CMPUT 396 – Maze Traversal

Thesus and Minotaur: Thesus entered the labyrinth where the Minotaur resided to kill it. He tied a string to the door to help him keep track of where he was and used it to get out of the Labyrinth once he had slayed the beast.

Hansel and Gretel: Used breadcrumbs in the forest to keep track of where they had come from.

Moral of these stories: use breadcrumbs to help keep track of where you've been. This is the key in mazes.

Martin Gardner:

- His first degree was in philosophy, then he went into the military. He was interested in magic and later became a writer.
- He was a mathematical games columnist in Scientific American (writing for the general public).
- His first column on hexaflexagons was in 1956 and he started writing it regularly in 1957.
- He wrote a column about Hex.
- The January 1959 column was about mazes and how they can be traversed.
- He was a skeptic: someone who doubts, asks questions.

Maze Traversal Algorithms

One algorithm if you are going through a maze is to always keep your hand on the left (or right) wall. This doesn't work if there are cycles in the maze.

Algorithm 0: Random Walk

We make the assumption that there is some path through the maze and that it is a finite maze (each cell has a finite number of neighbours).

The algorithm is basically to proceed following the current passage until a junction is reached, then make a random decision about the next direction to follow.

```
current cell <-- origin
while current cell != destination
   current cell <-- random neighbour of current cell
print current cell
```

Pros: Easy to describe, leave some implementation details of language implementers

Cons: Harder to learn, lose control to language implementers, can be extremely inefficient (you can't put an upper bound on the number of steps)

Analysis of algorithm:

Correct? Yes.

- With positive probability, every reachable location will be reached
- *Proof:* consider a path from the start to the goal with distance *t* and assume that each cell has at most *k* neighbours. After exactly *t* iterations, we have $\frac{1}{k^t}$ probability that we will reach the goal. This comes from the fact that we have a probability of $\frac{1}{k}$ that a neighbour of the start is on the path.

Efficient (with respect to time and/or space?): No.

- If at least one neighbour of the start is not the goal, then for any positive integer n, the probability that there are at least n iterations is positive.
- *Proof:* if I know I have a start, and end, and at least one neighbour that is not the goal, there is a positive probability that it could go back and forth indefinitely (start, not-goal, start, not-goal, start, …). The probability that we are on this path after n iterations is at least $\frac{1}{k^n}$.

Can this algorithm be improved? Yes. The answer comes from Thesus and Hansel & Gretel: breadcrumbs. We need to keep track of visited cells.

Algorithm 1: Random walk with remembering

This is basically the same as the random walk algorithm, but we mark cells that we visit and visit our unmarked neighbours.

```
current cell <-- origin
mark current cell
while current cell != destination
   current cell <-- unmarked random neighbour of current cell
   mark current cell
print current cell
```

Analysis of algorithm:

Correct? No.

- If you get to a deadend, you will be stuck because you need to go back to a node with a neighbour you haven't seen before.
- It's basically a useless algorithm because it doesn't do what it's supposed to do.

Efficient? Yes.

- For a maze with n cells, there will be at most n-1 iterations (because it will visit every node after the start node in the maze).

A maze and its adjacency graph

```
X X X X X X    a - b - c - d
X . . . . X    |       |
X . X . X X    e       f
X . . X . X    |
X . . . . X    g - h       i
X X X X X X    |   |       |
               j - k - l - m

nodes or points:  a b c ... m
edges or lines:  ab ae bc cd cf eg gh gj hk im jk kl lm
```

Graph traversal:

From a given start node, see what nodes we can reach by following edges. Maintain a list L of nodes seen but not yet explored. Do not add a node to a list if we have already seen it.

Breadth-first Search:

- L is a queue, FIFO list
- Remove the node appended earliest.

Iterative Depth-first Search:

- L is a stack, LIFO list
- Remove the node appended latest.

Algorithm for iterative BFS/DFS:

```
for all nodes v: v.seen <- False

L = [start],  start.seen = True
while not empty(L):
  v <- L.remove()
  for each neighbour w of v:
    if w.seen==False:
      L.append(w), w.seen = True
```

Example trace of BFS:

```
use the above adjacency graph, start from node j

initialize L = [ j ]

while loop execution begins
  remove j from L                L now [ ]
  nbrs of j are g,k
  append g to L, g.seen <- T     L now [ g ]
  append k to L, k.seen <- T     L now [ g, k ]

while loop continues (bfs, so L is queue, so remove g)
  remove g from L                L now [ k ]
  nbrs of g are e,h,j
  append e to L, e.seen <- T     L now [ k, e ]
  append h to L, h.seen <- T     L now [ k, e, h ]
  j already seen

while loop continues: remove k from L
  v <- k                         L now [ e, h ]
  nbrs of k are j,h,l
  j already seen
  h already seen
  append l to L, l.seen <- T     L now [ e, h, l ]

while loop continues: ...
```

Depth-first Search (recursive):

- Recursive (recurse as soon as unseen neighbour found)
- Differs slightly from iterative-dfs

Recursive DFS algorithm:

```
for all nodes v: v.seen <- False

def dfs(v):
  v.seen <- True
  for each neighbour w of v:
    if w.seen==False:
      dfs(w)

dfs(start)
```

Example DFS trace:

```
use above graph, start from j

dfs(j) nbrs g,k
. dfs(g) nbrs e,h,j
. . dfs(e) nbrs a,g
. . . dfs(a) nbrs b,e
. . . . dfs(b) nbrs a,c
. . . . . a already seen
. . . . . dfs(c) nbrs b,d,f
. . . . . . b already seen
. . . . . . dfs(d) nbrs c
. . . . . . . c already seen
. . . . . . dfs(f) nbrs c
. . . . . . . c already seen
. . . . e already seen
. . . g already seen
. . dfs(h) nbrs g,k
. . . g already seen
. . . dfs(k) nbrs g,h,l
. . . . g already seen
. . . . h already seen
. . . . dfs(l) nbrs k,m
. . . . . k already seen
. . . . . dfs(m) nbrs l,i
. . . . . . l already seen
. . . . . . dfs(i) nbrs m
. . . . . . . m already seen
. . j already seen
. k already seen
```

Example, from j, with the above maze:

```
- assume nbrs are stored in alphabetic order
- iterative bfs: order of removal from list?
    j g k e h l a m b i c d f
- iterative dfs: order of removal from list?
    j k l m i h g e a b c f d
- recursive dfs: order of dfs() calls?
    j g e a b c d f h k l m i

- iterative dfs with nbrs added to list in reverse order:
    j g e a b c d f k h l m i
notice that this differs from recursive dfs
```

Algorithm 2: use list to keep track of cells we have seen

BFS: list cells examined in FIFO manner (queue).

- Pros: finds the shortest path, will never get trapped in a blind alley
- Cons: if solution is far away it consumes time, memory constraints

DFS: list cells examined in FILO manner (stack). Use this algorithm if you have evidence that one neighbour is more likely than the others.

- Pros: memory requirement is linear wrt nodes, less time and space complexity than BFS
- Cons: Possible for it to get stuck following a deep path