Announcements

- Exam 1 grades out
  - B- cutoff ~ 55
  - A- cutoff ~ 85
- Exam 2 on Friday
  - In class
  - 3Qs in 45 min
  - Covers D&C and Greedy algorithms (through last week's lectures)
- No class on Monday

Last Time

- Greedy Algorithms
- Minimum Spanning Tree
- Tree Facts

Greedy Algorithms

General Algorithmic Technique:

1. Find decision criterion
2. Make best choice according to criterion
3. Repeat until done

Surprisingly, this sometimes works.

Trees

Definition: A tree is a connected graph, with no cycles.

A spanning tree in a graph G, is a subset of the edges of G that connect all vertices and have no cycles.

If G has weights, a minimum spanning tree is a spanning tree whose total weight is as small as possible.

Basic Facts about Trees

Lemma: For an undirected graph G, any two of the below imply the third:

1. $|E| = |V|-1$
2. G is connected
3. G has no cycles

Corollary: If G is a tree, then $|E| = |V|-1$.

Today

- Minimum Spanning Trees

Minimum Spanning Tree

<u>**Problem:**</u> Given a weighted, undirected graph G, find a spanning tree of G with the lowest possible weight.

Greedy Idea

How do you make an MST?

Greedy Idea

How do you make an MST?

- Try using the cheapest edges.

Greedy Idea

How do you make an MST?

- Try using the cheapest edges.

<u>Proposition:</u> In a graph G, let e be an edge of lightest weight. Then there exists an MST of G containing e. Furthermore, if e is the unique lightest edge, then *all* MSTs contain e.

Proof Idea

- Suppose that we have an MST T that *does not* contain e.

Proof Idea

- Suppose that we have an MST T that *does not* contain e.
- Modify T to get T' that does contain e and has wt(T') ≤ wt(T).

Proof Idea

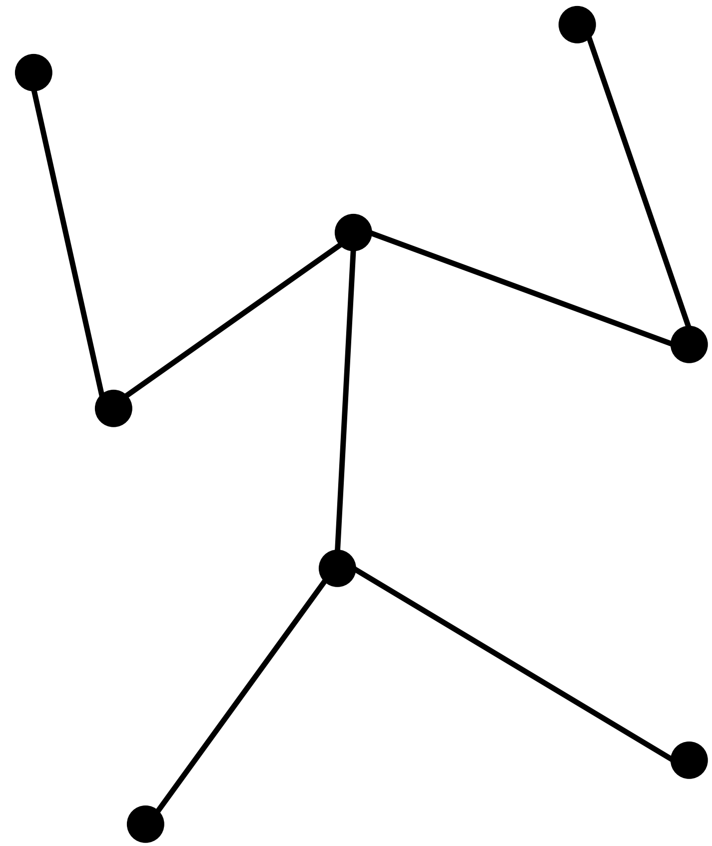
- Suppose that we have an MST T that *does not* contain e.
- Modify T to get T’ that does contain e and has wt(T’) ≤ wt(T).
- T’ will be a MST as well.

Proof Idea

- Suppose that we have an MST T that *does not* contain e.
- Modify T to get T' that does contain e and has wt(T') ≤ wt(T).
- T' will be a MST as well.
- Furthermore if e is the unique lightest edge, wt(T') < wt(T), so T could not have been minimal.

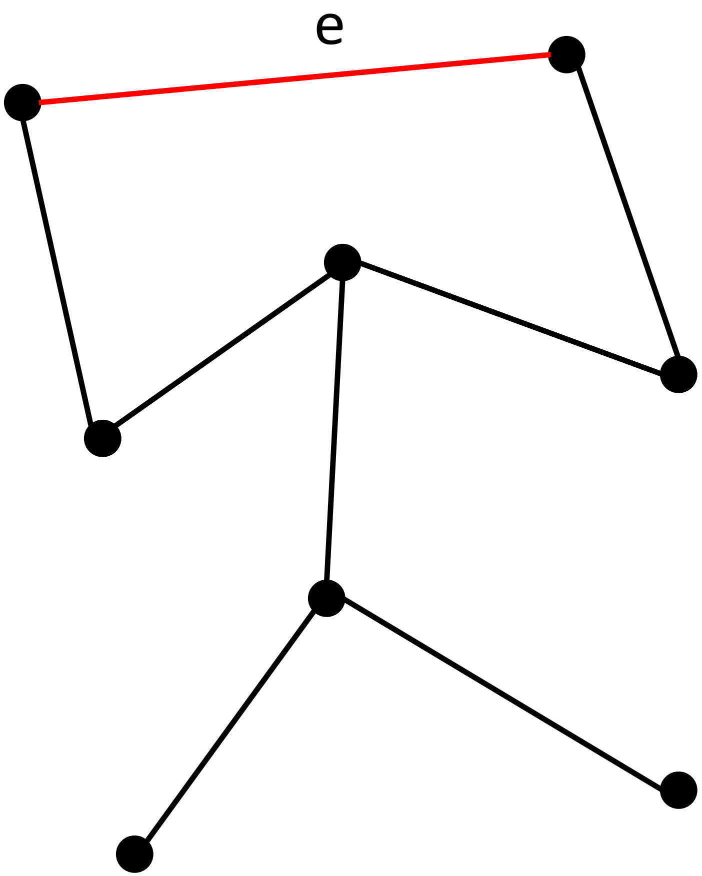
Proof

- Consider tree T not containing e.



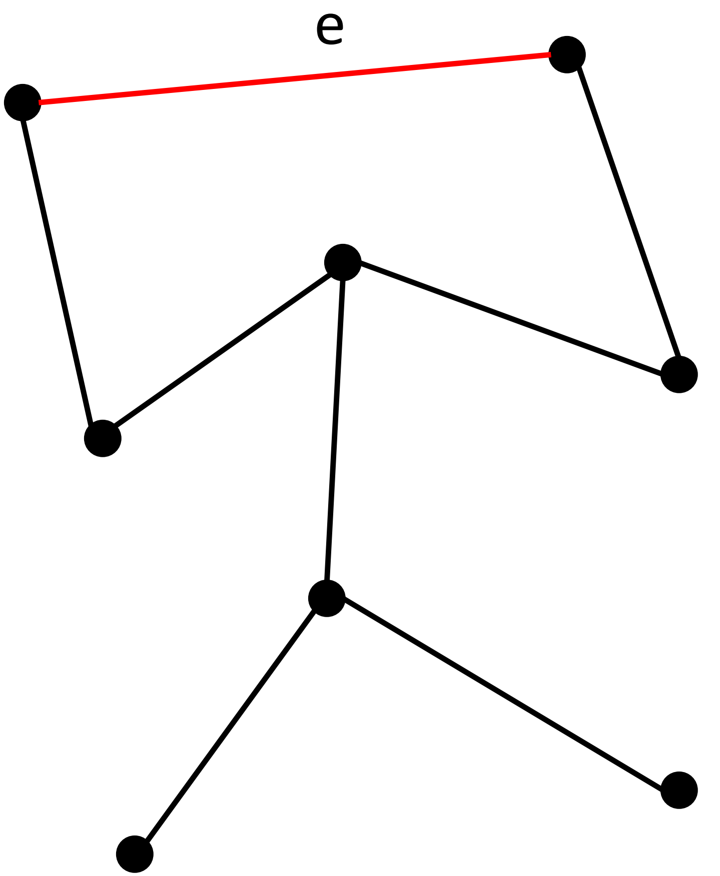
Proof

- Consider tree T not containing e.



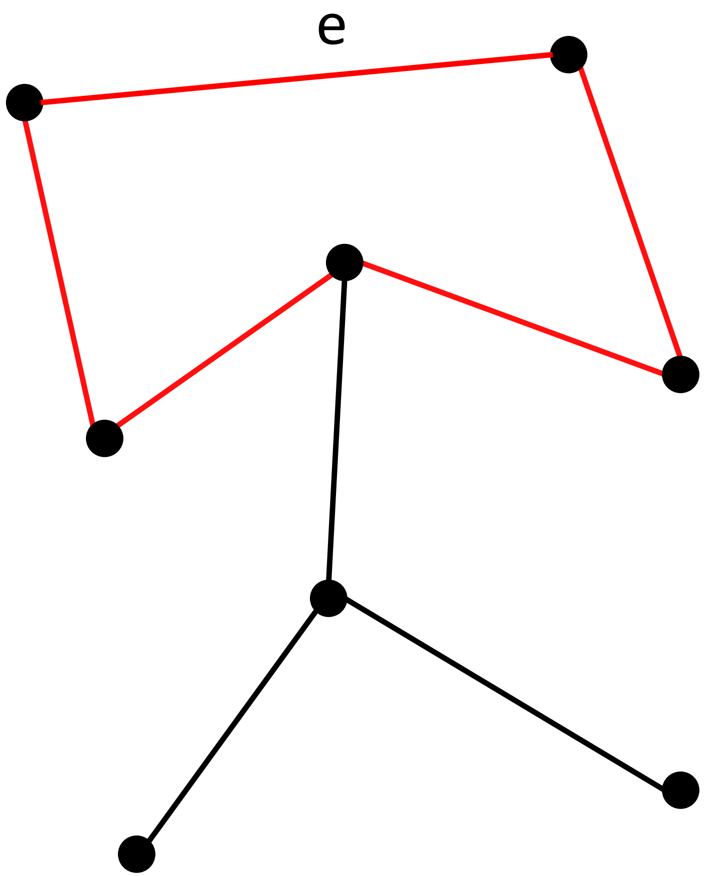
Proof

- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.



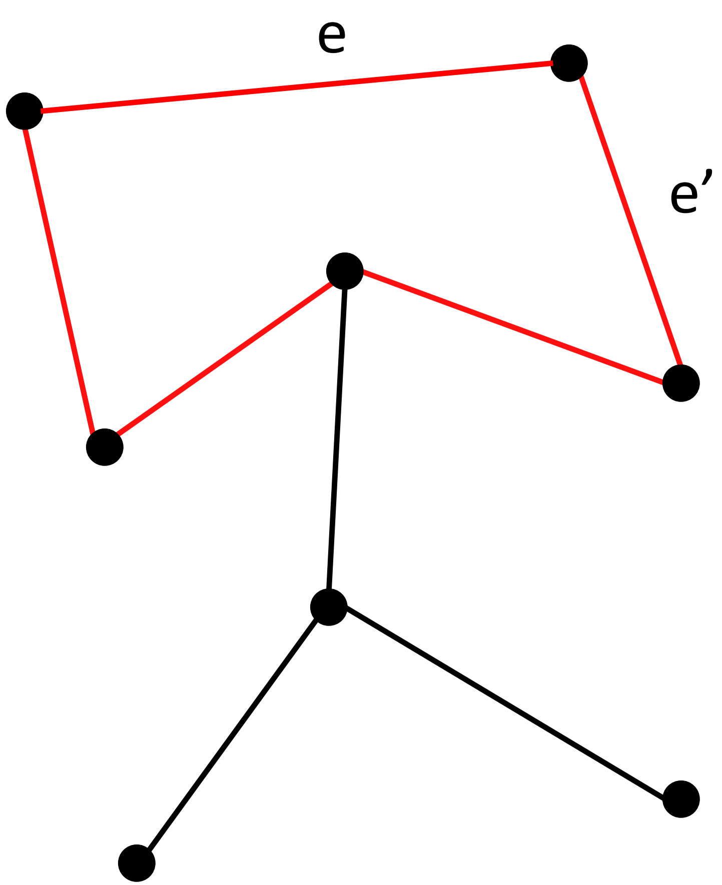
Proof

- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.



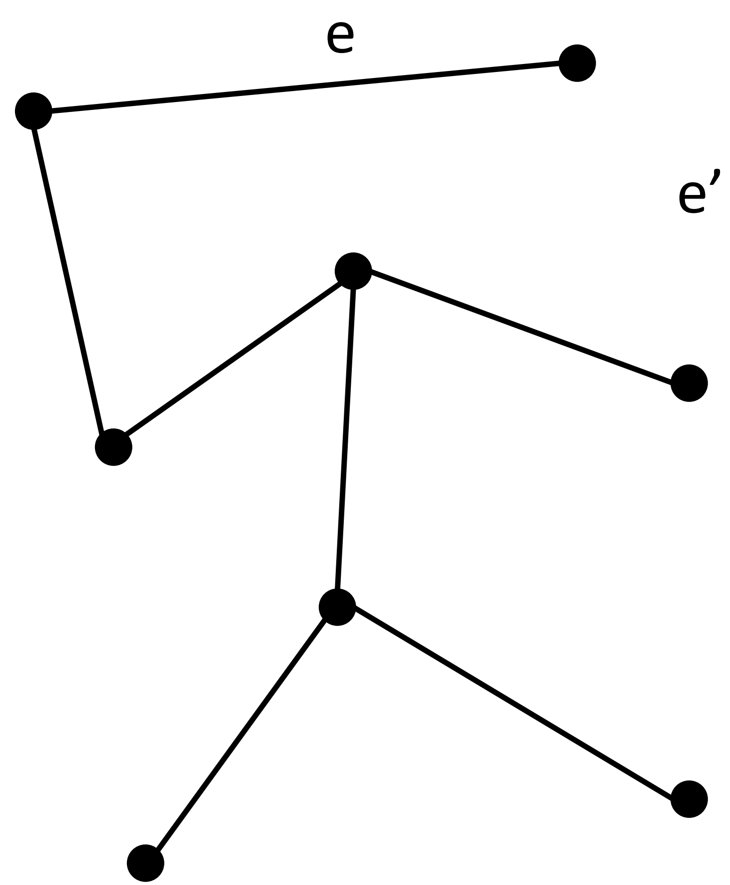
Proof

- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.
- Remove edge e’ from cycle to get T’.



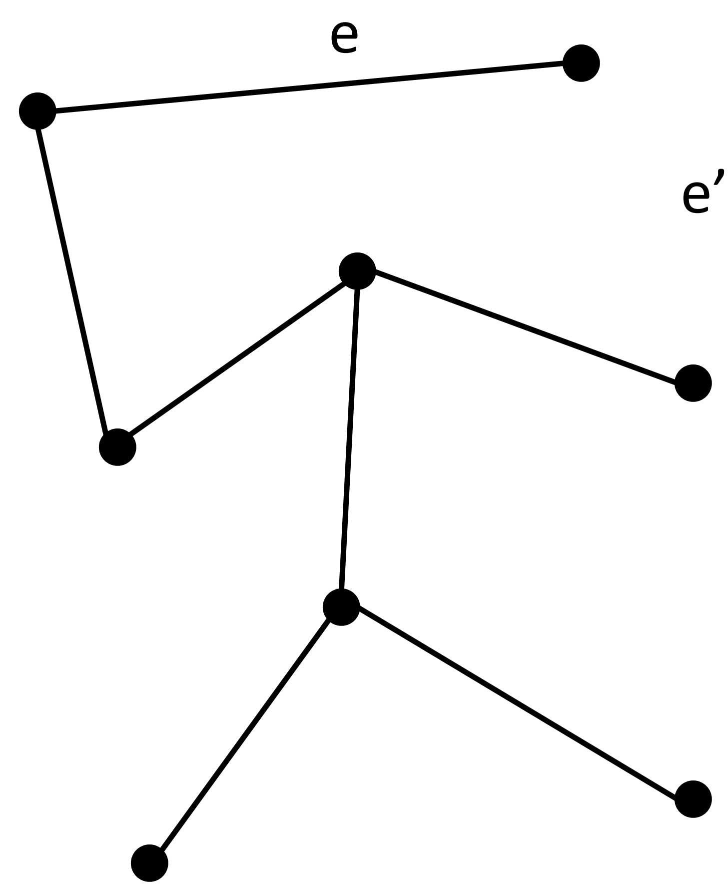
Proof

- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.
- Remove edge e’ from cycle to get T’.



Proof

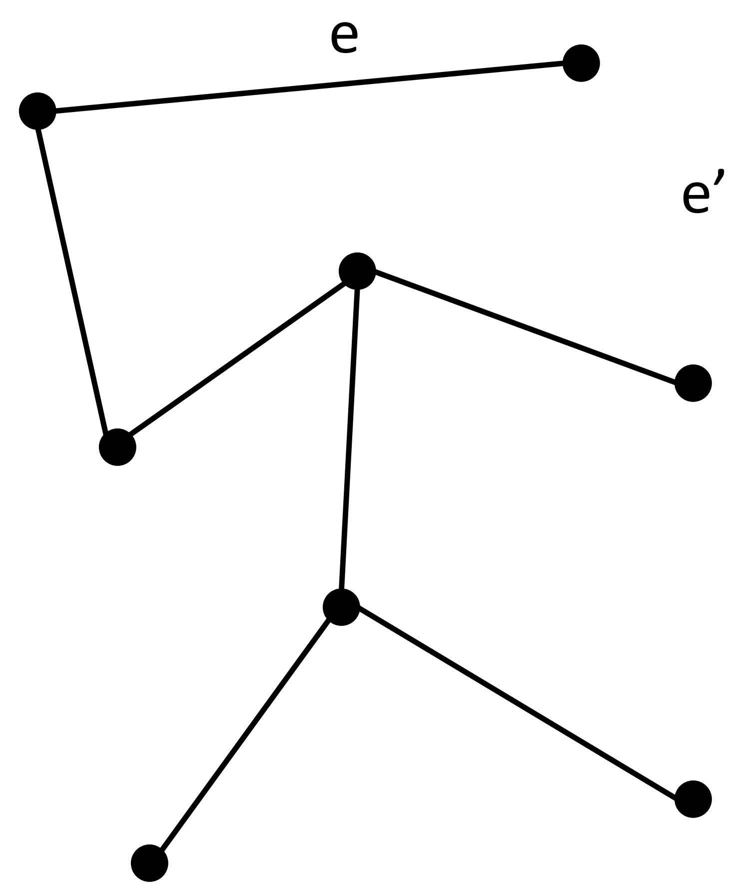
- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.
- Remove edge e’ from cycle to get T’.
- |T’|=|V|-1, and connected, so T’ is a tree.



Proof

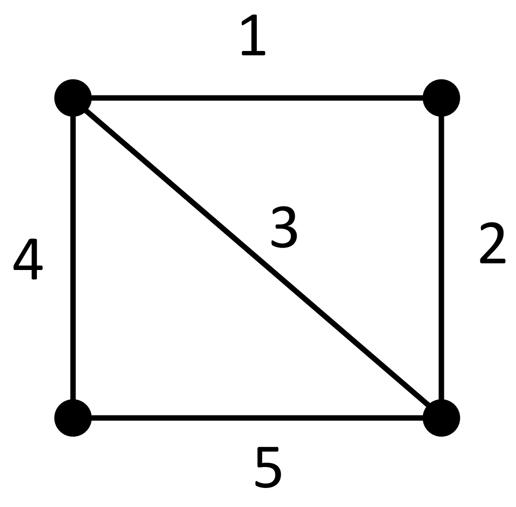
- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.
- Remove edge e’ from cycle to get T’.
- $|T'|=|V|-1$, and connected, so T’ is a tree.
- $wt(T') = wt(T)+wt(e)-wt(e')$
  $\leq wt(T)$

(because wt(e) is minimal).



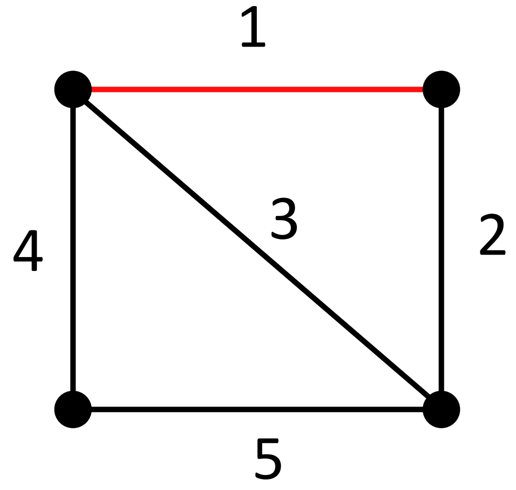
Example

- Lightest edge in MST.



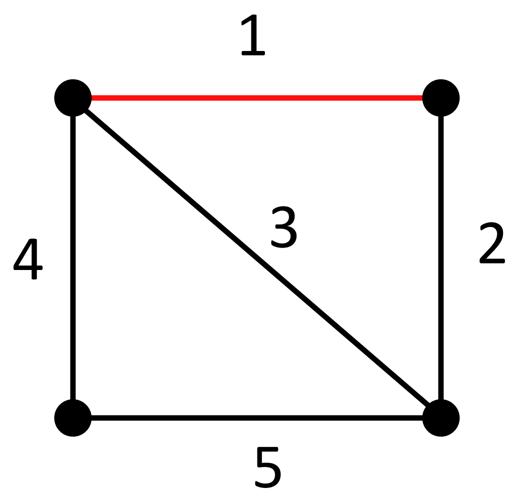
Example

- Lightest edge in MST.



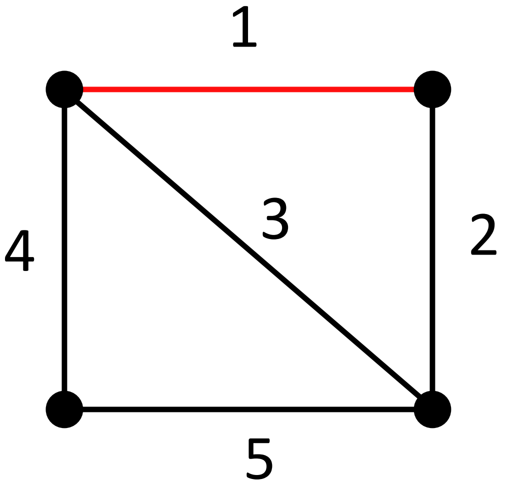
Example

- Lightest edge in MST.
  - Then what?



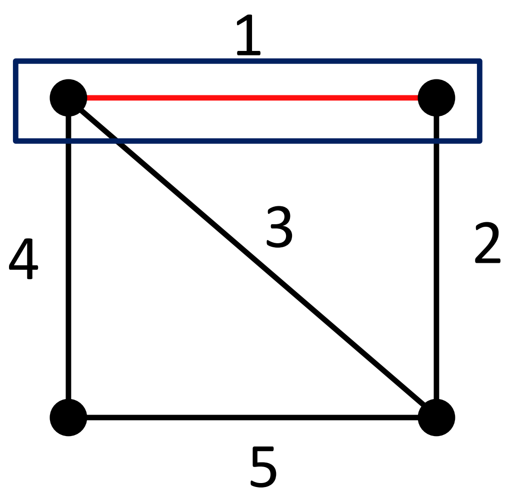
Example

- Lightest edge in MST.
  - Then what?
- Merge those vertices & repeat.



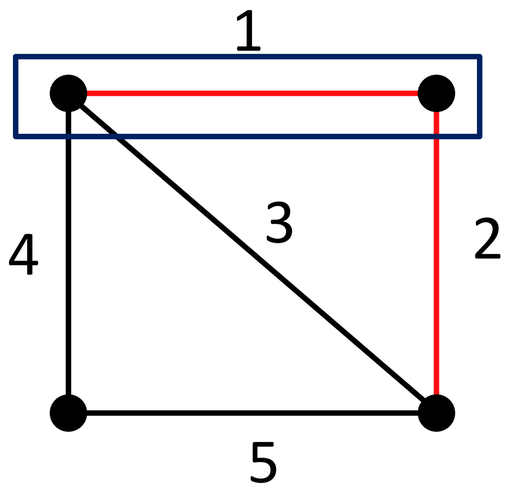
Example

- Lightest edge in MST.
  - Then what?
- Merge those vertices & repeat.



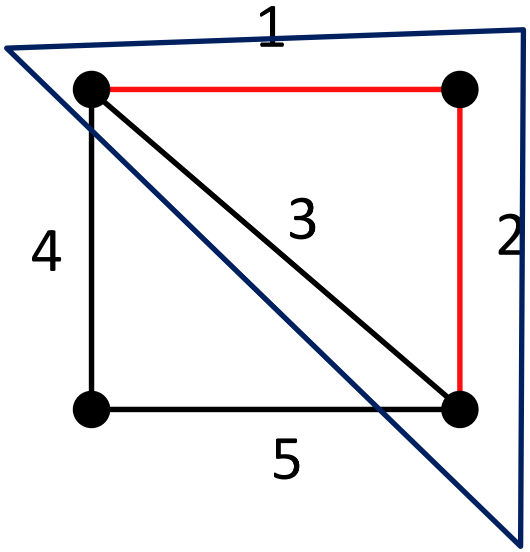
Example

- Lightest edge in MST.
  - Then what?
- Merge those vertices & repeat.



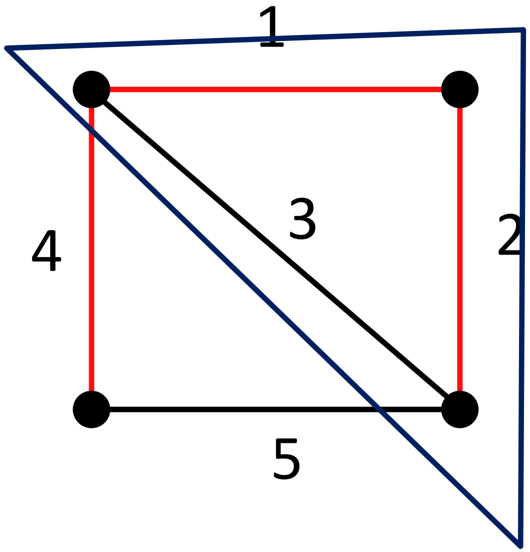
Example

- Lightest edge in MST.
  - Then what?
- Merge those vertices & repeat.



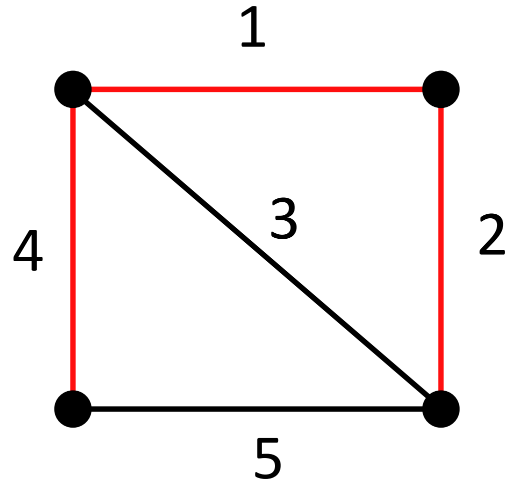
Example

- Lightest edge in MST.
  - Then what?
- Merge those vertices & repeat.



Example

- Lightest edge in MST.
  - Then what?
- Merge those vertices & repeat.



Algorithm

- When more than one vertex, add lightest edge, and merge.
  - Repeat and then undo merges.
- Easier: An edge hasn’t been merged away iff it does not create a cycle with already chosen edges.

Algorithm

```
Kruskal(G)
  T ← {}
  While(|T| < |V|-1)
    Find lightest edge e that
      doesn’t create cycle with T
    Add e to T
  Return T
```

Algorithm

```
Kruskal(G)
  T ← {}
  While(|T| < |V|-1)
    Find lightest edge e that
      doesn't create cycle with T
    Add e to T
  Return T
```

O(|V|) Iterations

Algorithm

```
Kruskal(G)
  T ← {}
  While(|T| < |V|-1)
    Find lightest edge e that
     doesn't create cycle with T
    Add e to T
  Return T
```

O(|V|) Iterations

O(|E|) edges

Algorithm

```
Kruskal(G)
  T ← {}
  While(|T| < |V|-1)
    Find lightest edge e that
     doesn't create cycle with T
    Add e to T
  Return T
```

O(|V|) Iterations

O(|E|) edges

O(|V|+|E|) time to check for cycle

Algorithm

```
Kruskal(G)
  T ← {}
  While(|T| < |V|-1)
    Find lightest edge e that
     doesn't create cycle with T
    Add e to T
  Return T
```

O(|V|) Iterations

O(|E|) edges

O(|V|+|E|) time to check for cycle

Runtime: O(|V||E|²)

Optimizations

Two things are slow here:

1) Testing every edge every iteration.
2) Needing to test connectivity for every edge.

Optimizations

Two things are slow here:

1) Testing every edge every iteration.
2) Needing to test connectivity for every edge.

To improve (1), if an edge forms a cycle, it will never later become viable.

Sort edges once and use in order.

Kruskal Version 2

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  For e ∈ E in increasing order
    If e does not form cycle
      Add e to T
  Return T
```

Kruskal Version 2

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  For e ∈ E in increasing order
    If e does not form cycle
      Add e to T
  Return T
```

O(|E| log|E|)

Kruskal Version 2

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  For e ∈ E in increasing order
    If e does not form cycle
      Add e to T
  Return T
```

O(|E| log|E|)

O(|E|) Iterations

Kruskal Version 2

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  For e ∈ E in increasing order
    If e does not form cycle
      Add e to T
  Return T
```

O(|E| log|E|)

O(|E|) Iterations

O(|V|+|E|)

Kruskal Version 2

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  For e ∈ E in increasing order
    If e does not form cycle
      Add e to T
  Return T
```

$O(|E| \log|E|)$

$O(|E|)$ Iterations

$O(|V|+|E|)$

Runtime: $O(|E|^2)$

Better Cycle Testing

How do we test if edge (v,w) forms a cycle?

Better Cycle Testing

How do we test if edge (v,w) forms a cycle?

If v and w are in the same connected component of the graph formed by T.

Better Cycle Testing

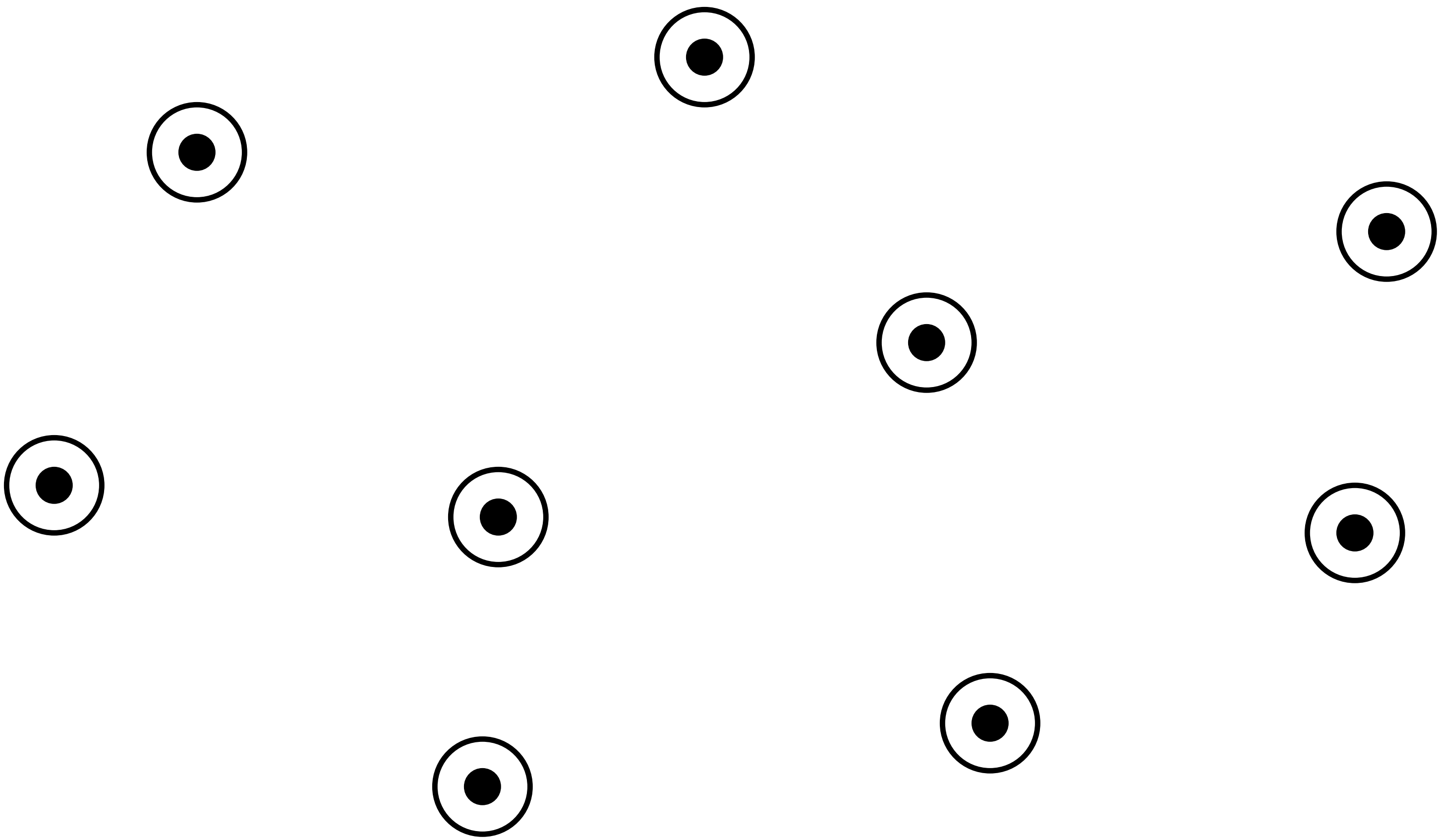
How do we test if edge (v,w) forms a cycle?

If v and w are in the same connected component of the graph formed by T.

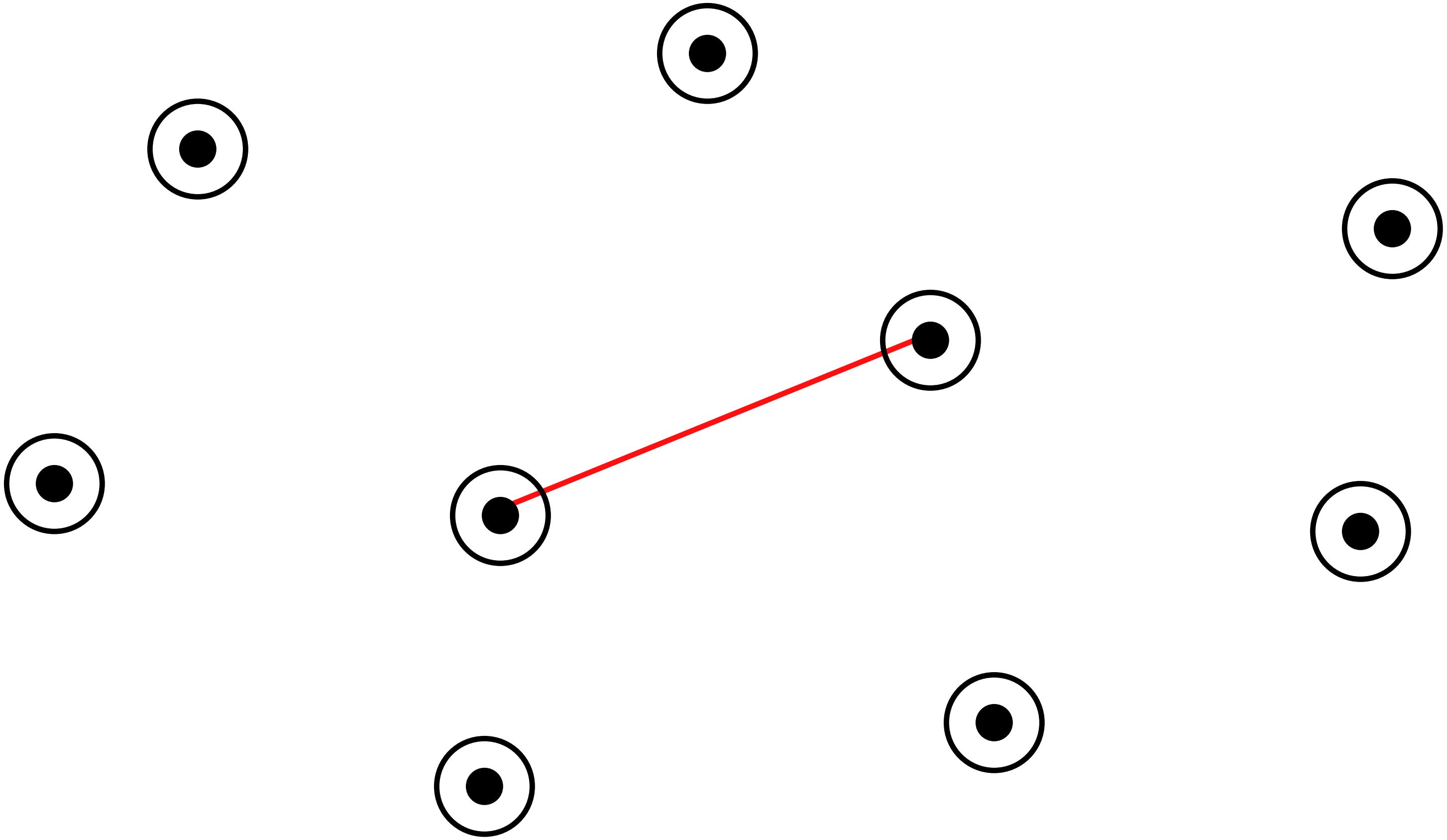
Need a data structure. That can:

- Add edges to T.
- Test if two vertices in same CC.

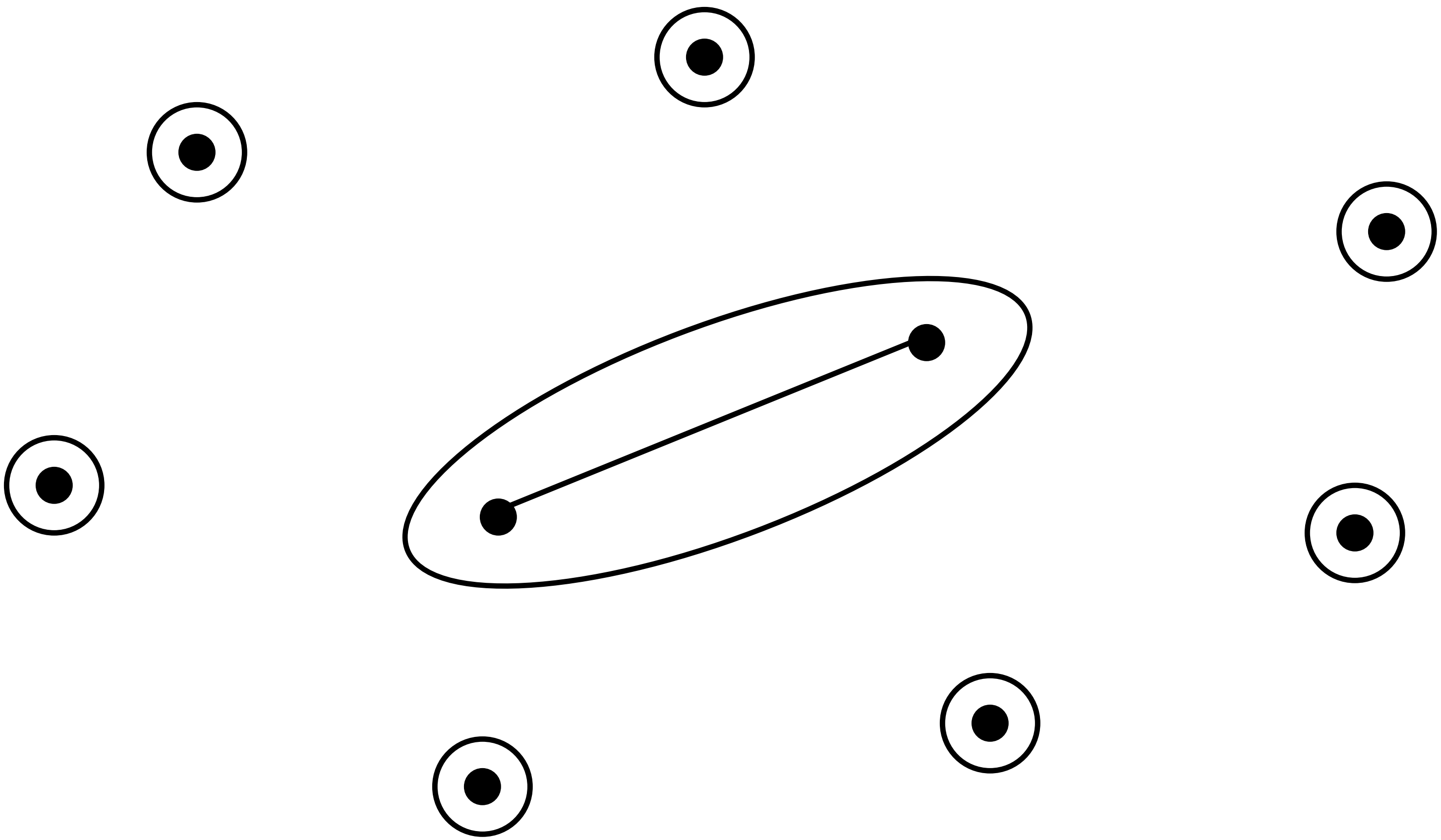
Components



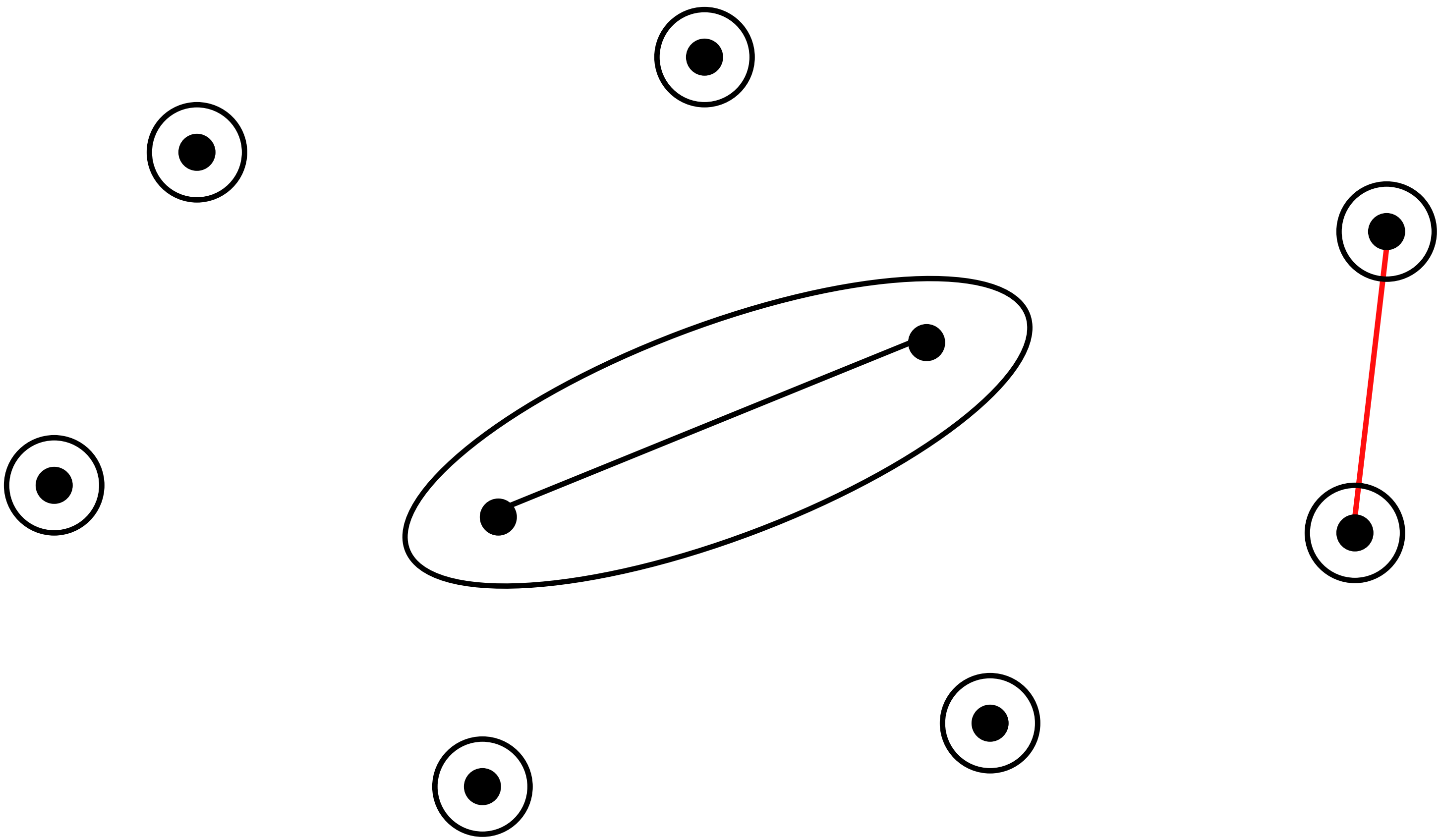
Components



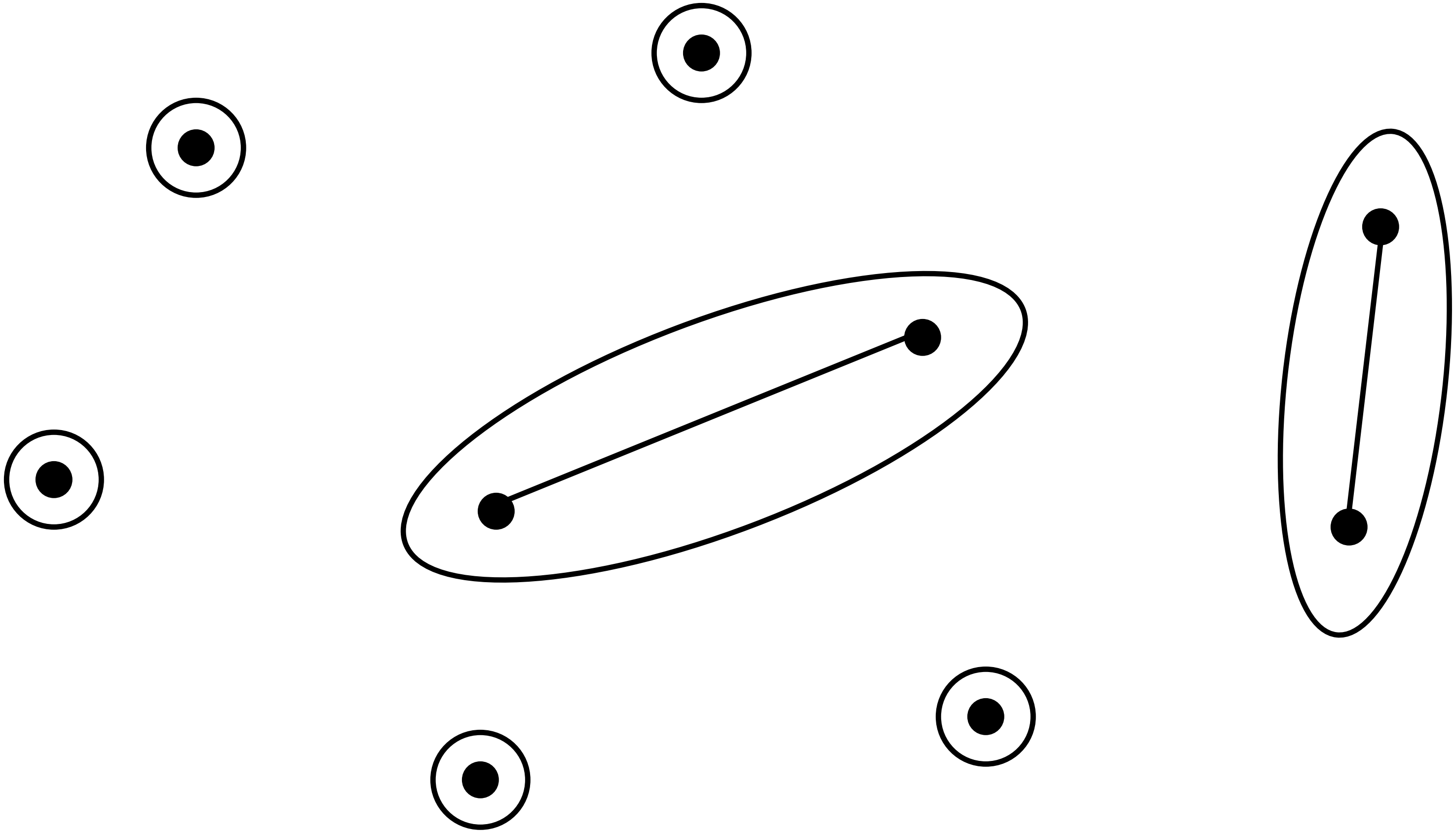
Components



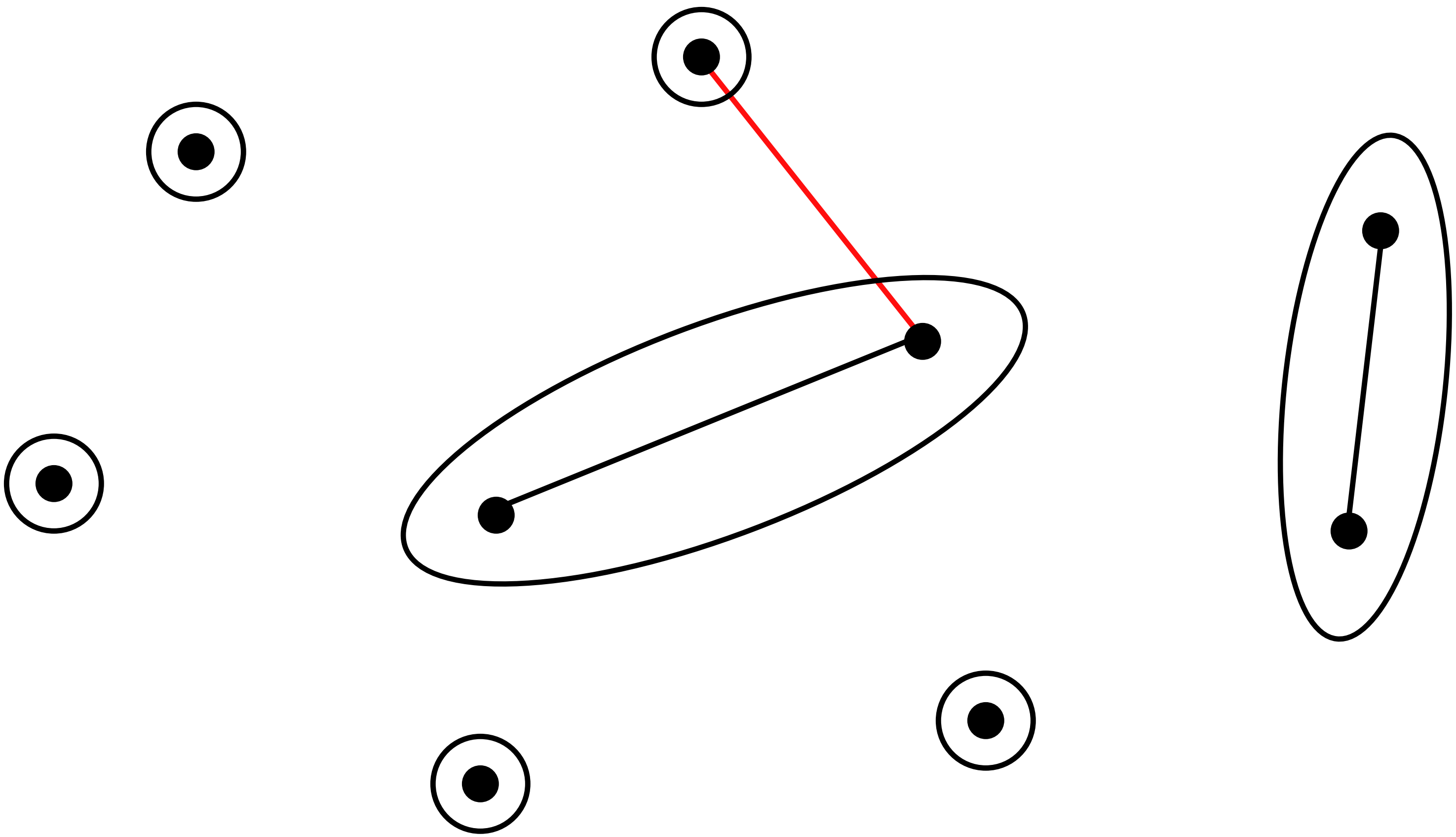
Components



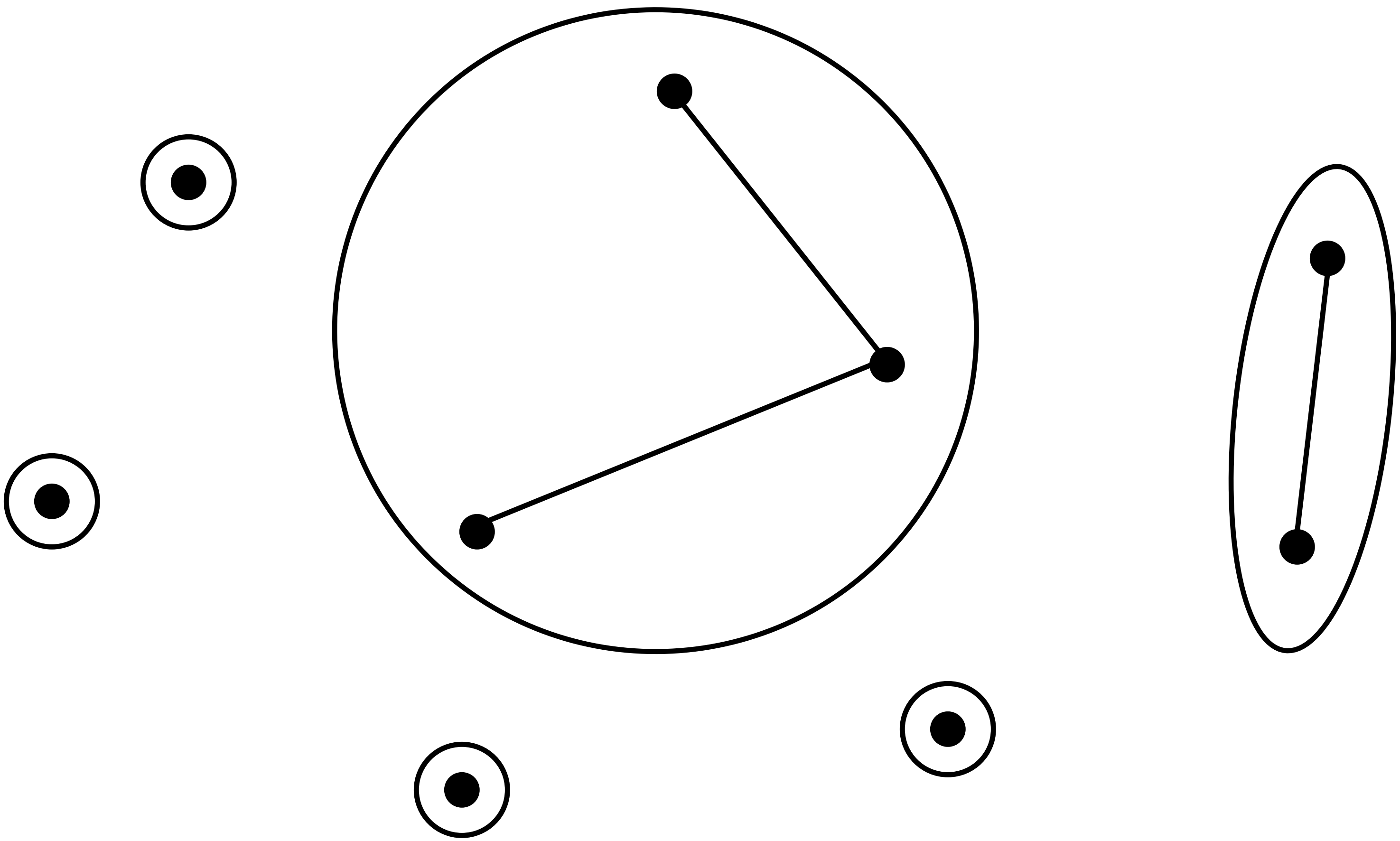
Components



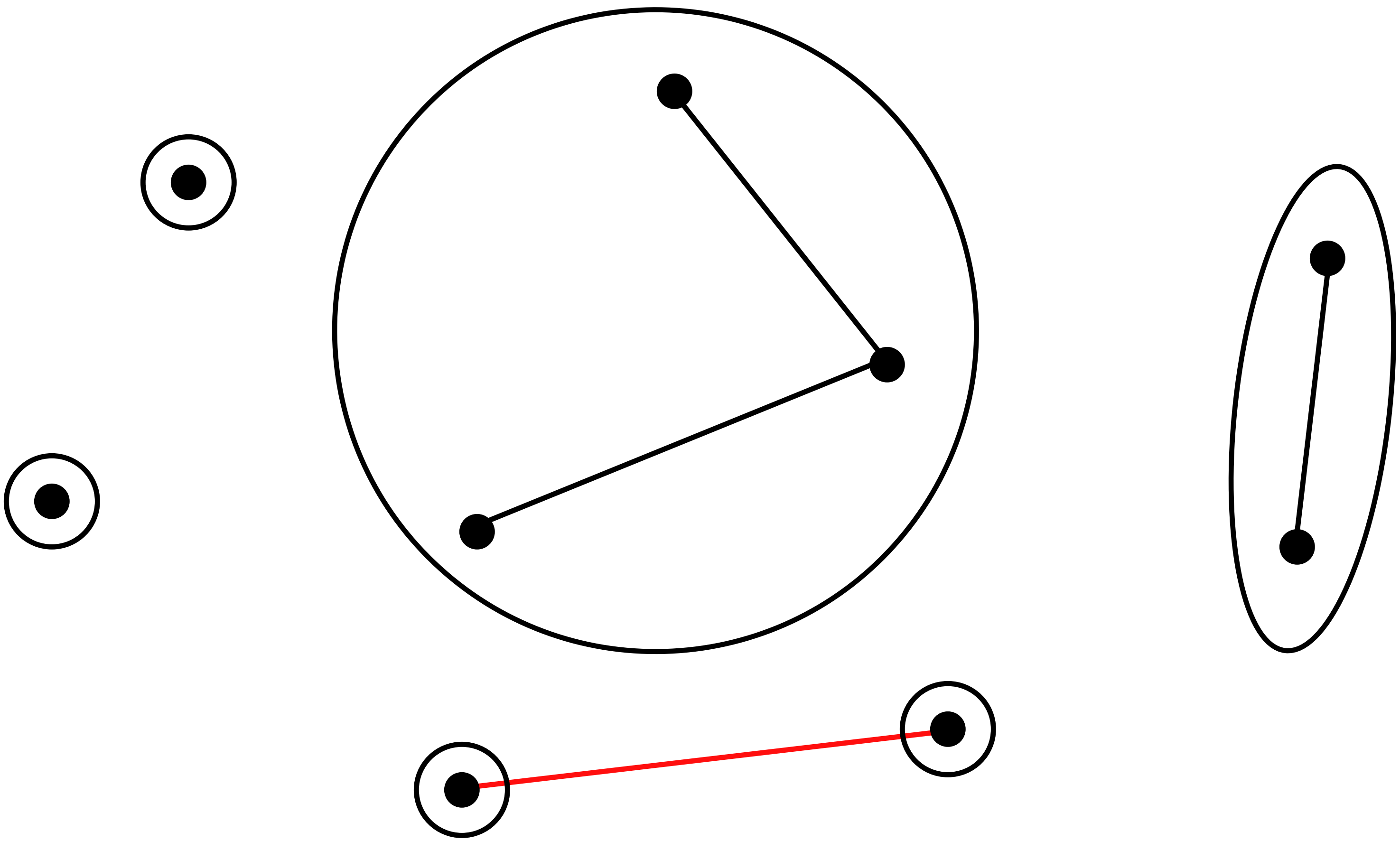
Components



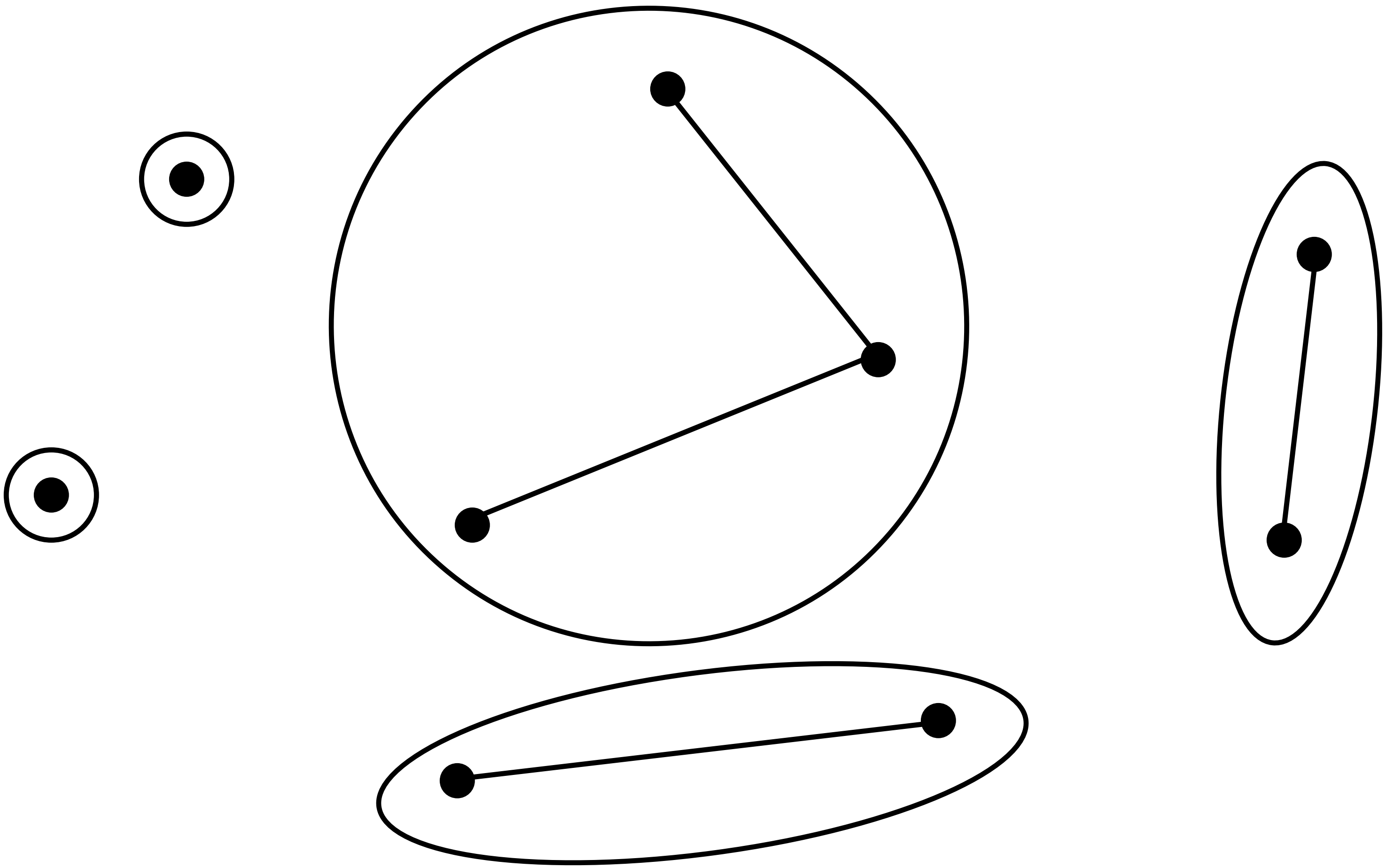
Components



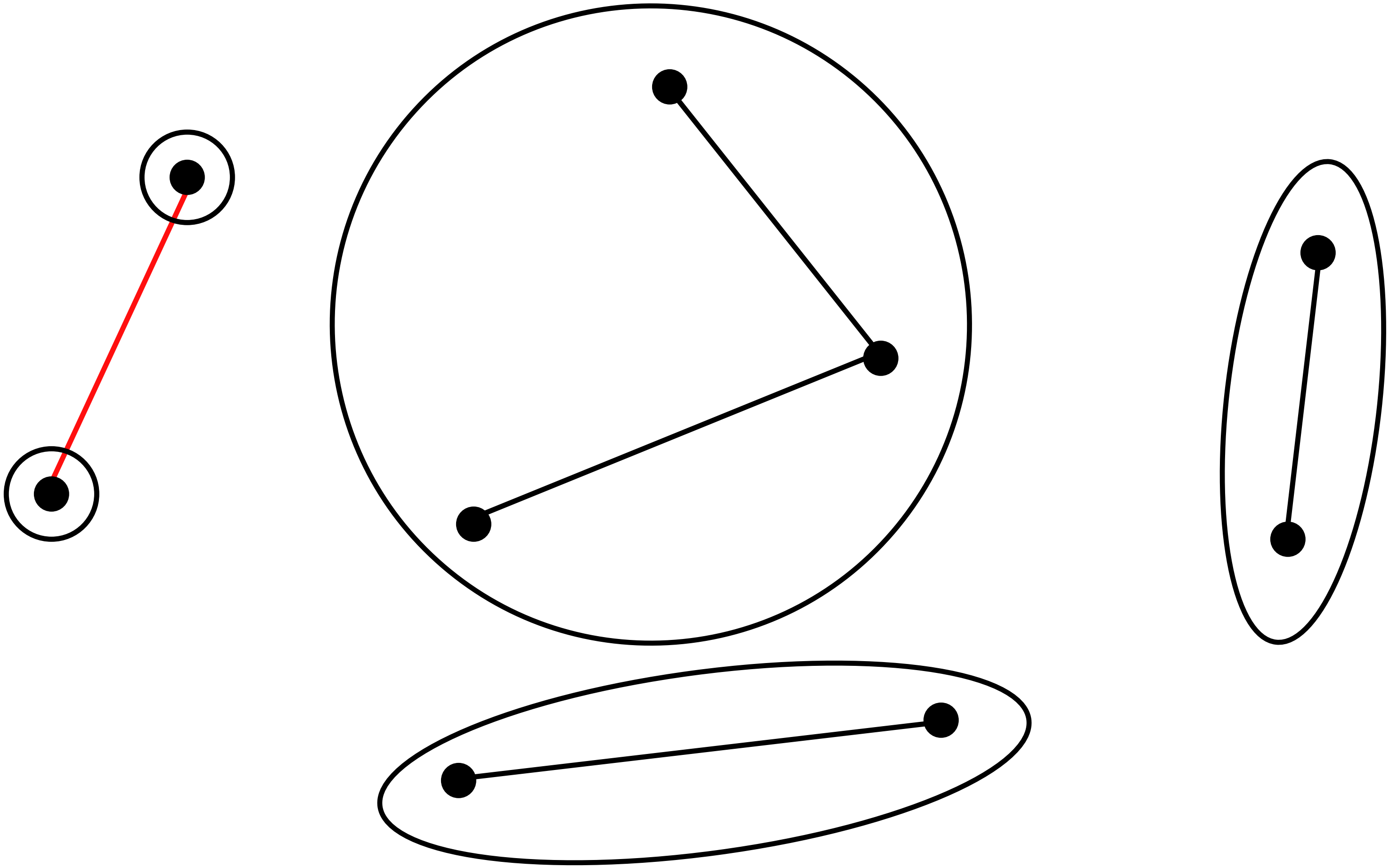
Components



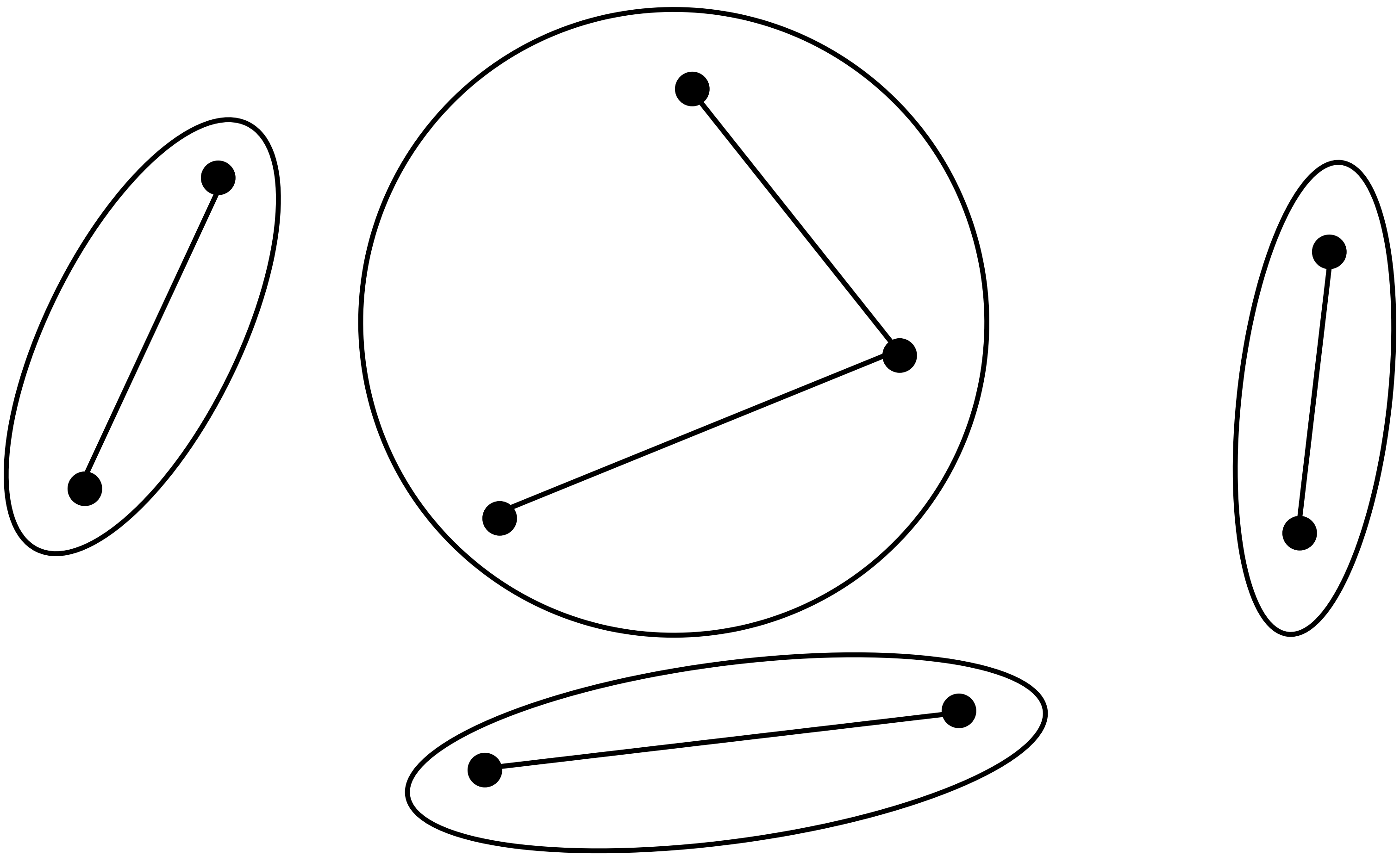
Components



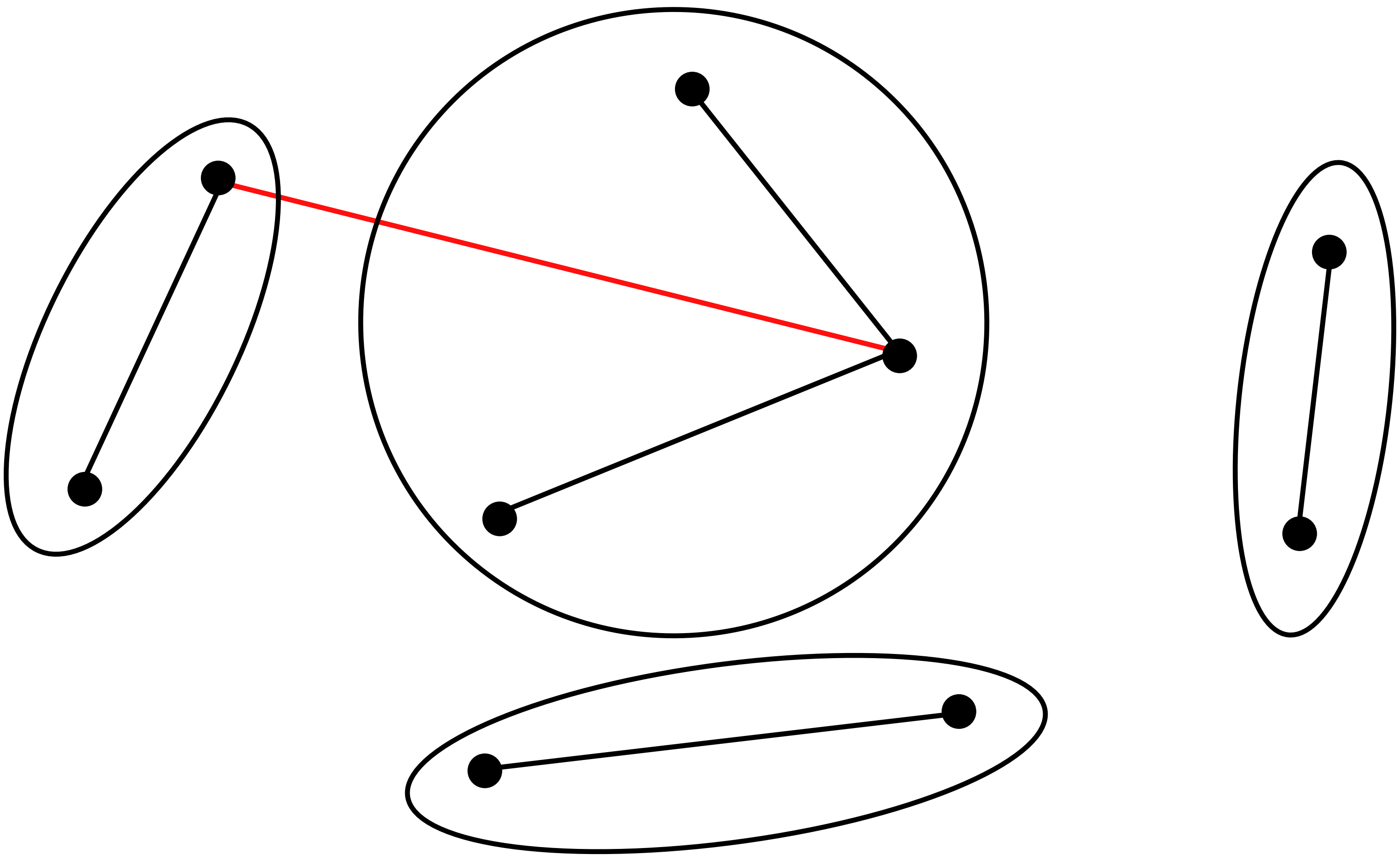
Components



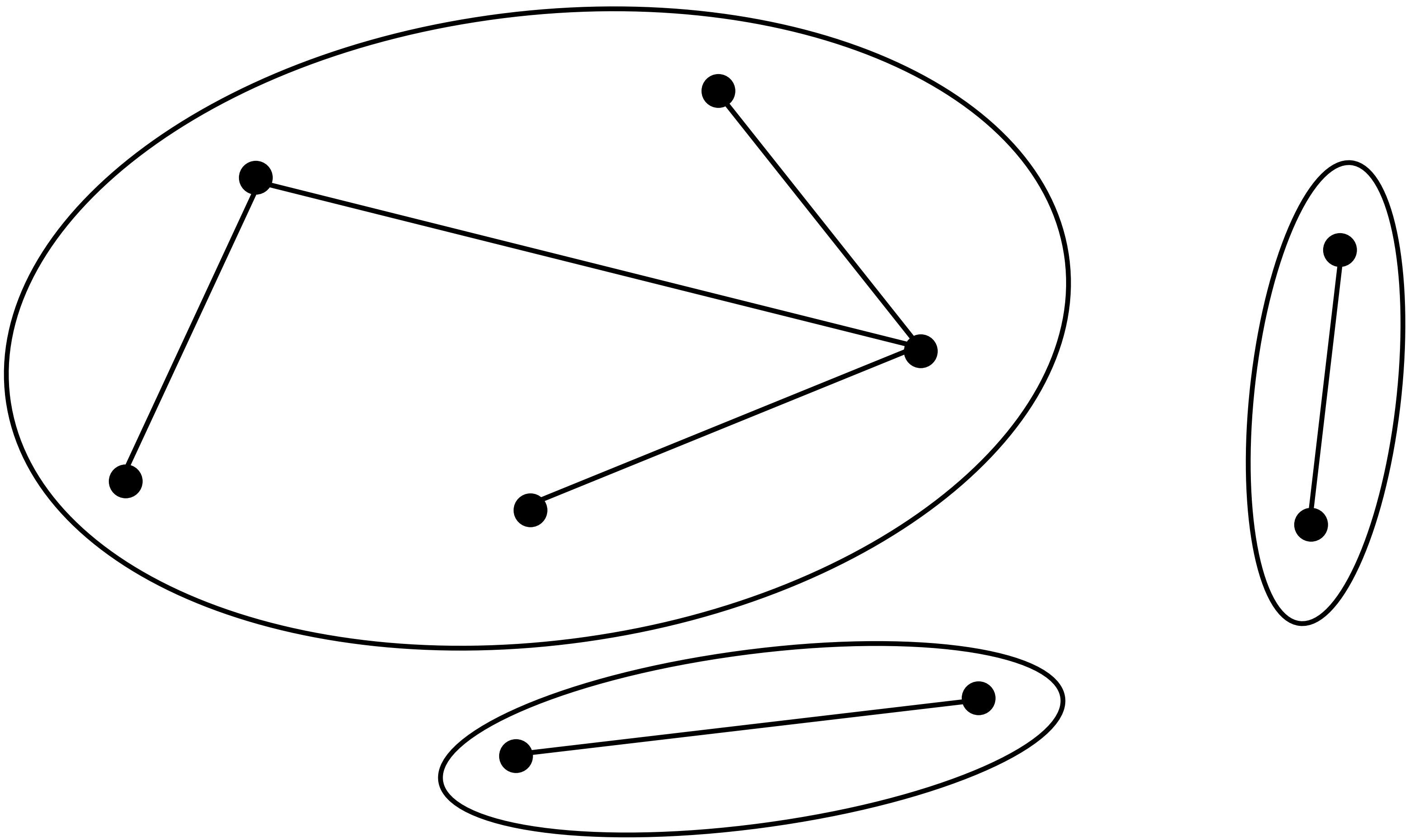
Components



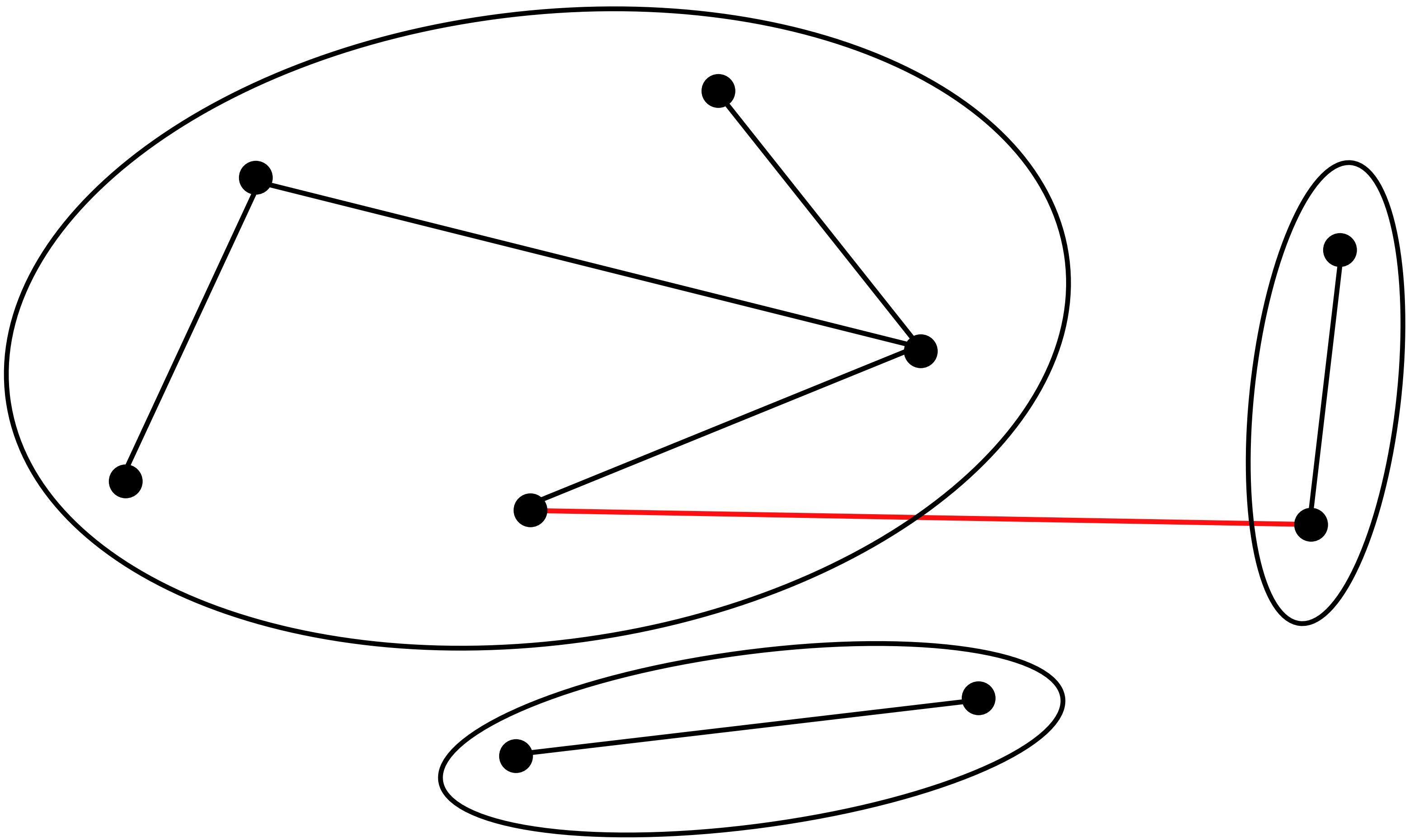
Components



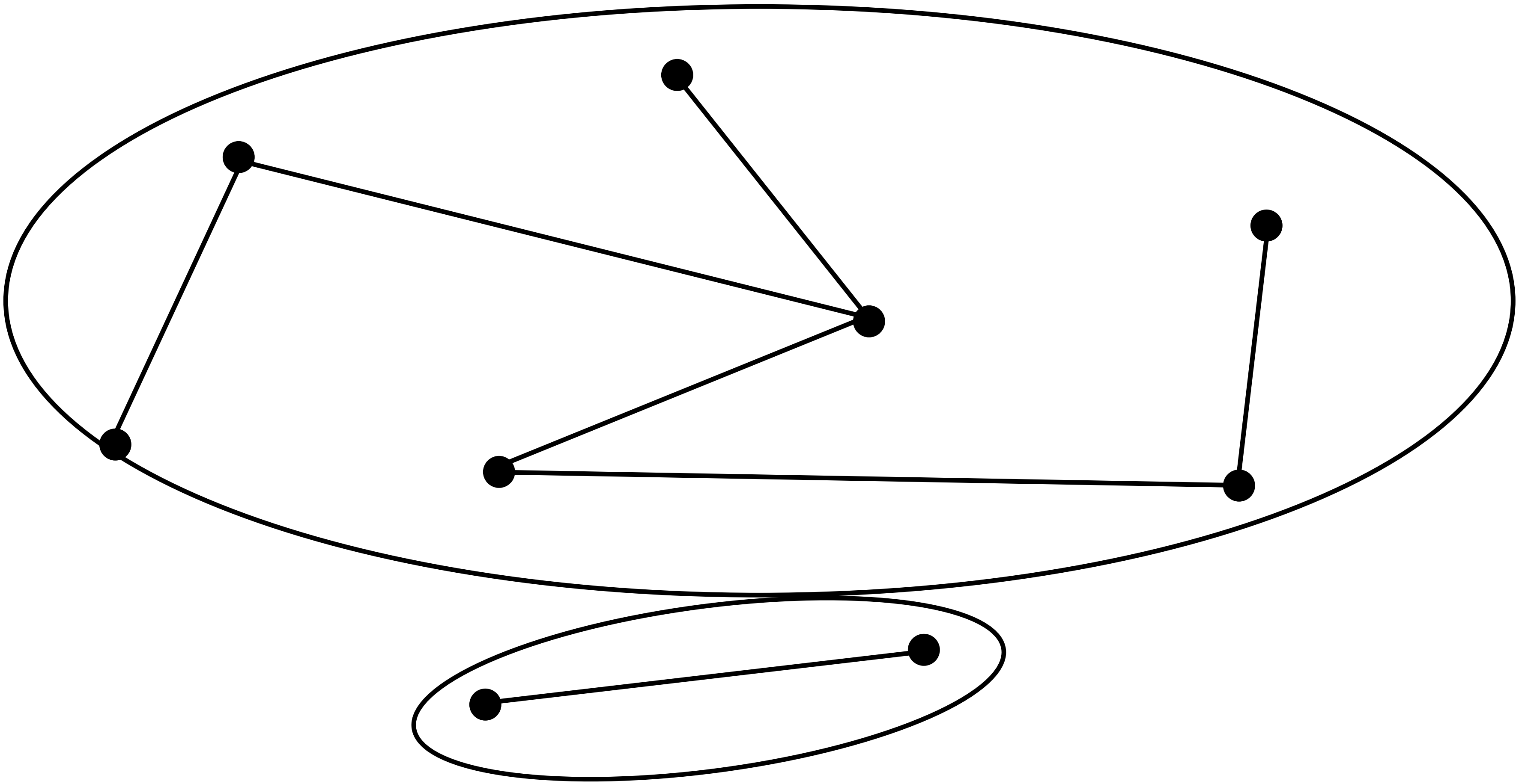
Components



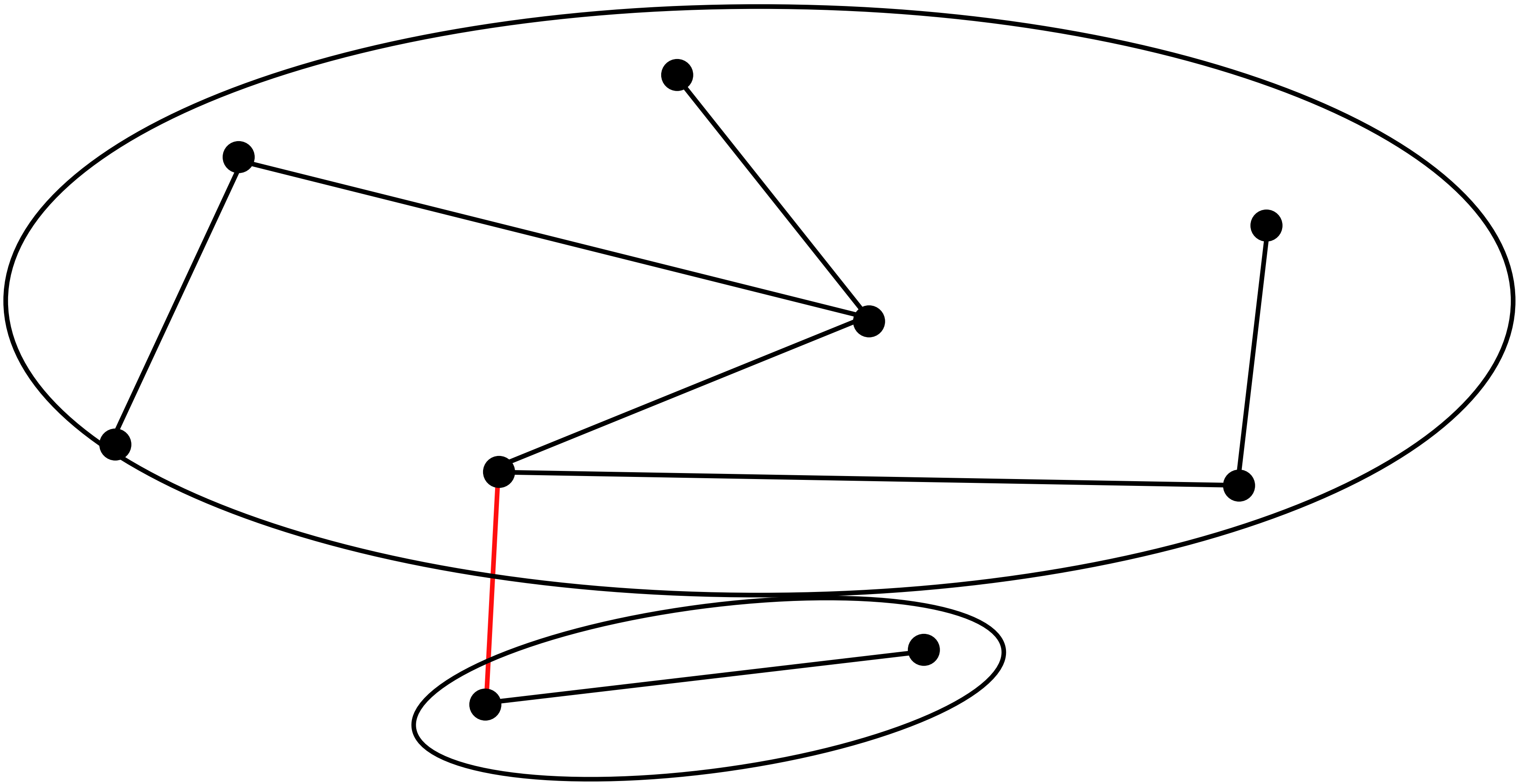
Components



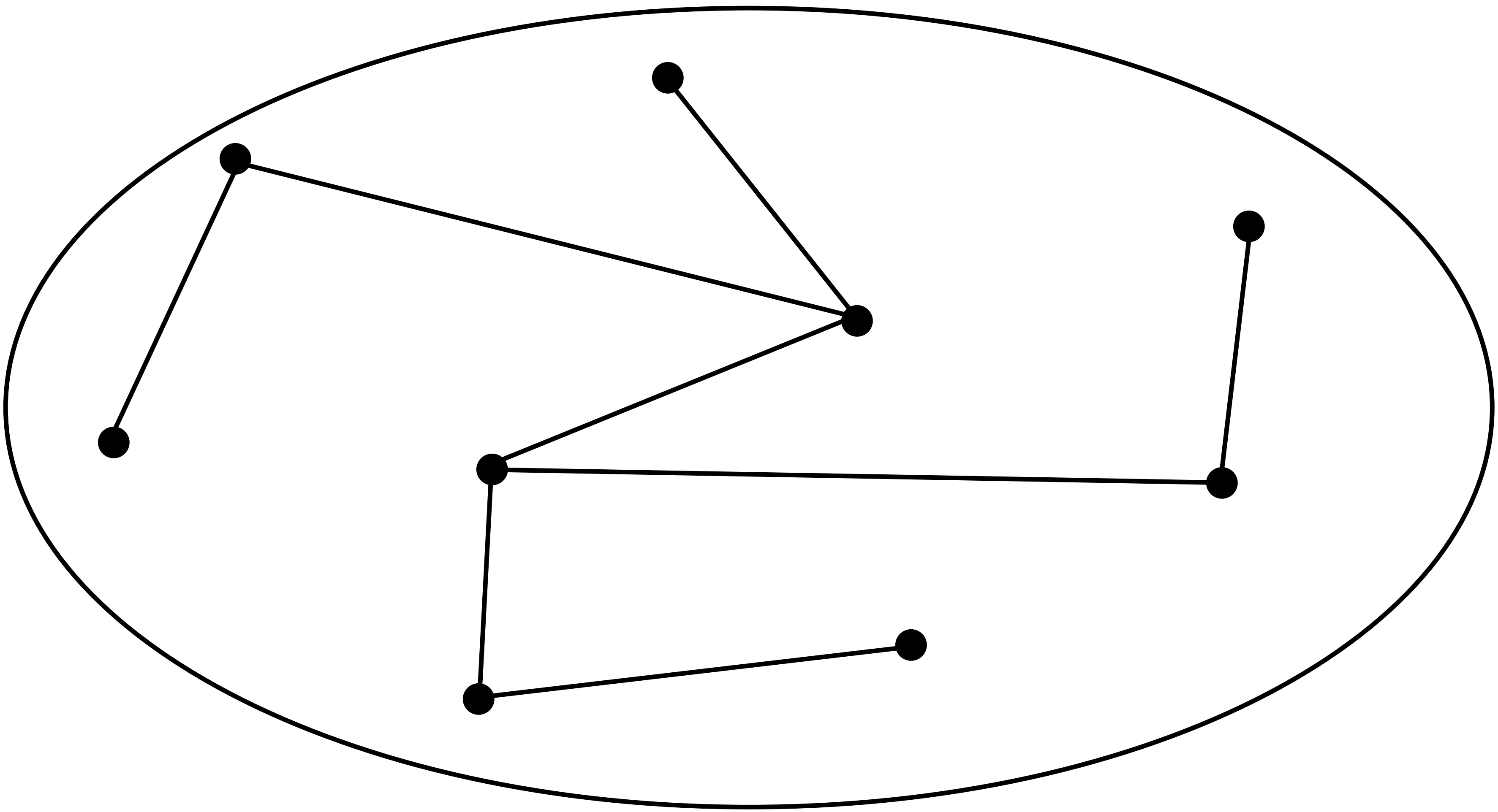
Components



Components



Components



Union Find Data Structure

Maintains several sets. Each has a representative element.

Union Find Data Structure

Maintains several sets. Each has a representative element.

<u>Operations:</u>

- New(e) – Creates a new set with element e.

Union Find Data Structure

Maintains several sets. Each has a representative element.

<u>Operations:</u>

- New(e) – Creates a new set with element e.
- Rep(a) – Returns the representative element of a's set.

Union Find Data Structure

Maintains several sets. Each has a representative element.

Operations:

- New(e) – Creates a new set with element e.
- Rep(a) – Returns the representative element of a's set.
- Join(a,b) – Merges a's set with b's.

Union Find Data Structure

Maintains several sets. Each has a representative element.

<u>Operations:</u>

- New(e) – Creates a new set with element e.
- Rep(a) – Returns the representative element of a's set.
- Join(a,b) – Merges a's set with b's.

<u>Note:</u> Check of v & w in same set by testing if Rep(v) = Rep(w).

Kruskal Version 3

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  Create Union Find
  For v ∈ V, New(v)
  For (v,w) ∈ E in increasing order
    If Rep(v) ≠ Rep(w)
      Add (v,w) to T
      Join(v,w)
  Return T
```

Kruskal Version 3

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  Create Union Find
  For v ∈ V, New(v)
  For (v,w) ∈ E in increasing order
    If Rep(v) ≠ Rep(w)
      Add (v,w) to T
      Join(v,w)
  Return T
```

$O(|E| \log|E|)$

Kruskal Version 3

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  Create Union Find
  For v ∈ V, New(v)
  For (v,w) ∈ E in increasing order
    If Rep(v) ≠ Rep(w)
      Add (v,w) to T
      Join(v,w)
  Return T
```

O(|E| log|E|)

O(|V|) New's

Kruskal Version 3

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  Create Union Find
  For v ∈ V, New(v)
  For (v,w) ∈ E in increasing order
    If Rep(v) ≠ Rep(w)
      Add (v,w) to T
      Join(v,w)
  Return T
```

O(|E| log|E|)

O(|V|) New's

O(|E|) Iterations

Kruskal Version 3

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  Create Union Find
  For v ∈ V, New(v)
  For (v,w) ∈ E in increasing order
    If Rep(v) ≠ Rep(w)
      Add (v,w) to T
      Join(v,w)
  Return T
```

O(|E| log|E|)

O(|V|) New's

O(|E|) Iterations

O(1) Join + Rep

Kruskal Version 3

```
Kruskal(G)
  Sort edges by weight
  T ← {}
  Create Union Find
  For v ∈ V, New(v)
  For (v,w) ∈ E in increasing order
    If Rep(v) ≠ Rep(w)
      Add (v,w) to T
      Join(v,w)
  Return T
```

O(|E| log|E|)

O(|V|) New's

O(|E|) Iterations

O(1) Join + Rep

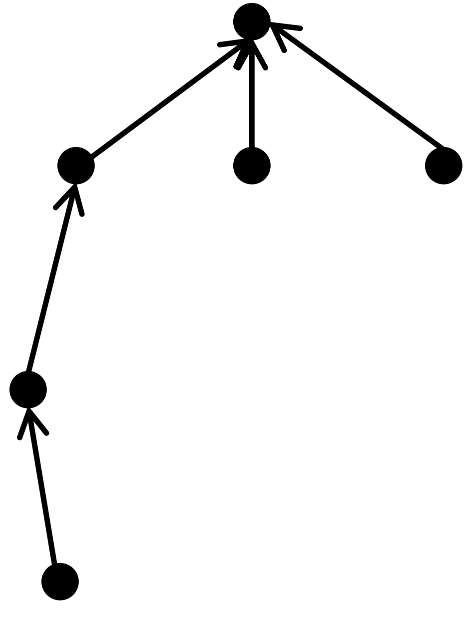
Runtime:O(|E|log|E|) +|E|(Union-Find Ops)

Union Find Implementation

<u>Basic Idea:</u> Each set is a rooted tree with edges pointing towards the representative.

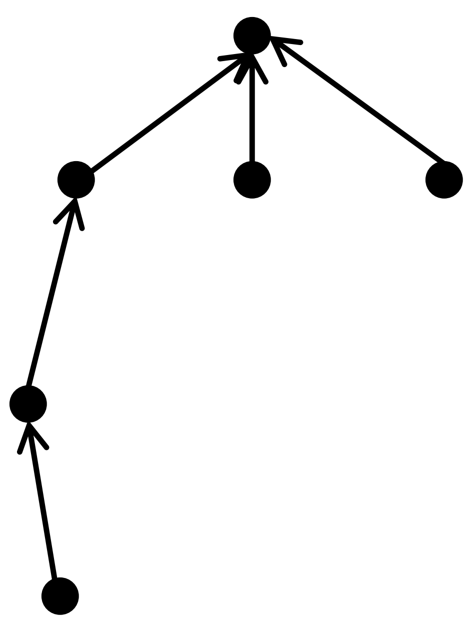
Union Find Implementation

Basic Idea: Each set is a rooted tree with edges pointing towards the representative.



Union Find Implementation

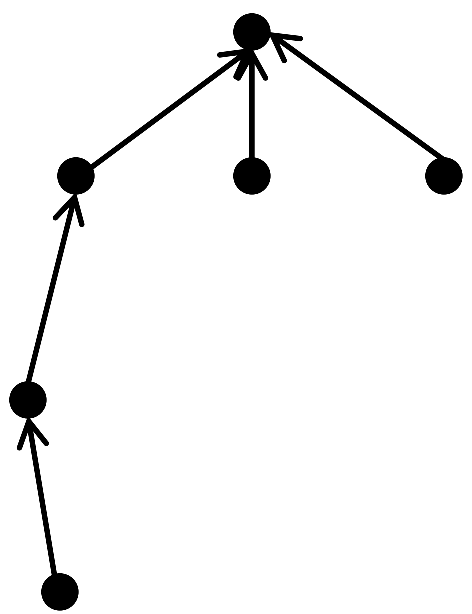
<u>Basic Idea:</u> Each set is a rooted tree with edges pointing towards the representative.



<u>New:</u> Create new node – O(1)

Union Find Implementation

Basic Idea: Each set is a rooted tree with edges pointing towards the representative.



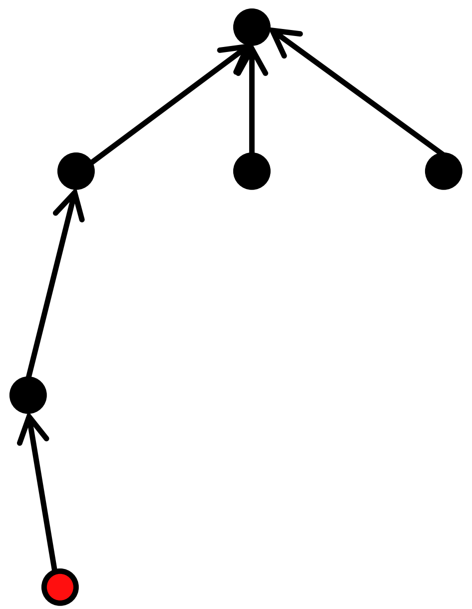
New: Create new node – O(1)

Rep: Follow pointers to root

- O(depth)

Union Find Implementation

Basic Idea: Each set is a rooted tree with edges pointing towards the representative.

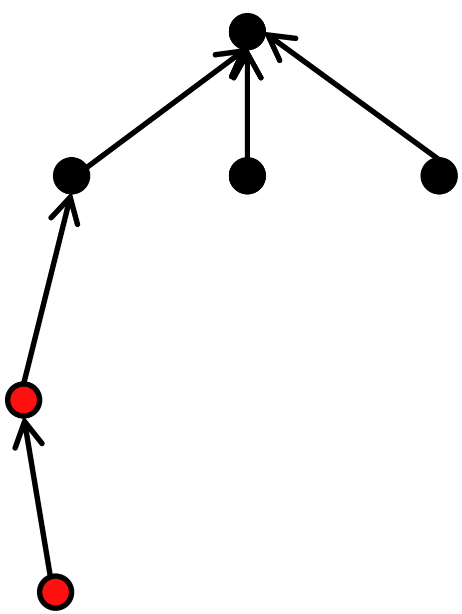


New: Create new node – O(1)

Rep: Follow pointers to root - O(depth)

Union Find Implementation

Basic Idea: Each set is a rooted tree with edges pointing towards the representative.



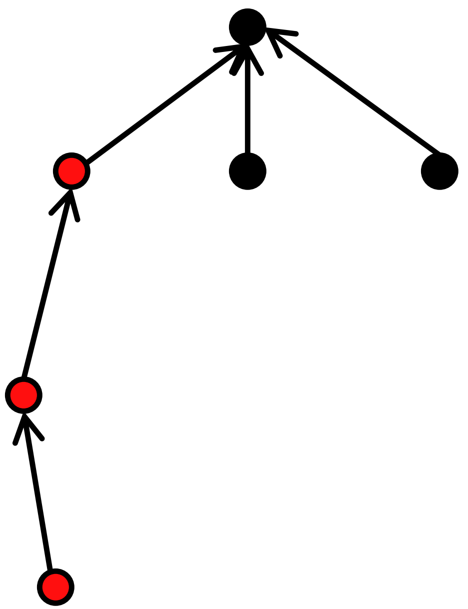
New: Create new node – O(1)

Rep: Follow pointers to root

- O(depth)

Union Find Implementation

Basic Idea: Each set is a rooted tree with edges pointing towards the representative.



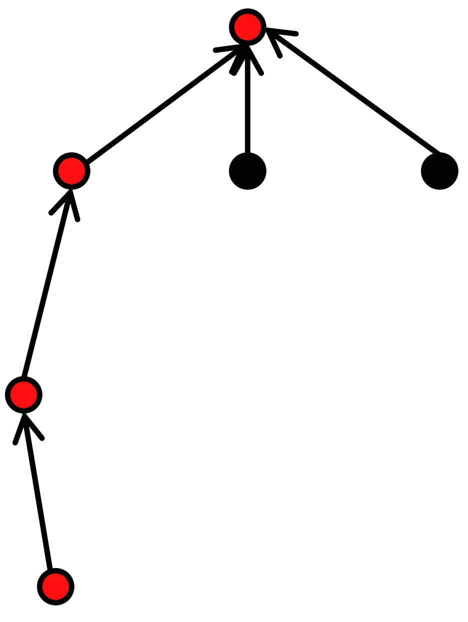
New: Create new node – O(1)

Rep: Follow pointers to root

- O(depth)

Union Find Implementation

Basic Idea: Each set is a rooted tree with edges pointing towards the representative.



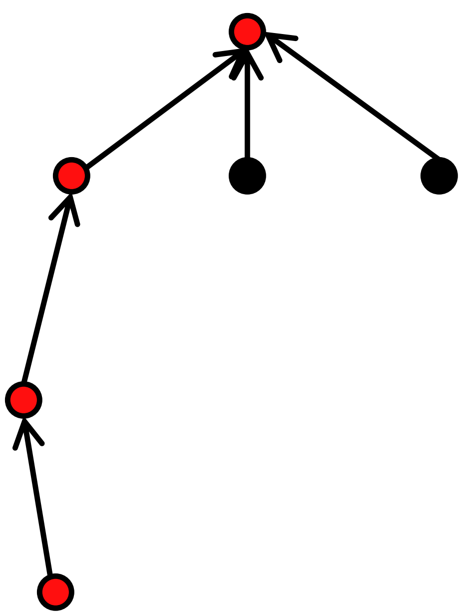
New: Create new node – O(1)

Rep: Follow pointers to root

- O(depth)

Union Find Implementation

Basic Idea: Each set is a rooted tree with edges pointing towards the representative.



New: Create new node – O(1)

Rep: Follow pointers to root

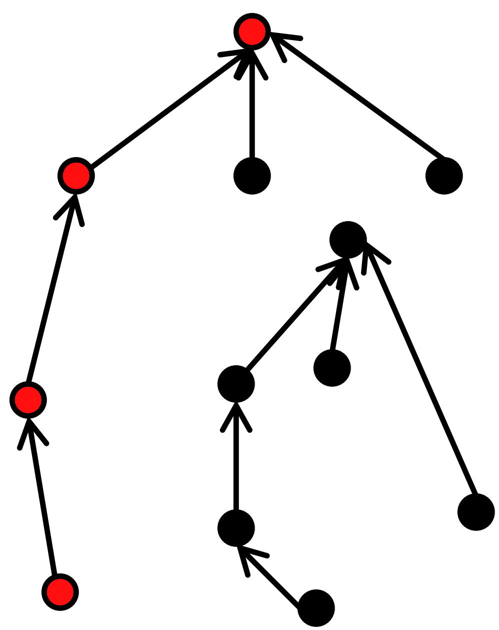
- O(depth)

Join: Have one Rep point to other.

- O(depth)

Union Find Implementation

Basic Idea: Each set is a rooted tree with edges pointing towards the representative.



New: Create new node – O(1)

Rep: Follow pointers to root

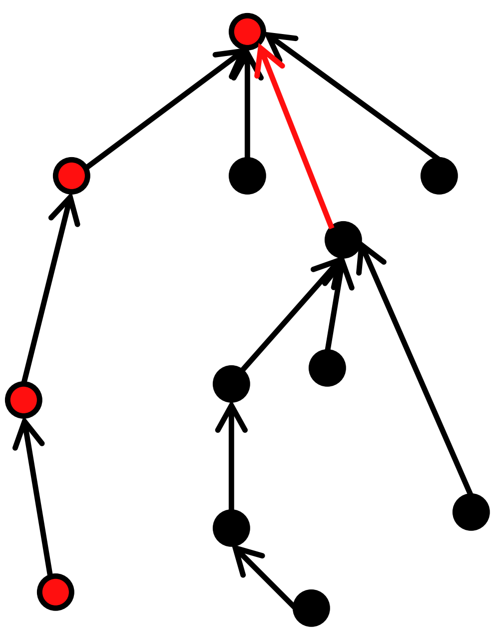
- O(depth)

Join: Have one Rep point to other.

- O(depth)

Union Find Implementation

Basic Idea: Each set is a rooted tree with edges pointing towards the representative.



New: Create new node – O(1)

Rep: Follow pointers to root

- O(depth)

Join: Have one Rep point to other.

- O(depth)

Depth

Need to ensure depth isn't too big.

Depth

Need to ensure depth isn't too big.

Idea: Always have shallower tree point to deeper one.

Depth

Need to ensure depth isn't too big.

Idea: Always have shallower tree point to deeper one.

Proposition: With the above rule any tree of depth n must have at least 2^n nodes.

Depth

Need to ensure depth isn't too big.

Idea: Always have shallower tree point to deeper one.

Proposition: With the above rule any tree of depth n must have at least 2^n nodes.

Proof: Induction on n. n = 0, done.
To get a tree of depth n, need to join two trees of depth n-1. Total of at least $2^{n-1}+2^{n-1} = 2^n$ nodes.

Runtime

Union-Find on n nodes runs operations in $O(\log(n))$ time.

Runtime

Union-Find on n nodes runs operations in $O(\log(n))$ time.

Kruskal runs in time $O(|E| \log|E|)$.

Runtime

Union-Find on n nodes runs operations in $O(\log(n))$ time.

Kruskal runs in time $O(|E| \log|E|)$.

Note: Using path compressions, union-find actually runs in $\alpha(n)$ time per operation.

Other Algorithms

There are many other ways to create MST algorithms. Kruskal searches the whole graph for light edges, but you can also grow from a point.

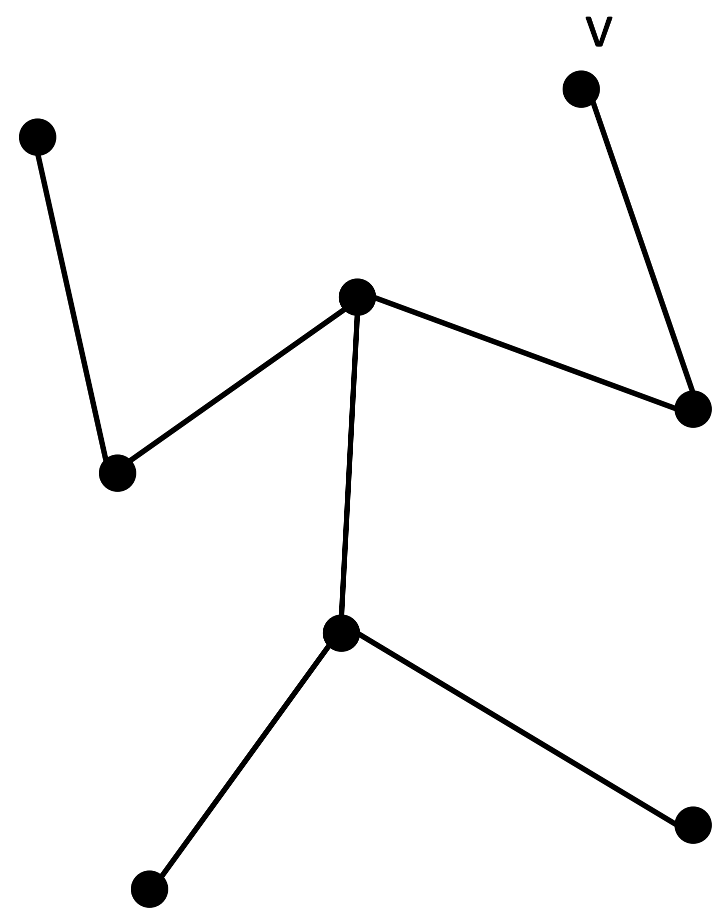
Other Algorithms

There are many other ways to create MST algorithms. Kruskal searches the whole graph for light edges, but you can also grow from a point.

Proposition: In a graph G, with vertex v, let e be an edge of lightest weight adjacent to v. Then there exists an MST of G containing e. Furthermore, if e is the unique lightest edge, then *all* MSTs contain e.

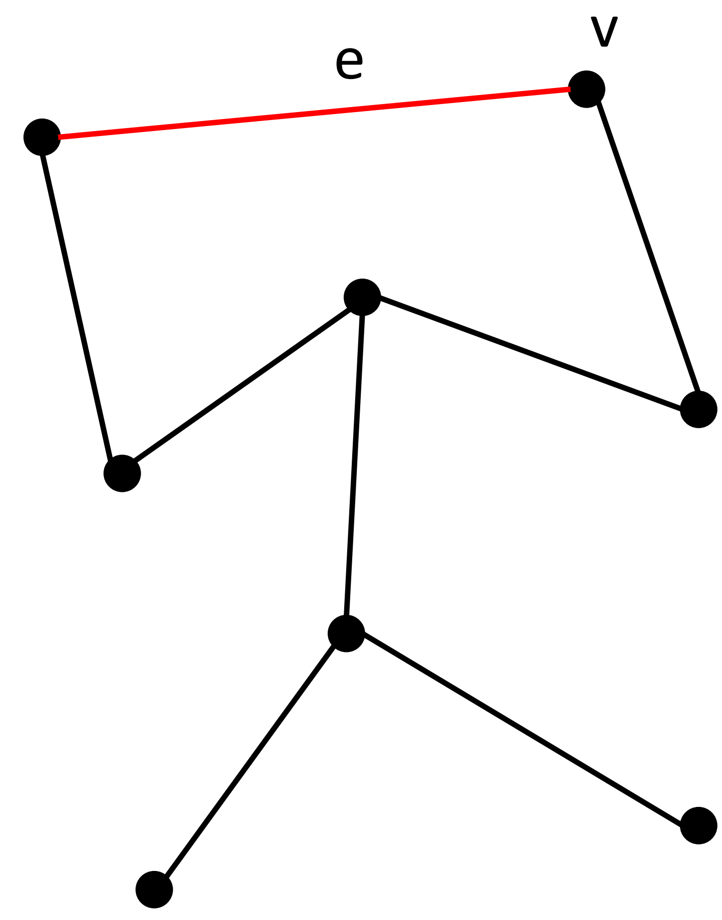
Proof

- Consider tree T not containing e.



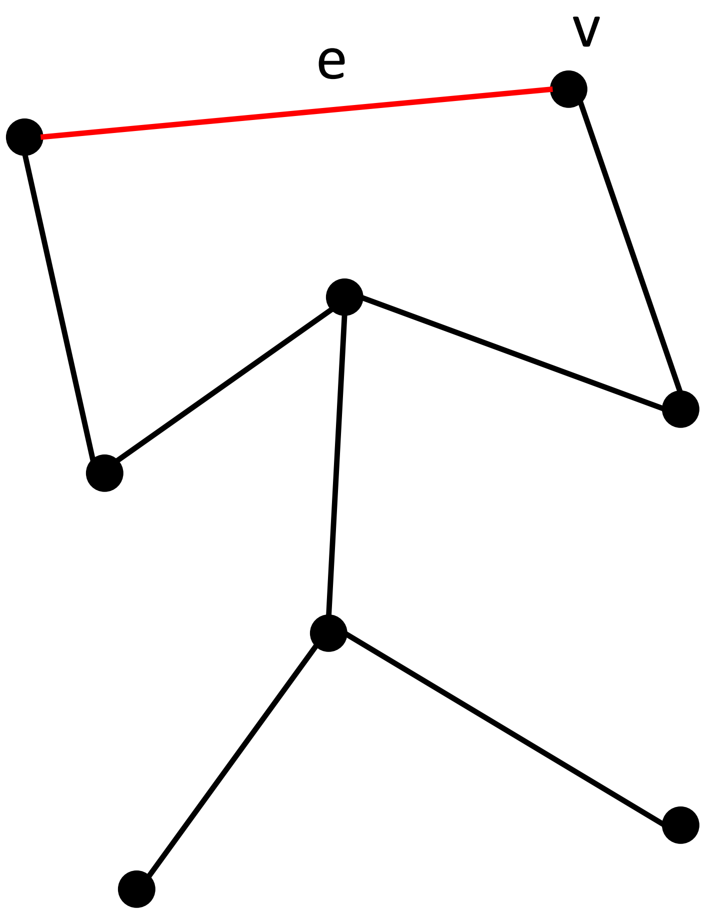
Proof

- Consider tree T not containing e.



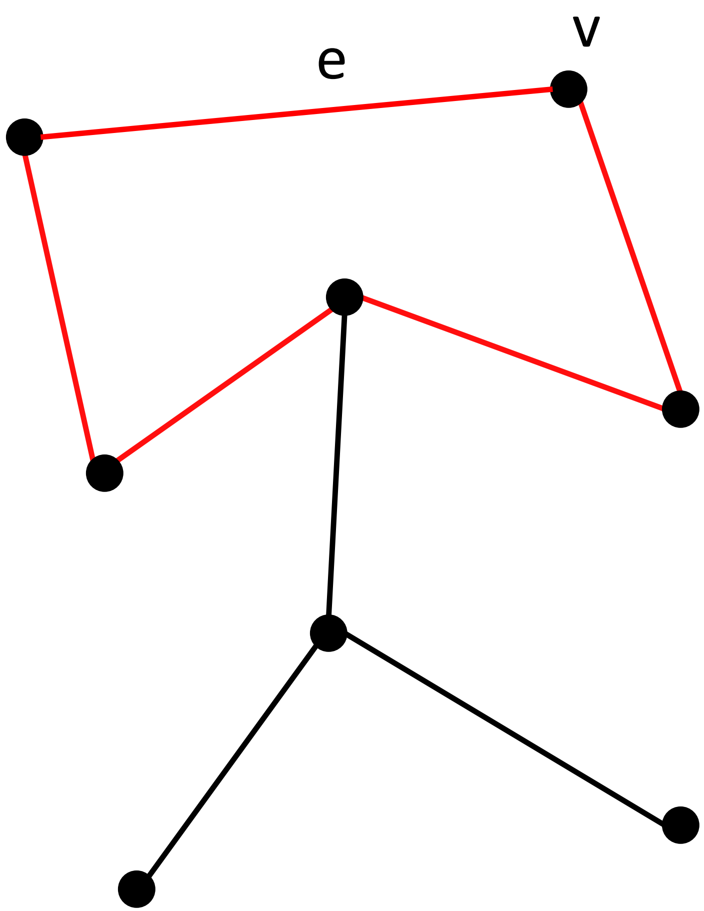
Proof

- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.



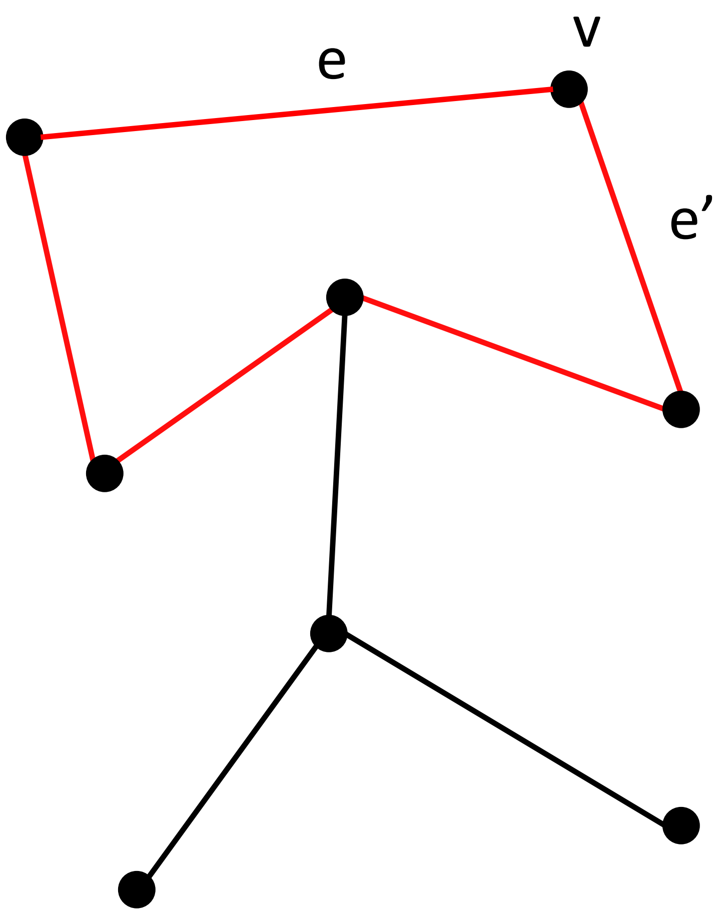
Proof

- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.



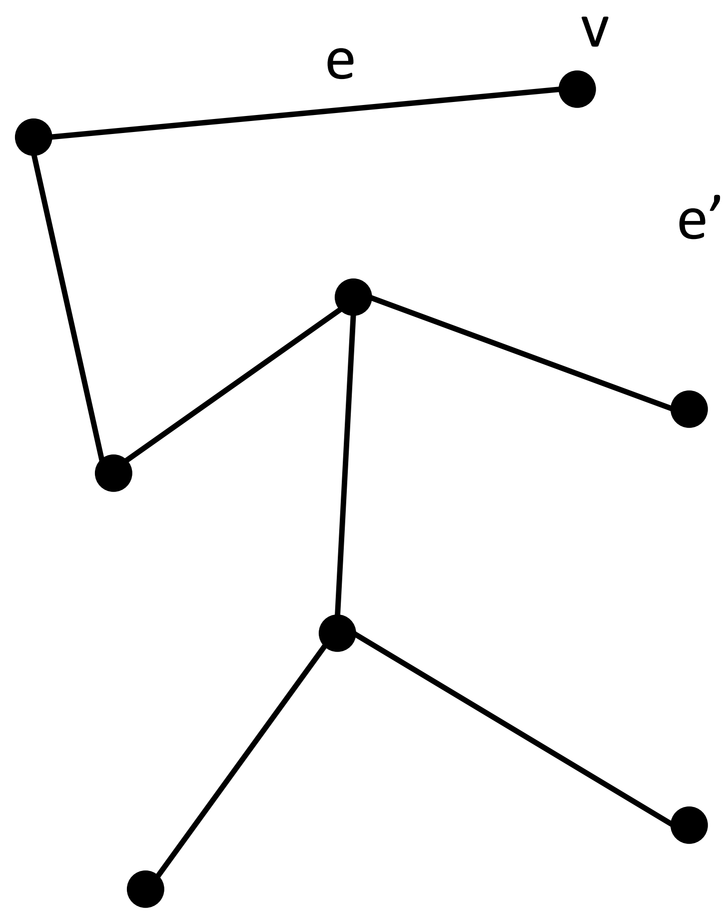
Proof

- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.
- Remove other edge e' adjacent to v from cycle to get T'.



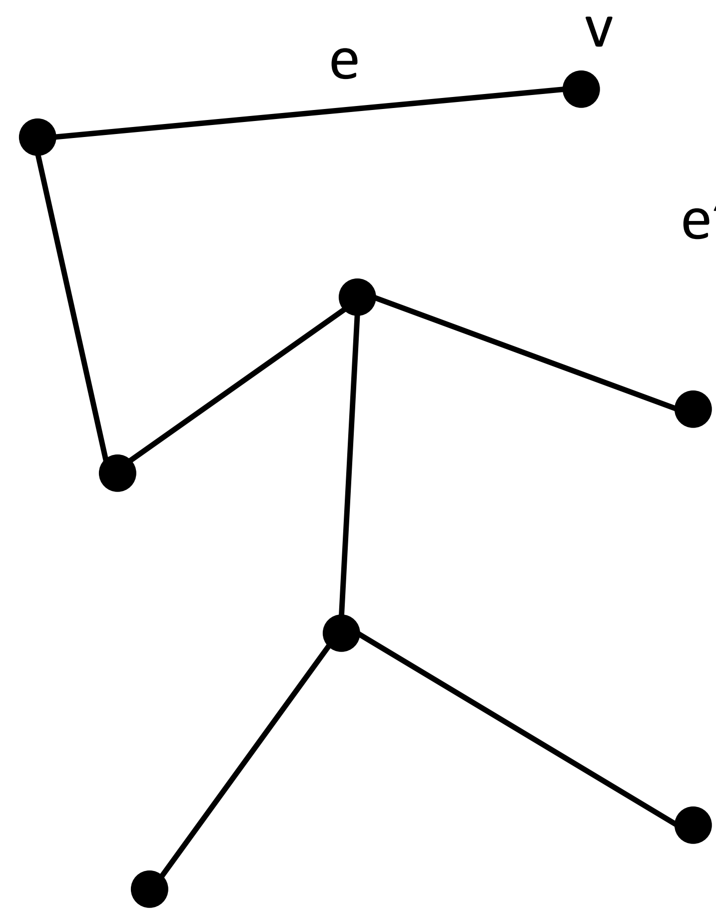
Proof

- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.
- Remove other edge e’ adjacent to v from cycle to get T’.



Proof

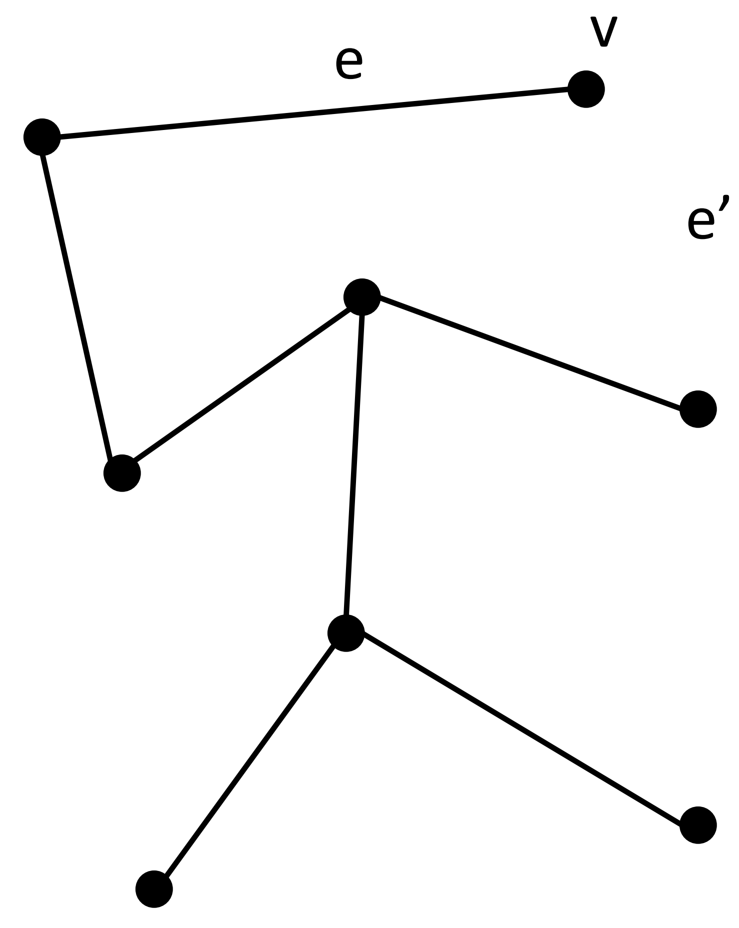
- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.
- Remove other edge e’ adjacent to v from cycle to get T’.
- |T’|=|V|-1, and connected, so T’ is a tree.



Proof

- Consider tree T not containing e.
- With extra edge, no longer a tree, must contain a cycle.
- Remove other edge e' adjacent to v from cycle to get T'.
- |T'|=|V|-1, and connected, so T' is a tree.
- wt(T') = wt(T)+wt(e)-wt(e')
  ≤ wt(T)

(because wt(e) is minimal).



Prim's Algorithm

So instead of checking *all* edges, you can just check edges from v.

Prim’s Algorithm

So instead of checking *all* edges, you can just check edges from v.

You can then contract edge and repeat.

Prim’s Algorithm

So instead of checking *all* edges, you can just check edges from v.

You can then contract edge and repeat.

Prim’s Algorithm: Add lightest edge that connects v to a new vertex.

Prim's Algorithm

So instead of checking *all* edges, you can just check edges from v.

You can then contract edge and repeat.

<u>**Prim's Algorithm:**</u> Add lightest edge that connects v to a new vertex.

Implementation very similar to Dijkstra.

Prim’s Algorithm

```
Prim(G,w)
  Pick vertex s                 \\ doesn’t matter which
  For v ∈ V, b(v) ← ∞           \\ lightest edge into v
  T ← {}, b(s) ← 0
  Priority Queue Q, add all v with key=b(v)
  While(Q not empty)
    u ← DeleteMin(Q)
    If u ≠ s, add (u,Prev(u)) to T
    For (u,v) ∈ E
      If w(u,v) < b(v)
        b(v) ← w(u,v)
        Prev(v) ← u
        DecreaseKey(v)
  Return T
```

Prim's Algorithm

```
Prim(G,w)
  Pick vertex s                  \\ doesn't matter which
  For v ∈ V, b(v) ← ∞            \\ lightest edge into v
  T ← {}, b(s) ← 0
  Priority Queue Q, add all v with key=b(v)
  While(Q not empty)
    u ← DeleteMin(Q)
    If u ≠ s, add (u,Prev(u)) to T
    For (u,v) ∈ E
      If w(u,v) < b(v)
        b(v) ← w(u,v)
        Prev(v) ← u
        DecreaseKey(v)
  Return T
```

Runtime:

$O(|V|\log|V| + |E|)$

Slightly better than Kruskal

Analysis

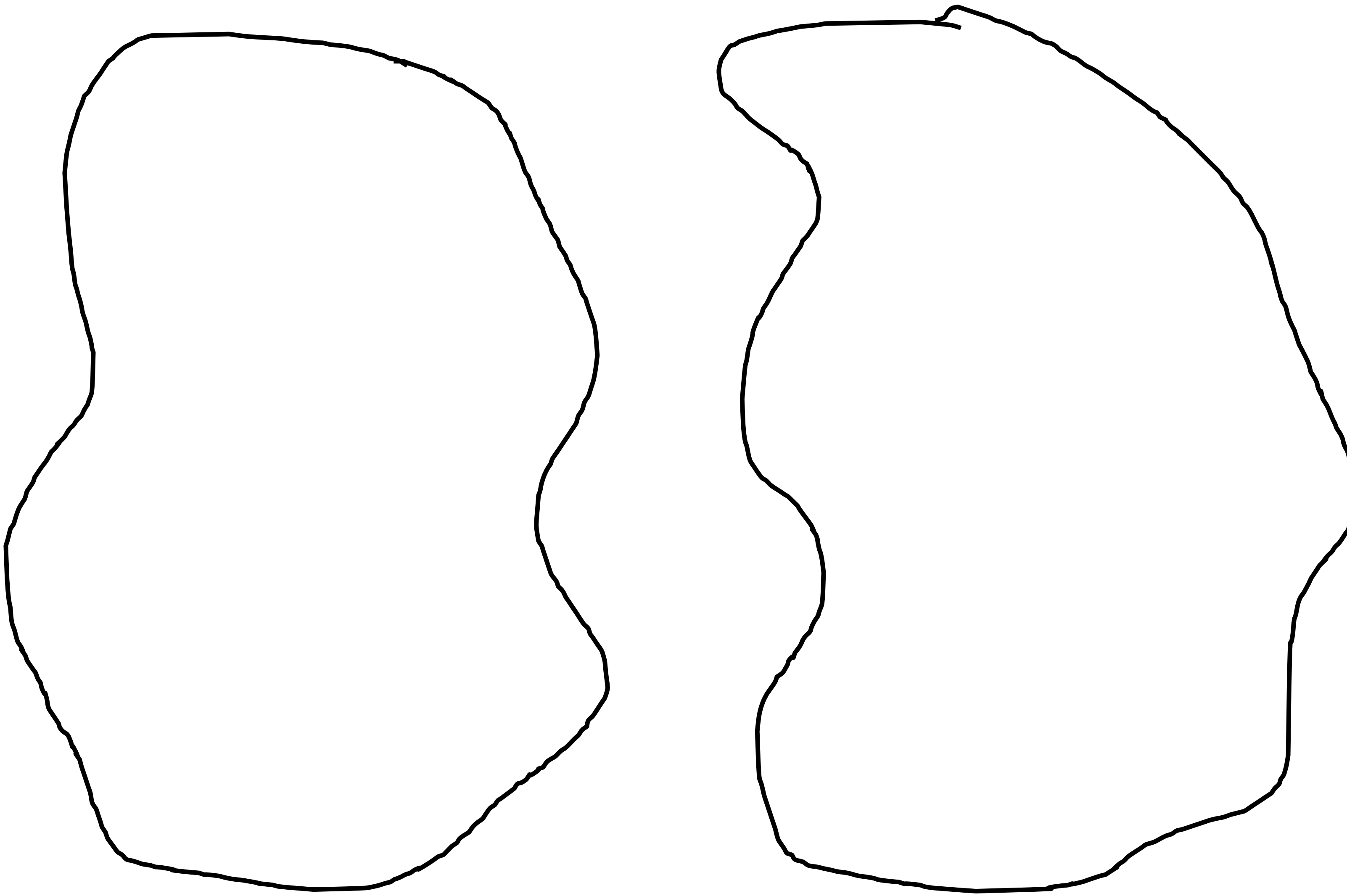
At any stage, have some set S of vertices connected to s. Find cheapest edge connecting S to S^C.

Analysis

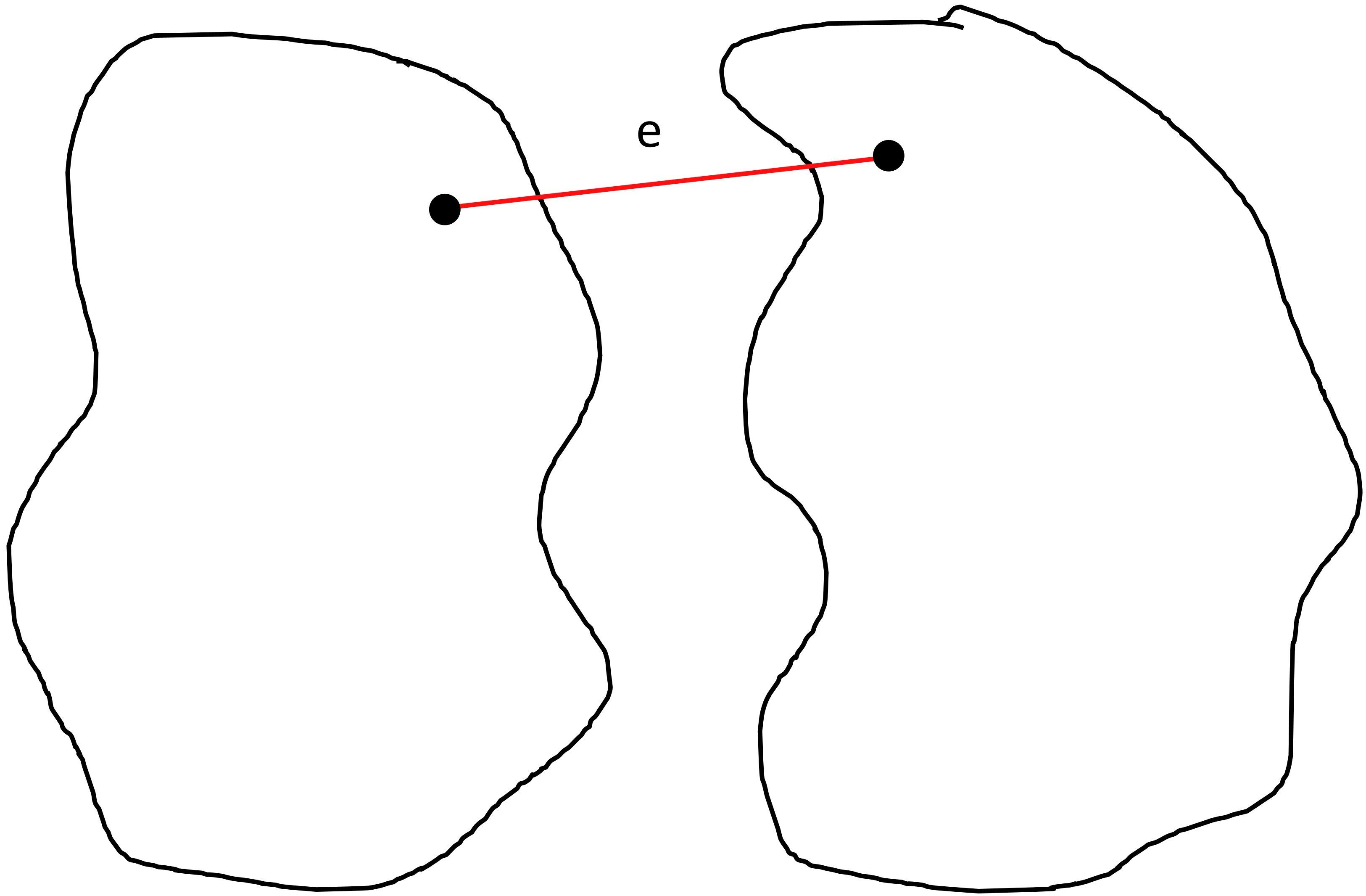
At any stage, have some set S of vertices connected to s. Find cheapest edge connecting S to S^C.

Proposition: In a graph G, with a cut C, let e be an edge of lightest weight crossing C. Then there exists an MST of G containing e. Furthermore, if e is the unique lightest edge, then *all* MSTs contain e.

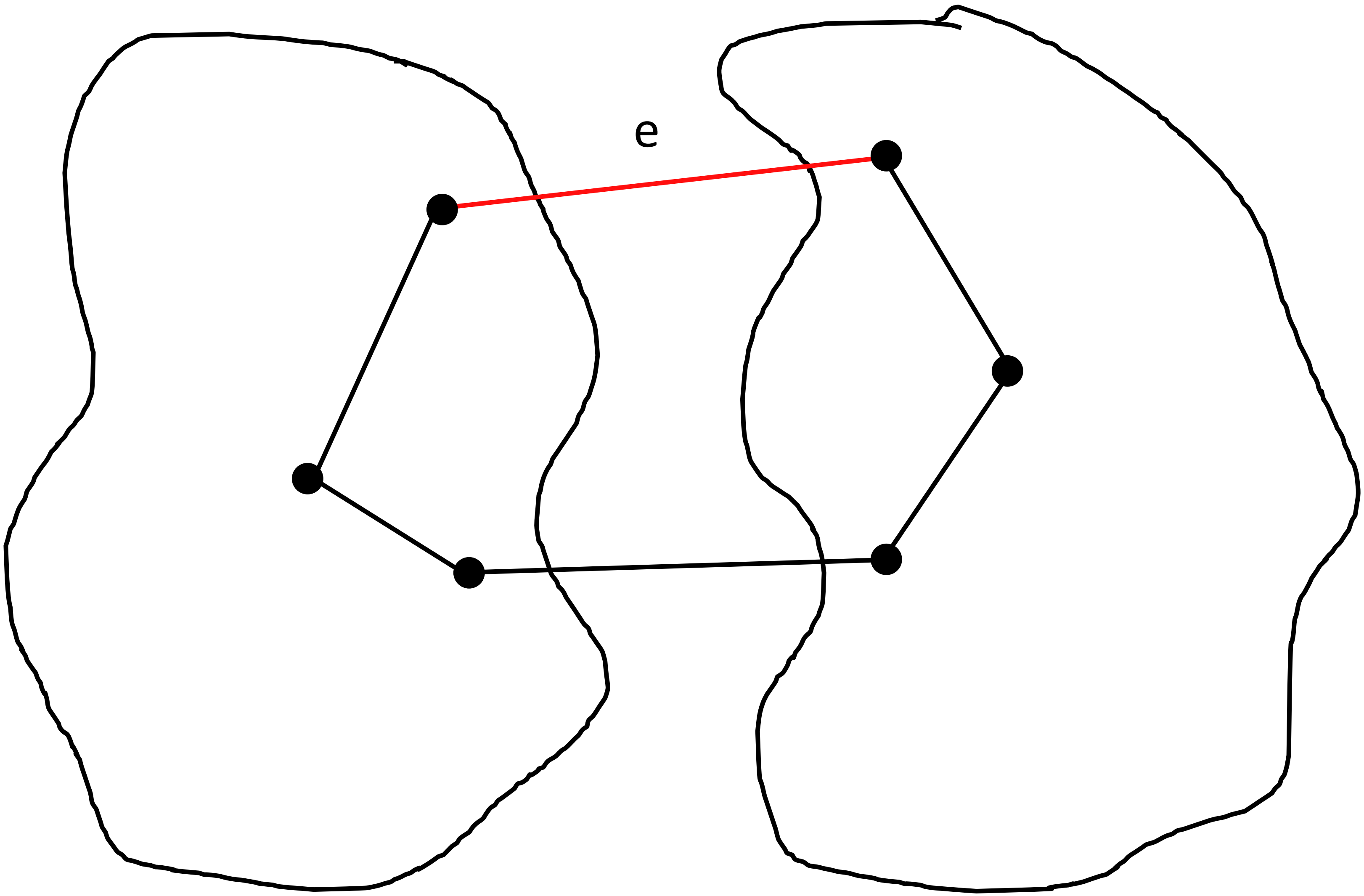
Proof



Proof

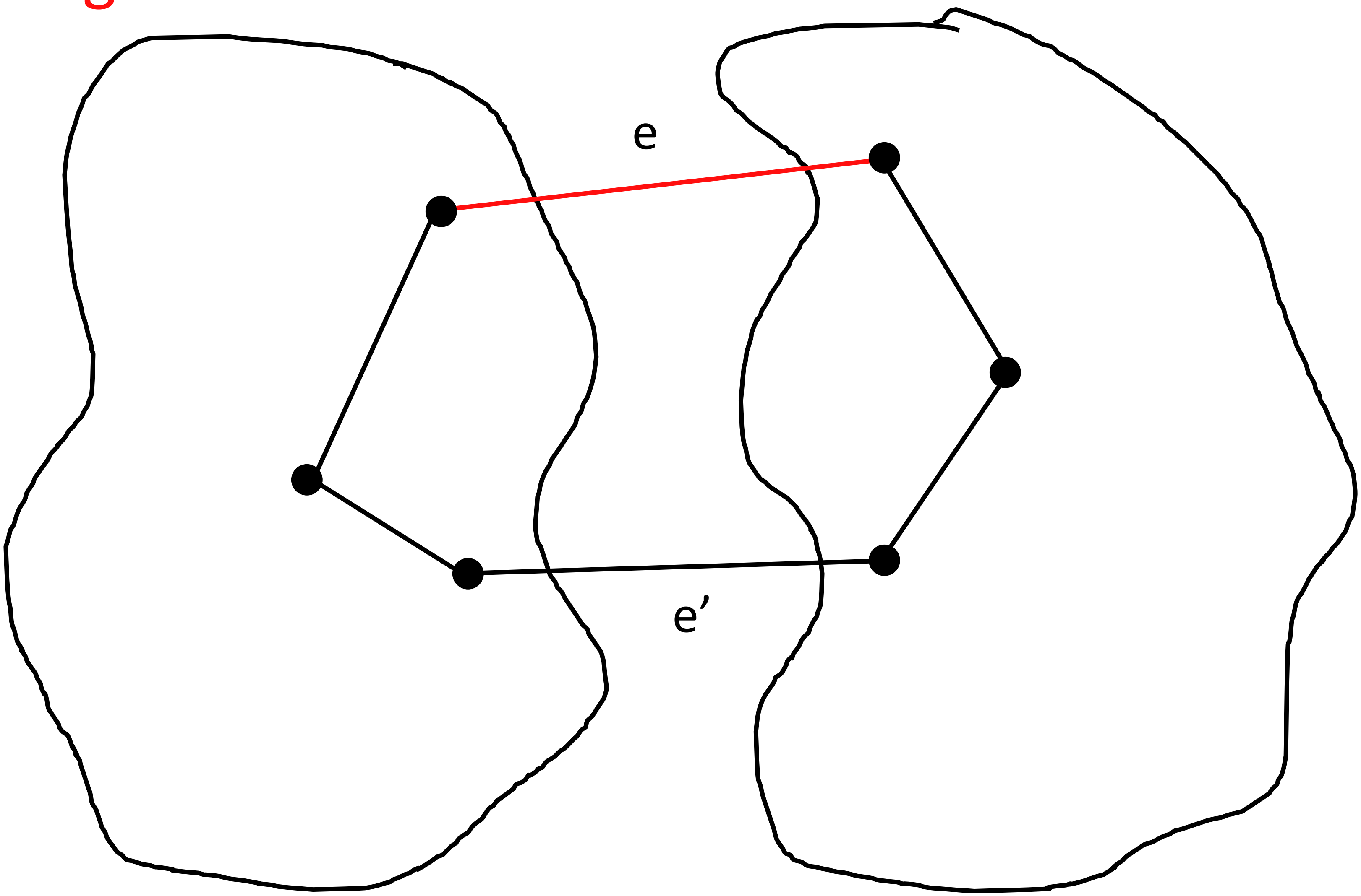


Proof



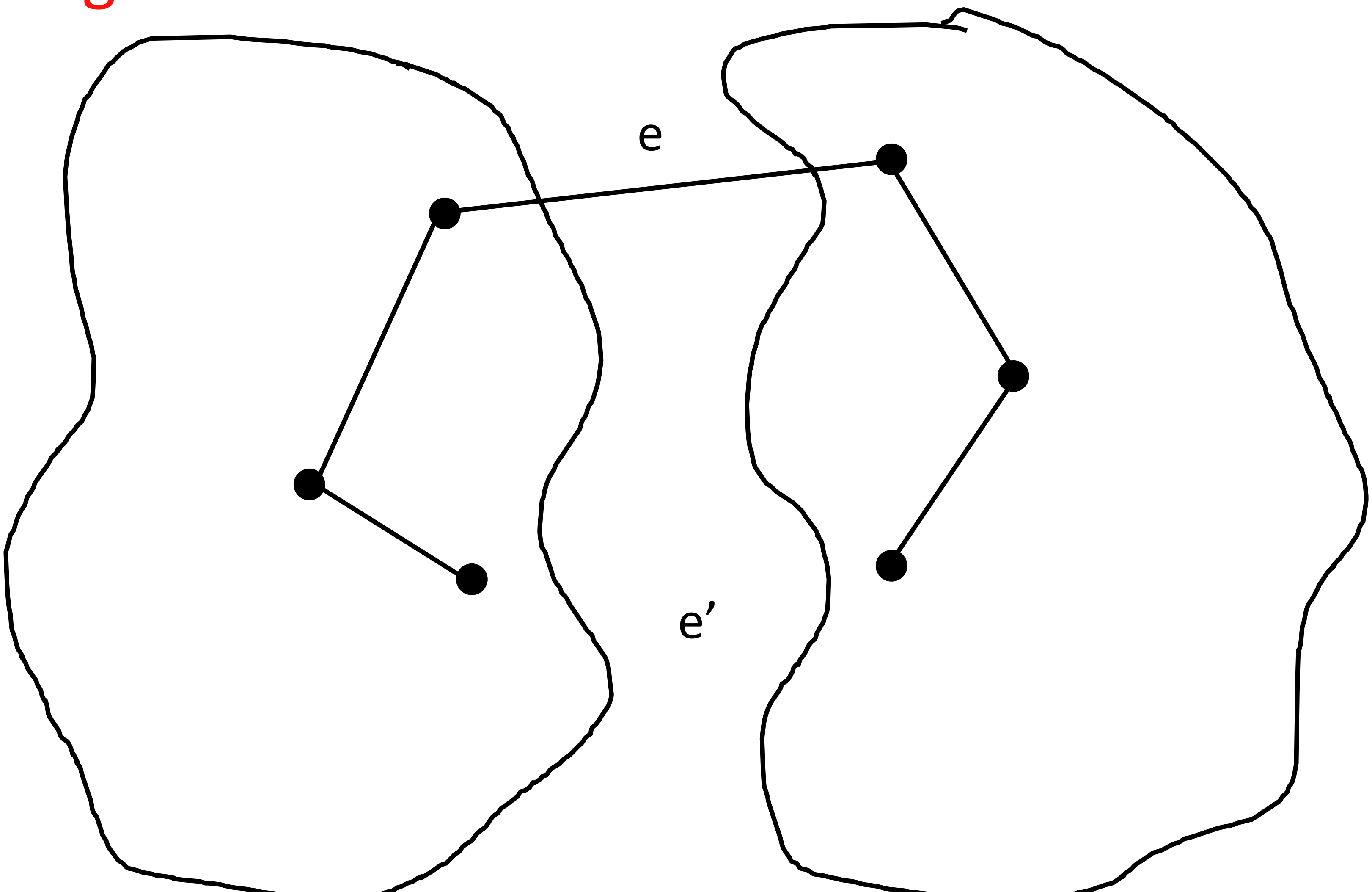
Cycle must have e' crossing cut!

Proof



Cycle must have e'
crossing cut!

Proof



Notes on MST

Minimum Spanning Tree is one of the best-studied algorithmic problems and there are many known algorithms.

Notes on MST

Minimum Spanning Tree is one of the best-studied algorithmic problems and there are many known algorithms.

- Randomized $O(|V|+|E|)$ by [Karger-Klein-Tarjan]

Notes on MST

Minimum Spanning Tree is one of the best-studied algorithmic problems and there are many known algorithms.

- Randomized $O(|V|+|E|)$ by [Karger-Klein-Tarjan]
- $O(|E|\ \alpha(|E|))$ by Chazelle

Notes on MST

Minimum Spanning Tree is one of the best-studied algorithmic problems and there are many known algorithms.

- Randomized $O(|V|+|E|)$ by [Karger-Klein-Tarjan]
- $O(|E|\ \alpha(|E|))$ by Chazelle
- Best algorithm known (not known whether it is $O(|V|+|E|)$)