lec16

August 12, 2016

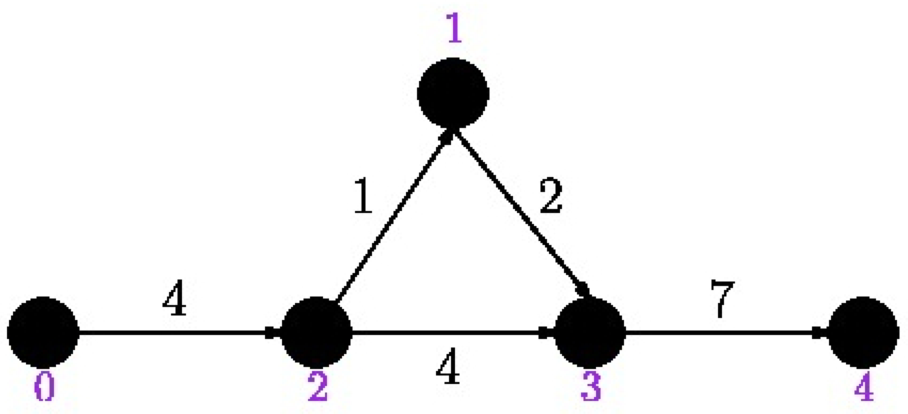
1 Memoization applied to graphs: shortest paths

Suppose we have a graph representing an airport network, a road network, etc. For example consider an airport network: vertices are airports, and edges represent direct flights from one airport to the next. Each edge now has an associated length corresponding to the length of the flight. Now how can we find the shortest path from one vertex to the others in such a graph? This can be done using recursion and memoization. The non-recursive, iterative implementation of this approach (that is, loops instead of recursion) is called the Bellman-Ford algorithm.

The basic idea is to create a recursive function `shortestPathHelper`(x, y, t) which finds the shortest path from x to y which takes at most t steps. One option is that it is the same as the shortest path taking at most $t-1$ steps, and the other is that we should travel to some vertex z first in $t-1$ steps then take the edge (z, y) in the t^{th} step. We recurse on both options and take the better of the two, and we use memoization to make the function faster. Note that if it's possible to get from x to y at all, then it is possible to do so in $n-1$ steps, where the graph has n vertices, so the length of the shortest path from x to y is `shortestPathHelper`$(x, y, n-1)$.

```
In [3]: # returns length of shortest path from x to y using at most t steps
        # B is an inverse adjacency list. That is, B[y] is a list [ [x0,w0], ...,
        # such that for each i there is an edge (xi, y) with weight wi
        def shortestPathHelper(B, x, y, t, mem, seen):
            if t == 0:
                if x == y:
                    return 0
                else:
                    return float('infinity')
            elif seen[y][t]:
                return mem[y][t]

            seen[y][t] = True

            # first option: do it in t-1 steps
            ans = shortestPathHelper(B, x, y, t-1, mem, seen)

            # second option: go to a vertex z that has an edge to y first, in
            # at most t-1 steps, then take the edge (z, y)
            for p in B[y]:
                z = p[0]
                weight = p[1]
                val = shortestPathHelper(B, x, z, t-1, mem, seen)
                ans = min(ans, weight + val)

            mem[y][t] = ans
            return ans

        # A is the adjacency list of the graph
        # A[u][i][0] is the ith neighbor of vertex u, and A[u][i][1] is the
        # weight of the edge (u, A[u][i][0])
        #
        # returns the length of the shortest path from x to y
        def shortestPath(A, x, y):
            # mem[i][j] should be float('infinity') if we can't get from x to i in
            # most j steps. Otherwise, it's the length of the shortest path from x
            # i taking at most j steps.
            mem = [ [0]*(len(A)+1) for i in range(len(A)) ]
            seen = [ [False]*(len(A)+1) for i in range(len(A)) ]

            # B is an inverse adjacency list. B[i] is a list of lists.
            # Each element of B[i] is a list [x, w] representing that (x,i)
            # is an edge in the graph, and w is its length
            B = [ [] for i in range(len(A)) ]
            for i in range(len(A)):
                for p in A[i]:
                    # p is the pair [j, length(i,j)]
                    B[p[0]] += [[i, p[1]]]

            return shortestPathHelper(B, x, y, len(A) - 1, mem, seen)
```

2 Example



title

```
In [6]: # as an adjacency list with weights
        A = [
                [[2, 4]], # (0,2) with weight 4
                [[3, 2]], # (1,3) with weight 2
                [[1, 1], [3, 4]], # (2,1) with weight 1, (2,3) with weight 4
                [[4, 7]], # (3,4) with weight 7
                [] # vertex 4 has no outgoing edges
            ]

In [7]: shortestPath(A, 0, 2)

Out[7]: 4

In [9]: # there are two routes
        # 0->2->3 and 0->2->1->3
        # 0->2->3 has total length 4+4 = 8
        # 0->2->1->3 has total length 4+1+2 = 7
        shortestPath(A, 0, 3)

Out[9]: 7

In [10]: shortestPath(A, 3, 0)

Out[10]: inf
```

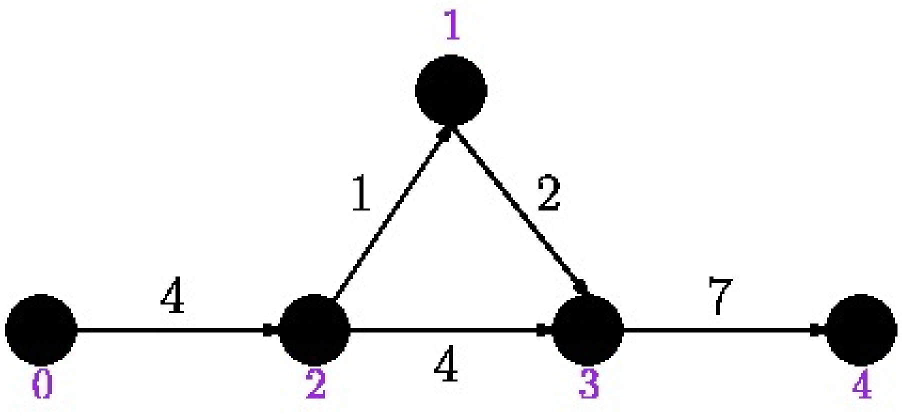
3 Finding the actual path

Typically we don't just want to know the length of the shortest path. We want to know what the shortest path is itself! What route should we take? Just as we remembered the choices we made for going to parties yesterday, we can similarly here also remember the choices we made. Below is a modified version of the above code which remembers our choices.

```
In [21]: # returns length of shortest path from x to y using at most t steps
         # B is an inverse adjacency list. That is, B[y] is a list [ [x0,w0], ...,
         # such that for each i there is an edge (xi, y) with weight wi
         def shortestPathHelper(B, x, y, t, mem, seen, choices):
             if t == 0:
                 if x == y:
                     return 0,choices
                 else:
                     return float('infinity'),choices
             elif seen[y][t]:
                 return mem[y][t],choices

             seen[y][t] = True

             # first option: do it in t-1 steps
             choices[y][t] = -1 # we use -1 to mean we actually just used t-1 steps
```

```
    ans,choices = shortestPathHelper(B, x, y, t-1, mem, seen, choices)

    # second option: go to a vertex z that has an edge to y first, in
    # at most t-1 steps, then take the edge (z, y)
    for p in B[y]:
        z = p[0]
        weight = p[1]
        val,choices = shortestPathHelper(B, x, z, t-1, mem, seen, choices)
        if weight + val < ans:
            # it is cheaper to go through z
            choices[y][t] = z
            ans = weight + val

    mem[y][t] = ans
    return ans,choices

# A is the adjacency list of the graph
# A[u][i][0] is the ith neighbor of vertex u, and A[u][i][1] is the
# weight of the edge (u, A[u][i][0])
#
# returns the length of the shortest path from x to y
def shortestPath(A, x, y):
    # mem[i][j] should be float(‘infinity’) if we can’t get from x to i i
    # most j steps. Otherwise, it’s the length of the shortest path from
    # i taking at most j steps.
    mem = [ [0]*(len(A)+1) for i in range(len(A)) ]
    seen = [ [False]*(len(A)+1) for i in range(len(A)) ]
    choices = [ [0]*(len(A)+1) for i in range(len(A)) ]

    # B is an inverse adjacency list. B[i] is a list of lists.
    # Each element of B[i] is a list [x, w] representing that (x,i)
    # is an edge in the graph, and w is its length
    B = [ [] for i in range(len(A)) ]
    for i in range(len(A)):
        for p in A[i]:
            # p is the pair [j, length(i,j)]
            B[p[0]] += [[i, p[1]]]

    return shortestPathHelper(B, x, y, len(A) - 1, mem, seen, choices)

# if no path, return [[], float('infinity')]
# else return a list of size 2: first element is an optimal path, startin
# and the second element is the weight of the path
def findPath(A, x, y):
    length,choices = shortestPath(A, x, y)
    if length == float('infinity'):
        return [[], length]
    path = [y]
```

```
        t = len(A) - 1
        while t > 0:
            if choices[y][t] != -1:
                path = [choices[y][t]] + path
                y = choices[y][t]
            t -= 1
        return [path, length]
```



title

```
In [25]: findPath(A, 0, 3)

Out[25]: [[0, 2, 1, 3], 7]
```