# Ricochet: A Family of Unconstrained Algorithms for Graph Clustering

**Abstract.** Partitional graph clustering algorithms like K-means and Star necessitate a priori decisions on the number of clusters and threshold on the weight of edges to be considered, respectively. These decisions are difficult to make and their impact on clustering performance is significant. We propose a family of algorithms for weighted graph clustering that neither requires a predefined number of clusters, unlike K-means, nor a threshold on the weight of edges, unlike Star. To do so, we use re-assignment of vertices as a halting criterion, as in K-means, and a metric for selecting clusters' seeds, as in Star. Pictorially, the algorithms' strategy resembles the rippling of stones thrown in a pond, thus the name 'Ricochet'. We evaluate the performance of our proposed algorithms using standard datasets. In particular, we evaluate the impact of removing the constraints on the number of clusters and threshold by comparing the performance of our algorithms with K-means and Star. We are also comparing the performance of our algorithms with Markov Clustering which is not parameterized by number of clusters nor threshold but has a fine tuning parameter that impacts the coarseness of the result clusters.

**Keywords:** Clustering, Weighted Graph clustering, Document clustering, K-Means Clustering, Star Clustering.

## 1 Introduction

Clustering algorithms partition a set of objects into subsets or clusters. Objects within a cluster should be similar while objects in different clusters should be dissimilar, in general. With a scalar similarity metric, the problem can be modeled as the partitioning of a weighted graph whose vertices represent the objects to be clustered and whose weighted edges represent the similarity values. For instance, in a document clustering problem (we use instances of this problem for performance evaluation) vertices are documents (vectors in a vector space model), pairs of vertices are connected (the graph is a clique) and edges are weighted with the value of the similarity of the corresponding documents (cosine similarity) [1, 2]. Partitional clustering graph algorithms, as the name indicates, partition the graph into subsets or regions trying to identify and separate dense regions from sparse regions in order to maximize intra-cluster density and inter-cluster sparseness [3].

Partitional graph clustering algorithms like K-means and Star require crucial a priori decisions on key parameters. The K-means clustering algorithm [4] requires the number of clusters to be provided before clustering. The Star clustering algorithm [5]

requires a threshold on the weight of edges to be fixed before clustering. The choice of the value of these parameters can greatly influence the effectiveness of the clustering algorithms.

In this paper, we propose a family of novel graph clustering algorithms that require neither the number of clusters nor a threshold to be determined before clustering. To do so, we combine ideas from K-means and Star. The proposed algorithms build sub graphs, assigning and dynamically reassigning vertices. We call the first vertex assigned to a sub graph the seed. The algorithms use degree of vertices and weights of adjacent edges for assignment and they use weights of adjacent edges for reassignment. During reassignment, some sub graphs disappear. The algorithms stop when there is no more reassignment. We call the intermediary and resulting sub graphs, clusters (for the sake of simplicity we may use cluster and sub graph interchangeably in the remainder of this paper).

Pictorially, the algorithms' strategy resembles the rippling (iterative assignment) caused by stones (seeds) thrown in a pond, thus the name 'Ricochet'.

Our contribution is the presentation of this family of novel graph clustering algorithms and their comparative performance analysis with state-of-the-art algorithms using real world and standard corpora for document clustering.

In the next section we discuss the main state-of-the-art weighted graph clustering algorithms, namely, K-means, Star and Markov Clustering. In section 3, we show how we devise unconstrained algorithms spinning the ricochet and rippling metaphor. In section 4, we empirically evaluate and compare the performance of our proposed algorithms. Finally, we synthesize our results and contribution in section 5.

## 2 Related Works

K-means [4], Star [5] and Markov Clustering (or MCL) [6] are typical partitional clustering algorithms. They all can solve weighted graph clustering problems. K-means and Star are constrained by the a priori setting of a parameter. Markov Clustering is not only unconstrained but also probably the state-of-the-art in graph clustering algorithms. Our proposal attempts to create an unconstrained algorithm by combining idea from K-means and Star. We review the three algorithms and their variants.

K-means clustering [4] divides the set of vertices into K clusters by choosing randomly K seeds or candidate centroids. The number of clusters, K, is provided a priori and does not change. K-means then assigns each vertex to the cluster whose centroid is the closest. K-means iteratively re-computes the position of the centroid based on the current members of each cluster. K-means converges because the average distance between vertices and their centroids monotonically decreases at each iteration. A variant of K-means efficient for document clustering is K-medoids [7]. We use K-medoids to compare with our proposed algorithms.

Unlike K-means, Star clustering [5], does not require the indication of an a priori number of clusters. It also allows the clusters produced to overlap. This is a generally desirable feature in information retrieval applications.

For document clustering, Star clustering analytically guarantees a lower bound on the topic similarity between the documents in each cluster and computes more accurate clusters than either the older single link [8] or average link [9] hierarchical clustering. The drawback of Star clustering is that the lower bound guarantee on the quality of each cluster depends on the choice of a threshold σ on the weight of edges in the graph.

To produce reliable document clusters of similarity σ, Star algorithm prunes the similarity graph of the document collection, removing edges whose cosine similarity in a vector space is less than σ. Star clustering then formalizes clustering by performing a minimum clique cover with maximal cliques on this σ-similarity graph. Since covering by cliques is an NP-complete problem [10, 11], Star clustering approximates a clique cover greedily by dense sub-graphs that are star shaped, consisting of a single Star center and its satellite vertices.

The selection of Star centers determines the Star cover of the graph and ultimately the quality of the clusters. [12] experimented with various metrics for the selection of Star centers to maximize the 'goodness' of the greedy vertex cover. The average metric (i.e. selecting Star centers in order of the average similarity between a potential Star center and the vertices connected to it) is a fast and good approximation to the expensive lower bound metric [5] that maximizes intra-cluster density in all variants of the Star algorithm. The average metric [12] is closely related to the notion of average similarity between vertices and their medoids in K-medoids [7].

Markov Clustering tries and simulates a (stochastic) flow (or random walks) in graphs [6]. From a stochastic view point, once inside a region, a random walker should have little chance to walk out [13]. The graph is first represented as stochastic (Markov) matrices where edges between vertices indicate the amount of flow between the vertices. MCL algorithm simulates flow using two alternating operations on the matrices: expansion and inflation. The flow is eventually separated into different regions, yielding a cluster interpretation of the initial graph. MCL does not require an a priori number of expected clusters nor a threshold on the similarity values. However, it requires a fine tuning inflation parameter that influences the coarseness and possibly the quality of the result clusters. We nevertheless consider it as an unconstrained algorithm, as, our experience suggests, optimal values for the parameter seem to be rather stable across applications.

Chameleon [14] is a hierarchical clustering algorithm that stems from the same motivation as the one that prompted our proposal: the need for dynamic decisions in place of a priori static parameters. However, in effect, Chameleon uses three parameters: the number of nearest neighbours, the minimum size of the sub graphs, and the relative weightage of inter-connectivity and closeness. Although the authors [14] observed that the parameters have a mild effect on 2D data, the users always need to preset the values of parameters and their effects are unknown for other types of data such as documents (high dimensional data). In a more recent paper [15], the author of Chameleon combines agglomerative hierarchical clustering and partitional clustering. Their experimental results suggest that the combined methods improve the clustering solution.

Our proposed family of algorithms, like Chameleon, is dynamic (we dynamically reassign vertices to clusters). However, our algorithms need no parameters. In spite of the absence of parameters, we hope to achieve effectiveness comparable to the best

settings of K-means, Star and MCL. We also hope to produce more efficient algorithms than K-means, Star and MCL.

## 3 A Family of Unconstrained Algorithms

Ricochet algorithms, like other partitional graph clustering algorithms, alternate two phases: the choosing of vertices to be the seeds of clusters and the assignment of vertices to existing clusters. The motivation underlying our work lies in the observation that: (1) Star clustering algorithm provides a metric of selecting Star centers that are potentially good cluster seeds for maximizing intra-cluster density, while (2) K-means provides an excellent vertices assignment and reassignment, and a convergence criterion that increases intra-cluster density at each iteration. By using Star clustering metric for selecting Star centers, we can find potential cluster seeds without having to supply the number of clusters. By using K-means re-assignment of vertices, we can update and improve the quality of these clusters and reach a termination condition without having to determine any threshold.

Hence, similar to Star and Star-ave algorithms [12], Ricochet chooses seeds in descending order of the value of a metric combining degree with the weight of adjacent edges. Similar to K-means, Ricochet assigns and reassigns vertices; and the iterative assignment of vertices is stopped once these conditions are met: (1) no vertex is left unassigned and (2) no vertex is candidate for re-assignment.

The Ricochet family is twofold. In the first Ricochet sub-family, seeds are chosen one after the other ('stones are thrown one by one'). In the second Ricochet sub-family, seeds are chosen at the same time ('stones are thrown together'). We call the former algorithms Sequential Rippling, and the latter Concurrent Rippling. The algorithms in the Sequential Rippling, because of the way they select seeds and assign or re-assign vertices, are intrinsically hard clustering algorithms, i.e. they produce disjoint clusters. The algorithms in the Concurrent Rippling are soft clustering algorithms, i.e. they produce possibly overlapping clusters.

The algorithms in the Sequential Rippling can be perceived as somewhat straightforward extensions to the K-means clustering. We nevertheless present them in this paper for the purpose of completeness and comparison with the more interesting algorithms we propose in the Concurrent Rippling.

### 3.1 Sequential Rippling

#### 3.1.1 Sequential Rippling (SR)

The first algorithm of the subfamily is call Sequential Rippling (or SR). In this algorithm, *vertices* are ordered in descending order of the average weight of their adjacent edges (later referred to as the weight of a vertex). The vertex with the highest weight is chosen to be the first seed and a cluster is formed by assigning all other vertices to the cluster of this first seed. Subsequently, new seeds are chosen one by one from the ordered list of vertices. When a new seed is added, vertices are re-

assigned to a new cluster if they are closer to the new seed than they were to the seed of their current cluster (if no vertex is closer to the new seed, no new cluster is created). If clusters are reduced to singletons during re-assignment, they are assigned to the nearest non-singleton cluster. The algorithm stops when all vertices have been considered.

The pseudocode of the Sequential Rippling algorithm is given in figure 1. The worst case complexity of Sequential Rippling algorithm is O ($N^3$) because in the worst case the algorithm has to iterate through at most N vertices, each time comparing the distance of N vertices to at most N centroids.

```
Given a Graph G = (V, E). V contains vertices, |V| = N. Each vertex has a weight which is
the average similarity between the vertex and its adjacent vertices. E contains edges in G
(self-loops removed) with similarity as weights.

Algorithm: SR ( )
Sort V in order of vertices' weights
Take the heaviest vertex v from V
listCentroid.add (v)
Reassign all other vertices to v's cluster
While (V is not empty)
    Take the next heaviest vertex v from V
    Reassign vertices which are more similar to v than to other centroid
    If there are re-assignments
        listCentroid.add (v)
        Reassign singleton clusters to its nearest centroid
For all i ε listCentroid return i and its associated cluster
```

**Fig. 1. Sequential Rippling Algorithm**

### 3.1.2 Balanced Sequential Rippling (BSR)

The second algorithm of the subfamily is called Balanced Sequential Rippling (BSR). The difference between BSR and SR is in its choice of subsequent seed. In order to balance the distribution of seeds in the graph, BSR chooses a next seed that is both a reasonable centroid for a new cluster (i.e. has large value of weight) as well as sufficiently far from the previous seeds. Subsequent seed is chosen to maximize the ratio of its weight to the sum of its similarity to the centroids of already existing clusters. This is a compromise between weight and similarity. We use here the simplest possible formula to achieve such compromise. It could clearly be refined and fine-tuned.

As in SR, when a new seed is added, vertices are re-assigned to a new cluster if they are closer to the new seed than they were to the seed of their current cluster. The algorithm terminates when there is no re-assignment of vertices.

The pseudocode of Balanced Sequential Rippling algorithm is given in figure 2. The worst case complexity of Balanced Sequential Rippling algorithm is O ($N^3$) because in the worst case the algorithm has to iterate through at most N vertices, each time comparing the distance of N vertices to at most N centroids.

```
Algorithm: BSR ( )
Sort V in order of vertices' weights
Take the heaviest vertex v from V
listCentroid.add (v)
Reassign all other vertices to v's cluster
Reassignment = true
While (Reassignment and V is not empty)
    Reassignment = false
    Take a vertex v ∉ listCentroid from V whose ratio of its weight to
     the sum of its similarity  to existing centroids is the maximum
    Reassign vertices which are more similar to v than to other centroid
    If there are re-assignments
        Reassignment = true
        listCentroid.add (v)
        Reassign singleton clusters to its nearest centroid
For all i ε listCentroid return i and its associated cluster
```

**Fig. 2. Balanced Sequential Rippling Algorithm**

### 3.2 Concurrent Rippling

Unlike sequential rippling which chooses centroids one after another, concurrent rippling treats all vertices as centroids of their own singleton clusters initially. Then, each centroid concurrently 'ripples its influence' using its adjacent edges to other vertices. As the rippling progresses, some centroids can lose their centroid status (i.e. become non-centroid) as the cluster of smaller weight centroid is 'engulfed' by the cluster of bigger weight centroid.

#### 3.2.1 Concurrent Rippling (CR)

The first algorithm of the sub-family is called Concurrent Rippling (CR). In this algorithm, for each vertex, the adjacent *edges* are ordered in descending order of weights. Iteratively, the next heaviest edge is considered. Two cases are possible: (1) if the edge connects a centroid to a non-centroid, the non-centroid is added to the cluster of the centroid (notice that at this point the non-centroid belongs to at least two clusters), (2) if the edge connects two centroids, the cluster of one centroid is assigned to the cluster of the other centroid (i.e. it is 'engulfed' by the other centroid), if and only if its weight is smaller than that of the other centroid. The two clusters are merged and the smaller weight centroid becomes a non-centroid. The algorithm terminates when the centroids no longer change. The pseudocode of Concurrent Rippling algorithm is given in figure 3.

Making sure that all centroids propagate their ripples at equal speed (lines 8 - 10 of Algorithm: CR () in figure 3), the algorithm requires the sorting of a list whose total size is the square of the number of vertices.

Concurrent Rippling algorithm requires O ($N^2$logN) complexity to sort the N-1 neighbors of the N vertices. It requires another O ($N^2$logN) to sort the $N^2$ number of edges. In the worst case, the algorithm has to iterate through all the $N^2$ edges. Hence, in the worst case the complexity of the algorithm is O ($N^2$logN).

For each vertex $v$, $v$.neighbor is the list of $v$'s adjacent vertices sorted by their similarity to $v$ from highest to lowest.
If $v$ is a centroid ($v$.centroid == 1); $v$.cluster contains the list of vertices $\neq v$ assigned to $v$

**Algorithm:** CR ( )
1. Sort E in order of the edge weights
2. public CentroidChange = true
3. index = 0
4. While (CentroidChange && index < N-1 && E is not empty)
   5. CentroidChange = false
   6. For each vertex $v$, take its edge $e_{vw}$ connecting $v$ to its next closest neighbor $w$; i.e. $w = v$.neighbor [index]
   7. Store these edges in S
   8. Find the lowest edge weight in S, say *low*, and empty S
   9. Take all edges from E whose weight >= *low*
   10. Store these edges in S
   11. **PropagateRipple** (S)
   12. index ++
13. For all $i \in V$, if $i$ is a centroid, return $i$ and $i$.cluster

**Sub Procedure:** PropagateRipple (list S)
/* This sub procedure is to propagate ripples for all the centroids. If the ripple of one centroid touches another, the heavier weight centroid will engulf the lighter centroid and its cluster. If the ripple of a centroid touches a non-centroid, the non-centroid is assigned to the centroid. A non-centroid can be assigned to more than one centroid, allowing overlapping between clusters, a generally desirable feature */

```
While (S is not empty)
    Take the next heaviest edge, say e_vw, from S
    If v ∉ x.cluster for all x ε V
            If w is a centroid, compare v's weight to w's weight
                If (w.weight > v.weight)
                    add v and v.cluster into w.cluster
                    Empty v.cluster
                    If v is a centroid
                        v.centroid = 0
                        CentroidChange = true
                Else
                    add w and w.cluster into v.cluster
                    Empty w.cluster
                    w.centroid = 0
                    CentroidChange = true
            Else if w is not a centroid
                v.cluster.add (w)
                If v is not a centroid
                    v.centroid = 1
                    CentroidChange = true
```

**Fig. 3. Concurrent Rippling Algorithm**

### 3.2.2 Ordered Concurrent Rippling (OCR)

The second algorithm of the sub-family is called Ordered Concurrent Rippling (OCR). In this algorithm, the constant speed of rippling is abandoned to be approximated by a simple ordering of adjacent edges according to their weights to the vertex (i.e. OCR abandons line 1, and lines 8 – 10 of Algorithm: CR () in figure 3).

The method allows not only to improve efficiency (although worst case complexity is the same) but also to process only the best 'ripple' (i.e. heaviest adjacent edge) for each vertex each time. The pseudocode of Ordered Concurrent Rippling algorithm is given in figure 4. The complexity of the algorithm is O ($N^2$logN) to sort the N-1 neighbors of the N vertices. The algorithm then iterates at most $N^2$ times. Hence the overall worst case complexity of the algorithm is O ($N^2$logN).

```
For each vertex v, v.neighbor is the list of v's adjacent vertices sorted by their similarity to
v from highest to lowest. If v is a centroid (i.e. v.centroid == 1); v.cluster contains the list
of vertices ≠ v assigned to v

Algorithm: OCR ( )
public CentroidChange = true
index = 0
While (CentroidChange && index < N-1)
    CentroidChange = false
    For each vertex v, take its edge e_vw connecting v to its next closest
    neighbor w; i.e. w = v.neighbor [index]
    Store these edges in S
    PropagateRipple (S)
    index ++
For all i ε V, if i is a centroid, return i and i.cluster
```

**Fig. 4. Ordered Concurrent Rippling Algorithm**

### 3.3 Maximizing Intra-cluster Similarity

The key point of Ricochet is that at each step it tries to maximize the intra-cluster similarity: the average similarity between vertices and the centroid in the cluster.

Sequential Rippling (SR and BSR) try to maximize the average similarity between vertices and their centroids by (1) selecting centroids in order of their weights and (2) iteratively reassigning vertices to the nearest centroids. As in [12], selecting centroids in order of weights is a fast approximation to maximizing the expected intra-cluster similarity. As in K-means, iteratively reassigning vertices to nearest centroids decreases the distance (thus maximizing similarity) between vertices and their centroids.

Concurrent Rippling (CR and OCR) try to maximize the average similarity between vertices and their centroids by (1) processing adjacent edges for each vertex in order of their weights from highest to lowest and (2) choosing the bigger weight vertex as a centroid whenever two centroids are adjacent to one another. (1) ensures that at each step, the best possible merger for each vertex $v$ is found (i.e. after merging a vertex to $v$ with similarity $s$, we can be sure that we have already found and merged all vertices whose similarity is better than $s$ to $v$); while (2) ensures that the centroid of a cluster is always the point with the biggest weight. Since we define a vertex weight as the average similarity between the vertex and its adjacent vertices; choosing a centroid with bigger weight is an approximation to maximizing the average similarity between the centroid and its vertices.

Further, although OCR is, like its family, a partitional graph clustering algorithm; its decision to only process the heaviest adjacent edge (hence the best possible merger) of each vertex at each step is compatible to the steps of agglomerative single-link hierarchical clustering. We believe therefore that OCR combines concepts from both partitional and agglomerative approaches. As in [15], such combination has been experimentally found to be successful than either of the methods alone.

## 4 Performance Analysis

In order to evaluate our proposed algorithms, we empirically compare, the performance our algorithms with the constrained algorithms: (1) K-medoids (that is given the optimum/correct number of clusters as obtained from the data set); (2) Star clustering algorithm, and (3) the improved version of Star (i.e. Star Ave) that uses average metric to pick star centers [12]. This is to investigate the impact of removing the constraints on the number of clusters (K-Medoids) and threshold (Star) on the result of clustering. We then compare, the performance of our algorithms with the state-of-the-art unconstrained algorithm, (4) Markov Clustering (MCL), varying MCL's fine-tuning inflation parameter.

We use data from Reuters-21578 [16], TIPSTER–AP [17] and a collection of web documents constructed using the Google News search engine [18] and referred to as Google.

The Reuters-21578 collection contains 21,578 documents that appeared in Reuter's newswire in 1987. The documents are partitioned into 22 sub-collections. For each sub-collection, we cluster only documents that have at least one explicit topic (i.e. documents that have some topic categories within its <TOPICS> tags). The TIPSTER–AP collection contains AP newswire from the TIPSTER collection. For the purpose of our experiments, we have partitioned TIPSTER-AP into 2 separate sub-collections. Our original collection: Google contains news documents obtained from Google News in December 2006. This collection is partitioned into 2 separate sub-collections. The documents have been labeled manually. In total we have 26 sub-collections. The sub-collections, their number of documents and topics/clusters are reported in Table 1.

By default and unless otherwise specified, we set the value of threshold $\sigma$ for Star clustering algorithm to be the average similarity of documents in the given collection.

In each experiment, we apply the clustering algorithms to a sub-collection. We study effectiveness (recall, r, precision, p, and F1 measure, F1 = (2 * p * r) / (p + r)), and efficiency in terms of running time.

In each experiment, for each topic, we return the cluster which best approximates the topic. Each topic is mapped to the cluster that produces the maximum F1-measure with respect to the topic:

$$\text{topic}\ (i) = \max_j \{\text{F1}\ (i, j)\}\ . \qquad (\mathbf{1})$$

where F1 $(i, j)$ is the F1 measure of the cluster number $j$ with respect to the topic number $i$. The weighted average of F1 measure for a sub-collection is calculated as follows:

$$F1 = \Sigma (t_i/S) * F1 (i, \text{topic } (i)); \text{ for } 0 \leq i \leq T . \quad (2)$$

$$S = \Sigma t_i; \text{ for } 0 \leq i \leq T \quad (3)$$

where T is the number of topics in the sub-collection; $t_i$ is the number of documents belonging to topic *i* in the sub-collection.

For each sub-collection, we calculate the weighted-average of precision, recall and F1-measure produced by each algorithm.

For each collection, we then present the average (using micro-averaging) of results produced by each algorithm.

Whenever we perform efficiency comparison with other graph-based clustering algorithms (Markov), we do not include the pre-processing time to construct the graph and compute all pair wise cosine-similarities. The pre-processing time of other graph-based clustering algorithms is the same as our proposed algorithm that is also graph-based. Furthermore, the complexity of pre-processing, O ($n^2$), may undermine the actual running time of the algorithms themselves. We however include this pre-processing time when comparing with non graph-based clustering algorithms (K-medoids) that do not require graph or all pair wise similarities to be computed. This is to illustrate the effect of pre-processing on the efficiency of graph-based clustering algorithms.

**Table 1.** Description of Collections

| Sub-collection | # of docs | # of topic | Sub-collection | # of docs | # of topic |
|---|---|---|---|---|---|
| reut2-000.sgm | 981 | 48 | Reut2-001.sgm | 990 | 41 |
| reut2-002.sgm | 991 | 38 | Reut2-003.sgm | 995 | 46 |
| reut2-004.sgm | 990 | 42 | Reut2-005.sgm | 997 | 50 |
| reut2-006.sgm | 990 | 38 | Reut2-007.sgm | 988 | 44 |
| reut2-008.sgm | 991 | 42 | Reut2-009.sgm | 495 | 24 |
| reut2-010.sgm | 989 | 39 | Reut2-011.sgm | 987 | 42 |
| reut2-012.sgm | 987 | 50 | Reut2-013.sgm | 658 | 35 |
| reut2-014.sgm | 693 | 34 | Reut2-015.sgm | 992 | 45 |
| reut2-016.sgm | 488 | 34 | Reut2-017.sgm | 994 | 61 |
| reut2-018.sgm | 994 | 50 | Reut2-019.sgm | 398 | 24 |
| reut2-020.sgm | 988 | 28 | Reut2-021.sgm | 573 | 24 |
| Tipster-AP1 | 1787 | 47 | Tipster-AP2 | 1721 | 48 |
| Google1 | 1019 | 15 | Google2 | 1010 | 14 |

## 4.1 Performance Results

### 4.1.1 Comparison with Constrained Algorithms

We first compare the effectiveness and efficiency of our proposed algorithms with those of K-medoids, Star, and (an improved version of Star) Star-ave, in order to determine the consequences of combining ideas from both algorithms to obtain an unconstrained family. There is, of course, a significant benefit per se in removing the

need for parameter setting. Yet the subsequent experiments show that this can be done, only in the some cases, at a minor cost in effectiveness.

In figure 5, we can see that the effect of combining ideas from both K-means and Star in our algorithms improves precision on Google data. Our algorithms also maintain recall therefore improving the F1-value. In particular, CR is better than K-medoids, Star and Star-ave in terms of F1-value. BSR and OCR are better than K-medoids and Star in terms of F1-value. In terms of efficiency (cf. figure 6), CR and OCR are faster than Star and Star-ave. K-medoids is much faster than all the graph-based clustering algorithms because it does not require computation of pair wise similarities between all the documents.

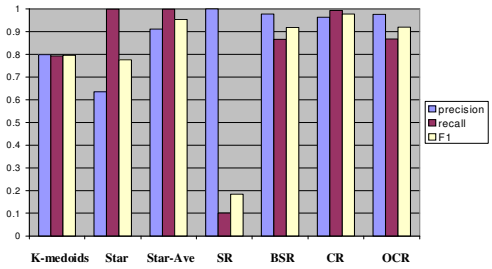


**Fig. 5. Effectiveness on Google**

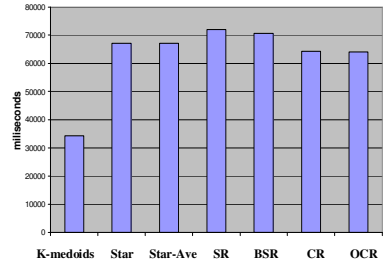


**Fig. 6. Efficiency on Google**

On Tipster-AP data (cf. figure 7), BSR and OCR yield a higher precision and F1-value than Star and Star-Ave. On Tipster-AP data, OCR performs the best among our proposed algorithms and its effectiveness is comparable to that of a K-medoids supplied with the correct number of clusters K. CR and OCR are also faster than Star and Star-ave (cf. figure 8).

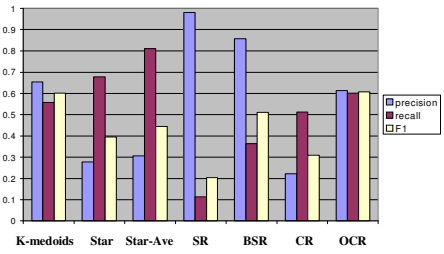


**Fig. 7. Effectiveness on Tipster-AP**

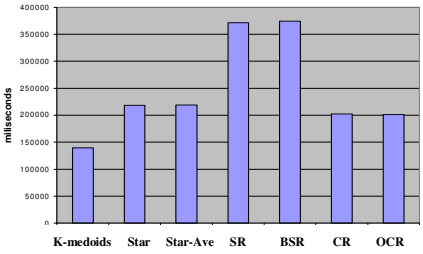


**Fig. 8. Efficiency on Tipster-AP**

On Reuters data (cf. figure 9), in terms of F1-value, OCR performs the best among our algorithms. OCR performance is better than K-medoids and is comparable to Star and Star-Ave. In terms of efficiency (cf. figure 10), OCR is faster than Star and Star-Ave but slower than K-medoids which does not require pair wise similarity computation between all the documents.

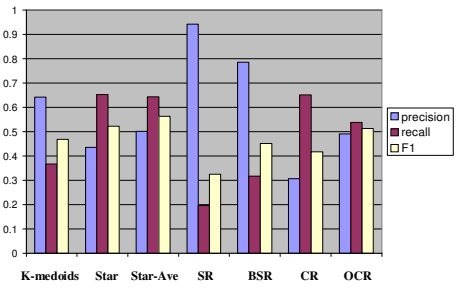


**Fig. 9. Effectiveness on Reuters**

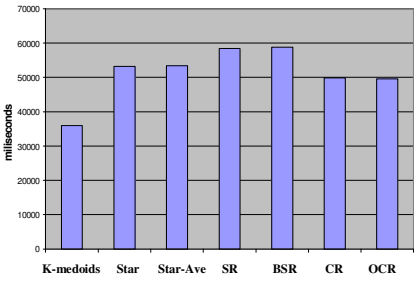


**Fig. 10. Efficiency on Reuters**

In summary, BSR and OCR are the most effective among our proposed algorithms. BSR achieves higher precision than K-medoids, Star and Star-Ave on all data sets. OCR achieves a balance between high precision and recall, and obtains higher or comparable F1-value than K-medoids, Star and Star-Ave on all data sets. Since our algorithms try to maximize intra-cluster similarity, it is precision that is mostly improved.

In particular, OCR is the most effective and efficient of our proposed algorithms. In terms of F1, OCR is 8.7% better than K-medoids, 23.5% better than Star, and 7.9% better than Star-Ave. In terms of efficiency, OCR is 56.5% slower than K-medoids (due to pre-processing time), 6.3% faster than Star and 6.5% faster than Star-Ave. When pre-processing time is not factored in (cf. Figure 11 to 13), the graph-based algorithms: Star, Star-Ave, CR, OCR are all faster than K-medoids. This shows that it is mostly pre-processing that has adverse effect on the efficiency of graph-based clustering algorithms.

In the next section, we compare our best performing algorithms: BSR and OCR, with the unconstrained algorithm: Markov Clustering.

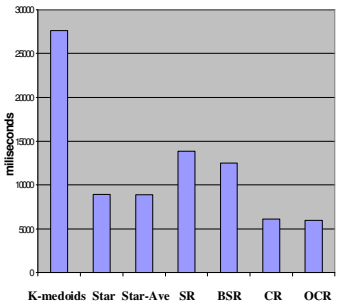


**Fig. 11. Efficiency on Google**

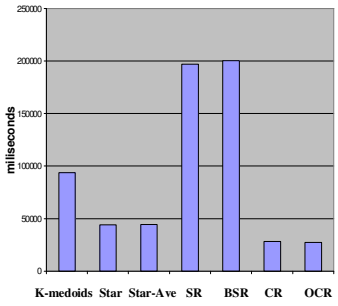


**Fig. 12. Efficiency on Tipster-AP**

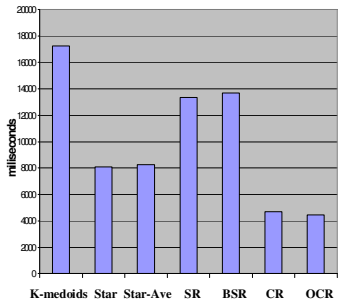


**Fig. 13. Efficiency on Reuters**

### 4.1.2 Comparison with Unconstrained Algorithms

We first illustrate the influence of MCL's inflation parameter on the algorithm's performance. We vary it between 0.1 and 30.0 (we have empirically verified that this range is representative of MCL performance on our data sets) and report results for representative values.

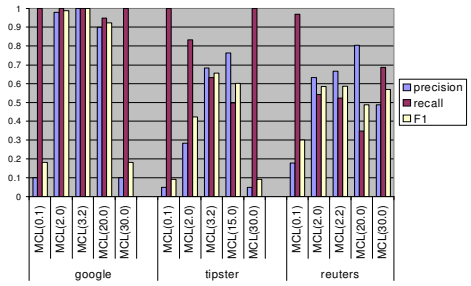


**Fig. 14. Effectiveness of MCL**

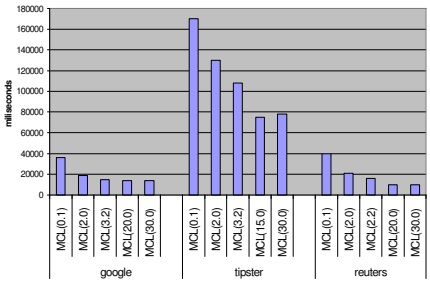


**Fig. 15. Efficiency of MCL**

As shown in figure 14, at a value of 0.1, the resulting clusters have high recall and low precision. As the inflation parameter increases, the recall drops and precision improves, resulting in higher F1-value. At the other end of the spectrum, at a value of 30.0, the resulting clusters are back to having high recall and low precision again. In terms of efficiency (cf. figure 15), as the inflation parameter increases, the running time decreases, indicating that MCL is more efficient at higher inflation value. From figure 14 and 15, we have shown empirically that the choice of inflation value indeed affects the effectiveness and efficiency of MCL algorithm. Both MCL's effectiveness and efficiency vary significantly at different inflation values. The optimal value seems however to always be around 3.0.

We now compare the performance of our best performing algorithms, BSR and OCR, to the performance of MCL algorithm at its best inflation value as well as at its minimum and maximum inflation values, for each collection.

On Google data (cf. figure 16), the effectiveness of BSR and OCR is competitive (although not equal) to that of MCL at its best inflation value. Yet, BSR and OCR are much more effective than MCL at the minimum and maximum inflation values. In terms of efficiency, both BSR and OCR are significantly faster than MCL at all inflation values (cf. figure 17).

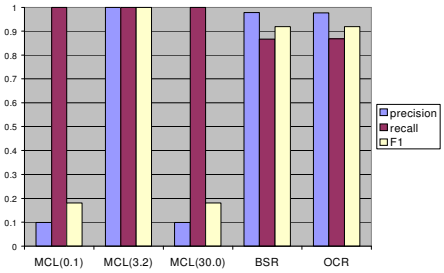


**Fig. 16. Effectiveness on Google**

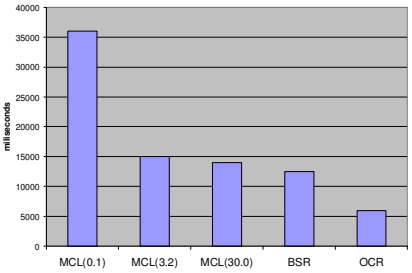


**Fig. 17. Efficiency on Google**

On Tipster-AP data (cf. figure 18), BSR and OCR are slightly less effective than MCL at its best inflation value. However, both BSR and OCR are much more effective than MCL at the minimum and maximum inflation values. In terms of efficiency (cf. figure 19), OCR is also much faster than MCL at all inflation values.

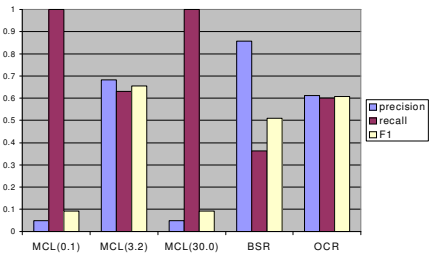


**Fig. 18. Effectiveness on Tipster-AP**

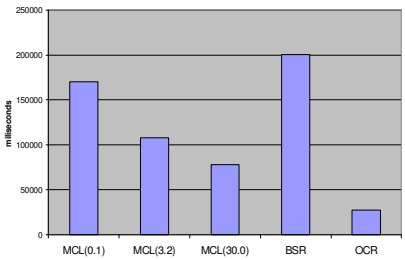


**Fig. 19. Efficiency on Tipster-AP**

The same trend is noticeable on Reuters data (cf. figure 20). BSR and OCR are slightly less effective than MCL at its best inflation value. However, BSR and OCR are more effective than MCL at the minimum and maximum inflation values. In terms of efficiency (cf. figure 21), OCR is much faster than MCL at all inflation values.

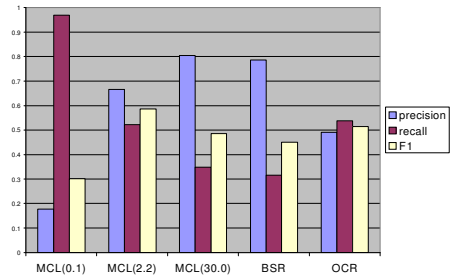


**Fig. 20. Effectiveness on Reuters**

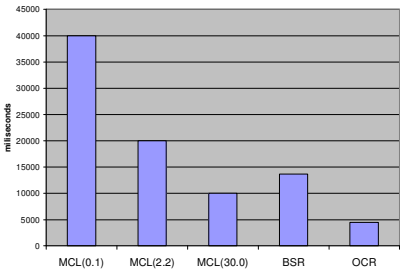


**Fig. 21. Efficiency on Reuters**

In summary, although MCL can be slightly more effective than our proposed algorithms at its best settings of inflation parameter (9% more effective than OCR), one of our algorithms, OCR, is not only respectably effective but is also significantly more efficient (70.8% more efficient). Furthermore OCR does not require the setting of any fine tuning parameters like the inflation parameter of MCL that, when set incorrectly, can have adverse effect on its performance (OCR is in average, 334% more effective than MCL at its worst parameter setup).

## 5 Conclusion

We have proposed a family of algorithms for the clustering of weighted graphs. Unlike state-of-the-art K-means and Star clustering algorithms, our algorithms do not require the a priori setting of extrinsic parameters. Unlike state-of-the-art MCL clustering algorithm, our algorithms do not require the a priori setting of intrinsic fine tuning parameters. We call our algorithms 'unconstrained'.

Our algorithms have been devised by spinning the metaphor of ripples created by the throwing of stones in a pond. Clusters' centroids are stones; and rippling is the iterative spread of the centroids' influence and the assignment and reassignment of vertices to the centroids' clusters. For the sake of completeness, we have proposed both sequential (in which centroids are chosen one by one) and concurrent (in which every vertex is initially a centroid) versions of the algorithms and variants.

After a comprehensive comparative performance analysis with reference data sets in the domain of document clustering, we conclude that, while all our algorithms are competitive, one of them, Ordered Concurrent Rippling (OCR), yields a very respectable effectiveness while being efficient. Since all our algorithms try to maximize intra-cluster similarity at each step, it is mostly precision that is improved.

However, like other graph clustering algorithms, the pre-processing time to build the graph and compute all pair-wise similarities in the graph remains a bottleneck

when compared to non graph-based clustering algorithms like K-means. We are exploring other ways to reduce this pre-processing time using indexing, stream processing or randomized methods.

We have therefore proposed a novel family of algorithms, called Ricochet algorithms, and, in particular, one new effective and efficient algorithm for weighted graph clustering, called Ordered Concurrent Rippling or OCR.

## References


1. Salton, G.: Automatic Text Processing: the transformation, analysis, and retrieval of information by computer. Addison-Wesley (1989)
2. Salton, G.: The Smart document retrieval project. In Proceedings of the Fourteenth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval (1991) 356-358
3. Brandes U., Gaertler M., Wagner D.: Experiments on Graph Clustering Algorithms. Lecture Notes in Computer Science. Di Battista and U. Zwick (Eds.) (2003) 568-579
4. MacQueen, J. B.: Some Methods for classification and Analysis of Multivariate Observations. Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability. Berkeley. University of California Press (1967) 1:281-297
5. Aslam, J., Pelekhov, K., Rus, D.: The Star Clustering Algorithm. In Journal of Graph Algorithms and Applications (2004) 8(1) 95-129
6. van Dongen, S. M.: Graph clustering by flow simulation - Tekst. - Proefschrift Universiteit Utrecht (2000)
7. Kaufman L., Rousseeuw P.: Finding groups in data: an introduction to cluster analysis. John Wiley and Sons, New York (1990)
8. Croft, W. B.: Clustering large files of documents using the single-link method. Journal of the American Society for Information Science (1977) 189-195
9. Voorhees, E.: The cluster hypothesis revisited. In Proceedings of the 8th SIGIR 95-104
10. Lund, C., Yannakakis, M.: On the hardness of approximating minimization problems. Journal of the ACM 41 (1994) 960-981
11. Press W., Flannery B., Teukolsky S., Vetterling W.: Numerical Recipes in C: The Art of Scientific Computing. Cambridge University Press (1988)
12. Wijaya D., Bressan S.: Journey to the Centre of the Star: Various Ways of Finding Star Centers in Star Clustering. 18th International Conference on Database and Expert Systems Applications DEXA (2007)
13. Nieland H: Fast Graph Clustering Algorithm by Flow Simulation. Research and Development ERCIM News No. 42 (2000)
14. Karypis G., Han E., Kumar V.: CHAMELEON: A Hierarchical Clustering Algorithm Using Dynamic Modeling. IEEE Computer Vol. 32 No. 8 (1999) 68-75
15. Zhao Y., Karypis G.: Hierarchical Clustering Algorithms for Document Datasets. Data Mining and Knowledge Discovery Vol. 10, No. 2, (2005) 141-168,
16. http://www.daviddlewis.com/resources/testcollections/reuters21578/ (visited on December 2006)
17. http://trec.nist.gov/data.html (visited on December 2006)
18. Google News (http://news.google.com.sg)