Visualising the Tutte Polynomial Computation

Bennett Thompson and David J. Pearce
Computer Science Group
Victoria University of Wellington, NZ
{thompsbenn,djp}@mcs.vuw.ac.nz

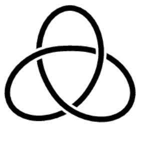
Gary Haggard
Department of Computer Science
Bucknell University
haggard@bucknell.edu

***Abstract*—The Tutte polynomial is an important concept in graph theory which captures many important properties of graphs (e.g. chromatic number, number of spanning trees etc). It also provides a normalised representation that can be used as an equivalence relation on graphs and has applications in diverse areas such micro-biology and physics. A highly efficient algorithm for computing Tutte polynomials has been elsewhere developed by Haggard and Pearce. This relies on various optimisations and heuristics to improve performance; however, understanding the effect of a particular heuristic remains challenging, since the computation trees involved are very large. Therefore, we have constructed a visualisation of the computation in order to study the effect of various heuristics on the algorithms' operation.**

I. Introduction

Tutte polynomials play an important role in graph theory, combinatorics [3], matroid theory, knot theory [1], and experimental physics [13]. For example, the polynomials can be evaluated to find the number of spanning trees in a graph, the number of forests in a graph, the number of connected spanning subgraphs, the number of spanning subgraphs, and the number of acyclic orientations. In addition, Tutte polynomials specialise to chromatic polynomials, flow polynomials, Jones polynomials for alternating links [11], and partition functions of the q-state Potts model of physics.

The Tutte Polynomial can also be used to classify *knots* which has practical applications in areas such as micro-biology. A knot can be thought of as a tangled cord with the ends joined. If the tangled cord is a *trivial knot*, it could be untangled with the ends still fused; however, if the tangled cord is a *nontrivial knot*, a cut is needed to untangle it. The most well-known example of a non-trivial knot is the *trefoil*:



Now, the double helix which constitutes DNA can be visualised as two very long strands that are intertwined and coiled so much as to form a knot. In order for DNA to replicate, however, it must first "untangle" itself and various enzymes are responsible for this. The type of knot involved affects this process, and a better understanding of this would yield additional insight into the replication, transcription, and recombination of DNA [8].

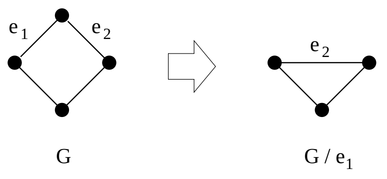
The problem, however, is that there are no practical algorithms available which can compute the Tutte polynomial of a graph of sufficient size to represent a DNA knot [2]. Haggard *et al.* have developed the most efficient algorithm currently available for this [5], based on earlier work on computing Chromatic Polynomials [6], [4]. The algorithm relies on various optimisations and heuristics to obtain good performance. However, the reason that a particular heuristic is effective often remains unclear. A better understanding of the known heuristics would, hopefully, suggest better heuristics and lead to a faster algorithm in practice.

In an effort to address this problem, we have developed a visualisation of the computation tree for a Tutte Polynomial, and we report on this here.

II. Computing Tutte Polynomials

A *graph* is defined as a pair (V, E), where V is the *vertex set* and $E \subseteq V \times V$ the *edge set*. In this paper, we consider only *undirected* graphs, meaning (x, y) is the same as (y, x). A *loop* is an edge (x, x) between the same vertex, whilst a *bridge* is an edge whose removal disconnects two or more vertices (i.e. there is no longer a path between them). The *degree* of a vertex is the number of vertices incident on it.

Two operations are essential to understanding the Tutte polynomial definition. These are: edge deletion, $G - e$; and edge contraction, G/e. The latter involves first deleting e, and then merging its endpoints as follows:



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

$$T(G) = \begin{cases} 1 & E(G) = \emptyset \quad (1) \\ xT(G/e) & e \in E \text{ and } e \text{ is a bridge} \quad (2) \\ yT(G-e) & e \in E \text{ and } e \text{ is a loop} \quad (3) \\ T(G-e) + T(G/e) & e \in E \text{ and } e \text{ is neither a loop nor a bridge} \quad (4) \end{cases}$$

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 [12], and to choose any edge to operate on at each stage. Figure 1 illustrates this recursive procedure applied to a simple graph to give the final

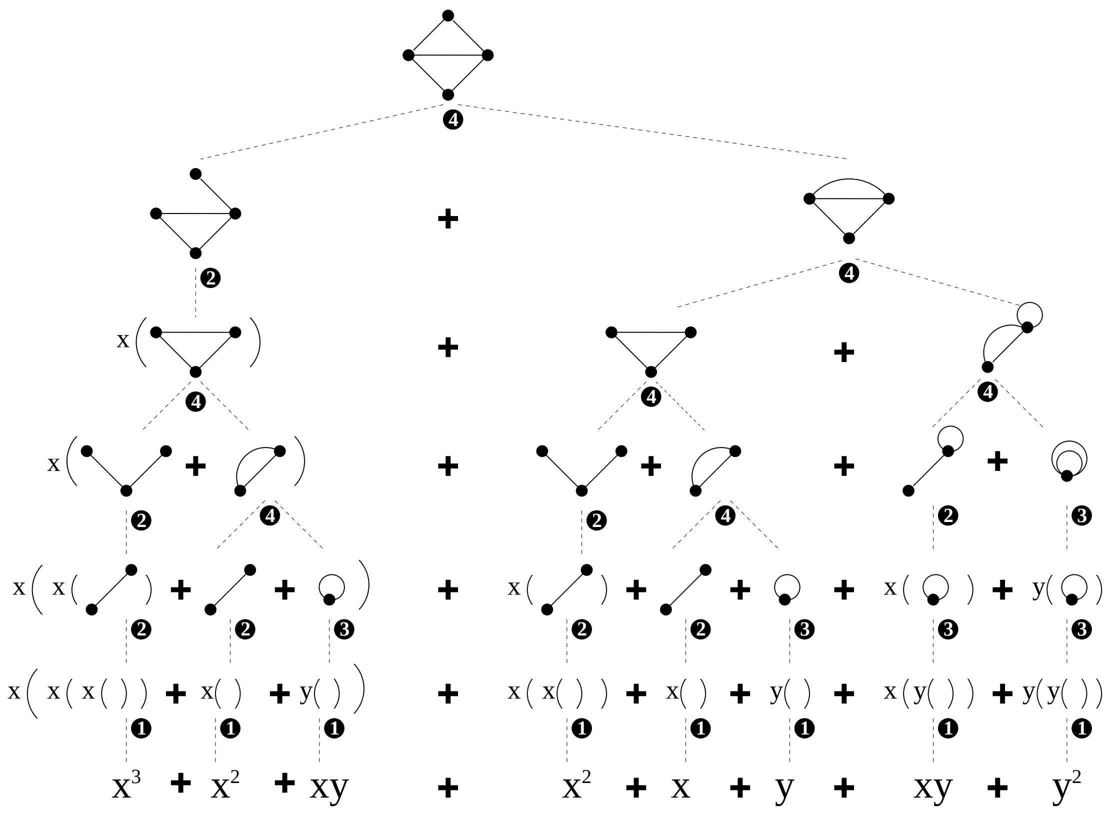


Fig. 1. Illustrating one example computation tree for a simple graph using the rules of Definition 1. The rule from Definition 1 applied at each stage is shown with a white number on a black circle. Observe that we do not draw vertices once they become isolated, since these play no further role in the computation.

polynomial. It is clear from Figure 1 that the computation forms a tree, and we refer to this as the *computation tree*.

The order in which we apply the rules of Definition 1 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 the removal. In Figure 1, for example, a loop arises on the rightmost branch of the computation tree at the third level. This loop is not reduced immediately and, instead, is propagated to the bottom of the tree; removing it immediately, however, reduces the cost of duplicating the graph when the branch forks further down.

A cache of computed polynomials for graphs encountered during the computation is maintained. Thus, when a graph isomorphic to one already resolved is encountered, we can 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. For example, in Figure 1, two of the branches on the third level start from the triangle; thus, after one of the branches has been reduced, the polynomial for the triangle (i.e. $x^2 + x + y$) will be stored in our cache and can be quickly recalled when the second triangle is encountered. To determine graph isomorphism, we employ McKay's *nauty* program [7].

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 subgraph isomorphic to one already seen. For example, selecting either of the multi-edges in the right branch on the second level in Figure 2 results in a graph isomorphic to another (i.e. the triangle) as shown; choosing any of the other edges, however, does not. We have elsewhere developed two simple *edge selection heuristics* which appear to perform well. 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 an arbitrary 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.

Understanding why the edge selection heuristics MINSDEG and VORDER perform so well is, unfortunately, not easy. This is because the computation trees we are interested in typically have hundreds of thousands of nodes, and it is difficult to gauge exactly what effect each heuristic is having. Understanding them better would, hopefully, allow us to design better heuristics. Therefore, we have developed a visualiser in Java for the Tutte polynomial computation.

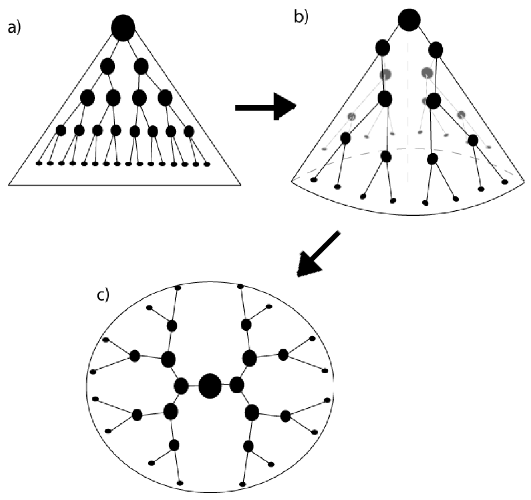


Fig. 2. The evolution of our visualisation of the Tutte computation: **a)** The classic 2D vertical method of visualising a tree; **b)** a cone shaped tree, which is more economic with space and could be "spun"; **c)** the radial tree, which is essentially a top down view of the cone.

III. Visualising the Computation Tree

The first stage in constructing our visualiser was to determine a suitable view method for the computation tree. Considering the size of computation trees we are interested in, it is clear that the whole tree could not be viewable at once. The standard vertical layout of a tree (e.g. Figure 1) is very popular since humans have a tendency to look at the top of a page or screen and scan downward. However, this approach is not particularly economic with space, since it produces large unused areas; furthermore, it results in the the view at the bottom of the vertical layout being rather cramped. We also considered a "cone" approach, where the vertical layout is augmented by having the tree wrapped around a cone. This helps cope with the problem of having a lot of space at the top and very little room at the bottom. In the end, however, we settled upon a radial layout of the computation tree. This is essentially a top down view of the cone which maximises the use of screen real estate. The radial method is simpler to implement than the cone method, although in theory they could be used in conjunction. Figure 2 illustrates the evolution of our visualisation to its current state.

There remains a considerable amount of wasted space with the radial display as seen in Figure 2. To resolve this, we split the radial view into an arrangement of concentric circles divided into *wedges*. We call this the *wedge display*, as illustrated in Figure 3. This provides an uncluttered view of the computation tree, with the flush proximity of nodes allowing node characteristics to be effectively summarised by colour.

Figure 4 illustrates two computation trees for the same starting graph, computed using the two different heuristics. It is immediately apparent from this that the effect of the heuristics can be significant. Figure 5 shows the visualiser's *application view*; this allows the user to manipulate his/her view of the computation tree through zooming, shifting and other effects. The *target viewport* provides the main view window for the computation tree. Within this view the user

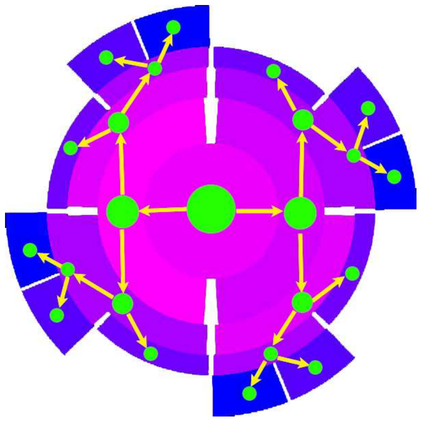


Fig. 3. A view of how the nodes in the computation tree are arranged into the wedge display. The key here is that each node is represented by a *wedge* of colour, rather than a small circle, making better use of space.

can click and drag the view, zoom in/out and select a node, amongst other things. The *macro view* shows an outline view of the computation tree, which helps the user navigate the computation tree; a box in the macro view indicates the size and location of the target viewport. The user can also reposition the target viewport by clicking on the macro view directly. The *node view* shows the graph at a particular node in the computation tree, which is selected by clicking on the target viewport. Figure 6 demonstrates another view produced by our visualiser that can provide some useful insight. This shows the distribution of matches in the cache; recall that the cache is used to store computed polynomials for intermediate graphs seen during the computation, so that they can be recalled when that intermediate graph is encountered again.

Finally, we are unaware of any other work on visualising the Tutte polynomial computation. However, similar work exists on visualising the computation tree of a SAT solver [10], [9].

References

[1] C. Adams. *The Knot Book.* W. H. Freeman and Company, 1994.

[2] Bollobas and Riordan. A tutte polynomial for coloured graphs. In *Combinatorics, Probability and Computing*, volume 8. Cambridge University Press, 1999.

[3] T. Brylawski and J. Oxley. The tutte poynomial and its applications. In *Encyclopedia Math. Appl.*, pages 123–225. Cambridge Univ. Press, 1992.

[4] G. Haggard and T. Mathies. The computation of chromatic polynomials. *Discrete Math*, 199:227–231, 1999.

[5] G. Haggard, D. J. Pearce, and G. Royle. Computing tutte polynomials. Technical report, 2007.

[6] 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.

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

[8] K. Murasugi. *Knot Theory and Its Applications*. Birkhäuser, 1996.

[9] C. Sinz. Visualizing SAT instances and runs of the DPLL algorithm. *Journal of Automated Reasoning*, 39(2):219–243, 2007.

[10] C. Sinz and M. Dieringer. DPvis–A tool to visualize the structure of SAT instances. In *Proc. SAT*, volume 8, pages 257–268, 2005.

[11] M. Thistlewaite. A spanning tree expansion of the jones polynomial. *Topology*, 26:297–309, 1987.

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

[13] D. J. A. Welsh and C. Merino. The Potts model and the Tutte polynomial. *Journal of Mathematical Physics*, 41(3):1127–1152, 2000.

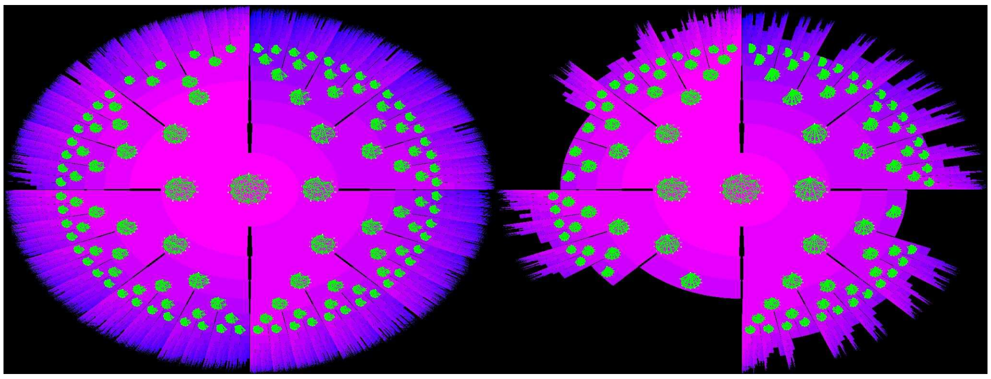
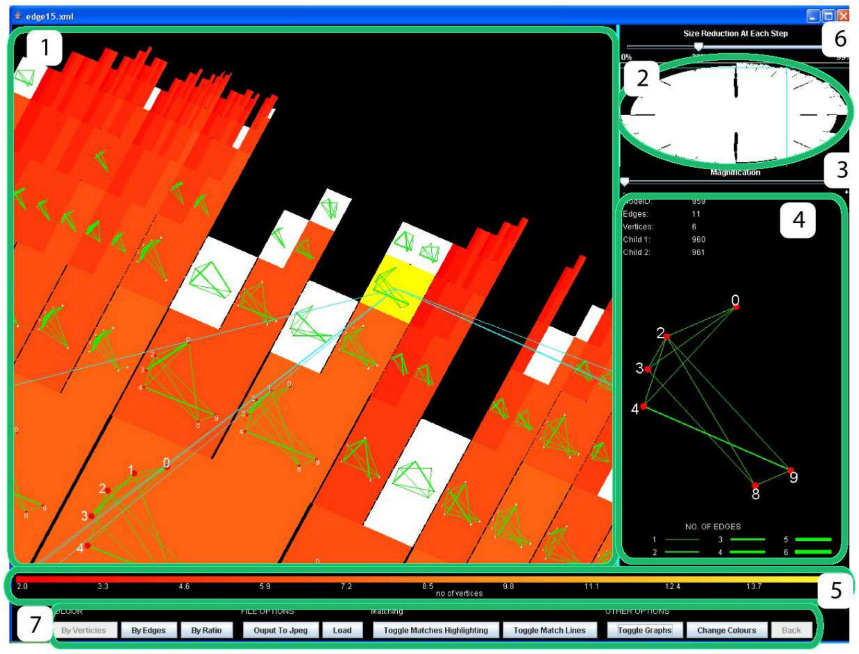


Fig. 4. Two different trees illustrating the Tutte polynomial computation of the same graph. Here, the VORDER heuristic (right) produces a computation tree with 325K steps, thus outperforming the MINSDEG heuristic (left) whose tree has 806K steps. The intermediate graphs of the Tutte computation are also shown when possible, but not displayed once their size is below a certain threshold.



```
NodeID:     959
Edges:      11
Vertices:   6
Child 1:    960
Child 2:    961
```

Fig. 5. The application view. The parts labelled refer to the following: **1.** Target View, **2.** Macro View, **3.** Magnification Slider, **4.** Node View, **5.** Colour Gradient Key, **6.** Squashing Slider, **7.** Function Panel.

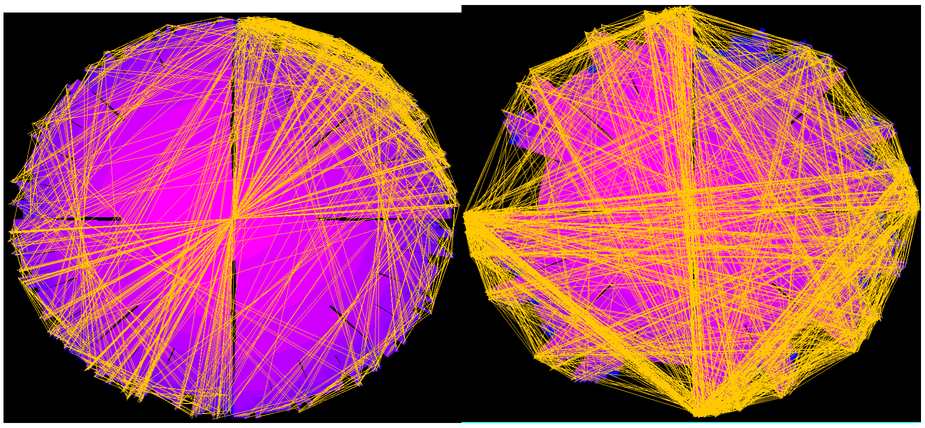


Fig. 6. Two different trees illustrating the Tutte polynomial computation of the same graph. Again, that for the MINSDEG heuristic is shown on the left and that for the VORDER heuristic on the right. Each line shown on the two diagrams connects the point when an intermediate graph is first encountered and stored in the cache, with a later point where that graph is recalled and used.