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, Park Row, Greenwich, London, SE10 9LS, UK.
email: {A.J.Soper, C.Walshaw}@gre.ac.uk

Mathematics Research Report 00/IM/58

April 12, 2000

Abstract

Graph partitioning divides a graph into several pieces by cutting edges. The graph partitioning problem is to divide so that the number of vertices in each piece is the same within some defined tolerance and the number of cut edges separating these pieces is minimised. Important examples of the problem arise in partitioning graphs known as meshes for the parallel execution of computational mechanics codes. Very effective heuristic algorithms have been developed for these meshes which run in real-time, but it is unknown 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 generate an effective linkage during 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.

Keywords: graph partitioning, evolutionary search, genetic algorithms, multilevel optimisation

1 Introduction

The need for graph partitioning arises naturally in many applications. For example in finite element (FE) and finite volume (FV) computational mechanics (CM) codes meshes composed of elements such as triangles or tetrahedra are often better suited than regularly structured grids for representing completely general geometries and resolving wide variations in behaviour via variable mesh densities. Meanwhile, the modelling of complex behaviour patterns means that the problems are often too large 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. [6], so in recent years much attention has been focused on developing suitable heuristics, and a range of powerful methods have been devised, e.g. [11].

Again, because of the complexity of the problem, it is impossible to verify heuristic solutions against a global optimum. Nonetheless, in this paper we aim to address this admittedly intractable issue in two ways. Firstly 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. The archive is maintained by one of the authors and will be updated as and when better results are found.

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 which address the graph partitioning problem are known as multilevel algorithms. 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 (§3.2 & §3.3). Related work is discussed in Section 3.6. 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 (§4.1) and tests for the effectiveness of the evolutionary search (§4.2).In Section 5 we then describe the public domain archive of the best results found so far (by any method) and finally, in Section 6, 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, [25]. Alternatively, there are many ways to parallelise the evolutionary search algorithm such as using a processor farm and distributing each partitioning calculation to an idle processor (assuming the memory of each processor is large enough to contain the whole graph and its multilevel components). 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.

2 Multilevel graph partitioning

In this section we discuss the graph partitioning problem and outline our multilevel algorithm, described in [24], for addressing it.

2.1 Notation and Definitions

Let $G = G(V, E)$ be an undirected graph of vertices V, with edges E. Typically for graphs arising from FE or FV meshes the edges will model the data dependencies in the mesh and the graph vertices can either represent mesh nodes (the nodal graph), mesh elements (the dual graph), a combination of both (the full or combined graph) or some other special purpose representation. 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. However, it is often the case that vertices and edges are given unit weights, $|v| = 1$ for all $v \in V$ and $|e| = 1$ for all $e \in E$ and indeed this is true for all of our test examples. 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 $\bigcup_P S_p = V$. 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.Vertices which have an edge in E_c (i.e. those which are adjacent to vertices in another subdomain) are referred to as *border* vertices. Finally, note that we use the words subdomain and processor more or less interchangeably: the mesh is partitioned into P subdomains; each subdomain S_p is assigned to a processor p and each processor p owns a subdomain S_p.

In the context of partitioning a mesh for a parallel application, the definition of the graph partitioning problem 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 $\overline{S} := \lceil |V|/P \rceil$[1] 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. [10], 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 $|S_p| \leq \overline{S}$ and such that $|E_c|$ is (approximately) minimised.

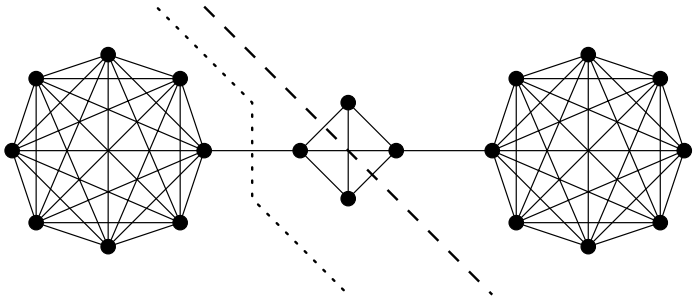

Figure 1: A graph illustrating that a certain amount of imbalance can improve partition quality (the dotted line) compared with perfect balance (the dashed line)

In fact it has been noted for some time that partition quality can often be improved if a certain amount of imbalance is allowed, [21]. Figure 1 illustrates this; if the graph is bisected and each subdomain must have 10 vertices, the minimum bisection cut-weight is 4 (e.g. the partition indicated by the dashed line) but if one subdomain is allowed 12 vertices the minimum cut-weight is 1 (the dotted line). If we allow θ% imbalance then the partitioning problem becomes 'find a partition π such that $|S_p| \leq \overline{S} \times (100 + \theta)/100$ and that $|E_c|$ is (approximately) minimised'.

2.2 The multilevel paradigm

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 a new graph and recursively iterate 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 optimised 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), [14], and other optimisation algorithms. The multilevel idea was first proposed by Barnard & Simon, [2], as a method of speeding up spectral bisection and improved by both Hendrickson & Leland, [11] and Bui & Jones, [3], who generalised it to encompass local refinement algorithms. Several algorithms for carrying out the matching of vertices have been devised by Karypis & Kumar, [12], while Walshaw & Cross describe a method for utilising imbalance in the coarsest graphs to enhance the final partition quality, [24].

[1] where the ceiling function $\lceil x \rceil$ returns the smallest integer greater than x

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. [19]. Unfortunately this is too slow for our purposes 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 & Leland, [11]. 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 & Kumar, [12], that it can be beneficial to the optimisation to collapse the most heavily weighted edges and our matching algorithm uses this heuristic.

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 the idea of Gupta, [9], 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. Unlike Gupta, however, we do not carry out repeated expansion/contraction cycles of the coarsest graphs to find a well balanced initial partition but instead, since our optimisation algorithm incorporates balancing, we commence on the expansion/optimisation sequence immediately.

Partition expansion. Having optimised the partition on a graph G_l, the partition must be interpolated onto its parent G_{l-1}. The interpolation itself is a trivial matter; 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.

2.3 The iterative optimisation algorithm

The iterative optimisation algorithm that we use at each graph level is a variant of the Kernighan-Lin (KL) bisection optimisation algorithm.Our implementation uses bucket sorting, the linear time complexity improvement of Fiduccia & Mattheyses, [5], which we have extended for use with non-integer gains by integer scaling (see below §2.4) and is a partition optimisation formulation; in other words it optimises a partition of P subdomains rather than a bisection. It is fully described in [24].

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. It uses two bucket sorting structures or *bucket trees* (binary trees of buckets – see below §2.4) and is initialised by calculating the gain – the potential improvement in the cost function (in this context the cut-weight) – for all border vertices and inserting them into one of the bucket trees. These vertices are referred to as *candidate* vertices and the tree containing them as the *candidate tree*.

The inner loop proceeds by examining candidate vertices, highest gain first (by always picking vertices from the highest ranked bucket), testing whether the vertex is acceptable for migration and then transferring it to the other bucket tree (the tree of *examined* vertices). If the candidate vertex is found acceptable, it is migrated, its neighbours have their gains updated and those which are not already in the examined tree are relocated in the candidate tree according to this updated gain. This inner loop terminates when the candidate tree is empty although it may terminate early if the partition cost rises too far above the cost of the best partition found so far. Once the inner loop has terminated any vertices remaining in the candidate tree are transferred to the examined tree and finally pointers to the two trees are swapped ready for the next pass through the inner loop.

The algorithm also uses a KL type local search strategy; in other words vertex migration from subdomain to subdomain can be *accepted* even if it degrades the partition quality and later, based on the subsequent evolution of the partition, either rejected or *confirmed*. During each pass through the inner loop, a record of the optimal partition achieved by migration within that loop is maintained together with a list of vertices which have migrated since that value was attained. If subsequent migration finds a 'better'

partition then the migration is *confirmed* and the list is reset. Note that it is possible to find better partitions despite selecting some vertices with negative gain because, as the optimiser runs, the gains of adjacent vertices will change and so the migration of a group of vertices some or all of which start with negative gain can in fact decrease the overall cost (i.e. produce a net positive gain). Once the inner loop is terminated, any vertices remaining in the list (vertices whose migration has not been confirmed) are migrated back to the subdomains they came from when the optimal cost was attained.

The algorithm, together with conditions for vertex migration acceptance and confirmation is fully described in [24].

2.4 Bucket sorting with non-integer edge weights

The bucket sort is an essential tool for the efficient and rapid sorting and adjustment of vertices by their gain. The concept was first suggested by Fiduccia & Mattheyses, [5], and 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 migrated from one subdomain to another then the gains of any affected vertices have to be 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 from its current bucket into another.

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. We have previously addressed this problem in [26] where the optimisation was aiming to minimise subdomain aspect ratio. In fact the solution 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, $\overline{v} \in G$, with the largest sum of edge weights. The maximum possible gain, $g_{\max}$, would then occur if $\overline{v}$, in the subdomain S_p say, were entirely surrounded by neighbours in a different subdomain, S_q say. The value for $g_{\max}$ is then given by $g_{\max} = \sum_{v' \in \Gamma(\overline{v})} |(\overline{v}, v')|$ (where $\Gamma(\overline{v})$ is the set of vertices $v' \in V$ adjacent to $\overline{v}$). Similarly the minimum gain is $-\sum_{v' \in \Gamma(\overline{v})} |(\overline{v}, v')| = -g_{\max}$. 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

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

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 (§3.2 & §3.3).

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 is generated of equal size to the current population, 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, [8]. 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 are the partitions) 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 best individuals 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), [23], and the Constrained Minimum Spanning Tree Problem (CMSTP), [22], 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: find a set of links (cut edges) subject to the constraint that they generate a partition with acceptable imbalance and that the sum of their edge weights is a minimum.

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, [8]. 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 be able to work with any graph partitioning heuristic which can deal with real (non-integer) edge weights. Indeed by suitable integer scaling the approach might even work with the more common partitioners which are restricted to integer edge weights.

The basic idea is that each vertex is assigned a bias greater than or equal to zero, 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, highest weight 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 become cut edges than those with higher weight.

It is easy to see for simple small meshes, simulating JOSTLE by hand, that particular partitions are achievable by choosing appropriate biases. However, we choose not to use the vertex biases as a representation for generating all partitions. Such a representation would have very high redundancy and would for the most part produce many partitions of low quality and hence little interest. Instead, for each offspring we explicitly construct a new set of biases, and hence a new partition, from one or more existing parent partitions. 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 search 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 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 of transfer of information to a partition of high quality can occur as required for a successful crossover operator.

Adding a constant to all bias values will have no affect on the partition produced. When considering their effect on graph contraction, only the relative ordering of the vertices by their biases and the edges at a vertex by their weights matters. However their relative size (over and above the effect on ordering) does alter what happens during the optimisation stage since the transfer of vertices between subdomains may proceed so as to decrease the sum of biased edge weights, but in fact *increase* the true total cut-weight. The greater the relative bias towards border vertices remaining so in the offspring, the greater the number that will be preserved on average, but at the cost of the offspring being more likely of poorer quality. The choice of a relative bias of 0.1 towards preserving border vertices is a value that has been shown by the experiments performed to give the right tradeoff between these competing effects.

Informally we are relying on JOSTLE to preserve common border vertices from different pairs of partitions that can be 'joined up' so as to produce a partition of high quality within the imbalance constraint, in effect finding an appropriate *linkage*, [8]. Linkage between genes is normally defined in terms of their distance apart on a chromosome, [20]. Nearby genes are referred to as tightly linked and are most likely transferred together into an offspring from a single parent under a traditional crossover operation and vice-versa. Specifying a linkage for the genes on a chromosome for a fixed crossover determines what information is likely to be transferred together from any parent. This needs to be chosen carefully if the resulting offspring are to have a good chance of high fitness and is known as the 'linkage problem'. We use the term linkage to directly mean the probabilities of pairs of cut edges from any parent being transferred together into an offspring. Clearly the linkage enforced by JOSTLE must be both dynamic and subtle to produce fit offspring partitions.

At each mating then, two, three or four mates were randomly chosen to crossover together, a range found to work well in trials on a number of graphs.

Figure 2 illustrates and motivates the aims of the crossover operator. Figures 2(a) & 2(b) show two possible (perfectly balanced) partitions of the example graph whilst Figure 2(d) shows the optimal partition of this graph into 4 subdomains. In Figure 2(c) the border vertices common to both partitions (a) & (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 2(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 retains information about common strengths of two or more parents whilst allowing further investigation of the search space.

Finally notice that Figure 2 (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 & C, the resulting biased graph will show common border vertices between

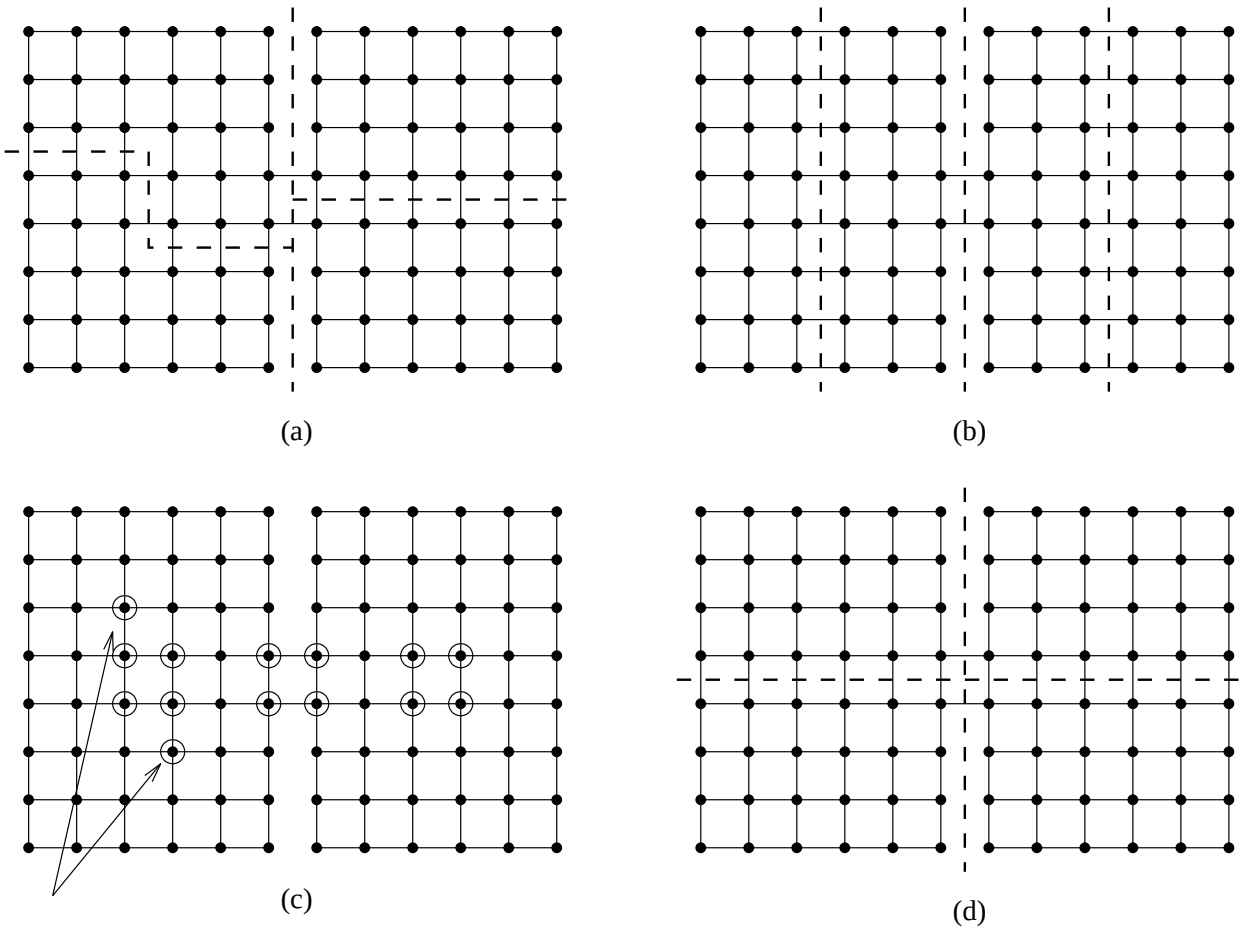

Figure 2: An illustration of the crossover operator: (a) & (b) partitions of an example graph, (c) common border vertices from these two partitions, (d) the optimal partition into 4 subdomains

A & B, B & C and C & A which means 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.

Motivations for this arrangement are the following:

(a) 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.

(b) 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. Hence a good place to look for another partition given an existing partition of good quality is within its trenches.

(c) It should be possible to find new partitions mainly within the trenches, that satisfy the balance constraint, since there is freedom for all subdomains to gain and lose similar numbers of vertices when boundaries are shifted to mainly parallel positions. Here we need to take into account that the width of a trench spans three edges and so allows some boundaries to move more than others. Clearly the extent to which this will hold depends on the graph structure and the number of subdomains and their shape.

(d) The trenches allows variations orthogonal to the optimisations performed by JOSTLE; i.e. there exist partitions for the most part lying in the trenches that will result from applying JOSTLE to bias values differing in their random parts only. This is because JOSTLE only considers the transfer of a limited number of vertices at a time between subdomains at any optimisation step. The movements of subdomain boundaries to adjacent parallel positions may require the transfer of larger numbers of vertices than JOSTLE allows.

Properties (a)-(d) taken together are desirable properties for a mutation operator since they should lead to the generation of 'nearby' partitions of similar, and hence sometimes superior quality.

The choice of the value 2.0 to bias JOSTLE towards placing subdomain boundaries in the trench is again a compromise figure. A larger value is likely to force more of the resulting border vertices in the trench but possibly at a cost of poorer quality partitions. Again this value was chosen by experimenting with a number of different graphs.

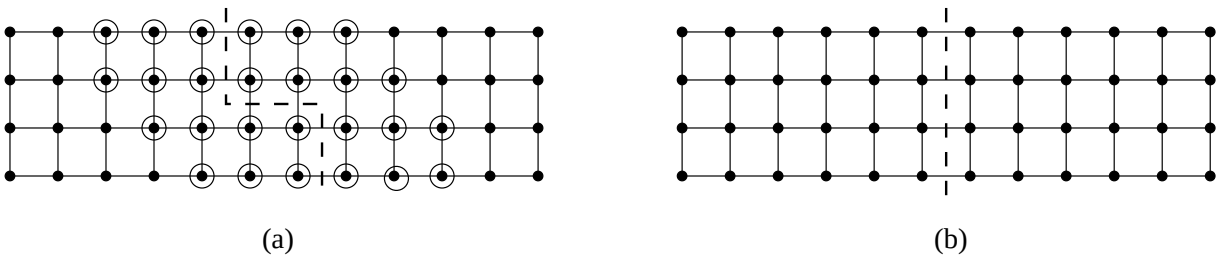

Figure 3: An illustration of the mutation operator: (a) an example partition with border vertices and their near neighbours ring, (b) the optimal partition into 2 subdomains

Figure 3 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 3(a) we have also ringed all the border vertices, neighbours of border vertices and neighbours of neighbours of border vertices. As for Figure 2 the lightest edges of the resulting biased graph are those between two ringed vertices whilst those between two unringed vertices will be substantially heavier. As a result JOSTLE will examine the partitions 'close to' the parent partition much more thoroughly. In this sense the global search properties of the multilevel partitioner are likely to be more focussed on these trench regions rather than over the whole graph.

3.4 Fitness function and imbalance

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 -1 times the number of cut edges times the imbalance. The imbalance was included in the fitness measure because the population of partitions can occasionally include partitions of greater imbalance than that sought. These can arise if JOSTLE fails to achieve the required balance and this usually occurs because of a limit on the number of calls to the load-balancer which is hard coded to prevent cyclic behaviour in controlling the two minimisation variables, the cut-weight and the maximum subdomain size. However, the code rarely reaches this limit and JOSTLE almost always achieves the required balance.

We also carried out some experimentation by explicitly creating imbalanced partitions. The required imbalance is an input parameter to JOSTLE when it is used to produce a new partition and we tried setting it to the required imbalance fifty percent of the time, and for the rest choosing an extra imbalance randomly from the set 1%, 2%, ... 10%. However, this did not lead to any noticeable improvement in the partitions found and the results in Section 4 all have the parameter set to either 0% or to 3% depending on the context.

This fitness function thus imposes a soft, but heavy penalty on partitions with greater imbalance; sufficiently heavy so that partitions within the balance constraint eventually dominated the population as evolutions progressed.

3.5 Genetic algorithm parameters

Due to the size of the meshes and the time required to execute JOSTLE, a fairly small population size of 50 was used. Small population sizes have been used successfully when hill-climbing is effective, and experiments with the mutation operator indicated that this was so.

Each new generation was produced as follows: 50 new offspring were produced by either crossover or mutation at a ratio of 7::3. Mating groups of individuals for crossover and candidates for mutation were selected randomly from the current generation, but with each parent participating in at least one trial. The union of the set of offspring and parents was then ranked according to the fitness of the individuals. The best 50 then formed 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, [8]. It is appropriate in this case because it encourages hill-climbing, and because most of the offspring generated are not of very high quality, [4].

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, [4], but lacks incest prevention and restarts. The experiments performed showed that the genetic algorithm was able to produce new best individuals until near the completion of the allotted evaluations.

3.6 Related work

There are a limited number of papers about topics related to the work presented here. In particular, schemes which use hybrid stochastic/local optimisation methods are of interest as are mesh partitioning techniques based on genetic algorithms.

Martin and Otto, [18], have also used a hybrid approach to graph partitioning. Their technique applied random changes to a partition, which was then subject to a local optimisation scheme (Kernighan-Lin) to improve it. Further changes and local optimisations were applied according to a simulated annealing scheme. However, the particular graphs used are not available for comparison.

Khan & Topping, [15], used a genetic algorithm for partitioning very small meshes (the coarse root meshes of a parallel mesh generator). However, rather than explicitly represently the partition, their approach used a population of cutting planes which bisected the finite element domain. A well-balanced partition was not sought by the technique, since it was designed for short run-times and thus used an estimation of the number of elements to appear in final refined subdomains.

Mansour and Fox, [16, 17], partitioned graphs with a genetic algorithm using a direct encoding, where the subdomain membership of each vertex was explicitly represented by the value of a gene. Since these values were unconstrained, partitions of arbitrary imbalance were possible. These genes were concatenated and subject to 2-point crossover. The imbalance constraint was progressively enforced during evolution through the use of a penalty term in the fitness function. Graphs of up to approximately 550 vertices were partitioned but again are not available for comparison.

Finally, a genetic algorithm using the same direct representation has also been applied to graph partitioning in the context of circuit partitioning, but was found to be outperformed by a mixed simulated annealing tabu search, [7]. In contrast to the above, the work described here uses an optimisation scheme (JOSTLE) as the basis of a crossover and mutation operator 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[2]. 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.

	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 1: A summary of the test meshes

The test graphs have been chosen to be a representative sample of small to medium scale real-life problems 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[3].

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. The degree information (the degree of a vertex is the number of vertices adjacent to it) gives some idea of the character of the graphs. These range from the relatively homogeneous dual graphs, where every vertex represents a mesh element, in these cases a triangle (2D) or tetrahedron (3D) and so every vertex has at most 3 or 4 neighbours respectively, to the non mesh-based graph such as vibrobox which has vertices of degree 120 and an average degree of 26.81 and bcsstk32 with maximum degree 215 and average 44.16. 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.

Table 2 shows the results of using the evolutionary search algorithm with a 0% imbalance tolerance (i.e. perfect balance is enforced) for four values of P (the number of processors/subdomains). The table shows the total weight of cut edges or cut-weight which we denote C_E^0, the superscript denoting the allowed imbalance and the subscript E denoting the evolutionary algorithm.

As mentioned previously (§2.1) graph partitioning algorithms can usually 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 therefore tested the evolutionary algorithm with various tolerances and Table 3 shows a comparison of the cut-weight results for the evolutionary search algorithm with 3% and 0% imbalance tolerances, C_E^3 and C_E^0 respectively. For each value of P, the first column shows the cut-weight allowing a 3% imbalance, C_E^3,

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

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

graph	$P=4$	$P=8$	$P=16$	$P=32$
uk	44	91	162	286
add32	37	76	129	234
crack	368	687	1124	1768
wing-nodal	3581	5443	8422	12297
vibrobox	19245	25593	34403	43786
4elt	327	556	968	1606
cti	977	1812	2915	4583
cs4	964	1496	2211	3189
bcsstk32	10578	22856	40275	66407
t60k	216	480	893	1480
wing	1666	2589	4198	6215
brack2	3090	7269	12324	18731

Table 2: The results of the evolutionary search algorithm with a 0% imbalance tolerance (i.e. perfect balance is enforced) showing the cut-weight C_E^0

	$P=4$		$P=8$		$P=16$		$P=32$	
graph	C_E^3	$\frac{C_E^3}{C_E^0}$	C_E^3	$\frac{C_E^3}{C_E^0}$	C_E^3	$\frac{C_E^3}{C_E^0}$	C_E^3	$\frac{C_E^3}{C_E^0}$
uk	42	0.95	82	0.90	154	0.95	265	0.93
add32	33	0.89	69	0.91	117	0.91	212	0.91
crack	360	0.98	678	0.99	1082	0.96	1679	0.95
wing-nodal	3566	1.00	5401	0.99	8316	0.99	12112	0.98
vibrobox	19245	1.00	24924	0.97	33864	0.98	43131	0.99
4elt	319	0.98	527	0.95	919	0.95	1537	0.96
cti	917	0.94	1716	0.95	2778	0.95	4403	0.96
cs4	943	0.98	1472	0.98	2126	0.96	3154	0.99
bcsstk32	9728	0.92	21553	0.94	38972	0.97	64318	0.97
t60k	213	0.99	467	0.97	852	0.95	1439	0.97
wing	1636	0.98	2567	0.99	4027	0.96	6033	0.97
brack2	2864	0.93	7080	0.97	11958	0.97	18020	0.96
Average		0.96		0.96		0.96		0.96

Table 3: A comparison of cut-weight results for the evolutionary search algorithm with 3% and 0% imbalance tolerances, C_E^3 and C_E^0 respectively

while the second column shows the ratio of cut-weight for 3% imbalance scaled by that for 0% imbalance, C_E^3/C_E^0. Thus the figure of 0.95 for the uk graph and $P=4$ means that the algorithm was able to find a partition 5% better if allowed a 3% imbalance tolerance. As can be seen, the improvement in quality for these tests is up to 11% and on average is around 4%.

4.1 Benchmarking of public domain packages

To demonstrate the quality of the partitions, we have compared the results in Tables 2 & 3 with those produced by several public domain partitioning packages including JOSTLE, [24], METIS, [13], and CHACO, [11]. 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) & CHACO 2.0 (October 1995). For METIS the algorithm chosen was `kmetis`, the multilevel k-way scheme, and for CHACO we used the multilevel KL method with recursive bisection and chose a coarsening threshold of 200.

Firstly Table 4 shows a comparison of the cut-weight results for the public domain version of JOSTLE compared to the evolutionary search algorithm. As in Table 3, for each value of P, the first column shows the JOSTLE cut-weight results allowing the default 3% imbalance, C_J^3, whilst the second column compares the results from JOSTLE scaled by the 3% imbalance results from Table 3, C_J^3/C_E^3. Thus the figure of 1.69 for the uk graph and $P=4$ means that the evolutionary algorithm was able to find a partition 69% 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 test. Nonetheless it is

graph	$P=4$		$P=8$		$P=16$		$P=32$	
	C_J^3	$\frac{C_J^3}{C_E^3}$	C_J^3	$\frac{C_J^3}{C_E^3}$	C_J^3	$\frac{C_J^3}{C_E^3}$	C_J^3	$\frac{C_J^3}{C_E^3}$
uk	71	1.69	106	1.29	180	1.17	315	1.19
add32	41	1.24	106	1.54	180	1.54	257	1.21
crack	413	1.15	751	1.11	1191	1.10	1804	1.07
wing-nodal	4055	1.14	5965	1.10	8947	1.08	12635	1.04
vibrobox	21844	1.14	30247	1.21	34521	1.02	45374	1.05
4elt	434	1.36	656	1.24	1012	1.10	1687	1.10
cti	1329	1.45	2086	1.22	3262	1.17	4683	1.06
cs4	1162	1.23	1588	1.08	2477	1.17	3330	1.06
bcsstk32	14887	1.53	25343	1.18	48395	1.24	74391	1.16
t60k	229	1.08	530	1.13	984	1.15	1588	1.10
wing	1844	1.13	2911	1.13	4681	1.16	6404	1.06
brack2	2999	1.05	7808	1.10	13164	1.10	19238	1.07
Average		1.26		1.20		1.17		1.10

Table 4: A comparison of cut-weight results for JOSTLE, C_J^3, against those of the evolutionary search algorithm, C_E^3, both with 3% imbalance tolerance

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 69%.

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.

graph	$P=4$		$P=8$		$P=16$		$P=32$	
	C_M^3	$\frac{C_M^3}{C_E^3}$	C_M^3	$\frac{C_M^3}{C_E^3}$	C_M^3	$\frac{C_M^3}{C_E^3}$	C_M^3	$\frac{C_M^3}{C_E^3}$
uk	52	1.24	116	1.41	195	1.27	303	1.14
add32	107	3.24	94	1.36	219	1.87	285	1.34
crack	474	1.32	784	1.16	1299	1.20	1910	1.14
wing-nodal	3801	1.07	6012	1.11	9161	1.10	12779	1.06
vibrobox	21461	1.12	28839	1.16	39240	1.16	45719	1.06
4elt	359	1.13	759	1.44	1104	1.20	1842	1.20
cti	1098	1.20	2287	1.33	3412	1.23	4649	1.06
cs4	1102	1.17	1738	1.18	2534	1.19	3424	1.09
bcsstk32	12967	1.33	25193	1.17	47052	1.21	76809	1.19
t60k	279	1.31	578	1.24	996	1.17	1649	1.15
wing	1997	1.22	3014	1.17	4456	1.11	6661	1.10
brack2	3216	1.12	7951	1.12	13082	1.09	19867	1.10
Average		1.37		1.24		1.23		1.14

Table 5: A comparison of cut-weight results for METIS, C_M^3, against those of the evolutionary search algorithm, C_E^3, both with 3% imbalance tolerance

Table 5 shows a similar comparison for METIS. Once again, for each value of P, the first column shows the METIS cut-weight results allowing the default 3% imbalance, C_M^3, whilst the second column compares the results from METIS scaled by the 3% imbalance results from Table 3, C_M^3/C_E^3. The overall comparison is similar to that arising from JOSTLE although there is one extreme example, the graph add32 for $P = 4$, where the METIS cut-weight is over three times worse than the evolutionary algorithm. The average difference in the quality ranges from 37% to 14% as P increases. It is difficult to spot any trends in the results other than that METIS does particularly badly on the add32 graph and best of all on the wing-nodal graph.

Table 6 shows a comparison of CHACO with the evolutionary algorithm in the same format as Tables 4 & 5. Note however that, since CHACO uses recursive bisection and thus produces partitions with perfect balance (i.e. 0% imbalance tolerance), it is fairer to compare it with the 0% imbalance results in Table 2 (rather than the 3% results in Table 3 as for JOSTLE & METIS). Again the quality is worse than

graph	$P=4$		$P=8$		$P=16$		$P=32$	
	C_C^0	$\frac{C_C^0}{C_E^0}$	C_C^0	$\frac{C_C^0}{C_E^0}$	C_C^0	$\frac{C_C^0}{C_E^0}$	C_C^0	$\frac{C_C^0}{C_E^0}$
uk	63	1.43	121	1.33	202	1.25	318	1.11
add32	53	1.43	133	1.75	215	1.67	340	1.45
crack	466	1.27	794	1.16	1246	1.11	1978	1.12
wing-nodal	3979	1.11	6145	1.13	9353	1.11	13272	1.08
vibrobox	26757	1.39	40198	1.57	48558	1.41	56380	1.29
4elt	384	1.17	682	1.23	1155	1.19	1745	1.09
cti	1006	1.03	1946	1.07	3091	1.06	4583	1.00
cs4	1124	1.17	1698	1.14	2518	1.14	3514	1.10
bcsstk32	17941	1.70	29949	1.31	49068	1.22	73992	1.11
t60k	244	1.13	526	1.10	994	1.11	1601	1.08
wing	2104	1.26	3409	1.32	4880	1.16	6800	1.09
brack2	4116	1.33	9183	1.26	14683	1.19	22610	1.21
Average		1.29		1.28		1.22		1.14

Table 6: A comparison of cut-weight results for CHACO, C_C^0, against those of the evolutionary search algorithm, C_E^0, both with 0% imbalance tolerance

that achieved by the evolutionary algorithm and the overall comparison is similar to those arising from JOSTLE & METIS; the average difference in the quality ranges from 29% to 14% as P increases. However 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 and the mutation & crossover operators contribute to finding the best partitions during an optimisation. Each such optimisation 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. In that sense one would 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) to run and only have one partitioning attempt. Therefore, in order to attempt to quantify the added value given by 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 §3.5).

graph	$P=4$		$P=8$		$P=16$		$P=32$	
	C_R^3	$\frac{C_R^3}{C_E^3}$	C_R^3	$\frac{C_R^3}{C_E^3}$	C_R^3	$\frac{C_R^3}{C_E^3}$	C_R^3	$\frac{C_R^3}{C_E^3}$
uk	42	1.00	89	1.09	164	1.06	281	1.06
add32	33	1.00	69	1.00	117	1.00	212	1.00
crack	362	1.01	689	1.02	1123	1.04	1748	1.04
wing-nodal	3619	1.01	5473	1.01	8534	1.03	12194	1.01
vibrobox	19430	1.01	25575	1.03	34027	1.00	43496	1.01
4elt	322	1.01	539	1.02	952	1.04	1586	1.03
cti	925	1.01	1812	1.06	2983	1.07	4477	1.02
cs4	1003	1.06	1580	1.07	2308	1.09	3245	1.03
bcsstk32	9837	1.01	22372	1.04	39768	1.02	65270	1.01
t60k	222	1.04	498	1.07	911	1.07	1509	1.05
wing	1780	1.09	2767	1.08	4354	1.08	6326	1.05
brack2	2938	1.03	7383	1.04	12301	1.03	18524	1.03
Average		1.02		1.04		1.04		1.03

Table 7: A comparison of cut-weight results for 50,000 graph variants with random edge weights, C_R^3 against the evolutionary search algorithm, C_E^3, both with 3% imbalance tolerance

Table 7 shows a comparison of the graphs with random edge weights against the evolutionary algo-

rithm in the same format as Tables 4-6. Overall the difference is not great, on average between 2% and 4% over the different values of P. However it should be noted that although this may only seem like a small improvement for the evolutionary approach (especially compared with the more dramatic comparisons with partitioning packages), it is a general rule of optimisation that the closer one gets to the optimum the more difficult it is to find an improvement. In this sense the evolutionary enhancements are dealing with the difficult task of guiding the algorithm towards the global minimum of the optimisation-space as opposed to just jumping around it at random.

Some interesting trends are also visible in the results. In particular evolutionary and random approaches do not differ much in the partitions they provide for the non-mesh based graphs (add32, vibrobox & bcsstk32). We speculate that this is a result of the irregularity of the graphs; possibly this may mean that local minima are not so 'close together' and so operators, both crossover and in particular mutation, are unable to help to find 'nearby' minima. Whether this implies that the evolutionary approach is not so good for such graphs is difficult to say; it may be that both approaches are able to find 'natural partitions' which may exist in these more irregular graphs simply by jumping around the optimisation-space and using the more local search characteristics of JOSTLE to reach a local minimum.

Another trend in the results would appear to be that the evolutionary search becomes better than random evaluation as V, the number of vertices, increases. This is easier to to provide a possible explanation for, since it seems likely that as the graph size increases the number of local minima (or 'good partitions') also increase. For the random approach to find the best of these we would really need to increase the number of evaluations as the graph size increases. This trend therefore suggests that the evolutionary algorithm is less dependent on the number of evaluations than a purely random journey around the optimisation space and that the crossover and mutation operators are having some appreciable effect.

5 The public domain partition archive

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. At the time of writing we measure the quality of partitions by the total weight of cut edges, however, given the discussion about appropriate metrics (§2.1) there is no reason why the archive should not be extended to incorporate partitions which minimise different cost functions.

There is less argument about the definition of balancing constraint but, to avoid any floating point problems, we define the target weight of a perfectly balanced partition with the following piece of C code:

```
int  target, vertex_weight, nprocs;
target = vertex_weight/nprocs;
if (vertex_weight%nprocs != 0)
     target += 1;
```

where `vertex_weight` is the total vertex weight, $|V|$, of the graph, `nprocs` is P the number of processors and `vertex_weight%nprocs` is just the C formulation for `vertex_weight` modulo `nprocs`. A graph is then said to be in perfect balance if the weight of each subdomain $|S_p|$ is less than or equal to `target`.

As stated previously, allowing a certain amount of imbalance can improve (sometimes dramatically) the partition quality. We therefore include in the archive partitions with 1%, 3% & 5% imbalance together with those in perfect balance (0% imbalance tolerance). Once again these are defined with integer arithmetic and a partition is said to be within a given tolerance `t`% if the weight of each subdomain, $|S_p|$, is less than or equal to `target_t` where

```
int  t, target_t, vertex_weight, nprocs;
target_t = ((100+t)*target)/100;
```

Note that this is slightly more generous than `((100+t)*vertex_weight)/(100*nprocs)`.

6 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 partitions especially when compared against state-of-the-art partitioning packages such as JOSTLE and METIS. As such we have used it to provide a public domain archive of partitions which can be used for benchmarking purposes.

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 (e.g. circuit partitioning).

References

[1] L. Altenberg. The Schema Theorem and Price's Theorem. In L. D. Whitley and M. D. Vose, editors, *Foundations of Genetic Algorithms 3*, pages 23–49. Morgan Kaufmann, 1995.

[2] S. T. Barnard and H. D. Simon. A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems. *Concurrency: Practice & Experience*, 6(2):101–117, 1994.

[3] T. N. Bui and C. Jones. A Heuristic for Reducing Fill-In in Sparse Matrix Factorization. In R. F. Sincovec *et al.*, editor, *Parallel Processing for Scientific Computing*, pages 445–452. SIAM, 1993.

[4] L. J. Eshelman. The CHC adaptive search algorithm: How to have safe search when engaging in nontraditional genetic recombination. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 265–283, San Mateo, 1991. Morgan Kaufmann.

[5] C. M. Fiduccia and R. M. Mattheyses. A Linear Time Heuristic for Improving Network Partitions. In *Proc. 19th IEEE Design Automation Conf.*, pages 175–181, IEEE, Piscataway, NJ, 1982.

[6] M. Garey, D. Johnson, and L. Stockmeyer. Some simplified NP-complete graph problems. *Theoret. Comput. Sci.*, 1:237–267, 1976.

[7] C. Gil, J. Ortega, A. F. Diaz, and M. G. Montoya. Annealing-based Heuristics and Genetic Algorithms for Circuit Partitioning in Parallel Test Generation. *Future Generation Comp. Syst.*, 14(5):439–451, 1998.

[8] D. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.

[9] A. Gupta. Fast and effective algorithms for graph partitioning and sparse matrix reordering. *IBM Journal of Research and Development*, 41(1/2):171–183, 1996.

[10] B. Hendrickson and T. G. Kolda. Graph Partitioning Models for Parallel Computing. To appear in *Parallel Comput.*

[11] B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In S. Karin, editor, *Proc. Supercomputing '95*, New York, NY 10036, 1995. ACM Press.

[12] G. Karypis and V. Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, 1998.

[13] G. Karypis and V. Kumar. Multilevel k-way Partitioning Scheme for Irregular Graphs. *J. Par. Dist. Comput.*, 48(1):96–129, 1998.

[14] B. W. Kernighan and S. Lin. An Efficient Heuristic for Partitioning Graphs. *Bell Systems Tech. J.*, 49:291–308, February 1970.

[15] A. I. Khan and B. H. V. Topping. Subdomain Generation for Parallel Finite Element Analysis. *Computing Systems Engrg.*, 4(4-6):473–488, 1993.

[16] N. Mansour and G. C. Fox. A Hybrid Genetic Algorithm for Task Allocation in Multicomputers. In R. K. Belew and L. B. Booker, editors, *Proc. 4th Int. Conf. Genetic Algorithms*, pages 466–473. Morgan Kaufmann, 1991.

[17] N. Mansour and G. C. Fox. Allocating Data to Distributed-memory Multiprocessors by Genetic Algorithms. *Concurrency: Practice & Experience*, 6(6):485–504, 1994.

[18] O. C. Martin and S. W. Otto. Partitioning of Unstructured Meshes for Load Balancing. *Concurrency: Practice & Experience*, 7(4):303–314, 1995.

[19] C. H. Papadimitriou and K. Stieglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, Englewood Cliffs, NJ, 1982.

[20] N. J. Radcliffe and P. D. Surry. Fitness Variance of Formae and Performance Prediction. In *Foundations of Genetic Algorithms 3*, pages 51–72. Morgan Kaufmann, 1995.

[21] H. D. Simon and S.-H. Teng. How Good is Recursive Bisection? *SIAM J. Sci. Comput.*, 18(5):1436–1445, 1997.

[22] A. J. Soper and S. McKenzie. 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*, pages 113–116, 1997. (2nd IEE Int. Conf. Genetic Algorithms: Innovations and Applications, Glasgow).

[23] C. L. Valenzuela and L. P. Williams. Improving Heuristic Algorithms for the Travelling Salesman Problem by Using a Genetic Algorithm to Perturb the Cities. In T. Back, editor, *Proc. 7th Int. Conf. Genetic Algorithms*, pages 458–464. Morgan Kaufmann, 1997.

[24] C. Walshaw and M. Cross. Mesh Partitioning: a Multilevel Balancing and Refinement Algorithm. To appear in *SIAM J. Sci. Comput.* (originally published as Univ. Greenwich Tech. Rep. 98/IM/35), 1998.

[25] C. Walshaw and M. Cross. Parallel Optimisation Algorithms for Multilevel Mesh Partitioning. To appear in *Parallel Comput.* (originally published as Univ. Greenwich Tech. Rep. 99/IM/44), 1999.

[26] C. Walshaw, M. Cross, R. Diekmann, and F. Schlimbach. Multilevel Mesh Partitioning for Optimising Domain Shape. *Int. J. High Performance Comput. Appl.*, 13(4):334–353, 1999. (originally published as Univ. Greenwich Tech. Rep. 98/IM/38).