Search space reduction through commitments in pathwidth computation: an experimental study

Yasuaki Kobayashi[1,a)] Keita Komuro[2,b)] Hisao Tamaki[2,c)]

Abstract: In designing an XP algorithm for pathwidth of digraphs, Tamaki introduced the notion of commitments and used them to reduce the search space with naively $O(n!)$ states to one with $n^{O(k)}$ states, where n is the number of vertices and k is the pathwidth of the given digraph. The goal of the current work is to evaluate the potential of commitments in heuristic algorithms for the pathwidth of *undirected graphs* that are aimed to work well in practice even for graphs with large pathwidth. We classify commitments by a simple parameter called depth. Through experiments performed on TreewidthLIB instances, we show that depth-1 commitments are extremely effective in reducing the search space and lead to a practical algorithm capable of computing the pathwidth of many instances for which the exact pathwidth was not previously known. On the other hand, we find that the additional search space reduction enabled by depth-d commitments with $2 \leq d \leq 10$ is limited and that there is little hope for effective heuristics based on commitments with such depth.

1. Introduction

Pathwidth [17] and treewidth [18] are among the central notions in the graph minor theory developed by Robertson and Seymour and have numerous applications in algorithm design. In particular, many NP-hard graph problems are fixed parameter tractable [11] when parameterized by pathwidth or treewidth: they have algorithms with running time $f(w)n^{O(1)}$, where n is the instance size, w is the pathwidth or treewidth of the instance graph, and f is a typically exponential function of w. Such algorithms are often practical when w is small and therefore computing these width parameters (and constructing associate graph decompositions) is of great practical importance. Theoretically, the problems of computing the treewidth and the pathwidth are both NP-hard [1], [12], although they are fixed parameter tractable admitting algorithms with running time linear in the graph size [4], [5].

Unfortunately, the running time of the fixed parameter algorithm given by [4], [5] has huge dependence on the width parameter and generally considered impractical. From the practical point of view, however, "finding a tree-decomposition of small width is far from hopeless" [8] and there has been a considerable amount of effort on turning this hope into reality [2], [8], [16]. For example, van den Broek and Bodlaender provide a benchmark suite TreewidthLIB [3] and lists known upper and lower bounds on the treewidth of most of the graph instances therein (see [2] for a method of computation used to derive such bounds). According to the description of the library [3], the instances there are collected with the criterion that finding tree-decompositions of small width is useful for solving some problem of real interest on those instances.

Compared with this situation for treewidth, the research effort on practically computing the pathwidth seems to be much more scarce. In particular, computing the pathwidth of TreewidthLIB instances would seem a valid goal of experimental research but no such report can be found in the literature.

The work reported in this paper significantly improves this situation. Our experimental results show that "finding a path-decomposition of small width is also far from hopeless" in practical situations. More specifically, our improved algorithms for pathwidth perform well on TreewidthLIB instances. Their performances are comparable to those of treewidth algorithms reported in the TreewidthLIB site. On some instances, they are even able to improve the best known treewidth upper bound by finding a path-decomposition, a special case of a tree-decomposition, with a smaller width.

Our basic algorithm is a backtrack search for a vertex sequence corresponding to an optimal path-decomposition, employing the standard memoization technique (see [10], for example). This basic algorithm may be viewed as a practical implementation of the standard vertex ordering approach for pathwidth [6] (see also [7] for a similar approach for treewidth): the dynamic programming algorithm of [6] stores in the table the solutions of all possible subproblems, while the corresponding table in our algorithm stores solutions of only those subproblems that are encountered in the backtrack search. To reduce the search space of this basic al-

[1] Gakushuin University, Toshima-ku, Japan 171-8588
[2] Meiji University, Kawasaki, Japan 214-8571
[a)] yasuaki.kobayashi@gakushuin.ac.jp
[b)] kouki-metal@cs.meiji.ac.jp
[c)] tamaki@cs.meiji.ac.jp

gorithm, we use the notion of *commitments* introduced by Tamaki [20]. He used commitments to obtain a theoretical result: an algorithm for the pathwidth of digraphs that runs in $n^{O(k)}$ time, where n is the number of vertices and k is the pathwidth of the given graph. This notion is also used in [14] to derive an $O(1.89^n)$ time algorithm for the pathwidth of directed and undirected graphs, which improves on the $O(1.9657^n)$ time algorithm of Suchan and Villanger [19] for the pathwidth of undirected graphs. The goal of this paper is to evaluate the potential of commitments in heuristic algorithms that are aimed at performing better in practice than guaranteed by the theoretical bounds.

A commitment occurs between a pair of search states S and T, where T is a descendant of S in the search tree. Under a certain condition, we discard all descendants of S but T, knowing that S leads to a successful computation if and only if T does (see Section 3 for more details of commitments). We define the *depth* of the commitment to be the depth of T in the search tree minus the depth of S. Some variants of depth-1 commitments can be found in theoretical work on pathwidth [14], [19]. Our experiments on TreewidthLIB show that depth-1 commitments are extremely effective in reducing the search space. Depth-1 commitments also have the advantage of being "cheap" in that they can be detected with very small computational effort. Indeed, the memoized backtrack search with depth-1 commitments performs well on small to medium-sized TreewidthLIB instances: out of the total of 162 instances therein with 300 or fewer vertices, our algorithm is successful in computing the exact pathwidth of 145 instances. This number compares favorably with 65, which is the number of instances, out of the same set, on which the exact treewidth is known. For fair comparisons, we need to note that those exact bounds for treewidth are rather old (obtained in 2007 or earlier). Despite continued efforts for improvements (see [9], for example), however, no new exact bounds have been reported in TreewidthLIB or elsewhere, to the best of the authors' knowledge.

Depth-d commitments for larger d are more costly. Naively, to find if there is a depth-d commitment from state S, we need an exhaustive search of depth d from S. In most practical situations, this cost is much greater than the gain we obtain from reducing the search space. There may be, however, some lower-cost heuristics that can be used to detect deep commitments not always but often enough to be useful. Our experiments on depth-d commitments for $d \geq 2$ give a ground for evaluating such heuristic potential of commitments. In the experiments, we assume an oracle that, given a search state S, detects a commitment from S within specified depth if one exists. If the reduction of the search space is found significant in these experiments, then it would be worthwhile to look for heuristic implementations of these oracles. Our experiments on TreewidthLIB instances show, however, that the space reduction effect of depth-d commitments for $2 \leq d \leq 10$ is rather limited and suggest that there is little hope of improvements by such heuristics over the algorithm with depth-1 commitments.

A byproduct of our experiments is a finding that the gap between the pathwidth and the treewidth may be dramatically smaller on practical instances than theoretically possible. See Subsection 4.5 for details.

The rest of this paper is organized as follows. Section 2 gives definitions and notation used in this paper. Section 3 describes the general principle of commitments together with necessary definitions. Section 4 describes our experiments. We conclude the paper with Section 5. A significantly more detailed version of this paper appears in [15].

2. Preliminaries

Let G be an undirected graph with vertex set $V(G)$ and edge set $E(G)$. For $v \in V(G)$, the set of neighbors of v is denoted by $N(v)$ and the number of them is denoted by $d(v)$. We extend this notation to sets: for $X \subseteq V(G)$, $N(X) = \bigcup_{v \in X} N(v) \setminus X$ and $d(X) = |N(X)|$.

A *path decomposition* of G is a sequence of subsets $(X_1, X_2, \ldots, X_t)$ of $V(G)$ satisfying the following conditions:

(1) $\bigcup_{1 \leq i \leq t} X_i = V(G)$,

(2) for each $\{u, v\} \in E(G)$, there is an index i such that $\{u, v\} \subseteq X_i$, and

(3) for each $v \in V(G)$, the set of indices i such that $v \in X_i$ forms a single interval.

The *width* of a path decomposition $(X_1, X_2, \ldots, X_t)$ is $\max_{1 \leq i \leq t} |X_i| - 1$. The *pathwidth* $\mathrm{pw}(G)$ of G is the smallest k such that G has a path decomposition of width k. In this paper, we use an alternative characterization of pathwidth, known as the *vertex separation number*, which we define below.

Let σ be a sequence of vertices. We assume all the sequences of vertices in this paper are without repetitions, i.e., all the elements in σ are distinct from each other. We denote by $V(\sigma)$ the set of vertices in σ and the length $|V(\sigma)|$ of σ by $|\sigma|$. When $V(\sigma) = V(G)$, we call σ a *permutation* of $V(G)$. Suppose σ and η are sequences with $V(\sigma) \cap V(\eta) = \emptyset$ and τ is the result of concatenating η after σ. Then, σ is a *prefix* of τ (a *proper prefix* if η is nonempty) and τ is an *extension* of σ (a *proper extension* if η is nonempty). For a non-negative integer k, σ is *k-feasible* if $d(V(\sigma')) \leq k$ for each prefix σ' of σ and is *strongly k-feasible* if there is a k-feasible extension of σ that is a permutation of $V(G)$. These notions are extended for sets: a set $S \subseteq V(G)$ is (strongly) k-feasible if there is some (strongly) k-feasible sequence σ such that $V(\sigma) = S$.

The *vertex separation number* of G is the minimum integer k such that $V(G)$ is k-feasible. It is known that the pathwidth of G equals the vertex separation number of G [13]. Our algorithm works on the vertex separation number, rather than directly on the pathwidth. We also note that our algorithm outputs a k-feasible sequence for $k = \mathrm{pw}(G)$, from which it is straightforward to construct a path decomposition of G of width k.

3. Commitments

Fix G and k. Our algorithm looks for k-feasible permutations working on the search tree whose nodes are k-feasible sets. To prune the search tree, we use the following notion of *commitments* introduced in [20].

Let σ be a k-feasible sequence. Following [20], we say that an extension τ of σ is a *k-committable extension* of σ if τ is a proper extension of σ, τ is k-feasible, and $d(X) \geq d(V(\tau))$ for

every X with $V(\sigma) \subseteq X \subseteq V(\tau)$. We also use the set version of this definition: T is a k-committable extension of S if there is a k-feasible sequence σ and a k-committable extension τ of σ such that $V(\sigma) = S$ and $V(\tau) = T$.

Lemma 1 ([20]) Suppose sequence σ is strongly k-feasible and τ is a k-committable extension of σ. Then τ is also strongly k-feasible.

Corollary 3.1 Suppose $S \subset V(G)$ is strongly k-feasible and T is a k-committable extension of S. Then T is also strongly k-feasible.

Thus, if a search-tree node S has a k-committable extension T then we may commit to T: all the descendants of S but T and its descendants may be ignored without losing the completeness of the search.

Although the use of commitments is a powerful pruning strategy leading to theoretical results in [14], [20], our preliminary experiments showed that the use of commitments in their full generality does not result in practically efficient algorithms. The reason of this is the huge overhead of finding k-committable extensions. A naive method of finding a k-committable extension of a given search node S is to do an exhaustive search through the descendants of S and the cost of this auxiliary search overweighs the gain in the reduction of the main search space.

However, in practical algorithms, it is not necessary to use a method that finds a k-committable extension whenever one exists. It is possible that a less costly heuristic method is effective if it succeeds in finding k-committable extensions often enough.

The goal of the current paper is not to evaluate the effectiveness of various heuristics in finding committable extensions but to evaluate the *potential* of the heuristic approach itself. For this goal, we assume oracles that, given a k-feasible set S, return either S itself or a k-committable extension of S. In this manner, we decouple the effect of search space reduction enabled by commitments from the cost of finding commitments. Only after we confirm significant reduction in the search space size, we may pursue efficient, but probably partial, implementations of those oracles. The details of the oracles we use in our experiments are described in the next section.

4. Experiments

4.1 Algorithm

The pseudocode listed below describes our recursive search procedure. For a fixed pair of graph G and positive integer k, it decides if the input vertex set S is strongly k-feasible or not. This procedure implements a standard backtrack search with memoization: it uses a table, called a *failure table*, which stores sets that are found not-strongly k-feasible in order to avoid duplicated search from such sets. It also uses an oracle f, an additional parameter given to the algorithm, for finding k-committable extensions: f can be an arbitrary function such that, for each k-feasible vertex set S, $f(S)$ is either a k-committable extension of S or is equal to S itself. We denote by f^* the limit of this function: for each S, $f^*(S) = f^m(S)$ where m is such that $f^m(S) = f^{m+1}(S)$.

The input for the initial call of this procedure is the empty set: the empty set is strongly k-feasible if and only if the vertex separation number of G is k or smaller.

Algorithm 1 Decides whether S is strongly k-feasible or not.

```
1: procedure STRONGLY-FEASIBLE(S)
2:     T ← f*(S).
3:     if T is in the failure table then
4:         return false
5:     end if
6:     if T = V(G) then
7:         return true
8:     end if
9:     for all v ∈ V \ T such that d(T ∪ {v}) ≤ k do
10:        if STRONGLY-FEASIBLE(T ∪ {v}, k) then
11:            return true
12:        end if
13:    end for
14:    Store T in the failure table.
15:    return false
16: end procedure
```

4.2 Oracles

In the experiments reported here, we use the following oracles. We say that a k-committable extension T of S is of *depth* d where $d = |T| - |S|$. For each non-negative integer d, we define function f_d as follows: $f_d(S)$ is a k-committable extension of the smallest depth d' with $1 \leq d' \leq d$ if one exists; $f_d(S) = S$ otherwise. For convenience, we are allowing d to be 0: f_0 is an "empty oracle" that, given S, always returns S itself.

In our experiments, the oracle f_d is implemented by an exhaustive search that costs $O(n^d)$ time. Recall that the purpose of the experiments is not to evaluate the overall efficiency of the search incorporating these oracles but to measure the search space reduction enabled by free uses of those oracles.

4.3 Search space reduction

The first part of our experiments measures the size of the search space in terms of the number of vertex sets stored in the failure table. Note that the number of successful vertex sets is at most the number of vertices of the graph and is negligibly smaller than that of unsuccessful ones in typical situations. We use the oracles f_0, f_1, f_5 and f_{10} in these experiments.

We have performed this experiment on 108 instances of TreewidthLIB. Remaining roughly 100 instances, for which running the algorithm with the empty oracle f_0 is already prohibitively time- or space-consuming, are excluded.

The parameter k given to the algorithm is exactly the pathwidth minus one. This is usually the last and the most time consuming step in the pathwidth computation: we know that the instance is $k + 1$-feasible from the previous steps and have to confirm the infeasibility for the current k.

Table 1 shows the size of the search space generated by our algorithm with oracles f_0, f_1, f_5, and f_{10}, in terms of the number of sets stored in the failure table, for some sample instances. Columns ltw and utw are the lower and upper bounds on the treewidth.The numbers for other instances show similar tendencies.

There are instances for which the reduction is small. For some of them, such as queen9_9 listed in Table 1, we know the reason: they are full of large cliques. For example, queen9_9, which

Table 1 Search space size of sample instances

instance	$\|V(G)\|$	$\|E(G)\|$	pw(G)	ltw	utw	f_0	f_1	f_5	f_{10}
1bx7	41	195	11	11	11	898	63	59	56
1g6x	52	405	19	19	19	1572	148	130	129
1bbz	57	543	25	25	25	5700	546	516	516
1a8o	64	536	25	23	25	33522	1350	1155	1149
queen9_9	81	1056	58	50	58	372842	256364	253980	253980
1aba	85	886	28	28	29	73212	1561	1456	1429
1d4t	102	1145	34	32	35	625556	8170	7159	7090
1f9m	109	1349	43	38	45	13299246	68317	59588	59062
ch150.tsp	150	432	13	8	15	663258	24236	22213	21814
u159.tsp	159	431	12	8	12	7801396	20231	17413	16434
kroA200.tsp	200	586	13	9	14	1371893	33807	32129	31870
tsp225.tsp	225	622	13	11	15	12079238	85824	80728	78654
diabetes	413	819	6	4	4	125888	95224	84718	84571

consists of 81 vertices, contains 20 cliques with 9 vertices.

See [15] for more details on the effect of search space reduction.

4.4 Performance of the algorithm with depth-1 commitments

The second part of our experiments focuses more on the actual performance of our algorithm with depth-1 commitments, in contrast to the first part which is concerned only with the size of the search space. Here, we are concerned with the actual running time, memory usage, and whether the algorithm is capable of solving each instance in a reasonable amount of time.

This part of the experiment is run on a machine with Intel Xeon E5606 (2.14GHz $\times$ 4) processor, 5.8GB RAM, and Ubuntu Linux. We do not use multi-thread for the execution of our algorithms. The heap space allocated for the Java VM is 1GB for all instances.

For each instance, we first compute an upper bound on the pathwidth by a simple greedy heuristic, set initial value of k to this upper bound, and repeat our algorithm for k-feasibility, decreasing k one by one until we find the instance not k-feasible: the final value of k is the pathwidth minus one. If this process is not completed in 30 minutes, we stop the execution and report the current upper bound on the pathwidth.

In the description of the experimental results below, when we say upper or lower bounds on the treewidth, they mean those bounds listed in TreewidthLIB. We also say that the exact treewidth is known for some instance, if the listed upper and lower bounds for the instance match.

We have run our algorithms on basically all instances in TreewidthLIB, for which the upper and lower bounds on the treewidth are listed. We have excluded, however, those instances that are the result of preprocessing, which simplifies the graph without changing the treewidth (but possibly changing the pathwidth). We have also excluded 3 weighted instances. We have selected 207 instances from TreewidthLIB by these criteria.

Table 2 summarizes the number of instances for which the pathwidth computation was successful, in the sense the exact pathwidth was obtained in 30 minutes, with oracle f_0 or f_1. Column 'total' shows the total number of tested instances in the specified range. The basic algorithm with the empty oracle is already effective for small instances. For instances with over 100 vertices, however, the algorithm with the depth-1 oracle clearly outperforms the basic algorithm.

Table 2 The number of instances for which the exact pathwidth computation is successful

$\|V(G)\|$	total	f_0	f_1	treewidth known
1 – 50	17	17	17	17
51 – 100	81	70	80	32
101 – 200	56	19	42	15
201 – 300	7	1	5	3
301 –	46	1	2	24

For comparison, Table 2 also lists the number of instances for which the exact treewidth is known. Note that this comparison is not meant for exactly measuring the relative difficulty of computing the exact pathwidth and the exact treewidth: the amount of time spent for the bounds listed in TreewidthLIB is typically smaller while the amount of time spent for unsuccessful efforts for improving the bounds is not known.

Table 3 lists more computational details of some selected instances. Columns ltw and utw are the lower and upper bounds on the treewidth; ipw is the pathwidth achieved by the initial greedy solution; pw_d, for $d = 0, 1$ is the upper bound on the pathwidth obtained by the iteration using Algorithm 1 with oracle f_d, which is the exact pathwidth unless the computation is aborted; t_d, for $d = 0, 1$ is the time, in seconds, consumed by oracle f_d. TLE means that the computation is aborted with the time limit of 30 minutes and MLE means that the computation is aborted because the heap space is exhausted.

From the results for larger instances in this list, we can see that the search space reduction effect of depth-1 commitments observed in the first part of the experiments indeed leads to dramatic improvements on the running time.

The largest two instances in this list are in contrast to each other: even though the exact pathwidth is not obtained for either of the instances, the upper bound obtained for u2319.tsp still improves the upper bound on the treewidth, while for BN_26 the computed upper bound on the pathwidth is far above the upper bound on the treewidth.

4.5 Comparisons between the pathwidth and the treewidth

Table 4 compares the pathwidth and the treewidth of the tested instances. For each range of the number of vertices, column 'total' shows the number of instances in the range for which the exact treewidth is known and moreover the exact pathwidth is obtained by our computation; other columns show the breakdown

Table 3 Computational details for some selected instances

G	$\|V(G)\|$	$\|E(G)\|$	ltw	utw	ipw	pw_0	pw_1	t_0	t_1
1g6x	52	405	19	19	23	19	19	0.13	0.12
1bbz	57	543	25	25	26	25	25	0.13	0.21
1a8o	64	536	23	25	27	25	25	0.24	0.46
1cc8	70	813	27	32	33	32	32	0.20	0.89
queen9_9	81	1056	50	58	61	58	58	2.5	94
1aba	85	886	28	29	30	28	28	0.56	0.96
1c5e	95	1148	33	36	38	34	34	1.3	2.4
1d4t	102	1145	32	35	41	34	34	4.4	4.3
1f9m	109	1349	38	45	47	43	43	251	60
bier127.tsp	127	368	8	15	22	15	15	39	4.4
ch150.tsp	150	432	8	15	17	13	13	7.5	1.5
u159.tsp	159	431	8	12	19	12	12	111	1.5
kroA200.tsp	200	586	9	14	25	13	13	25	2.3
tsp225.tsp	225	622	11	15	21	13	13	685	5.1
diabetes	413	819	4	4	31	6	6	5.6	2.6
celar06	100	350	11	11	18	11	11	TLE	5.1
graph01	100	358	21	24	38	23	23	TLE	1272
1bkb	131	1485	26	30	31	29	29	MLE	1.4
anna	138	493	12	12	24	15	14	TLE	TLE
a280.tsp	280	788	12	14	19	14	14	TLE	28
fpsol2.i.1	496	11654	66	66	79	75	67	TLE	323
u2319.tsp	2319	6869	41	56	70	50	47	TLE	TLE
BN_26	3025	14075	9	9	1005	103	103	TLE	TLE

Table 4 Classifying the instances based on the comparison between the pathwidth and the treewidth

$\|V(G)\|$	total	pw = tw	pw = tw + 1	pw = tw + 2	pw ≥ tw + 3
1 – 50	17	12	5(3, 3, 4, 9, 19)	0	0
51 – 100	32	31	1(9)	0	0
101 – 200	10	8	2(6, 9)	0	0
201 – 300	3	3	0	0	0
300 –	2	0	1(66)	1(4)	0

of this total number according to the difference between the pathwidth and the treewidth. For the columns with pw = tw + 1 and pw = tw + 2, the numbers in the parentheses show the treewidth of the individual instances counted in that column. Among all the tested instances, there are 16 instances with treewidth smaller than 10, 10 of which have 50 or fewer vertices. It is surprising that the two width parameters are identical for most of the instances in this category.

5. Conclusion

The results of our experiments show that depth-1 commitments are quite effective in reducing the search space and lead to dramatic improvements of the performance of the vertex ordering approach for pathwidth. On the other hand, the results on deeper commitments are negative. Even assuming the oracles that detect those commitments without any cost, the improvement over the depth-1 commitments would be slim. This suggests that looking for heuristics for detecting deeper commitments is probably not worth the effort. This conclusion, however, is not final, as experiments are done only for commitments of depth up to 10. There still remains a small possibility that commitments of much larger depth are useful in heuristic algorithms.

References

[1] Arnborg, S., Corneil, D., Proskurowski A.: Complexity of finding embeddings in a k-tree. SIAM Journal on Matrix Analysis and Applications 8(2), 277–284, (1987)

[2] Bachoore, E.H., Bodlaender, H.L.: A branch and bound algorithm for exact, upper, and lower bounds on treewidth, In: AAIM 2006, LNCS, vol. 4041, pp. 255–266. Springer, Heidelberg (2006)

[3] van den Broek, J., Bodlaender, H.L.: TreewidthLIB. `http://www.cs.uu.nl/research/projects/treewidthlib/`, accessed Feb 9 2014.

[4] Bodlaender, H.L., Kloks, T.: Efficient and constructive algorithms for the pathwidth and treewidth of graphs. Journal of Algorithms 21, 358–402 (1996)

[5] Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM Journal on Computing 25(6), 1305–1317 (1996)

[6] Bodlaender, H.L., Fomin, F.V., Koster, A.M.C.A., Kratsch, D., Thilikos, D.M.: A note on exact algorithms for vertex ordering problems on graphs. Theory of Computing Systems 50(3), 420-432 (2012)

[7] Bodlaender, H.L., Fomin, F.V., Koster, A.M.C.A., Kratsch, D., Thilikos, D.M.: On exact algorithms for treewidth. ACM Transactions on Algorithms 9(1): 12, 2012

[8] Bodlaender, H.L., Grigoriev, A., Koster, A.M.C.A.: Treewidth lower bounds with brambles. Algorithmica 51(1), 81–98 (2008)

[9] Bodlaender, H.L., Koster, A.M.C.A.: Treewidth Computations II. Lower Bounds. Information and Computtation, 209(7), 1103–1119, (2011).

[10] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. The MIT press, Boston (2001)

[11] Downey, R.G., Fellows, M.R.: Parameterized complexity. Springer, Berlin (1998)

[12] Kashiwabara, T., Fujisawa T.: NP-completeness of the problem of finding a minimum-clique-number interval graph containing a given graph as a subgraph. In: Proceedings of International Symposium on Circuits and Systems, pp. 657–660 (1979)

[13] Kinnersley, G.N.: The vertex separation number of a graph equals its path-width. Information Processing Letters 42(6), 345–350 (1992)

[14] Kitsunai, K., Kobayashi, Y., Komuro, K., Tamaki, H., Tano, T.: Computing directed pathwidth in $O(1.89^n)$ time. In: IPEC 2012, LNCS, vol. 7535, pp. 182–193. Springer, Heidelberg (2012)

[15] Kobayashi, Y., Komuro, K., Tamaki H.: Search Space Reduction through Commitments in Pathwidth Computation: An Experimental Study. In: SEA 2014 To appear

[16] Koster, A.M.C.A., Bodlaender, H.L., van Hoesel, S.P.: Treewidth: computational experiments. Electronic Notes in Discrete Mathematics 8, 54–57 (2001)

[17] Robertson, N., Seymour, P.D.: Graph minors. I. Excluding a forest. Journal of Combinatorial Theory, Series B 35(1), 39–61 (1983)

[18] Robertson, N., Seymour, P.D.: Graph minors. II. Algorithmic aspects

of tree-width. Journal of Algorithms 7(3), 309–322, (1984)
[19] Suchan, K., Villanger, Y.: Computing pathwidth faster than 2^n. In: IWPEC 2009, LNCS, vol. 5917, pp. 324–335. Springer, Heidelberg (2009)
[20] Tamaki, H.: A polynomial time algorithm for bounded directed pathwidth. In: WG 2011, LNCS, vol. 6986, pp. 331–342. Springer, Heidelberg (2011)