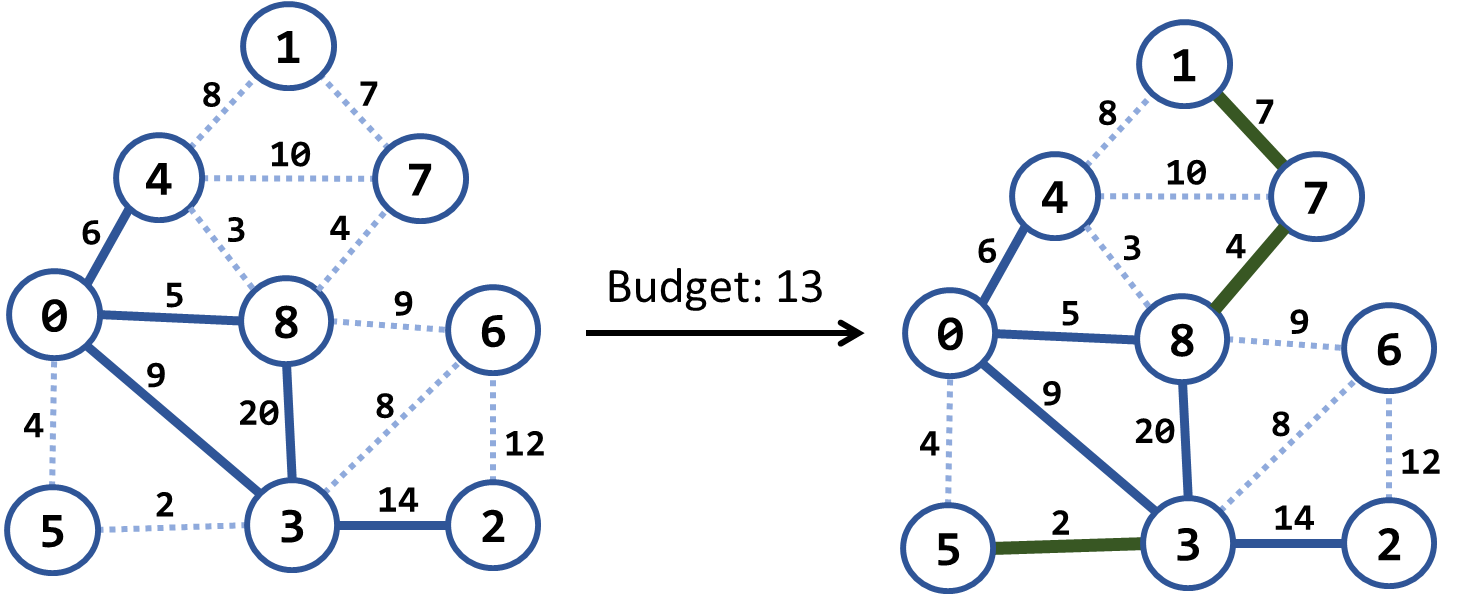
# Homework: Advanced Graph Algorithms

This document defines the **homework assignments** for the ["Algortihms" course @ Software University](https://softuni.bg/trainings/1194/Algorithms-September-2015). Please submit a single zip / rar / 7z archive holding the solutions (source code) of all below described problems.

## Extend a Cable Network

A cable networking company plans to extend its existing **cable network** by connecting as many customers as possible within a **fixed budget limit**. The company has calculated the **cost** of building some prospective connections. You are given the existing cable network (a set of **customers** and **connections** between them) along with the **estimated connection costs** between some pairs of customers and prospective customers. A customer can only be connected to the network via a direct connection with an already connected customer. Example:



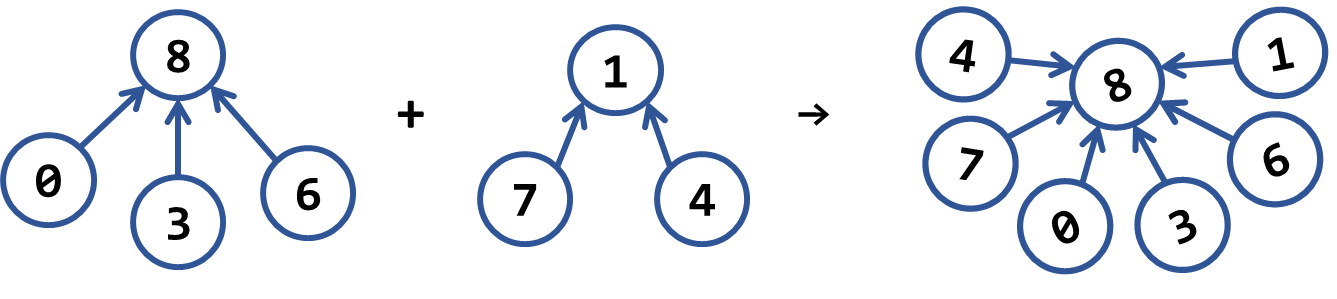
In the above example we have an existing cable network (the solid blue lines), the estimated costs for connecting some of the customers (dotted blue lines) and a budget limit of 20. Within this budget, the company can connect 3 new customers by the following new connections (solid green lines): {3 → 5}, {8 → 7} and {7 → 1}. The total cost for these new connections will be 2 + 4 + 7 = 13, which fits in the budget limit of 20. No more customers can be connected within this budget limit. Note that each edge, at the time of its addition to the network, connects a new customer with an existing one. Examples:

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Picture (Before)** | **Output** | **Picture (After)** |
| Budget: 20  Nodes: 9  Edges: 15  1 4 8  4 0 6 connected  1 7 7  4 7 10  4 8 3  7 8 4  0 8 5 connected  8 6 9  8 3 20 connected  0 5 4  0 3 9 connected  6 3 8  6 2 12  5 3 2  3 2 14 connected |  | {3, 5} -> 2  {8, 7} -> 4  {7, 1} -> 7  Budget used: 13 |  |
| Budget: 7  Nodes: 4  Edges: 5  0 1 9  0 3 4 connected  3 1 6  3 2 11 connected  1 2 5 |  | {1, 2} -> 5  Budget used: 5 |  |
| Budget: 20  Nodes: 8  Edges: 16  0 1 4  0 2 5  0 3 1 connected  1 2 8  1 3 2  2 3 3  2 4 16  2 5 9  3 4 7  3 5 14  4 5 12  4 6 22  4 7 9  5 6 6  5 7 18  6 7 15 |  | {1, 3} -> 2  {2, 3} -> 3  {3, 4} -> 7  Budget used: 12 |  |

**Hint**: Modify Prims’s algorithm. Until the budget is spent, connect the smallest possible edge from connected node to non-connected node.

## Modified Kruskal Algorithm

Implement Kruskal’s algorithm by keeping the **disjoint sets** in a **forest** where each node holds a **parent + children**. Thus, when two sets need to be merged, the result can be easily optimized to have two levels only: root and leaves. When two **trees are merged**, all nodes from the second (its root + root’s children) should be attached to the first tree’s root node:

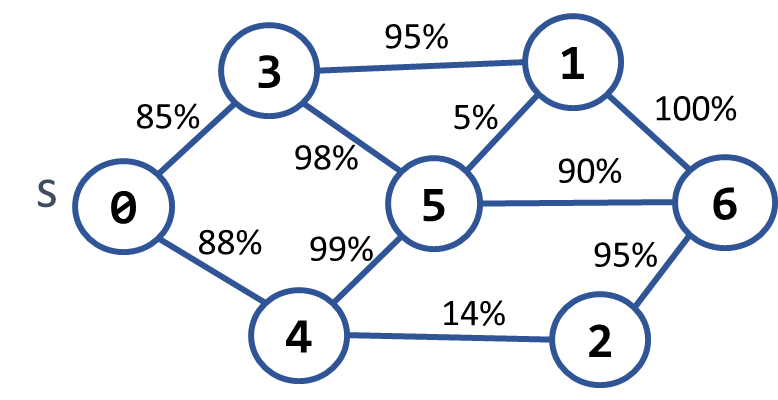
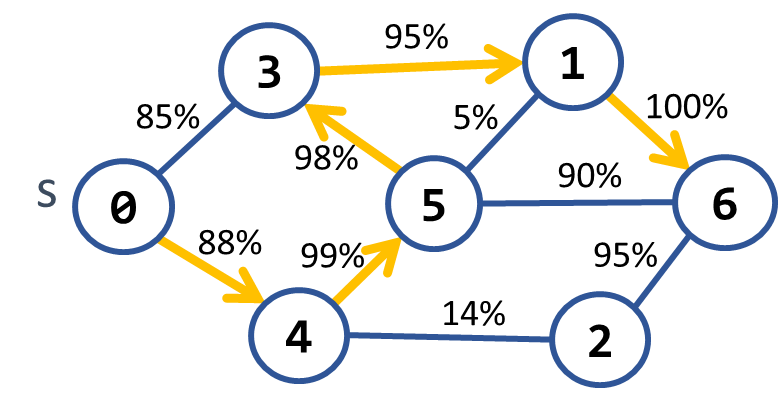


Sample input and output:

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Picture (Graph)** | **Output** | **Picture (MST)** |
| Nodes: 4  Edges: 5  0 1 9  0 3 4  3 1 6  3 2 11  1 2 5 |  | Minimum spanning forest weight: 15  (0 3) -> 4  (1 2) -> 5  (1 3) -> 6 |  |
| Nodes: 9  Edges: 15  1 4 8  4 0 6  1 7 7  4 7 10  4 8 3  7 8 4  0 8 5  8 6 9  8 3 20  0 5 4  0 3 9  6 3 8  6 2 12  5 3 2  3 2 14 |  | Minimum spanning forest weight: 45  (3 5) -> 2  (4 8) -> 3  (0 5) -> 4  (8 7) -> 4  (0 8) -> 5  (1 7) -> 7  (3 6) -> 8  (6 2) -> 12 |  |
| Nodes: 8  Edges: 16  0 1 4  0 2 5  0 3 1  1 2 8  1 3 2  2 3 3  2 4 16  2 5 9  3 4 7  3 5 14  4 5 12  4 6 22  4 7 9  5 6 6  5 7 18  6 7 15 |  | Minimum spanning forest weight: 37  (0 3) -> 1  (1 3) -> 2  (2 3) -> 3  (5 6) -> 6  (3 4) -> 7  (2 5) -> 9  (4 7) -> 9 |  |

## Most Reliable Path

We have a set of **towns** and some of them are connected by **direct paths**. Each path has a coefficient of reliability (in percentage): the chance to pass without incidents. Your goal is to compute the **most reliable path** between two given nodes. Assume all percentages will be integer numbers and round the result to the second digit after the decimal separator. For example, let’s consider the graph below:

The **most reliable path** **between 0 and 6** is shown on the right: 0 **→** 4 **→** 5 **→** 3 **→** 1 **→** 6. Its cost = 88% \* 99% \* 98% \* 95% \* 100% = **81.11%**. The table below shows the optimal reliability coefficients for all paths starting from node 0:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **v** | **0** | **1** | **2** | **3** | **4** | **5** | **6** |
| **reliability[s → d]** | 100% | 81.11% | 77.05% | 85.38% | 88% | 87.12% | 81.11% |

Sample input and output:

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Picture** |
| Nodes: 7  Path: 0 – 6  Edges: 10  0 3 85  0 4 88  3 1 95  3 5 98  4 5 99  4 2 14  5 1 5  5 6 90  1 6 100  2 6 95 | Most reliable path reliability: 81.11%  0 -> 4 -> 5 -> 3 -> 1 -> 6 |  |
| Nodes: 4  Path 0 – 1  Edges: 4  0 1 94  0 2 97  2 3 99  1 3 98 | Most reliable path reliability: 94.11%  0 -> 2 -> 3 -> 1 |  |

**Hint**: Modify Dijkstra’s algorithm.

## Shortest Paths between All Pairs of Nodes

Write a program to find the **shortest paths between all pairs of nodes** in an undirected graph. The input is given as number of nodes and list of edges. The output should be a matrix holding the shortest paths between all nodes.

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Picture** |
| Nodes: 4  Edges: 5  0 2 10  0 1 12  1 2 10  1 3 3  2 3 6 | Shortest paths matrix:  0 1 2 3  -----------  0 12 10 15  12 0 9 3  10 9 0 6  15 3 6 0 |  |
| Nodes: 10  Edges: 17  0 6 10  0 8 12  1 4 20  1 5 6  1 7 26  1 9 5  2 5 9  2 7 15  2 8 14  3 4 5  3 5 33  3 6 6  3 8 3  4 5 11  4 6 17  5 7 20  7 9 3 | Shortest paths matrix:  0 1 2 3 4 5 6 7 8 9  -----------------------------  0 37 26 15 20 31 10 41 12 42  37 0 15 22 17 6 28 8 25 5  26 15 0 17 20 9 23 15 14 18  15 22 17 0 5 16 6 30 3 27  20 17 20 5 0 11 11 25 8 22  31 6 9 16 11 0 22 14 19 11  10 28 23 6 11 22 0 36 9 33  41 8 15 30 25 14 36 0 29 3  12 25 14 3 8 19 9 29 0 30  42 5 18 27 22 11 33 3 30 0 |  |

**Hint**: Use Floyd-Warshall algorithm: <https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm>.

## \* Shortest Paths with Negative Edges

Dijkstra’s algorithm works for graphs without negative weight edges. Implement an algorithm for finding the **shortest paths** in the more common case, when some **edges have negative weights**. If the graph has negative-weight cycle, print one of the cycles. Otherwise print the shortest path length and the shortest path as sequence of nodes. Sample input and output:

|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Picture** |
| Nodes: 10  Path: 0 - 9  Edges: 19  0 3 -4  0 6 10  0 8 12  1 9 -50  2 5 12  2 7 -7  3 2 -9  3 5 15  3 6 6  3 8 -3  4 1 20  4 3 -5  5 1 -6  5 4 11  5 7 -20  6 4 17  7 1 26  7 9 3  8 2 15 | Distance [0 -> 9]: -57  Path: 0 -> 3 -> 2 -> 5 -> 1 -> 9 |  |
| Nodes: 4  Path: 0 - 3  Edges: 5  0 2 10  0 1 12  2 1 -10  1 3 3  3 2 6 | Negative cycle detected: 1 -> 3 -> 2 |  |

**Hint**: Use Bellman-Ford algorithm: <https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm>.

## \*\* Maximum Tasks Assignment

We have **L** persons and **R** tasks. **Each person can complete at most one task**. **One task can be completed by at most one person.** We have a table that shows which persons are able to complete which tasks. Find the **maximum assignment** that assigns tasks to persons in order to complete a maximum number of tasks.

Example: we have 3 people {A, B, C} and 3 tasks {1, 2, 3}. We have the following table that shows whether a person can complete a certain job.

|  |  |  |  |
| --- | --- | --- | --- |
|  | A | B | C |
| 1 | ✓ |  | ✓ |
| 2 |  | ✓ | ✓ |
| 3 | ✓ | ✓ |  |

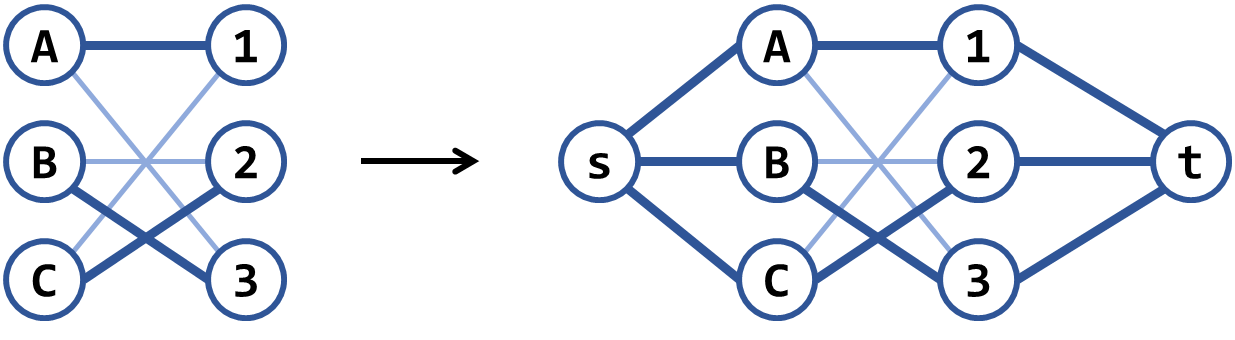
In the above table we should make the **maximal assignment**: select from each row and each column at most one checkmark value. A sample solution is shown below:

|  |  |  |  |
| --- | --- | --- | --- |
|  | A | B | C |
| 1 | **✓** |  | ✓ |
| 2 |  | ✓ | **✓** |
| 3 | ✓ | **✓** |  |

Sample input and output (assume persons will be marked by letters of the English alphabet and tasks by numbers starting from 1):

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Output** | **Table** | **Comments** |
| Persons: 3  Tasks: 3  YNY  NYY  YYN | A-1  B-3  C-2 | |  |  |  |  | | --- | --- | --- | --- | |  | A | B | C | | 1 | **✓** |  | ✓ | | 2 |  | ✓ | **✓** | | 3 | ✓ | **✓** |  | | Another correct solution:  A-3  B-2  C-1 |
| Persons: 4  Tasks: 4  YNYN  NYYN  YNYY  NNNY | A-1  B-2  C-3  D-4 | |  |  |  |  |  | | --- | --- | --- | --- | --- | |  | A | B | C | D | | 1 | **✓** |  | ✓ |  | | 2 |  | **✓** | ✓ |  | | 3 | ✓ |  | **✓** | ✓ | | 4 |  |  |  | **✓** | | Another correct solution:  A-3  B-2  C-1  D-4 |

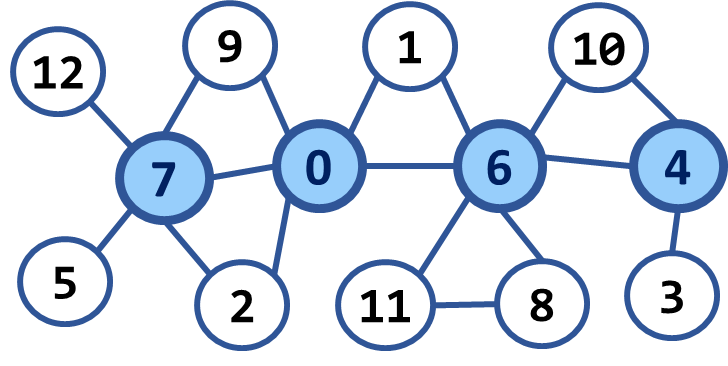
**Hint**: to solve the problem, we can model it as **bipartite graph** where the left nodes are the persons and the right nodes are the tasks and edges show who is able to complete each task. Then we can add **source** and **sink** and model the problem as **max-flow problem** as shown below (all edges have the same capacity 1):



There is another, similar **greedy algorithm**: repeat while possible: connect the nodes having the smallest number of edges, then remove all other nodes connected to these edges. Note that this algorithm works in most scenarios but is wrong in some cases. Can you find a counter-example?

## \*\* Find the Bi-Connected Components in a Graph

Finding the **articulation points** in an undirected graph is a well-known problem in computer science. A related problem (a bit harder) is to find the **bi-connected components** in a graph – its set of maximal bi-connected subgraphs. Each bi-connected component has the property that removing any of its nodes, does not break the paths between all its other nodes. Example: the below has 7 bi-connected components: {5, 7}, {12, 7}, {0, 2, 7, 9}, {1, 0, 6}, {6, 8, 11}, {4, 6, 10}, {3, 4}:

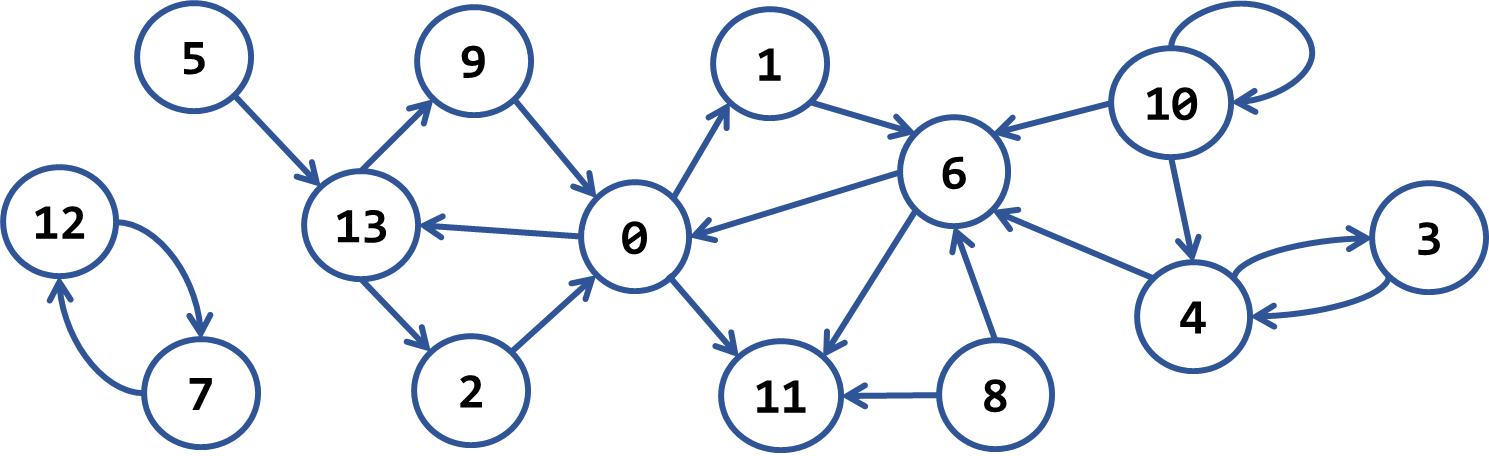


|  |  |  |
| --- | --- | --- |
| **Input** | **Output** | **Picture** |
| Nodes: 13  Edges: 17  0 1  0 2  0 6  0 7  0 9  1 6  2 7  3 4  4 6  4 10  5 7  6 8  6 10  6 11  7 9  7 12  8 11 | 5 7  12 7  0 2 7 9  1 0 6  6 8 11  4 6 10  3 4 |  |
| Nodes: 13  Edges: 20  0 1  0 2  0 6  0 7  0 9  0 11  1 6  2 7  3 4  3 8  4 6  4 10  5 7  5 12  6 8  6 10  6 11  7 9  7 12  8 11 | 12 7 5  9 0 2 7  1 6 10 4 3 8 11 0 |  |

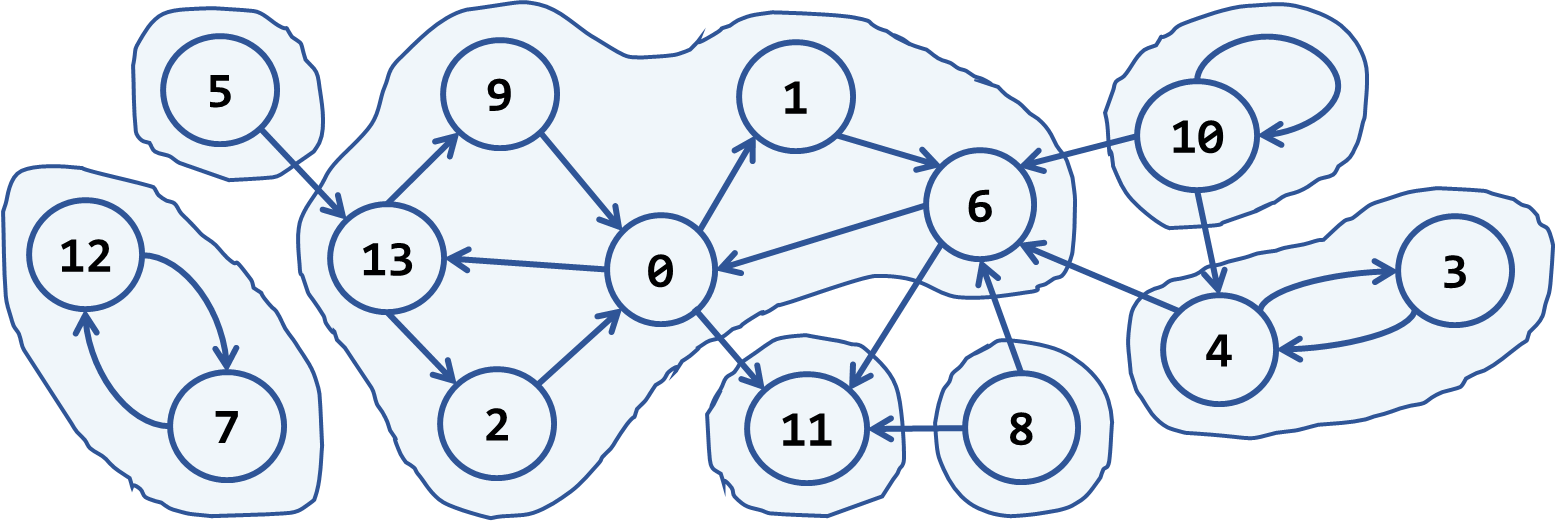
**Hint**: <http://www.cs.umd.edu/class/fall2005/cmsc451/biconcomps.pdf>

## \*\* Supplement Graph to Make It Strongly-Connected

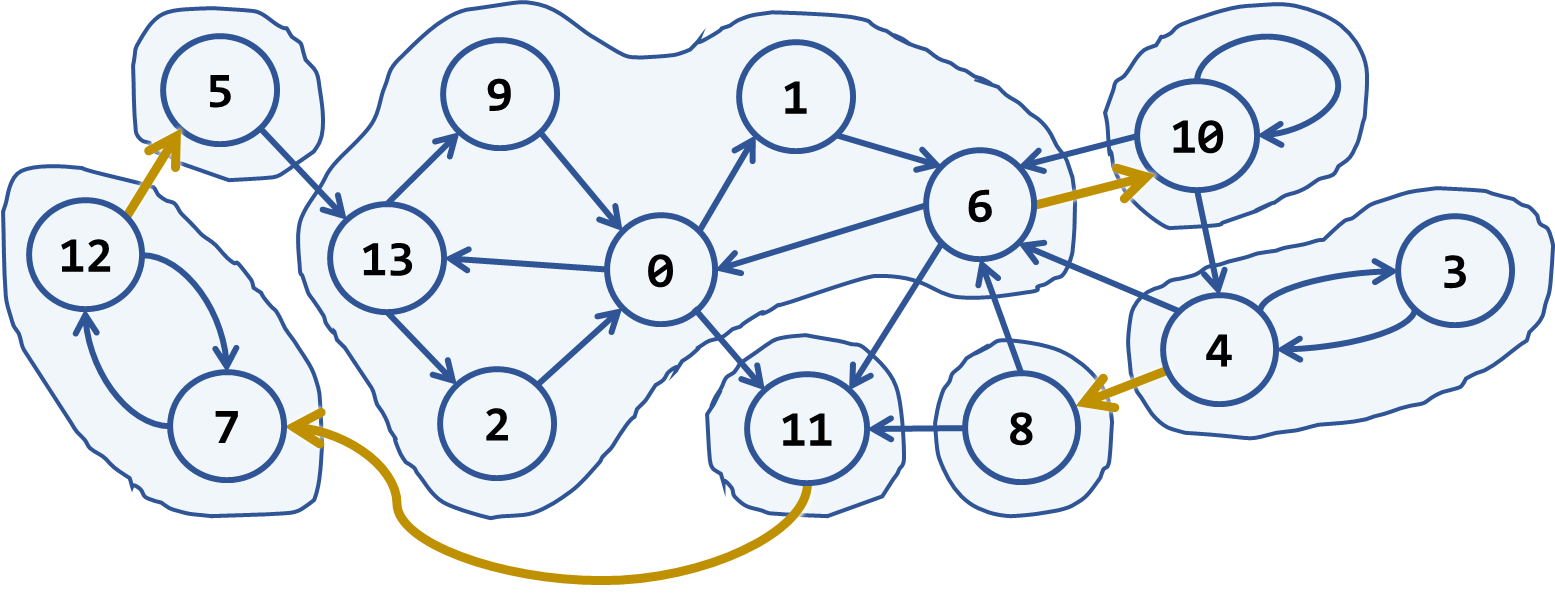
Find the minimum number of (directed) **edges to introduce into a directed graph to make it strongly connected** (from any vertex you can go to any other vertex). Also, find one configuration of edges to add that satisfies the property and reaches the minimum. For example, let’s consider the following graph:



It has the following 7 **strongly-connected components**:



To make it strongly-connected, we should connect each strongly-connected-component to some of the others. For our sample, we need to add at least **4 new edges**. A sample solution is given below:



Sample input and output:

|  |  |
| --- | --- |
| **Input** | **Output** |
| Nodes: 14  Edges: 21  12 -> 7  7 -> 12  5 -> 13  13 -> 9  13 -> 2  9 -> 0  2 -> 0  0 -> 13  0 -> 1  0 -> 11  1 -> 6  6 -> 0  6 -> 11  8 -> 11  8 -> 6  10 -> 6  10 -> 10  10 -> 4  4 -> 6  4 -> 3  3 -> 4 | New edges needed: 4  11 -> 7  12 -> 5  6 -> 10  4 -> 8 |

**Hint**: see <http://stackoverflow.com/questions/14305236/minimal-addition-to-strongly-connected-graph>.