Project 2:

Dynamic programming

Simon Balis

Gabriel Contreras

Ashton Honeggar

1. This problem can be solved by using iterative dynamic programing which works by starting at the opposite end of the goal and working its way back. For this problem, we start by treating the terrain as if it is only a single row. We are able to use a base case to solve this. From here, we add an additional row of terrain. With this new information and the shortest distance to the northern border for every other location, we are able to calculate this new row’s shortest distance to the northern border. Rows are able to be added and shortest distance solved until the entire terrain is present. Once the terrain is finished, the path with the shortest distance of the specified location can be backtracked.
2. The recurrence used finds the minimum distance of the following for each cell to determine the next destination of the shortest path:
   1. The shortest distance to the northern border of the cell to the northwest plus the distance to get there including the time penalty.
   2. The shortest distance to the northern border of the cell to the north plus the distance to get there.
   3. The shortest distance to the northern border of the cell to the northeast plus the distance to get there including the time penalty.
3. Similar to the recurrence, the base case finds the minimum of the following for each of the cells along the norther border:
   1. The left cells distance to the northern border plus the time penalty.
   2. The current cells distance to the northern border.
   3. The right cells distance to the northern border plus the time penalty.
4. Below is the pseudo code for the memorization algorithm

Input: 2d array of r x c containing the cost associated with traversing each cell

Input: row and column location in the array

Output: 2d array of r x c containing the smallest cost it would take to reach the northern border from the given point in the array

memoDPMap(in,i,j)

if ans[r][c] != NIL

return ans[r][c]

else if i == 0

ans[0,j] = in[0,j]

else

ans[i,j] = min{ if J >0 then memoDPMap(in,i-1,j-1) + 1.4\*in[i,j-1]

memoDPMap(in,i-1,j) + in[i,j]

if j < c-1 then memoDPMap(in,i-1,j+1) + 1.4\*in[i,j+1] }

end

return ans[][]

Input: a text file called “input.txt”

Output: The most effective paths to reach all of the southern border locations saved to “output.txt”

roughTerrainMemo()

in[r][c] = array input

ans[r][c] = NIL

for i to c-1

memoDPMap(in,r-1,i)

end

return ans[][]

1. Time complexity = number of cells \* cost per cell

O(n\*m)\*(1) = O(nm)

1. Below is the pseudo code for the iterative algorithm

Input: 2d array of r x c containing the cost associated with traversing each cell

Output: 2d array of r x c containing the smallest cost it would take to reach the northern border from the given point in the array

iterDPMap()

in[r][c] = input

ans[r][c]

for i = 0 to r-1

for j = 0 to c-1

if i == 0

ans[0,j] = in[0,j]

else

ans[i,j] = min{ if J >0 then ans[i-1,j-1] + 1.4\*in[i,j-1]

ans[i-1,j] + in[i,j]

if j < c-1 then ans[i-1,j+1] + 1.4\*in[i,j+1] }

end

end

end

return ans[][]

1. Below is the pseudo code for the optimal route algorithm

Input: 2d array returned from iterDPMap()

Input: 2d array of r x c containing the cost associated with traversing each cell

Input: row and column location in the array

Output: A stack with the locations that need to be traversed to reach the location provided with the least cost

fastestPaths(ans, in, r, c)

path = Stack()

ans = Queue()

for j to c-1

loc = j

for i to 1 starting at i = r-1

if loc > 0 && ans[i,loc] == ans[i-1,loc-1] + 1.4\*in[i,loc]

path.Push(SE)

loc--

else if loc < c-1 && ans[i,loc] == ans[i-1,loc+1] + 1.4\*in[i,loc]

path.Push(SW)

loc++

else

path.Push(S)

end

end

path.Push(loc)

while path !empty

ans.Enqueue(path.Pop())

end

end

return ans