Chapter 4

The Fibers and Range of Reduction Graphs

Abstract
The biological process of gene assembly transforms a nucleus (the MIC) into a functionally and physically different nucleus (the MAC). For each gene in the MIC (the input), recombination operations transform the gene to its MAC form (the output). Here we characterize which inputs obtain the same output, and moreover characterize the possible forms of the outputs. We do this in the abstract and more general setting of so-called legal strings.

4.1 Introduction

Ciliates form a large group of one-cellular organisms that are able to transform one nucleus, called the micronucleus (MIC), into an astonishing different one, called the macronucleus (MAC). This intricate DNA transformation process is called *gene assembly*. Each gene occurs both in the MIC and MAC, but in very different forms. During gene assembly each gene is transformed from its MIC form to its MAC form.

Formally, the gene in MIC form (the input) can be described by a so-called legal string [12], while the gene in MAC form, including additionally generated structures, (the output) can be described by a so-called reduction graph [6, 5]. The reduction graph is based on the notion of breakpoint graph in the theory of sorting by reversal [17, 1, 23].

Given the function $\mathcal{R}$ that assigns to each legal string u its reduction graph $\mathcal{R}_u$, we (1) characterize the range of $\mathcal{R}$ (up to graph isomorphism) in terms of easy-to-check conditions on graphs (cf. Theorem 24), and (2) characterize the fiber $\mathcal{R}^{-1}(\mathcal{R}_u)$ (modulo graph isomorphism) for each reduction graph $\mathcal{R}_u$ (cf. Theorem 34). In fact we show that $\mathcal{R}^{-1}(\mathcal{R}_u)$ is the 'orbit' of u under two types of string rewriting rules.

Result (1) characterizes which graphs are (isomorphic to) reduction graphs. Obviously, these graphs should have the 'look and feel' of reduction graphs. For instance, each vertex label should occur exactly four times, and the second type of edges connect vertices of the same label. Once these elementary and easy-to-check properties are satisfied, reduction graphs are characterized as having a connected pointer-component graph — a graph which represents the distribution of the vertex labels over the connected components, originally defined in [4]. This last condition can also be efficiently verified. The characterization implies restrictions on the form of the MAC structures that can possibly occur.

Result (2) determines, given two legal strings, whether or not they have the same reduction graph. This may allow one to determine which MIC genes obtain the same MAC structure. It turns out that two legal strings obtain the same reduction graph (up to isomorphism) exactly when they can be transformed into each other by two types of string rewriting rules. We will see that, surprisingly, these rules are in a sense dual to string rewriting rules in a model of gene assembly called string pointer reduction system (SPRS) [12].

The latter characterization has additional uses for the specific model SPRS as well. In this model, gene assembly is assumed to be performed by three types of recombination (splicing) operations that are modeled as types of string rewriting rules. The string negative rules form one of these types. It has been shown that the reduction graph allows for a complete characterization of applicability of the string negative rules during the transformation process [6, 4]. Moreover, it has been shown that the reduction graph does not retain much information about the applicability of the other two types of rules [4]. Therefore, the legal strings that obtain the same reduction graph are exactly the legal strings that have similar characteristics concerning the string negative rule.

To establish both main results, we augment the (abstract) reduction graph with a set of *merge-legal edges*. We will show that some "valid" sets of merge-legal edges for a reduction graph allows one to "go back" to a legal string corresponding to this (abstract) reduction graph. In this way the existence of such valid set determines which graphs are (isomorphic to) reduction graphs. The first main result shows that the existence of such valid set is computationally easy to verify. Moreover, the set of all sets of merge-legal edges can be transformed into each other by *flip operations*. These flip operations can be defined in terms of the above mentioned dual string pointer rules on legal strings. This will establish the other main result.

This chapter is organized as follows. Section 4.2 fixes notation of basic mathematical notions. In Section 4.3 we recall notions related to legal strings, in Section 4.4 we recall the reduction graph and the pointer-component graph, and in Section 4.5 we generalize the notion of reduction graph and give an extension through merge-legal edges. In Section 4.6 we provide a preliminary characterization that determines which graphs are (isomorphic to) reduction graphs. In the next three sections, we strengthen the result to allow for efficient algorithms: in

Section 4.7 we define the flip operation on sets of merge-legal edges, in Section 4.8 we show that the effect of flip operation corresponds to merging or splitting of connected components, and in Section 4.9 we prove the first main result, cf. Theorem 24. In Sections 4.10 and 4.11 we prove the second main result, cf. Theorem 34. We conclude this chapter with a discussion. A conference edition of this chapter, containing selected results without proofs, was presented at DLT '07 [2].

4.2 Mathematical Notation and Terminology

In this section we recall some basic notions concerning functions, strings, and graphs. We do this mainly to fix the basic notation and terminology.

The symmetric difference of sets X and Y, $(X \backslash Y) \cup (Y \backslash X)$, is denoted by $X \oplus Y$. As $\oplus$ is associative, one may define the symmetric difference of a finite family of sets $(X_i)_{i \in A}$ – it is denoted by $\bigoplus_{i \in A} X_i$. The *composition* of functions $f : X \to Y$ and $g : Y \to Z$ is the function $gf : X \to Z$ such that $(gf)(x) = g(f(x))$ for every $x \in X$. The restriction of f to a subset A of X is denoted by $f|A$. The range $f(X)$ of f will be denoted by $\mathrm{rng}(f)$. The *fiber* (or *preimage*) of $y \in Y$ under f, denoted by $f^{-1}(y)$, is $\{x \in X \mid f(x) = y\}$. The fibers form a partition of X. If $Y = X$, then f is called *self-inverse* if f^2 is the identity function. We will use λ to denote the empty string.

We now turn to graphs. A *(undirected) graph* is a tuple $G = (V, E)$, where V is a finite set and $E \subseteq \{\{x, y\} \mid x, y \in V\}$. The elements of V are the *vertices* of G and the elements of E are the *edges* of G. In this chapter we allow $x = y$, and therefore edges can be of the form $\{x, x\} = \{x\}$ — an edge of this form should be seen as an edge connecting x to x, i.e., a 'loop' for x. The *restriction of G to $E' \subseteq E$*, denoted by $G|_{E'}$, is the subgraph (V, E'). The *order* $|V|$ of G is denoted by $o(G)$.

A *multigraph* is a (undirected) graph $G = (V, E, \epsilon)$, where parallel edges are possible. Therefore, E is a finite set of edges and $\epsilon : E \to \{\{x, y\} \mid x, y \in V\}$ is the *endpoint mapping*. Note that for multigraphs, E is not specified in terms of V – the relationship between V and E is specified by ϵ.

A *coloured base* B is a 4-tuple (V, f, s, t) such that V is a finite set, $s, t \in V$, and $f : V \backslash \{s, t\} \to \Gamma$ for some Γ. The elements of V, $\{\{x, y\} \mid x, y \in V, x \neq y\}$, and Γ are called *vertices*, *edges*, and *vertex labels* for B, respectively.

An *n-edge coloured graph*, $n \geq 1$, is a tuple $G = (V, E_1, E_2, \cdots, E_n, f, s, t)$ where $B = (V, f, s, t)$ is a coloured base and, for $i \in \{1, \ldots, n\}$, E_i is a set of edges for B. We also denote G by $B(E_1, E_2, \cdots, E_n)$. We define $\mathrm{dom}(G) = \mathrm{rng}(f)$.

The previously defined notions and notation for graphs carry over to multigraphs and n-edge coloured graphs. Isomorphisms between graphs are defined in the usual way: graphs are considered *isomorphic* when they are equal modulo the identity of the vertices. However, the labels of the identified vertices in n-edge coloured graphs must be equal. Therefore n-edge coloured graphs $G = (V, E_1, .., E_n, f, s, t)$ and $G' = (V', E'_1, ..., E'_n, f', s', t')$ are *isomorphic*, denoted by $G \approx G'$, if there is a bijection $q : V \to V'$ such that $q(s) = s'$, $q(t) = t'$,

$f(v) = f'(q(v))$ for all $v \in V$, and $\{x, y\} \in E_i$ iff $\{q(x), q(y)\} \in E'_i$, for all $x, y \in V$, and $i \in \{1, \dots, n\}$. Also, multigraphs $G = (V, E, \epsilon)$ and $G' = (V', E, \epsilon')$ are *isomorphic*, denoted by $G \approx G'$, if there is a bijection $\alpha : V \to V'$ such that $\alpha\epsilon = \epsilon'$, or more precisely, for $e \in E$, $\epsilon(e) = \{v_1, v_2\}$ implies $\epsilon'(e) = \{\alpha(v_1), \alpha(v_2)\}$. We assume the reader is familiar with the notions of *cycle* and *connected component* in a graph. A graph is called *connected* if it has exactly one connected component, and it is called *acyclic* when it does not contain cycles.

4.3 Legal strings

Gene assembly transforms each gene from its MIC form to its MAC form. Formally, the MIC form of a gene (the input) is represented by a legal string u, while the MAC form of that gene, including the additionally generated structures, (the output) is represented by the reduction graph of u. We define the notion of legal string and some accompanying notions in this section, and the notion of reduction graph in the next section. We refer to [12] for a detailed motivation of the notions of this section.

We fix $\kappa \geq 2$, and define the alphabet $\Delta = \{2, 3, \dots, \kappa\}$. For $D \subseteq \Delta$, we define $\bar{D} = \{\bar{a} \mid a \in D\}$ and $\Pi = \Delta \cup \bar{\Delta}$. The elements of Π will be called *pointers*. We use the 'bar operator' to move from Δ to $\bar{\Delta}$ and back from $\bar{\Delta}$ to Δ. Hence, for $p \in \Pi$, $\bar{\bar{p}} = p$. For a string $u = x_1 x_2 \cdots x_n$ with $x_i \in \Pi$, the *inverse* of u is the string $\bar{u} = \bar{x}_n \bar{x}_{n-1} \cdots \bar{x}_1$. For $p \in \Pi$, we define $\mathbf{p} = \begin{cases} p & \text{if } p \in \Delta \\ \bar{p} & \text{if } p \in \bar{\Delta} \end{cases}$, i.e., $\mathbf{p}$ is the 'unbarred' variant of p. The *domain* of a string $v \in \Pi^*$ is $\mathrm{dom}(v) = \{\mathbf{p} \mid p \text{ occurs in } v\}$. A *legal string* is a string $u \in \Pi^*$ such that for each $p \in \Pi$ that occurs in u, u contains exactly two occurrences from $\{p, \bar{p}\}$. For a pointer p and a legal string u, if both p and $\bar{p}$ occur in u then we say that both p and $\bar{p}$ are *positive* in u; if on the other hand only p or only $\bar{p}$ occurs in u, then both p and $\bar{p}$ are *negative* in u.

Let $u = x_1 x_2 \cdots x_n$ be a legal string with $x_i \in \Pi$ for $1 \leq i \leq n$. For a pointer $p \in \Pi$ such that $\{x_i, x_j\} \subseteq \{p, \bar{p}\}$ and $1 \leq i < j \leq n$, the *p-interval of u* is the substring $x_i x_{i+1} \cdots x_j$. Two distinct pointers $p, q \in \Pi$ *overlap* in u if both $\mathbf{q} \in \mathrm{dom}(I_p)$ and $\mathbf{p} \in \mathrm{dom}(I_q)$, where I_p (I_q, resp.) is the p-interval (q-interval, resp.) of u.

We say that legal strings u and v are *equivalent*, denoted by $u \approx v$, if there is homomorphism $\varphi : \Pi^* \to \Pi^*$ with $\varphi(p) \in \{p, \bar{p}\}$ and $\varphi(\bar{p}) = \overline{\varphi(p)}$ for all $p \in \Pi$ such that $\varphi(u) = v$.

Example 1
Legal strings $2\bar{2}33$ and $\bar{2}233$ are equivalent, while $2\bar{2}33$ are $2\bar{2}\bar{3}3$ are not.

Note that $\approx$ is an equivalence relation. Equivalent legal strings are characterized by their 'unbarred version' and their set of positive pointers.

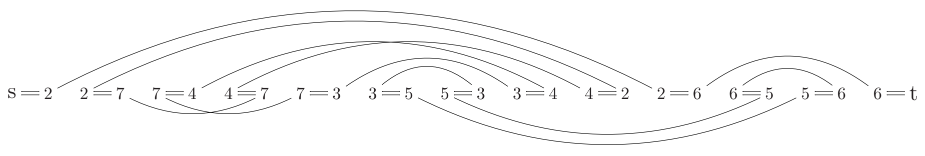


Figure 4.1: The reduction graph $\mathcal{R}_u$ of u in Example 2.

4.4 Reduction Graph

We now recall the definition of reduction graph. This definition is equal to the one in [4], and is in slightly less general form compared to the one in [6]. We refer to [6], where it was introduced, for a more detailed motivation and for more examples and results. The notion of reduction graph uses the intuition from the notion of breakpoint graph (or reality-and-desire diagram) known from another branch of DNA processing theory called sorting by reversal, see e.g. [23, 21]. From a biological point of view, the reduction graph represents the MAC form of a gene (including the additionally generated structures) given its MIC form. As the MIC form of a gene is represented by a legal string, reduction graphs are defined on legal strings.

Definition 1
Let $u = p_1 p_2 \cdots p_n$ with $p_1, \ldots, p_n \in \Pi$ be a legal string. The *reduction graph of* u, denoted by $\mathcal{R}_u$, is a 2-edge coloured graph (V, E_1, E_2, f, s, t), where

$$V = \{I_1, I_2, \ldots, I_n\} \cup \{I'_1, I'_2, \ldots, I'_n\} \cup \{s, t\},$$

$$E_1 = \{e_0, e_1, \ldots, e_n\} \text{ with } e_i = \{I'_i, I_{i+1}\} \text{ for } 0 < i < n, e_0 = \{s, I_1\}, e_n = \{I'_n, t\},$$

$$\begin{aligned} E_2 = \; & \{\{I'_i, I_j\}, \{I_i, I'_j\} \mid i, j \in \{1, 2, \ldots, n\} \text{ with } i \neq j \text{ and } p_i = p_j\} \;\cup \\ & \{\{I_i, I_j\}, \{I'_i, I'_j\} \mid i, j \in \{1, 2, \ldots, n\} \text{ and } p_i = \bar{p}_j\}, \text{ and} \end{aligned}$$

$$f(I_i) = f(I'_i) = \mathbf{p}_i \text{ for } 1 \leq i \leq n.$$

■

The edges of E_1 are called the *reality edges*, and the edges of E_2 are called the *desire edges*. Notice that for each $p \in \mathrm{dom}(u)$, the reduction graph of u has exactly two desire edges containing vertices labelled by p. It follows from the construction of the reduction graph that, given legal strings u and v, $u \approx v$ iff $\mathcal{R}_u = \mathcal{R}_v$.

In depictions of reduction graphs, we will represent the vertices (except for s and t) by their labels, because the exact identity of the vertices is not essential for the problems considered in this chapter. We will also depict reality edges as 'double edges' to distinguish them from the desire edges.

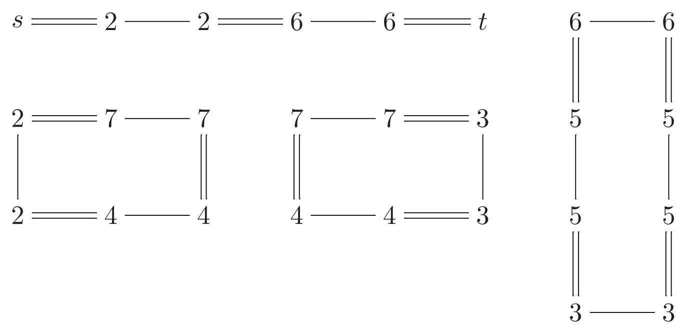


Figure 4.2: The reduction graph of Figure 4.1 obtained by rearranging the vertices.

Example 2
The reduction graph of $u = 2\bar{7}47353\bar{4}2656$ is depicted in Figure 4.1. Note how positive pointers are connected by crossing desire edges, while those for negative pointers are parallel. By rearranging the vertices we can depict the graph as shown in Figure 4.2.

Reality edges follow the linear order of the legal string, whereas desire edges connect positions in the string that will be joined when performing reduction rules, see [6].

We now recall the definition of pointer-component graph of a legal string, introduced in [4]. The graph represents how the labels of a reduction graph are distributed among its connected components. Surprisingly, this graph has different uses in this chapter compared to its original uses in [4]. There it was used in a specific model of gene assembly (which we do not assume here) to characterize a type of splicing operation called loop recombination.

Definition 2
Let u be a legal string. The *pointer-component graph of u (or of $\mathcal{R}_u$)*, denoted by $\mathcal{PC}_u$, is a multigraph (ζ, E, ϵ), where ζ is the set of connected components of $\mathcal{R}_u$, $E = \mathrm{dom}(u)$ and ϵ is, for $e \in E$, defined by $\epsilon(e) = \{C \in \zeta \mid C$ contains vertices labelled by $e\}$. ∎

Since for each $e \in \mathrm{dom}(u)$, there are exactly two desire edges connecting vertices labelled by e, $1 \leq |\epsilon(e)| \leq 2$, and therefore ϵ is well defined (recall that the case $|\epsilon(e)| = 1$ corresponds to a loop).

Example 3
The pointer-component graph of the reduction graph from Figure 4.2 is shown in Figure 4.3.

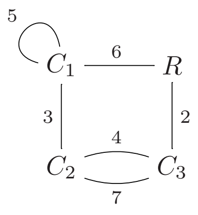


Figure 4.3: The pointer-component graph of the reduction graph from Figure 4.2.

4.5 Abstract Reduction Graphs and Extensions

In this section we generalize the notion of reduction graph as a starting point to consider which graphs are (isomorphic to) reduction graphs. Moreover, we extend the reduction graphs by a set of edges, called *merge edges*, such that, along with the reality edges, the linear structure of the legal string is preserved in the graph.

We will now define a set of edges for a given coloured base which has features in common with desire edges of a reduction graph.

Definition 3
Let $B = (V, f, s, t)$ be a coloured base. We say that a set of edges E for B is *desirable* if

1. for all $\{v_1, v_2\} \in E$, $f(v_1) = f(v_2)$,
2. for each $v \in V \backslash \{s, t\}$ there is exactly one $e \in E$ such that $v \in e$. ■

We now generalize the concept of reduction graph.

Definition 4
A 2-edge coloured graph $B(E_1, E_2)$ with $B = (V, f, s, t)$ is called an *abstract reduction graph* if

1. $\mathrm{rng}(f) \subseteq \Delta$, and for each $p \in \mathrm{rng}(f)$, $|f^{-1}(p)| = 4$,
2. for each $v \in V$ there is exactly one $e \in E_1$ such that $v \in e$,
3. E_2 is desirable for B. ■

The set of all abstract reduction graphs is denoted by $\mathcal{G}$.

Clearly, if $G \approx \mathcal{R}_u$ for some u, then $G \in \mathcal{G}$. Therefore, for abstract reduction graphs $G = B(E_1, E_2)$, the edges in E_1 are called *reality edges* and the edges in E_2 are called *desire edges*. For graphical depictions of abstract reduction graphs we will use the same conventions as we have for reduction graphs. Thus, edges in E_1 will be depicted as “double edges”, vertices are represented by their label, etc.

Example 4
The 2-edge coloured graph in Figure 4.4 is an abstract reduction graph.

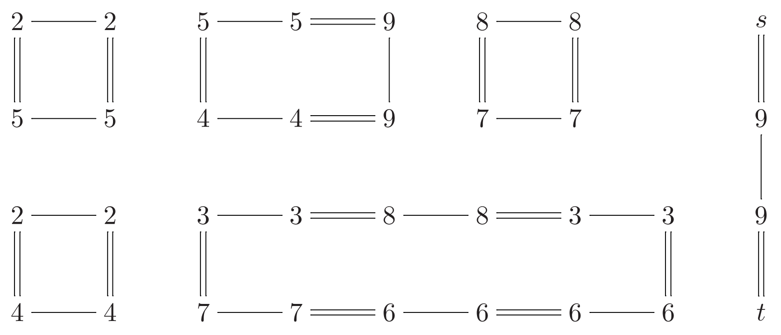


Figure 4.4: An abstract reduction graph.

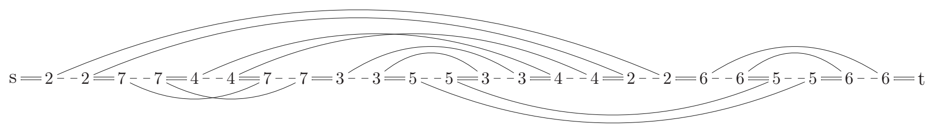


Figure 4.5: The extended reduction graph $\mathcal{E}_u$ of u given in Example 2.

Note that conditions (1) and (3) in the previous definition imply that for each $p \in \mathrm{rng}(f)$, there is a partition $\{e_1, e_2\}$ of $f^{-1}(p)$, denoted by $C_{G,p}$ or C_p when G is clear from the context, such that $e_1, e_2 \in E_2$.

We now introduce an extension to reduction graphs such that the 'generic' linear order of the vertices $s, I_1, I'_1, \ldots, I_n, I'_n, t$ is retained, even when we consider the graphs up to isomorphism.

Definition 5
Let u be a legal string. The *extended reduction graph of* u, denoted by $\mathcal{E}_u$, is a 3-edge coloured graph $B(E_1, E_2, E_3)$, where $\mathcal{R}_u = B(E_1, E_2)$ and $E_3 = \{\{I_i, I'_i\} \mid 1 \leq i \leq n\}$ with $n = |u|$. ■

The edges in E_3 are called the *merge edges of* u, denoted by M_u. In this way, the reality edges and the merge edges form a unique path which passes through the vertices in the generic linear order. This is illustrated in the next example. In figures merge edges will be depicted by "dashed edges".

Example 5
The extended reduction graph $\mathcal{E}_u$ of u given in Example 2 is shown in Figure 4.5, cf. Figure 4.1.

Remark
The notion of merge edges for (extended) reduction graphs is more closely related to the notion of reality edges for breakpoint graphs in the theory of sorting by reversal [17] compared to the notion of reality edges for (extended) reduction graphs. Thus in a way it would be more natural to call the merge edges reality

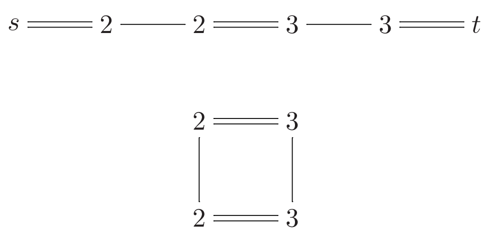


Figure 4.6: An abstract reduction graph.

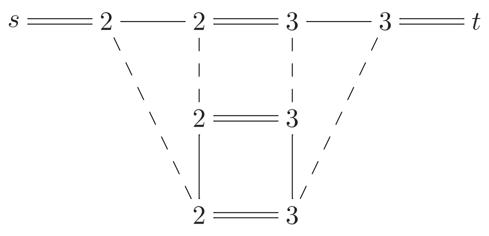


Figure 4.7: The abstract reduction graph of Figure 4.6 with a set of merge-legal edges.

edges for (extended) reduction graphs, and the other way around. However, to avoid confusion with earlier work, we do not change this terminology.

We now generalize this extension of reduction graphs to abstract reduction graphs.

Definition 6
Let $G = B(E_1, E_2) \in \mathcal{G}$, and let E be a set of edges for B. We say that E is *merge-legal* for G if E is desirable for B, and $E_2 \cap E = \varnothing$. We denote the set $\{E \mid E \text{ merge-legal for } G\}$ by ML_G. The set of all $E \in \mathrm{ML}_G$ where $B(E_1, E)$ is connected is denoted by CON_G. ■

For legal string u, we also denote $\mathrm{ML}_{\mathcal{R}_u}$ and $\mathrm{CON}_{\mathcal{R}_u}$ by ML_u and CON_u, respectively. Notice that $M_u \in \mathrm{CON}_u \subseteq \mathrm{ML}_u$. Therefore, merge-legal edges will also be depicted by "dashed edges".

Example 6
Let us consider the abstract reduction graph $G = B(E_1, E_2)$ of Figure 4.6. This graph is again depicted in Figure 4.7 including a merge-legal set E for G. In this way Figure 4.7 depicts the 3-edge coloured graph $B(E_1, E_2, E)$. Notice that $E \notin \mathrm{CON}_G$. In Figure 4.8, the abstract reduction graph is depicted with a merge-legal set in CON_G.

We now define a natural abstraction of the notion of extended reduction graph.

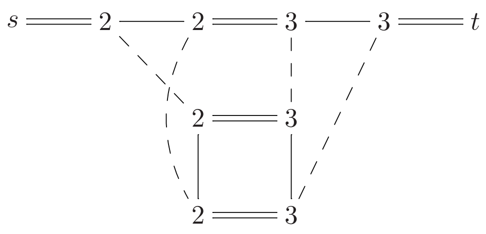


Figure 4.8: The abstract reduction graph of Figure 4.6 with another set of merge-legal edges.

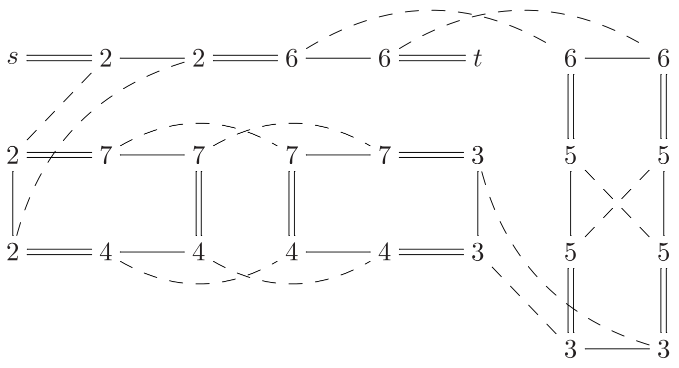


Figure 4.9: A extended abstract reduction graph obtained by augmenting the reduction graph of Figure 4.2 with merge edges.

Definition 7
Let $G = B(E_1, E_2) \in \mathcal{G}$ and $E \in \mathrm{CON}_G$. Then $G' = B(E_1, E_2, E)$ is called a *extended abstract reduction graph*. ∎

For each legal string u, $\mathcal{E}_u$ is an extended abstract reduction graph, since $M_u \in \mathrm{CON}_u$. Therefore, the edges in E (in the previous definition) are called the *merge edges (of G')*. Since $E \in \mathrm{CON}_G$, $B(E_1, E)$ has the following form:

$$s =\!=\!=\!= \mathbf{p}_1 - - - \mathbf{p}_1 =\!=\!=\!= \mathbf{p}_2 - - - \mathbf{p}_2 =\!=\!=\!= \cdots =\!=\!=\!= \mathbf{p}_n - - - \mathbf{p}_n =\!=\!=\!= t$$

Thus the property that reality and merge edges in an extended reduction graph induce a unique path from s to t that alternatingly passes through reality edges and merge edges is retained for extended abstract reduction graphs G in general.

Example 7
If we consider the reduction graph $\mathcal{R}_u = B(E_1, E_2)$ of Example 2 shown in Figure 4.2, then, of course, $B(E_1, E_2, M_u) = \mathcal{E}_u$ shown in Figure 4.5 is a extended abstract reduction graph. In Figure 4.9 another extended reduction graph is shown – it is $\mathcal{R}_u$ augmented with a set of merge edges E in CON_u. It is easy to see that indeed $E \in \mathrm{CON}_u$: simply notice that the path from s to t induced by the reality and merge edges will go through every vertex of the graph.

4.6 Back to Legal Strings

In this section we show that for extended abstract reduction graphs G we can 'go back' in the sense that there are legal strings u such that G is isomorphic to $\mathcal{E}_u$. Moreover we show how to obtain the set L_G of all legal strings that corresponds to G. We will show that the legal strings in L_G are equivalent, and thus that extended reduction graphs retain all essential information of the legal strings.

As extended abstract reduction graphs have a natural linear order of the vertices given by their reality edges and merge edges, we can infer whether or not desire edges 'cross' or not – thereby providing a way to define negative and positive pointers for extended abstract reduction graphs. This is formalized as follows.

Definition 8
Let $G = B(E_1, E_2, E_3)$ be an extended abstract reduction graph, let $G' = B(E_1, E_2)$, and let $\pi = (s, v_1, v'_1, \cdots, v_n, v'_n, t)$ be the path from s to t in $B(E_1, E_3)$. We say that $p \in \mathrm{dom}(G)$ is *negative in* G iff $C_{G',p} = \{\{v_i, v'_j\}, \{v'_i, v_j\}\}$ for some $i, j \in \{1, \ldots, n\}$ with $i \neq j$. Also, we say that $p \in \mathrm{dom}(G)$ is *positive in* G if p is not negative in G. ■

Clearly, $p \in \mathrm{dom}(G)$ is positive in G iff $C_{G',p} = \{\{v_i, v_j\}, \{v'_i, v'_j\}\}$ for some $i, j \in \{1, \ldots, n\}$ with $i \neq j$. It is easy to see that p is negative in legal string u iff p is negative in $\mathcal{E}_u$.

Next, we assign to each extended abstract reduction graph G a set of legal strings L_G. We subsequently show that these strings are precisely the legal strings u such that $\mathcal{E}_u \approx G$.

Definition 9
Let $G = B(E_1, E_2, E_3)$ be an extended abstract reduction graph, and let $H = B(E_1, E_3)$ be as follows:

$$s = \mathbf{p}_1 - - - \mathbf{p}_1 = \mathbf{p}_2 - - - \mathbf{p}_2 = \cdots = \mathbf{p}_n - - - \mathbf{p}_n = t$$

The *legalization* of G, denoted by L_G, is the set of legal strings $u = p_1 p_2 \cdots p_n$ with $p_i \in \{\mathbf{p}_i, \overline{\mathbf{p}}_i\}$ and p_i is negative in u iff p_i is negative in G. ■

Example 8
Let us consider the extended abstract reduction graph G of Figure 4.9. By rearranging the vertices we obtain Figure 4.10. From this figure it is clear that $v = 27426\bar{5}374356 \in L_G$.

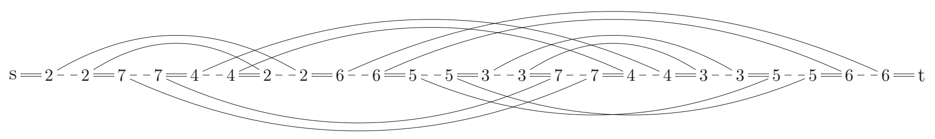


Figure 4.10: The extended abstract reduction graph G given in Example 8.

It is easy to see that, for a legal string u, we have $u \in L_{\mathcal{E}_u}$.

Note that L_G, for extended abstract reduction graph G, is an non-empty equivalence class w.r.t. to the $\approx$ relation (for legal strings). Since the definition of L_G does not depend on the exact identity of the vertices of G, we have, for extended abstract reduction graphs G and G', $G \approx G'$ implies $L_G = L_{G'}$.

Theorem 10

1. Let G and G' be extended abstract reduction graphs. Then $G \approx G'$ iff $L_G = L_{G'}$.

2. Let u and v be legal strings. Then $u \approx v$ iff $\mathcal{E}_u \approx \mathcal{E}_v$.

Proof

We first consider statement 1. We have already established the forward implication. We now prove the reverse implication. Let $G = B(E_1, E_2, E_3)$, $G' = B'(E'_1, E'_2, E'_3)$, and $L_G = L_{G'}$. By the definition of legalization, $B(E_1, E_3) \approx B'(E'_1, E'_3)$ and p is negative in G iff p is negative in G' for $p \in \mathrm{dom}(G) = \mathrm{dom}(G')$. Therefore, $G \approx G'$.

We now consider statement 2. We have $u \approx v$ iff $u, v \in L_{\mathcal{E}_u} = L_{\mathcal{E}_v}$ (since legalizations are equivalence classes of legal strings w.r.t $\approx$) iff $\mathcal{E}_u \approx \mathcal{E}_v$ (by the first statement). ■

Let G be an extended abstract reduction graph, and take $u \in L_G$ (such a u exists since L_G is nonempty). Since $u \in L_{\mathcal{E}_u}$ and legalizations are equivalence classes, we have $L_{\mathcal{E}_u} = L_G$ and therefore $G \approx \mathcal{E}_u$. Thus every extended abstract reduction graph G is isomorphic to an extended reduction graph. In fact, it is isomorphic to precisely those extended reduction graphs $\mathcal{E}_u$ with $u \in L_G$. Therefore, this u is unique up to equivalence.

Corollary 11

Let u and v be legal strings. If $\mathcal{R}_u \approx \mathcal{R}_v$, then there is a $E \in \mathrm{CON}_u$ such that $\mathcal{E}_v \approx B(E_1, E_2, E)$ with $\mathcal{R}_u = B(E_1, E_2)$.

Proof

Since $\mathcal{R}_u \approx \mathcal{R}_v$, there is a set of edges E for $\mathcal{R}_u$ such that $\mathcal{E}_v \approx B(E_1, E_2, E)$. Since $M_v \in \mathrm{CON}_v$, we have $E \in \mathrm{CON}_u$. ■

We end this section with a graph theoretical characterization of reduction graphs.

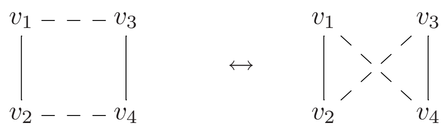


Figure 4.11: Flip operation for p. All vertices are labelled by p

Theorem 12
Let G be a 2-edge coloured graph. Then G is isomorphic to a reduction graph iff $G \in \mathcal{G}$ and $\mathrm{CON}_G \neq \varnothing$.

Proof
Let $G \approx \mathcal{R}_u$ for some legal string u. Then clearly, $G \in \mathcal{G}$. Also, $M_u \in \mathrm{CON}_u$ and hence $\mathrm{CON}_u \neq \varnothing$. Therefore, $\mathrm{CON}_G \neq \varnothing$.

Let $E \in \mathrm{CON}_G$. Then $G' = B(E_1, E_2, E)$ is an extended abstract reduction graph with $G = B(E_1, E_2)$. By the paragraph below Theorem 10, $G' \approx \mathcal{E}_u$ for some legal string u (take $u \in L_{G'}$). Hence, $G \approx \mathcal{R}_u$. ■

4.7 Flip Edges

In this section and the next two we provide characterizations of the statement $\mathrm{CON}_G \neq \varnothing$. This allows, using Theorem 12, for a characterization that corresponds to an efficient algorithm that determines whether or not a given $G \in \mathcal{G}$ is isomorphic to a reduction graph. Moreover, it allows for an efficient algorithm that determines a legal string u for which $G \approx \mathcal{R}_u$.

Let $G \in \mathcal{G}$. Then a merge-legal set for G is easily obtained as follows. For each $p \in \mathrm{dom}(G)$ with $C_p = \{\{v_1, v_2\}, \{v_3, v_4\}\}$, a merge-legal set for G must have either the edges $\{v_1, v_3\}$ and $\{v_2, v_4\}$ or the edges $\{v_1, v_4\}$ and $\{v_2, v_3\}$, see both sides in Figure 4.11. By assigning such edges for each $p \in \mathrm{dom}(G)$ we obtain a merge-legal set for G. Thus, $\mathrm{ML}_G \neq \varnothing$ for each $G \in \mathcal{G}$. Note that in particular, if $\mathrm{dom}(G) = \varnothing$, then $\mathrm{ML}_G = \{\varnothing\}$. However, CON_G can be empty as the next example illustrates.

Example 9
It is easy to see that the abstract reduction graph G of Figure 4.12 does not have a merge-legal set in CON_G.

We now formally define a type of operation that in Figure 4.11 transforms the situation on the left-hand side to the situation on the right-hand side, and the other way around. Informally speaking it "flips" edges of merge-legal sets.

Definition 13
Let $G = B(E_1, E_2) \in \mathcal{G}$, let f be the vertex labeling function of G, and let $p \in \mathrm{dom}(G)$. The *flip operation for p (w.r.t. G)* is the function $\mathrm{flip}_{G,p} : \mathrm{ML}_G \to \mathrm{ML}_G$

$$\begin{tikzcd}
s \arrow[r, equal] & 2 \arrow[r, dash] & 2 \arrow[r, equal] & 2 \arrow[r, dash] & 2 \arrow[r, equal] & t
\end{tikzcd}$$

$$\begin{tikzcd}
3 \arrow[r, equal] \arrow[d, dash] & 3 \arrow[d, dash] \\
3 \arrow[r, equal] & 3
\end{tikzcd}$$

Figure 4.12: An abstract reduction graph G for which $\mathrm{CON}_G = \varnothing$.

defined, for $E \in \mathrm{ML}_G$, by:

$$\mathrm{flip}_{G,p}(E) = \{\{v_1, v_2\} \in E \mid f(v_1) \neq p \neq f(v_2)\} \cup \{e_1, e_2\},$$

where e_1 and e_2 are the two edges with vertices labelled by p such that $e_1, e_2 \notin E_2 \cup E$. ■

When G is clear from the context, we also denote $\mathrm{flip}_{G,p}$ by flip_p.

Since by Figure 4.11, there are exactly two edges e_1 and e_2 with vertices labelled by p that are not parallel to both the edges in $E_2 \cup E$, flip_p is well defined. It is now easy to see that indeed $\mathrm{flip}_p(E) \in \mathrm{ML}_G$ for $E \in \mathrm{ML}_G$.

Example 10
Let G be the abstract reduction graph of Figure 4.6. If we apply $\mathrm{flip}_{G,2}$ to the set of merge-legal edges depicted in Figure 4.7, then we obtain the set of merge-legal edges depicted in Figure 4.8.

The next theorem follows directly from the previous definition and from the fact that Figure 4.11 contains the only possible ways in which edges in merge-legal sets for G can be connected.

Theorem 14
Let $G \in \mathcal{G}$, and denote by $\mathcal{F}$ be the group generated by the flip operations w.r.t. G under function composition. Then each element of $\mathcal{F}$ is self-inverse, thus $\mathcal{F}$ is Abelian, and $\mathcal{F}$ acts transitively on ML_G.

Let $D = \{p_1, \dots, p_l\} \subseteq \mathrm{dom}(G)$. Then we define $\mathrm{flip}_D = \mathrm{flip}_{p_l} \cdots \mathrm{flip}_{p_1}$. Since $\mathcal{F}$ is Abelian, flip_D is well defined. Moreover, since each each element in $\mathcal{F}$ is self-inverse, $\mathcal{F} = \{\mathrm{flip}_D \mid D \subseteq \mathrm{dom}(G)\}$. Also, if $D_1, D_2 \subseteq \mathrm{dom}(G)$ and $D_1 \neq D_2$, then $\mathrm{flip}_{D_1}(E) \neq \mathrm{flip}_{D_2}(E)$. Thus the following holds.

Theorem 15
Let $G \in \mathcal{G}$. Then there is a bijection $Q : 2^{\mathrm{dom}(G)} \to \mathcal{F}$ given by $Q(D) = \mathrm{flip}_D$. Moreover, for each $E \in \mathrm{ML}_G$, $\mathrm{ML}_G = \{\mathrm{flip}_D(E) \mid D \subseteq \mathrm{dom}(G)\}$.

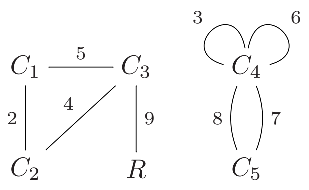


Figure 4.13: The pointer-component graph of the abstract reduction graph from Figure 4.4.

4.8 Merging and Splitting Connected Components

Let $G = B(E_1, E_2)$ be an abstract reduction graph and let $E \in \mathrm{ML}_G$. In this section we consider the effect of the flip operation on the pointer-component graph defined on the abstract reduction graph $H = B(E_1, E)$. If we are able to obtain, using flip operations, a pointer-component graph consisting of one vertex, then $\mathrm{CON}_G \neq \varnothing$, and consequently by Theorem 12, G is isomorphic to a reduction graph.

However, first we need to define the notion of pointer-component graph for abstract reduction graphs in general. Fortunately, this generalization is trivial.

Definition 16
Let $G \in \mathcal{G}$. The *pointer-component graph of* G, denoted by $\mathcal{PC}_G$, is a multigraph (ζ, E, ϵ), where ζ is the set of connected components of G, $E = \mathrm{dom}(G)$, and ϵ is, for $e \in E$, defined by $\epsilon(e) = \{C \in \zeta \mid C \text{ contains vertices labelled by } e\}$. ■

Example 11
The pointer-component graph of the graph from Figure 4.4 is shown in Figure 4.13.

Note that when $G = B(E_1, E_2) \in \mathcal{G}$ and $E \in \mathrm{ML}_G$, then E is desirable for B. Hence, $H = B(E_1, E)$ is also an abstract reduction graph. Therefore, e.g., $\mathcal{PC}_H$ is defined.

It is useful to distinguish the pointers that form loops in the pointer-component graph. Therefore, we define, for $G \in \mathcal{G}$, $\mathrm{bridge}(G) = \{e \in E \mid |\epsilon(e)| = 2\}$ where $\mathcal{PC}_G = (V, E, \epsilon)$. In [4], bridge$(G)$ is denoted as snrdom(G). However, this notation does not make sense for its uses in this chapter.

Example 12
From Figure 4.13 it follows that $\mathrm{bridge}(G) = \mathrm{dom}(G)\backslash\{3, 6\}$ for the abstract reduction graph G depicted in Figure 4.4.

Merge rules have been used for multigraphs, and pointer-component graphs in particular in [4]. The definition presented here is slightly different from the one in [4] – here the pointer p on which the merge rule is applied remains present after the rule is applied.

Definition 17
For each edge p, the *p-merge rule*, denoted by merge_p, is a rule applicable to (defined on) multigraphs $G = (V, E, \epsilon)$ with $p \in \text{bridge}(G)$. It is defined by

$$\text{merge}_p(G) = (V', E, \epsilon'),$$

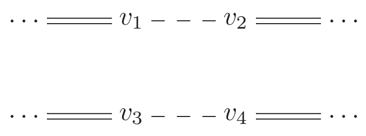
where $V' = (V \backslash \epsilon(p)) \cup \{v'\}$ with $v' \notin V$, and $\epsilon'(e) = \{h(v_1), h(v_2)\}$ iff $\epsilon(e) = \{v_1, v_2\}$ where $h(v) = v'$ if $v \in \epsilon(p)$, otherwise it is the identity. ■

It is easy to see that merge rules commute. We are now ready to state the following result which is similar to Theorem 27 in [4].

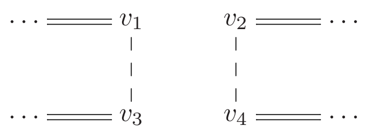
Theorem 18
Let $G = B(E_1, E_2) \in \mathcal{G}$, let $E \in \text{ML}_G$, let $H = B(E_1, E)$, and let, for $p \in \text{dom}(G)$, $H_p = B(E_1, \text{flip}_p(E))$.

- If $p \in \text{bridge}(H)$, then $\mathcal{PC}_{H_p} \approx \text{merge}_p(\mathcal{PC}_H)$
  (and therefore $o(\mathcal{PC}_{H_p}) = o(\mathcal{PC}_H) - 1$).
- If $p \in \text{dom}(H) \backslash \text{bridge}(H)$, then $o(\mathcal{PC}_H) \leq o(\mathcal{PC}_{H_p}) \leq o(\mathcal{PC}_H) + 1$.

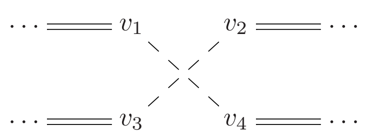
Proof
First let $p \in \text{bridge}(H)$. Let $C_{H,p} = \{\{v_1, v_2\}, \{v_3, v_4\}\}$. Then, H has the following form, where each of the two edges in $C_{H,p}$ are from different connected components in H and where, unlike our convention, we have depicted the vertices by their identity instead of their label:



Now, either $\{\{v_1, v_4\}, \{v_2, v_3\}\} \subseteq E_2$ or $\{\{v_1, v_3\}, \{v_2, v_4\}\} \subseteq E_2$. Thus H_p is of either



or

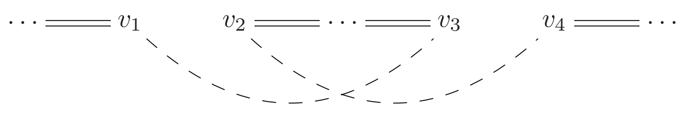


form, respectively. Thus in both cases, the two connected components are merged, and thus $\mathcal{PC}_{H_p}$ can be obtained (up to isomorphism) from $\mathcal{PC}_H$ by applying the merge_p operation.

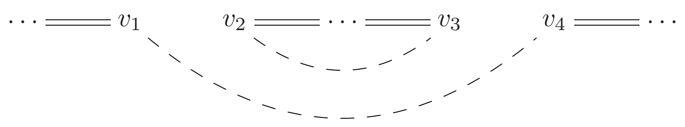
Now let $p \in \text{dom}(H)\backslash\text{bridge}(H)$. Then the edges in $C_{H,p}$ belong to the same connected component. Thus H has the following form

$$\cdots = v_1 - - - v_2 = \cdots = v_3 - - - v_4 = \cdots$$

where $C_{H,p} = \{\{v_1, v_2\}, \{v_3, v_4\}\}$. Again, we have either $\{\{v_1, v_4\}, \{v_2, v_3\}\} \subseteq E_2$ or $\{\{v_1, v_3\}, \{v_2, v_4\}\} \subseteq E_2$. Thus H_p is of either



or



form, respectively. Thus, H_p has either the same number of connected components of H or exactly one more, respectively. Thus, $o(\mathcal{PC}_H) \leq o(\mathcal{PC}_{H_p}) \leq o(\mathcal{PC}_H) + 1$. ■

Example 13
Let $G = B(E_1, E_2) \in \mathcal{G}$ be as in Figure 4.6. If we take $E \in \text{ML}_G$ as in Figure 4.7, then $2 \in \text{bridge}(H)$ with $H = B(E_1, E)$. Therefore, by Theorem 18 and the fact that G has exactly two connected components, $H_2 = B(E_1, \text{flip}_2(E))$ is a connected graph. Indeed, this is clear from Figure 4.8 (by ignoring the edges from E_2).

Informally, the next lemma shows that by applying flip operations, we can shrink a connected pointer-component graph to a single vertex. In this way, the underlying abstract reduction graph is a connected graph.

Remark
The next lemma appears to be similar to Lemma 29 in [4]. Although the flip operation (defined on graphs) and the rem operation (defined on strings) are quite distinct, they do have a similar effect on the pointer-component graph.

Lemma 19
Let $G = B(E_1, E_2) \in \mathcal{G}$, let $E \in \text{ML}_G$, let $H = B(E_1, E)$, and let $D \subseteq \text{dom}(G) = \text{dom}(H)$. Then $\mathcal{PC}_H|_D$ is a tree iff $B(E_1, \text{flip}_D(E))$ and H have 1 and $|D| + 1$ connected components, respectively.

Proof
Let $D = \{p_1, \ldots, p_n\}$. We first prove the forward implication. If $\mathcal{PC}_H|_D$ is a tree, then it has $|D|$ edges, and thus $|D| + 1$ vertices. Therefore, $\mathcal{PC}_H$ has $|D| + 1$

vertices, and consequently, H has $|D|+1$ connected components. Since $\mathcal{PC}_H|_D$ is acyclic, by Theorem 18,

$$\mathcal{PC}_{B(E_1,\mathrm{flip}_D(E))} = \mathcal{PC}_{B(E_1,(\mathrm{flip}_{p_n}\ \cdots\ \mathrm{flip}_{p_1})(E))} \approx (\mathrm{merge}_{p_n}\ \cdots\ \mathrm{merge}_{p_1})(\mathcal{PC}_H).$$

Now, applying $|D|$ merge operations on a graph with $|D|+1$ vertices, results in a graph containing exactly one vertex. Thus $B(E_1, \mathrm{flip}_D(E))$ has one connected component.

We now prove the reverse implication. Moving from $H = B(E_1, E)$ to graph $B(E_1, \mathrm{flip}_D(E))$ reduces the number of connected components in $|D|$ steps from $|D|+1$ to 1. By Theorem 18, each flip operation of flip_D corresponds to a merge operation. Therefore $(\mathrm{merge}_{p_n}\ \cdots\ \mathrm{merge}_{p_1})$ is applicable to $\mathcal{PC}_H$. Consequently, $\mathcal{PC}_H|_D$ is acyclic. Since this graph has $|D|+1$ vertices, $\mathcal{PC}_H|_D$ is a tree. ■

4.9 Connectedness of Pointer-Component Graph

In this section we use the results of the previous two sections to prove our first main result, cf. Theorem 24, which strengthens Theorem 12 by replacing the requirement $\mathrm{CON}_G \neq \varnothing$ by a simple test on $\mathcal{PC}_G$. We now characterize the connectedness of $\mathcal{PC}_G$.

Definition 20
Let $B = (V, f, s, t)$ be a coloured base. We say that a set of edges E for B is *well-coloured (for B)* if for each partition $\rho = (V_1, V_2)$ of V with $f(V_1) \cap f(V_2) = \varnothing$, there is an edge $\{v_1, v_2\} \in E$ with $v_1 \in V_1$ and $v_2 \in V_2$. ■

We call $G = B(E_1, E_2) \in \mathcal{G}$ *well-coloured* if E_1 is well-coloured for B.

Lemma 21
Let $G \in \mathcal{G}$. Then $\mathcal{PC}_G$ is a connected graph iff G is well-coloured.

Proof
Let $G = B(E_1, E_2)$ with $B = (V, f, s, t)$. We first prove the forward implication. Let G be not well-coloured. Then there is a partition $\rho = (V_1, V_2)$ of V with $f(V_1) \cap f(V_2) = \varnothing$ such that for each $e \in E_1$, either $e \subseteq V_1$ or $e \subseteq V_2$. Since for each $\{v_1, v_2\} \in E_2$ we have $f(v_1) = f(v_2)$, we have either $\{v_1, v_2\} \subseteq V_1$ or $\{v_1, v_2\} \subseteq V_2$. Therefore V_1 and V_2 induce two non-empty sets of connected components which have no vertex label in common. Therefore, $\mathcal{PC}_G$ is not a connected graph.

We now prove the reverse implication. Assume that $\mathcal{PC}_G = (\zeta, E, \epsilon)$ is not a connected graph. Then, by the definition of pointer-component graph, there is a partition (C_1, C_2) of ζ such that C_1 and C_2 have no vertex label in common. Let V_i be the set of vertices of the connected components in C_i ($i \in \{1, 2\}$). Then for partition $\rho = (V_1, V_2)$ of V we have $f(V_1) \cap f(V_2) = \varnothing$ and for each $e \in E_1 \cup E_2$, either $e \subseteq V_1$ or $e \subseteq V_2$. Therefore G is not well-coloured. ■

Clearly, if $G = B(E_1, E_2) \in \mathcal{G}$ is well-coloured and E is desirable for B (e.g., one could take $E \in \mathrm{ML}_G$), then $H = B(E_1, E) \in \mathcal{G}$ and H is well-coloured. Therefore, by Lemma 21, $\mathcal{PC}_G$ is a connected graph iff $\mathcal{PC}_H$ is a connected graph.

By Theorem 12 the next result is essential to efficiently determine which abstract reduction graphs are isomorphic to reduction graphs.

Theorem 22
Let $G \in \mathcal{G}$. Then $\mathcal{PC}_G$ is a connected graph iff $\mathrm{CON}_G \neq \varnothing$.

Proof
Let $G = B(E_1, E_2)$. We first prove the forward implication. Let $\mathcal{PC}_G$ be a connected graph and let $E \in \mathrm{ML}_G$. Then $\mathcal{PC}_H$ with $H = B(E_1, E)$ is a connected graph. Thus there exists a $D \subseteq \mathrm{dom}(G)$ such that $\mathcal{PC}_H|_D$ is a tree. By Lemma 19, $B(E_1, \mathrm{flip}_D(E))$ is a connected graph, and consequently $\mathrm{flip}_D(E) \in \mathrm{CON}_G$.

We now prove the reverse implication. Let $E \in \mathrm{CON}_G$. Thus, $H = B(E_1, E)$ is a connected graph, and hence $\mathcal{PC}_H$ is a connected graph. Therefore, $\mathcal{PC}_G$ is also a connected graph. ■

We can summarize the last two results as follows.

Corollary 23
Let $G \in \mathcal{G}$. Then the following conditions are equivalent:

1. G is well-coloured,
2. $\mathcal{PC}_G$ is a connected graph, and
3. $\mathrm{CON}_G \neq \varnothing$.

Example 14
By Figure 4.3 and Corollary 23, for (abstract) reduction graph G_1 in Figure 4.2 we have $\mathrm{CON}_{G_1} \neq \varnothing$. On the other hand, by Figure 4.13 and Corollary 23, for abstract reduction graph G_2 in Figure 4.4 we have $\mathrm{CON}_{G_2} = \varnothing$.

By Corollary 23 and Theorem 12 we obtain the first main result of this chapter. It shows that one needs to check only a few computationally easy conditions to determine whether or not a 2-edge coloured graph is (isomorphic to) a reduction graph. Surprisingly, the 'high-level' notion of pointer-component graph is crucial in this characterization.

Theorem 24
Let G be a 2-edge coloured graph. Then G isomorphic to a reduction graph iff $G \in \mathcal{G}$ and $\mathcal{PC}_G$ is a connected graph.

Note that in the previous theorem we can equally well replace "$\mathcal{PC}_G$ is a connected graph" by one of the other equivalent conditions in Corollary 23.

In Theorem 21 in [4] it is shown that the pointer-component graph of each reduction graph is a connected graph. We did not use that result here – in fact it is now a direct consequence of Theorem 24.

Not only is it computationally efficient to determine whether or not a 2-edge coloured graph G is isomorphic to a reduction graph, but, when this is the case, then it is also computationally easy to determine a legal string u for which $G \approx \mathcal{R}_u$. Indeed, we can determine such a u from $G = B(E_1, E_2)$ as follows:

1. Determine an $E \in \mathrm{ML}_G$. As we have mentioned before, such an E is easily obtained.
2. Compute $\mathcal{PC}_H$ with $H = B(E_1, E)$, and determine a set of edges D such that $\mathcal{PC}_H|_D$ is a tree.
3. Compute $G' = B(E_1, E_2, \mathrm{flip}_D(E))$, and determine a $u \in L_{G'}$.

As a consequence, pointer-component graphs of legal strings can, surprisingly, take all imaginable forms.

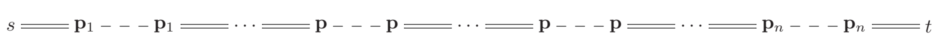
Corollary 25
Every connected multigraph $G = (V, E, \epsilon)$ with $E \subseteq \Delta$ is isomorphic to a pointer-component graph of a legal string.

4.10 Flip and the Underlying Legal String

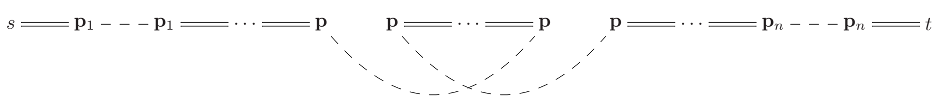
We now move to the second part of this chapter, where we characterize the fibers $\mathcal{R}^{-1}(\mathcal{R}_u)$ modulo graph isomorphism. Thus, we describe the set of strings that have the same reduction graph (up to isomorphism) as u. First we consider the effect of flip operations on the set of merge edges.

Lemma 26
Let u be a legal string and let $p \in \mathrm{dom}(u)$. If p is negative in u, then $\mathrm{flip}_p(M_u) \in \mathrm{CON}_u$. If p is positive in u, then $\mathrm{flip}_p(M_u) \notin \mathrm{CON}_u$. In other words, $\mathrm{flip}_p(M_u) \in \mathrm{CON}_u$ iff p is negative in u.

Proof
Let $\mathcal{R}_u = B(E_1, E_2)$. By the definition of flip_p, $\mathrm{flip}_p(M_u) \in \mathrm{ML}_u$. It suffices to prove that $G = B(E_1, \mathrm{flip}_p(M_u))$ is a connected graph when p is negative in u and not a connected graph when p is positive in u. Graph $B(E_1, M_u)$ has the following form:

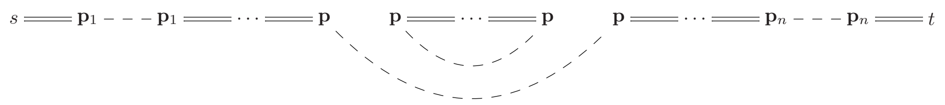


Now if p is negative in u, then G has the following form:



Thus in this case G is connected.

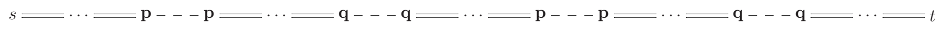
If p is positive in u, then G has the following form:



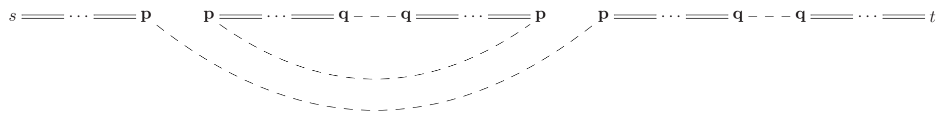
Thus in this case G is not connected. ∎

Lemma 27
Let u be a legal string and let $p, q \in \mathrm{dom}(u)$. If p and q are overlapping in u *and* not both negative in u, then $\mathrm{flip}_{\{p,q\}}(M_u) \in \mathrm{CON}_u$.

Proof
Let $\mathcal{R}_u = B(E_1, E_2)$. Then $B(E_1, M_u)$ has the following form (we can assume without loss of generality that p appears before q in the path from s to t):



Assume that p is positive in u – the other case (q is positive in u) is proved similarly. By the proof of Lemma 26 it follows that $B(E_1, \mathrm{flip}_p(M_u))$ has the following form:



Therefore, $q \in \mathrm{bridge}(B(E_1, \mathrm{flip}_p(M_u)))$. By Theorem 18, the pointer-component graph of $B(E_1, \mathrm{flip}_{\{p,q\}}(M_u))$ has only one vertex. Hence, $B(E_1, \mathrm{flip}_{\{p,q\}}(M_u))$ is connected and thus $\mathrm{flip}_{\{p,q\}}(M_u) \in \mathrm{CON}_u$. ∎

Lemma 28
Let u be a legal string, and let $D \subseteq \mathrm{dom}(u)$ be nonempty. If $\mathrm{flip}_D(M_u) \in \mathrm{CON}_u$, then either there is a $p \in D$ negative in u or there are $p, q \in D$ positive and overlapping in u.

Proof
Let $\mathcal{E}_u = B(E_1, E_2, M_u)$ and let $\mathrm{flip}_D(M_u) \in \mathrm{CON}_u$. Then $B(E_1, \mathrm{flip}_D(M_u))$ is a connected graph. Assume to the contrary that all elements in D are positive and pairwise non-overlapping in u. Then there is a $p \in D$ such that the domain of the p-interval does not contain an element in $D \backslash \{p\}$. By the proof of Lemma 26, $B(E_1, \mathrm{flip}_p(M_u))$ consists of two connected components, one of which does not have vertices labelled by elements in $D \backslash \{p\}$. Therefore $B(E_1, \mathrm{flip}_D(M_u))$ also contains this connected component, and thus $B(E_1, \mathrm{flip}_D(M_u))$ has more than one connected component – a contradiction. ∎

By the previous lemmata, we have the following result.

Theorem 29
Let u be a legal string, and let $D \subseteq \mathrm{dom}(u)$ be nonempty. If $\mathrm{flip}_D(M_u) \in \mathrm{CON}_u$, then either there is a $p \in D$ negative in u with $\mathrm{flip}_p(M_u) \in \mathrm{CON}_u$ or there are $p, q \in D$ positive and overlapping in u with $\mathrm{flip}_{\{p,q\}}(M_u) \in \mathrm{CON}_u$.

4.11 Dual String Pointer Rules

We now define the dual string pointer rules. These rules will be used to characterize the effect of flip operations on the underlying legal string. For all $p, q \in \Pi$ with $\mathbf{p} \neq \mathbf{q}$, we define

- the *dual string positive rule* for p by $\mathbf{dspr}_p(u_1 p u_2 p u_3) = u_1 p \bar{u}_2 p u_3$,
- the *dual string double rule* for p, q by $\mathbf{dsdr}_{p,q}(u_1 p u_2 q u_3 \bar{p} u_4 \bar{q} u_5) = u_1 p u_4 q u_3 \bar{p} u_2 \bar{q} u_5$,

where $u_1, u_2, \dots, u_5$ are arbitrary (possibly empty) strings over Π. Notice that the dual string pointer rules are self-inverse.

The names of these rules are due to their strong similarities with the two of the three types of string rewriting rules of a specific model of gene assembly, called string pointer reduction system (SPRS) [12]. In this model, gene assembly is performed by three types of recombination (splicing) operations that are subsequently modeled as string rewriting rules. For convenience we now recall these string rewriting rules.

For all $p, q \in \Pi$ with $\mathbf{p} \neq \mathbf{q}$, we define

- the *string negative rule* for p by $\mathbf{snr}_p(u_1 p p u_2) = u_1 u_2$,
- the *string positive rule* for p by $\mathbf{spr}_p(u_1 p u_2 \bar{p} u_3) = u_1 \bar{u}_2 u_3$,
- the *string double rule* for p, q by $\mathbf{sdr}_{p,q}(u_1 p u_2 q u_3 p u_4 q u_5) = u_1 u_4 u_3 u_2 u_5$,

where $u_1, u_2, \dots, u_5$ are arbitrary (possibly empty) strings over Π.

Notice the strong similarities between **dspr** and **spr**, and between **dsdr** and **sdr**. Both $\mathbf{dspr}_p$ and $\mathbf{spr}_p$ invert the substring between the two occurrences of p or $\bar{p}$. However, $\mathbf{dspr}_p$ is applicable when p is negative, while $\mathbf{spr}_p$ is applicable when p is positive. Also, $\mathbf{spr}_p$ removes the occurrences of p and $\bar{p}$, while **dspr** does not. A similar comparison can be made between **dsdr** and **sdr**.

The *domain* of a dual string pointer rule ρ, denoted by $\mathrm{dom}(\rho)$, is defined by $\mathrm{dom}(\mathbf{dspr}_p) = \{\mathbf{p}\}$ and $\mathrm{dom}(\mathbf{dsdr}_{p,q}) = \{\mathbf{p}, \mathbf{q}\}$ for $p, q \in \Pi$. For a composition $\varphi = \rho_n \ \cdots \ \rho_2 \ \rho_1$ of such rules $\rho_1, \rho_2, \dots, \rho_n$, the *domain*, denoted by $\mathrm{dom}(\varphi)$, is $\mathrm{dom}(\rho_1) \cup \mathrm{dom}(\rho_2) \cup \cdots \cup \mathrm{dom}(\rho_n)$. Also, we define $\mathrm{odom}(\varphi) = \bigoplus_{1 \leq i \leq n} \mathrm{dom}(\rho_i)$. Thus, $\mathrm{odom}(\varphi) \subseteq \mathrm{dom}(\varphi)$ consists of the pointers that are used an odd number of times. We call φ *reduced* if every $p \in \mathrm{dom}(\varphi)$ is used exactly once, i.e., $\mathrm{dom}(\rho_i) \cap \mathrm{dom}(\rho_j) = \varnothing$ for all $1 \leq i < j \leq n$. Note that if φ is reduced, then $\mathrm{dom}(\varphi) = \mathrm{odom}(\varphi)$.

Definition 30
Let u and v be legal strings. We say that u and v are *dual*, denoted by $\approx_d$ if there is a (possibly empty) sequence φ of dual string pointer rules applicable to u such that $\varphi(u) \approx v$. ∎

Notice that $\approx_d$ is an equivalence relation. Clearly, $\approx_d$ is reflexive. It is symmetrical since dual string pointer rules are self-inverse, and it is transitive by function composition: if $\varphi_1(u) \approx v$ and $\varphi_2(v) \approx w$, then $(\varphi_2\ \varphi_1)(u) \approx w$.

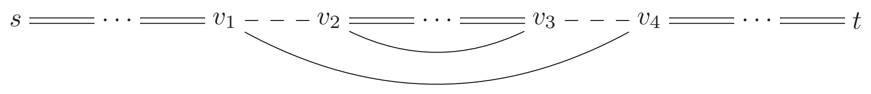
Since $\mathbf{dspr}_p$ is applicable when p is negative in u and $\mathbf{dsdr}_{p,q}$ is applicable when p and q are positive and overlapping, the following result is a direct corollary to Lemma 28.

Corollary 31
Let u be a legal string, and let $D \subseteq \mathrm{dom}(u)$ be nonempty. If $\mathrm{flip}_D(M_u) \in \mathrm{CON}_u$, then there is a dual string pointer rule ρ with $\mathrm{dom}(\rho) \subseteq D$ applicable to u.

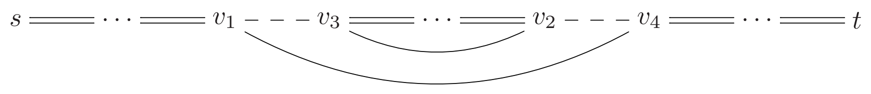
Let $G = B(E_1, E_2, E_3)$ be an extended abstract reduction graph, and let $D \subseteq \mathrm{dom}(G)$. Then we define $\mathrm{flip}_D(G) = B(E_1, E_2, \mathrm{flip}_{G',D}(E_3))$, where $G' = B(E_1, E_2)$.

Lemma 32
Let u be a legal string, and let φ be a sequence of dual string rules applicable to u. Then $\mathcal{E}_{\varphi(u)} \approx \mathrm{flip}_D(\mathcal{E}_u)$ with $D = \mathrm{odom}(\varphi)$. Consequently, $\mathcal{R}_{\varphi(u)} \approx \mathcal{R}_u$.

Proof
It suffices to prove the result for the case $\varphi = \mathbf{dspr}_p$ with $p \in \Pi$ and for the case $\varphi = \mathbf{dsdr}_{p,q}$ with $p, q \in \Pi$. We first prove the case where $\varphi = \mathbf{dspr}_p$ for some $p \in \Pi$ is applicable to u. Then by the second figure in the proof of Lemma 26 we see that the inversion of the substring between the two occurrences of p in u accomplished by φ faithfully simulates the corresponding effect of flip_p on $\mathcal{E}_u$. We only need to verify that p is negative in $\mathrm{flip}_p(\mathcal{E}_u)$. To do this, we depict $\mathcal{E}_u$ such that the vertices are represented by their identity instead of their label:



where the vertices v_i, $i \in \{1, 2, 3, 4\}$, are labelled by $\mathbf{p}$. Then $\mathrm{flip}_p(\mathcal{E}_u)$ is

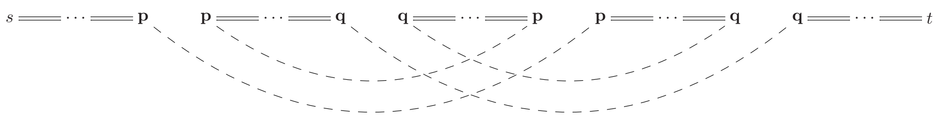


Therefore p is indeed negative in $\mathrm{flip}_p(\mathcal{E}_u)$, and consequently $\mathcal{E}_{\varphi(u)} \approx \mathrm{flip}_p(\mathcal{E}_u)$.

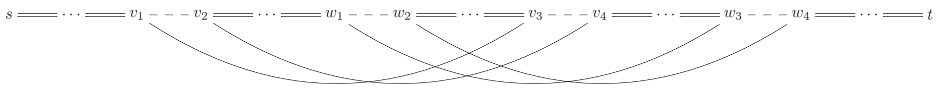
We now prove the case where $\varphi = \mathbf{dsdr}_{p,q}$ with $p, q \in \Pi$. Let $\mathcal{E}_u = B(E_1, E_2, E_3)$, then $\mathcal{E}_u$ has the following form

$$s = \cdots = \mathbf{p} - - - \mathbf{p} = \cdots = \mathbf{q} - - - \mathbf{q} = \cdots = \mathbf{p} - - - \mathbf{p} = \cdots = \mathbf{q} - - - \mathbf{q} = \cdots = t$$

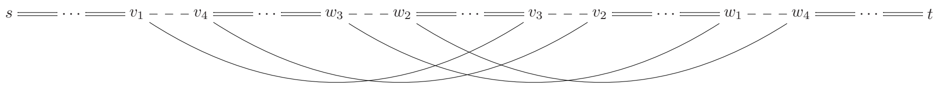
where we omitted the edges in E_2. Since p and q are positive in u, $\mathrm{flip}_{\{p,q\}}(\mathcal{E}_u)$ has the following form:



where we again omitted the edges in E_2. Thus, we see that interchanging the substring in u between p and q and the substring in u between $\bar{p}$ and $\bar{q}$ accomplished by φ faithfully simulates the corresponding effect of $\mathrm{flip}_{p,q}$ on $\mathcal{E}_u$. We only need to verify that both p and q are positive in $\mathrm{flip}_{p,q}(\mathcal{E}_u)$. To do this, we depict $\mathcal{E}_u$ such that the vertices are represented by their identity instead of their label:



where the vertices v_i and w_i, $i \in \{1,2,3,4\}$, are labelled by $\mathbf{p}$ and $\mathbf{q}$, respectively. Then $\mathrm{flip}_{p,q}(\mathcal{E}_u)$ is



Therefore both p and q are indeed positive in $\mathrm{flip}_{p,q}(\mathcal{E}_u)$, and consequently $\mathcal{E}_{\varphi(u)} \approx \mathrm{flip}_{p,q}(\mathcal{E}_u)$. ∎

Thus, if φ_1 and φ_2 are sequences of dual string pointer rules applicable to a legal string u with $\mathrm{odom}(\varphi_1) = \mathrm{odom}(\varphi_2)$, then $\mathcal{E}_{\varphi_1(u)} \approx \mathcal{E}_{\varphi_2(u)}$ and thus $\varphi_1(u) \approx \varphi_2(u)$.

Lemma 33
Let u be a legal string, and let $D \subseteq \mathrm{dom}(u)$. There is a reduced sequence φ of dual string pointer rules applicable to u such that $\mathrm{dom}(\varphi) = D$ iff $\mathrm{flip}_D(M_u) \in \mathrm{CON}_u$.

Proof
The forward implication follows directly from Lemma 32. We now prove the reverse implication. If $D = \varnothing$, we have nothing to prove. Let $D \neq \varnothing$. By Corollary 31, there is a dual string pointer rule ρ_1 with $D_1 = \mathrm{dom}(\rho_1) \subseteq D$ applicable to u. By Lemma 32, $\mathcal{E}_{\rho_1(u)} \approx \mathrm{flip}_{D_1}(\mathcal{E}_u)$ and $D_1 = \mathrm{odom}(\rho_1) = \mathrm{dom}(\rho_1)$. Thus, $\mathrm{flip}_{D \setminus D_1}(M_{\rho_1(u)}) \in \mathrm{CON}_{\rho_1(u)}$. Now by iteration, there is a reduced sequence φ of dual string pointer rules applicable to u such that $\mathrm{odom}(\varphi) = \mathrm{dom}(\varphi) = D$. ∎

It follows from Lemmata 32 and 33 that reduced sequences of dual string pointer rules are a normal form of sequences of dual string pointer rules. Indeed, by Lemma 32, if φ is a sequence of dual string pointer rules applicable to a legal string u with $D = \mathrm{odom}(\varphi)$, then $\mathrm{flip}_D(M_u) \in \mathrm{CON}_u$. By Lemma 33, there is a reduced sequence φ' of dual string pointer rules applicable to u such that $\mathrm{dom}(\varphi') = \mathrm{odom}(\varphi') = D$. By the paragraph below Lemma 32, we have $\varphi(u) \approx \varphi'(u)$.

We are now ready to prove the second (and final) main result of this chapter. It shows that the fiber $\mathcal{R}^{-1}(\mathcal{R}_u)$ for each legal string u is the 'orbit' of u under the dual string pointer rules. Hence, the partition of the set of all legal strings induced by the fibers under $\mathcal{R}$, and the one induced by $\approx_d$ coincide. Equivalently, the legal strings obtained from u by applying dual string pointer rules are exactly those legal strings that have the same reduction graph as u (up to isomorphism).

Theorem 34
Let u and v be legal strings. Then $\mathcal{R}_u \approx \mathcal{R}_v$ iff $u \approx_d v$.

Proof
The reverse implication follows directly from Lemma 32. We now prove the forward implication. Let $\mathcal{R}_u \approx \mathcal{R}_v$. By Corollary 11, there is a $E \in \mathrm{CON}_u$ such that $\mathcal{E}_v \approx B(E_1, E_2, E)$ with $\mathcal{R}_u = B(E_1, E_2)$. By Theorem 15, $E = \mathrm{flip}_D(M_u)$ for some $D \subseteq \mathrm{dom}(u)$. Since $\mathrm{flip}_D(M_u) \in \mathrm{CON}_u$, by Lemma 33, there is a reduced sequence φ of dual string pointer rules applicable to u such that $\mathrm{dom}(\varphi) = D$. Now by Lemma 32, $\mathcal{E}_{\varphi(u)} \approx \mathrm{flip}_D(\mathcal{E}_u) \approx \mathcal{E}_v$, and therefore, by Theorem 10, $\varphi(u) \approx v$. ■

4.12 Discussion

This chapter characterizes, letting $\mathcal{R}$ be the function that assigns to each legal string u its reduction graph $\mathcal{R}_u$, the range of $\mathcal{R}$ (Theorem 24) and each fiber $\mathcal{R}^{-1}(\mathcal{R}_u)$ (Theorem 34) modulo graph isomorphism.

The first characterization corresponds to a computationally efficient algorithm that determines whether or not a graph G is isomorphic to a reduction graph. Moreover, if this is the case, then the algorithm given below Theorem 24 allows for an efficient determination of a legal string u such that $G \approx \mathcal{R}_u$. The first characterization relies on the notion of merge-legal edges and its flip operation introduced in this chapter. In particular, the connected components in the subgraph induced by the reality edges and the merge-legal edges and the flip operation turn out to be relevant in this context.

The second characterization determines, given u, the whole set $\mathcal{R}^{-1}(\mathcal{R}_u)$. It turns out that $\mathcal{R}^{-1}(\mathcal{R}_u)$ is the orbit of u under the dual string pointer rules. Moreover, each two legal strings u and v in such a fiber can be transformed into each other by a sequence φ of dual string pointer rules without using any pointer more than once. Therefore, the number of dual string pointer rules in φ can be bounded by the size of the domain of u (and v). Surprisingly, the dual string pointer rules are very similar to those used in a specific model of gene assembly called SPRS.

The second characterization has additional uses for SPRS. The reduction graph of a legal string u in a certain sense retains all information regarding applicability of string negative rules (defined in SPRS) in transformations of u to its end result, while discarding almost all other information regarding the applicability of the other rules, see [4]. Therefore, the fibers in a sense provide the equivalence classes of legal strings having the same properties regarding the application of string negative rules.

From a biological point of view, the first characterization provide requirements on the structure of MAC genes, while the second characterization determines which types of MIC genes obtain the same MAC structure.