6.854 Advanced Algorithms

Problem Set 2

John Wang
Collaborators: Jason Hoch

Problem 1-a: Consider a second-level table, that in the static case uses $f(s) = O(s^2)$ space to store s items. We gave a construction of such a table that succeeds with probability $1/2$. Suppose that we take the following approach to insertions: if inserting the sth item violates perfection of the table, repeatedly build a table of size $f(2s)$ until you get a perfect one. Show that when s items are inserted (one at a time, with rebuilds as necessary) the expected total cost of all the rebuilds is $O(s)$.

Solution: Let us look at the second-level table after s items have already been inserted. We want to find out the number of rehashes that we did. To do this, we will look at inserting items $s/2$ through items s. Let us say we have just rehashed the table before we insert the $s/2$ item. Then we know that the size of the table is $(2s/2)^2 = s^2$. We know that the probability that items i and j collide is $1/s^2$. This means that the expected number of collisions is $\binom{s}{2}\frac{1}{s^2} \leq \frac{s^2}{2s^2} = \frac{1}{2}$.

The number of rehashes h we have to do, then is 0 times the probability of no collisions times $\frac{1}{2}$ times the number of rehashes we have to do if we do rehash. This can be written as $h = \frac{1}{2}0 + \frac{1}{2}(1+h)$, so that we find $h = 1$. Thus, we only expect to rehash a constant number of times when we insert items $s/2$ to s.

The total amount of work for the rehash is the number of items in the data structure, which is s. Therefore, we can use this argument recursively for items from $s/4$ to $s/2$, etc. This gives us the expression of the total amount of work: $\sum_{i=1}^{\log s} 2^i = 2^{\lg s} = O(s)$. Thus, we expect to do $O(s)$ work for all of the rebuilds. □

Problem 1-b: Let s_i be the number of items in bucket i in the second level table. Make a similar modification for the top level: any time the table on n items has bucket sizes s_i violating the requirement that $\sum s_i^2 = O(n)$, we rebuild the top table and all the second level tables. Show the expected total time is $O(n)$ when n items are inserted.

Solution: First, we will prove that we can rehash the entire table in $O(k)$ time, where k is the current number of elements in the table. Let p be the probability that choose a new hash function from the universal hash is successful. We know it requires ck time to rehash (for some constant c) and check if it is successful. Thus, we have a running time of $T(k) = pck + (1-p)T(k)$ for the entire series of rehashes (we keep picking new hash functions and testing them until one of them works). Thus, we can solve for $T(k)$ and we find that $pT(k) = pck$ so that $T(k) = O(k)$.

Now, we want to prove that if there are originally k elements in the data structure, then while inserting another k elements so that there are a total of $2k$ elements in the data structure, we will only expected $O(1)$ rehashing operations. We can see this by looking at the expected amount of space in the secondary arrays. We know that when there are $2k$ elements, we have expected $4k-1$ space used up in the secondary arrays (by the analysis of the space of secondary arrays given in class). This means that $E[\sum s_i^2] = 4k-1$ where s_i is the size of the ith bucket.

Now we can use Markov's inequality to bound the probability of rehash $P(\sum s_i^2 \geq kc)$:

$$P(\sum s_i^2 \geq kc) \leq \frac{1}{kc} E[\sum s_i^2] \tag{1}$$

$$= \frac{4k-1}{kc} \tag{2}$$

$$\leq \frac{4}{c} \tag{3}$$

We can use this to find the expected number of collisions h. We know that with probability $1 - \frac{4}{c}$, we will have no collisions. However, we know that with probability $\frac{4}{c}$, we will have one or more collisions. Thus, we

find:

$$h \leq \frac{c-4}{c}0 + \frac{4}{c}(1+h) \tag{4}$$

$$h\left(\frac{c-4}{4}\right) \leq 1 \tag{5}$$

$$h \leq \frac{4}{c-4} \tag{6}$$

Since we can pick $c > 4$ and we know that c is a constant, we find that the expected number of collisions is $h = O(1)$. This means that the total expected work done is $O(k)$ after k insertions, since each rehash requires $O(k)$ time and we only expect a constant number of them. After n insertions then, we would expect to have done $O(n)$ work. □

Problem 1-c: Now consider deletions. When an item is deleted, we can simply mark it as deleted without removing it (and if it reinserted, unmark the deletion). This makes deletion cheap. But if there are many deletions, the table might end up using much more space than it needs, which could mess up the amortized bounds of the previous parts. To fix this, we can rebuild the hash table any time the number of holes (items marked deleted) exceeds half the total items in the table. Show that this cleanup works does not change (can be charged against) the amortized $O(1)$ cost of insertions.

Solution: Let us use the potential function $\Phi = 2\#$ holes. Now, there are two functions that we will analyze: delete and cleanup. For delete, all we have to do is mark an item as deleted, which requires constant work. However, marking the item as deleted increases the potential by 2. Thus, the amortized runtime of delete is $a_i = 1 + 2 = 3 = O(1)$.

Now, let us examine cleanup. We know that we will rebuild a table only when the number of holes is less than half the size of the table. Let s be the size of the current table. Then we see that cleanup requires $s/2$ expected time to build the table and check if there are any collisions. It requires another $s/2$ cost to insert the items. The potential decreases by s, however, since we lose at least $s/2$ holes. This means that the amortized time of cleanup is $a_i = s - s = O(1)$.

Therefore, we see that deletions require $O(1)$ amortized time. □

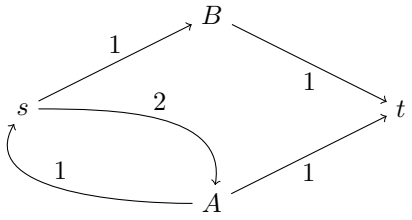
6.854 Advanced Algorithms

Problem Set 2

John Wang
Collaborators: Jason Hoch

Problem 2-a: In any maximum flow, and for all vertices v and w, either the raw flow from v to w or the raw flow from w to v is 0.

Solution: False. Consider the following graph.



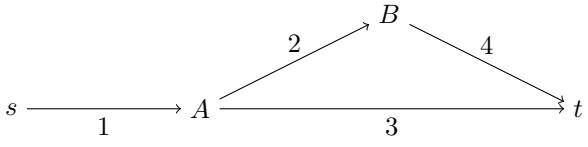
It is clear that the maximum flow is 2, where flow comes into t from nodes A and B, each with a flow of 1. However, we can see that there is non-zero raw flow from s to B and from B to s. Thus, there is a maximum flow where there exist two vertices v and w which break the above conjecture. □

Problem 2-b: There always exists a maximum flow such that, for all vertices v and w, either the raw flow from v to w or the raw flow from w to v is zero.

Solution: True. We will show this by contradiction. Suppose this is not the case. Then there is raw flow such that on some vertices v and w, the raw flow from v to w and from w to v are nonzero for all flows which are maximal. This means that any flow with zero raw flow in each direction is not a maximum flow (of flow f), so that this new flow would be less than f. Now, consider what happens when we set the raw flow from w to v to be zero, and the raw flow from v to w to be $rf(v,w) - rf(w,v)$ where rf denotes raw flow. In this way, we have created a graph G' which has the same amount of flow as before towards the sink (if not, the switch v and w) as in the original graph. This means that the new flow is also a maximum flow. However, we know that the flow on the vertices of the rest of the graph are unchanged, and we also assumed that there does not exist any maximum flow with zero raw flow between any vertices v and w. This is a contradiction. □

Problem 2-c: If all directed edges in a network have distinct capacities, then there is a unique maximum flow.

Solution: False. Consider the following counterexample.



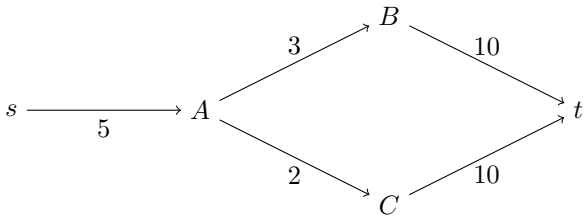
In the graph above, the maximum possible flow is $F = 1$. However, there are two possible ways that this flow can be achieved. In the first case, a flow of capacity 1 can move along the path S, A, t. In the second case, a flow of capacity 1 can move along the path S, A, B, t. □

Problem 2-d: If we replace each directed edge in a network with two directed edges in opposite directions with the same capacity and connecting the same vertices, then the value of the maximum flow remains unchanged.

Solution: False. Consider the simple graph with two nodes s and t. The only edge in this graph is a directed edge from t to s of capacity v. Clearly, the maximum flow from s to t is zero, because there is no way to reach t. However, if we augment the graph with two edges in opposite directions, both of capacity v, then the maximum flow becomes v. □

Problem 2-e: If we add the same positive number λ to the capacity of every directed edge, then the minimum cut (but not its value) remains unchanged.

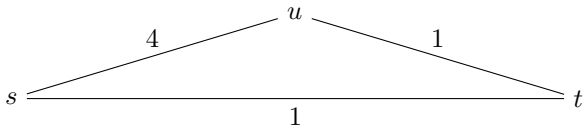
Solution: False. Consider the graph below.



We see that on the original graph, the unique $s-t$ cut is given by $S = \{s, A, B, C\}$ and $T = \{t\}$. However, when $\lambda = 1$ is added to all of the edge capacities, we see that the new unique $s - t$ cut is given by $S = \{s\}$ and $T = \{A, B, C, t\}$, since the bottleneck appears at A when its capacity is only increased by 1, and the capacities of both outgoing edges are both increased by 1, leading to a total increase of 2. □

Problem 2-f: Flow is transitive: if in a graph there is a flow of value v from s to t and there is a flow of value v from t to u, then there is a flow of value v from s to u.

Solution: False. Consider the following graph:



From s to t, there is a maximum flow of 1. From t to u there is a maximum flow of 1. However, from s to u, there is a maximum flow of 5. Thus, it is clear that the flow from s to u is not equal to the flow from s to t then from t to u, since there are other paths for the flow to travel on. □

6.854 Advanced Algorithms

Problem Set 2

John Wang
Collaborators: Jason Hoch

Problem 3-a: Given the table as described above, give an efficient, flow-based algorithm for deciding if there is some set of cell values that produces the disclosed values.

Solution: We will create a flow network with a source s and a sink t. Connected to the source, we will have nodes n_{r_i} which represent rows. There will be p of these nodes, and an edge (s, n_{r_i}) connecting the source to the node. The capacity of (s, n_{r_i}) will be the ith row sum of r_i. Next, we will have pq nodes $n_{d_{ij}}$ which represent the element d_{ij} in the matrix. Each node will be connected to the corresponding row node. Thus, if d_{ij} is in column i, then there will be an edge $(n_{r_i}, n_{d_{ij}})$ of infinite capacity connecting the row and the node. There will be a final set of nodes n_{c_j} which will represent columns. Each node $n_{d_{ij}}$ will have an edge $(n_{d_{ij}}, n_{c_j})$ to its corresponding column of infinite capacity. Finally each column node will be connected to the sink with an edge (n_{c_j}, t) of capacity equal to the column sum of c_j.

Thus, we should have three sets of nodes, the row nodes, the element nodes, and the column nodes. The ith row node should have q outgoing edges to the q corresponding nodes which belong to the ith row. The jth column node should have p incoming edges to the p corresponding nodes which belong to the jth column. Each element d_{ij} should have a single incoming edge connecting it to its row node and a single outgoing edge connecting it to its column node.

Now, if we are already given a value for d_{ij}, then we delete the node $n_{d_{ij}}$ and the edges connected to it. Then, we subtract the value of d_{ij} from the capacity of (s, n_{r_i}) and (n_{c_j}, t). This is equivalent to subtracting d_{ij} from the row and column sums, leaving the rest of the elements in each row as unknown.

Now, the find if there is some set of cell values that produces the disclosed values, we simply run a max flow algorithm from s to t and check if the max flow is equal $\sum_i c(s, n_{r_i}) = \sum_j c(n_{c_j}, s)$. In other words, we check if the flow saturates all of the row and column capacities. By the integrality theorem, our flow should be an integer, and the flows into each $n_{d_{ij}}$ node will be a value which satisfies the row and column sums according to the disclosed values. This is simply because we must have $\sum_j c(n_{r_i}, d_{ij}) = r_i - D_i$ where D_i is the disclosed values in row i. A similar expression can be made for the column sums. Thus, we see that the flows at $n_{d_{ij}}$ will be satisfactory according to the disclosed values.

This construction takes $O(pq)$ time since there are $O(pq)$ nodes and also $O(pq)$ edges which must be constructed. □

Problem 3-b: Given a graph G and a max-flow f that places flow value f_e on edge e, give an efficient algorithm for determining whether there is a max-flow that places a different flow value on edge e.

Solution: We will examine the residual graph G' of G. Recall that in the residual graph, for every edge (v, w) in G, there exist two edges (v, w) and (w, v), which combine to provide the net capacity still available to be used. In the algorithm, we will first find $e = (u_0, u_f)$ in the original graph and identify the two corresponding edges (u_0, u_f) and (u_f, u_0) in the residual graph.

Since we know we already have a max-flow in our graph, we simply want to see if we can find another flow with a different value of $f(u_0, u_f) - f(u_f, u_0)$ than f_e (where $f(v, w)$ is the flow from v to w) which has the same flow at sink t. To do this, we can first remove edge (u, v) from the residual graph and search for a path from v to u in the augmenting graph using BFS (treating edges with 0 capacity as non-existent). If there does exist such a path, then there exists a loop in the residual graph, which will have a flow of zero.

Thus, for each edge in the path from v to u, we can decrease the flow on the edge, and increase the flow on the backward edge. This will not affect the flow at the sink t, because the path forms a closed loop. Moreover, we will still have a valid flow after the edges have been decreased, since the incoming and outgoing flow will both be decreased by 1. We can perform the same procedure the (v, u) edge and look for a path from u to v using BFS. Thus, we have created a flow, which still retains the value of the maximum flow, which is valid and which uses a value different that f_e on edge e.

If neither of these two paths exist, then we know that there is no flow which places a different flow value on edge e. This follows because if there is no path from u to v or from v to u in the residual graph, then there is no way to increase or decrease the flow value at e while keeping the entire graph a valid flow network with the stated capacities, since we must have $\sum_w f(w,u) = \sum_v f(u,v)$ for each node. Therefore, one cannot change f_e without breaking this property at some edge in the graph. The algorithm runs in $O(n+m) = O(m)$ since it runs BFS twice and performs updates on at most $2m$ edges. □

Problem 3-c: Combine the previous two parts to solve the stated problem.

Solution: Note that once we have the flow graph from part a, we can determine if there is an unprotected value if we use the algorithm from part b. We do this by examining some node $n_{d_{ij}}$ and checking the edge $(n_{d_{ij}}, n_{c_j})$ so see if there is a max flow using a different flow value on the edge. If there is no different flow value possible, then we know that that element is unprotected because there is only a single value that the node is confined to which will work in a valid flow for the row and column sums. Note that checking the edge between the column node is equivalent to checking the edge to the row node because there is a single edge into $n_{d_{ij}}$ and a single edge out, which means the flow is constrained by the flow of a single one of these edges.

Therefore, it is sufficient to move through all edges $(n_{d_{ij}}, n_{c_j})$ and perform the algorithm from part b to determine d_{ij} is an unprotected element. If it is unprotected, we know its value since it is constrained to be the flow on the edge $(n_{d_{ij}}, n_{c_j})$. This algorithm will detect all unprotected values and runs in $O(p^2q^2)$, since the algorithm from part b is run $O(pq)$ times. □

6.854 Advanced Algorithms

Problem Set 2

John Wang
Collaborators: Jason Hoch

Problem 4-a: Suppose all people are initially in a single room s, and that the building has a single exit t. Show how to use maximum flow to find a fastest way to get everyone out of the building.

Solution: We will create an auxiliary graph G' where each node in G' represents a room at a given time step. Here $v(t) \in G'$ will represent the node $v \in G$ at time step t. For each node in G, there will be tT corresponding nodes in the graph G, where T is the total number of timesteps.

To compute the edges and capacities of G', we will start at timestep $t = 0$. For each edge (v, w) in G, we will create an edge from $v(0)$ to $w(1)$ in graph G' with capacity c. We will also create an edge from $v(0)$ to $v(1)$ for each node v so that people can stay in the same room. We will then increment the timestep to $t = 1$, and create edges from $v(1)$ to $w(2)$ for each path (v, w) in G, and edges from $v(1)$ to $v(2)$ for each node $v \in G$. This will continue until we reach timestep T.

We now have a graph detailing all the possible paths through the rooms in time T. To check if it is possible in T time, we look at the maximum flow from the original source vertex $s(0)$ and the final exit vertex $t(T)$. If the flows are the same, then it is possible to get everyone out of the building in T time.

To figure out the shortest possible time, try $T = 2^k$ for $k = 1, 2, \ldots$ until everyone can get out of the building. Then binary search on $T \in [2^{k-1}, 2^k]$ until we find the lowest possible T. This requires $O(\log T_f)$ tries, where T_f is the fastest possible time. □

Problem 4-b: Show that the same technique can be used when people are initially in multiple locations and there are multiple exits.

Solution: We will use the exact same algorithm. However, we will start by augmenting the original graph G with two super nodes s^* and t^*. Here s^* will have edges leading to each of the starting locations l. The edges to the starting locations (s^*, l) will have capacity equal to the number of people starting at l. The super node for the sink will be t^* and each exit location q will have an edge (q, t^*) with unlimited capacity.

Running the algorithm will now give the fastest way to get out of the building for multiple sinks and exits by using s^* as the starting vertex and t^* as the exit vertex in the algorithm provided in part a. This works because each starting location will get its correct number of people, allowing the flow algorithm to proceed normally. □

Problem 4-c: Generalize the approach to where different corridors have different (integral) transit times.

Solution: In this algorithm, transit times will affect which states in time t that we can connect to. We again start at timestep $t = 0$ and proceed in the following manner. At timestep t, for each edge in $(v, w) \in G$, we will create an edge in the auxiliary graph G' from $v(t)$ to $w(t + j)$ of capacity c, where j is the transit time of the corridor, if there exists a corridor from v to w. We will also create an edge from $v(t)$ to $v(t + 1)$ with infinite capacity to allow people to stay in a room.

The rest of the algorithm remains the same, and changing the transit times means that now, it takes j time steps before people can arrive at room w. Notice that in G', the corridor from v to w will be blocked off for j timesteps as people are attempting to pass through, which allows us to deal with different transit times for each corridor. □

6.854 Advanced Algorithms

Problem Set 2

John Wang
Collaborators: Jason Hoch

Problem 5-a: Argue that the running time of Dial's algorithm is $O(m + D)$ where D is the maximum distance, and that the algorithm still works if some edges have length 0.

Solution: First we will initialize an array storing the keys from 1 to nC. We know that this can be done in $O(1)$ time. Next, we will perform Dijkstra's algorithm by successively finding the minimum element in the queue, and expanding its neighbors, and decreasing the key by storing the reduced edge lengths $l^d_{vw} = l_{vw} + d_v - d_w$. The total reduced edge length along a path from the source to some node t is $\sum_{v,w} l^d_{vw}$ where v and w are intermediate nodes on the path. Since this sum telescopes, we see that the total reduced edge length is simply d_t. This implies that removing the lowest reduced edge length will remove the current shortest path to some node and will not affect the correctness of Dijkstra's.

This means that we can perform n inserts, n delete-mins, and m decrease-keys. We know the insert and decrease key operations cost $O(1)$ each. The delete-min operations cost $O(k)$ where k is the distance between the last minimum and the next minimum (since we have to crawl our way up k buckets until we find the minimum). This means that the total amount of work performed is just $\sum_{i=1}^{n} k_i$ where k_i is the distance between the minimums at the $i-1$st and ith steps. This comes out to the maximum distance, which is D. Thus total work is just $O(m + D)$.

The algorithm clearly still works for edge length of zero because we can initialize a bucket for edges of length 0, which will then always be the minimum in the set, so they will be picked first. □

Problem 5-b: Show that the reduced edge lengths defined above are all nonnegative integers.

Solution: We need to show that $l^d_{vw} = l_{vw} + d_v - d_w$ are all nonnegative. We know that d_v, d_w and l_{vw} are all nonnegative. Therefore, all we need to do is to show that $l_{vw} \geq d_v - d_w$. Let $\{s, v_1, v_2, \ldots, v\}$ be the shortest path from s to v with distance d_v. We know that $d_v + l_{vw} \leq d_w$ since the shortest path to w from s puts a lower bound on the distance of any path to w from s. This expression rearranges to $l_{vw} \geq d_v - d_w$ which is what we wanted to show. □

Problem 5-c: Show that the shortest paths under the reduced length function are the same as those under the original length function. What is the length of shortest paths under l^d?

Solution: Suppose by contradiction that under the reduced length function, there is some path P' from s to u which is shorter than P, where P is the shortest path under the original length function. Let $P = \langle s = v_1, v_2, v_3, \ldots, u = v_n \rangle$ of length n and $P' = \langle s = v'_1, v'_2, v'_3, \ldots, u = v'_{n'} \rangle$ of length n'. If P' is shorter than P under the reduced length function, then the following is true:

$$\sum_{i=1}^{n'} l^d_{v'_{i-1}v'_i} < \sum_{i=1}^{n} l^d_{v_{i-1}v_i} \tag{7}$$

$$\sum_{i=1}^{n'} l_{v'_{i-1}v'_i} - d_{v'_i} + d_{v'_{i-1}} < \sum_{i=1}^{n} l_{v_{i-1}v_i} - d_{v_i} + d_{v_{i-1}} \tag{8}$$

$$-d_u + \sum_{i=1}^{n'} l_{v'_{i-1}v'_i} < -d_u + \sum_{i=1}^{n} l_{v_{i-1}v_i} \tag{9}$$

$$\sum_{i=1}^{n'} l_{v'_{i-1}v'_i} < \sum_{i=1}^{n} l_{v_{i-1}v_i} \tag{10}$$

This occurs because $d_{v_i} - d_{v_{i-1}}$ leads to a telescoping sum. However, this means that path P' has a shorter length than P under the original length function, which is a contradiction of the fact that P was the shortest path under the original length function. Thus, the shortest path under the reduced length function is the same as the one under the original length function.

The length of the shortest paths under l^d is therefore $-d_u + \sum_{i=1}^{n} l_{v_{i-1}v_i}$ where d_u is the distance of the shortest path under the original length function. Thus, the shortest path under the reduced length function is just 0. □

Problem 5-d: Devise a scaling algorithm for shortest paths. Use the reduced edge lengths and Dial's bucketing algorithm to keep the task of shifting in a new bit sample.

Solution: Notice that in our proof that the reduced length function still represented a valid metric for the edge lengths, we only needed to show that $d_w \leq d_v + l_{vw}$ so that all edge lengths were non-negative. If this is the case, then part c follows with the same proof as before. Therefore, we can define d_u^i as $d_u \gg i$, or d_u right shifted i bits, and l_{vw}^i as $l_{vw} \gg i$ (again l_{vw} right shifted i bits).

Since $d_w \leq d_v + l_{vw}$ in our original edge lengths and distances, we can see that right shifting the bits does not affect the fact that $d_w^i \leq d_v^i + l_{vw}^i$ which means that this is a valid metric for the edge lengths. Therefore, we can use the following algorithmi.

First, set $l_{vw}^{\log C+1} = 0$ and $d_v^{\log C+1} = 0$ for all v and w which are nodes in the graph. Next, set $l_{vw}^{\log C}$ as $l_{vw} \gg \log C$ and compute the reduced lengths using $l_{vw}^d = l_{vw} + d_v - d_w$ where d is defined as the previous step's distance function. We then compute new distances based on these lengths using Dial's algorithm. We continue until we have made $\log C$ steps and have reached l_{vw}^1 and d_v^1. By this time, the correct distances will have been computed, and we know that the shortests paths will have been found. This is because in each step, the current graph G' based on the reduced length function will have the same shortest paths as those in the original graph G, as shown in part c of the problem.

Thus, the algorithm performs $\log C$ steps of Dial's algorithm. Notice that at each step, the distance of each of the shortest paths is bounded by 0 from our analysis in part c. This means that the maximum distance is 0 in the reduced graph and therefore n in the original graph, so that the total work done during each step of Dial's algorithm is $O(m+n) = O(m)$. The total time of the algorithm is therefore $O(m \log C)$. □

Problem 5-e: Is base 2 scaling the best possible or can you achieve (slightly) better running time by scaling with a different base?

Solution: If we use base k, then there will be $\log_k C$ iterations of Dial's algorithm. During each iteration, the maximum distance is bounded by nk so that the total time for Dial's is given by $m + nk$ at each step. Solving for the minimum value of k, we find that we can set $b = m/n$ and we can achieve a running time of $O(m \log_{m/n} C)$ for the entire algorithm, which is better than the base $k = 2$ case. □