Computing Tutte Polynomials

Gary Haggard[1], David J. Pearce[2], and Gordon Royle[3]

[1] Bucknell University
haggard@bucknell.edu
[2] Computer Science Group, Victoria University of Wellington,
david.pearce@mcs.vuw.ac.nz
[3] School of Mathematics and Statistics, University of Western Australia
gordon@maths.uwa.edu.au

Abstract. The Tutte polynomial of a graph, also known as the partition function of the q-state Potts model, is a 2-variable polynomial graph invariant of considerable importance in both combinatorics and statistical physics. It contains several other polynomial invariants, such as the chromatic polynomial and flow polynomial as partial evaluations, and various numerical invariants such as the number of spanning trees as complete evaluations. However despite its ubiquity, there are no widely-available effective computational tools able to compute the Tutte polynomial of a general graph of reasonable size. In this paper we describe the implementation of a program that exploits isomorphisms in the computation tree to extend the range of graphs for which it is feasible to compute their Tutte polynomials. We also consider edge-selection heuristics which give good performance in practice. We empirically demonstrate the utility of our program on random graphs. More evidence of its usefulness arises from our success in finding counterexamples to a conjecture of Welsh on the location of the real flow roots of a graph.

1 Introduction

The Tutte polynomial of a graph is a 2-variable polynomial of significant importance in mathematics, statistical physics and biology [25]. In a strong sense it "contains" *every* graphical invariant that can be computed by deletion and contraction. The Tutte polynomial can be evaluated at particular points (x, y) to give numerical graphical invariants, including the number of spanning trees, the number of forests, the number of connected spanning subgraphs, the dimension of the bicycle space and many more. The Tutte polynomial also specialises to a variety of single-variable graphical polynomials of independent combinatorial interest, including the *chromatic polynomial*, the *flow polynomial* and the *reliability polynomial*.

The Tutte polynomial plays an important role in the field of statistical physics where it appears as the partition function of the q-state Potts model $Z_G(q, v)$ (see [22, 28]). In fact, if G is a graph on n vertices then

$$T(G, x, y) = (x-1)^{-1}(y-1)^{-n} Z_G((x-1)(y-1), (y-1))$$

and so the partition function of the q-state Potts model is simply the Tutte polynomial expressed in different variables. There is a very substantial physics literature involving the calculation of the partition function for specific families of graphs, usually sequences of increasingly large subgraphs of various infinite lattices and other graphs with some sort of repetitive structure (see e.g [27, 23]).

In knot theory, the Tutte polynomial appears as the Jones polynomial of an alternating knot [5, 4]. Computing the Jones polynomial of a non-alternating knot requires a signed Tutte polynomial [16, 13], which is more involved. This has application in many areas, such as the analysis of knotted strands of DNA [4]

The Tutte polynomial also specialises to the chromatic polynomial, which emerged from work on the four-colour theorem [1] and plays a special role in combinatorics and statistical physics. In statistical physics, the chromatic polynomial occurs as a special limiting case, namely the zero-temperature limit of the anti-ferromagnetic Potts model, while in combinatorics its relationship to graph colouring and historical status as perhaps the earliest graph polynomial has given it a unique position. As a result, particularly in the combinatorics literature, far more is known about the chromatic polynomial than about the Tutte polynomial or any of its other univariate specialisations such as the flow polynomial, and there are still fundamental unresolved questions in these areas. Exploration of these questions is hampered by the lack of an effective general-purpose computational tool that is able to deal with larger problem instances than the naive implementations found in common software packages such as Maple and Mathematica. Previous algorithms for computing Tutte polynomials have either not scaled beyond small graphs [21, 20, 2]; or, have been restricted to specialised cases [8, 17, 7, 26].

Earlier work of the first author [11, 10] describes such a tool for the chromatic polynomial of a graph, where the task is considerably simpler as the chromatic polynomial is univariate and all the graphs can be taken to be simple. Dealing with a bivariate polynomial and manipulating graphs that may include loops and multiple edges introduces a range of different issues that must be resolved. In [21], an algorithm is described that will compute Tutte polynomials of graphs with no more than 14 vertices that depends on generating all spanning trees of a graph. The algorithm given in [19] for computing chromatic polynomials was extended in [20] to compute Tutte polynomials of moderate sized graphs, but is not effective much beyond 14 vertices. By comparison, our algorithm can process graphs with 14 vertices in a matter of seconds (as shown in §6). In [2] an alternate strategy that uses "roughly $2^{n+1}n$ words of memory for an n-vertex graph" is implemented. The authors comment that our algorithm is "the fastest current program to compute Tutte polynomials", although they identify certain graph classes where their approach is more efficient. Memory considerations create constraints on the practicality of the algorithm (see [3] App. D).

In this paper, we describe the implementation of an efficient algorithm for computing Tutte, Flow and Chromatic polynomials. The algorithm is based on the idea of caching intermediate graphs and their Tutte polynomials and using graph isomorphism to avoid unnecessary recomputation of branches of the

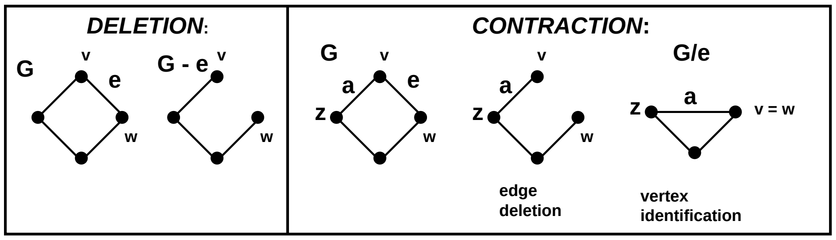

Fig. 1. Deletion and Contraction of an Edge

computation tree. We also employ several heuristics and report on experiments comparing them. Our findings indicate that, of these, two stand out from the rest. Furthermore, whilst one is the best choice on dense graphs, the other performs better on sparse graphs. We present some experimental results of this algorithm and discussion of the possible factors affecting its performance. In addition, as an example of its practical use, we present counterexamples to a conjecture of Welsh on the location of the roots of the flow polynomial of a graph, by finding for the first time graphs with real flow roots larger than 4. Finally, our implementation also supports efficient computation of chromatic and flow polynomials, based on the same techniques presented in this paper, and can be obtained from `http://www.mcs.vuw.ac.nz/~djp/tutte`.

2 Preliminaries

Let $G = (V, E)$ be an *undirected multi-graph*; that is, V is a set of vertices and E is a multi-set of unordered pairs (v, w) with $v, w \in V$. An edge (v, v) is called a *loop*. If an edge (u, v) occurs more than once in E it is called a *multi-edge*. The *underlying graph* of G is obtained by removing any duplicate entries in E.

Two operations on graphs are essential to understand the definition of the Tutte polynomial. The operations are: deleting an edge, denoted by $G - e$; and contracting an edge, denoted by G/e. See Figure 1.

Definition 1. *The* Tutte polynomial *of a graph $G = (V, E)$ is a two-variable polynomial defined as follows:*

$$T(G, x, y) = \begin{cases} 1 & E(G) = \emptyset \\ xT(G/e, x, y) & e \in E \textit{ and } e \textit{ is a bridge} \\ yT(G - e, x, y) & e \in E \textit{ and } e \textit{ is a loop} \\ T(G - e, x, y) + T(G/e, x, y) & e \textit{ is neither a loop nor} \\ & \textit{ a bridge} \end{cases}$$

The definition of a Tutte polynomial outlines a simple recursive procedure for computing it. However, we are free to apply its rules in whatever order we wish [25], and to choose any edge to operate on at each stage. Figures 2 and 3 illustrate this recursive procedure applied to a simple graph to give the

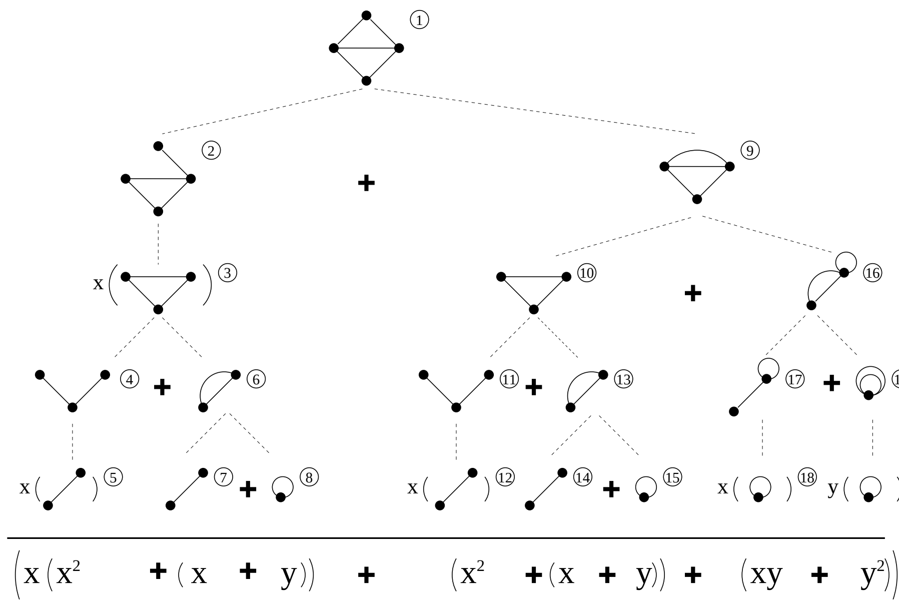
1
2
9
3
10
16
4
6
11
13
17
19
5
7
8
12
14
15
18
20
x
x
x
x
y
+
$\left(x\left(x^2 + (x + y)\right) + \left(x^2 + (x + y)\right) + \left(xy + y^2\right)\right)$

Fig. 2. Illustrating the application of Definition 1 to a small graph. Observe that vertices are not drawn once they become isolated, as they play no further role. Also, each graph is given a unique number to aid identification.

$$\begin{aligned}
&T(G_1 = \{(1,2),(2,3),(3,4),(4,1),(2,4)\}) &&= T(G_2 = G_1 - (4,1)) + T(G_9 = G_1/(4,1))\\
&T(G_2 = \{(1,2),(2,3),(3,4),(2,4)\}) &&= x \cdot T(G_3 = G_2 - (1,2))\\
&T(G_3 = \{(2,3),(3,4),(2,4)\}) &&= T(G_4 = G_3 - (2,4)) + T(G_6 = G_3/(2,4))\\
&T(G_4 = \{(1,2),(2,3),(3,4)\}) &&= x \cdot T(G_5 = G_4 - (3,4))\\
&T(G_5 = \{(1,2),(2,3),(2,4)\}) &&= x \cdot T(G_5 - (2,3) = \emptyset) = x \cdot 1\\
&T(G_6 = \{(2,3),(3,2)\}) &&= T(G_7 = G_6 - (2,3)) + T(G_8 = G_6/(2,3))\\
&T(G_7 = \{(2,3)\}) &&= x\\
&T(G_8 = \{(2,2)\}) &&= y\\
&T(G_9 = \{(4,2),(2,3),(3,4),(2,4)\}) &&= T(G_{10} = G_9 - (4,2)) + T(G_{16} = G_9/(4,2))\\
&T(G_{10} = \{(2,3),(3,4),(2,4)\}) &&= T(G_{11} = G_{10} - (4,2)) + T(G_{13} = G_9/(4,2))\\
&T(G_{11} = \{(2,3),(3,4)\}) &&= x \cdot T(G_{12} = G_{11} - (3,4))\\
&T(G_{12} = \{(2,3)\}) &&= x \cdot T(G_{12} - (2,3) = \emptyset) = x \cdot 1\\
&T(G_{13} = \{(2,3),(3,2)\}) &&= T(G_7 = G_{14} - (2,3)) + T(G_{15} = G_6/(2,3))\\
&T(G_{14} = \{(2,3)\}) &&= x\\
&T(G_{15} = \{(2,2)\}) &&= y\\
&T(G_{16} = \{(2,3),(3,2),(2,2)\}) &&= T(G_{17} = G_{16} - (2,3)) + T(G_{19} = G_9/(2,3))\\
&T(G_{17} = \{(3,2),(2,2)\}) &&= x \cdot T(G_{18} = G_{16} - (2,3))\\
&T(G_{18} = \{(2,2)\}) &&= y\\
&T(G_{19} = \{(2,2),(2,2)\}) &&= y \cdot T(G_{20} = G_{19} - (2,2))\\
&T(G_{20} = \{(2,2))\}) &&= y \cdot T(G_{19} - (2,2) = \emptyset) = y \cdot 1
\end{aligned}$$

Fig. 3. Illustrating an algebraic proof of the computation illustrated in Figure 2. Observe that the graph numbers given (e.g. G_1) align with those given in Figure 2.

final polynomial. It should be clear from these figures that the structure of the computation corresponds to a tree.

The order in which the rules of Definition 1 are applied significantly affects the size of the computation tree. An “efficient” order can reduce work in a number of ways. For example, there are two situations where an edge is associated with a factor directly: if the edge is a loop, the factor is y; likewise, if the edge is a bridge, the factor is x. Eliminating such edges as soon as possible and storing the factor for later incorporation into the answer reduces work by lowering the cost of operations (e.g. contracting, connectedness testing, etc.) on graphs in the subtrees below. In Figure 2, for example, the loop present in G_{16} is not reduced immediately and, instead, is propagated to the bottom of the computation tree; removing it immediately reduces, amongst other things, the cost of duplicating the graph when the branch forks further down.

Within a single computation tree, it often arises that a graph G occurs more than once. Thus, recomputing $T(G)$ from scratch each time is wasteful and should be avoided when possible. For example, the triangle occurs twice in Figure 2, both as G_3 and G_{10}. Thus, we can simplify the tree by simply reusing the result from $T(G_3)$ in place of $T(G_{10})$. This optimisation has a significant effect on the performance of our algorithm in practice (as shown in §6).

The choice of edge for a delete/contract operation can also greatly affect the size of the computation tree. In particular, it affects the likelihood of reaching a graph isomorphic to one already seen. For example, selecting $(4, 2)$ when evaluating $T(G_9)$ in Figure 3 yields the triangle (as shown); choosing any of the other edges, however, does not. We have not yet explored the effects of different *edge selection heuristics*, although this remains important future work.

Finally, an efficient algorithm for computing Tutte polynomials can be used immediately to compute chromatic, flow and reliability polynomials. For example, for the chromatic polynomial $P(G, \lambda)$ and flow polynomial $F(G, x)$ of a graph with n vertices, e edges and c connected components are derived as follows from the Tutte polynomial:

$$\begin{aligned} P(G, \lambda) &= (-1)^{n-c}\lambda \cdot T(G, (1-\lambda), 0) \\ F(G, x) &= (-1)^{e-n+c} \cdot T(G, 0, (1-x)) \end{aligned}$$

3 Roots of Flow Polynomials - Welsh's Conjecture

Our primary motivation for developing an efficient algorithm was to extend the range for which computational exploration of questions relating to Tutte polynomials is feasible, as there are a number of long-standing open questions for which our computational evidence is extremely limited.

Several of these problems relate to the location of the *roots* of the various single-variable specialisations of the Tutte polynomial mentioned earlier. In particular, the roots of the chromatic polynomial, or *chromatic roots* have been extensively studied while much less is known about the roots of the flow polynomial. One fundamental question about which very little is known is whether there is an upper bound on the value of real flow roots. As there are graphs (such as the Petersen graph) with no 4-flows, the strongest possible result would be that there are no real flow roots larger than 4.

Conjecture 1 (Dominic Welsh) *If G is a bridgeless graph with flow polynomial F_G, then $F_G(r) > 0$ for all $r \in (4, \infty)$.*

This conjecture is essentially a dual version of the famous Birkhoff-Lewis conjecture that planar graphs have no *chromatic* roots in $[4, \infty)$ which has been proved for $r = 4$ (the four-colour theorem) and for $[5, \infty)$.

In prior study of chromatic roots, cubic graphs of high girth have played an important role as they seem to exhibit qualitatively extremal behaviour (this is a deliberately imprecise statement) and for this reason, they are a natural class to examine for other questions related to Tutte polynomials. In this vein we computed the Tutte polynomials of cubic graphs of girth at least 7 on 24–32 vertices with the intention of testing a variety of conjectures against this data set.

An immediate positive outcome of this experiment was the discovery of a number of counterexamples to Welsh's conjecture. A specific example is the generalised Petersen graph $P(16, 6)$ which is a 32-vertex cubic graph of girth 7 shown in Figure 4 with flow polynomial $(t-1)(t-2)(t-3)Q(t)$ where

$$\begin{aligned}Q(t) = t^{14} &- 42\,t^{13} + 833\,t^{12} - 10358\,t^{11} + 90393\,t^{10} - 587074\,t^{9}\\ &+ 2934917\,t^{8} - 11515364\,t^{7} + 35798907\,t^{6} - 88275860\,t^{5}\\ &+ 171273551\,t^{4} - 256034548\,t^{3} + 282089291\,t^{2}\\ &- 207662412\,t + 77876944.\end{aligned}$$

This has real roots at two values $t_1 \approx 4.0252205$ and $t_2 \approx 4.2331455$ thereby demonstrating that 4 is not the upper limit for flow roots.

There are a variety of other examples on 28 and 36 vertices, but the smaller ones are more difficult to describe. The common features of the examples found are that the flow polynomial is a polynomial of reasonably high *odd* degree that has a negative derivative at $t = 4$ and is strongly positive at $t = 5$. The graphs on 30 and 34 vertices that were examined have flow polynomials of even degree with positive derivative at $t = 4$ and values that just keep increasing as t increases.

Given that 4 is not an upper limit for flow roots, what would be an appropriate replacement for Welsh's conjecture? The nature of these examples suggests that they will not give flow roots above 5, and yet there seems to be no strong reason to choose any value strictly between 4 and 5. Therefore we propose the following conjecture:

Conjecture 2 *If G is a bridgeless graph with flow polynomial F_G, then $F_G(r) > 0$ for all $r \in [5, \infty)$.*

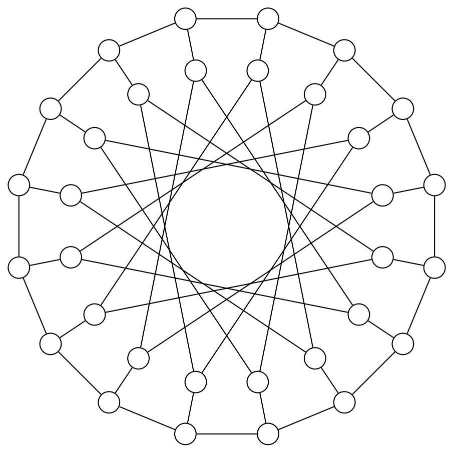

Fig. 4. The generalised Petersen graph $P(16,6)$

The case $r = 5$ is simply Tutte's 5-flow conjecture and so the truth of this conjecture (and the Birkhoff-Lewis conjecture) would give an appealing parallel between the flow roots of general graphs and chromatic roots of planar graphs.

Finally, we have sanity checked the Tutte polynomial computed for $P(16,6)$ by evaluating it at several known points (see §5.3).

4 Algorithmic Observations

In this section, we begin by detailing several well-known theorems about the Tutte polynomial and explain how these can be exploited to improve computational performance.

4.1 Known Reductions

There are numerous well-known properties of the Tutte polynomial definition that can be exploited to prune the computation tree and, hence, improve performance. The first of these exploits the fact that the Tutte polynomial is multiplicative over the blocks (i.e., maximal 2-connected components) of a graph and that these biconnected components can be determined in linear time.

Theorem 1. *Let $G = (V, E)$ be a graph with m blocks G_1, G_2, ..., G_m. Then* $T(G) = \prod_{i=1}^{m} T(G_i)$. □

At present, our system uses a standard algorithm for identifying biconnected components [24], extracting the non-trivial biconnected components (that is, those with more than 2 vertices). The trivial biconnected components are edges and multi-edges whose underlying graph is a forest, and these are processed in a single step using the following lemma, which is immediate from the definitions.

Lemma 1. *Let $G = (V, E)$ be a multi-graph whose underlying graph is a forest with s edges. Denote the multiplicity of each distinct edge in the graph by $d_1, \ldots, d_s$. Then,*

$$T(G) = \prod_{i=1}^{s}(x + y + y^2 + \ldots + y^{d_i - 1})$$

□

It is possible that further gains could be made by using a dynamic algorithm for biconnected components (e.g. [29, 18]). Furthermore, our algorithm makes no particular effort to select edges whose deletion helps to create separating vertices; instead, it simply exploits them when, by chance, they occur. There is a similar, though more complicated, algorithm for detecting separating pairs of vertices and decomposing the graph into *triconnected* components [12, 9], though we have not yet experimented with this.

An *ear* in a graph is a path $v_1 \sim v_2 \sim \cdots \sim v_n \sim v_{n+1}$ where $d(v_1) > 2$, $d(v_{n+1}) > 2$ and $d(v_2) = d(v_3) = \cdots = d(v_n) = 2$. A cycle is viewed as a "special" ear where $v_1 = v_{n+1}$ and the restriction on the degree of this vertex is lifted. If a graph contains a multi-edge or an ear, then all the edges involved can be removed in a single operation. We denote an edge of multiplicity p by e^p and an ear with s edges by E_s. Deletion of a multi-edge or ear is defined naturally as meaning the deletion of all the edges. Contraction of a multi-edge means to delete all the edges and identify the endvertices, while contraction of an ear means to delete all the edges and identify v_1 and v_{n+1}.

Theorem 2. *Suppose that G is a biconnected graph that is either equal to a multi-edge e^p of multiplicity p or properly contains a multi-edge e^p. Then*

$$T(G) = \begin{cases} (x + y + \cdots + y^{p-1}), & G = e^p; \\ (1 + y + \cdots + y^{p-1})T(G/e^p) + T(G - e^p), & \textit{otherwise.} \end{cases}$$

□

Ears are dual to multiple edges and so we have the dual result:

Theorem 3. *Suppose that G is a biconnected graph that is either equal to an ear E_s (which is necessarily a cycle of length s) or properly contains an ear E_s. Then*

$$T(G) = \begin{cases} (y + x + \cdots + x^{s-1}), & G = E_s; \\ (1 + x + \cdots + x^{s-1})T(G - E_s) + T(G/E_s), & \textit{otherwise.} \end{cases}$$

□

In matroid terminology, these results say that an entire parallel class or series class can be processed at once.

These two results follow immediately from the rules for deletion/contraction and Figure 5 visually outlines the proof of Theorem 3. The value of these two theorems is that we can exploit them to further prune the computation tree; we find that they offer significant performance improvements in practice.

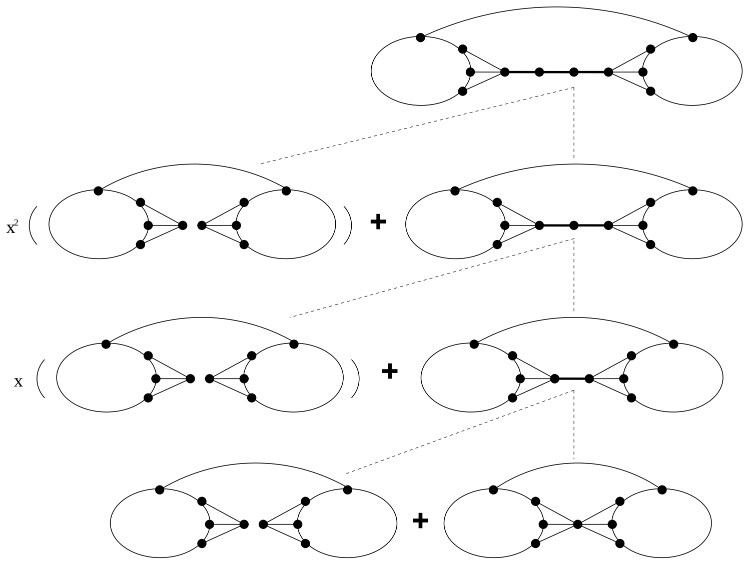

Fig. 5. Reduction of an ear. In the first delete/contract, we select an edge on the ear for removal; on the left branch, this leaves two single-edge biconnected components which immediately yield a factor of x^2.

There are more complex structures that can in principle be processed in a single step, such as when there is an ear in the underlying simple graph, but this ear contains multi-edges in the graph itself.

Theorem 4. *Let $G = (V, E)$ be a multi-graph whose underlying graph is an n-cycle. Denote the multiplicity of each distinct edge in the cycle by $d_1, \ldots, d_n$. Then,*

$$T(G) = \sum_{i=1}^{n} \left(\prod_{j=i+1}^{n} (x + y^{1 \ldots d_j - 1}) \prod_{k=1}^{i-1} (y^{0 \ldots d_k - 1}) \right) + (x + y^{d_n + d_{n-1} - 1}) \prod_{i=1}^{n-2} (y^{0 \ldots d_i - 1})$$

Proof. The proof is by induction. The first step is to use the Tutte recursion to reduce G into two smaller graphs. For the delete graph, we apply Lemma 1. For the contract graph we observe it is simply a $(k-1)$-multicycle to which the inductive hypothesis can be applied. Figure 6 outlines the proof for the special case $n = 6$. □

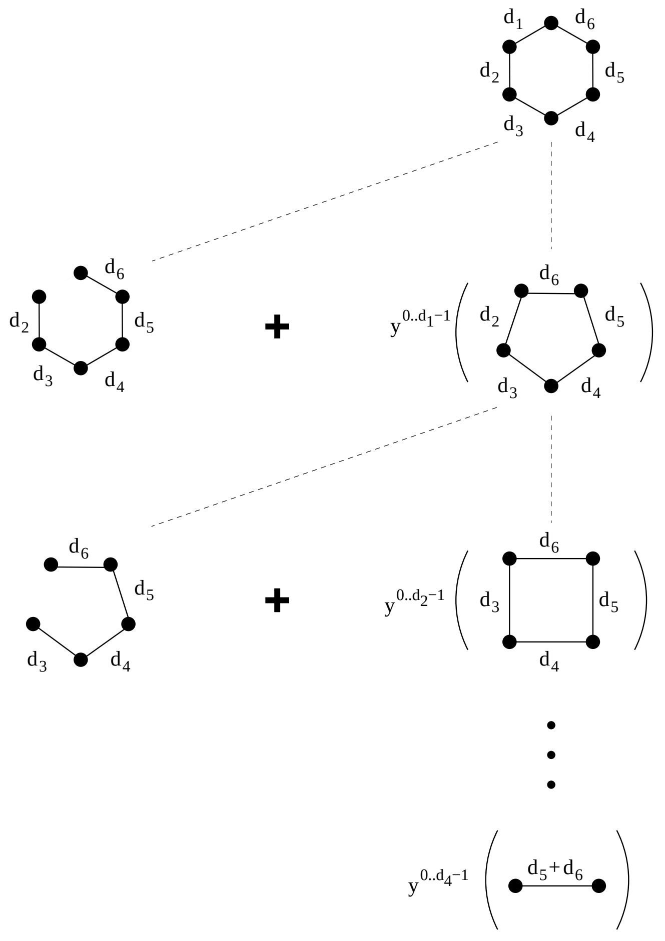

Fig. 6. An outline of the proof for reducing multi-cycles

5 Algorithm Overview

We now provide an overview of our algorithm to illustrate the main choices we have made. We also discuss some of the more practical, but nonetheless important issues which we faced when implementing our algorithm for computing Tutte Polynomials.

As the algorithm operates, it essentially traverses the computation tree in a depth-first fashion (although the whole tree is never held in memory at once). That is, when a delete/contract operation is performed on G, it recursively evaluates $T(G-e)$ until its polynomial is determined, before evaluating $T(G/e)$. Other traversal strategies are possible and could offer some benefit, although we have yet to explore this. At each node in the computation tree, the algorithm maintains and/or generates a variety of information on the graph being processed — such as whether it is connected or biconnected — to help identify opportunities for pruning the tree. In particular, the following approaches are employed:

i) Reductions. Known properties of Tutte polynomials are used to immediately reduce either the whole graph, or a subgraph, to a polynomial. For example, a tree with n edges can be immediately reduced to x^n by Definition 1. Likewise, for a graph containing n loops, we can immediately eliminate these and apply a factor of y^n to the polynomial of the remainder. Such optimisations simplify the computation tree and can speed up the various operations performed on graphs in the subtree (of course, if the whole graph is reduced there is no subtree!). In our system, trees, loops, cycles and multedges, multi-cycles, and multi-ears can be reduced immediately.

ii) Biconnectedness. Following Theorem 1, we break graphs which are not biconnected into their non-trivial biconnected components and the residual forest. The polynomials for the biconnected components are then computed independently, which is helpful as their computation trees may be significantly smaller. At present, our system uses a standard algorithm for identifying biconnected components [24], extracting the non-trivial biconnected components (that is, those with more than 2 vertices).

iii) Cache. Computed polynomials for graphs encountered during the computation are stored in a cache. Thus, if a graph isomorphic to one already resolved is encountered, we simply recall its polynomial from the cache. This optimisation typically has a significant effect, since the whole branch of the computation tree below the isomorph is pruned. To determine graph isomorphism, we employ McKay's *nauty* program [14]. The size of the cache employed and the replacement strategy used when the cache fills require further study as both can have significant effects.

iv) Edge Selection. As indicated already, the choice of edge for deletion and contraction affects the likelihood of reaching a graph isomorphic to one already seen (see §2). Furthermore, it affects the chance of exposing structures (e.g. cycles and trees) which can be immediately reduced. We have found that two edge-selection heuristics perform particularly well: *vertex order* and *minimising single degree*. In §5.2, we provide explore these and other heuristics in more detail.

The coefficients computed for the Tutte polynomial of even a small graph are large and can easily go beyond the size of a machines 32-bit or 64-bit word size. To address this, we have implemented a simple library for arbitrary sized integers.

5.1 Graph Isomorphism

To implement the cache for polynomials of graphs at nodes of the computation tree, we employ a simple hash map. This is keyed upon a canonical labeling of the graph obtained using *nauty* [14]. Since *nauty* accepts only simple graphs, we transform multigraphs into simple graphs by inserting additional vertices as necessary. We refer to such graphs as being "constructed". To avoid a constructed graph from clashing with a normal simple graph having the same number of vertices and edges, we exploit the fact that *nauty* allows vertices to be coloured and will reflect the colour class of a vertex in the canonical form. Thus, vertices added to represent multi-edges are coloured differently from normal vertices. An interesting issue here is that, at each node in the computation tree, we must *recompute the canonical labelling from scratch* as the graph, by definition, is different from its parent. While we have not explored this as yet, there is potential for exploiting an incremental graph isomorphism algorithm which could more efficiently determine the canonical labelling of a graph given that of its parent.

An important problem we face is what to do when the cache fills up, which happens frequently for large graphs, even when large amounts (e.g. > 2GB) of memory are available. To resolve this, we employ techniques from garbage collection: items in the cache are displaced and, to avoid memory fragmentation, those left are compacted into a contiguous block (this is similar to mark-and-sweep garbage collection). To determine which graphs to displace, we employ a simple policy based on counting the number of times a graph in the cache has been "hit". When the cache is full, graphs with a low hit count are displaced before those with higher counts.

Another problem is how much of the cache to displace. Clearly, displacing less means the cache will fill more frequently and, on average, contain more old items. Of course, the more items there are in the cache, the greater the potential for collisions when searching for an isomorph. In contrast, displacing more of the cache each time means that many graphs which may turn out to be useful later on will not survive. In our implementation, the default policy is to displace 30% of the cache in one go when it becomes full.

5.2 Edge-Selection Heuristics

The order in which edges are selected during the computation can have a dramatic effect on the runtime. Here, we consider several edge-secltion heuristics and identify two which perform particularly well. In the following Section, we also provide some experimental evidence of this.

Vertex Order Heuristic. The first heuristic we consider is the *vertex order* heuristic, or VORDER for short. In this heuristic, the vertices of the graph are given a fixed, predefined ordering. Then, as the computation proceeds, edges are selected from the lowest vertex in the order until it becomes disconnected; once this occurs, edges are selected from the next lowest vertex until it becomes disconnected and so on. For contractions, the resulting vertex maintains the lowest position in the order of those contracted. Furthermore, when selecting an edge for some vertex v, we choose one whose other end-point is also the lowest of any incident on v.

Figure 7 illustrates this process operating on a simple graph. In the figure, the position of each vertex in the order is shown next to it. Thus, for the first delete/contract, an edge from vertex 1, namely $(1, 2)$, is selected since 1 is lowest in the ordering and 2 is below the others adjacent to 1. On the deletion side, that edge is simply removed and, thus, the next edge selected is also from 1 (this time, it's $(1, 3)$; on the contract side, 1 and 2 are contracted, with the resulting vertex coming at the position previously occupied by 1 in the order.

In considering Figure 7 there is an interesting observation to make: the isomorphic hits that occur are almost always *immediately isomorphic* (i.e. no permutation of vertices is required). To see why this is significant, let us imagine a larger graph which differs only in that some big structure is reachable from vertex 5. Since this structure is not touched during the computation shown, we know that the same isomorphic hits will apply. As we will see, this is not always the case for other heuristics.

From the above line of reasoning, we conclude that VORDER promotes the likelihood of an isomorphic match occurring, and that this explains why it performs so well on dense graphs (as we will see in §6). Whilst studying this heuristic, we have also made some other observations. Firstly, contracting two vertices such that the resultant vertex assumes the *highest* position of either generally does not perform as well. Secondly, using an ordering where vertices with higher degree come lower in the ordering generally also gives better performance.

Degree Selection Heuristics. The second kind of heuristic we consider are those which select edges based on their degree. The idea is to choose an edge which either minimises or maximises the degree in some way, and we consider here a family of related heuristics:

– **Minimise Single Degree** (MINSDEG): here the edge selected at each point in the computation tree has an end-point with the minimal degree of any vertex.
– **Minimise Degree** (MINDEG): here the edge selected at each point in the computation has end-points whose degree sum is the least of any edge.

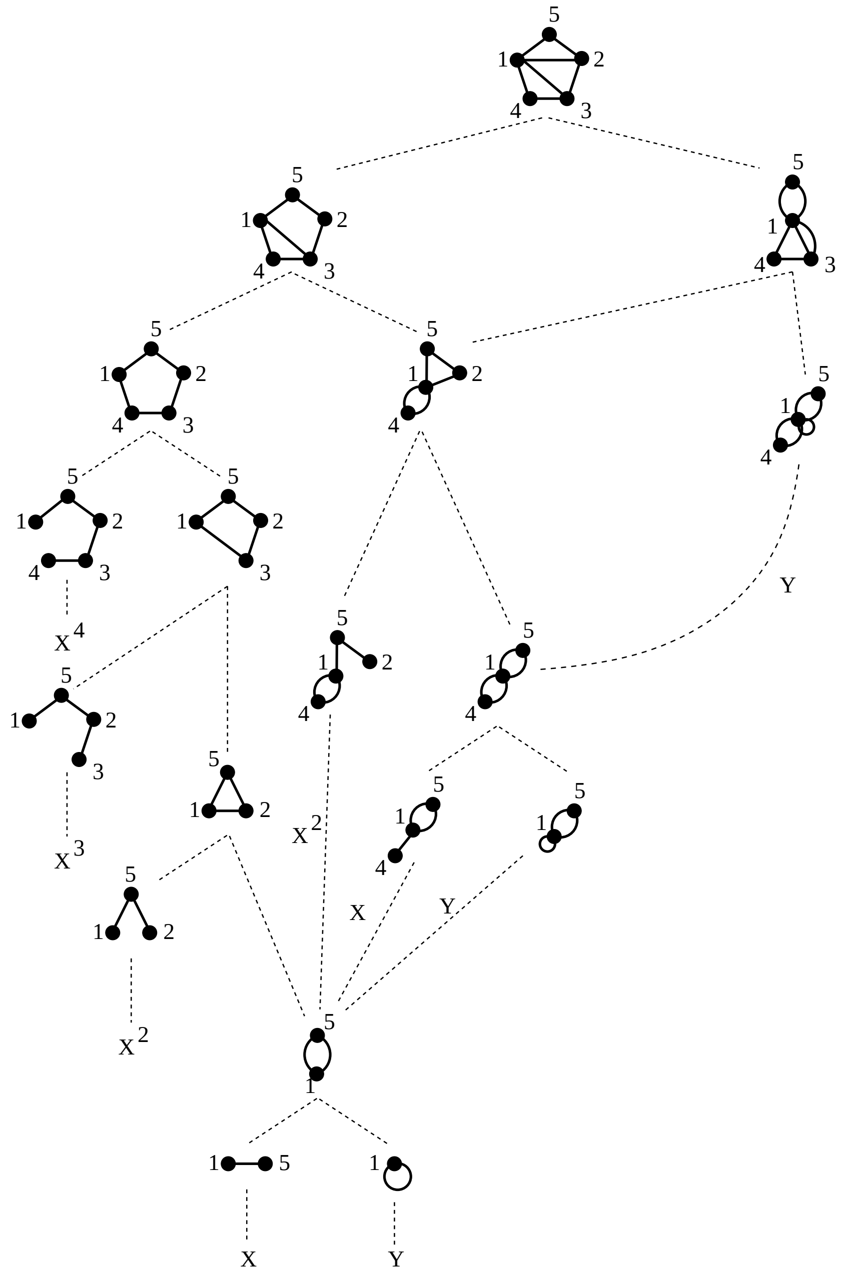

Fig. 7. Illustrating the VORDER heuristic on a small graph. The position of each vertex in the order is shown next to it. Thus, the heuristic works aggressively on the vertex labelled 1, since this is lowest in the order.

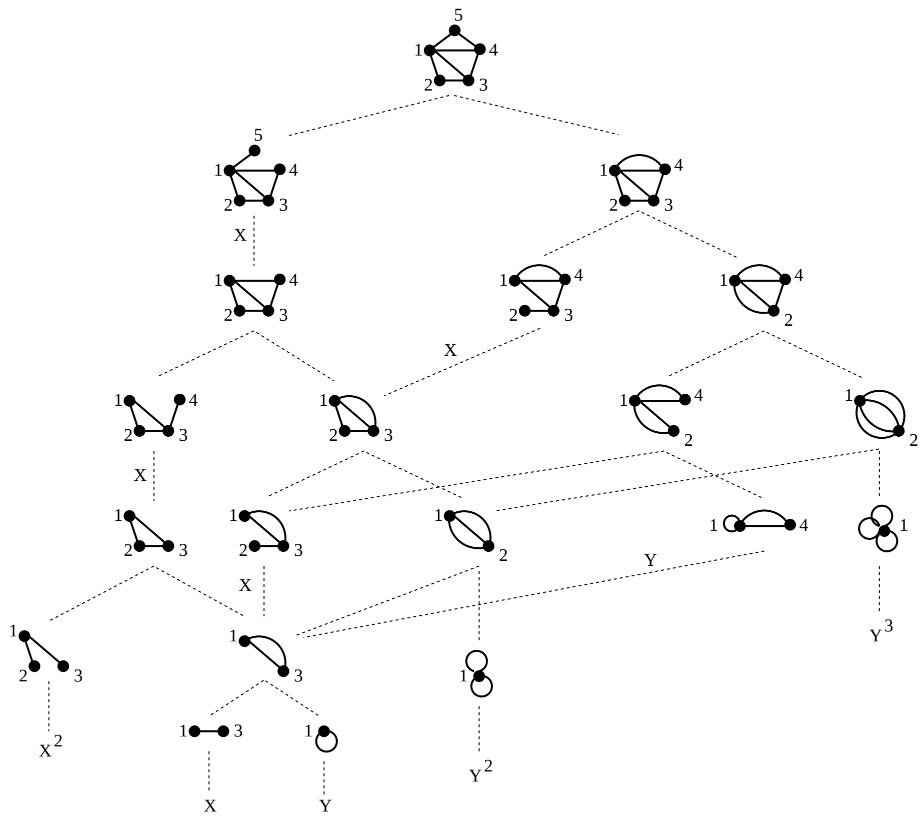

Fig. 8. Illustrating the MINSDEG heuristic on the small graph from Figure 7. This heuristic selects edges whose end-point has the lowest degree of any vertex. The vertices of each graph have been labelled to indicate how individual vertices progress through the computation.

- **Maximise Single Degree** (MAXSDEG): here the edge selected at each point in the computation has an end-point with the maximum degree of any vertex.
- **Maximise Degree** (MAXDEG): here the edge selected at each point in the computation has end-points whose degree sum is the most of any edge.

As we will see in §6, the MINSDEG heuristic is generally the best choice of these and, on sparse graphs, it outperforms the VORDER heuristic from §5.2. Figure 8 illustrates MINSDEG operating on the graph from Figure 7. Here, the shape of the computation tree seems quite different from Figure 7. For example, there are more terminations on single vertices with loops and, likewise, graphs with higher degree multi-edges are encountered. Furthermore, whilst the number of isomorphic hits is actually slightly higher than for Figure 7, most of these are not *immediately isomorphic*. This, we believe, indicates the heuristic will not promote isomorphic matching as well as VORDER. This is because, in larger graphs, more structures will be present that prevent isomorphic hits from occurring until much later in the computation. For example, in Figure 8 consider the isomorphic hit that occurs between the third and fourth levels. In this case, we have one graph with vertices $\{1, 2, 3\}$ and another with vertices $\{1, 3, 4\}$. Thus, if we imagine computing the polynomial of a larger graph which differs by having some structure adjacent to vertex 4, then we can see that this isomorphic hit would not occur. In practice, with MINSDEG, isomorphic hits typically occur at levels which are further down the computation tree than with VORDER. Nevertheless, we find that MINSDEG does outperform VORDER on sparse graphs (see §6), although the reason for this remains unclear. We believe, however, that it may be because MINSDEG tends to promote the breaking up of cycles which exposes large tree portions that can be automatically reduced.

5.3 Correctness

The output of any complex computer program should be treated with caution, if not outright suspicion, and carefully examined for internal consistency and cross-referenced against known values. Aside from manual testing on small graphs, we employ a number of "sanity checks" to increase our confidence in its correctness. The easiest check is to compute $T(G; 2, 2)$, which should give $2^{|E(G)|}$ [4], and compare this with a direct computation of $2^{|E(G)|}$. Another check is to compute $T(G; 1, 1)$, which gives the number of spanning trees in the graph. Then, we can check that these evaluations are constant for a given graph, regardless of what parameters are chosen for a particular run of our algorithm (e.g. cache size, which reductions are applied, edge selection strategy, vertex ordering, etc).

As an illustration, we have sanity-checked the generalised Petersen graph $P(16, 6)$ discussed in §3 as follows:

- $T(2, 2) = 2^{48}$ as required.
- $T(1, 1) = 115184214544$ is the number of spanning trees of $P(16, 6)$ as determined by the matrix-tree theorem.
- $(-1)\ T(1 - x, 0)$ equals the chromatic polynomial of $P(16, 6)$ which was independently verified by two separate programs.
- $T(-1, -1) = -2$ equals the expected value $(-2)^d$ where $d = 1$ is the dimension of the bicycle space of $P(16, 6)$ which can be computed by elementary linear algebra.
- $T(0, -3) = -480$ is the number of 4-flows of $P(16, 6)$ which is equal to the number of edge-3-colourings of $P(16, 6)$ which can easily be verified by a direct search.

6 Experimental Results

In this section, we report on some experimental results obtained using our system. In particular, we look at the effect of using the isomorph cache, and the edge-selection heuristics outlined in §5.2. The objective here is to give an indication of the effect that these features have on performance. We consider random connected graphs, random 3-regular and random planar graphs. The machine used for these experiments was an Intel Pentium IV 3GHz with 1GB of memory, running NetBSD v4.99.9.

6.1 Experimental Procedure

To generate *random connected graphs*, we employed the tool `genrang` (supplied with *nauty*) to construct random graphs with a given number of edges; from these, we selected connected graphs until there were 100 for each value of $|E|$ or $|V|$ (depending upon experiment). The `genrang` tool constructs a random graph by generating a random edge, adding it to the graph (if not already present), and then repeating this until enough edges have been added. We also used `genrang` to generate random simple regular graphs — this essentially works by generating a random regular multigraph and then throwing it out if it contains loops or multiple edges. Generating *random planar graphs* required a different approach since the number of randomly generated graphs that are planar is extremely small. Therefore, we employed a markov-chain approach; here, an edge was selected at random and added to the graph, provided it was not already present and the graph remained planar; otherwise, it was removed — again, provided that the graph remained planar. This procedure was repeated for $3n^2$ steps (which, according to [6], is well beyond the equilibrium point).

6.2 Experimental Results

Figure 9 presents the data from our experiments on random connected graphs. Data is provided for timings with and without the cache enabled. From the figure, it is immediately obvious that the cache has a critical effect on the performance of the algorithm. As expected, performance deteriorates as graph density increases; however, the algorithm appears to perform surprisingly well on very dense graphs. This stems from the increased regularity present in dense graphs which gives rise to a greater number of isomorphic hits in the cache.

Figure 10 reports the data from our experiments on random planar and 3-regular. From the graphs, it is clear that computing the Tutte polynomial for large graphs quickly becomes intractable. Nevertheless, using the isomorphism cache extends the size of graphs which can be computed. This is of significant value in practice, since it extends the range of graphs over which users of the tool can, for example, test a conjecture they are considering.

Figure 11 presents the data from our experiments on random connected graphs. Data is provided for each of the five heuristics, and observe that a log scale is used on the y-axis. Also, a timeout of 5000s was used to deal with long running computations, and this is explains why the data for Maxdeg flattens out at the top. From the figure, it is immediately obvious that the Vorder heuristic performs particularly well compared with the others. The reason for this, we believe, is that it promotes the chance of reaching a graph which is

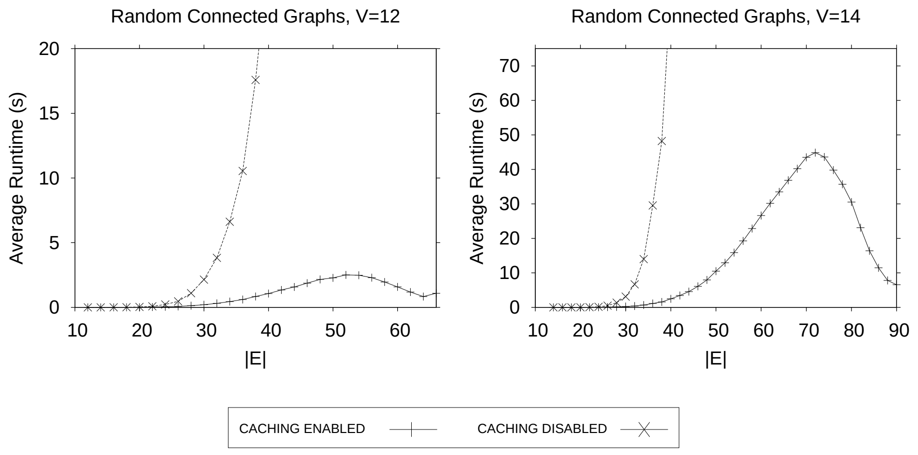

Fig. 9. Random Connected Graphs at $|V| = 12$, $|V| = 14$, and varying $|E|$, where each data-point is averaged over 100 graphs. Data illustrating the time taken with and without the cache are shown. For each experiment, the cache size was fixed at 768MB.

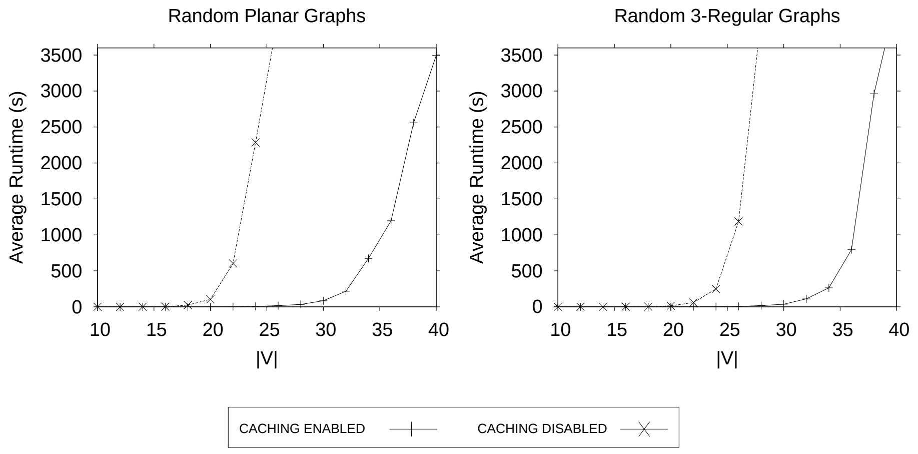

Fig. 10. Random Planar and 3-Regular Graphs with varying $|V|$, where each data-point is averaged over 100 simple graphs. Data illustrating the time taken with and without the cache are shown. For each experiment, the cache size was fixed at 768MB.

isomorphic to one already seen. Note, very dense graphs tend to be easier to solve, since their increased regularity leads to a greater number of isomorphic hits in the cache.

Figure 12 reports the data from our experiments on random 3-regular and 4-regular graphs. Data is provided for each of the five heuristics, and observe that a log scale is used on the y-axis. For the 3-regular graphs, we can see that the MINSDEG heuristic gives a significant performance benefit over the others. However, on the 4-regular graphs (which are denser) we see the gap between MINSDEG and VORDER closes. Furthermore, it is fairly evident that computing these graphs is considerably more expensive than for the 3-regular graphs.

7 Conclusion

Algorithms for computing Tutte polynomials have been, in general, rather simplistic. We have demonstrated a number of techniques which can greatly reduce the size of the computation tree. This, in turn, leads to an algorithm which can tackle significantly larger graphs than previously possible. While this task may seem futile (since the problem is #P-Hard), it is important to remember that, in practice, the applications of this tool (e.g. for classifying DNA knots) have finite requirements; thus, we are moving towards a system which can handle sufficiently large graphs to be of use to practitioners.

A number of interesting questions remain for further research. Firstly, the edge-selection heuristics we have explored seem rather simple; can we identify other heuristics which lead to even better selection orders? Secondly, at each node in the computation tree, we compute a canonical labelling of the corresponding graph so it can be stored in the cache (this enables later identification of isomorphs). But, should we do this at every node? For example, could we maintain the canonical labelling incrementally? Thirdly, can we strengthen the termination conditions? For example, computing the Chromatic polynomial of a complete graph is very easy. Thus, for dense graphs, *it makes more sense to move towards complete graphs than empty graphs.* Indeed, we have trialled this for computing chromatic polynomials with considerable success. However, it's unclear how this can be applied to the general Tutte computation.

Nevertheless, even with this list of interesting unanswered questions, the implementation described gives researchers an effective and efficient way to experiment with Tutte polynomials both to answer questions and test conjectures for a wide range of sizes of graphs [15]. The complete implementation of our algorithm can be obtained from `http://www.mcs.vuw.ac.nz/~djp/tutte`. This also supports efficient computation of chromatic and flow polynomials, based on the same techniques presented in this paper.

Acknowledgements. We wish to thank the Isaac Newton Institute for Mathematical Sciences, University of Cambridge, for generous support during the programme on Combinatorics and Statistical Mechanics (January – June 2008),

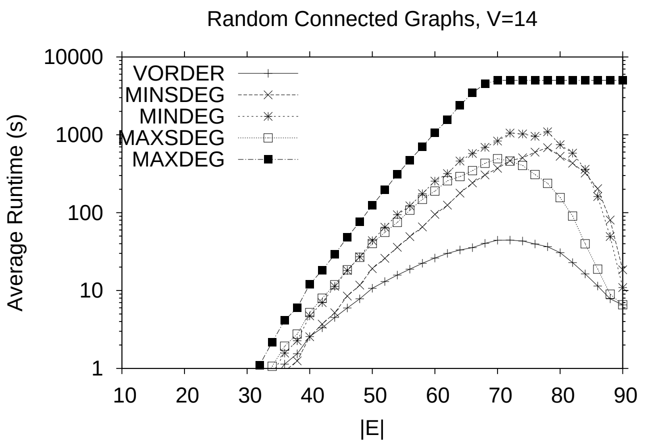

Fig. 11. Random Connected Graphs with $|V| = 14$ and varying $|E|$, where each data-point is averaged over 50 graphs. Data illustrating the time taken for the five different heuristics discussed in §5.2 are shown. For each experiment, the cache size was fixed at 512MB. A timeout of 5000s was used to deal with long running computations, and this is explains why the data for MAXDEG flattens out at the top.

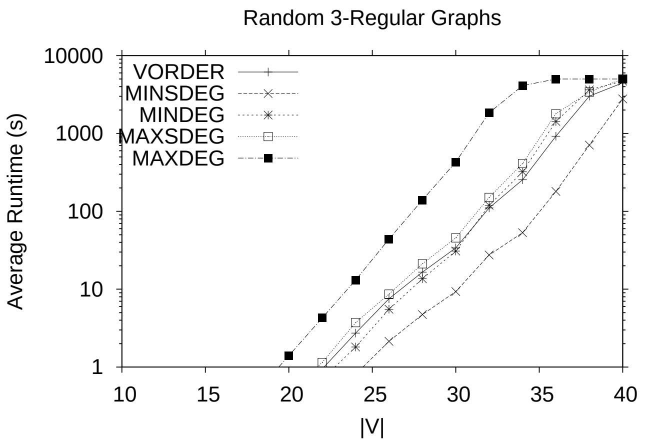

Fig. 12. Random 3-Regular Graphs with varying $|V|$, where each data-point is averaged over 50 graphs. Data illustrating the time taken for the five different heuristics discussed in §5.2 are shown. For each experiment, the cache size was fixed at 512MB. A timeout of 5000s was used to deal with long running computations, and this is explains why the data for MAXDEG flattens out at the top.

where this work was carried out. This enabled the authors to gather together in the same hemisphere and discuss the problem in person! Special thanks in particular go to Alan Sokal and Klas Markström for many interesting discussions. Gary Haggard was also supported by a National Science Foundation (NSF) grant (#0737730) during this time.

References

1. K. Appel, W. Haken, and J. Koch. Every planar map is four colorable. *Illinois Journal of Mathematics*, 21:439–567, 1977.
2. A. Björklund, T. Husfeldt, P. Kaski, and M. Koivistor. Computing the Tutte polynomial in vertex-exponential time. In *Proceedings of the IEEE Symposium on the Foundations of Computer Science (FOCS)*, pages 677–686, 2008.
3. A. Björklund, T. Husfeldt, P. Kaski, and M. Koivistor. Computing the Tutte polynomial in vertex-exponential time. Technical report, `arxiv.org/PS_cache/arxiv/pdf/0711/0711.2585v4.pdf`, 2008.
4. B. Bollobás. *Modern Graph Theory*. Number 184 in Graduate Texts in Mathematics. Springer, New York NY, 1998.
5. S.-C. Chang and R. Shrock. Zeros of Jones polynomials for families of knots and links. *Physica A:Statistical Mechanics and its Application*, 301:196–218, 2001.
6. A. Denise, M. Vasconcellos, and D. J. A. Welsh. The random planar graph. *Congressus Numerantium*, 113:61–79, 1996.
7. O. Giménez, P. Hlinený, and M. Noy. Computing the Tutte polynomial on graphs of bounded clique-width. *SIAM J. Discrete Math*, 20(4):932–946, 2006.
8. G. Gordon. Series-parallel posets and the Tutte polynomial. *Discrete Mathematics*, 158(1-3):63–75, 1996.
9. C. Gutwenger and P. Mutzel. A linear time implementation of spqr-trees. In *Proceedings of the 8th International Symposium on Graph Drawing (GD)*, pages 77–90, London, UK, 2001. Springer-Verlag.
10. G. Haggard and T. Mathies. The computation of chromatic polynomials. *Discrete Math*, 199:227–231, 1999.
11. G. Haggard and R. Read. Chromatic polynomials of large graphs. ii. isomorphism abstract data type for small graphs. *J. Math. Comput*, 3:35–43, 1993.
12. J. E. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM Journal on Computing*, 2(3):135–158, 1973.
13. Kauffman. A Tutte polynomial for signed graphs. *DAMATH: Discrete Applied Mathematics and Combinatorial Operations Research and Computer Science*, 25, 1989.
14. B. McKay. Nauty users guide (version 1.5). Technical report, Dept. Comp. Sci., Australian National University, 1990.
15. B. M. E. Moret. Towards A discipline of experimental algorithmics. In *Proceedings of the DIMACS Implementation Challenge*, volume 59 of *DIMACS Monographs*, pages 197–213. American Maths Society, 1990.
16. K. Murasugi. On invariants of graphs with applications to knot theory. *Transactions of the American Mathematical Society*, 314(1):1–49, 1989.
17. S. D. Noble. Evaluating the Tutte polynomial for graphs of bounded tree-width. *Combinatorics, Probability & Computing*, 7(3):307–321, 1998.
18. M. Rauch. Improved data structures for fully dynamic biconnectivity. In *Proceedings of the ACM symposium on Theory of computing*, pages 686–695. ACM, 1994.

19. R. Read. An improved method for computing chromatic polynomials of sparse graphs. Technical Report CORR 87-20, Dept. Comb. & Opt., University of Waterloo, 1987.
20. G. F. Royle. Computing the Tutte polynomial of sparse graphs. Technical Report CORR 88-35, Dept. Comb. & Opt., University of Waterloo, 1988.
21. K. Sekine, H. Imai, and S. Tani. Computing the Tutte polynomial of a graph of moderate size. *Lecture Notes in Computer Science*, 1004:224–233, 1995.
22. A. D. Sokal. The multivariate Tutte polynomial (alias Potts model) for graphs and matroids. In *Surveys in combinatorics 2005*, volume 327 of *London Math. Soc. Lecture Note Ser.*, pages 173–226. Cambridge Univ. Press, Cambridge, 2005.
23. A. D. Sokal. The multivariate Tutte polynomial (alias Potts model) for graphs and matroids. In *Surveys in combinatorics 2005*, volume 327 of *London Math. Soc. Lecture Note Ser.*, pages 173–226. Cambridge Univ. Press, Cambridge, 2005.
24. R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
25. W. Tutte. A contribution to the theory of chromatic polynomials. *Canadian Journal of Mathematics*, 6:80–81, 1954.
26. Vertigan. The computational complexity of Tutte invariants for planar graphs. *SICOMP: SIAM Journal on Computing*, 35, 2006.
27. D. Welsh. The Tutte polynomial. *Random Structures Algorithms*, 15(3-4):210–228, 1999. Statistical physics methods in discrete probability, combinatorics, and theoretical computer science (Princeton, NJ, 1997).
28. D. J. A. Welsh and C. Merino. The Potts model and the Tutte polynomial. *Journal of Mathematical Physics*, 41(3):1127–1152, 2000.
29. J. Westbrook and R. Tarjan. Maintaining bridge-connected and biconnected components on-line. *Algorithmica*, 7(1):433–464, 1992.