V. Program Descriptions

In this chapter we describe each of our programs. They are grouped according to use as in Table 4.1.

A. General Subroutines

CmprssBiList[] is used within other programs to combine elements in a list of pairs. Its arguments are a list of pairs, the number 1 or 2, and a function. If the second argument is 1, the program finds the list of distinct elements appearing as the first element in a pair. For each element in this list, a new pair is created. The first element of this pair is just the current element. For the second element, the program finds all of the original pairs that have the current element as their first element. Then the second element of the new pair is found by combining the second entries in these pairs according to the input function. The result is a list of pairs with distinct first elements. If the second argument of CmprssBiList[] is a 2, the roles of first and second elements are reversed.

NoSpcs[] prepares the input list-of-lists-of-lists-…, nested to any level, to be written to a file. If a nested list-of-lists is written to a file without this function, there will be spaces inserted within the inner levels. To prevent this, NoSpcs[] converts the input list to String format, making sure not to include any inner spacing. This string can then be written to a file without any embedded spaces.

WriteLstOfLst[] implements NoSpcs[] to write to a file. Its arguments are a file and a list-of-lists. The program applies NoSpcs[] to each element of the list before writing it to the given file. The resulting file will contain one element of the input list per line.

PrntCnsmptn[] has six arguments. The first two deal with memory and are in bytes. The first is a starting memory while the second is a measurement of memory taken some time later. The second two arguments are measurements of real time in seconds. The first of these is a starting time while the second is time after some code has been executed. The last two arguments are also in seconds but deal with time used by the CPU. The first of these is an initial measure of CPU time while the second is a measurement taken after something has been executed. Given these arguments, the program prints the change in memory, real time, and CPU time.

B. Poset Subroutines

For **Ideel[]**, the arguments are a poset and a set of generators. The smallest ideal containing these generators is returned. The program acts recursively by looking at the set of children for each generator and finding the ideal of the poset that is generated by each of these sets. The final step is to Union this result with the original generators. This also removes multiplicities and sorts. The desired ideal is returned with its elements listed in increasing order.

AntiChnsIdls[] runs through each of the elements of the given poset using the index ig. Suppose that the program has run through all elements less than ig. Then antchns contains all antichains that contain only elements smaller than ig and thrdscnds contains their corresponding ideals. Now for ig, this program runs through each element of antchns and tests two conditions to see if antchns and thrdscnds will be updated. For the jp-th antichain, we first test that ig is not in the ideal that corresponds to this antichain. The second test checks that the antichain and the ideal generated by ig do not overlap. If both conditions are satisfied then ig can be appended to this antichain and this is added to the list of antichains. At this point, the ideal generated by ig is combined with the jp-th ideal and the result is added to the list of ideals. The ordered pair that is returned consists of the list of all antichains and the list of all of their corresponding ideals.

For **PllOff[]**, the arguments are a poset in labeled child form and a list of elements in an ideal. It first removes from the poset the {child list, element} pairs for the elements in the ideal. Then for the remaining elements of the poset, all the elements of the ideal are removed from each of their child lists. These updated child lists then replace the original child lists in the poset and this poset is returned.

For **Relabl[]**, we consider the given poset to be labeled in black. The given inverse extension is a yellow-to-black transformation rule. For each of the black elements, the yellow labels for its children are found according to the inverse of this rule and then sorted into increasing order. By mapping these child lists onto the inverse extension, the child lists are put into the correct yellow order. The result is a poset labeled in yellow. The original poset can be obtained by relabeling this yellow-labeled poset according to the given rule.

NewChdLst[] provides a step of the StdFormIso[] program. It is given a poset that has black labels and a yellow-indexed list of some of its elements. The program finds the child list in the poset for the last element in this list. Each element in this black child list is then replaced by its yellow index. Sorting guarantees that these yellow children will be listed in increasing order. The result is the yellow child list of the last element in the given list.

For **ShrnkCnvxSt[]**, a poset as well as a list of elements in a convex subset of that poset is input. Each element of the convex subset is replaced by its child list for the given poset. Then for each of these child lists, the elements that are not members of the convex subset are removed. This gives a list of the lists of children of elements of the subset that lie within this subset. Replacing the original labels in this list with consecutive labels that start at 1 gives the child form of the convex subset.

InvStdTblx[] works in conjunction with ShrnkCnvxSt[]. It takes a poset, a list of elements in a convex subset of that poset and a program that finds inverse order extensions. The poset is considered to be labeled in black. The program uses ShrnkCnvxSt[] to find the child form of the given convex subset. This child form will have different labels; we'll say these are brown. Then all of the inverse extensions of this new brown-labeled poset are found using the specified program. Finally the list of elements in the convex subset is used to change from the brown labels to the original black labels in all of the inverse extensions. The result is the list of inverse extensions for the subset with respect to the black labeling.

C. Inverse Extensions and Isomorphism Functions

BasInvExts[] generates all inverse order extensions of the given poset. If we consider the poset to be labeled black, the program returns the list of all yellow-to-black rules for relabeling the poset in yellow. The program finds these by pulling off one minimal element at a time in all possible ways until only one element of the poset remains. Observe that each time a minimal element is removed, the result is a filter of the original poset. First the labeled child form for the poset is found. Writing the poset in this way allows the program to keep track of each element's children after other elements have been pulled off. Suppose that yel-1 elements have been removed. Then in the yel iteration, pairlist is a list of pairs where the first element in each pair is a list of partial pull-off sequences formed so far and the second is the corresponding remaining filter. For each filter in this list, the minimal elements are determined. For each minimal element, we form an ordered pair. The first member of the pair is the list of updated pull-off sequences; it is found by adding the minimal element to the partial black pull-off sequences that correspond with its filter. The second member is the filter found when this minimal element is pulled off. The set of all of these pairs is then stored in pairlist. Now it is possible that at a given stage of this iteration, there are elements in pairlist that have the same remaining filter. So every few iterations or if pairlist gets too long, we combine the pairs that have the same filter into one pair whose first element contains all the corresponding pull-off sequences. Once all but one element has been pulled off, the loop ends. Now the second element in each pair of pairlist is a set containing a single element. Each of these elements is added to its corresponding pull-off sequence. The lists of pull-off sequences are then merged into one and this list is returned.

InvExtsWRI[] also finds all inverse order extensions of the input poset. The program finds these differently depending on the size of the poset. If the size of the poset is at least eight, the program finds the extensions recursively. The poset is split approximately in half into an ideal and filter in all possible ways. Using itself as one of the arguments of InvStdTblx[], for each way the program finds the inverse order extensions for the ideal and filter. These are combined in all possible ways. These lists are then combined to give all the inverse order extensions of the poset. If the size of the poset is between four and seven, the program finds the inverse order extensions by accessing the information in the files stdisos*, lookups*, invexts* where * is the size of the poset. It does this by first getting the rule for putting the poset in standard form. The inverse extensions for its standard form are obtained from invexts*. Transforming these using the inverse relabeling rule gives the inverse order extensions of the given poset. If the size is less than four, the inverse order extensions are determined by considering each case separately based on the child lists of the poset. Once this is determined, the program returns the listed inverse order extensions for that case.

For the specified poset, **StdFormIso[]** returns an ordered pair containing the standard form of the poset and a rule for relabeling the poset to obtain this standard form. If the poset is labeled black, the program considers which of the yellow relabelings give the purple standard form. The program proceeds similar to BasInvExts[] by iteratively pulling off one minimal element at a time. Suppose that yel-1 elements have been removed. Then pairlst is a list of pairs where the first element in each pair is a list of partial pull-off sequences and the second is the corresponding remaining filter. For each filter in this list, the minimal elements are determined. For each of these elements, the program creates a new pair. The first element of the pair is found by adding the minimal element to each of the partial pull-off sequences. The second element is found by removing the minimal element from the current filter. All of these pairs are then stored in pairlst. Based on the partial pull-off sequences, the program determines what the yel-th child list of the new form would be: For each of these sequences, the yellow child list for the last element in the sequence is found and the smallest one becomes the next (purple) child list in the standard form. The pull-off sequences that don't give the earliest child list are then removed from the pairs in pairlst. Then the pairs whose remaining filter has no contending pull-off sequences are removed from pairlst. If pairlst gets too long, the program then combines pairs that have the same remaining filter. When all the elements have been removed from the poset, purplpst is the standard form and the first pull-off sequence in pairlst gives an inverse isomorphism from the poset to this form.

D. Poset Generation Main Programs

For the given size n > 0, **ThrghBldUp.nb** creates five files, each in a different subdirectory within the specified directory. In order for the program to be run for any n larger than one, it must have previously been run for n-1. First the program creates the list of all naturally labeled posets of size n. When n > 1, this is accomplished by reading in the naturally labeled posets of size n-1 and finding the antichains for each of these posets. Then for each antichain, dropping an edge from a new element to each element in the antichain creates a poset of size n. The resulting list is sorted to create an ordering of the posets. When the given size is 1, the only naturally labeled poset is {{}}. To find the standard forms of the posets, this list of all naturally labeled posets is stored in inputlist. At a given stage of the program, the first element in inputlist is added to the list of standard forms. Then all inverse order extensions of this element are found, as are all naturally labeled posets that can be produced from this poset. These alternate posets are removed from inputlist. Removing these guarantees that the new first element of inputlist is not a relabeling of any of the previous elements in the list of standard forms. After verifying that all of the alternate forms were in the list of naturally labeled posets, the newest inverse extensions are added to the list of inverse extensions and the alternate posets are added to the list of alternates. However, if all of the alternate forms were not in the list, it must be that not all naturally labeled posets were generated. In this case, the program quits after alerting the user of the problem. When all the posets have been removed from inputlist, all standard forms have been found and the loop ends and the desired information is written to the files NatLabs/natlabs*, ThrStds/thrstds*, InvExts/invexts*, StdIsos/stdisos*, and Lookups/lookups*, where * is the input size. For a description of the contents of these files, see the Data Files Description Chapter. Finally the consumption of time and memory is displayed.

QckBldUp.nb creates the file stdpsts* in the subdirectory StdPsts of the specified directory. For * = n, this file contains the standard posets for the input size n > 0. For n > 1, the stdpsts file for n-1 is needed. Note that ThrghBldUp.nb as well as QckBldUp.nb can be used to create this type of file, although a different subdirectory and file name would need to be specified. When the given size is 1, the list of standard posets is { {{}} }. Now for n > 1, to obtain posets of size n, all the standard posets of size n-1 are read in and the antichains for each of these posets is found. By dropping an edge from a new element to each element in each antichain, posets of size n are created. Since these posets may not be in standard form, the standard form for each of these posets is then found. After sorting and removing redundant posets, the list of standard forms is written to the file. The consumption of time and memory is then displayed.

Remark: As mentioned above, ThrghBldUp.nb's self-checking aspects guarantees that all naturally labeled posets of a given size are generated. Therefore when the list of these posets is sorted and the first element is chosen, we know that the unique standard form for each isomorphism class has been correctly identified. Then successfully comparing the file stdpsts* to thrstds* checks that the program QckStds.nb also gives the correct standard forms. The results from QckStds.nb depend on what is returned from StdFormIso[]. Thus comparing these files allows us to check that StdFormIso[] really returns the standard form of a poset despite the fact that this function does not compute all of the poset's alternate forms.

For an input size n >0, **BldUMxmls.nb** creates the file umaxmls*, where * = n, in the subdirectory UMaxmls of the specified directory. This file contains the posets of size n that have a unique maximal element. (The posets with a unique maximal element can also be generated by using UniqMaxmlQ[] in conjunction with Select.nb on thrstds* or stdpsts* for * = n.) If n = 1, BldUMxmls.nb starts with the list containing the empty set (or empty poset). Creating a single element then gives the poset of size 1. If n > 1, BldUMxmls.nb needs the file stdpsts* for * = n-1. First the program reads in the standard posets of size n-1 if n > 1. For each of these posets, dropping edges from a new element to each of its maximal elements creates a new poset of size n. By the proposition from Chapter III, each of the posets formed in this way is in standard form. The final list is sorted and then written to the file. At the end of the program, the consumption of time and memory is also displayed.

E. Routine Poset Property Test Functions

ConnctdQ[] returns True if the given poset is connected and False otherwise. It works by calling the recursive program CnnctdQ[], which has three arguments. The first argument of it is always the poset passed from ConnctdQ[]. In its first call, the second two arguments are {} and {1}. CnnctdQ[] finds all elements that cover 1 and calls itself with this as the third argument and {1} as the second. At any later stage, the second argument, sofar, is a list of the elements that are known to be connected to the element 1. The third argument, nw, is a list of elements that share an edge with elements in sofar but are not in sofar. The program then finds all elements that share an edge with the elements in nw and are not in sofar. CnnctdQ[] calls itself with this set as the third argument and the combined list of elements in sofar and nw as the second. Once the third argument is the empty set, the connected component based at 1 has been found and the recursion stops. Then the poset is connected only if the current second argument is the list of all elements in the poset.

The argument of **UniqMaxmlQ[]** is also a poset. UniqMaxmlQ[] consists of a single Boolean test. The program finds the number of elements in the poset that are covered by at least one other element. If this number is one less than the size of the poset, there is exactly one maximal element in the poset.

dCompltQ[] determines whether or not the given poset is d-complete. The program checks the poset to see if it fails any of the requirements to be d-complete. If any failure is found, the program returns False and stops. Once the program has determined that no elements in the poset have more than two parents, the shape of the Hasse diagram is considered. The program first ensures that there are no x shaped four element convex subsets. If every v-bottom is part of a diamond shape and if every diamond top covers exactly two elements, the program then runs through the v-bottoms. If a tail can be extended from the bottom of the diamond, dCompltQ[] checks that there is only one extension. If this is the case, it then checks that the neck can also be extended in exactly one way. If the poset has not been eliminated by one of the tests, it is d-complete and the program returns True.

F. Data File Inspection and Comparison Main Programs

In **Inspct.nb**, the user specifies a directory, a family of files, and a poset size to indicate which file will be examined. Inspct.nb reads in this file from the directory. Then when the user executes the first cell, the length of the file is displayed. Next the user may change the number of pairs of lines beyond two that will be printed. Then the user must execute the second cell. When this is done, the first two and last two lines of the file are printed, as well as intermediate lines if addpairs > 0.

Compar.nb is used to compare two files of posets. The user must enter the directory of these files, the two families of files and the poset size. The program then reads in these files and prints their lengths and the length of their intersection. Next it searches for their relationship. Once a relationship is found the program prints a statement indicating this relationship. If one list is smaller than the other, the program checks if that poset list is a subset of the other. If the files have the same length, it checks if the files are the same or the same after sorting. If none of these is true, the standard form of every element in the second file is found and this list is sorted. This new poset list is then compared to the first file to determine if there is a relationship between these files. If there is no relationship between this list and the first file, a statement indicating that the two files do not have a simple relationship is printed.

G. Data File Creation Main Programs

Select.nb tests posets to see if they have a specified property. Along with the property, the user must also specify the directory, the file name (including a subdirectory) of the posets to be tested, and a range of poset sizes to test. In order to be able to use the desired property, the file containing this property must also be entered. For each size, the posets that have the indicated property are selected from the input file with posets of the corresponding size. These posets are then written to a file. The name of this file is also entered by the user and should reflect the test being performed. For example, if Select.nb is used with ConnctdQ[], the connected posets of size 8 are written to connets8, which lies in the subdirectory Connets.

For **Intersct.nb**, the user must specify a directory, two file names (including subdirectories) of poset lists, and a range of poset sizes. For all sizes within the indicated range, the intersection of the list of posets of that size in the two files is found. This intersection is written to a file entered by the user. This file name should indicate the two intersected files. For example, if the files Connets/connets8 and dCmplts/dcmplts8 are intersected, the connected d-complete posets of size 8 are written to Cnctdcs/cnctdcs8.

Complemnt.nb is similar to Intersct.nb. The user specifies a directory, two file names (with subdirectories) of lists of posets, and a range of poset sizes. For each size in this range, the list of posets in the first file list that are not in the second file list is found. This list is then written to a file indicated by the user. Again, this file name should reflect the two input files. For example, if the first file is UMaxmls/umaxmls6 and the second file is Cnctdcs/cnctdcs6, the posets of size 6 with a unique maximal element that are not d-Complete are written to UMxndcs/umxndcs6.

H. JDT and L-R Properties Test Function and Main Program

JDTLRQ[] determines if the given poset has the jdt property and if it has the Littlewood-Richardson property. This poset is considered to have black labels. The program requires some outside data aside from its argument. This data must be provided in any program that uses JDTLRQ[], as in JDTLRscan.nb. JDTLRQ[] returns an ordered pair. The first element in this pair is True if the poset has the jdt property and False otherwise. The second element is True if the poset has the Littlewood-Richardson property and False if not. The poset is assumed to have both the properties at first, so the variables jdt and lr are initialized to True. If a violation of one of the properties is found, the corresponding variable is set to False. JDTLRQ[] uses two smaller programs, **Slde[]** and **Migrt[]**, that are defined in terms of the argument of JDTLRQ[].

Slde[] has two arguments. The first is a list of black elements in the poset that have been labeled red, listed in order of their red labels. The second argument is an element of the poset that is a green bubble. Slde[] determines the result of moving the red-labeled elements up as the green bubble moves down. It keeps track of the current black positions of the red labels and the current black position of the green bubble. While there are still red labels below the current position of the green bubble, Slde[] finds the largest red label that the green bubble covers. The position of the bubble and this red label is switched and the list of the current positions of the red labels is updated. When there are no more red labels, the green bubble has moved down as far as possible. Slde[] then returns a pair of elements. The first element is the final black position of the bubble and the second is a list of the final black positions of the red labels.

Migrt[] implements Slde[] to find the result of moving red labels up as more than one green bubble is moved down. It has three arguments. The first of these is a list of the black positions of green bubbles that have moved down as far as possible, listed in the green order. The second is a list of the black positions of the red labels listed in order of these red labels. The final argument of Migrt[] is a list of the black positions of green bubbles that need to be moved down listed in the green order. Migrt[] acts recursively. If there are still green bubbles to be moved down, Migrt[] uses Slde[] to find the result of moving the smallest green bubble down with the current red labeling. It then calls itself. The first argument in this call is the current list of final green bubble positions combined with the black position of the current bubble returned from Slde[]. The second argument is the resulting red label positions returned from Slde[]. The third argument is the current list of positions of green bubbles that need to be moved down with its first element removed. When the third argument of Migrt[] is the empty set, all of the green bubbles have moved down and the recursion stops. Migrt[] then returns a pair of elements. The first of these is the final black positions of the green bubbles listed in the green order and the second is the final black positions of the red labels listed in the red order.

Before JDTLRQ[] calls these two subprograms, more information is found. First a list of ideals of the input poset, grouped by size, are found for all sizes between two and two less than the size of the poset. The variable szord contains a list of ideal sizes in the mentioned range. JDTLRQ[] then runs through the ideal sizes in this listed order. For each size, the program runs through all ideals of that size. The corresponding filter for the current ideal is found. Then InvStdTblx[] is used with InvExtsWRI[] to find all possible ways of labeling the ideal red and all possible ways of labeling the filter green. For every red labeling of the ideal and green labeling of the filter, Migrt[] is used with these as its second two arguments and {} as its first argument to find the result of moving the green bubbles down and the red labels up.

Once these combinations of red and green labelings are found, JDTLRQ[] checks to see if the jdt property holds for the current ideal/filter pair. The results for the various combinations of red and green are compared. For a given red labeling of the ideal, the program looks at the resulting positions of the red labels for each of the green labelings of the filter. If any of these resulting red labelings are different, the poset does not have the jdt property and so the value of jdt is changed to False. Otherwise, it is still True. Once it has been found that the poset does not have the jdt property, the remaining red labelings do not need to be considered and the loop is exited.

After checking for the jdt propery, the program determines whether the Littlewood-Richardson property is satisfied for the current ideal/filter. For a given green labeling of the filter, JDTLRQ[] compares two lists. The first list is the sorted list of pairs containing the final positions of the red and green labels for each red labeling of the ideal. The second list is found by first forming the list of the distinct final green tableaux. For each of these tableaux, all red labelings of the corresponding filter are found using InvStdTblx[] and InvExtsWRI[]. The desired second list is then a list of all ordered pairs whose second element is one of these red labelings and whose first is its corresponding green labeled ideal. If the two lists are equal for any green labeling of the filter, the poset has the Littlewood-Richardson property. Then the remaining green labelings do not need to be considered and so the loop is exited. Otherwise, lr remains False and the program keeps checking. After each ideal and between ideal sizes, JDTLRQ[] checks the value of jdt and lr. If both jdt and lr are False, the poset does not have either property so the remaining ideals do not need to be considered. The program then exits all loops, returns {False, False}, and ends. Otherwise all the listed ideals are tested and the values of jdt and lr are returned.

JDTLRscan.nb uses JDTLRQ[] to determine which posets in a given file have the jdt property and which have the Littlewood-Richardson property. The directory, the file name (including a subdirectory) and a poset size must be specified by the user. The program first reads in stored information needed by JDTLRQ[]. For all sizes, *, between 4 and the input size– 2, the equations from stdisos* and lookups* are needed since JDTLRQ[] uses InvExtsWRI[]. The list of inverse extensions from invexts* are stored in invrsex[*] for this reason as well. The order by size for examining the ideals in JDTLRQ[] is also needed. This is stored in szord. The list of posets in the indicated file are read in. For each poset in this list, JDTLRscan.nb calls JDTLRQ[]. If the first element of the returned pair is True, the poset is added to the list of posets with the jdt property. If the second element of the pair is True, it is added to the list of posets with the Littlewood-Richardson property. If the input file is umxndcs*, the list of connected non-d-complete posets with the jdt property are written to the file CJDTndc/cjdtndc* and the list of connected non-d-complete posets with the Littlewood-Richardson property are written to the file CLRndcs/clrndcs*, where * is the input size. The consumption of time and memory is then printed to the screen.

THIS PAGE IS INTENTIONALLY BLANK