
An Ant System Algorithm for Graph Bisection

Thang N. Bui

Dept. of Computer Science
Penn State Harrisburg
Middletown, PA 17057

Lisa C. Strite

Dept. of Computer Science
Penn State Harrisburg
Middletown, PA 17057

Abstract

This paper gives an algorithm for the graph bisection problem using the Ant System (AS) technique. The ant algorithm given in this paper differs from the usual ant algorithms in that the individual ant in the system does not construct a solution to the problem nor a component of the solution directly. Rather the collective behavior of the two species of ants in the system induces a solution to the problem. The algorithm also incorporates local optimization algorithms to speed up the convergence rate and to improve the quality of the solutions. The results achieved by this algorithm on several classes of graphs equal the best known results for the majority of graphs tested, and are very close to the best known results for the remainder.

1 Introduction

Let $G = (V, E)$ be a graph on n vertices, where n is even. A *bisection* of G is a partition of the vertex set V into two disjoint sets A, B of equal size, i.e., $A \cup B = V$, $A \cap B = \emptyset$, and $|A| = |B|$. Such a bisection is denoted by (A, B) . The *cut size* of a bisection (A, B) is the number of edges that have one endpoint in A and the other endpoint in B . The *graph bisection problem* is the problem of finding a bisection of minimum cut size for a given graph. The graph bisection problem is well-known to be \mathcal{NP} -hard [13]. It arises in a wide variety of problems including VLSI placement and routing, sparse matrix computation, and processor allocation [6][7][16]. Since the problem is \mathcal{NP} -hard, efforts have been concentrated on designing efficient approximation algorithms and heuristics for solving it. In this paper we use ideas from Ant System (AS) [11] to design an algorithm for the graph bisection problem.

Our algorithm incorporates several AS features as well as local optimization techniques and graph preprocessing. The algorithm was tested on five classes of graphs ranging in size from 500 to 5,252 vertices with average degrees from 2 to 36. The results were compared with the best known results for each graph as well as results from several other heuristic algorithms. For the majority of graphs tested, the algorithm produced the best known results. For the remaining graphs the results produced by the algorithm are very close to the best known solutions. A major advantage of this algorithm compared to other existing algorithms for the graph bisection problem is that our algorithm is very amenable to parallelization.

The rest of the paper is organized as follows. In Section 2 we provide some background information on the graph bisection problem and AS. We describe our algorithm in Section 3. The performance of the algorithms on the test graphs is described in Section 4 and conclusions are given Section 5.

2 Preliminaries

2.1 The Graph Bisection Problem

As mentioned above the graph bisection problem is \mathcal{NP} -hard and thus we do not expect to have a polynomial time algorithm for solving it. For special types of graphs there are polynomial time algorithms for solving it exactly, e.g., k -outerplanar graphs, for fixed k , or planar graphs whose optimal bisection is of size $O(\log n)$ [8]. However, the complexity of the problem on planar graphs remains an open question.

One approach to \mathcal{NP} -hard problems is to find efficient approximation algorithms. Currently, the best known polynomial time approximation algorithm for bisecting graphs can find a solution that is within $O(\sqrt{n} \log n)$ of the optimal [12]. It has been shown that it is \mathcal{NP} -hard

to find a bisection that is within an additive factor of $O(n^{1/2-\epsilon})$ of the optimal, for any $\epsilon > 0$ [5].

Since the best approximation algorithm still has a rather large approximation ratio and is rather complicated to implement, heuristics are often used for the graph bisection problem in practice. Heuristics are algorithms that do not have performance guarantee as approximation algorithms do. However, they usually are fast and produce solutions that are very good. Generally, one can classify the heuristic algorithms for bisecting graphs into two main groups: local methods and global methods. Local methods include the greedy algorithm, Kernighan-Lin, simulated annealing and multi-level algorithms [14][15][16]. Global methods include spectral algorithms, flow based algorithms and genetic algorithms [4][7][19]. The Kernighan-Lin algorithm is one of the first efficient algorithms for the graph bisection problem. We briefly describe the Kernighan-Lin algorithm here since we use it in our algorithm later on.

2.2 The Kernighan-Lin Algorithm

Kernighan-Lin is a local optimization algorithm for the graph bisection problem [16]. The algorithm starts with a bisection (A, B) , either created randomly or as the result of some other algorithm. The algorithm consists of a number of passes. During one pass of the algorithm it interchanges equal sized subsets of A and B . The subsets to be interchanged are selected by first ordering the vertices of A and B , say $a_1, \dots, a_{n/2}$ and $b_1, \dots, b_{n/2}$. The algorithm then selects k such that swapping a_1, \dots, a_k with b_1, \dots, b_k will give the greatest reduction in the size of the current bisection over all choices of k . This constitutes one pass of the algorithm. The bisection produced by this pass is then used as input to the next pass. The algorithm may run for a fixed number of passes or until no more improvement can be made from the current bisection.

2.3 Ant System

Ant System (AS) is a heuristic technique that seeks to imitate the behavior of a colony of ants and their ability to collectively solve a problem. For example, it has been observed that a colony of ants is able to find the shortest path to a food source by marking their trails with a chemical substance called pheromone[3][11].

The Traveling Salesman problem was the first problem to which the Ant System (AS) technique was applied [2] [11]. Other problems that have been the focus of AS as well as Ant Colony Optimization (ACO) [10] work include the quadratic assignment, network routing, ve-

hicle routing, frequency assignment, graph coloring, shortest common supersequence, machine scheduling, multiple knapsack and sequential ordering problems [18] [3].

In addition to the idea of finding shortest paths, the idea of territorial colonization and swarm intelligence can also be utilized in ant algorithms. Kuntz and Snyers applied these concepts to a graph clustering problem [17]. The organisms are called *animats*, reflecting the fact that the system draws ideas from several sources, not just ant colonies.

We combine these two ideas of animats following paths and forming colonies, together with the use of graph preprocessing and local optimization to develop an Ant System algorithm for the graph bisection problem, which we call ASGB. Our algorithm is described in the next section.

3 Ant System Algorithm for Graph Bisection

3.1 Main Ideas

The basic foundation of the algorithm is to consider each vertex in the graph as a location that can hold any number of animats. The animats can move around the graph by moving across edges to reach a new vertex. Each animat belongs to one of two species (called species A and B). However, animats of both species follow the same rules. Since movement of animats from vertices to vertices is an important part of the algorithm, the first step in the algorithm is to add edges, called *free edges*, to the input graph to make it connected if it is not. This is accomplished in two steps as follows.

First, we add the necessary free edges to connect all disconnected subgraphs. This is done by randomly selected a starting vertex and performing a depth first search until no new vertices can be reached. If any vertices were not reached, an edge is randomly placed between a vertex that was found in the search and a vertex that was not found. The depth first search then continues again and the process is repeated until all vertices have been joined to the graph.

Next, we add free edges between a number of vertices to improve the animats' ability to move and explore. The number of free edges added in this step is proportional to the number of vertices in the graph, but inversely proportional to the average degree. Thus a graph with a large number of vertices and a very low average degree will have the most free edges added while a graph with few vertices and a high average de-

gree will have the least free edges added. To select the locations of the free edges, a pool of possible pairs of vertices are randomly selected. The number of pairs in the pool is τ times the number of pairs that will actually be chosen. The distance between each pair is determined, in terms of the minimum number of steps needed to move from one vertex to the other, up to a certain maximum distance. Then pairs of vertices are selected randomly from the pool but in proportion to the distance between them. In other words, a pair of vertices that is furthest apart has the greatest chance of being selected.

Once the free edges are added, the animats consider them in the same manner as regular edges in selecting moves. However, when the cut size of a partition is calculated, the free edges are ignored.

The algorithm now starts by distributing α animats on the graph. (Note that the values of all parameters used in the description of the algorithm are given in Table 1.) Their species and location are chosen randomly. At any point throughout the algorithm, the configuration of animats on the graph constitutes a partition of the graph in the following way. Each vertex is considered to be colonized by one species. At a given time, it is said to be colonized by whichever species that has the greater number of animats on it. Ties are broken in a random order by assigning the vertex to the species that results in a lower cut size. The set of all vertices colonized by each species, called *colony*, makes up one half of the partition. This partition is not necessarily a bisection, since one colony may contain more vertices than the other. Thus, other techniques are used at certain points in the algorithm to ensure that the final solution is a bisection.

In addition, each vertex can hold pheromone. The two species produce separate types of pheromone, so each vertex has an amount of A pheromone and B pheromone.

The idea of the algorithm is for each species of animats to form a colony consisting of a set of vertices that are highly connected to each other but highly disconnected from the other colony. The result should be two sets of vertices that are highly connected amongst themselves, but have few edges going between the two sets.

The ASGB algorithm is divided up into σ sets each comprised of γ iterations. In each iteration a percentage of animats are activated. When an animat is activated, it adds an amount of pheromone to the vertex it is currently at based on conditions at the vertex. It then may die with a certain probability or it may reproduce with a certain probability and then moves to

a new vertex. In each iteration, these activations are performed in parallel. After each iteration, the graph is updated with the new information.

After each set, the configuration of the graph is forced into a bisection using a greedy algorithm and a local optimization algorithm is run to help speed up the convergence rate. During each set, the parameters, which include probabilities for activation, death, reproduction and birth are varied. The parameters are varied in such a way that at the beginning of the set, the colonies change a great deal and by the end of the set the colonies have converged to a stable configuration. The next set begins at the state where the previous set ended. However, if the animats follow their usual rules immediately, they will not be able to move away from the local optimum that has been reached. So, for all but the initial set, a *jolt* is performed for a certain number of the first iterations to help move the configuration, or distribution of animats on the vertices, away from the local optimal solution to which it had converged. The jolt allows animats to select moves randomly instead of following the normal rules for movement. The length of the jolt is changed during the algorithm. The first jolt lasts for ν iterations and for subsequent jolts the length decreases linearly until the last set where there is no jolt. The idea is that with each successive set, the bisection should come closer to the optimal bisection, and thus shorter and shorter jolts are needed.

After σ sets have been completed, the solution is the best bisection that has been achieved. This is usually the bisection found by the last set; however, occasionally the best bisection is found earlier.

In the following subsections we will describe in detail what occurs in one iteration, what occurs when an animat is activated and what occurs between sets. The full ASGB algorithm is given in Figure 1.

3.2 Iteration

An iteration of the algorithm consists of a percentage of the animats being activated and then performing the necessary operations in parallel. The probability of an animat being activated changes during the set. At the beginning of the set, more animats are activated during each iteration. By the end of the set, only a small percentage of the animats are activated in each iteration. The actual probability of activation is a sigmoid-like function. The function starts at a maximum of $\pi_{a \max}$ and ends at $\pi_{a \min}$.

After the activations of animats have been completed, ϵ percent of the pheromone on each vertex is evap-

Figure 1: An Ant System algorithm for graph bisection

```

Preprocess the graph to make it connected
Randomly add  $\alpha$  animats to graph
For  $set=1$  to  $\sigma$ 
  For  $time=1$  to  $\gamma$ 
    For each animat do (in parallel):
      Activate animat with probability  $a(time)$ 
      If activated
        Add  $p(animat, time)$  pheromone to
          the animat's location
        Die with probability  $\pi_d$ 
        If not dead
          If animat meets reproduction criteria
            Then reproduce with probability  $\pi_r$ 
          If  $time$  is in a jolt period
            Then select move randomly
          Else select move based on pheromone
            and connectivity
        Endif
      Endif
    Endfor animat
    Evaporate  $\epsilon$  percent of the pheromone from
      each vertex
  Endfor  $time$ 
  Convert configuration to a bisection with
    a greedy algorithm
  Run Kernighan-Lin local optimization
  Reduce total number of animats
  Equalize number of animats in each species
Endfor  $set$ 
Return best bisection found

```

orated. This prevents pheromone from building up too much and highly populated vertices from being overemphasized, which in turn prevents the algorithm from converging prematurely.

3.3 Activation of an Animat

When an animat is activated, it deposits pheromone on its current vertex, dies with a certain probability or reproduces with a certain probability, and then moves to another vertex. These operations are performed by the animat by using local information to make decisions.

3.3.1 Pheromone

The purpose of pheromone is to allow the algorithm to retain a “memory” of good configurations that have been found in the past. The formula for the amount of pheromone to be deposited is:

$$p(a, i) = \frac{a_{col}}{a_{total}} \cdot \frac{i}{\gamma}$$

Table 1: Parameter values

Param.	Value	Description
γ	1000	Number of iterations per set
σ	10	Number of sets
ν	50	Maximum jolt length
α	10000	Initial number of animats
$\pi_{a\ max}$	0.8	Maximum activation probability
$\pi_{a\ min}$	0.2	Minimum activation probability
π_d	0.035	Death probability
β_{init}	4	Expected number of animats born in first iteration
β_{final}	2	Expected number of animats born in final iteration
β_{range}	50%	Percentage range from average number of animats born
π_r	0.01	Reproduction probability
η	10	Max number of offspring per animat
μ	5	Number of moves needed before animat can reproduce
ψ_{stay}	20%	Percentage of offspring that stay on old location when not colonized by animat's species
ω_{pmin}	0	Minimum pheromone weight
ω_{pmax}	1	Maximum pheromone weight
ω_{cmin}	250	Minimum connection weight
ω_{cmax}	500	Maximum connection weight
π_{min}	0.1	Minimum probability for moving to a vertex
ρ	0.9	Reduction factor for returning to previous location
ψ_{swap}	75%	Percentage of vertices needed for swap
ψ_{maj}	90%	Percentage of animats needed for majority
ϵ	0.2	Evaporation rate
λ	1000	Pheromone limit
τ	50	Free edge factor

where a is the animat, i is the iteration number, a_{col} is the number of vertices adjacent to the animat's current location which are colonized by the animat's species, and a_{total} is the total number of vertices adjacent to the animat's current location. The idea here is for an animat to deposit more pheromone at a vertex if that vertex is highly connected to vertices colonized by its own species. Also, less pheromone is used in early iterations to allow for more exploration and more pheromone is used later on to emphasize exploitation. Even though the number of neighbors of a seems to relate more directly to the cut size, the amount of pheromone deposited is made proportional to the fraction a_{col}/a_{total} , to prevent the amount of pheromone at any vertex from growing out of control, even with evaporation.

There is also a limit to the amount of pheromone of each species that can be stored on a vertex. The limit

for a vertex is the product of the degree of that vertex and the pheromone limit parameter (λ). This allows densely connected vertices to accumulate more pheromone. The more highly connected a vertex is, the more essential it is that it is colonized by the right species. This is because a mistake on a highly connected vertex will mean a much greater cut size.

3.3.2 Death

Next, the animat may be selected to die. The animat die with probability π_d , which is fixed throughout the algorithm. However, the activation probability changes throughout the set, so that early in the set, more animats are activated, and therefore more animats die early in the set. The purpose of this is to have shorter life spans in the beginning, which allows more turnover and change in the configuration. Later in the set, the animats live longer and thus there is less change and the solution is able to converge.

3.3.3 Reproduction

If the animat is not selected for death, the algorithm proceeds to the reproduction step. The animat is selected for reproduction with fixed probability π_r . However, the number of new animats that are produced depends on time. In the first iteration of a set, the average number of animats born is β_{init} and it decreases linearly over time to β_{final} in the last iteration. The changing birth rate serves to allow more change in earlier iterations, in which animats live for shorter lengths of time. In later iterations, fewer animats are born, but they live longer. The actual number of animats born is selected uniformly at random over a range centered on the average birth rate for the iteration. The number of animats born can be up to β_{range} more or less than the specified average.

If the vertex on which the parent is located is colonized by its own species, the offspring animats are all placed on that vertex. However, if the vertex is colonized by the opposite species, only ψ_{stay} percent of the offspring animats are placed there. The remaining new animats will be placed on the vertex to which the parent animat moves in the next step. The rationale is that if the parent animat is already in its own colony but moves to another vertex, it should leave its offspring behind to help maintain the majority on that vertex. However, if the parent's species is not in majority, it should take most of its children to the new vertex in which it is trying to create a colony. The parent leaves some of its offspring behind however, so that some of its species remain at the vertex (in case that vertex really should

be part of their colony).

There are two other constraints on reproduction. First, there is a limit to how many offspring an animat can produce during its lifetime (η). This value is fixed throughout the algorithm and is the same for each animat. Once the limit is reached, the animat can no longer reproduce. This serves to prevent one species from taking over the graph and forcing the other species into extinction.

To prevent a species from overemphasizing a vertex through reproduction, the animats are not allowed to reproduce until they have made a set minimum number of moves (μ). This ensures that the graph is explored and that new configurations are created by the reproduction and movement rather than being inhibited by these operations.

3.3.4 Movement

An animat can move to any vertex which is connected to its current location by an edge. There are two factors used to select a move from the set of possible moves. For each vertex to which the animat could move, the connectivity to other vertices is examined. The animat should move to a vertex that is highly connected to other vertices colonized by its own species. This factor gives an indication of the current configuration of the graph. In addition, the animat should learn from the past and take into account the pheromone that other animats have deposited. Throughout the course of a set, these two factors are weighted differently. Initially, the pheromone is weighted at ω_{pmin} with the weight increasing linearly to ω_{pmax} . Conversely, the connectivity is weighted at ω_{cmax} to begin and decreases linearly to ω_{cmin} . In this way, the configuration of the colonies changes greatly in early iterations and over time learning is incorporated into the algorithm. These basic factors drive the animats to create colonies of highly connected vertices which are highly disconnected from the vertices colonized by the opposing species.

These factors are the basis of move selection. The probability of moving to an adjacent vertex is proportional to the two combined factors. Specifically, the factors are combined as follows to create a "probability" of moving to a specific vertex v :

$$pr(v) = cv_c + pv_p + \pi_{min}$$

where v_c is the number of vertices adjacent to v that are colonized by the animat's own species, c is the connectivity weight and $\omega_{cmin} \leq c \leq \omega_{cmax}$, v_p is the amount of pheromone of the animat's species on

vertex v , p is the pheromone weight and $\omega_{pmin} \leq p \leq \omega_{pmax}$, and π_{min} is a fixed amount added to prevent any probabilities from being zero.

In addition, one more factor is considered in selecting a move. To encourage the animats to explore more of the graph in the early sets, the probability of selecting the move which would result in the animat returning to its previous location is reduced. The factor it is reduced by starts at ρ and decreases linearly after each set until it reaches zero in the final set. Then the probability of moving to a connected vertex is the resulting value divided by the sum of values over all possible moves.

3.4 Between Sets

After each set of iterations, several other operations are performed. They help nudge the configuration into a bisection, improve the bisection through local optimization and then prepare the configuration for the next set.

First, the algorithm looks for “mistakes” the animats have made. Here the algorithm looks for vertices in which a very high percentage (ψ_{swap}) of the adjacent vertices are colonized by the opposite species. In these cases, the vertex is swapped to the other colony. This is achieved by changing the species of animats on the vertex until the new species attains ψ_{maj} percent of the animats. In most cases, few such vertices are found.

Next the colonies are manipulated to produce a bisection. As was discussed earlier, any given configuration of animats on the graph does not necessarily induce a bisection. Therefore, if one species is colonizing more vertices than the other, some vertices will have to be swapped to the other species. The vertices to be swapped are selected from the set of fringe vertices, that is, vertices that are adjacent to a vertex of the opposite colony. By only changing the colonizing species on fringe vertices, the algorithm continues in the direction the animats were heading. Vertices are selected to be swapped by making the greedy choice from amongst the fringe vertices.

Using the bisection produced by this greedy optimization, a weak version of the Kernighan-Lin algorithm is run. Since the quality of the result produced by the Kernighan-Lin algorithm depends largely on the quality of the bisection used as input, it produces little if any improvement in early sets. However, in later sets, after the animats have begun to converge upon a good solution, it usually improves the solution slightly.

Even though we now have a bisection, the number of animats on the graph may differ from the initial num-

ber of animats of both species. To prevent the animat population from growing unchecked the algorithm removes animats at random until the population size reaches its initial value. This may disrupt the colonies, however, this is not a problem since each new set begins with a jolt anyway.

Finally, to prevent one species from dominating the graph, the number of animats in the two species is equalized. This is done by adding animats to equalize the number of animats in each species. Usually this is a very small number and thus is not problematic in consideration of the previous operation (reducing the number of animats to the initial number). The new animats are added only to vertices where their own species is already in majority. Thus, this operation does not significantly alter the configuration of the colonies; it merely gives added strength to the colonies in which animats are added.

Following this operation, a new set is begun. Again, the time is initialized to 0 and all probabilities relating to time are reset. Thus, as the animats have converged on a possible solution, starting a new set allows the animats to move away from that solution in expectation of finding a better solution in case this solution was a local optimum. After σ sets have been completed, the solution is the bisection with minimum cut size.

4 Results

Using the parameter values listed in Table 1, the algorithm was tested on a total of forty graphs of five different types to determine its behavior on a wide selection of inputs. These graphs are used as a benchmark as they have been used to test a number of different graph bisection algorithms [1][7]. Thus the results can be compared with other algorithms. The graphs range in size from 500 to 5,252 vertices and have average degrees from 2 to 36. The algorithm was implemented in C++ and run on a Pentium III 800MHz with 256 MB RAM. For each graph, the algorithm was run for 100 trials. These results are given in Table 2 which also gives average running time in seconds for one trial of each graph. In this section, the five graph types are described and the results for different graph types are discussed.

4.1 Graphs Types

In [14], Johnson *et al.* described two classes of graphs that we use to test our algorithm. The first type, $Gn.p$, is a random graph on n vertices where an edge is placed between two vertices with probability p , independent

of all other edges. The expected vertex degree is then $p(n - 1)$. These graphs are a good test case as they have large optimal bisections. The second type, *Un.d*, is a random geometric graph on n vertices with expected vertex degree d . It is generated by selecting n points within the unit square which represent the vertices. An edge is placed between two vertices if their Euclidean distance does not exceed t . It can be shown that the expected vertex degree is $d = n\pi t^2$. This type of graph is highly clustered so it provides a very different test case than the previous class of graphs.

Three other graph types were proposed by Bui *et al.* in [4]. They defined the class of random regular graphs, *Breg n .b*, on n vertices with degree 3 having an optimal cut size b with probability $1 - o(1)$. These graphs provide an interesting test case because of they are sparse and have a provable unique optimal bisection with high probability. A grid graph, *Grid n .b*, on n vertices is a grid with known optimal cut size b . A variation of this type is *W-Grid n .b* in which the grid boundaries are wrapped around. This class of graphs is highly structured with good connectivity. The last class of graphs used is the caterpillar graph, *Cat n* , on n vertices with an optimal cut size of 1. It is constructed by starting with a spine, which is a straight line in which all vertices except the two ends have degree 2. Then to each vertex on the spine, called a node, we add six legs each of which consist of adding a vertex and connecting it to the node on the spine. If the number of nodes on the spine is even, the optimal cut size of 1 is found by dividing the spine in half. In addition, *RCat n* is a caterpillar graph in which each node on the spine has degree \sqrt{n} . Caterpillar graphs seem simple but are difficult for local bisection algorithms.

4.2 Comparison with other algorithms

The results of the ASGB algorithm are compared with results from three other algorithms in Table 2. The table compares the best result achieved by each algorithm in a fixed number of trials. For the ASGB algorithm, 100 trials were run for each graph with either 10 or 25 sets depending on the difficulty of the graph for the algorithm. Results for other algorithms reflect 1,000 trials as this was the data that was available from the sources. The last three columns of Table 2 contain the average and the standard deviation of the solutions returned by ASGB in 100 trials, as well as the average running time.

Battitti and Bertossi gave a Reactive and Randomized Tabu Search (RRTS) in [1]. The Multi-Start Kernighan-Lin (KL) consists of running the Kernighan-Lin algorithm 65 times on a new random

bisection each time. The final result is the minimum cut size of the 65 results. Since the results of KL are greatly affected by the quality of input, it is necessary to run it many more times to achieve good results since random bisections usually have poor cut sizes. This allows us to compare KL with other algorithms which normally would outperform it. The results for Multi-Start KL, Simulated Annealing (SA) and the best known results are taken from [7]. The sources provide results for most graphs in the benchmark set, however, when the results for a graph are not provided by the source, the corresponding entry is left blank.

Overall, ASGB got the best known solution for 27 of the 40 graphs tested. When the best known solution is not 1, the best solution returned by ASGB is less than 5% away from the optimal, usually much less. Generally, ASGB performed best when the input graphs have some clustering structure and enough connectivity that allow the animats to discover a good bisection, e.g., *Un.d*, *Breg n .b* and grid graphs. The caterpillar graphs have regular structure, but they do not have enough connectivity to allow the animats to explore the graph easily. The effect of random free edges added in the preprocessing step is not enough to overcome this deficiency. We did observe that if the number of sets is increased to 30 ASGB returns the optimal answer for almost all caterpillar graphs. It seems that the larger the caterpillar graphs, the more sets are required to get the optimal solution. For the class *G n .p*, ASGB either produced the best known solution or solutions that are within at most 5% of the best known.

Generally, we can see from Table 2 that ASGB is better than Multi-Start KL and SA and is very competitive with RRTS, noting that the data from these algorithms are from 1,000 trials for each graph.

5 Conclusion

An algorithm, called ASGB, using Ant System techniques with local optimization and graph preprocessing was developed for solving the graph bisection problem. Animats from two different species are placed on a graph and follow a set of local rules. The emergent behavior of the population following these rules, coupled with a local optimization, results in a bisection of the graph with low cut size. The results achieved were equal or very close to the best known results for the set of benchmark graphs. Even though the results achieved by ASGB were not always as good as the results of RRTS it seems that ASGB is more amenable

Table 2: Comparison of ASGB results with other algorithms

Graph	Best known	ASGB	RRTS	Multi-Start KL	SA	ASGB Avg	S.D.	Time
G500.005	49	51	51	52	52	57.52	2.38	139.36
G500.01*	218	218	218	220	219	225.29	3.51	412.72
G500.02	626	626	626	627	628	633.62	3.19	139.47
G500.04	1744	1744	1744	1744	1744	1752.98	3.56	197.14
G1000.0025*	95	97	96	101	102	103.82	3.57	426.96
G1000.005*	445	450	447	457	451	459.25	4.03	460.04
G1000.01*	1362	1367	1362	1376	1367	1377.51	3.89	542.38
G1000.02	3382	3385	3382	3390	3389	3399.88	7.63	222.83
U500.05*	2	2	2	5	4	14.28	5.75	462.01
U500.10*	26	26	26	26	26	61.66	16.93	494.16
U500.20	178	178	178	178	178	209.75	28.92	207.72
U500.40	412	412	412	412	412	461.43	54.13	297.55
U1000.05*	1	3	1	15	3	21.76	6.56	395.02
U1000.10	39	39	39	39	39	116.59	31.74	172.45
U1000.20	222	222	222	222	222	293.48	58.87	249.92
U1000.40	737	737	737	737	737	873.58	145.31	333.40
Breg500.0	0	0	0	0	–	0.00	0.00	132.48
Breg500.12	12	12	12	12	–	13.36	8.06	123.39
Breg500.16	16	16	16	16	–	16.68	4.82	138.87
Breg500.20	20	20	20	20	–	20.00	0.00	123.20
Breg5000.0	0	0	0	0	–	0.00	0.00	323.28
Breg5000.4	4	4	4	4	–	4.00	0.00	324.84
Breg5000.8	8	8	8	8	–	8.00	0.00	309.73
Breg5000.16	16	16	16	16	–	16.00	0.00	328.04
Grid100.10	10	10	10	10	–	10.06	0.45	137.39
Grid1000.20	20	20	20	20	–	23.64	8.69	135.74
Grid500.21	21	21	21	21	–	23.00	4.62	120.69
Grid5000.50	50	50	50	50	–	53.98	12.84	394.90
W-Grid100.20	20	20	20	20	–	20.00	0.00	120.37
W-Grid1000.40	40	40	40	40	–	43.98	13.76	143.98
W-Grid500.42	42	42	42	42	–	44.50	4.31	128.36
W-Grid5000.100	100	100	100	100	–	104.64	16.98	368.78
Cat.352	1	3	1	3	–	6.08	1.64	139.03
Cat.702	1	3	1	13	–	10.18	2.54	18.82
Cat.1052	1	7	1	25	–	13.14	3.01	155.51
Cat.5252	1	14	–	165	–	31.30	4.21	382.69
RCat.134	1	1	1	1	–	2.68	0.97	150.03
RCat.554	1	3	1	1	–	7.00	1.61	194.23
RCat.994	1	5	1	3	–	9.42	1.91	210.48
RCat.5114	1	7	–	17	–	14.50	2.81	528.28

* graph is run with 25 sets instead of 10

to parallelization than the RRTS algorithm.

Acknowledgement

The authors would like to thank Karthik Balakrishnan and the anonymous referees for helpful suggestions.

References

- [1] R. Battiti and A. Bertossi, “Greedy, Prohibition, and Reactive Heuristics for Graph Partitioning,” *IEEE Trans. on Comp.*, 48(4), 1999, pp. 361–385.
- [2] E. Bonabeau, M. Dorigo and G. Theraulaz, *Swarm Intelligence*, Oxford University Press, 1999.
- [3] E. Bonabeau, M. Dorigo and G. Theraulaz, “Inspiration for Optimization from Social Insect Behavior,” *Nature*, Vol. 406, July 6, 2000, pp. 39–42.
- [4] T. N. Bui, S. Chaudhuri, F. T. Leighton and M. Sipser, “Graph Bisection Algorithms With Good Average Case Behavior,” *Combinatorica*, 7(2), 1987, pp. 171–191.
- [5] T. N. Bui and C. Jones, “Finding Good Approximate Vertex and Edge Partitions is NP-Hard,” *Inf. Processing Letters* 42 (1992), pp. 153–159.

- [6] T. N. Bui and C. Jones, "A Heuristic for Reducing Fill-In in Sparse Matrix Factorization," Proc. of the Sixth SIAM Conference on Parallel Processing for Scientific Computing, 1993, pp. 445–452.
- [7] T. N. Bui and B. R. Moon, "Genetic Algorithm and Graph Partitioning," IEEE Trans. on Computers, 45(7), 1996, pp. 841–855.
- [8] T. N. Bui and A. Peck, "Partitioning Planar Graphs," SIAM Journal of Computing, 21(2), April 1992, pp. 203–215.
- [9] A. Colomi, M. Dorigo and V. Maniezzo, "Distributed Optimization by Ant Colonies," Proc. of the First European Conference on Artificial Life, New York: Elsevier, 1991, pp. 134–142.
- [10] M. Dorigo and G. Di Caro, "The Ant Colony Optimization Meta-Heuristic," in New Ideas in Optimization, D. Corne, M. Dorigo and F. Glover, Editors, McGraw-Hill, 1999, pp. 11–32.
- [11] M. Dorigo and L. Gambardella, "Ant Colony System: A Cooperative Learning Approach to the Travelling Salesman Problem," IEEE Trans. on Evol. Computation, 1(1), 1997, pp. 53–66.
- [12] U. Feige, R. Krauthgamer and K. Nissim, "Approximating the Minimum Bisection Size," Proc. of the Thirty-Second Annual ACM Symposium on Theory of Computing, 2000, pp. 530–536.
- [13] Garey, M. R. and D. S. Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, 1979.
- [14] D. S. Johnson, C. Aragon, L. McGeoch, and C. Schevon, "Optimization by Simulated Annealing: An Experimental Evaluation, Part 1, Graph Partitioning," Operations Research, Vol. 37, 1989, pp. 865–892.
- [15] G. Karypis and V. Kumar, "Analysis of Multilevel Graph Partitioning," Proc. of the 7th Supercomputing Conference, 1995.
- [16] B. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graph," The Bell System Tech J., 49(2), 1970, pp. 291–307.
- [17] P. Kuntz and D. Snyers, "Emergent Colonization and Graph Partitioning," Proc. of the Third Int. Conference on Simulation of Adaptive Behavior: From Animals to Animats 3, 1994, pp. 494–500.
- [18] V. Maniezzo and A. Carbonaro, "Ant Colony Optimization: An Overview", in Essays and Surveys in Metaheuristics, C. Ribeiro editor, Kluwer Academic Publishers, 2001, pp. 21–44.
- [19] P. Merz and B. Freisleben, "Fitness Landscapes, Memetic Algorithms and Greedy Operators for Graph Bi-Partitioning," Evolutionary Computation, 8(1), 2000, pp. 61–91.