Lab 1 and 2

Contents

1 Logical indexing warm-up

In logical indexing, you use a logical array for the vector subscript. MATLAB extracts the vector elements corresponding to the nonzero values of the logical array.

```
v = [1 2 3 4 5]

indices = logical([1 1 0 0 1])
% or indices = [true true false false true]

disp(v(indices))

% [1 2 5]
```

2 Conditional indexing warm-up

You can filter the elements of an array by applying conditions to the array. Applying conditions to the array returns the logical array. Applying logical

array to the array filters out not needed elements. Using `find` function you can get the indices of the elements which satisfy the conditions.

```
v = [1 2 3 1 2 3]

mask = v==3
% [false false true false false true]

equal_to_three = v(mask)
% [3 3]

equal_to_three_index = find(mask)
% [3 6]

equal_to_three_index = find(mask, 1)
% 3
```

3 Part 1: Dijkstra algorithm preparation (recommended to complete before Lab 2)

The overall task of Labs 1 and 2 is to implement the Dijkstra algorithm. For your convenience, we have divided it into multiple small steps and explain them below. Try to implement them in one file, one step at a time. Do not try to overcomplicate things from the beginning, just solve one problem at a time and go step by step while making sure every step works well. The idea is to prepare all the small routines that you will use in the Dijkstra algorithm later, so make sure that the code you are writing is as general as possible and that it will work with any adjacency matrix (remember, that the final Dijkstra algorithm should work on any given adjacency matrix).

Step 1

Create a directed weighted graph with minimum 6 vertices using adjacency matrix (instead of '1' if the edge exists, use a weight value for the edge). If there is no edge, use 0[1]. Use positive integer values for weights. Plot the graph.

[1]Please be aware that, theoretically speaking, it better to use ∞ in the adjacency matrix if there is no edge. Or, even better, have weights stored in a separate vector and use just the ordinary 0/1 adjacency matrix. This way we ensure that our algorithm works with edges that have weight 0, which is perfectly fine in Dijkstra algorithm (Dijkstra

Step 2

Create a vector to keep the graph vertices. Initialize it with numbers from 1 to $|V|$, where $|V|$ is the number of vertices.

Step 3

Create a vector to keep labels of Dijkstra algorithm for the given graph. The elements of this vector will keep the current labels, the indices are the numbers of vertices. Initialize it with infinite values for each vertex (Hint: infinite value is 'Inf' in MATLAB, initialize the vector with ones and then multiply by Inf). Change the value for the starting vertex to be 0.

Step 4

Create a vector, which contains unvisited vertices (numbers). Initialize it with all vertices (all vertices are unvisited in the beginning).

Step 5

Create a vector, which contains labels for unvisited vertices. Initialize it using vectors from Step 3 and Step 4.

Step 6

Find the minimum current label, that is the minimum element of the vector from Step 5 (you can use MATLAB built-in function to find the minimum element of the vector).

Step 7

Find the vertex, which currently has the minimum label. Hint: use vector from Step 5 and value from Step 6 to create a mask, then use function 'find' to find the indices (use 1 as a second argument to get the first index if there are several, i.e., `find(mask, 1)`), finally apply the resulting index to vector from Step 4.

---

fails to work only if there is a negative cycle in the graph). However, to simplify our implementation we suggest to stick to the assumption that our program does not support edges with weight 0 and have the adjacency matrix where 0 represents no edge.

Step 8

Remove the found vertex from unvisited vertices vector (from Step 4). Hint: use conditional indexing.

4 Part 2: Dijkstra algorithm (recommended to start doing before Lab 2)

Now, after preparing the routines, you can combine them into a program which for a given graph $G = (V, E)$ finds the shortest path between the starting vertex and all other vertices of the graph using Dijkstra algorithm. At this point you will also need to implement the routine to update labels of vertices that are adjancent to the currently visited vertex (it is in the pseudocode between steps 7 and 8).

The algorithm is described in more details in the following pseudocode[2]:

Algorithm 1: Shortest Path (Dijkstra algorithm)

Input : Weighted adjacency matrix A for graph $G = (V, E)$;
`// Steps 1 and 2`

Output: The shortest paths from vertex 1 to all vertices of G

```
1  startingVertex = 1;
2  labels = ∞, ∀v ∈ V ;                                   // Step 3
3  labels(startingVertex) = 0 ;                           // Step 3
4  unvisited(v) = v, ∀v ∈ V ;                             // Step 4
5  for i = 1 TO length(V) do
6  |  currentLabel = min label among unvisited vertices ; // Steps 5
   |   and 6
7  |  currentVertex = vertex with label currentLabel ;    // Step 7
   |  /* Routine to update labels of vertices adjacent to
   |     currentVertex                                    */
8  |  for j = 1 TO length(V) do
9  |  |  Weight = weight of edge currentVertex, j;
10 |  |  if Weight > 0 then
11 |  |  |  newLabel = currentLabel + Weight;
12 |  |  |  if newLabel < labels(j) then
13 |  |  |  |  labels(j) = newLabel;
14 |  |  |  end
15 |  |  end
16 |  end
17 |  unvisited = unvisited \{currentVertex}; ;           // Step 8
18 end
```

[2]This pseudocode is a bit different from the one presented by Valentin during lectures. This pseudocode represents the same algorithm, but it is modified to guide you and help you with the implementation.