

The SymmetricGroup Class User's Manual for Partitioning and Ordering Problems

Victor D. Wiley
The University of Texas at Austin

October 24, 2000

Abstract

The Symmetric Group on n letters structure allows it to easily represent partitioning and ordering problems. This manual is intended to provide details of how the SymmetricGroup class has been derived and how it can be used.

Contents

1	The Symmetric Group on n letters	2
2	The SymmetricGroup class	2
2.1	Data members	2
2.1.1	longForm	2
2.1.2	cyclicForm	3
2.1.3	numLetters	3
2.1.4	cyclicFormValid	3
2.2	Methods	3
2.2.1	Constructors	3
2.2.2	Group methods	4
2.2.3	SymmetricGroup public methods	5
3	Calling the SymmetricGroup class	18
4	Using the SymmetricGroup class	18
4.1	The Vehicle Routing Problem	19
4.1.1	VRP Formulation	19
4.1.2	TabuSearch	22
4.1.3	Tabu Search and the SymmetricGroup class	24

1 The Symmetric Group on n letters

The Symmetric Group on n letters, S_n , is the group of permutations of the set $\{1, 2, \dots, n\}$ of integers from 1 to n . Elements of S_4 include:

4 1-cycles: $()$
1 2-cycle and 2 1-cycles: $(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)$
1 3-cycle and 1 1-cycle: $(1, 2, 3), (1, 2, 4), (1, 3, 4), (2, 3, 4),$
 $(1, 3, 2), (1, 4, 2), (1, 4, 3), (2, 4, 3)$
2 2-cycles: $(1, 2)(3, 4), (1, 3)(2, 4), (1, 4)(2, 3)$
1 4-cycle: $(1, 2, 3, 4), (1, 2, 4, 3), (1, 3, 2, 4),$
 $(1, 3, 4, 2), (1, 4, 2, 3), (1, 4, 3, 2)$

As S_4 demonstrates, the partitioning of letters corresponds to the cycle structure and the ordering corresponds to the arrangement of the letters within each cycle.¹

2 The SymmetricGroup class

The `SymmetricGroup` class is an extension of the `Group Class` (Wiley 2000). The `SymmetricGroup` class provides the user with the data structures necessary to represent elements of S_n and methods to manipulate these elements.

2.1 Data members

The `SymmetricGroup` class provides structure for both long and cyclic representations. It also contains additional bookkeeping members when using the cyclic form.

2.1.1 longForm

The long form representation of a `SymmetricGroup` class element is an integer array. This array stores the image of the letter represented by the index of the array. Basically, if a letter is being moved if it differs from its index position within the array. Example 1 shows the long form representation of the element $(1, 2)(3)(4)$ in S_4 .

Example 1 *Long Form:* $\begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 3 & 4 \end{pmatrix}$

The actual value stored by the `SymmetricGroup` class object would be $(2, 1, 3, 4)$.

¹By convention, 1-cycles are omitted in the cyclic representation of a permutation with the identity permutation being denoted by $()$.

2.1.2 cyclicForm

The cyclic form representation of a `SymmetricGroup` class element is a `Cycles` object that contains integer array objects. The `Cycles` object stores an integer array for each cycle of the element. Therefore, the cyclic representation of the Long Form Example would be stored in the `Cycles` as the arrays $(1, 2)$, (3) , and (4) . This form provides a more compact way of representing elements of S_n when n is large since, by convention, only the non-trivial (more than one element) cycles are displayed. The cycles stored in the `Cycles` object are automatically arranged in cyclic standard form, i.e. leftmost cycle contains the smallest letter and the smallest letter within a cycle occupies the leftmost position.

2.1.3 numLetters

The `numLetters` data member is an integer used as a bookkeeping measure when using the cyclic form representation. Since the permutation $(1, 2)$ can be drawn from any Symmetric Group with $n \geq 2$, this data member simply keeps track of the appropriate n .

2.1.4 cyclicFormValid

The `cyclicFormValid` data member is a boolean variable used as an indicator for the `SymmetricGroup` class. It is set to true when the cyclic form of a permutation is being used and to false when the long form is being used.

2.2 Methods

A number of methods have been defined for the `SymmetricGroup` class. Several different types of Constructor methods are provided for instantiating a `SymmetricGroup` class object. The Group methods are inherited as abstract methods and must be defined. Other support methods for manipulating the `SymmetricGroup` class are provided.

2.2.1 Constructors

Constructors initialize the data members of an object for use. Different constructors can be defined using the parameter list of the method to initialize data. The constructors for the `SymmetricGroup` class are as follows:

- The empty constructor – initializes all object data members to null and primitive data types to their default values.
- The n-letter constructor – initializes the cyclic form to a canonical n-letter solution (e.g. an element of S_4 is instantiated as $(1, 2, 3, 4)$).
- The long form constructor – initializes the long form by using an array argument representing a Symmetric Group element's long form image (i.e. row 2 of Example 1)

- The cyclic form constructor – initializes the cyclic form by using `Cycles` and `int` arguments representing a Symmetric Group element’s cyclic form and the cardinality of the Symmetric Group represented.
- The copy constructor – initializes the object by making a “heavy” copy of the `SymmetricGroup` argument. (A “heavy” copy establishes a new memory location to store the values while a “light” copy simply stores references to the values)

2.2.2 Group methods

The `SymmetricGroup` class inherits a number of methods from the `Group` Class that must be defined. Without these methods, a `SymmetricGroup` class object cannot be instantiated. These methods are:

- `getIdentity`,
- `invert`, and
- `operate`.

getIdentity The `getIdentity` method returns the identity element of the `SymmetricGroup` class. For the long form, this is an n -dimensional array that contains the available letters $\{1, 2, \dots, n\}$ in the order $1, 2, \dots, n$. For the cyclic form, the `Cycles` is initialized to be empty and the `numLetters` is set to n .

invert The `invert` method takes a `SymmetricGroup` class object and inverts its solution. For the long form, this method simply takes the current images and treats them as the indices of the array with the old indices being treated as the images. Example 2 shows how inverse is developed for the long form. The cyclic form is slightly different. For each cycle, the starting letter remains fixed and then order of the remaining letters is reversed. Example 3 shows the cyclic inverse Example 2. Note that the single city tour of (4) is omitted from the cyclic form and its inverse is itself since the first letter of each cycle remains fixed.

Example 2

$$\begin{aligned}
 \text{LongForm} &: \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} \\
 \text{Inverse}(\text{unordered}) &: \begin{pmatrix} 2 & 3 & 1 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} \\
 \text{Inverse}(\text{ordered}) &: \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{pmatrix}
 \end{aligned}$$

Example 3 *Cyclic Form:* (1,2,3) *Inverse:* (1,3,2)

operate The `operate` method takes a `SymmetricGroup class` object and left multiplies it with another `SymmetricGroup class` object. As with the `invert` method, this method performs differently depending on which form each object takes. In the long form, multiplication simply takes the image of each index from the left side object and returns the image of the right side corresponding to the left side image. In Example 4, the left side index 1 has an image of 2 and the right side's image of 2 is 1. Therefore, 1 points to itself, and, in fact, all the indices point to themselves. This is to be expected since the right side is the inverse of the left side thus resulting in the identity element.

Example 4 $\left(\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{array} \right) \otimes \left(\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 3 & 1 & 2 & 4 \end{array} \right) = \left(\begin{array}{cccc} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{array} \right)$

The cyclic form starts with the initial cycle and its first letter. The image of this letter is sought for in the left side and then this letter is located in the right side (if possible). If the letter is found in the right side, the letter's right hand side image is returned. If not found, the image in the left hand side is returned. In Example 5, start with letter 1. Its image is 2 and 2 is not found on the right hand side. Therefore, the image of 1 is 2. Now take the 2 and looks for its image in left hand side which is 3. The image of 3 in the right hand side is 4, so 2 points to 4. Now 4 is not found on the left hand side, so return its image from the right hand side, 1. Thus forming a 3-letter cycle. The letter 3 has an image of 1 on the left hand side and the image of 1 on the right hand side is 3. Thus, 3 points to itself and is not shown in the cyclic representation.

Example 5 $(1, 2, 3) \otimes (1, 3, 4) = (1, 2, 4)$

2.2.3 SymmetricGroup public methods

The `SymmetricGroup class` has a number of methods available to be called by the user. These methods include the following and will be explained in the sections to follow:

Action Methods

- `conjugate`,
- `leftMultiply`, and
- `rightMultiply`.

Action Retrieval Methods

- `getConjugate`,
- `getConjugateSet`,
- `getCyclesHash`,
- `getGeneratorGroup`,

- `getInverse`,
- `getLeftCoset`,
- `getProduct`, and
- `getRightCoset`.

Data Member Access Methods

- `getCyclicForm`,
- `getCyclicFormValid`,
- `getHash`,
- `getLongForm`,
- `getNumLetters`,
- `setCyclicForm`,
- `setCyclicFromLong`,
- `setForm`,
- `setLongForm`, and
- `setLongFromCyclicForm`.

Data Member Support Methods

- `equals`,
- `getValueFromCyclicForm`,
- `hashCode`,
- `indexOfCyclicForm`,
- `indexOfImageCyclicForm`,
- `indexOfImageLongForm`,
- `indexOfLongForm`, and
- `reorderArrayElementsInDictionaryOrder`.

Action Methods The action methods manipulate the values of the given `SymmetricGroup` class object by performing the appropriate actions. Unlike the action retrieval methods, these methods *alter the actual object* rather than returning a new object.

conjugate The `conjugate` method conjugates two `SymmetricGroup` class objects. It alters the object acting as the conjugatee. It is called using the following syntax: `p.conjugate(r)` where `p`, the conjugatee, and `r`, the conjugator, are `SymmetricGroup` class objects.

Definition 6 *Conjugation, usually denoted by p^r , represents the result of the following multiplication: $r^{-1} \otimes p \otimes r$ (or $r^{-1}pr$).*

For the long form, to conjugate `p` by `r`, `r` is inverted, multiplied by `p`, and then this result is multiplied by `r`. Example 4 in the `operate` method describes how multiplication works in the long form and Example 7 shows how conjugation is performed as a series of multiplications. Conjugation in the cyclic form is simpler. It makes use of a theorem that, simply stated, allows each letter in `p` to be replaced by its image in `r`. Example 8 shows how letter 1 in `p` is replaced by its image, letter 2, in `r`, and so on. Since letter 3 in `p` is not found in `r`, it does not change.

Example 7 Let $p = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix}$ and $r = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix}$, then

$$\begin{aligned} p^r &= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix}^{-1} \otimes \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 2 & 3 \end{pmatrix} \end{aligned}$$

Example 8 Let $p = (1, 2, 3)$ and $r = (1, 2, 4)$, then

$$p^r = (1, 2, 3)^{(1,2,4)} = (2, 4, 3)$$

leftMultiply and rightMultiply The `leftMultiply` and `rightMultiply` methods essentially call the same `multiply` method, but pass information through the parameter list in the appropriate order. Both methods perform multiplication as outlined in Section 2.2.2 using either the long or cyclic forms. The methods may be called using the syntax shown in Examples 9 and 10. Notice that the order of `p` and `r` are important since multiplication in the `SymmetricGroup` is not commutative.

Example 9 Let `p` and `r` be the same values as in Example 7, then

$$\begin{aligned} \text{Long Form: } p.\text{leftMultiply}(r) &= r \otimes p \\ &= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{pmatrix} \\ \text{Cyclic Form: } p.\text{leftMultiply}(r) &= r \otimes p = (1, 2, 4) \otimes (1, 2, 3) = (1, 3)(2, 4) \end{aligned}$$

Example 10 Let p and r be the same values as in Example 7, then

$$\begin{aligned} \text{Long Form : } p.\text{rightMultiply}(r) &= p \otimes r \\ &= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 1 & 4 \end{pmatrix} \otimes \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix} \end{aligned}$$

$$\text{Cyclic Form: } p.\text{rightMultiply}(r) = p \otimes r = (1, 2, 3) \otimes (1, 2, 4) = (1, 4) (2, 3)$$

Action Retrieval Methods The action retrieval methods perform a method but place the results of this method into a new `SymmetricGroup` class object rather than altering the `SymmetricGroup` class object calling the method.

getConjugate The `getConjugate` method calls the conjugate method described in Section 2.2.3 and places the result in a new `SymmetricGroup` class object. The method is called using the following syntax: $q = p.\text{getConjugate}(r)$ where p, r and q are `SymmetricGroup` class objects with q containing the result of p^r .

getConjugateSet The `getConjugateSet` method takes two arguments. The form of the two arguments can be one of three configurations:

$$(\text{SymmetricGroup}[], \text{SymmetricGroup}[]) \quad (1)$$

$$(\text{SymmetricGroup}, \text{SymmetricGroup}[]) \quad (2)$$

$$(\text{SymmetricGroup}[], \text{SymmetricGroup}) \quad (3)$$

Notice that 2 and 3 are just special cases of 1. Whichever configuration is used, the method takes every element of the left hand side argument and conjugates it with each and every element of the right hand side argument. The results of conjugating may produce duplicate elements, so the method checks for this and returns an array of `SymmetricGroup` class objects without duplicates. Example 11 shows the results of the `getConjugateSet` method using two small `SymmetricGroup` class object arrays where the objects are in cyclic form. The method is called using the following syntax: $\text{resultSet} = \text{getConjugateSet}(\text{conjugatees}, \text{conjugators})$ where the *conjugatees*, *conjugators* and *resultSet* are `SymmetricGroup` class object arrays.

Example 11

Let $\text{conjugatees} = \{(), (1, 2), (1, 3)\}$ and $\text{conjugators} = \{(2, 3), (1, 2, 3)\}$, then

$$\begin{aligned} ()^{(2,3)} &= () & ()^{(1,2,3)} &= () \\ (1, 2)^{(2,3)} &= (1, 3) & (1, 2)^{(1,2,3)} &= (2, 3) \\ (1, 3)^{(2,3)} &= (1, 2) & (1, 3)^{(1,2,3)} &= (2, 1) \end{aligned}$$

So the following set of solutions would be returned as an array of `SymmetricGroup` class objects

$$\{(), (1, 2), (1, 3), (2, 3)\}$$

getCyclesHash The `getCyclesHash` method simply returns the hash-code value of a cyclic form `SymmetricGroup class` object and in the form of an integer. The method is called using the following syntax: `hashValue = getCyclesHash(cycles, divisions)` where `cycles` represent the cyclic form data member of a `SymmetricGroup class` object, `divisions` is a static integer defining the resolution of the hash and `hashValue` is the actual integer value returned.

getGeneratorGroup The `getGeneratorGroup` method has two calling forms. The first uses a single `SymmetricGroup class` object (the generator) and repeatedly multiplies this object by itself until the identity element is returned, i.e. it generates the cyclic group of the single generator. Example 12 shows the set generated by the cyclic form element $(1, 2, 3, 4)$. The method is called using the following syntax: `resultSet = getGeneratorGroup(generator)` where `generator` is a single `SymmetricGroup class` object and `resultSet` is a `SymmetricGroup class` object array.

Example 12

$$\begin{aligned}
 p^1 &= (1, 2, 3, 4) \\
 p^2 &= (1, 2, 3, 4) \otimes (1, 2, 3, 4) = (1, 3)(2, 4) \\
 p^3 &= (1, 3)(2, 4) \otimes (1, 2, 3, 4) = (1, 4, 3, 2) \\
 p^4 &= (1, 4, 3, 2) \otimes (1, 2, 3, 4) = ()
 \end{aligned}$$

Therefore the generated set is: $\{(), (1, 3)(2, 4), (1, 2, 3, 4), (1, 4, 3, 2)\}$

The second form of this method uses an array of `SymmetricGroup class` objects (the generators). It uses Dimino's algorithm for creating the group associated with a set of generators. Basically, the smallest generator (by cardinality) is selected, and the first form of this method is called using this smallest generator (forming the smallest generator set). Then, the identity element is placed in a list that holds elements waiting to be processed. Each element of the list is right multiplied by the generators. If a new `SymmetricGroup class` object is created, it is added to the list and subsequently left multiplied by the each element of the smallest generator set. All results are placed in a set that checks for and discards duplicates. Example 13 shows the group generated by the generators $\{(1, 2), (1, 2, 3, 4)\}$. This second form is called using the following syntax: `resultSet = getGeneratorGroup(generators)` where `generators` and `resultSet` are `SymmetricGroup class` object arrays.

Example 13

The smallest generator in the set $\{(1, 2), (1, 2, 3)\}$ is the element $(1, 2)$, and its generator set is $\{(), (1, 2)\}$. The waiting list is initialized to $\{()\}$.

$() \otimes (1, 2) = (1, 2)$ (added to list)
 $() \otimes (1, 2) = (1, 2)$ (added to set)
 $(1, 2) \otimes (1, 2) = ()$ (added to set)
 $() \otimes (1, 2, 3) = (1, 2, 3)$ (added to list)
 $() \otimes (1, 2, 3) = (1, 2, 3)$ (added to set)
 $(1, 2) \otimes (1, 2, 3) = (1, 3)$ (added to set, finished with $()$)
 $(1, 2) \otimes (1, 2) = ()$ (a duplicate)
 $(1, 2) \otimes (1, 2, 3) = (1, 3)$ (a duplicate, finished with $(1, 2)$)
 $(1, 2, 3) \otimes (1, 2) = (2, 3)$ (added to list)
 $() \otimes (2, 3) = (2, 3)$ (added to set)
 $(1, 2) \otimes (2, 3) = (1, 3, 2)$ (added to set)
 $(1, 2, 3) \otimes (1, 2, 3) = (1, 3, 2)$ (a duplicate, finished with $(1, 2, 3)$)
 $(2, 3) \otimes (1, 2) = (1, 2, 3)$ (a duplicate)
 $(2, 3) \otimes (1, 2, 3) = (1, 2)$ (a duplicate, finished with $(2, 3)$)
List empty, generated group = $\{(), (1, 2), (1, 3), (2, 3), (1, 2, 3), (1, 3, 2)\}$

getInverse The `getInverse` method returns a `SymmetricGroup` class object that represents the inverse of the `SymmetricGroup` class object calling the method. The method is called using the following syntax: $q = p.\text{getInverse}()$ where p and q are `SymmetricGroup` class objects with q containing the result of p^{-1} .

getLeftCoset and getRightCoset The `getLeftCoset` and `getRightCoset` methods take a single `SymmetricGroup` class object and multiply it with each element of an array of `SymmetricGroup` class objects. **Note:** This method assumes that the array of `SymmetricGroup` class objects forms a subgroup, otherwise this is simply a method for multiplying a single `SymmetricGroup` class object to an array of `SymmetricGroup` class objects. For the `getLeftCoset` (`getRightCoset`) method, the single element left (right) multiplies each element of the array. All results are placed in a set that checks for duplicates, with duplicates being discarded. Examples 14 and 15 show the left and right cosets resulting from using the `SymmetricGroup` class object $p = (1, 2)$ and the `SymmetricGroup` class objects array $q = \{(), (1, 2, 3), (1, 3, 2)\}$. The methods may be called using the following syntax: $resultSet = \text{getLeftCoset}(p, q)$ ($resultSet = \text{getRightCoset}(p, q)$) where p is a single `SymmetricGroup` class object and q and $resultSet$ are `SymmetricGroup` class object arrays.

Example 14

$$\begin{aligned} (1, 2) \otimes () &= (1, 2) \\ (1, 2) \otimes (1, 2, 3) &= (1, 3) \\ (1, 2) \otimes (1, 3, 2) &= (2, 3) \end{aligned}$$

The resulting set returned is $\{(1, 2), (1, 3), (2, 3)\}$

Example 15

$$\begin{aligned} () \otimes (1, 2) &= (1, 2) \\ (1, 2, 3) \otimes (1, 2) &= (2, 3) \\ (1, 3, 2) \otimes (1, 2) &= (1, 3) \end{aligned}$$

The resulting set returned is $\{(1, 2), (1, 3), (2, 3)\}$

This is the same as Example 14,

but, in general, this is not the case.

getProduct The `getProduct` method takes a `SymmetricGroup` class object, right multiplies it with another `SymmetricGroup` class object, and then returns the result as a new `SymmetricGroup` class object. See Example 10 for an illustration of the right multiply method. The method may be called using the following syntax: $result = \text{getProduct}(p, r)$ where p , r and $result$ are `SymmetricGroup` class objects with $result$ containing the result of $p \otimes r$.

Data Member Access Methods The data member access methods simply provide the user with the ability to retrieve information about a `SymmetricGroup` class object without altering the data through *get* calls and to alter the information of a `SymmetricGroup` class object through *set* calls.

getCyclicForm The `getCyclicForm` method returns a copy of the `cyclicForm` data member of a `SymmetricGroup` class object. Example 16 shows the `Cycles` returned when the `getCyclicForm` method is called for the result of the cyclic form right multiplication in Example 10. If the `SymmetricGroup` is not in cyclic form, an empty `Cycles` will be returned. The method is called using the following syntax: $v = p.\text{getCyclicForm}()$ where p is a `SymmetricGroup` class object and v is a `Cycles` containing the cycles of p .

Example 16 Let p and r , be the same values as in Example 8, then

$$\begin{aligned} \text{Cycles } v &= (p \otimes r).\text{getCyclicForm}() \\ &= (1, 4)(2, 3) \end{aligned}$$

getCyclicFormValid The `getCyclicFormValid` method returns a copy of the `cyclicFormValid` data member of a `SymmetricGroup` class object. Example 17 shows the results if the method is called on the two different forms of the Symmetric Group. The method is called using the following syntax: `isCyclic = p.getCyclicFormValid()` where `p` is a `SymmetricGroup` class object and `isCyclic` is a **boolean** containing the current valid form of `p`.

Example 17 Let $p = (1, 2, 3)$ and $q = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$, then

```
boolean is_p_Cyclic = p.getCyclicFormValid()
                    = true
```

```
boolean is_q_Cyclic = q.getCyclicFormValid()
                    = false
```

getHash The `getHash` method simply returns the hashcode value of a `SymmetricGroup` class object in the form of an integer. The method is called using the following syntax: `hashValue = getHash(letters, divisions)` where `letters` is an array representing the order of letters within a `SymmetricGroup` class object, `divisions` is a static integer defining the resolution of the hash, and `hashValue` is the actual integer value returned.

getLongForm The `getLongForm` method returns a copy of the `longForm` data member of a `SymmetricGroup` class object. Example 18 shows the value returned when the `getLongForm` method is called for the result of the long form right multiplication in Example 10. If the Symmetric Group is not in long form, an empty array will be returned. The method is called using the following syntax: `a = p.getLongForm()` where `p` is a `SymmetricGroup` class object and `a` is an integer array containing the longForm array of `p`.

Example 18 Let p and r , be the same values as in Example 7, then

```
int[] a = (p ⊗ r).getLongForm()
          = [ 4  3  2  1 ]
```

getNumLetters The `getNumLetters` method returns a copy of the `numLetters` data member of a `SymmetricGroup` class object. Example 19 shows the value retrieved from the result of the cyclic form right multiplication in Example 10. The method is called using the following syntax: `n = p.getNumLetters()` where `p` is a `SymmetricGroup` class object and `n` is the cardinality of the `SymmetricGroup` class represented.

Example 19 Let p be the same as in Example 12, then

```
int n = p.getNumLetters()
      = 4
```

setCyclicForm The `setCyclicForm` method allows the user to initialize a new `SymmetricGroup` class object to a specified cyclic element. Alternately, it can be used to reset the `cyclicForm` data member of an existing `SymmetricGroup` class object. Example 20 shows how to reset the cyclic form of `p` to that of `r` from Example 8. The method is called using the following syntax: `p.setCyclicForm(sourceCycles, n)` where `p` is a `SymmetricGroup` class object, `sourceCycles` contains the new cyclic structure, and `n` is the order of the Symmetric Group represented.

Example 20 Let $p = (1, 2, 3)$ and $r = (1, 2, 4)$ with $n = 4$, then using

```
p.setCyclicForm(r.getCyclicForm(), r.getNumLetters)
= p.setCyclicForm((1, 2, 4), 4)
```

After the previous statement, the following call returns:

```
Cycles v = p.getCyclicForm()
          = (1, 2, 4)
boolean isCyclic = p.getCyclicFormValid()
          = true
```

setCyclicFromLong The `setCyclicFromLong` method allows the user to input a long form array representation, convert it into a cyclic form representation, change the `cyclicForm` data member of a `SymmetricGroup` class object, and set the `cyclicFormValid` data member value to **true**. Example 21 illustrates the use of this method. The method is called using the following syntax: `p.setCyclicFromLong(sourceArray)` where `p` is a `SymmetricGroup` class object and `sourceArray` is the long form array representation.

Example 21 Let $p = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix}$ and let $r = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix}$, then

```
p.setCyclicFromLong(r.getLongForm())
= p.setCyclicFromLong([ 2 4 3 1 ])
```

First the array `[2 4 3 1]` is transformed into the cycle $(1, 2, 4)$ and then a call similar to Example 20 is made.

After the previous statements, the following calls return:

```
Cycles v = p.getCyclicForm()
          = (1, 2, 4)
boolean isCyclic = p.getCyclicFormValid()
          = true
```

setForm The `setForm` method allows the user to initialize a new `SymmetricGroup` class object into a specified form based on the form of the `SymmetricGroup` class object in the argument list. Alternately, it can be used to reset the form of an existing `SymmetricGroup` class object. The method automatically sets the long or cyclic form based on the `SymmetricGroup` class object parameter. If the parameter is in cyclic form, see Example 20. If it is in long form, see Example 22. The method is called using the following syntax: `p.setForm(q)` where `p, q` are `SymmetricGroup` class objects.

setLongForm The `setLongForm` method allows the user to initialize a new `SymmetricGroup` class object to a specified long form element. Alternately, it can be used to reset the `longForm` data member of an existing `SymmetricGroup` class object. The argument list takes a long form array representation and assigns it to the `longForm` data member of a `SymmetricGroup` class object as well as assigning the `cyclicFormValid` data member value to **false**. Example 22 shows the results of a call to this method. The method is called using the following syntax: `p.setLongForm(sourceArray)` where `p` is a `SymmetricGroup` class object and `sourceArray` is the long form array representation.

Example 22 Let $p = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 2 & 1 \end{pmatrix}$ and let $r = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 4 & 3 & 1 \end{pmatrix}$, then

```
p.setLongForm(r.getLongForm())
= p.setLongForm([ 2 4 3 1 ])
```

After the previous statement, the following calls return:

```
int[] a = p.getLongForm()
      = [ 2 4 3 1 ]
boolean isCyclic = p.getCyclicFormValid()
              = false
```

setLongFromCyclic The `setLongFromCyclicForm` method allows the user to input a cyclic form representation (a `Cycles` and an integer), convert it into a long form representation, change the `longForm` data member of a `SymmetricGroup` class object, and then set the `cyclicFormValid` data member value to **false**. Example 23 illustrates the use of this method. The method is called using the following syntax: `p.setLongFromCyclicForm(sourceCycles, n)` where `p` is a `SymmetricGroup` class object and `sourceCycles` is the cyclic form `Cycles` and `n` is the `Symmetric Group` represented.

Example 23 Let $p = (1, 2, 3)$ and $r = (1, 2, 4)$ with $n = 4$, then

```
p.setLongFromCyclicForm(r.getCyclicForm(), r.getNumLetters())
= p.setLongFromCyclicForm((1, 2, 4), 4)
```

After the previous statement, the following calls return:

```
int[] a = p.getLongForm()
      = [ 2  4  3  1 ]
boolean isCyclic = p.getCyclicFormValid()
      = false
```

Data Member Support Methods The data member support methods provide additional means to retrieve specific information about the Symmetric Group element represented.

equals The `equals` method determines if two `SymmetricGroup` class objects represent the exact same element from S_n . It does so by first comparing the hashcode values (see `hashCode` later in this section) of the two `SymmetricGroup` class objects. If these are equal, then the data members of each object are compared. For long form objects, the length of their arrays are compared. If equal, then a letter by letter comparison of the image order takes place. If the array that stores the images for each object is the same, then the objects are determined to be the same. For cyclic form objects, the number of cycles are first compared. If equal, then each cycle is compared letter by letter. If all cycles are equal, then the objects are determined to be the same. It is called using the following syntax: `isEqual = p.equals(r)` where `p` and `r` are `SymmetricGroup` class objects and `isEqual` is a **boolean** variable.

getValueFromCyclicForm The `getValueFromCyclicForm` support method allows the user to retrieve the letter located at position j within cycle i . Example 24 displays the value returned when asking for the letter at the 1st position of the 2nd cycle of the `Cycles` of a `SymmetricGroup` class object. The method is called using the following syntax: `letter = p.getValueFromCyclicForm(indices)` where `p` is a `SymmetricGroup` class object, `indices` is a two-dimensional integer array with the first index representing the cycle and the second index the position within the cycle, and `letter` is the (integer) value returned.

Example 24 Let *Cycles* $v = (1,4)(2,3)$ be the cyclic form representation of the `SymmetricGroup` class object `p`. Let `indices = [1 0]` be the cycle and position of the letter retrieved.

```
int letter = p.getValueFromCyclicForm(indices)
           = p.getValueFromCyclicForm([ 1 0 ])
           = 2
```

hashCode The `hashCode` method evaluates a `SymmetricGroup` class object and returns an integer value representing the element. This integer value represents a quick way of determining if two `SymmetricGroup` class objects

are different. It is used in the `equals` method as a first cut evaluation. For the `SymmetricGroup` class, the hashcode is determined by the order of the letters stored in the object's data members. For the long form, the `longForm` array represents this ordering. Each value in the array is converted to an integer number using bitwise shifting and then bitwise `^` is applied to the current value of the hashcode. It is possible for two different elements in the Symmetric Group to evaluate to the same hashcode value, but the likelihood of this is fairly low. When two elements do evaluate to the same hashcode value, the `equals` method is used to ascertain whether the elements are same or different. The cyclic form is similar to the long form with the following difference. The cycles of the cyclic form are appended together to form a single array and then the long form approach is applied to this array. The hashcode method is automatically called whenever changes occur to a `SymmetricGroup` class object (including instantiation). The method may also be called using the following syntax: `hashValue = p.hashCode()` where `p` is a `SymmetricGroup` class object and `hashValue` is an `integer` variable.

indexOfCyclicForm The `indexOfCyclicForm` method allows the user to retrieve a two-dimensional array representing the cycle and position within the cycle of a specific letter when the cyclic form is being used. If the letter is not found, it will return -1 for both indices resulting in an error. Example 25 shows the indices array returned when requesting the position of letter 4 in the `SymmetricGroup` class object of Example 24. The method is called using the following syntax: `indices = p.indexOfCyclicForm(letter)` where `p` is a `SymmetricGroup` class object, `letter` is the letter being sought, and `indices` are the cycle and position within the cycle of the letter returned in the form of a 2-dimensional array.

Example 25 Let `p` be the same as in Example 24 and `letter = 4`, then

```
int[] indices = p.indexOfCyclicForm(letter)
              = p.indexOfCyclicForm(4)
              = [ 0  1 ]
```

indexOfImageCyclicForm The `indexOfImageCyclicForm` method allows the user to retrieve a two-dimensional array representing the cycle and position within the cycle of the image of a specific letter when the cyclic form is being used. If the letter is not found, it will return -1 for both indices resulting in an error. Example 26 shows the indices array returned when requesting the position of the image of letter 4 in the `SymmetricGroup` class object of Example 24. The method is called using the following syntax: `indices = p.indexOfImageCyclicForm(letter)` where `p` is a `SymmetricGroup` class object, `letter` is the letter whose image is being sought, and `indices` are the cycle and position within the cycle of the letter returned in the form of a 2-dimensional array.

Example 26 Let p be the same as in Example 24 and $letter = 4$, then

```
int[] indices = p.indexOfImageOfCyclicForm(letter)
              = p.indexOfImageOfCyclicForm(4)
              = [ 0  0 ]
```

indexOfImageLongForm The `indexOfImageLongForm` method allows the user to retrieve an integer representing the index position of the image of a specific letter when the long form is being used. If the letter is not found, it will return -1 resulting in an error. Example 27 shows the index returned when requesting the image of a letter in the `SymmetricGroup` class. The method is called using the following syntax: $index = p.indexOfImageLongForm(letter)$ where p is a `SymmetricGroup` class object, $letter$ is the letter whose image is being sought, and $index$ is the position of the image of the letter.

Example 27 Let $p = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 1 & 3 \end{pmatrix}$ and $letter = 4$ then

```
int index = p.indexOfImageLongForm(letter)
           = p.indexOfImageLongForm(4)
           = 4
```

indexOfLongForm The `indexOfLongForm` method allows the user to retrieve an integer representing the index position of a specific letter when the long form is being used. If the letter is not found, it will return -1 resulting in an error. Example 28 shows the index returned when requesting a letter in the `SymmetricGroup` class. The method is called using the following syntax: $index = p.indexOfLongForm(letter)$ where p is a `SymmetricGroup` class object, $letter$ is the letter being sought, and $index$ is the position of the letter.

Example 28 Let $p = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 2 & 1 & 3 \end{pmatrix}$ and $letter = 4$ then

```
int index = p.indexOfLongForm(letter)
           = p.indexOfLongForm(4)
           = 1
```

reorderArrayElementsInDictionaryOrder The `reorderArrayElementsInDictionaryOrder` method allows the user to input an array and transform it into lexicographic order. The method is automatically called whenever any changes occur to a `SymmetricGroup` class object. This is important for the hashcode of an object to remain consistent, especially in the cyclic form where $(2, 4, 1, 3) = (4, 1, 3, 2) = (1, 3, 2, 4) = (3, 2, 4, 1)$ all represent the same element. Therefore, this element will always be reordered to $(1, 3, 2, 4)$. The method is called using the following syntax: $p.reorderArrayElementsInDictionaryOrder(sourceArray)$

where p is a `SymmetricGroup` class object and `sourceArray` is the array being reordered.

3 Calling the `SymmetricGroup` class

Now that the user has been familiarized with the general `SymmetricGroup` class methods, there are a couple of ways to call the `SymmetricGroup` class. The first way is to simply include a `SymmetricGroup` class object as a data member within another class as in Example 29. The other way is to derive another class by extending the `SymmetricGroup` class as in Example 30. The chief difference between including (or containing) and extending is the level of access to the methods of the `SymmetricGroup` class. In an extension, the methods of the `SymmetricGroup` class would simply be inherited by the new class and unique methods to the derived class could then be added to these inherited methods. In containment, the `SymmetricGroup` class object would be embedded within the data membership of a class. Thus, all the methods of the `SymmetricGroup` class would be accessed through this data member while allowing the class containing the `SymmetricGroup` class object to be derived from another class.

Example 29 *The following statements show the `SymmetricGroup` contained within a class:*

```
public AMCSolution extends TSSolution {
    SymmetricGroup tankerAssignments;
    ...}
```

Example 30 *The following statement derives a class by extending another class:*

```
public AlternatingGroup extends SymmetricGroup {
    ...}
```

Example 29 demonstrates the form that the author anticipates to be the most prevalent use of the `SymmetricGroup` class and the form that the remainder of this manual will use.

4 Using the `SymmetricGroup` class

The Symmetric Group provides a natural means of describing the solutions of partitioning and ordering problems. The cycles associated with an element of the Symmetric Group represent the partitioning of letters and the arrangement of letters within the cycles represents the ordering. Since most combinatorial problems can be reduced to a partitioning, an ordering, or a partitioning and ordering problem, the Symmetric Group's elements can be used to represent specific solutions to these problems. This section will develop the Symmetric

Group with respect to the combinatorial problem known as the Vehicle Routing Problem and serve as an example for applying the `SymmetricGroup` class to other combinatorial problems.

4.1 The Vehicle Routing Problem

The Vehicle Routing Problem (VRP) consists of a fleet of vehicles with limited capacity that must service a set of customers. In servicing the customers, the most efficient means (fewest vehicles used, least distance traveled, and so on) is sought while meeting each customer's demands. These demands can include quantities of goods and timeliness. Since the demands of a customer along with the capacity of the vehicles serve as constraints, they do not change the essence of this problem. The VRP is a partitioning and ordering problem where the partitions represent an individual vehicle's tour and the arrangement of letters within the partition represent the order in which the vehicle services the customers it has been assigned. Thus, the Symmetric Group will be used to represent solutions to the VRP. A traditional Integer Program formulation is provided in Section 4.1.1. At the end of Section 4.1.1, the Symmetric Group representation will be related to the variables within the traditional VRP formulation. However, the VRP has been shown to be very difficult to solve by traditional means. Typically, some form of heuristic solution approach is used. In this case, the metaheuristic known as Tabu Search (TS) will be described in Section 4.1.2. Following this section will be the specific `SymmetricGroup` class implementation to the VRP.

4.1.1 VRP Formulation

The VRP can be formulated as a Mixed Integer Program (MIP). A typical MIP formulation contains the following sets

$$\begin{aligned}
 M &\equiv \{1, 2, \dots, m\} && \text{the set of depot nodes} \\
 C &\equiv \{m + 1, m + 2, \dots, n\} && \text{the set of customer nodes} \\
 N &\equiv \{1, 2, \dots, n\} && \text{the set of all nodes}
 \end{aligned}$$

and variables

$$x_{ij}^v \equiv \begin{cases} 1 & \text{if vehicle } v \text{ uses arc } (i, j) \\ 0 & \text{otherwise} \end{cases}$$

and parameters

c_{ij} \equiv the cost of travelling from node i to node j

t_{ij}^v \equiv the travel time for vehicle v from node i to node j

s_i^v \equiv the time required for vehicle v to deliver or service at node i

K_v \equiv the capacity of vehicle v

d_i \equiv the demand of customer j

V_i \equiv the number of vehicles available for use at node i

T_v \equiv the maximum time length for a route

with objective

$$\text{Minimize } \sum_{i=1}^n \sum_{j=1}^n \sum_{v=1}^w c_{ij} x_{ij}^v \quad (4)$$

with constraints

$$\sum_{i=1}^n \sum_{v=1}^w x_{ij}^v = 1 \quad \forall j \in (2, \dots, n) \quad (5)$$

$$\sum_{j=1}^n \sum_{v=1}^w x_{ij}^v = 1 \quad \forall i \in (2, \dots, n) \quad (6)$$

$$\sum_{i=1}^n x_{ip}^v - \sum_{j=1}^n x_{pj}^v = 0 \quad \forall p \in (1, \dots, n), v \in (1, \dots, w) \quad (7)$$

$$\sum_{i=1}^n d_i \left(\sum_{j=1}^n x_{ij}^v \right) \leq K_v \quad \forall v \in (1, \dots, w) \quad (8)$$

$$\sum_{i=1}^n s_i^v \left(\sum_{j=1}^n x_{ij}^v \right) + \sum_{i=1}^n \sum_{j=1}^n t_{ij}^v x_{ij}^v \leq T_v \quad \forall v \in (1, \dots, w) \quad (9)$$

$$\sum_{j=m+1}^n x_{ij}^v \leq 1 \quad \forall v \in (1, \dots, w), i \in (1, \dots, m) \quad (10)$$

$$\sum_{i=1}^m x_{ij}^v \leq 1 \quad \forall v \in (1, \dots, w), j \in (m+1, \dots, n) \quad (11)$$

$$\sum_{j=m+1}^n \sum_{v=1}^w x_{ij}^v \leq V_i \quad \forall i \in (1, \dots, m) \quad (12)$$

These constraints alone do not address the possibility of subtours (see Figure for an illustration of a subtour). To eliminate these subtours, use the following

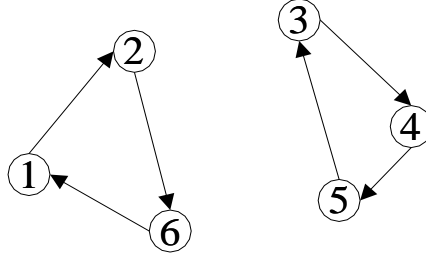


Figure 1: SubTour Example

notation:

$N^v \equiv$ the nodes from N assigned to vehicle $v \ni N^v \subseteq N$

and

$$\begin{aligned}
 N^v &= \emptyset && \text{for any vehicle } v \text{ not used,} \\
 \bigcup_{i=1}^w N^v &= N && \text{vehicles used cover all nodes,} \\
 \bigcap_{i=1}^w N^v &= \emptyset && \text{and vehicles nodes assignments are disjoint.}
 \end{aligned}$$

Using this notation, the subtour elimination constraints are included in the formulation as

$$\sum_{i \in Q} \sum_{j \in Q} x_{ij}^v \geq 1 \quad \text{for every nonempty subset } Q \text{ of } N^v \forall v \in (1, \dots, w) \quad (13)$$

An example of a subtour is shown in Figure 1 where the arcs $(3, 4)$, $(4, 5)$, and $(5, 3)$ represent a subtour. The constraints generated by equation 13 eliminate this possibility

This provides the necessary constraints for the multi-depot general vehicle routing problem. To add in time window considerations, the following parameters are used:

$$\begin{aligned}
 a_j &\equiv \text{arrival time to node } j \\
 e_j &\equiv \text{the earliest delivery time allowed at node } j \\
 l_j &\equiv \text{the latest acceptable delivery time allowed at node } j
 \end{aligned}$$

and are calculated as

$$a_j \geq (a_i + s_i^v + t_{ij}^v) - (1 - x_{ij}^v) T_v \quad \forall i, j, v \quad (14)$$

$$a_j \leq (a_i + s_i^v + t_{ij}^v) + (1 - x_{ij}^v) T_v \quad \forall i, j, v \quad (15)$$

with

$$\begin{aligned} a_i &= 0 & \forall i \in (1, \dots, m) \\ e_j \leq a_j \leq l_j & & \forall j \in (m + 1, \dots, n) \end{aligned}$$

This formulation has many more variants representing real-world problems. Finding the best solution to this problem is very difficult, and, in some variants (i.e. time windows), simply finding a feasible solution is hard. Because of this fact, Section 4.1.2 describes a very successful heuristic approach to solving the VRP known as Tabu Search (TS).

A solution to the formulation of this section can be represented by an element of the Symmetric Group. Each vehicle of the problem is represented by a cycle of the Symmetric Group. If a vehicle is assigned a customer, then the customer is placed in the vehicle's cycle which is equivalent to letting the variable $x_{ij}^v = 1$ where $i \in (1, \dots, m)$, $j \in (m + 1, \dots, n)$, and $v \in (1, \dots, w)$. The order in which customers appear within the vehicle's cycle represents the variable $x_{ij}^v = 1$ where $i \in (m + 1, \dots, n)$, $j \in (m + 1, \dots, n)$, and $v \in (1, \dots, w)$.

4.1.2 TabuSearch

TS is a metaheuristic search technique used to explore the solution space around an incumbent solution. The basic idea of TS is to avoid the local optima by maintaining a list of solutions recently visited and not allowing these solutions to be revisited by some local search heuristic. Figure 2 gives an illustration of this effect. Normally, a local search heuristic would be captured at one of the local optima shown in Figure 2. TS does not allow the local search heuristic to select a move that returns to a solution that has been recently visited. In other words, tabu search forces the local search heuristic to select other, possibly disimproving, moves which eventually allows the search to get over the hump and escape local optima.

More advanced implementations of TS allow for the tenure of the tabu list to be dynamically updated and for more drastic changes in the search procedure to take place. One drastic change involves using a diversification scheme. A diversification scheme allows the user to "restart" the solver at a new solution drastically different from the solutions previously seen in order to search a different part of the solution space. Figure 3 illustrates a solution space where feasible regions of the space are disjoint and reaching solutions in each feasible region might require a diversification scheme.

This section was not intended to be an extensive overview of TS, rather to give the reader the general idea about the TS approach to solving problems. For more details, the reader may refer to *Tabu Search* by Fred Glover and Manuel Laguna.

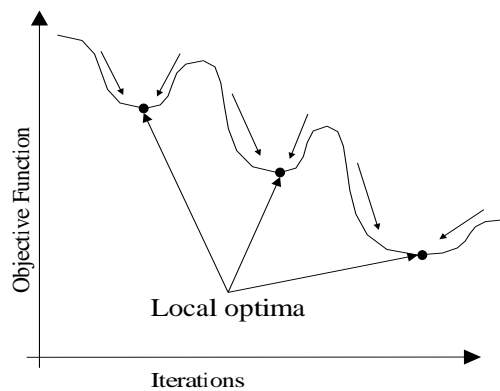


Figure 2: Tabu Search Illustration

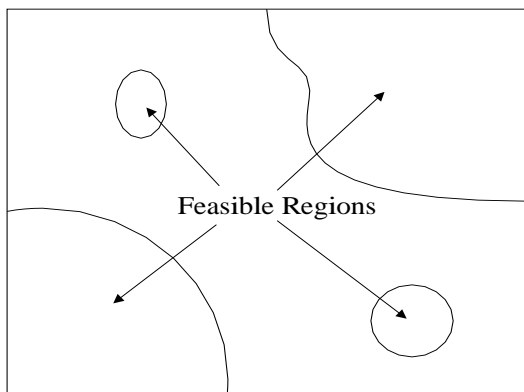


Figure 3: Tabu Search Solution Space

4.1.3 Tabu Search and the `SymmetricGroup` class

At the end of Section 4.1.1, the relationship between the variables of the classical VRP formulation and the Symmetric Group was described. In addition to this solution relationship, the Symmetric Group also has other relationships with TS. As mentioned in the previous section, Tabu Search chooses amongst moves from one solution to another seeking improving solutions. The moves that TS chooses from form a neighborhood. Depending on the type of move being considered (swapping two customers order, moving a customer from one vehicle route to another, and so on), various types of neighborhoods can be created. In turn, these neighborhoods can be efficiently represented and built using the Symmetric Group (Barnes & Colletti 1999).

The `SymmetricGroup` class described at the beginning of this paper was intentionally coded in Java. This was done in order to take advantage of the Java-based TS framework written by Robert Harder (available for download at <http://www.iharder.net>). Harder’s framework provides a basic TS “engine” to build upon and will be used in the following sections as building blocks for the interaction of TS and the `SymmetricGroup` class.

Tabu Search Solutions using the `SymmetricGroup` class The `SymmetricGroup` class describes the solution to a VRP but must be placed in a TS solution format. Harder’s TS code requires that the solution be derived from his base `TSSolution` class. Since the Java language does not allow classes to be derived from more than one class, the Symmetric Group has been placed inside another class with a statement similar to Example 29. By containing, the `SymmetricGroup` class object within another class, the TS solution retains all of the methods described in Section 2.2 in addition to the methods and data members inherited from the `TSSolution` class. In fact, the only additional requirement placed on a TS solution using Harder’s code is that the method `clone()` be defined for the derived class. An example of meeting this requirement can be found in the code of Example 31. This example defines the method `clone` by calling a copy constructor similar to the one described in Section 2.2.1.

Example 31

```
public Object clone() {  
    return new AMCSolution(this); }  
}
```

Tabu Search Moves using the `SymmetricGroup` class As mentioned in Section 4.1.3, the Symmetric Group may be used to represent neighborhoods of the current solution. These neighborhoods are traditionally represented as the solutions reachable in one move from the current solution using a particular move discipline. Another way of viewing the neighborhoods focuses on the moves themselves rather than the solutions. In this view, the neighborhood consists of the moves that transform one solution into another solution rather than the solutions themselves. Example 32 provides a traditional neighborhood

view and Example 33 provides a moves neighborhood view for the operation conjugation on the cyclic form representation of the Symmetric Group. I.e., given $p^r = q$ where p is the incumbent solution, r is the move, and q is the neighbor, a traditional neighborhood approach will store all neighbors q and the move neighborhood view will store all moves r .

Example 32 *Let $p = (1, 2, 3, 4, 5)$ and assume that the move discipline allows swaps between adjacent letters. The traditional neighborhood of p is:*

$$\{(2, 1, 3, 4, 5), (1, 3, 2, 4, 5), (1, 2, 4, 3, 5), (1, 2, 3, 5, 4), (5, 2, 3, 4, 1)\}$$

Example 33 *Let $p = (1, 2, 3, 4, 5)$ and assume that the move discipline allows swaps between adjacent letters. The conjugation moves neighborhood of p is:*

$$\{(1, 2), (2, 3), (3, 4), (4, 5), (5, 1)\}$$

There are advantages to each approach. The traditional view allows the user to know exactly what the solution will be when a neighbor is chosen while the moves view requires the operation be performed before the new solution is known. However, the moves view requires storing only knowledge of the letters being moved while the traditional view requires storing the entire solution of each neighbor. Also, the move neighborhood lends itself to efficient evaluation of the moves in determining the best move in the local search routine (Colletti & Barnes 2000).

As the previous two examples showed, the Symmetric Group easily can represent the solutions or moves of a given neighborhood. To capture this information within Harder's TS implementation, the `SymmetricGroup` class must either implement Harder's `TSMove` class or be contained within another class that implements `TSMove`. Examples 34 and 35 illustrate the two means of implementing Harder's `TSMove` class.

Example 34 *The following statements show the `SymmetricGroup` contained within a class:*

```
public AMCMove implements TSMove {
    SymmetricGroup tankerMoves;
    ....}
```

Example 35 *The following statement shows how to implement `TSMove` using the `SymmetricGroup`:*

```
public SymmetricGroup implements TSMove {
    ... }
```

The implementation of `TSMove` requires the definition of the method `operateOn`, otherwise Java will not allow instantiation of objects from this class. To maintain consistency with the derived class of the previous section, the form of

Example 34 is chosen. The purpose of the `operateOn` method is to apply a move to a solution to create a new solution. The `SymmetricGroup` class offers several ways of doing this. The `conjugate` and `multiply` (left and right) methods previously defined provide the means for applying a move to a solution. Example 33 showed an example of the moves associated with defining the `operateOn` method using conjugation. Example 36 provides an example of the source code for the method `operateOn` where `soln` acts as the conjugatee and `tankerMoves` is the conjugator in the call (see Section 2.2.3).

Example 36

```
public void operateOn (TSSolution solution) {
    AMCSolution soln = (AMCSolution) solution;
    soln.getTankerAssignments().conjugate(tankerMoves) }
```

Tabu Search Neighborhoods and the `SymmetricGroup` class Now that the form of the moves and its operation are known, a means to generate the moves that form a TS move neighborhood must be defined. Harder’s TS code requires the implementation of the `TSMoveManager` class to generate the necessary neighborhood. The specific call to implement the `TSMoveManager` is shown in the Example 37.

Example 37 *The following statement shows how to implement `TSMoveManager`:*

```
public AMCNeighborhood implements TSMoveManager {
    ... }
```

This implementation requires the definition of the method `getAllMoves`. This `getAllMoves` method then passes the moves within the neighborhood to the “engine” for evaluation. The specific call within the `getAllMoves` method would look like the following example.

Example 38

```
public TSMove[] getAllMoves(TSSolution solution) {
    AMCSolution soln = (AMCSolution) solution;
    SymmetricGroup currentAssignments = soln.getTankerAssignments();
    SymmetricGroup[] moves = twoLetterMoves(currentAssignments);
    AMCMove[] newMoves = new AMCMove[moves.length];
    for (int i = 0; i < moves.length; i++) {
        newMoves[i] = new AMCMove(moves[i]); }
    return newMoves; }
```

For the `SymmetricGroup` class under the operation of conjugation, a method that defines the moves of a neighborhood needs to be defined and then called by the `getAllMoves` method (as in `twoLetterMoves` of Example 38). An example of a move neighborhood generation method (`twoLetterMoves`) can be seen in Example 39. This example shows how to create an two-letter rearrangement move neighborhood for use with conjugation.

Example 39

```
private static SymmetricGroup[] twoLetterSwap(SymmetricGroup incumbent) {
    int[] oneBigArray = makeOneBigArray(incumbent);
    int num_Letters = incumbent.getNumLetters();
    Cycles twoLetterMoves = new Cycles(num_Letters)
    for (int i = 0; i < num_Letters; i++)
        for (int j = i + 1; j < i + num_Letters; j++) {
            Cycles tempCycles2 = new Cycles(1);
            int k = j % num_Letters;
            int[] tempArray = new int[2];
            if (oneBigArray[i] < oneBigArray[k]) {
                tempArray[0] = oneBigArray[i];
                tempArray[1] = oneBigArray[k]; }
            else {
                tempArray[0] = oneBigArray[k];
                tempArray[1] = oneBigArray[i]; }
            tempCycles2.add(tempArray);
            twoLetterMoves.addElement(
                new SymmetricGroup(tempCycles2, num_Letters)); }
    twoLetterMoves.trimToSize();
    SymmetricGroup[] twoLetterMovesArray =
        new SymmetricGroup[twoLetterMoves.size()];
    twoLetterMoves.copyInto(twoLetterMovesArray);
    return twoLetterMovesArray;
}
```

Tabu Search Objective Function and the *SymmetricGroup* class The neighborhoods generated by the move manager provide an array of *SymmetricGroup* class objects that need to be evaluated by the TS “engine”. The “engine” makes use of the interface *TSFunction*. This interface must be implemented by the TS program using a statement similar to Example 40.

Example 40 *The following statement shows how to implement *TSFunction*:*

```
public AMCObjective implements TSFunction {
    ... }
```

Within the *TSFunction*, the only restriction it enforces is that the method *evaluate* be defined. The *evaluate* method serves two roles. First, it needs to evaluate the initial solution passed in to the “engine” and, second, it needs to be able to evaluate the effect of a move on a current solution. The first role typically plays the part of doing a full evaluation of the solution (i.e. determining the initial route length of a starting solution). The second role provides the means to efficiently determine the effect of a move on a solution without having to perform a full evaluation (i.e. only the arcs that have been changed by the move need their values recomputed). The *SymmetricGroup* class provides an efficient

method for computing moves values using a move set (Colletti & Barnes 2000). Example 41 provides an illustration of an `evaluate` method code.

Example 41

```
public double[] evaluate(TSSolution solution, TSMove move) {
    if (move != null) {
        AMCMove mv = (AMCMove) move;
        AMCSolution soln = (AMCSolution) solution;
        double[] delta = mv.moveEvaluate(soln, costMatrix);
        return new double[] {delta[0] + soln.getObjectiveValue()[0]}; }
    else {
        AMCSolution soln = (AMCSolution) solution;
        SymmetricGroup solnOrder = soln.getTankerAssignments();
        double distance = 0;
        for (int i = 0; i < solnOrder.getCyclicForm().size(); i++) {
            int[] tempArray =
                (int[]) solnOrder.getCyclicForm().elementAt(i);
            for (int j = 0; j < tempArray.length; j++)
                distance += costMatrix[tempArray[j]]
                    [tempArray[(j + 1) % tempArray.length]]; }
        return new double[] {distance}; }
```

The efficient move evaluation of the `SymmetricGroup` class comes in the call to the method `moveEvaluate`. In this method, a call to another method that determines the arcs broken and replaced by the operation of the move upon the solution is made. The results of this method are then passed into another method that performs the actual computation. Source code for the computation method is shown in Example 42.

Example 42

```
private static double[] evaluateCyclicForm(SymmetricGroup currentSolution,
    SymmetricGroup currentMove, SymmetricGroup indexMoved,
    double[][] costMatrix) {
    SymmetricGroup currentMoveInverse = currentMove.getInverse();
    double delta = 0;
    for (int i = 0; i < indexMoved.getCyclicForm().size(); i++) {
        int[] tempArray =
            (int[]) indexMoved.getCyclicForm().elementAt(i);
        for (int j = 0; j < tempArray.length; j++) {
            int valueOfImageValueOfSolutionResult = -1;
            int[] indexOfImageValueOfOperand =
                currentSolution.indexOfImageCyclicForm(tempArray[j]);
```

Tabu Search Tabu List Structure and the `SymmetricGroup` class The final piece to using TS and the `SymmetricGroup` class is to define the formal

structure used for determining when a move is allowed. Harder’s code provides this structure through the `TSTabuList` class. This interface class is implemented as in Example 43 and requires the definition of two methods—`allowMove` and `registerMove`.

Example 43 *The following statement shows how to implement TSTabuList:*

```
public AMCTabuList implements TSTabuList {
    ... }
```

In the `allowMove` method, the `SymmetricGroup` class object defining a move, e.g. (1,2) from Example 33, is viewed along with the incumbent solution to determine if that move is allowed (not tabu). There are many types of tabu restrictions that can be implemented within the `allowMove` method. An example of one such restriction can be seen in the following code of Example 44. This restriction allows a move when it is not the inverse of the last move and when it has not been previously selected with the incumbent solution over the tabu tenure.

Example 44

```
public boolean allowMove(TSMove move, TSSolution solution) {
    AMCSolution soln = (AMCSolution) solution;
    AMCMove mv = (AMCMove) move;
    if (!solutionAllow(soln)) {
        for (int i = 0; i < solutionLinkedList.size(); i++) {
            AMCSolution tempSolution =
                (AMCSolution) solutionLinkedList.get(i);
            if (soln.equals(tempSolution)) {
                AMCMove moveAtTempIndex =
                    (AMCMove) moveLinkedList.get(i);
                if (moveAtTempIndex.equals(mv)) return false; } }
        AMCMove lastMove = new AMCMove(
            (AMCMove) moveLinkedList.getLast());
        lastMove.getMoveOrder().invert();
        if (lastMove.equals(mv)) return false; }
    else if (solutionLinkedList.size() > 0) {
        AMCMove lastMove = new AMCMove(
            (AMCMove) moveLinkedList.getLast());
        lastMove.getMoveOrder().invert();
        if (lastMove.equals(mv)) return false; } }
```

The other required method, `registerMove`, is called at the end of an iteration after a move has been selected from the move neighborhood. The purpose of this method is to store the moves and solutions that are considered tabu (or not allowed) over a specific tabu tenure. The form of this storage is left up to the user, but an example call to this method is provided in Example 45. This

example shows that the solution and moves are registered using specific calls for each which allows different structures to be used for each (if desired).

Example 45

```
public void registerMove(TSMove move, TSSolution solution) {
    AMCSolution soln = (AMCSolution) solution;
    solutionRegister(soln);
    AMCMove mv = (AMCMove) move;
    moveRegister(mv); }
```

References

- Barnes, J. W. & Colletti, B. W. (1999). Group theory and metaheuristic search neighborhoods, *Technical Report ORP99-02*, The University of Texas at Austin.
- Colletti, B. & Barnes, J. W. (2000). Using group theory to efficiently compute move values for routing problems, *Applied Mathematics Letters* **TBD**: TBD. to be published.
- Wiley, V. D. (2000). A step-by-step user's guide to implementating the group class in java, *Technical report*, The University of Texas at Austin.