Implementing Minimum Cycle Basis algorithms

Kurt Mehlhorn and Dimitrios Michail

Max-Planck-Institut für Informatik, Saarbrücken, Germany
{mehlhorn,michail}@mpi-sb.mpg.de

Abstract. In this paper we consider the problem of computing a minimum cycle basis of an undirected graph $G = (V, E)$ with n vertices and m edges. We describe an efficient implementation of an $O(m^3 + mn^2 \log n)$ algorithm presented in [1]. For sparse graphs this is the currently best known algorithm. This algorithm's running time can be partitioned into two parts with time $O(m^3)$ and $O(m^2 n + mn^2 \log n)$ respectively. Our experimental findings imply that the true bottleneck of a sophisticated implementation is the $O(m^2 n + mn^2 \log n)$ part. A straightforward implementation would require $\Omega(nm)$ shortest path computations, thus we develop several heuristics in order to get a practical algorithm. Our experiments show that in random graphs our techniques result in a significant speedup.
Based on our experimental observations, we combine the two fundamentally different approaches to compute a minimum cycle basis used in [1, 2] and [3, 4], to obtain a new hybrid algorithm with running time $O(m^2 n^2)$. The hybrid algorithm is very efficient in practice for random dense unweighted graphs.
Finally, we compare these two algorithms with a number of previous implementations for finding a minimum cycle basis in an undirected graph.

1 Introduction

Let $G = (V, E)$ be an undirected graph. A *cycle* of G is any subgraph in which each vertex has even degree. Associated with each cycle is an *incidence vector* x, indexed on E, where $x_e = 1$ if e is an edge of C, $x_e = 0$ otherwise. The vector space over $GF(2)$ generated by the incidence vectors of cycles is called the *cycle space* of G. It is well-known that this vector space has dimension $N = m - n + \kappa$, where m is the number of edges, n is the number of vertices, and κ the number of connected components of G. A maximal set of linearly independent cycles is called a *cycle basis*.

The edges of G have non-negative weights. The weight of a cycle is the sum of the weights of its edges. The weight of a cycle basis is the sum of the weights of its cycles. We consider the problem of computing a cycle basis of minimum weight in a graph; we use the abbreviation MCB to refer to a minimum cycle basis.

The problem has been extensively studied, both in its general setting and in special classes of graphs. Its importance lies in its use as a preprocessing step in several algorithms. Such algorithms include diverse applications like electrical circuit theory [5], structural engineering [6] and periodic event scheduling [1].

The first polynomial time algorithm for the minimum cycle basis problem was given by Horton [3] with running time $O(m^3 n)$. de Pina [1] gave an $O(m^3 + mn^2 \log n)$ algorithm by using a different approach. Golynski and Horton [4] improved Horton's algorithm to $O(m^\omega n)$ by using fast matrix multiplication. It is presently known [7] that $\omega < 2.376$. Recently Berger et al. [8] gave another $O(m^3 + mn^2 \log n)$ algorithm by using similar ideas as de Pina. Finally, Kavitha et al. [2] improved de Pina's algorithm into $O(m^2 n + mn^2 \log n)$ again by using fast matrix multiplication. In the same paper a faster $1 + \epsilon$ approximation algorithm, for any $\epsilon > 0$, is presented.

In this paper we report our experimental findings from our implementation of the $O(m^3 + mn^2 \log n)$ algorithm presented in [1]. Our implementation uses LEDA [9]. We develop a set of heuristics which improve the best-case performance of the algorithm while maintaining its asymptotics. Finally, we consider a hybrid algorithm obtained by combining the two different approaches used in [1, 2] and [3, 4] with running time $O(m^2 n^2)$, and compare the implementations. The new algorithm is motivated by our need to reduce the cost of the shortest path computations. The resulting algorithm seems to be very efficient in practice for random dense unweighted graphs. Finally, we compare our implementations with previous implementations of minimum cycle basis algorithms [3, 8].

The paper is organized as follows. In Section 2 we briefly describe the algorithms.In Section 2.1 we describe our heuristics and in 2.2 we present our new algorithm. In Section 3 we give and discuss our experimental results.

2 Algorithms

Let $G(V, E)$ be an undirected graph with m edges and n vertices. Let $l : E \mapsto \mathbb{R}_{\geq 0}$ be a non-negative length function on the edges. Let κ be the number of connected components of G and let T be any spanning forest of G. Also let $e_1, \ldots, e_N$ be the edges of $G \setminus T$ in some arbitrary but fixed order. Note that $N = m - n + \kappa$ is exactly the dimension of the cycle space.

The algorithm [1] computes the cycles of an MCB and their *witnesses*. A witness S of a cycle C is a subset of $\{e_1, \ldots, e_N\}$ which will prove that C belongs to the MCB. We view these subsets in terms of their incidence vectors over $\{e_1, \ldots, e_m\}$. Hence, both cycles and witnesses are vectors in the space $\{0, 1\}^m$. $\langle C, S \rangle$ stands for the standard inner product of vectors C and S. Since we are at the field $GF(2)$ observe that $\langle C, S \rangle = 1$ if and only if the intersection of the two edge sets has odd cardinality. Finally, adding two vectors C and S in $GF(2)$ is the same as the symmetric difference of the two edge sets. Algorithm 1 gives a full description.

The algorithm in phase i has two parts, one is the computation of the cycle C_i and the second part is the update of the sets S_j for $j > i$. Note that updating the sets S_j for $j > i$ is nothing more than maintaining a basis $\{S_{i+1}, \ldots, S_N\}$ of the subspace orthogonal to $\{C_1, \ldots, C_i\}$.

Algorithm 1 Construct an MCB

```
Set $S_i = \{e_i\}$ for all $i = 1, \dots, N$.
for $i = 1$ to $N$ do
  Find $C_i$ as the shortest cycle in $G$ s.t $\langle C_i, S_i \rangle = 1$.
  for $j = i + 1$ to $N$ do
    if $\langle S_j, C_i \rangle = 1$ then
      $S_j = S_j + S_i$
    end if
  end for
end for
```

Computing the cycles Given S_i, it is easy to compute a shortest cycle C_i such that $\langle C_i, S_i \rangle = 1$ by reducing it to n shortest path computations in an appropriate graph G_i. The following construction is well-known.

G_i has two copies v^+ and v^- of each vertex $v \in V$. For each edge $e = (u, v) \in E$ do: if $e \notin S_i$, then add edges (u^+, v^+) and (u^-, v^-) to the edge set of G_i and assign their weights to be the same as e. If $e \in S_i$, then add edges (u^+, v^-) and (u^-, v^+) to the edge set of G_i and assign their weights to be the same as e. G_i can be visualized as 2 levels of G (the $+$ level and the $-$ level). Within each level, we have edges of $E \setminus S_i$. Between the levels we have the edges of S_i. Call G_i, the *signed* graph.

Any v^+ to v^- path p in G_i corresponds to a cycle in G by identifying edges in G_i with their corresponding edges in G. If an edge $e \in G$ occurs multiple times we include it if the number of occurrences of e modulo 2 is 1. Because we identify v^+ and v^- with v, the path in G resulting from p is a cycle C. Since we start from a positive vertex and end in a negative one, the cycle has to change sign an odd number of times and therefore uses an odd number of edges from S_i. In order to find a shortest cycle, we compute a shortest path from v^+ to v^- for all $v \in V$.

Running time In each phase we have the shortest path computations which take time $O(n(m + n \log n))$ and the update of the sets which take $O(m^2)$ time. We execute $O(m)$ phases and therefore the running time is $O(m^3 + m^2 n + mn^2 \log n)$.

2.1 Heuristic improvements

In this section we present several heuristics which can improve the running time substantially. All heuristics preserve the worst-case time and space bounds.

Compressed representation (H1) All vectors (sets S and cycles C) which are handled by the algorithm are in $\{0, 1\}^m$. Moreover, any operations performed are normal set operations. This allows us to use a compressed representation where each entry of these vectors is represented by a bit of an integer. This allows us to save up space and at the same time to perform 32 or 64 bitwise operations in parallel.

Upper bounding the shortest path (H2) During phase i we might perform up to n shortest path computations in order to compute the shortest cycle C_i with

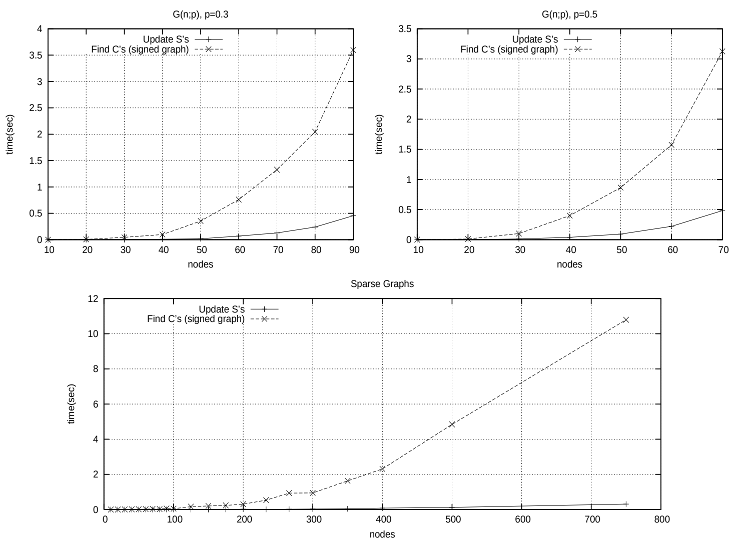

Fig. 1. Comparison of the time taken to update the sets S and the time taken to calculate the cycles on random weighted graphs, by Algorithm 1.

an odd intersection with the set S_i. Following similar observations of [10] we can use the shortest path found so far as an upper bound on the shortest path. This is implemented as follows; a node is only added in the priority queue of Dijkstra's implementation if its current distance is not more than our current upper bound.

Reducing the shortest path computations (H3) We come to the most important heuristic. In each of the N phases we are performing n shortest path computations. This results to $\Omega(mn)$ shortest path computations.

Let $S = \{e_1, e_2, \ldots, e_k\}$ be a *witness* at some point of the execution. We need to compute the shortest cycle C s.t $\langle C, S\rangle = 1$. We can reduce the number of shortest path computations based on the following observation.

Let $C_{\geq i}$ be the shortest cycle in G s.t $\langle C_{\geq i}, S\rangle = 1$, and $C_{\geq i} \cap \{e_1, \ldots, e_{i-1}\} = \emptyset$, and $e_i \in C_{\geq i}$. Then cycle C can be expressed as $C = \underset{i=1,\ldots,k}{min} C_{\geq i}$. We can compute $C_{\geq i}$ in the following way. We delete edges $\{e_1, \ldots, e_i\}$ from the graph G and the corresponding edges from the signed graph G_i. Let $e_i = (v, u) \in G$. Then we compute a shortest path in G_i from v^+ to u^+. The path computed will have an even number of edges from the set S, and together with e_i an odd number. Since we deleted edges $\{e_1, \ldots, e_i\}$ the resulting cycle does not contain any edges from $\{e_1, \ldots, e_{i-1}\}$.

Using the above observation we can compute each cycle in $O(kSP(n, m))$ time when $|S| = k < n$ and in $O(nSP(n, m))$ when $|S| \geq n$. Thus the running

Algorithm 2 Hybrid MCB algorithm

```
Ensure uniqueness of shortest path distances of $G$ ( lexicographically or by perturbation).
Construct superset (Horton set) $\mathcal{S}$ of MCB.
Set $S_i = \{e_i\}$ for all $i = 1, \dots, N$.
for $i = 1$ to $N$ do
  Find $C_i$ as the shortest cycle in $\mathcal{S}$ s.t $\langle C_i, S_i \rangle = 1$.
  for $j = i + 1$ to $N$ do
    if $\langle S_j, C_i \rangle = 1$ then
      $S_j = S_j + S_i$
    end if
  end for
end for
```

time for the cycles computations is equal to $SP(m, n) \cdot \sum_{i=1,\dots,N} min\{n, |S_i|\}$ where $SP(m, n)$ is the time to compute a single-source shortest path on an undirected weighted graph with m edges and n vertices.

2.2 A new hybrid algorithm

The first polynomial algorithm [3] developed, did not compute the cycles one by one but instead computed a superset of the MCB and then greedily extracted the MCB by Gaussian elimination. This superset contains $O(mn)$ cycles which are constructed in the following way.

For each vertex v and edge $e = (u, w)$, construct the cycle $C = SP(v, u) + SP(v, w) + (u, w)$ where $SP(a, b)$ is the shortest path from a to b. If these two shortest paths do not contain a vertex other than v in common then keep the cycle otherwise discard it. Let us call this set of cycles the *Horton set.* It was shown in [3] that the Horton set always contains an MCB. However, not every MCB is contained in the Horton set.

Based on the above and motivated by the need to reduce the cost of the shortest path computations we developed a new algorithm, which combines the two approaches. That is, compute the Horton set and extract the MCB not by using Gaussian elimination which would take time $O(m^3 n)$ but by using the orthogonal space of the cycle space as we did in Section 2. The Horton set contains an MCB but not necessarily all the cycles that belong to any MCB. We resolve this difficulty by ensuring uniqueness of the MCB. We ensure uniqueness by ensuring uniqueness of the shortest path distances on the graph (either by perturbation or by lexicographic ordering). After the preprocessing step, every cycle of the MCB will be contained in the Horton set and therefore we can query the superset for the cycles instead of the graph G. A succinct description can be found in Algorithm 2.

The above algorithm has worst case running time $O(m^2 n^2)$. This is because the Horton set contains at most mn cycles, we need to search for at most m cycles and each cycle contains at most n edges. The important property of this

n	m	N	$N(N-1)/2$	$max(\|S\|)$	$avg(\|S\|)$	$\# \langle S, C\rangle = 1$
sparse ($m \approx 2n$)						
10	19	10	45	4	2	8
104	208	108	5778	44	4	258
491	981	500	124750	226	7	2604
963	1925	985	484620	425	7	5469
2070	4139	2105	2214460	1051	13	20645
4441	8882	4525	10235550	2218	17	58186
$p = 0.3$						
10	13	4	6	2	2	2
25	90	66	2145	27	3	137
75	832	758	286903	370	6	3707
150	3352	3203	5128003	1535	9	22239
200	5970	5771	16649335	2849	10	49066
300	13455	13156	86533590	6398	10	116084
500	37425	36926	681746275	18688	14	455620
$p = 0.5$						
10	22	13	78	7	2	14
25	150	126	7875	57	4	363
75	1387	1313	861328	654	6	6282
150	5587	5438	14783203	2729	9	39292
200	9950	9751	47536125	4769	11	86386
300	22425	22126	244768875	10992	13	227548
500	62375	61876	1914288750	30983	15	837864

Table 1. Statistics about sets S sizes on sparse random graphs with $p = 4/n$ and dense random graphs for $p = 0.3$ and 0.5. Sets are considered during the whole execution of the algorithm. Column $\#\langle S, C\rangle = 1$ denotes the number of updates performed on the sets S. An upper bound on this is $N(N-1)/2$, which we actually use when bounding the algorithm's running time. Note that the average cardinality of S is very small compared to N although the maximum cardinality of some S is in $O(N)$.

algorithm is that the time to actually compute the cycles is only $O(n^2 m)$, which is by a factor of $\frac{m}{n} + \log n$ better than the $O(m^2 n + mn^2 \log n)$ time required by Algorithm 1. Together with the experimental observation that in general the linear independence step is not the bottleneck, we actually hope to have developed a very efficient algorithm.

3 Experiments

We perform several experiments in order to understand the running time of the algorithms using the previously presented heuristics. In order to understand the speedup obtained, especially from the use of the *H3* heuristic, we study in more detail the cardinalities of the sets S during the algorithm as well as how many operations are required in order to update these sets. We also compare the running times of Algorithms 1 and 2, with previous implementations.

All experiments are done using random sparse and dense graphs. All graphs were constructed using the $G(n;p)$ model, for $p = 4/n, 0.3, 0.5$ and 0.9. Our implementation uses LEDA [9]. All experiments were performed on a Pentium 1.7Ghz machine with 1 GB of memory, running GNU/Linux. We used the GNU g++ 3.3 compiler with the -O optimization flag. All other implementations, use the boost C++ libraries [11].

n	m	N	$max(\|S_i\|)$	$\lceil avg(\|S_i\|) \rceil$	$\|\{S_i : \|S_i\| < n\}\|$
sparse ($m \approx 2n$)					
10	19	10	4	2	10
104	208	108	39	5	108
491	981	498	246	13	498
963	1925	980	414	11	980
2070	4139	2108	1036	27	2108
4441	8882	4522	1781	33	4522
$p = 0.3$					
10	13	4	2	2	4
25	90	66	20	4	66
75	832	758	357	15	721
150	3352	3203	1534	18	3133
200	5970	5771	2822	29	5635
300	13455	13156	6607	32	12968
500	37425	36926	15965	39	36580
$p = 0.5$					
10	22	13	7	3	13
25	150	126	66	5	121
75	1387	1313	456	10	1276
150	5587	5438	2454	19	5338
200	9950	9751	4828	28	9601
300	22425	22126	10803	33	21875
500	62375	61876	30877	38	61483

Table 2. Statistics about sets S_i sizes on sparse random graphs with $p = 4/n$ and dense random graphs for $p = 0.3$ and 0.5, at the moment we calculate cycle C_i.

3.1 Updating S_i's

In this section we present experimental results which suggest that the dominating factor of the running time of Algorithm 1 (at least for random graphs) is not the time needed to update the sets S but the time to compute the cycles.

Note that the time to update the sets is $O(m^3)$ and the time to compute the cycles is $O(m^2n + mn^2 \log n)$, thus on sparse graphs this algorithm has the same running time $O(n^3 \log n)$ as the fastest known. The currently fastest algorithm [2] for the MCB problem has running time $O(m^2n + mn^2 \log n + m^\omega)$; the m^ω factor is dominated by the m^2n but we present it here in order to understand what type of operations the algorithm performs. This algorithm improves upon [1] w.r.t the time needed to update the sets S by using fast matrix multiplication techniques.

Although fast matrix multiplication can be practical for medium and large sized matrices, our experiments show that the time needed to update the sets S is a small fraction of the time needed to compute the cycles. Figure 1 presents a comparison of the required time to update the sets S_i and to calculate the cycles C_i by using the signed graph for random weighted graphs.

In order to get a better understanding of this fact, we performed several experiments. As it turns out, in practice, the average cardinality of the sets S is much less than N and moreover the number of times we actually perform set updates (if $\langle C_i, S_j \rangle = 1$) is much less than $N(N-1)/2$. Moreover, heuristic *H1* decreases the constant factor of the running time (for updating S's) substantially

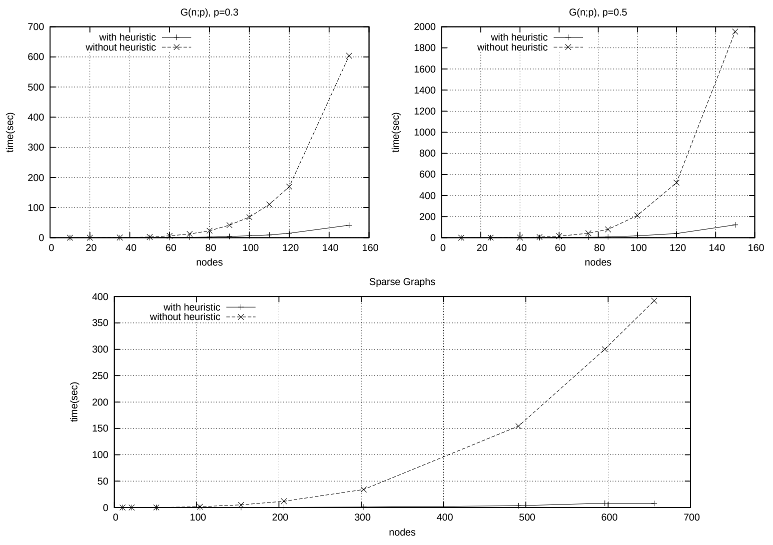

Fig. 2. Running times of Algorithm 1 with and without the *H3* heuristic. Without the heuristic the algorithm is forced to perform $\Omega(nm)$ shortest path computations.

by performing 32 or 64 operations in parallel. This constant factor decrease does not concern the shortest path computations. Table 1 summarizes our results.

3.2 Number of shortest path computations

Heuristic *H3* improves the best case of the algorithm, while maintaining at the same time the worst case. Instead of $\Omega(nm)$ shortest path computations we hope to perform much less. In Table 2 we study the sizes of the sets S_i for $i = 1, \dots, N$ used to calculate the cycles for sparse and dense graphs respectively.

In both sparse and dense graphs although the maximum set can have quite large cardinality, the average set size is much less than n. Moreover, in sparse graphs every set used has cardinality less than n. On dense graphs the sets with cardinality less than n are more than 95% percent. This implies a significant speedup due to the *H3* heuristic.

Figure 2 compares the running times of Algorithm 1 with and without the *H3* heuristic. As can easily be seen the improvement is more than a constant factor.

3.3 Running time

In this section we compare the various implementations for computing a minimum cycle basis. Except for Algorithms 1 (DP) and 2 (HYB) we include in the

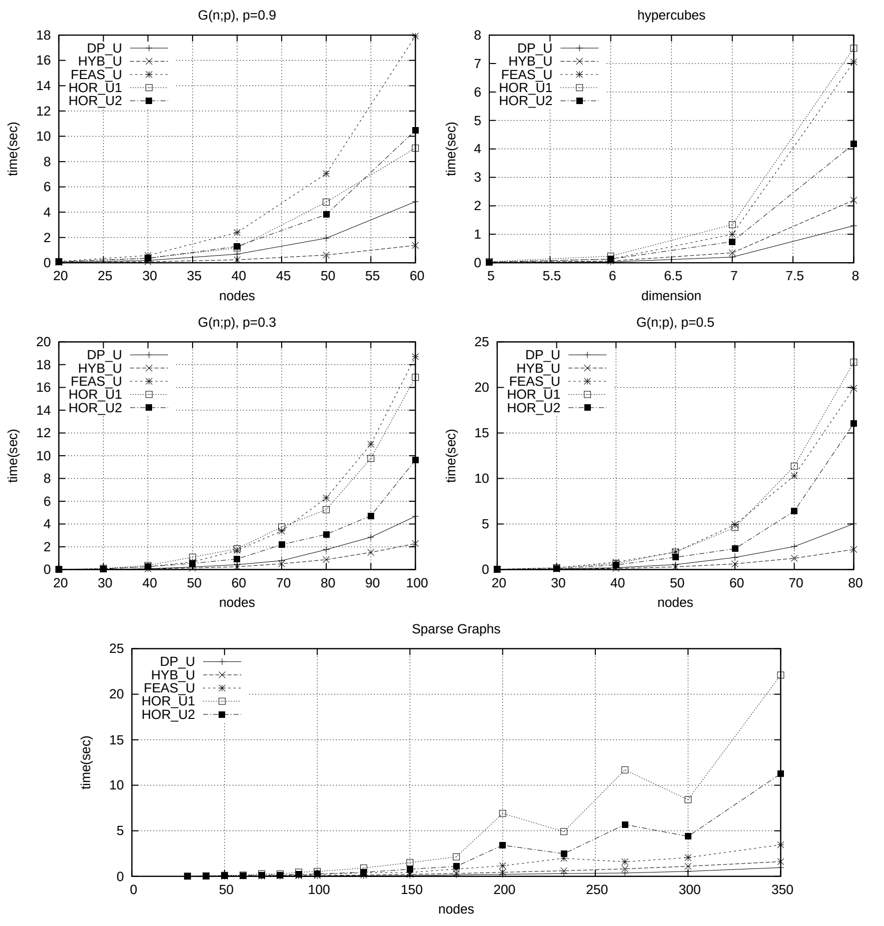

Fig. 3. Comparison of various algorithms for random *unweighted* graphs. Algorithm 1 is denoted as *DP_U* and Algorithm 2 as *HYB_U*. *HOR_U1* [12] and *HOR_U2* [13] are two different implementation of Horton's [3] algorithm. *FEAS_U* is an implementation [12] of an $O(m^3)$ algorithm described in [8].

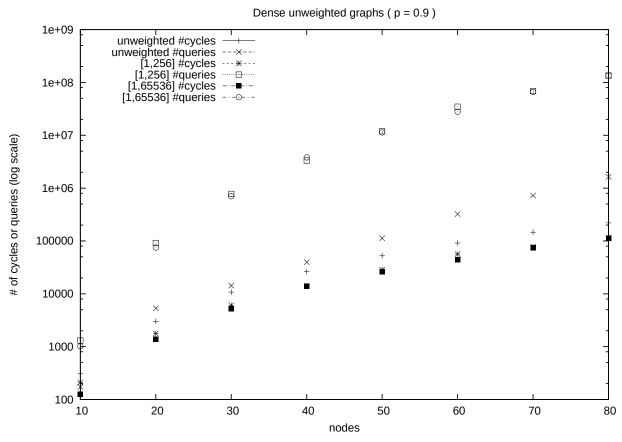

Fig. 4. Number of cycles in the Horton set (set with duplicates) and number of queries required in this set (set sorted by cycle weight) in order to extract the MCB for random dense graphs with random weights of different ranges. Each random graph is considered with three different edge weight ranges: (a) unweighted, (b) weights in $[1, 2^8]$, (c) weights in $[1, 2^{16}]$.

comparison two implementations [12, 13] (HOR) of Horton's algorithm with running time $O(m^3n)$ and an implementation [12] (FEAS) of the $O(m^3 + mn^2 \log n)$ algorithm presented in [8]. Algorithms 1 and 2 are implemented with compressed integer sets. Fast matrix multiplication [2, 4] can nicely improve many parts of these implementations with respect to the worst case complexity. We did not experiment with these versions of the algorithms.

The comparison of the running times is performed for three different type of undirected graphs: (a) random sparse graphs, where $m \approx 2n$, (b) random graphs from $G(n; p)$ with different density $p = 0.3, 0.5, 0.9$ and (c) hypercubes. Tests are performed for both weighted and unweighted graphs. In the case of weighted graphs the weight of an edge is an integer chosen independently at random from the uniform distribution in the range $[0 \ldots 2^{16}]$.

Figures 3 and 5 summarize the results of these comparisons. In the case of weighted graphs Algorithm 1 is definitely the winner. On the other hand in the case of dense unweighted graphs Algorithm 2 performs much better. As can be easily observed the differences on the running time of the implementations are rather small for sparse graphs. For dense graphs however, we observe a substantial difference in performance.

Dense unweighted graphs In the case of dense unweighted graphs, the hybrid algorithm performs better than the other algorithms. However, even on the exact same graph, the addition of weights changes the performance substantially. This change in performance is not due to the difference in size of the produced Horton

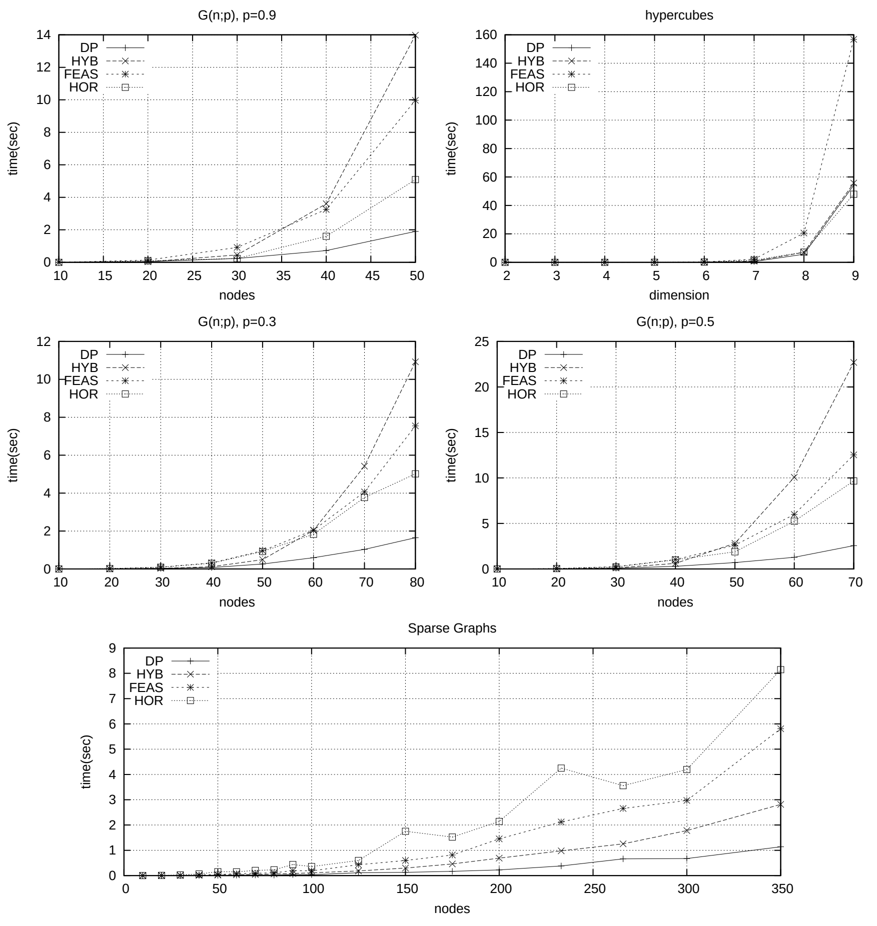

Fig. 5. Comparison of various algorithms for random *weighted* graphs. Algorithm 1 is denoted as *DP* and Algorithm 2 as *HYB*. *HOR* [12] is Horton's [3] algorithm. *FEAS* is an implementation [12] of an $O(m^3 + mn^2 \log n)$ algorithm described in [8].

set, between the unweighted and the weighted case, but due to the total number of queries that have to be performed in this set.

In the hybrid algorithm before computing the MCB, we sort the cycles of the Horton set. Then for each of the N phases, we *query* the Horton set from the least costly cycle to the most, until we find a cycle with an odd intersection with our current witness S. Figure 4 plots for dense graphs the number of cycles in the Horton set and the number of queries required in order to extract the MCB from this set. In the case of unweighted graphs, the number of queries is substantially smaller than in the case of weighted graphs. This is exactly the reason why the hybrid algorithm outperforms the others in unweighted dense graphs.

References

1. de Pina, J.: Applications of Shortest Path Methods. PhD thesis, University of Amsterdam, Netherlands (1995)
2. Kavitha, T., Mehlhorn, K., Michail, D., Paluch, K.E.: A faster algorithm for minimum cycle basis of graphs. In: 31st International Colloquium on Automata, Languages and Programming, Finland. (2004) 846–857
3. Horton, J.D.: A polynomial-time algorithm to find a shortest cycle basis of a graph. SIAM Journal of Computing **16** (1987) 359–366
4. Golynski, A., Horton, J.D.: A polynomial time algorithm to find the minimum cycle basis of a regular matroid. In: 8th Scandinavian Workshop on Algorithm Theory. (2002)
5. Chua, L.O., Chen, L.: On optimally sparse cycle and coboundary basis for a linear graph. IEEE Trans. Circuit Theory **CT-20** (1973) 495–503
6. Cassell, A.C., Henderson, J.C., Ramachandran, K.: Cycle bases of minimal measure for the structural analysis of skeletal structures by the flexibility method. Proc. Royal Society of London Series A **350** (1976) 61–70
7. Coppersmith, D., Winograd, S.: Matrix multiplications via arithmetic progressions. Journal of Symb. Comput. **9** (1990) 251–280
8. Berger, F., Gritzmann, P., , de Vries, S.: Minimum cycle basis for network graphs. Algorithmica **40** (2004) 51–62
9. Mehlhorn, K., Naher, S.: LEDA: A Platform for Combinatorial and Geometric Computing. Cambridge University Press (1999)
10. Bast, H., Mehlhorn, K., Schäfer, G.: A heuristic for dijkstra's algorithm with many targets and its use in weighted matching algorithms. Algorithmica **36** (2003) 75–88
11. Boost: C++ Libraries. (2001) http://www.boost.org.
12. Kreisbasenbibliothek: CyBaL. (2004) http://www-m9.ma.tum.de/dm/cycles/cybal.
13. Huber, M.: Implementation of algorithms for sparse cycle bases of graphs. (2002) http://www-m9.ma.tum.de/dm/cycles/mhuber.