Scientific Computation Project 4

Amaury Francou — CID : 01258326

March 31, 2022

Question 1

We implement a function to compute the WT distance between all pairs of nodes i and j, using the matrix-vector form statement $d_{ij}^2 = \|D^{-1/2}m_i - D^{-1/2}m_j\|^2$. We first compute the matrix product $D^{-1/2}M$ using the pre-implemented `numpy` method, before using the useful `scipy.spatial.distance.cdis` method, which computes the distance between each pair of vectors in two input matrices. As this operation is made row-wise, we use the transpose of the obtained product of matrices.

We further compare the computed WT distances to the conventional distances (number of edges on a shortest path) in a provided graph. We focus on the distances from node zero to all other nodes. This is given by the first row of our distance matrix, computed with `WTdist`. The conventional distances are given by the `shortest_path_length` method from the `networkx` package. The analysis is made in function `analyzeWTdist`. The results are displayed in figure 1. The x-axis gives the considered nodes j, the y-axis the corresponding distances between node zero and said nodes j. The conventional distances have been rescaled by a factor 10 for a better visualness of patterns. We observe that the WT distances are overall smaller in absolute value than the conventional distance corresponding ones. However, both distances roughly 'follow the same variations', for which more distant nodes from zero for one distance corresponds also to more distant nodes from zero for the other distance. Relative lengths to zero remain here globally preserved.

Question 2

We implement the `WTdist2` function that computes $r_{ab}^2 = \|D^{-1/2}\tilde{r}_a - D^{-1/2}\tilde{r}_b\|^2$, for all (a, b) pairs of the m provided communities. We here have that $\tilde{r}_a^T$ is the a-th row of the $m \times N$ matrix R, for which $R_{ij} = \frac{1}{l_i}\sum_{k \in C_i} M_{kj}$. We build R using vectorized `numpy` operations and compute an $m \times m$ matrix Y, storing the r_{ab}^2 entries, by using our previously implemented `WTdist` function.

Question 3

We implement the `main` function which performs the complete WT algorithm. We begin by loading the data and building the probability matrix M. We initialize the variable `Clist`, which is a list of lists containing the nodes belonging to the same community. At first, each node is alone in its own community. We then build the matrix Y, containing the distances between distinct pairs of communities. This matrix is computed *only once*. From Y, we compute the $m \times m$ matrix S containing the cost of merging distinct pairs of communities. This matrix is initialized as $S = \frac{1}{2N}Y$ - following the provided formula - and is fully computed *only once*. We further initialize an m-dimensional vector L containing the number of nodes for each community. Moreover, by implementing a function `makeCmat`, we build an $m \times m$ pseudo-adjacency matrix C, for which $C_{ab} = 1$ if communities a and b are linked, $C_{ab} = 0$ otherwise. This matrix is fully computed *only once.*

After initializations, we enter a master `while` loop in which we proceed with the WT algorithm. We first compute communities a^* and b^* for which $S_{a^*b^*}$ is minimum and $C_{a^*b^*} = 1$. Namely, we find the two linked communities minimizing the merging cost. This is efficiently made using `numpy` pre-implemented methods and slice notations. Recall that `numpy` is a `C`-wrapper, using pre-compiled code.

Before performing the merging, we secondly update the cost matrix S, using the provided update formula in a vectorized way. We update the a^*-th row of S, using previously stored values and the vector L. As S is symmetric, we equalize the a^*-th row and the a^*-th column. We then delete the b^*-th row and column of said matrix, using the `numpy.delete` function.

Thirdly, we update the C matrix, by adding the b^*-th row to the a^*-th row, and rescaling all positive values to one. Namely, community a^* inherits links from community b^* in the merging operation. We delete the one-value that naturally appears on the diagonal (as b^* was linked to a^*). We then copy the a^*-th row on the a^*-th column, as communities previously having a link with b^* now have a link with a^* (C is symmetric). We finish by deleting the b^*-th row and column of C. All said operations are made using vectorized `numpy` methods.

Fourthly, we update vector L, by adding the b^*-th value to the a^*-th value, before deleting it from the array. Finally, we merge community b^* into community a^*, using the `merge` helper function. We recompute the number of communities m, which follows $m_{new} = m - 1$. The stopping condition of the master `while` loop is $m > N_c$, where N_c is the entered desired number of communities. Overall, the function fully computes the working matrices only once, finds the minimizing pair of communities using pre-compiled `numpy` methods and follows single-row-wise vectorized updates in the master loop. In particular, this implementation doesn't recompute any 'linked-pairs dictionary' and leverages a smooth update on a specifically defined pseudo-adjacency matrix. For the provided 467-nodes graph and for $N_c = 2$ the wall time is roughly 1.2 seconds.

Question 4

We apply the WT algorithm to the provided graph representing the functional network of the human brain. The network is divided in two clusters '-' and '+', corresponding to the two hemispheres of the brain. The nodes are evenly separated in those two subclasses.

We first apply the WT method using $t = 6$ and requiring $N_c = 2$ clusters. We display the graph in figure 2 : we color each nodes according to its community (given by the WT algorithm) and label it according to its corresponding hemisphere. Although we could have expected having two clusters similar to the hemispheres classes in the best case scenario, we here see that we have two unbalanced communities. The WT algorithm has produced an extensive community including a majority of 459 nodes out of 467, leaving a smaller second cluster with only 8 nodes out of 467. However, the smaller community is made only of 'positive' hemisphere 'outer' nodes, which somewhat shows consistency in the WT clustering. Indeed, said smaller identified community lies on the 'outer rim' of the graph, is made of similar hemisphere nodes and has inner elements that are connected. As N_c increases, we observe that the more exterior nodes get clustered in their own community first. We re-apply the method, this time using $t = 8$ with $N_c = 2$. We further have that there are still two unbalanced communities (this time with 458 nodes and 9 nodes - see figure 3). Slightly varying t doesn't affect the overall community detection.

While focusing on the hemispheres partition, we observe the existence of two central heaps, each of them gathering positive or negative nodes. We seek to correctly separate said heaps using the WT process. In figure 4, we observe that said separation is effective for $N_c = 6$. Recall that the WT method is based on the fact that random walks tend to remain in the same cluster. In this case, sub-separating the hemispheres into smaller communities (2 for the negative nodes and 4 for the positive nodes), improves the effectiveness of the process.

To understand to what extent this may be valid, we introduce a clustering score :

$$s_{N_c} = \frac{1}{N_c} \sum_{i=0}^{N_c - 1} \max\,(p_i^{(+)}, p_i^{(-)})$$

This score sums the proportion of nodes being in the predominantly represented hemisphere for each cluster. Here $p_i^{(+)}$ is the proportion of positive hemisphere nodes in cluster i. The score s_{N_c} is upper bounded by 1, as in the case where all nodes are in there own cluster ($N_c = 467$), we have $\max\,(p_i^{(+)}, p_i^{(-)}) = 1$.

We scan the parameter N_c and compute corresponding s_{N_c} scores until one reaches 1. We obtain that $s_{N_c} = 1$ first for $N_c = 264$. Albeit for this value of N_c - that is greater than $\frac{N}{2}$ - we have that, in average communities are made of one single node, no opposite hemispheres nodes share a same community. We display the resulting clustered graph in figure 5.

As the value of N_c obtained for a perfect clustering is rather high, we ultimately focus on the effects of parameter t. Said parameter controls the number of steps taken by the random walks. By seeking for a bi-partition ($N_c = 2$) and substantially varying t, we obtain an imperfect though adequate clustering for $t = 10000$. As seen in figure 6, both hemispheres are almost perfectly separated in this case. We finally may suppose that the random walks perhaps need a more important number of steps, to better capture the community structure involved here.

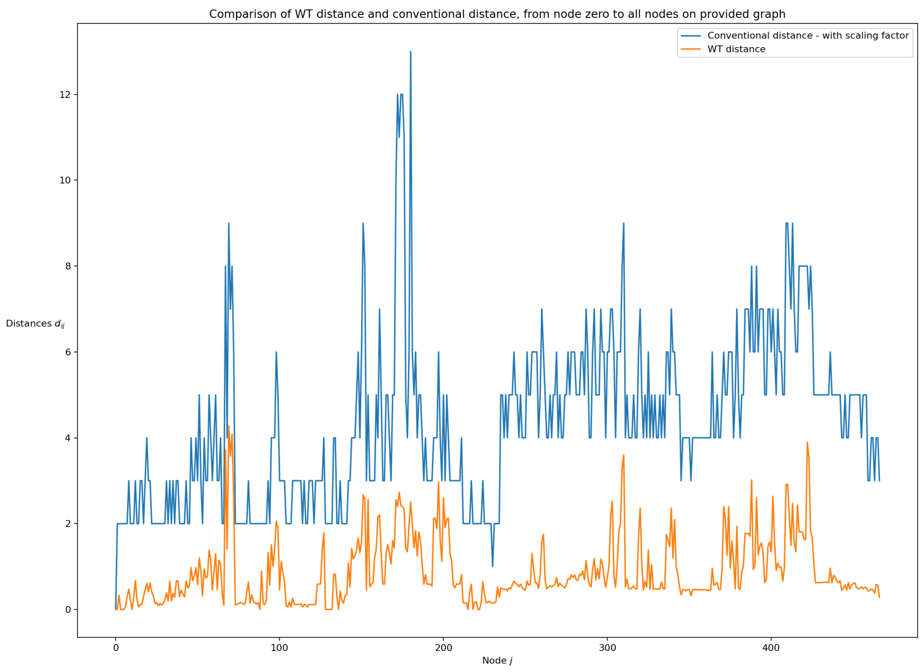

Figure 1: Figure for question 1 — Comparison of WT distance and conventional distance, from node zero to all nodes on provided graph

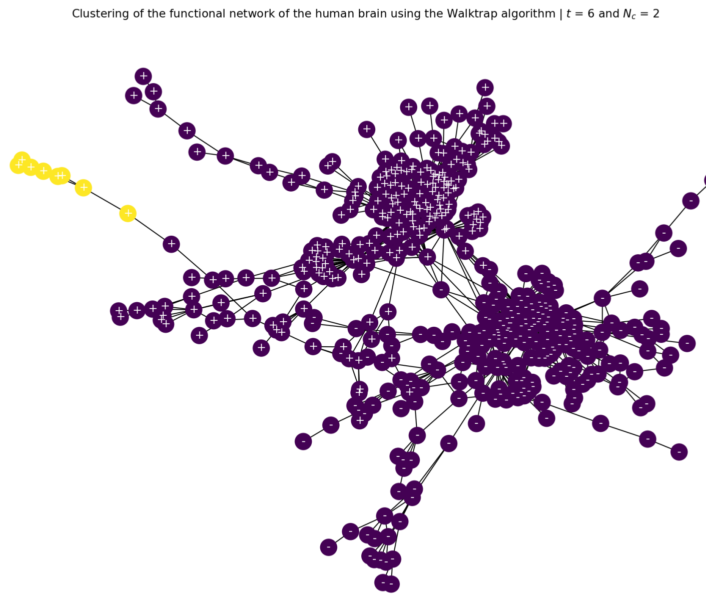

Figure 2: Figure for question 4 — Clustering of the functional network of the human brain using the Walktrap algorithm — $t = 6$ and $N_c = 2$

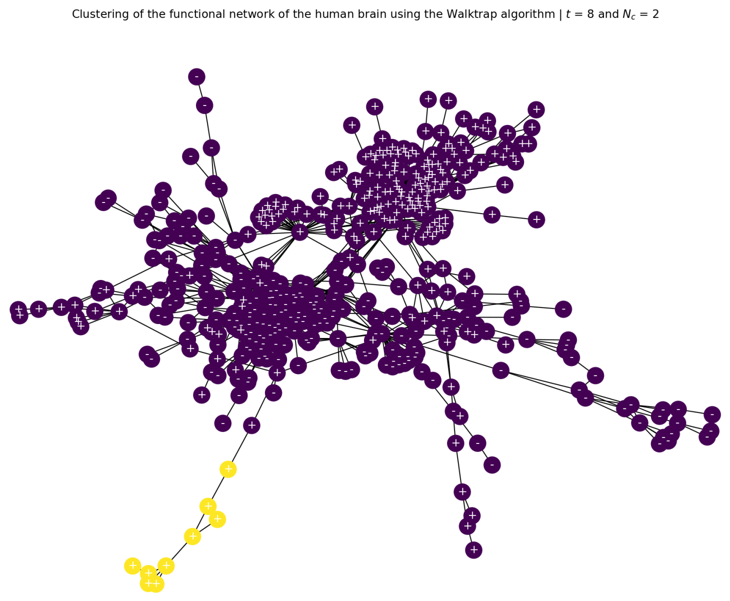

Figure 3: Figure for question 4 — Clustering of the functional network of the human brain using the Walktrap algorithm — $t = 8$ and $N_c = 2$

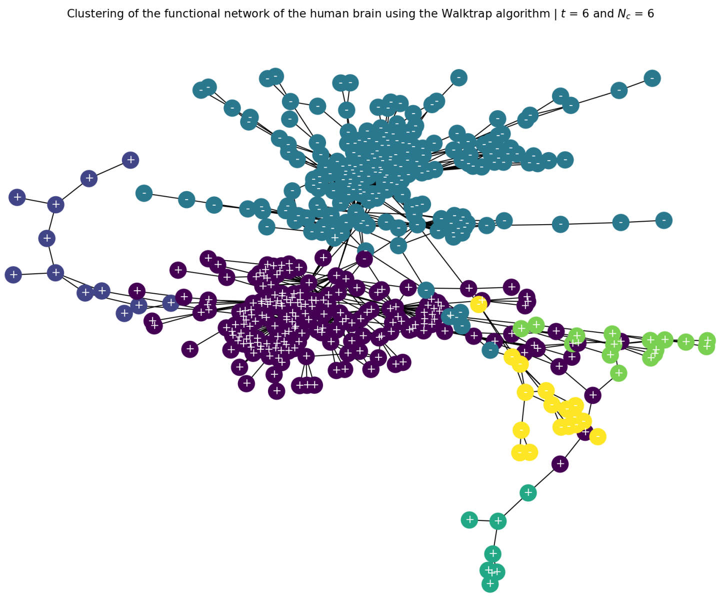

Figure 4: Figure for question 4 — Clustering of the functional network of the human brain using the Walktrap algorithm — $t = 6$ and $N_c = 6$

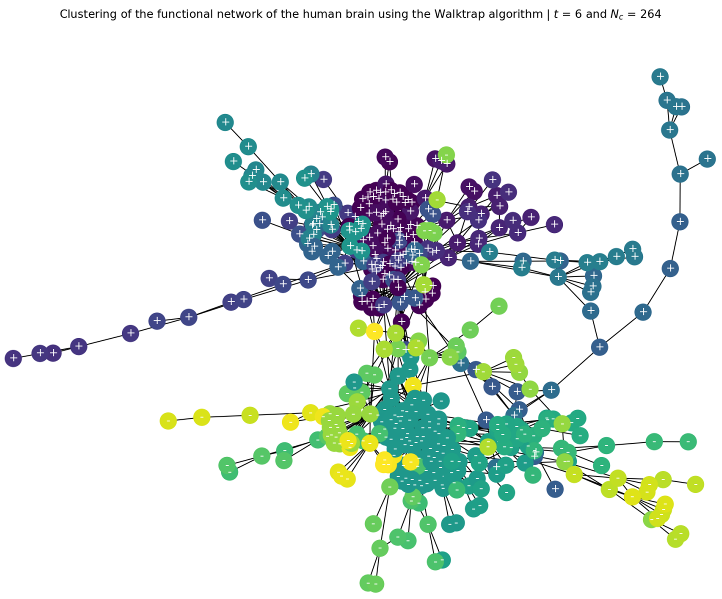

Figure 5: Figure for question 4 — Clustering of the functional network of the human brain using the Walktrap algorithm — $t = 6$ and $N_c = 264$

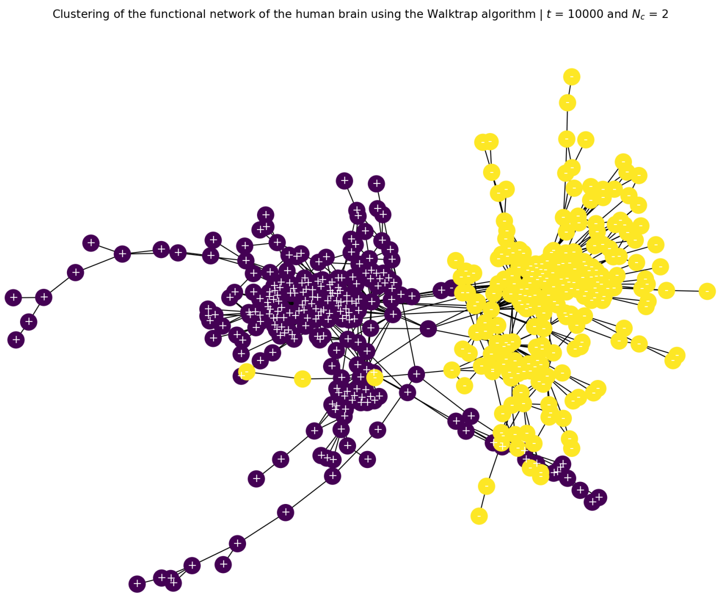
Clustering of the functional network of the human brain using the Walktrap algorithm | t = 10000 and N_c = 2

Figure 6: Figure for question 4 — Clustering of the functional network of the human brain using the Walktrap algorithm — $t = 10000$ and $N_c = 2$