



The Role of the Scientific Method in Software Development

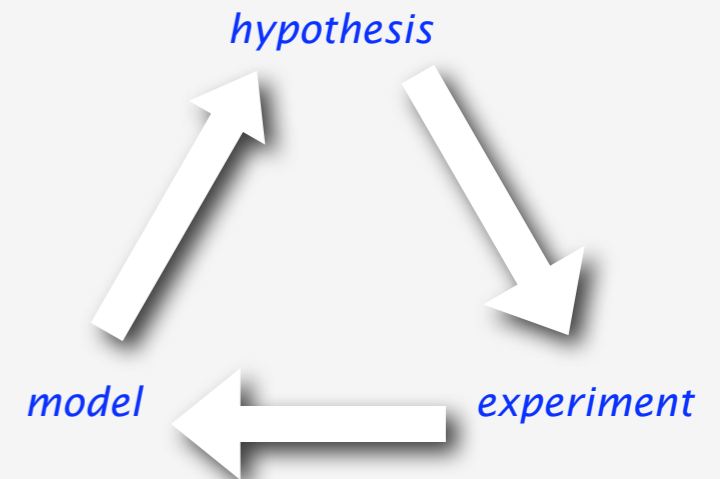
**Robert Sedgewick
Princeton University**

The scientific method

is **necessary** in algorithm design and software development

Scientific method

- **create a model** describing natural world
- use model to **develop hypotheses**
- **run experiments** to validate hypotheses
- refine model and repeat



1950s



2000s



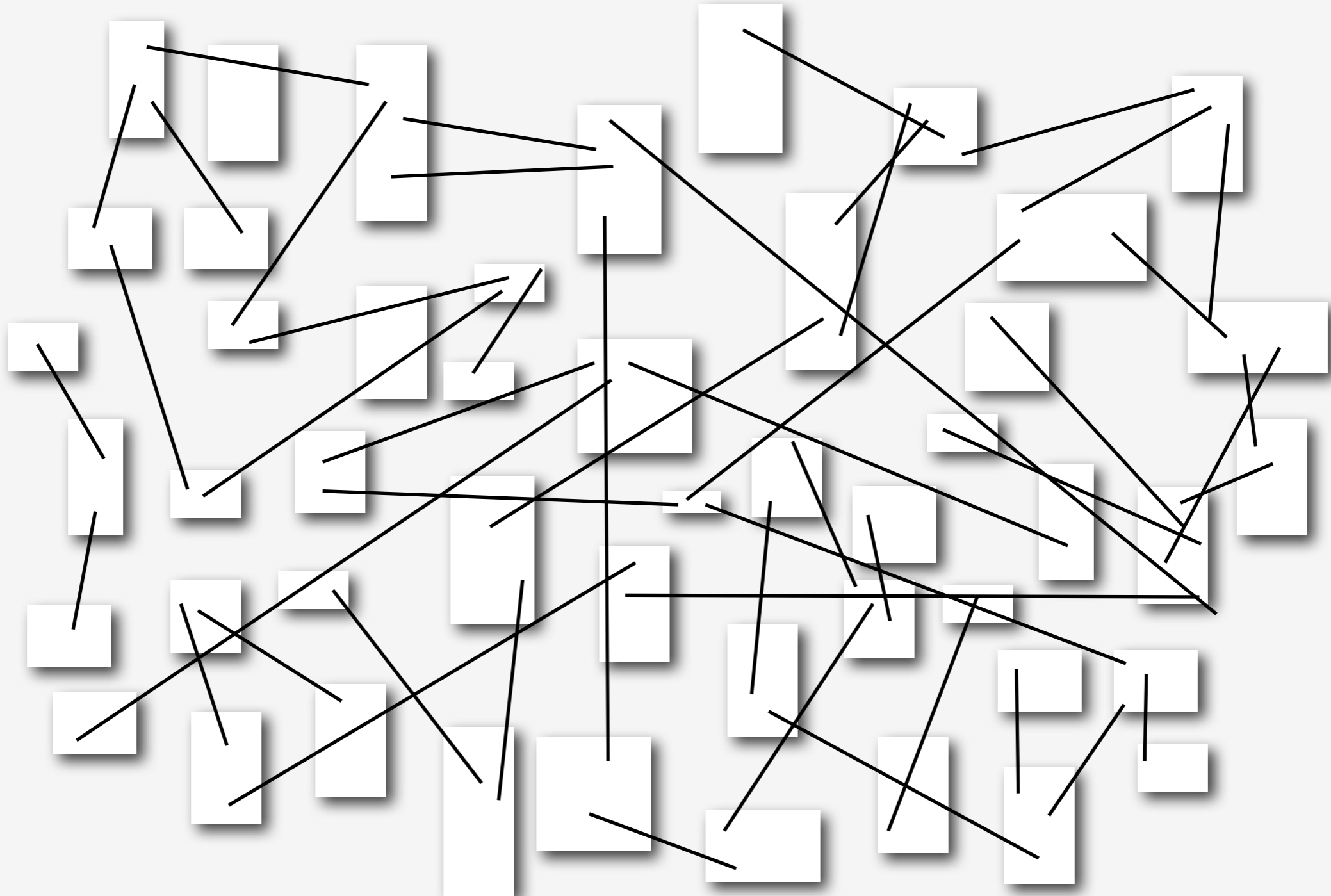
Algorithm designer who does not experiment gets lost in abstraction

Software developer who ignores cost risks catastrophic consequences

First hypothesis (needs checking)

Modern software development requires huge amounts of code

- Introduction
- Motivating example
- Grid graphs
- Search methods
- Small world graphs
- Conclusion

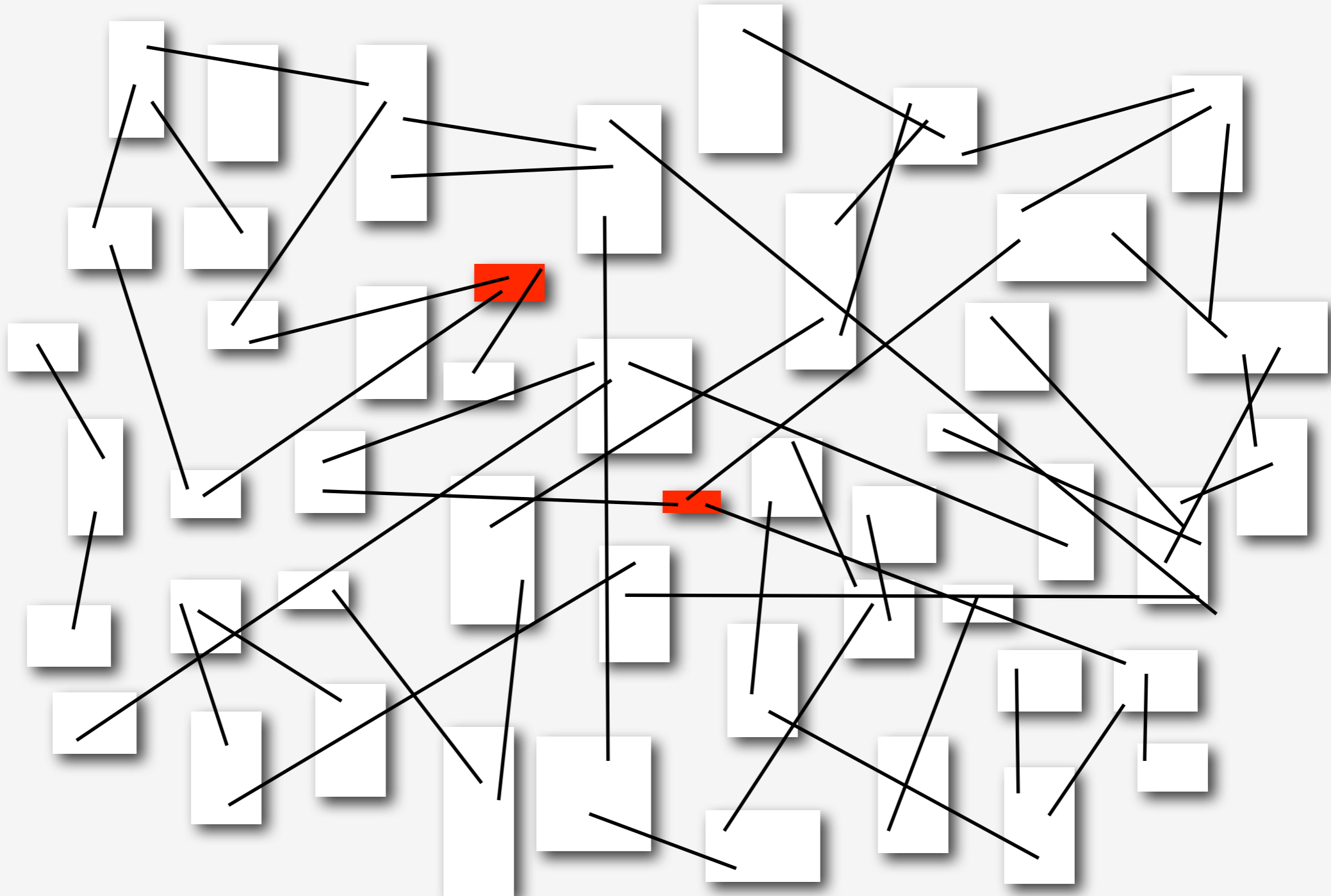


First hypothesis (needs checking)

- Introduction
- Motivating example
- Grid graphs
- Search methods
- Small world graphs
- Conclusion

Modern software development requires huge amounts of code
but

performance-critical code implements relatively few fundamental algorithms



Warmup: random number generation

- Introduction
- Motivating example
- Grid graphs
- Search methods
- Small world graphs
- Conclusion

Problem: write a program to generate random numbers

model: classical probability and statistics

hypothesis: frequency values should be uniform

weak experiment:

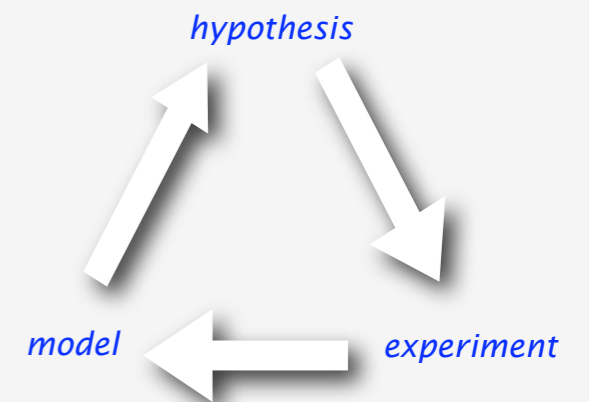
- generate random numbers
- check for uniform frequencies

better experiment:

- generate random numbers
- use χ^2 test to check frequency values against uniform distribution

better hypotheses/experiments still needed

- many documented disasters
- active area of scientific research
- applications: simulation, cryptography
- connects to core issues in theory of computation



```
int k = 0;
while ( true )
    System.out.print(k++ % V);
```

0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 ...
random?

```
int k = 0;
while ( true ) {
    k = k*1664525 + 1013904223);
    System.out.print(k % V);
}
```

textbook algorithm that flunks χ^2 test

Warmup (continued)

Q. Is a given sequence of numbers random?

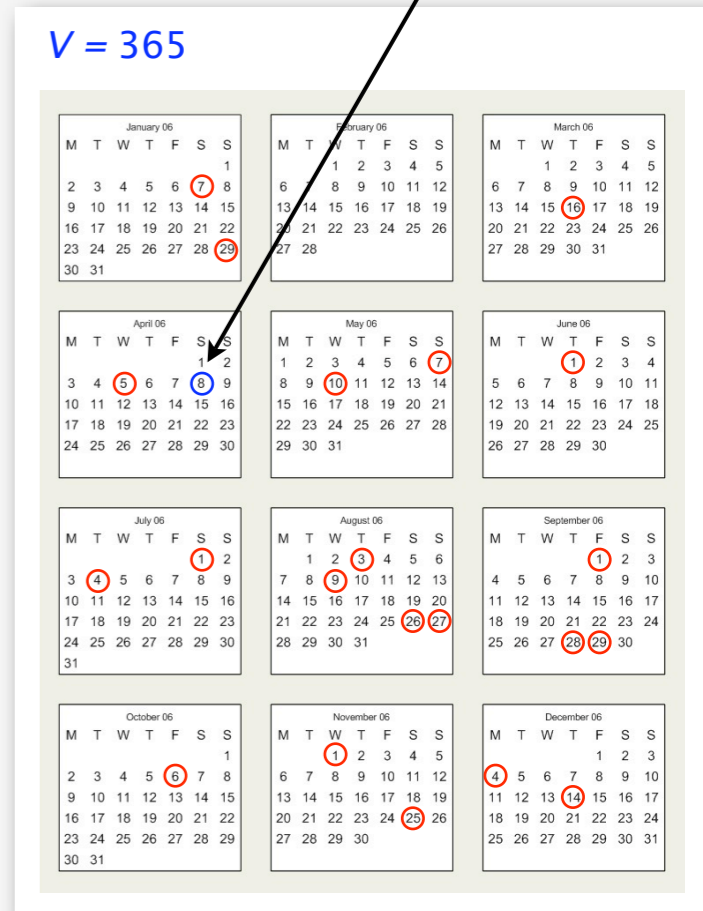
A. No.

Q. Does a given sequence exhibit some property that random number sequences exhibit?

average probes
until duplicate
is about 24

Birthday paradox

Average count of random numbers generated until a duplicate happens is about $\sqrt{\pi V/2}$



Example of a better experiment:

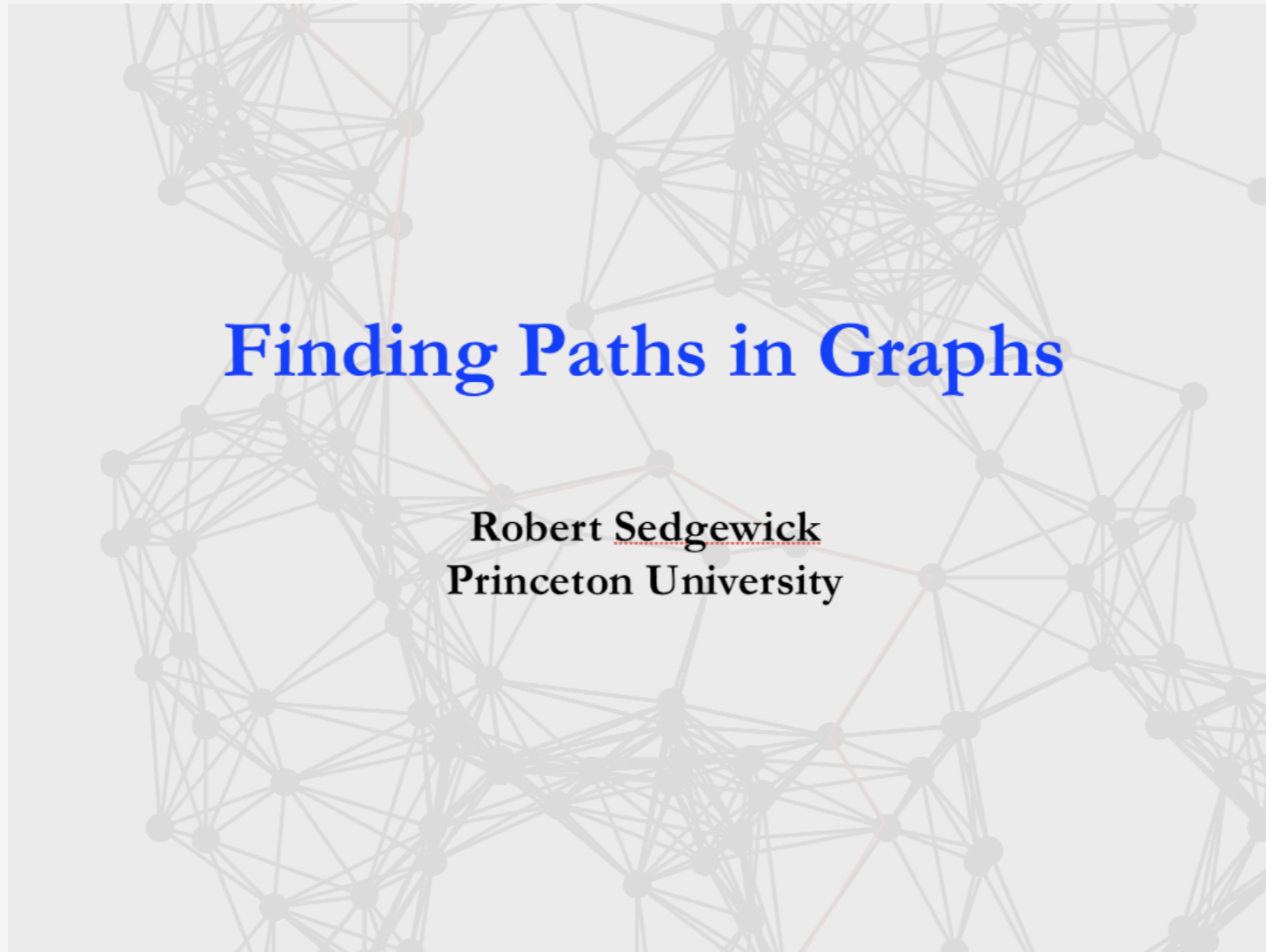
- generate numbers until duplicate
- check that count is close to $\sqrt{\pi V/2}$
- even better: repeat many times, check against distribution
- still better: run many similar tests for other properties

“Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin” — John von Neumann

Detailed example: paths in graphs

Introduction
Motivating example
Grid graphs
Search methods
Small world graphs
Conclusion

A lecture within a lecture



Finding an st -path in a graph

is a fundamental operation that demands understanding

Introduction

Motivating example

Grid graphs

Search methods

Small world graphs

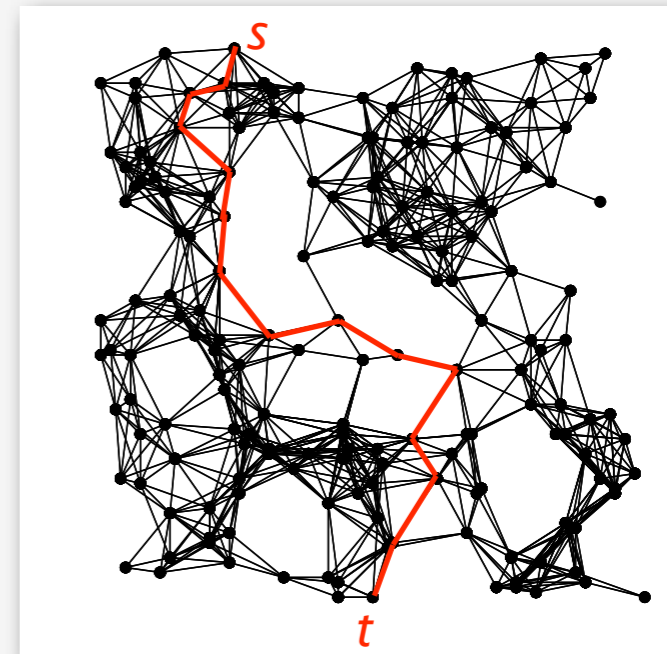
Conclusion

Ground rules for this talk

- work in progress (more questions than answers)
- basic research
- save “deep dive” for the right problem

Applications

- graph-based optimization models
- networks
- percolation
- computer vision
- social networks
- (many more)



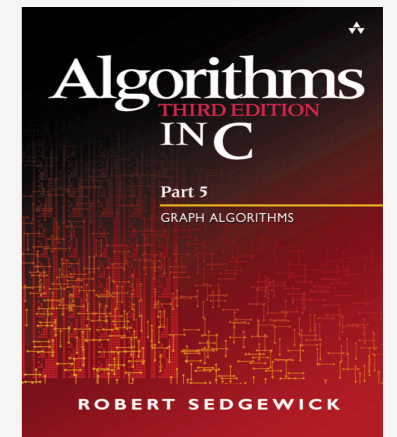
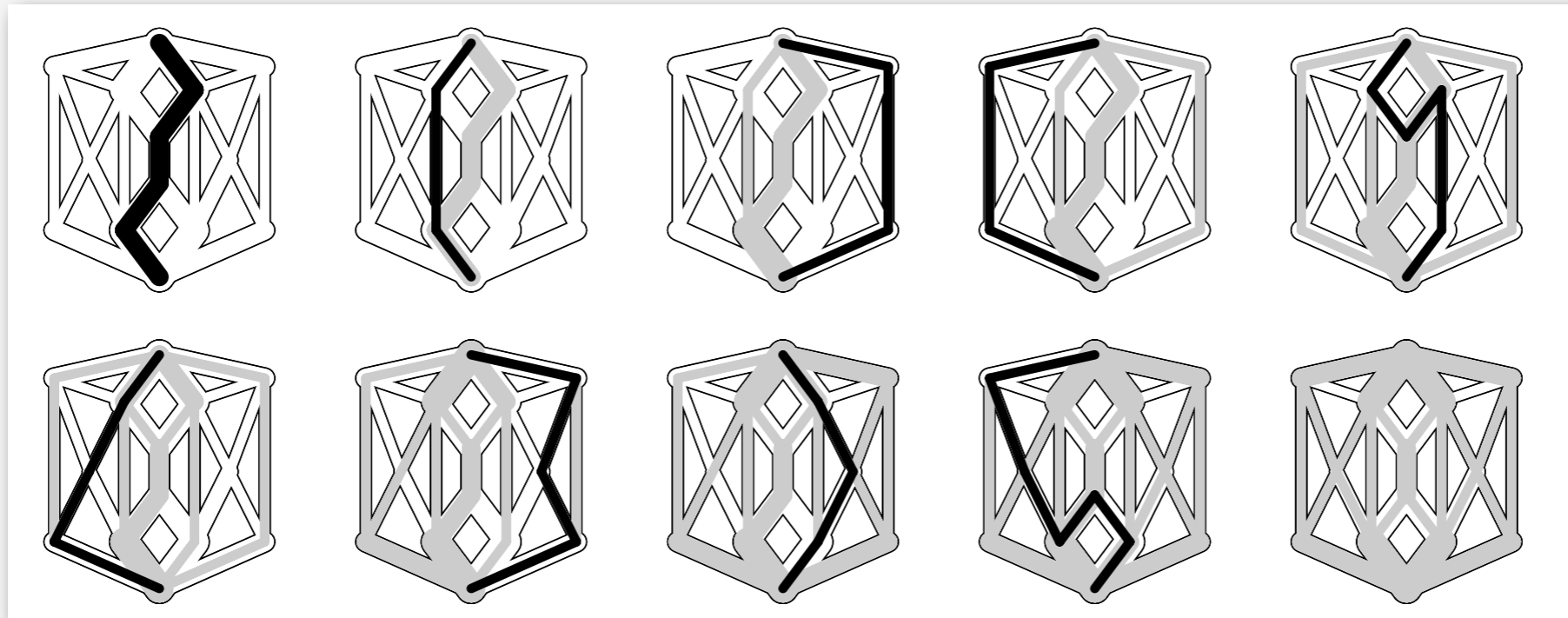
Basic research

- fundamental abstract operation with numerous applications
- worth doing even if no immediate application
- resist temptation to prematurely study impact

Motivating example: maxflow

Ford-Fulkerson maxflow scheme

- find any s-t path in a (residual) graph
- augment flow along path (may create or delete edges)
- iterate until no path exists



Goal: compare performance of two basic implementations

- **shortest** augmenting path
- **maximum capacity** augmenting path

Key steps in analysis

- How many augmenting paths?
- What is the cost of finding each path?

research literature

this talk

Motivating example: max flow

Compare performance of Ford-Fulkerson implementations

- shortest augmenting path
- maximum-capacity augmenting path

Graph parameters

- number of vertices V
- number of edges E
- maximum capacity C

How many augmenting paths?

	<i>worst case upper bound</i>
<i>shortest</i>	$VE/2$ VC
<i>max capacity</i>	$2E \lg C$

How many steps to find each path? E (worst-case upper bound)

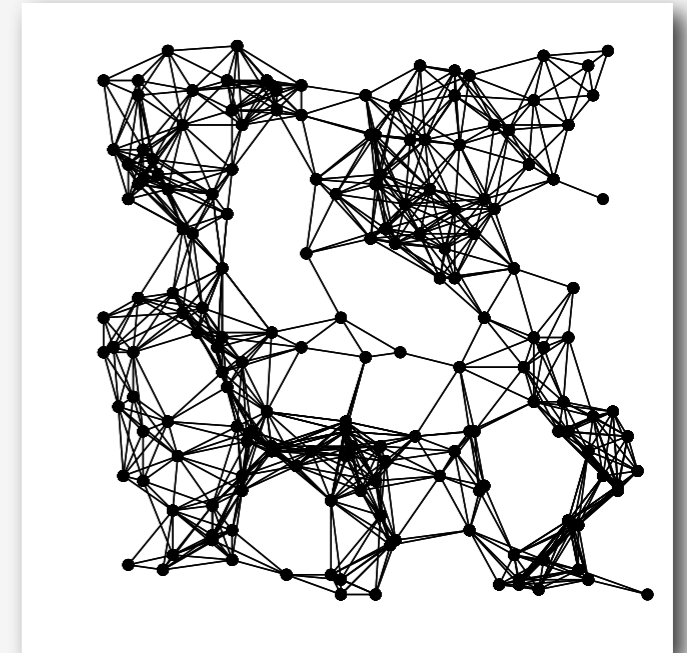
Motivating example: max flow

Compare performance of Ford-Fulkerson implementations

- shortest augmenting path
- maximum-capacity augmenting path

Graph parameters **for example graph**

- number of vertices $V = 177$
- number of edges $E = 2000$
- maximum capacity $C = 100$



How many augmenting paths?

	<i>worst case upper bound</i>	<i>for example</i>
<i>shortest</i>	$VE/2$ VC	177,000 17,700
<i>max capacity</i>	$2E \lg C$	26,575

How many steps to find each path? 2000 (worst-case upper bound)

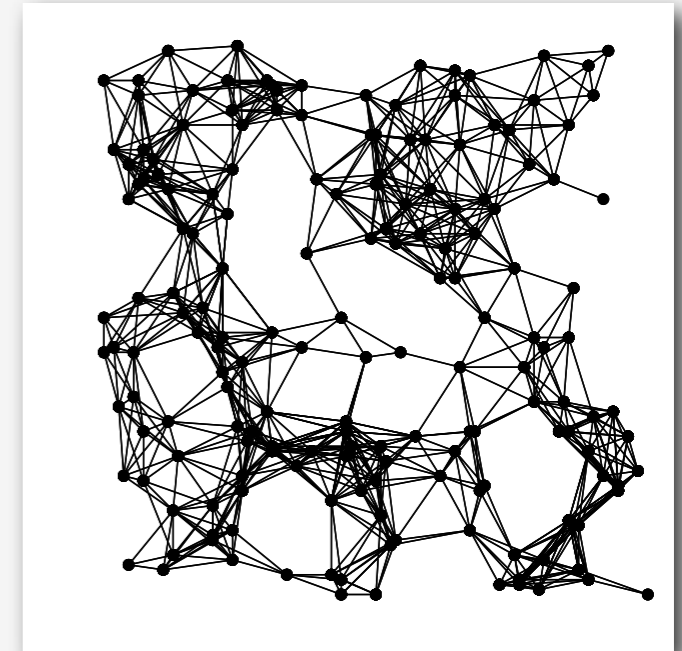
Motivating example: max flow

Compare performance of Ford-Fulkerson implementations

- shortest augmenting path
- maximum-capacity augmenting path

Graph parameters **for example graph**

- number of vertices $V = 177$
- number of edges $E = 2000$
- maximum capacity $C = 100$



How many augmenting paths?

	<i>worst case upper bound</i>	<i>for example</i>	<i>actual</i>
<i>shortest</i>	$VE/2$ VC	177,000 17,700	37
<i>max capacity</i>	$2E \lg C$	26,575	7

How many steps to find each path? **< 20, on average**

*total is a
factor of a million high
for thousand-node graphs!*

Motivating example: max flow

Compare performance of Ford-Fulkerson implementations

- shortest augmenting path
- maximum-capacity augmenting path

Graph parameters

- number of vertices V
- number of edges E
- maximum capacity C

Total number of steps?

	<i>worst case upper bound</i>
<i>shortest</i>	$VE^2/2$ VEC
<i>max capacity</i>	$2E^2 \lg C$

WARNING: The Algorithm General has determined that using such results to predict performance or to compare algorithms may be hazardous.

Motivating example: lessons

Goals of algorithm analysis

- **predict** performance (running time)
- **guarantee** that cost is below specified bounds



worst-case bounds

Common wisdom

- random graph models are unrealistic
- average-case analysis of algorithms is too difficult
- worst-case performance bounds are the standard

Unfortunate truth about worst-case bounds

- often useless for prediction (fictional)
- often useless for guarantee (too high)
- often misused to compare algorithms

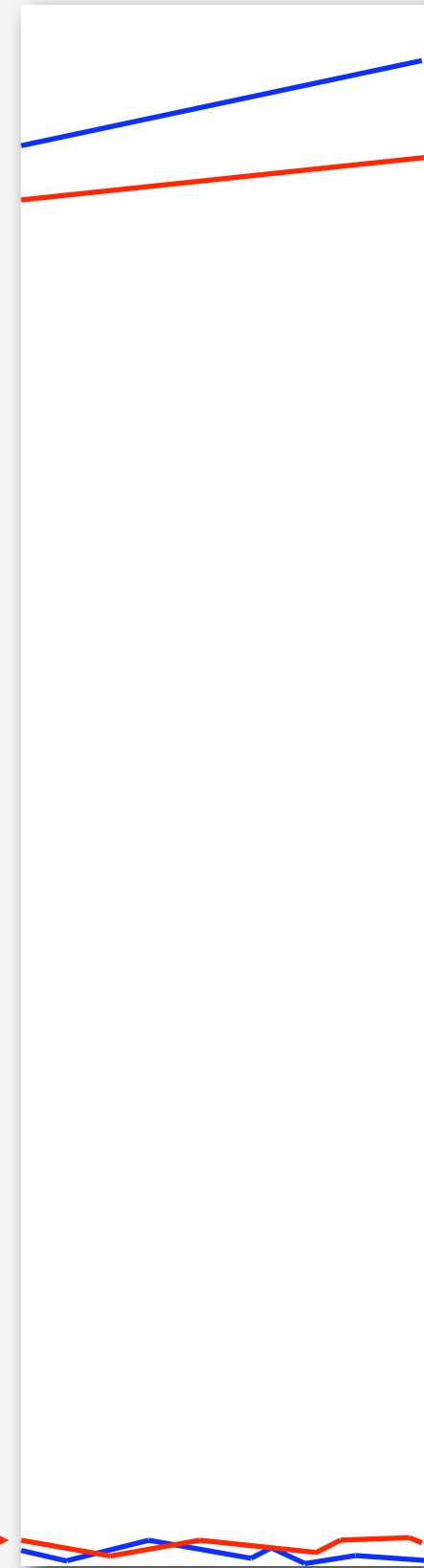
Bounds are useful in many applications:

Open problem: Do better!



which ones??

actual costs

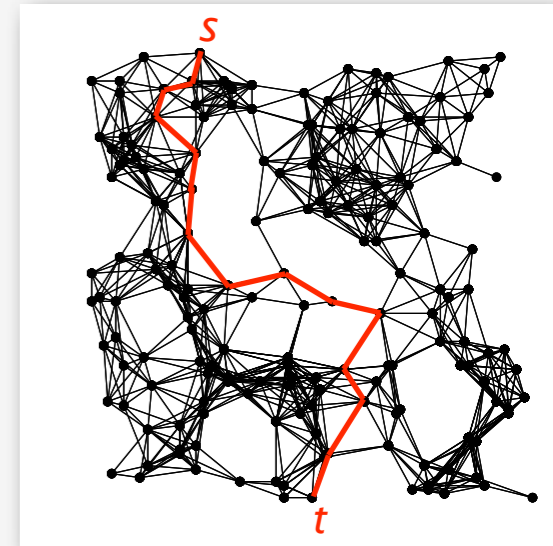


Finding an st -path in a graph

is a basic operation in a great many applications

Q. What is the **best** way to find an st -path in a graph?

- A.** Several well-studied textbook algorithms are known
- **Breadth-first search (BFS)** finds the shortest path
 - **Depth-first search (DFS)** is easy to implement
 - **Union-Find (UF)** needs two passes

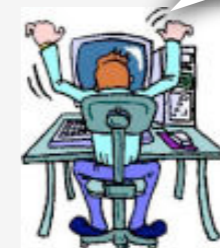


BUT

- all three process all E edges in the worst case
- diverse kinds of graphs are encountered in practice

Worst-case analysis is useless for predicting performance

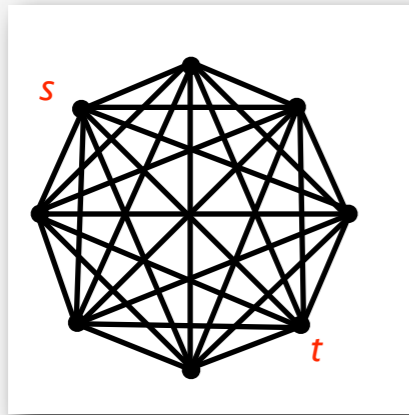
Which basic algorithm should a practitioner use?



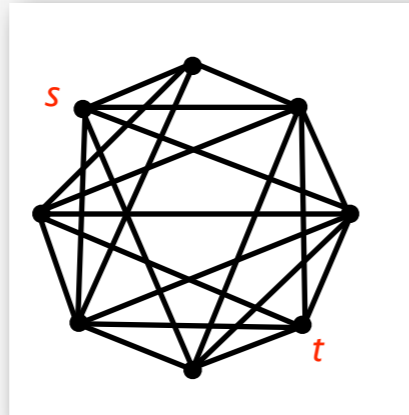
Grid graphs

Algorithm performance depends on the graph model

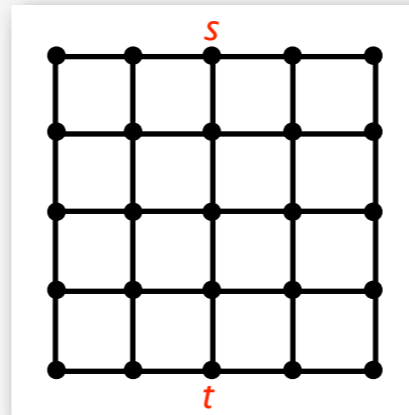
complete



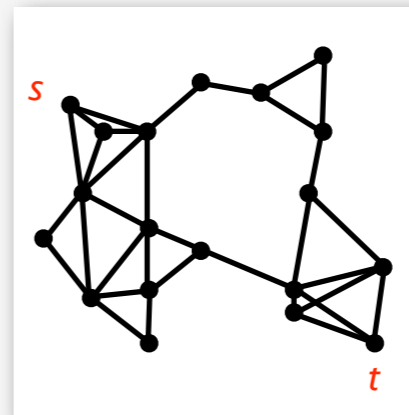
random



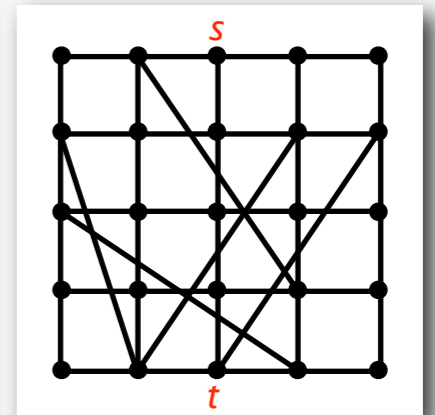
grid



neighbor



small-world



... (many appropriate candidates)

Initial choice: **grid graphs**

- sufficiently challenging to be interesting
- found in practice (or similar to graphs found in practice)
- scalable
- potential for analysis

Ground rules

- algorithms should work for all graphs
- algorithms should **not** use any special properties of the model

if vertices have positions we can find short paths quickly with A (stay tuned)*



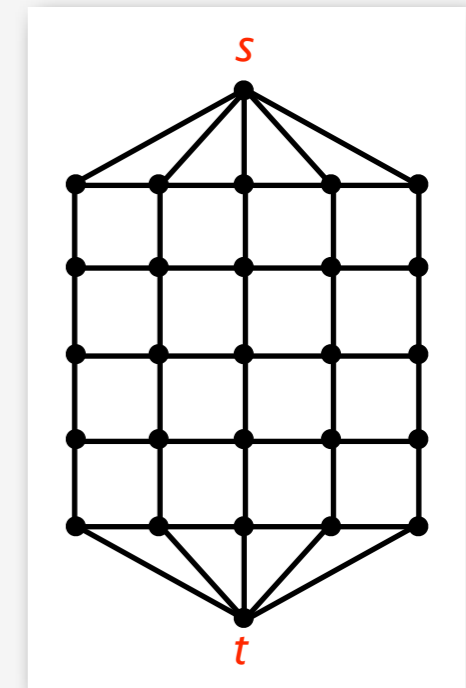
Applications of grid graphs

- Introduction
- Motivating example
- Grid graphs
- Search methods
- Small world graphs
- Conclusion

- conductivity*
- concrete*
- granular materials*
- porous media*
- polymers*
- forest fires*
- epidemics*
- Internet*
- resistor networks*
- evolution*
- social influence*
- Fermi paradox*
- fractal geometry*
- stereo vision*
- image restoration*
- object segmentation*
- scene reconstruction*

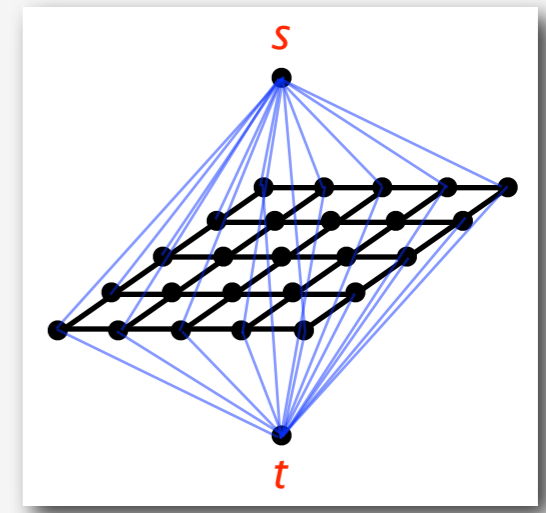
Example 1: Statistical physics

- percolation model
- extensive simulations
- some analytic results
- arbitrarily huge graphs



Example 2: Image processing

- model pixels in images
- maxflow/mincut
- energy minimization
- huge graphs



Finding an st -path in a grid graph

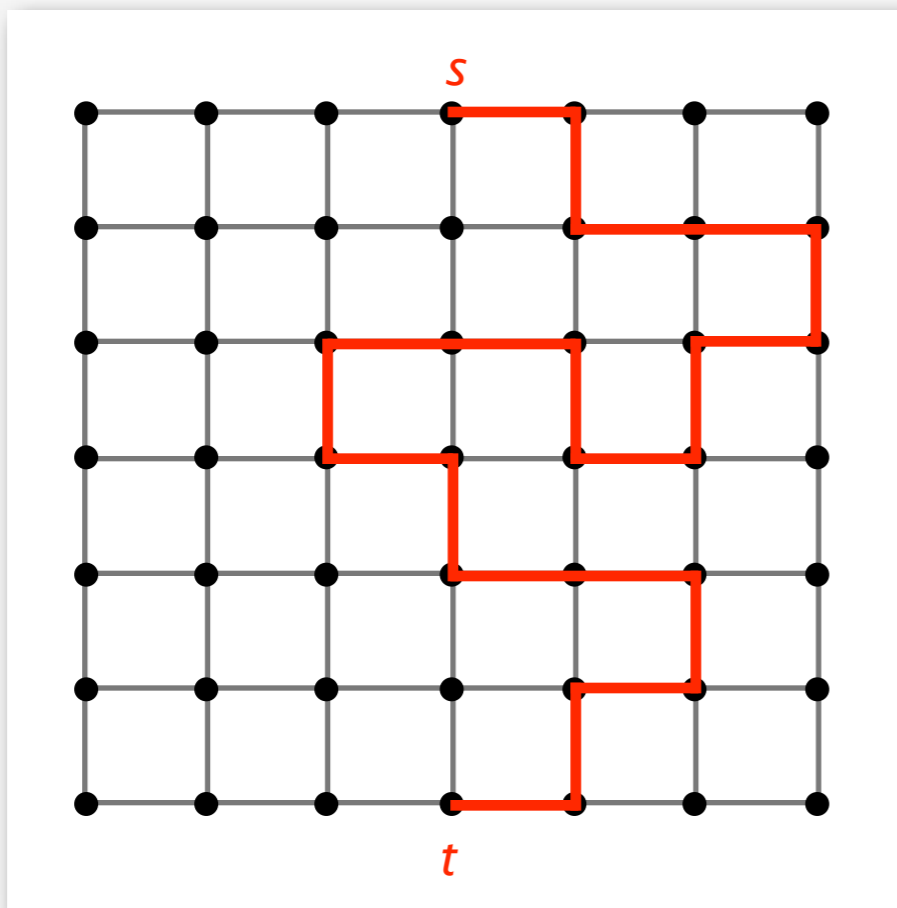
M by M grid of vertices

undirected edges connecting each vertex to its HV neighbors

source vertex s at center of top boundary

destination vertex t at center of bottom boundary

Find **any** path connecting s to t



M^2 vertices
about $2M^2$ edges

M	vertices	edges
7	49	84
15	225	420
31	961	1860
63	3969	7812
127	16129	32004
255	65025	129540
511	261121	521220

Cost measure: number of graph edges examined

Finding an st -path in a grid graph

Similar problems are covered extensively in the literature

- Percolation
- Random walk
- Nonselfintersecting paths in grids
- Graph covering

Which basic algorithm should a practitioner use to find a path in a grid-like graph?



Finding an st -path in a grid graph

Elementary algorithms are found in textbooks

- Depth-first search (DFS)
- Breadth-first search (BFS)
- Union-find

Which basic algorithm should a practitioner use to find a path in a grid-like graph?

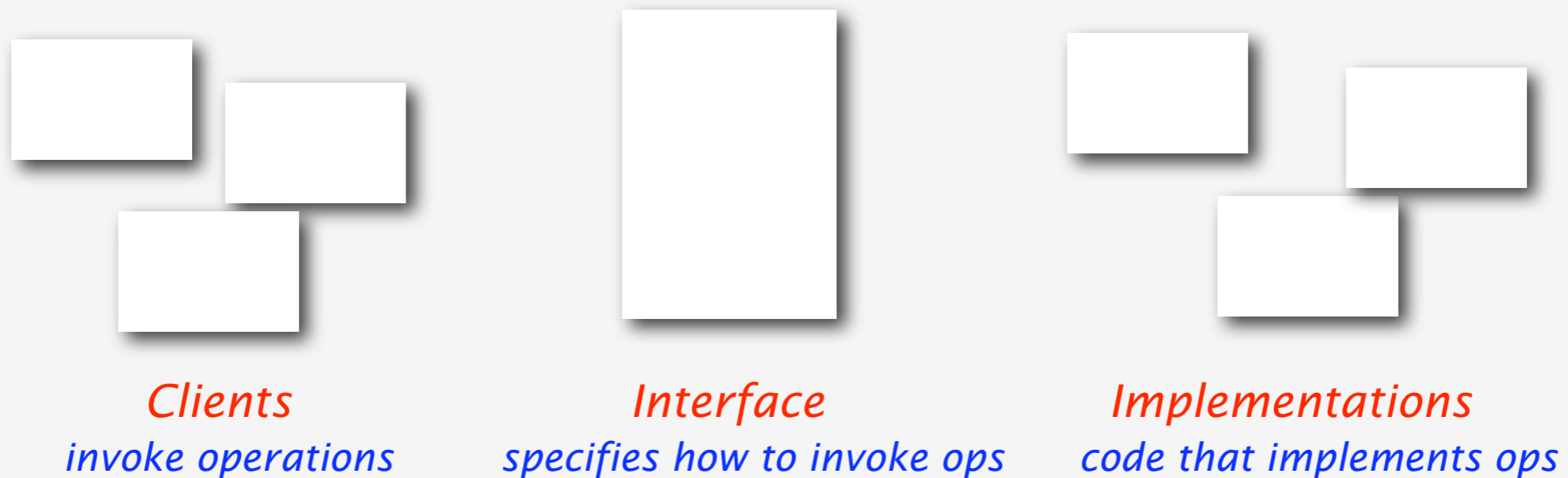


Abstract data types

separate clients from implementations

A **data type** is a set of values and the operations performed on them

An **abstract data type** is a data type whose representation is hidden



Implementation should **not** be tailored to particular client

Develop implementations that work properly for **all** clients
Study their performance for the client at hand

Graph abstract data type

Vertices are integers between 0 and $V-1$

Edges are vertex pairs

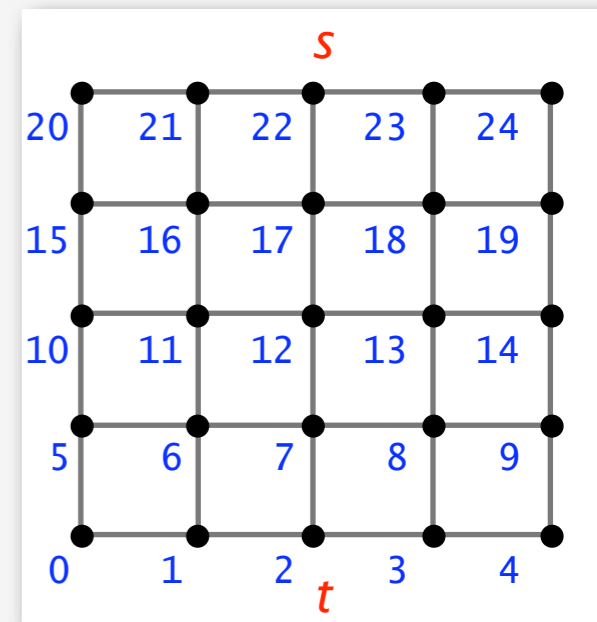
Graph ADT implements

- **Graph(Edge[])** to construct graph from array of edges
- **findPath(int, int)** to conduct search from s to t
- **st(int)** to return predecessor of v on path found

Example: client code for grid graphs

```
int e = 0;
Edge[] a = new Edge[E];
for (int i = 0; i < V; i++)
{
    if (i < V-M) a[e++] = new Edge(i, i+M);
    if (i >= M) a[e++] = new Edge(i, i-M);
    if ((i+1) % M != 0) a[e++] = new Edge(i, i+1);
    if (i % M != 0) a[e++] = new Edge(i, i-1);
}
GRAPH G = new GRAPH(a);
G.findPath(V-1-M/2, M/2);
for (int k = t; k != s; k = G.st(k))
    System.out.println(s + "-" + t);
```

$M = 5$



DFS: standard implementation

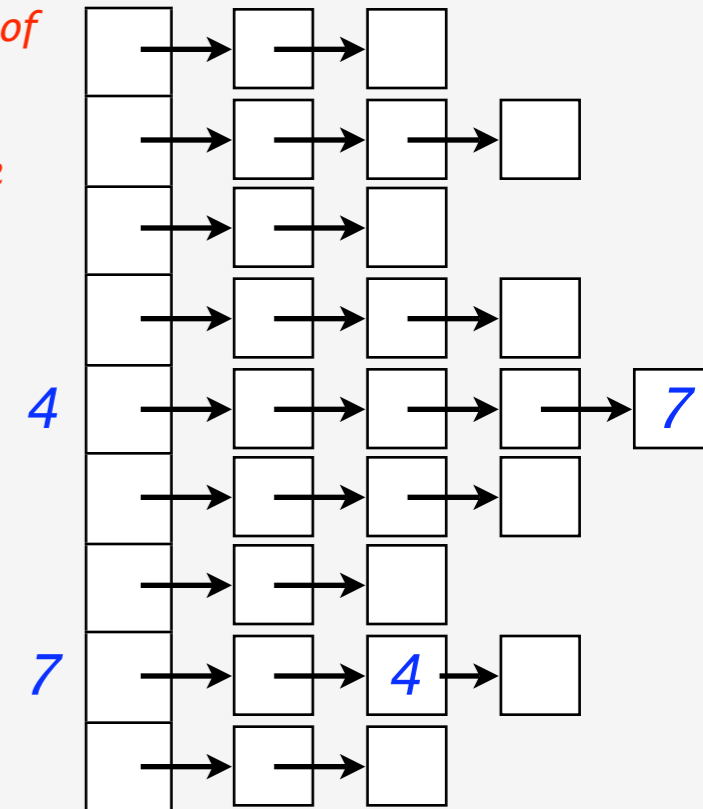
graph ADT constructor code

```
for (int k = 0; k < E; k++)  
  { int v = a[k].v, w = a[k].w;  
    adj[v] = new Node(w, adj[v]);  
    adj[w] = new Node(v, adj[w]);  
  }
```

graph representation

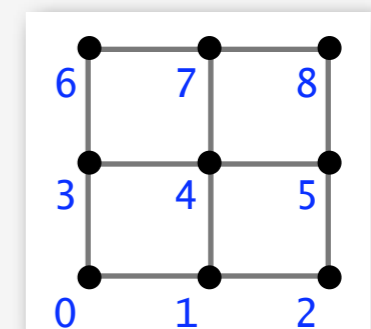
vertex-indexed array of
linked lists

two nodes per edge



DFS implementation (code to save path omitted)

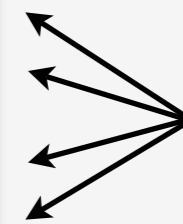
```
void findPathR(int s, int t)  
  { if (s == t) return;  
    visited(s) = true;  
    for(Node x = adj[s]; x != null; x = x.next)  
      if (!visited[x.v]) searchR(x.v, t);  
  }  
  
void findPath(int s, int t)  
  { visited = new boolean[V];  
    searchR(s, t);  
  }
```



Basic flaw in standard DFS scheme

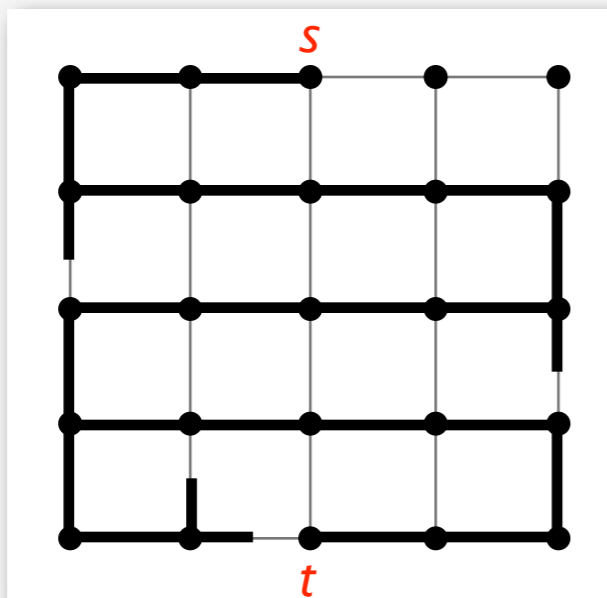
cost strongly depends on arbitrary decision in client code!

```
...  
for (int i = 0; i < V; i++)  
{  
    if ((i+1) % M != 0) a[e++] = new Edge(i, i+1);  
    if (i % M != 0) a[e++] = new Edge(i, i-1);  
    if (i < V-M) a[e++] = new Edge(i, i+M);  
    if (i >= M) a[e++] = new Edge(i, i-M);  
}  
...
```



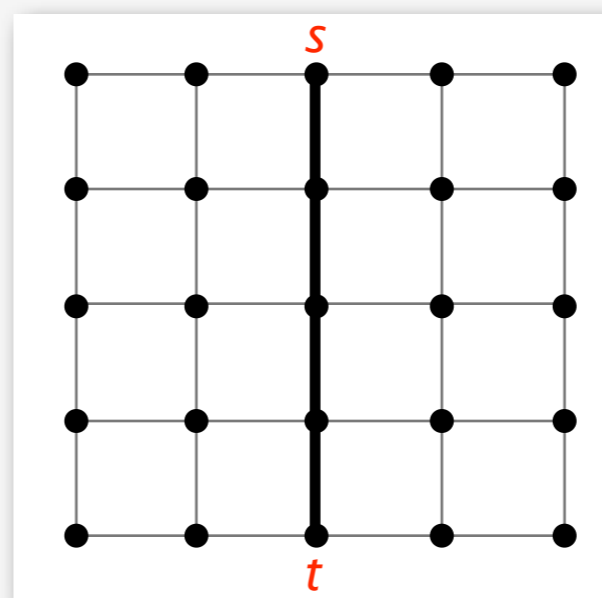
*order of these
statements
determines
order in lists*

west, east, north, south



$\sim E/2$

south, north, east, west



$\sim E^{1/2}$

*order in lists
has drastic effect
on running time*

*bad news
for ANY
graph model*

Addressing the basic flaw

Advise the client to randomize the edges?

- no, very poor software engineering
- leads to nonrandom edge lists (!)

Randomize each edge list before use?

- no, may not need the whole list

Solution: Use a **randomized iterator**

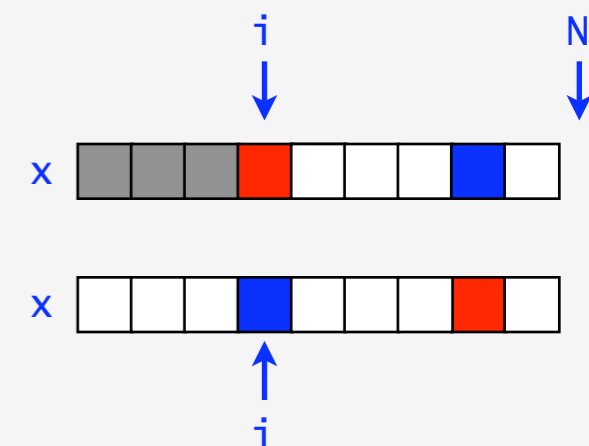
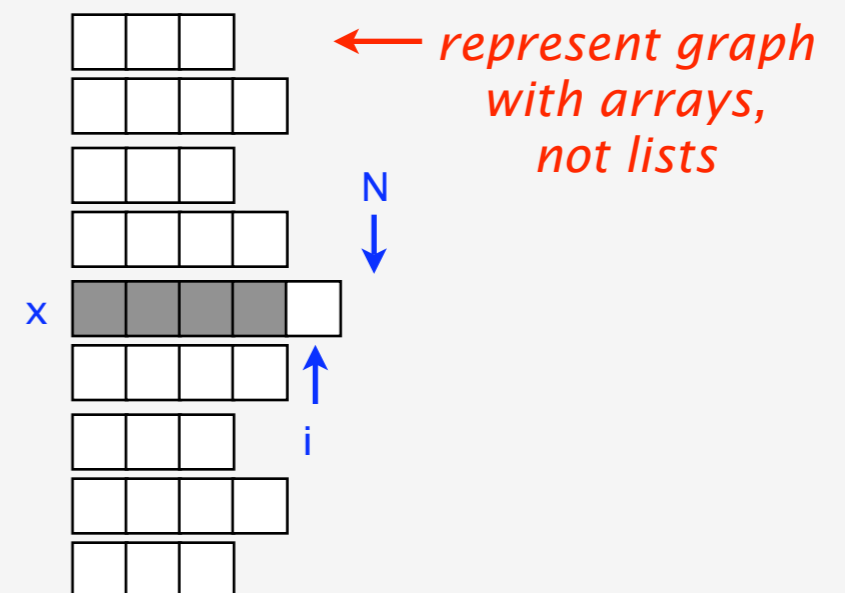
standard iterator

```
int N = adj[x].length;
for(int i = 0; i < N; i++)
  { process vertex adj[x][i]; }
```

randomized iterator

```
int N = adj[x].length;
for(int i = 0; i < N; i++)
  { exch(adj[x], i, i + (int) Math.random() * (N-i));
    process vertex adj[x][i];
  }
```

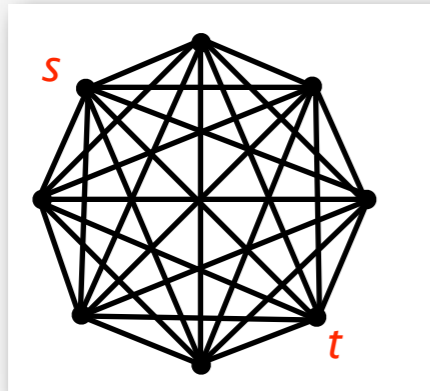
*exchange random vertex from
adj[x][i..N-1] with adj[x][i]*



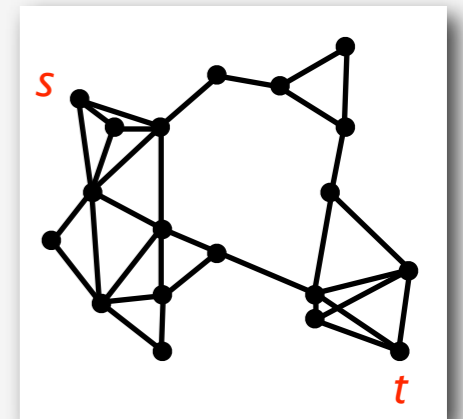
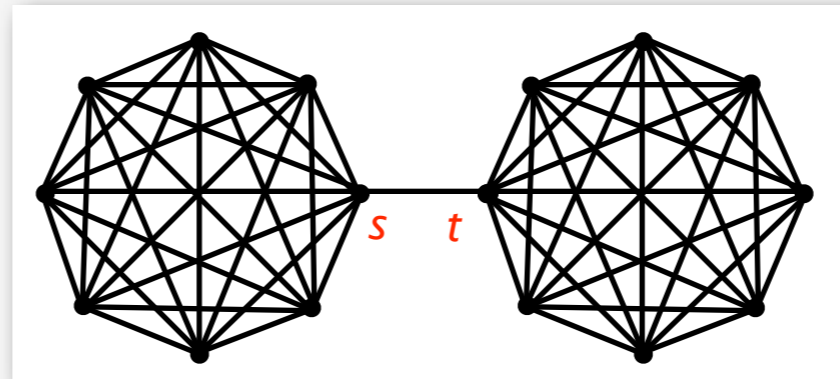
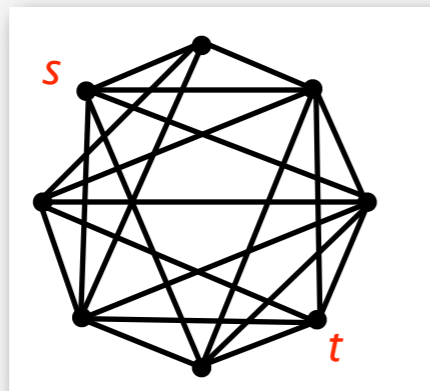
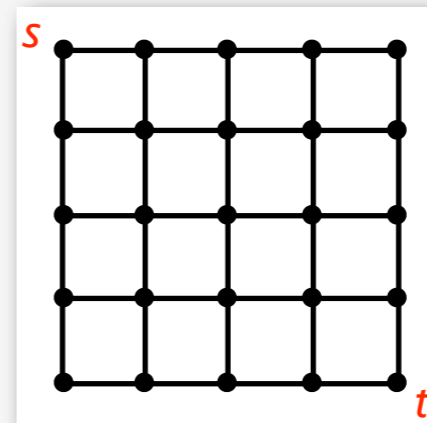
Use of randomized iterators

turns **every** graph algorithm into a randomized algorithm

Important practical effect: stabilizes algorithm performance



*cost depends on **problem**
not its representation*



Yields well-defined and fundamental analytic problems

- **Average-case analysis** of algorithm X for graph family $Y(N)$?
- **Distributions?**
- Full employment for algorithm analysts

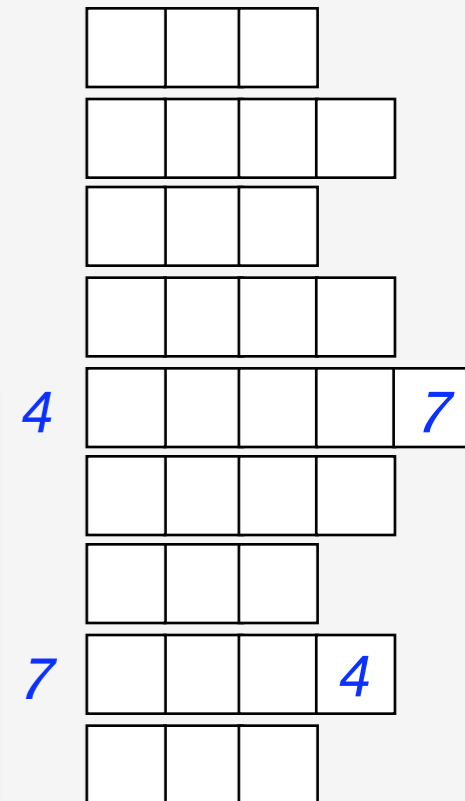
(Revised) standard DFS implementation

graph ADT constructor code

```
for (int k = 0; k < E; k++)  
  { int v = a[k].v, w = a[k].w;  
    adj[v][deg[v]++] = w;  
    adj[w][deg[w]++] = v;  
  }
```

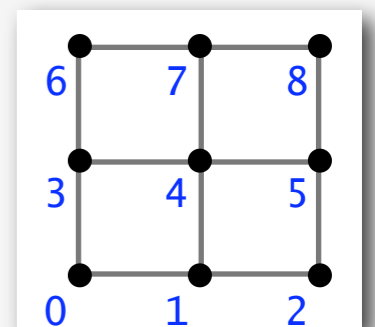
graph representation

vertex-indexed
array of variable-
length arrays



DFS implementation (code to save path omitted)

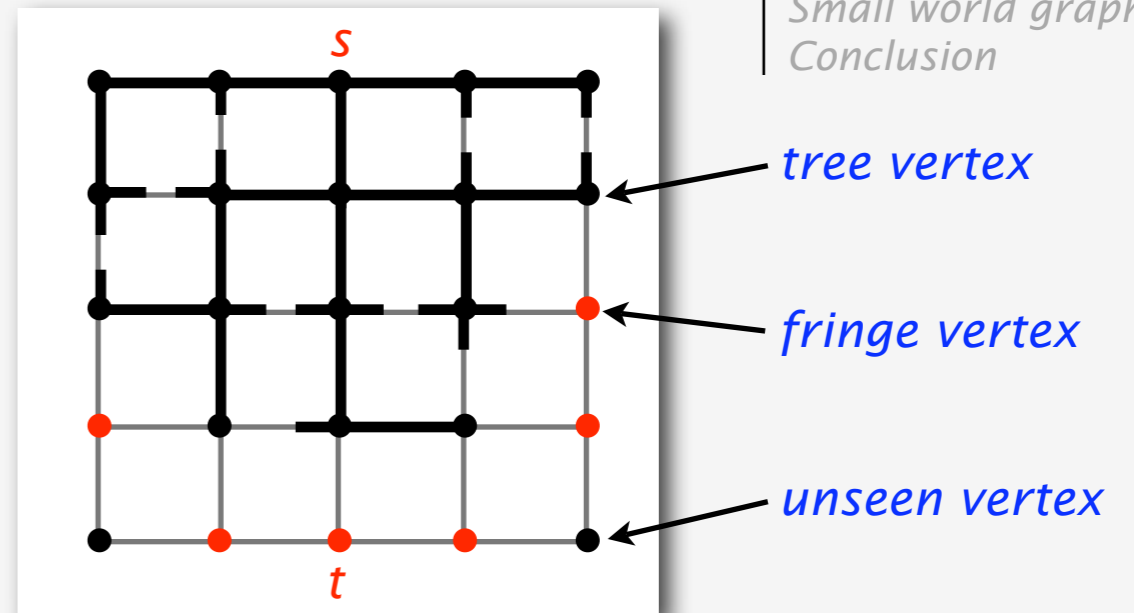
```
void findPathR(int s, int t)  
  { int N = adj[s].length;  
    if (s == t) return;  
    visited[s] = true;  
    for(int i = 0; i < N; i++)  
      { int v = exch(adj[s], i, i+(int) Math.random()*(N-i));  
        if (!visited[v]) searchR(v, t);  
      }  
  }  
  
void findPath(int s, int t)  
  { visited = new boolean[V];  
    findpathR(s, t);  
  }
```



BFS: standard implementation

Use a queue to hold **fringe** vertices

```
while Q is nonempty
    get x from Q
    done if x = t
    for each unmarked v adj to x
        put v on Q
        mark v
```



```
void findPath(int s, int t)
{ Queue Q = new Queue();
  Q.put(s); visited[s] = true;
  while (!Q.empty())
  { int x = Q.get(); int N = adj[x].length;
    if (x == t) return;
    for (int i = 0; i < N; i++)
    { int v = exch(adj[x], i, i + (int) Math.random()*(N-i));
      if (!visited[v])
        { Q.put(v); visited[v] = true; }
    }
  }
}
```

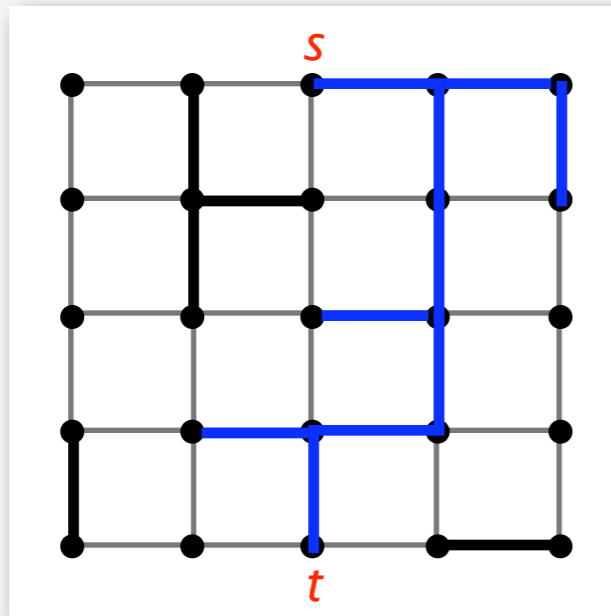
FIFO queue for BFS

randomized iterator

Generalized graph search: other queues yield A* and other graph-search algorithms

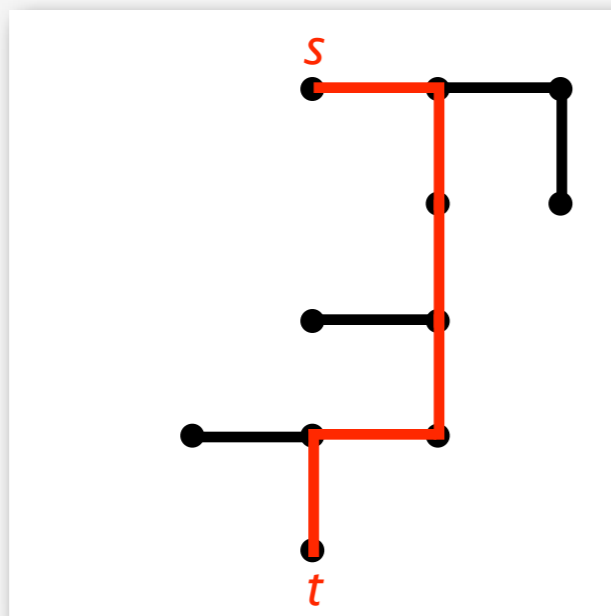
Union-Find implementation

1. Run union-find to find component containing s and t



initialize array of iterators
initialize UF array
while s and t not in same component
choose random iterator
choose random edge for union

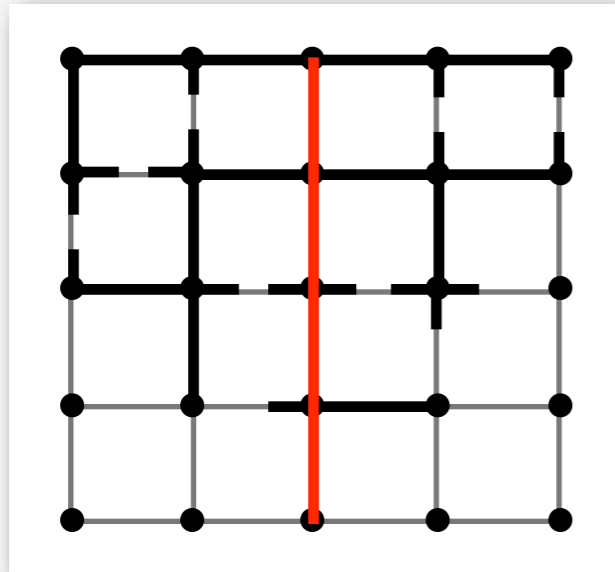
2. Build subgraph with edges from that component
3. Use DFS to find st -path in that subgraph



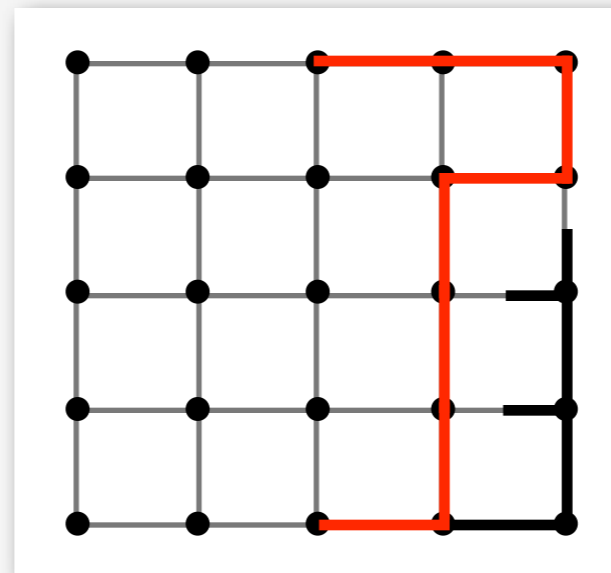
Animations

give intuition on performance

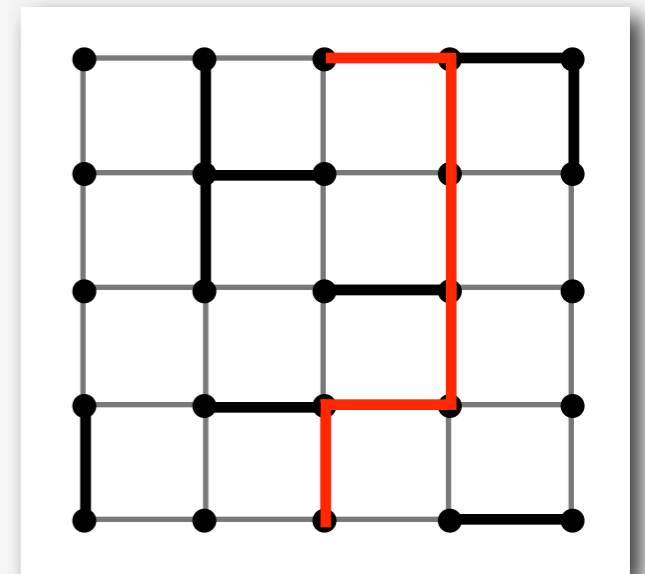
BFS



DFS



UF



and suggest hypotheses to verify with experimentation

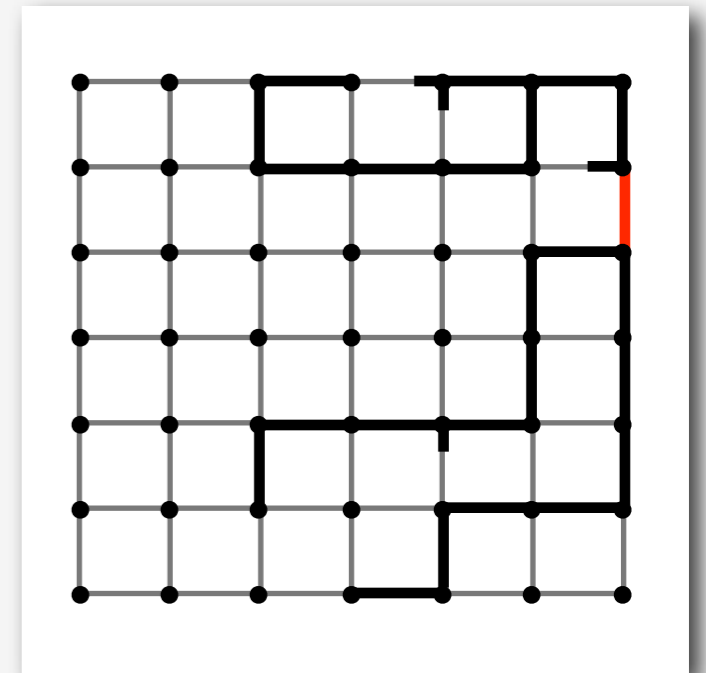
A faster algorithm

for finding an st -path in a graph

Use **two** depth-first searches

- one from the source s
- one from the destination t
- interleave the two

M	V	E	BFS	DFS	UF	two
7	49	168	.75	.32	1.05	.18
15	225	840	.75	.45	1.02	.13
31	961	3720	.75	.36	1.14	.15
63	3969	15624	.75	.32	1.05	.14
127	16129	64008	.75	.40	.99	.13
255	65025	259080	.75	.42	1.08	.12



Examines **13%** of the edges

3-8 times faster than standard implementations

Not $\log\log E$, but not bad!

Are other approaches faster?

Other search algorithms

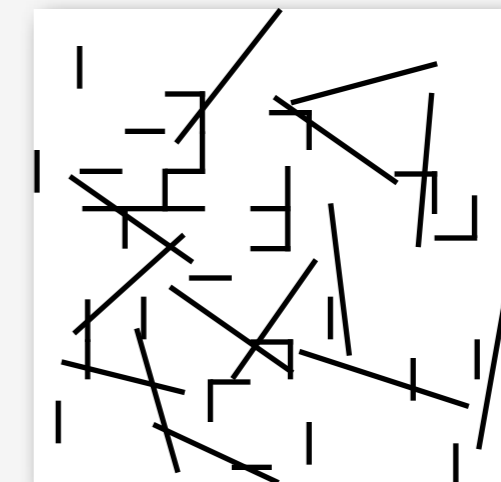
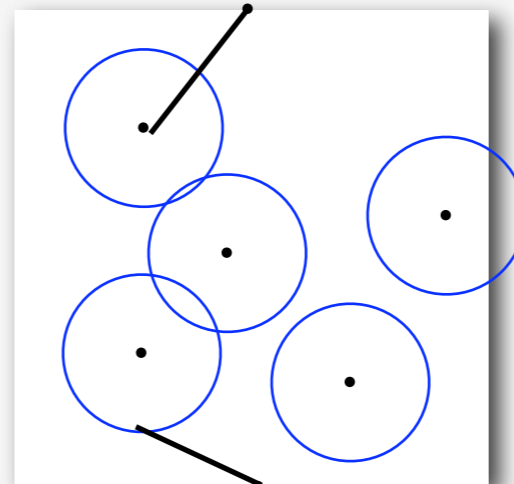
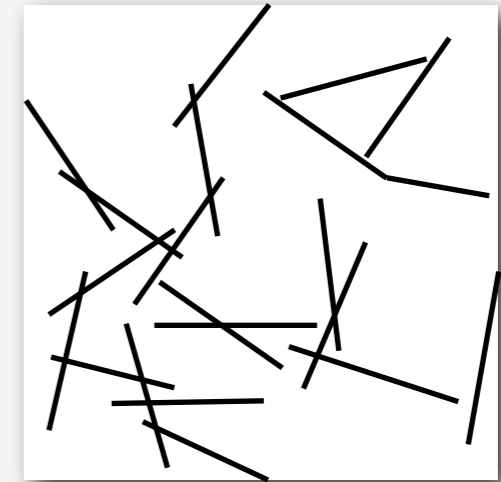
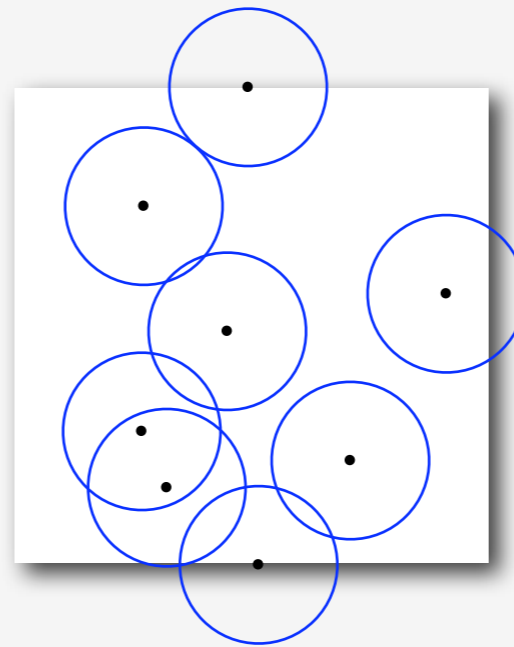
- randomized?
- farthest-first?

Multiple searches?

- interleaving strategy?
- merge strategy?
- how many?
- which algorithm?

Hybrid algorithms

- which combination?
- probabilistic restart?
- merge strategy?
- randomized choice?



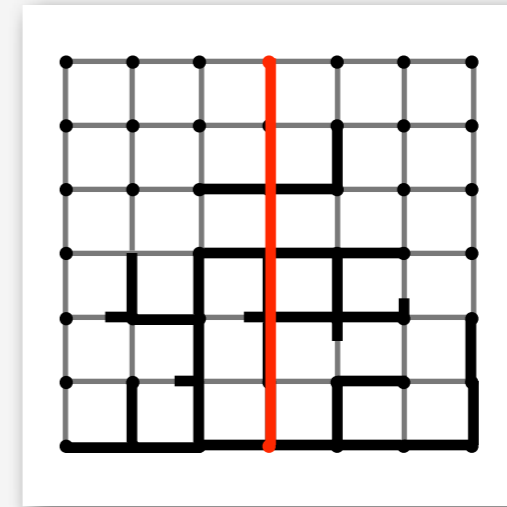
Better than constant-factor improvement possible? Proof?

Experiments with other approaches

Randomized search

- use random queue in BFS
- easy to implement

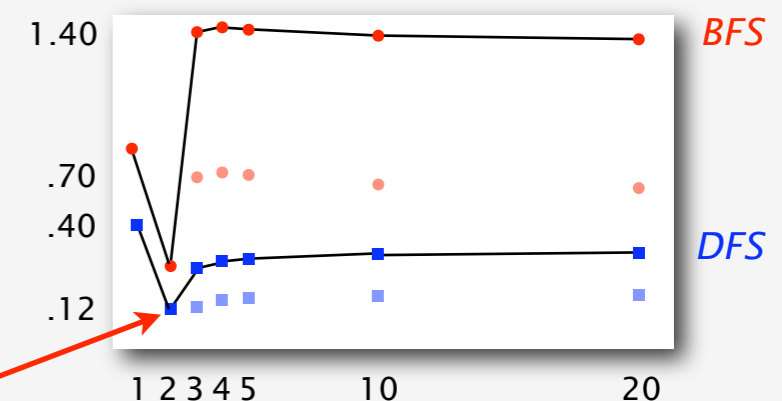
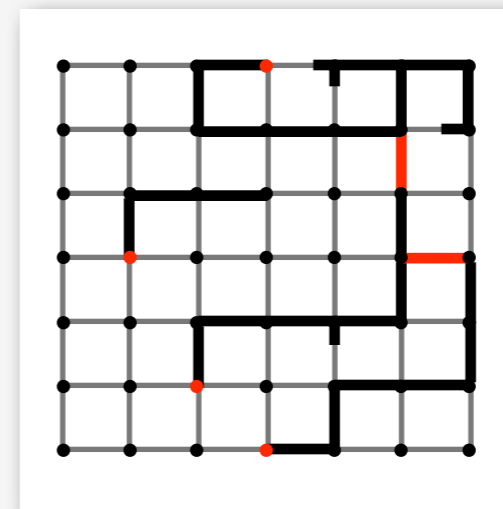
Result: not much different from BFS



Multiple searchers

- use N searchers
- one from the source
- one from the destination
- N-2 from random vertices
- Additional factor of 2 for $N > 2$

Result: not much help anyway



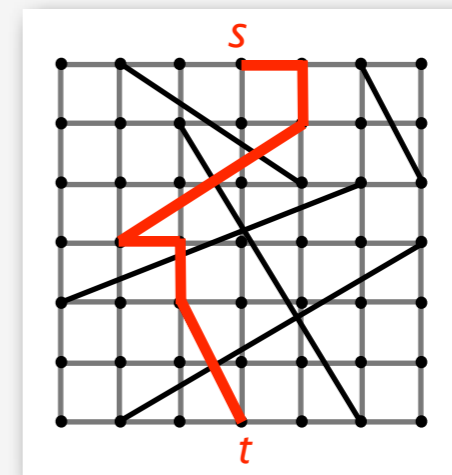
Best method found (by far): **DFS with 2 searchers**

Small-world graphs

are a widely studied graph model with many applications

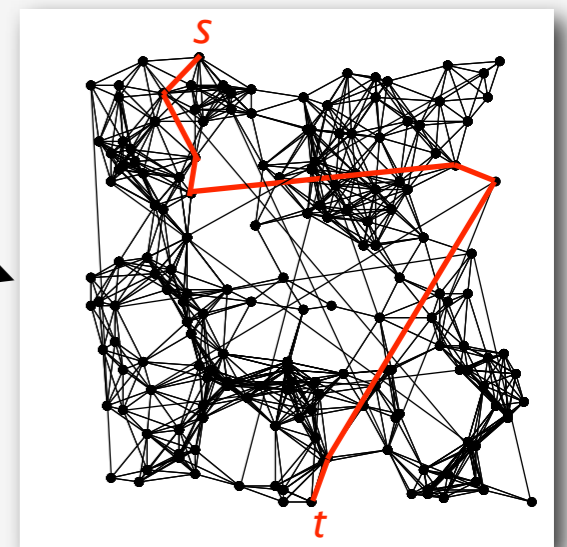
A **small-world** graph has

- large number of vertices
- low average vertex degree (sparse)
- low average path length
- local clustering



Examples:

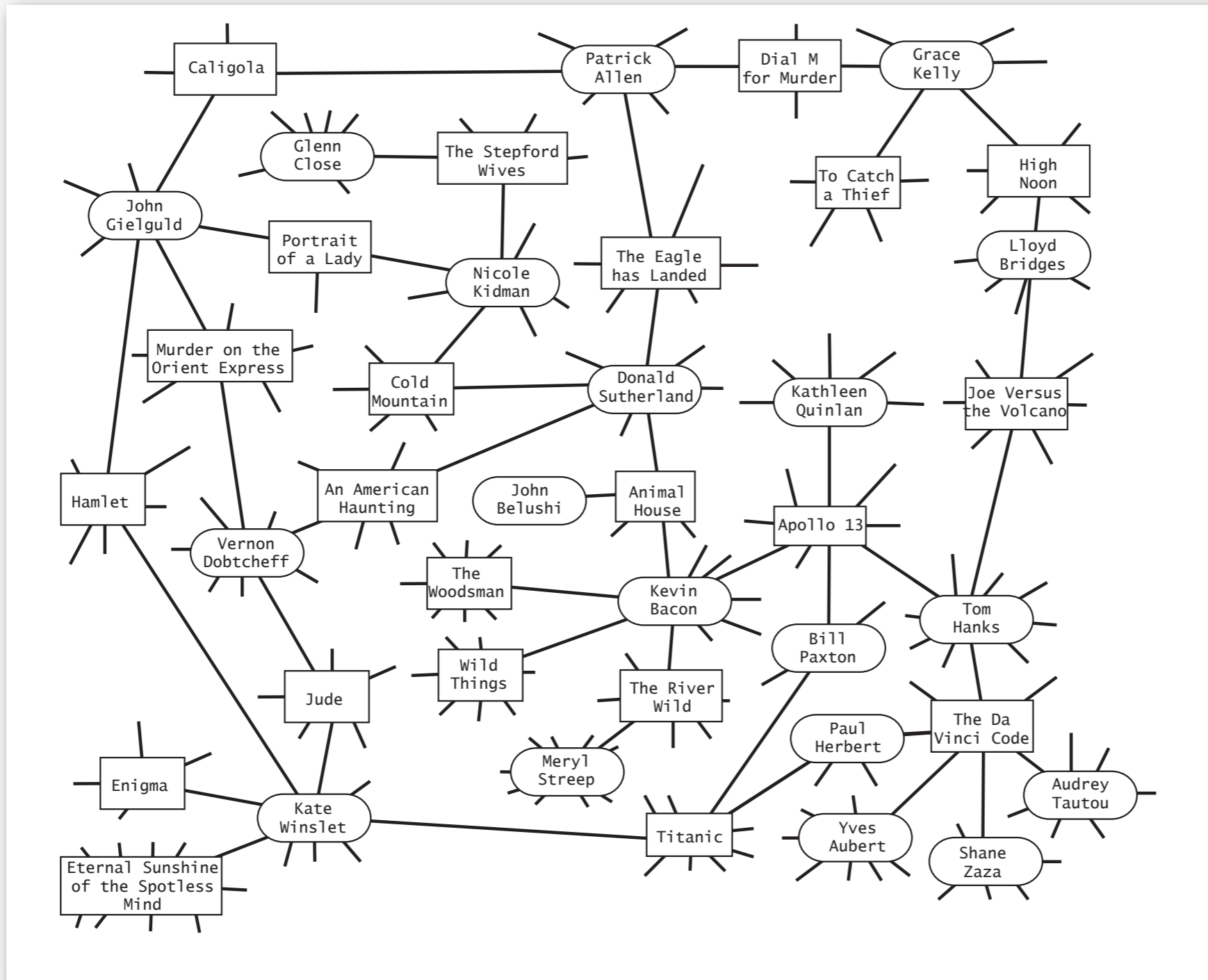
- Add random edges to grid graph
- Add random edges to **any** sparse graph with local clustering
- Many scientific models



Q. How do we find an st -path in a small-world graph?

Small-world graphs

model the **six degrees of separation** phenomenon



Example: Kevin Bacon number

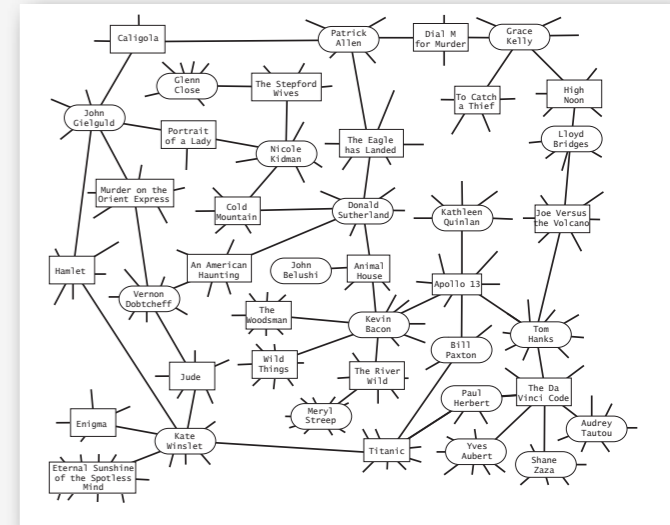
Applications of small-world graphs

- Introduction
- Motivating example
- Grid graphs
- Search methods
- Small-world graphs
- Conclusion

social networks
airlines
roads
neurobiology
evolution
social influence
protein interaction
percolation
internet
electric power grids
political trends

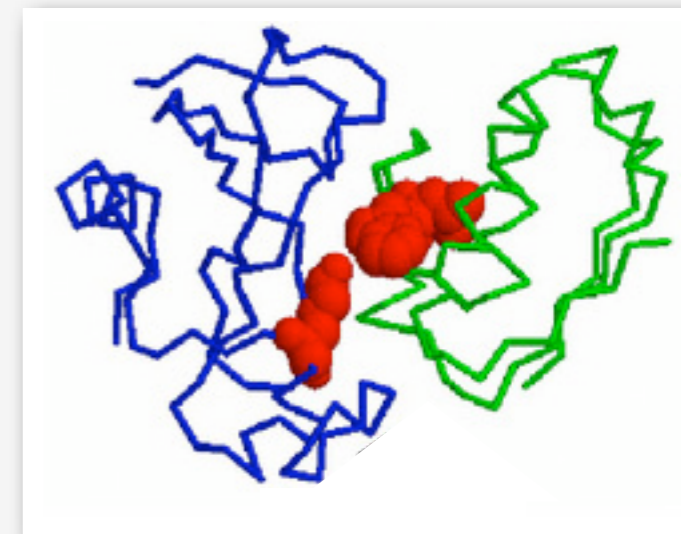
Example 1: Social networks

- infectious diseases
- extensive simulations
- some analytic results
- huge graphs



Example 2: Protein interaction

- small-world model
- natural process
- experimental validation



Finding a path in a small-world graph

is a heavily studied problem

Milgram experiment (1960)

Small-world graph models

- Random (many variants)
- Watts-Strogatz
- Kleinberg

*add V random shortcuts
to grid graphs and others*

A uses $\sim \log E$ steps to find a path*

How does 2-way DFS do in this model?

*no change at all in graph code
just a different graph model*

Experiment:

- add $M \sim E^{1/2}$ random edges to an M -by- M grid graph
- use **2-way DFS** to find path

Surprising result: Finds short paths in $\sim E^{1/2}$ steps!

Finding a path in a small-world graph

is much easier than finding a path in a grid graph

Introduction
Motivating example
Grid graphs
Search methods
Small-world graphs
Conclusion

Conjecture: Two-way DFS finds a short st -path in **sublinear** time in **any** small-world graph

Evidence in favor

1. Experiments on many graphs

2. Proof sketch for grid graphs with V shortcuts

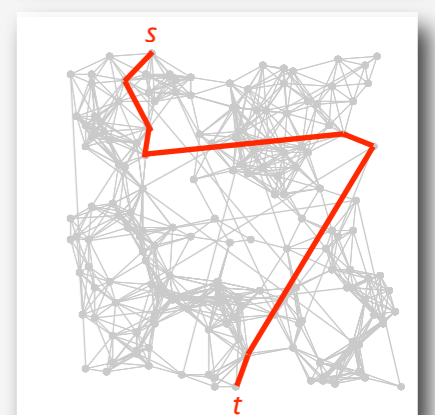
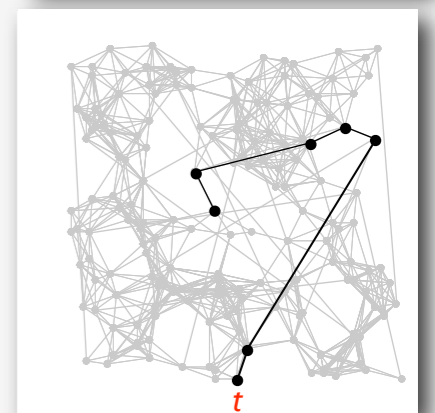
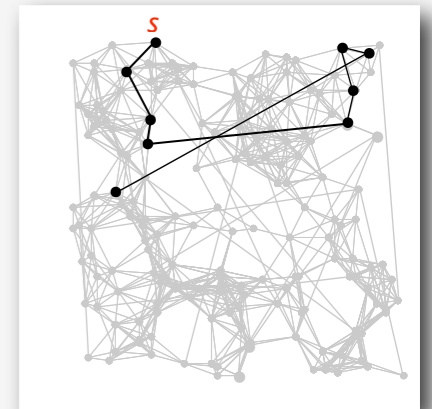
- step 1: $2 E^{1/2}$ steps $\sim 2 V^{1/2}$ random vertices
- step 2: like birthday paradox

two sets of $2V^{1/2}$ randomly chosen vertices are highly unlikely to be disjoint

Path length?

Multiple searchers revisited?

Next steps: refine model, more experiments, detailed proofs



Answers

- Randomization makes cost depend on graph, not representation.
- DFS is faster than BFS or UF for finding paths in grid graphs.
- Two DFSs are faster than 1 DFS — or N of them — in grid graphs.
- We can find short paths quickly in small-world graphs

Questions

- What are the BFS, UF, and DFS constants in grid graphs?
- Is there a sublinear algorithm for grid graphs?
- Which methods adapt to directed graphs?
- Can we precisely analyze and quantify costs for small-world graphs?
- What is the cost distribution for DFS for any interesting graph family?
- How effective are these methods for other graph families?
- Do these methods lead to faster maxflow algorithms?
- How effective are these methods in practice?
- ...

Lessons

Introduction
Motivating example
Grid graphs
Search methods
Small world graphs
Conclusion

We know much less than you might think about most of the algorithms that we use



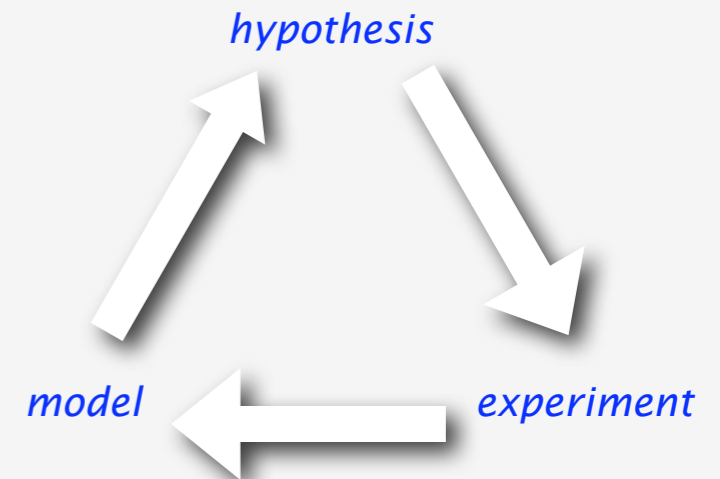
The scientific method is **necessary** in algorithm design and software development

The scientific method

is **necessary** in algorithm design and software development

Scientific method

- **create a model** describing natural world
- use model to **develop hypotheses**
- **run experiments** to validate hypotheses
- refine model and repeat



1950s



2000s



Algorithm designer who does not experiment gets lost in abstraction

Software developer who ignores cost risks catastrophic consequences