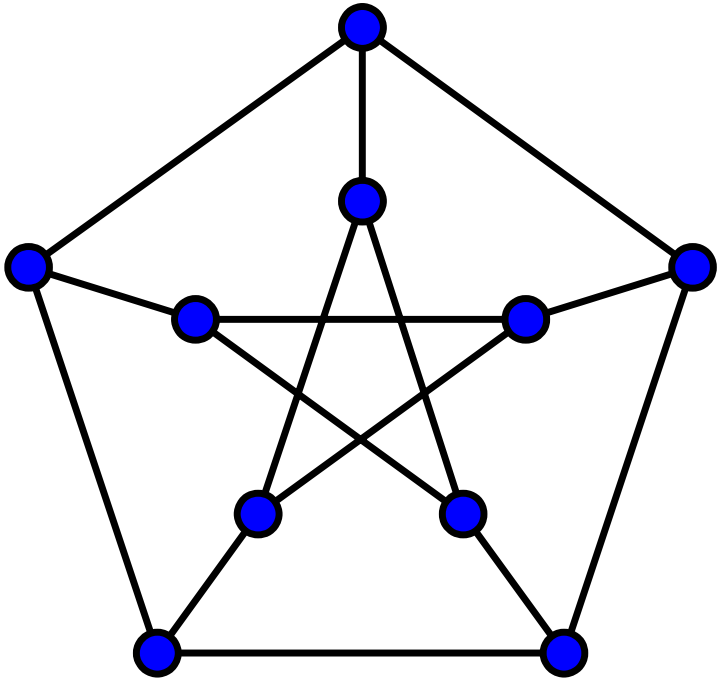
Graphs in Perl

Bob Mathews

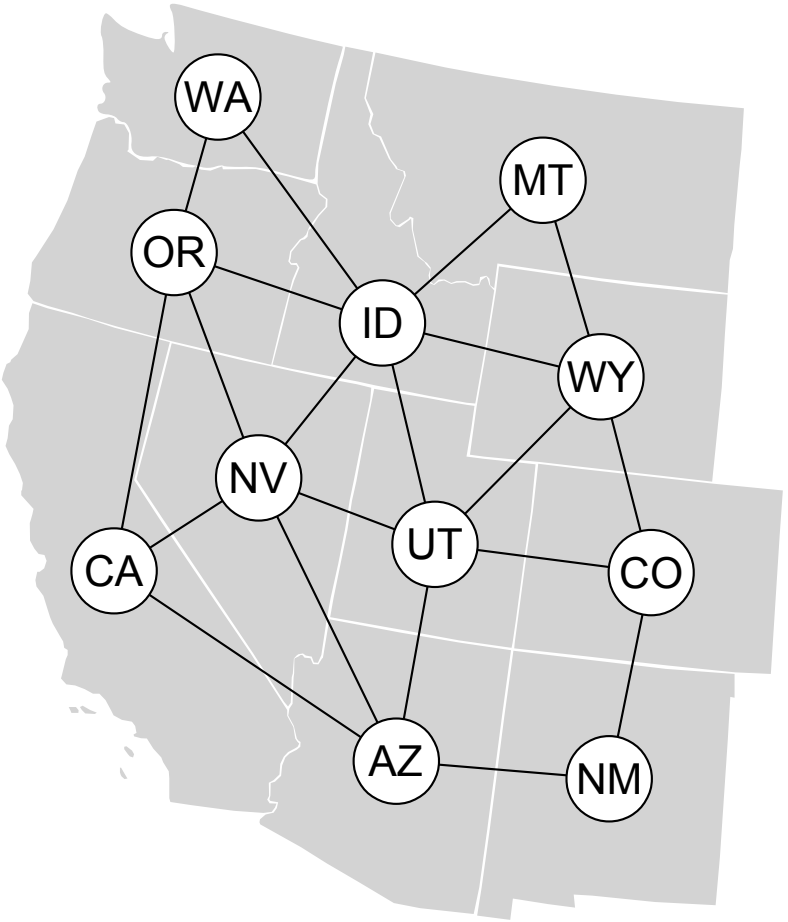


What kind of graphs?

Mathematically, a graph is a set of nodes (or vertices) connected by edges.

Graphs can represent many things:

- Cities connected by roads
- Borders between states
- Friends in a social network
- Module dependencies



Randomly place some nodes

I'm going to take an immediate left turn and talk about something not in the mathematical definition: node locations. Let's pick some at random.

```
my %graph;
for my $id (1 .. $num_nodes) {
   my $x = $node_rad + rand($im_wid - 2*$node_rad);
   my $y = $node_rad + rand($im_hgt - 2*$node_rad);
   $graph{$id} = { x => $x, y => $y };
}
```

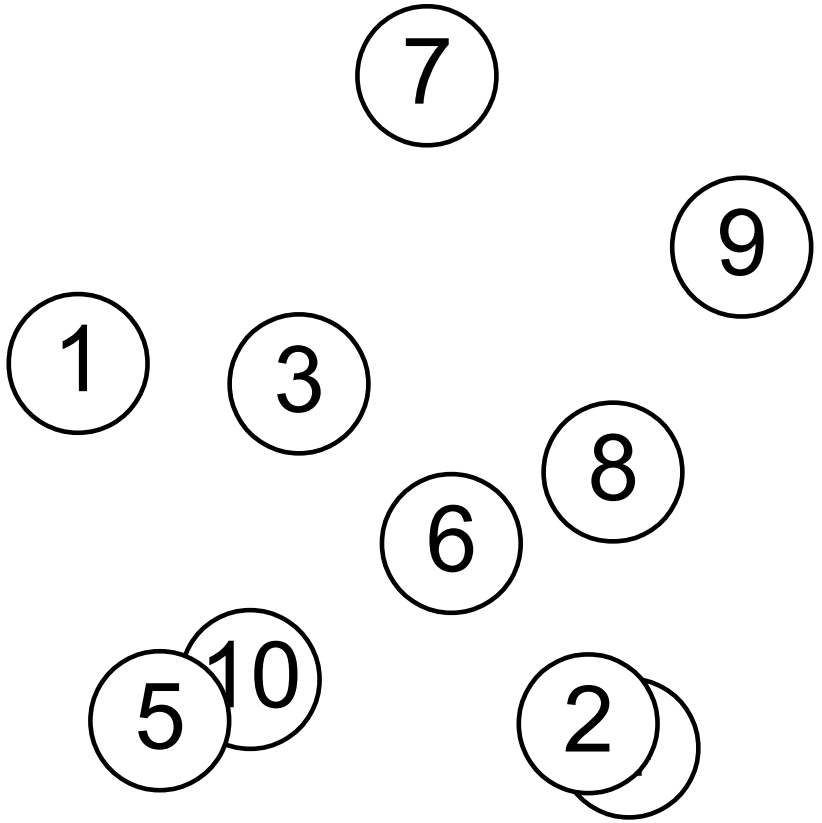
If you already have a dataset to display as a graph, maybe it includes coordinates. If it doesn't, picking “good” coordinates is a tricky problem.

Output graph as .SVG

```
print $OUT qq[<?xml version="1.0" encoding="UTF-8" ?>\n],
   qq[<svg xmlns="http://www.w3.org/2000/svg"],
   qq[ width="$im_wid" height="$im_hgt">\n];
while (my ($id, $node) = each %graph) {
   print $OUT qq[<g id="$id"],
      qq[ transform="translate($node->{x},$node->{y})">],
      qq[<circle cx="0" cy="0" r="$node_rad"],
      qq[ fill="#fff" stroke="#000" />],
      qq[<text x="0" y="$text_base"],
      qq[ font-family="$font_fam" font-size="$font_size"],
      qq[ text-anchor="middle">$id</text>],
      qq[</g>\n];
}
print $OUT qq[</svg>\n];
```

Putting the <circle> and <text> inside a <g> group keeps them together if we edit the image in a drawing program.

That's ugly



Overlapping nodes make the diagram ugly and confusing.

We can solve that problem...

Check for overlap

Every time we add a node, check to see if it's too close to the previous ones.

```
NODE: for my $id (1 .. $num_nodes) {
   my $x = $node_rad + rand($im_wid - 2*$node_rad);
   my $y = $node_rad + rand($im_hgt - 2*$node_rad);
   foreach my $node (values %graph) {
      # Warning: infinite loop?
      redo NODE if ($x - $node->{x})**2
         + ($y - $node->{y})**2 < $min_dist**2;
   }
   $graph{$id} = { x => $x, y => $y };
}
```

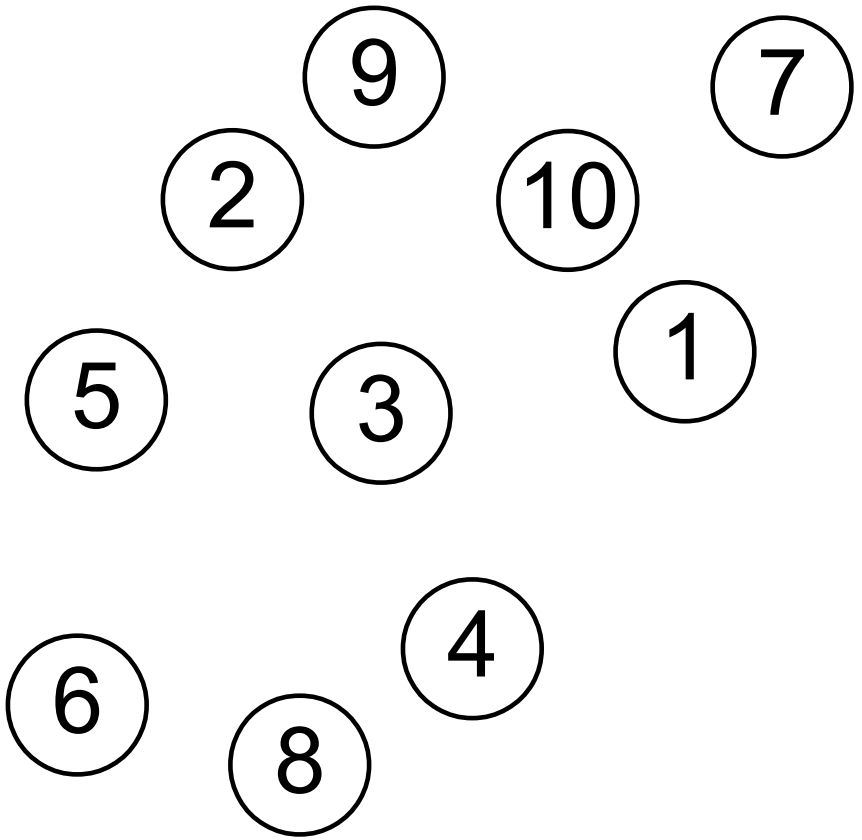
Note that you can get stuck if you try to jam too many nodes on the page.

That's slow

If you have like a thousand nodes, that will take a long time. Speed it up with a spatial index!

```
my $index = Algorithm::SpatialIndex->new(
   strategy => 'QuadTree', storage => 'Memory',
   limit_x_low => 0, limit_x_up => $im_wid,
   limit_y_low => 0, limit_y_up => $im_hgt);
...
   foreach my $node ($index->get_items_in_rect(
         $x - $min_dist, $y - $min_dist,
         $x + $min_dist, $y + $min_dist))
   {
      redo NODE if ($x - $node->[1])**2
         + ($y - $node->[2])**2 < $min_dist**2;
   }
   $index->insert($id, $x, $y);
```

That's better



Now that we have a reasonable set of nodes, we can move on to the next step:

Adding edges

Adding edges

We can just randomly connect nodes...

```
my @ids = keys %graph;
for (1 .. $num_edges) {
    my $id1 = $ids[rand @ids];
    my $id2 = $ids[rand @ids];
    redo if $id1 eq $id2;
    my $node1 = $graph{$id1} or die;
    my $node2 = $graph{$id2} or die;
    redo if $node1->{edges}{$id2};
    my $dist = sqrt(($node1->{x} - $node2->{x})**2
        + ($node1->{y} - $node2->{y})**2);
    $node1->{edges}{$id2} = $dist;
    $node2->{edges}{$id1} = $dist;
}
```

Edge weights

Notice that I'm storing the distance between nodes in the edges hash. This is an example of an “edge weight.” Other weights might be:

- Length of a road (longer if the road is windy)
- Travel time
- Tolls
- Traffic capacity

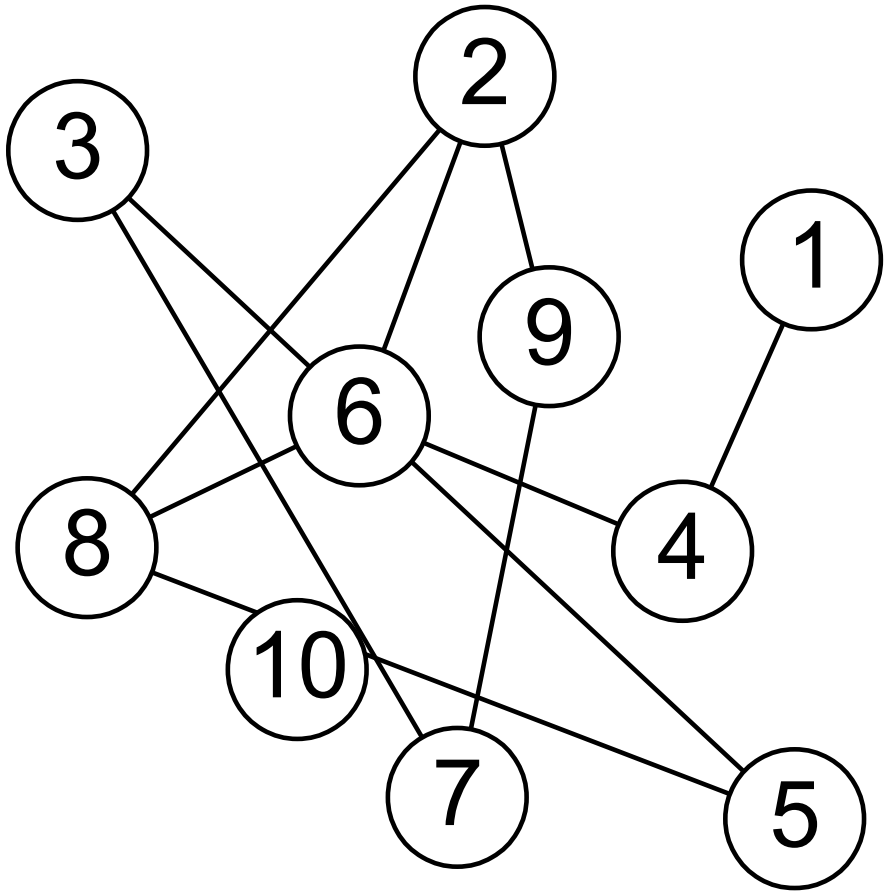
The edges hash only allows one edge between a pair of nodes. In some situations, “parallel edges” are important. This makes it hard to define a “best” graph data structure.

Output edges to .SVG

```
my %visited;
while (my ($id1, $node1) = each %graph) {
   $visited{$id1} = 1;
   while (my ($id2, $dist) = each %{$node1->{edges}}) {
      next if $visited{$id2}; # edge already drawn
      my $node2 = $graph{$id2} or die;
      print $OUT qq[<line stroke="#000"],
         qq[ x1="$node1->{x}" y1="$node1->{y}"],
         qq[ x2="$node2->{x}" y2="$node2->{y}" />\n];
   }
}
```

Note: we're drawing the edges from center to center. That will look bad unless you draw the nodes over the top of them (put the nodes later in the .svg file).

But that's still ugly



Just randomly connecting things produces a tangle of edges that cross over each other.

We want a “planar embedding” of our graph. (There may not be one.) We can try and build one by hand by moving things around...

Draggable edges

We can use Inkscape extensions to keep the edges attached to the nodes when you drag them around. (They will still display correctly in other SVG viewers.)

```
print $OUT qq[<?xml version="1.0" encoding="UTF-8" ?>\n],
   qq[<svg xmlns="http://www.w3.org/2000/svg"],
   qq[ xmlns:inkscape="http://www.inkscape.org/namespaces/inkscape"],
   qq[ width="$im_wid" height="$im_hgt">\n];
...
      print $OUT qq[<path stroke="#000"],
         qq[ d="M $x1,$y1 $x2,$y2"],
         qq[ inkscape:connector-type="polyline"],
         qq[ inkscape:connection-start="#$id1"],
         qq[ inkscape:connection-end="#$id2" />\n];
```

Edges now need to go after the nodes in the .svg file, so we'll have to calculate endpoints that don't overlap.

Calculate endpoints

Calculating the endpoints is a fairly simple exercise.

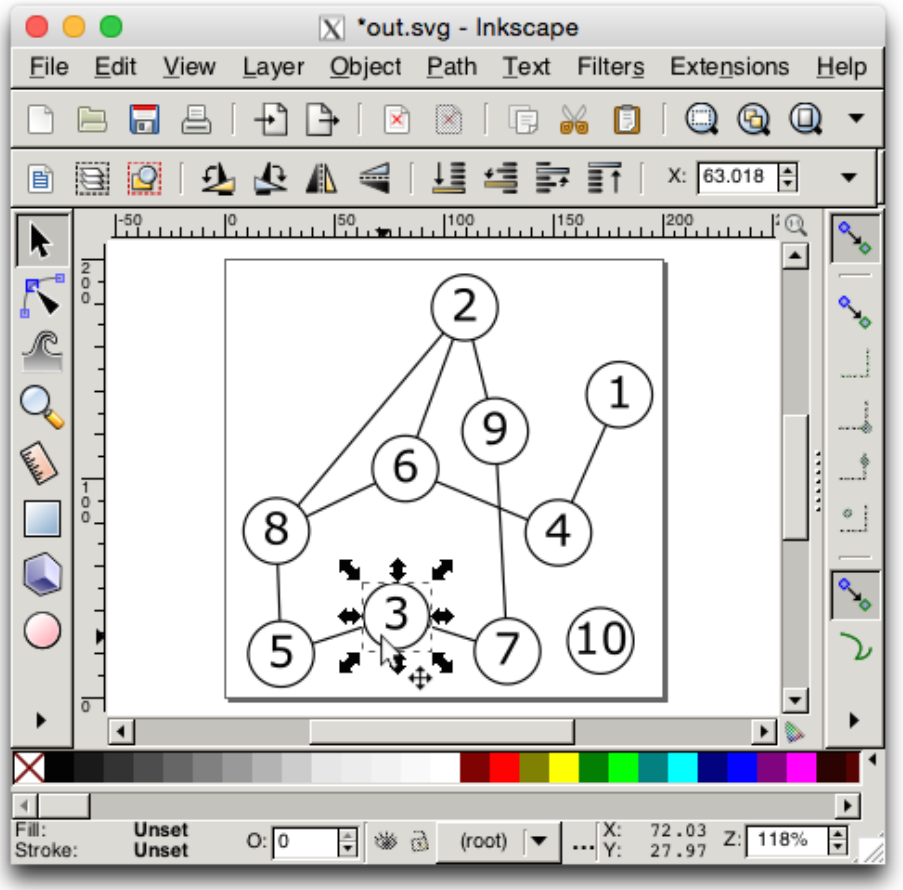
Close your eyes if you're allergic to math.

```
my $x1 = $node1->{x}; my $y1 = $node1->{y};
my $x2 = $node2->{x}; my $y2 = $node2->{y};
my $dx = $x2 - $x1; my $dy = $y2 - $y1;
my $d = $node_rad / sqrt($dx * $dx + $dy * $dy);
$dx *= $d; $dy *= $d;
$x1 += $dx; $y1 += $dy;
$x2 -= $dx; $y2 -= $dy;
```

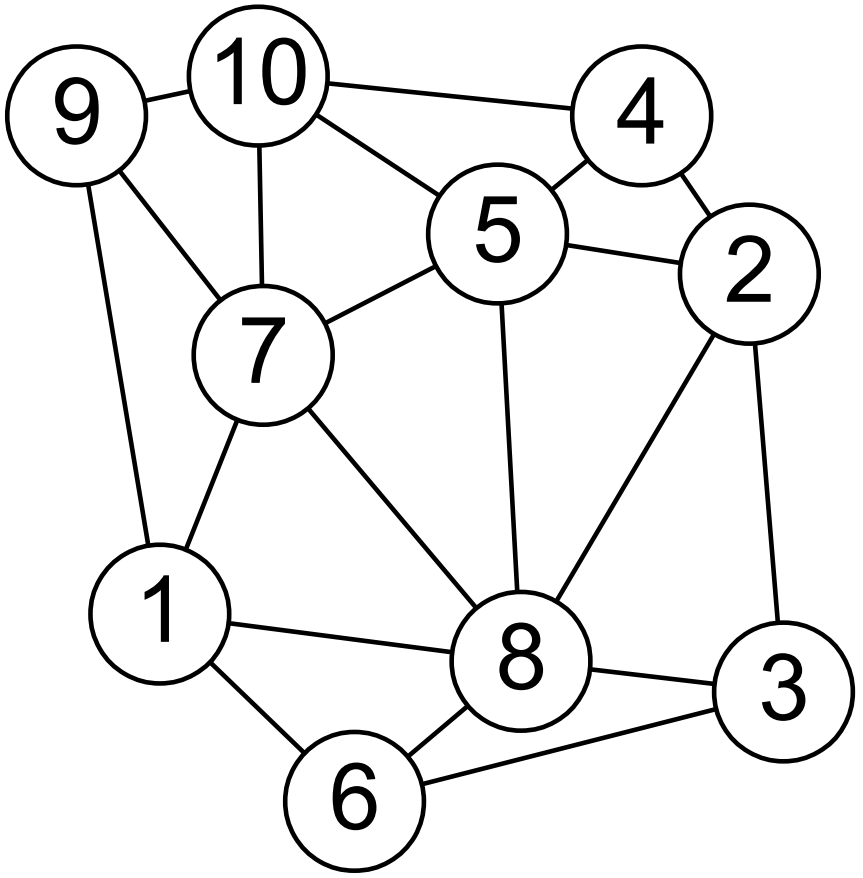
That's fun

Now you can open up the graph in Inkscape and try to drag nodes around until the edges don't cross.

This rapidly gets frustrating for large graphs. It's easier if we can just pick “good” edges in the first place.



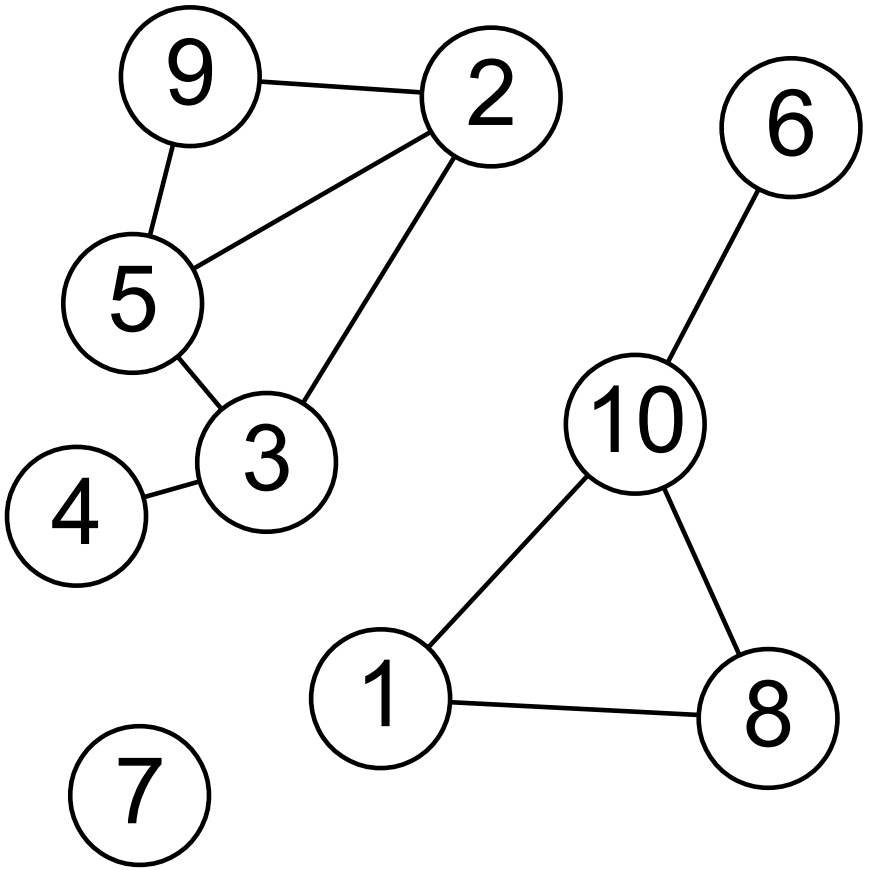
Generate non-crossing edges



Math::Geometry::Delaunay calculates a nice-looking triangulation with a maximal set of non-crossing edges.

A fully triangulated graph is quite dense with edges, so you may want to choose only some of them, at random.

That's not it either



We picked some random edges, but now the graph isn't connected. There are 3 “components.”

How can we tell if the graph is connected (has only one component)?

TIMTOWTDI

There is more than one way to do it.

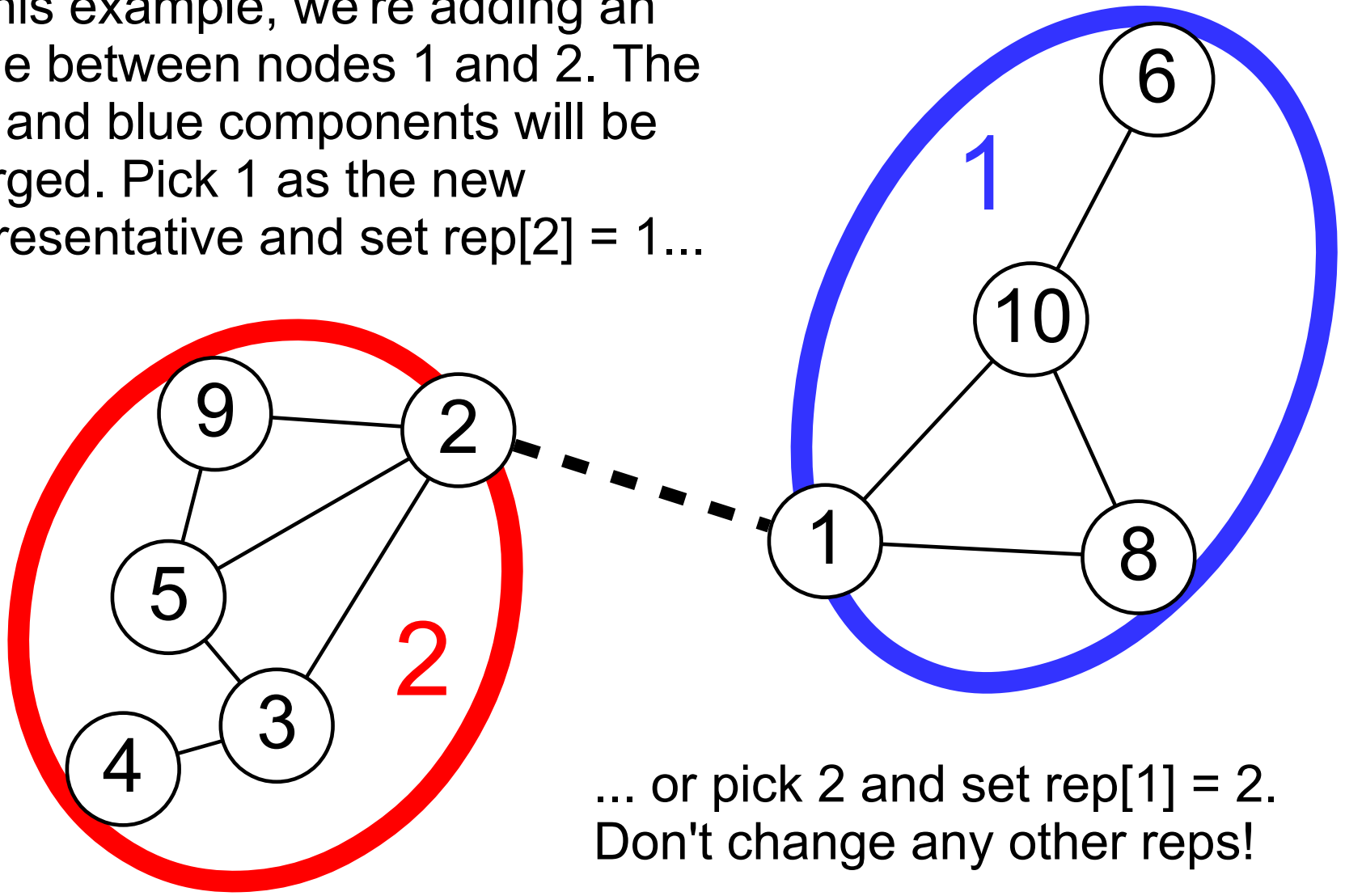
I'm using a “disjoint set” method:

- Start with each node in a separate component
- Add one edge at a time:
- If the ends are in the same component, do nothing
- If they aren't, merge the two components. Choose one node as the “representative” of the new component.

We need to be able to find the representative for any node, BUT we don't want to spend a lot of time re-labelling nodes with a new rep when we merge.

Merging components

In this example, we're adding an edge between nodes 1 and 2. The red and blue components will be merged. Pick 1 as the new representative and set rep[2] = 1...



... or pick 2 and set rep[1] = 2.
Don't change any other reps!

Counting components

Here's the Perl code for the main loop:

```
my $num_components = keys %graph;
my %reps;
while (my ($id1, $node1) = each %graph) {
   my $rep1 = find_rep($id1, \%reps);
   while (my ($id2, $dist) = each %{$node1->{edges}}) {
      my $rep2 = find_rep($id2, \%reps);
      if ($rep1 ne $rep2) {
         $reps{$rep2} = $rep1; # merge components
         --$num_components;
      }
   }
}
```

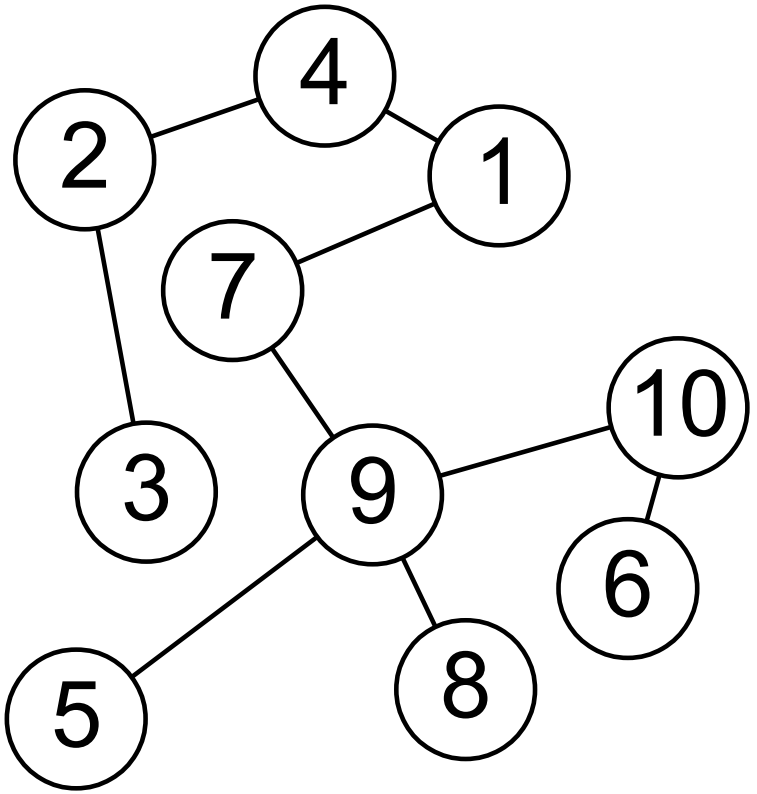
Arbitrarily choose $rep1 as the new rep. If the graph is really huge, “union by rank” is better (choose the rep with the shortest chain of underlings).

Find representative of a node

Avoid doing redundant work by flattening chains of old representatives (“path compression”).

```
sub find_rep {
   my ($id, $reps) = @_;
   my $rep = $id;
   # find representative
   while (defined(my $temp = $reps->{$rep})) {
      $rep = $temp;
   }
   # path compression
   while (defined(my $temp = $reps->{$id})) {
      $reps->{$id} = $rep;
      $id = $temp;
   }
   return $rep;
}
```

Spanning tree

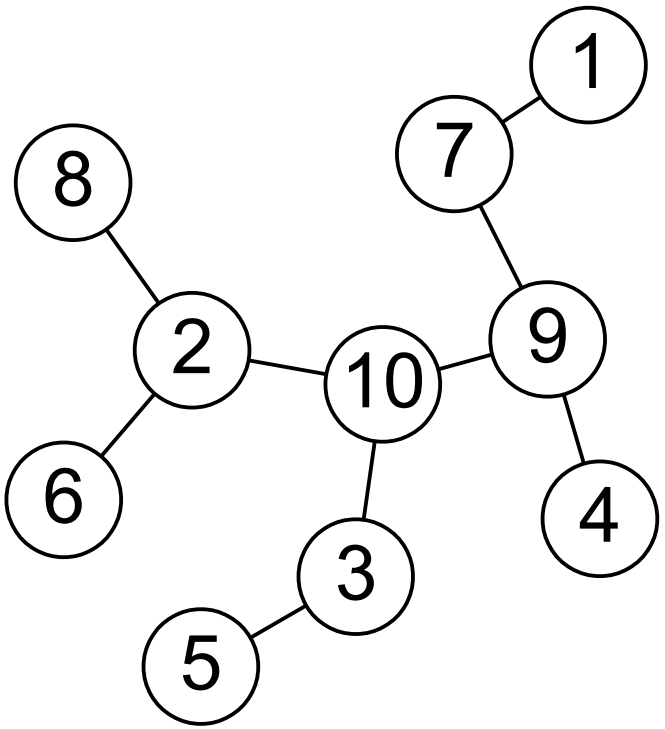


If you only add edges that cause a merge, you get a graph with no cycles.

There's no way to loop back where you started from without recrossing an edge.

This is called a “tree” or “acyclic graph.”

Hey, hey!



If you sort all the edges by weight first, you get a “minimum spanning tree.”

This is known as Kruskal's Algorithm.

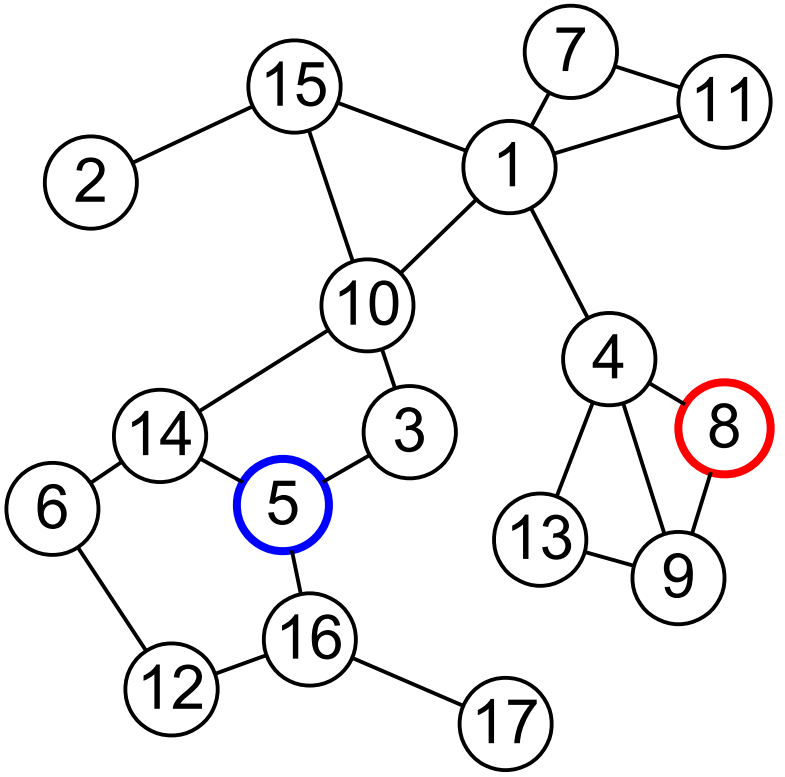
An MST is a neat and tidy little graph. Every node is connected the “best” way. To make things a bit more interesting, add a few random edges (from Delaunay).

Where were we going with this?

Now that we can build graphs, one obvious thing to do is find paths from one node to another.

How would we get from node 5 to 8?

Easy to solve by eye in a graph this small.



Dijkstra's shortest path

The classic algorithm from the guy with the weird name (“Dijk” sounds like “dike”):

Put the start node in the active list with distance 0

- Get the active node with the shortest distance
- If that's the end node, we're done
- For each edge attached to this node,
  - New distance is node distance plus edge distance
  - New node is the one on the other end of the edge
  - Add new node to active with new distance, unless new node's distance is already shorter

Dijkstra's shortest path code

```
my $pq = Array::Heap::ModifiablePriorityQueue->new();
$pq->add($start_id, 0);
$graph{$start_id}{dist} = 0;
while (my $id = $pq->get()) {
   last if $id eq $end_id;
   my $node = $graph{$id} or die;
   while (my ($id2, $len) = each %{$node->{edges}}) {
      my $node2 = $graph{$id2} or die;
      my $dist = $node->{dist} + $len;
      if (!exists($node2->{dist})
            || $dist < $node2->{dist})
      {
         $node2->{dist} = $dist;
         $node2->{path} = $id;
         $pq->add($id2, $dist);
      }
   }
}
```

Extracting the path

The shortest path is left in the graph structure. That's memory-efficient while the algorithm is running, but inconvenient afterwards. To extract the shortest path into an array:

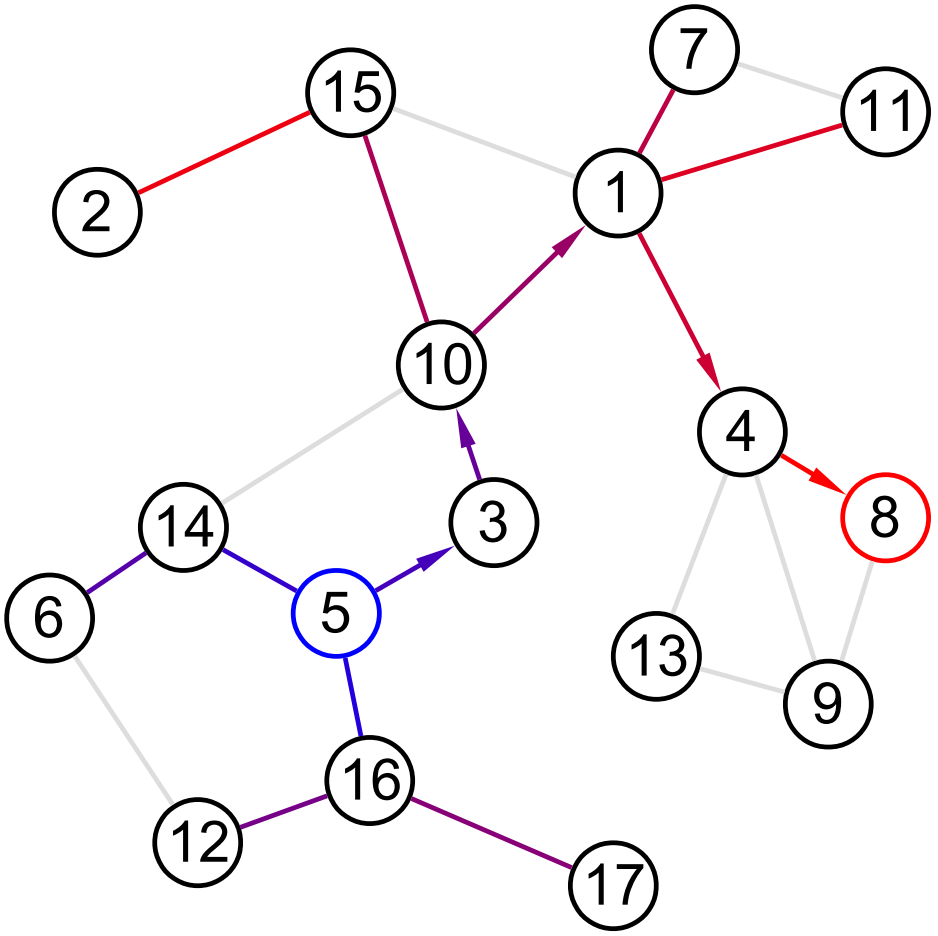
```
my $id = $end_id;
my @path;
while (defined $id) {
    unshift @path, $id;
    $id = $graph{$id}{path};
}
```

Dijkstra's results

The chosen path is
5 - 3 - 10 - 1 - 4 - 8

Explored almost the entire graph (colored edges) before finding the solution.

Dijkstra's algorithm knows nothing about node locations, but often that's useful information.



A* algorithm

It turns out that there's an easy way to add geometric information to Dijkstra's algorithm. We only have to change ONE LINE.

```
$pq->add($id2,  $dist + dist($node2, $graph{$end_id}));
```

When adding $node2 to the active queue, add an estimate of the remaining distance to the end point. (In this case, use the straight-line distance, whether or not an edge exists between those two nodes.)

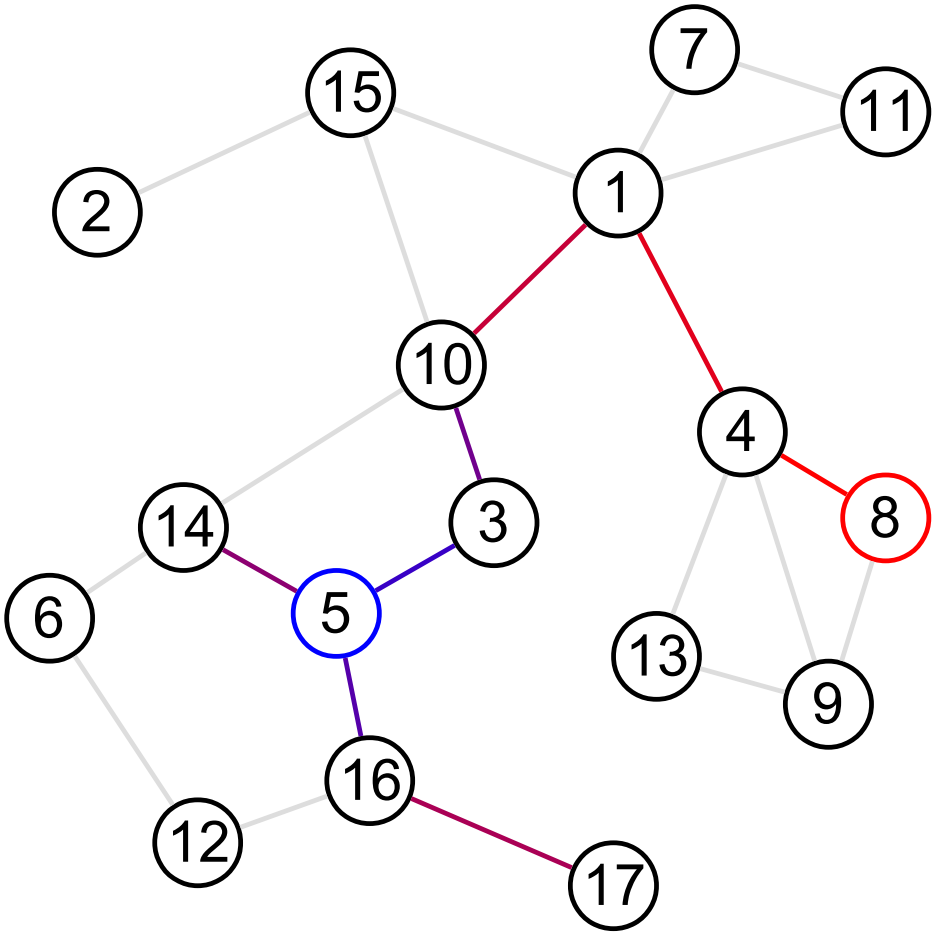
The estimate must be optimistic, meaning that it can't be greater than the actual distance to the end.

A* results

Same solution, but it avoided most nodes that are obviously in the wrong direction.

(Nodes 16 and 17 looked good, but turned out to be a dead end.)

Very important speedup in many real-world path-finding problems.



Checkpoint

These path algorithms are the most important topic I'm going to talk about today.

If you followed that, great!

If you're feeling a bit lost, yell at me!

Things are about to get a bit weird.

Thinking ahead

The A* algorithm has no overall knowledge of the graph structure to steer it.

If you could precompute some information about the graph (before knowing the start and end nodes), what would be useful?

You can compute paths between all pairs of nodes with the Floyd-Warshall algorithm, but that might take a lot of memory to store.

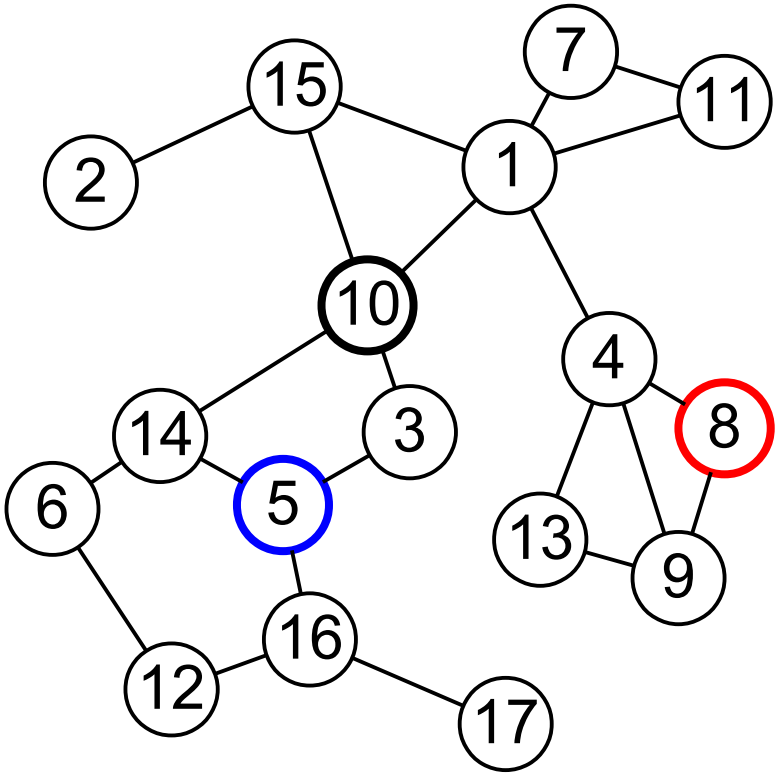
There are other features of the graph that might be useful to know, such as...

Cut vertices

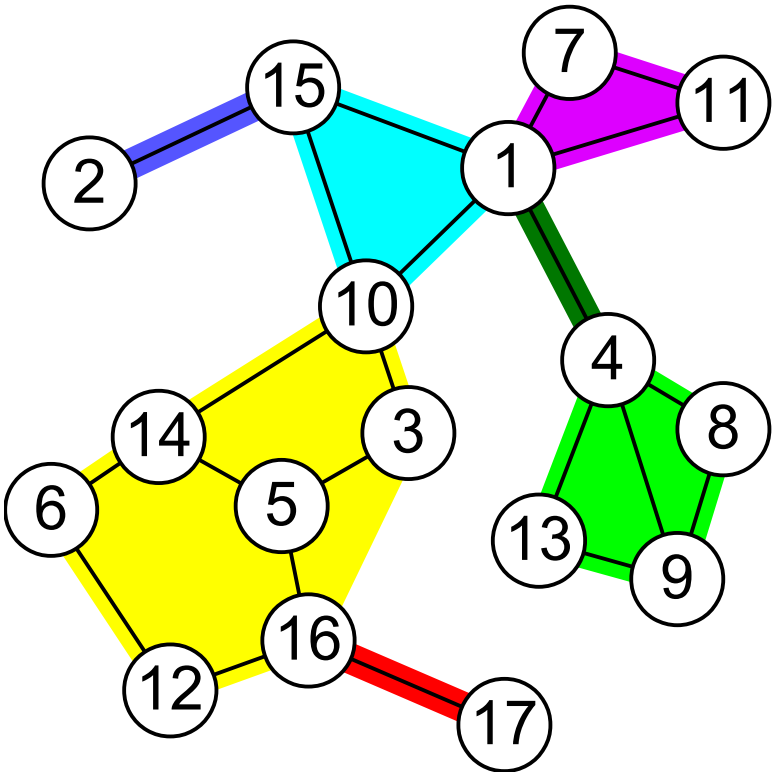
Can't get from 5 to 8 without visiting node 10.

Removing 10 would break the graph in two pieces.

Such nodes are called "cut vertices" or "articulation points."



Blocks



The cut vertices divide a graph into “blocks” or “biconnected components.”

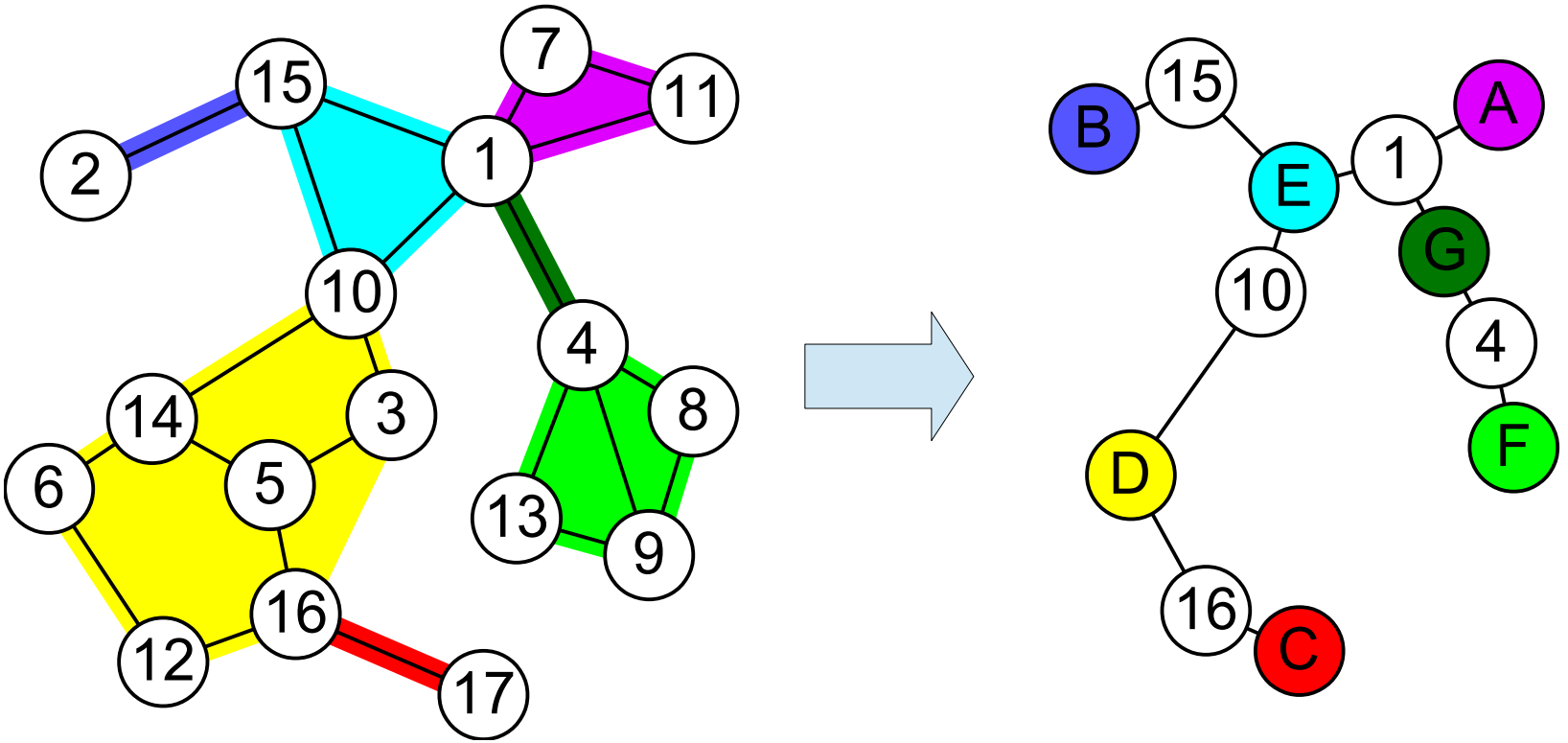
There are at least two separate paths between any two nodes in a block
(with at least 3 nodes).

Each cut vertex is part of two or more blocks.

The block outlines were found with Math::ConvexHull

Block-cut vertex tree

Build a new graph by replacing each block with a node.



The result is a tree. There's only one path between any two nodes. Routing is easy!

Block finding

The algorithm for finding cut vertices and blocks isn't too complicated. Basically,

- Pick a starting node and explore the graph, but don't retrace your steps.
- If you get back to a previously explored node, you know you're wandering around in a block.
- If you can't get back to a previous node, you've crossed a cut vertex.

Block finding code

```
sub find_lowpoint {
   my ($parent, $id, $depth) = @_;
   my $node = $graph{$id} or die;
   $node->{depth} = $depth;
   my $low = $depth;
   for my $child (keys %{$node->{edges}}) {
      next if $child eq $parent;
      my $cnode = $graph{$child} or die;
      if (defined(my $cdepth = $cnode->{depth})) {
         $low = $cdepth if $cdepth < $low;
      }
      else {
         my $clow = find_lowpoint($id, $child, $depth + 1);
         if ($clow < $depth) {
            $low = $clow if $clow < $low;
         }
         else { # found a new block
            build_block($id, $child);
         }
      }
   }
   return $low;
}
```

What does that get us?

Can use the block tree for “long-range” guidance.

BUT, road networks are mostly biconnected, so you get one big component. Not too useful.

In a computer network, a cut vertex is a single point of failure, so they tend to get engineered out of the network core. (Might use this type of algorithm to find them.)

In any case, simplifying a graph by grouping nodes together is often a useful idea.

That's all

Graphs are useful in a wide variety of situations.

But there are so many variations that it's hard to keep track of them all.

No do-everything graph module on CPAN.

There are lots of nifty algorithms that do things that might be useful.

Everyone should know Dijkstra and A*!