Solving combinatorial problems by message passing

Ben Young

May 31, 2001

Abstract

In this paper the use of the sum-product algorithm in solving the maximum independent set problem is studied. The algorithms speed, and quality of solution is measured, as well as the conditions under which the algorithm fails. It is found that while the algorithm is very fast, it can only be applied to a very limited sub-set of possible graphs, namely those with an edge density of approximately $p < \frac{4}{n-1}$, where n is the number of vertices in the graph. It is also found that, for graphs less dense than this the algorithm performs well, and typically finds an independent set at least 80% of the size of the maximum independent set.

A number of adaptations of the algorithm were introduced. The first was a set of random perturbations to the weights to try and break the symmetry of the problems. This was found to have a small positive effect on the results. The second adaptation was introducing an adaptive algorithm for the free parameter β. This was found to significantly improve the quality and range of the results over a fixed β algorithm.

Finally a simple genetic algorithm was introduced for comparative purposes. The sum-product algorithm was found to be less useful for most graphs, but better for low density ones.

Contents

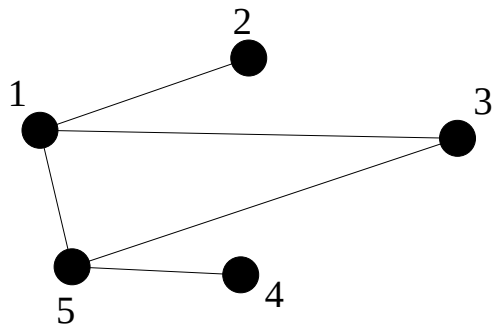


Figure 1: A simple graph

1 Introduction

The maximum independent set problem is one of a group of important mathematical problems, known as NP complete. This means that the once a solution has been found, it can be checked in polynomial time. The current thinking in the mathematical community is that the solution can actually only be found in exponential time, and because of the problems that this entails, finding good approximation techniques has become very important. By 'good' we mean that the approximation method will give a valid solution that is correct to within some factor (preferably high) of the true answer. For example: The travelling salesman problem is a very famous example of an NP complete problem. The task is to find the shortest route that visits each city once and only once. A good approximation algorithm would find a route which is 'close' to the correct solution, and therefore very useful in real world situations, but in significantly less time.

1.1 Definitions

A graph G is an object made of *vertices* and *edges*, so that $G = (V, E)$. A vertex is also known as a node. An edge is simply an object that connects two vertices together. For this paper it is assumed the the edges are undirected i.e. they can be traversed in either direction.

For each vertex $i \in V$ there is a *weight* associated w_i which must be positive. The *adjacency matrix* of G is defined by $a_{ij} = 1$ if $(i, j) \in E$ i.e. $a_{ij} = 1$ when vertices i and j are connected by an edge. The *complement* graph of G is defined to be $\bar{G} = G(V, \bar{E})$ so that if an edge is in E, it isn't in $\bar{E}$, and if it isn't in E, it is in $\bar{E}$.

The *degree* of a vertex i, is defined as the size of the subset of the graph which is adjacent to vertex i. This can therefore be written as:

$$D_i = \sum_j a_{ij} \tag{1}$$

A *clique* C of the graph is a subset of the G such that all its members

are adjacent to each other. The maximum clique is the clique with the most members. A *maximal* clique is a clique to which no members can be added without it ceasing to be a clique i.e. it is not a subset of any larger clique. In figure 1 for example, the subsets (2,4), (1,5), (1,2) and (1,2,3) are not a clique, a clique, a maximal clique, and the maximum clique respectively.

An independent set of the graph is a subset of the graph such that no members are adjacent to each other. The maximum independent set is the independent set with the most members. In figure 1 the subsets (1,5), (2,4) and (2,3,4) are not an independent set, an independent set and the maximum independent set respectively.

It is obvious that if C is the maximum clique of G then it is the maximum independent set of $\bar{G}$, and so any algorithm that finds the maximum clique can also find the maximum independent set, and the problems are interchangeable.

The weighted version of the maximum independent set problem takes into account the weights of each vertex. Here the task is to maximise the sum of the weights of the vertices in the independent set. For instance, in figure 1 if the weights of the vertices were 10, 1, 1, 1, 1 for each in turn then the best solution would be vertices 1 and 4, as they form an independent set which maximise the weights. The unweighted case can be considered as a special case where all the weights are equal to 1.

1.2 Motivation

So what are the reasons for wanting to solve the maximum independent set problem?. Firstly, the problem has a number of similarities with other problems, such as the class known as SAT. These problems are to do with constraint satisfaction and have many real world applications. This means that if a approximation method works well on the maximum independent set problem, then it can probably be adapted to help solve many other NP complete problems as well.

Secondly, the problem does have a number of real world applications of its own (see [1] pages 41-47). For instance, in coding theory the task is to find a set of codewords that are as 'different' from each other as possible, so that after they have been passed through the channel they are likely to still be distinguishable. One way to do this would be to represent each possible codeword as a vertex in the graph, and then link all codewords which are likely to be confusable after being passed through the channel by an edge. Finding the maximum independent set of this graph is then the same as finding the best non-confusable set of codewords.

There are many other uses in areas such as computer vision and fault diagnosis, and some more esoterical mathematical areas.

1.3 Existing methods

There are really two different methods of attempting to solve the problem; exact methods and heuristics. The methods described here are for the unweighted case but can be modified to work in the weighted case.

For the exact algorithms there are a number of different methods which have been tried.

Enumerative algorithms work by testing each possibility explicitly. These can be made more useful by using a *backtracking* method which removes some of the redundancy involved in generating the same clique. Other methods involve decomposing the graph into a number of subgraphs (chosen so that every clique is contained completely by at least one subgraph), and then finding the cliques (which is equivalent to finding the independent sets of the complement graph). The best performance of an enumerative algorithm was that produced by Tomita which had a time complexity of $O(3^{n/3})$ [1].

Another type of exact algorithms used for the unweighted case are known as the *branch and bound* methods. These work by breaking the problem down into a set of smaller subproblems and then branching between them. A recursive algorithm for the maximum independent set has been produced by this method which has a time complexity of $O(2^{n/3})$ [1], which is much faster than enumerative methods. The standard benchmark for both exact and heuristic algorithms is DFMAX, an exact solver based on the branch and bound method.

For further details on exact algorithms see [1].

When it comes to heuristic algorithms, there are a huge variety of different approaches:

- Standard genetic algorithms have been applied to the problem with varying degrees of success. A typical fitness function is, used in [2] is:

$$f(\bar{x}) = \sum_i x_i - n \cdot \sum_{i,j>i} a_{ij} x_i x_j \tag{2}$$

  where $\bar{x}$ is a binary string with $x_i = 1$ if vertex i is in the set. This penalises sets which are not independent sets.

- Sequential greedy heuristics are very simple algorithms. They work by repeatedly adding the best candidate and removing the worst from the set until a maximal set is formed. These can be very fast, but are unlikely to find the true maximum independent set.

- Simulated annealing: This is a method based on a physical interpretation of the system. An 'energy' function f is designed so that it is minimised by the desired solution. Random states are then generated from the current one. If the energy decreases then the new state is accepted. If the energy function increases then it is accepted with

probability $exp(\Delta f/\tau)$, where τ is the 'temperature' of the system. Then as the simulation is run, the temperature is gradually lowered until a stable state is reached.

This method seems to be very effective and was one of the best heuristics at the 1993 DIMACS[1] challenge, a competition to find the best heuristics for the problem.

- Neural networks and Hopfield networks [6].
- Replicator equations, as used in game theory.

For descriptions of more heuristic algorithms see [4], [5] and [1].

2 The Sum-Product algorithm

The first documented use of the sum-product algorithm was by Gallager [10] in 1963. He used it as the decoder for a set of low density parity check codes now known as Gallager codes. Now the algorithm is used in a wide variety of problems such as Bayesian networks, logic circuits, signal processing, Markov chain processes and computer vision. In fact the algorithm can be applied to almost any problem that can be expressed in terms of a factor graph [8]. A factor graph is a representation of how a global function of many variables can be decomposed into a product of many local functions of fewer variables e.g.

$$g(x, y, z, a, b, c, n, m, l) = f_A(x, y, z) f_B(a, b, c) f_C(n, m, l) \tag{3}$$

It takes the form of a bipartite[2] graph, made up of variable nodes (or vertices) and function nodes. Figure 2 shows a simple example of a factor graph.

The actual sum-product algorithm on the graph is then simply: *The message sent from a node v on edge e is the product of the local function at v with all messages received at v on edges other than e. If v is a variable node then the local function is simply unity* [7].

So how does this apply to the maximum independent set problem? The graphs on which we want to find the maximum independent set are not in general bipartite. If they were then finding the set would be trivial, as it would simply be the largest of the two separate subsets of the graph. Instead, it is possible to convert any graph into a bipartite one by simply counting all existing nodes as variable nodes and placing a function (or check) node on every edge. Thus every variable node will have a check node

[1] Center for Discrete Mathematics & Theoretical Computer Science.

[2] A bipartite graph is one made of two distinguishable sets, where every element of each set only connects to elements of the other set.

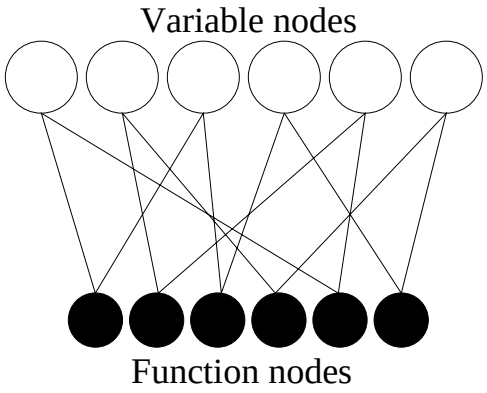


Figure 2: A example factor graph showing function nodes and variable nodes.

between it and the next variable node. We can then use the message passing algorithm to calculate the marginal posterior probability of each variable node being in the maximum independent set of the original graph. This is not necessarily going to converge to a definite result as the sum-product algorithm is only guaranteed to work if there are no cycles in the graph, and there will definitely be many cycles in a typical random graph. However although the algorithm will not stop on its own it may still converge enough to produce good results, as shown in the decoding of the Gallager codes [9].

The algorithm works with each variable node passing a message to its checks, stating how likely it "thinks" is that it is in the maximum independent set. Each check node then translates the message and passes it on to the next variable node. The translation takes the form of changing a message from "It is *this* likely that I am in the set." to "Given how likely it is that I am in the set, this is how likely it is that you are in the set."

The algorithm uses the following terms:

- $Q_{ij}^{(x)}$ is the message that variable node i sends to check node j. It is the probability that i is in state x (given the information it has received from all $j' \neq j$), with $x = 1$ meaning the probability that the node is in the set.
- $R_{ji'}^{(x)}$ is the message that check node j sends to variable node i'. It is the probability that, given the information j has received, i' is in state x.
- $P_i^{(x)}$ is the posterior probability that i is in state x given the information it has received.
- $g_i^{(x)}$ is the prior probability that node i is in state x.

The algorithm then works in the following way:

Step 1. The messages are initialised

$$Q_{ij}^{(1)} = g_i, \quad Q_{ij}^{(0)} = 1 - g_i \tag{4}$$

for all j.

Step 2. The messages are translated:

$$R_{ji'}^{(1)} = Q_{ij}^{(0)3} \tag{5}$$

$$R_{ji'}^{(0)} = 1 \tag{6}$$

These equations can be interpreted by the fact that the higher the probability that i isn't in the set, the higher the probability that i' is. However there is no penalty for i' not being in the set.

Step 3. Calculate the posterior probability:

$$P_i^{(x)} = \frac{g_i^{(x)} \cdot \prod_j R_{ji}^{(x)}}{Z} \tag{7}$$

where Z is a normalising constant. This works out the overall probability that given the information so far i is in the set. This probability is used as the stopping condition for the algorithm. When the posterior probability converges to a certain threshold the algorithm is stopped and the result taken.

Step 4. The new messages are formed:

$$Q_{ij}^{(x)} = \frac{g_i^{(x)} \cdot \prod_{j' \neq j} R_{j'i}^{(x)}}{Z'} \tag{8}$$

This looks similar to the posterior probability, but in this case when calculating the message back to the check node, the previous message from that node is ignored.

Step 5. Go to **Step 2.**

2.1 Implementation

As can be seen in the above description of the sum product algorithm, in this particular case the message passing to the check nodes is actually a redundant step. Therefore, **step 2** and **step 4** can actually be compressed into a single pass, and so the algorithm takes a particularly simple form:

$$Q_{ij}^{(1)} = \frac{g_i^{(1)} \cdot \prod_{j' \neq j} Q_{j'i}^{(0)}}{Z''} \tag{9}$$

[3]Because each check node is on the edge between only two variable nodes, knowing i and j uniquely defines i'.

$$Q_{ij}^{(0)} = \frac{g_i^{(0)}}{Z''} \tag{10}$$

where Z'' is the normalising constant *s.t.* $Q_{ij}^{(1)} + Q_{ij}^{(0)} = 1$.

Also, in the implementation of the algorithm, instead of explicitly defining the $g_i^{(x)}$ constants, they were defined as:

$$g_i^{(x)} = \frac{exp(\beta \cdot w_i)}{\alpha} \tag{11}$$

where again α is defined so $g_i^{(0)} + g_i^{(1)} = 1$. This was done so that the prior probabilities could be varied over a number of different scales, and also because, as seen later the β variable does indeed have some of the properties associated with temperature, as used in methods such as simulated annealing. Also this allows weights w_i to be introduced, and so the algorithm can be applied to the weighted form of the problem.

3 Experimental Results and Discussions

3.1 Single graph experiments

The first set of results obtained, were a set of tests on simple, randomly generated graphs. The graphs were created so that the probability of any two edges being linked by an edge was p. This is known as the *density* of the graph[4]. The density of a graph is directly related to the *mean degree* $< D >$[5] by,

$$< D >= (n-1) \cdot p = \frac{2 \cdot |E|}{n} \tag{12}$$

where n is the number of vertices in the graph, and $|E|$ is the number of edges.

It emerged from these tests that there are only two end states for the algorithm. The first is for the algorithm to converge towards a steady state solution. This typically occurs within 20 iterations and is likely to be much quicker than this. An example of this type of convergence is shown in figure 4.

The second end state for the algorithm is for it to enter into an "alternating" state. This is where the algorithm flips between two different states every iteration. Note that this state *is* stable, it is not a rounding error nor is it just "converging very slowly". An example of this type of end state is shown in figure 6.

To demonstrate the difference between simply converging slowly, and the alternating state, a final example is shown in figure 8. For this graph

[4]In the results, the value quoted as p has been calculated from the graph itself, as a finite size graph will not have exactly the density requested.

[5]Where the mean degree is simply the mean of the degree over the vertices.

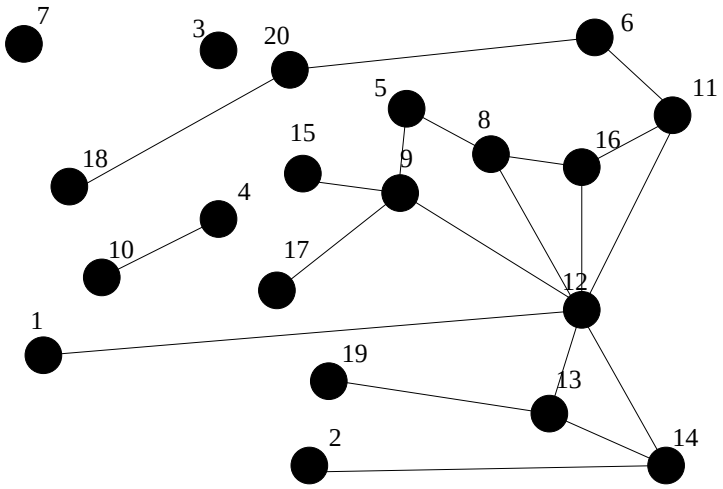


Figure 3: An example graph. The mean degree is 1.9.

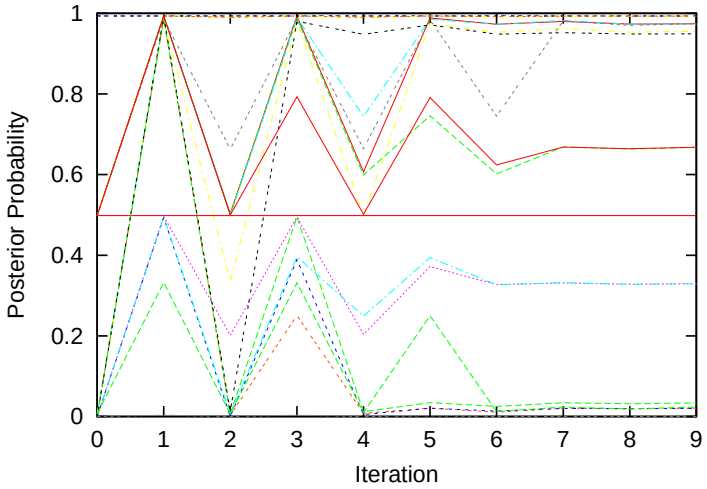


Figure 4: The convergence of the algorithm, $\beta = 5$. Each line represents the posterior probability of a vertex varying as the algorithm proceeds. The algorithm converges within 8 iterations.

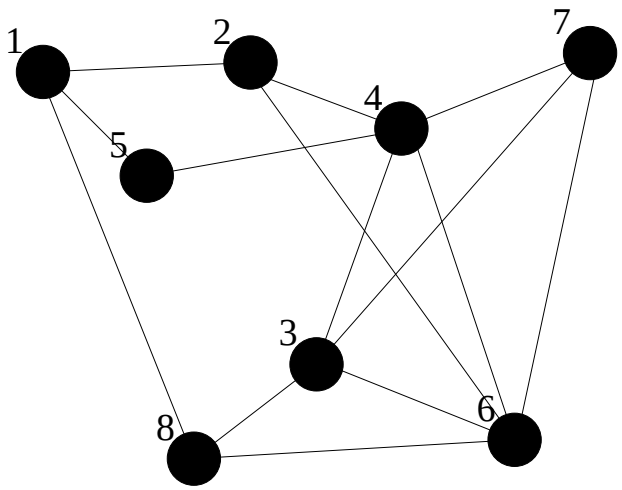


Figure 5: An example graph that demonstrates alternating behaviour. The mean degree is 3.5.

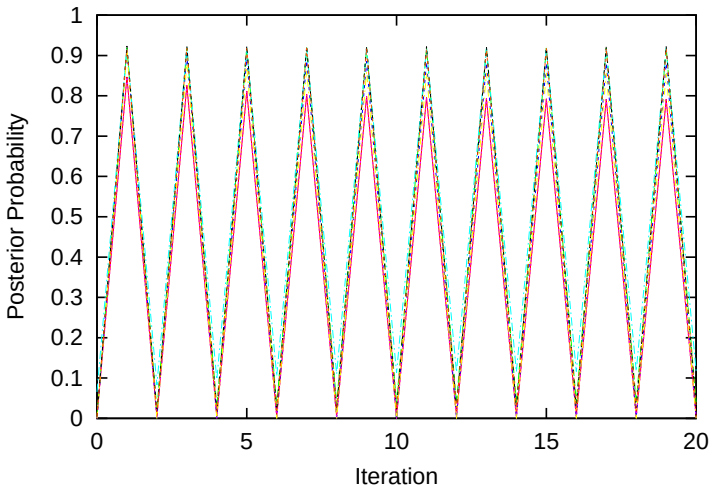


Figure 6: The alternating state of the algorithm $\beta = 2.5$.Each line represents the posterior probability of a vertex varying as the algorithm proceeds.

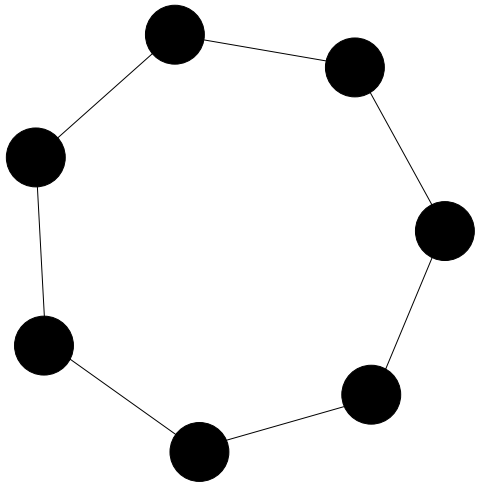


Figure 7: An example graph that demonstrates slow converging behaviour. The mean degree is 2.

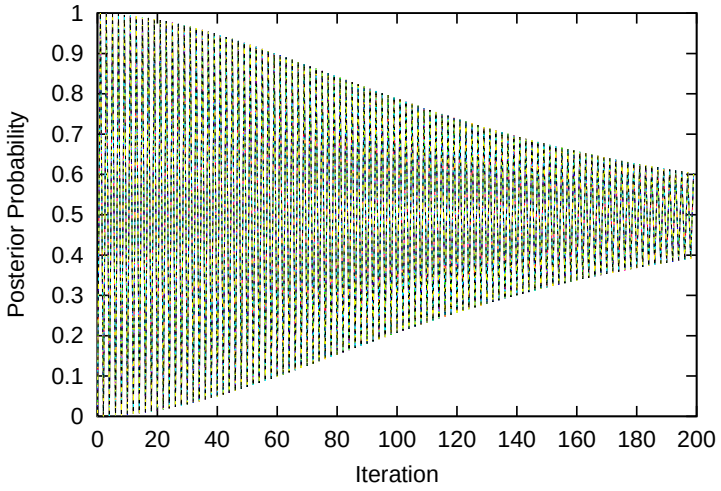


Figure 8: The slow convergence of the algorithm $\beta = 9.0$.Each line represents the posterior probability of a vertex varying as the algorithm proceeds.

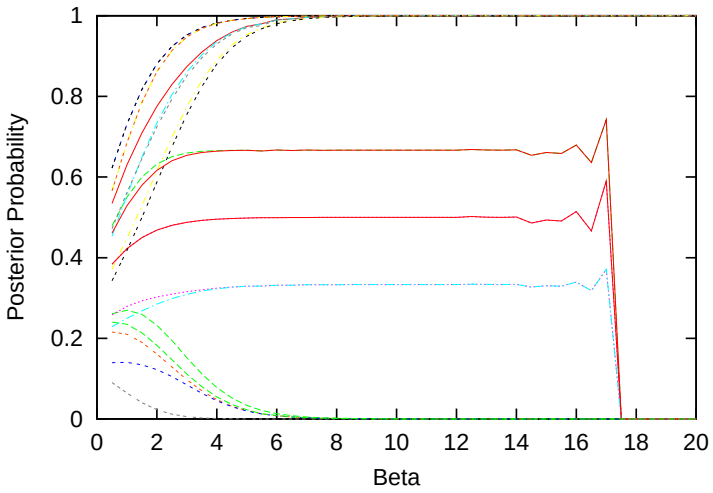


Figure 9: The variation of the posterior probability with β. Each line represents the final posterior probability of a particular vertex, over a number of different runs with varying values of β.

the algorithm has no way of distinguishing between the vertices, due to the symmetry of the graph. Because of the it converges very slowly, as the only converging factor is the fact that the prior is not equal to one.

From these results, stopping conditions for the algorithm were developed. Only two such conditions are required: One for when the algorithm converges, the other to catch when the algorithm goes into an alternating state. This is born out by all the other results in this paper, where the algorithm has always stopped, it has never simply run out of iterations.

The final test carried out on the single graphs was to discover what effects varying β has on the results for the graph. An example of the results of this test are shown in figure 9. In this case the graph used was the one in figure 3. This result shows that increasing β has the effect of increasing the difference in the posterior probability between vertices considered in the set, and those not in the set i.e. improving the quality of the solutions. Notice that at about $\beta = 14$ the lines become unstable. This is simply a result of equation 11, where, for large values of β the floating point routines reach their limit of numerical accuracy.

3.2 Convergence conditions

The next set of tests run were to attempt to find the general properties of the algorithm when run on any graph. To do this a large number of random graphs were generated with the desired properties for each test (size and density). Then the algorithm was run on each graph until it had converged or was alternating, and the results collated.

For the first test, the conditions under which the algorithm alternated

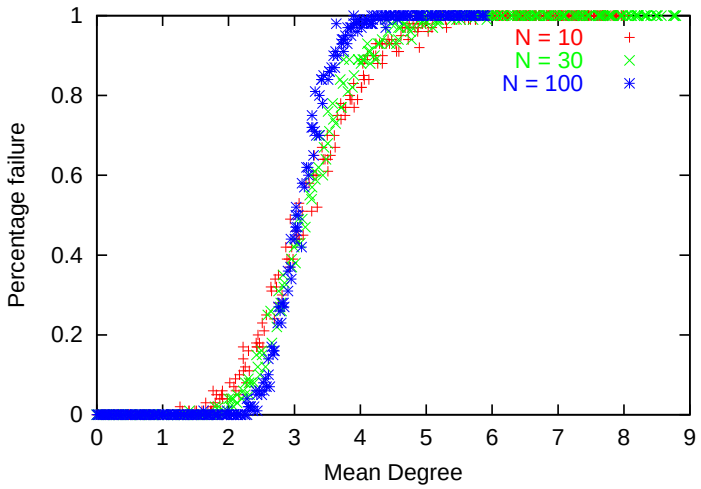


Figure 10: Conditions under which the algorithm alternates. Each point represents the fraction of 100 runs of the algorithm which did not converge. In this case the value of β used was 1.3.

were investigated. It became very quickly became obvious that it is not the density of the graph that is important, but the mean degree. This can be seen in figure 10. There is a very clear transition between graphs which converge to a steady state result, and those which alternate. The results also show that the action of the algorithm is almost independent of the size of the graph, as the transition point remains the same, even if the transition zone narrows. This indicates that for many properties of the algorithm the dependence on graph size does not need to be investigated, but their dependence on mean degree does. The makes sense as the algorithm is a local one.

The only other variable that the failure of the algorithm to converge could depend on is β. Figure 11 shows this dependence. Lower values of β push the transition to higher values of the mean degree. This is very useful as the best algorithm is one which converges for the widest variety of graphs. There is however a caveat, as for low values of β the algorithm does not give such confident predictions (figure 9.). This motivates having a β value which varies adaptively over the course of a single run, to get the best possible solution over the widest range of graphs, and a form of this is implemented later in the paper.

3.3 Timing

Possibly the most important motivation for designing new heuristic algorithms for the maximum independent set problem is that of speed. The next tests measure the speed of the sum-product algorithm under a number of conditions.

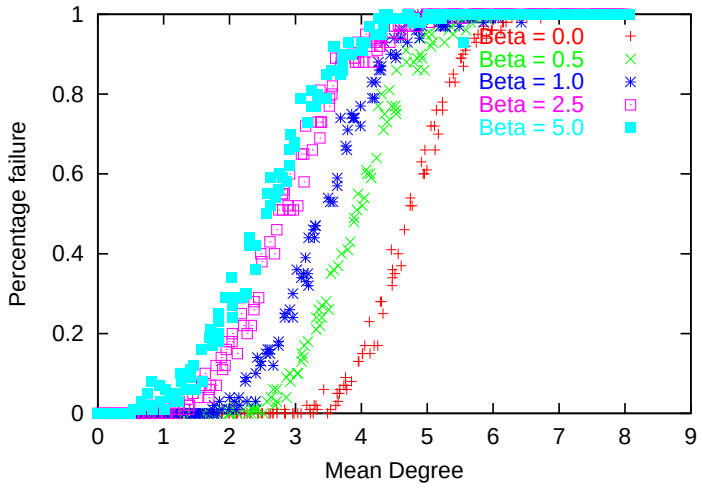


Figure 11: Conditions under which the algorithm alternates. Each point represents the fraction of 100 runs of the algorithm which did not converge. In this case the value of β used was 1.3.

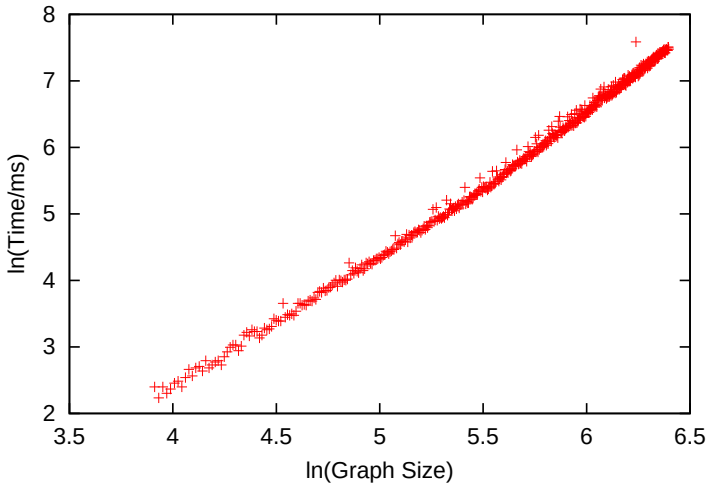


Figure 12: Variation in the speed of the algorithm with graph size.

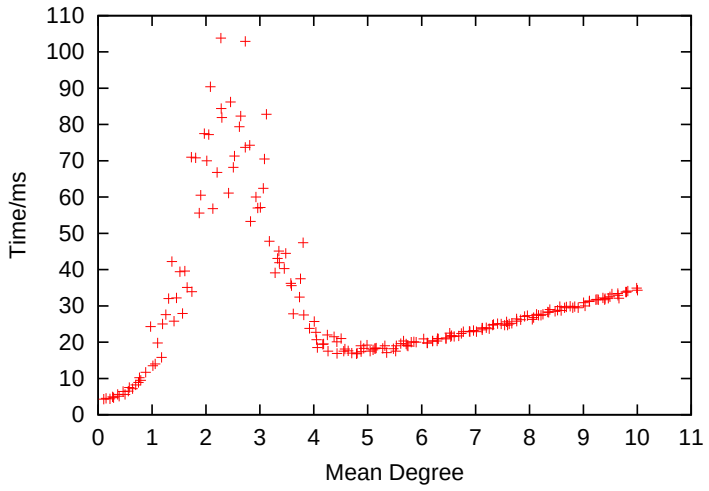


Figure 13: Variation in the speed of the algorithm with mean degree. Each point is an average over 100 runs of graphs, each of size 60.

The first test (figure 12.) measures the variation of the speed of the algorithm with graph size. Each point is an average over 30 graphs generated at the same size and with mean degree 1.5. The algorithm was run with a β value of 1.0. The results were taken so that if the graph alternated the time to alternation was also measured. This is because it is important to know how long it takes for the algorithm to fail to find a solution, as well as succeed. The best fit of the results gives the time proportional to n^2, which is consistent with the way the algorithm works. This is far superior to the possible exponential solution time of the exact solvers.

The next set of results in figure 13 show the variation in time with mean degree. These results are particularly interesting as they show an easy-hard-easy transition. For low values of the mean degree the algorithm finds the solutions very quickly. This is because in most cases the solution is actually trivial. As the mean degree increases, certain symmetrys and cycles are introduced into the graph and the algorithm takes longer to converge. As it increases further, it reaches the point where some of the graphs give alternating solutions. At high enough mean degree every graph starts alternating almost immediately and so the algorithm stops very quickly.

3.4 Quality of solutions

The most important test of any algorithm is: *Does it actually find good solutions?* To find this out an exact solver was implemented. This was the DFMAX algorithm [11], used as a standard benchmark for this problem in the 1993 DIMACS challenge. 450,000 random graphs were generated, with varying mean degree, each with 20 vertices. Both the sum-product algorithm and DFMAX were run on each graph, the sizes of the maximum

independent sets compared, expressed as a fraction and plotted. This made it possible to see how close the algorithm had got to finding the maximum independent set.

One problem found, was the fact that sometimes the algorithm was converging to a solution that wasn't actually an independent set. To get round this a "pruning" routine was added. This routine first removes all offending vertices, removing the ones with the highest degree first. Because this is not actually a result of the algorithm itself, these pruned points have been plotted in a different style.

The bands visible in figures 14(a), 15(a) and 16(a) are simply due to the finite size of the graph and the discrete nature of the problem. Histograms have also been generated over the whole set of data points so as to give a better picture of the quality of the results.

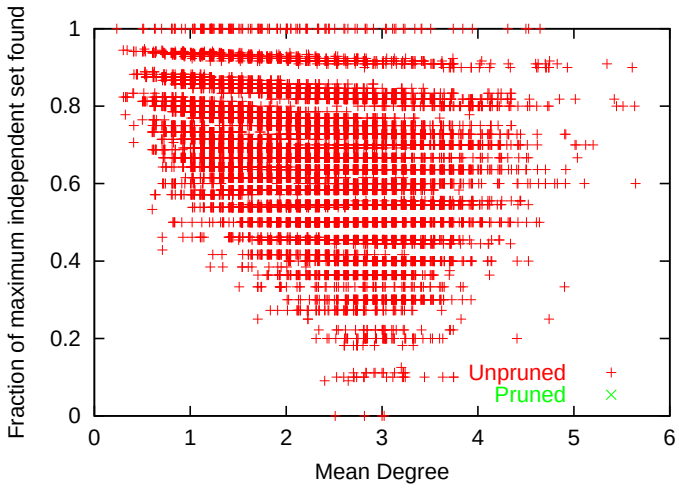
The results from these experiments confirm the early prediction that higher values of β lead to more accurate results and therefore independent sets closer to the maximum. However increasing β also brings the problem of finding solutions which are not independent sets, leading to the pruning exhibited in figure 16(c). Also, even for relatively high values of β there is still significant spread in the data, with a small but important fraction finding a set less than 80% the size of the maximum possible.

3.5 Random perturbations

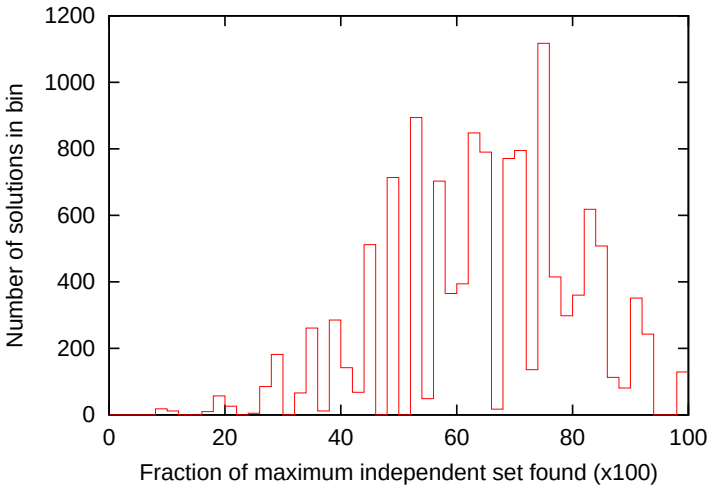
One of the biggest problems with the algorithm as described so far is its inability to resolve the symmetry in a problem, for instance, being unable to choose between vertices 4 and 10 in figure 3. To attempt to resolve this issue, two slightly different methods were introduced. The first was to perturb the weights of each vertex by a small, random amount, e.g a uniform value in the range $[-0.05, 0.05]$. The value was chosen so that it was large enough to produce an effect, but small enough that inappropriate vertices were not chosen, due to a large perturbation being applied.

The second method was much the same as above, but applied in a systematic manner. The method runs through each pair of edges that are connected by an edge and slightly increases the weight of one of them and deceases the other. This has the effect of giving one vertex in each pair a slight preference to be in the set.

These methods were then tested in all of the ways described in the previous sections and found to have almost exactly the same performance in most areas as the non perturbed algorithm. The only difference is an improvement in the quality of the results at the very low mean degree area of the graph. See figure 17(a).

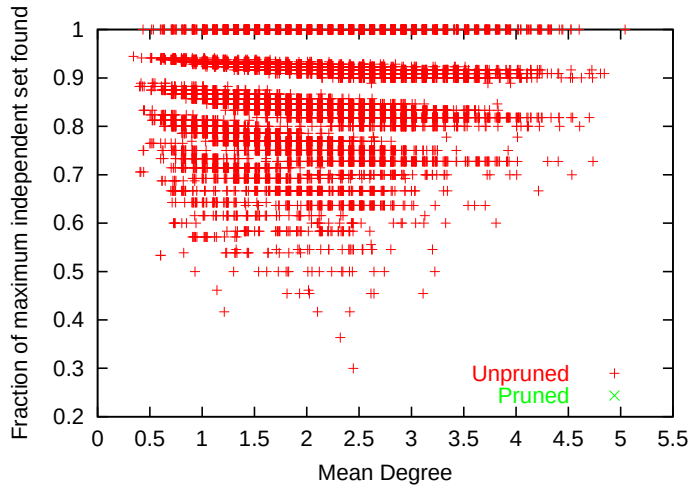


(a) Fraction of maximum set found. Each point has been randomly perturbed by a small amount in the x-axis to give some indication of the build up of points on top of each other.

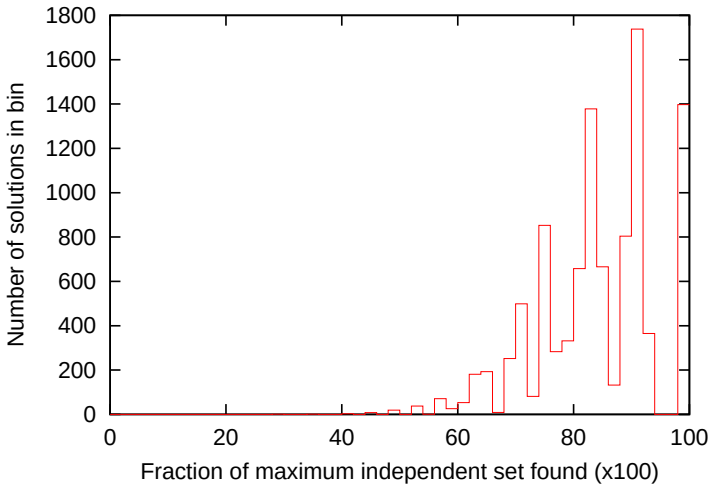


(b) Histogram over entire graph.

Figure 14: The quality of the solutions found with $\beta = 1$.

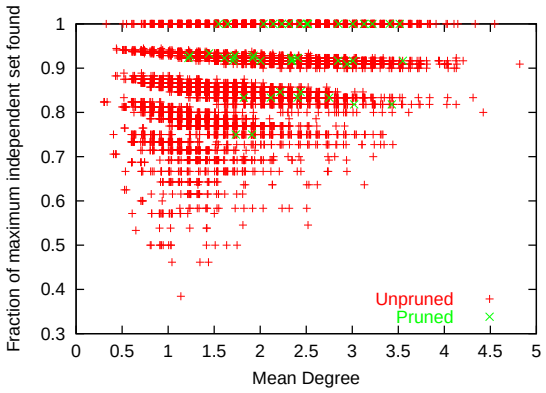


(a) Fraction of maximum set found.

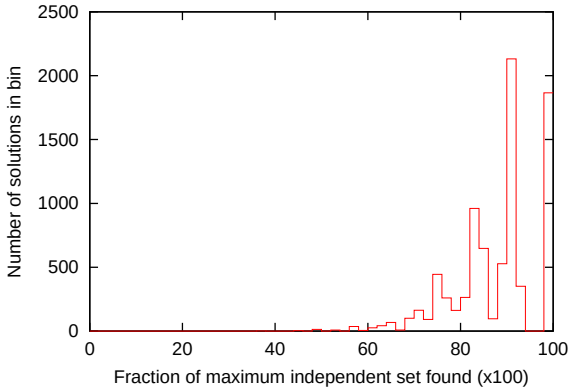


(b) Histogram over entire graph.

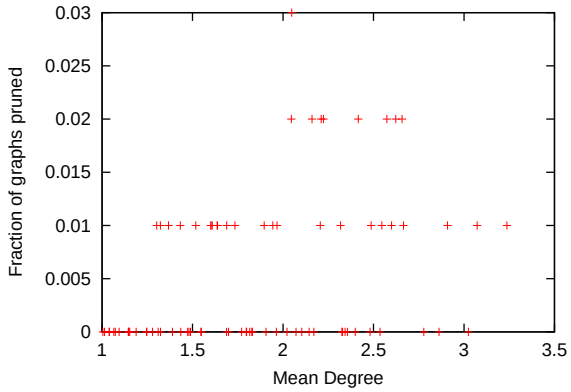
Figure 15: The quality of the solutions found with $\beta = 2$.



(a) Fraction of maximum set found.

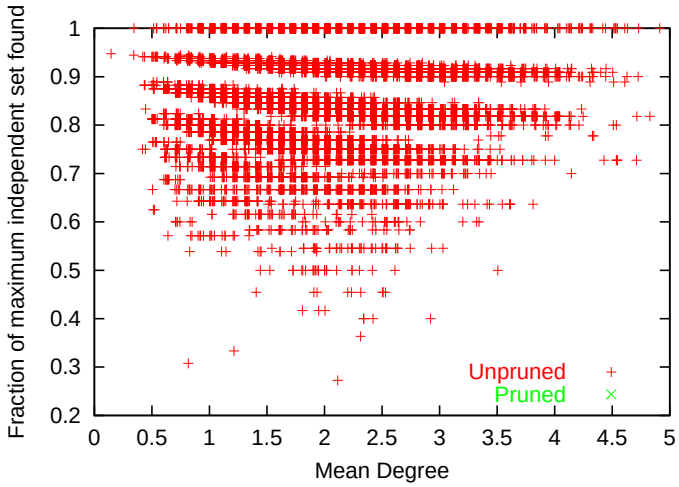


(b) Histogram over entire graph.

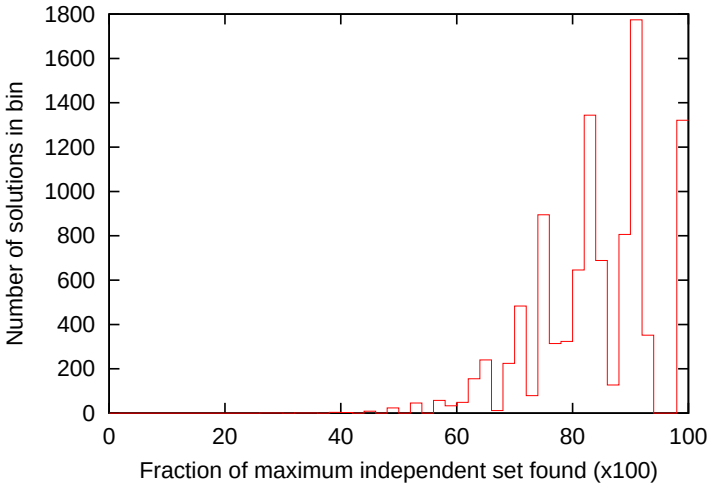


(c) Fraction of solutions that were pruned.

Figure 16: The quality of the solutions found with $\beta = 5$.



(a) Fraction of maximum set found.



(b) Histogram over entire graph.

Figure 17: The quality of the solutions found with $\beta = 2$, and using the basic perturbation method.

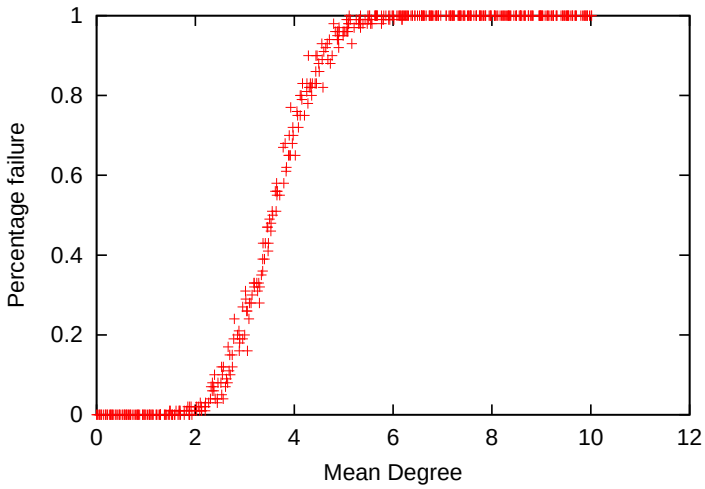


Figure 18: Conditions under which the algorithm alternates. Each point represents the fraction of 100 runs of the algorithm which did not converge. These results were generated by using the adaptive algorithm.

3.6 Varying β adaptively

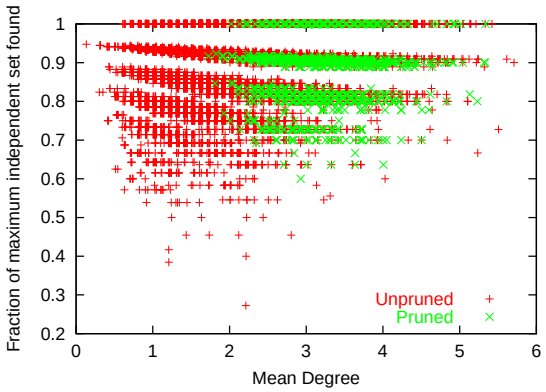
It was noted earlier that a low value of β gives a wider range of graphs which converge to a solution, while a higher value gives better results when the algorithm does converge. This suggests varying the value of β during the course of a single run.

In this case the method was introduced in a simple as possible form. Every iteration β is increased by 0.1. If the state starts to alternate then β is decreased by a variable $\Delta\beta$, which is then increased. If $\beta < 0$ then the algorithm stops. $\Delta\beta$ is initially set at 0.5. This rather complicated method ensures that the algorithm will always stop at some point, and does have a chance of recovering if it starts alternating. The stopping condition for convergence is the same as before.

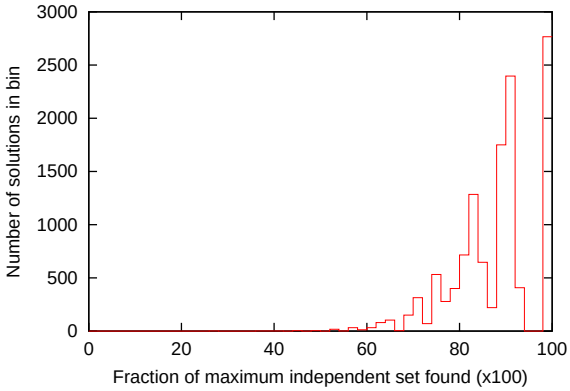
This algorithm does appear to have a number of advantages over previous methods. In figure 18 it is shown that the method has a higher transition point than a typical high β attempt, but it has the same quality of solution as a method run with this high β. The main disadvantage is the relatively high percentage of points pruned. The other disadvantage is the time penalty incurred in using this adaptive algorithm. Typical times are 5-10 times longer than the equivalent non-adaptive version.

3.7 Genetic algorithm

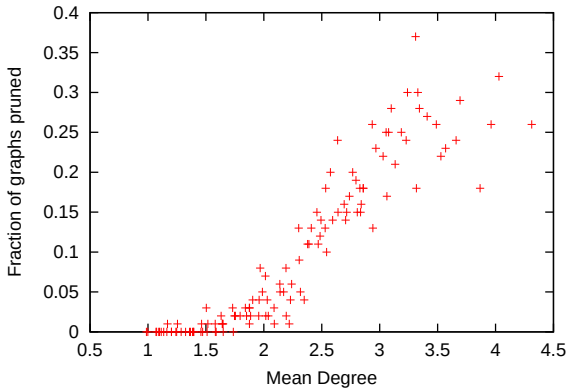
For comparative purposes a basic genetic algorithm was introduced. This was implemented using the genetic algorithm library GA-Lib [12]. The algorithm used was carried out over 400 generations, each with a population



(a) Fraction of maximum set found.



(b) Histogram over entire graph.



(c) Fraction of solutions that were pruned.

Figure 19: The quality of the solutions found using the adaptive algorithm.

of 50 genes, on graphs with 20 vertices. It had a mutation rate of 0.001 and a cross-over probability of 0.6. The fitness function used was $f(\bar{x}) = 0$ if the gene is not a independent set, and $f(\bar{x}) = k$, where k is the size of the set, if it is an independent set. This is the same fitness function as used in [3]

4 Further Discussions

4.1 Implications

The most important question to ask is: *Does the algorithm work?* The answer is, yes, but in a fairly restrictive range of graphs. The algorithm does however have a number of strong points.

4.1.1 Strengths

The algorithm is very fast. The sum-product algorithm typically takes just 10s of milliseconds to solve a 20 vertex graph, while the genetic algorithm takes almost a full second.

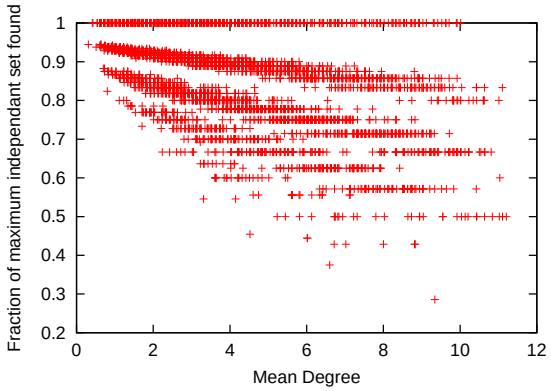
Accurate at low mean degree. For a mean degree of below about 4, the adaptive algorithm, with pruning, typically finds greater than 80% of the maximum independent set. The quality of the solutions can be improved by adding a small perturbation to the weights, which breaks the symmetry, but finding how much perturbation to use is not obvious, and depends on the particular graph that the algorithm is running on.

Computational simplicity. The algorithm only takes $O(n^2)$ computations to initialise and again, each iteration takes $O(n^2)$ computations to complete. Thus, a typical run consisting of 15 iterations, on a 20 vertex graph would only take about 200,000 multiplications, far less than the equivalent for other algorithms.

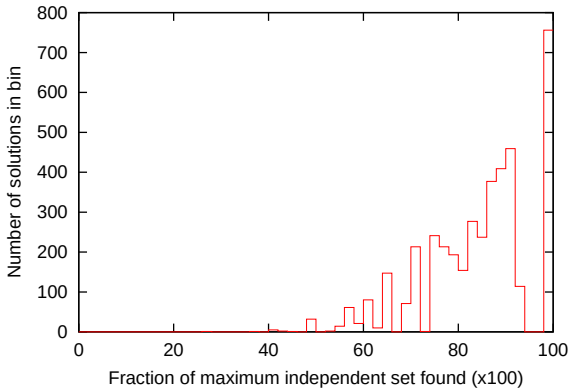
4.1.2 Weaknesses

Alternating solutions. Due to the alternating solution problem the algorithm can really only be applied to graphs with a mean degree of less than 3-4. For small graphs e.g. ones with 20 vertices, a mean degree of 3 means a density of about 15%. As the graph size increases the density for a given mean degree falls with n. This means that for a graph with 500 vertices the density is only about 0.6%. Therefore for most practical problems, the sum-product algorithm can only be applied to an extremely small subset of the possible graphs[6]. It is most

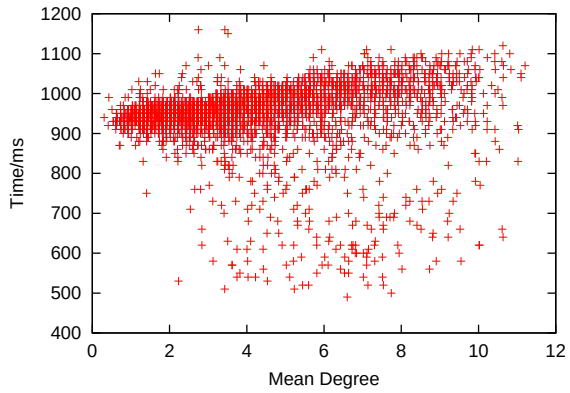
[6]In the 1993 DIMACS challenge, typical graphs had 100-4000 vertices and densities of about 0.5. This leads to mean degrees in the 100s



(a) Fraction of maximum set found.



(b) Histogram over entire graph.



(c) Time to find solution.

Figure 20: The quality and speed of the solutions found using a genetic algorithm.

likely that the alternating state is caused by having a large degree of local symmetry in the form of small cycles, as well as having a large number of inputs to each vertex.

Symmetry breaking. While for some uses of the sum-product algorithm, the fact that the symmetry of the problem is reflected in the answer is a strong point, this is not the case when it comes to finding the maximum independent set. Here the symmetry causes the algorithm to become "uncertain" and can lead to suboptimal solutions as in figure 8.

Finding non independent sets. One of the big problems with the adaptive version of the algorithm is its tendency, at higher values of the mean degree, to converge to solutions which are not independent sets. These results then need to be pruned to obtain valid results. It is not known why the algorithm converges to these solutions, but it does seem to be a result of using a higher value for β, as some of the points for $\beta = 5$ are pruned as well (see figure 16(c)).

4.2 Comparison with other algorithms

It is hard to compare the sum-product algorithm with other methods for solving the maximum independent set problem for precisely the reasons explained above. In most other work on the problem, the various algorithms tend to be tested on large (> 100 vertices), dense ($p > 0.5$) graphs, precisely the ones for which the sum-product algorithm does not work well for. Therefore the only comparisons possible are with the simple genetic algorithm implemented as described above.

In this case the sum-product algorithm compares quite well to the genetic algorithm. It is far faster, and, for the graphs it finds the solutions of, can be considered more accurate. However the genetic algorithm does not have any limitations on the mean degree. The quality falls of at higher values simply because it is harder for a random process to produce an independent set. This problem can be partially solved by using a better, graded fitness function like equation 2, or by using a different heuristic to "repair" each gene between generations [3]. Here repair means using a greedy heuristic to form a maximal independent set from the gene. This way the genetic algorithm is only used to choose between different maximal sets.

4.3 Future work

There are a number of ways in which the algorithm could be improved, or used in slightly different ways.

Damping of alternations. One way to try and stop the algorithm from alternating so easily would be to use some kind of damping mechanism.

This would take the form of the messages being a weighted average of the new messages and the old message e.g.

$$Q_{ij}^{(x)} = f \cdot Q_{ij}^{new} + (1 - f) \cdot Q_{ij}^{old}, \tag{13}$$

where f is a number in the range 0 to 1. This would hopefully remove some of the alternating effect, or at least delay its onset.

Change update schedule. In Hopfield network theory, the dynamics of the system are only guaranteed to minimise the energy if the neurons are updated asynchronously. If they are all updated at the same time then alternating states can form. Following this example, updating the sum-product algorithm one vertex at a time may help to remove the alternations.

Improve the adaptive algorithm. At the moment the adaptive algorithm is a very simple one which does not really take into account how well the algorithm is converging. As show in figure 9 the posterior probabilities have a ceiling value of β, beyond which the results do not improve. A better algorithm would perhaps converge β towards this ceiling in a exponential manner, and would decease β in a similar manner if alternation started.

Integration with a different heuristic. Good results have been obtained with genetic algorithms by combining them with a greedy heuristic [3]. Perhaps combining the sum-product algorithm with the genetic algorithm would also give improvements in the results.

Another possible way of combining the sum-product algorithm with a different heuristic would be to use the simulated annealing method for controlling the temperature to control the value of β.

5 Conclusions

The sum-product algorithm was implemented so as to solve the maximum independent set problem. The algorithm was found to have a number of strengths, in its speed and its accuracy on low mean degree graphs. However, the it was found to fail at higher mean degree, by finding alternating solutions.

Methods to try and break the symmetry of the problem were introduced but found to only have a very small effect.

A simple adaptive algorithm for β was introduced, and found to improve the quality of the solutions. However the adaptive algorithm reduced the speed of the algorithm by about 10 times. The results were compared to those for a simple genetic algorithm, and the sum-product algorithm found to be less useful for most graphs, but better for very low density ones.

References

[1] I. M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo, *The maximum clique problem.* In D.-Z. Du and P. M. Pardalos, editors, Handbook of Combinatorial Optimization, volume 4. Kluwer Academic Publishers, Boston, MA, 1999.

[2] Thomas Bäck, Sami Khuri, *An Evolutionary Heuristic for the Maximum Independent Set Problem.*

[3] Elena Marchiori, *A simple heuristic based genetic algorithm for the maximum clique problem,* Selected Areas in Cryptography, pp 366-373, 1998.

[4] M. Pelillo. *Heuristics For Maximum Clique And Independent Set.*

[5] A. Jagota, L. Sanchis, and R. Ganesan. *Approximately solving maximum clique using neural networks and related heuristics,* 1996.

[6] A. Jagota, *Approximating the Maximum Clique with a Hopfield Network,* 1992.

[7] F. R. Kschischang and B. Frey and H.-A. Loeliger, *Factor graphs and the sum-product algorithm,* IEEE Trans. Inform. Theory, 2001, volume 47, number 2, pp 498-519.

[8] B. J. Frey, F. R. Kschischang, H. A. Loeliger, and N. Wiberg, *Factor graphs and algorithms,* in Proceedings of the 35 th Allerton Conference, 1998.

[9] D. J. C. MacKay, *Good Error-Correcting Codes Based on Very Sparse Matrices,* 1999.

[10] R. G. Gallager, *Low density parity check codes,* IRE Trans. Inform. Theory, vol IT-8, pp 21-28, 1962.

[11] The DFMAX algorithm is available from:
ftp://dimacs.rutgers.edu/pub/challenge/graph/solvers.

[12] Genetic algorithm GALib library available from:
http://lancet.mit.edu/ga/.

A Main program files

This file is the header for the CGraph class which contains most of the implementation of the algorithm.

```
// CGraph.h: interface for the CGraph class.
//
//////////////////////////////////////////////////////////////////////


#include "HelperFunctions.h"

#include "CArray.h"
#include "CMatrix.h"
#include "CProb.h"
#include <string.h>
#include "ga/GASimpleGA.h"
#include "ga/GA1DBinStrGenome.h"

#define ALTERNATING 2
#define NMAX                    1000
#define REORDER


class CGraph
{

        // Required for message passing algorithm

CMatrix<CProb, CProb&>mMessages[2];
CMatrix<int, int>mEdges;

CArray<CProb, CProb&>aPrior;
CArray<CProb, CProb&>aPosterior;
CArray<float, float>        aWeight;

CArray<float, float>aResults[3];
CArray<int, int>                aResult;

int                             bResultFirst;


int nSize;
int nEdges;

int nNewMess;
int nOldMess;

int         bInitialised;
int bFirstTime;
int                             nNewArray;

// Required for dfmax algorithm

int                             bDfmaxFirstTime;
int                             nSetlim;
int                             nBestSize;

        CArray<int, int>                aBestSet;

// Requires for genetic algorithm
```

```
int                             bGAFirstTime;
float                           fBestScore;

CArray<int, int>                aGABestSet;



public:

CGraph(void);
~CGraph(void);

// Initialiseation and saving routines

int         Initialise(float fED, int nS);
int Initialise(char* strName);
int Initialise(int nS);
int Initialise_Binary_DIMACS(char* strName);

int SaveAll(char* strName);
int Save_Binary_DIMACS(char* strName);

// Message passing routines

void         InitMessages(void);
void         InitPrior(float fB);

void         SetAllPrior(float fB);
void         AddEdge(int v1, int v2);
void         SetWeight(int v, float fWeight);

float         CalcEdgeDensity(void);
float         CalcMeanEdgesPerVertex(void);

void            AddNoise(float fN);
void            Kick(float fK);
void         CalculatePosteriorProb(void);
void         DoIteration(float fBeta);

int FoundAnswer(void);

int             GetSize(void) {return nSize;};
int             GetEdge(int v1, int v2);
int             GetDegree(int v);
float           GetPosterior(int v);
float           GetPrior(int v);
float           GetWeight(int v);

void            SetResult(void);
int             GetResult(int v);
int             IsIndSet(void);
void            Prune(void);

// Dfmax routines

void            DoDFMax(int nSL);
int             MaxInd(int top, int goal, int *array,
       int depth, int *set);

int             GetBestSet(int v);
```

```
// Genetic routines

void            DoGA(int nPop, int nGen, float fMut,
     float fCross, float (*fp)(GAGenome &g));
int             GetGABestSet(int v);
float           GetBestScore(void);
};
```

This is the main CGraph file.

```
// CGraph.cpp: implementation of the CGraph class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "CGraph.h"

CGraph::CGraph(void)
{
        srand((((unsigned int)(time(NULL)))*rand())%1000000);
bInitialised = FALSE;
bFirstTime = TRUE;
bDfmaxFirstTime = TRUE;
bGAFirstTime = TRUE;
nNewMess = 1;
nOldMess = 0;
bResultFirst = TRUE;
};

CGraph::~CGraph(void)
{
bInitialised = FALSE;
};

int CGraph::Initialise(float fED, int nS)
{
int  i, j;


ASSERT(fED >= 0.0f && fED <= 1.0f);

if(bInitialised)
  {
    ASSERT(nS = nSize);
  }

nSize = nS;
nEdges = 0;

if(!bInitialised)
  {
    ASSERT(mMessages[0].SetSize(nSize, nSize));
    ASSERT(mMessages[1].SetSize(nSize, nSize));
```

```
    ASSERT(mEdges.SetSize(nSize, nSize));

    ASSERT(aPrior.SetSize(nSize));
    ASSERT(aPosterior.SetSize(nSize));
    ASSERT(aWeight.SetSize(nSize));

  };

bInitialised = TRUE;

mEdges.SetAll(FALSE);
aWeight.SetAll(1.0f);

for(i = 0; i < (nSize-1); i++)
{
for(j = i+1; j < nSize; j++)
{
if(drnd() < fED)
{
AddEdge(i+1, j+1);
}
};
};

CalcEdgeDensity();
CalcMeanEdgesPerVertex();

return TRUE;
};

int CGraph::Initialise(char* strName)
{
FILE* fStream;
int i,j, nConnections, nVert, nNull;
int nTemp;
float   fTemp;



ASSERT(!bInitialised);

nEdges = 0;

fStream = fopen(strName, "r");
if(!fStream) return FALSE;



fseek(fStream, 0L, SEEK_SET);

fscanf(fStream, "Graph description file\n");
if(fscanf(fStream, "Number of vertices: %i\n", &nSize) != 1) return FALSE;
if(fscanf(fStream, "Number of Edges: %i\n", &nEdges) != 1) return FALSE;


ASSERT(mMessages[0].SetSize(nSize, nSize));
ASSERT(mMessages[1].SetSize(nSize, nSize));
ASSERT(mEdges.SetSize(nSize, nSize));

ASSERT(aPrior.SetSize(nSize));
ASSERT(aPosterior.SetSize(nSize));
ASSERT(aWeight.SetSize(nSize));
```

```
fscanf(fStream, "Weights: ");
for(i = 0; i < nSize; i++)
{
if(fscanf(fStream, "%f", &fTemp) != 1) return FALSE;;
aWeight.Element(i) = fTemp;
};

bInitialised = TRUE;


for(i = 1; i <= nSize; i++)
{
if(fscanf(fStream, "\nVertex %i: Number of edges: %i. Connected to: ",
&nNull, &nConnections) != 2) return FALSE;
for(j = 0; j < nConnections; j++)
{
if(fscanf(fStream, "%i", &nVert) != 1) return FALSE;
AddEdge(i, nVert);
};
};


fclose(fStream);
return TRUE;
};

int CGraph::Initialise(int nS)
{
ASSERT(!bInitialised);

CProb pTemp;
pTemp.p = 0.5f;
pTemp.q = 0.5f;
nEdges = 0;
nSize = nS;

ASSERT(mMessages[0].SetSize(nSize,nSize));
ASSERT(mMessages[1].SetSize(nSize,nSize));
ASSERT(mEdges.SetSize(nSize, nSize));

ASSERT(aPosterior.SetSize(nSize));
ASSERT(aPrior.SetSize(nSize));
ASSERT(aWeight.SetSize(nSize));

aPosterior.SetAll(pTemp);
mEdges.SetAll(FALSE);
aWeight.SetAll(1.0f);

bInitialised = TRUE;

return TRUE;
};


void CGraph::SetAllPrior(float fB)
{
ASSERT(bInitialised);

int i;
```

```
CProb pInit;
float fNorm;

for(i = 0; i < nSize; i++)
{
fNorm = 1.0f + float(exp(fB*aWeight.Element(i)));

pInit.p = float(exp(fB*aWeight.Element(i))/fNorm);
pInit.q = 1.0f/fNorm;

aPrior.Element(i) = pInit;
};
};

void CGraph::AddEdge(int v1, int v2)
{
ASSERT(bInitialised);

v1--;
v2--;

if(mEdges.Element(v1, v2) == TRUE) return;
if(mEdges.Element(v2, v1) == TRUE) return;

mEdges.Element(v1,v2) = TRUE;
mEdges.Element(v2,v1) = TRUE;

nEdges++;
};

int CGraph::GetEdge(int v1, int v2)
{
  ASSERT(bInitialised);

  v1--;
  v2--;

  if(mEdges.Element(v1, v2) == TRUE) return TRUE;

  return FALSE;
};

float CGraph::CalcEdgeDensity(void)
{
float fEdgeDensity;

ASSERT(bInitialised);

int nMaxEdges;

nMaxEdges = (nSize*(nSize-1))/2;

fEdgeDensity = float(nEdges)/float(nMaxEdges);

return fEdgeDensity;
};

float CGraph::CalcMeanEdgesPerVertex(void)
{
float fMeanEdgesPerVertex;
```

```
ASSERT(bInitialised);

fMeanEdgesPerVertex = float(2*nEdges)/float(nSize);

return fMeanEdgesPerVertex;
};

void CGraph::InitPrior(float fB)
{

SetAllPrior(fB);
InitMessages();

};


void CGraph::InitMessages(void)
{
ASSERT(bInitialised);

int i, j;

for(i = 0; i < nSize; i++)
{
for(j = 0; j < nSize; j++)
{
if(mEdges.Element(i,j))
{
mMessages[0].Element(i,j) = aPrior.Element(i);
};
};
};
};

int CGraph::SaveAll(char* strName)
{
ASSERT(bInitialised);

FILE* fStream;
int i,j, nConnections;

fStream = fopen(strName, "w");
if(!fStream) return FALSE;

fprintf(fStream, "Graph description file\n");
fprintf(fStream, "Number of vertices: %i\n", nSize);
fprintf(fStream, "Number of Edges: %i\n", nEdges);

fprintf(fStream, "Weights: ");
for(i = 0; i < nSize; i++)
{
fprintf(fStream, "%f ", aWeight.Element(i));
};

for(i = 0; i < nSize; i++)
{
nConnections = 0;
for(j = 0; j < nSize; j++)
{
if(mEdges.Element(i,j)) nConnections++;
};
```

```
fprintf(fStream, "\nVertex %i: Number of edges: %i. Connected to: ",
i+1, nConnections);

for(j = 0; j < nSize; j++)
{
if(mEdges.Element(i,j)) fprintf(fStream, "%i ", j+1);
};
};

fclose(fStream);
return TRUE;
};

int CGraph::Save_Binary_DIMACS(char* strName)
{
int i, j, nLen = 0;
FILE* fStream;
char strInfo[1024];
unsigned char data[512][64];
int byte, bit, mask;
unsigned char  masks[8] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};

fStream = fopen(strName, "w");
if(!fStream) return FALSE;

nLen += sprintf(strInfo + nLen, "c Graph description file\n");
nLen += sprintf(strInfo + nLen, "c Number of vertices: %i\n", nSize);
nLen += sprintf(strInfo + nLen, "c Number of Edges: %i\n", nEdges);

nLen += sprintf(strInfo+ nLen, "c Weights: ");
for(i = 0; i < nSize; i++)
{
nLen += sprintf(strInfo + nLen, "%f ", aWeight.Element(i));
};

nLen += sprintf(strInfo + nLen, "\np DIMACS_DATA %i %i\n", nSize, nEdges);

fprintf(fStream, "%i\n", strlen(strInfo));
fprintf(fStream, "%s", strInfo);

for(i = 0; i < nSize; i++)
{
for(j = 0; j < i; j++)
{
bit = 7 - (j & 0x00000007);
byte = j >> 3;

mask = masks[bit];

if(mEdges.Element(i, j))
{
data[i][byte] |= mask;
}
else
{
data[i][byte] &= ~mask;
};
};
};

for(i = 0; i < nSize; i++)
{
```

```
fwrite(data[i],1, int((i + 8)/8), fStream);
};

fclose(fStream);
return TRUE;
};

int CGraph::Initialise_Binary_DIMACS(char* strName)
{
int i, j, nLen = 0;
FILE* fStream;
char strInfo[1024];
unsigned char data[512][64];
int byte, bit, mask;
unsigned char  masks[8] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};
char* cPos;
char strTemp[100];
char c;
int stop = FALSE;

fStream = fopen(strName, "r");
if(!fStream) return FALSE;

if(!fscanf(fStream, "%i\n", &nLen))
return FALSE;

fread(strInfo, 1, nLen, fStream);

nSize = 0;
cPos = strInfo;

while(!stop && (c = *cPos++) != '\0')
{
switch(c)
{
case 'c':
while((c = *cPos++) != '\n' && c != '\0');
break;

case 'p':
sscanf(cPos, "%s %i %i\n", strTemp, &nSize, &nEdges);
stop = TRUE;
break;

default:
break;
}
};

if(nSize == 0) return FALSE;
nEdges = 0;

ASSERT(mMessages[0].SetSize(nSize, nSize));
ASSERT(mMessages[1].SetSize(nSize, nSize));
ASSERT(mEdges.SetSize(nSize, nSize));

ASSERT(aPrior.SetSize(nSize));
ASSERT(aPosterior.SetSize(nSize));
ASSERT(aWeight.SetSize(nSize));

aWeight.SetAll(1.0f);
```

```
for(i = 0; i < nSize; i++)
{
fread(data[i], 1, int((i + 8)/8), fStream);
};

bInitialised = TRUE;

for(i = 0; i < nSize; i++)
{
for(j = 0; j < i; j++)
{
bit = 7 - (j & 0x00000007);
byte = j >> 3;

mask = masks[bit];

if((data[i][byte] & mask) == mask) AddEdge(i+1, j+1);
};
};

fclose(fStream);
return TRUE;
};


void CGraph::CalculatePosteriorProb(void)
{
int  i, j;


ASSERT(bInitialised);

for(i = 0; i < nSize; i++)
{

  aPosterior.Element(i) = aPrior.Element(i);
  for(j = 0; j < nSize; j++)
    {
      if(mEdges.Element(j, i) == TRUE)
{
  aPosterior.Element(i).p *=
    mMessages[nOldMess].Element(j, i).q;
};
    };
  aPosterior.Element(i).Normalise();
};

return;

};

int CGraph::FoundAnswer(void)
{

int       nAltCheckArray;
int    i, j;
int    bFoundAnswer;

if(bFirstTime)
{
bFirstTime = FALSE;
for(i = 0; i < 3; i++)
```

```
{
  nNewArray = 0;
  aResults[i].SetSize(nSize);
  aResults[i].SetAll(-1.0);
};
};

for(i = 0; i < nSize; i++)
{
aResults[nNewArray].Element(i) =
aPosterior.Element(i).p;
};

nAltCheckArray = nNewArray;
nNewArray++;
if(nNewArray >= 3) nNewArray = 0;

bFoundAnswer = TRUE;

for(i = 0; i < nSize; i++)
{
for(j = 1; j < 3; j++)
{
if(fabs(aResults[0].Element(i) -
aResults[j].Element(i)) > 0.01)
bFoundAnswer = FALSE;
};
};

if(!bFoundAnswer)
{
bFoundAnswer = ALTERNATING;
for(i = 0; i < nSize; i++)
{
if(fabs(aResults[nNewArray].Element(i) -
aResults[nAltCheckArray].Element(i)) > 0.001)
bFoundAnswer = FALSE;
};
};

return bFoundAnswer;
};

void CGraph::AddNoise(float fN)
{

  int i;
  for(i = 0; i < nSize; i++)
    {
      aWeight.Element(i) = aWeight.Element(i) + (2.0f*((float)drnd()-0.5f)*fN);
    }

  return;
}

void CGraph::Kick(float fK)
{

  int i, j;
  float fKick;

  for(i = 0; i < nSize-1; i++)
```

```
    {
      for(j = i+1; j < nSize; j++)
{
  if(mEdges.Element(i, j) == TRUE)
    {
      fKick = 2.0f*((float)drnd()-0.5f)*fK;

      aWeight.Element(i) = aWeight.Element(i) + fKick;
      aWeight.Element(j) = aWeight.Element(j) - fKick;

    };
};
    }

  return;
}

void CGraph::DoIteration(float fBeta)
{
int i, j, i2;
float                   fMaxNoise, fN;


ASSERT(bInitialised);

SetAllPrior(fBeta);

for(i = 0; i < nSize; i++)
{
  for(j = 0; j < nSize; j++)
    {
      if(mEdges.Element(i, j) == TRUE)
{
  mMessages[nNewMess].Element(i, j) =
    aPrior.Element(i);
  for(i2 = 0; i2 < nSize; i2++)
    {
      if(i2 == j) continue;
      if(mEdges.Element(i2, i) == TRUE)
{
  mMessages[nNewMess].Element(i, j).p *=
    mMessages[nOldMess].Element(i2, i).q;
};
    };
  mMessages[nNewMess].Element(i, j).Normalise();

};
    };
};

nNewMess = nOldMess;
nOldMess = 1 - nNewMess;
};


void CGraph::SetWeight(int v, float fWeight)
{
aWeight.Element(--v) = fWeight;
};


float CGraph::GetPosterior(int v)
```

```
{
  ASSERT(bInitialised);

  v--;

  return aPosterior.Element(v).p;
};

float CGraph::GetPrior(int v)
{
  ASSERT(bInitialised);

  v--;

  return aPrior.Element(v).p;
};

float CGraph::GetWeight(int v)
{
  ASSERT(bInitialised);

  v--;

  return aWeight.Element(v);
};

int CGraph::GetDegree(int v)
{
  int i;
  int nCount = 0;

  for(i = 1; i <= nSize; i++)
    {
      if(GetEdge(v, i) == TRUE) nCount++;
    };

  return nCount;
};


void CGraph::SetResult(void)
{
  int i;

  if(bResultFirst)
    {
      bResultFirst = FALSE;
      aResult.SetSize(nSize);
    };

  aResult.SetAll(0);

  for(i = 0; i < nSize; i++)
    {
      if(aPosterior.Element(i).p > 0.5) aResult.Element(i) = 1;
    };
};

int CGraph::GetResult(int v)
{
  v--;
```

```
  return aResult.Element(v);
};

int CGraph::IsIndSet(void)
{
  int i, j;


  for(i = 0; i < nSize-1; i++)
    for(j = i+1; j < nSize; j++)
      {
if(aResult.Element(i) == 1 && aResult.Element(j) == 1
   && GetEdge(i+1, j+1) == TRUE)
  {
    return FALSE;
  };
      };
  return TRUE;
};

void CGraph::Prune(void)
{
  int i, j;

  while(IsIndSet() == FALSE)
    {
      for(i = 0; i < nSize-1; i++)
for(j = i+1; j < nSize; j++)
  {
    if(aResult.Element(i) == 1 && aResult.Element(j) == 1
       && GetEdge(i+1, j+1) == TRUE)
      {
if(GetDegree(i+1) >= GetDegree(j+1)) aResult.Element(i) = 0;
else aResult.Element(j) = 0;
      };
  }
    }
};


// The dfmax routine - modified to be run as a subroutine



void CGraph::DoDFMax(int nSL)
{
        int i, j;
int cand,newcand;
int dmax, x, y;
int vertex[NMAX];
int set[NMAX];
int degree[NMAX];
int bestset[NMAX];

if(bDfmaxFirstTime == TRUE)
  {
    bDfmaxFirstTime = FALSE;
    aBestSet.SetSize(nSize);
  };

aBestSet.SetAll(0);
```

```
        nSetlim = nSL;

#ifdef REORDER

for (i=1;i<=nSize;i++)
  {
degree[i] = 0;
for (j=1;j<=nSize;j++)
if (!GetEdge(i,j)) degree[i]++;
  }

dmax = -1;
for(i=1;i<=nSize;i++)
  {
if (degree[i] > dmax)
  {
dmax = degree[i];
cand = i;
  }
  };
vertex[nSize] = cand;
for(j=nSize-1;j>=1;j--)
  {
degree[cand] = -9;
dmax = -1;
for (i=1;i<=nSize;i++)
  {
if (!GetEdge(cand,i)) degree[i]--;
if (degree[i] > dmax)
  {
dmax = degree[i];
newcand = i;
  }
  }
vertex[j] = cand = newcand;
  };
#else
for (j=1;j<=nSize;j++) vertex[j] = j;
#endif

nBestSize = 0;
nBestSize = MaxInd(nSize, nSetlim, vertex, 1, set);

for(i = 0; i < nSize; i++)
  {
    bestset[i-1] = FALSE;
  };
for(i = 1; i <= nBestSize; i++)
  {
    bestset[aBestSet.Element(i)-1] = TRUE;
  };
for(i = 0; i < nSize; i++)
  {
    aBestSet.Element(i) = bestset[i];
  };
};


int CGraph::MaxInd(int top, int goal, int *array, int depth, int *set)
{
        int newarray[NMAX];
int i,v,u,w,z;
```

```
int best, restbest, newgoal;
unsigned *bitloc;
int *pnew, *pold;
int canthrow;

if(top <= 1) {
if(top == 0) depth--;
if(depth > nBestSize) {
nBestSize = depth;
if(top == 1) set[nBestSize] = array[top];
for(i=1;i<=nBestSize;i++)
  aBestSet.Element(i) = set[i];
}
return(top);
}
best = 1;
newgoal = goal-1;
if (newgoal <= 1) newgoal = 1;
for (i = top; i >= goal; i--) {
pnew = newarray;
w = array[i];
set[depth] = w;
canthrow = i - goal;
pold = array+1;
while (pold<array+i) {
z = *pold++;
if (!GetEdge(z,w)) {
 *++pnew = z;
} else {
    if (canthrow == 0) goto breakout;
    canthrow--;
}
}
restbest = MaxInd(pnew-newarray,newgoal,newarray,depth+1,set);
if (restbest >= newgoal) {
best = newgoal = restbest+1;
goal = best+1;
}
if (top == nSize) {

}
    breakout:;
}
return best;
};

int CGraph::GetBestSet(int v)
{
  ASSERT(bInitialised);

  v--;

  return aBestSet.Element(v);
};

//GA routines


void CGraph::DoGA(int nPop, int nGen, float fMut,
  float fCross, float (*fp)(GAGenome &))
{
  /*
```

```
    Genetic algorithm used is GALib,

    Copyright 1995-1996 Massachusetts Institute of Technology (MIT)
                        all rights reserved
    Copyright 1996-1999 Matthew Wall (the Author)
                        all rights reserved
  */

  int i;



  if(bGAFirstTime == TRUE)
    {
      aGABestSet.SetSize(nSize);
      bGAFirstTime = FALSE;
    }


  aGABestSet.SetAll(0);



  GA1DBinaryStringGenome genome(nSize, fp);


  GASimpleGA ga(genome);
  ga.populationSize(nPop);
  ga.nGenerations(nGen);
  ga.pMutation(fMut);
  ga.pCrossover(fCross);
  ga.evolve();


  GA1DBinaryStringGenome & g =
    (GA1DBinaryStringGenome &)ga.statistics().bestIndividual();

  fBestScore = g.score();

  for(i = 0; i < nSize; i++)
    aGABestSet.Element(i) = g.gene(i);

  return;
};

int CGraph::GetGABestSet(int v)
{
  ASSERT(bInitialised);

  v--;

  return aGABestSet.Element(v);
};

float CGraph::GetBestScore(void)
{
  return fBestScore;
};
```

B Example program

This is an example of how the CGraph class can be used to solve the maximum independent set problem. This program implements the adaptive algorithm for β, and shows how to use the DFMAX and genetic methods. The graphs should be saved in the binary DIMACS format.

```
// sol.cpp : Defines the entry point for the console application.
//

/////////////////////////////////////////////////////////
// Includes
/////////////////////////////////////////////////////////

#include "../graph/stdafx.h"

#define STEP  0.5f
#define TOL   0.01f

float Objective(GAGenome& gen);

CGraph* g = NULL;

/////////////////////////////////////////////////////////
// Main Function
/////////////////////////////////////////////////////////

int main(int argc, char* argv[])
{

CGraph graph;
char   strName[40];
float  fBeta = 0.0;
float  fBackStep = 0.5f;
int    nIter, i;
int    nSize;
int    nStop = FALSE;

int    nFound;

int    bDF = FALSE;
int    bAddNoise = FALSE;
int    bKick = FALSE;
float  fN = 0.0;
float  fK = 0.0f;


if(argc < 2)
  {
    printf("Usage: sol name [dfmax] [a noise_val]");
    printf(" [k kick]\n");
    return 1;
  }

strcpy(strName, argv[1]);

if(argc > 2)
  {
    for(i = 2; i < argc; i++)
      {
```

```
switch(argv[i][0])
  {
  case 'd':
    bDF = TRUE;
    break;

  case 'a':
    bAddNoise = TRUE;
    fN = atof(argv[i+1]);
    i++;
    break;

  case 'k':
    bKick = TRUE;
    fK = atof(argv[i+1]);
    i++;
    break;


  default:
    printf("Usage: sol name [dfmax] [a noise_val]");
    printf(" [k kick]\n");
    return 1;
  }
      }
  }

printf("Solving %s\n", strName);

if(graph.Initialise_Binary_DIMACS(strName))
  {
    printf("Loaded\n");

  }
else
  {
    printf("Failed to load\n");
    return 1;
  };

nSize = graph.GetSize();

        if(bAddNoise == TRUE) graph.AddNoise(fN);
if(bKick == TRUE) graph.Kick(fK);


graph.InitPrior(fBeta);
for(nIter = 0; nIter < 10000; nIter++)
  {
    printf("%i %f\n", nIter, fBeta);

    graph.CalculatePosteriorProb();
    if((nFound = graph.FoundAnswer()) && nIter > 3)
      {

if(nFound == 1) nStop = TRUE;
else if(nFound == 2)
  {
    fBeta -= fBackStep;
    fBackStep += 0.5f;
    if(fBeta < 0.0f) nStop = TRUE;
  }
```

```
      };

    if(nStop == TRUE) break;

    fBeta += 0.1f;

    graph.DoIteration(fBeta);
  };

if(nFound == 2)
  {
    printf("Found alternating solution\n");


    printf("Doing DFMax\n");

    if(bDF == TRUE)
      {
graph.DoDFMax(1);
      };

    g = &graph;

    printf("Doing GA\n");

    graph.DoGA(50, 400, 0.001f, 0.6f, Objective);

    printf("Vertex\t\tGenetic\t\tDfmax\n");

    for(i = 1; i <= nSize; i++)
      {
printf("\n%i\t\t%i", i, graph.GetGABestSet(i));
if(bDF == TRUE) printf("\t\t%i", graph.GetBestSet(i));
      };

    printf("\n");

    return 0;
  }


if(bDF == TRUE)
  {
    printf("Doing Dfmax\n");
    graph.DoDFMax(1);
  };

printf("Found at beta: %f\n", fBeta);

printf("Doing GA\n");

g = &graph;

graph.DoGA(50, 400, 0.001f, 0.6, Objective);

printf("GA Complete\n");

graph.SetResult();

printf("Vertex\t\tResult\t\tWeight\t\tPrediction\tGenetic\t\tDfmax\n");

for(i = 1; i <= nSize; i++)
```

```
  {
    printf("\n%i\t\t%f\t%f\t%i\t\t%i", i, graph.GetPosterior(i),
   graph.GetWeight(i),
   graph.GetResult(i),
   graph.GetGABestSet(i));
    if(bDF == TRUE) printf("\t\t%i", graph.GetBestSet(i));
  };

if(graph.IsIndSet() == FALSE)
  {
    printf("\nNot an independant set");
    graph.Prune();
    printf("Pruned: ");
    for(i = 1; i <= nSize; i++)
      printf("%i ", graph.GetResult(i));
  };

printf("\n");

return 0;
}

float Objective(GAGenome& gen)
{
  int i, j;

  GA1DBinaryStringGenome & genome = (GA1DBinaryStringGenome &)gen;

  float score=0.0f;
  for(i=0; i<g->GetSize(); i++)
    {
      if(genome.gene(i) == 1) score += 1.0f;
    }

  for(i = 0; i<(g->GetSize()-1); i++)
    for(j = i+1; j<g->GetSize(); j++)
      {
if(g->GetEdge(i+1, j+1) == TRUE && genome.gene(i) == 1
   && genome.gene(j) == 1)
  {
    return 0.0f;
  };
      };

  return score;
}
```