# Programming Assignment 2 (due 3/17)

DS-563 / CS-543 @ Boston University

Spring 2023

## Before you start...

**Collaboration policy:** You may verbally collaborate on programming assignments, however, you must write your code and final report independently, i.e., without seeing other students' solutions. If you choose to collaborate on a problem, you are allowed to discuss it with at most **four** other students currently enrolled in the class.

The header of each assignment you submit must include the field "Collaborators:" with the names of the students with whom you have had discussions concerning your solutions. A failure to list collaborators may result in a credit deduction.

You may use external resources such as textbooks, lecture notes, and videos to supplement your general understanding of the course topics. You may use references such as books and online resources for well known facts. However, you must always cite the source.

You may **not** look up solutions to a programming assignment in the published literature or on the web. You may **not** share written work with anyone else. If you wish to make your code public (for instance, by making it publicly available on GitHub or GitLab), please wait till the end of the semester. (If you want to publish it earlier for whatever reason, please check with us first.)

**Submitting:** Your solution is to be submitted via Gradescope (entry code: 4VYBJ6). In particular, you should submit a pdf with your project report and a zip file with your code (or an equivalent of these as long as it's allowed by Gradescope). Don't forget to provide information how to obtain your data sets.

**Late policy:** No extensions. Submitting a solution one school day late (i.e., on Monday) may result in a deduction of 10% of points.

## Local maximal matching computation

In this programming assignment, we explore the complexity of different versions of a graph exploration method for deciding whether an edge is included in a maximal matching. As a reminder a *matching* in a graph is a subset of the graph's edges in which no two edges share an endpoint. A *maximal matching* is a matching that cannot be extended by adding another edge, i.e., all remaining edges share an endpoint with at least one edge that is already included in the matching.

First, consider the following randomized greedy algorithm for maximal matching:

**Algorithm 1:** Computing a random greedy maximal matching

```
(e_1, ..., e_m) ← a random permutation of all edges
M ← ∅
foreach i ∈ {1, ..., m} do
    if e_i shares no endpoint with any edge in M then
        M ← M ∪ {e_i}
return M
```

It's not difficult to show that it outputs a maximal matching. (Why?) This algorithm selects a random permutation, which requires a global view of the entire graph. Instead of that, we could assign a random number $r_e \in [0, 1]$ to every edge $e \in E$ independently, uniformly at random. The ordering of edges $e$ according to the their numbers $r_e$ is a random permutation and if we want to know whether an edge $e'$ should be considered before an edge $e''$, it suffices to check whether $r_{e'} < r_{e''}$.

Suppose now that we want to find out whether a given edge was selected to the maximal matching. We could use the following procedure, which recursively checks if it is blocked from entering the maximal matching by any of the adjacent edges that appear before it in the corresponding random permutation.

**Algorithm 2:** selected1$(e)$

```
selected ← true
foreach edge e' adjacent to e do
    if r_{e'} < r_e then
        if selected1(e') then
            selected ← false
return selected
```

This procedure may not seem very efficient because it keeps considering adjacent edges even after it discovers an edge that prevents it from ending up in the maximal matching. We can try to optimize this by stopping after discovering an adjacent edge in the maximal matching.

**Algorithm 3:** selected2$(e)$

```
(e_1, ..., e_k) ← a random permutation of edges adjacent to e
foreach i ∈ {1, ..., k} do
    if r_{e_i} < r_e then
        if selected2(e_i) then
            return false
return true
```

Perhaps an even better idea is to consider the adjacent edges in increasing order of their assigned random numbers. The intuition is that edges with lower assigned numbers should be more likely to be in the maximal

matching.

**Algorithm 4:** selected3($e$)

```
(e_1, ..., e_k) ← edges e' adjacent to e in increasing order of r_{e'}
foreach i ∈ {1, ..., k} do
    if r_{e_i} < r_e then
        if selected3(e_i) then
            return false
return true
```

## Your task

1. Implement the above exploration methods (selected1, selected2, and selected3) and at least one variation developed by you. You are free to do whatever you want, but describe the rationale behind your version in the report. For instance, you could store already computed values so you never recompute the solution for any edge if you visit it again.

   *Sanity check:* Verify the correctness of your implementation of the exploration methods. For instance, consider a few simple graphs with fixed sequences of numbers $r_e$ to see if your implementation gives expected results that match Algorithm 1.

2. Implement at least two methods for generating graphs on $n$ vertices that result in vertex degrees roughly $d$. For instance, you could use the following methods:

   (a) The graph is the union of $d$ random matchings, each of size $\lfloor n/2 \rfloor$.

   (b) Assign a random point in Euclidean square $[0, 1]^2$ to every vertex. For each vertex $v$, add edges from $v$ to $d$ vertices corresponding to the closest points in $[0, 1]^2$.

   *Note:* You might want to remove paralell edges but if you don't remove them, make sure that your implementation of the exploration methods handles them correctly.

   *Sanity check:* Verify the correctness of your graph generation procedure. Are degrees of vertices roughly $d$? For small $n$ and $d$, you may want to draw the graph and see if it looks as expected.

3. Run experiments to see how ~~these methods~~the exploration methods you implemented in Part 1 behave as a function of $d$. More precisely, for each exploration method and a graph of each vertex degree, what is the average number of *recursive calls* when starting from a random edge? Please include the results you obtain here, i.e., the average observed number of recursive calls, as a table with different values of $d$ as different rows.

   *Note:* Make sure to repeat the experiment many times with different starting edges and different settings of numbers $r_e$. Also make sure that $n$, the number of vertices, is sufficiently large so you don't see any unusual behavior due to the size of the graph.

4. Plot a graph visualizing the average number of recursive calls as a function of $d$ for each class of graphs and each graph exploration method.

5. What is the average number of calls you see as a function of $d$ for each class of graphs and each version of the exploration method? Does it look logarithmic, linear, quadratic, polynomial, exponential, etc.? Do you have any other interesting insights into their complexity?

### Additional notes

- Instead of assigning random real numbers $r_e$ in $[0, 1]$, you can generate random integers from a sufficiently large range so that seeing a pair of adjacent edges with the same assigned integer is unlikely.
- You don't have to generate all numbers $r_e$ in advance. You can speed up your experiments by generating them on the fly when they are needed and then storing them.

## Deliverables

Your final submission should include:

- Code:
  - Provide all the code you wrote for this assignment to run the described experiments.
  - Make your code readable (use reasonable variable and function names, etc.).
- Report:
  - It should include all important implementation details and discuss implementation decisions. In particular, explain how you chose to represent graphs.
  - It should address all required questions from the previous section. Use tables and/or graphs to display numerical results of your experiments.