Journal of Global Optimization **29:** 225–241, 2004.

A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning

A.J. SOPER, C. WALSHAW and M. CROSS
School of Computing and Mathematical Sciences, University of Greenwich, Old Royal Naval College, Greenwich, London, UK (*e-mail: {a.j.soper, c.walshaw}@gre.ac.uk.*)

(Received 7 February 2002; accepted 7 October 2003)

Abstract. The graph-partitioning problem is to divide a graph into several pieces so that the number of vertices in each piece is the same within some defined tolerance and the number of cut edges is minimised. Important applications of the problem arise, for example, in parallel processing where data sets need to be distributed across the memory of a parallel machine. Very effective heuristic algorithms have been developed for this problem which run in real-time, but it is not known how good the partitions are since the problem is, in general, NP-complete. This paper reports an evolutionary search algorithm for finding benchmark partitions. A distinctive feature is the use of a multilevel heuristic algorithm to provide an effective crossover. The technique is tested on several example graphs and it is demonstrated that our method can achieve extremely high quality partitions significantly better than those found by the state-of-the-art graph-partitioning packages.

Key words. evolutionary search, genetic algorithms, graph-partitioning, multilevel optimisation.

1. Introduction

The need for graph-partitioning arises naturally in many applications. For example in complex finite element & finite volume computational mechanics codes, the large meshes required are often too big to fit onto serial computers, either because of memory limitations or computational demands, or both. Distributing a graph (corresponding to the computational and communication requirements of the mesh) across a parallel computer so that the computational load is evenly balanced and the data locality maximised is known as graph-partitioning. It is well known that this problem is NP-complete (i.e. it is unlikely that an optimal solution can be found in polynomial time), e.g. [7], so in recent years much attention has been focused on developing suitable heuristics, and a range of powerful methods have been devised, e.g. [12].

In this paper we report on a technique, combining an evolutionary search algorithm together with a multilevel graph partitioner, which has enabled us to find partitions considerably better than those that can be found by any of the public domain graph-partitioning packages such as JOSTLE, METIS, CHACO, etc. We do not claim this evolutionary technique as a possible substitute for the aforementioned packages; the very long run times (e.g. up to a week) preclude

such a possibility for the typical applications in which they are used. However we do consider it of interest to find the best possible partitions and for certain applications such as circuit partitioning, where the quality of the partition is paramount, the computational resources required may be completely justified by the very high quality partitions that the technique is able to find. Even for applications where the partitioning overhead needs to be as small as possible, such as parallel scientific computing, we believe the results are useful for benchmarking purposes. As a consequence we also report on the establishment of a public domain archive containing what we believe to be the best partitions found so far for a range of public domain graphs.

1.1. OVERVIEW

The main focus of this paper is to describe a strategy for combining evolutionary search techniques with a standard graph-partitioning method. A particularly popular and successful class of algorithms that address the graph-partitioning problem are known as multilevel algorithms, e.g. [28]. They usually combine a graph contraction algorithm which creates a series of progressively smaller and coarser graphs together with a local optimisation method which, starting with the coarsest graph, refines the partition at each graph level. In Section 2 we outline such an algorithm and discuss the salient features. We employ the evolutionary search algorithm by constructing a population of variants of the original graph (differing from the original only by edge weighting) and then use this multilevel algorithm almost as a 'black box' operator to determine their fitness by computing a partition of each which hopefully will also be a good partition of the original graph. The population evolves either by individual members mutating or by several members crossing with each other to generate a different (and hopefully fitter) child. The details of this approach are described in Section 3, in particular the crossover & mutation operators (Sections 3.2 and 3.3). Related work is discussed in Section 3.5. We have conducted many experiments to test the technique and in Section 4 present some of the results including benchmarks of public domain partitioning packages (Section 4.1) and tests for the effectiveness of the evolutionary search (Section 4.2). Finally, in Section 5, we summarise the work, present some conclusions and list some suggestions for further research.

Note that although we describe a serial version of the multilevel algorithm, in principle, the same strategy could be used to enable a parallel version of the code by employing the parallel version of the multilevel algorithm, [30]. Alternatively, a processor farm could be utilised by distributing each partitioning calculation to an idle processor. However we have not implemented either strategy and indeed it might be difficult to obtain parallel resources for such long run-times.

The principal innovation described in this paper is the combination of evolutionary search techniques and a multilevel graph-partitioner. Most importantly we have devised and implemented new crossover and mutation operators

which can be applied to partitions with the aim of improving the overall fitness of each successive generation.

1.2. NOTATION AND DEFINITIONS

Let $G=G(V,E)$ be an undirected graph of vertices V, with edges E. We assume that both vertices and edges can be weighted and that $|v|$ denotes the weight of a vertex v and similarly for edges and sets of vertices and edges. Given that the mesh needs to be distributed to P processors, define a partition π to be a mapping of V into P disjoint subdomains S_p such that $\cup_P S_p = V$. The weight of a subdomain is just the sum of the weights of the vertices in the subdomain, $|S_p| = \sum_{v \in S_p} |v|$ and we denote the set of inter-subdomain or cut edges (i.e. edges cut by the partition) by E_c.

In the context of partitioning a mesh for a parallel application, the aim is to find a partition which evenly balances the load or vertex weight in each subdomain whilst minimising the communications cost. To evenly balance the load, the optimal subdomain weight is given by $\bar{S} := \lceil |V|/P \rceil$ (where the ceiling function $\lceil x \rceil$ returns the smallest integer greater than x) and the *imbalance* is then defined as the maximum subdomain weight divided by the optimal (since the computational speed of the underlying application is determined by the most heavily weighted processor). There is some discussion about the most appropriate metric for partitioning, e.g. [11], and indeed it is unlikely that any one metric is appropriate, however, it is common practice in graph-partitioning to approximate the communications cost by $|E_c|$, the weight of cut edges or *cut-weight.* The usual (although not universal) definition of the graph-partitioning problem is therefore to find π such that $|S_p| \leqslant \bar{S}$ and such that $|E_c|$ is (approximately) minimised.

2. Multilevel Graph-Partitioning

In recent years it has been recognised that an effective way of both speeding up graph-partitioning techniques and/or, perhaps more importantly, giving them a global perspective is to use multilevel techniques. The idea is to match pairs of vertices to form *clusters*, use the clusters to define the vertices of a new graph and recursively apply this procedure until the graph size falls below some threshold. The coarsest graph is then partitioned (possibly with a crude algorithm) and the partition is successively refined on all the graphs starting with the coarsest and ending with the original. This sequence of contraction followed by repeated expansion/optimisation loops is known as the multilevel paradigm and has been successfully developed as a strategy for overcoming the localised nature of the Kernighan-Lin (KL) [16], and other optimisation algorithms. The use of multilevel combinatorial refinement for partitioning was first proposed by both Hendrickson and Leland [12], and Bui and Jones [3], and was inspired by Barnard and Simon [2], who used a multilevel numerical algorithm to speed up spectral partitioning.

2.1. THE MULTILEVEL ALGORITHM

2.1.1. *Graph Contraction*

To create a coarser graph $G_{l+1}(V_{l+1}, E_{l+1})$ from $G_l(V_l, E_l)$ we find a maximal independent subset of graph edges, or a *matching* of vertices, and then collapse them. The set is independent if no two edges in the set are incident on the same vertex (so no two edges in the set are adjacent), and maximal if no more edges can be added to the set without breaking the independence criterion.Having found such a set, each selected edge is collapsed and the vertices, $u_1, u_2 \in V_l$ say, at either end of it are merged to form a new vertex $v \in V_{l+1}$ with weight $|v| = |u_1| + |u_2|$.

The problem of computing a matching of the vertices is known as the maximum cardinality matching problem. Although there are optimal algorithms to solve this problem, they are of at least $O(V^{2.5})$, e.g. [21]. Unfortunately this is generally too slow for partitioning and, since it is not too important for the multilevel process to solve the problem optimally, we use a variant of the edge contraction heuristic proposed by Hendrickson and Leland [12]. Their method of constructing a maximal independent subset of edges is to create a randomly ordered list of the vertices and visit them in turn, matching each unmatched vertex with an unmatched neighbouring vertex (or with itself if no unmatched neighbours exist). Matched vertices are removed from the list. If there are several unmatched neighbours the choice of which to match with can be random, but it has been shown by Karypis and Kumar [14], that it can be beneficial to the optimisation to collapse the most heavily weighted edges and our matching algorithm uses this heuristic.

2.1.2. *The Initial Partition*

Having constructed the series of graphs until the number of vertices in the coarsest graph is smaller than some threshold, the normal practice of the multilevel strategy is to carry out an initial partition. Here, following Gupta [10], we contract until the number of vertices in the coarsest graph is the same as the number of subdomains, P, and then simply assign vertex i to subdomain S_i.

2.1.3. *Partition Extension*

Having optimised, the partition on a graph G_l, the partition must be extended onto its parent G_{l-1}. This is a trivial process; if a vertex $v \in V_l$ is in subdomain S_p then the matched pair of vertices that it represents, $v_1, v_2 \in V_{l-1}$, will be in S_p.

2.2. THE ITERATIVE OPTIMISATION ALGORITHM

The iterative optimisation algorithm that we use at each graph level is a multi-way variant of the Kernighan-Lin (KL) bisection optimisation algorithm. The algorithm, as is typical for KL type algorithms, has inner and outer iterative loops with

the outer loop terminating when no migration takes place during an inner loop. The vertices are ranked by gain (the potential improvement in the cost function, in this context the cut-weight) and the inner loop examines vertices, highest gain first, and, provided the balance constraint is not broken, transfers them between subdomains if an improvement to the cost function accrues. The algorithm also has a limited ability to escape local minima by marking vertices with a negative gain for transfer and subsequently migrating a group of them if the aggregate transfer results in an improved cost. It is fully described in [29].

Our implementation also uses bucket sorting, the linear time complexity improvement introduced to partitioning by Fiduccia and Mattheyses [6], which we have extended for use with non-integer gains by integer scaling. The bucket sort is an essential tool for the efficient and rapid sorting and adjustment of vertices by their gain. The idea is that all vertices of a given gain g are placed together in an unsorted 'bucket' which is ranked g; finding a vertex with maximum gain then simply consists of finding the (non-empty) bucket with the highest rank and picking a vertex from it. If the vertex is subsequently transferred between subdomains then the gains of adjacent vertices are adjusted and the list of vertices which are candidates for migration (re)sorted by gain. Using a bucket sort for this operation simply requires recalculating the gain of each affected vertex and, if different, transferring it to the appropriate bucket.

The only difficulty in adapting this procedure for use with the evolutionary algorithm is that we wish to add small non-integer biases to the edge weights to influence the partitioner and as a result the gains are also real (non-integer) numbers. In fact the solution we use is to give each bucket an interval of gains rather than a single integer, e.g. the bucket ranked 1 could contain any vertex with gain in the interval $[0.5, 1.5]$. However, the issue of scaling then arises since the biases are often very small quantities. Fortunately, we can easily calculate the maximum possible gain by finding the vertex in the graph, $\bar{v} \in G$, with the largest sum of edge weights. The maximum possible gain, $g_{\max}$, would then occur if $\bar{v}$ insubdomain S_p say, were entirely surrounded by neighbours in different subdomains. The value for $g_{\max}$ is then given by $g_{\max} = \sum_{v' \in \Gamma(\bar{v})} |(\bar{v}, v')|$ (where $\Gamma(\bar{v})$ is the set of vertices $v' \in V$ adjacent to $\bar{v}$) and the minimum gain is $-g_{\max}$. This means we can easily choose the number of buckets, B say, and scale the gain accordingly so that for a gain g we calculate the appropriate bucket by finding the integer part of

$$\frac{gB}{g_{\max} - (-g_{\max})} = \frac{gB}{2\sum_{v' \in \Gamma(\bar{v})} |(\bar{v}, v')|}.$$

The number of buckets, B, which is inversely proportional to the length of each interval, should be chosen so that it is large enough to distinguish between reasonably different gains but not so large that every different gain value requires a different bucket (with the consequence that the search for a particular bucket becomes inefficient). The experiments carried out here all used a scaling which allowed a maximum of $B = 1,000$ buckets and we believe that this is large enough

to sufficiently distinguish between gains arising from different biases generated by the crossover & mutation operators (Sections 3.2 and 3.3). Finally note that the buckets are stored in a binary tree which allows $O(\log B)$ searches for the highest ranked nonempty bucket and that the algorithm simply picks the first vertex in the bucket rather than scanning the entire bucket for the precise vertex with highest gain.

3. Combining Evolutionary Search with the Multilevel Graph Partitioner

Evolutionary search is a stochastic search technique that generates new points (or individuals which in our case are partitions) in a search space using information from a finite population of already evaluated points. Typically a new population of equal size to the current population is generated, which in turn provides the basis for producing a further population (termed a generation) and so on. This process is given direction by selecting more information from the fitter individuals in the current population when producing new search points, [9]. In this context the fitness refers to the partition quality and takes account of the number of cut edges and the imbalance.

Each new search is produced by one of two operations: crossover which combines information from two or more randomly selected individuals in the current generation, and mutation which modifies a single, randomly selected, individual. The construction of successful crossover and mutation operators is problem specific and often complex, especially where individuals are subject to constraints (as for partitioning) so that information from different individuals cannot be arbitrarily combined or modified. Further, the information needs to be effectively exploited so that new individuals result that are fitter than the current fittest with sufficient probability even when the current generation is already very good, [1].

Evolutionary search algorithms have recently been successfully applied to a diverse set of problems providing useful examples of crossover and mutation operators which provide a guide for developing such operators for new problems. The operators described in this paper extend an approach to the Travelling Salesman Problem (TSP) [26], and the Constrained Minimum Spanning Tree Problem (CMSTP) [23], both of which require a search for a set of links satisfying constraints (forming a tour for the TSP) and for which the sum of their costs is a minimum. Clearly the graph-partitioning problem is of similar character.

The approach normally works by first defining a parametric representation for candidate tours (or CMSTs) upon which the many crossover and mutation operators available for parametric problems can then act, [9]. The parametric representations are produced by 'biasing' the link costs, i.e. adding spurious positive values to the cost of each link, and then applying a particular heuristic algorithm to produce the corresponding tour or CMST. The heuristic algorithm

used should give good solutions for a large range of different problems (sets of link costs) and in this context is the multilevel partitioner. We use biased edge weights to alter the output of this partitioner, but form our genetic operators differently.

3.1. INTERACTION WITH THE MULTILEVEL PARTITIONER

We first describe how the partitioner should respond to a graph with biased edge weights. In fact the multilevel partitioner used (as described in Section 2) is known as JOSTLE and for simplicity we shall henceforth refer to it as such, although in principle the evolutionary search should work with any graph-partitioning heuristic which accepts real (non-integer) edge weights. Indeed by suitable integer scaling the approach should work with the more common partitioners which are restricted to integer weights.

The basic idea is that each vertex is assigned a bias ($\geqslant 0$), and each edge a dependent weight of unity plus the sum of the biases of its end vertices. JOSTLE responds to these edge weights so that:

(a) when contracting a graph, heaviest edges are collapsed first (subject to their being independent);
(b) when performing iterative optimisation, vertex gains are calculated using the biased edge weights.

When applying JOSTLE to a graph with biased edge weights, the general effect will be that vertices with a small bias are more likely to appear at the boundary of a subdomain than those with a large one, and that edges with lower biased weight are more likely to be cut.

To generate each new offspring we construct a set of biases from one or more existing parent partitions and then use JOSTLE to create a new partition. The bias values are chosen carefully (although with a randomised component) so that JOSTLE's ability to produce partitions of good quality (with respect to the unweighted graph) is not too much impaired, while at the same time there is information transfer from the parents to the offspring. Since the bias values are discarded after a new offspring has been produced, the evolutionary algorithm described here is not a traditional genetic algorithm since no representation (or genotype) is maintained, as distinct from an actual partition.

3.2. CROSSOVER OPERATOR

We create a new set of biases from a selected number of parent partitions as follows:

For each vertex in the graph, examine whether in two or more of the parent partitions that vertex is a border vertex (ends a cut edge). If so, assign the vertex a bias value chosen randomly and uniformly from the range $[0, 0.01]$. *Otherwise assign a bias value of* 0.1 *plus a random number chosen in the same range.*

Border vertices common to two or more parents will occur where either identical or adjacent cut edges also occur. Either way we take this as evidence that the vertex should remain as a border one in the child—under the assumption that the presence of this particular border vertex is a contributing factor to the fitness of two or more fit parents. Hence a very small bias is assigned. We then make all other vertices less likely to become border vertices by assigning them a larger bias value. Since in both cases the bias value is small—much less than the unit weight of an edge—JOSTLE will produce a partition for the most part optimised with respect to the true edge weights. Hence the requirement for a successful crossover operator to transfer information is fulfilled. Finally, at each mating two, three or four mates are randomly chosen to crossover together, a range found to work well empirically.

Figure 1 illustrates and motivates the aims of the crossover operator. Figures 1(a) and 1(b) show two possible (perfectly balanced) partitions of the example graph whilst Figure 1(d) shows the optimal partition of this graph into 4 subdomains. In Figure 1(c) the border vertices common to both partitions (a) and (b) are shown ringed. Using the crossover scheme, despite a certain amount of random variation, the biased graph that is input to JOSTLE will have edges between two ringed vertices with the lightest weights whilst edges between two unringed vertices will be the heaviest. JOSTLE attempts to minimise the total weight of cut edges and so is more likely to cut between pairs of ringed vertices and hopefully would produce a solution closer to or the same as the optimal. However the Figure also illustrates the dangers of making the biasing too extreme. The two ringed vertices in Figure 1(c) indicated by arrows are not border vertices in the optimal partition. Thus, if the biasing is too heavy, JOSTLE is likely to find a good partition of the biased graph but which does not correspond to a good partition of the original. In this way the crossover operator aims to retain information about common strengths of two or more parents whilst allowing further investigation of the search space.

Finally notice that Figure 1 (for reasons of space) only illustrates the case when two parents are crossed and we look for border vertices common to both parents. In fact the operator can be even more powerful when combining three or more partitions as we can achieve genuinely constructive composition. For example with three parents A, B and C, the resulting biased graph will show common border vertices between A and B, B and C, and C and A which means

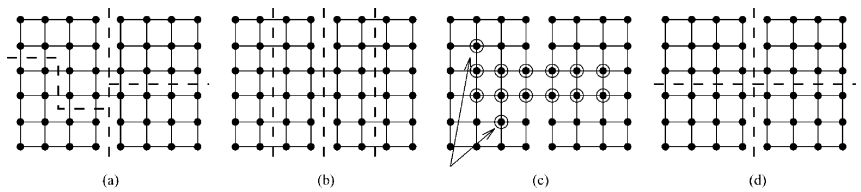

Figure 1. An illustration of the crossover operator: (a) and (b) partitions of an example graph, (c) common border vertices from these two partitions, (d) the optimal partition into 4 subdomains.

that the resulting partition will hopefully retain the best parts of all three of these combinations.

3.3. MUTATION OPERATOR

We create a new set of biases from a parent partition as follows:

For each vertex in the graph examine whether it is a border vertex, the neighbour of a border vertex or the neighbour of a neighbour of a border vertex. If so, assign the vertex a bias value chosen randomly and uniformly from the range $[0, 0.01]$. *Otherwise assign a bias value of* 2.0 *plus a random number chosen in the same range.*

Considering the vertex biases as forming a landscape over the graph with the bias at any vertex giving its height, the effect will be a deep, flat-bottomed trench along the partition boundaries. The trench is considered deep since edge weights within the trench will be 4.0 different from those outside, so that JOSTLE's optimisation stage will have a strong tendency to place boundaries within the trench. In addition, since the edge weight biases within the trenches are small compared with unity, the true edge weight, JOSTLE can still successfully optimise within the trenches with respect to the true total edge weight. This operator is partially motivated by the fact that certain graphs, and in particular those representing unstructured meshes, often show considerable regularity, especially locally in the form of translational symmetry, so that good quality partition boundaries are often found, nearby and locally parallel to each other. The choice of the bias value, 2.0, was again chosen by experimentation.

Figure 2 illustrates the mutation operator and shows (a) a partition of the given graph and (b) the optimal partition into two subdomains of this graph. In Figure 2(a) we have also ringed all the border vertices, neighbours of border vertices and neighbours of neighbours of border vertices. As for Figure 1 the lightest edges of the resulting biased graph are those between two ringed vertices whilst those between two unringed vertices will be substantially heavier.

3.4. GENETIC ALGORITHM PARAMETERS

The fitness function allows the evolutionary search algorithm to rank the partitions by their quality and the fitness of a partition was defined to be $-C\lambda$ where C is

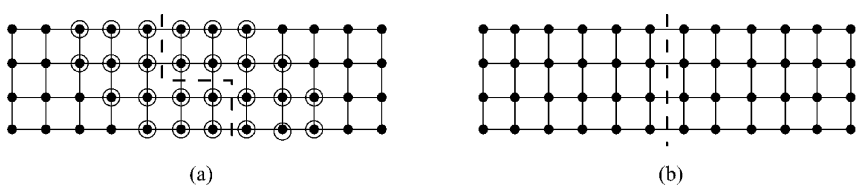

Figure 2. An illustration of the mutation operator: (a) an example partition with border vertices and their near neighbours ringed, (b) the optimal partition into 2 subdomains.

the number of cut edges and λ the imbalance. The fitness function thus imposes a soft, but heavy penalty on partitions with greater imbalance; sufficiently heavy so that partitions within the balance constraint eventually dominate the population as evolutions progress.

Due to the size of the meshes and the time required to execute JOSTLE, a fairly small population size of 50 is used. Each new generation is created as follows: 50 new offspring are produced by either crossover or mutation at a ratio of 7:3. Mating groups of individuals for crossover and candidates for mutation are selected randomly from the current generation, but with each parent participating in at least one trial. The union of the sets of offspring and parents are then ranked by fitness and the best 50 form the new generation. The fact that members of a population are only ever discarded when offspring of greater fitness are generated is known as an elitist strategy, [9]; it is appropriate in this case because most of the offspring generated are not of very high quality, [5].

The random initial population was generated by (for each individual) assigning to every vertex bias values chosen randomly and uniformly from [0, 0.1], and then using JOSTLE to generate a partition. 1000 generations were allowed for each run of the genetic algorithm, giving 50,000 evaluations of JOSTLE.

The genetic algorithm described here is a very simplified instance of the CHC Adaptive Search Algorithm, [5], but lacks incest prevention and restarts.

3.5. RELATED WORK

There are a number of papers about topics related to the work presented here. Amongst genetic algorithm approaches, the most common is based on a direct encoding, i.e. the subdomain membership of each vertex is explicitly represented by the value of one gene. A linear chromosome is formed by ordering and concatenating the genes, and crossover is of the normal type (i.e. based on natural evolutionary processes, [9]). After crossover, offspring are normally improved by local optimisation before being placed in the succeeding generation. In chronological order, this approach has been used by: Mansour and Fox [19], who progressively enforced the equi-partition constraint using a penalty term; Talbi and Bessiere [25], who employed a cellular population structure for the genetic algorithm; and Bui and Moon [4], who performed chromosomal alignment before a 5-point crossover and local optimisation based on Kernighan-Lin, Gil and Ortega have performed circuit partitioning using the direct encoding [8], whilst Kang and Moon [13], and Kim and Moon [18], have improved the local optimisation algorithms and performed more extensive tests on larger graphs (up to 5,000 vertices) divided into up to 32 subdomains.

von Laszewski has used an alternative ‘structural genetic operator’ which copies one entire subdomain from one parent into the other to form an offspring; this was tested on graphs of up to 1000 vertices, [27]. Khan and Topping partitioned very small meshes in parallel finite element analysis using a genetic algorithm [17],

and Muhlenbein et al. have experimented on small graphs (100 vertices) with a network algorithm which 'evolves' a partition by simulating deterministic, differential equations [20].

In contrast to the above, the work described here uses a multilevel optimisation scheme (JOSTLE) as the basis of crossover and mutation operators for acting on partitions of unstructured meshes.

4. Experimental Results

We have implemented the algorithms described here within the framework of JOSTLE, a mesh partitioning software tool developed at the University of Greenwich and freely available for academic and research purposes under a licensing agreement[1]. The experiments were carried out on a variety of different machines; with its very long runtimes (of several days in the case of the larger graphs), the evolutionary search approach can soak up CPU cycles and the tests were run so as to use up any spare capacity in the system. As a result we have not measured runtimes.

The test graphs have been chosen to be a representative sample of small to medium scale real-life problems (mostly mesh-based) and include both 2D and 3D examples of nodal graphs (where the mesh nodes are partitioned) and dual graphs (where the mesh elements are partitioned). In addition there is a 3D semi-structured graph, cti, which is unstructured in the $x-y$ plane but extended regularly along the z-axis. Finally the test suite includes three non mesh-based graphs (add32, vibrobox, bcsstk32) which arise from various scientific computing applications[2]. None of the graphs have either vertex or edge weights; such graphs are not widely available since most applications do not accurately instrument costs and it is difficult to draw meaningful conclusions from the few examples that we have access to. Table 1 gives a list of the graphs, their sizes, the maximum, minimum & average degree of the vertices and a short description. As the graphs are not weighted, the number of vertices in V is the same as the total vertex weight $|V|$ and similarly for the edges E.

It has been noted for some time, e.g. [22], that graph-partitioning algorithms can often find higher quality partitions if the balancing constraint is relaxed slightly. Indeed some of the public domain graph-partitioning packages such as JOSTLE & METIS have an in-built, although adjustable, imbalance tolerance of 3% (i.e. the largest subdomain is allowed to be up 1.03 times the size of the maximum allowed for perfect balance). We have tested the evolutionary algorithm with various tolerances but here restrict ourselves to reporting 3% imbalance results (further results for 0% imbalance can be found in [24]).

---

[1] available from http://www.gre.ac.uk/jostle

[2] the graphs are available from the Florida sparse matrix collection ftp://ftp.cis.ufl.edu/pub/umfpack/matrices/

Table 1. A summary of the test graphs

	Size		Degree			
Graph	V	E	Max	Min	Avg	Type
uk	4824	6837	3	1	2.83	2D dual graph
add32	4960	9462	31	1	3.82	32-bit adder (electronic circuit)
crack	10240	30380	9	3	5.93	2D nodal graph
wing-nodal	10937	75488	28	5	13.80	3D nodal graph
vibrobox	12328	165250	120	8	26.81	vibroacoustic matrix
4elt	15606	45878	10	3	5.88	2D nodal graph
cti	16840	48232	6	3	5.73	3D semi-structured graph
cs4	22499	43858	4	2	3.90	3D dual graph
bcsstk32	44609	985046	215	1	44.16	3D stiffness matrix
t60k	60005	89440	3	2	2.98	2D dual graph
wing	62032	121544	4	2	3.92	3D dual graph
brack2	62631	366559	32	3	11.71	3D nodal graph

Table 2 shows the results of using the evolutionary search algorithm for four values of P (the number of processors/subdomains). The table shows the number of cut edges or cut-weight which we denote C_E (the subscript E denoting the evolutionary algorithm).

4.1. BENCHMARKING OF PUBLIC DOMAIN PACKAGES

To assess the quality of the partitions, we have compared the results in Table 2 with those produced by several public domain partitioning packages including JOSTLE [29], METIS [15], and CHACO [12]. In all cases we have used the most up to date versions at the time of writing, JOSTLE 2.2 (March 2000), METIS 4.0 (September 1998) and CHACO 2.0 (October 1995). For METIS the algorithm chosen was kmetis the multilevel k-way scheme, and for CHACO we used the

Table 2. The results of the evolutionary search algorithm showing the cut-weight C_E

Graph	$P=4$	$P=8$	$P=16$	$P=32$
uk	41	83	157	266
add32	33	69	117	212
crack	361	676	1083	1699
wing-nodal	3590	5424	8361	12024
vibrobox	19245	24874	33676	43091
4elt	320	532	916	1540
cti	927	1716	2859	4438
cs4	936	1488	2204	3117
bcsstk32	9992	21307	38929	64433
t60k	215	469	886	1478
wing	1672	2551	4015	6039
brack2	2873	7114	12009	17952

Table 3. A comparison of cut-weight results for JOSTLE, C_J, against those of the evolutionary search algorithm, C_E

	$P=4$		$P=8$		$P=16$		$P=32$	
Graph	C_J	$\frac{C_J}{C_E}$	C_J	$\frac{C_J}{C_E}$	C_J	$\frac{C_J}{C_E}$	C_J	$\frac{C_J}{C_E}$
uk	71	1.73	106	1.28	180	1.15	315	1.18
add32	41	1.24	106	1.54	180	1.54	257	1.21
crack	413	1.14	751	1.11	1191	1.10	1804	1.06
wing-nodal	4055	1.13	5965	1.10	8947	1.07	12635	1.05
vibrobox	21844	1.14	30247	1.22	34521	1.03	45374	1.05
4elt	434	1.36	656	1.23	1012	1.10	1687	1.10
cti	1329	1.43	2086	1.22	3262	1.14	4683	1.06
cs4	1162	1.24	1588	1.07	2477	1.12	3330	1.07
bcsstk32	14887	1.49	25343	1.19	48395	1.24	74391	1.15
t60k	229	1.07	530	1.13	984	1.11	1588	1.07
wing	1844	1.10	2911	1.14	4681	1.17	6404	1.06
brack2	2999	1.04	7808	1.10	13164	1.10	19238	1.07
Average		1.26		1.19		1.16		1.10

multilevel KL method with recursive bisection and chose a coarsening threshold of 200.

Table 3 shows a comparison of the cut-weight results for the public domain version of JOSTLE compared to the evolutionary search algorithm. For each value of P, the first column shows the JOSTLE cut-weight results, C_J whilst the second column compares the results from JOSTLE scaled by the results from Table 2, C_J/C_E. Thus the figure of 1.73 for the uk graph and $P=4$ mean that the evolutionary algorithm was able to find a partition 73% better than JOSTLE in this case (although this is an extreme example). As can be seen JOSTLE provides partition qualities which are always worse, however this is hardly surprising since the JOSTLE algorithms lie at the heart of the evolutionary search scheme and JOSTLE is called 50,000 times for each experiment. Nonetheless it is interesting to see just how much better the partitions can be; the average difference in the quality ranges from 26% to 10% as P increases and can be as bad as 73%.

Note that differences in quality tend to diminish as P increases. It is tempting to speculate that this is because the margins for difference decrease as the number of vertices per subdomain ($\approx V/P$) decreases. Indeed in the limit where $V=P$ the only balanced partition (for an unweighted graph at least) is to put one vertex in each subdomain and so the differences vanish altogether.

We do not present detailed figures here for METIS and CHACO (although they can be found in [24]) but broadly the conclusions are the same as for JOSTLE. Thus METIS produces results which are on average from 37% to 14% worse than the evolutionary algorithm as P increases. Strictly speaking 5 out of the 48 results should not be admitted to this averaging as METIS failed to achieve the required imbalance and in the worst case (add32, $P=32$) the actual imbalance achieved was only 8.4%. Other than that it is difficult to spot any trends in the results other

than that METIS does particularly badly on the add32 graph (in one extreme case, $P=4$ the METIS cut-weight is over three times worse than the evolutionary algorithm) and best of all on the wing-nodal graph.

Similarly for CHACO, the average difference in the quality ranges from 29% to 14% as P increases, although CHACO uses recursive bisection and thus produces partitions with perfect balance (i.e. 0% imbalance tolerance) and so the comparisons are with the corresponding 0% results for the evolutionary scheme. CHACO does particularly well on the cti graph, perhaps as a result of its semi-structured nature which may suit the recursive bisection approach.

4.2. THE EFFECTIVENESS OF THE EVOLUTIONARY SEARCH ALGORITHM

It is of interest to ask how much the evolutionary search procedure contributes to finding the best partitions during an optimisation run. Each such run consists of 50,000 calls to JOSTLE, each with a slightly different graph (in terms of the edge weights) and may run for hours or even days and hence one would certainly expect the evolutionary approach to find higher quality partitions than any of the packages, all of which usually take less than a minute (and often less than a second) and only have one partitioning attempt. Therefore, with the aim of quantifying the added value of the evolutionary approach, we have run all the tests using graph variants with purely random biases to weight the edges. These biases were generated as if each new population were the initial one and had no dependence on the previous population (the random generation of an initial population is described in Section 3.4).

Figure 3 shows plots of the evolution of solution quality against the number of trials conducted for $P=8$ and $P=32$. Each point on the curves is calculated by averaging, for the entire test suite, the percentage excess in cut-weight over that for the best-known partition. The first 50 trials of both methods should produce identical results (discounting random noise) and hence the curves start close together. However, as can be seen, they rapidly separate and the evolutionary approach is shown to impart an advantage to solution quality which, although only small in absolute size, is nonetheless distinctive.

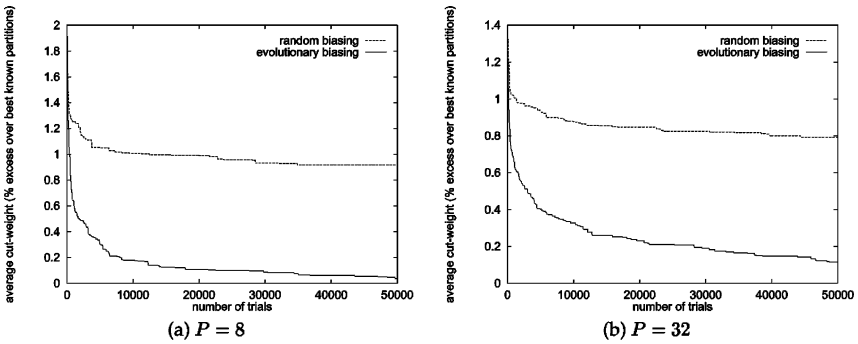

Figure 3. Plots of solution quality evolution against the number of trials.

The plots also give some indication of parameterisation for the evolutionary scheme. Thus the turning point in the gradient is between 5,000–10,000 trials suggesting that the algorithms should be applied for at least this long (100–200 generations) before the 'law of diminishing returns' starts to apply. Indeed the evolutionary algorithm does not even appear to have reached asymptotic convergence after 50,000 trials and so it could be legitimately argued that the evolutionary scheme could make use of further trials. In fact this is the reason why the curve fails to reach 0 as, for some of the graphs, we have generated better results than those reported here by running the algorithm for longer.

5. Summary and Future Research

We have described a new approach for addressing the graph-partitioning problem which combines an evolutionary search algorithm with a multilevel partitioner. Although not a practical method for applications in which the partition must be found rapidly, such as runtime partitioning of unstructured meshes, the approach is very successful at computing very high quality benchmark partitions especially when compared against state-of-the-art partitioning packages such as JOSTLE and METIS.

Despite the success of the testing we have not particularly addressed the partitioning of arbitrary graphs and have for the most part considered graphs arising from unstructured meshes. These tend to contain local connection patterns which are relatively homogeneous throughout the graph and, as such, tend to allow small incremental partition improvements. This is a boon to the crossover and mutation operators that we have devised and lead us to speculate that perhaps the technique might not be so useful on completely arbitrary graphs. However we believe that there is a large class of graphs with genuine applications for which the techniques will work.

We aim to investigate the techniques further by performing tests to quantify the performance of the evolutionary algorithm and to understand how it depends on the relative biases of boundary and interior vertices, and the number of parents during crossover. We are also very interested in looking at different types of application to which the strategy can be applied, and in particular those in which the long runtimes might not be considered a drawback provided the resulting partition was of the highest quality.

Finally, as a part of this work we have set up a public domain archive of various graphs together with the best partitions of them that we have been able to find. The archive is accessible via the world wide web at http://www.gre.ac.uk/~c.walshaw/partition and we invite researchers in the field to submit graphs and/or partitions for inclusion there. In particular the aim is to provide a benchmark against which partitioning algorithms can be tested and as a resource for experimentation.

References

1. Altenberg, L. (1995), The schema theorem and price's theorem, In: Whitley, L.D. and Vose, M.D. (eds), *Foundations of Genetic Algorithms 3*, Morgan Kaufmann, San Mateo, pp. 23–49.
2. Barnard, S.T. and Simon, H.D. (1994), A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems, *Concurrency: Practice & Experience*, 6(2), 101–117.
3. Bui, T.N. and Jones, C. (1993), A heuristic for reducing fill-in in sparse matrix factorization, In: Sincovec, R.F. et al. (ed.), *Parallel Processing for Scientific Computing*, SIAM, Philadelphia, pp. 445–452.
4. Bui, T.N. and Moon, B.R. (1996), Genetic algorithms and graph partitioning, *IEEE Trans. Comput.*, 45(7), 841–855.
5. Eshelman, L.J. (1991), The CHC adaptive search algorithm: How to have safe search when engaging in non-traditional genetic recombination, In: Rawlins, G.J.E. (ed.), *Foundations of Genetic Algorithms*, Morgan Kaufmann, San Mateo, pp. 265–283.
6. Fiduccia, C.M. and Mattheyses, R.M. (1982), A linear time heuristic for improving network partitions. In: *Proc. 19th IEEE Design Automation Conf.*, pp. 175–181. IEEE, Piscataway, NJ.
7. Garey, M.R., Johnson, D.S. and Stockmeyer, L. (1976), Some simplified NP-complete graph problems, *Theoret. Comput. Sci.*, 1, 237 –267.
8. Gil, C., Ortega, J., Diaz, A.F. and Montoya, M.G. (1998), Annealing-based heuristics and genetic algorithms for circuit partitioning in parallel test generation, *Future Generation Comput. Syst.*, 14(5), 439–451.
9. Goldberg, D. (1989), *Genetic Algorithms in Search, Optimization and Machine Learning*, Addison-Wesley.
10. Gupta, A. (1996), Fast and effective algorithms for graph partitioning and sparse matrix reordering, *IBM J. Res. Development*, 41(1/2), 171–183.
11. Hendrickson, B. and Kolda, T.G. (2000), Graph partitioning models for parallel computing. *Parallel Comput.*, 26(12), 1519–1534.
12. Hendrickson, B. and Leland, R. (1995), A multilevel algorithm for partitioning graphs, In: Karin, S. (ed.), *Proc. Supercomputing '95, San Diego*. ACM Press, New York.
13. Kang, S. and Moon, B.R. (2000), A hybrid genetic algorithm for multiway graph partitioning. In: Whitley, D. et al. (ed.), *Proc. Genetic & Evolutionary Comput. Conf.* (GECCO-2000), pp. 159–166. Morgan Kaufmann, San Francisco.
14. Karypis, G. and Kumar, V. (1998), A fast and high quality multilevel scheme for partitioning irregular graphs, *SIAM J. Sci. Comput.*, 20(1), 359–392.
15. Karypis, G. and Kumar, V. (1998), Multilevel k-way partitioning scheme for irregular graphs, *J. Parallel Distrib. Comput.*, 48(1), 96–129.
16. Kernighan, B.W. and Lin, S. (1970), An efficient heuristic for partitioning graphs, *Bell Syst. Tech. J.*, 49, 291–308.
17. Khan, A.I. and Topping, B.H.V. (1993), Subdomain generation for parallel finite element analysis, *Computing Systems Engrg.*, 4(4–6), 473–488.
18. Kim, J.P. and Moon, B.R. (2001), A hybrid genetic search for multi-way graph partitioning based on direct partitioning, In: Spector, L. et al. (ed.), *Proc. Genetic & Evolutionary Comput. Conf. (GECCO-2001)*, pp. 408–415. Morgan Kaufmann, San Francisco.
19. Mansour, N. and Fox, G.C. (1994), Allocating data to distributed-memory multiprocessors by genetic algorithms, *Concurrency: Practice & Experience*, 6(6), 485–504.
20. Muhlenbein, H., Gorges-Schleuter, M. and Kramer, O. (1988), Evolution algorithms in combinatorial optimisation, *Parallel Comput.*, 7(1), 65–85.
21. Papadimitriou, C.H. and Stieglitz, K. (1982), *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ.

22. Simon, H.D. and Teng, S.-H. (1997), How good is recursive bisection? *SIAM J. Sci. Comput.*, 18(5), 1436–1445.
23. Soper, A.J. and McKenzie, S. (1997), The use of a biased heuristic by a genetic algorithm applied to the design of multipoint connections in a local access network. In: *Proc. GALESIA '97*, pp. 113–116, (*2nd IEE Intl. Conf. Genetic Algorithms: Innovations and Applications*, Glasgow).
24. Soper, A.J., Walshaw, C. and Cross, M. (2000), A combined evolutionary search and multilevel optimisation approach to graph partitioning. Tech. Rep. 00/IM/58, Comp. Math. Sci., Univ. Greenwich, London SE10 9LS, UK.
25. Talbi, E. and Bessiere, P. (1991), A parallel genetic algorithm for the graph partitioning problem. In: *Proc. Intl. Conf. Supercomputing*, ACM Press, New York, pp. 312–320.
26. Valenzuela, C.L. and Williams, L.P. (1997), Improving heuristic algorithms for the travelling salesman problem by using a genetic algorithm to perturb the cities. In: Back, T. (ed.), *Proc. 7th Intl. Conf. Genetic Algorithms*, Morgan Kaufmann, San Mateo, pp. 458–464.
27. von Laszewski, G. (1991), Intelligent structural operators for the k-way graph partitioning problem, In: Belew, R.K. and Booker, L.B. (eds.), *Proc. 4th Intl. Conf. Genetic Algorithms*, Morgan Kaufmann, San Francisco, pp. 45–52.
28. Walshaw, C. (2002), An Exploration of Multilevel Combinatorial Optimisation. In: Cong, J. and Shinnerl, J. (eds.), *Multilevel Optimization in VLSIDAD*, Kluwer Academic Publishers, Boston, pp. 71–123. (Invited Paper).
29. Walshaw, C. and Cross, M. (2000), Mesh partitioning: a multilevel balancing and refinement algorithm. *SIAM J. Sci. Comput.* 22(1), 63–80 (originally published as Univ. Greenwich Tech. Rep. 98/IM/35).
30. Walshaw, C. and Cross, M. (2000), Parallel optimisation algorithms for multilevel mesh partitioning, *Parallel Comput.* 26(12), 1635–1660 (originally published as Univ. Greenwich Tech. Rep. 99/IM/44).