THE NATIONAL UNIVERSITY
of SINGAPORE

School of Computing
Lower Kent Ridge Road, Singapore 119260

TRB7/07

Ricochet: A Family of Unconstrained Algorithms for Graph clustering

Derry Tanti WIJAYA and Stephane BRESSAN

July 2007

Technical Report

Foreword

This technical report contains a research paper, development or tutorial article, which has been submitted for publication in a journal or for consideration by the commissioning organization. The report represents the ideas of its author, and should not be taken as the official views of the School or the University. Any discussion of the content of the report should be sent to the author, at the address shown on the cover.

JAFFAR, Joxan
Dean of School

Ricochet: A Family of Unconstrained Algorithms for Graph Clustering

Derry Tanti Wijaya
National University of Singapore
School of Computing
3 Science Drive 2, Singapore 117543
+65 6516 2727
derrytan@comp.nus.edu.sg

Stéphane Bressan
National University of Singapore
School of Computing
3 Science Drive 2, Singapore 117543
+65 6516 2727
steph@nus.edu.sg

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 can be 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.

Categories and Subject Descriptors
H.3.3 [**Information Search and Retrieval**]: Clustering

General Terms
Algorithms, Measurement, Performance, Experimentation.

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 algorithms need to select seeds (or candidate surrogate centroids for the cluster); we use local metrics (combining degree of vertices and weight of adjacent edges) for selecting clusters' seeds. The algorithms iteratively assign adjacent vertices to a cluster; we use re-assignment of vertices as a halting criterion (no vertex needs to be re-assigned). We call these algorithms Ricochet algorithms.

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 WORK
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. Our proposal attempts to create an unconstrained algorithm by combining idea from both K-means and Star.

Markov Clustering is not only unconstrained but also probably the state-of-the-art in graph clustering algorithms. We review the three algorithms and their variants.

K-means clustering algorithm [4] divides the set of vertices of a graph into K clusters by first choosing randomly K seeds or candidate centroids. It then assigns each vertex to the cluster whose centroid is the closest. K-means iteratively re-computes the position of the exact centroid based on the current members of each cluster, and reassigns vertices to the cluster with the closest centroid until a halting criterion is met (centroid no longer move). The number of clusters, K, is provided a priori and does not change.

K-means clustering attempts to maximize intra-cluster density by minimizing the average distance between vertices and their centroids or, equivalently, by maximizing the average similarity between vertices and their centroids. K-means converges because the average distance between vertices and their centroids monotonically decreases at each iteration. First, the average distance between vertices and their centroids decreases in the re-assignment step since each vertex is assigned to the closest centroid. Second, this value decreases in the recomputation step because the new centroid is the vertex for which this average distance between vertices and the centroid reaches its minimum.

A variant of K-means efficient for document clustering is K-medoids [7]. It uses medoids, i.e. document vectors in the cluster that are closest to the centroid, instead of centroids. Since medoids are sparse document vectors, distance computations are fast. In this paper, we use K-medoids to compare with the performance of our proposed algorithms.

Unlike K-means, Star clustering, first proposed by Aslam et al. in 1998 [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 methods.

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 (i.e. clusters where documents have pair-wise similarities of at least σ), the Star algorithm starts by pruning the similarity graph of the document collection, removing edges between documents whose cosine similarity in a vector space is lesser than σ. Star clustering then formalizes clustering by performing a minimum clique cover with maximal cliques on this σ-similarity graph where the cover is a vertex cover.

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 m satellite vertices, where there exist edges between the Star center and each satellite vertex. Star clustering guarantees pair-wise similarity of at least σ between the Star and each of the satellite vertices. Aslam et al. also derives a lower bound on the similarity between satellite vertices and predicts that the pair-wise similarity between satellite vertices in a Star-shaped sub-graph is high. Together with empirical evidence, Aslam et al. conclude that covering σ-similarity graph with Star-shaped sub-graphs is an accurate method for clustering.

The steps of Star clustering algorithm are as follows. The algorithm starts by sorting vertices in the σ-similarity graph by degree. Then it scans the sorted vertices from highest to lowest degree as a greedy search for Star centers. Only vertices that do not yet belong to a Star can become Star centers. Once a new Star center v is selected, its center and marked bit are set, and for all vertices w adjacent to v, w's marked bit is set. Only one scan of the sorted vertices is needed to determine all Star centers. Upon termination, i.e. when all vertices have their marked bits set, these conditions must be met: (1) the set of Star centers are the Star cover of the graph, (2) a Star center is not adjacent to any other Star center, and (3) every satellite vertex is adjacent to at least one center vertex of equal or higher degree.

Since the selection of Star centers determines the Star cover of the graph and ultimately the quality of the clusters, we experimented with various metrics for the selection of Star centers to maximize the 'goodness' of this greedy vertex cover. The result of our experiments [12] suggests that a selection of Star centers based on degree (as proposed by the original algorithm inventors) performs almost as poorly as a random selection. On the other hand, 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 (derived in [5]) that maximizes intra-cluster density in all variants of the Star algorithm. Notice that this average metric is closely related to the notion of average similarity between vertices and their medoids in K-medoids [6].

Markov Clustering tries and simulates a (stochastic) flow (or random walks) in graphs [6]. The aim of MCL is to separate the graph into regions with many edges inside and with only a few edges between regions by maximizing the flow inside the regions and minimizing the flow across regions. From a stochastic view point, once inside a region, a random walker should have little chance to walk out [13]. To do this, the graph is first represented as stochastic (Markov) matrices where edges between vertices indicate the amount of flow between the vertices: i.e. similarity measures or the chance of walking from one vertex to another in the graph. MCL algorithm simulates flow using two alternating simple algebraic operations on the matrices: expansion, which coincides with normal matrix multiplication, and inflation, which is a Hadamard power followed by a diagonal scaling. The expansion process causes flow to spread out and the inflation process models the contraction of flow: it becoming thicker in regions of higher current and thinner in regions of lower current. The flow is eventually separated into different regions, yielding a cluster interpretation of the initial graph. MCL does require neither 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.

The motivation underlying our work lies in the observation that Star clustering algorithm provides a metric of selecting Star

centers that are potentially good cluster seeds for maximizing the resulting intra-cluster density while K-means provides an excellent 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. We hope to achieve effectiveness comparable to the best settings of K-means, Star and MCL in spite of the absence of parameters. We also hope to produce more efficient algorithms than only K-means, Star and MCL.

3. A FAMILY OF UNCONSTRAINED ALGORITHMS

Ricochet algorithms, like most partitional graph clustering algorithms, alternate two phases: the choice vertices to be seeds of clusters and the assignment of vertices to existing clusters. Similarly to Star and Star-ave algorithms, Ricochet algorithms choose seeds in descending order of the value of a metric combining their degree with the weight of adjacent edges. We use the average weight of adjacent edges. Similarly to K-means the iterative assignment is stopped once no vertex is left unassigned and no vertex is candidate for re-assignment.

The family is twofold. In the first sub-family, seeds are chosen one after the other. Stones are thrown one by one. In the second 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 sub-family, 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 sub-family, because of are soft clustering algorithms, i.e. they produce possibly overlapping clusters.

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 first sorted 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 value is chosen to be the first seed. One cluster is formed that contains all other vertices.

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 clustered are reduced to singletons during re-assignment, they are deleted. If no vertex is closer to the seed, no new cluster is created.

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
```

Figure 1. Sequential Rippling Algorithm

3.1.2 Balanced Sequential Rippling (BSR)

The Second algorithm of the subfamily is called Balanced Sequential Rippling (BSR). In order to balance the distribution of seeds in the graph, we choose a next seed that is both a reasonable centroid for a new cluster (large value for the metric) as well as sufficiently far from the previous seeds.

As in the previous algorithm, the vertex with the highest weight is chosen to be the first seed. One cluster is formed that contains all other vertices.

Subsequently, a next seed is chosen that maximizes the ratio of its weight to the sum of its similarity to the centroids of 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 the previous algorithm, 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 clustered are reduced to singletons during re-assignment, they are deleted. If no vertex is closer to the seed, no new cluster is created. The algorithm terminates when there is no re-assignment of vertices.

The pseudocode of Balanced Sequential Rippling algorithm is given in figure 2.

```
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
```

Figure 2. Balanced Sequential Rippling Algorithm

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.

3.2 Concurrent Rippling

3.2.1 Concurrent Rippling (CR)

The first algorithm of the sub-family is called Concurrent Rippling (CR).

In this algorithm, each vertex is initially marked to be a seed. Pairs of vertices and seeds are ordered in ascending order of their similarity.

Iteratively, the next pair is considered. If the vertex is not a seed itself, it is assigned to the cluster of the corresponding seed (notice that at this point is belongs to at least two clusters). If the vertex is a seed itself it is assigned to the cluster of the corresponding seed, if and only if its weight is smaller than the one of the seed, and the two clusters are merged.

Making sure that the ripple propagates at equal speed for all seeds (with possible and occasional merging of clusters) requires the sorting of a list whose size is the square of the number of vertices. The algorithm terminates when the centroids no longer change.

The pseudocode of Concurrent Rippling algorithm is given in figure 3.

Concurrent Rippling algorithm requires O (N^2logN) complexity to sort the N-1 neighbors of the N vertices. It requires another O (N^2logN) 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^2logN).

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 vertices according to their distance to the candidate seed. The method allows not only to improve efficiency (although the worst case complexity is the same) but also to favor heavy seeds.

The key point of this algorithm is that at each time step it tries to maximize the average similarity between vertices and their centroids. The algorithm does this by processing adjacent vertices for each vertex in order of their similarity (from highest to lowest). This ensures that at each time step, the best possible merger for each vertex *v* is found. This means that 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*. At each time step therefore, the algorithm assigns vertices to their best possible cluster.

By choosing higher weight vertex as a centroid whenever two centroids are adjacent to one another, the algorithm ensures that the centroid of a cluster is always the point with the highest weight. Since we define a vertex weight as the average similarity between the vertex and its adjacent vertices; choosing a centroid with higher weight is an approximation to maximizing the average similarity between the centroid and its vertices.

The pseudocode of Ordered Concurrent Rippling algorithm is given in figure 4.

The Ordered Concurrent Rippling algorithm terminates when the centroids no longer change. The complexity of the algorithm is O (N^2logN) 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^2logN).

```
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: CR ( )
Sort E in order of the edge weights
public CentroidChange = true
index = 0
While (CentroidChange && index < N-1 && E is not empty)
    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
    Find the lowest edge weight in S, say low, and empty S
    Take all edges from E whose weight >= low
    Store these edges in S
    PropagateRipple (S)
    index ++
For all i ε 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
absorb 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 in IR*/

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
```

Figure 3. Concurrent Rippling Algorithm

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 $\neq v$ 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 $i \varepsilon$ V, if i is a centroid, return i and i.cluster

Figure 4. Ordered Concurrent Rippling Algorithm

4. PERFORMANCE ANALYSIS

4.1 Experimental Setup

In order to evaluate our proposed algorithms, we empirically compare, in subsection 4.2.1, 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) our 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, in subsection 4.2.2, 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 [14], TIPSTER–AP [15] and a collection of web documents constructed using the Google News search engine [16] 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, each of the first 21 sub-collections contain 1000 documents while the last sub-collection contains 578 documents. 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 during 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 σ to be the average similarity of documents in the given collection. We apply the clustering algorithms to each sub-collection. We present the results averaged for each collection.

We study effectiveness (recall, r, precision, p, and F1 measure [17], 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:

$$\text{topic}\ (i) = \max_j \{F1\ (i, j)\}$$

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 each sub-collection is calculated as follows:

$$F1 = \Sigma\ (n_i/S) * F1\ (i, \text{topic}\ (i)); \text{ for } 0 \le i \le N$$

$$S = \Sigma\ n_i; \text{ for } 0 \le i \le N$$

where N is the number of topics in the sub-collection; n_i is the number of documents belonging to topic i in the given collection. For each sub-collection, we calculate the weighted-average of precision, recall and F1-measure produced by each algorithm. We then present the average results produced by each algorithm over each collection.

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

4.2 Performance Results and their Discussion

4.2.1 Apples and Oranges: Comparison with Constrained Algorithms

We first compare the effectiveness and efficiency of our proposed algorithms with the ones of a variant of K-medoids, Star, and our improved version of Star, Star-ave, in order to determine the consequences of combining ideas from both algorithms to obtained an unconstrained family not requiring parameters. 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 with a significant gain in efficiency and, 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 proposed algorithms improves precision for Google data. With the exception of SR, our proposed algorithms also improve the F1-value and maintain recall. 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; and are comparable to

Star-ave. In terms of efficiency (cf. figure 6), SR and BSR is comparable to K-medoids. CR and OCR are much faster than K-medoids, Star and Star-ave.

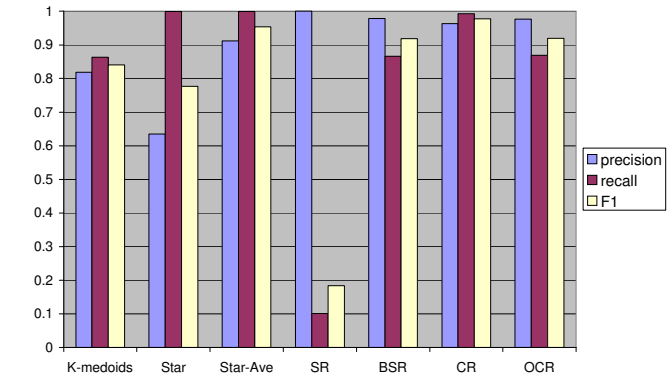

Figure 5. Effectiveness on Google Data

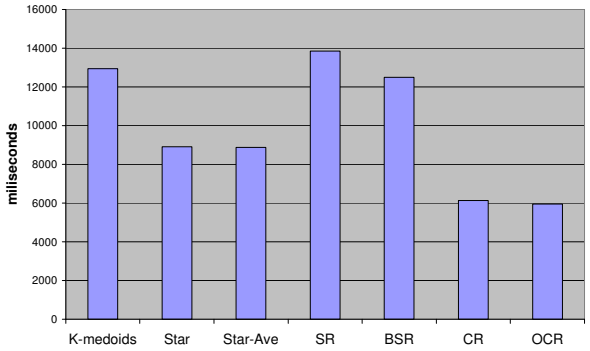

Figure 6. Efficiency on Google Data

On Tipster-AP data (cf. figure 7), SR, BSR and OCR again yield higher precision than Star and Star-Ave. BSR and OCR yield a higher F1-value than Star and Star-Ave. 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. OCR is also the fastest. It is significantly much faster than K-medoids (cf. figure 8).

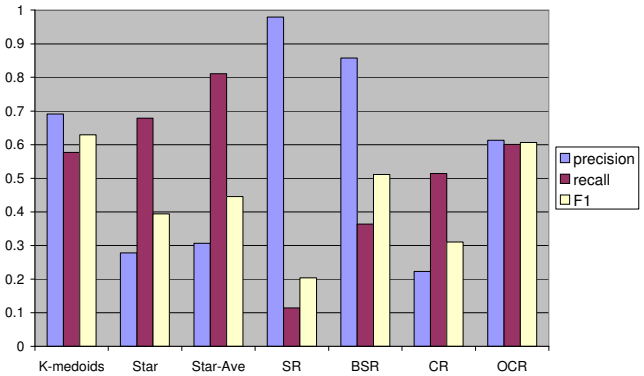

Figure 7. Effectiveness on Tipster-AP Data

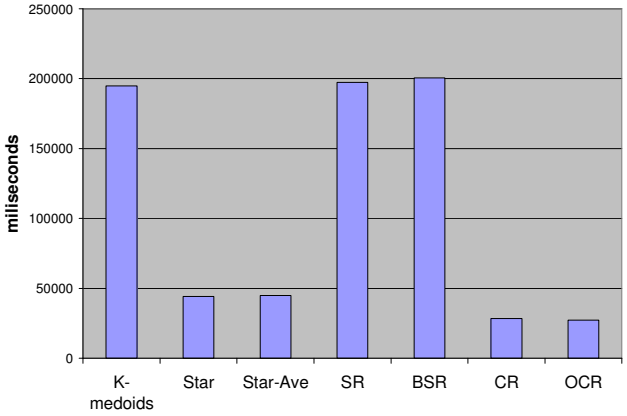

Figure 8. Efficiency on Tipster-AP Data

On Reuters data (cf. figure 9), SR and BSR again yield a higher precision than K-medoids, Star and Star-Ave. In terms of F1-value, it is OCR that performs the best among our proposed algorithms. OCR performance is better than K-medoids and is comparable to that of Star and Star-Ave. In terms of efficiency (cf. figure 10), OCR is again much faster than K-medoids, Star and Star-Ave.

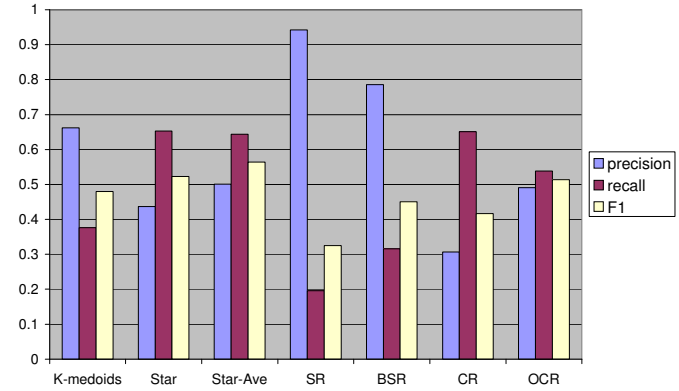

Figure 9. Effectiveness on Reuters Data

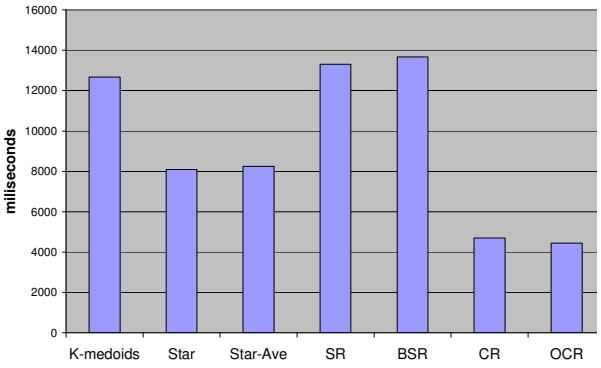

Figure 10. Efficiency on Reuters Data

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 three data sets. OCR achieves a

balance between high precision and recall, and obtains comparable or higher F1-value than K-medoids, Star and Star-Ave on the data sets. The sequential algorithm, SR and BSR are not efficient. The concurrent algorithms CR and OCR are significantly more efficient than all other algorithms.

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

4.2.2 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.

As shown in figure 11, 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 12), as the inflation parameter increases, the running time decreases, indicating that MCL is more efficient at higher inflation value. From figure 11 and 12, we have shown empirically that the choice of inflation value indeed affects the effectiveness and efficiency of MCL algorithm.

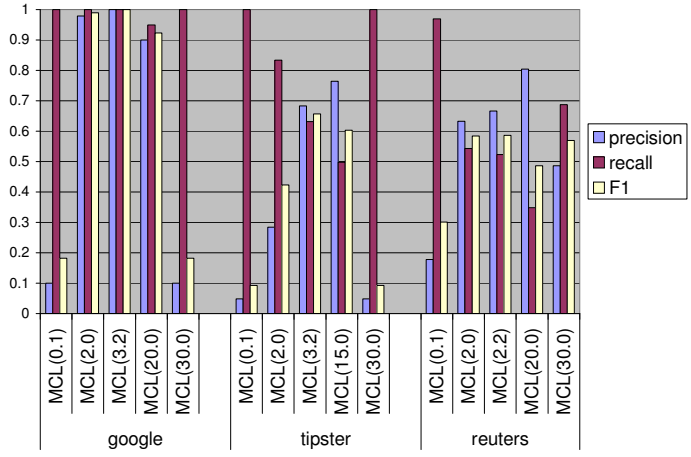

Figure 11. Effectiveness of MCL at Different Parameters

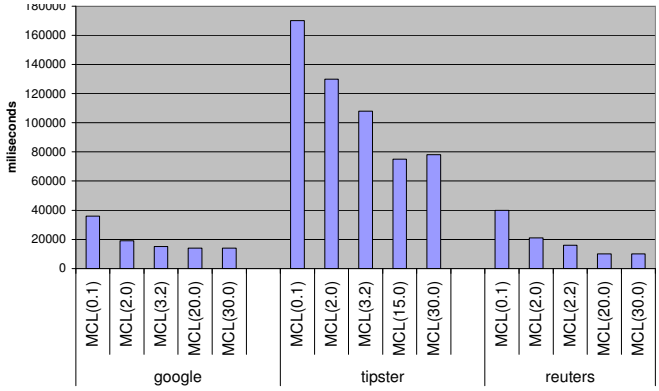

Figure 12. Efficiency of MCL at Different Parameters

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.

From figure 13, with Google data, we can see that the effectiveness of BSR and OCR is competitive but not equal to the one of MCL at its best inflation value. Yet they are more effective than MCL at the minimum and maximum inflation values. We also see in figure 14 that both BSR and OCR are significantly faster than MCL at all inflation values.

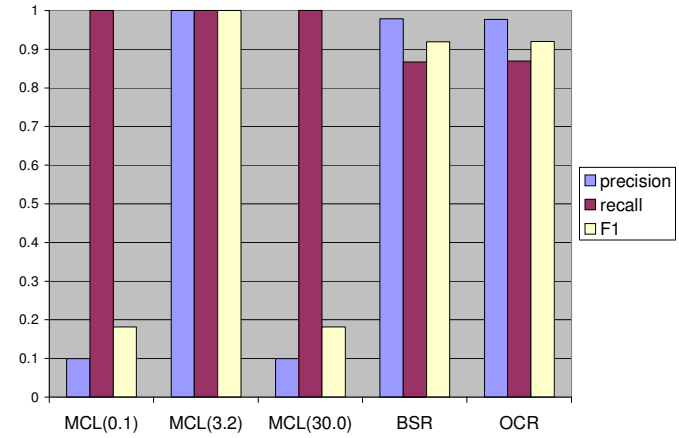

Figure 13. Effectiveness on Google Data

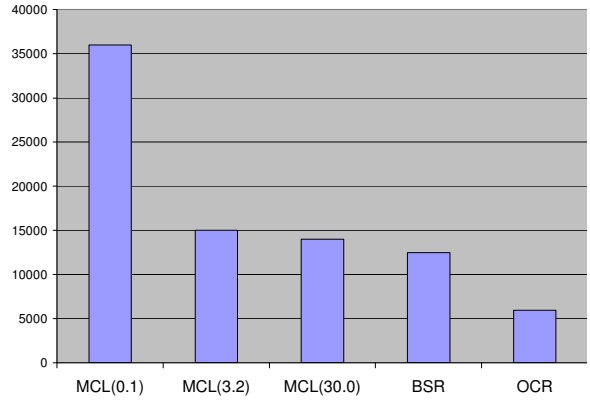

Figure 14. Efficiency on Google Data

On Tipster-AP data (cf. figure 15), BSR and OCR are slightly less effective than MCL at the best inflation value. However, both BSR and OCR are more effective than MCL at the minimum and maximum inflation values. In terms of efficiency (cf. figure 16), OCR is also much faster than MCL at all inflation values.

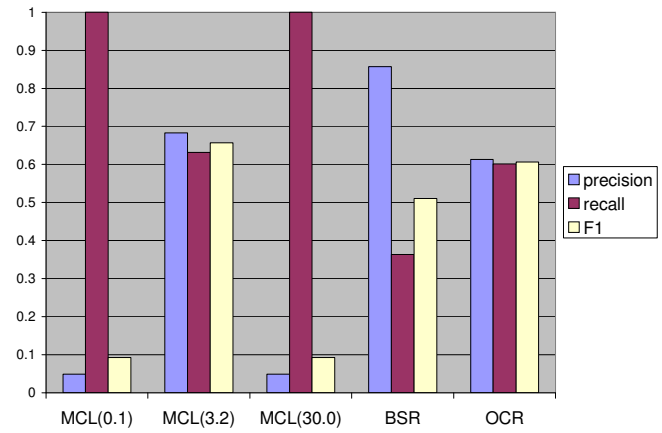

Figure 15. Effectiveness on Tipster-AP Data

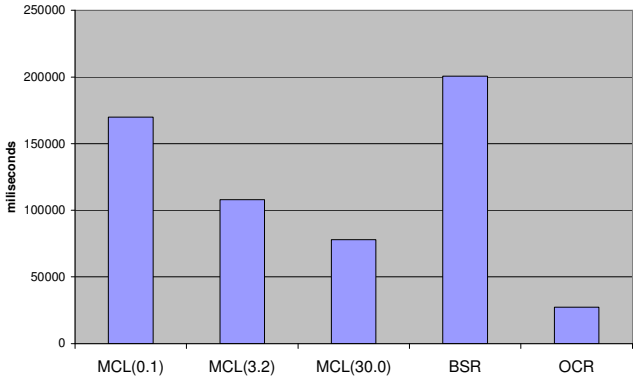

Figure 16. Efficiency on Tipster-AP Data

The same trend is also noticeable on Reuters data (cf. figure 17). 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 18), once again, OCR is much faster than MCL at all inflation values.

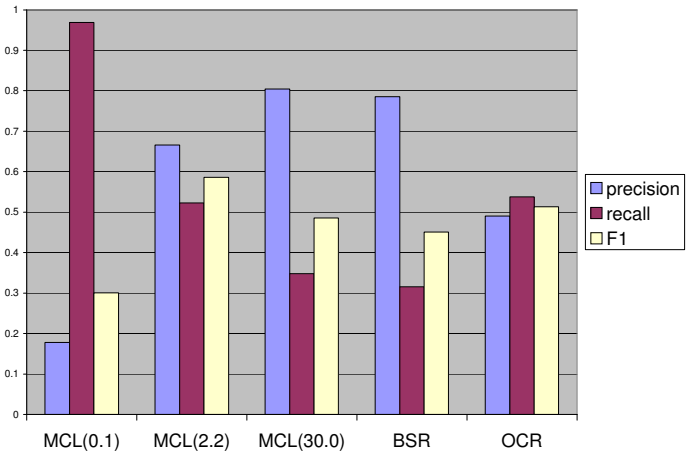

Figure 17. Effectiveness on Reuters Data

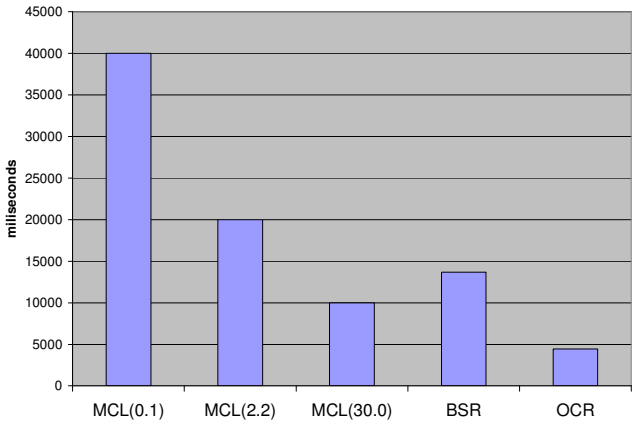

Figure 18. Efficiency on Reuters Data

In summary, although MCL can be slightly more effective than our proposed algorithms at the best settings and around, one of our algorithm, OCR, is not only respectably effective but also significantly more efficient.

5. CONCLUSIONS

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

The algorithms have been devised by spinning the metaphor of ripples created by the throwing of stones in a pond. Clusters' seeds are stones and rippling is the iterative assignment of objects to clusters.

We have proposed sequential (in which seeds are chosen one by one) and concurrent (in which every vertex is initially a seed) versions of the algorithms and variants.

After a comprehensive comparative performance analysis with reference data sets in the domain of document corpora clustering, we conclude that, while all our algorithms are competitive, one of them, Ordered Concurrent Rippling, yield a very respectable effectiveness while being the most efficient.

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

6. REFERENCES

[1] G. Salton, Automatic Text Processing: the transformation, analysis, and retrieval of information by computer, Addison-Wesley, 1989.

[2] G. Salton, The Smart document retrieval project, In Proceedings of the Fourteenth Annual International ACM/SIGIR Conference on Research and Development in Information Retrieval, pp 356-358, 1991.

[3] Ulrik Brandes, Marco Gaertler, Dorothea Wagner, Experiments on Graph Clustering Algorithms, Lecture Notes in Computer Science, Di Battista and U. Zwick (Eds.):568--579, 2003.

[4] J. B. MacQueen, Some Methods for classification and Analysis of Multivariate Observations, Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability, Berkeley, University of California Press, 1:281-297, 1967.

[5] J. Aslam; K. Pelekhov, D. Rus, The Star Clustering Algorithm, In Journal of Graph Algorithms and Applications, 8(1) 95–129, 2004.

[6] Stijn van Dongen, Graph clustering by flow simulation, 2000 - Tekst. - Proefschrift Universiteit Utrecht, 2000.

[7] L. Kaufman and P. Rousseeuw, Finding groups in data: an introduction to cluster analysis, New York: John Wiley and Sons, 1990.

[8] W. B. Croft, Clustering large files of documents using the single-link method, Journal of the American Society for Information Science, pp 189-195, November 1977.

[9] E. Voorhees, The cluster hypothesis revisited, In Proceedings of the 8th SIGIR, pp 95-104, 1985.

[10] C. Lund, and M. Yannakakis, On the hardness of approximating minimization problems, Journal of the ACM 41, pp. 1960-981, 1994.

[11] W. Press, B. Flannery, S. Teukolsky, W. Vetterling, Numerical Recipes in C: The Art of Scientific Computing, Cambridge University Press, 1988.

[12] Derry Wijaya and Stéphane Bressan, Journey to the Centre of the Star: Various Ways of Finding Star Centers in Star Clustering, accepted as a full paper in the 18th International Conference on Database and Expert Systems Applications (DEXA), 2007.

[13] Henk Nieland, Fast Graph Clustering Algorithm by Flow Simulation, Research and Development ERCIM News No. 42, July 2000.

[14] http://www.daviddlewis.com/resources/testcollections/reuters 21578 (visited in December 2006).

[15] http://trec.nist.gov/data.html (visited in December 2006).

[16] Google News (http://news.google.com.sg).

[17] B. Larsen and C. Aone, Fast and Effective Text Mining Using Linear-time Document Clustering, In KDD'99, San Diego, California, pp. 16–22, 1999.