Assignment-3

Design and Analysis of Algorithms

Trishan Mondal

Disclaimer. Consider the following set of students

$$\mathcal{P} = \left\{ \begin{matrix} \text{Aaratrick Basu, Deepta Basak, Priyatosh Jana,} \\ \text{Shubhrojyoti Dhara, Soumya Dasgupta, Trishan Mondal} \end{matrix} \right\}$$

Discussion of solutions to the assignment problems are limited to the people of set $\mathcal{P}$ only. Most of the problems in this assignment has general solution. If any other person have same solution as mine is not my fault.

---

§ Problem 10

Problem. Suppose $G = (V, E)$ is an undirected unweighted graph with n vertices and m edges. Suppose $s, t \in V$ are vertices of G whose distance in G is strictly greater than $n/2$. Show that there is a vertex (other than s and t) whose deletion disconnects s from t. Describe an algorithm (assume that adjacency lists are available) running in time $O(m + n)$.

Solution. Consider a BFS tree T of G with s as the root. We know in a BFS tree, a vertex v lies on a level which is equal to the length of shortest path (distance) from s to v. It is given that distance between s and t is at least $\lfloor \frac{n}{2} \rfloor + 1$. In the BFS tree where, s is root, t occurs at the level $\geq \lfloor \frac{n}{2} \rfloor + 1$. So, there is at-most $n - 2$ nodes (vertices) between the level 1 and $\lfloor \frac{n}{2} \rfloor$. It must happen that, one level $1 \leq \ell \leq \lfloor \frac{n}{2} \rfloor$ exists so that, it has only one node. Otherwise, if each level has ≥ 2 nodes the total number of nodes must be $\geq n$ but it is not possible.

Once we got the singleton node at some ℓ-th level of BFS tree starting at s, where $1 \leq \ell \leq \lfloor \frac{n}{2} \rfloor$. Call one of this node is w. I **claim** that, deletion of this vertex will disconnect s from t. If deletion of w does not disconnects s from t, there must exist an edge $\{w, p\}$, with `dist(s,p)>dist(s,w)` and $\{w, q\}$ with `dist(s,w)>dist(s,q)`. Then the distance between p, q will be less than the distance between them via the BFS tree. It is a contradiction. So there is a vertex by removing which we can disconnect s from t.

Algorithm

`Input:` $G = (V, E)$ undirected and unweighted graph with n vertices and m edges and s, t such that distance b/w s and t is strictly greater than $\frac{n}{2}$.

`Output:` A vertex v, removing which s and t will disconnects.

- **Step 1:** Do a BFS on G starting at s.
- **Step 2:** For each $1 \leq i \leq \lfloor \frac{n}{2} \rfloor$, let $L[i]$ be the list of vertex at level i.
- **Step 3:** Check $L[i]$ which has only one node (vertex), for $1 \leq i \leq \lfloor \frac{n}{2} \rfloor$. `Return` that vertex.

Correctness. By the first paragraph of previous part we can see there exist such vertex w. So the algorithm will **terminate**. Let, w be the vertex returned by the algorithm. We will show that deletion of w will disconnect s and t. If not, Then the distance between p, q will be less than the distance between them via the BFS tree. It is a contradiction. It proves the **correctness** of the algorithm.

Time complexity. For doing BFS tree we need $\sim O(m+n)$ time. For listing the vertices in **step 2** we need $\sim O(n)$ time and at the last step we need $\sim O(n)$ comparison to get the required result. So the time complexity of algorithm is $O(m+n)$. ■

§ Problem 11

Problem. Suppose $G = (V, E)$ is a connected undirected graph. Suppose DFS starting at a vertex v and BFS starting at the same vertex v produce the same tree. Then, show that G is a tree.

Solution. Let's denote the trees produced by BFS and DFS as T. Assume that G is not a tree. This implies the existence of an edge $e = \{p, q\} \in G$ such that $\{p, q\} \notin T$. In this scenario, within the DFS tree, either vertex p or q must serve as an ancestor of the other (resulting in a back edge). This arises from the fact that if, for instance, q is first discovered by DFS, our traversal must encounter p while still exploring q, or it would utilize the edge from p to q. Simultaneously, in the BFS tree, the levels of p and q can differ by only one.

Since both BFS and DFS trees are identical, it logically follows that one of the vertices, either p or q, must function as the ancestor of the other, with just a one-level difference. Consequently, the edge connecting them must be present in T. This leads us to a contradiction, which means G is a tree. ■

§ Problem 12

Problem. Suppose G is a directed graph with n vertices and m edges. Describe an algorithm (assuming adjacency lists are available) running in time $O(m+n)$, if G has a vertex v from which every other vertex is reachable.

Solution. Let's look at the meta-graph of the given directed graph $G = (V, \vec{E})$, which is made of treating the SCC's as a vertex. We know that this meta-graph G' is *acyclic*. In a DAG (directed acyclic graph) there is always a source and a sink. Let, S be the source of G'. If there is a vertex v from where we can travel every other vertex, define it by *good vertex*, it must lie in S of the meta-graph G'. If it lie in any other component S' then we can't travel to the source, as there is no edge coming in at the source vertex.

Observation 1: If there are more than one source in the meta-graph then it is not possible to get a 'good vertex'. As we have seen previously the good vertex must lie in a source of G' but then we can't get back to other source as there is no edge coming inwards to source.

Observation 2: If there is only one source component S in the meta-graph G', every $v \in S$ is a good vertex. Let, u be a vertex in other SSC. Call this component T_1. Now define a sequence of component's (vertices in G') $\{T_i\}$, such that, there is an edge from T_{i+1} to T_i. For example T_2 is the component such that there is an edge from T_2 and T_1. Since the directed graph is finite this sequence will stop at some stage. If the sequence ends at T_n, then T_n must be the source S (by uniqueness of source) as there is no inwards edge to S. G' is the meta-graph and it is DAG, so $T_1 \leftarrow T_2 \leftarrow \cdots S$ is a path where no component (or vertex of G') is visited again. This will give us a path to reach u from v. Thus v is a 'good vertex'.

Algorithm

`Input:` A directed graph $G = (V, \vec{E})$

`Output:` `NULL` if there is no such 'good vertex' v. `Return` v if v is a 'good vertex'.

- **Step 1:** Choose any $v \in V$ and do a `dfs(G,v)`. Take a vertex u with maximum post visit number in this `dfs`.

• **Step 2:** Again do the dfs with the vertex u, i.e. `dfs(G,u)`. If the maximum post visit number of this new dfs is greater than post-visit number of u. Then return `NULL`. Otherwise, return `v`.

Correctness. At-first we perform `dfs` on G with respect to some vertex v. Then the vertex with highest post-visit u, will lie in the source S of the meta graph G' (this was proved in class while doing Kosaraju's algorithm). Now perform dfs again with respect to u. If the maximum of post visit number in this dfs not equal to the post visit number of u, then there is another source component in this directed graph G. By Observation 1 we can't have a good vertex. That's why this algorithm will return `NULL`. Else if the post visit number of u is equal to the maximum post-visit number, there is only one source component. By Observation 2, v is good vertex. Since the graph is finite, dfs will terminate and hence our algorithm will terminate. Thus our algorithm is **correct**.

Time complexity. We are running `dfs` two times, so we need $\sim O(m+n)$ time to run the algorithm. ■

§ Problem 13

Problem. In the 2SAT problem, you are given a set of *clauses*, where each clause is the disjunction (`OR`) of two literals (a literal is a Boolean variable or the negation of a Boolean variable). You are looking for a way to assign a value `true` or `false` to each of the variables so that *all* clauses are satisfied - that is, there is at least one true literal in each clause. For example, here's an instance of 2SAT:

$$(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (x_1 \vee x_2) \wedge (\bar{x}_3 \vee x_4) \wedge (\bar{x}_1 \vee x_4).$$

This instance has a satisfying assignment: set x_1, x_2, x_3 and x_4 to `true`, `false`, `false`, and `true`, respectively.

(a) Are there other satisfying truth assignments of this 2SAT formula? If so, find them all.

(b) Give an instance of 2SAT with four variables, and with no satisfying assignment.

The purpose of this problem is to lead you to a way of solving 2SAT efficiently by reducing it to the problem of finding the strongly connected components of a directed graph. Given an instance I of 2SAT with n variables and m clauses, construct a directed graph $G_I = (V, E)$ as follows.

- G_I has $2n$ nodes, one for each variable and its negation.
- G_I has $2m$ edges: for each clause $(\alpha \vee \beta)$ of I (where α, β are literals), G_I has an edge from the negation of α to β, and one from the negation of β to α.

Note that the clause $(\alpha \vee \beta)$ is equivalent to either of the implications $\bar{\alpha} \Rightarrow \beta$ or $\bar{\beta} \Rightarrow \alpha$. In this sence, G_I records all implications in I.

(c) Carry out this construction for the instance of 2SAT given above, and for the instance you constructed in (b).

(d) Show that if G_I has a strongly connected component containing both x and $\bar{x}$ for some variable x, then I has no satisfying assignment.

(e) Now show the converse of (d): namely, that if none of G_I's strongly connected components contain both a literal and its negation, then the instance I must be satisfiable.

(f) Conclude that there is a linear-time algorithm for solving 2SAT.

Solution. **!**The solution to this problem is more or less everyone must have done in same way. Please don't cut marks for plagiarism**!**

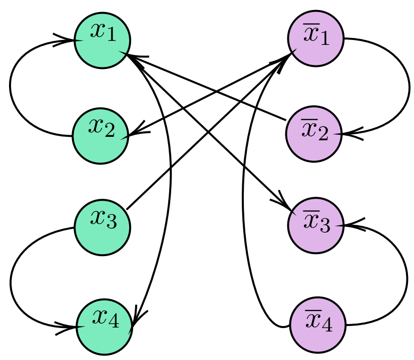
(a) It is given that, $(x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (x_1 \vee x_2) \wedge (\bar{x}_3 \vee x_4) \wedge (\bar{x}_1 \vee x_4)$ is `true`. So, each $(x_1 \vee \bar{x}_2)$, $(\bar{x}_1 \vee \bar{x}_3)$, $(x_1 \vee x_2)$, $(\bar{x}_3 \vee x_4)$, $(\bar{x}_1 \vee x_4)$ will be `ture`. Since, $(x_1 \vee \bar{x}_2)$ and $(x_1 \vee x_2)$ are `true`, we must have $x_1 =$ `ture`. Now, $(\bar{x}_1 \vee \bar{x}_3) =$ `true` will tell us that, $x_3 =$ `false`. From the fact, $(\bar{x}_1 \vee x_4), (\bar{x}_3 \vee x_4) =$ `ture` we will get, $x_4 =$ `true`. We only have freedom for x_2 and x_1, x_3, x_4 are derived as above.

(b) Consider the following 2SAT, which don't have any satisfying assignment as it will always give `flase`,

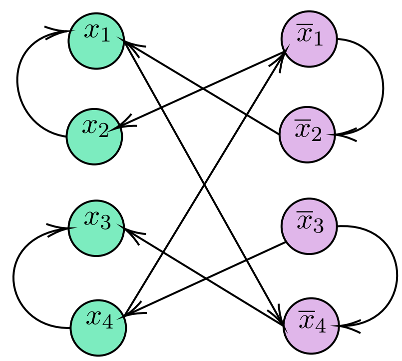
$$(x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (x_3 \vee x_4) \wedge (\bar{x}_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_4)$$

If the above was `true` in some case, then $x_1 =$ `true` and $x_4 =$ `true` but then $\bar{x}_1 \vee \bar{x}_2$ is `flase`.

(c) Here the following graphs $G_{I'}$ constructed for the instance $I' = (x_1 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee \bar{x}_3) \wedge (x_1 \vee x_2) \wedge (\bar{x}_3 \vee x_4) \wedge (\bar{x}_1 \vee x_4)$ and G_I is constructed for $I = (x_1 \vee x_2) \wedge (x_1 \vee \bar{x}_2) \wedge (x_3 \vee x_4) \wedge (\bar{x}_3 \vee x_4) \wedge (\bar{x}_1 \vee \bar{x}_4)$ which is the thing we constructed in part (b).



$G_{I'}$



G_I

(d) We begin by observing that an edge (p, q) exists in G_I if and only if the clause $(\overline{p} \vee q)$ is present in I. Now, let's consider the presence of a strongly connected component in G_I that includes both x and $\overline{x}$. According to its definition, this implies the existence of directed paths $(x, x_1, \ldots, x_n, \overline{x})$ and $(\overline{x}, y_1, \ldots, y_m, x)$ in G_I. Consequently, we deduce that I must contain the following sub-instance:

$$(\overline{x} \vee x_1) \wedge (\overline{x}_1 \vee x_2) \cdots (\overline{x}_n \vee \bar{x}) \wedge (\overline{x} \vee y_1) \wedge (\overline{y}_1 \vee y_2) \cdots (\overline{y}_m \vee x)$$

Let's consider the case when $x =$ `true`. The first clause above implies that $x_1 =$ `true`, the second one $x_2 =$ `true`, and so forth, until $x_n =$ `true`. This in turn leads to the conclusion that $\overline{x} =$ `true`, which contradicts our initial assumption. Now, let's consider the case when $x =$ `false`. In this scenario, $\overline{x} =$ `true`, and consequently, $y_1 =$ `true`. This then implies $y_2 =$ `true`, and so on, until $y_m =$ `true`, eventually leading to $x =$ `true`, which once again contradicts our initial assumption.

Therefore, as this sub-instance of I does not have any satisfying assignments in either case, we can conclude that I itself does not have any satisfying assignments. This completes the proof.

(e) Let there are no strongly connected components in G_I that contain both a variable and its negation. We assert that if we iteratively select a sink strongly connected component, set all the literals represented as vertices to `true`, and then remove them, we will eventually obtain a satisfying assignment.

We will prove this assertion through induction on the number n of literals involved in I. The statement holds true when $n = 1$ because G_I must contain two isolated vertices representing the literal and its negation. Consequently, I is satisfied by the assignments $\overline{x} =$ `true` and $x =$ `true`, and these are the only assignments produced by the described procedure.

Now, assume the assertion holds for some n, and let G_I be the graph corresponding to an instance I involving $n+1$ literals. First, we identify any sink strongly connected component S. Since S is a sink component, there are no edges (u, v) in the graph where $u \in S$ and $v \notin S$. Also, the clause $\overline{u} \vee v$ is not present in I for $u \in S$ and $v \notin S$. In other words, for $u \in S$, the clause $\overline{u} \vee v$ exists in I only if $v \in S$. Similarly, an edge (v, u) in G_I corresponds to the clause $\overline{v} \vee u$. Therefore, by setting all literals in S to `true`, any clause involving such literals or their negations must evaluate to true. Consequently, whether I has a satisfying assignment depends totally on the literals not in S. Therefore, we examine G_I after removing all vertices (including negations) corresponding to literals occurring in S. This results in a graph representing an instance with at most n literals and satisfies the assumption that no strongly connected component contains both a literal and its negation. Hence, by induction, any instance I where G_I satisfies this assumption must have a satisfying assignment.

(f) We will take the idea of Kosaraju's algorithm and we will find all the SCC and assignment of corresponding literals. The algorithm described as follows,

- **Step 1:** At first create directed graph G_I, from the given instance I. Now consider the reveresd graph G_I^R. Run DFS on it and record the post-visit numbers and get the SCC of G_I by running DFS on the vertices in increasing order of post number.
- **Step 2:** If any strongly connected component of G_I has both literal and it's negation (We can do this by checking their SCC number) then by part (d) and part (e), we can say I do not have any satisfying assignment.
- **Step 3:** For the component of I containing sink, set all literals on the same SCC of source as `true`. Then delete all the vertices in this SCC as well as the negation of the literals in this SCC.
- **Step 4:** Continue doing the above steps till there is no SCC left and then return assignment of each literals.

• Correctness. The correctness of this algorithm follows from part(d) and part(e) of the question.

• Time complexity. For creating a graph $G_I = (V, E)$ we need $\sim 2n + 2m$ time, then we are doing Kosaraju's algorithm to get SCC's in some topological order. This will take $|V| + |E| \sim O(n+m)$ time. So **step 1** takes $\sim O(n+m)$ time. In **step 2** we are looking at the SCC number of vertex v and $\bar{v}$. If they are same, it will take $\sim O(n)$ time. **Step 3** does everything in constant time. Finally **step 4** executes **step 3**, $\#\{\text{SCC in } G_I\}$ times, which can be atmost the number of vertices, i.e. $2n$. So total time complexity is $\sim O(m+n)$. ■

§ Problem 14

Problem. *Generalized shortest-paths problem.* In Internet routing, there are delays on lines but also, more significantly, delays at routers. This motivates a generalized shortest-paths problem. Suppose that in addition to having edge lengths $\{l_e : e \in E\}$, a graph also has vertex costs $\{c_v : v \in V\}$. Now define the cost of a path to be the sum of its edge lengths, plus the costs of all vertices on the path (including the endpoints). Give an efficient algorithm for the following problem.

- **Input:** A directed graph $G = (V, E)$; positive edge lengths l_e and positive vertex costs c_v; a starting vertex $s \in V$.
- **Output:** An array `cost[·]` such that for every vertex u, `cost[u]` is the least cost of any path from s to u (i.e. the cost of the cheapest path), under the definition above.

Notice that `cost[s]` $= c_s$.

Solution. This problem is similar to the problem of solving for the shortest path from a source, where the weights of edges are positive. We will try to retrace the Dijkstra's algorithm but in place of length-weights we will consider vertex-weights. The algorithm is given as following (in terms of pseudocode),

---

```
def gen_short_path(G,l,c,s)
# G = (V,E) is the graph given in terms of adjacency list
# ℓ is the set of edge lengths
# c is the set of vertex costs
# s is the starting vertex
for all v ∈ V
   cost[v] = ∞
   prev(v) = nil
cost[s] = c_s

H = makequeue (V)
 while H is not empty:
    u = deletemin(H)
    for all edges (u,v) ∈ E:
      if cost[v] > cost[u] + ℓ_{u,v} + c_v :
            cost[v] = cost[u] + ℓ_{u,v} + c_v
            prev(v) = u
            decreasekey(H,v)
```

Here we have just modified the Dijkstra's algorithm done in class and also written in **DPV** page.115.

Correctness and Time complexity. The Correctness of the algorithm follows from the correctness of the Dijkstra's algorithm, which was done in class. In this case time complexity is, $|V| \times$ `deletemin` $+ (|V| + |E|) \times$ `decreasekey`. In this case we are using Binary heap for priority queue implementatiom. So, `deletemin` $= \log|V| =$ `decreasekey`. And hence time complexity is $\sim O((m+n)\log n)$. This is same as the complexity of Dijkstra's algorithm. ■