Appendix

adjmat(R1,R2):

This method takes as input two level sets of partitions, *R1* and *R2*, and returns the adjacency matrix where the (i, j) entry is either 1 if the ith member of *R1* covers the jth entry of *R2* or 0 otherwise.

Parameters:

R1, R2: two level sets of partitions of the same value n as created by the latex command *Partitions(n,length=k)*.

Returns:

the adjacency matrix where the (i, j) entry is either 1 if the ith member of *R1* covers the jth entry of *R2* or 0 otherwise.

covers(P1,P2):

This method takes two partitions, *P1* and *P2*, as inputs and determines whether *P1* covers *P2*.

Parameters:

P1, P2: two partitions of the same value n taken from the list generated by the latex command *Partitions(n,length=k)*.

Returns: 1 if *P1* covers *P2*, 0 otherwise.

CRTUnimodal(fileList, primeList, start, finish):

(partially pseudocode for parallelization). This method uses the matrices generated by *PPrime* to recover $p(n, k)$ values and, in turn, determine whether $\mathcal{P}_n$ is unimodal for *start* $\leq n \leq$ *finish*.

Parameters:

fileList: a list of the files that contain the matrices generated by *PPrime*

primeList: a list of the moduli used to create the matrices in *PPrime*.

start: the minimum value of n for which $\mathcal{P}_n$ is to be tested for unimodality.

finish: the minimum value of n for which $\mathcal{P}_n$ is to be tested for unimodality.

Returns: 1 if each specified $\mathcal{P}_n$ is unimodal and 0 otherwise (along the first value of n which is not unimodal).

dana2(R1,R2,B,numtimes):

This function conducts a random walk along matchings of two inputted consecutive level sets with each step determined as follows: A random edge is selected. Then, if the edge can be added to the graph of the two level sets, it is added. Next, if the edge can be removed, it is removed with probability $1/B$. Otherwise, if the edge can be neither added nor removed, a new edge is selected and the process is repeated. Last, if the resulting graph is a maximal matching, the counter of each included edge is incremented. Steps are taken until *numtimes* maximum matchings are encountered, and the list of edgecounts is returned. Note: *danafixed* begins with user-specified edges already included as per the additional argument *fixededges*.

Parameters:

R1,R2: two level sets of partitions of the same value n as created by the latex command *Partitions(n,length=k)*.
B: the reciprocal of the probability with which an edge is removed in the random walk algorithm.
numtimes: the number of maximum matchings that will be found before the method returns the edgecounts.

Returns:

edgecount: a list of the edgecounts used in the recorded maximum matchings. (This list uses the same ordering as *edgelist*.
matchinglist: [only returned by *dana3*] - a list of the maximum matchings found throughout the duration of the method.
hashcount: [only returned by *dana3*] - a list of how many times each maximum matching occurred throughout the method (with the same ordering as *matchinglist*).

dana3(R1,R2,B,numtimes):

See *dana2*.

danafixed(R1,R2,B,numtimes,fixededges):

See *dana2*.

depthfirstsearch(sourceIndex, sinkIndex, edgeCapacity, nodesvisited, listOfEdges, done):

This method performs a depth first search to find a path through the graph as part of the *fordfulkerson* algorithm.

Parameters:

sourceIndex: an integer corresponding to the node where the search is to begin.
sinkIndex: an integer corresponding to the node to be found.
edgeCapacity: a list of integers representing the capacity of all of the edges in the graph. It is indexed in the same fashion as listofedges.
nodesvisited: a list of booleans. Each entry corresponds to one node (vertex) in the graph. Initially, every entry in this list should

be False.

listOfEdges: a list of two-item lists. Each element represents an edge in the graph. The edge is identified by numbers corresponding to the vertices to which is adjacent. These vertices are numbered in the same fashion that they are numbered in the list nodesvisited.

done: boolean indicating when the search has completed. It should initially be set to false.

Returns: a list corresponding to the path, if a path is found, or False, if no path exists.

edgelist(nodelist, size1, size2):

This method takes a list of all the nodes in a graph and returns a list of all edges in the graph.

Parameters:

nodelist: list of all of the nodes in the graph.

size1: the cardinality of the first level set

size2: the cardinality of the second level set

Returns: a list of two-item lists. Each element represents an edge in the graph. The edge is identified by numbers corresponding to the vertices to which is adjacent. These vertices are numbered in the same fashion that they are numbered in the parameter nodelist.

edgelist(R1,R2):

This function takes two level sets, *R1* and *R2*, as inputs and returns a list of ordered pairs $[i, j]$ such that *R1*[i] and *R2*[j] form an edge according to the covering relation.

Parameters:

R1, R2: two level sets of partitions of the same value n as created by the latex command *Partitions(n,length=k)*.

Returns:

edgelist: a list of ordered pairs $[i, j]$ such that *R1*[i] and *R2*[j] form an edge according to the covering relation.

edgestats(R1,R2):

This method takes two adjacent level sets, *R1* and *R2*, as inputs. Then, for each edge $(P1, P2)$, where P1 is an element of *R1* which covers P2 (an element of *R2*), the function displays how many of the total possible maximum matchings between the level sets include the given edge.

Parameters:

R1, R2: two level sets of partitions of the same value n as created by the latex command *Partitions(n,length=k)*.

Returns: nothing, but displays the appropriate data.

fordfulkerson(R1, R2):

This method uses the Ford Fulkerson algorithm with a depth first search to determine if a maximum matching exists.

Parameters:

R1, R2: lists of partition objects representing two adjacent level sets

Returns: True if a maximum matching exists between the two level sets, False otherwise.

fordfulkerson(R1, R2, nodelist, listofedges, edgeCapacity):

This method uses the Ford-Fulkerson algorithm to determine if two adjacent level sets have a maximum matching.

Parameters:

R1, R2: lists representing two adjacent level sets
nodelist: a list of all nodes (vertices) in the graph
listofedges:a list of two-item lists. Each element represents an edge in the graph. The edge is identified by numbers corresponding to the vertices to which is adjacent. These vertices are numbered in the same fashion that they are numbered in the lists nodelist and edgeCapacity.
edgeCapacity: list of integers corresponding to the capacities of all edges in the graph. This list is indexed in the same fashion as listofedges.

improvedunimodal(input,start,finish):

This method uses the recurrence $p(n, k) = p(n-1, k-1)+p(n-k, k)$ to determine the values of $p(n, k)$ for $start \leq n \leq finish$. It then tests these values along the way to determine if each corresponding $\mathcal{P}_n$ is unimodal. Additionally, as it progresses, the method deletes the $p(n, k)$ values which become unnecessary for future computations. In this manner, it reduces the necessary storage capacity and allows the program to be run for larger values of n than would be possible for P.

Parameters:

input: a list of lists generated by another execution of *improvedunimodal* that can be used to continue starting at the n value one greater than the *finish* value of the previous execution. If $start = 1$, this parameter should be an empty list.
start: the maximum value of n for which $\mathcal{P}_n$ is to be tested for unimodality.

finish: the minimum value of n for which $\mathcal{P}_n$ is to be tested for unimodality.

Returns: 0 if a $\mathcal{P}_n$ is determined to not be unimodal (along with a display of the n value). Otherwise, the method returns the list *[L,modes]*, with

L: a list of the currently stored $p(n,k)$ values to possibly be used in further executions of *improvedunimodal* where the future execution's value of *start* is set to the current execution's value of $\mathit{finish}+1$.

modes: a list of the ranks of the modes of $\mathcal{P}_{start}$ through $\mathcal{P}_{finish}$, respectively.

incidenceMatrix(R1,R2):

This function takes two level sets, *R1* and *R2*, as inputs and returns the incidence matrix formed by the partitions of *R1* and *R2* using the covering relation.

Parameters:

R1, R2: two level sets of partitions of the same value n as created by the latex command *Partitions(n,length=k)*.

Returns:

M: the incidence matrix determined by the vertices $V = R1 \cup R2$ and the edges $E = \{(v_1, v_2) : v_1, v_2 \in V$ and either $v_1 \lessdot v_2$ or $v_2 \lessdot v_1\}$.

matchingByRank(n,k):

This method uses the rank of the incidence matrix to determine whether or not there exists a maximum matching between $\Lambda_{n,k}$ and $\Lambda_{n,k+1}$. (A maximum matching exists iff the rank of the incidence matrix is equal to one less than the total number of partitions in the two level sets [**?**]).

Parameters:

n: The specified number to be partitioned.

k: the smaller of the two ranks of the consecutive level sets to be tested.

Returns: 1, if a maximum matching exists, 0, otherwise

matchingdata(start, finish, B, numtimes):

This method uses *numtimes* steps in *dana2* to approximate the relative frequencies of each edge for consecutive pair of level sets for each value of n in [*start,finish*].

Parameters:

start: the minimum value of n for which data will be displayed.

finish: the maximum value of n for which data will be displayed.

B: the reciprocal of the probability with which an edge is removed in the random walk executed by *dana2*.

B: the number of maximum matchings *dana2* finds before it returns.

Returns: nothing, but prints the appropriate data to the screen.

mode(M):

This method determines the first occurence of the maximum of the largest element in each row of the inputted matrix. In practice, this method is used to determine modes of $\mathcal{P}_n$ using a matrix generated by the method P. The maximum row elements (modes) are returned in a list in the order of the rows of the inputted matrix.

Parameters:

M: the inputted matrix.

Returns:

modes: a list of the maximum row elements (modes) in the order of the rows of the inputted matrix.

order(R1):

This method converts a level set of partitions from Sage's proprietary data format into a list of lists, where each interior list represents a single partition. Each individual partition is sorted lexicographically; then, the list of partitions is sorted lexicographically.

Parameters:

R1: two partitions of the same value n taken from the list generated by the latex command *Partitions(n,length=k)*.

Returns:

modes: the level set stored as a list of lists.

P(howfar):

This method uses the recurrence $p(n,k) = p(n-1,k-1)+p(n-k,k)$ to determine the values of $p(n,k)$ for $\textit{start} \leq n \leq \textit{finish}$. The values are returned in a matrix where the (i,j) entry equals $p(i,j)$.

Parameters:

howfar: the largest value of n for which the method calculates $p(n,k)$.

Returns:

M: a $\textit{howfar} \times \textit{howfar}$ matrix where the (i,j) entry equals $p(i,j)$.

perm(M):

This method uses the algorithm described by Mittal and Al-Kurdi

[**?**] to determine the permanent of the inputted matrix. This algorithm is specifically designed to calculate the permaments of sparse 0-1 matrices more quickly.

Parameters:

M: the inputed matrix.

Returns:

the permanent of *M*.

Pmod(howfar,p):

This method uses the recurrence $p(n,k) = p(n-1,k-1)+p(n-k,k)$ to determine the values of $p(n,k)$ for $start \leq n \leq finish$ modulo the specified number p. The values are returned in a matrix where the (i,j) entry equals $p(i,j) \pmod{p}$.

Parameters:

howfar: the largest value of n for which the method calculates $p(n,k) \pmod{p}$.

p: the modulus in which all $p(n,k)$ values will be calculated.

Returns:

M: a *howfar* × *howfar* matrix where the (i,j) entry equals $p(i,j) \pmod{p}$.

PPrime(howfar,howbig,howmany):

This method generates a list of the first *howmany* primes after *howbig*. It then uses *Pmod* to construct a matrix of $p(n,k)$ values modulo p for each p in the generated list of primes. By calculating modulo primes, we decrease the necessary storage size for each individual matrix.

Parameters:

howfar: the value of n up to which the $p(n,k)$ values will be calculated for the matrix

howbig: the program makes a list of the first *howmany* primes after this value

howmany: determines the number of primes in the generated list to be used as moduli

setup(L1,L2):

This method creates a partial matching and then calls *fordfulkerson* to determine if a maximum matching exists.

Parameters:

L1, L2: two lists representing two adjacent level sets.

Returns: True, if a maximum matching exists, False, otherwise

topToMode2(n,mode):

This method determines whether the "match furthest available to the left lexicographically" scheme from section [?] creates a maximum matching between each consecutive pair of levels above and including the mode for $\mathcal{P}_n$.

Parameters:

n: the n value for which $\mathcal{P}_n$ will be checked.

mode: the rank of the mode of $\mathcal{P}_n$.

Returns: 1 if the scheme works; otherwise, it returns 0, along with the first row numbers of $\mathcal{P}_n$ for which the scheme does not work.

unimodal(M):

This method determines if each row of the inputted matrix is unimodal.

Parameters:

M: the inputted matrix.

Returns: 1 if each row is unimodal; otherwise, it returns 0, along with the row number of the matrix row which is not unimodal.