CSCI 665 Foundation of Algorithms
Due 12/09 11:59pm

Homework 6 **Guo, Zizhun (zg2808)**

Problem 1:

Algorithm Description

For this question, I implemented the Bellman-Ford algorithm to determine whether *G* contains a negative-weight cycle. This algorithm is implemented as dynamic programming algorithm. The heart of solution is:

$\mathrm{d[v]} = the\ sum\ of\ weights\ of\ the\ shortest\ path\ from\ vertex\ 0\ to\ v$

$\text{foreach edge } u \rightarrow v$

$$\mathrm{d[v]} = d[u] + w(u,v) \quad if\ d[v] > d[u] + w(u,v)$$

Phase 1: Initialize each element of *d* array with value of IFNT (in this case is 10000), *d[0]* is 0.

Phase 2: Doing *(n - 1) times* outer for-loops, and *m times* of **relaxation,** which means to update each edge by traverse each vertex's neighbors.

Phase 3: Use a marker variable that **after (n-1) times** vertices traversal loops**, the relaxation still exists (**the value of ***d*** can still be changed**),** the marker variable would mark this graph ***g***, **contains the negative cycle**, because the cycle would be deducted to infinity.

Correctness:

The reason why it has at most ***(n - 1) times*** **relaxation is,** it only needs at most ***(n - 1) times*** for the starting vertex to reach the last vertex so that guarantee each vertex is at least being updated for once. In this case, m equals to (n – 1).

Running Time Estimate

Phase 1: $\boldsymbol{O(n)}$**.**

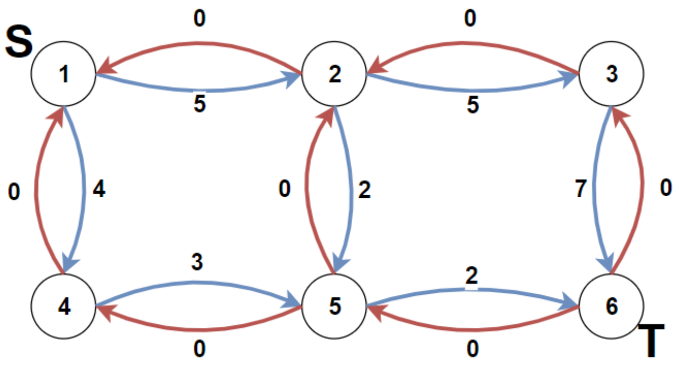
Phase2 & Phase3: $\boldsymbol{O(nm)}$**.**

So, overall, the running time estimate is bounded by $\boldsymbol{O(nm)}$**.**

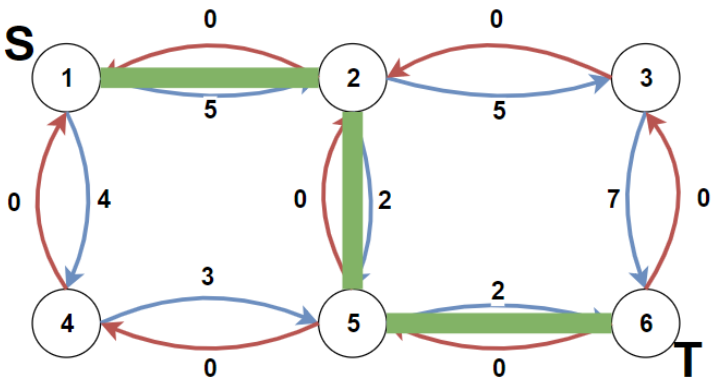
Pseudocode

```
Bellman-ford:
    Initialization
    foreach i from 0 to (n – 1):
        foreach e from 0 to m:
            if(d[u] > d[v] + w(u, v)):
                d[u] = d[v] + w(u, v)
    marker ← false
    foreach e 0 to m:
        if(d[u] > d[v] + w(u, v)):
            marker ← true
    print YES, if marker is true
    print No, if marker is false
```

Problem 2:

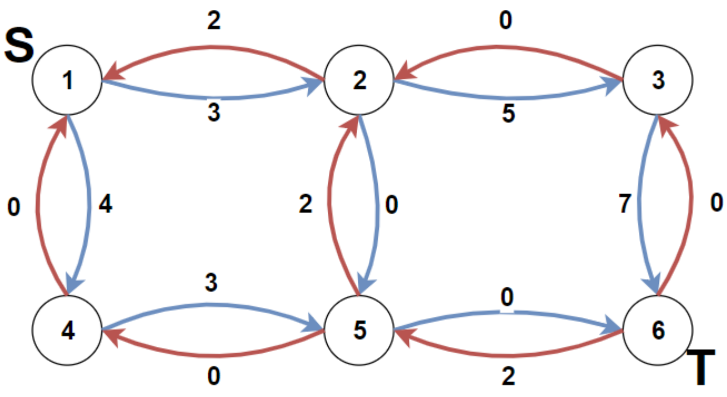


Residual graph 1 (original graph)

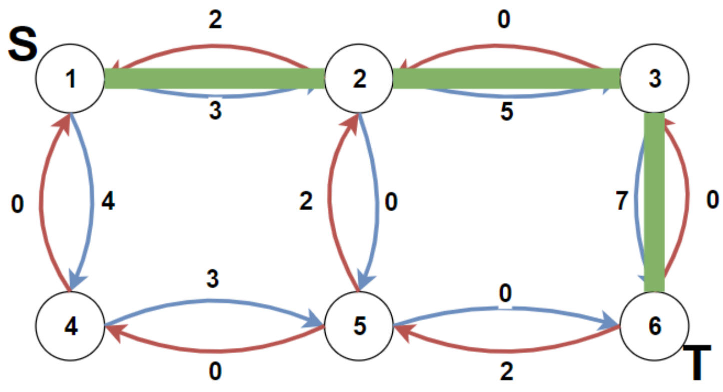


Augment path 1

Max flow for iteration 1 is 2.

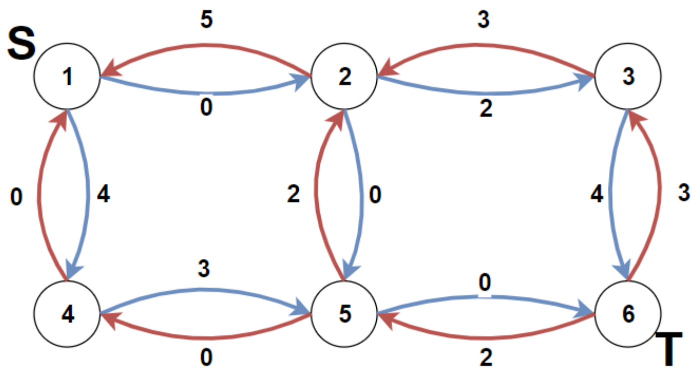


Residual graph 2

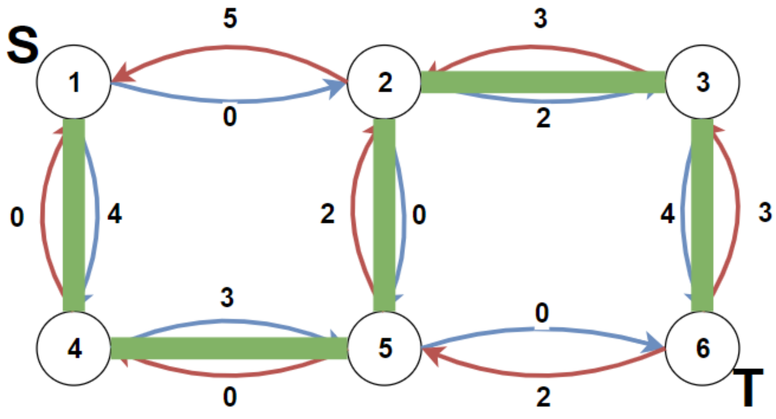


Augment path 2

Max flow for iteration 2 is 3.

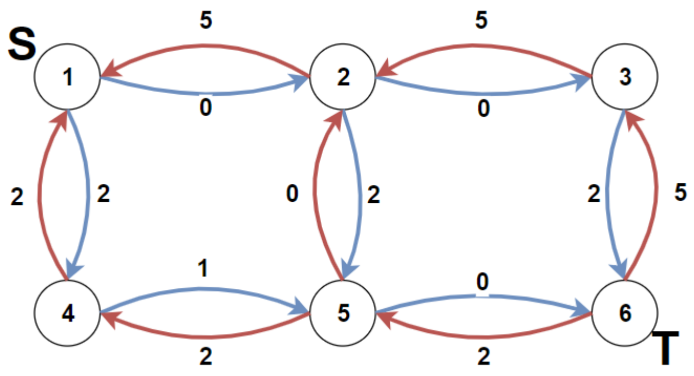


Residual graph 3



Augment path 3

Max flow for iteration 2 is 2. This path uses backward edge (5, 2).



Final residual graph

(current residual graph after ran the algorithm)

The maximum flow of G is 7.

Problem 3

Algorithm Description

For this problem, I implemented the Edmonds – Karp Algorithm to find the endpoints from two residual machines that are not connected before. The algorithm would be divides into 4 phases:

Phase 1: (Edmonds – Karp Algorithm)

The algorithm here follows the typical implementation of Edmonds- Karp algorithm that uses BFS to find an augment path each time, and get the max flow of that path, deducts the value of the max flow for forward edges and adds the value of the max flow for backward edges. Update the residual graph till there are no more augment paths that can be found. Stop and return the residual graph.

Data structure to implement Edmonds-Karp algorithm:

Use an adjacent array list to represent the overall graph. For every connected edge, assign both **backward edge** and **forward edge** in the same time.

Use adjacent matrix (two-dimensional array) to represent the **weights** and make change on it.

Phase 2: (Mark all vertices from two groups)

Use BFS starting from both ***Source*** vertex and ***Sink*** vertex.

For ***Source*** vertex, check **forward edges** to see if it is connected, and check the value of **forward edges** and make sure the value is greater than 0. If it is greater than 0, the BFS does not reach to endpoint. If it is equal to 0, the BFS reaches to the endpoint.

For ***Source*** vertex, check **backward edges** to see if it is connected, and check the value of **forward edges** and make sure the value is greater than 0. If it is greater than 0, the BFS does not reach to endpoint. If it is equal to 0, the BFS reaches to the endpoint.

Phase 3: (Sorting all vertices from two groups)

Use merge sort to sort all vertices from two groups in ascending lexicographically order.

Phase 4: (Check if the edge had ever existed)

Check vertices from two groups that if they were edges or not, two for loop iterates them till one pair match. Since the vertices are sorted in the ascending order, so as long they are found, print out the two vertices.

Running Time Estimate

Phase 1: (Edmonds – Karp Algorithm)

Based on the slides, the running time is $O\big(mn(m + n)\big)$, which equals to $O(nm^2 + mn^2)$. Since $\boldsymbol{m}$ is upper bounded by $\boldsymbol{n^2}$(reason: $\mathrm{m} = \binom{n}{2} = \frac{1}{2}\mathrm{n}(\mathrm{n} - 1) = \mathrm{O}(n^2)$), so mn^2 is upper bounded by nm^2. So overall, the estimate time is $\boldsymbol{O(nm^2)}$**.**

Phase 2: (Mark all vertices from two groups)

Based on running time for BFS, it is $\boldsymbol{O(mn)}$**.**

Phase 3: (Sorting all vertices from two groups)

Based on running time for *merge sort*, the it is $\boldsymbol{O(nlogn)}$**.**

Phase 4: (Check if the edge had ever existed)

For each vertex, since the data structure is adjacent array list, the time to check the edge is $\boldsymbol{O(deg(v))}$**,** which is upper bounded by $\boldsymbol{O(n)}$**.**

The two for-loops to traverse vertices taking less then $\boldsymbol{\binom{n}{2}}$**,** which is upper bounded by $\boldsymbol{O(m)}$**.**

So the overall time estimate for this phase is $\boldsymbol{O(\binom{n}{2}deg(v))}$**.** So the running time for this phase is $\boldsymbol{O(nm)}$**.**

Overall, this algorithm's estimated running time is upper bounded by $\boldsymbol{O(nm^2)}$**.**

Pseudocode

Phase 2 (Mark all vertices from two groups):

```
BFS for group that has Source vertex:
……
        get all neighbors of current vertex u:
            for each neighbor v:
                if !seen[v] and (u, v) > 0:
                    mark u
                else:
                    break
…..
```

```
BFS for group that has Sink vertex:
……
        get all neighbors of current vertex u:
            for each neighbor v:
                if !seen[v] and (v, u) > 0:
                    mark v
                else:
                    break
…..
```

Problem 4:

Q1

Input:

$$\begin{cases} G_1 = G \ (\forall w_{1i} = w_i) \\ s_1 = s \\ t_1 = t \end{cases}$$

The input for Q1 keeps the same as P, since they are both unweighted undirected graph G and two vertices s and t.

Output:

For Q2's result *l*, check the *l*'s value with number of (n - 1), if the ***l* = (n - 1),** the machine output ***true***, which for G(s, t) that it **has an s-t Hamiltonian path**.

If the ***l* < (n - 1),** the machine output ***false***, which for G(s, t) that it **does not has an s-t Hamiltonian path**.

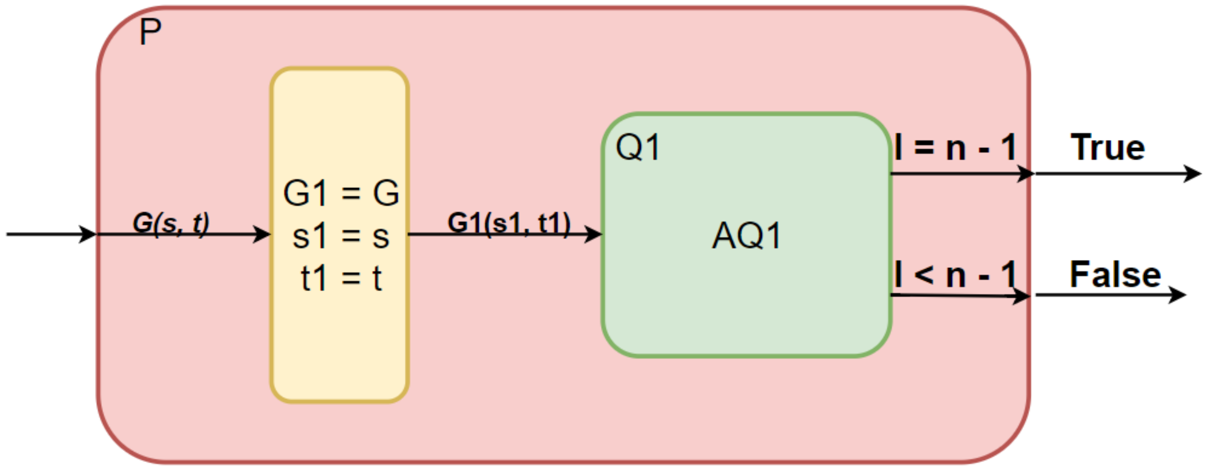


Figure 1: Poly – reduction construction for P∈ Q1

Poly-reducible bound's correctness of Q1 with P:

If there exists a longest path that has length of (n - 1), the path goes through every vertex. If the length of the longest path is smaller than (n - 1), the path fails to go through all vertices since there must exists at least 1 vertex that cannot be reached without being visited twice. If the longest path cannot reach all vertices, other paths that are shorter than the longest path would indeed cannot reach all vertices. This proves there does not exists a path that can go through all vertices from s to t.

Q2

Input:

$$\begin{cases} G_1 = -G\ (\forall w_{1i} = -w_i) \\ s_1 = s \\ t_1 = t \end{cases}$$

Construct the graph G_1as the $-\ G$dereived from G by changing every weight to its negation.

Output:

For Q2's result *l*, check the *l*'s value with number of (n - 1), if the ***l* = (n - 1),** the machine output ***true***, which for G(s, t) that it **has an s-t Hamiltonian path**.

If the ***l* < (n - 1),** the machine output ***false***, which for G(s, t) that it **does not has an s-t Hamiltonian path**.

Same as Q1.

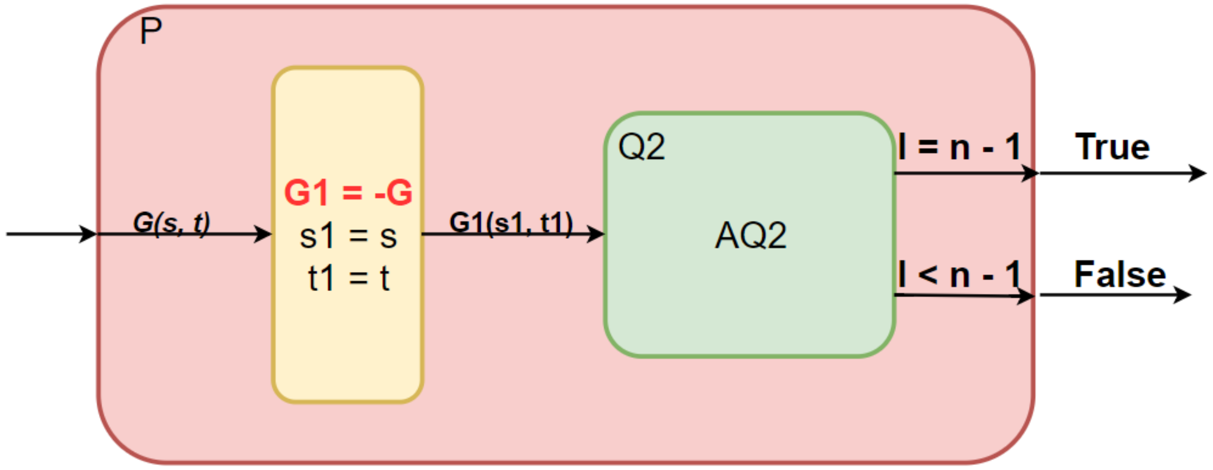


Figure 2: Poly – reduction construction for P∈ Q2

Problem 0:

Algorithm Description

For this question, I implemented the algorithm that get strong connected component from the graph, and to check if it has such edge that by change the direction of the edge, the graph can become strong connected.

Phase 1: (Topological Sort)

This phase employs a DFS algorithm to get the **topological sorting** order of all vertices and sorting them in **decreasing order** in finishing time. In this case, I used a stack to store them, so it would pop in the expected order.

Phase 2: (Getting all SCC and mark different SCC)

Reverse the current graph and use DFS traversing sorted vertices from the stack. Since the graph has been reversed, when traverse different vertex, the DFS will stop when the SCC is found, which means all SCC can be found and be marked. In this case, I used an array to represent each vertices' group.

Phase 3: (Find the reversable (v, u))

Check all edges and mark the **(u, v)** that **u** is from one group of SCC and **v** is from another group of SCC. If such type of edge exists, add the (**u, v**) into the list **U**, or add the **(v, u)** into list **V**. (The reason is, the edges are directed, we need to specify the direction). After traversed all edges, **check two lists**, if any of them **has size greater than 2**, it means any of edge from this list can be used as **changeable (u, v) to print. (This guarantee that after change the direction of this edge, the graph is strong connected, because vertex can go through path from its own group to another and have a path going back)**

Running Time Estimate

Phase 1: Topological sort is $\boldsymbol{O(nm)}$**.**

Phase 2: DFS modified algorithm has same running time as DFS which in this case is $\boldsymbol{O(n+m)}$**.**

Phase 3: Check all edges is $\boldsymbol{O(m)}$**.**

Overall, the time estimate is bounded by $\boldsymbol{O(nm)}$.

Pseudocode

```
Phase 2 (find SCC and mark vertices)
     boss[V] ← -1
     numofSCC ← 0
     seen[V] ← false
          foreach v pop from stack (topological decreasing order):
               if seen[v] → false
                    boss[v] ← SCCnumber
                    dfs(boss, seen, numofSCC)
                    SCCnumber++
dfs(boss, seen, numberofSCC):
......
          foreach s of neighbor of v
               If !seen[s]
                    boss[s] ← SCCnumber
                    …
......
```

```
Phase 3 (Check all edges)
     foreach (u, v):
          if u > v:
               groupA ←append(u, v)
          if u > v:
               groupB ←append(v, u)
     if groupA's or groupB's cout >= 2
          extract one (u, v) from A or B, and print it.
```