INVERSE MONOIDS PRESENTED BY A SINGLE SPARSE RELATOR

by

Steven P. Lindblad

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Doctor of Philosophy

Major: Mathematics

Under the Supervision of Professors Susan M. Hermiller and John C. Meakin

Lincoln, Nebraska

December, 2003

INVERSE MONOIDS PRESENTED BY A SINGLE SPARSE RELATOR

Steven P. Lindblad, Ph.D.

University of Nebraska, 2003

Advisors: Susan M. Hermiller and John C. Meakin

We study a class of inverse monoids of the form $M = \mathrm{Inv}\langle X \mid w = 1\rangle$, where the single relator w has a combinatorial property that we call **sparse**. For a sparse word w, we prove that the Schützenberger complex of the identity of M has a particularly nice topology. We analyze the manner in which the Schützenberger complex is constructed using an iterative procedure, due to Stephen, analogous to the Todd-Coxeter procedure. We define an appropriate notion of dual graph of the Schützenberger complex, and prove that this dual graph is a tree if w is sparse. We use this tree to construct a pushdown automaton that encodes the information contained in the Schützenberger complex. This pushdown automaton provides us with an algorithm that, given a word $u \in (X \cup X^{-1})^*$, determines whether or not $u = 1$ in M. This, together with results of Stephen and the fact due to Ivanov, Margolis, and Meakin that the inverse monoid is E-unitary, shows that the word problem is solvable for M. Finally, we provide an implementation, in the C++ programming language, of the algorithm to determine whether or not $u = 1$ in M.

ACKNOWLEDGEMENTS

I would like to thank my advisors, Susan Hermiller and John Meakin, for their valuable advice, encouragement and support. I am also grateful to the other members of my Supervisory Committee, Sharad Seth, Thomas Shores, and Roger Wiegand. In particular, I would like to thank Professors Shores and Wiegand for reading my dissertation and making many valuable comments. I am grateful to Steve Haataja for explaining his work on inverse monoids defined by a relator which is a product of commutators. Finally, I wish to thank my parents, Philip and Barbara Lindblad, for a lifetime of patience, generosity, and encouragement.

Contents

1 Introduction **1**

2 Background **6**

2.1 Preliminaries 6

2.1.1 Semigroups, monoids, and words 6

2.1.2 Green's relations 7

2.1.3 Inverse semigroups 8

2.1.4 Congruences and inverse monoid presentations 8

2.1.5 Graphs, inverse word graphs, and geodesics 9

2.1.6 Automata and pushdown automata 10

2.2 Schützenberger graphs 13

2.2.1 Cayley graphs and Schützenberger graphs 13

2.2.2 Cone types 14

2.2.3 Iterative construction of Schützenberger graphs 15

2.2.4 Munn trees and the free inverse monoid 17

2.2.5 E-unitary inverse monoids 17

2.3 CW-complexes and Schützenberger complexes 18

2.3.1 CW-complexes 18

2.3.2 The Schützenberger complex 19

2.4 w-CW complexes 20
2.4.1 Cyclic substrings 20
2.4.2 w-CW complexes 22
2.4.3 Schützenberger approximation sequences 24

3 Inverse monoids defined by a sparse relator 29

3.1 Relator equal to a product of two commutators 30
3.2 Sparse words 32
3.2.1 Definitions 33
3.2.2 Examples of sparse and nonsparse words 38
3.3 The Schützenberger complex 40
3.4 The dual graph 53

4 The word problem 57

4.1 The word problem is solvable 57
4.2 Face types and the PDA for $SC(1)$ 58
4.3 Iterative construction of the PDA 66
4.3.1 Data objects 67
4.3.2 Construction algorithm 71
4.4 Geodesics in $S\Gamma(1)$ 76

5 Implementation and sample runs 79

5.1 Sample run 80
5.1.1 Entering the relator 80
5.1.2 Sample output for $w = abABcdCD$ 80
5.1.3 Sample output for $w = abcxaxabxabcy$ 89
5.2 Overview of the program 94
5.2.1 The `word` program component 95

5.2.2 The `sc` program component 96

A Computer source code **98**

A.1 Procedures for words 98

A.1.1 Source file `word.h` 98

A.1.2 Source file `word.cc` 101

A.2 Procedures for Schützenberger complexes 106

A.2.1 Source file `sc.h` 106

A.2.2 Source file `sc.cc` 110

A.3 Front-end program: `spar.cc` 125

Bibliography **132**

List of Figures

3.1 Part of $SC(1)$ for $M = \mathrm{Inv}\langle a, b, c, d \,|\, abABcdCD = 1\rangle$ 30

3.2 Pointed pieces and segments of $w = abABcdCD$ 33

3.3 A nonmaximal pointed piece (dotted segments) 34

3.4 Noncomparable pointed pieces 35

3.5 Calculation of the start vertex index inside an associated segment . . 36

3.6 Product of three commutators: $w = abABcdCDefEF$ 39

3.7 One commutator: $w = abAB$ 39

3.8 $w = abcxaxabxabcy$ 40

3.9 Unique F—no other face B can exist 42

3.10 Proof of Claim 1 45

3.11 Proof of Claim 2 Case I 46

3.12 Proof of Claim 2 Case II 46

3.13 Two definitions of dual graph 53

Chapter 1

Introduction

In this dissertation, we examine a class of inverse monoids presented by a single relator, $M = \mathrm{Inv}\langle X \,|\, w = 1\rangle$, in which the relator w satisfies a condition we call "sparse." We show that the word problem for such monoids is solvable, and provide an implementation of an algorithm to determine, given input $u \in (X \cup X^{-1})^*$, whether or not $u = 1$ in M. The existence of such an algorithm guarantees that the word problem for M is solvable, by [Ste93].

The word problem for groups and monoids has been studied since the early twentieth century, and is undecidable in general. A group, monoid, or inverse monoid has a solvable word problem if, given a set of elements which generate the group or monoid, there is an algorithm which will determine, given two products in the generators u and v, whether or not $u = v$. In 1947, Post and Markov independently proved that the word problem for monoids is unsolvable. Novikov in 1955 and Boone in 1959 proved that the word problem for finitely presented groups is unsolvable in general. Since a group is an inverse monoid, this also shows that the word problem for inverse monoids is undecidable in general.

Munn [Mun74] proved in 1974 that the word problem for the free inverse monoid

is solvable. In 1987 Stephen [Ste87, Ste90] began a study of inverse monoid presentations, and used his methods to solve the word problems for certain inverse monoid presentations. There has been an extensive study of inverse monoid word problems. Birget, Margolis, and Meakin [BMM94] proved that the word problem is solvable for inverse monoids of the form $\mathrm{Inv}\langle X \mid e = 1\rangle$, where e is an idempotent in the free inverse monoid (i.e., reduces to 1 in the free group). Margolis and Meakin [MM93] proved that inverse monoids of the form $\mathrm{Inv}\langle X \mid e_i = f_i, i \in I\rangle$, where e_i, f_i are idempotents and I is finite, have solvable word problem, and Silva [Sil92] found an alternate proof of the same result. Inverse monoid presentations have also been studied in [IKT02], [Sil95], [Ste00], and [Yam01].

In studying group presentations $G = \mathrm{Gp}\langle X \mid T\rangle$, the "Cayley graph" has been useful. The Cayley graph is the graph whose vertices are the elements of the group and whose edges are labeled by the alphabet, or set of generators, X, so that there is an edge from u to v labeled by x if $ux = v$ in the group. A similar graph, also called the Cayley graph, is defined for inverse monoid presentations. However, unlike the case for groups, $ux = v$ does not necessarily imply $vx^{-1} = u$, so edges in the Cayley graph of an inverse monoid do not necessarily have corresponding inverse edges. As a result, the strongly connected components of the Cayley graph, that is, the subgraphs in which each edge has a corresponding inverse edge, are more useful for studying inverse monoid presentations. These strongly connected components are called Schützenberger graphs, and they may also be viewed as the subgraphs of the Cayley graph restricted to vertices which are equivalent under Green's $\mathcal{R}$ relation for the monoid. From the Schützenberger graph, the Schützenberger complex can be defined as the complex whose 1-skeleton is the Schützenberger graph and whose faces have boundaries labeled by the sides of relations [Ste03]. This is analogous to the relationship between the Cayley complex and Cayley graph for groups.

Stephen developed a method for iteratively building approximations to the Schützenberger graph of an element of the inverse monoid, similar to the Todd-Coxeter procedure for building approximations to the Cayley graph of a group. His method consists of iteratively applying "expansions" and "foldings." Stephen proved that these processes are confluent, so that although there are choices for the order in which to perform expansions and foldings on a given approximation, additional expansions and foldings may be performed on the results to arrive at a common approximation. He also proved that the Schützenberger graph is the direct limit, in an appropriate category, of all approximations built in this manner.

Stephen [Ste93] applied his methods to the case of a single relator of the form $w = 1$. In particular, he observed that if the inverse monoid $M = \mathrm{Inv}\langle X \,|\, w = 1\rangle$ is E-unitary, then the word problem for M is decidable if there is an algorithm to decide, for any word u, whether or not $u = 1$. Furthermore, Ivanov, Margolis, and Meakin [IMM01] proved that if w is cyclically reduced, then $M = \mathrm{Inv}\langle X \,|\, w = 1\rangle$ is E-unitary. Thus the word problem for $M = \mathrm{Inv}\langle X \,|\, w = 1\rangle$, w cyclically reduced, is reduced to understanding the Schützenberger graph of 1 in M.

The goal of this paper is to follow this program when w satisfies a condition we call "sparse." We define a "pointed piece" to be a pair of locations in w at which the same subword is seen, one of which includes the beginning or end of the word. Then w is "sparse" if all of its pointed pieces are separated from each other by at least one letter. This can be viewed as roughly analogous to the small cancellation hypotheses for group theory, except that the locations of pointed pieces, rather than their lengths, are relevant.

When w is sparse, Stephen's iterative process can be used to construct the Schützenberger graph of 1 in a well-behaved way. We prove that the Schützenberger graph has a tree-like structure; specifically, the faces of the Schützenberger complex are the

vertices of an underlying directed graph we call the "dual graph" and this dual graph is a tree. In addition, we prove that the iterative construction always grows away from the origin. Thus the word problem is solvable because we can build the ball of any required radius in the Schützenberger graph after a bounded number of steps.

However, in addition to this we take a more efficient approach to understanding the Schützenberger graph. The use of the dual graph enables us to encode the information contained in the Schützenberger graph of 1 in a pushdown automaton (PDA). We show that each face in the Schützenberger complex is one of finitely many types. Once the face types are known, we can read inside the Schützenberger complex by reading within these face types and pushing an appropriate marker on the stack whenever we move down in the dual graph from one face type to another.

We also use face types to analyze geodesics and cone types in Schützenberger graphs. A geodesic is a word which labels a geodesic (shortest) path in the graph. The cone type of a vertex of a Schützenberger graph is the set of suffixes, starting at the vertex, of geodesic words which start at the origin and pass through the selected vertex. A finite state automaton is naturally associated with the cone types, and this automaton recognizes the language of geodesics. We show that a finite state automaton which recognizes geodesics can immediately be obtained from the pushdown automaton for the Schützenberger complex of 1, from which it follows that the minimized form of this automaton is the cone type automaton. As a result, there are finitely many cone types.

The following chapters are arranged as follows: Chapter 2 provides basic definitions and results from inverse monoid theory as well as Stephen's methods and a description of the specific way in which we apply his iterative construction. Chapter 3 formally defines "sparse" and proves that the dual graph is a tree. Chapter 4 presents a solution to the word problem, and defines the pushdown automaton and cone type

automaton. In addition, the algorithm used by the C++ program to produce the PDA is described, and its correctness is proved. Chapter 5 presents sample runs of the C++ program, `spar`, and gives an overview of the structure of the program. The complete source code for the program is given in the appendix, and is available at `http://www.math.unl.edu/~slindbla/sparse/` [Lin].

Chapter 2

Background

2.1 Preliminaries

2.1.1 Semigroups, monoids, and words

A **semigroup** $(S, \cdot)$ consists of a set S and an binary operation $\cdot$ such that $a \cdot b \in S$ for all $a, b \in S$ and $\cdot$ satisfies the associative law:

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c \quad \text{for all } a, b, c \in S.$$

When there is no danger of confusion, we will frequently suppress the operator $\cdot$ and write ab for $a \cdot b$.

An **idempotent** of S is an element $e \in S$ such that $e^2 = e$. The set of all idempotents of S is denoted $E(S)$.

An **identity** is an element $1 \in S$ such that $1a = a1 = a$ for all $a \in S$. A semigroup which contains an identity is a **monoid**. A **zero** of S is an element $0 \in S$ such that $0a = a0 = 0$ for all $a \in S$. Note it follows directly from these definitions that a semigroup has at most one identity and one zero.

A semigroup and monoid of particular interest are the **free semigroup** and **free monoid**, defined as follows: Let X be a set, called an **alphabet**, whose elements are **letters**. (For our purposes, alphabets will be finite.) A **word** over X, of **length** n, is a sequence $a_1 \dots a_n$ of letters $a_i \in X$. We call the word of length zero the **empty word** and denote it ϵ. We define multiplication of words by concatenation, so $a_1 \dots a_n \cdot b_1 \dots b_m = a_1 \dots a_n b_1 \dots b_m$. Since this multiplication is associative, the set of all words of positive length is a semigroup, the free semigroup X^+. The empty word is an identity under the operation of concatenation, so the set of all words is a monoid, the free monoid X^*. A subset of X^* or X^+ is called a **language**.

2.1.2 Green's relations

If S is any semigroup, we define $S^1 = S$ if S is a monoid, otherwise we define $S^1 = S \cup \{1\}$ (where 1 is some element not in S) with $1s = s1 = s$ for $s \in S$. Given $t \in S$, the principal left ideal generated by t is the set $S^1t = \{st \mid s \in S^1\}$. The principal right and two-sided ideals tS^1 and S^1tS^1 are defined similarly.

The elements of a semigroup are partitioned by five equivalence relations, $\mathcal{L}$, $\mathcal{R}$, $\mathcal{H}$, $\mathcal{D}$, and $\mathcal{J}$, called **Green's relations**, defined as follows:

- $a\mathcal{L}b$ if and only if $S^1a = S^1b$.
- $a\mathcal{R}b$ if and only if $aS^1 = bS^1$.
- $a\mathcal{H}b$ if and only if $a\mathcal{L}b$ and $a\mathcal{R}b$; i.e., $\mathcal{H} = \mathcal{L} \cap \mathcal{R}$.
- $a\mathcal{D}b$ if and only if there exists $c \in S$ such that $a\mathcal{L}c$ and $c\mathcal{R}b$; i.e., $\mathcal{D} = \mathcal{L} \circ \mathcal{R}$. Equivalently, $\mathcal{D} = \mathcal{R} \circ \mathcal{L}$ [Pet84, p. 27] and $\mathcal{D}$ is the smallest equivalence relation containing $\mathcal{L}$ and $\mathcal{R}$ [Pet84, p. 9].
- $a\mathcal{J}b$ if and only if $S^1aS^1 = S^1bS^1$.

The $\mathcal{L}$-, $\mathcal{R}$-, $\mathcal{H}$-, $\mathcal{D}$- and $\mathcal{J}$-(equivalence) classes of a are denoted L_a, R_a, H_a, D_a, and J_a, respectively.

2.1.3 Inverse semigroups

An element $a \in S$ is **regular** if $a = aba$ for some $b \in S$. S is regular if all of its elements are regular. S is an **inverse** semigroup if S is regular and its idempotents commute. Equivalently, S is inverse if every $a \in S$ has a unique **inverse** $a^{-1} \in S$ such that $a = aa^{-1}a$ and $a^{-1} = a^{-1}aa^{-1}$ [Pet84, p. 78].

Note that elements of the form aa^{-1} and $a^{-1}a$ are idempotents since $(aa^{-1})^2 = aa^{-1}aa^{-1} = aa^{-1}$ and similarly for $a^{-1}a$. (A **group** is an inverse monoid in which the identity is the only idempotent, so $aa^{-1} = a^{-1}a = 1$.) If e is an idempotent, then $eee = e$, so $e^{-1} = e$ and $e = ee^{-1} = e^{-1}e$. Thus every idempotent of an inverse semigroup can be written in the forms aa^{-1} and $a^{-1}a$. If S is an inverse semigroup, then $a\mathcal{L}b \Leftrightarrow a^{-1}a = b^{-1}b$ and $a\mathcal{R}b \Leftrightarrow aa^{-1} = bb^{-1}$ [Pet84, p. 80]. It follows that each $\mathcal{L}$-class L_a and $\mathcal{R}$-class R_a contains exactly one idempotent, namely $a^{-1}a$ and aa^{-1}, respectively.

The **natural partial order** $\leq$ on an inverse semigroup S is defined by $a \leq b$ if and only if $a = be$ for some idempotent $e \in E(S)$. Equivalently, $a \leq b$ if and only if $a = ba^{-1}a$, $a \leq b$ if and only if $a = fb$ for some $f \in E(S)$, and $a \leq b$ if and only if $a = aa^{-1}b$ [Law98, p. 21]. The natural partial order is the identity relation if and only if S is a group [Law98, p. 21].

2.1.4 Congruences and inverse monoid presentations

A **congruence** on a semigroup S is an equivalence relation ρ which is compatible with the semigroup operation; i.e., if $a\rho b$ then $(ac)\rho(bc)$ and $(ca)\rho(cb)$ for all $a, b, c \in S$. The congruence class of a is denoted $a\rho$. If $a\rho a'$ and $b\rho b'$ then $(ab)\rho(a'b)$ and $(a'b)\rho(a'b')$, so

$(ab)\rho(a'b')$. It follows that the multiplication of congruence classes $(a\rho)(b\rho) := (ab)\rho$ is well-defined and forms a semigroup, the **quotient semigroup** S/ρ.

If X is an alphabet, let X^{-1} denote a disjoint set of inverses. The **Vagner congruence** ρ is the congruence generated by

$$\{(uu^{-1}u, u), (uu^{-1}vv^{-1}, vv^{-1}uu^{-1}) \mid u, v \in (X \cup X^{-1})^*\}.$$

It follows that $\mathrm{FIM}(X) := (X \cup X^{-1})^*/\rho$ is a inverse monoid, the **free inverse monoid**. (The relations $(uu^{-1}vv^{-1}, vv^{-1}uu^{-1})$ capture the notion of idempotents commuting.)

If $T \subset (X \cup X^{-1})^* \times (X \cup X^{-1})^*$ and τ is the congruence generated by $\rho \cup T$ (where ρ is the Vagner congruence), then $M = \mathrm{Inv}\langle X \mid T\rangle := (X \cup X^{-1})^*/\tau$ is the **inverse monoid presented by the set X of generators and the set T of relations**. For $w \in (X \cup X^{-1})^*$, we denote the inverse monoid element $w\tau$ by $\overline{w}$. If X is finite, then M is said to be **finitely generated**. If both X and T are finite, then M is **finitely presented**. Elements (u, v) of T are **relations** and are written $u = v$. If a relation is of the form $w = 1$, then w is a **relator**.

The **word problem** for $\mathrm{Inv}\langle X \mid T\rangle$ is the question of whether there is an algorithm which, given any two words $w, u \in (X \cup X^{-1})^*$, will determine whether $\overline{w} = \overline{u}$. The word problem is in general unsolvable; that is, there is no such algorithm in general. This follows from the fact that groups are inverse monoids and Novikov and Boone proved that the word problem for finitely presented groups is unsolvable.

2.1.5 Graphs, inverse word graphs, and geodesics

A **directed graph** Γ consists of a set of **vertices** $V(\Gamma)$ and **edges** $E(\Gamma) \subseteq V(\Gamma) \times V(\Gamma)$. An edge (u, v) is the directed edge from its **initial vertex** u to its **terminal**

vertex v. In general, there may or may not be a corresponding edge (v, u).

A **labeled directed graph** over X is a directed graph in which the edges are labeled by elements of X. We write (u, x, v) to denote the edge (u, v) labeled by x. A **path** (of **length** n) is a sequence of edges

$$((v_0, x_1, v_1), (v_1, x_2, v_2), \ldots, (v_{n-1}, x_n, v_n))$$

such that the initial vertex of each edge (except the first) equals the terminal vertex of the previous edge. If $v_0 = v_n$ the path is a **circuit**. We say that the path is **labeled** by the word $w = x_1 \ldots x_n$ and that w can be **read** in the graph starting at v_0. A labeled directed graph is **strongly connected** if there is a path from u to v for every pair of vertices u, v.

An **inverse word graph over** X is a labeled directed graph over $X \cup X^{-1}$ such that the labeling is consistent with involution; that is, (u, x, v) is an edge if and only if (v, x^{-1}, u) is an edge. A **birooted inverse word graph** is an inverse word graph Γ with vertices $\alpha, \beta \in V(\Gamma)$ identified as the **start** and **end** vertices, respectively. The language $L[A]$ of a birooted inverse graph $A = (\alpha, \Gamma, \beta)$ is the set of words which can be read from α to β in Γ.

Given a labeled directed graph Γ with $u, v \in V(\Gamma)$ a **geodesic path** from u to v is a path of minimum length. A word which labels a geodesic path is called a **geodesic word**. The **path metric** maps vertices u, v to the length $d(u, v)$ of a geodesic path.

2.1.6 Automata and pushdown automata

We relax the definition of finite state automaton from [HU79] to allow infinitely many states:

Definition 2.1. *An* **automaton** *is a five-tuple* $A = (Q, \Sigma, \delta, q_0, F)$*, where*

- *Q is a set whose elements are called* **states**,
- *Σ is an alphabet, called the* **input alphabet**,
- *$\delta\colon Q\times\Sigma \to Q$ is a partial function (i.e., not necessarily defined on all of $Q\times\Sigma$), called the* **transition function**,
- *$q_0 \in Q$ is the* **initial state**, *and*
- *$F \subseteq Q$ is the set of* **final states** *(or* **accept** *or* **terminal** *states).*

If Q is finite, then the automaton is a **finite state automaton**.

We may view A as a labeled directed graph with

- $V(A) = Q$ and
- $E(A) = \{(q, x, r) \mid \delta(q, x) = r\}$.

We say A **accepts** or **recognizes** a word $u \in \Sigma^*$ if u labels a path in the labeled directed graph from q_0 to f for some $f \in F$. The **language** $L(A)$ of A is the set of all words accepted by A. A language is **regular** if it is the language of a finite state automaton.

Given a regular language L, there may be many different finite state automata which recognize L. However, there is a unique (up to isomorphism) finite state automaton with a minimum number of states [HU79]. In addition, given any automaton recognizing a language, the minimum state automaton may be computed by applying the minimization algorithm of [HU79].

The following definition is a special case of a deterministic pushdown automaton as defined in [HU79]:

Definition 2.2. *A* **pushdown automaton** *(PDA) is a seven-tuple*

$$D = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$$

where

- Q *is a* finite *set of* **states**,
- Σ *is the* **input alphabet**,
- Γ *is the* **stack alphabet**,
- $\delta \colon Q \times \Sigma \times \Gamma \to Q \times \Gamma^*$ *is a partial function called the* **transition function**,
- $q_0 \in Q$ *is the* **initial state**,
- $Z_0 \in \Gamma$ *is the* **initial stack symbol**, *and*
- $F \subseteq Q$ *is the set of* **final, or accept, states**.

An **instantaneous description (ID)** *is a triple* (q, w, γ) *where* $q \in Q$, $w \in \Sigma^*$, *and* $\gamma \in \Gamma^*$. *For a PDA* D, *we say* $(q, aw, Z\gamma) \vdash (r, w, z\gamma)$ *if* $\delta(q, a, Z) = (r, z)$, *and we write* $\vdash^*$ *for the reflexive and transitive closure of* $\vdash$. *We say* D **accepts** *a word* $u \in \Sigma^*$ *if* $(q_0, u, Z_0) \vdash^* (f, \epsilon, \gamma)$ *for some* $f \in F$ *and* $\gamma \in \Gamma^*$.

The intuitive interpretation of this is that a PDA is a machine whose state at any given instant is described by an ID (q, w, t), where q is the current state, w is the portion of the input remaining to be read, and t is the current stack (the left-most letter of which is the top or front of the stack). When u is given as input to the machine, D starts in ID (q_0, u, Z_0), and u is accepted if the machine can read all of the word and be left in one of the final, or accept, states.

If $\delta(q, a, Z) = (r, Z)$ then a moves the machine from q to r without changing the stack. If $\delta(q, a, Z) = (r, \epsilon)$ then the letter Z is "popped" off the top of the stack when a is read at state q. If $\delta(q, a, Z) = (r, YZ)$ for $Y \in \Gamma$ then Y is "pushed" onto the stack when a is read. In general, $\delta(q, a, Z) = (r, z)$ may be viewed as popping Z and then pushing the letters of z all in one step. However, all of our PDAs have simple transition functions which either push or pop a single letter or leave the stack alone.

2.2 Schützenberger graphs

2.2.1 Cayley graphs and Schützenberger graphs

Let $M = \mathrm{Inv}\langle X \,|\, T\rangle$ be the inverse monoid presented by X and T. The **Cayley graph** of M with respect to the the presentation $\mathrm{Inv}\langle X \,|\, T\rangle$ is the labeled directed graph Γ over $X \cup X^{-1}$ such that

- $V(\Gamma) = M$ and
- $E(\Gamma) = \{(m_1, x, m_2) \,|\, m_1, m_2 \in M \text{ and } m_1\overline{x} = m_2\}$.

Note that if M is a group, Γ is the Cayley graph in the usual group-theoretic sense.

Since $maa^{-1} = m$ does not necessarily hold in an inverse monoid, the Cayley graph is not necessarily an inverse word graph and is not necessarily strongly connected. If $m = maa^{-1}$ then $mm^{-1} = maa^{-1}m^{-1} = (ma)(ma)^{-1}$, so $m\mathcal{R}(ma)$. Conversely,

$$\begin{aligned} m\mathcal{R}(ma) &\Rightarrow mm^{-1} = (ma)(ma)^{-1} = maa^{-1}m^{-1} \\ &\Rightarrow m = mm^{-1}m = maa^{-1}m^{-1}m = mm^{-1}maa^{-1} = maa^{-1}, \end{aligned}$$

since idempotents (aa^{-1} and $m^{-1}m$) commute in an inverse monoid. Thus each $\mathcal{R}$-class is a strongly connected component of the Cayley graph. The restriction of the

Cayley graph to the strongly connected component with vertices R_m is called the **Schützenberger graph** of m, denoted $S\Gamma(m)$, with respect to the presentation. That is,

- $V(S\Gamma(m)) = R_m$ and
- $E(S\Gamma(m)) = \{(m_1, x, m_2) \mid m_1, m_2 \in R_m \text{ and } m_1\overline{x} = m_2\}$.

If $w \in (X \cup X^{-1})^*$ labels a path from m to m' in R_m, then $m' = m\overline{w}$ and $m\overline{w}\,\overline{w}^{-1} = m$ so $S\Gamma(m)$ is an inverse word graph.

Note that for $m \in M$ and $w \in (X \cup X^{-1})^*$, $m \leq \overline{w}$ if and only if $mm^{-1}\overline{w} = m$, so a word w labels a path from mm^{-1} to m in $S\Gamma(m)$ if and only if $\overline{w} \geq m$ in the natural partial order of M. In this way we can view the birooted inverse word graph $(mm^{-1}, S\Gamma(m), m)$ as an automaton, the **Schützenberger automaton**, with initial vertex mm^{-1} and terminal vertex m which recognizes words which map to the **order filter** of m, $m\uparrow = \{\overline{w} \in M \mid m \leq \overline{w}\}$.

2.2.2 Cone types

Let $M = \mathrm{Inv}\langle X \mid T\rangle$, $e \in E(M)$ and let $y \in (X \cup X^{-1})^*$ label a geodesic path from e to $e\overline{y}$ in $S\Gamma(e)$. Then the **cone type of y relative to** e is the set

$$C_e(y) = \{u \in (X \cup X^{-1})^* \mid yu \text{ labels a geodesic path from } e \text{ to } e\overline{yu} \text{ in } S\Gamma(e)\}.$$

If y_1 and y_2 are both geodesic, then $C_e(y_1) = C_e(y_2)$ if and only if $e\overline{y_1} = e\overline{y_2}$, so $C_e(y)$ depends only on $e\overline{y}$. Thus for all $m \in R_e$ we define

$$C_e(m) = C_e(y) \text{ such that } y \text{ labels a geodesic path from } e \text{ to } m \text{ in } S\Gamma(m).$$

We define an automaton whose states are the cone types relative to e and whose

transition function is given by $\delta(C_e(m), x) = C_e(m\overline{x})$ [ECH+92]. If $C_e(e)$ is the initial state and all states accept, then this automaton accepts precisely the set of words which label geodesic paths in $S\Gamma(e)$. We call this the **language of geodesics**. If there are finitely many cone types, then the automaton is a finite state automaton and the language of geodesics is regular. Since each state corresponds to a different cone type, by definition, this automaton must be the minimum state automaton accepting the language of geodesics.

2.2.3 Iterative construction of Schützenberger graphs

In this section we summarize the iterative procedure described by Stephen [Ste87, Ste90] for building a Schützenberger graph. Let $\mathrm{Inv}\langle X \mid T\rangle$ be a presentation of an inverse monoid.

Given a word $u = a_1 \dots a_n \in (X \cup X^{-1})^*$, the **linear graph** of u is the birooted inverse word graph $(\alpha_u, \Gamma_u, \beta_u)$ consisting of a set of vertices

$$V((\alpha_u, \Gamma_u, \beta_u)) = \{\alpha_u, \beta_u, \gamma_1, \dots, \gamma_{n-1}\}$$

and edges

$$(\alpha_u, a_1, \gamma_1), (\gamma_1, a_2, \gamma_2), \dots, (\gamma_{n-2}, a_{n-1}, \gamma_{n-1}), (\gamma_{n-1}, a_n, \beta_u),$$

together with the corresponding inverse edges.

Let (α, Γ, β) be a birooted inverse word graph over $X \cup X^{-1}$. The following operations may be used to obtain a new birooted inverse word graph $(\alpha', \Gamma', \beta')$:

- **Determination** or **folding**: Let (α, Γ, β) be a birooted inverse word graph with vertices v, v_1, v_2, $v_1 \neq v_2$, and edges (v, x, v_1) and (v, x, v_2) for some $x \in X \cup X^{-1}$.

Then we obtain a new birooted inverse word graph $(\alpha', \Gamma', \beta')$ by taking the quotient of (α, Γ, β) by the equivalence relation which identifies the vertices v_1 and v_2 and the two edges. In other words, edges with the same label coming out of a vertex are folded together to become one edge.

- **Elementary expansion** or **sewing**: Let $r = s$ be a relation in T and suppose that r can be read from v_1 to v_2 in Γ, but s cannot be read from v_1 to v_2. Then we define $(\alpha', \Gamma', \beta')$ to be the quotient of $\Gamma \cup (\alpha_s, \Gamma_s, \beta_s)$ by the equivalence relation which identifies vertices v_1 and α_s and vertices v_2 and β_s. In other words, we "sew" on a linear graph for s from v_1 to v_2 to complete the other half of the relation $r = s$.

A graph is **deterministic** if no foldings can be performed and **closed** if it is deterministic and no elementary expansions can be performed. Note that given a finite inverse word graph it is always possible to produce a determinized form for the graph, because determination reduces the number of vertices and this must stop after finitely many steps.

If $(\alpha_1, \Gamma_1, \beta_1)$ is obtained from (α, Γ, β) by an elementary expansion, and $(\alpha_2, \Gamma_2, \beta_2)$ is the determinized form of $(\alpha_1, \Gamma_1, \beta_1)$, then we write $(\alpha, \Gamma, \beta) \Rightarrow (\alpha_2, \Gamma_2, \beta_2)$ and say that (α, Γ, β) is obtained from $(\alpha_2, \Gamma_2, \beta_2)$ by an **expansion**. The reflexive and transitive closure of $\Rightarrow$ is denoted $\Rightarrow^*$.

For $u \in (X \cup X^{-1})^*$, an **approximate graph** of $(\overline{uu^{-1}}, S\Gamma(\overline{u}), \overline{u})$ is a birooted inverse word graph $A = (\alpha, \Gamma, \beta)$ such that $u \in L[A]$ and $\overline{y} \geq \overline{u}$ for all $y \in L[A]$. Stephen showed in [Ste87, Ste90] that the linear graph of u is an approximate graph of $(\overline{uu^{-1}}, S\Gamma(\overline{u}), \overline{u})$. He also proved the following:

Theorem 2.3. *Let $u \in (X \cup X^{-1})^*$ and let (α, Γ, β) be an approximate graph of $(\overline{uu^{-1}}, S\Gamma(\overline{u}), \overline{u})$. If $(\alpha, \Gamma, \beta) \Rightarrow^* (\alpha', \Gamma', \beta')$ and $(\alpha', \Gamma', \beta')$ is closed, then $(\alpha', \Gamma', \beta')$*

is the Schützenberger graph $(\overline{uu^{-1}}, S\Gamma(\overline{u}), \overline{u})$.

In [Ste87], Stephen showed that the class of all birooted inverse word graphs over $X \cup X^{-1}$ is a cocomplete category and that the directed system of all finite expansions of the linear graph of u has a direct limit. Since the directed system includes all possible expansions, this limit must be closed. Therefore, by Theorem 2.3, the Schützenberger graph is the direct limit.

2.2.4 Munn trees and the free inverse monoid

If there are no relations, so $M = \mathrm{FIM}(X) = \mathrm{Inv}\langle X \mid \emptyset \rangle$ (the free inverse monoid), then $(mm^{-1}, S\Gamma(m), m)$ is a tree, the **Munn tree** of $m \in \mathrm{FIM}(X)$. The Munn tree for $\overline{u}$, given $u \in (X \cup X^{-1})^*$, may be constructed by building the linear graph of u and folding, since there are no relations by which to expand. Thus the word problem for $\mathrm{FIM}(X)$ is solvable: two words are equal if and only if they have the same Munn tree. This is the solution is due to Munn [Mun74].

2.2.5 E-unitary inverse monoids

An inverse monoid $M = \mathrm{Inv}\langle X \mid T \rangle$ is **E-unitary** if the natural morphism σ from M to its maximal group image $G = \mathrm{Gp}\langle X \mid T \rangle$ is **idempotent-pure**; that is, $\sigma^{-1}(1) = E(M)$. There are many other equivalent characterizations of this important property. The following statements are equivalent:

- M is E-unitary.
- For all $e, m \in M$, $e \leq m$ and $e \in E(M)$ implies $m \in E(M)$.
- Each Schützenberger graph naturally embeds into the Cayley graph of G. (This is due to Meakin [Ste90].)

In [Ste93], Stephen studied inverse monoid of the form $M = \mathrm{Inv}\langle X \,|\, w_i = 1,\, i \in I\rangle$, where I is finite. Let $T = \{w_i = 1 \,|\, i \in I\}$ be a set of relators, let $M = \mathrm{Inv}\langle X \,|\, T\rangle$ and $G = \mathrm{Inv}\langle X \,|\, T\rangle$, and let $\sigma\colon M \to G$ be the natural morphism. Let $\Gamma(X,T)$ be the Cayley graph of G with respect to the generators X. Then σ maps the vertices of $S\Gamma(1)$ into $\Gamma(X,T)$. If $(m_1, x, m_2) \in E(S\Gamma(1))$ whenever $(\sigma(m_1), x, \sigma(m_2)) \in E(\Gamma(X,T))$ then we say $S\Gamma(1)$ is a **full subgraph** of $\Gamma(X,T)$. Stephen showed that if M is E-unitary, then $S\Gamma(1)$ is a full subgraph of $\Gamma(X,T)$.

We say a word $w = a_1 \dots a_n \in (X \cup X^{-1})^*$ is **cyclically reduced** if no two adjacent letters are mutually inverse and if $a_1 \neq a_n^{-1}$.

The following important theorem was proved in [IMM01]:

Theorem 2.4. *If w is a cyclically reduced word, then $M = \mathrm{Inv}\langle X \,|\, w = 1\rangle$ is E-unitary.*

Thus, it follows from this theorem and Stephen's results that $S\Gamma(1)$ is a full subgraph of $\Gamma(X,T)$. As a consequence, Stephen's results show that the Schützenberger graph of a word $u \in (X \cup X^{-1})^*$ may be viewed as the Munn tree of u with a copy of $S\Gamma(1)$ sewn on at each vertex, after folding. From this it follows that the word problem for $\mathrm{Inv}\langle X \,|\, w = 1\rangle$ is solvable if there is an algorithm to determine whether or not a word u labels a path from 1 to 1 in $S\Gamma(1)$; that is, whether or not $\overline{u} = 1$.

2.3 CW-complexes and Schützenberger complexes

2.3.1 CW-complexes

We now turn our attention to some topological concepts. Let

$$D^n = \{x \in \mathbb{R}^n \,|\, |x| \leq 1\},\ \mathring{D}^n = \{x \in \mathbb{R}^n \,|\, |x| < 1\},\ \text{and } \partial D^n = \{x \in \mathbb{R}^n \,|\, |x| = 1\}$$

be the closed and open unit Euclidean disks and their boundary, respectively. A space is called an **open cell** of dimension n, or an n**-cell**, if it is homeomorphic to $\mathring{D}^n$.

A topological space S is **Hausdorff** if for every pair of points $p, q \in S$, $p \neq q$, there are open sets P and Q such that $p \in P$, $q \in Q$, and $P \cap Q = \emptyset$.

The following definition is due to J. H. C. Whitehead:

Definition 2.5. *A* **CW-complex** *is a Hausdorff space S and a collection of open cells $\{e_\alpha\}_{\alpha \in I}$ such that the following hold:*

(1) $S = \coprod_{\alpha \in I} e_\alpha$.

(2) For each n-cell e_α there is a map $\theta_\alpha \colon D^n \to S$ such that θ_α induces a homeomorphism $\theta_\alpha|_{\mathring{D}^n} : \mathring{D}^n \to e_\alpha$ and $\theta_\alpha(\partial D^n)$ is contained in a finite union of open cells e_β of dimension less than n.

(3) A set $A \subseteq S$ is open in S if and only if $A \cap \overline{e}_\alpha$ is open in $\overline{e}_\alpha$ for all $\alpha \in I$, where $\overline{e_\alpha}$ is the closure of e_α in S.

We denote the set of 0-cells or **vertices** of a CW-complex by $V(S)$, the set of closed 1-cells or **edges** by $E(S)$, and the set of closed 2-cells or **faces** by $F(S)$. We do not have a need to discuss cells of higher dimension in this paper. Given a vertex $v \in V(S)$, the **star set** $\mathrm{Star}(v)$ of v is the union of faces $F \in F(S)$ such that $v \in F$.

Given an integer m, the m**-skeleton** of a CW-complex S, denoted $S^{(m)}$, is the union of the open cells of dimension at most m. A **cellular map** is a continuous map between CW-complexes which maps open cells to open cells of the same dimension.

2.3.2 The Schützenberger complex

We now expand our notion of Schützenberger graph from Section 2.2.1. Let $M = \mathrm{Inv}\langle X \,|\, T\rangle$ be an inverse monoid presentation and $m \in M$.

Following Steinberg [Ste03], we define the **Schützenberger complex** $SC(m)$ for $m \in M$ as follows:

(1) The 1-skeleton of $SC(m)$ is the Schützenberger graph $S\Gamma(m)$.

(2) For each relation $r = s$ in T and vertex v, there is a face with boundary given by the pair of paths labeled by r and s starting from v.

It should be noted that in contrast to the above, Steinberg's definition of "Schützenberger complex" includes vertices for the entire inverse monoid; that is, it is the complex we might, following the analogous language in this paper, call the "Cayley complex" of the inverse monoid. Our Schützenberger complexes are strongly connected components of Steinberg's Schützenberger complex, with the vertices restricted to an $\mathcal{R}$-class.

In a similar manner, Stephen's approximate graphs can be viewed as approximate complexes by sewing on a face each time an elementary expansion is performed, and identifying faces if a determination results in their entire boundaries being identified.

2.4 w-CW complexes

We now define a particular type of CW-complex that is useful in studying inverse monoid presentations of the form $M = \mathrm{Inv}\langle X \mid w = 1\rangle$, with w cyclically reduced. In this setting, the 1 side of the relation can be read at any vertex, and sewing on the other side means attaching a circuit labeled by w and a face which has the circuit as its boundary.

2.4.1 Cyclic substrings

We use the following notation for referring to words and substrings of words:

Definition 2.6. *Let $w = a_1 \dots a_n \in (X \cup X^{-1})^*$ be a word of length n, where the subscripts are integers mod n. Then for $i \in \mathbb{Z}/n\mathbb{Z}$ and $l \in \mathbb{Z}$, the* **cyclic substring** *$w[i,l]$ at* **index** *i with* **length** *l is defined by*

$$w[i,l] = \begin{cases} a_{i+1} \dots a_{i+l} & \text{if } l > 0 \\ a_i^{-1} \dots a_{i-(-l-1)}^{-1} = a_i^{-1} \dots a_{i+l+1}^{-1} & \text{if } l < 0 \\ \epsilon \text{ (the empty string)} & \text{if } l = 0 \end{cases}.$$

We immediately establish the following useful facts for substrings:

Proposition 2.7. *(Properties of Cyclic Substrings) If $w \in (X \cup X^{-1})^*$ then for $i \in \mathbb{Z}/|w|\mathbb{Z}$ and $l \in \mathbb{Z}$,*

(1) $w[i,l] = w^{-1}[-i,-l] = w[i+l,-l]^{-1}$, and

(2) $w[i,\eta l] = \prod_{j=0}^{l-1} w[i+\eta j, \eta]$ for $l \geq 0$ and $\eta = \pm 1$.

Proof. Let $w = a_1 \dots a_n$. Then $w^{-1} = a_n^{-1} \dots a_1^{-1} = b_1 \dots b_n$, where $b_j = a_{n+1-j}^{-1} = a_{-j+1}^{-1}$, since the letters are indexed mod n. If $l > 0$, it follows that $w^{-1}[-i,-l] = b_{-i}^{-1} \dots b_{-i-l+1}^{-1} = (a_{-(-i)+1}^{-1})^{-1} \dots (a_{-(-i-l+1)+1}^{-1})^{-1} = a_{i+1} \dots a_{i+l} = w[i,l]$, and $w[i+l,-l]^{-1} = \left(a_{i+l}^{-1} \dots a_{i+l-l+1}^{-1}\right)^{-1} = \left(a_{i+l}^{-1} \dots a_{i+1}^{-1}\right)^{-1} = a_{i+1} \dots a_{i+l} = w[i,l]$. Similar arguments show (1) holds for $l < 0$ as well. For (2), if $\eta = +1$ we have $w[i,l] = a_{i+1} \dots a_{i+l} = w[i,1]w[i+1,1] \dots w[i+l-1,1] = \prod_{j=0}^{l-1} w[i+j,1]$, and if $\eta = -1$ then $w[i,-l] = a_i^{-1} \dots a_{i-l+1}^{-1} = w[i,-1]w[i-1,-1] \dots w[i-l+1,-1] = \prod_{j=0}^{l-1} w[i-j,-1]$. □

Proposition 2.8. *If $w \in (X \cup X^{-1})^*$ is cyclically reduced, then $w[i,l] \neq w[i,l]^{-1}$ for all $i \in \mathbb{Z}/|w|\mathbb{Z}$ and $l \in \mathbb{Z}$, $l \neq 0$.*

Proof. Since $w[i,l] = w^{-1}[i,-l]$, it suffices to prove this for $l > 0$. Let $w = a_0 \dots a_{n-1}$. If $w[i,l] = w[i,l]^{-1}$, then $a_i \dots a_{i+l-1} = a_{i+l-1}^{-1} \dots a_i^{-1}$, so $a_{i+j} = a_{i+l-1-j}^{-1}$ for $j =$

$0, \ldots, l-1$. If l is even, let $j = l/2$. Then $a_{i+l/2} = a_{i+j} = a_{i+l-1-j}^{-1} = a_{i+l/2-1}^{-1}$, so w contains a substring xx^{-1} for $x = a_{i+l/2-1}$, which contradicts w being cyclically reduced. If l is odd, let $j = (l-1)/2$. Then $a_{i+(l-1)/2} = a_{i+j} = a_{i+l-1-j}^{-1} = a_{i+(l-1)/2}^{-1}$, which contradicts the fact that $X \cap X^{-1} = \emptyset$. □

2.4.2 w-CW complexes

In this section, we define the restricted notion of CW-complex for building Schützenberger complexes of $M = \mathrm{Inv}\langle X \mid w = 1\rangle$. First, we describe the structure of the faces:

Let $w \in (X \cup X^{-1})^*$ be a cyclically reduced word of length n. Let P be a regular n-gon regarded as a CW-complex, with n vertices, n edges and a single face. Let O_P be a distinguished vertex of P, and orient and label the edges of P so that w is read clockwise from O_P to O_P in the boundary of P. We call P the **building block polygon** of w.

Definition 2.9. *Let $w \in (X \cup X^{-1})^*$ be a cyclically reduced word of length n, and let P be the building block polygon of w. A w-**CW complex** is a triple (S, O, σ), such that*

- *S is a 2-dimensional CW-complex in which each edge has an orientation and a label $x \in X \cup X^{-1}$,*
- *$O \in V(S)$ is a vertex designated as the **origin** or **initial vertex**,*
- *for each face $F \in F(S)$ there is a cellular map $\lambda_F \colon P \to F$, called the **labeling map** of F, which respects the orientations and labels of the edges, and*
- *the **start vertex map** $\sigma \colon F(S) \to V(S)$ is the map such that for each $F \in F(S)$, $\sigma(F) = \lambda_F(O_P)$.*

When there is no danger of confusion, the name S will be used for both the w-CW complex as well as the underlying CW-complex.

We see immediately that (P, O_P, σ_P), where λ_P is the identity map and σ_P is the map $P \mapsto O_P$, is a w-CW complex. Note also that the 1-skeleton of P, and hence the 1-skeleton of any w-CW complex, may be viewed as an inverse word graph where the edge labeled by $x \in X \cup X^{-1}$, oriented from vertex v_0 to v_1, gives rise to the edges (v_0, x, v_1) and (v_1, x^{-1}, v_0) in the inverse word graph.

The process of reading w from O_P to O_P around the boundary of P assigns a mod-$|w|$ numbering to the vertices of P corresponding to indices of the word (Definition 2.6). It is frequently useful to refer to this numbering, which is formally defined below:

Definition 2.10. *Let $w = a_1 \dots a_n \in (X \cup X^{-1})^*$ be a cyclically reduced word. Let P be the building block for w, and let*

$$(v_0, a_1, v_1), (v_1, a_2, v_2), \dots, (v_{n-1}, a_n, v_n)$$

be the $n = |w|$ edges comprising the boundary of P, where $v_0 = v_n = O_P$, and the subscripts are integers mod n. We define two maps as follows:

$$v_P \colon \mathbb{Z}/|w|\mathbb{Z} \to V(P) \quad \textit{maps} \quad i \mapsto v_i$$
$$i_P \colon V(P) \to \mathbb{Z}/|w|\mathbb{Z} \quad \textit{maps} \quad v_i \mapsto i$$

In an arbitrary w-CW complex, v_P induces a corresponding map v_F for each face F. If v_F is injective (as is the case in Chapter 3), there is a corresponding map i_F.

The notion of "fully folded" below is analogous to Stephen's "determinized form." It follows immediately that P is a fully folded w-CW complex, since w is cyclically reduced.

Definition 2.11. *Given a w-CW complex (S, O, σ), a triple (v, e_1, e_2) is called a* **folding point** *of S if $v \in V(S)$ and $e_1 = (v_1, x_1, u_1)$ and $e_2 = (v_2, x_2, u_2)$ are distinct edges with $x_1 = x_2$ and either $v = v_1 = v_2$ or $v = u_1 = u_2$. S is called* **fully folded** *if it has no folding points.*

The following is the formal definition of a map between w-CW complexes:

Definition 2.12. *A* **w-CW complex map** *$\phi\colon S \to T$ is a cellular map of w-CW complexes (S, O_S, σ_S) and (T, O_T, σ_T) which is compatible with origins and the labeling maps; that is, $\phi(O_S) = O_T$, and for each $F \in F(S)$, $\lambda_{\phi(F)} = \phi \circ \lambda_F$.*

Note a consequence of this definition is $\phi(\sigma_S(F)) = \sigma_T(\phi(F))$ for all $F \in F(S)$.

2.4.3 Schützenberger approximation sequences

We now consider a method of defining a sequence of fully folded w-CW complexes for the presentation $M = \mathrm{Inv}\langle X \mid w = 1\rangle$, which we call a **Schützenberger approximation sequence**. Intuitively, the complexes in this sequence serve as approximations of $SC(1)$.

Step 0

Let $S_0 = (S_0, O_0, \sigma_0)$ be a w-CW complex consisting of a single vertex O_0 with the empty start vertex map σ_0. A single vertex is trivially an inverse word graph. Also, there are no folding points since there are no edges, so S_0 is a fully folded w-CW complex. Note also that S_0 is the linear graph of the empty word ϵ.

Step i construction

For each $i \geq 1$, S_i is described in the steps below:

(1) **Choose** s_i

Choose $s_i \in V(S_{i-1}) \backslash \sigma_{i-1}(F(S_{i-1}))$—that is, a vertex at which no face has yet been attached—so that $d(O_{i-1}, s_i)$ is as small as possible (where d is the path metric in S_{i-1}). We assume there is such an s_i. If there is no such s_i, then $S_0, \ldots, S_{i-1}$ is the complete sequence of w-CW complexes, and it follows from [Ste90] that S_{i-1} is the Schützenberger complex.

(2) **Construct** S_i^0

Let $S_i^0 = (S_{i-1} \amalg P)/\equiv$, where $\equiv$ is the the equivalence relation with quotient map $\phi_i^0 \colon S_{i-1} \amalg P \to S_i^0$, which identifies the vertices s_i and O_P and nothing else. Define the initial vertex as $O_i^0 = \phi_i^0(O_{i-1})$ and define $\lambda_{[f]} = [\lambda_f(x)]$ for $f \in F(S_{i-1} \amalg P)$ and $x \in P$, where $[f]$ denotes the $\equiv$-class of f.

The edge set and the λ maps are well-defined since $\equiv$ does not identify any edges or faces. Thus S_i^0 is a w-CW complex and $\phi_i^0 : S_{i-1} \amalg P \to S_i^0$ is a w-CW map.

(3) **Construct** S_i^j**,** $j > 0$

Before we proceed to S_i, we build a sequence $S_i^1, \ldots, S_i^{J_i}$ of w-CW complexes, where $S_i^{J_i}$ is a fully folded w-CW complex. (Note that the superscripts are indices, not exponents.)

Let $j \geq 1$, and suppose we have built $S_i^0, \ldots, S_i^{j-1}$. If S_i^{j-1} is fully folded, we set $J_i = j - 1$ and continue to Step 4. If S_i^{j-1} is not fully folded, let $(v, (v, a, u_1), (v, a, u_2))$ or $(v, (u_1, a, v), (u_2, a, v))$ be a folding point. Define $S_i^j = S_i^{j-1}/\sim$, where $\sim$ is an equivalence relation with quotient map ϕ_i^j which identifies $u_1 \sim u_2$ and the two edges. Furthermore, if there are faces A and B with $\sigma_i^{j-1}(A) = \sigma_i^{j-1}(B)$ such that A and B will have identical boundaries after the

two edges are identified (in other words, this edge identification is the last one needed), then $A \sim B$ also.

Define $\lambda_{[f]}(x) = [\lambda_f(x)]$ for $f \in F(S_i^{j-1})$ and $x \in P$. This map is well-defined since the only faces identified are those whose vertices and edges have been identified in a manner which respects the start vertices and the labeling by w. It follows that $\sigma_i^j([f]) = [\sigma_i^{j-1}(f)]$ is well-defined as well. Thus S_i^j is a w-CW complex.

Also, note for the face $F = (\phi_i^j \circ \cdots \circ \phi_i^0)(P)$ attached in this step, $\lambda_F = (\phi_i^j \circ \cdots \circ \phi_i^0)\big|_P : P \to S_i^j$.

(4) **Finish folding and construct** S_i

The process above must eventually terminate in some $S_i^{J_i}$ because each step reduces the number of edges in the w-CW complex (by identifying exactly two of them), and each w-CW complex constructed here has finitely many edges. Now, at the end of step i, we make the following definitions:

$O_i = O_i^{J_i}$, $S_i = S_i^{J_i}$, and $\sigma_i = \sigma_i^{J_i}$

$\phi_{i-1,i} = \phi_i^{J_i} \circ \cdots \circ \phi_i^0 : S_{i-1} \amalg P \to S_i$

$\phi_{k,i} = \phi_{i-1,i}|_{S_{i-1}} \circ \phi_{k,i-1} : S_k \amalg P \to S_i$, for each $k \in \{0, \ldots, i-2\}$, and

$\lambda_{\Pi_k} = \phi_{k-1,i}|_P : P \to \Pi_k$ for each $\Pi_k \in F(S_i)$ such that $\Pi_k = \phi_{k-1,i}(P)$.

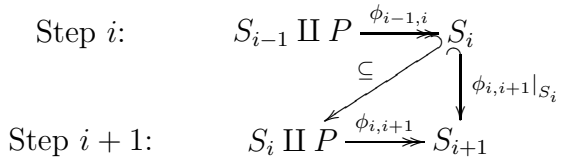
It is clear from how they were constructed that S_i is a fully folded w-CW complex and the maps above are w-CW maps. The construction process continues by returning to Step 1 after incrementing i.

A sequence of w-CW complexes defined in the above manner is called a **Schützenberger approximation sequence**.

Note that the face P attached in step i maps to face $\phi_{i-1,I}(P)$ in step I, and every face F in S_I has a unique i so that $\phi_{i-1,I}(P) = F$. Thus, step i is the step in which

F was created.

The following diagram helps visualize the maps $\phi_{k,i}$ $(k < i)$. Each $\phi_{k,i}$ is a composition of the maps from $S_k \amalg P$ in the middle column of step $k+1$, right to S_{k+1}, and then down along the rightmost column to S_i.



The full Schützenberger complex

The following lemma shows that in a Schützenberger approximation sequence, a face will eventually get attached at any given vertex:

Lemma 2.13. *Let $(S_i)_{i\in\mathbb{N}}$ be a Schützenberger approximation sequence constructed as in Section 2.4.3. Let $v \in V(S_k)$ for some k. Then there is some $I \geq k$ such that $\phi_{k,I}(v) \in \sigma(F(S_I))$; that is, v is the start vertex of some face in step I.*

Proof. If the sequence terminates, it terminates in the full Schützenberger complex, which must have a face attached at every vertex. Suppose the sequence does not terminate. Let $r = d(O_k, v)$. For $i \geq k$, each vertex of S_i can be on at most $2n$ $(n = |w|)$ edges because each edge must be labeled by a letter from w or w^{-1}. Any S_i can therefore have at most $(2n)^r$ vertices within r of O. Let $I = (2n)^r + 1$. Since s_i is always chosen to minimize the distance from O_{i-1}, it follows that every vertex within r of O_I must be the start vertex of some face. Since $\phi_{k,I}(v)$ is within r of O_I, we must have $\phi_{k,I}(v) \in \sigma(F(S_I))$. □

A formal category-theoretical argument similar to that used by Stephen in [Ste87] shows that a Schützenberger approximation sequence has a direct limit. By the above lemma, a circuit labeled by w is attached at every vertex, so the limit is closed (in the

sense of Section 2.2.3). Therefore, by Theorem 2.3, the 1-skeleton of the direct limit of a Schützenberger approximation sequence is the Schützenberger graph of 1. Since the approximation sequence attaches faces whenever a circuit labeled by w is attached, it must also follow that the limit of the Schützenberger approximation sequence is the Schützenberger complex.

Chapter 3

Inverse monoids defined by a sparse relator

We now turn to the particular class of inverse monoids which are the primary focus of this dissertation, those presented by a single cyclically reduced relator, $w = 1$. The first suggestions of work along these lines were observations by Steve Haataja about inverse monoids defined by a relator which is the product of n commutators such as $M = \mathrm{Inv}\langle a, b, c, d \,|\, [a, b][c, d] = 1\rangle = \mathrm{Inv}\langle a, b, c, d \,|\, aba^{-1}b^{-1}cdc^{-1}d^{-1} = 1\rangle$ for $n = 2$. Haataja proved that such monoids are E-unitary and have solvable word problem [Mea93].

Subsequent to Haataja's work, Ivanov, Margolis, and Meakin proved in [IMM01] that in fact all monoids of the form $M = \mathrm{Inv}\langle X \,|\, w = 1\rangle$, with w cyclically reduced, are E-unitary (see Theorem 2.4). Given that M is E-unitary, the goal of this chapter is to present a generalization of the characteristics exhibited by Haataja's product of commutators, namely that the relator w overlaps itself in only a limited or "sparse" way. In subsequent chapters, we prove that in this case M has solvable word problem.

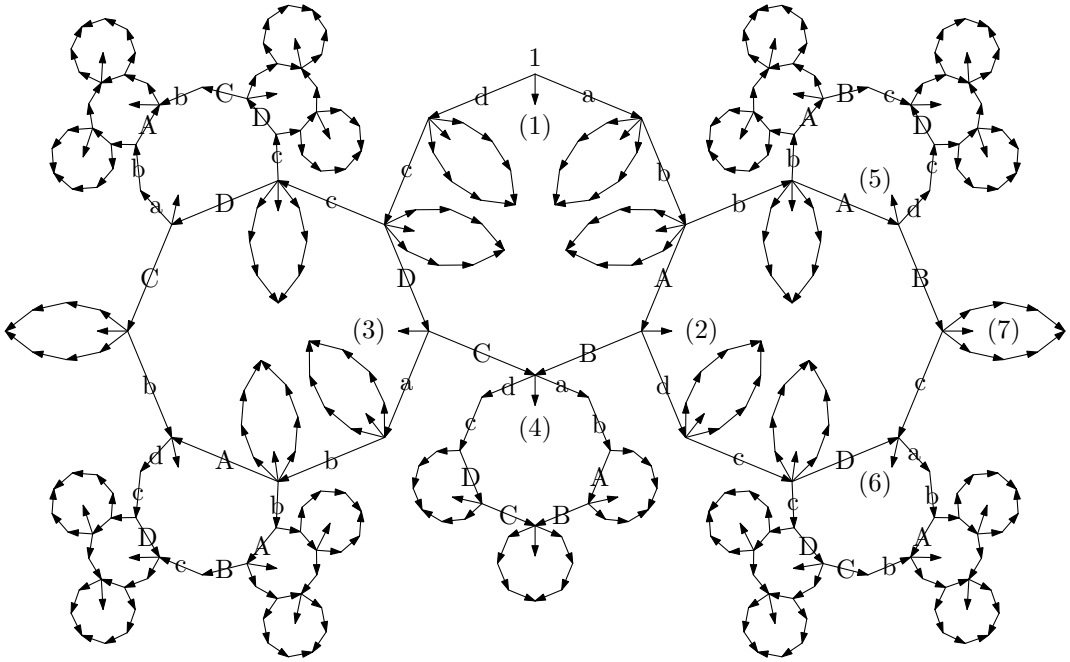


Figure 3.1: Part of $SC(1)$ for $M = \mathrm{Inv}\langle a, b, c, d \,|\, abABcdCD = 1\rangle$

3.1 Relator equal to a product of two commutators

We first consider $M = \mathrm{Inv}\langle a, b, c, d \,|\, abABcdCD = 1\rangle$, our motivating example. A portion of the Schützenberger complex of 1 is presented in Figure 3.1. This illustrates some conventions we make:

(1) Upper-/lower-case letters are used to denote the inverses of their lower-/upper-case counterparts. ($a^{-1} = A$ and $A^{-1} = a$.)

(2) The letter labeling each edge indicates the letter read while traversing the edge in the direction of the arrow. The edge may be traversed in the opposite direction by reading the inverse letter. We either use the "geodesic labeling," with arrows drawn in the geodesic direction away from the origin (as in Figure 3.1), or draw the arrows around the entire face in the direction in which the word w is read.

(3) A small arrow pointing into the interior of the polygon is used to indicate the start point of that face. In Figure 3.1, seven of the faces have been labeled (1) through (7). This numbering is for purposes of the discussion below, and is not part of the Schützenberger complex.

A striking feature of the picture is its tree-like structure. Each face has an unambiguous predecessor and never folds back onto a face other than its predecessor. This is the key property that enables us to solve the word problem.

Some of the faces, such as (4) and (7), do not share edges with the face to which they are attached. Thus these faces, and the faces attached to them, look like another copy of the face (1) and its descendants. This is best seen in face (4), which looks like another copy of the entire picture restarted at $abAB$.

Faces (2) and (3), on the other hand, do share an edge with the first face. The edge shared by (1) and (2) arises because the a edge coming out of the start vertex of face (2) folds back on to the A edge of face (1). As a consequence, the geodesic entry point of face (2) is not at its start vertex abA, but at ab. Thus, the geodesic labeling differs from that of face (1). The geodesic labeling of the faces (2) and (3) differ from each other as well.

We therefore have three main faces, (1), (2), and (3). These are clearly visible as the three largest regular octagons in the picture. Inspection reveals that the other faces in the picture all have geodesic labeling identical to one of these three faces. For example, face (5) is labeled in the same manner as face (2), so face (5) and its descendants are isomorphic to face (2) and its descendants. Thus, as is shown in the next chapter, all faces in the Schützenberger complex are one of these three types. (However, for technical reasons there are actually four face types, the fourth being a trivial "face" type consisting of just the vertex 1.)

What causes the tree-like structure in this example? Consider face (5) again. It is attached to face (2). If w had been such that face (5) folded back onto the b edge from ab to abb, then (5) would also share a vertex with face (1), and it would not be clear whether the predecessor of (5) is (1) or (2). Thus it is the existence of edges like b separating the shared edge of (2) and (5) from the shared edge of (1) and (2) which causes the tree-like structure.

Shared edges in the Schützenberger complex correspond to substrings of the word w which appear twice in w, once at the beginning of the word, and once elsewhere. To guarantee the tree-like structure of the Schützenberger complex, we need all of these segments of w to be separated by at least one letter. When w has this property, we say w is "sparse."

3.2 Sparse words

As is done in group theory, we find it convenient to think of cyclically reduced words as labeling the edges of an n-gon like the building block P of Section 2.4.2, even when we are not talking about a specific face in a Schützenberger complex. As provided by the maps v_P and i_P, the vertices of the n-gon are labeled by the integers mod n so that vertex 0 is the start vertex and w can be read around the n-gon starting at the start vertex.

The word $w = abABcdCD$ is illustrated by the octagon in Figure 3.2 (without the dotted arrows outside the perimeter). Note that here we have chosen to label the edges (i.e., orient the arrows) in the direction of the word w, rather than in the geodesic direction as in Figure 3.1.

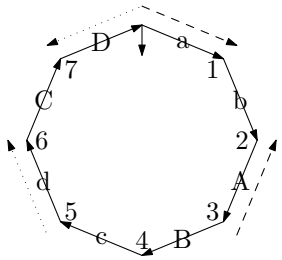


Figure 3.2: Pointed pieces and segments of $w = abABcdCD$

3.2.1 Definitions

Figure 3.2 shows the octagon for $w = abABcdCD$ together with additional markings indicating portions of the word corresponding to edges that are shared with other faces in the Schützenberger complex. Each arrow outside the octagon indicates a **segment** of the word. A pair of segments with the same labeling, at least one of which contains the start vertex, is a **pointed piece** of the word. Thus Figure 3.2 illustrates two pointed pieces, one indicated by the two dashed arrows and the other by the dotted arrows. The segment which contains the start vertex is called the **home segment** and the other segment is the **associated segment**.

Pointed pieces capture the ways in which faces of the Schützenberger complex share edges. One of the segments (the home segment) contains the start vertex because when building the Schützenberger complex of an inverse monoid, we are only allowed to attach new faces at their start vertices.

Formally, we define the notions of segment and pointed piece as follows:

Definition 3.1. *Let $w \in (X \cup X^{-1})^*$ be a cyclically reduced word. A* **segment** *of w is an ordered triple (i, u, η) with $i \in \mathbb{Z}/|w|\mathbb{Z}$, $u \in (X \cup X^{-1})^*$ $(|u| > 0)$ and $\eta = \pm 1$, such that $u = w[i, \eta|u|]$. If q is a segment with $q = (i, u, \eta)$, we write u_q for u, i_q for i and η_q for η for notational convenience.*

If q and r are segments of w, then q is **contained in** *r, written $q \preceq r$ if*

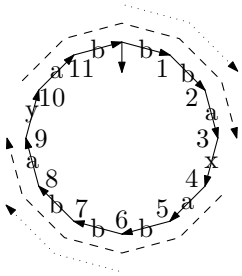


Figure 3.3: A nonmaximal pointed piece (dotted segments)

$z(q) \subseteq z(r)$, *where* $z(q)$ *is the* **zone** *of* q, *defined by* $z(i,u,\eta) := i + \eta\{0,\ldots,|u|\} = \{i, i+\eta, \ldots, i+\eta|u|\}$. *If* $q \preceq r$ *and* $r \preceq q$ *we write* $q \approx r$.

Intuitively, $q \approx r$ means that q and r occupy the same vertices within the building block P for w. For example, $(i,u,1) \approx (i+|u|, u^{-1}, -1)$ because both segments traverse the same vertices ($z(i,u,1) = \{i, i+1, \ldots, i+|u|\} = z(i+|u|, u^{-1}, -1)$), even though the segments have opposite orientation.

Definition 3.2. *Let* w *be a cyclically reduced word. A* **pointed piece** *of* w *is a pair of segments* (q,r), *the* **associated** *and* **home** *segments, respectively, such that*

(1) $u_q = u_r$,

(2) $q \not\approx r$, *and*

(3) $0 \in z(r)$.

The **length** *of the pointed piece is the length of the word* $u_q = u_r$.

This definition does not guarantee that a pointed piece is in any way "maximal." For example, $w = bbaxabbbayab$ has pointed pieces $((10, abbba, 1), (4, abbba, 1))$ and $((0, bb, 1), (6, bb, 1))$, as shown in Figure 3.3, but the later pointed piece is really just the same as the former but with some of the common overlap simply ignored. We would like a way of saying that $((0, bb, 1), (6, bb, 1))$ is contained within

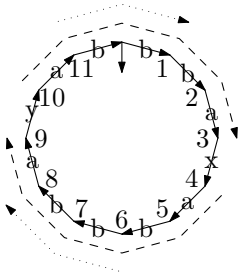


Figure 3.4: Noncomparable pointed pieces

$((10, abbba, 1), (4, abbba, 1))$, so that we may disregard $((0, bb, 1), (6, bb, 1))$ and instead focus our attention on $((10, abbba, 1), (4, abbba, 1))$.

However, it is not sufficient simply to say that one pointed piece is contained in another if its segments are contained within the corresponding segments of the other pointed piece. For example, $((11, bb, 1), (6, bb, 1))$ is a pointed piece of $w = bbaxabbbayab$ whose segments are contained within the segments of $((10, abbba, 1), (4, abbba, 1))$ (see Figure 3.4). Nevertheless, as we can see from the picture, $((11, bb, 1), (6, bb, 1))$ is a maximal pointed piece in the sense that it cannot be enlarged to another pointed piece because the edge immediately before or after the segment $(11, bb, 1)$ does not match the edge immediately before or after $(6, bb, 1)$.

What makes these two examples different is that in Figure 3.3, both pointed pieces have the property that vertex 6 inside the associated segment corresponds to vertex 0 (the start vertex), whereas in the pointed piece $((11, bb, 1), (6, bb, 1))$ (Figure 3.4), it is vertex 7, not 6, that corresponds to the start vertex. Thus, we say that one pointed piece is contained within another if its segments are contained within the corresponding segments of the other pointed piece, and the locations of the start vertices within the associated segments are the same.

Before we make this last statement more precise, consider a pointed piece (q, r). The home segment r starts at vertex i_r. If $\eta_r = +1$, then r is read forward in the

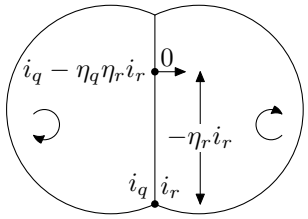


Figure 3.5: Calculation of the start vertex index inside an associated segment

word, so i_r must precede the start vertex 0 (in the orientation given the vertices as w is read around the boundary of the face) and the start vertex 0 must be attained in $-i_r$ steps (that is, in m steps where $m \equiv -i_r \pmod{|w|}$ and $0 \leq m < |w|$). Similarly, if $\eta_r = -1$, then r is read backward, and 0 must be attained in i_r steps. Thus, in either case there are $-\eta_r i_r$ edges from i_r to 0 in segment r. Since the associated segment starts at i_q and either reads forward ($\eta_q = +1$) or backward ($\eta_q = -1$), the vertex inside the associated segment which corresponds to the start vertex must have index $i_q + \eta_q(-\eta_r i_r) = i_q - \eta_q\eta_r i_r$.

This is illustrated in Figure 3.5. In this picture, each circular region represents a copy of the polygon for w. The shared boundary represents the segments of the pointed piece—when viewed as the boundary of the polygon on the left, the shared boundary corresponds to segment q, and when viewed as the boundary of the polygon on the right, it corresponds to segment r. The small curved arrows indicate the direction of the labeling by w inside the polygons. Thus, this picture illustrates the case for $\eta_q = -1$ and $\eta_r = +1$.

We are now prepared to give the formal definition of containment of pointed pieces:

Definition 3.3. *If (q, r) and (s, t) are pointed pieces, then (q, r) is* **contained in** *(s, t), written $(q, r) \preceq (s, t)$, if $q \preceq s$, $r \preceq t$, and $i_q - \eta_q\eta_r i_r = i_s - \eta_s\eta_t i_t$.*

We write $(q, r) \approx (s, t)$ if $(q, r) \preceq (s, t)$ and $(q, r) \succeq (s, t)$. A pointed piece (q, r) is **maximal** *if for all pointed pieces (s, t) with $(s, t) \succeq (q, r)$, it follows that $(s, t) \approx (q, r)$.*

Recall that what gives the Schützenberger complex for $w = abABcdCD$ its tree-like structure is the fact that the places in which faces share edges are far apart, separated by at least an edge. In terms of pointed pieces, this means that no two maximal pointed pieces overlap, or intersect, each other. Maximal pointed pieces must be used, otherwise we would view pointed pieces such as those in Figure 3.3 as overlapping, when they really just represent the same pair of faces in the Schützenberger complex, at different stages of folding. Similarly, overlaps of associated segments only count if they are from different pointed pieces, because the associated segment of a pointed piece is always going to intersect itself. On the other hand, an intersection of the home and associated segments of a single pointed piece is a genuine overlap.

Definition 3.4. *Let w be cyclically reduced word, and (q, r) and (s, t) be maximal pointed pieces of w. Then (q, r) and (s, t)* **overlap** *if*

- *$z(q) \cap z(t)$ or $z(r) \cap z(s)$ is nonempty, or*
- *$z(q) \cap z(s)$ is nonempty and $(q, r) \not\approx (s, t)$.*

This finally brings us to our definition of sparse. We specifically exclude words of length one and words which are proper powers (i.e., $w = u^n$ for some $u \in (X \cup X^{-1})^*$ and $n > 1$) to avoid technical problems with the definitions. A relator of length one is not of interest (the resulting inverse monoid is isomorphic to the free inverse monoid with an extra letter, the relator, equal to 1). A proper power clearly does not satisfy our intuitive notion of sparse, nor does it give rise to tree-like Schützenberger complexes. For example, a word which is a proper square has "pillows" in its Schützenberger complex, from the boundaries of two different faces folding together completely.

Definition 3.5. *A cyclically reduced word $w \in (X \cup X^{-1})^*$ is* **sparse** *if it has length greater than* 1*, is not a proper power, and has no maximal pointed pieces which overlap.*

3.2.2 Examples of sparse and nonsparse words

We now summarize the examples we discussed above, and present some others:

Example 3.1 *(Product of two commutators: $w = abABcdCD$).* From Figure 3.2 it is easy to see that the maximal pointed pieces of $w = abABcdCD$ are $((3, a, -1), (0, a, 1)) \approx ((2, A, 1), (1, A, -1))$ and $((5, d, 1), (0, d, -1)) \approx ((6, D, -1), (7, D, 1))$. These do not overlap, so w is sparse.

Example 3.2 *($w = bbaxabbbayab$).* From Figure 3.4 we can see that $((6, bb, 1), (11, bb, 1))$ and $((4, abbba, 1), (10, abbba, 1))$ are maximal pointed pieces which overlap, so w is not sparse.

Example 3.3 *(Product of three commutators: $w = abABcdCDefEF$).* This word is sparse for basically the same reason $abABcdCD$ is sparse. To prove it is sparse, we must find all maximal pointed pieces and verify that they do not overlap. We know that the home segment of a pointed piece must contain an edge and it must contain the start vertex. Therefore, the letter $w[0, +1] = a$ or $w[0, -1] = f$ (or its inverse) must appear in any pointed piece. The only places at which these letters can be read is vertices 3 and 9 (see Figure 3.6). It is easy to see then that $((3, a, -1), (0, a, 1)) \approx ((2, A, 1), (1, A, -1))$ and $((9, f, 1), (0, f, -1)) \approx ((10, F, -1), (11, F, 1))$ are the only maximal pointed pieces of w. These do not overlap, so w is sparse.

Example 3.4 *(One commutator: $w = abAB$).* We can see in Figure 3.7 that $((3, a, -1), (0, a, 1))$ and $((1, b, 1), (0, b, -1))$ are maximal pointed pieces. The home segment $(0, a, 1)$ and the associated segment $(1, b, 1)$ both contain vertex 1, so w is not sparse.

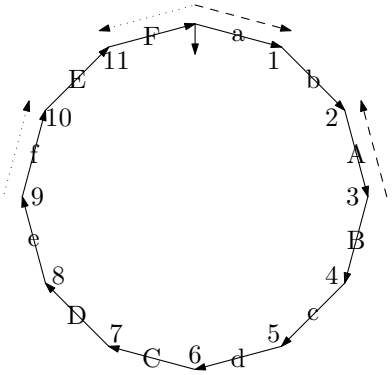


Figure 3.6: Product of three commutators: $w = abABcdCDefEF$

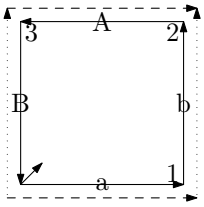


Figure 3.7: One commutator: $w = abAB$

This agrees with the fact that the Schützenberger complex for w (the upper-right quadrant of a rectangular grid) is not tree-like. The inverse monoid presented by this relator has an easily solvable word problem, but it does not fall within the class of inverse monoids being studied in this paper.

Example 3.5 *($w = abcxaxabxabcy$).* From Figure 3.8, we can see that the maximal pointed pieces of this word are

$$((4, a, 1), (0, a, 1)),\ ((6, ab, 1), (0, ab, 1)),\ ((9, abc, 1), (0, abc, 1)),$$

and their equivalent pointed pieces reading in the opposite direction. None of these pointed pieces overlap, so w is sparse.

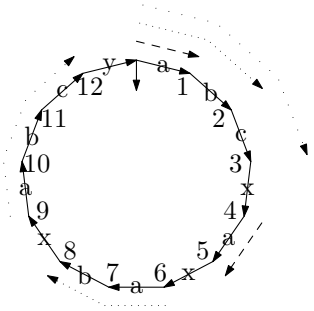


Figure 3.8: $w = abcxaxabxabcy$

3.3 The Schützenberger complex

In this section we prove the key lemma of this dissertation. This lemma captures the intuition that the Schützenberger complex for a sparse word is built in a very orderly way, always growing away from the origin. Specifically, each time a new face is attached and folds onto the existing complex (if it folds at all), its start vertex must belong to only one face (the face that is its predecessor in the dual graph, to be defined in Section 3.4), the edges onto which it folds will belong only to that predecessor face, and it will not, at any future step, fold further on to the existing complex. Since this lemma closely examines the order in which faces are attached and edges are folded, explicit use of the ϕ maps is made in the statement and proof. Once we have proved this lemma we will no longer have a need to use the ϕ maps explicitly.

In stating and proving this lemma, we define the **path in P corresponding to a segment** r to be the edge path p_r which is labeled by u_r from vertex $v_P(i_r)$ to vertex $v_P(i_r + \eta_r|u_r|)$ in P. Also recall the notation S_I^j, S_I, ϕ_I^j, and $\phi_{k,I}$ defined in Section 2.4.3.

Lemma 3.6. *Let w be sparse, and suppose $(S, O, \sigma) = S_I$ is constructed from a Schützenberger approximation sequence of length I. Let $\Pi_1, \ldots, \Pi_I$ be the faces of S,*

with face Π_i *created in step* i *(i.e.,* $\Pi_i = \phi_{i-1,I}(P)$*).*

Then $0 \leq J_I < |w|$ *and*

(1) *If the number of edge foldings* J_I *is positive, then there is a unique face* $F \in S_{I-1}$ *containing the start vertex* s_I *of* Π_I*. For* $0 \leq j \leq J_I$ *we define* $S^j = (\phi_I^j \circ \cdots \circ \phi_I^0)(S_{I-1})$*,* $F^j = (\phi_I^j \circ \cdots \circ \phi_I^0)(F)$*, and* $P^j = (\phi_I^j \circ \cdots \circ \phi_I^0)(P)$*. (Note that* $\Pi_I = P^{J_I}$ *and note the distinction between* S^j *and* S_I^j*:* S^j *is the image of* S_{I-1} *in* S_I^j *and contains* F^j *but not* P^j*. Thus,* $S_I^j = S^j \cup P^j$*.)*

Then for $0 \leq j \leq J_I$*:*

(a) *The map* ϕ_I^j *is an injective* w*-CW map when restricted to* S^{j-1} *or* P^{j-1}*. Thus, the maps*

$$(\phi_I^j \circ \cdots \circ \phi_I^0)\big|_{S_{I-1}} : S_{I-1} \to S_I^j \text{ and } \lambda_{P^j} := (\phi_I^j \circ \cdots \circ \phi_I^0)\big|_P : P \to S_I^j$$

are injective, and S^j *and* P^j *are fully folded* w*-CW complexes.*

(b) $S^j \cap P^j = F^j \cap P^j$*, and this intersection is* $\sigma(P^j)$ *if* $j = 0$*, or there is a pointed piece* (q, r) *of length* j *such that* $S^j \cap P^j = \lambda_{F^j}(p_q) = \lambda_{P^j}(p_r)$*.*

(c) P^j *shares no edges with a face other than* F^j*.*

(2) *For all* k*,* $0 \leq k < I$*, the maps* $\phi_{k,I}|_{S_k} : S_k \to S_I$ *and* $\lambda_{\Pi_{k+1}} = \phi_{k,I}|_P : P \to S_I$ *are injective.*

(3) *If* $a < b$ *and* Π_a *and* Π_b *share a vertex, then* $\sigma(\Pi_b) \in V(\Pi_a)$ *and* $\sigma(\Pi_a) \notin V(\Pi_b)$*. Consequently, the start vertex map* $\sigma\colon F(S_I) \to V(S_I)$ *is injective.*

(4) *For any two faces* Π_a *and* Π_b *with* $a < b$*, either* $\Pi_a \cap \Pi_b$ *is empty,* $\Pi_a \cap \Pi_b$ *is the vertex* $\sigma(\Pi_b)$*, or there is a maximal pointed piece* (q, r) *(of length* J_b*) such that* $\Pi_a \cap \Pi_b = \lambda_{\Pi_a}(p_q) = \lambda_{\Pi_b}(p_r)$*.*

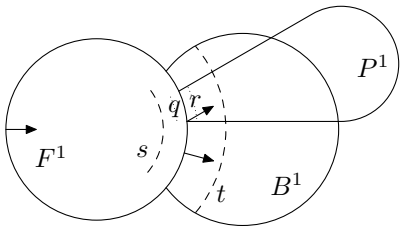


Figure 3.9: Unique F—no other face B can exist

(5) Π_b *shares edges with at most one face in* $\phi_{b-1,I}(S_{b-1})$*. Thus, each edge is on the boundary of at most two faces.*

Proof. We prove this by induction on I. The case $I = 1$ (a single face) holds because w is cyclically reduced and so there can be no edge foldings within a single face. Suppose $I \geq 2$ and the lemma holds for all Schützenberger approximation sequences of length less than I.

Part 1: If $J_I > 0$, then S_I^0 must contain a folding point. Since S^0 and P^0 are fully folded, the only possible folding point in S_I^0 is (s_I^0, e_1, e_2), where $s_I^0 = \phi_I^0(s_I) = \phi_I^0(O_P)$, $e_1 \in E(S^0)$ and $e_2 \in E(P^0)$. Note that ϕ_I^0 is an injective w-CW map on S_{I-1} and on P since it does nothing other than identify the vertices $s_I \in S_{I-1}$ and $O_P \in P$.

Unique F**:** To see that there is exactly one face F in S_{I-1} which contains s_I, note first that every vertex in S_{I-1} belongs to a face (since $I - 1 \geq 1$), so there is at least one F. To see that there is only one, suppose B were another. By induction, either $\sigma_{I-1}(B) \in F$ or $\sigma_{I-1}(F) \in B$ (Part 3 of the lemma). Without loss of generality, suppose $\sigma_{I-1}(B) \in F$. Since s_I^0 is the start vertex of P^0, it follows from the construction that no face starts at s_I in S_{I-1}, so s_I is not the start vertex of B. Thus F and B share a vertex in addition to s_I. Therefore (by induction, Part 4) there is a maximal pointed piece (s, t) such that $F \cap B = \lambda_F(p_s) = \lambda_B(p_t)$.

(An example of faces satisfying this setting is illustrated in Figure 3.9. Note this figure illustrates the faces in S^1, after the edges e_1 and e_2 have been identified. The

edge $\phi_I^1(e_1) = \phi_I^1(e_2)$ is the one labeled on each side by q and r. The dashed arcs labeled s and t indicate segments inside faces F^1 and B^1, respectively. That is, the dashed arcs are to be interpreted as indicating the fact that $F^1 \cap B^1 = \lambda_{F^1}(p_s) = \lambda_{B^1}(p_t)$. Finally, note that in this figure, P^1 and B^1 share no edges or vertices other than those they also share with F^1. One should view P^1 as coming out of the page.)

Continuing with the proof, we know that by assumption e_1 and e_2 will be identified by ϕ_I^1, so there is a pointed piece (q, r) such that $\phi_I^1(e_1) = \phi_I^1(e_2) = \lambda_{F^1}(p_q) = \lambda_{P^1}(p_r)$ and $0 \in z(r)$. Since $\sigma_{I-1}(B) \neq s_I$, the indices of s_I and $\sigma_{I-1}(B)$ in F do not match; that is, $i_q - \eta_q \eta_r i_r \neq i_s - \eta_s \eta_t i_t$. Thus $(q, r) \not\preceq (s, t)$. We may therefore expand (q, r) to a maximal pointed piece (q', r') with $(q, r) \preceq (q', r')$ and $(q', r') \not\approx (s, t)$. Since $\lambda_F(p_q) \cap \lambda_F(p_s)$ is nonempty, $p_q \cap p_s$ is nonempty (since λ is injective), so $z(q) \cap z(s)$ and hence $z(q') \cap z(s)$ are nonempty. Thus the maximal pointed pieces (q', r') and (s, t) overlap, and w is not sparse. This contradicts the fact that w is sparse, so there must be exactly one face F in S_{I-1} containing s_I.

Induction on j: Now we prove (1a)–(1c) using a subinduction on j (holding I fixed). Statements (1a)–(1c) hold for $j = 0$ since ϕ_I^0 is an injective w-CW map on S_{I-1} and P, and $S^0 \cap P^0 = F^0 \cap P^0 = \{\phi_I^0(O_P)\}$.

Now suppose (1a)–(1c) are true for j. If $j = J_I$, we are done. We need to prove that the $(j+1)$th statement is true if $0 \leq j < J_I$. Before doing so, we first define some additional notation and prove two claims.

Since $j < J_I$, there is some folding point (v, e_1, e_2) in S_I^j (e_1 and e_2 may not be the same as e_1 and e_2 used in the Unique F proof above). Since S^j and P^j are fully folded (by the induction on j), neither S^j nor P^j can contain both e_1 and e_2. Thus, we may assume $e_1 \in E(S^j) \backslash E(P^j)$, $e_2 \in E(P^j) \backslash E(S^j)$, and $v \in V(S^j \cap P^j) = V(F^j \cap P^j)$.

If $j = 0$ then v is the single point of $F^0 \cap P^0$. Since F^0 is the only face of S^0 which contains v, $e_1 \in F^0$.

If $j > 0$, then $S^j \cap P^j = \lambda_{F^j}(p_q) = \lambda_{P^j}(p_r)$ for some pointed piece (q, r) of length j. It follows that

$$\begin{aligned} & \lambda_{F^j}(\Big(v_P(i_q + k\eta_q), w[i_q + k\eta_q, \eta_q], v_P(i_q + (k+1)\eta_q)\Big)) \\ = \quad & \lambda_{P^j}(\Big(v_P(i_r + k\eta_r), w[i_r + k\eta_r, \eta_r], v_P(i_r + (k+1)\eta_r)\Big)) \end{aligned}$$

for $k = 0, \ldots, j-1$ (that is, F^j and P^j already share the edges containing the $j-1$ interior vertices), so $v = \lambda_{F^j}(v_P(i_q + k\eta_q)) = \lambda_{P^j}(v_P(i_r + k\eta_r))$ for either $k = 0$ or $k = j$. Since for any face, $\lambda(p_{(i,u,\eta)}) = \lambda(p_{(i+\eta|u|,u^{-1},-\eta)})$ (the two segments have the same zones and corresponding paths simply traverse the same vertices in opposite directions), we may assume without loss of generality that $k = j$. Therefore,

$$e_2 = \lambda_{P^j}(\Big(v_P(i_r + j\eta_r), w[i_r + j\eta_r, \eta_r], v_P(i_r + (j+1)\eta_r)\Big)).$$

Let v_1 and v_2 be the other end points of e_1 and e_2. Then $v_2 = \lambda_{P^j}(v_P(i_r + (j+1)\eta_r))$. It follows that $e_1 = (v, x, v_1)$ is an edge in S^j labeled by $x = w[i_q + j\eta_q, \eta_q] = w[i_r + j\eta_r, \eta_r]$. Since

$$\lambda_{F^j}(\Big(v_P(i_q + j\eta_q), w[i_q + j\eta_q, \eta_q], v_P(i_q + (j+1)\eta_q)\Big))$$

is another such edge in S^j starting at v, and S^j is fully folded, it must be the case that

$$e_1 = \lambda_{F^j}(\Big(v_P(i_q + j\eta_q), w[i_q + j\eta_q, \eta_q], v_P(i_q + (j+1)\eta_q)\Big)),$$

so $e_1 \in F^j$ and $v_1 = \lambda_{F^j}(v_P(i_q + (j+1)\eta_q))$.

Claim 1: $v_1 \notin P^j$ and $v_2 \notin S^j$

Proof of Claim 1: Suppose $v_1 \in P^j$. Since $e_1 \notin P^j$, $F^j \cap P^j$ must be all of the

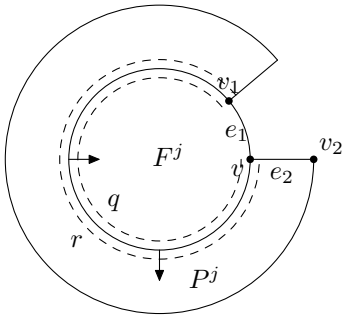


Figure 3.10: Proof of Claim 1

boundary of F^j except e_1, and $j = |w| - 1$ and $v_1 = \lambda_{F^j}(v_P(i_q))$. (An example of this is illustrated in Figure 3.10.) Since e_1 and e_2 will be folded together by ϕ_I^{j+1}, we have segments $q' = (i_q, w[i_q, \eta_q|w|], \eta_q)$ and $r' = (i_r, w[i_r, \eta_r|w|], \eta_r)$ with $z(q') = z(r') = \mathbb{Z}/n\mathbb{Z}$. It follows that (q', r') and (r', q') are pointed pieces, and they are necessarily maximal since their zones contain all $|w|$ indices. Since $z(q') \cap z(q')$ is nonempty, these pointed pieces overlap. This contradicts w being sparse, so we must have $v_1 \notin P^j$.

Similarly, if $v_2 \in S^j$ then $v_2 \in S^j \cap P^j = F^j \cap P^j$ by induction and (1b), so $v_2 \in F^j$. An argument similar to the one above leads to a contradiction. Thus Claim 1 is true.

Note that it also follows from the proof of Claim 1 that $J_I < |w|$.

Claim 2: $e_1 \notin B^j$ for all $B^j \in S^j$, $B^j \neq F^j$.

Proof of Claim 2: If $j = 0$ then the vertex of the folding point (v, e_1, e_2) is $v = \phi_I^0(s_I) = \phi_I^0(O_P)$. Since F^0 is the only face containing s_I, it must be the only face containing e_1.

If $j > 0$ and $e_1 \in B^j$ for some $B^j \neq F^j$, then F^j and B^j share e_1. By induction, there is a maximal pointed piece (s, t) such that

- Case I: $\sigma(B^j) \in F^j$ and $F^j \cap B^j = \lambda_{F^j}(p_s) = \lambda_{B^j}(p_t)$, or

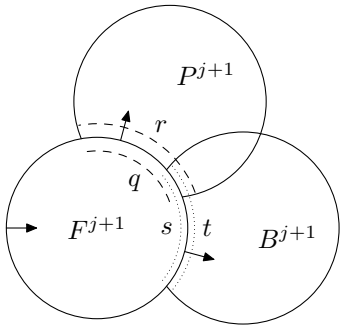


Figure 3.11: Proof of Claim 2 Case I

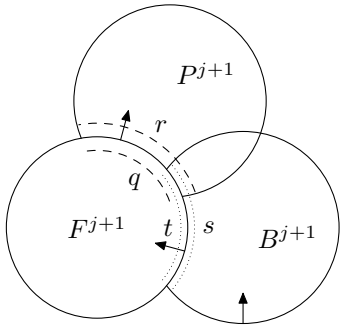


Figure 3.12: Proof of Claim 2 Case II

- Case II: $\sigma(F^j) \in B^j$ and $F^j \cap B^j = \lambda_{B^j}(p_s) = \lambda_{F^j}(p_t)$.

Case I: By an argument similar to the argument above used to prove that F is unique, (q, r) may be extended to a maximal pointed piece (q', r') which overlaps (s, t). Since this contradicts the fact that w is sparse, Case I cannot occur. (An example of this case is illustrated in Figure 3.11. The figure shows the configuration in S^{j+1}, after e_1 and e_2 have been folded together.)

Case II: Again, we may expand (q, r) to a maximal pointed piece (q', r'). Since $v \in F^{j+1} \cap B^{j+1} = \lambda_{F^{j+1}}(p_t)$ and $v \in F^{j+1} \cap P^{j+1} \subseteq \lambda_{F^{j+1}}(p_{q'})$, $\lambda^{-1}_{F^{j+1}}(v) \in p_{q'} \cap p_t$, and so $z(q') \cap z(t)$ is nonempty. Therefore (q', r') and (s, t) overlap. (See Figure 3.12 for an example of this case.)

Since neither of these cases can occur, there can be no face B^j, other than F^j,

which contains e_1. This proves Claim 2.

We can now finish the proof of Part 1 of Lemma 3.6. Recall that we are assuming the jth Part 1 statement is true, and want to prove the $j+1$ statement. The statements of the parts are repeated below with "j" replaced with "$j+1$" to make it clear what we are proving.

Part 1(a): *The map ϕ_I^{j+1} is an injective w-CW map when restricted to S^j or P^j. Thus, the maps $(\phi_I^{j+1} \circ \cdots \circ \phi_I^0)\big|_{S_{I-1}} : S_{I-1} \to S_I^{j+1}$ and $(\phi_I^{j+1} \circ \cdots \circ \phi_I^0)\big|_P : P \to S_I^{j+1}$ are injective, and S^{j+1} and P^{j+1} are fully folded w-CW complexes.*

The only way ϕ_I^{j+1} would not be injective when restricted to S^j or P^j is if it identified two points from S^j or two points from P^j. By definition, ϕ_I^{j+1} identifies $v_1 \in S^j$ with $v_2 \in P^j$ and $e_1 \in S^j$ with $e_2 \in P^j$. Since $v_2 \notin S^j$ (Claim 1) and $e_2 \notin S^j$, ϕ_I^{j+1} must be injective on S^j. Similarly, since $v_1 \notin P^j$ and $e_1 \notin P^j$, ϕ_I^{j+1} is injective on P^j. Since $(\phi_I^j \circ \cdots \circ \phi_I^0)\big|_{S_{I-1}}$ and $(\phi_I^j \circ \cdots \circ \phi_I^0)\big|_P$ are injective by induction, the result follows.

Part 1(b): *$S^{j+1} \cap P^{j+1} = F^{j+1} \cap P^{j+1}$, and this intersection is $\sigma(P^{j+1})$ if $j+1 = 0$, or there is a pointed piece (q', r') of length $j+1$ such that $S^{j+1} \cap P^{j+1} = \lambda_{F^{j+1}}(p_{q'}) = \lambda_{P^{j+1}}(p_{r'})$.*

Here $j \geq 0$ so $j+1 > 0$ is the only case to consider. We have $S^{j+1} \cap P^{j+1} = \phi_I^{j+1}(S^j \cap P^j) \cup \phi_I^{j+1}(e_1)$. Since $e_1 \in F^j$, $\phi_I^{j+1}(e_1) \in F^{j+1}$. Also, $S^j \cap P^j \subseteq F^j \cap P^j$ (by induction), so $\phi_I^{j+1}(S^j \cap P^j) \subseteq F^{j+1} \cap P^{j+1}$. It follows that $S^{j+1} \cap P^{j+1} = \phi_I^{j+1}(S^j \cap P^j) \cup \phi_I^{j+1}(e_1) \subseteq F^{j+1} \cap P^{j+1}$. The reverse inclusion is obvious, so $S^{j+1} \cap P^{j+1} = F^{j+1} \cap P^{j+1}$.

By the comments preceding Claim 1, it follows that if $q' = (i_q, w[i_q, (j+1)\eta_q], \eta_q)$ and $r' = (i_r, w[i_r, (j+1)\eta_r], \eta_r)$, then (q', r') is a pointed piece of length $j+1$ and

$$F^{j+1} \cap P^{j+1} = \lambda_{F^{j+1}}(q') = \lambda_{P^{j+1}}(r').$$

Part 1(c): *P^{j+1} shares no edges with a face other than F^{j+1}.*

We know P^{j+1} and F^{j+1} share the edge $\phi_I^{j+1}(e_1)$. By Claim 2, there is no face B^j which shares e_1 with F^j, so there is no other face of S_I^{j+1} which shares $\phi_I^{j+1}(e_1)$. By induction, P^j shares no edge with a face other than F^j. Thus P^{j+1} shares no edge with a face other than F^{j+1}.

This completes the induction on j. We can now complete the remaining parts of Lemma 3.6:

Part 2: *For all k, $0 \leq k < I$, the maps $\phi_{k,I}|_{S_k} : S_k \to S_I$ and $\lambda_{\Pi_{k+1}} = \phi_{k,I}|_P : P \to S_I$ are injective.*

By Part 1, $\phi_{I-1,I} = \phi_I^{J_I} \circ \cdots \circ \phi_I^0$ is injective when restricted to S_{I-1} or P. Thus Part 2 is true for $k = I-1$. For $k < I-1$, we have $\phi_{k,I} = \phi_{I-1,I}|_{S_{I-1}} \circ \phi_{k,I-1}$. Since $\phi_{I-1,I}|_{S_{I-1}}$ is injective as noted above, and $\phi_{k,I-1}$ is injective when restricted to S_k or P by induction, it follows that $\phi_{k,I}$ is injective when restricted to S_k or P.

Part 3: *If $a < b$ and Π_a and Π_b share a vertex, then $\sigma(\Pi_b) \in V(\Pi_a)$ and $\sigma(\Pi_a) \notin V(\Pi_b)$. Consequently, the start vertex map $\sigma\colon F(S_I) \to V(S_I)$ is injective.*

If $b < I$ then this follows by induction on I. If $b = I$ then $\Pi_b = P^{J_I}$ and $\Pi_a = F^{J_I}$, and so $\sigma(P^{J_I}) \in V(F^{J_I})$. It must also be the case that $\sigma(F^{J_I}) \notin V(P^{J_I})$, for otherwise we would obtain pointed pieces which overlap, in the manner described in Claim 1.

Part 4: *For any two faces Π_a and Π_b with $a < b$, either $\Pi_a \cap \Pi_b$ is empty, $\Pi_a \cap \Pi_b$ is the vertex $\sigma(\Pi_b)$, or there is a maximal pointed piece (q, r) (of length J_b) such that $\Pi_a \cap \Pi_b = \lambda_{\Pi_a}(p_q) = \lambda_{\Pi_b}(p_r)$.*

If $b < I$ then as in Part 2 this follows by induction and the fact that $\phi_{I-1,I}|_{S_{I-1}}$ is injective. If $b = I$ then either $\Pi_a \cap \Pi_b$ is empty, or $\Pi_b = P^{J_I}$ and $\Pi_a = F^{J_I}$. In the latter case, if $J_I = 0$ then there were no folding points and $\Pi_a \cap \Pi_b$ is the single vertex $\phi_I^0(s_I) = \sigma(\Pi_b)$. If $J_I > 0$ then by Part 1(b), there is a pointed piece (q, r) of length J_I such that $\Pi_a \cap \Pi_b = \lambda_{\Pi_a}(p_q) = \lambda_{\Pi_b}(p_r)$. Since J_I is the full number of

foldings, (q, r) must be a maximal pointed piece.

Part 5: Π_b *shares shares edges with at most one face in* $\phi_{b-1,I}(S_{b-1})$. *Thus, each edge is on the boundary of at most two faces.*

If $b < I$ then this follows by induction and the fact that $\phi_{I-1,I}|_{S_{I-1}}$ is injective. If $b = I$ then either $J_I = 0$ and Π_b shares no edges with another face, or $J_I > 0$ and $\Pi_b = P^{J_I}$ shares edges with only F^{J_I} by Part 1(c).

This completes the proof of Lemma 3.6. □

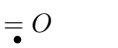
In the proof of Lemma 3.6 we used the ϕ maps carefully to examine the Schützenberger approximations at each stage of sewing on faces and folding. Lemma 3.6 shows that earlier approximations embed in later ones. From this point forward, we will abuse the notation slightly by dropping the ϕ maps and viewing earlier approximations as actually being contained within later approximations: $S_0 \subseteq S_1 \subseteq S_2 \subseteq \ldots$, so the (full) Schützenberger complex is $S = \bigcup_{i=0}^{\infty} S_i$.

We are now in a position to prove the Catalog Theorem below:

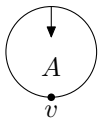
Theorem 3.7 (Catalog of Vertex Star Sets for Approximations). *Let w be sparse and let $M = \mathrm{Inv}\langle X \mid w = 1\rangle$. Let $(S, O, \sigma) = S_I$ be a Schützenberger approximation containing I faces. Given some vertex $v \in V(S)$, let m denote the number of faces which contain v. The cases below indicate the possible configurations for* $\mathrm{Star}(v)$. *The pictures in each case depict precisely the vertices, edges, and faces containing v, and the edges and vertices contained in a face containing v. There is no sharing of edges or vertices of the faces in* $\mathrm{Star}(v)$ *other than explicitly indicated.*

(1) If no face starts at v then $0 \leq m \leq 2$ and the star set is one of these:

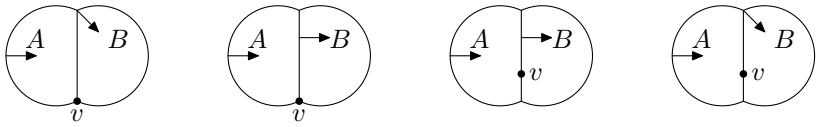
- $m = 0$*: This can only happen if $I = 0$, so S_I is the single vertex O.*



- $m = 1$: *Vertex v is on exactly one face A with $v \neq \sigma(A)$.*

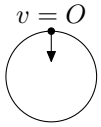


- $m = 2$: *Vertex v is on two faces A and B, $\sigma(B) \in A \cap B$, $\sigma(A) \notin A \cap B$, $v \neq \sigma(B)$ and there is a maximal pointed piece (q, r) such that $A \cap B = \lambda_A(p_q) = \lambda_B(p_r)$. There are four possibilities, depending on whether v and $\sigma(B)$ each is at an endpoint or in the interior of $A \cap B$.*

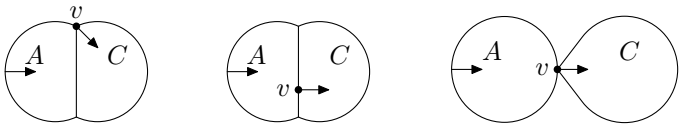


(2) If a face starts at v, then $1 \leq m \leq 3$ and the star set is one of the following:

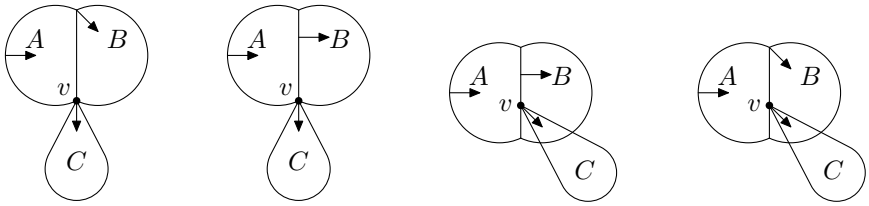
- $m = 1$: *In this case $v = O$ and $\mathrm{Star}(v) = \sigma^{-1}(O)$, the face attached at the initial vertex O.*



- $m = 2$: *Vertex v is on two faces A and C with $v = \sigma(C)$ and $\sigma(A) \notin A \cap C$. Also, either $A \cap C = v$, or there is a maximal pointed piece (q, r) such that $A \cap C = \lambda_A(p_q) = \lambda_C(p_r)$.*



- $m = 3$: *Here, v is on three faces A, B, and C, with $v = \sigma(C) = C \cap A = C \cap B$, $\sigma(B) \in A \cap B$ and $\sigma(A) \notin A \cap B$. Furthermore, there is a maximal pointed piece (q, r) such that $A \cap B = \lambda_A(p_q) = \lambda_B(p_r)$. There are four possibilities, depending on whether v and $\sigma(B)$ are each in the interior or at an endpoint of $A \cap B$.*



Proof. For part (1), assume no face starts at v. When a face is sewn on to a Schützenberger approximation, the new face contains the vertex at which it was attached as well as all new vertices created on the boundary of the face. Hence every vertex is contained in at least one face in any approximation S_I, $I \geq 1$. Thus if $m = 0$, we must have $I = 0$ and v is the sole vertex O of S_0.

If $m = 1$, then since no face starts at v, the face A containing v must start at some other vertex, and we get the configuration shown in part $(1)(m = 1)$.

If $m = 2$, then we have two faces containing v. Let $A = \Pi_a$ and $B = \Pi_b$ be these faces, with $a < b$ (where, as usual, Π_i is the face created in step i). By Lemma 3.6 Part 3, $\sigma(\Pi_b) \in V(\Pi_a)$ and $\sigma(\Pi_a) \notin V(\Pi_b)$. Thus $\sigma(\Pi_b) \neq v$ since no face starts at v. Since $v, \sigma(\Pi_b) \in \Pi_a \cap \Pi_b$, it follows from Lemma 3.6(4) that there is a maximal pointed piece (q, r) such that $\Pi_a \cap \Pi_b = \lambda_{\Pi_a}(p_q) = \lambda_{\Pi_b}(p_r)$. The vertices v and $\sigma(\Pi_b)$ may appear at an end or in the interior of $\Pi_a \cap \Pi_b$, giving rise to the four possibilities in $(1)(m = 2)$.

If $m \geq 3$, let Π_a, Π_b and Π_c be three of the faces containing v, with $a < b < c$. By arguments analogous to that for Π_a and Π_b in the $m = 2$ case above, $\Pi_a \cap \Pi_c$ and $\Pi_b \cap \Pi_c$ must both contain edges. Thus Π_c shares edges with two different faces. This contradicts Lemma 3.6(5). Thus $m \ngeq 3$. This completes the proof of part 1.

For part 2, if a face Π_c starts at v then $m \geq 1$. Since Π_c was attached at v in step c, $v \in V(S_{c-1})$ and there is no face starting at v in S_{c-1}. Thus, Star(v) contains $m - 1$ faces in S_{c-1} and one of the possibilities enumerated in part 1 must apply to v in S_{I-1}:

If $m-1=0$, then $m=1$ and $v=O$, and since Π_c is attached at v, we obtain the configuration shown in $(2)(m=1)$.

If $m-1=1$, then there is a single face $A \in S_{c-1}$ containing v. When the face Π_c is attached at v, either no folding occurs and the intersection is the single vertex v, or some folding occurs and there is a pointed piece (q,r) with $A \cap C = \lambda_A(p_q) = \lambda_C(p_r)$, in which v can appear in the interior or at an endpoint of the shared path. These possibilities are illustrated in part $(2)(m=2)$.

Finally, if $m-1=2$, then there are two faces $A, B \in S_{c-1}$ containing v. As shown in the $m=2$ case in part 1, $A \cap B$ contains an edge. Thus when the face Π_c is attached at v, no folding can occur for otherwise we would get a face which shares edges with more than one other face, in violation of Lemma 3.6(5). Therefore, $m=3$ and $\mathrm{Star}(v)$ must be one of the four possibilities enumerated in $(1)(m=2)$, but with a face attached at v, as illustrated in $(2)(m=3)$. □

Corollary 3.8 (Catalog of Vertex Star Sets for Schützenberger Complexes). *Let w be sparse and let $M = \mathrm{Inv}\langle X \mid w = 1\rangle$. Let (S, O, σ) be the Schützenberger complex of 1 in M. Given some vertex $v \in V(S)$, let m denote the number of faces which contain v. Then $1 \leq m \leq 3$ and one of possibilities listed in part (2) of Theorem 3.7 holds.*

Proof. Let $\{S_i\}_{i=0}^{\infty}$ be an Schützenberger approximation sequence so that $S_0 \subseteq S_1 \subseteq S_2 \subseteq \ldots$ and $S = \bigcup_{i=0}^{\infty} S_i$. Then $m \geq 1$ because there is a face F starting at every vertex v. If $m \geq 4$ then there would have to be an approximation S_I in which $\mathrm{Star}(v)$ contains these four faces, since S is a union of nested spaces. This would violate the Catalog Theorem for Schützenberger Approximations. Thus $1 \leq m \leq 3$, and $\mathrm{Star}(v)$ is described by part (2) of the Catalog Theorem for Schützenberger Approximations, which is the statement of this corollary. □

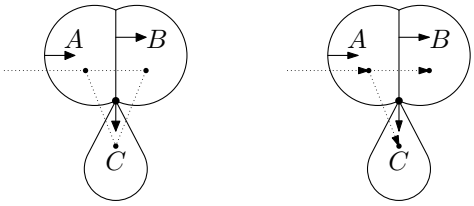


Figure 3.13: Two definitions of dual graph

3.4 The dual graph

The Catalog Theorem enables us to define a dual graph for the Schützenberger complex S. Our definition differs slightly from the usual definition. Usually, one defines the dual graph of a two-dimensional CW-complex as the graph with vertices corresponding to the faces of the complex, and with an edge between any pair of faces which share a vertex. Here, we define a directed edge going to a face from its "predecessor"—that is, the face in the complex which must exist in any approximation containing the destination face. The difference between the two definitions is illustrated in Figure 3.13. The figure on the left illustrates the usual definition of dual graph and the figure on the right illustrates our definition. In the latter there is no edge between B and C because A is the predecessor for both B and C. Since in any Schützenberger approximation sequence, A must have been created before B or C, it makes sense to speak of A has being an "earlier" face of $SC(1)$ than B or C. We will see that the dual graph is a tree, so that when there is a directed path from vertex A to vertex B in the dual graph, there are corresponding faces A and B of $SC(1)$ such that, in any Schützenberger approximation, A must have been created before B.

More formally, we define the dual graph as follows:

Definition 3.9. *Let w be sparse, and let (S, O, σ) be either the Schützenberger complex of 1, or a Schützenberger approximation of length $I \geq 1$. The* **dual graph** *of S is the directed graph $D = \mathcal{D}(S)$ such that*

- $V(D) = F(S)$ *(the faces of* S*), and*
- $E(D)$ *is the set of directed edges, each from a vertex* A *to a vertex* C *(written* (A, C)*), such that* $A \neq C$ *and either*
  - A *and* C *share an edge in* S *and* $\sigma(C) \in A$*, or*
  - $A \cap C$ *is a vertex* v *with* $\sigma(C) = v$*, and for any other face* B *containing* v*,* $\sigma(B) \in A$*.*

We need the lemma below to prove that the dual graph is a tree.

Lemma 3.10. *Let* $w \in (X \cup X^{-1})^*$ *be a sparse word and* $M = \mathrm{Inv}\langle X \mid w = 1\rangle$*. Let* S *be either the Schützenberger complex of* 1 *or a Schützenberger approximation* S_I *of length* I*. Then in the dual graph* $D = \mathcal{D}(S)$*, no two edges have a common destination.*

Proof. If two edges in D do have a common destination, we would have edges (A, C) and (B, C), where A, B, and C are distinct faces of S. Then $\sigma(C) \in A$ and $\sigma(C) \in B$, so the vertex $\sigma(C)$ is contained in all three faces A, B, and C of S. By the Catalog Theorem, each vertex is contained in at most three faces, so A, B, C comprise all the faces of $\mathrm{Star}(\sigma(C))$. Since A and B share $\sigma(C)$, either $\sigma(A) \in B$ and $\sigma(B) \notin A$, or $\sigma(B) \in A$ and $\sigma(A) \notin B$. Assume (without loss of generality) that $\sigma(A) \in B$ and $\sigma(B) \notin A$. Also, $\sigma(A) \neq \sigma(C)$, so A and B share at least two vertices ($\sigma(A)$ and $\sigma(C)$) and hence they must share an edge. Thus C cannot share an edge with either A or B. Since A and C do not share an edge and $\sigma(B) \notin A$, there can be no edge (A, C) in the dual graph, by Definition 3.9. This contradicts our assumption that $(A, C) \in E(D)$. Thus, there cannot be two edges with the same destination. □

Theorem 3.11. *Let* $w \in (X \cup X^{-1})^*$ *be a sparse word and* $M = \mathrm{Inv}\langle X \mid w = 1\rangle$*. Let* S *be the Schützenberger complex of* 1 *or a Schützenberger approximation* S_I *of length*

I. Then the dual graph $D = \mathcal{D}(S)$ is a tree of infinite height in which each vertex has at most $|w| - 1$ children.

Proof. Suppose D is not a tree. Then it must contain an undirected circuit composed of vertices

$$F_0, F_1, F_2, \ldots, F_{n-1}, F_n$$

such that $F_0 = F_n$ and for each i, F_i is a face of S and either (F_i, F_{i+1}) or (F_{i+1}, F_i) is an edge of D. We may assume without loss of generality that this circuit has no repeated vertices, for if some vertex were repeated we could delete the vertices between the repeated ones (and one of the repeated vertices) and obtain a smaller circuit.

Now, if S is the full Schützenberger complex, then since there are finitely many faces F_i, there must be a Schützenberger approximation S_I containing all of the faces F_i. Let $\Pi_1, \ldots, \Pi_I$ be the faces of S_I with Π_i created in step i, and let j_i, $0 \leq i \leq n$ be defined so that $F_i = \Pi_{j_i}$. We therefore have the circuit

$$\Pi_{j_0}, \Pi_{j_1}, \Pi_{j_2}, \ldots, \Pi_{j_{n-1}}, \Pi_{j_n}$$

with $j_0 = j_n$.

Since no vertex is repeated in the circuit each vertex is contained in exactly two edges of the circuit. From this and the fact that no two edges of D can have a common destination (Lemma 3.10), the circuit must consist of edges pointing in a single direction. That is, either $(\Pi_{j_i}, \Pi_{j_{i+1}})$ is an edge of S_I for $0 \leq i \leq n-1$ or $(\Pi_{j_{i+1}}, \Pi_{j_i})$ is an edge of S_I for $0 \leq i \leq n-1$. By Lemma 3.6(3), it follows that either $j_1 < j_2 < \cdots < j_{n-1} < j_n = j_0 < j_1$ or $j_1 > j_2 > \cdots > j_{n-1} > j_n = j_0 > j_1$, either of which implies $j_1 < j_1$, a contradiction. Thus no circuit can exist, and the dual graph must be a tree.

Since each face F has $|w|-1$ vertices other than its start vertex, the vertex F has at most $|w|-1$ children (it could have fewer than $|w|-1$ children if F folded on to its predecessor, so that the faces attached at the shared vertices are children of F's predecessor, not of F). In addition, no face folds completely onto its predecessor, by the Catalog Theorem. Therefore every face has a child, and the height of the tree must be infinite. □

In the following chapters, we use the fact that the dual graph is a tree to provide an efficient solution to the word problem.

Chapter 4

The word problem

4.1 The word problem is solvable

Theorem 4.1. *If $w \in (X \cup X^{-1})^*$ is sparse, then the word problem for $M = \mathrm{Inv}\langle X \mid w = 1\rangle$ is solvable.*

Proof. As noted in Section 2.2.5, it is sufficient to prove that there is an algorithm which takes a word $u \in (X \cup X^{-1})^*$ as input, and outputs whether or not $\overline{u} = 1$. Given a sparse word w, the following procedure is such an algorithm for $M = \mathrm{Inv}\langle X \mid w = 1\rangle$:

Following the iterative construction of Section 2.4.3, we may build an arbitrarily long sequence $S_1, S_2, S_3, ..., S_I$ of w-CW complexes (each with finitely many vertices, edges, and faces). By the results of Section 3.3, this sequence is a sequence of containments, all contained in the infinite Schützenberger complex: $S_1 \subseteq S_2 \subseteq \cdots \subseteq S_I \subseteq S$. In addition, since the dual graph S has infinite height and each vertex of the dual graph has finitely many children, there must be an I such that $d(O, s_{I+1}) > |u|$. Since s_i is always chosen to minimize $d(O, s_i)$, we may compute I by successively building each S_i until there is no vertex v remaining in S_I such that $d(O, v) \leq |u|$ with no face is attached at v. It follows that after I steps, we have a w-CW complex S_I which

contains every vertex, edge, or face of S which contains a vertex within $|u|$ of O. Thus u labels a path from O to O in S if and only if it labels such a path in S_I. Since $\overline{u} = 1$ if and only if u labels a path from O to O, the algorithm outputs $\overline{u} = 1$ if u labels a path from O to O in S_I, and outputs $\overline{u} \neq 1$ otherwise. □

4.2 Face types and the PDA for $SC(1)$

We now describe a method of defining a pushdown automaton which encodes the information of the Schützenberger complex of 1. This gives us a more practical and efficient solution to the word problem than that described in Theorem 4.1. As usual, let $M = \mathrm{Inv}\langle X \mid w = 1\rangle$ with $w = a_1 \dots a_n$ sparse, and let (S, O, σ) be the Schützenberger complex of 1 for M.

The **augmented dual graph** $D' = \mathcal{D}'(S)$ is defined from the dual graph $D = \mathcal{D}(S)$ as follows:

- $V(D') = \{O\} \cup V(D)$
- $E(D') = \{(O, \sigma^{-1}(O))\} \cup E(D)$

(Note that $\sigma^{-1}(O)$ is the first face of the Schützenberger complex, the one attached at the origin O.) The augmented dual graph is simply the dual graph together with an additional vertex equal to the initial vertex O of S and the additional directed edge $(O, \sigma^{-1}(O))$. Therefore, the augmented dual graph is a tree.

The **owner** map $\Omega : V(S) \to V(\mathcal{D}'(S))$ is defined by $\Omega(O) = O$ and for $v \neq O$, $\Omega(v) = F$ where $F \in F(S)$ is the earliest face, in the sense defined by the dual graph $D(S)$, such that $v \in F$.

Let F be a face of $SC(1)$, and let $v_0, \dots, v_{n-1}$ be the vertices of F, so that $(v_0, a_1, v_1), \dots, (v_{n-1}, a_n, v_n)$ are the edges of F (where the subscripts are integers

mod n). Let $B = \Omega(\sigma(F))$ be the face (or O) on which F is attached. Then F cannot fold completely on to B by sparseness, so there are integers (mod n) called b ("back") and f ("forward"), indicating how far back and forward F folds on to B. The edges

$$(v_b, a_{b+1}, v_{b+1}), \ldots, (v_{n-1}, a_n, v_0), (v_0, a_1, v_1), \ldots, (v_{f-1}, v_f, v_f) \in B \cap F,$$

are the **folded portion** of F, and the edges

$$(v_f, a_{f+1}, v_{f+1}), \ldots, (v_{b-1}, a_b, v_b) \in F \backslash B,$$

are the **unfolded portion** of F. When speaking of individual points of these edges, the end points v_b and v_f are considered to be in the folded portion but not in the unfolded portion of F. Also, it follows from the definition of Ω that the vertices $v_{f+1}, \ldots, v_{b-1}$ of the unfolded portion are owned by F, and the vertices $v_b, \ldots, v_f$ of the folded portion are owned by B.

Although b and f are integers mod n, we will usually view them as their integer representatives such that $b \in \{1, \ldots, n\}$ and $f \in \{0, \ldots, n-1\}$. Thus $f < b$. (If F does not fold at all on to B, then $b = n$ and $f = 0$.) In this case, $b - f$ is the number of edges in the unfolded portion of F, and $m = n - (b - f)$ is the number of edges which folded on to B. It follows that m is the length of the segment $(b, w[b, m], 1)$. Since the zones of the segments of a pointed piece cannot intersect if w is sparse, $m < n/2$. In particular, since $n \geq 2$ (words of length 1 are not sparse, by definition), $b - f \geq 2$ so the unfolded portion consists of at least two edges.

Let $e := d(O, v_f) - d(O, v_b)$ and $k := \frac{f+b-e}{2}$. Since there is a path of length m from v_b to v_f (the folded portion of F), we know $|e| \leq m < b - f$ and $f < k < b$. Thus k may be viewed as the index of a point $K = v_k$ in the unfolded portion of F, where, for noninteger k, v_k is defined to be the midpoint of the edge $(v_{\lfloor k \rfloor}, a_{\lceil k \rceil}, v_{\lceil k \rceil})$.

By the Catalog Theorem and the fact that the dual graph is a tree, any path from O to a point on the unfolded portion of F must pass through v_b or v_f. Thus, if $e \geq 0$ then

$$d(O, v_{b-e}) = d(O, v_b) + d(v_b, v_{b-e}) = d(O, v_b) + e = d(O, v_f).$$

Furthermore, the path from v_f to K in the unfolded portion of F, and the path from v_{b-e} to K both have length $\frac{b-f-e}{2}$. Thus, $d(O, K) = d(O, v_f) + \frac{b-f-e}{2}$, and any other point on the unfolded portion of F is closer to O. In a similar manner, we also see that K is the furthest point from O on the unfolded portion of F if $e < 0$. We therefore call K the **geodesic sink** for (the unfolded portion of) F.

From the above, it also follows that k (or e) determines the geodesic directions of the edges on the unfolded portion of F. The edges

$$(v_f, a_{f+1}, v_{f+1}), \ldots, (v_{k_f-1}, a_{k_f}, v_{k_f})$$

and

$$(v_b, a_b^{-1}, v_{b-1}), \ldots, (v_{k_b+1}, a_{k_b+1}^{-1}, v_{k_b})$$

(where $k_f = \lfloor k \rfloor$ and $k_b = \lceil k \rceil$) are all oriented away from the origin. It is nevertheless possible that $k = f + \frac{1}{2}$, in which case $k_f = f$, v_f and v_{f+1} are equidistant from O, and (v_f, a_{f+1}, v_{f+1}) does not point away from the origin (i.e., the first list above is empty). Similarly, it may be the case that $k = b - \frac{1}{2}$ so $k_b = b$ and (v_b, a_b^{-1}, v_{b-1}) does not point away from the origin.

The triple (b, f, k) is called the **face type** of F, denoted $\mathrm{ft}(F)$. In addition, we define $\mathrm{ft}(O) = O$ (O is its own face type), even though O is not a face, so that we may speak of the "face type" of any vertex of the augmented dual graph.

We use face types and the owner map to define an equivalence relation on the

Schützenberger complex S:

Definition 4.2. *Let $\approx$ be the equivalence relation on the Schützenberger complex S generated by the following equivalences:*

- *$F_1 \approx F_2$ for $F_1, F_2 \in F(S)$ with $\mathrm{ft}(F_1) = \mathrm{ft}(F_2)$.*
- *$v_1 \approx v_2$ for $v_1, v_2 \in V(S)$ if $\Omega(v_1) \approx \Omega(v_2)$ and there exists an index $i \in \mathbb{Z}/n\mathbb{Z}$ such that $v_1 = v_{\Omega(v_1)}(i)$ and $v_2 = v_{\Omega(v_2)}(i)$.*
- *$(v_1, x_1, u_1) \approx (v_2, x_2, u_2)$ for $(v_1, x_1, u_1), (v_2, x_2, u_2) \in E(S)$ if $v_1 \approx v_2$, $x_1 = x_2$, and $u_1 \approx u_2$.*

For $z \in V(S) \cup E(S) \cup F(S)$, we denote the equivalence class of z as $[z]_\approx$ or simply $[z]$.

This suggests the following theorem:

Theorem 4.3. *Let $w \in (X \cup X^{-1})^*$ be a sparse word, and let S be the Schützenberger complex of 1 for $M = \mathrm{Inv}\langle X \mid w = 1\rangle$. Let $u_1, u_2 \in V(S)$ such that $u_1 \approx u_2$. Let $A_i = \Omega(u_i)$ and $B_i = \sigma^{-1}(u_i)$ be the owner of, and face attached at, u_i ($i = 1, 2$), respectively. Then $A_1 \approx A_2$ and $B_1 \approx B_2$.*

In addition, there is an edge (u_1, x, v_1) if and only if there is an edge (u_2, x, v_2). If there are such edges, exactly one of the following holds:

(1) $\Omega(u_i) = \Omega(v_i) = A_i$ (for $i = 1, 2$) and $v_1 \approx v_2$,

(2) $\Omega(u_i) \neq \Omega(v_i)$, $(\Omega(u_i), \Omega(v_i)) \in E(\mathcal{D}'(S))$, and $v_1 \approx v_2$, or

(3) $\Omega(u_i) \neq \Omega(v_i)$ and $(\Omega(v_i), \Omega(u_i)) = (\Omega(\sigma(A_i)), A_i) \in E(\mathcal{D}'(S))$.

Proof. If u_1 or u_2 is O, then $u_1 = u_2 = O$ and the theorem follows. If neither is O, then from the definitions of Ω and $\approx$, we have $A_1 \approx A_2$, and u_1 and u_2 are vertices

on the unfolded portions of A_1 and A_2 with the same index. Let $(b, f, k) = \text{ft}(A_i)$. Since the vertex indices, edge labels, and geodesic orientations of the edges in the unfolded portion of A_i are determined by the parameters b, f, and k, there is a bijection between the unfolded portions of A_1 and A_2 which respects vertex indices, edge labels, and geodesic orientations. Furthermore, sparseness guarantees that the only vertices and edges B_i can share with A_i are vertices and edges from the unfolded portion of A_i. It follows that if b' and f' indicate how far back and forward B_1 folds onto A_1, they also indicate how far back and forward B_2 folds onto A_2, and if $e' = d(O, v_{B_1}(f')) - d(O, v_{B_1}(b'))$ then $e' = d(O, v_{B_2}(f')) - d(O, v_{B_2}(b'))$ as well. Thus $\text{ft}(B_1) = \text{ft}(B_2)$.

Now, let (u_1, x, v_1) be any edge of S. If $(u_1, x, v_1) \in A_1$, then it follows from the above that there is an edge (u_2, x, v_2) in A_2 with $v_i = v_{A_i}(j)$ for some index j. It also follows that either $\Omega(u_i) = \Omega(v_i)$ for $i = 1, 2$ or $\Omega(u_i) \neq \Omega(v_i)$ for $i = 1, 2$. If $\Omega(u_i) = \Omega(v_i)$, then both u_i and v_i are on the unfolded portion of A_i, and we have $\Omega(u_i) = \Omega(v_i) = A_i$ and $v_1 \approx v_2$. If $\Omega(u_i) \neq \Omega(v_i)$, then since u_i is on the unfolded portion of A_i (by definition), v_i must be on the folded portion of A_i, so $\Omega(v_i)$ is the predecessor of A_i in the augmented dual graph, $\Omega(\sigma(A_i))$. That is, $(\Omega(v_i), \Omega(u_i)) = (\Omega(\sigma(A_i)), A_i) \in E(\mathcal{D}'(S))$.

If (u_1, x, v_1) is an edge of B_1 but not A_1, then we must have an edge (u_2, x, v_2) in B_2 but not A_2. Since (u_i, x, v_i) is an edge of B_i but not A_i, v_i must be on the unfolded portion of B_i. Thus $\Omega(u_i) \neq \Omega(v_i)$, $(\Omega(u_i), \Omega(v_i)) = (A_i, B_i) \in E(\mathcal{D}'(S))$, and $v_1 \approx v_2$.

Finally, if (u_1, x, v_1) is not an edge of A_1 or B_1, it must be an edge of some third face C_1. Since $\sigma(C_1) \neq u_1$ (because $\sigma(B_1) = u_1$) and C_1 contains the point u_1 from the unfolded portion of A_1, it follows from the Catalog Theorem that A_1 and C_1 share an edge and $\sigma(C_1)$ is on the unfolded portion of A_1. From the first part of this

theorem, it follows that for the corresponding face C_2, with $\sigma(C_2) = v_{A_2}(i_{A_1}(\sigma(C_1)))$, we have $\sigma(C_1) \approx \sigma(C_2)$ and $C_1 \approx C_2$. Moreover, there is an edge (u_2, x, v_2) of C_2 which is not on either A_2 or B_2. It follows that v_i is on the unfolded portion of C_i. Thus $\Omega(u_i) \neq \Omega(v_i)$, $(\Omega(u_i), \Omega(v_i)) = (A_i, C_i) \in E(\mathcal{D}'(S))$, and $v_1 \approx v_2$. □

We now define the pushdown automaton which encodes the information in the Schützenberger complex of 1:

Definition 4.4. *Let (S, O, σ) be the Schützenberger complex of 1 for $M = \mathrm{Inv}\langle X \mid w = 1\rangle$, for a sparse word w. Let $\mathcal{P} = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ be the pushdown automaton defined as follows:*

- *states:* $Q = Q_{\mathcal{P}} = \{[v] \mid v \in V(S)\}$
- *input alphabet:* $\Sigma = X \cup X^{-1}$
- *stack alphabet:* $\Gamma = Q_{\mathcal{P}} = \{[v] \mid v \in V(S)\}$
- *initial state:* $q_0 = [O]$
- *initial stack symbol:* $Z_0 = [O]$
- *final or accept state:* $F = \{[O]\}$
- *transition function: The partial function* $\delta\colon Q \times \Sigma \times \Gamma \to Q \times \Gamma^*$, *is defined so that* $\delta([u], x, t)$ *is*
  - $([v], t)$ *if there is an edge* $(u, x, v) \in E(S)$ *with* $\Omega(u) = \Omega(v)$,
  - $([v], [\sigma(\Omega(v))]t)$ *if there is an edge* $(u, x, v) \in E(S)$ *with* $\Omega(u) \neq \Omega(v)$ *and* $(\Omega(u), \Omega(v)) \in E(\mathcal{D}'(S))$,
  - $([v], \epsilon)$ *if there is an edge* $(u, x, v) \in E(S)$ *with* $\Omega(u) \neq \Omega(v)$, $(\Omega(v), \Omega(u)) \in E(\mathcal{D}'(S))$, *and* $t = [\sigma(\Omega(u))]$, *or*

– *undefined if none of the above cases hold.*

Note that $\delta([u], x, t)$ is undefined if and only if there is no edge $(u, x, v) \in E(S)$ or there is no such edge with $t = [\sigma(\Omega(u))]$. The fact that δ is well-defined follows directly from Theorem 4.3. The usefulness and interpretation of the PDA is provided by the following theorem:

Theorem 4.5. *Let w be sparse, and let S be the Schützenberger complex of 1 for $M = \mathrm{Inv}\langle X \mid w = 1\rangle$. If $y, z \in (X \cup X^{-1})^*$, $v \in V(S)$, and $s \in Q_{\mathcal{P}}^*$, then*

$$([O], yz, [O]) \vdash^* ([v], z, s)$$

if and only if

(1) y labels a path from O to v in S, and

(2) $s = [\sigma(F_m)] \cdots [\sigma(F_2)][\sigma(F_1)][O] = [\sigma(F_m)] \cdots [\sigma(F_2)][O][O]$, where $F_m = \Omega(v)$, $\sigma(F_1) = O$, and $O, F_1, \ldots, F_m$ is the unique directed path from O to F_m in the augmented dual graph. In particular, $s = [O]$ (indicating the empty path staying at O in the augmented dual graph) if and only if $y = \epsilon$ (so $\overline{y} = 1$ and $v = O$).

Proof. We prove this by induction on the length of y. If $|y| = 0$, then the statement holds because $y = \epsilon$ does label a path from O to O in S (with corresponding path $s = [O]$ in the augmented dual graph).

Now, suppose the theorem holds for y with $|y| = k$, $k \geq 0$. We want to show it also holds for a string yx, $x \in X \cup X^{-1}$, of length $k+1$. Let us prove the forward implication first. If $([O], yxz, [O]) \vdash^* ([v], z, s')$, then $([O], yxz, [O]) \vdash^* (q, xz, s) \vdash ([v], z, s')$ for some $q \in Q_{\mathcal{P}}$ and $s \in Q_{\mathcal{P}}^*$. By induction, $q = [u]$, where y labels a path from O to u in S, and $s = [\sigma(F_m)][\sigma(F_{m-1})] \cdots [\sigma(F_1)][O]$ indicates the path $O, F_1, \ldots, F_m$ to $\Omega(u) = F_m$ in the augmented dual graph.

Since $([u], xz, s) \vdash ([v], z, s')$, we must have $\delta([u], x, t) = ([v], t')$, such that (i) there is an edge $(u, x, v) \in E(S)$, (ii) $t = [\sigma(F_m)] = [\sigma(\Omega(u))]$ is the first letter of s, and (iii) t' is a prefix of s'. Thus yx labels a path from O to v in S since y labels a path from O to u and $(u, x, v) \in E(S)$, so (1) holds.

For the inductive step of (2), we have the following cases, from the definition of δ (Definition 4.4):

Case 1: $\Omega(u) = \Omega(v)$: Here, $t' = t$ and so $s' = s$. Since $\Omega(u)$ and $\Omega(v)$ are the same cell, s' must represent the unique directed path to $\Omega(v)$ in the augmented dual graph since s does.

Case 2: $\Omega(u) \neq \Omega(v)$ and $(\Omega(u), \Omega(v)) \in E(\mathcal{D}'(S))$: In this case,
$t' = [\sigma(\Omega(v))][\sigma(\Omega(u))]$, so

$$s' = [\sigma(\Omega(v))][\sigma(\Omega(u))][\sigma(F_{m-1})] \cdots [\sigma(F_1)][O].$$

Since $O, F_1, \ldots, F_{m-1}, \Omega(u)$ is the directed path to $\Omega(u)$ in $\mathcal{D}'(S)$ and $(\Omega(u), \Omega(v)) \in E(\mathcal{D}'(S))$, $O, F_1, \ldots, F_{m-1}, \Omega(u), \Omega(v)$ must be the directed path to $\Omega(v)$. (The path is unique since the augmented dual graph is a tree.)

Case 3: $\Omega(u) \neq \Omega(v)$ and $(\Omega(v), \Omega(u)) \in E(\mathcal{D}'(S))$: First, since $(\Omega(v), \Omega(u)) \in E(\mathcal{D}'(S))$, $\Omega(u) \neq O$, so $m \geq 1$. We have $t' = \epsilon$, so $s' = [\sigma(F_{m-1})] \cdots [\sigma(F_1)][O]$ if $m > 1$ or $s' = [O]$ if $m = 1$. If $m = 1$, then $\Omega(u) = \sigma^{-1}(O)$, so $\Omega(v) = O$ since O is the only vertex before $\sigma^{-1}(O)$ in the augmented dual graph. Thus the empty path O is the directed path from O to $\Omega(v) = O$.

If $m > 1$, then we have $(F_{m-1}, \Omega(u))$ and $(\Omega(v), \Omega(u))$ are both edges in the augmented dual graph. Since the augmented dual graph is a tree, $\Omega(u)$ must have a unique predecessor, so $F_{m-1} = \Omega(v)$. Thus $O, F_1, \ldots, F_{m-1}$ is the directed path from O to $\Omega(v)$ in the augmented dual graph, as required. The completes the proof of (2)

for the forward implication of the inductive step.

For the reverse implication of the inductive step, if yx labels a path from O to some $v \in V(S)$, and $s = [\sigma(F_m)] \cdots [\sigma(F_2)][\sigma(F_1)][O]$ indicates the path in the augmented dual graph from O to $\Omega(v)$, then y must label a path to some $u \in V(S)$ and there must be an edge $(u, x, v) \in E(S)$. We may construct s' in the manner described in the three cases above to obtain the string of stack letters indicating the unique directed path from from O to $\Omega(v)$ in the augmented dual graph. By induction it follows that $([O], yxz, [O]) \vdash^* ([u], xz, s) \vdash ([v], z, s')$, or $([O], yxz, [O]) \vdash^* ([v], z, s')$, as required. □

Corollary 4.6. *Let w be sparse, and let $M = \mathrm{Inv}\langle X \mid w = 1\rangle$. Let $y \in (X \cup X^{-1})^*$. Then*

(1) $\overline{y} = 1$ in M if and only if $([O], y, [O]) \vdash^* ([O], \epsilon, [O])$. Thus, the language of words equal to 1 is context-free.

(2) $\overline{y} \in R_1$ if and only if $([O], y, [O]) \vdash^* (q, \epsilon, s)$ for some $q \in Q$ and $s \in \Gamma^*$. Thus, the language of words equal to an element of R_1 in M is context-free.

Proof. This follows directly from the fact that $\overline{y} = 1$ if and only if y labels a path from 1 to 1 in the Schützenberger graph of 1, and $\overline{y} \in R_1$ if and only if y labels a path from 1 in $S\Gamma(1)$. □

4.3 Iterative construction of the PDA

We now describe an algorithm which constructs the pushdown automaton $\mathcal{P}$ of Definition 4.4. More precisely, we construct an isomorphic PDA $\mathcal{P}'$. This is the algorithm used by the C++ program of the next chapter. We use names in a Courier typeface

(for example, `edge`) to emphasize the connections to the computer program. The C++ source code for these algorithms can be found in Section A.2 of the Appendix.

4.3.1 Data objects

We first review some basic ideas from computer science. In the C++ programming language, as is the case in most high-level programming languages, data is stored in *variables*, and each variable has a *data type* which indicates the data that can be stored in the variable. C++ comes equipped with built-in, or atomic, data types such as integers, characters, and strings of characters. We are also permitted to define composite data types, called *data structures* or *classes*, made up of a number of component variables, each of which may be of any other data type. This permits us to store data in a manner that is natural for the setting in which we are working. In this section, we define the primary data structures, `Vertex` and `Face`, used to store the PDA $\mathcal{P}'$.

In the following, let S be the Schützenberger complex of 1 with origin O. Note that for any vertices $u, v \in V(S)$, $u \approx v$ implies $[\Omega(u)] \approx [\Omega(v)]$, so each equivalence class of vertices is owned by an equivalence class of faces in the same way that each vertex of S is owned by a face. Similarly, $\approx$ respects the indices of vertices within their owners, so an equivalence class of vertices has an index within the equivalence class of faces which owns it. Also, each equivalence class of faces has an unfolded portion and owns the equivalence classes of the vertices on its unfolded portion. Thus, each equivalence class of vertices of S is uniquely identified by the equivalence class of faces which owns it, plus its index. The data structures `Vertex` and `Face`, defined below, store equivalence classes of vertices and faces, respectively.

In the following, we use the term "`Vertex`" when specifically referring to states of the $\mathcal{P}'$ as stored in the computer. Otherwise, we ignore the distinction between the

states of $\mathcal{P}$ and $\mathcal{P}'$ and use the term "state." Consequently, the set of all states or `Vertex`'s is denoted $Q_{\mathcal{P}}$.

We now define the data structures. First, we define a `FaceType` to be an element of $\{O, \texttt{EmptyFaceType}\} \cup \text{ft}(F(S))$; that is, `FaceType` is either O, a triple (b, f, k), or a special indicator `EmptyFaceType` to indicate an unknown face type. In the C++ program, a `FaceType` is literally a triple of integers $(b, f, \texttt{sink2})$, where $\texttt{sink2} = 2k$ so that k may be stored as an integer. The `FaceType`'s O and `EmptyFaceType` are indicated by special values of $b, f, \texttt{sink2}$ which cannot arise as an actual face type.

Definition of `Vertex`

A `Vertex` is a data structure consisting of the following. (The source code defining a `Vertex` is at (A4) in Section A.2.)

- A `Face` named `owner`. This indicates that this `Vertex` is on the unfolded portion of `owner`. The `Face` data structure is formally defined below. Note that `owner` is actually a reference, or pointer, to the `Face` which owns this `Vertex`. This prevents the definitions of `Vertex` and `Face` from being circular definitions.

- An integer `index`. The variable `owner` and `index` together uniquely identify an equivalence class of vertices of S.

- A `facetype` named `attachedfacetype`. This is the face type of the faces of S that are attached at the vertices of S which are in the equivalence class identified by this `Vertex`. (The face type is unique by Theorem 4.3.) If `attachedfacetype` = `EmptyFaceType`, then the face type has not yet been computed.

- A partial map

  $\texttt{edge}\colon (X \cup X^{-1}) \times (Q_{\mathcal{P}} \cup \{0\}) \to Q_{\mathcal{P}} \times (\{\texttt{NONE}, \texttt{POP}\} \cup (\{\texttt{PUSH}\} \times Q_{\mathcal{P}})) \times \{-1, 0, +1\}$

  which is the set of transitions of the PDA $\mathcal{P}'$ out of this `Vertex`. Note that while the PDA $\mathcal{P}$ has a single transition function δ, in $\mathcal{P}'$ each `Vertex` has its own `edge` map defining the transitions *out* of the `Vertex`. The coordinates of the domain of this map are interpreted as follows:

	$X \cup X^{-1}$	label of edge
$\times$	$(Q_{\mathcal{P}} \cup \{0\})$	stack letter on the top of the stack; 0 indicates anything can be on the top of the stack

  The range is interpreted as follows:

	$Q_{\mathcal{P}}$	destination `Vertex` of this transition
$\times$	$\{\texttt{NONE}, \texttt{POP}\} \cup (\{\texttt{PUSH}\} \times Q_{\mathcal{P}})$	stack operation
$\times$	$\{-1, 0, +1\}$	geodesic direction

If the stack operation of an `edge` is a `PUSH`, we write $\texttt{PUSH}\, t$ for the stack operation rather than $(\texttt{PUSH}, t)$. Note that in this case, the state $t \in Q_{\mathcal{P}}$ is being viewed as an element of the stack alphabet.

We now describe the manner in which `edge` maps elements. Let q and q' denote `Vertex`'s, $x \in X \cup X^{-1}$, and let $t \in Q_{\mathcal{P}}$ be a stack letter. Each mapping of elements by the `edge` map for q takes one of the following forms:

- $(x, 0) \mapsto (q', \texttt{NONE}, +1)$: This is the transition of $\mathcal{P}'$ corresponding to the transition $\delta(q, x, t') = (q', t')$ (for all stack letters $t' \in \Gamma$) in $\mathcal{P}$. This means that for each $u, v \in V(S)$ with $u \in q$ and $v \in q'$ (taking the point of view of `Vertex`'s

as equivalence classes of vertices of S), $\Omega(u) = \Omega(v)$ and there is an edge $(u, x, v) \in E(S)$. The geodesic direction $+1$ indicates that $d(O, u) < d(O, v)$. There may similarly be `edge`'s with geodesic directions -1 ($d(O, u) > d(O, v)$) and 0 ($d(O, u) = d(O, v)$).

- $(x, 0) \mapsto (q', \texttt{PUSH}\, t, +1)$, where $t = [\sigma(\Omega(v))]$ for some $v \in V(S)$ with $[v] = q'$ and $\Omega(t) = \Omega(q)$ (as noted above, the owner map Ω can be viewed as a map on equivalence classes of $V(S)$). This corresponds to $\delta(q, x, t') = ([v], [\sigma(\Omega(v))]t')$ in $\mathcal{P}$. This means that if $u, v \in V(S)$ with $u \in q$ and $v \in q'$ then $(\Omega(u), \Omega(v)) \in E(\mathcal{D}'(S))$ and $d(O, u) < d(O, v)$, so the geodesic direction of the `edge` is $+1$.

- $(x, t) \mapsto (q', \texttt{POP}, -1)$: This corresponds to $\delta(q, x, t) = (q', \epsilon)$ in $\mathcal{P}$, where there is an edge $(u, x, v) \in E(S)$ with $(\Omega(v), \Omega(u)) \in E(\mathcal{D}'(S))$ and $t = [\sigma(\Omega(u))]$. We must have $d(O, u) > d(O, v)$, so the geodesic direction is -1.

Definition of `Face`

A `Face` consists of the following. (The source code defining a `Face` is at (A5) in Section A.2.)

- A `FaceType` called `facetype` with `facetype` $\neq$ `EmptyFaceType`. This is the face type of the faces of S which are in the equivalence class represented by this `Face`.

- An array `v` of `Vertex`'s.

The `Vertex`'s `v` of a `Face` with `facetype` $\neq O$ are the `Vertex`'s on the unfolded portion of the `Face`. If `facetype` $= O$, then there is one `Vertex`, $\texttt{v}_{O,0}$, corresponding to the equivalence class $[O]$.

4.3.2 Construction algorithm

We now describe the algorithm which produces the PDA $\mathcal{P}'$, stored in the data structures described in the preceding section.

We start by creating a `Face` with `facetype` $= O$ (the `BASECOMPLEX` face), containing the state $\mathtt{v}_{O,0} = [O]$. Initially, the `edge` map for $\mathtt{v}_{O,0}$ is empty and `attachedfacetype` = `EmptyFaceType`. Also, throughout this procedure and the procedure `attachface`, a list `ftyptbl` of all `Face`'s, indexed by `facetype`, is maintained. The `BASECOMPLEX Face` is initially the only `Face` in `ftyptbl`.

The PDA $\mathcal{P}'$ is constructed by calling `attachface` (see below) at the state $\mathtt{v}_{O,0}$. After `attachface` returns, the entire PDA has been constructed, and control passes to the Conclusion at the end of this section.

In addition to `attachface`, the procedure `buildfacebottom` (which is called by `attachface`) is defined below. The C++ source code for the initial set up described above and the procedures `attachface` and `buildfacebottom` can be found at (A8), (A10), and (A9), respectively, in Section A.2. The `ftyptbl` list is defined at (A6).

Procedure `attachface`

This procedure is always called with respect to (or "at") a specific `Vertex` we denote q. The `attachedfacetype` of q is assumed to be `EmptyFaceType`. The algorithm proceeds as follows:

(1) Let T be the `facetype` of the `owner` of q. If $T = O$, then $q = \mathtt{v}_{O,0}$, and this is the initial instance of `attachface` called when $\mathtt{v}_{O,0}$ has no edges. There must be a face of type $(0, 0, n/2)$ attached at O in S. We set $b = n$, $f = 0$, $k = n/2$, and $q_f = q_b = q$, and jump to step 3.

(2) If $T \neq O$ then there is a vertex $v \in V(S)$ with $q = [v]$. The face $F = \sigma^{-1}(v)$

attached at v has some type (b, f, k), with the folded portion shared with the predecessor face $\Omega(v)$. We compute (b, f, k) as follows, with $e = 0$ initially. (The C++ code for this is at (A11).)

(a) We successively traverse `Vertex`'s and `edge`'s in order to find the integer $f \in \{0, \ldots, n-1\}$ such that $w[0, f]$ can be read in the traversed edges. We denote the final `Vertex` attained by q_f. For each `edge` traversed, we increment e by the geodesic direction η of the `edge`.

(b) Similarly, we find $b \in \{n, \ldots, 1\}$ so that $w[0, b-n]$ can be read in the traversed edges. (Note $b - n \leq 0$ so we are reading backward in w.) We denote the final state attained by q_b. While traversing these edges, we decrement e by the geodesic directions η of the `edge`'s.

(c) We set $k = \frac{f+b-e}{2}$.

Since w is sparse, the `edge`'s traversed in these steps correspond to edges on the unfolded portion of $\Omega(v)$ in S, and there are vertices $v_F(b), v_F(f)$ at the endpoints of the folded portion of F and on the unfolded portion of $\Omega(v)$ such that $q_f = [v_F(f)]$ and $q_b = [v_F(b)]$. Since each edge in S corresponding to an `edge` traversed in (a) and (b) above is oriented away from O if and only if the geodesic direction η of the `edge` is $+1$ and oriented toward O if and only if $\eta = -1$, the net value of the incrementing of e must equal $d(O, v_F(f)) - d(O, v)$ in (a) and $d(O, v) - d(O, v_F(b))$ in (b). Thus the combined effect of (a) and (b) gives $e = d(O, v_f) - d(O, v_b)$, which is the value of e as defined in Section 4.2. Thus $k = \frac{f+b-e}{2}$ is the correct geodesic sink, and (b, f, k) is the correct face type of the face F which is attached at the vertex v of S.

(3) At this step, we have determined that the face type of F is (b, f, k). If there is no `Face` in the `ftyptbl` list of this type, we call `buildfacebottom` (see below)

to create it, and we add the new `Face` to `ftyptbl`. Then we recursively call `attachface` at each of the new `Vertex`'s. Note that at this point we have not yet created the correct `PUSH`/`POP` `edge`'s for the new `Vertex`'s. However, due to the sparseness of the relator, these `edge`'s cannot be traversed in step 2 of the recursive call, so the face type of the `Face` attached at each `Vertex` is computed correctly.

(4) We now know that the `ftyptbl` contains the `Face` $[F]$. The Schützenberger complex S contains edges

$$(v_F(f), w[f,1], v_F(f+1))$$

and

$$(v_F(b), w[b,-1], v_F(b-1)),$$

and these are the only edges from a vertex of $\Omega(v_F(f)) = \Omega(v_F(b)) = \Omega(v)$ to a vertex of $\Omega(v_F(f+1)) = \Omega(v_f(b-1)) = F$. In addition, $(\Omega(v), F)$ is an edge in the augmented dual graph. Therefore, we create the following `edge`'s:

in q_f, the `edge`	$(w[f,1], 0)$	$\mapsto$	$(\mathtt{v}_{(b,f,k),f+1}, \mathtt{PUSH}\, q, +1)$
in $\mathtt{v}_{(b,f,k),f+1}$, the `edge`	$(w[f,1]^{-1}, q)$	$\mapsto$	$(q_f, \mathtt{POP}, -1)$
in q_b, the `edge`	$(w[b,-1], 0)$	$\mapsto$	$(\mathtt{v}_{(b,f,k),b-1}, \mathtt{PUSH}\, q, +1)$
in $\mathtt{v}_{(b,f,k),b-1}$, the `edge`	$(w[b,-1]^{-1}, q)$	$\mapsto$	$(q_b, \mathtt{POP}, -1)$

The above `edge`'s correspond to the following transitions in $\mathcal{P}$, respectively (for

all stack letters $t \in \Gamma = Q_{\mathcal{P}}$):

$$\begin{aligned}
\delta([v_F(f)], w[f,1], t) &= ([v_F(f+1)], [v]t) \\
\delta([v_F(f+1)], w[f,1]^{-1}, [v]) &= ([v_F(f)], \epsilon) \\
\delta([v_F(b)], w[b,-1], t) &= ([v_F(b-1)], [v]t) \\
\delta([v_F(b-1)], w[b,-1]^{-1}, [v]) &= ([v_F(b)], \epsilon)
\end{aligned}$$

Since $(v_F(f), w[f,1], v_F(f+1))$ and $(v_F(b), w[b,-1], v_F(b-1))$ are the only edges from a vertex of $\Omega(v)$ to a vertex of F, the `edge`'s created above must be the only `edge`'s between vertices of the `Face`'s $[\Omega(v)]$ and $[F]$.

(5) The `attachedfacetype` of q is set to (b, f, k), and `attachface` returns to the place from which it was called. If `attachface` was called recursively, it returns to Step 3 of the prior invocation.

After all recursion has completed in Step 3, control passes back to the initial procedure, from which it proceeds to the Conclusion below.

Procedure `buildfacebottom`

This procedure creates a new `Face` and builds the "bottom," or unfolded, portion of the `Face`. That is, it creates the `Vertex`'s in the `v` array of the `Face`, and creates the `edge`'s between these `Vertex`'s. Note that these `edge`'s have stack operations of type `NONE` since they are are edges between `Vertex`'s belonging to the same `Face`.

Procedure `buildfacebottom` is called with the `facetype` (b, f, k) of the `Face` to be constructed. The `Vertex`'s

$$\mathtt{v}_{(b,f,k),f+1}, \ldots, \mathtt{v}_{(b,f,k),b-1},$$

are created. These `Vertex`'s correspond to the equivalence classes of the vertices on

the unfolded portion of faces in the equivalence class represented by the `Face`. The `attachedfacetype` for each of these `Vertex`'s is set to `EmptyFaceType`, and the `edge` maps are initially defined as follows:

(1) For $f+1 \leq i \leq k-1$,

- the `edge` map for $\mathtt{v}_{(b,f,k),i}$ maps $(w[i,1],0) \mapsto (\mathtt{v}_{(b,f,k),i+1}, \mathtt{NONE}, +1)$
- the `edge` map for $\mathtt{v}_{(b,f,k),i+1}$ maps $(w[i,1]^{-1},0) \mapsto (\mathtt{v}_{(b,f,k),i}, \mathtt{NONE}, -1)$.

(Note that if $w = a_1 \ldots a_n$ then $w[i,1] = a_{i+1}$.) These are the edges from vertex $f+1$ of the face to the geodesic sink of the face.

(2) For $b-1 \geq j \geq k+1$,

- the `edge` map for $\mathtt{v}_{(b,f,k),j}$ maps $(w[j,-1],0) \mapsto (\mathtt{v}_{(b,f,k),j-1}, \mathtt{NONE}, +1)$
- the `edge` map for $\mathtt{v}_{(b,f,k),j-1}$ maps $(w[j,-1]^{-1},0) \mapsto (\mathtt{v}_{(b,f,k),j}, \mathtt{NONE}, -1)$.

(Here, $w[j,-1] = a_j^{-1}$.) These are the edges from vertex $b-1$ to the geodesic sink.

(3) If k is not an integer, then for $i = \lfloor k \rfloor$ (the greatest integer less than k),

- the `edge` map for $\mathtt{v}_{(b,f,k),i}$ maps $(w[i,1],0) \mapsto (\mathtt{v}_{(b,f,k),i+1}, \mathtt{NONE}, 0)$
- the `edge` map for $\mathtt{v}_{(b,f,k),i+1}$ maps $(w[i,1]^{-1},0) \mapsto (\mathtt{v}_{(b,f,k),i}, \mathtt{NONE}, 0)$.

This defines the two edges between the vertices on either side of the geodesic sink of the face, if there are two vertices in the face which are equidistant from the origin.

The above defines all `edge`'s between `Vertex`'s of this `Face`. After these `edge`'s have been created, control is returned to the calling procedure (`attachface`), which is responsible for creating the `PUSH` and `POP` `edge`'s between this `Face` and its predecessor.

Conclusion

Since every time a new `Face` was created, `attachface` was called at each of the new `Vertex`'s, there can be no `Vertex`'s remaining with `attachedfacetype` = `EmptyFaceType`, and the set of all `Vertex`'s must contain all states of $\mathcal{P}$. Also, since `buildfacebottom` created all `edge`'s of stack operation type `NONE` within each `Face`, and `attachface` created all `PUSH`/`POP` `edge`'s between `Vertex`'s belonging to different `facetype`'s, every `edge` between any two given `Vertex`'s was created. Therefore, the PDA defined by the `Vertex`'s and `edge`'s is $\mathcal{P}$.

4.4 Geodesics in $S\Gamma(1)$

We now define a finite state automaton, and prove that it recognizes the language of words labeling geodesic paths from O in the Schützenberger complex of 1:

Definition 4.7. *Let w be sparse and let (S, O, σ) be the Schützenberger complex of* 1 *for $M = \mathrm{Inv}\langle X \mid w = 1\rangle$. Let $(Q, \Sigma, \delta, q_0, F)$ be the finite state automaton defined as follows:*

- *states:* $Q = Q_{\mathcal{P}} = \{[v] \mid v \in V(S)\}$
- *input alphabet:* $\Sigma = X \cup X^{-1}$
- *initial state:* $q_0 = [O]$
- *final (accept) states:* $F = Q$ *(all states accept)*
- *transition function: the partial function* $\delta \colon Q \times \Sigma \to Q$*, is defined by* $\delta([u], x) = [v]$ *if there is an edge* $(u, x, v) \in E(S)$ *and one of the following hold:*
  - $\Omega(u) = \Omega(v)$*, and*

* $i_{\Omega(u)}(u) < i_{\Omega(u)}(v) \leq k$ *or*
* $i_{\Omega(u)}(u) > i_{\Omega(u)}(v) \geq k$,

where $(b, f, k) = \mathrm{ft}(\Omega(u))$, *or*

– $(\Omega(u), \Omega(v)) \in E(\mathcal{D}'(S))$.

This transition function is well-defined because it is a restriction of the transition function for the PDA in Definition 4.4, which is well-defined.

Theorem 4.8. *The language of the finite state automaton in Definition 4.7 is the language of geodesics for* $SC(1)$.

Proof. Let $(u, x, v) \in E(S)$. Consider the transition function of $\mathcal{P}$. If $(\Omega(v), \Omega(u)) \in E(\mathcal{D}'(S))$, then v must be on the folded portion of $\Omega(u)$ and u must be on the unfolded portion of $\Omega(u)$. Thus $d(O, v) < d(O, u)$. On the other hand, if $(\Omega(u), \Omega(v)) \in E(\mathcal{D}'(S))$ then $d(O, v) > d(O, u)$. Finally, if $\Omega(u) = \Omega(v)$, then u and v are both vertices on the unfolded portion of $\Omega(u)$. Since the point at index k is the geodesic sink, it follows that $d(O, v) > d(O, u)$ if and only if $i_{\Omega(u)}(u) < i_{\Omega(u)}(v) \leq k$ or $i_{\Omega(u)}(u) > i_{\Omega(u)}(v) \geq k$.

By Theorem 4.5, if all states of the PDA $\mathcal{P}$ in Definition 4.4 are allowed to accept, $\mathcal{P}$ accepts all words that label paths from O in $SC(1)$. Such a path will be geodesic if and only if $d(O, v) > d(O, u)$ for each edge (u, x, v) traversed. The finite state automaton defined in Definition 4.7 is precisely the underlying finite statement automaton of $\mathcal{P}$ consisting only of transitions associated with edges (u, x, v) such that $d(O, v) > d(O, u)$. Thus this finite state automaton excepts precisely the words which label geodesic paths in S. □

Note that the finite state automaton constructed in Definition 4.7 may not be the minimal FSA, but can be minimized. The corresponding minimized automaton is the cone type automaton defined in Section 2.2.2.

In the C++ program, the cone type automaton is created by taking the `edge`'s of the PDA $\mathcal{P}'$ which have a direction of $+1$ (ignoring the stack information), and applying the minimization algorithm of [HU79].

Chapter 5

Implementation and sample runs

In this section, we describe `spar` (short for "sparse"), an implementation, in the C++ programming language [Str00], of the algorithms of the previous chapter. The program does the following, on input of a word $w \in (X \cup X^{-1})^*$:

- Checks whether w is sparse, and if not, asks the user to enter a sparse relator.
- On input $u \in (X \cup X^{-1})^*$, outputs whether or not $\overline{u} = 1$ in $M = \mathrm{Inv}\langle X \mid w = 1\rangle$ (i.e., whether u labels a path from O to O in the Schützenberger complex of 1) and whether or not $\overline{u} \in R_1$ (whether u labels any path in $SC(1)$);
- On input $u, v \in (X \cup X^{-1})^*$, outputs whether or not $\overline{u}, \overline{v} \in R_1$ and if so, outputs whether or not $\overline{u} = \overline{v}$.
- Outputs the cone type automaton of the Schützenberger complex of 1, the finite state automaton which recognizes the language of words which label geodesic paths in the Schützenberger complex of 1.

5.1 Sample run

We begin with a sample run of `spar` using a number of different relators.

5.1.1 Entering the relator

When you first start the program, it asks you for the relator w you wish to use. The program checks to verify that your relator is sparse. If not, it asks you to enter another relator.

```
$ spar

Welcome to 'spar'.  This program computes the PDA associated with the
Schutzenberger complex of 1 for inverse monoids of the form
M=Inv<X|w=1>, where w is a sparse word, and allows you to test whether
words are accepted by this PDA.

(Note: When this program asks for input, you may respond with ? or
'help' to get more information on what kind of response is required.
Also, you may run this program by giving operands and commands on
the command line and in files; type 'spar help' at the command line
for more information.)

Enter your relator w: bbaxabbbayab
Your relator, bbaxabbbayab, is not sparse because the pointed piece
((0,bb,1),(11,bb,1)) overlaps itself.
Please enter a sparse relator.

Enter your relator w: abAB
Your relator, abAB, is not sparse because the pointed pieces
((2,A,1),(1,A,-1)) and ((1,b,1),(0,b,-1)) overlap.
Please enter a sparse relator.
```

5.1.2 Sample output for $w = abABcdCD$

Once you have entered a sparse relator, `spar` computes the PDA for the Schützenberger complex of 1. It will use this internally to help it perform calculations for M. It then presents you with a number of choices for further calculations.

```
Enter your relator w: abABcdCD

I have computed the Schutzenberger complex of 1 for
M=Inv<X|w=1>, w=abABcdCD.

Choose from among the following options:
  1) List the PDA (face types and transitions) for your inverse monoid
  2) Interactively test whether or not u=1; i.e., whether SC(1) accepts u
  3) Test whether u,v are R-related to 1, and if so, whether u=v
  4) Compute the cone type automaton which recognizes geodesics in SC(1)
  8) Enter a new relator w (your current monoid is discarded)
  9) Leave this program (quit and Ctrl-D also work)
What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)?
```

Displaying the PDA for the inverse monoid

The output of option 1 (list the PDA for your inverse monoid) is shown below. This output gives the face types, vertices (states), and transitions which comprise the PDA for the Schützenberger complex of 1. The `BASECOMPLEX` is the single vertex, corresponding to monoid element 1, comprising the first vertex of the augmented dual graph. The other faces are the various face types comprising the augmented dual graph. For example, face type `(0,0,4)` corresponds to faces (1), (4), and (7) in Figure 3.1, face type `(0,1,5)` corresponds to faces (2) and (5), and face type `(7,0,3)` corresponds to faces (3) and (6). Note that what `spar` calls a `Face` contains the vertices comprising the unfolded portion of a face.

The output below also illustrates some notational conventions of `spar`. Vertices (states of the PDA) have names of the form `(`*facetype*`)`*vertexnumber*. Within each face, the vertex number corresponds to the index within the word w (as in the definition of the v_P map). The base complex has a single vertex named `(epsilon)0`.

Each vertex has a number of transitions defined. For example, the transition

```
( a ,          t )  ->  (     (0,0,4)1 ,  (epsilon)0 t )  +1
```

for vertex `(epsilon)0`, means that if a letter a is read in the input with the stack

letter t (i.e., anything) on top of the stack, the PDA pushes `(epsilon)0` on top of the stack and moves to vertex `(0,0,4)1`. The symbol `t` here simply indicates that it doesn't matter what is on the top of the stack.

The inverse transition for this is the transition

```
( A ,   (epsilon)0 )  ->  (   (epsilon)0 ,          eps )  -1
```

for vertex `(0,0,4)1`. This indicates that if an A is read with `(epsilon)0` on top of the stack, the PDA pops the stack (leaving the empty word ϵ or "`eps`" in place of the top stack letter `(epsilon)0`) and moves to vertex `(epsilon)0`.

A transition such as

```
( b ,           t )  ->  (     (0,0,4)2 ,            t )  +1
```

moves to another vertex within the current face, without changing the stack.

The "`+1`", "`-1`", or "`0`" notations at the end of each transition indicate whether the corresponding edge in the Schützenberger complex is oriented away from the origin (+1), toward the origin (−1), or is an edge between vertices an equal distance from the origin (0). This information is used by `spar` to produce the cone type automaton for reading the language of geodesic words in the Schützenberger complex of 1.

The following is the output from Option 1:

```
What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)? 1

Your monoid is M=InvMonoid<X|w=abABcdCD=1>  4 face types:
FaceType(epsilon): BASECOMPLEX initial=(epsilon)0, terminal=(epsilon)0;
1 vertex
  Vertex (epsilon)0: attachedfacetype=(0,0,4), 2 edges:
    ( a ,           t )  ->  (     (0,0,4)1 ,  (epsilon)0 t )  +1
    ( d ,           t )  ->  (     (0,0,4)7 ,  (epsilon)0 t )  +1

FaceType(0,0,4): FACE  7 vertices
  Vertex (0,0,4)1: attachedfacetype=(0,0,4), 17 edges:
    ( a ,           t )  ->  (     (0,0,4)1 ,    (0,0,4)1 t )  +1
```

```
  ( b ,           t )  ->  (     (0,0,4)2 ,            t )  +1
  ( d ,           t )  ->  (     (0,0,4)7 ,    (0,0,4)1 t )  +1
  ( A ,   (epsilon)0 )  ->  (   (epsilon)0 ,          eps )  -1
  ( A ,     (0,0,4)1 )  ->  (     (0,0,4)1 ,          eps )  -1
  ( A ,     (0,0,4)2 )  ->  (     (0,0,4)2 ,          eps )  -1
  ( A ,     (0,0,4)4 )  ->  (     (0,0,4)4 ,          eps )  -1
  ( A ,     (0,0,4)6 )  ->  (     (0,0,4)6 ,          eps )  -1
  ( A ,     (0,0,4)7 )  ->  (     (0,0,4)7 ,          eps )  -1
  ( A ,     (0,1,5)2 )  ->  (     (0,1,5)2 ,          eps )  -1
  ( A ,     (0,1,5)4 )  ->  (     (0,1,5)4 ,          eps )  -1
  ( A ,     (0,1,5)6 )  ->  (     (0,1,5)6 ,          eps )  -1
  ( A ,     (0,1,5)7 )  ->  (     (0,1,5)7 ,          eps )  -1
  ( A ,     (7,0,3)1 )  ->  (     (7,0,3)1 ,          eps )  -1
  ( A ,     (7,0,3)2 )  ->  (     (7,0,3)2 ,          eps )  -1
  ( A ,     (7,0,3)4 )  ->  (     (7,0,3)4 ,          eps )  -1
  ( A ,     (7,0,3)6 )  ->  (     (7,0,3)6 ,          eps )  -1
Vertex (0,0,4)2: attachedfacetype=(0,0,4), 5 edges:
  ( a ,           t )  ->  (     (0,0,4)1 ,    (0,0,4)2 t )  +1
  ( A ,           t )  ->  (     (0,0,4)3 ,            t )  +1
  ( b ,           t )  ->  (     (0,1,5)2 ,    (0,0,4)3 t )  +1
  ( B ,           t )  ->  (     (0,0,4)1 ,            t )  -1
  ( d ,           t )  ->  (     (0,0,4)7 ,    (0,0,4)2 t )  +1
Vertex (0,0,4)3: attachedfacetype=(0,1,5), 3 edges:
  ( a ,           t )  ->  (     (0,0,4)2 ,            t )  -1
  ( B ,           t )  ->  (     (0,0,4)4 ,            t )  +1
  ( d ,           t )  ->  (     (0,1,5)7 ,    (0,0,4)3 t )  +1
Vertex (0,0,4)4: attachedfacetype=(0,0,4), 4 edges:
  ( a ,           t )  ->  (     (0,0,4)1 ,    (0,0,4)4 t )  +1
  ( b ,           t )  ->  (     (0,0,4)3 ,            t )  -1
  ( c ,           t )  ->  (     (0,0,4)5 ,            t )  -1
  ( d ,           t )  ->  (     (0,0,4)7 ,    (0,0,4)4 t )  +1
Vertex (0,0,4)5: attachedfacetype=(7,0,3), 3 edges:
  ( a ,           t )  ->  (     (7,0,3)1 ,    (0,0,4)5 t )  +1
  ( C ,           t )  ->  (     (0,0,4)4 ,            t )  +1
  ( d ,           t )  ->  (     (0,0,4)6 ,            t )  -1
Vertex (0,0,4)6: attachedfacetype=(0,0,4), 5 edges:
  ( a ,           t )  ->  (     (0,0,4)1 ,    (0,0,4)6 t )  +1
  ( c ,           t )  ->  (     (7,0,3)6 ,    (0,0,4)5 t )  +1
  ( C ,           t )  ->  (     (0,0,4)7 ,            t )  -1
  ( d ,           t )  ->  (     (0,0,4)7 ,    (0,0,4)6 t )  +1
  ( D ,           t )  ->  (     (0,0,4)5 ,            t )  +1
Vertex (0,0,4)7: attachedfacetype=(0,0,4), 17 edges:
  ( a ,           t )  ->  (     (0,0,4)1 ,    (0,0,4)7 t )  +1
  ( c ,           t )  ->  (     (0,0,4)6 ,            t )  +1
  ( d ,           t )  ->  (     (0,0,4)7 ,    (0,0,4)7 t )  +1
```

```
    ( D ,   (epsilon)0 )  ->  (   (epsilon)0 ,          eps )  -1
    ( D ,     (0,0,4)1 )  ->  (     (0,0,4)1 ,          eps )  -1
    ( D ,     (0,0,4)2 )  ->  (     (0,0,4)2 ,          eps )  -1
    ( D ,     (0,0,4)4 )  ->  (     (0,0,4)4 ,          eps )  -1
    ( D ,     (0,0,4)6 )  ->  (     (0,0,4)6 ,          eps )  -1
    ( D ,     (0,0,4)7 )  ->  (     (0,0,4)7 ,          eps )  -1
    ( D ,     (0,1,5)2 )  ->  (     (0,1,5)2 ,          eps )  -1
    ( D ,     (0,1,5)4 )  ->  (     (0,1,5)4 ,          eps )  -1
    ( D ,     (0,1,5)6 )  ->  (     (0,1,5)6 ,          eps )  -1
    ( D ,     (0,1,5)7 )  ->  (     (0,1,5)7 ,          eps )  -1
    ( D ,     (7,0,3)1 )  ->  (     (7,0,3)1 ,          eps )  -1
    ( D ,     (7,0,3)2 )  ->  (     (7,0,3)2 ,          eps )  -1
    ( D ,     (7,0,3)4 )  ->  (     (7,0,3)4 ,          eps )  -1
    ( D ,     (7,0,3)6 )  ->  (     (7,0,3)6 ,          eps )  -1

FaceType(7,0,3): FACE  6 vertices
  Vertex (7,0,3)1: attachedfacetype=(0,0,4), 6 edges:
    ( a ,           t )  ->  (     (0,0,4)1 ,    (7,0,3)1 t )  +1
    ( b ,           t )  ->  (     (7,0,3)2 ,           t )  +1
    ( d ,           t )  ->  (     (0,0,4)7 ,    (7,0,3)1 t )  +1
    ( A ,     (0,0,4)5 )  ->  (     (0,0,4)5 ,         eps )  -1
    ( A ,     (0,1,5)5 )  ->  (     (0,1,5)5 ,         eps )  -1
    ( A ,     (7,0,3)5 )  ->  (     (7,0,3)5 ,         eps )  -1
  Vertex (7,0,3)2: attachedfacetype=(0,0,4), 5 edges:
    ( a ,           t )  ->  (     (0,0,4)1 ,    (7,0,3)2 t )  +1
    ( A ,           t )  ->  (     (7,0,3)3 ,           t )  +1
    ( b ,           t )  ->  (     (0,1,5)2 ,    (7,0,3)3 t )  +1
    ( B ,           t )  ->  (     (7,0,3)1 ,           t )  -1
    ( d ,           t )  ->  (     (0,0,4)7 ,    (7,0,3)2 t )  +1
  Vertex (7,0,3)3: attachedfacetype=(0,1,5), 3 edges:
    ( a ,           t )  ->  (     (7,0,3)2 ,           t )  -1
    ( B ,           t )  ->  (     (7,0,3)4 ,           t )  -1
    ( d ,           t )  ->  (     (0,1,5)7 ,    (7,0,3)3 t )  +1
  Vertex (7,0,3)4: attachedfacetype=(0,0,4), 4 edges:
    ( a ,           t )  ->  (     (0,0,4)1 ,    (7,0,3)4 t )  +1
    ( b ,           t )  ->  (     (7,0,3)3 ,           t )  +1
    ( c ,           t )  ->  (     (7,0,3)5 ,           t )  -1
    ( d ,           t )  ->  (     (0,0,4)7 ,    (7,0,3)4 t )  +1
  Vertex (7,0,3)5: attachedfacetype=(7,0,3), 3 edges:
    ( a ,           t )  ->  (     (7,0,3)1 ,    (7,0,3)5 t )  +1
    ( C ,           t )  ->  (     (7,0,3)4 ,           t )  +1
    ( d ,           t )  ->  (     (7,0,3)6 ,           t )  -1
  Vertex (7,0,3)6: attachedfacetype=(0,0,4), 7 edges:
    ( a ,           t )  ->  (     (0,0,4)1 ,    (7,0,3)6 t )  +1
    ( c ,           t )  ->  (     (7,0,3)6 ,    (7,0,3)5 t )  +1
```

```
    ( d ,           t )  ->  (     (0,0,4)7 ,    (7,0,3)6 t )  +1
    ( D ,           t )  ->  (     (7,0,3)5 ,           t )  +1
    ( C ,     (0,0,4)5 )  ->  (     (0,0,4)6 ,         eps )  -1
    ( C ,     (0,1,5)5 )  ->  (     (0,1,5)6 ,         eps )  -1
    ( C ,     (7,0,3)5 )  ->  (     (7,0,3)6 ,         eps )  -1

FaceType(0,1,5): FACE  6 vertices
  Vertex (0,1,5)2: attachedfacetype=(0,0,4), 7 edges:
    ( a ,           t )  ->  (     (0,0,4)1 ,    (0,1,5)2 t )  +1
    ( A ,           t )  ->  (     (0,1,5)3 ,           t )  +1
    ( b ,           t )  ->  (     (0,1,5)2 ,    (0,1,5)3 t )  +1
    ( d ,           t )  ->  (     (0,0,4)7 ,    (0,1,5)2 t )  +1
    ( B ,     (0,0,4)3 )  ->  (     (0,0,4)2 ,         eps )  -1
    ( B ,     (0,1,5)3 )  ->  (     (0,1,5)2 ,         eps )  -1
    ( B ,     (7,0,3)3 )  ->  (     (7,0,3)2 ,         eps )  -1
  Vertex (0,1,5)3: attachedfacetype=(0,1,5), 3 edges:
    ( a ,           t )  ->  (     (0,1,5)2 ,           t )  -1
    ( B ,           t )  ->  (     (0,1,5)4 ,           t )  +1
    ( d ,           t )  ->  (     (0,1,5)7 ,    (0,1,5)3 t )  +1
  Vertex (0,1,5)4: attachedfacetype=(0,0,4), 4 edges:
    ( a ,           t )  ->  (     (0,0,4)1 ,    (0,1,5)4 t )  +1
    ( b ,           t )  ->  (     (0,1,5)3 ,           t )  -1
    ( c ,           t )  ->  (     (0,1,5)5 ,           t )  +1
    ( d ,           t )  ->  (     (0,0,4)7 ,    (0,1,5)4 t )  +1
  Vertex (0,1,5)5: attachedfacetype=(7,0,3), 3 edges:
    ( a ,           t )  ->  (     (7,0,3)1 ,    (0,1,5)5 t )  +1
    ( C ,           t )  ->  (     (0,1,5)4 ,           t )  -1
    ( d ,           t )  ->  (     (0,1,5)6 ,           t )  -1
  Vertex (0,1,5)6: attachedfacetype=(0,0,4), 5 edges:
    ( a ,           t )  ->  (     (0,0,4)1 ,    (0,1,5)6 t )  +1
    ( c ,           t )  ->  (     (7,0,3)6 ,    (0,1,5)5 t )  +1
    ( C ,           t )  ->  (     (0,1,5)7 ,           t )  -1
    ( d ,           t )  ->  (     (0,0,4)7 ,    (0,1,5)6 t )  +1
    ( D ,           t )  ->  (     (0,1,5)5 ,           t )  +1
  Vertex (0,1,5)7: attachedfacetype=(0,0,4), 6 edges:
    ( a ,           t )  ->  (     (0,0,4)1 ,    (0,1,5)7 t )  +1
    ( c ,           t )  ->  (     (0,1,5)6 ,           t )  +1
    ( d ,           t )  ->  (     (0,0,4)7 ,    (0,1,5)7 t )  +1
    ( D ,     (0,0,4)3 )  ->  (     (0,0,4)3 ,         eps )  -1
    ( D ,     (0,1,5)3 )  ->  (     (0,1,5)3 ,         eps )  -1
    ( D ,     (7,0,3)3 )  ->  (     (7,0,3)3 ,         eps )  -1

What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)?
```

Interactively testing whether the PDA accepts a word

Option 2 illustrated below allows the user to see how an input word progresses through the PDA for the Schützenberger complex of 1. After each step, the current vertex (state), remaining input to be read, and stack is displayed. The program pauses, and the user may press ENTER to read one more letter of the input, or may enter the number of letters to read before the program pauses again. This process continues until the entire word has been read or until when the next letter cannot be read in the PDA.

Consistent with the conventions in [HU79], the top of the stack is the left end of the string of stack letters. The initial stack letter, denoted here by `(stackstart)` appears at the end (bottom) of the stack. The PDA never pops this letter off the stack.

```
What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)? 2

Enter the word u which you wish to try to read in SC(1).
You will see the attempt to read u in SC(1) step-by-step.
u=abbbbABcdCDBcdCDaBABcdCD

Step 0: State=(epsilon)0  Remaining=abbbbABcdCDBcdCDaBABcdCD  Stack:
(stackstart)
Next (or num of steps):
Step 1: State=(0,0,4)1  Remaining=bbbbABcdCDBcdCDaBABcdCD  Stack:
(epsilon)0 (stackstart)
Next (or num of steps):
Step 2: State=(0,0,4)2  Remaining=bbbABcdCDBcdCDaBABcdCD  Stack:
(epsilon)0 (stackstart)
Next (or num of steps): 99
Step 3: State=(0,1,5)2  Remaining=bbABcdCDBcdCDaBABcdCD  Stack: (0,0,4)3
(epsilon)0 (stackstart)
Step 4: State=(0,1,5)2  Remaining=bABcdCDBcdCDaBABcdCD  Stack: (0,1,5)3
(0,0,4)3 (epsilon)0 (stackstart)
Step 5: State=(0,1,5)2  Remaining=ABcdCDBcdCDaBABcdCD  Stack: (0,1,5)3
(0,1,5)3 (0,0,4)3 (epsilon)0 (stackstart)
Step 6: State=(0,1,5)3  Remaining=BcdCDBcdCDaBABcdCD  Stack: (0,1,5)3
(0,1,5)3 (0,0,4)3 (epsilon)0 (stackstart)
```

```
Step 7: State=(0,1,5)4  Remaining=cdCDBcdCDaBABcdCD  Stack: (0,1,5)3
(0,1,5)3 (0,0,4)3 (epsilon)0 (stackstart)
Step 8: State=(0,1,5)5  Remaining=dCDBcdCDaBABcdCD  Stack: (0,1,5)3
(0,1,5)3 (0,0,4)3 (epsilon)0 (stackstart)
Step 9: State=(0,1,5)6  Remaining=CDBcdCDaBABcdCD  Stack: (0,1,5)3
(0,1,5)3 (0,0,4)3 (epsilon)0 (stackstart)
Step 10: State=(0,1,5)7  Remaining=DBcdCDaBABcdCD  Stack: (0,1,5)3
(0,1,5)3 (0,0,4)3 (epsilon)0 (stackstart)
Step 11: State=(0,1,5)3  Remaining=BcdCDaBABcdCD  Stack: (0,1,5)3 (0,0,4)3
(epsilon)0 (stackstart)
Step 12: State=(0,1,5)4  Remaining=cdCDaBABcdCD  Stack: (0,1,5)3 (0,0,4)3
(epsilon)0 (stackstart)
Step 13: State=(0,1,5)5  Remaining=dCDaBABcdCD  Stack: (0,1,5)3 (0,0,4)3
(epsilon)0 (stackstart)
Step 14: State=(0,1,5)6  Remaining=CDaBABcdCD  Stack: (0,1,5)3 (0,0,4)3
(epsilon)0 (stackstart)
Step 15: State=(0,1,5)7  Remaining=DaBABcdCD  Stack: (0,1,5)3 (0,0,4)3
(epsilon)0 (stackstart)
Step 16: State=(0,1,5)3  Remaining=aBABcdCD  Stack: (0,0,4)3 (epsilon)0
(stackstart)
Step 17: State=(0,1,5)2  Remaining=BABcdCD  Stack: (0,0,4)3 (epsilon)0
(stackstart)
Step 18: State=(0,0,4)2  Remaining=ABcdCD  Stack: (epsilon)0 (stackstart)
Step 19: State=(0,0,4)3  Remaining=BcdCD  Stack: (epsilon)0 (stackstart)
Step 20: State=(0,0,4)4  Remaining=cdCD  Stack: (epsilon)0 (stackstart)
Step 21: State=(0,0,4)5  Remaining=dCD  Stack: (epsilon)0 (stackstart)
Step 22: State=(0,0,4)6  Remaining=CD  Stack: (epsilon)0 (stackstart)
Step 23: State=(0,0,4)7  Remaining=D  Stack: (epsilon)0 (stackstart)
Step 24: State=(epsilon)0  Remaining=epsilon  Stack: (stackstart)

SC(1) accepts u. Therefore, u = 1.

What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)?
```

Testing two words for equality in $SC(1)$

Option 3 allows you two enter two test words. It checks to see whether or not they are $\mathcal{R}$-related to 1 (label paths in $SC(1)$), and if so, it reports whether or not they are equal in M.

```
What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)? 3
```

```
Enter words u and v.  This will determine whether or not
u and v are R-related to 1, and if so, whether or not u=v:
u=abAB
v=dcDC
Your word u IS R-related to 1.
Your word v IS R-related to 1.
The words u and v are EQUAL in M.

What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)?
```

The cone type automaton

Option 4 produces the finite state automaton which recognizes words which label geodesic paths in $SC(1)$. Each state (vertex) is listed, followed by the transitions out of that state. The same naming scheme is used for the states (vertices) of the FSA as is used for the PDA, to help indicate the connection between the PDA and the FSA. Note, however, that the notion of "face" does not make sense here, since a vertex from the PDA may get identified with a vertex from a different face when the FSA minimization is performed. When this is done, one of the names is retained and the other is discarded. For example, vertices `(epsilon)0` and `(0,0,4)4` are equivalent in the FSA; `(epsilon)0` is retained and all references to `(0,0,4)4` are discarded. Since there are nineteen states in the minimized FSA, and each state corresponds to a different cone type, there are nineteen cone types in $SC(1)$.

```
What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)? 4

The following is the cone type automaton.  It recognizes
the language of geodesic words in the Schutzenberger
graph of 1:

Geodesic word acceptor for SC(1) (19 cone types):
(epsilon)0: a->(0,0,4)1 d->(0,0,4)7
(0,0,4)1: a->(0,0,4)1 b->(0,0,4)2 d->(0,0,4)7
(0,0,4)2: a->(0,0,4)1 A->(0,0,4)3 b->(0,1,5)2 d->(0,0,4)7
(0,0,4)3: B->(epsilon)0 d->(0,1,5)7
(0,0,4)7: a->(0,0,4)1 c->(0,0,4)6 d->(0,0,4)7
```

```
(0,0,4)6: a->(0,0,4)1 c->(7,0,3)6 d->(0,0,4)7 D->(0,0,4)5
(7,0,3)6: a->(0,0,4)1 c->(7,0,3)6 d->(0,0,4)7 D->(7,0,3)5
(7,0,3)5: a->(7,0,3)1 C->(7,0,3)4
(7,0,3)1: a->(0,0,4)1 b->(7,0,3)2 d->(0,0,4)7
(7,0,3)2: a->(0,0,4)1 A->(7,0,3)3 b->(0,1,5)2 d->(0,0,4)7
(7,0,3)3: d->(0,1,5)7
(0,1,5)7: a->(0,0,4)1 c->(0,1,5)6 d->(0,0,4)7
(0,1,5)6: a->(0,0,4)1 c->(7,0,3)6 d->(0,0,4)7 D->(0,1,5)5
(0,1,5)5: a->(7,0,3)1
(0,1,5)2: a->(0,0,4)1 A->(0,1,5)3 b->(0,1,5)2 d->(0,0,4)7
(0,1,5)3: B->(0,1,5)4 d->(0,1,5)7
(0,1,5)4: a->(0,0,4)1 c->(0,1,5)5 d->(0,0,4)7
(7,0,3)4: a->(0,0,4)1 b->(7,0,3)3 d->(0,0,4)7
(0,0,4)5: a->(7,0,3)1 C->(epsilon)0

What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)?
```

5.1.3 Sample output for $w = abcxaxabxabcy$

This example has some features not seen in the previous example. First, there are faces in the Schützenberger complex which share multiple edges (corresponding to the subwords ab and abc). Also, there are faces in which the geodesic sink occurs at the midpoint of an edge rather than at a vertex. In addition, this example shows that there can be two face types (b, f, k) differentiated only by the geodesic sink index k. Finally, the output of Option 4 shows that the minimization of the geodesic word acceptor can involve more identification of vertices than in the previous example; the PDA has 62 states in its seven face types, but the cone type automaton has only 36 states.

Note: To keep the listing to a reasonable length, the edges within most of the vertices has been suppressed from the output.

```
What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)? 8

Enter your relator w: abcxaxabxabcy

I have computed the Schutzenberger complex of 1 for
```

```
M=Inv<X|w=1>, w=abcxaxabxabcy.

Choose from among the following options:
  1) List the PDA (face types and transitions) for your inverse monoid
  2) Interactively test whether or not u=1; i.e., whether SC(1) accepts u
  3) Test whether u,v are R-related to 1, and if so, whether u=v
  4) Compute the cone type automaton which recognizes geodesics in SC(1)
  8) Enter a new relator w (your current monoid is discarded)
  9) Leave this program (quit and Ctrl-D also work)
What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)? 1

Your monoid is M=InvMonoid<X|w=abcxaxabxabcy=1>  7 face types:
FaceType(epsilon): BASECOMPLEX initial=(epsilon)0, terminal=(epsilon)0;
1 vertex
  Vertex (epsilon)0: attachedfacetype=(0,0,6.5), 2 edges:
    ( a ,            t )  ->  (   (0,0,6.5)1 ,  (epsilon)0 t )  +1
    ( Y ,            t )  ->  (  (0,0,6.5)12 ,  (epsilon)0 t )  +1

FaceType(0,0,6.5): FACE  12 vertices
  Vertex (0,0,6.5)1: attachedfacetype=(0,0,6.5), 47 edges:
    ( a ,            t )  ->  (   (0,0,6.5)1 ,  (0,0,6.5)1 t )  +1
    ( b ,            t )  ->  (   (0,0,6.5)2 ,            t )  +1
    ( Y ,            t )  ->  (  (0,0,6.5)12 ,  (0,0,6.5)1 t )  +1
    ( A ,   (epsilon)0 )  ->  (   (epsilon)0 ,          eps )  -1
    ( A ,   (0,0,6.5)1 )  ->  (   (0,0,6.5)1 ,          eps )  -1
    ( A ,   (0,0,6.5)2 )  ->  (   (0,0,6.5)2 ,          eps )  -1
    ( A ,   (0,0,6.5)3 )  ->  (   (0,0,6.5)3 ,          eps )  -1
    ( A ,   (0,0,6.5)5 )  ->  (   (0,0,6.5)5 ,          eps )  -1
    ( A ,   (0,0,6.5)7 )  ->  (   (0,0,6.5)7 ,          eps )  -1
    ( A ,   (0,0,6.5)8 )  ->  (   (0,0,6.5)8 ,          eps )  -1
    ( A ,  (0,0,6.5)10 )  ->  (  (0,0,6.5)10 ,          eps )  -1
    ( A ,  (0,0,6.5)11 )  ->  (  (0,0,6.5)11 ,          eps )  -1
    ( A ,  (0,0,6.5)12 )  ->  (  (0,0,6.5)12 ,          eps )  -1
    ( A ,   (0,1,6.5)2 )  ->  (   (0,1,6.5)2 ,          eps )  -1
    ( A ,   (0,1,6.5)3 )  ->  (   (0,1,6.5)3 ,          eps )  -1
    ( A ,   (0,1,6.5)5 )  ->  (   (0,1,6.5)5 ,          eps )  -1
    ( A ,   (0,1,6.5)7 )  ->  (   (0,1,6.5)7 ,          eps )  -1
    ( A ,   (0,1,6.5)8 )  ->  (   (0,1,6.5)8 ,          eps )  -1
    ( A ,  (0,1,6.5)10 )  ->  (  (0,1,6.5)10 ,          eps )  -1
    ( A ,  (0,1,6.5)11 )  ->  (  (0,1,6.5)11 ,          eps )  -1
    ( A ,  (0,1,6.5)12 )  ->  (  (0,1,6.5)12 ,          eps )  -1
    ( A ,     (0,2,8)3 )  ->  (     (0,2,8)3 ,          eps )  -1
    ( A ,     (0,2,8)5 )  ->  (     (0,2,8)5 ,          eps )  -1
    ( A ,     (0,2,8)7 )  ->  (     (0,2,8)7 ,          eps )  -1
    ( A ,     (0,2,8)8 )  ->  (     (0,2,8)8 ,          eps )  -1
```

```
    ( A ,    (0,2,8)10 )  ->  (    (0,2,8)10 ,          eps )  -1
    ( A ,    (0,2,8)11 )  ->  (    (0,2,8)11 ,          eps )  -1
    ( A ,    (0,2,8)12 )  ->  (    (0,2,8)12 ,          eps )  -1
    ( A ,   (0,2,6.5)3 )  ->  (   (0,2,6.5)3 ,          eps )  -1
    ( A ,   (0,2,6.5)5 )  ->  (   (0,2,6.5)5 ,          eps )  -1
    ( A ,   (0,2,6.5)7 )  ->  (   (0,2,6.5)7 ,          eps )  -1
    ( A ,   (0,2,6.5)8 )  ->  (   (0,2,6.5)8 ,          eps )  -1
    ( A ,  (0,2,6.5)10 )  ->  (  (0,2,6.5)10 ,          eps )  -1
    ( A ,  (0,2,6.5)11 )  ->  (  (0,2,6.5)11 ,          eps )  -1
    ( A ,  (0,2,6.5)12 )  ->  (  (0,2,6.5)12 ,          eps )  -1
    ( A ,   (0,3,9.5)5 )  ->  (   (0,3,9.5)5 ,          eps )  -1
    ( A ,   (0,3,9.5)7 )  ->  (   (0,3,9.5)7 ,          eps )  -1
    ( A ,   (0,3,9.5)8 )  ->  (   (0,3,9.5)8 ,          eps )  -1
    ( A ,  (0,3,9.5)10 )  ->  (  (0,3,9.5)10 ,          eps )  -1
    ( A ,  (0,3,9.5)11 )  ->  (  (0,3,9.5)11 ,          eps )  -1
    ( A ,  (0,3,9.5)12 )  ->  (  (0,3,9.5)12 ,          eps )  -1
    ( A ,     (0,3,9)5 )  ->  (     (0,3,9)5 ,          eps )  -1
    ( A ,     (0,3,9)7 )  ->  (     (0,3,9)7 ,          eps )  -1
    ( A ,     (0,3,9)8 )  ->  (     (0,3,9)8 ,          eps )  -1
    ( A ,    (0,3,9)10 )  ->  (    (0,3,9)10 ,          eps )  -1
    ( A ,    (0,3,9)11 )  ->  (    (0,3,9)11 ,          eps )  -1
    ( A ,    (0,3,9)12 )  ->  (    (0,3,9)12 ,          eps )  -1
  Vertex (0,0,6.5)2: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,0,6.5)3: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,0,6.5)4: attachedfacetype=(0,1,6.5), 3 edges:
  Vertex (0,0,6.5)5: attachedfacetype=(0,0,6.5), 5 edges:
  Vertex (0,0,6.5)6: attachedfacetype=(0,2,8), 3 edges:
    ( a ,           t )  ->  (   (0,0,6.5)7 ,            t )   0
    ( X ,           t )  ->  (   (0,0,6.5)5 ,            t )  -1
    ( Y ,           t )  ->  (    (0,2,8)12 ,  (0,0,6.5)6 t )  +1
  Vertex (0,0,6.5)7: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,0,6.5)8: attachedfacetype=(0,0,6.5), 5 edges:
  Vertex (0,0,6.5)9: attachedfacetype=(0,3,9.5), 3 edges:
  Vertex (0,0,6.5)10: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,0,6.5)11: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,0,6.5)12: attachedfacetype=(0,0,6.5), 48 edges:

FaceType(0,1,6.5): FACE  11 vertices
  Vertex (0,1,6.5)2: attachedfacetype=(0,0,6.5), 9 edges:
  Vertex (0,1,6.5)3: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,1,6.5)4: attachedfacetype=(0,1,6.5), 3 edges:
  Vertex (0,1,6.5)5: attachedfacetype=(0,0,6.5), 5 edges:
  Vertex (0,1,6.5)6: attachedfacetype=(0,2,8), 3 edges:
  Vertex (0,1,6.5)7: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,1,6.5)8: attachedfacetype=(0,0,6.5), 5 edges:
```

```
  Vertex (0,1,6.5)9: attachedfacetype=(0,3,9.5), 3 edges:
  Vertex (0,1,6.5)10: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,1,6.5)11: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,1,6.5)12: attachedfacetype=(0,0,6.5), 10 edges:

FaceType(0,2,6.5): FACE  10 vertices
  Vertex (0,2,6.5)3: attachedfacetype=(0,0,6.5), 6 edges:
  Vertex (0,2,6.5)4: attachedfacetype=(0,1,6.5), 3 edges:
  Vertex (0,2,6.5)5: attachedfacetype=(0,0,6.5), 5 edges:
  Vertex (0,2,6.5)6: attachedfacetype=(0,2,8), 3 edges:
  Vertex (0,2,6.5)7: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,2,6.5)8: attachedfacetype=(0,0,6.5), 5 edges:
  Vertex (0,2,6.5)9: attachedfacetype=(0,3,9.5), 3 edges:
  Vertex (0,2,6.5)10: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,2,6.5)11: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,2,6.5)12: attachedfacetype=(0,0,6.5), 7 edges:

FaceType(0,2,8): FACE  10 vertices
  Vertex (0,2,8)3: attachedfacetype=(0,0,6.5), 6 edges:
  Vertex (0,2,8)4: attachedfacetype=(0,1,6.5), 3 edges:
  Vertex (0,2,8)5: attachedfacetype=(0,0,6.5), 5 edges:
  Vertex (0,2,8)6: attachedfacetype=(0,2,6.5), 3 edges:
  Vertex (0,2,8)7: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,2,8)8: attachedfacetype=(0,0,6.5), 5 edges:
  Vertex (0,2,8)9: attachedfacetype=(0,3,9.5), 3 edges:
  Vertex (0,2,8)10: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,2,8)11: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,2,8)12: attachedfacetype=(0,0,6.5), 7 edges:

FaceType(0,3,9): FACE  9 vertices
  Vertex (0,3,9)4: attachedfacetype=(0,1,6.5), 3 edges:
  Vertex (0,3,9)5: attachedfacetype=(0,0,6.5), 5 edges:
  Vertex (0,3,9)6: attachedfacetype=(0,2,6.5), 3 edges:
  Vertex (0,3,9)7: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,3,9)8: attachedfacetype=(0,0,6.5), 5 edges:
  Vertex (0,3,9)9: attachedfacetype=(0,3,9.5), 3 edges:
  Vertex (0,3,9)10: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,3,9)11: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,3,9)12: attachedfacetype=(0,0,6.5), 5 edges:

FaceType(0,3,9.5): FACE  9 vertices
  Vertex (0,3,9.5)4: attachedfacetype=(0,1,6.5), 7 edges:
  Vertex (0,3,9.5)5: attachedfacetype=(0,0,6.5), 5 edges:
  Vertex (0,3,9.5)6: attachedfacetype=(0,2,6.5), 3 edges:
  Vertex (0,3,9.5)7: attachedfacetype=(0,0,6.5), 4 edges:
```

```
  Vertex (0,3,9.5)8: attachedfacetype=(0,0,6.5), 5 edges:
  Vertex (0,3,9.5)9: attachedfacetype=(0,3,9), 3 edges:
  Vertex (0,3,9.5)10: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,3,9.5)11: attachedfacetype=(0,0,6.5), 4 edges:
  Vertex (0,3,9.5)12: attachedfacetype=(0,0,6.5), 9 edges:
    ( a ,            t )  ->  (   (0,0,6.5)1 , (0,3,9.5)12 t )  +1
    ( C ,            t )  ->  (  (0,3,9.5)11 ,             t )  +1
    ( x ,            t )  ->  (     (0,3,9)4 ,  (0,3,9.5)9 t )  +1
    ( Y ,            t )  ->  (  (0,0,6.5)12 , (0,3,9.5)12 t )  +1
    ( y ,   (0,0,6.5)9 )  ->  (   (0,0,6.5)9 ,           eps )  -1
    ( y ,   (0,1,6.5)9 )  ->  (   (0,1,6.5)9 ,           eps )  -1
    ( y ,     (0,2,8)9 )  ->  (     (0,2,8)9 ,           eps )  -1
    ( y ,   (0,2,6.5)9 )  ->  (   (0,2,6.5)9 ,           eps )  -1
    ( y ,     (0,3,9)9 )  ->  (     (0,3,9)9 ,           eps )  -1

What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)? 4

The following is the cone type automaton.  It recognizes
the language of geodesic words in the Schutzenberger
graph of 1:

Geodesic word acceptor for SC(1) (36 cone types):
(epsilon)0: a->(0,0,6.5)1 Y->(0,0,6.5)12
(0,0,6.5)1: a->(0,0,6.5)1 b->(0,0,6.5)2 Y->(0,0,6.5)12
(0,0,6.5)2: a->(0,0,6.5)1 c->(0,0,6.5)3 Y->(0,0,6.5)12
(0,0,6.5)3: a->(0,0,6.5)1 x->(0,0,6.5)4 Y->(0,0,6.5)12
(0,0,6.5)4: a->(0,0,6.5)5 Y->(0,0,6.5)12
(0,0,6.5)5: a->(0,0,6.5)1 b->(0,0,6.5)2 x->(0,1,6.5)6 Y->(0,0,6.5)12
(0,1,6.5)6: Y->(0,2,8)12
(0,2,8)12: a->(0,0,6.5)1 C->(0,2,8)11 x->(0,3,9.5)4 Y->(0,0,6.5)12
(0,2,8)11: a->(0,0,6.5)1 B->(0,2,8)10 Y->(0,0,6.5)12
(0,2,8)10: a->(0,0,6.5)1 A->(0,2,8)9 Y->(0,0,6.5)12
(0,2,8)9: X->(0,0,6.5)2 Y->(0,3,9.5)12
(0,0,6.5)12: a->(0,0,6.5)1 C->(0,0,6.5)11 x->(0,3,9.5)4 Y->(0,0,6.5)12
(0,0,6.5)11: a->(0,0,6.5)1 B->(0,0,6.5)10 Y->(0,0,6.5)12
(0,0,6.5)10: a->(0,0,6.5)1 A->(0,0,6.5)9 Y->(0,0,6.5)12
(0,0,6.5)9: X->(0,0,6.5)8 Y->(0,3,9.5)12
(0,0,6.5)8: a->(0,0,6.5)1 B->(epsilon)0 c->(0,2,8)3 Y->(0,0,6.5)12
(0,2,8)3: a->(0,0,6.5)1 x->(0,2,8)4 Y->(0,0,6.5)12
(0,2,8)4: a->(0,2,8)5 Y->(0,0,6.5)12
(0,2,8)5: a->(0,0,6.5)1 b->(0,0,6.5)2 x->(epsilon)0 Y->(0,0,6.5)12
(0,3,9.5)12: a->(0,0,6.5)1 C->(0,3,9.5)11 x->(0,3,9)4 Y->(0,0,6.5)12
(0,3,9.5)11: a->(0,0,6.5)1 B->(epsilon)0 Y->(0,0,6.5)12
(0,3,9)4: a->(0,3,9)5 Y->(0,0,6.5)12
(0,3,9)5: a->(0,0,6.5)1 b->(0,0,6.5)2 x->(0,3,9)6 Y->(0,0,6.5)12
```

```
(0,3,9)6: a->(0,3,9)7 Y->(0,0,6.5)12
(0,3,9)7: a->(0,0,6.5)1 b->(0,3,9)8 Y->(0,0,6.5)12
(0,3,9)8: a->(0,0,6.5)1 c->(0,0,6.5)3 x->(0,3,9)9 Y->(0,0,6.5)12
(0,3,9)9: Y->(0,3,9.5)12
(0,3,9.5)4: a->(0,3,9.5)5 Y->(0,0,6.5)12
(0,3,9.5)5: a->(0,0,6.5)1 b->(0,0,6.5)2 x->(0,3,9.5)6 Y->(0,0,6.5)12
(0,3,9.5)6: a->(0,3,9.5)7 Y->(0,0,6.5)12
(0,3,9.5)7: a->(0,0,6.5)1 b->(0,3,9.5)8 Y->(0,0,6.5)12
(0,3,9.5)8: a->(0,0,6.5)1 c->(0,0,6.5)3 x->(0,3,9.5)9 Y->(0,0,6.5)12
(0,3,9.5)9: Y->(0,3,9)12
(0,3,9)12: a->(0,0,6.5)1 C->(0,3,9)11 x->(0,3,9.5)4 Y->(0,0,6.5)12
(0,3,9)11: a->(0,0,6.5)1 B->(0,3,9)10 Y->(0,0,6.5)12
(0,3,9)10: a->(0,0,6.5)1 A->(0,3,9)9 Y->(0,0,6.5)12

What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)? 9
Good-bye!
$
```

5.2 Overview of the program

The `spar` program consists of three components and five source files:

- **word** (source files `word.h` and `word.cc`): This component provides the definitions and procedures for for working with words. See Section 5.2.1.

- **sc** (source files `sc.h` and `sc.cc`): This is the main component of the program. It provides the definitions and procedures for building and working with the Schützenberger complex of 1 in inverse monoids of the form $M = \mathrm{Inv}\langle X \mid w = 1\rangle$ such that w is sparse. See Section 5.2.2.

- **spar** (source file `spar.cc`): This is the front-end program which calls the above components. The C++ source file `spar.cc` is listed in Section A.3 of the appendix.

5.2.1 The `word` program component

The `word` component of the program provides the definitions and functions for working with words in $(X \cup X^{-1})^*$, where $X = \{a, b, c, \ldots, y, z\}$ and $X^{-1} = \{A, B, C, \ldots, Y, Z\}$. This includes, for example, the function `substr` (beginning at (A2) in Section A.1), which computes the cyclic substring defined in Section 2.4.1: if w is a word, then w`.substr(`i`,`l`)` $= w[i, l]$.

The most important function in the `word` program component is the function `check_sparse_with_violators()` (see the code beginning at (A3)), which checks whether or not a word is sparse, and if it is not sparse, returns a pair of pointed pieces which overlap. It makes use of the `pointedpiece` data structure at (A1). In Section 3.2.1 a pointed piece, with respect to a cyclically reduced word w, was defined as a pair $((i_q, u, \eta_q), (i_r, u, \eta_r))$ of segments, the associated and home segments, respectively. The information stored in the computer is equivalent, but slightly different. Note that if

$$((i_q, u, \eta_q), (i_r, u, \eta_r))$$

is a pointed piece, then

$$((i_q + \eta_q|u|, u^{-1}, -\eta_q), (i_r + \eta_r|u|, u^{-1}, -\eta_r))$$

is an equivalent (with respect to $\approx$) pointed piece, so we may assume all pointed pieces are of the form $((i_q, u, 1), (i_r, u, \eta))$. In addition, the word u is determined by i_q and its length m: $u = w[i_q, m] = w[i_r, \eta m]$. Thus we store the four integers $i = i_q$, $j = i_r$, m, and η, together with the word w for each pointed piece, as shown in the definition of `pointedpiece` at (A1).

Function `check_sparse_with_violators()` searches for pointed pieces by looking for places in the interior of w at which the beginning or end of the word can be

seen. When it finds such a place (`k`), it counts forward (`fl`) and backward (`bl`) to find the maximal pointed piece. After it finds a pointed piece, it first checks to see if the pointed piece overlaps itself. If not, it checks the pointed piece against the set of pointed pieces already found to see if it overlaps any of those pointed pieces. If the new pointed piece does not overlap any of the previous pointed pieces, the new pointed piece is added to the set of pointed pieces itself, and the loop continues looking for additional pointed pieces.

When two pointed pieces are found to overlap, the function returns false and returns the two pointed pieces. This allows the calling program to know where sparseness was violated. If the `k` loop completes without finding pointed pieces that overlap, there can be no overlaps, and the function returns true.

5.2.2 The `sc` program component

The `sc` program component provides the definitions and functions for building and working with the Schützenberger complex of 1 for inverse monoids of the form $M = \mathrm{Inv}\langle X \mid w = 1\rangle$ where w is sparse. The `InvMonoid` data structure (A6) serves as a container to hold the relator w and a pointer to the `BASECOMPLEX`, i.e., the vertex O of $SC(1)$. The main function of `sc` is the function `attachface` (A10), which recursively calls itself (after initially being called from (A8)) to build the PDA which represents $SC(1)$, as described in Section 4.3.2.

Another useful function of the `sc` program component is the function `advancedto` (A12). This function advances an instantaneous description (ID) for the PDA—that is, given an ID I_1 it computes an ID I_2 such that $I_1 \vdash^* I_2$—until either (1) the entire word is read, (2) a specific terminal vertex is attained, or (3) a maximum number of steps is reached. The calling program may use one or more of these stopping criteria as needed. For example, `spar` calls `advancedto` at (A17) to accomplish the step-

by-step reading of a word in the PDA, as illustrated by the sample run in Section 5.1.2.

Instantaneous descriptions are stored in the data structure `InstDesc` (A7). This is basically the same as an ID as defined in Section 2.1.6, except that rather than storing only the remaining portion of the word being read, the entire word is stored along with an index (`i`) to the beginning of the unread portion of the word.

The function `ConeTypeFSA` (A14) produces the cone type automaton for $SC(1)$. This is essentially a straightforward implementation of the FSA minimization algorithm from [HU79, p. 70]. One important thing to keep in mind is that the minimization algorithm of [HU79] assumes that the transition function of the FSA is a total function (defined on all of its domain). Since that may not be the case with the PDA for $SC(1)$, the `ConeTypeFSA` function uses an imaginary fail state to which all missing edges map. It accomplishes this by numbering the vertices and using the special number `FAILSTATE` $= -1$ for the fail state. The function `dest` (A13) then returns, as needed, either the actual destination vertex of an edge, or `FAILSTATE`.

When the minimization algorithm completes, it identifies sets of equivalent vertices, as indicated by the `marked` set, by changing the vertex numbering so all vertex numbers point to the smallest-numbered equivalent vertex (A15). The routine (A16) uses this information to output the minimized FSA.

Appendix A

Computer source code

This appendix contains a complete listing of the source code of the program used to produce the sample output in Section 5.1. The source files `word.cc`, `sc.cc`, and `spar.cc` contain the C++ algorithms for the corresponding components of the program, as described in Section 5.2. The two header files, `word.h` and `sc.h`, provide the definitions for those components which are needed by a program (such as `spar` or another program) which wishes to make use of the `word` and `sc` components.

A.1 Procedures for words

The source files `word.h` and `word.cc` provide the definitions and functions for working with words.

A.1.1 Source file `word.h`

```
// word.h - Definitions for words and wordsets

#ifndef WORD_H
#define WORD_H

#include <string>
```

```
#include <set>
#include <list>
#include <vector>
#include <map>
#include <algorithm>
#include <iostream>
#include <iomanip>
#include <fstream>
#include <cstdlib>
#include <cctype> // for toupper and tolower
#include <cstdio> // for sprintf

#include <limits.h>
static const int INFTY=INT_MAX;


/*
** Miscellaneous
*/

char *itoa (int n);
string reducestr (string x);
inline char cinv (const char x) {
    return (islower(x))?toupper(x):tolower(x); }
inline int mod (const int a, const int n)
   { return (n==0)?a:((a<0)?(((a%n)+n)%n):(a%n)); }


/*
** Words and Sets of Words
*/

class pointedpiece;


class Word {

  public:
    Word (const string& x="1"); // the constructor; default value 1
    Word operator+ (const Word& y) const // free monoid product
        { return Word(w+y.w); }
    Word operator* (const Word& y) const // free group product
        { return Word(reducestr(w+y.w)); }
    bool operator< (const Word& y) const;
    bool operator== (const Word& y) const {return w==y.w;}
```

```
    Word reduced() const {return Word(reducestr(w));}
    string str() const {return w;}
    int len() const {return w.size();}
    Word substr(int pos, int len) const;
    Word right(int pos) const { return substr(pos,len()-pos); }
    Word inv() const { return substr(0,-len()); }
    bool iscycred() const;
    bool issparse() const;
    bool check_sparse_with_violators
        (pointedpiece& olx, pointedpiece& oly) const;
    // check_sparse_with_violators() assumes w is cyclically reduced.
    friend ostream& operator<< (ostream& s, const Word& x);
    friend ostream& operator<< (ostream& s, const set<Word>& A);

  private:
    string w;
};

extern const Word Word1; // empty word
```

(A1)

```
class pointedpiece {
  public:
    // The pointed piece has associated segment (i,w[i,m],+1)
    // and home segment (j,w[j,eta*m],eta) .
    Word w; // word for which this is a pointed piece.
    int i; // starting position of associated segment
    int j; // starting position of home segment
    int eta; // direction (+/-1) of home segment
    int m; // length of segments (>0)
    pointedpiece(Word u=Word1, int ti=0, int tj=0, int teta=0, int tm=0)
        { w=u; i=mod(ti,w.len()); j=mod(tj,w.len()); eta=teta; m=tm; }
    bool operator< (const pointedpiece& y) const {
        if (w<y.w) return true; else if (w>y.w) return false;
        if (i<y.i) return true; else if (i>y.i) return false;
        if (j<y.j) return true; else if (j>y.j) return false;
        if (m<y.m) return true; else if (m>y.m) return false;
        if (eta<y.eta) return true; else return false;
    }
    bool operator== (const pointedpiece& y) const {
        return w==y.w && i==y.i && j==y.j && m==y.m && eta==y.eta; }
    bool overlaps (const pointedpiece& y) const;
    friend ostream& operator<< (ostream& s, const pointedpiece& x);
};

#endif // WORD_H
```

A.1.2 Source file `word.cc`

```
// word.cc - Words and sets of words

#include "word.h"


Word::Word (const string& x)
{
  if (x=="1" || x=="$" || x=="word1" || x=="eps" || x=="epsilon")
    w="";
  else
    w=x;
}


const Word Word1=Word(""); // empty word


bool Word::operator< (const Word& y) const
{
  int i,n=w.length(),s=n-y.w.length();
  if (s<0)
    return true;
  if (s>0)
    return false;
  const char *a=w.c_str(),*b=y.w.c_str();
  const char *ap,*bp;
  for (ap=a, bp=b; *ap==*bp && *ap; ++ap,++bp)
      ;
  if (!*ap)
    return false; // strings are equal
  if (toupper(*ap)==toupper(*bp))
    return *ap>*bp; // I want lower<upper, ASCII is lower>upper.
  else
    return toupper(*ap)<toupper(*bp);
}



Word Word::substr(int pos, int len) const                    (A2)
{
    int n=w.size();
    string s="";
    if (n>0 && len!=0) {
```

```
        if (len>0)
            while (len--)
                s+=w[mod(pos++,n)];
        else
            while (len++)
                s+=cinv(w[mod(--pos,n)]);
    }
    return Word(s);
}


bool Word::iscycred() const
{
    return len()==0 || (substr(0,-1)!=substr(0,+1) && w==reducestr(w));
}


bool inzone(int k, int i, int m, int n)
// Returns whether k is in {i,i+1,...,i+m} if m>0;
//                       or {i,i-1,...,i-|m|} if m<0,
// where the values are mod n.
{
    if (m>0) {
        int d=mod(k-i,n);
        if (0<=d && d<=m) return true;
        else return false;
    }
    else { // m<=0
        int d=mod(i-k,n);
        if (0<=d && d<=-m) return true;
        else return false;
    }
}


bool zonesintersect(int i, int mi, int j, int mj, int n)
// Returns whether or not z(i,mi) intersects z(j,mj) based
// on a word length of n.  The zones intersect if the beginning
// or end of either is in the zone of the other.
{
    if (inzone(i   ,j,mj,n)) return true;
    if (inzone(i+mi,j,mj,n)) return true;
    if (inzone(j   ,i,mi,n)) return true;
    if (inzone(j+mj,i,mi,n)) return true;
    return false;
```

```
}


bool pointedpiece::overlaps (const pointedpiece& y) const
{
    // This function determines whether the pointed pieces overlap.
    // Think of *this as pointed piece (q,r) in the definition of
    // overlap and y as pointed piece (s,t).

    // z(q) and z(t) intersect
    if (zonesintersect(i,  m,  y.j,y.eta*y.m,w.len()))
        return true;

    // z(s) and z(r) intersect
    if (zonesintersect(y.i,y.m,  j,eta*m,    w.len()))
        return true;

    // z(q) and z(s) intersect and (q,r) and (s,t) aren't the same
    if (*this==y) return false;
    if (zonesintersect(i,m,y.i,y.m,w.len()))
        return true;

    return false;
}
```

(A3)

```
bool Word::check_sparse_with_violators
   (pointedpiece& olx, pointedpiece& oly) const
{
    // This function assumes w is cyclically reduced and of length at
    // least 2.  If w is not sparse, this function returns false and
    // olx and oly are pointed pieces that overlap.  If w is sparse,
    // this function returns true and the values of olx and oly are
    // undefined.

    set<pointedpiece> pp; // set of pointed pieces found so far

    for (int k=1; k<w.size(); ++k) {
        int eta;
        if (substr(k,+1)==substr(0,+1)||substr(k,-1)==substr(0,-1))
            eta=+1;
        else if (substr(k,+1)==substr(0,-1)||substr(k,-1)==substr(0,+1))
            eta=-1;
        else
            continue; // no pointed piece here; continue with next k
```

```
        int fl=0; // length of forward matching
        while (substr(k+fl,+1)==substr(0+eta*fl,eta) && fl<len())
            ++fl;
        int bl=0; // length of backword matching
        if (fl<len()) {
            while (substr(k-bl,-1)==substr(0-eta*bl,-eta))
                ++bl;
        }
        else {
            // Word is a proper power.  The forward matching shows
            // this (by matching the full length of the word), so just
            // let bl be 0.  The pointed piece is going to be found to
            // overlap itself in the next step.
        }

        olx=pointedpiece(*this,k-bl,0-eta*bl,eta,bl+fl);
        // We explicitly check for olx overlapping itself rather than
        // just putting olx in pp and allowing the check to occur
        // inside the loop because if olx does overlap itself, we
        // prefer to report that as being the sparseness violation.
        if (olx.overlaps(olx)) {
            oly=olx;
            return false;
        }
        for (set<pointedpiece>::const_iterator
                 p=pp.begin(); p!=pp.end(); ++p) {
            if (olx.overlaps(*p)) {
                oly=*p;
                return false;
            }
        }
        pp.insert(olx);
    }

    return true;
}


bool Word::issparse() const
{
    if (len()<2)
        return false;
    if (!iscycred())
        return false;
    pointedpiece olx,oly;
```

```
    return check_sparse_with_violators(olx,oly);
}



/*
** Miscellaneous
*/

char *itoa (int n)
{
  static char buf[100];
  sprintf(buf,"%d",n);
  return buf;
}

string reducestr (string x) // reduce word x (cancel inverses)
{
  string::size_type i=0;
  while (i+1<x.length()) {
    if (x[i]==cinv(x[i+1])) {
      x.erase(i,2);
      if (i>0) --i;
    }
    else
      ++i;
  }
  return x;
}



/*
** Output routines
*/

ostream& operator<< (ostream& s, const Word& x)
{
  return s << (x.w==""?"epsilon":x.w);
}


ostream& operator<< (ostream& s, const set<Word>& A)
{
  int n=0;
```

```
  s << "{";
  for (set<Word>::iterator i=A.begin(); i!=A.end(); ++i,++n) {
    if (n>0) s << ',';
    s << *i;
  }
  s << "}";
  return s;
}


ostream& operator<< (ostream& s, const pointedpiece& x)
{
    return s << "(("
             << x.i << ","
             << x.w.substr(x.i,x.m) << ","
             << "1),("
             << x.j << ","
             << x.w.substr(x.j,x.eta*x.m) << ","
             << x.eta << "))";
}
```

A.2 Procedures for Schützenberger complexes

The source files `sc.h` and `sc.cc` contain the definitions and functions for storing and using inverse monoids and Schützenberger complexes.

A.2.1 Source file `sc.h`

```
// sc.h - Definitions for Schutzenberger complexes

#ifndef SC_H
#define SC_H

#include "word.h"

const int fieldwidth=13;

class InvMonoid;
class SchComplex;
class FaceType;
class Face;
```

```
class Vertex;
class TransDom;
class TransRan;
class InstDesc;


class TransDom { // domain of transition function
  public:
    Word x;    // letter labeling edge
    Vertex *t; // top stack letter
               // (0 indicates it can be anything---i.e., "t")
    TransDom(Word y=Word1, Vertex *s=0) { x=y; t=s; }
    bool operator< (const TransDom& y) const {
        if (t<y.t) return true; else if (t>y.t) return false;
        if (x<y.x) return true; else            return false;
    }
    bool operator== (const TransDom& y) const {
        return (t==y.t) && (x==y.x); }
    friend ostream& operator<< (ostream& s, const TransDom& td);
};


class TransRan { // range of transition function
  public:
    int dir; // direction (-1:toward, 0:neither, +1:away from origin)
    Vertex *destp; // destination vertex of edge
    enum TransStackOp { NONE, PUSH, POP } stkop;
    Vertex *pushed; // vertex to push if stkop==PUSH
    TransRan(int d=0, Vertex *dp=0, TransStackOp so=NONE, Vertex *vp=0) {
        dir=d; destp=dp; stkop=so; pushed=vp; }
    bool operator== (const TransRan& y) const {
        return (dir==y.dir) && (destp==y.destp) && (stkop==y.stkop)
            && (pushed==y.pushed); }
    friend ostream& operator<< (ostream& s, const TransRan& tr);
};

extern const TransRan NullTransRan;


class FaceType {
  public:
    InvMonoid* Mp; // Mp==0 indicates empty (dummy) facetype
    enum { BASECOMPLEX, FACE } mode;
    Word baseword; // BASECOMPLEX: ==Word1, word used to build complex
    int b,f;   // FACE: last index folded when reading fwd or back
```

```
    int sink2; // FACE: twice the index of geodesic sink

    FaceType() { Mp=0; } // EmptyFaceType
    FaceType (InvMonoid* Mptr, Word t);
    FaceType (InvMonoid* Mptr, int j=0, int i=0, int k=0);
    int bvmod(int j) const; // 1..n
    int fvmod(int i) const; // 0..n-1
    int bv(void) const { return bvmod(b); }
    int fv(void) const { return fvmod(f); }
    bool operator< (const FaceType& y) const {
        if (Mp<y.Mp) return true; else if (Mp>y.Mp) return false;
        if (mode<y.mode) return true; else if (mode>y.mode) return false;
        if (mode==BASECOMPLEX) return baseword<y.baseword;
        if (f<y.f)         return true; else if (f>y.f) return false;
        if (b<y.b)         return true; else if (b>y.b) return false;
        if (sink2<y.sink2) return true; else            return false;
    }
    bool operator== (const FaceType& y) const {
        if (Mp!=y.Mp) return false;
        if (mode!=y.mode) return false;
        if (mode==BASECOMPLEX) return baseword==y.baseword;
        return (f==y.f) && (b==y.b) && (sink2==y.sink2);
    }
    friend ostream& operator<< (ostream& s, const FaceType& ct);
};

extern const FaceType EmptyFaceType;
```

(A4)

```
class Vertex {
 public:
    Face *ownerp; // face to which vertex belongs
    int index; // index of vertex inside its face (the vp[] array)
    FaceType attachedfacetype; // type of face attached at this vertex
    map<TransDom,TransRan> edge; // edges to other vertices
    Vertex(Face *owner=0, int i=0)  {
        ownerp=owner; index=i; attachedfacetype=EmptyFaceType; }
    void fixgeodesics(void);
    void maxfoldparams(int* bip, Vertex* *bpp, int* fip, Vertex* *fpp,
                       int* ep);
    void attachface(void);
    TransRan findtrans (TransDom td);
    friend string vertexstr (const Vertex* vp);
    friend ostream& operator<< (ostream& s, const Vertex* vp);
    friend ostream& operator<< (ostream& s, const Vertex& v);
};
```

(A5)

```
class Face {
 public:
    FaceType facetype;
    vector<Vertex *> vp; // ptrs to vertices of face
    // facetype.mode==FACE: indexed facetype.fv()+1..facetype.bv()-1.
    // facetype.mode==BASECOMPLEX: index 0 only

    // for facetype.mode==BASECOMPLEX:
    Vertex *initp; // initial vertex of base complex
    Vertex *termp; // terminal vertex of base complex

    Face (FaceType ftyp);
    void buildfacebottom (int f, int b, int sink2);
    void BuildBaseComplex(void);
    friend ostream& operator<< (ostream& s, const Face& C);
};


class SchComplex {
 public:
    InvMonoid *Mp; // inverse monoid to which this SchComplex belongs
    Face *bcp; // underlying base complex
    SchComplex(InvMonoid* Mptr, Word t);
    bool accepts(Word t);
    InstDesc advance (InstDesc id, Vertex *term, int plus=0);
    friend ostream& operator<< (ostream& s, const SchComplex& sc);
};
```

(A6)

```
class InvMonoid { // =Inv<X|w=1>
 public:
    Word w; // relator for inverse monoid
    int n; // word len
    SchComplex *sc1p; // SchComplex of 1
    map<FaceType, Face *> ftyptbl; // all known face types
    InvMonoid() { n=0; sc1p=0; } // dummy InvMonoid
    InvMonoid (Word u);
    Word trim_identity_words (Word t);
    bool sc1accepts (Word u) { return sc1p->accepts(u); }
    friend ostream& operator<< (ostream& s, const InvMonoid& M);
};
```

```
class InstDesc { // instantenous description                    (A7)
  public:
    SchComplex *scp;
    Vertex *qp; // current state in PDA
    Word u;     // word being read
    int i;      // index to current position in word
    list<Vertex *> stk; // stack
    InstDesc() { scp=0; }
    InstDesc(SchComplex *sp, Word t=Word1, int j=0);
    InstDesc advancedto (Vertex *term=0,
                         int plus=0, int maxcount=-1) const;
    bool futequal (const InstDesc& y) const;
    friend ostream& operator<< (ostream& s, const InstDesc& id);
};


class ConeTypeFSA {
  public:
    SchComplex *scp; // SchComplex on which this FSA is based
    map<Vertex *,int> iofv;
    vector<Vertex *> vofi;
    vector<int> iofi;
    int nstate;
    set<Word> alphabet;
    ConeTypeFSA(SchComplex *sp);
    int dest(int vi, Word x);
    friend ostream& operator<< (ostream& s, const ConeTypeFSA& fsa);
};


// Constants
extern const FaceType EmptyFaceType;
extern const TransRan NullTransRan;
extern Vertex DummyVertex;
extern Vertex *const DummyStackLetter;

#endif // SC_H
```

A.2.2 Source file `sc.cc`

```
// sc.cc - Build Schutzenberger complex
```

```
#include "sc.h"
#include <sstream>


InvMonoid::InvMonoid (Word u)
{
    w=u;
    n=w.len();
    sc1p=new SchComplex(this,Word1);
}


SchComplex::SchComplex (InvMonoid *Mptr, Word t)
{
    /* assumes t=Word1 */
    Mp=Mptr;
    bcp=new Face(FaceType(Mp,t));
    bcp->vp[0]->attachface(); // build SC(1)
    Mp->ftyptbl[bcp->facetype]=bcp;
}


Face::Face (FaceType ftyp)
// This function creates the vertices of a face and creates all
// transitions between vertices within the face.  No push/pop
// transitions between faces are created; they are created in
// SchComplex() or attachface().
{
    facetype=ftyp;
    if (ftyp.mode==FaceType::BASECOMPLEX) {                          (A8)
        vp.reserve(1);
        initp=termp=new Vertex(this,0);
        vp.push_back(initp); // set vp[0]=initp=termp
    }
    else { // FACE
        vp.reserve(facetype.Mp->w.len());
        buildfacebottom(facetype.bv(),facetype.fv(),facetype.sink2);
    }
}


FaceType::FaceType (InvMonoid* Mptr, Word t)
{
    mode=BASECOMPLEX;
    Mp=Mptr;
```

```
    baseword=Mp->trim_identity_words(t);
}


FaceType::FaceType (InvMonoid* Mptr, int j=0, int i=0, int k=0)
{
    mode=FACE;
    Mp=Mptr;
    b=fvmod(j);
    f=fvmod(i);
    sink2=k;
}


int FaceType::bvmod(int j) const { return mod(j-1,Mp->n)+1; } // 1..n
int FaceType::fvmod(int i) const { return mod(i,  Mp->n)  ; } // 0..n-1

void Face::buildfacebottom (int b, int f, int sink2)                    (A9)
// This routine creates the "bottom half," or unfolded portion, of a
// new face, added to the vp[] array of this face.  These are the new
// edges of the face, which do not fold onto its predecessor or the
// pre-existing edges of a base complex.
//
// f, b, sink2 = vertices with face indices f+1 .. b-1 are created,
// and the geodesic directions (on edges) are set so that the sink is
// at index sink2/2.
//
// Note that this function does not create any transitions from the
// vertices corresponding to face indices f and b.  That is the
// responsibility of the calling routine.
{
    vp.resize(b);
    // Vector vp[] now has entries vp[f+1],...,vp[b-1],
    // initialized to 0.  This is where pointers to the new vertices
    // will be stored:
    for (int i=f+1; i<=b-1; ++i)
        vp[i]=new Vertex(this,i);

    // Create edges going away from index f:
    for (int i=f+1; 2*(i+1)<=sink2; ++i) {
        Word x=facetype.Mp->w.substr(i,+1);
        vp[i]  ->edge[TransDom(x,0)]
            =TransRan(+1,vp[i+1],TransRan::NONE);
        vp[i+1]->edge[TransDom(x.inv(),0)]
            =TransRan(-1,vp[i]  ,TransRan::NONE);
    }
```

```
    // Create edges going away from index b:
    for (int j=b-1; 2*(j-1)>=sink2; --j) {
        Word x=facetype.Mp->w.substr(j,-1);
        vp[j]  ->edge[TransDom(x,0)]
            =TransRan(+1,vp[j-1],TransRan::NONE);
        vp[j-1]->edge[TransDom(x.inv(),0)]
            =TransRan(-1,vp[j]  ,TransRan::NONE);
    }

    // Create the last edge (with no geodesic direction), if the
    // sink lands in the middle of an edge:
    if (sink2%2) {
        // if sink2 is odd, there's an edge with no direction

        int i=sink2/2;
        Word x=facetype.Mp->w.substr(i,+1);
        vp[i]  ->edge[TransDom(x,0)]
            =TransRan( 0,vp[i+1],TransRan::NONE);
        vp[i+1]->edge[TransDom(x.inv(),0)]
            =TransRan( 0,vp[i]  ,TransRan::NONE);
    }
}
```

```
void Vertex::attachface(void)                                    (A10)
// This function determines what kind of face should be attached
// at the vertex, by looking forward and back to see how much folding
// can occur.  If a FACE of the required type exists, appropriate
// PUSH/POP transitions are added.  If a FACE of the required type
// does not exist, it is created and the PUSH/POP transitions are
// added.  If new vertices are created, attachface() is called
// recursively to attach faces at the new vertices.
{
    InvMonoid *Mp=ownerp->facetype.Mp;
    //int n=Mp->n;
    int b,f,e;
    Vertex *bp,*fp;
    maxfoldparams(&b,&bp,&f,&fp,&e);

    attachedfacetype=FaceType(Mp,b,f,b+f-e);
    // int r=j-i; // number of unfolded edges
    // e=d(0,fp)-d(0,bp)
    // kf=f+(r-e)/2, kb=b-(r+e)/2
    //   => k=(kb+kf)/2 => 2k=kb+kf=f+b-e = sink2
```

```
    // Get the desired face
    Face *ncp;
    if (Mp->ftyptbl.find(attachedfacetype)==Mp->ftyptbl.end()) {
        // FACE doesn't exist; create it.
        ncp=new Face(attachedfacetype);
        Mp->ftyptbl[attachedfacetype]=ncp;
        for (int i=f+1; i<=b-1; ++i)
            ncp->vp[i]->attachface();
    }
    else {
        ncp=Mp->ftyptbl[attachedfacetype];
    }

    // Create the between-face (pushing/popping) transitions
    Word x;
    Vertex *np;
    x=Mp->w.substr(f,+1);
    np=ncp->vp[f+1];
    // Stack "letter" (Vertex *)0 means that it doesn't matter what's
    // on the top of the stack.  This trick seems nicer than creating
    // transitions for all possible vertices.
    fp->edge[TransDom(x,0)]         =TransRan(+1,np,TransRan::PUSH,this);
    np->edge[TransDom(x.inv(),this)]=TransRan(-1,fp,TransRan::POP);
    x=Mp->w.substr(b,-1);
    np=ncp->vp[b-1];
    bp->edge[TransDom(x,0)]         =TransRan(+1,np,TransRan::PUSH,this);
    np->edge[TransDom(x.inv(),this)]=TransRan(-1,bp,TransRan::POP);
}
```

```
void Vertex::maxfoldparams(int* bip, Vertex* *bpp, int* fip,          (A11)
                           Vertex* *fpp, int* ep)
// This function looks to see what happens if a face is attached at
// the current vertex and folded.  It returns b (in 1..n) and f (in
// 0..n-1) as the vertex indices (within the face being sewn on) at
// which the folding ends.  bp and bf are the corresponding Vertex *'s
// of the actual vertices (in the predecessor face).  e is the net
// count of geodesic directions traversed; in other words, e is
// d(0,fp)-d(0,bp), or how much closer b is to 0 (the origin of the
// Sch complex) than f.
{
    InvMonoid *Mp=ownerp->facetype.Mp;
    int n=Mp->n;
    int b=n; Vertex* bp=this;
    int f=0; Vertex* fp=this;
```

```
    int e=0; // d(0,fp)-d(0,bp), how much closer to 0 b is than f
    // e<0 => face folded back toward origin

    /* Look forward/backward for end of part onto which the new face
       folds.  Sparseness guarantees these loops will terminate. */
    map<TransDom,TransRan>::const_iterator dp;
    while ((dp=fp->edge.find(TransDom(Mp->w.substr(f,+1),0)))
           !=fp->edge.end()) {
        fp = dp->second.destp;
        e += dp->second.dir;
        ++f;
    }

    while ((dp=bp->edge.find(TransDom(Mp->w.substr(b,-1),0)))
           !=bp->edge.end()) {
        bp = dp->second.destp;
        e -= dp->second.dir;
        --b;
    }

    // store the answers
    *bip=b;
    *bpp=bp;
    *fip=f;
    *fpp=fp;
    *ep=e;
}



/*
** Routines to read within a PDA
*/

InstDesc::InstDesc(SchComplex *sp, Word t, int j)
{
    scp=sp;
    qp=scp->bcp->initp;
    u=t;
    i=j;
    stk.push_front(DummyStackLetter); // initial letter on stack
}


bool InstDesc::futequal (const InstDesc& y) const
```

```
// Returns whether the two IDs are equal, ignoring the portions
// of the words already read.
{
    if (scp!=y.scp) return false;
    if (qp!=y.qp) return false;
    if (u.right(i)!=y.u.right(y.i)) return false;
    if (stk!=y.stk) return false;
    return true;
}


TransRan Vertex::findtrans (TransDom td)
{
    map<TransDom,TransRan>::const_iterator ep;
    for (ep=edge.begin(); ep!=edge.end(); ++ep) {
        if (ep->first.x==td.x) {
            if (ep->first.t==0 || ep->first.t==td.t)
                return ep->second;
        }
    }
    return NullTransRan;
}
```

(A12)

```
InstDesc InstDesc::advancedto (Vertex *term=0, int plus=0,
                               int maxcount=-1) const
/* Reads id in the PDA *scp, until the end of the word is reached,
until Vertex *term is reached, or until maxcount transitions have been
made, whichever comes first.  If plus is set, then at least one edge
must be read before Vertex *term is attained.  A maxcount of -1 means
no maximum count.  The function returns the final InstDesc.
Advancedto() may be called with all arguments 0 (the default) to force
as much of the word as possible to be read.  Note that term is assumed
to be in the underlying Munn tree.  Then, the only way it can be
reached is when the stack is empty. */
{
    InstDesc id=*this;
    TransRan tr;
    int maxi;

    if (maxcount==-1)
        maxi=u.len(); // read to the end of the word
    else
        maxi=id.i+maxcount;
    while (
        (id.i<maxi)
```

```
        && ( id.qp!=term || (plus && id.i==i) )
        && (tr=id.qp->findtrans(
            TransDom(id.u.substr(id.i,1),id.stk.front()) ))!=NullTransRan
        ) {

        ++id.i;
        id.qp=tr.destp;
        if (tr.stkop==TransRan::PUSH)
            id.stk.push_front(tr.pushed);
        else if (tr.stkop==TransRan::POP)
            id.stk.pop_front();
    }
    return id;
}


Word InvMonoid::trim_identity_words (Word t)
// This routine requires sc1p be properly set for nonempty words.  It
// correctly returns the empty word if t is empty, regardless of
// whether sc1p has been defined or not.
{
    if (t.len()==0)
        return t;
    int i=0;
    while (i<t.len()) {
        InstDesc newid=InstDesc(sc1p,t,i).advancedto(sc1p->bcp->termp,1);
        if (i!=newid.i && newid.qp==sc1p->bcp->termp) {
            t=t.substr(0,i)+t.right(newid.i); // cut out part equal to 1
            // keep i at its current value to check next substr
        }
        else {
            ++i;
        }
    }
    return t;
}


bool SchComplex::accepts(Word t)
{
    InstDesc id=InstDesc(this,t,0).advancedto();
    return id.qp==bcp->termp && id.i==t.len();
    /* This word is accepted if all of it was read (id.i=t.len) AND
       reading stopped at the terminal vertex. */
}
```

```
/*
** Routines to produce the cone type automaton
*/

// The routines below implement the finite-state automaton
// minimization algorithm in Hopcroft and Ullman to produce the cone
// type automaton.  Note that the minimization algorithm assumes there
// is a transition for every letter of the alphabet.  Since this may
// not be case in the edge's, the implementation below makes use of an
// imaginary fail state, FAILSTATE.  All other states are accept
// states.

const int FAILSTATE=-1; // -1 indicates sink state (the only fail state)

class statepair {
public:
    int x,y;
    statepair(int a, int b) {
        if (a>b) { x=b; y=a; } else { x=a; y=b; }
    }
    bool operator< (const statepair& t) const {
        if (x<t.x) return true; else if (x>t.x) return false;
        if (y<t.y) return true; else return false;
    }
};

int ConeTypeFSA::dest(int vi, Word x)                                    (A13)
{
    if (vi==FAILSTATE)
        return FAILSTATE;
    map<TransDom,TransRan>::const_iterator
        dp=vofi[vi]->edge.find(TransDom(x,0));
    if (dp==vofi[vi]->edge.end() || dp->second.dir!=+1)
        return FAILSTATE;
    return iofv[dp->second.destp];
}


void FsaNumberChildren(Vertex* vp,
   map<Vertex *,int>& iofv, vector<Vertex *>& vofi, set<Word>& alphabet)
{
    if (iofv.find(vp)==iofv.end()) {
        // haven't done this vertex yet, add to the list
```

```
        int n=vofi.size();
        iofv[vp]=n;
        vofi.push_back(vp); // add vofi[n] entry
        for (map<TransDom,TransRan>::const_iterator i=vp->edge.begin();
             i!=vp->edge.end(); i++) {
            alphabet.insert(i->first.x);
            if (i->second.dir==+1) {
                FsaNumberChildren(i->second.destp,iofv,vofi,alphabet);
            }
        }
    }
}


void markpair(statepair pq, set<statepair>& marked,
              map<statepair,set<statepair> >& listof)
{
    if (marked.find(pq)==marked.end()) {
        marked.insert(pq);
        for (set<statepair>::const_iterator sp=listof[pq].begin();
             sp!=listof[pq].end(); ++sp) {
            markpair(*sp,marked,listof);
        }
    }
}


ConeTypeFSA::ConeTypeFSA(SchComplex *sp)                              (A14)
{
    scp=sp;
    FsaNumberChildren(scp->bcp->initp,iofv,vofi,alphabet);
    nstate=vofi.size();
    set<statepair> emptyset;
    map<statepair,set<statepair> > listof;
    set<statepair> marked;

    for (int p=0; p<nstate; ++p)
        marked.insert(statepair(p,FAILSTATE));

    for (int p=-1; p<nstate; ++p) {
        for (int q=p+1; q<nstate; ++q) {
            listof[statepair(p,q)]=emptyset;
        }
    }

    for (int p=0; p<nstate; ++p) {
```

```
        for (int q=p+1; q<nstate; ++q) {
            int foundmarked=0;
            for (set<Word>::const_iterator xp=alphabet.begin();
                 (!foundmarked) && xp!=alphabet.end(); ++xp) {
                statepair destpair=statepair(dest(p,*xp),dest(q,*xp));
                if (marked.find(destpair)!=marked.end()) {
                    foundmarked=1;
                    markpair(statepair(p,q),marked,listof);
                }
            }

            if (!foundmarked) {
                for (set<Word>::const_iterator xp=alphabet.begin();
                     xp!=alphabet.end(); ++xp) {
                    statepair destpair=statepair(dest(p,*xp),dest(q,*xp));
                    if (destpair.x!=destpair.y) {
                        listof[destpair].insert(statepair(p,q));
                    }
                }
            }
        }
    }

    // Set iofi array so iofi[j] gives the new index of state j.      (A15)
    for (int i=0; i<nstate; ++i)
        iofi.push_back(i);
    for (int i=0; i<nstate; ++i) {
        if (iofi[i]==i) { // this index hasn't changed
            for (int j=i+1; j<nstate; ++j) {
                statepair ij=statepair(i,j);
                if (marked.find(ij)==marked.end()) {
                    // i and j are the same
                    iofi[j]=i;
                }
            }
        }
    }
}



/*
** Output Routines
*/
```

```
ostream& operator<< (ostream& s, const ConeTypeFSA& fsa)                (A16)
{
    int nconetype=0;
    for (int k=0; k<fsa.nstate; ++k) {
        Vertex *vp=fsa.vofi[k];
        if (fsa.iofi[k]==k) {
            ++nconetype;
        }
    }
    s << "Geodesic word acceptor for SC(1) ("
      << nconetype
      << " cone types):" << endl;
    for (int k=0; k<fsa.nstate; ++k) {
        if (fsa.iofi[k]==k) {
            Vertex *vp=fsa.vofi[k];
            s << vp << ":";
            for (map<TransDom,TransRan>::const_iterator i=vp->edge.begin();
                 i!=vp->edge.end(); i++) {
                if (i->second.dir==+1) {
                    int j=fsa.iofi[fsa.iofv.find(i->second.destp)->second];
                    s << " " << i->first.x << "->" << fsa.vofi[j];
                }
            }
            s << endl;
        }
    }
    return s;
}


ostream& operator<< (ostream& s, const FaceType& bt)
{
    if (bt==EmptyFaceType)
        return s << "(emptyfacetype)";
    else if (bt.mode==FaceType::BASECOMPLEX)
        return s << "(" << (bt.baseword) << ")";
    else return s
            << "("
            << (bt.b) << ","
            << (bt.f) << ","
            << (bt.sink2/2)
            << ((bt.sink2%2)?".5":"")
            << ")";
}
```

```
void VertexEdgeOut(ostream& s, Vertex& v)
{
    const char *arrow;
    s << "  Vertex " << v << ": attachedfacetype="
      << v.attachedfacetype << ", "
      << v.edge.size() << " edges:" << endl;
    for (map<TransDom,TransRan>::const_iterator
              i=v.edge.begin(); i!=v.edge.end(); i++) {
        if (i->second.dir>0)
            arrow="+1";
        else if (i->second.dir<0)
            arrow="-1";
        else
            arrow="0";
        s << "    ( " << i->first.x << " ,";
        if (i->first.t==0)
            s << setw(fieldwidth) << "t";
        else
            s << setw(fieldwidth) << vertexstr(i->first.t).c_str();
        // I need the c_str() to make setw(10) work properly.
        s << " )  ->  ("
          << setw(fieldwidth) << vertexstr(i->second.destp).c_str()
          << " , ";
        if (i->second.stkop==TransRan::PUSH)
            s << setw(fieldwidth-2)
              << vertexstr(i->second.pushed).c_str()
              << " t";
        else if (i->second.stkop==TransRan::POP)
            s << setw(fieldwidth) << "eps";
        else
            s << setw(fieldwidth) << "t";
        s << " )" << setw(4) << arrow << endl;
    }
}


string vertexstr (const Vertex* vp)
{
    if (vp==0)
        return "-";
    else {
        ostringstream s;
        s << vp->ownerp->facetype << vp->index;
        return s.str();
```

```
    }
}


ostream& operator<< (ostream& s, const Vertex* vp)
{
    if (vp==0)
        return s << "-";
    else if (vp==DummyStackLetter)
        return s << "(stackstart)";
    else
        return s << *vp;
}


ostream& operator<< (ostream& s, const Vertex& v)
{
    return s << v.ownerp->facetype << v.index;
}


ostream& operator<< (ostream& s, const TransDom& td)
{
    s << "(" << td.x << "," << *td.t << ")";
    return s;
}


ostream& operator<< (ostream& s, const TransRan& tr)
{
    s << "("
      << tr.dir << ","
      << tr.destp << ","
      << tr.stkop << ","
      << tr.pushed
      << ",this=" << &tr << ")";
    return s;
}


ostream& operator<< (ostream& s, const Face& C)
{
    s << "FaceType" << C.facetype << ": ";
    if (C.facetype.mode==FaceType::BASECOMPLEX) {
        s << "BASECOMPLEX initial=" << C.initp
```

```
          << ", terminal=" << C.termp
          << "; 1 vertex" << endl;
        VertexEdgeOut(s,*C.vp[0]);
    }
    else {
        s << "FACE  "
          << (C.facetype.bv()-C.facetype.fv()-1) << " vertices"
          << endl;
        if (C.facetype.sink2==0) {
            VertexEdgeOut(s,*C.vp[0]);
        }
        else {
            for (int i=C.facetype.fv()+1; i<=C.facetype.bv()-1; ++i) {
                VertexEdgeOut(s,*C.vp[i]);
            }
        }
    }
    return s;
}


ostream& operator<< (ostream& s, const SchComplex& sc)
{
    s << "SchComplex for u=" << sc.bcp->facetype.baseword
      << " in M=Inv<X|w=" << sc.Mp->w << "=1>" << endl;
    s << "Base complex: " << *sc.bcp << endl;
    return s;
}


ostream& operator<< (ostream& s, const InvMonoid& M)
{
    s << "InvMonoid<X|w=" << M.w << "=1>  "
      << M.ftyptbl.size() << " face types:" << endl;
    for (map<FaceType,Face *>::const_iterator btap=M.ftyptbl.begin();
         btap!=M.ftyptbl.end(); btap++) {
        s << *(btap->second) << endl;
    }
    return s;
}


ostream& operator<< (ostream& s, const InstDesc& id)
{
    s << "ID("
```

```
      << id.scp->bcp->facetype.baseword << ","
      << id.u << ","
      << id.i << "; "
      << *(id.qp) << ", "
      << id.u.right(id.i) << ",";
    for (list<Vertex *>::const_iterator
             i=id.stk.begin(); i!=id.stk.end(); ++i)
        s << " " << *i;
    s << ")";
    return s;
}



/*
** Miscellaneous definitions
*/

const FaceType EmptyFaceType;
const TransRan NullTransRan;
Vertex DummyVertex;
  /* don't make const--want to be able to pass DummyStackLetter to
     places where ptrs to non-const Vertex are allowed. */
Vertex *const DummyStackLetter=&DummyVertex;
```

A.3 Front-end program: `spar.cc`

The following is the front-end program, which takes input from the user and calls the appropriate functions to answer questions about inverse monoids presented by a sparse relator. This program can also be run by giving it commands and operands on the command line, rather than interactively answering questions from the terminal. The code which provides this functionality has been suppressed here, in order to to keep the listing to a reasonable length. The suppressed code does not contribute to the understanding of the mathematics, and was not used by the sample runs of Section 5.1. This code can be viewed by consulting the `spar.cc` source file directly.

```
// spar.cc - Analyze the SC(1) for M=Inv<X|w=1> for w sparse.

#include "word.h"
#include "sc.h"

/*
** Interactive mode
*/

string wordhelp=
"Input a word, such as abABcdCD.  Use upper-/lowercase letters for the
inverses of their lower-/uppercase counterparts (a^-1=A, A^-1=a, etc.).
You may enter 1, eps, epsilon, or nothing to specify the empty word.";

string stephelp=
"Enter an integer number of steps to perform (letters to read) before
I stop and ask you again.  The process automatically stops when the
entire word has been read or can be read no further, so you may enter
a large number such as 999 to continue without stopping.  You may
also simply press ENTER to do just one more step.";

string menuhelp=
"Choose from among the following options:
  1) List the PDA (face types and transitions) for your inverse monoid
  2) Interactively test whether or not u=1; i.e., whether SC(1) accepts u
  3) Test whether u,v are R-related to 1, and if so, whether u=v
  4) Compute the cone type automaton which recognizes geodesics in SC(1)
  8) Enter a new relator w (your current monoid is discarded)
  9) Leave this program (quit and Ctrl-D also work)";

string ask(const string& prompt, const string& help)
{
    string input;
    bool notdone=1;
    while (notdone) {
        cout << prompt;
        if (!getline(cin,input) || strcasecmp(input.c_str(),"quit")==0) {
            cout << "Good-bye!" << endl;
            exit(0);
        }
        if (input=="?" || strcasecmp(input.c_str(),"help")==0) {
            cout << help << endl;
        }
        else
            notdone=0;
```

```
    }
    return input;
}


void interactive_u_equals_v(InvMonoid& M)
{
    cout<< "\n"
        << "Enter words u and v.  This will determine whether or not\n"
        << "u and v are R-related to 1, and if so, whether or not u=v:"
        << endl;
    Word u=ask("u=",wordhelp);
    Word v=ask("v=",wordhelp);
    InstDesc idu=InstDesc(M.sc1p,u).advancedto();
    InstDesc idv=InstDesc(M.sc1p,v).advancedto();
    bool uR1=(idu.i==idu.u.len());
    bool vR1=(idv.i==idv.u.len());
    cout<< "Your word u " << (uR1?"IS":"is NOT")
        << " R-related to 1." << endl;
    cout<< "Your word v " << (vR1?"IS":"is NOT")
        << " R-related to 1." << endl;
    if (uR1 && vR1) {
        if (idu.qp==idv.qp && idu.stk==idv.stk)
            cout<< "The words u and v are EQUAL in M." << endl;
        else
            cout<< "The words u and v are EQUAL in M." << endl;
    }
    cout<< endl;
}


void interactive_step_id_display(InstDesc id)
{
    cout<< "Step " << id.i << ":"
        << " State=" << id.qp
        << "  Remaining=" << id.u.right(id.i)
        << "  Stack:";
    for (list<Vertex *>::const_iterator
             i=id.stk.begin(); i!=id.stk.end(); ++i)
        cout << " " << *i;
    cout<< endl;
}

void interactive_accepts_u_step(InvMonoid& M)
{
```

```
    cout<< "\n"
        << "Enter the word u which you wish to try to read in SC(1).\n"
        << "You will see the attempt to read u in SC(1) step-by-step."
        << endl;
    Word u=ask("u=",wordhelp);
    cout<< endl;

    InstDesc id=InstDesc(M.sc1p,u);
    bool notdone=1;
    interactive_step_id_display(id);

    while (notdone) {
        string nstep=ask("Next (or num of steps): ",stephelp);
        int n;
        if (nstep=="")
            n=1;
        else
            n=atoi(nstep.c_str());
        while (n-- && notdone) {
            InstDesc newid=id.advancedto(0,0,1);                    (A17)
            notdone=(id.i<newid.i) && (newid.i<newid.u.len());
            id=newid;
            interactive_step_id_display(id);
        }
    }

    cout<< endl;
    if (id.i<id.u.len()) {
        cout<< "The first " << id.i
            << " letters of u can be read in SC(1),\n"
            << "but not all of u.  Therefore, u is not R-related to 1."
            << endl;
    }
    else if (id.qp!=id.scp->bcp->termp) {
        cout<< "All of u can be read inside SC(1), but the final state\n"
            << "attained is not the accept state, "
            << id.scp->bcp->termp << ".\n"
            << "Therefore, u is NOT equal to 1.  However, this does mean\n"
            << "u is equivalent to 1 under Green's R-relation."
            << endl;
    }
    else {
        cout << "SC(1) accepts u. Therefore, u = 1." << endl;
    }
    cout<< endl;
}
```

```
void interactive_list_M(InvMonoid& M)
{
    cout << endl << "Your monoid is M=" << M;
}


void interactive_conetypefsa(InvMonoid& M)
{
    cout<< "\n"
        << "The following is the cone type automaton.  It recognizes\n"
        << "the language of geodesic words in the Schutzenberger\n"
        << "graph of 1:\n"
        << endl;
    cout << ConeTypeFSA(M.sc1p) << endl;
}


void interactive_mode(void)
{
    cout<<"
Welcome to 'spar'.  This program computes the PDA associated with the
Schutzenberger complex of 1 for inverse monoids of the form
M=Inv<X|w=1>, where w is a sparse word, and allows you to test whether
words are accepted by this PDA.

(Note: When this program asks for input, you may respond with ? or
'help' to get more information on what kind of response is required.
Also, you may run this program by giving operands and commands on
the command line and in files; type 'spar help' at the command line
for more information.)
";

    bool notdone=1;
    while (notdone) {
        Word w=ask("\nEnter your relator w: ",wordhelp);

        if (w.len()<2) {
            cout<< "Your relator, " << w << ", has length "
                << w.len() << ".\n"
                << "Please enter a sparse relator of length "
                << "at least 2.\n"
                << "If you wish to stop this program, enter  quit ."
                << endl;
```

```
        continue;
    }

    if (!w.iscycred()) {
        cout<< "Your relator, " << w
            << ", is not cyclically reduced.\n"
            << "Please enter a cyclically reduced and sparse "
            << "relator." << endl;
        continue;
    }

    pointedpiece olx,oly;
    if (!w.check_sparse_with_violators(olx,oly)) {
        if (olx==oly) {
            cout<< "Your relator, " << w
                << ", is not sparse because the pointed piece\n"
                << olx << " overlaps itself." << endl;
        }
        else {
            cout<< "Your relator, " << w
                << ", is not sparse because the pointed pieces\n"
                << olx << " and " << oly << " overlap." << endl;
        }
        cout<< "Please enter a sparse relator." << endl;
        continue;
    }

    InvMonoid M=InvMonoid(w);
    cout<< "\nI have computed the Schutzenberger complex of 1 for\n"
        << "M=Inv<X|w=1>, w=" << w << "." << endl;
    cout<< endl << menuhelp << endl;
    string menuprompt=
        "What next (1:list,2:u=1,3:u=v,4:geo,8:newrel,9:quit)? ";
    string option;
    while ((option=ask(menuprompt,menuhelp))!="9" && option!="8") {
             if (option=="1") interactive_list_M(M);
        else if (option=="2") interactive_accepts_u_step(M);
        else if (option=="3") interactive_u_equals_v(M);
        else if (option=="4") interactive_conetypefsa(M);
        else cout << endl << menuhelp << endl;
    }
    notdone=(option=="8");
}

cout << "Good-bye!" << endl;
```

```
    exit(0);
}


int main (int argc, char **argv)
{
    if (argc==1)
        interactive_mode();
}
```

Bibliography

[BMM94] Jean-Camille Birget, Stuart W. Margolis, and John C. Meakin, *The word problem for inverse monoids presented by one idempotent relator*, Theoret. Comput. Sci. **123** (1994), no. 2, 273–289. MR 94j:20057

[ECH+92] David B. A. Epstein, James W. Cannon, Derek F. Holt, Silvio V. F. Levy, Michael S. Paterson, and William P. Thurston, *Word processing in groups*, Jones and Bartlett Publishers, Boston, MA, 1992. MR 93i:20036

[HU79] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to automata theory, languages, and computation*, Addison-Wesley Publishing Co., Reading, Mass., 1979, Addison-Wesley Series in Computer Science. MR 83j:68002

[IKT02] Isamu Inata, Naoki Kataumi, and Takatoshi Tobita, *The word problem for the braid inverse monoid*, Sūrikaisekikenkyūsho Kōkyūroku (2002), no. 1268, 105–109, Algorithms in algebraic systems and computation theory (Japanese) (Kyoto, 2002). MR 1 955 839

[IMM01] S. V. Ivanov, S. W. Margolis, and J. C. Meakin, *On one-relator inverse monoids and one-relator groups*, J. Pure Appl. Algebra **159** (2001), no. 1, 83–111. MR 2001m:20107

[Law98] Mark V. Lawson, *Inverse semigroups*, World Scientific Publishing Co. Inc., River Edge, NJ, 1998, The theory of partial symmetries. MR 2000g:20123

[Lin] Steven P. Lindblad, *Spar software program*, Available at http://www.-math.unl.edu/˜slindbla/sparse/.

[Mea93] John C. Meakin, *An invitation to inverse semigroup theory*, Proceedings of the conference on ordered structures and algebra of computer languages, Hong Kong, 26-29 June 1991, World Scientific, Hong Kong, 1993, pp. 91–115.

[MM93] Stuart W. Margolis and John C. Meakin, *Inverse monoids, trees and context-free languages*, Trans. Amer. Math. Soc. **335** (1993), no. 1, 259–276. MR 93h:20062

[Mun74] W. D. Munn, *Free inverse semigroups*, Proc. London Math. Soc. (3) **29** (1974), 385–404. MR 50 #13328

[Pet84] Mario Petrich, *Inverse semigroups*, Pure and Applied Mathematics, John Wiley & Sons Inc., New York, 1984, A Wiley-Interscience Publication. MR 85k:20001

[Sil92] Pedro V. Silva, *Rational languages and inverse monoid presentations*, Internat. J. Algebra Comput. **2** (1992), no. 2, 187–207. MR 93e:20074

[Sil95] ———, *The word problem for nilpotent inverse monoids*, Semigroup Forum **51** (1995), no. 3, 285–293. MR 97b:20091

[Ste87] J. B. Stephen, *Applications of automata theory to presentations of monoids and inverse monoids*, Ph.D. thesis, University of Nebraska-Lincoln, 1987.

[Ste90] ———, *Presentations of inverse monoids*, J. Pure Appl. Algebra **63** (1990), no. 1, 81–112. MR 91g:20083

[Ste93] ———, *Inverse monoids and rational subsets of related groups*, Semigroup Forum **46** (1993), no. 1, 98–108. MR 93k:20095

[Ste00] Benjamin Steinberg, *Fundamental groups, inverse Schützenberger automata, and monoid presentations*, Comm. Algebra **28** (2000), no. 11, 5235–5253. MR 2001e:20060

[Ste03] ———, *A topological approach to inverse and regular semigroups*, Pacific J. Math. **208** (2003), no. 2, 367–396. MR 1 971 670

[Str00] Bjarne Stroustrup, *The C++ programming language*, Addison-Wesley, Reading, Massachusetts, 2000, Special edition.

[Yam01] Akihiro Yamamura, *Presentations of Bruck-Reilly extensions and decision problems*, Semigroup Forum **62** (2001), no. 1, 79–97. MR 2002h:20094