Algorithms at Scale

(Week 5)

Puzzle of the Day:

Five prisoners are each given a black or white hat, each with probability 1/2.

- Can see others hats. Cannot see own hat.
- All guess simultaneously: “black” or “white” or “NO GUESS”
- WIN if at least *one* correct guess, no incorrect guess.

What is their best strategy? (Hint: solve 3 prisoners first!)

Summary

Last Week: Streaming

Misra-Gries:

- Item frequency
- Heavy Hitters

Flajolet-Martin:

- Number of distinct elements
- Median-of-means technique
- Chebychev+Chernoff

Problem Set:

- Alternate solution to item frequency and heavy hitters.

Today: Graph Streaming

Connectivity

- Is the graph connected?

Bipartite

- Is the graph bipartite?

MST

- Find a minimum spanning tree

Spanners

- Find approximate shortest paths

Matching

- Find an (approximate) maximum matching.

Streaming a Graph

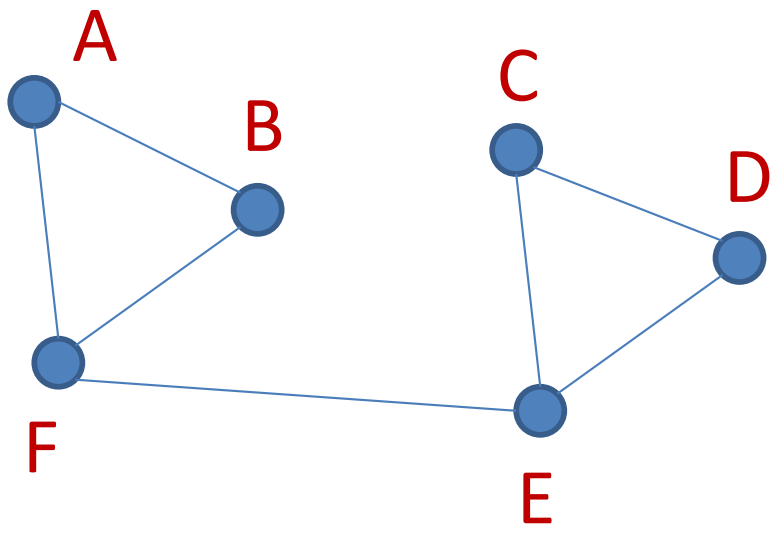
Data arrives in a stream: $S = s_1, s_2, \ldots, s_T$

Each s_j is an edge in the graph.

- ⇒ Each edge shows up exactly once.
- ⇒ Edges show up in an arbitrary (worst-case) order.

Example:

S = (A,B), (C,D), (F,E), (C,E), (E,D), (A,F), (B,F)



Beware alternatives:

- ⇒ Edges may be repeated.
- ⇒ Edges may be *added* and *deleted*.
- ⇒ Edges are a random permutation.

Streaming a Graph

Data arrives in a stream: $S = s_1, s_2, \ldots, s_T$

Each s_j is an edge in the graph.

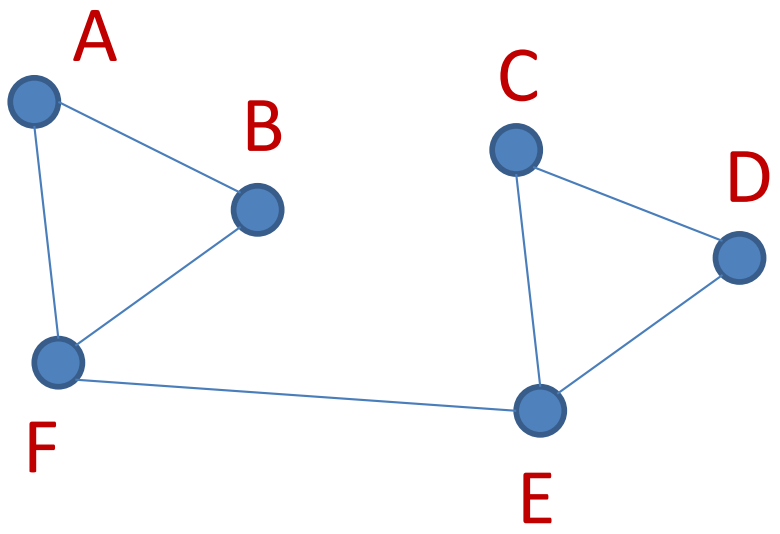
- ⇒ Each edge shows up exactly once.
- ⇒ Edges show up in an arbitrary (worst-case) order.

Example:

S = (A,B), (C,D), (F,E), (C,E), (E,D), (A,F), (B,F)

Goal: minimize space

- ⇒ Sublinear space is often impossible.
- ⇒ Best possible: O(n log n) space.
- ⇒ Focus on dense graphs.



Connected Components

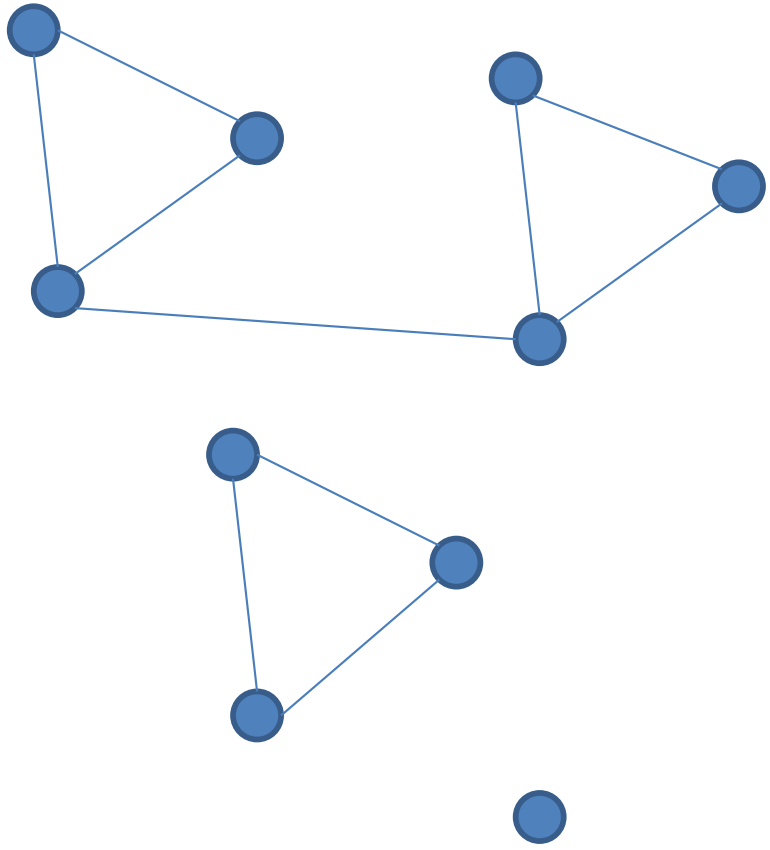
Assumptions:

Graph G = (V,E)

- Undirected
- n nodes
- m edges
- Goal: O(n log n) space.

Output:

Number of connected components.

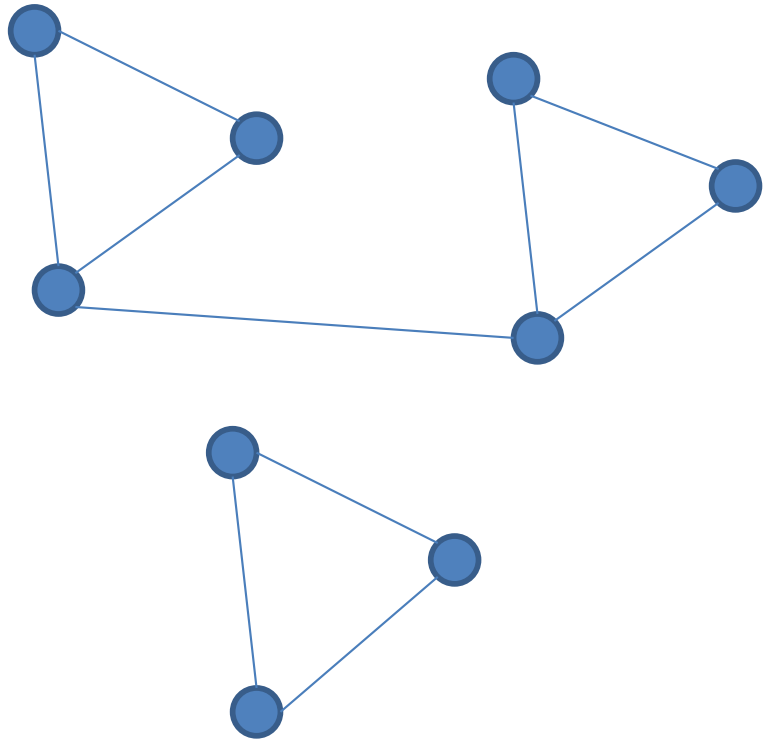


Example: output 3

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n

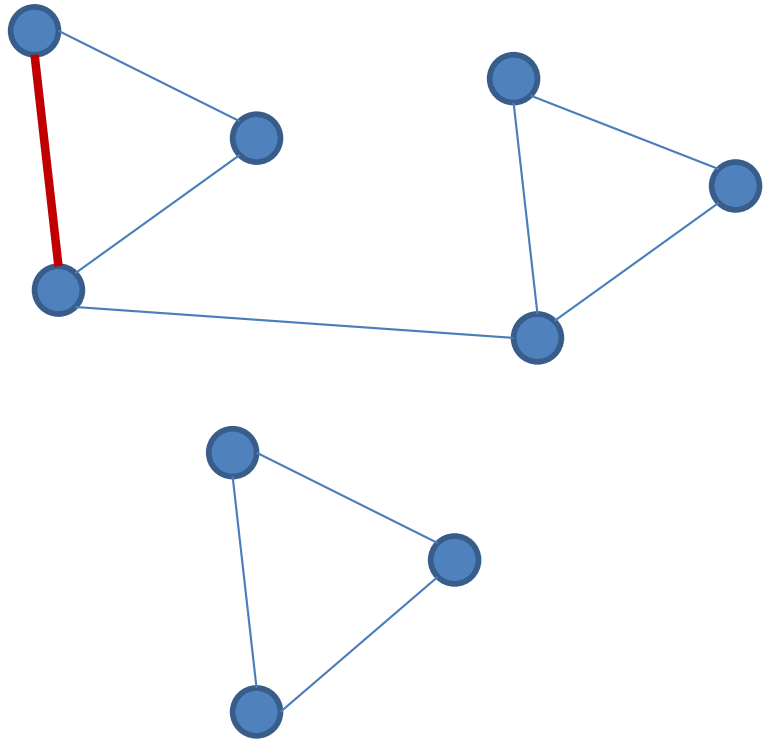


Maintain spanning forest of the graph.

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n

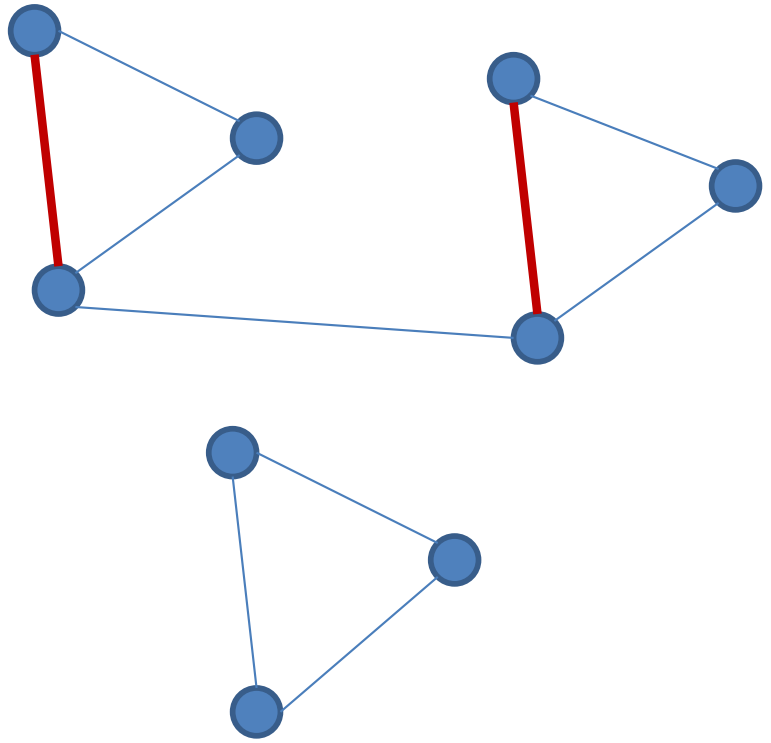


Maintain spanning forest of the graph.

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n

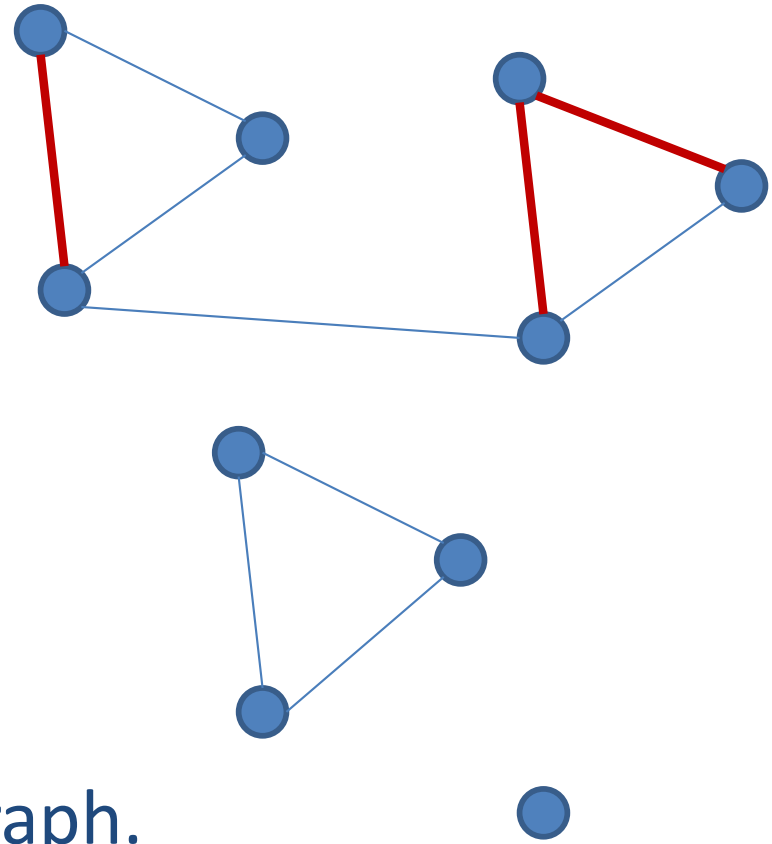


Maintain spanning forest of the graph.

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F $\cup$ e has no cycles then
        add e to F
n = # of components in F.
return n

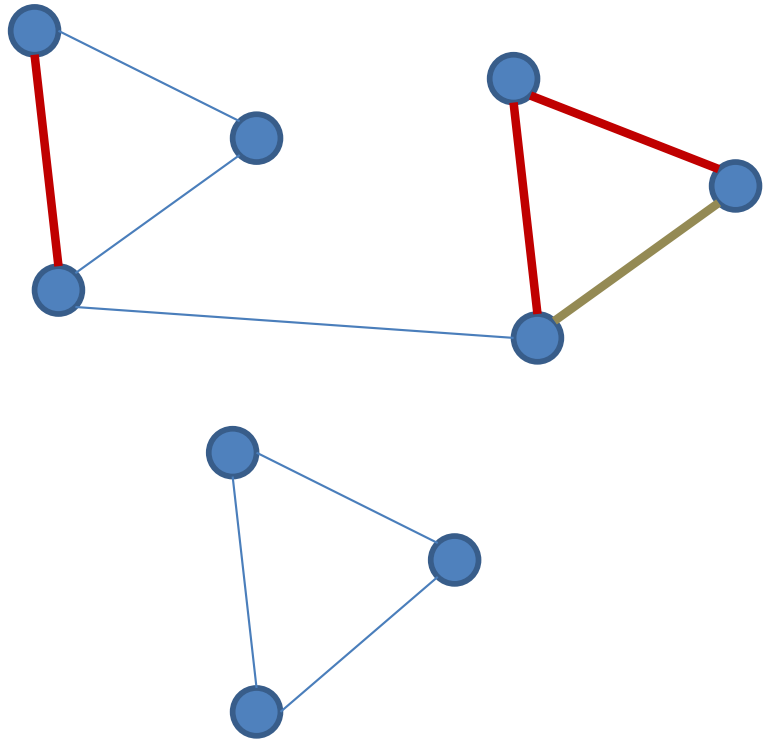


Maintain spanning forest of the graph.

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n

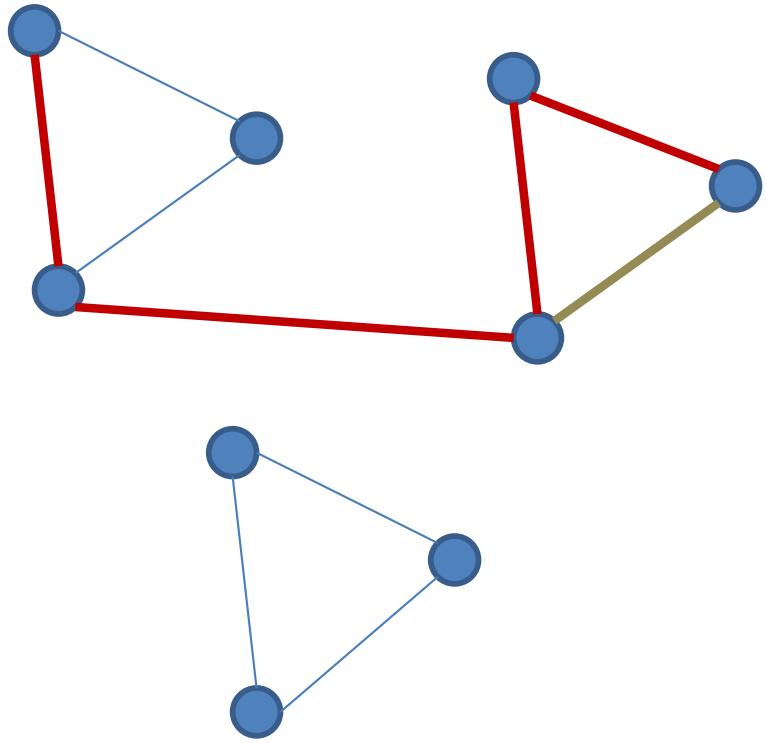


Maintain spanning forest of the graph.

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n

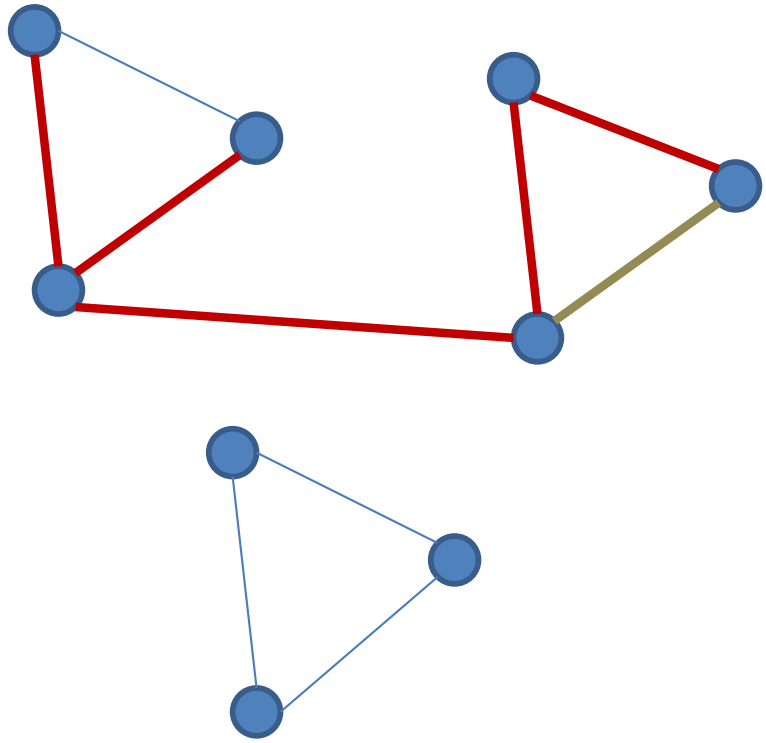


Maintain spanning forest of the graph.

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n

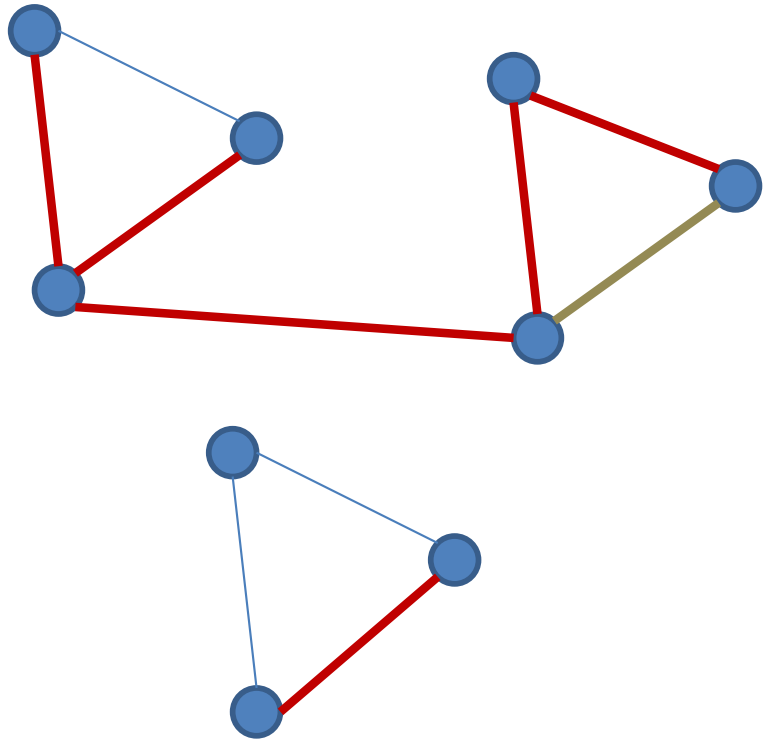


Maintain spanning forest of the graph.

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n

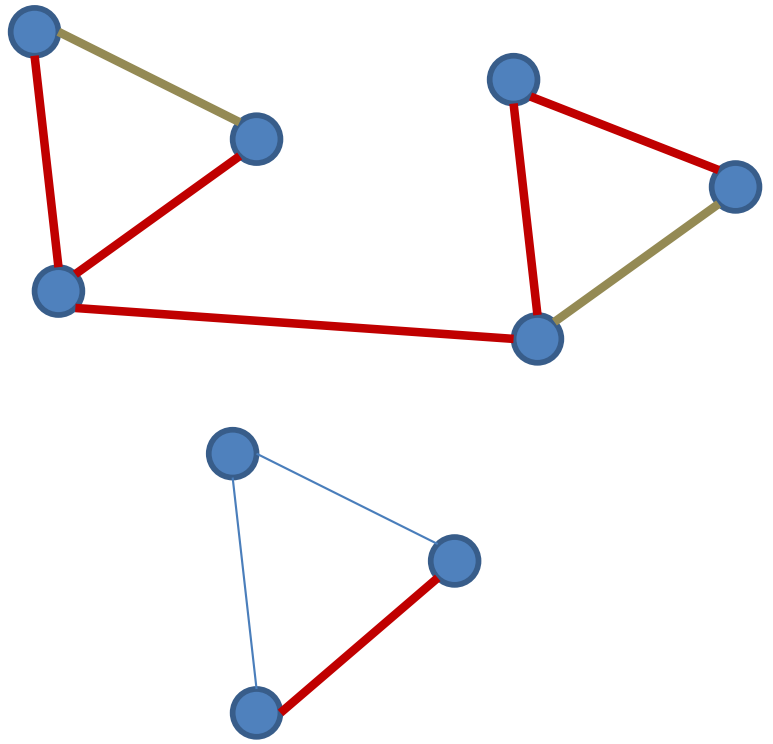


Maintain spanning forest of the graph.

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n

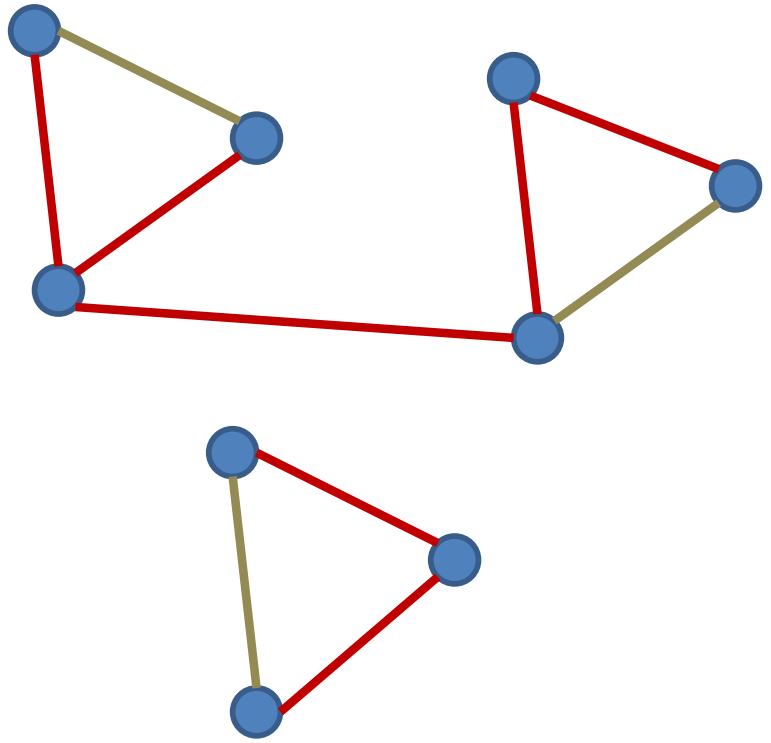


Maintain spanning forest of the graph.

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n

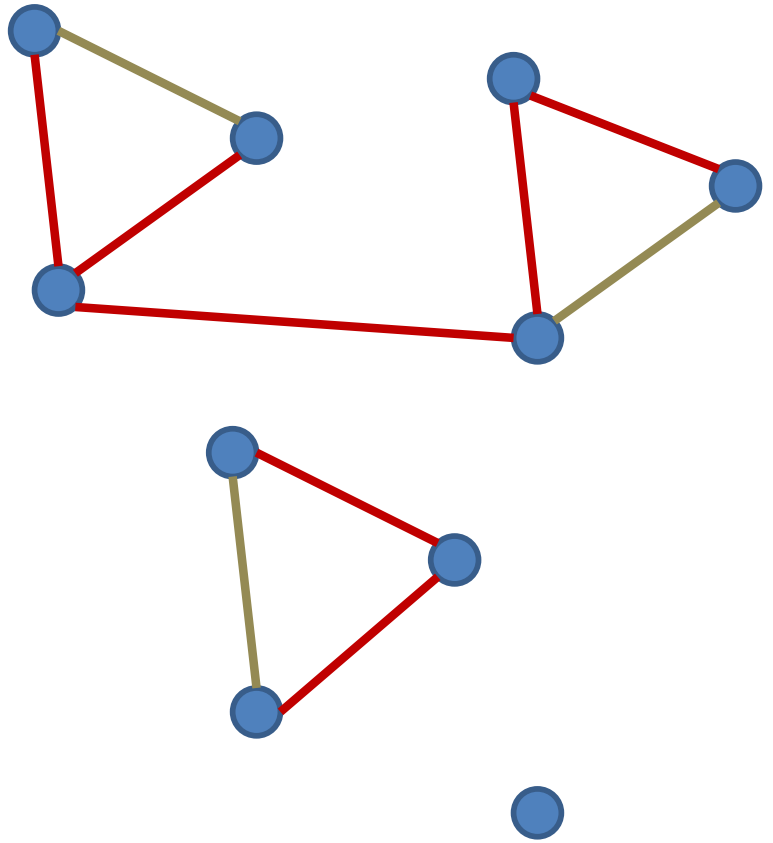


Maintain spanning forest of the graph.

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n

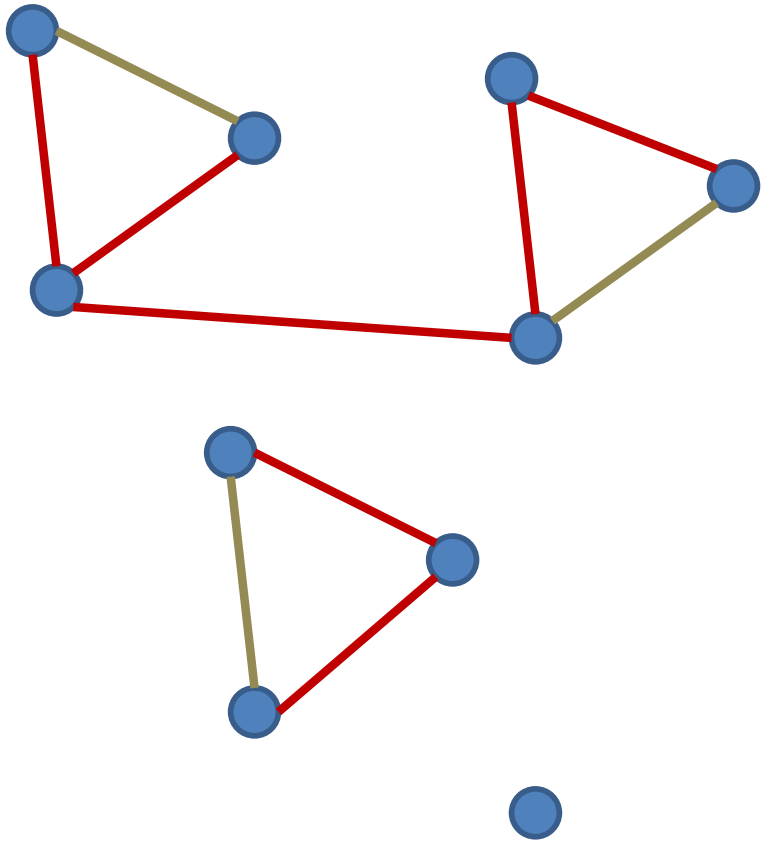


Proof: obvious.

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n

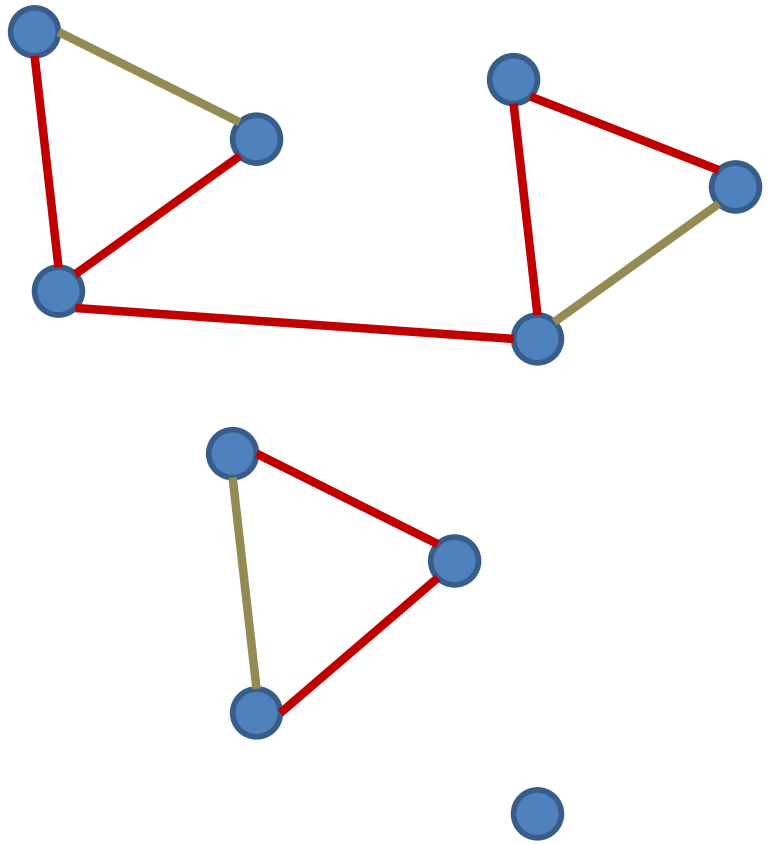


Space:

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n

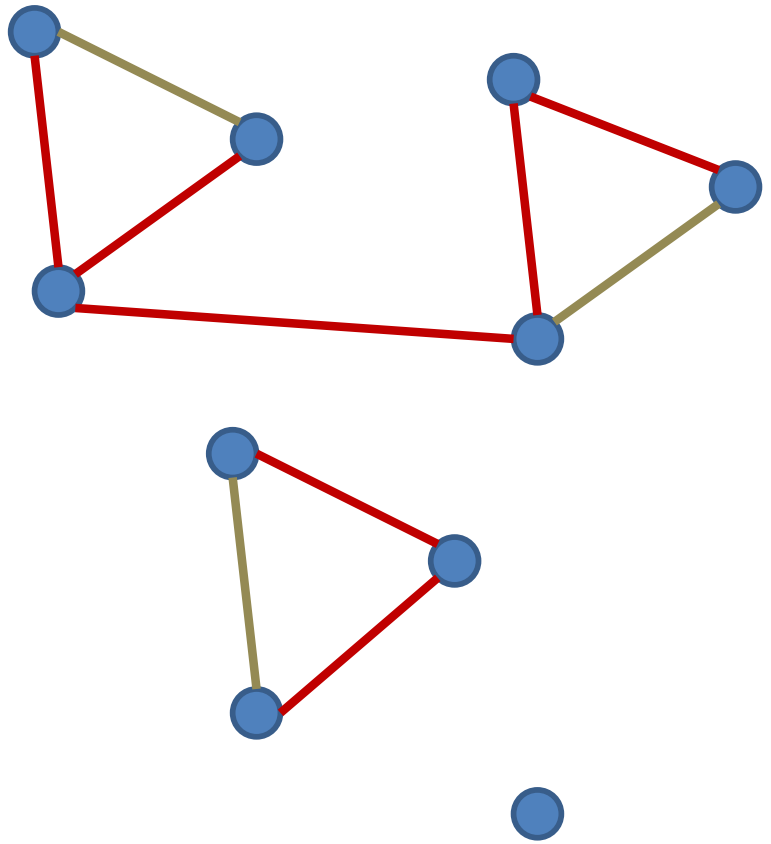


Space: O(n log n)

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n



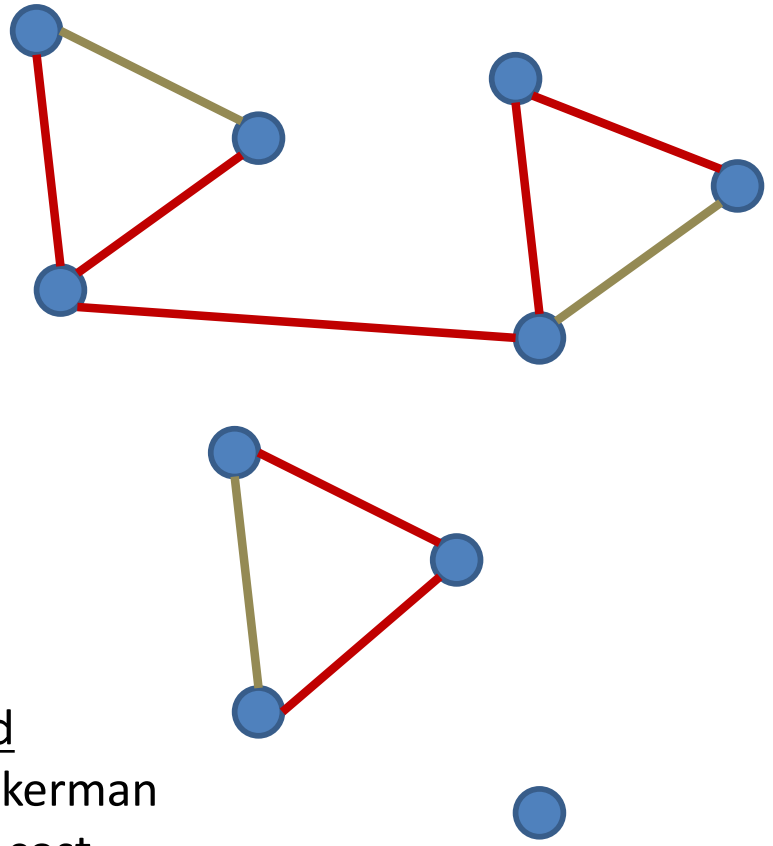
Space: O(n log n)

Update cost:

Connected Components

Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F
n = # of components in F.
return n



Space: O(n log n)

Update cost: O(α(n, n))

Union-Find
Inverse-Ackerman
amortized cost.

Is graph bipartite?

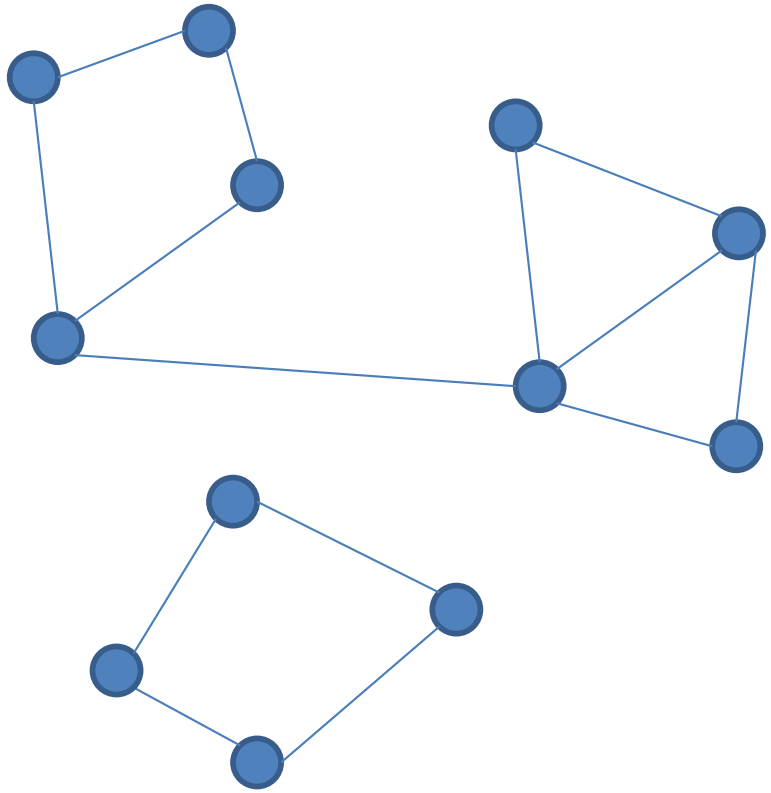
Assumptions:

Graph G = (V,E)

- Undirected
- n nodes
- m edges
- Goal: O(n log n) space.

Output:

Is the graph bipartite?



Example: NO

Is graph bipartite?

Assumptions:

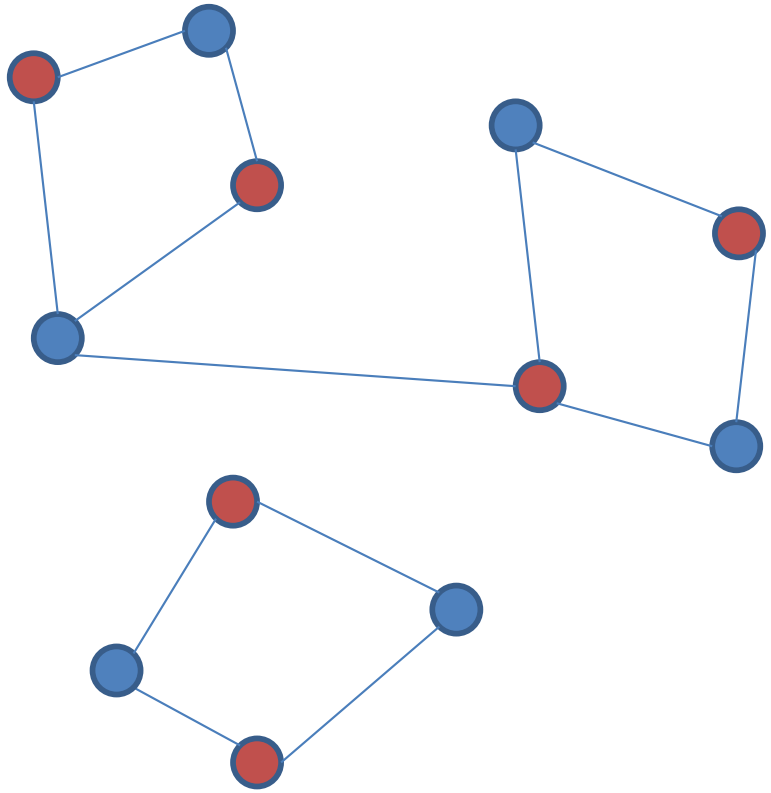
Graph G = (V,E)

- Undirected
- n nodes
- m edges
- Goal: O(n log n) space.

Output:

Is the graph bipartite?

Can the graph be 2-colored?



Example: YES

Is graph bipartite?

Assumptions:

Graph G = (V,E)

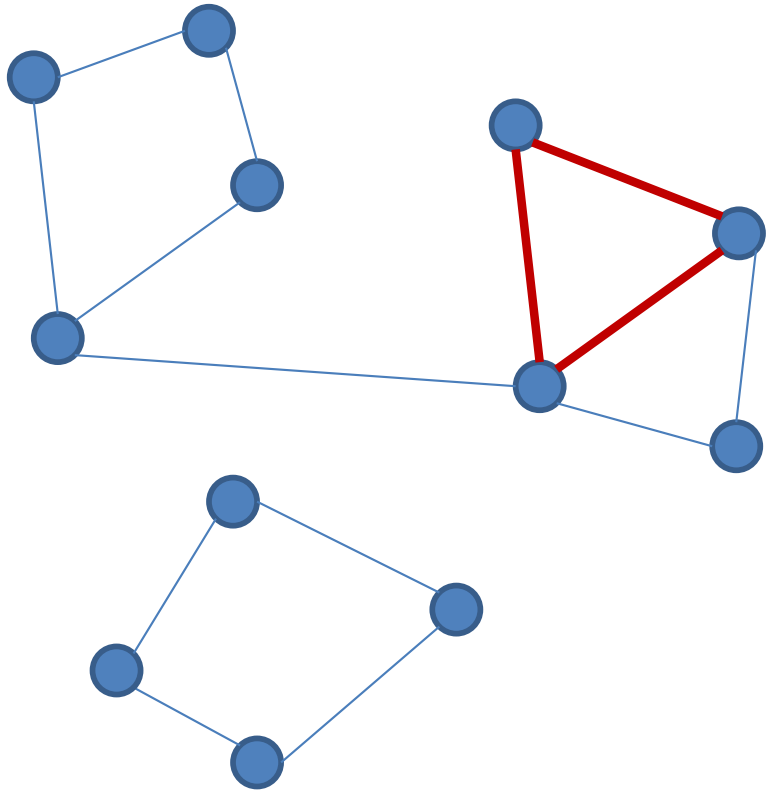
- Undirected
- n nodes
- m edges
- Goal: O(n log n) space.

Output:

Is the graph bipartite?

Can the graph be 2-colored?

Does the graph have no odd-length cycles?

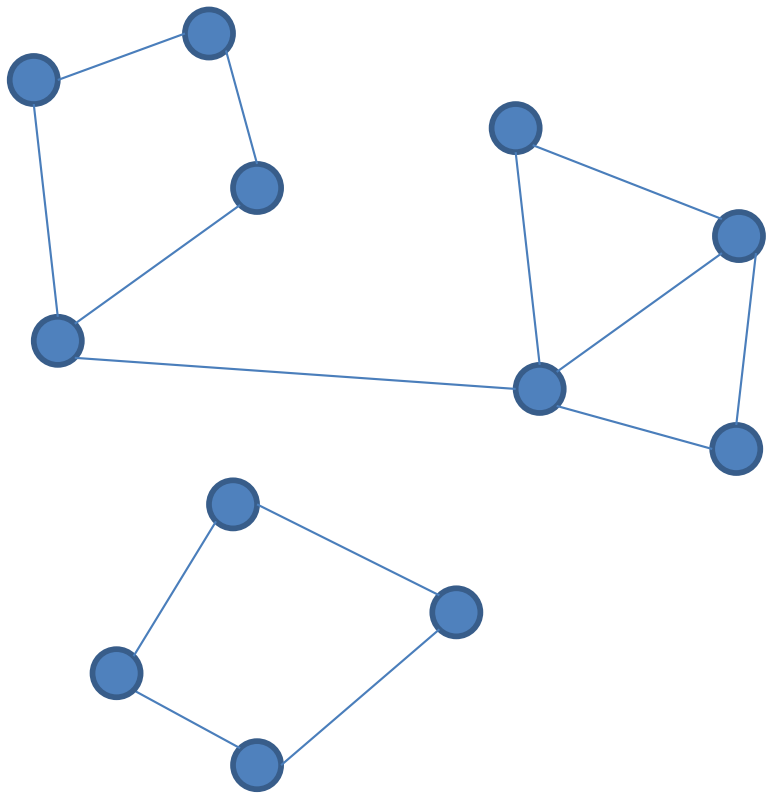


Example: NO

Is graph bipartite?

Bipartite Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F.
    if F ∪ e has odd cycle then
        return NO.
return YES

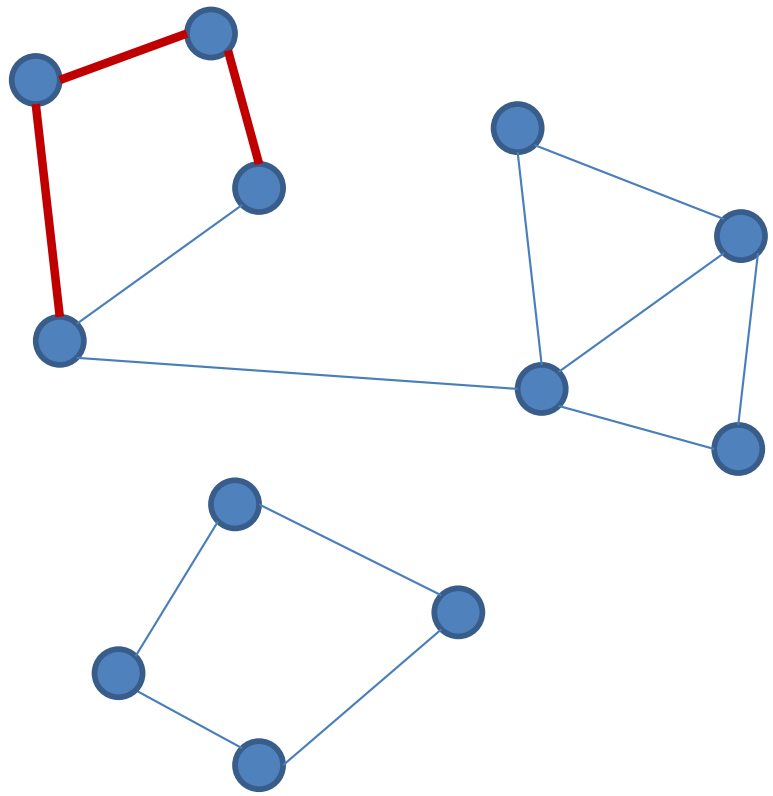


Maintain spanning forest of the graph.

Is graph bipartite?

Bipartite Spanning Forest

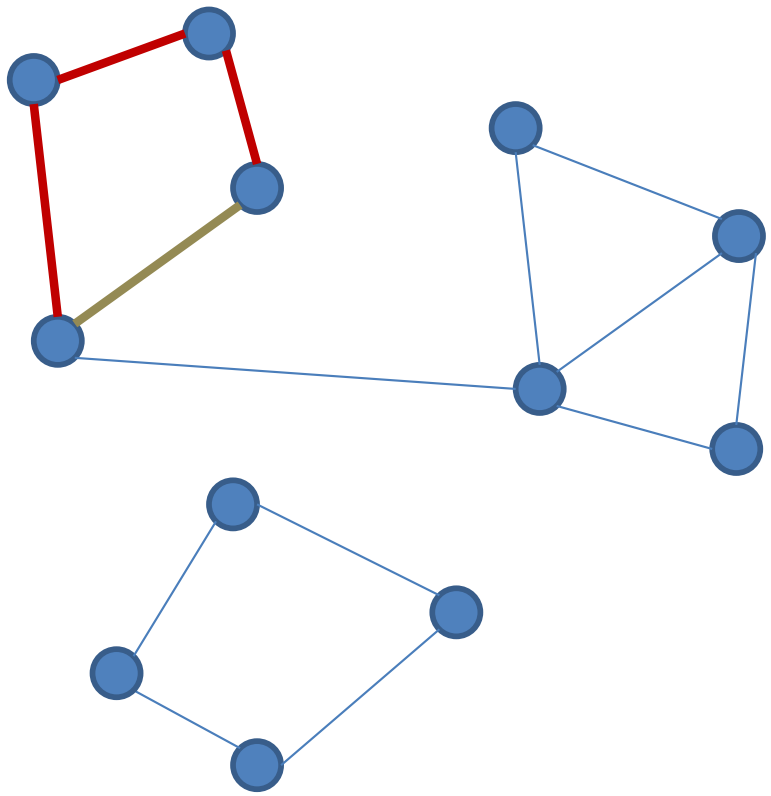
F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F.
    if F ∪ e has odd cycle then
        return NO.
return YES



Is graph bipartite?

Bipartite Spanning Forest

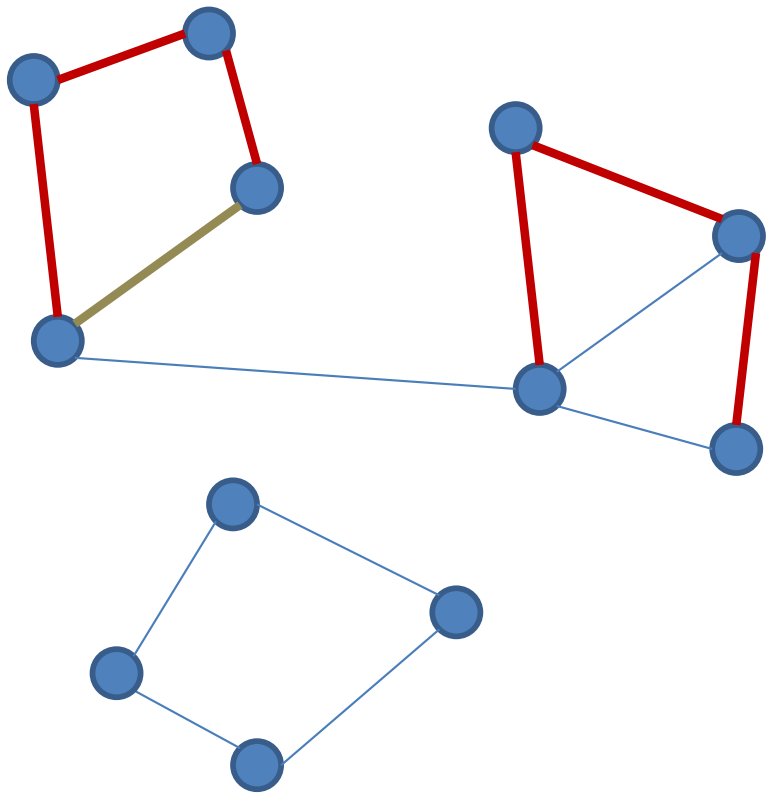
F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F.
    if F ∪ e has odd cycle then
        return NO.
return YES



Is graph bipartite?

Bipartite Spanning Forest

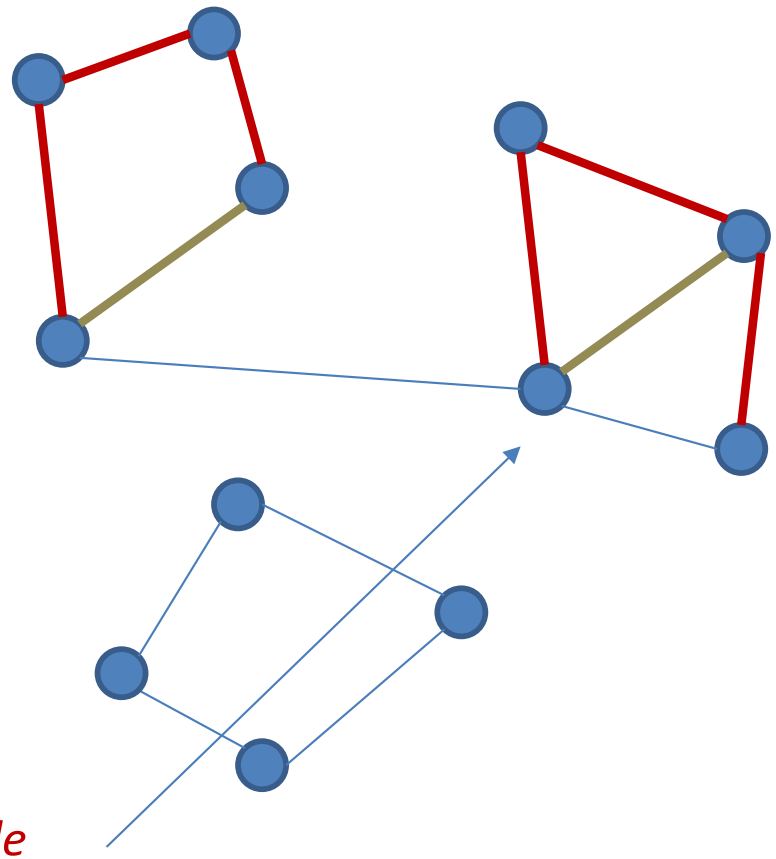
F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F.
    if F ∪ e has odd cycle then
        return NO.
return YES



Is graph bipartite?

Bipartite Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F.
    if F ∪ e has odd cycle then
        return NO.
return YES

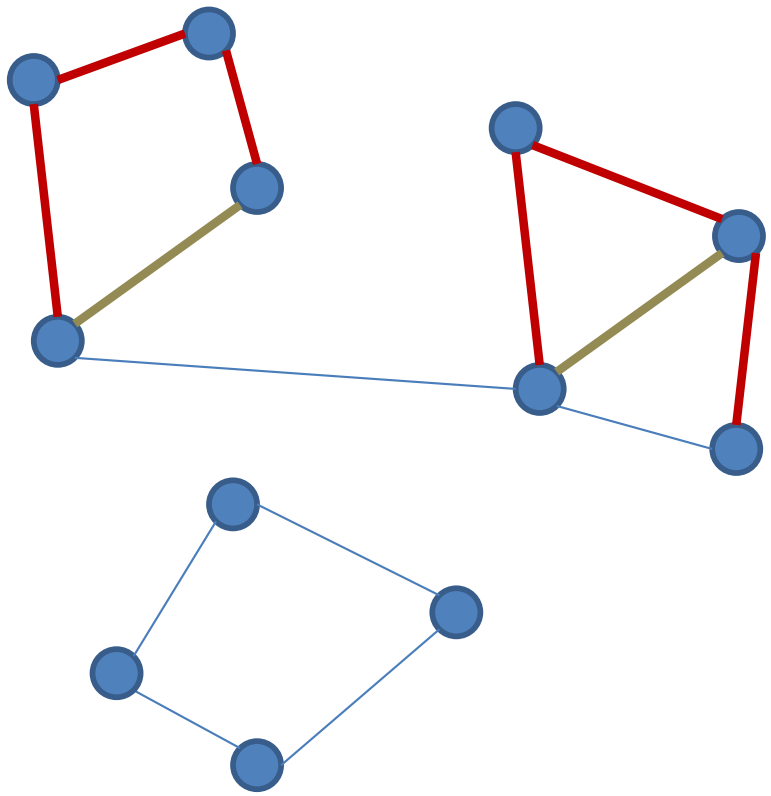


odd cycle
return NOT BIPARTITE

Is graph bipartite?

Bipartite Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F.
    if F ∪ e has odd cycle then
        return NO.
return YES

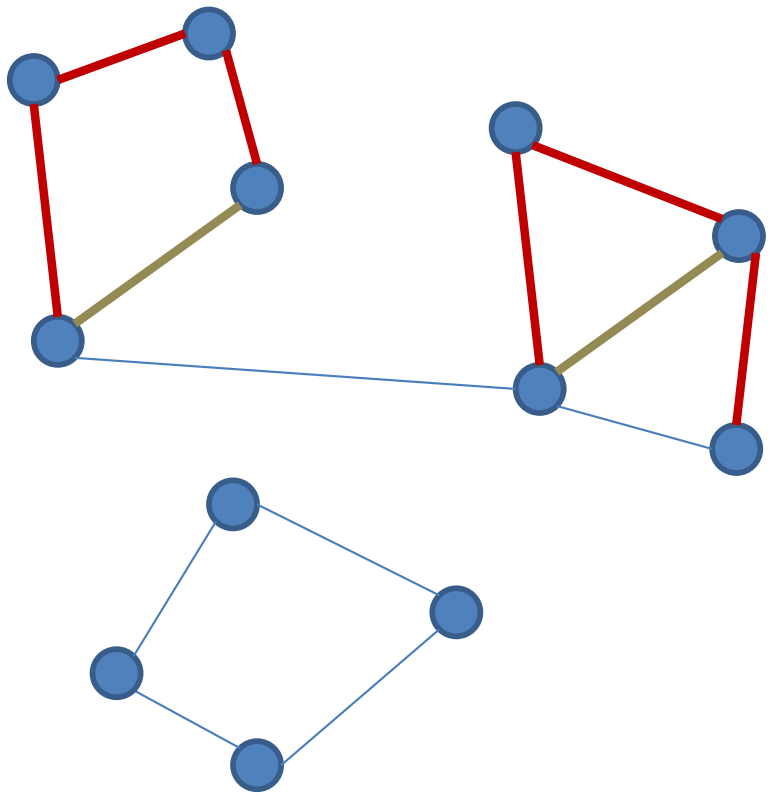


Proof:

Is graph bipartite?

Bipartite Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F.
    if F ∪ e has odd cycle then
        return NO.
return YES



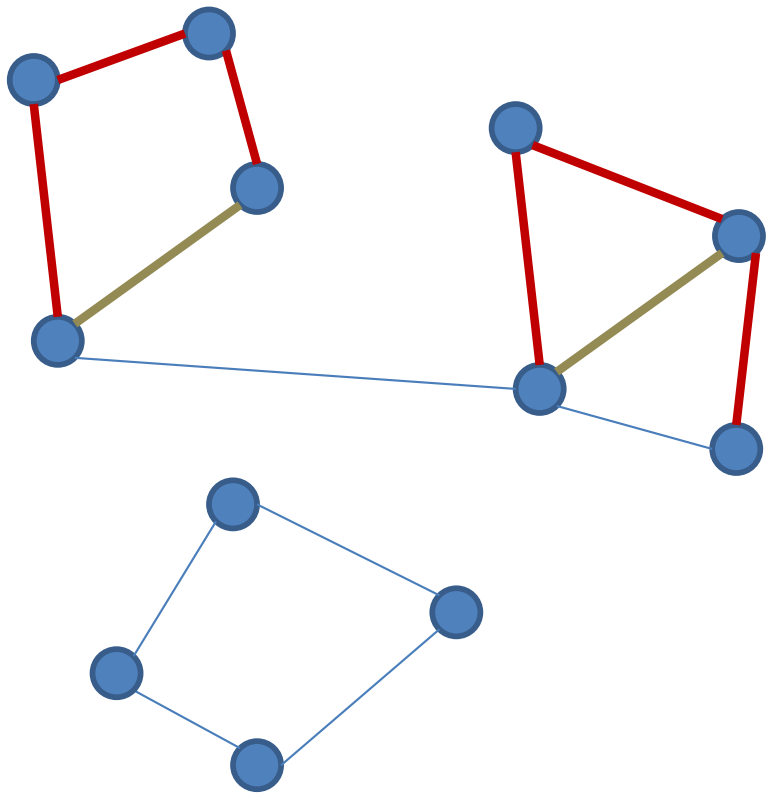
Proof:

If G is bipartite, always return YES because there are no odd cycles.

Is graph bipartite?

Bipartite Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F.
    if F ∪ e has odd cycle then
        return NO.
return YES



Proof:
If G is not bipartite?

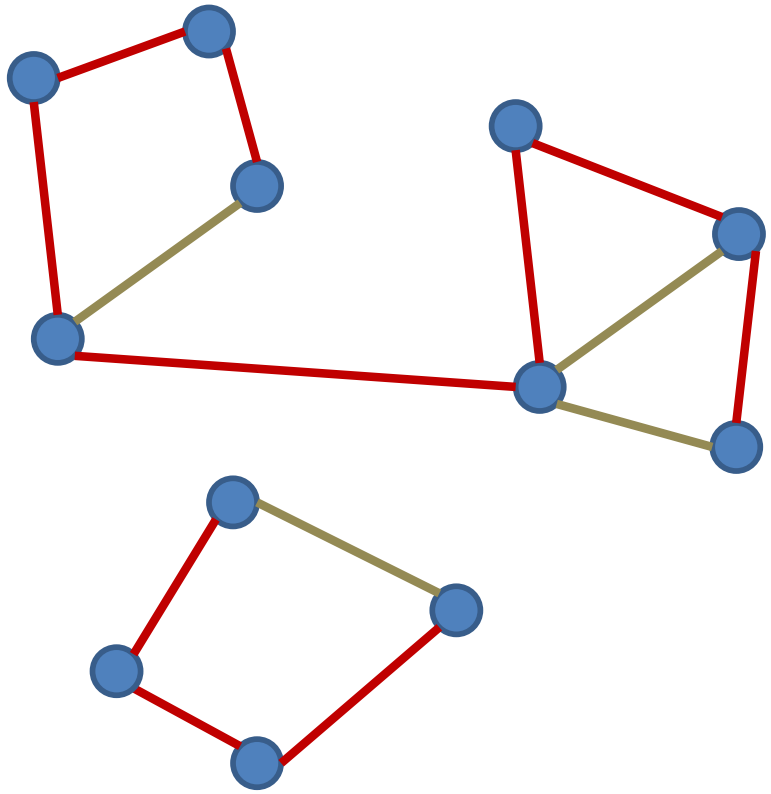
Is graph bipartite?

Bipartite Spanning Forest

Proof:

Assume G is not bipartite,
not detected.

Look at final forest.



Is graph bipartite?

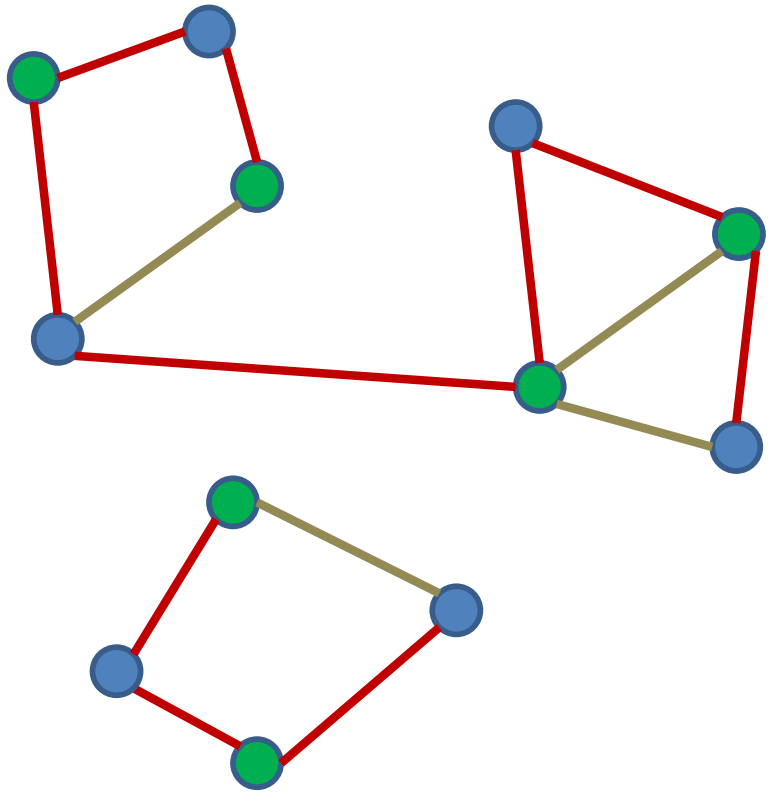
Bipartite Spanning Forest

Proof:

Assume G is not bipartite,
not detected.

Look at final forest.
2-color the nodes in the forest.

(Note: coloring must fail for graph, because graph is not bipartite. But can be good for forest.)



Is graph bipartite?

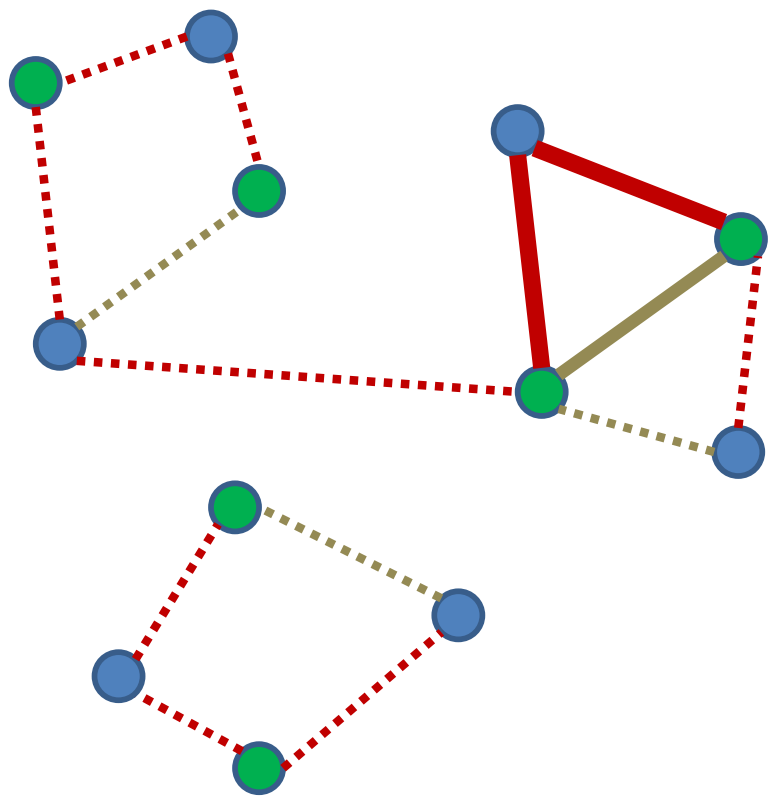
Bipartite Spanning Forest

Proof:

Assume G is not bipartite,
not detected.

Look at final forest.
2-color the nodes in the forest.

Look at mis-colored edge that was
not included in forest.



Is graph bipartite?

Bipartite Spanning Forest

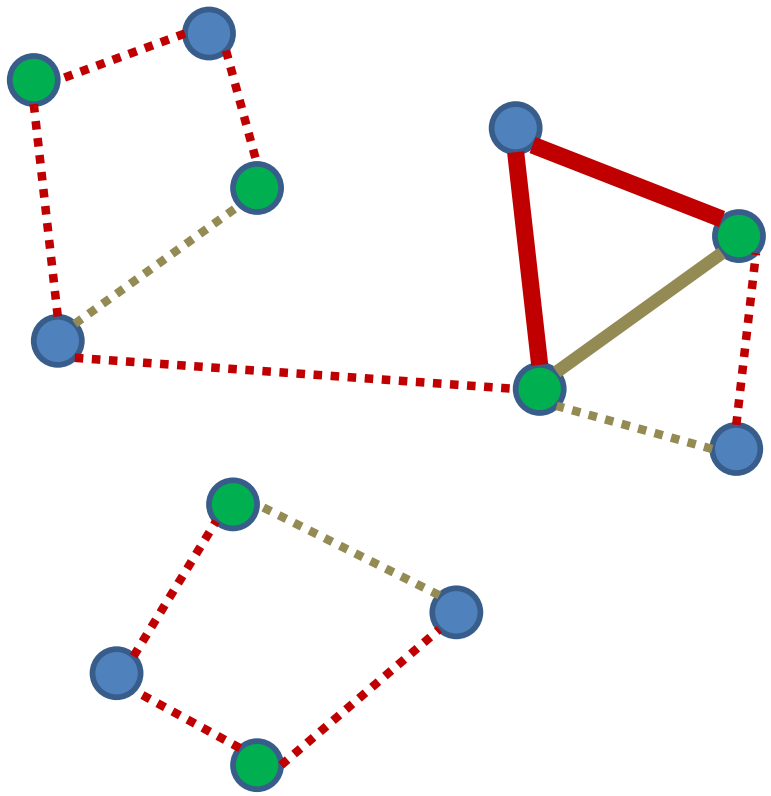
Proof:

Assume G is not bipartite,
not detected.

Look at final forest.
2-color the nodes in the forest.

Look at mis-colored edge that was not included in forest.

When edge e was seen in stream, there was an even cycle in $F \cup e$.



Is graph bipartite?

Bipartite Spanning Forest

Proof:

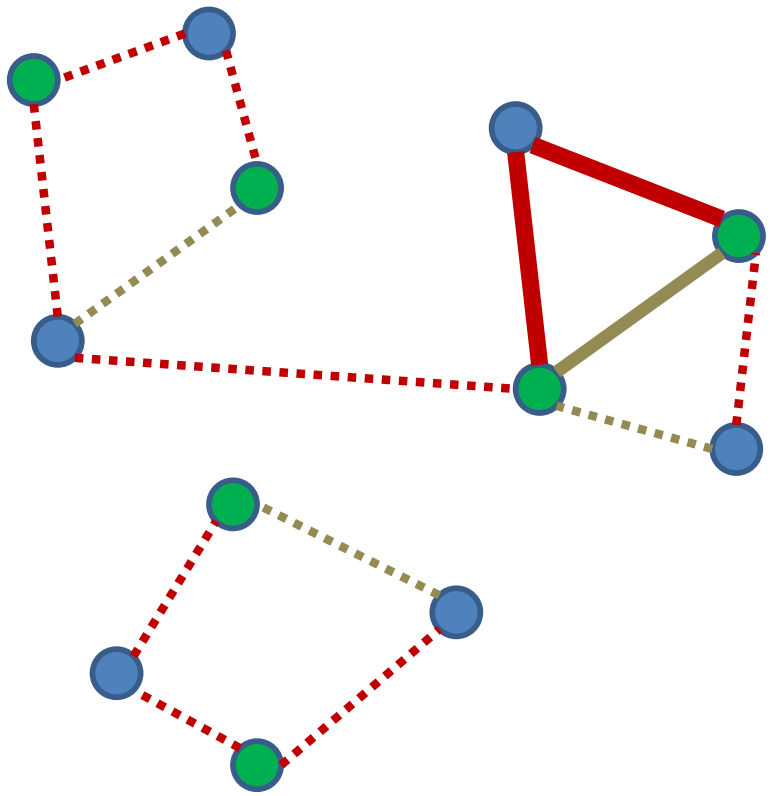
Assume G is not bipartite,
not detected.

Look at final forest.
2-color the nodes in the forest.

Look at mis-colored edge that was not included in forest.

When edge e was seen in stream, there was an even cycle in $F \cup e$.

Even cycle ➔ properly colored.



Is graph bipartite?

Bipartite Spanning Forest

Proof:

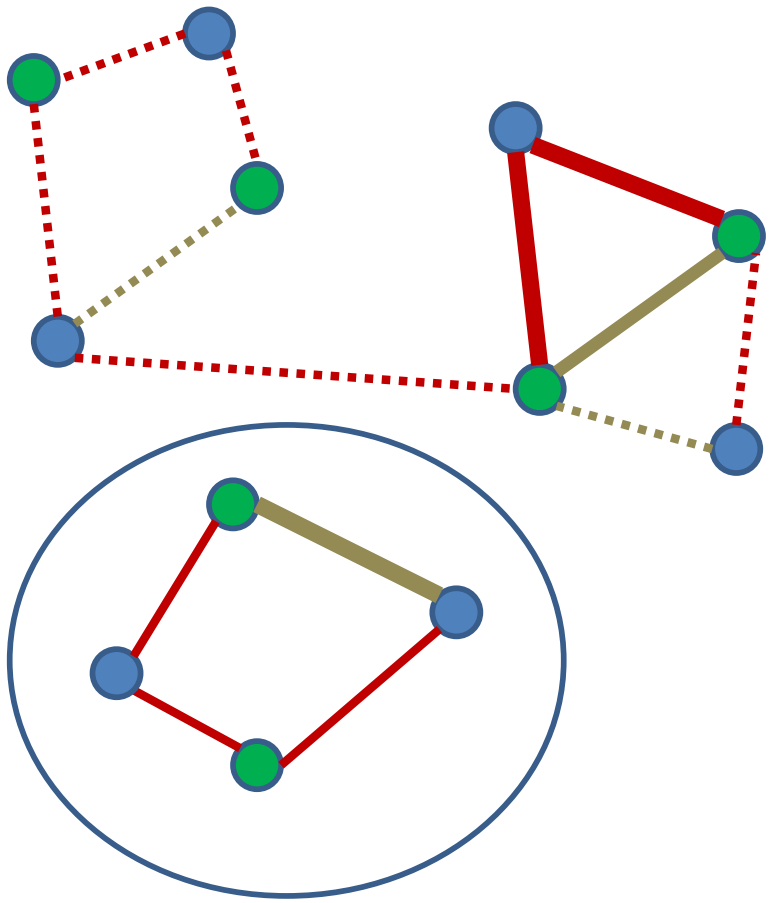
Assume G is not bipartite,
not detected.

Look at final forest.
2-color the nodes in the forest.

Look at mis-colored edge that was not included in forest.

When edge e was seen in stream, there was an even cycle in $F \cup e$.

Even cycle ➔ properly colored.



Is graph bipartite?

Bipartite Spanning Forest

Proof:

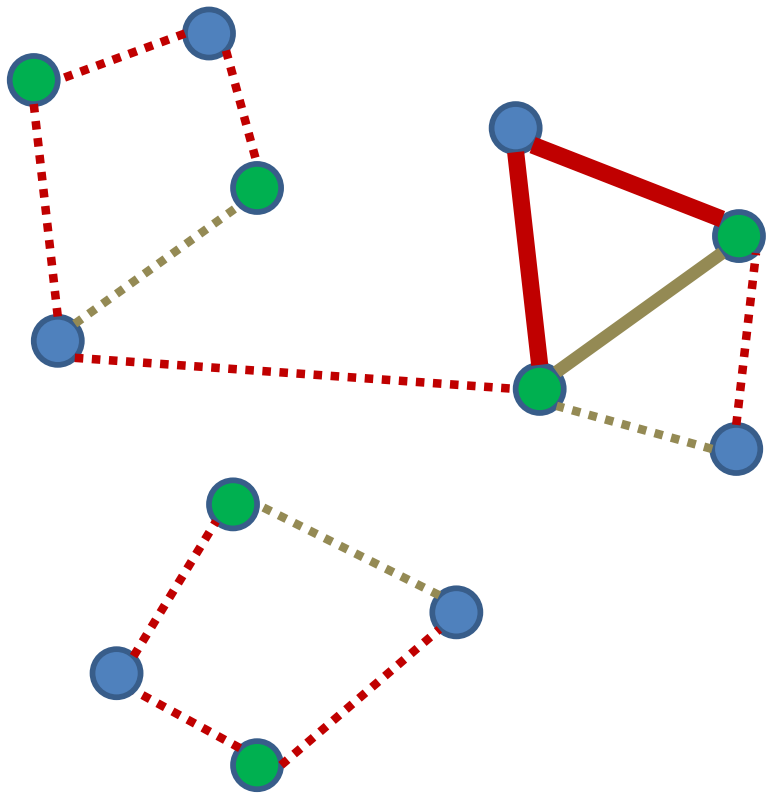
Assume G is not bipartite,
not detected.

Look at final forest.
2-color the nodes in the forest.

Look at mis-colored edge that was not included in forest.

When edge e was seen in stream, there was an even cycle in F ∪ e.

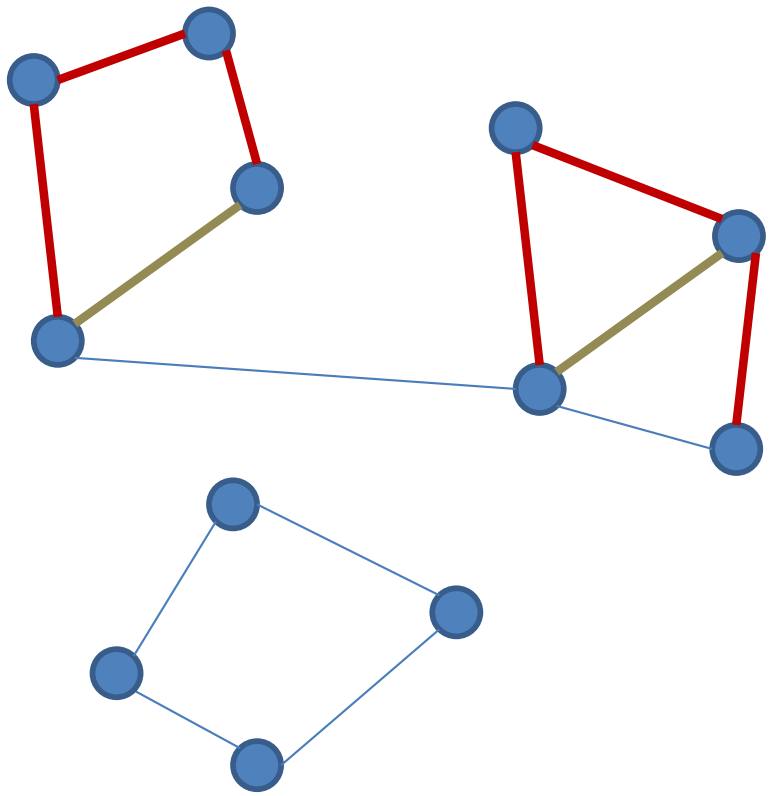
Even cycle ➔ properly colored.
Contradiction.



Is graph bipartite?

Bipartite Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F.
    if F ∪ e has odd cycle then
        return NO.
return YES

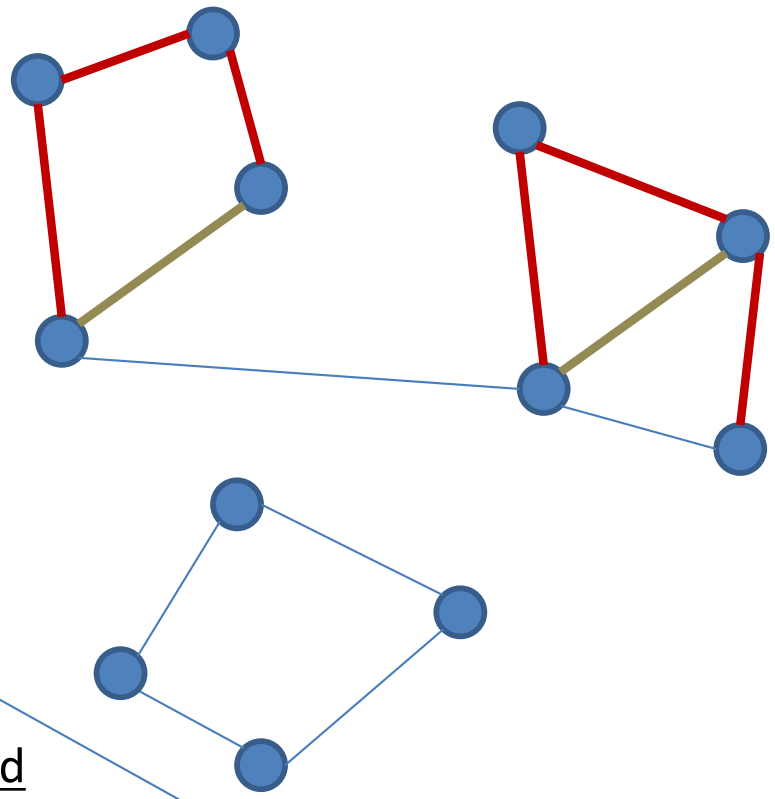


Space: O(n log n)

Is graph bipartite?

Bipartite Spanning Forest

F : forest, initially empty
for each edge e in stream:
    if F ∪ e has no cycles then
        add e to F.
    if F ∪ e has odd cycle then
        return NO.
return YES



Space: O(n log n)
Update cost: O(α(n, n,))

Union-Find
Inverse-Ackerman
amortized cost.

Coloring
Maintain 2-coloring

Shortest Paths

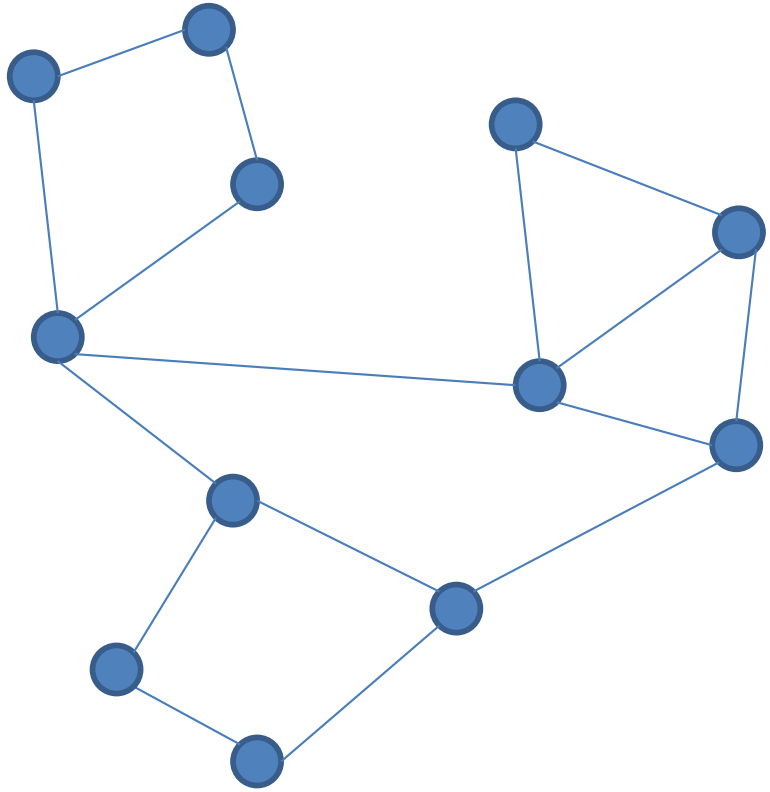
Assumptions:

Graph G = (V,E)

- Undirected
- n nodes
- m edges
- Goal: O(n log n) space.

Output:

Find a shortest path from u to v?



Shortest Paths

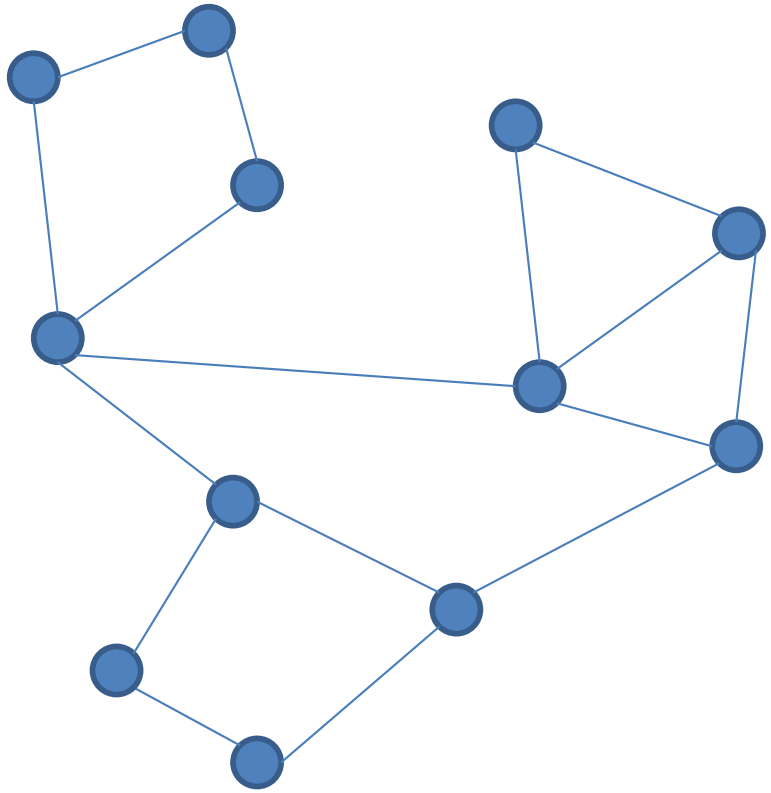
Assumptions:

Graph G = (V,E)

- Undirected
- n nodes
- m edges
- Goal: O(n log n) space.

Output:

Find an APPROXIMATE shortest path between all pairs?

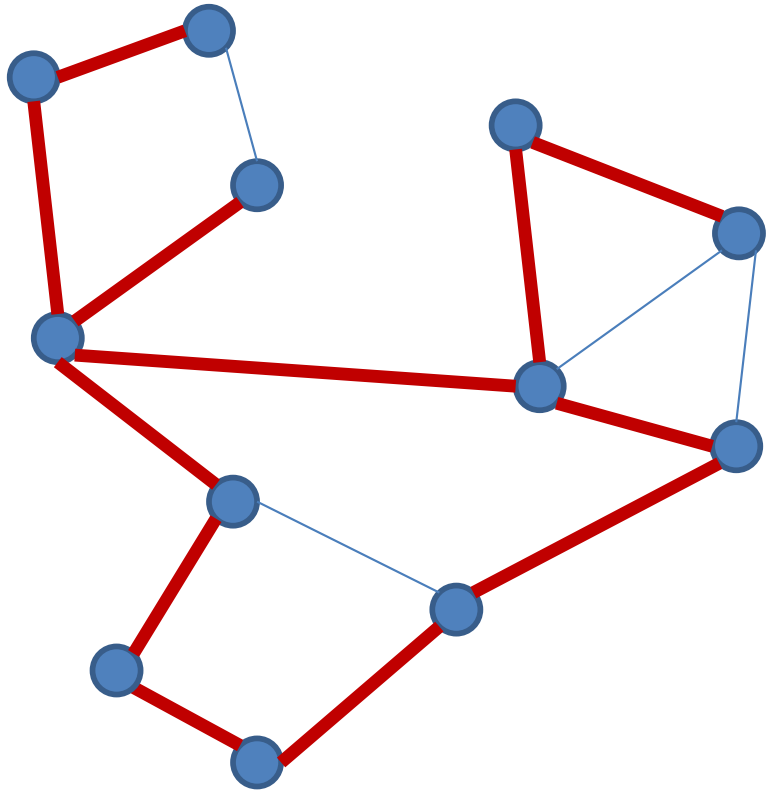


Shortest Paths

Assumptions:

Graph G = (V,E)

- Undirected
- n nodes
- m edges
- Goal: O(n log n) space.



Output:

Find an APPROXIMATE shortest path between all pairs?

Find a “good” subgraph…

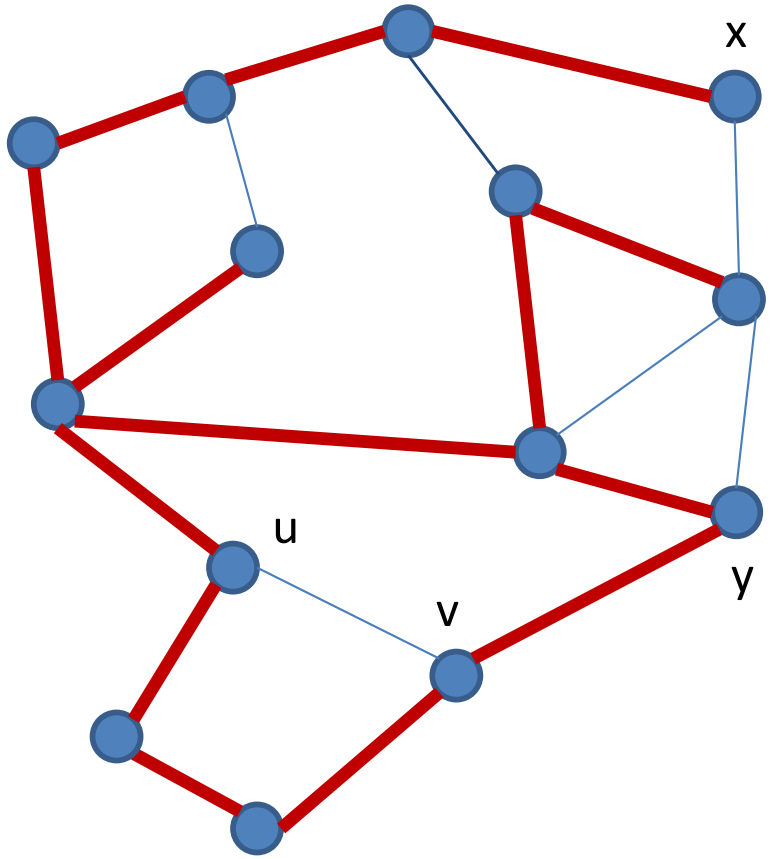
Spanner

Find a subgraph $H \subseteq G$:

- H is sparse
  (not too many edges)
- For all pairs nodes (u, v):

$$d_G(u,v) \leq d_H(u,v) \leq \alpha d_G(u,v)$$

"Shortest path in H is close to the real shortest path."



Stretch: α

"ratio of spanner shortest path to real shortest path"

$$d_G(u,v) = 1 \quad d_G(x,y) = 2$$
$$d_H(u,v) = 3 \quad d_H(x,y) = 6$$

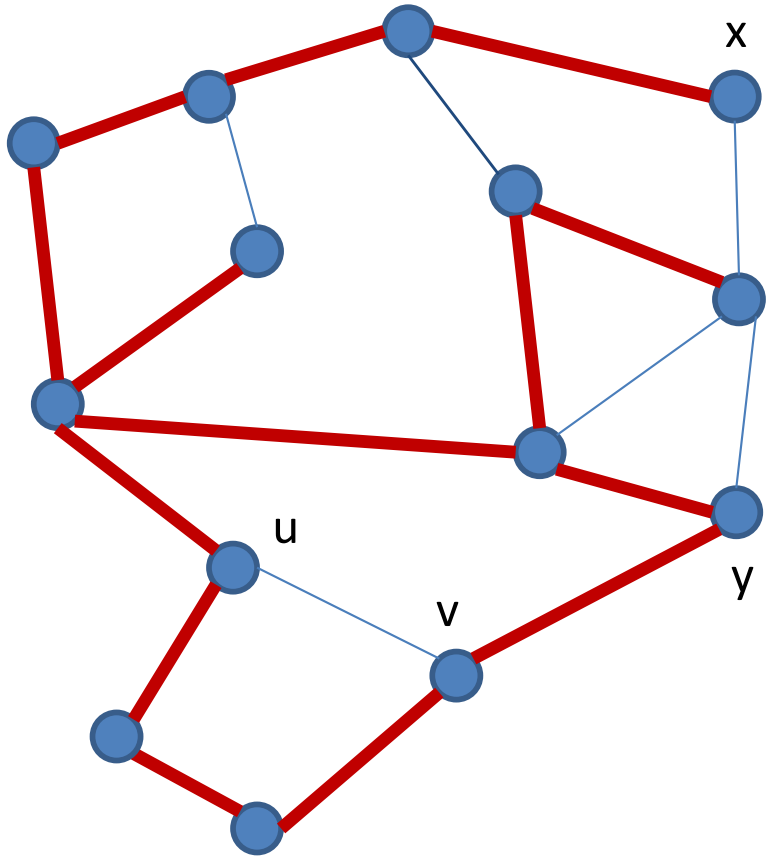
Spanner

Find a subgraph $H \subseteq G$:

- H is sparse
  (not too many edges)
- For all pairs nodes (u, v):

$$d_G(u,v) \leq d_H(u,v) \leq \alpha d_G(u,v)$$

"Shortest path in H is close to the real shortest path."



Stretch: α

"ratio of spanner shortest path to real shortest path"

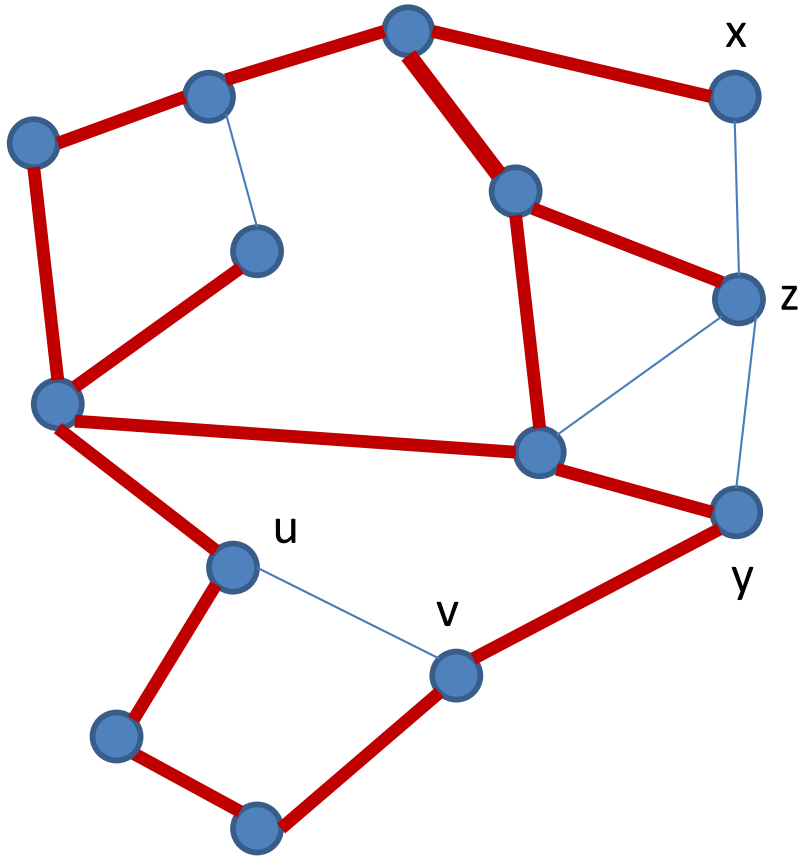
$$d_G(u,v) = 1 \quad d_G(x,y) = 2$$
$$d_H(u,v) = 3 \quad d_H(x,y) = 6$$

Spanner

To find stretch:

- Look at every edge (u,v) in the graph G.
- Take the maximum value of:

$$\frac{d_H(u,v)}{d_G(u,v)}$$



Spanner

To find stretch:

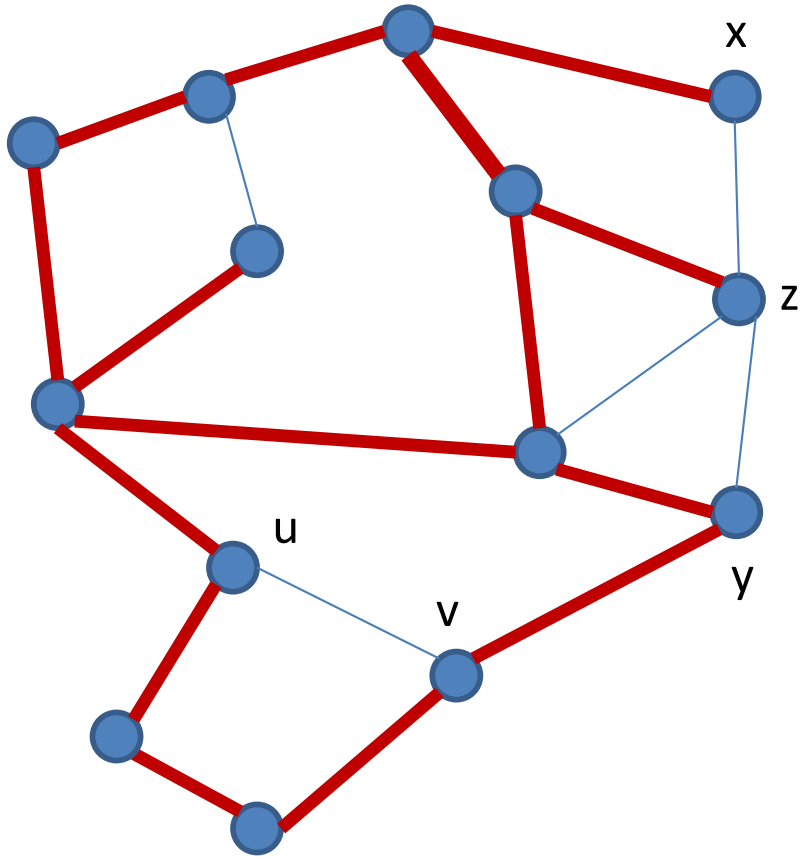
- Look at every edge (u,v) in the graph G.
- Take the maximum value of:

$$\frac{d_H(u,v)}{d_G(u,v)}$$

What about other (x,y)?

$$P = (x, u_1, u_2, u_3, u_4, y)$$

shortest path from x to y in graph G



Spanner

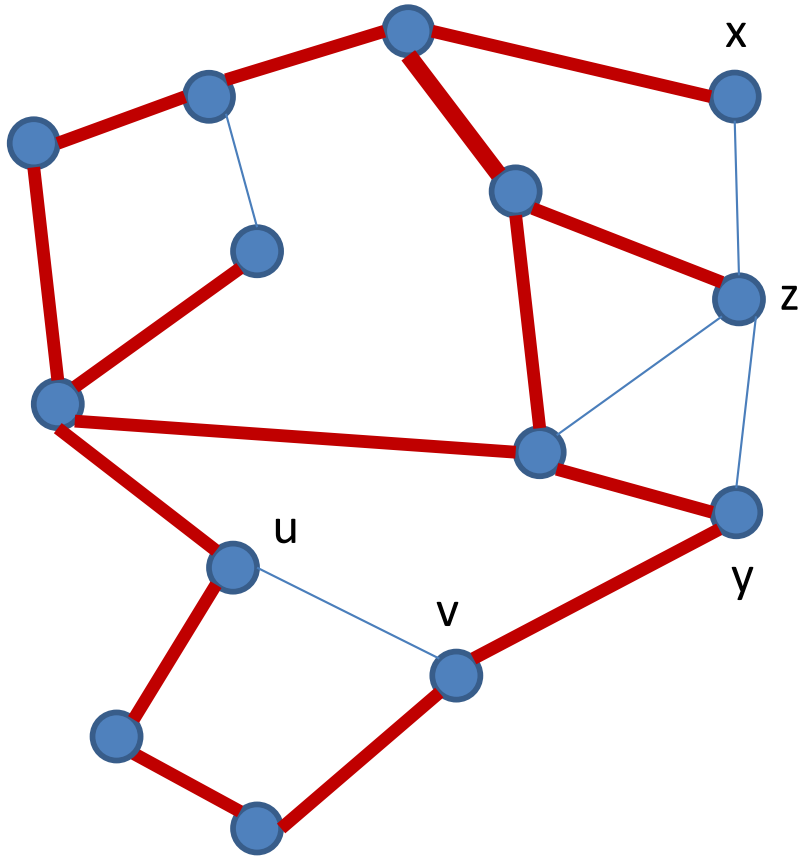
What about other (x,y)?

$$P = (x, u_1, u_2, u_3, u_4, y)$$

shortest path from x to y in graph G

$$\begin{aligned} P_1 &= (x, u_1) \\ P_2 &= (u_1, u_2) \\ P_3 &= (u_2, u_3) \\ P_4 &= (u_3, u_4) \\ P_5 &= (u_4, y) \end{aligned}$$

shortest paths in graph H



Spanner

What about other (x,y)?

$$P = (x, u_1, u_2, u_3, u_4, y)$$

shortest path from x to y in graph G

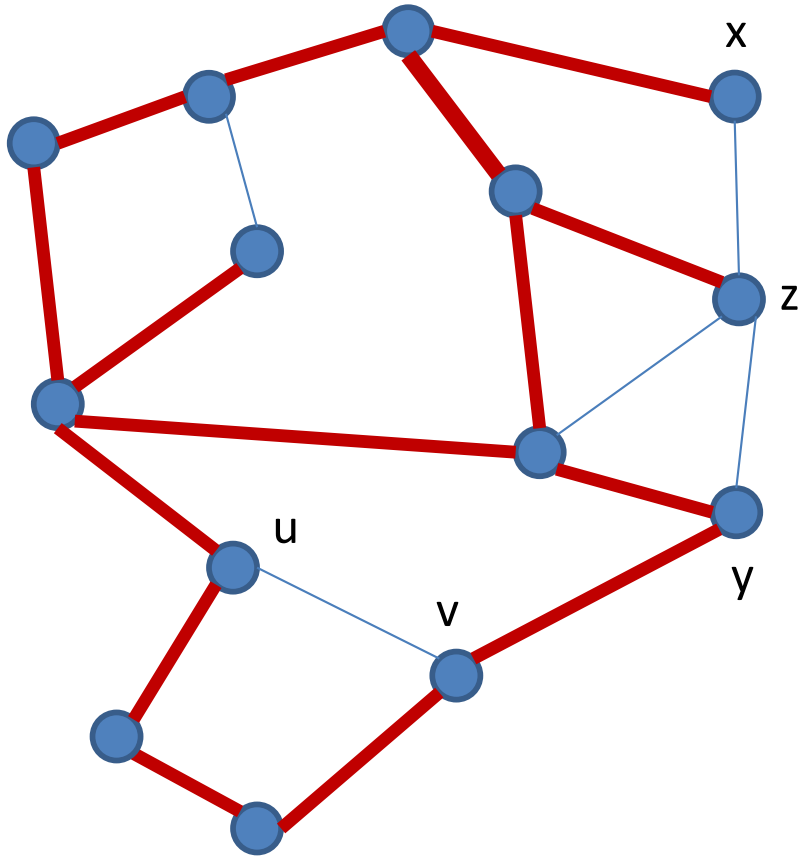
$P_1 = (x, u_1)$ ⇒ distance α in graph H

$P_2 = (u_1, u_2)$ ⇒ distance α in graph H

$P_3 = (u_2, u_3)$ ⇒ distance α in graph H

$P_4 = (u_3, u_4)$ ⇒ distance α in graph H

$P_5 = (u_4, y)$ ⇒ distance α in graph H



Spanner

What about other (x,y)?

$$P = (x, u_1, u_2, u_3, u_4, y)$$

shortest path from x to y in graph G

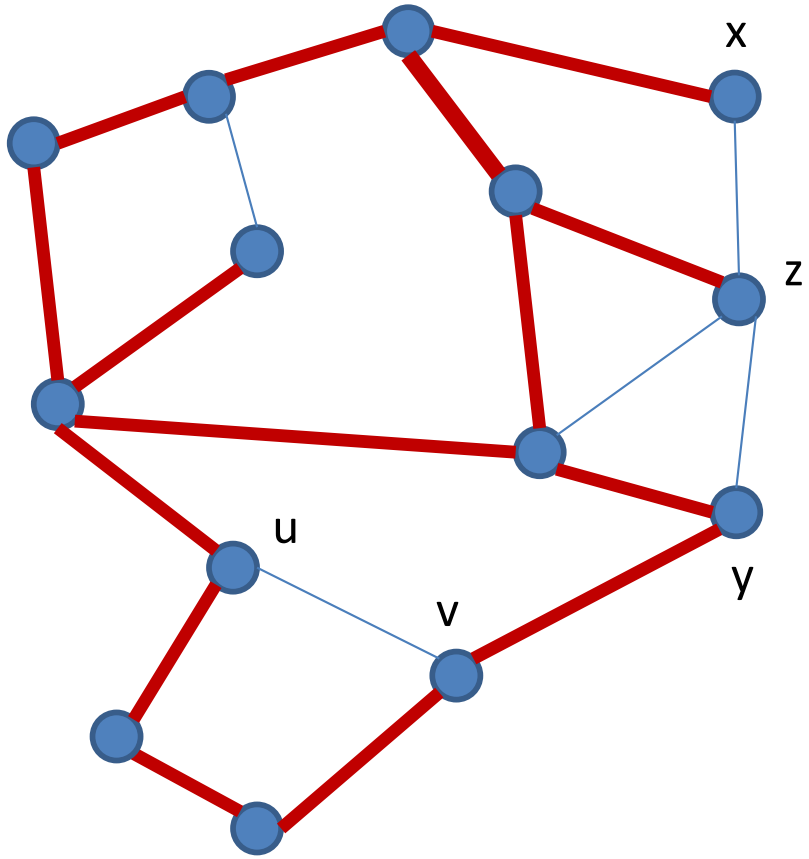
$P_1 = (x, u_1)$ ⇒ distance α in graph H

$P_2 = (u_1, u_2)$ ⇒ distance α in graph H

$P_3 = (u_2, u_3)$ ⇒ distance α in graph H

$P_4 = (u_3, u_4)$ ⇒ distance α in graph H

$P_5 = (u_4, y)$ ⇒ distance α in graph H



$P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_5$ ⇒ path of length $\alpha|P|$ in graph H

Spanner

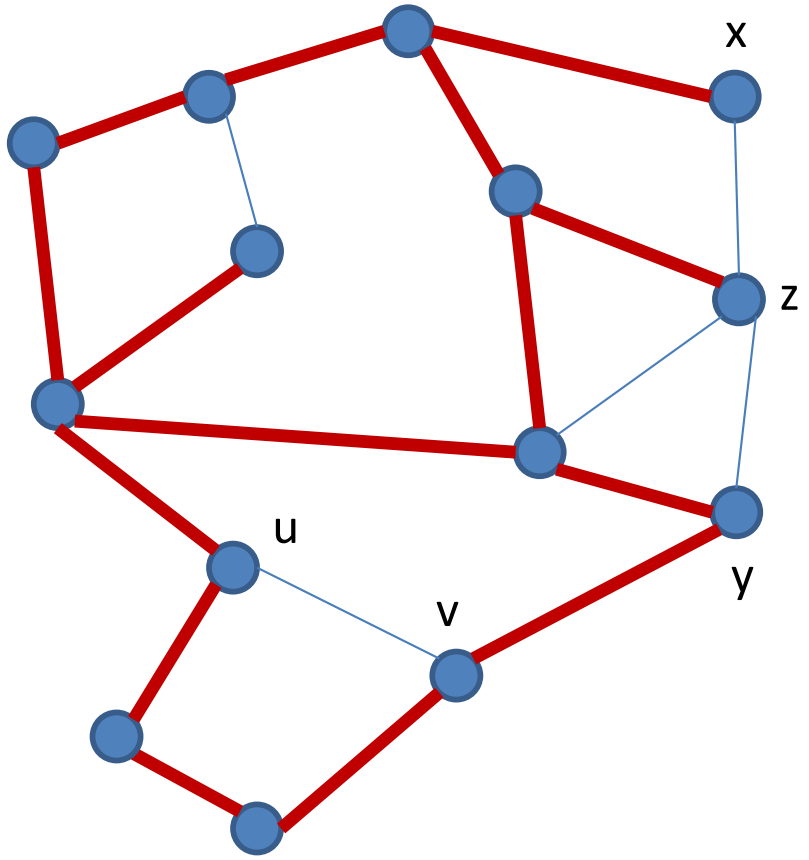
To find stretch:

- Look at every edge (u,v) in the graph G.
- Take the maximum value of:

$$\frac{d_H(u,v)}{d_G(u,v)}$$

Strategy:

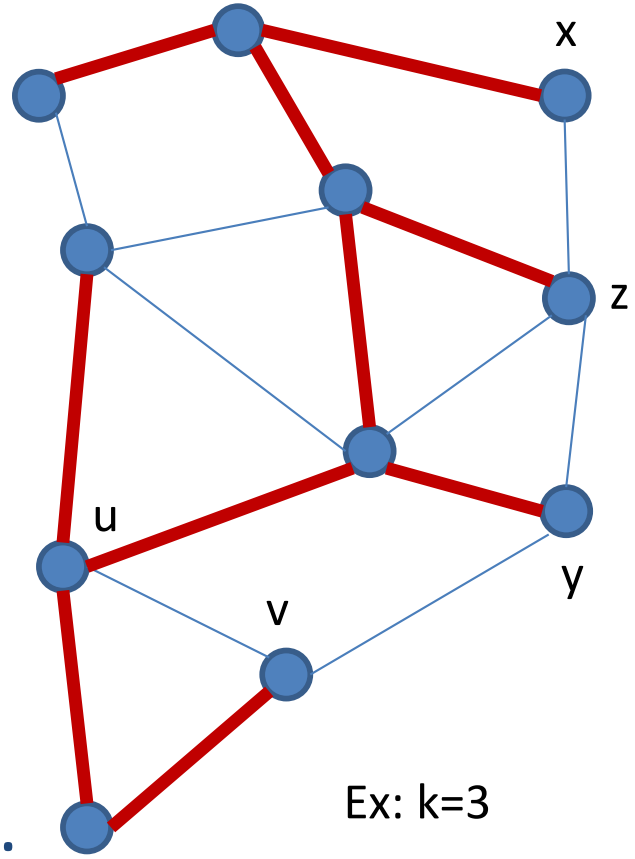
- Remove edges on short cycles.
- If alternative path is < 2k, then delete edge.



Spanner

Spanner Construction(k)

H : subgraph, initially empty
for each edge e = (u,v) in stream:
    if $d_H(u,v) > 2k-1$ (in H)
        add e to H.
return H



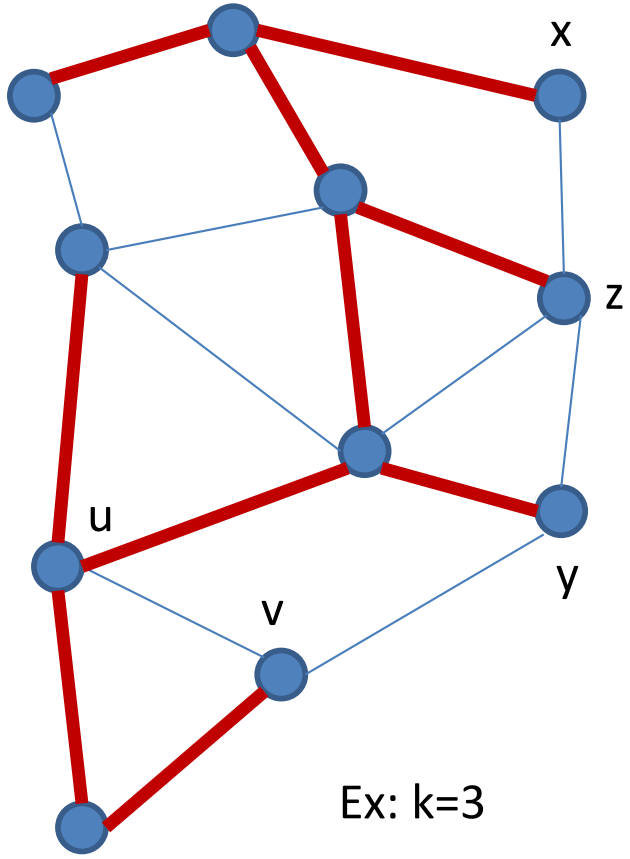
Ex: k=3

Remove all small cycles from the graph.
Parameter: k

Spanner

Spanner Construction(k)

H : subgraph, initially empty
for each edge e = (u,v) in stream:
    if $d_H(u,v) > 2k-1$ (in H)
        add e to H.
return H



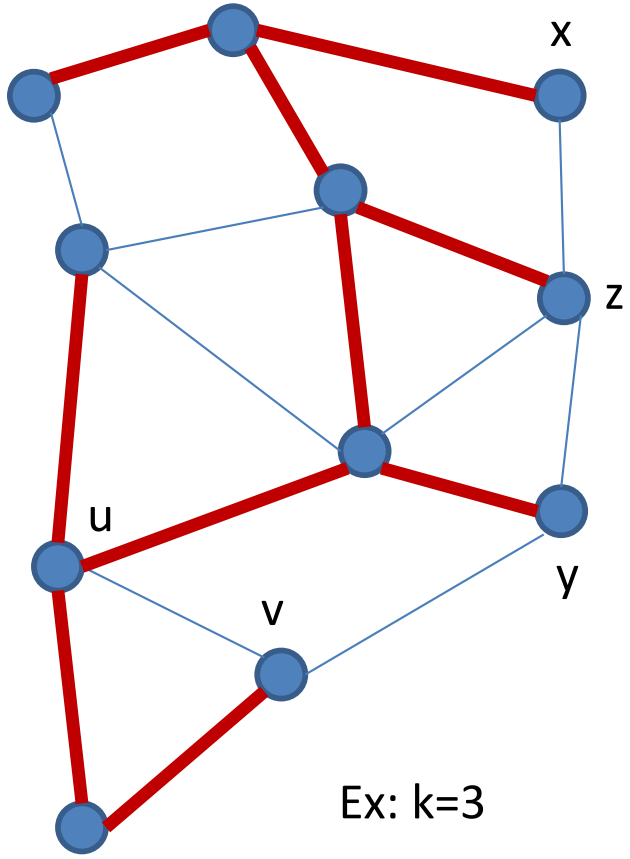
Ex: k=3

Claim:
H has stretch at most 2k-1.

Spanner

Spanner Construction(k)

H : subgraph, initially empty
for each edge e = (u,v) in stream:
    if $d_H(u,v) > 2k-1$ (in H)
        add e to H.
return H



Ex: k=3

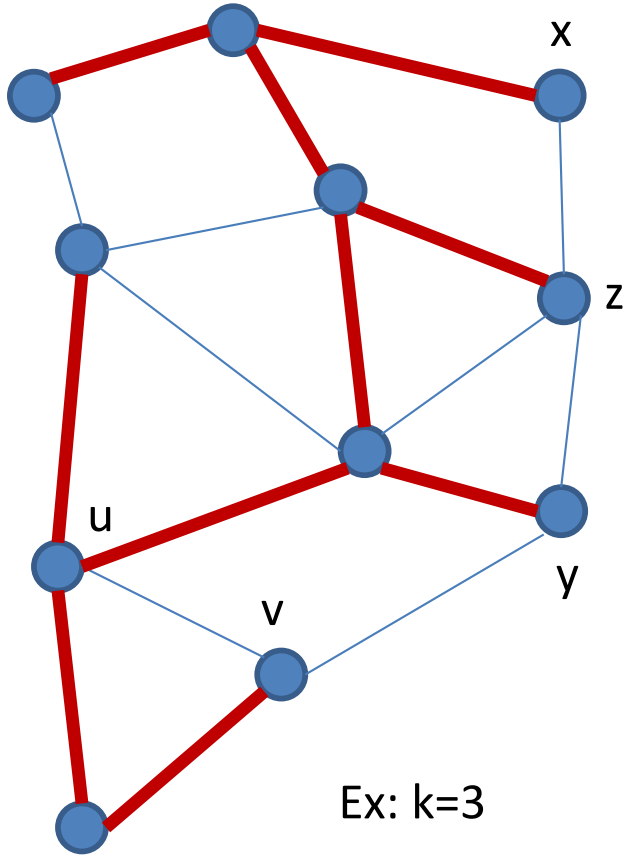
Claim:
H has stretch at most 2k-1.

Proof: only delete edge if there is path in H ≤ 2k-1.

Spanner

Spanner Construction(k)

H : subgraph, initially empty
for each edge e = (u,v) in stream:
    if $d_H(u,v) > 2k-1$ (in H)
        add e to H.
return H



Ex: k=3

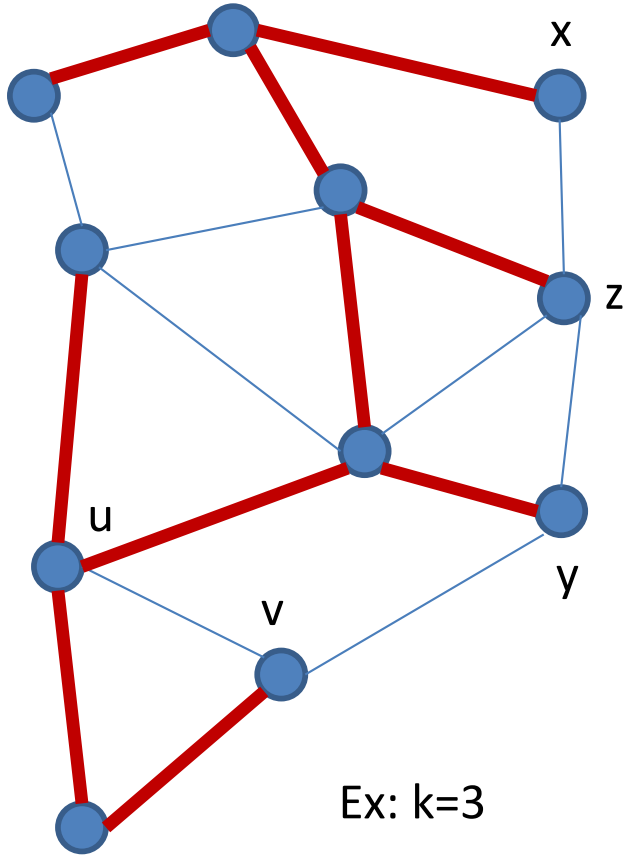
Claim:
H has no cycles of size ≤ 2k.

Proof: only add edge if there is no path in H ≤ 2k-1.

Spanner

Spanner Construction(k)

H : subgraph, initially empty
for each edge e = (u,v) in stream:
    if $d_H(u,v) > 2k-1$ (in H)
        add e to H.
return H



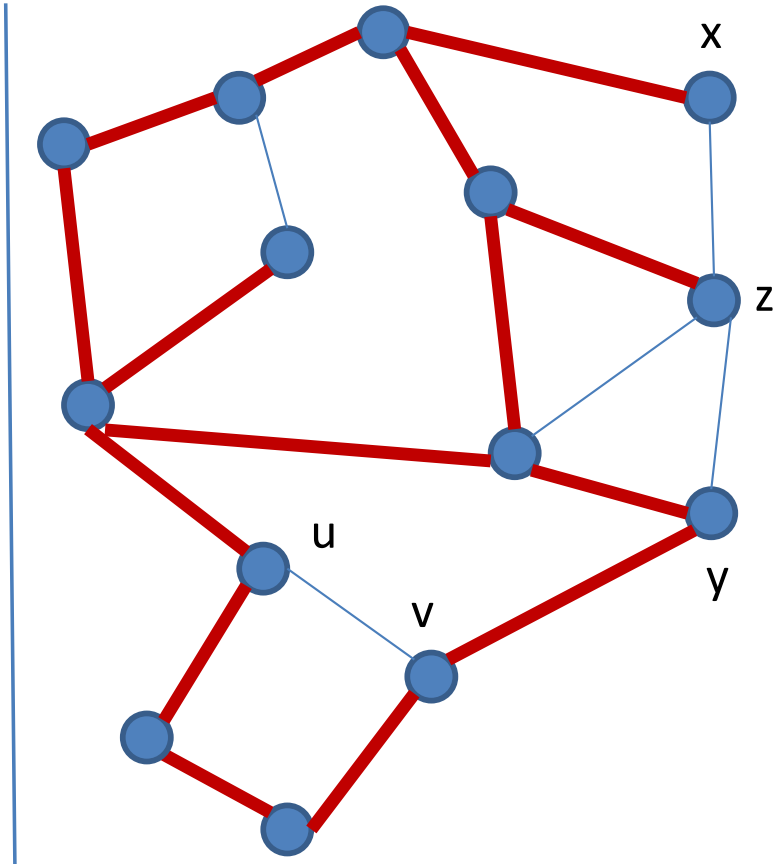
Ex: k=3

Key Question:
How big is H?

Spanner

Definition:

girth(G) = size of smallest cycle in G.



Example:
Girth = 6

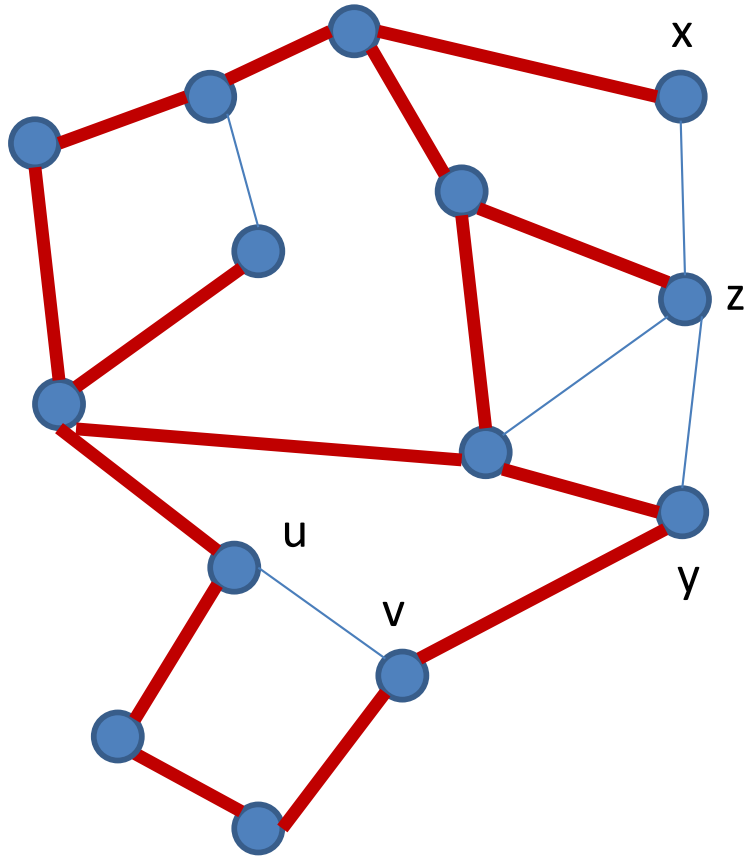
Spanner

Definition:

girth(G) = size of smallest cycle in G.

Theorem:

If graph G has girth(G) > 2k, then it has $O(n^{1+\frac{1}{k}})$ edges.



Example:
Girth = 6

Spanner

Theorem:

If graph G has girth(G) > 2k, then it has $O(n^{1+\frac{1}{k}})$ edges.

Proof:

Let H be a graph with $> 10n^{1+\frac{1}{k}}$ edges and girth(G) > 2k.

Spanner

Theorem:

If graph G has girth(G) > 2k, then it has $O(n^{1+\frac{1}{k}})$ edges.

Proof: (by contradiction)

Let H be a graph with $> 10n^{1+\frac{1}{k}}$ edges and girth(G) > 2k.

Kill low degree nodes:

Repeat: if node u has degree $\leq 2n^{\frac{1}{k}}$ then delete it (and adjacent edges.

Spanner

Theorem:

If graph G has girth(G) > 2k, then it has $O(n^{1+\frac{1}{k}})$ edges.

Proof: (by contradiction)

Let H be a graph with > $10n^{1+\frac{1}{k}}$ edges and girth(G) > 2k.

Kill low degree nodes:

Repeat: if node u has degree ≤ $2n^{\frac{1}{k}}$ then delete it (and adjacent edges.

➔ Removes at most $2\mathrm{n} * n^{\frac{1}{k}} = 2n^{1+\frac{1}{k}}$ edges ➔ graph H is not empty.

➔ Graph H has no low degree nodes.

Spanner

Theorem:

If graph G has girth(G) > 2k, then it has $O(n^{1+\frac{1}{k}})$ edges.

Proof: (by contradiction)

H' is a graph with $> 8n^{1+\frac{1}{k}}$ edges and girth(G) > 2k, graph H has no nodes with degree $< 2n^{\frac{1}{k}}$.

Spanner

Theorem:

If graph G has girth(G) > 2k, then it has $O(n^{1+\frac{1}{k}})$ edges.

Proof: (by contradiction)

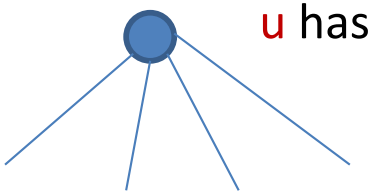
H' is a graph with > $8n^{1+\frac{1}{k}}$ edges and girth(G) > 2k, graph H has no nodes with degree < $2n^{\frac{1}{k}}$.

Choose a node u in H'. Let T be all the nodes at distance ≤ k from u.

Spanner

Choose a node u in H'. Let T be all the nodes at distance ≤ k from u.

u has degree ≥ $2n^{\frac{1}{k}}$



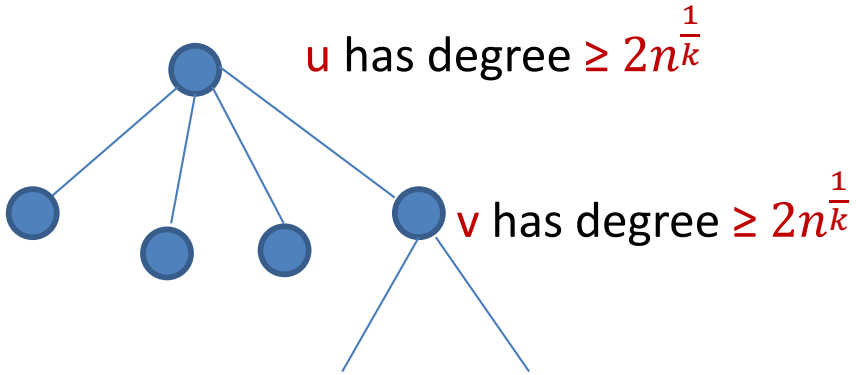
Claim: T is a tree

No cycles of length 2k.

Spanner

Choose a node u in H'. Let T be all the nodes at distance ≤ k from u.

Claim: T is a tree



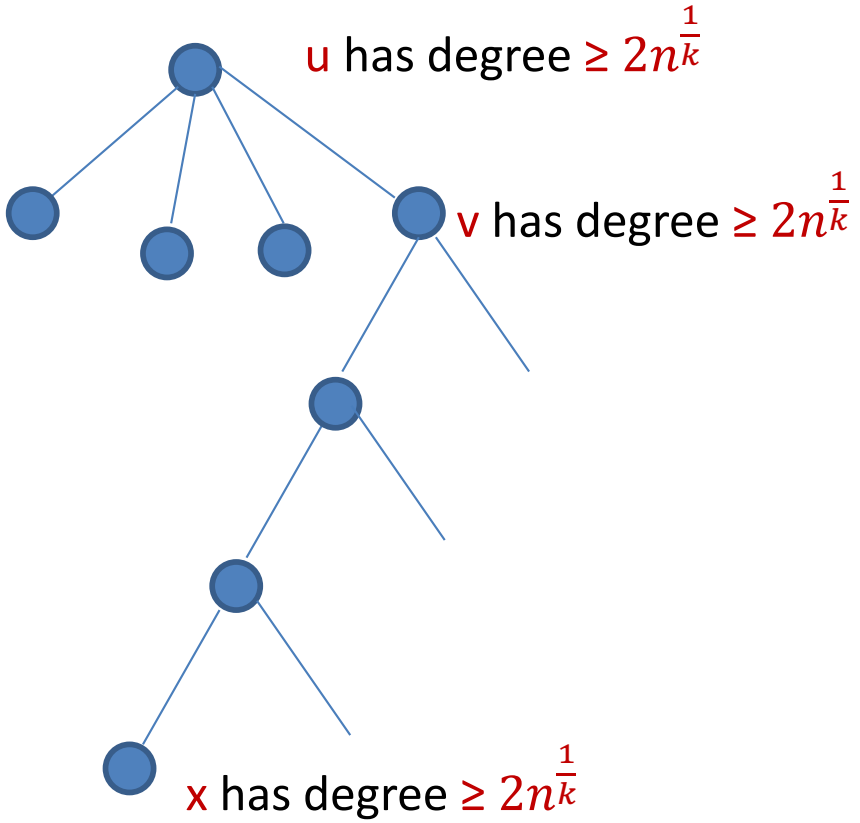
No cycles of length 2k.

Spanner

Choose a node u in H'. Let T be all the nodes at distance ≤ k from u.

Claim: T is a tree

No cycles of length 2k.



Spanner

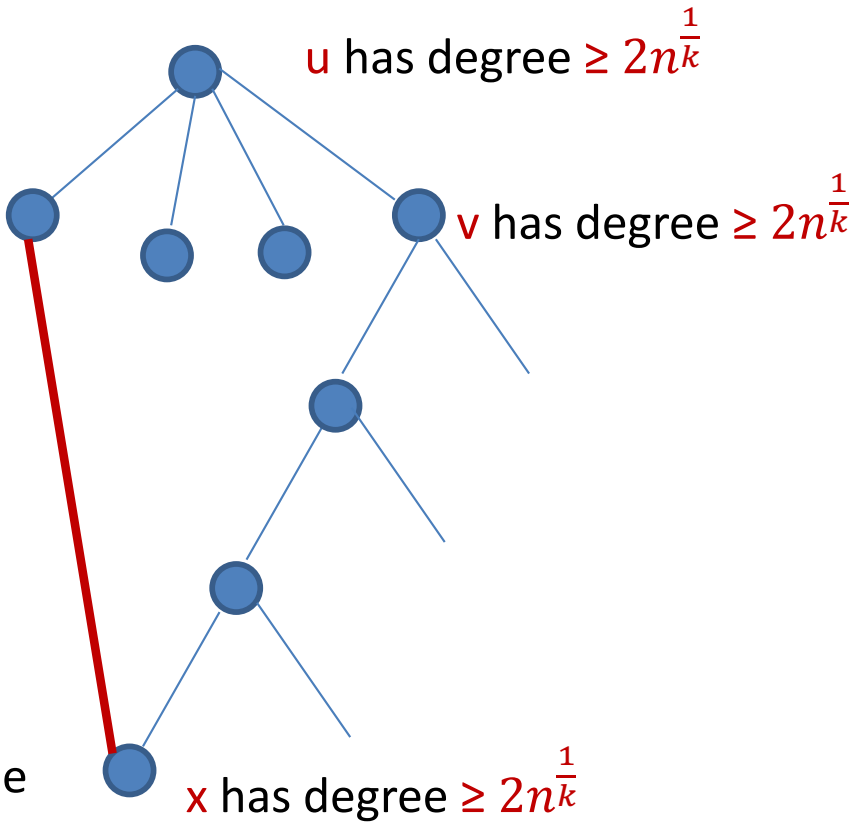
Choose a node u in H'. Let T be all the nodes at distance ≤ k from u.

Claim: T is a tree

No cycles of length 2k.

➔

T is a tree.



Any cross edge creates a cycle of length ≤ 2k.

(Ignore edges between leaves.)

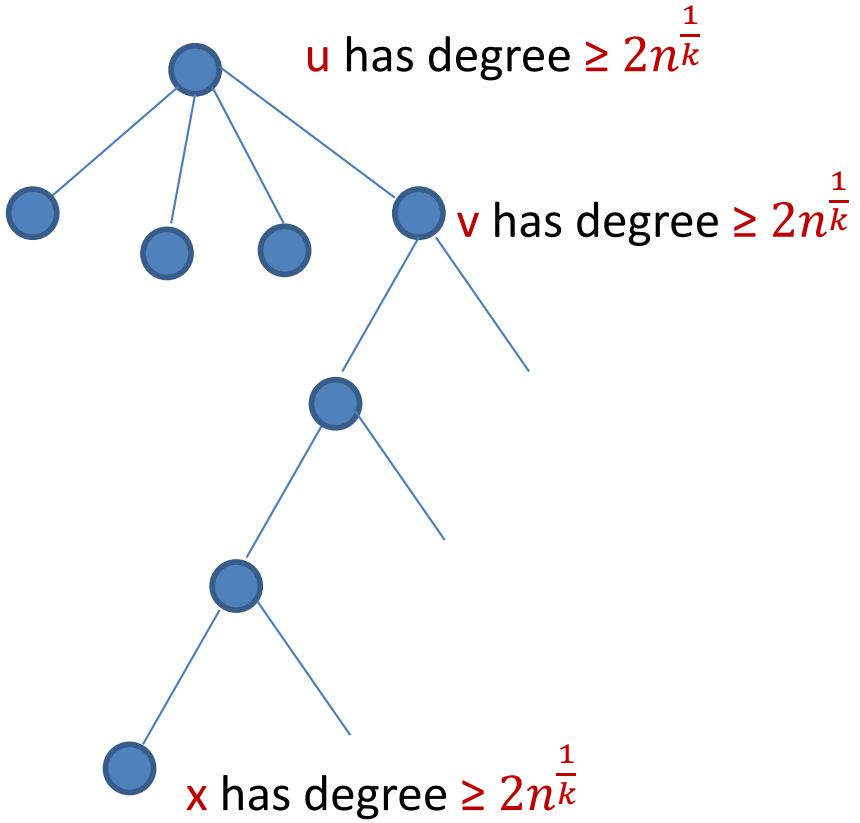
Spanner

Choose a node u in H'. Let T be all the nodes at distance ≤ k from u.

Claim: T is a tree

Number of nodes in tree T:

$$\geq \left(2n^{1/k}\right)^{k} > n$$



Contradiction!

Graph H cannot exist.

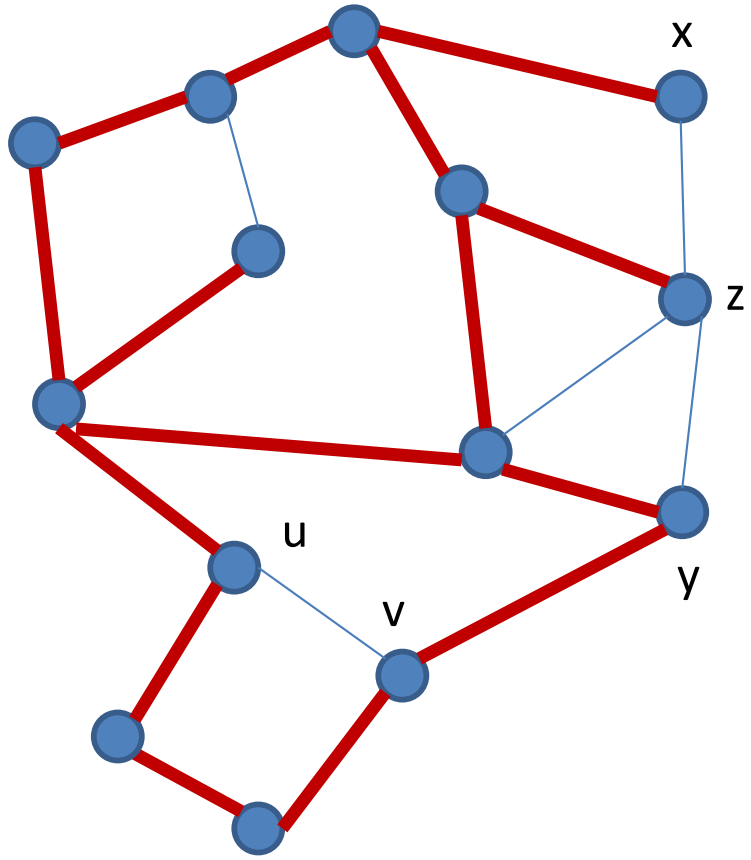
Spanner

Definition:

girth(G) = size of smallest cycle in G.

Theorem:

If graph G has girth(G) > 2k, then it has $O(n^{1+\frac{1}{k}})$ edges.

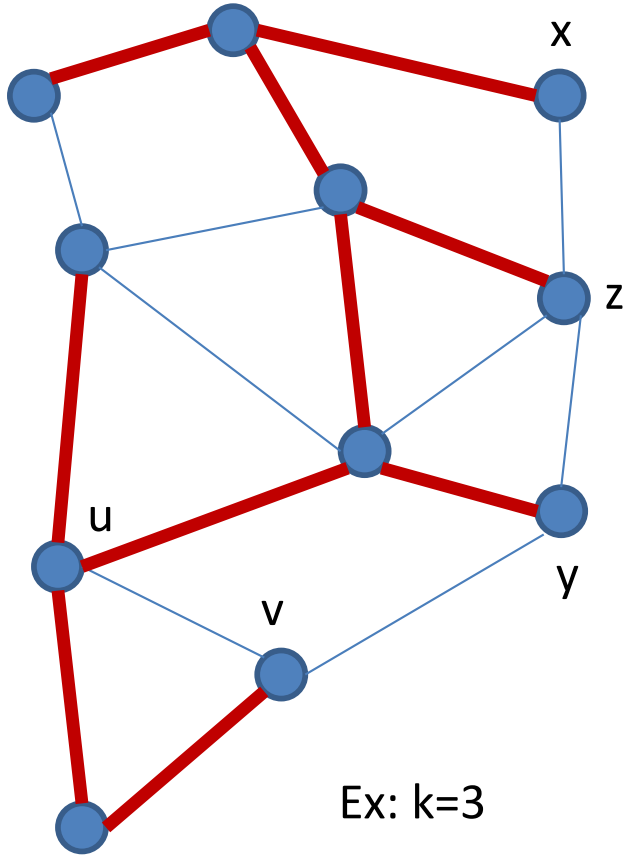


Example:
Girth = 6

Spanner

Spanner Construction(k)

H : subgraph, initially empty
for each edge e = (u,v) in stream:
    if $d_H(u,v) > 2k-1$ (in H)
        add e to H.
return H



Ex: k=3

Size of H:

Graph H has girth(H) > 2k.

➔ Graph H has $O(n^{1+\frac{1}{k}})$ edges.

Spanner

Spanner Construction(k)

Size of H:

Graph H has girth(H) > 2k.

➔ Graph H has $O(n^{1+\frac{1}{k}})$ edges.

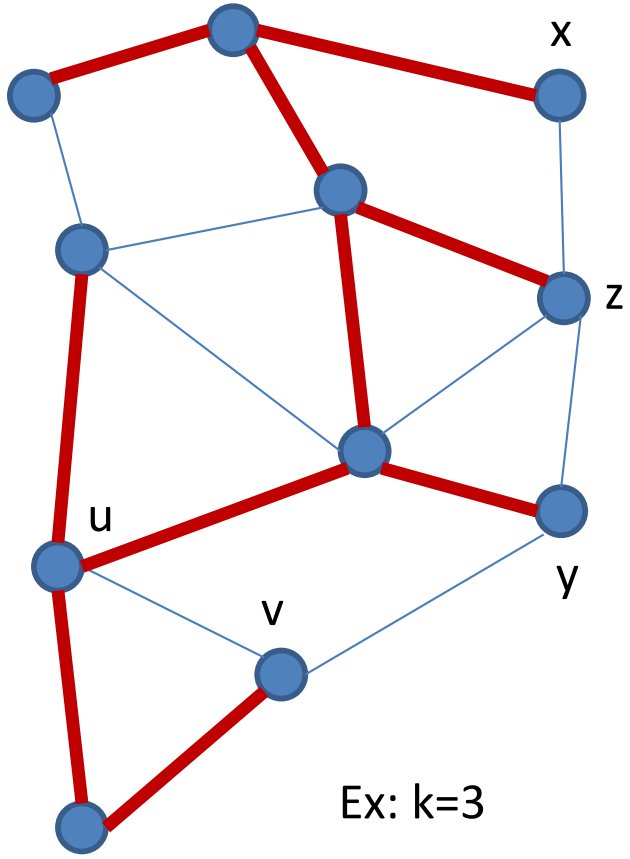
1. k = 2

   3-spanner, space: $O(n^{3/2}\log n)$

2. k = log(n)

   log(n)-spanner, space:

$$O(n^{1+1/\log n}\log n) = O(n\log n)$$

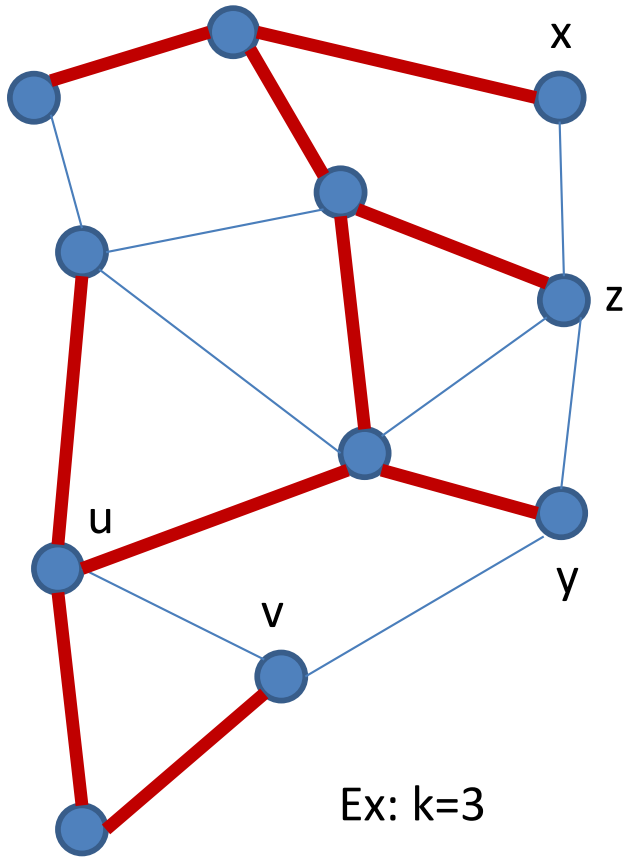


Ex: k=3

Spanner

Can we do better?

Not if the Erdos Girth Conjecture is true!



Matching

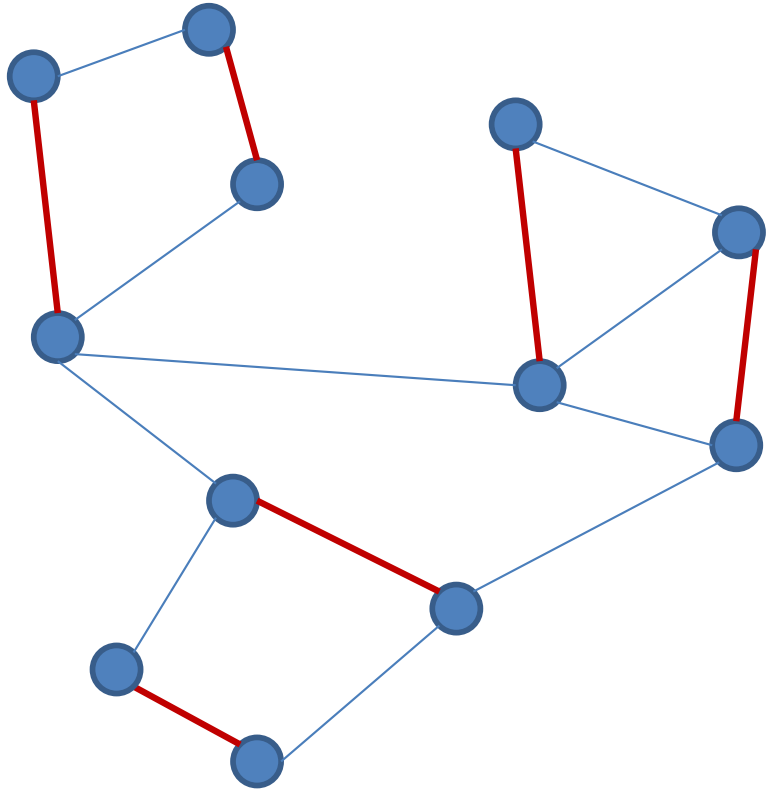
Assumptions:

Graph G = (V,E)

- Undirected
- n nodes
- m edges
- Goal: O(n log n) space.

Output:

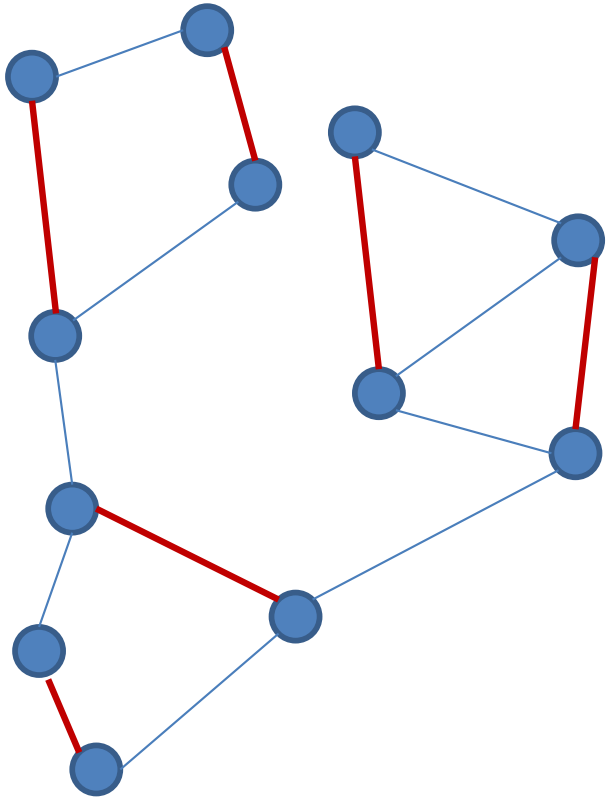
Find a maximum sized matching.



Matching

Greedy Match

M : matching, initially empty
for each edge e = (u,v) in stream:
    if u and v are not matched in M
        add (u,v) to M.
return M

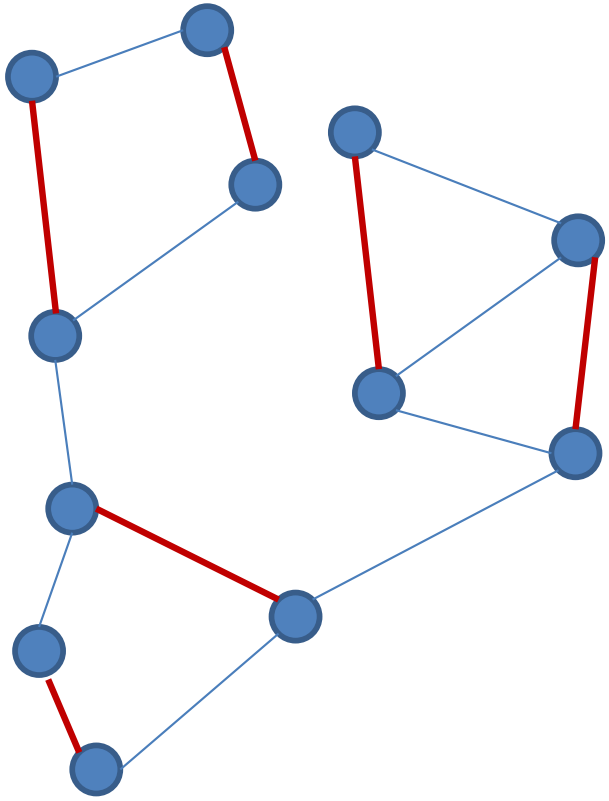


Key idea:
Add edge whenever it does not conflict with an existing edge.

Matching

Greedy Match

M : matching, initially empty
for each edge e = (u,v) in stream:
    if u and v are not matched in M
        add (u,v) to M.
return M

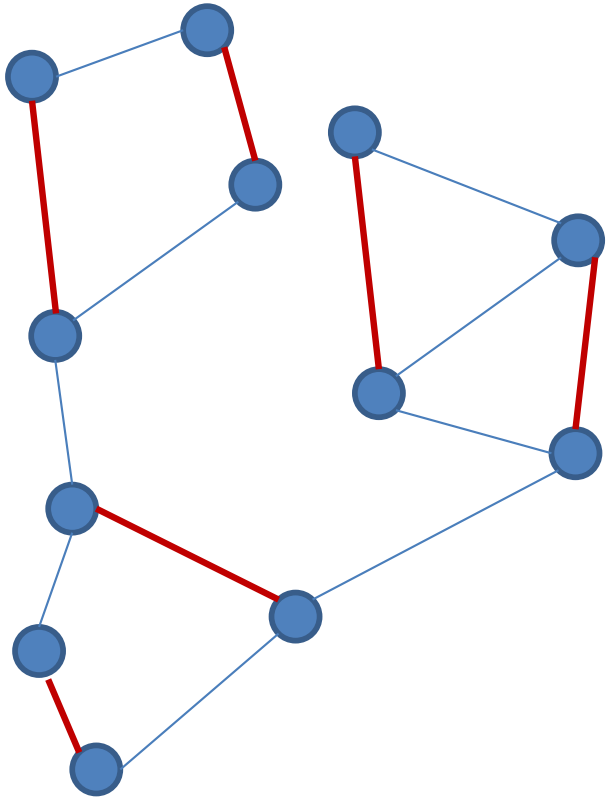


Claim:
M is a legal matching.

Matching

Greedy Match

M : matching, initially empty
for each edge e = (u,v) in stream:
    if u and v are not matched in M
        add (u,v) to M.
return M



Let M* be a maximum matching.

Claim:

$|M^*| \leq 2|M|$ ➔ 2-approximation

Matching 2-approximation

Proof:

Charging argument:

Matching 2-approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

Matching 2-approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

1. If e is in M (our matching), charge 1 to e.

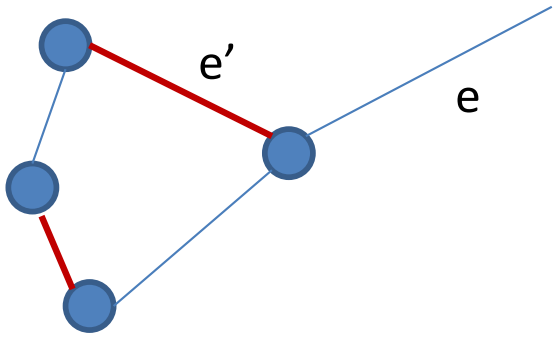
Matching 2-approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

1. If e is in M (our matching), charge 1 to e.
2. Otherwise, there exists an edge e' in M (our matching) adjacent to e.



(If not, we would have added e to M.)

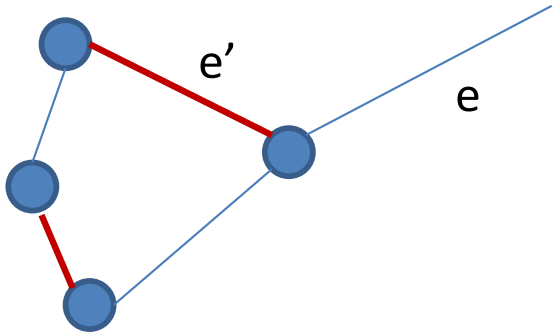
Matching 2-approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

1. If e is in M (our matching), charge 1 to e.
2. Otherwise, there exists an edge e' in M (our matching) adjacent to e.
   Charge 1 to e'.



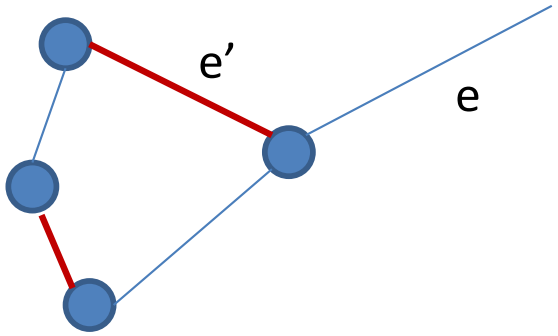
Matching 2-approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

1. If e is in M (our matching), charge 1 to e.
2. Otherwise, there exists an edge e' in M (our matching) adjacent to e.
   Charge 1 to e'.



Total charges: |M*|

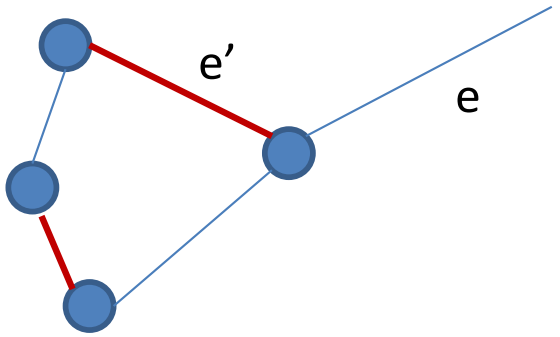
Matching 2-approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

1. If e is in M (our matching), charge 1 to e.
2. Otherwise, there exists an edge e' in M (our matching) adjacent to e.
   Charge 1 to e'.



Claim: only edges in M are charged.

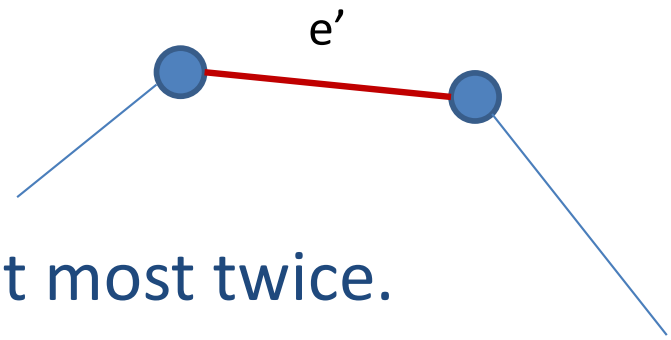
Matching 2-approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

1. If e is in M (our matching), charge 1 to e.
2. Otherwise, there exists an edge e' in M (our matching) adjacent to e.
   Charge 1 to e'.



Claim: each edge in M is charged at most twice.

(Either case (1) holds once or case (2) holds at most twice.)

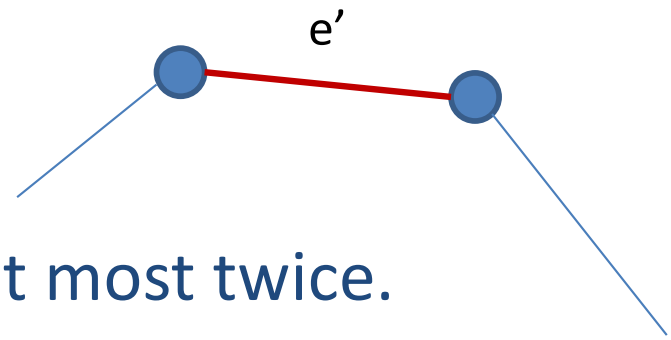
Matching 2-approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

1. If e is in M (our matching), charge 1 to e.
2. Otherwise, there exists an edge e' in M (our matching) adjacent to e.
   Charge 1 to e'.



Claim: each edge in M is charged at most twice.

Claim: total charge is $\leq 2|M|$.

Matching 2-approximation

Proof:

Charging argument:

Total charge =|M*|

Total charge is ≤ 2|M|

➔

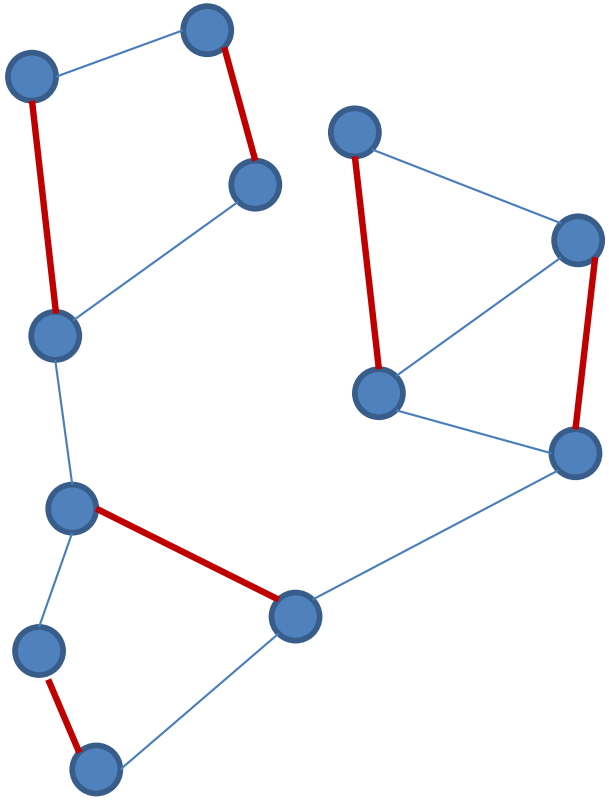
Theorem: 2-approximation

$$|M^*| \leq 2|M|$$

Matching

Greedy Match

M : matching, initially empty
for each edge e = (u,v) in stream:
    if u and v are not matched in M
        add (u,v) to M.
return M



Let M* be a maximum matching.

Claim:

$|M^*| \leq 2|M|$ ➔ 2-approximation

Weighted Matching

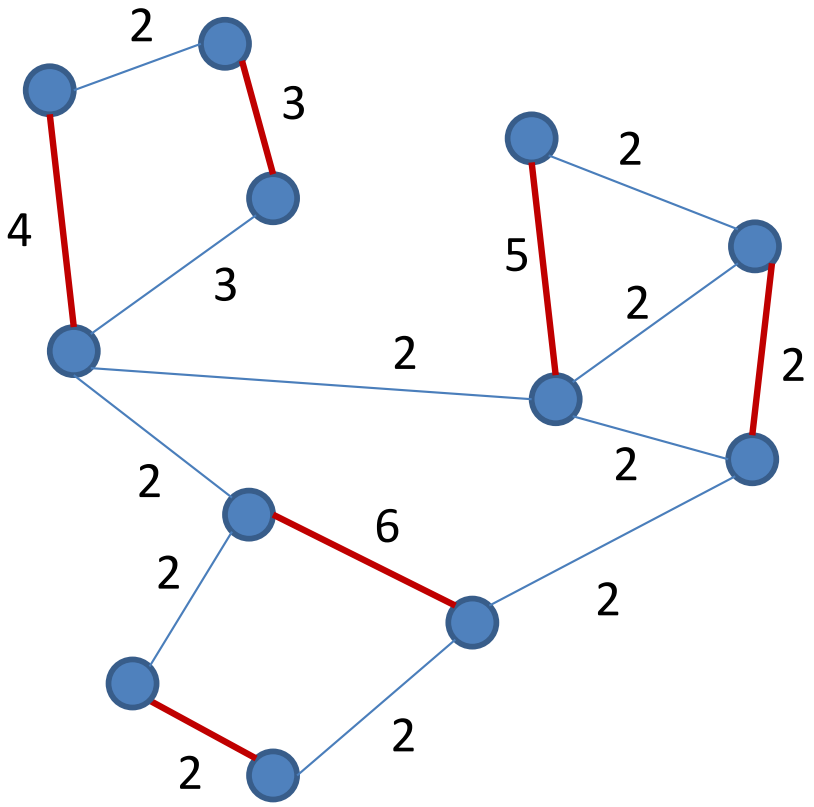
Assumptions:

Graph G = (V,E)

- Undirected
- n nodes
- m edges
- Goal: O(n log n) space.

Output:

Find a maximum weight matching.

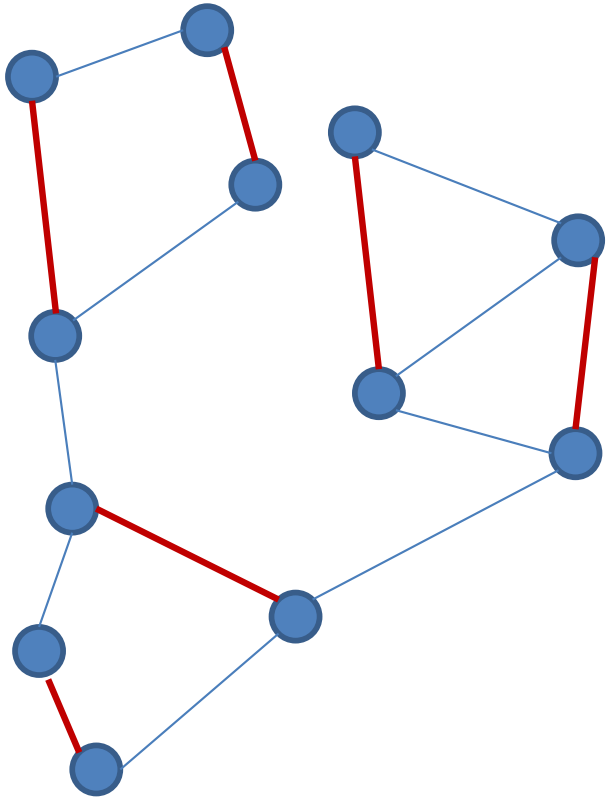


Example:
weight = 22

Matching

Greedy Match

M : matching, initially empty
for each edge e = (u,v) in stream:
    Let C be edges adjacent to u
    and v in M.
    if w(e) > w(C):
        remove C from M.
        add e to M.



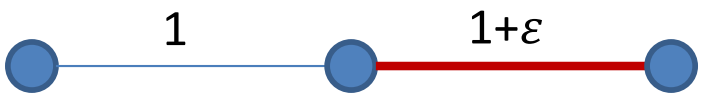
Does it work?

Matching

Greedy Match

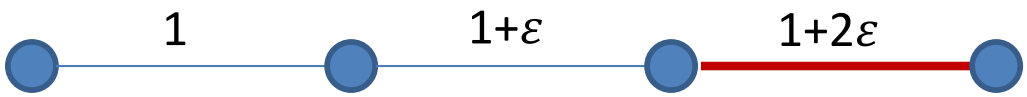
Matching

Greedy Match



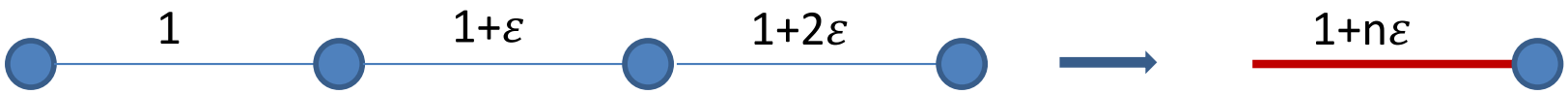
Matching

Greedy Match



Matching

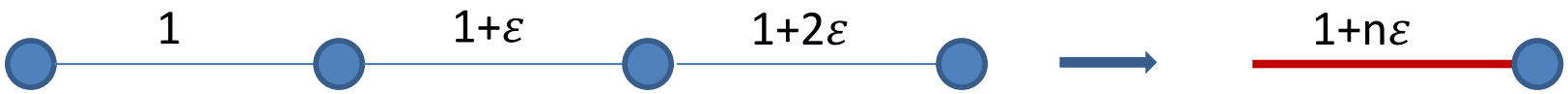
Greedy Match



Choose $\varepsilon = 1/n$.

Matching

Greedy Match

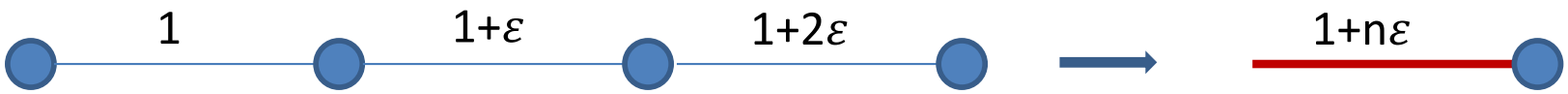


Choose $\varepsilon = 1/n$.

Your matching: weight = 2

Matching

Greedy Match



Choose $\varepsilon = 1/n$.

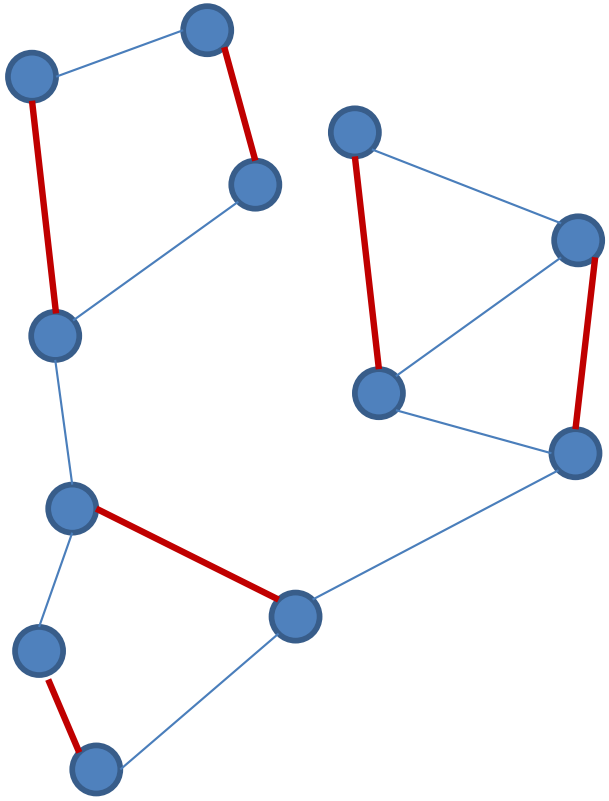
Your matching: weight = 2

OPT matching: weight > n/2

Matching

Less Greedy Match

M : matching, initially empty
for each edge e = (u,v) in stream:
    Let C be edges adjacent to u
    and v in M.
    if w(e) > (1+γ) w(C):
        remove C from M.
        add e to M.

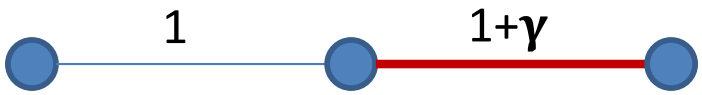


Matching

Less Greedy Match

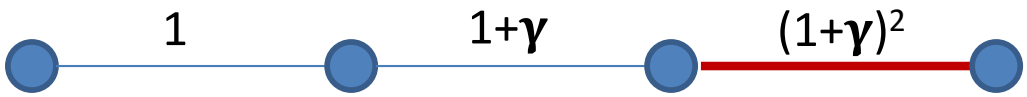
Matching

Less Greedy Match



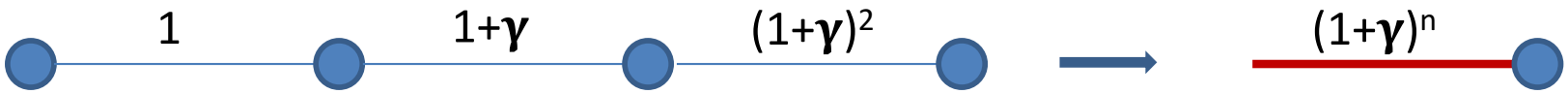
Matching

Less Greedy Match



Matching

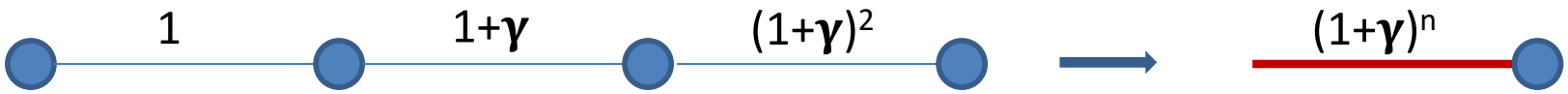
Less Greedy Match



Your matching: $(1+\gamma)^n$

Matching

Less Greedy Match



Your matching: $(1+\gamma)^n$

Best matching: $1 + (1+\gamma) + (1+\gamma)^2 + \ldots + (1+\gamma)^n \leq (1+\gamma)^{n+1}$

Approximation: (for this example only)

$$\frac{(1+\gamma)^{n+1}}{(1+\gamma)^n} \leq (1+\gamma)$$

Less Greedy Matching

Terminology:

Define:

- Edge e is born if/when added to M.
- Edge e is killed by e’ if e is removed when e’ is born.
- Edge e is a survivor if it is born and never killed.

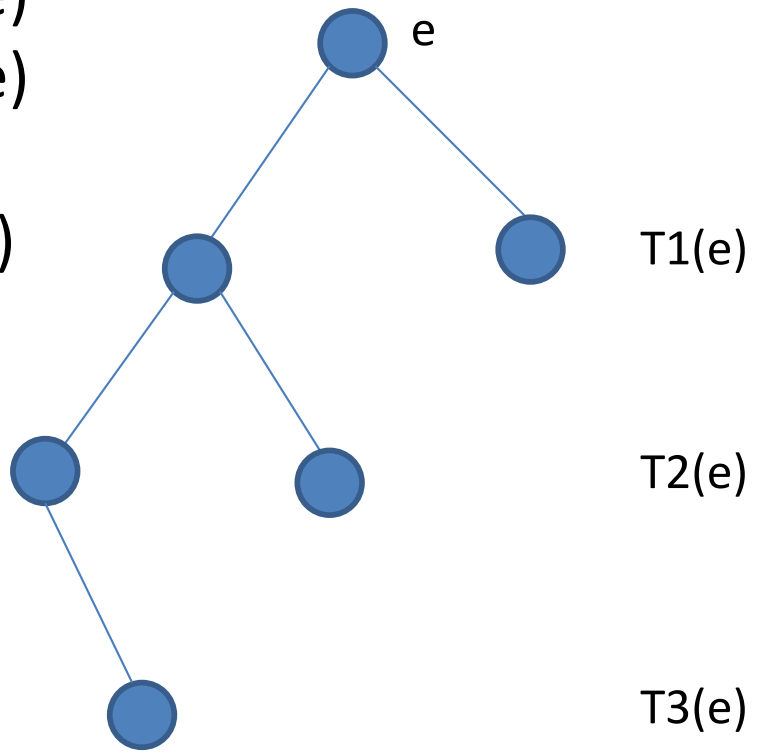
Less Greedy Matching

Tree of the Dead

If e is a survivor::

- $T_0(e) = \{e\}$
- $T_1(e)$ = edges killed by edges in $T_0(e)$
- $T_2(e)$ = edges killed by edges in $T_1(e)$

  …
- $T_j(e)$ = edges killed by edge in $T_{j-1}(e)$



Less Greedy Matching

Tree of the Dead

If e is a survivor::

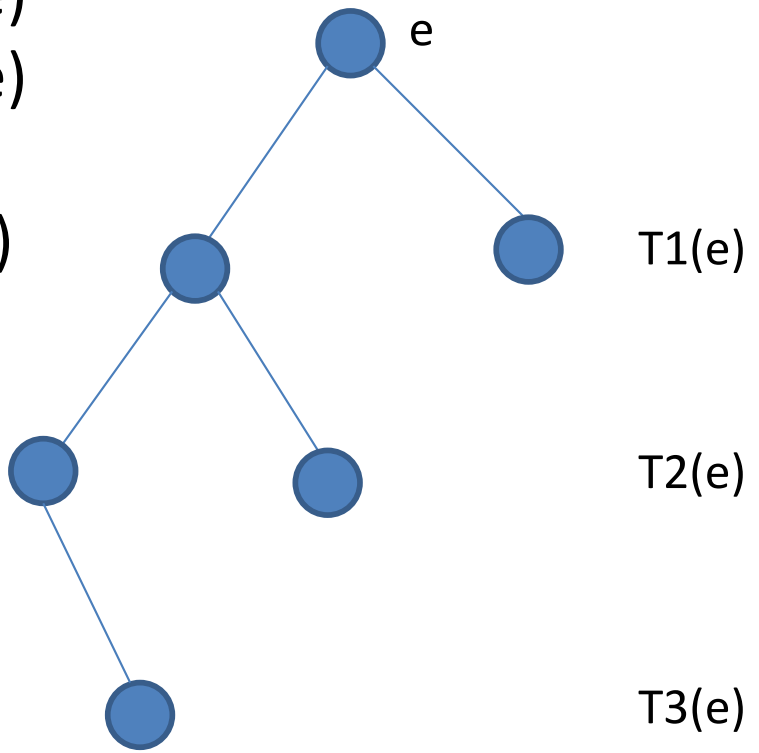
- $T_0(e) = \{e\}$
- $T_1(e)$ = edges killed by edges in $T_0(e)$
- $T_2(e)$ = edges killed by edges in $T_1(e)$

  …
- $T_j(e)$ = edges killed by edge in $T_{j-1}(e)$

Claim:

$$W(T_j(e)) > (1+\gamma)W(T_{j+1}(e))$$

(Because edges in j+1 were killed by edges in because they were smaller.)



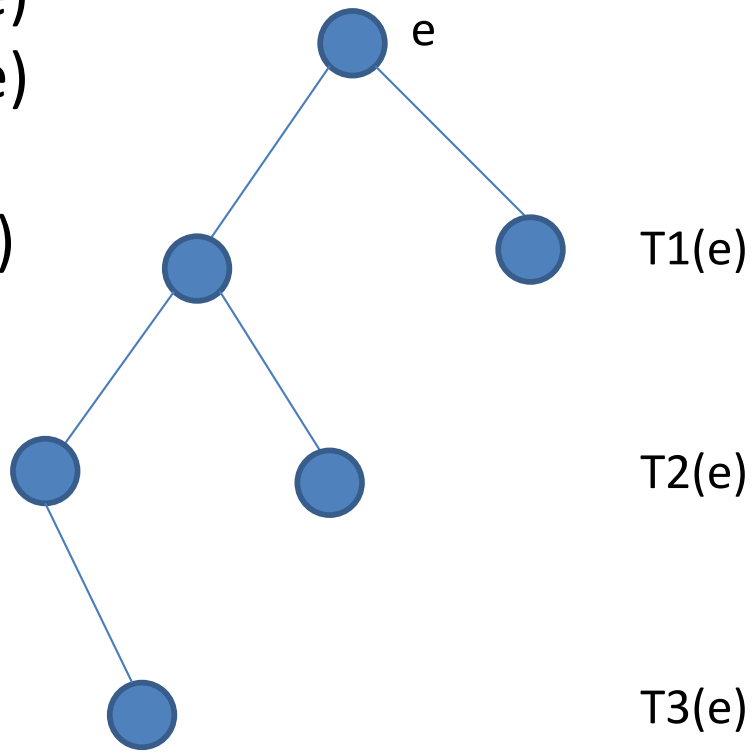
Less Greedy Matching

Tree of the Dead

If e is a survivor::

- $T_0(e) = \{e\}$
- $T_1(e)$ = edges killed by edges in $T_0(e)$
- $T_2(e)$ = edges killed by edges in $T_1(e)$

  …
- $T_j(e)$ = edges killed by edge in $T_{j-1}(e)$



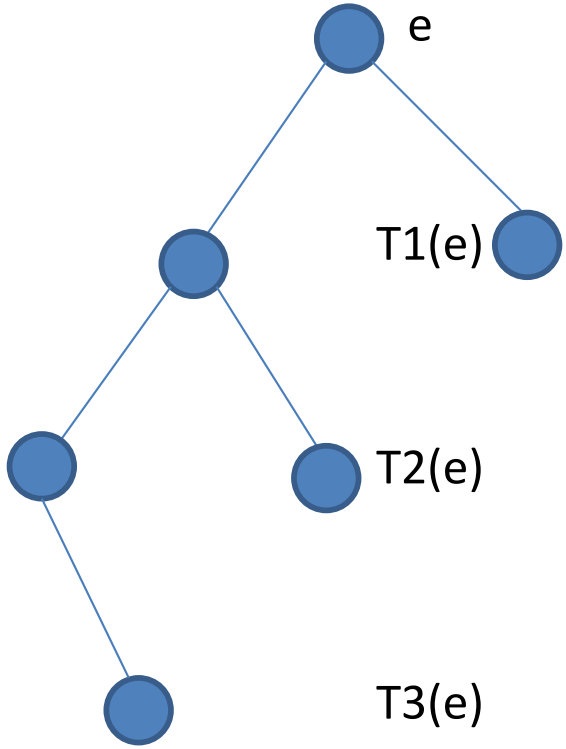
Calculate: weight of edges in the tree of the dead for e.

Less Greedy Matching

Tree of the Dead

$$W(T_j(e)) > (1+\gamma)W(T_{j+1}(e))$$

$$(1+\gamma)W(T(e)) \quad = \quad (1+\gamma)\sum_{j=1}^{\infty} W(T_j(e))$$

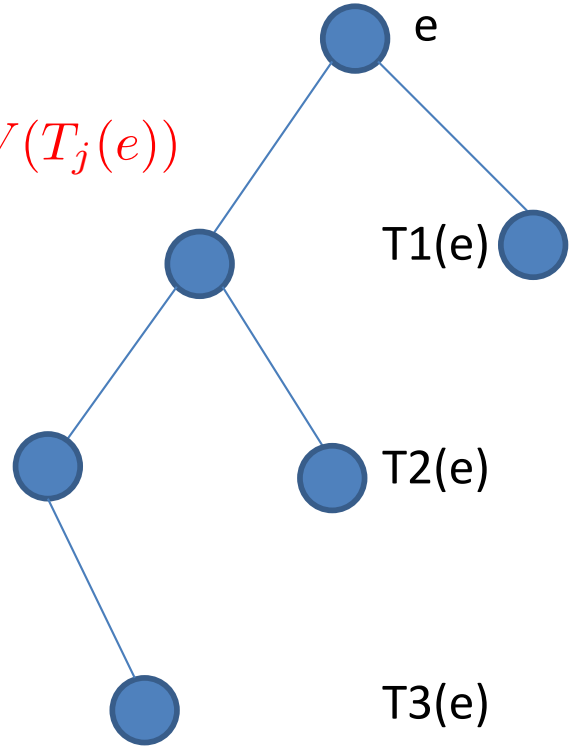


Less Greedy Matching

Tree of the Dead

$$W(T_j(e)) > (1+\gamma)W(T_{j+1}(e))$$

$$\begin{aligned}(1+\gamma)W(T(e)) &= (1+\gamma)\sum_{j=1}^{\infty} W(T_j(e)) \\ &< (1+\gamma)w(T_1(e)) + (1+\gamma)\sum_{} W(T_j(e))\end{aligned}$$

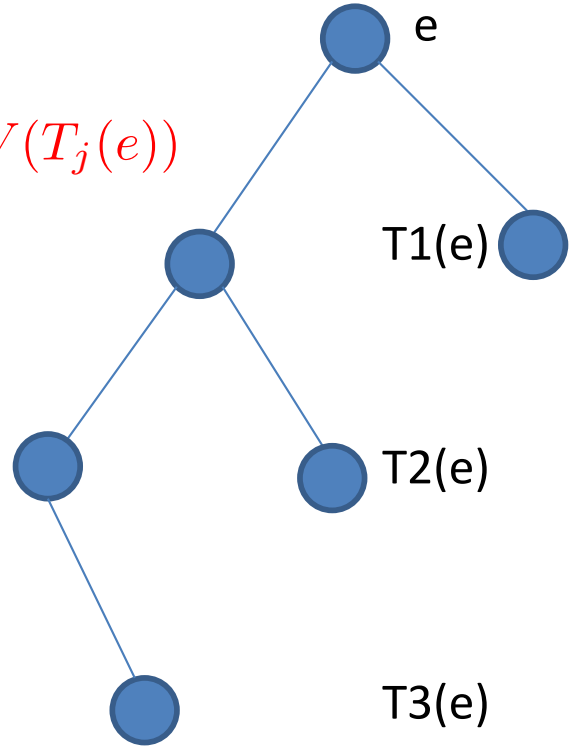


Less Greedy Matching

Tree of the Dead

$$W(T_j(e)) > (1+\gamma)W(T_{j+1}(e))$$

$$\begin{aligned}(1+\gamma)W(T(e)) &= (1+\gamma)\sum_{j=1}^{\infty} W(T_j(e)) \\ &< (1+\gamma)w(T_1(e)) + (1+\gamma)\sum_{j=2}^{\infty} W(T_j(e)) \\ &\leq w(e) + (1+\gamma)\sum_{j=2}^{\infty} W(T_j(e))\end{aligned}$$

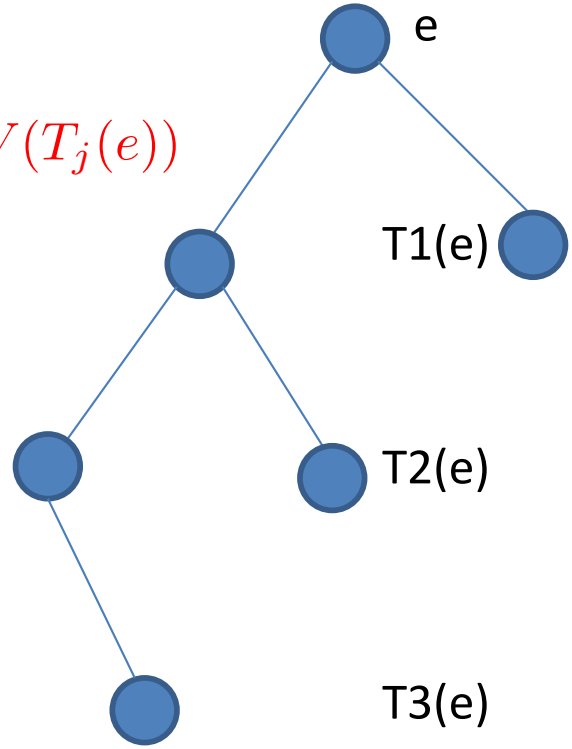


Less Greedy Matching

Tree of the Dead

$$W(T_j(e)) > (1+\gamma)W(T_{j+1}(e))$$

$$\begin{aligned}
(1+\gamma)W(T(e)) &= (1+\gamma)\sum_{j=1}^{\infty} W(T_j(e)) \\
&< (1+\gamma)w(T_1(e)) + (1+\gamma)\sum_{j=2}^{\infty} W(T_j(e)) \\
&\leq w(e) + (1+\gamma)\sum_{j=2}^{\infty} W(T_j(e)) \\
&\leq w(e) + \sum_{j=1}^{\infty} W(T_j(e))
\end{aligned}$$

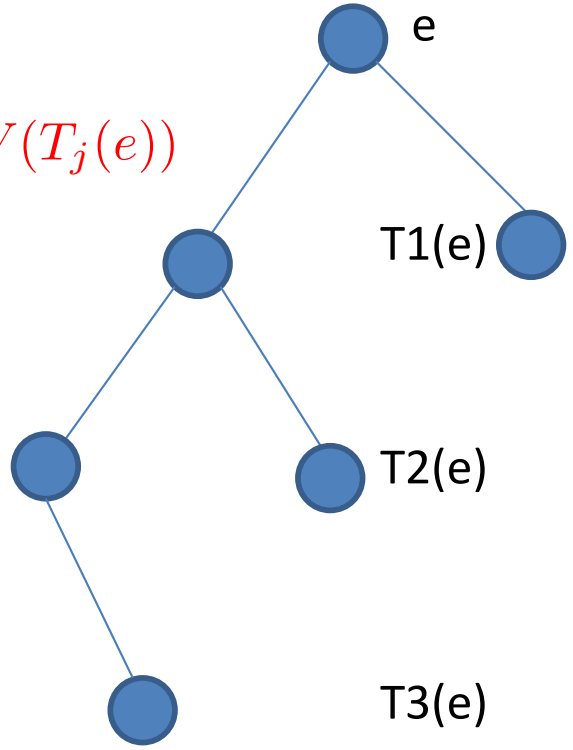


Less Greedy Matching

Tree of the Dead

$$W(T_j(e)) > (1+\gamma)W(T_{j+1}(e))$$

$$\begin{aligned}
(1+\gamma)W(T(e)) &= (1+\gamma)\sum_{j=1}^{\infty} W(T_j(e)) \\
&< (1+\gamma)w(T_1(e)) + (1+\gamma)\sum_{j=2}^{\infty} W(T_j(e)) \\
&\leq w(e) + (1+\gamma)\sum_{j=2}^{\infty} W(T_j(e)) \\
&\leq w(e) + \sum_{j=1}^{\infty} W(T_j(e)) \\
&\leq w(e) + w(T(e))
\end{aligned}$$



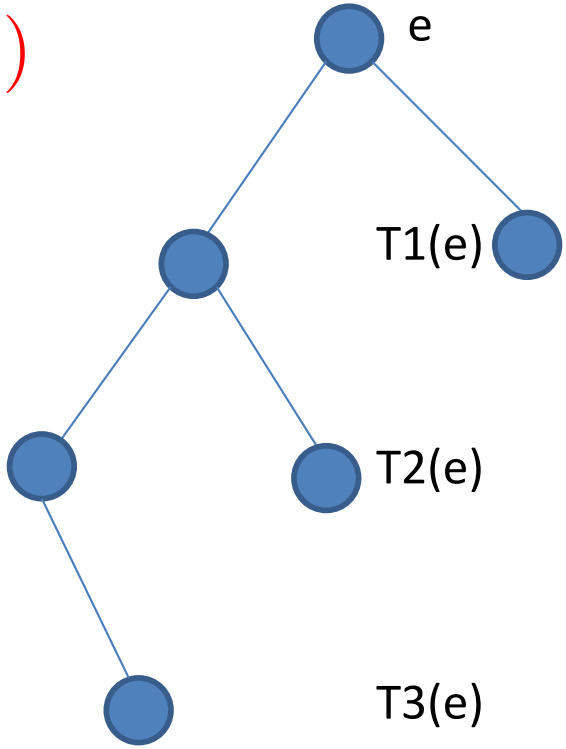
Less Greedy Matching

Tree of the Dead

$$W(T_j(e)) > (1+\gamma)W(T_{j+1}(e))$$

➔

$$(1+\gamma)W(T(e)) \quad < \quad w(e) + w(T(e))$$



Less Greedy Matching

Tree of the Dead

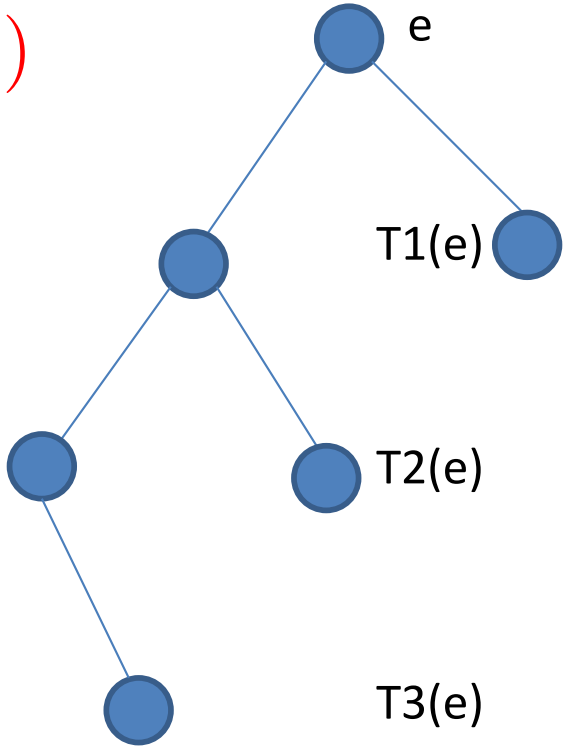
$$W(T_j(e)) > (1+\gamma)W(T_{j+1}(e))$$

➔

$$(1+\gamma)W(T(e)) \quad < \quad w(e) + w(T(e))$$

➔

$$\gamma W(T(e)) \quad < \quad w(e)$$



Less Greedy Matching

Tree of the Dead

$$W(T_j(e)) > (1+\gamma)W(T_{j+1}(e))$$

➔

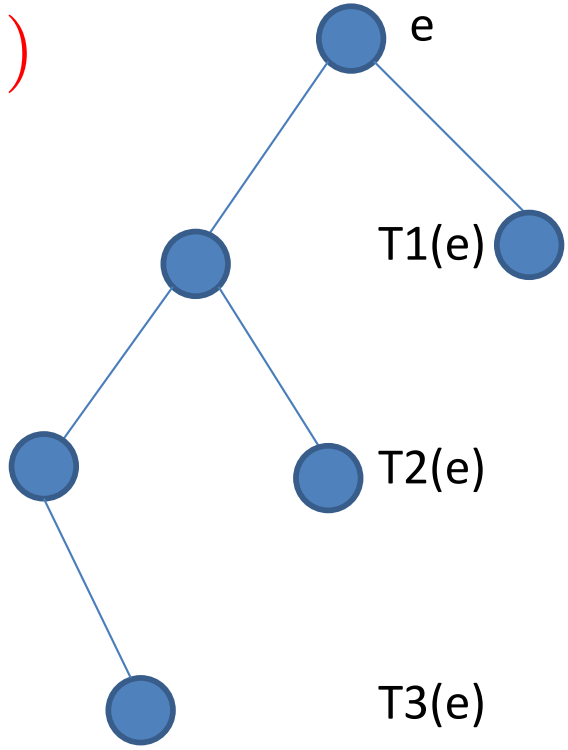
$$(1+\gamma)W(T(e)) \quad < \quad w(e) + w(T(e))$$

➔

$$\gamma W(T(e)) \quad < \quad w(e)$$

➔

$$W(T(e)) \quad < \quad w(e)/\gamma$$



Matching approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

Matching approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

If e is in T(e') for some survivor e' (or if e is a survivor):

charge w(e) to e.

Matching approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

If e is in T(e') for some survivor e' (or if e is a survivor):
charge w(e) to e.

Else: e was never born!

Why wasn't e born? Some set C of neighbors was too big.

Matching approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

If e was never born: $w(e) \leq (1+\gamma)w(C)$

Case (1) $C = \{e_1\}$

Charge w(e) to e_1.

Matching approximation

Proof:

Charging argument:

Let e = some edge in M* (optimal maximum matching).

If e was never born: $w(e) \leq (1+\gamma)w(C)$

Case (2) $C = \{e_1, e_2\}$

Charge: $\frac{w(e)w(e_1)}{w(e_1)+w(e_2)}$ to e_1.

Charge: $\frac{w(e)w(e_2)}{w(e_1)+w(e_2)}$ to e_2.

Matching approximation

Proof:

Why?

Case (2) C = {e_1, e_2}

Charge: $\frac{w(e)w(e_1)}{w(e_1)+w(e_2)}$ to e_1.

Charge: $\frac{w(e)w(e_2)}{w(e_1)+w(e_2)}$ to e_2.

Matching approximation

Proof:

Why? $w(e) \leq (1+\gamma)(w(e_1)+w(e_2)$

$$\frac{w(e)w(e_1)}{w(e_1)+w(e_2)} \leq (1+\gamma)(w(e_1)+w(e_2)\frac{w(e_1)}{w(e_1)+w(e_2)}$$
$$\leq (1+\gamma)w(e_1)$$

Case (2) C = {e_1, e_2}

Charge: $\frac{w(e)w(e_1)}{w(e_1)+w(e_2)}$ to e_1.

Charge: $\frac{w(e)w(e_2)}{w(e_1)+w(e_2)}$ to e_2.

Matching approximation

Proof:

Why? $w(e) \leq (1+\gamma)(w(e_1)+w(e_2)$

$$\frac{w(e)w(e_1)}{w(e_1)+w(e_2)} \leq (1+\gamma)(w(e_1)+w(e_2)\frac{w(e_1)}{w(e_1)+w(e_2)}$$
$$\leq (1+\gamma)w(e_1)$$

All charges to edge e are at most $(1+\gamma)w(e)$.

Matching approximation

Proof:

Total charges: $w(M^*)$

Matching approximation

Proof:

Total charges: w(M*)

Each edge is charged:

- Either for 2 unborn edges in M*.
- Or for 1 killed in M*.

Matching approximation

Proof:

Total charges: w(M*)

Each edge is charged:

- Either for 2 unborn edges in M*.
- Or for 1 killed in M*.

Can't be both: you only are charged for a killing when you are in M and killed!*

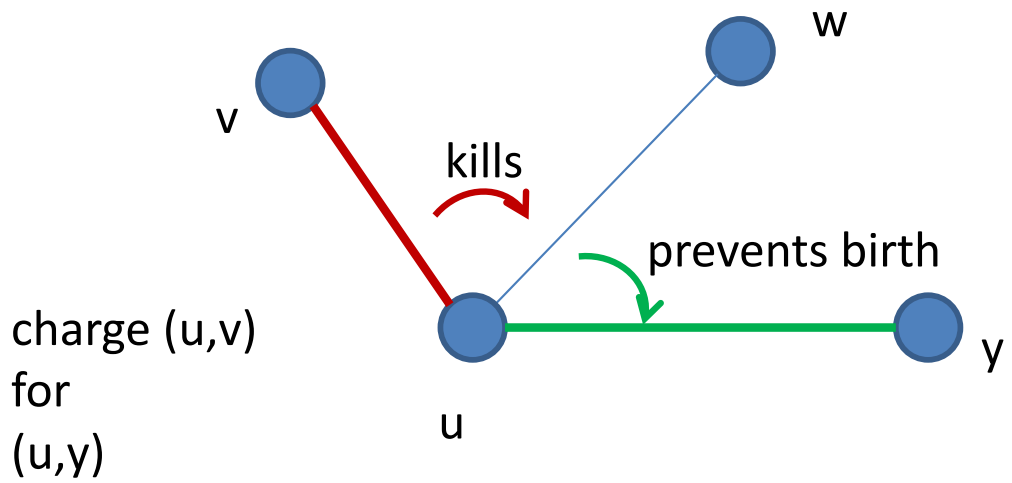
When you are in M, you have no neighbors in M* to prevent being born.*

Matching approximation

Proof:

Better: charge shifting

If e=(u,v) kills e'=(u,w) and e' is charged for unborn (u,y), then move charge for (u,y) to e=(u,v).



Matching approximation

Proof:

Total charges: $w(M^*)$

No edge more than 2 charges.

No killed edge (in tree of dead) has more than one charge.

Each edge is charged:

- Either for 2 unborn edges in M^*.
- Or for 1 killed in M^*.

Matching approximation

Math on board…

Total charges: $w(M^*)$

No edge more than 2 charges.

No killed edge (in tree of dead) has more than one charge.

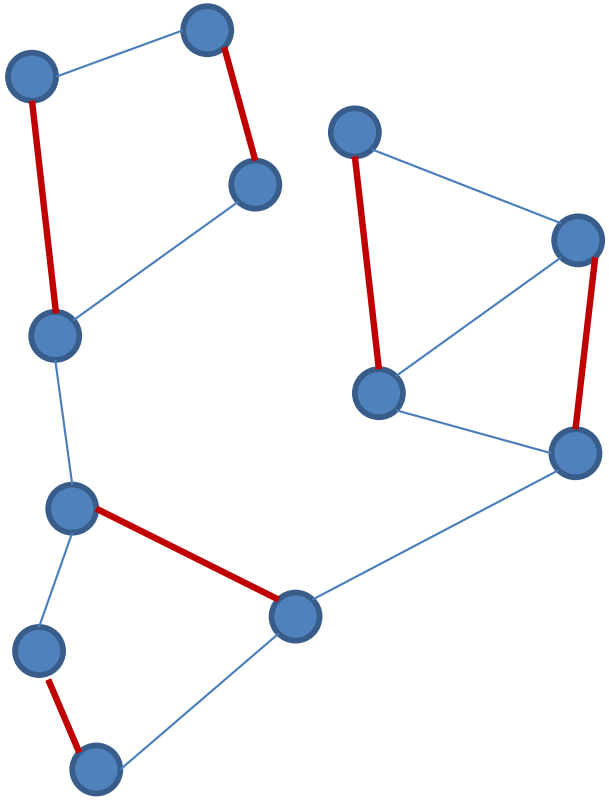
Each edge is charged:

- Either for 2 unborn edges in M^*.
- Or for 1 killed in M^*.

Matching

Less Greedy Match

M : matching, initially empty
for each edge e = (u,v) in stream:
    Let C be edges adjacent to u
    and v in M.
    if w(e) > (1+γ) w(C):
        remove C from M.
        add e to M.

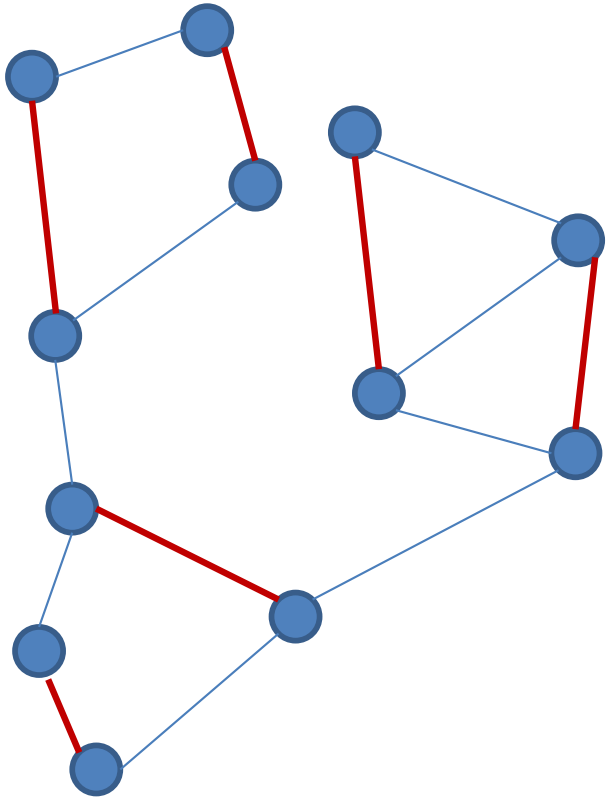


Claim: 6-approximation of optimal

Matching

Less Greedy Match

M : matching, initially empty
for each edge e = (u,v) in stream:
    Let C be edges adjacent to u
    and v in M.
    if w(e) > (1+γ) w(C):
        remove C from M.
        add e to M.



Better algorithm: (2+ε)-approximation of optimal

Summary

Last Week: Streaming

Misra-Gries:

- Item frequency
- Heavy Hitters

Flajolet-Martin:

- Number of distinct elements
- Median-of-means technique
- Chebychev+Chernoff

Problem Set:

- Alternate solution to item frequency and heavy hitters.

Today: Graph Streaming

Connectivity

- Is the graph connected?

Bipartite

- Is the graph bipartite?

MST

- Find a minimum spanning tree

Spanners

- Find approximate shortest paths

Matching

- Find an (approximate) maximum matching.