Kliment Mamykin, UNI 2770

Algorithms for Data Science, Homework 2

Problem 1

Let D_y be the number of shortest paths from v to y in graph $G = (V, E), v \in V, y \in V$. We need to find D_w.

Claim: Given BFS tree of graph G with level sets $\{L_0, L_1, \dots\}$, for any node y connected to node v, the number of shortest paths from v to y equals the sum of number of shortest paths from all nodes connected to y and already discovered on previous levels of BFS.

$$D_y = \sum_{(x,y)\in E, x\in L_{i-1}} D_x, y \in L_i, i > 1$$

$$D_v = 1, i = 0$$

Proof by induction

Basis:

It is true for L_0, there is only one node v at level L_0 and the number of shortest paths is 1 ($\{v\}$)

Hypothesis:

Suppose there is BFS level L_i, with all nodes at this level at the shortest distance from v. Suppose D_{node} for each node in this level contains the number of shortest paths from the root of the BFS tree to this node.

Step:

Consider a node y in level L_{i+1}. There are three cases for each node x adjacent to y:

1. x was discovered by BFS in the previous level L_i (adjacent nodes can be discovered with at most 1 level difference)
2. x was discovered by BFS at the same level L_{i+1}
3. x was discovered by BFS at the next level L_{i+2}

For cases 2 and 3 x does not belong to the shortest path between v and y, and they don't affect the number of shortest paths from v to y.

For case 1, each node x_k adjacent to y that is also in the previous layer L_i is part of the shortest path. Therefor one can make a shortest path from y to v through each of x_k, and the number of shortest paths $D_y = \sum_{x_k} D_{x_k}$

Algorithm

We use a modified BFS algorithm to traverse graph G and keep track of the number of shortest paths $D[node]$ to each of the nodes. At the end we return $D[w]$ for the final answer. BFS is an $O(n+m)$ algorithm and since we add a constant time operations, it is still an $O(n+m)$ algo.

```
Number_of_Shortest_Paths(G, start_node, end_node )
    array discovered[V] initialized to 0
    array dist[V] initialized to ∞
    array parent[V] initialized to NIL
    array D[V] initialized to 0
    queue q
    discovered[start_node] = 1
    dist[start_node] = 0s
    parent[start_node] = NIL
    D[start_node] = 1 // added
    enqueue(q, start_node)
    while size(q) > 0 do
        u = dequeue(q)
        for (u, v) ∈ E do
            if discovered[v] == 0 then
                discovered[v] = 1
                dist[v] = dist[u] + 1
                parent[v] = u
                enqueue(q, v)
            end if
            // added, if discovered and on previous level
            if discovered[v] == 1 and dist[v] < dist[u] then
                D[u] = D[u] + D[v]
            end if
        end for
    end while
    return D[end_node] // added
```

Problem 2

Will use a modified Djikstra algorithm (v3 with min-priority queue) to find shortest paths from root s to all other nodes in the graph. In each Update(u, v) procedure we will keep track of the $best[v]$ (array to represent the number of edges in the shortest path).

Correctness: assume Djikstra algorithm v3 is correct. We prove correctness of the modified algorithm using loop invariant.

Invariant: for all iterations of the while loop the following invariants are true

- Q queue contains a set of un-processed nodes ordered by distance (possibly overestimated) from s
- S (not maintained but calculated as V-Q) contains a set of processed nodes
- dist[v] contains length of the shortest path (possibly over-estimated) from s to v

- prev[v] contains previous node in the shortest path to v
- best[v] contains the min number of edges in the shortest path to v

Initialization: dist initialized to ininity except for s, where dist = 0. This also orders elements in Q to have s as the node with min key/priority = 0 and all other nodes with key/priority = Inf. prev is initialized to NIL as there are no shortest paths found yet. best initialized to infinity. S (calculated) is an empty set (nothing processed yet).

Maintenance: inside the loop, u is extracted from Q. This removes the node u from Q and implicitly updates S. The invariants for Q and S are maintained. dist, prev invariants are part of the proof for Djikstra algorith and assumed to be maintained. We just need to proof the maintenance of best[] array for each iteration. When node u is extracted from Q, it is a node on the shortest path from s through some node $x \in S$. When Update(u, v) is called for all edges from u to v, there may be 3 cases: 1) distance to v through u is larger then previously estimated distance to v through some previously processed node. In this case we dont need to do anything to maintain invariant, best[v] already contains the min number of edges in the shortest path. 2) distance to v though u is smaller then previously estimated. This means (s, ..., u, v) is the shortest path to v. Update best[v] = best[u] + 1, prev[v] = u. 3) distance to v through u is the same as through some previously processed node and both are the shortest paths (the other node in S which caused dist[v] to be updated is on the shortest path proven by Djikstra algo). Here we check which path has fewer edges and update best[v] = min(best[v], best[u]+1), prev[v] = (best[u] + 1 < best[v]) ? u : prev[v]. This maintains the invariant on array best.

Termination: The loop will terminate when the queue Q is empty, and all shortest paths have been accounted for. At this point best[] will contain the minimal number of edges in shortest paths.

Complexity: (see line by line runtime in the comments) Min-priority queue (implementation using binary tree) has BuildQueue runtime $O(n)$ and ExtractMin, DecreaseKey runtime $O(logn)$. Update(u,v) runtime consists of constant runtime lines + DecreaseKey, with resulting runtime $O(logn)$. The runtime of the algorithm
$T(n) = O(n) + O(n) + n * O(log\, n) + 2 * m * O(log\, n) = O(n\, log\, n + m\, log\, n)$

```
Shortest_Path_With_Fewer_Edges(G = (V, E, w), s)
    // inputs G: graph with V set of nodes, E set of edges, w matrix
 of weights for each edge
    //        s: initial node to calculate shortest paths from
    // returns best[] - array of minimum number of edges in a shorte
st path from s to each node
    n = |V|, m = |E|
    dist[] array size n
    prev[] array size n
    best[] array size n

    def initialize(G, s)
        dist[1..n] = infinity
        dist[s] = 0
        prev[1..n] = NIL
        best[1..n] = infinity
        best[s] = 0
    def Update(u, v)
        // u, v: connected nodes
        new_dist = dist[u] + w[u,v]
        if new_dist == dist[v] and best[u] + 1 < best[v]
            prev[v] = u
            best[v] = best[u] + 1
        else if new_dist < dist[v]
            DecreaseKey(Q, v, new_dist) // O(log n)
            dist[v] = new_dist
            prev[v] = u
            best[v] = best[u] + 1
        end if // ignore the case when new_dist > dist[v], (u,v) is
not part of the shortest path

    initialize(G, s) // O(n)
    Q = BuildQueue(V, dist) // O(n)
    while not empty(Q)      // loop executed n times
        u = ExtractMin(Q)  // O(log n)
        for each (u, v) in E // loop executed deg(u) times for each
node, 2m for all n total.
            Update(u, v)   // O(log n)
        end for
    end while
    return best
```

Problem 3

Let C_{ij} be the cost (penalty) of travel from hotel at mile post a_i to hotel at mile post a_j. Since we can start at the very beginning, we denote that location as a_0. We can pre-calculate the cost matrix C to travel from from i to j, $(0 \le i < j, 1 < j \le n)$, using formula $(200 - (a_j - a_i))^2$. C will be a top triangular matrix that we can calculate with $O(n^2)$ time.

Using Dynamic Programming approach, let an optimal cost $OPT(j)$ to travel from the beginning to hotel a_j be

$$OPT(j) = \begin{cases} 0 & j = 0 \\ \min(OPT(i) + C_{ij}) & 0 \le i < j, 1 \le j \le n \end{cases}$$

$OPT(n)$ will be an optimal cost to travel from the begining to the last stop, hotel a_n.

Proof by strong induction:

Base case: for $j = 0$ there is no need to travel, and the optimal cost $OPT(0) = 0$. For $j = 1$ the cost of travel from a_0 to a_1 is one hop with the value of cost/penalty = C_{01}.

Hypothesis: for some $j > 1$ assume $OPT(0), OPT(1), \dots, OPT(j)$ are all values of optimal travel costs up to and including j.

Induction step: for some $j + 1$, construct a set of travel options with next to last stop at some point i. Since we can only stop at a point before $j + 1$, we have the optimal costs for all stops by hypothesis. For each travel option (when i is fixed), the optimal cost will be the optimal cost to travel to point i and the cost to travel from point i to $j + 1$. Finding a minimal cost across all travel options results in a min travel cost to the point $j + 1$.

```
Optimal_Trip_Cost(A)
    // input A - array 1..n of distances
    // returns the minimal cost/panalty to travel to the last hotel
    let n = |A|
    let C[0..n-1][1..n] be array initialized to infinity values

    // calculate the cost matrix of each hop block
    for j = 1..n
        for i = 0..j-1
            C[i,j] = (200 - (A[j] - (A[i] || 0)))**2

    // find an optimal solution block
    let OPT[n] = array initialized to infinity values
    OPT[1] = C[0,1] // trivial case
    for j = 2..n
        for i = 0..j-1
            // find the min value across all next to last stop optio
ns
            OPT[j] = min(OPT[j], OPT[i] + C[i, j])
    return OPT[n]
```

Runtime analysis: The cost matrix and the optimal solutions blocks can be computed with running time $O(n^2)$, because of the arithmetic series number of solutions to solve in each block (each solution takes $O(1)$ time)

Problem 4

Let a sequence of indices $[k_1, k_2, \ldots, k_n]$ represent the optimal sequence to serve n customers with given serving times $\{t_1, t_2, \ldots, t_n\}$, such that the sequence $[t_{k_1}, t_{k_2}, \ldots, t_{k_n}]$ minimizes the total waiting time of all customers.

We initially approach the problem with a dynamic programming solution.

Let OPT(S) be the optimal total waiting time for a set of individual serving times $S = \{t_1, t_2, \ldots, t_n\}$. The recursive solution:

$$OPT(S) = \begin{cases} t_i & , S = \{t_i\}, |S| = 1 \\ \min(OPT(S - \{t_i\}) + \sum_{t_j \in S} t_j) & , |S| > 1, 1 \leq i \leq |S| \end{cases}$$

Here we notice that the problem has an optimal solution to use a **greedy algorithm**: serve the customers in the order or serving times from smallest to largest. Formally we serve customers in sequence $[k_1, k_2, \dots, k_n]$ such that $t_{k_1} \le t_{k_2} \le \cdots \le t_{k_n}$. Prove by induction:

Proposition: $P(n)$ - given a monotonically non-decreasing sequence of serving times $t_{k_1} \le t_{k_2} \le \cdots \le t_{k_n}$ the total serving time $T(n) = \sum_{i=1}^{n} \sum_{j=1}^{i} t_{k_j}$ will be minimal from any other sequence of serving times (not monotonically non-decreasing).

Base case: $P(1)$ is true because there is only one element $\{t_{k_1}\}$ to arrange, and $T(1)$ is the minimum of possible arrangements.

Hypothesis: Assume $P(n)$ is true for any n > 1.

Inductive step: Prove that $P(n+1)$ is also true. We can expand the sum $T(n+1)$

$$T(n+1) = \sum_{i=1}^{1} t_{k_i} + \cdots + \sum_{i=1}^{n} t_{k_i} + \sum_{i=1}^{n} t_{k_i} + t_{k_{n+1}}$$

where the terms up to the last two terms are $T(n)$ expanded, and the last two terms are the waiting time for the $n+1$ customer. Because $T(n)$ is minimal by the hypothesis, all terms in the sum are minimal (there are no negative waiting times). The second to last term is also minimal as it is the same as one of $T(n)$ terms. The last term $t_{k_{n+1}}$ has the largest value by hypothesis. For any other permutation of customer serving order $t_{k_{n+1}}$ would increase some term in $T(n)$, and since $T(n)$ is optimal by hypothesis, no other permitation will be optimal.

Use a modified mergesort algo that also returns indices in the original array.

```
Optimal_Customer_Serving_Sequence(A)
    // ingnore the first returned value, don't need the sorted arra
y, just indices
    _, I = Modified_Merge_Sort(A, 1, A.size)
    return I
end

Modified_Merge_Sort(A, left, right)
    // return sorted array and array of indices in A that make up so
rted array
    if right == left then return A,
    mid = left + floor((right − left)/2)
    a1, i1 = Modified_Merge_Sort(A, left, mid)
    a2, i2 = Modified_Merge_Sort(A, mid + 1, right)
    return Modified_Merge(a1, i1, a2, i2)
end

Modified_Merge(al, il, ar, ir)
    // al, il - array of values and array of indices in the left arr
ay to merge
    // ar, ir - array of values and array of indices in the right ar
ray to merge
    A = array of size |a1| + |ar|
    I = array of size |il| + |ir|
    li = 1, ri = 1, Ai = 1
    while Ai <= A.size do
        // Let x, y be the elements pointed to by pL, pR
        // Compare x, y and append the smaller to the output
        if a1[pl] < ar[pr]
            A[Ai] = al[pl]
            I[Ai] = il[pl]
            pl = pl + 1
        else
            A[Ai] = ar[pr]
            I[Ai] = ir[pr]
            pr = pr + 1
        end if
        Ai = Ai + 1
    end while
    Return A, I
```

Complexity added modifications are $O(1)$ and do not make running time worse, therefore $T(n) = O(n\,log\,n)$

Problem 5

For both part (a) and part (b) will compute the same imbalance matrix $I^{n \times n}$ defined as imbalance of a partition from index i to index j

$$I_{i,j} = \left| \sum_{x=i}^{j} A_x - AVG \right| , 1 \leq i \leq j \leq n$$

$$AVG = \frac{\sum_{l=1}^{n} A[l]}{k+1}$$

This is a top triagular matrix, and can be computed in $O(n^2)$ running time by using dynamic programming with recurrence

$$I'_{i,j} = \begin{cases} A_i & , i = j \\ I'_{i,j-1} + A_j & , 1 \leq i < j \leq n \end{cases}$$
$$I = |I' - AVG|$$

and filling it out row by row. This recurrence uses the fact that $\sum_{x=1..j} A_{i,x} = \sum_{x=1..j-1} A_{i,x} + A_{i,j}$. This computes $n(n-1)/2$ problems with each problem taking $O(1)$ time with overall $T(n,k) = O(n^2)$ running time.

```
Imbalance_Matrix(A, n, k)
    I[1..n,1..n] initialized with 0
    for j in 1..n
        for j in 1..j
            if i == j
                I[i,j] = A[i]
            else
                I[i,j] = I[i,j-1] + A[j]
    return I - Average(A, k)
end

Average(A, k)
    s = 0
    for i in 1..A.size
        s = s + A[i]
    return s / (k+1)
end
```

(a)

Define $OPT(k,l)$ as optimal partition of an array with l elements into $k+1$ partitions using k indices $j_1, \ldots, j_k$, $k < n$. Optimal solution for this problem can be expressed as recurrence:

$$OPT(k,l) = \begin{cases} I_{1,l} & , k = 0 \\ min_{j_k} \left(max(OPT(k-1, j_k), I_{j_k+1,l}) \right) & , k < j_k < l \end{cases}$$

When $k = 0$ we have 1 partition containing all elements in A up to l, and the optimal solution is entry in the imbalance matrix for indices $1, j$. When $k > 0$, we have more then 1 partition, and multiple choices of the last partition index j_k. For each choice of j_k we calculate the imbalance of this partition as a max of optimal partition up to index j_k and imbalance value from $j_k + 1$ to l ($I_{j_k+1,l}$). Overall the optimal partition choice will be the min of all partition choices for this k and l.

First, algorithm will compute the imbalance matrix I.

Second, algorithm will fill out a solution matrix of size $(k+1) \times n$ row by row, starting at $k=0$ with values $I_{1,l}, l \leq (n-k)$. Then proceed to the next row, as each row only depends on previous row. Important to note that bacause we need at least $k+1$ elements to split into k partitions, we do not need to compute elements $k \geq l$, so the elements on and below the diagonal from top left corner of the matrix will be empty. Similarly, the first partition can not start after $n-k$ element as there needs to be space to place the rest of the partirion indices. Therefor we don't need to compute solutions where $l > row + (n-k)$, which is above the diagonal from the bottom right. We only need to fill out solutions in a parallelogram shape, $n-k$ width and k height (important for the tight runtime bound). The final optimal imbalance $OPT(k, n)$ and will be found as the bottom right element of the solution matrix.

And last, algorithm will recreate the solution of partition indices $j_1, \ldots, j_k$ by backtracing from the final optimal imbalance value for each k to find j_k using the following recursion:

$$j_k = argmin_i \left(max(OPT(k-1, i), I_{i+1,l}) \right)$$

```
Imbalance_Problem(A, k)
    n = A.size
    I = Imbalance_Matrix(A, n, k)
    S = Solution_Matrix(I, n, k)
    return Recreate_Partition(S, I, n, k)
end

Solution_Matrix(I, n, k)
    S[0..k, 1..n] initialized to 0
    // fill out first for for k=0
    for l in 1..n
        S[0,l] = I[0,l]
    for row in 1..k
        for l in row+1..row+n-k
            m = infinity
            for jk in row..l-1
                m = min(m, max(S[row-1, jk], I[jk+1,l]))
            S[row,l] = m
        end for
    end for
    return S
end

Recreate_Partition(S, I, n, k)
    J[1..k] initialized to 0
    l = n
    for row in k..1 // downstep
        min_i = infinity
        for i in row+1..row+n-k
            if S[row, i] = max(S[row-1, i], I[i+1,l])
                min_i = i
            end if
        end for
        J[row] = min_i
    end for
    return J
end
```

Running time:

Imbalance matrix running time $O(n^2)$

Solution matrix computation depends on both k and n, and only fills out diagonals in parallelogram shape as described above, $(n-k)$ elements for each row with k rows. For each solution we need to consider varying number of elements to compute the min, for the first element in the row - min over one value, for the last element in the row - min over $n-k$ elements. So for each row we have arithmetic progression number of computation from 1 to $n-k$, and so the runtime of each row $T_{row}(n,k) = (n-k)(n-k-1)/2 = O((n-k)^2)$. Need to compute $k+1$ rows, so the final running time for solution matrix is $T(k,n) = O((k+1)(n-k)^2) = O(k(n-k)^2)$

Recreating the partition indices will be performed k times each time comparing $n-k$ values. Running time $T(k,n) = O(k(n-k))$

Total running time:

$$\begin{aligned} T(n,k) &= O(n^2) + O(k(n-k)^2) + O(k(n-k)) \\ &= O(n^2) + O(k(n-k)^2) \\ &= O(n^2 + kn^2 - 2nk^2 + k^3) \\ &= O(kn^2 + k^3) \\ &= O(kn^2) \end{aligned}$$

(b)

If the imbalance formula is redefined, the recurrence changes to

$$OPT(k,l) = \begin{cases} I_{1,l} & ,k=0 \\ min_{j_k}\left(OPT(k-1,j_k) + I_{j_k+1,l}\right) & ,k < j_k < l \end{cases}$$

The overall structure and runtime of the solution does not change from (a).