Chapter 7

Graph Algorithms

Exercise 7.1 *(Final exam - Question 1. Network reliability. - Fall 2008)*

Consider a telecommunication directed network $G = (V, E)$ where V is the set of nodes $n = |V|$ and E is the set of directed links ($m = |E|$). Assuming you will never face more than a network failure at a time, for every pair of nodes. You want to check whether the network has always two disjoint paths. A pair of paths can be node-disjoint or link-disjoint. It is said to be node disjoint if the two paths do not share any node. It is said link disjoint of the two paths do not share any link.

a. *What is the most stringent condition: To ask for two node-disjoint paths, or to ask for two link-disjoint paths?*

b. *Built an example (i.e., a small network) where you have only one of the two conditions satisfied.*

c. *Consider a given pair of nodes: v and v'. Assume you want to find two link-disjoint paths. Show that using a max-flow algorithm between v and v', where all capacities are unit capacities, you can easily answer the question. What is the complexity of finding two link-disjoint paths?*

d. *Assume you want to find two node-disjoint paths. Show that, thanks to a transformation of the graph, it is possible to use a max-flow algorithm between v and v', where all capacities are unit capacities, in order to determine whether the network has a pair of node-disjoint paths from v to v'.*

e. *Deduce from the previous questions, whether a network is bi-connected, i.e., for each pair of nodes (v, v'), there exists a pair of two node-disjoint paths. What is the complexity of your algorithm?*

f. *If you want that one of the paths is a shortest path when searching for two link-disjoint paths, how do you need to modify your algorithm of* **c.** *? Is the complexity modified? If yes, how?*

Solution

a. The most stringent condition is node disjoint. In Figure 7.1, one can observe two link disjoint paths which are not node disjoint: v_5, v_4, v_7, v_{13} on the one hand, and v_1, v_3, v_4, v_6, and v_{12} on the other hand.

Note that any pair of node-disjoint paths is also a pair of link-disjoint paths.

b. See Figure 7.1. There are two link disjoint paths, which are not node-disjoint.

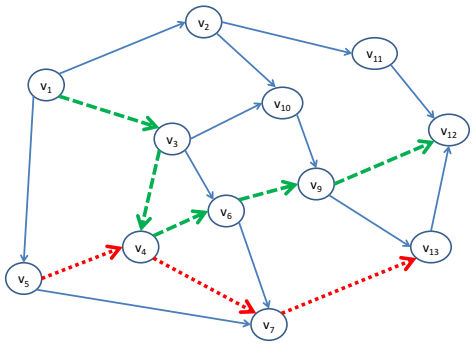


Figure 7.1: Example 1

c. We assign unit capacity on all arcs of the graph. We add two nodes, say S and D, with one arc from S to v, and one arc from v' to D, both arcs with a capacity of 2 (as we are looking for two link disjoint paths). We search for a maximum flow from S to D. The result is 2 link disjoint paths if a maximum flow of value 2 exists, see Figure 7.2 for an illustration.

Complexity is then the same than the one for computing a maximum flow between two nodes, i.e., $O(|E| \times |f^\star|)$, where $f^\star$ is the value of the maximum flow .

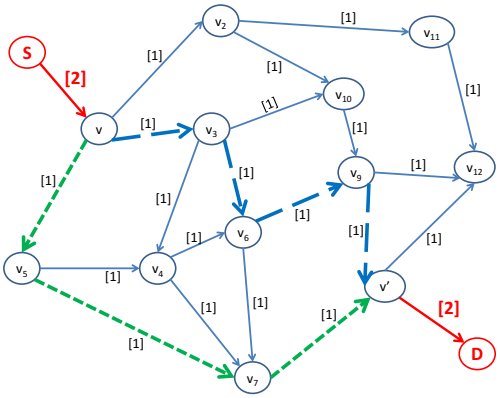


Figure 7.2: Example 2

d. We use the transformation that is described in Figure 7.3. The one unit capacity link between v^{IN} and v^{OUT} forbids going more than once through node v, and therefore if the maximum flow value is two, it leads to two node disjoint paths.

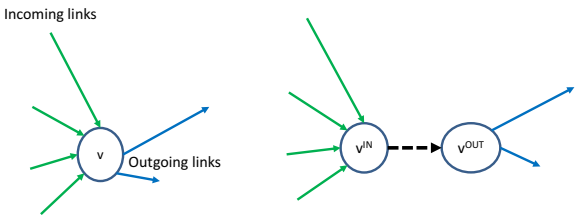


Figure 7.3: Node Transformation

e. You apply the maximum flow algorithm $O(n^2)$ times on the modified graph, as described in question **d.**, as there are $O(n^2)$ node pairs. For each node pair, the complexity is $O(|E| \times |f^\star|)$ if we use the Ford Fulkerson algorithm, where $f^\star$ is the value of the maximum flow in order to determine where there exists two node disjoint paths, using the results of the previous question. As the maximum flow is at most 2, the overall complexity is $O(n^2|E|)$.

f. We first compute a shortest path from v to v', then remove the arcs supporting the shortest path, and search for a path from v to v' on the modified graph.

Complexity: Complexity of computing a shortest path is $O(|E| \log |V|)$ if we use Dijkstra's algorithm) + complexity of checking whether a path from v to v' exists ($O(|E|)$ if we use BFS. The overall complexity is $O(|E| \log |V| + |E|)$ for a given node pair.

Complexity for all node pairs: $O(|V| \times |E| \log |V| + |V|^2 \times |E|)$.

Note that if even if the graph is biconnected, we might not be successful with such an algorithm, imposing that one of the path is a shortest path. In addition, there may be more than one shortest path for a given node pair.

Exercise 7.2 *(Final exam - Fall 2008)*

Let $G = (V, E)$ a directed graph. It is said that a flow is maximum if there is no augmenting path in the residual graph of G. However, you discover the following algorithm in a book on Operations Research where they propose the following algorithm:

1. *Look for augmenting paths in the original graph. As long as you can find an augmenting path in the original graph, iterate on the original graph, searching, at each iteration, for the augmenting path that can carry the largest possible amount of flow.*
2. *When there exists no more augmenting path on the original graph, built the residual graph, look for an augmenting path on the residual graph. If there exists none,* EXIT, *otherwise update the flows on the original graph and go back step 1.*

a. *What is the complexity of step 1.*

b. *Is the algorithm exact? Justify your answer.*

c. *Build an example showing that an algorithm that would consists only of step 1 would not be an exact algorithm.*

Solution

a. Searching whether an augmenting path exists: $O(|E|)$.
The overall complexity of step 1 is the same as the complexity of the Ford Fulkerson algorithm assuming we stop when no augmenting path exists, as in the worst case, one can reach the optimal solution using only step 1: $O(|E|f^\star)$, where $f^\star$ is the value of the maximum flow.

b. The algorithm is exact. Indeed, the second step corresponds to the Ford Fulkerson algorithm, while the first step is a heuristic.

c. Assume that we are obtained the following flow, see Figure 7.4, using the first step of the algorithm. Then, there is no way to increase the value of the flow using the first step of the algorithm.

Let us build the residual graph G_R, see Figure 7.5. G_R contains a path that links the source to the sink, therefore there is an augmenting path, and consequently the flow represented in Figure 7.4 is not maximum.

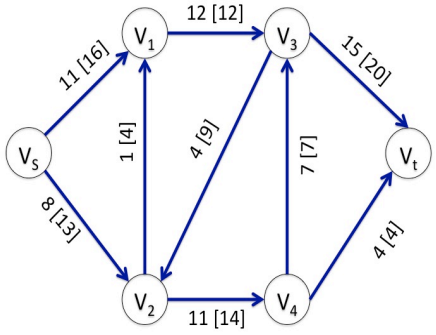


Figure 7.4: Is the flow maximum? No

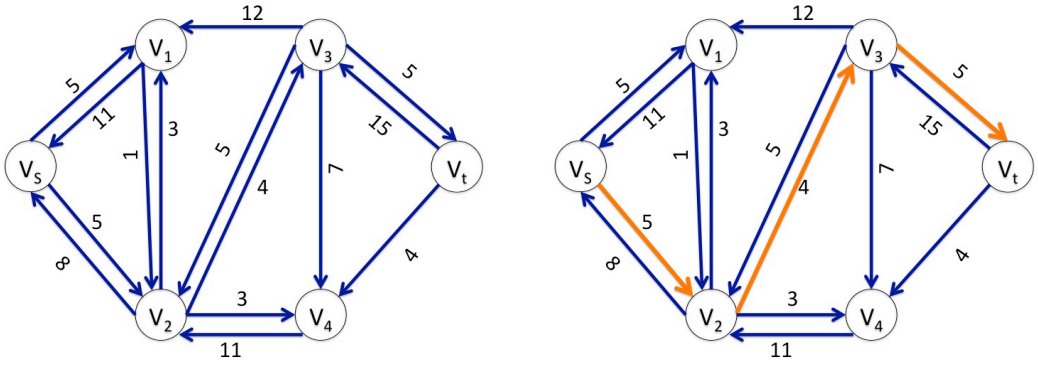


Figure 7.5: Residual Graph

Exercise 7.3 *(Final exam - Question 3 - Fall 2007)*

Let $G = (V, E)$ be a directed network with an arc length (or arc cost) c_{ij} associated with each arc $(v_i, v_j) \in E$. Answers without proper justification will not be considered.

a. *Do you agree or disagree with the following claim: "Dijktra's algorithm finds shortest paths from the source node to all other nodes in an acyclic network."*

b. *Let us now assume that G is not necessarily acyclic. Under which condition Dijkstra's algorithm remains a valid algorithm for computing the shortest path from a given source to a given destination?*

c. *What is the complexity of Dijkstra's algorithm?*

d. *Is it true that if all arcs in a network have different costs, the network has a unique shortest path tree?*

Solution

Exercise 7.4 *(Final exam - Question 4. Cormen et al. [3], Exercise 23.4, p.577. - Fall 2007)*

In this problem, we give the pseudo-code of three different algorithms. Each one takes a connected graph as input and returns a set of edges T. For each algorithm, you must either prove that T is a minimum spanning tree or prove that T is not a minimum spanning tree. Describe the most efficient implementation of each algorithm, whether or not it computes a minimum spanning tree, i.e., provide its complexity and justify it.

a. ***MAYBE - MST - A*** (G, w)

1. *sort the edges into non-increasing order of edge weights w*
2. $T \leftarrow E$
3. ***for*** *each edge $e \in E$, taken in non-increasing order by weight*
4. ***do if*** $T \setminus \{e\}$ *is a connected graph*
5. ***then*** $T \leftarrow T \setminus \{e\}$
6. ***return*** T

b. ***MAYBE - MST - B*** (G, w)

1. $T \leftarrow \emptyset$
2. ***for*** *each edge $e \in E$, taken in arbitrary order*
3. ***do if*** $T \cup \{e\}$ *has no cycle*
4. ***then*** $T \leftarrow T \cup \{e\}$
5. ***return*** T

c. ***MAYBE - MST - C*** (G, w)

1. $T \leftarrow \emptyset$
2. ***for*** *each edge $e \in E$, taken in arbitrary order*
3. ***do*** $T \leftarrow T \cup \{e\}$
4. ***if*** T *has a cycle c*
5. ***then*** *let e' be the maximum-weight edge on c*
6. $T \leftarrow T \setminus \{e'\}$
7. ***return*** T

Solution

a. We prove the algorithm correct as follows. Firstly, we argue that at the end of the algorithm T is a spanning tree. Secondly, we argue that T is minimal.

Assume to the contrary that T is not a spanning tree. If T is not a tree, then T is disconnected or contains cycles. T cannot be disconnected because line 4 of the algorithm would prevent the removal of any edge that would disconnect the graph. Therefore T cannot be disconnected. Now, assume that T contains a cycle, c. If this is true, then the graph $T \setminus \{e_c\}$ would be connected for every vertex $e_c \in c$. Once again, line 4 would ensure that e_c would be removed from T. Hence, T cannot contain any cycles. Since T is connected and acyclic, T must be a tree. Since the original T contained included every vertex in G, and the final T must be connected, T must be a spanning tree.

Now, assume to the contrary that after the algorithm, T is not a minimal spanning tree. Then, there exist edges $e \in T$ and $e' \in T$, such that the following holds:

1. For their respective weights: $w > w'$
2. e and e' are on the same cycle c in T.
3. c is the only cycle containing both e and e'.

Since $e \in T$ and $e' \in T$, and $w > w'$, we know the following two things:

1. The for-loop examines e before e' since $w > w'$
2. Lines 4-5 did not remove e, but later removed e'

This is a contradiction. Since e and e' are on cycle c, e will be removed when it is examined in lines 4-5, eliminating the cycle. Then, e' will be a bridge, and will not be removed by lines 4 and 5. This is in direct contradiction to our assumption that T is not a minimal spanning tree. Therefore, T must be a minimal spanning tree.

Complexity. $O(|E|^2)$ as Step 1 requires $O(|E \log |E|)$, Step 2 takes $O(|E|)$, Steps 3-6 requires $(O(|E|^2)$.

b. We prove this algorithm incorrect by example. Consider the following 3-vertex graph described as follows:

Graph G	Vertices:	$\{x, y, z\}$
	Edges:	$\{\{x, y\}, \{x, z\}, \{y, z\}\}$
	Weights:	$\{w\{x, y\} = 1, w\{x, z\} = 2, w\{y, z\} = 3\}$.

Let the arbitrary order of edges be as follows:

$$\{y, z\}, \{x, z\}, \{x, y\}.$$

The minimum spanning tree T is computed to be $\{\{y, z\}, \{x, z\}\}$. The sum of the edge weights is 5. However, $\{\{x, y\}, \{x, z\}\}$ is also a spanning tree with the sum of edge weights = 3. Therefore, Maybe-MST-B failed to correctly identify the minimum spanning tree.

Complexity. $O(|E|^2)$ as checking whether a cycle exists can be done in $O(|E|)$ using DFS or BFS.

c. Here, we prove the correctness of Maybe-MST-C. First, we observe that T is a forest, not a tree. A tree is a connected acyclic graph. A forest is a connected or disconnected acyclic graph. Hence, every tree is a forest, but every forest is not a tree.

The loop invariant is as follows:

After the ith iteration of the for-loop, T is a minimum spanning forest over the subgraph $S^i = (V^i, E^i)$. V^i is the set of all vertices adjacent to at least one edge in $\{e_1, e_2, \ldots, e_i\}$. $E^i \subseteq \{e_1, e_2, \ldots, e_i\}$ as some edges may have been removed if the addition of a new edge leads to some cycle (see below the details of Case 3.

1. **Initialization.** Since no edge has been considered, $V^1 = \{\}, E^1 = \{\}$.
2. **Maintenance.** Assume the loop invariant holds for the first k iterations of the for-loop. Consider the subgraph $S^k = (V^k, E^k)$. Now, consider edge $e_{k+1} = \{u, v\}$. We must consider 3 cases:

   Case 1 If $u \notin V^k$ and $v \notin V^k$, then $C = (\{u, v\}, \{e_{k+1}\})$ forms its own connected component. Since u and v have not been explored before, $T_C = \{e_{k+1}\}$ is a minimum spanning tree over C. By the induction hypothesis, S^k is a minimum spanning forest. So $S^{k+1} = (V^{k+1}, E^{k+1})$, where $V^{k+1} = V^k \cup \{u, v\}$ and $E^{k+1} = E^k \cup \{e^{k+1}\}$, is a minimum spanning forest.

   Case 2 If, $u \in V^k$ and $v \notin V^k$, or $u \notin V^k$ and $v \in V^k$, then adding edge e^{k+1} to T cannot create a cycle in T. Without loss of generality, suppose $u \in V^k$ and $v \notin V^k$. Then, $S^{k+1} = (V^{k+1}, E^{k+1})$, where $V^{k+1} = V^k \cup \{v\}$ and $E^{k+1} = E^k \cup \{e_{k+1}\}$. We discovered vertex v when we examined edge e_{k+1}. By the induction hypothesis, S^k is a minimum spanning forest. Therefore, S^{k+1} is a minimum spanning forest since e_{k+1} is the only edge in E^{k+1} that is adjacent to v.

   Case 3 If $u \in V^k$ and $v \in V^k$. Then, we need to distinguish two cases.
   Case 3a. Edge e_{k+1} connects two different forests. Nothing to be done.
   Case 3b. Edge e_{k+1} adds a cycle c to S^k.
   By the induction hypothesis, no cycle existed in S^k. Therefore, c is the only possible cycle that could result by adding just one edge to S^k. Furthermore, if we remove any edge in c, the resulting graph will become acyclic. Let e_c be the maximum weight edge in c, including the possibility that $e_c = e_{k+1}$. Then, by removing e_c, we construct minimum spanning forest S^{k+1} as follows:

$$
\begin{aligned}
V^{k+1} &= V^k \\
E^{k+1} &= E^k && \text{if } e_c = e_{k+1}; \\
E^{k+1} &= (E^k \setminus \{e_c\}) \cup \{e_{k+1}\} && \text{if } e_c \neq e_{k+1};
\end{aligned}
$$

3. **Termination.** By the induction hypothesis, after the final iteration of the for-loop, T will be a minimum spanning forest over all of the vertices in G. We argue that T is a minimum spanning tree of G if and only G is connected. Assume that T is a minimum spanning tree of G. By the definition of minimum spanning tree, T must be connected. Since T contains all of the vertices of G and some subset of the edges, G is connected. Now, assume that T is not a minimum spanning tree of G. By the loop invariant, T is a spanning forest of G. Therefore, T must be a minimum spanning forest of G. Now, assume conversely that G is connected. Then, there exists and edge between each spanning forest of T. Since each edges connects two non-adjacent spanning forests of

T, adding them to T would not introduce cycles. Since the for-loop ensures that every edge is examined, these edges would have been added to T by the correctness of the loop invariant. Therefore, T must be connected.

By showing **Initialization**, **Maintenance**, and **Termination** for the loop invariant, we establish the correctness of Maybe-MST-C.

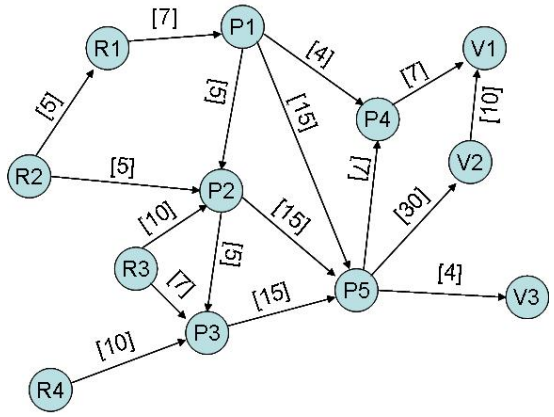
Complexity. $O(|E|^2)$ as checking whether a cycle exists can be done in $O(|E|)$ using DFS or BFS..

Exercise 7.5

Three cities V1, V2 and V3 are supplied with water through R1, R2, R3, and R4 reserves. The daily available water reserves are 15,000 cubic meters for R1, 10,000 cubic meters for R2, 15,000 for R3, and 15,000 for R4. The network distribution is pictured below, where the maximum capacity values are indicated between brackets on each arc, expressed in thousand cubic meter per day.

The three cities wish to enhance their network provisioning in order to satisfy their future needs. A study has been conducted and has identified the expected future maximum daily needs, namely:

$$V1:\ 15,000m^3 \qquad V2:\ :20,000m^3 \qquad V3:\ :15,000m^3.$$



Water Network Distribution

(a) *Recall the definition of a residual graph and of an augmenting path.*

(b) *In order to accelerate the Ford-Fulkerson's algorithm to provide the maximum flow, it is useful to search for augmenting paths in the initial graph instead of searching for them in the residual graph. At what time does it become necessary to search for augmenting paths in the residual graph?*

(c) *Determine the maximum flow value and its corresponding minimum cut. You can start by considering paths (R1, P1, P2, P3, P5, P4, V1) and (R1, P1, P4, V1) to construct the augmenting paths.*

(d) *The value of the maximum flow computed in (c) is evaluated as non-satisfactory. If the city council considers to enhance the capacity of two water pipelines (links), which ones do you suggest to consider first for a capacity increase? Determine the capacities to forecast for each of the two pipelines and the value of the new maximum flow.*

Solution (a)

After some flows have been assigned to the arcs, the residual network shows the remaining arc capacities (called residual capacities) for assigning additional flows.

Mathematically speaking the residual graph is defined as follows:

$$\begin{array}{ll} \text{Initial Graph} & \\ G = (X, A) & \text{capacities } u_{ij} \text{ on arcs } (v_i, v_j) \in A \\ \text{Residual Graph} & \\ \bar{G} = (X, \overline{A}) & \text{flow } \varphi_{ij} \text{ on arcs } (v_i, v_j) \in A \\ & 0 \leq \varphi_{ij} \leq u_{ij} \end{array}$$

$$\overline{A} = \begin{cases} (v_i, v_j): & (v_i, v_j) \in A \text{ and } \varphi_{ij} < u_{ij} \text{ residual capacity } r_{ij} = u_{ij} - \varphi_{ij} \\ (v_j, v_i): & (v_i, v_j) \in A \text{ and } \varphi_{ij} > 0 \text{ residual capacity } r_{ij} = \varphi_{ij} \end{cases}$$

An augmenting path is a directed path from the source to the sink in the residual network such that every arc on this path has a strictly positive residual capacity.

Solution (b)

We consider the residual graph when there are no more existing paths with a positive flow on each link of the path, connecting the source to the sink in the original graph We use the following algorithm:

$$\begin{cases} r_{ij} \leftarrow u_{ij} & \text{for all } (v_i, v_j) \in A \\ \varphi_{ij} \leftarrow 0 & \text{for all } (v_i, v_j) \in A \end{cases}$$

while $\exists$ paths connecting source and sink **do**
    Select a path P connecting the source and sink.
    Let φ be the minimum residual capacity of P
    $\forall (v_i, v_j) \in P \quad r_{ij} \leftarrow r_{ij} - \varphi \; ; \; \varphi_{ij} \leftarrow \varphi_{ij} + \varphi$
end while

Solution (c)

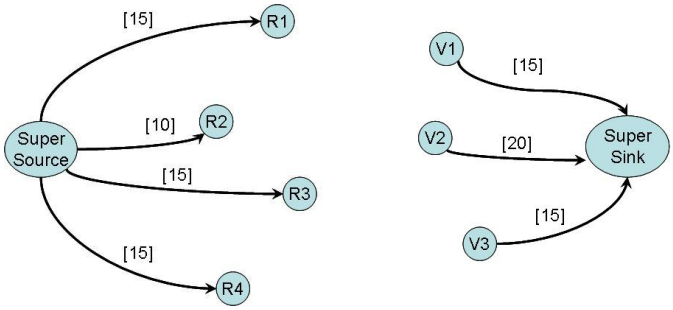


Figure 7.6: Super Source; Super Sink

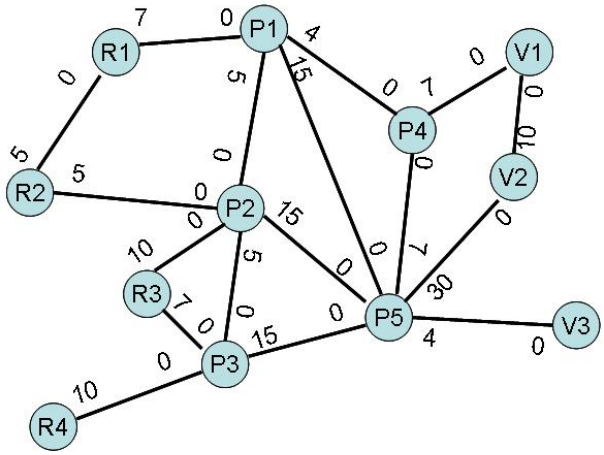


Figure 7.7: Residual Graph

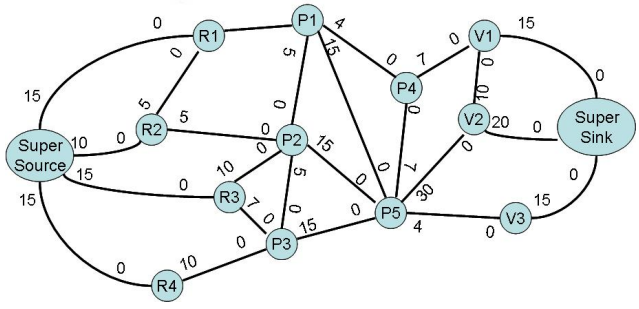


Figure 7.8: Residual Graph with Super Source and Super Sink

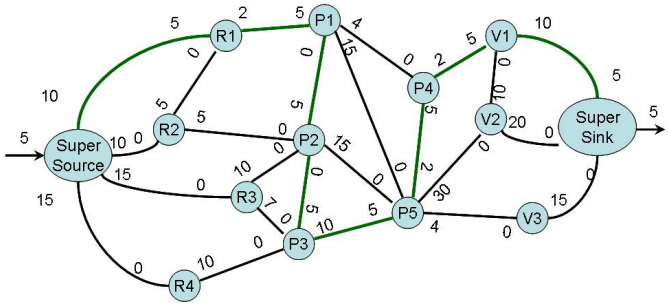


Figure 7.9: Augmenting Path: Source, $R1, P1, P2, P3, P5, P4, V1$, Sink - 5 flow units

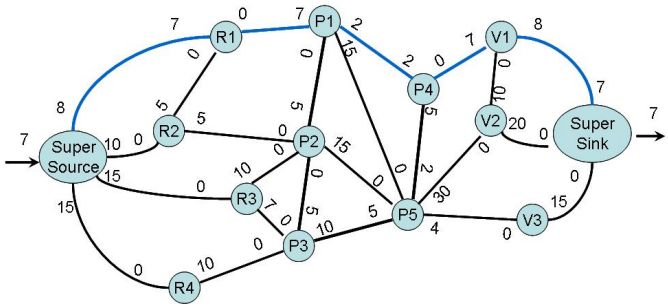


Figure 7.10: Augmenting Path: Source, $R1, P1, P4, V1$, Sink - 2 flow units

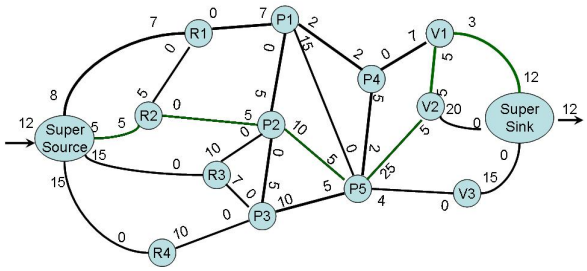


Figure 7.11: Augmenting Path: Source, $R2, P2, P5, V2, V1$, Sink - 5 flow units

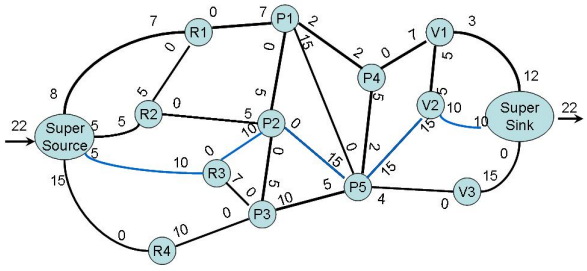


Figure 7.12: Augmenting Path: Source, $R3, P2, P5, V2$, Sink - 10 flow units

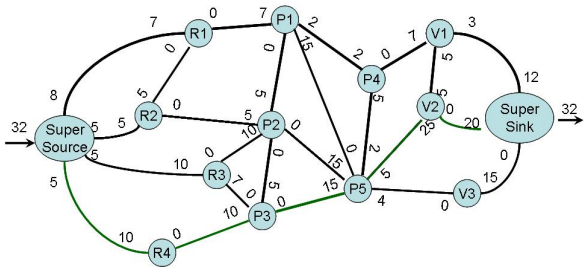


Figure 7.13: Augmenting Path: Source, $R4, P3, P5, V2$, Sink - 10 flow units

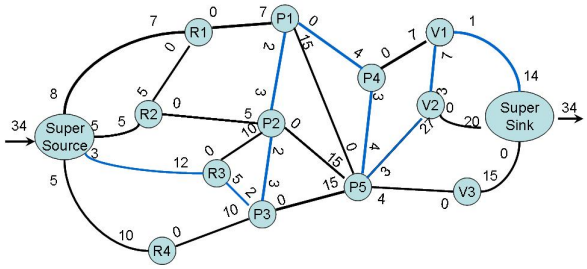


Figure 7.14: Augmenting Path: Source, $R3, P3, P2, P1, P4, P5, V2, V1$, Sink - 2 flow units

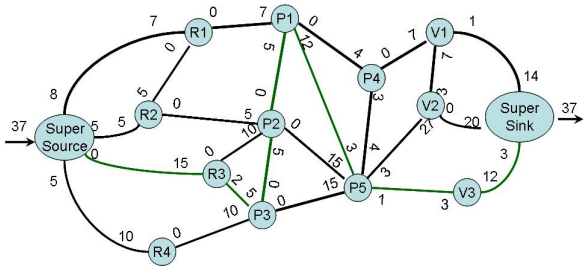


Figure 7.15: Augmenting Path: Source, $R3, P3, P2, P1, P5, V3$, Sink - 3 flow units

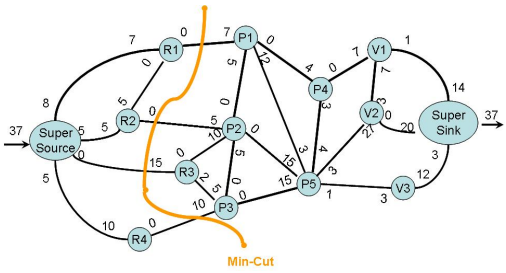


Figure 7.16: Min cut found with capacity 0 in the residual graph: Max Flow 37

Solution (d)

One of the selected link needs to belong to the minimum cut found in the previous question, as otherwise, independently of the link choice, no overall flow increase will be possible. Consider increasing the capacity of link (R1, P1). Looking at the incoming links of node R1, we can increase the incoming flow up to 5+8 = 13. Let us now check if by increasing only one more capacity, we can push 13 more units to the sink node. Looking at the incoming link of the super sink, two links are not fully capacitated. In order to increase the flow value on those links, especially on (V3, super sink), the bottleneck link is (P5, V3). Therefore, second selected link is that last link. However, we can easily check that we cannot push more than 12 units of flow if we are entitled to increase the capacity of only one additional link. Consequently, results are as described in Figure 7.17.

In order to check that our intuition is correct, we need to re-apply Ford-Fulkerson with the new transport capacities, and double check that the resulting maximum flow is now 37 + 12 = 49. That is what is done in Figures 7.18 to 7.21.

Similar reasoning can be done with the other links of the cut, and identifying the second link with added capacity, so that the resulting new maximum flow is the largest possible one.

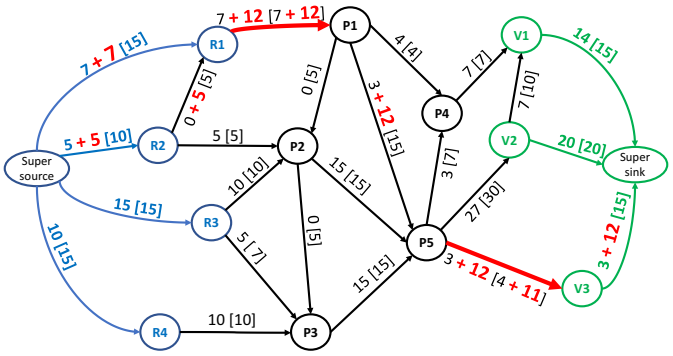


Figure 7.17: The two selected pipelines

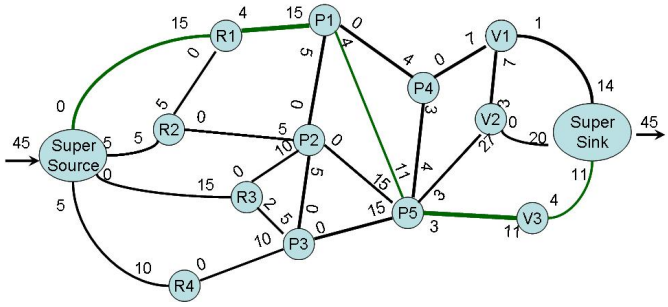


Figure 7.18: Augmenting path $\rightsquigarrow$ + 8 flow units: Super source, R1, P1, P5, V3, Sink

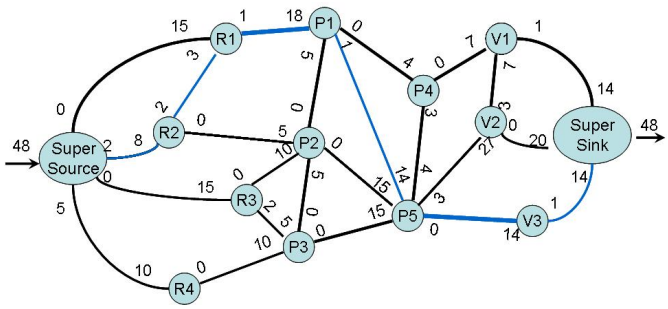


Figure 7.19: Augmenting path $\rightsquigarrow$ + 3 flow units: Super source, R2, R1, P1, P5, V3, Super sink

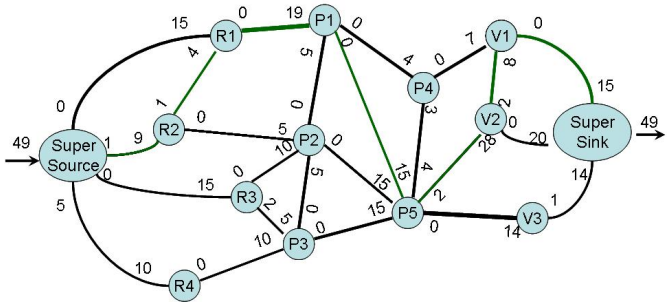


Figure 7.20: Augmenting path $\rightsquigarrow$ + 1 flow unit: Super source, R2, R1, P1, P5, V2, V1,Super sink

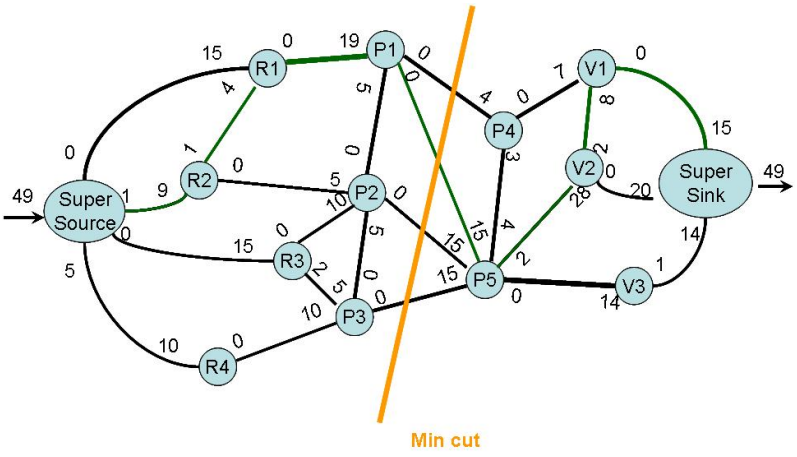


Figure 7.21: Min cut found with capacity 0 in the residual graph: Max Flow 49

Exercise 7.6

Let $G = (V, E)$ be a directed graph with nodes $v_1 ... v_n$. We say that G is an ordered graph if it has the following properties.

- *(i) Each edge goes from a node with a lower index to a node with a higher index. That is, every directed edge has the form (v_i, v_j) with $i < j$.*
- *(ii) Each node except v_n has at least one edge leaving it. That is, for every node v_i, $i = 1, 2, ..., n-1$, there is at least one edge of the form (v_i, v_j).*

The length of a path is the number of edges in it. The goal in this question is to solve the following problem (see Figure for an example). Given an ordered graph G, find the length of the longest path that begins at v_1 and ends v_n.

(a) Show that the following algorithm does not correctly solve this problem, by giving an example of an ordered graph on which it does not return the correct answer.

———————————————————————————

Set $w = v_1$
Set $L = 0$
While *there is an edge out of the node* w
{
Choose the edge (w, v_i) *for which*
j *is as small as possible*
Set $w = v_j$
Increase L *by* 1
}
end While
Return L *as the length of the longest path.*
———————————————————————————-

In your example, say what the correct answer is and also what the algorithm above finds.

(b) Give an efficient algorithm that takes an ordered graph G and returns the length of the longest path that begins at v_1 and ends at v_n. (Again, the length of a path is the number of edges in the path.)

- *(i) Show that the following algorithm does not correctly solve this problem, by giving an example of an ordered graph on which it does not return the correct answer.*
- *(ii) Give an efficient algorithm that takes an ordered graph G and returns the length of the longest path that begins at v_1 and ends at v_n. (Again, the length of a path is the number of edges in the path.)*

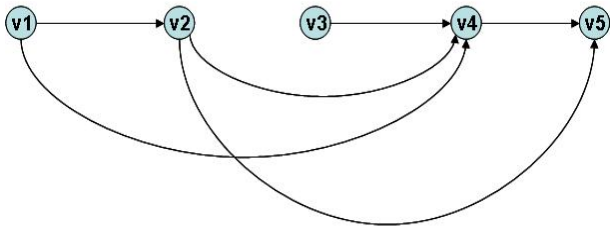


Figure 7.22: The correct answer for this ordered graph is 3: The longest path from v_1 to v_n uses the three edges (v_1, v_2), (v_2, v_4), and (v_4, v_5).

Solution

Exercise 7.7 *(Final exam - Question 6. Cormen et al. [3], Exercise 24.3-2, p.600 - Fall 2007)*

Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers.

Solution

If the graph has a negative cycle (i.e., a cycle such that the sum of the weight of its edges is negative), then the problem of finding a shortest is undefined. This does not provide a proper example to illustrate that Dijkstra's algorithm produces incorrect answers when a graph has negative-weight edges.

Let us consider an example of a graph, in Figure 7.23, with no negative cycle, but still with some negative-weight edges.
Look for a shortest path from v_1 (source) to v_4 (destination).
Red labels are temporary labels. Blue labels are permanent labels: note that Dijkstra's algorithm assumes that label values are increasing, and as soon as a label becomes permanent, it is never modify.

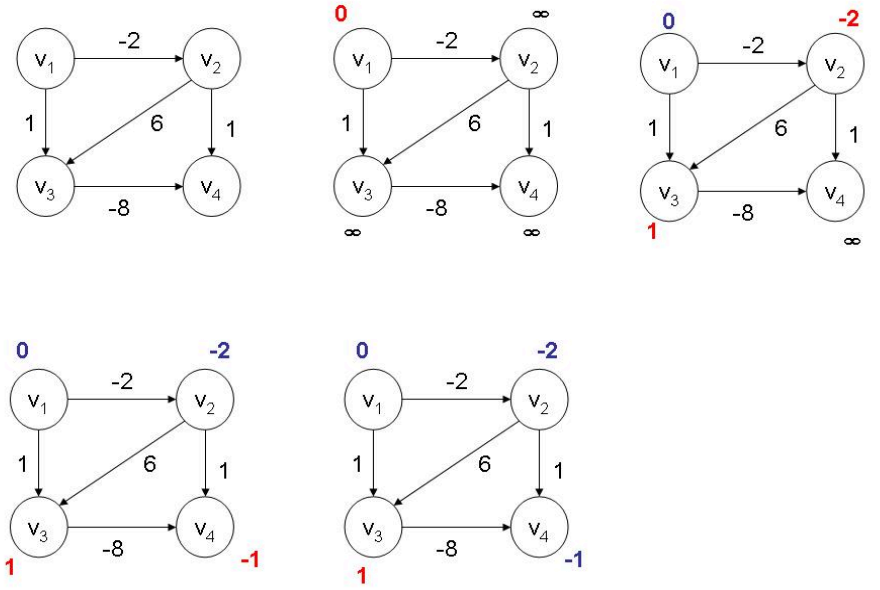


Figure 7.23: Example of a graph with negative-weight edges, but with no negative cycle

Exercise 7.8 *(MidTerm exam - Fall 2010 - Exercise 7 p. 417 - Book of Kleinberg and Tardos [11])*

Consider a set of mobile computing clients in a given town where each client needs to be connected to one of several possible base stations. We suppose there are n clients, with the position of each client specified by its (x, y) coordinates in the plane. There are also k base stations; the position of each of these is specified by (x, y) coordinates as well.

For each client, we wish to connect it to exactly one of the base stations. Our choice of connections is constrained in the following ways. There is a range parameter r - a client can only be connected to a base station that is within distance r. There is also a load parameter t - no more than t clients can be connected to any single base station.

Your goal is to design a polynomial-time algorithm for the following problem. Given the positions of a set of clients and a set of base stations, as well as the range and load parameters, decide whether every client can be connected simultaneously to a base station, subject to the range and load conditions in the previous paragraph.

Solution

The problem can be reduced to a maximum flow problem using the following graph, where there is a node associated with each base station (BS), with each client, and two extra nodes (one for the source, one for the sink). There is a link from the source to each client with capacity one, as each client can chose only one base station. There is a link from a client to a base station if the distance between them is less than r. There is a link from each base station to the sink with capacity t as a given base station can serve at most t clients.

If the maximum flow is equal to n, i.e., the number of clients, then there is a coverage solution which allows serving all clients.

a. Construction of the graph. Three sets of links.

- Between the source and the clients $\hookrightarrow \quad O(n)$
- Between the clients and the base stations $\hookrightarrow \quad O(nk)$
- Between the base stations and the sink $\hookrightarrow \quad O(k)$
- Overall: $O(n) + O(nk) + O(k) = O(nk)$

b. Ford Fulkerson's algorithm: $O(\#$ of links $\times$ upper bound on the maximum flow$) = O(nk \times n) = O(n^2 \times k)$.

c. Overall complexity: $O(nk) + O(n^2k) = O(n^2k)$.

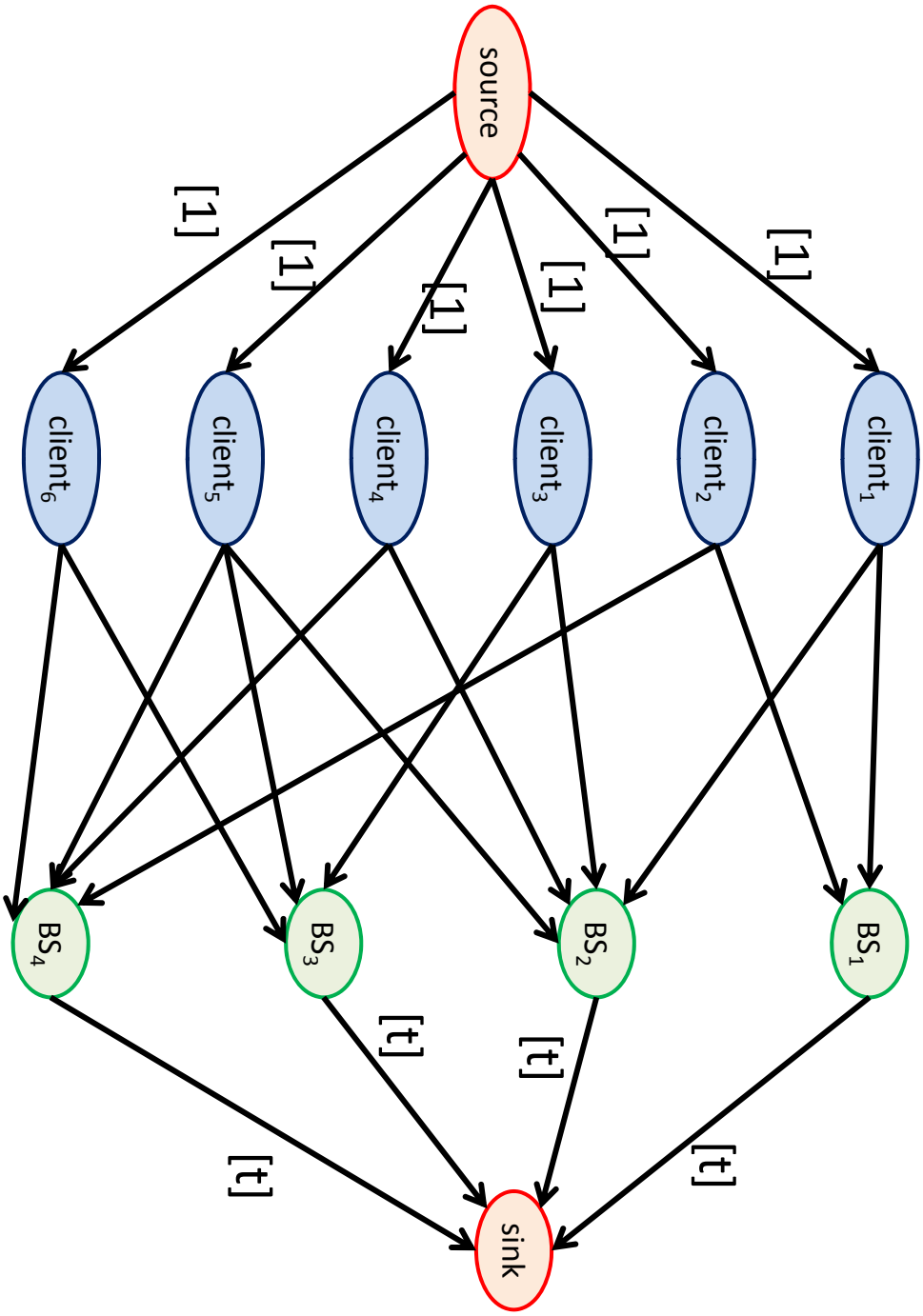


Figure 7.24: Graph Model

Exercise 7.9

Let $G = (V, E)$ be a flow network with source s and sink t, and suppose that each link $e \in E$ has capacity $c(e) = 1$. Assume also, for convenience, that $|E| = \Omega(|V|)$.

(i) Suppose that we implement the Ford-Fulkerson maximum-flow algorithm by using depth-first search to find augmenting paths in the residual graph. What is the worst-case running time of this algorithm on G?

(ii) Suppose that a maximum flow for G has been computed using Ford-Fulkerson, and a new link with unit capacity is added to E. Describe how the maximum flow can be efficiently updated. (Note: It is not the value of the flow that must be updated, but the flow itself.) Analyze the complexity of your algorithm.

(iii) Suppose that a maximum flow for G has been computed using Ford-Fulkerson, but a link is now removed from E. Describe how the maximum flow can be efficiently updated. Analyze the complexity of your algorithm.

Solution

(i) The running time is: $O(|V| \times |E|)$. Indeed, the running time is $O(f^\star \times |E|)$, where $f^\star$ is the value of the maximum flow. However, $f^\star = O(|V|)$ because the flow is bounded by the capacity of any cut, and the capacity of the cut $(\{s\}, V \setminus \{s\})$ is $O(|V|)$ because there are $O(|V|)$ edges leaving s, and every edge has capacity equal to 1.

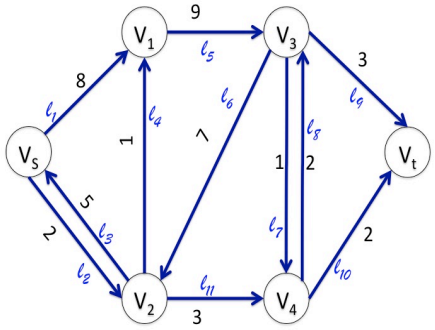
(ii) Just execute one more iteration of the Ford-Fulkerson algorithm. The new edge in E adds a new edge to the residual graph, so look for an augmenting path and update the flow if a path is found. The time is $O(|V| + |E|) = O(|E|)$ if a path is found with depth-first or breadth-first search. Only 1 iteration is needed because adding an edge of unit capacity increases the capacity of the min cut by at most 1. Thus the maximum flow (which is equal to the min cut capacity) increases by at most 1. Since all edge capacities are 1, any augmentation increases flow by 1, so only one augmentation is needed.

(iii) Denote by (u, v) the removed edge. If (u, v) has no flow, we do not need to do anything. If (u, v) has flow, the network flow must be updated. Indeed, there is now 1 unit of flow coming into u than is going out (and 1 more unit going out of v than is coming in). The idea is to try to reroute this unit of flow so that it goes out of u and into v via some other path. If that is not possible, we must reduce the flow from s to u and from v to t by 1 unit.
So look for an augmenting path from u to v. (Note: not from s to t) If there is such a path, augment the flow along that path. If there is no such path, reduce the flow from s to u by augmenting the flow from u to s. That is, find an augmenting path $u \rightsquigarrow s$ and augment the flow along that path. (There definitely is such a path, because there is flow from s to u.) Similarly augment the path from t to v. The time is: $O(|V| + |E|) = O(|E|)$ if finding path with DFS or BFS.

Exercise 7.10

(i) Let $G = (V, E)$ be a directed graph. Recall the algorithm of a DFS (Depth First Search) algorithm. You need to write a pseudo-code.

*(ii) Apply a DFS algorithm on the following residual graph in order to determine whether there exists a path between the source and the sink node. Indicate the order in which you go through the links. Indices of $\ell_1, \ell_2, \ldots$ are the link indices. Assume that the graph is represented by its adjacency lists, and that links are in the order of their **decreasing** indices.*



(iii) What is the complexity of a DFS algorithm? Describe which data structure to use in order to implement a DFS efficiently. Illustrate the use of that data structure for the path search of the previous question.

(iv) Let $G = (V, E)$ be a directed graph. Recall the algorithm of a BFS (Breadth First Search) algorithm. You need to write a pseudo-code.

*(v) Apply a BFS algorithm on the following residual graph in order to determine whether there exists a path between the source and the sink node. Indicate the order in which you go through the links. Indices of $\ell_1, \ell_2, \ldots$ are the link indices. Assume that the graph is represented by its adjacency lists, and that links are in the order of their **increasing** indices.*

(vi) What is the complexity of a BFS algorithm? Describe which data structure to use in order to implement a BFS efficiently. Illustrate the use of that data structure for the path search of the previous question.

(vii) In the Ford-Fulkerson algorithm, at which stage do you need to use a DFS or a BFS algorithm? Does it make a difference whether you use a BFS or a DFS algorithm on either the efficiency (computational times in practice) or the complexity?

Solution (i)

Input: A directed graph $G = (V, L)$ and a vertex $v \in V$

Output: A labeling of the links (discovery vs. back)

procedure DFS(G, v)

1. `label` v `as explored`
2. **for** `each link` $\ell \in$ ADJ(v) **do**
3. `let` $\ell = (u, v)$
4. **if** `vertex` v `is unexplored` **then**
5. `label` ℓ `as a discovery edge`
6. `recursively call` DFS(G, v)
7. **else**
8. `label` ℓ `as a back link`

Solution (ii)

Solution (v)

1. procedure BFS(G, v_s)
2. **for all** $v \in V[G]$ **do**
3. `visited`$[v] \leftarrow$ `false`
4. $Q \leftarrow$ `EmptyQueue`
5. `visited`$[s] \leftarrow$ `true`
6. `Enqueue`(Q, v_s)
7. **while** `not Empty`(Q) **do**
8. $u \leftarrow$ `Dequeue`(Q)
9. **for all** $w \in$ ADJ$[u]$ **do**
10. **if** `not visited`$[w]$ **then**
11. `visited`$[w] \leftarrow$ `true`
12. `Enqueue`(Q, w)

The size of Q is never larger than the number of nodes (n). The running time of BFS is $O(m+n)$.

Solution (iv)

$$\ell_1, \ell_2, \ell_5, \ell_3, \ell_4, \ell_{11}, \ell_6, \ell_7, \ell_9$$

Then, a path is found from v_s to v_t.

Solution (vi)

The running time of BFS is $O(m+n)$.

Data Structure: Queue.

Initialization. Queue is empty, all nodes are univisited

Step 1.

Enqueue	→	v_s	→	Dequeue

`visited`[v_s]=1

Step 2. Dequeue v_1.
Enqueue (in this order) v_1, v_2

Enqueue	→	v_2	v_1	→	Dequeue

`visited`[v_1]=1
`visited`[v_2]=1

Step 3. Dequeue v_1.
Enqueue v_3

Enqueue	→	v_3	v_2	→	Dequeue

`visited`[v_3]=1

Step 4. Dequeue v_2.
Enqueue v_4
Enqueue v_t

Enqueue	→	v_t	v_4	v_3	→	Dequeue

visited[v_4]=1
visited[v_t]=1
$\rightsquigarrow$ we have found a path from v_s to v_t

Solution (vii)

We need DFS/BFS in order to check where there exists one path from v_s to v_t. Due to the definition of the reduced graph, if there exists a path, then it is an augmented with at least one unique of flow that can go from v_s to v_t.

Since there is no strategy in order to select the order in which we examine the nodes/links, then DFS or BFS is equivalent in terms of performance.

Exercise 7.11 *Midterm - Fall 2018*

Let $G = (V, L)$ be a directed graph with edge capacities $\text{CAP} : L \to \mathbb{Z}^+$, a source vertex v_s, and a target vertex v_t. Suppose someone hands you an arbitrary function $\varphi : L \to \mathbb{Z}$. Describe and analyze the complexity of fast and simple algorithms to answer the following questions:

(i) Is φ a feasible (s,t)-flow in G?

(ii) Is φ a maximum (s,t)-flow in G?

(iii) Is φ the unique maximum (s,t)-flow in G?

Solution (i)

A feasible flow on G is a function $\varphi : V \times V \to \mathbb{R}$ satisfying the following:

- Capacity constraint: For all $\ell = (u, v) \in L$, $\quad 0 \leq \varphi_\ell \leq \text{CAP}_\ell$.
- Flow conservation: For all $u \in V \setminus \{v_s, v_t\}$, $\quad \sum_{v \in V} \varphi(u, v) = \sum_{v \in V} \varphi(v, u)$.

Pay attention that in the statement of the exercise, φ is defined with values in $\mathbb{R}$, and not in $\mathbb{R}^+$, therefore one has to check that the flow value is ≥ 0.

Algorithm for checking capacity constraints:
SUCCESS $\leftarrow$ TRUE
$\ell \to 1$
While $\ell \leq |L|$ and SUCCESS = TRUE **do**
    if $\varphi_\ell < 0$ or $\varphi_\ell > \text{CAP}_\ell$ then SUCCESS $\leftarrow$ FALSE
    $\ell \leftarrow \ell + 1$
EndWhile
If SUCCESS $\leftarrow$ TRUE **then** flow is feasible with respect to capacity constraints
else flow is not feasible with respect to capacity constraints
endif

Algorithm for checking flow conservation:
Assume $V \setminus \{v_s, v_t\}$ (i.e., set of nodes except for the source and sink nodes) is equal to $\{v_1, v_2, \ldots, v_n\}$
SUCCESS $\leftarrow$ TRUE
$v \to v_1$
While ($v = v_i$ is such that $i \leq n$) and (SUCCESS = TRUE) **do**
    if $\sum_{u \in V} \varphi(u, v) \neq \sum_{u \in V} \varphi(v, u)$ then SUCCESS $\leftarrow$ FALSE
    $v \leftarrow$ next node in $\{v_1, v_2, \ldots, v_n\}$

EndFor
If SUCCESS ← TRUE **then** flow is feasible with respect to flow conservation constraints
else flow is not feasible with respect to flow conservation constraints
endif

A flow is feasible if it is feasible with respect to both sets of constraints: capacity and flow conservation.

Overall complexity: $O(|L|)$.

Note that **overall** complexity of checking flow conservation entails going through each link twice + going through each node, hence $O(|V|)$, and not $O(|V|^2)$ with the reasoning that a node has at most $|V|$ incoming/outgoing links. Such a complexity analysis can be viewed as an **amortized complexity** (aggregated method).

Solution (ii)

Check whether there exists one augmented path in the reduced graph G_R. In order to perform such a check, use DFS or BFS.

In an exam: provide the definition of G_R + the description of DFS (or BFS).

Overall complexity: $O(|L|)$, i.e., complexity of DFS (we assume here that the graph is connected, hence $|L| \geq |V| - 1$ and therefore, no need to write $O(|L| + O(|V|)$.

Solution (iii)

Run FF (Ford Fulkerson) once and save the value of the flow $\text{VALUE}(\varphi^\star)$ and the flow over all links $\varphi(\ell)$ for each link ℓ with $\varphi(\ell) > 0$.

Set capacity (in this iteration) of link ℓ to: $\text{CAP}(\ell) = \varphi(\ell) - 1$ and run FF. If the value of the flow is the same as in the original graph, then there exists another way to push the max flow through the network and we are done - the max flow is not unique. Hence, we return "not unique". Otherwise we continue.

Assume we are done with looping without finding another max flow of same value, that means max flow is unique $\rightsquigarrow$ return "unique"

Sufficiency. If the algorithm finds a different flow with the same total result, then obviously the new flow is legal, and, also, necessarily, at least one of the links is flowing a different amount than it did before.

Necessity. Suppose there is a different flow $\tilde{\varphi}$ than the original one $\varphi^\star$ (with the same total value: $\text{VALUE}(\varphi^\star) = \text{VALUE}(\tilde{\varphi})$), with at least one of the edges flowing a different amount. Say that, for each link, the flow in the alternative solution is not less than the flow in the original solution. Since

the flows are different, there must be at least a single link where the flow in the alternative solution increased. Without a different link decreasing the flow, though, there is either a violation of the conservation of flow, or the original solution was suboptimal. Hence, there is some link ℓ where the flow in the alternative solution is lower than in the original solution. Since it is an integer flow network, the flow must be at least 1 lower on ℓ. By definition, though, reducing the capacity of ℓ to at least 1 lower than the current flow, will not make the alternative flow illegal. Hence, some alternative flow must be found if the capacity is decreased for ℓ.

Complexity. $O(|L|^2 \times \varphi^\star)$.

Be aware that unique maximum flow is not equivalent to unique min cut, see an example in Figure 7.25. There is a max-flow/min-cut equal to one starting in v_1, ending in v_5. While there is a single minimum cut ($\{v_1\}$ and $\{v_2, v_3, v_4, v_5\}$), there are a lot of possible maximum flows (take the unit of flow entering v_2 and distribute it between v_3 and v_4 as you like).

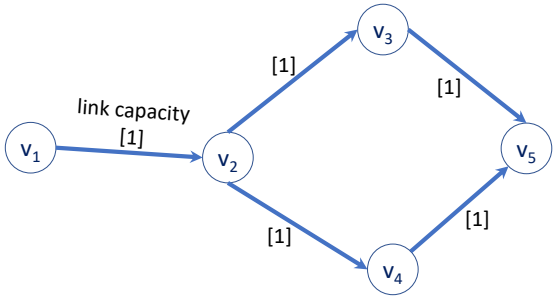
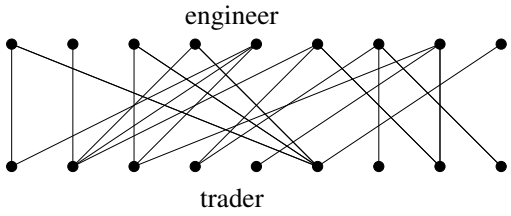


Figure 7.25: Unique Maximum Flow vs. Unique Min Cut

Exercise 7.12 *Matching Problem*

A managing director has to launch the marketing of a new product. Several candidate products are at his disposal and he has to choose the best one. Hence, he let each of these products be analyzed by a team made of an engineer and a trader who write a review together. The teams are made along the following graph; each edge corresponds to a product and its end vertices to the engineer and trader examining it.



How many people at least does the managing director gather in order to have the report on all the products? (The report can be given by either the engineer or the trader.)

Solution

Exercise 7.13 *Matching Problem*

The French figure skating federation wants to form couples (one woman and one man) for dance on ice in order to prepare the Olympic Winter Games. Six women and six men of high enough level volunteer. In view of temper incompatibility between some women and some men as well as estethic criteria supposedly displeasing the judges (the federation does not have enough money to corrupt all judges), the following table has been designed. A cross in a square means that the two ice skaters cannot form a couple.

	Woman 1	*Woman 2*	*Woman 3*	*Woman 4*	*Woman 5*	*Woman 6*
Man 1	×	×		×	×	
Man 2						×
Man 3	×			×	×	×
Man 4	×				×	×
Man 5		×	×			
Man 6	×		×	×	×	

How many couples at most can the federation form? Justify your answer

Solution

Exercise 7.14 *2-satisfiability*

Given a boolean formula $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ in conjunctive normal form with m clauses and n literals $x_1, x_2, \ldots, x_n$ such that each clause has exactly two literals, find a satisfying assignment (if one exists).

Solution Sketch

Form the implication digraph with $2n$ vertices (one per literal and its negation). For each clause $x \vee y$, include edges from $\neg y = \overline{y}$ to x and from $\neg x = \overline{x}$ to y.

Claim. The formula is satisfiable if and only if no variable x is in the same strong component as its negation $\neg x$. Moreover, a topological sort of the kernel DAG (contract each strong component to a single vertex) yields a satisfying assignment.

Exercise 7.15 *Shortest directed cycle.*

Given a digraph, design an algorithm to find a directed cycle with the minimum number of edges (or report that the graph is acyclic). The running time of your algorithm should be proportional to $|E| \times |V|$ in the worst case.

Application: *Consider a set of patients in need of kidney transplants, where each patient has a family member willing to donate a kidney, but of the wrong type. Each patient is willing to donate to another person provided their family member gets a kidney. Then hospital performs a "domino surgery" where all transplants are done simultaneously.*

Solution Sketch

Run BFS from each vertex s. The shortest cycle through s is an edge $v \to s$, plus a shortest path from s to v.

Exercise 7.16 *Nesting boxes*

A d-dimensional box with dimensions $(a_1, a_2, \ldots, a_d)$ *nests inside a box with dimensions* $(b_1, b_2, \ldots, b_d)$ *if the coordinates of the second box can be permuted so that* $a_1 < b_1, a_2 < b_2, \ldots, a_d < b_d$.

(a) *Give an efficient algorithm for determining where one d-dimensional box nests inside another.*

(b) *Show that nesting is transitive: if box i nests inside box j and box j nests inside box k, then box i nests inside box k.*

(b) *Given a set of n d-dimensional boxes, given an efficient algorithm to find the most boxes that can be simultaneously nested.*

Hint: Create a digraph with an edge from box i to box j if box i nests inside box j. Then run topological sort.

Solution

Exercise 7.17 *Nesting boxes*

Tamarindo University and many other schools worldwide are doing a joint project on multimedia. A computer network is built to connect these schools using communication links that form a tree. The schools decide to install a file server at one of the schools to share data among all the schools. Since the transmission time on a link is dominated by the link setup and synchronization, the cost of a data transfer is proportional to the number of links used. Hence, it is desirable to choose a "central" location for the file server. Given a tree T and a node v of T, the eccentricity of v is the length of a longest path from v to any other node of T. A node of T with minimum eccentricity is called a center of T.

1. *Design an efficient algorithm that, given an n-node tree T, computes a center of T. Provide a detailed description of the algorithm , its complexity and the justification of its complexity.*
2. *Is the center unique? If not, how many distinct centers can a tree have?*

Solution

It can be show that the center is on the diameter, which is the longest path in the tree. Use one DFS to find the diameter and then find the center of this path.

The diameter of a tree is the number of nodes on the longest path between two leaves in the tree. Figure 7.26 below shows two trees each with diameter nine, the leaves that form the ends of a longest path are colored (note that there may be more than one path in tree of same diameter).

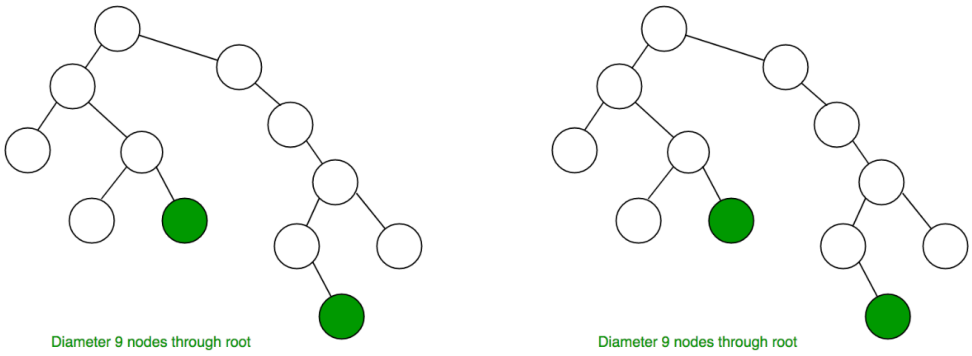


Figure 7.26: Two different cases

Computation of the diameter. The longest path will always occur between two leaf nodes. We start DFS from a random node v_1 and then see which node is farthest from it. Let the node

farthest be v_2. It is clear that v_2 will always be a leaf node and a corner of DFS. Now if we start DFS from v_2 and check the farthest node v_3 from it, we will get the diameter of the tree.

In discrete mathematics, a centered tree is a tree with only one center, and a bicentered tree is a tree with two centers.

Given a graph, the eccentricity of a vertex v is defined as the greatest distance from v to any other vertex. A center of a graph is a vertex with minimal eccentricity. A graph can have an arbitrary number of centers. However, Jordan (1869) has proved that for trees, there are only two possibilities:

- The tree has precisely one center (centered trees).
- The tree has precisely two centers (bicentered trees). In this case, the two centers are adjacent.