Edge-Selection Heuristics for Computing Tutte Polynomials

David J. Pearce
Computer Science Group,
Victoria University of Wellington,
New Zealand
djp@ecs.vuw.ac.nz

Gary Haggard
Bucknell University,
USA
haggard@bucknell.edu

Gordon Royle
School of Mathematics and Statistics,
University of Western Australia
gordon@maths.uwa.edu.au

October 7, 2009

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. We have developed the most efficient algorithm to-date for computing the Tutte polynomial of a graph. An important component of the algorithm affecting efficiency is the choice of edge to work on at each stage in the computation. In this paper, we present and discuss two edge-selection heuristics which (respectively) give good performance on sparse and dense graphs. We also present experimental data comparing these heuristics against a range of others to demonstrate their effectiveness.

1 Introduction

The *Tutte polynomial* of a graph G (which may have loops and multiple edges) is a 2-variable polynomial $T(G, x, y)$ that encodes a significant amount of information about the graph — indeed in a strong sense it "contains" *every* graphical invariant that can be computed by deletion and contraction. In particular, the Tutte polynomial can be explicitly evaluated at particular points (x, y) to give numerical graphical invariants such as the number of spanning trees, the number of forests, the number of connected spanning subgraphs, the dimension of the bicycle space and many more. In addition, the Tutte polynomial 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 also plays an important role in the field of statistical physics where it appears as the partition function of the q-state Potts model $Z_G(q, v)$. 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.

In knot theory, the Tutte polynomial appears in yet another guise as the Jones polynomial of an alternating knot. However, computing the Jones polynomial of a non-alternating knot, which may have some application in analysing knotted strands of DNA, requires the use of a signed Tutte

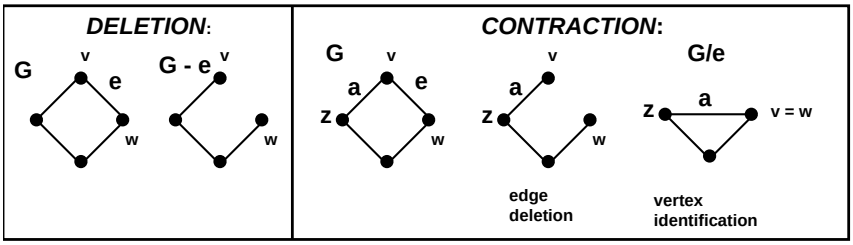


Figure 1: Deletion and Contraction of an Edge

polynomial, which is considerably more involved. A more complete discussion of the theory and applications of Tutte polynomials is found in Bollabás [2].

Of all the invariants associated with the Tutte polynomial, the chromatic polynomial plays a special role in both 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.

We have developed the most efficient algorithm available for computing the Tutte polynomial of a graph, and a detailed discussion of how it works is given in [3]. Our implementation has been used by several different researchers (e.g. [4]) to quickly test conjectures over a large range of graphs. In particular, we ourselves managed to find several counterexamples to a conjecture of Dominic Welsh on the location of the real flow roots of a graph [3]. Since these counterexamples had 28 vertices or more, it seems unlikely they would have been found by hand.

In [7], 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. Similarly, an algorithm is given in [6] to compute Tutte polynomials of moderate sized graphs, but again is not effective much beyond 14 vertices. In contrast, our algorithm can deal with graphs of this size in a relatively short amount of time, and easily scales beyond this (see §5). More recently, an alternate strategy that uses "roughly $2^{n+1}n$ words of memory for an n-vertex graph" has been developed [1]. 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.

In this paper, we concentrate on an important aspect of our algorithm not touched upon in [3] — namely, the *edge-selection heuristics*. We discuss several heuristics and report on experiments comparing them. Our findings indicate that, of these, two stand out from the rest. Furthermore, whilst one appears to perform particularly well on dense graphs, the other performs better on sparse graphs. Finally, our implementation also supports efficient computation of chromatic and flow polynomials, based on the same techniques presented in this and our previous paper, and can be obtained from `http://www.ecs.vuw.ac.nz/~djp/tutte`.

2 Preliminaries

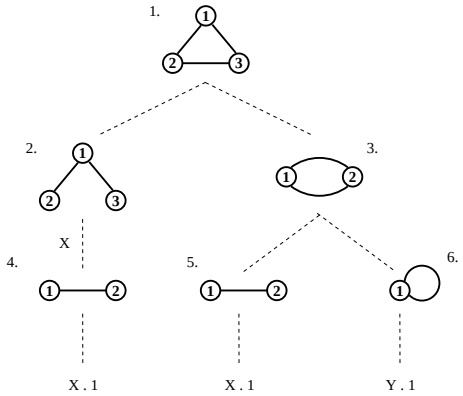
Let $G = (V, E)$ be an *undirected multi-graph*; that is, V is a set of vertices and E is a multi-set of unordered pairs (v, w) with $v, w \in V$. An edge (v, v) is called a *loop*. If an edge (u, v) occurs more than once in E it is called a *multi-edge*. The *underlying graph* of G is obtained by removing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

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

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 [9], and to choose any edge to operate on at each stage. The following illustrates this recursive procedure being applied to a simple graph:



Here, we have labelled each graph in the computation tree with a unique number. Looking at G_1 (i.e. the graph labelled 1), we see that it contains no loops or bridges (edges which, if deleted, would disconnect the graph). Thus, the delete/contract rule from Definition 1 applies:

$$T(G_1) = T(G_2) + T(G_3)$$

where $G_2 = G_1 - (2, 3)$ and $G_3 = G_1/(2, 3)$. Then, looking at G_2 we see that both edges are bridges and, thus, the bridging rule from Definition 1 applies to give:

$$T(G_2) = x \cdot T(G_4)$$

where $G_4 = G_2 - (1, 3)$. Again, the bridging rule applies to G_4 giving:

$$T(G_4) = x \cdot T(G_4 - (1, 2)) = x \cdot 1$$

Here, $T(G_4 - (1, 2)) = 1$ since its edge-set is empty. Continuing along these lines, we end up with the following Tutte polynomial for G_1:

$$T(G_1) = x^2 + x + y$$

As an example of its utility, evaluating this polynomial at $(x = 1, y = 1)$ gives the total number of spanning trees for G_1 (i.e. 3). Many more evaluation points are known in the literature (see e.g. [2]), and there is active research studying such properties of the polynomial.

Figure 2 provides a larger example to further illustrate the procedure, and to highlight some important points. The Tutte polynomial for this graph is:

$$\begin{aligned} T(G_1) \quad = \quad & y + 2y^2 + y^3 + x + 4xy + 2xy^2 + 3x^2 + \\ & 3x^2y + 3x^3 + x^4 \end{aligned}$$

In Figure 2, several aspects of the problem become apparent. Firstly, in several cases we apply known properties of the Tutte polynomial to reduce a graph in a single step. For example, $T(G_8) = x^4$ as it is well-known that a tree with n edges must reduce to x^n. Secondly, it often

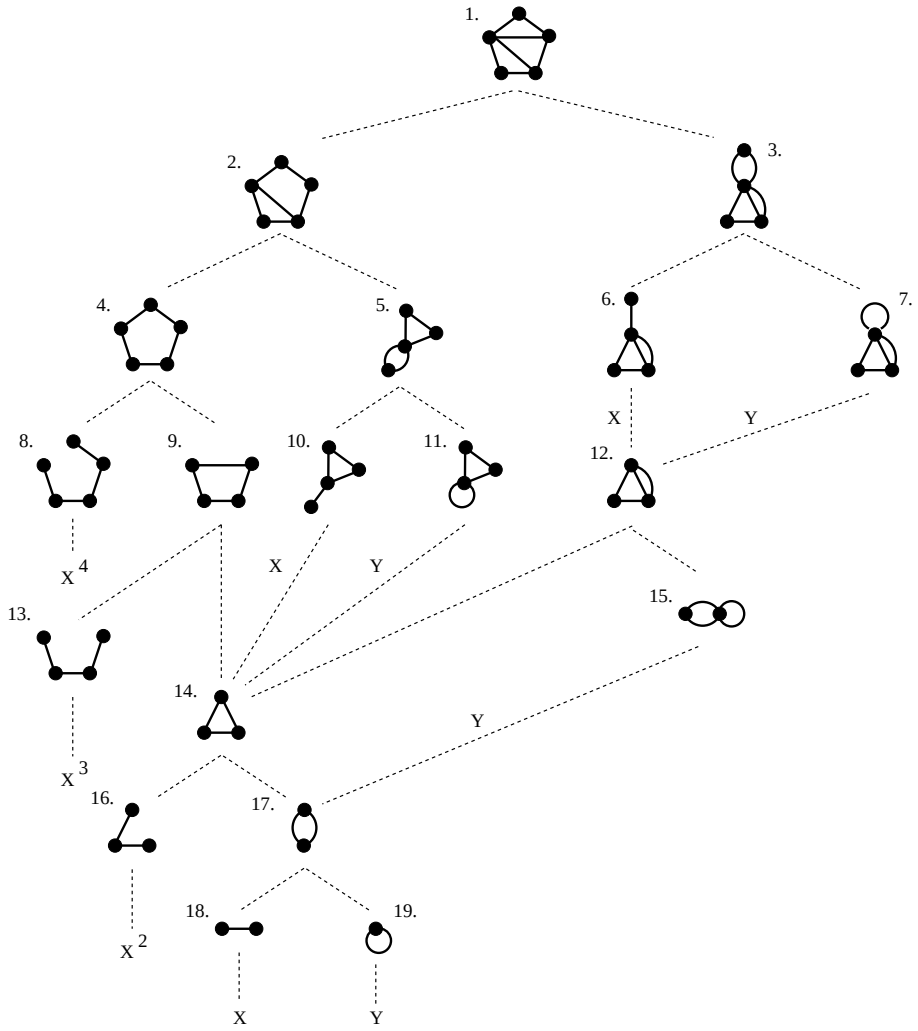


Figure 2: Illustrating the application of Definition 1 to a small graph. Observe that vertices are not drawn once they become isolated, as they play no further role.

arises during the computation that we reach a graph G which is isomorphic to one already seen. Thus, recomputing $T(G)$ from scratch each time is wasteful and should be avoided when possible. Instead, we want to store the polynomial for G the first time it is computed, and then reuse this cached result whenever G is encountered again. For example, in Figure 2 the triangle is reached four times during the computation.

The order in which the rules of Definition 1 are applied does not affect the polynomial which results; however, it can significantly affect the size of the computation required. An "efficient" order can reduce work in a number of ways. In particular, choices leading to graphs isomorphic to those already encountered are clearly favourable. In Figure 2, there are several choices we might have made which would have reduced the number of isomorphic "hits". For example, had we not eliminated the loop in G_7 immediately, we would not have reached a graph isomorphic to G_{12}. Likewise, in G_{10}, choosing any of the other edges to delete/contract upon would not have lead to the triangle and, hence, another isomorphic hit.

In this paper, we are concerned with heuristics that promote good edge selections and, hence, reduce the amount of computation. We explore a range of edge selection heuristics and observe that, of these, two perform considerably better than the others. The first, called MINSDEG, minimises the degree of either end-point; that is, it chooses an edge where one endpoint has the smallest degree of any. The second, called VORDER, relies on a fixed ordering of the vertices; starting from the first vertex in the order, it continuously selects edges from the same vertex until none remain, before moving on to the next vertex in the ordering. This is discussed in more detailed in §4.

There are many other well-known properties of the Tutte polynomial definition that can be exploited to further prune the computation tree and, hence, improve performance. An important example 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 biconnnected 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 it follows that $T(G) = \prod_{i=1}^{m} T(G_i)$.* □

The above was first shown by Tutte [9]. Several other interesting properties that can be exploited to further prune the computation tree are discussed further in [3]. In particular, in certain situations, we can immediately reduce the whole graph, or a subgraph, to a polynomial:

- **Trees**. A tree with k edges can be immediately reduced to x^k by Definition 1.
- **Loops**. If the graph contains a vertex with k loops, then we can immediately remove them and apply a factor of y^k by Definition 1.
- **Cycles**. A cycle with k vertices can be immediately reduced to $y + x + x^2 + \ldots + x^{k-1}$.
- **Multi-edges.** If the graph contains k copies of an edge e, then we can immediately remove all but one. If the remaining occurrence of e is a bridge, then we must apply apply a factor of $x+y+y^2+\ldots y^{k-1}$ when it is removed (instead of just x); otherwise, when we delete-contract upon it, we must apply a factor of $1 + y + y^2 + \ldots y^{k-1}$ to contract graph.
- **Ears**. An ear is a connected subgraph with vertices $v_1, v_2, \ldots, v_k$, where $v_2, \ldots v_{k-1}$ all have degree two. If the graph contains an ear of length k, we can immediately replace this with a single edge (v_1, v_k). If this is a bridge, then we must apply a factor of $y + x + x^2 + x^{k-1}$ when it is removed (instead of just x); otherwise, when we delete-contract upon this edge, we must apply a factor $1 + x + x^2 + x^{k-1}$ to the delete graph.
- **Multi-cycles.** A multi-cycle (i.e. a cycle with at least one multi-edge) with k vertices can be immediately reduced to a polynomial. The resulting polynomial is somewhat more complex than the others, and the reader should consult [3] for details.
- **Multi-ears.** A multi-ear (i.e. an ear with at least one multi-edge) with k vertices can be immediately reduced to a polynomial factor. Again, the factor is somewhat more complex than we have space for here, and the reader should consult [3] for details.

3 Algorithm Overview

We now provide a brief overview of our algorithm to illustrate the main choices we have made. A more detailed discussion of how the algorithm works can be found in [3].

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. We exploit those properties discussed at the end of §2 which allow us to immediately reduce either the whole graph, or a subgraph, to a polynomial. For example, our algorithm immediately reduces a tree with k edges to x^k. Likewise, we also eliminate any loops in the graph immediately, whilst applying a factor of y^k. 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!).

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 [8], extracting the non-trivial biconnected components (that is, those with more than 2 vertices).

iii) Isomorph 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.

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. Furthermore, it affects the chance of exposing structures (e.g. cycles and trees) which can be immediately reduced. *The contribution of this paper is an exploration of several edge-selection heuristics and, in particular, the identification of two which perform well.*

To implement the cache for polynomials of graphs at nodes of the computation tree, a hash map is used. This is keyed upon a canonical labeling of the graph obtained using *nauty* [5]. Since *nauty* accepts only simple graphs, we transform multigraphs into simple graphs by inserting additional vertices as necessary. Note that the isomorph check is performed at every step in the computation; whilst this may seem expensive (and, indeed it is), the benefits significantly outweigh the costs. A proper study of the benefits from using the isomorph cache can be found in [3].

4 Edge-Selection Heuristics

In this section, we will present a range of edge-selection heuristics, some of which we have found to perform well on certain classes of graph.

4.1 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 3 illustrates this process operating on the graph from Figure 2. 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 comparing Figures 2 and 3 there are several interesting observations. Firstly, while the number of isomorphic hits is the same with VORDER, those hits that do occur are at levels higher in the computation tree. More importantly, 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. However, a detailed analysis of Figure 2 reveals that this is not true of all the isomorphic hits that occur there; for example, the triangle produced from G_{10} in Figure 2 is not immediately isomorphic to that of G_{14}; likewise, the triangle produced from G_9 may or may not be immediately isomorphic to G_{14} — this depends upon exactly which edges were chosen earlier on (and, since Figure 2 represents an arbitrary computation, we have no guarantee of this, unlike with VORDER).

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 §5). 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.

4.2 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.
- **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 §5, the MINSDEG heuristic is generally the better choice of these and, on sparse graphs, it outperforms the VORDER heuristic from §4.1. Figure 4 illustrates MINSDEG operating on the graph from Figure 2. Here, the shape of the computation tree seems quite different from those of Figures 2 and 3. 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 3, 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 4 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

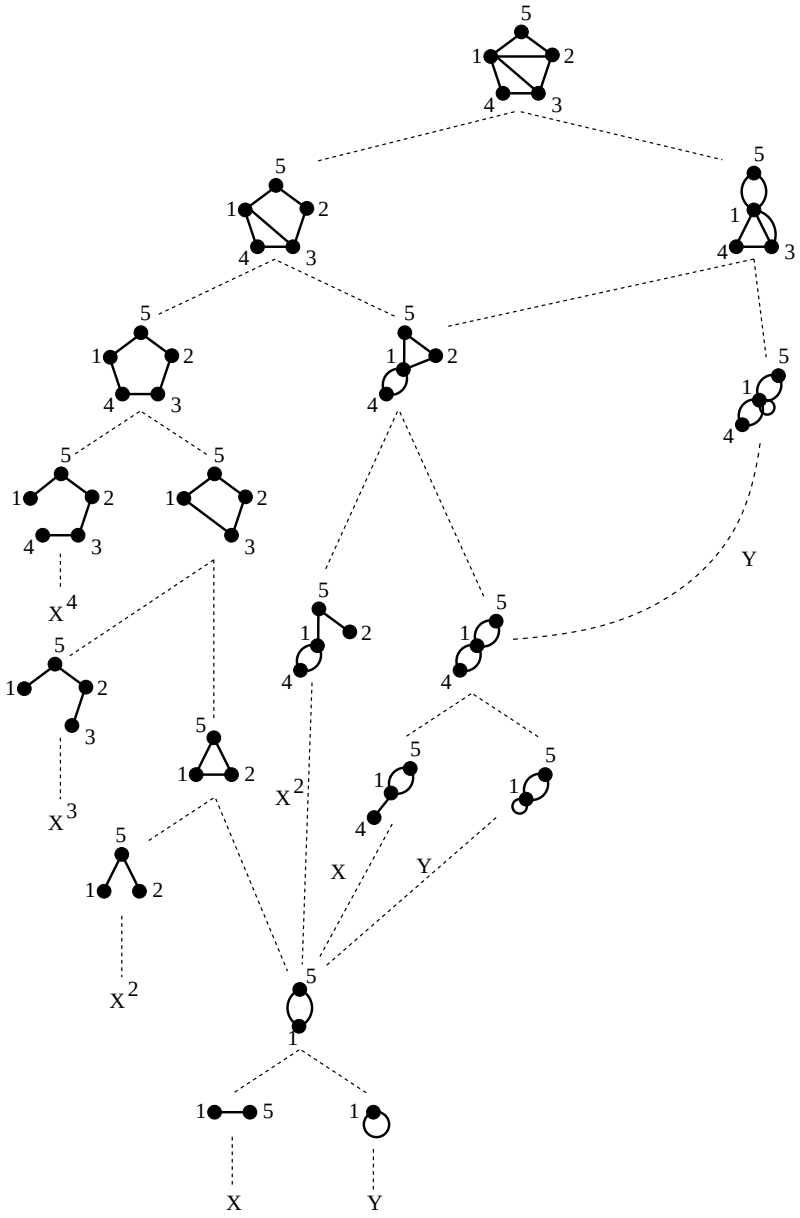


Figure 3: Illustrating the VORDER heuristic on the small graph from Figure 2. The position of each vertex in the order is shown next to it. Thus, the heuristic works aggressively on the vertex labelled 1, since this is lowest in the order.

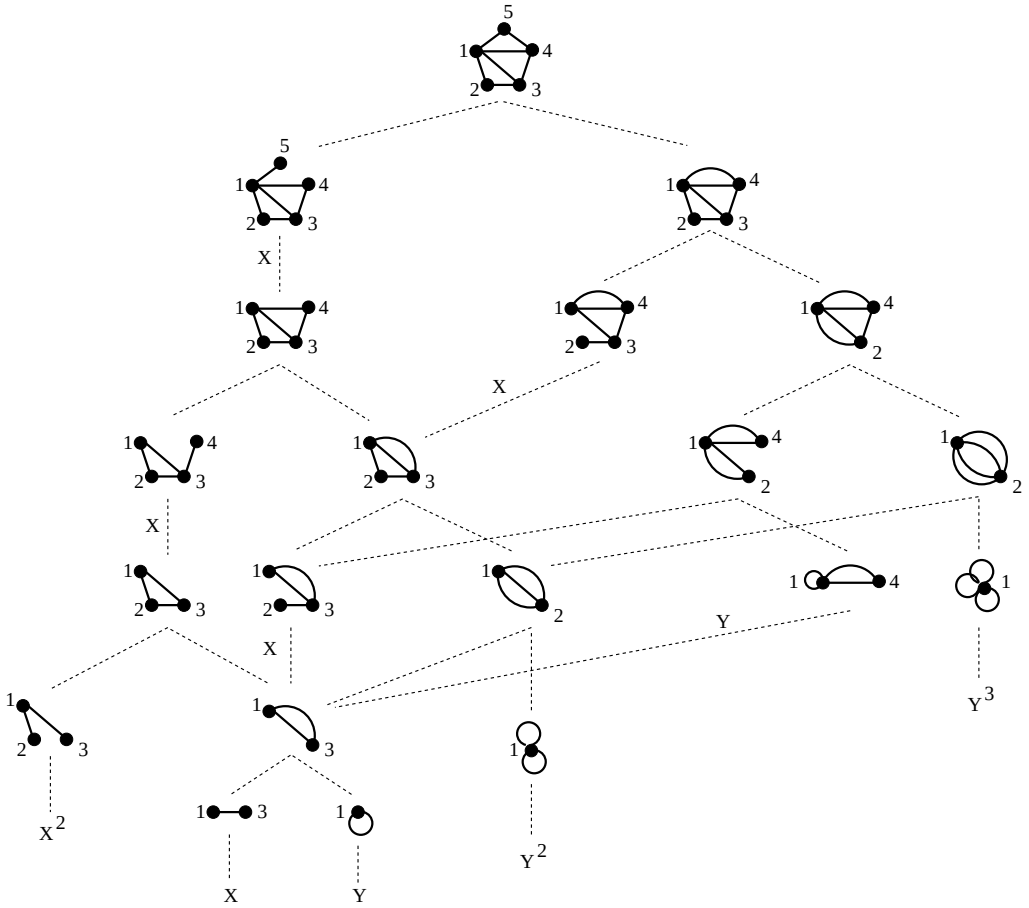


Figure 4: Illustrating the MINSDEG heuristic on the small graph from Figure 2. This heuristic selects edges whose end-point has the lowest degree of any vertex. The vertices of each graph have been labelled to indicate how individual vertices progress through the computation.

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 §5), 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 Experimental Results

In this section, we report on some experimental results comparing the performance of the different heuristics discussed in §4 on a selection of random graphs. We consider (simple) random connected graphs and (simple) random 3-regular and 4-regular graphs. The objective here is to give an indication of the relative performance of these heuristics, and we must emphasise caution about drawing more general conclusions from the results. Nevertheless, our personal experience in using the tool on general graphs is indeed reflected in the results.

5.1 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 50 for each value of $|E|$. 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. To generate *random regular graphs*, we again used `genrang`. The machine used for these experiments was an Intel Pentium IV 3GHz with 1GB of memory, running NetBSD v4.99.9. The executables were compiled using gcc 4.1.3, with optimisation level "-O2" and timing was performed using the `gettimeofday` function, which gives microsecond resolution.

5.2 Experimental Results

Figure 5 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 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 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.

Figures 6 and 7 report 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.

From the plots, it is clear that computing the Tutte polynomial for large graphs quickly becomes intractable. Thus, the improvements offered by MINSDEG may seem somewhat futile (since we only extend the tractable range by a few vertices). However, it is important to consider that any improvements to the range of graphs whose Tutte polynomial can be computed may be of considerable benefit to the users of our program, who are typically searching for a counter-example to some conjecture.

6 Conclusion

In this paper, we have discussed several edge-selection heuristics in the context of a delete/contract-style algorithm for computing Tutte polynomials. Of these, we have identified two which appear to perform particularly well, compared with the others. To support this claim, we reported upon experiments comparing their performance over random connected graphs, random 3-regular and random 4-regular graphs. We believe that these heuristics provide a useful step in the search for efficient algorithms to compute Tutte polynomials.

Finally, the complete implementation of our algorithm for computing Tutte polynomials can be obtained from `http://www.ecs.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. Thanks go to the anonymous reviewers for CATS who provided valuable insights and comments. Thanks also go to the Isaac Newton Institute for funding the Workshop on Combinatorics and Statistical Mechanics in 2008. 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] 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.

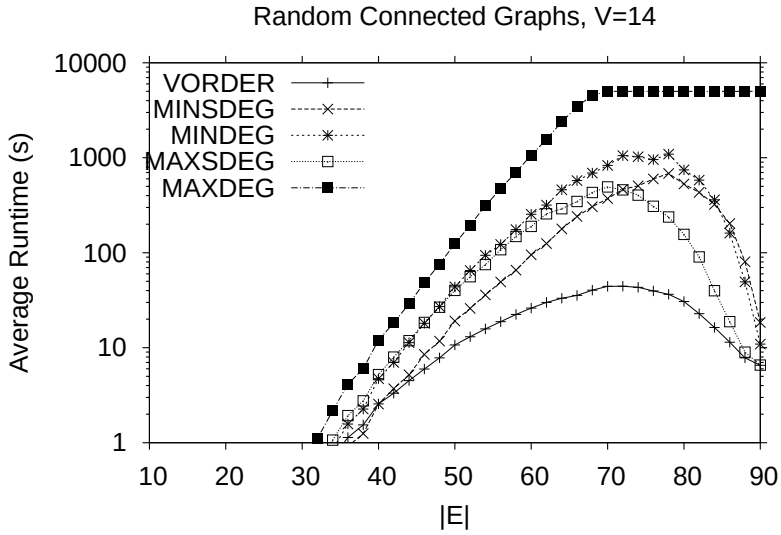


Figure 5: Random Connected Graphs with $|V| = 14$ and varying $|E|$, where each data-point is averaged over 50 graphs. Data illustrating the time taken for the five different heuristics discussed in §4 are shown. For each experiment, the cache size was fixed at 512MB. A timeout of 5000s was used to deal with long running computations, and this explains why the data for MAXDEG flattens out at the top.

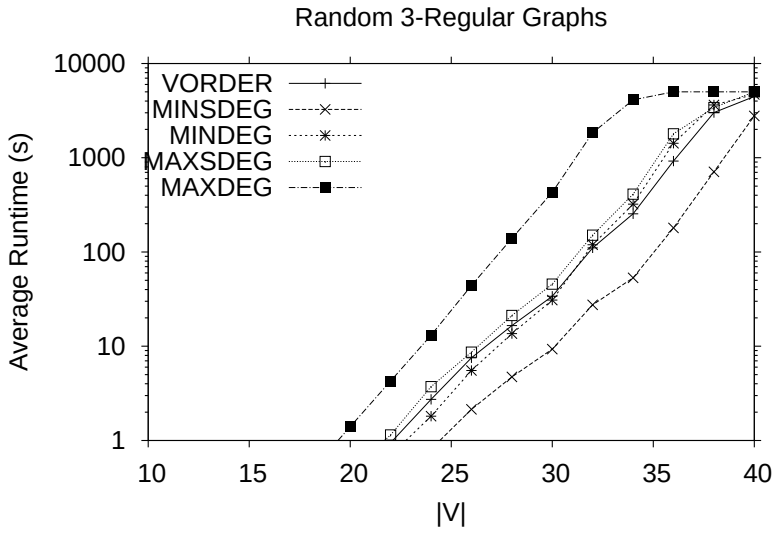


Figure 6: Random 3-Regular Graphs with varying $|V|$, where each data-point is averaged over 50 graphs. Data illustrating the time taken for the five different heuristics discussed in §4 are shown. For each experiment, the cache size was fixed at 512MB. A timeout of 5000s was used to deal with long running computations, and this explains why the data for MAXDEG flattens out at the top.

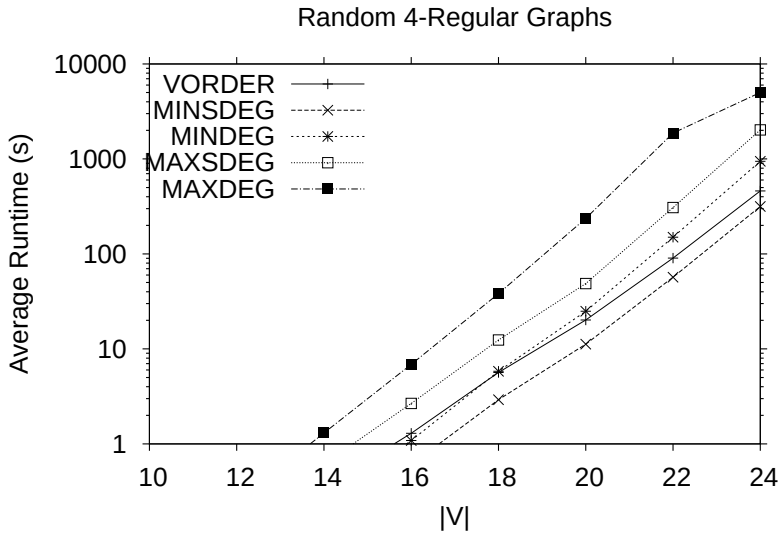


Figure 7: Random 4-Regular Graphs with varying $|V|$, where each data-point is averaged over 50 graphs. Data illustrating the time taken for the five different heuristics discussed in §4 are shown. For each experiment, the cache size was fixed at 512MB. A timeout of 5000s was used to deal with long running computations.

[2] B. Bollobás. *Modern Graph Theory*. Number 184 in Graduate Texts in Mathematics. Springer, New York NY, 1998.

[3] G. Haggard, D. J. Pearce, and G. Royle. Computing Tutte polynomials. Technical Report #NI09024-CSM, Isaac Newton Institute for Mathematical Sciences, Cambridge, UK, 2009.

[4] B. Jackson. An inequality for Tutte polynomials. Technical report, School of Mathematical Sciences, Queen Mary, University of London, 2008.

[5] B. McKay. Nauty users guide (version 1.5). Technical report, Dept. Comp. Sci., Australian National University, 1990.

[6] G. F. Royle. Computing the Tutte polynomial of sparse graphs. Technical Report CORR 88-35, Dept. Comb. & Opt., University of Waterloo, 1988.

[7] 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.

[8] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[9] W. Tutte. A contribution to the theory of chromatic polynomials. *Canadian Journal of Mathematics*, 6:80–81, 1954.