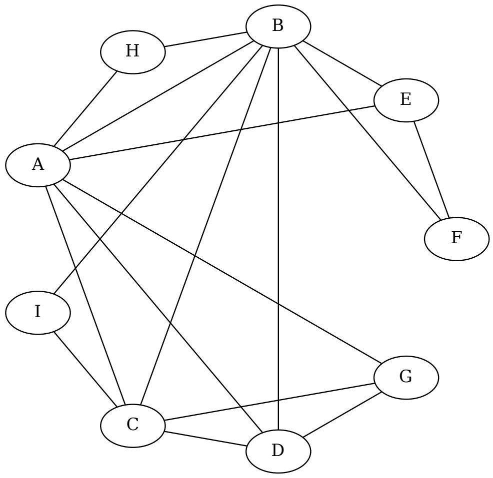
TD 1: Introduction

1 Adjacency Matrices and Lists

Give the adjacency matrix representation and the adjacency list representation of the graph below. Calculate the degree of each vertex.



Solution:

Adjacency Matrix:

	A	B	C	D	E	F	G	H	I
A	0	1	1	1	1	0	1	1	0
B	1	0	1	1	1	1	0	1	1
C	1	1	0	1	0	0	1	0	1
D	1	1	1	0	0	0	1	0	0
E	1	1	0	0	0	1	0	0	0
F	0	1	0	0	1	0	0	0	0
G	1	0	1	1	0	0	0	0	0
H	1	1	0	0	0	0	0	0	0
I	0	1	1	0	0	0	0	0	0

Adjacency lists and degrees:

A	B,C,D,E,G,H	6
B	A,C,D,E,F,H,I	7
C	A,B,D,G,I	5
D	A,B,C,G	4
E	A,B,F	3
F	B,E	2
G	A,C,D	3
H	A,B	2
I	B,C	2

2 O-notation reminder

Sort the following functions of n in a table where, whenever $f(n) = \Theta(g(n))$ you place f, g on the same row, and whenever $f(n) = o(g(n))$ you place f below g.

$$n^2 + 15n, \frac{n^3}{2}, \log^5 n, (\log n)^{\log n}, \log(n!), 3n \log n, 2^{\sqrt{\log n}}, n^{\log n}, \binom{n}{2}$$

Solution:

- $n^{\log n}$
- $(\log n)^{\log n}$
- $\frac{n^3}{2}$
- $n^2 + 15n, \binom{n}{2}$
- $\log(n!), 3n \log n$
- $2^{\sqrt{\log n}}$
- $\log^5 n$

3 Graph Square

If $G = (V, E)$ is a graph, we define as the square of G, denoted G^2 the graph which has the same vertex set as G and in which two vertices u, v are adjacent if and only if they are at distance at most 2 in G. (This means that u, v are adjacent in G^2 if they are adjacent in G or they have a common neighbor in G.)

Give an algorithm that takes as input G (in matrix or list representation) and outputs G^2 (in the same representation). What is the time complexity of your algorithm?

Solution:

Adjacency Matrix: We are given the adjacency matrix A of G. We observe that the matrix $B = A^2$ has the following property: $B[i, j] \neq 0$ if and only if i, j have a common neighbor in G, that is, there exists k such that $A[i, k] = A[k, j] = 1$. Our algorithm is then the following:

- Compute $B = A^2$.
- For each $i, j \in \{1, \dots, n\}$ place an edge between i, j in G^2 if (i) $A[i, j] = 1$ or (ii) $B[i, j] \neq 0$

The complexity of this algorithm is dominated by the matrix multiplication step. The obvious way to perform this step has time complexity $O(n^3)$, but there exist more sophisticated algorithms, with the current best being slightly better than $O(n^{2.4})$.

Adjacency Lists: Consider the following algorithm:

for $i \in \{1, \dots, n\}$ **do**
    Initialize **array** $A_i[1 \dots n]$ to be 0 everywhere
    for $j \in N(i)$ **do**
        $A_i[j] \leftarrow 1$
        for $k \in N(j)$ **do**
            $A_i[k] \leftarrow 1$
        end for
    end for
    $N^2(i) \leftarrow \{j \mid A_i[j] = 1\}$
end for

For each vertex i of the graph we initialize an array of size n where we will store all the neighbors of i in G^2. Initially the array is 0 everywhere. Then, for each neighbor j of i we mark that j is a neighbor of i in G^2; and we also mark that the neighbors of j are neighbors of i in G^2. Finally, we convert the array into a list.

It should be noted here that the reason we use this intermediate array, rather than a list into which we add all the neighbors of neighbors of i, is that there could be many paths of length two connecting i to k. We want to make sure that, despite this, k is added only once in the final solution.

The running time of this algorithm can be upper-bounded as follows: for a given vertex i, the outer for loop will make $\deg(i)$ iterations, while the inner loop will make $\deg(j)$ iterations. So, the total running time is proportional to $\sum_{i\in\{1,\dots n\}}\sum_{j\in N(i)}\deg(j)$. We now observe that if we fix a $j \in \{1,\dots,n\}$, the term $\deg(j)$ appears exactly $\deg(j)$ times in this double sum (once for each neighbor of j). Hence, the running time is proportional to $\sum_{j\in\{1,\dots,n\}}\deg(j)^2$. We have $\deg(j)^2 \le n\deg(j)$ and $\sum_{j\in\{1,\dots,n\}}\deg(j) = 2m$, so the running time is at most $O(nm)$, although this is a loose upper bound (we used the fact that all degrees are at most n). We should also note that we spend some time converting between arrays and lists. This costs $O(n)$ per vertex, so $O(n^2)$ in total. However, if $m = \Omega(n)$, which is true unless the graph contains a large number of isolated vertices, $n^2 = O(nm)$, so this part of the algorithm does not contribute much to the asymptotic complexity. In case the graph is very sparse (e.g. $\sum\deg(i)^2 = O(n)$), the use of arrays does become a bottleneck, but can be avoided by using more sophisticated data structures which eliminate duplicate elements.

4 Universal Sink

In a directed graph, a **sink** is a vertex of outdegree 0 (and a **source** is a vertex of indegree 0). A **universal sink** is a sink of indegree $n-1$. Give an algorithm that takes as input the adjacency matrix of a digraph and outputs a universal sink, or correctly reports that no such vertex exists.

Solution:

Recall that if A is the adjacency matrix of a digraph G, then $A[i,j] = 1$ if and only if we have the arc $i \to j$ in G. A universal sink is an index c such that for all $i \neq c$ we have $A[i,c] = 1$ and $A[c,i] = 0$. In other words, the c-th row must contain only 0 entries, and the c-th column must contain only 1 entries (other than the element on the main diagonal). It is now easy to check if a specific index c satisfied this condition in time $O(n)$, giving an algorithm running in time $O(n^2)$.

Surprisingly, it is possible to do better. Consider the following algorithm: we maintain two variables, c and u where c can be thought of as the current candidate sink and u an upper bound on the vertices we have explored so far. More precisely, we have the following invariants: (i) $c \le u$ always (ii) for all $j \in \{1,\dots,u-1\}\setminus\{c\}$ we know that j is not a universal sink. In the following, we suppose that the procedure Check takes as input a vertex c and checks if c is a universal sink in time $O(n)$.

$c \leftarrow 1$
$u \leftarrow 1$
while $c \le n$ **do**
    while $A[c,u] = 0$ and $u \le n$ **do**
        $u{+}{+}$
    end while
    if $u = n+1$ **then** Output Check(c)
    else
        $c \leftarrow u$
    end if
end while

We observe now that if $A[c,u] = 0$, then u cannot be a universal sink, so the inner while loop maintains the invariants. Once we exit the loop, one of two things may happen: (i) $u = n+1$, which by the invariant means that the only possible universal sink is c, so we check this possibility and output the result (ii) $A[c,u] = 1$, in which case c is not a sink, so by setting $c \leftarrow u$ we maintain the invariants.

For the running time, we observe that u never decreases, so the total number of iterations is $O(n)$. Once we have found a candidate c, the Check procedure also takes $O(n)$ time.

5 Triangle Detection

Give an algorithm which takes as input a graph G (in adjacency matrix or list form) and decides if G contains a triangle, that is, three vertices x, y, z which are pairwise adjacent.

Solution:

Adjacency Matrix: Recall that if A is the adjacency matrix of G, then $B = A^2$ has the property that $B[i,j] \neq 0$ if and only if i, j have a common neighbor in G. Our algorithm is then to compute $B = A^2$ and then compute the matrix C with $C[i,j] = A[i,j] \cdot B[i,j]$. A triangle exists if and only if $C[i,j]$ has a non-zero entry. Indeed, if a triangle i, j, k exists, then $A[i,j] = 1$ and $B[i,j] > 0$, so $C[i,j] > 0$. Conversely, if $C[i,j] > 0$, then $A[i,j] = 1$ and $B[i,j] > 0$, so i, j form a triangle with some other vertex. The complexity of this algorithm is dominated by the matrix multiplication step.

Adjacency Lists: We use the following algorithm.

for $i \in \{1, \dots, n\}$ **do**
    for $j \in N(i)$ **do**
        if $N(i) \cap N(j) \neq \emptyset$ **then**
            Output Yes
        end if
    end for
end for
Output No

Correctness is not hard to see: we consider all edges ij, and for each such edge check if i, j have a common neighbor (which is necessary and sufficient for a triangle to exist).

Complexity is trickier, because we are using the operation $N(i) \cap N(j) \neq \emptyset$. It is important to be careful here, because in some high-level languages (Python!) such operations may appear as elementary, while they are not. For instance, for the Python `set` type, the intersection operation between two sets has **average** complexity proportional to the size of the smaller set, but **worst-case** complexity proportional to the product of the sizes of the two sets![1]

Let us think of a basic way in which we could implement this operation: if we sort all the lists $N(i)$ beforehand, then we can perform a Merge operation between $N(i)$ and $N(j)$ to test disjointness. This has complexity $O(|N(i)| + |N(j)|)$. We then obtain a running time proportional to $\sum_{ij \in E}(\deg(i) + \deg(j)) = O(\sum_{i \in \{1,\dots,n\}} \deg(i)^2)$, which can be upper-bounded as in a previous exercise.

6 Ramsey

Prove that in any group of 6 people, there are either 3 people who all know each other or 3 people who do not know each other. Show that this is false for groups of 5 people.

Generalization: prove that for all k, in any group of 4^k people, there are either at least k who all know each other, or at least k who do not know each other.

Solution:

6 people: we model this with a graph on 6 vertices and prove that there exists a clique or an independent set of size at least 3. Let a be the vertex of highest degree. If the degree of a is at most 2, then the graph is a union of paths and cycles, so there is an independent set of size 3. If not, we check to see if $N(a)$ induces any edges. If yes, we have a triangle; otherwise we have an independent set of size at least 3. For 5 people, it suffices to consider a C_5.

4^k people: we prove that for positive integers s, c any graph with at least 2^{s+c} vertices contains an independent set of size s or a clique of size c. By setting $s = c = k$ we obtain the result.

To prove the claim we use induction on $s + c$. For $s + c = 2$ (which is the minimum value) the statement holds. Consider now two fixed values s, c and suppose the statement is shown for any smaller pair. Take a graph $G = (V, E)$ with at least 2^{s+c} vertices and take an arbitrary vertex x. If $|N(x)| \geq 2^{s+c-1}$, then $G[N(x)]$ contains either a clique of size $c - 1$ or an independent set of size s; in the latter case we are done,

[1] `https://wiki.python.org/moin/TimeComplexity`

in the former case we form a clique of size c by adding x. Otherwise, $|N(x)| \leq 2^{s+c-1} - 1$, therefore, $|V \setminus N(x)| \geq 2^{s+c-1} + 1$. Consider then the graph induced by $V \setminus (N(x) \cup \{x\})$, which has at least 2^{s+c-1} vertices. By inductive hypothesis this graph has at least a clique of size c (in which case we are done) or an independent set of size $s - 1$, to which we can add x to obtain an independent set of size s in G.