CSOR:4246 Assignment 2

Vibhuti Mahajan (UNI: vm2486)

16 October 2016

1 Solution

Algorithm

1. Use BFS-algorithm for finding the shortest distance between two nodes (u,v) with a simple modification.
2. Keep a record of a variable `count` for each node initialised at 0. Increment this counter for node y by the sum of $\Sigma_i count[x_i]$ whenever (x_i,y) is a tree edge or a cross edge such that dist[y]-dist[x_i]=1.
3. Return count[v] as output

Pseudo-code

```
No-of-paths(G(V,E),u,v)
    array discovered[V] initialized to 0
    array dist[V] initialized to $\infty$
    array count[V] initialised to 0
    array parent[V] initialized to NIL
    queue q
    discovered[u] = 1
    count[u] =1
    dist[u] = 0
    parent[u] = NIL
    enqueue(q, u)
    while size(q)>0 do
        x =dequeue(q)
        for (x, y) $\in$ E do
            if discovered[y] == 0 then
                discovered[y] = 1
                dist[y] = dist[x] + 1
                parent[y] = x
                count[y]=count[y]+count[x]
                enqueue(q,y)
            else if discovered[y] == 1 and dist[y]-dist[x]==1 then
                count[y]=count[y]+count[x]
            end if
        end for
    end while
    output count[v]
```

Analysis of Algorithm

Correctness:

- There can be a unique shortest length. In this case it is dist[u].

- Since the graph is undirected, we can have only back-edges and cross edges along with tree edges. Also the graph is unweighted hence shortest path between any two nodes will be the path having minimum number of nodes in the path.
- Shortest paths cannot contain cycles, hence no back-edges.
- If v is not connected to u then algorithm return count[v]=0 which is true. For connected nodes u and v, to show that algorithm gives the total number of shortest paths I will use induction on the height of the tree (h):

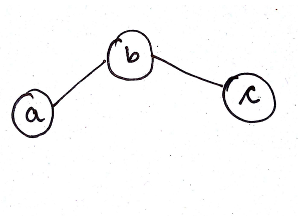
  - Base Case: If h= 1. Then either v occurs in the BFS tree of u or not.
    If v doesn't occur in the tree then count[v] remains 0.
    If v occurs in the tree then v is a leaf of the tree and the algorithm returns count[v]=1 which is true since no other shortest path is possible from u to v other than a direct edge from u to v. Let there exist another shortest path from u to v that passes through a node x. Then the the distance of the path will be ≥ 2 which is not the shortest.
  - Hypothesis : Let for some h, the algorithm return the correct number of shortest paths from u to all the nodes occurring till height h.
  - Induction step: Let the height of node x be h+1, or dist[x]=h+1. Then a path from u to x will contain atleast 1 neighbour of x, say y. There can be 3 possible states for y:
    1. y is not visited before visiting x. Then, y is not contained in any shortest path.
    2. y is visited before visiting x and dist[y]=dist[x]. This gives us a cross edge between the same layer of the BFS tree. The distance of the path will be dist[y]+1> dist[x]. Hence, a shortest path cannot contain y.
    3. y is visited before x and dist[x]-dist[y]=1. This implies that the length of the path will be dist[y]+1=dist[x], which is actually the shortest length of a path between u and x. Thus, y will be contained in the shortest path for x. By induction hypothesis, we already know the number of shortest paths for y. All these paths will be a part of the shortest path for x.

    Thus when we add the total number of shortest paths for neighbors of type 3, we get the total number of shortest paths for node x. x can be any arbitary node, hence, we get the total number of shortest paths from u to v.
- There is no duplication of shortest paths as the variable count is updated only once for a particular neighbor of type3.

Time Complexity:
We have used here a modified version of BFS by introducing a node attribute and updating it accordingly.

- The initialisations in the beginning will take atmost linear time in n.
- For a particular x, the for loop is repeated for deg(x) times and all computations are primitive calculations within he loop which takes constant time for execution.
- There can be at most n possible values of x since a node is added to the queue only if it has not been visited before. Hence, each node is visited exactly once i the while loop.
- Hence, running time = $\Sigma_{i=1}^{n} deg(x_i) = \mathcal{O}(n+m)$

2 Solution



- Here, diam(G)=dist(a,c) =2, but apd(G)=$\frac{dis(a,b)+dist(b,c)+dist(a,c)}{3} = \frac{1+1+2}{3} = \frac{4}{3}$

- $$apd(G) = \frac{\Sigma_{u,v \in V} dist(u,v)}{\binom{n}{2}}$$

  By definition, dist(u,v)≤diam(G) for all u, v ∈ V. Hence,

  $$apd(G) \leq \frac{\Sigma_{u,v \in V} diam(G)}{\binom{n}{2}}$$

  diam(G) is a constant quantity and can be taken out of the summation expression. Now we get

  $$apd(G) \leq \frac{diam(G)\Sigma_{u,v \in V}}{\binom{n}{2}} = \frac{diam(G)\binom{n}{2}}{\binom{n}{2}} = diam(G)$$

  $$\rightarrow apd(G) \leq diam(G)$$

  $$\frac{diam(G)}{apd(G)} \geq 1$$

  Hence the statement is false since there does not exist a positive constant such that $\frac{diam(G)}{apd(G)} \leq c$

- Algorithm:
  - Run BFS from the root (s) of the tree.
  - Let x be the node which is discovered at the end of the BFS.(Basically, any node having maximum distance from s will also work)
  - Run BFS again on the tree but this time taking x to be the root.
  - Let y be the node discovered last. (Again any node with maximum distance from x will also work)
  - The diameter of the tree is the distance between x and y or the height of the tree rooted at x.

  Correctness of Algorithm:
  - Tree is an acyclic and connected graph.
  - There is a unique path that exists between any 2 nodes of the tree.
  - Height of a tree is the maximum distance between the root and a leaf of the tree. We can get the height of the tree by running BFS from the root and keeping a track of the layer number of the nodes in the process.
  - Let diam(G)=dist[a,b]. We would like to show that dist[a,b]=dist[x,y]
  - There can be 2 cases:
    1. s lies on the path joining a and b
    2. s does not lie on the path joining a and b

  1. Case1: s lies on the path joining a and b; or a lies on p1.

  - Since s is the root, diameter will be the distance between 2 leaves which are the furthest from the source having no common ancestor other than s itself.
  - After running the first BFS we get the leaf (x) which is reached last and is subsequently the farthest from s. By previous statement, x is an end point of the diameter.
  - Hence, by running BFS starting from x, we can find the diameter of the tree by recording the distance between x and the node which is found last (y).
  - Hence, diam(G)=dist[x,y]

2. Case2: s does not lie on the path joining a and b.
  - WLOG, let $dist[s,a] \leq dist[s,b]$. Then, we have to show that x=b to prove the correctness of the algorithm.
  - Let us assume the contradictory, i.e. $x \neq b$, or mode generally, x is not an end point of any diameter of G$\rightarrow$ $dist[s,b] < dist[s,x]$. Then, the paths joining (s,x) and (a,b) must intersect. Let the first point of intersection be a node w.
  - As per our assumption, $dist[s,b] < dist[s,x]$
  - w lies on the intersection, hence $dist[w,b] < dist[w,x]$
  - This implies that $dist[a,w] + dist[w,b] < dist[a,w] + dist[w,x] \rightarrow dist[a,b] < dist[a,x]$
  - This contradicts the fact that (a,b) is a diameter.
  - Hence, b=x. Further, diameter of G can be found by running BFS starting from x (or b in this case).

Time Complexity This algorithm is just simple BFS running 2 times on the same G(V,E). Hence time to run this algorithm = $\mathcal{O}(m+n)$

3 Solution

Algorithm

- Use a modified version of Dijkstra's algorithm. Here the distance between any two nodes is edge weights + vertex weights. i.e. $dist[x_1, x_n] = c_i + \Sigma_{i=1}^{n}(w_{x_i x_{i+1}} + c_{x_{i+1}})$, where (x_i, x_j) is an edge in the path.
- In the distance updation step, we just need to add the vertex weight before comparing which one is smaller.

Pseudo-code: (I will use the Dijkstra-v2 version here. The same modifications can be done for Dijkstra-v3 version as taught in class)

```
Dijkstra-modified(G = (V, E, w, c), s ∈ V )
Initialize(G, s)
    S= Θ
while (S≠V) do
        Pick u so that dist[u] is minimum among all nodes in V-S
        S=S ∪ {u}
    for (u,v)∈ E do
            Update(u,v)
        end for
    end while

Update(u,v)
    if dist[v]> dist[u] + $w_{uv}$ + $c_v$ then
        dist[v] = dist[u] + $w_{uv}$ + $c_v$
        prev[v] = u
end if
```

```
Initialise(G,s):
    for v ∈ V do:
        dist[u]=$\infty$
        prev[v]=NIL
    end for
    dist[s]=$c_s$
```

Correctness of Algorithm:

- The distance between two nodes is the sum of edge weights and weights of the nodes with end points included. Proof:
  base case: dist[s]=c_s which is true because of the Initialise procedure.
  Hypothesis: For a path from s to x, all the nodes encountered before encountering x have the distance as the sum of the weights of the path from s to that node plus the weights of all nodes in the path including the edge nodes.
  Induction step: During the updation step, either dist[x] remains unchanged or dist[x]=dist[x.prev]+$w_{x.prev,x}+c_x$. If dist[x] remains unchanged then x has already been visited atleast once before visiting it again and hence our claim holds true by the induction hypothesis. Otherwise, the expression for dist[x]
  = dist[x.prev]+$w_{x.prev,x} + c_x$
  =sum of all nodes and edge weights in path from s to x.prev + $w_{x.prev,x} + c_x$
  = sum of all nodes from s to x + sum of all edge weights from s to x (in path form s to x)
- The distance at end of the algorithm is the smallest distance from s to that node. This is because it follows the Djikstra's greedy algorithm the correctness of which is proved below:

Running Time:

- The Initialising step takes $\mathcal{O}(n)$ time for initialising all the nodes
- Update procedure takes constant time for a particular u and v
- For a single iteration of the while loop, update procedure is repeated out-degree(u) times. Picking the minimum dist[u] takes $\mathcal{O}(n)$time in total.
- Hence time taken = $\mathcal{O}(n) + \mathcal{O} + \mathcal{O}(n)(\Sigma_{u\in V} out - deg(u)) = \mathcal{O}(|E|) = \mathcal{O}(n^2)$
- If we use priority heaps instead, the running time will be $\mathcal{O}(m \log n)$

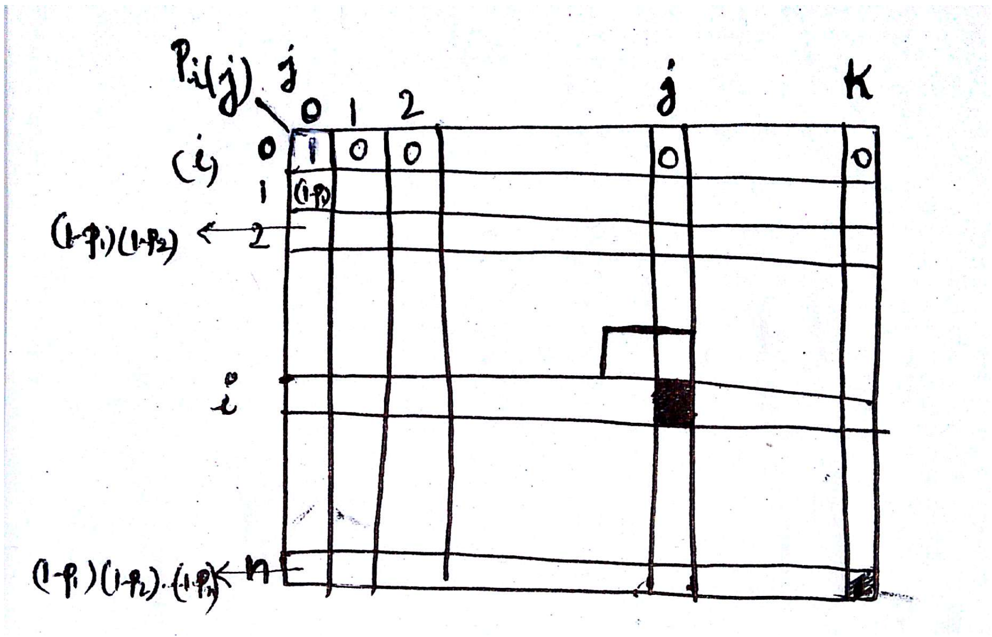
4 Solution

Algorithm:

- Let for a general case, we want to find the probability of getting exactly j heads if we flip i biased coins, denoted by $P_i(j)$.
- There can be 2 possible states till i-1 flips.
  1. There have been j-1 occurrences of heads. Hence j-th flip has to be a head. Probability of that happening = $P_{i-1}(j-1)p_n$
  2. There have been j occurrences of heads. Hence j-th flip cannot be a head. probability of that happening= $P_{i-1}(j).(1-p_n)$
- $P_i(j)$ can be written as the sum of these 2 probabilities :

$$P_i(j) = P_{i-1}(j).(1-p_n) + P_{i-1}(j-1)p_n$$

- Thus, we require just [i-1,j] and [i-1,j-1] entries from the probability matrix to find $P_i(j)$ value.



- i ranges from 1 to n-1 (but for convenient , $1 \le i \le n$) and j ranges from 0 to k. We are interested in finding $P_n(k)$.
- Initialisation of rows: We consider an extra row for 0 flips, i.e. $P_0(j)$ and initialise it as : $P_0(j) = 1$ if j=0 and $P_0(j) = 0$ otherwise.
- We begin updating the matrix row-wise as starting from row-1 (i=1) as follows:

$$P_i(j) = P_{i-1}(j).(1 - p_n); j = 0$$

$$P_i(j) = P_{i-1}(j).(1 - p_n) + P_{i-1}(j - 1).p_n; otherwise$$

- To save on the space, we can take just 2 rows (i-1,1) for calculating the probability values.

Time Complexity

- Initialisation of row takes $\mathcal{O}(k)$ time.
- Updating each cell of matrix takes constant time since the only computations are fixed number of multiplications and sum of 2 numbers $\in [0, 1]$.
- Hence time taken to fill the matrix $= c_1(k + 1) + c_2(n \times (k + 1)) = \mathcal{O}(nk)$

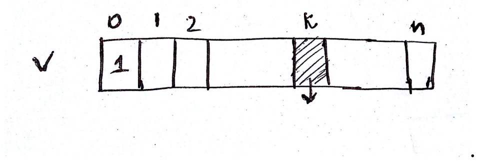
5 Solution

Algorithm:

- We can have 2 possible cases: either s[1,...,n] is a sequence of valid dictionary words or it is not.
  - If s is a valid string then s[i,...,n] is a dictionary word for some $1 \le i < n$
  - If s is not a valid string, then s[i,...,n] is not a dictionary word for any $1 \le i \le n$
- Let V_n denote that s[1,...,n] is a string of valid words. Hence,
  $V_n = 1$ if s[1,...,n] is a string of valid words; $V_n = 0$ otherwise.

- We can form a recursive definition:
  $V_n = 1$ iff $dict(s[i, n]) = 1$ and $V_{i-1} = 1$ for some$1 \leq i \leq n$

  Generally,
  $V_k = 1$ iff $dict(s[i, k]) = 1$ and $V_{i-1} = 1$ for some $1 \leq i \leq k$, or
  $V_k = \max_{1 \leq i \leq k} (dict(s[i, k]) \wedge V_{i-1})$
- For calculating V_k we require V_0 to V_{k-1}
- Allocate an array for calculating V_k and initialise $V_0 = 1$.



- Update array as $V_k = \max_{1 \leq i \leq k} (dict(s[i, k]) \wedge V_{i-1})$, ranging from k=1 to k=n. Return the solution for V_n.
  Time Complexity

- For a fixed k and i, it takes constant time to evaluate the update equation.
- For a fixed k, it takes exactly k+1 constant time computations. Hence, it take $\mathcal{O}(k)$ time to fill a cell.
- To fill the entire array:
  Time Complexity=$\mathcal{O}(1 + 2 + 3 + ... + n + (n + 1)) = \mathcal{O}(n^2)$