UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO
http://www.icmc.usp.br
Departamento de Ciências de Computação

SCC-211 - Capítulo 12
Grids

João Luís Garcia Rosa[1]

[1] Departamento de Ciências de Computação
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo - São Carlos
`http://www.icmc.usp.br/~joaoluis`

2011

Sumário

Grids: mais comuns do que se pensa

- *Grids* são a base para várias estruturas naturais.
- Tabuleiros de xadrez são *grids*.
- As quadras de uma cidade são tipicamente arranjadas em uma *grid*.
- De fato, a medida de distância mais natural - a distância de Manhattan - pressupõe uma *grid*.
- O sistema de longitudes e latitudes define uma *grid* sobre a Terra, ainda que na superfície de uma esfera ao invés de um plano.

Grids: mais comuns do que se pensa

- *Grids* estão em todo lugar porque são a forma mais natural de dividir espaços em regiões tal que as porções possam ser identificadas.
- No limite, essas células podem ser pontos individuais, mas está-se interessado em *grids* maiores cujas células são grandes o suficiente para ter uma forma.
- Em *grids* regulares, cada uma das formas é idêntica e ocorre em um padrão regular.
- Subdivisões retangulares e retílineas são as *grids* mais comuns, devido a sua simplicidade, mas *grids* hexagonais baseadas em triângulos também são importantes.

Material do Steven Skiena

Os próximos 14 slides contêm material de Steven Skiena disponíveis em [2].

Rectilinear Grids

Rectilinear grids are typically defined by regularly spaced horizontal and vertical lines.

There are three important components of the planar grid: the vertices, the edges, and the cell interiors. Sometimes we are interested in the interiors of the cells, as in geometric applications where each cell describes a region in space. Sometimes we are interested in the vertices of the grid, such as in addressing the pieces on a chessboard. Sometimes we are interested in the edges of the grid, such as when finding routes to travel in a city where buildings occupy the interior of the cells.

Vertices in planar grids each touch four edges and the interiors of four cells, except for vertices on the boundaries. Vertices in 3D grids touch on six edges and eight cells. In d-dimensions, each vertex touches $2d$ edges and 2^d cells. Cells in a planar grid each touch eight faces, four diagonally through vertices and four through edges. Cells in a 3D grid each touch 26 other cells, sharing a face with 6 of them, an edge with 12 of them, and just a vertex with the other 8.

Traversal

It is often necessary to traverse all the cells of an $n \times m$ rectilinear grid. Any such traversal can be thought of as a mapping from each of the nm ordered pairs to a unique integer from 1 to nm.

The most important traversal methods are —

- *Row Major* – Here we slice the matrix between rows, so the first m elements visited belong to the first row, the second m elements to the second row, and so forth.
- *Column Major* – Here we slice the matrix between columns, so the first n elements belong to the first column, the second n elements to the second column, and so forth. This can be done by interchanging the order of

the nested loops from row-major ordering. Knowing whether your compiler uses row-major or column-major ordering for matrices is important when optimizing for cache performance and when attempting certain pointer-arithmetic operations.

- *Diagonal Order* – Here we march up and down diagonals. Note that an $n \times m$ grid has $m+n-1$ diagonals, each with a variable number of elements. This is a trickier task than it appears at first glance.

Dual Graphs and Representations

Two-dimensional arrays are the natural choice to represent planar rectilinear grids. We can let `m[i][j]` denote either the (i, j)th vertex or the (i, j)th face, depending on which we are interested in. The four neighbors of any cell follow by adding ± 1 to either of the coordinates.

A useful concept in thinking about problems on planar subdivisions is that of the *dual graph*, which has one vertex for each region in the subdivision, and edges between the vertices of any two regions which are neighbors of each other. Observe that the dual graphs of both rectangular and hexagonal lattices are slightly smaller rectangular and hexagonal lattices. This is why whatever structure we use to represent vertex connectivities can also be used to represent face connectivities.

An adjacency representation is the natural way to represent an edge-weighted rectilinear grid. This might be most easily done by creating a three-dimensional array `m[i][j][d]`, where d ranges over four values (north, east, south, and west) which denote the edge directions from point (i, j).

Triangular Lattices

Triangular lattices are constructed from three sets of equally spaced lines, consisting of a horizontal “row” axis, a “column” axis 60^{o} from horizontal, and a “diagonal” axis 120^{o} from horizontal.

Vertices of this lattice are formed by the intersection of three axis lines, so each face of the lattice is an equilateral triangle. Each vertex v is connected to six others, those immediately above and below v on each of the three axes.

To identify the proper neighbors of each vertex requires keeping track of two types of coordinate systems:

- *Triangular/Hexagonal Coordinates* – Here, one vertex is designated as the origin of the grid, point $(0, 0)$. We must assign the coordinates such that the logical neighbors of each vertex are easily obtainable. In a standard rectilinear coordinate system, the four neighbors of (x, y) follow by adding ± 1 to either the row or column coordinates.

Although the intersection of three lines defines each grid vertex, in fact the row and column dimensions to specify location.

A vertex (x, y) lies x rows above the origin, and y (60^{o})-columns to the right of the origin. The neighbors of a vertex v can be found by adding the following pairs to the coordinates of v, in counterclockwise order: $(0, 1)$, $(1, 0)$, $(1, -1)$, $(0, -1)$, $(-1, 0)$, and $(-1, 1)$.

- *Geometrical Coordinates* – The vertices of a regular triangular grid occur in half-staggered rows.

Assume that each lattice point is a distance d from its six nearest neighbors, and that point $(0, 0)$ in triangular coordinates in fact lies at geometric point $(0, 0)$. Then triangular-coordinate point (x_t, y_t) must lie at geometric point

$$(x_g, y_g) = (d(x_t + (y_t \cos(60^\text{o}))), dy_t \sin(60^\text{o}))$$

by simple trigonometry, where $\cos(60^\text{o}) = 1/2$ and $\sin(60^\text{o}) = \sqrt{3}/2$,

Hexagonal Lattices

Deleting every other vertex from a triangular lattice leaves us with a *hexagonal* lattice. Now the faces of the lattice are regular hexagons, and each hexagon is adjacent to six other hexagons. The vertices of the lattice now have degree 3, because this lattice is the dual graph of the triangular lattice.

Hexagonal lattices have many interesting and useful properties, primarily because hexagons are "rounder" than squares.

To convert between triangular/hexagonal coordinates and geometrical coordinates, we assume that the origin of both systems is the center of a disk at $(0, 0)$.

The hexagonal coordinate `(xh,yh)` refers to the center of the disk on the horizontal row `xh` and diagonal column `yh`.

The geometric coordinate of such a point is a function of the radius of the disk r, half that of the diameter d described in the previous section:

```
hex_to_geo(int xh, int yh, double r, double *xg, double *yg)
{
        *yg = (2.0 * r) * xh * (sqrt(3)/2.0);
        *xg = (2.0 * r) * xh * (1.0/2.0) + (2.0 * r) * yh;
}

geo_to_hex(double xg, double yg, double r, double *xh, double *yh)
{
        *xh = (2.0/sqrt(3)) * yg / (2.0 * r);
        *yh = (xg - (2.0 * r) * (*xh) * (1.0/2.0) ) / (2.0 * r);
}
```

The row-column nature of the hexagonal coordinate system implies a very useful property, namely that we can efficiently store a patch of hexagons in a matrix `m[row][column]`. By using the index offsets described for triangular grids, we can easily find the six neighbors of each hexagon.

There is a problem, however. Under the hexagonal coordinate system, the set of hexagons defined by coordinates (hx, hy), where $0 \leq hx \leq x_{\max}$ and $0 \leq hy \leq y_{\max}$, forms a diamond-shaped patch, not a conventional axis-oriented rectangle. However, for many applications we are interested in rectangles instead of diamonds.

To solve this problem, we define array coordinates so that `(ax,ay)` refers to the position in an axis-oriented rectangle with $(0, 0)$ as the lower-left-hand point in the matrix:

```
array_to_hex(int xa, int ya, int *xh, int *yh)
{
        *xh = xa;
        *yh = ya - xa + ceil(xa/2.0);
}

hex_to_array(int xh, int yh, int *xa, int *ya)
{
        *xa = xh;
        *ya = yh + xh - ceil(xh/2.0);
}
```

Longitude and Latitude

A particularly important coordinate grid is the system of longitude and latitude which uniquely positions every location on the surface of the Earth.

The lines that run east-west, parallel to the equator, are called lines of *latitude*. The equator has a latitude of 0^{o}, while the north and south poles have latitudes of 90^{o} North and 90^{o} South, respectively.

The lines that run north-south are called lines of *longitude* or *meridians*. The *prime meridian* passes through Greenwich, England, and has longitude 0^{o}, with the entire range of longitudes spanning from 180^{o} West to 180^{o} East.

Every location on the surface of the Earth is described by the intersection of a latitude line and a longitude line. For example, the center of the universe (Manhattan) lies at $40^{\text{o}}47'$ North and $73^{\text{o}}58'$ West.

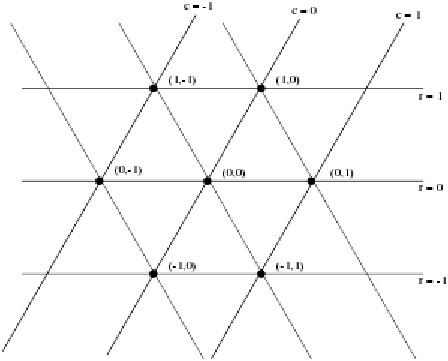
A *great circle* is a circular cross-section of a sphere which passes through the center of the sphere. The shortest distance between points x and y turns out to be the arc length between x and y on the unique great circle which passes through x and y.

Denote the position of point p by its longitude-latitude coordinates, $(p_{\text{lat}}, p_{\text{long}})$, where all angles are measured in radians. Then the great-circle distance between points p and q is

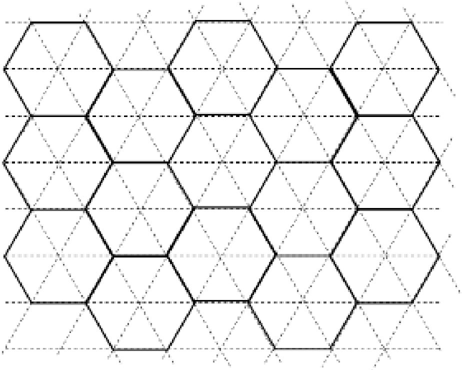
$$d(f,s) = R \times \arccos((\sin(l[f]) \times \sin(l[s])) + (\cos(l[f]) \times \cos(l[s]) \times \cos(dg)))$$

where R is the radius of the Great circle, dg = longitude(f)-longitude(s), and $l[f]$ = lattitude of f.

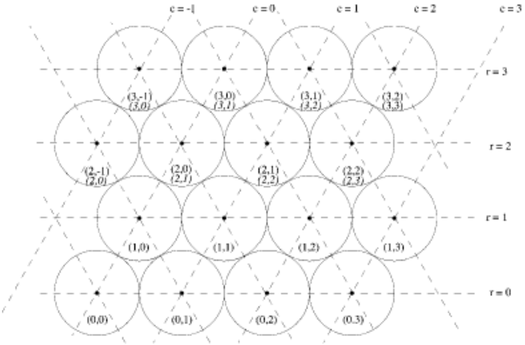
A row-column coordinate-system for **triangular grids** [1]



Deleting alternate vertices from a triangular lattice yields a **hexagonal lattice** [1]



A disk packing with **hexagonal coordinates**, as well as differing array coordinates (below in italics) [1]

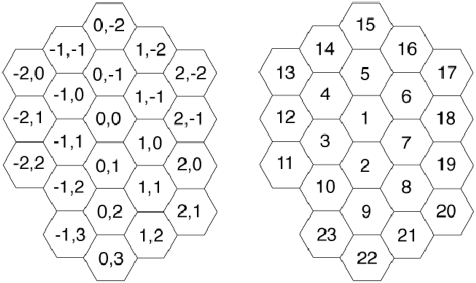


Exercício para Nota

- Bee Maja (10182)

Bee Maja (10182)

- Maja is a bee. She lives in a bee hive with thousands of other bees. This bee hive consists of many hexagonal honey combs where the honey is stored in.
- But bee Maja has a problem. Willi told her where she can meet him, but because Willi is a male drone and Maja is a female worker they have different coordinate systems.



Bee Maja (10182)

- Maja's Coordinate System (left):
  - Maja who often flies directly to a special honey comb has laid an advanced two dimensional grid over the whole hive.
- Willi's Coordinate System (right):
  - Willi who is more lazy and often walks around just numbered the cells clockwise starting from 1 in the middle of the hive.
- Help Maja to convert Willi's system to hers. Write a program which for a given honey comb number gives the coordinates in Maja's system.

Bee Maja (10182)

- Input Specification:
  - The input file contains one or more integers which represent Willi's numbers. Each number stands on its own in a separate line, directly followed by a newline. The honey comb numbers are all less than 100 000.
- Output Specification:
  - You should output the corresponding Maja coordinates to Willi's numbers, each coordinate pair on a separate line.

Bee Maja (10182)

- Sample Input
```
1
2
3
4
5
```
- Sample Output
```
0 0
0 1
-1 1
-1 0
0 -1
```

Referências I

[1] Skiena, Steven S.; Revilla, Miguel A.
Programming Challenges - The Programming Contest Training Manual - chapter 11.
Springer, 2003.

[2] Skiena, Steven
CSE 392 - Programming Challenges. Spring 2010 - Lecture 12 (week12.pdf).
http://www.cs.sunysb.edu/~skiena/392/newlectures/