build tool?

- Instructive to see how tools like Make are constructed
  - Good application for object-oriented design
  - Shows how useful graph theory is
  - Another example of reading input
  - Helps you see that the tools you use are just like the programs you write (just bigger)

Strategy

- Major parts
  - reading build files
  - executing rules
- Steps:
  - requirements
  - design
  - implementation
  - testing

Requirements

- Command-line tool, not GUI
- Read rules from a single input file
  - Real Make allows files to include other files
- Ignore macros, pattern rules, special variables, etc.
  - None of these are especially difficult but we will keep it simple to start with

Elements of a Makefile

- Blank lines
- Comments
- Rules consisting of:
  - A *head* made up of a target and some prerequisites (dependencies), and
  - A *body* containing zero or more actions

Plan of Attack

- Build a placeholder Rule class
  - A single target, a set of prerequisites, and a list of actions
- Build a parser
- Go back and improve the Rule representation

Notes

- Often write temporary *placeholders (we call these stubs)*, then replace them.

- Much easier to do if the system is *modular*
  - If two pieces communicate through an interface, they can be modified independently
  - Modular systems are not just easier to maintain, they are also easier to build

Design

Implementation: simple Rule class

```
class Rule {
        /** Construct empty rule. */
        public Rule() {
        }

        // Target, pre-reqs, and actions.
        protected String fTarget;
        protected List   fPrereqs;
      protected List   fActions;
}
```

Implementation: simple Parse class

```
class Parse {
        /** Construct empty rule. */
        public Parse() {
        }

        public Rule[] parse(String strFileContents )
}
```

Notes on Style

- Order of a class
  - Static variables and methods
  - Public methods
  - Protected or private methods
  - Member variables
- Use “fName” for fields
  - Less error-prone than using nothing, or this.name
  - What about Hungarian Notation?
    - http://en.wikipedia.org/wiki/Hungarian_notation

Style

- Put common initialization in a protected method
- Declare members to be of most abstract types (*e.g.* List instead of ArrayList)
  - Makes it easy to change mind later
- Prefer protected to private
  - Sooner or later, will want to override it ...
  - ... or test it
- Everything is negotiable
  - Consistency is more important than any detail of style

First Parser

- Parser class uses static methods
- Reads to the end of stream, creating rules
  - Can handle standard input as well as files
- Restrict input
  - No blank lines or comments
  - One pre-requisite per rule
  - One action per rule
- Rules for exploratory programming:
  - Do it
  - Do the simplest thing that could possibly work
  - Build one to throw away

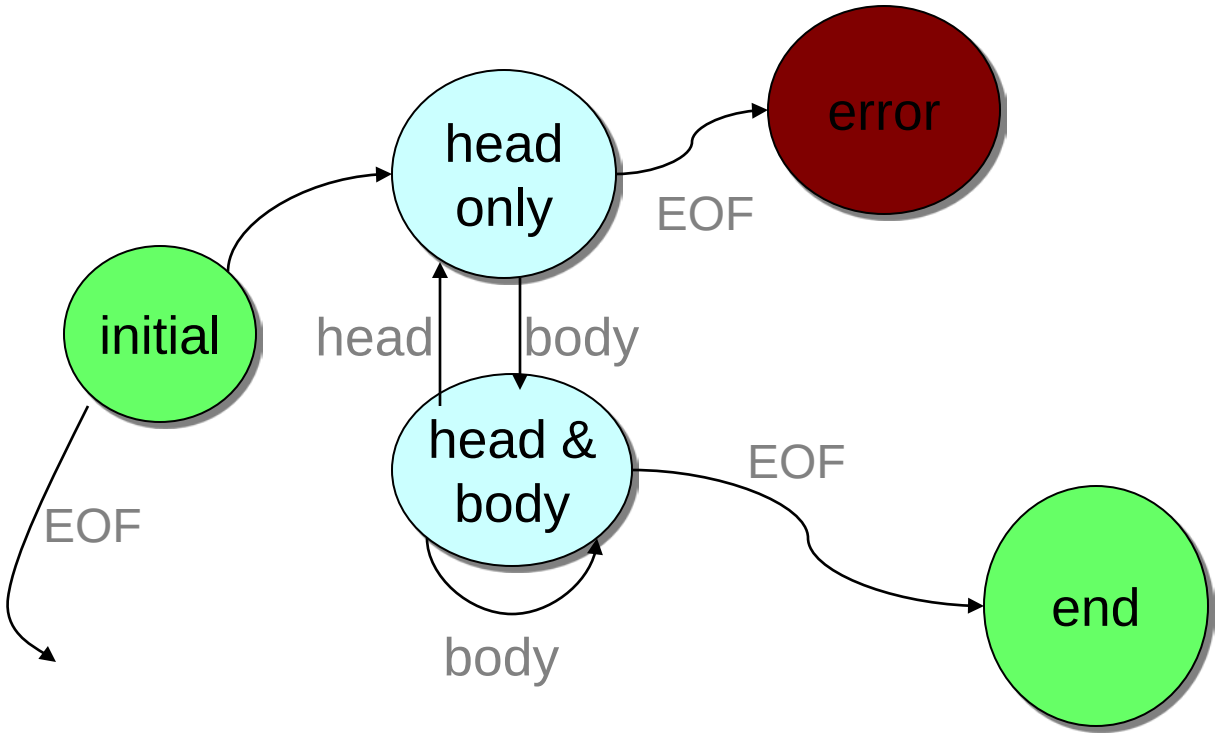
First Parser: code

State Machine Parser

- States (based on FSM)
  - Initial, head only, head and body
- Classifiers
  - head line, body line
- Handlers

Finite State Machine for our Parser

Finite State Machine for our Parser



Finite State Machine Parser: code

Comments and Blank Lines