Exercises in Algorithms No 4

Alexandros Kyriakakis (el12163)

January 2019

1 Friend Suggestion

The solution of this problem is easy and tricky at the same time. So, let's begin. We construct a directed weighted graph $G(V, E, w)$ where V is every person in the social media platform that we study, E are all connection between the people who's $t(i, j) \neq 0$ and, now the tricky part, weight (w) for each edge between vertices i, j is $log(1/t(i, j))$, with $\mathcal{O}(V + E)$ time complexity. Then we run Dijkstra's algorithm for the shortest path starting from vertex i but instead of adding the weights of the edges when we update the key of the distances from i on the queue, we add the logarithmic weights and we store them on the same way. For example, if s starting vertex is connected with i and i with j on this direction. Normal Dijkstra's algorithm would update $w_{(s,i)}+w_{(i,j)}$ the key of j on the queue and we replace that with $\log{(w_{(s,i)})} + \log{(w_{(i,j)})}$. This action can be proved with the following property: $\log{(\Pi_{q=0}^{l-1} t(q, q-1))} = \Sigma_{q=0}^{l-1} \log 1/t(q, q-1)$. Also we store at every vertex the distance l from s based on the path we chose to follow from s to j and we update it in the same way as key. We stop Dijkstra's algorithm if all vertices have distance $l > k$. Then with a single passage from vertices we check if $\Pi_{q=0}^{l-1} t(q, q-1) \geq b_l$ and we return the vertices (people) which satisfy the condition. This trick can be done because the condition of comparison is based on logarithmic multiplication so $a * b \geq c \leftrightarrow (1/a) * (1/b) \leq 1/c \leftrightarrow \log{((1/a) * (1/b))} \leq \log{(1/c)} \leftrightarrow \log{(1/a)} + \log{(1/b)} \leq \log{(1/c)}$. Based on that we search for the shortest path using modified Dijkstra. The complexity of the algorithm is Dijkstra's because we just changed the addition with multiply on the algorithm and we stopped every search at constant distance k so we did k repeats. So the total time complexity of the algorithm is $\mathcal{O}(E + V * k)$.

2 Dijkstra's Algorithm Modification

2.1 Small Weights C

Starting with a look at the complexity of the algorithm that we want to develop, the obvious is that we don't use neither a queue nor a heap for the construction of Dijkstra's algorithm. So we recognize that it's going to be replaced with a something using C. So this way led us to the following modification to Dijkstra's algorithm. So instead of using the queue or a heap we construct an array of size $n * C$. Each element of the array is the starting point (head) of a linked list. Then every time we want to store the distance of a vertex from starting point we go at the corresponding index of the array and we save it there. So in a random moment if vertex u is stored in index fifteen then the current distance of u from starting vertex is fifteen. So replacing the queue with this array we have $\mathcal{O}(C)$ time complexity for storing and also updating the key. So total complexity of the algorithm is $\mathcal{O}(V * C + E)$.

2.2 Large Weights 2^C

This time with the knowledge from the first problem we know once more that we are going to replace the queue, according to the complexity. So we construct a binary heap using the weights of the edges and we do the same as the previous problem we store at each node of the heap a linked list with the vertices that led with the current weighted edge, and the distance of each vertex from the starting vertex. So we need $\mathcal{O}(log2^C) = \mathcal{O}(C)$ time to insert a vertex, delete a vertex or update a key. So same as before we need $\mathcal{O}((V + E) * C)$ total time complexity.

3 Shortest Paths with two conditions

3.1 Zeros and Aces

The idea for this problem began from the Floyd-Warshall algorithm. We construct a new Graph $G'(V, E')$ with only the edges with zero cost. Then we construct another $G''(V, E'')$ without the edges with cost one and we connect every vertex from G'' which would have an one-edge with the corresponding vertex from G'. We continue likewise with G''' and G'' until $G^{(k)}$ Then we merge the starting edges s end we start Dijkstra's Algorithm to the final graph. We return the minimum distance between $\{s - u, s - u', ..., s - u^{(k)}\}$. The construction of the graph has $\mathcal{O}(E * k)$ time complexity and running Dijkstra has $\mathcal{O}(V * E + 2V^2 * \log n) = \mathcal{O}(n^3)$.

3.2 Informal Knapsack

At this problem we just do the same as above but more general. We construct the same large graph with C layers but for every edge with cost c_i from the starting graph we don't connect it with the graph right above but with the layer that's c_i layers above. We need $\mathcal{O}((C * E)$ time complexity for the construction. After the construction we run Dijkstra on the final graph and we return once more the minimum distance between $\{s-u, s-u', ..., s-u^{(C)}\}$. Total complexity of the problem is $\mathcal{O}(C * E + C * V * \log C * V)$.

4 Commercial Adds at the Internet

At first we construct a graph $G(V, E, f)$ (f: flow) with n vertices for each person in the network, n vertices for each subset of the main categories that n people belong, m vertices for each subset of the main categories that each company aim for and m vertices for each company. Also we add a starting vertex s and an ending vertex f. Then we add the edges. At first we connect each n people with starting vertex s without blocking flow quantity $flow = \infty$. Then we connect each person with each n subset that we created for each specific person from the main categories, with blocking flow quantity equal to 1 at each edge. After that we connect the n and m subsets based on the condition, they have at least one common category, with no blocking flow value $flow = \infty$. Then we connect each subset, companies aim, with each company with capacity value $c_i,\ i\epsilon\{1, 2, ..., m\}$. At last we connect all the companies with the finishing point s. Now we have constructed a bipartite graph which is proved by construction. So we run Dinitz's algorithm for maximum flow on this graph which has $\mathcal{O}(\sqrt{V} * E)$ time complexity for bipartite graphs where $V = 2*n+2*m$ and $E = n*m$. After the end of Dinitz's algorithm we return the edges that the algorithm used to define the maximum flow between the two subsets, which represents an "$1 - 1$" matching between subsets' vertices. So we need $\mathcal{O}(n + m + n * m)$ time to construct the graph and $\mathcal{O}(\sqrt{n + m} * n * m)$ time to run Dinitz's algorithm. So the total time complexity is $\mathcal{O}(\sqrt{n + m} * n * m)$.

5 Reductions

We can easily prove that all the problems belong in NP class because all are polynomial balanced and responsive so we assume that's a proved fact at all the solutions. Also based on the previous fact we don't have to prove the reduction for the direction to the NP complete problem that we use at each case. We are going use proof by contradiction.

5.1 3-Partitioning

We assume that we have an instance for the well known problem 2-Partitioning $A = \{w_1, ..., w_n\}$ and we add one more element to the set A which is $w_{n+1} = \frac{w_1+...+w_n}{2}$. So we constructed $A' = \{w_1, ..., w_n, \frac{w_1+...+w_n}{2}\}$ and we use it as input to the problem "3-partitioning". We assume that this problem gives a solution for this instance in polynomial time. If this happens, then we have a solution for the 2-Partitioning, so we led to contradiction because we guessed that in polynomial time we solve a NP complete problem which is false.

5.2 Approximate Subset Sum

We assume that we have an instance for the Subset Sum $A = \{w_1, ..., w_n\}$ and we multiply all the elements of A with $x+1$ so now we have $A' = \{w_1*(x+1), ..., w_n*$

$(x+1)\}$. Then we use set A' as input to the current problem. We assume that this problem gives as a solution in polynomial time. So in polynomial time we know that $\exists S \subseteq A' \ : \ (x+1) * B - x \leq (x+1) * w(S) \leq (x+1) * B$. We notice that all of the three products are integer multiples of $x+1$ and more specifically at the centre of the inequality there is only one solution. So if we solve this problem in polynomial time we have solved the Subset Sum problem for the starting instance for $k = B$ which led us to $P = NP$.

5.3 Approximate Hamilton Circle

We assume that we have an instance for the Hamilton Circle problem $G(V,E)$. We double the number of vertices and at every vertex we hang one extra vertex with one edge. Then we use this graph $G'(2V, E+V)$ as input for the current problem. We assume that we have a solution to this problem in polynomial time. This means that the solution expend each second pass from every vertex for tracking the hanged vertices. So, this means that we have a solution for the Hamilton circle in polynomial time which is not proved.

5.4 4-CNF with Conditions

We assume that we have an instance for 3-CNF $\varphi = \bigwedge_{j=1}^{m}(l_{j1} \vee l_{j2} \vee l_{j3})$ and we transform it to $\varphi' = \bigwedge_{j=1}^{m}(l_{j1} \vee l_{j2} \vee l_{j3} \vee z) \wedge (l_{j1} \vee l_{j2} \vee l_{j3} \vee \neg z)$. We use φ' as input for the current problem. We assume that we have a solution to this problem at polynomial time. This means that with this solution we have a solution in 3-CNF, because each set of four literals is true so if $l_{j1} \vee l_{j2} \vee l_{j3} = false$ and $z = true$ at the next set $\neg z = false$ which leads us that $l_{j1} \vee l_{j2} \vee l_{j3} = true$. So in polynomial time we have solution to 3-CNF which is not proved.

5.5 Independent Set Selection

We assume that we have an instance $S_G = \{v_1, ..., v_k\}$ from $G(V,E)$ for the problem K Independent Set. For every vertex from G we create a set S_i. Then for every edge from G we add the same integer at both sets S_i that represent the vertices which the current edge connects. We use this as input to the current problem. We assume that we have a solution at polynomial time. If this exists, means that we've found k independent sets of numbers which leads us from construction that we found k vertices without common edge. So we solve the K Independent Set at polynomial time which is not proved.

5.6 Shortest Path with conditions

We assume that we have an instance for decision knapsack problem with n objects with cost c_i and weight w_i. For every object from the above instance we create a vertex in a full directed graph. Every edge that goes away from each vertex has weight w_i and cost c_i. We use this as input to the current problem. We assume that we take a solution in polynomial time. If this exists, means that we have a solution for knapsack problem in polynomial time because a path $s-t$ pass only once from every vertex and edge. So we showed that knapsack can be solved with polynomial time which is wrong.