MINIMUM CYCLE BASIS ALGORITHMS FOR THE CHEMISTRY DEVELOPMENT TOOLKIT

ULRICH BAUER

Abstract. The Chemistry Development Toolkit [CDK] is a comprehensive library for computational chemistry. It is used in several projects such as [Jmol] and [JChemPaint].

We present details about the implementation of several algorithms related to minimum cycles bases, also known as SSSR (smallest set of smallest rings). A minimum cycle basis is a set of minimum weight cycles which form a basis for the space of all cycles in a graph. The algorithms are described in [BGdV04a] and [BGdV04b].

1. Overview

The *Smallest Set of Smallest Rings* (SSSR), the chemical equivalent to a minimum cycle basis, plays an important role in computational chemistry. However, an efficient and exact algorithm for computing an SSSR was missing from the Chemistry Development Toolkit [CDK]; instead, a heuristic [Fig96] was used which was not efficient for larger molecules and did not always terminate or give the correct results. A new algorithm by Franziska Berger, Peter Gritzmann and Sven deVries [BGdV04a] achieves a time complexity of $\mathcal{O}(m^3)$, which is the fastest algorithm currently known for sparse graphs such as molecular graphs. This algorithm was implemented in the CDK to replace the old heuristic.

Moreover, several other algorithms based on minimum cycle bases, described in [BGdV04b] by the same authors, have been implemented to complement the SSSR and to give additional information about the structure of a molecule: the set of relevant and essential cycles (union and intersection of all minimum cycle bases of a graph), and interchangeability classes of the basis cycles according to the *interchangeability relation* introduced in [GLS00].

[CDK] is a library of classes and algorithms related to computational chemistry. It provides several features such as data types for atom, molecules and other chemical objects, supports input/output of various

file formats, rendering of molecules, database access, to name only a few.

Section 2 lists our most important improvements to the algorithms. In Section 3 we give a short overview of the algorithms that have been implemented in this project. Section 4 presents a few implementation-specific technical details. Section 5 show some results of running time tests of the algorithms.

2. New results of the project

The following are the most significant new results that have been found during the project. This list is especially interesting for those readers that are familiar with the underlying papers [BGdV04a] and [BGdV04b][1].

- A new algorithm for finding all optimal solutions to the single pair shortest path problem has been developed. The algorithm has a running time of $\mathcal{O}(m + k \cdot n)$ for unweighted graphs and $\mathcal{O}(m+n \log n+k \cdot n)$ for weighted graphs, where k is the number of solutions. This algorithm is described in 3.2.
- The algorithms for essential cycles and relevant cycles have been modified to use this new shortest-path algorithm instead of Eppstein's algorithm for the more general k *shortest paths* problem. These algorithms are described in 3.3.
- The theoretical complexity for the implemented algorithms on unweighted graphs has been improved by using this new algorithm and breadth-first search for finding shortest paths. The new running time bounds are $\mathcal{O}(m^3)$ for finding a minimum cycle basis, $\mathcal{O}(m^\omega + m^2 n+$ for finding the set of essential cycles (where ω denotes the matrix multiplication constant), $\mathcal{O}(m^\omega + m^2 n + n^2 m|\mathcal{R}|)$ for finding the set $\mathcal{R}$ of relevant cycles, and $\mathcal{O}(m^3 n)$ for computing the interchangeability classes of a cycle basis.
- A small optimisation from the minimum cycle basis algorithm has been applied to the algorithms for essential cycles and relevant cycles which had a dramatic impact on performance of the algorithm on certain instances. This is described in 3.3 as well.
- In the algorithm for computing interchangeability, to improve the performance of the computation, only non-essential cycles are checked, since essential cycles form singleton equivalence classes. Details can be found in 3.5.

[1]This is the first implementation of the algorithms presented in [BGdV04b].

3. The Algorithms

Algorithm 1 Computing a minimum cycle basis

Input: Undirected biconnected simple weighted graph G
Output: Minimum cycle basis $\mathcal{B}$ of G

1: Set $\mathcal{D}_0 := \emptyset, P := \emptyset$
2: **while** there exists an edge $e \in E$ that is not covered by $\mathcal{D}_0$ **do**
3: Compute a shortest cycle $C(e)$ through e
4: $\mathcal{D}_0 := \mathcal{D}_0 \cup \{C(e)\}$
5: $P := P \cup \{e\}$
6: **end while**
7: $r := |\mathcal{D}_0|$
8: Construct a fundamental tree basis $\mathcal{T}_0 = \{F_1, \ldots, F_{\mu-r}\}$ of the graph without the picked edges $(V, (E \setminus P))$
9: $\mathcal{B}_0 := \mathcal{D}_0 \cup \mathcal{T}_0$
10: **for** $i = 1$ to $(\mu - r)$ **do**
11: Find a shortest cycle C_i that is linearly independent from $\mathcal{B}_i \setminus \{F_i\}$ with Subroutine 2
12: **if** $\omega(C_i) < \omega(F_i)$ **then**
13: $\mathcal{B}_i := (\mathcal{B}_{i-1} \setminus \{F_i\}) \cup \{C_i\}$
14: **end if**
15: **end for**
16: Output $\mathcal{B} := \mathcal{B}_\mu$

3.1. **Minimizing a cycle basis.** The algorithm used for computing a minimum cycle basis is described in [Be04], section 5.3.5. This is basically the algorithm from [BGdV04a] with a preprocessing step.

The input graph is decomposed into its biconnected components using the algorithm described in 3.4. Then minimum cycle bases are computed for each biconnected component. The cycle basis of the whole graph is the union of the cycles in these bases.

To compute a minimum cycle basis of a biconnected component, the algorithm starts by computing a set $\mathcal{D}_0$ of linearly independent cycles which are guaranteed to be of minimum weight. To do this, we repeatedly pick an arbitrary edge e (that is not covered by the cycles already found) and find a shortest cycle through e. (Such a cycle exists, since the graph is biconnected). The edges chosen in this loop and the corresponding cycles are added to the edge and cycle lists, respectively, in reverse order to ensure that the resulting cycle-edge incidence matrix ist in upper triangular form (which is needed by the algorithm later).

After that, the set of chosen cycles is extended to a cycle basis by computing a *fundamental tree basis*, $\mathcal{T}$, on the graph without the already chosen edges edges. A fundamental tree basis is a cycle basis that is constructed as follows: first, compute a (minimum) spanning tree T. Then, for every non-tree edge e, create the cycle consisting of e and edges of T. Such a cycle is called *fundamental*; the set of all fundamental cycles is called *fundamental tree basis*.

A fundamental basis is generally not of minimum weight, therefore we minimize the fundamental cycles with the minimum cycle basis algorithm from [BGdV04a]. This algorithm basically replaces each fundamental cycle F_i with the shortest possible cycle C_i that is linearly independent to the other basis cycles. This is done with Subroutine 2 by constructing an auxiliary graph (with Subroutine 3) and generating shortest paths between certain vertex pairs. These paths correspond, by construction of the auxiliary graph, to shortest cycles in the main graph that are linearly independent to $\mathcal{B} \setminus F_i$, where $\mathcal{B}$ is the current basis.

Subroutine 2 Constructing a shortest feasible cycle C

Input: Index $i \geq 1$, Matrix A_{i-1}
Output: Shortest feasible cycle C

Form $A_{i-1}^{(i)}$ by removing row i from A_{i-1}
Construct kernel vector u by setting $u_i = 1$, $u_j = 0$ for $j > i$ and solving $A_{i-1}^{(i)} u = 0$
Form graph G_u as described in Subroutine 3
$\mathcal{C} := \emptyset$
for all vertices v incident to an edge e in G with $u_e = 1$ **do**
    Find a shortest v–v' path P_v in G_u with respect to ω_u
    Let $W(P_v)$ denote the closed walk in G obtained by replacing each vertex x' of P_v by x
    $\mathcal{C} := \mathcal{C} \cup W(P_v)$
end for
if $\min_{D \in \mathcal{C}} \omega(D) < \omega(F_i)$ **then**
    Let $C := \operatorname{argmin}_{D \in \mathcal{C}} \omega(D)$
    Form A_i by inserting C into row i of $A_{i-1}^{(i)}$ and performing Gaussian elimination on row i
else
    $C := F_i, A_i := A_{i-1}$
end if
return C

The auxiliary graph G_u is constructed from the graph G and the binary vector u as follows: for each vertex v in G, there are two vertices v and v' in the G_u. So the vertex set of G_u is partitioned into two sets V and V'. For each edge $e = (v, w)$ in G, there are two edges (of the same weight as e) in G_u: if $u_e = 1$, the edges (v', w) and (v', w) crossing the partition; otherwise, the edges (v, w) and (v', w'). Now, each path P_v from a vertex v to v' crosses the partition an odd number of times (has odd parity in regard to u). By replacing each vertex x' of P_v by x, we obtain a closed path in G. This might not be a cycle; however, it always contains an odd parity simple cycle as a subpath (see [BGdV04a]).

If u is a kernel vector of $A^{(i)}$ (the cycle-edge incidence matrix of $\mathcal{B} \setminus \{F_i\}$) and not a kernel vector of A, a shortest v–v' path in G_u corresponds to a shortest closed path in G with weight $\omega(F_i)$ containing v that is linearly independent to $\mathcal{B} \setminus \{F_i\}$. Iterating over all vertices incident to an edge e in G with $u_e = 1$ yields the shortest linearly independent closed path. Obviously, this path must be a simple cycle: else it would contain a strictly smaller linearly independent cycle as a subpath. Hence, the obtained path is the desired minimum basis cycle.

Subroutine 3 Constructing the auxiliary graph G_u

Input: Graph G, kernel vector u
Output: Auxiliary graph G_u

for all vertices v of G **do**
  Add two vertices v and v' to G_u
end for
for all edges $e = (x, y)$ of G **do**
  if $u_e = 1$ **then**
    Add the two edges $(x, y'), (x', y)$ to G_u
  else
    Add the two edges $(x, y), (x', y')$ to G_u
  end if
end for

Since molecular graphs are unweighted, we can use breadth-first search to find the shortest paths. This leads to an improvement in asymptotic running time, therefore we provide a short reanalysis of the time complexity.

Theorem 1. *A minimum cycle basis of an unweighted, biconnected graph $G = (V, E)$ can be computed using Algorithm 1 in time $\mathcal{O}(m^3)$.*

Proof. Computing the set $\mathcal{D}_0$ takes $\mathcal{O}(m^2)$ (see [Be04]). The fundamental tree basis can be constructed in time $\mathcal{O}(m+n)$. The loop is executed $\mathcal{O}(m)$ times, and each execution takes $\mathcal{O}(m^2)$ time:

The kernel vector u can be constructed in time $\mathcal{O}(m^2)$; construction of G_u takes linear time. Finding at most n shortest paths takes time $\mathcal{O}(n(m+n))$. Updating A_i requires linear time; gaussian elimination takes time $\mathcal{O}(m^2)$. □

3.2. **Finding all shortest paths between two vertices.** In the algorithms described later in 3.3 for computing relevant and essential cycles, the authors propose to use Eppstein's k-shortest-paths algorithm on an auxiliary graph to enumerate all relevant cycles. For our implementation, a new algorithm for enumerating only the optimal paths was developed within the present project instead.

Eppstein's algorithm lists all paths between two vertices, ordered by their weight. This algorithm requires a quite complicated heap construction to run in optimal time[2] $\mathcal{O}(m + n\log n + k)$, and in fact all available implementations are based on a simpler variant, running in $\mathcal{O}(m + n\log n + k\log k)$. However, even the "basic" variant of the algorithm is still quite tricky and deals with a number of auxiliary graphs, and only gives an implicit representation of the paths; this is something one might not want to mess with if it is not really necessary.[3]

So back to the actual problem: enumerating all relevant cycles in the graph G. This is done by enumerating k shortest (v, v')-paths in the auxiliary graph G_u, until the first path with greater weight is encountered. In fact, we are searching only for the minimum weight paths between two vertices, and no other paths. Since the main difficulty for the k-shortest-paths problem lies in enumerating the paths in order of their weight, which is not needed in our case, we can construct a much simpler algorithm which produces all minimum weight paths between a pair of vertices. Basically, we modify Dijkstra's algorithm (or breadth first search for unweighted graphs) to produce a shortest path digraph instead of a shortest path tree. The shortest path digraph is the union of all possible shortest path trees (directed).

Let v denote the vertex that is visited in the current iteration. Instead of adding only *one* of several possible edges coming out of v to the shortest path tree, we add *all* possible edges to the shortest path

[2]In fact, the total running time is $\mathcal{O}(m+n\log n+k^2n)$, since the algorithm gives only an implicit representation of the paths, and computing the paths explicitly takes $\mathcal{O}(k^2n)$.

[3][JM99] shows that Eppstein's algorithm has a significant overhead and is impractical even for quite large graphs.

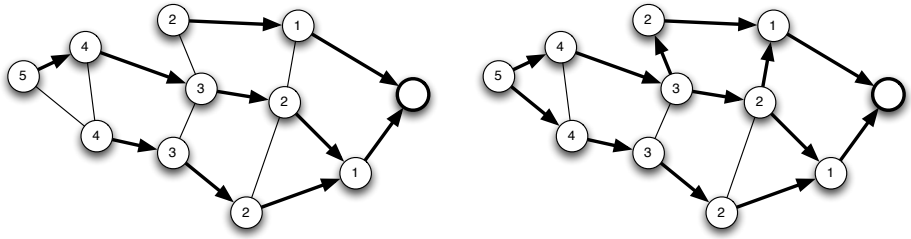

FIGURE 1. Shortest path tree and shortest path digraph

digraph, i.e. those edges (v, u) for which $d(v, t) = w(v, u)+d(u, t)$ holds. We can stop after we have reached the vertex s. This is quite important for our application, since we will often search for relatively short paths in large graphs.

Now we simply have to enumerate all possible s–t paths in this digraph. This can be achieved by starting from s, traversing each outgoing edge until t is reached, and returning the edges in the order of traversal (like a depth first search, only with the difference that each vertex can be visited several times). Assuming there are only positive edge weights in the input graph, this algorithm terminates and enumerates all optimal solutions of the shortest-path problem.

Theorem 2. *Algorithm 4 finds all k solutions of the single-pair-shortest-path problem in time $\mathcal{O}(m + n \log n + k \cdot n)$ for graphs with strictly positive edge weights, and $\mathcal{O}(m + k \cdot n)$ for unweighted graphs.*

Proof. To show that the algorithm correctly finds all shortest paths, we observe that the shortest path digraph is the union of all shortest v–t paths for all $v \in V$. This is done by induction over the vertices: when a new vertex v is visited, we add not only one arc to the digraph, but all arcs v–w that create a shortest path to t. Since the edge weights are strictly positive, any w has already been visited: if there exists a shortest v–t path containing w, then $d(w, t) < d(v, t)$, and so w must already have been visited. That means that any shortest v–t in G is present in the digraph.

On the other hand, every v–t path in the digraph is optimal, which can easily be seen from the fact that all arcs from v to another vertex are added in the iteration in which the v is marked as visited. Since only arcs that create a shortest path to t are added, by induction every v–t path is optimal.

The running time of the algorithm is $\mathcal{O}(m + n \log n)$ (creating the shortest path digraph using Fibonacci heaps) + $\mathcal{O}(k \cdot n)$ (enumerating

Algorithm 4 Enumerating all shortest s–t paths in a graph

Input: Graph G with strictly positive-weighted cycles, vertices u, v
Output: Enumeration of all shortest s–t paths in G

```
    {Generate the shortest path digraph}
 1: Mark t as visited
 2: while vertex s is not marked as visited do
 3:    Choose nearest unvisited vertex v (using Dijkstra or BFS)
 4:    Mark v as visited
 5:    for all vertices u adjacent to v do
 6:       if vertex u is already visited and d(v,t) = w(v,u) + d(u,t)
          then
 7:          add arc (v,u) to digraph
 8:       end if
 9:    end for
10: end while
    {Output all (s,t)-paths in the digraph}
11: Push the vertex s on the empty stack S
12: while S is not empty do
13:    Set vertex v := top(S) to the vertex on top of the stack
14:    if there is an unvisited arc e = (v,w) then
15:       Mark (v,w) as visited
16:       Push w on the stack S
17:       Set vertex v := w
18:    else
19:       if v = t then
20:          Output the path defined by following the vertices on the
             stack S (from bottom to top)
21:       end if
22:       Pop the vertex on top of the stack S
23:    end if
24: end while
```

all s–t paths) $= \mathcal{O}(m+n\log n+k\cdot n)$, where k is the number of shortest paths[4]. For unweighted graphs, the running time is $\mathcal{O}(m+n)+\mathcal{O}(k\cdot n)=\mathcal{O}(m+k\cdot n)$ □

The algorithm described finds all solutions to the single pair shortest path problem in a directed or undirected, weighted (with strictly

[4]This appears to be better than Eppstein's $\mathcal{O}(m+n\log n+k^2n)$, but this is only due to the fact that, in the k shortest path problem, non-optimal paths can visit a vertex more than once. If we use Eppstein's algorithm to find only all optimal paths, the running time is $\mathcal{O}(m+n\log n+k\cdot n)$ as well.

positive edge weights) or unweighted graph. Since the shortest path digraph is rooted at only one vertex (see Fig. 1), the same technique can also be used for the single source shortest path problem.

3.3. **Finding relevant and essential cycles.** The algorithms for finding relevant and essential cycles (union and intersection of all minimum cycle bases of a graph) are based on the algorithms described in [BGdV04b]. The main idea of the algorithm is to remove a cycle from the basis and then to enumerate all cycles that complete the remaining cycles to a basis, i.e. that are linearly independent to the remaining cycles. This yields all relevant cycles. If no other cycle can replace a certain basis cycle, this cycle is essential.

Algorithm 5 Compute the set of relevant cycles $\mathcal{R}$

Input: Undirected edge-weighted graph G
Output: The set of relevant cycles $\mathcal{R}$

```
1: Compute a minimum cycle basis, B, of G.
2: Let A be the cycle-edge incidence matrix of B, ordered such that
   the last m − μ columns correspond to the edges of a spanning tree
3: Compute the μ kernel vectors u_i of A^(i), i = 1, ..., μ.
4: for i = 1 to μ do
5:    for all vertices v incident to an edge e in G with (u_i)_e = 1 do
6:       Find all v–v' paths P_v in G_{u_i} with weight ω(a_i) using Algorithm
         4, and add the corresponding cycles in G to R
7:    end for
8: end for
9: Output R
```

One important modification to both algorithms has been proposed in the present project: similarly to the algorithm for constructing a minimum cycle basis, we only need to check for shortest cycles in the auxiliary graph that are guaranteed to replace a basis cycle (i.e., only for vertices v incident to an edge e with $u_e = 1$, find a shortest v–v' path). If we do not check for $u_e = 1$, searching for a shortest v–v' path in the auxiliary graph might take much more time, because we might do much more shortest path searches than actually necessary. If the graph has very small cycles and many vertices, this can lead to a dramatic increase of running time.

Since we know in advance the weight ω_i of the desired shortest paths, we can interrupt the construction of the shortest path digraph in Algorithm 4 when we reach a vertex v with $d(v, t) > \omega_i$. In this case, no shortest path exists.

Algorithm 6 Compute the set of essential cycles $\mathcal{E}$

Input: Undirected edge-weighted graph G
Output: The set of essential cycles $\mathcal{E}$

```
1: Compute a minimum cycle basis, B, of G.
2: Let A be the cycle-edge incidence matrix of B, ordered such that
   the last m − μ columns correspond to the edges of a spanning tree
3: Compute the μ kernel vectors u_i of A^(i), i = 1, ..., μ.
4: for i = 1 to μ do
5:    Set E := ∅.
6:    for all vertices v incident to an edge e in G with (u_i)_e = 1 do
7:       Find at most 3 v–v' paths P_v in G_{u_i} with weight ω(a_i) using
         Algorithm 4
8:       if each path P_v corresponds to a_i then
9:          Add a_i to E
10:      end if
11:   end for
12: end for
13: Output E
```

We now provide a reanalysis of the time complexity of both algorithms on unweighted graphs.

Theorem 3. *Given a minimum cycle basis of an unweighted, biconnected graph G, Algorithm 5 takes time $\mathcal{O}(m^\omega + m^2n + mn^2|\mathcal{R}|)$ to compute the set $\mathcal{R}$ of relevant cycles, where ω denotes the matrix multiplication constant. Algorithm 6 needs time $\mathcal{O}(m^\omega + m^2n)$ to compute the set of essential cycles.*

Proof. Computing the kernel vectors u_i takes $\mathcal{O}(m^\omega)$ (invert the first $\mu \times \mu$ part of A). The outer loop is executed $\mathcal{O}(m)$ times, and each iteration takes time $\mathcal{O}(mn+n^2|\mathcal{R}|)$: at most $2\cdot|\mathcal{R}|$ paths of weight $\omega(a_i)$ are found for each vertex v, taking time $\mathcal{O}(m+n|\mathcal{R}|)$; $\mathcal{O}(n)$ vertices are examined. Thus, the total time complexity is $\mathcal{O}(m^\omega + m^2n + mn^2|\mathcal{R}|)$.

For finding the essential cycles, we only need to compute at most 3 instead of $\mathcal{O}(|\mathcal{R}|)$ shortest paths for each v. Therefore, the complexity is $\mathcal{O}(m^\omega + m^2n + mn^2) = \mathcal{O}(m^\omega + m^2n)$.

□

3.4. **Finding the biconnected components of a graph.** The algorithm used to compute the biconnected components of a graph is taken from [GT02]. It is listed here as Algorithm 7. For a detailed

description and proof of correctness of the algorithm we refer to this book; we only provide a short explanation of the idea of the algorithm.

Algorithm 7 Computing biconnected components

Input: Undirected connected graph G
Output: Biconnected components of G

```
1: H empty graph
2: DFS traversal of G starting at an arbitrary vertex
3: insert each tree edge f of G as a vertex in H and mark f als
   "isolated"
4: for all v ∈ V(G) do
5:    let p(v) be the parent of v in the DFS
6: end for
7: for all v ∈ V(G) in order of the DFS do
8:    for all back edges e = (u, v) do
9:       insert e as vertex in H
10:      while u ≠ v do
11:         let f = (u, p(u))
12:         insert (e, f) as edge in H
13:         if f is isolated then
14:            mark f als connected
15:            u := p(u)
16:         else
17:            u := v
18:         end if
19:      end while
20:   end for
21: end for
22: return connected components of H
```

The basic idea of the algorithm is to consider an auxiliary graph H over the edges of the input graph G as vertices, where connected components correspond to biconnected components in G. This is achieved by doing a depth first search (DFS) on G and creating edges (e, f_i) in H between a back edge e in G and the tree edges $f_1, \ldots, f_k$ that form a cycle with e in G. Since two edges lie in the same biconnected component iff there is a cycle containing both edges, two vertices in the same connected component in H are edges in the same biconnected component in G (a detailed proof is given in [GT02]).

It suffices to only create a spanning forest of the graph H to find its connected components, thus the algorithm achieves a running time of $\mathcal{O}(n + m)$ instead of $\mathcal{O}(n \cdot m)$.

3.5. **Computing the interchangeability relation.** The *interchangeability* equivalence relation introduced in [GLS00] can be roughly explained as follows: two cycles C, C' are interchangeable iff they are of the same weight and C' can be expressed as a sum of C and some other (smaller or same size, linearly independent) relevant cycles.

The algorithm for computing the interchangeability equivalence relation consists of two steps: first, we check for all pairs of basis cycles a_i, a_j of same weight ψ if there is a cycle C that can replace both basis cycles a_i, a_j while maintaining a minimum cycle basis. This is done again with an auxiliary graph G_{u_i,u_j} similar to the one presented in 1. For every edge $e = (v, w) \in G$, there are four edges in G_{u_i,u_j}: $((v; x), (w; x \oplus (u_{i_e}, u_{j_e}))$, for every $x \in \{0, 1\}^2$. Now if we can find a $(v; (0, 0))$–$(v; (1, 1))$ path with weight ψ, then these two cycles are interchangeable.

Unfortunately, this is not a necessary condition for interchangeability (a counterexample is given in [BGdV04b]). We also have to check if for some strictly smaller basis cycle a_k there are two cycles C_i and C_j of weight ψ, such that C_i can replace both a_i, a_k and C_j can replace both a_j, a_k in a cycle basis. If this is the case, a_i and a_j are interchangeable as well.

Since the second step is much more time-consuming than the first, we first partition the basis into *pre-classes* that fulfil the first condition. We start with singleton classes and merge two classes if their representants fulfil the first condition. Since both conditions define equivalence relations and therefore are transitive, we only need to check single representants of each class. To save as much time as possible, we only check non-essential cycles for equivalence; since essential cycles are not interchangeable with any other cycle, we can omit these cycles. After that, we check the second condition only on representatives of each pre-class. The resulting partition of the basis cycles represents the equivalence classes.

The partition of the basis cycles is represented by an auxiliary graph H over the basis cycles where two cycles are connected by an edge if they are found to be in the same equivalence class. By transitivity, connected components of this graph correspond to equivalence classes.

4. Remarks on the Implementation

4.1. **General remarks.** One common problem in the implementation of graph algorithms is to define and access functions on graph vertices or graph edges. Basically, there are two options to implement these functions: either to save the function values internally together with

Algorithm 8 Computing interchangeability classes

Input: Undirected edge-weighted graph G, minimum cycle basis $\mathcal{B}$
Output: Partition of $\mathcal{B}$ into equivalence classes $\mathcal{W}_i, i = 1, \ldots, r$.

```
Obtain set of essential cycles, E with Algorithm 6, and kernel vectors, u_i, i = 1, ..., μ
Define a graph, H, that contains one vertex for each cycle a_i ∈ B
for all pairs (i, j), 1 ≤ i < j ≤ μ with ω(a_i) = ω(a_j), a_i and a_j not connected in H do
    if there is a shortest v–v' path P_v in G_{u_i,u_j} with ω(P_v) = ω(a_i) then
        add an edge (a_i, a_j) to H
    end if
end for
for all pairs (i, j), 1 ≤ i < j ≤ μ with ψ := ω(a_i) = ω(a_j), a_i and a_j not connected in H do
    for all basis cycles, a_k, of weight ω(a_k) < ψ do
        if there is a shortest v–v' paths P_v in G_{u_i,u_k} with ω(P_v) = ψ and a shortest w–w' paths P_w in G_{u_j,u_k} with ω(P_w) = ψ then
            add an edge (a_i, a_j) to H
        end if
    end for
end for
Return the connected components of H, W_i, i = 1, ..., r.
```

the edges or vertices (by subclassing and adding instance variables representing the function values), or to map edges or vertices externally to function values by dictionaries such as search trees or hash tables.

The internal approach has the advantage that it has better performance since accessing the function values can be done in constant time. However, it is not as flexible as the external approach since it requires the additional fields in the edge and vertex data classes, while the external approach has no additional requirements on the graph data structure. In this implementation, we always use hash tables for functions on edges or vertices, since this allows us to use the same graph in several algorithms without copying it into appropriate data structures first.

4.2. **Used libraries.** For graph data structures and simple algorithms, the open source library [JGraphT] has been used. It provides data structures for several types of graphs (directed and undirected, weighted

and unweighted, etc.) and algorithms (connected components, shortest path). Algorithms for finding biconnected components and finding all shortest s–t paths, as well as the cycle basis algorithms, have been implemented using this library.

An additional utility class, `MolecularGraphs`, is used to convert [CDK] Molecules to [JGraphT] graphs; in the class `SSSRFinder`, sets of cycles (implemented as subgraphs of [JGraphT] graphs) are converted back to [CDK] data structures.

4.3. **File listing.** The following Java classes have been added to the CDK source tree:

```
org.openscience.cdk.graph.BiconnectivityInspector
org.openscience.cdk.graph.MinimalPathIterator
org.openscience.cdk.graph.MolecularGraphs
org.openscience.cdk.ringsearch.SSSRFinder
org.openscience.cdk.ringsearch.cyclebasis.CycleBasis
org.openscience.cdk.ringsearch.cyclebasis.SimpleCycle
org.openscience.cdk.ringsearch.cyclebasis.SimpleCycleBasis
```

Additionally, unit tests for most of the classes have been provided:

```
org.openscience.cdk.test.graph.BiconnectivityInspectorTest
org.openscience.cdk.test.graph.MinimalPathIteratorTest
org.openscience.cdk.test.ringsearch.RingSearchTest
org.openscience.cdk.test.ringsearch.cyclebasis.CycleBasisTest
org.openscience.cdk.test.ringsearch.cyclebasis.SimpleCycleBasisTest
```

Details about the classes are presented in the following subsections.

4.4. **SimpleCycle.** Since we only have to deal with cycles that occur in a cycle basis, and these cycles are guaranteed to be simple, we do not need a more general class for cycles and can use some of the properties of a simple cycle in our methods.

The class `SimpleCycle` is implemented as a subclass of `Subgraph`, corresponding to the definition of a simple cycle: A *simple cycle* C in a graph G is a connected subgraph of G in which all vertices have degree two.

This property, however, is not checked by the class; it is assumed that the edges passed to the constructor `SimpleCycle(edges, g)` form a cycle. The constructor computes the vertices induces by the edges passed in the argument to form the subgraph.

The method `vertexList()` gives the vertices of the cycle in the order of a traversal. Since the cycle is simple, each vertex has degree two, and we can just at each vertex follow the edge we have not already seen immediately before.

4.5. **CycleBasis.** This class represents a cycle basis for a graph. We can speed up the computation of a cycle basis significantly if we decompose the graph into its biconnected components and perform the search on these components; this is done by the class `SimpleCycleBasis`. The class `SimpleCycleBasis` only puts together the results of these components. The cycle basis constructed by this class is guaranteed to be of minimum weight.

4.6. **SimpleCycleBasis.** This class represents a cycle basis for a biconnected graph. The actual algorithms are implemented in this class. The cycle basis constructed by this class is guaranteed to be of minimum weight as well.

The cycle basis is represented by a `List` of `Cycle`s. Since we need to operate on a cycle-edge incidence matrix in triangular form in the algorithms, we also need to remember a certain ordering of the edge in the graph; therefore the class also contains a `List` of the edges. A `HashMap` maps the edges back to the indices in the ordering.

4.6.1. *createMinimumCycleBase().* This method is called by the constructor. It computes a minimum cycle basis with Algorithm 1: it constructs a set of minimum basis cycles $\mathcal{D}_0$ and a set of fundamental cycles. The fundamental cycles are minimized with the private method `minimize()`.

4.6.2. *minimize().* This method implements the second part of Algorithm 1: minimizing an existing cycle basis. Since $\mathcal{D}_0$ already contains a set of minimum basis cycles, this method takes an argument that indicates at which cycle of the basis the minimization should start. It is assumed that the cycle-edge incidence matrix is in upper triangular form when this method is called.

We do not need to create the matrix $A_{i-1}^{(i)}$ (which is A_{i-1} without the ith row) explicitly; since we only look at the first $i-1$ rows of the matrix, we can simply use the matrix A_{i-1} instead.

The Gaussian elimination on the newly inserted row in A_i is very simple, since the matrix has upper triangular form: for each entry equal to 1 at position (i, j) with $j < i$, we add the jth row to the inserted row. After that, the entry at (i, j) is 0, since the entry (j, j) is always 1.

4.6.3. *essentialCycles().* The algorithm implemented in this method is described in 3.3 as Algorithm 6.

To obtain the kernel vectors u_i of the matrices $A^{(i)}$, we invert the $\mu \times \mu$ matrix consisting of the first μ columns of A. The rows of this

matrix give the first μ entries of the kernel vectors; the remaining $m-\mu$ entries are set to 0.

Our way to check if a basis cycle a_i is essential differs slightly from the one described in [BGdV04b]. The authors propose to count shortest v–v' paths and check if there are three or more; then the cycle is not essential. We do not count the paths, but instead iterate over all shortest v–v' paths until we find out that either the first path—and consequently all other shortest paths—are longer than the cycle a_i, or that a path corresponds to a different cycle (which means that a_i is not essential). If we do not find any path corresponding to a different cycle, we know that the cycle a_i is essential.

4.6.4. *relevantCycles().* This method implements Algorithm 5 described in 3.3; the implementation is very similar to `essentialCycles()`.

This method does not return a set, but a `HashMap` mapping each relevant cycle to the basis cycle that produced this cycle. Note that this mapping is not unique, since a relevant cycle can belong to several basis cycles that are in the same equivalence class.

4.6.5. *equivalenceClasses().* This is the implementation of Algorithm 8 described in 3.5 for computing interchangeability of basis cycles.

As explained in 3.5, the equivalence classes of the interchangeability relation are represented by the connected components of an auxiliary graph `h` over the basis cycles.

To quickly access all cycles of the same weight, the cycles are saved in an array, ordered by their weight. Since essential cycles are not exchangeable with any other cycle, we first search for the essential cycles and skip these cycles when iterating over the basis cycles and testing equivalence. We also skip the test if two cycles are already known to be in the same class, i.e. if they are in the same connected components of `h`.

4.6.6. *AuxiliaryGraph, AuxiliaryGraph2.* These classes implement the data structures and the generation of the auxiliary graphs G_u and G_{u_i,u_j} described in 3.3 and 3.5. These graphs take an input graph and one (two) binary kernel vectors u (u_i, u_j), each entry of a kernel vector corresponding to an edge of the input graph; the edge ordering is defined by the `List edgeList`. The vertex set of the input graph is mapped to two (four) sets of vertices of the auxiliary graph by `HashMap`s; the edges of the auxiliary graph are mapped back to edges of the input graph. The vertices of G_u are generated by appending `"-0"` and `"-1"` to the string representation of each vertex (for G_{u_i,u_j}, `"-00"`, `"-01"`, `"-10"`, and `"-11"`)

To optimize performance, the auxiliary graph is not constructed as a whole, but step by step when the graph is traversed. That way, we can save time if we search for small cycles in large graphs, since only the relevant part of the auxiliary graph is generated.

4.7. **MinimalPathIterator.** This class provides an iterator over all the shortest paths between two vertices, which are specified in the constructor.

Since an iterator is called again and again and has to return the next element, the state of the iterator has to be saved between the calls. For that reason, the `hasNext()` method performs the next step in the iteration, while two stacks save the state information. One stack contains the vertices visited on the path from source vertex to target vertex; the other stack contains the iterators that enumerate the outgoing edges of a vertex. This provides a simple way to generate all paths from source to target: at every vertex, every edge is followed until the target is reached.

Similar to the generation of the auxiliary graph, the vertices of the shortest path digraph are only created when they are needed, keeping it only as small as needed.

4.8. **BiconnectivityInspector.** This class provides the biconnected components of a graph. Its inspector interface borrows from JGraphT's `ConnectivityInspector` which computes the connected components of a graph.

The implementation is pretty straightforward. The DFS implementation from JGraphT `DepthFirstIterator` could not be used, since it does not provide the parent of a vertex in the traversal.

4.9. **MolecularGraphs.** This is an adapter that converts an instance of a CDK `Molecule` to a JGraphT `SimpleGraph`. Multiple bonds (double bonds, triple bonds) are not represented as multiple edges in a multigraph as proposed in [BGdV04a], but completely disregarded, conforming with the former SSSR implementation in the CDK.

4.10. **SSSRFinder.** This class is the interface to use for CDK developers. It provides the algorithms for molecules and replaces the old implementation of Figureas' heuristic which has been moved to `FigureasSSSRFinder`.

Moreover, the old interface, which provided only a static method, has been replaced with a new interface that caches a cycle basis for a molecule once it is computed. That way, the cycle basis does not have to be computed again if the other algorithms which rely on a cycle basis are performed.

5. Running time tests on the algorithms

Here we provide some results of running time test on the implemented algorithms. The tests have been performed on an Apple Powerbook G4 with 1 GHz processor speed and 1 GB of RAM. The runtime environment used was Java 1.4.2.

5.1. **Small molecular graphs.** In this test, we ran the algorithms for finding a cycle basis and finding the essential cycles on a number of medium-sized protein graphs. Since in all of these molecules besides Ethanonaphtalene, any cycle is an essential cycle, the algorithms for relevant cycles and for the equivalence classes have the same running time as the algorithm for the essential cycles and are not listed in the table. The running time for these algorithms on Ethanonaphtalene (14 ms and 15 ms) do not differ much from the running time for computing essential cycles (11 ms).

molecule	$\|V\|$	$\|E\|$	cycle basis	ess. cycles	(Figueras)
Ethanonaphtalene	12	14	1 ms	11 ms	3 ms
Perhydrophenalene	14	15	1 ms	14 ms	6 ms
Acridanone	24	27	2 ms	16 ms	5 ms
Diazaspirodecane	24	27	2 ms	8 ms	5 ms
Metoxepine	24	27	2 ms	16 ms	5 ms
Gelamine	24	29	3 ms	31 ms	5 ms
Apovincaminic acid	25	29	3 ms	35 ms	6 ms
Strictamine	26	30	3 ms	32 ms	4 ms

We do not see much here besides the fact that the algorithms do not have a measurable constant time overhead: small molecules give small running times. The minimum cycle basis algorithm performs roughly as well as Figueras' heuristic.

5.2. **Larger molecular graphs.** In the next test, we ran the same algorithms on a number of medium-sized protein graphs. Since in all of these graphs, any cycle is an essential cycle, the algorithms for relevant cycles and for the equivalence classes have the same running time as the algorithm for the essential cycles.

As we can see, the heuristic by Figueras is completely unusable for larger, sparse graphs such as molecular graphs of proteins. The implemented algorithm, however, is very fast on these graphs since many of the biconnected components consist only of a single cycle of weight 5 or 6.

$\|V\|$	$\|E\|$	cycle basis	essential cycles	cycle basis (Figueras)
1605	1622	0.154 s	0.027 s	159.213 s
2626	2644	0.345 s	0.030 s	810.694 s
3123	3149	0.406 s	0.044 s	1715.041 s
3944	3970	0.540 s	0.040 s	3800.991 s
4570	4626	0.734 s	0.128 s	
5020	5086	0.848 s	0.124 s	
5961	6034	1.026 s	0.117 s	
6290	6373	1.176 s	0.436 s	
6675	6760	1.261 s	0.525 s	
7422	7538	1.589 s	0.579 s	

5.3. **Fullerene C_{60}.** The C_{60} fullerene is a molecule composed entirely of carbon, whose structure resembles a soccerball. This is a very interesting instance for testing our algorithms, since the molecular graph is quite dense and biconnected.

	BGdV	Figureas
cycle basis	0.290 s	0.369 s
essential cycles	1.776 s	
relevant cycles	1.979 s	
interchangeability	11.373 s	

6. Conclusion

The goal of the present project was to create an efficient implementation of the algorithms related to minimum cycle bases presented in [BGdV04a] and [BGdV04b] for use in other projects. For the algorithms from the latter paper, this is the first implementation, so the results for these are especially interesting.

We showed that the minimum cycle basis algorithm, especially in the form presented in [Be04] with the additional preprocessing step, shows a performance that roughly compares with the previously used heuristic from [Fig96] for small molecules, while it can also be applied to larger molecules such as proteins, where the heuristic can not be applied anymore. So the running time of this algorithm is not only asymptotically the best currently available, but also very good especially for molecular graphs.

The other algorithms for essential and relevant cycles and for interchangeability perform very well, especially on the sparse graphs of proteins, but also on smaller and more dense molecular graphs. Unfortunately, another algorithm for comparison was not available. The good performance of the algorithms for essential and relevant cycles is

to a significant amount due to the optimizations developed during the project, especially the replacement of Eppstein's algorithm by a much simpler variant.

For applications in computational chemistry, the minimum cycle basis algorithm is the most interesting result of the project at the moment. The algorithm implemented in this project is already in use in several places, e.g. for layouting molecular graphs in [JChemPaint]. The other algorithms have not yet found use in other project, mainly due to the fact that these concepts are relatively new and not that many chemists are familiar with them.

References

[Be04] Franziska Berger, *Minimum Cycle Bases in Graphs*, PhD thesis, 2004

[BGdV04a] Franziska Berger, Peter Gritzmann, Sven de Vries, *Minimum Cycle Bases for Network Graphs*, to appear in Algorithmica

[BGdV04b] Franziska Berger, Peter Gritzmann, Sven de Vries, *Computing Cyclic Invariants for Molecular Graphs*, to be published

[CDK] The Chemistry Development Kit, Java utitility classes for ChemoInformatics and Computational chemistry, `http://cdk.sourceforge.net/`

[Fig96] John Figueras, *Ring Perception Using Breadth-First Search*, J. Chem. Inf. Comput Sci., 1996, 36, 986-991

[GLS00] P.M. Gleiss, J. Leydold, and P.F. Stadler, *Interchangeability of relevant cycles in graphs*, Electronic J. Comb., 7:#R16, 2000

[GT02] Michael T. Goodrich, Roberto Tamassia. *Algorithm Design: Foundations, Analysis, and Internet Examples*, John Wiley & Sons, Inc., 2002

[JM99] Víctor Jiménez, Andrés Marzal, *Computing the K Shortest Paths: a New Algorithm and an Experimental Comparison*, "Algorithm Engineering", J. S. Vitter and C. D. Zaroliagis (eds.), Springer-Verlag, Lecture Notes in Computer Science series, vol. 1668, pages 15–29, 1999. © Springer-Verlag. (Proceedings of the 3rd Int. Workshop on Algorithm Engineering, WAE'99, London, July 1999.)

[Jmol] Jmol, free, open source molecule viewer for students, educators, and researchers in chemistry and biochemistry, `http://jmol.sourceforge.net/`

[JChemPaint] JChemPaint, editor for 2D molecular structures, `http://jchempaint.sourceforge.net/`

[JGraphT] JGraphT, free Java graph library, `http://jgrapht.sourceforge.net/`