

Approaches to Brain Parcellation using Energy Statistics and Graph Partitioning

Felix Xiao

March 22, 2016

Contents

1	Criteria for Evaluating Parcellations	1
1.1	Within-Parcel Similarity	1
1.2	Between-Parcel Dissimilarity	2
1.3	Graph Cuts	3
1.4	Balance and Jaggedness	4
1.5	Comparing Multiple Parcellations	4
2	Local Search and Graph Growing Heuristics	5
2.1	Unconstrained Add-Edge	5
2.2	Size-Constrained Add-Edge	6
2.2.1	Union-Find	6
2.3	The Contractible Graph Data Structure and Edge-Contraction Algorithm	7
2.3.1	Implementation using Multi-Edge Adjacency List and Priority Queue	9
2.3.2	Results and Optimal Number of Components	11

Abstract

We formulate the task of brain parcellation (dividing the brain into functionally homogenous regions) as a graph partitioning problem. We devise new model-free criteria for validating parcellations based on statistical dependency between parcels using distance correlation. Based off our criteria we pose a new graph cut-type problem called Max Average Within Edge, wherein the objective is to maximize, for each component, the summed weight of all edges with both endpoints in the component divided by the number of edges within the component.

We propose a family of heuristic algorithms for attaining good solutions for MAWE.

Chapter 1

Criteria for Evaluating Parcellations

In chapter one we discussed our graphical approach to the brain parcellation problem. We construct a weighted undirected graph where each vertex is a voxel. The graph reflects the spatial position of the voxels; it connects each vertex to the vertices representing the voxel's six cubically adjacent neighbors. The weights on these edges are sample energy distance correlation statistics between the adjacent voxels in the time series and they measure statistical dependence between the voxels. Let $G(V, E)$ denote this graph, its vertices, and its edges.

In this context, a valid k -fold partition \mathcal{P}_k of the graph G is a collection of vertex subsets (V_1, \dots, V_k) satisfying the following:

1. $V_i \neq \emptyset$ for all $V_i \in \mathcal{P}_k$
2. $\bigcup_{i=1}^k V_i = V$
3. $V_i \cap V_j = \emptyset$ for all $V_i, V_j \in \mathcal{P}_k$
4. V_i is connected (i.e. for every two vertices in V_i , there is a path between them) for all $V_i \in \mathcal{P}_k$

In this chapter we will suggest various criteria for measuring the goodness-of-fit of partitions and discuss their statistical and computational advantages and drawbacks.

1.1 Within-Parcel Similarity

Voxels in the same parcel are ideally highly dependent on one another in the time series of fMRI data. As discussed in the previous chapter, distance correlation is a good measure of dependence. The distance correlation between two random vectors

equals zero if and only if the two random vectors are independent, which is not true of correlation statistics such as Pearson's.

Let $\mathcal{R}(x, y)$ denote the distance correlation between two voxels x and y . Let V and W be any parcels. We will use E_V to denote the set of edges with one end in V and one end not in V , and $E_{V,W}$ the set of edges with one end in V and one in W .

Definition 1.1.1 (Within-Score)

$$\frac{1}{k} \sum_{V \in \mathcal{P}_k} \frac{1}{|V|^2} \sum_{x, y \in V} \mathcal{R}(x, y)$$

The Within-Score is non-spatial; it considers all pairs of voxels equally regardless of whether they are adjacent. As a result, it is a good measure of how much the voxels within each parcel are dependent on each other as a set. The downside of this criterion is that it is very expensive to compute. With over 300,000 voxels in an fMRI data set we would potentially have to compute tens of billions of distance correlation statistics, each of which takes time proportional to the number of samples squared.

An alternative and far less expensive criterion that measures within- parcel similarity works by counting distance correlations between adjacent pairs of voxels.

Definition 1.1.2 (Adjacent-Score)

$$\frac{1}{k} \sum_{V \in \mathcal{P}_k} \frac{1}{|E_{V,V}|} \sum_{(x,y) \in E_{V,V}} \mathcal{R}(x, y)$$

Rather than treat parcels as sets with no spatial information, the Adjacent-Score does the opposite by only considering the pairwise dependency of adjacent voxels. For sparse graphs such as ours, the number of distance correlation computations is proportional to the number of vertices.

An intermediate possibility we did not explore is to consider all pairs of voxels up to some maximum spatial distance from each other and perform a weighted averaging of sample pairwise distance correlations, with weights that depend on spatial distance.

1.2 Between-Parcel Dissimilarity

To evaluate parcellation quality, it is also useful to measure how dependent voxels belonging to different parcels are on each other. To this end we define two criterion similar to the Within-Parcel criterion; a non-spatial metric called the Between-Score and its spatial metric the Boundary-Score.

Definition 1.2.1 (Between-Score)

$$\frac{1}{\binom{k}{2}} \sum_{V, W \in \mathcal{P}_k, V \neq W} \frac{1}{|V||W|} \sum_{x \in V, y \in W} \mathcal{R}(x, y)$$

Definition 1.2.2 (Boundary-Score)

$$\frac{1}{\binom{k}{2}} \sum_{V, W \in \mathcal{P}_k, V \neq W} \frac{1}{|V||W|} \sum_{(x, y) \in E_{V, W}} \mathcal{R}(x, y)$$

Generally both of these quantities are more expensive to compute than their Within-Parcel counterparts. Boundary-Score is easy enough to compute for validation purposes, but does not convey much additional information beyond what the Adjacency-Score does, in the sense that the edges used in the computation of Adjacency-Score are the complement of the edges used in the Boundary-Score.

The ability of distance correlation to generalize to pairs of random vectors of arbitrary dimension gives us another way of computing the dependency between two parcels. The Multivariate Between-Score defined below treats parcels as random vectors and computes the distance correlation at the parcel level rather than voxel level. The result is a measure of non-spatial between-parcel similarity that is also computationally feasible. For this reason we will use Multivariate Between-Score as our primary measure of parcel dissimilarity.

Definition 1.2.3 (Multivariate Between-Score)

$$\frac{1}{\binom{k}{2}} \sum_{V, W \in \mathcal{P}_k, V \neq W} \mathcal{R}(V, W)$$

1.3 Graph Cuts

Closely related to the Boundary-Score is the notion of a graph cut from computer science. A *cut set* is the set of edges with endpoints in different parcels. The *cut weight* is the sum of weights of all edges in the cut set and can be expressed as

$$\frac{1}{2} \sum_{V \in \mathcal{P}_k} \sum_{x, y \in E_V} \mathcal{R}(x, y)$$

The *ratio cut* defined below is a weighted version of the cut weight:

$$\frac{1}{2} \sum_{V \in \mathcal{P}_k} \frac{1}{|V|} \sum_{x, y \in E_V} \mathcal{R}(x, y)$$

The subfield of graph partitioning is concerned with minimizing cut weight, ratio cut, and several other related quantities. Over the last several decades a number of highly effective approximation algorithms have been developed to find partitions of graphs that minimize these quantities. Later chapters will explore how these methods work for brain parcellation.

1.4 Balance and Jaggedness

In addition to the above distance correlation based criteria, there are two additional metrics concerned with parcel shape.

Definition 1.4.1 (Balance)

$$\frac{1}{k} \frac{1}{\max_{V \in \mathcal{P}_k} |V|} \sum_{V \in \mathcal{P}_k} |V|$$

The Balance-Score ranges from 1 (all equally sized parcels) to 0 (two parcels with one of size zero).

Definition 1.4.2 (Jaggedness)

$$\frac{1}{k} \sum_{V \in \mathcal{P}_k} \frac{|E_V|^{\frac{3}{2}}}{|V|}$$

The $\frac{3}{2}$ power makes the ratio invariant on parcel size. For instance, a $n \times n \times n$ cube of vertices would have a compactness of $6^{\frac{3}{2}}$.

1.5 Comparing Multiple Parcellations

Chapter 2

Local Search and Graph Growing Heuristics

We introduce several algorithms for generating brain parcellations. The algorithms in this chapter are all local search heuristics; they begin with n unconnected vertices and iteratively join adjacent ones into components until some stopping criterion is met.

For each algorithm, the resulting parcellation is presented, discussed, and evaluated according to the criteria introduced in the previous chapter.

2.1 Unconstrained Add-Edge

The first and simplest algorithm starts with an empty graph of n vertices and sequentially adds edges between adjacent voxels in order of highest sample distance correlation, until the graph has some prespecified number of connected components k .

We will refer to this algorithm as “Unconstrained Add-Edge”. A naive implementation would re-compute the number of connected components in the graph (using linear-time breadth-first or depth-first search) after each addition of an edge, resulting in a costly $O(EN)$ time complexity. A more efficient implementation takes advantage of the fact that each addition of an edge decreases the number of components in the graph by at most 1. Hence the algorithm needs only to compute the number of connected components after adding $c - k$ edges, where c is the current number of connected components of the graph, beginning at n .

Another implementation uses a binary search-type strategy and is $O((n+E) \log E)$. The idea is to “search” for the last edge to add to the graph by maintaining a range of possible last edges. In each iteration, the algorithm would add to the graph edges 1 to the midpoint of this range, compute the number of connected components, and adjust the range based on whether the number of components is higher or lower than the target K .

The Unconstrained Add-Edge algorithm produces severely imbalanced partitions. In the 100-component graph, there was one component containing over 99.9% of all the vertices in the graph. This leads to a modification that prevents some edges from being added when a size constraint is violated.

2.2 Size-Constrained Add-Edge

The Size-Constrained Add-Edge algorithm works in a similar manner to the unconstrained version, adding edges to the graph in decreasing order of distance correlation. The Size-Constrained version differs by applying a filter to each edge considered, adding the edge only if at least one of the two following conditions are met:

1. At least one of the two components bridge by the edge is of size less than some prespecified parameter s_{\min} .
2. The union of the two components is of size $\leq s_{\max}$.

The restriction on adding new edges was not successful in creating balanced partitions. For sake of completeness, we documented our implementation.

The naive implementation must use BFS/DFS in each iteration to compute the sizes of the two components to be connected by an edge, and hence must have time complexity $O(EN)$. Fortunately, there is a way to sublinearly update information on the components of the graph, using the union-find data structure.

2.2.1 Union-Find

The core Union-Find data structure begins with an empty graph of N vertices and supports two operations. `union(i, j)` adds an edge between vertices i and j . `root(i)` returns an identifier for the component to which vertex i belongs. All vertices in the same component have the same root. We modified Union-Find to support an additional operation. `component_size(i)` returns the number of vertices belonging to the component containing i .

Union-Find represents each component as a rooted tree, with vertices in the graph mapping to nodes in the tree. Information about the tree is stored in two arrays of length N , `parent` and `size`, which are subject to the following invariants.

1. For each node i , `parent[i]` = node i 's parent on the tree, unless i is a root node. If i is a root node, then `parent[i]` = i .
2. Nodes i and j are in the same component if and only if they are in the same tree, if and only if they share the same root node.
3. If i is a root node, then `size[i]` = the size of the component, or the number of nodes in the tree. If i is not a root node, then `size[i]` can be anything.

Algorithm 1 Union-Find

```
function ROOT(i)
  while parent[i]  $\neq$  i do
    i  $\leftarrow$  parent[i]
  end while
  return i
end function

function UNION(i, j)
  parent[root(j)]  $\leftarrow$  root(i)
end function

function COMPONENT_SIZE(i)
  return size[root(i)]
end function
```

A baseline implementation of the three functions is

In addition to the baseline code above, there are two important optimizations:

1. Weighted union maintains information of the sizes of each component so that the root of the smaller component always becomes a child of the larger component's root.
2. Path compression flattens the tree with each call to root. Specifically, when root is called on node i , each node traversed from i to the root has its parent set to be the root.

With these two optimizations, the time complexity of root, union, and component_size has been shown to be at least as good as $O(\log^* N)$ where \log^* is the iterated logarithm, defined as the number of times the natural log must be applied to N so that it becomes less than or equal to 1.

2.3 The Contractible Graph Data Structure and Edge-Contraction Algorithm

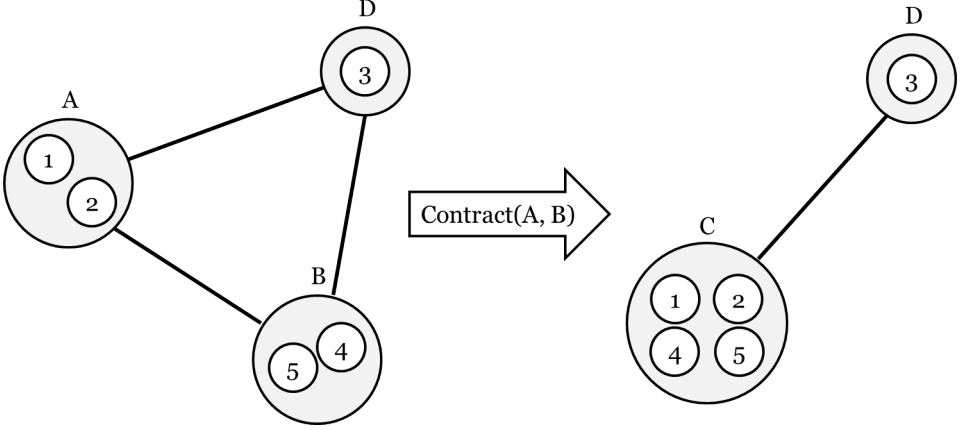
We propose a new data structure called the *Contractible Graph* (CG) for brain parcellation. The rationale behind the CG is a heuristic procedure for partitioning a graph into somewhat balanced components so as to minimize the Boundary-Score (1.2.2) or maximize the Adjacency-Score (??).

The CG is a mapping of the vertices of the original graph to the vertices of a new graph. The vertices of the CG are called *components* and between any two components there exists exactly one weighted edge, henceforth called a *link*. The weight of a link $w(A, B)$ between two components A and B in the CG equals the

average weight of all edges in the original graph between vertices mapped to A and vertices mapped to B . If no such edges exist, the weight of the link is 0. Formally,

$$E(A, B) = \{(i, j) \in E : i \in A, j \in B\}$$

$$w(A, B) = \begin{cases} \frac{1}{|E(A, B)|} \sum_{(i, j) \in E(A, B)} w_{ij} & \text{if } |E(A, B)| > 0 \\ 0 & \text{otherwise} \end{cases}$$



The *size* of a component is the number of vertices it contains. A *contraction* of a link (A, B) in a CG replaces components A and B with a new component (call it C) containing all vertices mapped to A or B , as illustrated in the figure above. Component C has one link to every other component in the CG, whose weights are the mean of the weights of the corresponding vertex edges, or 0 if no edge exists. Thus the contraction operation maintains the link-invariant property of CG. This leads to the Edge-Contraction algorithm, which begins with the original graph with all vertices as singleton components and contracts edges in a certain order until the graph has only k components in all.

Algorithm 2 Edge-Contraction

Input: Undirected positive-weighted graph G and target component number k
 Create a CG from G so that every vertex maps to a unique component

repeat

$\mathcal{S} \leftarrow$ smallest component(s) in the CG

$(A, B) \leftarrow \underset{A \in \mathcal{S}}{\operatorname{argmax}} w(A, B)$

Contract (A, B)

until CG has k components

Output: Components of CG

Why does Edge-Contraction work better than the previous algorithms? The Edge-Contraction algorithm attempts to address two problems of the Size-Constrained

Add-Edge algorithm: poor Adjacent-Score relative to randomized graph and unbalanced parcels. We hypothesized that one reason for a relatively low Adjacent-Score might be the following scenario: when a vertex is added to a component, it might have multiple edges to that component. One edge might have a very high weight; this is the one that is officially “added”. However, the other edges with far lower weights are implicitly added as well, lowering the average edge weights within the component.

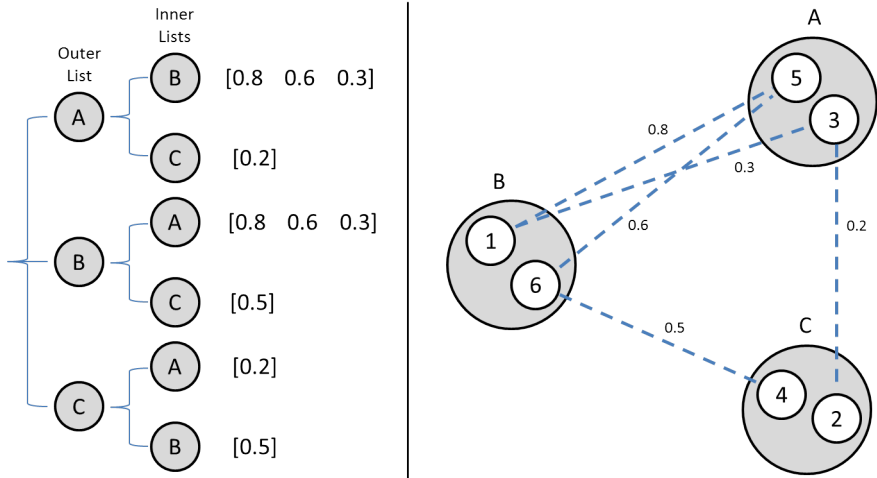
The Edge-Contraction algorithm handles this issue by maintaining that there can be at most one edge between any two components A and B , and further that the weight on such an edge is the mean of the weights on all edges that connect a vertex in A with a vertex in B .

2.3.1 Implementation using Multi-Edge Adjacency List and Priority Queue

The adjacency list is the most common implementation choice for sparse graphs and it lists, for each node in the graph, the nodes connected to it by an edge and the weight of that edge.

In a Contractible Graph, the weight of the link between two components A and B depends on all edges between vertices in A and vertices in B . Therefore our implementation of the CG data structure uses an extended variant of the adjacency list that allows for multiple edges between two nodes.

The Multi-Edge Adjacency List associates every *pair* of components connected by a non-zero link to the set of edges that comprise that link. In \mathbf{R} , this is achieved by a list of lists.



Implementing the contraction of components B and C into a new component D on this list of lists requires the following steps:

1. Compute \mathcal{X} , the set of all components that either B or C is linked to.

2. Create a new element in the outer list, D , and associate it with an empty inner list.
3. For each component $X \in \mathcal{X}$,
 - Find $w(X)$, the weights of the set of all edges connecting X to B and X to C .
 - Delete elements B and C from the inner list of X . Add a new element D and map it to $w(X)$.
 - In the inner list of D , add a new element X and map it to $w(X)$.
4. Delete B and C from the outer list.

Incidentally, we note that it is also possible to implement the average edge property of the CG without the Multi-Edge Adjacency List, by recording for each pair of components, the *sum* of edge weights and the number of edges rather than a set of all original edge weights. This allows for a more space and time efficient implementation, but this simplification was unfortunately overlooked in the first implementation of Edge-Contract.

Having described the contraction step, we will next discuss how to efficiently locate the link to be contracted. In computer science, a *Priority Queue* data type is a set of ordered (in the sense that for any two objects one has greater or equal priority to the other) objects that supports the following operations:

- *add(obj)*: Adds an object to the set.
- *remove_minimum()*: Removes and returns an object with the smallest priority in the set.

Using the heap data structure, the above two operations both run in $O(\log n)$ time.

Each component on the CG will be associated with an element of the priority queue. The priority of component A is defined as

$$|A| - \max w(A)$$

where $w(A)$ is the set of weights of links between A and any other component. Since our graph edge weights are all between 0 and 1, the smallest priority element in the queue is the first to undergo contraction in our algorithm, provided the max link weight is up-to-date.

If components A and B are contracted, and there is a component C with positive links to both A and B , then the $C - A$ and $C - B$ links will be replaced by a $C - (AB)$ link with a different weight. If either $C - A$ or $C - B$ links happened to be the maximum-weighted links of C , then C 's position on the priority queue may no longer be accurate, and its true position may be further down the queue.

To address this issue, the max weight link of the minimum priority component must be re-computed when the element is removed from the queue. If the component's actual priority is not the minimum, then it is re-inserted into the queue with

Figure 2.1: Results of Edge-Contract for Different Component Numbers

Number of Components	Adjacency	Multi-Boundary	Smoothness Ratio	Balance
500	0.7991	0.7578	54.65	0.327
400	0.7990	0.7723	59.20	0.343
300	0.7974	0.7921	65.20	0.297
250	0.7955	0.7991	69.51	0.356
200	0.7941	0.8101	75.47	0.357
150	0.7931	0.8225	83.88	0.418
116	0.7913	0.8389	92.14	0.439
100	0.7911	0.8488	97.63	0.383

updated priority. Additionally, the minimum priority component may no longer exist in the CG due to contraction with another component. In this case it is simply discarded.

Without using an efficient priority queue, the linear searching method of finding the next link to contract results in a $O(n(n-k))$ time algorithm. Using the priority queue the time complexity of Edge-Contraction is $O((n-k)(m+\log n))$, where m is the average number of positive links a component has.

2.3.2 Results and Optimal Number of Components

The Edge-Contraction algorithm has the benefit of requiring only one parameter, the target component number.

Bibliography