Walktimefunc (Written in GNU Octave, should work in MATLAB)

Author: Gareth Rees, University of Cambridge, July 2022
wgr2@cam.ac.uk

What it does:

Calculates the minimum time taken to walk from a given location within a Digital Elevation Model (DEM) to all other locations within the DEM. It assumes that walking speed depends only on the gradient of the terrain, using a simple model (Rees 2004a), and implements Dijkstra's least-cost algorithm (Rees 2004b). It assumes that all areas within the DEM are equally navigable, so it will happily cross bodies of water, fences etc.

Calling the function:

[K,R]=walktimefunc(D, x0, y0, c, x1, y1, s)

Arguments

D: 2D array holding the DEM. Heights are in metres
x0, y0: Grid coordinates of the centre of the top-left (north west) cell in the DEM
c: DEM cell size in metres
x1, y1: Grid coordinates of starting point (must be within the DEM – no error checking performed)
s: Subsampling factor to generate the route maps, explained below.

Outputs

K: 2D array containing the minimum walking times, in seconds. This has the same dimensions as the DEM.
R: (n × 2) vector representing the route maps, explained below

Route map

The least-time routes are calculated from the starting point to a subset of the cells within the DEM. They are calculated as an ordered set of vertices, representing the route from the start to the first end point cell, then back to the start, then to the second end point cell, back to the start, and so on. They are stored as single vector, so if this is simply plotted from beginning to end it will trace the entire set of routes from the starting point. The subsampling factor s is used to control the number of cells in the subset. If s=1, routes to all the cells are calculated. This would be useless for plotting, since the lines representing the different routes would touch one another. The greater the value of s, the smaller the number of end-point cells used in the route calculation. Setting s=0 switches off the route map calculation altogether and returns an empty vector for R.

How it works

1. Setup

The DEM contains *rr* rows × *cc* columns = *nn* cells. These can be indexed by row and column number, or serially. Lookup table **a1**(*i, j*) (*rr*×*cc)* stores the serial number of the cell at row *i*, column *j*. For example, for a 6×4 array **a1** would be

1	7	13	19

2	8	14	20
3	9	15	21
4	10	16	22
5	11	17	23
6	12	18	24

Lookup table **a2**(*i*, *j*) (*nn*×2) holds the row (*j*=1) and column (*j*=2) indices for the cell with serial number *i*:

1	1
2	1
3	1
4	1
5	1
6	1
1	2
2	2
…	…

The vector **neightot**(*i*) (*nn*×1) contains the number of neighbouring gridcells (minimum possible=3, maximum=8) for the cell with serial number *i*. The table **neighlist**(*i*, *j*) (*nn*×8) is the serial number of the *j*th neighbour of cell *i*.

The table **cost**(*i*, *j*) (*nn*×8) contains the walking time (in seconds) from cell with serial *i* to the cell with serial number neighlist(*i*, *j*).

Walking times are calculated using the formula

$$t = a\,x + b\,\frac{h^2}{x}$$

where *x* is the distance between cells and *h* is the height difference between them. The coefficients are taken as $a = 0.75$ s m^{-1} and $b = 14.6$ s m^{-1}.

2. Initialisation

Vector **costab**(*i*) (*nn*) holds the minimum cost (=time) to reach the target cell from cell *i*. It is initialised to infinity for all cells except the target, which is initialised to zero. Vector **prev**(*i*) (*nn*) is the serial number of the cell from which cell *i* is reached. It is initialised to zero for all cells. Vector **Q** (*nn*) is the queue of cells still needing to be assigned. It is initialised to the vector 1:*nn*.

3. Iteration

Simple implementation of Dijkstra algorithm:

```
Iterate until Q is empty
    find nu = the node in Q with the minimum cost
    remove node nu from Q
    for every neighbour n2 of node nu
        calculate alternative cost to reach target from it via nu
        if this cost is less than previously assigned cost
            set its cost to the alternative
```

set the prev pointer for this cell to point to nu

When complete, reformat the cost table into **costarr** with the same dimensions as the DEM.

Execution time

Using GNU Octave on MacBook Pro 2.2 GHz Quad-Core Intel Core i7 with 16 GB 1600 MHz DDR3 (approximate calculations for 40000 < *nn* < 100000):

Calculating **neighlist**: 7.5 nn^2 ns
Calculating **cost**: 0.4 $nn^{5/3}$ µs
Dijkstra iteration: 0.4 $nn^{5/3}$ µs

I do not currently understand the 5/3 exponents. But extrapolating, this suggests that all steps in the process could be completed in

1 minute	for	45000 cells
5 minutes		112000
10 minutes		165000
30 minutes		305000
1 hour		448000

And just the Dijkstra iteration itself could be completed in

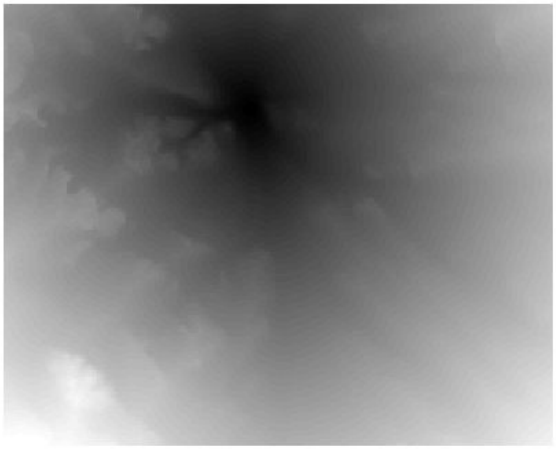
1 minute	for	80000 cells
5 minutes		211000
10 minutes		320000
30 minutes		620000
60 minutes		940000

Example

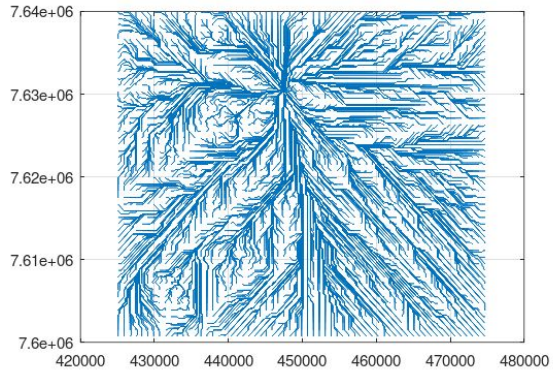
GMTED.txt is an extract of the GMTED for North Norway.

```
a=load('GMTED.txt');
[K,R]=walktimefunc(a,425100,7639900,200,447680,7630580,4);
```

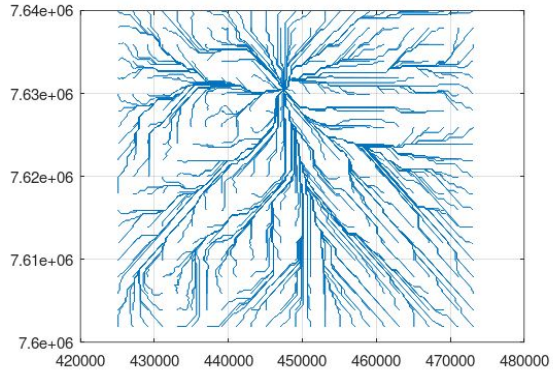
Walktime array K:



plot(R(:,1),R(:,2))



Recalculated with s=10:



Planned enhancements

• Very simple to implement: Save the results of the setup stage so that the Dijkstra iteration can be re-run for different target locations without having to recalculate the neighbour or cost tables.
• Implement barriers (e.g. because of water bodies, political boundaries) as infinite costs.
• Implement Floyd-Warshall algorithm instead. This is unlikely to be an improvement, however, since the graph is very sparse (density ~ 8/*nn*), unless it is desired to calculate a very large number of routes within the DEM.

References

Rees, W.G. 2004a. ‘Naismith’s Rule Overhauled’. *Scottish Mountaineering Club Journal* 39: 666–69.

———. 2004b. ‘Least-Cost Paths in Mountainous Terrain’. *Computers & Geosciences* 30 (3): 203–9. https://doi.org/10.1016/j.cageo.2003.11.001.