**A. InitialSpinConfig**

*1. Description*

Sets inital potts spins configuration.

*2. Input parameters*

`N` Number of points.

`Q` Number of Potts spin values.

*3. Output parameters*

`Spin` Spin[i] is the spin value ascociated with vertex i.

*4. file*

aux.c

**B. NewSpinConfig**

*1. Description*

assign each cluster a random spin value.

*2. Input parameters*

`N` Number of points.

`Block` point i belongs to cluster Block[i].

`MBlk` number of clusters.

`Q` Number of Potts spin values.

`NewSpinValue` Previously allocated workspace.
Must have minimal size of N*sizeof(unsigned int).

*3. Output parameters*

`Spin` Spin[i] is the new spin value ascociated with vertex i.

*4. file*

aux.c

**C. DeletionProbabilities**

*1. Description*

Gives the deletion probabilities for a satisfied bond. If bond is unsatisfied its deletion probability is 1.

*2. Input parameters*

`T` Temperature.

`J` J[i][j] is the interaction between i and NK.p[i][j], where NK is the edges array.

*3. Output parameters*

`P` P.p[i][j] is the deletion probability of a satisfied bond between vertices i and NK.p[i][j] when their spin values are equal.

*4. file*

aux.c

**D. SetBond**

*1. Description*

Decides which bonds to freeze.

*2. Input parameters*

`P` P.p[i][j] is the deletion probability of the edge between vertices i and NK.p[i][j] if they have the same spin.

`Spin` Spin[i] is the current spin of vertex i.

`NK` nearest neighbours array. For each vertex the list of neighbours *must* be in ascending order.

`KN` the inverse nearest neighbours array.

*3. Output parameters*

`Bond` Frozen bond array. Bond.p[i][j] = 1 $\longrightarrow$ bond between i and NK.p[i][j] is frozen. Bond.p[i][j] = 0 $\longrightarrow$ the bond is deleted.

*4. Return value*

`nb` the number of frozen bonds.

*5. file*

aux.c

**E. Coarsening**

*1. Description*

Uses the frozen bonds to indentify the connected components (clusters). Assign each vertex its cluster number.

*2. Input parameters*

`Bond` Frozen bonds array. Bond.p[i][j] = 1 $\longrightarrow$ the bond between i and NK.p[i][j] is frozen. Bond.p[i][j] = 0 $\longrightarrow$ the bond is deleted.

`NK` nearest neighbours array.

`Stack` previously allocated workspace of size >= N*sizeof(unsigned int).

*3. Output parameters*

`Block` assignment to clusters. Spin i belongs to cluster Block[i].

`ClusterSize` cluster sizes. ClusterSize[k] is th size of cluster k.

*4. Return value*

the number of clusters.

*5. file*

aux.c

**F. OrderingClusters**

*1. Description*

reassign cluster numbers according to cluster sizes, so if i < j $\longrightarrow$ cluster i is larger than cluster j.

*2. Input parameters*

`N` number of vertices.

`nblock` number of clusters.

`Block` vertex i belongs to cluster Block[i].

`Size` Cluster sizes. Cluster i contains Size[i] vertices.

`Indx` previously allocated workspace of size >= 2*N*sizeof(unsigned int).

*3. Output parameters*

`Block` new cluster number assigned to each vertex.

`Size` cluster sizes, ordered in descending order, according to their new numbers.

*4. file*

aux.c

### G. CheckParam

*1. Description*

Check evironment parameters in order to prevent runtime error.

*2. file*

aux.c

### H. CheckParam

*1. Description*

Set default values to environmrnt parameters.

*2. file*

aux.c

### I. Energy

*1. Description*

Calculate the energy for a given configuration:

$$E(\{s_i\}) = \sum_{i,j} J_{ij}(1 - \delta_{s_i s_j})$$

*2. Input parameters*

`Block` vertex i belongs to cluster Block[i].

`J` Interactions array.

`NK` Neighbours array.

*3. Return value*

the energy

*4. file*

aux.c

**J. AverageInteraction**

*1. Description*

calculates the average interaction between neighbors.

*2. Input parameters*

`J` the interaction array. J.p[i][j] is the interaction between point i and its jth neighbour.

*3. Return value*

returns "the Characteristic Distance".

*4. file*

aux2.c

**K. GlobalCorrelation**

*1. Description*

Builds the correlations array. If two points belongs to the same cluster, one is added to the coresponding matrix element.

*2. Input parameters*

`CorrN` the correlations array. CorrN.p[i][j] is the number of times vertex i and vertex NK.p[i][j] were in the same cluster so far.

`NK` nearest neighbours array. NK.p[i][j] is the j-th neighbour of vertex i.

`Block` the cluster each vertex belongs to. vertex i belongs to cluster number Block[i].

*3. Output parameters*

`CorrN` the updated correlation array.

*4. file*

aux2.c

**L. FourPointCorrelation**

*1. Description*

Builds the four point correlations array. If four points belongs to the same cluster, one is added to the coresponding matrix element.

*2. Input parameters*

`FPCorr` the four point correlation array. CorrN.p[i][j].p[k][l] is the number of times vertices i, NK.p[i][j], k and NK.p[k][l] were in the same cluster so far.

`NK` nearest neighbours array. NK.p[i][j] is the j-th neighbour of vertex i.

`Block` the cluster each vertex belongs to. vertex i belongs to cluster number Block[i].

*3. Output parameters*

`FPCorr` the updated four point correlation array.

*4. file*

aux2.c

**M. Magnetization**

*1. Description*

Obtain how many points have each spin value. order the groups in discending order of size. Calculate the magnetization for each spin color as

$$m_q = \frac{QN_q - N}{N(Q-1)},$$

where $N_q$ in the number of points with spin color $q$.

*2. Input parameters*

`N` number of points.

`Q` number of spin values (colors).

`nc` number of clusters.

`*ClusterSize` cluster sizes.

$N_q$ previously allocated workspace of size >= Q*sizeof(unsigned int).

*3. Output parameters*

`mag` the magnetization vector.

*4. Auxiliary function*

int uicompare(const void *i, const void *j)

*5. file*

aux2.c

**N. OrderClusterSize**

*1. Description*

Order cluster sizes list in discending order..

*2. Input parameters*

`nc` number of clusters.

`*ClusterSize` cluster sizes.

*3. Output parameters*

`*ClusterSize` ordered cluster sizes.

*4. Auxiliary function*

int uicompare(const void *i, const void *j)

*5. file*

aux2.c

**O. ClusterAverage**

*1. Description*

Calculate cluster size averages. Returns the mean value and variance of the larger, second larger, .., smaller cluster sizes.

*2. Input parameters*

`nc` number of clusters.

`*Size1` cluster size conumlant. The i-th element is the comulant of the i-th larger cluster size.

`*size2` cluster size$^2$ comulant.

*3. Output parameters*

`*Size1` sizes mean value.

`*Size2` sizes variance.

*4. file*

aux2.c

**P. Susceptibility**

*1. Description*

Calculate magnatizations mean value and susceptibilities.

*2. Input parameters*

`Q` number of spin values.

`ncy` number of SW sweeps performed = numbers of samples taken.

`*M1` magnetizations comulant.

`*M2` magnetizations$^2$ comulant.

*3. Output parameters*

`*M1` magnetizations mean value.

`*xi` susceptibilities.

*4. file*

aux2.c

**Q. ReadEdgeFile**

*1. Description*

Add vertices from a file to the vertex array.

*2. Input parameters*

`N` Number of points.

`*nk` if nk→n = N it is a ragged array of vertices. otherwise, it is an uninitialized array.

*3. Output parameters*

`*nk` the list of vertices, including the ones read from the file.

*4. file*

edge.c

**R. OrderEdges**

*1. Description*

order the sub arrays of a ragged array in ascending order.

*2. Input parameters*

`NK` the ragged array to be ordered.

*3. Output parameters*

`NK` The ordereded array. NK.p[i][j] $<$ NK.p[i][l] $\iff$ j $<$ l.

*4. Auxiliary function*

int uicomp(const void *i, const void *j)

*5. file*

edge.c

**S. InvertEdges**

*1. Description*

creates an inverted ragged array.

*2. Input parameters*

`NK` the ragged array to be inverted.

*3. Return value*

`M` The inverted array. If k=NK.p[i][j] and i=NK.p[k][l] then M.p[i][j]=l.

*4. file*

edge.c

**T. knn**

*1. Description*

Creates a mutual K nearest neighbours array. Fuses it with a minimal spanning tree if required.

*2. Input parameters*

`N` Number of points.

`D` Dimension of vertex vector.
$D = 0 \longrightarrow$ use distances.

`X` $D = 0$: X is the distance matrix.
$D > 0$: X[i] is the $D$-dimentional vector ascociated with vertex i.

*3. Return value*

`nk` nk.p[i] is the list of neighbours of vertex i.

*4. file*

edge.c

**U. mstree**

*1. Prim's Algorithm*

1. Set $i = 0$, $S_0 = \{u_0 = s\}$, $L(u_0) = 0$, and $L(v) = \inf$ for $v \neq u_0$. If $|V| = 1$ then stop, otherwise go to step 2.
2. For each $v$ in $V \setminus S_i$, replace $L(v)$ by $\min\{L(v), d_{v,u_i}\}$. If $L(v)$ is replaced, put a label $(L(v), u_i)$ on $v$.
3. Find a vertex $v$ which minimizes $\{L(v) | v \in V \setminus S_i\}$, say $u_{i+1}$.
4. Let $S_{i+1} = S_i \cup \{u_{i+1}\}$.
5. Replace $i$ by $i + 1$. If $i = |V| - 1$ then stop, otherwise go to step 2.

The time required by Prim's algorithm is $O(|V|^2)$.
It can be reduced to $O(|E| \log |V|)$ if heap is used (but i didn't bother).

*2. Input parameters*

`N` number of points

`d` distance matrix

*3. Output parameters*

`**edg` the edges of the minimal spanning tree. Edge i is between vertices edg[i][0] and edg[i][1].

*4. file*

edge.c

### V. Use of Parameter environment

This environment allows you to set parameters, so that their values can be read at any time and anywhere in the program. The parameters can be written and read as a char string, an integer, or a floating point argument. The following functions are available:

**int SetParam( char* n, char* v )**
Parameter n is set to string v. If v=NULL the parameter is set to a zero length string. Return 1 if parameter was already set, and 0 otherwise.

**int UnsetParam( char* n )**
Unset (delete) the parameter n. returns 1 if parameter existed, 0 otherwise.

**char* GetParam( char* n )**
Returns a pointer to the string value of parameter n. If the parameter was set with NULL pointer, the function returns a pointer to zero length string. If the parameter is not set, the function returns NULL.

**int IGetParam( char* n )**
Return the value of parameter n as an integer. If the parameter does not describe an integer or is not set, the function returns 0.

**int FGetParam( char* n )**
Return the value of parameter n as a float. If the parameter does not describe a float or is not set, the function returns 0.

**int FSetParam( char* n, float l )**
Sets the value of parameter n to represent the floating point argument l. Returns 1 if parameter was already set, 0 otherwise.

**int ISetParam( char* n, int j )**
Sets the value of parameter n to represent the integer j. Returns 1 if parameter was already set, 0 otherwise.

### W. DSortIndex

*1. Description*

Sort an the indices vector $j$ of a double array $a$
so that $m < n \longrightarrow a[j[m]] < a[j[n]]$.

*2. Input parameters*

`n` Number of elements in the array.

`a` The double array.

*3. Output parameters*

`j` The ordered indices array.

*4. Auxiliary function*

dindcmp()

*5. file*

utilities.c

**X. Environment Parameters**

To make the program simpler and clearer we use environment parameters, in which we keep data and flags to be use by different functions, without the need to pass them as arguments. This also allows an easy way to add more flags and options to the program wuthout littering the code too much. The parameters used in the programm so far are:

`AverageInteraction`
: the average interaction between neighbours.

`ClustersReported`
: how many cluster sizes are reported.

`ClusterMinSizeReported`
: minimal size of cluster to be reported. all clusters above or equal this size will be reported, even if their number exceeds the `ClustersReported` value.

`CharDist`
: characteristic distance between neighbours (used for calculating the interaction).

`DataFile`
: the file containing the coordinate of the point, or the distances matrix.

`DataIsInteraction`
: The distances obtained should be cosidered as the interaction.

`Dimensions`
: dimention of the varctors describing the points. if 0, the data is expected to be a distances matrix.

`DirectedGrowth`
: Report cluster obtained from the correlations by directed growth.

`DataIsMatrix`
: the data file is organized as a distance matrix and not as a list.

`EdgeFile`
: file containing edges to be added to the list of nearest neighbours.

`FourPointCorr`
: Sample and report four point correlations.

`ForceNN`
: Use this (false) number of nearest neighbours for calculating the interaction.

`ForceRandomSeed`
: Random seed to start the program with, if we want this run to be identicle to a previous one. If not set, the seed is taken as the clock value.

`ForceChD`
: Use this (false) characteristic distance for calculating the interaction.

`InfMetric`
: Use infinity metric to calculate distances.

`KNearestNeighbours`
: maximal number of nearest neighbours (used in the knn algorithm).

`Lambda`
: $\lambda$ value to be used if `UseZ` is set.

`MinTemp`
: The lowest temperature to use..

`MaxTemp`
: The maximal temp. to use.

`MSTree`
: add the edges of the minimal spanning tree.

`NumberOfPoints`
: number of points.

`NumberOfEdges`
: the total number of edges.

`NearestNeighbrs`
: average number of nearest neighbours (used for calculating the interaction).

`OutFile`
: prefix for the output files.

`PottsSpins`
: The number of different spin values.

`PrevTempFile`
: prefix for former output files, used to obtain initial states for the threshold and directed growth calculations.

`RandomSeed`
: The random seed with which the program has started.

`SusceptColors`
: number of susciptibilities from the susceptibility vector to report.

`SaveAverages`
: sample and report data on Swensen-Wang averages.

`SaveSuscept`
: sample and report magnetization and susceptibility data.

`SWCycles`
: number Swensen-Wang sweeps.

`SWFraction`
: the fraction SW sweeps for which averages are calculated. The first (1-SWfract)*cyc sweeps are discarded.

`Threshold`
: Report cluster obtained from the correlations by thresholding.

`ThresholdTheta`
: The threshold on correlations between neighbours, above which they are assumed to belong to the same cluster.

`ThresholdMin`

`ThresholdMax`

`ThresholdStep`
: If those variables are set, the programs report the results obtained for different thresholds, taken from min to max values with the step indicated.

`TempStep`
: the steps in temperature from min to max values in which the simulation should be ran.

`Timing`
: report the time required by the programm for each step.

`UseZ`
: Scale the interaction according to a given $\lambda$.

`WriteLabels`
: Write the label of the cluster each spin belongs to.

`WriteCorFile`
: Write the correlations between each pair of neighbours.