1 Minimum Edges From DG to SC Algorithm

In this report, we will determine the minimum number of edges we should add to a directed graph so the entire graph becomes strongly connected. We will divide this into two parts: [1] the theory, and [2] the algorithms. The first part will prove the number of minimum edges to add. In the second part, develop an algorithm to compute the specific edges to add. In the end, We shall present an appendix for the implementation of the algorithm in C++.

This problem was taken from the CSES problem set (short for Code Submission Evaluation System), a popular problem set for competitive programming [2]. The graph theory notations are developed from Professor Richard [4] lecture notes on the Graph Theory course this report is accompanying. The LaTeX styling template is from Evan Chen [1].

§1.1 The Theory

It turns out that without knowing what specific vertex-to-vertex connections we need to add, we can figure out the number of minimum edges K with just theory alone.

Definition (Strongly Connected)**.** A directed graph $G = (V, E)$ is strongly connected if for any $x, y \in V$, there exists one path from x to y.

Definition (Strongly Connected Component)**.** A strongly connected component describes a maximal connected subgraph that is strongly connected.

Definition (Contraction)**.** Let $H = (V_H, E_H)$ be a connected subgraph of a graph $G = (V, E)$. The contraction of H to a vertex is the replacement of V_H by a single vertex k. Any edge between a vertex of V_H and a vertex $V \backslash V_H$ is replaced by an edge between k and the same element of $V \backslash V_H$, while all edges of E_H do not appear in the contraction.

Question. Given a directed graph $G = (V, E)$, find one possible set of edges E_{new} such that $G_{\text{new}} = (V, E \cup E_{\text{new}})$ is strongly connected, and the edges set is of minimal size.

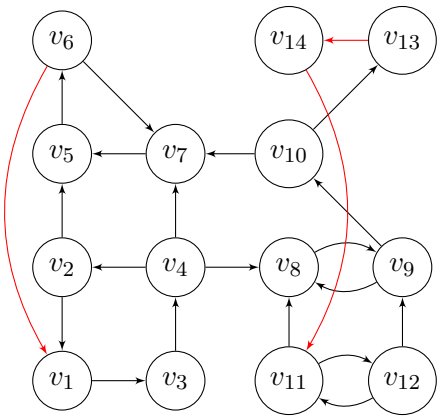

Figure 1.1: The minimum edges to make this example graph G strongly connected is 3, and the set of edges E_{new} in red is one possible construction.

In this section, we will figure out the size of such E_{new}.

Proposition (Observation 1)

For a directed graph $G = (V, E)$, if we keep contracting all the strongly connected components of size > 1 until there are no strongly connected components of size > 1 left. Then the final graph $G' = (V', E')$ has no cycles, or in other words, no non-trivial closed paths.

Proof. Assume there exists a cycle of length $N > 1$ with its walk $W = (x_0, e_1, x_1, e_2, \ldots, x_{N-1}, e_N, x_N)$, where $x_0 = x_N$. Then we see that $\forall x_i \in \{x_0, \ldots, x_{N-1}\}$, then there exists a path from x_i to x_j for $\forall x_j \in \{x_0, \ldots, x_{N-1}\}$. We know this new path exists because we can just start from x_i and move along in W until we get x_j.
Therefore, we can see that if such a cycle exists, then the vertices in the W it covers must also be a strongly connected component. But we know this can't be true as we have already contracted all the strongly connected components, reaching contradiction. □

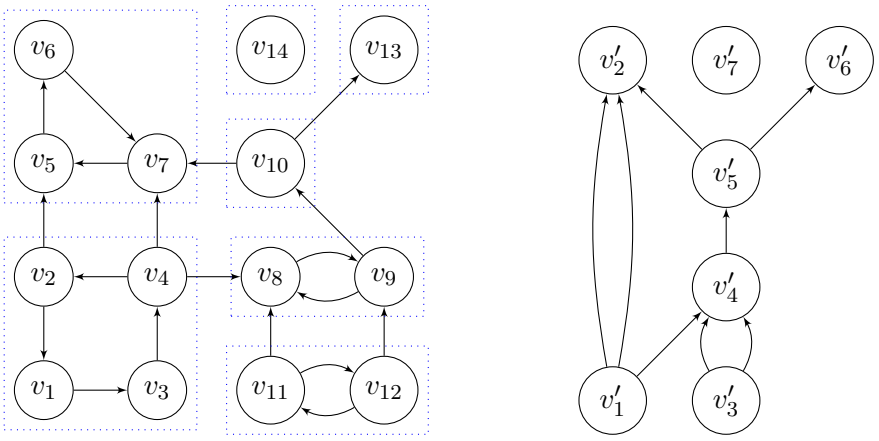

Figure 1.2: The strongly connected components of the graph G on the left are boxed in blue. And after contraction the graph G' on the right has no cycles.

Definition (Directed Acyclic Graph)**.** A directed graph without cycles is acyclic.

Proposition (Observation 2)

A directed acyclic graph $G = (V, E)$ will always have a non-empty set of vertices with no indegree and a non-empty set of vertices with no outdegree. This means there exists a set of vertices that are not the initial vertex, and another set of vertices that are not the terminal vertex, for any of the edges $e \in E$.

Proof. Let's divide the proof into two parts. We first show the set of vertices with no indegree is non-empty. Then we show the set of vertices with no outdegree is also non-empty.

1. Assume all vertices $v \in V$ have at least one indegree. Let's start at a vertex $v_1 \in V$, we have that there exists $v_2 \in V, e_1 \in E$ such that $i(e_1) = (v_2, v_1)$. We can then

start at vertex v_2, and see there exists $v_3 \in V, e_2 \in E$ such that $i(e_2) = (v_3, v_2)$. Since each vertex has an indegree, we can repeat this process for $n = |V| + 1$ times. Now during our walk of length n, we are guaranteed to hit one vertex at least twice because there can only be $|V|$ unique vertices. Let's call the vertex we encountered twice v_k, then we see there exists a cycle from v_k to itself by following a segment of the walk we generated in reverse. This is not possible if our graph starts off as acyclic, reaching contradiction.

2. We flip each edge in the original graph G such that the initial and terminal vertices are swapped.
   We know this flipped graph must have a non-zero set of vertices with no indegree. So we must have a set of vertices with no outdegree for the unflipped graph.

□

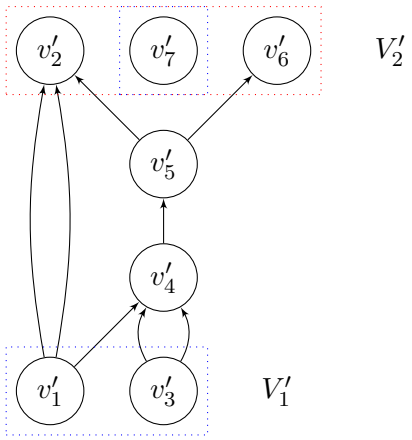

Figure 1.3: Graph G'. A Set of vertices V_1' with no indegree is boxed in blue, and a set of vertices V_2' with no outdegree are boxed in red.

From here, in our original problem, if we truncate the original graph $G = (V, E)$ to $G' = (V', E')$ so it becomes acyclic, then we see that there will always be a set of non-empty vertices with no indegree V_1' and another non-empty set of vertices with no outdegree V_2'.

Let $V'' \coloneqq V_1' \cup V_2'$, then create a bipartite graph $G'' = (V'', E'')$, such that vertex $v_i \in V''$ can only have an edge $e \in E''$ to vertex $v_j \in V''$ if one can traverse from v_i to v_j in G'.

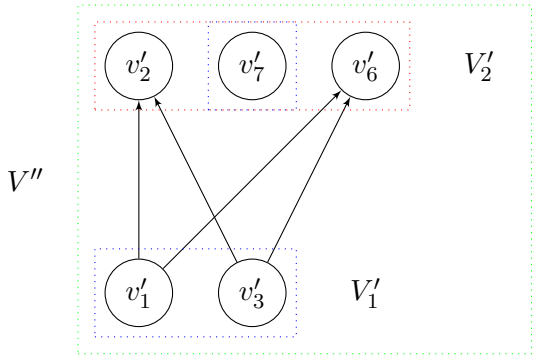

Figure 1.4: Graph G''.

Definition (Useless Vertex)**.** A vertex is useless if it has non-zero indegree and outdegree.

> **Remark.** I named this type of vertex useless because its name will soon make sense in Observation 6.

Proposition (Observation 3)

If we add some edges E''_{new} to G'' so it becomes strongly connected. Then if we add those same edges to G' so it becomes $G'_{\text{new}} = (V', E' \cup E''_{\text{new}})$, we will also have a strongly connected graph.

Proof. Consider G'. For every useless vertex $v_a \in V'$, there exist $v_i \in V'_1, v_j \in V'_2$ such that the path from v_i to v_j contains v_a. We can see this by turning G' into an undirected graph, then taking one traversal where the vertices are chosen if the edge between the current vertex and the following vertex is traversable in the original G' and taking another traversal where the vertices are chosen if the edge between them is not traversable in the original G'. We will eventually reach a vertex where there is no indegree and a vertex with no outdegree. Otherwise, we will have a cycle as proved by Observation 2, which cannot be the case.
Since G'' is defined that $\forall v_i \in V'_1$ to $\forall v_j \in V'_2$ if v_i and v_j has an edge, then one can traverse from v_i to v_j in the original G'. So if G'' is strongly connected by the new edges, then we can go from any $v_j \in V'_2$ to any $v_i \in V'_1$ and vice versa in G' as well.
Since every useless vertex is part of a path from $v_i \in V'_1$ to $v_j \in V'_2$, and V'_1 and V'_2 are strongly connected, we have that every useless vertex v_a can go to v_j, and therefore, go to any vertex in V'_1 and then any vertex in V'_2. If we want to go to another useless vertex v_b, we can just find the path between a vertex $v'_i \in V'_1$ and $v'_j \in V'_2$ that v_b belongs to and let v_a traverse to v'_i first, then to v_b. □

Proposition (Observation 4)

If we add some edges E''_{new} to G' so it becomes strongly connected. Then if we "add" those same edges E_{new} to G, we will also have a strongly connected graph.

> **Remark.** We define "add" as for every $e'' \in E''_{\text{new}}$, $i(e) = (v_i, v_j) \in V'' \times V'' \subset V' \times V'$, we add e to initially empty E_{new}, where $i(e)$ is an edge from any arbitrary vertex in the strongly connected component v_i represents to any arbitrary vertex in the strongly connected component v_j represents before contraction.

Proof. Assume there exists a pair of vertices $v_a, v_b \in V$ such that there is no path between them in $G_{\text{new}} = (V, E \cup E_{\text{new}})$. There are two cases we need to consider:

1. Case 1: Let v_a and v_b be part of the same strongly connected component.
   Since they are already part of the same strongly connected component, by definition of such component, we can traverse between v_a and v_b already. We arrive at a contradiction with our initial assumption.

2. Case 2: Let v_a and v_b be part of the different strongly connected components.
   Since we know if we add E''_{new} to G' we also have a strongly connected component as

shown from the previous observation, we see that any strongly connected component can traverse to any other strongly connected component. So v_a can just traverse to the strongly connected component v_b belongs, and through this strongly connected component, traverse to v_b. We arrive at a contradiction with our initial assumption.

□

Proposition (Observation 5)

The size of a minimum set of edges to make G'' strongly connected K is bounded by $\max(|V_1'|, |V_2'|) \leq K \leq |V_1'| + |V_2'|$

Proof. We can divide this proof into two parts.

1. Show $K \leq |V_1'| + |V_2'|$:
   As part of the proof for Observation 1, we see that if a cycle exists, then all the vertices in this cycle can traverse to each other, forming a strongly connected component in itself. So just by ignoring the already established edges E'' between V'', we can just form a cycle between those $|V''| = |V_1'| + |V_2'|$ vertices.
   The edges we need to add are equal to the number of vertices we have. This can be done by simply starting from one vertex $v_i \in V''$, constructing a directed edge to $v_j \in V''$ that is not traversed before, and then traversing to it. We do this for $|V''| - 1$ different times so we can traverse to all the vertices. Then we form a directed edge from our final vertex after $|V'| - 1$ traversals back to the initial vertex we started from.

2. Show that $\max(|V_1'|, |V_2'|) \leq K$:
   Assume the minimum set of edges to make G'' strongly connected is less than $\max(|V_1'|, |V_2'|)$. Then we will always see there is at least either an originally indegree 0 vertex that still has indegree 0, or an originally outdegree 0 vertex that still has outdegree 0. To make all vertices strongly connected, we need to have each vertex at least 1 indegree and outdegree in order for the vertex to be traversed to and from, reaching contradiction for the assumption.

□

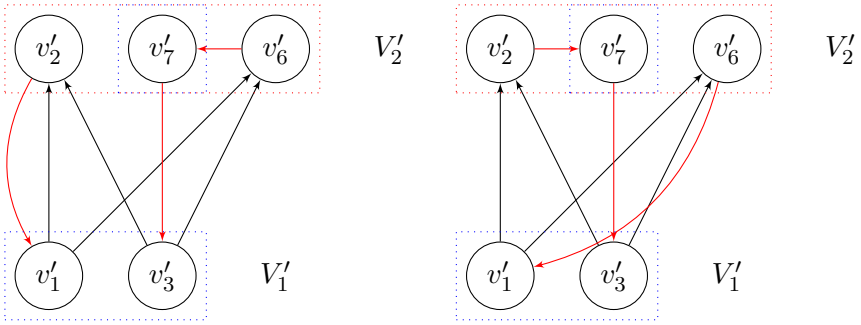

Figure 1.5: Two possible minimum sets of edges to make Graph G'' strongly connected denoted in red.

Proposition (Observation 6)

The number of minimum edges to make G'' strongly connected is equal to $K = \max(|V_1'|, |V_2'|)$.

Proof. To make G'' strongly connected, we need to fulfill the quota of adding at least one indegree to each vertex in V_1', and adding at least one outdegree to each vertex in V_2'. The idea is that it is in our best interest to add edges in a way that each edge can try to contribute to both indegree and outdegree quota at the same time. In other words, maximize the number of useless vertices we can create for each added edge until all vertices become useless in a way so everything belongs to one strongly connected component.

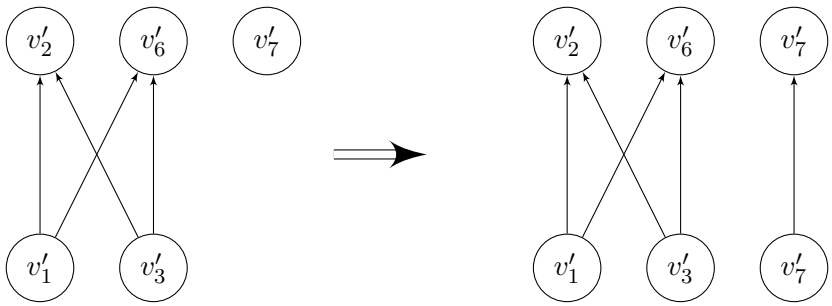

Figure 1.6: We treat a vertex with no indegree nor outdegree as two vertices that share the original vertex. One is responsible for incoming edges, and another for outgoing edges. This still makes sure that the vertex is counted in both V_1' and V_2'.

For the sake of not spending another paragraph dedicated to explaining edge cases arises from the below proof to singular vertices, which really just follows the same procedure. We will just treat a vertex with both 0 indegree and outdegree as two vertices one pointing to another, demonstrated in Figure 1.6. With this in mind, we can divide the proof into two steps.

1. If the sizes of the sets for zero indegrees and zero outdegree vertices are > 1, we can reduce the problem of finding K to a graph that either has $\|V_1'\| = 1$ or $\|V_2'\| = 1$. Let $G_0'' = (V_0'', E_0'') \coloneqq G'' = (V'', E'')$. We add an edge from vertex $v_a \in V_0''$ with no outdegree to $v_b \in V_0''$ with no indegree, such that v_a have at least one outward edge that doesn't point to v_b and v_b have at least one inward edge that doesn't come from v_a. We see that by adding this edge, we create two useless vertices from v_a and v_b. We know we can always find such a pair because no such pair exists only when the set of zero indegree or zero outdegree vertices is equal to exactly 1, which can't happen in this step. Now let's create $G_1'' = (V_1'', E_1'')$ so it is the same as G_0'' initially, but we remove v_a and v_b from V_1'' and the corresponding edges that start or end at v_a or v_b, then we add an edge to every vertex that has an edge to v_a in G_0'' to every vertex that has an edge from v_b in G_0''. This way we preserve the traversability of all vertices to others by adding edges but still able to make the graph simpler with fewer vertices. We can repeat our above step to create $G_2'', \ldots, G_n''$, until we exhaust the set of vertices with no indegree or vertices with no outdegree till the size of 1.

G''_n will either look like one vertex having one directed edge to the rest of vertices, or one vertex having one directed edge from the rest of vertices.

2. If G''_0 already only has one vertex with no indegree or with no outdegree, then we have $n = 0$ in our G''_n in the previous step.
   We see that the answer to make G''_0 strongly connected is equal to the answer to make G''_n strongly connected plus n. We know it cannot be any less than n since each edge's utility is maximized by contributing one indegree and one outdegree at the same time.
   Now we claim that the minimum number of edges to make G''_n strongly connected is to add $\max(|V_n^1|, |V_n^2|)$ edges, where V_n^1 and V_n^2 denotes the set of vertices in V''_n that have no indegree and outdegree correspondingly. We will first show this number allows us to turn G''_n strongly connected, then show it is the minimum number.
   We can see this number is true by simply connecting all vertices from V_n^2 to all vertices in V_n^1. This number indeed is also the minimum because it is the lower bound of minimum edges to make G''_n strongly connected as proved in Observation 5.

We have $|V_n^1| = |V'_1| - n$ and $|V_n^2| = |V'_2| - n$, then if G''_n takes $\max(|V_n^1|, |V_n^2|)$ edges to strongly connect, we have G'' takes $\max(|V'_1| - n, |V'_2| - n) + n = \max(|V'_1|, |V'_2|)$ edges to strongly connect. Because this number is also shown as the lower bound of K from Observation 5, we have this the minimum number. □

Proposition (Observation 7)

The number of minimum edges to make G'' strongly connected is the same as the number of minimum edges to make G' strongly connected.

Proof. The number of minimum edges to make G' strongly connected is lower bounded by $\max(|V'_1|, |V'_2|)$ as well. The same proof from Observation 5 can be used here.
And since we know the edges to make G'' strongly connected makes G' strongly connected from Observation 3. We see that the minimum edges K to make G'' strongly connected is the lower bound for G'. As this K is the lowest possible for G', we have Observation 7 proven. □

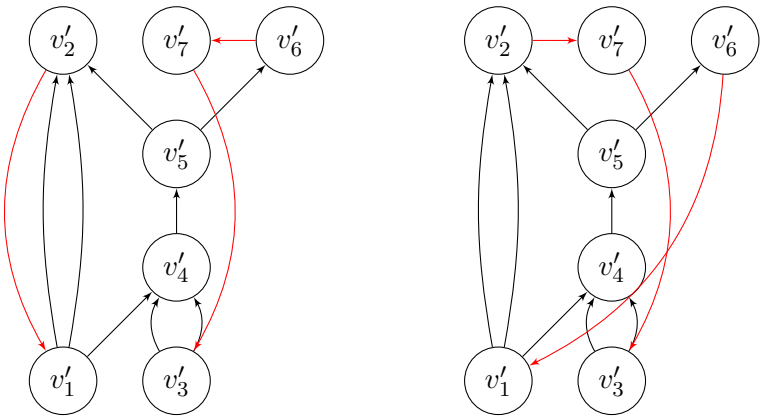

Figure 1.7: Two possible minimum sets of edges to make Graph G' strongly connected denoted in red.

Proposition (Observation 8)

The number of minimum edges to make G' strongly connected is the same as the number of minimum edges to make G strongly connected.

Proof. The number of minimum edges to make G strongly connected is lower bounded by $\max(|V_1'|, |V_2'|)$ as well. The same proof from Observation 5 can be used here by just adding that we need at least $|V_1'|$ directed edges that have a terminal vertex in strongly connected components that have no indegree when truncated, and we need at least $|V_2'|$ directed edges that have a starting vertex in strongly connected components that have no outdegree when truncated.
And since we know the edges to make G'' strongly connected makes G strongly connected from Observation 4. We see that the minimum edges K to make G'' strongly connected is the lower bound for G. As this K is the lowest possible for G, we have Observation 8 proven. □

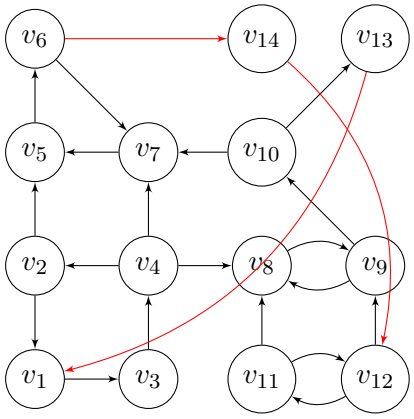

Figure 1.8: Other than Figure1.1, this is another set of minimum edges to make Graph G strongly connected.

Now we have shown that from Observation 6,7,8 that the size K of E_{new} to make G a strongly connected graph is exactly $\max(|V_1'|, |V_2'|)$, we complete this section of the report.

§1.2 The Algorithm

With knowing the exact number of edges K we need to add to G, in this section we develop a relatively fast algorithm to compute the K edges that when added to the graph G, we get a strongly connected graph.

Question. Let's restate the question in terms of computer-friendly format:

1. **Statement:**
   Given N vertices labeled from 1 to N and M directed edges, what is the minimum additional edges we need to add so every vertex can traverse to every other vertex.

2. **Input:**
   First line two integers N and M.
   Next M lines describe the directed edges. Each line has two integers a and b, meaning an edge starts from vertex a and ends at vertex b.

3. **Output:**
   Print an integer K in the first line.
   For the next K lines, print a and b on each line. Describing the new edges to be added that start from vertex a and end at vertex b.

4. **Constraints:**

$$1 \leq N \leq 10^5$$
$$1 \leq M \leq 2 \cdot 10^5$$
$$1 \leq a, b \leq N$$
Time Limit: 1.00s

5. **Sample Input:**

```
14 19
2 1
1 3
3 4
4 2
11 12
12 11
11 8
12 9
8 9
9 8
4 8
9 10
4 7
2 5
7 5
10 7
5 6
6 7
10 13
```

6. **Sample Output:**

```
3
6 1
13 14
14 11
```

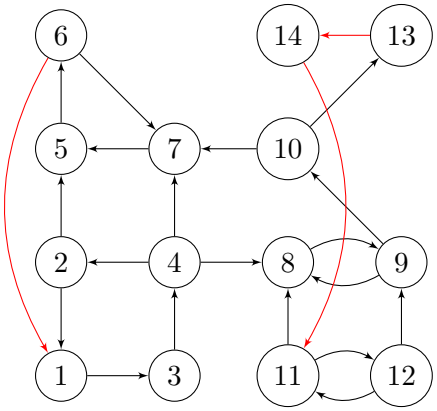

Figure 1.9: Sample input visualized. The red edges are the sample output.

The goal is to create an algorithm that runs the input and returns the correct output under the time limit without errors. The general rule of thumb is that average computers can roughly take $10^{8\pm1}$ computations per second. So our algorithm's time complexity can't be for example $\mathcal{O}(N^2 + M)$, as that would take in the worst case $(10^5)^2 + (2 \cdot 10^5) \approx 10^{10}$ computations, or around 100 seconds to finish the job.
In this section, we will develop an algorithm that successfully runs in $\mathcal{O}(N + M) = \mathcal{O}(|V| + |E|)$. We will use $|V|$ and N, $|E|$ and M interchangeably. As well as using the same mathematical notations developed in the previous section, such as G, G', G'', G''_n, $E_{\text{new}}, V'_1, V'_2$, and et cetera.

Algorithm (Kosaraju's algorithm [3]) — A way to find all the maximal strongly connected components in a graph $G = (V, E)$ in time complexity $\mathcal{O}(|V| + |E|)$. The algorithm is composed of two Depth First Searches.

Remark. Technically, the method involved in the proof for Observation 6 is an algorithm in itself. However in the worst case, assuming the given G is already bipartite $G'' = G' = G$ and every no indegree vertex has an edge to every no outdegree vertex, it takes up to $\frac{|V|^2}{4}$ iterations just to get $G''_0 \to G''_1$. This renders such algorithm ineffective under our problem time constraints.

Over here we still keep the same notion that a vertex represents a strongly connected component with both 0 indegree and outdegree as two vertices one pointing to another.

Algorithm (DG to SC algorithm) — Our algorithm's general layout

1. Process the inputs and construct the graph G with an adjacency list.
2. Run the modified Kosaraju's algorithm to detect all S strongly connected components and label them from 1 to S. The algorithm is modified in such a way, that it also records the directed edges between two vertices from different strongly connected components in an array.
3. Create a map $m : V \to V'$ that maps each vertex to its corresponding strongly connected component. We have $|V'| = S$.
4. Create a map $m' : V' \to V$ that maps each strongly connected component s to an arbitrary vertex v in G such that $m(v) = s$.

5. Construct the graph G' with an adjacency list. The edges for G' are from the array generated by the modified Kosaraju's algorithm.

6. Create four arrays V_1', V_2', and W_1', W_2'. The initial size of V_1' and V_2' are empty. W_1' and W_2' have an initial size of S and the values are all 0. The later two arrays represent the number of indegrees and outdegrees each vertex labeled between 1 to S has respectively.

7. Then read from the array generated by the modified Kosaraju's algorithm, and add 1 to either W_1' or W_2' if a vertex shows up as the initial or final vertex for an edge.

8. Read from the two arrays W_1' and W_2', add vertices with no indegree or no outdegree to V_1' and V_2' respectively.

9. Create an array A, this will be the array where we will store the minimum added edges to turn G' strongly connected.
   Create an array M of size S with initial values 0. This array will help us remember which vertices in the following we have visited previously.

10. Perform one Depth First Search of the graph G' starting from each vertex in V_1'. For each vertex v we have traversed to, we mark vth number in the array M 1. So for every future Depth First Search we perform, we will ignore vertices that are previously marked by reading from array M. For each Depth First Search, we will not stop until we exhaust our frontier, or encounter a vertex that belongs to V_2'.

11. For the subset of vertices $\{(v_1')_1, \ldots, (v_1')_k\}$ in V_1' that found a vertex in V_2' (collectively those vertices compose a subset $\{(v_2')_1, \ldots, (v_2')_k\}$). Through the above Depth First Search, we add k edges to array A.
    The edges are constructed in the following manner:
    $\{((v_2')_1, (v_1')_2), ((v_2')_2, (v_1')_3), \ldots, ((v_2')_k, (v_1')_1)\}$.

12. If $|V_1'| \leq |V_2'|$. For the remaining $|V_1'| - k$ vertices with no indegree, and $|V_2'| - k$ vertices with no outdegree, we add $|V_1'| - k$ edges to A. The edges are added that we pair each of those remaining vertex in V_2' to V_1'.

13. For the remaining $|V_2'| - |V_1'|$ vertices with no outdegree in V_2', we add $|V_2'| - |V_1'|$ edges to A. The edges are added so that each remaining vertex from V_2' is paired to an arbitrary vertex in V_1'.

14. In the case where $|V_1'| \geq |V_2'|$. We do the vice versa for steps 12 to 13.

15. Process array A with map m' so the vertices for each edge belong to G and not G'. Then output the answer in the format asked in the question.

The steps of this algorithm generally match the steps in the first section of this report. The only difference arises when we compute the edges that make G' strongly connected directly instead of transforming G' to G'' first. So to complete the algorithm, we will follow through with a proof that shows why steps 10-14 guarantee the edges set for G' make it strongly connected.

Proof. Between steps 10-14, we are adding $[k] + [\min(|V_1'| - k, |V_2'| - k)] + [\max(|V_1'|, |V_2'|) - \min(|V_1'|, |V_2'|)] = \max(|V_1'|, |V_2'|)$ edges. So we first see the algorithm will always add the minimum edges theoretically allowed.

At step 11, the k edges are constructed in a way such that $\forall v_i, v_j \in \{(v_1')_1, \ldots, (v_1')_k\} \cup \{(v_2')_1, \ldots, (v_2')_k\}$, we can have a traversal from v_i to v_j. So we have all those vertices under one strongly connected component.
At step 12, we see that by adding an edge from an unpaired vertex v_i from V_2' to an unpaired vertex v_j from V_1', we add both of them to this central strongly connected component that originally started by $\{(v_1')_1, \ldots, (v_1')_k\} \cup \{(v_2')_1, \ldots, (v_2')_k\}$. This is true because we can always have a vertex from the central strongly connected component traversable from v_j and another vertex from the central strongly connected component traversable to v_i (this statement is true because of the way step 10 picks the vertices to form the starting central connected component). By joining v_i to v_j, we make every vertex in the central strongly connected component traversable to v_i and v_j.
At step 13, we would have $\max(|V_1'|, |V_2'|) - \min(|V_1'|, |V_2'|)$ unpaired vertices either entirely from V_1' or entirely from V_2' left. If the case is former, we can simply establish an edge from an arbitrary vertex in V_2' to every vertex remaining in V_1', since all those remaining vertices can traverse to a vertex in V_2' that belongs to the central strongly connected component, this construction makes those rest vertices join the central strongly connected component as well. If the case is the latter, we can simply establish an edge for each vertex remaining in V_2' to an arbitrary vertex in V_1', since all those remaining vertices can be traversed from a vertex in V_2' that belongs to the central strongly connected component, this construction makes those rest vertices join the central strongly connected component as well. □

§1.3 Appendix

My C++ implementation of the said algorithm in the previous section is shown below. No dependencies are needed. The language is C++11.

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef pair<int,int> pii;
typedef tuple<int,int,int> tiii;
#define ff first
#define ss second

void toposort(vector<vector<int>>& graph, vector<bool>& visited, int node,
    vector<int>& topo){
    visited[node] = true;
    for (int i: graph[node]){
        if (visited[i]) continue;
        toposort(graph, visited, i, topo);
    }
    topo.push_back(node);
}

void scc(vector<vector<int>>& graph, vector<int>& component, int node, int type,
    vector<pii>& edges){
    component[node] = type;
    for (int i: graph[node]){
        if (component[i]){
            if (component[i] != type)
                edges.push_back(make_pair(component[i],component[node]));
            continue;
        }
        scc(graph, component, i, type, edges);
    }
}

int dfs(vector<vector<int>>& graph, vector<bool>& visited, int node, vector<int>&
    indegree, vector<int>& outdegree){
    if (visited[node]){
```

```
        return 0;
    }
    visited[node] = true;
    if (outdegree[node] == 0){
        return node;
    }
    for (auto curr: graph[node]){
        int res = dfs(graph, visited, curr, indegree, outdegree);
        if (res){
            return res;
        }
    }
    return 0;
}


int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<int>> graph(n+1);
    vector<vector<int>> reversed(n+1);
    for (int i = 0; i < m; i++){
        int a,b;
        cin >> a >> b;
        graph[a].push_back(b);
        reversed[b].push_back(a);
    }

    vector<bool> visited(n+1);
    vector<int> topo;
    for (int i = 1; i <= n; i++){
        if (visited[i]) continue;
        toposort(graph, visited, i, topo);
    }

    vector<int> component(n+1,0);
    vector<int> head;
    vector<pii> edges;
    for (int i = n-1; i >= 0; i--){
        if (component[topo[i]]) continue;
        head.push_back(topo[i]);
        scc(reversed, component, topo[i], head.size(), edges);
    }

    int p = head.size();
    vector<vector<int>> reduced_graph(p+1);
    vector<int> indegree(p+1);
    vector<int> outdegree(p+1);
    for (int i = 0; i < edges.size(); i++){
        reduced_graph[edges[i].ff].push_back(edges[i].ss);
        indegree[edges[i].ss]++;
        outdegree[edges[i].ff]++;
    }

    vector<int> matched_sinks;
    vector<int> matched_sources;
    vector<int> unmatched_sinks;
    vector<int> unmatched_sources;
    vector<bool> filled_sinks(n);
    fill(visited.begin(), visited.end(), 0);
    for (int i = 1; i <= p; i++){
        if (indegree[i] == 0){
            int node = dfs(reduced_graph, visited, i, indegree, outdegree);
            if (node){
                matched_sources.push_back(i);
                matched_sinks.push_back(node);
                filled_sinks[node] = true;
            }else{
                unmatched_sources.push_back(i);
            }
        }
```

```
    }
    for (int i = 1; i <= p; i++){
        if (outdegree[i] == 0 && !filled_sinks[i]){
            unmatched_sinks.push_back(i);
        }
    }

    int sources_remain = unmatched_sources.size();
    int sinks_remain = unmatched_sinks.size();
    int lr = min(sources_remain,sinks_remain);
    int matched = matched_sources.size();
    vector<pii> ans;
    if (head.size() == 1){
        sources_remain = 0;
        sinks_remain = 0;
        matched = 0;
    }
    for (int i = 0; i < matched; i++){
        ans.push_back(make_pair(matched_sinks[i],matched_sources[(i+1)%
        matched_sources.size()]));
    }
    for (int i = 0; i < lr; i++){
        ans.push_back(make_pair(unmatched_sinks[i],unmatched_sources[i]));
    }
    for (int i = lr; i < sources_remain; i++){
        ans.push_back(make_pair(matched_sinks[0],unmatched_sources[i]));
    }
    for (int i = lr; i < sinks_remain; i++){
        ans.push_back(make_pair(unmatched_sinks[i],matched_sources[0]));
    }

    cout << ans.size() << " \n";
    for (int i = 0; i < ans.size(); i++){
        cout << head[ans[i].ff-1] << " " << head[ans[i].ss-1] << "\n";
    }

    return 0;
}
```

The output above program produces from the same sample input in section 2 problem statement is of below:

```
3
14 11
13 1
5 14
```

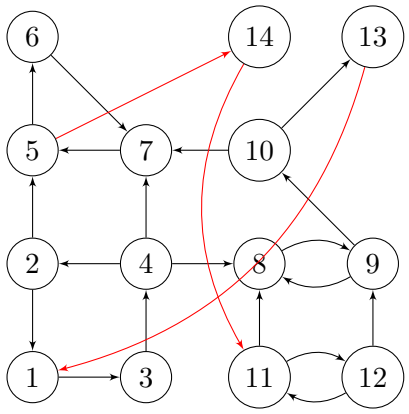

Figure 1.10: The red edges are generated by my algorithm implemented in C++.

Bibliography

[1] Evan Chen. *Latex styling template*. 2024. URL: https://github.com/vEnhance/dotfiles/blob/main/texmf/tex/latex/evan/evan.sty.

[2] Antti Laaksonen. *CSES Problem Set introduction*. 2018. URL: https://cses.fi/problemset/text/2433.

[3] Antti Laaksonen. *Kosaraju's algorithm*. 2018. URL: https://cses.fi/book/book.pdf#page=168.

[4] Serge Richard. *Graph Theory lecture notes*. 2024. URL: https://www.math.nagoya-u.ac.jp/~richard/teaching/s2024/Graph.pdf.