Heuristics and Identities for Computing the Tutte polynomial

by

Alan Wong

Dissertation Submitted in Partial Fulfillment
of the Requirements for the Degree of
Bachelor of Science (Honours)

in the
Department of Mathematics
Faculty of Science

© Alan Wong 2015
SIMON FRASER UNIVERSITY
Fall 2015

All rights reserved.
However, in accordance with the *Copyright Act of Canada*, this work may be reproduced without authorization under the conditions for "Fair Dealing." Therefore, limited reproduction of this work for the purposes of private study, research, criticism, review and news reporting is likely to be in accordance with the law, particularly if cited appropriately.

Approval

Name: Alan Wong

Degree: Bachelor of Science (Honours) (Mathematics)

Title: *Heuristics and Identities for Computing the Tutte polynomial*

Examining Committee: **Dr. Marni Mishna** (chair)
Professor

Dr. Michael Monagan
Supervisor
Professor

Date Defended: 14 Dec 2015

Abstract

The Tutte polynomial of a graph is a generalization of famous graph polynomials such as the chromatic and flow polynomials. One of the popular methods of computing it is implementing Tutte's deletion-contraction recurrence. Currently, the Tutte polynomial can only be computed for small graphs within a reasonable amount of time. Since this problem is NP-hard, an efficient universal algorithm is not likely to exist, but heuristics can improve the speed of the deletion-contraction algorithm for graphs that satisfy certain properties. We have significantly reduced the computation time of the Tutte polynomial for sparse graphs, by storing intermediate results and developing an edge selection technique that increases the number of identical graphs that appear in the computation tree. We provide benchmarks comparing this technique to other recent deletion-contraction implementations as well as other algorithms. This research should give further understanding into the types of graphs which are more difficult for implementations based on deletion-contraction, particularly denser graphs with large girth. Additionally, we have found and experimented with identities for splitting up the computation of the Tutte polynomial when a graph has a small separating set, and we have also found recurrences for some families of graphs.

Keywords: Tutte polynomials, NP-hard, deletion-contraction, recurrence relations

Acknowledgements

I would like to thank:

Michael Monagan, for all the support and guidance during my research terms.

Marni Mishna, for teaching the Honours project course, and helpful comments on this thesis.

The SFU Office of the Vice-President, Research for funding towards my undergraduate research semester in Summer 2013.

Table of Contents

Approval ii

Abstract iii

Acknowledgements iv

Table of Contents v

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Preliminary Definitions 3

1.2 Sparse Graphs 6

2 Edge Selection Heuristics 7

2.1 Existing Heuristics 8

2.1.1 Degree-based 8

2.1.2 Vertex ordering 8

2.1.3 VORDER-push 9

2.1.4 Possible vertex orderings 9

2.1.5 SHARC vertex ordering 9

2.2 ModSHARC - An improvement to SHARC 10

2.2.1 Vertex Scores 10

2.2.2 Visualization of Scores 11

2.2.3 Algorithm 12

2.3 Benchmarks 13

2.4 Comparison with an evaluation/interpolation approach 13

3 Splitting Formulas 15

3.1 Formulas 15

3.2 Experiments 16

3.2.1 2-separations 16
3.2.2 3-separations 17
3.3 Alternative Proof for 2-separation formula 18

4 Recurrences and Explicit Formulas for Restricted Families 23
4.1 Ladder Graph 23
4.2 Prism Graph 25
4.3 Double Ladder Graph 28

5 Conclusion 30
5.1 Summary of Contributions 30
5.2 Future Work 30

Bibliography 31

Appendix A Code 33

List of Tables

Table 2.1 Timings for biconnected random graphs 13
Table 2.2 Timings for K_n 14

List of Figures

Figure 1.1 Deletion-Contraction example 5

Figure 2.1 Deletion-contraction computation trees 7
Figure 2.2 MINSDEG/MAXSDEG heuristics 8
Figure 2.3 VORDER-pull heuristic. 8
Figure 2.4 VORDER-push heuristic 9
Figure 2.5 SHARC ordering 10
Figure 2.6 Comparison of scores and computation time. 12

Figure 3.1 Splitting a graph G by its 2-separation. 15
Figure 3.2 Timings for splitting by 2-separation 17
Figure 3.3 Timings for 3-separation 18
Figure 3.4 Constructing G_1 and G_2 19
Figure 3.5 4-separation matrix 21

Figure 4.1 Prism graph 25

Chapter 1

Introduction

The Tutte polynomial of a graph is a bivariate polynomial that contains many connectivity and combinatorial properties. It has wide range of applications in, for example, knot theory, theoretical computer science, statistical physics, and chemistry. Due to this, computing Tutte polynomials of graphs has long been of interest. Current algorithms that compute the Tutte polynomial can only do so for small graphs within a reasonable amount of time, 144 vertices for planar lattice graphs [16], around 50 vertices for random sparse cubic graphs [12], and up to 22 vertices for arbitrary graphs [4]. We are interested in finding faster methods to compute the Tutte polynomial, particularly for sparse graphs.

To illustrate the wealth of information that the Tutte polynomial contains, we provide a non-exhaustive list of graph invariants that the Tutte polynomial contains, which includes:

- the chromatic polynomial, which gives the number of proper λ-colourings that G has, for any positive integer λ. In particular, the chromatic number of G is the least positive integer which is not a root of the chromatic polynomial;
- the reliability polynomial, which gives the probability that G stays connected if each edge is deleted independently with some probability $0 \leq p \leq 1$;
- the flow polynomial, which gives the number of nowhere-zero flows of G;
- the number of spanning subgraphs, etc.

Having the Tutte polynomial of G allows us to obtain all the above information simply by evaluating at points and lines on the xy-plane (the two variables of the Tutte polynomial is canonically chosen to be x, y). However, computing the Tutte polynomial itself is extremely computationally expensive, as the problem is #P-hard, so it is at least as difficult as NP-complete problems, and perhaps even more so. The fact that computing the Tutte polynomial is NP-hard follows from that, if we are given the Tutte polynomial of G then we can efficiently decide whether G is 3-colourable (by checking whether 3 is a root of its chromatic polynomial), which is a well known NP-hard problem. While some NP-hard

problems exhibit the property that only special instances are very difficult, this is not the case for the Tutte polynomial, as it is NP-hard to calculate even evaluations of the Tutte polynomial at almost the entire region of the xy-plane [10].

The Tutte polynomial can be equivalently defined in several different ways, which gives rise to several different general methods of computing it. We go into further detail about these definitions in the next section. A recursive algorithm based on the deletion-contraction definition is the most popular method, and the main focus of this thesis. The basic implementation is shown by [16] to require time proportional to the number of edges multiplied by the number of spanning trees. Both of these quantities are lower for sparse graphs compared to dense graphs, and so deletion-contraction performs much better on sparse graphs. Another method, by Björklund et al. [4], computes the *multivariate Tutte polynomial* [17] through an evaluation/interpolation scheme, then evaluates the multivariate version to recover the Tutte polynomial. This method computes the Tutte polynomial in vertex-exponential time, however there is an exponential memory requirement that limits the number of vertices to about 25 for practical purposes. The authors presented an experiment which shows that it outperforms the deletion-contraction implementation of Haggard, Pearce, and Royle [8] on some random dense graphs (complements of 4-regular graphs were used) between 14 to 18 vertices.

Recent work has been made in improving upon the basic deletion-contraction algorithm by identifying isomorphic subgraphs in the computation tree and reusing those intermediate results. Graph isomorphism is a problem for which we do not know whether a polynomial time algorithm exists, in November 2014 Babai [2] presents an algorithm in quasipolynomial time. While a fast practical algorithm is available [11], graph isomorphism testing is still expensive for identifying isomorphic graphs in the computation tree, as this needs to be done at every step. Sekine, Imai, and Tani [16] restricted to 2-isomorphism with respect to a fixed edge ordering, and were able to compute the Tutte polynomial of a 14 vertex graph (with up to 91 edges) within an hour on a typical workstation (by 1995 standards), and the planar 12x12 grid graph with the same resources. In 2010, Haggard, Pearce, and Royle [8] used McKay's *nauty* program [11] for isomorphism testing, and presented experiments on several edge selection heuristics, which increased the number of isomorphic graphs seen in the computation tree, particularly when the vertices are given an ordering (VORDER) and edges are chosen based on the order of its incident vertices. In 2012, Monagan [12] presents two heuristics, a variation of VORDER, and an actual vertex ordering called the **Sh**ort **Arc** (SHARC) ordering. He found that the combination of these two heuristics performs very well on sparse graphs, and eliminated the need for isomorphism testing. His Maple code computed the Tutte polynomial of the truncated isocahedron's dual graph in 9 minutes on a single desktop, compared to a previous result [8] of one week on 150 computers. In Chapter 2 we give an overview of the heuristics and describe a new variation of the SHARC ordering that we found to be an improvement, and present benchmarks that support this.

Another technique for reducing the computation tree is to apply identities which allow us to split the graph into k-connected components (it has a separating set of size at most $k-1$), and compute the Tutte polynomial of each part separately. While theoretical advancements have been made in the area, the implementations of the algorithm mentioned above only test for biconnected components. By splitting up the graph, we can take advantage of parallelization as each part of the computation is mostly independent. Chapter 3 outlines our method of finding these formulas, and some experiments we conducted using the formula for graphs which can be split up this way.

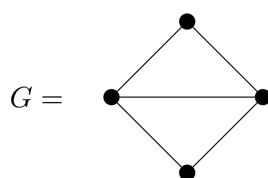
Our final approach is to explore certain simple graph classes, since the Tutte polynomials of some classes of graphs satisfy a linear recurrence and thus an explicit formula can be found. Biggs et al. [3] have dubbed these classes as "recursive families of graphs". A list of some known recurrences and formulas can be found in [19]. We construct recurrences for two recursive families for which recurrences has yet to be proved in Chapter 4.

1.1 Preliminary Definitions

In this thesis, we assume that all graphs are undirected, but may contain multi-edges (more than one edge with the same incident vertices) and loops (an edge from a vertex to itself). We use the notation and definitions of the text *Introduction to Graph Theory* by West [20]. We also assume graphs are connected unless stated otherwise, since the Tutte polynomial is multiplicative with respect to the components of the graph. We denote the Tutte polynomial of a graph G as $T(G, x, y)$, however to ease notation we use $T(G)$ when there is no ambiguity over the variables involved.

Previously we mentioned the information that the Tutte polynomial captures, we now show the precise relations. Let n be the number of vertices and m be the number of edges in G.

Chromatic polynomial	$\chi_G(\lambda) = (-1)^{(n-1)}\lambda T(G, 1-\lambda, 0)$
Reliability polynomial	$R_G(p) = (1-p)^{(n-1)}p^{(m-n+1)}T(G, 1, p^{-1})$
Flow polynomial	$C_G(u) = (-1)^{(m-n+1)}T(G, 0, 1-u)$
Number of spanning subgraphs	$T(G, 1, 2)$

$G =$ 

Tutte polynomial: $T(G, x, y) = x^3 + 2x^2 + 2xy + y^2 + x + y$
Chromatic polynomial: $\chi_G(\lambda) = \lambda(\lambda-1)(\lambda-2)^2$
Reliability polynomial: $R_G(p) = (4p^2 + 3p + 1)(1-p)^3$
Flow polynomial: $C_G(u) = u^2 - 3u + 2$
Number of spanning subgraphs: 14

We begin with Tutte's original definition for the Tutte polynomial, which at the time he referred to as the *dichromate*. This definition relies on the concept of counting internally/externally active edges of a spanning tree. We will not define this concept here, but

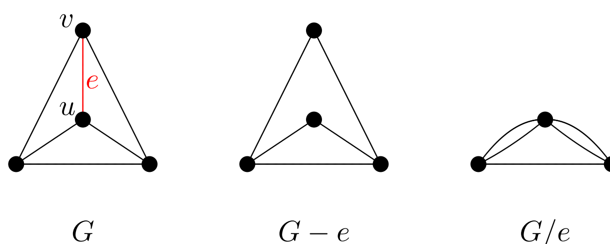
we note these quantities can be computed in polynomial time. From this definition, an immediate consequence is that $T(G,1,1)$ counts the number of spanning trees, and that the Tutte polynomial can be computed in time proportional to a polynomial factor of the number of the spanning trees.

Definition (Tutte [18])**.** *Let G be a graph, and let S be the set of all spanning trees of G. For a spanning tree T, let $r(T)$ and $s(T)$ be the number of edges of T which are internally and externally active, respectively. Then*

$$T(G,x,y) = \sum_{T \in S} x^{r(T)} y^{s(T)}$$

The definition of the Tutte polynomial that we will mostly use is a recurrence based on deletion-contraction. We first define edge deletion and contraction below.

Definition. *Let G be a graph, and $e = uv \in E(G)$ an edge incident to vertices u, v. Then $G - e$ is defined to be the graph obtained by deleting e from G. G/e is defined to be the graph obtained by contracting e (removal of e then joining u, v into a single vertex).*



As demonstrated, multi-edges can arise from edge contraction of a simple graph, and loops arise from contracting a multiple edge. Graphs which are obtained from G through a sequence of edge deletions/contractions are called **minors** of G. We are now ready to give the definition of the Tutte polynomial in terms of deletion-contraction.

Definition (Tutte [18])**.** *Let G be a connected undirected graph. The Tutte polynomial $T(G,x,y)$ is the unique bivariate polynomial which satisfies*

$$T(G) = \begin{cases} 1 & \text{if } |E(G)| = 0, \\ x\,T(G/e) & \text{if } e \text{ is a cut-edge in } G, \\ y\,T(G-e) & \text{if } e \text{ is a loop in } G \\ T(G-e) + T(G/e) & \text{if the edge } e \text{ is neither a loop nor a cut-edge in } G. \end{cases}$$

A cut-edge e in G is defined as an edge such that $G - e$ is not connected. Note that this recurrence is well-defined, choosing any sequence of edges to apply the recurrence gives the

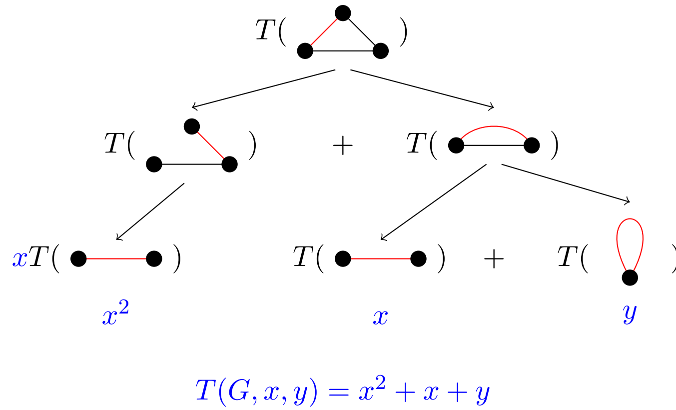
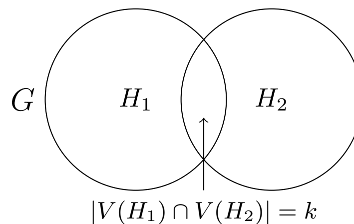


Figure 1.1: Example of applying the deletion-contraction recurrence to compute the Tutte polynomial of the triangle graph.

same result. Also note that every intermediate graph in the comptuation tree is a minor. Figure 1.1 shows an example of using this recurrence to compute the Tutte polynomial.

Certain graphs may be structured in a way that it is composed of several parts that are, in a sense, loosely connected with each other. One area where these graphs may arise from are networks which have to cross difficult geological features (such as rivers, mountain ranges, etc.) and few crossing links are made due to the cost required. For these graphs, there are formulas that allow us to split and recover the Tutte polynomial of the original graph, which is discussed in detail in Chapter 3. This notion of having "loosely connected parts" can be formally defined as a k-separation below.

Definition. *A k-separation of a graph G are subgraphs (H_1, H_2) which satisfy: the union of the subgraphs is G, there are no edges between the subgraphs, and the subgraphs share exactly k vertices. A proper k-separation is one such that the subgraphs are non-empty.*



A k-connected graph is a graph which does not have a proper $(k-1)$-separation. The 2-connected (or biconnected) components of a graph are eloquently named as "blocks". Tutte proved in his original paper that the Tutte polynomial is multiplicative over blocks.

Theorem (Tutte [18])**.** *Let G be a graph with m blocks $B_1, B_2, \ldots, B_m$. Then*

$$T(G, x, y) = \prod_{i=1}^{m} T(B_i, x, y). \tag{1.1}$$

This theorem allows us to split a graph into its blocks, compute the Tutte polynomial of each block, and simply multiply them together to get the Tutte polynomial of the entire graph. For this reason, we will generally assume that the graphs we are working with are 2-connected to avoid this trivial reduction.

1.2 Sparse Graphs

In this thesis, we will mostly be considering sparse graphs. Sparse graphs have relatively few edges (as opposed to dense graphs, which have close to the maximum). There is no precise universally accepted definition of a sparse graph, but for this thesis we will work with graphs with average vertex degree around 3 and 4. Sparse graphs are interesting since they have many applications, especially for communication networks. Building links is costly, so the number of links need to be kept low while still maintaining a reliable and efficient network. For example, graphs that represent the famous ARPANET [9] throughout its evolution (images can be found at `http://som.csudh.edu/cis/lpress/history/arpamaps/`) are sparse, as node generally has at most 3 links.

In addition, many theoretically interesting graphs are sparse. Most of the well-known graphs in the literature have relatively few edges, including the elusive snarks, which are 3-regular (all vertices have degree 3). Studying these classes and computing their Tutte polynomials may lead to intriguing experimental results. A concrete example of using the Tutte polynomial in this way can be found in [8], where the authors find a counterexample to a conjecture about flow polynomials by computing the Tutte polynomials of 3-regular graphs.

Chapter 2

Edge Selection Heuristics

The minors that result from deletion-contraction depends on the choice of the edges used to apply the recurrence, the aim is to develop heuristics that would end up in getting as many isomorphic minors high up in the computation tree as possible, thereby increasing the usage of cached results. The example in Figure 2.1 shows two possible sequences of edge choices, one of which results in two isomorphic minors appearing in the second level, whereas the other does not.

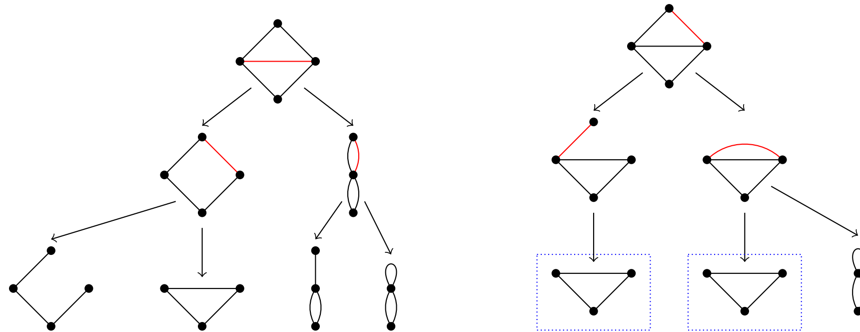


Figure 2.1: Two different deletion-contraction computation trees for the graph at the top. The red edges show the selected edge that is used to apply the recurrence in the next step, the blue boxes highlight the isomorphic minors that appear.

We will describe the various heuristics that have been experimented on, and we highlight two main persisting ideas in these heuristics: working in a **sparse** area of the graph first and maintaining **locality** in the edge selections. By starting in a sparse area, we delay operating on the complex parts of the graph until much lower in the computation tree. However, this is not always possible, if the graph is regular (all vertices have the same degree). By keeping the selections local, we not only increase the number of isomorphic graphs, but the number of identical graphs.

2.1 Existing Heuristics

2.1.1 Degree-based

Edge selection heuristics where the edges were chosen solely based on the degrees of its incident vertices were investigated in [15]. Their idea was to test a group of heuristics which chose edges that were incident to vertices of extremal degree. From their data (Figure 5 of [15]), they found that choosing an edge incident to a vertex of minimum degree (MINSDEG) performed the best (relatively) for sparse graphs, and choosing an incident edge to a maximum degree vertex (MAXSDEG) was the best for dense graphs. See Figure 2.2 for an example. The observed break-even point seems to be just over 70 edges for random connected 14 vertex graphs. The authors also tested heuristics based on the the sum of the degrees of the incident vertices. This was not found to be better than the previously mentioned schemes.

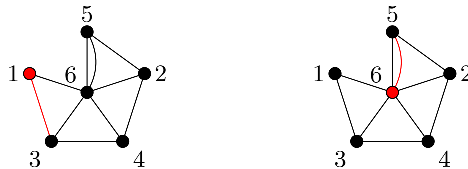


Figure 2.2: MINSDEG heuristic on the left, MAXSDEG on the right. The vertex with the minimum degree is vertex 1 so MINSDEG chooses an edge incident to 1, and the vertex with the maximum degree is vertex 6.

2.1.2 Vertex ordering

The VORDER heuristic [15] (we will refer to this as VORDER-pull to distinguish it from a variation we discuss in the next subsection) chooses edges with respect to some predetermined ordering $v_1, ..., v_n$ of the vertices. The edge is chosen so that it incident to v_1 and the first vertex adjacent to v_1. When an edge is deleted, no adjustment needs to be made to the ordering. When an edge is contracted, the ordering is adjusted by assigning v_1 as the resulting merged vertex (see Figure 2.3).

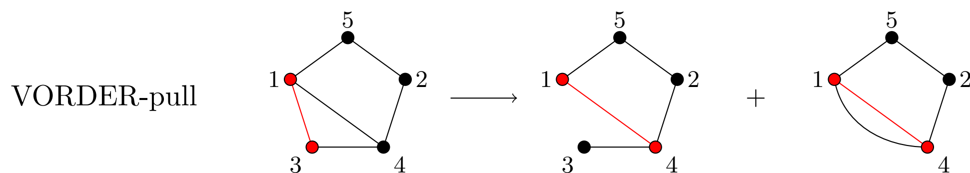


Figure 2.3: VORDER-pull heuristic.

2.1.3 VORDER-push

A variation of the VORDER heuristic was developed in [12], called VORDER-push. The distinction is as follows. Suppose we contract the edge incident to v_1 and v_i. Then the merged vertex is given the order v_i, as opposed to v_1. Additionally, the vertices are canonically re-numbered to $v_1, ..., v_j$, where j is the number of vertices in the minor. This is demonstrated in Figure 2.4. The authors of [15] had considered this variation and mentioned that it "generally does not perform as well", however Monagan [12] found that if VORDER-push is used in tandem with the SHARC vertex ordering (see Subsection 2.1.5), then it works very well on sparse graphs.

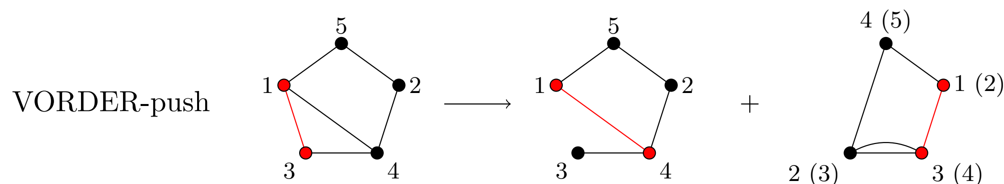


Figure 2.4: VORDER-push heuristic. On the far right, we relabel the vertices after contraction.

2.1.4 Possible vertex orderings

The performance of the algorithm using the VORDER heuristic relies on the initial ordering of the vertices. Random ordering immediately comes to mind, and was tested in [12]. Ordering by ascending vertex degree is another possibility, but this does not maintain locality well. One idea to increase locality is using by starting a breadth-first search (BFS) algorithm at a vertex v, and ordering vertices by the order that the algorithm encounters them. BFS ordering was tested in [12], but in his experiments it was not as good as the SHARC ordering that he later presented.

2.1.5 SHARC vertex ordering

The SHARC (**Sh**ort **Arc**) vertex ordering was introduced by Monagan [12] and extends the idea of breadth-first search. The scheme first picks a vertex v and uses breadth-first search to find a shortest cycle that includes v. The ordering algorithm starts with the vertices of that cycle, and those vertices are added in a set S. Then we iteratively search breadth-first from S to look for the first path that starts and ends at vertices in S, and we append that path to the ordering and add those vertices to S for the next iteration. The stopping condition is when $V(G) \setminus S = \emptyset$. This algorithm repeatedly adds shortest arcs to the ordering, lending its name. We consider an example of applying the SHARC order on the graph G (left side of Figure 2.5). BFS starts at 1 and finds $1 \to 7 \to 8 \to 1$ as the

shortest cycle involving 1, so the ordering starts $1, 7, 8$ and $S = \{1, 7, 8\}$. Next, the path $1 \to 4 \to 6 \to 8$ is found as the shortest path from S back to S (note that the left side path isn't chosen as it requires one more edge to get back to S), then $4, 6$ is appended to both the ordering and S. In the final iteration, $1 \to 3 \to 2 \to 5 \to 8$ is found, and so $\text{SHARC}(G, 1) = [1, 7, 8, 4, 6, 3, 2, 5]$.

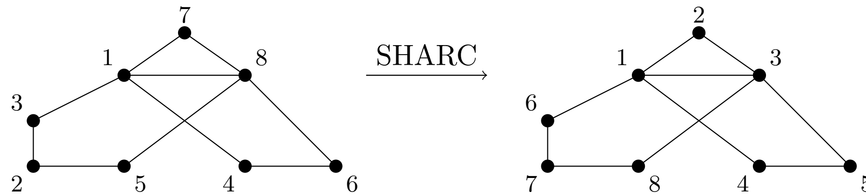


Figure 2.5: Relabelling the vertices of a graph based on a computed SHARC ordering.

Monagan [12] performed experiments with VORDER-push and SHARC on random 3-regular graphs, generalized Petersen graphs $P(n, k)$ for k up to 6, and the dual of the truncated icosahedron. He observed that VORDER-push and SHARC performed much better on these sparse graphs compared to the heuristics that Haggard, Pearce, and Royle were using, and obtained much better timings.

2.2 ModSHARC - An improvement to SHARC

Assembling the above ideas, we aimed to find an ordering which starts in a region of the graph that is relatively sparse, and runs through as short cycles/arcs as we can find.

2.2.1 Vertex Scores

For each vertex v, we assign it scores based on these two factors, and compute the score so that it reflects whether a SHARC ordering that starts at v has these desirable properties. Hence we begin by computing a SHARC ordering starting at $v = v_1$, and suppose it is $[v_1, v_2, ..., v_n]$. The idea behind computing the score is to weight heavily the properties that occur early in the ordering.

We note that, since the score depends on a given SHARC ordering (which is not unique), our vertex scores is not well-defined. In a sense, we are computing scores for the SHARC orderings themselves, and imposing the score on its start vertex for convenience of notation. As we are interested in the behaviour of a SHARC ordering when it reaches that vertex, using the data from another SHARC ordering that starts at that vertex is reasonable for practical purposes.

To capture the relative sparsity of the area, we let the DegreeScore of v be a weighted sum of the vertices in its ordering as follows:

$$\text{DegreeScore}(v) = \sum_{i=1}^{n} \frac{1}{\sqrt{i}} \deg(v_i)$$

For example, going back to the graph in Figure 2.5, DegreeScore(1) is $4+\frac{2}{\sqrt{2}}+\frac{4}{\sqrt{3}}+\frac{2}{\sqrt{4}}+\frac{2}{\sqrt{5}}+\frac{2}{\sqrt{6}}+\frac{2}{\sqrt{7}}+\frac{2}{\sqrt{8}} \approx 11.90$. On the same graph, we compute that the SHARC ordering $[6, 7, 8, 1, 3, 2, 4, 5]$ will achieve the lowest DegreeScore of ≈ 10.64.

CycleScore is determined by the lengths of the initial cycle and the first c cycles/arcs of that SHARC ordering. Suppose the SHARC ordering starting at v starts with a cycle of length a_0 and then arcs of length $a_1, a_2, ..., a_c$, in that order.

$$\text{CycleScore}(v) = a_0 + \sum_{i=1}^{c} \frac{1}{\sqrt{i+1}}(a_i - 2)$$

For example, if we take $c = 2$, and suppose the first iteration of the SHARC ordering starting at v takes a cycle of length 5, then the second to third iterations takes an arc of length 4 and an arc of length 6, respectively. In this case CycleScore(v) would be $5+\frac{2}{\sqrt{2}}+\frac{4}{\sqrt{3}} \approx 8.72$. In our data, we find that $c = 4$ worked the best for random biconnected graphs, and c as large as possible for random regular graphs.

We compute an aggregate TotalScore for v from a linear combination of the normalized DegreeScore and CycleScore.

$$\text{TotalScore}(v) = \alpha_1 \,\text{DegreeScore}(v) + \alpha_2 \,\text{CycleScore}(v)$$

We have found that taking $\alpha_1 = 2\alpha_2$ (as we only use the scores for comparison, only their ratio is relevant) works well.

2.2.2 Visualization of Scores

The scores are constructed so that low scores correspond to low degree vertices and short cycles, in Figure 2.6 we present a visual demonstration of the correlation between the scores and Tutte polynomial computation times using a SHARC order starting at each vertex.

The correlation is not perfect, as we can see that some high scores have been assigned to vertices which turned out to be reasonable candidates for starting the ordering. It is **extremely** important that the worst vertices are identified (for example, the red vertex in the center of the graph on the right of Figure 2.6), since the time difference between starting from the best and worst vertices can be orders of magnitude. On the other hand, if we falsely proclaim a good starting vertex to be bad, that error is much more tolerable. In our example, we can see that most vertices are in the blue/cyan range, so using any of them is acceptable.

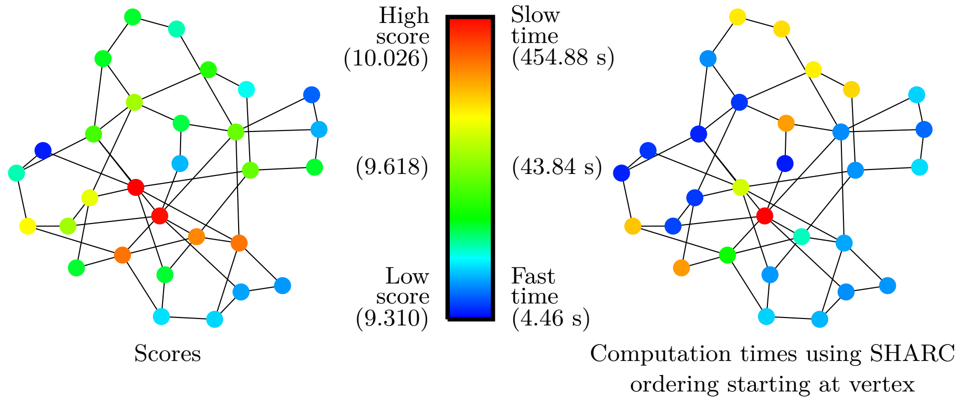


Figure 2.6: Comparison of scores and computation time.

2.2.3 Algorithm

Our implementation is based on the one described in Section 3 of [12], similarly we use an adjacency list representation of the graph as input and cache intermediate results with Maple's `option remember`. One difference is that we do not use any isomorphism test, since the experiments from [12] show that for sparse graphs, almost all isomorphic graphs will be handled by `option remember`.

Given an input graph G, the algorithm first splits the graph into blocks and handles each block separately. While some implementations such as [8] test for blocks at every step, we chose to only do it at the start as this does not seem to offer a significant improvement for sparse graphs. Then, starting from every vertex, the SHARC ordering is executed to collect the scores as outlined above. After this analysis has been done, SHARC order is run for a final time, starting at the vertex with the lowest TotalScore. In the final ordering, if at any step the SHARC algorithm needs to choose between multiple cycle/arcs of the same length, then we use the average scores of the vertices involved to be a tiebreaker. We will call this the **Mod**ified **SHARC** ordering.

The algorithm then relabels the graph based on the computed ModSHARC ordering, canonically from 1 to n. This allows us to simply choose the edge that is incident to vertex 1 and its first neighbor, and we readjust the vertex numbers after applying deletion and edge contraction (using VORDER-push). The recurrence is now ready to be applied. At each step of the recurrence, the algorithm checks if the graph has any multi-edge or loop, and processes those first. Otherwise, an edge is selected (as described above) and the deletion/contraction graphs are constructed by adjusting the relevant entries in the adjacency list, and their Tutte polynomial are recursively computed. If at any point in the recursion, a graph has already been seen, `option remember` will find it and return the cached result.

Our Maple code along with a help file for this algorithm is included in Appendix A.

2.3 Benchmarks

As the ModSHARC ordering specifically looks for differences in vertex degree, to see potential improvement we tested on graphs which had non-uniform degree sequences. The natural choice was sparse random graphs, for which we used Maple's `RandomGraph(n,m)` command to get a random graph on n vertices and m edges. Since we chose m such that the result graph is sparse, we found that many of our generated graphs contained trivial blocks consisting of a single edge (which we immediately split off from the computation due to Equation 1.1). Hence we restricted our testing to biconnected random graphs.

For $24 \leq n \leq 39$, we generated 200 random biconnected graphs with average degree $\frac{10}{3} \approx 3.33$. For the experiments below, we used a server with 2 Intel Xeon E5-2660 8 core CPUs at 2.2/3.0 GHz with 64 GB RAM. Table 2.1 shows the average, median, and maximum time needed to compute the Tutte polynomial using VORDER-push with SHARC and ModSHARC orderings.

		SHARC			Mod-DegreeScore Only			Mod-TotalScore		
n	m	avg	med	max	avg	med	max	avg	med	max
24	40	1.8	1.1	13.1	0.9	0.7	3.1	0.9	0.7	4.5
27	45	6.3	3.9	75.5	2.6	1.8	33.3	2.3	1.7	14.9
30	50	30.3	13.1	494.0	7.9	4.1	98.5	8.5	4.9	85.5
33	55	775.2	47.7	81157.1	23.1	11.2	393.6	27.3	12.7	511.3
36	60				956.3	49.2	144790.1	227.0	53.4	9365.1
39	65							2193.7	182.6	55095.4

Table 2.1: Timings in CPU seconds for random biconnected graphs with n vertices, m edges using VORDER-push and variations of ModSHARC.

We tested with using DegreeScore only, and the TotalScore as described above. Introducing the CycleScore component seems to have a negligible effect until we reach $n = 36$, but does improve the worse case graph in that set by at least 2 orders of magnitude over DegreeScore. The main observation is that we have significant improvement over SHARC as n grows, particularly the gains on SHARC's slowest times. This validates our attempt to avoid the worst case.

2.4 Comparison with an evaluation/interpolation approach

We compared our algorithm with a method developed Björklund et al. [4] that does not use the deletion-contraction recurrence. Instead, the multivariate Tutte polynomial [17] is computed using the Fortuin-Kasteleyn identity [7] that relates the multivariate Tutte polynomial to the q-state Potts model in statistical physics. They presented several variations of their algorithm which computes the Tutte polynomial in vertex-exponential time, and

they have made available an implementation (a C program called `tutte_bhkk` uploaded to GitHub) which takes time and space $\sigma(G)n^{O(1)}$, where $\sigma(G)$ denotes the number of connected induced subgraphs.

We implemented one variation of their algorithm in C++ described in Section 4.1 of [4]. We briefly list the some implementation details, which includes:

- using classical arithmetic polynomial arithmetic over 31-bit primes, using Chinese Remainder Theorem as necessary;
- observing that powers of z past n were not being used, so polynomial multiplication was done modulo z^{n+1};
- caching results that needed reuse (in particular $F(X, 1, i)$ for all $X \subseteq V(G), i$ from 0 to n).
- avoid taking powers of polynomials by using the identity $F(X, q, n) = F(X, q-1, n)F(X, 1, n)$ for $q > 1$

The experiments below use three methods, our Maple code using the ModSHARC ordering, our C++ implementation, and `tutte_bhkk`. Computations in this section were done on a 2.66 GHz i7 desktop with 6 GB RAM. The data we collected for computing the Tutte polynomial of complete graphs K_n is shown in Table 2.2, and we also tested five 3-regular graphs on 16 vertices. The average time taken for Maple was 0.0826s, compared to 15.8s for C++, and 10.9s for `tutte_bhkk`.

n	ModSHARC	C++	`tutte_bhkk`
10	0.323	0.2	0.18
12	1.69	2.91	3.45
14	9.21	21.35	29.8

Table 2.2: Timings in CPU seconds for complete graph K_n

While Table 2.2 shows that our C++ code is faster, we do not claim to have a superior implementation, as our code was developed for experimental purposes and does not handle general cases well. In the above tests, our Maple code was much faster, this was expected for the types of graphs that we tested with, as the authors of [4] said that graphs which are fairly sparse (i.e. 3-regular graphs) and graphs which have many symmetries (i.e. complete graphs) are "amenable to many of the previously existing techniques".

Their main experiment used dense graphs without such symmetries, in particular complements of 4-regular graphs. We generated a random graph with 14 vertices and 63 edges, and computed its Tutte polynomial with the three methods. Our Maple code took 207s, compared to 16.87s for our C++ code, and 19.22s for `tutte_bhkk`.

Chapter 3

Splitting Formulas

Graphs with a small vertex separation allow us to split the graph into minors, compute the Tutte polynomial of the smaller minors, and reassemble the solution. This dramatically reduces the computation tree, and gives a natural parallel algorithm for these types of graphs. For example, consider the graph G in Figure 3.1 containing a proper 2-separation (H, K) and decomposing it into minors that are necessary to compute its Tutte polynomial.

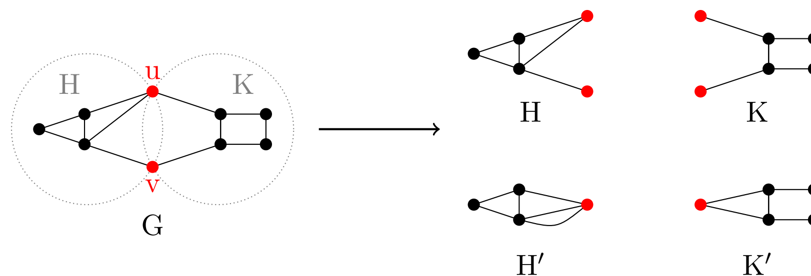


Figure 3.1: Splitting a graph G by its 2-separation.

3.1 Formulas

We used a constructive strategy to find a formula that reassembles the Tutte polynomial in this manner. For more details, we included a proof in Section 3.3, and we verified the formula experimentally for thousands of graphs that contain a 2-separation. We then found that analogous formulas have been discovered by Brylawski [6] in 1971, later formally stated by Oxley and Welsh [14]. Their formulas are stated in terminology of matroids, we will restate it in terms of graphs.

Theorem (Oxley,Welsh [14])**.** *Let G be a graph with a 2-separation (H, K) and separation pair $u, v \in V(G)$ such that H, K are connected. Let H', K' be the graphs obtained from*

H, K, respectively, by identifying u, v.

Define $A_{11} = (x-1)T(H') - T(H)$ and $A_{00} = (y-1)T(H) - T(H')$.

Then

$$T(G) = \frac{1}{xy - x - y}(A_{11}T(K') + A_{00}T(K)).$$

We extended our strategy to find a formula for the case of a 3-separation. For 2-separations, we needed to construct an additional graph for each side. Here we need four additional graphs for each side, one for each possible set of identified vertices. We then discovered that in 1997 Andrzejak [1] proves the splitting formula for 3-sum of matroids, and the matrix he presents is simply a permutation of the one we found, shown in the theorem below.

Theorem (Andrzejak [1]). *Let G be a graph with a 3-separation (H, K) and separating set $\{\mathbf{1}, \mathbf{2}, \mathbf{3}\} \subset V(G)$ such that H, K are connected. Let H_S, where S is a sequence of vertex names, be the graph obtained from H by identifying the vertices in S (and define K_S similarly). For example, K_{12} is the graph obtained from K by identifying 1 and 2.*

Let $C = xy - x - y$, and let

$$\begin{bmatrix} A \\ A_{12} \\ A_{23} \\ A_{13} \\ A_{123} \end{bmatrix} = \frac{1}{C(C-1)} \begin{bmatrix} (1-y)^2 & 1-y & 1-y & 1-y & 2 \\ 1-y & C & 1 & 1 & 1-x \\ 1-y & 1 & C & 1 & 1-x \\ 1-y & 1 & 1 & C & 1-x \\ 2 & 1-x & 1-x & 1-x & (1-x)^2 \end{bmatrix} \begin{bmatrix} T(H) \\ T(H_{12}) \\ T(H_{23}) \\ T(H_{13}) \\ T(H_{123}) \end{bmatrix}$$

Then $T(G) = \begin{bmatrix} A \\ A_{12} \\ A_{23} \\ A_{13} \\ A_{123} \end{bmatrix} \bullet \begin{bmatrix} T(K) \\ T(K_{12}) \\ T(K_{23}) \\ T(K_{13}) \\ T(K_{123}) \end{bmatrix}$.

3.2 Experiments

We've conducted experiments to test the performance gain from using these theorems, by generating random biconnected graphs H, K and joining k of their vertices to construct a graph G with a k-separation.

3.2.1 2-separations

We ran our Tutte polynomial code on G, and then compared the computation time with running the code on the four minors H, H', K, K' then assembling the Tutte polynomial of

G by applying the formula. This was repeated for 48 graphs, Figure 3.2 shows a plot of the data we collected [13] for the 10 worst times of each scheme.

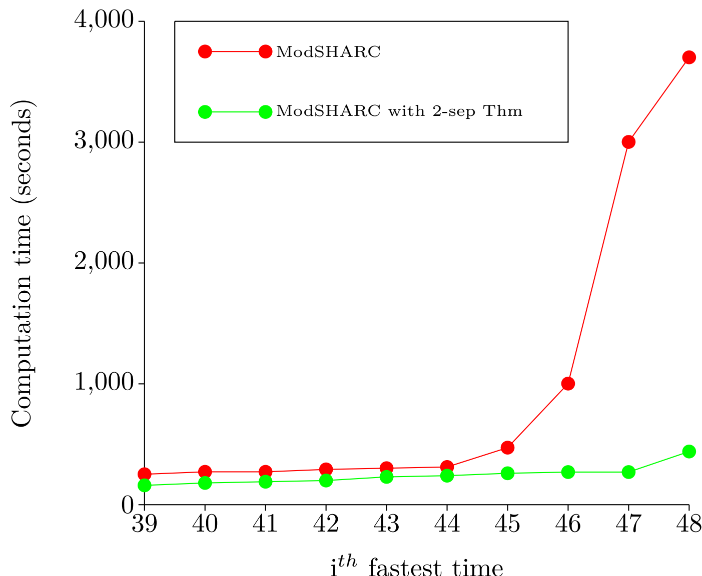


Figure 3.2: Timings (in seconds) of biconnected graphs with a 2-separation (H, K) where H and K both have 32 vertices and 54 edges

We observe that most of the times are very similar except for the very worst ones. This is likely due to the fact that the SHARC ordering is localized, so that in most cases it orders one side of the separation before the other side. However, even in these cases it may still be better to split up the graph as it lends an opportunity for parallelization. Since the bulk of the work is done in the computation of the Tutte polynomials of the four minors (applying the formulas themselves requires only arithmetic of bivariate polynomials), we are likely to get an excellent speedup by processing each minor in parallel.

3.2.2 3-separations

To construct graphs with a 3-separation, we generated two random biconnected graphs H and K, added 3 vertices for the separating set, and added 12 edges (4 for each vertex) from the separation vertices to random vertices of H and K. We are interested in investigating the effect of parallel computation, so we used Maple's `Grid:-Map` command which spawns parallel processes to apply a function to each element of the input list. In this case, the function is our Tutte polynomial procedure using ModSHARC, and the list contains the graphs of the ten minors $H, H_{12}, ..., K_{123}$ as required by the formula. We generated twenty sets of random graphs, and ran the experiments on a server with 2 Intel Xeon E5-2680 10 core CPUs at 3.0/3.6 GHz and 128 GB RAM. The data is shown in Figure 3.3.

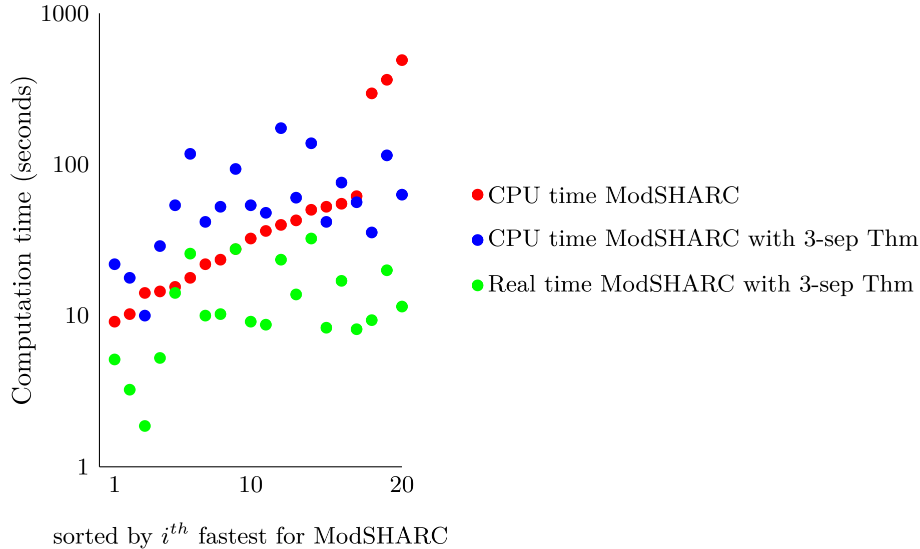


Figure 3.3: Timings (in seconds, logarithmic scale) of biconnected graphs with a 3-separation (H, K) where H and K both have 23 vertices and 46 edges

We observe that the CPU time (summed over the computations on the minors) is often actually higher than simply processing the entire graph, however, looking at the real time (with 10+ cores used) there does seem to be an improvement. Still, only for very few graphs do we see a 10x speedup, and the reason for this seems to be that there is sometimes a large variation in the computation time between the related minors. Although these minors clearly have very similar structure, our algorithm can take much longer to process one over the other (in this dataset, one of the H_{13} graphs took 2s compared to 41s for the related H_{123}). This gives rise to a possibility of further improving the ModSHARC ordering, by studying how and why these similar graphs behave so differently under our heuristic, and whether it is possible to change the slow computation trees to resemble the faster ones.

3.3 Alternative Proof for 2-separation formula

We present an alternative proof to the formula for 2-separation. First recall the theorem conditions: G is a graph with a proper 2-separation (H, K) and separation pair $u, v \in V(G)$ such that H, K are connected. We call H', K' the graphs obtained from H and K, respectively, by identifying the separation pair.

Proof. We claim $T(G)$ is of the form

$$T(G) = p_1(x, y)T(K) + p_2(x, y)T(K'), \tag{3.1}$$

where p_1 and p_2 are bivariate polynomials. Further, we claim that p_1 and p_2 relies on H only (i.e. no dependence on K).

Consider a computation tree of the deletion-contraction recurrence on G using only edges of H, until all edges of H are deleted/contracted. In the computation tree, there are only two minors that remain, K and K'. The computation expresses the Tutte polynomial of G as a sum of bivariate monomials multiplied by $T(K)$ or $T(K')$, and by collecting the $T(K), T(K')$ terms we obtain p_1 and p_2. Since K is assumed to be connected, the structure of K does not affect whether the selected edges in H are loops/cut-edges, so it does not affect p_1 and p_2, establishing our claims.

Construct the graph G_1 by adding an edge from u to v (possibly creating a multiedge), and G_2 by adding two edges from u to v, as shown in Figure 3.4. Now we denote L_1 be the single edge graph on 2 vertices u and v, and L_2 be the double edge graph on u and v. Then G_1 and G_2 has 2-separations (H, L_1) and (H, L_2), respectively.

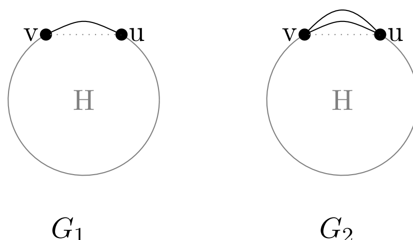


Figure 3.4: Constructing G_1 and G_2

Then let L'_1 be L_1 with u, v identified (loop on a single vertex), and L'_2 similarly (double loop on a single vertex). Since we showed that p_1, p_2 do not depend on K, we have the following equations:

$$T(G_1) = p_1 T(L_1) + p_2 T(L'_1)$$
$$T(G_2) = p_1 T(L_2) + p_2 T(L'_2)$$

To keep our notation consistent with the theorem statement, we let $A_{00} = p_1$ and $A_{11} = p_2$. Then we have

$$T(G_1) = T(L_1)A_{00} + T(L'_1)A_{11}$$
$$T(G_2) = T(L_2)A_{00} + T(L'_2)A_{11}$$

We substitute the known Tutte polynomials of L_1, L'_1, L_2, L'_2 to obtain the following system of linear equations.

$$T(G_1) = xA_{00} + yA_{11} \tag{3.2}$$
$$T(G_2) = (x+y)A_{00} + y^2 A_{11} \tag{3.3}$$

We want to express $T(G_1)$ and $T(G_2)$ in terms of $T(H)$ and $T(H')$, so we apply deletion-contraction on G_1 and G_2 on edge uv.

$$T(G_1) = T(H) + T(H')$$
$$T(G_2) = T(G_1) + yT(H') = T(H) + (y+1)T(H')$$

Substituting into Equation 3.2 and 3.3, we get:

$$T(H) + T(H') = xA_{00} + yA_{11}$$
$$T(H) + (y+1)T(H') = (x+y)A_{00} + y^2 A_{11}$$

Solving the above linear system, we obtain $A_{11} = (xy - x - y)^{-1}((x-1)T(H', x, y) - T(H, x, y))$ and $A_{00} = (xy - x - y)^{-1}((y-1)T(H, x, y) - T(H', x, y))$. Substituting into Equation 3.1 and factoring out $(xy - x - y)^{-1}$ gives the result as stated. □

We have developed a similar result for a 4-separation, using techniques analogous to our proof of the 2-separation case. Note that, the formula for the Tutte polynomial of k-sums of matroids has been solved in general in [5], however the theorem is very complex and requires substantial background in matroid theory. We have not seen the formula for a 4-separation stated explicitly in the literature.

Theorem. *Let G be a graph with a 4-separation (H, K) and separation set $\{1, 2, 3, 4\} \in V(G)$ such that H, K are connected. For $i, j, k, l \in \{1, 2, 3, 4\}$, let H_{ij}, K_{ij} be the graphs obtained from H, K by identifying vertices i and j. Let H_{ijk}, K_{ijk} be the graphs obtained from H, K by identifying vertices i, j, k. Let $H_{ij,kl}, K_{ij,kl}$ be the graphs obtained from H, K by first identifying vertices i, j then identifying k, l. Let H_{1234}, K_{1234} be the graphs obtained from H, K by identifying the vertices $1, 2, 3, 4$.*

Let $X = x - 1, Y = y - 1, C = xy - x - y, D = C^2 - C - 1, Z = C - 1$, let M be the matrix shown in Figure 3.5.

$$
\begin{bmatrix}
Y^3 & -Y^2 & -Y^2 & -Y^2 & -Y^2 & -Y^2 & -Y^2 & 2Y & 2Y & 2Y & 2Y & Y & Y & Y & -6 \\
-Y^2 & YZ & Y & Y & Y & Y & Y & -C & -C & -2 & -2 & -Z & -1 & -1 & 2X \\
-Y^2 & Y & YZ & Y & Y & Y & Y & -C & -2 & -C & -2 & -1 & -Z & -1 & 2X \\
-Y^2 & Y & Y & YZ & Y & Y & Y & -2 & -C & -C & -2 & -1 & -1 & -Z & 2X \\
-Y^2 & Y & Y & Y & YZ & Y & Y & -C & -2 & -2 & -C & -1 & -1 & -Z & 2X \\
-Y^2 & Y & Y & Y & Y & YZ & Y & -2 & -C & -2 & -C & -1 & -Z & -1 & 2X \\
-Y^2 & Y & Y & Y & Y & Y & YZ & -2 & -2 & -C & -C & -Z & -1 & -1 & 2X \\
2Y & -C & -C & -2 & -C & -2 & -2 & \frac{C^2}{Y} & \frac{C+2}{Y} & \frac{C+2}{Y} & \frac{C+2}{Y} & \frac{C}{Y} & \frac{C}{Y} & \frac{C}{Y} & -\frac{X(C+2)}{Y} \\
2Y & -C & -2 & -C & -2 & -C & -2 & \frac{C+2}{Y} & \frac{C^2}{Y} & \frac{C+2}{Y} & \frac{C+2}{Y} & \frac{C}{Y} & \frac{C}{Y} & \frac{C}{Y} & -\frac{X(C+2)}{Y} \\
2Y & -2 & -C & -C & -2 & -2 & -C & \frac{C+2}{Y} & \frac{C+2}{Y} & \frac{C^2}{Y} & \frac{C+2}{Y} & \frac{C}{Y} & \frac{C}{Y} & \frac{C}{Y} & -\frac{X(C+2)}{Y} \\
2Y & -2 & -2 & -2 & -C & -C & -C & \frac{C+2}{Y} & \frac{C+2}{Y} & \frac{C+2}{Y} & \frac{C^2}{Y} & \frac{C}{Y} & \frac{C}{Y} & \frac{C}{Y} & -\frac{X(C+2)}{Y} \\
Y & -Z & -1 & -1 & -1 & -1 & -Z & \frac{C}{Y} & \frac{C}{Y} & \frac{C}{Y} & \frac{C}{Y} & \frac{D}{Y} & \frac{1}{Y} & \frac{1}{Y} & -\frac{XC}{Y} \\
Y & -1 & -Z & -1 & -1 & -Z & -1 & \frac{C}{Y} & \frac{C}{Y} & \frac{C}{Y} & \frac{C}{Y} & \frac{1}{Y} & \frac{D}{Y} & \frac{1}{Y} & -\frac{XC}{Y} \\
Y & -1 & -1 & -Z & -Z & -1 & -1 & \frac{C}{Y} & \frac{C}{Y} & \frac{C}{Y} & \frac{C}{Y} & \frac{1}{Y} & \frac{1}{Y} & \frac{D}{Y} & -\frac{XC}{Y} \\
-6 & 2X & 2X & 2X & 2X & 2X & 2X & -\frac{X(C+2)}{Y} & -\frac{X(C+2)}{Y} & -\frac{X(C+2)}{Y} & -\frac{X(C+2)}{Y} & -\frac{XC}{Y} & -\frac{XC}{Y} & -\frac{XC}{Y} & \frac{X^2(C+2)}{Y}
\end{bmatrix}
$$

Figure 3.5: The matrix M used in the formula for a 4-separation, where $X = x-1, Y = y-1, C = xy-x-y, D = C^2-C-1, Z = C-1$

$$\text{Let } \vec{H} = \begin{bmatrix} T(H,x,y) \\ T(H_{12},x,y) \\ T(H_{13},x,y) \\ T(H_{14},x,y) \\ T(H_{23},x,y) \\ T(H_{24},x,y) \\ T(H_{34},x,y) \\ T(H_{123},x,y) \\ T(H_{124},x,y) \\ T(H_{134},x,y) \\ T(H_{234},x,y) \\ T(H_{12,34},x,y) \\ T(H_{13,24},x,y) \\ T(H_{14,23},x,y) \\ T(H_{1234},x,y) \end{bmatrix}, \vec{K} = \begin{bmatrix} T(K,x,y) \\ T(K_{12},x,y) \\ T(K_{13},x,y) \\ T(K_{14},x,y) \\ T(K_{23},x,y) \\ T(K_{24},x,y) \\ T(K_{34},x,y) \\ T(K_{123},x,y) \\ T(K_{124},x,y) \\ T(K_{134},x,y) \\ T(K_{234},x,y) \\ T(K_{12,34},x,y) \\ T(K_{13,24},x,y) \\ T(K_{14,23},x,y) \\ T(K_{1234},x,y) \end{bmatrix}, \vec{A} = \begin{bmatrix} A \\ A_{12} \\ A_{13} \\ A_{14} \\ A_{23} \\ A_{24} \\ A_{34} \\ A_{123} \\ A_{124} \\ A_{134} \\ A_{234} \\ A_{12-34} \\ A_{13-24} \\ A_{14-23} \\ A_{1234} \end{bmatrix} = \frac{1}{C(C-1)(C-2)} M\vec{H}$$

Then $T(G,x,y) = \vec{A} \cdot \vec{K}$*.*

Due to the length, we will not present a proof of this, but we have tested the formula using Maple on many graphs constructed with a 4-separation. Additionally, we observe that the patterns in the matrix with respect to the previous formulas suggests its correctness. In particular, there seems to be a factor of $\prod_{i=2}^{k} \frac{1}{C-i+2}$ appearing in the formula for a k-separation.

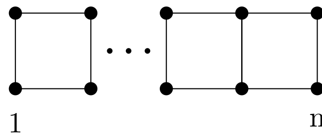
Chapter 4

Recurrences and Explicit Formulas for Restricted Families

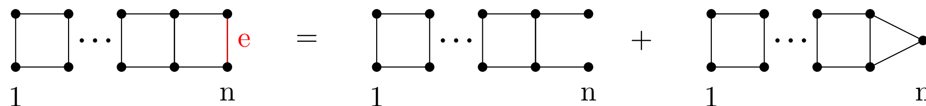
We will denote the path graphs on n vertices as P_n, and the cycle graphs on n vertices as C_n. To ease our notation, an equation involving graphs will denote the equations of the Tutte polynomials of the respective graphs.

4.1 Ladder Graph

The class of ladder graphs L_n is defined as the Cartesian product of the paths P_2 and P_n.



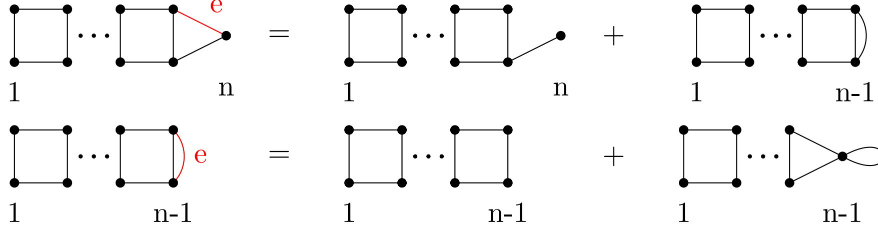
We will show the construction of the recurrence for the Tutte polynomials of ladder graphs, to illustrate the methodology that we will apply to the graphs in the following sections. We apply deletion-contraction strategically to decompose the main graph into graphs from a small set of intermediate classes. This will give us a system of recurrences in which we algebraically manipulate to obtain a single recurrence for the main graph. We begin by applying deletion-contraction to L_n on the edge e highlighted in red.



We will denote the graph resulting from contracting e as a triangle ladder A_n (simply a ladder with the final . Observe that in the graph, we have two cut edges, and deleting them we get L_{n-1}. Thus,

$$T(L_n) = x^2 T(L_{n-1}) + T(A_n) \tag{4.1}$$

Now we apply deletion-contraction on A_n



So we have that

$$T(A_n) = xT(L_{n-1}) + T(L_{n-1}) + yT(A_{n-1}) \tag{4.2}$$

This gives us a system of recurrence relations, and we want a single recurrence relation involving only the Tutte polynomials of ladder graphs. First add (1) + (2) to obtain

$$T(L_n) = (x^2 + x + 1)T(L_{n-1}) + yT(A_{n-1}) \tag{4.3}$$

Then we observe that by lowering the index n in (1) to $n-1$ and multiplying the equation by y, we have

$$\begin{aligned} yT(L_{n-1}) &= x^2 yT(L_{n-2}) + yT(A_{n-1}) \\ yT(A_{n-1}) &= yT(L_{n-1}) - x^2 yT(L_{n-2}) \end{aligned} \tag{4.4}$$

Adding (3) + (4), giving us an order two recurrence for the Tutte polynomials of ladder graphs

$$T(L_n) = (x^2 + x + y + 1)T(L_{n-1}) - x^2 yT(L_{n-2}) \tag{4.5}$$

For the initial conditions of the recurrence, observe that $T(L_1) = T(P_2) = x$ and $T(L_2) = T(C_4) = x^3 + x^2 + x + y$. Now we can solve this recurrence (for example, by using the characteristic polynomial $\lambda^2 - (x^2 + x + y + 1)\lambda + x^2 y$) and obtain the formula

$$T(L_n) = \frac{1}{2^n \sqrt{u}} \Big((x\sqrt{u} + v)(w + \sqrt{u})^{n-1} + (x\sqrt{u} - v)(w - \sqrt{u})^{n-1} \Big) \tag{4.6}$$

where $w = x^2 + x + y + 1$, $u = w^2 - 4x^2 y$, and $v = xw + 2(y - xy)$.

4.2 Prism Graph

The class of prism graphs Pr_n are the Cartesian products of P_2 and C_n, they correspond to the prisms with an n-polygon base in geometry. An example of a prism graph (hexagonal prism) Pr_6 is shown in Figure 4.1.

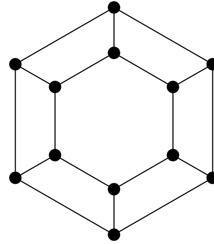
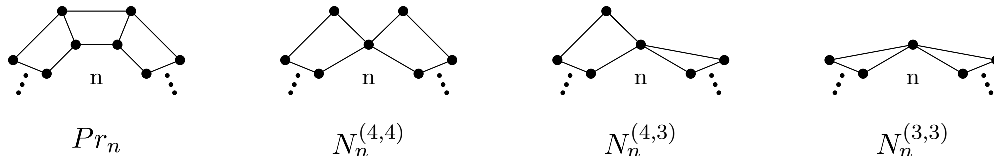


Figure 4.1: Prism graph

In [3], the recurrence for the Tutte polynomials of prism graphs was given but not proved. Using the strategy outlined in Section 4.1, we have constructed the same recurrence and used Maple to obtain the formula, shown below (for convenience we let $u = x^4 + 2x^3 - 2x^2y + 3x^2 + 2xy + y^2 + 2x + 2y + 1$ and $v = x^2 - 2xy + y^2 + 4x + 4y + 4$).

$$T(Pr_n) = \frac{x^2y^2 + x^{n+1}y - 2\,x^2y - 2\,xy^2 - x^{n+1} - x^ny + x^2 + xy + y^2 + x + y - 1}{x-1} + \frac{(xy - x - y)\left(\left(x + y + 2 + \sqrt{v}\right)^n + \left(x + y + 2 - \sqrt{v}\right)^n\right)}{2^n\,(x-1)} + \frac{\left(x^2 + x + y + 1 + \sqrt{u}\right)^n + \left(x^2 + x + y + 1 - \sqrt{u}\right)^n}{2^n\,(x-1)}$$

We will repeatedly make use of several intermediate families in constructing the recurrence (including the ladder graphs L_n), which we name below. The subscript n denotes the number of polygons in the graph.



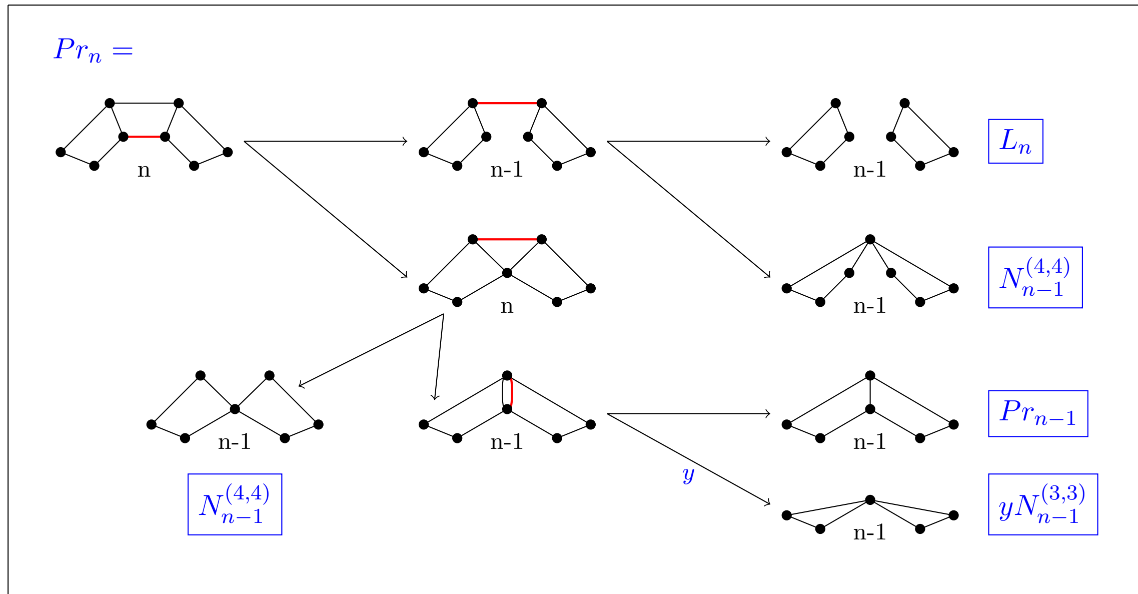
In order to keep our figures of the construction from being too complex, we use some simplifications. First we drop the $T(..)$ when referring to the Tutte polynomial, and any ellipsis will also be dropped. In the figures, red edges denote the edge that we will apply deletion-contraction to in the next step, and the arrows point to the resulting graphs in the computation tree. Our convention is that, when applying deletion-contraction to edge e, the top (or left) arrow goes $G - e$, and the bottom (or right) arrow goes to G/e. When we

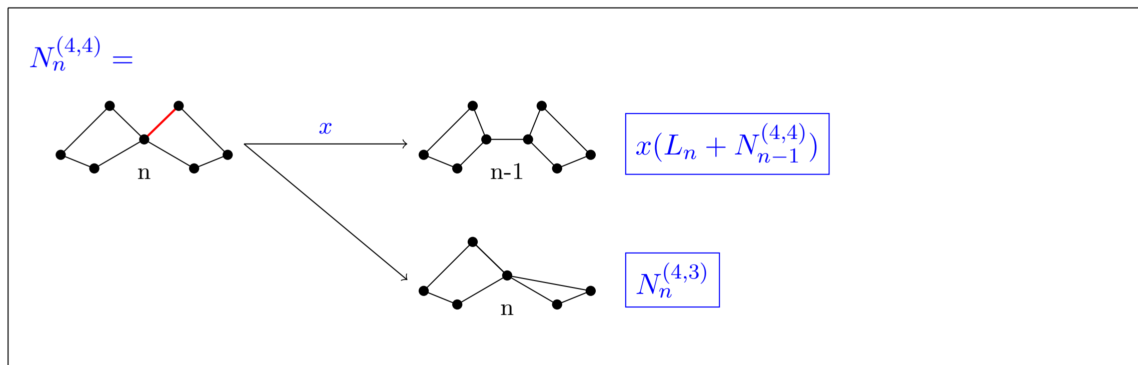
encounter cut edges and loops, instead of showing the intermediate graph, we automatically delete the cut edge/loop and show the factor that the edge introduced by labeling the arrow. These factors are accumulated at the leaves of our computation trees. We also process more than one edge per step when we encounter situations outlined by the following lemmas [8].

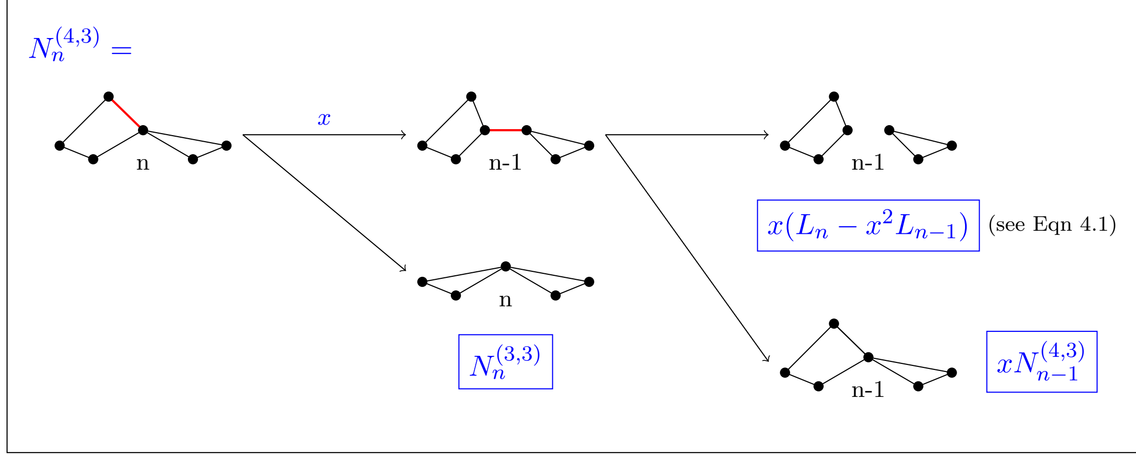
Lemma. *Let $E = v_1 \to v_2 \to ... \to v_{s+1}$ be a path in G such that the end vertices have degree higher than 2, and all internal vertices have degree exactly 2 (such a path is called an* ear*). Then $T(G) = (x^{s-1} + ... + x + 1)T(G - E) + T(G/E)$, where G/E denotes the graph with the entire ear contracted (all edges of ear removed, and the end vertices are joined).*

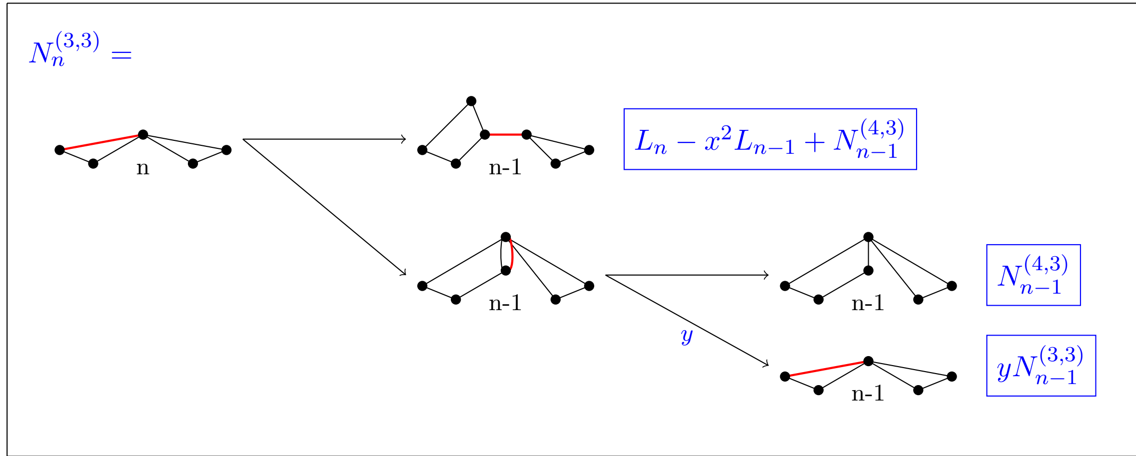
Lemma. *Let E be a multi-edge of G with multiplicity s. Then $T(G) = (y^{s-1} + ... + y + 1)T(G - E) + T(G/E)$.*

These lemma can be proven by induction on s, in both cases applying basic deletion-contraction on one of the edges. We begin the construction by applying deletion-contraction on a general prism graph:









For each of the four figures above, we sum up the results at the leaves of the computation tree (labeled with blue rectangles), which results in the following four linear equations:

$$T(Pr_n) = T(L_n) + T(Pr_{n-1}) + 2T(N_{n-1}^{(4,4)}) + yT(N_{n-1}^{(3,3)}) \tag{4.1}$$

$$T(N_n^{(4,4)}) = xT(L_n) + xT(N_{n-1}^{(4,4)}) + T(N_n^{(4,3)}) \tag{4.2}$$

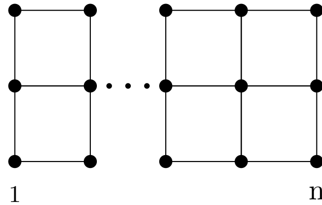
$$T(N_n^{(4,3)}) = xT(L_n) - x^3T(L_{n-1}) + xT(N_{n-1}^{(4,3)}) + T(N_n^{(3,3)}) \tag{4.3}$$

$$T(N_n^{(3,3)}) = T(L_n) - x^2T(L_{n-1}) + 2T(N_{n-1}^{(4,3)}) + yT(N_{n-1}^{(3,3)}) \tag{4.4}$$

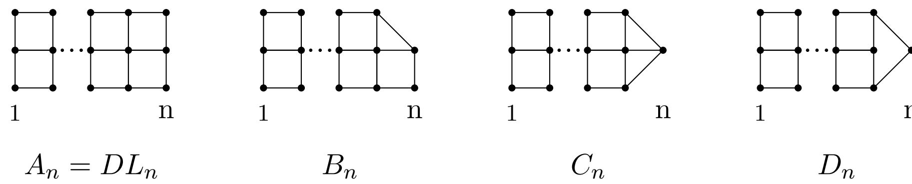
This system of recurrence relations can then be algebraically manipulated to get a linear homogeneous recurrence relation involving only the $T(Pr_n)$. We omit the details but note that an immediate first step is rewriting Equation 4.2 as $T(N_n^{(4,3)}) = T(N_n^{(4,4)}) - xT(L_{n-1}) + xT(N_{n-1}^{(4,4)})$ and substituting all instances to $T(N_n^{(4,3)})$ to eliminate it. Continuing in this manner, we obtain an order 6 recurrence for $T(Pr_n)$ that matches the one presented in [3].

4.3 Double Ladder Graph

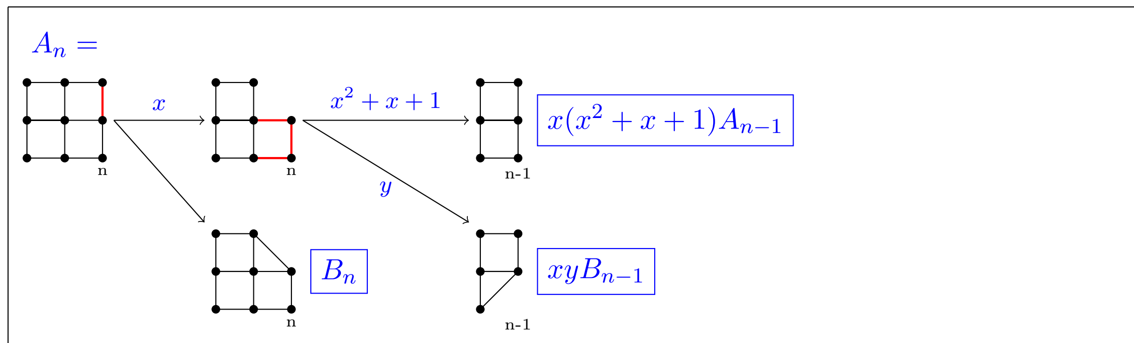
We will call the double ladder graphs DL_n as the Cartesian product of the paths P_3 and P_n.

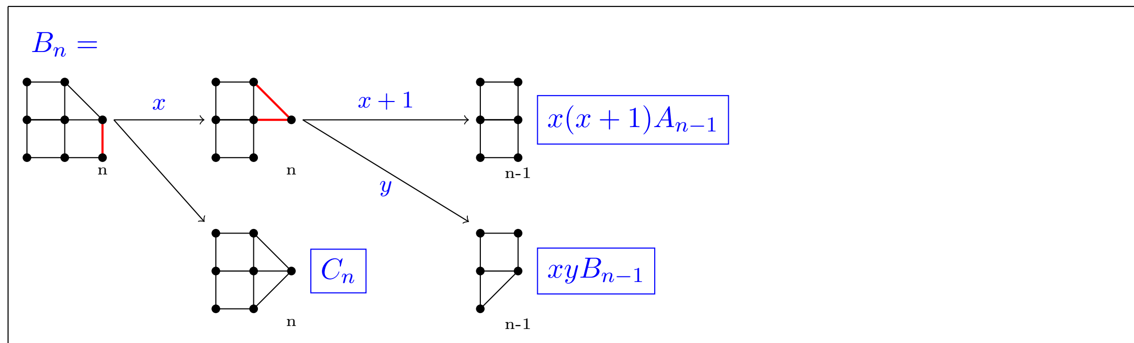


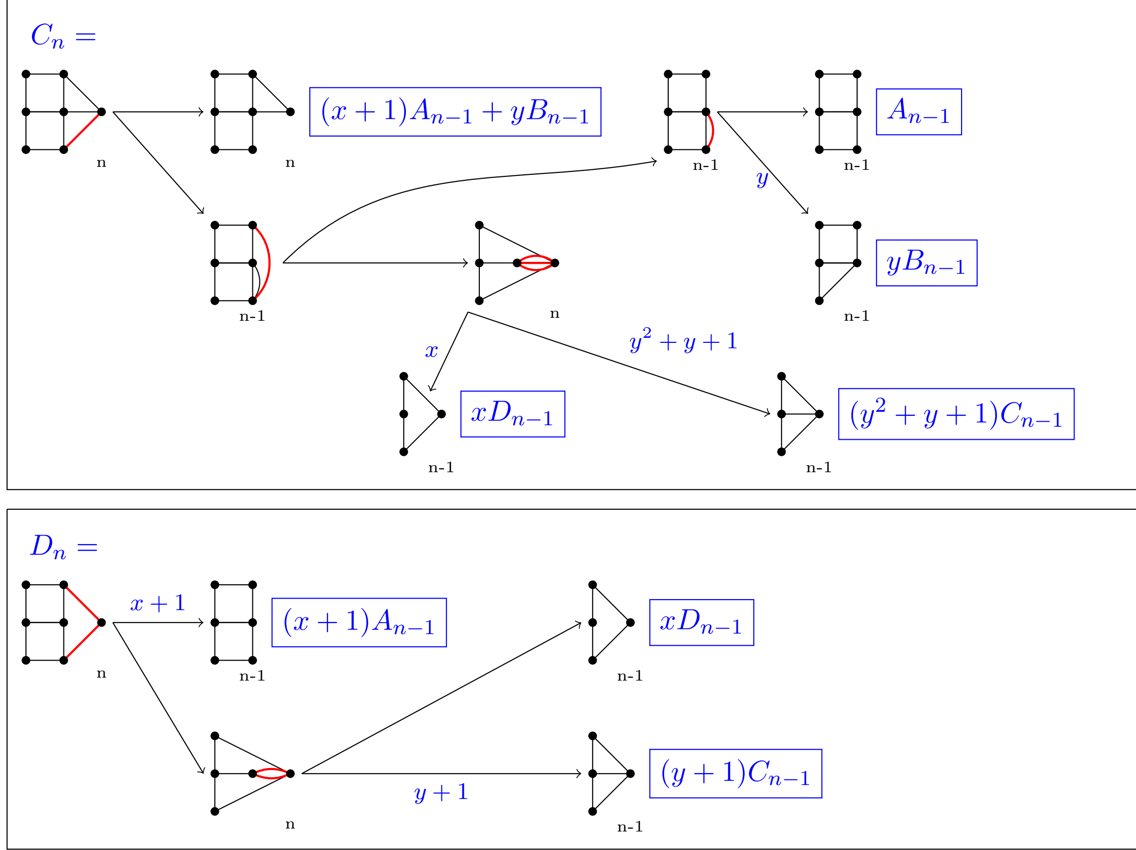
We will construct an order 5 recurrence, but since the characteristic polynomial of the recurrence does not factor, the formula obtained does not seem to have any practical usage. In the proof we will repeatedly make use of several intermediate families, which we name below (the names have no special meaning and were simply chosen alphabetically, here A_n and C_n no longer refer to the triangle ladder and cycle graphs).



The figures below will follow the notation and conventions of the previous section. We begin with applying deletion-contraction on A_n.







Again we sum up the results from each of the figures to get a system of linear equations:

$$T(A_n) = (x^3 + x^2 + x)T(A_{n-1}) + T(B_n) + xyT(B_{n-1}) \tag{4.1}$$

$$T(B_n) = (x^2 + x)T(A_{n-1}) + xyT(B_{n-1}) + T(C_n) \tag{4.2}$$

$$T(C_n) = (x+2)T(A_{n-1}) + 2yT(B_{n-1}) + (y^2 + y + 1)T(C_{n-1}) + xT(D_{n-1}) \tag{4.3}$$

$$T(D_n) = (x+1)T(A_{n-1}) + (y+1)T(C_{n-1}) + xT(D_{n-1}) \tag{4.4}$$

We transform this into a recurrence for $T(DL_n)$ with the techniques described previously. We let

$$a_1 = x^3 + 2x^2 + xy + y^2 + 4x + 3y + 3$$
$$a_2 = x^4y + x^3y^2 + x^4 + 3x^3y + 2x^2y^2 + xy^3 + 3x^3 + 4x^2y + 4xy^2 + 4x^2 + 5xy + 3x$$
$$a_3 = x^4y^3 + x^5y + 2x^4y^2 + 3x^4y + 2x^3y^2 + x^2y^3 + 2x^3y + 2x^2y^2 + x^2y$$
$$a_4 = x^5y^3$$

Then $T(DL_n) = a_1T(DL_{n-1}) - a_2T(DL_{n-2}) + a_3T(DL_{n-3}) - a_4T(DL_{n-4})$.

Chapter 5

Conclusion

5.1 Summary of Contributions

We have developed a new edge selection heuristic based on the SHARC vertex ordering [12] that performs well on random sparse graphs. We implemented a deletion-contraction Tutte polynomial algorithm using this heuristic in Maple and compared it to existing implementations, as well as investigating an algorithm that does not use the deletion-contraction method to see the limitations of our algorithm. In addition, we proved splitting formulas for the Tutte polynomial over vertex separations of size two to four, and compared them to the more general results for matroids have already been discovered. We also presented a family of graphs whose Tutte polynomials satisfy a linear homogeneous recurrence relation, and proved the recurrence constructively.

5.2 Future Work

As the ModSHARC vertex ordering was developed experimentally, it's likely possible that further improvements can be made to increase the range of graphs (both in terms of size and structure) for which it is effective. From the data we observed by splitting graphs with a 3-separation, computing the Tutte polynomials of very similar graphs using our implementation resulted in drastically different timings, investigating this area could lead to both an improved heuristic and a much higher speedup from parallel processing.

Bibliography

[1] Artur Andrzejak. Splitting formulas for tutte polynomials. *Journal of Combinatorial Theory, Series B*, 70(2):346 – 366, 1997.

[2] Laci Babai. Graph isomorphism in quasipolynomial time i: The local certificates algorithm, 2015. [Online; accessed 14-December-2015].

[3] N.L Biggs, R.M Damerell, and D.A Sands. Recursive families of graphs. *Journal of Combinatorial Theory, Series B*, 12(2):123 – 131, 1972.

[4] Andreas Bjorklund, Thore Husfeldt, Petteri Kaski, and Mikko Koivisto. Computing the tutte polynomial in vertex-exponential time. In *Foundations of Computer Science, 2008. FOCS'08. IEEE 49th Annual IEEE Symposium on*, pages 677–686. IEEE, 2008.

[5] Joseph Bonin and Anna De Mier. Tutte polynomials of generalized parallel connections. *Advances in Applied Mathematics*, 32(1):31–43, 2004.

[6] Thomas H Brylawski. A combinatorial model for series-parallel networks. *Transactions of the American Mathematical Society*, 154:1–22, 1971.

[7] C.M. Fortuin and P.W. Kasteleyn. On the random-cluster model. *Physica*, 57(4):536 – 564, 1972.

[8] Gary Haggard, David J Pearce, and Gordon Royle. Computing tutte polynomials. *ACM Transactions on Mathematical Software (TOMS)*, 37(3):24, 2010.

[9] Frank Heart, Alex McKenzie, JM McQuillan, and DC Walden. Arpanet completion report. *BBN Report. Bolt, Beranek and Newman Inc.(BBN). Also published in an edited version as BBN Report*, 4799:58–63, 1978.

[10] François Jaeger, Dirk L Vertigan, and Dominic JA Welsh. On the computational complexity of the jones and tutte polynomials. In *Mathematical Proceedings of the Cambridge Philosophical Society*, volume 108, pages 35–53. Cambridge Univ Press, 1990.

[11] Brendan D McKay. Nauty user's guide (version 2.4). *Computer Science Dept., Australian National University*, pages 225–239, 2007.

[12] Michael Monagan. A new edge selection heuristic for computing the tutte polynomial of an undirected graph. *DMTCS Proceedings*, 0(01), 2012.

[13] Michael Monagan and Alan Wong. Identities and heuristics for computing the tutte polynomial. Poster presented at 2013 SFU Symposium on Mathematics and Computation.

[14] J.G. Oxley and D.J.A. Welsh. Tutte polynomials computable in polynomial time. *Discrete Mathematics*, 109(1-3):185 – 192, 1992.

[15] David J Pearce, Gary Haggard, and Gordon Royle. Edge-selection heuristics for computing tutte polynomials. In *Proceedings of the Fifteenth Australasian Symposium on Computing: The Australasian Theory-Volume 94*, pages 153–162. Australian Computer Society, Inc., 2009.

[16] Kyoko Sekine, Hiroshi Imai, and Seiichiro Tani. Computing the tutte polynomial of a graph of moderate size. In *Algorithms and Computations*, pages 224–233. Springer, 1995.

[17] Alan D Sokal. The multivariate tutte polynomial (alias potts model) for graphs and matroids. *Surveys in combinatorics*, 327:173–226, 2005.

[18] William T Tutte. A contribution to the theory of chromatic polynomials. *Canad. J. Math*, 6(80-91):3–4, 1954.

[19] Eric W. Weisstein. Tutte polynomial. From MathWorld—A Wolfram Web Resource. Last visited on 21/10/2015.

[20] D.B. West. *Introduction to Graph Theory*. Featured Titles for Graph Theory Series. Prentice Hall, 2001.

Appendix A

Code

Included in this appendix is our Maple implementation of the Tutte polynomial algorithm that is outlined in Section 2.2.3. The first line contains the required Maple libraries that need to be loaded in order for the procedures to work.

The code is a collection of procedures, and the main procedure is `TuttePoly(G,x,y)`, which computes the Tutte polynomial of G. The function names are chosen so that none of them are protected by Maple as of Maple 2015 (for this reason the ordering procedure was named `ModSHARCorder` as `SHARCorder` was protected).

One method to use this code is to insert the functions into a worksheet and call the procedures. Note that there may be bugs due to character conversions when viewing this document; single quotation marks are supposed to be used in the code, and the quotations in the third-last line of the `LBlocks` procedure are supposed to be grave accents.

The main procedure `TuttePoly(G,x,y)` expects an adjacency list input for G, this can be obtained by the `Neighbors(G)` command in Maple. The example below shows a simple experiment for computing the Tutte polynomial of a random graph (the ... represents the our Tutte polynomial procedures) in a Maple worksheet.

```
>restart; with(GraphTheory): with(ListTools): with(RandomGraphs):

...

>G := RandomGraph(20,40);
>AdjListG := Neighbors(G);
>st := time(): TuttePoly(AdjListG,x,y)); time()-st;
```

This will print out the Tutte polynomial of G along with the CPU time required for the computation.

```
with(GraphTheory): with(ListTools):

# Output a shortest path (as a list) from a vertex in S to a vertex in S
# that has at least one vertex z not in S.
# For example, consider the cycle  [[2,6],[1,3],[2,4],[3,5],[1,4]]
# The shortest path from {1,3} to {1,3} is 1 -> 2 -> 3 not 1 -> 5 -> 4 -> 3
ModSHARC_BFS := proc(G::list,S::set(posint))
local n,m,P,Q,QQ,s,e,u,v,w,base,savepath,count;
   n := nops(G);
   m := add( nops(G[u]), u=1..n )/2;
   Q := Array(1..m+n);
   P := Array(1..n);
   base := Array(1..n);
   s := 1;
   e := 0;
   count := 0;
   savepath := [];
   for u in S do e := e+1; Q[e] := u; P[u] := u; base[u] := u; od;
   while s <= e do
       u := Q[s]; s := s+1; # remove from front of queue
       for v in {op(G[u])} do
           if member(u,S) and member(v,S) then next fi; # skip these edges
           if P[v] = 0 then # new vertex
              P[v] := u;
              if member(u,S) then base[v] := v; else base[v] := base[u] fi;
              e := e+1; Q[e] := v; # insert at back of queue
           elif P[u] <> v and base[u] <> base[v] then
              QQ := copy(Q);
              # Build path in Q in the order from S to u to v to S
              m := 1;
              w := u;
              while P[w] <> w do w := P[w]; m := m+1; od;
              n := m;
              QQ[n] := u;
              w := u;
              while P[w] <> w do w := P[w]; n := n-1; QQ[n] := w; od;
              n := m+1;
              QQ[n] := v;
              w := v;
              while P[w] <> w do w := P[w]; n := n+1; QQ[n] := w; od;
              if (n mod 2) = 0 and n < nops(savepath) then
                 return [seq( QQ[n-w+1], w=1..n )];
              elif count = 0 then
                 savepath := [seq( QQ[n-w+1], w=1..n )]; count := 1 fi;
           fi;
       od;
   od;
```

```
   savepath;
end:

# Output the final ordering that the ModSHARC ordering will use
# based on the scores computed
FinalSHARC_BFS := proc(G::list,S::set(posint),scores::list)
local a1,j,n,m,P,Q,QQ,s,e,u,v,w,base,path,savepath,count,arcscore;
   n := nops(G);
   m := add( nops(G[u]), u=1..n )/2;
   Q := Array(1..m+n);
   P := Array(1..n);
   base := Array(1..n);
   s := 1;
   e := 0;
   count := 0;
   savepath := []; arcscore := 0;
   for u in S do e := e+1; Q[e] := u; P[u] := u; base[u] := u; od;
   while s <= e do
       u := Q[s]; s := s+1; # remove from front of queue
       for v in {op(G[u])} do
           if member(u,S) and member(v,S) then next fi; # skip these edges
           if P[v] = 0 then # new vertex
              P[v] := u;
              if member(u,S) then
                 base[v] := v;
              else
                 base[v] := base[u]
              fi;
              e := e+1; Q[e] := v; # insert at back of queue
           elif P[u] <> v and base[u] <> base[v] then
              QQ := copy(Q);
              # Build path in Q in the order from S to u to v to S
              m := 1;
              w := u;
              while P[w] <> w do w := P[w]; m := m+1; od;
              n := m;
              QQ[n] := u;
              w := u;
              while P[w] <> w do w := P[w]; n := n-1; QQ[n] := w; od;
              n := m+1;
              QQ[n] := v;
              w := v;
              while P[w] <> w do w := P[w]; n := n+1; QQ[n] := w; od;
              if count = 0 then
                 savepath := [seq( QQ[n-w+1], w=1..n )]; count := 1;
                 for j in savepath[2..-2] do
                     arcscore := arcscore + scores[j];
```

```
                od;
             elif n < nops(savepath) then
                savepath := [seq( QQ[n-w+1], w=1..n )]; arcscore = 0;
                for j in savepath[2..-2] do
                    arcscore := arcscore + scores[j];
                od;
             elif n = nops(savepath) then
                path := [seq( QQ[n-w+1], w=1..n )]; a1 := 0;
                for j in path[2..-2] do
                    a1 := a1 + scores[j];
                od;
                if a1 < arcscore then
                   savepath := path; arcscore := a1
                fi;
             fi;
          fi;
      od;
   od;
   savepath;
end:

# Build a vertex ordering by shortest cycles back to the set of
# vertices seen so far.
ModSHARC := proc(G::list,s::posint,CYC,scores::list)
local score, left, seen, n, P, i, u, path, cyc, k, m, a1, a2, cycle, pos,
deg, Deg, minc, maxc, mind, maxd;
   n := nops(G);
   if nargs=1 then
      cycle := []; Deg := [];
      for i to n do
          cyc,deg := CycleTester(G,i); m := min(5,nops(cyc));
          a1 := add(cyc[k]/sqrt(k*1.0),k=1..m);
          cycle := [op(cycle),a1];
          a2 := add(deg[k]/sqrt(k*1.0),k=1..nops(deg));
          Deg := [op(Deg),a2];
      od;
      minc := FindMinimalElement(cycle);
      maxc := FindMaximalElement(cycle);
      mind := FindMinimalElement(Deg);
      maxd := FindMaximalElement(Deg);
      if mind = maxd and minc <> maxc then
          score := cycle;
      elif minc = maxc and mind <> maxd then
          score := Deg;
      elif minc = maxc and mind = maxd then
          score := cycle;
      else
```

```
         score := Deg/(maxd-mind)+0.5*cycle/(maxc-minc);
      fi;
      pos := FindMinimalElement(score,'position');
      return ModSHARC(G,pos[2],'cyc',score);
   fi;
   P[1] := s;
   seen := {s};
   left := {$1..n} minus seen;
   n := 1;
   cyc := [];
   while left <> {} do
      path := FinalSHARC_BFS(G,seen,scores);
      if cyc=[] then cyc := [nops(path)-1]
      else cyc := [op(cyc),nops(path)-2]; fi;
      if path=[] then u := left[1]; path := [u];
      else path := path[2..-2] fi;
      for u in path do n := n+1; P[n] := u; od;
      left := left minus {op(path)};
      seen := seen union {op(path)};
   od;
   if nargs=3 then CYC := cyc; fi;
   [seq( P[u], u=1..n )];
end:

CycleTester := proc(G::list,s::posint)
   local left, P, seen, n, u, path, cyc, deg;
   P[1] := s;
   n := nops(G);
   seen := {s};
   left := {$1..n} minus seen;
   n := 1;
   cyc := [];
   while left <> {} do
      path := ModSHARC_BFS(G,seen); # find next path from seen back to seen
      if cyc=[] then cyc := [nops(path)-1];
         deg := [seq(nops(G[path[i]]),i=1..nops(path)-1)]
      else
         cyc := [op(cyc),nops(path)-2];
         deg := [op(deg), seq(nops(G[path[i]]),i=2..nops(path)-1)]
      fi;
      if path=[] then
         u := left[1]; path := [u];
      else
         path := path[2..-2]
      fi;
      for u in path do n := n+1; P[n] := u; od;
      left := left minus {op(path)};
```

```
      seen := seen union {op(path)};
   od;
   cyc,deg;
end:

ModSHARCorder := proc(G::Graph) local V,i,n,A,N,C,H,sigma,e;
  sigma := NULL;
  for C in ConnectedComponents(G) do
      if nops(C) < 3 then sigma := sigma,op(C);
      else
           H := InducedSubgraph(G,C);
           V := op(3,H);
           n := nops(V);
           A := op(4,H);
           N :=  [seq( [op(A[i])], i=1..n )];
           if NumberOfEdges(G) > numelems(Edges(G)) then
               for i to nops(N) do
                    e := N[i];
                    if member(i,e) then
                        N := subsop(i=subs(i=NULL,e),N);
                    end if;
               od;
           fi;
           sigma := sigma, seq( V[i], i=ModSHARC(N) ); # use ModSHARC
     fi;
  od;
  [sigma];
end:

# Convert a Graph into a list of lists
NConvert := proc(G::Graph)
    return Neighbors(G);
end:

# BFS on a single component
BFS := proc(L::list,v::integer,A::Array)
    local Q; local N; local w;
    local front;
    local back;
    A[v] := 1;
    Q := Array(1..nops(L));
    front := 1;
    back := 1;
    Q[1] := v;
    while front <= back do
        N := L[Q[front]]; front := front + 1;
        for w in N do
```

```
            if A[w] = 0 then
               Q[back+1] := w; A[w] := 1; back := back + 1
            end if;
        od;
    od;
    return [seq(Q[i],i=1..back)];
end:

# BFS on multiple components
FullBFS := proc(L::list)
    local A;
    local Q;
    local i;
    A := Array(1..nops(L));
    Q := [];
    for i to nops(L) do
       if i = 1 then
          Q := [[op(BFS(L,i, A))]];
       elif A[i] = 0 then
          Q := [op(Q),BFS(L,i, A)]
       end if;
    od;
    return Q;
end:

# Relabelling a graph based on a given ordering
Relabel := proc(L::list,O::list)
local i, j, p, N;
    N := [seq([],i=1..nops(O))];
    for i to nops(L) do
        N[i] := L[O[i]];
        for j to nops(N[i]) do
            member(N[i][j], O, 'p');
            N[i][j] := p;
        od;
        N[i] := sort(N[i]);
    od;
    return N;
end:

# Finds the blocks of a graph
LBlocks := proc(L::list,A::name)
local i, n, mark, M, bcount, blocks, lowpt, stack, DFS, artp, root, dfi, dfic;
    n := nops(L);
    mark := Array(1..n);
    lowpt := Array(1..n);
    stack := Array(1..n);
```

```
    blocks := Array(1..n);
    artp := Array(1..n);
    dfi := Array(1..n);
    DFS := proc(u,parent)
        local v, top;
        mark[u] := 1;
        if u <> root then
           lowpt[u] := dfi[parent]
        end if;
        M := M + 1; stack[M] := u;
        dfi[u] := dfic; dfic := dfic + 1;
        for v in L[u] do
            if v = parent then
            elif mark[v] = 1 then lowpt[u] := min(lowpt[u],dfi[v]);
            else top := M; DFS(v,u); lowpt[u] := min(lowpt[u],lowpt[v]);
                if lowpt[v] = dfi[u] then
                   bcount:=bcount+1; artp[u] := artp[u]+1;
                   blocks[bcount]:=[u,seq(stack[i],i=top+1..M)];
                   M:=top
                end if;
            end if;
        od;
    end;
    bcount:=0; dfic := 1;
    for root to n do
        if mark[root]=0 and 0<nops(L[root]) then M:=0; DFS(root,0);
            if artp[root] < 2 then artp[root] := 0 end if;
        elif mark[root]=0 and nops(L[root])=0 then
            bcount:=bcount+1; blocks[bcount]:=[root]
        end if ;
    od;
    if 1 < nargs then
       A := [seq(`if`(artp[u] = 0, NULL, u), u = 1 .. n)]
    end if;
    blocks:=[seq([seq(u,u=blocks[i])],i=1..bcount)]
end:

# Splits a graph into its blocks
BlockInduce := proc(L::list,blocks::list,artp::list)
    local i, j, k, m, N, BlockRelabel;
    N := [];
    BlockRelabel := proc(L::list,b::list)
        local i, j, p, N;
        N := [seq([],i=1..nops(L))];
        for i to nops(L) do
            N[i] := L[i];
            for j to nops(N[i]) do
```

```
                member(N[i][j], b, 'p');
                N[i][j] := p;
            od;
            N[i] := sort(N[i]);
        od;
    return N;
    end:
    for i to nops(blocks) do
        if nops(blocks[i]) = 1 then
            N := [op(N),[]];
        elif nops(blocks[i]) = 2 then
            m := Occurrences(blocks[i][1],L[blocks[i][2]]);
            N := [op(N),[[seq(2,i=1..m)],[seq(1,i=1..m)]]];
        else
            N := [op(N),[]];
            for j to nops(blocks[i]) do
                if blocks[i][j] in artp then
                    N[i] := [op(N[i]),[]];
                    for k to nops(L[blocks[i][j]]) do
                        if L[blocks[i][j]][k] in blocks[i] then
                            N[i][j] := [op(N[i][j]), L[blocks[i][j]][k]];
                        end if;
                    od;
                else
                    N[i] := [op(N[i]),L[blocks[i][j]]];
                end if;
            od;
            N[i] := BlockRelabel(N[i],blocks[i]);
        end if;
    od;
    return N;
end:

EdgeDeletion := proc(L::list,u::integer,v::integer)
local N;
    N := subsop(u = L[u][1..nops(L[u])-1], v = L[v][1..nops(L[v])-1], L);
    return N;
end:

EdgeContraction := proc (G, i, j)
local H, k;
   H := subsop(i = NULL, j = [op(G[i]), op(subs(i = NULL, G[j]))], G);
   H := map(sort, subs([i = j-1, seq(k = k-1, k = i+1 .. nops(G))], H));
   return H
end:

ECutTest := proc(L::list,u::integer,v::integer)
```

```
    local i, seen, new, neighboors;
    if nops(L[v]) = 0 then return true fi;
    seen := {};
    new := {u};
    while new <> {} do
        seen := seen union new;
        neighboors := {seq(op(L[i]),i=new)};
        new := neighboors minus seen;
        if member(v,new) then return false fi;
    od;
    return true
end:

TuttePoly := proc(G::list,x::algebraic,y::algebraic)
    local B, BB, L, TPoly, T, GG, z;
    TPoly := proc(L::list,x::algebraic,y::algebraic)
        option remember;
        local EDel, ECon, e, c, i, j, k, f, l, N;

        # Case 1 (e = 0)
        if nops(L) = 1 and nops(L[1]) = 0 then return 1 end if;

        # Case 2 (Loop or MultiCutEdge)
        for i to nops(L) do
            e := L[i];
            if e=[] then
                N:=subs([seq(k=k - 1,k=i+1..nops(L))],subsop(i=NULL,L));
               return TPoly(N,x,y)
            elif member(i,e) then
                c := numboccur(e,i);
                N := subsop(i=subs(i=NULL,e),L);
                return expand(y^c*TPoly(N,x,y))
            elif e[1] = e[-1] then
                c := nops(e); j := e[1];
                ECon := subsop(i = [], j = subs(i = NULL, L[j]), L);
                return normal(x+add(y^i, i = 1 .. c-1))*TPoly(ECon,x,y);
            end if;
        od;

        # Failing the three above cases, we choose an edge
        EDel := subsop(1 = L[1][2..nops(L[1])],
           L[1][1] = L[L[1][1]][2..nops(L[L[1][1]])], L);
        if ECutTest(EDel,1,L[1][1]) then
           return expand(x*TPoly(EdgeContraction(EDel,1,L[1][1]),x,y))
        else
           return expand(TPoly(EdgeContraction(EDel,1,L[1][1]),x,y))+
              expand(TPoly(EDel,x,y))
```

```
        end if;
    end:

    B := LBlocks(G,'A');
    if nops(B) > 1 then
        BB := BlockInduce(G,B,A);
        GG := Array(1..nops(B));
        for z to nops(B) do
            GG[z] := Graph(BB[z]);
            BB := subsop(
               z = Relabel(BB[z],ModSHARCorder(GG[z])),
               BB);
        od;
        return expand(mul(TPoly(BB[i],x,y),i=1..nops(BB)));
    end if;
    L := Relabel(G,ModSHARCorder(Graph(G)));
    T := TPoly(L,x,y);
    TPoly := subsop(4=NULL,eval(TPoly));
    return T;
end:
```