

<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *digraph search*
- ▶ *topological sort*
- ▶ *strong components*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

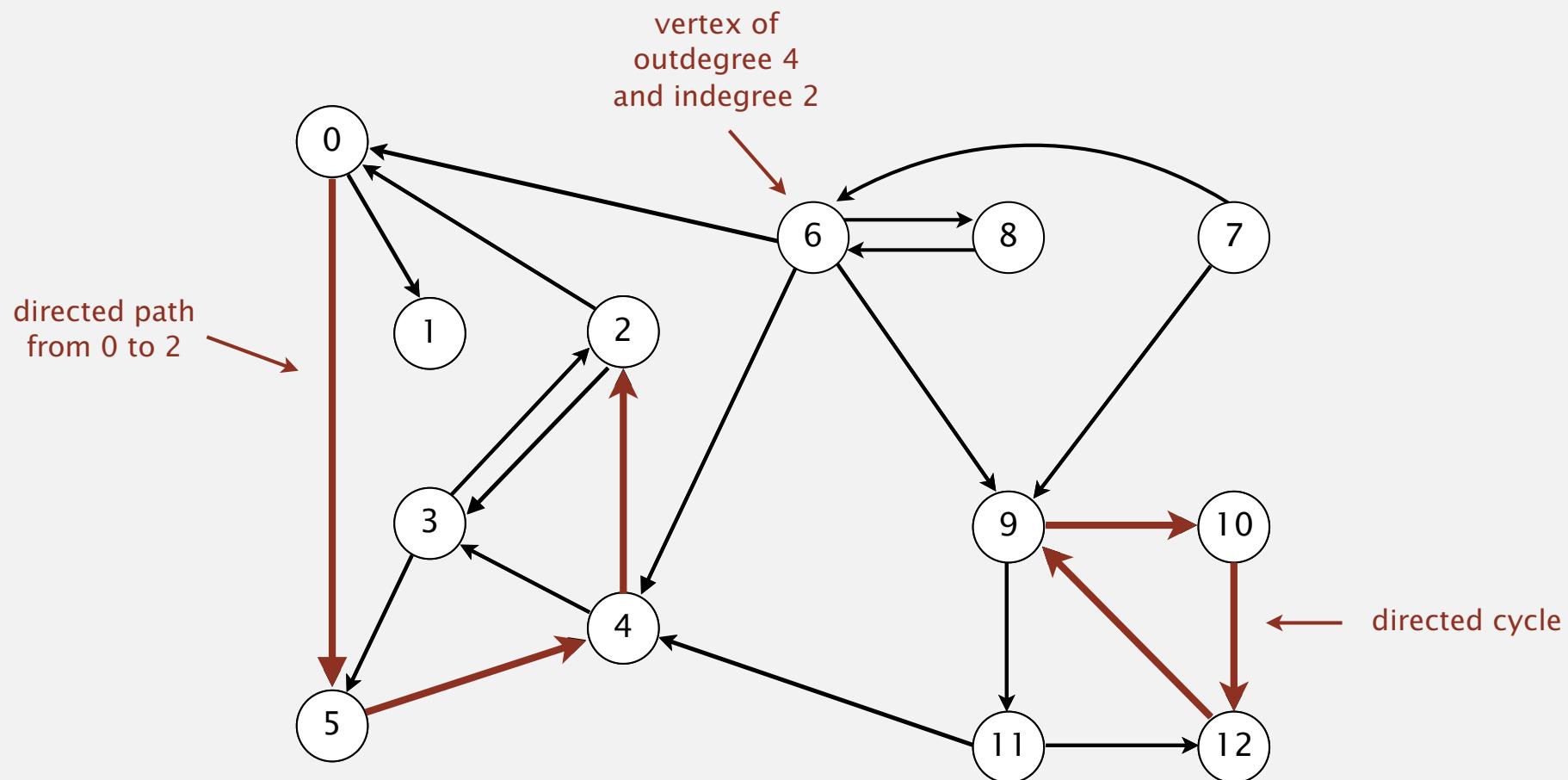
<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *digraph search*
- ▶ *topological sort*
- ▶ *strong components*

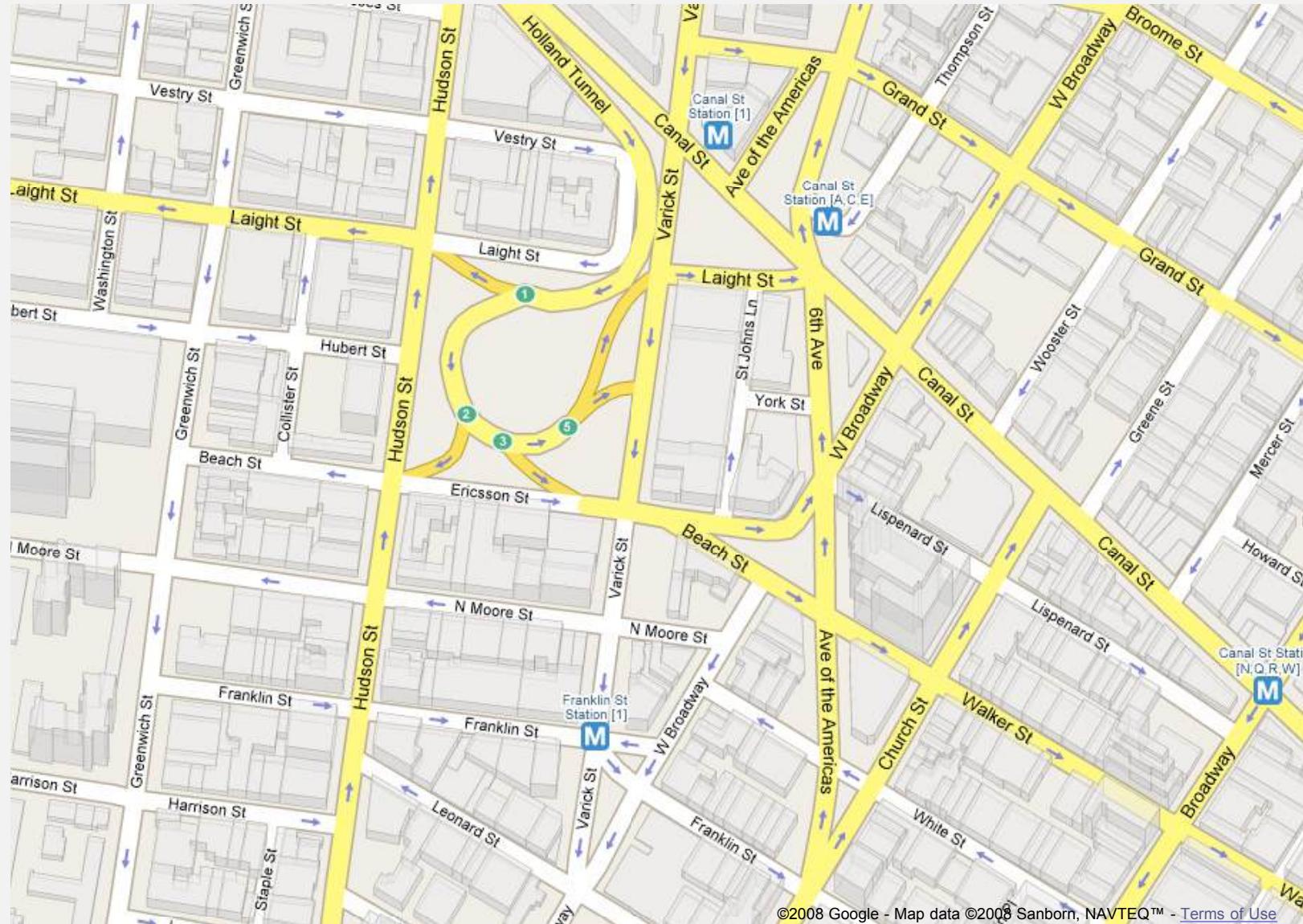
Directed graphs

Digraph. Set of vertices connected pairwise by **directed** edges.



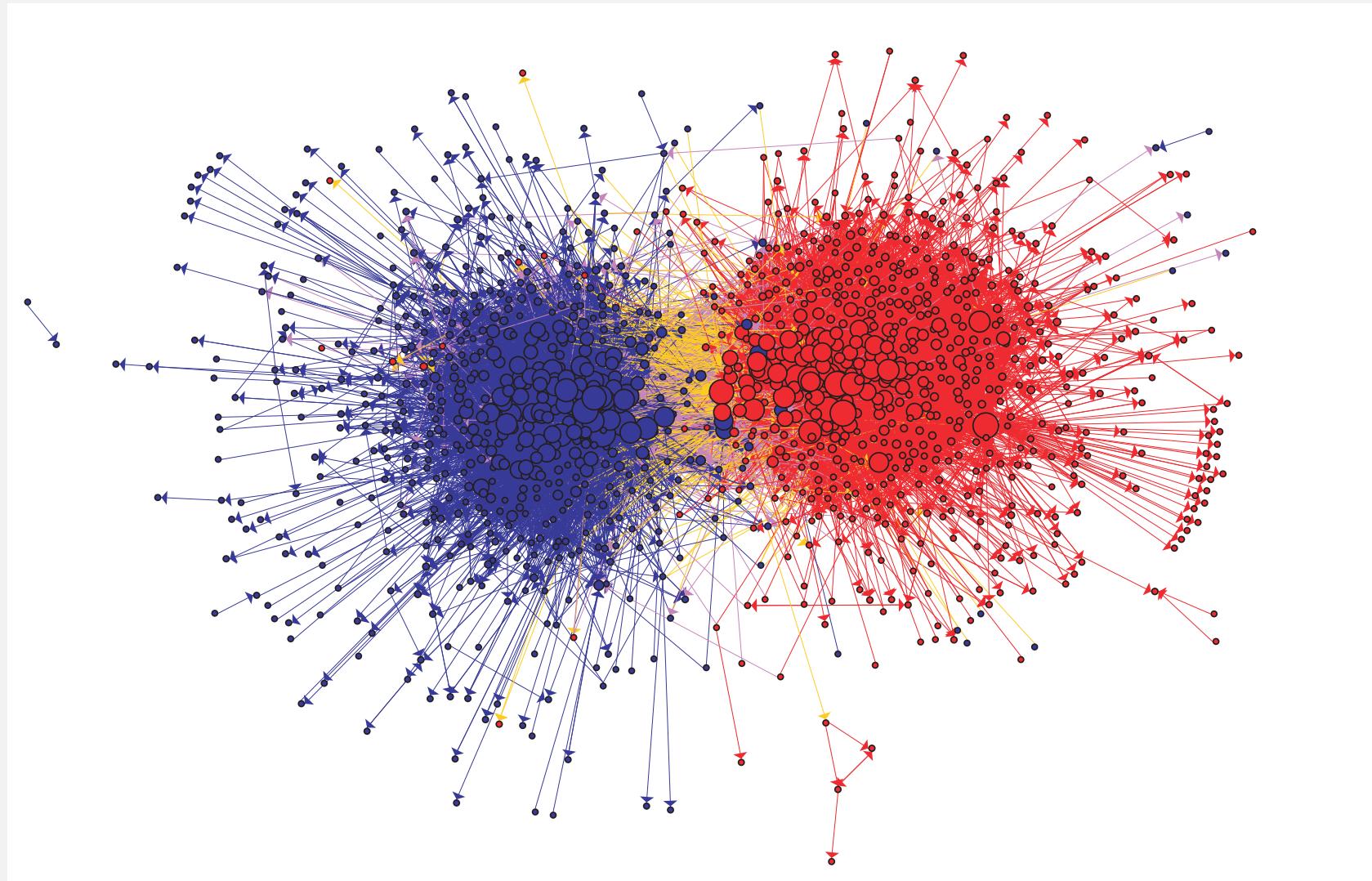
Road network

Vertex = intersection; edge = one-way street.



Political blogosphere graph

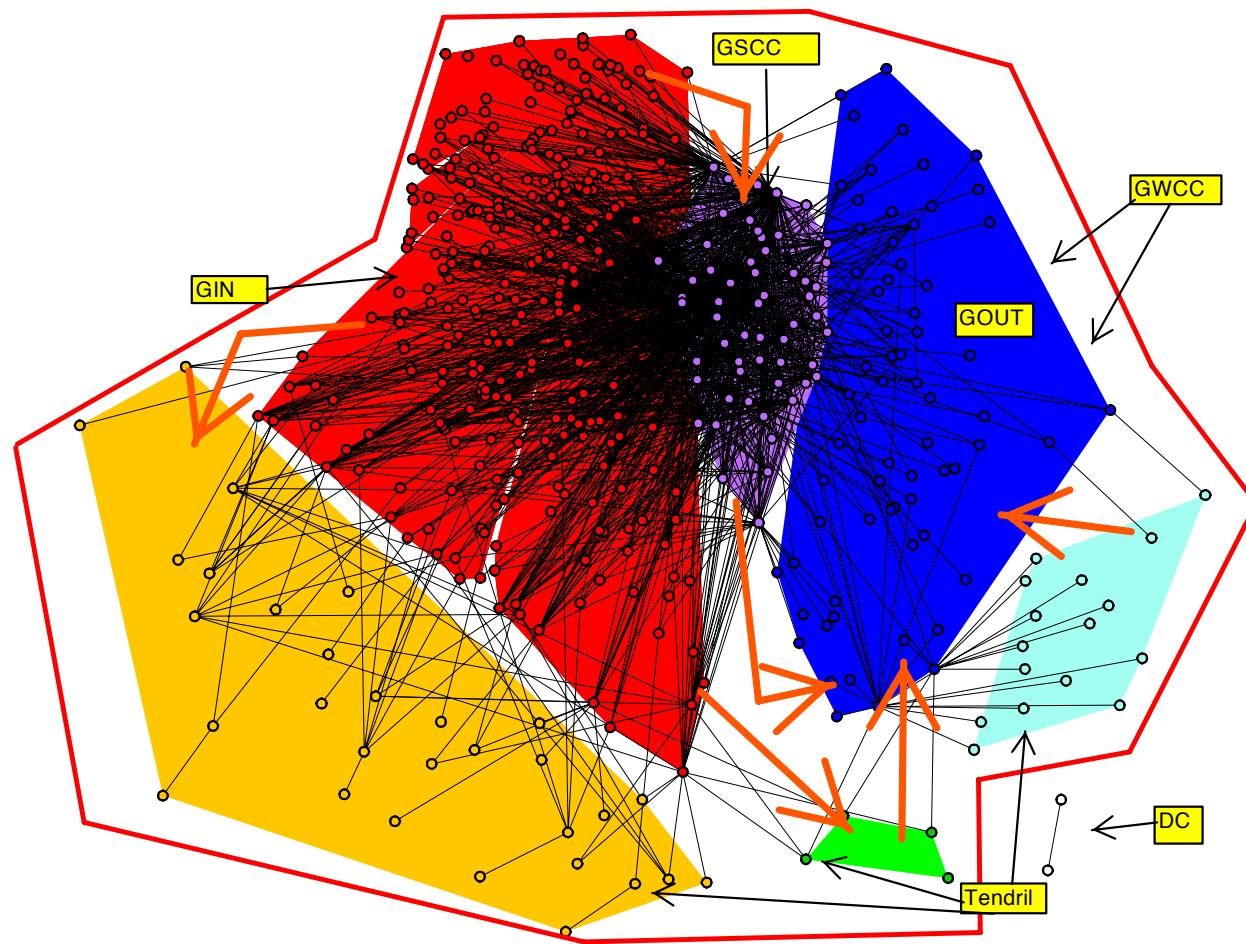
Vertex = political blog; edge = link.



The Political Blogosphere and the 2004 U.S. Election: Divided They Blog, Adamic and Glance, 2005

Overnight interbank loan graph

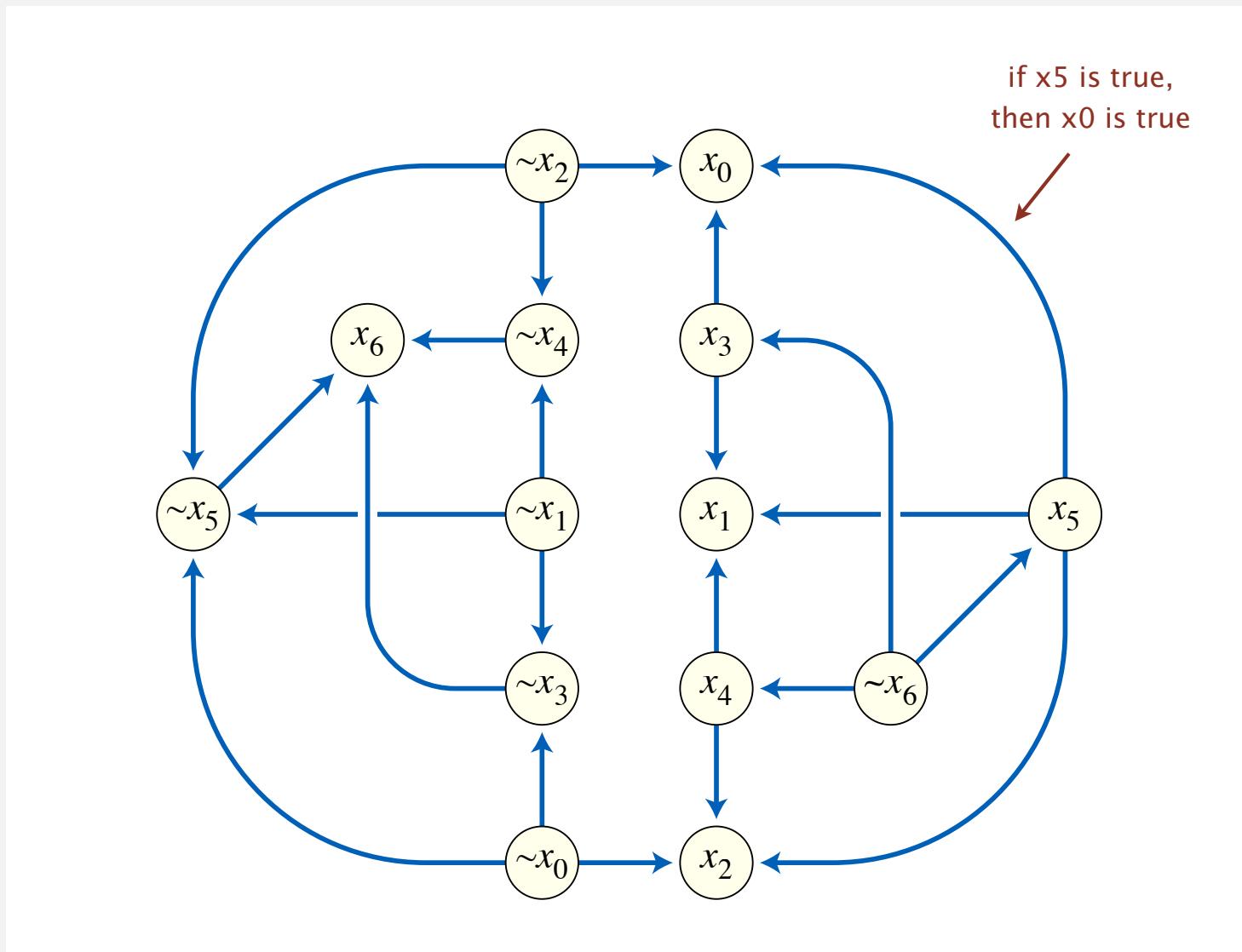
Vertex = bank; edge = overnight loan.



The Topology of the Federal Funds Market, Bech and Atalay, 2008

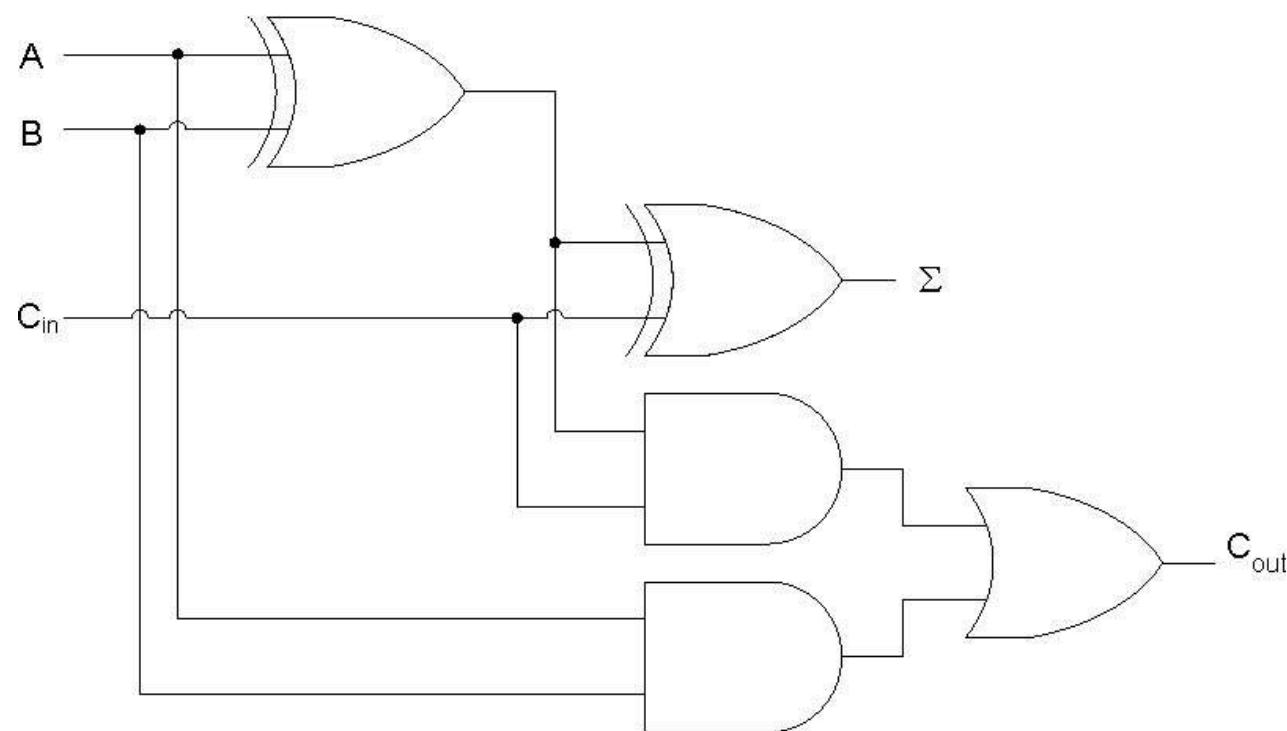
Implication graph

Vertex = variable; edge = logical implication.



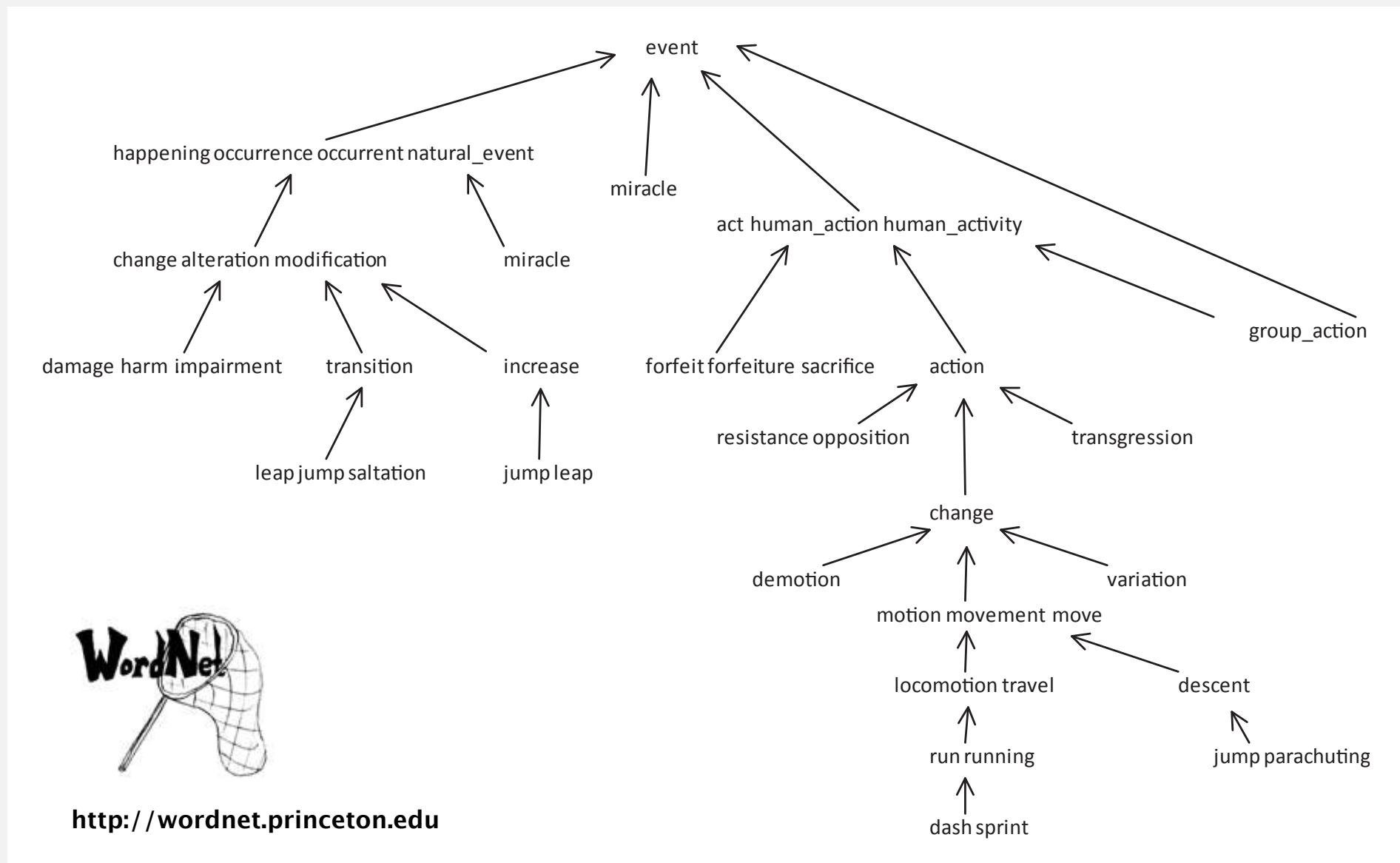
Combinational circuit

Vertex = logical gate; edge = wire.



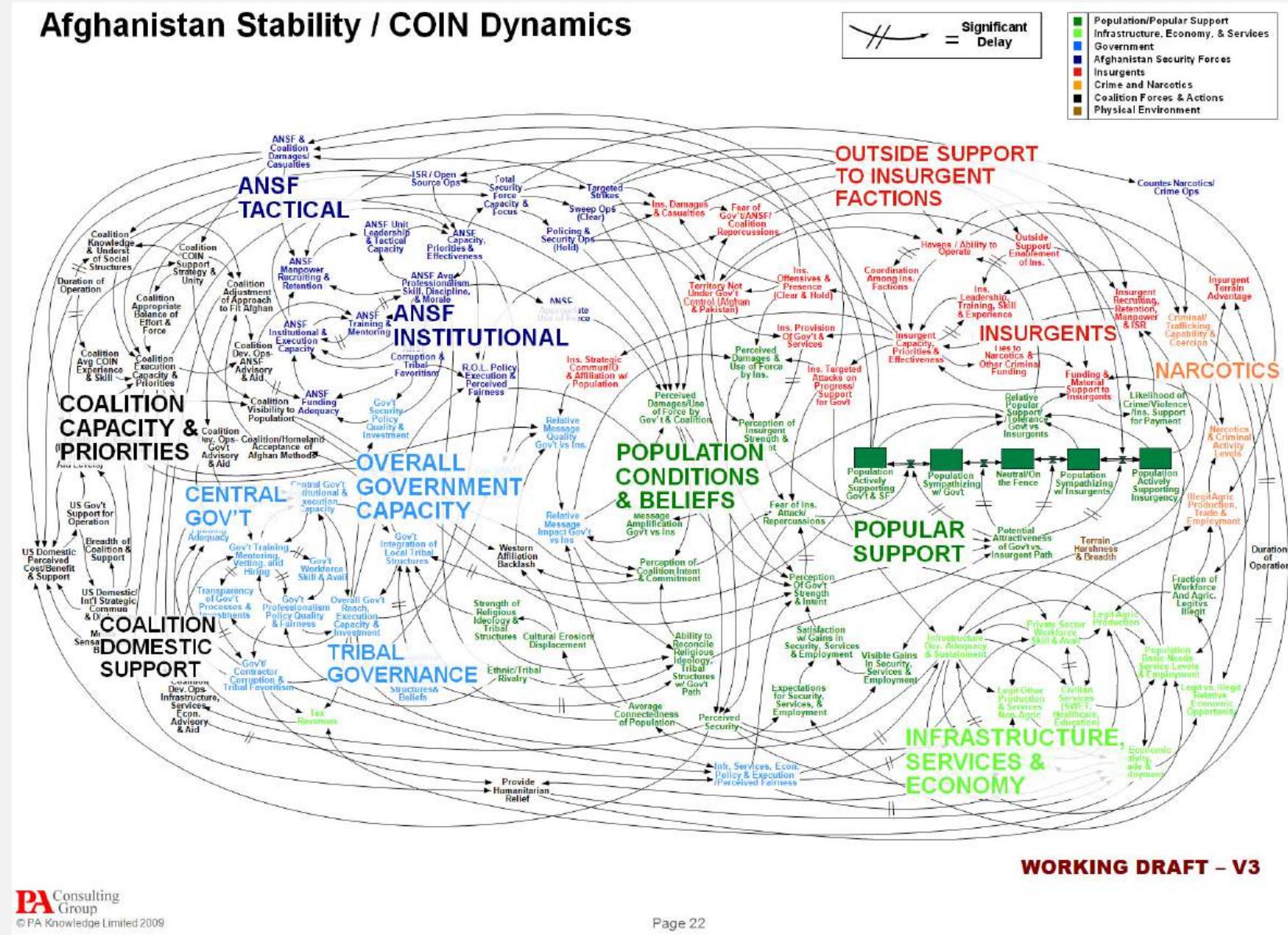
WordNet graph

Vertex = synset; edge = hypernym relationship.



<http://wordnet.princeton.edu>

The McChrystal Afghanistan PowerPoint slide

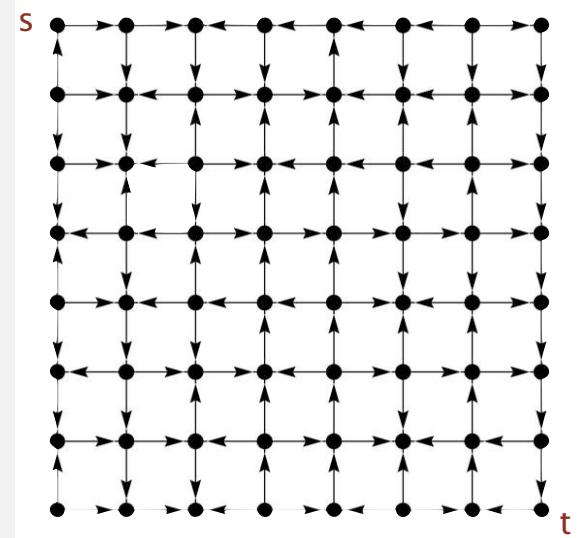


Digraph applications

digraph	vertex	directed edge
transportation	street intersection	one-way street
web	web page	hyperlink
food web	species	predator-prey relationship
WordNet	synset	hypernym
scheduling	task	precedence constraint
financial	bank	transaction
cell phone	person	placed call
infectious disease	person	infection
game	board position	legal move
citation	journal article	citation
object graph	object	pointer
inheritance hierarchy	class	inherits from
control flow	code block	jump

Some digraph problems

Path. Is there a directed path from s to t ?



Shortest path. What is the shortest directed path from s to t ?

Topological sort. Can you draw a digraph so that all edges point upwards?

Strong connectivity. Is there a directed path between all pairs of vertices?

Transitive closure. For which vertices v and w is there a path from v to w ?

PageRank. What is the importance of a web page?

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *digraph search*
- ▶ *topological sort*
- ▶ *strong components*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

- ▶ *introduction*
- ▶ ***digraph API***
- ▶ *digraph search*
- ▶ *topological sort*
- ▶ *strong components*

Digraph API

public class Digraph	
Digraph(int V)	<i>create an empty digraph with V vertices</i>
Digraph(In in)	<i>create a digraph from input stream</i>
void addEdge(int v, int w)	<i>add a directed edge $v \rightarrow w$</i>
Iterable<Integer> adj(int v)	<i>vertices pointing from v</i>
int V()	<i>number of vertices</i>
int E()	<i>number of edges</i>
Digraph reverse()	<i>reverse of this digraph</i>
String toString()	<i>string representation</i>

```
In in = new In(args[0]);
Digraph G = new Digraph(in);
```

← read digraph from
input stream

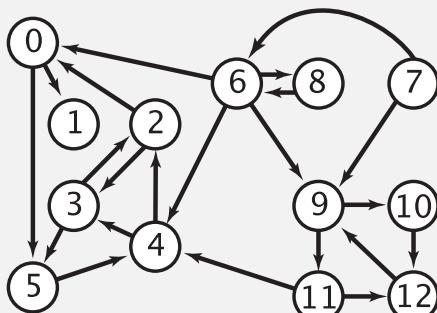
```
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);
```

← print out each
edge (once)

Digraph API

tinyDG.txt
V → 13
E ← 22

4 2
2 3
3 2
6 0
0 1
2 0
11 12
12 9
9 10
9 11
7 9
10 12
11 4
4 3
3 5
6 8
8 6
:



```
% java Digraph tinyDG.txt
0->5
0->1
2->0
2->3
3->5
3->2
4->3
4->2
5->4
:
11->4
11->12
12->9
```

```
In in = new In(args[0]);
Digraph G = new Digraph(in);
```

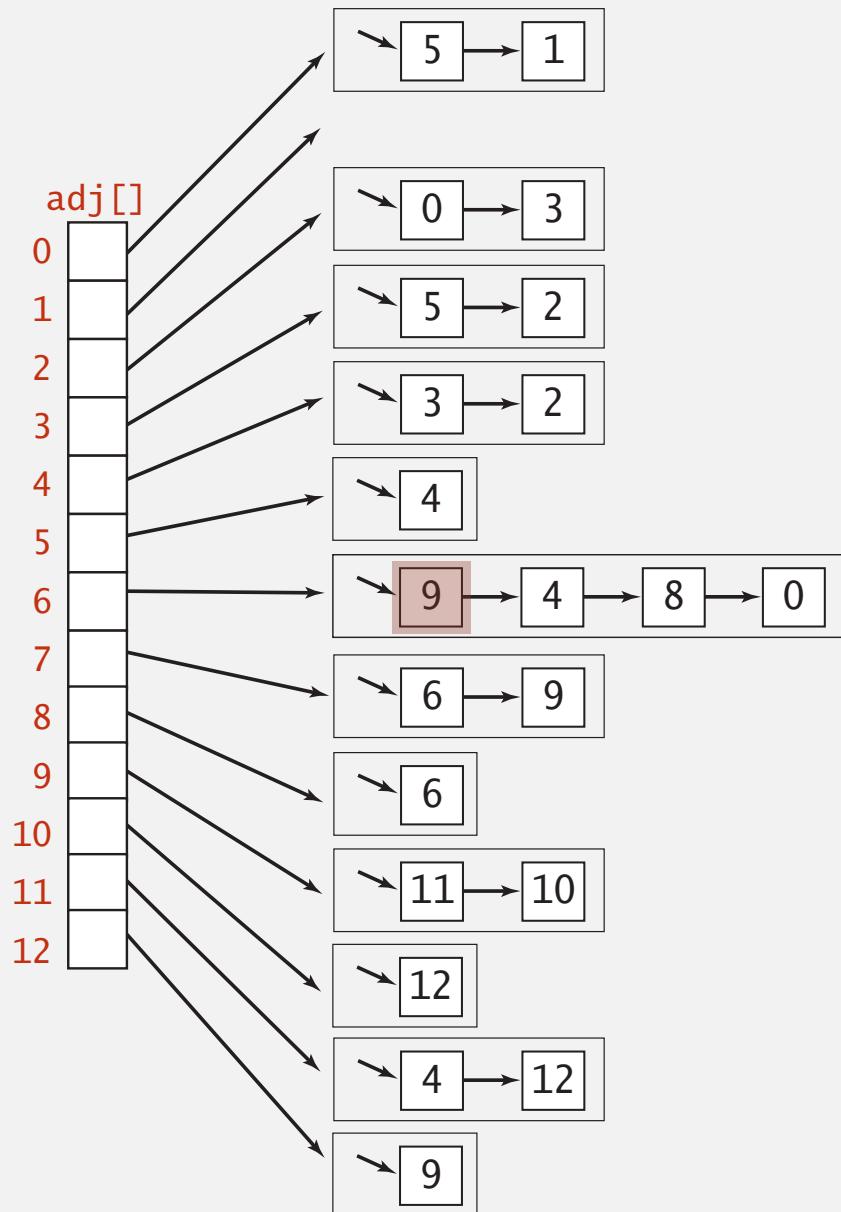
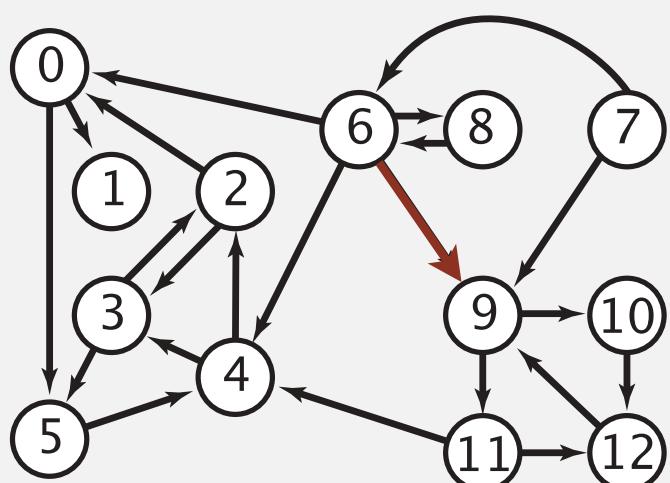
read digraph from
input stream

```
for (int v = 0; v < G.V(); v++)
    for (int w : G.adj(v))
        StdOut.println(v + "->" + w);
```

print out each
edge (once)

Adjacency-lists digraph representation

Maintain vertex-indexed array of lists.



Adjacency-lists graph representation (review): Java implementation

```
public class Graph
{
    private final int V;
    private final Bag<Integer>[] adj; ← adjacency lists

    public Graph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w) ← add edge v-w
    {
        adj[v].add(w);
        adj[w].add(v);
    }

    public Iterable<Integer> adj(int v) ← iterator for vertices
    {   return adj[v];  }
}
```

Adjacency-lists digraph representation: Java implementation

```
public class Digraph
{
    private final int V;
    private final Bag<Integer>[] adj;           ← adjacency lists

    public Digraph(int V)
    {
        this.V = V;
        adj = (Bag<Integer>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<Integer>();
    }

    public void addEdge(int v, int w)           ← add edge v→w
    {
        adj[v].add(w);
    }

    public Iterable<Integer> adj(int v)         ← iterator for vertices
    {   return adj[v];  }
}
```

Digraph representations

In practice. Use adjacency-lists representation.

- Algorithms based on iterating over vertices pointing from v .
- Real-world digraphs tend to be sparse.

huge number of vertices,
small average vertex degree

representation	space	insert edge from v to w	edge from v to w ?	iterate over vertices pointing from v ?
list of edges	E	1	E	E
adjacency matrix	V^2	1^\dagger	1	V
adjacency lists	$E + V$	1	outdegree(v)	outdegree(v)

Bac ra cua v

Bac ra cua v
† disallows parallel edges

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

- ▶ *introduction*
- ▶ ***digraph API***
- ▶ *digraph search*
- ▶ *topological sort*
- ▶ *strong components*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

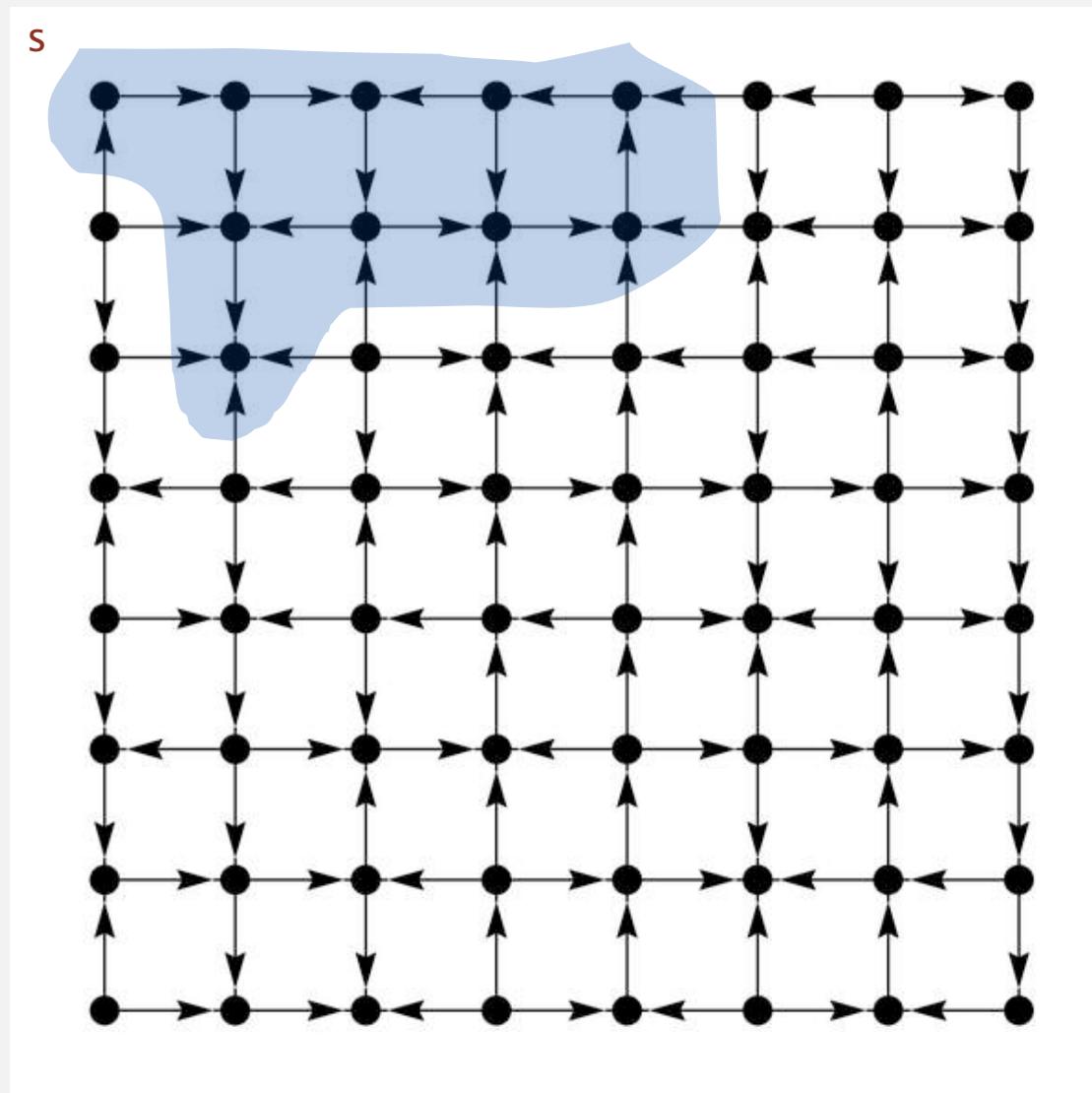
<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

- ▶ *introduction*
- ▶ *digraph API*
- ▶ ***digraph search***
- ▶ *topological sort*
- ▶ *strong components*

Reachability

Problem. Find all vertices reachable from s along a directed path.



Depth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- DFS is a **digraph** algorithm.

DFS (to visit a vertex v)

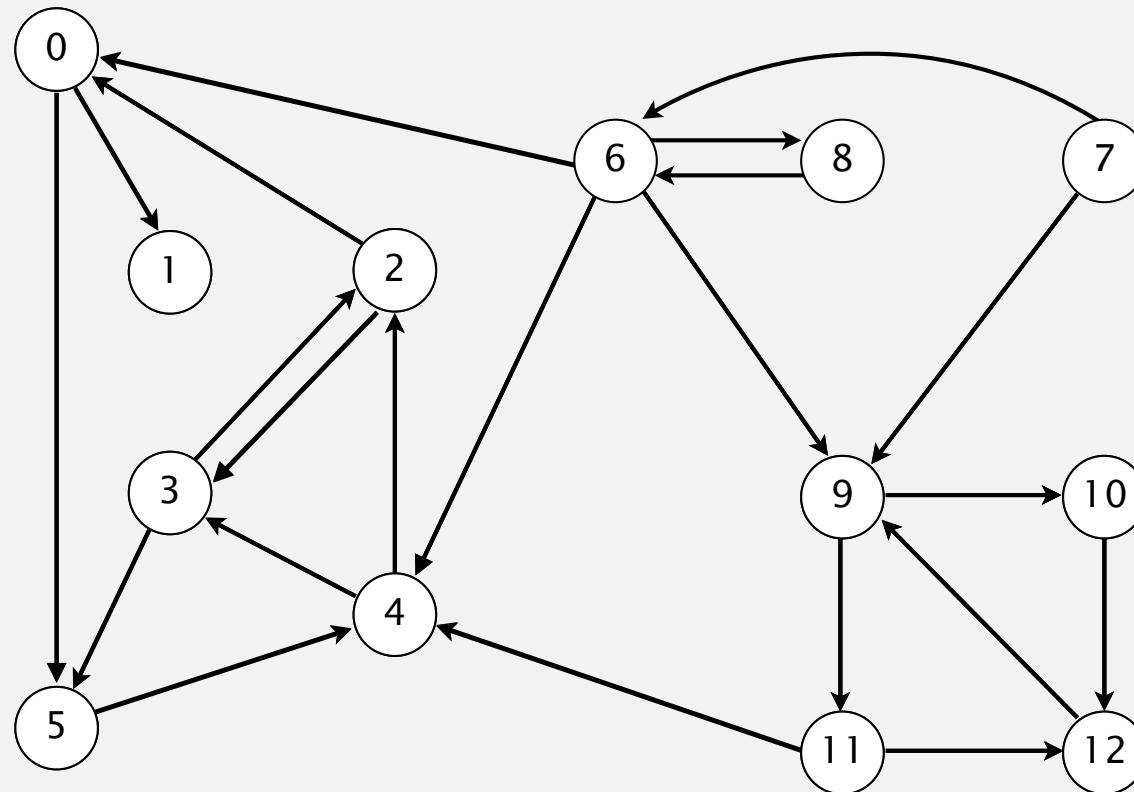
Mark v as visited.

Recursively visit all unmarked
vertices w pointing from v.

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v .



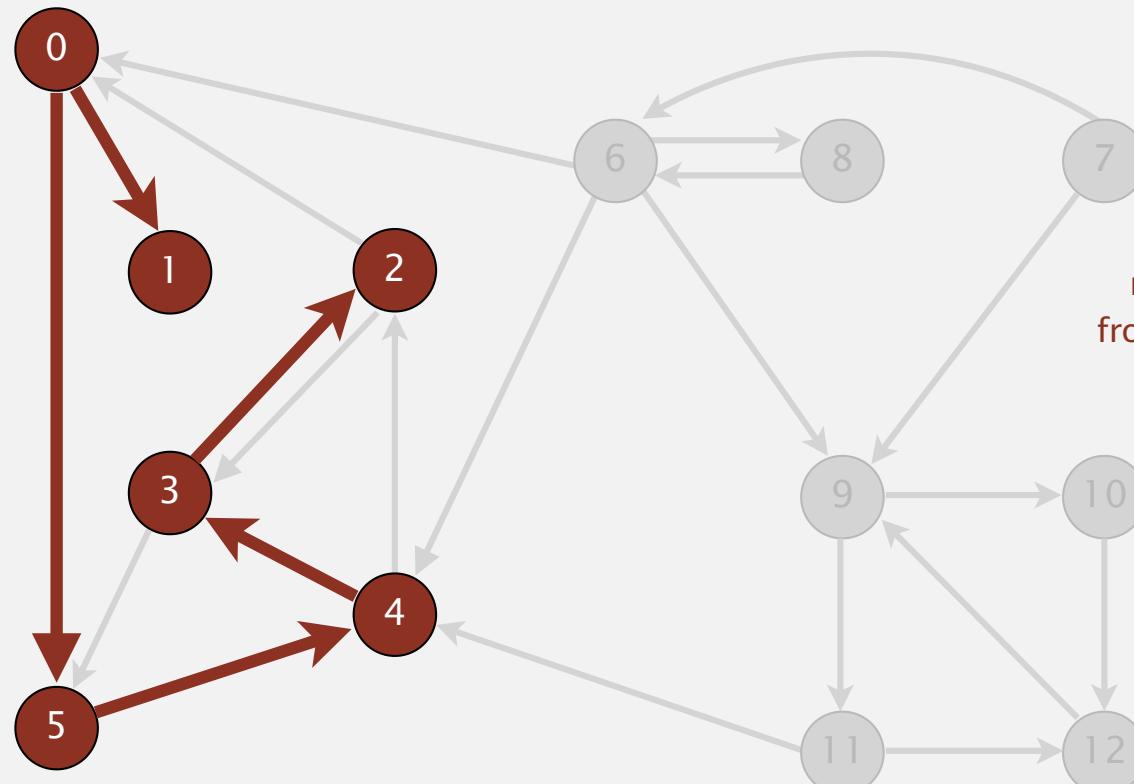
a directed graph

4→2
2→3
3→2
6→0
0→1
2→0
11→12
12→9
9→10
9→11
8→9
10→12
11→4
4→3
3→5
6→8
8→6
5→4
0→5
6→4
6→9
7→6

Depth-first search demo

To visit a vertex v :

- Mark vertex v as visited.
- Recursively visit all unmarked vertices pointing from v .



v	marked[]	edgeTo[]
0	T	-
1	T	0
2	T	3
3	T	4
4	T	5
5	T	0
6	F	-
7	F	-
8	F	-
9	F	-
10	F	-
11	F	-
12	F	-

reachable from 0

Depth-first search (in undirected graphs)

Recall code for **undirected** graphs.

```
public class DepthFirstSearch
{
    private boolean[] marked;           ← true if path to s

    public DepthFirstSearch(Graph G, int s)
    {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private void dfs(Graph G, int v)     ← recursive DFS does the work
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }

    public boolean visited(int v)        ← client can ask whether any
    {   return marked[v]; }             vertex is connected to s
}
```

Depth-first search (in directed graphs)

Code for **directed** graphs identical to undirected one.
[substitute Digraph for Graph]

```
public class DirectedDFS
{
    private boolean[] marked;           ← true if path from s

    public DirectedDFS(Digraph G, int s)
    {
        marked = new boolean[G.V()];
        dfs(G, s);
    }

    private void dfs(Digraph G, int v)   ← recursive DFS does the work
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
    }

    public boolean visited(int v)       ← client can ask whether any
    {   return marked[v]; }             vertex is reachable from s
}
```

Reachability application: program control-flow analysis

Every program is a digraph.

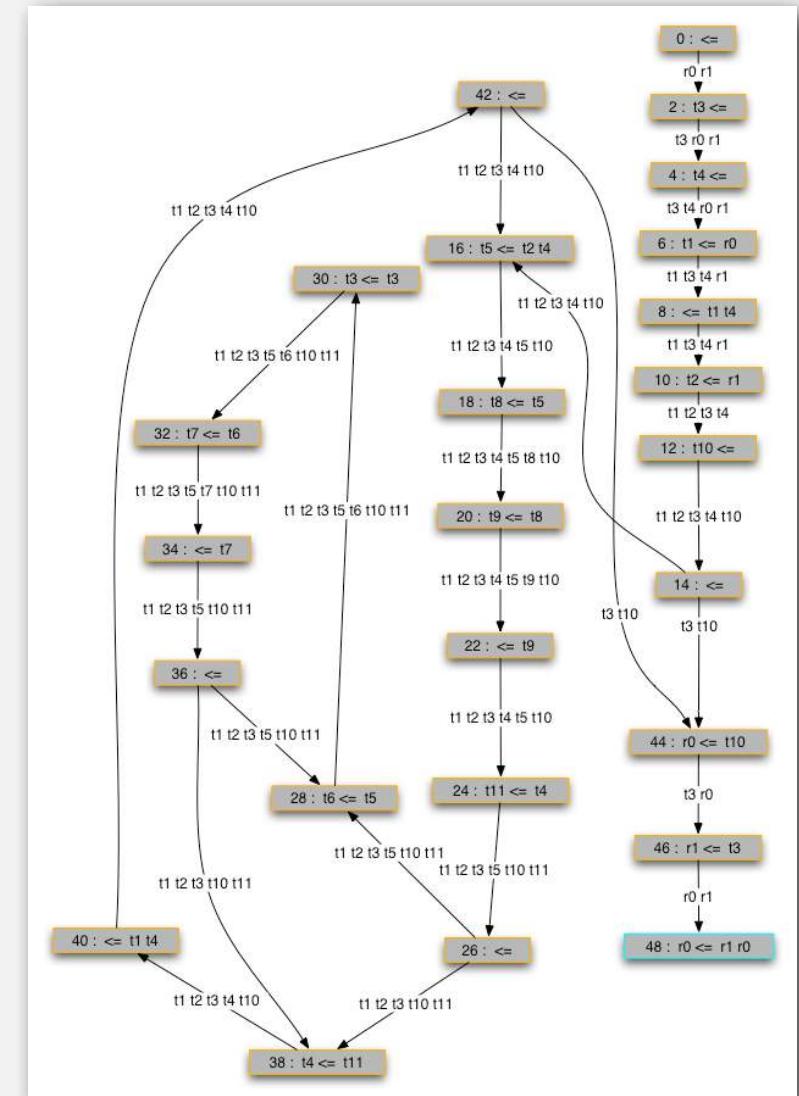
- Vertex = basic block of instructions (straight-line program).
- Edge = jump.

Dead-code elimination.

Find (and remove) unreachable code.

Infinite-loop detection.

Determine whether exit is unreachable.



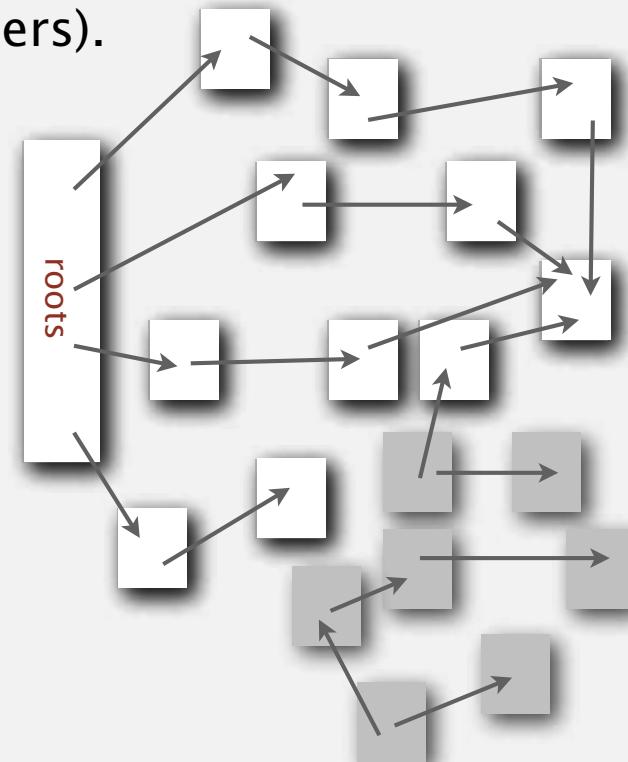
Reachability application: mark-sweep garbage collector

Every data structure is a digraph.

- Vertex = object.
- Edge = reference.

Roots. Objects known to be directly accessible by program (e.g., stack).

Reachable objects. Objects indirectly accessible by program (starting at a root and following a chain of pointers).

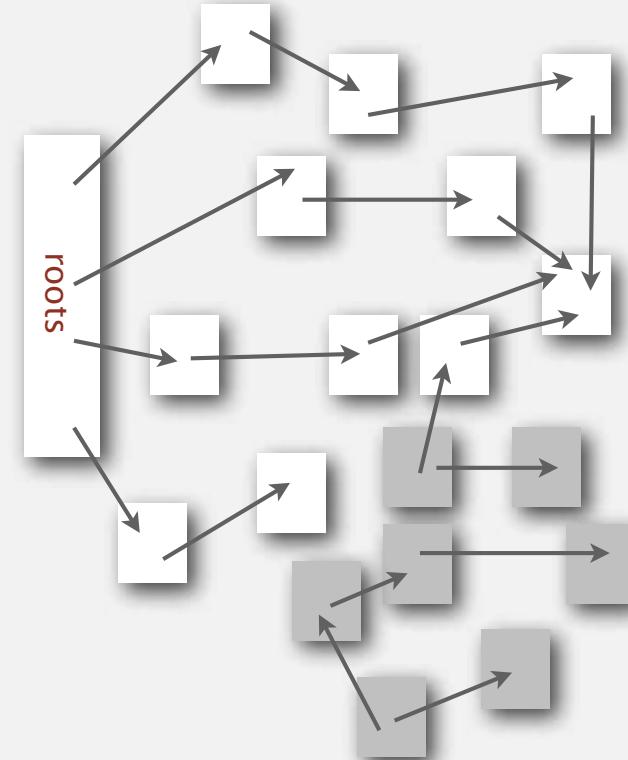


Reachability application: mark-sweep garbage collector

Mark-sweep algorithm. [McCarthy, 1960]

- Mark: mark all reachable objects.
- Sweep: if object is unmarked, it is garbage (so add to free list).

Memory cost. Uses 1 extra mark bit per object (plus DFS stack).



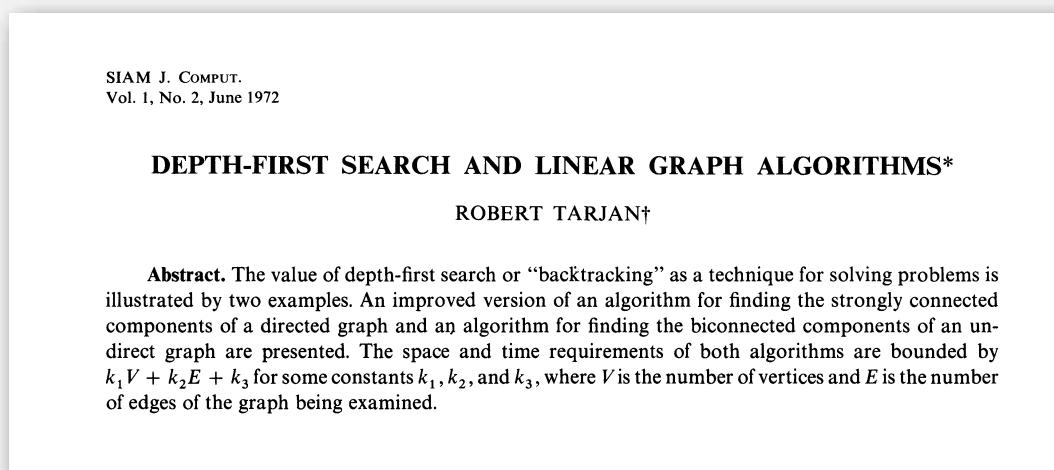
Depth-first search in digraphs summary

DFS enables direct solution of simple digraph problems.

- ✓ • Reachability.
- Path finding.
- Topological sort.
- Directed cycle detection.

Basis for solving difficult digraph problems.

- 2-satisfiability.
- Directed Euler path.
- Strongly-connected components.



Breadth-first search in digraphs

Same method as for undirected graphs.

- Every undirected graph is a digraph (with edges in both directions).
- BFS is a **digraph** algorithm.

BFS (from source vertex s)

Put s onto a FIFO queue, and mark s as visited.

Repeat until the queue is empty:

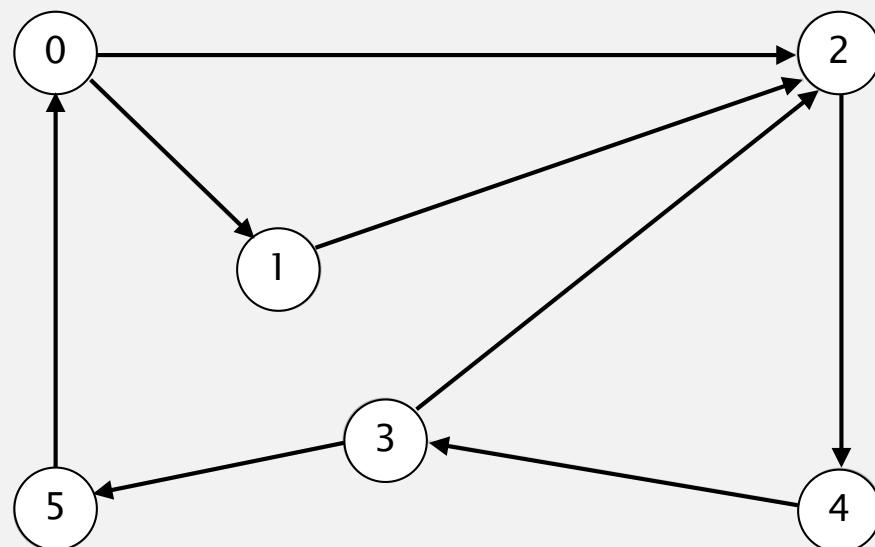
- **remove the least recently added vertex v**
- **for each unmarked vertex pointing from v:**
add to queue and mark as visited.

Proposition. BFS computes shortest paths (fewest number of edges) from s to all other vertices in a digraph in time proportional to $E + V$.

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



tinyDG2.txt

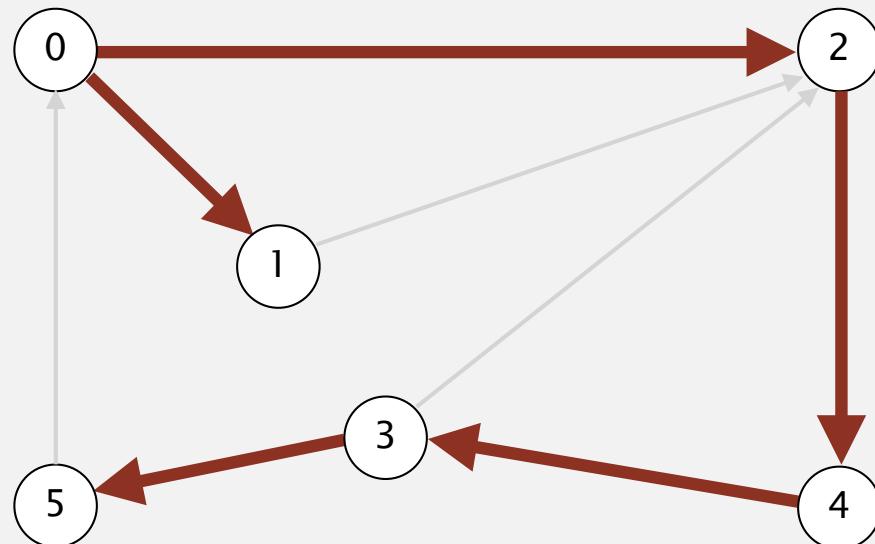
V → 6
E → 8
5 0
2 4
3 2
1 2
0 1
4 3
3 5
0 2

graph G

Directed breadth-first search demo

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices pointing from v and mark them.



v	edgeTo[]	distTo[]
0	-	0
1	0	1
2	0	1
3	4	3
4	2	2
5	3	4

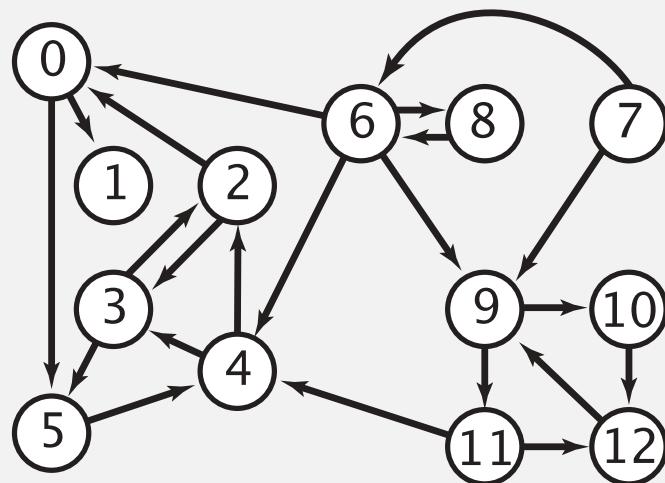
done

Multiple-source shortest paths

Multiple-source shortest paths. Given a digraph and a **set** of source vertices, find shortest path from any vertex in the set to each other vertex.

Ex. $S = \{ 1, 7, 10 \}$.

- Shortest path to 4 is $7 \rightarrow 6 \rightarrow 4$.
- Shortest path to 5 is $7 \rightarrow 6 \rightarrow 0 \rightarrow 5$.
- Shortest path to 12 is $10 \rightarrow 12$.
- ...



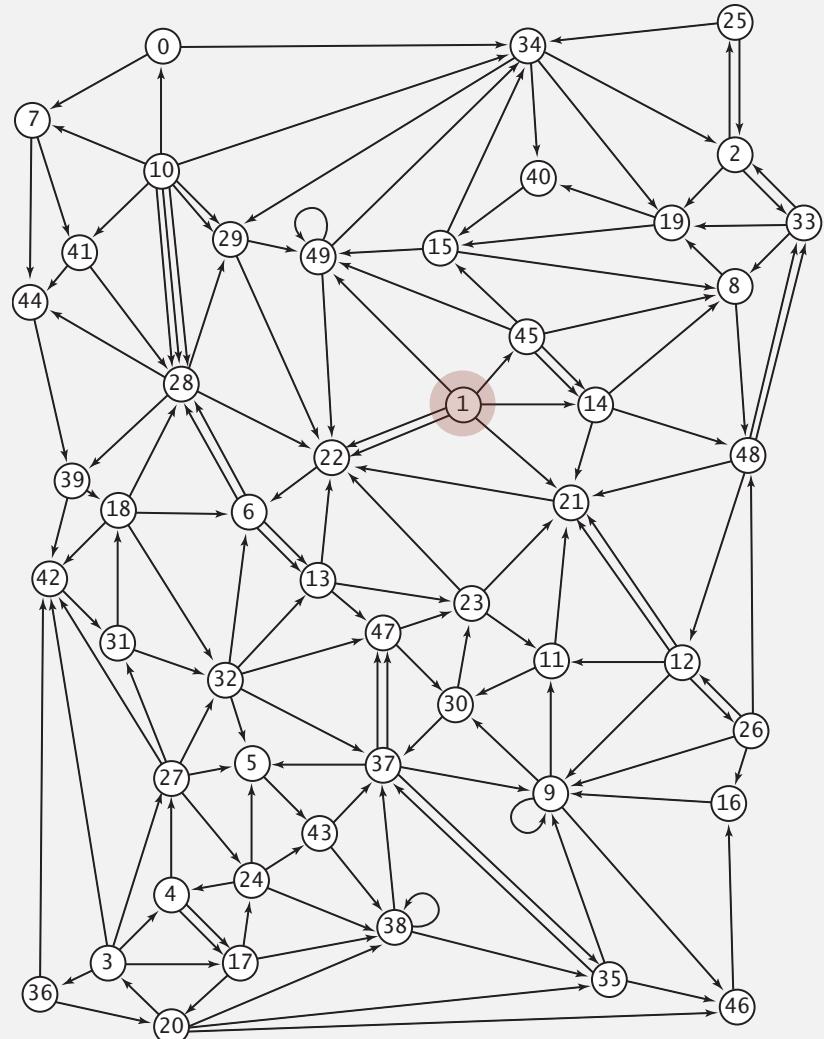
- Q. How to implement multi-source shortest paths algorithm?
A. Use BFS, but initialize by enqueueing all source vertices.

Breadth-first search in digraphs application: web crawler

Goal. Crawl web, starting from some root web page, say `www.princeton.edu`.

Solution. [BFS with implicit digraph]

- Choose root web page as source s .
- Maintain a Queue of websites to explore.
- Maintain a SET of discovered websites.
- Dequeue the next website and enqueue websites to which it links
(provided you haven't done so before).



Q. Why not use DFS?

Bare-bones web crawler: Java implementation

```
Queue<String> queue = new Queue<String>();           ← queue of websites to crawl
SET<String> marked = new SET<String>();             ← set of marked websites

String root = "http://www.princeton.edu";
queue.enqueue(root);
marked.add(root);

while (!queue.isEmpty())
{
    String v = queue.dequeue();
    StdOut.println(v);
    In in = new In(v);
    String input = in.readAll();

    String regexp = "http://(\w+\.\w+)*(\w+)";
    Pattern pattern = Pattern.compile(regexp);
    Matcher matcher = pattern.matcher(input);           ← use regular expression to find all URLs
                                                       in website of form http://xxx.yyy.zzz
                                                       [crude pattern misses relative URLs]

    while (matcher.find())
    {
        String w = matcher.group();
        if (!marked.contains(w))
        {
            marked.add(w);
            queue.enqueue(w);                           ← if unmarked, mark it and put
                                                       on the queue
        }
    }
}
```

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

- ▶ *introduction*
- ▶ *digraph API*
- ▶ ***digraph search***
- ▶ *topological sort*
- ▶ *strong components*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *digraph search*
- ▶ ***topological sort***
- ▶ *strong components*

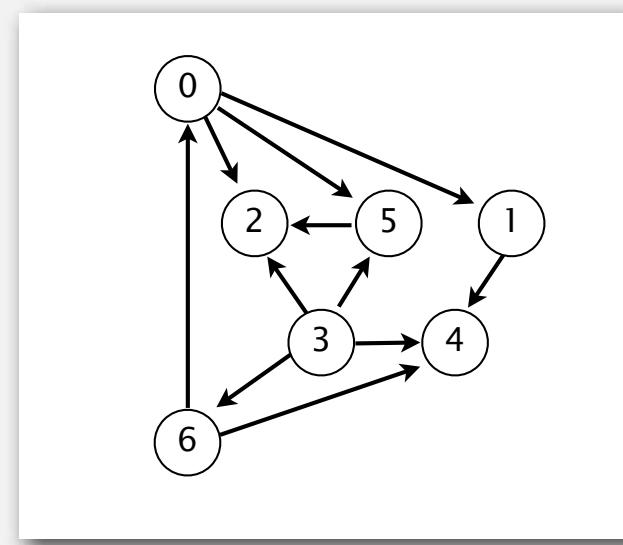
Precedence scheduling

Goal. Given a set of tasks to be completed with precedence constraints, in which order should we schedule the tasks?

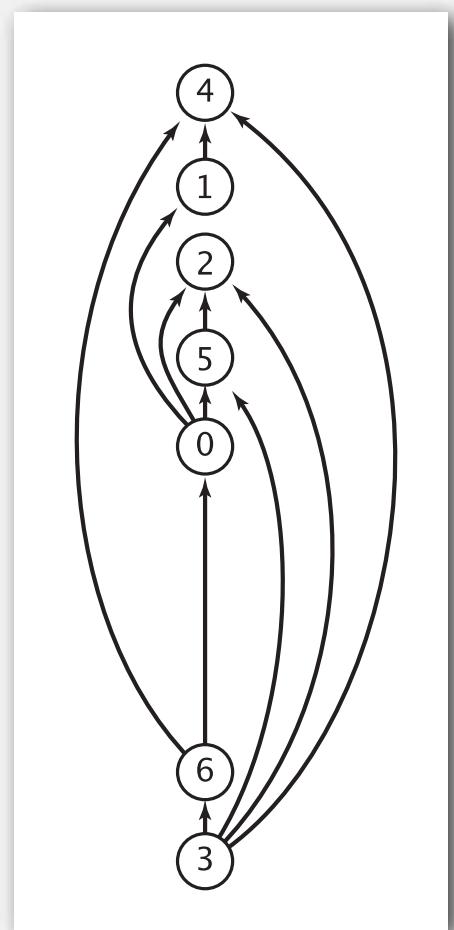
Digraph model. vertex = task; edge = precedence constraint.

- 0. Algorithms
- 1. Complexity Theory
- 2. Artificial Intelligence
- 3. Intro to CS
- 4. Cryptography
- 5. Scientific Computing
- 6. Advanced Programming

tasks



precedence constraint graph



feasible schedule

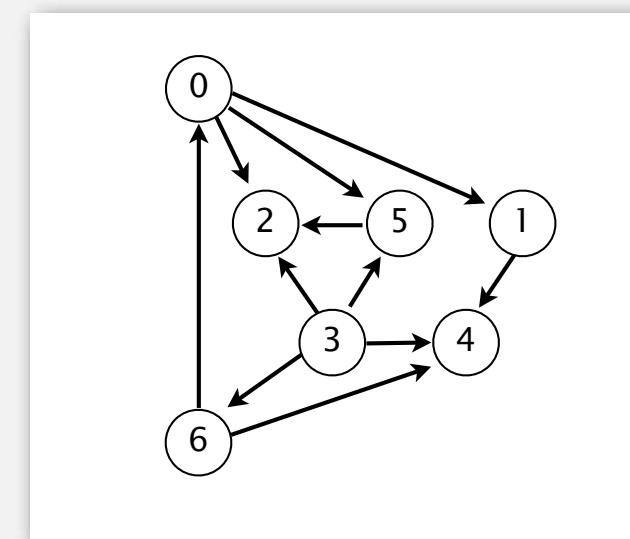
Topological sort

DAG. Directed acyclic graph.

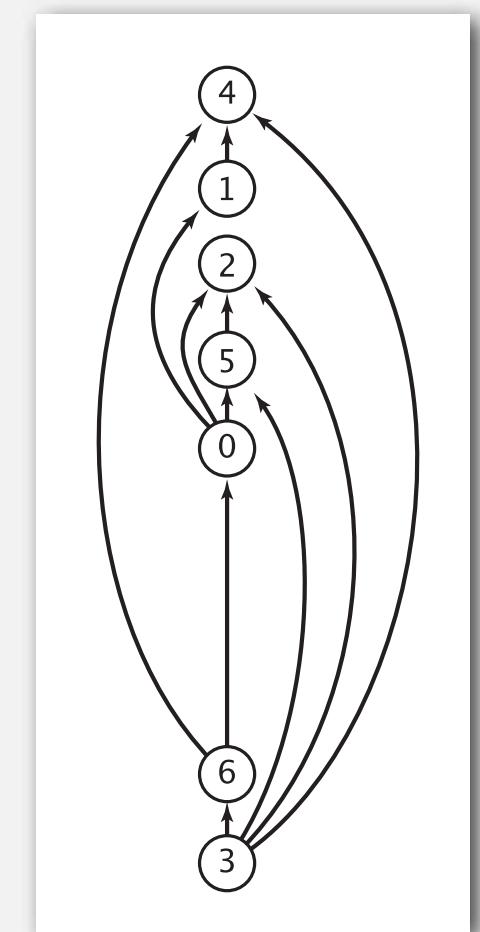
Topological sort. Redraw DAG so all edges point upwards.

$0 \rightarrow 5$	$0 \rightarrow 2$
$0 \rightarrow 1$	$3 \rightarrow 6$
$3 \rightarrow 5$	$3 \rightarrow 4$
$5 \rightarrow 2$	$6 \rightarrow 4$
$6 \rightarrow 0$	$3 \rightarrow 2$
$1 \rightarrow 4$	

directed edges



DAG

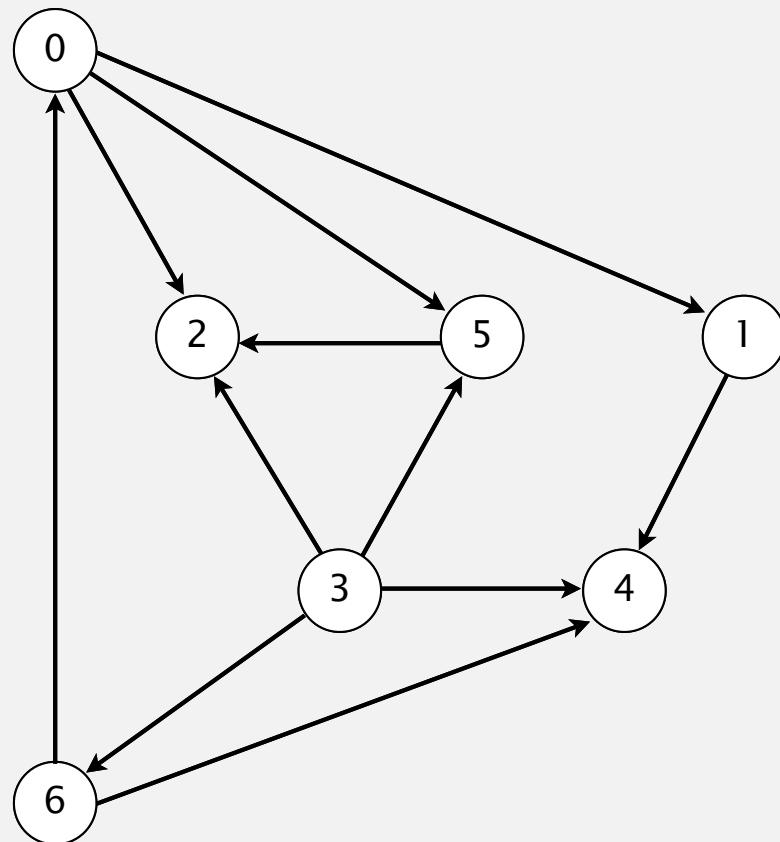


topological order

Solution. DFS. What else?

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.

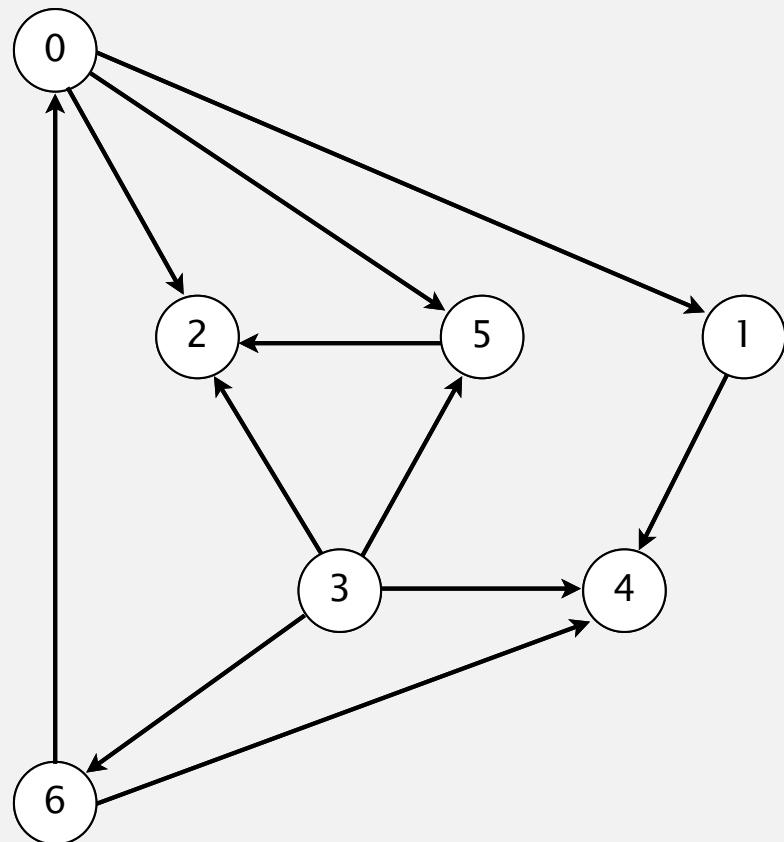


$0 \rightarrow 5$
 $0 \rightarrow 2$
 $0 \rightarrow 1$
 $3 \rightarrow 6$
 $3 \rightarrow 5$
 $3 \rightarrow 4$
 $5 \rightarrow 2$
 $6 \rightarrow 4$
 $6 \rightarrow 0$
 $3 \rightarrow 2$
 $1 \rightarrow 4$

a directed acyclic graph

Topological sort demo

- Run depth-first search.
- Return vertices in reverse postorder.



4 1 2 \Rightarrow 0 6 3
postorder

4 1 2 5 0 6 3

topological order

3 6 0 5 2 1 4

done

Depth-first search order

```
public class DepthFirstOrder
{
    private boolean[] marked;
    private Stack<Integer> reversePost;

    public DepthFirstOrder(Digraph G)
    {
        reversePost = new Stack<Integer>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G, v);
    }

    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G, w);
        reversePost.push(v);
    }

    public Iterable<Integer> reversePost() ←
    {   return reversePost;  }
}
```

returns all vertices in
“reverse DFS postorder”

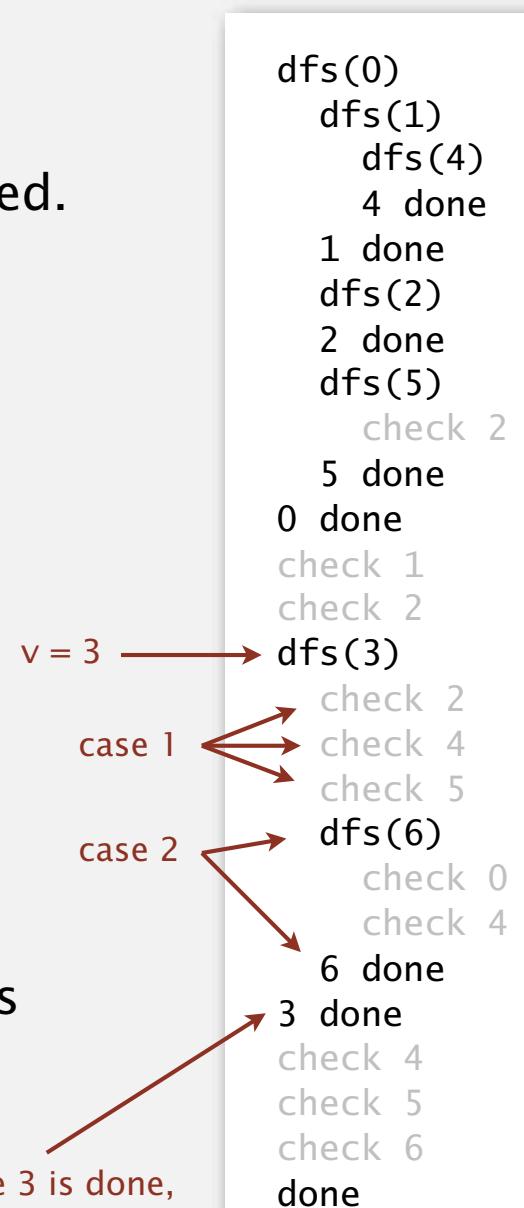
Topological sort in a DAG: correctness proof

Proposition. Reverse DFS postorder of a DAG is a topological order.

Pf. Consider any edge $v \rightarrow w$. When $\text{dfs}(v)$ is called:

- Case 1: $\text{dfs}(w)$ has already been called and returned.
Thus, w was done before v .
- Case 2: $\text{dfs}(w)$ has not yet been called.
 $\text{dfs}(w)$ will get called directly or indirectly
by $\text{dfs}(v)$ and will finish before $\text{dfs}(v)$.
Thus, w will be done before v .
- Case 3: $\text{dfs}(w)$ has already been called,
but has not yet returned.
Can't happen in a DAG: function call stack contains
path from w to v , so $v \rightarrow w$ would complete a cycle.

all vertices pointing from 3 are done before 3 is done,
so they appear after 3 in topological order

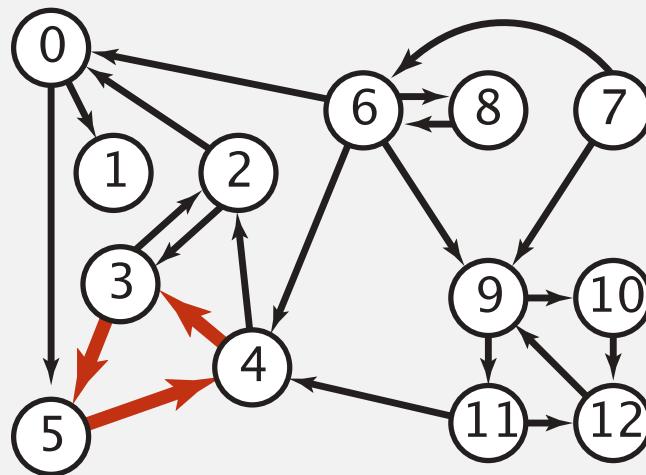


Directed cycle detection

Proposition. A digraph has a topological order iff no directed cycle.

Pf.

- If directed cycle, topological order impossible.
- If no directed cycle, DFS-based algorithm finds a topological order.



Goal. Given a digraph, find a directed cycle.

Solution. DFS. What else? See textbook.

Directed cycle detection application: precedence scheduling

Scheduling. Given a set of tasks to be completed with precedence constraints, in what order should we schedule the tasks?

PAGE 3

DEPARTMENT	COURSE	DESCRIPTION	PREREQS
COMPUTER SCIENCE	CPSC 432	INTERMEDIATE COMPILER DESIGN, WITH A FOCUS ON DEPENDENCY RESOLUTION.	CPSC 432

<http://xkcd.com/754>

Remark. A directed cycle implies scheduling problem is infeasible.

Directed cycle detection application: cyclic inheritance

The Java compiler does cycle detection.

```
public class A extends B
{
    ...
}
```

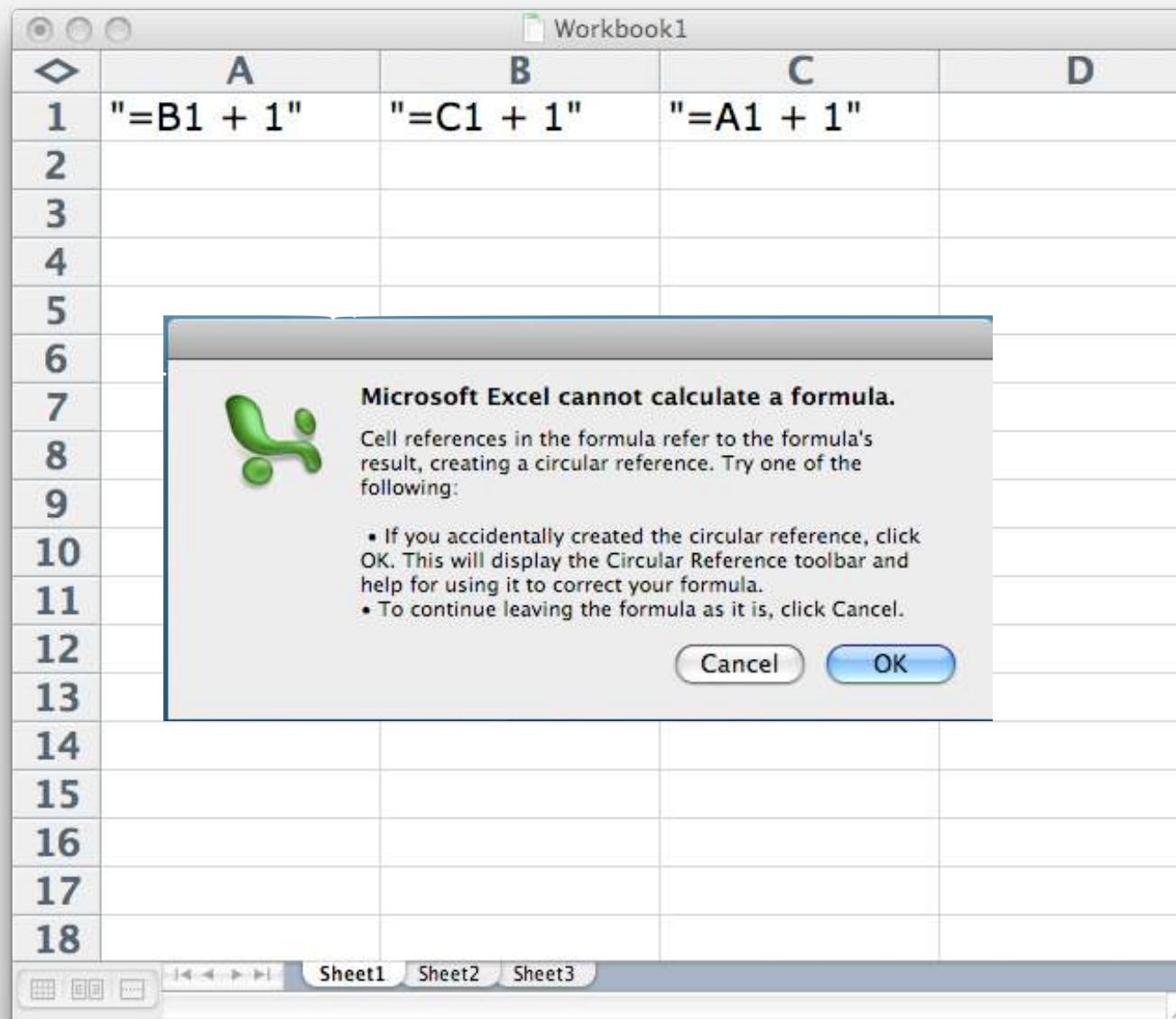
```
public class B extends C
{
    ...
}
```

```
public class C extends A
{
    ...
}
```

```
% javac A.java
A.java:1: cyclic inheritance
involving A
public class A extends B { }
^
1 error
```

Directed cycle detection application: spreadsheet recalculation

Microsoft Excel does cycle detection (and has a circular reference toolbar!)





Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *digraph search*
- ▶ ***topological sort***
- ▶ *strong components*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *digraph search*
- ▶ *topological sort*
- ▶ ***strong components***

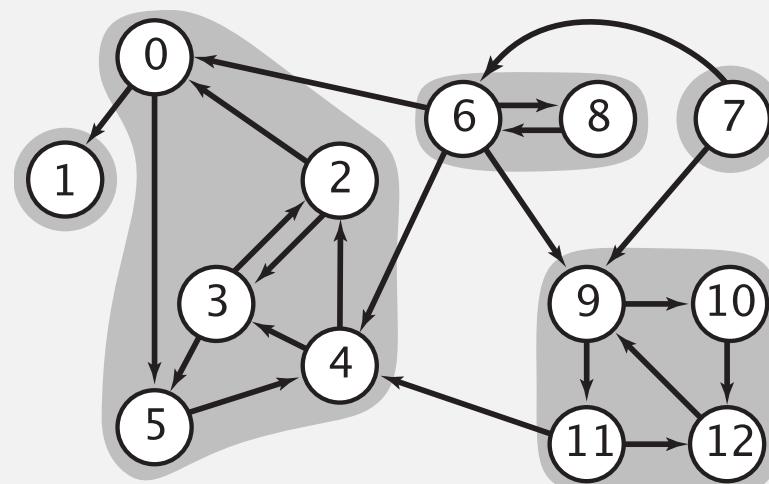
Strongly-connected components

Def. Vertices v and w are **strongly connected** if there is both a directed path from v to w **and** a directed path from w to v .

Key property. Strong connectivity is an **equivalence relation**:

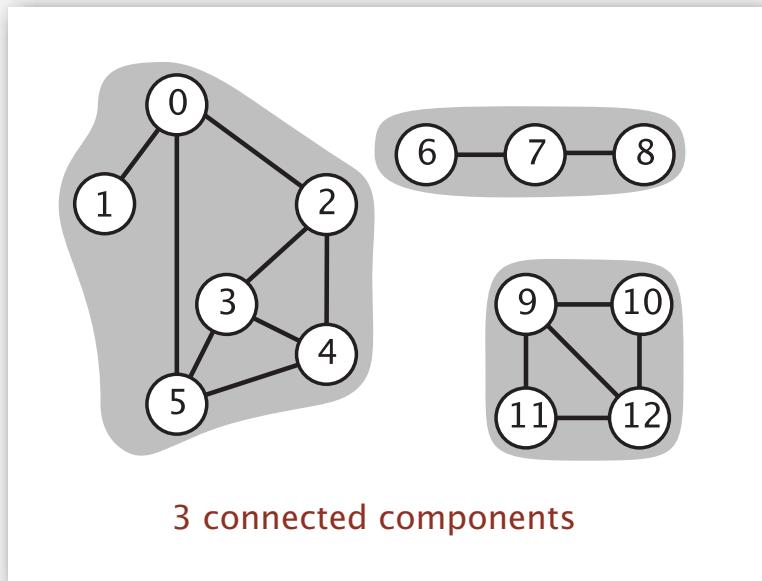
- v is strongly connected to v .
- If v is strongly connected to w , then w is strongly connected to v .
- If v is strongly connected to w and w to x , then v is strongly connected to x .

Def. A **strong component** is a maximal subset of strongly-connected vertices.

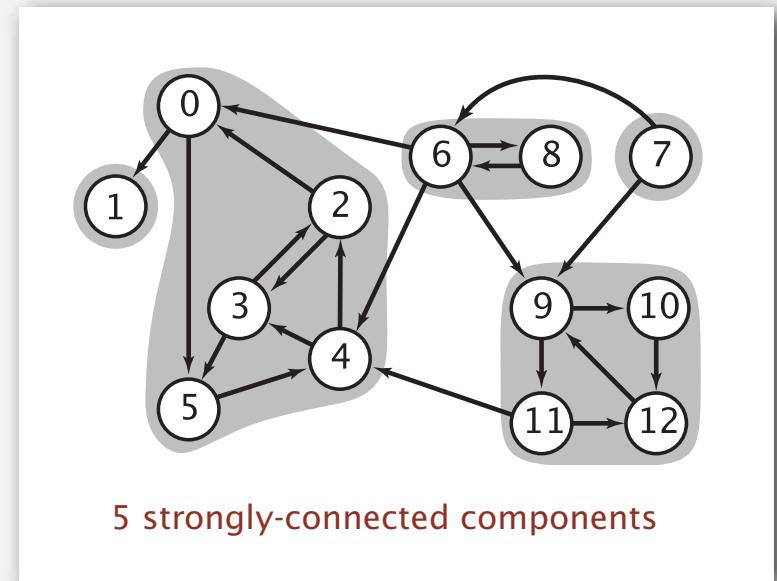


Connected components vs. strongly-connected components

v and w are **connected** if there is a path between v and w



v and w are **strongly connected** if there is both a directed path from v to w and a directed path from w to v



connected component id (easy to compute with DFS)

cc[0]	0	1	2	3	4	5	6	7	8	9	10	11	12
cc[1]	0	0	0	0	0	0	1	1	1	2	2	2	2

```
public int connected(int v, int w)
{   return cc[v] == cc[w]; }
```

constant-time client connectivity query

strongly-connected component id (how to compute?)

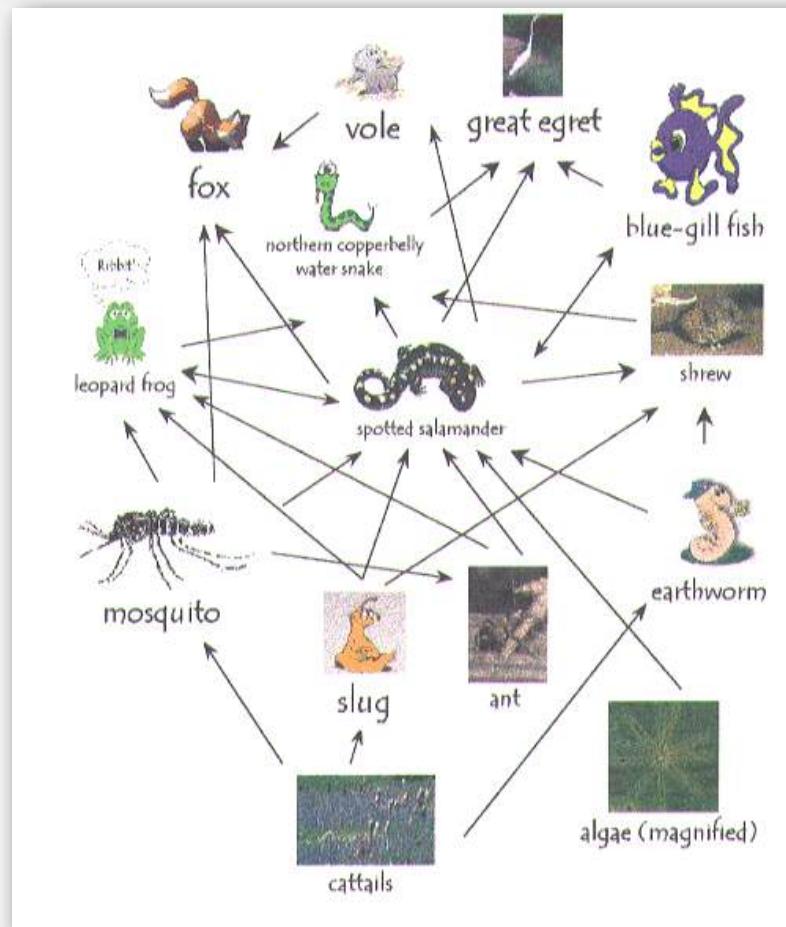
scc[0]	0	1	2	3	4	5	6	7	8	9	10	11	12
scc[1]	1	0	1	1	1	1	3	4	3	2	2	2	2

```
public int stronglyConnected(int v, int w)
{   return scc[v] == scc[w]; }
```

constant-time client strong-connectivity query

Strong component application: ecological food webs

Food web graph. Vertex = species; edge = from producer to consumer.



<http://www.twinklives.district96.k12.il.us/Wetlands/Salamander/SalGraphics/salfoodweb.gif>

Strong component. Subset of species with common energy flow.

Strong components algorithms: brief history

1960s: Core OR problem.

- Widely studied; some practical algorithms.
- Complexity not understood.

1972: linear-time DFS algorithm (Tarjan).

- Classic algorithm.
- Level of difficulty: Algs4++.
- Demonstrated broad applicability and importance of DFS.

1980s: easy two-pass linear-time algorithm (Kosaraju-Sharir).

- Forgot notes for lecture; developed algorithm in order to teach it!
- Later found in Russian scientific literature (1972).

1990s: more easy linear-time algorithms.

- Gabow: fixed old OR algorithm.
- Cheriyan-Mehlhorn: needed one-pass algorithm for LEDA.

Kosaraju-Sharir algorithm: intuition

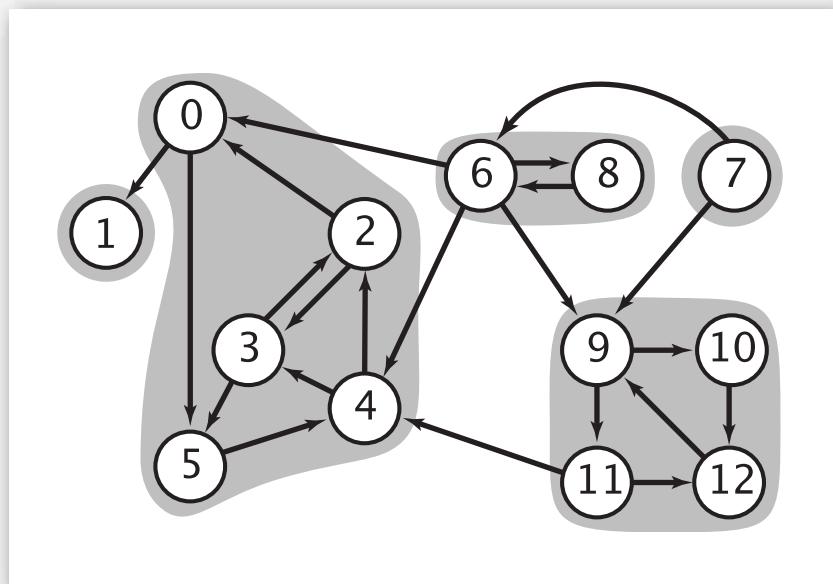
Reverse graph. Strong components in G are same as in G^R .

Kernel DAG. Contract each strong component into a single vertex.

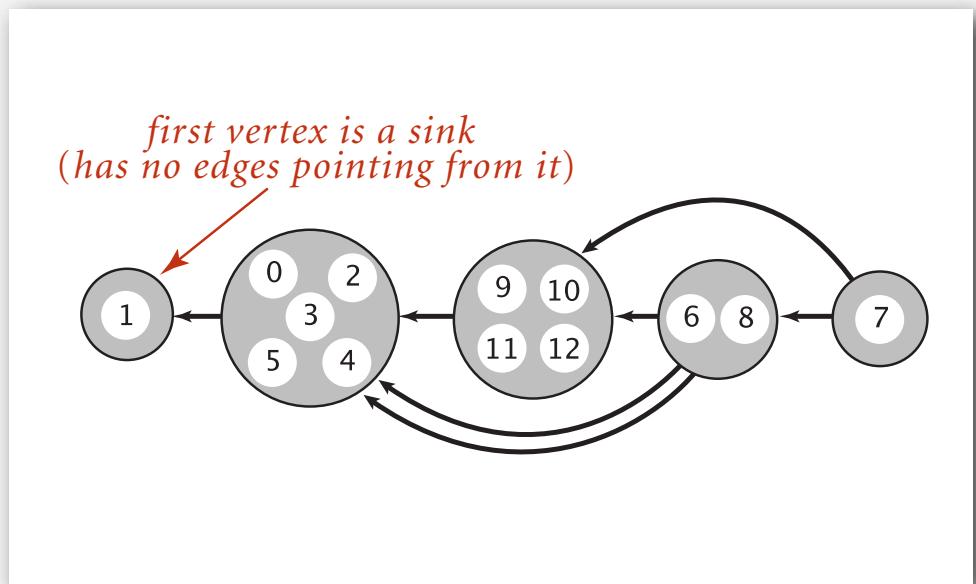
Idea.

- Compute topological order (reverse postorder) in kernel DAG.
- Run DFS, considering vertices in reverse topological order.

how to compute?



digraph G and its strong components

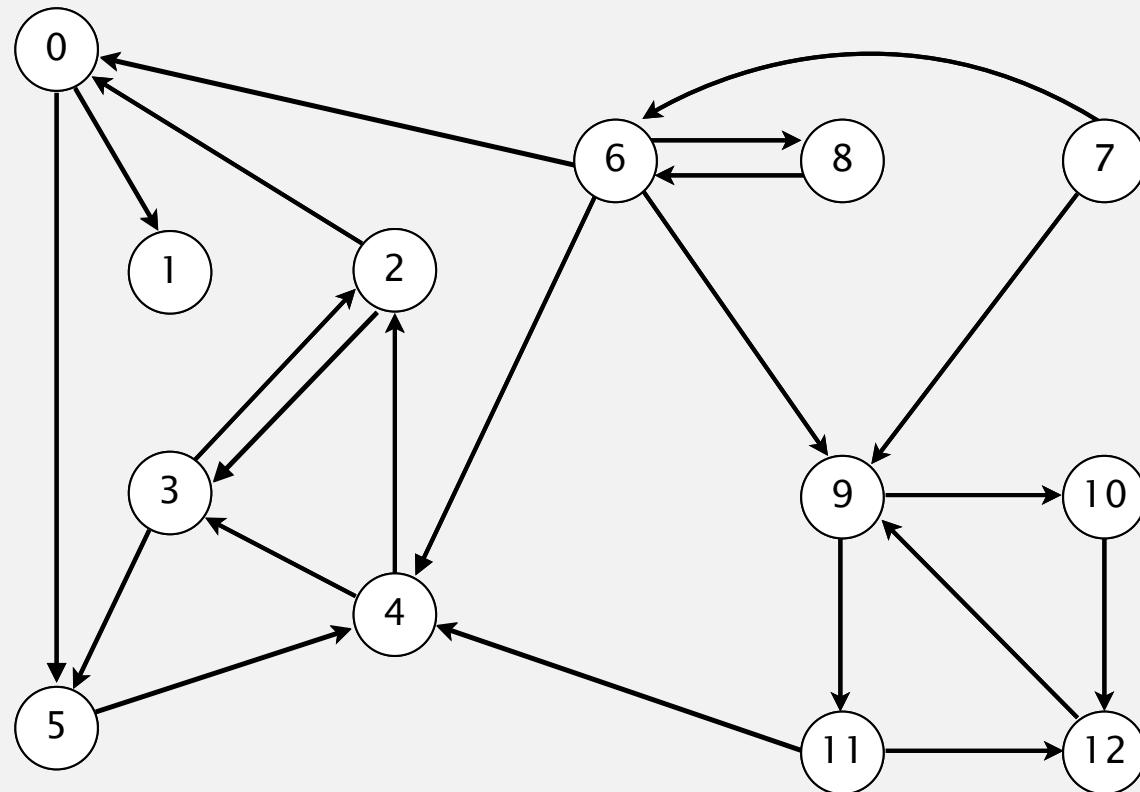


kernel DAG of G (in reverse topological order)

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R .

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

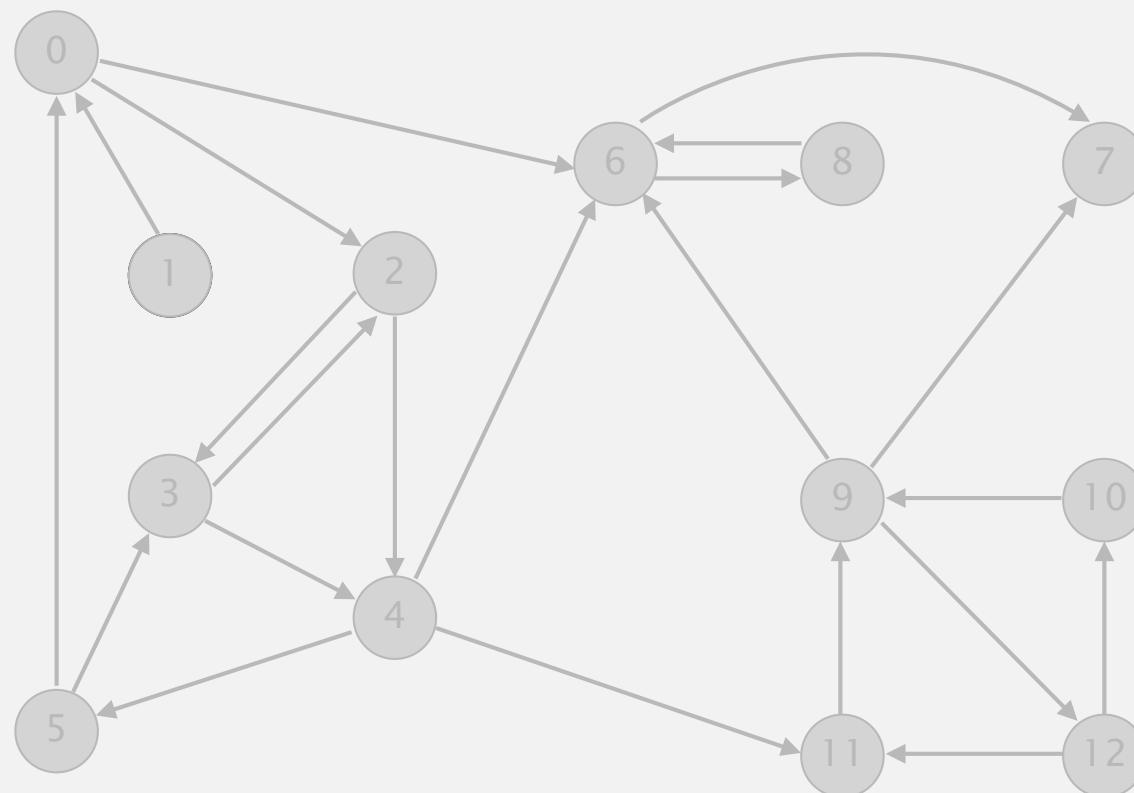


digraph G

Kosaraju-Sharir algorithm demo

Phase 1. Compute reverse postorder in G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8

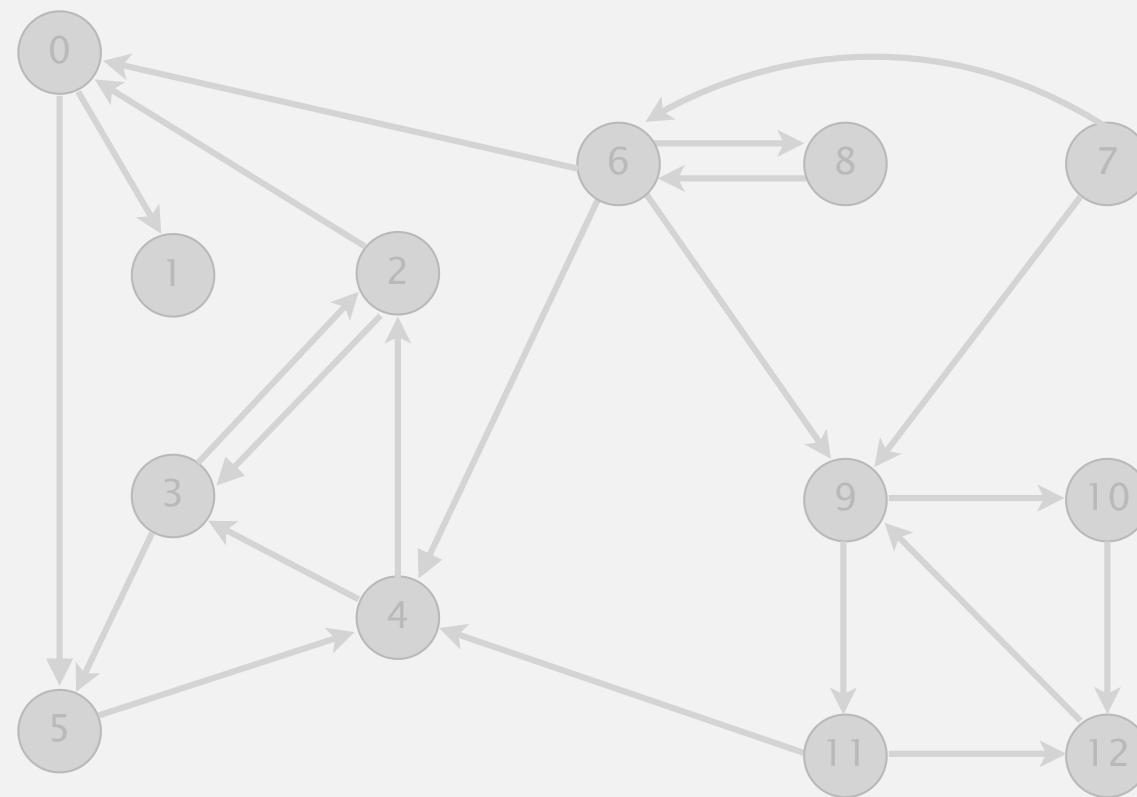


reverse digraph G^R

Kosaraju-Sharir algorithm demo

Phase 2. Run DFS in G , visiting unmarked vertices in reverse postorder of G^R .

1 0 2 4 5 3 11 9 12 10 6 7 8



v	scc[]
0	1
1	0
2	1
3	1
4	1
5	3
6	4
7	4
8	3
9	2
10	2
11	2
12	2

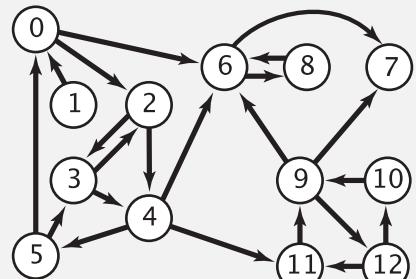
done

Kosaraju-Sharir algorithm

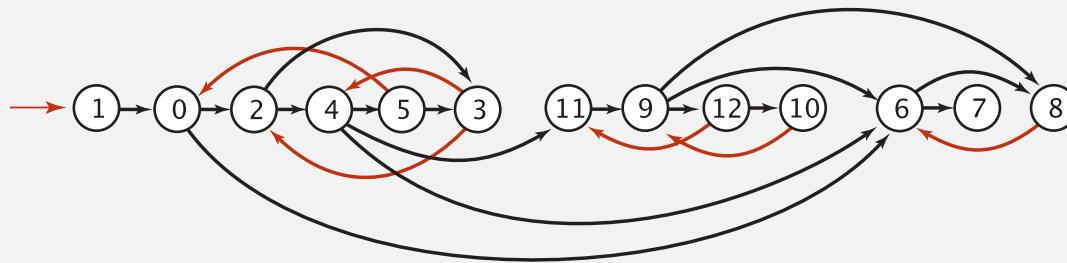
Simple (but mysterious) algorithm for computing strong components.

- Phase 1: run DFS on G^R to compute reverse postorder.
- Phase 2: run DFS on G , considering vertices in order given by first DFS.

DFS in reverse digraph G^R



check unmarked vertices in the order
0 1 2 3 4 5 6 7 8 9 10 11 12



reverse postorder for use in second dfs()
1 0 2 4 5 3 11 9 12 10 6 7 8

```
dfs(0)
  dfs(6)
    dfs(8)
      check 6
      8 done
      dfs(7)
      7 done
      6 done
    dfs(2)
      dfs(4)
        dfs(11)
        dfs(9)
          dfs(12)
            check 11
            dfs(10)
              check 9
              10 done
            12 done
            check 7
            check 6
  ...

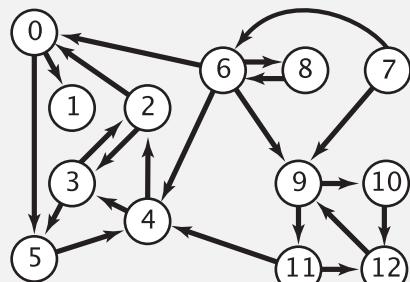
```

Kosaraju-Sharir algorithm

Simple (but mysterious) algorithm for computing strong components.

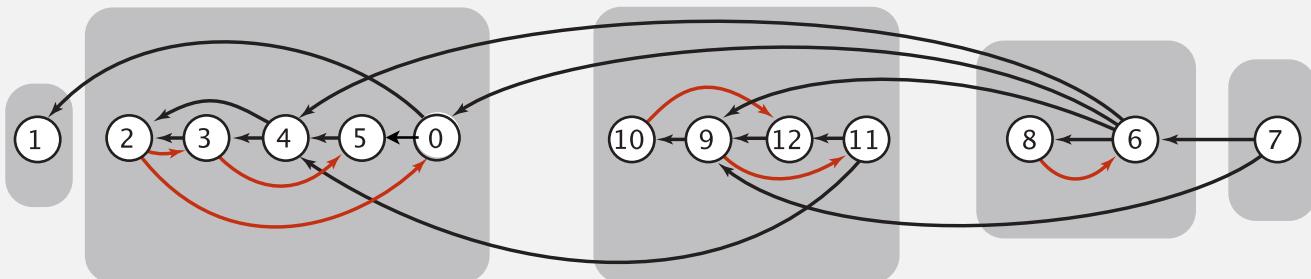
- Phase 1: run DFS on G^R to compute reverse postorder.
- Phase 2: run DFS on G , considering vertices in order given by first DFS.

DFS in original digraph G



check unmarked vertices in the order

1 0 2 4 5 3 11 9 12 10 6 7 8



dfs(1)
1 done

dfs(0)
dfs(5)
dfs(4)
dfs(3)
check 5
dfs(2)
check 0
check 3
2 done
3 done
check 2
4 done
5 done
check 1
0 done
check 2
check 4
check 5
check 3

dfs(11)
check 4
dfs(12)
dfs(9)
check 11
dfs(10)
check 12
10 done
9 done
12 done
11 done
check 9
check 12
check 10

dfs(6)
check 9
check 4
dfs(8)
check 6
8 done
check 0
6 done

dfs(7)
check 6
check 9
7 done
check 8

Kosaraju-Sharir algorithm

Proposition. Kosaraju-Sharir algorithm computes the strong components of a digraph in time proportional to $E + V$.

Pf.

- Running time: bottleneck is running DFS twice (and computing G^R).
- Correctness: tricky, see textbook (2nd printing).
- Implementation: easy!

Connected components in an undirected graph (with DFS)

```
public class CC
{
    private boolean marked[];
    private int[] id;
    private int count;

    public CC(Graph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];

        for (int v = 0; v < G.V(); v++)
        {
            if (!marked[v])
            {
                dfs(G, v);
                count++;
            }
        }
    }

    private void dfs(Graph G, int v)
    {
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
    }

    public boolean connected(int v, int w)
    { return id[v] == id[w]; }
}
```

Strong components in a digraph (with two DFSs)

```
public class KosarajuSharirSCC
{
    private boolean marked[];
    private int[] id;
    private int count;

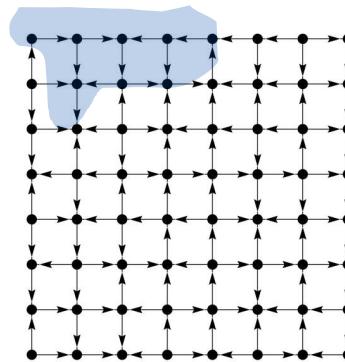
    public KosarajuSharirSCC(Digraph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
        for (int v : dfs.reversePost())
        {
            if (!marked[v])
            {
                dfs(G, v);
                count++;
            }
        }
    }

    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        id[v] = count;
        for (int w : G.adj(v))
            if (!marked[w])
                dfs(G, w);
    }

    public boolean stronglyConnected(int v, int w)
    {   return id[v] == id[w];  }
}
```

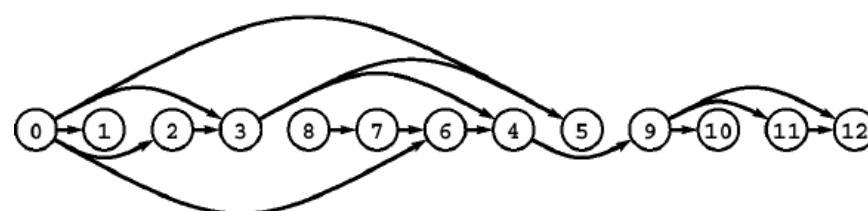
Digraph-processing summary: algorithms of the day

single-source
reachability
in a digraph



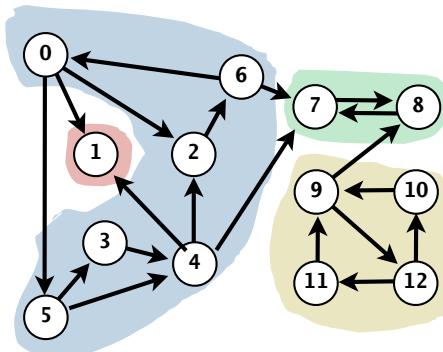
DFS

topological sort
in a DAG

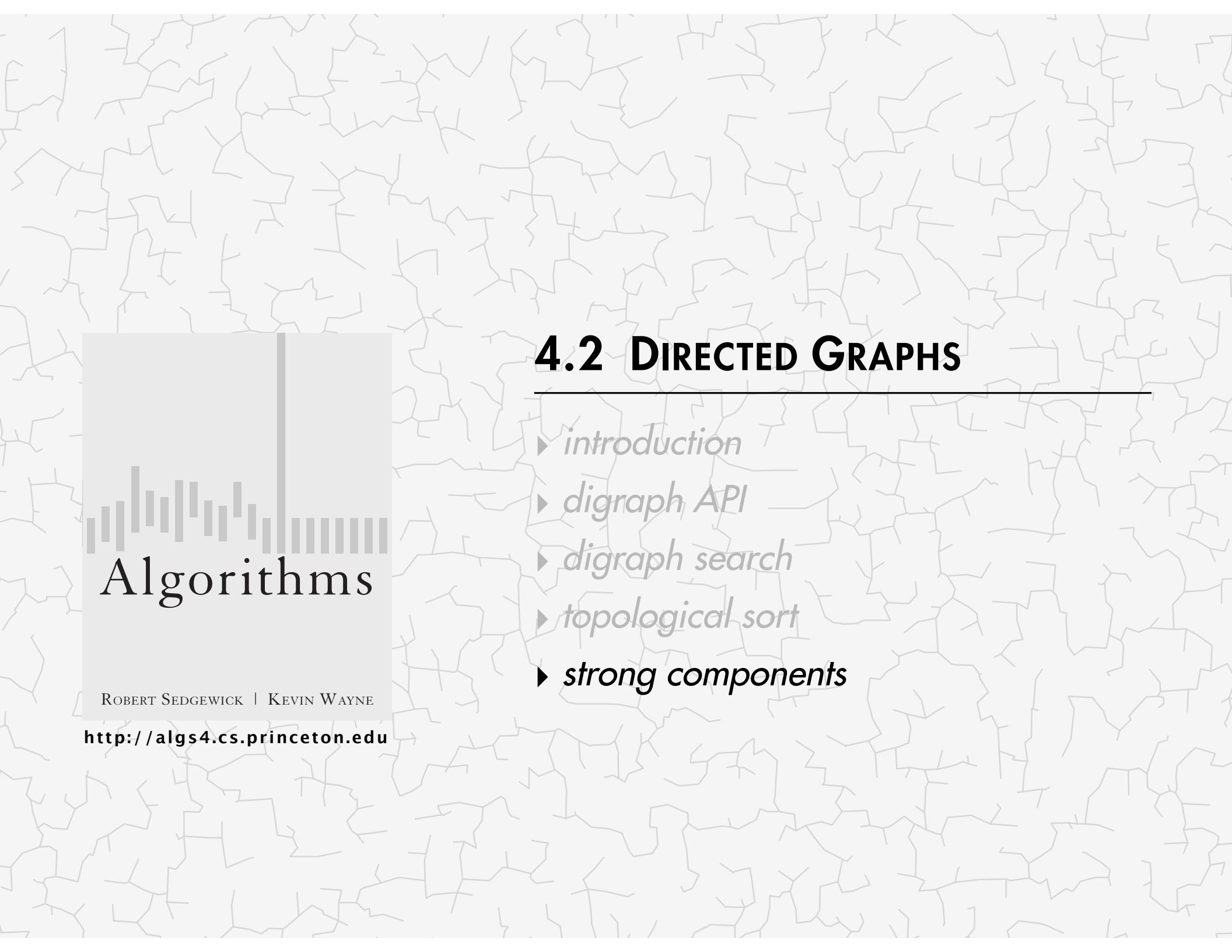


DFS

strong
components
in a digraph



Kosaraju-Sharir
DFS (twice)



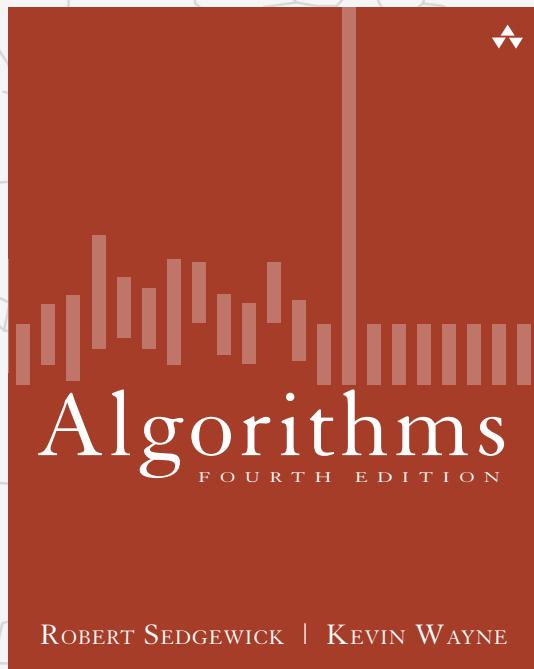
Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *digraph search*
- ▶ *topological sort*
- ▶ ***strong components***



<http://algs4.cs.princeton.edu>

4.2 DIRECTED GRAPHS

- ▶ *introduction*
- ▶ *digraph API*
- ▶ *digraph search*
- ▶ *topological sort*
- ▶ *strong components*



ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.4 SHORTEST PATHS

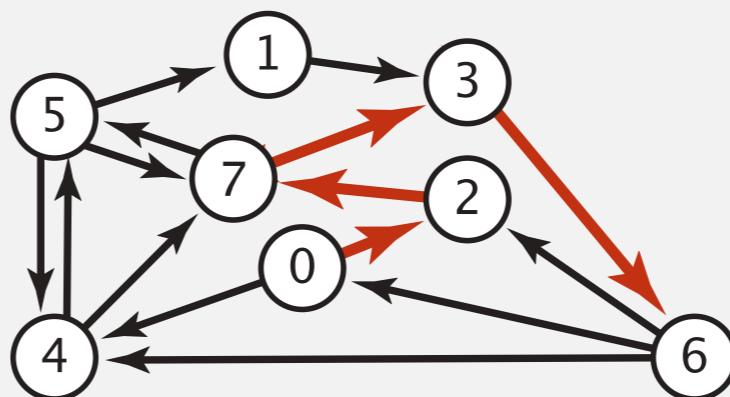
- ▶ *APIs*
- ▶ *shortest-paths properties*
- ▶ *Dijkstra's algorithm*
- ▶ *edge-weighted DAGs*
- ▶ *negative weights*

Shortest paths in an edge-weighted digraph

Given an edge-weighted digraph, find the shortest path from s to t .

edge-weighted digraph

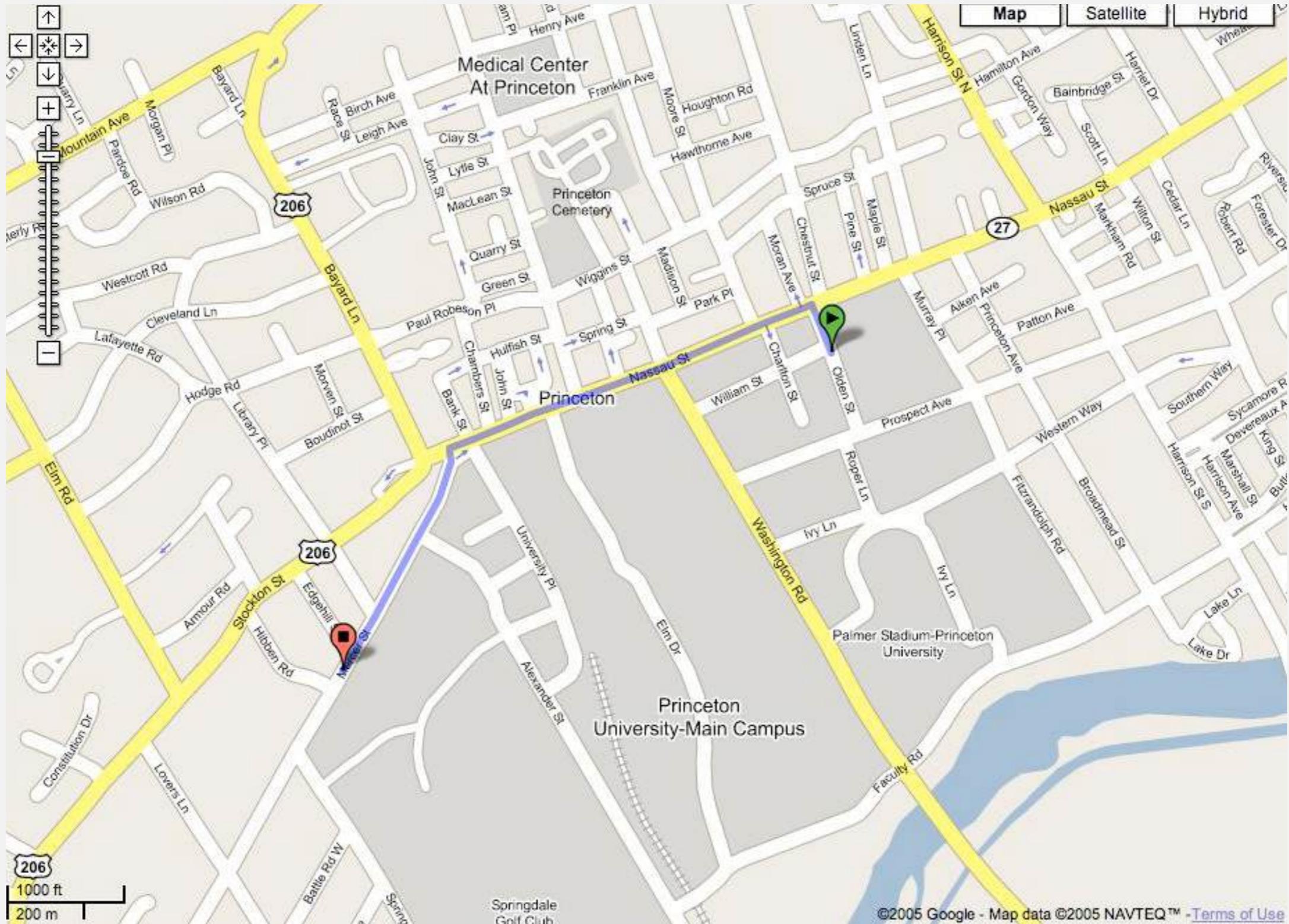
4->5	0.35
5->4	0.35
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



shortest path from 0 to 6

0->2	0.26
2->7	0.34
7->3	0.39
3->6	0.52

Google maps



Car navigation



Shortest path applications

- PERT/CPM.
- Map routing.
- Seam carving.
- Robot navigation.
- Texture mapping.
- Typesetting in TeX.
- Urban traffic planning.
- Optimal pipelining of VLSI chip.
- Telemarketer operator scheduling.
- Routing of telecommunications messages.
- Network routing protocols (OSPF, BGP, RIP).
- Exploiting arbitrage opportunities in currency exchange.
- Optimal truck routing through given traffic congestion pattern.



http://en.wikipedia.org/wiki/Seam_carving



Reference: Network Flows: Theory, Algorithms, and Applications, R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Prentice Hall, 1993.

Shortest path variants

Which vertices?

- Single source: from one vertex s to every other vertex.
- Source-sink: from one vertex s to another t .
- All pairs: between all pairs of vertices.

Restrictions on edge weights?

- Nonnegative weights.
- Euclidean weights.
- Arbitrary weights.

Cycles?

- No directed cycles.
- No "negative cycles."

Simplifying assumption. Shortest paths from s to each vertex v exist.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

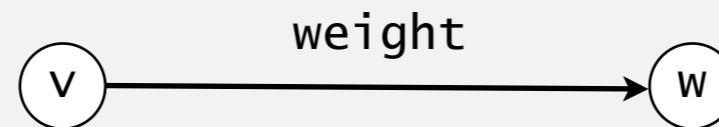
<http://algs4.cs.princeton.edu>

4.4 SHORTEST PATHS

- ▶ **APIs**
- ▶ *shortest-paths properties*
- ▶ *Dijkstra's algorithm*
- ▶ *edge-weighted DAGs*
- ▶ *negative weights*

Weighted directed edge API

```
public class DirectedEdge  
  
    DirectedEdge(int v, int w, double weight)      weighted edge v→w  
  
    int from()                                     vertex v  
  
    int to()                                       vertex w  
  
    double weight()                                weight of this edge  
  
    String toString()                             string representation
```



Idiom for processing an edge e: `int v = e.from(), w = e.to();`

Weighted directed edge: implementation in Java

Similar to Edge for undirected graphs, but a bit simpler.

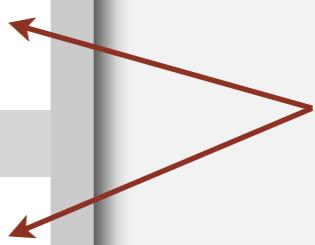
```
public class DirectedEdge
{
    private final int v, w;
    private final double weight;

    public DirectedEdge(int v, int w, double weight)
    {
        this.v = v;
        this.w = w;
        this.weight = weight;
    }

    public int from()
    {   return v;   }

    public int to()
    {   return w;   }

    public int weight()
    {   return weight;   }
}
```



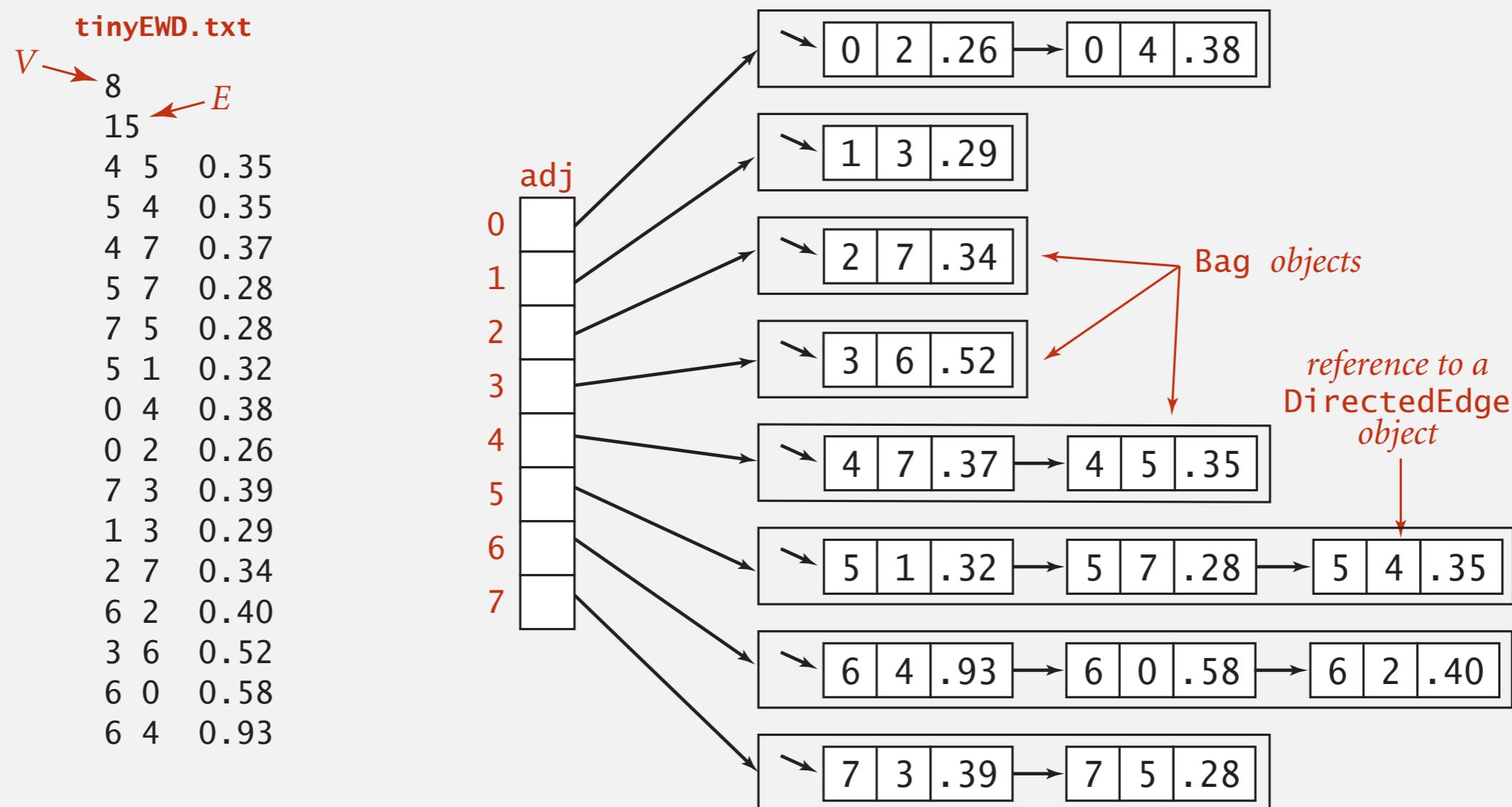
from() and to() replace
either() and other()

Edge-weighted digraph API

public class EdgeWeightedDigraph	
EdgeWeightedDigraph(int V)	<i>edge-weighted digraph with V vertices</i>
EdgeWeightedDigraph(In in)	<i>edge-weighted digraph from input stream</i>
void addEdge(DirectedEdge e)	<i>add weighted directed edge e</i>
Iterable<DirectedEdge> adj(int v)	<i>edges pointing from v</i>
int V()	<i>number of vertices</i>
int E()	<i>number of edges</i>
Iterable<DirectedEdge> edges()	<i>all edges</i>
String toString()	<i>string representation</i>

Conventions. Allow self-loops and parallel edges.

Edge-weighted digraph: adjacency-lists representation



Edge-weighted digraph: adjacency-lists implementation in Java

Same as EdgeWeightedGraph except replace Graph with Digraph.

```
public class EdgeWeightedDigraph
{
    private final int V;
    private final Bag<DirectedEdge>[] adj;

    public EdgeWeightedDigraph(int V)
    {
        this.V = V;
        adj = (Bag<DirectedEdge>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<DirectedEdge>();
    }

    public void addEdge(DirectedEdge e)
    {
        int v = e.from();
        adj[v].add(e);
    }

    public Iterable<DirectedEdge> adj(int v)
    { return adj[v]; }
}
```

←
add edge $e = v \rightarrow w$ to
only v 's adjacency list

Single-source shortest paths API

Goal. Find the shortest path from s to every other vertex.

```
public class SP
```

```
    SP(EdgeWeightedDigraph G, int s) shortest paths from s in graph G
```

```
    double distTo(int v) length of shortest path from s to v
```

```
    Iterable <DirectedEdge> pathTo(int v) shortest path from s to v
```

```
    boolean hasPathTo(int v) is there a path from s to v?
```

```
SP sp = new SP(G, s);
for (int v = 0; v < G.V(); v++)
{
    StdOut.printf("%d to %d (%.2f): ", s, v, sp.distTo(v));
    for (DirectedEdge e : sp.pathTo(v))
        StdOut.print(e + " ");
    StdOut.println();
}
```

Single-source shortest paths API

Goal. Find the shortest path from s to every other vertex.

```
public class SP
```

```
    SP(EdgeWeightedDigraph G, int s) shortest paths from s in graph G
```

```
    double distTo(int v) length of shortest path from s to v
```

```
    Iterable <DirectedEdge> pathTo(int v) shortest path from s to v
```

```
    boolean hasPathTo(int v) is there a path from s to v?
```

```
% java SP tinyEWG.txt 0
0 to 0 (0.00):
0 to 1 (1.05): 0->4 0.38  4->5 0.35  5->1 0.32
0 to 2 (0.26): 0->2 0.26
0 to 3 (0.99): 0->2 0.26  2->7 0.34  7->3 0.39
0 to 4 (0.38): 0->4 0.38
0 to 5 (0.73): 0->4 0.38  4->5 0.35
0 to 6 (1.51): 0->2 0.26  2->7 0.34  7->3 0.39  3->6 0.52
0 to 7 (0.60): 0->2 0.26  2->7 0.34
```

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.4 SHORTEST PATHS

- ▶ **APIs**
- ▶ *shortest-paths properties*
- ▶ *Dijkstra's algorithm*
- ▶ *edge-weighted DAGs*
- ▶ *negative weights*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.4 SHORTEST PATHS

- ▶ APIs
- ▶ *shortest-paths properties*
- ▶ *Dijkstra's algorithm*
- ▶ *edge-weighted DAGs*
- ▶ *negative weights*

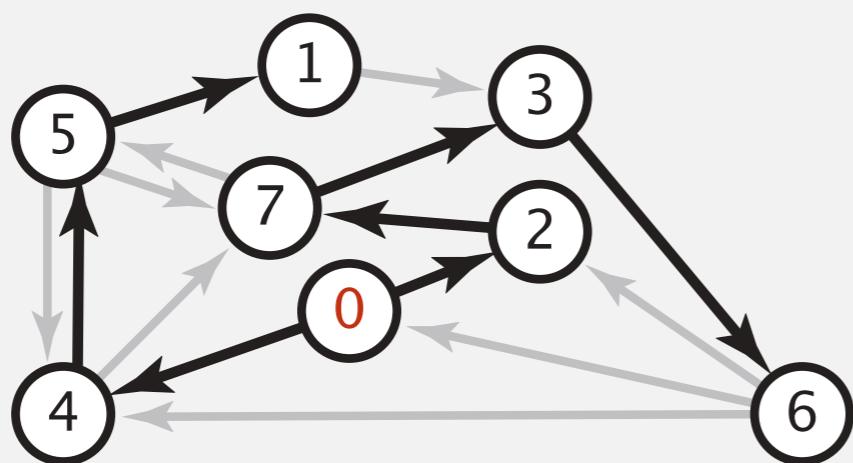
Data structures for single-source shortest paths

Goal. Find the shortest path from s to every other vertex.

Observation. A shortest-paths tree (SPT) solution exists. Why?

Consequence. Can represent the SPT with two vertex-indexed arrays:

- $\text{distTo}[v]$ is length of shortest path from s to v .
- $\text{edgeTo}[v]$ is last edge on shortest path from s to v .



shortest-paths tree from 0

	edgeTo[]	distTo[]
0	null	0
1	5->1	0.32
2	0->2	0.26
3	7->3	0.37
4	0->4	0.38
5	4->5	0.35
6	3->6	0.52
7	2->7	0.34

parent-link representation

Data structures for single-source shortest paths

Goal. Find the shortest path from s to every other vertex.

Observation. A **shortest-paths tree** (SPT) solution exists. Why?

Consequence. Can represent the SPT with two vertex-indexed arrays:

- $\text{distTo}[v]$ is length of shortest path from s to v .
- $\text{edgeTo}[v]$ is last edge on shortest path from s to v .

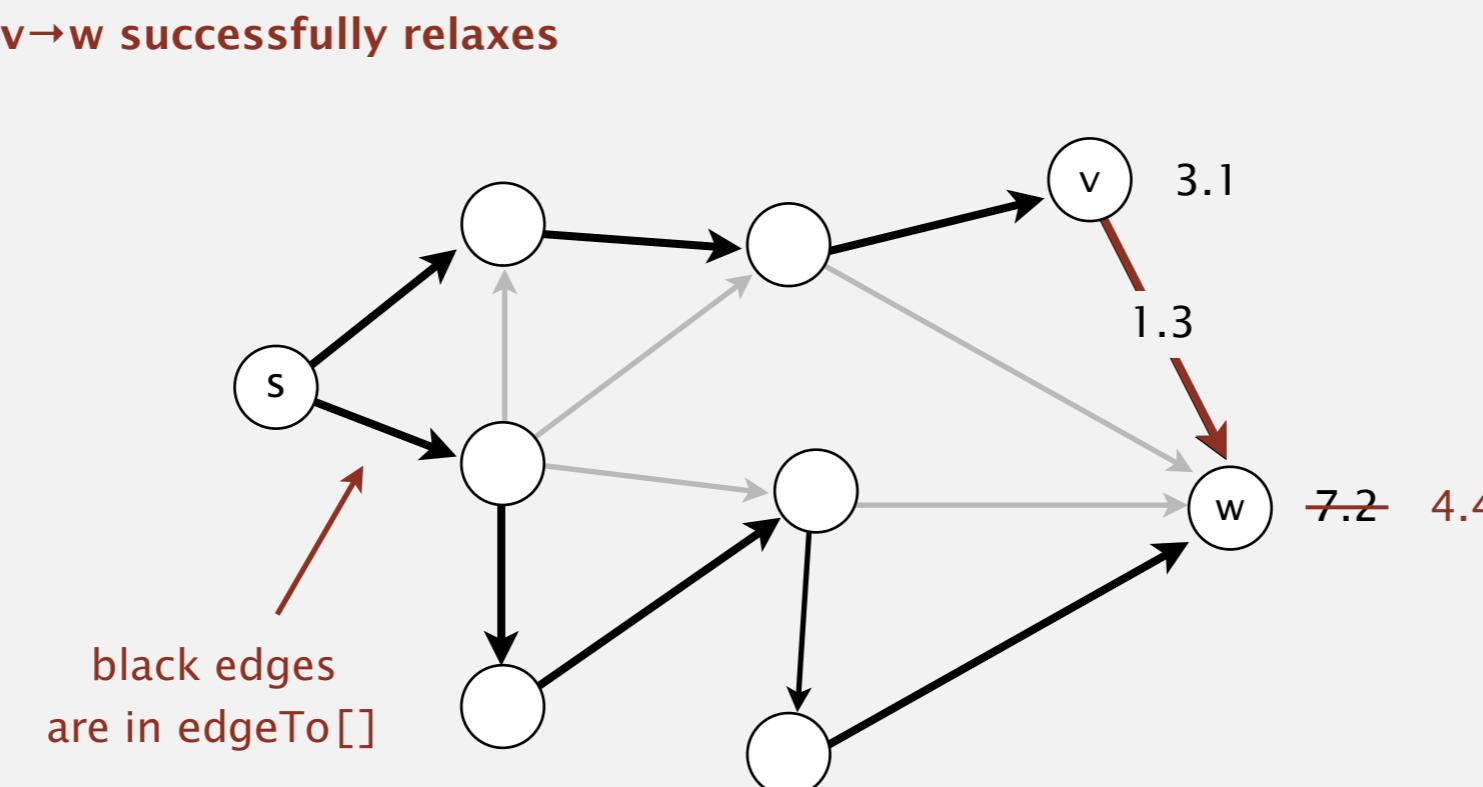
```
public double distTo(int v)
{   return distTo[v]; }

public Iterable<DirectedEdge> pathTo(int v)
{
    Stack<DirectedEdge> path = new Stack<DirectedEdge>();
    for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.from()])
        path.push(e);
    return path;
}
```

Edge relaxation

Relax edge $e = v \rightarrow w$.

- $\text{distTo}[v]$ is length of shortest **known** path from s to v .
- $\text{distTo}[w]$ is length of shortest **known** path from s to w .
- $\text{edgeTo}[w]$ is last edge on shortest **known** path from s to w .
- If $e = v \rightarrow w$ gives shorter path to w through v ,
update both $\text{distTo}[w]$ and $\text{edgeTo}[w]$.



Edge relaxation

Relax edge $e = v \rightarrow w$.

- $\text{distTo}[v]$ is length of shortest **known** path from s to v .
- $\text{distTo}[w]$ is length of shortest **known** path from s to w .
- $\text{edgeTo}[w]$ is last edge on shortest **known** path from s to w .
- If $e = v \rightarrow w$ gives shorter path to w through v ,
update both $\text{distTo}[w]$ and $\text{edgeTo}[w]$.

```
private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
    }
}
```

Shortest-paths optimality conditions

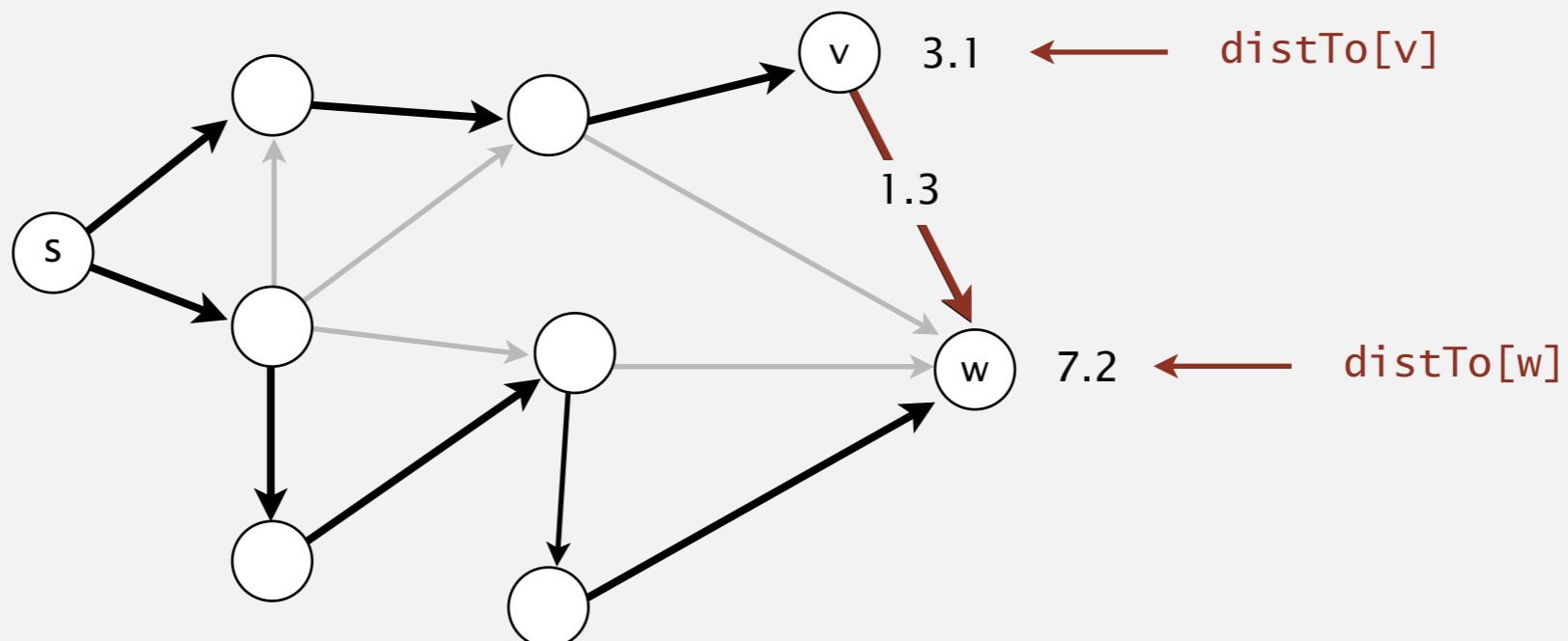
Proposition. Let G be an edge-weighted digraph.

Then $\text{distTo}[]$ are the shortest path distances from s iff:

- $\text{distTo}[s] = 0$.
- For each vertex v , $\text{distTo}[v]$ is the length of some path from s to v .
- For each edge $e = v \rightarrow w$, $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.

Pf. \Leftarrow [necessary]

- Suppose that $\text{distTo}[w] > \text{distTo}[v] + e.\text{weight}()$ for some edge $e = v \rightarrow w$.
- Then, e gives a path from s to w (through v) of length less than $\text{distTo}[w]$.



Shortest-paths optimality conditions

Proposition. Let G be an edge-weighted digraph.

Then $\text{distTo}[]$ are the shortest path distances from s iff:

- $\text{distTo}[s] = 0$.
- For each vertex v , $\text{distTo}[v]$ is the length of some path from s to v .
- For each edge $e = v \rightarrow w$, $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.

Pf. \Rightarrow [sufficient]

- Suppose that $s = v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k = w$ is a shortest path from s to w .
- Then, $\text{distTo}[v_1] \leq \text{distTo}[v_0] + e_1.\text{weight}()$
 $\text{distTo}[v_2] \leq \text{distTo}[v_1] + e_2.\text{weight}()$
 \dots
 $\text{distTo}[v_k] \leq \text{distTo}[v_{k-1}] + e_k.\text{weight}()$

$e_i = i^{\text{th}}$ edge on shortest path from s to w

- Add inequalities; simplify; and substitute $\text{distTo}[v_0] = \text{distTo}[s] = 0$:

$$\text{distTo}[w] = \text{distTo}[v_k] \leq e_1.\text{weight}() + e_2.\text{weight}() + \dots + e_k.\text{weight}()$$

weight of shortest path from s to w

- Thus, $\text{distTo}[w]$ is the weight of shortest path to w . ■

weight of some path from s to w

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat until optimality conditions are satisfied:

- Relax any edge.
-

Proposition. Generic algorithm computes SPT (if it exists) from s .

Pf sketch.

- Throughout algorithm, $\text{distTo}[v]$ is the length of a simple path from s to v (and $\text{edgeTo}[v]$ is last edge on path).
- Each successful relaxation decreases $\text{distTo}[v]$ for some v .
- The entry $\text{distTo}[v]$ can decrease at most a finite number of times. ■

Generic shortest-paths algorithm

Generic algorithm (to compute SPT from s)

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat until optimality conditions are satisfied:

- Relax any edge.
-

Efficient implementations. How to choose which edge to relax?

Ex 1. Dijkstra's algorithm (nonnegative weights).

Ex 2. Topological sort algorithm (no directed cycles).

Ex 3. Bellman-Ford algorithm (no negative cycles).

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.4 SHORTEST PATHS

- ▶ APIs
- ▶ *shortest-paths properties*
- ▶ *Dijkstra's algorithm*
- ▶ *edge-weighted DAGs*
- ▶ *negative weights*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.4 SHORTEST PATHS

- ▶ APIs
- ▶ *shortest-paths properties*
- ▶ *Dijkstra's algorithm*
- ▶ *edge-weighted DAGs*
- ▶ *negative weights*

Edsger W. Dijkstra: select quotes

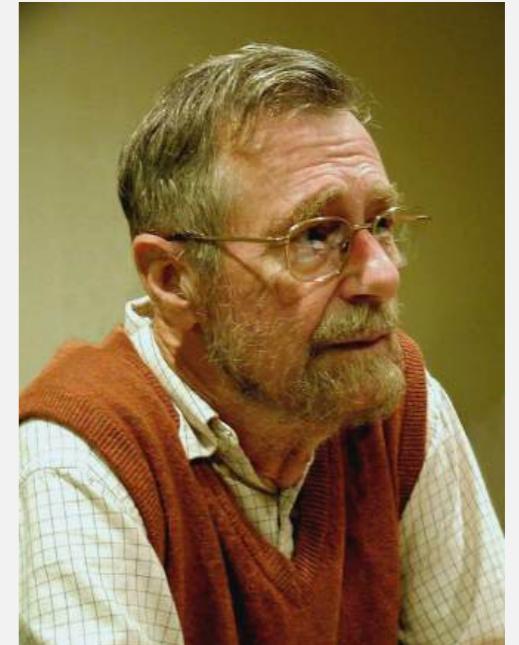
“Do only what only you can do.”

“In their capacity as a tool, computers will be but a ripple on the surface of our culture. In their capacity as intellectual challenge, they are without precedent in the cultural history of mankind.”

“The use of COBOL cripples the mind; its teaching should, therefore, be regarded as a criminal offence.”

“It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration.”

“APL is a mistake, carried through to perfection. It is the language of the future for the programming techniques of the past: it creates a new generation of coding bums.”



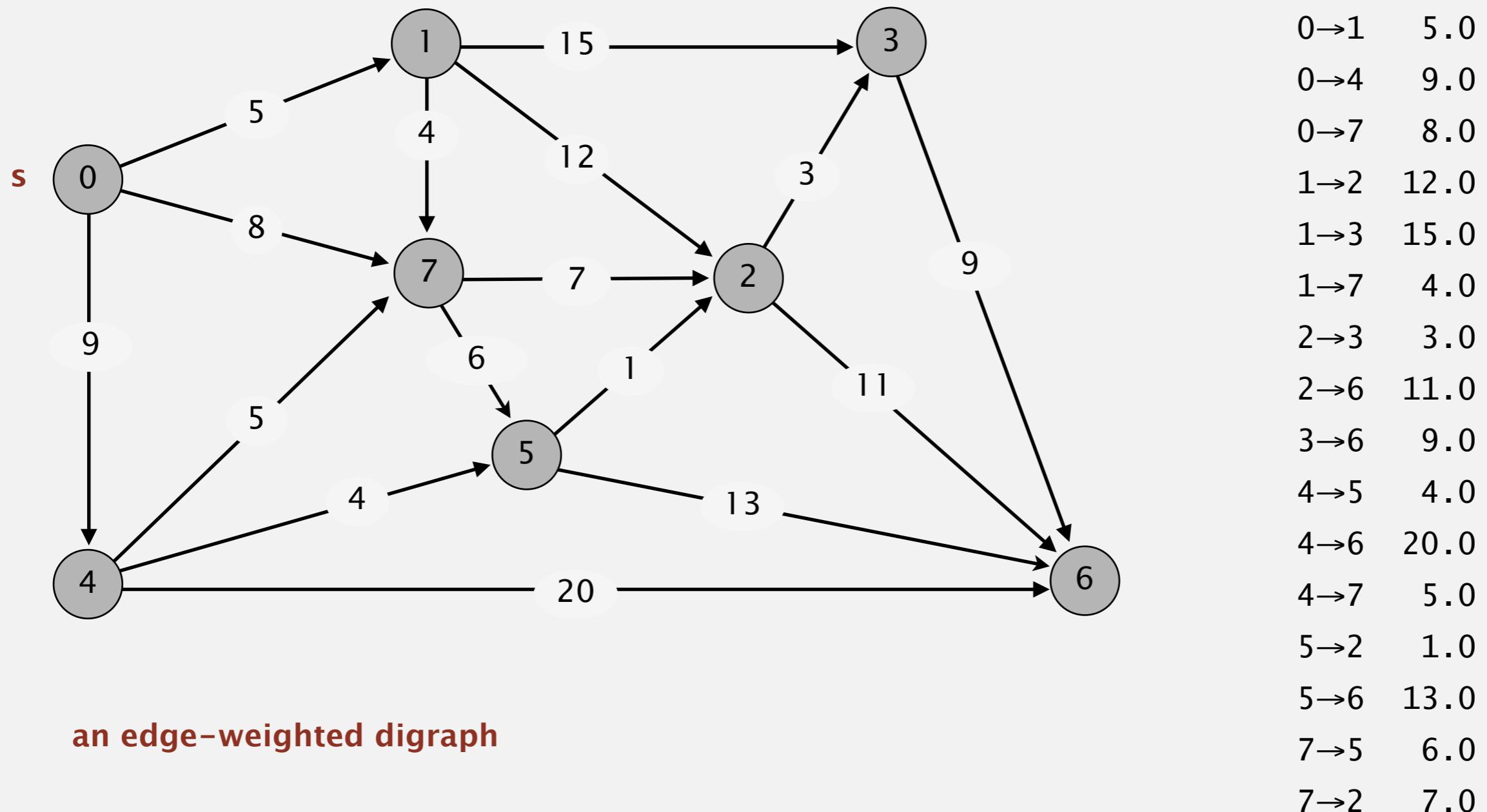
Edsger W. Dijkstra
Turing award 1972

Edsger W. Dijkstra: select quotes



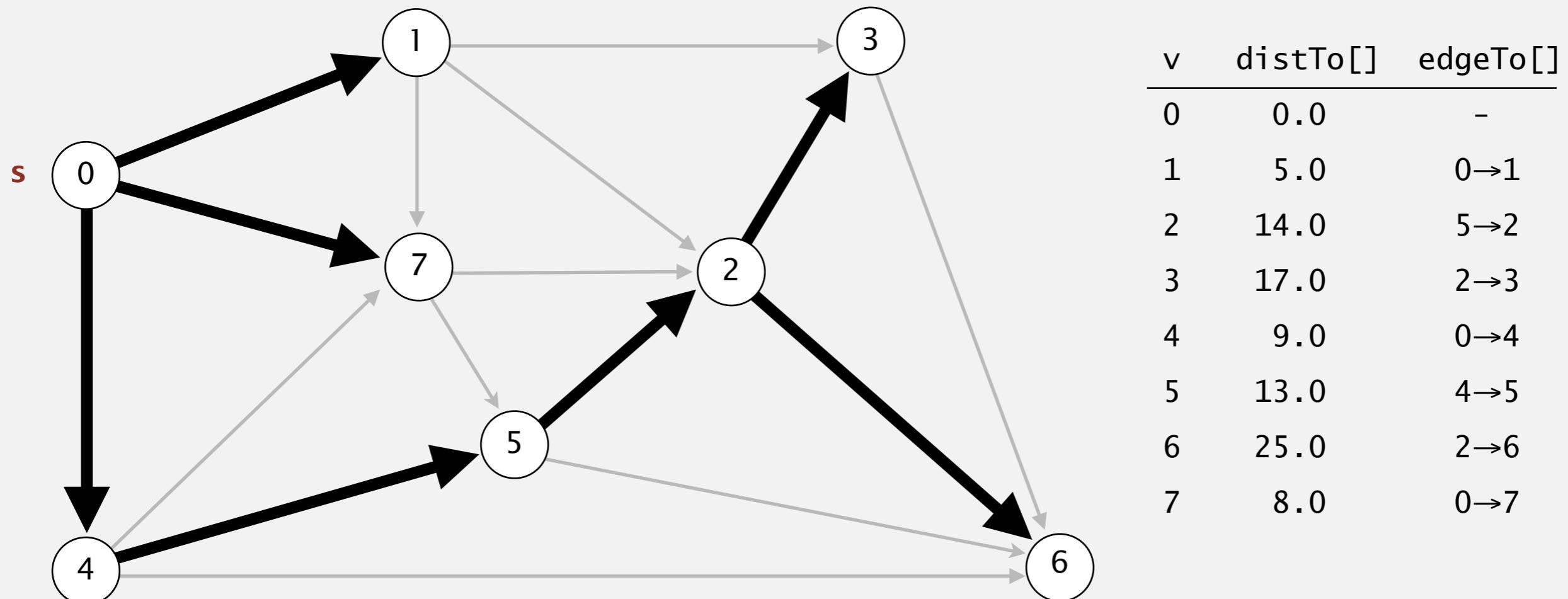
Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.



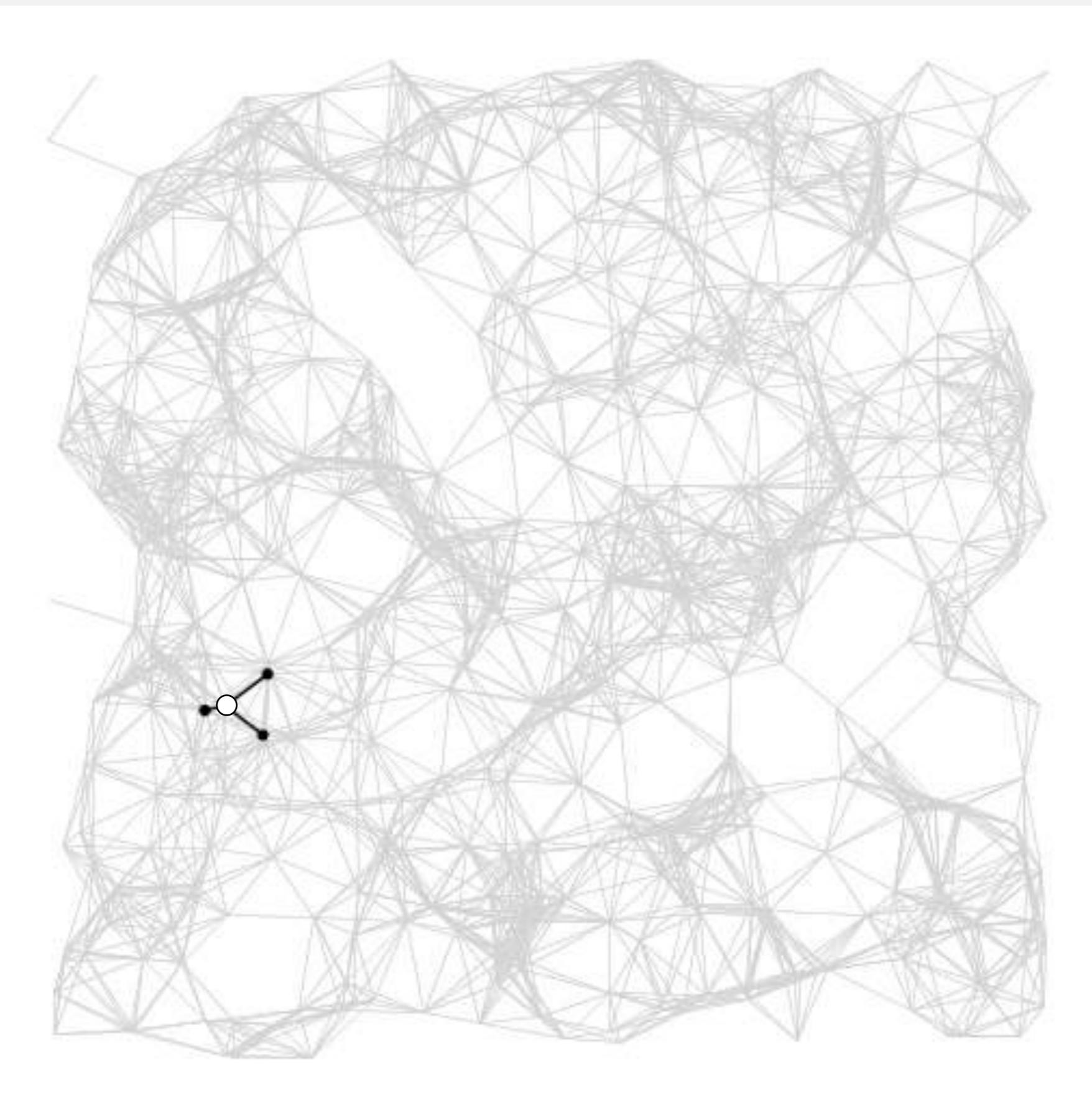
Dijkstra's algorithm demo

- Consider vertices in increasing order of distance from s (non-tree vertex with the lowest $\text{distTo}[]$ value).
- Add vertex to tree and relax all edges pointing from that vertex.

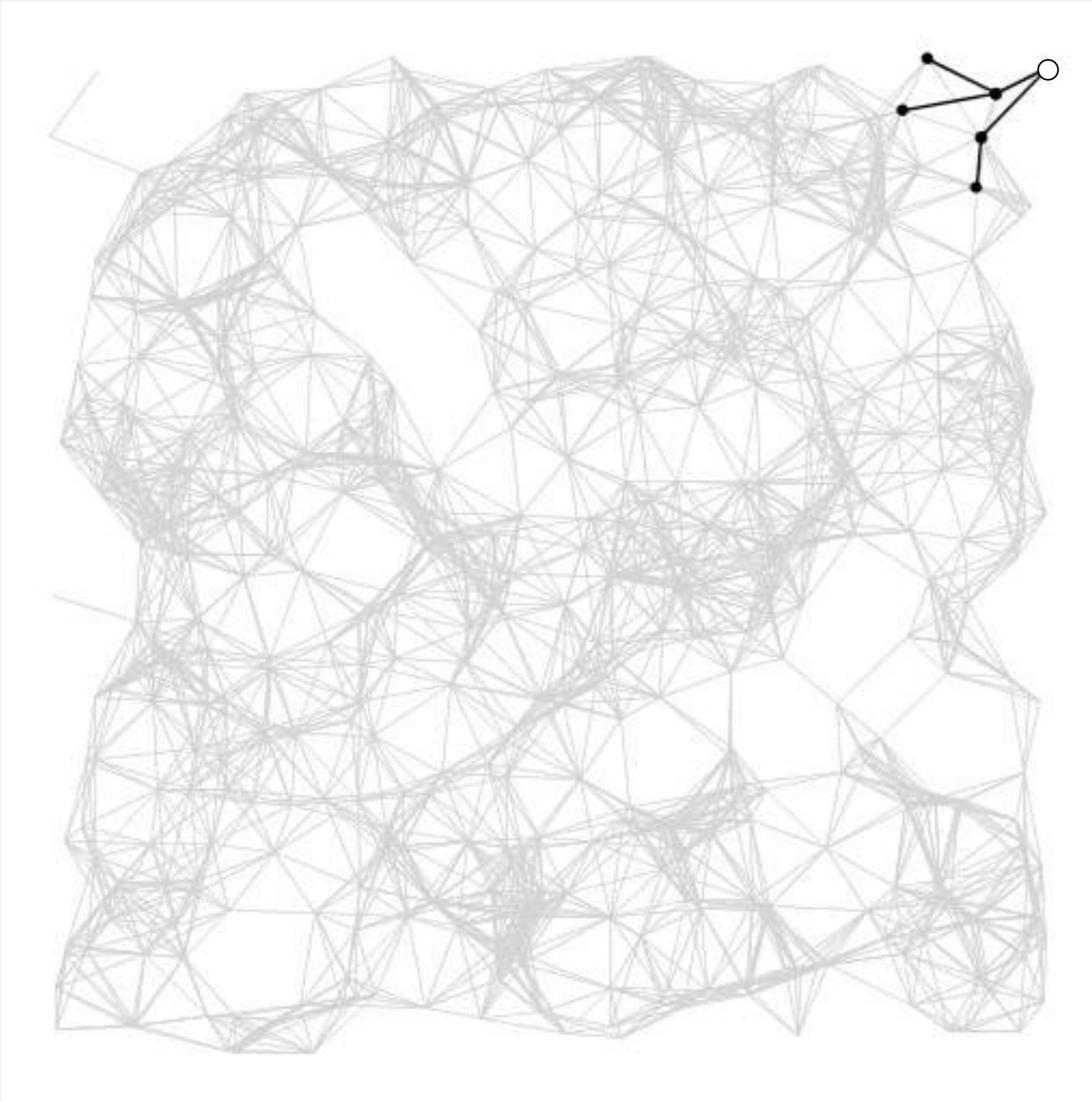


shortest-paths tree from vertex s

Dijkstra's algorithm visualization



Dijkstra's algorithm visualization



Dijkstra's algorithm: correctness proof

Proposition. Dijkstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

Pf.

- Each edge $e = v \rightarrow w$ is relaxed exactly once (when v is relaxed), leaving $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.
- Inequality holds until algorithm terminates because:
 - $\text{distTo}[w]$ cannot increase ← distTo[] values are monotone decreasing
 - $\text{distTo}[v]$ will not change ← we choose lowest distTo[] value at each step
(and edge weights are nonnegative)
- Thus, upon termination, shortest-paths optimality conditions hold. ■

Dijkstra's algorithm: Java implementation

```
public class DijkstraSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;

    public DijkstraSP(EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        pq.insert(s, 0.0);
        while (!pq.isEmpty())
        {
            int v = pq.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
}
```

←
relax vertices in order
of distance from s

Dijkstra's algorithm: Java implementation

```
private void relax(DirectedEdge e)
{
    int v = e.from(), w = e.to();
    if (distTo[w] > distTo[v] + e.weight())
    {
        distTo[w] = distTo[v] + e.weight();
        edgeTo[w] = e;
        if (pq.contains(w)) pq.decreaseKey(w, distTo[w]);
        else                  pq.insert      (w, distTo[w]);
    }
}
```



update PQ

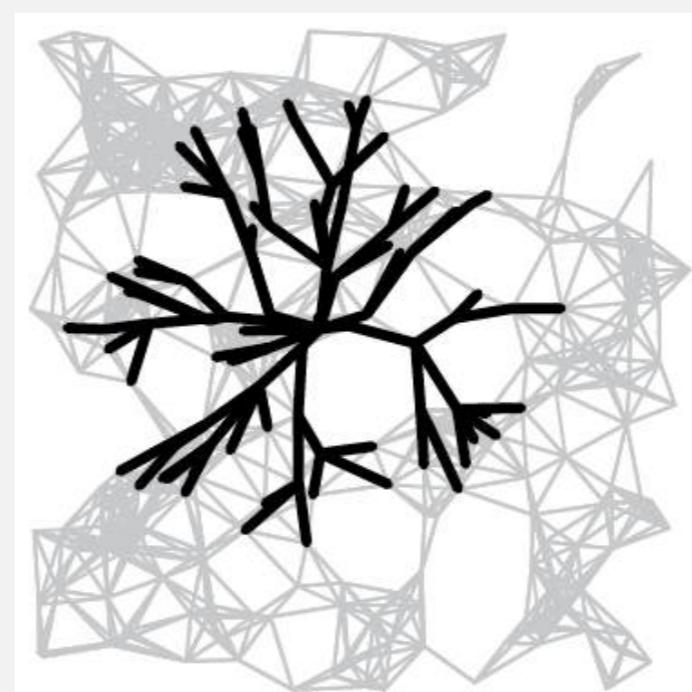
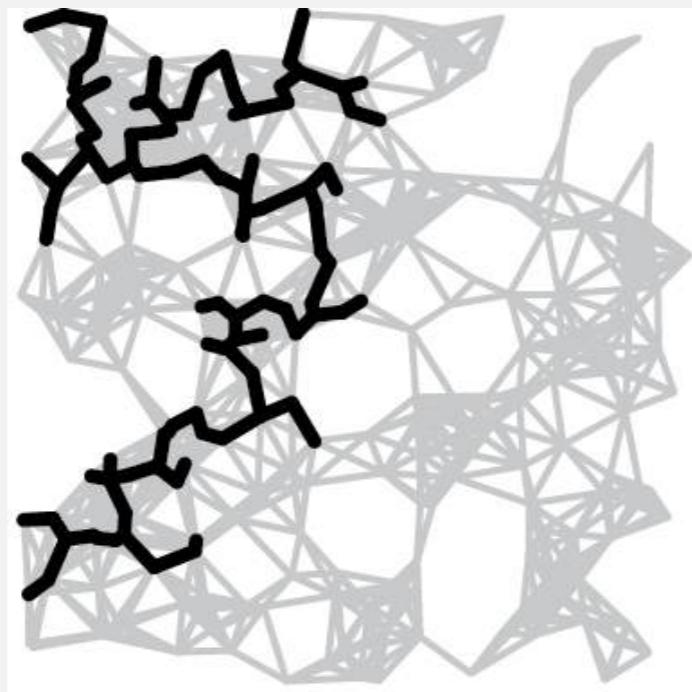
Computing spanning trees in graphs

Dijkstra's algorithm seem familiar?

- Prim's algorithm is essentially the same algorithm.
- Both are in a family of algorithms that compute a graph's spanning tree.

Main distinction: Rule used to choose next vertex for the tree.

- Prim's: Closest vertex to the **tree** (via an undirected edge).
- Dijkstra's: Closest vertex to the **source** (via a directed path).



Note: DFS and BFS are also in this family of algorithms.

Dijkstra's algorithm: which priority queue?

Depends on PQ implementation: V insert, V delete-min, E decrease-key.

PQ implementation	insert	delete-min	decrease-key	total
unordered array	1	V	1	V^2
binary heap	$\log V$	$\log V$	$\log V$	$E \log V$
d-way heap (Johnson 1975)	$\log_d V$	$d \log_d V$	$\log_d V$	$E \log_{E/V} V$
Fibonacci heap (Fredman-Tarjan 1984)	1^\dagger	$\log V^\dagger$	1^\dagger	$E + V \log V$

\dagger amortized

Bottom line.

- Array implementation optimal for dense graphs.
- Binary heap much faster for sparse graphs.
- 4-way heap worth the trouble in performance-critical situations.
- Fibonacci heap best in theory, but not worth implementing.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.4 SHORTEST PATHS

- ▶ APIs
- ▶ *shortest-paths properties*
- ▶ *Dijkstra's algorithm*
- ▶ *edge-weighted DAGs*
- ▶ *negative weights*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.4 SHORTEST PATHS

- ▶ APIs
- ▶ *shortest-paths properties*
- ▶ *Dijkstra's algorithm*
- ▶ **edge-weighted DAGs**
- ▶ *negative weights*

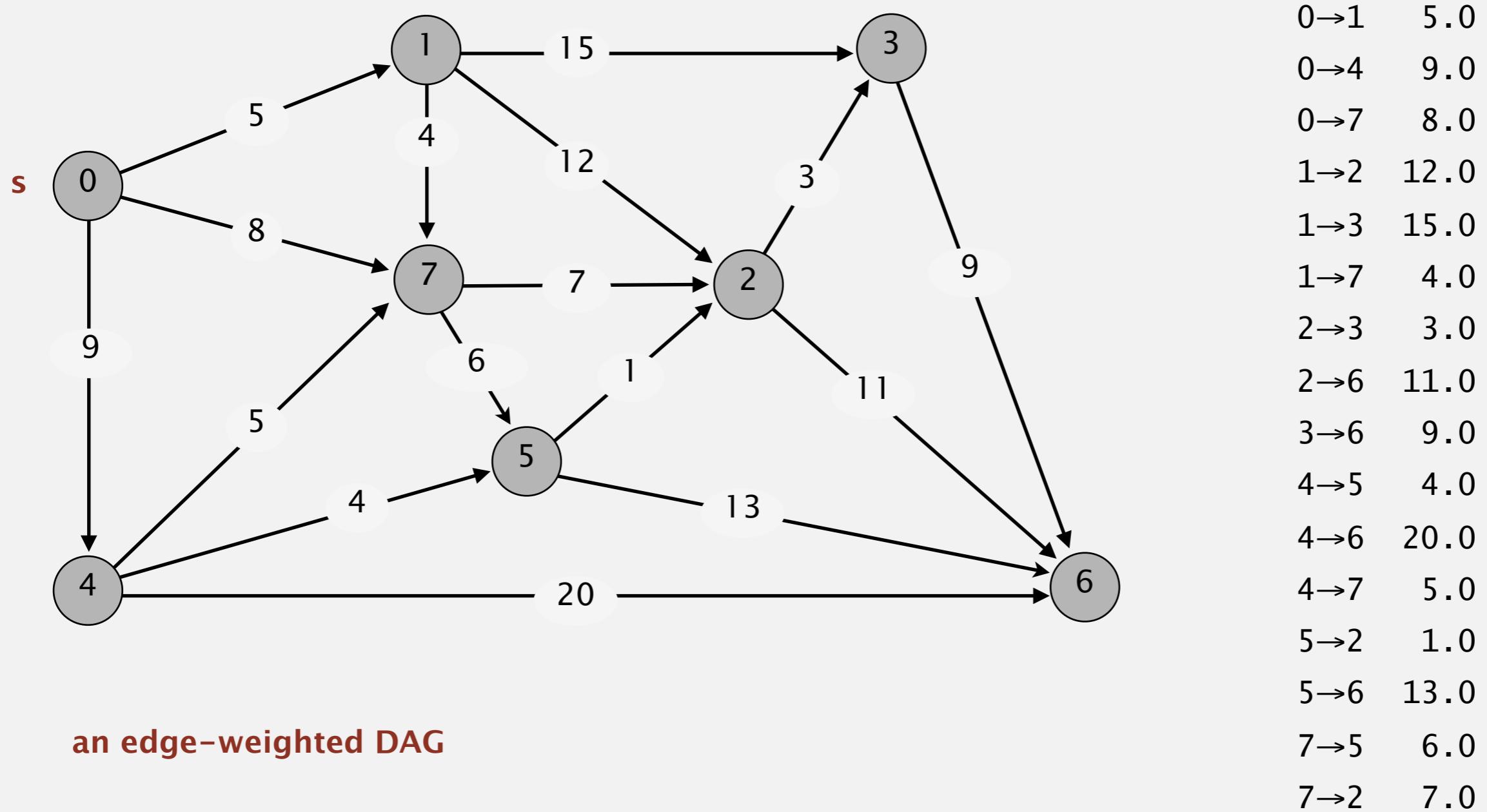
Acyclic edge-weighted digraphs

Q. Suppose that an edge-weighted digraph has no directed cycles.
Is it easier to find shortest paths than in a general digraph?

A. Yes!

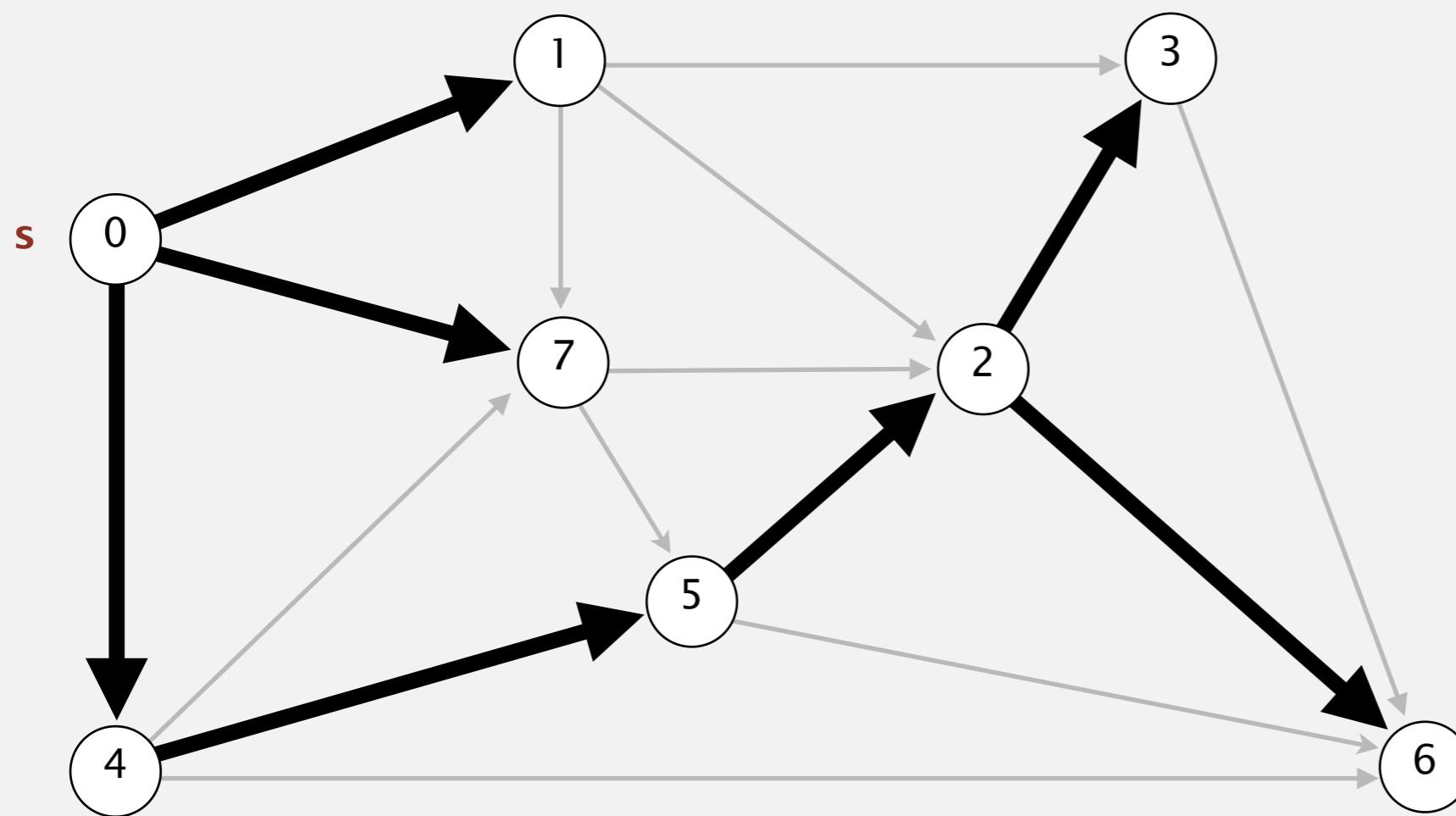
Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



Acyclic shortest paths demo

- Consider vertices in topological order.
- Relax all edges pointing from that vertex.



shortest-paths tree from vertex s

v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

Shortest paths in edge-weighted DAGs

Proposition. Topological sort algorithm computes SPT in any edge-weighted DAG in time proportional to $E + V$.

edge weights
can be negative!

Pf.

- Each edge $e = v \rightarrow w$ is relaxed exactly once (when v is relaxed), leaving $\text{distTo}[w] \leq \text{distTo}[v] + e.\text{weight}()$.
- Inequality holds until algorithm terminates because:
 - $\text{distTo}[w]$ cannot increase ← $\text{distTo}[]$ values are monotone decreasing
 - $\text{distTo}[v]$ will not change ← because of topological order, no edge pointing to v will be relaxed after v is relaxed
- Thus, upon termination, shortest-paths optimality conditions hold. ■

Shortest paths in edge-weighted DAGs

```
public class AcyclicSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;

    public AcyclicSP(EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        Topological topological = new Topological(G); ← topological order
        for (int v : topological.order())
            for (DirectedEdge e : G.adj(v))
                relax(e);
    }
}
```

Content-aware resizing

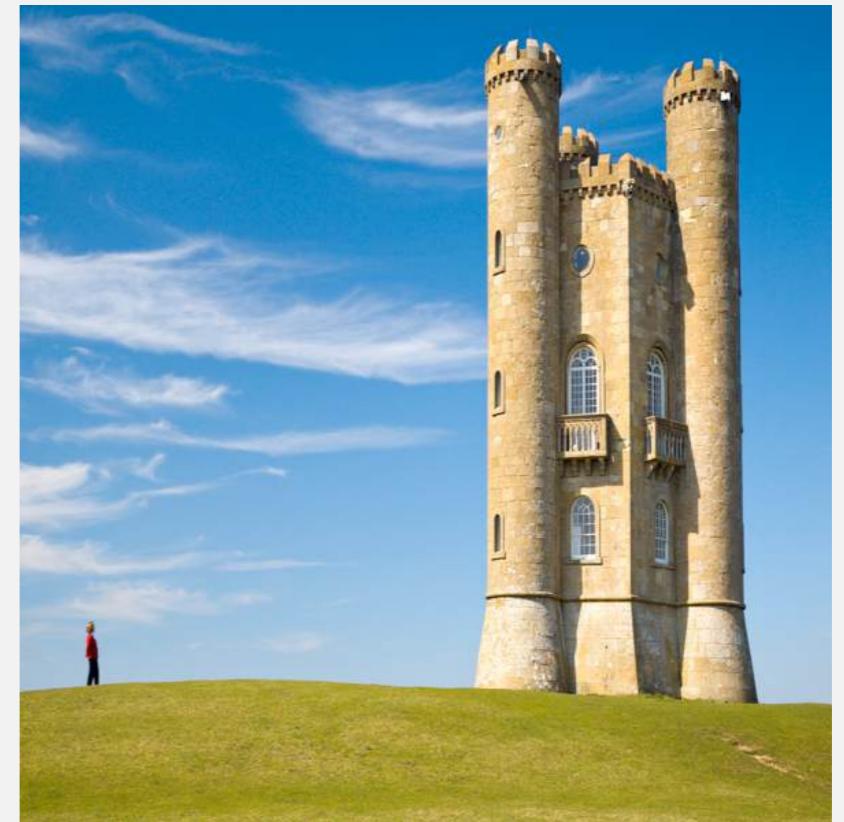
Seam carving. [Avidan and Shamir] Resize an image without distortion for display on cell phones and web browsers.



<http://www.youtube.com/watch?v=vIFCV2spKtg>

Content-aware resizing

[Seam carving.](#) [Avidan and Shamir] Resize an image without distortion for display on cell phones and web browsers.



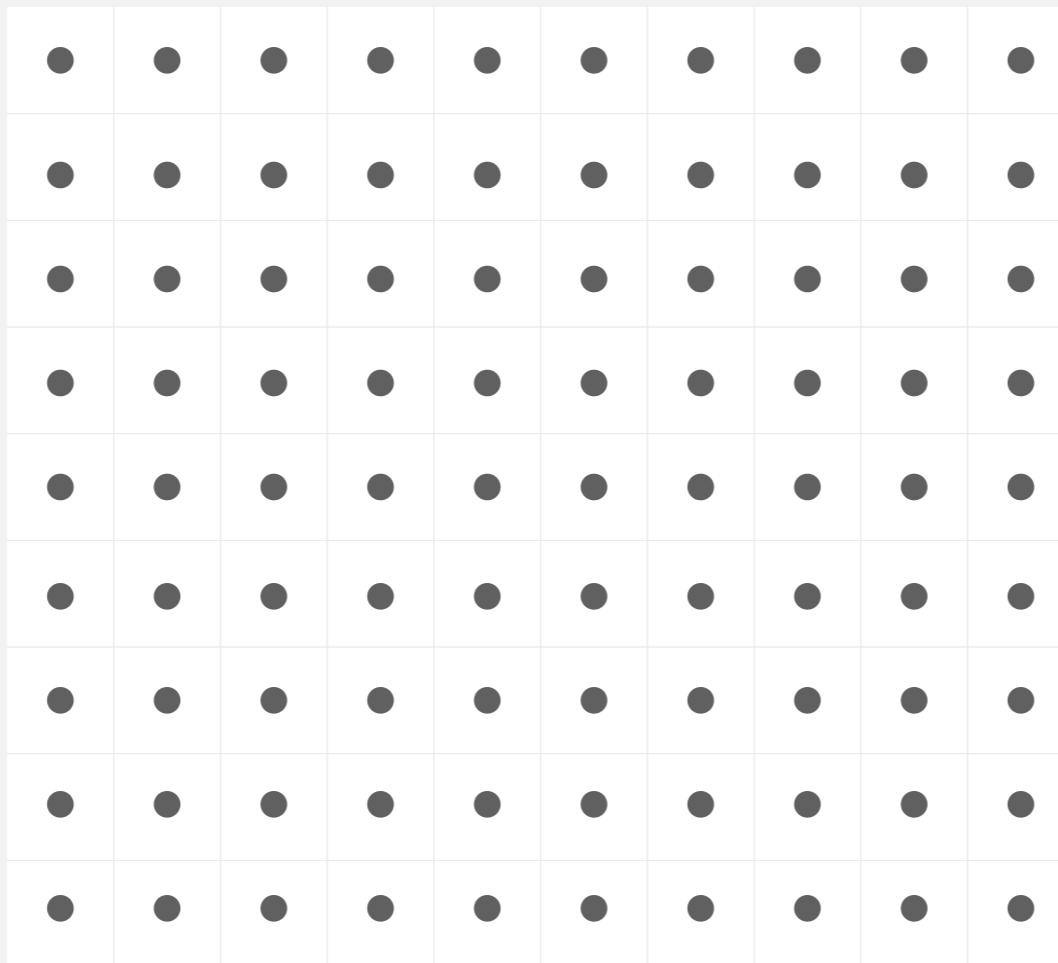
[In the wild.](#) Photoshop CS 5, Imagemagick, GIMP, ...



Content-aware resizing

To find vertical seam:

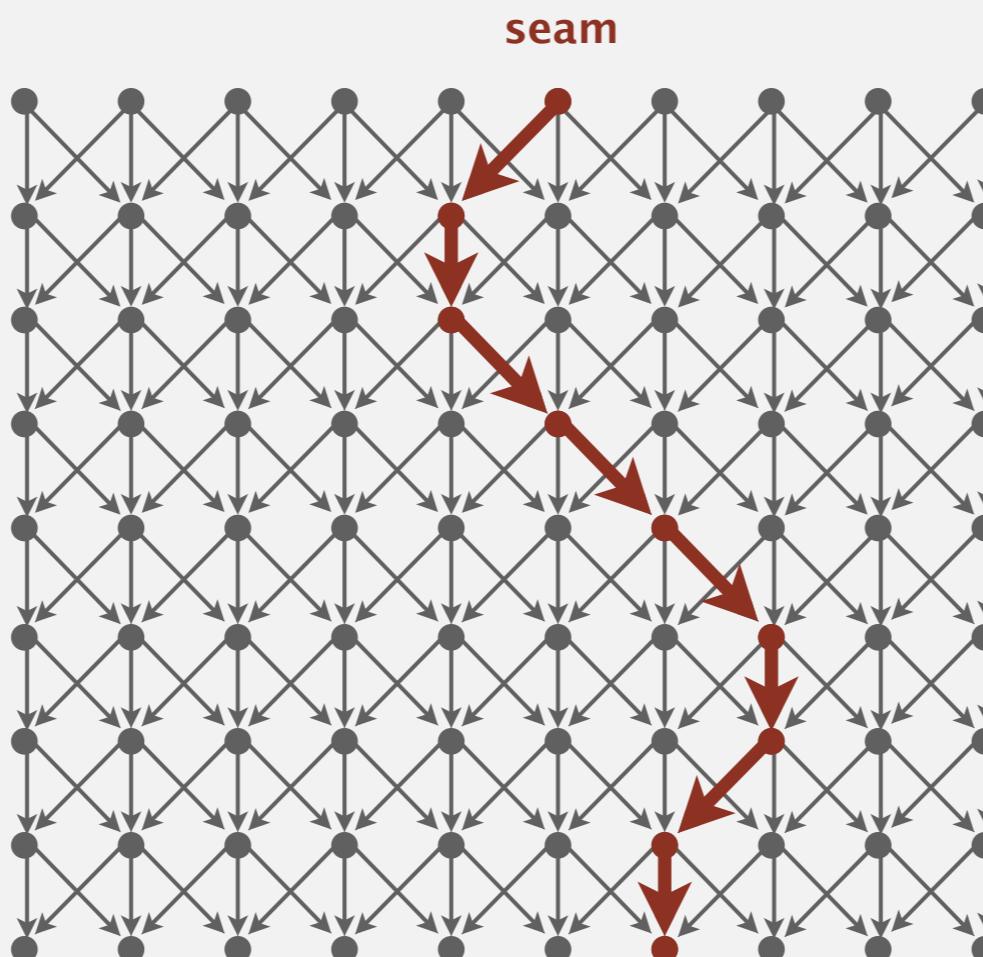
- Grid DAG: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = energy function of 8 neighboring pixels.
- Seam = shortest path (sum of vertex weights) from top to bottom.



Content-aware resizing

To find vertical seam:

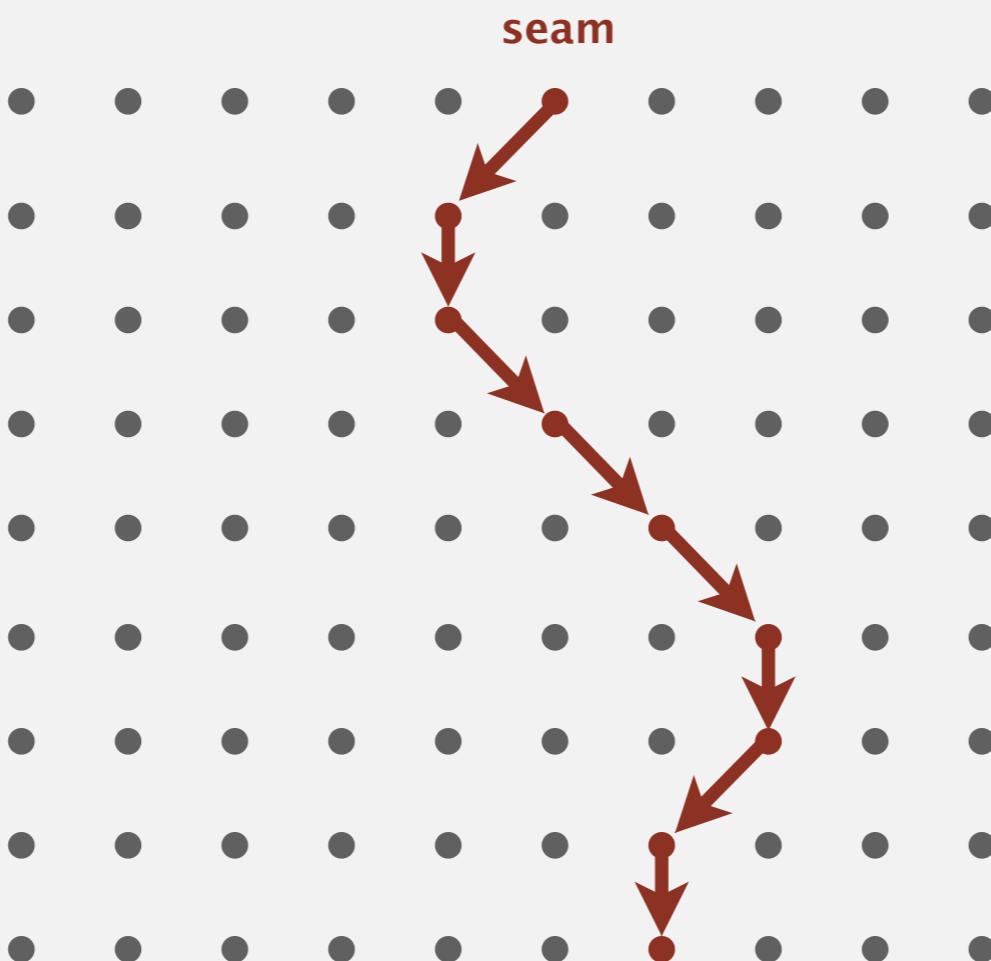
- Grid DAG: vertex = pixel; edge = from pixel to 3 downward neighbors.
- Weight of pixel = energy function of 8 neighboring pixels.
- Seam = shortest path (sum of vertex weights) from top to bottom.



Content-aware resizing

To remove vertical seam:

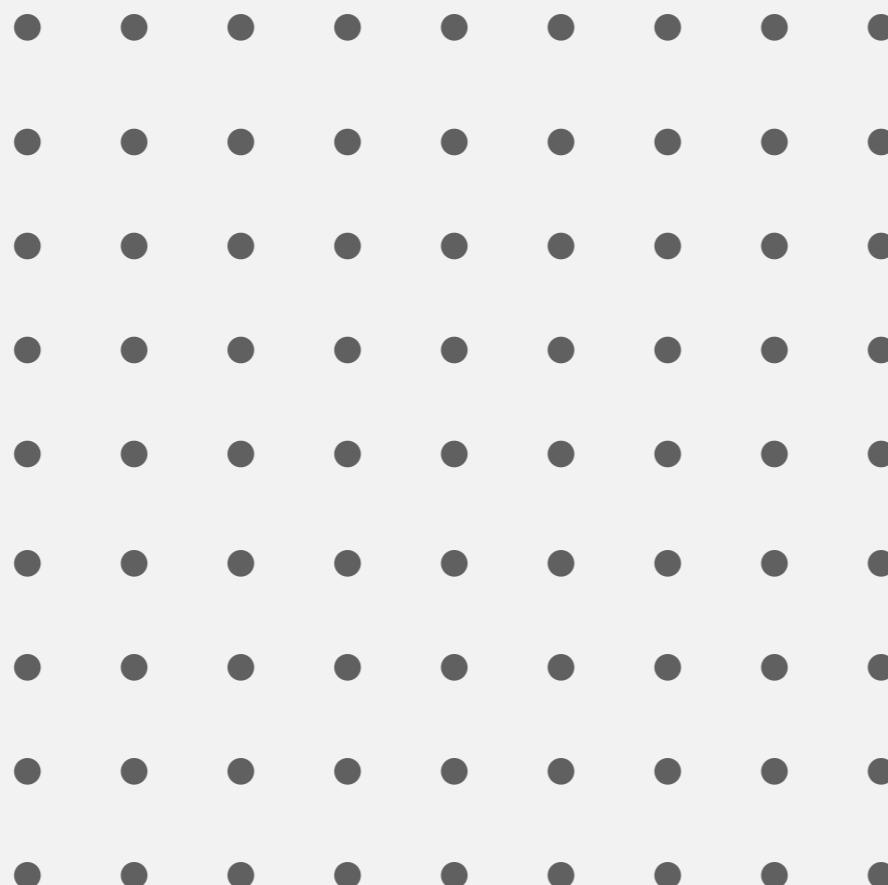
- Delete pixels on seam (one in each row).



Content-aware resizing

To remove vertical seam:

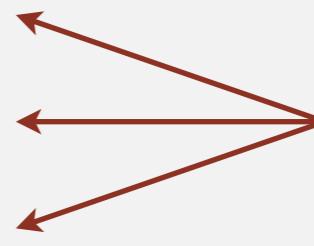
- Delete pixels on seam (one in each row).



Longest paths in edge-weighted DAGs

Formulate as a shortest paths problem in edge-weighted DAGs.

- Negate all weights.
- Find shortest paths.
- Negate weights in result.



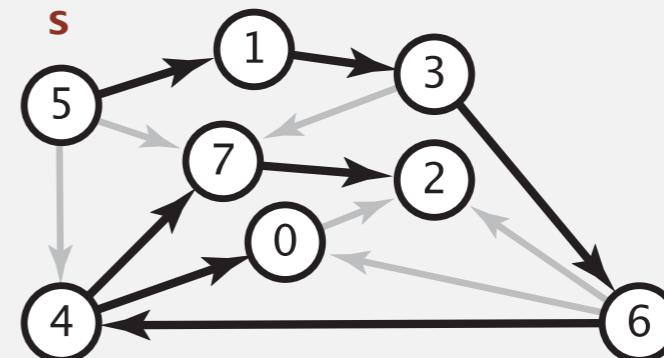
equivalent: reverse sense of equality in relax()

longest paths input

5->4	0.35
4->7	0.37
5->7	0.28
5->1	0.32
4->0	0.38
0->2	0.26
3->7	0.39
1->3	0.29
7->2	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93

shortest paths input

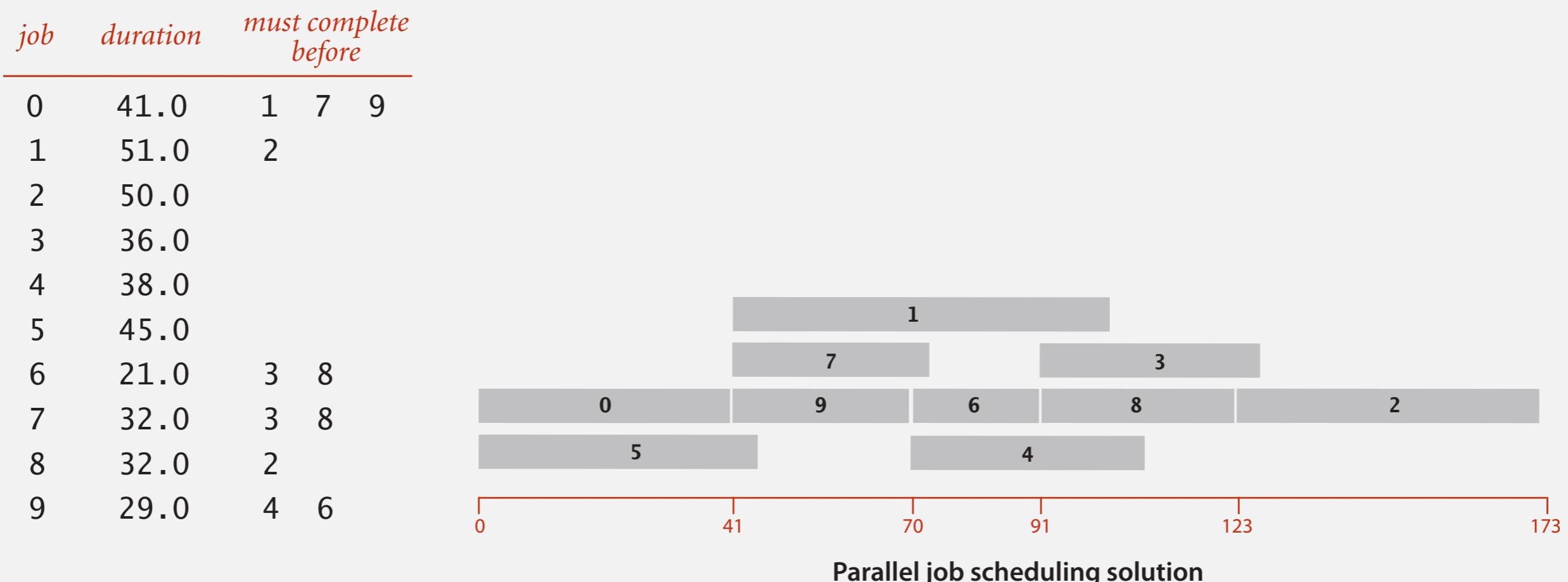
5->4	-0.35
4->7	-0.37
5->7	-0.28
5->1	-0.32
4->0	-0.38
0->2	-0.26
3->7	-0.39
1->3	-0.29
7->2	-0.34
6->2	-0.40
3->6	-0.52
6->0	-0.58
6->4	-0.93



Key point. Topological sort algorithm works even with negative weights.

Longest paths in edge-weighted DAGs: application

Parallel job scheduling. Given a set of jobs with durations and precedence constraints, schedule the jobs (by finding a start time for each) so as to achieve the minimum completion time, while respecting the constraints.

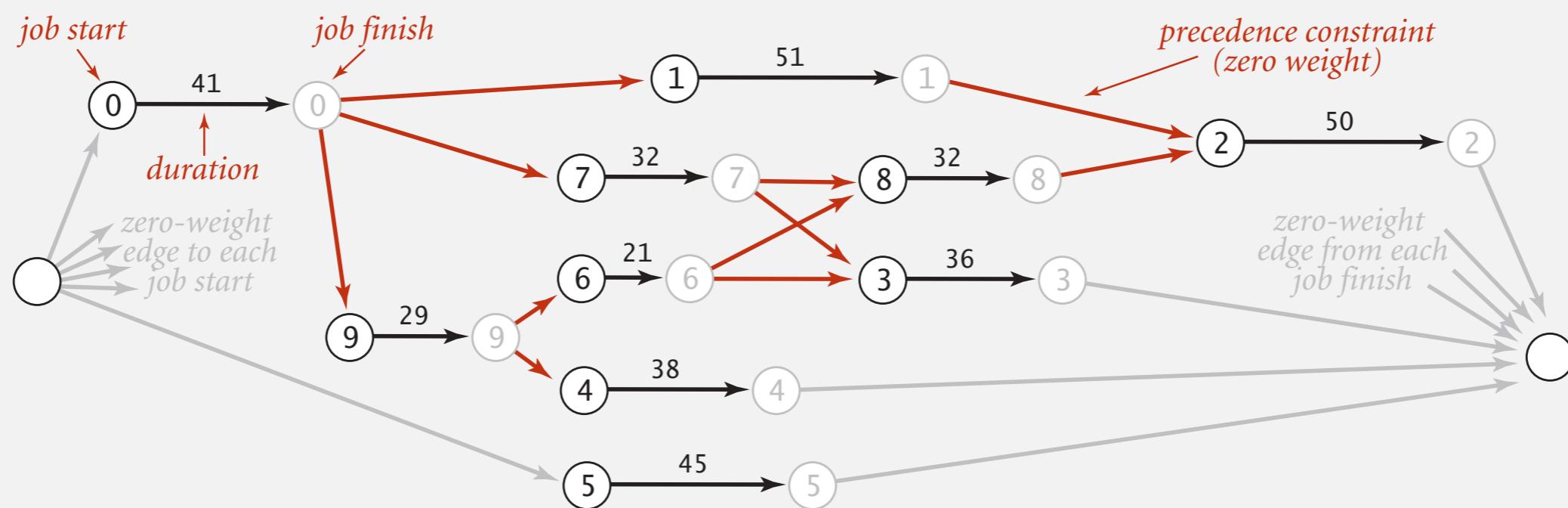


Critical path method

CPM. To solve a parallel job-scheduling problem, create edge-weighted DAG:

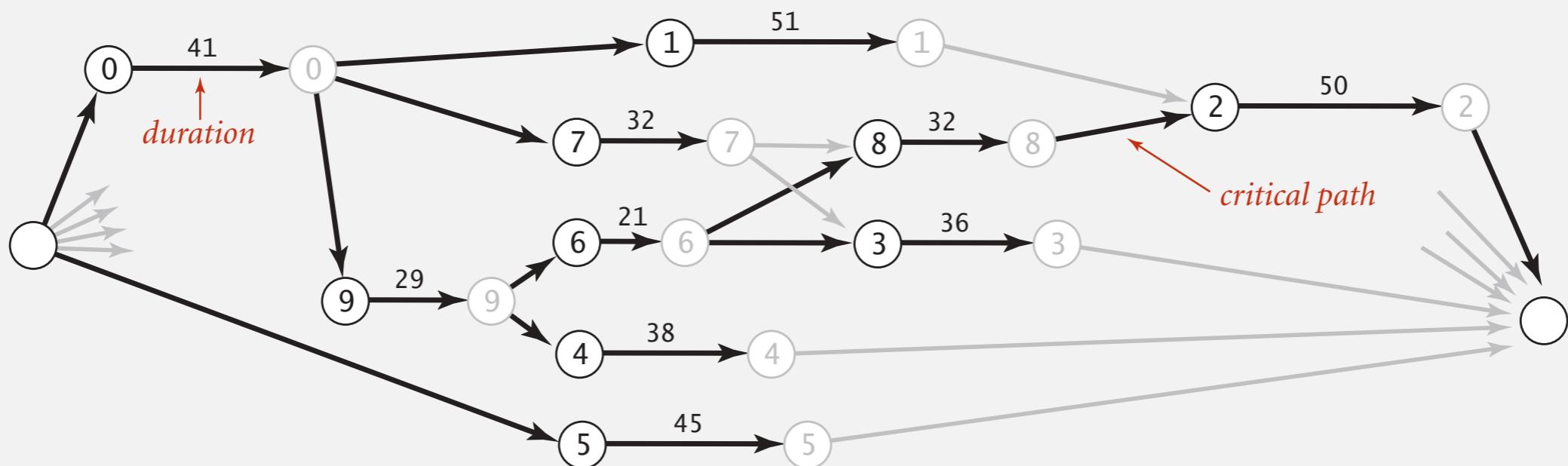
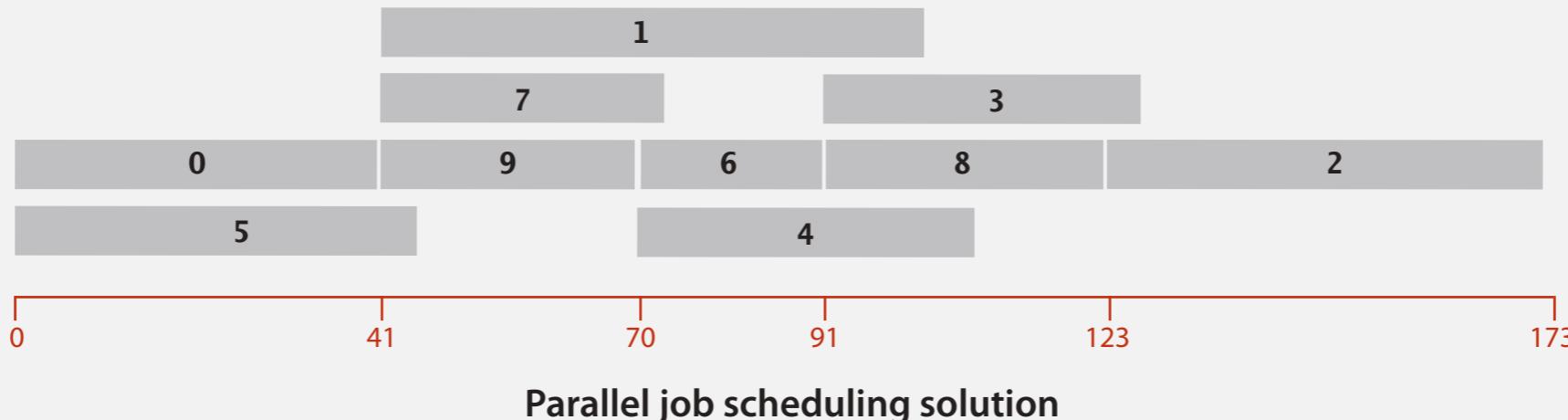
- Source and sink vertices.
- Two vertices (begin and end) for each job.
- Three edges for each job.
 - begin to end (weighted by duration)
 - source to begin (0 weight)
 - end to sink (0 weight)
- One edge for each precedence constraint (0 weight).

<i>job</i>	<i>duration</i>	<i>must complete before</i>		
0	41.0	1	7	9
1	51.0	2		
2	50.0			
3	36.0			
4	38.0			
5	45.0			
6	21.0	3	8	
7	32.0	3	8	
8	32.0	2		
9	29.0	4	6	



Critical path method

CPM. Use **longest path** from the source to schedule each job.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.4 SHORTEST PATHS

- ▶ APIs
- ▶ *shortest-paths properties*
- ▶ *Dijkstra's algorithm*
- ▶ **edge-weighted DAGs**
- ▶ *negative weights*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

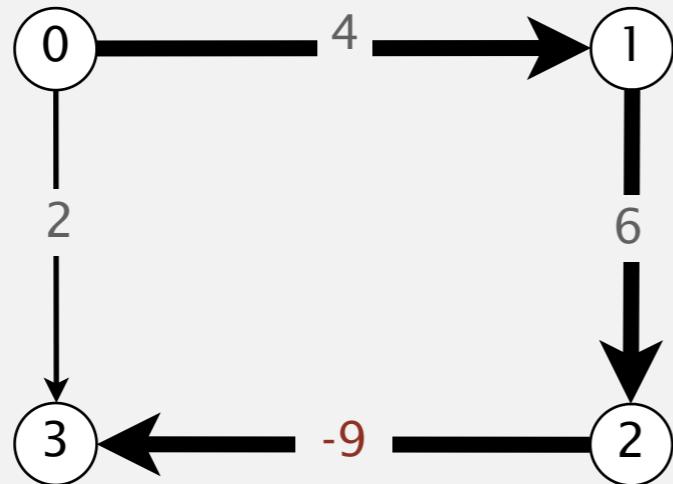
<http://algs4.cs.princeton.edu>

4.4 SHORTEST PATHS

- ▶ *APIs*
- ▶ *shortest-paths properties*
- ▶ *Dijkstra's algorithm*
- ▶ *edge-weighted DAGs*
- ▶ ***negative weights***

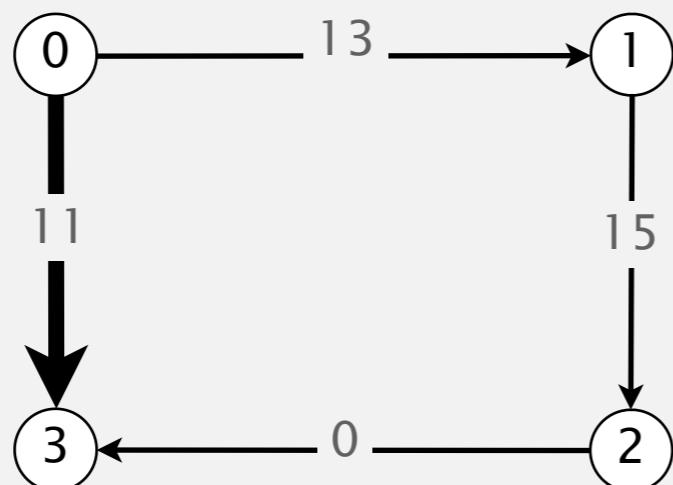
Shortest paths with negative weights: failed attempts

Dijkstra. Doesn't work with negative edge weights.



Dijkstra selects vertex 3 immediately after 0.
But shortest path from 0 to 3 is $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$.

Re-weighting. Add a constant to every edge weight doesn't work.



Adding 9 to each edge weight changes the
shortest path from $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ to $0 \rightarrow 3$.

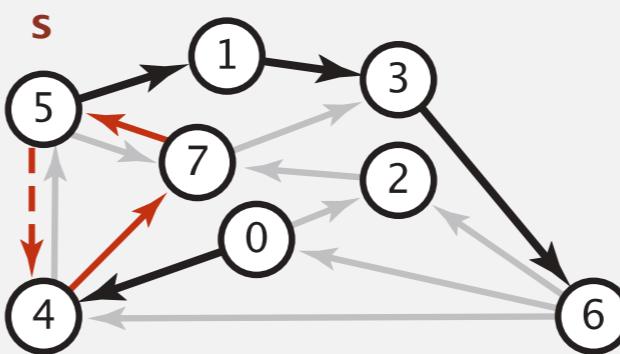
Conclusion. Need a different algorithm.

Negative cycles

Def. A **negative cycle** is a directed cycle whose sum of edge weights is negative.

digraph

4->5	0.35
5->4	-0.66
4->7	0.37
5->7	0.28
7->5	0.28
5->1	0.32
0->4	0.38
0->2	0.26
7->3	0.39
1->3	0.29
2->7	0.34
6->2	0.40
3->6	0.52
6->0	0.58
6->4	0.93



negative cycle $(-0.66 + 0.37 + 0.28)$

$5 \rightarrow 4 \rightarrow 7 \rightarrow 5$

shortest path from 0 to 6

$0 \rightarrow 4 \rightarrow 7 \rightarrow 5 \rightarrow 4 \rightarrow 7 \rightarrow 5 \dots \rightarrow 1 \rightarrow 3 \rightarrow 6$

Proposition. A SPT exists iff no negative cycles.

assuming all vertices reachable from s

Bellman-Ford algorithm

Bellman-Ford algorithm

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat V times:

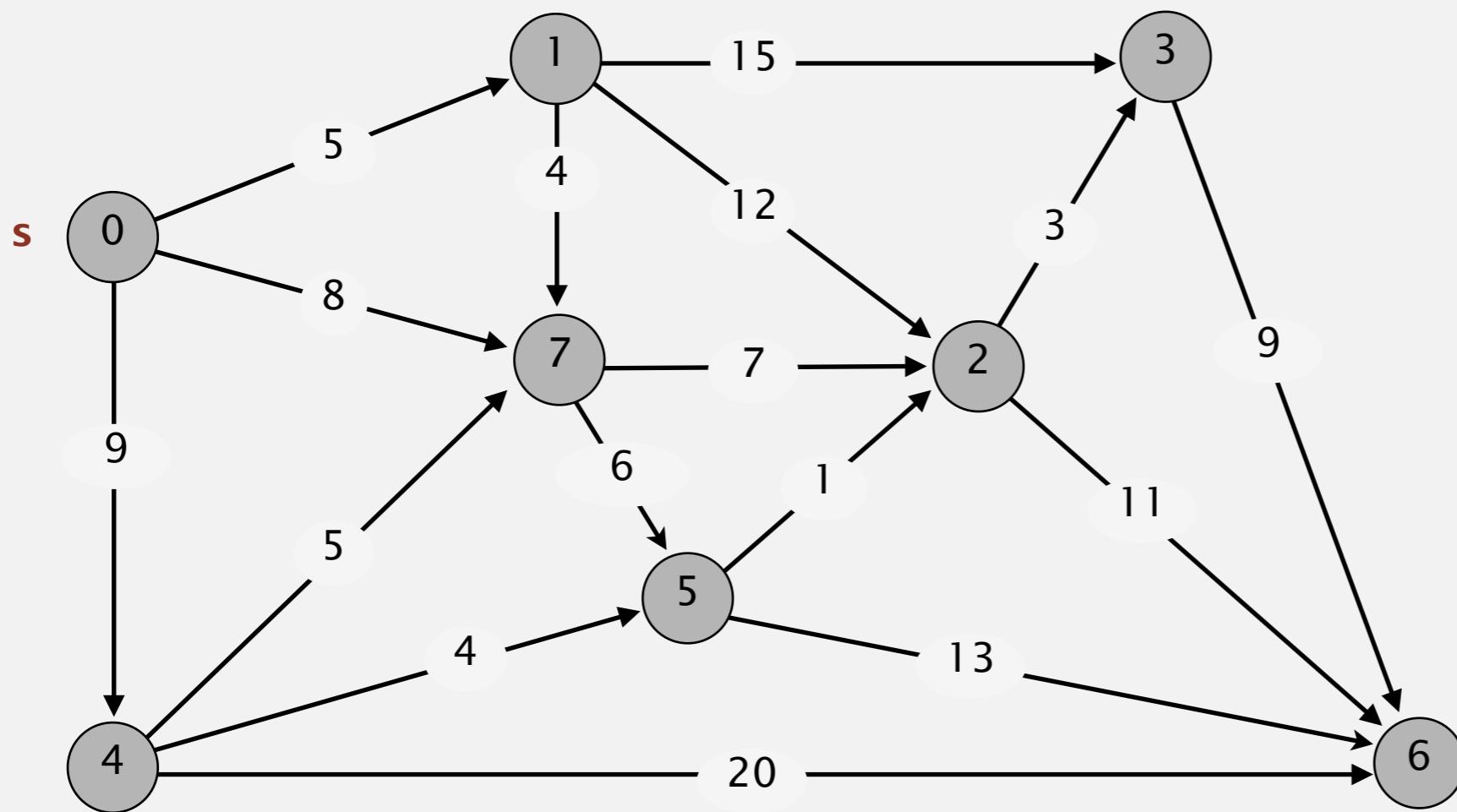
- Relax each edge.

```
for (int i = 0; i < G.V(); i++)
    for (int v = 0; v < G.V(); v++)
        for (DirectedEdge e : G.adj(v))
            relax(e);
```

← pass i (relax each edge)

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.

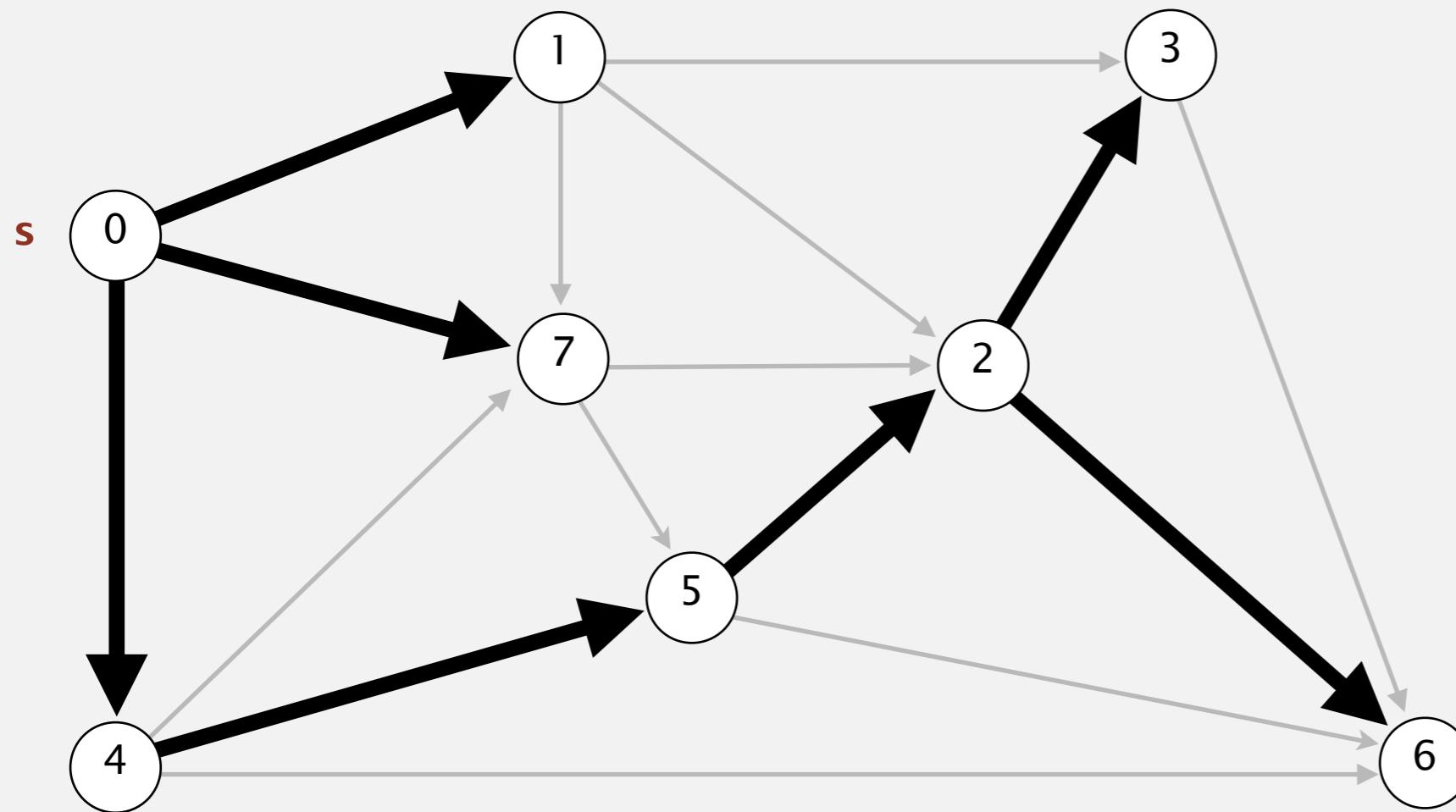


0→1	5.0
0→4	9.0
0→7	8.0
1→2	12.0
1→3	15.0
1→7	4.0
2→3	3.0
2→6	11.0
3→6	9.0
4→5	4.0
4→6	20.0
4→7	5.0
5→2	1.0
5→6	13.0
7→5	6.0
7→2	7.0

an edge-weighted digraph

Bellman-Ford algorithm demo

Repeat V times: relax all E edges.



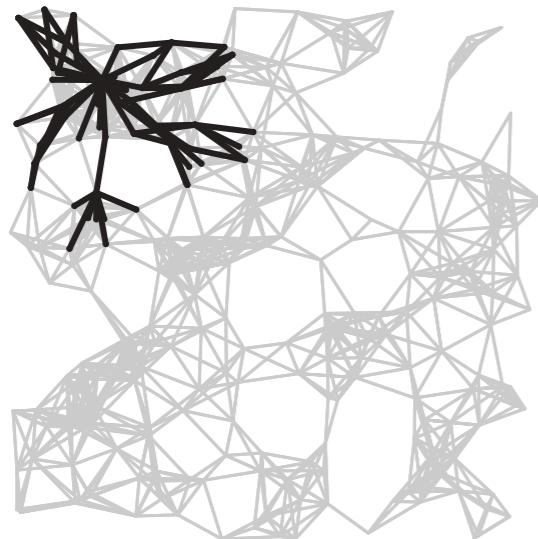
v	distTo[]	edgeTo[]
0	0.0	-
1	5.0	0→1
2	14.0	5→2
3	17.0	2→3
4	9.0	0→4
5	13.0	4→5
6	25.0	2→6
7	8.0	0→7

shortest-paths tree from vertex s

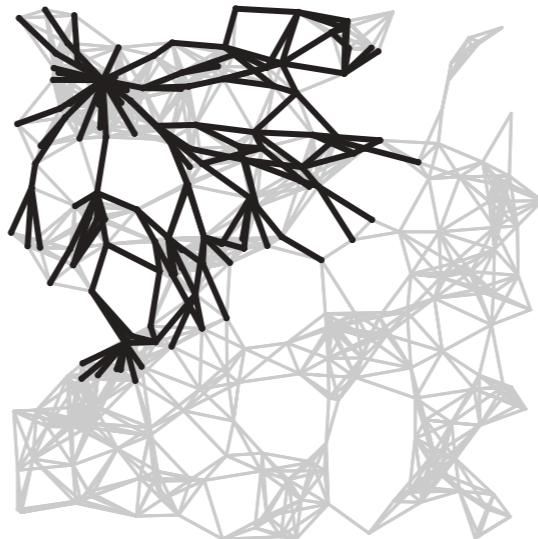
Bellman-Ford algorithm visualization

passes

4



7



10



13



SPT



Bellman-Ford algorithm: analysis

Bellman-Ford algorithm

Initialize $\text{distTo}[s] = 0$ and $\text{distTo}[v] = \infty$ for all other vertices.

Repeat V times:

- Relax each edge.
-

Proposition. Dynamic programming algorithm computes SPT in any edge-weighted digraph with no negative cycles in time proportional to $E \times V$.

Pf idea. After pass i , found shortest path containing at most i edges.

Bellman-Ford algorithm: practical improvement

Observation. If $\text{distTo}[v]$ does not change during pass i , no need to relax any edge pointing from v in pass $i+1$.

FIFO implementation. Maintain **queue** of vertices whose $\text{distTo}[]$ changed.



be careful to keep at most one copy
of each vertex on queue (why?)

Overall effect.

- The running time is still proportional to $E \times V$ in worst case.
- But much faster than that in practice.

Single source shortest-paths implementation: cost summary

algorithm	restriction	typical case	worst case	extra space
topological sort	no directed cycles	$E + V$	$E + V$	V
Dijkstra (binary heap)	no negative weights	$E \log V$	$E \log V$	V
Bellman-Ford	no negative cycles	$E V$	$E V$	V
Bellman-Ford (queue-based)		$E + V$	$E V$	V

Remark 1. Directed cycles make the problem harder.

Remark 2. Negative weights make the problem harder.

Remark 3. Negative cycles makes the problem intractable.

Finding a negative cycle

Negative cycle. Add two method to the API for SP.

boolean hasNegativeCycle()

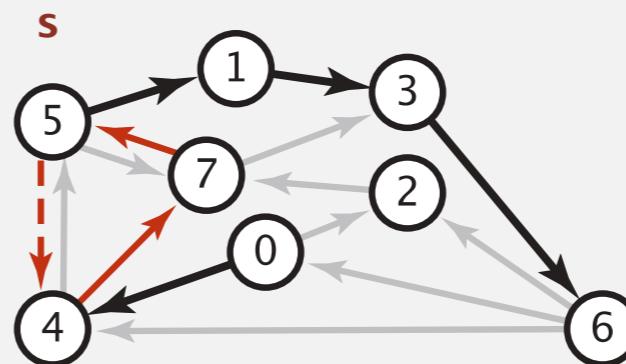
is there a negative cycle?

Iterable <DirectedEdge> negativeCycle()

negative cycle reachable from s

digraph

```
4->5  0.35
5->4 -0.66
4->7  0.37
5->7  0.28
7->5  0.28
5->1  0.32
0->4  0.38
0->2  0.26
7->3  0.39
1->3  0.29
2->7  0.34
6->2  0.40
3->6  0.52
6->0  0.58
6->4  0.93
```

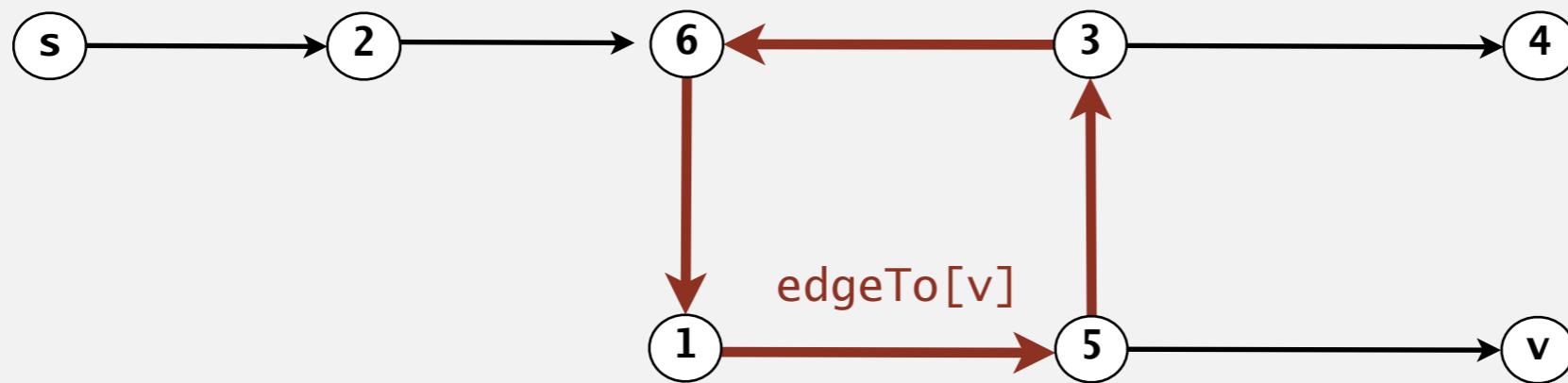


negative cycle (-0.66 + 0.37 + 0.28)

5->4->7->5

Finding a negative cycle

Observation. If there is a negative cycle, Bellman-Ford gets stuck in loop, updating `distTo[]` and `edgeTo[]` entries of vertices in the cycle.



Proposition. If any vertex v is updated in phase v , there exists a negative cycle (and can trace back `edgeTo[v]` entries to find it).

In practice. Check for negative cycles more frequently.

Negative cycle application: arbitrage detection

Problem. Given table of exchange rates, is there an arbitrage opportunity?

	USD	EUR	GBP	CHF	CAD
USD	1	0.741	0.657	1.061	1.011
EUR	1.350	1	0.888	1.433	1.366
GBP	1.521	1.126	1	1.614	1.538
CHF	0.943	0.698	0.620	1	0.953
CAD	0.995	0.732	0.650	1.049	1

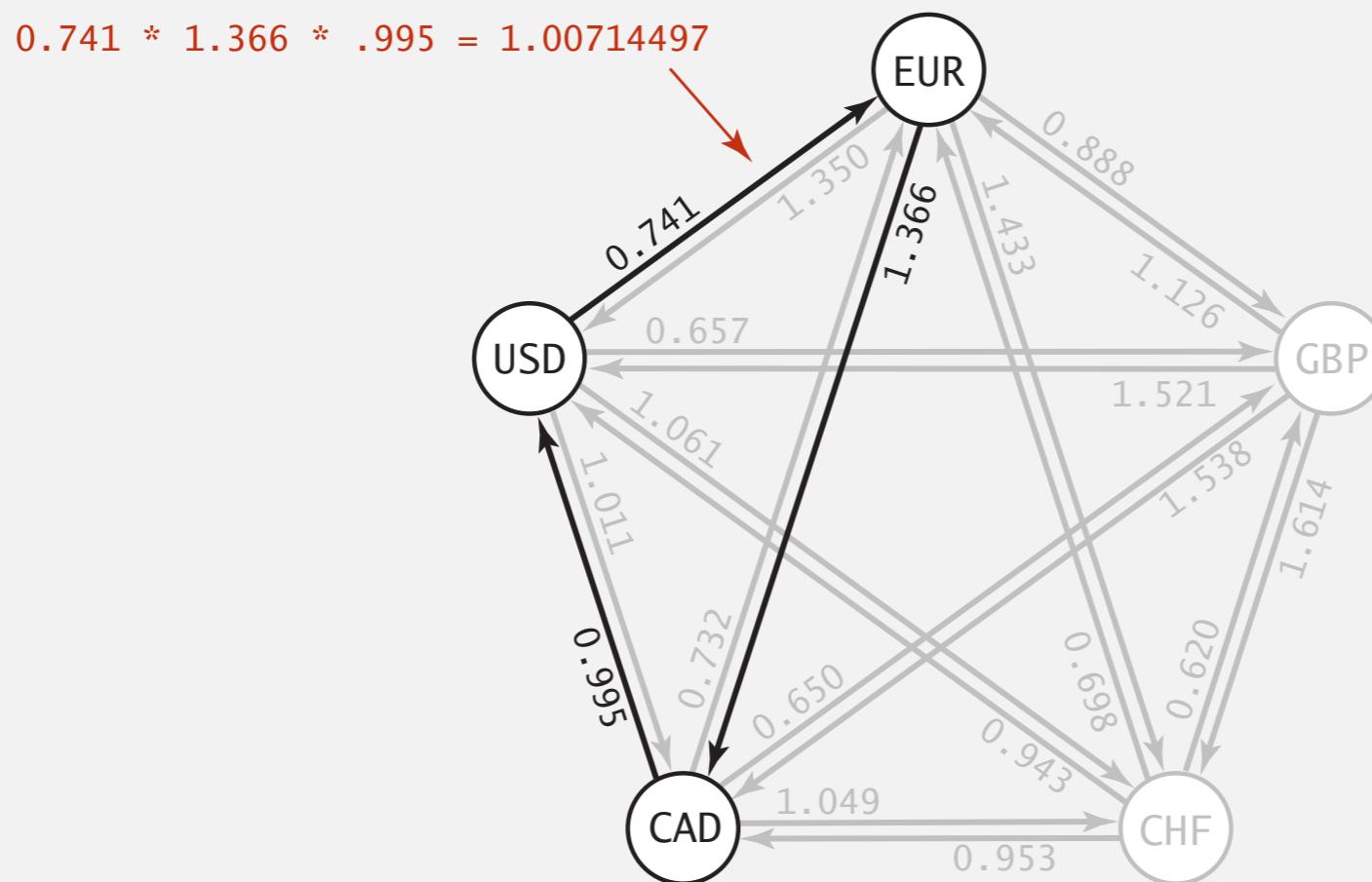
Ex. \$1,000 \Rightarrow 741 Euros \Rightarrow 1,012.206 Canadian dollars \Rightarrow \$1,007.14497.

$$1000 \times 0.741 \times 1.366 \times 0.995 = 1007.14497$$

Negative cycle application: arbitrage detection

Currency exchange graph.

- Vertex = currency.
- Edge = transaction, with weight equal to exchange rate.
- Find a directed cycle whose product of edge weights is > 1 .

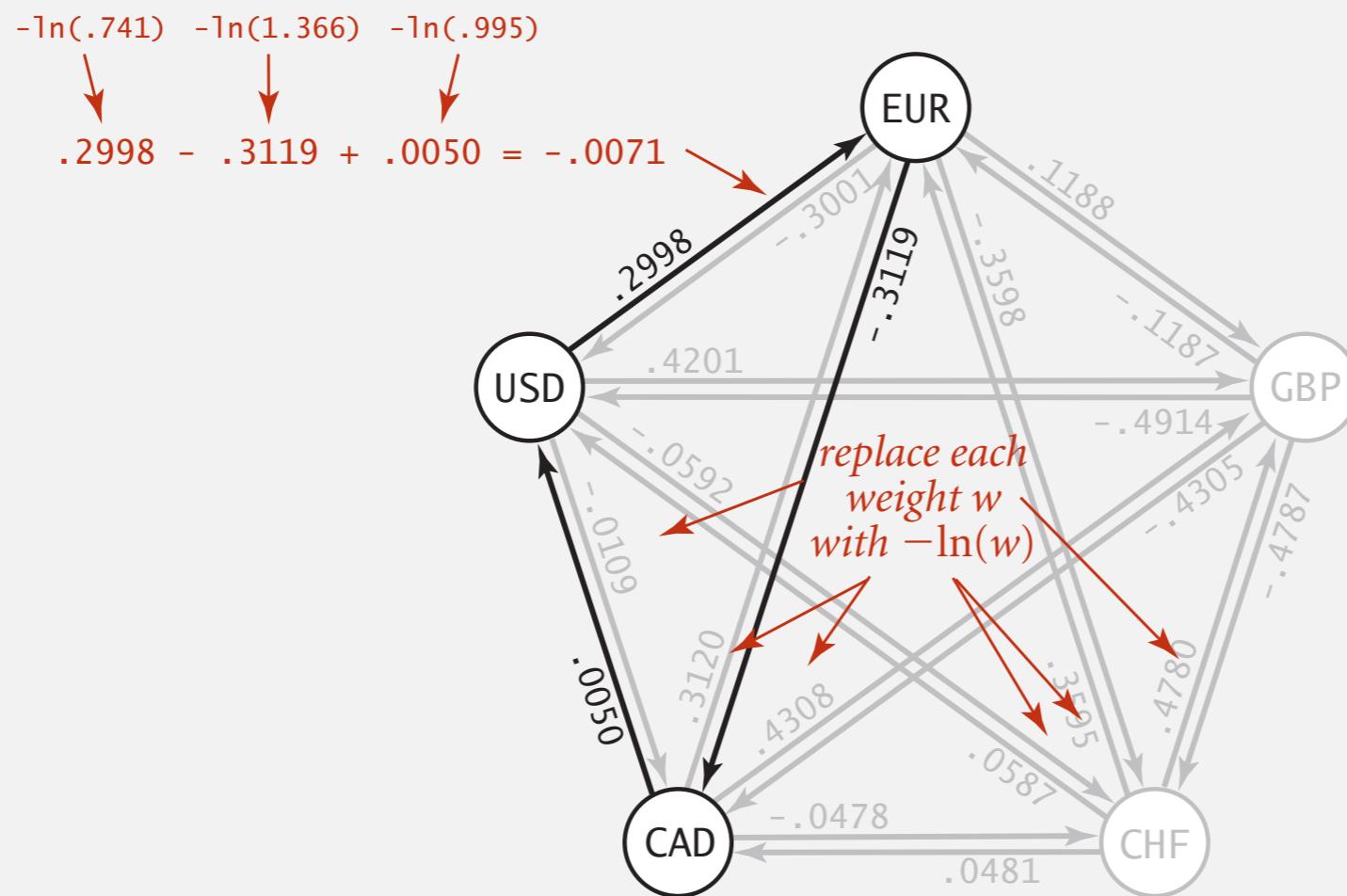


Challenge. Express as a negative cycle detection problem.

Negative cycle application: arbitrage detection

Model as a negative cycle detection problem by taking logs.

- Let weight of edge $v \rightarrow w$ be $-\ln$ (exchange rate from currency v to w).
- Multiplication turns to addition; > 1 turns to < 0 .
- Find a directed cycle whose sum of edge weights is < 0 (negative cycle).



Remark. Fastest algorithm is extraordinarily valuable!

Shortest paths summary

Dijkstra's algorithm.

- Nearly linear-time when weights are nonnegative.
- Generalization encompasses DFS, BFS, and Prim.

Acyclic edge-weighted digraphs.

- Arise in applications.
- Faster than Dijkstra's algorithm.
- Negative weights are no problem.

Negative weights and negative cycles.

- Arise in applications.
- If no negative cycles, can find shortest paths via Bellman-Ford.
- If negative cycles, can find one via Bellman-Ford.

Shortest-paths is a broadly useful problem-solving model.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.4 SHORTEST PATHS

- ▶ *APIs*
- ▶ *shortest-paths properties*
- ▶ *Dijkstra's algorithm*
- ▶ *edge-weighted DAGs*
- ▶ ***negative weights***



ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

4.4 SHORTEST PATHS

- ▶ *APIs*
- ▶ *shortest-paths properties*
- ▶ *Dijkstra's algorithm*
- ▶ *edge-weighted DAGs*
- ▶ *negative weights*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



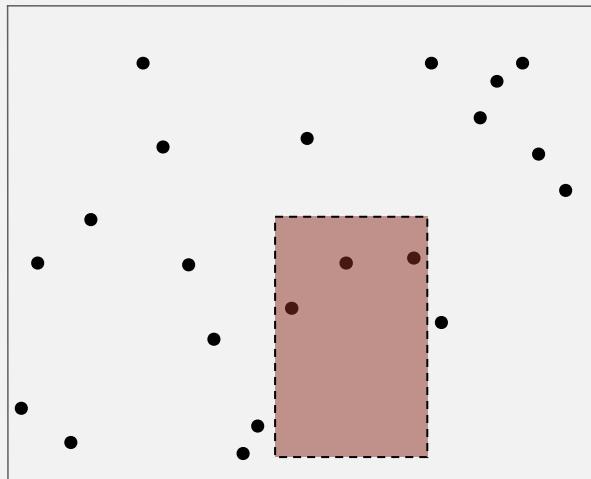
<http://algs4.cs.princeton.edu>

GEOMETRIC APPLICATIONS OF BSTs

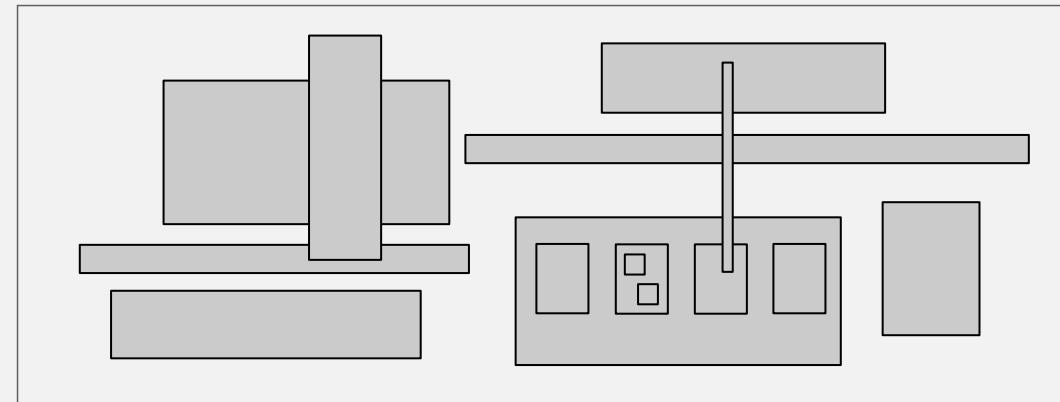
- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

Overview

This lecture. Intersections among **geometric objects**.



2d orthogonal range search



orthogonal rectangle intersection

Applications. CAD, games, movies, virtual reality, databases, GIS,

Efficient solutions. **Binary search trees** (and extensions).

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

1d range search

Extension of ordered symbol table.

- Insert key-value pair.
- Search for key k .
- Delete key k .
- Range search: find all keys between k_1 and k_2 .
- Range count: number of keys between k_1 and k_2 .

Application. Database queries.

Geometric interpretation.

- Keys are point on a line.
- Find/count points in a given 1d interval.



insert B	B
insert D	B D
insert A	A B D
insert I	A B D I
insert H	A B D H I
insert F	A B D F H I
insert P	A B D F H I P
count G to K	2
search G to K	H I

1d range search: elementary implementations

Unordered list. Fast insert, slow range search.

Ordered array. Slow insert, binary search for k_1 and k_2 to do range search.

order of growth of running time for 1d range search

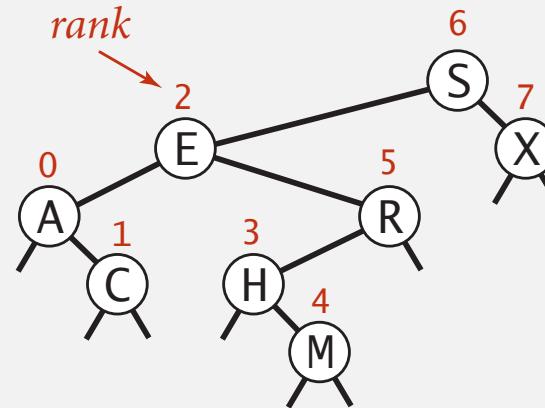
data structure	insert	range count	range search
unordered list	1	N	N
ordered array	N	log N	R + log N
goal	log N	log N	R + log N

N = number of keys

R = number of keys that match

1d range count: BST implementation

1d range count. How many keys between l_o and h_i ?



```
public int size(Key l_o, Key h_i)
{
    if (contains(h_i)) return rank(h_i) - rank(l_o) + 1;
    else                 return rank(h_i) - rank(l_o);
}
```

number of keys < h_i

Proposition. Running time proportional to $\log N$.

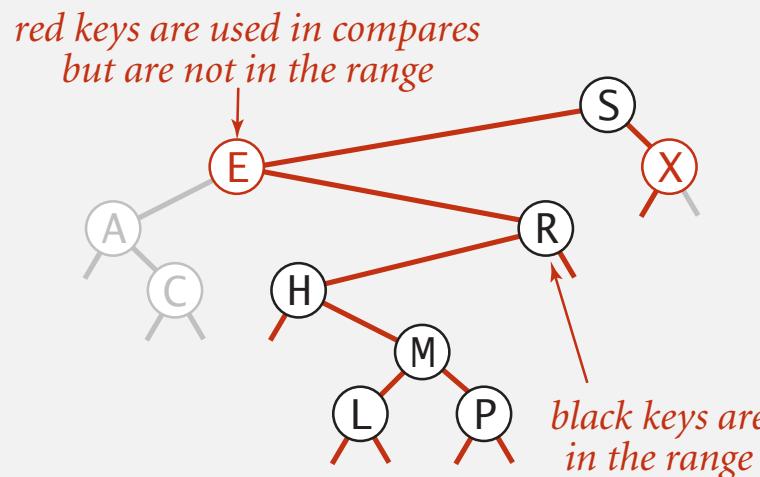
Pf. Nodes examined = search path to l_o + search path to h_i .

1d range search: BST implementation

1d range search. Find all keys between l_0 and h_i .

- Recursively find all keys in left subtree (if any could fall in range).
- Check key in current node.
- Recursively find all keys in right subtree (if any could fall in range).

searching in the range $[F \dots T]$



Proposition. Running time proportional to $R + \log N$.

Pf. Nodes examined = search path to l_0 + search path to h_i + matches.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

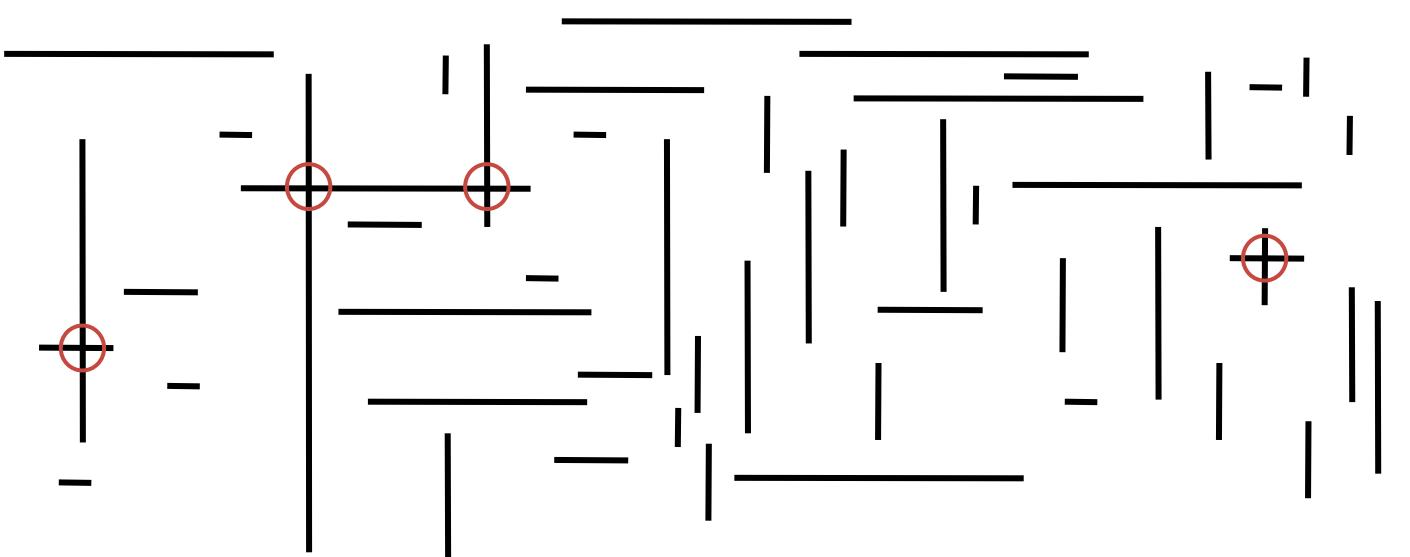
<http://algs4.cs.princeton.edu>

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

Orthogonal line segment intersection

Given N horizontal and vertical line segments, find all intersections.



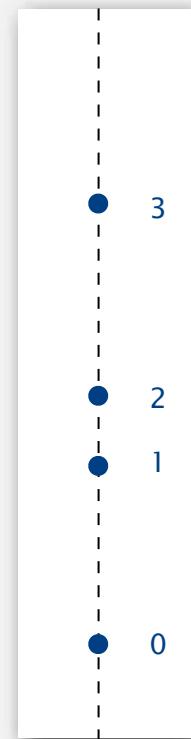
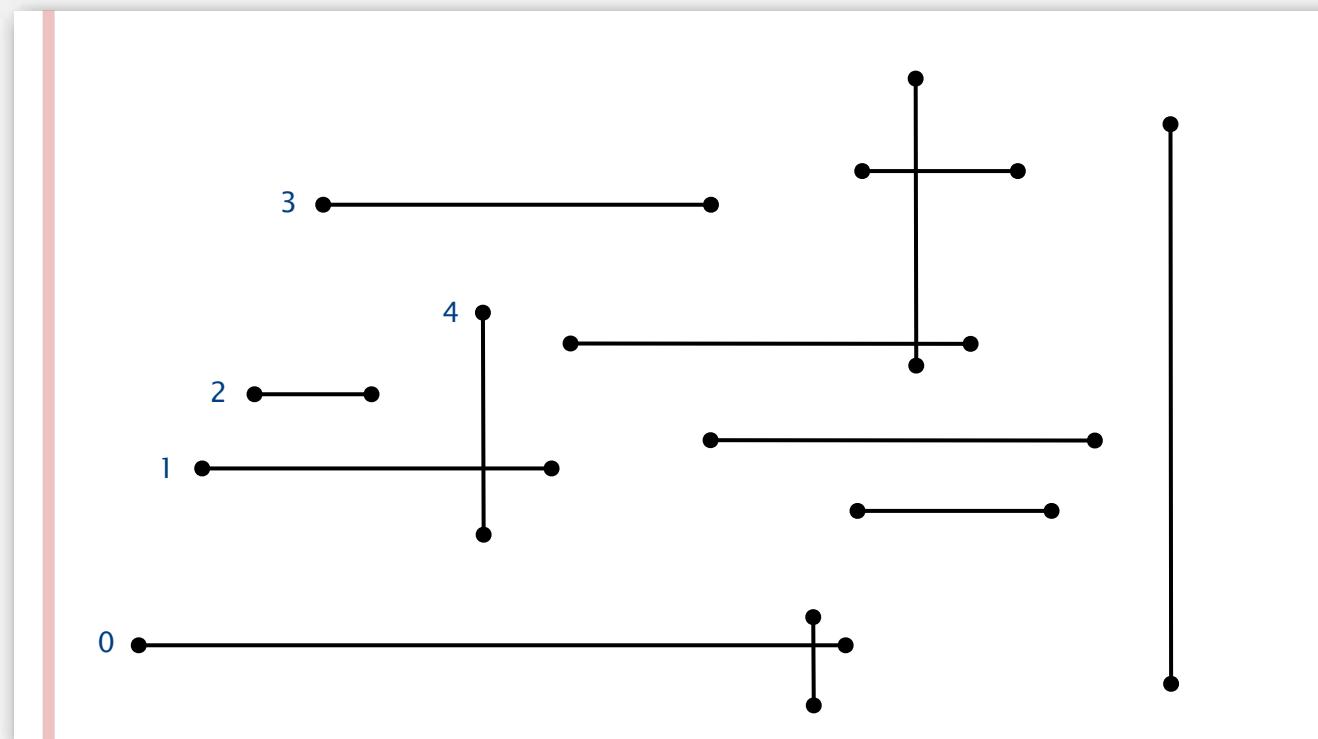
Quadratic algorithm. Check all pairs of line segments for intersection.

Nondegeneracy assumption. All x - and y -coordinates are distinct.

Orthogonal line segment intersection: sweep-line algorithm

Sweep vertical line from left to right.

- x -coordinates define events.
- h -segment (left endpoint): insert y -coordinate into BST.

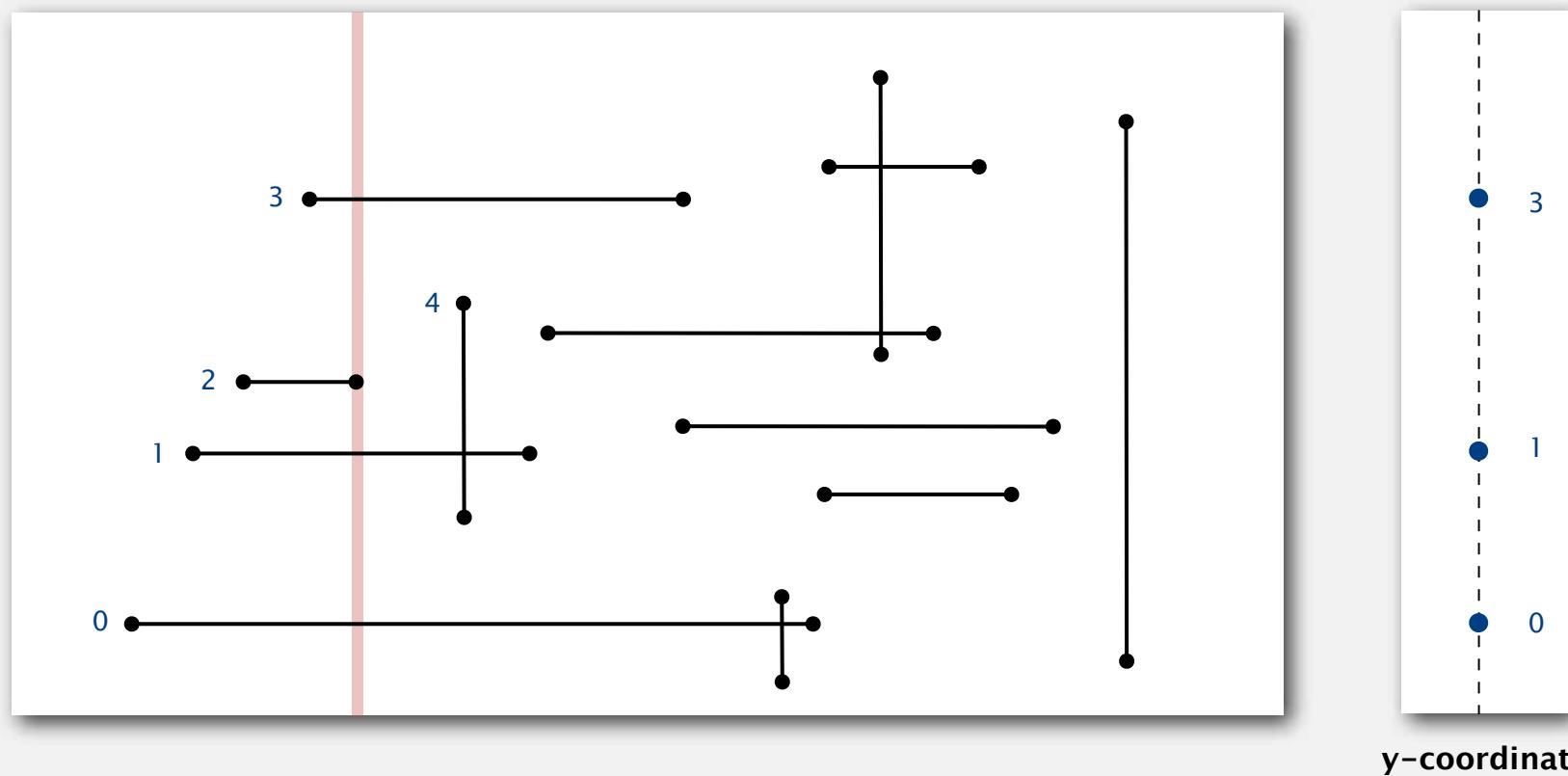


y-coordinates

Orthogonal line segment intersection: sweep-line algorithm

Sweep vertical line from left to right.

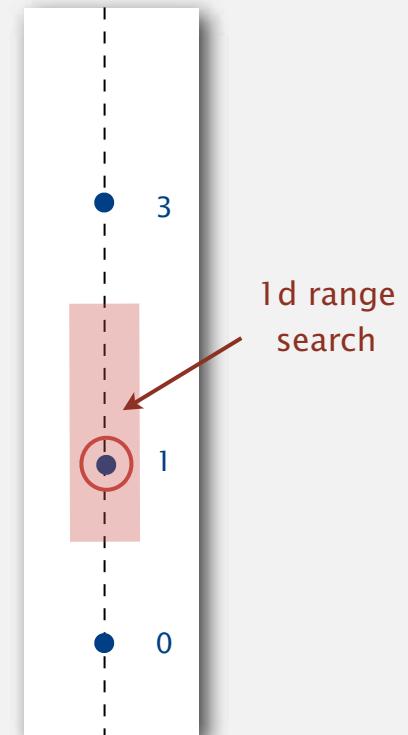
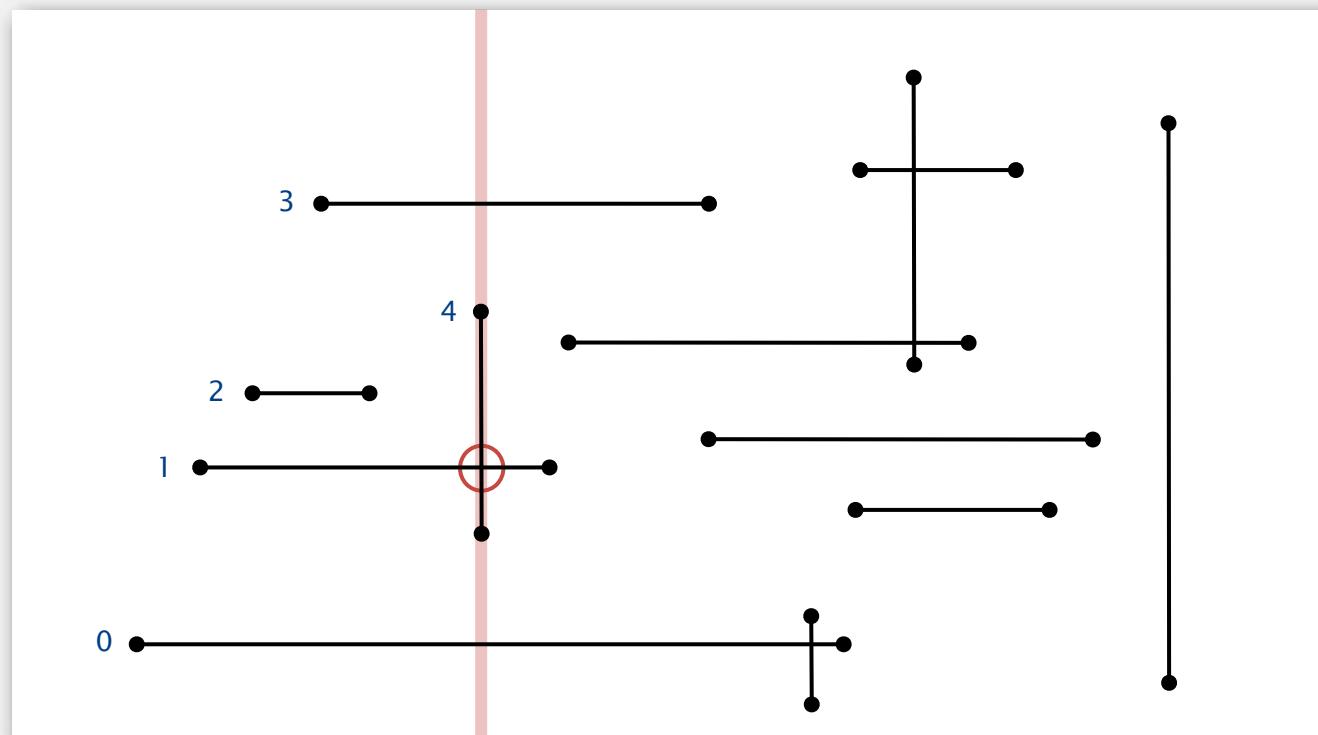
- x -coordinates define events.
- h -segment (left endpoint): insert y -coordinate into BST.
- h -segment (right endpoint): remove y -coordinate from BST.



Orthogonal line segment intersection: sweep-line algorithm

Sweep vertical line from left to right.

- x -coordinates define events.
- h -segment (left endpoint): insert y -coordinate into BST.
- h -segment (right endpoint): remove y -coordinate from BST.
- v -segment: range search for interval of y -endpoints.



y -coordinates

Orthogonal line segment intersection: sweep-line analysis

Proposition. The sweep-line algorithm takes time proportional to $N \log N + R$ to find all R intersections among N orthogonal line segments.

Pf.

- Put x -coordinates on a PQ (or sort). $\leftarrow N \log N$
- Insert y -coordinates into BST. $\leftarrow N \log N$
- Delete y -coordinates from BST. $\leftarrow N \log N$
- Range searches in BST. $\leftarrow N \log N + R$

Bottom line. Sweep line reduces 2d orthogonal line segment intersection search to 1d range search.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

2-d orthogonal range search

Extension of ordered symbol-table to 2d keys.

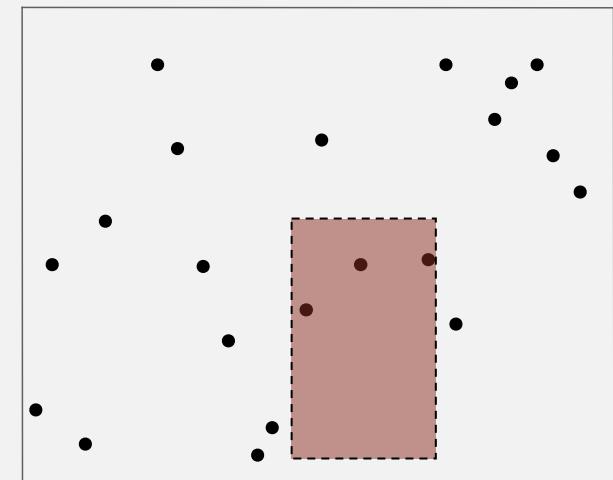
- Insert a 2d key.
- Delete a 2d key.
- Search for a 2d key.
- Range search: find all keys that lie in a 2d range.
- Range count: number of keys that lie in a 2d range.

Applications. Networking, circuit design, databases, ...

Geometric interpretation.

- Keys are point in the plane.
- Find/count points in a given *h-v* rectangle

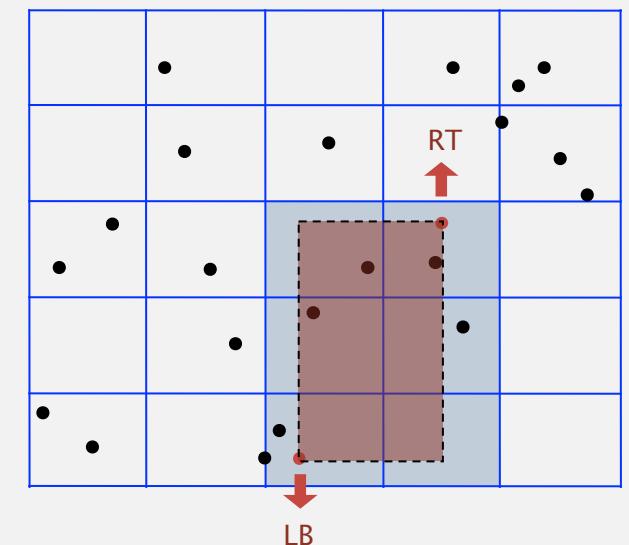
↑
rectangle is axis-aligned



2d orthogonal range search: grid implementation

Grid implementation.

- Divide space into M -by- M grid of squares.
- Create list of points contained in each square.
- Use 2d array to directly index relevant square.
- Insert: add (x, y) to list for corresponding square.
- Range search: examine only squares that intersect 2d range query.



2d orthogonal range search: grid implementation analysis

Space-time tradeoff.

- Space: $M^2 + N$.
- Time: $1 + N/M^2$ per square examined, on average.

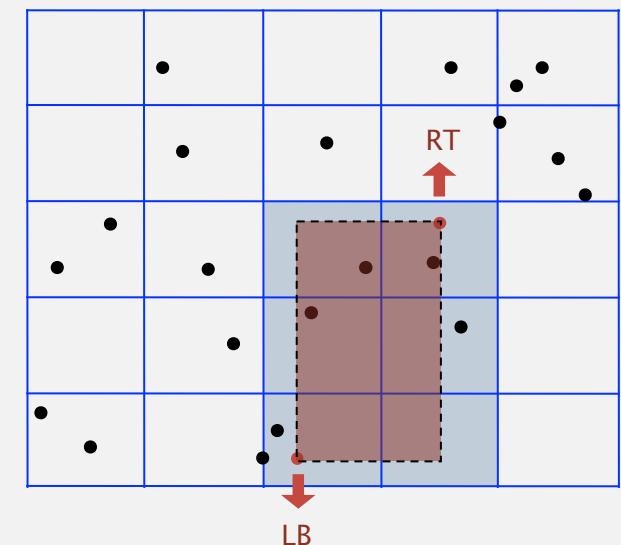
Choose grid square size to tune performance.

- Too small: wastes space.
- Too large: too many points per square.
- Rule of thumb: \sqrt{N} -by- \sqrt{N} grid.

Running time. [if points are evenly distributed]

- Initialize data structure: N .
- Insert point: 1.
- Range search: 1 per point in range.

choose $M \sim \sqrt{N}$

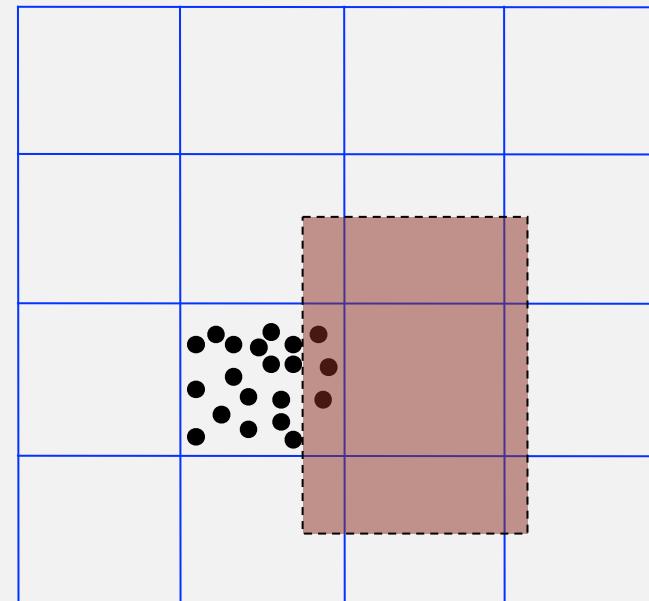


Clustering

Grid implementation. Fast, simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.

- Lists are too long, even though average length is short.
- Need data structure that adapts gracefully to data.



Clustering

Grid implementation. Fast, simple solution for evenly-distributed points.

Problem. Clustering a well-known phenomenon in geometric data.

Ex. USA map data.



13,000 points, 1000 grid squares



half the squares are empty

half the points are
in 10% of the squares

Space-partitioning trees

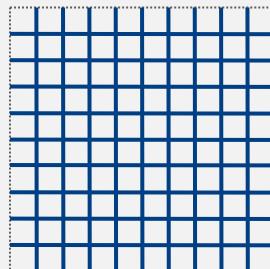
Use a **tree** to represent a recursive subdivision of 2d space.

Grid. Divide space uniformly into squares.

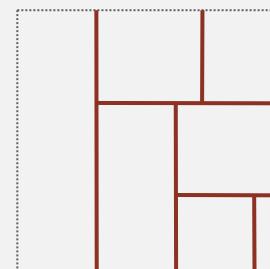
2d tree. Recursively divide space into two halfplanes.

Quadtree. Recursively divide space into four quadrants.

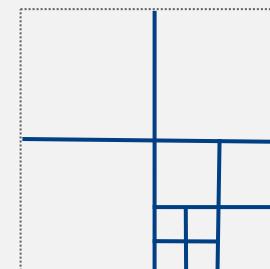
BSP tree. Recursively divide space into two regions.



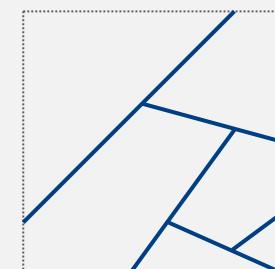
Grid



2d tree



Quadtree

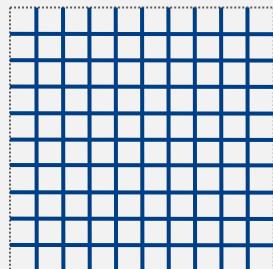


BSP tree

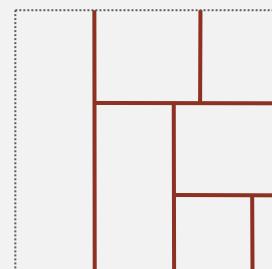
Space-partitioning trees: applications

Applications.

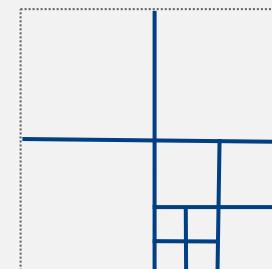
- Ray tracing.
- **2d range search.**
- Flight simulators.
- N-body simulation.
- Collision detection.
- Astronomical databases.
- **Nearest neighbor search.**
- Adaptive mesh generation.
- Accelerate rendering in Doom.
- Hidden surface removal and shadow casting.



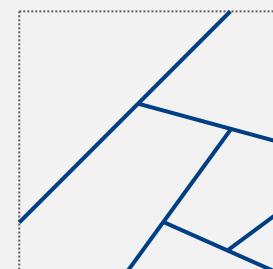
Grid



2d tree



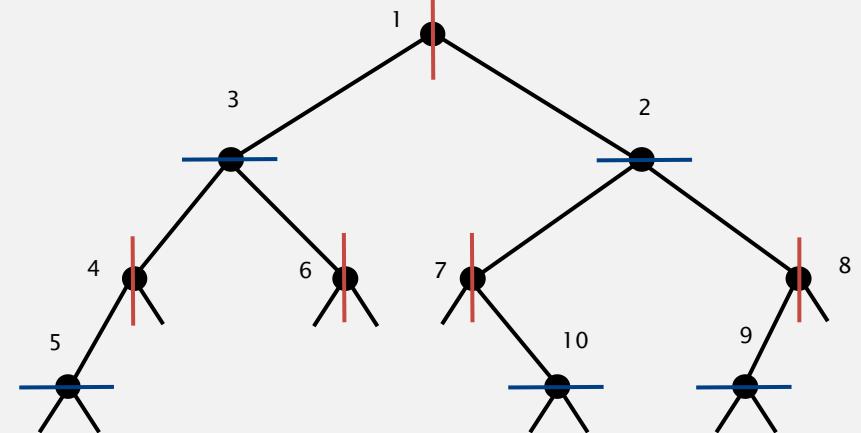
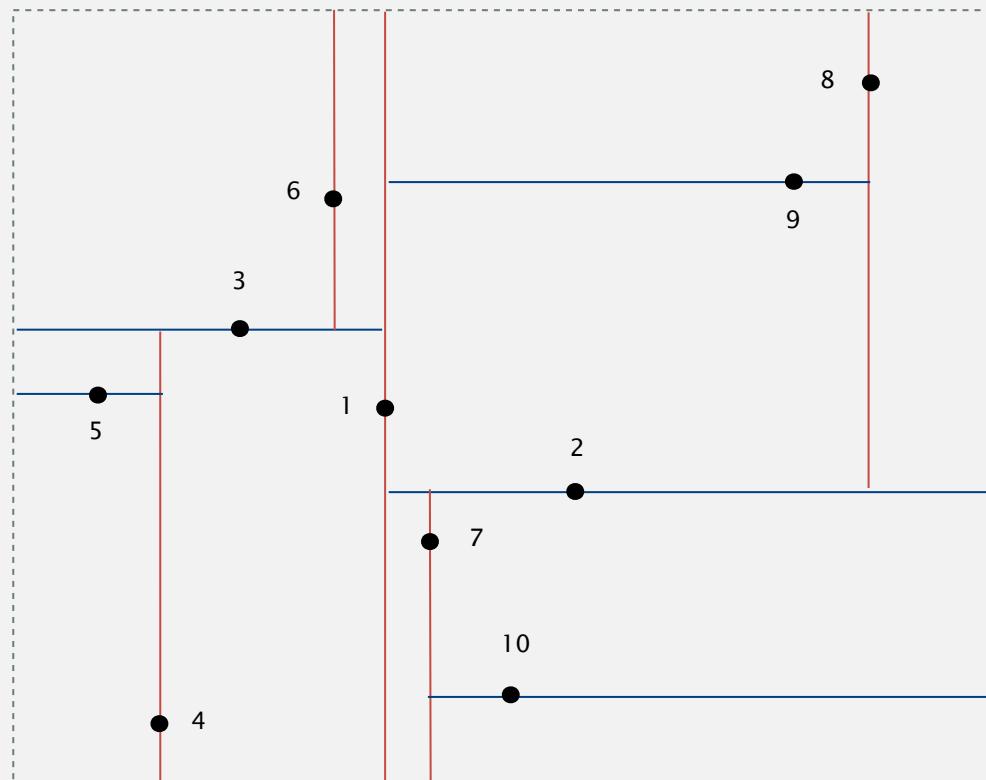
Quadtree



BSP tree

2d tree construction

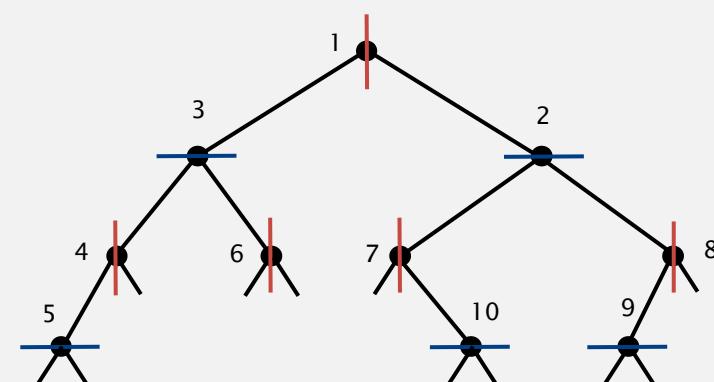
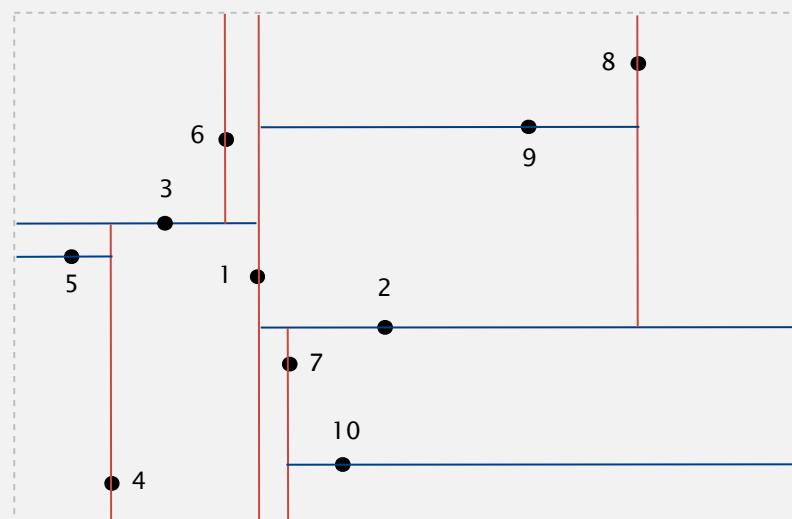
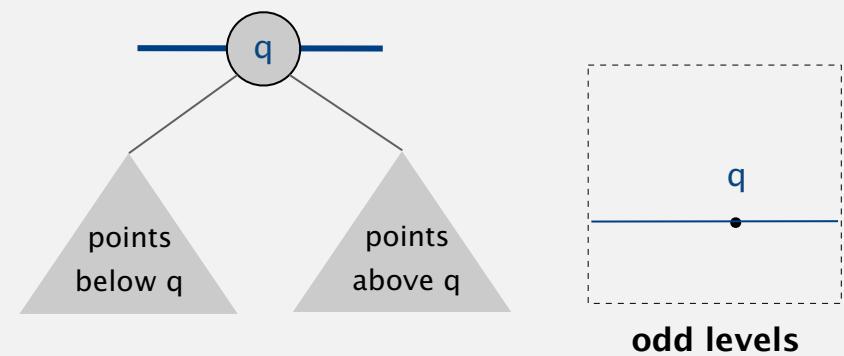
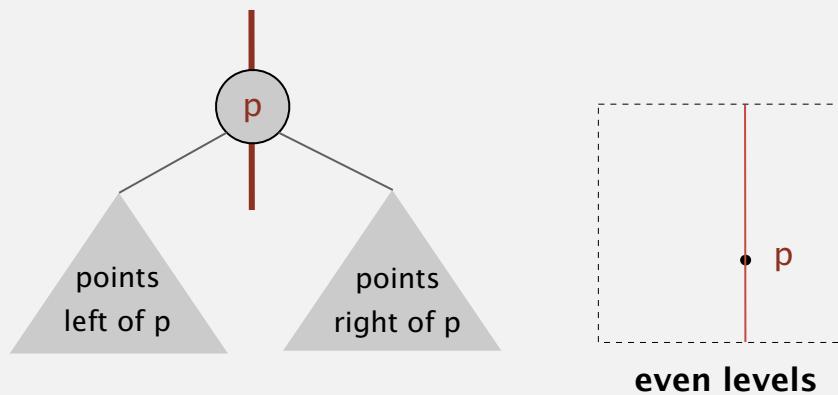
Recursively partition plane into two halfplanes.



2d tree implementation

Data structure. BST, but alternate using x - and y -coordinates as key.

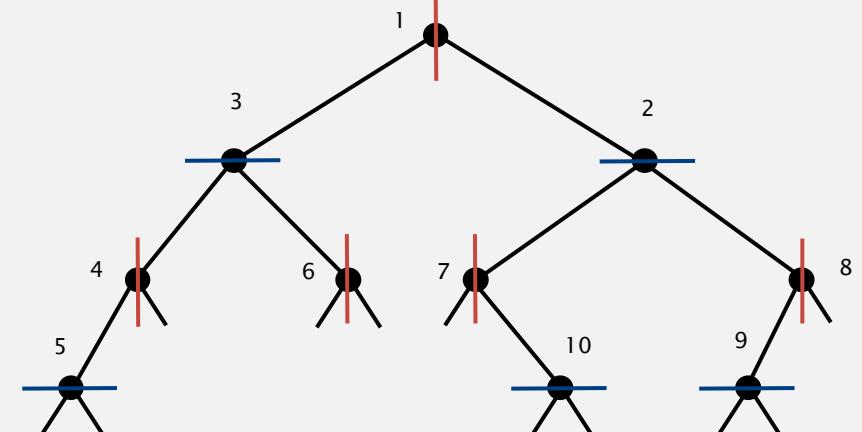
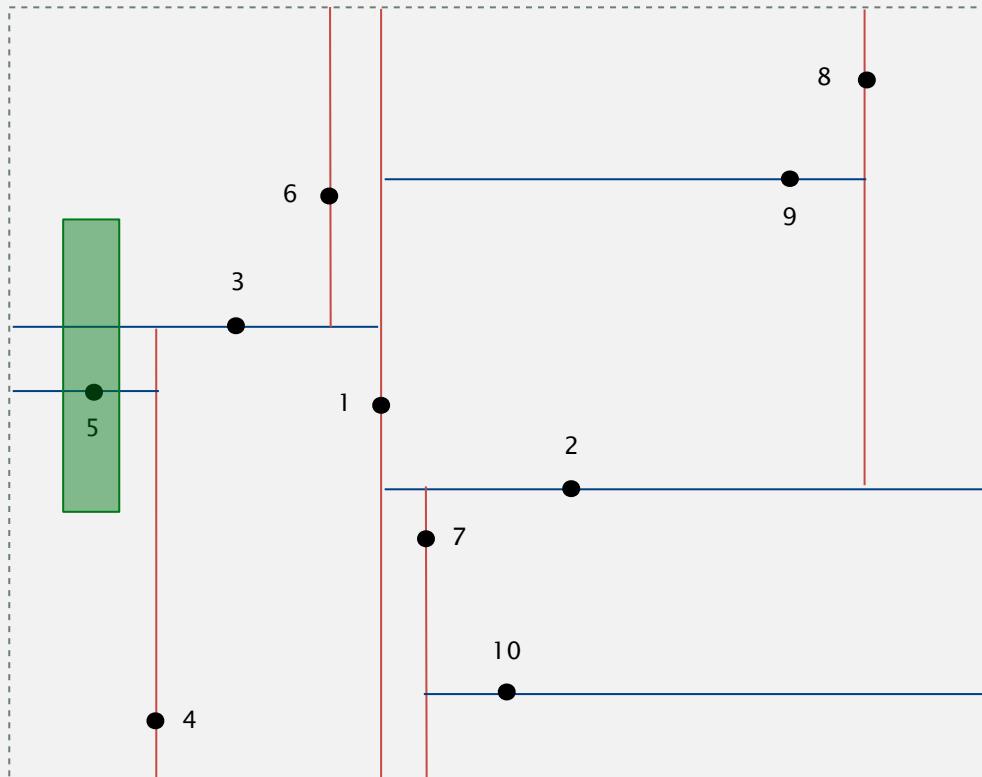
- Search gives rectangle containing point.
- Insert further subdivides the plane.



Range search in a 2d tree demo

Goal. Find all points in a query axis-aligned rectangle.

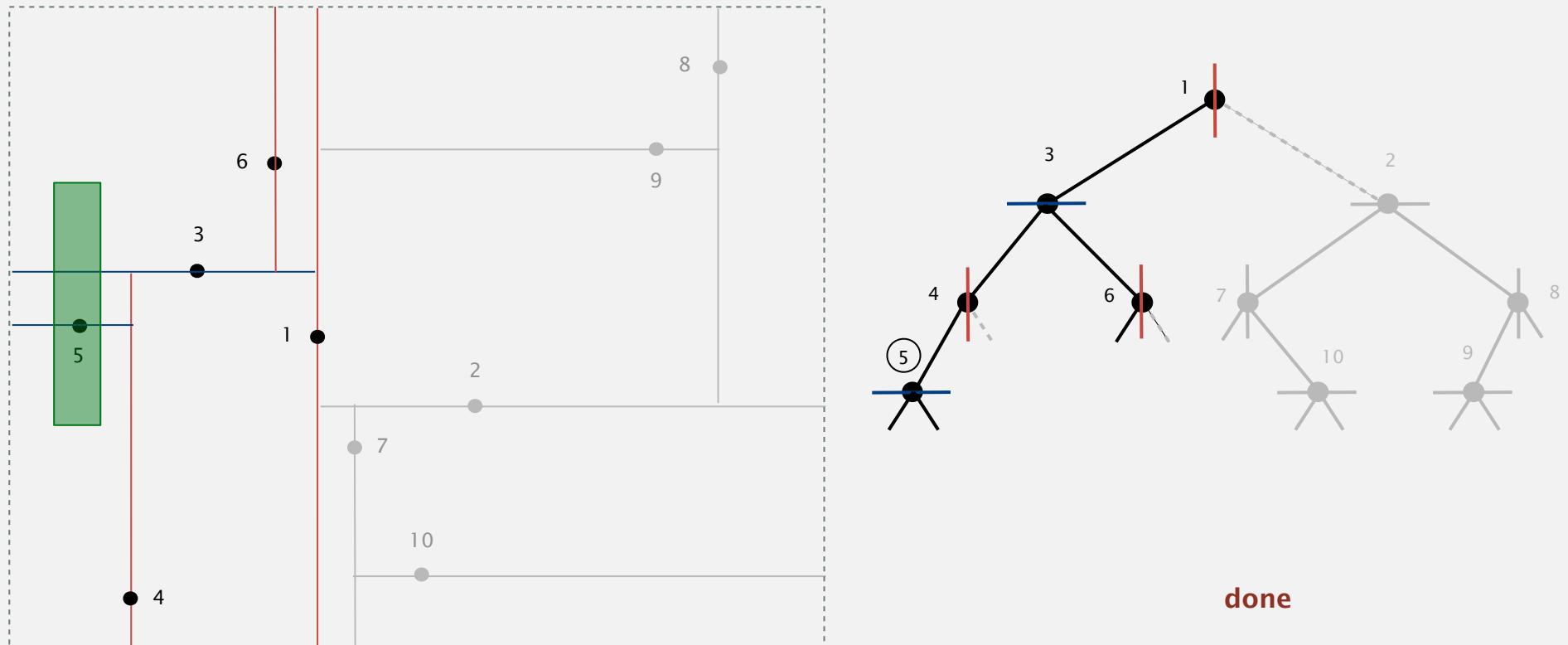
- Check if point in node lies in given rectangle.
- Recursively search left/bottom (if any could fall in rectangle).
- Recursively search right/top (if any could fall in rectangle).



Range search in a 2d tree demo

Goal. Find all points in a query axis-aligned rectangle.

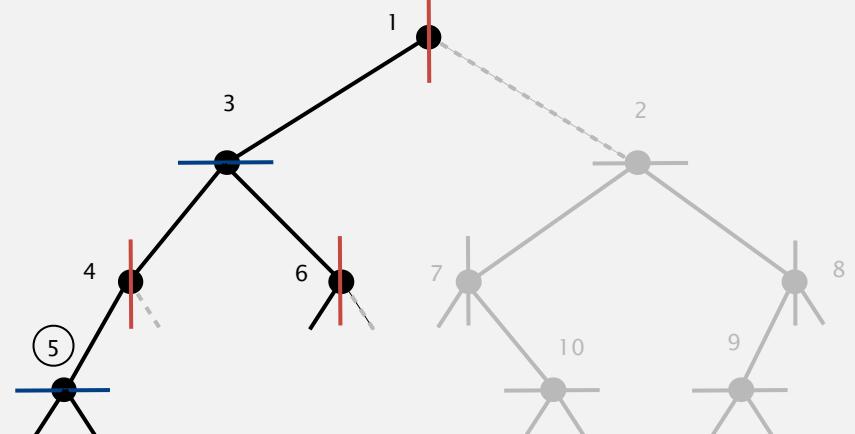
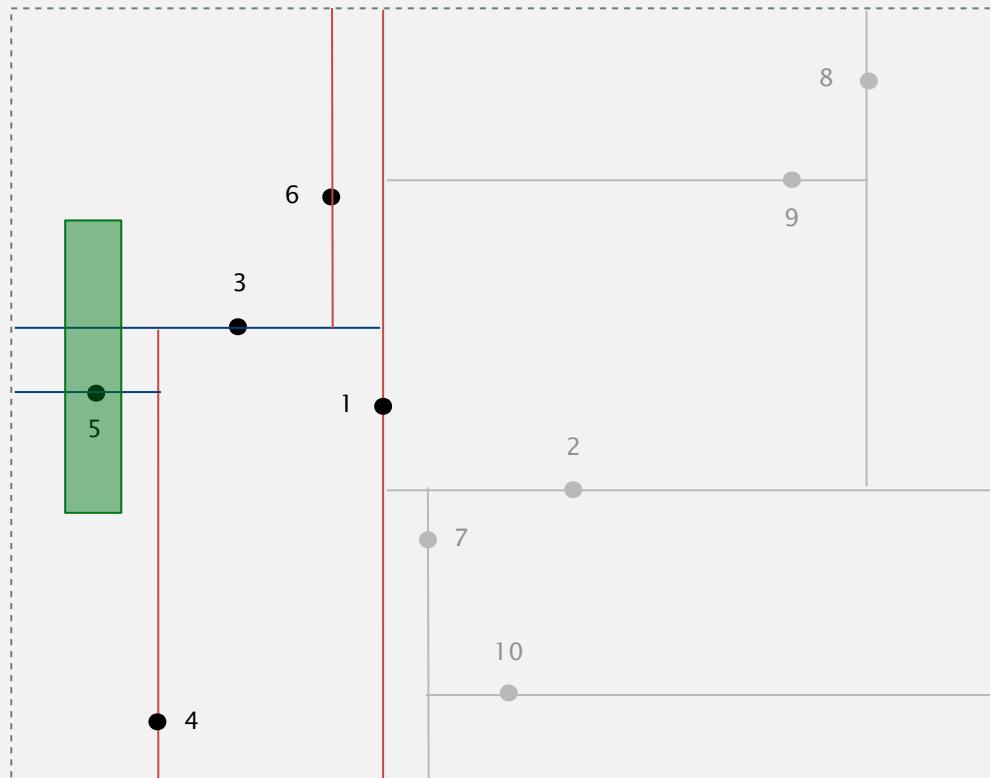
- Check if point in node lies in given rectangle.
- Recursively search left/bottom (if any could fall in rectangle).
- Recursively search right/top (if any could fall in rectangle).



Range search in a 2d tree analysis

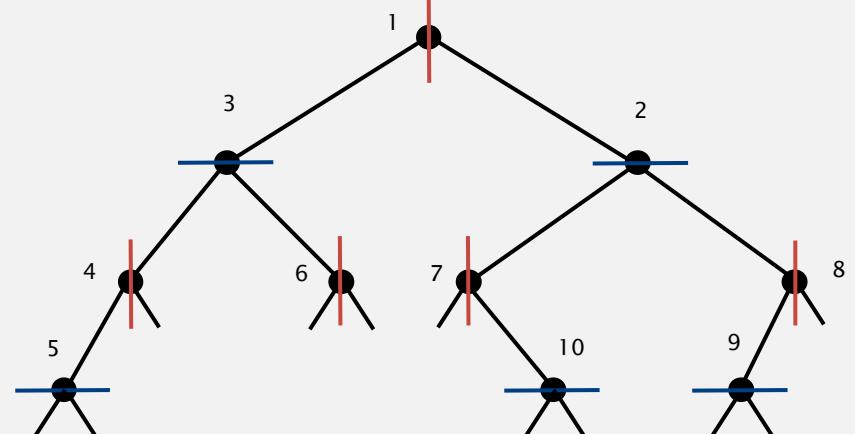
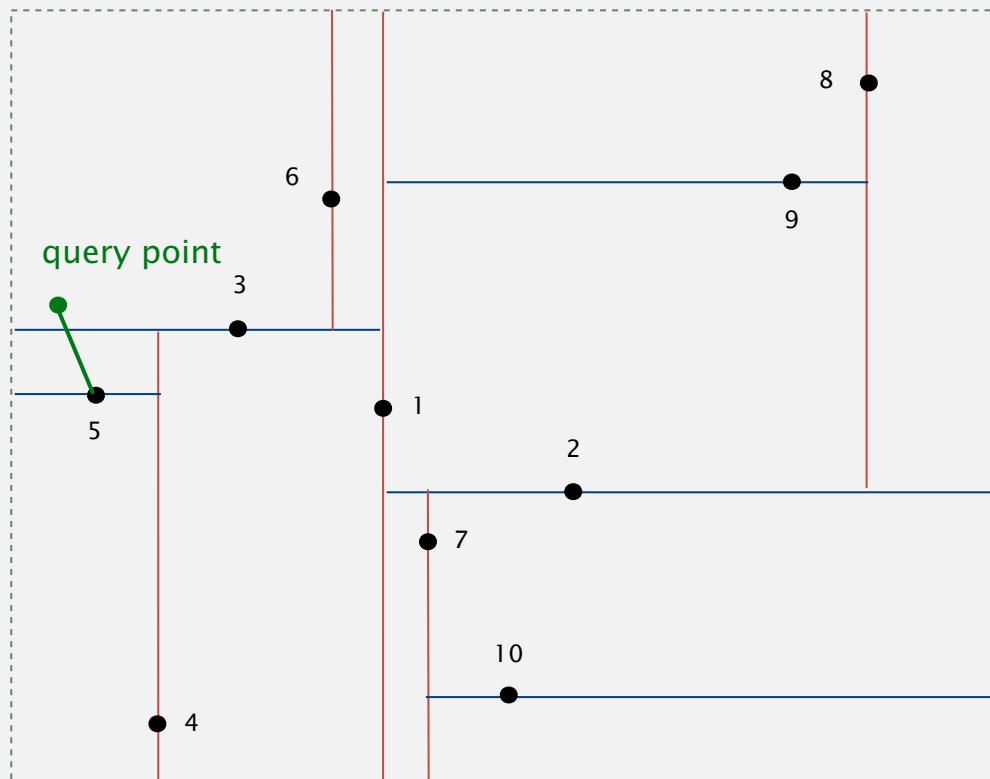
Typical case. $R + \log N$.

Worst case (assuming tree is balanced). $R + \sqrt{N}$.



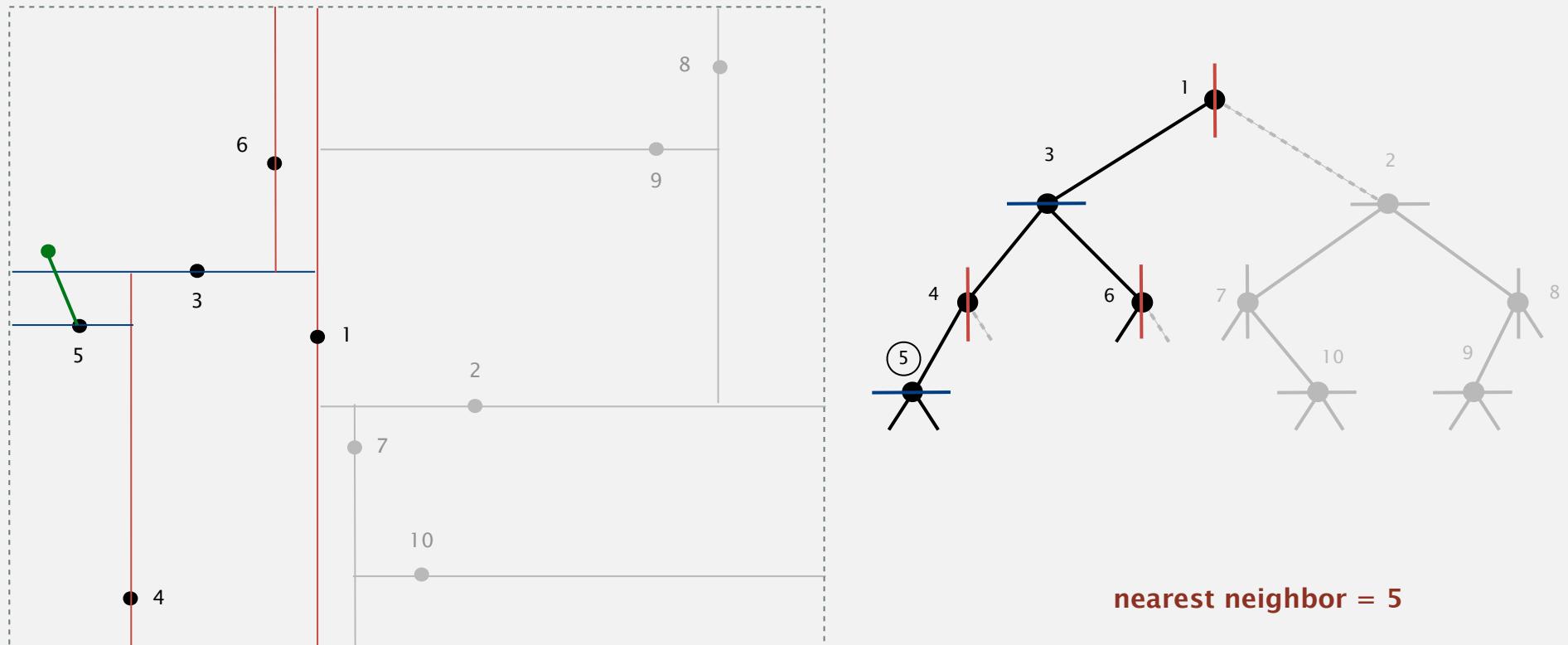
Nearest neighbor search in a 2d tree demo

Goal. Find closest point to query point.



Nearest neighbor search in a 2d tree demo

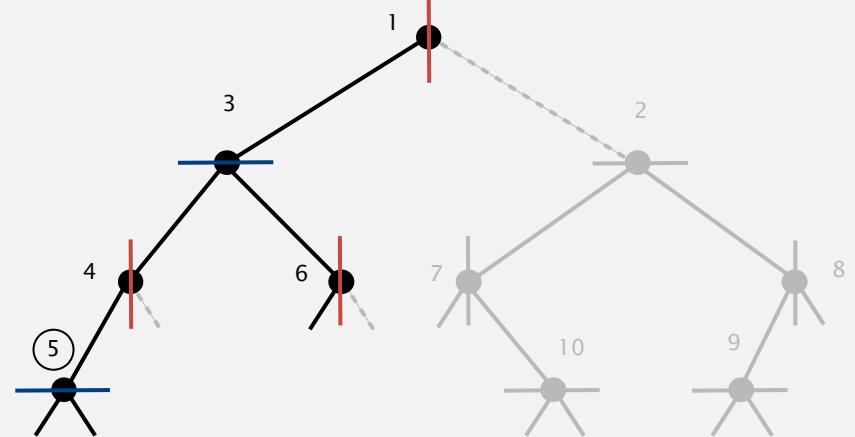
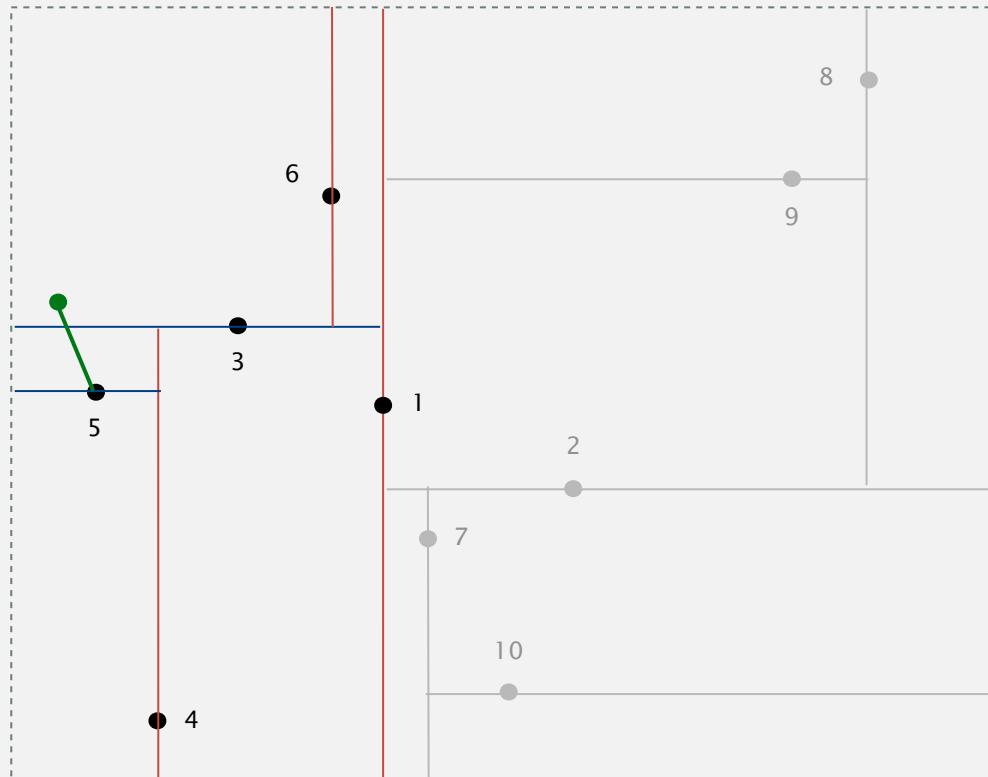
- Check distance from point in node to query point.
- Recursively search left/bottom (if it could contain a closer point).
- Recursively search right/top (if it could contain a closer point).
- Organize method so that it begins by searching for query point.



Nearest neighbor search in a 2d tree analysis

Typical case. $\log N$.

Worst case (even if tree is balanced). N .



nearest neighbor = 5

Flocking birds

Q. What "natural algorithm" do starlings, migrating geese, starlings, cranes, bait balls of fish, and flashing fireflies use to flock?

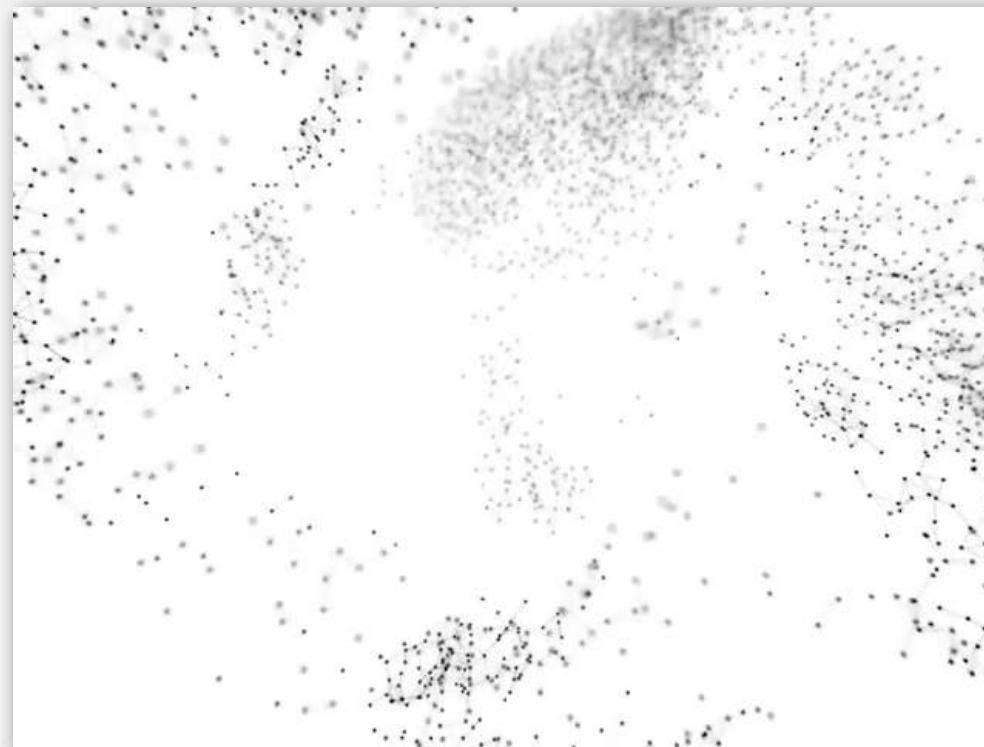


<http://www.youtube.com/watch?v=XH-groCeKbE>

Flocking boids [Craig Reynolds, 1986]

Boids. Three simple rules lead to complex emergent flocking behavior:

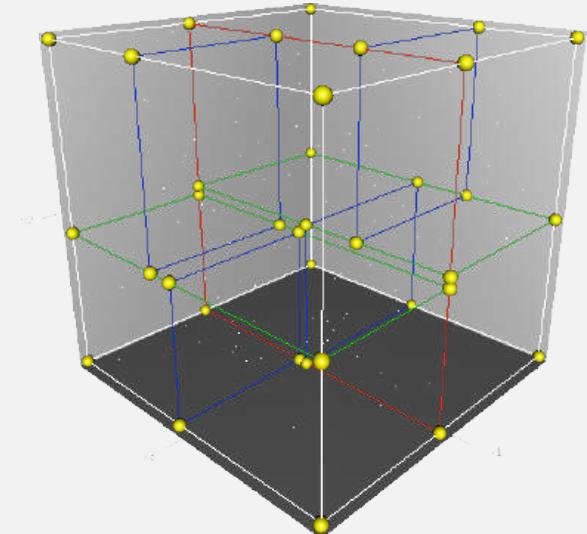
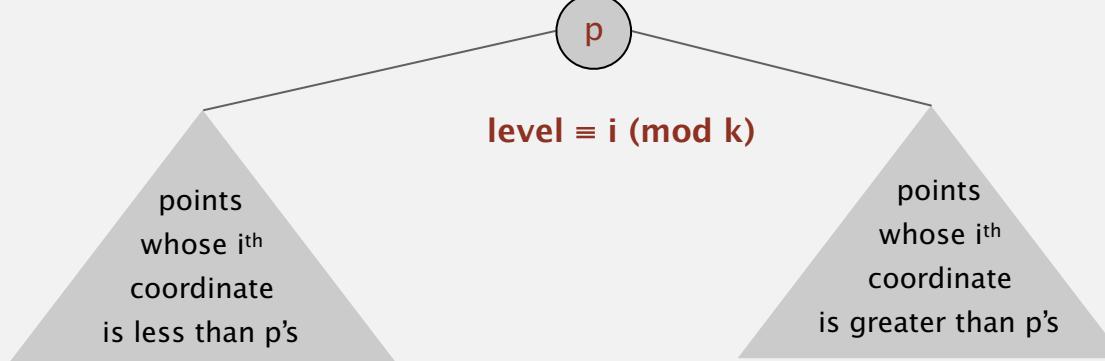
- Collision avoidance: point away from **k nearest** boids.
- Flock centering: point towards the center of mass of **k nearest** boids.
- Velocity matching: update velocity to the average of **k nearest** boids.



Kd tree

Kd tree. Recursively partition k -dimensional space into 2 halfspaces.

Implementation. BST, but cycle through dimensions ala 2d trees.



Efficient, simple data structure for processing k -dimensional data.

- Widely used.
- Adapts well to high-dimensional and clustered data.
- Discovered by an undergrad in an algorithms class!



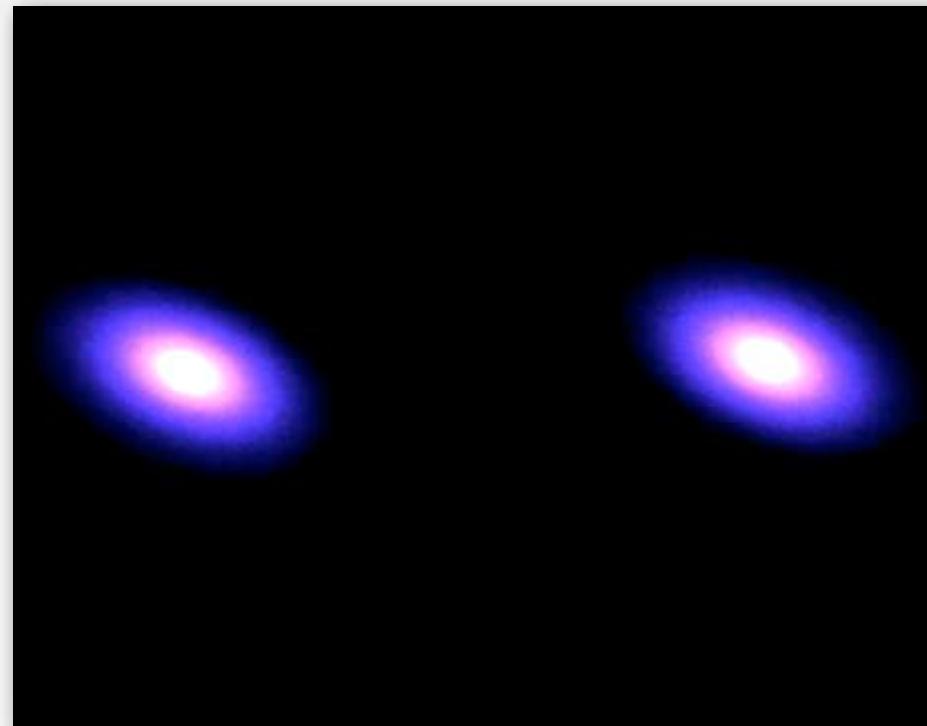
Jon Bentley

N-body simulation

Goal. Simulate the motion of N particles, mutually affected by gravity.

Brute force. For each pair of particles, compute force: $F = \frac{G m_1 m_2}{r^2}$

Running time. Time per step is N^2 .



http://www.youtube.com/watch?v=ua7YlN4eL_w

Appel's algorithm for N-body simulation

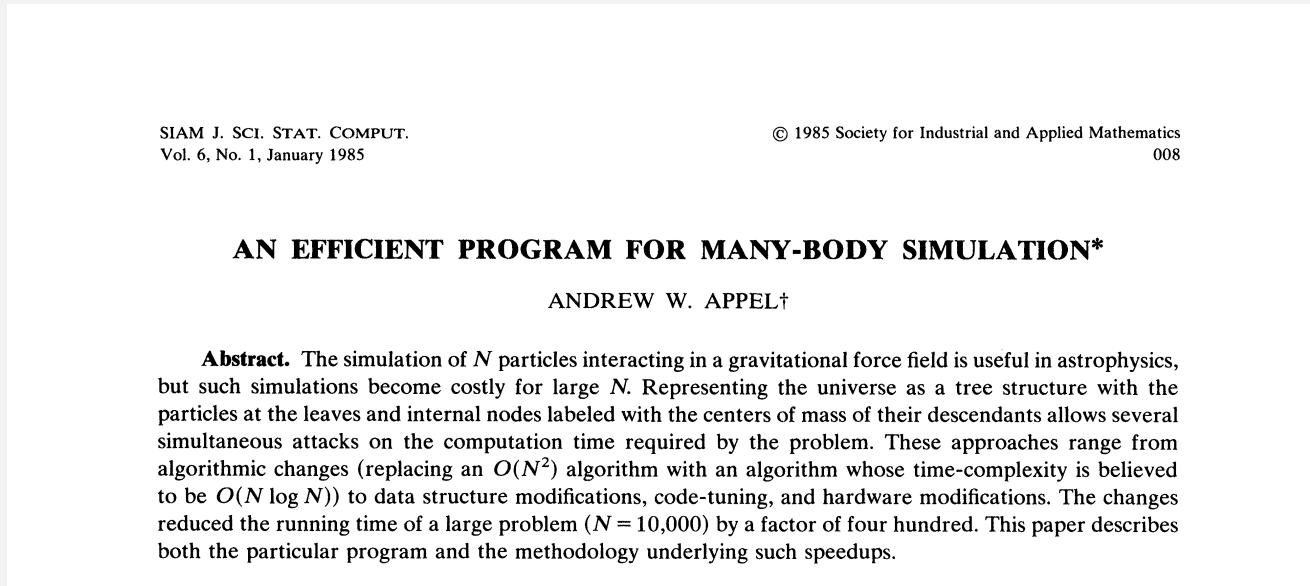
Key idea. Suppose particle is far, far away from cluster of particles.

- Treat cluster of particles as a single aggregate particle.
- Compute force between particle and **center of mass** of aggregate.



Appel's algorithm for N-body simulation

- Build 3d-tree with N particles as nodes.
- Store center-of-mass of subtree in each node.
- To compute total force acting on a particle, traverse tree, but stop as soon as distance from particle to subdivision is sufficiently large.



Impact. Running time per step is $N \log N \Rightarrow$ enables new research.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

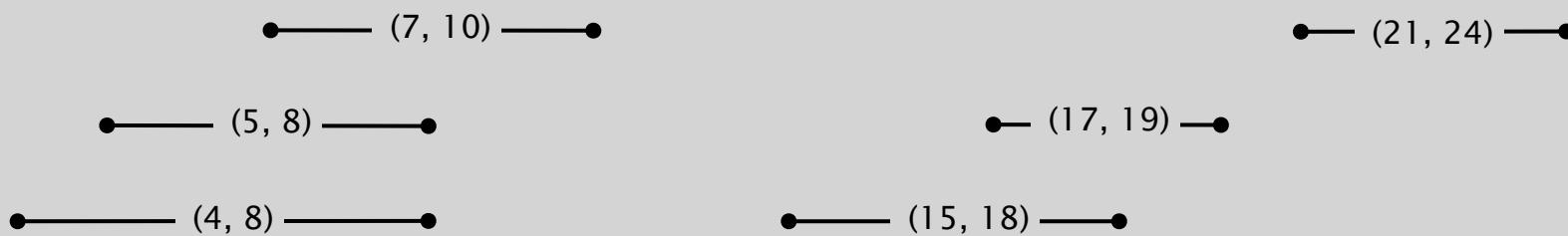
1d interval search

1d interval search. Data structure to hold set of (overlapping) intervals.

- Insert an interval (lo, hi).
- Search for an interval (lo, hi).
- Delete an interval (lo, hi).
- Interval intersection query: given an interval (lo, hi), find all intervals (or one interval) in data structure that intersects (lo, hi).

Q. Which intervals intersect (9, 16)?

A. (7, 10) and (15, 18).



1d interval search API

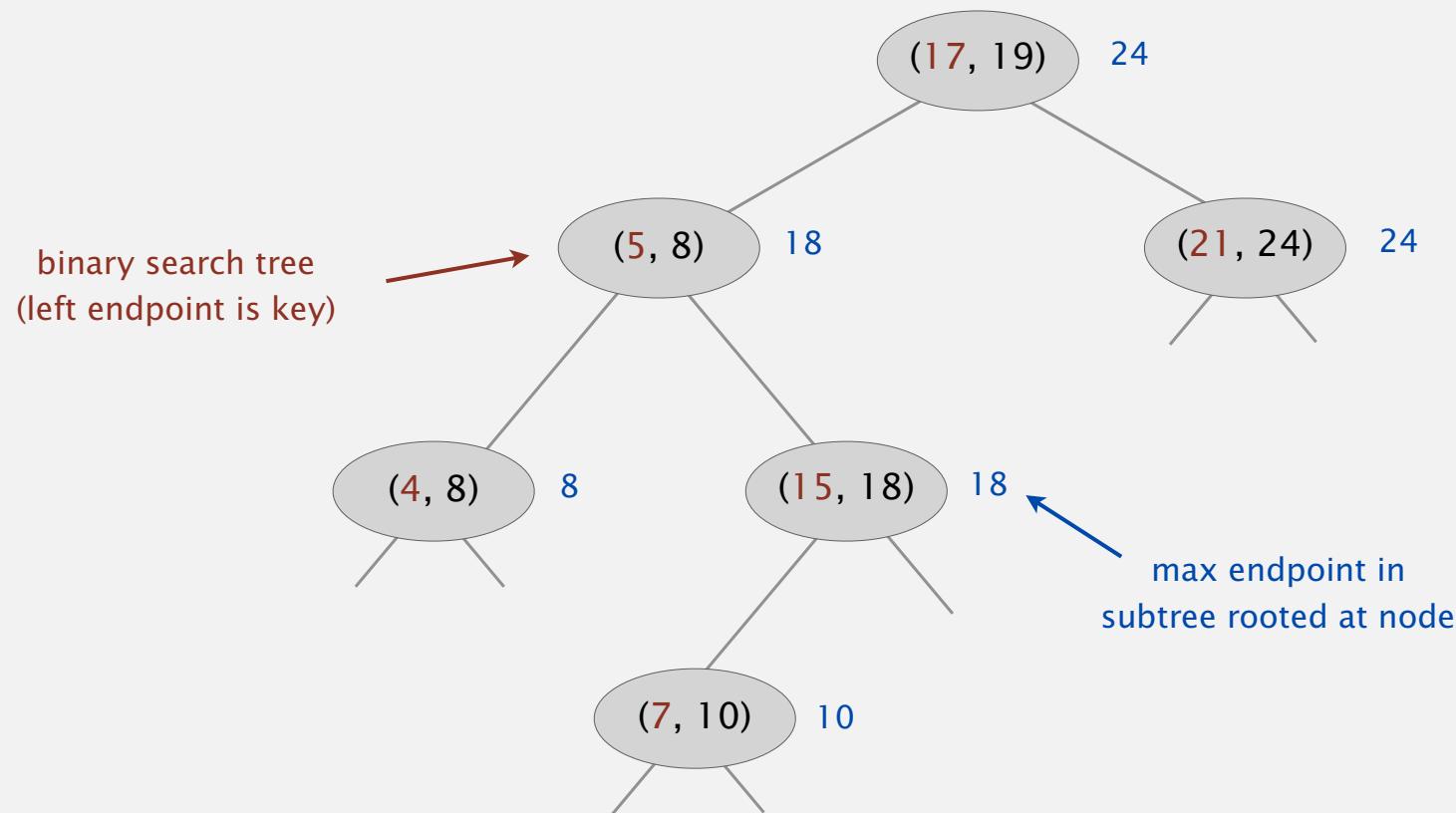
public class IntervalST<Key extends Comparable<Key>, Value>	
IntervalST()	<i>create interval search tree</i>
void put(Key lo, Key hi, Value val)	<i>put interval-value pair into ST</i>
Value get(Key lo, Key hi)	<i>value paired with given interval</i>
void delete(Key lo, Key hi)	<i>delete the given interval</i>
Iterable<Value> intersects(Key lo, Key hi)	<i>all intervals that intersect the given interval</i>

Nondegeneracy assumption. No two intervals have the same left endpoint.

Interval search trees

Create BST, where each node stores an interval (lo, hi).

- Use left endpoint as BST **key**.
- Store **max endpoint** in subtree rooted at node.



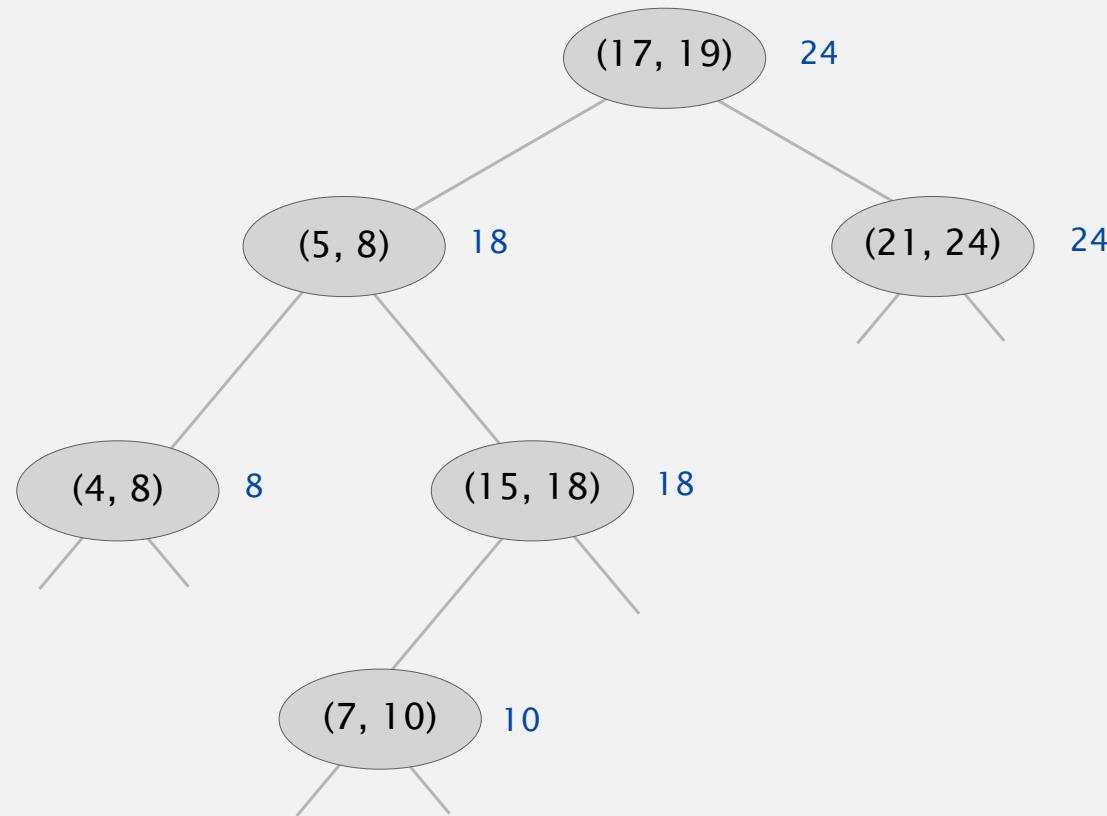
Interval search tree demo

To insert an interval (lo, hi) :

- Insert into BST, using lo as the key.
- Update max in each node on search path.



insert interval (16, 22)

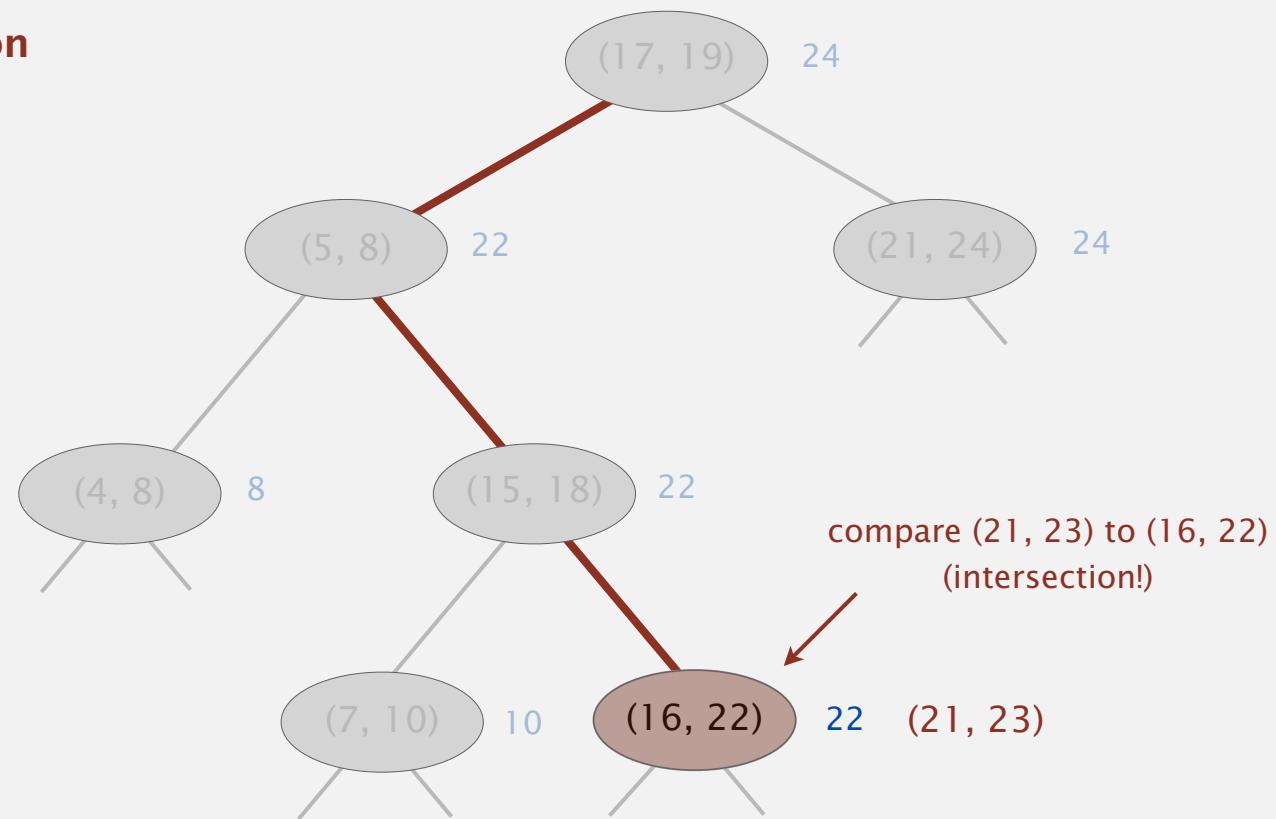


Interval search tree demo

To search for any one interval that intersects query interval (lo, hi):

- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than lo , go right.
- Else go left.

interval intersection
search for (21, 23)



Search for an intersecting interval implementation

To search for any one interval that intersects query interval (lo, hi):

- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than lo , go right.
- Else go left.

```
Node x = root;
while (x != null)
{
    if      (x.interval.intersects(lo, hi)) return x.interval;
    else if (x.left == null)                  x = x.right;
    else if (x.left.max < lo)                x = x.right;
    else                                      x = x.left;
}
return null;
```

Search for an intersecting interval analysis

To search for any one interval that intersects query interval (lo, hi):

- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than lo , go right.
- Else go left.

Case 1. If search goes **right**, then no intersection in left.

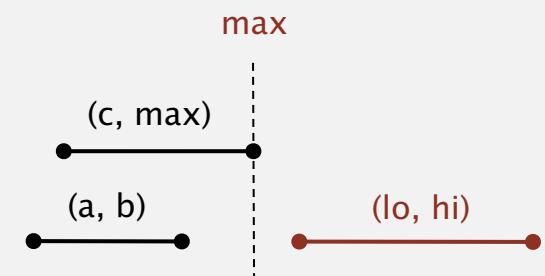
Pf. Suppose search goes right and left subtree is non empty.

- Max endpoint max in left subtree is less than lo .
- For any interval (a, b) in left subtree of x ,

we have $b \leq max < lo$.

definition of max

reason for going right



left subtree of x

right subtree of x

Search for an intersecting interval analysis

To search for any one interval that intersects query interval (lo, hi) :

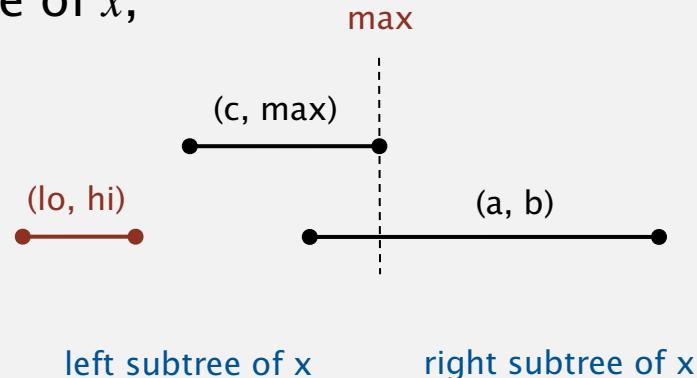
- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than lo , go right.
- Else go left.

Case 2. If search goes **left**, then there is either an intersection in left subtree or no intersections in either.

Pf. Suppose no intersection in left.

- Since went left, we have $lo \leq max$.
- Then for any interval (a, b) in right subtree of x ,
 $hi < c \leq a \Rightarrow$ no intersection in right.

no intersections in left subtree intervals sorted by left endpoint



Interval search tree: analysis

Implementation. Use a red-black BST to guarantee performance.

easy to maintain auxiliary information
using $\log N$ extra work per op

operation	brute	interval search tree	best in theory
insert interval	1	$\log N$	$\log N$
find interval	N	$\log N$	$\log N$
delete interval	N	$\log N$	$\log N$
find any one interval that intersects (lo, hi)	N	$\log N$	$\log N$
find all intervals that intersects (lo, hi)	N	$R \log N$	$R + \log N$

order of growth of running time for N intervals

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

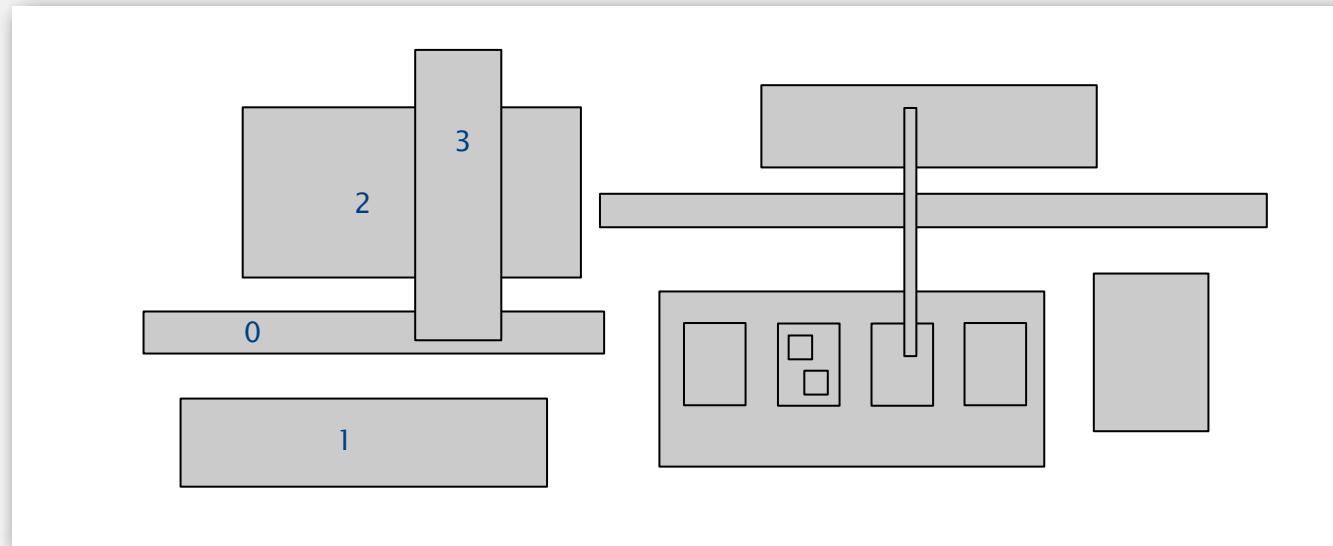
GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

Orthogonal rectangle intersection

Goal. Find all intersections among a set of N orthogonal rectangles.

Quadratic algorithm. Check all pairs of rectangles for intersection.



Non-degeneracy assumption. All x - and y -coordinates are distinct.

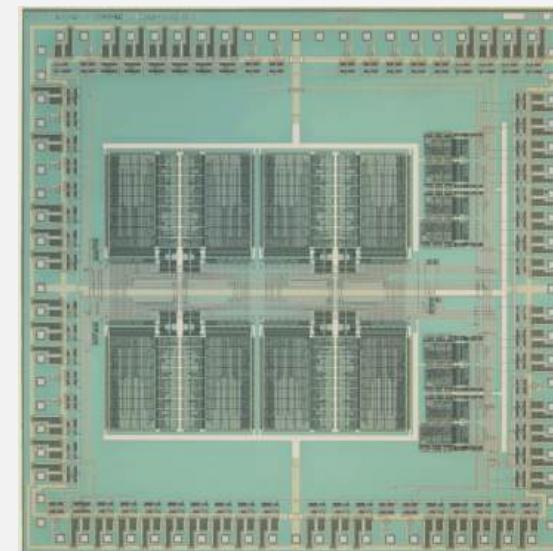
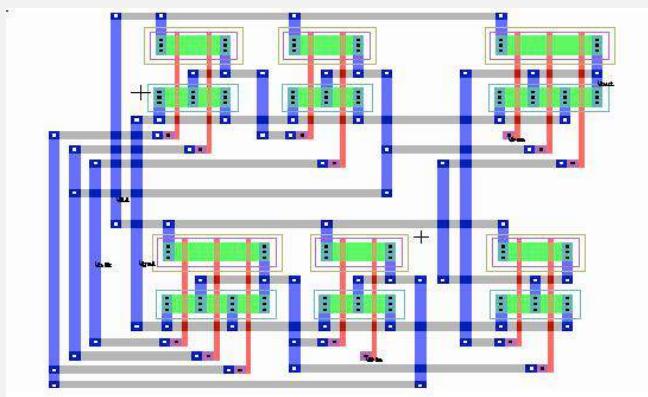
Microprocessors and geometry

Early 1970s. microprocessor design became a **geometric** problem.

- Very Large Scale Integration (VLSI).
- Computer-Aided Design (CAD).

Design-rule checking.

- Certain wires cannot intersect.
- Certain spacing needed between different types of wires.
- Debugging = orthogonal rectangle intersection search.



Algorithms and Moore's law

"Moore's law." Processing power doubles every 18 months.

- $197x$: check N rectangles.
- $197(x+1.5)$: check $2N$ rectangles on a $2x$ -faster computer.



Gordon Moore

Bootstrapping. We get to use the faster computer for bigger circuits.

But bootstrapping is not enough if using a quadratic algorithm:

- $197x$: takes M days.
- $197(x+1.5)$: takes $(4M)/2 = 2M$ days. (!)

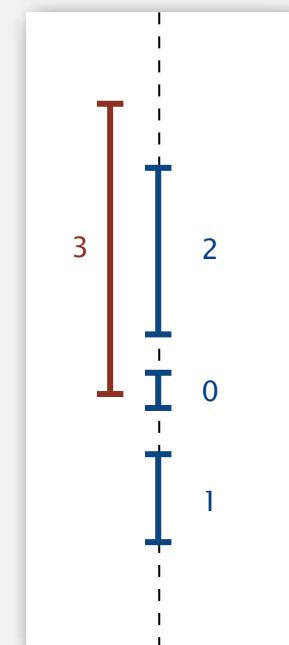
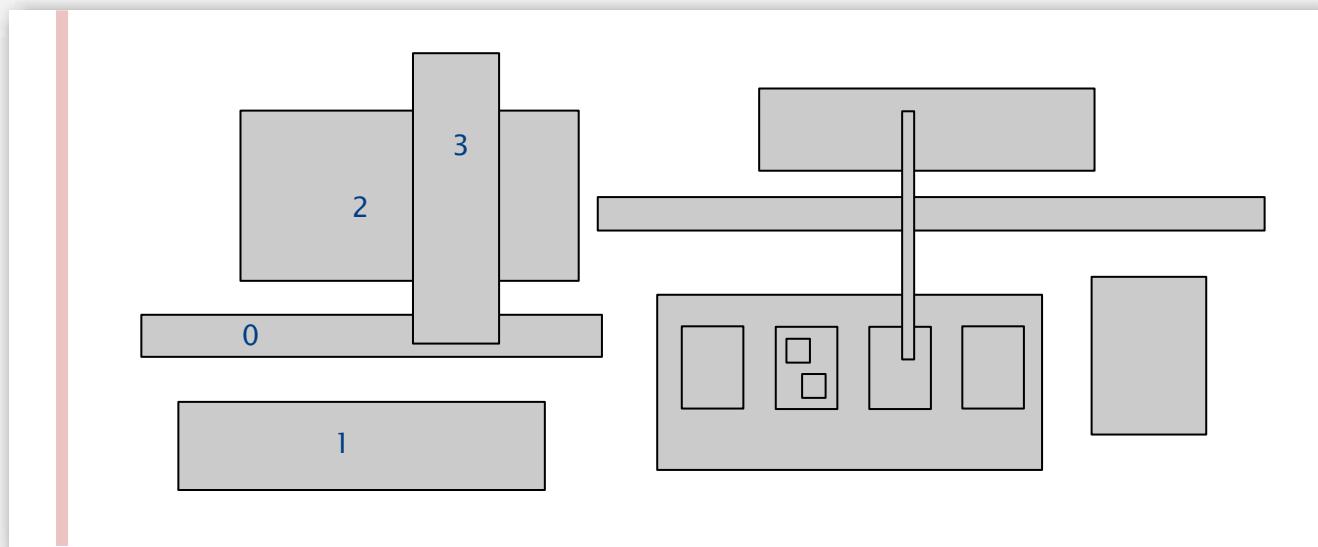


Bottom line. Linearithmic algorithm is **necessary** to sustain Moore's Law.

Orthogonal rectangle intersection: sweep-line algorithm

Sweep vertical line from left to right.

- x -coordinates of left and right endpoints define events.
- Maintain set of rectangles that intersect the sweep line in an interval search tree (using y -intervals of rectangle).
- Left endpoint: interval search for y -interval of rectangle; insert y -interval.
- Right endpoint: remove y -interval.



y-coordinates

Orthogonal rectangle intersection: sweep-line analysis

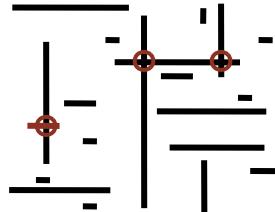
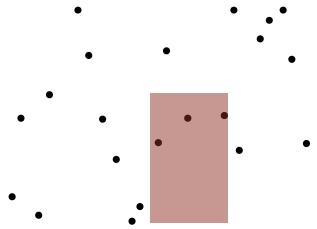
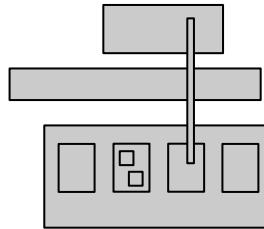
Proposition. Sweep line algorithm takes time proportional to $N \log N + R \log N$ to find R intersections among a set of N rectangles.

Pf.

- Put x -coordinates on a PQ (or sort). $\leftarrow N \log N$
- Insert y -intervals into ST. $\leftarrow N \log N$
- Delete y -intervals from ST. $\leftarrow N \log N$
- Interval searches for y -intervals. $\leftarrow N \log N + R \log N$

Bottom line. Sweep line reduces 2d orthogonal rectangle intersection search to 1d interval search.

Geometric applications of BSTs

problem	example	solution
1d range search	BST
2d orthogonal line segment intersection		sweep line reduces to 1d range search
kd range search		kd tree
1d interval search		interval search tree
2d orthogonal rectangle intersection		sweep line reduces to 1d interval search

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



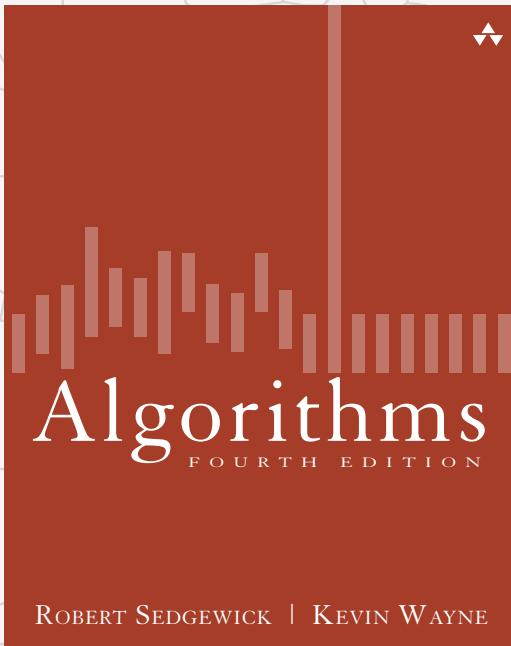
<http://algs4.cs.princeton.edu>

GEOMETRIC APPLICATIONS OF BSTs

- ▶ *1d range search*
- ▶ *line segment intersection*
- ▶ *kd trees*
- ▶ *interval search trees*
- ▶ *rectangle intersection*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ *Java implementation*
- ▶ *applications*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

6.4 MAXIMUM FLOW

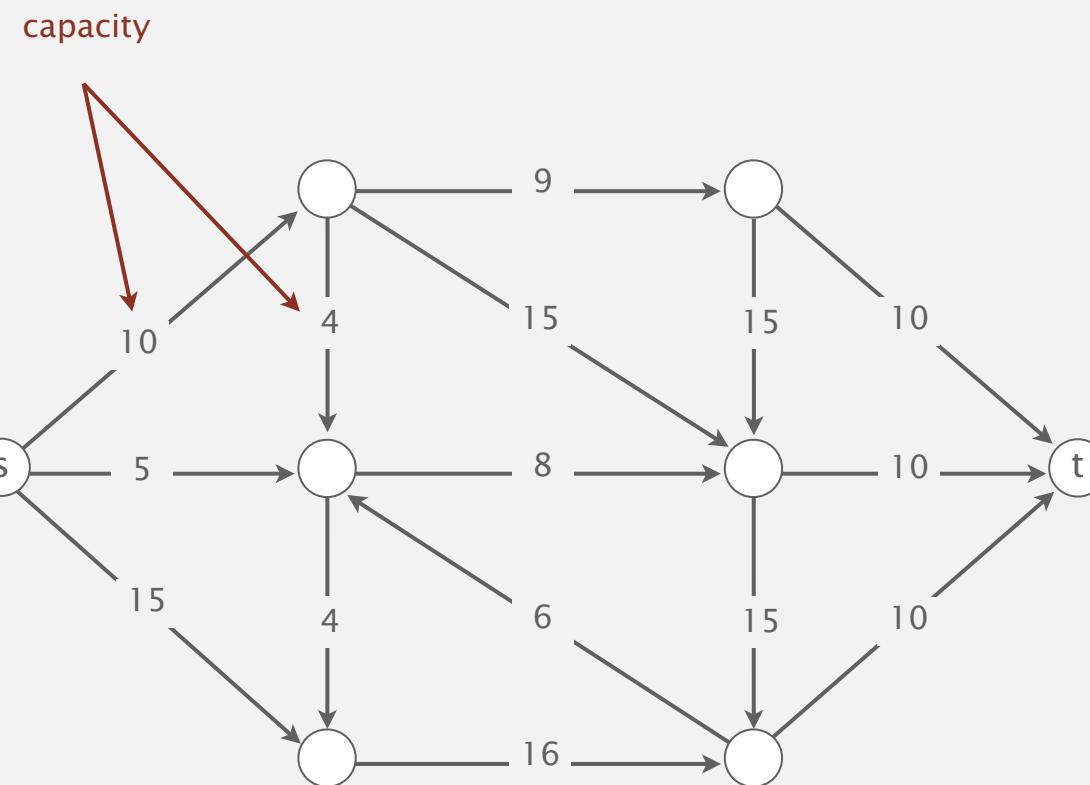
- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ *Java implementation*
- ▶ *applications*

Mincut problem

Input. An edge-weighted digraph, source vertex s , and target vertex t .



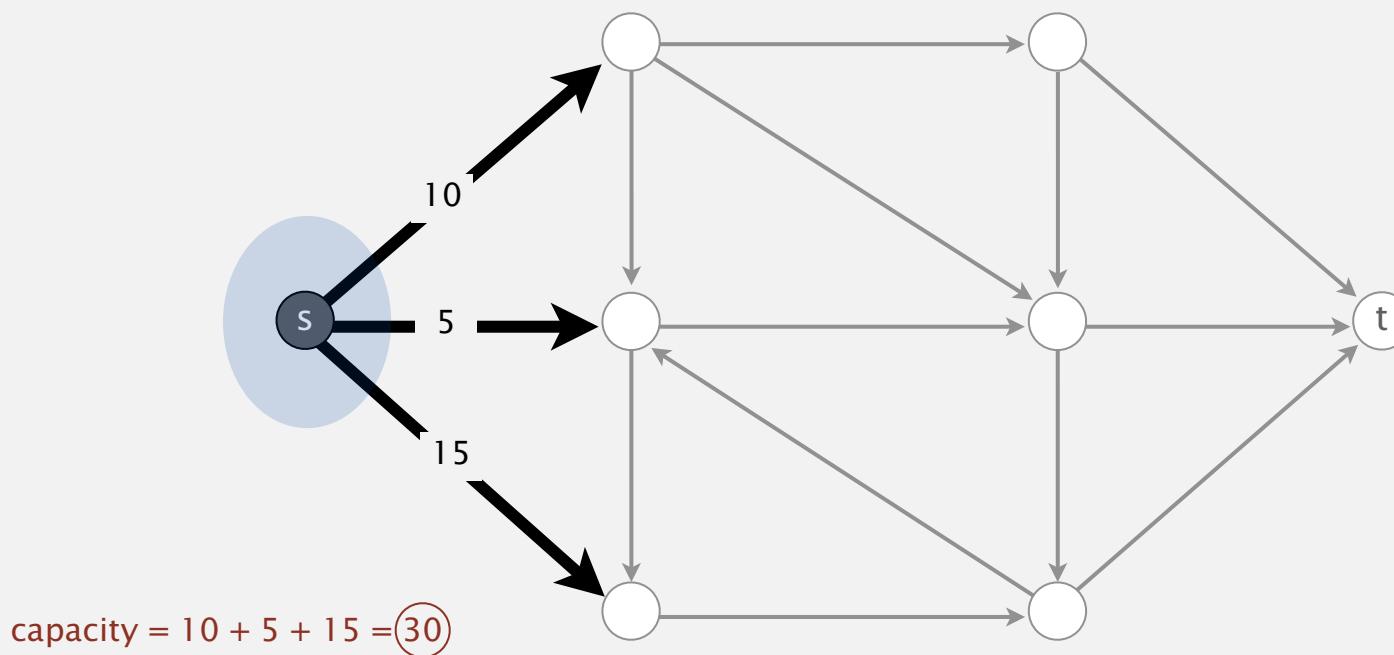
each edge has a
positive capacity



Mincut problem

Def. A *st-cut (cut)* is a partition of the vertices into two disjoint sets, with s in one set A and t in the other set B .

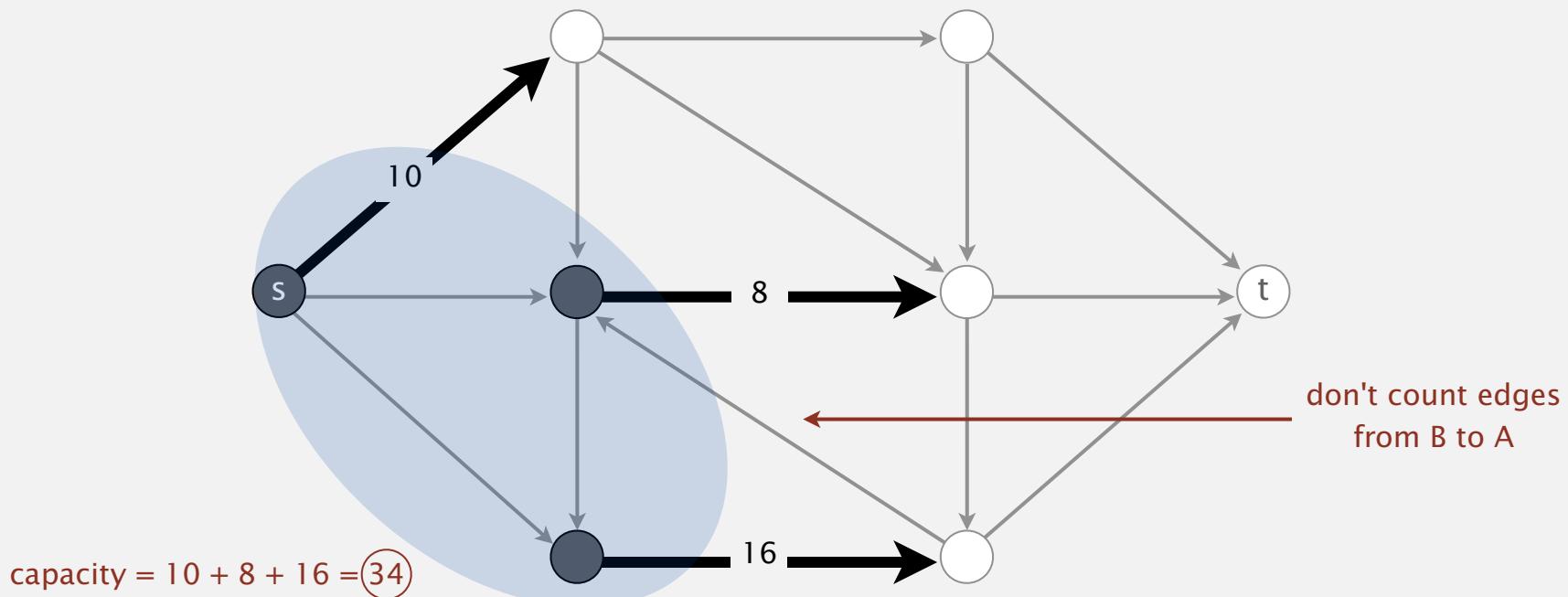
Def. Its **capacity** is the sum of the capacities of the edges from A to B .



Mincut problem

Def. A *st-cut* (cut) is a partition of the vertices into two disjoint sets, with s in one set A and t in the other set B .

Def. Its **capacity** is the sum of the capacities of the edges from A to B .

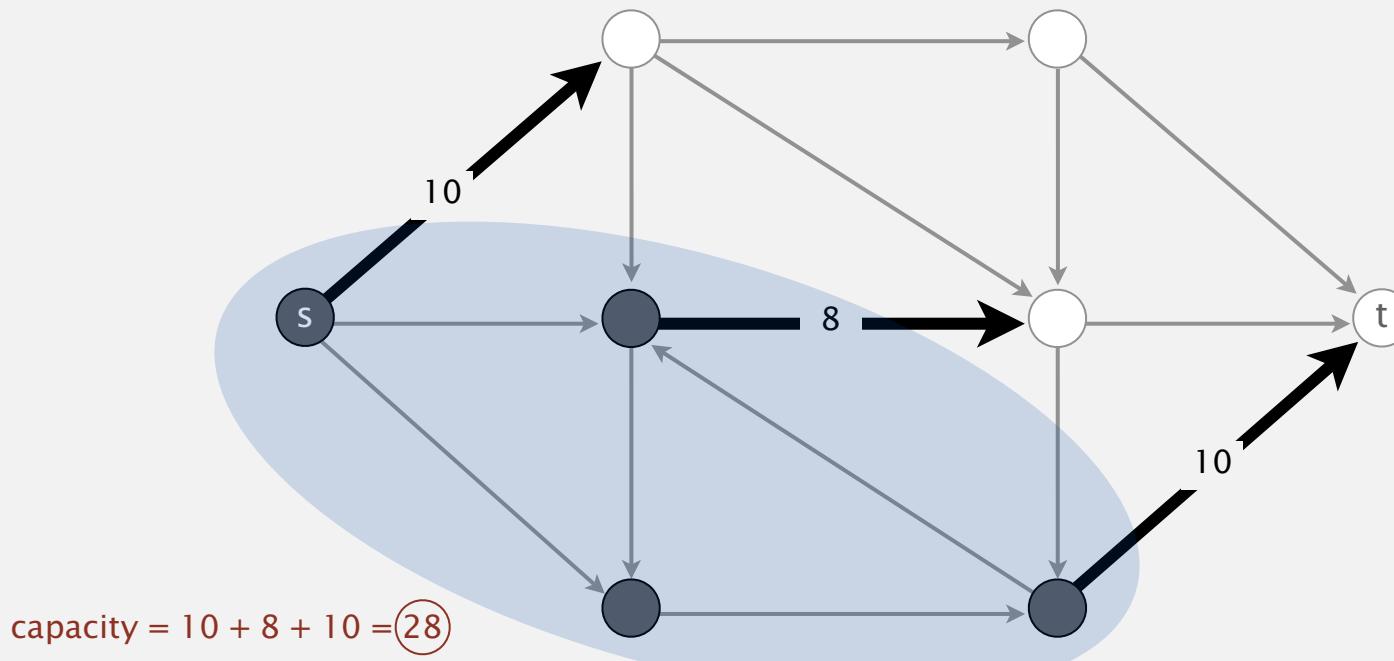


Mincut problem

Def. A *st-cut (cut)* is a partition of the vertices into two disjoint sets, with s in one set A and t in the other set B .

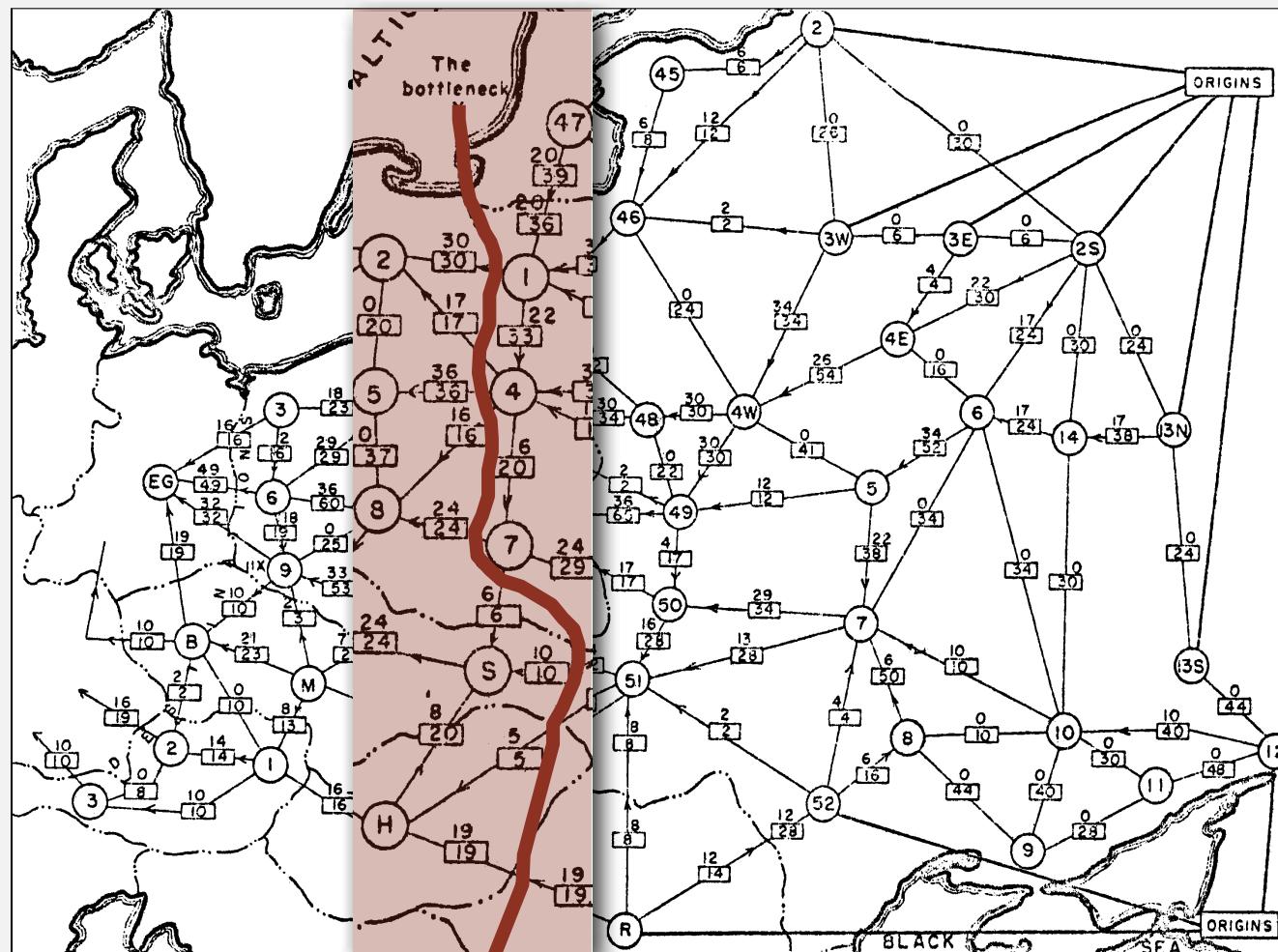
Def. Its **capacity** is the sum of the capacities of the edges from A to B .

Minimum st-cut (mincut) problem. Find a cut of minimum capacity.



Mincut application (1950s)

"Free world" goal. Cut supplies (if cold war turns into real war).



rail network connecting Soviet Union with Eastern European countries
(map declassified by Pentagon in 1999)

Potential mincut application (2010s)

Government-in-power's goal. Cut off communication to set of people.

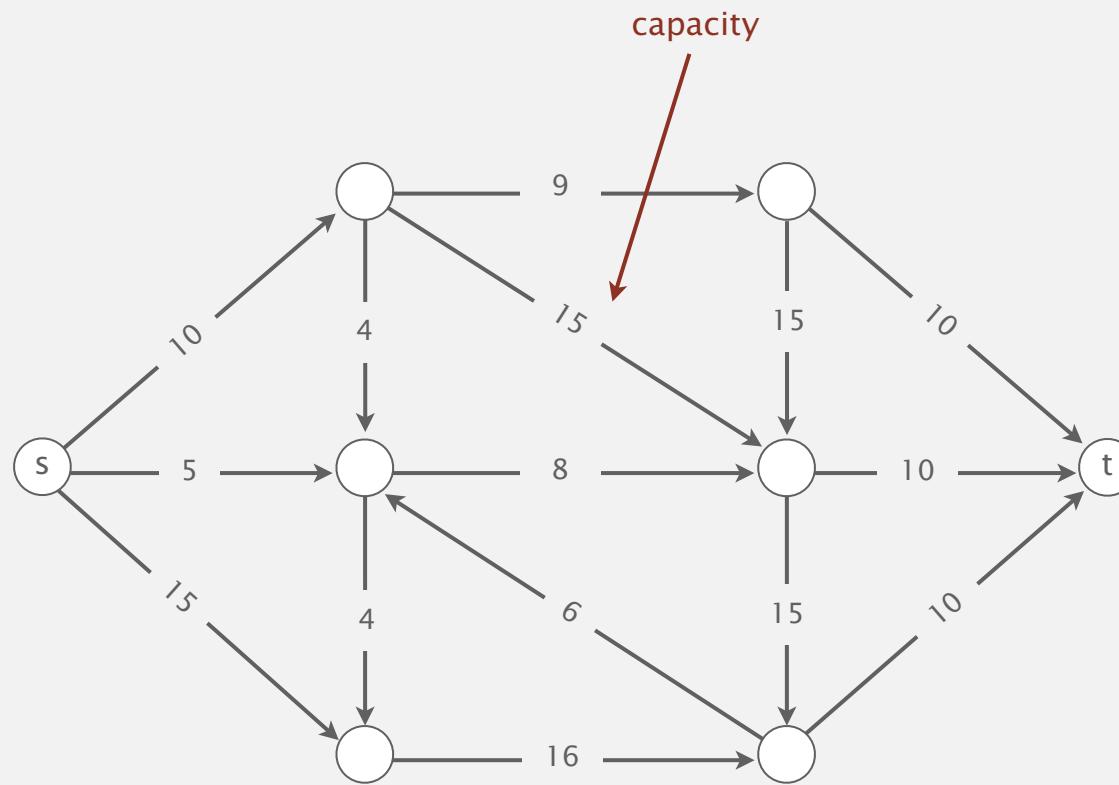


Maxflow problem

Input. An edge-weighted digraph, source vertex s , and target vertex t .



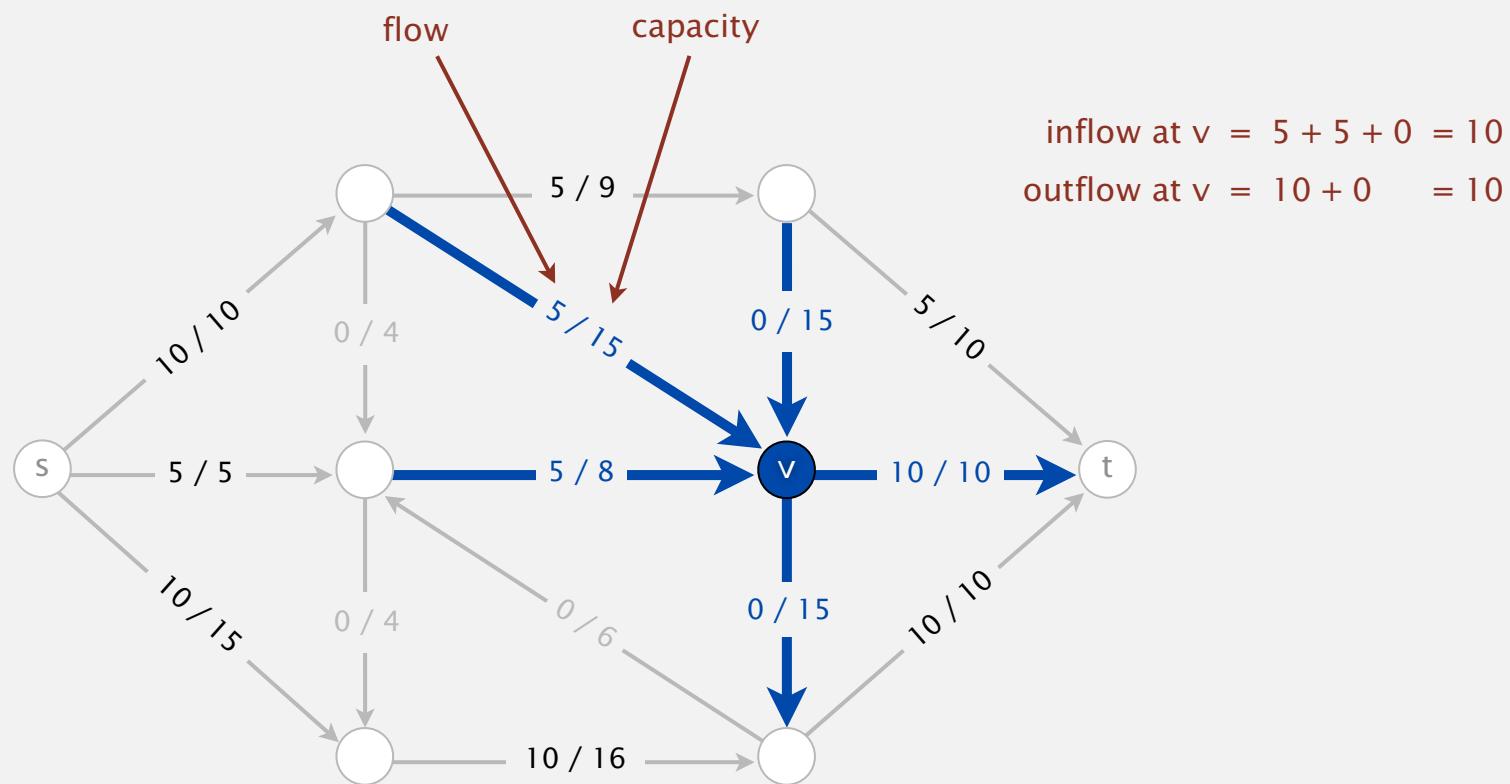
each edge has a
positive capacity



Maxflow problem

Def. An *st-flow (flow)* is an assignment of values to the edges such that:

- Capacity constraint: $0 \leq$ edge's flow \leq edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except s and t).



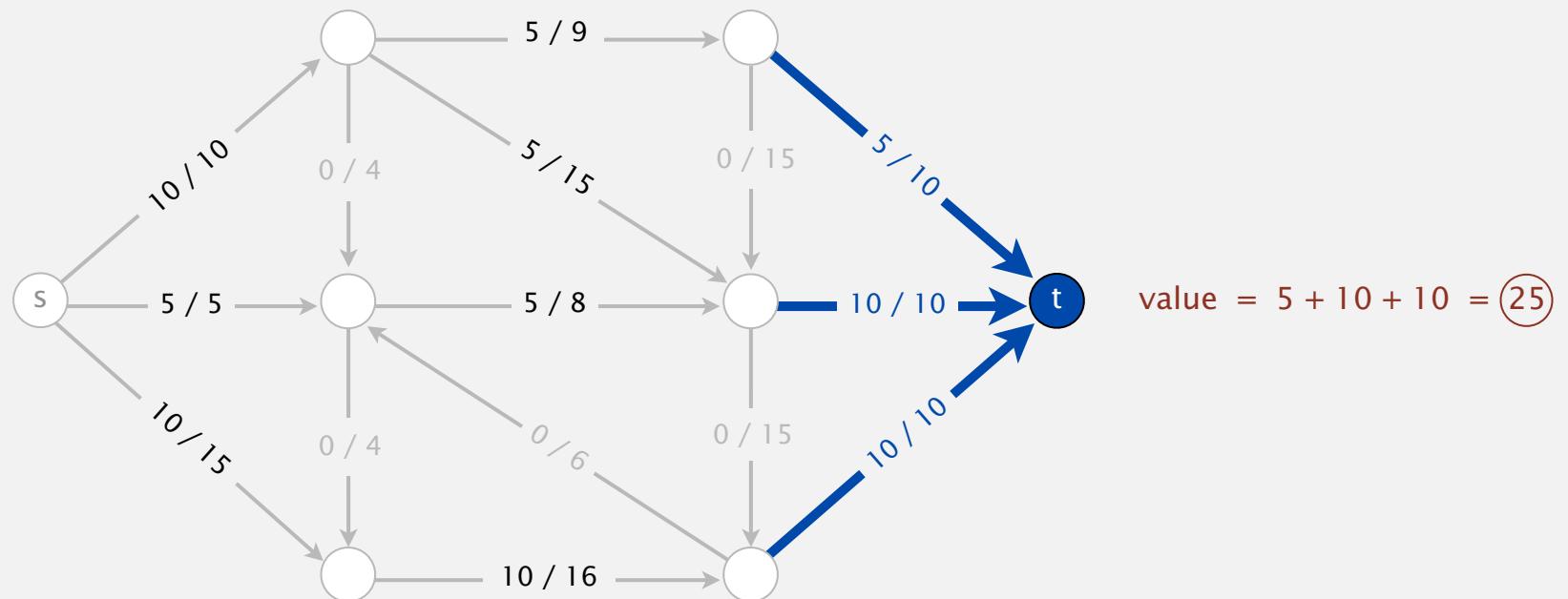
Maxflow problem

Def. An *st-flow (flow)* is an assignment of values to the edges such that:

- Capacity constraint: $0 \leq$ edge's flow \leq edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except s and t).

Def. The **value** of a flow is the inflow at t .

we assume no edge points to s or from t



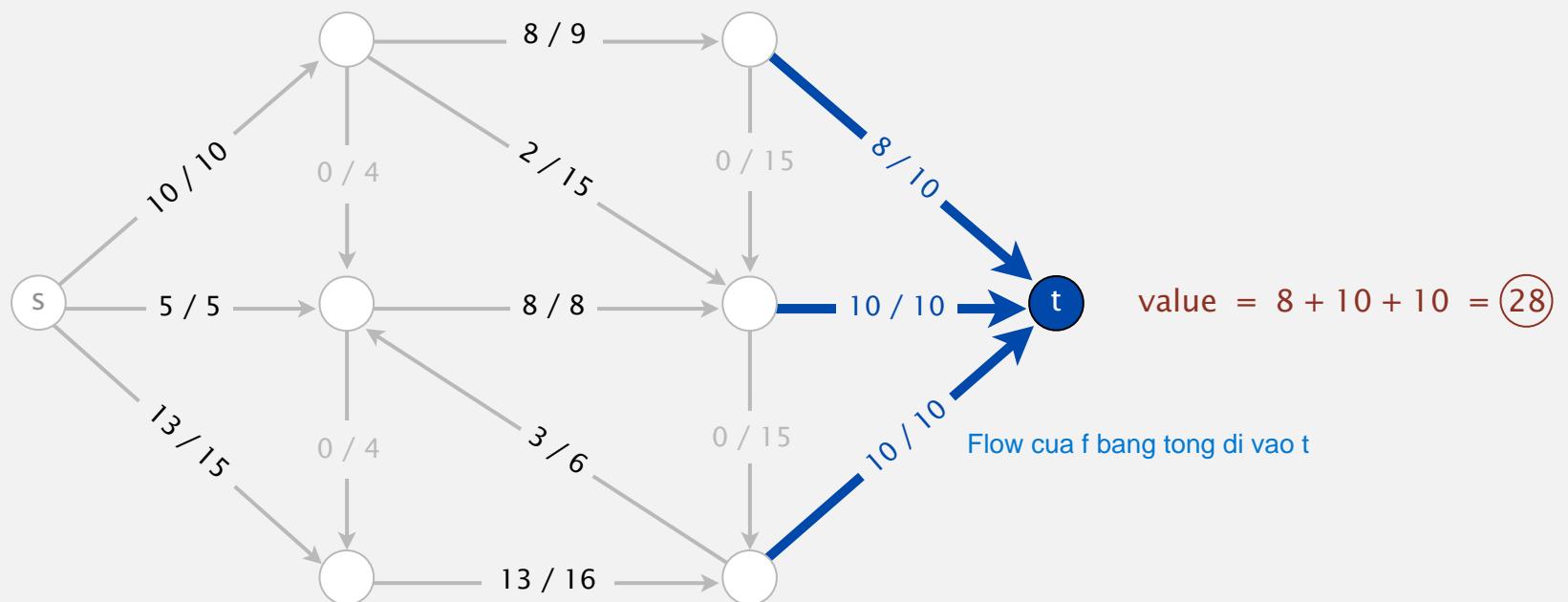
Maxflow problem

Def. An *st-flow (flow)* is an assignment of values to the edges such that:

- Capacity constraint: $0 \leq$ edge's flow \leq edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except s and t).

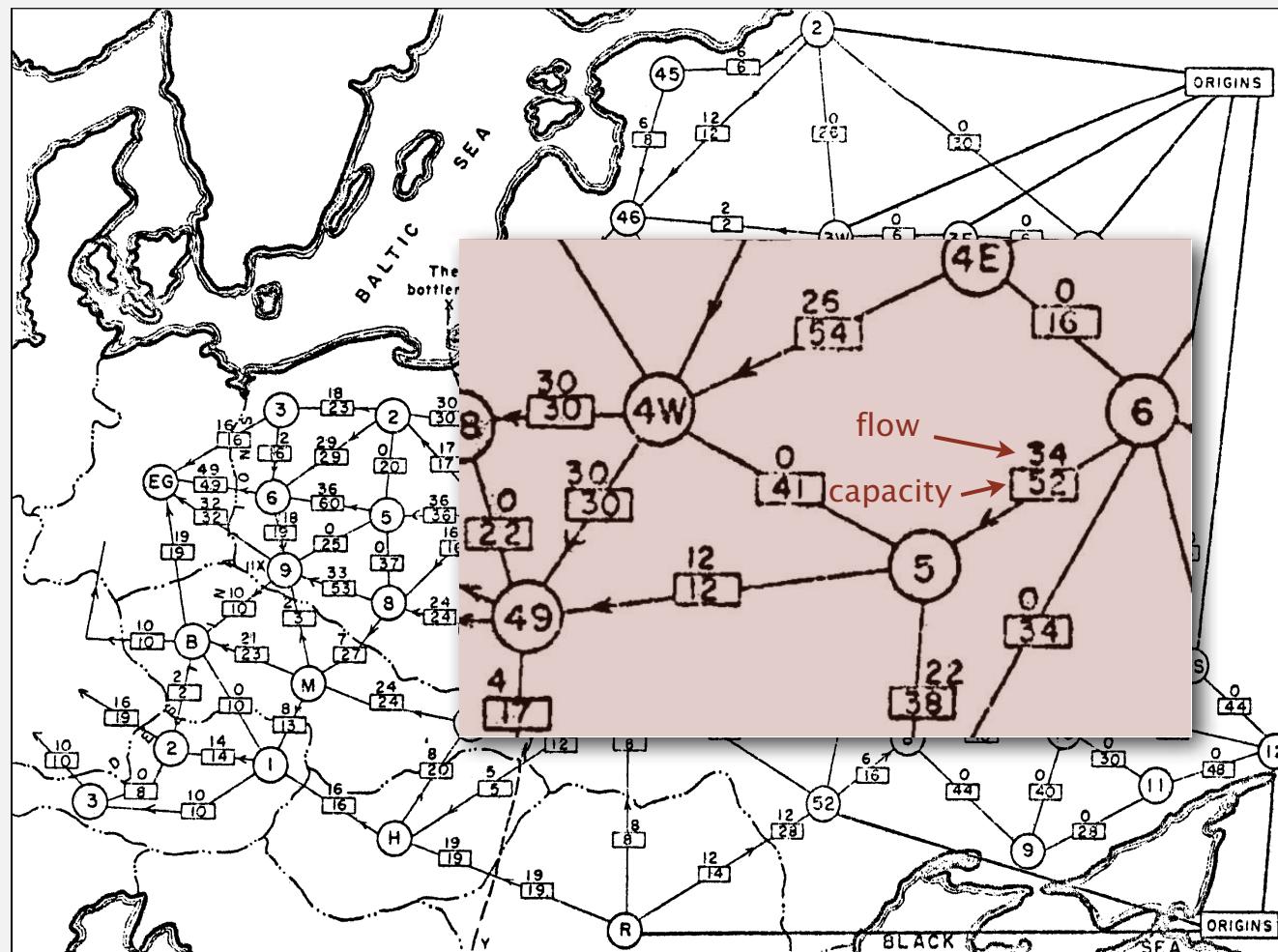
Def. The **value** of a flow is the inflow at t .

Maximum st-flow (maxflow) problem. Find a flow of maximum value.



Maxflow application (1950s)

Soviet Union goal. Maximize flow of supplies to Eastern Europe.



rail network connecting Soviet Union with Eastern European countries

(map declassified by Pentagon in 1999)

Potential maxflow application (2010s)

"Free world" goal. Maximize flow of information to specified set of people.



facebook graph

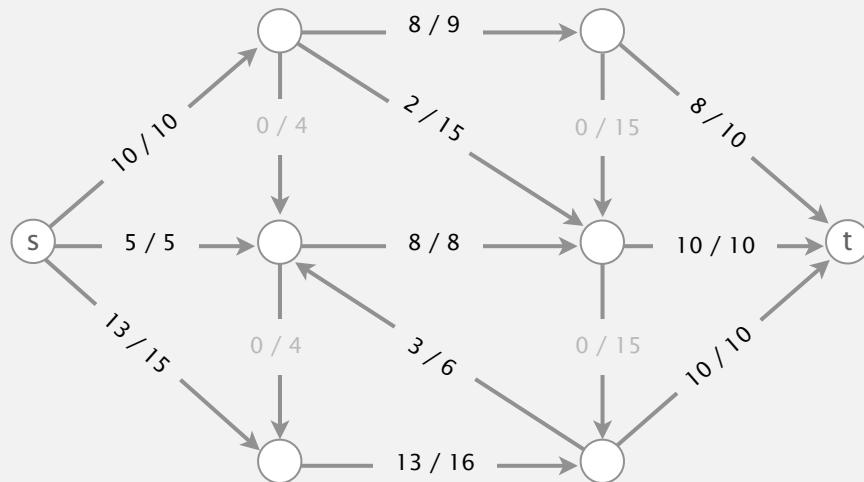
Summary

Input. A weighted digraph, source vertex s , and target vertex t .

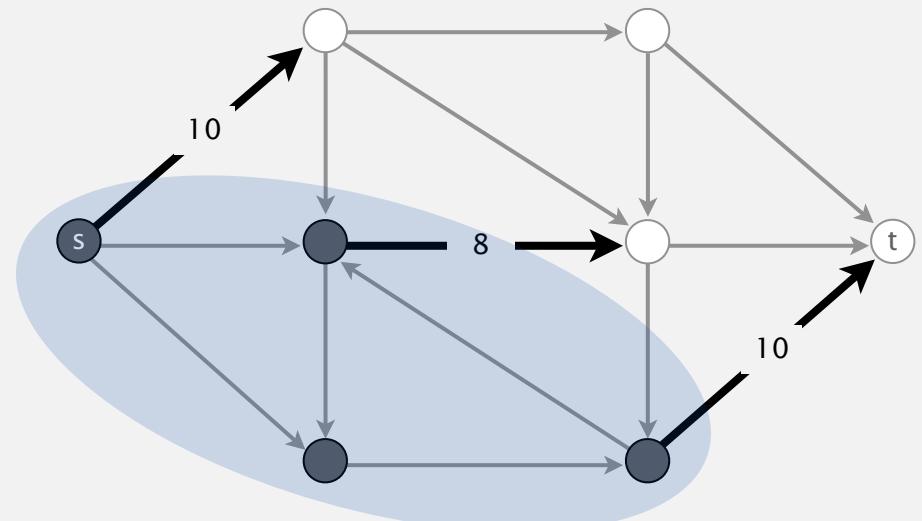
Mincut problem. Find a cut of minimum capacity.

Maxflow problem. Find a flow of maximum value.

Kinda same problems



value of flow = 28



capacity of cut = 28

Remarkable fact. These two problems are dual!

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ *Java implementation*
- ▶ *applications*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

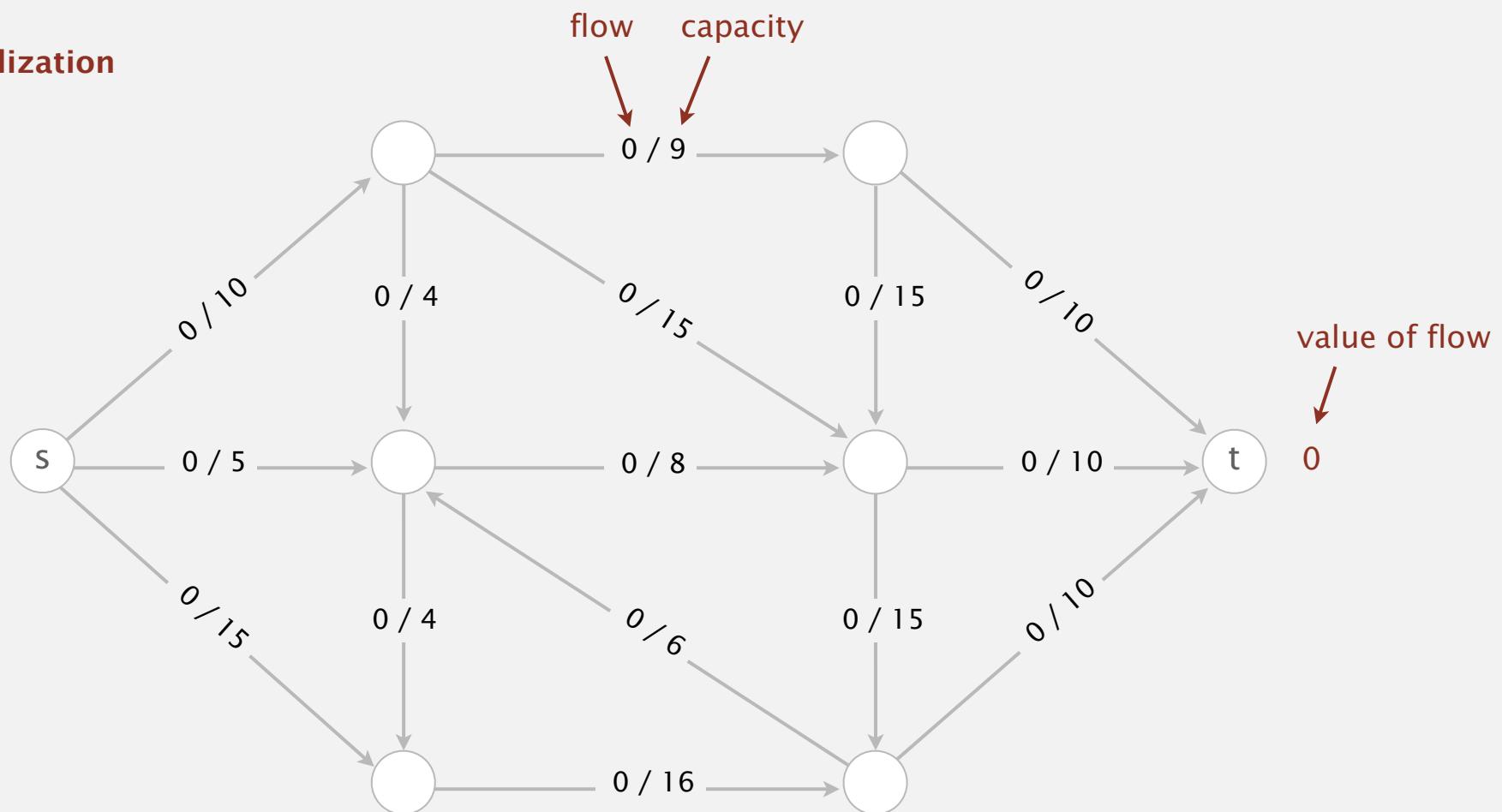
6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ ***Ford-Fulkerson algorithm***
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ *Java implementation*
- ▶ *applications*

Ford-Fulkerson algorithm

Initialization. Start with 0 flow.

initialization

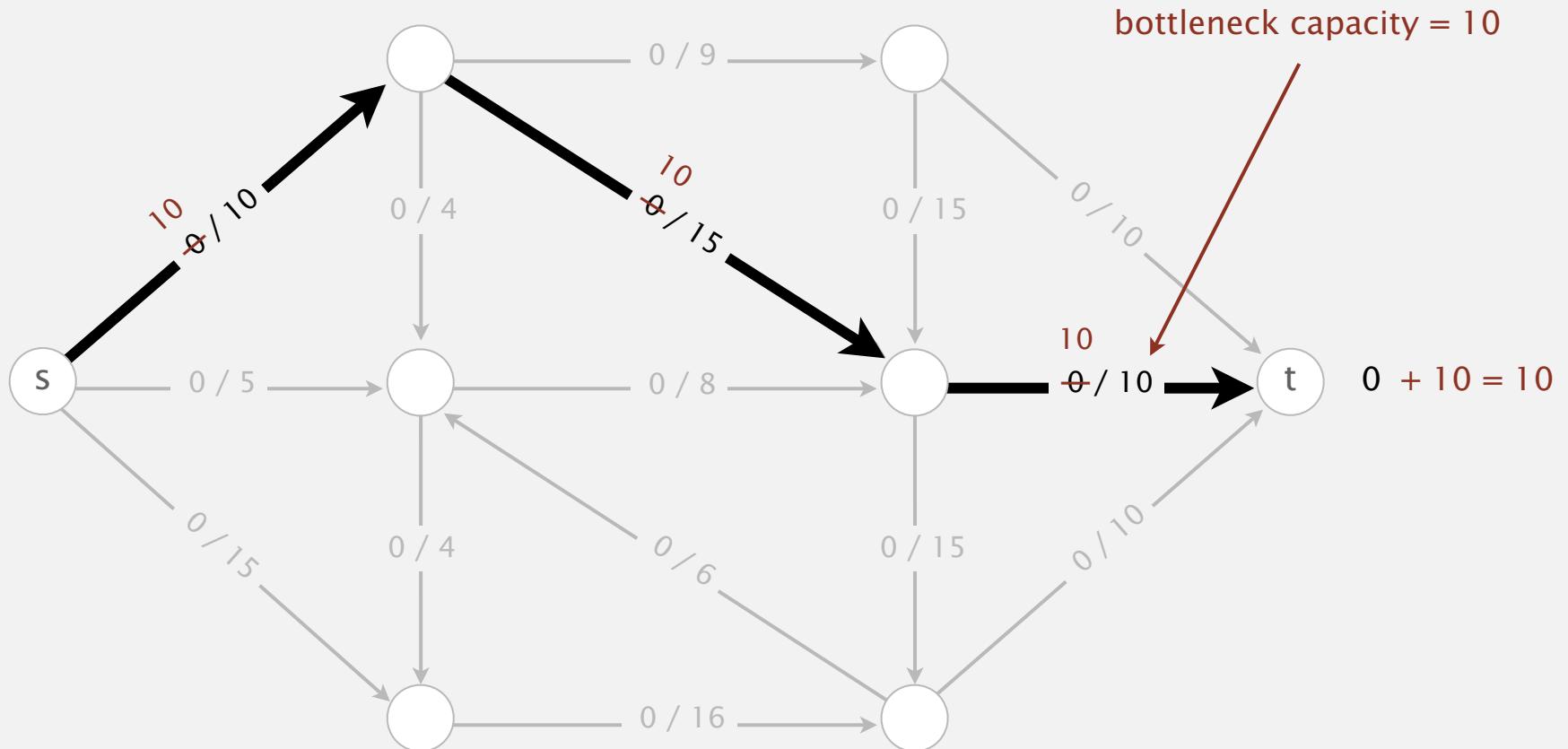


Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

1st augmenting path

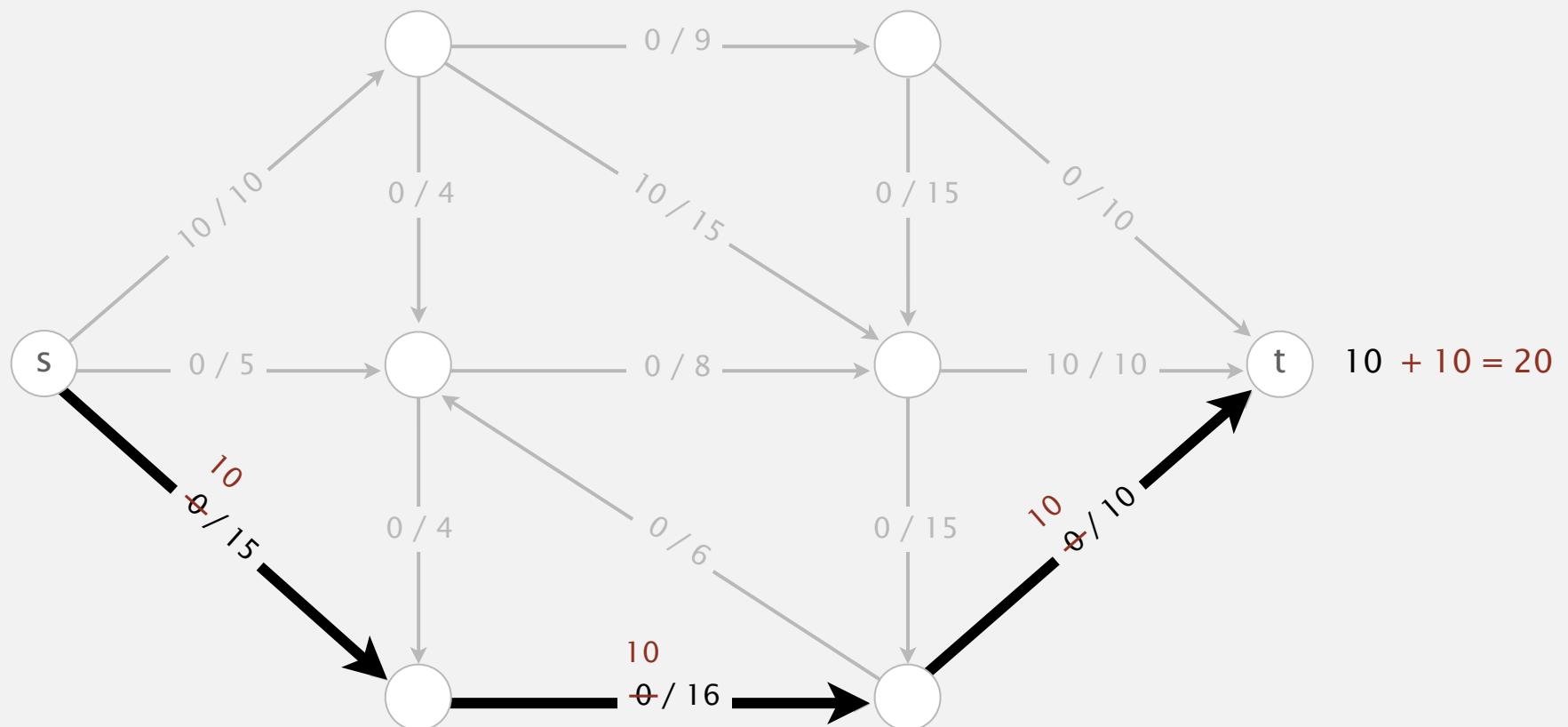


Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

2nd augmenting path

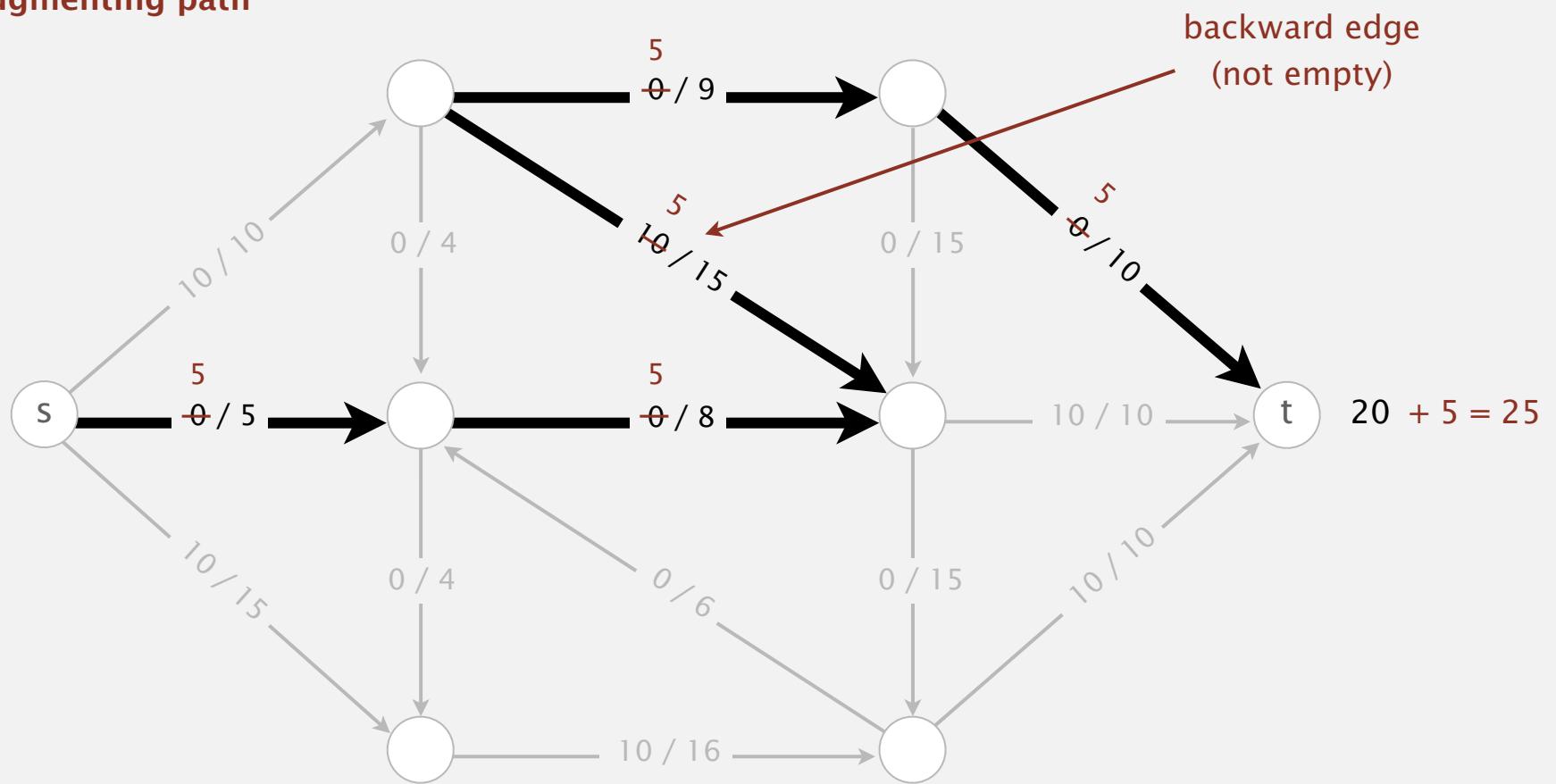


Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

3rd augmenting path

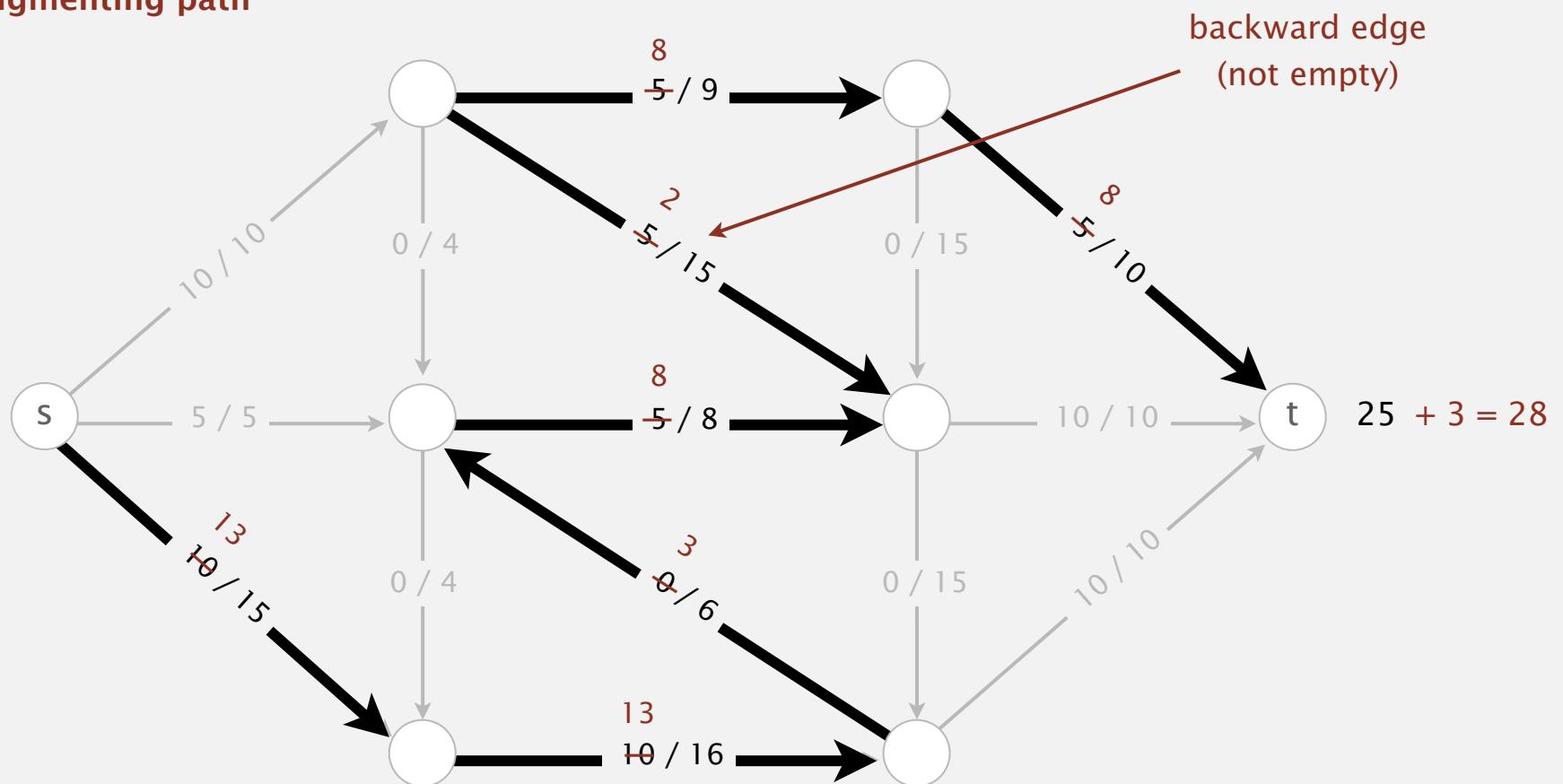


Idea: increase flow along augmenting paths

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

4th augmenting path

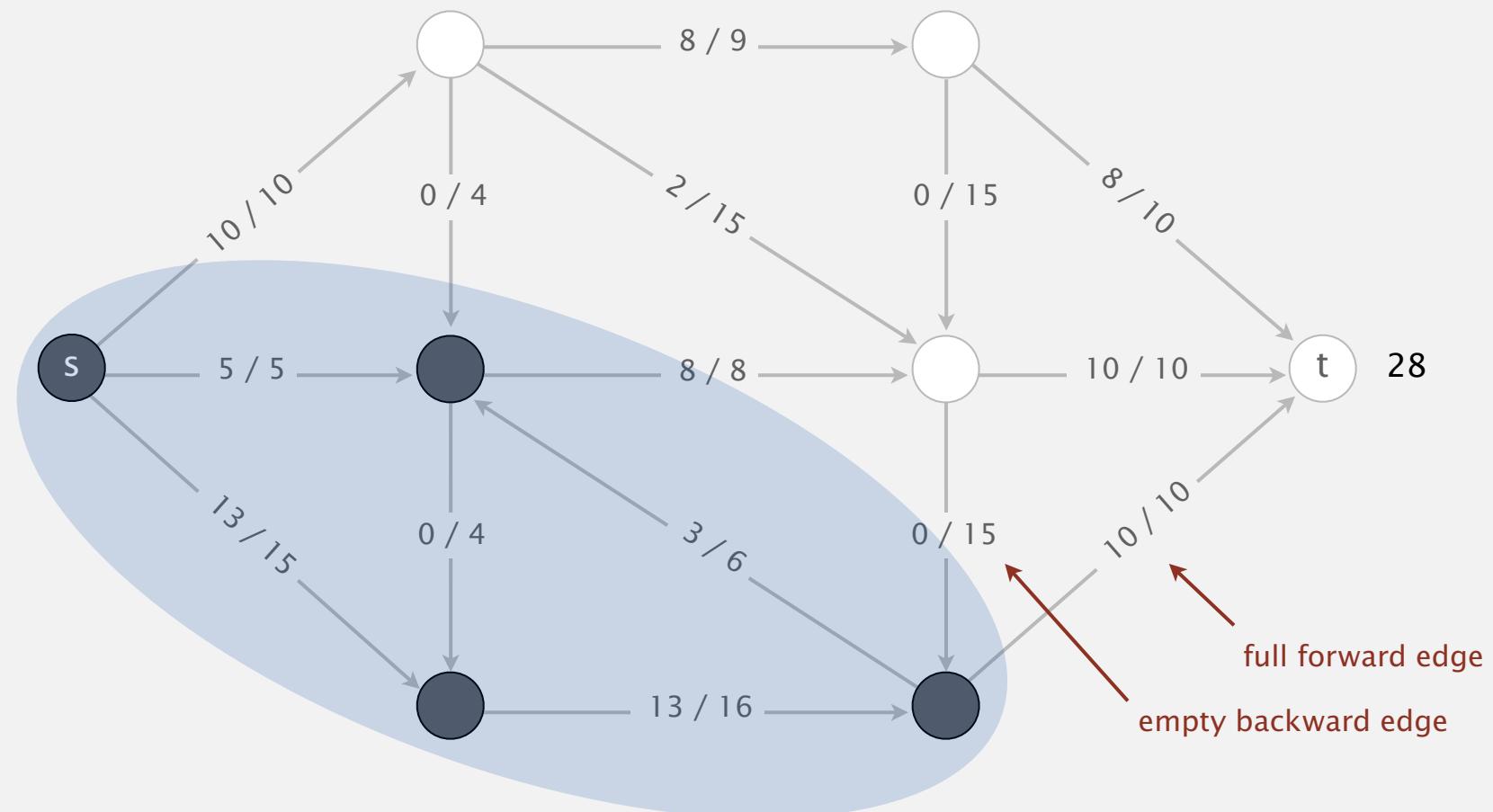


Idea: increase flow along augmenting paths

Termination. All paths from s to t are blocked by either a

- Full forward edge.
- Empty backward edge.

no more augmenting paths



Ford-Fulkerson algorithm

Ford-Fulkerson algorithm

Start with 0 flow.

While there exists an augmenting path:

- **find an augmenting path**
 - **compute bottleneck capacity**
 - **increase flow on that path by bottleneck capacity**
-

Questions.

- How to compute a mincut?
- How to find an augmenting path?
- If FF terminates, does it always compute a maxflow?
- Does FF always terminate? If so, after how many augmentations?

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ ***Ford-Fulkerson algorithm***
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ *Java implementation*
- ▶ *applications*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

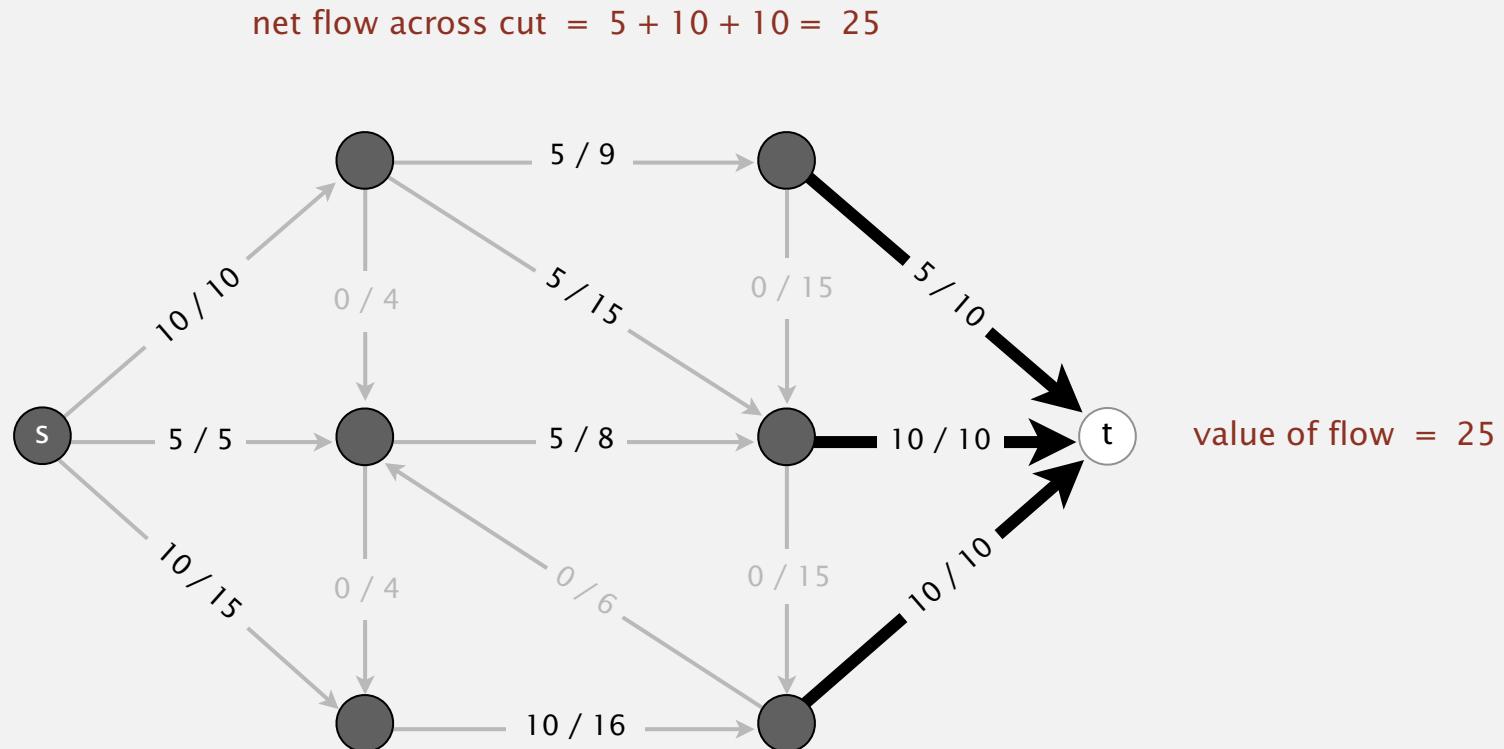
6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ *Java implementation*
- ▶ *applications*

Relationship between flows and cuts

Def. The **net flow across** a cut (A, B) is the sum of the flows on its edges from A to B minus the sum of the flows on its edges from B to A .

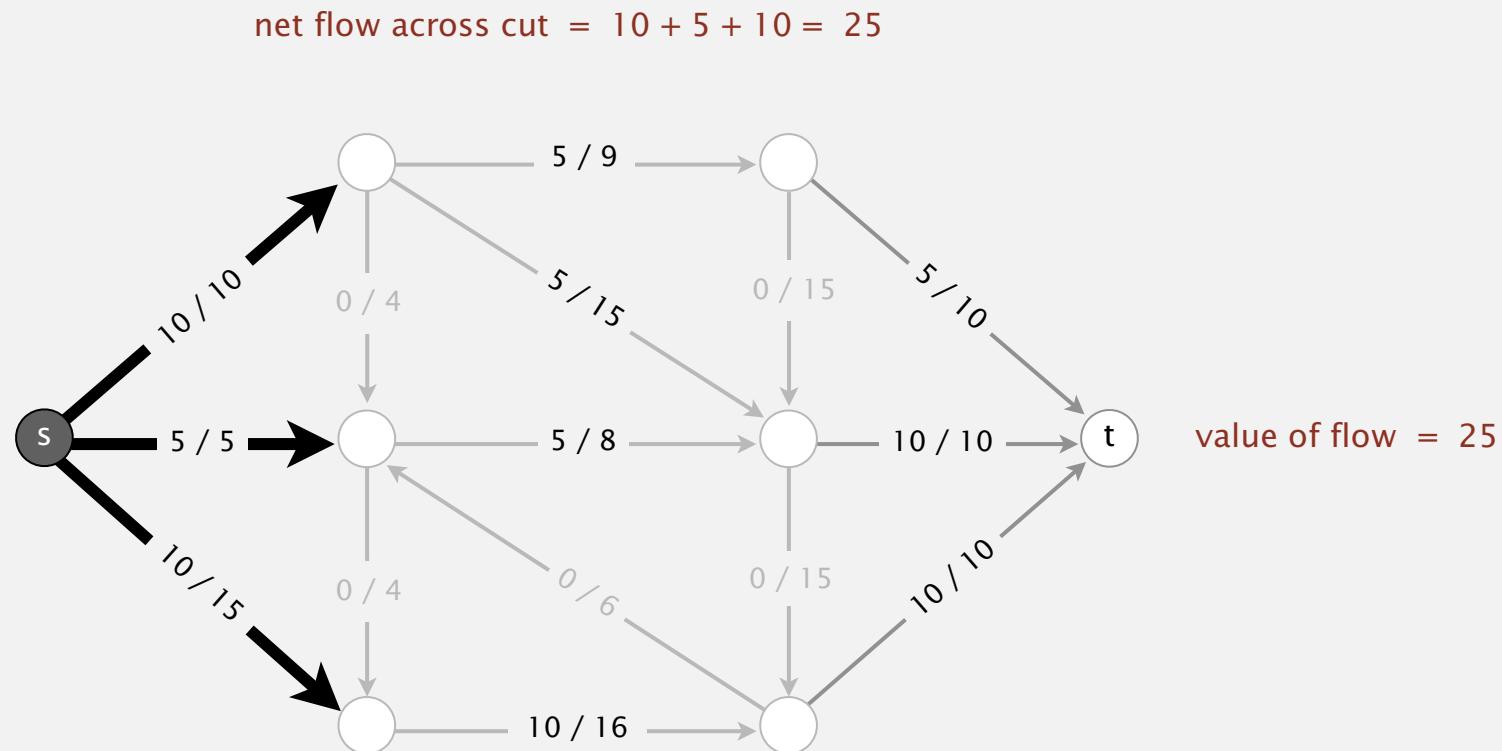
Flow-value lemma. Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f .



Relationship between flows and cuts

Def. The **net flow across** a cut (A, B) is the sum of the flows on its edges from A to B minus the sum of the flows on its edges from B to A .

Flow-value lemma. Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f .

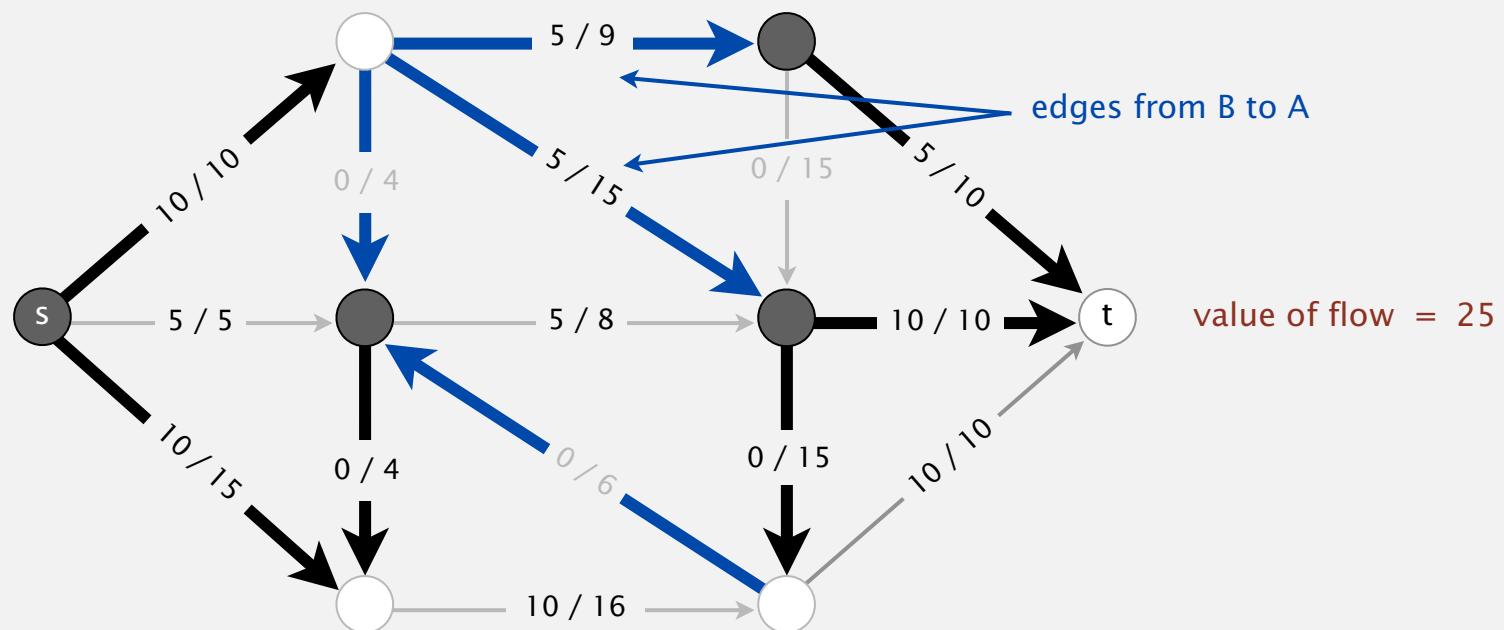


Relationship between flows and cuts

Def. The **net flow across** a cut (A, B) is the sum of the flows on its edges from A to B minus the sum of the flows on its edges from B to A .

Flow-value lemma. Let f be any flow and let (A, B) be any cut. Then, the net flow across (A, B) equals the value of f .

$$\text{net flow across cut} = (10 + 10 + 5 + 10 + 0 + 0) - (5 + 5 + 0 + 0) = 25$$



Relationship between flows and cuts

Def. The **net flow across** a cut (A, B) is the sum of the flows on its edges from A to B **minus** the sum of the flows on its edges from B to A .

Flow-value lemma. Let f be any flow and let (A, B) be any cut. Then, the **net flow across (A, B)** equals the value of f .

Pf. By induction on the size of B .

- Base case: $B = \{t\}$.
- Induction step: remains true by local equilibrium when moving any vertex from A to B .

Corollary. Outflow from s = inflow to t = value of flow.

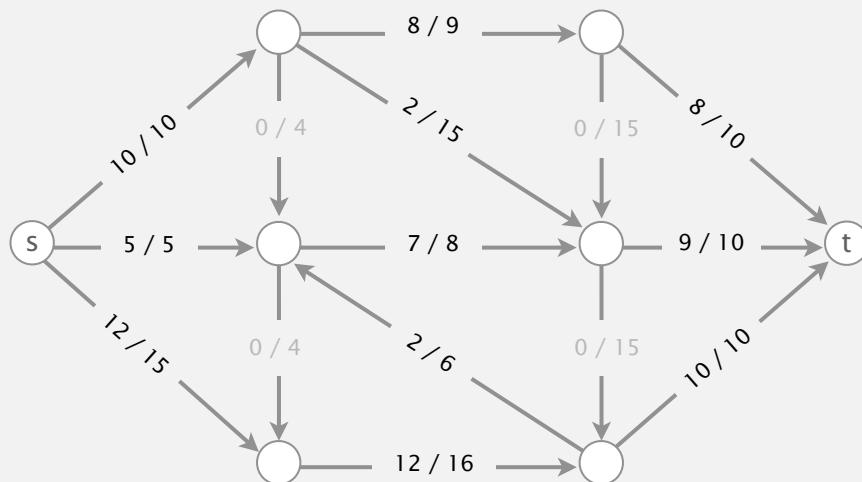
Relationship between flows and cuts

Weak duality. Let f be any flow and let (A, B) be any cut.
Then, the value of the flow \leq the capacity of the cut.

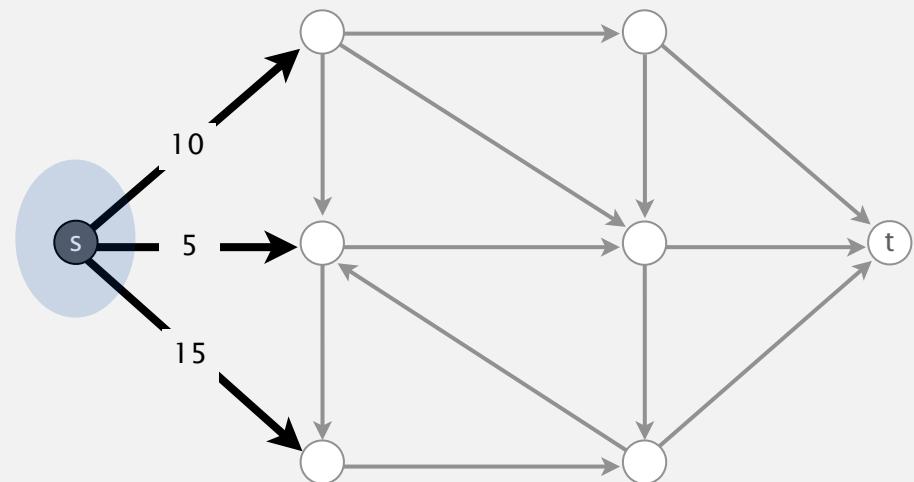
Pf. Value of flow $f =$ net flow across cut $(A, B) \leq$ capacity of cut $(A, B).$

↑
flow-value lemma

↑
flow bounded by capacity



value of flow = 27



capacity of cut = 30

Maxflow-mincut theorem

Augmenting path theorem. A flow f is a maxflow iff no augmenting paths.

Maxflow-mincut theorem. Value of the maxflow = capacity of mincut.

Pf. The following three conditions are equivalent for any flow f :

- i. There exists a cut whose capacity equals the value of the flow f .
- ii. f is a maxflow.
- iii. There is no augmenting path with respect to f .

[i \Rightarrow ii]

- Suppose that (A, B) is a cut with capacity equal to the value of f .
- Then, the value of any flow $f' \leq$ capacity of $(A, B) =$ value of f .
- Thus, f is a maxflow.

↑
weak duality

↑
by assumption

Maxflow-mincut theorem

Augmenting path theorem. A flow f is a maxflow iff no augmenting paths.

Maxflow-mincut theorem. Value of the maxflow = capacity of mincut.

Pf. The following three conditions are equivalent for any flow f :

- i. There exists a cut whose capacity equals the value of the flow f .
- ii. f is a maxflow.
- iii. There is no augmenting path with respect to f .

[ii \Rightarrow iii] We prove contrapositive: \sim iii \Rightarrow \sim ii.

- Suppose that there is an augmenting path with respect to f .
- Can improve flow f by sending flow along this path.
- Thus, f is not a maxflow.

Maxflow-mincut theorem

Augmenting path theorem. A flow f is a maxflow iff no augmenting paths.

Maxflow-mincut theorem. Value of the maxflow = capacity of mincut.

Pf. The following three conditions are equivalent for any flow f :

- i. There exists a cut whose capacity equals the value of the flow f .
- ii. f is a maxflow.
- iii. There is no augmenting path with respect to f .

[iii \Rightarrow i]

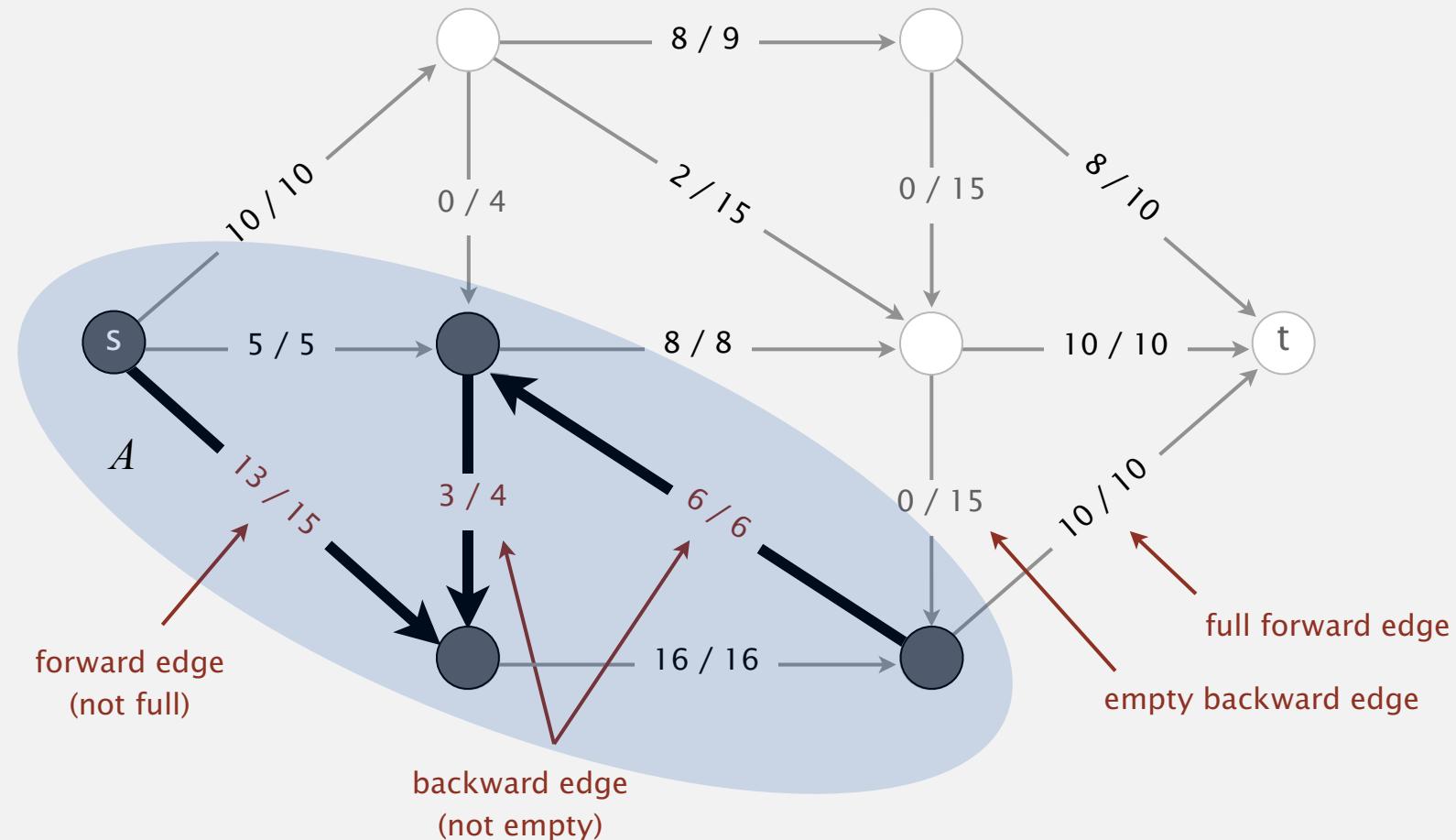
Suppose that there is no augmenting path with respect to f .

- Let (A, B) be a cut where A is the set of vertices connected to s by an undirected path with no full forward or empty backward edges.
- By definition, s is in A ; since no augmenting path, t is in B .
- Capacity of cut = net flow across cut \leftarrow forward edges full; backward edges empty
= value of flow f . \leftarrow flow-value lemma

Computing a mincut from a maxflow

To compute mincut (A, B) from maxflow f :

- By augmenting path theorem, no augmenting paths with respect to f .
- Compute $A = \text{set of vertices connected to } s \text{ by an undirected path}$ with no full forward or empty backward edges.



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ *Java implementation*
- ▶ *applications*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ *Java implementation*
- ▶ *applications*

Ford-Fulkerson algorithm

Ford-Fulkerson algorithm

Start with 0 flow.

While there exists an augmenting path:

- find an augmenting path
- compute bottleneck capacity
- increase flow on that path by bottleneck capacity

Questions.

- How to compute a mincut? Easy. ✓
- How to find an augmenting path? BFS works well.
- If FF terminates, does it always compute a maxflow? Yes. ✓
- Does FF always terminate? If so, after how many augmentations?

yes, provided edge capacities are integers
(or augmenting paths are chosen carefully)

requires clever analysis

Ford-Fulkerson algorithm with integer capacities

Important special case. Edge capacities are integers between 1 and U .

Invariant. The flow is integer-valued throughout Ford-Fulkerson.

Pf. [by induction]

- Bottleneck capacity is an integer.
- Flow on an edge increases/decreases by bottleneck capacity.

Proposition. Number of augmentations \leq the value of the maxflow.

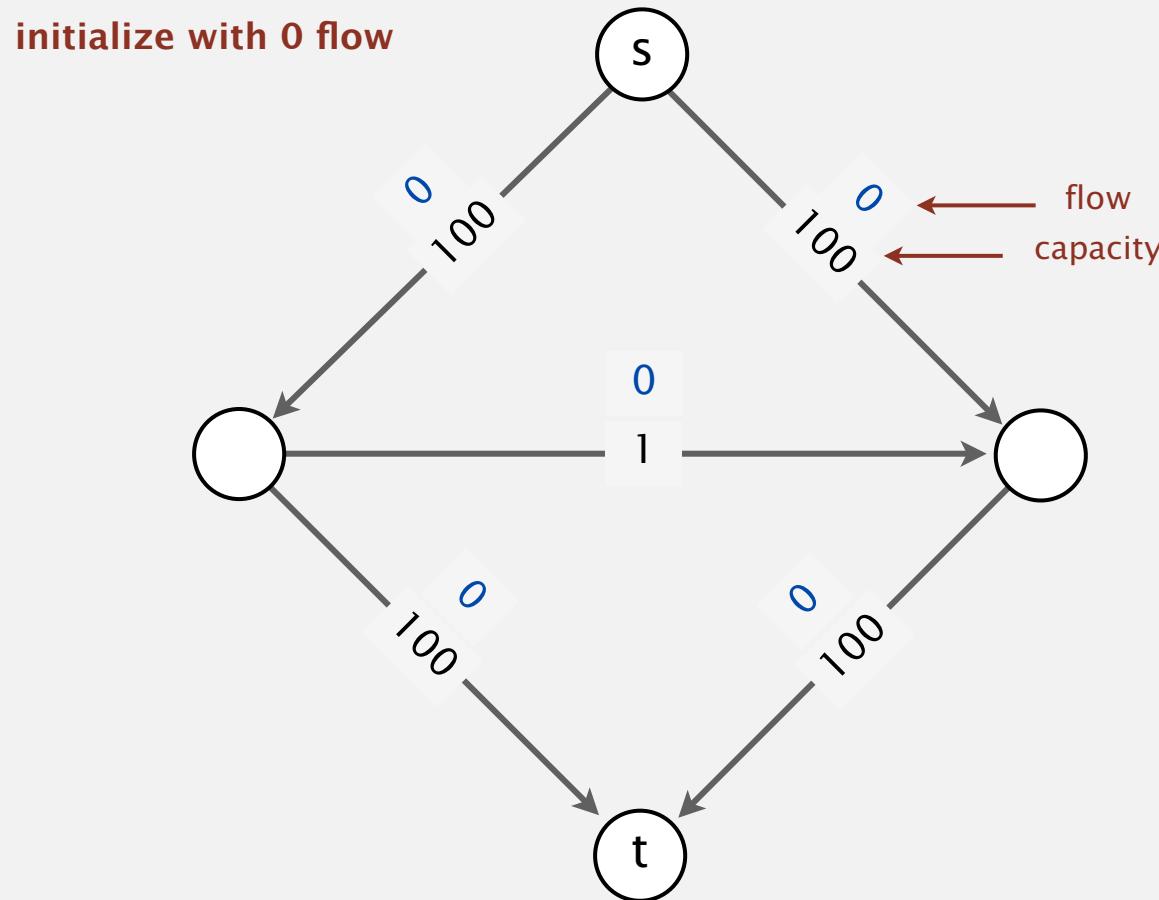
Pf. Each augmentation increases the value by at least 1.

Integrality theorem. There exists an integer-valued maxflow.

Pf. Ford-Fulkerson terminates and maxflow that it finds is integer-valued.

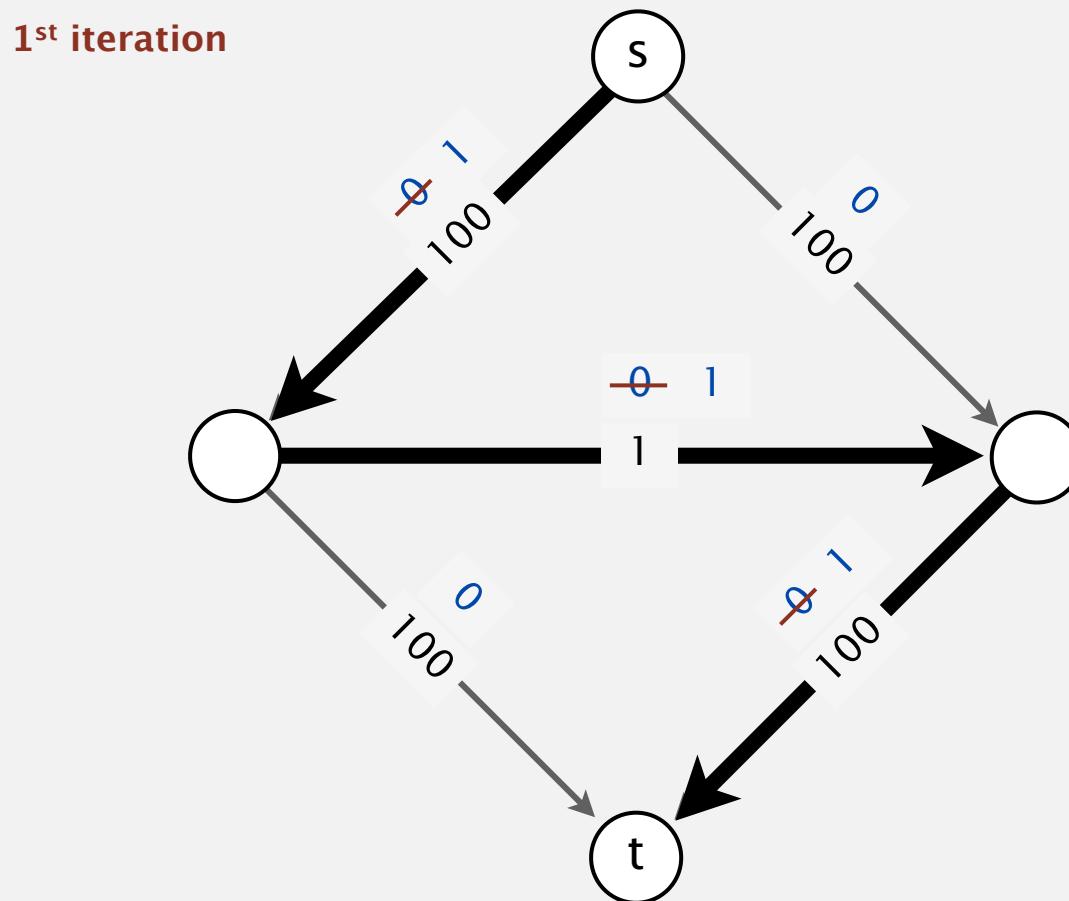
Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



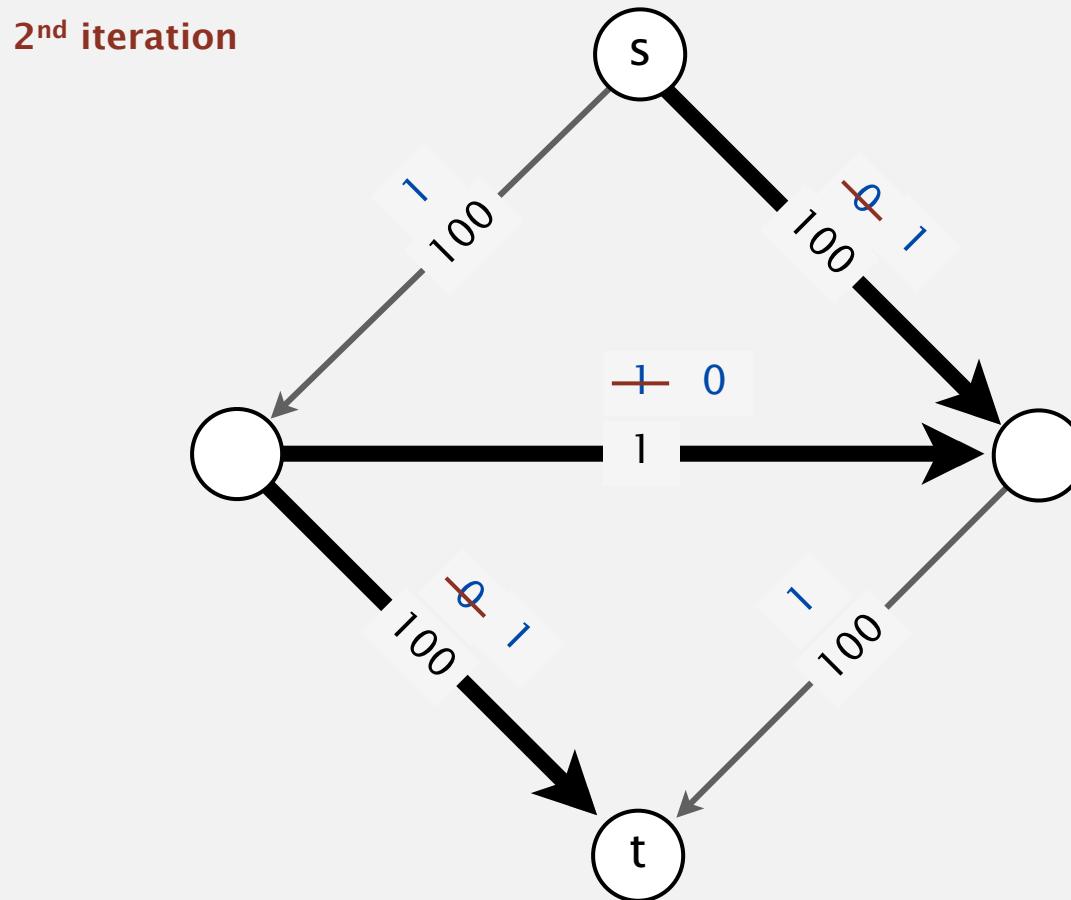
Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



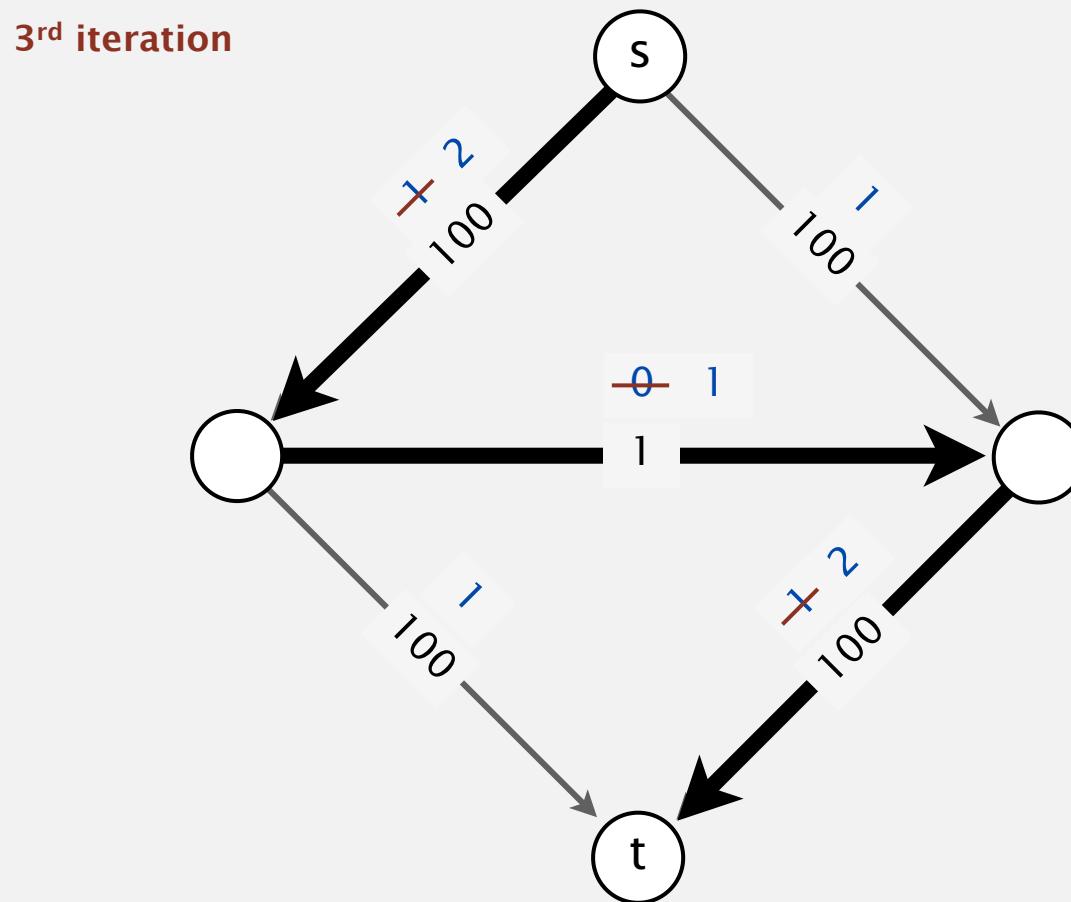
Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



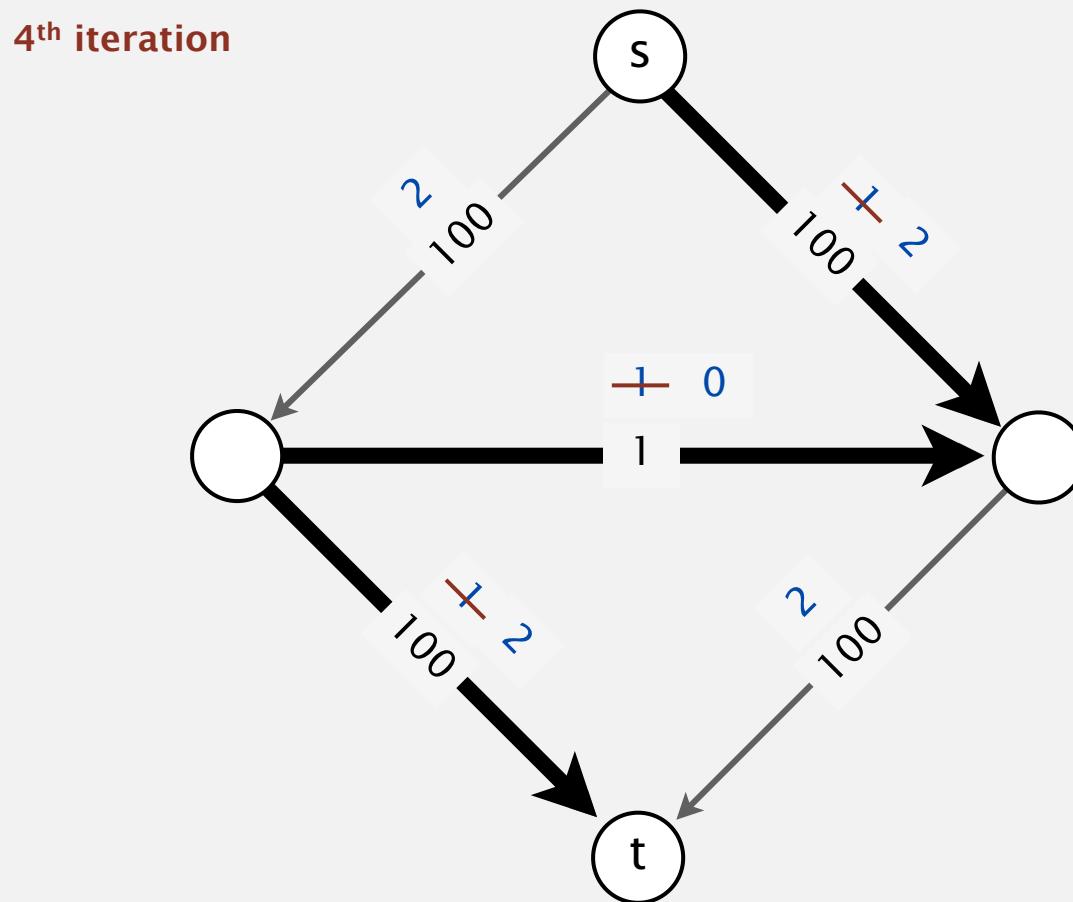
Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



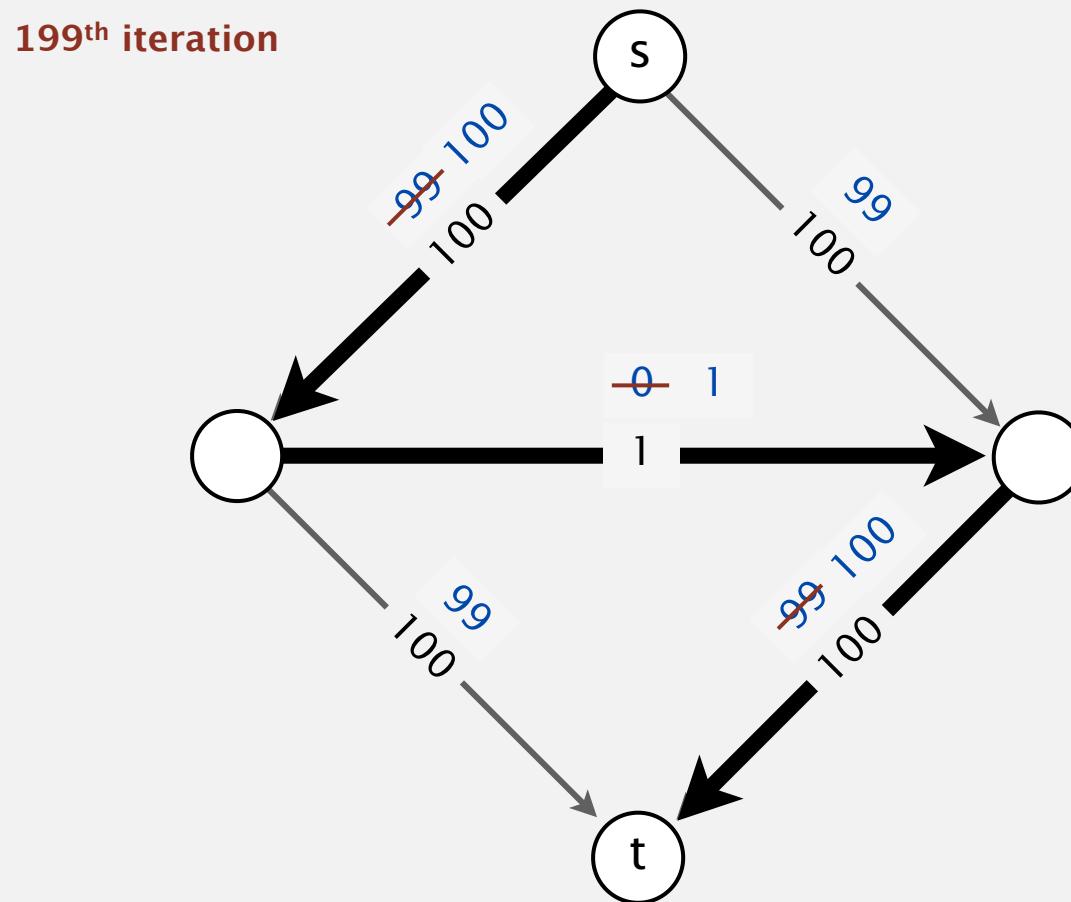
Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



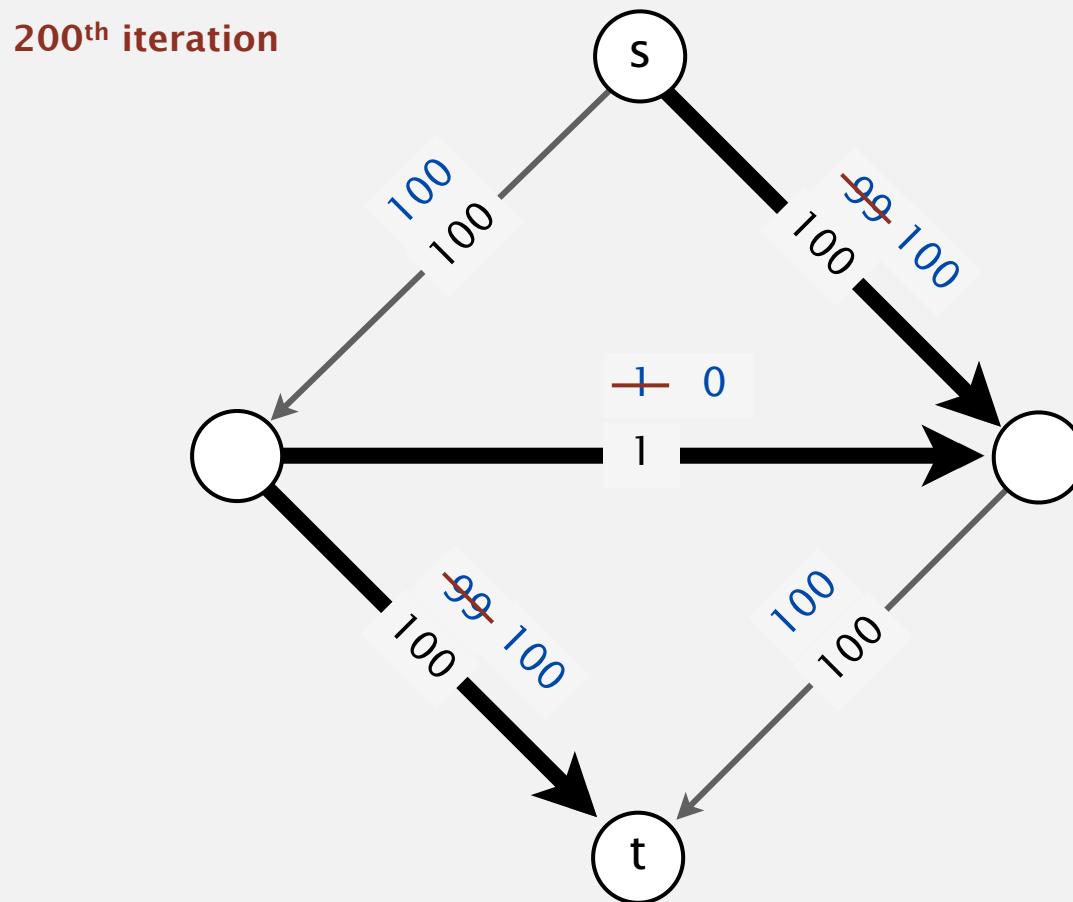
Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.



Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

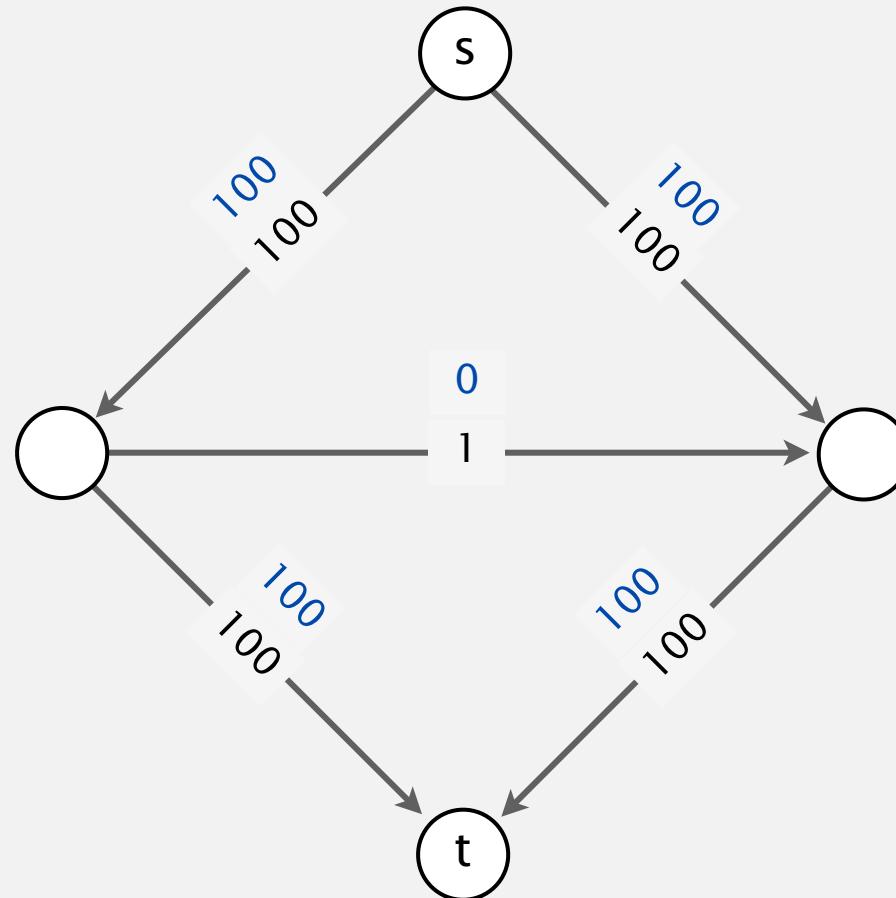


Bad case for Ford-Fulkerson

Bad news. Even when edge capacities are integers, number of augmenting paths could be equal to the value of the maxflow.

can be exponential in input size

Good news. This case is easily avoided. [use shortest/fattest path]

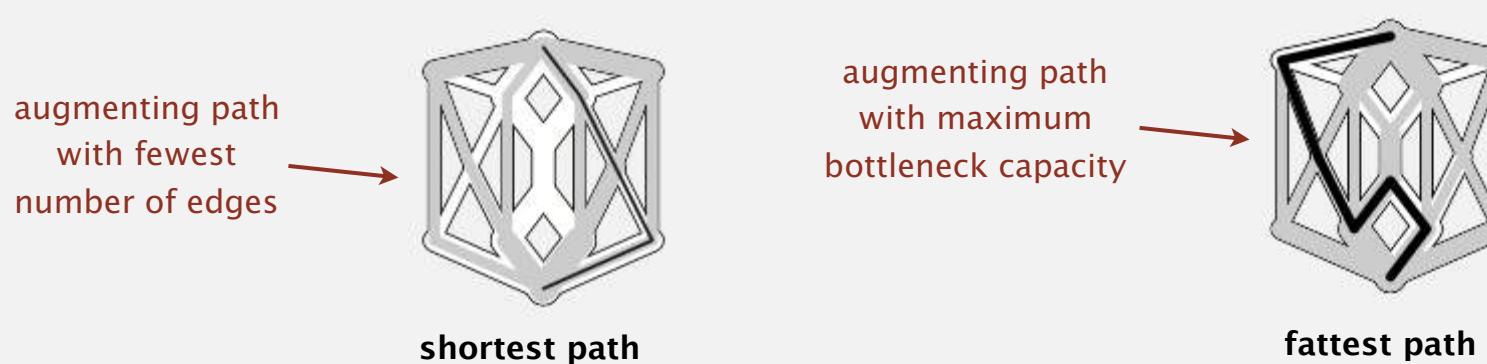


How to choose augmenting paths?

FF performance depends on choice of augmenting paths.

augmenting path	number of paths	implementation
shortest path	$\leq \frac{1}{2} E V$	queue (BFS)
fattest path	$\leq E \ln(E U)$	priority queue
random path	$\leq E U$	randomized queue
DFS path	$\leq E U$	stack (DFS)

digraph with V vertices, E edges, and integer capacities between 1 and U



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ *Java implementation*
- ▶ *applications*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

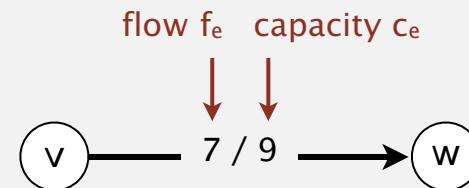
<http://algs4.cs.princeton.edu>

6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ ***Java implementation***
- ▶ *applications*

Flow network representation

Flow edge data type. Associate flow f_e and capacity c_e with edge $e = v \rightarrow w$.



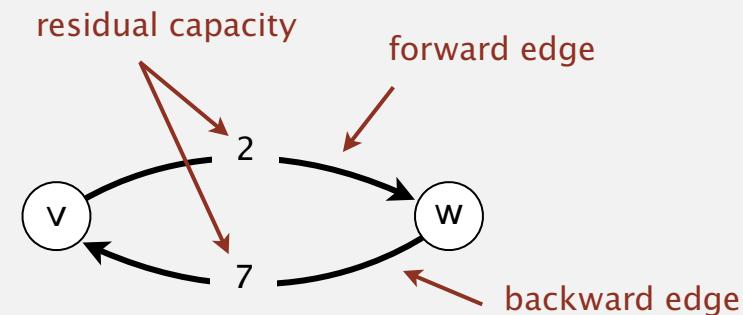
Flow network data type. Need to process edge $e = v \rightarrow w$ in either direction:
Include e in both v and w 's adjacency lists.

Residual capacity.

- Forward edge: residual capacity $= c_e - f_e$.
- Backward edge: residual capacity $= f_e$.

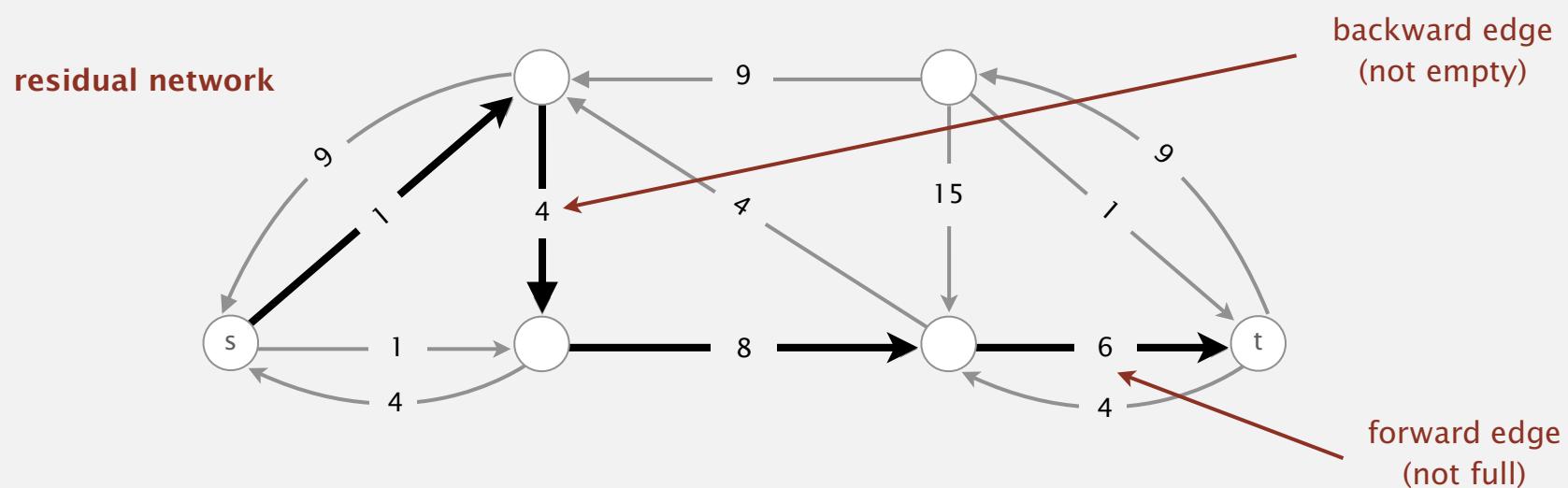
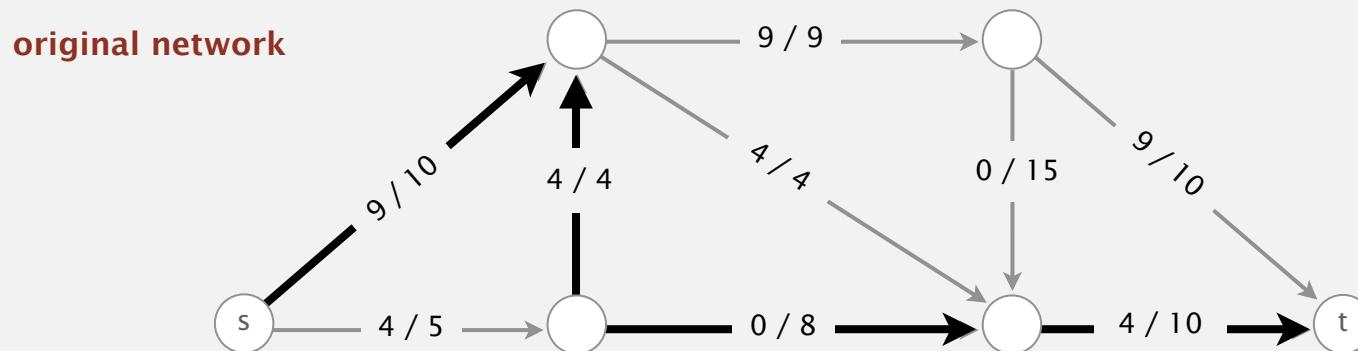
Augment flow.

- Forward edge: add Δ .
- Backward edge: subtract Δ .



Flow network representation

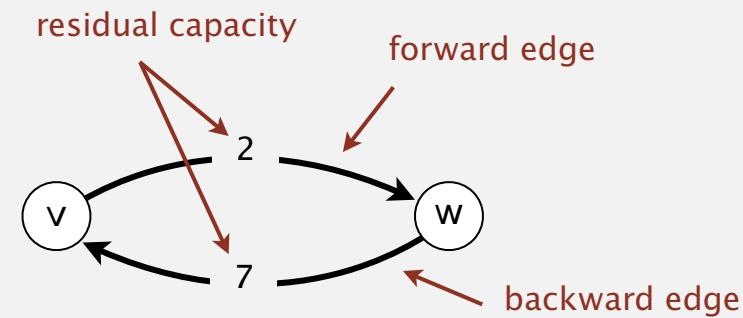
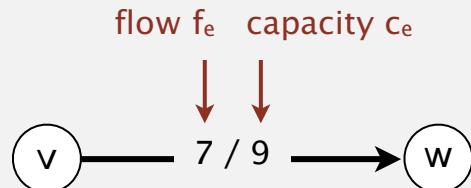
Residual network. A useful view of a flow network.



Key point. Augmenting path in original network is equivalent to directed path in residual network.

Flow edge API

public class FlowEdge	
FlowEdge(int v, int w, double capacity)	<i>create a flow edge v→w</i>
int from()	<i>vertex this edge points from</i>
int to()	<i>vertex this edge points to</i>
int other(int v)	<i>other endpoint</i>
double capacity()	<i>capacity of this edge</i>
double flow()	<i>flow in this edge</i>
double residualCapacityTo(int v)	<i>residual capacity toward v</i>
void addResidualFlowTo(int v, double delta)	<i>add delta flow toward v</i>
String toString()	<i>string representation</i>



Flow edge: Java implementation

```
public class FlowEdge
{
    private final int v, w;          // from and to
    private final double capacity;   // capacity
    private double flow;            // flow
```

← flow variable
(mutable)

```
public FlowEdge(int v, int w, double capacity)
{
    this.v      = v;
    this.w      = w;
    this.capacity = capacity;
}

public int from()      { return v; }
public int to()        { return w; }
public double capacity() { return capacity; }
public double flow()    { return flow; }

public int other(int vertex)
{
    if      (vertex == v) return w;
    else if (vertex == w) return v;
    else throw new RuntimeException("Illegal endpoint");
}

public double residualCapacityTo(int vertex)      {...}
public void addResidualFlowTo(int vertex, double delta) {...}
```

← next slide

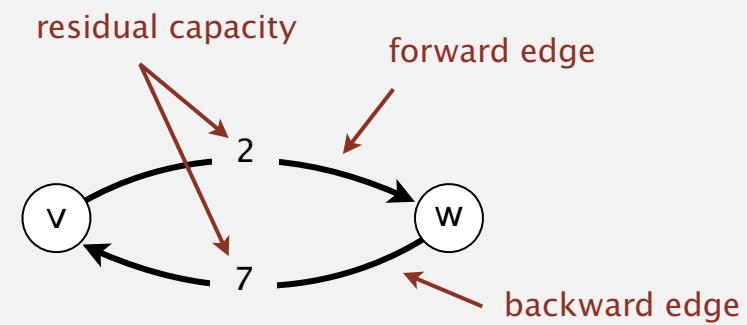
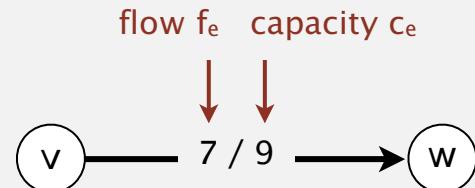
Flow edge: Java implementation

```
public double residualCapacityTo(int vertex)
{
    if (vertex == v) return flow;
    else if (vertex == w) return capacity - flow;
    else throw new IllegalArgumentException();
}
```

← backward edge
← forward edge

```
public void addResidualFlowTo(int vertex, double delta)
{
    if (vertex == v) flow -= delta;
    else if (vertex == w) flow += delta;
    else throw new IllegalArgumentException();
}
```

← backward edge
← forward edge



Flow network API

public class FlowNetwork	
FlowNetwork(int V)	<i>create an empty flow network with V vertices</i>
FlowNetwork(In in)	<i>construct flow network input stream</i>
void addEdge(FlowEdge e)	<i>add flow edge e to this flow network</i>
Iterable<FlowEdge> adj(int v)	<i>forward and backward edges incident to v</i>
Iterable<FlowEdge> edges()	<i>all edges in this flow network</i>
int V()	<i>number of vertices</i>
int E()	<i>number of edges</i>
String toString()	<i>string representation</i>

Conventions. Allow self-loops and parallel edges.

Flow network: Java implementation

```
public class FlowNetwork
{
    private final int V;
    private Bag<FlowEdge>[] adj;

    public FlowNetwork(int V)
    {
        this.V = V;
        adj = (Bag<FlowEdge>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<FlowEdge>();
    }

    public void addEdge(FlowEdge e)
    {
        int v = e.from();
        int w = e.to();
        adj[v].add(e);
        adj[w].add(e);
    }

    public Iterable<FlowEdge> adj(int v)
    { return adj[v]; }
}
```

same as EdgeWeightedGraph,
but adjacency lists of
FlowEdges instead of Edges

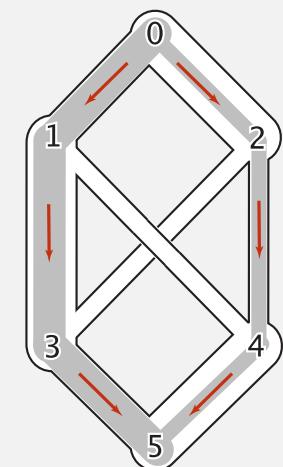
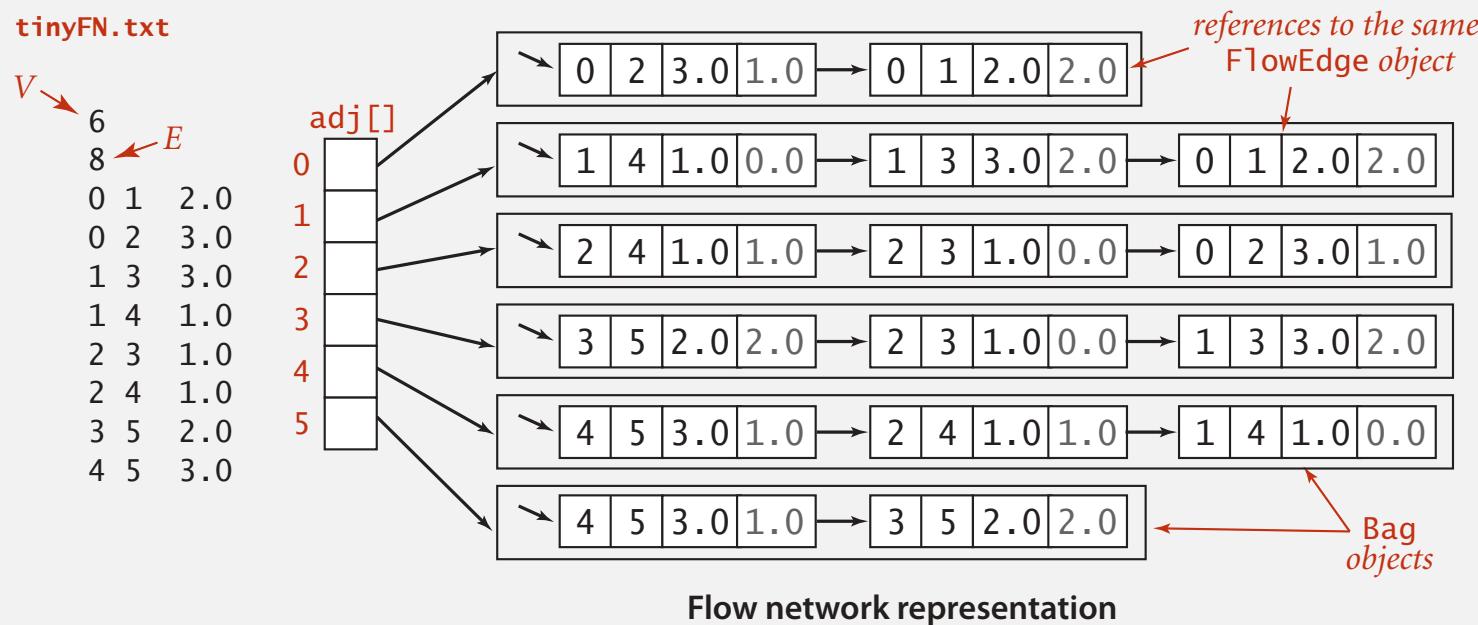


add forward edge
add backward edge



Flow network: adjacency-lists representation

Maintain vertex-indexed array of FlowEdge lists (use Bag abstraction).



Ford-Fulkerson: Java implementation

```
public class FordFulkerson
{
    private boolean[] marked;      // true if s->v path in residual network
    private FlowEdge[] edgeTo;     // last edge on s->v path
    private double value;         // value of flow

    public FordFulkerson(FlowNetwork G, int s, int t)
    {
        value = 0.0;
        while (hasAugmentingPath(G, s, t))
        {
            double bottle = Double.POSITIVE_INFINITY;
            for (int v = t; v != s; v = edgeTo[v].other(v))
                bottle = Math.min(bottle, edgeTo[v].residualCapacityTo(v)); ← compute bottleneck capacity

            for (int v = t; v != s; v = edgeTo[v].other(v))
                edgeTo[v].addResidualFlowTo(v, bottle); ← augment flow

            value += bottle;
        }
    }

    private boolean hasAugmentingPath(FlowNetwork G, int s, int t)
    { /* See next slide. */ }

    public double value()
    { return value; }

    public boolean inCut(int v) ← is v reachable from s in residual network?
    { return marked[v]; }
}
```

Finding a shortest augmenting path (cf. breadth-first search)

```
private boolean hasAugmentingPath(FlowNetwork G, int s, int t)
{
    edgeTo = new FlowEdge[G.V()];
    marked = new boolean[G.V()];

    Queue<Integer> queue = new Queue<Integer>();
    queue.enqueue(s);
    marked[s] = true;
    while (!queue.isEmpty())
    {
        int v = queue.dequeue();

        for (FlowEdge e : G.adj(v))          found path from s to w
        {                                     in the residual network?
            int w = e.other(v);
            if (e.residualCapacityTo(w) > 0 && !marked[w])
            {
                edgeTo[w] = e;           save last edge on path to w;
                marked[w] = true;        ← mark w;
                queue.enqueue(w);       add w to the queue
            }
        }
        return marked[t];      ← is t reachable from s in residual network?
    }
}
```

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ ***Java implementation***
- ▶ *applications*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

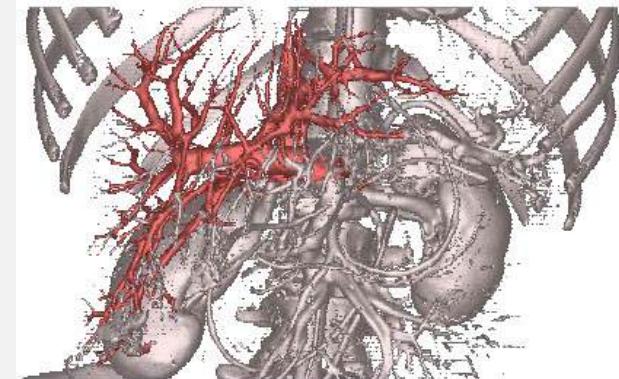
6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ *Java implementation*
- ▶ ***applications***

Maxflow and mincut applications

Maxflow/mincut is a widely applicable problem-solving model.

- Data mining.
- Open-pit mining.
- Bipartite matching.
- Network reliability.
- Baseball elimination.
- Image segmentation.
- Network connectivity.
- Distributed computing.
- Security of statistical data.
- Egalitarian stable matching.
- Multi-camera scene reconstruction.
- Sensor placement for homeland security.
- Many, many, more.



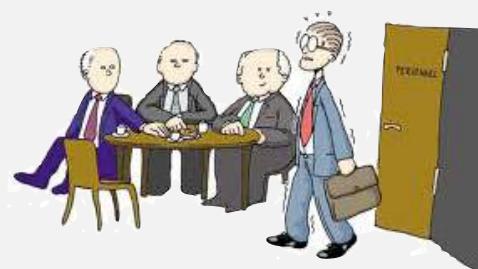
liver and hepatic vascularization segmentation

Bipartite matching problem

N students apply for N jobs.



Each gets several offers.



Is there a way to match all students to jobs?



bipartite matching problem

1	Alice	6	Adobe
	Adobe		Alice
	Amazon		Bob
	Google		Carol
2	Bob	7	Amazon
	Adobe		Alice
	Amazon		Bob
3	Carol		Dave
	Adobe		Eliza
	Facebook	8	Facebook
	Google		Carol
4	Dave	9	Google
	Amazon		Alice
	Yahoo		Carol
5	Eliza	10	Yahoo
	Amazon		Dave
	Yahoo		Eliza

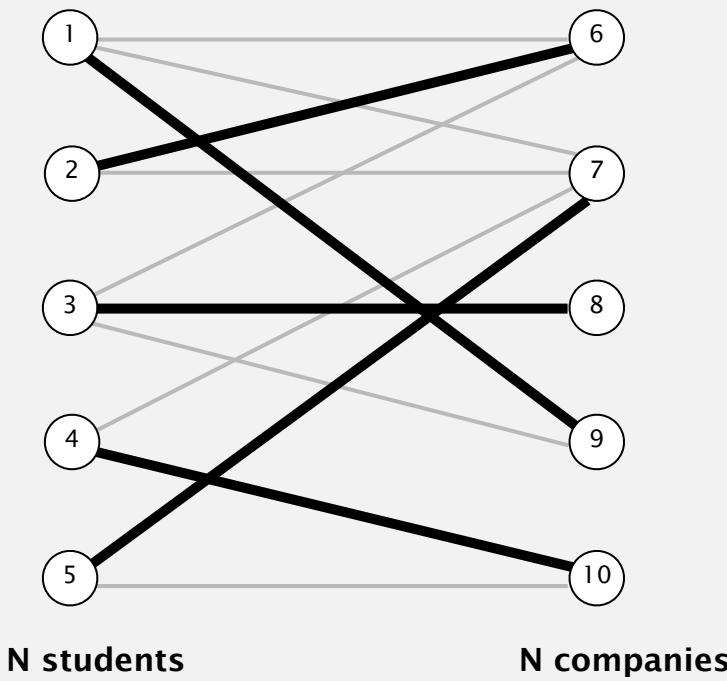
Bipartite matching problem

Given a bipartite graph, find a perfect matching.

perfect matching (solution)

Alice	— Google
Bob	— Adobe
Carol	— Facebook
Dave	— Yahoo
Eliza	— Amazon

bipartite graph

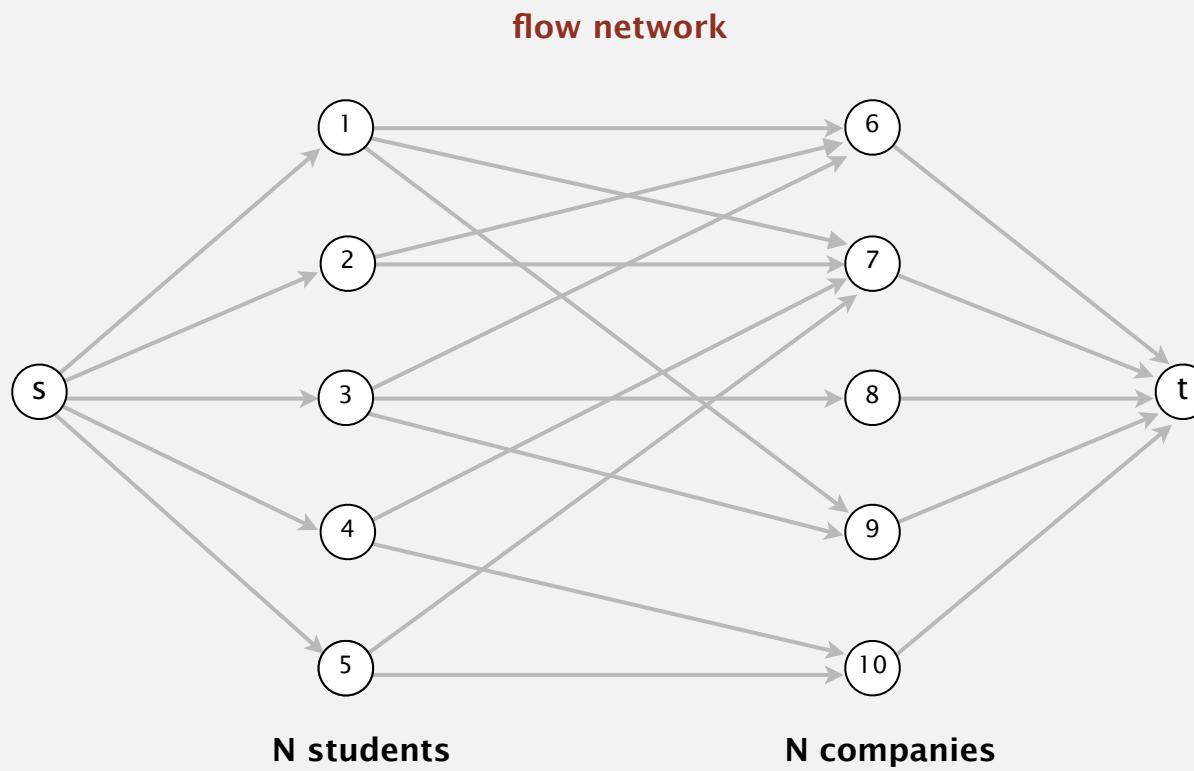


bipartite matching problem

1	Alice	6	Adobe
		7	Alice
2	Bob	Adobe	Adobe
		8	Bob
3	Carol	Amazon	Amazon
		9	Carol
4	Dave	Facebook	Facebook
		10	David
5	Eliza	Google	Eliza
		6	Eliza

Network flow formulation of bipartite matching

- Create s, t , one vertex for each student, and one vertex for each job.
- Add edge from s to each student (capacity 1).
- Add edge from each job to t (capacity 1).
- Add edge from student to each job offered (infinite capacity).

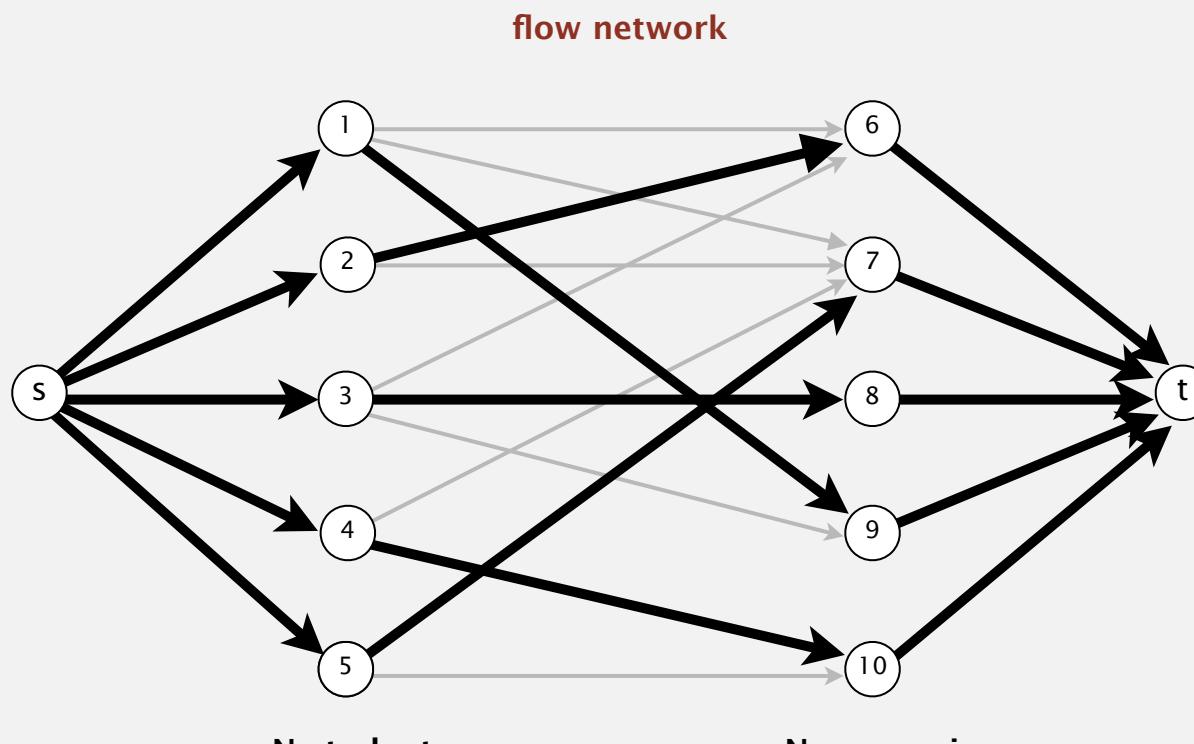


bipartite matching problem

1	Alice	6	Adobe
	Adobe		Alice
	Amazon		Bob
	Google		Carol
2	Bob	7	Amazon
	Adobe		Alice
	Amazon		Bob
3	Carol	8	Dave
	Adobe		Eliza
	Facebook		Facebook
	Google		Carol
4	Dave	9	Google
	Amazon		Alice
	Yahoo		Carol
5	Eliza	10	Yahoo
	Amazon		Dave
	Yahoo		Eliza

Network flow formulation of bipartite matching

1-1 correspondence between perfect matchings in bipartite graph and integer-valued maxflows of value N .

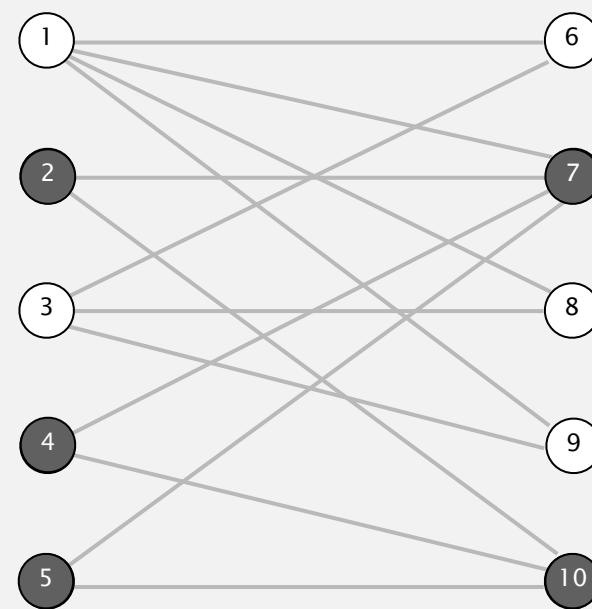


bipartite matching problem

1	Alice	6	Adobe
	Adobe		Alice
	Amazon		Bob
	Google		Carol
2	Bob	7	Amazon
	Adobe		Alice
	Amazon		Bob
3	Carol	8	Dave
	Adobe		Eliza
	Facebook		Facebook
	Google		Carol
4	Dave	9	Google
	Amazon		Alice
	Yahoo		Carol
5	Eliza	10	Yahoo
	Amazon		Dave
	Yahoo		Eliza

What the mincut tells us

Goal. When no perfect matching, explain why.



no perfect matching exists

$$S = \{ 2, 4, 5 \}$$
$$T = \{ 7, 10 \}$$

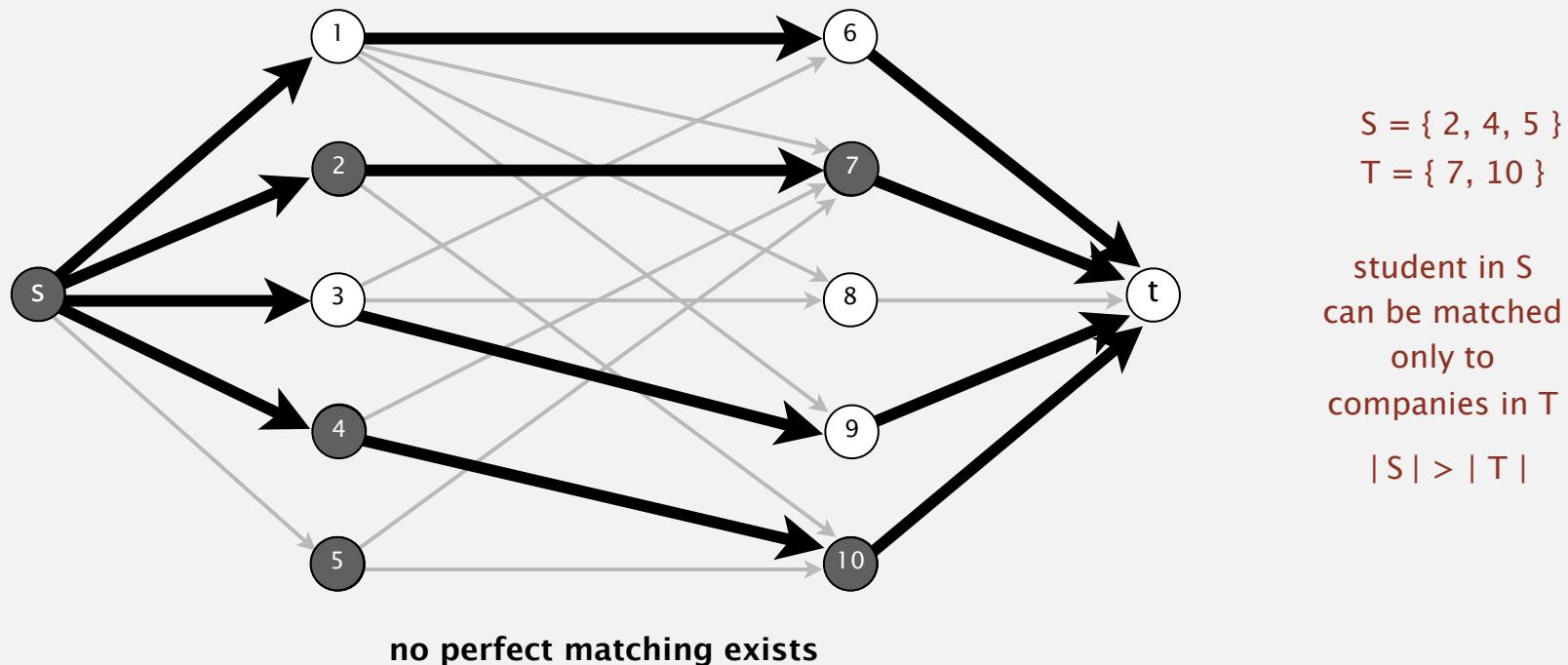
student in S
can be matched
only to
companies in T

$$|S| > |T|$$

What the mincut tells us

Mincut. Consider mincut (A, B) .

- Let S = students on s side of cut.
- Let T = companies on s side of cut.
- Fact: $|S| > |T|$; students in S can be matched only to companies in T .



Bottom line. When no perfect matching, mincut explains why.

Baseball elimination problem

Q. Which teams have a chance of finishing the season with the most wins?

i	team	wins	losses	to play	ATL	PHI	NYM	MON
0	 Atlanta	83	71	8	-	1	6	1
1	 Philly	80	79	3	1	-	0	2
2	 New York	78	78	6	6	0	-	0
3	 Montreal	77	82	3	1	2	0	-

Montreal is mathematically eliminated.

- Montreal finishes with ≤ 80 wins.
- Atlanta already has 83 wins.

Baseball elimination problem

Q. Which teams have a chance of finishing the season with the most wins?

i	team	wins	losses	to play	ATL	PHI	NYM	MON
0	 Atlanta	83	71	8	-	1	6	1
1	 Philly	80	79	3	1	-	0	2
2	 New York	78	78	6	6	0	-	0
3	 Montreal	77	82	3	1	2	0	-

Philadelphia is mathematically eliminated.

- Philadelphia finishes with ≤ 83 wins.
- Either New York or Atlanta will finish with ≥ 84 wins.

Observation. Answer depends not only on how many games already won and left to play, but on **whom** they're against.

Baseball elimination problem

Q. Which teams have a chance of finishing the season with the most wins?

i	team	wins	losses	to play	NYY	BAL	BOS	TOR	DET	
0		New York	75	59	28	-	3	8	7	3
1		Baltimore	71	63	28	3	-	2	7	4
2		Boston	69	66	27	8	2	-	0	0
3		Toronto	63	72	27	7	7	0	-	0
4		Detroit	49	86	27	3	4	0	0	-

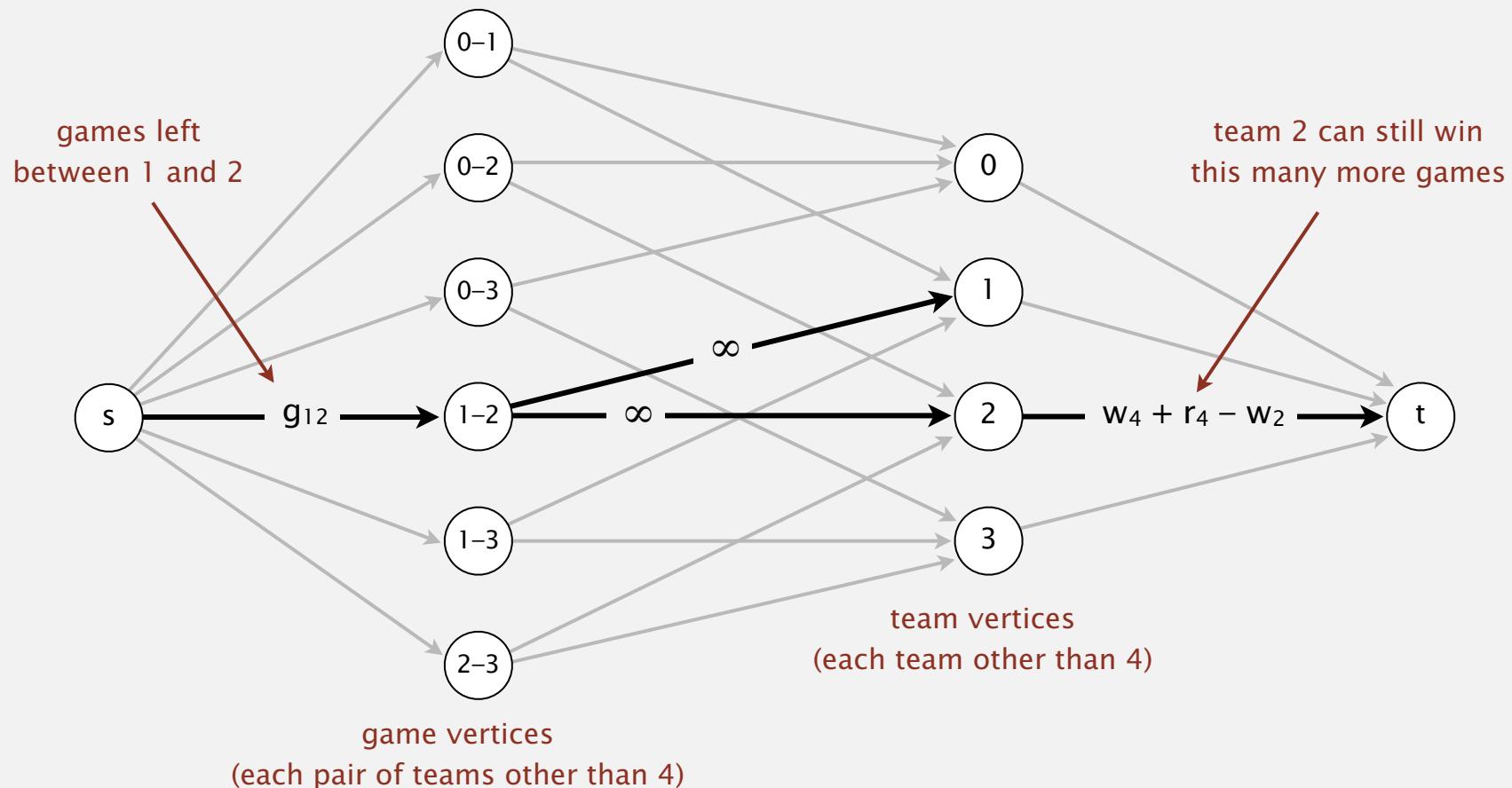
AL East (August 30, 1996)

Detroit is mathematically eliminated.

- Detroit finishes with ≤ 76 wins.
- Wins for $R = \{ \text{NYY}, \text{BAL}, \text{BOS}, \text{TOR} \} = 278$.
- Remaining games among $\{ \text{NYY}, \text{BAL}, \text{BOS}, \text{TOR} \} = 3 + 8 + 7 + 2 + 7 = 27$.
- Average team in R wins $305/4 = 76.25$ games.

Baseball elimination problem: maxflow formulation

Intuition. Remaining games flow from s to t .



Fact. Team 4 not eliminated iff all edges pointing from s are full in maxflow.

Maximum flow algorithms: theory

(Yet another) holy grail for theoretical computer scientists.

year	method	worst case	discovered by
1951	simplex	$E^3 U$	Dantzig
1955	augmenting path	$E^2 U$	Ford-Fulkerson
1970	shortest augmenting path	E^3	Dinitz, Edmonds-Karp
1970	fattest augmenting path	$E^2 \log E \log(EU)$	Dinitz, Edmonds-Karp
1977	blocking flow	$E^{5/2}$	Cherkasky
1978	blocking flow	$E^{7/3}$	Galil
1983	dynamic trees	$E^2 \log E$	Sleator-Tarjan
1985	capacity scaling	$E^2 \log U$	Gabow
1997	length function	$E^{3/2} \log E \log U$	Goldberg-Rao
2012	compact network	$E^2 / \log E$	Orlin
?	?	E	?

maxflow algorithms for sparse digraphs with E edges, integer capacities between 1 and U

Maximum flow algorithms: practice

Warning. Worst-case order-of-growth is generally not useful for predicting or comparing maxflow algorithm performance in practice.

Best in practice. Push-relabel method with gap relabeling: $E^{3/2}$.

On Implementing Push-Relabel Method for the Maximum Flow Problem

Boris V. Cherkassky¹ and Andrew V. Goldberg²

¹ Central Institute for Economics and Mathematics,
Krasikova St. 32, 117418, Moscow, Russia
cher@cemii.msk.su

² Computer Science Department, Stanford University
Stanford, CA 94305, USA
goldberg@cs.stanford.edu

Abstract. We study efficient implementations of the push-relabel method for the maximum flow problem. The resulting codes are faster than the previous codes, and much faster on some problem families. The speedup is due to the combination of heuristics used in our implementations. We also exhibit a family of problems for which the running time of all known methods seem to have a roughly quadratic growth rate.



European Journal of Operational Research 97 (1997) 509–542

EUROPEAN
JOURNAL
OF OPERATIONAL
RESEARCH

Theory and Methodology

Computational investigations of maximum flow algorithms

Ravindra K. Ahuja ^a, Murali Kodialam ^b, Ajay K. Mishra ^c, James B. Orlin ^{d,*}

^a Department of Industrial and Management Engineering, Indian Institute of Technology, Kanpur, 208 016, India

^b AT & T Bell Laboratories, Holmdel, NJ 07733, USA

^c KATZ Graduate School of Business, University of Pittsburgh, Pittsburgh, PA 15260, USA

^d Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA 02139, USA

Received 30 August 1995; accepted 27 June 1996

Summary

Mincut problem. Find an st -cut of minimum capacity.

Maxflow problem. Find an st -flow of maximum value.

Duality. Value of the maxflow = capacity of mincut.

Proven successful approaches.

- Ford-Fulkerson (various augmenting-path strategies).
- Preflow-push (various versions).

Open research challenges.

- Practice: solve real-world maxflow/mincut problems in linear time.
- Theory: prove it for worst-case inputs.
- Still much to be learned!

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

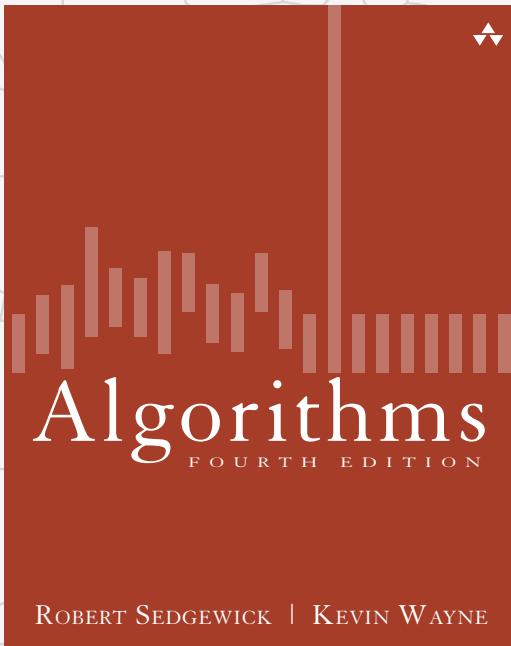
<http://algs4.cs.princeton.edu>

6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ *Java implementation*
- ▶ ***applications***

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

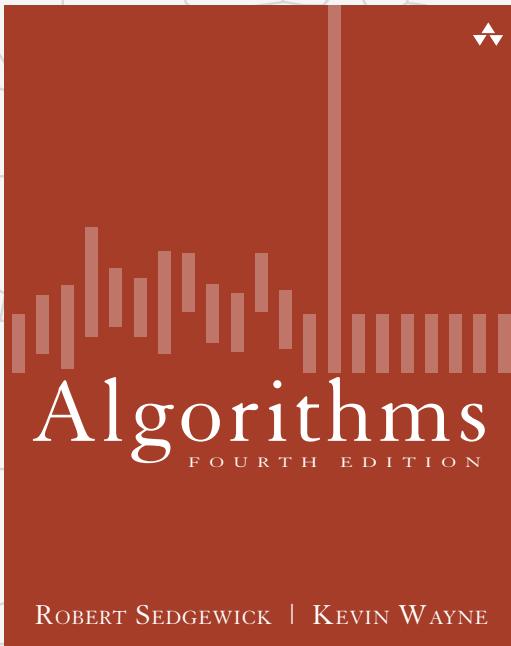


6.4 MAXIMUM FLOW

- ▶ *introduction*
- ▶ *Ford-Fulkerson algorithm*
- ▶ *maxflow-mincut theorem*
- ▶ *running time analysis*
- ▶ *Java implementation*
- ▶ *applications*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE



<http://algs4.cs.princeton.edu>

5.2 TRIES

- ▶ *R-way tries*
- ▶ *ternary search tries*
- ▶ *character-based operations*

Summary of the performance of symbol-table implementations

Order of growth of the frequency of operations.

implementation	typical case			ordered operations	operations on keys
	search	insert	delete		
red-black BST	$\log N$	$\log N$	$\log N$	yes	<code>compareTo()</code>
hash table	$1 \ddagger$	$1 \ddagger$	$1 \ddagger$	no	<code>equals()</code> <code>hashCode()</code>

\ddagger under uniform hashing assumption

Q. Can we do better?

A. Yes, if we can avoid examining the entire key, as with string sorting.

String symbol table basic API

String symbol table. Symbol table specialized to string keys.

```
public class StringST<Value>
```

```
    StringST()
```

create an empty symbol table

```
    void put(String key, Value val)
```

put key-value pair into the symbol table

```
    Value get(String key)
```

return value paired with given key

```
    void delete(String key)
```

delete key and corresponding value

```
    :
```

Goal. Faster than hashing, more flexible than BSTs.

String symbol table implementations cost summary

implementation	character accesses (typical case)					dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt	
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4	
hashing (linear probing)	L	L	L	$4N$ to $16N$	0.76	40.6	

Parameters

- N = number of strings
- L = length of string
- R = radix

file	size	words	distinct
moby.txt	1.2 MB	210 K	32 K
actors.txt	82 MB	11.4 M	900 K

Challenge. Efficient performance for string keys.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.2 TRIES

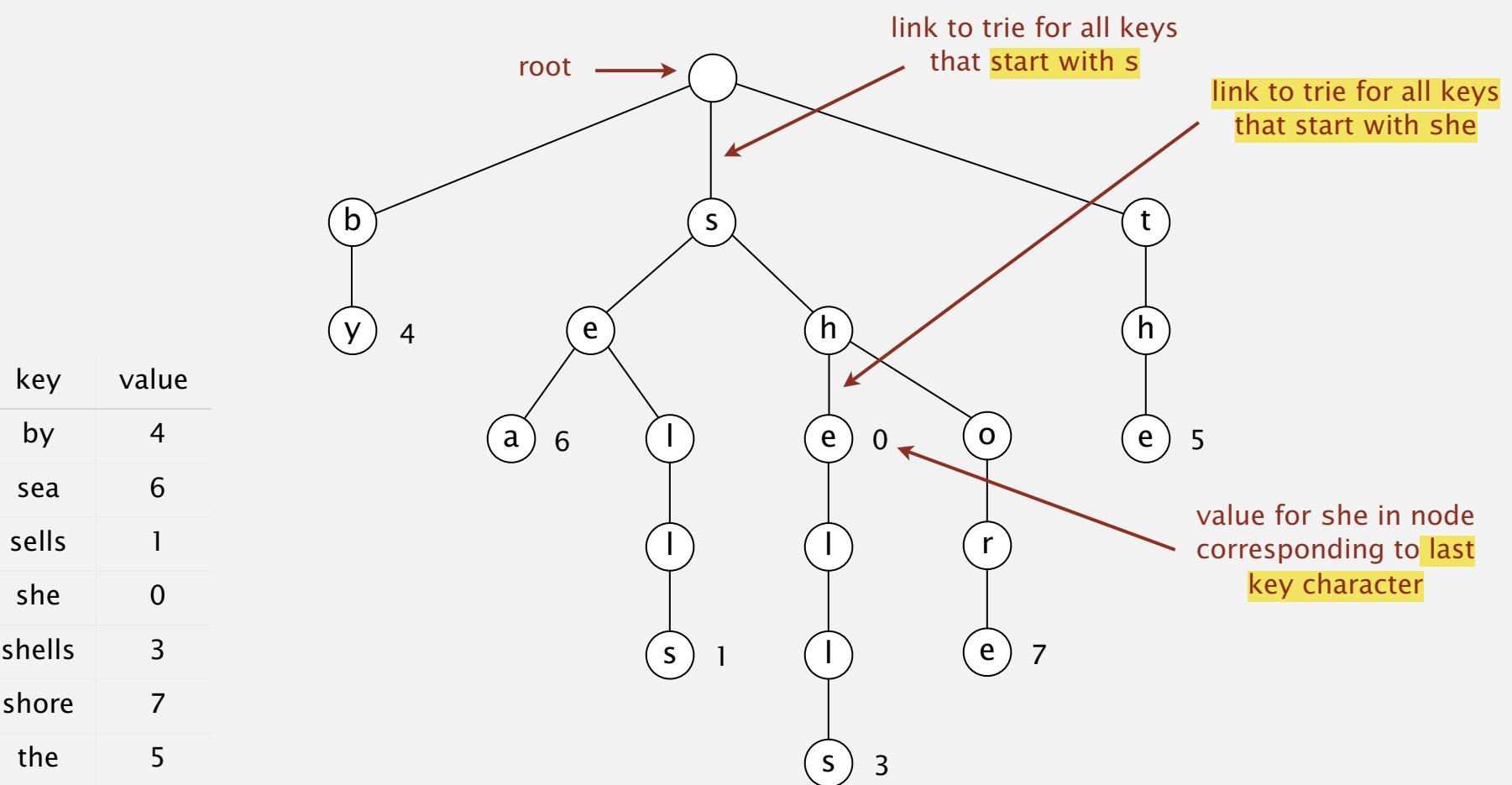
- ▶ *R-way tries*
- ▶ *ternary search tries*
- ▶ *character-based operations*

Tries

Tries. [from retrieval, but pronounced "try"]

- Store characters in nodes (not keys).
- Each node has R children, one for each possible character.
- For now, we do not draw null links.

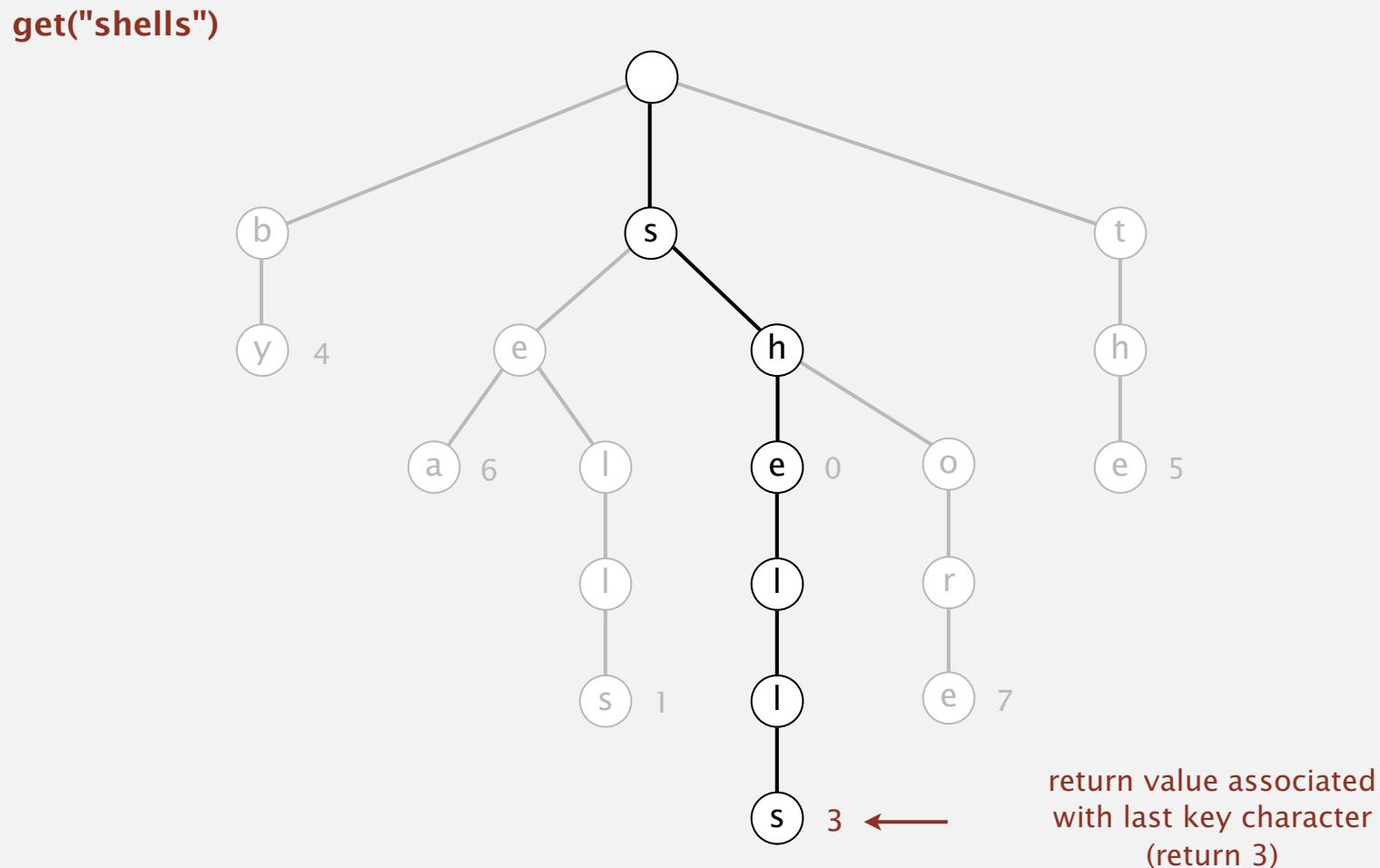
for now we do not
draw null links



Search in a trie

Follow links corresponding to each character in the key.

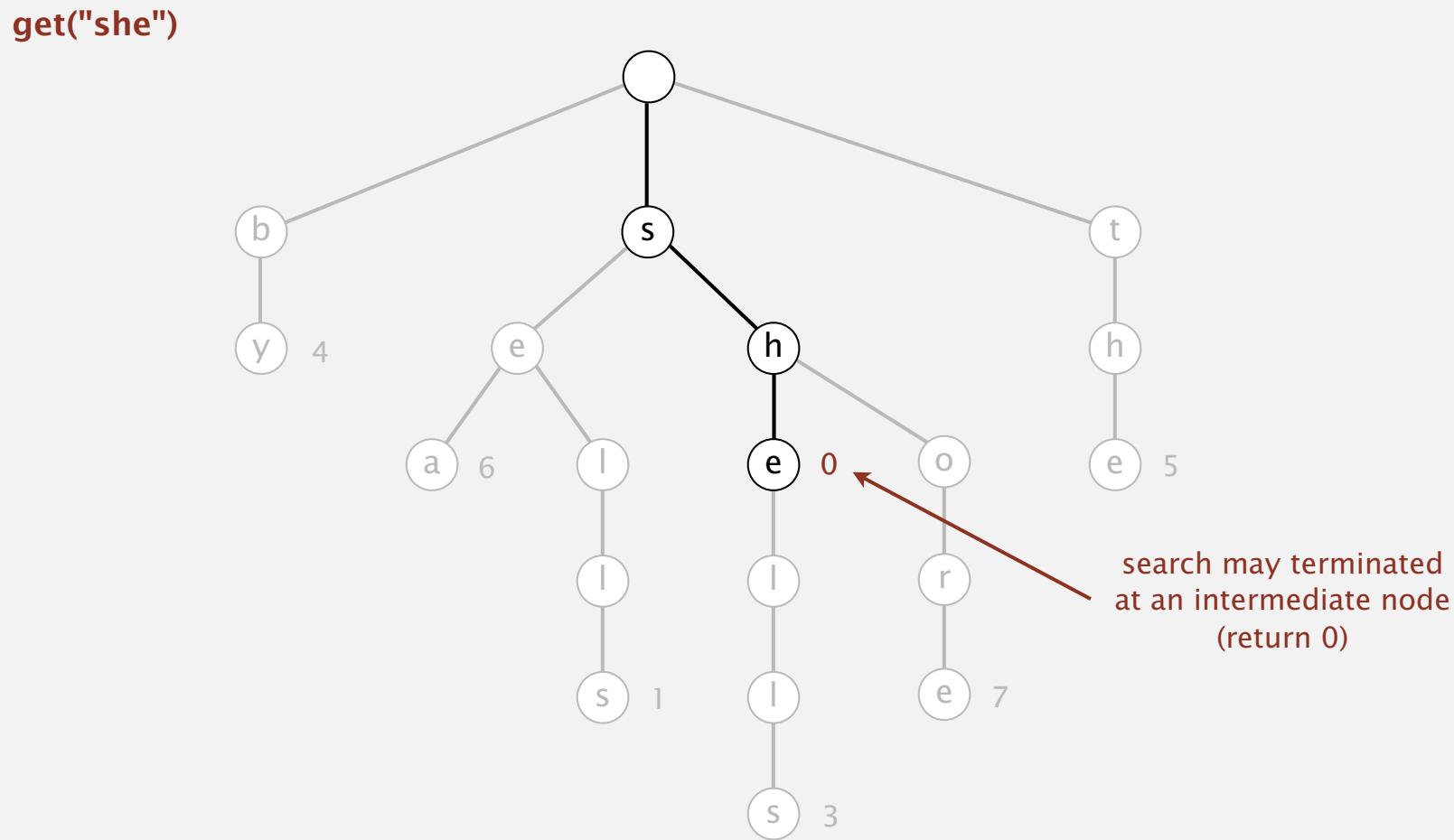
- **Search hit:** node where search ends has a non-null value.
- **Search miss:** reach null link or node where search ends has null value.



Search in a trie

Follow links corresponding to each character in the key.

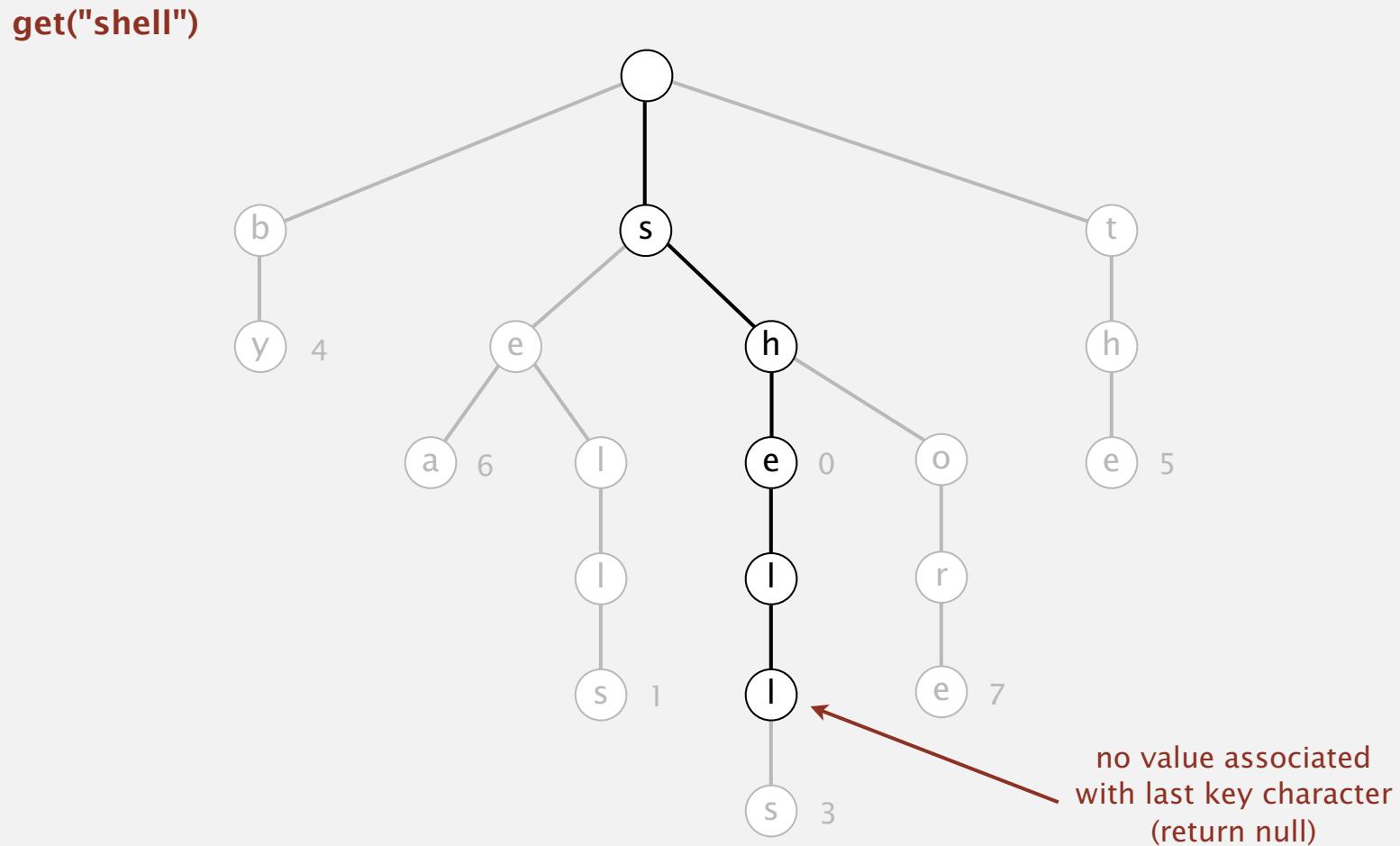
- **Search hit:** node where search ends has a non-null value.
- **Search miss:** reach null link or node where search ends has null value.



Search in a trie

Follow links corresponding to each character in the key.

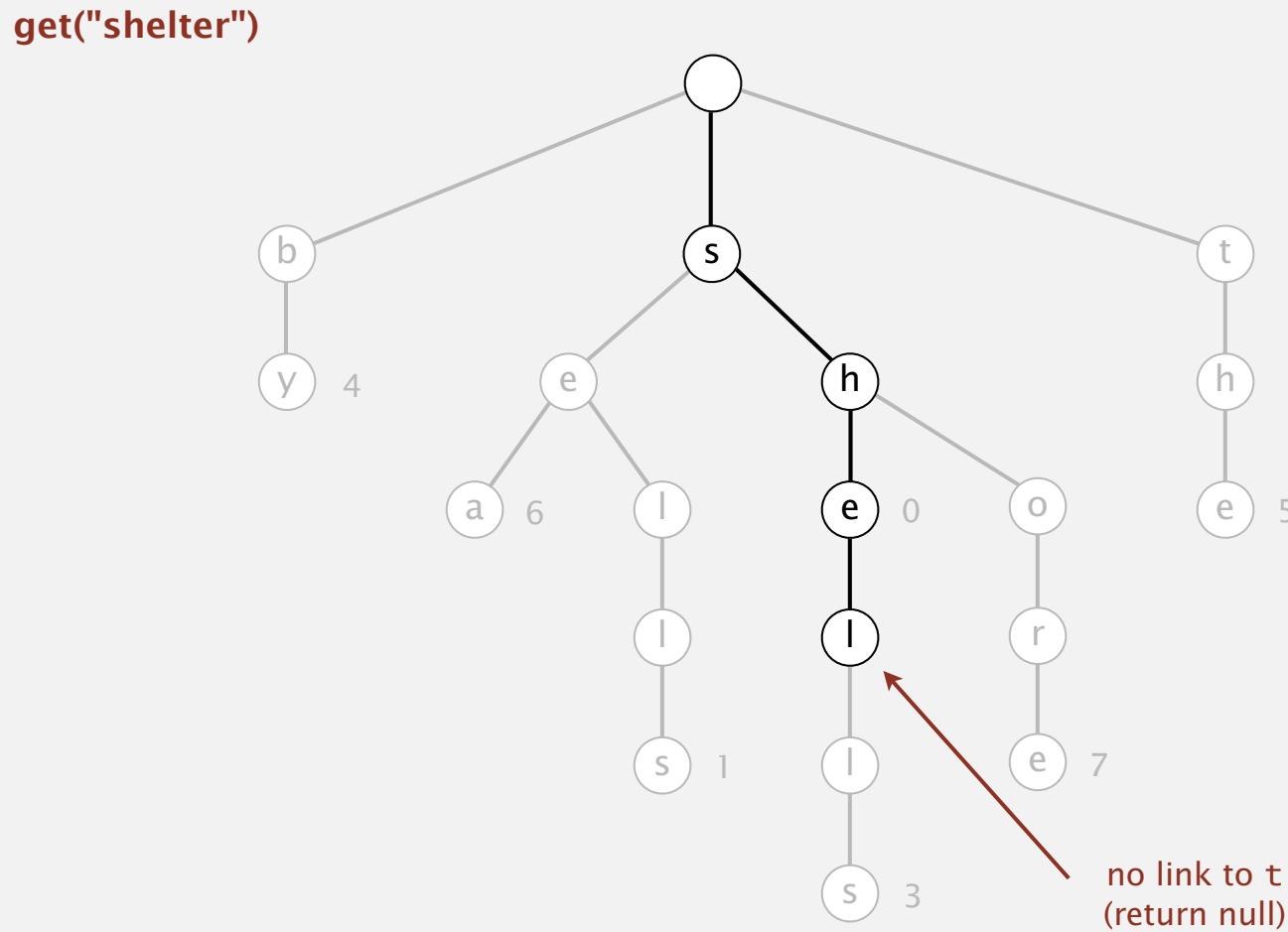
- Search hit: node where search ends has a non-null value.
- Search miss: reach null link or node where search ends has null value.



Search in a trie

Follow links corresponding to each character in the key.

- Search hit: node where search ends has a non-null value.
- Search miss: reach null link or node where search ends has null value.

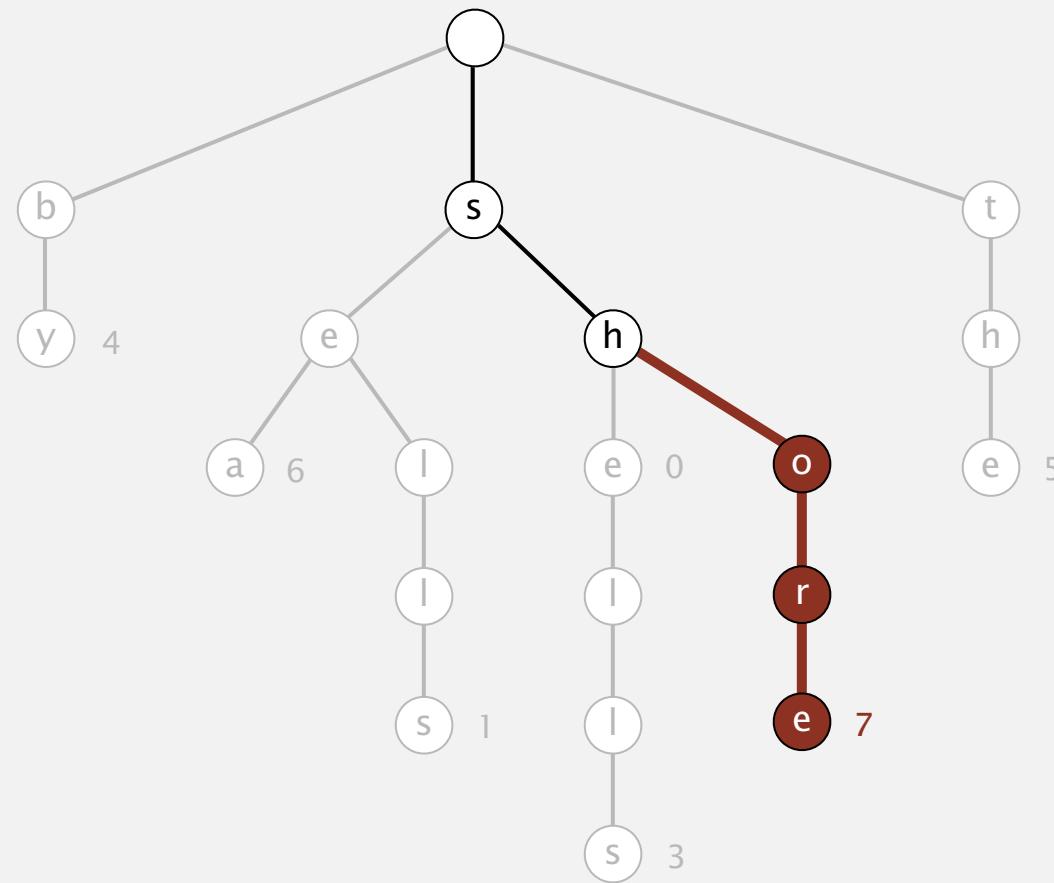


Insertion into a trie

Follow links corresponding to each character in the key.

- Encounter a null link: create new node.
- Encounter the last character of the key: set value in that node.

`put("shore", 7)`



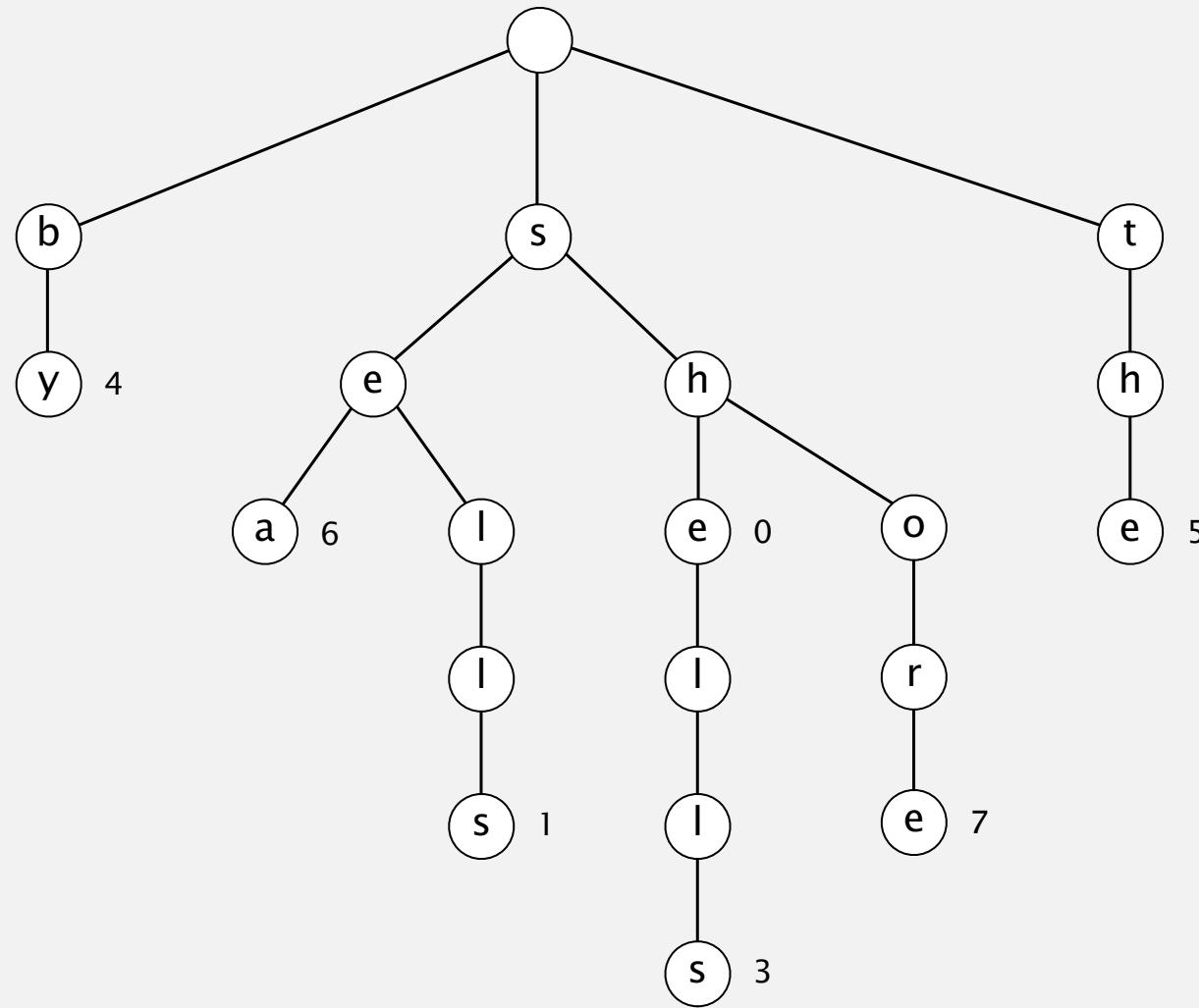
Trie construction demo

trie



Trie construction demo

trie

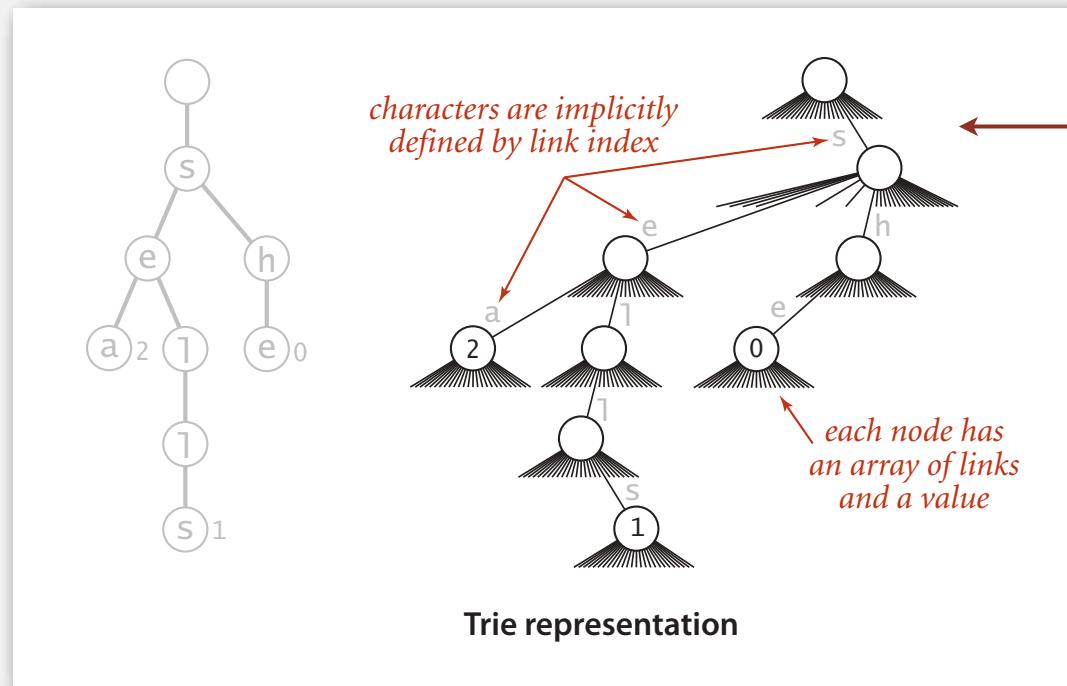


Trie representation: Java implementation

Node. A value, plus references to R nodes.

```
private static class Node
{
    private Object value;
    private Node[] next = new Node[R];
}
```

use Object instead of Value since
no generic array creation in Java



R-way trie: Java implementation

```
public class TrieST<Value>
{
    private static final int R = 256;      ← extended ASCII
    private Node root = new Node();

    private static class Node
    { /* see previous slide */ }

    public void put(String key, Value val)
    { root = put(root, key, val, 0); }

    private Node put(Node x, String key, Value val, int d)
    {
        if (x == null) x = new Node();
        if (d == key.length()) { x.val = val; return x; }
        char c = key.charAt(d);
        x.next[c] = put(x.next[c], key, val, d+1);
        return x;
    }

    ...
}
```

R-way trie: Java implementation (continued)

```
:
public boolean contains(String key)
{  return get(key) != null;  }

public Value get(String key)
{
    Node x = get(root, key, 0);
    if (x == null) return null;
    return (Value) x.val; ← cast needed
}

private Node get(Node x, String key, int d)
{
    if (x == null) return null;
    if (d == key.length()) return x;
    char c = key.charAt(d);
    return get(x.next[c], key, d+1);
}

}
```

Trie performance

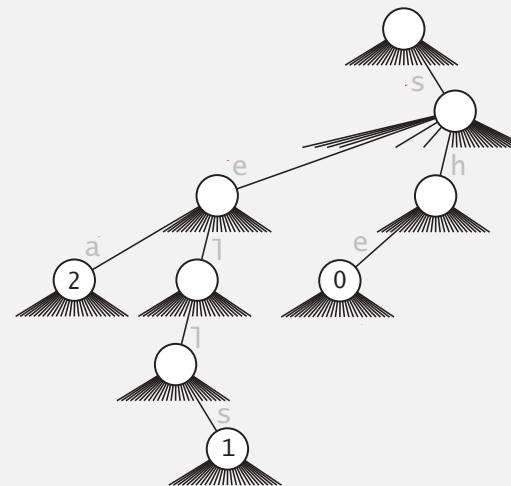
Search hit. Need to examine all L characters for equality.

Search miss.

- Could have mismatch on first character.
- Typical case: examine only a few characters (sublinear).

Space. R null links at each leaf.

(but sublinear space possible if many short strings share common prefixes)



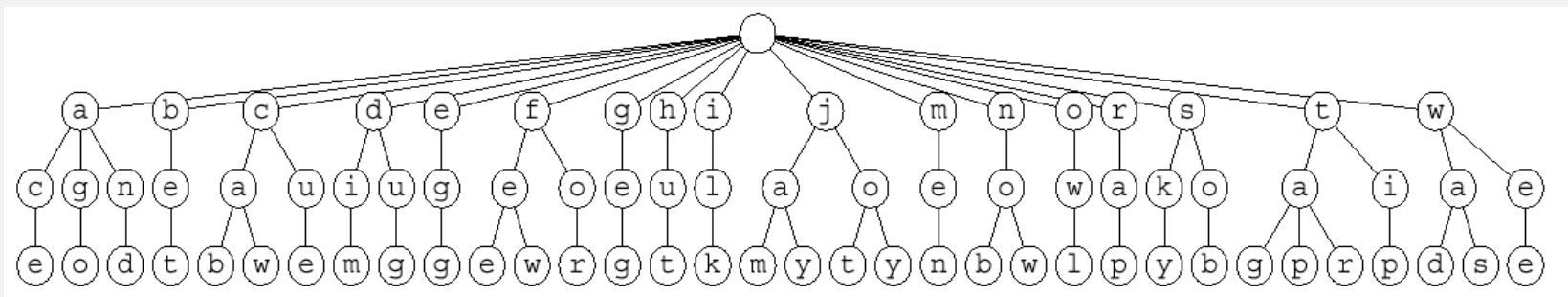
Bottom line. Fast search hit and even faster search miss, but wastes space.

Popular interview question

Goal. Design a data structure to perform efficient spell checking.

Solution. Build a 26-way trie (key = word, value = bit).

English-language text



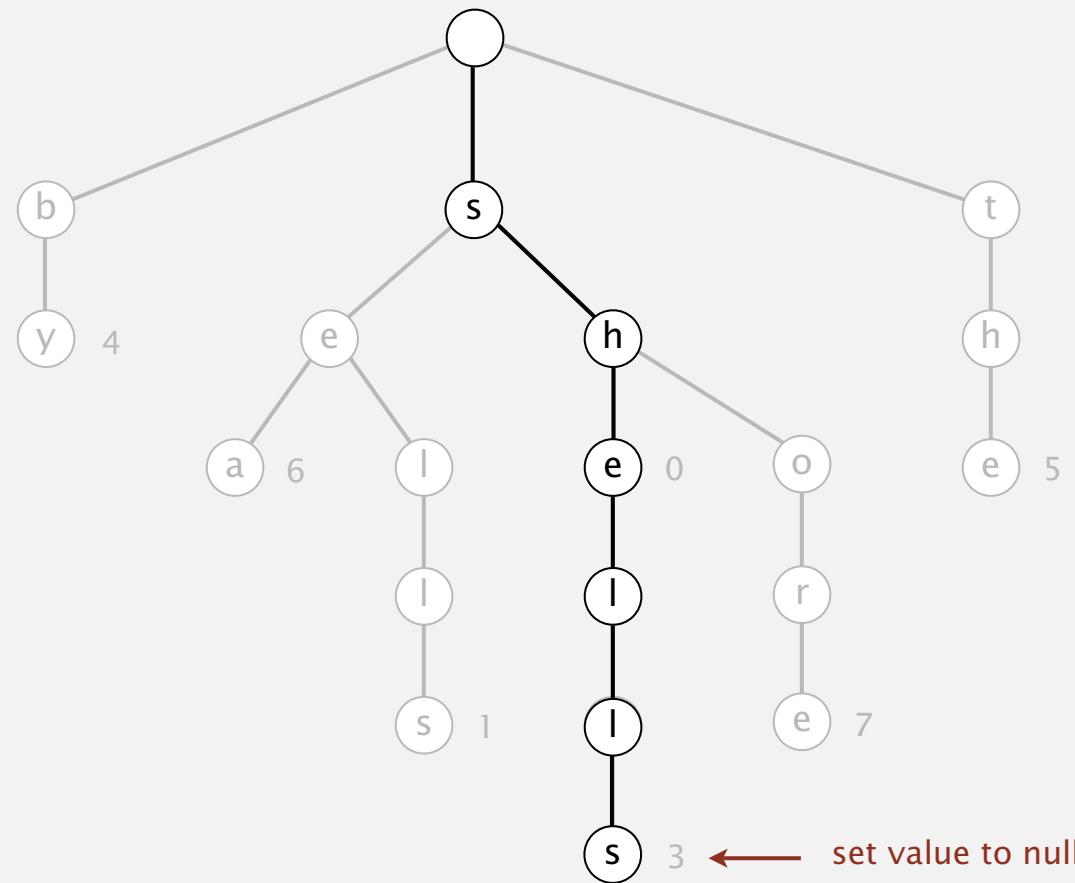
ace
ago
and
bet
cab
caw
cue
dim
dug
egg
fee
few
for
gig
hut
ilk
jam
jay
jot
joy
men
nob
now
owl
rap
sky
sob
tag
tap
tar
tip
wad
was
wee

Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

`delete("shells")`

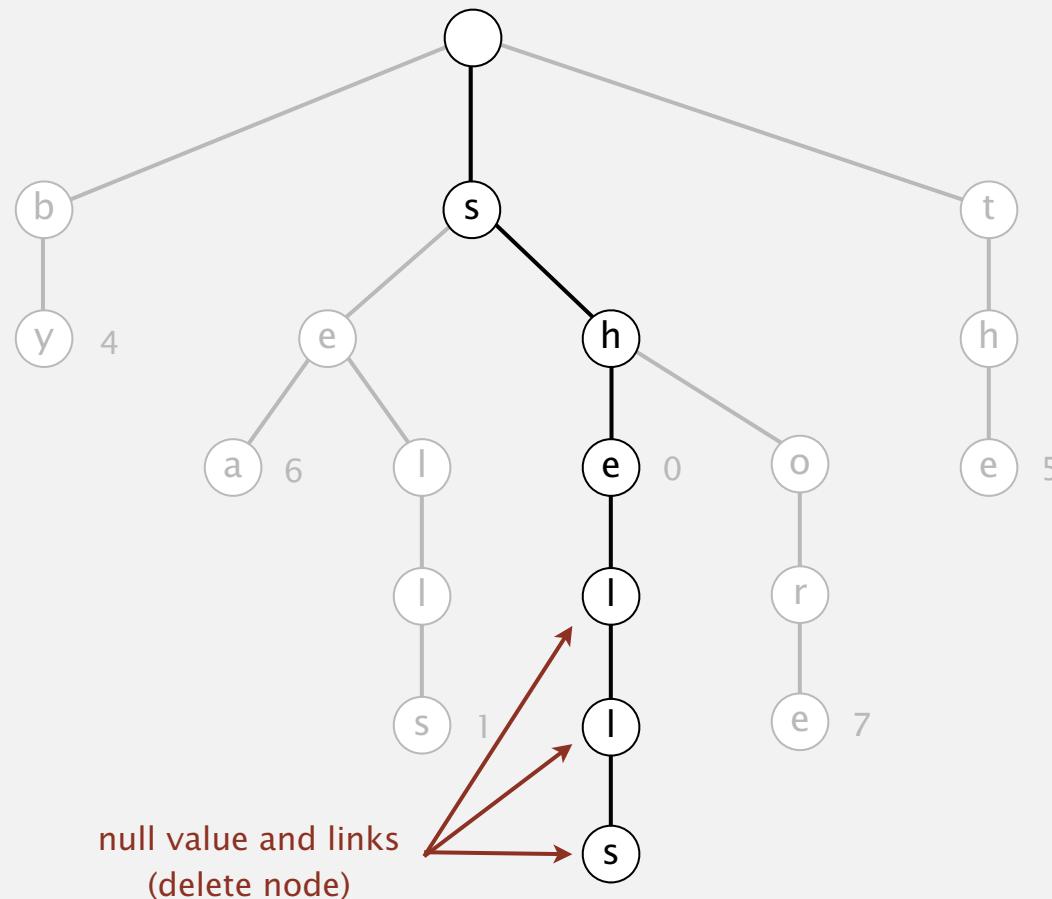


Deletion in an R-way trie

To delete a key-value pair:

- Find the node corresponding to key and set value to null.
- If node has null value and all null links, remove that node (and recur).

delete("shells")



String symbol table implementations cost summary

implementation	character accesses (typical case)					dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt	
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4N$	1.40	97.4	
hashing (linear probing)	L	L	L	$4N$ to $16N$	0.76	40.6	
R-way trie	L	$\log_R N$	L	$(R+1) N$	1.12	out of memory	

R-way trie.

- Method of choice for small R .
- Too much memory for large R .

Challenge. Use less memory, e.g., 65,536-way trie for Unicode!

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.2 TRIES

- ▶ *R-way tries*
- ▶ *ternary search tries*
- ▶ *character-based operations*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.2 TRIES

- ▶ *R-way tries*
- ▶ *ternary search tries*
- ▶ *character-based operations*

Ternary search tries

- Store characters and values in nodes (not keys).
- Each node has 3 children: smaller (left), equal (middle), larger (right).

Fast Algorithms for Sorting and Searching Strings

Jon L. Bentley* Robert Sedgewick#

Abstract

We present theoretical algorithms for sorting and searching multikey data, and derive from them practical C implementations for applications in which keys are character strings. The sorting algorithm blends Quicksort and radix sort; it is competitive with the best known C sort codes. The searching algorithm blends tries and binary search trees; it is faster than hashing and other commonly used search methods. The basic ideas behind the algo-

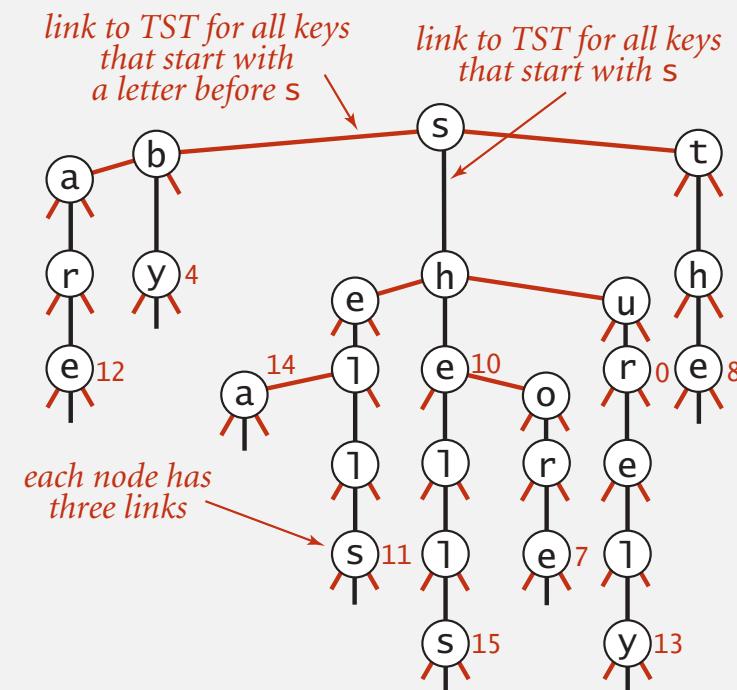
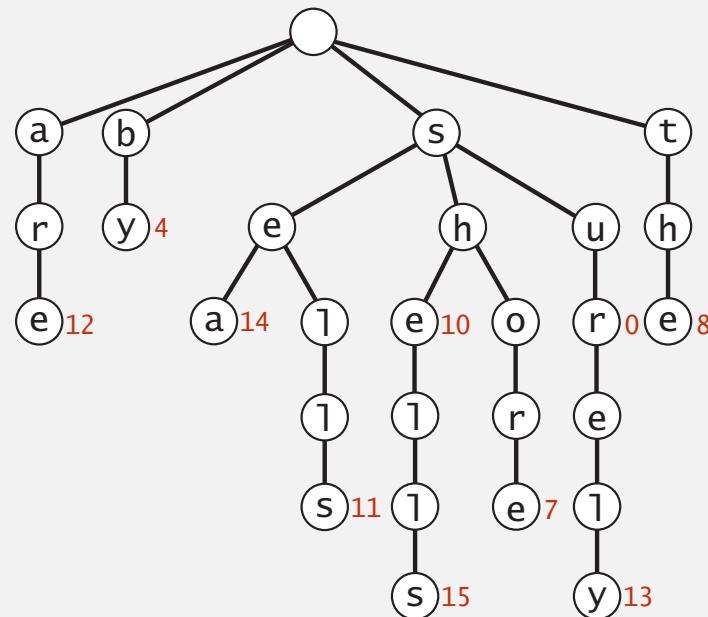
that is competitive with the most efficient string sorting programs known. The second program is a symbol table implementation that is faster than hashing, which is commonly regarded as the fastest symbol table implementation. The symbol table implementation is much more space-efficient than multiway trees, and supports more advanced searches.

In many application programs, sorts use a Quicksort implementation based on an abstract compare operation,



Ternary search tries

- Store characters and values in nodes (not keys).
- Each node has 3 children: smaller (left), equal (middle), larger (right).



TST representation of a trie

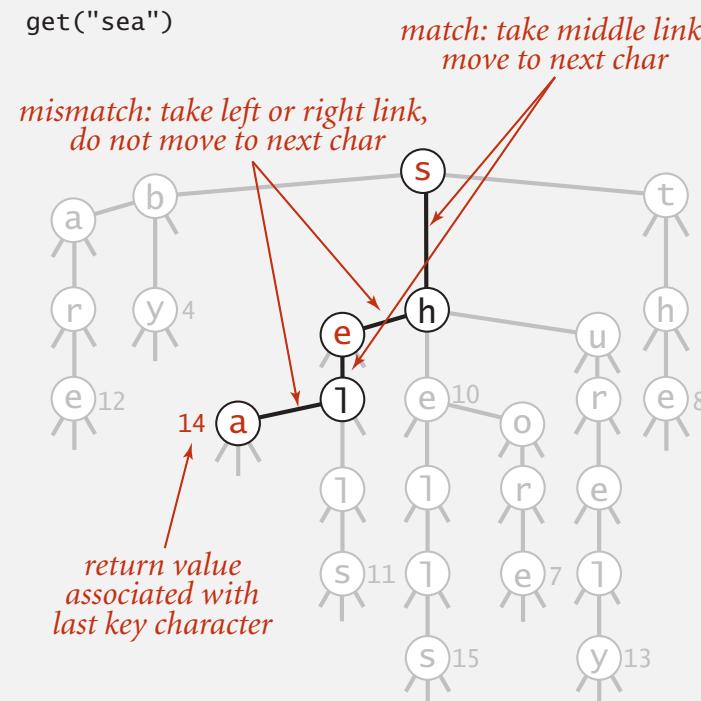
Search in a TST

Follow links corresponding to each character in the key.

- If less, take left link; if greater, take right link.
- If equal, take the middle link and move to the next key character.

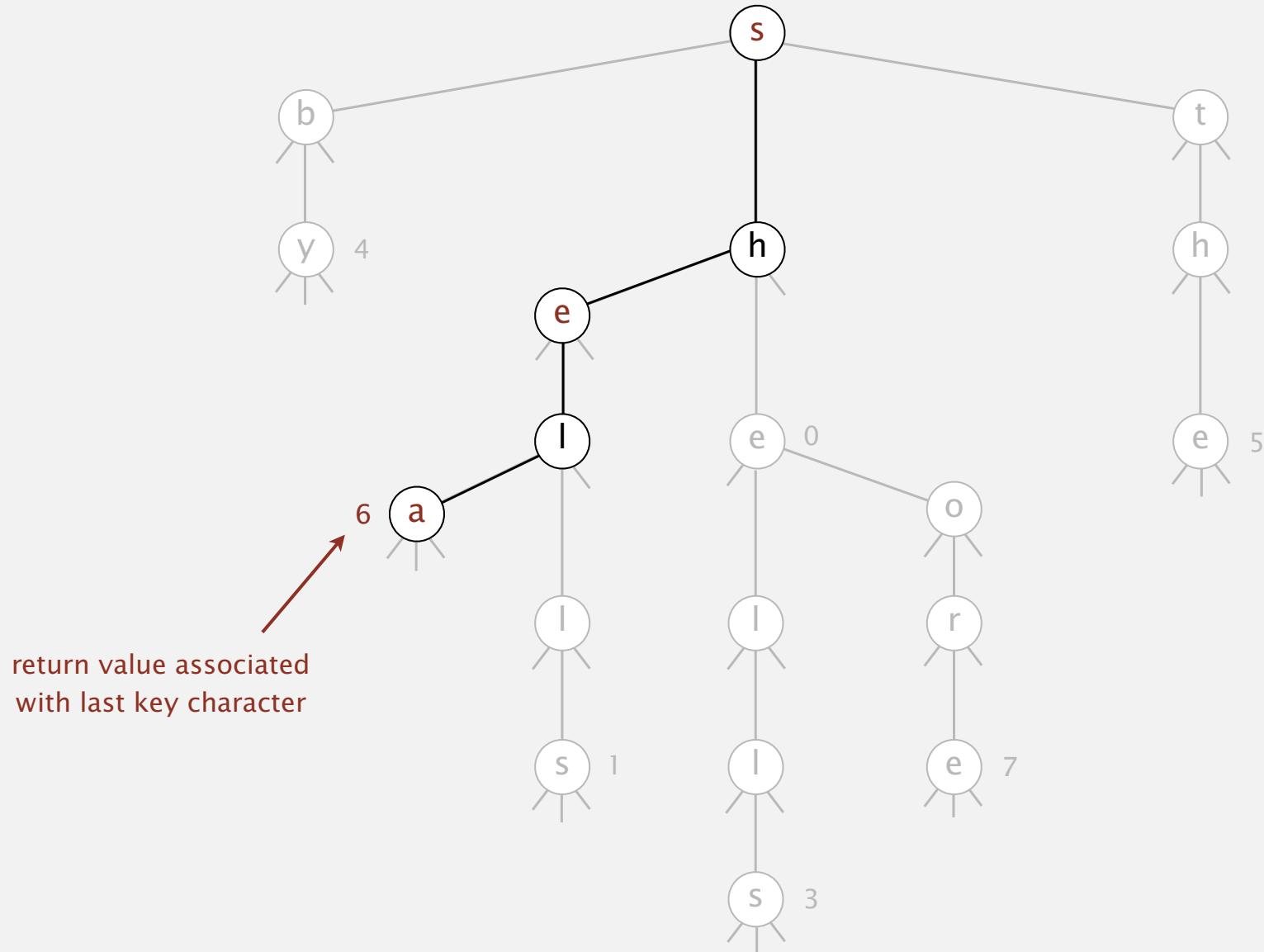
Search hit. Node where search ends has a non-null value.

Search miss. Reach a null link or node where search ends has null value.



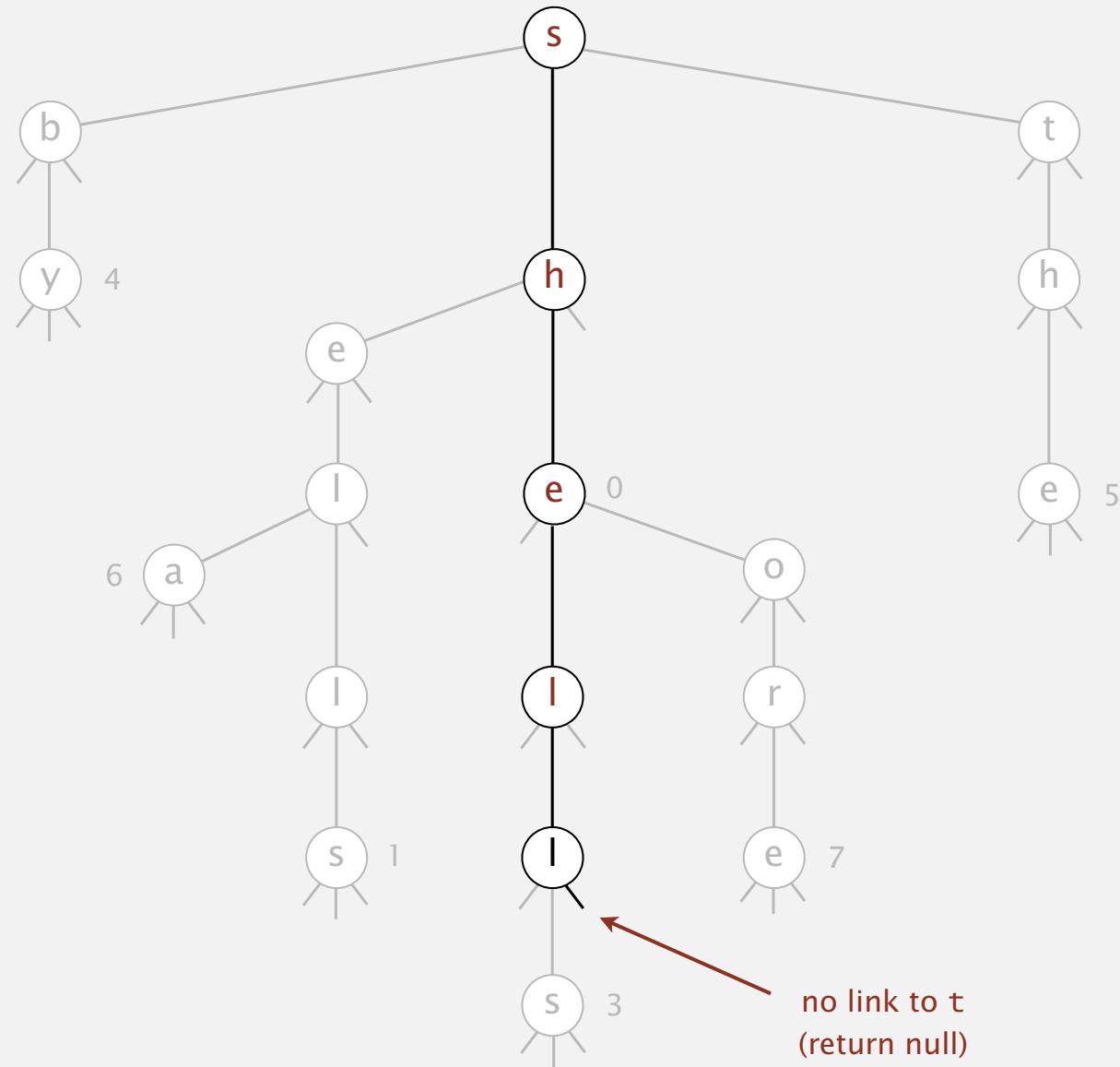
Search hit in a TST

get("sea")



Search miss in a TST

get("shelter")



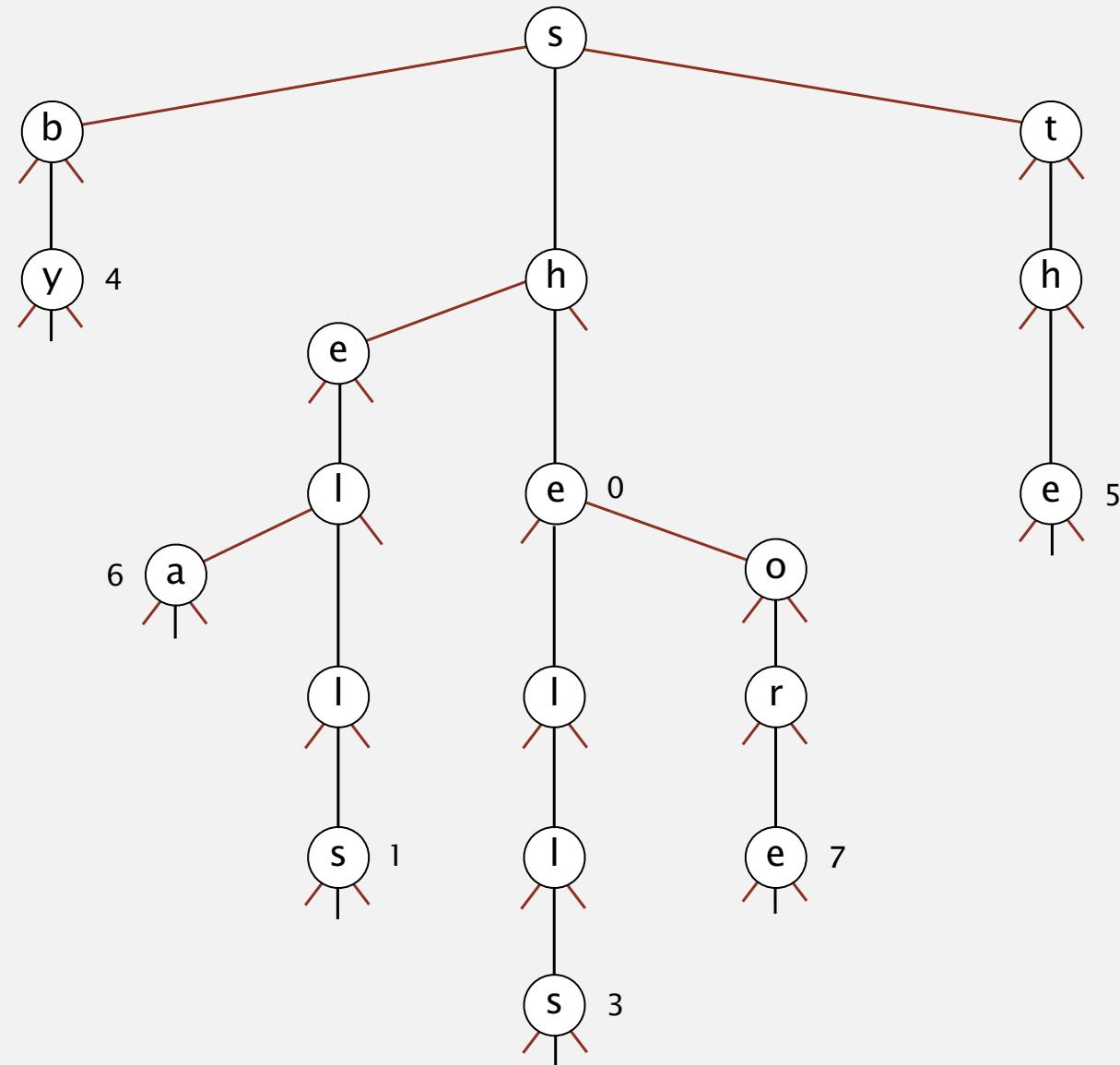
Ternary search trie construction demo

ternary search trie



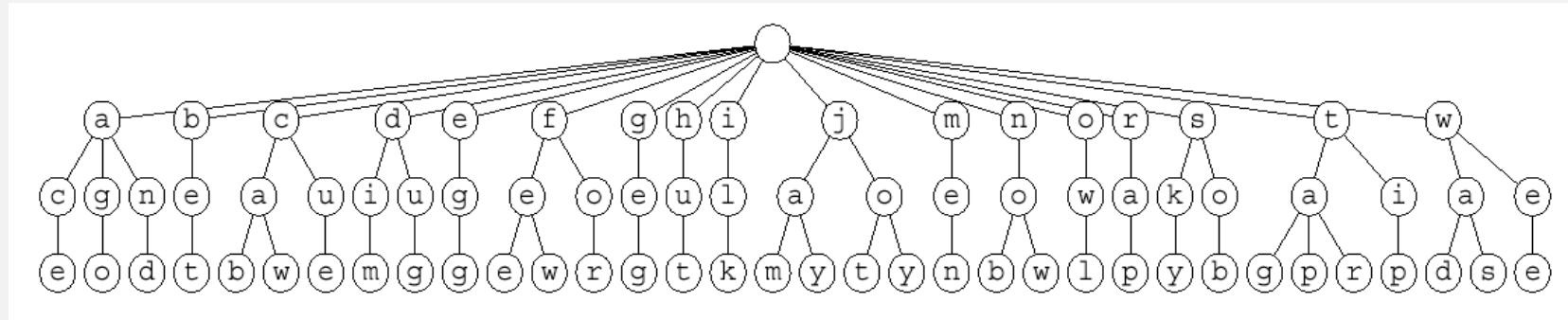
Ternary search trie construction demo

ternary search trie



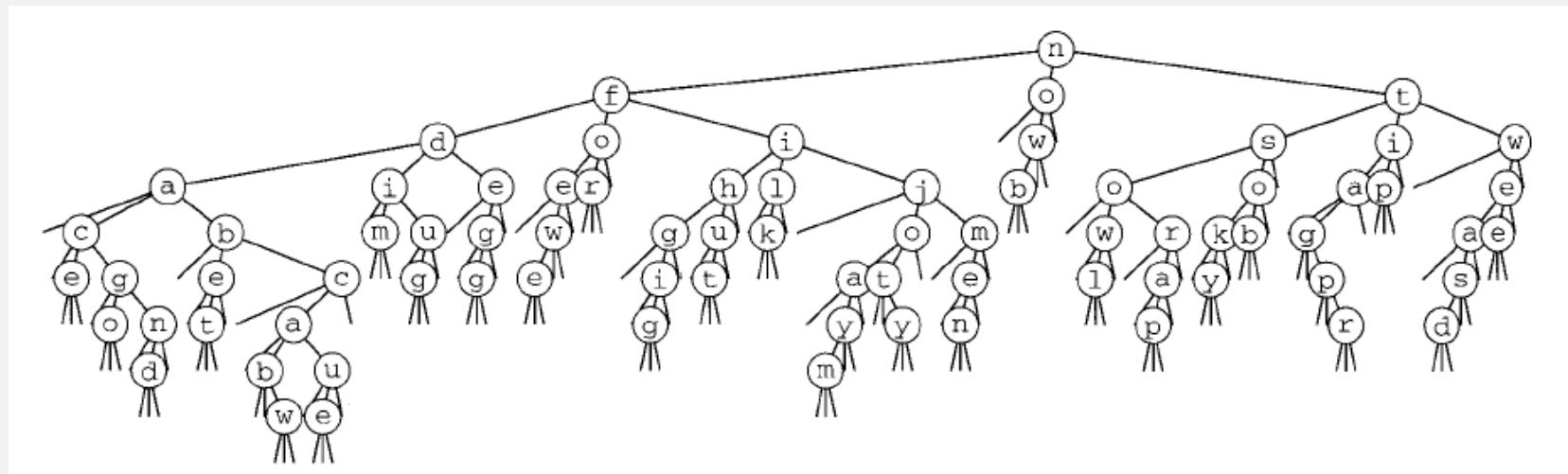
26-way trie vs. TST

26-way trie. 26 null links in each leaf.



26-way trie (1035 null links, not shown)

TST. 3 null links in each leaf.



TST (155 null links)

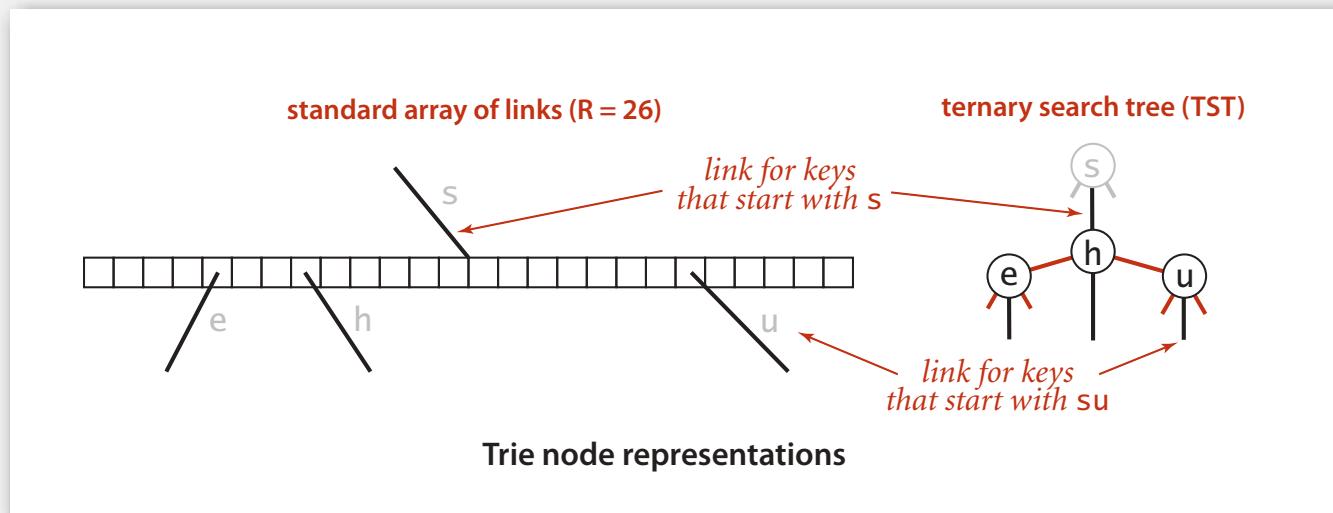
now
for
tip
ilk
dim
tag
jot
sob
nob
sky
hut
ace
bet
men
egg
few
jay
owl
joy
rap
gig
wee
was
cab
wad
caw
cue
fee
tap
ago
tar
jam
dug
and

TST representation in Java

A TST node is five fields:

- A value.
- A character c .
- A reference to a left TST.
- A reference to a middle TST.
- A reference to a right TST.

```
private class Node
{
    private Value val;
    private char c;
    private Node left, mid, right;
}
```



TST: Java implementation

```
public class TST<Value>
{
    private Node root;

    private class Node
    { /* see previous slide */ }

    public void put(String key, Value val)
    { root = put(root, key, val, 0); }

    private Node put(Node x, String key, Value val, int d)
    {
        char c = key.charAt(d);
        if (x == null) { x = new Node(); x.c = c; }
        if (c < x.c)             x.left  = put(x.left,  key, val, d);
        else if (c > x.c)       x.right = put(x.right, key, val, d);
        else if (d < key.length() - 1) x.mid   = put(x.mid,   key, val, d+1);
        else                      x.val   = val;
        return x;
    }

    ...
}
```

TST: Java implementation (continued)

```
:
public boolean contains(String key)
{  return get(key) != null;  }

public Value get(String key)
{
    Node x = get(root, key, 0);
    if (x == null) return null;
    return x.val;
}

private Node get(Node x, String key, int d)
{
    if (x == null) return null;
    char c = key.charAt(d);
    if      (c < x.c)                  return get(x.left,  key, d);
    else if (c > x.c)                  return get(x.right, key, d);
    else if (d < key.length() - 1)    return get(x.mid,   key, d+1);
    else                                return x;
}
```

String symbol table implementation cost summary

implementation	character accesses (typical case)					dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt	
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4 N$	1.40	97.4	
hashing (linear probing)	L	L	L	$4 N$ to $16 N$	0.76	40.6	
R-way trie	L	$\log_R N$	L	$(R + 1) N$	1.12	out of memory	
TST	$L + \ln N$	$\ln N$	$L + \ln N$	4 N	0.72	38.7	

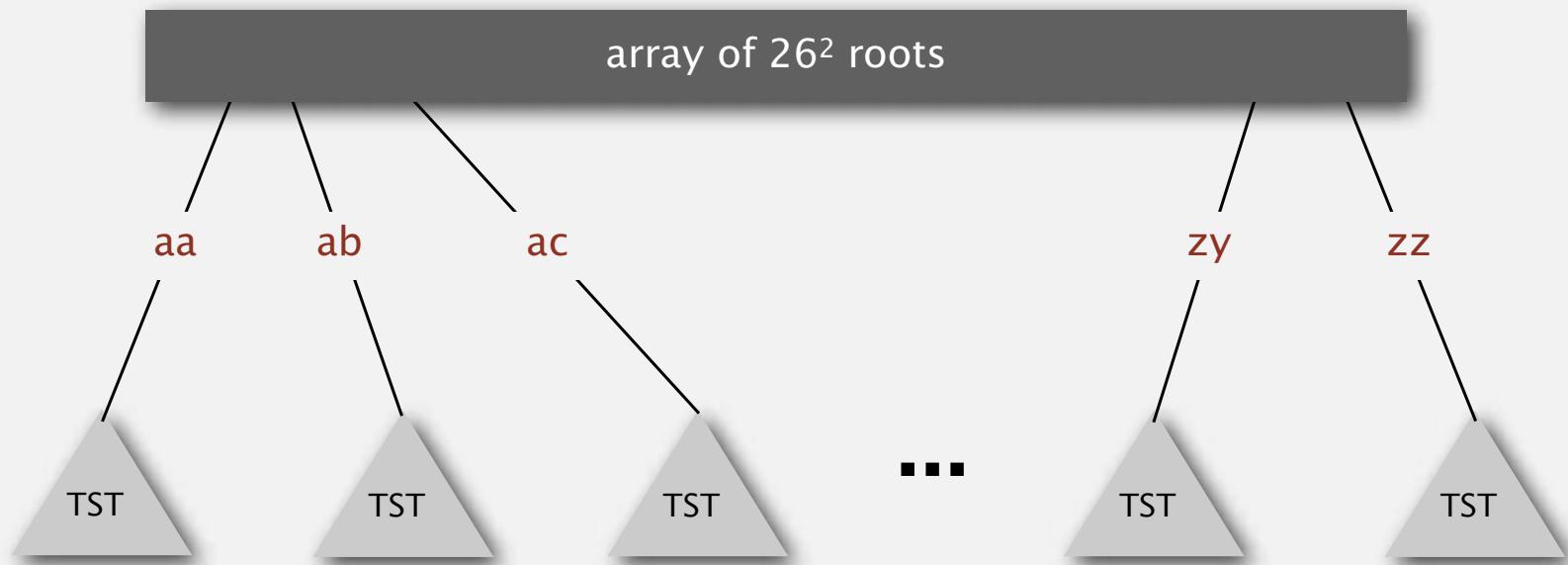
Remark. Can build balanced TSTs via rotations to achieve $L + \log N$ worst-case guarantees.

Bottom line. TST is as fast as hashing (for string keys), space efficient.

TST with R^2 branching at root

Hybrid of R-way trie and TST.

- Do R^2 -way branching at root.
- Each of R^2 root nodes points to a TST.



Q. What about one- and two-letter words?

String symbol table implementation cost summary

implementation	character accesses (typical case)					dedup	
	search hit	search miss	insert	space (references)	moby.txt	actors.txt	
red-black BST	$L + c \lg^2 N$	$c \lg^2 N$	$c \lg^2 N$	$4 N$	1.40	97.4	
hashing (linear probing)	L	L	L	$4 N$ to $16 N$	0.76	40.6	
R-way trie	L	$\log_R N$	L	$(R + 1) N$	1.12	out of memory	
TST	$L + \ln N$	$\ln N$	$L + \ln N$	$4 N$	0.72	38.7	
TST with R^2	$L + \ln N$	$\ln N$	$L + \ln N$	$4 N + R^2$	0.51	32.7	

Bottom line. Faster than hashing for our benchmark client.

TST vs. hashing

Hashing.

- Need to examine entire key.
- Search hits and misses cost about the same.
- Performance relies on hash function.
- Does not support ordered symbol table operations.

TSTs.

- Works only for strings (or digital keys).
- Only examines just enough key characters.
- Search miss may involve only a few characters.
- Supports ordered symbol table operations (plus others!).

Bottom line. TSTs are:

- Faster than hashing (especially for search misses).
- More flexible than red-black BSTs. [stay tuned]

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.2 TRIES

- ▶ *R-way tries*
- ▶ *ternary search tries*
- ▶ *character-based operations*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.2 TRIES

- ▶ *R-way tries*
- ▶ *ternary search tries*
- ▶ *character-based operations*

String symbol table API

Character-based operations. The string symbol table API supports several useful character-based operations.

key	value
by	4
sea	6
sells	1
she	0
shells	3
shore	7
the	5

Prefix match. Keys with prefix sh: she, shells, and shore.

Wildcard match. Keys that match .he: she and the.

Longest prefix. Key that is the longest prefix of shellsort: shells.

String symbol table API

```
public class StringST<Value>
```

```
    StringST()
```

create a symbol table with string keys

```
    void put(String key, Value val)
```

put key-value pair into the symbol table

```
    Value get(String key)
```

value paired with key

```
    void delete(String key)
```

delete key and corresponding value

```
    :
```

```
Iterable<String> keys()
```

all keys

```
Iterable<String> keysWithPrefix(String s)
```

keys having s as a prefix

```
Iterable<String> keysThatMatch(String s)
```

keys that match s (where . is a wildcard)

```
    String longestPrefixOf(String s)
```

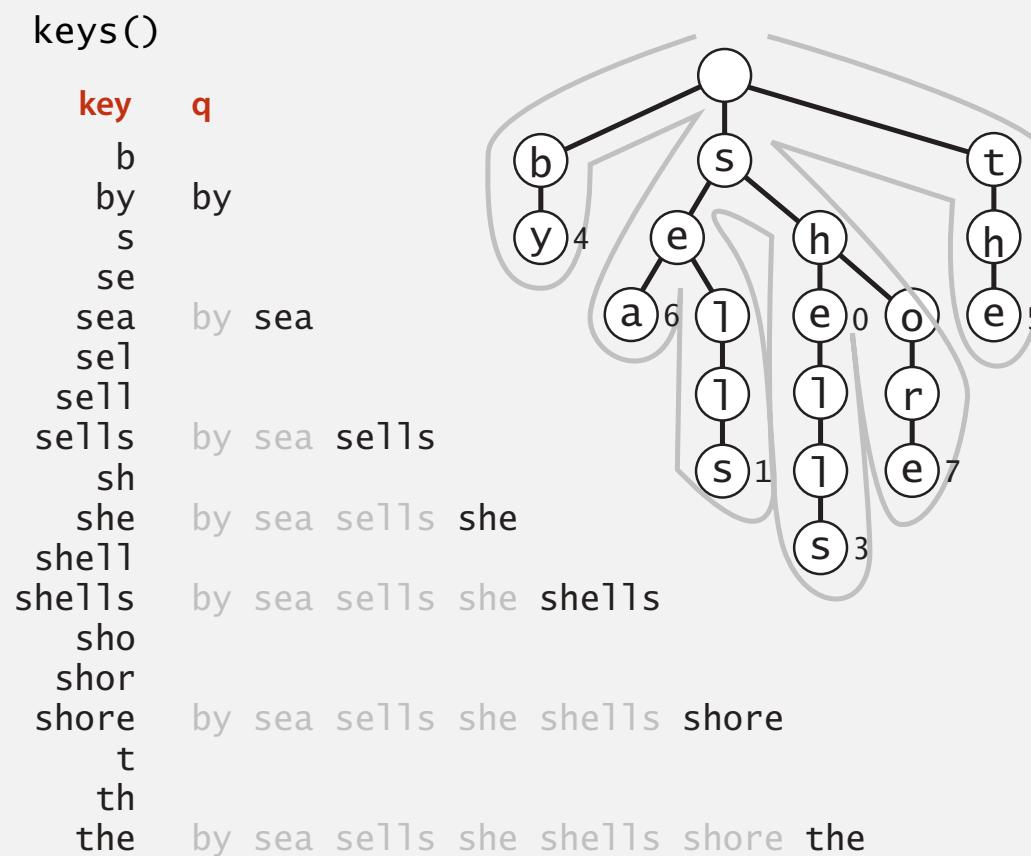
longest key that is a prefix of s

Remark. Can also add other ordered ST methods, e.g., floor() and rank().

Warmup: ordered iteration

To iterate through all keys in sorted order:

- Do inorder traversal of trie; add keys encountered to a queue.
- Maintain sequence of characters on path from root to node.



Ordered iteration: Java implementation

To iterate through all keys in sorted order:

- Do inorder traversal of trie; add keys encountered to a queue.
- Maintain sequence of characters on path from root to node.

```
public Iterable<String> keys()
{
    Queue<String> queue = new Queue<String>();
    collect(root, "", queue);
    return queue;
}

private void collect(Node x, String prefix, Queue<String> q)
{
    if (x == null) return;
    if (x.val != null) q.enqueue(prefix);
    for (char c = 0; c < R; c++)
        collect(x.next[c], prefix + c, q);
}
```

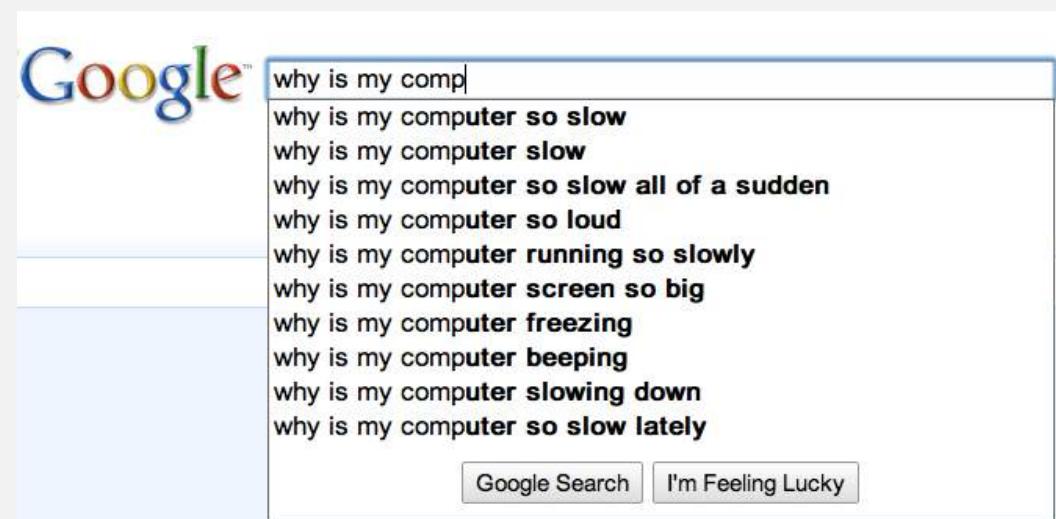
sequence of characters
on path from root to x

Prefix matches

Find all keys in a symbol table starting with a given prefix.

Ex. Autocomplete in a cell phone, search bar, text editor, or shell.

- User types characters one at a time.
- System reports all matching strings.



why is my comp

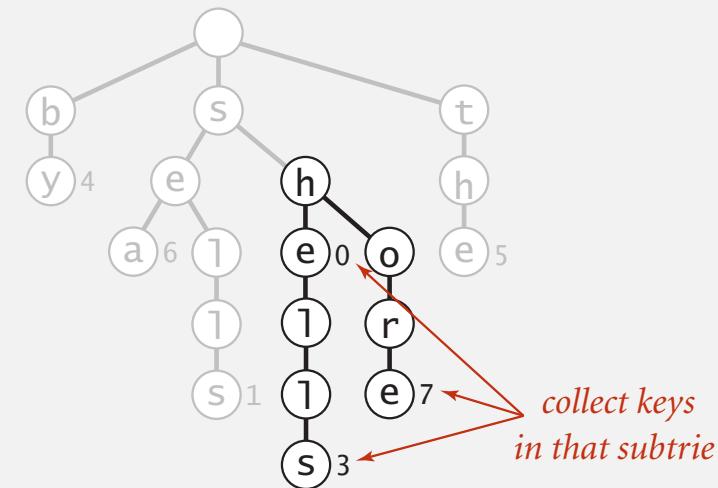
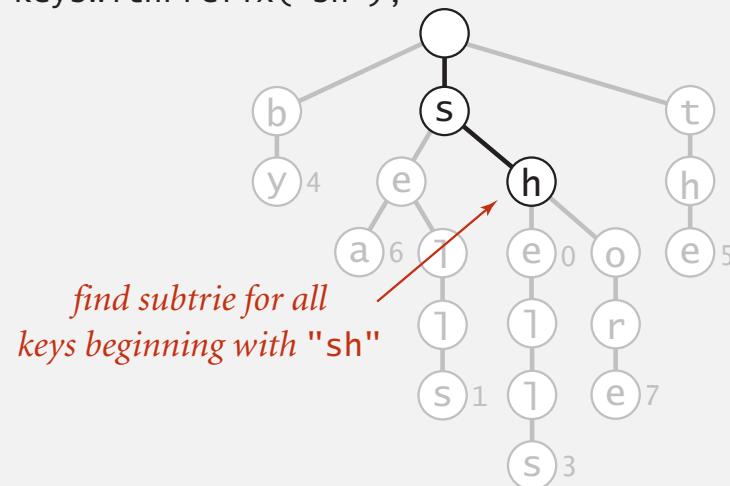
- why is my computer so slow
- why is my computer slow
- why is my computer so slow all of a sudden
- why is my computer so loud
- why is my computer running so slowly
- why is my computer screen so big
- why is my computer freezing
- why is my computer beeping
- why is my computer slowing down
- why is my computer so slow lately

Google Search I'm Feeling Lucky

Prefix matches in an R-way trie

Find all keys in a symbol table starting with a given prefix.

keysWithPrefix("sh");



```
public Iterable<String> keysWithPrefix(String prefix)
{
    Queue<String> queue = new Queue<String>();
    Node x = get(root, prefix, 0);
    collect(x, prefix, queue);
    return queue;
}
```

root of subtrie for all strings beginning with given prefix

key	queue
sh	she
she	
shel	
shell	
shells	she shells
sho	
shor	
shore	she shells shore

Longest prefix

Find longest key in symbol table that is a prefix of query string.

Ex. To send packet toward destination IP address, router chooses IP address in routing table that is longest prefix match.

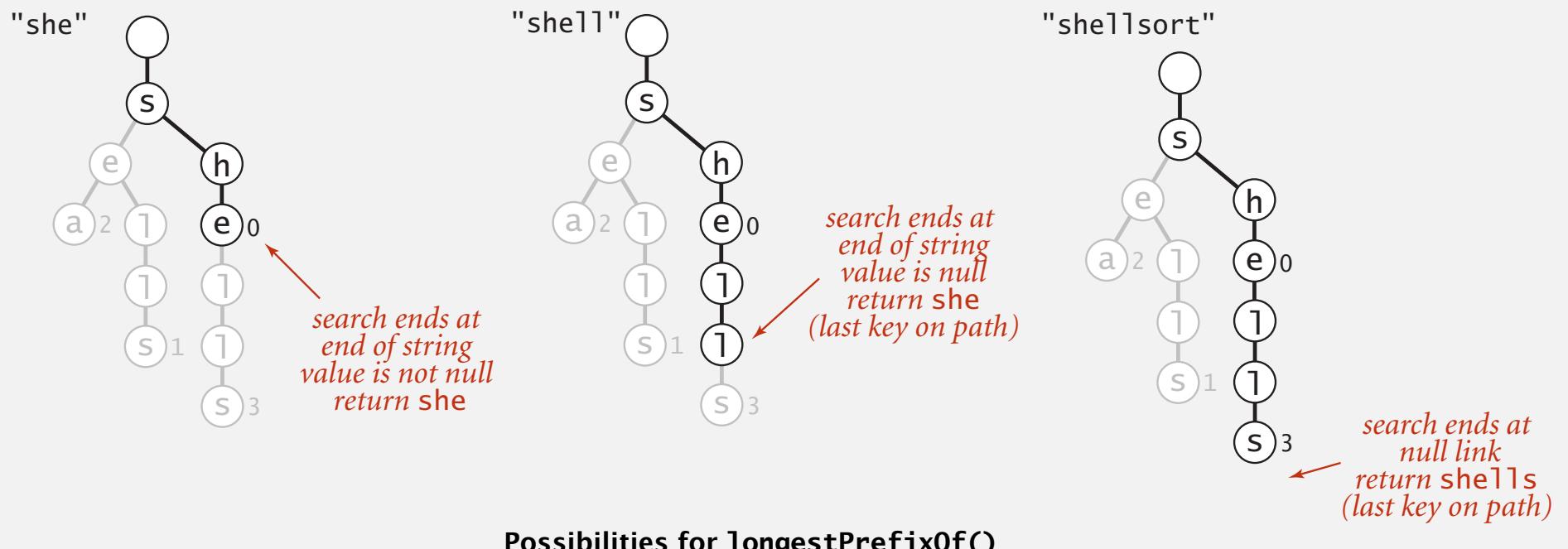
"128"	represented as 32-bit binary number for IPv4 (instead of string)
"128.112"	
"128.112.055"	
"128.112.055.15"	
"128.112.136"	<code>longestPrefixOf("128.112.136.11") = "128.112.136"</code>
"128.112.155.11"	<code>longestPrefixOf("128.112.100.16") = "128.112"</code>
"128.112.155.13"	<code>longestPrefixOf("128.166.123.45") = "128"</code>
"128.222"	
"128.222.136"	

Note. Not the same as floor: `floor("128.112.100.16") = "128.112.055.15"`

Longest prefix in an R-way trie

Find longest key in symbol table that is a prefix of query string.

- Search for query string.
- Keep track of longest key encountered.



Longest prefix in an R-way trie: Java implementation

Find longest key in symbol table that is a prefix of query string.

- Search for query string.
- Keep track of longest key encountered.

```
public String longestPrefixOf(String query)
{
    int length = search(root, query, 0, 0);
    return query.substring(0, length);
}

private int search(Node x, String query, int d, int length)
{
    if (x == null) return length;
    if (x.val != null) length = d;
    if (d == query.length()) return length;
    char c = query.charAt(d);
    return search(x.next[c], query, d+1, length);
}
```

T9 texting

Goal. Type text messages on a phone keypad.

Multi-tap input. Enter a letter by repeatedly pressing a key until the desired letter appears.

T9 text input.

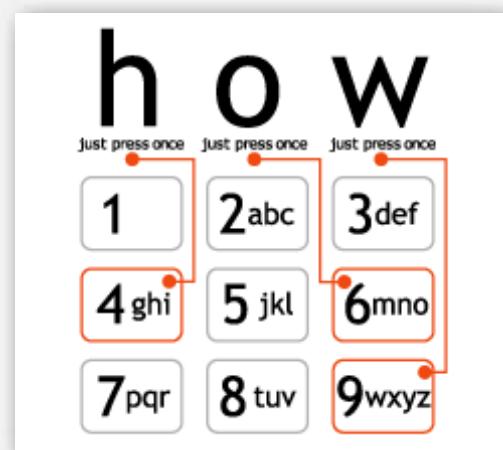
"a much faster and more fun way to enter text"



- Find all words that correspond to given sequence of numbers.
- Press 0 to see all completion options.

Ex. hello

- Multi-tap: 4 4 3 3 5 5 5 5 5 6 6 6
- T9: 4 3 5 5 6



www.t9.com

Q. How to implement?

A letter to t9.com

To: info@t9support.com

Date: Tue, 25 Oct 2005 14:27:21 -0400 (EDT)

Dear T9 texting folks,

I enjoyed learning about the T9 text system from your webpage, and used it as an example in my data structures and algorithms class. However, one of my students noticed a bug in your phone keypad

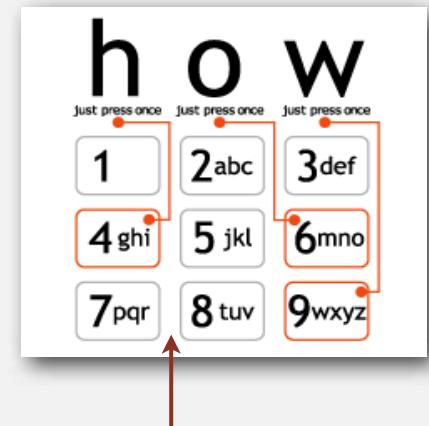
<http://www.t9.com/images/how.gif>

Somehow, it is missing the letter s. (!)

Just wanted to bring this information to your attention and thank you for your website.

Regards,

Kevin



where the @#\$% is the "s" ???

A world without 's' ?

To: "'Kevin Wayne'" <wayne@CS.Princeton.EDU>

Date: Tue, 25 Oct 2005 12:44:42 -0700

Thank you Kevin.

I am glad that you find T9 o valuable for your cla. I had not noticed thi before. Thank for writing in and letting u know.

Take care,

Brooke nyder
OEM Dev upport
AOL/Tegic Communication
1000 Dexter Ave N. uite 300
eattle, WA 98109

ALL INFORMATION CONTAINED IN THIS EMAIL IS CONSIDERED
CONFIDENTIAL AND PROPERTY OF AOL/TEGIC COMMUNICATIONS

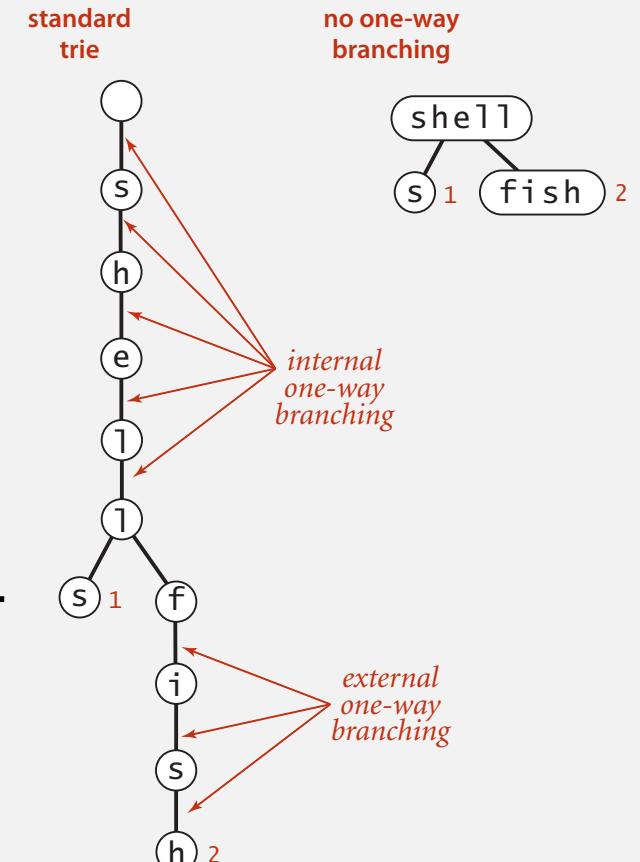
Patricia trie

Patricia trie. [Practical Algorithm to Retrieve Information Coded in Alphanumeric]

- Remove one-way branching.
- Each node represents a sequence of characters.
- Implementation: one step beyond this course.

```
put("shells", 1);
put("shellfish", 2);
```

standard
trie



Applications.

- Database search.
- P2P network search.
- IP routing tables: find longest prefix match.
- Compressed quad-tree for N-body simulation.
- Efficiently storing and querying XML documents.

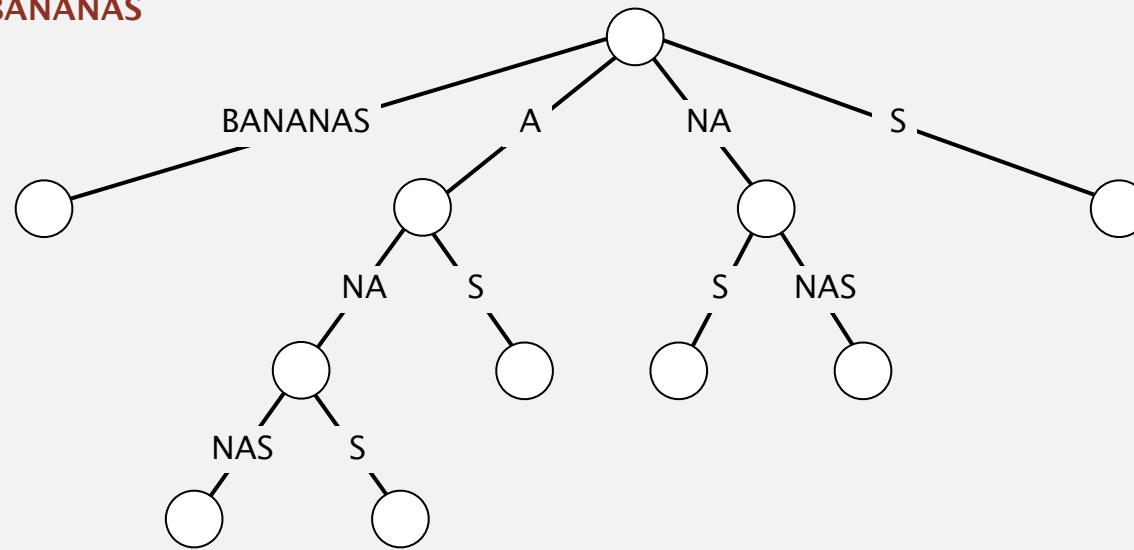
Also known as: crit-bit tree, radix tree.

Suffix tree

Suffix tree.

- Patricia trie of suffixes of a string.
- Linear-time construction: beyond this course.

suffix tree for BANANAS



Applications.

- Linear-time: longest repeated substring, longest common substring, longest palindromic substring, substring search, tandem repeats,
- Computational biology databases (BLAST, FASTA).

String symbol tables summary

A success story in algorithm design and analysis.

Red-black BST.

- Performance guarantee: $\log N$ key compares.
- Supports ordered symbol table API.

Hash tables.

- Performance guarantee: constant number of probes.
- Requires good hash function for key type.

Tries. R-way, TST.

- Performance guarantee: $\log N$ **characters** accessed.
- Supports character-based operations.

Bottom line. You can get at anything by examining 50-100 bits (!!?)

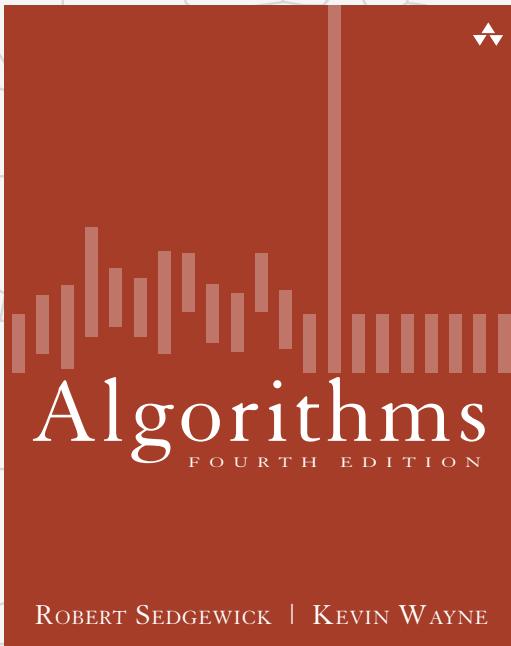
Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.2 TRIES

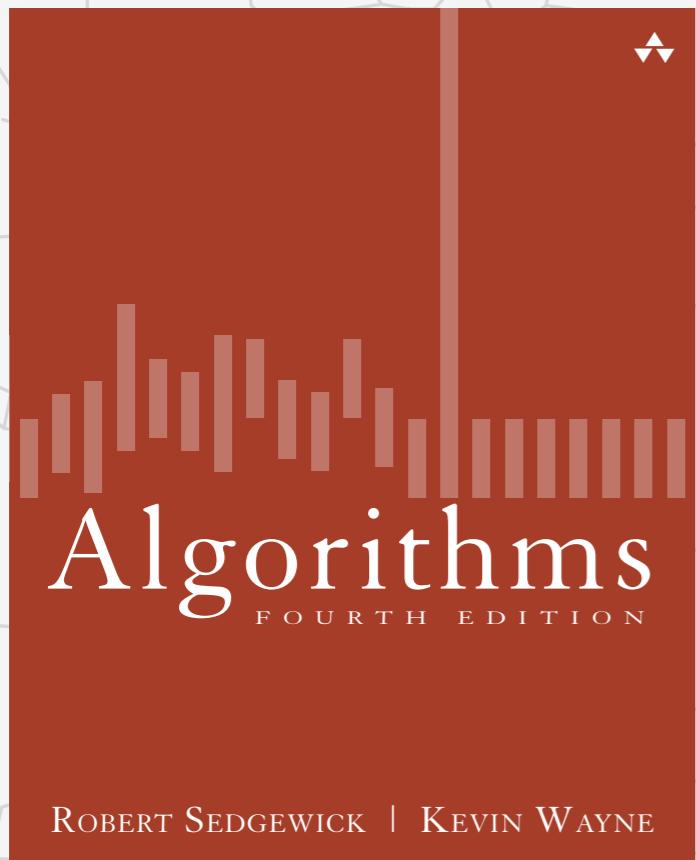
- ▶ *R-way tries*
- ▶ *ternary search tries*
- ▶ *character-based operations*



<http://algs4.cs.princeton.edu>

5.2 TRIES

- ▶ *R-way tries*
- ▶ *ternary search tries*
- ▶ *character-based operations*



ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.1 STRING SORTS

- ▶ *strings in Java*
- ▶ *key-indexed counting*
- ▶ *LSD radix sort*
- ▶ *MSD radix sort*
- ▶ *3-way radix quicksort*
- ▶ *suffix arrays*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.1 STRING SORTS

- ▶ *strings in Java*
- ▶ *key-indexed counting*
- ▶ *LSD radix sort*
- ▶ *MSD radix sort*
- ▶ *3-way radix quicksort*
- ▶ *suffix arrays*

String processing

String. Sequence of characters.

Important fundamental abstraction.

- Information processing.
- Genomic sequences.
- Communication systems (e.g., email).
- Programming systems (e.g., Java programs).
- ...

“The digital information that underlies biochemistry, cell biology, and development can be represented by a simple string of G's, A's, T's and C's. This string is the root data structure of an organism's biology.” — M. V. Olson

The char data type

C **char** data type. Typically an 8-bit integer.

- Supports 7-bit ASCII.
- Can represent only 256 characters.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2	SP	!	"	#	\$	%	&	'	()	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{		}	~	DEL

Hexadecimal to ASCII conversion table

A á ð ö

U+0041 U+00E1 U+2202 U+1D50A

Unicode characters

Java **char** data type. A 16-bit unsigned integer.

- Supports original 16-bit Unicode.
- Supports 21-bit Unicode 3.0 (awkwardly).

I (heart) Unicode



The String data type

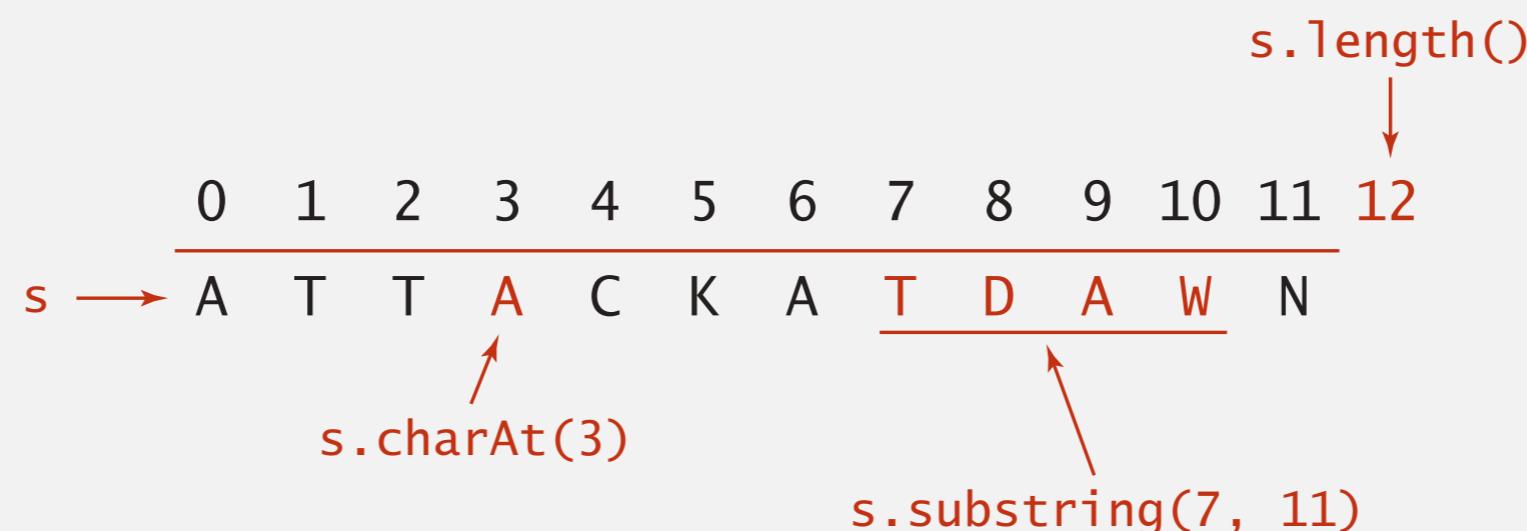
String data type in Java. Sequence of characters (immutable).

Length. Number of characters.

Indexing. Get the i^{th} character.

Substring extraction. Get a contiguous subsequence of characters.

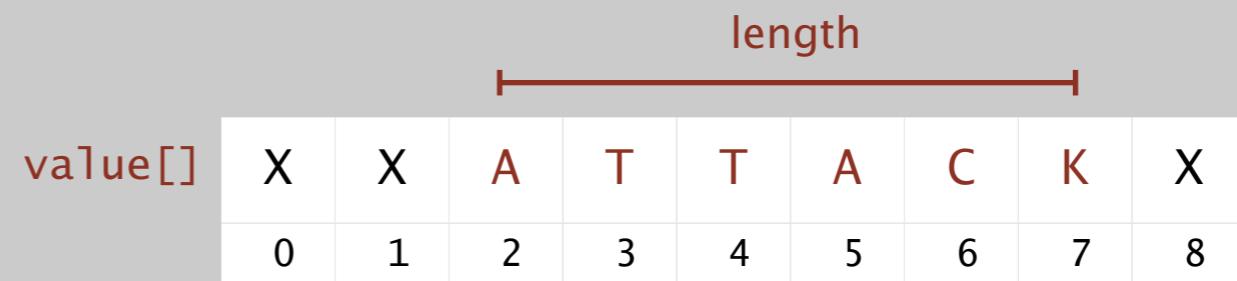
String concatenation. Append one character to end of another string.



The String data type: Java implementation

```
public final class String implements Comparable<String>
{
    private char[] value; // characters
    private int offset; // index of first char in array
    private int length; // length of string
    private int hash; // cache of hashCode()
```

```
public int length()
{ return length; }
```



```
public char charAt(int i)
{ return value[i + offset]; }
```

```
private String(int offset, int length, char[] value)
{
```

```
    this.offset = offset;
    this.length = length;
    this.value = value;
}
```

copy of reference to
original char array

```
public String substring(int from, int to)
{ return new String(offset + from, to - from, value); }
```

...

The String data type: performance

String data type (in Java). Sequence of characters (immutable).

Underlying implementation. Immutable char[] array, offset, and length.

String		
operation	guarantee	extra space
length()	1	1
charAt()	1	1
substring()	1	1
concat()	N	N

Memory. $40 + 2N$ bytes for a virgin String of length N .

can use byte[] or char[] instead of String to save space
(but lose convenience of String data type)

The StringBuilder data type

StringBuilder data type. Sequence of characters (mutable).

Underlying implementation. Resizing char[] array and length.

	String		StringBuilder	
operation	guarantee	extra space	guarantee	extra space
length()	1	1	1	1
charAt()	1	1	1	1
substring()	1	1	N	N
concat()	N	N	1 *	1 *

* amortized

Remark. StringBuffer data type is similar, but thread safe (and slower).

String vs. StringBuilder

Q. How to efficiently reverse a string?

A.

```
public static String reverse(String s)
{
    String rev = "";
    for (int i = s.length() - 1; i >= 0; i--)
        rev += s.charAt(i);
    return rev;
}
```

make a copy nên sẽ là $O(N^2)$

← quadratic time

B.

```
public static String reverse(String s)
{
    StringBuilder rev = new StringBuilder();
    for (int i = s.length() - 1; i >= 0; i--)
        rev.append(s.charAt(i));
    return rev.toString();
}
```

$O(N)$

← linear time

String challenge: array of suffixes

Q. How to efficiently form array of suffixes?

input string

a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14

suffixes

0	a	a	c	a	a	g	t	t	t	a	c	a	a	g	c
1	a	c	a	a	g	t	t	t	a	c	a	a	g	c	
2	c	a	a	g	t	t	t	a	c	a	a	g	c		
3	a	a	g	t	t	t	a	c	a	a	g	c			
4	a	g	t	t	t	a	c	a	a	g	c				
5	g	t	t	t	a	c	a	a	g	c					
6	t	t	t	a	c	a	a	g	c						
7	t	t	a	c	a	a	g	c							
8	t	a	c	a	a	g	c								
9	a	c	a	a	g	c									
10	c	a	a	g	c										
11	a	a	g	c											
12	a	g	c												
13	g	c													
14	c														

String vs. StringBuilder

Q. How to efficiently form array of suffixes?

A.

```
public static String[] suffixes(String s)
{
    int N = s.length();
    String[] suffixes = new String[N];
    for (int i = 0; i < N; i++)
        suffixes[i] = s.substring(i, N);
    return suffixes;
}
```

↑↑

linear time and
linear space

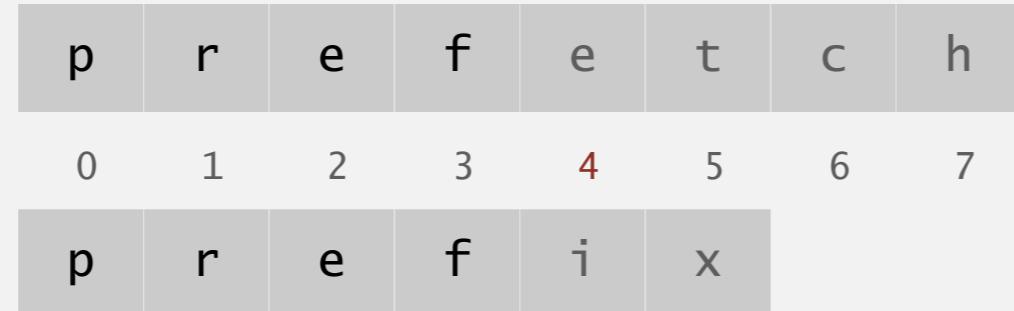
B.

```
public static String[] suffixes(String s)
{
    int N = s.length();
    StringBuilder sb = new StringBuilder(s);
    String[] suffixes = new String[N];
    for (int i = 0; i < N; i++)
        suffixes[i] = sb.substring(i, N);
    return suffixes;
}
```

quadratic time and
quadratic space

Longest common prefix

Q. How long to compute length of longest common prefix?



```
public static int lcp(String s, String t)
{
    int N = Math.min(s.length(), t.length());
    for (int i = 0; i < N; i++)
        if (s.charAt(i) != t.charAt(i))
            return i;
    return N;
}
```

linear time (worst case)
sublinear time (typical case)

Running time. Proportional to length D of longest common prefix.

Remark. Also can compute compareTo() in sublinear time.

Alphabets

Digital key. Sequence of digits over fixed alphabet.

Radix. Number of digits R in alphabet.

name	$R()$	$\lg R()$	characters
BINARY	2	1	01
OCTAL	8	3	01234567
DECIMAL	10	4	0123456789
HEXADECIMAL	16	4	0123456789ABCDEF
DNA	4	2	ACTG
LOWERCASE	26	5	abcdefghijklmnopqrstuvwxyz
UPPERCASE	26	5	ABCDEFGHIJKLMNOPQRSTUVWXYZ
PROTEIN	20	5	ACDEFGHIKLMNPQRSTVWY
BASE64	64	6	ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/-
ASCII	128	7	<i>ASCII characters</i>
EXTENDED_ASCII	256	8	<i>extended ASCII characters</i>
UNICODE16	65536	16	<i>Unicode characters</i>

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.1 STRING SORTS

- ▶ *strings in Java*
- ▶ *key-indexed counting*
- ▶ *LSD radix sort*
- ▶ *MSD radix sort*
- ▶ *3-way radix quicksort*
- ▶ *suffix arrays*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.1 STRING SORTS

- ▶ *strings in Java*
- ▶ ***key-indexed counting***
- ▶ *LSD radix sort*
- ▶ *MSD radix sort*
- ▶ *3-way radix quicksort*
- ▶ *suffix arrays*

Review: summary of the performance of sorting algorithms

Frequency of operations = key compares.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	yes	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	yes	compareTo()
quicksort	$1.39 N \lg N^*$	$1.39 N \lg N$	$c \lg N$	no	compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1	no	compareTo()

* probabilistic

Lower bound. $\sim N \lg N$ compares required by any compare-based algorithm.

- Q. Can we do better (despite the lower bound)?
A. Yes, if we don't depend on key compares.

Key-indexed counting: assumptions about keys

Assumption. Keys are integers between 0 and $R - 1$.

Implication. Can use key as an array index.

Applications.

- Sort string by first letter.
- Sort class roster by section.
- Sort phone numbers by area code.
- Subroutine in a sorting algorithm. [stay tuned]

Remark. Keys may have associated data \Rightarrow
can't just count up number of keys of each value.

input		sorted result (by section)
name	section	
Anderson	2	Harris 1
Brown	3	Martin 1
Davis	3	Moore 1
Garcia	4	Anderson 2
Harris	1	Martinez 2
Jackson	3	Miller 2
Johnson	4	Robinson 2
Jones	3	White 2
Martin	1	Brown 3
Martinez	2	Davis 3
Miller	2	Jackson 3
Moore	1	Jones 3
Robinson	2	Taylor 3
Smith	4	Williams 3
Taylor	3	Garcia 4
Thomas	4	Johnson 4
Thompson	4	Smith 4
White	2	Thomas 4
Williams	3	Thompson 4
Wilson	4	Wilson 4

↑
*keys are
small integers*

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.



$R = 6$

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	
0	d	
1	a	use a for 0
2	c	b for 1
3	f	c for 2
4	f	d for 3
5	b	e for 4
6	d	f for 5
7	b	
8	f	
9	b	
10	e	
11	a	

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

count frequencies → for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	offset by 1 [stay tuned]
0	d	
1	a	
2	c	
3	f	
4	f	
5	b	
6	d	
7	b	
8	f	
9	b	
10	e	
11	a	

a	0
b	2
c	3
d	1
e	2
f	1
-	3

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

compute cumulates → for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	r	count[r]
0	d	a	0
1	a	b	2
2	c	c	5
3	f	d	6
4	f	e	8
5	b	f	9
6	d	-	12
7	d	-	-
8	e	-	-
9	f	-	-
10	b	-	-
11	e	-	-
12	a	-	-

6 keys < d, 8 keys < e
so d's go in a[6] and a[7]

Diagram illustrating the state of arrays a and $count[r]$. Array a contains the sequence: d, a, c, f, f, b, d, d, b, e, f, b, e, a. Array $count[r]$ shows the cumulative counts: 0, 2, 5, 6, 8, 9, 12. Red arrows point from the first two 'd's in a to the 6th and 7th slots in a , indicating they are being moved there.

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

move items → for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

i	a[i]	i	aux[i]
0	d	0	a
1	a	1	a
2	c	2	b
3	f	3	b
4	f	4	b
5	b	5	c
6	d	6	d
7	b	7	d
8	f	8	e
9	b	9	f
10	e	10	f
11	a	11	f

Key-indexed counting demo

Goal. Sort an array $a[]$ of N integers between 0 and $R - 1$.

- Count frequencies of each letter using key as index.
- Compute frequency cumulates which specify destinations.
- Access cumulates using key as index to move items.
- Copy back into original array.

```
int N = a.length;
int[] count = new int[R+1];

for (int i = 0; i < N; i++)
    count[a[i]+1]++;

for (int r = 0; r < R; r++)
    count[r+1] += count[r];

for (int i = 0; i < N; i++)
    aux[count[a[i]]++] = a[i];

for (int i = 0; i < N; i++)
    a[i] = aux[i];
```

copy
back

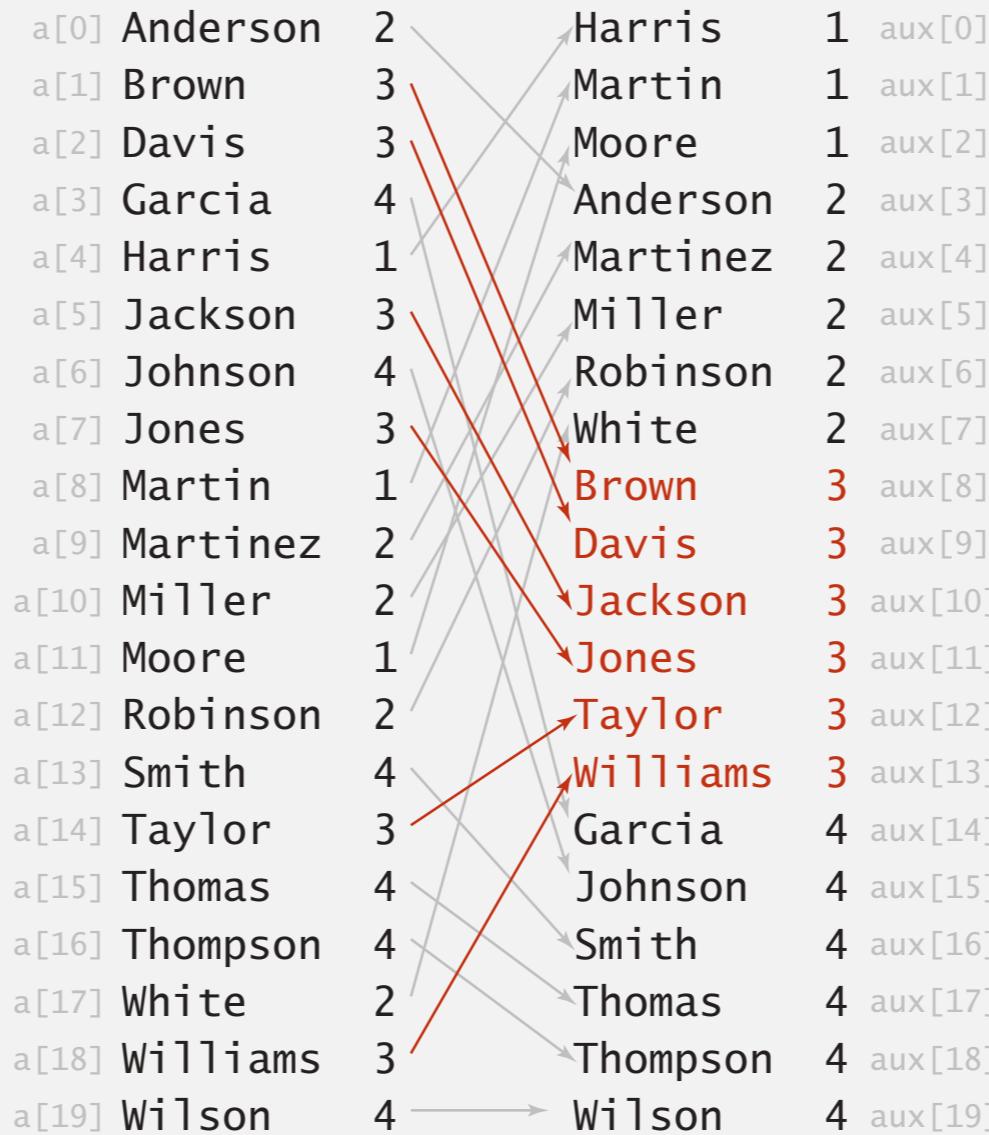
i	a[i]	i	aux[i]
0	a	0	a
1	a	1	a
2	b	2	b
3	b	3	b
4	b	4	b
5	c	5	c
6	d	6	d
7	d	7	d
8	e	8	e
9	f	9	f
10	f	10	f
11	f	11	f

Key-indexed counting: analysis

Proposition. Key-indexed counting uses $\sim 11N + 4R$ array accesses to sort N items whose keys are integers between 0 and $R - 1$.

Proposition. Key-indexed counting uses extra space proportional to $N + R$.

Stable? ✓



Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.1 STRING SORTS

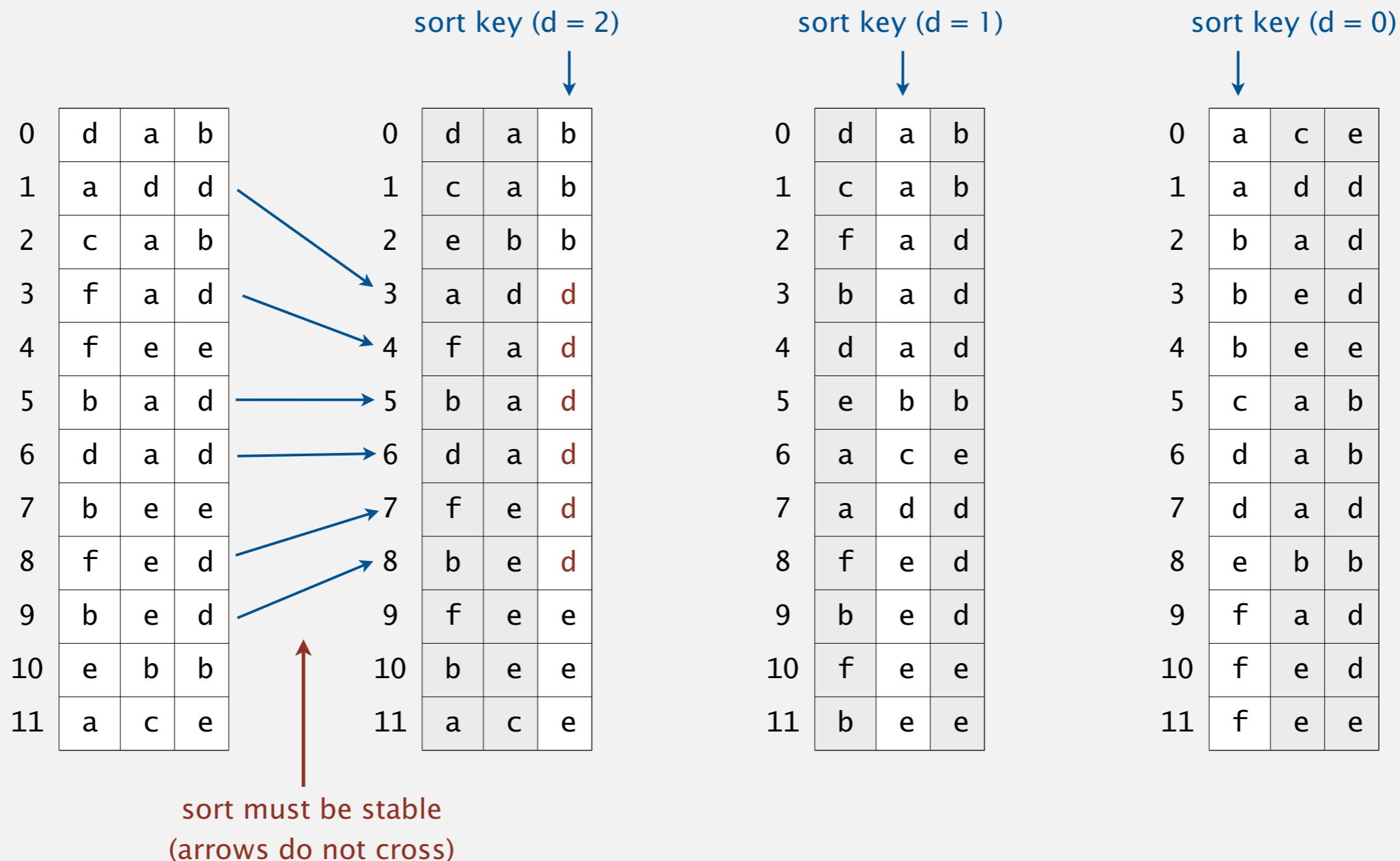
- ▶ *strings in Java*
- ▶ *key-indexed counting*
- ▶ *LSD radix sort*
- ▶ *MSD radix sort*
- ▶ *3-way radix quicksort*
- ▶ *suffix arrays*

Least-significant-digit-first string sort

same length

LSD string (radix) sort.

- Consider characters from right to left.
- Stably sort using d^{th} character as the key (using key-indexed counting).



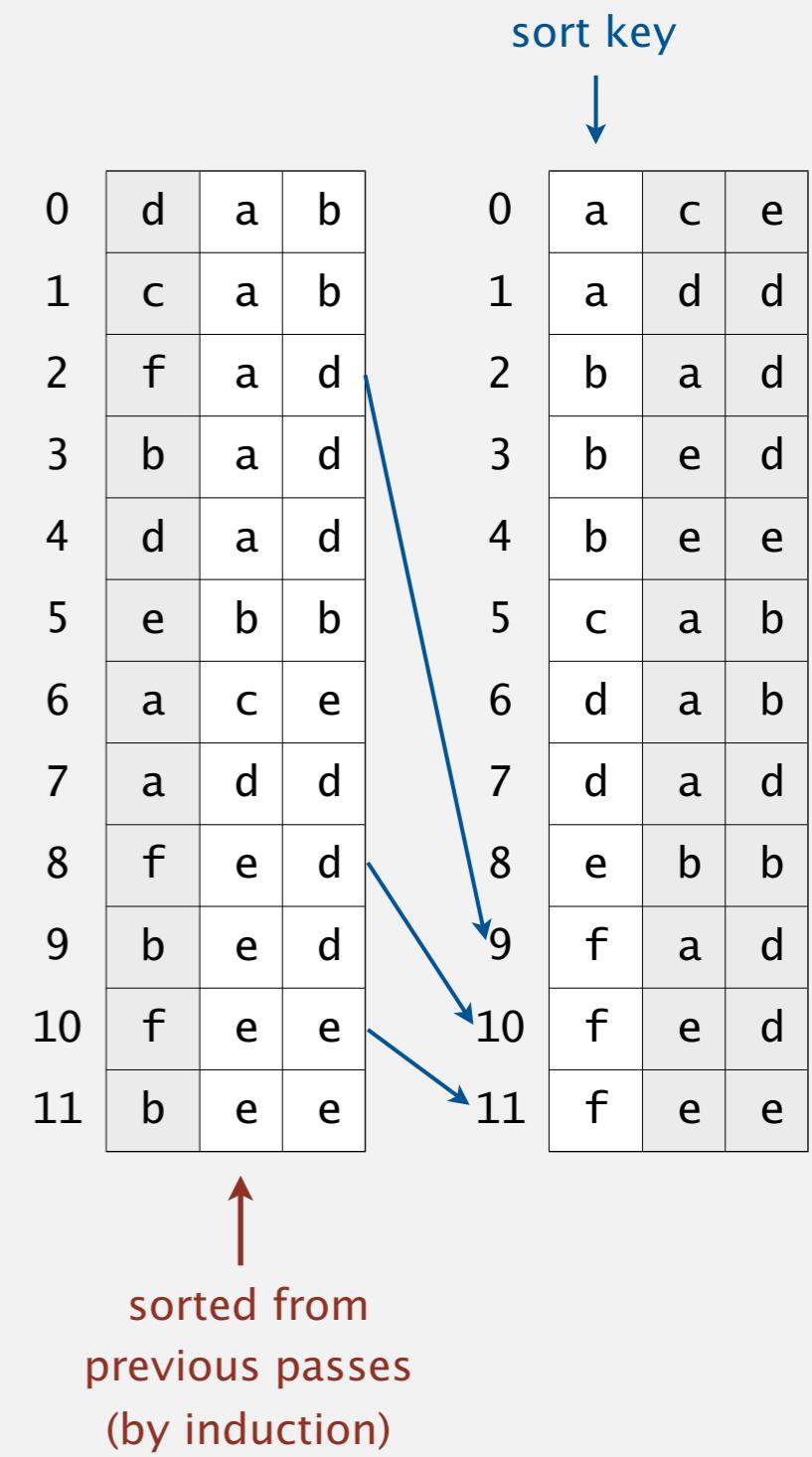
LSD string sort: correctness proof

Proposition. LSD sorts fixed-length strings in ascending order.

Pf. [by induction on i]

After pass i , strings are sorted by last i characters.

- If two strings differ on sort key, key-indexed sort puts them in proper relative order.
- If two strings agree on sort key, stability keeps them in proper relative order.



Proposition. LSD sort is stable.

LSD string sort: Java implementation

```
public class LSD
{
    public static void sort(String[] a, int W)
    {
        int R = 256;
        int N = a.length;   So luong chuoi trong mang
        String[] aux = new String[N];

        for (int d = W-1; d >= 0; d--)
        {
            int[] count = new int[R+1];
            for (int i = 0; i < N; i++)
                count[a[i].charAt(d) + 1]++;
            for (int r = 0; r < R; r++)
                count[r+1] += count[r];
            for (int i = 0; i < N; i++)
                aux[count[a[i].charAt(d)]++] = a[i];
            for (int i = 0; i < N; i++)
                a[i] = aux[i];
        }
    }
}
```

Chieu dai co dinh cua cac chuoi
fixed-length W strings

radix R

do key-indexed counting
for each digit from right to left

key-indexed counting

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	yes	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	yes	compareTo()
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$	no	compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1	no	compareTo()
LSD †	$2 W N$	$2 W N$	$N + R$	yes	charAt()

* probabilistic

† fixed-length W keys

Q. What if strings do not have same length?

String sorting interview question

Problem. Sort one million 32-bit integers.

Ex. Google (or presidential) interview.

Which sorting method to use?

- Insertion sort.
- Mergesort.
- Quicksort.
- Heapsort.
- LSD string sort.

$$2^{139} \times 2^{20} \times 32 \times 1000000$$

$$2^{139} \times (2^{20} + 2) \times 32$$

$$2^{139} \times (2^{20} + 16) \times 5$$



How to take a census in 1900s?

1880 Census. Took 1,500 people 7 years to manually process data.

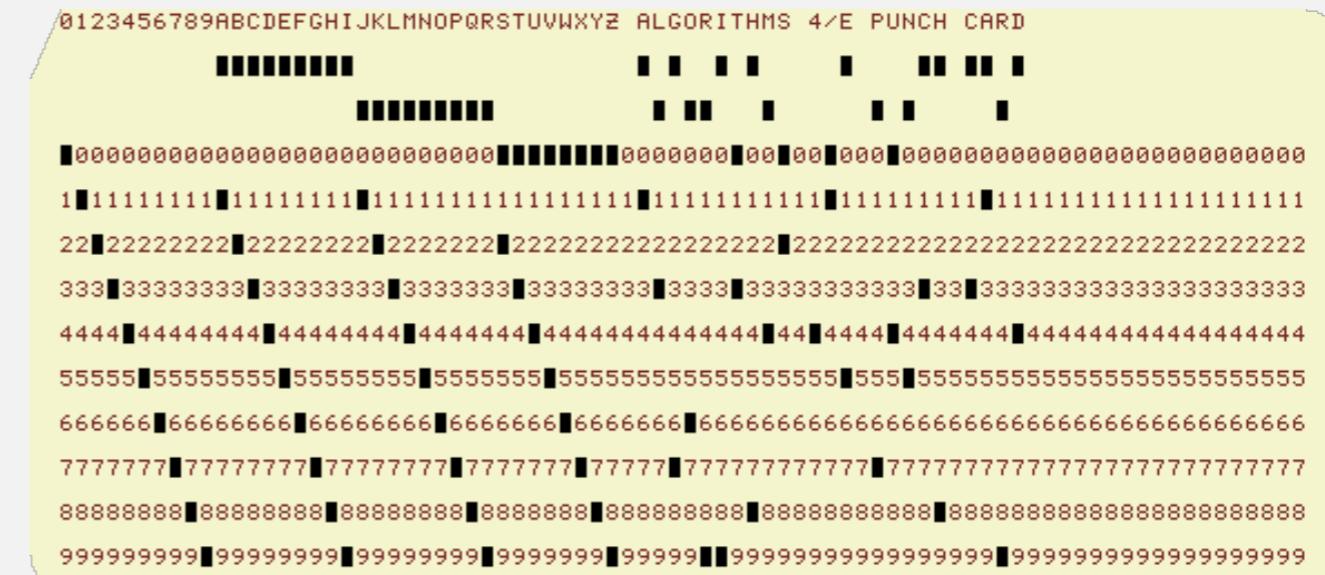


Herman Hollerith. Developed counting and sorting machine to automate.

- Use punch cards to record data (e.g., gender, age).
- Machine sorts one column at a time (into one of 12 bins).
- Typical question: how many women of age 20 to 30?



Hollerith tabulating machine and sorter



punch card (12 holes per column)

1890 Census. Finished months early and under budget!

How to get rich sorting in 1900s?

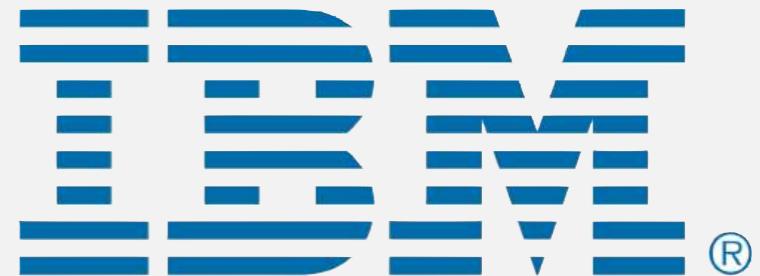
Punch cards. [1900s to 1950s]

- Also useful for accounting, inventory, and business processes.
- Primary medium for data entry, storage, and processing.

Hollerith's company later merged with 3 others to form Computing Tabulating Recording Corporation (CTR); company renamed in 1924.



IBM 80 Series Card Sorter (650 cards per minute)



LSD string sort: a moment in history (1960s)



card punch



punched cards



card reader



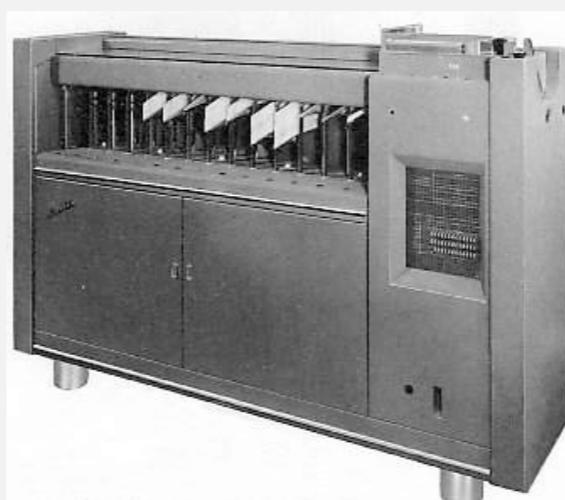
mainframe



line printer

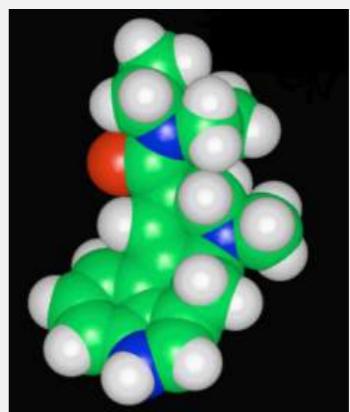
To sort a card deck

- start on right column
- put cards into hopper
- machine distributes into bins
- pick up cards (stable)
- move left one column
- continue until sorted



card sorter

not related to sorting



Lysergic Acid Diethylamide
(Lucy in the Sky with Diamonds)

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.1 STRING SORTS

- ▶ *strings in Java*
- ▶ *key-indexed counting*
- ▶ *LSD radix sort*
- ▶ *MSD radix sort*
- ▶ *3-way radix quicksort*
- ▶ *suffix arrays*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.1 STRING SORTS

- ▶ *strings in Java*
- ▶ *key-indexed counting*
- ▶ *LSD radix sort*
- ▶ **MSD radix sort**
- ▶ *3-way radix quicksort*
- ▶ *suffix arrays*

Most-significant-digit-first string sort

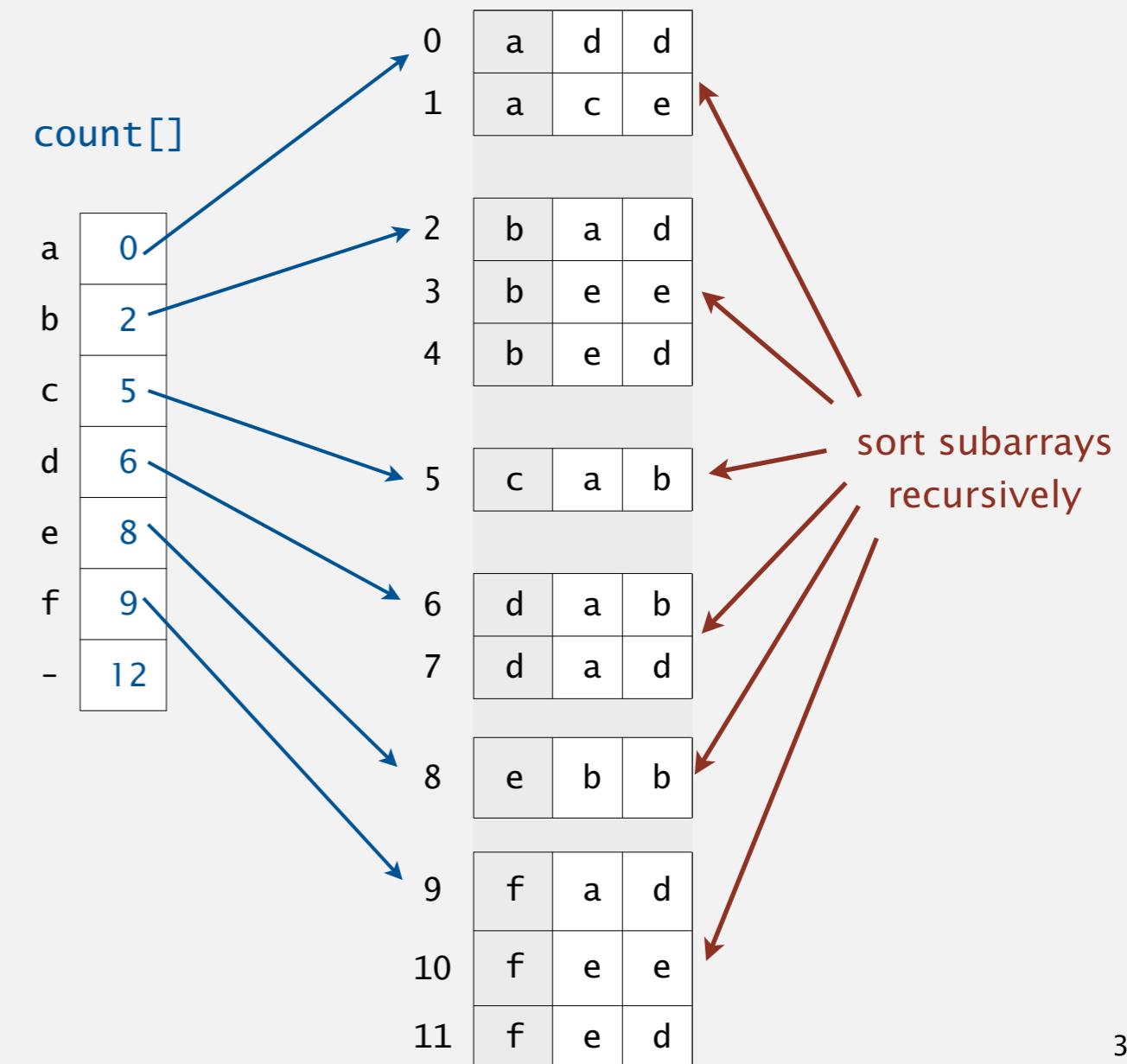
MSD string (radix) sort.

- Partition array into R pieces according to first character (use key-indexed counting).
- Recursively sort all strings that start with each character (key-indexed counts delineate subarrays to sort).

0	d	a	b
1	a	d	d
2	c	a	b
3	f	a	d
4	f	e	e
5	b	a	d
6	d	a	d
7	b	e	e
8	f	e	d
9	b	e	d
10	e	b	b
11	a	c	e

0	a	d	d
1	a	c	e
2	b	a	d
3	b	e	e
4	b	e	d
5	c	a	b
6	d	a	b
7	d	a	d
8	e	b	b
9	f	a	d
10	f	e	e
11	f	e	d

sort key



MSD string sort: example

input		d							
she	are								
sells	by	to	by						
seashells	she	seas	seashells	sea	seashells	sea	seashells	seashells	sea
by	sells	seashells	sea	seashells	seashells	seashells	seashells	seashells	seashells
the	seashells	sea	seashells						
sea	sea	seals							
shore	shore	seashells	seals						
the	shells	she							
shells	she	shore							
she	sells	shells							
sells	surely	she							
are	seashells	surely							
surely	the	hi	the						
seashells	the								

need to examine every character in equal keys	end of string goes before any char value	output
are	are	are
by	by	by
sea	sea	sea
seashells	seashells	seashells
seashells	seashells	seashells
sells	sells	sells
sells	sells	sells
she	she	she
shore	shore	shore
shells	shells	shells
she	she	she
surely	surely	surely
the	the	the
the	the	the

Trace of recursive calls for MSD string sort (no cutoff for small subarrays, subarrays of size 0 and 1 omitted)

Variable-length strings

Treat strings as if they had an extra char at end (smaller than any char).

0	s	e	a	-1
1	s	e	a	s
2	s	e	l	l
3	s	h	e	-1
4	s	h	e	-1
5	s	h	e	l
6	s	h	o	r
7	s	u	r	e

why smaller?

she before shells

```
private static int charAt(String s, int d)
{
    if (d < s.length()) return s.charAt(d);
    else return -1;
}
```

C strings. Have extra char '\0' at end \Rightarrow no extra work needed.

MSD string sort: Java implementation

```
public static void sort(String[] a)
{
    aux = new String[a.length]; ←
    sort(a, aux, 0, a.length-1, 0);
}

private static void sort(String[] a, String[] aux, int lo, int hi, int d)
{
    if (hi <= lo) return;
    int[] count = new int[R+2]; → key-indexed counting
    for (int i = lo; i <= hi; i++)
        count[charAt(a[i], d) + 2]++;
    for (int r = 0; r < R+1; r++)
        count[r+1] += count[r];
    for (int i = lo; i <= hi; i++)
        aux[count[charAt(a[i], d) + 1]++] = a[i];
    for (int i = lo; i <= hi; i++)
        a[i] = aux[i - lo];

    for (int r = 0; r < R; r++) → sort R subarrays recursively
        sort(a, aux, lo + count[r], lo + count[r+1] - 1, d+1);
}
```

can recycle aux[] array
but not count[] array

key-indexed counting

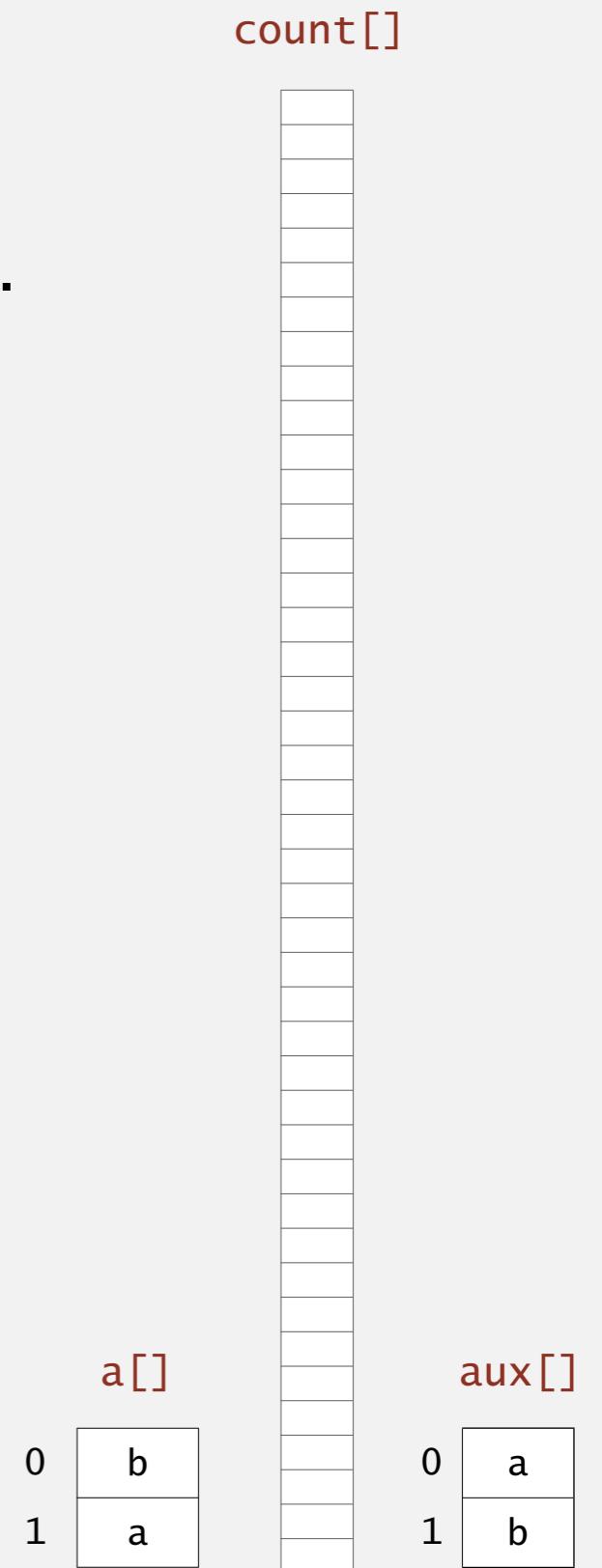
sort R subarrays recursively

MSD string sort: potential for disastrous performance

Observation 1. Much too slow for small subarrays.

- Each function call needs its own count[] array.
- ASCII (256 counts): 100x slower than copy pass for $N=2$.
- Unicode (65,536 counts): 32,000x slower for $N=2$.

Observation 2. Huge number of small subarrays because of recursion.



Cutoff to insertion sort

Solution. Cutoff to insertion sort for small subarrays.

- Insertion sort, but start at d^{th} character.
- Implement `less()` so that it compares starting at d^{th} character.

```
public static void sort(String[] a, int lo, int hi, int d)
{
    for (int i = lo; i <= hi; i++)
        for (int j = i; j > lo && less(a[j], a[j-1], d); j--)
            exch(a, j, j-1);
}
```

```
private static boolean less(String v, String w, int d)
{   return v.substring(d).compareTo(w.substring(d)) < 0; }
```

in Java, forming and comparing substrings is faster than directly comparing chars with `charAt()`

MSD string sort: performance

Number of characters examined.

- MSD examines just enough characters to sort the keys.
- Number of characters examined depends on keys.
- Can be sublinear in input size!



compareTo() based sorts
can also be sublinear!

Random (sublinear)	Non-random with duplicates (nearly linear)	Worst case (linear)
1EI0402	are	1DNB377
1HYL490	by	1DNB377
1R0Z572	sea	1DNB377
2HXE734	seashells	1DNB377
2IYE230	seashells	1DNB377
2XOR846	sells	1DNB377
3CDB573	sells	1DNB377
3CVP720	she	1DNB377
3IGJ319	she	1DNB377
3KNA382	shells	1DNB377
3TAV879	shore	1DNB377
4CQP781	surely	1DNB377
4QGI284	the	1DNB377
4YHV229	the	1DNB377

Characters examined by MSD string sort

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	yes	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	yes	compareTo()
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$	no	compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1	no	compareTo()
LSD †	$2 NW$	$2 NW$	$N + R$	yes	charAt()
MSD ‡	$2 NW$	$N \log_R N$	$N + DR$	yes	charAt()

D = function-call stack depth
(length of longest prefix match)



* probabilistic
† fixed-length W keys
‡ average-length W keys

MSD string sort vs. quicksort for strings

Disadvantages of MSD string sort.

- Extra space for aux[].
- Extra space for count[].
- Inner loop has a lot of instructions.
- Accesses memory "randomly" (cache inefficient).

Disadvantage of quicksort.

- Linearithmic number of string compares (not linear).
- Has to rescan many characters in keys with long prefix matches.

Goal. Combine advantages of MSD and quicksort.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.1 STRING SORTS

- ▶ *strings in Java*
- ▶ *key-indexed counting*
- ▶ *LSD radix sort*
- ▶ **MSD radix sort**
- ▶ *3-way radix quicksort*
- ▶ *suffix arrays*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

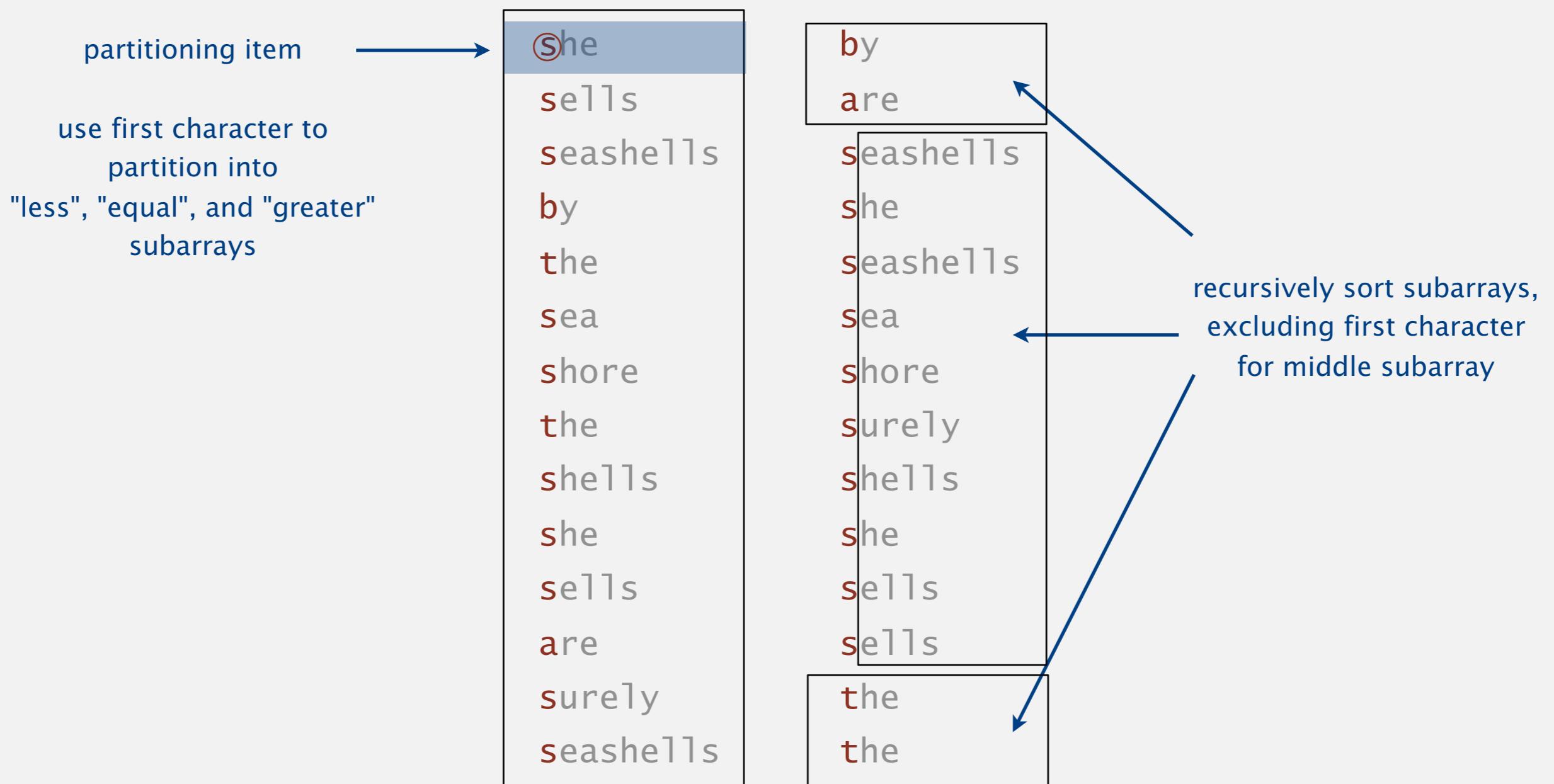
5.1 STRING SORTS

- ▶ *strings in Java*
- ▶ *key-indexed counting*
- ▶ *LSD radix sort*
- ▶ *MSD radix sort*
- ▶ ***3-way radix quicksort***
- ▶ *suffix arrays*

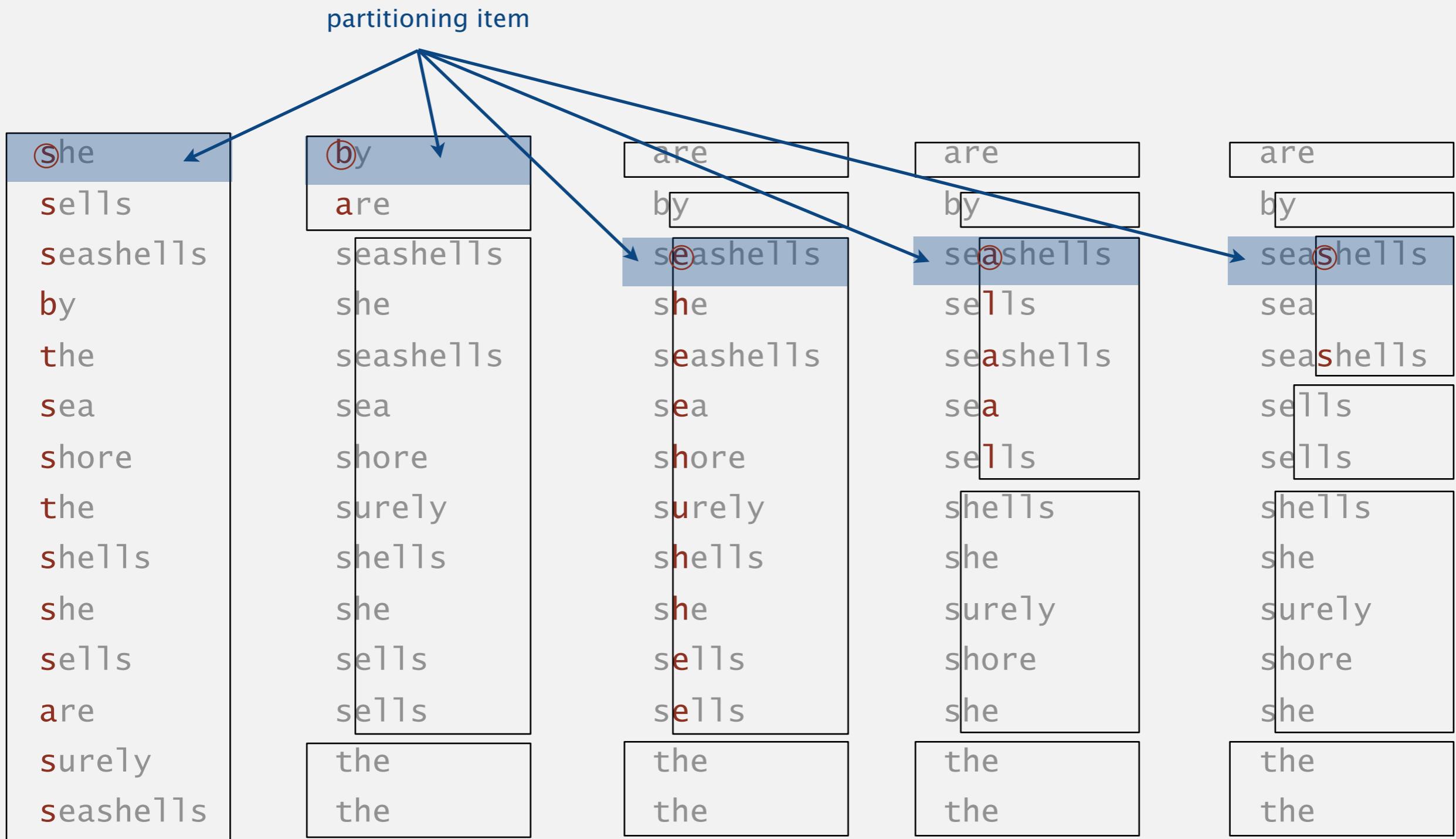
3-way string quicksort (Bentley and Sedgewick, 1997)

Overview. Do 3-way partitioning on the d^{th} character.

- Less overhead than R -way partitioning in MSD string sort.
- Does not re-examine characters equal to the partitioning char
(but does re-examine characters not equal to the partitioning char).



3-way string quicksort: trace of recursive calls



Trace of first few recursive calls for 3-way string quicksort (subarrays of size 1 not shown)

3-way string quicksort: Java implementation

```
private static void sort(String[] a)
{  sort(a, 0, a.length - 1, 0);  }

private static void sort(String[] a, int lo, int hi, int d)
{
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    int v = charAt(a[lo], d); 3-way partitioning  
(using dth character)
    int i = lo + 1;
    while (i <= gt)
    {
        int t = charAt(a[i], d);
        if      (t < v) exch(a, lt++, i++);
        else if (t > v) exch(a, i, gt--);
        else              i++;
    }
    sort(a, lo, lt-1, d);
    if (v >= 0) sort(a, lt, gt, d+1); sort 3 subarrays recursively
    sort(a, gt+1, hi, d);
}
```

to handle variable-length strings

3-way string quicksort vs. standard quicksort

Standard quicksort.

- Uses $\sim 2N \ln N$ **string compares** on average.
- Costly for keys with long common prefixes (and this is a common case!)

3-way string (radix) quicksort.

- Uses $\sim 2N \ln N$ **character compares** on average for random strings.
- Avoids re-comparing long common prefixes.

Fast Algorithms for Sorting and Searching Strings

Jon L. Bentley* Robert Sedgewick#

Abstract

We present theoretical algorithms for sorting and searching multikey data, and derive from them practical C implementations for applications in which keys are character strings. The sorting algorithm blends Quicksort and radix sort; it is competitive with the best known C sort codes. The searching algorithm blends tries and binary

that is competitive with the most efficient string sorting programs known. The second program is a symbol table implementation that is faster than hashing, which is commonly regarded as the fastest symbol table implementation. The symbol table implementation is much more space-efficient than multiway trees, and supports more advanced searches.

3-way string quicksort vs. MSD string sort

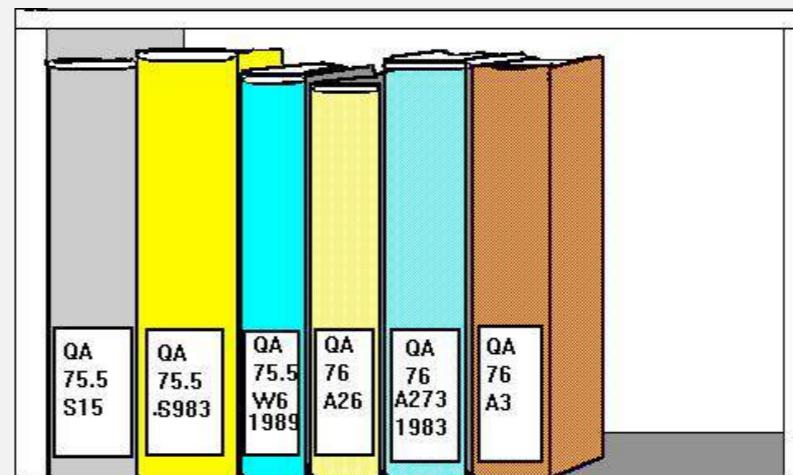
MSD string sort.

- Is cache-inefficient.
- Too much memory storing count[].
- Too much overhead reinitializing count[] and aux[].

3-way string quicksort.

- Has a short inner loop.
- Is cache-friendly.
- Is in-place.

library of Congress call numbers



Bottom line. 3-way string quicksort is method of choice for sorting strings.

Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	$\frac{1}{2} N^2$	$\frac{1}{4} N^2$	1	yes	compareTo()
mergesort	$N \lg N$	$N \lg N$	N	yes	compareTo()
quicksort	$1.39 N \lg N$ *	$1.39 N \lg N$	$c \lg N$	no	compareTo()
heapsort	$2 N \lg N$	$2 N \lg N$	1	no	compareTo()
LSD †	$2 NW$	$2 NW$	$N + R$	yes	charAt()
MSD ‡	$2 NW$	$N \log_R N$	$N + D R$	yes	charAt()
3-way string quicksort	$1.39 W N \lg R$ *	$1.39 N \lg N$	$\log N + W$	no	charAt()

* probabilistic

† fixed-length W keys

‡ average-length W keys

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.1 STRING SORTS

- ▶ *strings in Java*
- ▶ *key-indexed counting*
- ▶ *LSD radix sort*
- ▶ *MSD radix sort*
- ▶ ***3-way radix quicksort***
- ▶ *suffix arrays*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.1 STRING SORTS

- ▶ *strings in Java*
- ▶ *key-indexed counting*
- ▶ *LSD radix sort*
- ▶ *MSD radix sort*
- ▶ *3-way radix quicksort*
- ▶ ***suffix arrays***

Keyword-in-context search

Given a text of N characters, preprocess it to enable fast substring search
(find all occurrences of query string context).

```
% more tale.txt
it was the best of times
it was the worst of times
it was the age of wisdom
it was the age of foolishness
it was the epoch of belief
it was the epoch of incredulity
it was the season of light
it was the season of darkness
it was the spring of hope
it was the winter of despair
:
```

Applications. Linguistics, databases, web search, word processing,

Keyword-in-context search

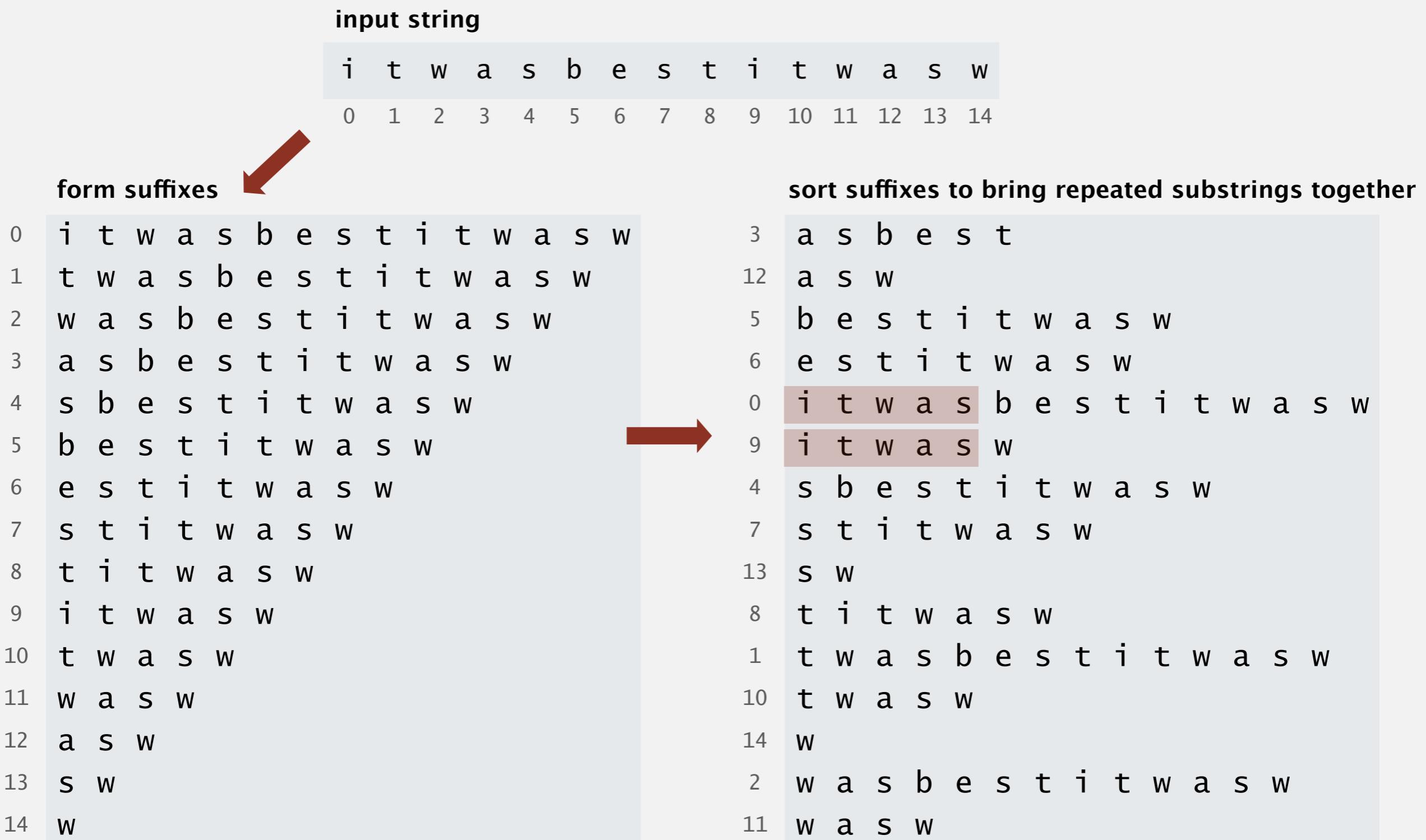
Given a text of N characters, preprocess it to enable fast substring search (find all occurrences of query string context).

```
% java KWIC tale.txt 15 ← characters of
search                                              surrounding context
o st giless to search for contraband
her unavailing search for your fathe
le and gone in search of her husband
t provinces in search of impoverishe
dispersing in search of other carri
n that bed and search the straw hold

better thing
t is a far far better thing that i do than
some sense of better things else forgotte
was capable of better things mr carton ent
```

Applications. Linguistics, databases, web search, word processing,

Suffix sort



Keyword-in-context search: suffix-sorting solution

- Preprocess: **suffix sort** the text.
- Query: **binary search** for query; scan until mismatch.

KWIC search for "search" in Tale of Two Cities

	:
632698	s e a l e d _ m y _ l e t t e r _ a n d _ ...
713727	s e a m s t r e s s _ i s _ l i f t e d _ ...
660598	s e a m s t r e s s _ o f _ t w e n t y _ ...
67610	s e a m s t r e s s _ w h o _ w a s _ w i ...
4430	s e a r c h _ f o r _ c o n t r a b a n d ...
42705	s e a r c h _ f o r _ y o u r _ f a t h e ...
499797	s e a r c h _ o f _ h e r _ h u s b a n d ...
182045	s e a r c h _ o f _ i m p o v e r i s h e ...
143399	s e a r c h _ o f _ o t h e r _ c a r r i ...
411801	s e a r c h _ t h e _ s t r a w _ h o l d ...
158410	s e a r e d _ m a r k i n g _ a b o u t _ ...
691536	s e a s _ a n d _ m a d a m e _ d e f a r ...
536569	s e a s e _ a _ t e r r i b l e _ p a s s ...
484763	s e a s e _ t h a t _ h a d _ b r o u g h ...
	:

Longest repeated substring

Given a string of N characters, find the longest repeated substring.

```
a a c a a g t t a c a a g c a t g a t g c t g t a c t a  
g g a g a g t t a t a c t g g t c g t c a a a c c t g a a  
c c t a a t c c t t g t g t a c a c a c a c t a c t a  
c t g t c g t c g t c a t a t a t c g a g a t c a t c g a  
a c c g g a a g g c c g g a c a a g g c g g g g g t a t  
a g a t a g a t a g a c c c c t a g a t a c a c a t a c a  
t a g a t c t a g c t a g c t c a t c g a t a c a  
c a c t c t c a c a c t c a a g a g t t a t a c t g g t c  
a a c a c a c t a c t a c g a c a g a c g a c c a a c c a  
g a c a g a a a a a a a a c t c t a t a t c t a t a a a a
```

Applications. Bioinformatics, cryptanalysis, data compression, ...

Longest repeated substring: a musical application

Visualize repetitions in music. <http://www.bewitched.com>

Mary Had a Little Lamb



Bach's Goldberg Variations



Longest repeated substring

Given a string of N characters, find the longest repeated substring.

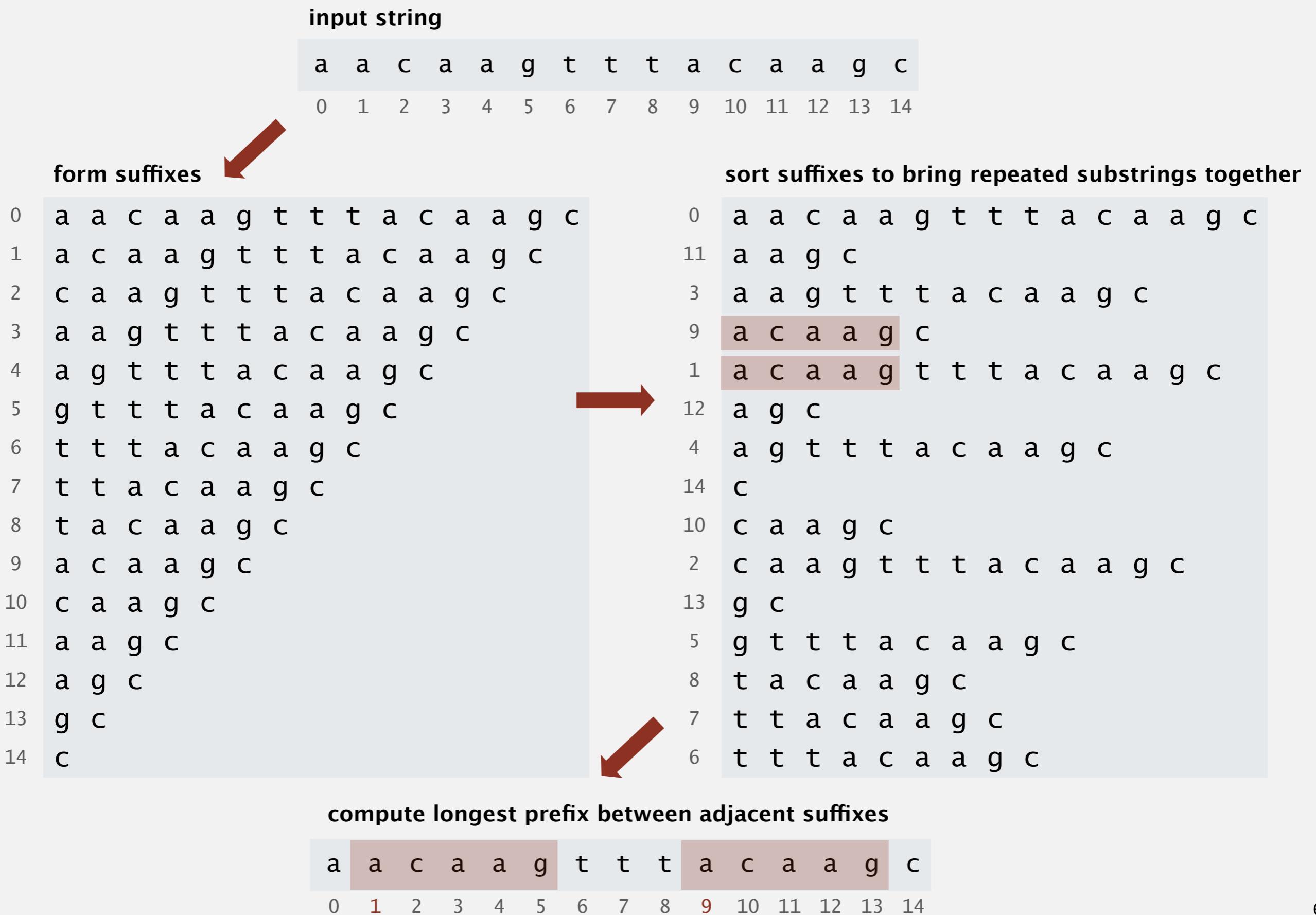
Brute-force algorithm.

- Try all indices i and j for start of possible match.
- Compute longest common prefix (LCP) for each pair.



Analysis. Running time $\leq D N^2$, where D is length of longest match.

Longest repeated substring: a sorting solution



Longest repeated substring: Java implementation

```
public String lrs(String s)
{
    int N = s.length();

    String[] suffixes = new String[N];
    for (int i = 0; i < N; i++)
        suffixes[i] = s.substring(i, N);

    Arrays.sort(suffixes);

    String lrs = "";
    for (int i = 0; i < N-1; i++)
    {
        int len = lcp(suffixes[i], suffixes[i+1]);
        if (len > lrs.length())
            lrs = suffixes[i].substring(0, len);
    }
    return lrs;
}
```

create suffixes
(linear time and space)

sort suffixes

find LCP between
adjacent suffixes in
sorted order

```
% java LRS < moby dick.txt
,- Such a funny, sporty, gamy, jesty, joky, hoky-poky lad, is the Ocean, oh! Th
```

Sorting challenge

Problem. Five scientists A , B , C , D , and E are looking for long repeated substring in a genome with over 1 billion nucleotides.

- A has a grad student do it by hand.
- B uses brute force (check all pairs).
- C uses suffix sorting solution with insertion sort.
- D uses suffix sorting solution with LSD string sort.
- ✓ • E uses suffix sorting solution with 3-way string quicksort.



but only if LRS is not long (!)

Q. Which one is more likely to lead to a cure cancer?

Longest repeated substring: empirical analysis

input file	characters	brute	suffix sort	length of LRS
LRS.java	2,162	0.6 sec	0.14 sec	73
amendments.txt	18,369	37 sec	0.25 sec	216
aesop.txt	191,945	1.2 hours	1.0 sec	58
mobydick.txt	1.2 million	43 hours †	7.6 sec	79
chromosome11.txt	7.1 million	2 months †	61 sec	12,567
pi.txt	10 million	4 months †	84 sec	14
pipi.txt	20 million	forever †	???	10 million

† estimated

Suffix sorting: worst-case input

Bad input: longest repeated substring very long.

- Ex: same letter repeated N times.
- Ex: two copies of the same Java codebase.

form suffixes	sorted suffixes
0 t w i n s t w i n s	9 i n s
1 w i n s t w i n s	8 i n s t w i n s
2 i n s t w i n s	7 n s
3 n s t w i n s	6 n s t w i n s
4 s t w i n s	5 s
5 t w i n s	4 s t w i n s
6 w i n s	3 t w i n s
7 i n s	2 t w i n s t w i n s
8 n s	1 w i n s
9 s	0 w i n s t w i n s

LRS needs at least $1 + 2 + 3 + \dots + D$ character compares,
where $D = \text{length of longest match}$.

Running time. Quadratic (or worse) in D for LRS (and also for sort).

Suffix sorting challenge

Problem. Suffix sort an arbitrary string of length N .

Q. What is worst-case running time of best algorithm for problem?

- Quadratic.
- ✓ • Linearithmic. ← Manber-Myers algorithm
- ✓ • Linear. ← suffix trees (beyond our scope)
- Nobody knows.

Suffix sorting in linearithmic time

Manber-Myers MSD algorithm overview.

- Phase 0: sort on first character using key-indexed counting sort.
- Phase i : given array of suffixes sorted on first 2^{i-1} characters, create array of suffixes sorted on first 2^i characters.

Worst-case running time. $N \lg N$.

- Finishes after $\lg N$ phases.
- Can perform a phase in linear time. (!) [ahead]

Linearithmic suffix sort example: phase 0

original suffixes

0	b a b a a a a b c b a b a a a a a 0
1	a b a a a a b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0
3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
7	b c b a b a a a a a 0
8	c b a b a a a a a 0
9	b a b a a a a a 0
10	a b a a a a a 0
11	b a a a a a a 0
12	a a a a a a 0
13	a a a a a 0
14	a a a a 0
15	a a 0
16	a 0
17	0

key-indexed counting sort (first character)

17	0
1	a b a a a a b c b a b a a a a a 0
16	a 0
3	a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0
15	a a 0
14	a a a 0
13	a a a a 0
12	a a a a a 0
10	a b a a a a a 0
0	b a b a a a a b c b a b a a a a a 0
9	b a b a a a a a 0
11	b a a a a a 0
7	b c b a b a a a a a 0
2	b a a a a b c b a b a a a a a 0
8	c b a b a a a a a 0

↑
sorted

Linearithmic suffix sort example: phase 1

	original suffixes	index sort (first two characters)
0	b a b a a a a b c b a b a a a a a 0	17 0
1	a b a a a a b c b a b a a a a a 0	16 a 0
2	b a a a a b c b a b a a a a a 0	12 a a a a a a 0
3	a a a a b c b a b a a a a a 0	3 a a a a a b c b a b a a a a a 0
4	a a a b c b a b a a a a a 0	4 a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0	5 a a b c b a b a a a a a 0
6	a b c b a b a a a a a 0	13 a a a a 0
7	b c b a b a a a a a 0	15 a a 0
8	c b a b a a a a a 0	14 a a a a 0
9	b a b a a a a a 0	6 a b c b a b a a a a a 0
10	a b a a a a a 0	1 a b a a a a b c b a b a a a a a 0
11	b a a a a a 0	10 a b a a a a a 0
12	a a a a a 0	0 b a b a a a a b c b a b a a a a a 0
13	a a a a 0	9 b a b a a a a a 0
14	a a a 0	11 b a a a a a 0
15	a a 0	2 b a a a a b c b a b a a a a a 0
16	a 0	7 b c b a b a a a a a 0
17	0	8 c b a b a a a a a 0

↑
sorted

Linearithmic suffix sort example: phase 2

	original suffixes	index sort (first four characters)
0	b a b a a a a b c b a b a a a a a 0	17 0
1	a b a a a a b c b a b a a a a a 0	16 a 0
2	b a a a a b c b a b a a a a a 0	15 a a 0
3	a a a a b c b a b a a a a a 0	14 a a a 0
4	a a a b c b a b a a a a a 0	3 a a a a b c b a b a a a a a 0
5	a a b c b a b a a a a a 0	12 a a a a a a 0
6	a b c b a b a a a a a 0	13 a a a a 0
7	b c b a b a a a a a 0	4 a a a b c b a b a a a a a 0
8	c b a b a a a a a 0	5 a a b c b a b a a a a a 0
9	b a b a a a a a 0	1 a b a a a a a b c b a b a a a a 0
10	a b a a a a a 0	10 a b a a a a a 0
11	b a a a a a a 0	6 a b c b a b a a a a a 0
12	a a a a a a 0	2 b a a a a a b c b a b a a a a 0
13	a a a a a 0	11 b a a a a a a 0
14	a a a a 0	0 b a b a a a a b c b a b a a a a 0
15	a a 0	9 b a b a a a a a 0
16	a 0	7 b c b a b a a a a a 0
17	0	8 c b a b a a a a a 0

↑
sorted

Linearithmic suffix sort example: phase 3

original suffixes	index sort (first eight characters)
0 b a b a a a a b c b a b a a a a a 0	17 0
1 a b a a a a b c b a b a a a a a 0	16 a 0
2 b a a a a b c b a b a a a a a 0	15 a a 0
3 a a a a b c b a b a a a a a 0	14 a a a 0
4 a a a b c b a b a a a a a 0	13 a a a a 0
5 a a b c b a b a a a a a 0	12 a a a a a 0
6 a b c b a b a a a a a 0	3 a a a a b c b a b a a a a a 0
7 b c b a b a a a a a 0	4 a a a b c b a b a a a a a 0
8 c b a b a a a a a 0	5 a a b c b a b a a a a a 0
9 b a b a a a a a 0	10 a b a a a a a 0
10 a b a a a a a 0	1 a b a a a a b c b a b a a a a a 0
11 b a a a a a 0	6 a b c b a b a a a a a 0
12 a a a a a 0	11 b a a a a a 0
13 a a a a 0	2 b a a a a b c b a b a a a a a 0
14 a a a 0	9 b a b a a a a a 0
15 a a 0	0 b a b a a a a b c b a b a a a a a 0
16 a 0	7 b c b a b a a a a a 0
17 0	8 c b a b a a a a a 0

finished (no equal keys) 

Constant-time string compare by indexing into inverse

	original suffixes	index sort (first four characters)	inverse[]
0	b a b a a a a b c b a b a a a a a 0	17 0	0 14
1	a b a a a a b c b a b a a a a a 0	16 a 0	1 9
2	b a a a a b c b a b a a a a a 0	15 a a 0	2 12
3	a a a a b c b a b a a a a a 0	14 a a a 0	3 4
4	a a a b c b a b a a a a a 0	3 a a a a b c b a b a a a a a 0	4 7
5	a a b c b a b a a a a a 0	12 a a a a a 0	5 8
6	a b c b a b a a a a a 0	13 a a a a a 0	6 11
7	b c b a b a a a a a 0	4 a a a b c b a b a a a a a 0	7 16
8	c b a b a a a a a 0	5 a a b c b a b a a a a a 0	8 17
9	b a b a a a a a 0	1 a b a a a a a b c b a b a a a a a 0	9 15
10	a b a a a a a 0	10 a b a a a a a 0	10 10
11	b a a a a a a 0	6 a b c b a b a a a a a 0	11 13
12	a a a a a a 0	2 b a a a a a b c b a b a a a a a 0	12 5
13	a a a a a 0	11 b a a a a a a 0	13 6
14	a a a a 0	0 b a b a a a a b c b a b a a a a a 0	14 3
15	a a 0	9 b a b a a a a a 0	15 2
16	a 0	7 b c b a b a a a a a 0	16 1
17	0	8 c b a b a a a a a 0	17 0

$\text{suffixes}_4[13] \leq \text{suffixes}_4[4]$ (because $\text{inverse}[13] < \text{inverse}[4]$)
 so $\text{suffixes}_8[9] \leq \text{suffixes}_8[0]$

String sorting summary

We can develop linear-time sorts.

- Key compares not necessary for string keys.
- Use characters as index in an array.

We can develop sublinear-time sorts.

- Input size is amount of data in keys (not number of keys).
- Not all of the data has to be examined.

3-way string quicksort is asymptotically optimal.

- $1.39 N \lg N$ chars for random data.

Long strings are rarely random in practice.

- Goal is often to learn the structure!
- May need specialized algorithms.

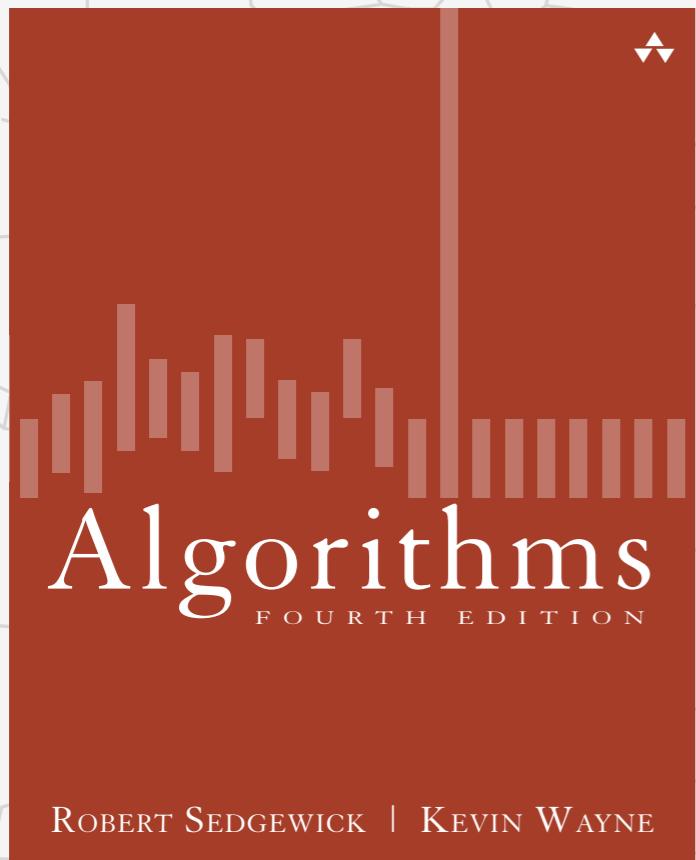
Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.1 STRING SORTS

- ▶ *strings in Java*
- ▶ *key-indexed counting*
- ▶ *LSD radix sort*
- ▶ *MSD radix sort*
- ▶ *3-way radix quicksort*
- ▶ ***suffix arrays***

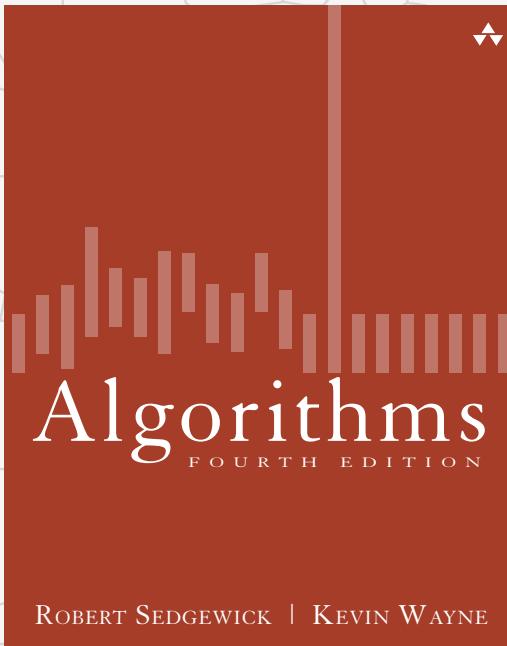


ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.1 STRING SORTS

- ▶ *strings in Java*
- ▶ *key-indexed counting*
- ▶ *LSD radix sort*
- ▶ *MSD radix sort*
- ▶ *3-way radix quicksort*
- ▶ *suffix arrays*



<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

- ▶ *introduction*
- ▶ *brute force*
- ▶ *Knuth-Morris-Pratt*
- ▶ *Boyer-Moore*
- ▶ *Rabin-Karp*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

- ▶ *introduction*
- ▶ *brute force*
- ▶ *Knuth-Morris-Pratt*
- ▶ *Boyer-Moore*
- ▶ *Rabin-Karp*

Substring search

Goal. Find pattern of length M in a text of length N .

→ typically $N \gg M$

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A
 ↑
 match

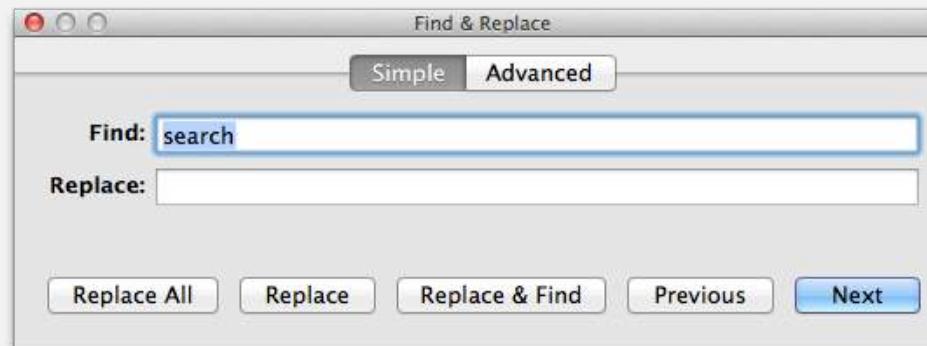
Substring search applications

Goal. Find pattern of length M in a text of length N .

↑
typically $N \gg M$

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A
 ↑
 match



Substring search applications

Goal. Find pattern of length M in a text of length N .

↑
typically $N \gg M$

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A
 ↑
 match

Computer forensics. Search memory or disk for signatures,
e.g., all URLs or RSA keys that the user has entered.



<http://citp.princeton.edu/memory>

Substring search applications

Goal. Find pattern of length M in a text of length N .

↑
typically $N \gg M$

pattern → N E E D L E

text → I N A H A Y S T A C K N E E D L E I N A
 ↑
 match

Identify patterns indicative of spam.

- PROFITS
- LOSE WEIGHT
- herbal Viagra
- There is no catch.
- This is a one-time mailing.
- This message is sent in compliance with spam regulations.



Substring search applications

Electronic surveillance.



Need to monitor all
internet traffic.
(security)



Well, we're mainly
interested in
“ATTACK AT DAWN”

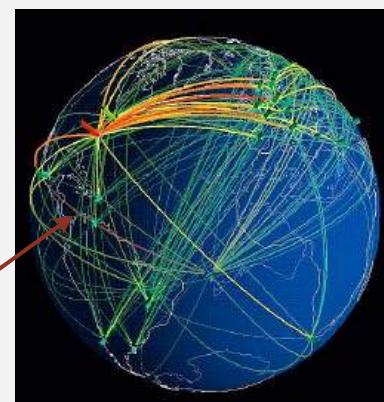
No way!
(privacy)



OK. Build a
machine that just
looks for that.



“ATTACK AT DAWN”
substring search
machine
found



Substring search applications

Screen scraping. Extract relevant data from web page.

Ex. Find string delimited by and after first occurrence of pattern Last Trade::



<http://finance.yahoo.com/q?s=goog>

```
...
<tr>
<td class= "yfnc_tablehead1"
width= "48%">
Last Trade:
</td>
<td class= "yfnc_tabledata1">
<big><b>452.92</b></big>
</td></tr>
<td class= "yfnc_tablehead1"
width= "48%">
Trade Time:
</td>
<td class= "yfnc_tabledata1">
...
```

Screen scraping: Java implementation

Java library. The `indexOf()` method in Java's string library returns the index of the first occurrence of a given string, starting at a given offset.

```
public class StockQuote
{
    public static void main(String[] args)
    {
        String name = "http://finance.yahoo.com/q?s=";
        In in = new In(name + args[0]);
        String text = in.readAll();
        int start      = text.indexOf("Last Trade:", 0);
        int from       = text.indexOf("<b>", start);
        int to         = text.indexOf("</b>", from);
        String price = text.substring(from + 3, to);
        StdOut.println(price);
    }
}
```

```
% java StockQuote goog  
582.93
```

```
% java StockQuote msft  
24.84
```

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

- ▶ *introduction*
- ▶ *brute force*
- ▶ *Knuth-Morris-Pratt*
- ▶ *Boyer-Moore*
- ▶ *Rabin-Karp*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

- ▶ *introduction*
- ▶ ***brute force***
- ▶ ***Knuth-Morris-Pratt***
- ▶ ***Boyer-Moore***
- ▶ ***Rabin-Karp***

Brute-force substring search

Check for pattern starting at each text position.

i	j	i+j	0	1	2	3	4	5	6	7	8	9	10
			A	B	A	C	A	D	A	B	R	A	C
0	2	2	A	B	R	A							
1	0	1		A	B	R	A						
2	1	3			A	B	R	A					
3	0	3				A	B	R	A				
4	1	5					A	B	R	A			
5	0	5						A	B	R	A		
6	4	10							A	B	R	A	

txt → A B A C A D A B R A C

entries in red are mismatches

entries in gray are for reference only

entries in black match the text

return i when j is M

match

Brute-force substring search: Java implementation

Check for pattern starting at each text position.

i	j	i + j	0	1	2	3	4	5	6	7	8	9	10
			A	B	A	C	A	D	A	B	R	A	C
4	3	7					A	D	A	C	R		
5	0	5						A	D	A	C	R	

```
public static int search(String pat, String txt)
{
    int M = pat.length();
    int N = txt.length();
    for (int i = 0; i <= N - M; i++)
    {
        int j;
        for (j = 0; j < M; j++)
            if (txt.charAt(i+j) != pat.charAt(j))
                break;
        if (j == M) return i; ← index in text where
                           pattern starts
    }
    return N; ← not found
}
```

Brute-force substring search: worst case

Brute-force algorithm can be slow if text and pattern are repetitive.

i	j	$i+j$	0	1	2	3	4	5	6	7	8	9
			txt →	A	A	A	A	A	A	A	A	B
0	4	4	A	A	A	A	B	← pat				
1	4	5		A	A	A	A	B				
2	4	6			A	A	A	A	B			
3	4	7				A	A	A	A	B		
4	4	8					A	A	A	A	B	
5	5	10						<u>A</u>	<u>A</u>	<u>A</u>	<u>A</u>	B

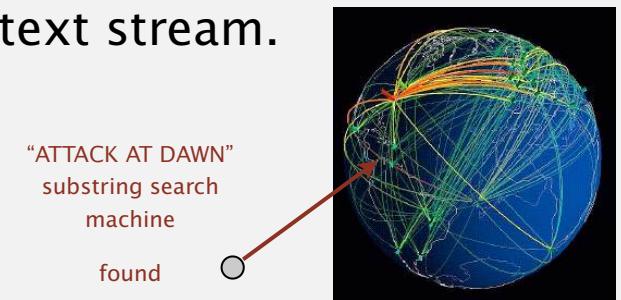
\uparrow
match

Worst case. $\sim MN$ char compares.

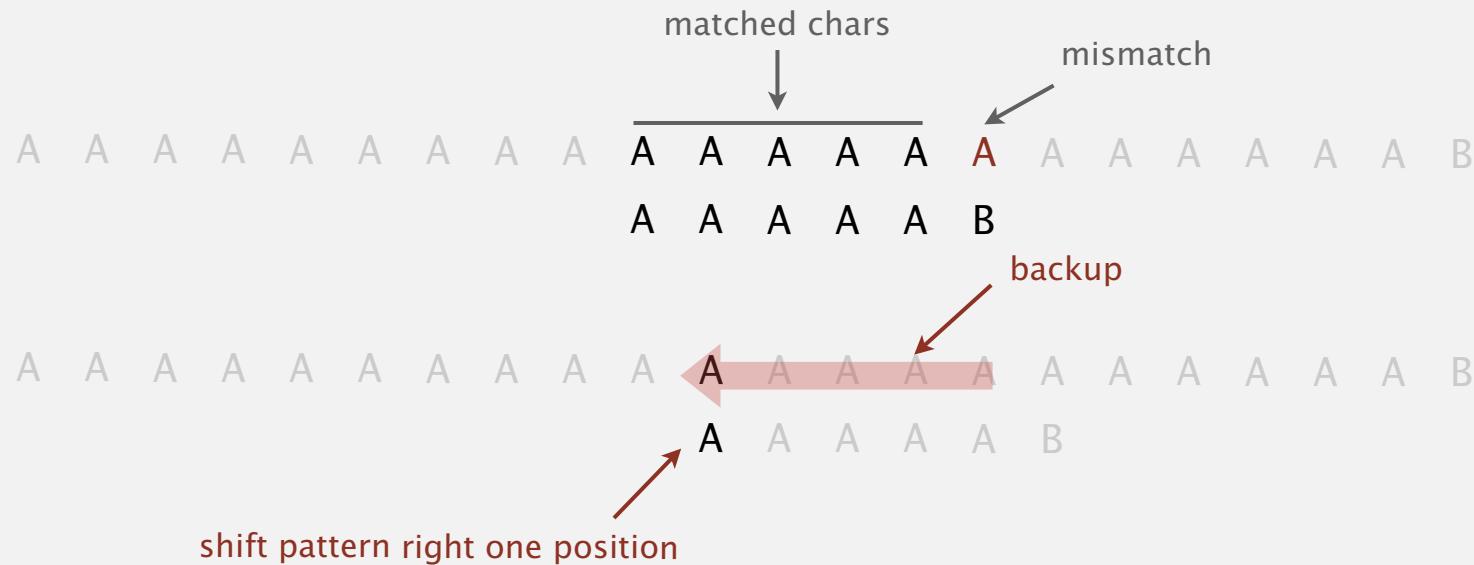
Backup

In many applications, we want to avoid **backup** in text stream.

- Treat input as stream of data.
- Abstract model: standard input.



Brute-force algorithm needs backup for every mismatch.



Approach 1. Maintain buffer of last M characters.

Approach 2. Stay tuned.

Brute-force substring search: alternate implementation

Same sequence of char compares as previous implementation.

- i points to end of sequence of already-matched chars in text.
- j stores # of already-matched chars (end of sequence in pattern).

<u>i</u>	<u>j</u>	0	1	2	3	4	5	6	7	8	9	10
		A	B	A	C	A	D	A	B	R	A	C
7	3				A	D	A	C	R			
5	0				A	D	A	C	R			

```
public static int search(String pat, String txt)
{
    int i, N = txt.length();
    int j, M = pat.length();
    for (i = 0, j = 0; i < N && j < M; i++)
    {
        if (txt.charAt(i) == pat.charAt(j)) j++;
        else { i -= j; j = 0; } ← explicit backup
    }
    if (j == M) return i - M;
    else return N;
}
```

Algorithmic challenges in substring search

Brute-force is not always good enough.

Theoretical challenge. Linear-time guarantee. ← fundamental algorithmic problem

Practical challenge. Avoid backup in text stream. ← often no room or time to save text

Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for each good person to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party. Now is the time for all people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many good people to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for a lot of good people to come to the aid of their party. Now is the time for all of the good people to come to the aid of their party. Now is the time for all good people to come to the aid of their **attack at dawn** party. Now is the time for each person to come to the aid of their party. Now is the time for all good Republicans to come to the aid of their party. Now is the time for all good people to come to the aid of their party. Now is the time for many or all good people to come to the aid of their party. Now is the time for all good Democrats to come to the aid of their party.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

- ▶ *introduction*
- ▶ ***brute force***
- ▶ ***Knuth-Morris-Pratt***
- ▶ ***Boyer-Moore***
- ▶ ***Rabin-Karp***

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

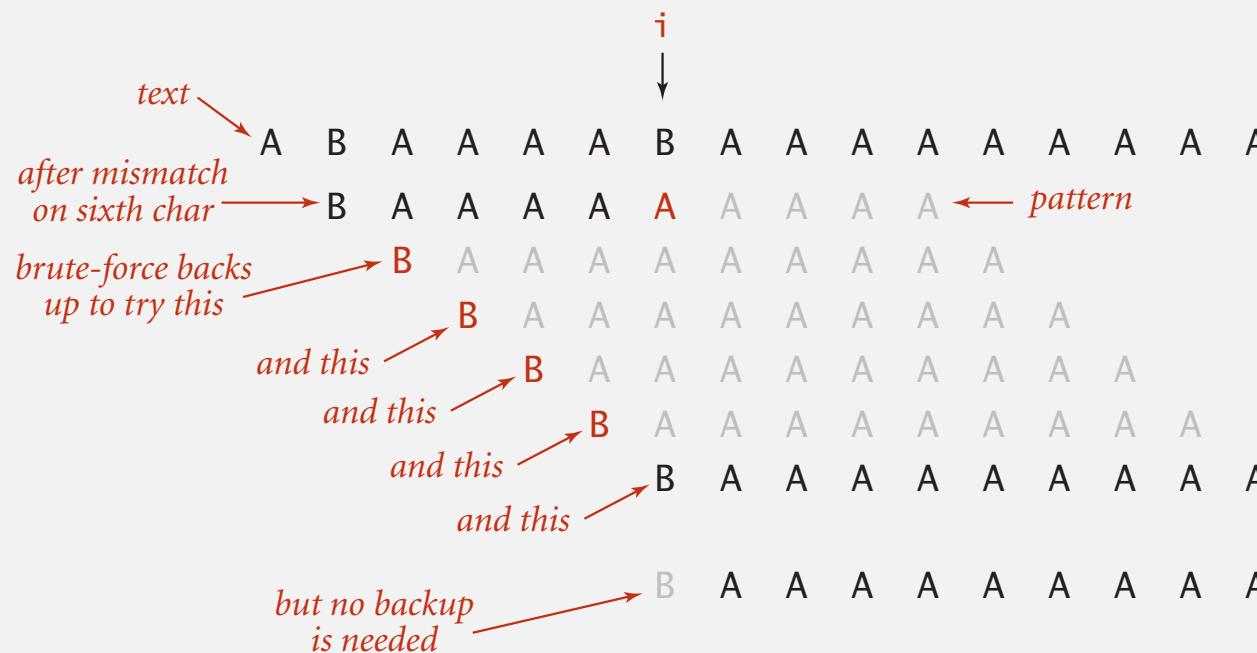
- ▶ *introduction*
- ▶ *brute force*
- ▶ ***Knuth-Morris-Pratt***
- ▶ *Boyer-Moore*
- ▶ *Rabin-Karp*

Knuth-Morris-Pratt substring search

Intuition. Suppose we are searching in text for pattern BAAAAAAAAAA.

- Suppose we match 5 chars in pattern, with mismatch on 6th char.
- We know previous 6 chars in text are BAAAAB.
- Don't need to back up text pointer!

assuming { A, B } alphabet



Knuth-Morris-Pratt algorithm. Clever method to always avoid backup. (!)

Deterministic finite state automaton (DFA)

DFA is abstract string-searching machine.

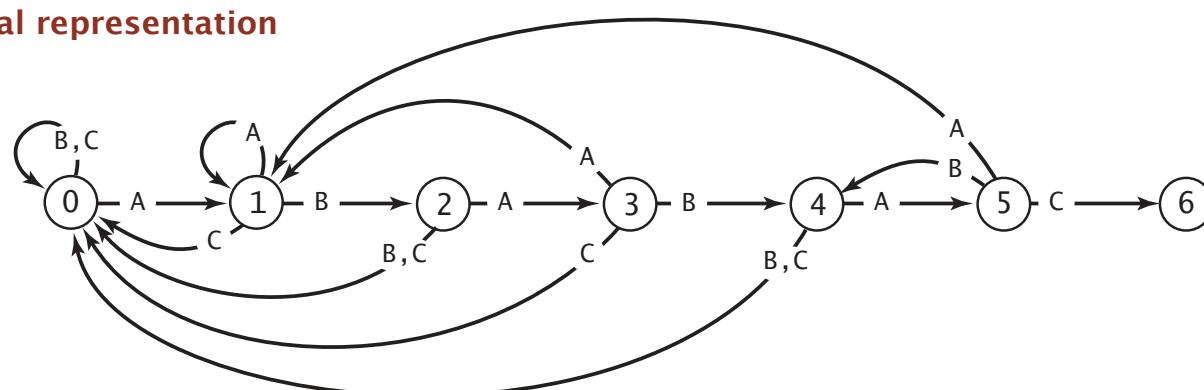
- Finite number of states (including start and halt).
- Exactly one transition for each char in alphabet.
- Accept if sequence of transitions leads to halt state.

internal representation

j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	1	1	3	1	5	1
	0	2	0	4	0	4
	0	0	0	0	0	6

If in state j reading char C:
if j is 6 halt and accept
else move to state dfa[c][j]

graphical representation

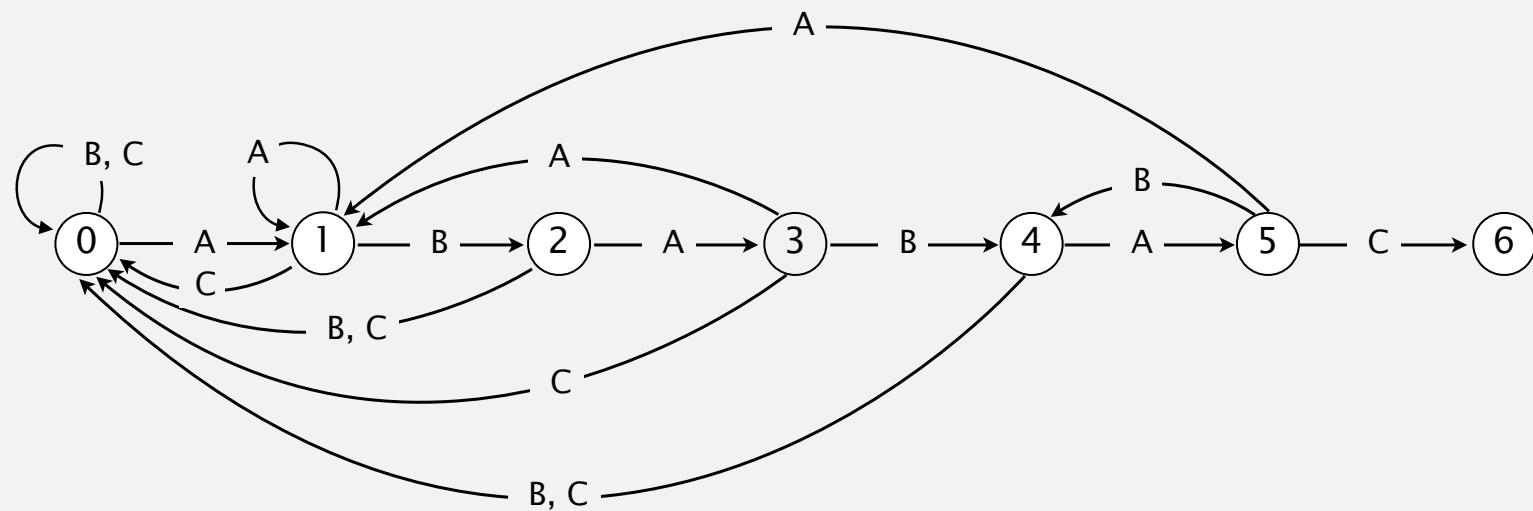


DFA simulation demo

A A B A C A A B A B A C A A



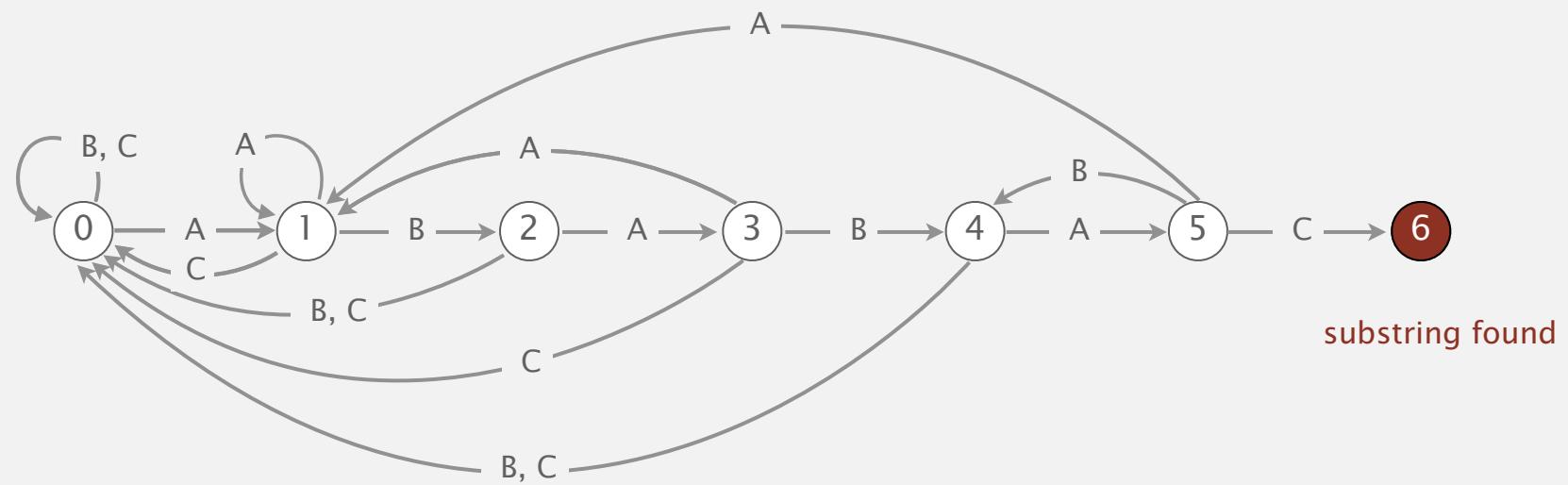
		0	1	2	3	4	5
pat.charAt(j)	A	A	B	A	B	A	C
	B	1	1	3	1	5	1
dfa[][][j]	C	0	2	0	4	0	4
	C	0	0	0	0	0	6



DFA simulation demo

A A B A C A A B A B A C A A
↑

pat.charAt(j)	0	1	2	3	4	5
A	A	B	A	B	A	C
dfa[][][j]	1	1	3	1	5	1
B	0	2	0	4	0	4
C	0	0	0	0	0	6



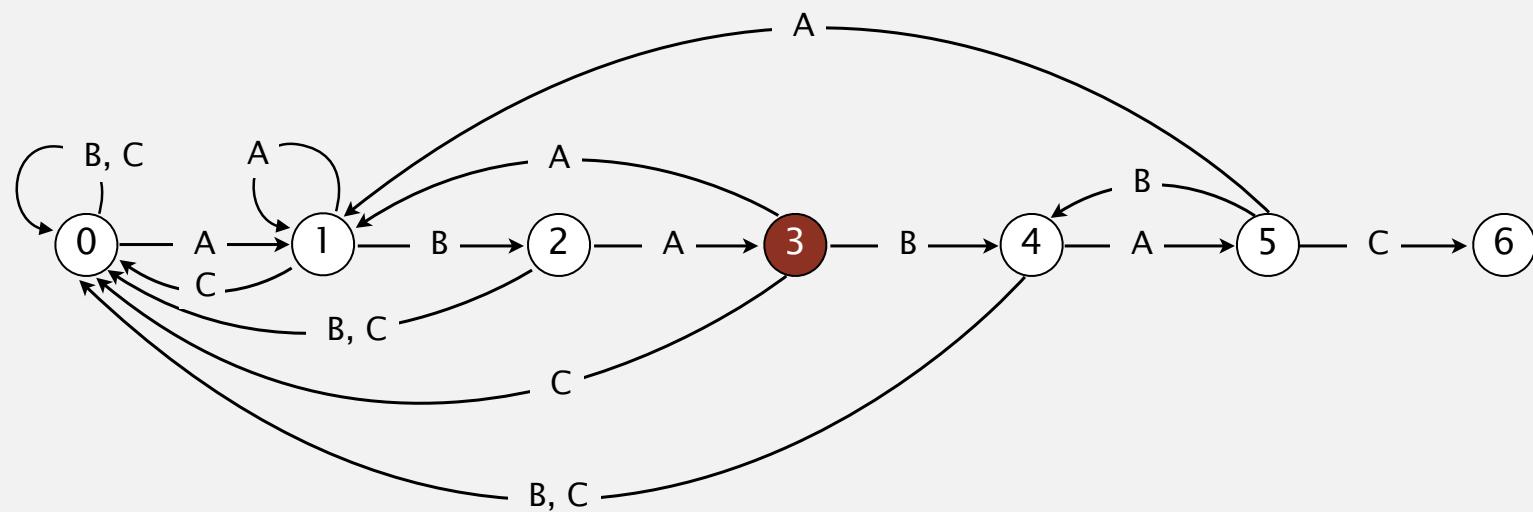
Interpretation of Knuth-Morris-Pratt DFA

Q. What is interpretation of DFA state after reading in $\text{txt}[i]$?

A. State = number of characters in pattern that have been matched.

length of longest prefix of $\text{pat}[]$
that is a suffix of $\text{txt}[0..i]$

Ex. DFA is in state 3 after reading in $\text{txt}[0..6]$.



Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute $\text{dfa}[][]$ from pattern.
- Text pointer i never decrements.

```
public int search(String txt)
{
    int i, j, N = txt.length();
    for (i = 0, j = 0; i < N && j < M; i++)
        j = dfa[txt.charAt(i)][j];           ← no backup
    if (j == M) return i - M;
    else        return N;
}
```

Running time.

- Simulate DFA on text: at most N character accesses.
- Build DFA: how to do efficiently? [warning: tricky algorithm ahead]

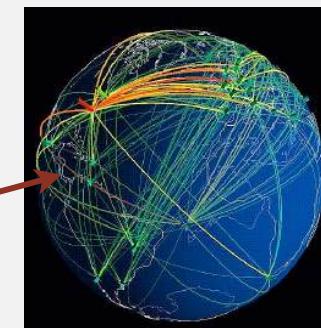
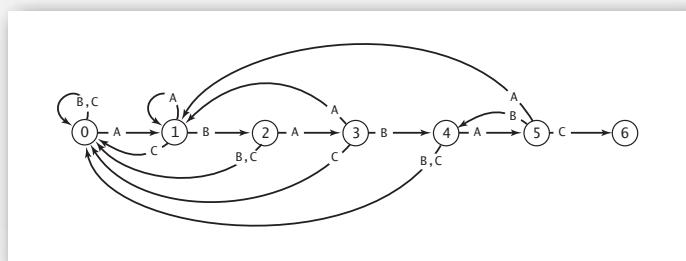
Knuth-Morris-Pratt substring search: Java implementation

Key differences from brute-force implementation.

- Need to precompute `dfa[][]` from pattern.
- Text pointer `i` never decrements.
- Could use **input stream**.

```
public int search(In in)
{
    int i, j;
    for (i = 0, j = 0; !in.isEmpty() && j < M; i++)
        j = dfa[in.readChar()][j];
    if (j == M) return i - M;
    else         return NOT_FOUND;
}
```

no backup



Knuth-Morris-Pratt construction demo

Include one state for each character in pattern (plus accept state).



pat.charAt(j)	0	1	2	3	4	5
A	A	B	A	B	A	C
dfa[][][j]	A	B				
C						

Constructing the DFA for KMP substring search for A B A B A C

0

1

2

3

4

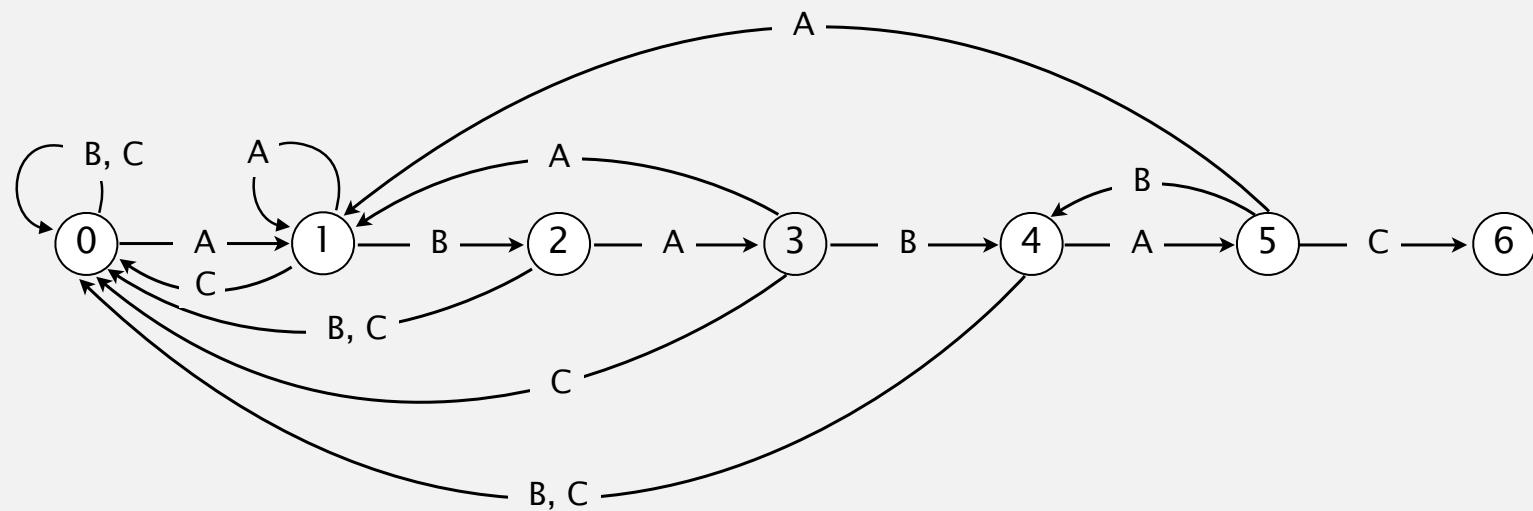
5

6

Knuth-Morris-Pratt construction demo

	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
dfa[][][j]	A	1	1	3	1	1
	B	0	2	0	4	0
	C	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C



How to build DFA from pattern?

Include one state for each character in pattern (plus accept state).



How to build DFA from pattern?

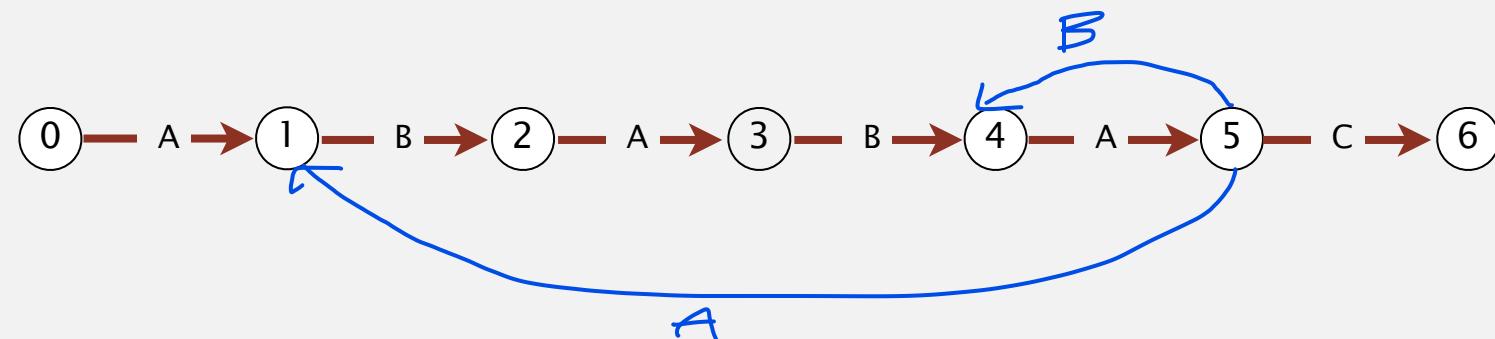
Match transition. If in state j and next char $c == \text{pat.charAt}(j)$, go to $j+1$.

first j characters of pattern
have already been matched

next char matches

now first $j+1$ characters of
pattern have been matched

		0	1	2	3	4	5
pat.charAt(j)	A	1		3		5	
	B		2		4		
	C					6	



How to build DFA from pattern?

Mismatch transition. If in state j and next char $c \neq \text{pat.charAt}(j)$, then the last $j-1$ characters of input are $\text{pat}[1..j-1]$, followed by c .

To compute $\text{dfa}[c][j]$: Simulate $\text{pat}[1..j-1]$ on DFA and take transition c .

Running time. Seems to require j steps.

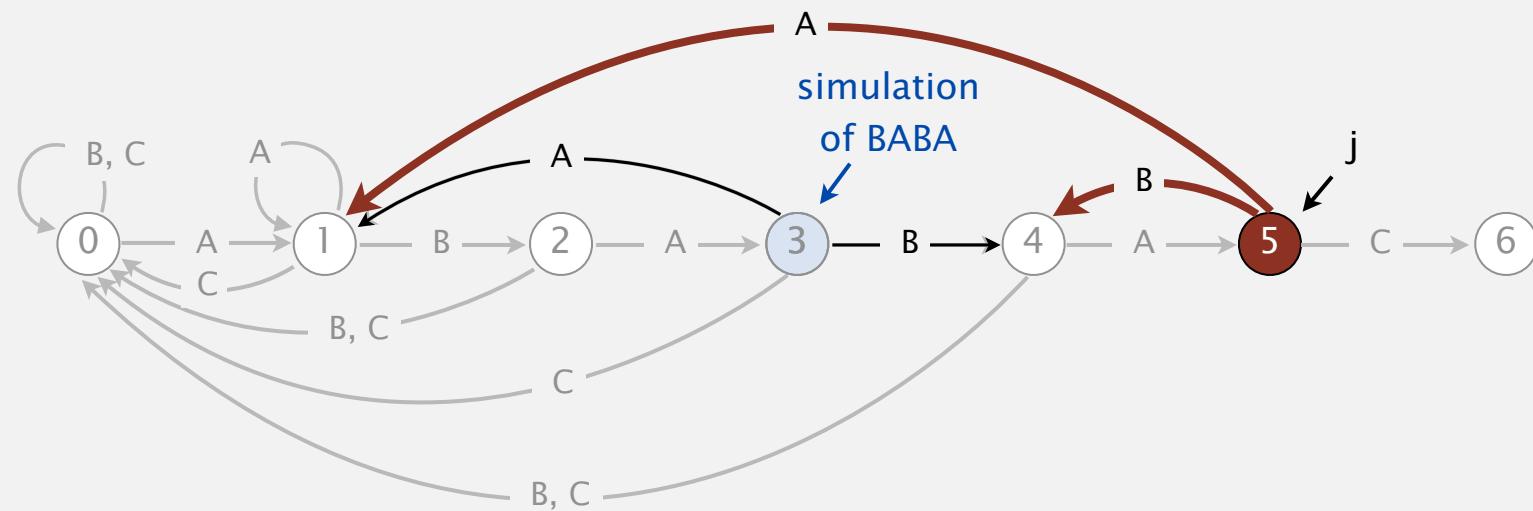
still under construction (!)

Ex. $\text{dfa}['A'][5] = 1$; $\text{dfa}['B'][5] = 4$

simulate BABA;
take transition 'A'
 $= \text{dfa}['A'][3]$

simulate BABA;
take transition 'B'
 $= \text{dfa}['B'][3]$

	j	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C	



How to build DFA from pattern?

Mismatch transition. If in state j and next char $c \neq \text{pat.charAt}(j)$, then the last $j-1$ characters of input are $\text{pat}[1..j-1]$, followed by c .

To compute $\text{dfa}[c][j]$: Simulate $\text{pat}[1..j-1]$ on DFA and take transition c .
Running time. Takes only constant time if we maintain state X .

Ex. $\text{dfa}['A'][5] = 1$; $\text{dfa}['B'][5] = 4$;

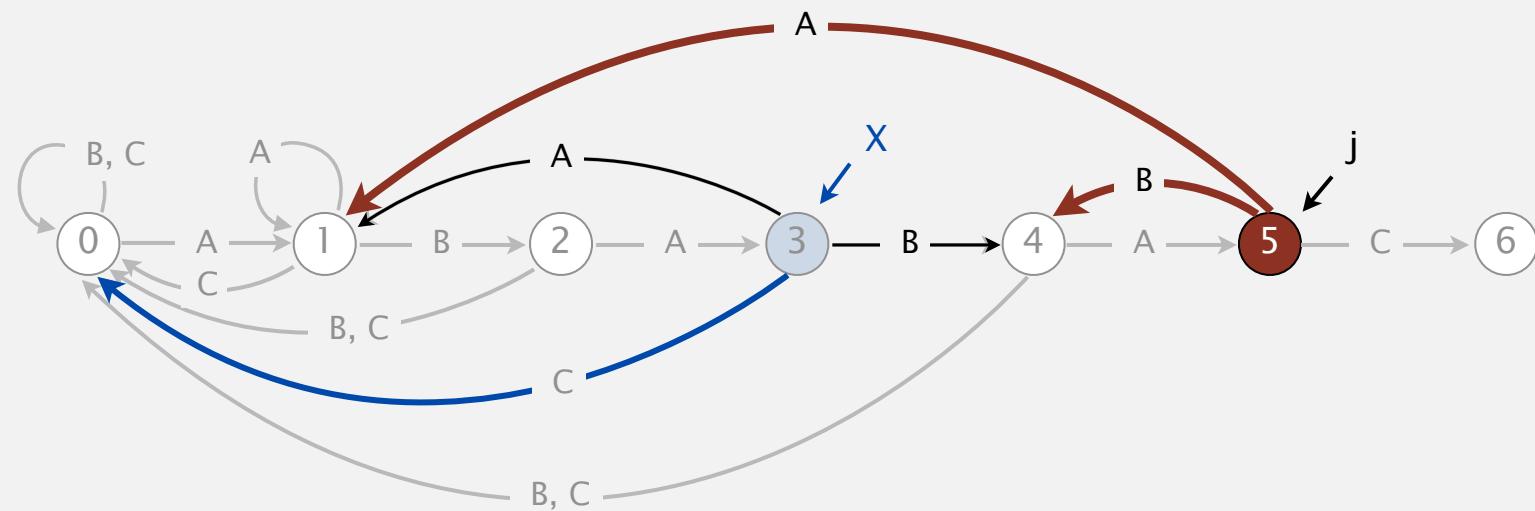
from state X ,
take transition 'A'
 $= \text{dfa}['A'][X]$

from state X ,
take transition 'B'
 $= \text{dfa}['B'][X]$

$X' = 0$

from state X ,
take transition 'C'
 $= \text{dfa}['C'][X]$

0	1	2	3	4	5
A	B	A	B	A	C



Knuth-Morris-Pratt construction demo (in linear time)

Include one state for each character in pattern (plus accept state).



pat.charAt(j)	0	1	2	3	4	5
A	A	B	A	B	A	C
dfa[][][j]	A	B				
	C					

Constructing the DFA for KMP substring search for A B A B A C

0

1

2

3

4

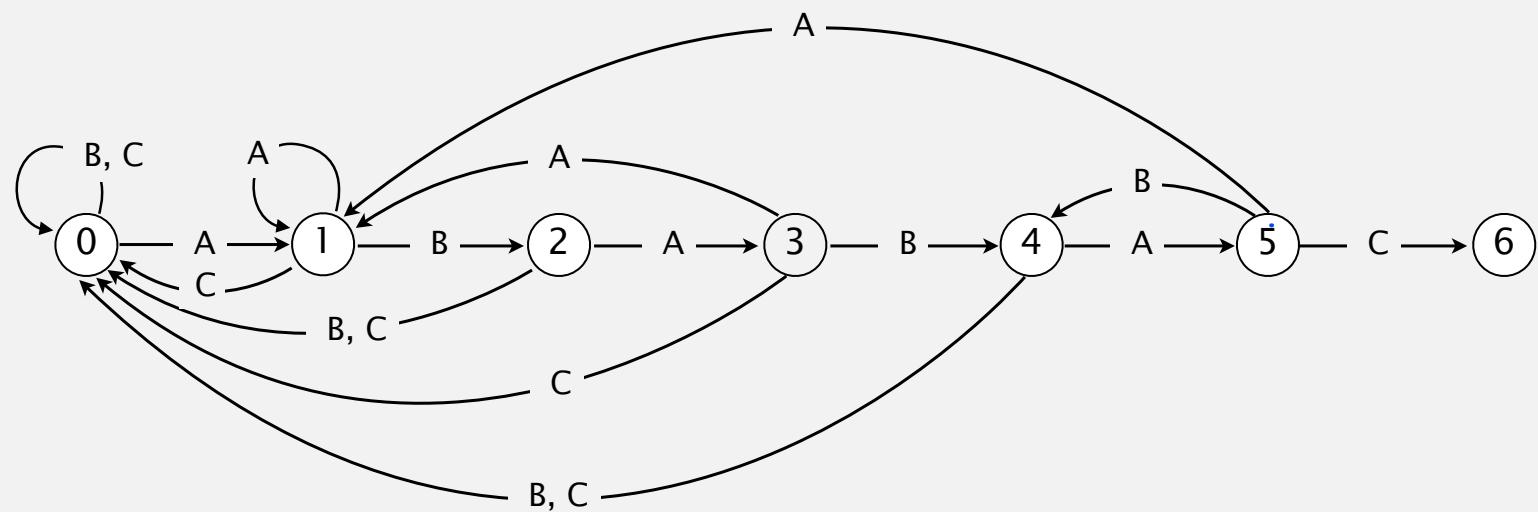
5

6

Knuth-Morris-Pratt construction demo (in linear time)

	0	1	2	3	4	5
pat.charAt(j)	A	B	A	B	A	C
A	1	1	3	1	5	1
dfa[][][j]	B	0	2	0	4	0
C	0	0	0	0	0	6

Constructing the DFA for KMP substring search for A B A B A C



Constructing the DFA for KMP substring search: Java implementation

For each state j :

- Copy $\text{dfa}[][\text{X}]$ to $\text{dfa}[][\text{j}]$ for mismatch case.
- Set $\text{dfa}[\text{pat.charAt(j)}][\text{j}]$ to $\text{j}+1$ for match case.
- Update X .

```
public KMP(String pat)
{
    this.pat = pat;
    M = pat.length();
    dfa = new int[R][M];
    dfa[pat.charAt(0)][0] = 1;
    for (int X = 0, j = 1; j < M; j++)
    {
        for (int c = 0; c < R; c++)
            dfa[c][j] = dfa[c][X]; ← copy mismatch cases
        dfa[pat.charAt(j)][j] = j+1; ← set match case
        X = dfa[pat.charAt(j)][X]; ← update restart state
    }
}
```

Running time. M character accesses (but space/time proportional to $R M$).

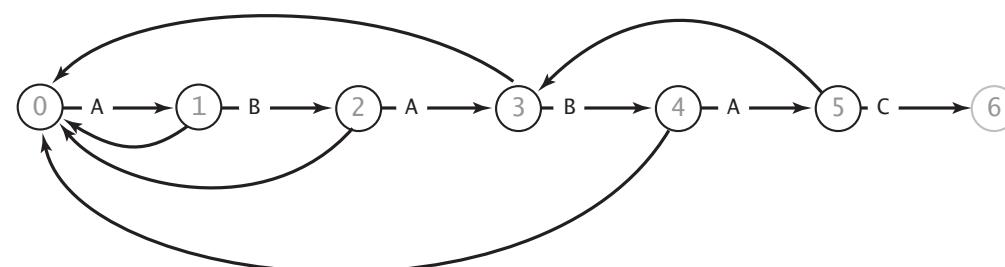
KMP substring search analysis

Proposition. KMP substring search accesses no more than $M + N$ chars to search for a pattern of length M in a text of length N .

Pf. Each pattern char accessed once when constructing the DFA; each text char accessed once (in the worst case) when simulating the DFA.

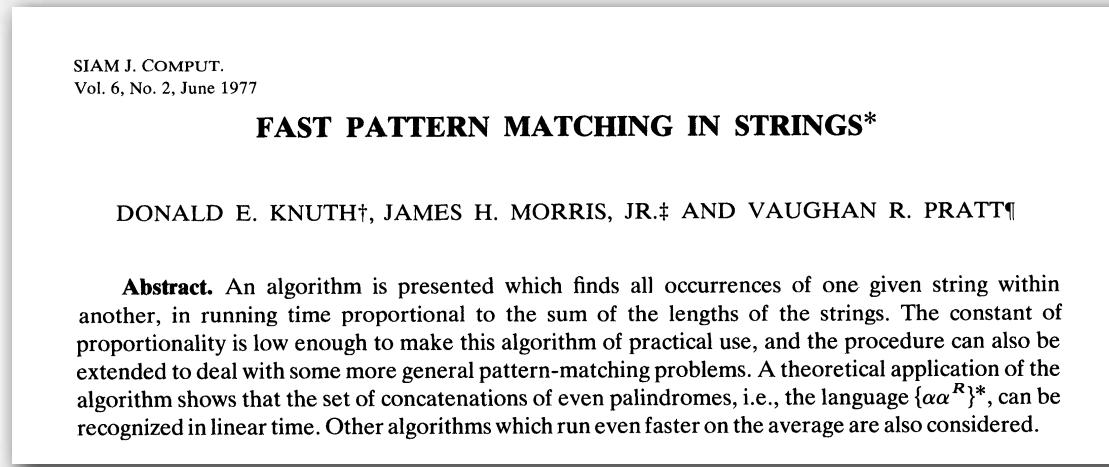
Proposition. KMP constructs `dfa[][]` in time and space proportional to $R M$.

Larger alphabets. Improved version of KMP constructs `nfa[]` in time and space proportional to M .



Knuth-Morris-Pratt: brief history

- Independently discovered by two theoreticians and a hacker.
 - Knuth: inspired by esoteric theorem, discovered linear algorithm
 - Pratt: made running time independent of alphabet size
 - Morris: built a text editor for the CDC 6400 computer
- Theory meets practice.



Don Knuth



Jim Morris



Vaughan Pratt

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

- ▶ *introduction*
- ▶ *brute force*
- ▶ ***Knuth-Morris-Pratt***
- ▶ *Boyer-Moore*
- ▶ *Rabin-Karp*

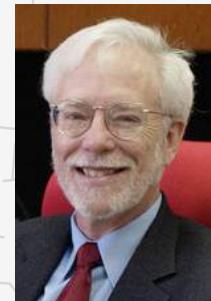
Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

- ▶ *introduction*
- ▶ *brute force*
- ▶ *Knuth-Morris-Pratt*
- ▶ **Boyer-Moore**
- ▶ *Rabin-Karp*



Robert Boyer J. Strother Moore

Boyer-Moore: mismatched character heuristic

Intuition.

- Scan characters in pattern from right to left.
- Can skip as many as M text chars when finding one not in the pattern.

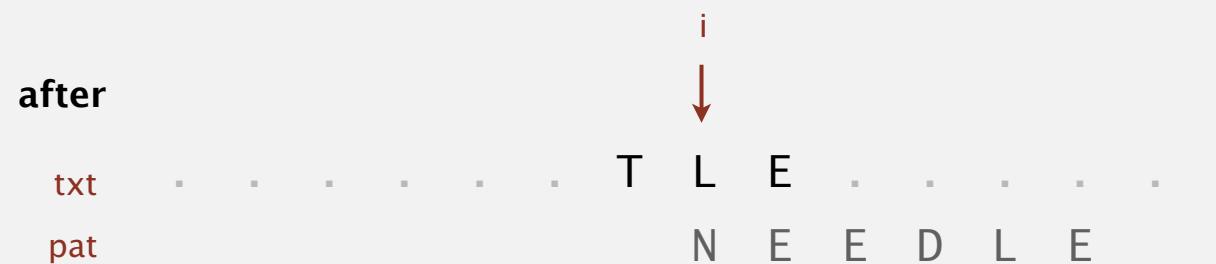
i	j	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
		F	I	N	D	I	N	A	H	A	Y	S	T	A	C	K	N	E	E	D	L	E	I	N	A
		text →																							
0	5	N	E	E	D	L	E	← pattern																	
5	5								N	E	E	D	L	E											
11	4														N	E	E	D	L	E					
15	0																	N	E	E	D	L	E		

return i = 15

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 1. Mismatch character not in pattern.

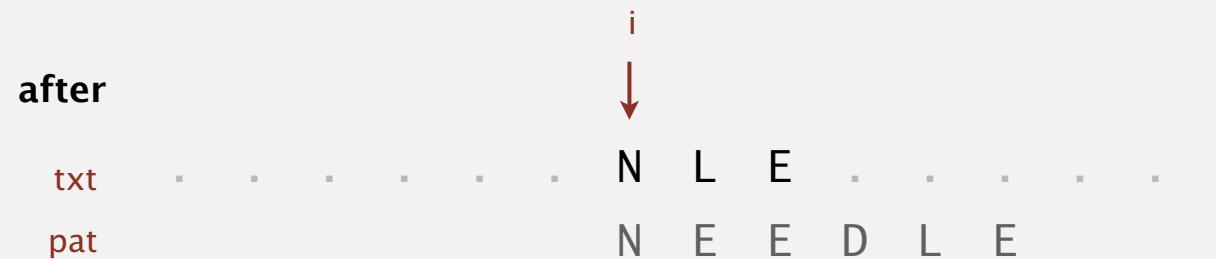


mismatch character 'T' not in pattern: increment i one character beyond 'T'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2a. Mismatch character in pattern.



mismatch character 'N' in pattern: align text 'N' with rightmost pattern 'N'

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).



mismatch character 'E' in pattern: align text 'E' with rightmost pattern 'E' ?

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

Case 2b. Mismatch character in pattern (but heuristic no help).

before															
txt	E	L	E
pat				N	E	E	D	L	E						

after															
txt	E	L	E
pat				N	E	E	D	L	E						

mismatch character 'E' in pattern: increment i by 1

Boyer-Moore: mismatched character heuristic

Q. How much to skip?

A. Precompute index of rightmost occurrence of character c in pattern
(-1 if character not in pattern).

```
right = new int[R];
for (int c = 0; c < R; c++)
    right[c] = -1;
for (int j = 0; j < M; j++)
    right[pat.charAt(j)] = j;
```

c	N	E	E	D	L	E	right[c]
	0	1	2	3	4	5	
A	-1	-1	-1	-1	-1	-1	-1
B	-1	-1	-1	-1	-1	-1	-1
C	-1	-1	-1	-1	-1	-1	-1
D	-1	-1	-1	-1	3	3	3
E	-1	-1	1	2	2	5	5
...							-1
L	-1	-1	-1	-1	-1	4	4
M	-1	-1	-1	-1	-1	-1	-1
N	-1	0	0	0	0	0	0
...							-1

Boyer-Moore skip table computation

Boyer-Moore: Java implementation

```
public int search(String txt)
{
    int N = txt.length();
    int M = pat.length();
    int skip;
    for (int i = 0; i <= N-M; i += skip)
    {
        skip = 0;
        for (int j = M-1; j >= 0; j--)
        {
            if (pat.charAt(j) != txt.charAt(i+j))
            {
                skip = Math.max(1, j - right[txt.charAt(i+j)]);
                break;
            }
        }
        if (skip == 0) return i; ← match
    }
    return N;
}
```

compute
skip value

in case other term is nonpositive

Boyer-Moore: analysis

Property. Substring search with the Boyer-Moore mismatched character heuristic takes about $\sim N/M$ character compares to search for a pattern of length M in a text of length N .

sublinear!

Worst-case. Can be as bad as $\sim MN$.

i	skip	0	1	2	3	4	5	6	7	8	9
	txt →	B	B	B	B	B	B	B	B	B	B
0	0	A	B	B	B	B	B	B	B	B	B
1	1		A	B	B	B	B				
2	1			A	B	B	B	B			
3	1				A	B	B	B	B		
4	1					A	B	B	B	B	
5	1						A	B	B	B	B

Boyer-Moore variant. Can improve worst case to $\sim 3N$ character compares by adding a KMP-like rule to guard against repetitive patterns.

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

- ▶ *introduction*
- ▶ *brute force*
- ▶ *Knuth-Morris-Pratt*
- ▶ ***Boyer-Moore***
- ▶ *Rabin-Karp*

Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

- ▶ *introduction*
- ▶ *brute force*
- ▶ *Knuth-Morris-Pratt*
- ▶ *Boyer-Moore*
- ▶ ***Rabin-Karp***



Michael Rabin, Turing Award '76

Dick Karp, Turing Award '85

Rabin-Karp fingerprint search

Basic idea = modular hashing.

- Compute a hash of pattern characters 0 to $M - 1$.
- For each i , compute a hash of text characters i to $M + i - 1$.
- If pattern hash = text substring hash, check for a match.

pat.charAt(i)														
i	0	1	2	3	4									
	2	6	5	3	5									
	2	6	5	3	5	% 997	=	613						
txt.charAt(i)														
i	0	1	2	3	4	5	6	7	8	9	10	11	12	13
	3	1	4	1	5	9	2	6	5	3	5	8	9	7
0	3	1	4	1	5	% 997	=	508						
1		1	4	1	5	9	% 997	=	201					
2			4	1	5	9	2	% 997	=	715				
3				1	5	9	2	6	% 997	=	971			
4					5	9	2	6	5	% 997	=	442		
5						9	2	6	5	3	% 997	=	929	
6	← return	i = 6					2	6	5	3	5	% 997	=	613
														match

Efficiently computing the hash function

Modular hash function. Using the notation t_i for `txt.charAt(i)`, we wish to compute

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0 \pmod{Q}$$

Intuition. M -digit, base- R integer, modulo Q .

Horner's method. Linear-time method to evaluate degree- M polynomial.

pat.charAt()					
i	0	1	2	3	4
	2	6	5	3	5
0	2	% 997 = 2			
1	2	6	% 997 = (2*10 + 6) % 997 = 26	<i>R</i>	<i>Q</i>
2	2	6	5 % 997 = (26*10 + 5) % 997 = 265		
3	2	6	5 3 % 997 = (265*10 + 3) % 997 = 659		
4	2	6	5 3 5 % 997 = (659*10 + 5) % 997 = 613		

```
// Compute hash for M-digit key
private long hash(String key, int M)
{
    long h = 0;
    for (int j = 0; j < M; j++)
        h = (R * h + key.charAt(j)) % Q;
    return h;
}
```

Efficiently computing the hash function

Challenge. How to efficiently compute x_{i+1} given that we know x_i .

$$x_i = t_i R^{M-1} + t_{i+1} R^{M-2} + \dots + t_{i+M-1} R^0$$

$$x_{i+1} = t_{i+1} R^{M-1} + t_{i+2} R^{M-2} + \dots + t_{i+M} R^0$$

Key property. Can update hash function in constant time!

$$x_{i+1} = (x_i - t_i R^{M-1}) R + t_{i+M}$$

↑ ↑ ↑ ↑
current subtract multiply add new
value leading digit by radix trailing digit (can precompute R^{M-1})

i	...	2	3	4	5	6	7	...
current value		1	4	1	5	9	2	6 5
new value		4	1	5	9	2	6	5

\Rightarrow text

4	1	5	9	2	current value
-	4	0	0	0	
1	5	9	2		subtract leading digit
*	1	0			multiply by radix
1	5	9	2	0	
			+	6	add new trailing digit
1	5	9	2	6	new value

Rabin-Karp substring search example

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	
	3	1	4	1	5	9	2	6	5	3	5	8	9	7	9	3	
0	3	1	%	997	=	3											
1	3	1	%	997	=	(3*10 + 1) % 997 = 31	Q										
2	3	1	4	%	997	=	(31*10 + 4) % 997 = 314										
3	3	1	4	1	%	997	=	(314*10 + 1) % 997 = 150									
4	3	1	4	1	5	%	997	=	(150*10 + 5) % 997 = 508	RM	R						
5	1	4	1	5	9	%	997	=	((508 + 3*(997 - 30))*10 + 9) % 997 = 201								
6	4	1	5	9	2	%	997	=	((201 + 1*(997 - 30))*10 + 2) % 997 = 715								
7	1	5	9	2	6	%	997	=	((715 + 4*(997 - 30))*10 + 6) % 997 = 971								
8	5	9	2	6	5	%	997	=	((971 + 1*(997 - 30))*10 + 5) % 997 = 442								
9	9	2	6	5	3	%	997	=	((442 + 5*(997 - 30))*10 + 3) % 997 = 929							match	
10	← return i-M+1 = 6	2	6	5	3	5	%	997	=	((929 + 9*(997 - 30))*10 + 5) % 997 = 613							↓

Rabin-Karp: Java implementation

```
public class RabinKarp
{
    private long patHash;          // pattern hash value
    private int M;                 // pattern length
    private long Q;                // modulus
    private int R;                 // radix
    private long RM;               //  $R^{M-1} \bmod Q$ 

    public RabinKarp(String pat) {
        M = pat.length();
        R = 256;
        Q = LongRandomPrime();           ← a large prime
                                         (but avoid overflow)

        RM = 1;
        for (int i = 1; i <= M-1; i++)
            RM = (R * RM) % Q;
        patHash = hash(pat, M);
    }

    private long hash(String key, int M)
    { /* as before */ }

    public int search(String txt)
    { /* see next slide */ }
}
```

a large prime
(but avoid overflow)

← precompute $R^{M-1} \bmod Q$

Rabin-Karp: Java implementation (continued)

Monte Carlo version. Return match if hash match.

```
public int search(String txt)
{
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
    {
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) % Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        if (patHash == txtHash) return i - M + 1;
    }
    return N;
}
```

check for hash collision
using rolling hash function

Las Vegas version. Check for substring match if hash match;
continue search if false collision.

Rabin-Karp analysis

Theory. If Q is a sufficiently large random prime (about MN^2), then the probability of a false collision is about $1/N$.

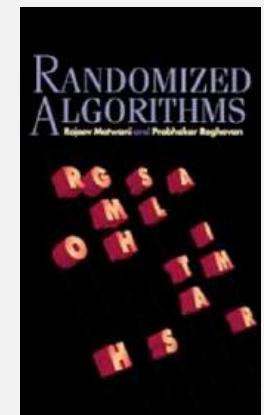
Practice. Choose Q to be a large prime (but not so large to cause overflow). Under reasonable assumptions, probability of a collision is about $1/Q$.

Monte Carlo version.

- Always runs in linear time.
- Extremely likely to return correct answer (but not always!).

Las Vegas version.

- Always returns correct answer.
- Extremely likely to run in linear time (but worst case is MN).



Rabin-Karp fingerprint search

Advantages.

- Extends to 2d patterns.
- Extends to finding multiple patterns.

Disadvantages.

- Arithmetic ops slower than char compares.
- Las Vegas version requires backup.
- Poor worst-case guarantee.

Q. How would you extend Rabin-Karp to efficiently search for any one of P possible patterns in a text of length N ?



Substring search cost summary

Cost of searching for an M -character pattern in an N -character text.

algorithm	version	operation count		backup in input?	correct?	extra space
		guarantee	typical			
brute force	—	MN	$1.1 N$	yes	yes	1
Knuth-Morris-Pratt	<i>full DFA</i> (Algorithm 5.6)	$2N$	$1.1 N$	no	yes	MR
	<i>mismatch transitions only</i>	$3N$	$1.1 N$	no	yes	M
Boyer-Moore	<i>full algorithm</i>	$3N$	N/M	yes	yes	R
	<i>mismatched char heuristic only</i> (Algorithm 5.7)	MN	N/M	yes	yes	R
Rabin-Karp [†]	<i>Monte Carlo</i> (Algorithm 5.8)	$7N$	$7N$	no	yes^{\dagger}	1
	<i>Las Vegas</i>	$7N^{\dagger}$	$7N$	yes	yes	1

[†] probabilistic guarantee, with uniform hash function

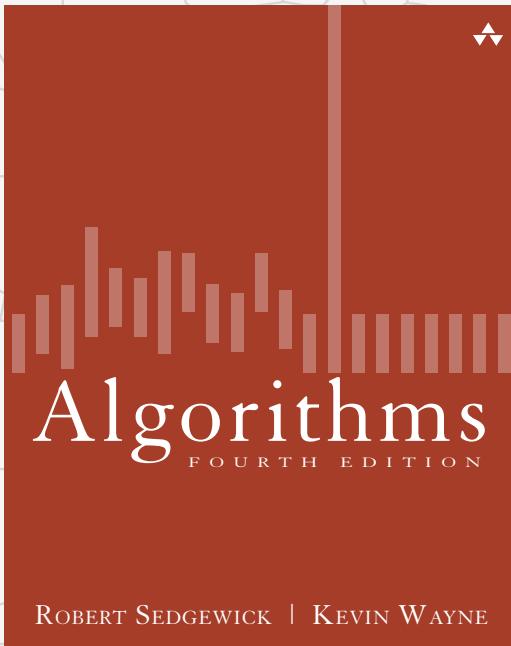
Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

- ▶ *introduction*
- ▶ *brute force*
- ▶ *Knuth-Morris-Pratt*
- ▶ *Boyer-Moore*
- ▶ ***Rabin-Karp***



<http://algs4.cs.princeton.edu>

5.3 SUBSTRING SEARCH

- ▶ *introduction*
- ▶ *brute force*
- ▶ *Knuth-Morris-Pratt*
- ▶ *Boyer-Moore*
- ▶ *Rabin-Karp*