Algorithm Homework 02

Qiu Yihang

March 2022

1 Problem 01

1.1 Assessment of r

Proof. Considering

$$
\begin{aligned}
r^\star < r \iff & r^\star = \max_C r(C) = \max_C \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}} < r \\
\iff & \forall C,\ \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}} < r \\
\iff & \forall C,\ \sum_{(u,v)\in C} p_v < r \sum_{(u,v)\in C} c_{uv} = \sum_{(u,v)\in C} r c_{uv} \\
\iff & \forall C,\ \sum_{(u,v)\in C} r c_{uv} - p_v > 0, \\
r^\star > r \iff & \exists\ C,\ \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}} > r \\
\iff & \exists\ C,\ \sum_{(u,v)\in C} p_v > r \sum_{(u,v)\in C} c_{uv} = \sum_{(u,v)\in C} r c_{uv} \\
\iff & \exists\ C,\ \sum_{(u,v)\in C} r c_{uv} - p_v < 0,
\end{aligned}
$$

we can derive a new graph $G_r' = (V', E', \texttt{weight})$ from the original graph $G = (V, E)$, where $V' = V, E' = E$, and the weight of edges is assigned as follows.

$$\forall (u,v) \in E,\ \texttt{weight}\left((u,v)\right) = r c_{uv} - p_v.$$

By the analyses above, when $r^\star < r$, we know all cycles on G_r' is of positive weight. When $r^\star > r$, exists a negative cycle C in graph G_r'.

Thus, we just need to apply **Bellman-Ford** Algorithm on G_r' to see whether there exists a negative cycle or not. If exists a negative cycle, $r^\star > r$. If not, then $r^\star < r$. ∎

1.2 Algorithm Design

Solution. By **1.1**, when $r < r^\star$, exists a negative cycle C on G'_r, which is also a cycle on G, s.t.

$$\sum_{(u,v)\in C} rc_{uv} - p_v < 0, \text{ i.e. } \frac{\sum_{(u,v)\in C} p_v}{\sum_{(u,v)\in C} c_{uv}} > r, \text{ i.e. } r(C) > r,$$

Thus, for any given ϵ, if we can find a $\hat{r}$ which satisfies the following requirements,

- on $G'_{\hat{r}+\epsilon}$ (the graph derived with $r = \hat{r}+\epsilon$), we cannot find any negative cycle.
- on $G'_{\hat{r}}$ (the graph derived with $r = \hat{r}$), we can find a negative cycle C.

we have $\hat{r} < r^\star < \hat{r}+\epsilon,\ r(C) > \hat{r} \Longrightarrow r(C) > \hat{r} > r^\star - \epsilon$, i.e. we find a good-enough cycle C.

Based on the idea above, we design the following algorithm.

Algorithm 1: Good-Enough Cycle Search

```
Function Bellman-Ford (G)
    Pick s ∈ V as the source node;                                  // G = (V, E).
    dist(s) ← 0;                     // dist(v) denotes the shortest distance from s to v.
    for v ∈ V \ {s} do   dist(v) ← ∞;
    repeat for |V| − 1 times
        for (u, v) ∈ E do
            if update(u, v) then   prev(v) ← u;
            // if taking edge (u, v) will generate a shorter path from s to v,
               update(u, v) updates dist(v) and returns True.
            // Otherwise, update(u, v) returns False
        end
    end
    for (u, v) ∈ E do
        if update(u, v) then
            Go back with the help of prev(·) and finds a cycle C;
            Return: C;
        end
    end
    Return: ∅
end

Function Good-Enough Cycle Search (G, ε, r_min, r_max)
    r̂ ← ⌊(r_min + r_max)/2⌋;
    C ← Bellman-Ford(Generate(G, r̂));
                                // The process of Generate(G, r) is explained in 1.1.
    if C = ∅ then
        Return: Good-Enough Cycle Search(G, ε, r_min, r̂);
    else
        if Bellman-Ford(Generate(G, r̂ + ε)) = ∅ then Return: C;
        else Return: Good-Enough Cycle Search(G, ε, r̂, r_max);
    end
end
```

The correctness of the algorithm is thoroughly explained on the previous page. As long as the range $[r_{min}, r_{max}]$ contains $r^\star$, we can always find a good-enough cycle.

Now we prove $r^\star \in [0, R]$. Obvious $r^\star \geq 0$. (Since $\forall u \in V,\ p_u > 0; \forall (u,v) \in E,\ c_{uv} > 0$.)

By the definition of $r^\star$, exists cycle $C^\star$ s.t. $r^\star = \frac{\sum_{(u,v)\in C^\star} p_v}{\sum_{(u,v)\in C^\star} c_{uv}}$. Since $C^\star$ is a cycle, $\sum_{(u,v)\in C^\star} p_v = \sum_{(u,v)\in C^\star} p_u$.

Meanwhile, since $R = \max\limits_{(u,v)} \{p_u / c_{uv}\}$, we know

$$r^\star = \frac{\sum_{(u,v)\in C^\star} p_v}{\sum_{(u,v)\in C^\star} c_{uv}} = \frac{\sum_{(u,v)\in C^\star} p_u}{\sum_{(u,v)\in C^\star} c_{uv}} \leq \frac{\sum_{(u,v)\in C^\star} Rc_{uv}}{\sum_{(u,v)\in C^\star} c_{uv}} = R.$$

Thus, *Good-Enough Cycle Search*$(G, \epsilon, 0, R)$ can return a good-enough cycle.

Now we analyze the time complexity of the algorithm given above. Let the time complexity be $T(|V|, \epsilon, range)$, where $range = r_{max} - r_{min}$.

We know the time complexity of *Bellman-Ford* is $T(\textit{Bellman-Ford}) = O(|V||E|) = O(|V|^3)$. (When $|E|$ is unknown, we have $|E| \leq 2 \times \frac{|V|(|V|-1)}{2} = |V|(|V|-1)$, i.e. $|E| = O\left(|V|^2\right)$.)

Moreover, when $range < \epsilon$, it is trivial that $T\left(|V|, \epsilon, range\right) = 2 \times T(\textit{Bellman-Ford})$.

Therefore,

$$\begin{aligned}
T(|V|, \epsilon, range) &\leq 2 \times T(\textit{Bellman-Ford}) + T\left(|V|, \epsilon, \frac{range}{2}\right) \\
&= 2T(\textit{Bellman-Ford}) + T\left(|V|, \epsilon, \frac{range}{2}\right) \\
&= 2T(\textit{Bellman-Ford}) + 2T(\textit{Bellman-Ford}) + T\left(|V|, \epsilon, \frac{range}{2^2}\right) \\
&= \ldots \\
&= (\log(range) - \log(\epsilon)) \times 2T(\textit{Bellman-Ford}) + T\left(|V|, \epsilon, \epsilon\right) \\
&= (\log(range) - \log(\epsilon))\, O(|V|^3).
\end{aligned}$$

We run *Good-Enough Cycle Search*$(G, \epsilon, 0, R)$ to get the result, i.e. $range = R$.

Thus, the time complexity of our algorithm is $O\left(|V|^3 \left(\log(R) - \log(\epsilon)\right)\right)$. ■

2 Problem 02

2.1 Eulerian Circuit and Eulerian Path

Solution. We use $\texttt{deg}_{\texttt{in}}(v)$ and $\texttt{deg}_{\texttt{out}}(v)$ to denote the in-degree and out-degree of vertex v respectively.

First we prove that a strongly connected directed graph $G = (V, E)$ contains Eulerian circuits **iff.** the in-degree and out-degree of each vertex $v \in V$ are the same.

Proof of Neccesity.

We prove the necessity by contradiction.

Assume exists $u \in V$ s.t. $\texttt{deg}_{\texttt{in}}(u) \neq \texttt{deg}_{\texttt{out}}(u)$, while G contains Eulerian circuits.

Without loss of generality, suppose $\texttt{deg}_{\texttt{in}}(u) > \texttt{deg}_{\texttt{out}}(u)$.

Since $\sum_{v\in V} \texttt{deg}_{\texttt{in}}(v) = \sum_{v\in V} \texttt{deg}_{\texttt{out}}(v)$, we know exists $u' \neq u$ s.t. $\texttt{deg}_{\texttt{in}}(u') < \texttt{deg}_{\texttt{out}}(u')$.

By the definition of Eulerian circuit, each edge will be visited once and only once. Then all edges adjacent to u will be visited once and only once. Thus, the Eulerian circuit will visit u through an edge and leave from u through another unvisited edge.

After visiting u $\texttt{deg}_{\texttt{out}}(u)$ times, we find that there are still $(\texttt{deg}_{\texttt{in}}(u) - \texttt{deg}_{\texttt{out}}(u))$ edges adjacent to u remaining unvisited. However, if we take any of these edges to visit u, we cannot find any unvisited edge out of it, i.e. the last vertex in the Eulerian circuit is u.

Similarly, after visiting u' $\texttt{deg}_{\texttt{in}}(u')$ times, we cannot find any unvisited edge into u', i.e. the first vertex in the Eulerian circuit is u'.

Meanwhile, the Eulerian circuit is a circuit, i.e. the first vertex and the last vertex must be the same. Thus, $u = u'$. **Contradiction!**

Therefore, a strongly connected graph $G = (V, E)$ contains an Eulerian circuit

$$\implies \forall v \in V, \texttt{deg}_{\texttt{in}}(v) = \texttt{deg}_{\texttt{out}}(v).$$

Proof of Sufficiency.

We define a cycle-search action on strongly connected graph $\hat{G}$ as follows.

- Select any vertex $u \in V$ s.t. on graph $\hat{G}$, $\texttt{deg}_{\texttt{in}}(u) = \texttt{deg}_{\texttt{out}}(u) > 0$.
- Since $\hat{G}$ is strongly connected, we can always find a cycle C on G starting from u and ending at u.
- Find the place of u in C_{Euler}. Replace it with C, i.e. to insert cycle C into C_{Euler}.

Then we can construct an Eulerian circuit by following steps.

1. $G_0 = G$.
2. Apply cycle-search action on G_t. In this process, we update C_{Euler}.
3. We can derive a new graph G_{t+1} from G_t by removing C from G_t.
4. For each strongly connected component of G_{t+1} which contains more than one vertex, we repeat step **2** to step **4** until $G_{t+1} = (V, E_{t+1}), E_{t+1} = \varnothing$.
5. C_{Euler} is an Eulerian circuit.

We can prove C_{Euler} is an Eulerian circuit. It is trivial that C_{Euler} is a circuit.

It is also trivial that in step **4**, there are no edges between each strongly connected component. Otherwise, since we always remove even edges in step **3** (this is guaranteed by the property of circuit), exists a component $\mathbf{c}$ s.t. $\texttt{deg}_{in}(\mathbf{c}) \neq \texttt{deg}_{out}(\mathbf{c})$, which contradicts to $\forall v \in V, \texttt{deg}_{in}(v) = \texttt{deg}_{out}(v)$.

During the process, all edges are removed (for only once), i.e. all edges appears in C_{Euler} once. Therefore, C_{Euler} is an Eulerian circuit, i.e.

$$\text{a strongly connected graph } G = (V, E) \text{ contains an Eulerian circuit}$$
$$\implies \forall v \in V, \texttt{deg}_{\texttt{in}}(v) = \texttt{deg}_{\texttt{out}}(v).$$

In conclusion,

a strongly connected directed graph $G = (V, E)$ contains Eulerian circuits **iff.** the in-degree and out-degree of each vertex $v \in V$ are the same. ∎

The sufficient and necessary condition for the existence of an Eulerian path on a strongly connected graph $G = (V, E)$ is that exactly one of the following two conditions is satisfied.

- For any vertex $v \in V$, $\texttt{deg}_{\texttt{in}}(v) = \texttt{deg}_{\texttt{out}}(v)$.
- Exists exactly one $u \in V$ s.t. $\texttt{deg}_{\texttt{in}}(u) = \texttt{deg}_{\texttt{out}}(u) + 1$ and exactly one w s.t. $\texttt{deg}_{\texttt{in}}(w) = \texttt{deg}_{\texttt{out}}(w) - 1$. For any vertex $v \in V \setminus \{u, w\}$, $\texttt{deg}_{\texttt{in}}(v) = \texttt{deg}_{\texttt{out}}(v)$.

End of Solution. ∎

2.2 Algorithm Design

Solution. In fact, the process of our algorithm to find an Eulerian circuit is fully explained in **2.1** *Proof of Sufficiency.* The pseudo-code is given below.

Algorithm 2: Eulerian Circuit Search

```
Function Eulerian Circuit Search (G)
    // Note that G = (V, E).
    Select u ∈ V randomly;
    C_Euler ← ∅;                  // We use single linked list to record the cycle.
    ∀v ∈ V, place(v) ← ∅;
    // place(v) denotes one of the appearances of v in C_Euler, i.e.  a pointer
       directing to a unit of C_Euler.

    while E ≠ ∅ do
        Select (u, v) ∈ E;
        // This guarantees that u and v are in a strongly connected component of
           G which contains more than one vertex.
        C ← Travel(G, u), which also remove C from E;
                                  // Detailed process of Travel(·) is defined below.
        C_Euler ← C_Euler ∪ C;    // Detailed process of this step is discussed below.
    end
    Return: C_Euler
end
```

(cont'd)

Function *Travel* (G, u_0)

```
    // Note that G = (V, E).
    C ← ∅;                                  // We use single linked list to record C.
    u ← u_0, v ← anything but u_0;
    while v ≠ u_0 do
        Select an edge (u, v);
        remove (u, v) from E;
        // With help of adjacency list, we just need to remove a unit in the
           adjacency list of u.  This can be completed with O(1) time.
        C ← C ∪ (u, v);
        u ← v;
    end
end
```

We use an adjacency list to store E, use single linked list to store cycle C and C_{Euler}.

How we realize $C_{Euler} \leftarrow C_{Euler} \cup C$ is explained as follows. Considering C is generated by Travel(G, u), we know the first and the last vertex C visits are both u. Therefore, by inserting C between *place*(u) and the next unit of *place*(u), we successfully insert C into C_{Euler}.

Now we analyze the time complexity of our algorithm.

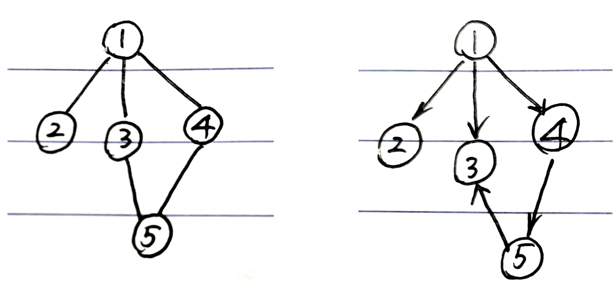
By the analyses above, both removing an edge and inserting C into C_{Euler} takes $O(1)$ time. Both node selection in *Eulerian Circuit Search*(G) and edge selection in $Travel(G, u)$ takes $O(1)$ time. During the process, each edge is visited once and only once, i.e. taking $O(|E|)$ time.

Thus, the total time complexity of our algorithm is $O(|E|)$. ■

3 Problem 03

3.1 G' is Not Necessarily Strongly Connected

Solution. A counter-example is as follows. ■



3.2 No Cut Edge Exists in G If G' is Strongly Connected

Proof. Here we use $p_{u\to v}$ to denote the set of all edges on a particular path from u to v.

Since G' is strongly connected, we know for any vertices $u, v \in V$, exists a path $p_{u\to v}$ from u to v and a path $p_{v\to u}$ from v to u. Obvious $p_{u\to v} \cap p_{v\to u} = \varnothing$. Thus, in undirected G, exist two totally different paths between u and v, i.e. $p_{u\to v}$ and $p_{v\to u}$. Removing any single edge from G will destroy at most 1 path between u and v, but the other path between u and v remains.

Therefore, for any vertices $u, v \in V$, after removing any single edge, u and v are still connected, i.e. removing any single edge from G will still give a connected graph.

Qed. ∎

3.3 G' is Strongly Connected If No Cut Edge Exists in G

Proof. Use $\texttt{des}(u)$ and $\texttt{anc}(u)$ to denote the set containing all descendants and ancestors of vertex u in the DFS tree respectively.

We prove the proposition by contradiction.

Assume when removing any edge from G still gives a connected graph, exists $u, v \in V$ s.t. there is no path from u to v on G'.

Since G' is still connected after removing a single edge, G itself is connected. Then there exists a path $p_{v\to u}$ from v to u on G', otherwise u and v are not connected in G. Meanwhile since DFS will definitely visit all vertices in G, we know v is an ancestor of u.

It is trivial that $\forall w \in \{u\} \cup \texttt{des}(u)$, $\forall x \in \{v\} \cup \texttt{anc}(v)$, there are no paths from w to x. Otherwise, $u \to w \to x \to v$ is a path from u to v. This yields that in graph G, all paths between x to w must contain edges in $p_{v\to u}$.

Thus, if we remove any edge in $p_{v\to u}$ from G, there is no path between w and x. **Contradiction** to G's property, i.e. removing any single edge from G will still give a connected graph.

Therefore, if removing any single edge from G can still give a connected graph, G' are strongly connected.

Qed. ∎

3.4 Algorithm Design

Solution. We know an edge removing which would make a undirected graph no longer connected is called a *cut edge*.

Inspired by **3.2** and **3.3**, we generate a G' from G by orienting edges as follows.

For each edge in the DFS tree, the direction is from the parent to the child; for other edges, the direction is from the descendant to the ancestor.

By the analyses in **3.2** and **3.3**, within a strongly connected component $C = (V_c, E_c)$ of G', there are no cut edge among V_c on G and cut edges appear and only appear between different strongly connected component.

Therefore, edges between two vertices in different strongly connected components of G' are cut edges. Based on the idea, we can design an algorithm as follows.

Algorithm 3: Cut Edge Search

Procedure *Generate* (G)
- $T \leftarrow DFS(G)$; `// T is the DFS tree.`
  - `// Meanwhile, orient edges on` G `from the parent to the children.`
- **for** $e \in E$ **do**
  - **if** $e \notin T$ **then** Orient e from the descendant to the ancestor in T;
- **end**

end

Function *Cut Edge Search* (G)
- $G' \leftarrow G, G' \leftarrow Generate(G')$;
- *Strongly Connected Component Search*(G');
- `// Implement the algorithm we discussed in Lecture 4.`
- `// Use` $comm(v)$ `to record the strongly connected component involving` v`.`
- *Cut Edges*$\leftarrow \varnothing$;
- **for** $(u, v) \in E$ **do**
  - **if** $comm(u) \neq comm(v)$ **then** *Cut Edges*$\leftarrow$*Cut Edges* $\cup \{u, v\}$;
- **end**
- **Return:** *Cut Edges*

end

Now we analyze the time complexity of the algorithm above.

We know *Strongly Connected Component Search*(G) runs DFS twice on the graph G, taking $O(|V| + |E|)$ time. Meanwhile, *Generation*(G) runs a DFS on G and then scans all edges in E. Thus, *Generation*(G) take $O(|V| + |E|)$ time.

Therefore, our algorithm takes $O(|V| + |E|)$ time. ■

4 Problem 04

4.1 Dijkstra-Variant's Faliure on DAG

Disproof. A counter-example is as follows, where G is a directed graph.

$$\begin{aligned} G = (V, E, \texttt{weight}), \\ V = \{1,2,3\}, \\ E = \{(1,2),(1,3),(2,3)\}, \\ \texttt{weight}(1,2) = -100, \texttt{weight}(1,3) = 1, \texttt{weight}(2,3) = 100. \end{aligned}$$

If we apply the Dijkstra-Variant Algorithm on G, we get $G' = (V, E, w')$, where $w'(1,2) = 0$, $w'(1,3) = 101, w'(2,3) = 200$. The shortest path from 1 to 3 on G' is $1 \to 3$. However, the shortest path from 1 to 3 on G is $1 \to 2 \to 3$, whose total weight is 0, smaller than 1, the weight of $1 \to 3$.

Therefore, the algorithm does not work on directed acyclic graphs. ∎

4.2 Dijkstra-Variant's Success on Directed Grids

Proof. We define $\texttt{rank}(v_{ij}) \triangleq i + j$.

We use $\texttt{weight}_G(\cdot)$ to denote the weight of a path on graph G. Let $\mathcal{P}_{u\to w}$ be the set containing all paths from u to w on G'.

In a directed grid, for any edge $e \in E$, it is either from v_{ij} to $v_{(i+1)j}$ or from v_{ij} to $v_{i(j+1)}$. Obvious for any edge $(u, w) \in E$, $\texttt{rank}(w) = \texttt{rank}(u) + 1$.

Thus, along any path on G, the $\texttt{rank}$ of the vertices is monotonously increasing. Moreover, the difference of $\texttt{rank}$ of two adjacent vertices on any path is exactly 1.

Therefore, for any $u, w \in V$, if exists a path from u to w on G',

- For any $p \in \mathcal{P}_{u\to w}$, p is also a path from u to w on G.
- $\texttt{rank}(w) > \texttt{rank}(u)$;
- All paths from u to w consists of $(\texttt{rank}(w) - \texttt{rank}(u))$ edges.
- For any $p \in \mathcal{P}_{u\to w}$, $\texttt{weight}_G(p) = \texttt{weight}_{G'}(p) + (\texttt{rank}(w) - \texttt{rank}(u))\, W$.
- Obvious $\min\limits_{p\in\mathcal{P}_{u\to w}} \texttt{weight}_G(p) = \min\limits_{p\in\mathcal{P}_{u\to w}} \texttt{weight}_{G'}(p)$.
  (Since for fixed u and w, $(\texttt{rank}(w) - \texttt{rank}(u))\, W$ is a constant.)

Thus, the shortest path p from u to w on G' found by Dijkstra-Variant Algorithm is also a shortest path from u to w on G, i.e.

the variant of Dijkstra algorithm works on directed grids. ∎

5 Rating and Feedback

The completion of this homework takes me five days, about 27 hours in total. Still, writing a formal solution is the most time-consuming part. But I suppose I am getting familiar with *latex*.

The ratings of each problem is as follows.

Problem	Rating
1.1	3
1.2	2
2.1	2
2.2	2
3.1	1
3.2	2
3.3	3
3.4	2
4.1	1
4.2	2

Table 1: Ratings.

This time I finish all problems on my own. (It is possible that some ideas come from some gossips with Sun Yilin.)