UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG
School of Computer Science

COMS1000: Data and Data Structures
Project 2: Maze Search

Due Date: 17:00 on Sep 12, 2013
Revision 0.1

1 Description

The problem of finding a shortest path in a maze with obstacles occurs frequently in applications such as computer games, satellite navigation and VLSI routing.

Consider an `m`×`n` grid which contains obstacles. A mouse is placed in a location `(x, y)`. The task is for the mouse to find a shortest path from its initial location to a piece of cheese if such a path exists. The mouse can only move North, South, East and West. We assume that the mouse has a map of the maze and, being a smart mouse, can therefore compute the optimal, shortest path.

2 Algorithm

Because each step in the path towards the cheese costs the same amount, we can use a standard breadth first search to find the shortest path. The first path starting at the mouse and ending at the cheese will be the shortest path. Starting at the mouse, the algorithm proceeds as follows:

1. Starting at the source, find all new cells that reachable at distance 1, i.e. all paths that are just 1 unit in length, and mark them with that length.
2. Using the distance 1 cells, find all new cells which are reachable at distance 2.
3. Using all cells at distance 2 from the source, find all cells with distance 3.
4. Repeat until the target is found. This expansion creates a wave front of paths that search broadly from the source cell until target cell is hit.
5. From the target cell, select a shortest path (any shortest path) back to the source and mark the cells along the path.

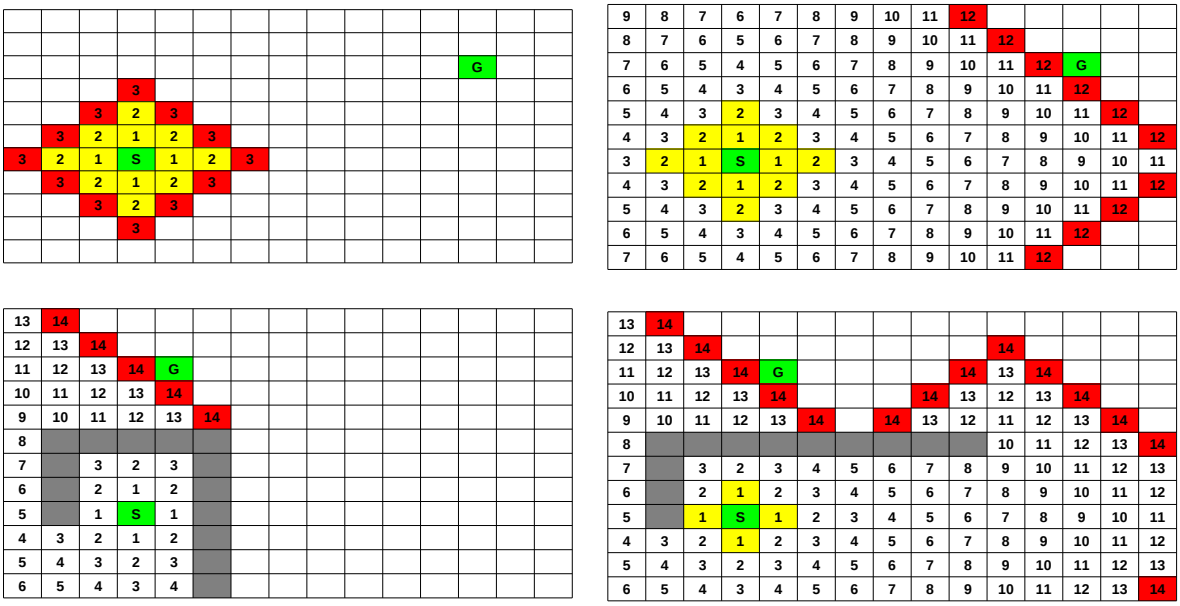
This 'wavefront' is called the fringe – the edge of what we've seen so far. At each iteration, we take a cell from the fringe and look at its undiscovered neighbours. Note that if it takes n steps to get to an item in the fringe, it then takes $n + 1$ steps to get to any of its undiscovered neighbours. By checking all paths of length n first, we can be sure that there is no quicker way to get to an undiscovered neighbour.[1] The fringe can be represented using a queue, this means that in iteration i, dequeue a cell from the fringe and enqueue all of its neighbours with a path length of $i + 1$. Because the fringe is FIFO, we will dequeue all cells at level i before we dequeue any cells at level $i + 1$.

It turns out that this is a simplified, special case of Dijkstra's algorithm for finding the shortest path in a graph. You'll learn more about this next year.

[1] See if you can prove this, try proofs by induction and/or contradiction.

3 Examples

The figures below show how the wavefront propagates through the maze and around obstacles. The red blocks indicate the cells in the fringe (*wavefront*). The green blocks indicate the start and end points. The numbers shows how many steps the mouse must make to get from the start point to the relevant block.



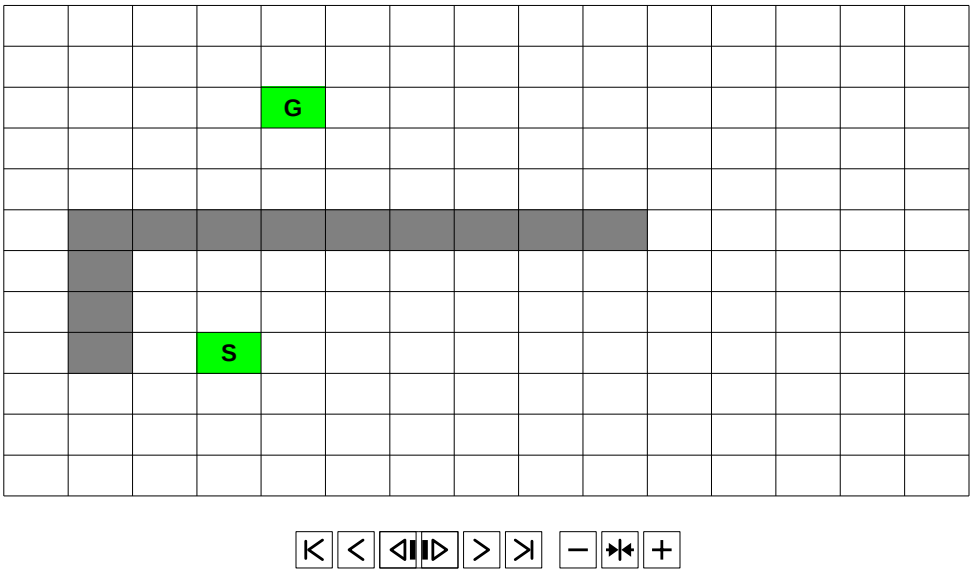


Figure 1: Animation of the algorithm running. If this doesn't play, try opening it with Acrobat Reader.

4 Formal Algorithm

When run on a grid where each step costs the same, the shortest path alrogithm reduces to a normal breadth first search. In the algorithm that follows, we make use of a queue to keep track of the fringe. We use a parent array to keep track of the row/column indices of the cell through which we travelled on the way to the current cell. The distance array keeps track of how long the path is to get from the source to the current cell. When we have found a path to the goal, we follow the parent array back to the source pushing cell indices onto the stack as we go. We can then pop the values off the stack to get the correct path from the source to the goal.

```
1:  function FINDSHORTESTPATH(Grid world)
2:      q ←new Queue;
3:      s ←new Stack;
4:
5:      Initialise ParentR[Num Rows][Num Cols] to -2;
6:      Initialise ParentC[Num Rows][Num Cols] to -2;
7:      Initialise Distance[Num Rows][Num Cols] to MAX_INT;
8:
9:      Distance[startR][startC] ← 0;
10:     ParentR[startR][startC] ← -1;
11:     ParentC[startR][startC] ← -1;
12:
13:     Enqueue(q, startR, startC);
14:     while q isn't empty and we haven't discovered the goal do
15:         currR, currC ← Dequeue(q);
16:
17:         if world[currR-1][currC] is open and undiscovered then
18:             Distance[currR-1][currC] = Distance[currR][currC] + 1;
19:             ParentR[currR-1][currC] = currR;
20:             ParentC[currR-1][currC] = currC;
21:             Enqueue(q, currR-1, currC);
22:         end if
23:
24:         if world[currR+1][currC] is open and undiscovered then
25:             Distance[currR+1][currC] = Distance[currR][currC] + 1;
26:             ParentR[currR+1][currC] = currR;
27:             ParentC[currR+1][currC] = currC;
28:             Enqueue(q, currR+1, currC);
29:         end if
30:
31:         if world[currR][currC-1] is open and undiscovered then
32:             Distance[currR][currC-1] = Distance[currR][currC] + 1;
33:             ParentR[currR][currC-1] = currR;
34:             ParentC[currR][currC-1] = currC;
35:             Enqueue(q, currR, currC-1);
36:         end if
```

```
37:
38:         if world[currR][currC+1] is open and undiscovered then
39:             Distance[currR][currC+1] = Distance[currR][currC] + 1;
40:             ParentR[currR][currC+1] = currR;
41:             ParentC[currR][currC+1] = currC;
42:             Enqueue(q, currR, currC+1);
43:         end if
44:     end while
45:
46:     if the queue is empty and we never found the goal then
47:         return No Path to goal
48:     else
49:         currR, currC ← goalR, goalC;
50:         while we haven't reached the source do
51:             Push(s, ParentR[currR][currC], ParentC[currR][currC]);
52:             currR, currC ← ParentR[currR][currC], ParentC[currR][currC];
53:         end while
54:         // Popping items off of S will give the path in the correct order
55:         return the stack, s.
56:     end if
57: end function
```

5 Input

You must read from a text file that has the following format.

- The first line will contain 2 integers, `m` and `n`, separated by a space. These numbers represent the number of rows and columns in the grid, respectively.
- The rest of the file will contain the layout of the grid.
- There will then be `m` lines each with `n` columns that represent the world.
  - A pipe symbol (`|`) represents the left or right side of the grid world. The mouse may not travel through these blocks.
  - A minus sign (`-`) represents the top or bottom of the grid world. The mouse may not travel through these blocks.
  - A `space` represents an open block in the world. The mouse may travel through this block.
  - An `x` represents an obstacal. The mouse may not travel through this block.
  - A `S` represents the source. Where the mouse starts.
  - A `G` represents the goal, i.e. the cheese.

Example input file:

```
14 17
-----------------
|                |
|                |
|             G  |
|                |
|                |
|                |
|                |
|  xxxxxxxxx     |
|          x     |
|   S      x     |
|          x     |
|                |
-----------------
```

6 Requirements

In your program you need to implement a number of functions, which are listed below.

1. `ReadWorld` must open the input file, read the first line, dynamically allocate a valid 2D character array and fill it by reading in the rest of the input file. The array should me `m×n`. The function should return a pointer to a Grid which contains the array, the width and the height of the world.

   - `Grid* ReadWorld(char* filename);`

2. You will need to implement both a queue and a stack. You should use a linked list implementation. The following functions are required for stacks and queues respectively.

   - `Stack* stack_init();`
   - `void stack_push(Stack *s, int row, int col);`
   - `void stack_pop(Stack *s, int *row, int *col);`
   - `int stack_size(Stack *s);`

   - `Queue* queue_init();`
   - `void queue_enqueue(Queue *q, int row, int col);`
   - `void queue_dequeue(Queue *q, int *row, int *col);`
   - `int queue_size(Queue *q);`

   The following Linked Lists functions are provided. You may only use these functions to access the list. The marker will detect if you attempt to access the links directly.

   - `LinkList* ll_init();`
   - `void ll_free(LinkList *l);`
   - `void ll_addFront(LinkList *l);`
   - `void ll_addBack(LinkList *l);`
   - `void ll_getFront(LinkList *l, int *x, int *y);`
   - `int ll_size(LinkList *l);`
   - `void ll_print(LinkList *l);`

3. `FindPath` must accept the Grid from `ReadWorld()` and perform the search. It should return a `Search` pointer, which contains the length of the path, the distance array, the parent array and the stack containing the relevant path. All dynamic memory allocation should occur. If there is no path from the mouse to the cheese, the path length should be -1 and the stack should be empty, not NULL.

   - `Search* FindPath(Grid *world);`

7 Grading

If you incorrectly access the Linked List, you may get 0 for both the Stack and Queue implementations.

Description	**Marks**
Correct Stack Functions	20
Correct Queue Functions	20
Correct implementation of ReadWorld()	10
Correct BFS Distances and Valid Paths	40
Correctly Detects when there is no Path	10
Total	**100**