Master of Science in Advanced Mathematics and Mathematical Engineering

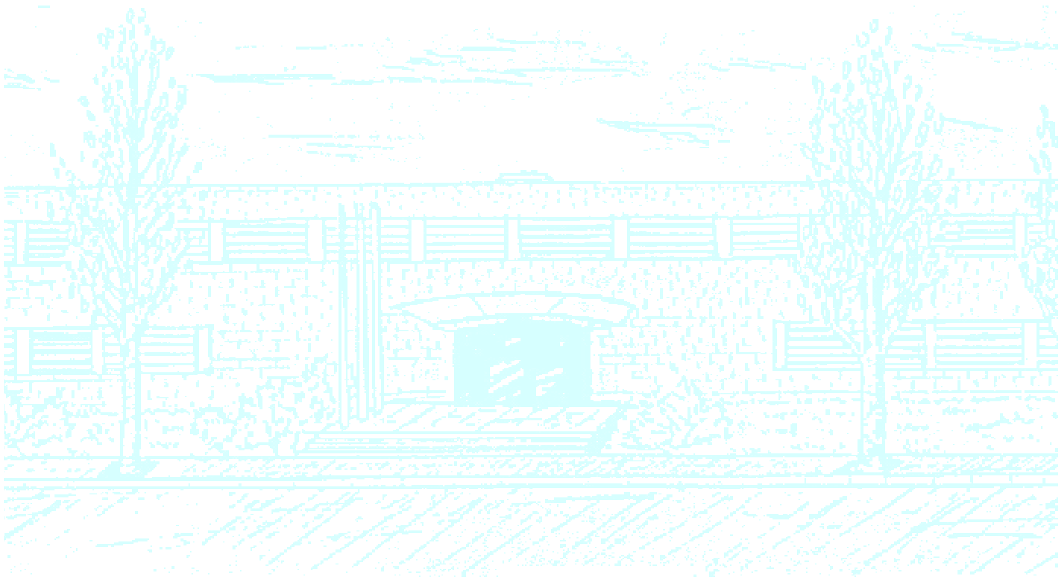
Title: Enumeration and Random Generation of Planar Maps

Author: Ivan Geffner

Advisor: Marc Noy

Department: Matemàtica Aplicada IV

Academic year: 2013 - 2014



Universitat Politècnica de Catalunya
Facultat de Matemàtiques i Estadística

Master's Final Thesis

Enumeration and Random Generation of Planar Maps

Ivan Geffner

Advisor: Marc Noy

Departament de Matemàtica Aplicada IV

I would like to thank Marc Noy for his dedication and encouragement. Thanks to him I have learned a lot of Mathematics and I have also had a lot of fun and challenges while doing this work.

Abstract

Keywords: Maps, combinatorics, random generation, outerplanar maps

MSC2000: 200005C99

In this work we will discuss several previous results in the area of graph theory related to planar rooted maps. Moreover we will give a new way to encode outerplanar maps, which improves several results of [**2**], and perform an algorithm to generate random planar maps of a given size, all of them with equal probability.

The first three sections are dedicated to introduce the basic concepts and properties of planar maps: definitions, different representations and some results. These will be the main tools that are going to be used in the next sections.

In the fourth section we will explain carefully a bijection between outerplanar maps of size n and special kinds of Dyck paths. This bijection will allow us to encode outerplanar maps with their correspondent Dyck path. Concretely it will allow us to encode simple outerplanar maps with n nodes using $3n$ bits, which is shown also in [**2**] but in a much more complicated way.

Finally in the last three sections we will discuss an algorithm to randomly generate maps of a given size. We will also attach a code in C++ that performs such an algorithm to fill all the details and we will carefully explain each of the functions of the code along with their inputs and outputs. In the last section we will use the program to compute accurately some parameters of planar maps.

Contents

1. Introduction

A planar graph, is a graph that can be embedded in the plane, which means that can be drawn in the plane in such a way that its edges intersect only at their endpoints.

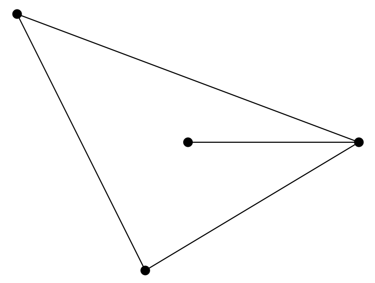
Since every graph that can be drawn in the plane can also be drawn in the sphere, we can define a planar map as an equivalence class of the topologically equivalent drawings on the sphere whenever the graphs are connected.

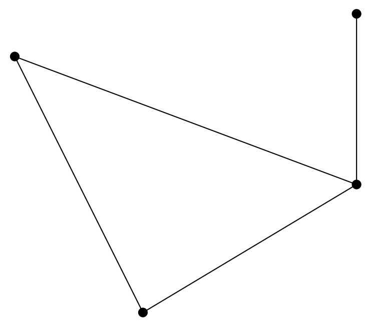
This means that in addition to the adjacency information of the graph, to represent a map we also need its embedding.

Two planar graphs G and G' are isomorphic if there exists a bijection f between the vertices of G and G' such that the adjacency is preserved by f, or in other words, that if u and v are adjacent in G if and only if $f(u)$ and $f(v)$ are adjacent in G'.

To define an isomorphism between two maps A and B we need more than just adjacency since we have to take into account their embeddings. Hence what we need is an homeomorphism ψ of the sphere such that $\psi(A) = B$.

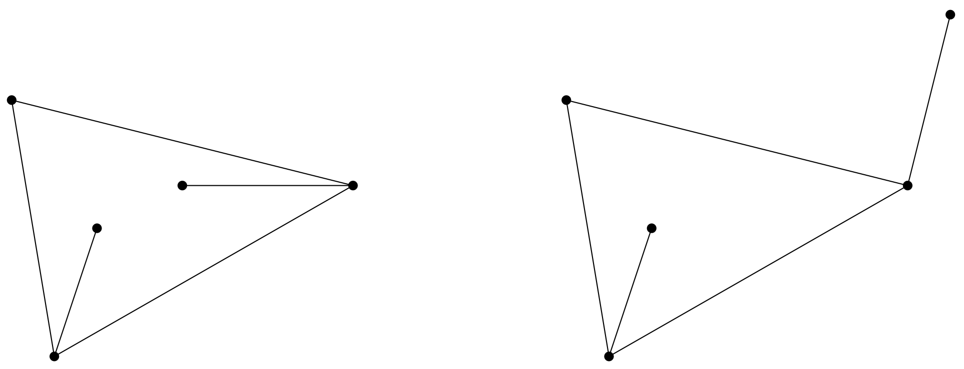
For instance these two planar maps are isomorphic[1], and so are their respective graphs:





[1]We may recall that we are working in the sphere, and hence we can swap the exterior face with the interior face

But the following maps are not isomorphic, even though their graphs are, since the first one has a 3-face and a 7-face while the second one has two 5-faces:



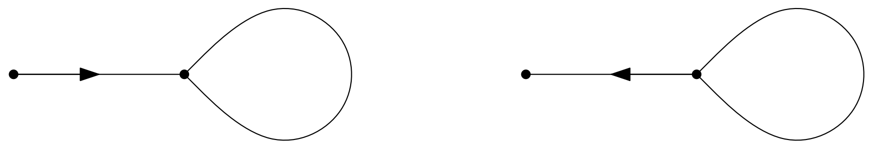
By the nature of these maps, counting them is extremely difficult. An idea to make it easier was to orientate one of the edges, which is referred as the root, leaving the exterior face at its right. By doing this we are fixing the exterior face along with an edge, and therefore it is easy to see that every other face of the map can be distinguished as well. From now on we will be working exclusively with rooted maps.

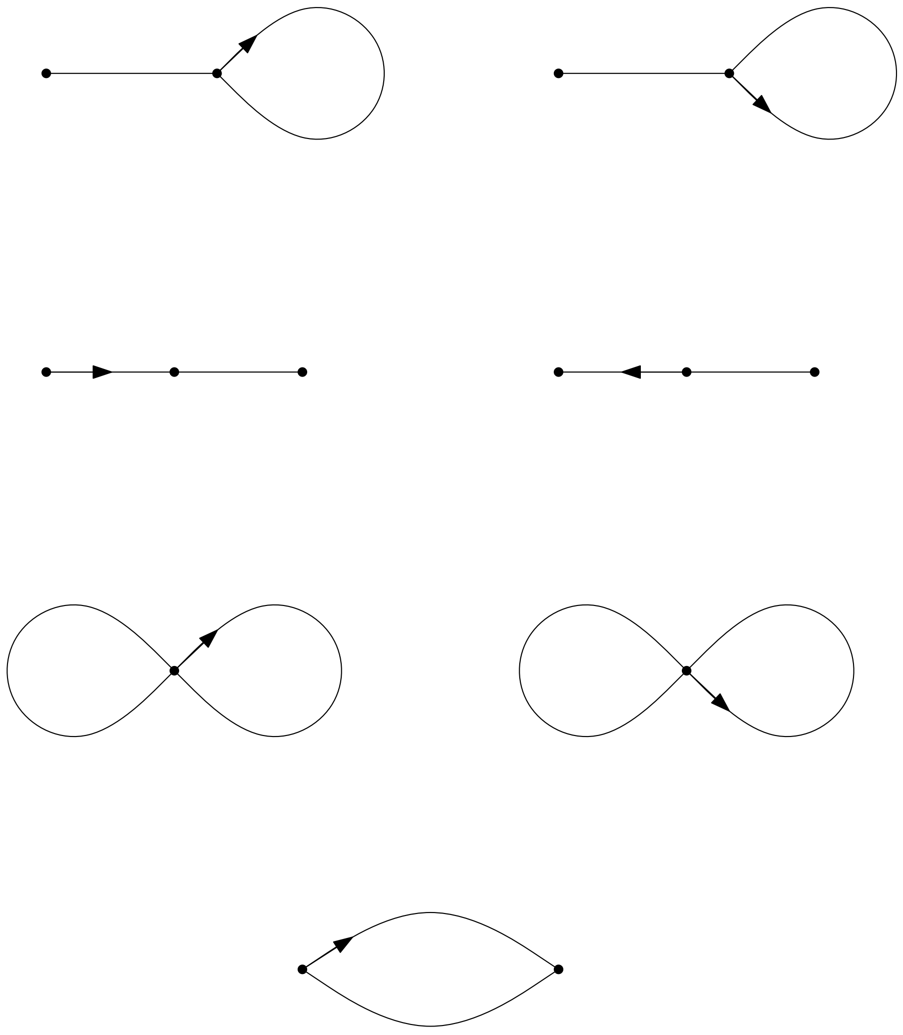
These new kind of maps were introduced first by Tutte (1968). He was able to count them successfully achieving the following result: if $M(n)$ is the total number of planar maps of size n (with n edges), then

$$M(n) = \frac{2 \cdot 3^n}{n+2} C_n$$

where C_n is the n-th Catalan number.

For example, with $n = 2$ these are all the nine possible maps:





The aim of this work is first to perform a random map generator, in which all maps of a given size have equal probability to appear, and then counting certain types of planar maps by setting bijections with simpler combinatorical objects.

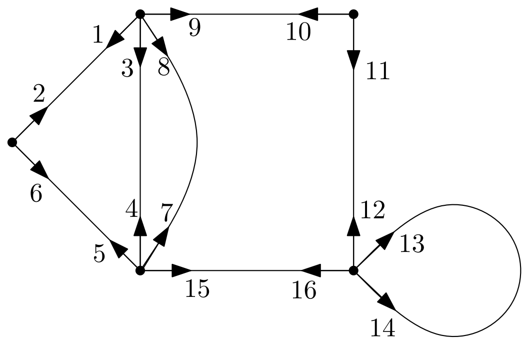
2. Representation of Planar Maps

As we remarked in the last section, the nature of maps makes it difficult to encode them in an easy way. For instance, to fully codify a graph we only need its adjacency list, but in our case we have seen that this is not enough, since isomorphic graphs can be embedded into non-isomorphic maps.

There are several ways to encode a map. For example, if we want to use the adjacency list, we can just save for each node the order of its incident edges of each node in counterclockwise order. Actually there are several ways to code a planar map, but the one that we will detail next outstands for its simplicity and elegance.

2.1. The permutation encoding. Given a map M with n edges amb m nodes, for each $i \in [n]$ we will split the i-th edge (taken in any order) in half, and assign to one of the halves the number $2i$, and assign $2i+1$ to the other one (it can be done in any way). Now every node $v_i \in M$ with degree g_i is incident to the semi-edges $a_{i_1}, a_{i_2}, \dots, a_{i_{g_i}}$ and we will suppose that these edges are sorted in counterclockwise order.

Here is a picture of a possible configuration:



Now we will define the transposition $\tau_i = (2i\ \ 2i{+}1)$, and the cycle $\sigma_i = (a_{i_1}\ a_{i_2}\ \dots\ a_{i_{g_i}})$.

If $\tau = \prod_{i=1}^{n} \tau_i$ and $\sigma = \prod_{i=1}^{m} \sigma_i$, our map will be coded as (τ, σ).

In our example we have

$$\begin{aligned}\tau &= (1\ 2)(3\ 4)(5\ 6)(7\ 8)(9\ 10)(11\ 12)(13\ 14)(15\ 16)\\ \sigma &= (1\ 3\ 8\ 9)(2\ 6)(15\ 7\ 4\ 5)(10\ 11)(12\ 16\ 14\ 13)\end{aligned}$$

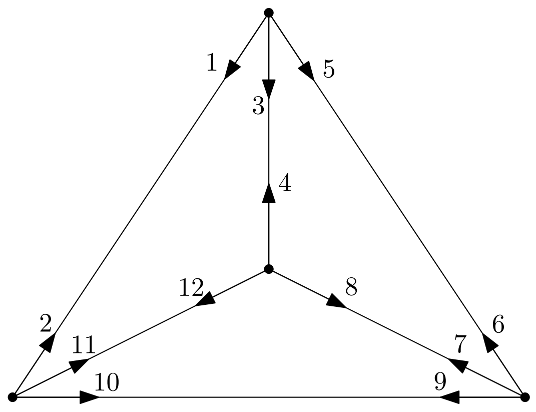
This codification holds two important properties.

(1) If we orientate each of these semi-edges from its node to the middle of the edge and then we apply the permutation $\sigma\tau$, it can be easily checked that every semi-edge maps exactly to the consecutive semi-edge that leaves the same face

at its right. Hence, the cycles of $\sigma\tau$ are precisely the faces of M. This property is essential to find the dual map of M. In our case we have

$$\sigma\tau = (1\ 6\ 15\ 14\ 12\ 10)(2\ 3\ 5)(4\ 8)(7\ 9\ 11\ 16)(13)$$

(2) If we change the order of the incident semi-edges in each node, we will get an embedding of M in a compact open surface. Moreover, we can compute the genus of the surface, since we have the number of nodes, edges and faces[2], since this last parameter is the number of cycles in $\sigma\tau$. For instance, let's see two possible embeddings of K_4:
In the first picture we have the classical embedding of K_4 in the sphere. We have 6 edges, 4 vertices of degree 4 and 4 3-faces:



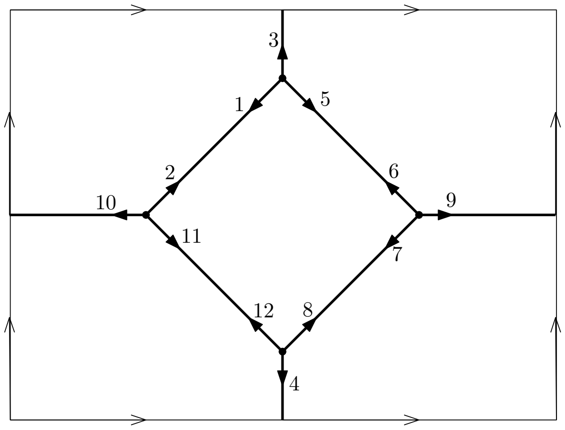
$$\tau = (1\ 2)(3\ 4)(5\ 6)(7\ 8)(9\ 10)(11\ 12)$$
$$\sigma = (1\ 3\ 5)(12\ 8\ 4)(6\ 7\ 9)(10\ 11\ 2)$$
$$\sigma\tau = (1\ 10\ 6)(2\ 3\ 12)(4\ 5\ 7)(8\ 9\ 11)$$

But if we change the order of incidence of the edges in each node, for instance with the following codification:

$$\tau = (1\ 2)(3\ 4)(5\ 6)(7\ 8)(9\ 10)(11\ 12)$$
$$\sigma = (1\ 5\ 3)(6\ 7\ 9)(4\ 8\ 12)(2\ 10\ 11)$$
$$\sigma\tau = (1\ 10\ 6\ 3\ 8\ 9\ 11\ 4)(5\ 7\ 12\ 2)$$

we can get the following embedding on a torus:

[2]Let's recall the Euler formula $F + V - E = 2 - 2g$, where F is the number of faces, V the number of vertices, E the number of edges and g the genus of the surface



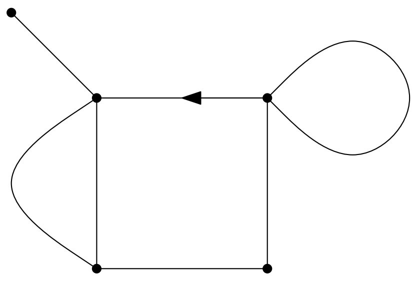
In this case we have 6 edges and 4 nodes of degree 3, but we have an 8-face and a 4-face.

2.2. Quadrangulations. A quadrangulation is a map where all its faces have degree 4. We will show that all maps of size n (i.e. with n edges) can be put in bijective correspondence with quadrangulations with n faces.

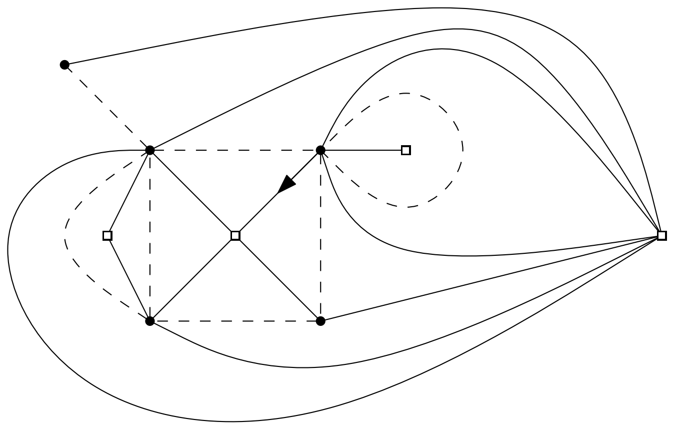
The bijection goes as follows: given a map M of size n, first we draw a vertex over each of its faces and connect them to the vertices of their face. Now we just have to erase all the original edges, and since each of them was exactly in two faces (that may happen to be the same), the new faces will be all of degree 4.

The last part of these mapping is to set a new root. Here we have several possibilities, actually any canonical choosing is correct. For instance if the previous root was uv (from u to v), we can set the new root as the first edge incident to u in counterclockwise order starting from uv.

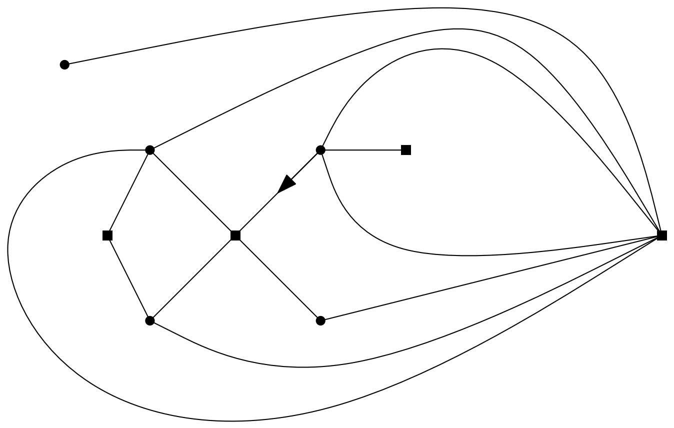
For instance, we will construct the quadrangulation of the following map:



First we draw a new vertex on each face and connect them to their adjacent vertices in the original map. These new vertices will be marked as squares, while the edges of the original map are dashed:



Finally we erase the edges of the original map and we get the desired quadrangulation.



This application is one to one since we can reconstruct the original map: if we do a bicoloring of the quadrangulation, one color represents the vertices of the original map, while the other color represents the vertices put into its faces. We can know which color represents the original map, since the root is rooted in one of its vertices. Finally, we move the root to its consecutive edge in clockwise order and erase all the edges of the original quadrangulation. This gives the orignal map.

Remark: A quadrangulation of a map of size n has n faces of degree 4, therefore it has $2n$ edges, and by Euler's formula it has $n + 2$ vertices.

3. Enumeration of Planar Maps

We will show two ways to get to Tutte's result. Both of them count the total number of maps of a given size by constructing an arbitrary map and computing how many ways we had to perform such construction. This helps to understand the true nature of maps, giving a relation between them and the Catalan numbers.

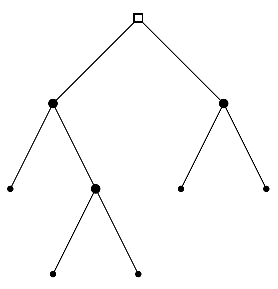
The first of these constructions was first made by G. Schaeffer (1998), in which he established a bijection between quadrangulations and balanced blossomed binary trees.

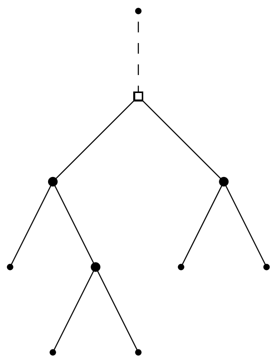
3.1. Balanced binary trees. Let us suppose that we have a binary tree with exactly n interior nodes and with a root-leaf hanging out from the root of the tree in order that every interior node has degree 3. Now we may add into each of these nodes a single blossom, which we may think of it as an additional leaf hanging from it, pointing at one of the three possible directions. The resulting tree must have exactly $n+2$ leaves and n blossoms. This structure is called a **blossomed binary tree**.

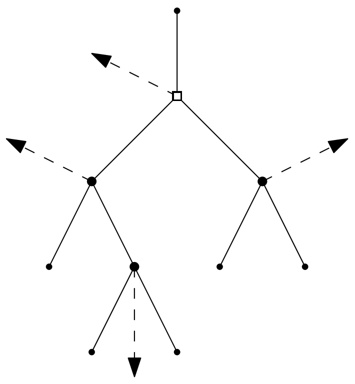
Next, go along the tree in counterclockwise order starting at an arbitrary leaf or blossom and we will create an associate word to the tree by writing x whenever we find a leaf, and $\overline{x}$ whenever we find a blossom. The resulting cyclic word will consist of $n+2$ "x" and n "$\overline{x}$". We will connect each $\overline{x}$ with the previous non-connected x, and we shall draw an edge between the blossom and the leaf. After this process only two leaves will remain free, which we will connect and mark this edge as the root in an arbitrary direction.

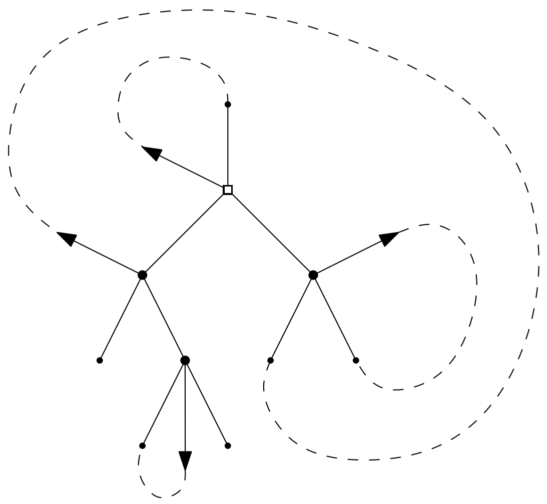
After completing this, all the interior nodes of the tree will have degree 4, while the blossoms and the leaves will have degree 2. We will proceed contracting these nodes of degree 2 to their respective parents to obtain a 4-regular connected map. By duality, changing nodes with faces, we would also obtain a map in which every face has degree exactly 4. Hence we have found an application from these blossomed binary trees to quadrangulations.

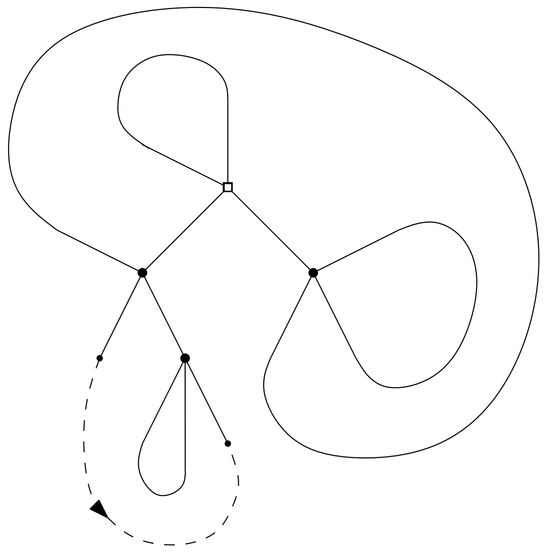
Here we will put an example of the construction of a quadrangulation from a binary tree with $n = 4$:

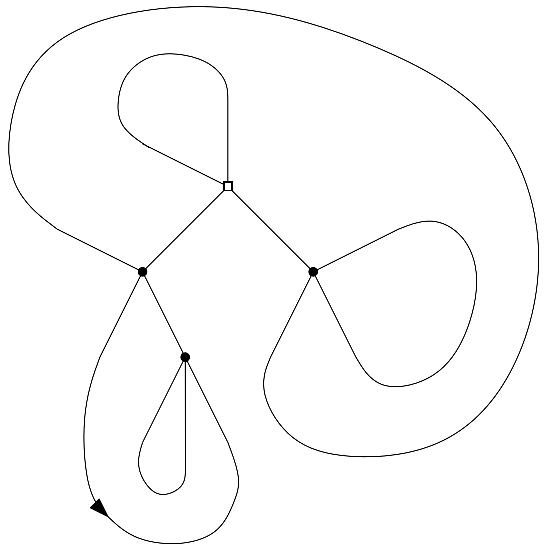












This application is not one to one, since more than one blossomed binary tree goes to the same quadrangulation. In fact, fixing a blossomed binary tree, its associate cyclic word is invariant by changing its root-leaf, and therefore the resulting quadrangulation is invariant too. Actually, it can be shown that these are the only possibilities for two quadrangulations being the same. This implies that the equivalence classes induced by our application are precisely the non-rooted binary trees, and all of them have cardinality $n + 2$ since it is the number of ways of choosing their root-leaf.

If our application is surjective it follows that if T_n is the number of blossomed binary trees with n interior nodes, the total number of planar maps with n edges is

$$\frac{2}{n+2}T_n$$

since each quadrangulation is counted exactly $n+2$ times and can be rooted in two possible orientations. Now using that the total number of binary trees with n interior nodes is the n-th Catalan number C_n, and that each blossom can be inserted in three ways, we have that $T_n = 3^n C_n$ and finally

$$M(n) = \frac{2 \cdot 3^n}{n+2}C_n.$$

To prove that our application is bijective we can construct its inverse in the following way:

We start at the root edge, and we move along the exterior face. At each step we do the following:

(1) If the edge in which we are placed is not a bridge we open it, which means that we replace it by a leaf in its start point and a blossom in its endpoint. An exceptional case happens when this edge is the root, in which we put two leaves.
(2) We move to the next edge in counterclockwise order following the exterior face

It can be shown that this is the inverse application that we are looking for, but a rigorous proof needs harder work and can be found in [**1**, Chapter 2]

3.2. Well-labeled Trees. Another way to count planar maps is due to R. Cori and B. Vauquelin (1981), which was improved by Schaeffer (1998). They showed that there exists a bijection between planar maps and a special kind of trees calles **well labeled trees**. These trees have a label on each vertex such that:

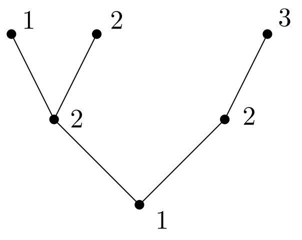
(1) The root has label 1
(2) Each label is positive (strictly greater than 0)
(3) If u and v are adjacent vertices, then their labels differ at most in 1 in absolute value.

For instance these are all the possible well labeled trees with 3 nodes:

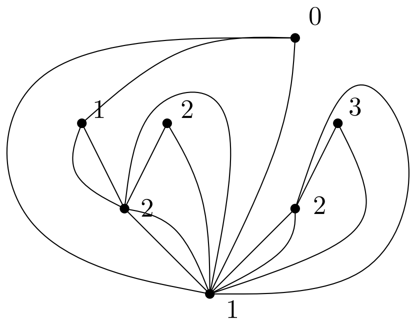
The bijection goes as follows:

Given a well labeled tree, we first add a vertex labelled with 0 and then go over it in clockwise direction and, for each node v, we draw an edge from v to the immediately next node w that has a value strictly lower (we always can find such a node because there is a vertex with label 0).

For instance let us consider the following well labeled tree:

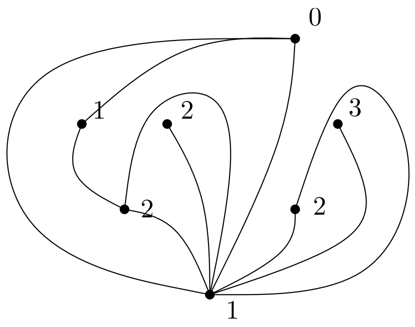


And this would be the result of applying the previous process to it:



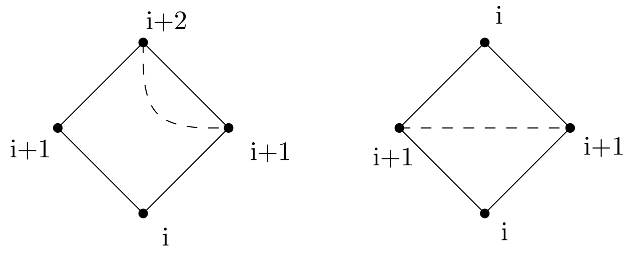
If we erase all the edges of the original tree, after analyzing a few cases we get that every resulting face has degree 4, and hence the result is a quadrangulation.

This would be the quadrangulation obtained from the previous tree:



This application is bijective since there is a way to construct its inverse: Given a planar map let us consider its associated quadrangulation. Each node of the quadrangulation will be labeled with its distance to the root. Now each of the faces of the quadrangulation can have one or two maximal nodes (with the maximum label in the face). If there is only one, we draw an edge from it to the next node in clockwise order, else, we draw an edge between the two maximal nodes.

These are the two possible cases:



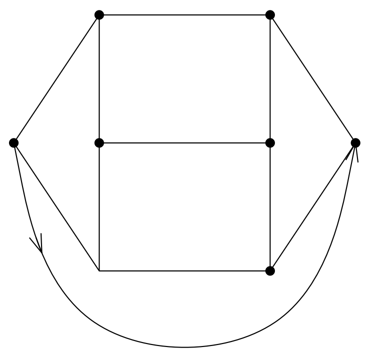
After erasing the original quadrangulation it is easy to see that the root node is isolated. Also, the remaining component has n edges (since we draw an edge for each face) and $n + 1$ nodes, therefore if we prove that it has no cycles it has to be a tree.

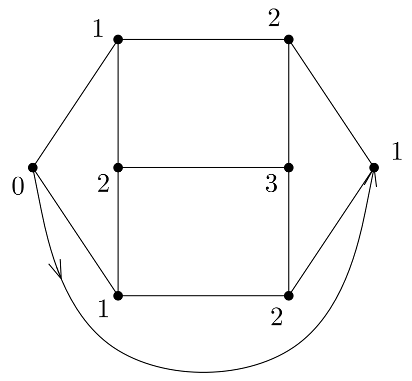
This last property comes by construction: if there is a cycle, then all the labels must be the same, otherwise we would have picked an edge in counterclockwise order. If all the labels are the same then the cycle splits the plane in its interior and its exterior, therefore vertices with less distance to the root than the nodes in the cycle which are at the other side of the cycle (in which the root is not) cannot exist since every path to the root passes through one of the vertices of the cycle. This contradicts that the vertices of the cycle are all pairs of maximals in their faces.

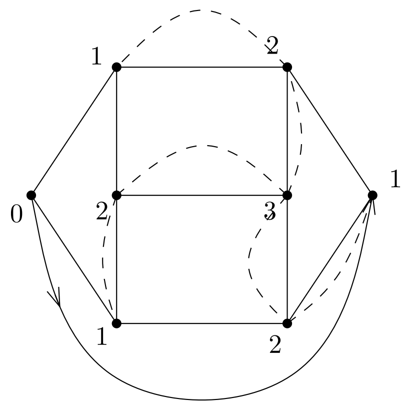
We have that the resulting map is the union of the root vertex and a tree, which is well labeled by construction. After erasing the root we would get a well labeled tree rooted at the vertex that was incident to the root (other than the root of course), which coincides with the inverse of the quadrangulation by the previous application[3].

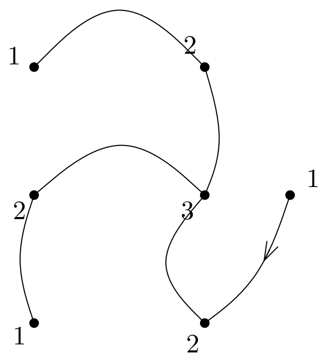
Here is the construction of the inverse for a given quadrangulation:

[3]For a full proof see [**1**, Chapter 6]




1
2
0
2
3
1
1
2


1
2
0
2
3
1
1
2


1
2
2
3
1
1
2

4. Outerplanar Maps

An **outerplanar map** is a map in which all the vertices are adjacent to the exterior face.

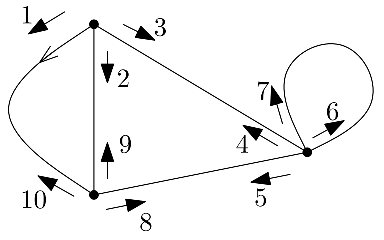
The aim of this section is first to construct a bijection between outerplanar maps and Dyck paths with marked D-steps, and then refine this bijection to characterize special kinds of outerplanar maps. Finally we will show a way to encode simple outerplanar maps of size n with $3n$ bits.

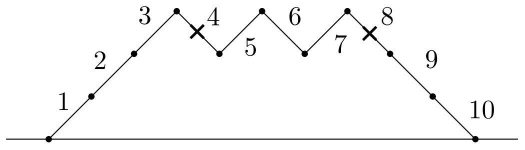
Let us start with the first bijection. We will always be supposing that the face at the right of the rooted edge is the external one.

PROPOSITION 4.1. *There exists a bijection between outerplanar maps with n edges and Dyck paths of length $2n$ with marked D-steps.*

PROOF. Given a map with n edges we will start in the rooted vertex and move counterclockwise. Every time we meet a new edge we will draw a U-step, and every time we meet a previously visited edge we will draw a D-step. When we finish visiting all the edges incident to the first vertex we just have to move to the last of the visited vertices x. We will repeat the process setting as initial edge the last one incident to x, but in order to save the information that we are changing the vertex we will mark the first step, that by construction it will always be a D-step.

The following figure shows an example of such construction:



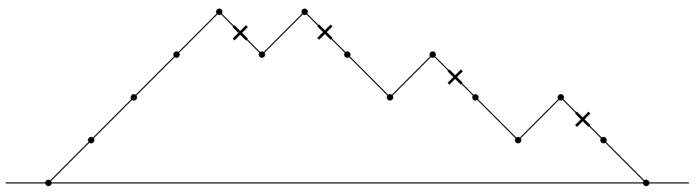


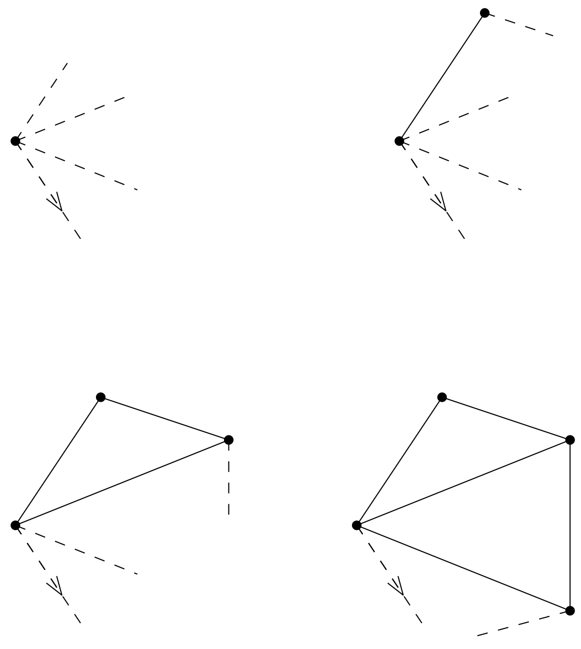
Clearly, every edge appears exactly two times (one for each incident vertex, that may be the same) and hence the path drawn contains exactly $2n$ edges. Moreover, the path cannot be beyond the x axis, since it would mean that we have checked for the second time more edges than for the first time. Thus, this is an injective application between outerplanar maps and Dyck paths of length $2n$ with marked D-steps.

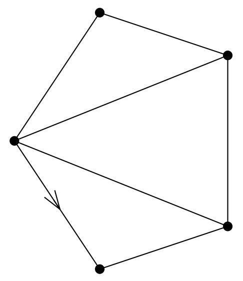
To see that it is actually a bijection we just have to see that we can construct an inverse: given a Dyck path of this kind, we have to find a way to construct its corresponding outerplanar map. We will begin with the root and start drawing "open" edges (meaning that we still don't know their endpoints besides the ones that are closed by D-steps) in counterclockwise order. Whenever we find a marked D-step we draw a new vertex in the last open edge and repeat the process in this new vertex.

Every reconstruction step is quite clear except when we have to close an edge and there is more than one possibility (more than one open edge). In this case there is only one possibility between all these edges, that is the last one that has been opened, and this holds because if we close another one, by construction we will form a cycle with open edges in the middle, which would lead to a non-outerplanar map since these edges have to end in some new vertices still undiscovered. Hence, we can reconstruct the maps from its marked Dyck paths and the claim follows. □

Here is an example of a possible reconstruction:







Since the number of marked D-steps in the Dyck path is one less than the number of vertices in the map, we get the following result:

Corollary 4.2. *The number of outerplanar maps with k vertices and n edges is equal to*

$$\binom{n}{k-1}C_n$$

We will continue by finding a bijection between loopless outerplanar maps of size n and **small Schröder paths**, which are sequences of U-steps, D-steps and horizontal steps of length two, in which there are no horizontal steps at level 0.

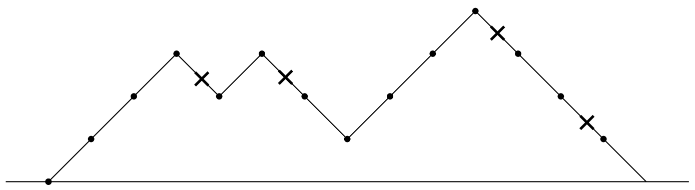
Proposition 4.3. *There exists a bijection between loopless outerplanar maps of size n and small Schröder paths of length n.*

Proof. A loop is created when a vertex has first opens and then closes an edge. Hence, if the D-step of an upper peak is not marked it forms a loop, and reciprocally if all the peaks are marked, no loops can be formed since a vertex cannot first open an edge and then closing it. Hence, an outerplanar map is loopless if and only if all the peaks in its Dyck path are marked.

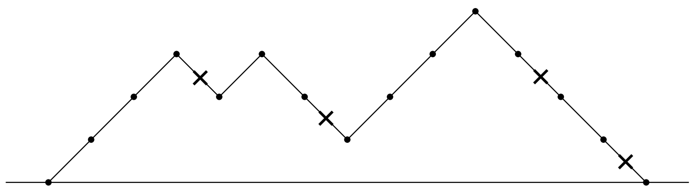
We will see that actually these types of Dyck paths can be put in correspondence with the small Schröder paths. First of all let us modify a little our Dyck paths: it is easy to see that if we apply a reversion of marks in each maximal descending intervals of our Dyck paths, the paths that have all the peaks marked are in correspondence with the paths that have marked every last D-step in the maximal descending intervals (these are the D-steps in the valleys and the last one).

Now we will work with these last types of Dyck paths. To construct our bijection we will do the following: for every pair of U and D-step such that the D-step closes the U-step, if the D-step is marked then we do nothing, but if the D-step is not marked, then we replace the U-step by a horizontal step of length 2. For instance, if the path is of the form $\alpha U\beta D\gamma$, where β is also a Dyck path (maybe already transformed), then if the D-step is not marked we would replace it by $\alpha H\beta\gamma$. It is clear that since every D-step at level 0 is marked, that no horizontal steps will be at level 0, and hence the result after unmarking all the D-steps will be a small Schröder path.

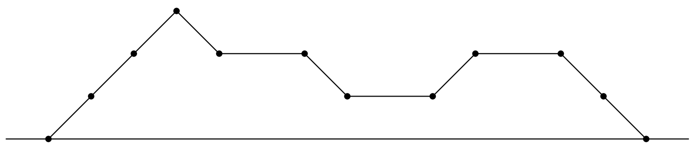
For example let us take the following Dyck path with marked D-steps:



After applying the reversion we get

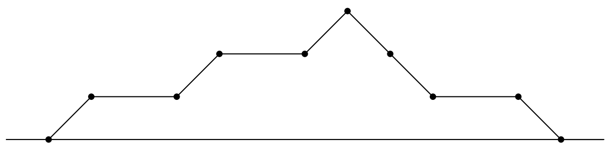


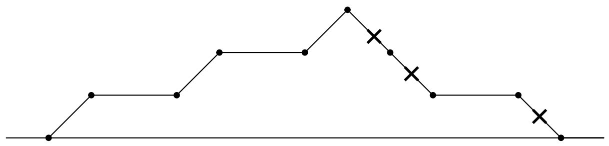
And after substituting the unmarked pairs for H-steps:

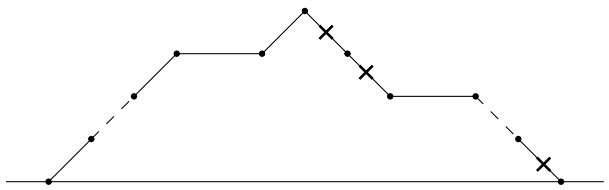


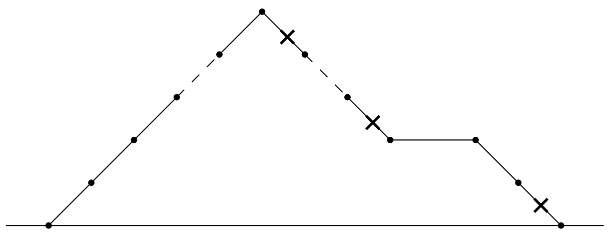
Finally we will construct its inverse, given a small Schröder path we will find a way to find the original Dyck path. First we mark all the D-steps, then for every horizontal step we find the first D-step that goes below the level of the horizontal step, that since we know that this level is always positive, we will always find such a step. We will put a U-step in the place where the horizontal step was, and a D-step without mark just before the D-step that we found. In this case if the path is of the form $\alpha H \beta D^* \gamma$, where β is a small Schröder path and D^* is a marked D-step, then we would replace it by $\alpha U \beta D D^* \gamma$. This completes our bijection. □

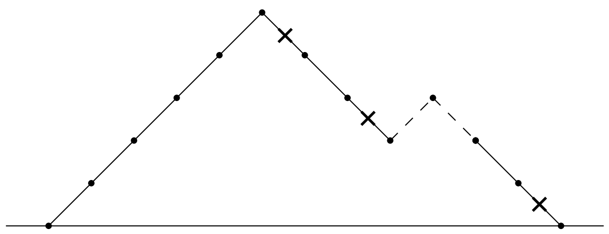
This would be an example of the reconstruction of a Dyck path:

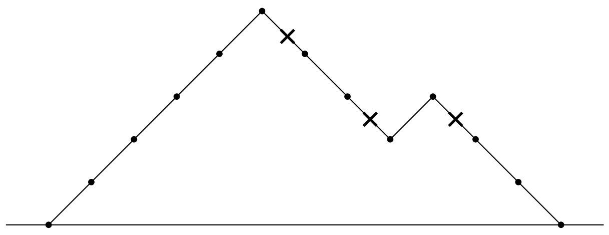












Finally we will show our last bijection, which leads to the encoding of a simple outerplanar map of size n with $3n$ bits.

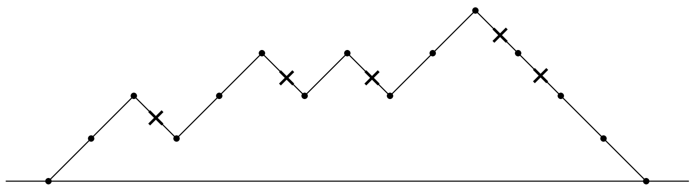
PROPOSITION 4.4. *There exists a bijection between simple outerplanar maps with $k+1$ nodes and Dyck paths of length k with marked U-steps but no marks on level 0.*

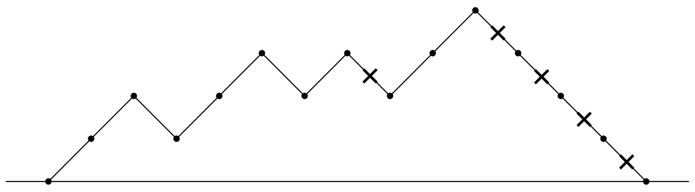
PROOF. Firstly, if there are no loops, then all the peaks are marked. Secondly, if a D-step x is not marked, since it is not a peak, it must follow that there is at least another D-step y just before x. If we consider their corresponding U-steps in the Dyck path, it must hold that they belong to two different maximal ascending intervals, because otherwise it means that they are closing edges from the same vertex and the map would not be simple. In other words, in the Dyck path, whenever a new maximal ascending interval begins (i.e. a valley), which we will suppose that is at level k, it allows the first D-step that goes below k to avoid being marked. And these are the only vertices which we may not mark.

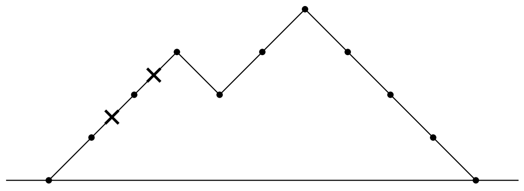
Now we will change a little the shape of these Dyck paths. We just saw that every valley allows exactly one D-step to avoid being marked, and hence we will say then that this valley and the D-step are in correspondence. Two or more valleys may allow the same D-step if they are in the same level, if this happens we will consider that the valley that appears first is the one correspondent to the D-step. Now we will go over the path from the beginning, and every time we find a not-marked D-step, we will swap it with the D-step of its correspondent valley. Then it follows that all these Dyck paths are in correspondence with the Dyck paths such that the only D-steps that we can leave unmarked are the ones in a certain type of valleys (if two or more valleys are at the same level and there are not steps that go below it in between, we can avoid marking just the first one).

Finally, in these new paths, we will go across from the beginning and whenever we find a non-marked D-step (in a valley), we will erase it along with its correspondent U-step, and we will mark the U-step in the valley. In other words, if the path was of the form $\alpha U\beta DU\gamma$, we will replace it by $\alpha\beta U^*\gamma$. The marked U-step cannot disappear, since it would mean that its correspondent D-step is in a non-marked valley, but this cannot happen since the valley where the marked U-step is comes first and is at the same level. The remaining path will be a Dyck path with all the D-steps marked, and some U-steps marked too. Since none of the valleys at level 0 of the original path had a not-marked D-step, none of the U-steps at level 0 in the new paths are marked neither. After unmarking all the D-steps the resulting Dyck path will be of length k, where k is the number of marked D-steps, and hence the number of nodes, and none of the U-steps at level 0 are marked.

This is an example of such construction:



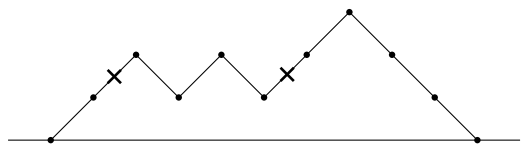


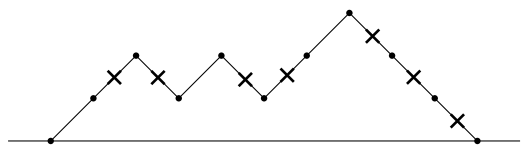


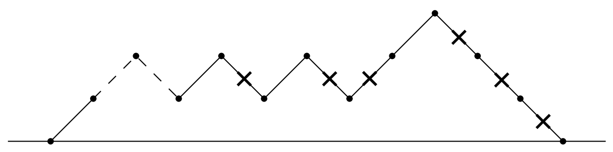
We will show that actually what we have done is a bijection between our new Dyck paths with exactly k marks, and Dyck paths of length $2k$ with all the U-steps at level 0 unmarked. To do so, we will construct an inverse for each element: First of all we mark all the D-steps, then let us go across the path from left to right, whenever we find a marked U-step it means that just before there was a non-marked D-step x and let us suppose that it is at level $k \geq 1$, it remains to know where was its correspondent U-step. If we look at all the points at level k, then between two consecutive of them there is a path with a $\cup$ form, which we will call A-paths, or with a $\cap$ form, which we will call B-paths. The U-step will be, if we go through the path starting at x and going backwards (from right to left), at the beginning of a A-path, or at the last possible point of level k. In other words, if the path is of the form $\alpha AB_1B_2 \dots B_nU^*\beta$, then it will remain as $\alpha AUB_1B_2 \dots B_nDU\beta$.

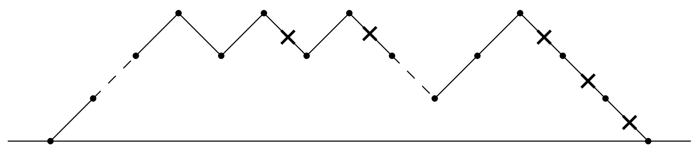
This happens because if we put an A-path in between, there will be steps at level k between the U-step and the D-step, which contradicts the assumption that they were correspondent, and if we don't put in between all the possible B-paths, then the place where we put the U-step would be another valley at level k, which would contradict the assumption that if two valleys were at the same level we would chose the first one. After applying these transformations we just have to unmark all the U-steps and we are done. Hence the inverse is uniquely determined and our application is a bijection. □

Here is an example of the inverse application:









This last bijection allows us to encode a simple outerplanar map in an easy way:

Corollary 4.5. *There exists a way to encode every simple outerplanar map with $n+1$ nodes using $3n$ bits.*

Proof. We can translate the sequence of D-steps and marked U-steps with the following prefix-free alphabet:

0 $\leftarrow$ D-step
10 $\leftarrow$ non-marked U-step
11 $\leftarrow$ marked U-step

Since every simple outerplanar map with $n+1$ nodes uses exactly n D-steps and n U-steps of some kind, the resulting sequence of bits has fixed length $3n$. $\square$

This encoding improves the results of [**2**], in which the encoding of simple outerplanar maps into sequences of asymptotically $3n$ bits is done in a much more complicated way.

5. Random generation of Planar Maps

The aim of this section is to perform an algorithm that generates planar maps of a given size with equal probability to appear. This may seem hard at first sight, but we have already shown two ways of constructing planar maps from binary trees or well labeled trees, which are much simpler to generate.

We will actually do it both ways with linear time and memory cost. Here is a sketch of each of them:

Algorithm using binary trees:

(1) Generation of a random Dyck path of length $n =$ [Map Size].
(2) Construction of a binary tree with n interior nodes from the generated Dyck path.
(3) Construction of the balanced blossomed binary tree.
(4) Application of Schaeffer's first bijection to obtain a quadrangulation of size $2n$.
(5) Obtaining the planar map associated to the quadrangulation.

Algorithm using well labeled trees:

(1) Generation of a random Dyck path of length $n =$ [Map Size].
(2) Construction of an ordered tree with $n+1$ nodes from the generated Dyck path.
(3) Construction of the well labeled tree from the tree obtained previously.
(4) Application of Schaeffer's second bijection to obtain a quadrangulation of size $2n$.
(5) Obtaining the planar map associated to the quadrangulation.

We will focus on steps 1,2 and 3 of each algorithm since we already know steps 4 and 5. However, we will carefully explain how to perform these algorithms in C++ in the last section of our work.

5.1. Algorithm using binary trees. Generation of a random Dyck path

Let S be the set of all the Dyck paths of length n (with n U-steps and n D-steps). Now let us add to each of the elements of S a D-step at the end. Now, since all the paths of S end at level -1, they have n U-steps and $n+1$ D-steps and they remain at non-negative levels until the last step.

We claim that for every sequence s of n U-steps and D-steps, there exists a unique cyclic shift σ_s such that $\sigma_s(s) \in S$.

If this claim is true, then we just have to generate a random sequence of n U-steps and $n+1$ D-steps, and find this cyclic shift. Since every element of S can be generated by exactly $2n+1$ sequences, they would appear with the same probability. We would just need to erase the last step to obtain a random generated Dyck path.

It remains to prove the claim: we will show that the only possible cyclic shift is the one that takes the first vertex at the lowest level to the starting position (hence the computation can be done in linear time!).

Let us notice that if we put a vertex v_i (with $i \neq 0$)) at level h_{v_i} in the first place it means that the nodes from v_i to v_{2n+1} will increase $-h_v$ levels, while the nodes from v_0 to v_{i-1} will increase by $-h_{v_i} - 1$. This means that if v_i is not at the lowest level, or if v_i is not the first of the vertices at the lowest level, v_j, then v_j will clearly be under level 0 after applying the cyclic shift. Reciprocally, if we pick v_j, since all h_{v_k} are strictly greater than h_{v_j} for $k < j$, and all the h_{v_k} are greater than h_{v_j} for $k \geq j$, all the heights will be non-negative after applying the cyclic shift, and the claim follows.

Therefore, the steps would be the following:

(1) Generation of a random sequence of n U-steps and $n+1$ D-steps.
(2) Do a cyclic shift that puts the first node at the lowest level in the first place
(3) Erase the last step.

Construction of the binary tree from the Dyck path

For this step we will use a well known bijection between Dyck paths of length n and binary trees with n internal nodes.

Given a Dyck path, first we create the root node and next we start going along the path starting from the second step (since the first corresponds to the root, which we already created). Now, every tine we find an U-step we create an interior node in the first available place, which is defined as the first missing descendant on an interior node by going in counterclockwise order[4], while every time that we find a D-step we create a leaf in the first available place. After going across every step, we create a leaf at the last available place. We have created a tree with a total of n nodes with two descendants and $n+1$ nodes of degree 1, which is a binary tree.

This application is bijective because we can construct its inverse. Given a binary tree with n interior nodes, we start at the root and we draw an U-step. Next, we go along the tree in counterclockwise order, and every time we find a new interior node we create an U-step, while every time we find a leaf we create a D-step. We just have to skip the last leaf and we will have a path with n U-steps and n D-steps. This path is always above level 0 by an analogous reasoning to the remark.

Construction of the balanced blossomed binary tree

In this step we have to add randomly a blossom to each of the internal nodes of our binary tree to get a randomly generated blossomed binary tree.

[4]Remark: This spot always exists because since a Dyck path is always above level 0, the subtree created after k steps will always have more internal nodes than leaves for every k, while the number of leaves in a binary tree is exactly one more than the number of internal nodes.

After applying the first step in Schaeffer's bijection we should be able locate the two leaves that are not paired. Now we just have to choose as root one of the two parents of the leaves to get a balanced blossomed binary tree[5].

5.2. Algorithm using well-labeled trees. Since we know already how to generate a random Dyck path, we will give details for the construction of the ordered tree and then the well labeled tree.

Construction of a random ordered tree

As in the case of the binary tree with n interior nodes, there is a well known bijection between Dyck paths of length n and ordered trees of size $n + 1$.

First of all we create the root of the tree and we set our initial position as the root. Next, we go across the Dyck path (this time starting in the first step), and whenever we find an U-step we create a descendant from our current position at the last possible place in counterclockwise order and we update our position to the newly created descendant. Else, if we find a D-step, we just have to change our current position to its parent. The fact that we never go below level 0 implies that we are never updating our position to a parent of the root.

This application is one-to-one because we can also construct its inverse: given a tree, we start at the root and read it in counterclockwise order. Whenever we move from a parent to a descendant we create a U-step and we move in the opposite direction we create a D-step. The resulting path is a Dyck path of length n since the tree has n edges that we go across in both directions. Moreover every edge is crossed always first in the parent-child way, and therefore we are never going below level 0. A straightforward computation show that these two applications are inverses one of each other.

Construction of a well labeled tree

To construct a well labeled tree we will do the following: first we generate a random ordered tree. We give the root label 1, and then we assign labels recursively in the following way: if x is the parent's label, then we will assign randomly one of the three numbers $x - 1, x, x + 1$.

The resulting tree has no need to be well labeled since we can get values of 0 or lower. In this case we will pick randomly one of the vertices with the lowest label and we will make it the new root. To make the tree well labeled we just have to add to all the vertices the necessary value to make the root reach the value 1.

[5]Actually we don't need the tree to be balanced, since we would get quadrangulations with the same probability because all the equivalent classes have the same cardinality.

6. Codes

Finally, we will give a sample of a code in C++ that generates random rooted maps of a given size n in which all of them have equal probability to appear.

We will follow the first of the two algorithms explained in the last section: the one that uses binary trees.

First of all we will be using these C++ libraries, typedefs and structures:

```
#include <iostream>
#include <algorithm>
#include <vector>
#include <ctime>
#include <cstdlib>
#include <stack>
#include <queue>

using namespace std;

typedef vector <int> VI;
typedef vector <VI> VII;
typedef stack <int> STI;
typedef vector <bool> VB;

struct BN{
  int id;
  int asc;
  int dright;
  int dleft;
  int random;
};

struct BNlabel{
  int lasc;
  int lright;
  int lleft;
  int lrandom;
};

struct RM{
  VII Nodes;
  VII Edges;
  VII Faces;
};
```

The struct BN refers to a node of a blossomed binary tree, where each of them has an id, the id of its parent (asc), the id of its two childs (dright and dleft), and the position of the random blossom (random).

BNlabel is the struct in which we store the labels of the semiedges of each of the nodes of the blossomed binary tree, the notation is the same as in the previous struct.

Finally the struct RM refers to a rooted map, which is stored using the permutation coding. Each permutation is stored in a vector of vector of integers, since we can represent a permutation in its cycle decomposition, and store each cycle in a vector.

Now that we are done with the basics we will start with the functions:

First of all we have to generate a random Dyck path of length n. We have the following function:

Name: Dyck
Purpose: Generates a random Dyck path of length n, given by input
Input: An integer n
Output: Random Dyck path of length n, stored as a vector of 1s and -1s, which means U-steps and D-steps respectively
Cost: $\Theta(n)$

```
//Returns a random Dyck path of length n
VI dyck (int n){
  /*First we create an arbitrary Dyck path
  with n U-steps and n+1 D-steps*/
  int N = 2*n+1;
  VI dyck(N, -1);
  for (int i = 0; i < n; ++i) dyck[i] = 1;
  //We randomize it with a random shuffle
  random_shuffle (dyck.begin(), dyck.end());
  int level = 0, minlevel = 0, posmin = 0;
  /*We search for the first time we
  get to the minimum level*/
  for (int i = 0; i < N; ++i){
    if (dyck[i] > 0) ++level;
    else --level;
    if (level < minlevel){
      posmin = i+1;
      minlevel = level;
    }
  }
  //We apply the cyclic shift and return the new vector
  VI Dyckfinal(N-1);
  for (int i = 0; i < N-1; ++i) Dyckfinal[i] = dyck[(posmin+i)%N];
  return Dyckfinal;
}
```

Next we have to create a binary tree from the Dyck path:

Name: BT
Purpose: Generates a binary tree of a given size n
Input: An integer n
Output: Random binary tree of size n, stored as a vector of BN
Cost: $\Theta(n)$

```
//Returns a random binary tree with n+1 nodes
```

```
vector <BN> BT(int n){
  //We generate a random Dyck path
  VI Dyck = dyck(n);
  //Initializing the binary tree with a given size
  BN Z;
  Z.asc = Z.dleft = Z.dright = -1; //-1 = !Exist
  vector <BN> bt(2*n+1, Z);
  for (int i = 0; i < 2*n; ++i) bt[i].id = i;
  //We start moving from the root (position 0)
  int actpos = 0;
  for (int i = 1; i < 2*n; ++i){
    if (Dyck[i] > 0){
      bt[i].asc = actpos;
      if (bt[actpos].dleft < 0) bt[actpos].dleft = i;
      else bt[actpos].dright = i;
      actpos = i;
    }
    else{
      bt[i].asc = actpos;
      if (bt[actpos].dleft < 0) bt[actpos].dleft = i;
      else bt[actpos].dright = i;
      while (bt[actpos].dright > 0) actpos = bt[actpos].asc;
    }
  }
  //Adding the last leaf and returning the tree
  bt[2*n].asc = actpos;
  bt[actpos].dright = 2*n;
  return bt;
}
```

Now we have to create the map from the binary tree. We will use the following function:

Name: Random_Map
Purpose: Creates a random map of a given size n
Input: An integer n
Output: A random map of size n, stored as a RM.
Cost: $\Theta(n)$

```
//returns a random map of size n
RM Random_Map(int n){
  //We create a random binary tree of size n
  vector <BN> bt = BT(n);
  //Assigning a blossom randomly to each interior node
  for (int i = 0; i < 2*n+1; ++i) if (bt[i].dleft != -1) bt[i].↩
      random = random_mod(3);
  VI Labels; //Stores the labels of the semiedges comming from leafs↩
       or blossoms
  VI Types; //Stores if the labels come from leafs(1) of blossoms(0)
  VI Edge(2);
  VI Node(4);
  VII Edges; //Stores the edges with the permutation coding
  VII Nodes; //Stores the nodes with the permutation coding
  int actpos = 0; //Keeps track of our actual position
int cont = 0; //Used to asign labels to the semiedges
BNlabel Empty;
Empty.lasc = Empty.lleft = Empty.lright = -1;
vector <BNlabel> Visit(2*n+1, Empty);
//We start in position 0 and start moving counterclockwise
while (actpos != -1){
  if (bt[actpos].dleft == -1){
    //If it has no sons we go back
    actpos = bt[actpos].asc;
  }
  else {
    //Now we are in the case in which actpos is interior
    if (Visit[actpos].lleft == -1){
      //If we haven't labeled the left child yet:
      if(bt[actpos].random == 0){
        //Case in which the blossom is before the left child
        //We put him label "cont" of type 0 and we update cont
        Visit[actpos].lrandom = cont;
        Labels.push_back(cont);
        Types.push_back(0);
        ++cont;
      }
      //We label the semiedge with cont
      Visit[actpos].lleft = cont;
      if (bt[bt[actpos].dleft].dleft != -1){
        //If the edge doesn't end in a leaf we label the other
        //semiedge with cont+1 and add the edge (cont, cont+1)
        Visit[bt[actpos].dleft].lasc = cont+1;
        Edge[0] = cont;
        Edge[1] = cont+1;
        Edges.push_back(Edge);
        cont+=2;
      }
      else {
        //If the edge end in a leaf we add a label cont of type 1
        Labels.push_back(cont);
        Types.push_back(1);
        ++cont;
      }
      actpos = bt[actpos].dleft;
    }
    else if (Visit[actpos].lright == -1){
      //Case in which we have unlabeled the right child
      if (bt[actpos].random == 1){
        //Case in which the blossom is just before
        Visit[actpos].lrandom = cont;
        Labels.push_back(cont);
        Types.push_back(0);
        ++cont;
      }
      //Same procedure as before
      Visit[actpos].lright = cont;
      if (bt[bt[actpos].dright].dleft != -1){
        Visit[bt[actpos].dright].lasc = cont+1;
        Edge[0] = cont;
        Edge[1] = cont+1;
        Edges.push_back(Edge);
        cont+=2;
      }
      else {
        Labels.push_back(cont);
        Types.push_back(1);
        ++cont;
      }
      actpos = bt[actpos].dright;
    }
    else {
      //Case when all childs are labeled
      if (bt[actpos].random == 2){
        //Case when the blossom is just after the right child
        Visit[actpos].lrandom = cont;
        Labels.push_back(cont);
        Types.push_back(0);
        ++cont;
      }
      //We return to the parent
      actpos = bt[actpos].asc;
    }
  }
}
//Putting label and type to the last leaf
Visit[0].lasc = cont;
Labels.push_back(cont);
Types.push_back(1);
//Now we have to join the leafs with the blossoms
int N = Labels.size();
STI S;
VB used (N, false);
for (int i = 0; i < 2*N; ++i){//After at most two cycles we will be done
  if (used[i%N]) continue; //If we have already joined it we return
  if (Types[i%N] == 0){
    //If we find a blossom we put it in a stack
    S.push(Labels[i%N]);
    used[i%N] = true;
  }
  else{
    //We join the leaf with the blossom on the top of the stack
    if (not S.empty()){ //If there is any... else we continue
      Edge[0] = S.top();
      S.pop();
      Edge[1] = Labels[i%N];
      Edges.push_back(Edge);
      used[i%N] = true;
    }
  }
}
//We join the remaining two leaves
int pos = 0;
for (int i = 0; i < N; ++i){
  if (not used[i]){
   Edge[pos] = Labels[i];
   ++pos;
  }
}
```

```
  Edges.push_back(Edge);
  //Now it is time to encode the nodes
  for (int i = 0; i < 2*n+1; ++i){
    if (bt[i].dleft != -1){
      //We have just to consider the three possible positions of the blossom
      if (bt[i].random == 0){
        Node[0] = Visit[i].lasc;
        Node[1] = Visit[i].lrandom;
        Node[2] = Visit[i].lleft;
        Node[3] = Visit[i].lright;
      }
      else if (bt[i].random == 1){
        Node[0] = Visit[i].lasc;
        Node[1] = Visit[i].lleft;
        Node[2] = Visit[i].lrandom;
        Node[3] = Visit[i].lright;
      }
      else{
        Node[0] = Visit[i].lasc;
        Node[1] = Visit[i].lleft;
        Node[2] = Visit[i].lright;
        Node[3] = Visit[i].lrandom;
      }
      Nodes.push_back(Node);
    }
  }
  //Finally we create the quadrangulation
  RM M;
  M.Faces = Nodes;
  M.Nodes = comp_perm(Nodes, Edges);
  M.Edges = Edges;
  return Quad_to_Map (M); //We return its conversion to map
}
```

This last function uses several sub-functions to help simplifying the code. For instance we have the following functions that are needed to compose permutations:

Name: Cicperm
Purpose: Transforms a permutation in its cycle decomposition to its original form
Input: A permutation σ in its cycle decomposition, stored as a vector of vector of integers
Output: σ in its original form, stored as a vector of integers where the i-th position corresponds to $\sigma(i)$
Cost: $\Theta(n)$, where n is the size of the permutation

```
VI cicperm(VII &a){
  int n = a.size();
  int length = 0;
  for (int i = 0; i < n; ++i) length += a[i].size();
  VI P(length);
  for (int i = 0; i < n; ++i){
    int m = a[i].size();
    for (int j = 0; j < m; ++j) P[a[i][j]] = a[i][(j+1)%m];
  }
  return P;
}
```

Name: Permcic
Purpose: Transforms a permutation in its original form into its cycle decomposition
Input: A permutation σ in its original form
Output: σ in its cycle decomposition. Stored in a vector of cycles, each of them represented as a vector of integers
Cost: $\Theta(n)$, where n is the size of the permutations

```
VII permcic (VI &a){
  int n = a.size();
  VB used(n, false);
  VII P;
  int cont = -1;
  for (int i = 0; i < n; ++i){
    if (not used[i]){
      P.push_back(VI(0));
      ++cont;
      int j = i;
      while (not used[j]){
        P[cont].push_back(j);
        used[j] = true;
        j = a[j];
      }
    }
  }
  return P;
}
```

Name: Comp_perm
Purpose: Composes two permutations given by input
Input: Two permutations σ and τ in their cycle decomposition, both of the same size
Output: $\sigma \circ \tau$ in its cycle decomposition
Cost: $\Theta(n)$, where n is the size of the permutations

```
VII comp_perm(VII &a, VII &b){
  int n = a.size();
  VI p1 = cicperm(a);
  VI p2 = cicperm(b);
  n = p1.size();
  VI P(n);
  for (int i = 0; i < n; ++i) P[i] = p1[p2[i]];
  return permcic(P);
}
```

Finally we have the function that transforms a quadrangulation into its map form:

Name: Quad_to_Map
Purpose: Transforms a quadrangulation into its correspondant map
Input: A quadrangulation M, given as a RM struct
Output: Its correspondent map, given as a RM struct
Cost: $\Theta(n)$, where n is the size of the quadrangulation

```
//Returns the map associated to a quadrangulation
RM Quad_to_Map (RM &M){
  int n = M.Edges.size();
  n*= 2;
  int m = M.Nodes.size();
  VII Adj = Adjacency(M); //We create the adjacency matrix
  VI Marks(m, -1);
  //Now we associate the endpoints of the diagonals of each face
  VI Matching(n);
  for (int i = 0; i < (int)M.Faces.size(); ++i){
    for (int j = 0; j < 4; ++j) Matching[M.Faces[i][j]] = M.Faces[i↩
        ][(j+2)%4];
  }
  //We do a BFS starting from node 0 to bicolor the quadrangulation
  int InitialNode = 0;
  Marks[InitialNode] = 0; //We assign color 0 to the first node
  queue <int> Q;
  Q.push(InitialNode);
  while (not Q.empty()){
    int CurrentNode = Q.front();
    Q.pop();
    int Color = Marks[CurrentNode];
    for (int i = 0; i < (int)Adj[CurrentNode].size(); ++i){
      if (Marks[Adj[CurrentNode][i]] == -1){
        Marks[Adj[CurrentNode][i]] = 1 - Color;
        Q.push(Adj[CurrentNode][i]);
      }
    }
  }
  int cont = 0; //Keeps the label of the new nodes
  VI Edge(2);
  VII FEdges; //Edges of the final map
  VII FNodes; //Nodes of the final map
  VI Seen(n, -1);
  for (int i = 0; i < m; ++i){
    if (Marks[i] == 0){ //We keep the nodes with color 0
      int k = M.Nodes[i].size();
      VI Node(k); //The degree of each node is preserved
      for (int j = 0; j < (int)M.Nodes[i].size(); ++j){
        Node[j] = cont;
        Seen[M.Nodes[i][j]] = cont;
        ++cont;
        if (Seen[Matching[M.Nodes[i][j]]] != -1){
          //We join it only with the diagonals of the faces, which ↩
              are stored in the Matching vector
          Edge[0] = Node[j];
          Edge[1] = Seen[Matching[M.Nodes[i][j]]];
          FEdges.push_back(Edge);
        }
      }
      FNodes.push_back(Node);
    }
  }
  //We join the final nodes, edges and faces into the final map and ↩
      return it
  VII FFaces = comp_perm(FNodes, FEdges);
  RM FM;
  FM.Nodes = FNodes;
  FM.Edges = FEdges;
  FM.Faces = FFaces;
  return FM;
}
```

Finally, we had to use a function that returns the adjacency list of a given map in order to do a BFS:

Name: Adjacency
Purpose: Returns the adjacency list of a given map
Input: A map M, stored as a RM struct
Output: The adjacency list of M stored as a vector of vectors of integers, where the i-th position of the first vector contains a vector with the id of the nodes adjacent to the i-th node
Cost: $\Theta(n)$, where n is the size of the map

```
VII Adjacency (RM &M){
  //Returns the Adjacency matrix of a map
  int n = M.Edges.size();
  n*= 2;
  VI EN (n);
  int m = M.Nodes.size();
  for (int i = 0; i < m; ++i){
    for (int j = 0; j < (int)M.Nodes[i].size(); ++j){
      //We associate the semiedges with their respective nodes in ↩
          the EN vector
      EN[M.Nodes[i][j]] = i;
    }
  }
  VII Adj(m, VI(0));
  for (int i = 0; i < n/2; ++i){
    //Now we just have to join the nodes associated to the endpoints↩
        of every edge
    int nod1 = EN[M.Edges[i][0]];
    int nod2 = EN[M.Edges[i][1]];
    Adj[nod1].push_back(nod2);
    Adj[nod2].push_back(nod1);
  }
  return Adj;
}
```

7. Execution and Applications

7.1. Testing the Code. To test the code's correctness we can compute known parameters of planar maps, for instance let us take the mean of the number of vertices generated with our code.

After executing 10 times the function Random_Map for $n = 10000$ we get the following values for the number of vertices:

$$\begin{aligned} v_1 &= 5018 \\ v_2 &= 5054 \\ v_3 &= 5053 \\ v_4 &= 4990 \\ v_5 &= 5101 \\ v_6 &= 4978 \\ v_7 &= 4961 \\ v_8 &= 5042 \\ v_9 &= 5006 \\ v_{10} &= 4913 \end{aligned}$$

Since this value has mean $n/2 = 5000$ and has typical deviation of $\Theta(\sqrt{n})$, the results obtained seem to be in the right way. To give a more precise evaluation let us make the experiment 1000 times and compute the mean and standard deviation. We get

$$v_M = 5000,22$$
$$\sigma_v = 36,8697$$

Now let us take $n = 1000000$ and do the experiment 100 times. We obtain

$$v_M = 500526$$
$$\sigma_b = 455,29$$

The values obtained seem plausible, hence we can continue to find other parameters.

7.2. k-Core of a map. The k-core of a map is defined as the maximal connected submap such that every node has degree at least k. It can be constructed by repeatedly erasing the nodes with degree less than k.

It is easy to see that the 2-core of a map is connected, but it may not happen for larger values of k. The expected size of the 2-core of a graph has been proved to be of the order of $\frac{\sqrt{6}}{3}n$ with variance $n/6$, but we have still no results about the behaviour of the maximal connected component of the 3-core. However it is believed that its size depends linearly on n.

We will use the following programs to compute experimentally these parameters for $k = 3, 4$ and 5.

Name: Maxcomp
Purpose: Computes the size of the maximal connected component of a graph
Input: The adjacency list of a graph, given as a vector of vector of integers

Output: An integer denoting the size of the maximal connected component
Cost: $\Theta(n)$, where n is the number of edges in the graph

```
int Maxcomp (VII &Adj){
  //Returns the maximal connected component of the graph with ↩
      adjacency list Adj
  int n = Adj.size();
  if (n == 0) return 0;
  VI visit(n, false);
  int res = 0;
  for (int i = 0; i < n; ++i){
    if (not visit[i]){
      //We do a BFS from each non-visited vertex and we keep the ↩
          maximum of the components visited
      queue <int> Q;
      Q.push(i);
      visit[i] = true;
      int cont = 0;
      while (not Q.empty()){
        int t = Q.front();
        Q.pop();
        int m = Adj[t].size();
        cont+= m;
        for (int j = 0; j < m; ++j){
          if (not visit[Adj[t][j]]){
            visit[Adj[t][j]] = true;
            Q.push(Adj[t][j]);
          }
        }
      }
      res = max(res, cont);
    }
  }
  return res/2;
}
```

Name: Computecore
Purpose: Computes the size of the maximal connected component of the k-Core of a random map of size n given n and k
Input: Two integers k and n
Output: An integer m, representing the size of the maximal connected component of the k-core of a randomly generated map of size n
Cost: $\Theta(n)$

```
int Computecore(int n, int k){
  RM M = Random_Map(n);
  int m = M.Nodes.size();
  VII Adj = Adjacency(M);
  VI Sizes(m);
  for (int i = 0; i < m; ++i){
    Sizes[i] = Adj[i].size();
  }
  //We put all the nodes with size less than k in a queue and we mark them as visited
  VB visited(m, false);
  queue <int> Q;
  for (int i = 0; i < m; ++i){
    if (Sizes[i] < k){
      visited[i] = true;
      Q.push(i);
    }
  }
  while (not Q.empty()){
    int a = Q.front();
    Q.pop();
    for (int i = 0; i < (int)Adj[a].size(); ++i){
      //We decrease the size of the neighbours of the erased nodes and add them to the queue if their size goes below k
      int x = Adj[a][i];
      --Sizes[x];
      if (not visited[x] and Sizes[x] < k){
        Q.push(x);
        visited[x] = true;
      }
    }
  }
  //We relabel every non-erased noce
  int cont = 0;
  VI Newlabel(m, -1);
  for (int i = 0; i < m; ++i){
    if (not visited[i]){
      Newlabel[i] = cont;
      ++cont;
    }
  }
  //Create new Adjacency matrix without the erased nodes
  VII NewAdj(cont, VI(0));
  for (int i = 0; i < m; ++i){
    if (not visited[i]){
      for (int j = 0; j < (int)Adj[i].size(); ++j){
        if (not visited[Adj[i][j]]) NewAdj[Newlabel[i]].push_back(Newlabel[Adj[i][j]]);
      }
    }
  }
  //We return the maximal connected component of the remaining graph
  return Maxcomp(NewAdj);
}
```

Executing them for several values we obtain the following table, where n is the size of the original map, c is the number of cases, m is the mean of the size of the maximal connected component of the k-core and V is its variance:

Values for the 2-core

n	c	m	m/n	V	V/n
1000	10000	$816,083$	$0,816083$	$166,047$	$0,166047$
5000	2000	$4082,14$	$0,816428$	$860,133$	$0,172027$
20000	1000	$16327,6$	$0,816379$	$3408,91$	$0,170445$
100000	500	$81654,8$	$0,816548$	$17950,8$	$0,179508$
1000000	100	816507	$0,816507$	157311	$0,157311$

Values for the 3-core

n	c	m	m/n	V	V/n
1000	10000	$598,062$	$0,598062$	$1883,3$	$1,8833$
5000	2000	$2992,5$	$0,59850$	$22116,6$	$4,42333$
20000	1000	$11996,2$	$0,599812$	164054	$8,20272$
100000	500	$59900,8$	$0,599008$	3034240	$30,3424$
1000000	100	600468	$0,600468$	4873840	$4,87384$

Values for the 4-core

n	c	m	m/n	V	V/n
1000	10000	$347,064$	$0,347064$	$5036,05$	$5,03605$
5000	2000	$1718,22$	$0,343644$	$66445,4$	$13,2891$
20000	1000	$6881,33$	$0,344066$	561657	$28,0828$
100000	500	$34703,5$	$0,347035$	5395443	$53,9544$
1000000	100	347128	$0,347128$	18334530	$183,345$

Values for the 5-core

n	c	m	m/n	V	V/n
1000	10000	$152,124$	$0,152124$	$3940,36$	$3,94036$
5000	2000	$588,686$	$0,117737$	$51737,4$	$10,3475$
20000	1000	$1857,95$	$0,0928975$	495159	$24,758$
100000	500	$6873,2$	$0,068732$	7078913	$70,78913$
1000000	100	$42230,2$	$0,04223$	290893243	$290,893$

Given these results we can formulate the following conjecture:

Conjecture 7.1. *The expected value of the size of the k-core of random a map of size n is linear in n for $k = 3$ and 4, but sublinear for $k \geq 5$.*

References

[1] G. Schaeffer, *Conjugaison d'arbres et cartes combinatoires aléatoires*. PhD thesis, Université Bordeaux I (1998).

[2] N. Bonichon, C. Gavoille, N. Hanusse, *Canonical Decomposition of Outerplanar Maps and Application to Enumeration, Coding and Generation*, Journal of Graph Algorithms and Applications, vol. 9, no. 2, pp. 185 - 204 (2005).

[3] W.T. Tutte. *A census of planar maps* Canad. J. Math., 15:249?271, (1963)

[4] I.P. Goulden and D.M. Jackson. *Combinatorial enumeration*, Volume 19. Wiley New York (1983)

[5] O. Bernardi, *Scaling limit of random planar maps*, Workshop on randomness and enumeration (2008)