Problem

Let G = (V,E) be the complete, undirected, weighted graph, such that each vertex corresponds to a column of the matrix D, and the weight of an edge equals the coherence between the two columns it connects. The problem is then to define a cut through this graph that partitions G into two disjoint components, A(V_a, E_a) and B(V_b, E_b). Let X((V_x, E_x) be the subgraph of all the edges crossing the cut.

Then

$$\mu_a = \max \{ w(e) \mid e \in E_a \}$$

$$\mu_b = \max \{ w(e) \mid e \in E_b \}$$

$$\mu_m = \max \{ w(e) \mid e \in E_x \}$$

$$\mu_d = \max \{ w(e) \mid e \in E \} = \max \{\mu_a, \mu_b, \mu_m\}$$

Our goal is to maximize $\mu_m - \max\{\mu_a, \mu_b\}$

Algorithm 1:

```
• Initialize X = empty graph
• Sort all edges in decreasing order of weights in list W
• For every edge e(=pq) in W:
    o X' = X U e (That is, $V_{x'} = V_x \cup \{p, q\}$ & $E_{x'} = E_x \cup e$)
    o If X' is a bipartite graph
        ▪ X = X'
    o Else
        ▪ Break
    o Endif

• Assign vertices in $V_x$ to partition A or B in one pass, using breadth first search over each
  connected component in X
• Assign all unassigned vertices – $V \setminus V_x$ to partition A or B arbitrarily, or based on other
  consideration.
```

Proof of correctness

Proof by contradiction:

Assume that a better solution exists. That is, in an assignment $\hat{A}, \hat{B}$

$\hat{\mu}_m - \max\{\hat{\mu}_a, \hat{\mu}_b\}$

This can happen if at least one of the following is true:

$$\hat{\mu}_m > \mu_m \quad \text{..... (1)}$$

$$\text{or} \quad \max\{\hat{\mu}_a, \hat{\mu}_b\} < \max\{\mu_a, \mu_b\} \quad \text{.....(2)}$$

Eq (1) can be shown to be shown to be trivially impossible. In the very first iteration of the algorithm, the maximum weight edge is assigned to the cut. By definition, the largest edge in the cut cannot be larger than this.

Eq (2) is also impossible. Let the algorithm described terminate after k+1 iterations. That is, the k+1th largest edge could not be assigned to the cut. For eq (2) to hold, it must be possible to assign the k' largest edges to the cut, where k' > k. But since adding just the largest k+1 edges creates a cut that is impossible, any cut that is a superset of this, must also be impossible. Therefore, at most only the k largest edges can be added to the cut.

Hence, the assignment obtained must be optimal.

Time complexity:

Let the number of vertices (or columns) be n. Since it is a complete graph, the number of edges is $O(n^2)$

Sorting: $O(n^2 \log n)$

Number of iterations: $O(n^2)$

Each iteration: $O(1)$

Overall time complexity = **$O(n^2 \log n)$**

<u>Algorithm 2</u>

- Initialize A, B = empty set of vertices
- Initialize P = empty set of vertices to store the Processed vertices
- Initialize Q = empty max-heap of candidate edges
- Find the max weight edge in G, say e = uv.
- Assign A = {u}, B = {v}
- Push all edges emanating from u or v in Q
- P = P U {u,v}
- While Q is not empty
  - Pop the largest edge e' = u'v' in Q
  - If one vertex of e' is in A (say u'), and the other (v') unassigned, assign v' to B
  - If one vertex of e' is in B (say u'), and the other (v') unassigned, assign v' to A
  - Push all edges emanating from v', with the other end not in P to Q
  - Delete all edges emanating from v', with the other end in P from Q
  - P = P U v'
- EndWhile

Proof of Correctness

<To be developed> My intuition is that this algorithm is also correct, and in fact, in whatever I simulated, the two algorithms generated the same results. But I have not yet been able to find a definitive proof.

Time complexity

Finding max edge: O(n)

Number of iterations: O(n^2)

Popping min: O(1)

Pushing n edges: O(nlogn)

Deleting n edges: O(n^2)

Overall time complexity: **O(n^4)**

Implementation with list instead of heap:

Finding max edge: O(n)

Number of iterations: O(n^2)

Finding min: O(n^2)

Add n edges: O(n)

Deleting n edges: O(n)

Overall time complexity: **O(n^4)**