CS535 Design and Analysis of Algorithms - Assignment 5

Batkhishig Dulamsurankhor - A20543498

November 20, 2024

1. Even if the pivot selection is random, the time complexity will not necessarily change. It is still $O(n)$ because we are running the same algorithm with a random selection.

   With the original Median of Medians algorithm, we know that the pivot will be selected between 30 to 70 percentile when the batch size is 5. However, with random selection, the pivot quality can become worse compared to the original algorithm. Because in worst case scenario, there is a batch with really skewed values to the bottom or top. And it is kept on chosen by the random selection on each recursion, resulting in the final answer. The possiblity of this is unlikely but not 0. It will still perform better than completely random selection.

2. Professor Piks' is different from the normal skip list by introducing blocks of k items. After tossing a fair coin, each item isn't directly promoted, instead it becomes a candidate to represent the whole block for the next level. Only one of them is chosen as a representative. If a coin toss for each item in the block is tail and there is no possible candidates, the block will not have a representative in the next level.

   (a) Let's analyze the probability that all the blocks have a representative at the second level. We know that each coin toss is fair 1/2. The chance for k items to all hit tail and produce no candidates is $P_{no\_cand} = 0.5^k$.

   This makes the possiblity that there is at least one canditate is chosen $P_{cand} = 1 - P_{no\_cand} = 1 - 0.5^k$.

   Let's say we had total of n items. So the number of blocks is n/k. So the probability that all the blocks have a representative at the second level is $P_{all\_cand} = P_{cand}^{n/k} = (1 - 0.5^k)^{n/k}$.

   (b) Let's calculate the expected number of blocks that do have a representative at the second level. We already know the possiblity for a block to have a candidate is $P_{cand} = 1 - 0.5^k$.

   Therefore, the expected number of blocks having a representative is $cnt_{cand} = n/k * P_{cand} = n/k * (1 - 0.5^k)$.

3. We can convert the problem to a graph problem. Each course represents a vertix and we add edges between two courses that have overlapping schedule. Now our goal translates to finding the maximum cuts in our undirected graph $G = (V, E)$ into 3 subsets (3 classrooms) to minimize the clashes. We need to maximize the cut to make sure there is as less conflict as possible within each subset. This is similar to Randomized Min-Cut problem discussed in the class.

   The algorithm works as follows:

   - Find all overlaps. We have to check courses pairwise, so it takes $O(V^2)$.
   - Like the original algorithm, we pick each course and add it to one of the subsets uniformly at random. When we add a vertex to a subset, we also have to contract the edges. The contraction time is bounded by $O(E)$ and since we have V courses, the total time is $O(V * E)$. We repeat this until we assign all edges to the subsets.

The total time complexity of the algorithm is $O(V^2 + V * E)$.

Let's calculate the expected number of non-overlapping pairs using this algorithm. The probability of any pair of courses that have overlap assigned to different classrooms is:

$P_{no\_overlap} = 1 - P_{overlap} = 1 - 1/3 = 2/3$.

This is because we have 3 classrooms and the chances of two classes with the time conflict to be assigned into the same classroom is $P_{overlap} = 1/3$.

The expected number of non-overlapping is therefore $E * P_{no\_overlap} = 2/3 * E$.

Since the optimal number of non-overlapping pairs c^* cannot exceed E, the total number of conflicts:

$E * P_{no\_overlap} = 2/3 * E \geq 2/3 * c^*$.

4. At a glance, we could perform dfs from a vertex and at the end check if all vertices are visited. If not, we choose another vertex and so on. This algorithm works, but the time complexity will be $O(V(V + E))$ since we are running dfs for every vertex in worst case.

   We can employee graph algorithms to improve this brute force time complexity. The idea is first to detect strongly connected components and represent each components as a vertex. This converts the graph into directed acyclic graph (DAG). We can use Kosaraju's Algorithm in this step.

   With DAG, we can perform topological sort. Topological sorting gives order of vertices. Since the top vertex has the highest chance of reaching every vertex, we run dfs from this candidate. At the end of the dfs, if not all vertices are visited, there is no path between them thus making no vertex from which all vertices are reachable. On the other hand if we succeed, we can reach all vertices from this vertex (in case it is a component, all vertices from this component).

   The algorithm in pseudocode:

```
function DFS1(G, v, stack, visited):
    visited[v] <- true
    for u in G.[v].neighbors:
        if visited[u] = false:
        DFS1(G, u, stack, visited)
    stack.push(v)

function invertGraph(G):
    TG <- a new graph of N vertices
    for v in G.V:
        for u in G.[v].neighbors:
            TG.[u].neighbors.add(v)
    return TG

function DFS2(G, v, visited, DAG, map):
    visited[v] <- true
    map.add[v,DAG[-1]]
    for u in G.[v].neighbors:
        if visited[u] = false:
            DFS2(G, u, visited)
        else:
            DAG.[-1].neighbors.add(map.get[u])

function KosarajusAlgorithm(G):
    visited[size of G.V]
    stack

    for v in G.V:
        if visited[v] = false:
            DFS1(G, v, priority, visited)

    TG <- invertGraph(G)
    visited[size of G.V]

    DAG <- a new acyclic graph
    map <- key-vertices, value-components

    while stack is not empty:
        v <- stack.pop()
        if visited[v] <- false:
            DAG.addvertex[v]
            DFS2(TG, v, visited, DAG, map)

    return DAG, map


function DFS3(G, v, visited, orderstack):
    visited[v] <- true
    for u in G.[v].neighbors:
        if visited[u] = false:
            DFS3(G, u, visited, orderstack)
    orderstack.push[v]

function TopologicalSort(DAG):
    orderstack <- []
    visited[size of DAG.V]
    for v in DAG.V:
        if visited[v] = false:
            DFS3(DAG, v, priority, visited, orderstack)
    return orderstack


function DFS4(G, v, visited):
    visited[v] <- true
    for u in G.[v].neighbors:
        if visited[u] = false:
            DFS4(G, u, visited)

// starts here
DAG, map = KosarajusAlgorithm(G)
orderstack = TopologicalSort(DAG)

candidate=orderstack.pop
visited[size of DAG.V]
DFS4(DAG, candidate, visited)

if (unvisited v exists in visited):
    return -1
else:
    return map.findValue[candidate]
```

The time complexity of the algorithm is:

- Kosaraju's Algorithm:

  - First dfs that populates stack is $O(V + E)$.
  - Inverting graph is $O(V + E)$ because we are inverting direction for each edges for adjacency list.
  - Second dfs is also $O(V + E)$.

  The total time complexity for Kosaraju's Algorithm is $O((V+E)+(V+E)+(V+E)) = O(V+E)$.

- Topological sort is dfs that eventually traverses the whole graph while keeping order in stack. Adding element to stack is constant time. So the time complexity for this step is $O(V + E)$.
- Final dfs for the candidate is also $O(V + E)$.

The total time complexity of the algorithm is $O((V + E) + (V + E) + (V + E)) = O(V + E)$ linear time.

5. For the three properties to be equal, they need to imply each other. In other words, we need to prove that $(a) \Rightarrow (b)$, $(b) \Rightarrow (c)$ and $(c) \Rightarrow (a)$.

   - $(a) \Rightarrow (b)$
     We are given that removing any one vertex cannot disconnect graph. So we need to remove at least two vertices to disconnect the graph, say v and u. Let's say there is only one vertices-disjoint paths between vertices v and u. However, we know that removing u will not disconnect the graph and meaning we can reach u from v with at least two vertices-disjoint paths. This contradicts property (a) so (a) must imply (b).
   - $(b) \Rightarrow (c)$
     We are given that between any pair of vertices $u, v \in V$, there exist at least two vertices-disjoint paths. Let's assume that the two vertices-disjoint paths are P_1 and P_2. We can reach from u to v using P_1. On the way back, we can return from v to u using P_2 which is a vertices-disjoint path from P_1. We travelled from u to v back to u using disjoint paths and this forms a cycle connecting these vertices. Therefore property (b) implies property (c).
   - $(c) \Rightarrow (a)$
     We are given that for each pair nodes $u, v \in V$, there exists a simple cycle connecting u and v. Let's assume that property (a) doesn't hold and removing a vertex t disconnects a graph into V_1 and V_2. This means that t is an articulation point in the graph G. But we know that there is cycle connecting a vertex v in V_1 and a vertex u in V_2. Removing t can destroy the cycle but can't disconnect these two vertices. So t cannot be an articulation point and there must be another vertex s that forms a cycle containing t, u and v. This results in a contradiction thus property (c) must imply property (a).

6. If we represent junctions are vertices and roads as edges in a graph, we have to traverse the graph by visiting each edge once. This means we have to find Eulerian path in the graph. Here I am making assumption that a road can be one way or the graph is directed.

   - First we have to determine if such path exists in the graph. We need to calculate all in-degree and out-degree edges for each vertex. If the number of in-degree and out-degree are equal and there are 0 or 2 vertices with odd number of edges, we have a valid path.
   - We start with a vertex with odd edges and more out-degree than in-degree.
   - Then perform dfs until we reach the other odd number edge vertex. If we traversed all the edges, this means we found the path. If not, we have to backtrack and take different path. We should keep on doing this until we traverse all the edges.

```
countDegree(G):
    for v in G.V:
        for u in v.neighbors:
                out[v]++
                in[u]++

hasEulerianPath(in, out):
    start,end = 0,0
    for i size of G.V:
        if out[i]-in[i]>1 || in[i]-out[i]>1:
            return false
        if out[i]-in[i]==1:
            start++
        if in[i]-out[i]==1:
            end++
    if (end==0 && start==0) || (end==1 && start==1):
        return true
    else:
        return false

startVertex(G):
    start=0
    for i size of G.V:
        if out[i]-in[i]==1:
            return i
    return start

dfs(cur, path):
    while(out[cur] != 0):
        out[cur]--
        next = G.V[cur].neighbors[out[cur]]
        dfs(next)
    path.push(cur)

G <- adjacency lists
in <- [] size of V
out <- [] size of v

countDegree(G)
if !hasEulerianPath(in,out):
    return false

path <- stack
dfs(startVertex(G), path)

res=[]
while(stack):
    res.add(stack.pop())
return res
```

The time complexity for if Eulerian path exists is $O(E)$ since it count degree for all the edges. The dfs is linear time $O(V+E)$. So the total time complexity is $O(E)+O(V+E)=O(V+E)$.