Partition-Based Clustering in Object Bases: From Theory to Practice

Carsten Gerlhof[*] *Alfons Kemper*[*] *Christoph Kilger*[†] *Guido Moerkotte*[†]

[*]RWTH Aachen
Lehrstuhl für Informatik III
Ahornstraße 55
W-5100 Aachen, Germany
[*gerlhof* | *kemper*]@informatik.rwth-aachen.de

[†]Universität Karlsruhe
Fakultät für Informatik
Am Fasanengarten
W-7500 Karlsruhe, Germany
[*kilger* | *moer*]@ira.uka.de

Abstract

We classify clustering algorithms into *sequence-based* techniques—which transform the object net into a linear sequence—and *partition-based* clustering algorithms. Tsangaris and Naughton [TN91, TN92] have shown that the partition-based techniques are superior. However, their work is based on a single partitioning algorithm, the Kernighan and Lin heuristics, which is not applicable to realistically large object bases because of its high running-time complexity.

The contribution of this paper is two-fold: (1) we devise a new class of greedy object graph partitioning algorithms (GGP) whose running-time complexity is moderate while still yielding good quality results. For large object graphs GGP is the best known heuristics with an acceptable running-time. (2) We carry out an extensive quantitative analysis of all well-known partitioning algorithms for clustering object graphs. Our analysis yields that no *one* algorithm performs superior for *all* object net characteristics. Therefore, we derive a multi-dimensional grid: the dimensions correspond to particular characteristics of the object base configurations and the grid entries indicate the best clustering algorithm for the particular configuration. We propose an *adaptable* clustering strategy by determining first the characteristics of the clustering problem—given by, e.g., number and size of objects, degree of object sharing—and then applying the most suitable algorithm which is obtained from the multi-dimensional grid.

1 Introduction

Clustering of logically related objects on the same page is a very powerful optimization concept. Unlike other, more restricted models—e.g., the hierarchical model or the nested relational model—the object-oriented data models allow arbitrary object graphs. From the object graph a *cluster graph* is derived whose vertices correspond to objects and whose weighted edges correspond to logical relationships between objects, e.g., traversal frequencies of inter-object references. *Clustering*, then, consists of mapping the objects of the cluster graph onto pages. We distinguish between *sequence-based* and *partition-based* clustering. Under sequence-based clustering the cluster graph is transformed into a linear sequence of objects which is then sequentially assigned to pages. Under partition-based clustering the cluster graph is partitioned into object partitions that fit onto a single page. The goal, of course, is to minimize the total weight of those edges of the cluster graph that cross page boundaries.

Tsangaris and Naughton [TN91, TN92] showed that the partition-based clustering is superior to sequence-based clustering. Unfortunately, the well-known partitioning algorithms—such as the Kernighan and Lin (KL) algorithm, which was the only one investigated in [TN91, TN92]—have a very high running-time complexity. This makes their application to realistically large object bases impossible.

Therefore, in this work we develop a new class of *greedy graph partitioning (GGP)* heuristics which have a moderate running-time complexity while still yielding good quality results. The starting point for this class of algorithms is Kruskal's algorithm for computing the minimum spanning trees [Kru56]. The basic GGP algorithm is an adaptation of Kruskal's algorithm to graph partitioning—obtaining, however, only a sub-optimal solution for the NP-complete partitioning problem. From the basic GGP heuristics we develop more sophisticated heuristics by incorporating a *look-ahead* with the possibility of rejecting less promising choices and a *new-chance* for those rejected choices at a later stage.

Our extensive quantitative analysis—of which only a fraction could be covered in the paper—indicates that the quality of the (sophisticated) greedy partitioning is slightly inferior to the computationally complex algorithms, such as KL. However, the greedy heuristics are indispensable for two reasons:

1. They are extremely useful for pre-partitioning the cluster graph. These pre-partitions can then be improved using the computationally complex algorithms, e.g., KL.

2. For (realistically) large object bases the computationally complex algorithms cannot be employed because of their enormous running-time (e.g., KL needs more than 24 hours for clustering 16.000 objects). In this case we can employ our greedy heuristic which is far better than the best known sequence-based algorithms—the *best-first* heuristics.

The above discussion leads to the conclusion that there is no *one* single cluster heuristics which is superior for *all* object base configurations. This motivates us to propose an *adaptable* cluster strategy: depending on the characteristics of the object base configuration—given by, e.g., the number and size of objects, the degree of object sharing—the best known algorithm should be applied. To support this (algorithmic) adaptability we derive a multi-dimensional grid from our extensive benchmarks: the dimensions of the grid correspond to particular characteristics of the object base configuration and the grid entries indicate the most suitable cluster heuristics for the particular characteristics.

The rest of the paper is organized as follows. In Section 2 the cluster problem is defined and the related work is classified. Then, in Section 3 the well-known partitioning algorithms are reviewed and the basics of our greedy graph partitioning heuristics is introduced. In Section 4 the basic GGP heuristics is enhanced by the *look-ahead* and *new-chance* heuristics. Section 5 contains some of our extensive quantitative analysis of many different partition-based cluster heuristics. Based on the outcome of this analysis, we derive the adaptable cluster strategy controlled by the multi-dimensional grid, in Section 6. Section 7 concludes the paper. The Appendix contains the complete specification of our sophisticated greedy partitioning algorithm.

2 Clustering as a Graph Partitioning Problem

The clustering problem is closely related to the graph partitioning problem where some graph is to be partitioned into several disconnected subgraphs (partitions). The *object graph (OG)* is constructed considering objects as vertices and the inter-object references as directed edges [TN91]. Clustering algorithms partition the OG by assigning objects to equally sized pages. Instead of the OG clustering algorithms often use a more specific graph as input that is derived from the OG and/or from information about the applications' access behavior, e.g., access traces. We follow [TN91] and call this graph the *clustering graph (CG)*. The vertices and edges of the CG are labeled with weights: vertex weights represent object sizes and edge weights represent the application's access behavior (higher weights denote that the start and terminal object of the edge are more often accessed in succession). Generally, the set of edges of the CG may be a multi-set. However, in order to simplify the problem, we propose to transform the CG into a simple graph, i.e., every edge occurs only once, by accumulating the weights of edges between the same start and terminal vertices. For a given partitioning of the CG, the total weight of all edges crossing partition borders (i.e., page borders) are the *external costs* of this partitioning.[1] The clustering problem is to find a partitioning of the CG such that the size of each partition, i.e., the total size of its objects, is less or equal the page size and the external costs are minimized. This problem is an instance of the *free partitioning problem* as the number of pages taken to store the objects of the CG is not part of the problem specification. It is known that this problem is strongly NP-complete, even if all objects are of equal size and all edge weights equal 1. The problem is in P if the CG is a tree and all objects have the same size [GJ79] or all edge weights equal 1 [Had74].

For a given CG and a given number of partitions r a lower bound for the optimum external costs was found by Barnes [Bar82]:

$$EC_{\mathrm{opt}} \;\; \geq \;\; EC_{\mathrm{max}} - \frac{1}{2} \sum_{i=1}^{r} B * \lambda_i(C)$$

where EC_{opt} and EC_{max} denote the optimum and maximum external costs, respectively, C is the weighted adjacency matrix of the CG, and B is the maximum number of ob-

[1] The external costs are often called the *cut size* of the partitioning.

jects per page. $\lambda_i(C)$ denotes the ith largest eigenvalue of C. A stronger—but more complicated—lower bound for EC_{opt} was proposed by [DH73].

There are two dimensions along which clustering algorithms can be classified: (1) the determination of the access patterns, i.e., edge weights, and (2) the algorithm that is applied to map objects onto pages. Along the first dimension—the determination of the access patterns—*static* and *dynamic methods* can be distinguished. Along the second dimension we distinguish *sequence-based* from *partition-based* mapping algorithms.

2.1 Determination of the Access Patterns

Static methods are based on analyzing either the schema or the object base—or both, in combination. In [GKKM92] we proposed to extract information on the access behavior of operations (that are part of the schema) by analyzing their data flow. The result of this analysis is a set of weighted path expressions, that are matched with the object graph in order to estimate the transition counts of the inter-object references. These were then used to label the edges, yielding the CG. Other static methods are based on the static reference counts of objects[2] [Sta84] or request input from the database programmer, e.g., [Ben90, CDRS86].

Dynamic methods analyze the access trace of former database applications by monitoring the system. For example, the number of times a reference is traversed can be counted yielding the *dynamic transition count* of this reference [Sta84, HK89]. The dynamic transition count of some reference is used to label the corresponding edge in the CG.

Usually, it is far more expensive to gather dynamic information on the access behavior of applications since this requires monitoring the applications. For a thorough discussion of the pros and cons of static versus dynamic access information see [GKKM92]. In the remainder of this paper we assume the weights of the edges of the CG to be given—either by dynamic or static analysis.

2.2 Sequence-Based Mapping Algorithms

Sequence-based algorithms partition the CG by applying a graph traversal algorithm producing a linear sequence of all objects from the CG. This sequence of objects is then segmented—from left to right—into partitions. Sequence-based algorithms can be denoted in the style of a UNIX pipe consisting of the two steps *PreSort* and *Traversal*:

$$\textit{PreSort} \quad | \quad \textit{Traversal}$$

The *Traversal* component starts with some object and traverses all objects that are reachable from the start object and have not been visited before. After the current traversal is finished, e.g., because all reachable objects have been visited, a new (unvisited) object is selected to start the next traversal. The *PreSort* method is used for sorting all objects—and, by that, specifying the order in which objects are considered as starting points for new traversals.

[2] The static reference count of some object equals the number of references pointing to that object.

For the *PreSort* order many sortings have been proposed, e.g. ordering by type [HZ87, Sta84, Ben90], ordering by decreasing dynamic reference counts[3] [HK89], and using some arbitrary ordering [Sta84, BKKG88].

The simplest *Traversal* algorithm maps objects to pages according to their position in the presort order; it is applied in [Sta84, HZ87]. The depth-first and breadth-first traversal algorithms for graphs were applied in [Sta84, BKKG88]. In Cactis [HK89] the usage of a best-first (BSTF) traversal algorithm is proposed:

> The first object according to the presort order is stored on a new page. To choose the next object all edges emanating from the current page are inserted into an *EdgeList*, which is then ordered by descending weights. The first edge is removed from the *EdgeList* and the terminal object of this edge is stored on the current page. If the current page is full[4] a new page is started and the next object is selected according to the presort order.

There are three heuristics that refine the *Traversal* component of sequence-based algorithms:

- *New page new traversal (NPNT):* This heuristic suggests to terminate the current traversal and start a new one if the current page is filled up.
- *New traversal new page (NTNP):* If the current traversal terminates because all objects that were accessible from the start objects have been clustered, a new page is set up to start the next traversal. The NTNP heuristic is beneficial if the CG is disconnected into several subgraphs.
- *Using undirected edges (UDE):* The edges of the CG are directed. Nevertheless, the direction of the edges is not important for clustering decisions and, therefore, should not be considered by the *Traversal* component.

We have intensively examined the impact of the three enhancements NPNT, NTNP and UDE on the performance of algorithms based on BSTF traversal. We have found, that in almost all cases the combination of all three heuristics achieves the best results. Interestingly, switching from directed to undirected edges is not beneficial for tree-like structures, i.e., if the degree of sharing[5] of the CG is below $5-10\%$ [Fah93].

A full algorithmic description of BSTF is given in the appendix. BSTF was the best *Traversal* algorithm in many comparison studys of clustering algorithms, e.g. [TN91, GKKM92]. Therefore, in the remainder of this paper, we investigate only BSTF supplemented by the NPNT, NTNP and UDE heuristics as the representative for sequence-based clustering.

[3]The *dynamic reference count* of an object equals the number of times the object is referenced by a sampling application.

[4]Actually, the clustering strategy of Cactis incorporates the NPNT heuristics that is described subsequently.

[5]The degree of sharing is the ratio of the number of shared objects to the total number of objects.

2.3 Partition-Based Mapping Algorithms

Partition-based clustering algorithms segment the CG utilizing graph partitioning algorithms. In the literature graph partitioning algorithms for clustering were only considered in the work of Tsangaris and Naughton [TN91, TN92], where the algorithm by Kernighan and Lin (KL) was applied [KL70]. The work of Tsangaris and Naughton indicates that the clustering results obtained by the KL algorithm are far superior to those obtained by sequence-based algorithms.

Although the work of Tsangaris and Naughton lead the way to partition-based clustering their results were based on benchmarks run on a rather small object base where all objects were of equal size. In practice there is a large diversity of object nets whose characteristics are essential for the performance and the cluster quality of partitioning algorithms:

- The size of the object net which may range from a few hundred to hundreds of thousands of objects.
- The size of the objects which may strongly vary.
- The degree of sharing: there are tree-like object nets and others which exhibit a highly connected network strucure.

It turns out that no one algorithm, e.g. KL, can be identified as superior in all (or even most of the) cases. Therefore, we investigate a wide range of partitioning algorithms for the clustering problem.

3 The Graph Partitioning Algorithms

According to [SM91] graph partitioning algorithms can be divided into two classes: *constructive partitioning* and *iterative improvement*. Constructive algorithms build a partitioning of a graph from scratch; iterative algorithms start with some initial partitioning and repeatedly try to improve this partitioning. Iterative algorithms usually produce better results than constructive algorithms, but have a very large running-time compared to constructive algorithms. However, as object bases may be of arbitrary size, speed is an important factor when selecting an algorithm for clustering.

In the remainder of this section we sketch four graph partitioning algorithms known from the literature. Further, we describe the basics of our new greedy heuristics for graph partitioning—more sophisticated heuristics based on greedy graph partitioning are developed in Section 4.

3.1 Iterative Algorithms

3.1.1 The Kernighan-Lin Heuristics

The Kernighan-Lin algorithm (KL) [KL70] was designed for the placement of VLSI chips—that is why KL is not well suited for clustering in object bases:

1. The asymptotic running-time of the KL algorithm is $O(n^{2.4})$ where n is the number of vertices of the graph. This indicates that the KL heuristics is prohibitively expensive for realistically large object bases.

2. The KL algorithm starts with an arbitrary partitioning of the CG and iterates over all pairs of partitions, trying to improve the partitioning by exchanging objects between the pair of partitions currently considered. Thus, the object sizes must not be strongly diverging for allowing objects to be swapped. While this may be true for VLSI applications, it is in general not true for object bases.

Presupposed that the objects are of similar size and the object base is small the partitioning results obtained by the KL algorithm are of high quality (in terms of external costs).

Subsequently, we discuss two iterative algorithm that are extensions of the KL heuristics.

3.1.2 The Fiduccia-Mattheyses Heuristics

The graph partitioning algorithm by Fiduccia and Mattheyses (FM) [FM82] is derived from the KL algorithm. The main modification is that objects are *moved* across partition borders instead of being *swapped*. Throughout the iteration process a minimum balance criterion for the partitions is maintained. The main advantage of the FM heuristics is that single objects can move across the cut. This modification allows for handling unbalanced partitions and non-uniform object sizes. On the other hand, swapping as employed in the KL heuristics, is advantageous in case of uniform size and nearly full pages—in this case no objects can be moved by the FM heuristics. Further, the FM heuristics is able to adapt the number of objects of the partitions (KL remains always stuck with the number of objects per partition that was initially chosen), and even the number of partitions (if all objects are removed from some partition, the partition can be deleted).

Fiduccia and Mattheyses propose a very efficient data structure for choosing the next object to be moved. Unfortunately, this data structure requires two one-dimensional arrays of size $2 * B_{\max} * D_{\max}$ where $B_{\max}$ is the maximum number of objects per page and $D_{\max}$ is the maximum vertex degree of CG. As there may be an arbitrary number of edges emanating from (or leading to) an object—e.g., think of set-structured objects—this running-time improvement of the FM heuristics cannot be employed for clustering in object bases. Thus, the complexity of the FM heuristics is about the complexity of KL—i.e., $O(n^{2.4})$.

3.1.3 Hierarchical Partitioning

The hierarchical partitioning algorithm (HP) is a combination of the KL and FM algorithms: KL is used to compute a good initial partitioning which is then improved using the FM heuristics.

The HP algorithm works as follows: All objects are assigned to one partition whose size is $P = 2^x * PageSize$ where x is chosen to be the smallest integer such that the total size of all objects is less than or equal to P. This partition is split such that the objects are distributed equally (by size) among both partitions. The resulting partitions are improved using the KL heuristics. Then, all partitions are split, KL is utilized again to improve the partitions, and so on. This method is recursively applied until the size of every partition is below the page size. After the termination of this process FM is applied to further improve this partitioning.

The idea that lead to the HP algorithm was taken from the work of Breuer [Bre77].

3.2 Constructive Algorithms

3.2.1 Optimum Partitioning of Trees

Lukes [Luk74] presents a pseudopolynominal-time algorithm for computing an optimum partitioning for the vertices of a tree, based on dynamic programming. The complexity of the algorithm is $O(n * B_{\max}^2)$ where n is the number of vertices of the tree and $B_{\max}$ is the maximum number of objects per page.

This algorithm can only be applied for those parts of the CG forming a tree.

3.2.2 The Greedy Graph Partitioning Heuristics

Because of the very good clustering results but poor running-time performance of known partition-based clustering algorithms, we have developed a new heuristics for graph partitioning, called *Greedy Graph Partitioning (GGP)*. (The GGP algorithm was first proposed in [GKKM92].) Graph partitioning is strongly related to subset optimization problems for which greedy algorithms often find good solutions very efficiently.

INPUT: The clustering graph CG;
OUTPUT: A list of partitions;
PartList := $\langle\ \rangle$;
Assign each object to a new partition and insert this partition into *PartList*;
Let *EdgeList* be a list of tuples of the form (o_1, o_2, w_{o_1,o_2})
    where w_{o_1,o_2} is the total weight of all edges between o_1 and o_2;
Sort *EdgeList* by descending weights;
foreach (o_1, o_2, w_{o_1,o_2}) **in** *EdgeList* **do begin**
  Let P_1, P_2 be the partitions containing objects o_1, o_2;
  if $P_1 \neq P_2$ **and** the total size of all objects of P_1 and P_2
          is less than the page size **then begin**
    Move all objects from P_2 to P_1;
    Remove P_2 from *PartList*;
  end if;
end foreach;

Figure 1: The Greedy Graph Partitioning Algorithm

The GGP algorithm (see Fig. 1) is based on a simple greedy heuristics that was derived from Kruskal's algorithm for computing the minimum weight spanning tree of a graph [Kru56]).

First, all partitions are inhabited by a single object only, and all partitions are inserted into the list *PartList*. For all objects o_1, o_2 connected by some edge in the CG with weight w_{o_1,o_2} a tuple (o_1, o_2, w_{o_1,o_2}) is inserted into the list *EdgeList*. All tuples of *EdgeList* are visited in the order of descending weights. Let (o_1, o_2, w_{o_1,o_2}) be the current tuple. Let

P_1, P_2 be the partitions to which the objects o_1 and o_2 are assigned. If $P_1 \neq P_2$ and if the total size of all objects assigned to P_1 and P_2 is less than the page size the two partitions are joined.[6] Otherwise, the edge is merely discarded—and the partitions remain invariant.

While Kruskal's algorithm yields the optimal solution to the minimum spanning tree problem, the GGP heuristics, of course, generates only a suboptimal solution to the partitioning problem.

Let e be the number of edges of the CG. The running-time complexity of the GGP algorithm is $O(e \log e)$—the dominating factor being the sorting of the list *EdgeList*.

4 New Heuristics for Greedy Graph Partitioning

We have extensively benchmarked the GGP heuristics on various forms of CG's comparing it to several clustering strategies for object-oriented database systems, e.g., the clustering algorithms of the database systems O_2 [Ben90] and Cactis [HK89], and the algorithms proposed in [TN91] and [CH91]. The results are presented in [GKKM92].

In these experiments the performance of GGP—both, in terms of running-time as well as clustering quality—was determined as quite good in comparison to the other algorithms. Nevertheless, there are situations in which GGP makes non-optimal choices when selecting two partitions that are to be joined. In this section we present two heuristics that improve the results of GGP:

1. *bounded look-ahead:* Before the partitions incident to the current edge, i.e., the edge with maximum weight, are being joined a BSTF traversal is performed locally in order to look for better candidate partitions to be joined.

2. *new-chance:* If the bounded look-ahead heuristics finds potentially better candidate partitions the current edge is rejected. Rejected edges need be re-considered (triggered) if the condition that lead to the rejection changes.

We identify the GGP algorithm that incorporates the above two heuristics by GGPla (*GGP with look-ahead*). Subsequently, both improvements—look-ahead and new-chance—that lead to the GGPla algorithm are discussed in more detail. The complete description of the algorithm GGPla can be found in the Appendix.

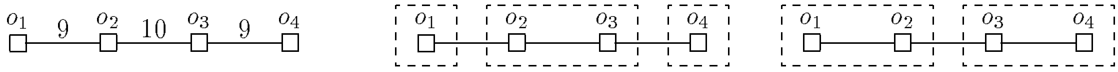
4.1 k-Look-Ahead

During each iteration the GGP algorithm takes the edge with maximum weight from the *EdgeList* and tries to join the partitions of the objects incident to that edge. Of course, this is not necessarily the best decision. Consider the example shown in Fig. 2. The CG is visualized in Fig. 2 (a); the maximum external costs are $EC_{\max} = 28$.[7] We assume uniform object sizes and a page capacity of 2 objects. Without look-ahead GGP decides to assign the objects o_2 and o_3 to the same partition resulting in a final partitioning with external costs of 18, visualized in Fig. 2 (b). Obviously, the optimum partitioning of this CG has only external costs of 10 (Fig. 2 (c)). The idea of the bounded look-ahead is to

[6] Partitions are represented as binary trees to accelerate the join operation.

[7] Squares denote objects, dashed boxes denote partitions.

detect situations where it is advantageous to *reject* the current edge, i.e., the edge with maximum weight, and to consider other edges first. Subsequently, this rough outline of bounded look-ahead is detailed.



(a) $EC_{\text{max}} = 28$ (b) $EC = 18$ (c) $EC = 10$

Figure 2: An Example for Non-Optimal GGP Clustering

Consider the CG depicted in Fig. 3. The objects o_l and o_r have been assigned to the partitions P_l and P_r. Besides the edge (o_l, o_r) there are n external edges incident to o_l with weights $w_{l_1}, \ldots, w_{l_n}$ and m external edges incident to o_r with weights $w_{r_1}, \ldots, w_{r_m}$. Assume the next edge to be considered by GGP is (o_l, o_r) with weight w, which implies $w \geq \max\{w_{l_1}, \ldots, w_{l_n}, w_{r_1}, \ldots, w_{r_m}\}$. Further assume that the partitions P_l and P_r could be joined because their total size being less than or equal to the page size.

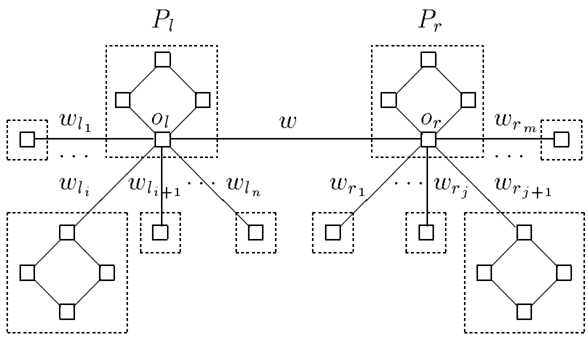


Figure 3: Clustering Graph Before Analyzing the Edge (o_l, o_r)

To decide whether it is beneficial to join the partitions P_l and P_r (or not) we proceed in two steps:

1. We evaluate the potential gain in the case P_l and P_r are being joined and the resulting partition is being filled up to the maximum size. For that, a BSTF traversal is performed on the external edges of the CG starting at the objects $\{o_l, o_r\}$. [8] The traversal is terminated if either (1) the depth of the traversal exceeds some integer bound k, or (2) the total size of all partitions encountered (including P_l and P_r) exceeds the page size. Let W_{join} equal w plus the total weight of all edges accessed during the traversal. Assume that these are the edges labeled $w_{l_1}, \ldots, w_{l_i}, w_{r_1}, \ldots, w_{r_j}$ (see Fig. 4); thus, $W_{\text{join}} = w + w_{l_1} + \ldots + w_{l_i} + w_{r_1} + \ldots + w_{r_j}$. W_{join} denotes the potential gain in the case that the partitions P_l and P_r are being joined and the resulting partition were filled up completely using a BSTF traversal starting at the objects $\{o_1, o_2\}$.

2. We evaluate the effect of not joining P_l and P_r. To do so, we perform two BSTF traversals (on the external edges of the CG) starting at o_l and o_r, respectively. The

[8]Note that the BSTF search inspects only paths that emanate from o_l and o_r—instead of inspecting all objects already in partitions P_l and P_r. Our experiments showed that the running-time becomes prohibitively large, otherwise.

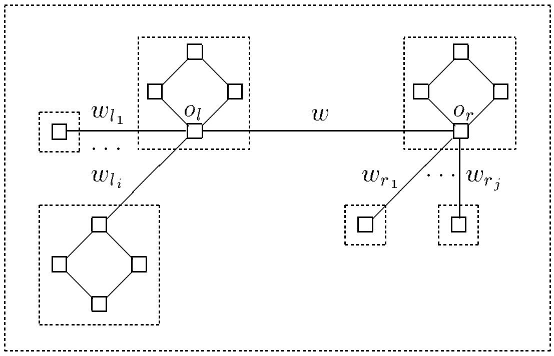


Figure 4: BSTF Look-Ahead Starting From Objects o_l and o_r

first traversal is called *left traversal*, the latter is called *right traversal*. The left (right) traversal is terminated as soon as the depth of the traversal exceeds k or when the total size of all partitions encountered including P_l (P_r) exceeds the page size. Let W_{left} denote the total weight of all edges traversed by the left traversal, and let $\mathcal{P}_{\text{left}}$ denote the set of partitions encountered during the left traversal (including P_l). W_{left} denotes the gain that can potentially be achieved by GGP on the left part of the CG if the partitions P_l and P_r are *not* joined in the current stage of GGP. W_{right} and $\mathcal{P}_{\text{right}}$ are defined analogously for the right traversal.

Based on the definitions made above we can define two heuristics H1 and H2:

Heuristics H1: The edge (o_l, o_r) is **rejected**, i.e., P_l and P_r are not joined in the current stage of GGP, if either

- $W_{\text{left}} > W_{\text{join}}$ and the total size of all objects on pages in $\mathcal{P}_{\text{left}} \cup \{P_r\}$ exceeds the page size, or
- $W_{\text{right}} > W_{\text{join}}$ and the total size of all objects on pages in $\mathcal{P}_{\text{right}} \cup \{P_l\}$ exceeds the page size.

Heuristics H1 rejects the edge (o_l, o_r) if either the gain on the left part of the CG (rooted at o_l) or on the right part of the CG (rooted at o_r) is larger than the gain achieved by joining the partitions P_l and P_r.
Note, however, that by this heuristics the optimal partitioning of the CG shown in Fig. 2 is not found. This observation leads to the definition of

Heuristics H2: The edge (o_l, o_r) is **rejected**, i.e., P_l and P_r are not joined in the current iteration of GGP, if

- $W_{\text{left}} + W_{\text{right}} > W_{\text{join}}$ and the total size of all objects on pages in $\mathcal{P}_{\text{left}} \cup \mathcal{P}_{\text{right}}$ exceeds the page size.

Under heuristics H2, the total gain that can be potentially achieved by not joining the partitions P_l and P_r is compared to the gain potentially achieved by joining P_l and P_r.

It is easy to see that the condition of H2 is always true if the condition of H1 is fulfilled—thus, H2 is a strengthening of H1.

A similar heuristics that introduces more look-ahead into the Fiduccia-Mattheyses algorithm was proposed by Krishnamurthy [Kri84].

4.2 New-Chance

In this section we discuss the "fate" of edges that were rejected by the k-look-ahead heuristics. First, we observe that rejected edges must not be removed from the *EdgeList* because they should be re-considered before termination of GGP. For example, in the CG visualized in Fig. 5 (a), let us assume that the edge (o_1, o_2) is rejected by the k-look-ahead heuristics H1 ($k > 2$). Obviously, because (o_1, o_2) may be the only (external) edge incident to the partition P_2 this edge should be re-considered in order to retry the join of P_1 and P_2. Technically, in our implementation, edges are given a new chance by moving them to the end of the *EdgeList* when they are rejected (see Fig. 5 (b)).

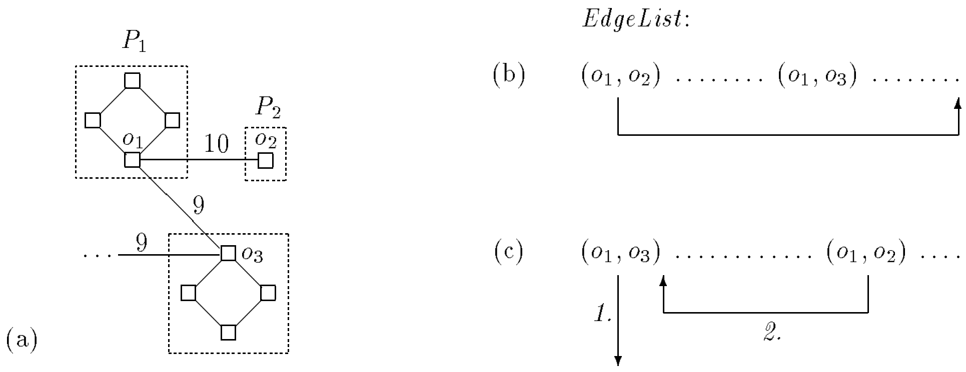


Figure 5: Triggering Rejected Edges

The set of the external edges which has been visited in the k-look-ahead for some rejected edge e_r is called the *trigger-set* of e_r. For example, in Fig. 5, the trigger-set of (o_1, o_2) is $\{(o_1, o_3), \cdots\}$. Now let us assume that some edge e is considered and is not being rejected during the current iteration of GGP. Further assume that e is in the trigger-set of the rejected edge e_r. In this case the edge e_r is said to be *triggered*. After the join of the partitions incident to e the local situation in the CG that was responsible for rejecting the edge e_r has changed. In this new situation the edge e_r would possibly survive (i.e., not being rejected). Thus, we propose to retry any rejected edge when it is triggered.[9]

Triggered edges are moved to the first position of the *EdgeList* as their weight is greater than or equal to the maximum of the weights of all edges that have not yet been considered. For example, in the CG visualized in Fig. 5 (a), the rejected edge (o_1, o_2) is moved to the front of the *EdgeList* when it is triggered by the edge (o_1, o_3) (see Fig. 5 (c)).

4.3 Discussion

In our experiments, we have found that the results of the GGPla algorithm using the bounded look-ahead and the new-chance heuristics are better than those of the basic GGP algorithm. Especially, if the degree of sharing of the CG is small, i.e., the CG has a tree-like structure, GGPla performs much better than GGP. This issue is discussed in more detail in Section 5.

[9]To avoid exhaustive searching we employ special data structures facilitating the direct access from the current edge to the set of triggered edges.

Of course there are many further heuristics that could be applied to the basic GGP algorithm. In fact, we have experimented with lots of heuristics that were very attractive at first glance but did not produce the expected results. Some of these heuristics are:

- *sorting the edges by the descending ratio of edge weights to partition sizes.* This heuristics was applied to the global *EdgeList* and to the BSTF traversal of the bounded look-ahead heuristics. In the first case it lead to a tremendous increase in running-time because of the necessary resorting of the *EdgeList* after every iteration. In the latter case, the partitioning results became worse.

- *unification of all external edges incident to a pair of partitions to one edge with accumulation of the edge weights.* This heuristics requires the global *EdgeList* to be resorted after each iteration and, therefore, leads to a tremendous increase in running-time.

In our experience other improvements of the basic GGP heuristics we have tried always lead to a drastic prolongation of the running-time—preventing them from being applied to large object bases.

5 Quantitative Analysis

In this section we investigate the quality of clustering algorithms (1) in terms of quality, i.e., external costs and (2) in terms of running-time. In the related work clustering algorithms are often evaluated by running specific benchmark applications and measuring the miss ratio of the cache or the average working set size. We decided to use the external costs of the partitioning to evaluate the quality of the clustering because this measure does not depend on the choice of the benchmark applications, the buffer size, the replacement policy, nor the working set window size.

The experiments were carried out using a graph generator—that is part of our simulation workbench TEXAS.[10] The graphs used in the experiments consist of 10 *modules*, each constituting a highly connected subgraph. The modules are tree-structured with five successors per inner node. Dependent on the degree of sharing further edges between randomly selected objects of the same module are inserted into the graph. These edges are called *additional edges*. For example, a degree of sharing of 50 % with 2 additional edges per shared object denotes that 50 % of all objects are shared, and that each shared object is hit by 3 references (2 additional edges and 1 edge of the original tree). Every module is connected to one randomly selected module by an edge with minimal weight (1). The weights of the intra-module edges are uniformly distributed in the range $2, \ldots, 20$. The average object size in all experiments was 300 bytes; the standard deviation of the object sizes varied and is specified separately for each experiment. The partition (page) size was varied in many of the experiments. The cardinality of the entire database was 1000 objects unless stated otherwise.

We classify the characteristics of the examined clustering graphs according to five dimensions:

[10] The EXtensible Access Simulator for object bases.

database cardinality	average number of objects per page	deviation of object sizes	degree of sharing	static reference count (SRC)

The cardinality of the database is the dominant factor of the running-time of the algorithms, whereas the remaining four parameters influence the quality of the partitionings generated by the algorithms. The degree of sharing is stated in % and determines the ratio of the number of shared objects to the total number of objects. The static reference count of an object equals the number of references referring to ("hitting") the object. Thus, the SRC of shared objects is larger than 1, and the SRC of non-shared objects is 0 or 1. We conducted experiments on many combinations of the five parameters.

The following algorithms were measured:

Algorithms	*Remarks*
BSTF, GGP, HP	BSTF with NPNT/NTNP and UDE
GGPla	Heuristics H1, H2, new-chance of rejected edges
Lukes	approximated, see below
KL(x), FM(x)	with pre-partitioning produced by x, for $x \in \{$ GGPla, BSTF $\}$

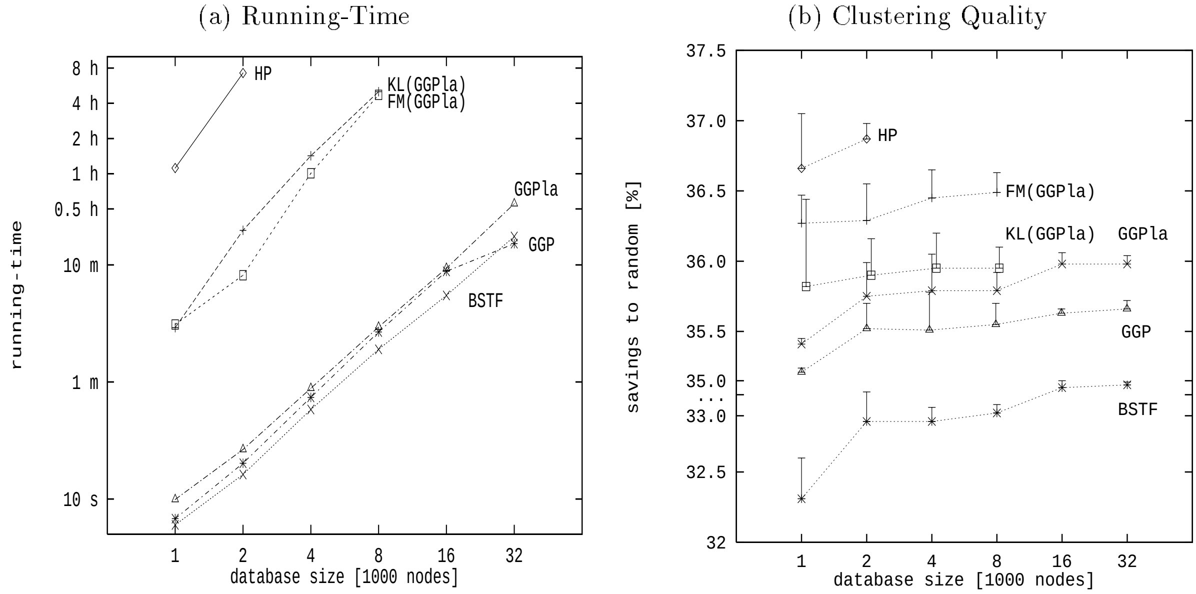
The (iterative) algorithms KL(x) and FM(x) were run until no further improvement was achieved (or maxint was reached). To compensate for one major disadvantage of Lukes' algorithm (producing memory overflow when working on high trees) we limited the keeping of the best assignments of objects to partitions only up to the average number of objects per page.

5.1 Running-Time vs. Quality of Cluster Algorithms

The first experiment evaluates the cluster algorithms BSTF, GGP, GGPla, FM(GGPla), KL(GGPla), and HP in terms of (1) their partitioning quality—that is, we compute the relative savings in external costs in relation to a random placement—and (2) their running-time dependent on the database cardinality.

The algorithms worked on a scalable database of n modules—each constituting a random graph with 1000 objects and 4 edges per object on the average. The object sizes have a standard deviation of 300 bytes with mean value 300; the average page capacity is 15 objects. Fig. 6 visualizes the results where the database size is plotted against the x-axis. In Fig. 6(a) the running-time is plotted against the y-axis, in Fig. 6(b) the relative savings of the obtained clustering to a random placement is plotted against the y-axis. The experiment was run on several randomly chosen graphs; the vertical (error) bars in the plots of Fig. 6 (b) visualize the standard deviation of the clustering results for the different graphs.

The most important result of this experiment is that the running-times of the algorithms are drastically different. Whereas the constructive algorithms GGPla and GGP, and the sequenced-based technique BSTF are very fast (30 minutes for 32000 objects), the running-time of the iterative algorithms KL(GGPla), FM(GGPla), and HP is not acceptable for large databases. We limited the maximal running-time to 24 hours; HP exceeded this limit at a database cardinality of 4000 objects, and KL/FM exceeded this limit at a database cardinality of 16000 objects. Thus, HP is only feasible for small databases up to 2000 objects and KL/FM may be feasible for databases up to 8000 objects—if the long reorganization time is tolerable at all.



database cardinality	average number of objects per page	deviation of object sizes	degree of sharing	static reference count (SRC)
variable	15	300	100%	4

Figure 6: Performance of Clustering Algorithms Dependent on the Database Size

The clustering results of all algorithms range between 32% to 37% savings to random placement. Note, that even *small* differences in relative savings exhibit enormous differences in absolute external costs—e.g., 40273 between BSTF and GGPla at a database cardinality of 32000 objects. Even though, these absolute savings are in terms of external costs of the partitioning they ultimately lead to a proportional saving in page faults—as our experiments (not reported here) indicate.

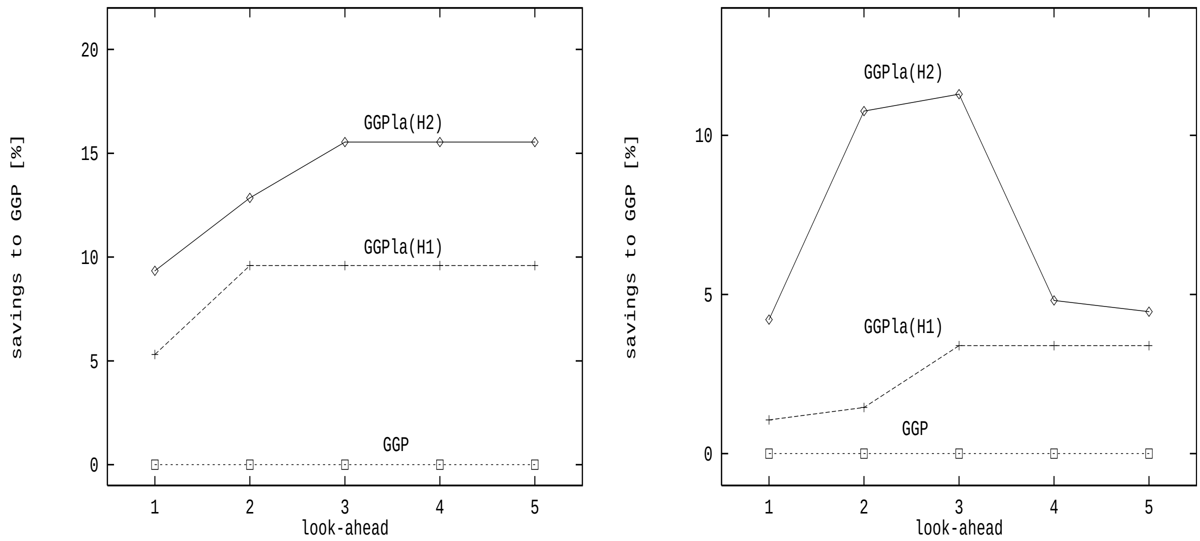
The iterative algorithms HP, KL(GGPla), and FM(GGPla) exhibited the best partitioning results. Nevertheless, the results of the KL and FM heuristics strongly depend on the quality of the pre-partitioning. Utilizing a poor-quality pre-partitioning KL and FM cannot achieve such good results. Among the algorithms whose running-time is reasonably low GGP and GGPla performed best. The relative ordering of the algorithms in both aspects, i.e., in running-time as well as in their quality of clustering results, that was observed in this experiment remains stable in the majority of the experiments we conducted—as indicated by the error bars in Fig. 6 (b) which represent standard deviation.

5.2 Testing the New Heuristics

From the first experiment we can conclude that the GGP heuristics represents an interesting alternative to the computationally complex iterative algorithms. In this experiment we focus on the impact of the *look-ahead* heuristics H1 and H2 combined with the *new-chance* heuristics (see Section 4) on the clustering results of GGP. This experiment was run on a graph with varying object sizes, an SRC of 2 for shared objects, and an average page capacity of 20 objects. The results are visualized in Fig. 7 (a) for a sharing of 0 % and Fig. 7 (b) for a sharing of 20 %. The depth k of the look-ahead is plotted against the

x-axis, the relative savings of GGPla to GGP is plotted against the y-axis.

(a) Degree of Sharing 0 % (b) Degree of Sharing 20 %



	database cardinality	average number of objects per page	deviation of object sizes	degree of sharing	static reference count (SRC)
(a)	1000	20	300	0%	0
(b)	1000	20	300	20%	2

Figure 7: Performance of the New Heuristics

Both *look-ahead* heuristics H1 and H2 improve the performance of GGP. The version of GGPla using the H2 heuristics clearly outperforms the version using the heuristics H1—indicating that H2 actually refines H1.

In the plots of Fig. 7 (a) where sharing is 0 the maximal improvement by H2 is 15 %.[11] Thus, the heuristics work very well on trees.

In the plots of Fig. 7 (b) where 20 % of the objects are shared (1) the overall improvement is lower compared to trees and (2) the savings to GGP *decreases* if k exceeds 3. The reason for this behavior lies in the sharing of objects: the BSTF traversal employed by the look-ahead heuristics tries to pull objects into the current partition—tearing them away from other partitions. Thus, if k increases the BSTF traversal gains more influence and—because of that—may cause more damage (generally, GGP is a better heuristics than BSTF when working on shared objects). We have found that a good estimate for the depth k of the look-ahead is given by

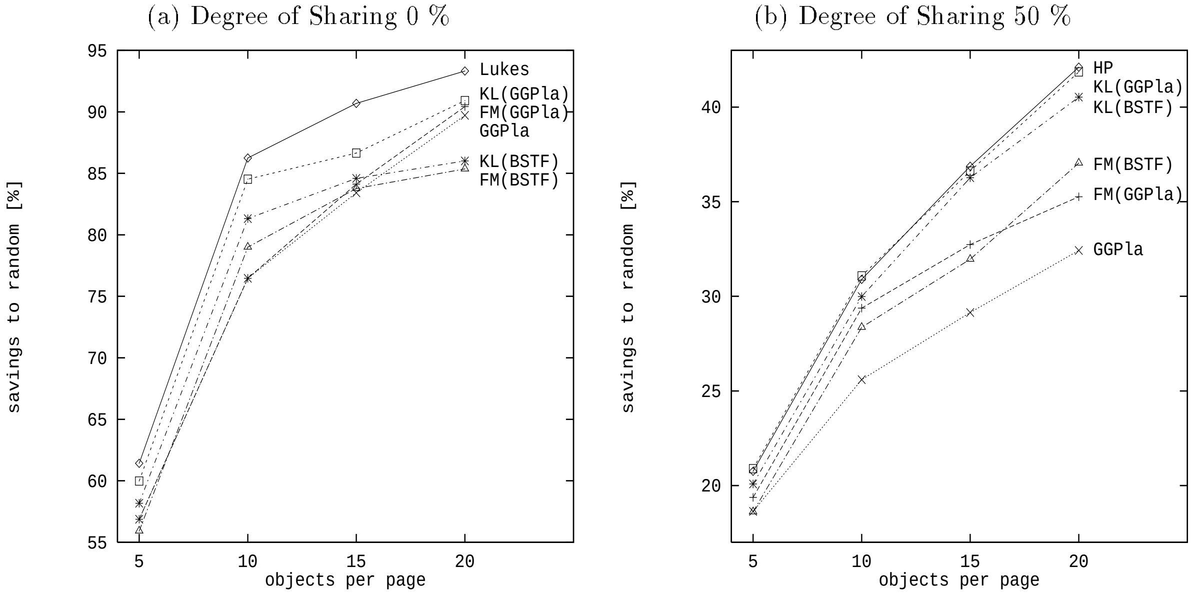
$$k \;=\; \left\lfloor (1 - \frac{S}{100}) * \frac{N}{R} \right\rfloor + 1$$

where S is the degree of sharing in %, N is the average number of objects per page, and R is the average number of references emanating from an object.

[11] In individual cases we even observed savings of more than 20%.

5.3 Changing the Partition Size

The number of objects per page has an important influence on the clustering results. Therefore, the page size is an effective tuning parameter in physical database design. However, if the page capacity is one object per page or if all objects can be stored on one page any clustering strategy is optimal. In the subsequent experiment we measure the performance of the clustering algorithms Lukes, HP, KL, FM, and GGPla under varying page sizes. Note, that varying the average object size instead of the page size would have the analogous effect. Fig. 8 (a) visualizes the results for a tree-structured graph (sharing = 0) and Fig. 8 (b) visualizes the result for a graph with 50 % shared objects and an SRC of 11 for shared objects. All objects were of equal size. Thus, in this case our approximation of Lukes' algorithm is optimal.



	database cardinality	average number of objects per page	deviation of object sizes	degree of sharing	static reference count (SRC)
(a)	1000	variable	0	0%	0
(b)	1000	variable	0	50%	11

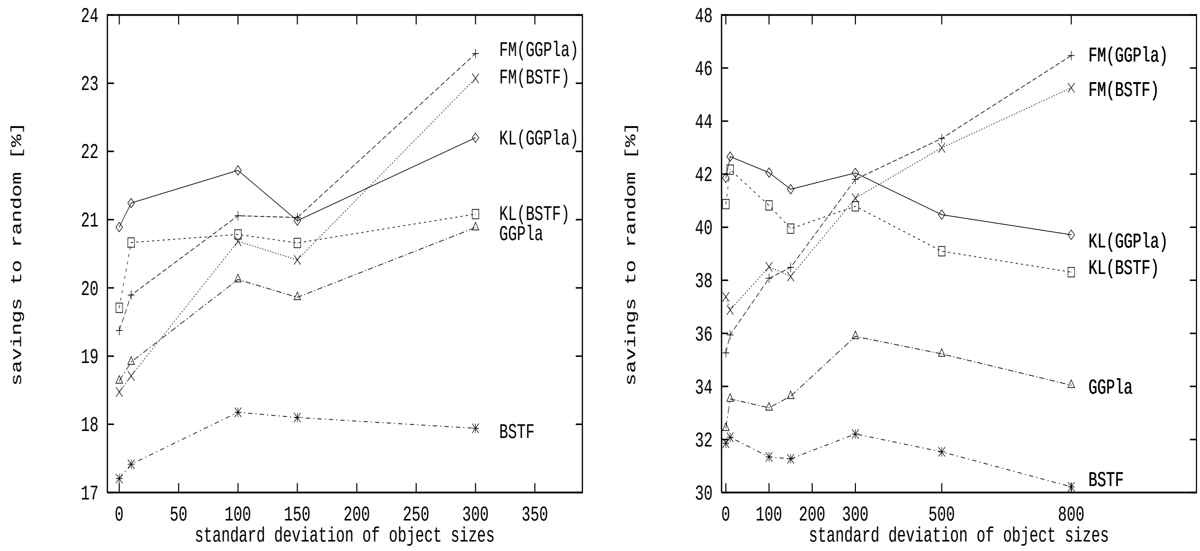
Figure 8: Influence of the Page Size on the Clustering Results

Lukes' algorithm achieves 93 % savings compared to random placement. The plot of the results of KL(GGPla) runs very close to this optimum—indicating that KL with the high quality pre-partitioning of GGPla is a good candidate for clustering trees of moderate size. However, as the experiments in Section 5.4 demonstrate, KL achieves such good result only with uniform object sizes—under varying object sizes FM achieves better results.

The clustering results on the graph—plotted in Fig. 8 (b)—range from 20 % to 40 % savings to random. Thus, if the degree of sharing is high and the page size is small, only small improvements can be achieved by clustering. With an increasing page size, the potential improvement by clustering increases, too. The plots of Fig. 8 (b) indicate that KL outperforms FM—due to the uniform object sizes in this experiment—independent of the pre-partitioning. This issue is examined in more detail in the subsequent experiment.

5.4 Varying Object Sizes

Usually, the objects' sizes vary significantly—note, that the object net is composed of objects of many different types. E.g., there may be tuple-structured objects of just 50 bytes, whereas the size of set- or list-structured objects easily may even reach the page size.[12] In this experiment we investigate the impact of varying object sizes on clustering of a graph with sharing of 50 % and an SRC of 11 for shared objects. Fig. 9 (a) visualizes the results for a page capacity of 5 objects, and Fig. 9 (b) visualizes the results for a page size of 20 objects.



	database cardinality	average number of objects per page	deviation of object sizes	degree of sharing	static reference count (SRC)
(a)	1000	5	variable	50%	11
(b)	1000	20	variable	50%	11

Figure 9: Performance of Clustering Algorithms Under Varying Object Sizes

The main observation from the plots of the two diagrams is that the relative saving increase upon increasing the deviation of object sizes.

The second result of this experiment is that FM performs better than KL if the object sizes vary substantially. Below a page capacity of 20 objects the break even point between KL and FM is at a standard deviation of object sizes of about 300 (see Fig. 9 (b)). If the page capacity is only 5 objects per page on the average, the break even point is moved to a standard deviation of 150 (witness Fig. 9 (a)). To understand this result, consider Fig. 10. Neither KL nor FM can improve the partitioning of the graph depicted in Fig. 10 (a), where strong internal edges prevent from moving or swapping objects between the two partitions. In the graphs depicted in Fig. 10 (b) and (c) FM can improve the partitioning by moving a single object from one partition to the other. However, KL cannot improve the partitioning as there are no pairs of objects to be swapped. Obviously, if the number

[12]Objects larger than the page size are of no interest for clustering.

of objects decreases the probability of finding clustering subgraphs similar to those in Fig. 10 (b) and (c) increases—leading to better results of FM relative to KL.

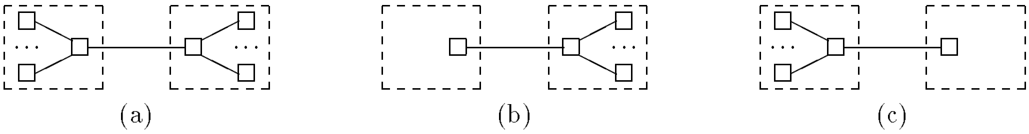
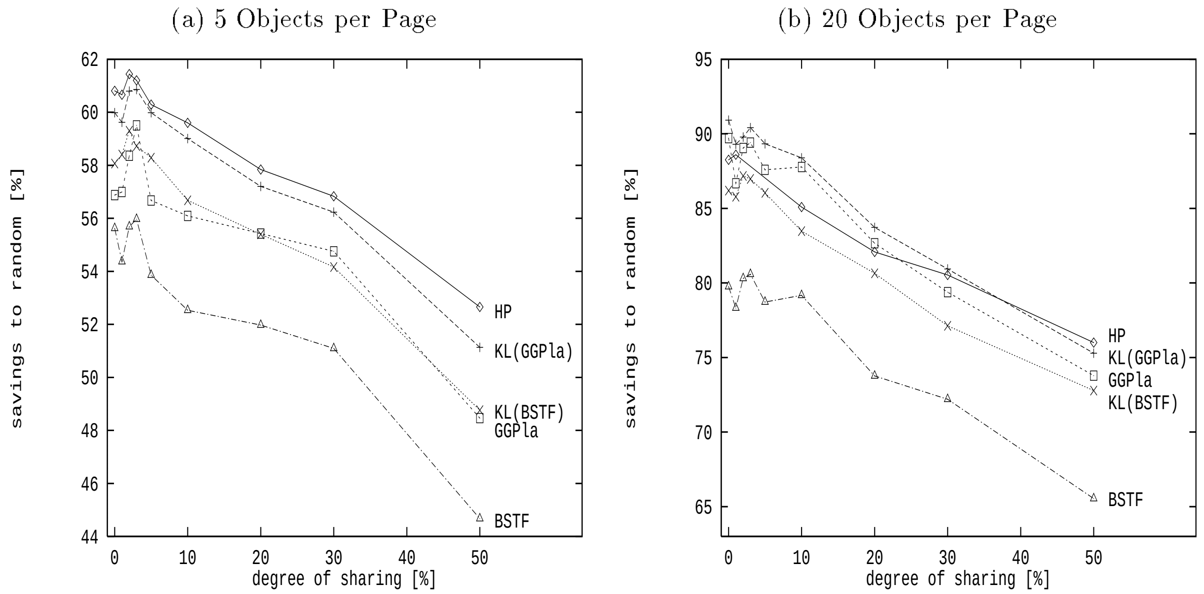


Figure 10: Three Partitioning Problems

5.5 Degree of Sharing

The experiment in Section 5.3 already indicated that object sharing makes clustering more difficult. The next experiment is designed to investigate the influence of the degree of sharing on the clustering results. We measure the algorithms HP, KL(GGPla), KL(BSTF), GGPla, and BSTF on a graph with uniform object sizes[13] and an SRC of 2 for shared objects. Fig. 11 (a) visualizes the results for a page capacity of 5, Fig. 11 (b) for a page capacity of 20 objects per page on the average. The degree of sharing varies from 0 % to 50 % (plotted against the x-axis).



	database cardinality	average number of objects per page	deviation of object sizes	degree of sharing	static reference count (SRC)
(a)	1000	5	0	variable	2
(b)	1000	20	0	variable	2

Figure 11: Performance of Clustering Algorithms Under Varying Degrees of Sharing

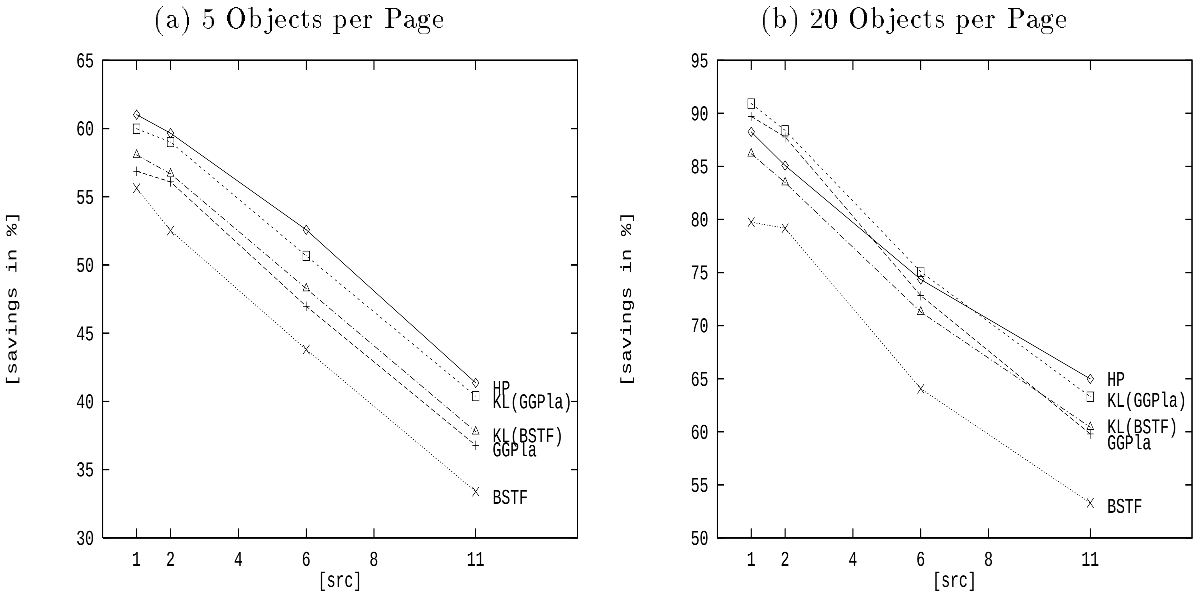
[13]The object sizes were chosen to be equal in order to omit the influence of varying object sizes. As indicated by the preceding experiments FM achieves only moderate clustering results for uniform object sizes and, therefore, has not been measured in this experiment.

As expected, the plots of all algorithms show a steep decline as the degree of sharing increases. With 5 objects per page on the average the savings relative to random placement drop by about 10 %, with 20 objects per page even by 15 %.

In both experiments, HP and KL(GGPla) yield the best clustering results among the algorithms tested.

5.6 Static reference count

This experiment investigates the impact of the variation of the static reference count on the clustering results. We measured the same algorithms as in the preceding experiment — HP, KL(GGPla), KL(BSTF), GGPla, and BSTF — on a graph with uniform object sizes. The degree of sharing was 10 %; the SRC varied from 0 to 11. As in the preceding experiment the average page capacity was 5 object per page and 20 objects per page. The results are visualized in Fig. 12 (a) and (b).



	database cardinality	average number of objects per page	deviation of object sizes	degree of sharing	static reference count (SRC)
(a)	1000	5	0	10	variable
(b)	1000	20	0	10	variable

Figure 12: Performance of Clustering Algorithms Under Varying Static Reference Counts

Both plots show a deep decline upon the increase of the SRC. Thus, with increasing connectivity of the graph the clustering results become worse.

Upon a page capacity of 5 objects per page the algorithm HP performs best independently of the value of SRC (see Fig. 12 (a)). Upon a page capacity of 20 objects per page (Fig. 12 (b)) there is a break even point between the plots of HP and KL(GGPla)/GGPla at a SRC of about 6. Thus, for larger page capacities, KL(GGPla) and GGPla outperform HP if the SRC is small. This result is similar to the results of the preceding experiment, where GGPla and KL(GGPla) outperform HP for large page capacities and a small degree of sharing (rf. Fig. 11 (b)). From these two experiments we can draw two conclusions about the clustering results of the algorithms HP, KL(GGPla), and GGPla depending on the connectivity of the graph, i.e., the SRC and the degree of sharing, and the average page capacity:

1. If the average page capacity is small (e.g., 5 objects per page), HP shows the best results indepently of the connectivity of the graph.
2. Upon larger page capacities (e.g., 20 objects per page), KL(GGPla) and GGPla perform better than HP as long as the connectivity of the graph is low. In particular, in the above experiments, KL(GGPla) shows better results than HP for a degree of sharing below 40 % (SRC = 2) and for a SRC below 7 (degree of sharing = 10 %).

For large pages, i.e., a page capacity over 20 objects per page, the plots of GGPla and KL(GGPla) in the Figures 12 (b) and 11 (b) run very close. In these cases the KL-heuristics does not improve the initial GGPla-clustering significantly. Therefore, GGPla should be used without KL (or FM) for large pages and small connectivity of the graph.

The variation of the SRC (in this experiment) and the variation of the degree of sharing (in the preceding experiment) indicate the following correlation between graph connectivity and clustering results: Both parameters, the SRC and the degree of sharing, are measures for the connectivity of the graph. The results of all clustering algorithms measured in this paper show a large decline upon an increase of the graph connectivity — either by increasing the SRC or the degree of sharing or both.

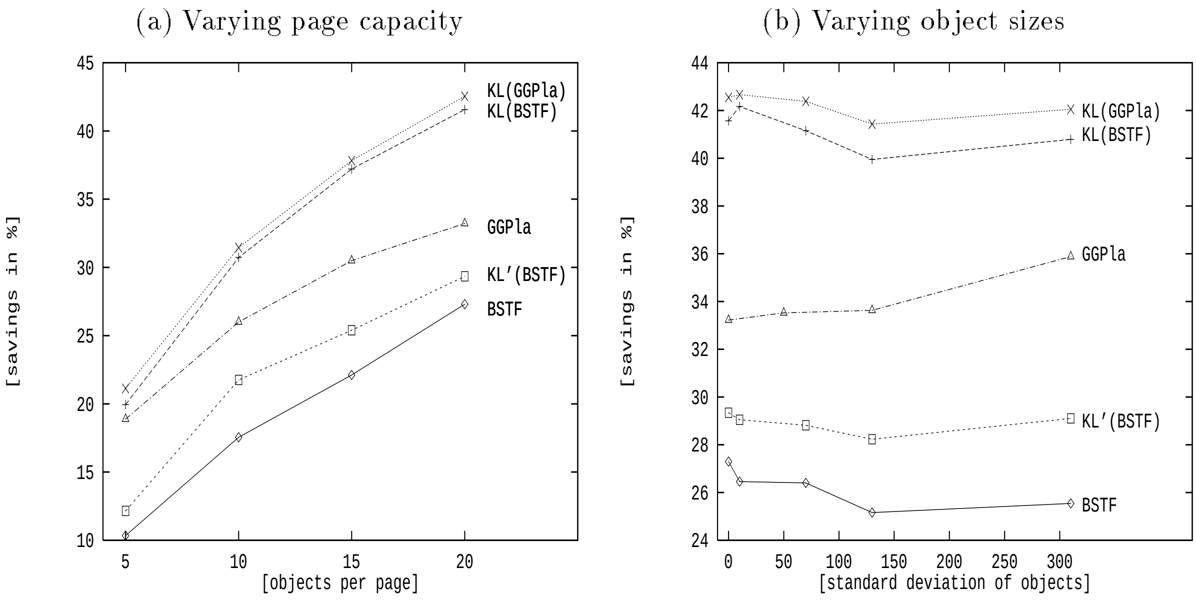
5.7 Time Limit for KL(BSTF)

The experiments in Section 5.1 have shown that the algorithm GGPla performs better than BSTF in terms of external costs, but GGPla needs a longer running time than BSTF. On the other hand using the KL heuristics to improve an initial partitioning constructed by BSTF achieves better results than GGPla. This observation leads to the idea to measure the performance of KL(BSTF) vs. GGPla under the constraint that both algorithms use (approximately) the same amount of time. I.e., in this experiment, we use the time saved by BSTF compared to GGPla to improve the BSTF-partitioning employing the KL heuristics.

In the preceding experiments, KL(BSTF) terminated after no further improvement of the current partitioning could be achieved. In this experiment, we first conducted the GGPla algorithm and measured its running time. Then, we used a modified version of KL(BSTF) — called KL′(BSTF) — which terminates after the current iteration when the running time of GGPla is reached.

Fig. 13 (a) visualizes the results under a varying page capacity and uniform object sizes. Fig. 13 (b) visualizes the results under varying standard deviation of object sizes and an average page capacity of 20 objects per page. In both plots the results of GGPla are better than those obtained from KL′(BSTF).

Note, that the time consumed by the original KL(BSTF) algorithm — with no time constraint — exceeded the running time of GGPla by a factor of 20.



	database cardinality	average number of objects per page	deviation of object sizes	degree of sharing	static reference count (SRC)
(a)	1000	variable	0	50	11
(b)	1000	20	variable	50	10

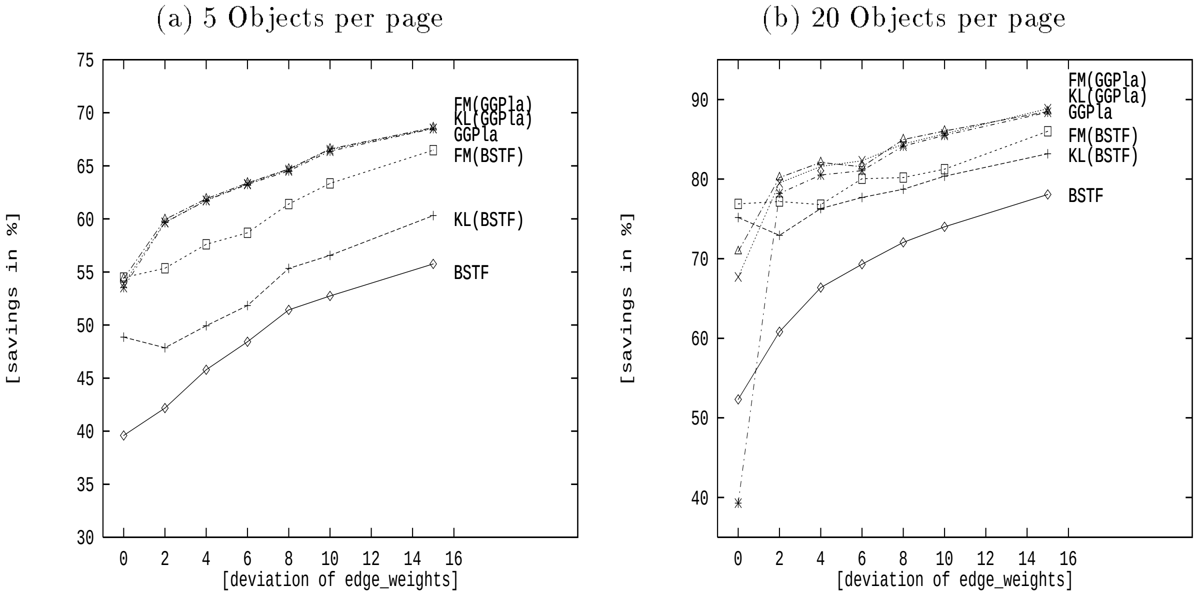
Figure 13: Performance of KL(BSTF) with time limit

5.8 Changing the Edge Weights

In all preceding experiments the application profiles, i.e., the edge weights, were uniformly distributed between 2 and 20. In this experiment, we wanted to investigate the effect of changing the application profile on the clustering results. The edge weights were determined by a normal distribution with mean value 20 and varying standard deviations (see Fig. 14). All edge weights were non-negative (negative numbers produced by the random number generator were discarded). The table in Fig. 14 visualizes the remaining parameters of this experiment. The plots in Fig. 14 (a) shows the results using an average page capacity of 5 objects and the plots in Fig. 14 (b) show the results using an average page capacity of 20 objects per page.

All algorithms perform better if the deviation of the edge weights increases. The average improvement of a standard deviation of 15 compared to a standard deviation of 0 is 10 % (5 objects per page) and 50 % (20 objects per page), respectively. Because the algorithm GGPla is based on sorting the edge list GGPla performs worst if all edges have the same weight, i.e., the standard deviation is 0.

Both figures show that GGPla performs better than KL(BSTF) and FM(BSTF) if the standard deviation of the edge weights is greater than 2. Using KL or FM to improve the initial partitioning obtained by GGPla is only beneficial if the standard deviation of the edge weights is less than 10; for larger standard deviations the reduction of the external costs achieved by KL(GGPla) and FM(GGPla) compared to GGPla remains under 1 %. Contrary to GGPla the partitioning computed by BSTF is improved using KL or FM — even under a large deviation of the edge weights.



	database cardinality	average number of objects per page	deviation of object sizes	degree of sharing	static reference count (SRC)
(a)	1000	5	300	10	2
(b)	1000	20	300	10	2

Figure 14: Performance of Clustering Algorithms Under Varying Edge Weights

The results of FM are — in this experiment — better than the results of KL because of the large standard deviation of the object sizes (300).

6 The Case for Adaptable Algorithms

The results of the quantitative analysis presented in the previous section lead to the conclusion that no *one* algorithm performs superior for *all* object base configurations. Therefore, we derive a multi-dimensional grid: the dimensions correspond to particular characteristics of the object base configurations and the grid entries determine the best clustering algorithm for the particular configuration. Thus, the adaptable cluster strategy consists of determining the characteristics of the object base and then applying the best algorithm which is obtained from the multi-dimensional grid.

The grid is shown in Fig. 15. Three dimensions are visualized by the axes of the grid, i.e., the average number of objects per page, the standard deviation of the object sizes, and the degree of sharing. The fourth dimension, i.e., the size of the database in number of objects, is visualized by the coloring of the grid entries shown on the right hand side. Here, we distinguish three configurations: (1) a database with less than 2000 objects, (2) a database with more than 2000 and less than 8000 objects, and (3) a database with more than 8000 objects. For each configuration, the coloring (shading) of the grid indicates the most appropriate clustering algorithm. Subsequently, we briefly discuss the entries of the grid.

Let us start with the left hand side of the grid—representing a sharing of 0%. In this

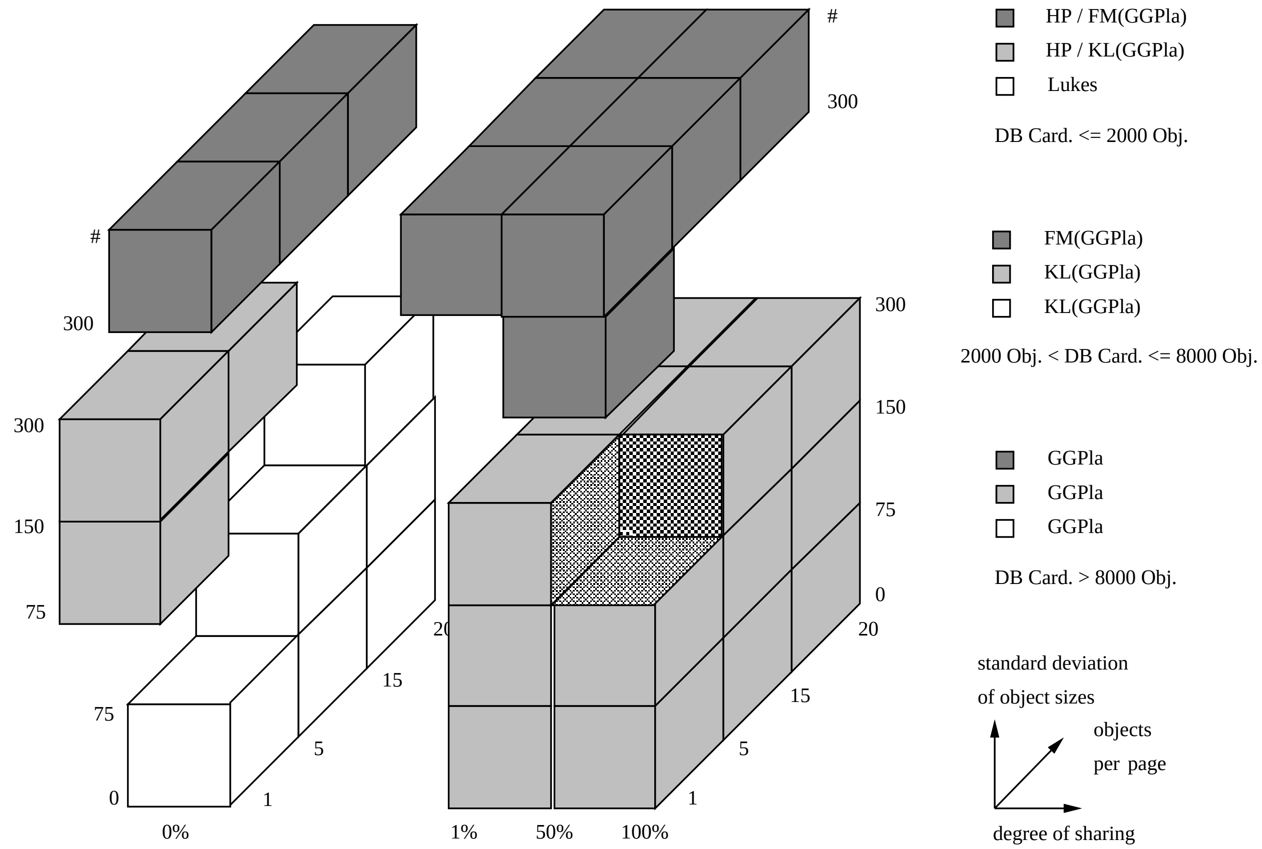


Figure 15: Selecting the Best Clustering Algorithm

case, the approximation of Lukes' algorithm is optimal if all objects are of equal size. Our experiments further indicate that the approximation of Lukes' algorithm yields good results as long as the ratio of the standard deviation of the object size to the average number of objects per page is small (this area is colored white in the grid). When this ratio increases other algorithms perform better than the approximation of Lukes, e.g., HP.

Lukes' algorithm needs a tremendous amount of main memory, especially if the height of the tree becomes large. We have found that beyond a database cardinality of 2000 objects Lukes' algorithm ran out of memory. That is why we propose to use KL/FM(GGPla) for medium cardinality and GGPla for large cardinality object bases even if the CG forms a tree.

Next we investigate the right hand side of the grid, where the degree of sharing is greater than 0. If the database is of low or medium cardinality, i.e., below 8000 objects, KL(GGPla) or FM(GGPla) are the algorithms of choice:

- Because KL is based on swapping objects, it is superior to FM if the standard deviation of the object size is small.
- Beyond a standard deviation of around 300, FM achieves better clustering results.

For a database cardinality below 2000 objects HP can be used as an alternative to KL/FM (in the majority of the experiments we conducted HP produced the best results). However, HP cannot possibly be applied to databases with more than 2000 objects because of its high running-time.

Consider the upper right hand corner of the grid, where sharing is high, the number of objects per page is small, and the deviation of the object size is between 150 and 300. Here, the FM heuristics is superior to KL because in this special case, moving objects achieves better results than swapping objects (this is explained in detail in Section 5.4).

Beyond a database cardinality of 8000 objects, the GGPla heuristics is the only method yielding acceptable cluster quality under reasonably low running-times. Therefore, GGPla is the method of choice for large databases.

7 Conclusion

Reconsider the title of this paper "Partition-Based Clustering in Object Bases: From Theory to Practice". The theory of partition-based clustering was developed by Tsangaris and Naughton [TN91]. They, in particular, proved the superiority of partition-based techniques over the sequence-based approaches.

However, the hitherto known partition-based cluster approaches had to rely on the known computationally rather complex iterative partitioning algorithms, e.g., the Kernighan and Lin heuristics. Therefore, partition-based clustering was impractical for realistically large databases—e.g., an object base exceeding 8000 objects.

Our greedy graph partitioning heuristics GGPla supplemented with the *look-ahead* and *new-chance* of once rejected choices has a much lower running-time complexity than the iterative algorithms. Nevertheless, the partitioning quality generated by GGPla is quite good compared to the computationally complex iterative partitioning heuristics and far better than sequence-based clustering algorithms. This makes the GGPla heuristics indispensable for two purposes:

1. As a pre-partitioning heuristics for small to medium cardinality object nets—on which iterative algorithms are then applied for improvement.

2. As the only applicable method on large cardinality object nets.

In conclusion, we therefore propose an adaptable clustering strategy which applies the most appropriate iterative partitioning method—in conjunction with a pre-partitioning by GGPla—on small to medium cardinality object nets and (pure) GGPla on large cardinality object nets. The adaptability is controlled by a multi-dimensional grid whose dimensions correspond to the object net characteristics which have to be established by, e.g., sampling the object base extension. The entries of the grid then determine the most appropriate cluster algorithm for the particular object base configuration.

Acknowledgement This work was partially supported by the German Research Council DFG under contracts Ke 401/6-1 and SFB 346.

We thank S. Fahrig for helping in carrying out the implementation and extensive benchmarking of the various partitioning algorithms.

References

[Bar82] E. R. Barnes. An algorithm for partitioning the nodes of a graph. *SIAM Journal of Algebraic and Discrete Mathematics*, 3(4):541–550, 1982.

[Ben90] V. Benzaken. An evaluation model for clustering strategies in the O_2 object-oriented database system. In *Proc. of the Int. Conf. on Database Theory (ICDT)*, pages 126–140. Springer-Verlag, 1990.

[BKKG88] J. Banerjee, W. Kim, S. J. Kim, and J. F. Garza. Clustering a DAG for CAD databases. *IEEE Trans. Software Eng.*, 14(11):1684–1699, Nov 1988.

[Bre77] M. A. Breuer. A class of min-cut placement algorithms. In *Proc. of the Design Automation Conference*, pages 284–290, 1977.

[CDRS86] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 91–100, Kyoto, Japan, Aug 86.

[CH91] J. R. Cheng and A. R. Hurson. Effective clustering of complex objects in object-oriented databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 22–32, Denver, CO, May 1991.

[DH73] W. E. Donath and A. J. Hoffman. Lower bounds for the partitioning of graphs. *IBM Journal of Research and Development*, 17(9):420–425, 1973.

[Fah93] Stephan Fahrig. Untersuchung von cluster-algorithmen basierend auf graphpartitionierung. Master's thesis, RWTH Aachen, D-5100 Aachen, Mar 1993.

[FM82] C. Fidducia and R. M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proc. of the Design Automation Conference*, pages 175–181, 1982.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness*. Freeman, San Francisco, 1979.

[GKKM92] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Clustering in object bases. Technical Report 6/92, Fakultät für Informatik, Universität Karlsruhe, D-7500 Karlsruhe, Jun 1992.

[Had74] F. O. Hadlock. Minimum spanning forests of bounded trees. In *Proc. of the Southeastern Conference on Combinatorics, Graph Theory and Computing (Congressus Numerantium X)*, pages 7–18, 1974.

[HK89] S. E. Hudson and R. King. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Trans. on Database Systems*, 14(3):291–321, Sep 1989.

[HZ87] M. Hornick and S. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Trans. Office Inf. Syst.*, 5(1):70–95, Jan 1987.

[KL70] B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal*, 49(2):291–307, Feb 1970.

[Kri84] B. Krishnamurthy. An improved min-cut algorithm for partitioning VLSI networks. *IEEE Trans. on Comp.*, 33(5):438–446, 1984.

[Kru56] J. B. Kruskal. On the shortest spanning subgraph of a graph and the travelling salesman problem. *Proc. of the Amer. Math. Soc.*, 7:48–50, 1956.

[Luk74] J. Lukes. Efficient algorithm for the partitioning of trees. *IBM Journal of Research and Development*, 18:217–224, 1974.

[SM91] K. Shahookar and P. Mazumber. VLSI cell placement techniques. *ACM Computing Surveys*, 23(2):143–220, Jun 1991.

[Sta84] J. Stamos. Static grouping of small objects to enhance performance of a paged virtual memory. *ACM Trans. Comp. Syst.*, 2(2):155–180, May 1984.

[TN91] M. M. Tsangaris and J. F. Naughton. A stochastic approach for clustering in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 12–21, Denver, CO, May 1991.

[TN92] M. M. Tsangaris and J. F. Naughton. On the performance of object clustering techniques. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 144–153, San Diego, CA, Jun 1992.

A The Algorithm GGPla

```
algorithm GGPla(CG = (Obj, Ref), k)
  !! Additional data operators on nodes: lock(node), unlock(node).
  !! Additional data operators on edges: set_mark(edge,mark), get_mark(edge).
  !! Data objects are shared, that means that each node or edge only has a reference to it's data.
  !! The labels NONE, REJECT, TRIGGER and CLOSE describe the edges' states.

  Let EdgeList be a list containing all edges Ref;
  Sort EdgeList (decreasing) according to their weights;
  Let RootSet be a set of binary trees;
  Initialize RootSet := ∅;
  forall o in Obj do begin
    Let LocalEdgeList(o) be a list of edges (o, o_i) emanating from object o;
    Sort LocalEdgeList(o) (decreasing) according to weight((o,o_i))/size(o_i);
    Let RejectSet(o) := ∅;   !! a set of rejected edges for object o
    Let r := make_node(o);   !! a singe tree node containing o
    Insert r into RootSet;
  end;

  while EdgeList is not empty do begin
    Remove the first element (o_1, o_2) from EdgeList;
    Let r_1 := root(o_1);   !! the root of the tree that contains o_1
    Let r_2 := root(o_2);
    if r_1 ≠ r_2 and all objects in r_1 ∪ r_2 fit on one page then begin
      if reject((o_1, o_2), r_1, r_2, k) then begin
        if get_mark((o_1, o_2)) = NONE then begin
          EdgeList := EdgeList + (o_1, o_2);
        end
        set_mark((o_1, o_2), REJECT);
      end else begin
        Let r := join_trees(r_1,r_2);   !! left(r) = r_1, right(r) = r_2
        Remove r_1, r_2 from RootSet;
        Insert r into RootSet;
        set_mark((o_1, o_2), CLOSE);
        forall (o_i, o_j) in RejectSet(o_1) ∪ RejectSet(o_2) do begin
          exclude ((o_i, o_j)) from RejectSet(o_1) or RejectSet(o_2);
          if get_mark((o_i, o_j)) = REJECT then begin
            EdgeList := (o_i, o_j) + EdgeList;
            set_mark((o_i, o_j), TRIGGER);
          end;
        end;
      end;
    end;
  end;
end algorithm;
```

```
boolean procedure reject((o_1, o_2), r_1, r_2, k)
  lock(r_1); lock(r_2);
  Let (weight_greedy, dummy, dummy) := best-first-lookahead({o_1, o_2}, {r_1, r_2}, k);
  Let (weight_left, E_left, R_left) := best-first-lookahead({o_1}, {r_1}, k);
  Lock all roots in R_left;
  Let (weight_right, E_right, R_right) := best-first-lookahead({o_2}, {r_2}, k);
  Unlock all roots in R_left;
  Let gain_left := weight_left > weight((o_1, o_2)) + weight_greedy ∧
        all objects in R_left ∪ r_1 ∪ r_2 don't fit on one page;
  Let gain_right := weight_right > weight((o_1, o_2)) + weight_greedy ∧
        all objects in R_right ∪ r_1 ∪ r_2 don't fit on one page;
  Let gain_all := weight_left + weight_right > weight((o_1, o_2)) + weight_greedy ∧
        all objects in R_left ∪ R_right ∪ r_1 ∪ r_2 don't fit on one page;
  Let reject := gain_left ∨ gain_right;          !! Heuristics H1
  if ¬ reject ∧ gain_all then                    !! Heuristics H2
    Let reject := gain_left := gain_right := True;
  end;
  if gain_left then
    forall (o_i, o_j) ∈ E_left do
      Include (o_1, o_2) into RejectSet(o_j);
    end;
  end;
  if gain_right then
    forall (o_i, o_j) ∈ E_right do
      Include (o_1, o_2) into RejectSet(o_j);
    end;
  end;
  unlock(r_1); unlock(r_2);
end procedure;


(unsigned, set, set) procedure best-first-lookahead(SetOfObjects, SetOfRoots, k)
  Let w := 0;
  Let R := ∅;    !! set of roots
  Let E := ∅;    !! set of edges;
  Let L := ∅;    !! list of best-first edges
  forall (o, o_i) ∈ Ref : o ∈ SetOfObjects ∧ get_mark(o, o_i) ∈ { NONE, TRIGGER } do
    include (o, o_i) into L;    set_depth(o_i,1);    !! object o_i on depth 1
  end;
  Sort L (decreasing) according to weight((o,o_i))/size(o_i);
  while L is not empty do
    Remove the first element (o, o') from L;
    Let r := root(o');
    if ¬ locked(r) ∧ all objects in SetOfRoots ∪ R ∪ r fit on one page then
      include r into R;    lock(r);    include (o, o') into E;
      w := w + weight((o, o'));
      if get_depth(o') < k then
        forall (o', o'') ∈ Ref: get_mark(o', o'') ∈ { NONE, TRIGGER } do
          include (o', o'') into L;    set_depth(o'', get_depth(o') + 1);
        end;
        Sort L (decreasing) according to weight((o_i,o_j))/size(o_j);
      end;
    end;
  end;
  unlock all roots in R;
  return(w, E, R);
end procedure;
```

B The Algorithm BSTF

algorithm Bstf ($CG = (Obj, Ref), \leq_{Obj}, \leq_{Ref}, Sel_{Succ}$):
    !! $\leq_{Obj}$ defines an order on all nodes.
    !! $\leq_{Ref}$ defines an order on all edges.
    !! $Sel_{Succ}(o, \leq_{Ref})$ selects all successors of node o. The successors are sorted
    !! according the specified order. Note that heuristics UDE is incorporated in Sel_{Succ}.
    Let o^* be the first object of Obj according to $\leq_{\mathrm{Obj}}$;
    Let P be a new page;
    while $o^* \neq$ NULL **do**
        Let $L_1 := < (o^*, o^*) >$;   !! Dummy edge to initialize the list L_1.
        while L_1 is not empty **do**
            Remove the first element (o, o') from L_1;
            if o' is not marked as visited **then**
                if o' does not fit on page P **then**
                    Let P be a new page;
                    [break;]   !! Only in algorithms Bstf.npnt and Bstf.npnt.ntnp.
                end if;
                Store o' on P and mark o' as visited;
                $L_1 := L_1 + Sel_{\mathrm{Succ}}(o', \leq_{Ref})$;
                Sort L_1 by $\leq_{Ref}$;
            end if;
        end while;
        Let o^* be the next non-visited object of Obj according to $\leq_{\mathrm{Obj}}$;
        [Let P be a new page;]   !! Only in algorithms Bstf.ntnp and Bstf.npnt.ntnp
    end while;
end algorithm;