A new edge selection heuristic for computing the Tutte polynomial of an undirected graph.

Michael Monagan[1†]

[1] *Department of Mathematics, Simon Fraser University, Burnaby, B.C., V5A 1S6, CANADA.*

Abstract We present a new edge selection heuristic and vertex ordering heuristic that together enable one to compute the Tutte polynomial of much larger sparse graphs than was previously doable. As a specific example, we are able to compute the Tutte polynomial of the truncated icosahedron graph using our Maple implementation in under 15 minutes on a single CPU. This compares with a recent result of Haggard, Pearce and Royle whose special purpose C++ software took one week on 150 computers.

Keywords: Tutte polynomials, edge deletion and contraction algorithms, NP-hard problems.

1 Introduction

Let G be an undirected graph. The Tutte polynomial of G is a bivariate polynomial $T(G, x, y)$ which contains information about how G is connected. It is also the most general graph invariant that can be defined by edge deletion and edge contraction. We recall Tutte's original definition for $T(G, x, y)$. Let $e = (u, v)$ be an edge in G. Let $G - e$ denote the graph obtained by deleting e and let $G \,/\, e$ denote the graph obtained by contracting e, that is, first deleting e then joining vertexes u and v. Figure 1 below shows an example of edge contraction.

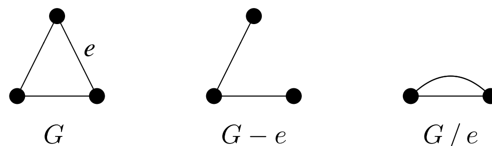


Fig. 1: Graph edge deletion and contraction.

†This work was supported by the Mprime NCE of Canada

subm. to DMTCS © by the authors Discrete Mathematics and Theoretical Computer Science (DMTCS), Nancy, France

Definition 1 *Let G be a connected undirected graph. The* Tutte polynomial $T(G,x,y)$ *is the bivariate polynomial defined by*

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

This definition immediately gives a recursive algorithm for computing $T(G,x,y)$. In general, a naive implementation of the algorithm will make an exponential number of recursive calls, because of the last case in (1). If G has n vertexes and m edges, the number of recursive calls $C(n+m)$ is bounded by

$$C(n+m) \le C(n+m-1) + C(n-1+m-1).$$

This is the Fibonacci sequence. Hence $C(n+m) \in O(1.618^{n+m})$. One way to reduce $C(n+m)$ is to apply the following theorem.

Theorem 1 (Tutte [9]) *Let G be a graph with m biconnected components (blocks) $B_1, B_2, \ldots, B_m$. Then $T(G,x,y) = \Pi_{i=1}^{m} T(B_i,x,y)$.*

Another way to reduce $C(n+m)$ is to "remember" the Tutte polynomials computed in the computation tree and use a graph isomorphism test to test whether a graph in the computation tree has been seen before. In [3], Haggard, Pearce and Royle present timings for random graphs (cubic, quartic, and dense) that shows that employing graph isomorphism is very effective; it roughly doubles the size (doubles $n+m$) of the graphs that can be handled in a given time. A factor in determining how effective this isomorphism testing is, is the order in which the edges are selected. In [7], Haggard, Pearce and Royle investigate various edge ordering heuristics. Two heuristics, which they call MINDEG and VORDER, are found to perform consistently better than random selection. In section 2 we describe the MINDEG and VORDER heuristics and present a new edge selection heuristic. The VORDER heuristic, and our new heuristic, also depend on the ordering of the vertexes in G. We present an ordering that we have found works particularly well with our edge selection heuristic. In section 3 we describe our Maple implementation and explain how we test for isomorphic graphs in the computation tree. Another experimental finding in this paper is that the VORDER edge selection heuristic, and our new edge selection heuristic result in over 95% of isomorphic graphs in the computation tree being identical. In section 4 we present benchmarks comparing the three heuristics with and without the new vertex ordering. The data presented shows that our new heuristic again roughly doubles the size of sparse graphs that can be handled in a given time.

We end the introduction with some further information about available software for computing Tutte polynomials and related polynomials. Useful references include the very good Wikipedia webpage http://en.wikipedia.org/wiki/Tutte_polynomial and Bollobás' text [1]. The graph theory packages in Mathematica and Maple include commands for computing Tutte polynomials. The Mathematica algorithm does not look for identical or isomorphic graphs in the computation tree (see [8]). The TuttePolynomial command in Maple 11 and more recent versions (see [4]) uses the VORDER heuristic and hashing to test for identical graphs in the computation tree. The fastest available software for computing Tutte polynomials and related polynomials is that of Haggard, Pearce and Royle [7, 3]. It is available on David Pearce's website at `http://homepages.ecs.vuw.ac.nz/~djp/tutte/`. It uses the canonical graph ordering available in Brendan Mckay's nauty package (see [5]) to identify isomorphic graphs.

Definition 2 *Let G be an undirected graph. The* reliability polynomial *of G, denoted $R_p(G)$, is the probability that G remains connected when each edge in G fails with probability p.*

Definition 3 *Let G be an undirected graph. The* chromatic polynomial *of G, denoted $P_\lambda(G)$, counts the number of ways the vertexes of G can be colored with λ colors.*

For example, $R_p(\bullet\!\!-\!\!\!-\!\!\bullet) = 1 - p$ and $P_\lambda(\bullet\!\!-\!\!\!-\!\!\bullet) = \lambda(\lambda - 1)$. The reliability and chromatic polynomials can also be computed by the edge deletion and contraction algorithm. If G has n vertexes and m edges, the they are related to the Tutte polynomial as follows:

$$R_p(G) = (1-p)^{(n-1)}\, p^{(m-n+1)}\, T(G, 1, p^{-1}), \tag{2}$$

$$P_\lambda(G) = (-1)^{(n-1)}\, \lambda\, T(G, 1-\lambda, 0). \tag{3}$$

Since graph coloring is NP$-$complete, it follows that computing the the chromatic polynomial is NP$-$hard. Thus (3) implies computing the Tutte polynomial is also NP$-$hard. It is also known that computing $R_p(G)$ is NP-hard (see [6]). This does not mean, however, that computing the Tutte polynomial for a given graph is not polynomial time. Our new edge selection heuristic is polynomial time for some non-trivial structured sparse graphs.

2 Edge selection heuristics.

In applying the identity $T(G) = T(G-e) + T(G/e)$ we are free to choose any edge which is neither a cut-edge nor a loop. [If G has a cut-edge or loop, then those edges should be processed first.] In [7], Haggard, Pearce and Royle propose two heuristics, the minimum degree heuristic (MINDEG) and the vertex order heuristic (VORDER). We describe the heuristics here and introduce our new heuristic which is a variation on VORDER.

2.1 The minimum degree heuristic: MINDEG

Consider the graph G in Figure 2 below.

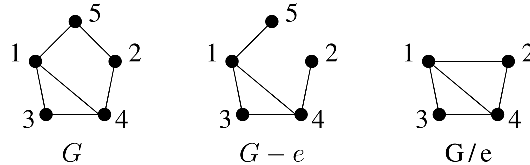


Fig. 2: The minimum degree heuristic.

The minimum degree heuristic picks the edge $e = (u, v)$ where u is the first vertex of minimum degree ($u = 2$ in the example) and v is the first vertex adjacent to u of minimum degree ($v = 5$ in the example). Hence $e = (2, 5)$ is chosen. Shown in the figure are the graphs $G - e$ and $G\,/\,e$. The reader can see that the next edge that will be selected in $G - e$ is the edge $(2, 4)$, which is a cut-edge. The algorithm will then contract the edge $(2, 4)$, then select the edge $(1, 5)$, another cut-edge. After contracting $(1, 5)$ what is left is the triangle on vertexes $1, 3, 4$. For the graph $G\,/\,e$, the MINDEG heuristic selects the edge (2,1). After deleting (2,1), MINDEG will select and contract the edge (2,4) again yielding the triangle $1, 3, 4$. This example shows how identical graphs in the computation tree arise.

2.2 The vertex order heuristic: VORDER

Consider again the graph G shown in the Figure 3 below.

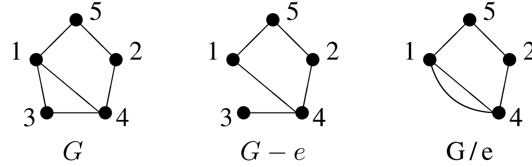


Fig. 3: The VORDER-pull heuristic.

The vertex order heuristic picks the edge $e = (u, v)$ where u is simply the first vertex in the G and v is the first vertex adjacent to u. In our example $u = 1$, $v = 3$, hence $e = (1, 3)$ is chosen. Shown in Figure 3 are the graphs $G - e$ and G / e where when we contracted the edge $e = (1, 3)$ we "pulled" vertex 3 down to vertex 1. The next edge selected in G / e will be one of the edges (1,4).

There is alternative choice here when constructing the graph G / e. Instead of "pulling" vertex $v = 3$ down to $u = 1$, if instead we "push" vertex $u = 1$ up to $v = 3$ we get the contracted graph shown in Figure 4 below. Observe that the two contracted graphs G / e in figures 3 and 4 are isomorphic. However, in the vertex order heuristic, the next edge selected in G / e is different. In figure 3 the vertex order heuristic selects edge (1,4). In figure 4 it selects edge (2,4). We will call the two vertex order heuristics VORDER-pull and VORDER-push, respectively.

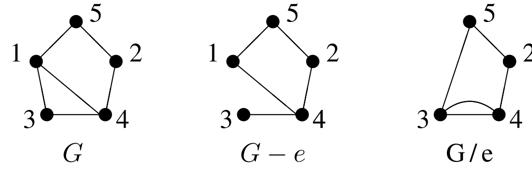


Fig. 4: The VORDER-push heuristic.

To visualize the difference between VORDER-pull and VORDER-push, picture the computation tree of graphs produced by the algorithm as it applies the identity

$$T(G) = T(G - e) + T(G / e).$$

On the left of the computation tree we repeatedly delete edges. On the right of the tree we repeatedly contract edges. The two vertex order heuristics differ when we contract. In the VORDER-pull heuristic, we always select the same first vertex and contract (pull) other vertexes to it thus typically increasing the degree of the first vertex. In the VORDER-push heuristic, we select the first vertex and push it away (into the middle of the graph) and move on to the next vertex in the ordering. Thus one measurable difference between VORDER-pull and VORDER-push is that the degree of the vertex u selected will generally be greater in VORDER-pull than in VORDER-push.

2.3 The vertex label ordering

The VORDER-pull and VORDER-push heuristics, and also to a lesser extent, the MINDEG heuristic, also depend on the input permutation of the labels of the vertexes in G. All three heuristics are sensitive to this ordering with a random ordering producing a bad behavior. In [7], Haggard, Pearce and Royle state "using an ordering where vertexes with higher degree come lower in the ordering generally also gives better performance". Their idea is to increase the probability that more identical graphs appear *higher* in the computation tree. To achieve this we propose to label the vertexes in the input graph in an order so that the algorithm deletes and contracts edges *locally*. We found that the following vertex ordering heuristic works well. To simplify the presentation we assume G is connected and has no vertex of degree 1. We describe it below with pseudo-code and an example.

Algorithm **SHARC** - short arc order.
Input: An undirected graph G on $n > 0$ vertexes $V = \{1, 2, ..., n\}$.
Assumes G is connected and has no vertex of degree 1.

1 Initialize the ordered list $S = [1]$

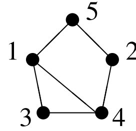
2 **while** $|S| < n$ do the following

3 Using breadth first search (BFS), starting from the vertexes in S find the first path from S back to S which includes at least one new vertex, that is, find a path $u \rightarrow v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_m \rightarrow w$ where $u \in S$, $w \in S$, $m > 0$, $v_i \in V \backslash S$. We argue that such a path must exist if G is connected and has no vertex of degree 1. Now append $v_1, v_2, ..., v_m$ to S.

end while

5 **output** S.

We explain step 3 with an example. Consider again the graph G below.



Initially we have $S = [1]$. Using BFS we insert all vertexes adjacent to the vertexes in S not already in S into a queue Q. In the example, we obtain $Q = [3, 4, 5]$. Hence we have paths $1 \rightarrow 3$, $1 \rightarrow 4$ and $1 \rightarrow 5$ which we maintain in an array $P = [0, 0, 1, 1, 1]$, that is $P_3 = 1$ stores the edge from 1 to 3 and $P_1 = 0$ indicates the end of a path. We take the first vertex 3 from Q and consider the new edge $(3, 4)$. Since P_4 is not zero we know there is a path from 1 back to 4 stored in P. Since 3 came from Q we know there is a path from 1 to 3 stored in P. Thus we are done this iteration; we extract the path $1 \rightarrow 3 \rightarrow 4 \rightarrow 1$ from P and append $3, 4$ to S obtaining $S = [1, 3, 4]$. In the second iteration the algorithm will find the path $1 \rightarrow 5 \rightarrow 2 \rightarrow 4$ and set $S = [1, 3, 4, 5, 2]$. Since $|S| = 5$ the algorithm stops. The reader can see that the algorithm finds a short cycle in the first iteration, then in the subsequent iterations, finds short arcs from S back to S. We will call this a short arc ordering (SHARC). By picking the first path found using BFS, the short arc ordering maintains locality in S. Although it would be simpler to order the vertexes in simple breadth first search order, that ordering did not prove to be as good as SHARC in our experiments.

3 Maple Implementation

We use a list of neighbors representation for a multi-graph in our Maple implementation. We illustrate with an example in Figure 5 below.

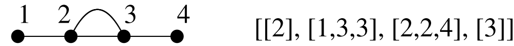


Fig. 5: Maple list of lists data structure for G

To identify identical graphs in the computation tree we make use of `option remember`. This is a feature of the Maple programming language that enables our Maple procedure to automatically identify identical graphs in the computation tree using hashing. For this to work we canonically re-label vertexes to be $1, 2, ..., n-1$ after edge contraction.

To identify non-equal isomorphic graphs we have implemented our own graph isomorphism test for multi-graphs as the `IsIsomorphic` command in Maple's `GraphTheory` package (see [4]) treats simple graphs only. Instead of searching all previous graphs, we first hash on the characteristic polynomial of the Laplacian matrix of G, a known graph invariant. The Laplacian matrix is an n by n matrix $D - A$ where D is the degree matrix of G and A is the adjacency matrix of G. For increased efficiency, we compute the characteristic polynomial of $D - A$ modulo a machine prime p. This can be computed in $O(n^3)$ arithmetic operations in $\mathbb{F}_p$. See Algorithm 2.2.9 in Chapter 2 of [2].

Our Maple code may be downloaded from `http://www.cecm.sfu.ca/~mmonagan/tutte`

4 Experiments

4.1 Random cubic graphs

In this experiment we generated five random connected cubic graphs on n vertexes. We computed the average and median time it takes our Maple program to compute the Tutte polynomial, using the MINDEG, VORDER-pull and VORDER-push heuristics, on a 2.66 Ghz Intel Core i7 computer. We do this for two permutations of the vertex labels, random (see Table 1) and SHARC (the short arc ordering) (see Table 2). The data shows that VORDER-push with SHARC is *much* better than VORDER-pull, and VORDER-push with SHARC is *much* better than VORDER-push with the input random ordering. This extends the size of the graphs for which Tutte polynomials can be computed. We find similar results for random quartic graphs.

4.2 Generalized Petersen graphs.

The generalized Petersen graph $P(n, k)$ with $1 \leq k < n/2$ is a cubic graph on $2n$ vertexes. Figure 6 shows $P(5, 1)$ and $P(5, 2)$. $P(5, 2)$ is the familiar Petersen graph. To construct $P(n, k)$ the vertexes are divided into two sets $1, 2, ..., n$ and $n + 1, n + 2, ..., 2n$, which are placed on two concentric circles as shown in figure 4. The first set of vertexes are connected in a cycle $1, 2, ..., n, 1$. The second set are connected to the first with vertex i connected to $n + i$ for $1 \leq i \leq n$. The second parameter governs how the second set is connected. Connect $n + i$ to $n + (n + i \pm k \mod n)$ for $1 \leq i \leq n$.

		MINDEG heuristic		VORDER pull		VORDER push	
n	m	ave	med	ave	med	ave	med
16	24	0.47	0.50	0.70	0.63	0.22	0.15
18	27	1.66	1.72	2.45	2.28	0.96	0.94
20	30	5.27	4.73	7.31	7.91	2.06	1.75
22	33	16.48	14.43	23.91	17.74	9.26	9.45
24	36	85.58	72.49	136.33	94.60	48.14	58.65

Tab. 1: Timings in CPU seconds for random cubic graphs with n vertices using random vertex order.

		MINDEG heuristic		VORDER pull		VORDER push	
n	m	ave	med	ave	med	ave	med
16	25	0.23	0.20	0.41	0.36	0.03	0.03
18	27	0.77	0.83	1.22	1.11	0.05	0.04
20	30	2.32	2.06	3.94	4.16	0.08	0.07
22	33	7.52	5.76	15.67	10.86	0.32	0.26
24	36	31.88	31.78	63.68	52.76	0.51	0.70
26	39					0.78	0.42
30	45					2.63	2.42
34	51					8.59	4.93
38	57					76.09	7.86
42	63					159.61	57.96
46	69					463.08	390.98

Tab. 2: Timings in CPU seconds for random cubic graphs with n vertices using SHARC vertex order.

The SHARC vertex order for $P(5,1)$ and $P(5,2)$ is [1,2,7,6,10,5,4,3,8,9] and [1,5,4,3,2,8,6,9,10,7] respectively. In Table 3 below, we compare the time it takes to compute the Tutte polynomials of $P(n,3)$ for increasing n using the VORDER-push heuristic. Shown in the column labelled #calls is the number of recursive calls made by the algorithm. Column #ident counts the number of recursive calls for which the graph is identical to a graph previously computed in the computation tree. Column #isom counts the number of recursive calls for which the graph is not identical but isomorphic to a graph previously computed in the computation tree. Both heuristics are using the SHARC vertex order which is better than the vertex order in which the graphs are constructed.

In comparing the data for $P(n,3)$, it's clear that VORDER-push is much better than VORDER-pull.

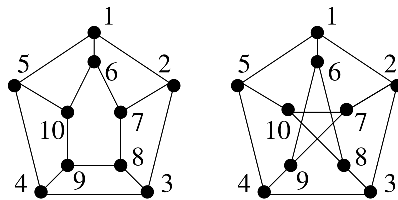


Fig. 6: Petersen graphs $P(5,1)$ and $P(5,2)$.

			VORDER-pull with SHARC				VORDER-push with SHARC			
n	$\|V\|$	m	time	#calls	#ident	#isom	time	#calls	#ident	#isom
8	16	24	1.30	28641	10419	0	0.11	2980	1181	0
9	18	27	1.47	30235	9818	3	0.07	1423	543	1
10	20	30	4.83	90772	31049	22	0.23	4739	1889	7
11	22	33	28.27	434402	149286	244	0.76	12833	5176	20
12	24	36	36.84	471530	152284	978	1.26	18644	7454	31
13	26	39	190.76	1668636	552034	7072	2.55	30168	12124	63
14	28	42	875.05	4035615	1346519	45340	4.50	41706	16691	184
15	30	44	2227.01	6330229	2016961	149699	7.35	52753	20818	388
16	32	48	9312.27	18010314	5813528	799349	11.47	66086	25975	687
18	36	54					22.48	93584	36495	1294
20	40	60					37.58	122869	47766	2002
22	44	66					53.46	151954	58873	2746
24	48	72					81.56	181918	70346	3487
26	52	78					114.26	211681	81767	4240
28	56	84					156.69	241364	93134	4995
30	60	90					210.17	271434	104649	5740

Tab. 3: Data for $P(n, 3)$ for VORDER-pull and VORDER-push.

In fact, VORDER-push is polynomial time in n. The reader can see that the number of graphs (column #calls) is increasing linearly with n by approximately $15,000/2$. We find the same linear increase for VORDER-push for $P(n, 1)$, $P(n, 2)$, $P(n, 3)$ and $P(n, 4)$. For $P(n, 5)$ and $P(n, 6)$ the data is not clear. The data for $P(n, 3)$ also shows that a high percentage of isomorphic graphs in the computation tree are identical (compare columns #ident and #isom). In Table 4 we show data for $P(n, 6)$.

		VORDER-pull				VORDER-push			
n	girth	time(s)	#calls	#ident	#isom	time(s)	#calls	#ident	#isom
13	5	85.55	875232	270060	5562	1.41	31968	7821	0
14	6	1262.37	5524084	1807371	76980	5.12	97699	26474	1
15	5					17.67	317811	89410	26
16	7					87.36	1015162	289361	29
17	6					25.12	374269	109601	88
18	3					3.32	36638	8721	97
19	6					28.32	282938	76715	489
20	7					479.94	3271429	963127	1393
21	7					1630.94	8789909	2706765	901

Tab. 4: Data for $P(n, 6)$ for VORDER-pull and VORDER-push.

The irregularity of the data in Table 4 is partly explained by low girth. In particular, $P(18, 6)$ has 6 triangles. The girth of $P(n, k)$ is a minimum when k divides n where the girth is n/k. In Table 5 we have fix n to be 17 and vary k to show the dependence on the girth.

			VORDER-pull			VORDER-push		
k	$\|V\|$	girth	time(s)	#calls	deg	time(s)	#calls	deg
1	28	4	6.12	54040	6.48	0.16	693	2.10
2	28	5	209.33	1362412	5.19	0.65	4727	2.30
3	28	6	806.92	4035615	4.32	3.82	40142	2.47
4	28	7	2273.75	8430139	4.61	7.71	88579	2.49
5	28	6	1218.51	6208087	4.49	5.62	71717	2.50
6	28	6	979.73	5524084	4.44	6.43	71054	2.47

Tab. 5: Data for $P(14, k)$ for VORDER-pull and VORDER-push. Column deg shows the average degree of the first vertex in the computation tree.

4.3 The truncated icosahedron graph.

The Tutte polynomial of a planar graph G and its dual G^* are related by $T(G, x, y) = T(G^*, y, x)$. Shown in Figure 7 is the truncated icosahedron graph TI and it's dual TI^*.

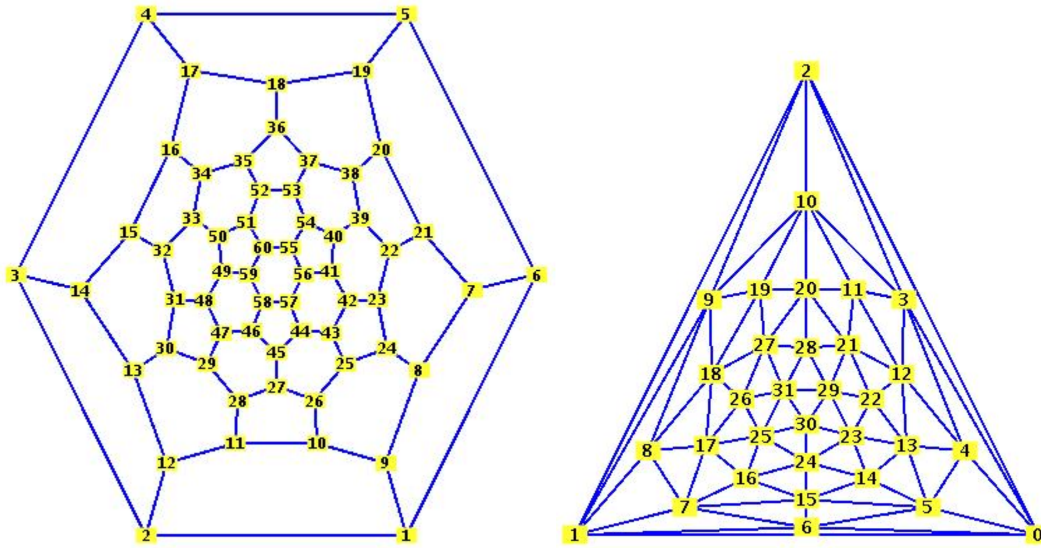


Fig. 7: The truncated icosahedron graph and its dual.

In [3], Haggard, Pearce and Royle report that they computed the Tutte polynomial for TI^* in one week on 150 computers. They used the VORDER-pull heuristic. Using the VORDER-push heuristic, and the vertex ordering as shown in the figure 7, we computed the Tutte polynomial for TI on a single core of a 2.66 Ghz Intel Core i7 desktop in under 14 minutes and 2.6 gigabytes, and for TI^* in under 18 minutes and 7.3 gigabytes. Notice that the vertexes of TI (and also TI^*) are numbered in concentric cycles. This

was the ordering that we input the graph from a picture. Notice that the vertex ordering is a short arc ordering. This is why we tried the short arc ordering.

4.4 Dense graphs.

Up to this point, the data shows that VORDER-push is much better than VORDER-pull. This, however, is not the case for dense graphs. In Table 6 we give data for the complete graphs K_n on n vertexes. VORDER-pull is clearly better than VORDER-push.

		VORDER-pull				VORDER-push			
n	m	time(s)	#calls	#ident	deg	time	#calls	#ident	deg
10	45	0.08	2519	1002	7.40	0.24	7448	2826	4.84
11	55	0.18	5075	2024	8.27	0.64	17178	6667	5.25
12	66	0.46	10191	4070	9.12	1.72	38940	15372	5.70
13	78	1.07	20427	8164	10.02	4.57	87070	34829	6.10
14	91	2.43	40903	16354	10.90	12.64	192544	77838	6.54
15	105	6.10	81859	32736	11.81	39.00	421922	172047	6.95
16	120	16.36	163775	65502	12.71	113.42	917540	376848	7.37

Tab. 6: Data for K_n for VORDER-pull and VORDER-push

4.5 Monitoring execution for large graphs.

For large graphs, the user of software for computing Tutte polynomials will need a way to know how far a large computation has progressed and how much memory has been consumed so that the user can stop the computation when it becomes obvious that it not going to terminate in a reasonable time. When the Tutte polynomial for a graph G of size n vertexes in the computation tree is computed for the first time, we display the additional time it took to compute $T(G)$ since the time it took to compute the Tutte polynomial for a graph of size $n-1$ for the first time, and the total space used after $T(G)$ is computed. The output for $n \geq 20$ for the truncated icosahedron is shown in Table 7.

n	time(s)	space	n	time	space	n	time	space	n	time(s)	space
21	0.08	0.564gb	31	2.45	0.564gb	41	21.42	0.564gb	51	107.66	1.361gb
22	0.11	0.564gb	32	0.02	0.564gb	42	26.54	0.564gb	52	87.20	1.587gb
23	0.20	0.564gb	33	4.35	0.564gb	43	0.03	0.564gb	53	47.92	1.740gb
24	0.01	0.564gb	34	0.02	0.564gb	44	7.08	0.564gb	54	59.78	1.894gb
25	0.19	0.564gb	35	0.88	0.564gb	45	29.09	0.564gb	55	74.56	2.065gb
26	0.01	0.564gb	36	0.02	0.564gb	46	34.32	0.648gb	56	2.28	2.083gb
27	0.92	0.564gb	37	8.51	0.564gb	47	0.03	0.648gb	57	96.12	2.335gb
28	0.01	0.564gb	38	0.02	0.564gb	48	64.73	0.854gb	58	26.18	2.393gb
29	1.29	0.564gb	39	14.66	0.564gb	49	48.11	0.986gb	59	45.46	2.529gb
30	0.01	0.564gb	40	0.02	0.564gb	50	0.10	0.989gb	60	9.00	2.546gb

Tab. 7: Trace of time and space for the truncated icosahedron. Total time 822.29 seconds.

5 Conclusion

We have presented a new edge selection heuristic that for sparse graphs, outperforms the previous heuristics considered by Haggard, Pearce and Royle in [7] by several orders of magnitude and which significantly increases the range of graphs that can be computed for what is an NP-hard problem. For some graphs, including grid graphs and the Petersen graphs $P(n,k)$ for $1 \leq k \leq 4$, our new heuristic automatically finds polynomial time constructions for the Tutte polynomial. At this point we only have a partial understanding of why and when VORDER-push is so effective. Graphs with large girth appear to be more difficult. It seems likely to us that further significant gains in efficiency will be made by refining the vertex ordering to further improve locality.

References

[1] Béla Bollobás. *Modern Graph Theory*. Springer Graduate Texts in Mathematics. No. **184**, 1998.

[2] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Springer-Verlag Graduate Texts in Mathematics. No. **138**, 1993.

[3] Gary Haggard, David Pearce, and Gordon Royle. Computing Tutte Polynomials. *Transactions on Mathematical Softare* **37**:3 (2011) article 24.

[4] J. Farr, M. Khatarinejad, S. Khodadad, M. Monagan. A Graph Theory Package for Maple. *Proceedings of the 2005 Maple Conference*, pp. 260–271, Maplesoft, 2005. Also available at `http://www.cecm.sfu.ca/CAG/papers/GTpaper.pdf`.

[5] Brendan Mckay. The nauty page. `http://cs.anu.edu.au/~bdm/nauty/`

[6] James Oxley and Dominic Welsh. Chromatic, Flow and Reliability Polynomials: The Complexity of their Coefficients. *J. Combinatorics, Probability and Computing* **11** (2002) 403−426.

[7] Pearce D.J., Haggard G., Royle G. Edge-Selection Heuristics for Computing Tutte Polynomials. *Chicago J. of Theoretical Computer Science*, Vol. 2010, (2010) article 6.

[8] Pemmaraju, S., and Skiena, S. *Computation Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*. Cambridge University Press, 2003.

[9] William Tutte. A contribution to the theory of chromatic polynomials. *Can. J. Math.* **6** (1954) 80−91.