DIRECTING-CROWD METHOD

Prashant Kumar
prashant.dwivedi@outlook.com

Directing-Crowd method after completion, creates a directed graph, which shows the path from any node to the exit point. Thus this method works for multi-source or single-source-unique-destination problems.

Its called the "Directing-Crowd" method as every node in the graph, after parsing, points to its next node which takes us one step closer to the exit point, no matter where we start from (here, we start from 'E' point).

This method is exactly like you are searching for a person in the crowd and every guy in the crowd points you towards that guy who is one step closer to the person you are looking for.

The method consists of 3 steps:

1. Reading the map and forming its "wire-frame" in the form of a graph.

2. Processing of the wire-frame to figure out the paths from every node to the exit point.

Intermediate step: changing the symbols of the nodes, forming the path, from '0' to 'P', to denote the path

3. Writing the wire-frame back to the file.

EXPLANATION:

DATA STRUCTURES:

point:
it denotes a coordinate on a graph with x and y values.

- exit point (xit: 'X')
- entry point (entry: 'E')
- boundary point (boundry): it represents the maximum x and y points in the map

node:
it denotes a node on a graph with properties:

- weight: lenght of the path this node to the exit point
- child: a point representing coordinates o its child. child is that next node whose weight is one 1 than this node
- self: coordinates of this node itself
- status: status representing whether the node has been visited or not (while parsing)
- symbol: the symbol (#, E, P, X, 0) held by point on the map having same coordinates as this node.

wire-frame (map[][]): an object of type node, it is the frame of the map inputted through the map.txt file. All the parsing is done on this wire-frame only. map[][] is a table and every point having coordinate (x, y) is stored in map[y][x].

PROGRAM LOGIC:

main() calls each of these functions one by one according to the steps below:

Step 1:
in function readMap()

a) open the file
b) read a character and push it to the wire-frame map[y][x]
   first read char is at coordinate (0, 0), next is at (0, 1) and first char in second line is at (1, 0) and so on.
c) every read char is assigned a weight (infinity). If the char is a newline character, y is incremented.
   i. x is incremented everytime.
   ii. If the character is a 'X', its coordinates are saved in 'xit'. Similarly 'E' is also saved in "entry".
   iii. If the boundary coordinates of the map (heightest x coordinate, heightst y coordinate) is less than the present x and y values

iv. of the map, they are updated.

d) As soon as the EOF is encountered, mapping is complete and files is closed.

Step 2:

parseMap()is called

a) exit node (xit) is marked as 0 weighted (since it is the exit point and is enqued in the parsingQ[] from rear (which is 0 yet)

b) Frontmost node from the parsingQ[] is extracted from front and stored in 'dummy'.
if dummy is the entry point, goto step 3.

c) dummy's 'unvisited' marked neighbours are discovered in the order:

```
(*) *  *              * (*) *              * *  (*)
 *  d  *     then     *  d  *     then     *  d  *
 *  *  *              *  *  *              *  *  *

then

 *  *  *              *  *  *
(*) d  *     then     *  d (*)
 *  *  *              *  *  *

then

 *  *  *              *  *  *              *  *  *
 *  d  *     then     *  d  *      then    *  d  *
(*) *  *              * (*) *              *  * (*)
```

(*)--> this node is discovered and processed.

d) Each 'unvisited' marked neighbour is enqued in the parsingQ from rear and marked as 'visited'.

i. This node's child coordinate is set to dummy's coordinates.

ii. Its weight is set to dummy.weight+1. (i.e., 1 more than the weight of its child.

iii. So the node now points towards dummy, the node which is one step closer to the exit point.'

e) Goto step 2.b if there is a node left in the parsing queue.

note: its a kind of depth-first search.

Step 3:

function locatePath() is called

a) entry point is marked as 'current' node.

b) if current node's symbol is an 0 (i.e., symbol of a movable path), it is changed to 'P'.
variable 'steps' is incremented by 1.
end if

c) if current node's symbol is not a movable path, then
if currents node's symbol is also not exit point (X), then path is broken. steps = -1.
end if.

goto step 4.

end if
d) current's child is set as the current node.
e) If current node's symbol is a movable point (i.e., it is 0) goto step 3.b else goto step 4

Step 4:
in function writeMap()
write the symbols stored in map[][] in order in the file.

Step 5:
in function displayMap()
a) display the map[][] in order on the screen.

NOTES:

1. this program works even for non-rectangular arena. Just a connectivity of movable points (0) is required. like:

```
       E
#000000000
0#
#0######0000000
00000000000000000#
#000000000000000
#####00000000000#
           X
```

then output is:

```
       E
#PPPP00000
P#
#P######0000000
00PPPP0000000000#
#00000P000000000
#####00P00000000#
           X
```

---

2. in case the input map is like:

```
      00000000000000000000
      00000000000000000000
a)    00000000000000000000
      00000000000000000000
      00E000000000X00000000
```

then output will be

```
000000PPP000000000000
```

```
      00000P000P00000000000
b)    0000P00000P00000000000 => no of P=9
      000P0000000P000000000
      00E000000000X00000000
```

which is also the shortest path apart from another shortest path possible

```
      00000000000000000000
      00000000000000000000
c)    00000000000000000000 => no of P=9
      00000000000000000000
      00EPPPPPPPPPX00000000
```

- This is a valid path as every P is at diogonal from each other and number of P are ealso qual to 9 only, which is same as in the shortest possible path.
- This movement is due to the pattern in which nodes are discovered (see pattern in step 2.c of explaination).
- Although it can be easily fixed to display path as in c) but fixing is unnecessary unless we don’t have an option to move diagonally since the distance (and hence any incurring cost) remains same in both the cases.
- This method looks similar to Dijstra's method but it is a multi-source-single-destination method and Dijstra's is a single-source-multi-destination method.