CS1114 Assignment 3

out: Feb 24, 2012
due: **March 9, 2012 by 5pm**

1 Previously, on Assignment 2

In the last assignment we implemented several robust ways of finding the lightstick center. In this assignment, we will do even better, by using graph traversal to find all connected components of red pixels, then identifying the largest one and finding its centroid.

2 Linked lists

In order to find connected components, we need to implement a graph traversal algorithm. But to implement graph traversal, we need a way of maintaining a "todo list" that keeps track of unvisited vertices. Two options for implementing the todo list are a queue and a stack: in this part of the assignment, we will implement queues and stacks in Matlab using doubly-linked lists; this is your first task.

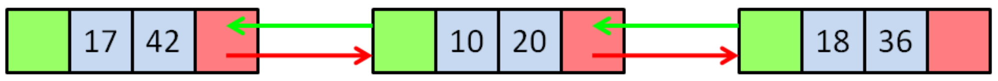
In class, we talked about how to implement a linked list that stores single integers; however, for images, it is easier to identify vertices with two numbers (the row and column). Thus, our linked lists will be a little different: they will store **two** integers in each cell. Thus, each cell will have four elements (and will therefore take up four adjacent memory elements):

1. A pointer to the previous cell in the list.
2. The row of the pixel.
3. The column of the pixel.
4. A pointer to the next cell in the list.

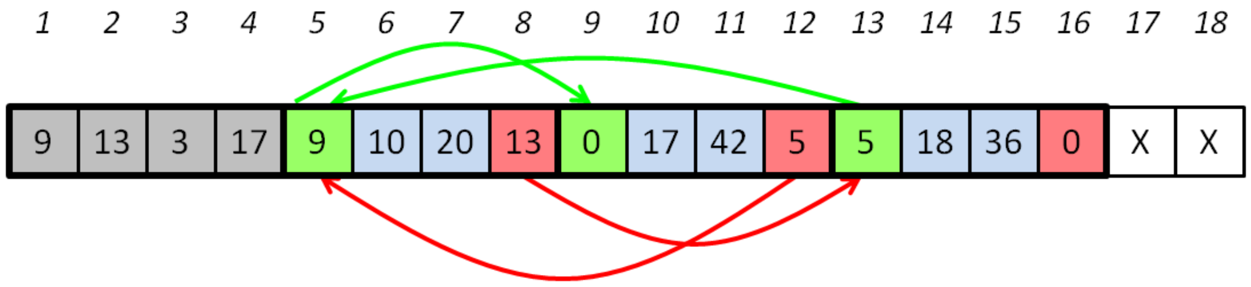
This time, the header will have four elements:

1. A pointer to the first cell in the list.
2. A pointer to the last cell in the list.
3. The size of the list.
4. A pointer to the first free cell.

(You won't have to worry about updating the fourth element—the free cell—in this assignment.) Thus a list with the elements (17,42), (10,20), (18,36) conceptually looks like:



This list could be represented in memory as:



where green entries are backwards pointers, red entries are forwards pointers, blue entries are values, and grey entries represent header information (X's are free cells). Remember that a `0` in a pointer slot is a *null pointer* signifying the beginning or end of the list.

We provide you with a function to create and list and allocate a new element:

- `newLinkedList()`: This function returns a new array to use as the "memory" for storing a linked list. The header will be properly initialized to `[ 0, 0, 0, first free cell ]`—two null pointers and a size of 0, all signifying an empty list (and a pointer to the first free memory cell). To use this function, you can enter, for instance:

  ```
  >> list = newLinkedList();
  ```

- `getNewListCell(list)`: This function takes a list, allocates space for a new cell (of size 4), and returns the index of the new cell. For instance, we can enter:

  ```
  >> [newIndex, list] = getNewListCell(list);
  ```

  if we need to create a new cell. Note that it is important to include `list` in the output variables, or else the consistency of the list will not be maintained!

Your first task is to implement a function for printing out the contents of a linked list. The output of this function should look like this (given as input the list above):

```
>> printLinkedList(list);
[17, 42] -> [10, 20] -> [18, 36]
```

⟹ Implement the `printLinkedList` function in the file `printLinkedList.m`. You may find the `fprintf` function helpful.

To create a linked list for testing, you can use the `newLinkedList` and `listInsertAtStart` functions. For instance:

```
>> list = newLinkedList();
>> list = listInsertAtStart(list, 18, 36);
>> list = listInsertAtStart(list, 10, 20);
>> list = listInsertAtStart(list, 17, 42);
```

will create the linked list pictured above.

Next, you'll need to implement a couple of functions for inserting and deleting elements. To get you started, we provide you with functions for inserting and deleting from the start of a list:

- `listInsertAtStart(list, row, col)`: This function takes a list, a row, and a column, and inserts a cell containing `(row,col)` at the beginning of the list. The function returns the new list. For instance, to insert `(17,42)` at the start of a list called `list`, we could do:

  ```
  >> list = listInsertAtStart(list, row, col)
  ```

  Again, it is important that we assign the result of the function back to the variable `list`.

  The `listInsertAtStart` function demonstrates how different cases are handled when inserting an element into a list—you can refer to it when implementing the `listInsertAtEnd` function below.

- `listDeleteAtStart(list)`: This function takes a list and deletes the first element in the list. The function returns the new list. For instance:

  ```
  >> list = listDeleteAtStart(list)
  ```

  The `listDeleteAtStart` function demonstrates how different cases are handled when deleting an element from a list—you can refer to it when implementing the `listDeleteAtEnd` function below.

You will need to implement functions for inserting and deleting from the end of the list. You will also need to implement the `listIsEmpty` function, which returns 1 if the list is empty, and 0 otherwise.

⟹ Implement two functions, `listInsertAtEnd` and `listDeleteAtEnd`, for insertion and deletion from the end of a list. `listInsertAtEnd` takes three arguments (a list, a row, and a column) and returns the new list, and `listDeleteAtEnd` takes a list and return the new list. Be careful to handle all the different cases properly, and remember that each cell has both a forwards and a backwards pointer. Headers are provided for these functions in `listInsertAtEnd.m` and `listDeleteAtEnd.m`.

⟹ Implement `listIsEmpty`, in `listIsEmpty.m`.

To test your functions, you can create a list as described above. We will also provide you with additional functions for use in testing.

3 Stacks and queues

We can now implement stacks and queues using linked lists. For stacks, we need two functions: `stackPush` and `stackPop` (as well as `listIsEmpty`, implemented above). For queues, we also need two functions: `enqueue` and `dequeue`. The `stackPush` and `enqueue` functions take a list and a row and column of a pixel, and return the updated stack or queue. The `stackPop` and `dequeue` functions take a list, and return the appropriate row and column, as well as the new list. Here are a few examples demostrating their expected functionality:

```
>> myQueue = newLinkedList();
>> myQueue = enqueue(myQueue, 1, 2);
>> myQueue = enqueue(myQueue, 10, 20);
>> [row, col, myQueue] = dequeue(myQueue);  % row, col should be 1, 2
>> [row, col, myQueue] = dequeue(myQueue);  % row, col should be 10, 20

>> myStack = newLinkedList();
>> myStack = stackPush(myStack, 1, 2);
>> myStack = stackPush(myStack, 10, 20);
>> [row, col, myStack] = stackPop(myStack); % row, col should be 10, 20
>> [row, col, myStack] = stackPop(myStack); % row, col should be 1, 2
```

⟹ Implement the five functions `stackPush`, `stackPop`, `enqueue`, and `dequeue`, in terms of the insertion and deletion functions in Section 2 (these five functions will be fairly short). Headers for these functions are provided for you in `listIsEmpty.m`, `stackPush.m`, `stackPop.m`, `enqueue.m`, and `dequeue.m`.

3 Connected components

A connected component of an undirected graph is a maximal set of vertices which has the property that you can go from any vertex to every other by traveling through the edges in the graph. A connected component in the graph can be found using depth-first search (DFS) or breadth-first search (BFS), as described in lecture. To do graph traversal, we need a “todo list” to keep track of unvisited vertices. DFS is implemented using a stack as the todo list, and BFS is implemented using queues.

1	1	1	1	0
0	0	0	0	0
0	1	1	1	0
0	1	1	0	1
0	0	0	0	1

Figure 1: Use this example to test your implementation of connected-component-finding using BFS and DFS.

We encourage you to use the builtin Matlab connected components function `bwlabel` to debug your code. It takes two parameters, as follows:

```
L = bwlabel(image, 4)
```

The number 4 indicates the neighborhood system (left, down, right, up). The size of the output matrix `L` is same as the input image, but each connected component is marked with a unique number, starting from 1 and going until the number of connected components in the graph is reached. In other words, all points in blob 1 will have a value of 1, all points in blob 2 will have a value of 2, and so on and so forth. You can use another Matlab function called `label2rgb` to visualize different connected components:

```
 >> L = bwlabel(image, 4);
 >> RGB = label2rgb(L, 'jet', 'blue');
```

You can use Matlab's `help` command to see more information about these functions.

4 What To Do

You can find templates for all of the functions you need to implement in the

`cs1114/student_files/A3`

directory. Start by copying all of the files in this folder to your `myfiles` directory.

1. Implement the linked list functions described in Section 2: `printLinkedList`, `listInsertAtEnd`, `listDeleteAtEnd`, and `listIsEmpty`. It might be helpful to look at the implementation of `listInsertAtStart`.
2. Implement the queue and stack functions described in Section 3: `stackPush`, `stackPop`, `enqueue`, and `dequeue`.

3. Implement and test **connected components** on images, using the 4-connected neighborhood system. You must use both DFS and BFS. In your DFS implementation, use the code for stacks given above, and in your BFS implementation use the code for queues. The input to these functions will be a binary image, a root pixel location, a label number, and a image labeled with the previously detected connected components, and the output will be a labeled image in which every pixel in the same component as the root pixel has the value of label. Note that you don't have to do this for each connected component in the image—we take care of this for you with functions called `CC_DFS` and `CC_BFS`.

   You can test your code by calling `CC_DFS` or `CC_BFS`, which take as input a binary image, and return a labeled image where each component has a different label (as well as the number of components). We provide you with a function `ccExample` that returns the example binary image above, which can be used for testing, e.g.:

   ```
   >> [labeled, num_components] = CC_DFS(ccExample());
   >> % CC_DFS calls your CC_DFS_student function
   ```

   There are also several binary images in the `student_files/A3` directory for use in testing.

   (a) Your **DFS** implementation should have the following prototype:

   ```
   function [ labeled ] =
        CC_DFS_student(bimage, root_row, root_col, label, labeled)
   ```

   Place your code in `CC_DFS_student.m`.

   (b) Your **BFS** implementation should have the following prototype:

   ```
   function [ labeled ] =
        CC_BFS_student(bimage, root_row, root_col, label, labeled)
   ```

   Place your code in `CC_BFS_student.m`.

4. Using your DFS implementation, compute the connected components in an input image and return only the **largest one**. Your implementation should have the following prototype:

   ```
   function labeled = big_red_student(bimage)
   ```

   Place your code in `big_red_student.m`.

   Note that it might be helpful to know that given a matrix `A`, the Matlab command `A == k` returns a binary image where all elements of `A` with value `k` are 1, and the rest of the elements are 0. The `find` function might also be helpful here.

5. Using your big red region code and your `median_vector_student.m` code from Assignment 2, find the **median center** of the biggest red region. Your implementation should have the following prototype:

   ```
   function [r,c] = big_red_median_vector_student(img)
   ```

   Place your code in `big_red_median_vector_student.m`.

6. Write a function which drives a robot based on the position of the largest connected component and its area. The goal will be to move the robot to keep the lightstick at a fixed position and size: if the lightstick center is to the left of the center of the image, the robot should turn left, and if the lightstick center is to the right of the center of image, the robot should turn right. If the area of the lightstick is larger than some constant size (say 30,000 pixels), the robot should move backwards; if it is smaller than this size, the robot should move forwards.

   You will need to use the following robot functions `robotInit`, `robotDriveStraight`, and `robotTurn`. Place your code in `CC_robots_student.m`.