Dijkstra Maps in Roguelikes

Jeremy Mates

September 20, 2018

Simple Pathfinding

Consider the following map, wherein there are three velociraptors `R` surrounding the player `@` on an open field[Munroe(2006)]. How can the raptors reach the player?

```
......R
.R.....
.......
...@...
.......
.......
..R....
```

One solution to this problem is the so-called Dijkstra Map[Pender(2010)]. A map this small can easily be worked out on graph paper; set the location of the player to the value `0`. Then in the squares adjacent to that point write the next highest number, `1`.

```
.......
.......
...1...
..101..
...1...
.......
.......
```

And so forth until the map is filled. This is a Dijkstra Map.

```
6543456
5432345
4321234
3210123
4321234
5432345
6543456
```

Note that diagonal moves were not considered–only those points North South East or West of the given square. This has ramifications on more complicated maps that contain obstacles. Meanwhile, each velociraptor can use this map to find the player by walking "downhill" to `0`, or can flee by picking a cell with a higher value.

Obstacles

Many maps will be more complicated than the above and will contain obstacles, typically walls, or lava, or there may be monsters that can pass through walls but not across holy water–spectres perhaps–or anything else you can imagine. Let us for now only consider physical walls, represented using # as is typical for roguelikes.

```
R....
.###.
.#@..
.#.##
.#.#.
...#.
```

A Dijkstra Map for this might look like

8	7	6	5	4
9	-1	-1	-1	3
8	-1	0	1	2
7	-1	1	-1	-1
6	-1	2	-1	?
5	4	3	-1	?

where the impassable walls are represented by `-1`. One may use `Inf` or an object for such cells, but I assume here an integer or fixnum array. Another option would be to leave walls at the maximum integer or `MOST-POSITIVE-FIXNUM` value and to ignore them while calculating the costs. Note the two squares lower right that have no path to the player; these after calculation will typically remain at the maximum integer value the field was filled with before the pathfinding pass. This map by the way was made with my `Game::DijkstraMap` module[Mates(2018b)].

```
#!/usr/bin/env perl
use strict;
use warnings;
use Game::DijkstraMap;
my $dm = Game::DijkstraMap->new( str2map => <<'EOM' );
.....
.###.
.#x..
.#.##
.#.#.
...#.
EOM
print $dm->to_tsv;
```

Pathfinding on this map must ignore `-1` and look for values equal to or greater than `0` when routing to a goal. Here, the raptor should move horizontally to reach the player in as few moves as possible.

Multiple Goals

A Dijkstra Map may contain multiple goals; pathfinding will find the (or a) nearest goal. Crabs `c` for example may wish to retreat to water ~ when threatened by the player.

```
..~~......~.....
.~~~~......~~...
...~~..c..~~....
..........c.....
...~~...........
..~~...c........
...~~...........
.....~.........@
```

2	1	0	0	1	2	3	4	2	1	0	1	1	2	3	4
1	0	0	0	0	1	2	3	4	2	1	0	0	1	2	3
2	1	1	0	0	1	2	3	2	1	0	0	1	2	3	4
3	2	2	1	1	2	3	4	3	2	1	1	2	3	4	5
4	3	1	0	0	1	2	3	4	3	2	2	3	4	5	6
2	1	0	0	1	2	3	4	5	4	3	3	4	5	6	7
3	2	1	0	0	1	2	3	4	5	4	4	5	6	7	8
4	3	2	1	1	0	1	2	3	4	5	5	6	7	8	9

This map however does not consider other crabs–can two occupy the same square, or will one need to find a longer route to a free water cell?–nor the influence of the player; a monster may realistically have multiple desires: get to water while also avoiding the player. This requires a combination of Dijkstra Maps, the inverse distance to the player combined with the above water map. (This is an area I and my code need to level up a bit more in, so I will say no more about it.)

The Diagonal

Costs in the above maps have been done without consideration for diagonal moves. Consider the following map, where `x` is the goal.

```
@#..
#...
...x
```

?	-1	3	2
-1	3	2	1
3	2	1	0

The player here cannot pathfind to the goal as the diagonal move was not considered by the 4-way algorithm that only consults cells North South East and West. Various roguelikes (Angband, Dungeon Crawl Stone Soup) permit such diagonal moves, so will need to use an 8-way algorithm when constructing a Dijkstra Map. Other roguelikes

(Brogue, POWDER) may deny such diagonal moves so can use the 4-way algorithm. This choice also influences level design. 4-way and 8-way roguelikes require rather different diagonal corridors:

```
8-way    @####   4-way    @.###
corridor #.###   corridor #..##
         ##.##            ##.##
         ###x#            ##.x#
         #####            #####
```

Brogue is complicated in that it allows some diagonal moves. The choices for a two dimensional roguelike are pure 8-way, pure 4-way, or hybrid models somewhere between those two extremes. A 4-way Dijkstra Map algorithm can be used with 8-way motion provided 4-way moves are possible to everywhere that must be reached. Diagonal moves in such a case exist as shortcuts–moving diagonally along the above 4-way corridor (which Brogue would not permit, nor POWDER unless one is ploymorphed into a grid bug).

Diagonal Maps

8-way maps typically assign equal costs to all adjacent squares. The original raptor map thus looks like:

```
......R  3333333
.R.....  3222223
.......  3211123
...@...  3210123
.......  3211123
.......  3222223
..R....  3333333
```

This while traditional for roguelikes is not actually correct; diagonals under euclidean geometry should instead use $\sqrt{x^2 + y^2}$ or $\sqrt{2}$ instead of 1 for the closest diagonal. Various roguelikes are actually non-euclidean: Brogue and Dungeon Crawl Stone Soup apply the same cost to a move in any direction, diagonal or otherwise. Anyways! Our original diagonal map that stumped the 4-way map under (non-euclidean) 8-way is:

```
@#..
#...
...x
```

3	-1	2	2
-1	2	1	1
3	2	1	0

And the player can path to the goal.

Super Dimensional Dijkstra Maps

We need not confine ourselves to two, or even three dimensions; Dijkstra Maps can be built in arbitrary numbers of dimensions (memory requirements, implementation demands, and sanity permitting). The following is a four-dimensional map with an implementation[Mates(2018a)] that does not consider diagonal moves legal.

```
% sbcl --noinform --load dijkstramap.lisp
* (setf *dimap-cost-max* 99)

99
* (defparameter level
    (make-array '(3 3 3 3)
                :initial-contents
                '((((99 -1 99) (-1 99 -1) (99 -1 99))
                   ((-1 99 99) (99 99 99) (99 99 -1))
                   ((99 99 99) (99 99 99) (99 99 99)))
                  (((-1 99 99) (99 99 99) (99 99 -1))
                   ((99 99 99) (99  0 99) (99 99 99))
                   ((99 99 99) (99 99 99) (99 99 99)))
                  (((99 99 99) (99 99 99) (99 99 99))
                   ((99 99 99) (99 99 99) (99 99 99))
                   ((99 99 99) (99 99 99) (99 99 99))))))

LEVEL
* (dimap-calc level)

4
* level

#4A((((99 -1 4) (-1 2 -1) (4 -1 99))
     ((-1 2 3) (2 1 2) (3 2 -1))
     ((4 3 4) (3 2 3) (4 3 4)))
    (((-1 2 3) (2 1 2) (3 2 -1))
     ((2 1 2) (1 0 1) (2 1 2))
     ((3 2 3) (2 1 2) (3 2 3)))
    (((4 3 4) (3 2 3) (4 3 4))
     ((3 2 3) (2 1 2) (3 2 3))
     ((4 3 4) (3 2 3) (4 3 4))))
*
```

Productive uses for such maps are left as an exercise to the reader.

References

[Mates(2018a)] Jeremy Mates. dijkstramap.lisp. `https://github.com/thrig/ministry-of-silly-vaults/blob/master/dijkstramap.lisp`, 2018a. [Online; accessed 20-September-2018].

[Mates(2018b)] Jeremy Mates. Game::DijkstraMap. `https://metacpan.org/pod/Game::DijkstraMap`, 2018b. [Online; accessed 20-September-2018].

[Munroe(2006)] Randall Munroe. Substitute. `https://www.xkcd.com/135/`, 2006. [Online; accessed 20-September-2018].

[Pender(2010)] Pender. The Incredible Power of Dijkstra Maps. `http://www.roguebasin.com/index.php?title=The_Incredible_Power_of_Dijkstra_Maps`, 2010. [Online; accessed 20-September-2018].