Exercise 1. *Crossing Lemma*

1. *Proof.* For the given graph with $m \geq 6n$ edges to be reasonable $m \leq \frac{n(n-1)}{2}$ needs to hold, since that is the number of edges in a complete graph and a graph with more edges than a complete graph is unreasonable. Therefore we know the following:

$$\frac{n(n-1)}{2} \geq 6n \quad \Rightarrow \quad n \geq 13$$

For $n \geq 13$, we know that the graph cannot be 2-planar, since for a two planar graph $m \leq 5n - 10$ needs to hold and this cannot be true for a graph with the conditions $m \geq 6n$ and $n \geq 13$.

The reasoning for getting to the expression stated in the exercise is now the following: If we remove the maximum number of edges that can be in a 2-planar graph from G, the number of edges that remain did produce at least one crossing. The same is true if we remove the maximum number of edges in a 1-planar or a maximum planar graph from G. Therefore we can find the following expression:

$$\operatorname{cr}(G) \geq \underbrace{m - (5n-10)}_{a} + \underbrace{m - (4n-8)}_{b} + \underbrace{m - (3n-6)}_{c} \tag{1}$$

Here terme a corresponds to the number of edges contributing at least 3 crossings, term b is the number of edges contributing at least 2 crossings and term c the number of edges contributing at least 1 crossing. Therefore the total expression on the right hand side is the total number of crossings, ignoring the fact that edges can have more than 3 crossings and therefore this is a lower bound to the number of crossings in G. We can simplify (1) to the following:

$$\operatorname{cr} \geq 3m - 12n + 24$$

□

2. This proof follows the proof of Theorem 5 in the first lecture on planarity.

Proof. First, we define a probability $p = \frac{6n}{m} \leq 1$. We choose every vertex in G with this probability p to form a subgraph $G_p \subseteq G$. This subgraph has the random variables $n_p = |V(G_p)|$, $m_p = |E(G_p)|$ and $\operatorname{cr}(G_p)$. The expected values for these random variables are the following: $E(n_p) = pn$, $E(G_p) = p^2 m$, since an edge is only in G_p if both its ending vertices are in G_p, $E(\operatorname{cr}(G_p)) = p^4 \operatorname{cr}(G)$, since a crossing is only in G_p if both edges forming the crossing are in G_p.

Note that for the expected graph, the condition $m \geq 6n$ does still hold, as can be seen in the following way:

$$E(n_p) = pn = \frac{6n^2}{m}, \quad E(m_p) = p^2 m = \frac{36n^2}{m} \quad \Rightarrow \quad E(m_p) \geq E(n_p) \quad \checkmark$$

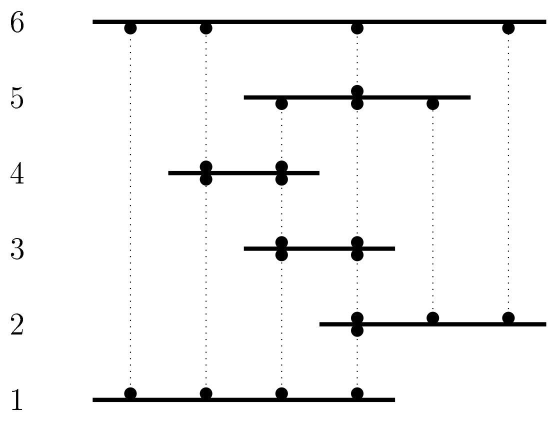
Therefore we know the following as shown in 1 (dropping the constant for simplicity):

$$\begin{aligned}
E(c_p) &\geq 3E(m_p) - 12E(n_p) \\
p^4 \mathrm{cr}(G) &\geq 3p^2 m - 12pn \\
\mathrm{cr(G)} &\geq \frac{3m}{p^2} - \frac{12n}{p^3} \\
&= \frac{m^3}{12n^2} - \frac{m^3}{18n^2} \\
&= \frac{1}{36}\frac{m^3}{n^2}
\end{aligned}$$

□

Exercise 2. *An Application of the Canonical Ordering*

a) The following is a visibility representation of the given graph. The thin dotted lines represent the edges that are present in the graph.



b) The following algorithm can be used for a visibility representation of a planar triangulation.

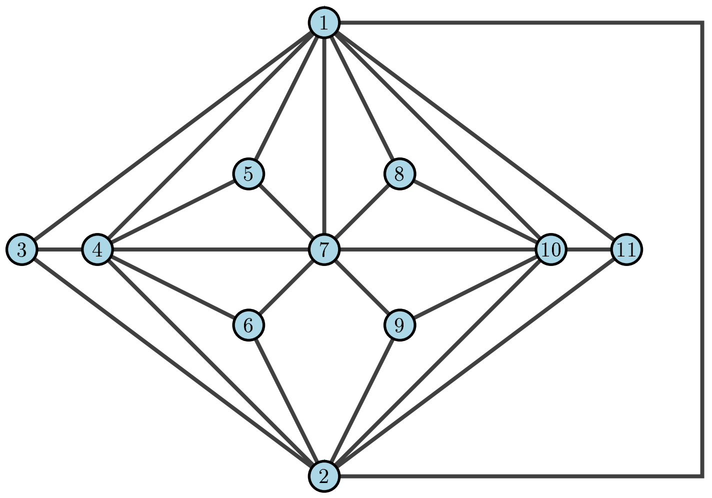
Algorithm 1: Algorithm for calculating a visibility representation of a planar triangulation.

```
Input: A planar triangulation $G = (V, E)$
Result: The visibility representation $R$ as a list of 3-tuples where the first two
        entries are the left and right $x$ coordinates and the third entry is the $y$
        coordinate
$v_1, v_2, \dots, v_n$ = a canonical ordering of the vertices $V$;
$R[1] = (0,1,3)$,   $R[2] = (2,2,2)$,   $R[3] = (1,3,2)$;
$w[1] = 1$,   $w[2] = 3$,   $w[3] = 2$;
$r = 3$;
for $i \leftarrow 4$ to $n$ do
    $p = i$;
    $q = 1$;
    for $j \leftarrow 1$ to $r$ do
        if $\exists(v_i, v_{w[j]}) \in E$ then
            if $j < p$ then
                $p = j$;
            end
            if $j > q$ then
                $q = j$;
            end
        end
    end
    $s_1 = R[w[p]][1]$;
    $s_2 = R[w[q]][2] - 1$;
    for $j \leftarrow 1$ to $i - 1$ do
        for $k \leftarrow 1$ to $2$ do
            if $R[j][k] > s_2$ then
                $R[j][k] = R[j][k] + 2$;
            else if $R[j][0] > s_1$ then
                $R[j][k] = R[j][k] + 1$;
        end
    end
    $R[i] = (s_1 + 1, s_2, i)$;
    $w_{\text{old}} = w$;
    for $j \leftarrow 0$ to $r - q$ do
        $w[p + 2 + j] = w_{\text{old}}[q + j]$;
    end
    $w[p + 2] = i$;
    $r = r - (q - p) + 1$;
end
```

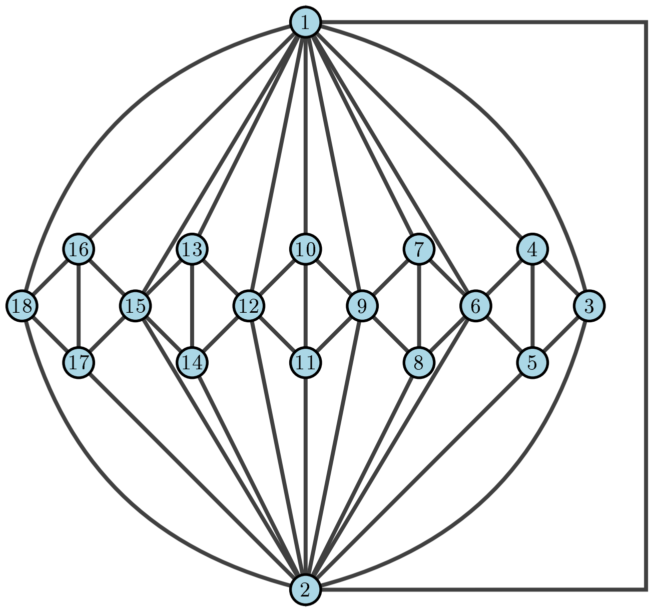
In this algorithm, the outermost loop performs $n-4$ iterations and the inner loops r, $i-1$ and $r-q$ iterations. r, $r-q$ and $i-1$ are in $\mathcal{O}(n)$, therefore the entire algorithm runs in $\mathcal{O}(n^2)$, ignore the calculation of the canonical ordering or assuming, that the graph is already given with a canonical ordering.

Exercise 3. *Canonical Order*

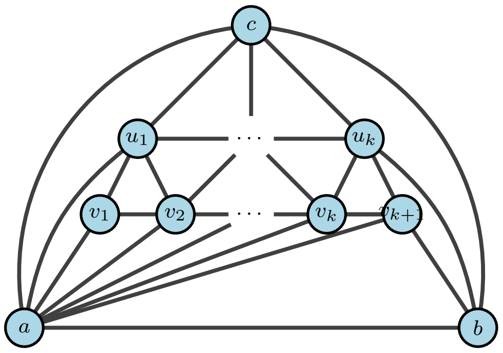
a) (a)



(b)



b) Consider the following family of graphs:

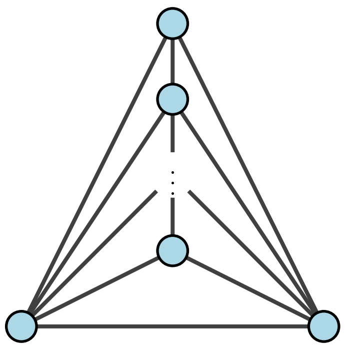


Note that this family of graphs is maximum planar since it is a triangulation.

In such a graph we can start numbering the vertices starting with a as 1 and b as 2. The next vertex that can then be chosen is only v_{k+1}, which gets the label 3. Next we can number the remaining v vertices downwards where the vertex v_i will be assigned the label $3+k-i$. When all vs and none of the us are labeled, we can label the us in any order that we choose, which means for the label $3+k+j$, we have $k-j$ possible choice, which leads to a total of $k!$ possible labellings for the u-vertices. $k = \frac{n}{2} - 2 = \Theta(\frac{n}{2})$, therefore the total number of possible labellings in that way is $\Theta(\frac{n}{2}!)$. Note that we could also start labeling the us in between the vs, which leads to more possible orderings, but not more than $\Theta(\frac{n}{2}!)$.

If we start searching for a canonical ordering from another starting point (e.g. with a as 1 and c as 2) we can also find more canonical orderings, but never more than $\Theta(\frac{n}{2}!)$, therefore the total number of canonical orderings is $\Theta(\frac{n}{2}!)$.

c) Consider the following family of planar graphs:



Note that this family of graphs is maximum planar since it is a triangulation.

If one fixes any v_1, v_2 and v_n, then the canonical ordering is uniquely defined. As an example, we choose the lower left vertex as v_1 and the lower right one as v_2. We choose the bottom most remaining vertex as v_n, then the other vertices must be labeled in ascending order from top to bottom.

Exercise 4. *Separator for Weighted Trees*

A $\left(1, \frac{1}{2}\right)$ Seperator for a weighted tree T with weight function $c : V \to R_{\geq 0}$ will consist of one vertex, separating the tree of the CC of the parent and the CC of every child with its respective weight sums.

Algorithm 2: Algorithm to calculate a Separator of a given Weighted Tree

```
Input: Weighted Tree $T$ with weight function $c$
Result: Set of all $\left(1, \frac{1}{2}\right)$ separator vertices
1 weight_sum ← $\sum_{v \in V} c(v)$
2 Candidates  ← $V$
3 SubtreeWeightMap ← $\emptyset$
4 r ← $root(T)$/* If $T$ was not a rooted tree, make it to one by
     picking a random root                                     */
5 Helper($r$)
6 return Candidates
```

Algorithm 3: Helper(v)

```
Input: vertex $v$ of a tree $T$ with weight function $c$
Result: Mapping of all vertices of the tree starting at $v$ as key and the weight
        sum of the respective subtree as value
1 if $v$ Leaf then
2     SubtreeWeightMap.append($v, c(v)$)
3     if $c(v) < \frac{weight\_sum}{2}$ then
4         Candidates ← Candidates$\setminus\{v\}$
          /* If the weight is less than half of the total sum, then
             the leaf is not a separator                           */
5 else
6     for $w.children\ of\ v$ do
7         Helper($w$)
          /* recursively compute all the subresults necessary      */
8     SubtreeWeightMap.append$\big((v, c(v) + \sum_{w \in \texttt{children of } v} SubtreeWeightMap(w))\big)$
      /* The subtree weight sums are stored in the map starting at
         root $v$                                                  */
9     if $\left(SubtreeWeightMap[v] < \frac{weight\_sum}{2}\right)$ or
       $\left(\exists w.\ Children\ of\ v\ and\ \texttt{SubtreeWeightMap}[w] > \frac{weight\_sum}{2}\right)$ then
10        Candidates ← Candidates$\setminus\{v\}$
          /* If the whole subtree with $v$ as root weighs less than half
             of the total sum, then the CC resulting of $G$ without the
             subtree violates the constraint. Similarily for the
             children subtree weight sum case.                     */
```

Correctness

A separating vertex v will always separate a tree in multiple CC:

- $G[V - V(Subtree(v))]$ as the residual graph without the subtree rooted at v
- $V[Subtree(w)]$ for every child w of v

This is a result from the fact that a tree is per definition 1-connected, without any cycles and has exactly $n-1$ edges. The algorithm 2 will at first compute the total sum `weight_sum` of the tree for comparison reasons. A tree is either rooted or free. If the given tree was free, then it can be altered to a rooted tree T' by setting an arbitrary vertex to the root. At first, all the vertices are candidates to be a $\left(1, \frac{1}{2}\right)$ separator. With help of a recursive Helper function starting at root, a mapping is constructed where every vertex serving as key will have the total weight of its subtree as value.
In the recursion, the terminating premisse is that the current vertex is a leaf. Then, the subtree is the vertex with its weight. Else, the helper function is called recursively for every child of v since the subresults are necessary in order to decide whether or not v is a separator.
If the total weight of the subtree of v is smaller than half of the total weight of T, then v cannot be a separator since the remaining CC is greater than half of the total weight. Similarily, if a subtree of a child of v is in total sum greater than half of the total weight sum of T, v is not a seperator and therefore removed from the set of candidates.
The algorithm will terminate sind the trees are finite and every leaf will be a terminating case of the recursion tree. In every step of the recursion, every subresult necessary for the parental decision making is computed. Finally, it will be decided whether or not the root of T (or T') is a separator. Therefore, the algorithm is correct.

Runtime

The computation of the total weight sum of T is in $\mathcal{O}(n)$. The recursion works bottom up for every tree (leaves first) and every vertex is visited exactly once, since T is a tree. The amount of arithmetic operations, logical conditions and decisions are constant in every recursion step, except for the comparison whether or not a child subtree violates the constraint (Helper function, line 9). In the amortized analysis, there will be $\mathcal{O}(n)$ childs for a given tree T to compare during the recursion. Therefore, the total runtime is in $\mathcal{O}(n)$.