Algorithms for Three Versions of the Shortest Common Superstring Problem

Maxime Crochemore, Marek Cygan, Costas S. Iliopoulos, Marcin Kubica, Jakub Radoszewski, Wojciech Rytter, Tomasz Walen

Algorithms for Three Versions of the Shortest Common Superstring Problem

Maxime Crochemore[1,3], **Marek Cygan**[2], **Costas Iliopoulos**[1,4], **Marcin Kubica**[2], **Jakub Radoszewski**[2], **Wojciech Rytter**[2,5], and **Tomasz Waleń**[2]

[1] King's College London, London WC2R 2LS, UK
`maxime.crochemore@kcl.ac.uk, csi@dcs.kcl.ac.uk`
[2] Dept. of Mathematics, Computer Science and Mechanics, University of Warsaw, Warsaw, Poland
`[cygan,kubica,jrad,rytter,walen]@mimuw.edu.pl`
[3] Université Paris-Est, France
[4] Digital Ecosystems & Business Intelligence Institute, Curtin University of Technology, Perth WA 6845, Australia
[5] Dept. of Math. and Informatics, Copernicus University, Toruń, Poland

Abstract. The input to the Shortest Common Superstring (SCS) problem is a set S of k words of total length n. In the classical version the output is an explicit word $SCS(S)$ in which each $s \in S$ occurs at least once. In our paper we consider two versions with multiple occurrences, in which the input includes additional numbers (multiplicities), given in binary. Our output is the word $SCS(S)$ given implicitly in a compact form, since its real size could be exponential. We also consider a case when all input words are of length two, where our main algorithmic tool is a compact representation of Eulerian cycles in multigraphs. Due to exponential multiplicities of edges such cycles can be exponential and the compact representation is needed. Other tools used in our paper are a polynomial case of integer linear programming and a min-plus product of matrices.

1 Introduction

The algorithmic aspects of the SCS problem are thoroughly studied in theoretical as well as in practical computer science. In this paper we consider two variations of the SCS problem related to the number of occurrences of input words: Uniform Multiple Occurrences SCS (SUM-SCS) and Multiple Occurrences SCS Problem (MULTI-SCS). Our algorithms use several interesting combinatorial tools: Eulerian cycles in multigraphs, shortest paths via matrix multiplication and integer linear programming with a constant number of variables.

The SCS problem and its variations are studied in their own and also from the point of view of computational biologists. Recent advances, based either on sequencing by synthesis or on hybridisation and ligation, are producing millions of short reads overnight. An important problem with these technologies is how to efficiently and accurately map these short reads to a reference genome [12] that serves as a framework. The Genome Assembly Problem is as follows: given a large number of short DNA sequences (often called "fragments") generated by a sequencing machine like the ones mentioned above, put them back together to create a representation of the chromosome that sequences were taken from (usually a single organism). There are several software tools for this (called assemblers), to name a few: Celera Assembler [13] (mostly long reads), SSAKE [16], SHARCGS [5], SHRAP [14], Velvet [17]. When the problem of genome assembly arisen, computer scientists came up with the above abstraction of the SCS Problem, which they showed to be NP-hard [8], and developed several efficient approximation algorithms for it (see for example [1–3, 7, 15]). However, the SCS problem actually does not address the following issue of the biological Genome Assembly Problem — multiple occurrences of the fragments in the assembly. The shortest common superstring algorithms result in a shortest word but ignore repeated occurrences. In our paper we consider some variations of the problem in which we deal with multiple occurrences. First we consider a special case of SCS in which all fragments are of length 2. It happens that even this simple version is nontrivial in presence of multiplicities. Then a special case of constant number of words in S is considered. It is also nontrivial: we use sophisticated polynomial time algorithms for integer linear programs with constant number of variables.

Definitions of problems.
Let $S = \{s_1, s_2, \dots, s_k\}$ be the set of input words, $s_i \in \Sigma^*$. In all the problems defined below, we assume that S is a factor-free set, i.e. none of the words s_i is a factor of any other word from S. Let n denote the total length of all words in S. Let $\#occ(u, v)$ be the number of occurrences (as a factor) of the word u in the word v. We consider three problems:

SUM-SCS(k):
Given a positive integer m, find a shortest word u such that

$$\textstyle\sum_{i=1}^{k} \#occ(s_i, u) \geq m \ .$$

MULTI-SCS(k):
Given a sequence of non-negative integers $m_1, m_2, \dots, m_k$, find a shortest word u such that: $\#occ(s_i, u) \geq m_i$ for each $i = 1, 2, \dots, k$.

MULTI-SCS$_2$(k):
A special case of the **MULTI-SCS(k)** problem where all input words s_i are of length 2.

We assume, for simplicity, that the binary representation of each of the numbers in the input consists of $O(n)$ bits, i.e. $m = O(2^n)$ in **SUM-SCS(k)** and $m_i = O(2^n)$ in **MULTI-SCS(k)**. Moreover, we assume that such numbers fit in a single memory cell in the RAM model, and thus operations on such numbers can be performed in constant time (if this is not the case, one would need to multiply the time complexities of the algorithms presented here by a polynomial of the length of binary representation of numbers). Also, the total size of input data in each of the problems is $O(n)$.

By *finding* the SCS in each of the problems we mean computing its length and its compressed representation which is of size polynomial in n, that can be used to reconstruct the actual word in a straightforward manner in $O(\ell)$ time, where ℓ is the length of the word (this could be a context-free grammar, a regular expression etc).

2 Preliminaries

Let the overlap $ov(s,t)$ of two non-empty words, s and t, be the longest word y, such that $s = xy$ and $t = yz$ for some words $x \neq \varepsilon$ and z. We define $ov(s,t) = \varepsilon$ if $s = \varepsilon$ or $t = \varepsilon$. Also, let the prefix $pr(s,t)$ of s, w.r.t. t, be the prefix of s of length $|s| - |ov(s,t)|$ — therefore $s = pr(s,t)ov(s,t)$. For a given set $S = \{s_1, s_2, \ldots, s_k\}$ of words, the *prefix graph* of S is a directed graph with labeled and weighted edges defined as follows. The set of vertices is $\{0, 1, 2, \ldots, k, k+1\}$; vertices $1, 2, \ldots, k$ represent the words $s_1, s_2, \ldots, s_k$ and 0, $k+1$ are two additional vertices called *source* and *destination*, each of which corresponds to an empty word $s_0 = s_{k+1} = \varepsilon$. The edges are labeled with words, and their lengths (weights) are just the lengths of the labels. For all $0 \le i, j \le k+1$, $i \neq k+1$, $j \neq 0$, there is an edge (s_i, s_j) labeled with $pr(s_i, s_j)$. Note that, for a factor-free set S, the concatenation of labels of all edges in a path of the form $0 = v_1, v_2, v_3, \ldots, v_{p-1}, v_p = k+1$, i.e.

$$pr(s_{v_1}, s_{v_2})pr(s_{v_2}, s_{v_3}) \ldots pr(s_{v_{p-1}}, s_{v_p}) \ ,$$

represents a shortest word containing words $s_{v_2}, s_{v_3}, \ldots, s_{v_{p-1}}$ in that order. The prefix graph can easily be constructed in $O(k \cdot \sum_{i=1}^{k} |s_i|)$ time, using the prefix function from the Morris-Pratt algorithm [4]. However, this can also be done in the optimal time complexity $O(\sum_{i=1}^{k} |s_i| + k^2)$ — see [9].

Let A and B be matrices of size $(k+2) \times (k+2)$ containing non-negative numbers. The min-plus product $A \oplus B$ of these matrices is defined as:

$$(A \oplus B)[i,j] = \min\{A[i,q] + B[q,j] \ : \ q = 0, 1, \ldots, k+1\} \ .$$

We assume that the reader is familiar with a basic theory of formal languages and automata, see [10].

3 MULTI-SCS$_2(k)$ problem

First, let us investigate a variant of the **MULTI-SCS**(k) problem in which all input words s_i are of length 2. Note that in such a case $n = 2k$. It is a folklore knowledge that **MULTI-SCS**$_2(k)$ can be solved in polynomial time when all multiplicities m_i are equal to one. We prove a generalization of this result for the **MULTI-SCS**$_2(k)$ problem:

Theorem 1. *The* **MULTI-SCS**$_2(k)$ *problem can be solved in* $O(n^2)$ *time. The length of the shortest common superstring can be computed in* $O(n)$ *time, and its compact representation of size* $O(n^2)$ *can be computed in* $O(n^2)$ *time. (The real size of the output could be exponential.)*

Proof. Let us construct a multigraph $G = (V, E)$, such that each vertex corresponds to a letter of the alphabet Σ, and each edge corresponds to some word s_i — $(u, v) \in E$ if the word uv is an element of S. Each word s_i has a given multiplicity m_i, therefore we equip each edge $e \in E$ corresponding to s_i with its multiplicity $c(e) = m_i$. Using this representation the graph G has size $O(k)$ ($|E| = k$, and $|V| = O(k)$ if we remove isolated vertices). We refer to such an encoding as to a *compact multigraph representation.*

Any solution for the **MULTI-SCS**$_2(k)$ problem can be viewed as a path in some supergraph $G' = (V, E \cup E')$ of G, passing through each edge $e \in E$ at least $c(e)$ times. We are interested in finding the shortest solution, consequently we can reduce the problem to finding the smallest cardinality multiset of edges E' such that the multigraph $(V, E \cup E')$ has an Eulerian path. To simplify the description, we find the smallest multiset E' for which the graph $(V, E \cup E')$ has an Eulerian cycle, and then reduce the cycle to a path (if $E' \neq \emptyset$).

MULTI-SCS$_2(k)$ can be solved using the following algorithm:

```
1: construct the multigraph $G = (V, E)$
2: find the smallest cardinality multiset of edges $E'$ such that $G' =$
   $(V, E \cup E')$ has an Eulerian cycle
3: find an Eulerian cycle $C$ in $G'$
4: if $E' \neq \emptyset$ then
5:     return path $P$ obtained from $C$ by removing one edge from $E'$
6: else
7:     return $C$
```

As a consequence of the following Lemma 1, the smallest cardinality multiset E' can be computed in $O(|V| + |E|)$ time. The compact representation of an Eulerian cycle can be computed, by Lemma 2, in $O(|V| \cdot |E|)$ time. This gives us an $O(|V| \cdot |E|) = O(n^2)$ time algorithm for the **MULTI-SCS**$_2(k)$ problem. □

Lemma 1. *For a given compact representation of a directed multigraph* $G = (V, E)$*, there exists an* $O(|V| + |E|)$ *time algorithm for computing the smallest cardinality multiset of edges* E'*, such that the multigraph* $G' = (V, E \cup E')$ *has an Eulerian cycle.*

Proof. In the trivial case, when G already has an Eulerian cycle, we return $E' = \emptyset$. Let $C_1, C_2, \ldots, C_q$ ($C_i \subseteq V$) be connected components in the undirected version of the multigraph G. For each component we compute its demand D_i defined as follows:

$$D_i = \sum_{v \in C_i} \max(indeg(v) - outdeg(v), 0)$$

where $indeg(v)$ (resp. $outdeg(v)$) is the in (resp. out) degree of a vertex v. Let the demand of the vertex v be defined as $d(v) = |indeg(v) - outdeg(v)|$. Let $V^+(C_i)$ (resp. $V^-(C_i)$) be the set of vertices $v \in V(C_i)$, such that $indeg(v) > outdeg(v)$ (resp. $indeg(v) < outdeg(v)$). We describe an algorithm that computes the smallest cardinality multiset E' of the size:

$$|E'| = \sum_{i=1}^{q} \max(1, D_i) \ .$$

First, let us observe that an edge $e = (u, v) \in E'$ can have one of the following contributions:

- if $u, v \in C_i$ then e can decrease the demand D_i by at most 1,
- if $u \in C_i$ and $v \in C_j$ ($i \neq j$, with $D_i > 0$ and $D_j > 0$) then e merges C_i, C_j into a single component with demand at least $D_i + D_j - 1$,
- if $u \in C_i$ and $v \in C_j$ ($i \neq j$, with $D_i = 0$ or $D_j = 0$) then e merges C_i, C_j into a single component with demand at least $D_i + D_j$.

To construct the optimal set E', we use the following algorithm (see Fig. 1):

```
1: $E' = \emptyset$
2: first we connect all components:
3: **while** number of components $> 1$ **do**
4:     let $C_i, C_j$ be two different components
5:     if $D_i \neq 0$ then let $u$ be a vertex from the set $V^+(C_i)$, otherwise from $V(C_i)$
6:     if $D_j \neq 0$ then let $v$ be a vertex from the set $V^-(C_j)$, otherwise from $V(C_j)$
7:     add to $E'$ a single edge $(u, v)$ (with $c((u, v)) = 1$)
8: from now on we have only one component $C$ in $G$, we reduce its demand to 0:
9: **while** $V^+(C) \neq \emptyset$ **do**
10:    let $v^+ \in V^+(C)$, $v^- \in V^-(C)$
11:    add to $E'$ an edge $(v^+, v^-)$ with multiplicity $\min(d(v^+), d(v^-))$
12: **return** $E'$
```

Observe that if a component C_i admits $D_i > 0$ it means that both sets $V^-(C_i)$ and $V^+(C_i)$ are nonempty. In the first phase we add exactly $q - 1$ edges to E'. In the second phase we reduce the total demand to 0, each iteration of the **while** loop reduces the demand of at least one vertex to 0. Hence we add at most $O(|V|)$ different edges in the second phase. If we store E' using a compact representation, its size is $O(|V| + |E|)$ and the above algorithm computes it in such time complexity. $\square$

Lemma 2. *For a given compact representation of a directed Eulerian multigraph $G = (V, E)$, there exists an $O(|V| \cdot |E|)$ time algorithm for computing the representation of an Eulerian cycle in G.*

Proof. The Eulerian cycle can be of exponential size, therefore we construct its compressed representation. Such a representation is an expression of the form

$$\pi = w_1^{p_1} w_2^{p_2} \dots w_\ell^{p_\ell} \ ,$$

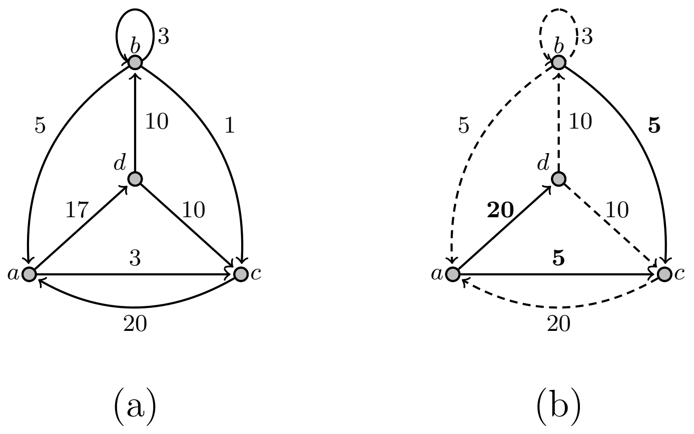

Fig. 1. (a) Multigraph $G = (V, E)$ for a set of words $S = \{ad, ac, ba, bb, bc, ca, db, dc\}$ with a sequence of multiplicities $(m_i)_{i=1}^{8} = (17, 3, 5, 3, 1, 20, 10, 10)$. It consists of a single component C with $V^+(C) = \{a, b\}$ and $V^-(C) = \{c, d\}$. (b) Eulerian multigraph $G' = (V, E \cup E')$ obtained from G by adding the following minimal multiset of edges: $E' = \{(a, d) \cdot 3,\ (a, c) \cdot 2,\ (b, c) \cdot 4\}$

where w_i is a path in G, and p_i is a non-negative integer. We say that an occurrence of $v \in w_i$ is *free* if $p_i = 1$. The algorithm is a slight modification of the standard algorithm for computing Eulerian cycles in graphs, see Fig. 2 and 3.

```
Function EulerianCycle($G$) {assume $G$ is an Eulerian multigraph}
 1: find any simple cycle $C$ in $G$
 2: let $c_{\min} = \min\{c(e) : e \in C\}$
 3: let $\pi = C^1\ C^{c_{\min}-1}$ { so that each $v \in V(C)$ has a free occurrence }
 4: for all $e \in C$ do
 5:     decrease $c(e)$ by $c_{\min}$, if $c(e) = 0$ remove $e$ from $E(G)$
 6: for all strongly connected components $W_i$ of $G$ do
 7:     let $v_i$ be any common vertex of $V(C)$ and $V(W_i)$
 8:     let $\pi_i$ = EulerianCycle($W_i$)
 9:     insert cycle $\pi_i$ to $\pi$ after some free occurrence of vertex $v_i$
10: return $\pi$
```

We can find a simple cycle in the Eulerian graph G in $O(|V|)$ time by going forward until we get to an already visited vertex. Each cycle removes at least one edge from E, so the algorithm investigates at most $|E|$ cycles. A simple implementation of lines 6-9 yields $O(|V| \cdot |E|)$ time complexity per recursive step, however, with a careful approach (G not decomposed to W_i explicitly, π implemented as a doubly-linked list) one can obtain $O(|V|)$ time complexity of these lines. Thus the time complexity and the total size of the representation π is $O(|V| \cdot |E|)$. □

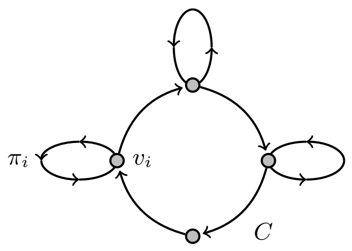

Fig. 2. Construction of an Eulerian cycle by algorithm EulerianCycle(G)

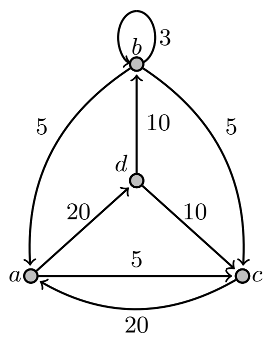

Fig. 3. Eulerian multigraph obtained for a set of words $S = \{ad, ac, ba, bb, bc, ca, db, dc\}$ with a sequence of multiplicities $(m_i)_{i=1}^{8} = (20, 5, 5, 3, 5, 20, 10, 10)$. The compressed representation of an Eulerian cycle can have the following form: $(a{\to}d{\to}b)(b{\to}b)^3(b{\to}a)(a{\to}d{\to}b{\to}a)^4(a{\to}c{\to}a)^5(a{\to}d{\to}b{\to}c{\to}a)^5(a{\to}d{\to}c{\to}a)^{10}$, which corresponds to a word $adb(b)^3a(dba)^4(ca)^5(dbca)^5(dca)^{10}$

4 MULTI-SCS(k) problem for $k = O(1)$

Let us consider a prefix graph G of $S = \{s_1, s_2, \ldots, s_k\}$. In order to solve the general **MULTI-SCS(k)** problem, it suffices to find the shortest path π from 0 to $k+1$ in G that passes through each vertex $i \in V(G)$, for $1 \le i \le k$, at least m_i times. We assume that $k = O(1)$.

Let us treat G as a (deterministic) finite automaton A: 0 is its start state, $k+1$ is its accept state, and an edge from i to j in G is identified by a triple $(i, j, |pr(s_i, s_j)|)$ which represents its starting and ending vertex and its length. Let $\Gamma \subseteq \{0, \ldots, k{+}1\} \times \{0, \ldots, k{+}1\} \times (\mathbb{Z}_+ \cup \{0\})$ be the set of triples identifying all edges of G. Each path from 0 to $k+1$ corresponds to a word (from Γ^*) in the language accepted by A.

Let $\alpha(A)$ be a regular expression corresponding to the language accepted by A — its size is $O(1)$ and it can be computed in $O(1)$ time [10] (recall that $k = O(1)$).

Definition 1. *We call two words $u, v \in \Gamma^*$* commutatively equivalent *(notation: $u \approx v$) if for any $a \in \Gamma$, $\#occ(a, u) = \#occ(a, v)$. We call two regular languages*

L_1, L_2 commutatively equivalent *(notation: $L_1 \approx L_2$) if for each word $u \in L_1$ ($u \in L_2$) there exists a word $v \in L_2$ ($v \in L_1$) such that $u \approx v$.*

Lemma 3. *The regular expression $\alpha(A)$ can be transformed in $O(1)$ time into a regular expression $\beta(A)$ such that:*

$$\mathcal{L}(\beta(A)) \subseteq \mathcal{L}(\alpha(A)) \text{ and } \mathcal{L}(\beta(A)) \approx \mathcal{L}(\alpha(A)) \tag{1}$$

and $\beta(A)$ is in the following normal form:

$$\beta(A) = B_1 + B_2 + \ldots + B_k$$

where $B_i = C_{i,1}C_{i,2}\ldots C_{i,l_i}$ and each $C_{i,j}$ is:

- *either $a \in \Gamma$,*
- *or $(a_1a_2\ldots a_p)^*$, where $a_r \in \Gamma$.*

Proof. The proof contains an algorithm for computing $\beta(A)$ in $O(1)$ time.

In the first step we repetively use the following transformations in every possible part of $\alpha(A)$ until all Kleene's stars contain only concatenation of letters from Γ (all letters in the transformations denote regular expressions):

$$\big(\gamma(\delta+\sigma)\rho\big)^* \rightarrow (\gamma\delta\rho)^*(\gamma\sigma\rho)^* \tag{2}$$

$$(\gamma\delta^*\sigma)^* \rightarrow \big(\gamma\delta^*\sigma(\gamma\sigma)^*\big) + \varepsilon\ . \tag{3}$$

Since then, it suffices to repetively use the following transformation to obtain the required normal form:

$$\gamma(\delta+\sigma)\rho \rightarrow (\gamma\delta\rho) + (\gamma\sigma\rho)\ . \tag{4}$$

It is easy to check that each of the transformations (2)–(4) changes the regular expression into another regular expression such that the language defined by the latter is a commutatively equivalent sublanguage of the language defined by the former. □

Let us notice that, due to the conditions (1), from our point of view $\beta(A)$ may serve instead of $\alpha(A)$ — for each path generated by the expression $\alpha(A)$ there exists a path generated by $\beta(A)$ such that the multisets of edges visited in both paths are exactly the same (thus the paths are of the same length).

To compute the result for $\beta(A)$, we process each B_i (see Lemma 3) separately and return the minimum of values computed. When computing the result (the shortest path generated by it that visits each vertex an appropriate number of times) for a given B_i, the only choices we might have are in those $C_{i,j}$'s that are built using Kleene's star. For each of them we introduce a single integer variable x_j that is used to denote the number of times we take the given fragment of the expression in the path we are to construct. For a given set of values of variables x_j, it is easy to compute, for each vertex y of the graph, how many times it is visited in the word, representing a path, generated by B_i:

$$\#(y, B_i) = \sum_{j=1}^{l_i} \#(y, C_{i,j})$$

and what is the total length of the word:

$$len(B_i) = \sum_{j=1}^{l_i} len(C_{i,j}) \ .$$

If $C_{i,j} = a$, for $a = (v, w, \ell) \in \varGamma$, then

$$\#(y, C_{i,j}) = \delta_{wy} \qquad len(C_{i,j}) = \ell$$

and if $C_{i,j} = (a_1 a_2 \dots a_p)^*$, where $a_r \in \varGamma$ for $1 \le r \le p$, then

$$\#(y, C_{i,j}) = x_j \cdot \sum_{r=1}^{p} \#(y, a_r) \qquad len(C_{i,j}) = x_j \cdot \sum_{r=1}^{p} len(a_r) \ .$$

Here δ_{xy} denotes the Kronecker delta: $\delta_{x,x} = 1$, and $\delta_{x,y} = 0$ for $x \neq y$.

If values of the variables are not fixed, we can treat $\#(y, B_i)$ and $len(B_i)$ as (linear) expressions over those variables. Our goal is, therefore, to minimize the value of $len(B_i)$ under the following constraints on variables $x_j \in \mathbb{Z}$:

$$\begin{aligned} x_j &\ge 0 \\ \#(y, B_i) &\ge m_y \quad \text{for} \quad y = 1, 2, \dots, k \ . \end{aligned}$$

But this is exactly an integer linear programming problem (see Example 1). For a fixed number of variables and constraints it can be solved in polynomial time in the length of the input, see Lenstra's paper [11], and even in linear time in terms of the maximum encoding length of a coefficient, see Eisenbrand's paper [6].

Example 1. Assume that $S = \{s_1, s_2\}$, $m_1 = 2010$, $m_2 = 30$. Note that the set of symbols in the regular expressions $\alpha(A)$ and $\beta(A)$ is $\varGamma \subseteq \{0, \dots, 3\} \times \{0, \dots, 3\} \times (\mathbb{Z}_+ \cup \{0\})$. Let

$$B_i = (0,1,0)(1,1,7)^*(1,2,3)\big((2,1,2)(1,1,7)(1,2,3)\big)^*(2,3,5)$$

be a part of the expression $\beta(A)$ for which we are to compute the shortest path satisfying the occurrence conditions.

Observe that we can interpret our task as a graph problem. We are given a directed graph having a form of a directed path (a backbone) with disjoint cycles attached, in which vertices are labeled with indices from the set $\{0, \dots, 3\}$ and edges are labeled with lengths, as in Fig. 4. In this graph we need to find the length of the shortest path from the vertex labeled 0 to the vertex labeled $k + 1 = 3$ visiting at least $m_1 = 2010$ 1-labeled vertices and at least $m_2 = 30$ 2-labeled vertices.

In the integer program we introduce two variables x_1, x_2 that uniquely determine a word generated by B_i:

$$(0,1,0)(1,1,7)^{x_1}(1,2,3)\big((2,1,2)(1,1,7)(1,2,3)\big)^{x_2}(2,3,5) \ .$$

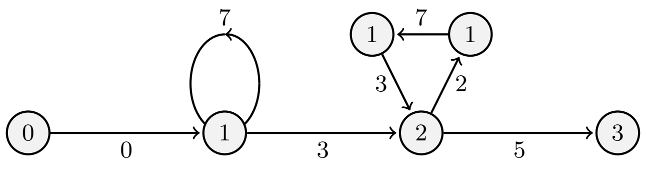

Fig. 4. Labeled graph corresponding to B_i from Example 1. The variables x_1 and x_2 from the integer program correspond to the number of times the loop $(1 \rightarrow 1)$ and the cycle $(2 \rightarrow 1 \rightarrow 1 \rightarrow 2)$ are traversed in the shortest path

The integer program for this example looks as follows:

$$\begin{aligned} x_1, x_2 &\geq 0 \\ 1 + x_1 + 2x_2 = \#(1, B_i) &\geq m_1 = 2010 \\ 1 + x_2 = \#(2, B_i) &\geq m_2 = 30 \end{aligned}$$

and we are minimizing the expression

$$len(B_i) = 0 + 7x_1 + 3 + 12x_2 + 5 \ .$$

To recompute the actual SCS, we choose the one B_i that attains the globally smallest value of $len(B_i)$. Note that solving the integer program gives us the values of variables x_j, from which we can restore the corresponding word v generated by the regular expression B_i simply by inserting the values of x_j instead of Kleene's stars, resulting in a polynomial representation of the shortest common superstring.

Theorem 2. MULTI-SCS(k) *can be solved in $O(poly(n))$ time for $k = O(1)$.*

5 SUM-SCS(k) problem

Let G be the prefix graph of $S = \{s_1, s_2, \ldots, s_k\}$. We are looking for the shortest word containing m occurrences of words from S. Recall that such a word corresponds to the shortest path in G from the source to the destination, traversing $m+1$ edges. Let M be the adjacency matrix of G. The length of the shortest path from the vertex 0 to the vertex $k+1$ passing through $m+1$ edges equals $M^{m+1}[0, k+1]$, where M^{m+1} is the $(m+1)^{th}$ power of M w.r.t. the min-plus product. M^{m+1} can be computed in $O(k^3 \log m)$ time by repeated squaring, i.e. using identities:

$$M^{2p} = (M^p)^2 \qquad\qquad M^{2p+1} = M \oplus M^{2p} \ .$$

Having computed the described matrices, we can also construct a representation of SCS of size $O(poly(n))$ by a context-free grammar. The set of terminals is Σ. For each of the matrices M^p that we compute, we create an auxiliary matrix K_p containing distinct non-terminals of the grammar. If M^p (for $p > 1$) is computed in the above algorithm using M^a and M^b then we add the following production from $K_p[i,j]$:

$$K_p[i,j] \Rightarrow K_a[i,q]K_b[q,j] \qquad \text{where} \quad M^p[i,j] = M^a[i,q] + M^b[q,j] \ .$$

The production from $K_1[i,j]$ is defined as:

$$K_1[i,j] \Rightarrow pr(s_i, s_j) \ .$$

The starting symbol of the grammar is $K_{m+1}[0, k+1]$.

Clearly, this representation uses $O(k^2 \log m)$ memory and the only word generated by this grammar is the requested SCS. Hence, we obtain the following theorem:

Theorem 3. *The* **SUM-SCS(**k**)** *problem can be solved in* $O(n + k^3 \log m)$ *time and* $O(n + k^2 \log m)$ *memory.*

References

1. C. Armen and C. Stein. A 2 2/3-approximation algorithm for the shortest superstring problem. In D. S. Hirschberg and E. W. Myers, editors, *CPM*, volume 1075 of *Lecture Notes in Computer Science*, pages 87–101. Springer, 1996.
2. A. Blum, T. Jiang, M. Li, J. Tromp, and M. Yannakakis. Linear approximation of shortest superstrings. *J. ACM*, 41(4):630–647, 1994.
3. D. Breslauer, T. Jiang, and Z. Jiang. Rotations of periodic strings and short superstrings. *Journal of Algorithms*, 24(2):340–353, 1997.
4. M. Crochemore and W. Rytter. *Jewels of Stringology.* World Scientific Publishing Company, 2002.
5. J. C. Dohm, C. Lottaz, T. Borodina, and H. Himmelbauer. SHARCGS, a fast and highly accurate short-read assembly algorithm for de novo genomic sequencing. *Genome research*, 17(11):1697–1706, November 2007.
6. F. Eisenbrand. Fast integer programming in fixed dimension. In G. D. Battista and U. Zwick, editors, *ESA*, volume 2832 of *Lecture Notes in Computer Science*, pages 196–207. Springer, 2003.
7. J. Gallant, D. Maier, and J. A. Storer. On finding minimal length superstrings. *J. Comput. Syst. Sci.*, 20(1):50–58, 1980.
8. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., New York, NY, USA, 1979.
9. D. Gusfield, G. M. Landau, and B. Schieber. An efficient algorithm for the all pairs suffix-prefix problem. *Inf. Process. Lett.*, 41(4):181–185, 1992.
10. J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979.
11. H. W. Lenstra, Jr. Integer programming with a fixed number of variables. *Mathematics of Operations Research*, 8(4):538–548, 1983.
12. H. Li, J. Ruan, and R. Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome Research*, 18(11):1851–1858, November 2008.

13. E. W. Myers et al. A whole-genome assembly of drosophila. *Science*, 287(5461):2196–2204, Mar. 2000.
14. A. Sundquist, M. Ronaghi, H. Tang, P. Pevzner, and S. Batzoglou. Whole-genome sequencing and assembly with high-throughput, short-read technologies. *PLoS ONE*, 2(5):e484, 2007.
15. J. Tarhio and E. Ukkonen. A greedy approximation algorithm for constructing shortest common superstrings. *Theor. Comput. Sci.*, 57(1):131–145, 1988.
16. R. L. Warren, G. G. Sutton, S. J. Jones, and R. A. Holt. Assembling millions of short DNA sequences using SSAKE. *Bioinformatics*, 23(4):500–501, February 2007.
17. D. R. Zerbino and E. Birney. Velvet: algorithms for de novo short read assembly using de Bruijn graphs. *Genome research*, 18(5):821–829, May 2008.