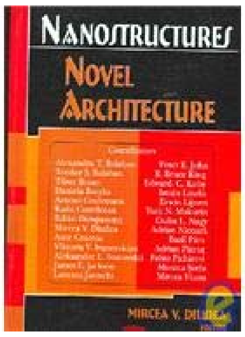


Nanostructures: Novel Architecture

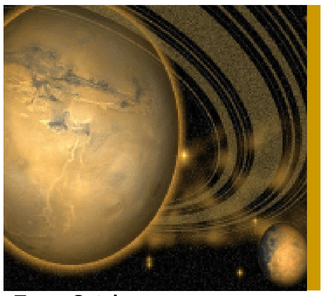
Author(s): Diudea, Mircea V.; Balaban, Alexandru T. (CON); Balaban, Teodor S. (CON); Braun, Tibor (CON)

ISBN10: 1594544999
ISBN13: 9781594544996
Format: Hardcover
Pub. Date: 1/15/2006
Publisher(s): Nova Science Pub Inc

Table Of Contents

Preface vii
A Simplistic Approach of Revealing the Coming of Age of Nanoscience 1 (8)
Tibor Braun
Ildiko Diospatonyi
Fullerene Peapods and Related Nanomaterials: Synthesis, Structure, and Electronic Properties 9 (16)
Viktoria V. Ivanovskaya
Yurii N. Makurin
Aleksander L. Ivanovskii
Reaction Pathways in the Coalescence of Fullerenes 25 (36)
Csaba L. Nagy
Mircea V. Diude
Teodor S. Balaban
Periodic Finite Nanostructures 61 (24)
Mircea V. Diudea
Csaba L. Nagy
Ante Graovac
Theoretical Investigations of Single-Wall Nanocones 85 (26)
Alexandru T. Balaban
Polyhex Tori Originating In Square Tiled Tori 111(16)
Mircea V. Diudea
Distance Counting in Tubes and Tori: Wiener Index and Hosoya Polynomial 127(40)
Monica Stefu
Mircea V. Diudea
Counting Kekule Numbers In Torenes A Conjecture 167(8)
Peter E. John
Mircea V. Diudea
Fully-Resonant-Azulenoid Networks: Some Aspects of Construction And Taxonomy 175(18)
Edward. C. Kirby
Topological Coordinates for Schlegel Diagrams of Fullerenes and other Planar Graphs 193(10)
Istvan Laszlo
Covering Nanostructures 203(40)
Mircea V. Diudea
Algorithms for Basic Operations on Maps 243(26)
Monica Stefu
Daniela Butyka
Mircea V. Diudea
Lorentz Jantschi
Bazil Parv
Zeolite-Like Possible Carbon and Boron Nitride Allotropes 269(30)
R. Bruce King
The Symmetry of the Dyck Graph: Group Structure and Molecular Realization 299(12)
Erwin Lijnen
Arnout Ceulemans
Nanoporous Carbon Structures 311(24)
Csaba L. Nagy
Mircea V. Diudea
Cubanoid Amphiphiles as Building Blocks for the Development of Functional Nanostructured Materials 335(10)
Fabio Pichierri
Construction of Self-Assembled Nanostructures by Means of Dihydrogen Bonding 345(16)
Mircea Vlassa
Radu Custelcean
James E. Jackson
Nanosized Inorganic Metal-Oxygen Clusters 1. Basic Structures 361(38)
Adrian Patrut

Nanosized Inorganic Metal-Oxygen Clusters 2. Special Structures Adrian Patrut Adrian Nicoara	399(16)
Index	415



Contact Information

Online Contact Form

Nova Science Publishers, Inc
400 Oser Ave.
Suite 1600
Hauppauge NY
11788-3619

Phone: (631)231-7269
Fax: (631)231-8175
Email: Novascience@earthlink.net

ALGORITHMS FOR BASIC OPERATIONS ON MAPS

Monica Ştefu,[a]* Daniela Butyka,[a] Mircea V. Diudea,[a] Lorentz Jäntschi[b] and Bazil Pârv[c]

[a]Faculty of Chemistry and Chemical Engineering

"Babeş-Bolyai" University, 400084, Cluj, Romania

[b]Technical University, Cluj, Romania

[c]Department of Computer Science, Faculty of Mathematics and Computer Science

"Babeş-Bolyai" University, 400084, Cluj, Romania

*e-mail: mstefu02@yahoo.com

Abstract. Covering operations in nanostructure modeling are presented as algorithms implemented in the original software program CageVersatile 1.1. The basic operations described are: Dual, Medial, Stellation, Truncation, Leapfrog, Quadruple and Capra. Output file objects are illustrated.

1. Introduction

Covering a local planar surface by various polygonal or curved regions is nowadays a mathematically founded science.[1] The Geometry of (regular) polygons and polyhedra, Graph Theory and Set Theory concertate in the ground statement of an interdisciplinary science, often inspired from the Arts and Architecture, and implemented in Computer Science.

Covering transformation is one of the ways in understanding chemical reactions occurring in nanostructures.[2-4]

A map *M* is a combinatorial representation of a closed surface.[5,6] Several transformations (*i.e.*, operations) on maps are known and used for various purposes.

Recall some basic relations in a map:

$$\sum d\, v_d = 2e \tag{1}$$

$$\sum s\, f_s = 2e \tag{2}$$

where v_d and f_s are the number of vertices of degree *d* and number of *s*-gonal faces, respectively. The two relations are joined in the famous Euler formula:

$$v - e + f = \chi(M) = 2(1 - g) \tag{3}$$

with χ being the Euler *characteristic* and *g* the genus[7] of a graph (*i.e.*, the number of handles attached to the sphere to make it homeomorphic to the surface on which the given graph is embedded; *g* = 0 for a planar graph and 1 for a toroidal graph). Positive/negative χ values indicate positive/negative curvature of a lattice.

The above relations, along with some rules for checking the connectivity and cycle and/or polyhedral face identification were kept in mind when the CageVersatile CV1.1 software program was written.

2. Input, Output, and Helper Functions

The molecules are considered from the Graph Theory point of view, with atoms as vertices and bonds between atoms as edges. The map operations, theoretically founded in ref. 8, are presented in the order: Dual, Medial, Stellation, Truncation, Leapfrog, Quadruple and Capra. Within these operations, new vertices are put and linked, by the map operations rules, while the old vertices and edges are, in general, removed. For details see ref. 5.

The program, written in the PHP (Pre Hypertext Processed) programming language, has as input a .hin (Hyperchem) file and writes the results as output .txt and/or .hin files. It can be executed online at the internet address http://193.226.7.140/~monica/graph/cv1.1/

2.1. Input Data

In the first part of the program, the .hin file is tacken into array ***a*.** Then, the cycles with chosen maximal size are searched; the distinct faces are put into the array ***c***. The basic condition is: an edge is shared to maximum two cycles/rings/faces.

The ***a*** array contains the list of vertices of the (initial) molecular graph hydrogen depleted. The element $a[i]$ (i=1,n; n is the number of vertices in the initial molecular graph) is an array containing the coordinates, valence and the vertices linked to the vertex i.

In the array ***c***, $c[i,0]$ is the number of faces contained in the vector $c[i]$, which begins at the vertex i; $nv = c[i,j,0]$ is the number of vertices of the face $c[i,j]$, while $c[i,j,1]$ ... $c[i,j,nv]$ is the list of vertices of the face $c[i,j]$.

The ***m*** array stores the list of edges of the initial molecular graph and some additional new vertices, used within operations. It has a structure similar to the array ***c***; $m[i,j]$ represents an edge with the endpoints in $m[i,j,1]$ and $m[i,j2]$. In some cases, $m[i,j,3]$ .. $m[i,j,6]$ are used to memorize new vertices in connection with the edge $m[i,j]$.

The algorithms given below are based on the arrays ***a***, ***c*** and ***m***, derived from the input .hin file.

2.2. Output Files

All basic operations on maps described here provide .hin files as the output. The general procedure for generating an output file is: (1) generate an operation-specific array, (2) concatenate the array elements in a string and (3) write the resulted strings in the .hin format file.

The operation-specific arrays have the same structure for all operations. For example, in the MEDIAL operation, the operation-specific array ***med*** has for every atom/ vertex *i* an element *med*[i] with the .hin file structure:

med[*i*,1] = "atom ".*i*." - ".*tipa*." CA - 0 ".*x*." ".*y*." ".*z*." ";

where *i* is the atom number, *tipa* is the atom type and *x*, *y*, *z* are the coordinates;

med[*i*,2] contains the atom valence (denoted by *v*);

med[*i*,3] ... *med*[*i*,*v*+2] contain the atoms adjacent to atom *i*.

Before producing the content of the .hin file, a header must also be provided for compatibility with the Hyperchem program. This header has the general form:

;*OpName* + newLine + "forcefield mm+" + newLine + "sys 0" + *texti* + "mol 1" + newLine

where + means string concatenation, *OpName* is the name of the operation, hard-coded in every basic map operation, newLine is the line terminator, while *texti* is an explanatory text to be included in the header; mol1 marks the beginning of the first molecular graph.

2.3. Helper Operations

This section presents the helper functions, invoked in the basic map operations as follows:

- for `Dual`: `Center`, `OnSphere`, `Much_D`, `ad2l`, `WriteF`
- for `Medial`: `AdMe`, `ad2l`, `WriteF`
- for `Stellation`: `Center_D`, `Ini`, `WriteF`
- for `Truncation`: `Ad_CoordT`, `Ad_P`, `AdTr`, `WriteF`
- for `LeapFrog`: `Ad_Coord`, `Center_D`, `P_New`, `ad2l`, `Much_D`, `WriteF`
- for `Quadruple`: `IniQ`, `AdCycleInt`, `AdCoordQ`, `AdLegQ`, `MeanQ`, `WriteF`
- for `Capra`: `AdPi`, `Center_D`, `Ad_P_Middle`, `AdLeg_C`, `P_New_Capra`, `AlreadyVisited`, `Nrc`, `Verif_M2`, `Verif_M`, `Inv_56`, `WriteF`.

Function `Grow` increases the values of coordinates of the initial molecular graph by the factor `mar` = 2 to 6 times. It is used before the calling of the procedures for the operations.

```
function Grow(a, mar){
// in-out:
//    a: graph vertices array
// in:
//    mar: increasing factor
// purpose: increases the coordinates
  n := count(a); //number of vertices
  for i := 1 to n
    a[i,2]:= a[i,2]*mar;
    a[i,3]:= a[i,3]*mar;
    a[i,4]:= a[i,4]*mar;
  endfor i
end function Grow
```

Function `Center` computes the center of the molecular graph defined by the coordinates (`xc, yc, zc`) and the radius `r` of the sphere.

```
function Center(n, a, xc, yc, zc, r)
// in: n, a
// out: xc, yc, zc, r
// purpose: computes the center of the molecular graph by the coordinates
//          xc,yc, zc and the radius of the sphere.
  // the center coordinates, as arithmetic mean
  // the sphere radius is the arithmetic mean of the distances
  // between the center and the vertices.
  xc := 0; yc := 0; zc := 0;
  n := count(a); //number of vertices
  for i := 1 to n
    xc := xc + a[i,2];
    yc := yc + a[i,3];
    zc := zc + a[i,4];
  endfor i
  xc := xc/n; yc := yc/n; zc := zc/n;
  r := 0;
  for i := 1 to n
    r := r + sqrt((xc-a[i,2])^2 + (yc-a[i,3])^2 +(zc-a[i,4])^2);
  endfor i
  r := r/n;
end function Center
```

Function `OnSphere` returns the adapted coordinates (`x, y, z`) for the new vertices lying on a sphere with the center of coordinates `xc, yc, zc` and radius `r`.

```
function OnSphere(xc, yc, zc, r, x, y, z)
// in: xc, yc, zc, r
// in-out: x, y, z
// purpose: computes the adapted coordinates for a vertex lying on a
//          sphere of center (xc, yc, zc) and radius r.
  // rp is the distance between the old vertex and the sphere center.
  rp := sqrt((xc-x)^2 + (yc-y)^2 + (zc-z)^2);
  x := x*r/rp; y := y*r/rp; z := z*r/rp;
```

```
end function OnSphere
```

Function `Much_D` visits all the edges and puts the new vertex `v` on a side of the edge (`m1,m2`), in the array elements `m[i,j,5]` or `m[i,j,6]`. The function is used in the `Dual` and the `Leapfrog` operations.

```
function Much_D(v, m1, m2, n, m)
// in: v, n, m1, m2
// in-out: m
// purpose: puts the new vertex v on a side of the edge (m1, m2).
  for i := 1 to n
    for j := 1 to m[i,0]
      if ((m[i,j,1] = m1) and m[i,j,2] = m2)) or
         ((m[i,j,2] = m1) and m[i,j,1] = m2)) then
        if m[i,j,5] = 0 then {
          m[i,j,5] := v;
          m[i,j,6] := 0;     }
        else
          m[i,j,6] := v;
        endif
    endfor j
  endfor i
end function Much_D
```

The function `AdLeg` puts the link (`from, to`) in the array `med` and concatenates the corresponding strings.

```
function AdLeg(from, to, med)
// in: from, to
// in-out: med
// purpose: puts in the array med the link from 'from' to 'to' vertices.
  med[from, 2] := med[from, 2] + 1;
  loc := med[from, 2];
  med[from, loc+2] := to & " a";
end function AdLeg
```

The function `ad2l` links in both directions the vertices `l1` and `l2`.

```
function ad2l(l1,l2,med)
// in: l1, l2
// in-out: med
// purpose: links the given vertices in both directions
  adleg(l1,l2,med);
  adleg(l2,l1,med);
end function ad2l
```

The function `AdMe` links the vertices lying in the middle of the edges (`m1`, `m2`) and (`m2`, `m3`). It is invoked in the `Medial` operation.

```
function AdMe(m1, m2, m3, n, m, med)
// in: m1, m2, m3, n, m
// in-out: med
// purpose: links the vertices stored in the m[k,kk,5] array elements of
//          the edges (m1, m2) and (m2, m3).
  for k := 1 to n
    for kk := 1 to m[k,0]
      if ((m1 = m[k,kk,1]) and (m2 = m[k,kk,2])) or
         ((m1 = m[k,kk,2]) and (m2 = m[k,kk,1]))
```

```
        leg1 := m[k,kk,5];
      if ((m2 = m[k,kk,1]) and (m3 = m[k,kk,2])) or
         ((m2 = m[k,kk,2]) and (m3 = m[k,kk,1]))
        leg2 := m[k,kk,5];
    endfor kk
    ad2l(leg1, leg2, med);    // links vertices leg1 and leg2
  endfor k                    // in both directions.
end function AdMe
```

Function `Center_D` returns the coordinates `x, y, z` of the new vertex `nn`, the center of the face `c[i,j]`. The function is called in the `Leapfrog` and `Capra` operations.

```
function Center_D(i, j, nn, a, tipa, x, y, z, c, st)
// in: i, j, a, tipa
// out: x, y, z
// in-out: nn, c, st
// purpose: computes the coordinates of the new vertex which is the
//          center of the face c[i,j] and stores the new vertex in
//         c[i,j,c[i,j,0]+1].
  x := 0; y := 0; z := 0;
  nn := nn + 1;
  st[nn,1] := "atom " + nn + " - " + tipa + " CA - 0 ";
  for k := 1 to c[i, j, 0]
    x := x + a[c[i,j,k]],2];  // the coordinates of the center of the
    y := y + a[c[i,j,k]],3];  // face c[i,j] is the average of the
    z := z + a[c[i,j,k]],4];  // coordinates of the boundary vertices.
  endfor k
  x := x/c[i,j,0]; y := y/c[i,j,0]; z := z/c[i,j,0];
  st[nn,1] := st[nn, 1] + x + " " + y + " " + z + " ";
  c[i,j,c[i,j,0]+1] := nn;
end function Center_D
```

Function `Ini` concatenates the information about valences, old coordinates and old adjacent vertices to the string `st`, in order to preserve the old vertices in the output file. It is called in the `Stellation` operation.

```
function Ini(a, n, tipa, st)
// in: a, n, tipa
// in-out: st
// purpose: updates the string st with the valences, old coordinates and
//          old adjacent vertices info.
  for i := 1 to n
    st[i, 2] := a[i,8];      // old valence
    // old coords
    st[i, 1] := "atom " + i + " - " + tipa + " CA - 0 " +
                a[i, 2] + " " + a[i, 3] + " " + a[i, 4] + " ";
    k := 0;
    for j := 9 to a[i,8]*2 + 8 step 2
      k := k + 1;            // old adjacent vertices of the vertex i
      st[i, k+2] := a[i,j] + " a";
    endfor j
  endfor i
end function Ini
```

Function `Ad_CoordT` computes the coordinates (`x, y, z`) of a new vertex lying on the edge `m[i,j]` and puts the new vertex with the valency 0 in the final array `tr`. The parameter k is either 2 or 0.5, defining the position of the new vertex with respect to the ends of the edge. This function is used in the Truncation and the Capra operations.

```
function Ad_CoordT(m, i, j, k, x, y, z, nn, tr, tipa, a)
// in: m, i, j, k, tipa, a
// out: x, y, z
// in-out: nn, tr
// purpose: computes the coordinates (x,y,z) of a new vertex lying on the
//          edge m[i,j]. There are two new vertices trisecting the edge.
//          The parameter k defines the position of the new vertex with
//          respect to the edge bounds. Valid values are 2 or 0.5.
  nn := nn + 1;              // the number of vertices in the new graph
  tr[nn,2] := 0;
  tr[nn,1] := "atom " + nn + " - " + tipa + " CA - 0 ";
  x := (a[m[i,j,1],2] + k * a[m[i,j,2],2])/(1+k);
  y := (a[m[i,j,1],3] + k * a[m[i,j,2],3])/(1+k);
  z := (a[m[i,j,1],4] + k * a[m[i,j,2],4])/(1+k);
  tr[nn,1] := tr[nn,1] + x + " " + y + " " + z + " ";
end function Ad_CoordT
```

Function Ad_P puts two new vertices on the edge m[i,j] by calling Ad_CoordT twice with k = 2 and k = 0.5, and stores them in m[i,j,5] and m[i,j,6]. Both Ad_P and AdTr functions are invoked in the Truncation operation.

```
function Ad_P(tr, m, i, j, nn, tipa, a)
// in: i, j, tipa, a
// in-out: tr, m, nn
// purpose: puts two new vertices on the edge m[i,j] by calling Ad_CoordT
  Ad_CoordT(m,i,j,2,x,y,z,nn,tr,tipa,a); // the vertex closer to m[i,j,2]
  m[i,j,5] := nn;
  Ad_CoordT(m,i,j,0.5,x,y,z,nn,tr,tipa,a)// the vertex closer to m[i,j,1]
  m[i,j,6] := nn;
  ad2l(nn-1,nn,tr);  // links the new vertices related to the old edge
end function Ad_P    // m[i,j], in the final array tr.
```

Function AdTr adds a link between the two new vertices located on the edges (m1, m2) and (m2, m3), nearer to the vertex m2.

```
function AdTr(m1, m2, m3, n, m, tr)
// in: m1, m2, m3, n, m
// in-out: tr
// purpose: links the two new vertices located on the edges (m1, m2) and
//          (m2, m3), that is closer to the vertex m2.
  for k := 1 to n
    for kk := 1 to m[k,0]
      if ((m1 = m[k,kk,1]) and (m2 = m[k,kk,2])) leg1 := m[k,kk,5];
      if ((m1 = m[k,kk,2]) and (m2 = m[k,kk,1])) leg1 := m[k,kk,6];
      if ((m2 = m[k,kk,1]) and (m3 = m[k,kk,2])) leg2 := m[k,kk,6];
      if ((m2 = m[k,kk,2]) and (m3 = m[k,kk,1])) leg2 := m[k,kk,5];
    endfor kk
    ad2l(leg1, leg2, tr);    // links the vertices leg1 and leg2
  endfor k                   // in both directions.
end function AdTr
```

The function `Ad_Coord` sums up the coordinates of the vertex `vc` in the corresponding sum variables `x`, `y`, and `z`. The function is called by the `P_New` function.

```
function Ad_Coord(x, y, z, vc, a)
// in: vc, a
// in-out: x, y, z
// purpose: sums the coordinates of a[vc] into x, y, and z.
  x := x + a[vc,2];
  y := y + a[vc,3];
  z := z + a[vc,4];
end function Ad_Coord
```

Function `P_New` puts a new vertex inside the face `c[i,j]`, in the final array `leaf`, near the edge (`c[i,j,k]`, `c[i,j,kk]`). It is used in the `Leapfrog` and `Capra` operations.

```
function P_New(i, j, k, kk, a, c, tipa, xf, yf, zf, nn, leaf, c2)
// in: i, j, k, kk, a, c, tipa, xf, yf, zf,
// in-out: nn, leaf, c2
// purpose: adds a new vertex inside the face c[i,j] in the array leaf,
//          near the edge (c[i,j,k], c[i,j,kk]).
//          (xf, yf, zf) define the center of the face c[i,j].
  nn := nn + 1;     // new vertex
  leaf[nn, 1] := "atom " + nn + " - " + tipa + " CA - 0 ";
  x := 0; y := 0; z := 0;
  Ad_Coord(x, y, z, c[i,j,k], a);
  Ad_Coord(x, y, z, c[i,j,kk], a);
  x = (x + xf)/3;   // Computes the coordinates x,y and z as the mean of
  y = (y + yf)/3;   // the coordinates of the two ends of the edge
  z = (z + zf)/3;   // (c[i,j,k), c[i,j,kk)) and the face center.
  leaf[nn,1] := leaf[nn,1] + x + " " + y + " " + z + " ";
  c2[i,j,k] := nn;
  leaf[nn,2] := 0;
end function P_New
```

Function `IniQ` puts the coordinates of old vertices into the `qa` string. It is invoked in the `Quadrupling` and `Capra` operations.

```
function IniQ(a, n, tipa, qa)
// in: a, n, tipa
// in-out: qa
// purpose: initializes the array qa with the old vertices.
  for i := 1 to n
    qa[i,1] := "atom " + i + " - " + tipa + " CA - 0 ";
    qa[i,1] := qa[i,1] + a[i,2] + " " + a[i,3] + " " + a[i,4] + " ";
    qa[i,2] := 0;
  endfor i
end function IniQ
```

Function `WriteF` writes the final version of the array `qa` to the output file referred by the file pointer `fp`.

```
function WriteF(fp, n, qa)
// in: fp, n, qa
// purpose: writes the array qa to the text file referred by fp.
  for i := 1 to n                  // + means string concatenation.
    print fp, qa[i,1] + qa[i,2];  // writes the coordinates and
    for j := 1 to qa[i,2]         // valence of the vertex i.
```

```
      print fp, " " + qa[i,j+2];  // Writes the linked vertices.
    endfor j
    print fp, newLine;            // Writes the line terminator(s).
  endfor i
  print fp, "endmol 1";
end function WriteF
```

Function `MeanQ` returns the coordinates of the new vertex corresponding to `p2` as the weighted average of three consecutive vertices `p1`, `p2` and `p3` in the final string `strc`. This function is used in the `AdCoordQ` function.

```
function MeanQ(a, p1, p2, p3, strc)
// in: a, p1, p2, p3
// out: strc
// purpose: computes the coordinates of a new vertex as weighted average
//          of the coordinates of vertices p1, p2, p3 and returns them
//          in the string strc.
  x := (a[p1,2] + a[p2,2]*2 + a[p3,2])/4;
  y := (a[p1,3] + a[p2,3]*2 + a[p3,3])/4;
  z := (a[p1,4] + a[p2,4]*2 + a[p3,4])/4;
  strc := x + " " + y + " " + z + " ";
end function MeanQ
```

Function `AdCoordQ` puts the vertex coordinates of the new cycle inside the old cycle `c[i,j]`. `AdCoordQ` and `AdLegQ` functions are called by the `AdCycleInt` function, in the `Quadrupling` operation. They update the final string `qa`.

```
function AdCoordQ(a, c, no, i, j, qa)
// in: a, c, no, i, j
// in-out: qa
// purpose: puts the coordinates of the new cycle inside the old cycle
//          c[i,j].
  lc := c[i,j,0];
  n := no - lc;
  MeanQ(a, c[i,j,lc], c[i,j,1], c[i,j,2], strc);
  qa[n+1,1] := qa[n+1,1] + strc;  // the coords of the first vertex
  MeanQ(a, c[i,j,lc-1], c[i,j,lc], c[i,j,1], strc);
  qa[no,1] := qa[no,1] + strc;    // the coords of the last vertex
  for k := 2 to c[i,j,0]-1
    MeanQ(a, c[i,j,k-1], c[i,j,k], c[i,j,k+1], strc);
    qa[n+k,1] := qa[n+1,1] + strc;  // the coords of the other vertices
  endfor k
end function AdCoordQ
```

Function `AdLegQ` adds the edges of the new cycle located inside the old cycle `c[i,j]`.

```
function AdLegQ(a, c, no, i, j, qa)
// in: a, c, no, i, j
// in-out: qa
// purpose: adds the edges of the new cycle located inside the old
//          cycle c[i,j].
  lc := c[i,j,0];      // the size of the cycle
  p1 := no - lc + 1;   // the first vertex in the cycle
  // Adds the links between the vertices of the new cycle.
  ad2l(p1, no, qa);
  for k := 1 to c[i,j,0]-1
    ad2l(p1+k-1, p1+k, qa);
  endfor k
  // Adds the links between the vertices of the new cycle and the
  // corresponding vertices of the old cycle.
  for k := 1 to c[i,j,0]
    Ad2l(p1+k-1, c[i,j,k], qa);
  endfor k
end function AdLegQ
```

Function `AdCycleInt` adds a similar inner cycle for every old cycle and puts the edges for the new cycle between the new and corresponding old vertices. It implements an important step in the `Quadrupling` operation.

```
function AdCycleInt(a, n, c, tipa, qa, no)
// in: a, n, c, tipa
// out: no
// in-out: qa
// purpose: adds a similar inner cycle for every old cycle and updates
            the links between the new and the corresponding old vertices.
  no := n;                      // The old vertices are kept.
  for i := 1 to n
    for j := 1 to c[i,0]        // for every cycle c[i,j]
      for k := 1 to c[i,j,0]    // for every vertex c[i,j,k] of the cycle
        no := no + 1;           // a new vertex to be added
        qa[no, 1] := "atom " + no + " - " + tipa + " CA - 0 ";
        qa[no, 2] := 0;
      endfor k
      AdCoordQ(a, c, no, i, j, qa);
      AdLegQ(a, c, no, i, j, qa);
    endfor j
  endfor i
end function AdCycleInt
```

Function `P_New_Capra` puts a new vertex inside the face `c[i,j]`, in the array `leaf`, close to the edge (`c[i,j,k]`, `c[i,j,kk]`). Unlike the function `P_New`, the new vertex will be selected. This function is called by the `Ad_P_Middle` function, described below.

```
function P_New_Capra(i, j, k, kk, a, c, tipa, xf, yf, zf, nn, cip, c2)
// in: i, j, k, kk, a, c, tipa, xf, yf, zf
// in-out: nn, cip, c2
// purpose: adds a new vertex inside the face c[i,j] in the array cip,
//          close to the edge (c[i,j,k], c[i,j,kk]).
//          (xf, yf, zf) define the center of the face c[i,j]
  P_New(i, j, k, kk, a, c, tipa, xf, yf, zf, nn, cip, c2);
  // the vertices within the new cycle are marked by 's'.
  cip[nn,1] := replace("CA - 0", "CA s 0", cip[nn,1]); // string repl.
end function P_New_Capra
```

Function `Ad_P_Middle` adds a new cycle, of the same folding, in the center of old cycle `c[i,j]`. This function is used by the `Capra` operation.

```
function Ad_P_Middle(cip, m, i, j, nn, c, x, y, z, a, tipa, c2)
// in: m, i, j, x, y, z, a, tipa
// in-out: cip, nn, c, c2
// purpose: adds a new cycle in the center of the old cycle c[i,j].
```

```
  nn := nn - 1;  // Removes the vertex from the center.
  nv = c[i,j,0];
  c[i,j,nv+2] := "x"; // Marks the visited cycle.
  // Adds the vertices of the new cycle inside the old cycle.
  for k := 1 to nv-1
    P_New_Capra(i, j, k, k+1, a, c, tipa, x, y, z, nn, cip, c2);
  endfor k
  P_New_Capra(i, j, nv, 1, a, c, tipa, x, y, z, nn, cip, c2);
  // Adds edges of the new cycle.
  for k := 1 to nv-1
    Ad2l(c2[i,j,k], c2[i,j,k+1], cip);
  endfor k
  Ad2l(c2[i,j,1], c2[i,j,nv], cip);
end function Ad_P_Middle
```

All the functions described below in this section are invoked by the `Capra` operation. The function `Inv56` returns the value `5` if receives the value `6`, and `6` otherwise.

```
function Inv56(i)
// in: i
// purpose: if the argument has the value 6, returns 5;
//          otherwise returns 6.
  if i = 6
    then return 5
    else return 6
  endif
end function Inv56
```

The function `Verif_M2` returns the position (of the added points on the edges) put by the last link.

```
function Verif_M2(m, i, j, c)
// in: m, i, j, c
// purpose: returns the position of vertex put by the last link
//          if no link is found, returns 0.
  for k := 1 to c[i,j,0]        // Verifies if the links exist
    ii := c[i,j,k];             // from the central cycle at the first or
    if k < c[i,j,0]             // second vertex on the old edge.
      then iiu := c[i,j,k+1];
      else iiu := c[i,j,1];
    endif
    for jj := 1 to m[ii,0]
      if m[ii,jj,2] = iiu then     // for the edge ii, iiu
        if m[ii,jj,7] > 0 then     // If the link was found
        // returns the position of the last link on the edge m[ii,jj].
            return m[ii,jj,7];
      endif
    endfor jj
    for jj := 1 to m[iiu, 0]
      if m[iiu,jj,2] = ii then     // for the edge iiu, ii
        if m[iiu,jj,7] > 0 then    // If the link was found
        // returns the position of the last link on the edge m[ii,jj].
            return Inv56(m[iiu,jj,7]);
      endif
    endfor jj
  endfor k
  return 0;
```

```
end function Verif_M2
```

Function `Verif_M` puts a link between the central cycle and a new vertex on the edges.

```
function Verif_M(cip, m, ii, iiu, m56, pc)
// in: ii, iiu, m56, pc
// in-out: cip, m
// purpose: puts the link from the vertex pc of the central cycle to
//          a vertex from the edge ii - iiu, in the final array cip.
  for jj := 1 to m[ii, 0]
    if m[ii,jj,2] = iiu then        // for the edge ii, iiu
      ad2l(m[ii,jj,m56], pc, cip);  // links the vertices
      m[ii,jj,7] := m56;
    endif
  endfor jj
end function Verif_M
```

Function `AlreadyVisited` checks if the cycle was already visited or it shares an edge with a visited cycle.

```
function AlreadyVisited(c, i, j, m)
// in: c, i, j, m
// purpose: returns true (0) if
//   a) the cycle was already visited
//   b) the cycle shares an edge with an already visited cycle.
  nv = c[i,j,0];
  if c[i,j,nv+2] = "x" then return true; //  The cycle was visited.
  if Verif_M2(m, i, j, c) = 0 then return true;
  return false;
end function AlreadyVisited
```

The function `Nrc` counts the number of cycles.

```
function Nrc(c, n, nci)
// in: c, n
// out: nci
// purpose: returns the number of cycles.
  nci := 0;
  for i := 1 to n
    for j := 1 to c[i,0]
      nci := nci + 1;
    endfor j
  endfor i
end function Nrc
```

Function `AdLeg_C` adds edges connecting the vertices of the new cycle lying in the center of the old cycle, to the old cycle, `c[i,j]`.

```
function AdLeg_C(cip, m, i, j, nn, c, c2, n, p12)
// in: i, j, nn, c2, n, p12
// in-out: cip, m, c
// purpose: puts the links between the central cycle and the points added
//          on the edges of the old cycle c[i,j].
  nv := c[i,j,0];
  poz1 := Verif_M2(m, i, j, c);
// the position of the last link on the edges
  if poz1 = 0 then poz1 := 4 + p12;
  poz2 := inv56(poz1);
  for k := 1 to c[i,j,0]     // for every vertex c[i,j,k] from c[i,j]
    ii := c[i,j,k];
    if k < c[i,j,0]
      then iiu := c[i,j,k+1]
      else iiu := c[i,j,1]
    endif
    Verif_M(cip, m, ii, iiu, poz2, c2[i,j,k]); // Puts a link to the new
    Verif_M(cip, m, iiu, ii, poz1, c2[i,j,k]); // vertex added on the old
  endfor k                                     // edge ii, iiu.
end function AdLeg_C
```

The function `Ad_Pi` adds two vertices on each link and then links to each other and to the ends of the edge.

```
function Ad_Pi(cip, m, i, j, nn, tipa, a)
// in: i, j, tipa, a
// in-out: cip, m, nn
// purpose: adds two vertices on each link and then links them with
//          the ends of the edge.
  Ad_CoordT(m, i, j, 2, x, y, z, nn, cip, tipa, a);
  ad2l(m[i,j,2], nn, cip);
  m[i,j,6] := nn;
  Ad_CoordT(m, i, j, 0.5, x, y, z, nn, cip, tipa, a);
  m[i,j,5] := nn;
  // Links the new vertices to each other.
  ad2l(nn-1, nn, cip);
  // Links the new vertices with the ends of the edge.
  ad2l(m[i,j,1], nn, cip);
end function Ad_Pi
```

3. Basic Map Operations

3.1. Dual

Function `Dual` operates all the faces of the array `c` by putting a new point in the center of each face. Every two such points are then joined if their corresponding faces share a common edge (Figure 1).

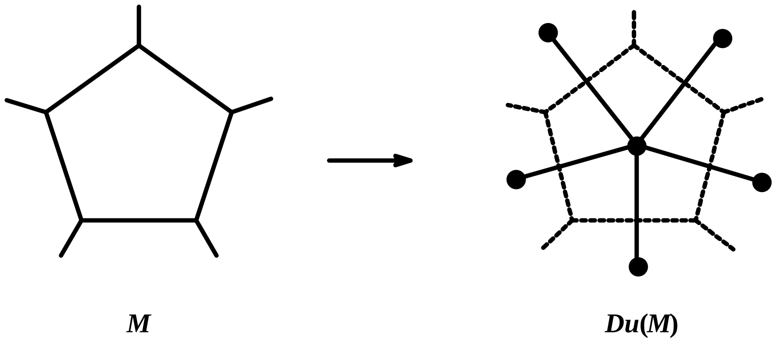


Figure 1: Dual operation.

```
function Dual(a, n, c, fp, m, tipa, sfer, texti)
// in: a, n, c, fp, tipa, sfer, texti
// in-out: m
// purpose: puts a new point in the center of each face
//          joins every two such new points if their corresponding faces
//          share a common edge.
  // textd: the .hin file header
  // newLine is the line terminator.
  textd := "Dual " + newLine + "forcefield mm+" + newLine +
           "sys 0" + texti + "mol 1" + newLine;
  print fp, textd;
  Center(n, a, xc, yc, zc, r);
  nn := 0;  // the number of initial faces = number of the new vertices
  for i := 1 to n  // Visits the faces to compute their center coords.
    for j := 1 to c[i,0];
      nn := nn + 1;
      du[nn,2] := 0;             // Initializes the valences.
      du[nn,1] := "atom " + nn + " - " + tipa + " CA - 0 ";
      x := 0; y := 0; z := 0;
      for k := 1 to c[i,j,0]
        x := x + a[c[i,j,k],2];  // Coordinates x, y, z of the center
        y := y + a[c[i,j,k],3];  // of the face c[i,j]
        z := z + a[c[i,j,k],4];  // are averages of vertex coords.
      endfor k
      x := x / c[i,j,0]; y := y / c[i,j,0]; z := z / c]i,j,0];
      if sfer then               // if the new vertices lye on the sphere
        OnSphere(xc, yc, zc, r, x, y, z);
      endif
      du[nn,1] := du[nn,1] + x + " " + y + " " + z + " ";
      c[i,j, c[i,j,0]+1] := nn;  // the new vertex for the face c[i,j]
    endfor j
  endfor i
  // adds the links in the new graph
  for i := 1 to n
    for j := 1 to m[i,0]
      m[i,j,5] := 0;
    endfor j
  endfor i
  for i := 1 to n
    for j := 1 to c[i,0]
      v := c[i,j,c[i,j,0]+1];
      for k := 1 to c[i,j,0] - 1
```

```
        m1 := c[i,j,k];
        m2 := c[i,j,k+1];
        Much_D(v, m1, m2, n, m); // Puts the new vertex v in the array m.
      endfor k
      Much_D(v, c[i,j,1], c[i,j,c[i,j,0]], n, m);
    endfor j
  endfor i
  for i := 1 to n
    for j := 1 to m[i,0]           // Visits the edges and links the
      if m[i,j,6] > 0 then         // vertices across the shared edge.
        ad2l(m[i,j,5], m[i,j,6], du);
      endif
    endfor j
  endfor i
  // Write the strings from du in the output file.
  WriteF(fp, nn, du);
end function Dual
```

3.2. Medial

By this operation, a new vertex is put in the middle of every old edge and the new vertices are linked if they belong to consecutive edges, within a rotational path around their common vertex. Only the new vertices are retained. Optionally, the new vertices can be embedded on the sphere.

In the medial transform, the number of vertices equals the number of edges in the parent map. Function `Medial` visits the edges and puts a new vertex in the middle of every edge. The local array `med` will contain the new structure/map.

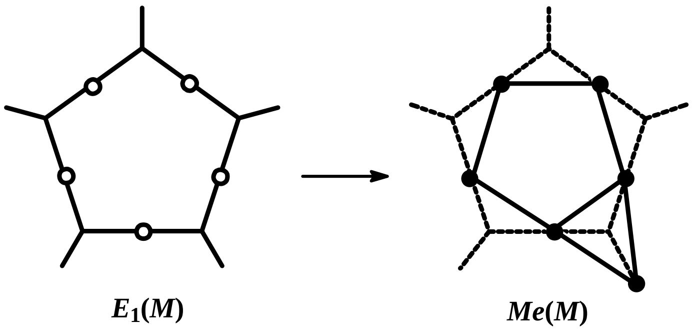


Figure 2: Medial operation

```
function Medial(n, a, c, m, fp, tipa, sfer, texti)
// in: n, a, c, fp, tipa, sfer, texti
// out: m
// purpose: puts a new point in the middle of every old edge
//          the new vertices are linked if they belong to consecutive
//          edges within a rotational path around their common vertex.
  // .hin file header
  // newLine is the line terminator.
  textm := "Medial " + newLine + "forcefield mm+" + newLine +
           "sys 0" + texti + "mol 1" + newLine;
  print fp, textm;
  Center(n, a, xc, yc, zc, r);
  nn := 0;  // the number of resulting vertices
  for i := 1 to n
    for j := 1 to m[i,0]     // For every edge
      nn := nn + 1;          // adds a new vertex.
      med[nn,2] := 0;
      med[nn,1] := "atom " + nn + " - " + tipa + " CA - 0 ";
      // the coords of the edge middle
      x := (a[m[i,j,1],2] + a[m[i,j,2],2])/2;
      y := (a[m[i,j,1],3] + a[m[i,j,2],3])/2;
      z := (a[m[i,j,1],4] + a[m[i,j,2],4])/2;
      if sfer then   // if the new points will lye on the sphere
        OnSphere(xc, yc, zc, r, x, y, z);
      endif
      med[nn,1] := med[nn,1] + x + " " + y + " " + z + " ";
      m[i,j,5] := nn;
    endfor j
  endfor i
  // Add the links to the other new vertices.
  for ii := 1 to n
    for jj := 1 to c[ii,0]  // for each face
      for kk := 1 to c[ii,jj,0] - 2
        // Connects the new vertices located at every consecutive edges.
        // (m1, m2) and (m2, m3)
        m1 := c[ii, jj, kk];
        m2 := c[ii, jj, kk+1];
        m3 := c[ii, jj, kk+2];
        AdMe(m1, m2, m3, n, m, med);
      endfor kk
      // Connects the start and end vertices, closing the ring.
      AdMe(c[ii,jj,c[ii,jj,0]-1],c[ii,jj,c[ii,jj,0]],c[ii,jj,1],n,m,med);
      AdMe(c[ii,jj,c[ii,jj,0]],c[ii,jj,1],c[ii,jj,2],n,m,med);
    endfor jj
  endfor ii
  // Writes the strings from med in the output file.
  WriteF(fp, nn, med);
end function Medial
```

3.3. Stellation

This operation adds a new vertex in the center of every face and connect it with the boundary vertices. The old vertices and edges are kept, and all the faces of the resulting graph are triangles. Function `Stellation` builds the array `st` and writes it in the output file.

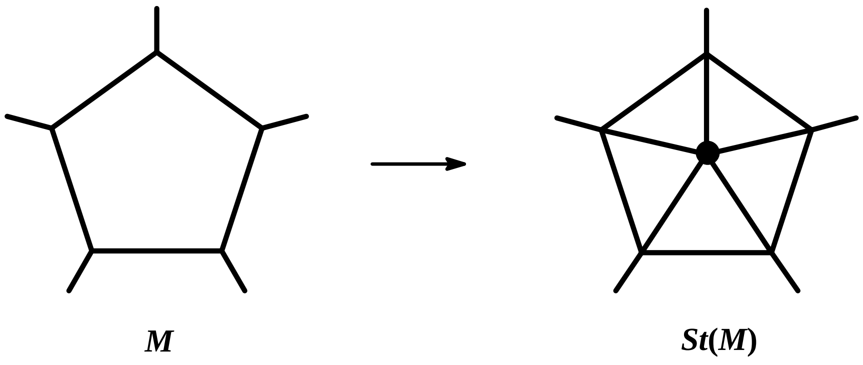


Figure 3: Stellation operation

```
function Stellation(n, a, c, fp, m, tipa, texti)
// in: n, a, c, fp, tipa, texti
// in-out: m
// purpose: adds a new vertex in the center of every face
//          and connects it with the boundary vertices. The old vertices
//          and edges are kept.
  // .hin file header
  // newLine is the line terminator.
  texts := "Stellation " + newLine + "forcefield mm+" + newLine +
           "sys 0" + texti + "mol 1" + newLine;
  print fp, texts;
  Ini(a, n, tipa, st);      // Puts the initial vertices and edges in st.
  nn := n;                  // nn stores the number of final vertices.
  for i := 1 to n
    for j := 1 to c[i,0]    // Visits all parent faces;
      Center_D(i,j,nn,a,tipa,x,y,z,c,st);  // the nn-th vertex is the
      st[nn,2] := 0;                      // center of the c[i,j] face.
      for k := 1 to c[i,j,0]
        ad2l(nn,c[i,j,k],st);  // Joins all boundary vertices to the
      endfor k                 // center of the face.
    endfor j
  endfor i
  // Writes the strings from st in the output file.
  WriteF(fp, nn, st);
end function Stellation
```

3.4. Truncation

This operation puts a new vertex on each edge incident in an old vertex, and joins them around the old vertex, which is finally cut off. In the function `Truncation`, the final vertices are put in the local array `tr`, which is then written in the .hin file.

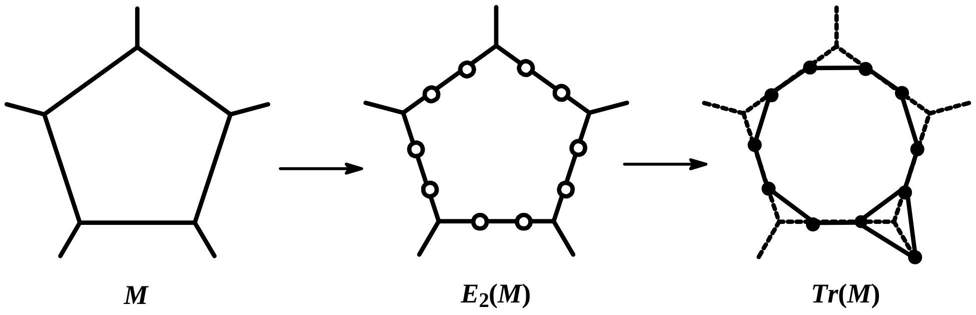


Figure 4: Truncation operation

```
function Truncation (n, a, c, m, fp, tipa, texti)
// in: n, a, c, m, fp, tipa, texti
// purpose: adds a new vertex on each edge around an old vertex
//          and connects them around the old vertex which is cut off;
//         connects the two vertices on the same edge.
  // .hin file header
  // newLine is the line terminator.
  textt := "Truncation " + newLine + "forcefield mm+" + newLine +
           "sys 0" + texti + "mol 1" + newLine;
  print fp, textt;
  nn := 0;  // the number of final vertices
  for i := 1 to n
    for j := 1 to m[i,0]                    // On each edge two vertices
      Ad_P(tr, m, i, j, nn, tipa, a);       // are added and linked.
    endfor j
  endfor i
  for ii := 1 to n
    for jj := 1 to c[ii,0]
      for kk := 1 to c[ii,jj,0]-2           // for each cycle c[ii,jj]
        m1 := c[ii,jj,kk];        // Connects the new vertices
        m2 := c[ii,jj,kk+1];      // lying on the edges (m1,m2) and (m2,m3)
        m3 := c[ii,jj,kk+2];      // around the old vertex m2.
        AdTr(m1, m2, m3, n, m, tr);
      endfor kk
      AdTr(c[ii,jj,c[ii,jj,0]-1],c[ii,jj,c[ii,jj,0]],c[ii,jj,1],n,m,tr);
      AdTr(c[ii,jj,c[ii,jj,0]],c[ii,jj,1],c[ii,jj,2],n,m,tr);
    endfor jj
  endfor ii
  // Writes the strings from array tr in the output file.
  WriteF(fp, nn, tr);
end function Truncation
```

3.5. Leapfrog

This operation puts vertices on both sides of the old edges. Next, it links these new vertices inside of each face if they correspond to consecutive edges and then links every two vertices from different faces if they correspond to the same edge. Only the new vertices and edges remain in the resulting graph. During the construction, the local array `leaf` contains information about new vertices, which is written in the .hin file.

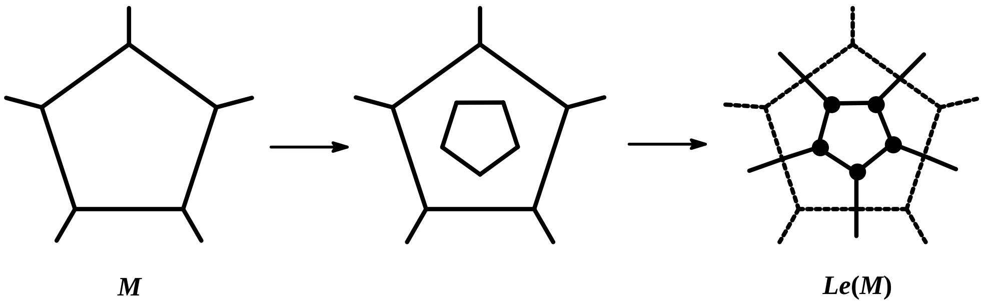


Figure 5: Leapfrog operation.

```
function Leapfrog(a, n, c, fp, m, tipa, texti)
// in: a, n, c, fp, tipa, texti
// in-out: m
// purpose: puts vertices on both sides of the old edges;
//          links the new vertices inside the faces if they correspond to
//          consecutive edges and links every two vertices from different
//          faces if they correspond to the same edge;
//           only the new vertices remain.
  // .hin file header
  // newLine is the line terminator.
  textl := "Leap Frog " + newLine + "forcefield mm+" + newLine +
           "sys 0" + texti + "mol 1" + newLine;
  print fp, textl;
  nn := 0;                   // the number of new vertices
  nf := 0;                   // the number of new faces
  for i := 1 to n
    for j := 1 to c[i,0]     // For every face c[i,j]
      // computes the coordinates (xf,yf,zf) of the face center.
      Center_D(i,j,nf,a,tipa,xf,yf,zf,c,st);
      nv := c[i,j,0];        // the number of vertices in the face c[i,j]
      v := c[i,j,nv+1];
      for k := 1 to nv-1
      // For each boundary edge, put in c2 a new vertex inside the face.
        P_New(i,j,k,k+1,a,c,tipa,xf,yf,zf,nn,leaf,c2);
      endfor k
      P_New(i,j,nv,1,a,c,tipa,xf,yf,zf,nn,leaf,c2);
      for k := 1 to nv-1
        ad2l(c2[i,j,k],c2[i,j,k+1],leaf);   // Adds the links between the
        m1 := c[i,j,k];                     // new vertices c2[i,j,k],
        m2 := c[i,j,k+1];                   // c2[i,j,k+1] inside the
        Much_D(c2[i,j,k],m1,m2,n,m);        // face c[i,j].
      endfor k
      ad2l(c2[i,j,1],c2[i,j,nv],leaf);     // Links vertices at the end
      Much_D(c2[i,j,nv],c[i,j,1],c[i,j,nv],n,m); // of the cycle c[i,j].
    endfor j
  endfor i
  for i := 1 to n
    for j := 1 to m[i,0]
```

```
      if m[i,j,6] > 0 then
        ad2l(m[i,j,5],m[i,j,6], leaf);
      endif
    endfor j
  endfor i
// Writes the strings from leaf in the output file.
  WriteF(fp, nn, leaf);
end function Leapfrog
```

3.6. Quadrupling

This operation adds a new cycle inside of each cycle/face and connects the two cycles vertex by vertex. Next, the old edges are deleted. The transformation preserves the initial orientation of all parent faces in the map as well as the initial vertices (Figure 6).

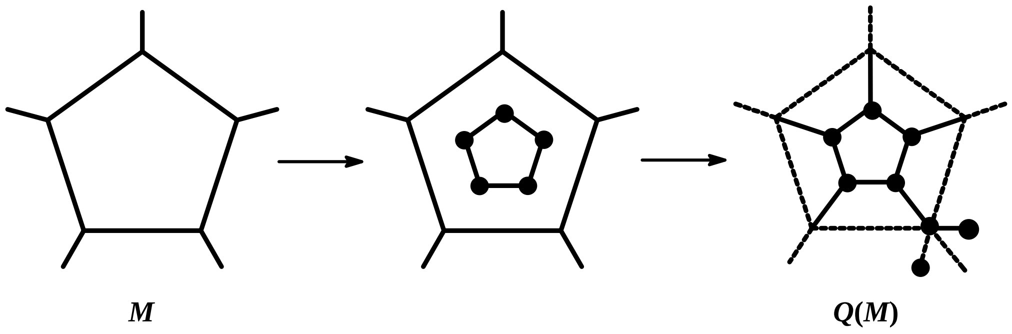


Figure 6: Quadrupling operation

```
function Quadrupling (a, n, c, fp, tipa, texti)
// in: a, n, c, fp, tipa, texti
// purpose: adds a new cycle inside of each cycle/face and connects the
//          two cycles vertex by vertex.
//          The transformation preserves the initial orientation of all
//          parent faces. The initial vertices are preserved while the
//          old edges are removed.
  // .hin file header
  // newLine is the line terminator.
  textq := "Quadrupling " + newLine + "forcefield mm+" + newLine +
           "sys 0" + texti + "mol 1" + newLine;
  print fp, textq;
  IniQ(a,n,tipa, qa); // Puts in qa the initial vertices without edges.
  AdCycleInt(a,n,c,tipa,qa,no);
  // Writes the strings from qa in the output file.
  WriteF(fp, no, qa);
end function Quadrupling
```

3.7. Capra

This transformation puts two new points on each edge of the map. Next, puts a new cycle of the same folding in the center of each face and makes (1, 4) connections to the boundary points, starting with a new vertex[9,10].

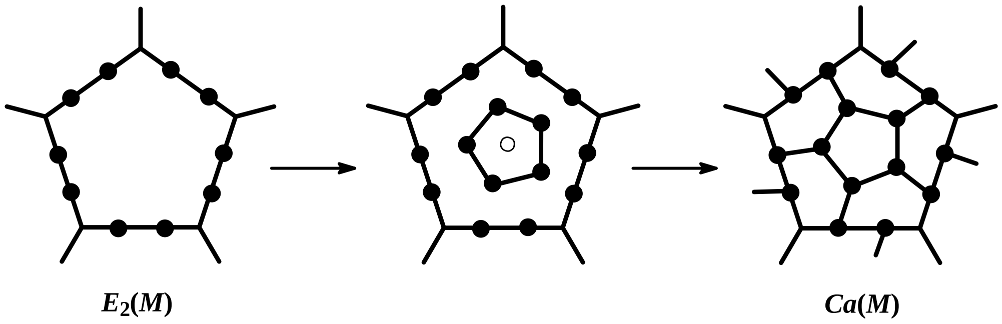


Figure 5: Leapfrog operation

```
function Capra(a, n, c, m, fp, tipa, texti, p12)
// in: a, n, c, m, fp, tipa, texti, p12
// purpose: puts two new points on each edge of the map.
//          Puts a new cycle of the same size in the center of each face
//          and makes (1,4) connections to the boundary points,
//          starting with a new vertex.
  // .hin file header
  // newLine is the line terminator.
  textc := "Capra " + newLine + "forcefield mm+" + newLine +
           "sys 0" + texti + "mol 1" + newLine;
  print fp, textc;
  Nrc(c, n, nci);
  IniQ(a,n,tipa,cip);        // Puts in the final array cip the initial
  nn := n;                   // vertices without edges.
  for i := 1 to n            // Initialization
    for j := 1 to m[i,0]
      m[i,j,7] := "";        // for vertices related to the edges
    endfor j
    for j := 1 to c[i,0]     // and for vertices related to the cycles.
      nv := c[i,j,0];
      c[i,j,nv+2] := "";
    endfor j
  endfor i
  for i := 1 to n
    for j := 1 to m[i,0]
      Ad_Pi(cip,m,i,j,nn,tipa,a);  // Adds two new linked vertices on
    endfor j                       // every old edge.
  endfor i
  nc := 0;
  do while nc < nci                // do while exists a cycle not visited
    for ix := 1 to n
      for jx := 1 to c[ix,0]
        i := ix; j := jx;
        if (nc = 0) or (AlreadyVisited(c,i,j,m)) then
          // if it is the first cycle or the current cycle shares an edge
          // with an already visited cycle
          nc := nc + 1;
          v := c[i,j,0];
          // Adds a vertex in the face center; it will be deleted later.
          Center_D(i,j,nn,a,tipa,x,y,z,c,c3);
          c[i,j,0] := v;
          // Adds a rotated new cycle in the middle of the old cycle.
          Ad_P_Middle(cip,m,i,j,nn,c,x,y,z,a,tipa,c2);
          // Adds the links between the new cycle and the vertices
          // added on the old edges.
         AdLeg_C(cip,m,i,j,nn,c,c2,n,p12);
        endif
      endfor jx
    endfor ix
  loop
  // Writes the strings from cip in the output file.
  WriteF(fp, no, cip);
end function Capra
```

4. Conclusions

Based on the Graph Theory, a self-consistent software program was developed. Its user interface lists all the faces of an input map/polyhedron, and allows the user to select the above-described map operations in order to obtain transformed maps. The program was tested on objects having up to thousands points, both closed and open.

References

1. B. Grünbaum and G. C. Shephard, *Tilings and Patterns*, Freeman, New York, **1985.**
2. D. J. Klein and H. Zhu, in: A. T. Balaban, (Ed.), *From Chemical Topology to Three - Dimensional Geometry*, Plenum Press, New York, **1997**, pp. 297-341.
3. B. de La Vaissière, P. W. Fowler, and M. Deza, *J. Chem. Inf. Comput. Sci.*, **2001**, *41*, 376-386.
4. M. Deza, P. W. Fowler, M. Shtorgin, and K. Vietze, Codes in Platonic, Archimedean, Catalan, and Related Polyhedra: A Model for Maximum Addition Patterns in Chemical Cages, *J. Chem. Inf. Comput. Sci.*, **2000**, *40*, 1325-1332.
5. T. Pisanski, and M. Randić, in *Geometry at Work*, M. A. A. Notes, **2000**, *53*, 174-194.
6. P. W. Fowler and T. Pisanski, *J. Chem. Soc. Faraday Trans*. **1994**, *90*, 2865-2871.
7. F. Harary, *Graph Theory*. Addison-Wesley, Reading, MA, **1969**.
8. Diudea, M. V.; John, P. E.; Graovac,A.; Primorac, M.; Pisanski, T. Leapfrog and Related Operations on Toroidal Fullerenes. *Croat. Chem. Acta*, **2003**, *76*, 153-159.
9. M. V. Diudea, Covering Nanostructures, in: M. V. Diudea, Ed., *Nanostructures-Novel Architecture*, NOVA, New York, **2004** (in press).
10. M. V. Diudea, *Studia Univ.“Babes-Bolyai”*, **2003**, 48 (2), 3-16.