On Caching for Local Graph Clustering Algorithms

René Speck and Axel-Cyrille Ngonga Ngomo

Universität Leipzig, Institut für Informatik, AKSW,
Postfach 100920, D-04009 Leipzig, Germany,
{speck|ngonga}@informatik.uni-leipzig.de

Abstract. In recent years, local graph clustering techniques have been utilized as devices to unveil the structured hidden of large networks. With the ever growing size of the data sets generated in domains of applications as diverse as biomedicine and natural language processing, time-efficiency has become a problem of growing importance. We address the improvement of the runtime of local graph clustering algorithms by presenting the novel caching approach SGD*. This strategy combines the Segmented Least Recently Used and Greedy Dual strategies. By applying different caching strategies to the unprotected and protected segments of a cache, SGD* displays a superior hitrate and can therewith significantly reduce the runtime of clustering algorithms. We evaluate our approach on four real protein-protein-interaction graphs. Our evaluation shows that SGD* achieves a considerably higher hitrate than state-of-the-art approaches. In addition, we show how by combining caching strategies with a simple data reordering approach, we can significantly improves the hitrate of state-of-the-art caching strategies.

Keywords: caching, local graph clustering, large networks

1 Introduction

Graphs are a natural representation for a large number of real-world problems and datasets ranging from protein-protein-interaction networks [1] to external memory data [2]. Over the last years, a large number of approaches have been developed to achieve the goal of clustering graphs with high accuracy [3, 4]. While the accuracy of these approaches is being studied continuously, improving their performance remains a major challenge [4, 5]. Current approaches to graph clustering can be subdivided into two main categories: global approaches, which require knowledge about the whole graph for clustering and local approaches, which find a solution vertex-wise without necessitating knowledge of the whole graph [2]. Local graph clustering algorithms were originally conceived to allow the detection of clusters around a small set $\mathcal{N}$ of nodes of interest, especially when dealing with very large graphs. However, local clustering approaches are nowadays often used to cluster whole graphs [6, 7]. One problem that then arises is the scalability of these approaches [5]. In this paper, we address the problem of improving the runtime of *local graph clustering algorithms* that allow *overlapping clusters*, especially when the magnitude of the set $\mathcal{N}$ of input vertices to process is close to the magnitude of the set of vertices. We present the novel caching strategy SGD* (Segmented Greedy Dual). SGD* combines the Segmented Least Recently Used

(SLRU) [8] and Greedy Dual (GD*) strategies to improve the hitrate during the clustering process so as to further reduce its runtime. In addition, we show how a simple node reordering strategy can further improve the hitrate of caching algorithms. We evaluate our approaches by using the BorderFlow algorithm[1] [9] on protein-protein-interaction (PPI) networks [1]. We chose BorderFlow because of its superior accuracy on PPI networks [6] and because it has already been applied in several domains including concept location in software development [7] and query clustering for benchmarking [10]. Our experiments show that SGD* outperforms state-of-the-art approaches with respect to its hit-rate and space requirements. In addition, we can more than quadruple the hitrate of common caching strategies and of SGD* by combining them with node reordering. By these means, we can reduce the runtime of BorderFlow to less than 25% of its original.

The rest of this paper is structured as follows: In the next section, we present some work related to this paper. Then, we present necessary preliminaries. Thereafter, we present our approaches, SGD* and RP. In the evaluation section, we compare our approaches with seven state-of-the-art caching approaches. Finally, we present relevant related work on caching for local graph clustering and conclude.

2 Related Work

A vast amount of literature has been produced to elucidate the problem of graph clustering [3, 4]. Still, with the growth of the size of the dataset at hand, improving the runtime of graph clustering becomes an increasingly urgent problem. Several approaches have been developed with the goal of improving the performance of graph clustering approaches. Overall, most of these approaches fall into one of the following two categories: sampling (also called graph sparsification) [11, 5] and caching [12]. Sampling is a generic solution to reducing the runtime of algorithms [13]. The idea here is to reduce the runtime of clustering approaches by computing a smaller representative subset of the data at hand and running the computation on this data set. While this approach can get rid of noise in the data, the alteration of the data set at hand might lead to undesired side-effects when combined with certain clustering strategies.

Caching follows a different idea and tries to store and reuse as much intermediary knowledge as possible to improve the runtime of the given algorithm. One of the most commonly used approaches is the Least Recently Used algorithm [14]. The idea behind this approach is simply to evict the entry that led to the oldest hit when the cache gets full. One of the main drawbacks of this approach is that the cache is not scan-resistant. Meanwhile, a large number of scan-resistant extensions of this approach have been created. For example, SLRU [8] extends LRU by splitting the cache into a protected and an unprotected area. The Least Frequently Used (LFU) [15] approach relies on a different intuition. Here, a count of the number of accesses to entries in the cache is kept. The cache evicts the entries with the smallest frequency count when necessary. This approach is scan-resistant but does not make use of the locality of reference. Consequently, it was extended by window-based LFU [16], sliding window-based approaches [17] and dynamic aging (LFUDA) [18] amongst others. Another commonly

[1] We used the free version of the algorithm whose code is available at `http://borderflow.sf.net`.

used caching strategy is based on the idea of first-in-first-out (FIFO) lists [19]. When the cache is full, this approach evicts the entry that have been longest in the cache. The main drawback of this approach is that it does not make use of locality. Thus, it was extended in several ways, for example by the "FIFO second chance" approach [19]. Other strategies such as Greedy Dual (GD*) [20] use a cost model to determine which entries to evict.

3 Preliminaries and Notation

3.1 Caching

The aim of caching is to reduce the runtime of algorithms by storing intermediate results of expensive computations. Formally, let $\mathcal{O} = \{o_1 ... o_n\}$ be a set of results that can be cached. Let $cost : O \to \mathbb{R}^+$ be a function that maps each object o with the cost of its computation. Furthermore, let $size : O \to \mathbb{R}^+$ be a function that maps each object o to its size. A cache $\mathcal{C}$ of maximal size $\mathcal{C}_{max}$ (with $\mathcal{C}_{max} \geq \max_{o \in \mathcal{O}} size(o)$) is a subset of $\mathcal{O}$ such that $\sum_{x \in \mathcal{C}} size(x) \leq \mathcal{C}_{max}$. An algorithm $\mathcal{A}$ that relies on caching issues a query sequence $\sigma : T \to \mathcal{O}$ ($T \subseteq \mathbb{N}$) to the cache $\mathcal{C}$. At each time $t \in T$, the query $\sigma(t)$ for an object $o \in O$ is sent to the cache $\mathcal{C}$. If the cache contains the object o, it simply returns the corresponding solution to the clustering problem (this is usually called a *cache hit*). Else, $\mathcal{C}$ returns $\emptyset$ (*cache miss*). In case of a hit, the cost for $cost(\sigma(t))$ is a constant c called the cache latency. In case of a miss, $\mathcal{A}$ must compute o with the cost $cost(o)$, leading to $cost(\sigma(t)) = cost(o)$. The result of the computation is then forwarded to $\mathcal{C}$. As $cost(o)$ is usually vastly superior to c, we will assume $c = 0$ in the remainder of this paper. Caching algorithms aim to minimize the total cost $\sum_{t \in T} cost(\sigma(t))$ of the sequence σ by generating a sequence of cache states $\mathcal{C}^t$ for each time t that abide by $|\mathcal{C}^t|$.

3.2 Local Graph Clustering with Overlapping Clusters

Let $G = (V, E, w)$ be a graph, where V is the set of edges, $E \subseteq V \times V$ is the set of edges and $w : E \to \mathbb{R}^+$ the weight function that assign a weight to each edge of the graph G. A graph clustering algorithm aims to determine a set $\mathcal{V} = \{V_1, V_2, ..., V_n\}$ of subsets of V that maximize a certain fitness function [3]. Some local graph clustering algorithms allow for clusters to share nodes, i.e., $|V_i \cap V_j| > 0$ with $i \neq j$. Such algorithms are called *non-partitioning approaches* [21]. For the purpose of clustering, local graph clustering algorithms rely on a set of nodes $\mathcal{N} \subseteq V$ as input. For each node of interest $n \in \mathcal{N}$, they aim to discover a nearby cluster[2]. This is carried out by running iterative approaches of which most rely on local search [22, 23, 9] and random walks [24–26]. The idea behind these iterative procedures is to carry a simple operation repeatedly until the fitness function is maximized. For example, search algorithms begin with an initial solution $S^0(n)$ for $n \in \mathcal{N}$. At each step t they compute the current solution $S^t(n)$ by

[2] In most cases, n must be an element of this cluster.

altering the previous solution $S^{t-1}(n)$. This is carried out by adding a subset of the adjacent nodes of $S^{t-1}(n)$ to the solution and simultaneously removing a subset of the nodes of $S^{t-1}(n)$ from it until the fitness function is maximized. They then return the final solution $S(n)$. The insight that makes caching utilizable to reduce the runtime of local graph clustering algorithms is that if an algorithm generates the same intermediate solution for two different nodes, then the final solution for these nodes will be the same, i.e., $\forall n, n' \in \mathcal{N}\ S_1^t(n) = S_2^t(n') \rightarrow S(n) = S(n')$. Thus, by storing some elements of the sequence of solutions computed for previous nodes n and the solution $S(n)$ to which they led, it becomes possible to return the right solution of a node n' without having to compute the whole sequence of solutions. However, it is impossible to store all elements of the sequence of solutions generated by local graph clustering algorithms for large input graphs and large $\mathcal{N}$. The first innovation of this paper is a novel caching approach for local graph clustering dubbed SGD* that outperforms the state-of-the-art w.r.t. its hitrate. Note that $\mathcal{N}$ is a set, thus the order in which its nodes are processed does not affect the final solution of the clustering. Consequently, by finding an ordering of nodes that ensures that the sequence of solutions generated for subsequent nodes share a common intermediate as early as possible in the computation, we can improve the hitrate of caching algorithms and therewith also the total runtime of algorithms. This is the goal of the second innovation of this paper, our node reordering strategy.

4 Segmented Greedy Dual

SGD* combines the ideas of two caching strategies: SLRU and GD*. SLRU is a scan-resistant extension of LRU [14], one of the most commonly used caching strategies. Like LRU, it does not take the cost of computing an object into consideration and thus tends to evict very expensive objects for the sake of less expensive one. GD* on the other hand is an extension of the Landlord algorithm [27] which takes the costs and the number $hit(o)$ of cache hit that return o into consideration.

The idea behind SGD* is to combine these strategies to a scan-resistant and cost-aware caching approach. To achieve this goal, SGD* splits the cache into two parts: a protected segment and an unprotected segment. The unprotected segment stores all the $S^t(n) \subseteq V$ that are generated while computing a solution for the input node $n \in \mathcal{N}$. The protected segment on the other hand stores all the results that have been accessed at least once and contain at least two nodes. While SLRU uses LRU on both the protected and unprotected area, SGD* uses the GD* strategy on the unprotected area and the LRU approach on the protected area. An overview of the resulting caching approach is given in Algorithm 1. For each node n, we begin by computing the first intermediary result for n. Then we iterate the following approach. We ask the cache for the head (i.e., the first element) of the list S. If this element is not in the cache, we compute the next intermediary solution and add it to the head of the list. The iteration is terminated out in one of the following two cases. In the first case, the iteration terminates for n, returning $\perp$. Then, the result is added to the list S and S is cached. In the second case, a solution is found in the cache. Then this solution is cached. Note that this approach works for every caching mechanism. The main difference between caching approaches is how they implement the storage method $cachePut$ and the data fetching method $cacheGet$.

Algorithm 1 Caching for local graph clustering

Require: Set of nodes $\mathcal{N}$
  List S
  Buffer B, id
  Result $R = \emptyset$
  Protected segment $P = \emptyset$
  Unprotected segment $U = \emptyset$
  for all $n \in \mathcal{N}$ **do**
    S =compute(n)
    id =cacheGet(S)
    while $id == -1$ **do**
      B =compute(S)
      if $B == \perp$ **then**
        cachePut(S)
        $R = R \cup (n$,cacheGet(S))
        break
      end if
      S = append(B, S)
      id =cacheGet(S)
    end while
    $R = R \cup (n, id)$
  end for
  return R

The fetching data algorithm of the SGD* cache has two functions and is summarized in Algorithm 2. First, it allows checking whether the data that is being required is in the cache. Concurrently, it reorganizes the data in the cache in case of a cachehit. The SGD* data fetching approach and works as follows: In case there is no cachehit, the cache simply returns -1. A cache hit can occur in two ways: First, the current solution can be contained in the protected segment of the cache. In this case, SGD* simply updates the credit of the entry o, i.e., of the cached entry that led to finding the cached solution to the clustering task for the current node. It then computes the id of the answer to the current caching and returns it. Note that there is then no need to evict data, as no new data is added. If the cache hit occurs within the unprotected segment U of the cache, the algorithm moves the entry o that led to the hit from U to the protected segment P of the cache. Should P exceeds its maximal size, then the elements with the smallest credit score are evicted to the unprotected segment U of the cache until there is enough space for o in P. o then gets inserted into P and its credit score is computed. The final step in case of a cachehit consists of assigning all the steps that led to the solution mapped to s. For this purpose (see Algorithm 3), each single component o_i of the solution of S is inserted into U. In case the cache would exceed its maximal size when accommodating o_i, the elements of U are evicted in ascending order of credit until enough space is available for o_i. [3]

[3] We implemented the approach and made it freely available at `http://sourceforge.net/projects/cugar-framework`.

Algorithm 2 SGD*'s $cacheGet$

Require: Solution S
  $s = head(S)$
  $id = -1$
  if $s \in P$ **then**
    $credit(s) = min + \frac{(hit(o)cost(o))^{\frac{1}{b}}}{size(o)}$
    $id = id(s)$
  else
    if $s \in U$ **then**
      $id = id(s)$
      $U = U \backslash s$
      $P = P \cup s$
      while $|P| > \mathcal{C}_{max}/2$ **do**
        $o = \arg\min_{o' \in U} credit(o')$
        $U = U \cup \{o\}$
        $P = P \backslash \{o\}$
      end while
    end if
  end if
  if $id \neq -1$ **then**
    for all $s_i \in S$ **do**
      cachePut(s_i, id)
    end for
  end if
  return id

5 Node Reordering

While SGD* outperforms the state of the art as shown in our experiments, the general behavior of caching algorithms can be further improved when assuming that the set $\mathcal{N}$ is known at the beginning of the clustering. Note that this condition is not always given, as many practical clustering approaches process the results for known nodes of interest to generate novel nodes of interest. Yet, when this condition is given and when in addition the computation of a cluster for a node n does not affect the set $\mathcal{N}$ or the computation of a cluster for another node n', the order in which the nodes are drawn from $\mathcal{N}$ does not alter the result of the clustering and can be dynamically changed during the computation. By choosing the order in which this selection is carried out, we can drastically improve the locality of caching algorithms. We propose a simple and time-efficient approach to achieve this goal: the use of a FIFO list. For this purpose, we extend the cachePut method as shown in Algorithm 4.

The FIFO list L simply stores the elements of $\mathcal{N}$ that were part of a solution (note that the elements of a solution must not all belong to $\mathcal{N}$). Instead of drawing n from $\mathcal{N}$ as described in Algorithm 1, we draw the node n by taking the first element of L if is not empty (in which case we draw one at random from $\mathcal{N}$). The rationale behind using a FIFO list is that by processing nodes n' that are closest to the input node n first, we can reduce the number of iterations necessary for a cache hit to occur. While this

Algorithm 3 SGD*'s $cachePut$

Require: Solution S_i
Require: Set of input nodes $\mathcal{N}$
Require: ID id
  min = 0 // minimal credit
  while $|P| + |U| + size(S_i) > \mathcal{C}_{max}$ **do**
    $o = \arg\min_{o' \in U} credit(o')$
    min = $credit(o)$
    $U = U \backslash \{o\}$
  end while
  $credit(S_i) = min + \frac{(hit(o)cost(o))^{\frac{1}{b}}}{size(o)}$
  $id(S_i) = id$
  $U = U \cup \{S_i\}$

assumption might appear simplistic, our evaluation shows that it suffices to reduce the space requirement of caches by a factor up to 40.

Algorithm 4 SGD*'s $cachePut$ with node reordering

Require: Solution S_i
Require: Set of input nodes $\mathcal{N}$
Require: ID id
  min = 0 // minimal credit
  for all $x \in S_i$ **do**
    if $x \notin L \wedge x \in \mathcal{N}$ **then**
      $L = append(L, x)$
    end if
  end for
  while $|P| + |U| + size(S_i) > \mathcal{C}_{max}$ **do**
    $o = \arg\min_{o' \in U} credit(o')$
    min = $credit(o)$
    $U = U \backslash \{o\}$
  end while
  $credit(S_i) = min + \frac{(hit(o)cost(o))^{\frac{1}{b}}}{size(o)}$
  $id(S_i) = id$
  $U = U \cup \{S_i\}$

6 Evaluation

6.1 Experimental Setup

As experimental data, we used four graphs resulting from high-throughput experiments utilized in [1]. The high-throughput graphs were computed out of the datasets published

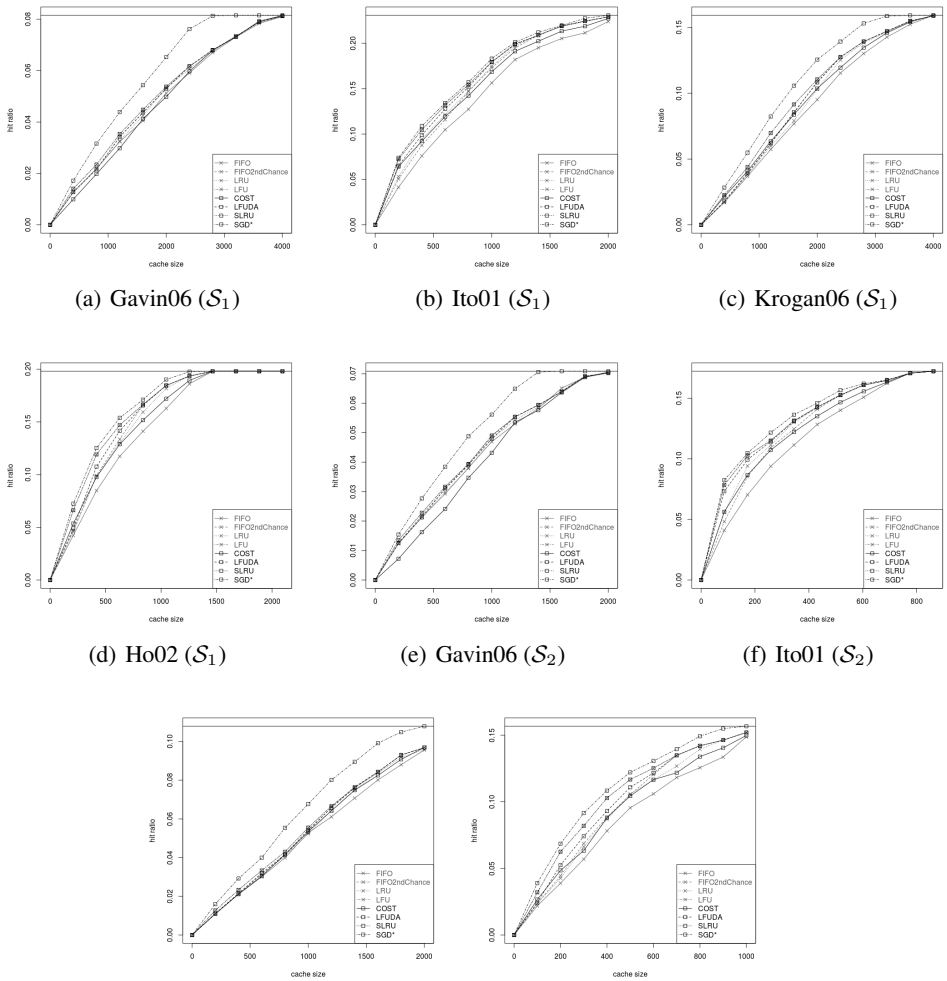
in [28] (Gavin06), [29] (Ho02), [30] (Ito01) and [31] (Krogan06).[4] The graphs were undirected and unweighted. We used the BorderFlow algorithm as clustering algorithm because it has been shown to perform best on these data sets [6]. We compared our approach against the standard strategies FIFO, FIFO second chance, LRU, LFU, LFU-DA and SLRU strategies. In addition, we developed the COST strategy, which evicts the entries with the highest costs. The idea here is that solutions S^t with high costs are usually generated after a large number t of iterations. Thus, it is more sensible to store the entries $S^{t'}$ that are less costly than S^t, as a corresponding cache hit is more probable and would reduce the total runtime of the algorithm. We compare these caching approaches in two series of experiments. In the first series of experiments, we compared the hitrate of the different caching approaches without node reordering. In the second series, we clustered exactly the same data with node reordering. In each series of experiment, we used two different settings for $\mathcal{N}$. In the first setting, $\mathcal{S}_1$, we used all nodes of each graph. In the second setting, $\mathcal{S}_2$, we only considered the nodes with a connectivity degree least or equal to the average degree of the graph. All measurements were carried out at least four times on an Intel Core i3-2100 (3.1GHz) with 4GB DDR3 SDRAM running Windows 7 SP1. In the following, we report the best runtime for each of the caching strategies. Note that the cache size is measured in the number of intermediary results it contains.

6.2 Results

The results of the first series of experiments are shown in Fig. 1. The caching strategies display similar behaviors in both settings $\mathcal{S}_1$ and $\mathcal{S}_2$. In both settings, our results clearly show that the hitrate of SGD* is superior to that of all other strategies. Especially, we outperform the other approaches by more than 2% hitrate on the Gavin06 graph. SGD* seems to perform best when faced with graphs such that the baseline (i.e., the hitrate with an infinite cache) is low. This can be clearly seen in the experiments carried out with the graph Gavin06 (see Fig. 1(a) and 1(e))). In the first setting, SGD* reaches the baseline hitrate of 8% with a cache size of 2800 (see Fig. 1(a)), while all other approaches require a cache size of at least 4000. Similarly, in the second setting (see Fig. 1(e)), the maximal hitrate of 7% is reached for a cache size of 1600. An analogical behavior of can be observed when processing the Krogan06 graph (see Fig. 1(c) and 1(g))). The consequences of this behavior are obviously that our approach requires significantly less space to achieve better runtime improvements (see Table 1). Note that COST achieves runtimes similar to that of common strategies such as LRU and FIFO.

The results of the second series of experiments are shown in Fig. 2. The reordering of nodes significantly improves the runtime of all caching strategies in all settings, allowing all strategies apart from Cost and LFU to reach the baseline with a cache size of 100 on the full Krogan06 and Ho02 graphs (i.e., in setting $\mathcal{S}_1$, see Fig 2(c) and 2(d)). In setting $\mathcal{S}_2$, the baseline hitrate is reached with a cache size of 50 on the same graphs (see Fig. 2(g) and 2(h)). Therewith, node reordering can make most caching strategies more than 40 times more space-efficient (compare Fig. 1(c) and 2(c)). The COST

[4] All data sets used for this evaluation can be found at `http://rsat.bigre.ulb.ac.be/~sylvain/clustering_evaluation/`



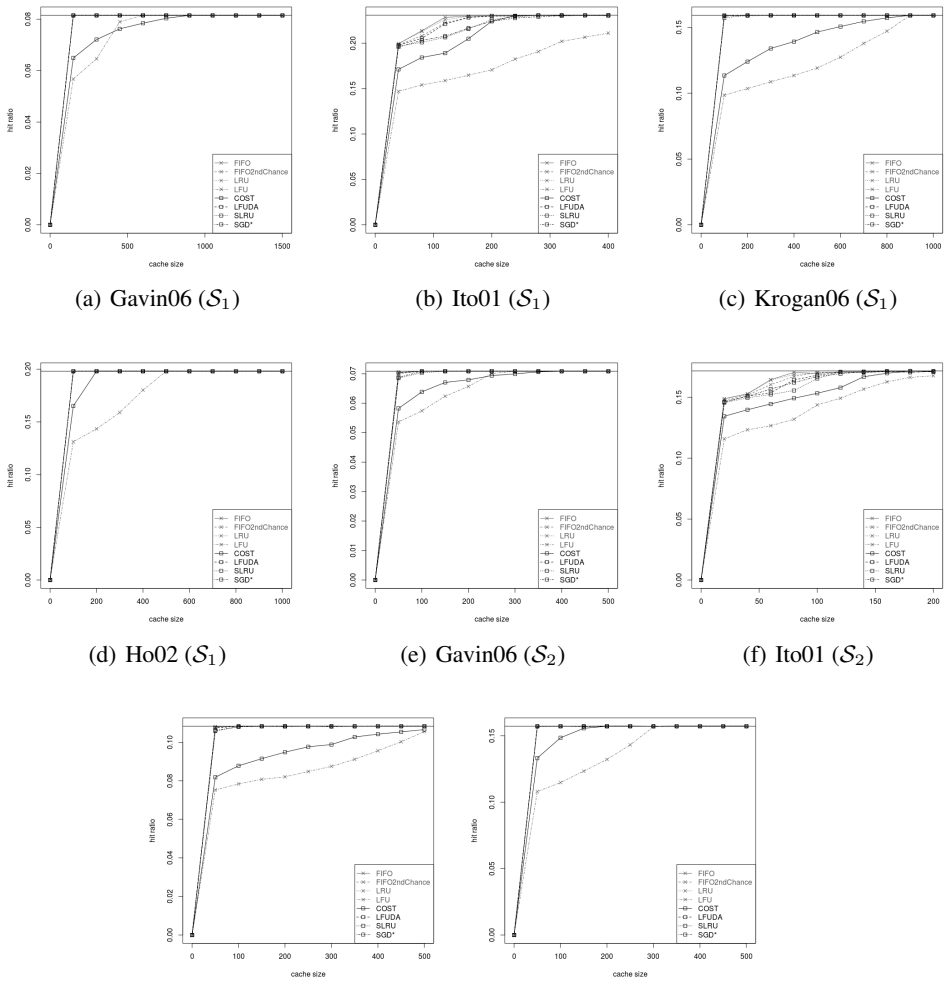
(a) Gavin06 ($\mathcal{S}_1$) (b) Ito01 ($\mathcal{S}_1$) (c) Krogan06 ($\mathcal{S}_1$)

(d) Ho02 ($\mathcal{S}_1$) (e) Gavin06 ($\mathcal{S}_2$) (f) Ito01 ($\mathcal{S}_2$)

(g) Krogan06 ($\mathcal{S}_2$) (h) Ho02 ($\mathcal{S}_2$)

Fig. 1. Comparison of the hitrate of SGD* against seven other approaches

and LFU approaches not profiting maximally from the node reordering is simply due to cache pollution. The cost-based approach deletes those elements, which required a long processing time, the idea being that they are unlikely that they appear again. Yet, this approach leads to the content of the cache remaining static early in the computation. Consequently, reordering the nodes does not improve the hitrate of such caches as significantly as that of FIFO, SLRU and other strategies, especially when the cache is small. LFU behaves similarly with respect to the hitrate score of the elements in the cache. Overall, by combining SGD* and node reordering, we can improve the runtime of BorderFlow to less than 25% of its original runtime on the Ito01 graph.



(a) Gavin06 ($\mathcal{S}_1$) (b) Ito01 ($\mathcal{S}_1$) (c) Krogan06 ($\mathcal{S}_1$)

(d) Ho02 ($\mathcal{S}_1$) (e) Gavin06 ($\mathcal{S}_2$) (f) Ito01 ($\mathcal{S}_2$)

(g) Krogan06 ($\mathcal{S}_2$) (h) Ho02 ($\mathcal{S}_2$)

Fig. 2. Comparison of the hitrate of SGD* against seven other approaches with node reordering

7 Conclusion and Future Work

SGD* addresses the cost-blindness of SLRU by combining it with the GD* caching strategy. We showed that this combined strategy outperforms state-of-the-art approaches with respect to its hitrate. We also presented an approach to improve the locality of caching when dealing with clustering approaches where the order of the input nodes does not alter the result of the clustering. One interesting results was that once we apply reordering to the input data (therewith improving the locality of the clustering), we could boost the results of the FIFO caching approach and make it outperform all others

Table 1. Comparison of runtimes with cache size 300 in setting $\mathcal{S}_1$. The best runtimes are in bold font. All runtimes are in ms. The columns labeled "Default" contain the runtime of our approaches without node reordering. The columns labeled "Reordered" contain the runtimes after the reordering has been applied.

	Gavin06		Ho02		Ito01		Krogan06	
	Default	Reordered	Default	Reordered	Default	Reordered	Default	Reordered
Baseline	14944	14944	8938	8938	42775	42775	25630	25630
FIFO	14196	**9921**	6848	**3759**	17316	9578	23758	**12558**
FIFO2ndChance	14071	9937	6692	3790	15303	9594	23868	12604
LRU	14164	9968	6708	3790	13525	9578	23899	12667
LFU	14008	12214	6099	5038	12324	11029	23244	19156
LFU-DA	14102	9984	6645	3790	12745	9609	23821	12698
SLRU	13946	9937	6052	3790	12370	**9531**	23197	**12558**
SGD*	**13821**	9968	**5912**	**3759**	**12261**	9687	22916	12682
COST	13915	11200	6318	3775	12604	9578	**21980**	14835

in most cases. In future work, we will combine our caching approach with other clustering algorithms. Note that our caching approach is not limited to graph clustering and can be easily applied to any other problem that necessitates caching. Consequently, we will also apply SGD* to other tasks such as the management of very large graphs.

References

1. Brohee, S., van Helden, J.: Evaluation of clustering algorithms for protein-protein interaction networks. BMC Bioinformatics **7** (November 2006) 488–506
2. Dalvi, B.B., Kshirsagar, M., Sudarshan, S.: Keyword search on external memory data graphs. PVLDB **1**(1) (2008) 1189–1204
3. Schaeffer, S.: Graph clustering. Computer Science Review **1**(1) (2007) 27–64
4. Fortunato, S.: Community detection in graphs. Physics Reports **486**(3-5) (2010) 75 – 174
5. Satuluri, V., Parthasarathy, S., Ruan, Y.: Local graph sparsification for scalable clustering. In: SIGMOD '11. (2011) 721–732
6. Ngonga Ngomo, A.: Parameter-free clustering of protein-protein interaction graphs. In: Proceedings of Symposium on Machine Learning in Systems Biology 2010. (2010)
7. Scanniello, G., Marcus, A.: Clustering support for static concept location in source code. In: ICPC. (2011) 1–10
8. Karedla, R., Love, J.S., Wherry, B.G.: Caching strategies to improve disk system performance. Computer **27** (1994) 38–46
9. Ngonga Ngomo, A.C., Schumacher, F.: Borderflow: A local graph clustering algorithm for natural language processing. In: CICLing. (2009) 547–558
10. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.C.N.: Dbpedia sparql benchmark - performance assessment with real queries on real data. In: International Semantic Web Conference. (2011) 454–469
11. Kanjirathinkal, R.C., Sudarshan, S.: Graph clustering for keyword search. In: COMAD. (2009)
12. Kumar, M., Agrawal, K.K., Arora, D.D., Mishra, R.: Implementation and behavioural analysis of graph clustering using restricted neighborhood search algorithm. International Journal

of Computer Applications **22**(5) (May 2011) 15–20 Published by Foundation of Computer Science.
13. Provost, F., Kolluri, V.: A survey of methods for scaling up inductive algorithms. Data Mining and Knowledge Discovery **3** (1999) 131–169
14. O'Neil, E.J., O'Neil, P.E., Weikum, G.: The lru-k page replacement algorithm for database disk buffering. SIGMOD Rec. **22** (1993) 297–306
15. Breslau, L., Cao, P., Fan, L., Phillips, G., Shenker, S.: Web caching and zipf-like distributions: Evidence and implications. In: INFOCOM. (1999) 126–134
16. Karakostas, G., Serpanos, D.N.: Exploitation of different types of locality for web caches. In: Proceedings of the Seventh International Symposium on Computers and Communications. (2002) 207–2012
17. Hou, W.C., Wang, S.: Size-adjusted sliding window lfu - a new web caching scheme. In: Proceedings of the 12th International Conference on Database and Expert Systems Applications. (2001) 567–576
18. Arlitt, M., Cherkasova, L., Dilley, J., Friedrich, R., Jin, T.: Evaluating content management techniques for web proxy caches. SIGMETRICS Performance Evaluation Review **27**(4) (2000) 3–11
19. Tanenbaum, A.S., Woodhull, A.S.: Operating systems - design and implementation (3. ed.). Pearson Education (2006)
20. Jin, S., Bestavros, A.: Greedydual* web caching algorithm – exploiting the two sources of temporal locality in web request streams. In: 5th International Web Caching and Content Delivery Workshop. (2000) 174–183
21. Schlitter, N., Falkowski, T., Lässig, J.: Dengraph-ho: Density-based hierarchical community detection for explorative visual network analysis. In Springer, ed.: Proceedings of the 31st SGAI International Conference on Artificial Intelligence. (2011)
22. Schaeffer, S.: Stochastic local clustering for massive graphs. In Ho, T., Cheung, D., Liu, H., eds.: Proceedings of the Ninth Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD-05). Volume 3518 of LNCS., Springer (2005) 354–360
23. Felner, A.: Finding optimal solutions to the graph partitioning problem with heuristic search. Ann. Math. Artif. Intell. **45**(3-4) (2005) 293–322
24. Alamgir, M., von Luxburg, U.: Multi-agent random walks for local clustering on graphs. In: ICDM. (2010) 18–27
25. Spielman, D.A., Teng, S.H.: A local clustering algorithm for massive graphs and its application to nearly-linear time graph partitioning. CoRR **abs/0809.3232** (2008)
26. Biemann, C., Teresniak, S.: Disentangling from babylonian confusion - unsupervized language identification. In: Proceedings of CICLing-2005, Computational Linguistics and Intelligent Text Processing, Springer (2005) 762–773
27. Young, N.E.: On-line file caching. In: Proceedings of the ninth annual ACM-SIAM symposium on Discrete algorithms. (1998) 82–86
28. Gavin, A.C., et al.: Proteome survey reveals modularity of the yeast cell machinery. Nature (January 2006)
29. Ho, Y., et al.: Systematic identification of protein complexes in saccharomyces cerevisiae by mass spectrometry. Nature **415**(6868) (January 2002) 180–183
30. Ito, T., et al.: A comprehensive two-hybrid analysis to explore the yeast protein interactome. Proc Natl Acad Sci U S A **98**(8) (April 2001) 4569–4574
31. Krogan, N., et al.: Global landscape of protein complexes in the yeast saccharomyces cerevisiae. Nature (March 2006)