Maze PA Explanation

Mark Redekopp
Maze Solver

- Consider this maze
  - S = Start
  - F = Finish
  - . = Free
  - # = Wall

- Find the shortest path

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>S</td>
<td>#</td>
<td>F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#</td>
</tr>
</tbody>
</table>
Maze Solver

• To find a (there might be many) shortest path we use a breadth-first search (BFS)

• BFS requires we visit all nearer squares before further squares
  – A simple way to meet this requirement is to make a square "get in line" (i.e. a queue) when we encounter it
  – We will pull squares from the front to explore and add new squares to the back of the queue
Maze Solver

• We start by putting the starting location into the queue

Maze array:

Queue
Maze Solver

• We start by putting the starting location into the queue

• Then we enter a loop...while the queue is not empty
  – Extract the front location, call it "curr"
  – Visit each neighbor (N,W,S,E) one at a time
  – If the neighbor is the finish
    • Stop and trace backwards
  – Else if the neighbor is a valid location and not visited before
    • Then add it to the back of the queue
    • Mark it as visited so we don't add it to the queue again
    • Record its predecessor (the location [i.e. curr] that found this neighbor)
Maze Solver

- We start by putting the starting location into the queue
- Then we enter a loop...while the queue is not empty
  - Extract the front location, call it "curr"
  - Visit each neighbor (N,W,S,E) one at a time
  - If the neighbor is the finish
    - Stop and trace backwards
  - Else if the neighbor is a valid location and not visited before
    - Then add it to the back of the queue
    - Mark it as visited so we don't add it to the queue again
    - Record its predecessor (the location [i.e. curr] that found this neighbor)
Maze Solver

- We start by putting the starting location into the queue
- Then we enter a loop...while the queue is not empty
  - Extract the front location, call it "curr"
  - Visit each neighbor (N,W,S,E) one at a time
  - If the neighbor is the finish
    - Stop and trace backwards
  - Else if the neighbor is a valid location and not visited before
    - Then add it to the back of the queue
    - Mark it as visited so we don't add it to the queue again
    - Record its predecessor (the location [i.e. curr] that found this neighbor)
Maze Solver

• We start by putting the starting location into the queue
• Then we enter a loop...while the queue is not empty
  – Extract the front location, call it "curr"
  – Visit each neighbor (N,W,S,E) one at a time
  – If the neighbor is the finish
    • Stop and trace backwards
  – Else if the neighbor is a valid location and not visited before
    • Then add it to the back of the queue
    • Mark it as visited so we don't add it to the queue again
    • Record its predecessor (the location [i.e. curr] that found this neighbor)
Maze Solver

• We start by putting the starting location into the queue
• Then we enter a loop...while the queue is not empty
  – Extract the front location, call it "curr"
  – Visit each neighbor (N,W,S,E) one at a time
  – If the neighbor is the finish
    • Stop and trace backwards
  – Else if the neighbor is a valid location and not visited before
    • Then add it to the back of the queue
    • Mark it as visited so we don't add it to the queue again
    • Record its predecessor (the location [i.e. curr] that found this neighbor)
Maze Solver

• We start by putting the starting location into the queue
• Then we enter a loop...while the queue is not empty
  – Extract the front location, call it "curr"
  – Visit each neighbor (N,W,S,E) one at a time
  – If the neighbor is the finish
    • Stop and trace backwards
  – Else if the neighbor is a valid location and not visited before
    • Then add it to the back of the queue
    • Mark it as visited so we don't add it to the queue again
    • Record its predecessor (the location [i.e. curr] that found this neighbor)
Maze Solver

• We start by putting the starting location into the queue

• Then we enter a loop...while the queue is not empty
  – Extract the front location, call it "curr"
  – Visit each neighbor (N,W,S,E) one at a time
  – If the neighbor is the finish
      • Stop and trace backwards
  – Else if the neighbor is a valid location and not visited before
      • Then add it to the back of the queue
      • Mark it as visited so we don't add it to the queue again
      • Record its predecessor (the location [i.e. curr] that found this neighbor)

Visitors

Queue

Predecessor

Maze array:

Visited

Found the Finish at (1,2)
Maze Solver

• Now we need to backtrack and add *'s to our shortest path
• We use the predecessor array to walk backwards from curr to the start
  – Set maze[curr] = ‘*’
    • Not real syntax (as ‘curr’ is a Location struct)
  – Change curr = pred[curr]
Maze Solver

• Now we need to backtrack and add *'s to our shortest path
  – Set maze[curr] = ‘*’
    • Not real syntax (as ‘curr’ is a Location struct)
  – Change curr = pred[curr]

• We use the predecessor array to walk backwards from curr to the start
  – curr = pred[curr]
Maze Solver

• Now we need to backtrack and add *'s to our shortest path

• We use the predecessor array to walk backwards from curr to the start
  – Set maze[curr] = ‘*’
    • Not real syntax (as ‘curr’ is a Location struct)
  – Change curr = pred[curr]
Need to Do

- Queue class
  - Make internal array to be of size = max number of squares
  - Should it be dynamic?
  - We need to keep track of the “front” and “back” since only a portion of the array is used
  - Just use integer indexes to record where the front and back are

Maze array:

<table>
<thead>
<tr>
<th></th>
<th>(0,0)</th>
<th>(0,1)</th>
<th>(0,2)</th>
<th>(0,3)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(1,0)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(1,1)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(2,0)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(2,1)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(2,2)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(2,3)</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(3,0)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(3,1)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(3,2)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>(3,3)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>
Need to Do

• Allocate 2D arrays for maze, visited, and predecessors
  – Note: in C/C++ you cannot allocate a 2D array with variable size dimensions
    • BAD: new int[numrows][numcols];
  – Solution:
    • Allocate 1 array of NUMROW pointers
    • Then loop through that array and allocate an array of NUMCOL items and put its start address into the i-th array you allocated above

Maze array:

```
(0,0) * (0,1) * (0,2) * (0,3) 
(1,0) S # F #
(2,0) . # . #
(3,0) . . . #
```

Each entry is int *

```
0 1 2 3
```

```
0 0 0 0
1a0 2c0 1b4 3e0
0 0 0 0
```

Thus t is int **
BACKUP
Maze Solver

• To find a (there might be many) shortest path we use a breadth-first search (BFS)

• BFS requires we visit all nearer squares before further squares
  – A simple way to meet this requirement is to make a square "get in line" (i.e. a queue) when we encounter it
  – We will pull squares from the front to explore and add new squares to the back of the queue
Maze Solver

• To find a (there might be many) shortest path we use a breadth-first search (BFS)
• BFS requires we visit all nearer squares before further squares
  – A simple way to meet this requirement is to make a square "get in line" (i.e. a queue) when we encounter it
  – We will pull squares from the front to explore and add new squares to the back of the queue
Maze Solver

• To find a (there might be many) shortest path we use a breadth-first search (BFS)

• BFS requires we visit all nearer squares before further squares
  – A simple way to meet this requirement is to make a square "get in line" (i.e. a queue) when we encounter it
  – We will pull squares from the front to explore and add new squares to the back of the queue

<table>
<thead>
<tr>
<th>Maze array:</th>
<th>Visited</th>
<th>Predecessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,0) . . . .</td>
<td>(0.0) 0 0 0 0</td>
<td>(0.0) -1 -1 -1 -1</td>
</tr>
<tr>
<td>(1,0) S . . .</td>
<td>(1.0) 1 0 0 0</td>
<td>(1.0) -1 -1 -1 -1</td>
</tr>
<tr>
<td>(2,0) . . . #</td>
<td>(2.0) 0 0 0 0</td>
<td>(2.0) -1 -1 -1 -1</td>
</tr>
<tr>
<td>(3,0) . . .</td>
<td>(3.0) 0 0 0 0</td>
<td>(3.0) -1 -1 -1 -1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Queue</th>
<th>1,0</th>
</tr>
</thead>
</table>