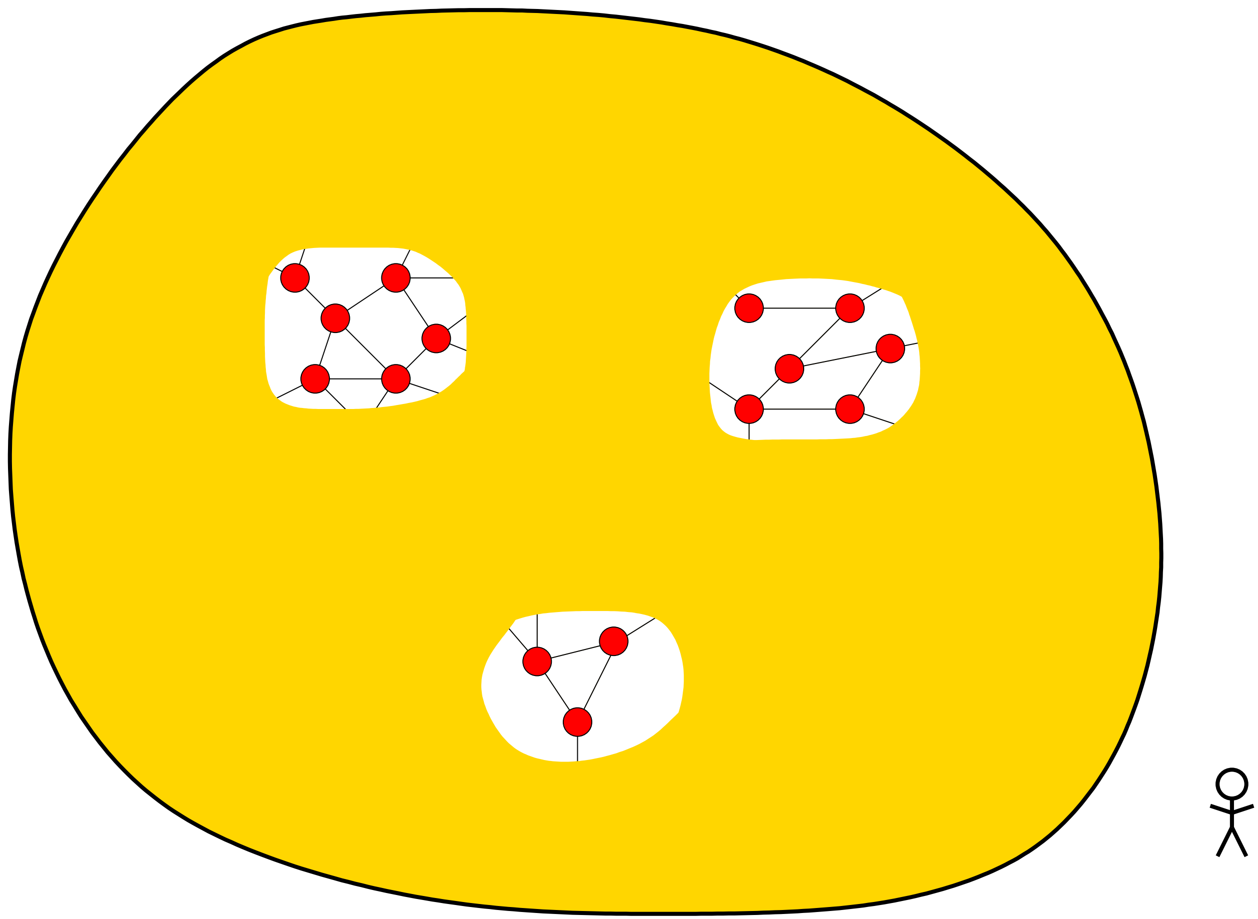
Sublinear Graph Approximation Algorithms

Krzysztof Onak

IBM Research

Sublinear-Time Algorithms

Sublinear-Time Algorithms

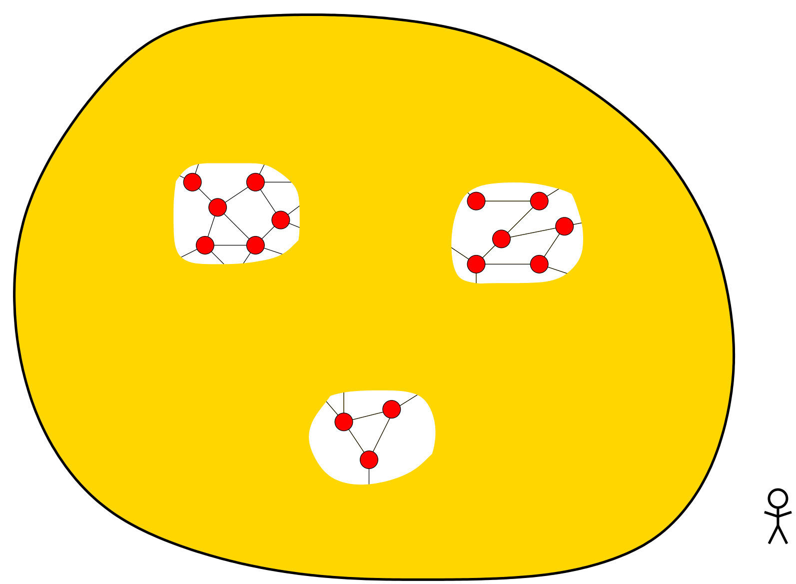


Sublinear-time algorithms:

Fast answer based on inspecting
a tiny fraction of the input

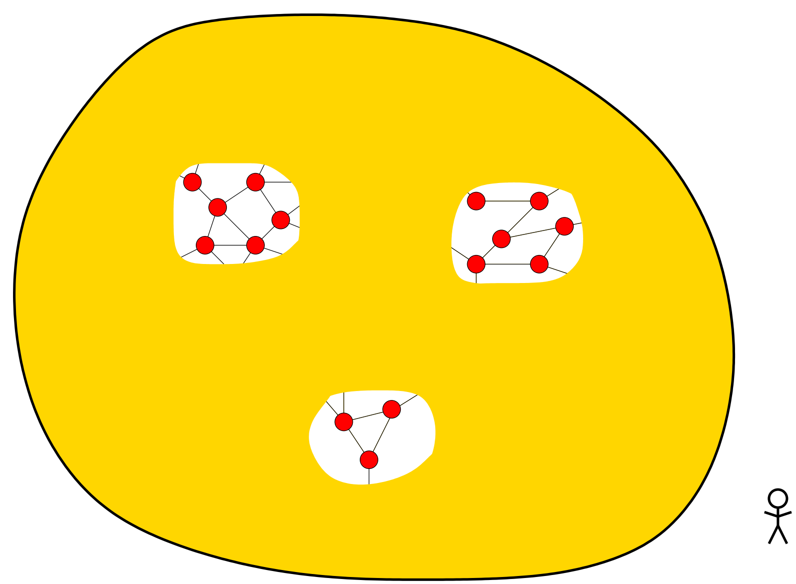
Focus: Parameters of Graphs

- Want to inspect only a small fraction of the graph and learn something about it



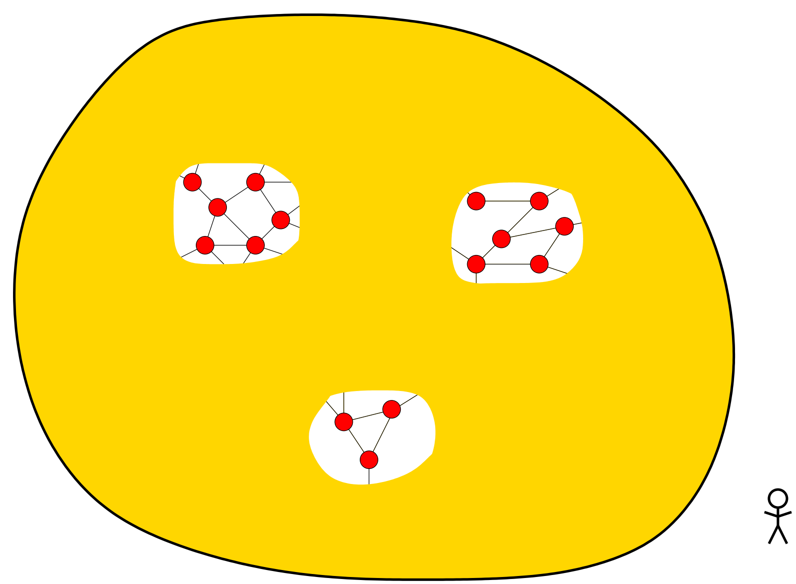
Focus: Parameters of Graphs

- Want to inspect only a small fraction of the graph and learn something about it
- Classical graph parameters:
  - the minimum vertex cover size
  - the maximum matching size
  - the independence number
  - the minimum dominating set size



Focus: Parameters of Graphs

- Want to inspect only a small fraction of the graph and learn something about it
- Classical graph parameters:
  - the minimum vertex cover size
  - the maximum matching size
  - the independence number
  - the minimum dominating set size
- Very fast algorithms!
- Much faster then computing a corresponding approximate solution



The Model

A not very important assumption:

- the maximum vertex degree $\leq d$
- easy to replace max degree with average degree

The Model

A not very important assumption:

- the maximum vertex degree $\leq d$
- easy to replace max degree with average degree

Input access:

- Can obtain a random vertex

The Model

A not very important assumption:

- the maximum vertex degree $\leq d$
- easy to replace max degree with average degree

Input access:

- Can obtain a random vertex
- Can query the degree $\deg(v)$ of a specific vertex v

The Model

A not very important assumption:

- the maximum vertex degree $\leq d$
- easy to replace max degree with average degree

Input access:

- Can obtain a random vertex
- Can query the degree $\deg(v)$ of a specific vertex v
- For each vertex v and each $i \in \{1, 2, \ldots, \deg(v)\}$, can obtain the i-th neighbor of v

The Model

A not very important assumption:

- the maximum vertex degree $\leq d$
- easy to replace max degree with average degree

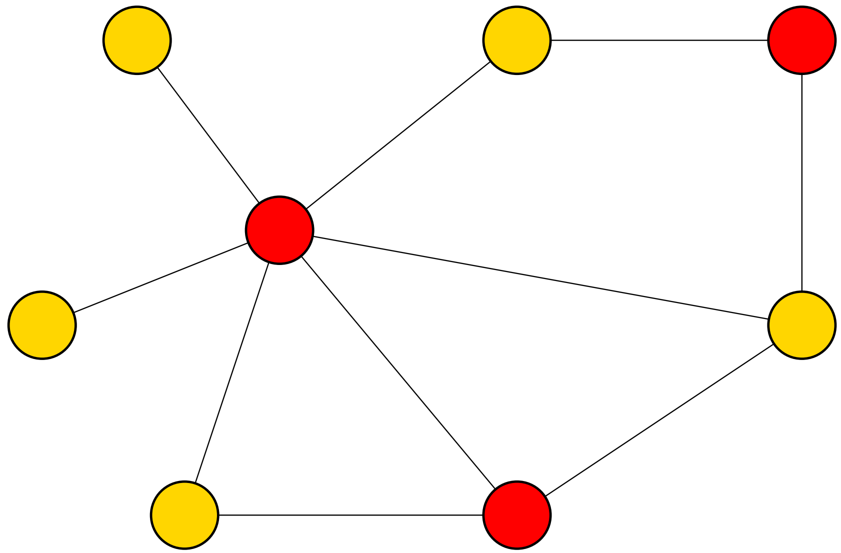
Input access:

- Can obtain a random vertex
- Can query the degree $\deg(v)$ of a specific vertex v
- For each vertex v and each $i \in \{1, 2, \ldots, \deg(v)\}$, can obtain the i-th neighbor of v

Essentially: query access to adjacency lists

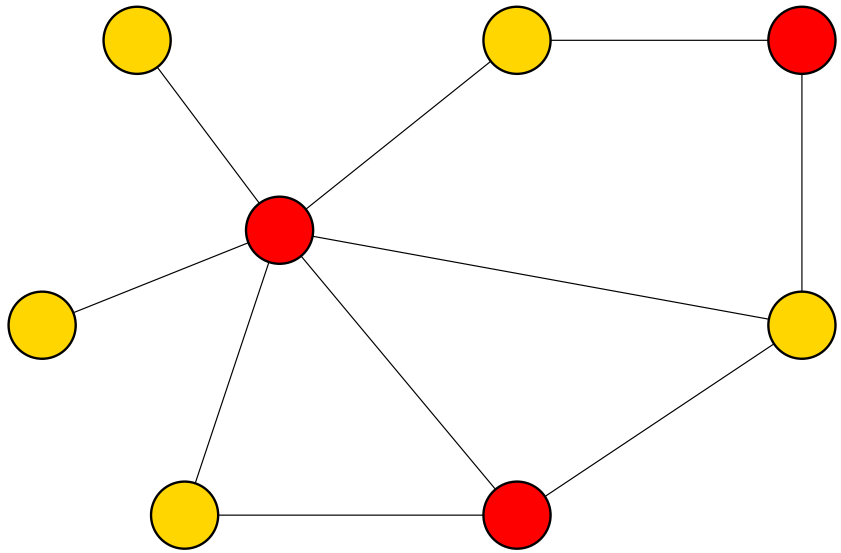
Example: Vertex Cover

Goal: find smallest set S of vertices such that each edge has endpoint in S



Example: Vertex Cover

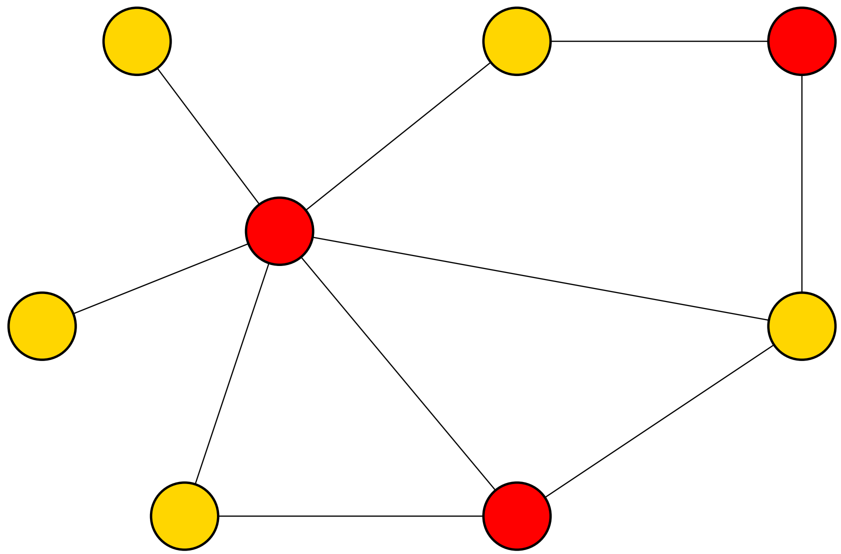
Goal: find smallest set S of vertices such that each edge has endpoint in S



- Best polynomial time algorithm: 2-approximation

Example: Vertex Cover

Goal: find smallest set S of vertices such that each edge has endpoint in S



- Best polynomial time algorithm: 2-approximation
- Here:

$$\mathrm{VC} - \epsilon n \leq \text{(computed value)} \leq 2 \cdot \mathrm{VC} + \epsilon n$$

where $\mathrm{VC} =$ minimum vertex cover size
$n =$ number of vertices

Essential Technique

- We develop a local computation method

Essential Technique

- We develop a local computation method
- Multiple applications:
  - vertex cover approximation
  - maximum matching approximation
  - computing nice partitions of graphs
  - local distributed algorithms
  - approximate planarity verification
  - local computation algorithms

Essential Technique

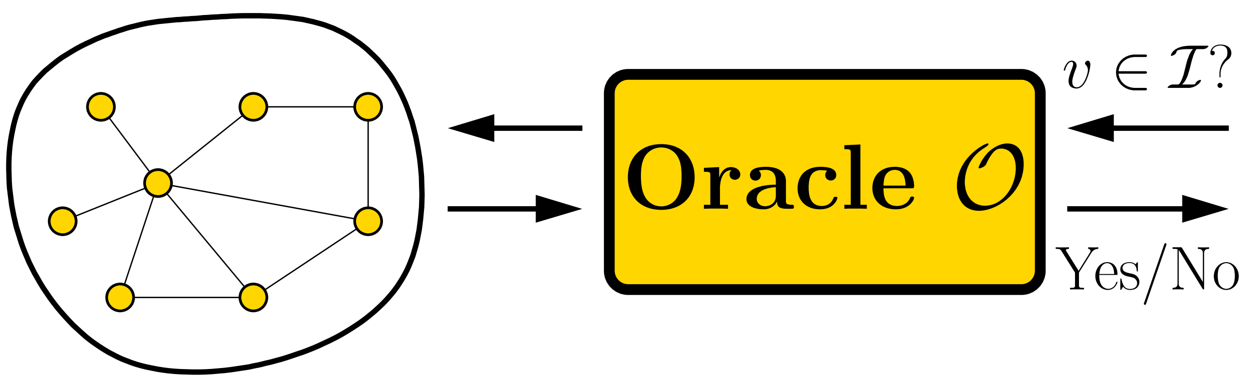
- We develop a local computation method
- Multiple applications:
  - vertex cover approximation
  - maximum matching approximation
  - computing nice partitions of graphs
  - local distributed algorithms
  - approximate planarity verification
  - local computation algorithms
- Will present and apply a less general version: local computation of maximal independent set

Main Tool:
Constructing a Maximal Independent Set Locally

Oracle for Maximal Independent Set

Want to construct oracle $\mathcal{O}$:

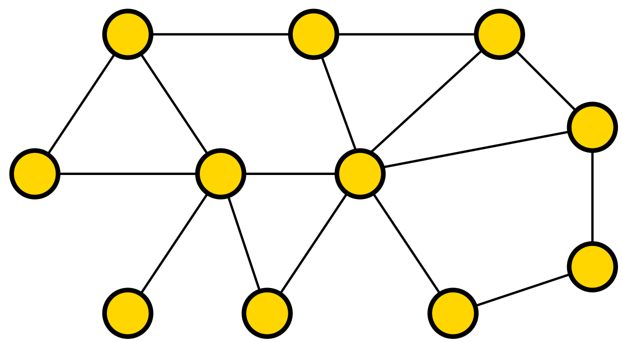
- $\mathcal{O}$ has query access to $G = (V, E)$
- $\mathcal{O}$ provides query access to maximal independent set $\mathcal{I} \subseteq V$
- $\mathcal{I}$ is not a function of queries
  it is a function of G and random bits



Goal: Minimize the query processing time

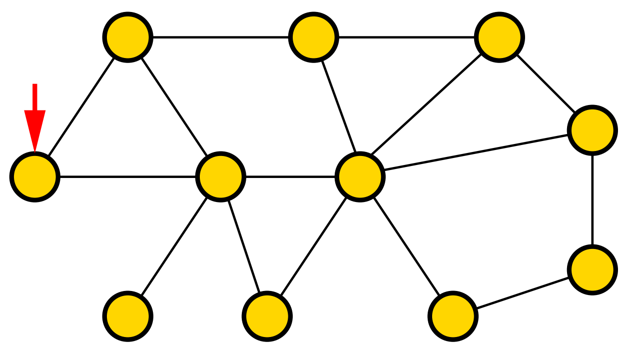
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



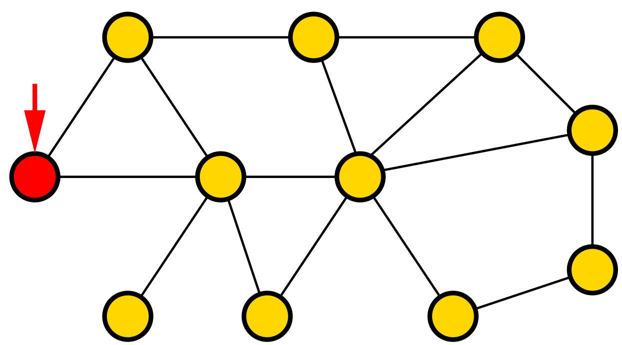
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



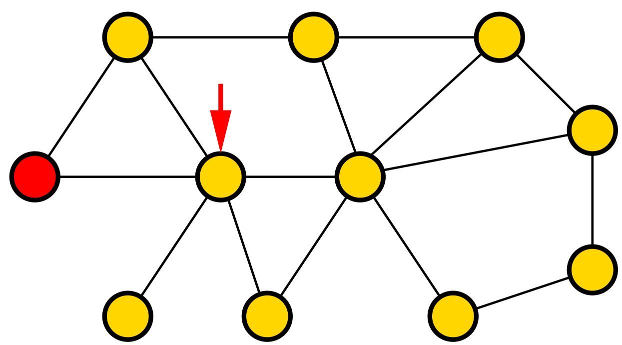
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



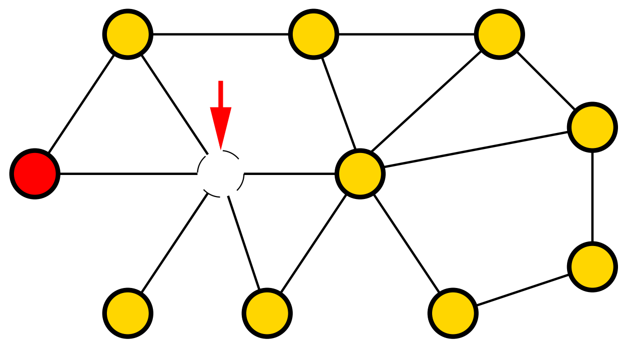
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



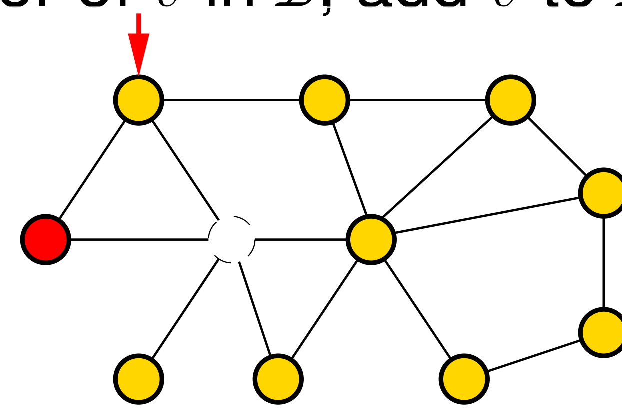
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



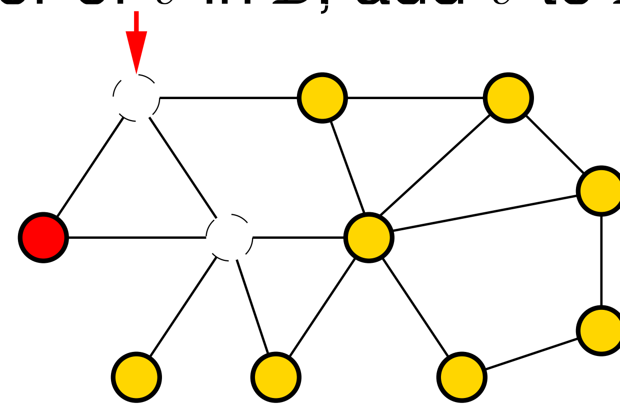
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



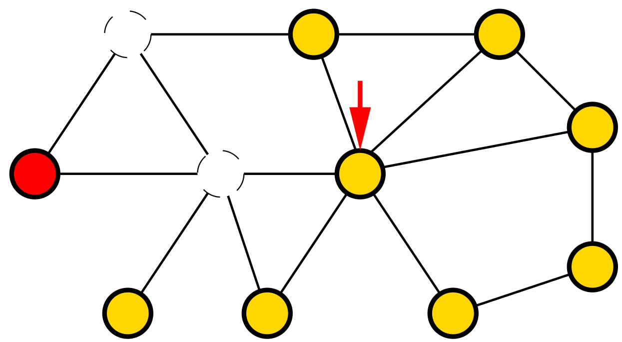
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



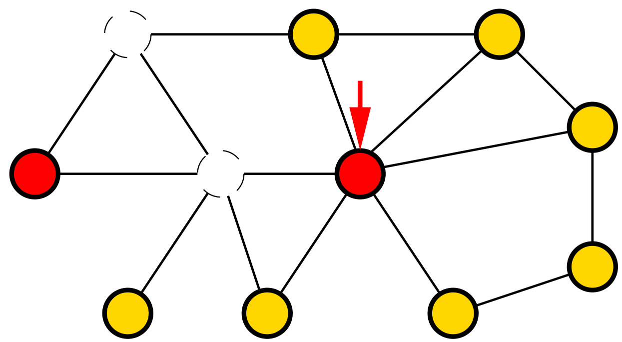
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



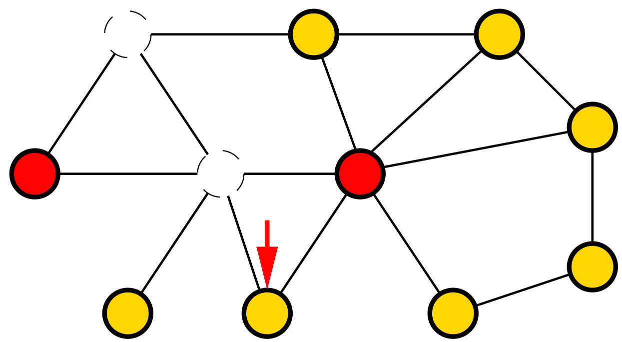
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



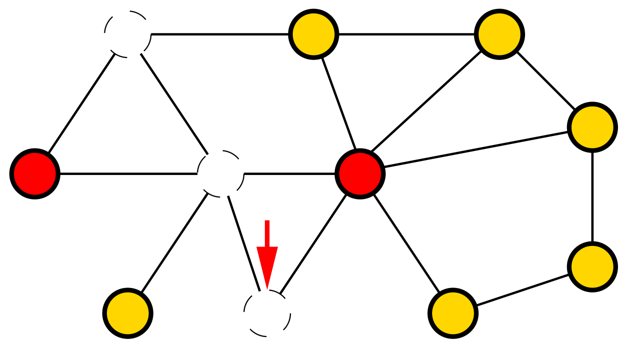
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



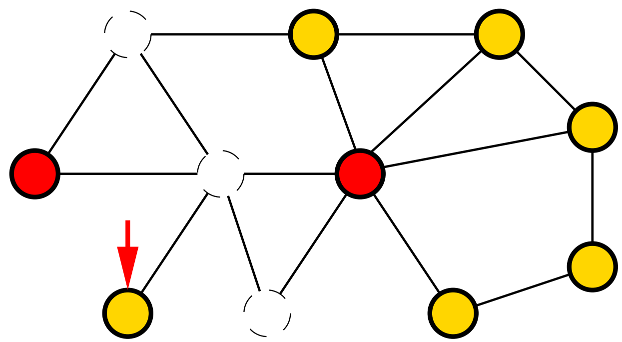
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



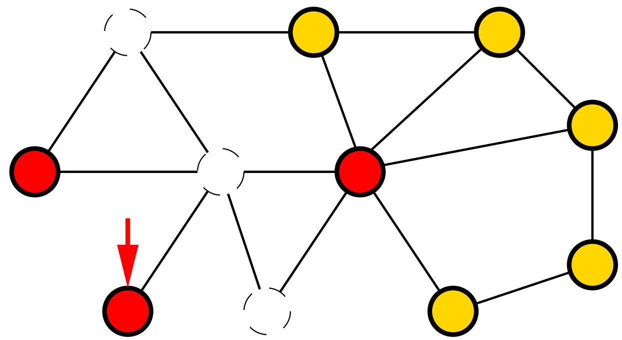
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



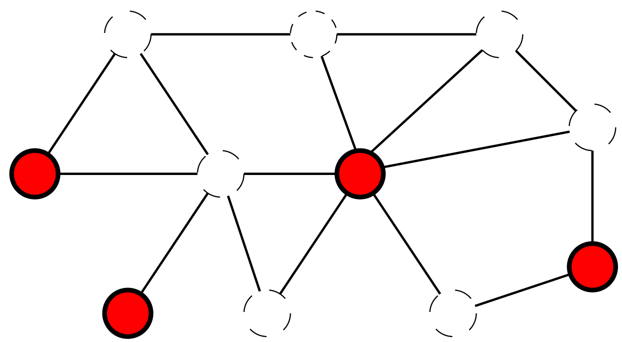
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



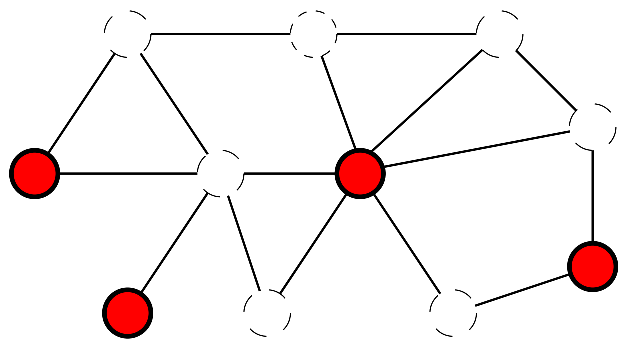
Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



Challenge of Locality

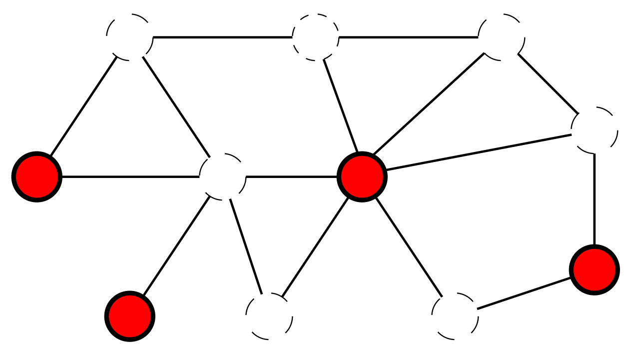
- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



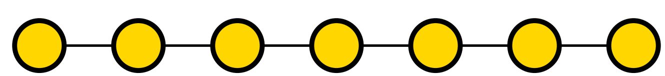
- Want to simulate this algorithm locally:
  Check what happened to earlier neighbors

Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$

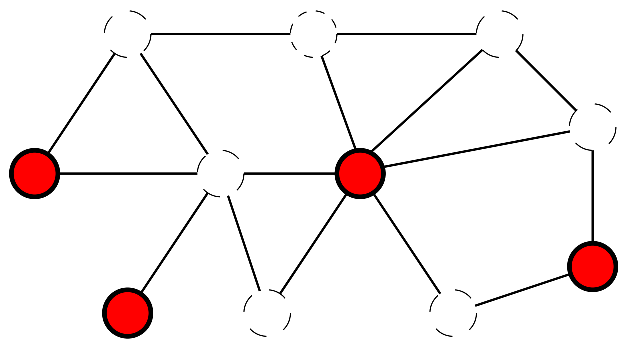


- Want to simulate this algorithm locally:
  Check what happened to earlier neighbors
- Problem: long chains of dependencies

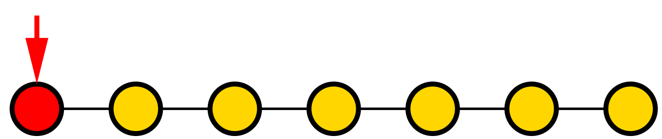


Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$

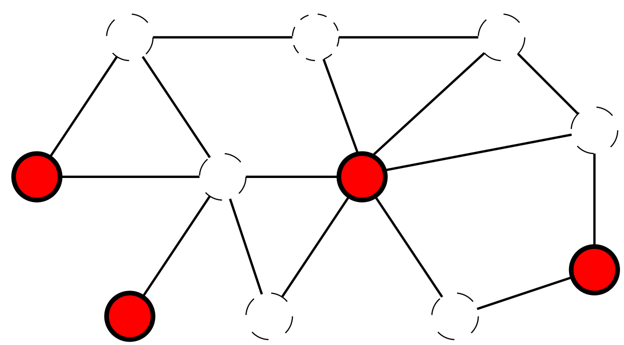


- Want to simulate this algorithm locally:
  Check what happened to earlier neighbors
- Problem: long chains of dependencies

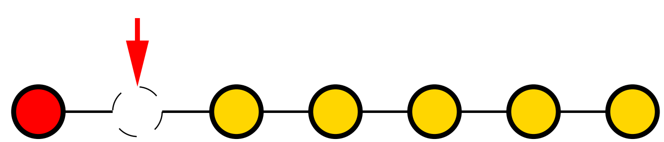


Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$

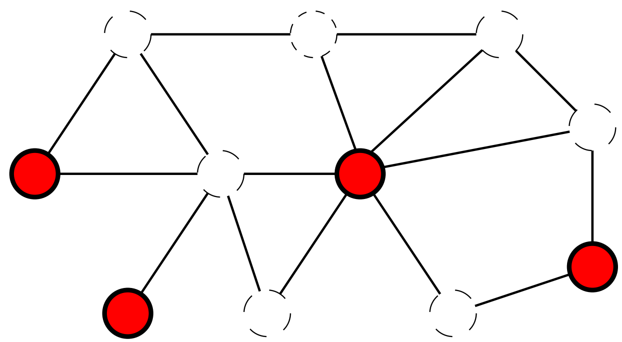


- Want to simulate this algorithm locally:
  Check what happened to earlier neighbors
- Problem: long chains of dependencies

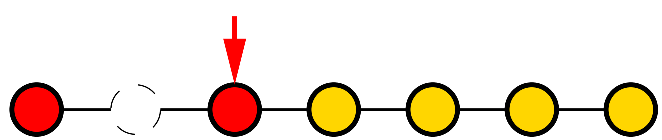


Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$

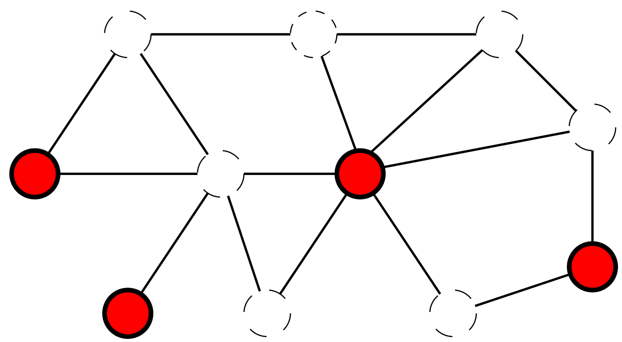


- Want to simulate this algorithm locally:
  Check what happened to earlier neighbors
- Problem: long chains of dependencies

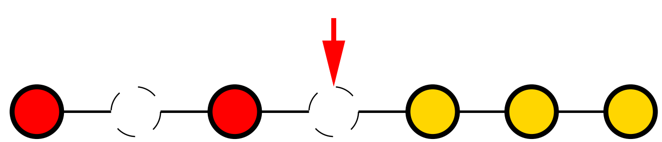


Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$

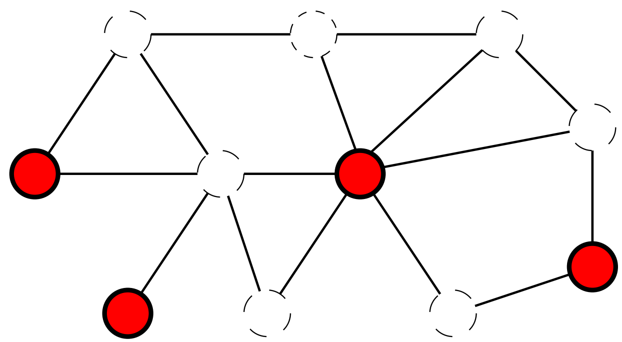


- Want to simulate this algorithm locally:
  Check what happened to earlier neighbors
- Problem: long chains of dependencies

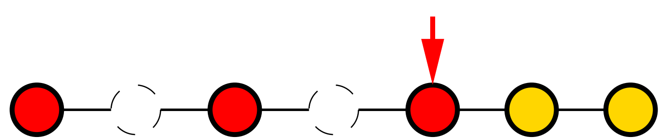


Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$

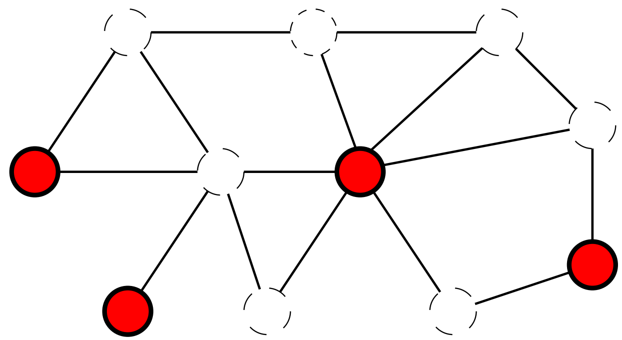


- Want to simulate this algorithm locally:
  Check what happened to earlier neighbors
- Problem: long chains of dependencies

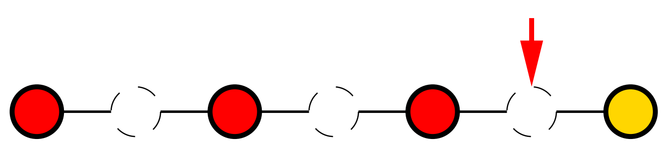


Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$

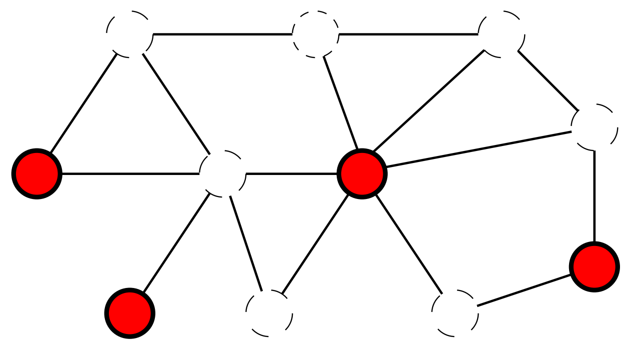


- Want to simulate this algorithm locally:
  Check what happened to earlier neighbors
- Problem: long chains of dependencies

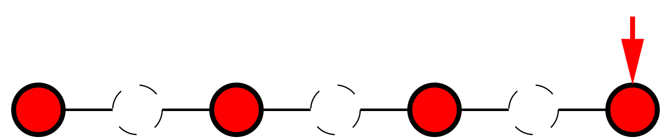


Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$

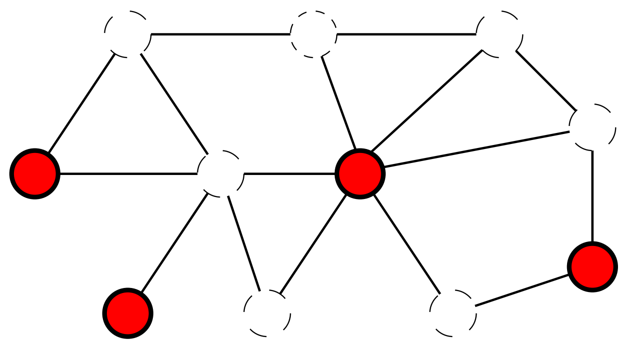


- Want to simulate this algorithm locally:
  Check what happened to earlier neighbors
- Problem: long chains of dependencies

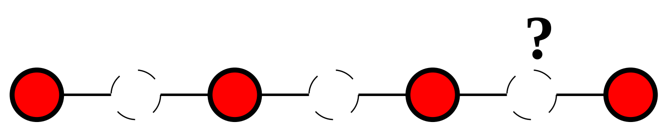


Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$

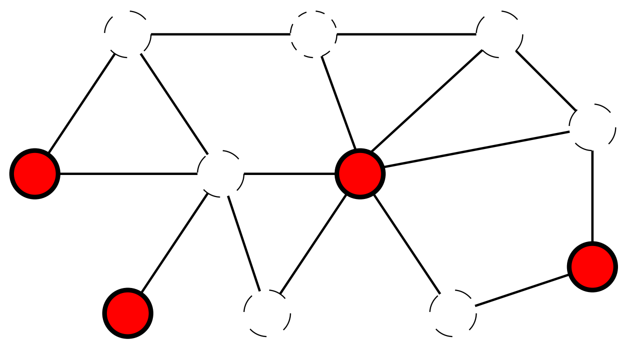


- Want to simulate this algorithm locally:
  Check what happened to earlier neighbors
- Problem: long chains of dependencies

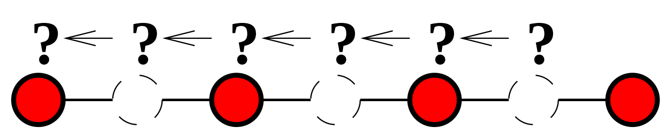


Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$

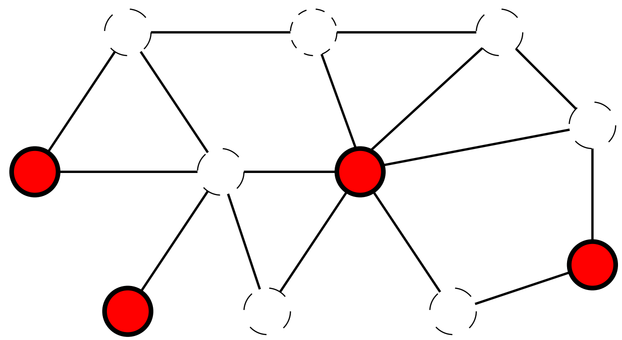


- Want to simulate this algorithm locally:
  Check what happened to earlier neighbors
- Problem: long chains of dependencies

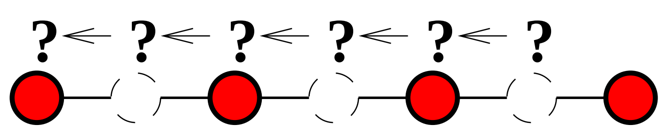


Challenge of Locality

- Simplest algorithm for Maximal Independent Set $\mathcal{I}$:
  - Start with $\mathcal{I} = \emptyset$
  - Consider vertices v one by one:
    - If no neighbor of v in $\mathcal{I}$, add v to $\mathcal{I}$



- Want to simulate this algorithm locally:
  Check what happened to earlier neighbors
- Problem: long chains of dependencies



- Our solution: consider vertices in random order

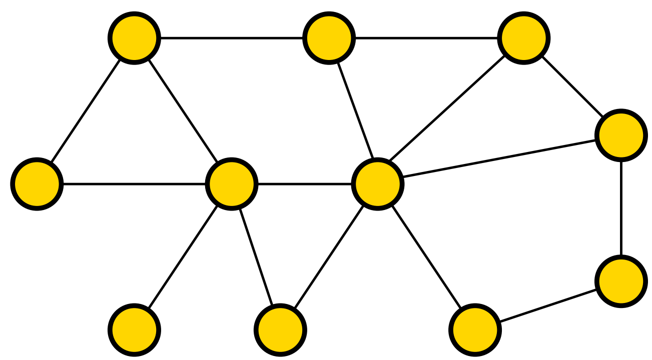
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



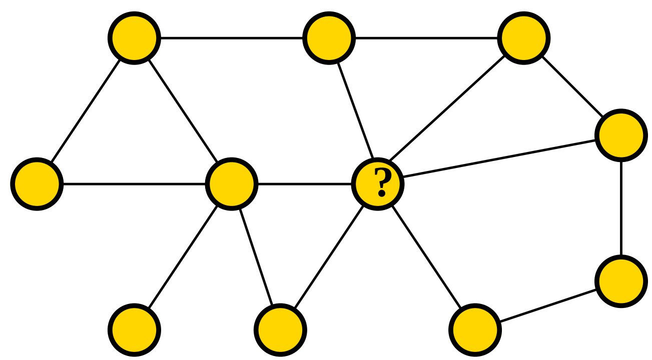
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

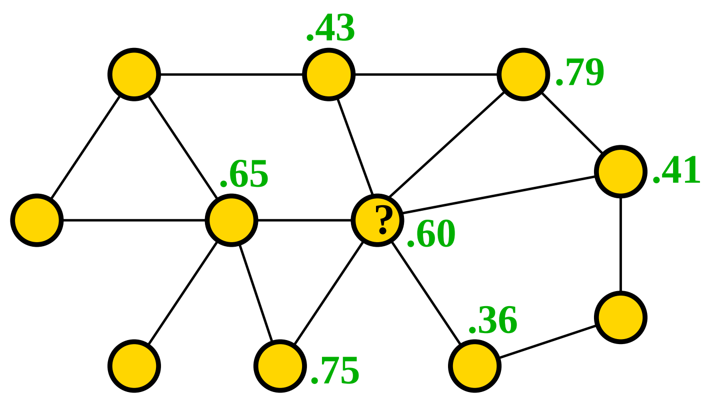
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

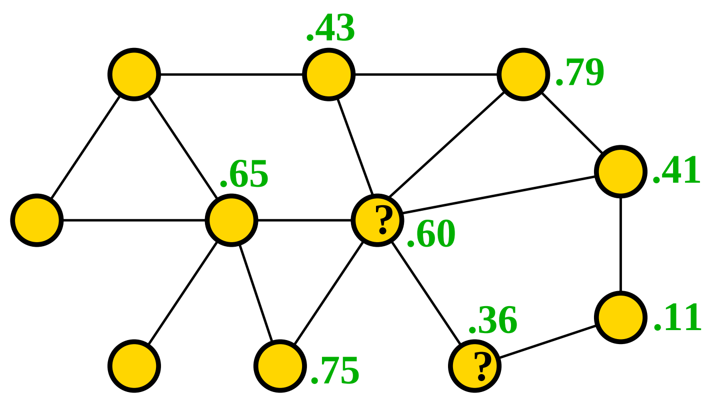
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

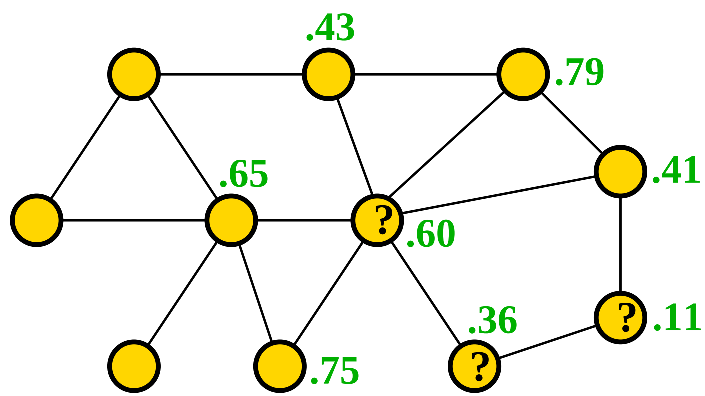
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

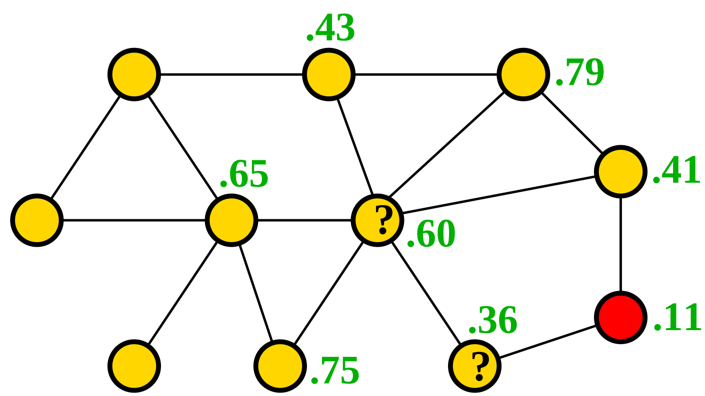
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

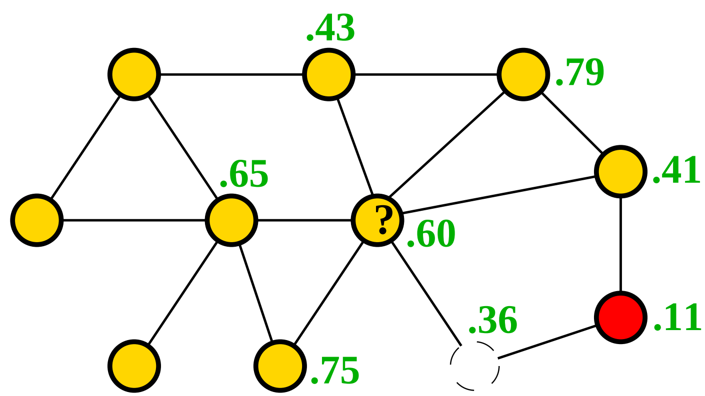
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

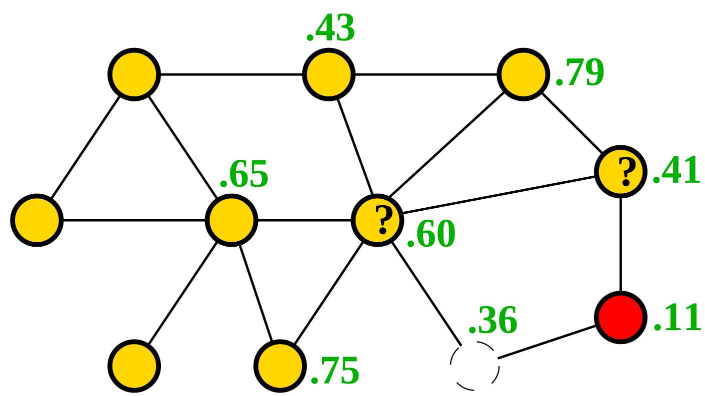
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

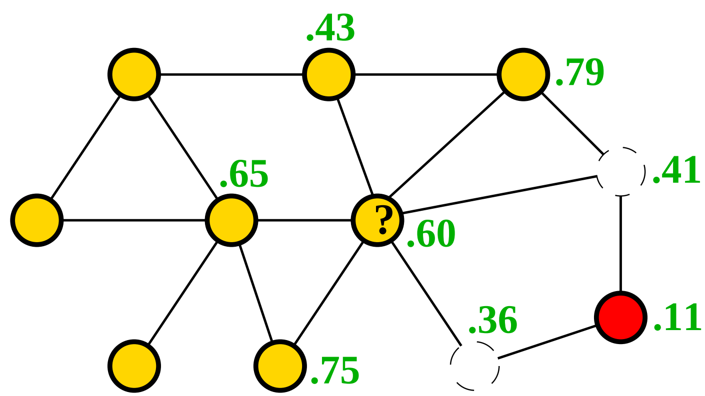
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

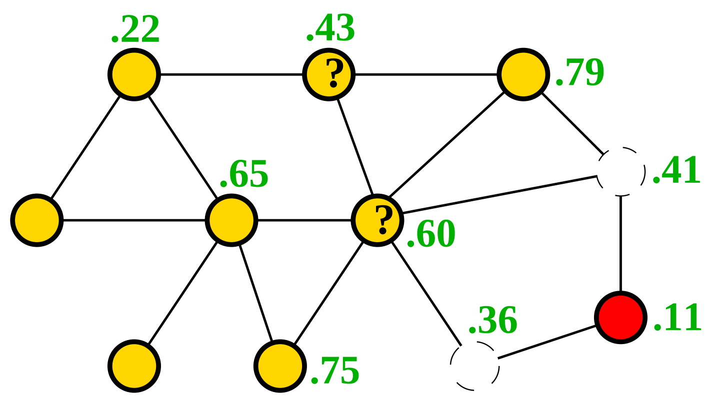
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

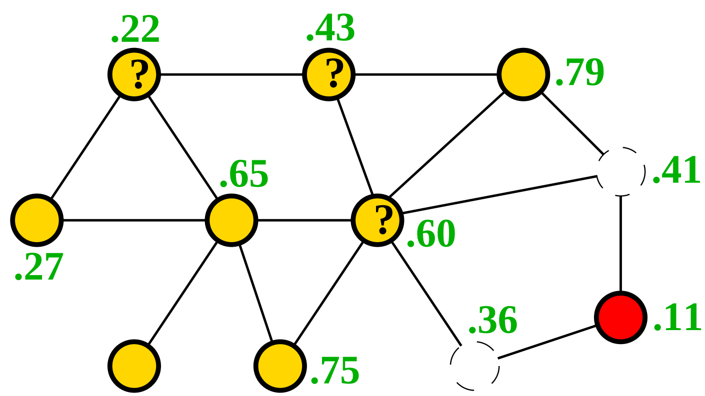
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

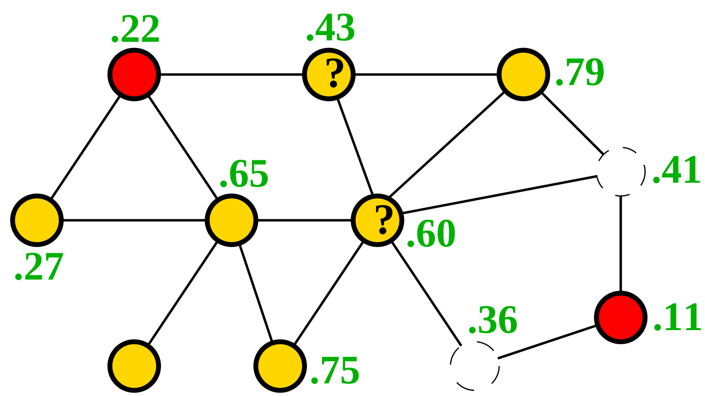
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

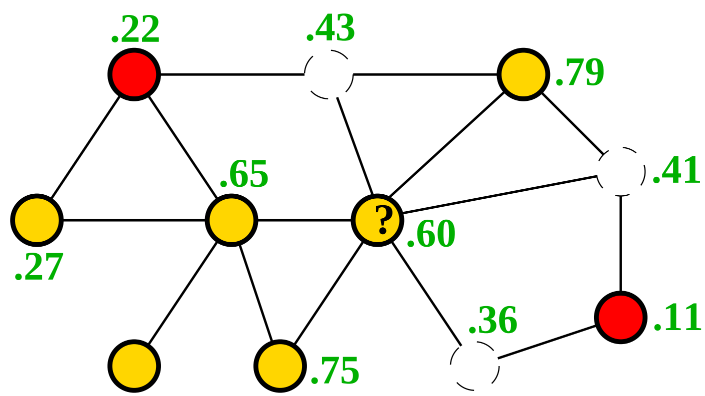
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

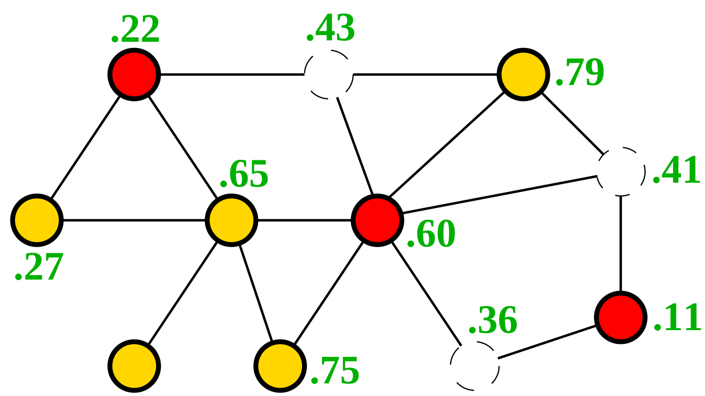
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

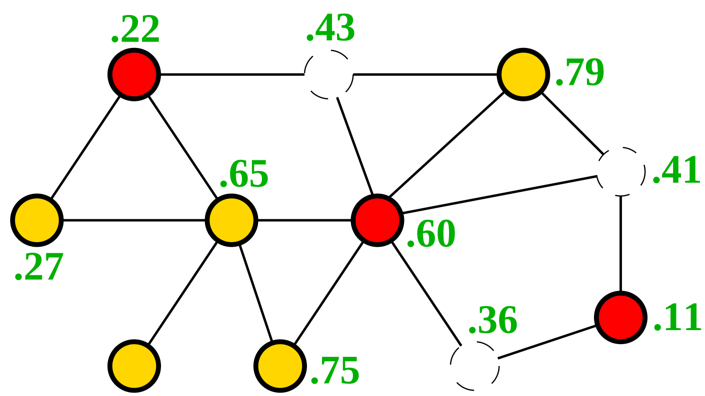
Our Method

Nguyen, O. (2008)

Main idea:

- select maximal independent set greedily
- consider vertices in random order

Random order $\equiv$ random numbers $r(v)$ assigned to each vertex



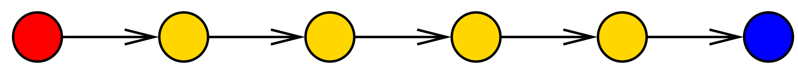
To check if $v \in \mathcal{I}$

- recursively check if neighbors w s.t. $r(w) < r(v)$ are in $\mathcal{I}$
- $v \in \mathcal{I} \iff$ none of them in $\mathcal{I}$

$\mathrm{E}[$#visited vertices$]$ and query complexity of order $2^{O(d)}$

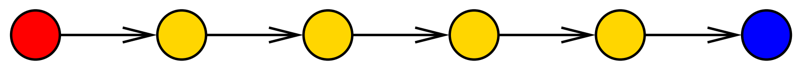
Bounding Expected Query Complexity

- $\Pr[\text{a given path of length } k \text{ is explored}] = 1/(k+1)!$

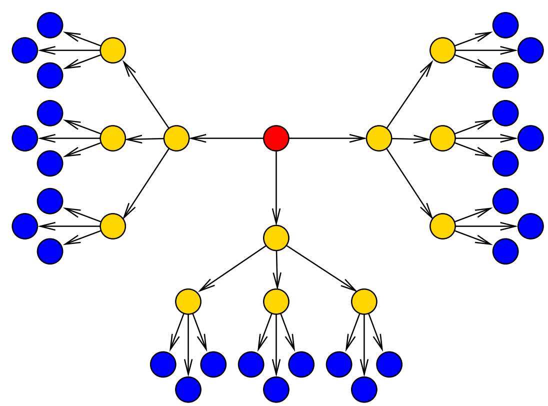


Bounding Expected Query Complexity

- $\Pr[\text{a given path of length } k \text{ is explored}] = 1/(k+1)!$

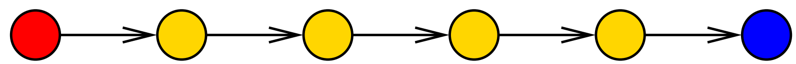


- (number of vertices at distance k) $\leq d^k$

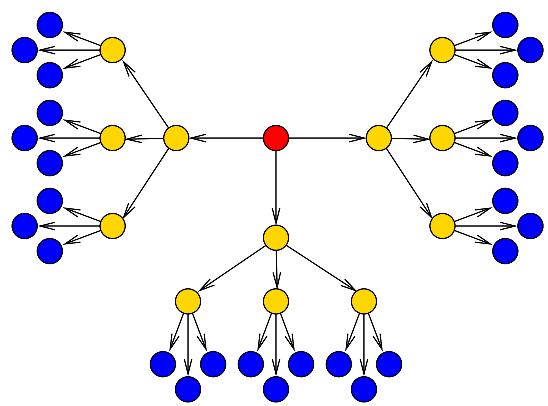


Bounding Expected Query Complexity

- $\Pr[$a given path of length k is explored$] = 1/(k+1)!$



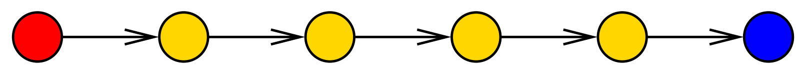
- (number of vertices at distance k) $\leq d^k$



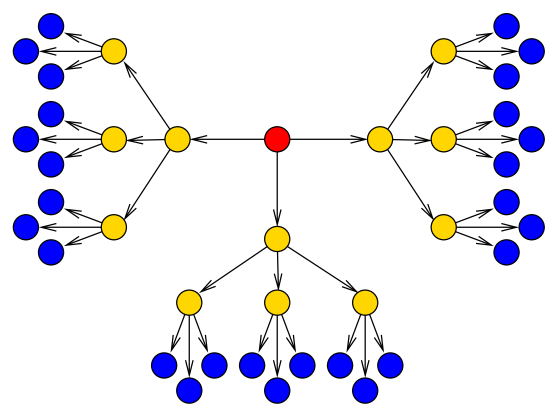
- $E[$number of vertices explored at distance $k] \leq d^k/(k+1)!$

Bounding Expected Query Complexity

- $\Pr[\text{a given path of length } k \text{ is explored}] = 1/(k+1)!$



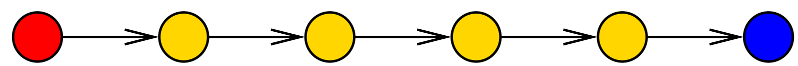
- $(\text{number of vertices at distance } k) \leq d^k$



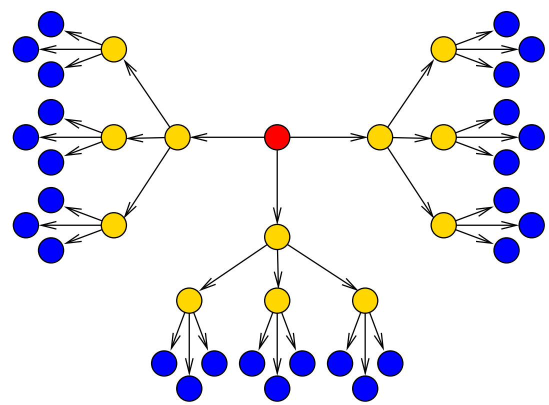
- $E[\text{number of vertices explored at distance } k] \leq d^k/(k+1)!$
- $E[\text{number of explored vertices}] \leq \sum_{k=0}^{\infty} d^k/(k+1)! \leq e^d/d$

Bounding Expected Query Complexity

- $\Pr[\text{a given path of length } k \text{ is explored}] = 1/(k+1)!$



- $(\text{number of vertices at distance } k) \leq d^k$

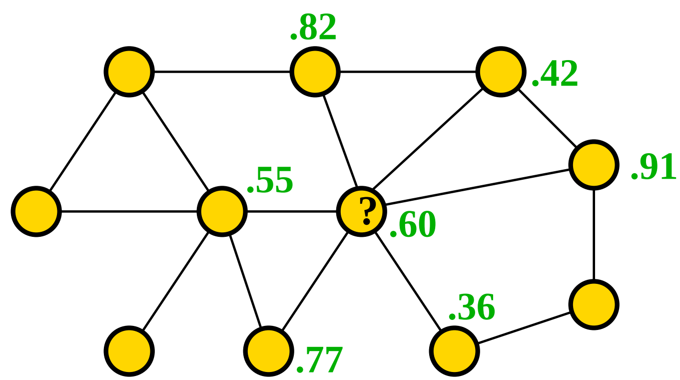


- $E[\text{number of vertices explored at distance } k] \leq d^k/(k+1)!$
- $E[\text{number of explored vertices}] \leq \sum_{k=0}^{\infty} d^k/(k+1)! \leq e^d/d$
- Expected query complexity $= O(d) \cdot e^d/d = O(e^d)$

Improvement

Heuristic:

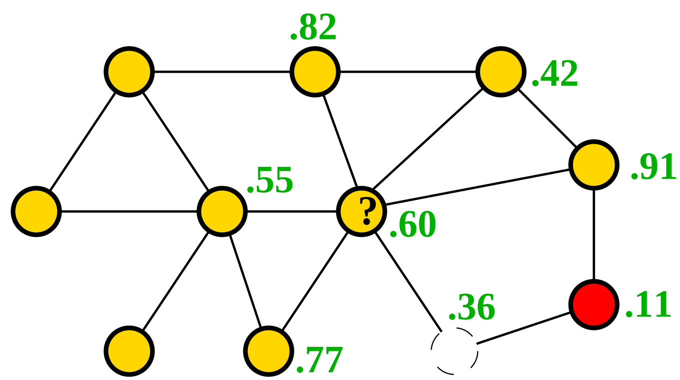
- Consider neighbors w of v in ascending order of $r(w)$
- Once you find $w \in \mathcal{I}$, $v \notin \mathcal{I}$
  (i.e., don't check other neighbors)



Improvement

Heuristic:

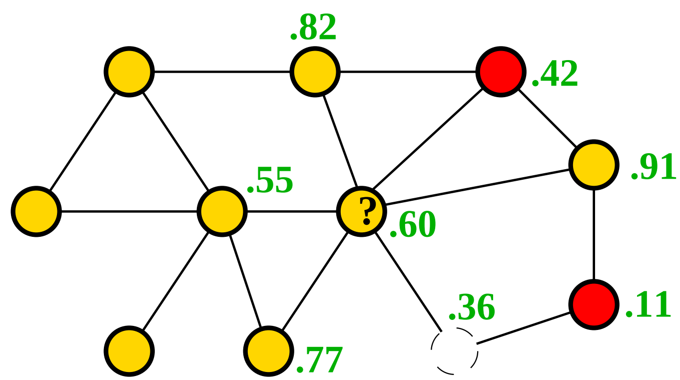
- Consider neighbors w of v in ascending order of $r(w)$
- Once you find $w \in \mathcal{I}$, $v \notin \mathcal{I}$
  (i.e., don't check other neighbors)



Improvement

Heuristic:

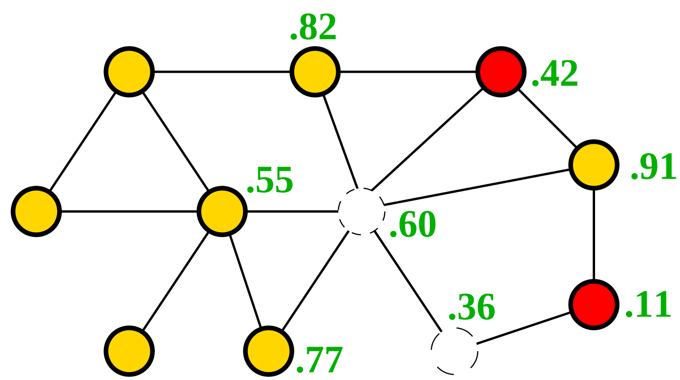
- Consider neighbors w of v in ascending order of $r(w)$
- Once you find $w \in \mathcal{I}$, $v \notin \mathcal{I}$
  (i.e., don't check other neighbors)



Improvement

Heuristic:

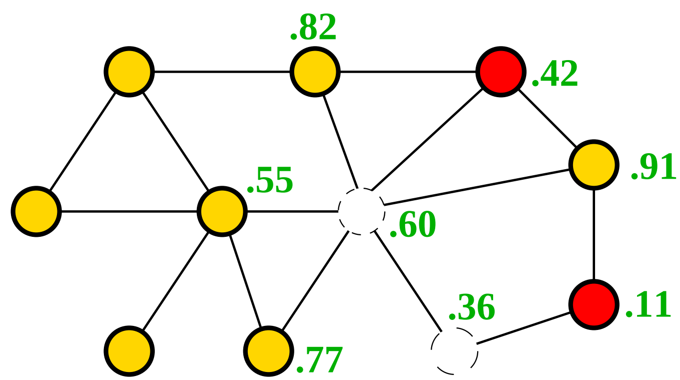
- Consider neighbors w of v in ascending order of $r(w)$
- Once you find $w \in \mathcal{I}$, $v \notin \mathcal{I}$
  (i.e., don't check other neighbors)



Improvement

Heuristic:

- Consider neighbors w of v in ascending order of $r(w)$
- Once you find $w \in \mathcal{I}$, $v \notin \mathcal{I}$
  (i.e., don't check other neighbors)



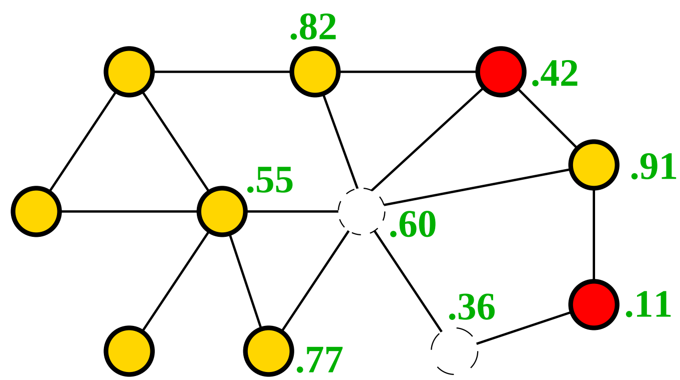
Yoshida, Yamamoto, Ito (STOC 2009):

$$\mathrm{E}_{\text{permutations, } \textcolor{red}{\text{start vertex}}}[\text{\#recursive calls}] \leq 1 + \frac{m}{n}$$

Improvement

Heuristic:

- Consider neighbors w of v in ascending order of $r(w)$
- Once you find $w \in \mathcal{I}$, $v \notin \mathcal{I}$
  (i.e., don't check other neighbors)



Yoshida, Yamamoto, Ito (STOC 2009):

$$\mathrm{E}_{\text{permutations, start vertex}}[\text{\#recursive calls}] \le 1 + \frac{m}{n}$$

Which gives:

expected query complexity for random vertex = $O(d^2)$

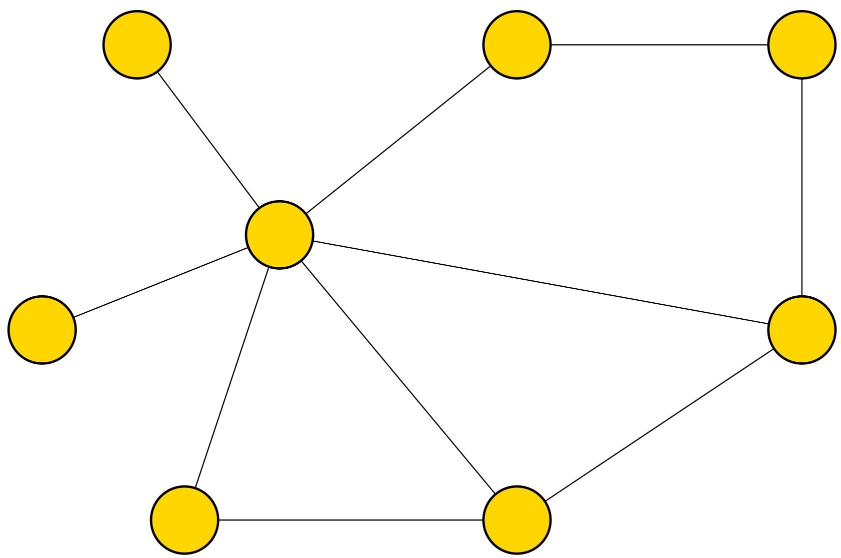
Algorithm for Vertex Cover

Vertex Cover

Goal: find smallest set S of nodes such that each edge has endpoint in S

Classical 2-approximation algorithm [Gavril & Yannakakis]:

- Greedily find a maximal matching M
- Output the set of nodes matched in M

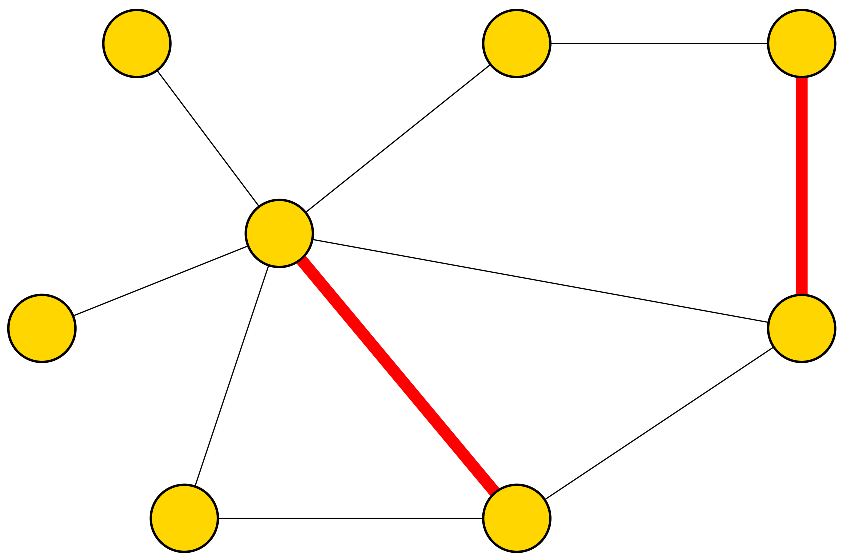


Vertex Cover

Goal: find smallest set S of nodes such that each edge has endpoint in S

Classical 2-approximation algorithm [Gavril & Yannakakis]:

- Greedily find a maximal matching M
- Output the set of nodes matched in M

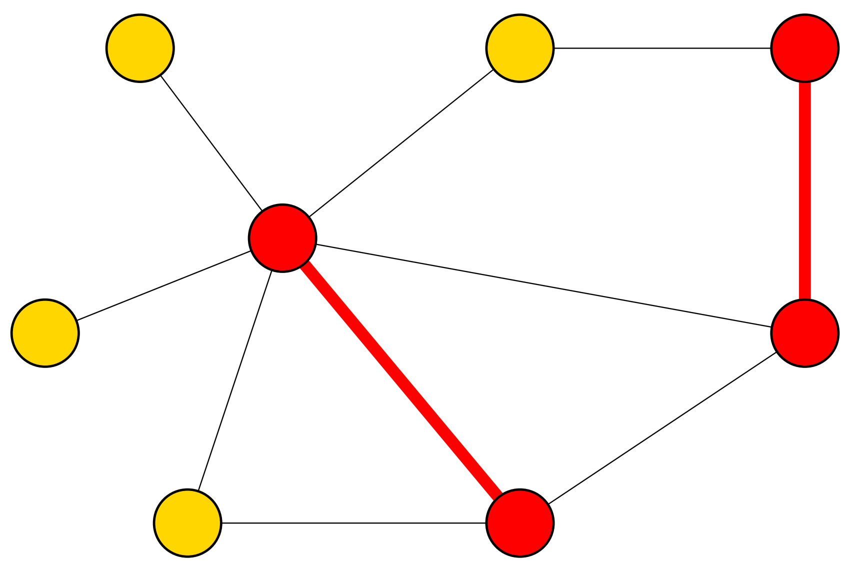


Vertex Cover

Goal: find smallest set S of nodes such that each edge has endpoint in S

Classical 2-approximation algorithm [Gavril & Yannakakis]:

- Greedily find a maximal matching M
- Output the set of nodes matched in M



Sublinear-Time Algorithm

General Idea:

- construct oracle $\mathcal{O}$ that answers queries: Is $e \in E$ in M? for a fixed maximal matching M

Sublinear-Time Algorithm

General Idea:

- construct oracle $\mathcal{O}$ that answers queries: Is $e \in E$ in M? for a fixed maximal matching M

maximal matching

$\equiv$

maximal independent set in the line graph

Sublinear-Time Algorithm

General Idea:

- construct oracle $\mathcal{O}$ that answers queries: Is $e \in E$ in M? for a fixed maximal matching M

  maximal matching
  $$\equiv$$
  maximal independent set in the line graph

- approximate the number of vertices matched in M up to $\pm\epsilon n$ by checking for $O(1/\epsilon^2)$ vertices if they are matched

  #queries to $\mathcal{O}$ = (#tested nodes) $\cdot$ (max-degree) $= O(d/\epsilon^2)$

Sublinear-Time Algorithm

General Idea:

- construct oracle $\mathcal{O}$ that answers queries: Is $e \in E$ in M? for a fixed maximal matching M

maximal matching

$\equiv$

maximal independent set in the line graph

- approximate the number of vertices matched in M up to $\pm\epsilon n$ by checking for $O(1/\epsilon^2)$ vertices if they are matched

$$\#\text{queries to } \mathcal{O} = (\#\text{tested nodes}) \cdot (\text{max-degree}) = O(d/\epsilon^2)$$

This gives:

$$\mathrm{VC} - \epsilon n \leq (\text{computed value}) \leq 2 \cdot \mathrm{VC} + \epsilon n$$

Sublinear-Time Algorithm

General Idea:

- construct oracle $\mathcal{O}$ that answers queries: Is $e \in E$ in M? for a fixed maximal matching M

maximal matching

$\equiv$

maximal independent set in the line graph

- approximate the number of vertices matched in M up to $\pm\epsilon n$ by checking for $O(1/\epsilon^2)$ vertices if they are matched

#queries to $\mathcal{O}$ $=$ (#tested nodes) $\cdot$ (max-degree) $= O(d/\epsilon^2)$

This gives:

$$\mathrm{VC} - \epsilon n \leq (\text{computed value}) \leq 2 \cdot \mathrm{VC} + \epsilon n$$

Running time: $2^{O(d)}/\epsilon^2$

$$\mathrm{VC} - \epsilon n \leq \textbf{output} \leq 2 \cdot \mathrm{VC} + \epsilon n$$

Parnas, Ron (2007): $d^{O(\log(d)/\epsilon^3)}$ queries

- via simulation of local distributed algorithms

$\mathrm{VC} - \epsilon n \leq \textbf{output} \leq 2 \cdot \mathrm{VC} + \epsilon n$

Parnas, Ron (2007): $d^{O(\log(d)/\epsilon^3)}$ queries

- via simulation of local distributed algorithms

Marko, Ron (2007): $d^{O(\log(d/\epsilon))}$ queries

- via Luby’s algorithm

$\mathrm{VC} - \epsilon n \le \textbf{output} \le 2 \cdot \mathrm{VC} + \epsilon n$

Parnas, Ron (2007): $d^{O(\log(d)/\epsilon^3)}$ queries

- via simulation of local distributed algorithms

Marko, Ron (2007): $d^{O(\log(d/\epsilon))}$ queries

- via Luby's algorithm

Nguyen, O. (2008): $2^{O(d)}/\epsilon^2$ queries

- the algorithm and proof presented here

$\mathrm{VC} - \epsilon n \le \textbf{output} \le 2 \cdot \mathrm{VC} + \epsilon n$

Parnas, Ron (2007): $d^{O(\log(d)/\epsilon^3)}$ queries

- via simulation of local distributed algorithms

Marko, Ron (2007): $d^{O(\log(d/\epsilon))}$ queries

- via Luby's algorithm

Nguyen, O. (2008): $2^{O(d)}/\epsilon^2$ queries

- the algorithm and proof presented here

Yoshida, Yamamoto, Ito (2009): $O(d^4/\epsilon^2)$ queries

- the Nguyen, O. algorithm $+$ analysis of the heuristic

$\mathrm{VC} - \epsilon n \le \textbf{output} \le 2 \cdot \mathrm{VC} + \epsilon n$

Parnas, Ron (2007): $d^{O(\log(d)/\epsilon^3)}$ queries

- via simulation of local distributed algorithms

Marko, Ron (2007): $d^{O(\log(d/\epsilon))}$ queries

- via Luby's algorithm

Nguyen, O. (2008): $2^{O(d)}/\epsilon^2$ queries

- the algorithm and proof presented here

Yoshida, Yamamoto, Ito (2009): $O(d^4/\epsilon^2)$ queries

- the Nguyen, O. algorithm $+$ analysis of the heuristic

O., Ron, Rosen, Rubinfeld (2012): $\tilde{O}(d/\epsilon^3)$ queries

- further refinements of Nguyen, O. and YYI
- sampling from the neighbor sets
- near optimal: $\Omega(d)$ lower bound due to Parnas, Ron (2007)

Lower Bounds

Trevisan 2007:

- $(c, \epsilon n)$-approximation requires $\Omega(\sqrt{n})$ queries for $c < 2$

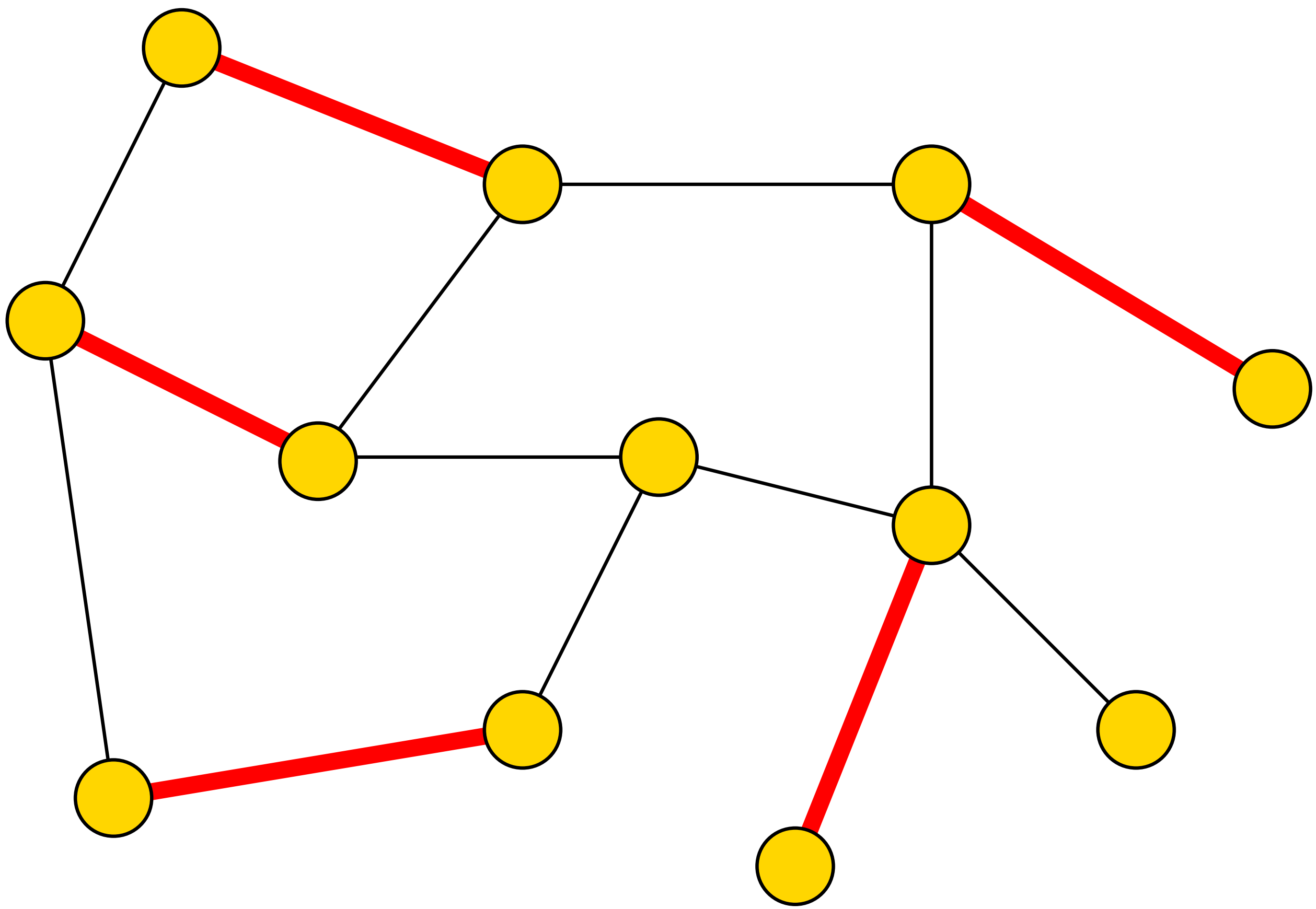
Parnas, Ron 2007:

- $(O(1), \epsilon n)$-approximation requires $\Omega(d)$ queries

Better Approximation for Maximum Matching

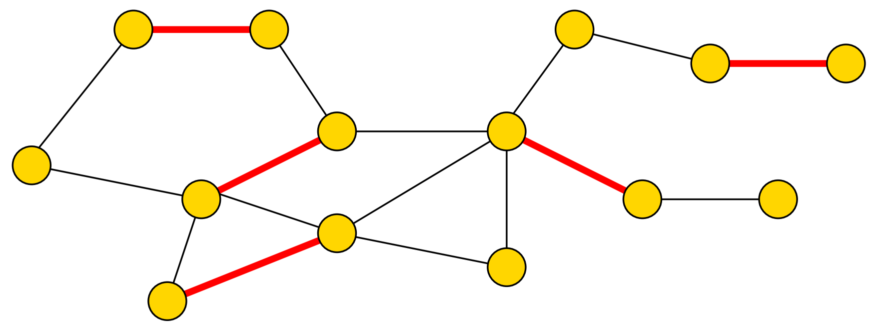
Maximum Matching

Goal: find a set of disjoint edges of maximum cardinality



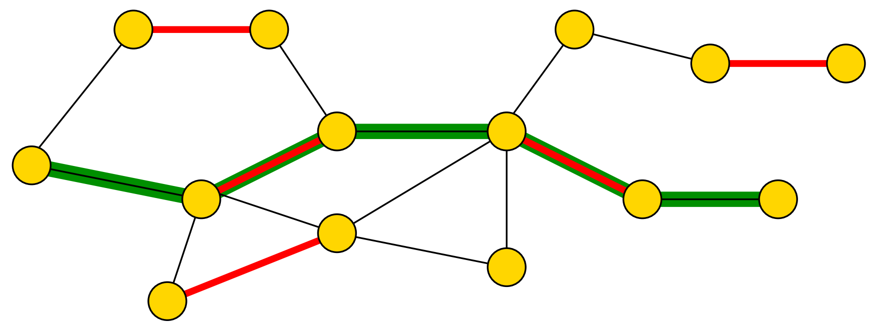
Review of Properties

Augmenting Path: a path that improves matching



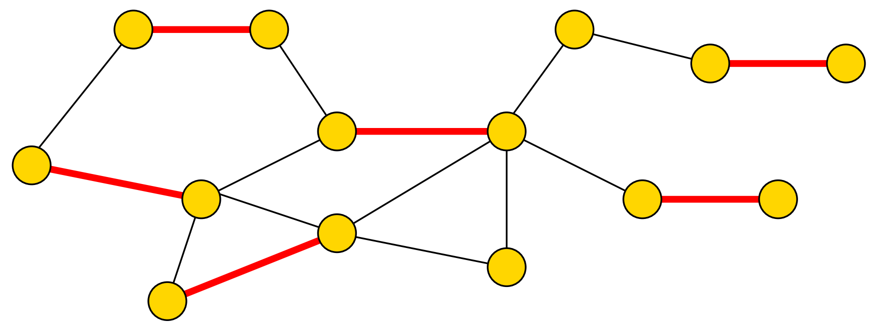
Review of Properties

Augmenting Path: a path that improves matching



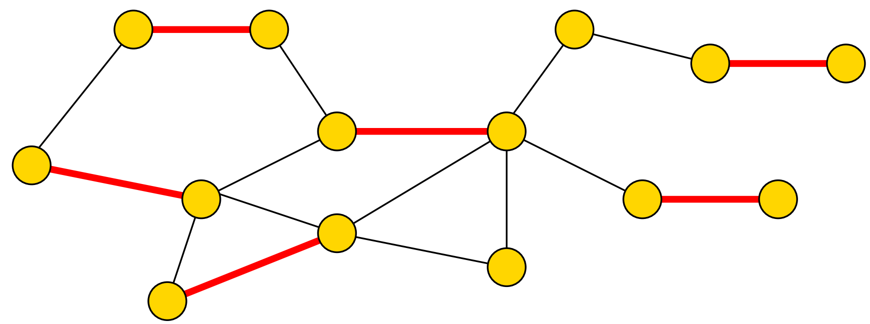
Review of Properties

Augmenting Path: a path that improves matching



Review of Properties

Augmenting Path: a path that improves matching

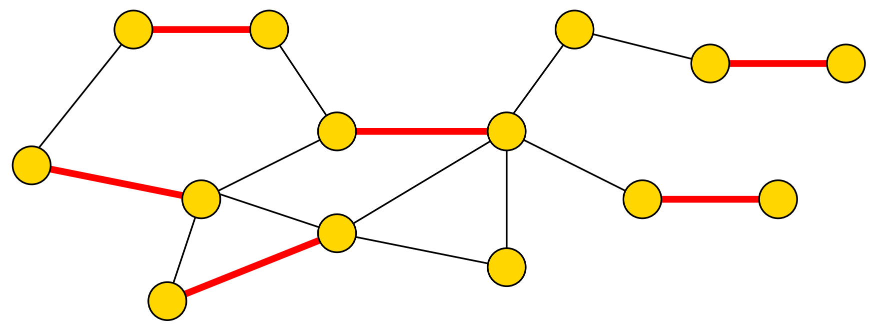


M = matching, M^* = maximum matching

Fact: There are $|M^*| - |M|$ disjoint augmenting paths for M

Review of Properties

Augmenting Path: a path that improves matching



M = matching, M^* = maximum matching

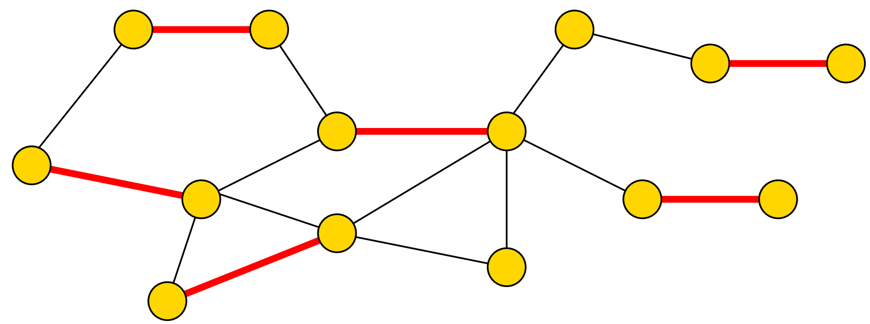
Fact: There are $|M^*| - |M|$ disjoint augmenting paths for M

Fact:

No augmenting paths of length $< 2k+1 \;\Rightarrow\; |M| \geq \frac{k}{k+1}|M^*|$

Review of Properties

Augmenting Path: a path that improves matching



M = matching, M^* = maximum matching

Fact: There are $|M^*| - |M|$ disjoint augmenting paths for M

Fact:
No augmenting paths of length $< 2k+1 \;\Rightarrow\; |M| \geq \frac{k}{k+1}|M^*|$

To get $(1+\epsilon)$-approximation, set $k = \lceil 1/\epsilon \rceil$

Standard Algorithm

Lemma [Hopcroft, Karp 1973]:

M = matching with no augmenting paths of length $< t$

P = maximal set of vertex-disjoint augmenting paths of length t for M

M' = M with all paths in P applied

Claim: M' has only augmenting paths of length $> t$

Standard Algorithm

Lemma [Hopcroft, Karp 1973]:

M = matching with no augmenting paths of length $< t$

P = maximal set of vertex-disjoint augmenting paths of length t for M

M' = M with all paths in P applied

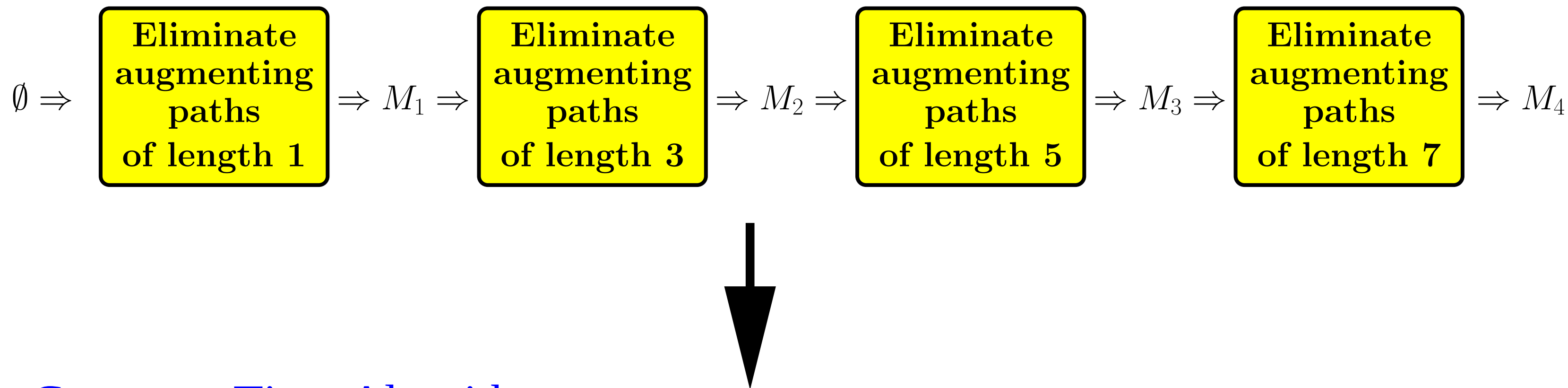
Claim: M' has only augmenting paths of length $> t$

Algorithm:

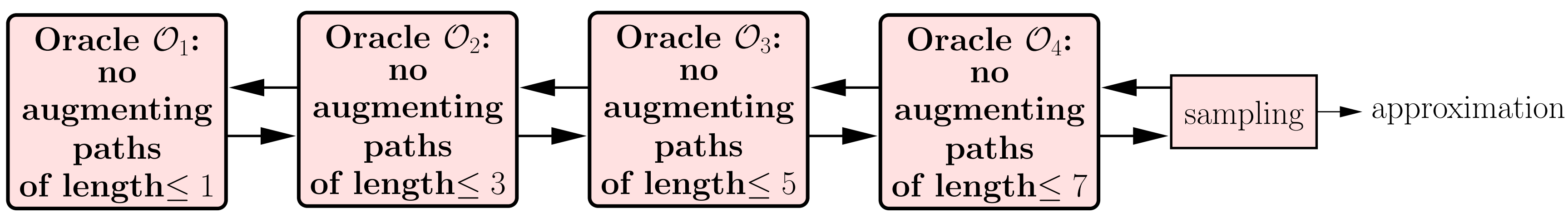
$M :=$ empty matching
for $i = 1$ to k:
  find maximal set of disjoint augmenting paths of length $2i - 1$
  apply all paths to M
return M

Transformation

Standard Algorithm:



Constant−Time Algorithm:

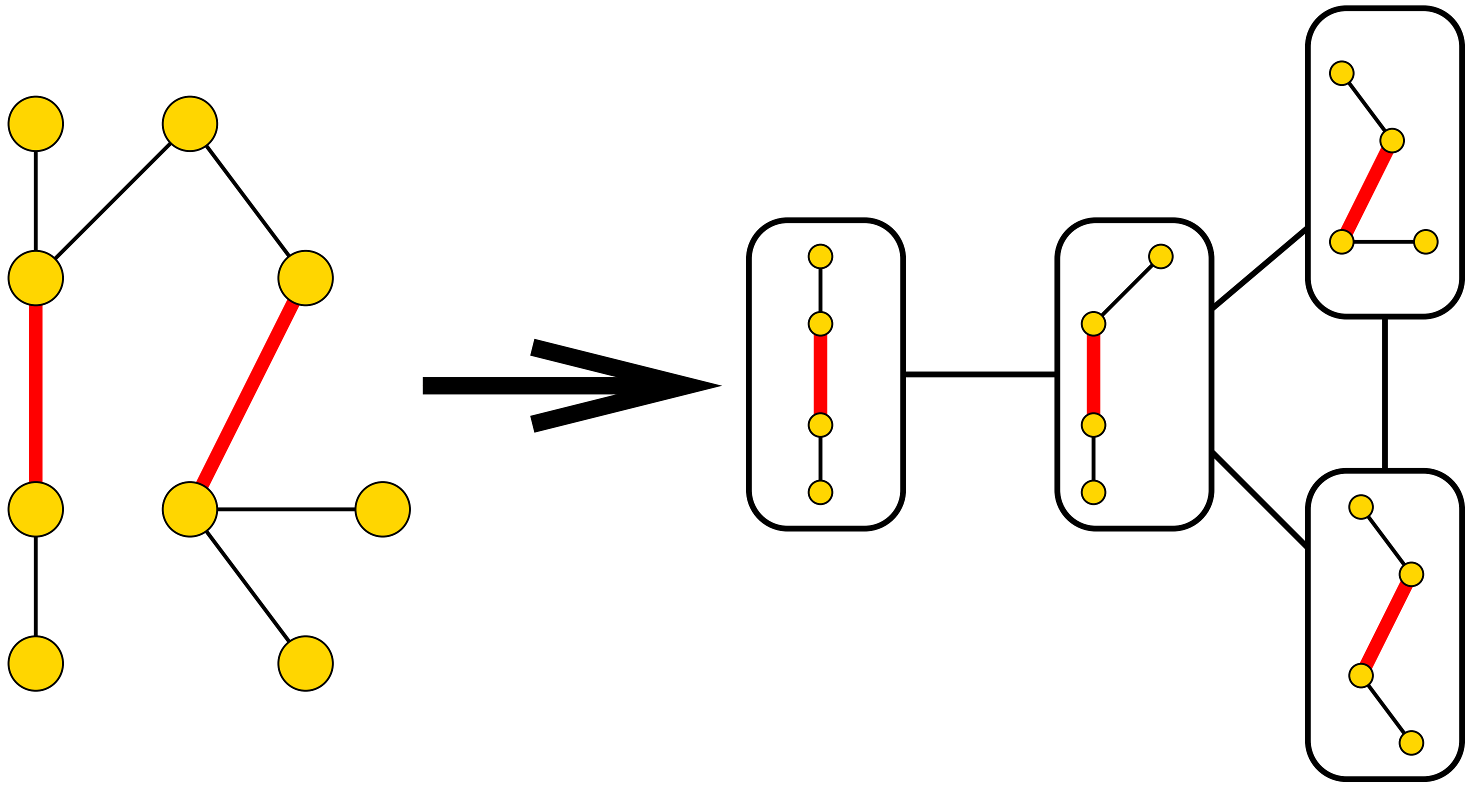


Oracle $\mathcal{O}_i$:

- provides query access to M_i
- simulates applying to M_{i-1} a maximal set of disjoint augmenting paths of length $2i-1$

Transformation

Sample graph considered by $\mathcal{O}_2$:



$\mathcal{O}_i$'s graph has degree $d^{O(i)}$

Query Complexity

Can't apply the previous approach!

- every query may disclose some information about the random numbers
- algorithm could use it to form a malicious query

Query Complexity

Can't apply the previous approach!

- every query may disclose some information about the random numbers
- algorithm could use it to form a malicious query

Locality Lemma:

for q queries, needs to visit at most $q^2 \cdot 2^{O(d^4)}/\delta$ vertices with probability $1 - \delta$

Query Complexity

Can't apply the previous approach!

- every query may disclose some information about the random numbers
- algorithm could use it to form a malicious query

Locality Lemma:

for q queries, needs to visit at most $q^2 \cdot 2^{O(d^4)}/\delta$ vertices with probability $1-\delta$

Query complexity: $2^{d^{O(1/\epsilon)}}$ queries for $(1, \epsilon n)$-approximation

Query Complexity

Can't apply the previous approach!

- every query may disclose some information about the random numbers
- algorithm could use it to form a malicious query

Locality Lemma:

for q queries, needs to visit at most $q^2 \cdot 2^{O(d^4)}/\delta$ vertices with probability $1 - \delta$

Query complexity: $2^{d^{O(1/\epsilon)}}$ queries for $(1, \epsilon n)$-approximation

Yoshida, Yamamoto, Ito (2009)

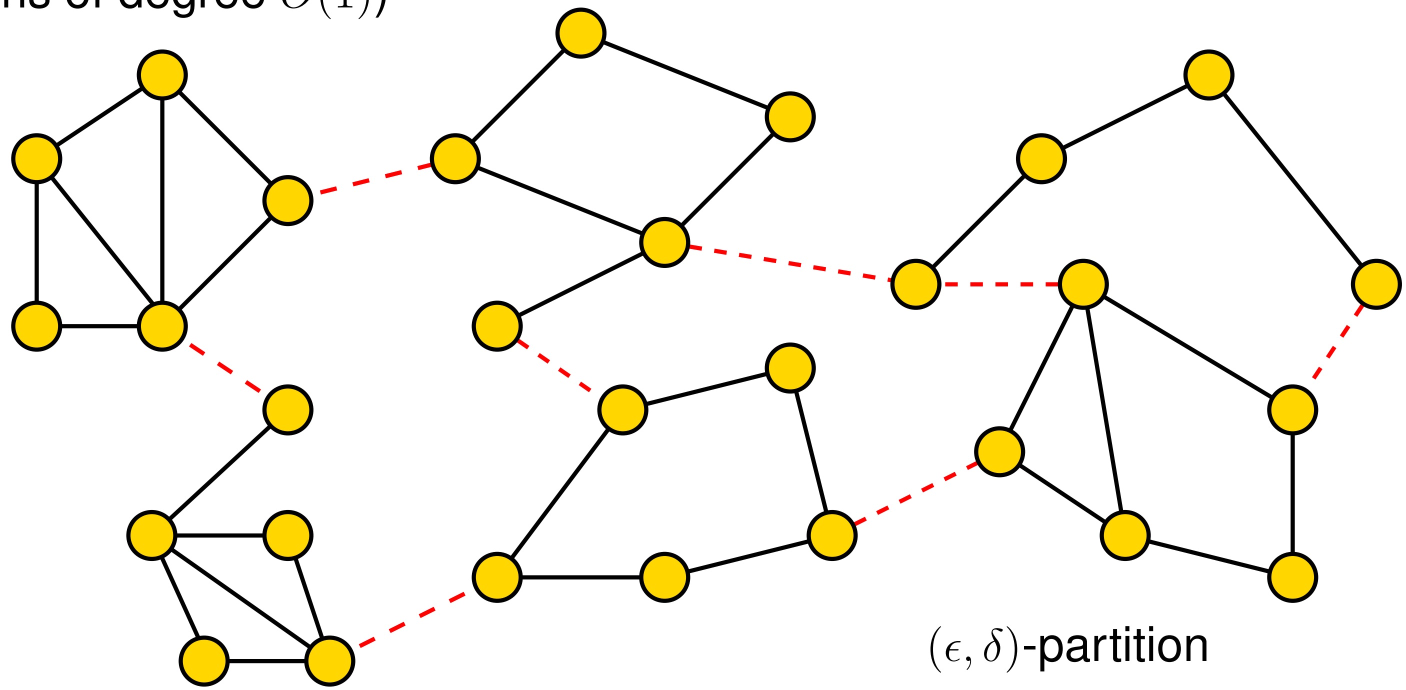
- Query complexity: $d^{O(1/\epsilon^2)}$
- uniform on higher level $\Rightarrow$ close to uniform on lower

Local Graph Partitions

[Hassidim, Kelner, Nguyen, O. 2009]

Hyperfinite Graphs

(All graphs of degree $O(1)$)

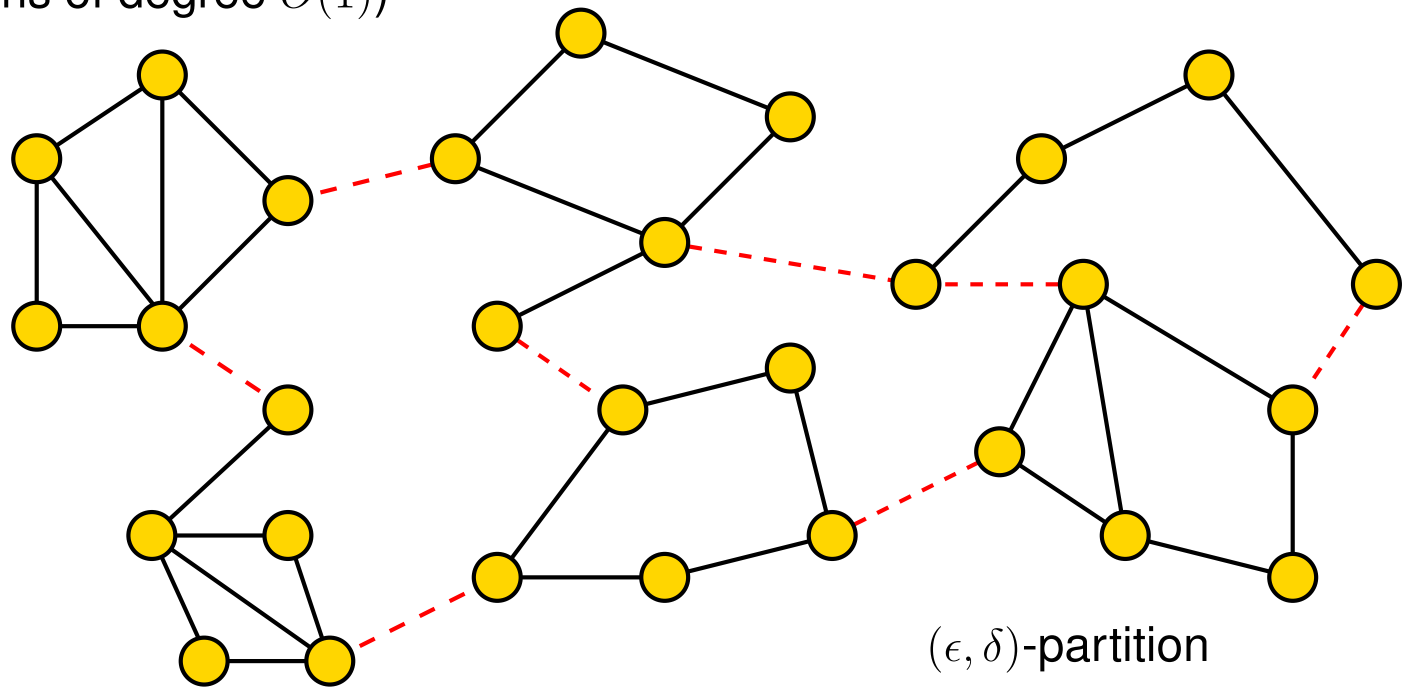


(ϵ, δ)-partition

- (ϵ, δ)-hyperfinite graphs: can remove $\epsilon|V|$ edges and get components of size at most δ

Hyperfinite Graphs

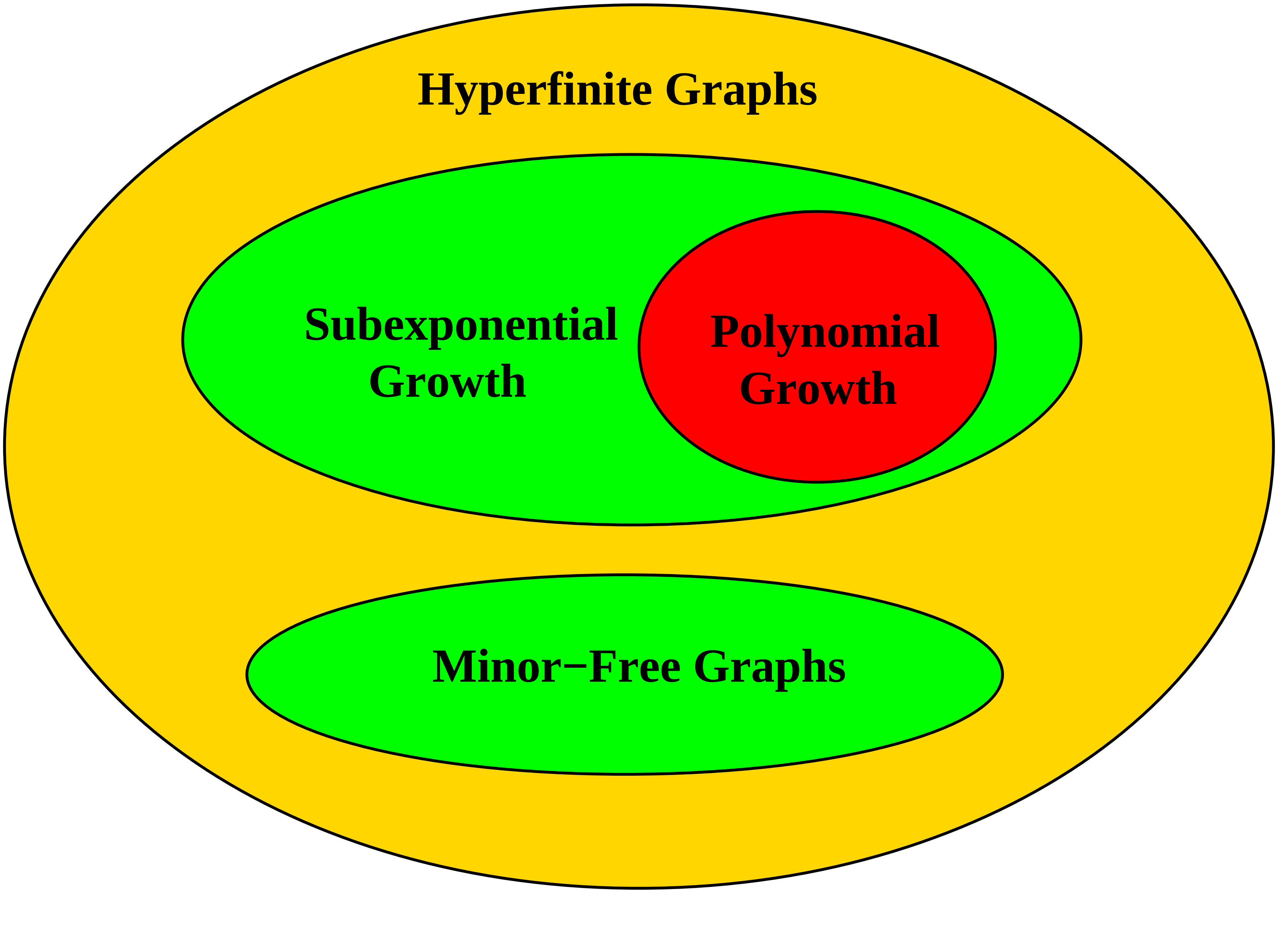
(All graphs of degree $O(1)$)



(ϵ, δ)-partition

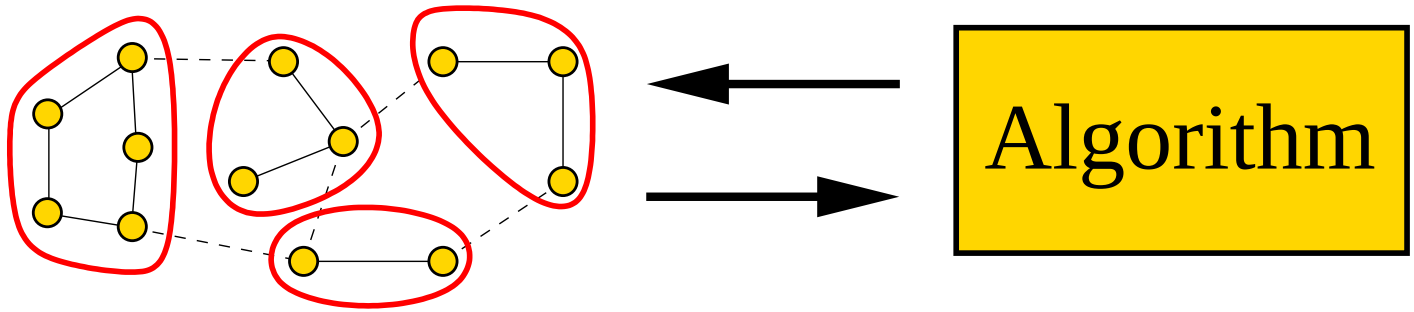
- (ϵ, δ)-hyperfinite graphs: can remove $\epsilon|V|$ edges and get components of size at most δ
- hyperfinite family of graphs: there is ρ such that all graphs are $(\epsilon, \rho(\epsilon))$-hyperfinite for all $\epsilon > 0$

Taxonomy


Hyperfinite Graphs
Subexponential Growth
Polynomial Growth
Minor−Free Graphs

Using a Partition

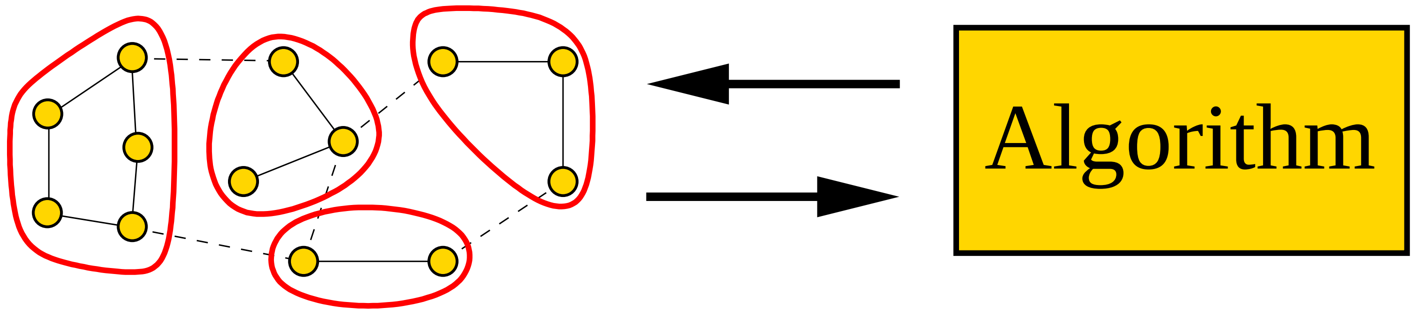
If someone gave us a $(\epsilon/2, \delta)$-partition:



- Sample $O(1/\epsilon^2)$ vertices
- Compute minimum vertex cover for the sampled components
- Return the fraction of the sampled vertices in the covers

Using a Partition

If someone gave us a $(\epsilon/2, \delta)$-partition:



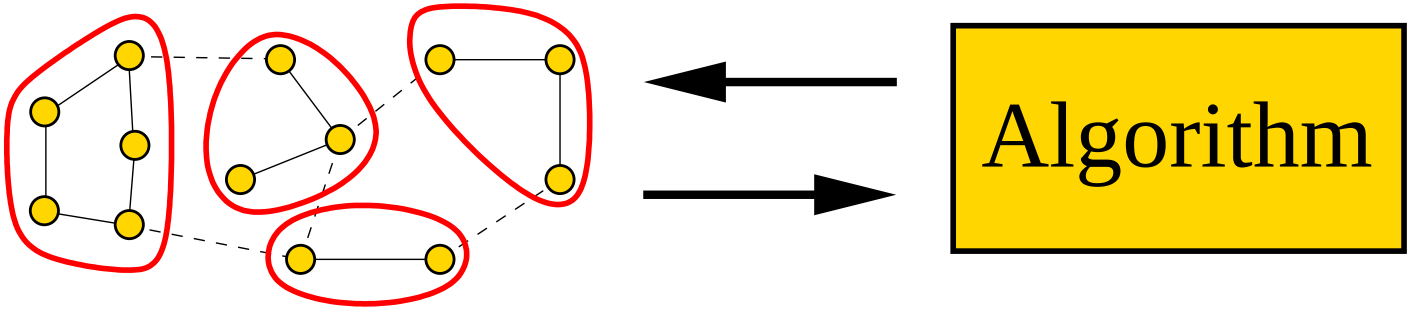
- Sample $O(1/\epsilon^2)$ vertices
- Compute minimum vertex cover for the sampled components
- Return the fraction of the sampled vertices in the covers

This gives $\pm\epsilon$ approximation to $\mathrm{VC}(G)/n$ in constant time:

- Cut edges change $\mathrm{VC}(G)$ by at most $\epsilon n/2$
- Can compute vertex cover separately for each component

Using a Partition

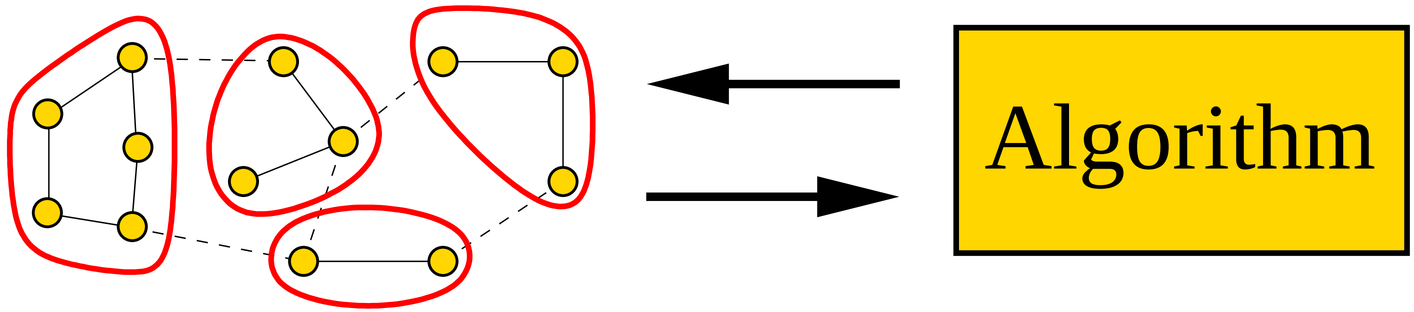
If someone gave us a $(\epsilon/2, \delta)$-partition:



- We can compute the partition without looking at the entire graph

Using a Partition

If someone gave us a $(\epsilon/2, \delta)$-partition:



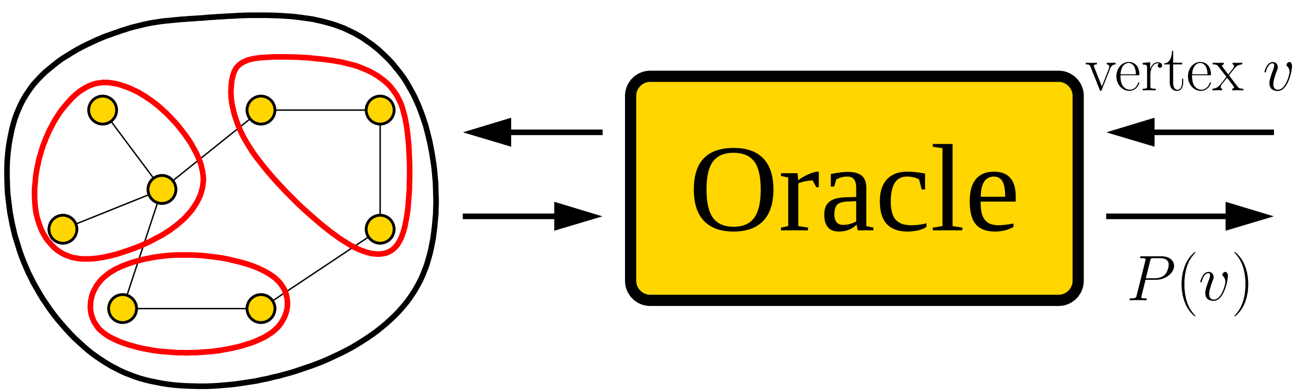
- We can compute the partition without looking at the entire graph

New Tool: Partitioning Oracles

Partitioning Oracle

$\mathcal{C} =$ fixed hyperfinite class

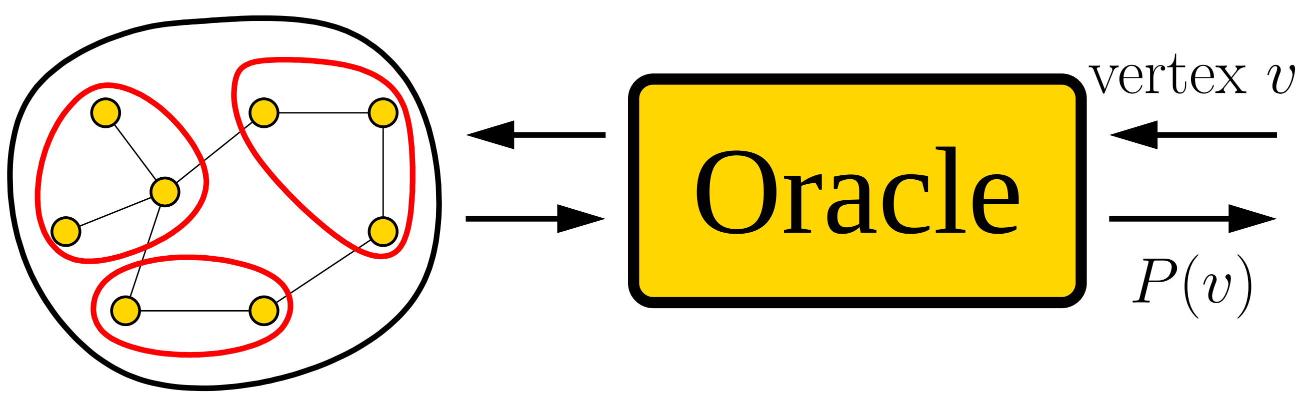
- oracle has query access to $G = (V, E)$
  (G need not be in $\mathcal{C}$)



Partitioning Oracle

$\mathcal{C} =$ fixed hyperfinite class

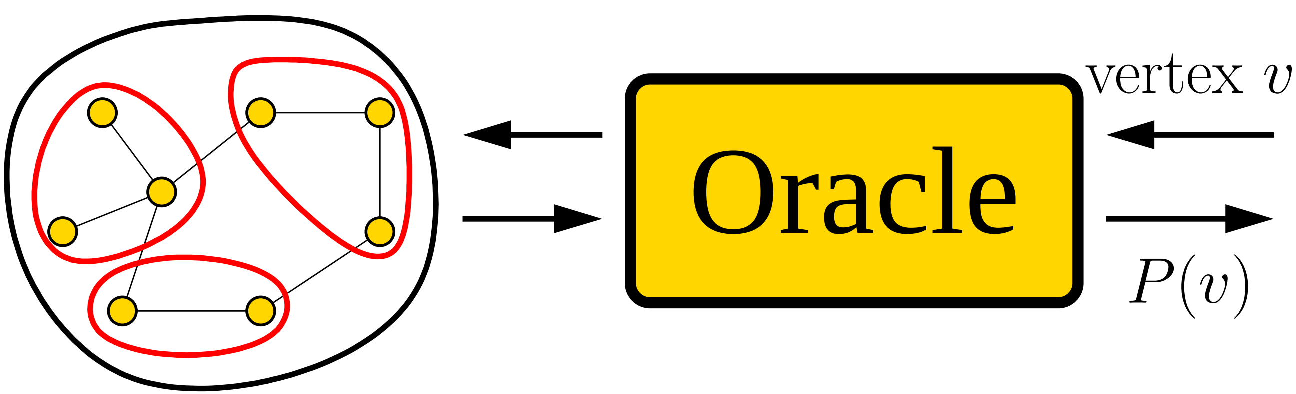
- oracle has query access to $G = (V, E)$
  (G need not be in $\mathcal{C}$)
- oracle provides query access to partition P of V;
  for each v, oracle returns $P(v) \subseteq V$ s.t. $v \in P(v)$



Partitioning Oracle

$\mathcal{C} =$ fixed hyperfinite class

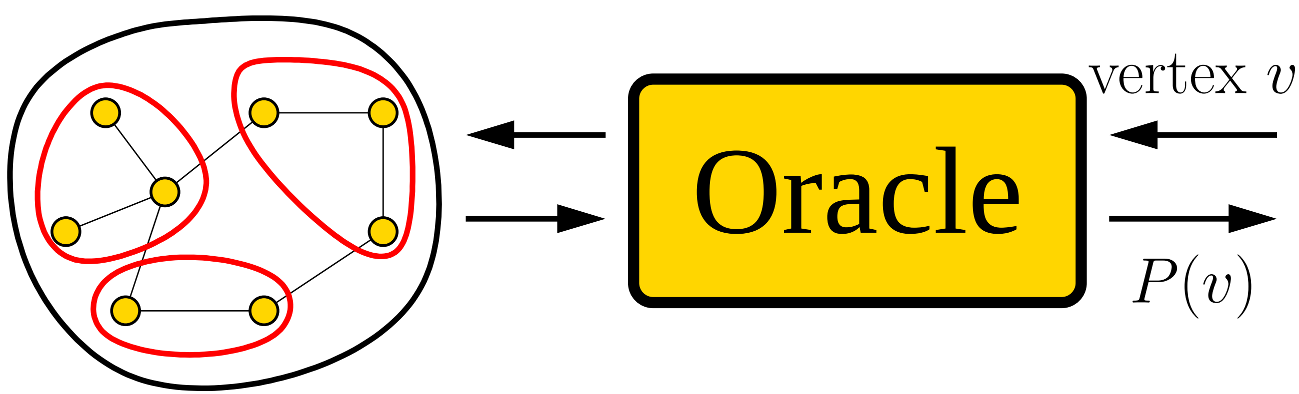
- oracle has query access to $G = (V, E)$
  (G need not be in $\mathcal{C}$)
- oracle provides query access to partition P of V;
  for each v, oracle returns $P(v) \subseteq V$ s.t. $v \in P(v)$
- Properties of P:
  - each $|P(v)| = O(1)$



Partitioning Oracle

$\mathcal{C} =$ fixed hyperfinite class

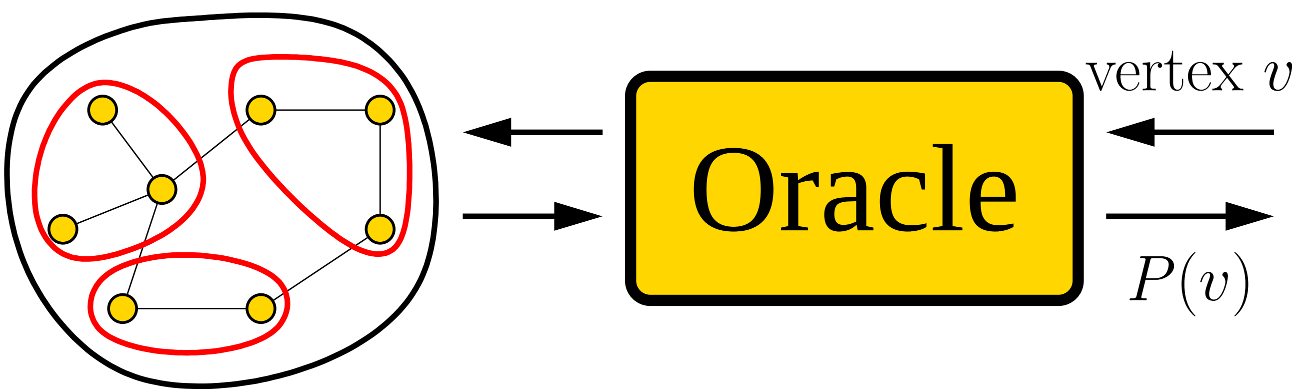
- oracle has query access to $G = (V, E)$
  (G need not be in $\mathcal{C}$)
- oracle provides query access to partition P of V; for each v, oracle returns $P(v) \subseteq V$ s.t. $v \in P(v)$
- Properties of P:
  - each $|P(v)| = O(1)$
  - If $G \in \mathcal{C}$, number of cut edges $\leq \epsilon n$ w.p. $\frac{99}{100}$



Partitioning Oracle

$\mathcal{C} =$ fixed hyperfinite class

- oracle has query access to $G = (V, E)$
  (G need not be in $\mathcal{C}$)
- oracle provides query access to partition P of V;
  for each v, oracle returns $P(v) \subseteq V$ s.t. $v \in P(v)$
- Properties of P:
  - each $|P(v)| = O(1)$
  - If $G \in \mathcal{C}$, number of cut edges $\leq \epsilon n$ w.p. $\frac{99}{100}$
  - partition $P(\cdot)$ is not a function of queries,
    it is a function of graph structure and random bits



Oracle Implementations

- Generic oracle for any hyperfinite class of graphs
  - Query complexity: $2^{d^{O(\rho(\epsilon^3/C))}}$ for some constant C
  - Via local simulation of a greedy partitioning procedure (uses [Nguyen, O. 2008])

Oracle Implementations

- Generic oracle for any hyperfinite class of graphs
  - Query complexity: $2^{d^{O(\rho(\epsilon^3/C))}}$ for some constant C
- For minor-free graphs:
  - Query complexity: $d^{\mathrm{poly}(1/\epsilon)}$
  - Via techniques from distributed algorithms [Czygrinow, Hańćkowiak, Wawrzyniak 2008]

Oracle Implementations

- Generic oracle for any hyperfinite class of graphs
  - Query complexity: $2^{d^{O(\rho(\epsilon^3/C))}}$ for some constant C
- For minor-free graphs:
  - Query complexity: $d^{O(\log^2(1/\epsilon))}$
  - Via techniques from distributed algorithms [Czygrinow, Hańćkowiak, Wawrzyniak 2008]
  - Improved by Levi and Ron (2013)

Oracle Implementations

- Generic oracle for any hyperfinite class of graphs
  - Query complexity: $2^{d^{O(\rho(\epsilon^3/C))}}$ for some constant C
- For minor-free graphs:
  - Query complexity: $d^{O(\log^2(1/\epsilon))}$
- For $\rho(\epsilon) \leq \mathrm{poly}(1/\epsilon)$:
  - Query complexity: $2^{\mathrm{poly}(d/\epsilon)}$
  - Via methods from distributed algorithms and partitioning methods of Andersen and Peres (2009)

Oracle Implementations

- Generic oracle for any hyperfinite class of graphs
  - Query complexity: $2^{d^{O(\rho(\epsilon^3/C))}}$ for some constant C
- For minor-free graphs:
  - Query complexity: $d^{O(\log^2(1/\epsilon))}$
- For $\rho(\epsilon) \leq \mathrm{poly}(1/\epsilon)$:
  - Query complexity: $2^{\mathrm{poly}(d/\epsilon)}$
- Constant Treewidth:
  - Query complexity: $\mathrm{poly}(d/\epsilon)$
  - Edelman, Hassidim, Nguyen, O. (2011)

Two Applications

1. Approximately learning hyperfinite graphs
   - Then solve an arbitrary problems on almost the same graph

Two Applications

1. Approximately learning hyperfinite graphs
   - Then solve an arbitrary problems on almost the same graph

2. Testing minor-closed properties
   - Simpler proof of the result due to Benjamini, Schramm, and Shapira (2008)
   - Much faster tester

Application 1: Learning

- Input graphs can be decomposed into constant size components by cutting few edges
- Algorithm:
  - sample large constant number of vertices
  - query their components
  - approximately learn the distribution of components

Application 1: Learning

- Input graphs can be decomposed into constant size components by cutting few edges
- Algorithm:
  - sample large constant number of vertices
  - query their components
  - approximately learn the distribution of components
- component size $\leq k \Rightarrow \leq 2^{k^2}$ different component types
- Can learn a graph close to the input by sampling $2^{k^2} \cdot O(1/\epsilon^2)$ vertices

Application 1: Learning

- Input graphs can be decomposed into constant size components by cutting few edges
- Algorithm:
  - sample large constant number of vertices
  - query their components
  - approximately learn the distribution of components
- component size $\leq k \Rightarrow \leq 2^{k^2}$ different component types
- Can learn a graph close to the input by sampling $2^{k^2} \cdot O(1/\epsilon^2)$ vertices
- Application: solve any testing or approximation problem on almost the same graph
- First proof: Newman and Sohler (2011)

Application 2: Testing

Testing H-minor-freeness in the sparse graph model of Goldreich and Ron (1997)

- Input: query access to constant degree graph G
  & parameter $\epsilon > 0$
- Goal: w.p. $2/3$
  - accept H-minor-free graphs
  - reject graphs far from H-minor-freeness: $\geq \epsilon n$ edges must be removed to achieve H-minor-freeness

Application 2: Testing

Testing H-minor-freeness in the sparse graph model of Goldreich and Ron (1997)

- Input: query access to constant degree graph G
  & parameter $\epsilon > 0$
- Goal: w.p. $2/3$
  - accept H-minor-free graphs
  - reject graphs far from H-minor-freeness: $\geq \epsilon n$ edges must be removed to achieve H-minor-freeness

Time and query complexity:

- Goldreich, Ron (1997): cycle-freeness in $\mathrm{poly}(1/\epsilon)$ time
- Benjamini, Schramm, Shapira (2008): any minor in $2^{2^{2^{\mathrm{poly}(1/\epsilon)}}}$ time

Application 2: Testing

Testing H-minor-freeness in the sparse graph model of Goldreich and Ron (1997)

- Input: query access to constant degree graph G
  & parameter $\epsilon > 0$
- Goal: w.p. $2/3$
  - accept H-minor-free graphs
  - reject graphs far from H-minor-freeness: $\geq \epsilon n$ edges must be removed to achieve H-minor-freeness

Time and query complexity:

- Goldreich, Ron (1997): cycle-freeness in $\mathrm{poly}(1/\epsilon)$ time
- Benjamini, Schramm, Shapira (2008): any minor in $2^{2^{2^{\mathrm{poly}(1/\epsilon)}}}$ time
- Via partitioning oracles: $2^{\mathrm{polylog}(1/\epsilon)}$ and simpler proof

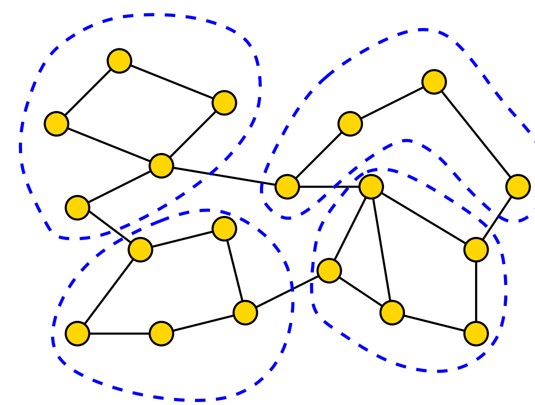
Application 2: Testing

Example: Testing planarity
(i.e., K_5- and $K_{3,3}$-minor-freeness)

Application 2: Testing

Example: Testing planarity
(i.e., K_5- and $K_{3,3}$-minor-freeness)

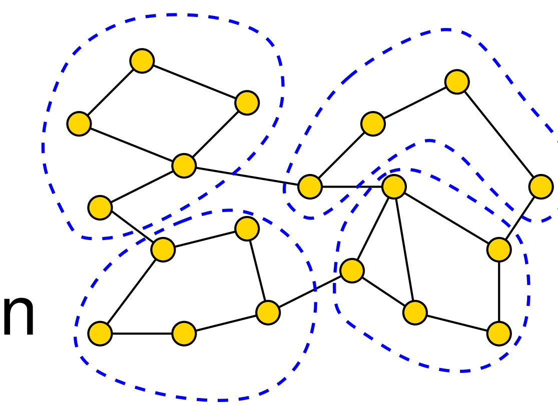
- Algorithm (given partitioning oracle for planar graphs that usually cuts $\leq \epsilon n/2$ edges):
  - Estimate the number of cut edges by sampling
  - If greater than $\epsilon n/2$, reject
  - Check a few random components if planar
  - If any non-planar found, reject otherwise, accept



Application 2: Testing

Example: Testing planarity
(i.e., K_5- and $K_{3,3}$-minor-freeness)

- Algorithm (given partitioning oracle for planar graphs that usually cuts $\leq \epsilon n/2$ edges):
  - Estimate the number of cut edges by sampling
  - If greater than $\epsilon n/2$, reject
  - Check a few random components if planar
  - If any non-planar found, reject otherwise, accept

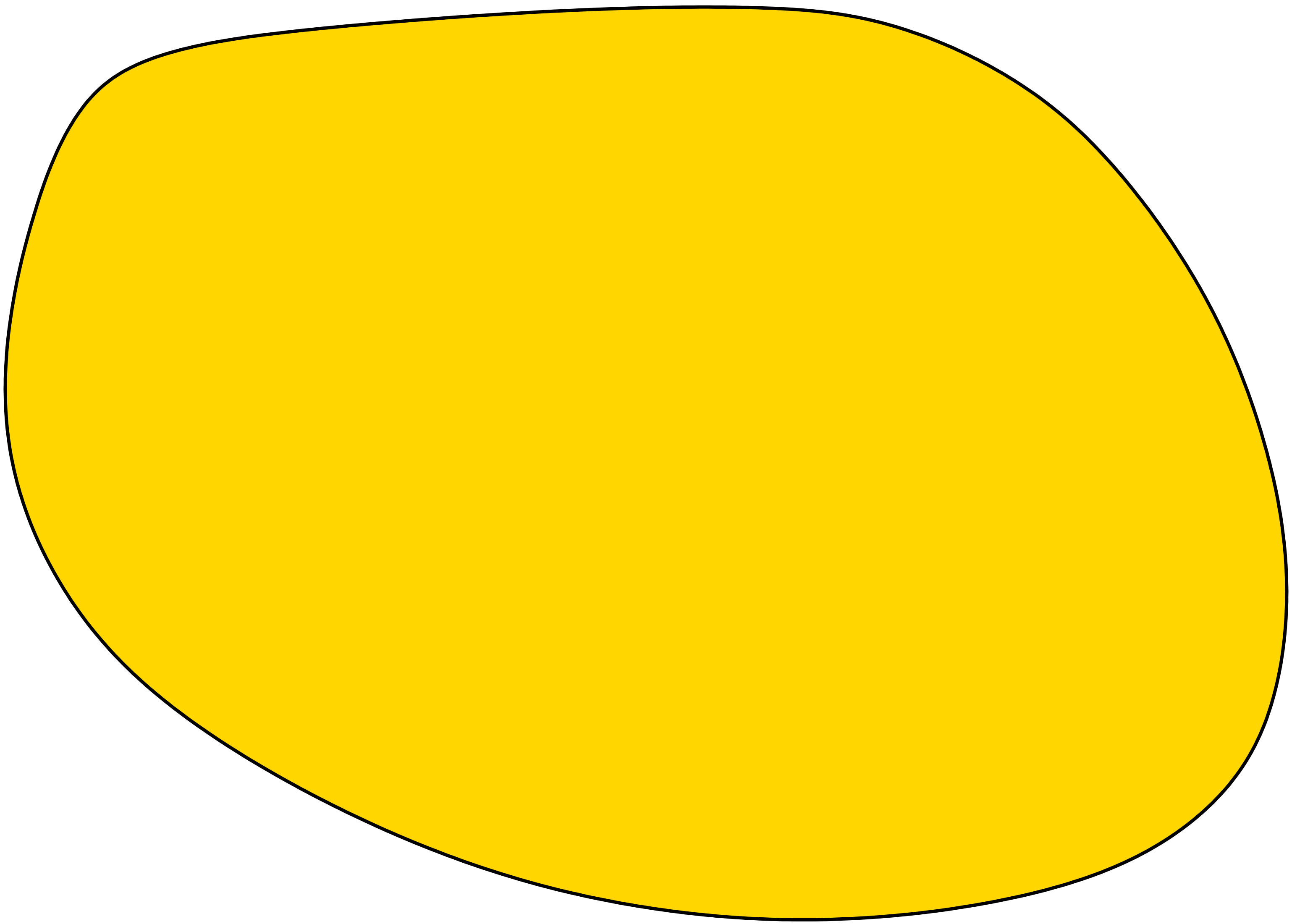


- Why it works:
  - planar: few edges cut in the partition
  - ϵ-far: either many edges cut or many copies of $K_{3,3}$ or K_5

Simplest Oracle

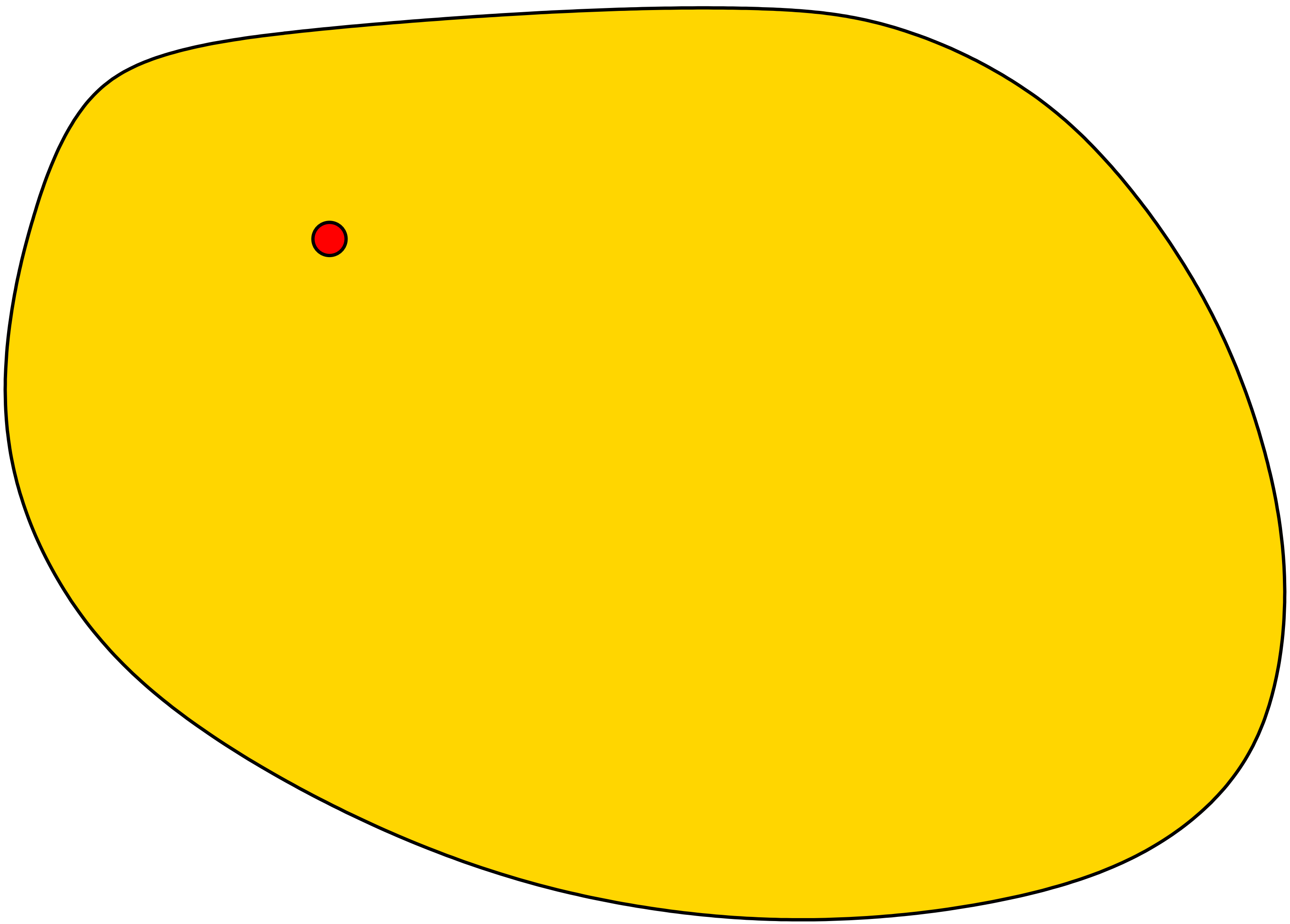
Iterative Procedure

Global procedure:



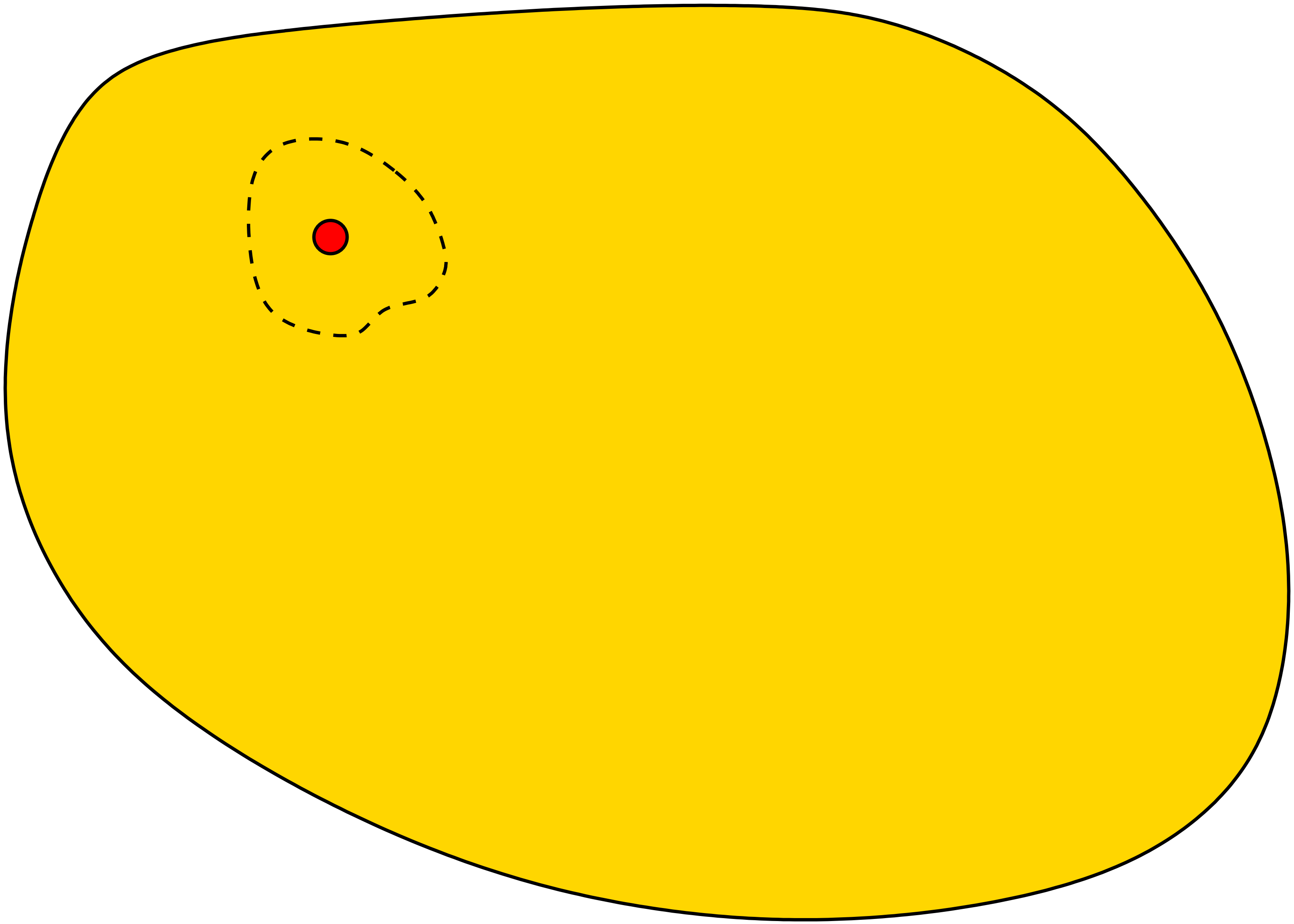
Iterative Procedure

Global procedure:



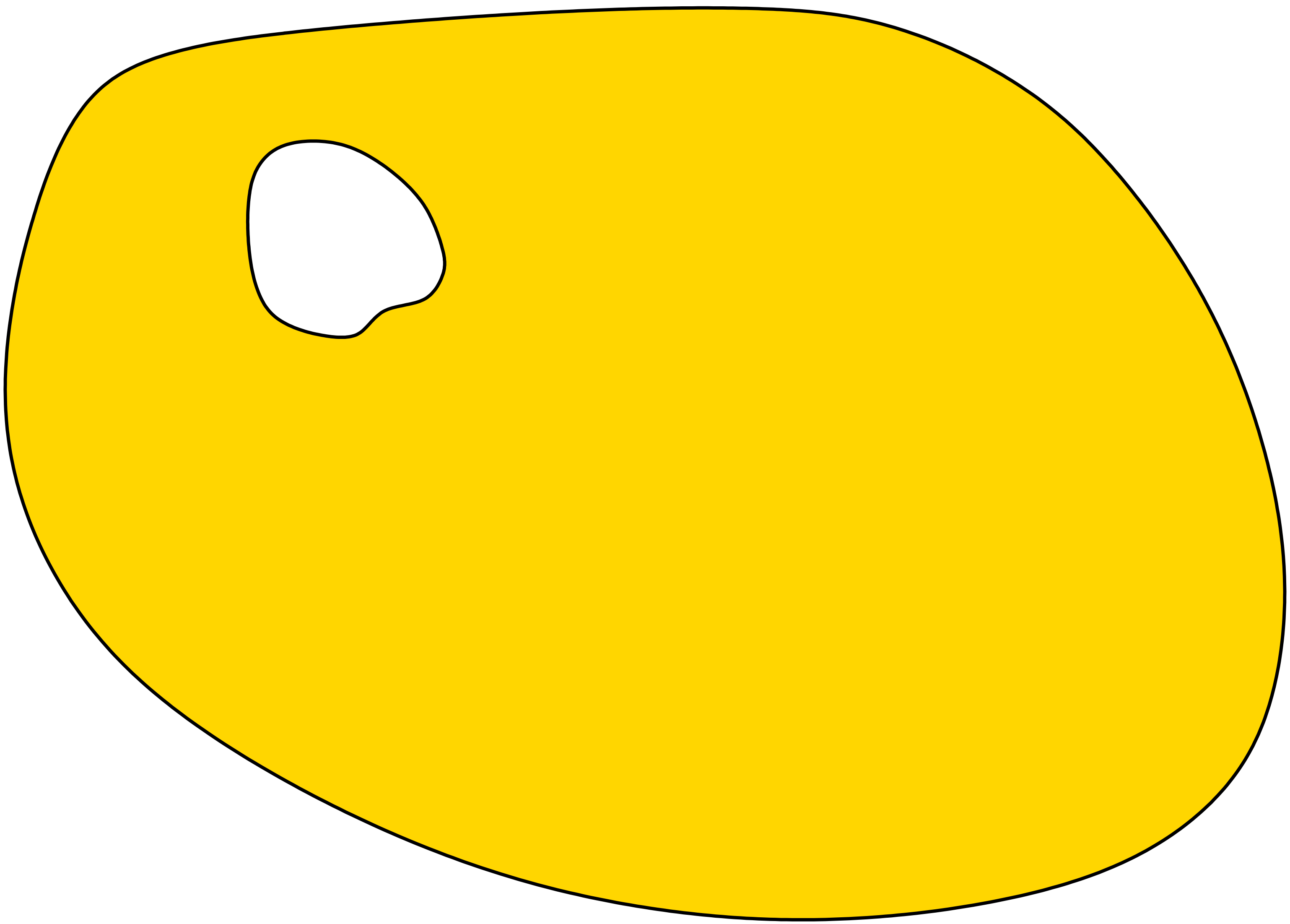
Iterative Procedure

Global procedure:



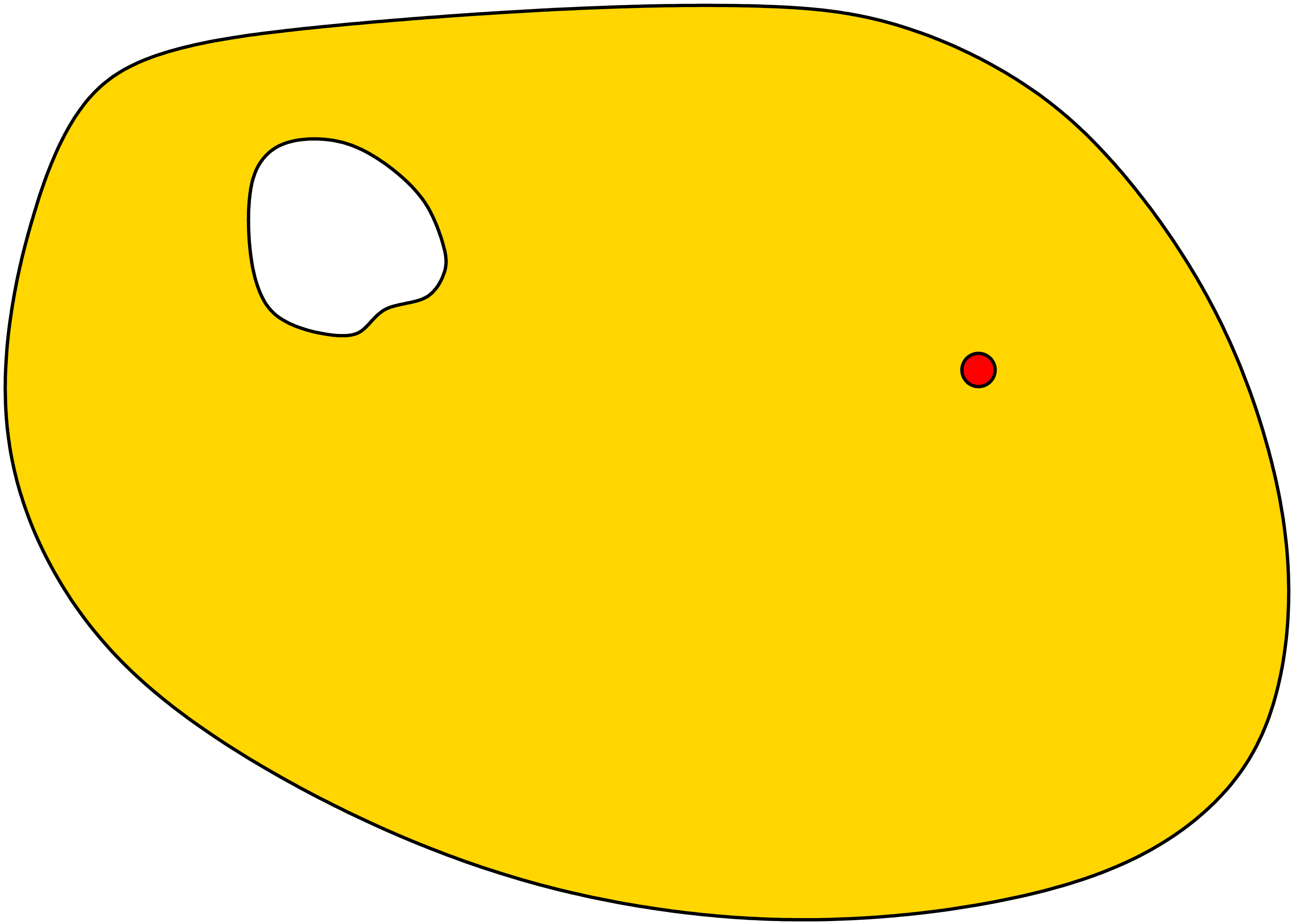
Iterative Procedure

Global procedure:



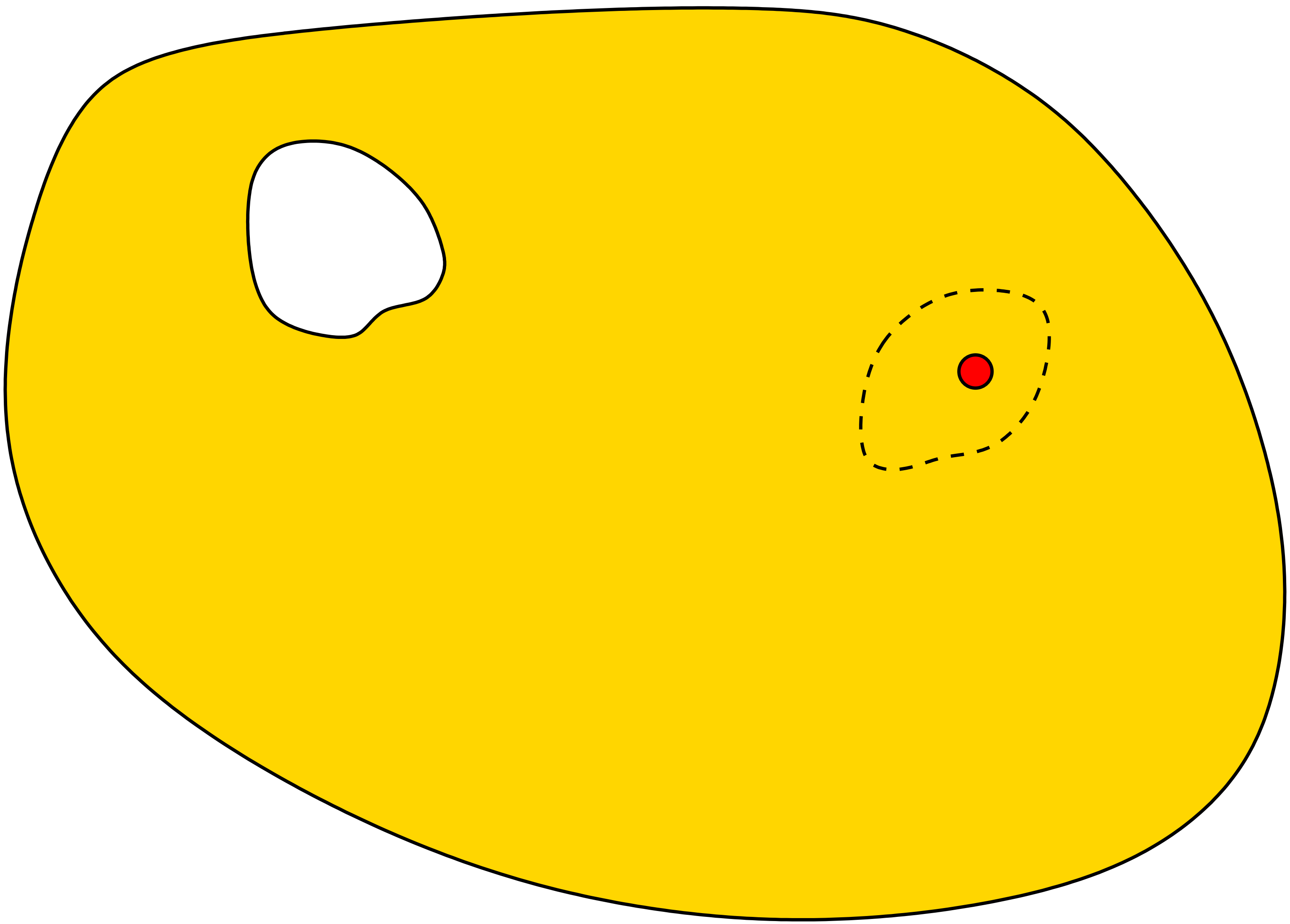
Iterative Procedure

Global procedure:



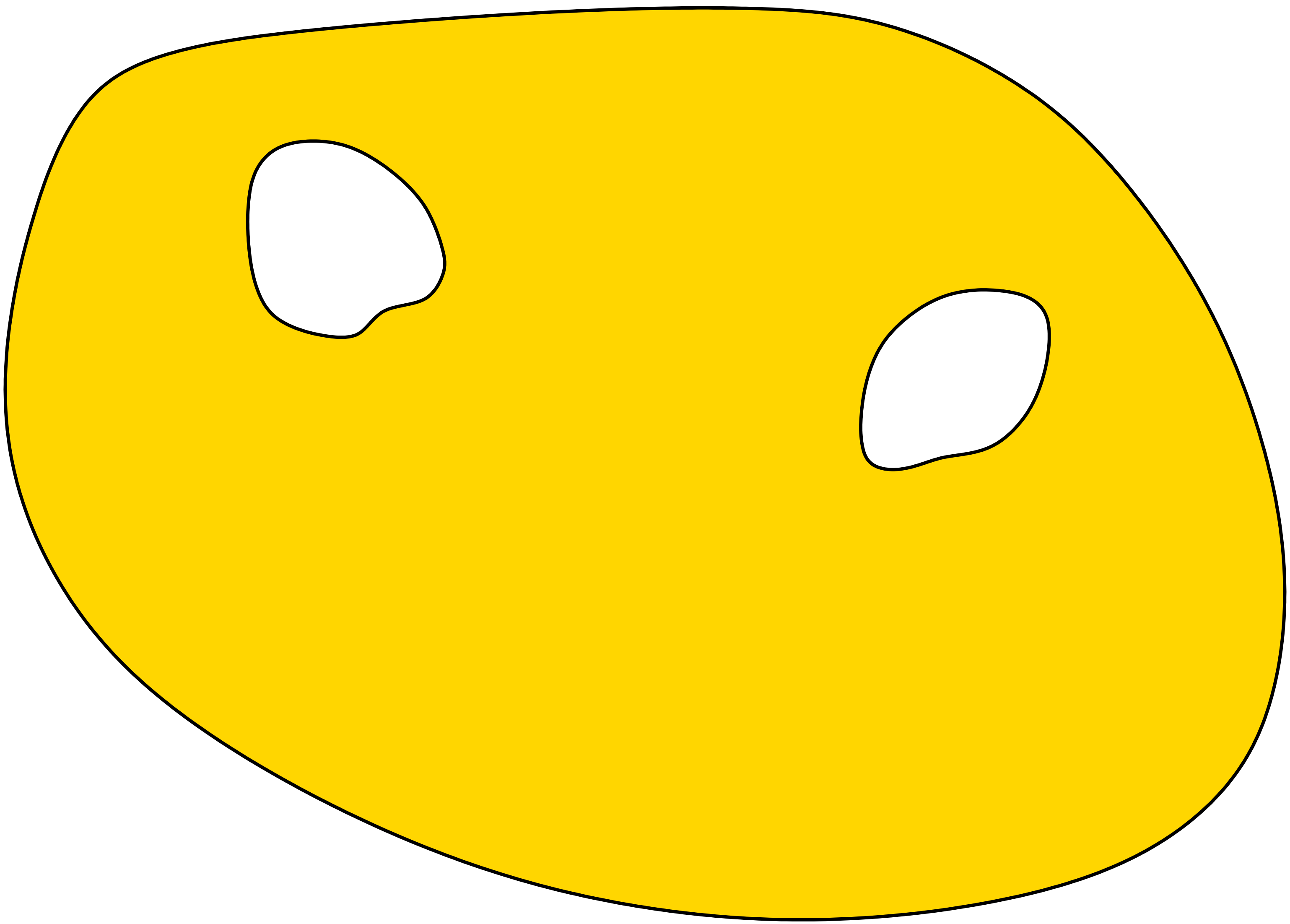
Iterative Procedure

Global procedure:



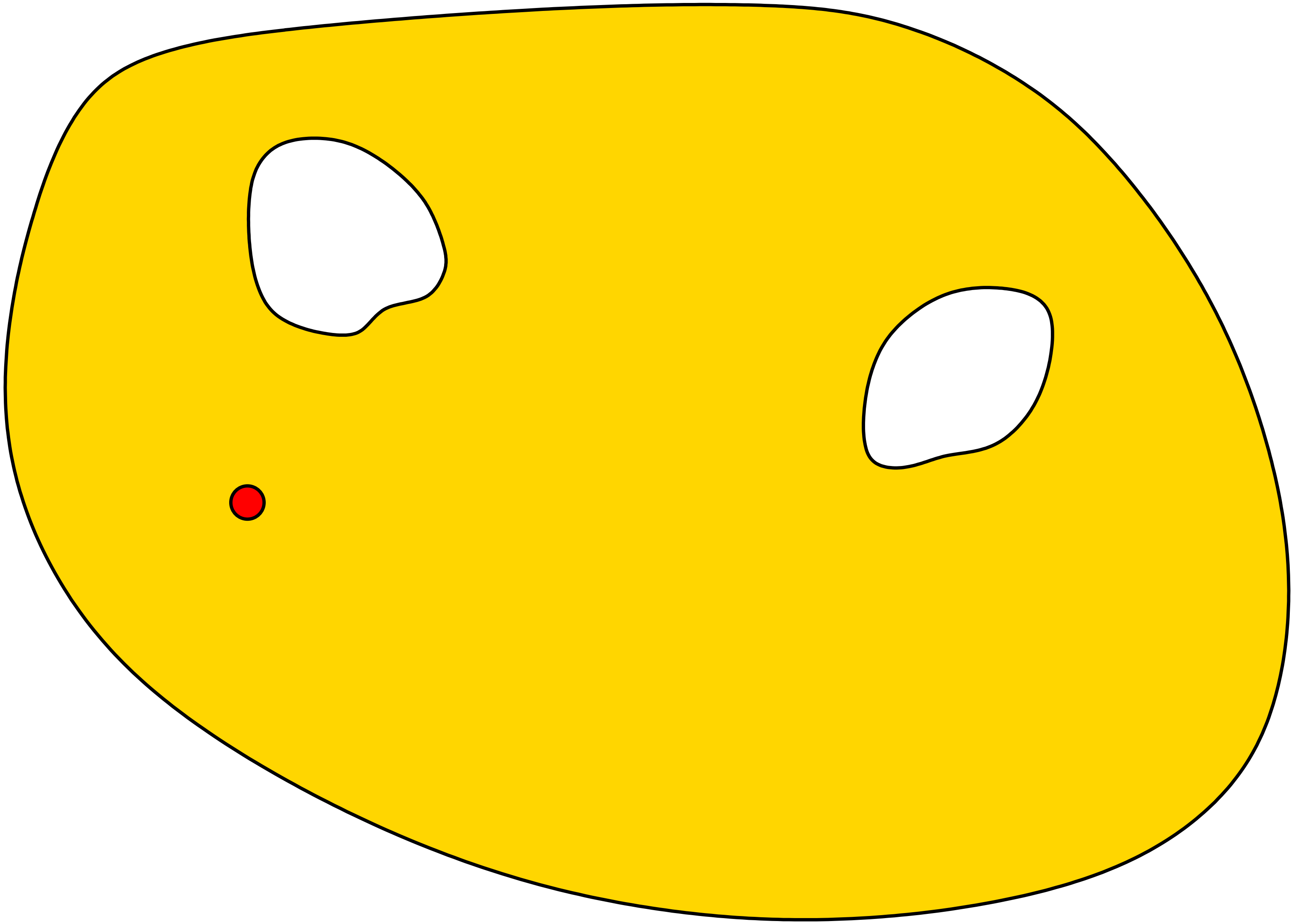
Iterative Procedure

Global procedure:



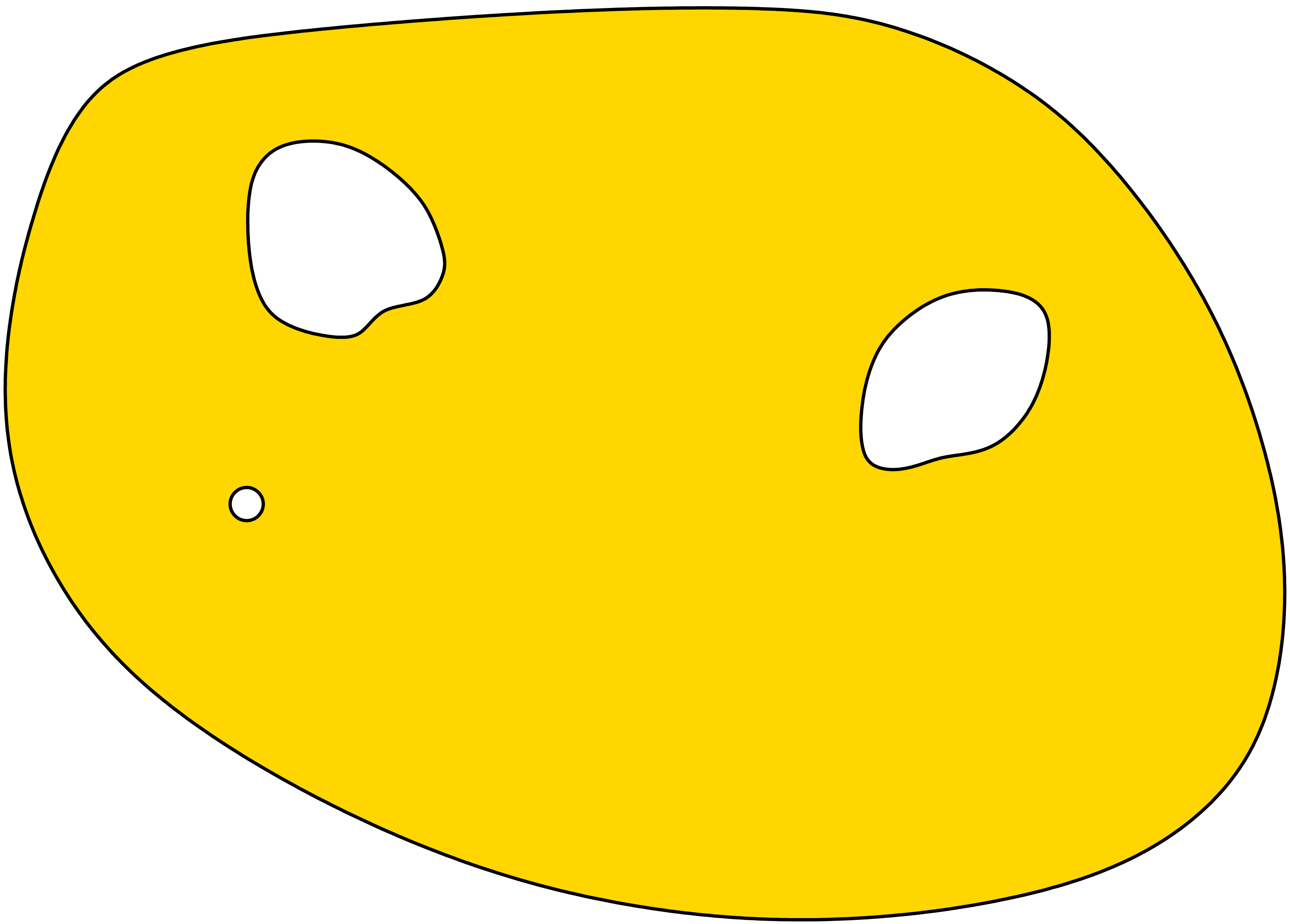
Iterative Procedure

Global procedure:

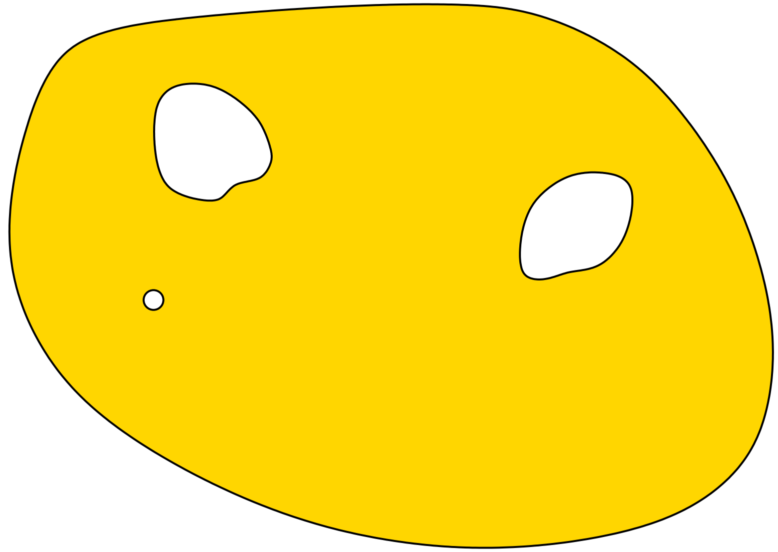


Iterative Procedure

Global procedure:



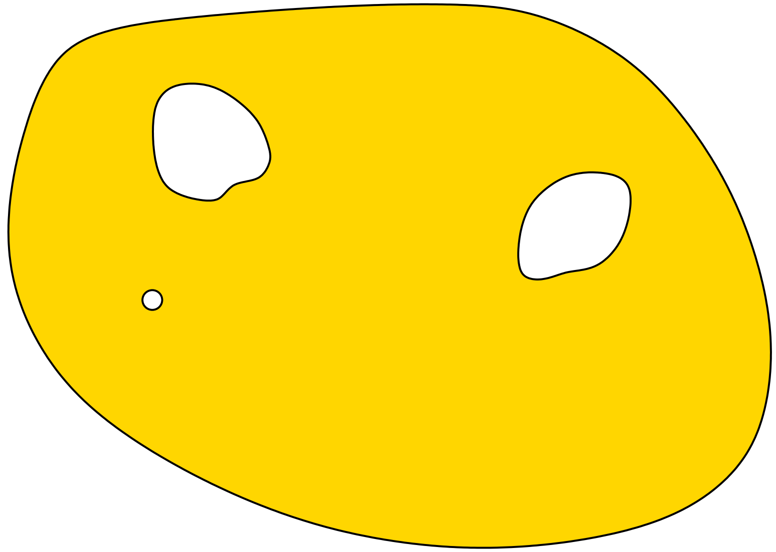
Local simulation



Use technique of Nguyen and O. (2008):

- Random numbers assigned to vertices generate a random permutation

Local simulation



Use technique of Nguyen and O. (2008):

- Random numbers assigned to vertices generate a random permutation
- To find a component of v:
  - recursively check what happened to close vertices with lower numbers
  - if v still in graph, try to construct a component

Open Questions

Open Questions

- Is there a $\mathrm{poly}(d/\epsilon)$-time algorithm for approximating maximum matching size up to $\pm\epsilon n$?

Open Questions

- Is there a $\mathrm{poly}(d/\epsilon)$-time algorithm for approximating maximum matching size up to $\pm\epsilon n$?
- Can planarity be tested in $\mathrm{poly}(d/\epsilon)$ time?