

# Random Graph Generation and Processing

Supervisor: Professor F. A. Novikov, Doctor of Technical Sciences  
Authors: Seraphim K. Ivanov, Roman P. Cherepanov



**СПБГЭТУ «ЛЭТИ»**  
ПЕРВЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ

**Goal:** Investigate algorithms on random graphs.

**Objectives:**

1. Develop an algorithm for random graph generation;
2. Conduct statistical analysis of generated graphs;
3. Analyze time efficiency of breadth-first and depth-first search;
4. Analyze time efficiency of minimum spanning tree algorithms;
5. Compare graph colouring algorithms for random graphs;

# Table of Contents

- 1 Problem Statement
- 2 Graph Generation
  - Erdős–Rényi Model
  - Gilbert Model
- 3 Graph Traversal Algorithms
  - Erdős–Rényi Model
  - Gilbert Model
- 4 Minimum Spanning Trees
- 5 Graph Colourings
- 6 References

# Random Graph Models

## Graph Model Description

Two popular random graph models are described in literature:

1. **Uniform model (Erdős-Rényi model)**  $G(p, q)$ : given  $p, q$  – number of vertices and edges. The graph is defined by uniformly random  $q$  edges from possible  $C(p, 2)$  [1].
2. **Binomial model (Gilbert model)**  $G(p, \pi)$ : given  $p, \pi$  – number of vertices and probability of each edge appearing in the graph [2].

The uniform and binomial models generate similar graphs as  $p$  increases.

- Average vertex degree in uniform model:  $2q/p$ ;
- Let  $\sigma$  be a random variable representing vertex degree,  $q_m = p(p - 1)/2$ .

Then by LLN:

$$G(p, \pi): \quad \mathbb{E}(\sigma) = \pi(p - 1), \quad q/q_m \xrightarrow[p \rightarrow \infty]{} \pi.$$

The binomial model is *much simpler* to study. In particular, vertex degree distribution can be easily calculated using the *binomial distribution*: each edge is inserted independently with fixed probability.

To guarantee graph connectivity, a random tree can be generated first and then enriched with edges. For tree generation, the *Prüfer code* is used.

- There exists a *bijection* between sequences  $\{a_i\}_{i=1..p-2}$ , where  $a_i \in \{1..p\}$ , and *labeled trees* with  $p$  numbered vertices;
- The sequence corresponding to a tree is called Prüfer code;
- The original tree can be reconstructed from the sequence using the *unpacking algorithm* [3, c. 304].
- Thus, to generate uniformly random trees, it's sufficient to generate uniformly random Prüfer sequences.

pruferTree:

**Input:** Number of vertices  $p \in \mathbb{N}$ .

**Output:** Tree  $T(V, E)$ , defined by edge set  $E$ , with vertices numbered  $1..p$ .

$t := \text{randSeq}(p - 2, p)$  // generate  $p - 2$  random numbers in range  $1..p$

$T := \text{unpack}(t)$  // obtain tree from code

**return**  $T$

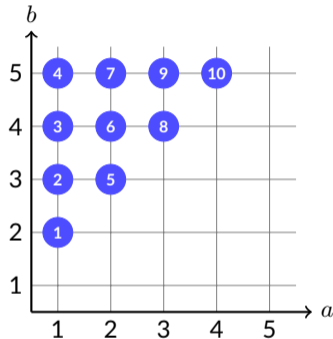
# Model $G(p, q)$

## Edge Generation (1/3)

1. In a complete graph with  $p$  vertices, there are  $p(p-1)/2 = C(p, 2)$  edges.
2. An edge can be represented as a unique *ordered pair* of incident vertices  $(a, b)$ , where  $a < b$ .
3. The set of such pairs can be ordered. The index  $i$  of pair  $(a, b)$  in the ordered list is its *index*.
4. The number of pairs with a fixed  $a$  and  $b > a$  is  $p - a$ .
5. The number of pairs with  $a' < a$  is obtained by summing:

$$\sum_{a'=1}^{a-1} (p - a') = \frac{(a-1)(2p-a)}{2}.$$

6. Within groups with a fixed  $a$ , the index shifts by  $b - a$ .



getIndex:

**Input:** edge  $(a, b) \in \{1..p\}^2 : a < b$ , number of vertices  $p \in \mathbb{N}$ .

**Output:** index of edge  $i \in \{1, p(p-1)/2\}$ .

**return**  $(a-1)(2p-a)/2 + (b-a)$

7. Conversely,  $a$  is the largest integer solution of the following system of inequalities:

$$\begin{cases} (a-1)(2p-a) < 2i, \\ a < p; \end{cases}$$

8. It can be found using binary search;  
9. Alternatively, by solving the equation

$$x^2 + (-2p-1)x + (2p+2i) = 0$$

for  $x$ . Then  $a = \lceil x_{\min} \rceil - 1$ , where  $x_{\min}$  is the smaller root.

10. The weakness of the second method lies in extracting the root during the solution search.

getEdge:

**Input:** index of edge  $i \in \{1..p(p-1)/2\}$ ,  
number of vertices  $p \in \mathbb{N}$ .

**Output:** edge  $(a, b) \in \{1..p\}^2 : a < b$ .

```
a := ceil( (2p+1-sqrt((2p+1)^2-4(2i+2p))) / 2 ) - 1
// [x_min] - 1
b := i + a - (a-1)(2p-a)/2
// offset recalculation
return (a, b)
```

# Model $G(p, q)$

## Edge Generation (3/3)

Introducing indices significantly simplifies edge generation.

Let  $X_p = \{x \in \mathbb{N} \mid x < p(p-1)/2\}$ . Then:

1. All  $p-1$  tree edges are mapped to *indices*  $T = \text{array}[1..p-1]$  of  $X_p$ ;
2. A *density*  $d$  is defined, from which we compute  $l = d \cdot p(p-1)/2$  – the number of new edges;
3.  $l$  elements are randomly selected from  $X_p$ ;
4. Array  $T$  is extended to length  $l$  by appending generated indices *without duplicates*.

**Note.** Despite the apparent simplicity of generation, the complexity of the `set` procedure is *quadratic* with respect to the number of vertices  $p$ . Moreover, a “fair” choose function (which does not require storing the entire set  $X_p$ ) is unavailable in most widely used programming languages. Therefore, `genGraph` requires  $\mathcal{O}(|X_p|)$  space and  $\mathcal{O}(p^2)$  time, excluding conversion overhead.

`genGraph`:

**Input:** Tree  $T$ : **array**[1.. $p-1$ ] of  $X_p$  as an index array, density  $d \in (0, 1]$

**Output:** Graph  $G$ , represented as an adjacency list

$l := \lfloor d \cdot p(p-1)/2 \rfloor$

**if**  $l > p-1$  **then** // select  $l$  elements from  $X_p$

$E := \text{choose}(l, X_p)$  // randomly select elements

$G := \text{set}(E, T)$  // merge  $E$  with  $T$

**end if**

**return** `indToList`( $G$ ) // convert indices to adjacency list

`set`:

**Input:** Graph  $G$ : **array**[1.. $p-1$ ] of  $X_p$ , edges  $E$ : **array**[1.. $l$ ] of  $X_p$

**Output:** Graph  $G$ , represented as an index array

**for**  $e \in E$  **do**

**if**  $e \notin G$  **then**

$G := G + e$

**end if**

**if**  $|G| = l$  **then return**  $G$

**end if**

**end for** // terminates by the pigeonhole principle

# Model $G(p, q)$

## Random pairs (1/3)

**Alternative approach.** One can generate the set of edges without using indices, by selecting vertex pairs at each step.

rndPairs:

**Input:** tree  $T$ : **array**[1.. $p$ ] of  $V$  as adjacency lists,  $d \in \mathbb{N}$  – number of edges to insert.

**Output:** Graph  $G$  represented as an adjacency list: **array**[1.. $p$ ] of  $V$

**while**  $d > 0$  **do**

$a := \text{rand}(1, p)$ ,  $b := \text{rand}(1, p)$

**if**  $a \neq b$  &  $T[b] \notin T[a].i$  **then**

        Append( $T[a].i$ ,  $T[b]$ )

        Append( $T[b].i$ ,  $T[a]$ )

$d := d - 1$

**end if**

**end while**

**return**  $T$

This algorithm uses no intermediate data structures, so its memory complexity is  $\mathcal{O}(1)$ .

# Model $G(p, q)$

## Random pairs (2/3)

Time complexity can be estimated based on the following considerations:

1. The probability of generating a loop is  $1/p$ ;
2. The probability of selecting an already existing edge is the current number of edges  $k$  divided by the maximum possible number  $p(p-1)/2$ ;
3. The number of trials until the first success follows a *shifted geometric distribution*:  
 $\xi \sim \text{Geom}(p)$ ,  $\mathbb{E}\xi = 1/p$ ;
4. If there are currently  $k$  edges, the probability of generating a suitable edge is:

$$\mathbb{P}(q_{k+1}) = 1 - \frac{1}{p} - \frac{2k}{p^2} = 1 - \frac{p + 2k}{p^2} = \frac{p^2 - 2k - p}{p^2}.$$

The total expected number of trials to generate edges from the  $p$ -th to the  $q$ -th:

$$\sum_{k=p}^q \frac{p^2}{p^2 - 2k - p}.$$

The number of edges to be inserted *can be upper-bounded* by  $q = p(p-1)/4$ .

# Model $G(p, q)$

## Random Pairs (3/3)

The maximum value of the term in the sum is achieved at  $k = q$ :  $\frac{p^2}{p^2 - p - p(p-1)/2} = 2 + o(1/p)$ .

1. We need to generate  $q - p + 1$  edges, and each generation *on average* takes up to two attempts;
2. Time complexity is  $\mathcal{O}(2q) + \mathcal{O}(2qL) + \mathcal{O}(qA)$ , where  $L$  and  $A$  are constants depending on the complexity of element search and list insertion;
3. The last edge will require about  $p^2/2$  attempts, the second-to-last  $- p^2/4$  attempts. This is a *harmonic series* scaled by a factor of  $p^2/2$ ;
4. The expected number of edge generations for the complete graph with  $q_m = p(p-1)/2$  edges from a tree with  $p-1$  edges is  $p^2/2 \cdot H(q_m - p + 1)$ ;
5. To estimate the number of operations for generating  $q < q_m$  edges, subtract the terms corresponding to edges  $q+1, q+2, \dots, q_m-1, q_m$ .

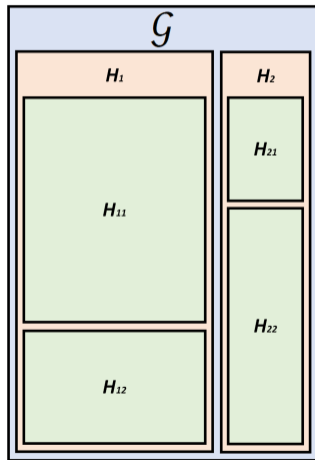
As a result, a tail of the finite harmonic series remains:

$$\frac{p^2}{2} \cdot \left( H(q_m - p + 1) - H(q_m - q) \right) \approx \frac{p^2}{2} \cdot \ln \frac{q_m - p + 1}{q_m - q}.$$

# Model $G(p, \pi)$ (1/3)

In the  $G(p, q)$  model, the question of *representativeness* was not addressed. Now let  $\mathcal{G}(p)$  denote the set of graphs with  $p$  vertices. Introduce the equivalence relations: edge isomorphism  $\cong$  and equality of edge count  $\sim$ .

1. The structure  $\mathcal{G}/\sim$  and  $\mathcal{G}/\cong$  allows for refinement  $(\mathcal{G}/\sim)/\cong$ ;
2. Let  $(\mathcal{G}/\sim) = \{H_i\}$ , and  $(H_i/\cong) = \{H_{ij}\}$ .  
Denote  $|H_i| = A_i$ ,  $|H_{ij}| = A_{ij}$ .
3. The probability of a graph falling into a factor set  $\mathcal{G}/\cong$  is:  
$$\pi(G \in H_{ij}) = \pi(G \in H_i) \cdot \pi(G \in H_{ij}|H_i) = \pi(G \in H_i) \cdot A_{ij}/A_i.$$
4. If  $\pi(G \in H_i) = A_i/|\mathcal{G}| = \sum_j A_{ij}/|\mathcal{G}|$  is chosen, then  
$$\pi(G \in H_{ij}) = A_{ij}/|\mathcal{G}|.$$



## Model $G(p, \pi)$ (2/3)

- The sizes of the classes are known:  $\pi(q = q_0) \sim \mathcal{G}(p, q_0) = C(q_m, q_0)$ ;
- Binomial coefficients grow rapidly. For example,  $C(5000, 2500) \approx 1.59 \cdot 10^{1503}$ .  
Simulating a discrete distribution with such weights is *difficult*;
- It is more convenient to switch to the *Gilbert model* with independent edge probability  $\pi_0$ ;
- Consider a graph with numbered edges. The probability of generating it is equal to  $\prod_{i=1}^{q_m} h(i)$ , where

$$h(i) = \begin{cases} \pi_0, & \text{if the edge with number } i \text{ is in } G(V, E), \\ 1 - \pi_0, & \text{otherwise;} \end{cases}$$

- When  $\pi_0 = 0.5$ , the probability of each graph is  $2^{-q_m}$ .

In the context of random graph construction, the following theorems for the Gilbert model are useful, enabling one to *bypass* the preliminary generation of spanning trees.

**Theorem 1.** If  $\pi > \ln p/p$ , then the graph is *asymptotically almost surely* connected. If  $\pi < \ln p/p$ , then the graph is a.a.s. disconnected. If  $\pi = \ln p/p$ , then the probability that the graph is connected a.a.s. is  $1/e$  [4].

**Example.** Suppose the graph represents telephone lines between cities. Further, assume  $\pi = 3 \ln p/p$  – the probability of stable communication between any two cities. Then the probability that the graph is connected is at least  $1 - 1/p$ . For example, for  $p = 1000$ , we have  $\pi = 0.021$ , but the probability of connectedness is 0.999.

**Theorem 2.** If the density  $d \geq \sqrt{2 \ln p/p}$ , then the graph a.a.s. has diameter 2. If the density  $d < \sqrt{2 \ln p/p}$ , then the graph a.a.s. has diameter greater than 2 [5].

# Model $G(p, \pi)$ (3/3)

All graphs are *equiprobable*, meaning the probability of generating an element belonging to the equivalence class  $H_{ij}$  under the relation  $\mathcal{G} / \cong$  is proportional to  $|H_{ij}|$ .

**Input:** Number of vertices  $p \in \mathbb{N}$ , edge probability  $\pi_0 \in (0, 1]$

**Output:** Connected graph

$G(V, E), |V| = p$

$V := \text{array}[1..p]$  of  $N$

**repeat**

$E := \{\}$

**for**  $i$  **from** 1 **to**  $p - 1$  **do**

**for**  $j$  **from**  $i + 1$  **to**  $p$  **do**

**if**  $r(\pi_0)$  **then**

$E := E + (i, j)$

**end if**

**end for**

**end for**

**until**  $\text{con}(V, E)$  // connectivity

**return**  $\{V, E\}$

1.  $r(\pi_0)$  – generates a random number from the Bernoulli distribution with  $p = \pi$ .

2. Apply **Theorem 1**:

$\pi = 0.5, p = 5000, 3 \ln p/p < 0.006 \implies$   
 $\pi_c \leq 2 \cdot 10^{-4}$ .

The probability of disconnectedness is sufficiently low.

3. Time complexity of the algorithm:

$\mathcal{O}(p^2) + \mathcal{O}(p + p^2) = \mathcal{O}(p^2)$ .

# Traversal Algorithms

Implementation [3, p. 258]

**Input:** Graph  $G(V, E)$  represented by adjacency lists  $\Gamma$ .

**Output:** Sequence of visited vertices.

```
for  $v \in V$  do  $x[v] := 0$  end for  
select  $v \in V$   
 $v \rightarrow T$  // select an arbitrary vertex  
 $x[v] := 1$  // mark it  
repeat  
   $u \leftarrow T$  // extract next vertex  
  yield  $u$   
  for  $w \in \Gamma(u)$  do  
    if  $x[w] = 0$  then // if not visited  
       $w \rightarrow T$  // add to data structure  
       $x[w] := 1$  // mark it  
    end if  
  end for  
until  $T = \emptyset$ 
```

1. The traversal type is determined by the data structure  $T$ : a stack results in depth-first search, a queue in breadth-first search;
2. The search algorithm starts by randomly selecting a start and a destination vertex. The algorithm stops once the destination is reached;
3. The number of marked vertices is used as a comparison metric for traversal strategies;
4. The geodesic distance between the start and destination vertices is tracked.

# Traversal Algorithms

Model  $G(p, q)$ . Statistical Analysis (1/2)

The study is conducted in the space  $(d, r, \log_2(\text{BFS/DFS}))$ .

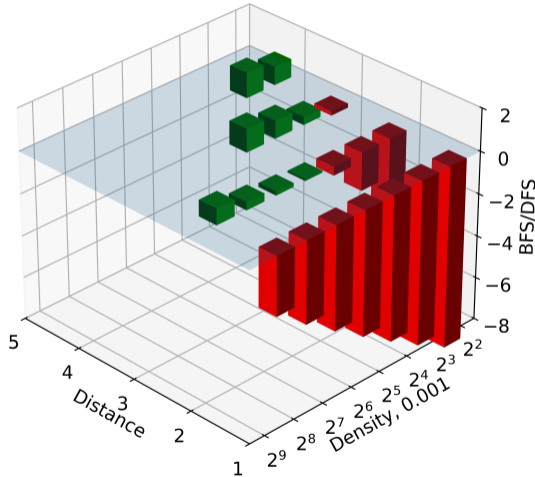
A rough averaging over all distances shows that the algorithms behave asymptotically the same.

**Observation:** despite extremely low densities, the average distance between vertices becomes less than 2 starting from a density of 0.032. This phenomenon is explained by **Theorem 2**.

$d$	$p$	avg BFS	avg DFS	avg $r$
0.0005	5000	2500.5	2499.8	11.26
0.001	5000	2498.9	2498.4	5.54
0.002	5000	2498.6	2503.1	3.95
0.004	5000	2500.9	2505.3	3.13
0.008	5000	2500.3	2499.7	2.71
0.016	5000	2502.3	2504.2	2.26
0.032	5000	2499.9	2504.5	1.97
0.064	5000	2496.8	2504.1	1.94
0.128	5000	2501.6	2499.1	1.87
0.256	5000	2505.4	2499.6	1.75
0.512	5000	2502.0	2500.6	1.49

# Traversal Algorithms

Model  $G(p, q)$ . Statistical Analysis (2/2)



At a *fixed density*, the efficiency of depth-first search increases with distance.

At a *fixed distance*, the efficiency of depth-first search increases with density.

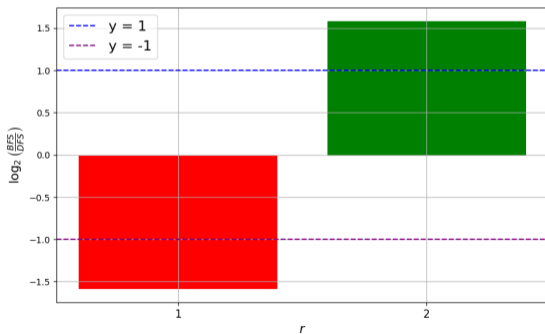
# Traversal Algorithms

Model  $G(p, \pi)$ . Statistical Analysis (1/4)

The analysis is conducted in the space  $(d, r, \log_2(\text{BFS}/\text{DFS}))$ .

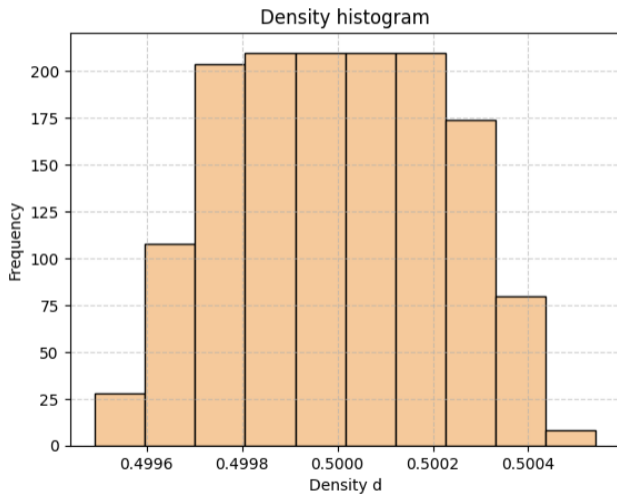
1. A total of 5716 connected graphs of the Gilbert model  $G(p = 5000, \pi_0 = 0.5)$  were generated;
2. On each graph, 500 iterations of both breadth-first and depth-first search were performed.

$r$	$p$	avg BFS	avg DFS
1	5000	1251.1	3749.8
2	5000	3750.7	1252.0
avg		2500.9	2500.9



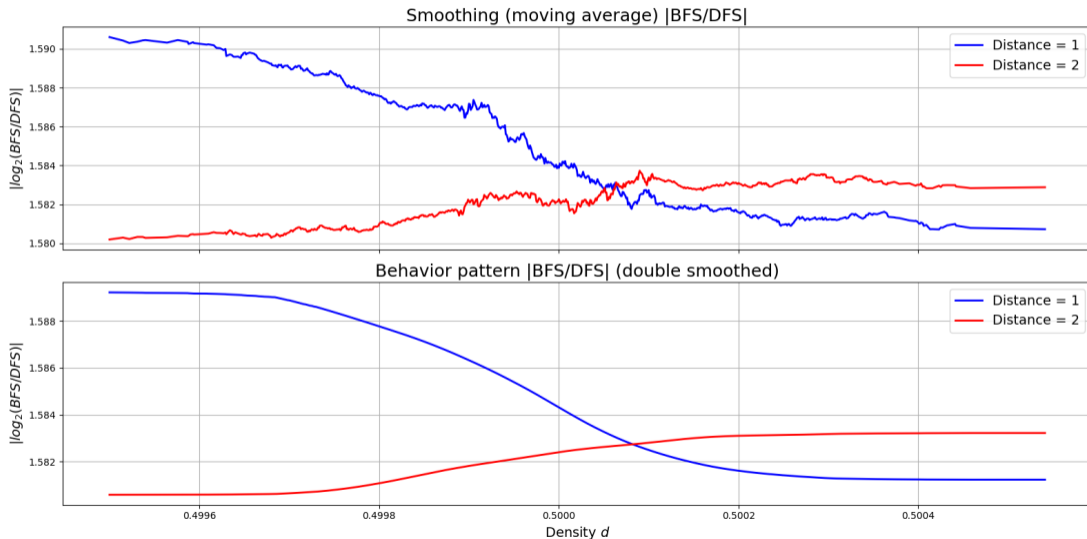
# Traversal Algorithms

Model  $G(p, \pi)$ . Statistical Analysis (2/4)



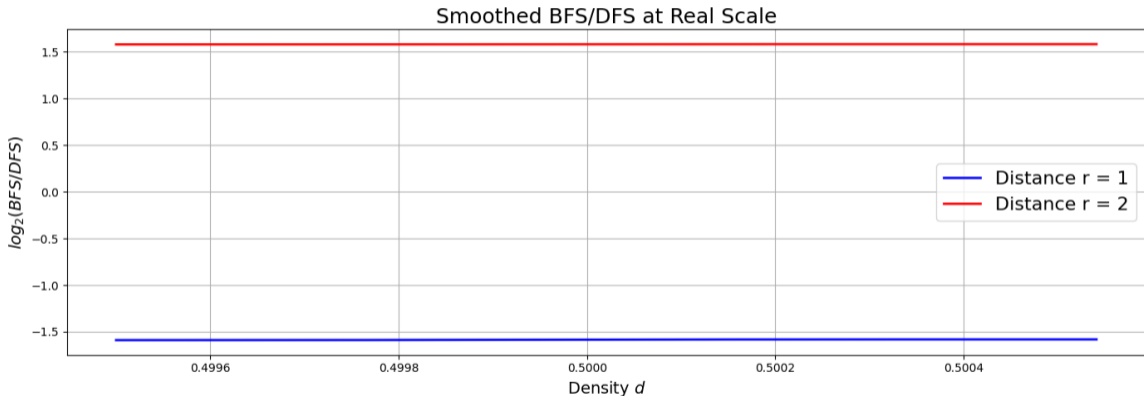
# Traversal Algorithms

Model  $G(p, \pi)$ . Statistical Analysis (3/4)



# Traversal Algorithms

Model  $G(p, \pi)$ . Statistical Analysis (4/4)



As a result of applying the Monte Carlo method to compare breadth-first and depth-first search algorithms, we observe:

1. As the distance increases, *at fixed density*, the BFS algorithm becomes more efficient than DFS;
2. As the density increases, *at fixed distance*, the BFS algorithm becomes less efficient than DFS;
3. For random graphs of models  $G(p, q)$  and  $G(p, \pi)$ , the second effect is almost negligible compared to the first.

# Spanning Trees

Prim's Algorithm [3, pp. 330] (1/3)

**Note.**  $\Gamma(u)$  – list of vertices adjacent to  $u$ .

● – critical points for performance evaluation.

prim:

**Input:** Graph  $G(V, E)$  represented by edge weight matrix  $C$

**Output:** Set  $T$  of edges of the minimum spanning tree

**select**  $u \in V$  //select arbitrary vertex

$S := \{u\}$  //  $S$  – set of tree vertices

$T := \emptyset$  //  $T$  – set of tree edges

**for**  $v \in V - u$  **do**

**if**  $v \in \Gamma(u)$  **then** ●

$\alpha[v] := u$  //  $u$  – closest tree vertex

$\beta[v] := C[u, v]$  //  $C[u, v]$  – edge weight

**else**

$\alpha[v] := 0$  //closest vertex unknown

$\beta[v] := \infty$  //distance unknown

**end if**

**end for**

**for**  $i$  **from** 1 **to**  $p - 1$  **do**

$x := \infty$  //initial value for search

**for**  $v \in V \setminus S$  **do**

**if**  $\beta[v] < x$  **then** ●

$w := v$  //found a closer vertex

$x := \beta[v]$  //and its distance

**end if**

**end for**

$S := S + w$  //add vertex to the tree

$T := T + (\alpha[w], w)$

**for**  $v \in \Gamma(w)$  **do**

**if**  $v \notin S$  **then** ●

**if**  $\beta[v] > C[v, w]$  **then** ●

$\alpha[v] := w$  //update closest vertex

$\beta[v] := C[v, w]$  //and the weight

**end if**

**end if**

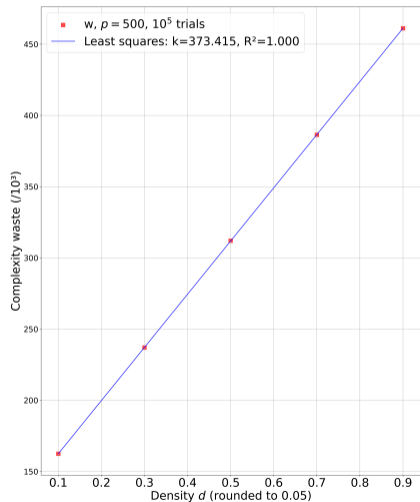
**end for**

**end for**

# Spanning Trees

## Prim's Algorithm (2/3)

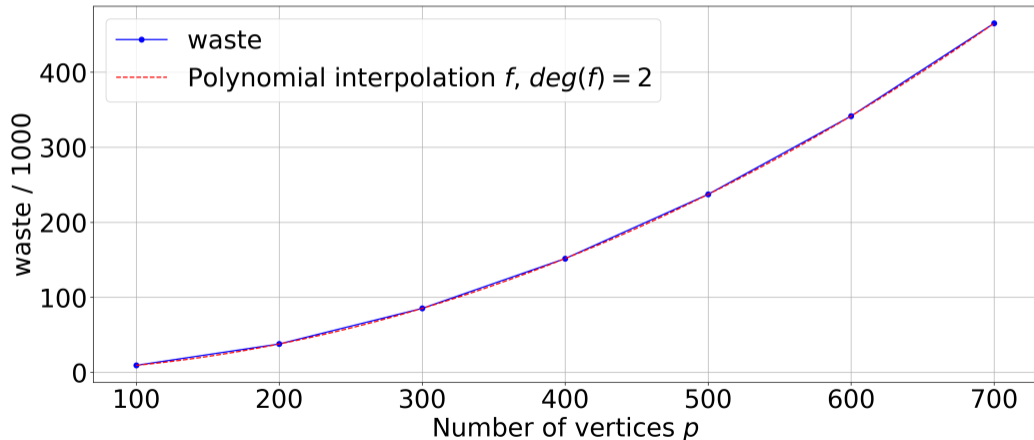
1. Theoretical complexity of Prim's algorithm is  $\mathcal{O}(p^2) = \mathcal{O}(q)$ ;
2. This suggests that:
  - 2.1 Prim's algorithm is more efficient on denser graphs;
  - 2.2 The complexity depends linearly on edge density  $\pi$  for fixed  $p$ :  $q = dp(p-1)/2$ .
3. The operations counted at the critical points have comparable complexity, which allows us to sum counters to estimate algorithm efficiency;
4. Numerical experiments were conducted on the  $G(p, \pi)$  model, with  $p = \{200, \dots, 500\}$ ,  $\pi = \{0.1, \dots, 0.9\}$ .



# Spanning Trees

## Prim's Algorithm (3/3)

The dependence of computational complexity on the graph size is expected and does not depend on the weight distribution:



# Spanning Trees

## Kruskal's Algorithm (1/3)

Kruskal's algorithm is implemented using a *disjoint set system* corresponding to the connected components of the MST graph.

**kruskal:**

**Input:** List  $E$  of edges  $(l, r) \in [1..p]^2$  of graph  $G$  with weights, sorted in ascending order.

**Output:** set  $T$  of edges of the minimum spanning tree.

$C := [\{1\}, \{2\}, \dots, \{p\}]$  //initialize connected components

**for**  $e \in E$  **do**

**if**  $C[e.l] \neq C[e.r]$  **then** ● //edge between components

$C := \text{mergeC}(C, e.l, e.r)$

$T := T + e$

**if**  $|T| = p - 1$  **then**

**return**  $T$

**end if**

**end if**

**end for**

**mergeC:**

**Input:** Disjoint set system  $C$  and two vertices  $l, r$  to be connected by an edge.

**Output:** Updated disjoint set system  $C$ .

$C[l] := C[l] + C[r]$  //enrich first component

$d := C[r]$  //remember second component

**for**  $c$  **in**  $C$  **do**

**if**  $c = d$  **then** ● //all copies of second component

$c := C[l]$  //replace

**end if**

**end for**

**return**  $C$

# Spanning Trees

## Kruskal's Algorithm (2/3)

1. The theoretical complexity of Kruskal's algorithm consists of two estimates:

1.1 Sorting complexity – in our case quicksort – is  $\mathcal{O}(q \log q)$ ;

1.2 The complexity of the main part strongly depends on the chosen *data structures*. In our case  $\mathcal{O}(q \cdot \alpha(p))$  where  $\alpha(p)$  is the *inverse Ackermann function*;

The final estimate is  $(C_1 \cdot q \log q + C_2 \cdot q \cdot \alpha(p))$ ;

2. **Assumption:** the efficiency of the second estimate correlates with the type of edge weight distribution;

3. Asymptotically the estimate

$$\mathcal{O}(q \log q) = \mathcal{O}(dp^2 \log dp);$$

4. The total estimate is well-known and not the focus of the study;

5. This work investigates the second part of the estimate for distributions:

5.1 Normal  $u \sim \mathcal{N}(\mu, \sigma^2)$ ,

5.2 Discrete uniform  $u \sim \mathcal{U}(l, r)$ ,

5.3 Constant  $w = 1$ ,

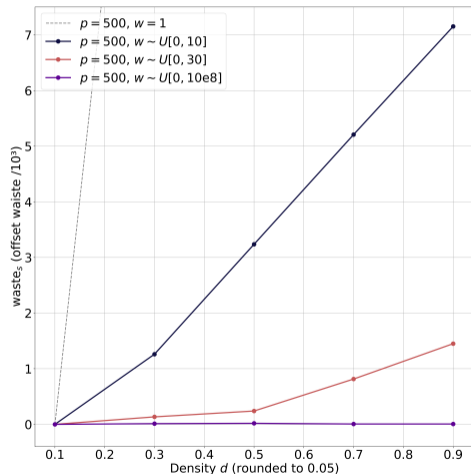
6. The Gilbert model  $G(p, \pi)$  is used with  $p \in \{100, 200, \dots, 500\}$ ,  $\pi \in \{0.1, 0.3, \dots, 0.9\}$ ;

7. **Note.** For the problem only variance  $\sigma^2$  and support size  $\text{supp}(\xi)$  of the weight distribution matter.

# Spanning Trees

## Kruskal's Algorithm (3/3)

1. On the ordinate axis, the shifted number of operations is plotted:  $\text{waste}[i] - \min \{ \text{waste} \}$
2. The "elbow" of the curve of complexity dependence on density is clearly visible in the plots;
3. This is not a statistical error: any such curve has a bend;
4. **Observation:** increasing the number of unique weights in the graph significantly reduces the complexity of the main loop of Kruskal's algorithm;
5. For  $p = 500$  the complete graph has  $q = 124750$ , and for a graph with density  $\pi = 0.2$  we have  $q = 24950$  which is 3 orders of magnitude larger than the support size of  $\mathcal{U}[0, 30]$ ;
6. With further increase of  $r$  the curve approaches the x-axis, i.e. the complexity of the algorithm *loses dependence* on density!



# Spanning Trees

## Auxiliary Experiment

Kruskal's algorithm processes edges of the *graph* whose weights do not exceed the weight of the heaviest edge in the *tree*. To estimate their number, the following experiment was conducted:

1. Construct a random graph in the Gilbert model  $G(p, \pi)$ .
2. Assign random *real-valued* weights to the edges.
3. Build the spanning tree  $T(G)$ .
4. Find the heaviest edge in the tree  $e_t \in E(T)$ .
5. Determine the number of edges  $c_w$  in the original graph whose weights are less than the weight of  $e_t$ .

The parameter  $c_w$  serves as an estimate of the number of edges processed by the algorithm. For each density  $\pi \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ , 500 experiments were conducted on graphs with  $p = 500$ . The results are shown in the table:

$\pi$	0.1	0.3	0.5	0.7	0.9
$\bar{c}_w$	1705.58	1658.972	1684.958	1695.988	1697.906

As a result of the numerical experiments:

- For Prim's algorithm, it was established that the operation count estimate grows linearly with the number of edges in the graph;
- For Kruskal's algorithm, an interesting dependence was found between the number of processed edges and the distribution of their weights. For sufficiently large supports, the number of processed edges does not depend on the number of edges in the graph.

Further research could examine more weight distributions to clarify the rate at which Kruskal's algorithm loses dependence on the number of edges in the graph as the support size increases.

# Graph Colouring

Naive Colouring and Colouring with Heuristic [3, pp. 359, 360]

Algorithms for naive and greedy colourings are shown. The greedy algorithm colours vertices in descending order of their degrees.

**Input:** Graph  $G$ .

**Output:** Graph colouring – an array  $C$  : array [1.. $p$ ]  
of 1.. $p$

```
for  $v \in V$  do
   $C[v] := 0$  //all vertices are uncoloured
end for
for  $v \in V$  do
   $A := \{1, \dots, p\}$  //available colours
  for  $u \in \Gamma(v)$  do
     $A := A \setminus \{C[u]\}$  //used colours
  end for
   $C[v] := \min A$ 
end for
```

**Input:** Graph  $G$ .

**Output:** Colouring  $C$  : array [1.. $p$ ] of 1.. $p$

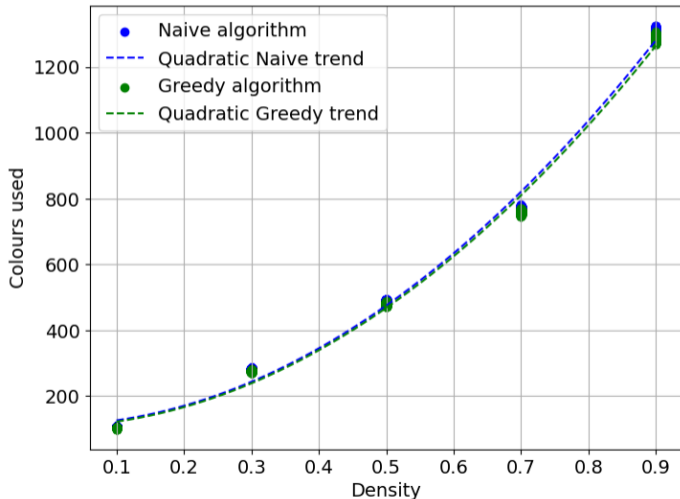
```
Sort( $V$ ) //by non-increasing  $\sigma(v)$ 
 $c := 1$  for  $v \in V$  do  $C[v] := 0$  end for
while  $V \neq \emptyset$  do //vertices without colours remain
  for  $v \in V$  do
    for  $u \in \Gamma^+(v)$  do
      if  $C[u] = c$  then //neighbor with colour  $c$ 
        next for  $v$ 
      end if
    end for
     $C[v] := c; V := V \setminus \{v\}$  //colour vertex  $v$ 
  with  $c$ 
  end for
   $c := c + 1$ 
end while
```

- In both algorithms, the main operation is checking the colours of a vertex's neighbors. For this reason, no significant efficiency difference is expected apart from the sorting overhead.
- It is interesting to compare the accuracy of the algorithms without computing the exact minimal colouring – it is enough to compare the number of colours used on the same random graph.
- Another research direction is to analyze how many colours are needed to colour a graph depending on its density.

The experiment used the  $G(p, \pi)$  model with  $p = 5000$  and  $\pi \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ . For each density, 10,000 graphs were generated, and each graph was coloured by both methods.

# Graph Colouring

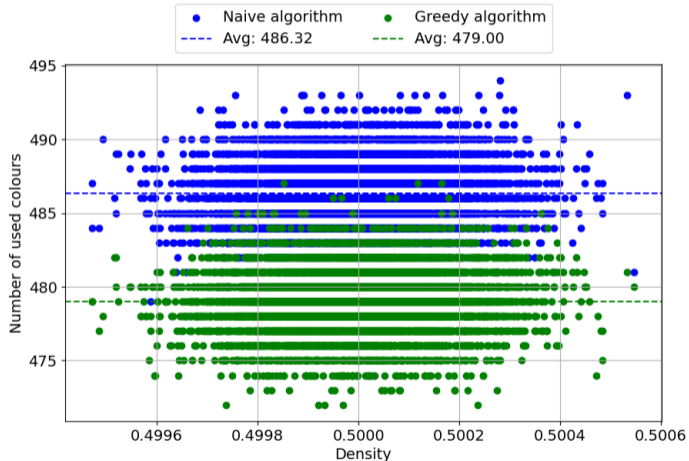
## Accuracy Comparison (1/3)



1. The algorithms show very similar results, with a slight advantage for the greedy algorithm.
2. It was not possible to reliably determine the growth rate of the number of required colours based on only 5 points.

# Graph Colouring

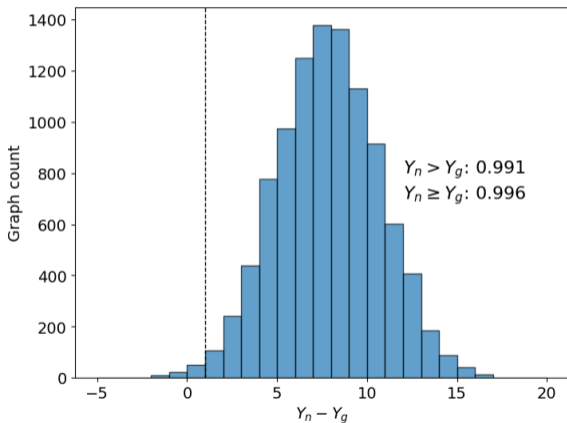
## Accuracy Comparison (2/3)



1. At density 0.5 (where the sample is representative), the greedy algorithm shows slightly better results.
2. The numbers of colours required differ only slightly, with some overlap in the data points.

# Graph Colouring

## Accuracy Comparison (3/3)



1. In rare cases, the greedy algorithm performs no better, or even worse, than the naive algorithm.
2. The maximum loss of the greedy algorithm (5 colours) was smaller than its maximum gain (20 colours).

Based on the comparison of the two algorithms:

- On average, the greedy algorithm finds a colouring using slightly fewer colours;
- In rare cases, the greedy algorithm yields a colouring with more colours than the naive algorithm.

Future research could explore a wider range of densities to better estimate how the number of colours required by approximate colouring algorithms grows as the number of edges in the graph increases.

- [1] P. Erdős и A. Rényi. “On random graphs”. В: *Publicationes Mathematicae* 6.3-4 (нояб. 1958), с. 290–297. DOI: 10.5486/PMD.1959.6.3-4.12. URL: <https://doi.org/10.5486/PMD.1959.6.3-4.12>.
- [2] E. N. Gilbert. “Random Graphs”. В: *The Annals of Mathematical Statistics* 30.4 (1959), с. 1141–1144. DOI: 10.1214/aoms/1177706098. URL: <https://doi.org/10.1214/aoms/1177706098>.
- [3] Новиков Ф. А. *Дискретная математика для программистов: Учебник для вузов. 3-е изд.* Питер, 2009, с. 384. ISBN: 978-5-91180-759-7. URL: <https://stugum.wordpress.com/wp-content/uploads/2014/03/novikov.pdf>.
- [4] Райгородский А. М. *Модели случайных графов.* МЦНМО, 2011, с. 42–46. ISBN: 978-5-94057-840-6. URL: <https://old.mccme.ru/free-books/dubna/raigor-4.pdf>.

- [5] A. Blum, J. Hopcroft и R. Kannan. *Foundations of Data Science*. Cambridge University Press, 2018, с. 256–258. ISBN: 9781108617369. URL: <https://www.cs.cornell.edu/jeh/book.pdf>.