Amortized Cost Analysis & Shortest Path & Minimum Spanning Tree

Julius January 31, 2024

1 Amortized Cost Analysis

Amortized cost analysis is performed on the cost of operations on a data structure.
An algorithm does not have an amortized cost, because it might cause an exaggeration of cost.

1.1 Aggregate Method

Specify the set of operations involved, show that a sequence of n operations takes **worst case time** $T(n)$ **total**.
In the worst case, the amortized cost (average cost) of each operation is $T(n)/n$.
For example, consider a stack starting form empty with the following operations: Push, Pop, and Multipop, which would clear the stack and pop all the existing elements. The worst case time for Push and Pop is $O(1)$, for Multipop is $O(n)$, worst case runtime complexity $O(n^2)$, which is actually **exaggerated**.
However, Multipop takes $O(n)$ time iff there are n elements pushed on the stack. So actually, the worst case total time $T(n)$ for n operations from empty stack is $\Theta(n)$, and the average cost (amortized cost) of each operation is $T(n)/n = \Theta(1)$. So the worst cast runtime complexity is actually $\Theta(n)$, which is more **realistic**.

1.2 Accounting Method

Assign different charges to different operations. If the charge of an operation exceeds its actual cost, the excess is stored as credit. The credit can later help pay for operations whose actual cost is higher than their amortized cost. **Total credit at any time should be non-negative = Total amortized cost - Total actual cost**
Consider the same example of stack operation. We assign the following charges: Push 2, Pop 0, Multipop 0, which is actually the amortized cost $\Theta(1)$ of each operation.
The Accounting Method gives us the flexibility to accout different amortized costs for different operations. And not every amortized cost is constant time. It could in some data structure be logarithmic or linear time.

1.3 Fibonacci Heap

Fibonacci heaps are loosely based on binomial heaps. It is a collection of min-heap trees similar to binomial heaps. However, trees in a Fibonacci heap are not ordered and not constrained to be binomial trees.
There is **a pointer to the minimum node** in the heap, so the Find Min operation takes $O(1)$ time.
For **delete operation**, we cannot procrastinate anymore and have to tidy up the data structure, and there would be so many pointers connecting things together to help perform these merging and splitting operations.

Operations	Binary Heap(Worstcast)	Binomial Heap(Worstcase)	Fibonacci Heap(Amortized)
Find Min	O(1)	O(lgn)	O(1)
Insert	O(lgn)	O(lgn)	O(1)
Extract Min	O(lgn)	O(lgn)	O(lgn)
Delete	O(lgn)	O(lgn)	O(lgn)
Decrease Key	O(lgn)	O(lgn)	O(1)
Merge	O(n)	O(lgn)	O(1)
Construct	O(n)	O(n)	O(n)

Ex1. The values $1, 2, \ldots, 63$ are all inserted in any order into an initially empty min-heap. What's the smallest number that could be a leaf node?
Because for a complete binary tree $63 = 2^6 - 1$, the nodes are inserted from left to right and from top to bottom. There should be at least 6 levels, and the smallest number that could be on the 6th level is 6 with 1->2->3->4->5->6 as the left-most branch.
Ex2. Suppose we have two min-heaps, A and B, with a total of n elements between them. We want to discover if A and B have a key in common. Give a solution that takes $O(nlgn)$ time.
Look at the min a_i, b_i from A and B and compare them. If they are equal, we find a common key. If not, pop the smaller element from corresponding heap and continue this process until one of the heap is empty.
Because we have The total n elements in two heaps and each pop takes $O(lgn)$ time, the total time is $O(nlgn)$.

2 Shortest Path

2.1 Positive weighted graph with cycles

Shortest Path with only positive weight: Given $G = (V, E)$ with $w(u, v) \geq 0$ for each edge $(u, v) \in E$, find the shortest path from vertex $s \in V$ to $V - \{s\}$.

Dijkstra's Algorithm:

```
Initially, $S = Null, d[s] = 0$, $d[u] = \infty$ for all other nodes u.
While $S \neq V$ :
    Select a vertex $t \in V - S$ with the $min\{dist[t]\}$ into Set S.
    $S = S \cup \{t\}$.
    Use $t$ to update all the distance of its neighbor nodes $j$ by $dist[j] = min(dist[j], dist[t] + w(t, j))$ if node $j$ is in $V - S$. And if we do update, record $parent[j] = t$.
Repeat until $S = V$, that's all nodes have find their shortest path from s.
```

At each step, Dijkstra's algorithm finds the shortest path to a new node in the graph.

Proof by strong induction. Base case: $|S| = 1, S = \{s\}, dist[s] = 0$.
Inductive hypothesis: Suppose the claim holds when $|S| = k$ for some $k \geq 1$, i.e. Dijkstra's Algorithm has found the shortest path from s to the first k nodes in the graph.
Inductive step: we now grow S to size k+1 and prove that we have found the shortest path to a new node. Suppose Dijkstra's alg. chooses v at the k+1 iteration, and that u is the neighbor that gives v its shortest distance from s. We claim that $dist[u] + w(u, v)$ is the shortest path from s to v:
Let's assume there is a shorter path to v, which must cross the boundary of S and $V - S$. Let $w(u', v')$ be the edge crossing the boundary on that path. Because Dijkstra's algorithm picks node v, we must have $dist[u] + w(u, v) < dist[u'] + w(u', v')$. Since there are no negative weight edges, the path from v' to v will only add to the length of that path. □

Dijkstra's Algorithm with Priority Queue:

```
Initialize a min-heap q to store the vertices in $V - S$ in form of <distance, node idx>, $dist[s] = 0, dist[u] = \infty$ for all other nodes u. Push $\{0, s\}$ into the heap and S is NULL.
While heap is not empty:
    Pop the top element in heap. (n times)
    If the node in the element is already in S(is already visited), continue.
    Put the node into S (Label the node as visited).
    Iterate all the neighbors of this node: For each vertex $j \in Adj(t)$ :
        If we can update the distance of the neighbor node, update it and push the new distance and node idx into the heap. (m times)
        If dist[j] > dist[t] + w(t, j):
            Decrease_key(heap, j, dist[t]+w(t,j));
Repeat until heap is empty, that's all nodes have find their shortest path from s. If there is still infty distance in the dist array, it means the node is not reachable from s.
```

Time Complexity	Binary Heap(Worstcast)	Binomial Heap(Worstcase)	Fibonacci Heap(Amortized)
n Extract-Min's	O(nlgn)	O(nlgn)	O(nlgn)
m Push-Update's	O(mlgn)	O(mlgn)	O(m)
Total	O(mlgn)	O(mlgn)	O(m+nlgn)
Sparse Graphs	O(nlgn)	O(nlgn)	O(nlgn)
Dense Graphs	O(n^2lgn)	O(n^2lgn)	O(n^2)

Even for dense graphs, the Fibonacci heap implementation will not outperform the binary heap implementation in pratice due to: **Higher memory requirements and Higher constants involved**
Both Classical and Priority Queue Dijkstra can not ensure get the correct shortest path for graph with negative weight but no negative cycles. The generalized Dijkstra, allowing a vertex to be enqueued more than once (more like the following Bellman-Ford Algorithm), is correct in the presence of negative edge weights but no negative cycles, but its running time is exponential in the worst case.

2.2 Weighted Graph with no negative cycles

Bellman-Ford Algorithm: Given $G = (V, E)$ with $w(u, v)$ for each edge $(u, v) \in E$, find the shortest path from vertex $s \in V$ to $V - \{s\}$ with at most k edges. If there is a negative cycle, no shortest path.

Bellman-Ford Algorithm O(|V|*|E|) or O(k*|E|):

```
Initialize $dist[s] = 0, dist[u] = \infty$ for all other nodes u.
For i = 0 to k, iterate k times for at most k edges:
    Back up the current $dist$ array to $dist\_old$.
    For each edge $(u, v) \in E$:
        $dist[v] = min(dist[v], dist\_old[u] + w(u, v))$.
Return dist array, and check if dist[u] > INF / 2: there is no path from s to u.
```

2.3 Directed Acyclic Graph

Shortest Path with only positive weight for directed acyclic graph: Given $G = (V, E)$ with $w(u, v) \geq 0$ for each edge $(u, v) \in E$, find the shortest path from vertex $s \in V$ to $V - \{s\}$.

Topological Sort and Relaxation O(|V|+|E|):

```
Find the Topological order of the graph by Kahn's Algorithm in O(|V|+|E|).
Initialize $dist[s] = 0, dist[u] = \infty$ for all other nodes u.
For each node $u$ in the topological order:
    For each edge $(u, v) \in E$:
        $dist[v] = min(dist[v], dist[u] + w(u, v))$.
Return dist array.
```

3 Minimum Spanning Tree for Connected Weighted Undirected Graph

Spanning Tree: Any tree that covers all nodes of a graph, if n nodes with only n-1 edges.
Minimum Spanning Tree: A spanning tree with minimum total edge cost.
Union-Find Set: A data structure that keeps track of a set of elements partitioned into a number of disjoint (non-overlapping) subsets. It provides operations for **MakeSet O(1), FindSet O(lgn), and Union O(1)** with Array implementation.
The following three algorithms are all for undirected weighted graphs, we need to use ChuLiu/Edmonds for Minimum Spanning Arborescence (MSA) for directed weighted graphs.

3.1 Kruskal's Algorithm

Kruskal's Algorithm with Union-Find Set O(mlgm + mlgn):

```
Sort all the edges in the graph by their weights. O(mlgm)
Initialize the weights of edges res = 0, and the number of edges in the MST cnt = 0.
For each edge $(u, v) \in E$ in the sorted list of edges:
    If $Find\_Set(u) \neq Find\_Set(v)$: O(mlgn)
        combine two sets $p[fa] = fb; res+ = edge[i].w; cnt + +;$
If $cnt < |V| - 1$, there is no spanning tree for the graph.
Else return res.
```

Fact: Let S be any subset of nodes that is neither empty nor equal to all nodes set V, and let edge e = (u, v) be the minimum cost edge with one end in S and the other end in V-S. Then every MST contains the edge e.

Proof by Contradiction. Suppose there is an MST does not contain e, then we can add e to this MST, there must be an cycle and this cycle must contains another edge (u', v') with u' in S and v' in V-S with $w(u', v') \geq w(u, v)$. We can replace (u', v') with (u, v), and the total weight of this MST will not increase. □

Proof the correctness of Kruskal's Algorithm: Using the above fact, we can get each edge chosen by Kruskal's algorithm is the lowest cost edge connecting S to V-S, then these edges must be in MST.
Limitation: Kruskal's algorithm is Not Designed for Directed Graphs, so it can't get Minimum Spanning Arborescence (MSA); Can't handle the graph with negative weight edges.

3.2 Prim's Algorithm

```
Prim's Algorithm O(mlgn):
Initialize the distance of every node to tree as $\infty$
Initialize the MST weight res = 0;
for(int i = 0; i < n; i++):
    Look through all nodes not labeled for the minimum distance node t.
    If i > 0 and dist[t] == INF, return INF; ecause this node is not connected with set.
    If i > 0, add the distance of t to res.
    Label t as visited by st[t] = true;
    Update the distance of all the unvisited neighbors of t $dist[j] = min(dist[j], w(t, j))$
Return res.
```

Proof the correctness of Prim's Algorithm: Also use the above fact, and because each edge chosen by Prim's algorithm is the lowest cost edge connecting S to V-S, then these edges must be in MST.
Prim's Algorithm is really similar to Dijkstra's Algorithm. $dist[N]$ in Dijkstra is the distance of every node to source node, while in Prim is the distance of every node to current spanning tree nodes set. Consequently, priority_queue< PII, vector<PII>, greater<PII> > can help to find the minimum distance node not in set. Prim's Algorithm requires all nodes can be reached from the source node, this might not be true for directed graphs.

3.3 BFS/DFS for MST of Sparse Graph

Sparse Graph: A connected undirected graph G(V, E) with a small number of edges relative to the number of nodes, say |E| = |V| + C. We can find the MST of G in O(|V|) time.

```
BFS/DFS for MST of Sparse Graph O(n):
Say |V| = n, |E| = n + C.
Run BFS/DFS C+1 times:
    If we find a cycle, then remove the highest cost edge in the cycle.
    else the graph is not connected.
Repeat C+1 times, there is no cycle in the graph, there are n-1 edges left for the MST.
```

3.4 Reverse-Delete Algorithm

```
Backward version of Kruskal's Algorithm (Reverse-Delete in O(m(m+n))):
Sort the edges of E into a non-increasing order of cost. O(mlgm)
For each edge $(u, v) \in E$ in the sorted list of edges: O(m)
    If removing $(u, v)$ does not disconnect the graph, remove it. (Remove the edge and try to
find a path by DFS/BFS in O(m+n))
Return the remaining edges.
```

Proof the correctness of Reverse-Delete: Fast: the highest cost edge in a cycle cann't belong to a MST, because we can substitute a lower cost edge for this highest cost edge, which does not destruct MST. Then we can use this fact to show that every step Reverse-Delete is eliminating an edge that can't belong to any MST.

4 Application Problems

4.1 Hardy want to find a route that goes entirely uphill and then entirely downhill during his running from home to workplace. There is a map with k intersections and m uphill or downhill roads. Give an efficient algorithm to find the shortest path that satisfies Hardy's requirement.

Find the shortest path from home and workplace using Dijkstra's algorithm. When there are intersections of two shortest paths finding, we can update the res = min(res, dist(home, intersection) + dist(workplace, intersection)). We can stop search until any one of Dijkstra's searching reach one node with shortest distance longer than res.

4.2 Given a Directed Acyclic Graph as several career paths available in industry. Each node represents a position and there is an edge from node v to node u if and only if v is a prerequisite for u. Top positions are the ones which are not prerequisites for any positions. The cost of an edge (v, u) is the effort required to go from one position to another. Ivan wants to start a career and achieve a top position with minimum effort. Provide an algorithm with the same run time complexity as Dijkstra's algorithm.

Just add a vitual node s with 0 cost edge to all entry nodes, and then use Dijkstra's algorithm to find the shortest path from s to the first Top position.

4.3 Use Stack with Push and Pop operations in O(1) to implement a FIFO queue with Enqueue and Dequeue operations. What's the amortized cost of each operation?

Worstcase: (n-1) enqueue and 1 dequeue with $TotalTime = (n-1) + 2*(n-1) + 1 = 3*(n-1) + 1$, so $T(n) = \Theta(n) \rightarrow T(n)/n = \Theta(1)$. For accounting method, we can charge 3 for enqueue and 0 for dequeue. Because the actual dequeue need additional one pop and push before pop.

4.4(a) Suppose we are given an instance of MST problem on graph G. Assume that all edges costs are distinct. Let T be a MST for this instance. Now suppose that we replace each edge cost c_e by its square c_e^2. Is T still a MST for the new instance?

If there are edges with negative weights, the MST may not be unique. If all edges are positive, then T is still a MST for the new Graph.

4.4(b) Suppose we are given an instance of MST problem on graph G. Assume that all edges costs are distinct. Let T be a MST for this instance. Now suppose that we replace each edge cost c_e by $c_e + 1$. Is T still a MST for the new instance?

Yes, because the $+1$ will not change the order of the edges, and the MST is still the same.